A Text Book on Data Structure Using C++

School of Computer Science, YCMOU, Nashik

Ch-1 Introduction to Data Structure

Objectives

By the end of this session, learners should be able to:

1. Understand the Concept of Data Structures
2. Understand Basic Terminology of Data Structures
3. Classify Different Types of Data Structures
4. Understand Real-World Applications of Data Structures

1.1 Introduction

In real life we always deal with different sets of values such as name, cost of item, roll number, Aadhar number, pin code of city, and marks in examination. These sets of values represent facts and they are called as Data. Data item refers to single set of value such as Aadhar number.

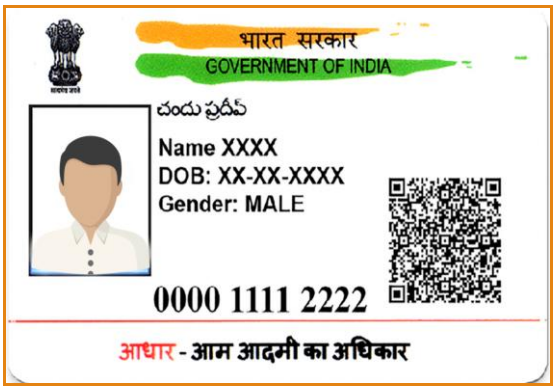


Fig1.1: Sample Aadhar Card containing Various Data Items

In this chapter we will discuss what we mean by data structure, basic terns associated with data structure, data organization, and classification of data structures.

Collection of data needs to be organized in memory so that we can make an efficient access to it. Data structure is a way to organize data in computers so that it can be accessed efficiently. Programmer need to select appropriate data structure for storing of data elements in memory. It is considered as a first step towards good programming.

1.2.1 Basic Terminology

In this section we will understand some basic terms that are associated with data items such as group items, elementary items, and entity set.

Group Items

There are different ways to organize data and accordingly there are different types of data structures. Data items that can be further divided into sub items are called as group items. For example, name can be further divided into first name, middle name and last name.

Group Item-→Name of Student: First Name + Middle Name + Last Name

Elementary Items

Data items that cannot be further subdivided are called as elementary data items. Ex: Roll number. Data items that can be further subdivided are called as group items. For Ex: Name of student.

Data with properties (attributes) is called as entity that can be assigned numeric or non numeric vale. For Ex: Attribute Age, value 23.

Entity Set

Entities with similar attributes are called as entity set. For Ex: Students of a college or employees of university are entity set.

Every attribute of entity set has a range of values, the set of all possible values. For Ex: Range of age can be 0-100

Data with given attributes is called as information, that is, processed data is called as information. For Ex: Pass or Fail students.

Activity 1 Understand data elements in a given set

Sr.No.	Data Item	Type
1	Employees	Entity Set
2	0-100	Range of Value
3	Employee ID number	Elementary Data Item
4	Address	Group Item

1.2.2 Elementary Data Organization

Data can be organized into hierarchy of fields, records and files. Field represents an attribute of entity. For Ex: Roll number, Name of student, and marks of student are different fields

Record is a collection of field values. For Ex: 1211, Prashant, 55 will be one record. File is a collection of records. For Ex. We can have a file of data of all students in a class.

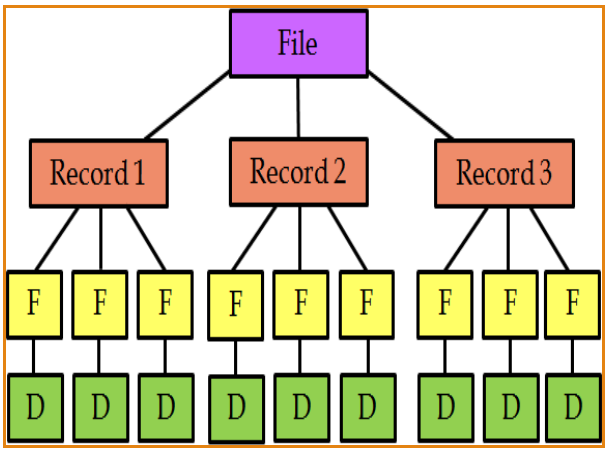


Fig 1.2: File, Records and Fields

Fig 1.2 shows structure of File, Record and Fields.

Example of file:

File: Employees File
Records: Employye1, Employee2, ……Employee100
Fields: Name, ID, Photo, Department, Address, Sarary
Data Items: Characters, Numbers, Image, Strings, ….

Primary Key

A record may contain many fields but one of them can have a unique value and it is called as primary key. For Ex: Roll number of a student can be primary key.

A file can have a fixed length records or variable length records. The minimum and maximum length of record can be specified. For Ex, we can have 4 digits for roll number, 20 characters for name and 2 digits for marks.

Activity 2: Identify if field is a primary key in following example.

Sr.No.	**Field Name**	**Primary Key (Y/N)**
1	Student Name	N

2	Roll Number	Y
3	Address	N
4	ID Number of Employee	Y
5	Salary	N

Complex Data Structures

Fields, records and files are not sufficient to process all type of data and hence we require more complex data structures. Complex data structures are arrays, linked lists, stacks, trees, queues and graphs. Good programming begins with correct choice of Data Structure. It requires less execution time and memory space. In the next sessions we will discuss about complex data structures.

1.2.3 Data Structure Classification

The logical and mathematical model used for organization of data is called as Data Structure. Data structures are classified into primitive and non primitive data structures. Figure 1.3 shows different types of data structures.

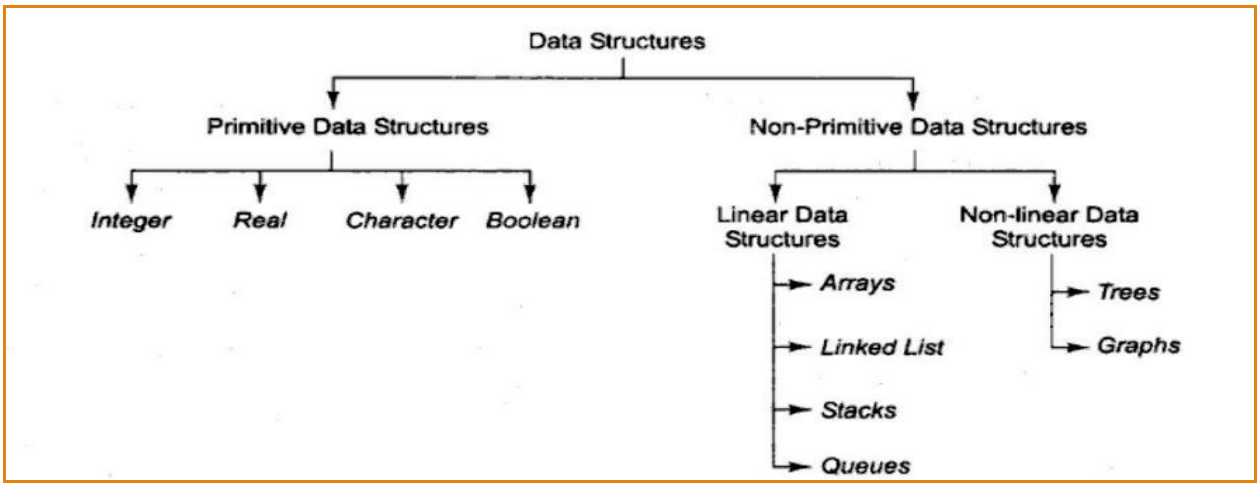


Fig: 1.3 Types of Data Structures

Primitive Data Structures

Primitive data structures cannot be further divided and they are called as simple data types. For Ex: Integer, Real, Character and Boolean.

Table 1.1 shows examples of different primitive data structures.

Sr.No.	Primitive Data Structures	Examples
1	Integer	101, 564, 77665
2	Real	2.4, 76.88, -2.5
3	Character	A, B, f, r, N, 6
4	Boolean	T, F

Non Primitive Data Structures

Non primitive data structures are classified into linear and non linear data structures. Fig 1.4 shows different non primitive data structures.

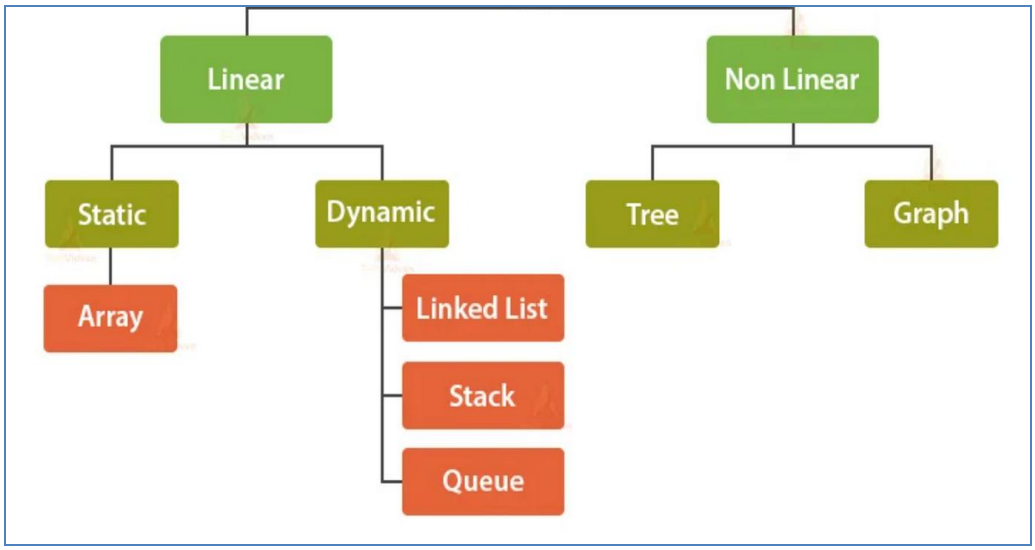


Fig 1.4: Non Primitive Data Structures

1.2.4 Linear Data Structures

Data structure is called as Linear if its elements form a sequence. For Ex: Arrays, linked list, stacks and queues are linear data structures.

Arrays

An array is a linear list of elements. It can have one or more dimensions. A subscript/index is used to refer to elements of array. For Ex: A[5] refers to 5^{th} element of array A. Fig 1.5 shows elements of array with index.

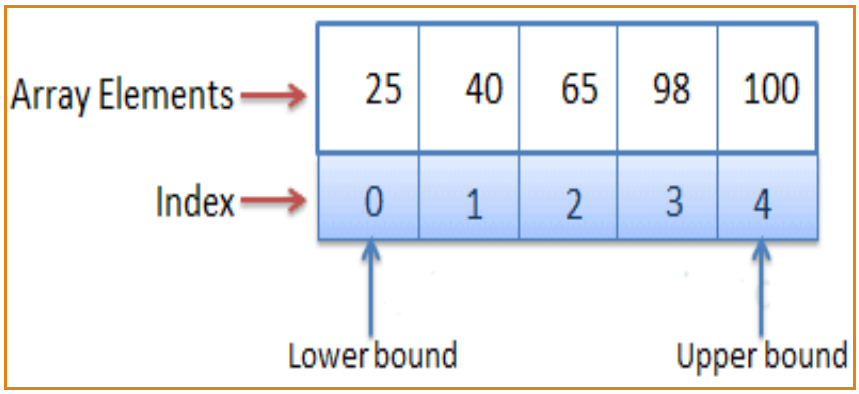


Fig 1.5: Array Elements

Observe that index can start from 0. So, first element of array is A[0]. It is also called as lower bound. Upper bound refers to last element in array.

Arrays are used to store multiple values in a single variable instead of declaring separate variables for each value. They can be used to store a wide range of data types, including integers, floating-point numbers, characters, and even complex data values.

Linked List

A linked list is a list in which each element has a link/pointer to next element. Linked lists are more efficient than arrays. Linked list is a collection nodes where every node consist of Value and pointer. Figure 1.6 shows linked list.

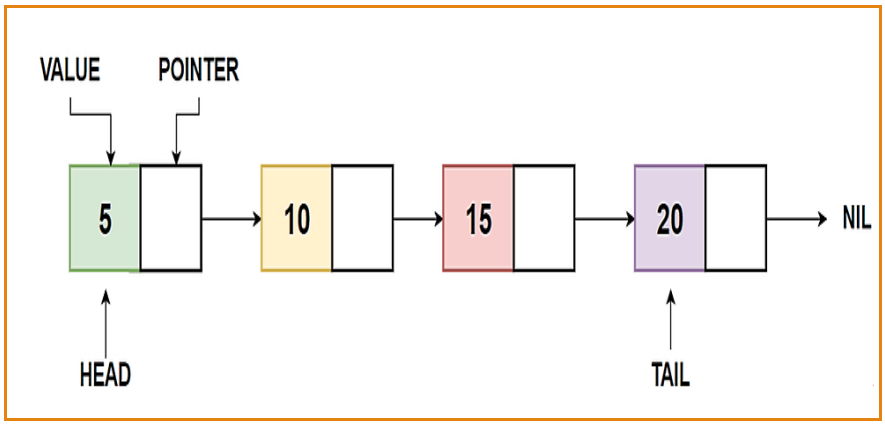


Fig 1.6: Linked List

Linked list contain following elements:

Head: It is a pointer that points to first element of list.

Pointer: It points to next element. It basically contain address of next element.

Tail: It is a last element of linked list. Its pointer value is NIL or NULL.

Linked lists use only as much memory as needed for the actual data. Arrays might have unused pre-allocated space. They do not require contiguous blocks of memory like arrays.

Linked list is used in addition of long integers, representation of metrics and Polynomial representation. They are used in implementing stack and queues and hash tables. Undo and Redo buttons in browsers makes use of linked lists. Next and previous buttons in image viewers are also implemented by using linked lists.

Stack

Stack is a linear list in which insertion and deletion takes place at one end only. It is also called as LIFO (Last In First Out) system. Fig 1.7 shows stack.

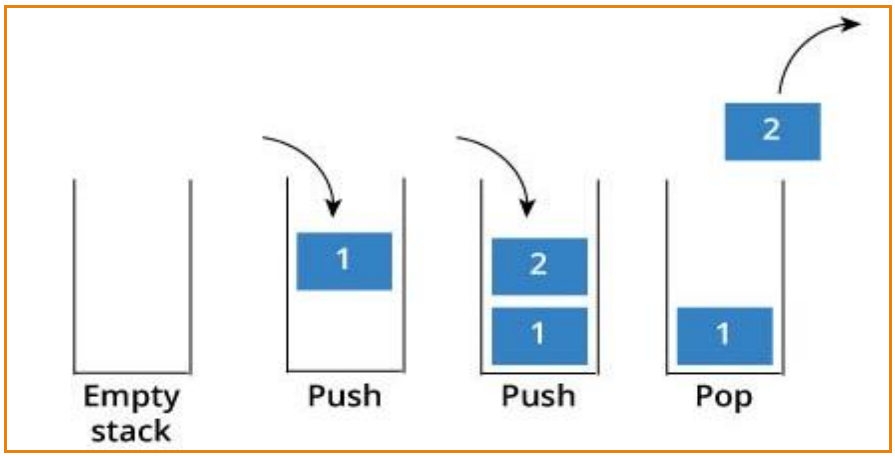


Fig 1.7: Stack

The stack is like a cylinder with one end closed and other end open. We can perform push and pop operations on stack.

Push: Add an element into stack

Pop: Remove an element from stack.

A stack can be used for evaluating expressions. An expression that consists of operands and operators. Stacks can be used for backtracking, i.e., to check parenthesis matching in an expression. It can also be used to convert one form of expression to another form. It can be used for systematic Memory Management like process scheduling in operating systems. Many Computer Science algorithms are implemented by using stack memory.

Queue

Queue is a linear list in which insertion takes place at rear end and deletion takes place at front end. It is FIFO (First In First Out) system. Fig 1.8 shows Queue data structure.

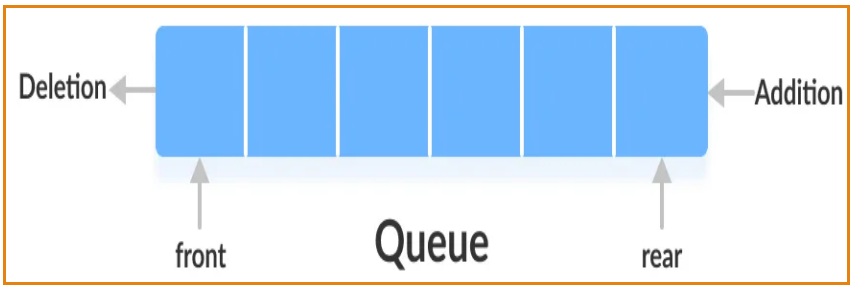


Fig 1.8: Queue

Queues are used in many applications, such as task scheduling, managing requests, handling interrupts, website traffic, networking and media players.

1.2.5 Non Linear Data Structures

If the elements of data cannot be arranged sequentially then it is called as non linear. For Ex: Trees and Graphs are non linear data structures.

Tree

Tree is a data structure that has a root and leaves. It represents hierarchical relationship between elements. Fig 1.9 shows tree data structure.

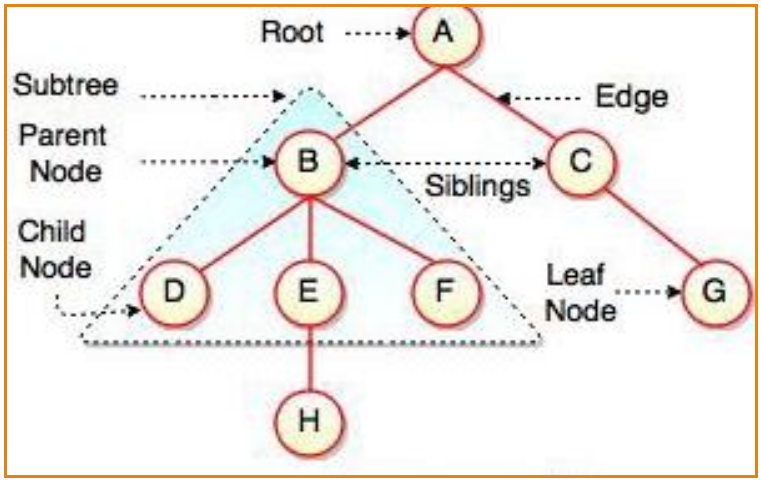


Fig 1.9: Tree

A tree consists of nodes. All nodes are connected by edge. Topmost node is called as root and nodes below it are called as Childs.

Trees are used in applications such a file systems for directory structure. It is used to organize subdirectories and files. They are used in decision making in expert systems and game playing.

Graph

Graph is a pictorial representation of elements along with links. There may not be hierarchical relationship. Fig 1.10 shows graph.

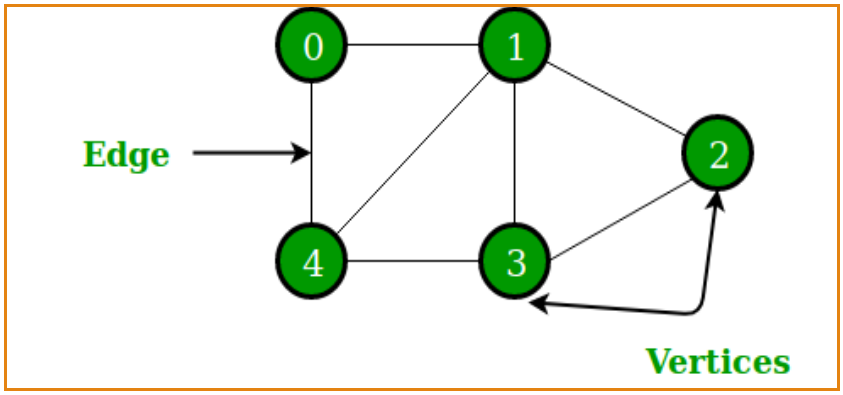


Fig 1.10: Graph

A graph consists of vertices and edges. Each vertex is named by a number. There can be two or more edges from each vertex. They are used to represent relationships and connections between objects.

Graphs are used in many applications such as Social networks, Maps and navigation, Computer networks, Biology, Computer vision, Natural language processing, Telecommunication and Circuit design.

Activity 3 Match the pair

Data Structure	**Application**
1. Graph	a. Expert System
2. Array	b. Social Networks
3. Tree	c. Expression Evaluation
4. Queue	d. Long Integer Addition
5. Linked List	e. Multiple values in single variable
6. Stack	f. Task Scheduling

(Ans: 1:b, 2:e, 3:a, 4:f, 5:d,6:c)

Questions

1. **Answer the following questions.**

   (i) What is data structure?

   (ii) State different types of data structure.

   (iii) State applications of various data structures.

   (iv) What do you mean by linear and non linear data structure?

   (v) What are primitive data structures?

   (vi) What is file, record and fields?

   (vii) With suitable example explain primary key in data.

2. **Multiple Choice Questions**

(i) What is a data structure?

Options:

A) A way to store and organize data in computers to ensure efficient access.

B) A specific programming language used to write algorithms.

C) A type of computer memory used for faster processing.

D) A collection of unrelated data stored in files.

Answer: A) A way to store and organize data in computers to ensure efficient access.

(ii) Why is selecting the appropriate data structure important?

Options:

A) It ensures better programming efficiency and organization.

B) It allows direct access to hardware memory.

C) It makes programming languages easier to learn.

D) It reduces the need for software updates.

Answer: A) It ensures better programming efficiency and organization.

(iii) What does the term "entity set" relate to in the context of data structures?

Options:

A) A group of data items with specific attributes.

B) A single variable stored in memory.

C) A specific programming syntax for accessing data.

D) A file system used to store programs.

Answer: A) A group of data items with specific attributes.

(iv) What is an example of a **group item** in data structures?

Options:

A) A single integer value.

B) A student's name divided into first name, middle name, and last name.

C) A list containing only numeric values.
D) A Boolean value (True/False).
Answer: B) A student's name divided into first name, middle name, and last name.

(v) Which statement best describes a **group item**?
Options:
A) A data item that cannot be further subdivided.
B) A data item composed of multiple sub-items.
C) A single element stored in a data structure.
D) A function used to access data in memory.
Answer: B) A data item composed of multiple sub-items.

(vi) What are data items that cannot be further subdivided called?
Options:
A) Group items
B) Elementary data items
C) Entity set
D) Complex data items
Answer: B) Elementary data items

(vii) Which of the following is an example of a group item?
Options:
A) Roll number
B) Name of a student (First Name + Middle Name + Last Name)
C) Attribute Age
D) Numeric value
Answer: B) Name of a student (First Name + Middle Name + Last Name)

(viii) What is an entity set?
Options:
A) A collection of unrelated entities
B) Entities with similar attributes
C) A single attribute assigned a value
D) Data processed into information
Answer: B) Entities with similar attributes

(ix) What does the term "range of values" refer to in an entity set?
Options:
A) The actual data assigned to an entity
B) The set of all possible values for an attribute
C) A single entity's value
D) The memory size of a data structure
Answer: B) The set of all possible values for an attribute

(x) What is processed data with given attributes called?
Options:
A) Entity
B) Information
C) Field
D) Record
Answer: B) Information

(xi) What is a primary key?

Options:

A) A unique field in a record

B) A collection of records in a file

C) A range of values assigned to an entity

D) A data structure for processing tasks

Answer: A) A unique field in a record

(xii) Which of the following is a complex data structure?

Options:

A) Integer

B) Linked list

C) Boolean

D) Character

Answer: B) Linked list

(xiii) Which of these is NOT a linear data structure?

Options:

A) Array

B) Queue

C) Tree

D) Stack

Answer: C) Tree

(xiv) Which of the following is an example of a non-linear data structure?

Options:

A) Array

B) Stack

C) Tree

D) Queue

Answer: C) Tree

(xv) Which of the following uses linked lists?

Options:

A) Backtracking in algorithms

B) Expression evaluation

C) File systems

D) Undo and redo operations in browsers

Answer: D) Undo and redo operations in browsers

(xvi) Graphs are commonly used in which of the following applications?

Options:

A) Task scheduling

B) Maps and navigation

C) Expression evaluation

D) File storage systems

Answer: B) Maps and navigation

(xvii) What are the two primary operations performed on a stack?

Options:

A) Add and delete

B) Enqueue and dequeue

C) Push and pop

D) Insert and retrieve
Answer: C) Push and pop

(xviii) In a queue, where does insertion occur?
Options:
A) Front
B) Rear
C) Middle
D) Any position
Answer: B) Rear

(xix) What does A[0] represent in an array?
Options:
A) The last element
B) The middle element
C) The first element (lower bound)
D) An invalid index
Answer: C) The first element (lower bound)

(xx) What is a record?
Options:
A) A collection of field values
B) A collection of files
C) A single data item
D) A pointer to the next file
Answer: A) A collection of field values

(xxi) Which application commonly uses trees?
Options:
A) Networking
B) Directory structures in file systems
C) Task scheduling
D) Social networks
Answer: B) Directory structures in file systems

3. State True or False

1. Data items that can be further divided into sub-items are called elementary data items.
   Answer: False

2. The name of a student, consisting of first name, middle name, and last name, is an example of a group item.
   Answer: True

3. Data with properties (attributes) is referred to as an entity.
   Answer: True

4. An entity set is a collection of entities with different attributes.
   Answer: False

5. The range of values for an attribute defines all possible values that attribute can take.
   Answer: True

6. Processed data with attributes is called information.
   Answer: True

7. A primary key is a unique field within a record that can identify it uniquely.
   Answer: True

8. Records are a collection of files, each containing multiple fields.
   Answer: False

9. Primitive data structures, such as integers and characters, cannot be further divided.
   Answer: True

10. Non-linear data structures include arrays, stacks, and queues.
    Answer: False

11. Arrays use indexes to refer to specific elements, starting with 1 as the lower bound.
    Answer: False

12. Linked lists use pointers to connect elements, allowing efficient memory usage.
    Answer: True

13. A stack is a linear data structure that uses the FIFO (First In First Out) system.
    Answer: False

14. Queues are used for task scheduling and handling requests in systems.
    Answer: True

15. Trees represent a hierarchical relationship and have a root and child nodes.
    Answer: True

16. Graphs show relationships between objects and always have a hierarchical structure.
    Answer: False

17. A file is a collection of records, and each record consists of multiple fields.
    Answer: True

18. Linked lists are not used in applications like undo and redo operations in browsers.
    Answer: False

19. Non-linear data structures are best suited for tasks requiring sequential data processing.
    Answer: False

20. A primary key ensures that each record in a file is unique.
    Answer: True

4. **Fill in the blanks**

(i) Data structure is a way to _____ data in computers so that it can be accessed efficiently.
Answer: organize

(ii) _____ items can be further divided into sub-items, whereas _____ items cannot.
Answer: Group, elementary

(iii) A data item with properties or attributes is called an _____.
Answer: entity

(iv) Entities with similar attributes are referred to as an _____ _____.
Answer: entity set

(v) The set of all possible values for an attribute is called its _____ of _____.
Answer: range, values

(vi) Data that has been processed is called _____.
Answer: information

(viii) A _____ represents an attribute of an entity, while a _____ is a collection of field values.
Answer: field, record

(ix) A _____ key is a field that has a unique value for each record in a file.
Answer: primary

(x) Arrays are _____ data structures, whereas trees and graphs are _____ data structures.
Answer: linear, non-linear

(xi) The first element of an array is referred to by the _____ bound, and the last element by the _____ bound.
Answer: lower, upper

(xii) In a linked list, the _____ points to the first element, and the _____ is the last element with a pointer value of NULL.
Answer: head, tail

(xiii) A stack operates on a _____ (Last In First Out) principle, while a queue operates on a _____ (First In First Out) principle.
Answer: LIFO, FIFO

(xiv) The two primary operations on a stack are _____, which adds an element, and _____, which removes an element.
Answer: push, pop

(xv) Trees are used in applications like _____ systems for organizing files and directories.
Answer: file

(xvi) A graph consists of _____ and _____, where vertices represent elements, and edges represent connections.
Answer: vertices, edges

Summary

Collection of data needs to be organized in memory so that we can make an efficient access to it. Data structure is a way to organize data in computers so that it can be accessed efficiently.
Various terms such as Group Items, Elementary Items are defined. Data can be structured in a hierarchy of fields, records, and files. Primitive Data Structures are Integer, Real, Character, Boolean. Non-Primitive Data Structures are further classified into linear and non-linear structures. Linear Data Structures are Arrays, Linked Lists, Stacks (LIFO) and Queues (FIFO). Non-Linear Data Structures are trees and graphs. Understanding data structures is crucial for efficient programming and problem-solving. Choosing the right data structure optimizes memory usage and execution time.

1.4 Recommended Books for Further Reading

1. "Data Structures and Algorithms in C++" – Michael T. Goodrich, Roberto Tamassia, David M. Mount
2. "Data Structures Using C++" – D. S. Malik
3. "Data Structures and Algorithm Analysis in C++" – Mark Allen Weiss

5. "Data Structures and Algorithms Made Easy in C++" – Narasimha Karumanchi
6. "Algorithms in C++" – Robert Sedgewick
7. "Programming: Principles and Practice Using C++" – Bjarne Stroustrup (Includes fundamental programming concepts along with data structures)

1.5 **References to Online Study Resources**

GeeksforGeeks - Data Structures in C++
https://www.geeksforgeeks.org/data-structures/

Udemy - Mastering Data Structures & Algorithms using C++ (Abdul Bari) *(Paid Course)*
https://www.udemy.com/course/datastructurescncpp/

Coursera - Data Structures and Algorithms Specialization (UC San Diego & HSE) *(Free/Paid Certificate)*
https://www.coursera.org/specializations/data-structures-algorithms

EdX - Data Structures and Algorithms (UC San Diego)
https://www.edx.org/course/data-structures-fundamentals-2

CS50: Introduction to Computer Science (Harvard) *(Free)*
https://cs50.harvard.edu/

Free Online Books & PDFs

Open Data Structures (in C++) – By Pat Morin *(Free PDF Book)*
https://opendatastructures.org/ods-cpp.pdf

MIT OpenCourseWare - Introduction to Algorithms *(Lecture Notes & Videos)*
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

Coding Practice Platforms

LeetCode - Data Structures Practice *(Interview Prep & Problem Solving)*
https://leetcode.com/problemset/all/

HackerRank - Data Structures Challenges *(Practice & Competitions)*
https://www.hackerrank.com/domains/tutorials/10-days-of-data-structures

CodeChef - Data Structures Certification & Problems *(Competitive Programming)*
https://www.codechef.com/certification/data-structures-and-algorithms/prepare

Ch-2 Operations on Data Structure

Objectives

By the end of this session, learners should be able to:

1. Understand Basic Data Structure Operations
2. Learn fundamental operations such as insertion, deletion, searching, updating, and traversal.
3. Understand how operations affects element indices and pointer values.

2.1 Introduction

Data stored in data structures need to be manipulated. To manipulate the data we need to use certain method. Data structure operations are the methods used to manipulate the data in a data structure. The most common data structure operations are: Insertion, Deletion, Traversing, Searching, Merging and Sorting.

In this chapter we will understand what these operations are.

2.2 Insertion Operation

Adding new element into data structure is called as insertion. Fig 2.1 shows insertion operations on various data structures. By insertion operation we can add element at start, in between or at end of the data structure.

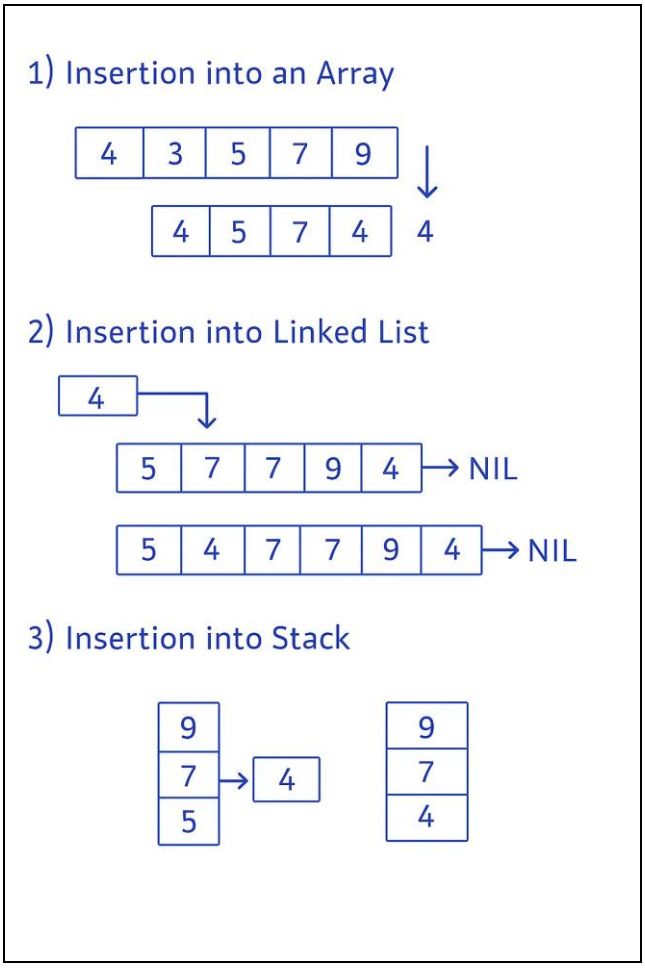


Fig 2.1: Insertion Operation

Observe that after insertion operation number of elements and their indices are changed. In linked lists the pointer values also gets changed.

2.3 Deletion Operation

Removing an element from data structure is called as deletion. Fig 2.2 shows deletion operations on various data structures. By deletion operation we remove an element anywhere in the data structure. Fig 2.2 shows deletion operation on various data structures.

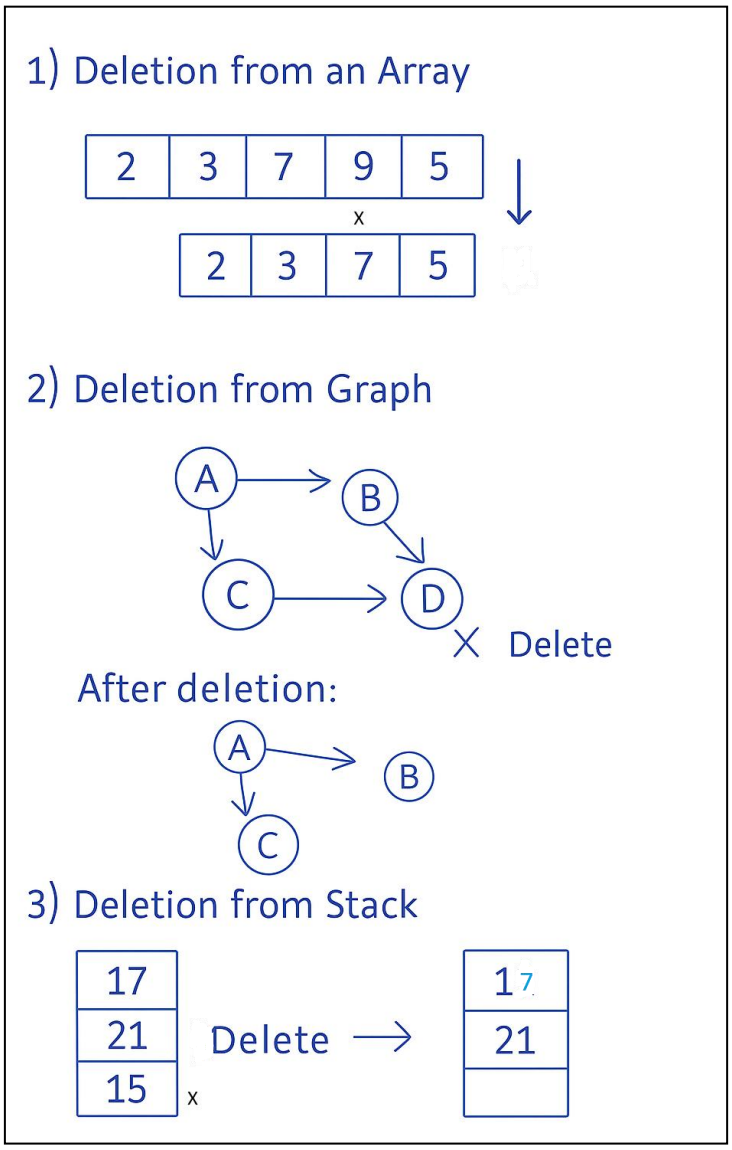


Fig 2.2: Deletion Operation

Observe that after deletion operation number of elements and their indices are changed. In graph the edges also gets deleted. In stacks, top of stack may get changed.

2.4 Traversing Operation

Traversing means accessing each element of data structure. Fig 2.3 shows traversing operation on various data structures. Observe that during traversing operation there is no change in data elements as we are just visiting each and every element and we are not making any change in it.

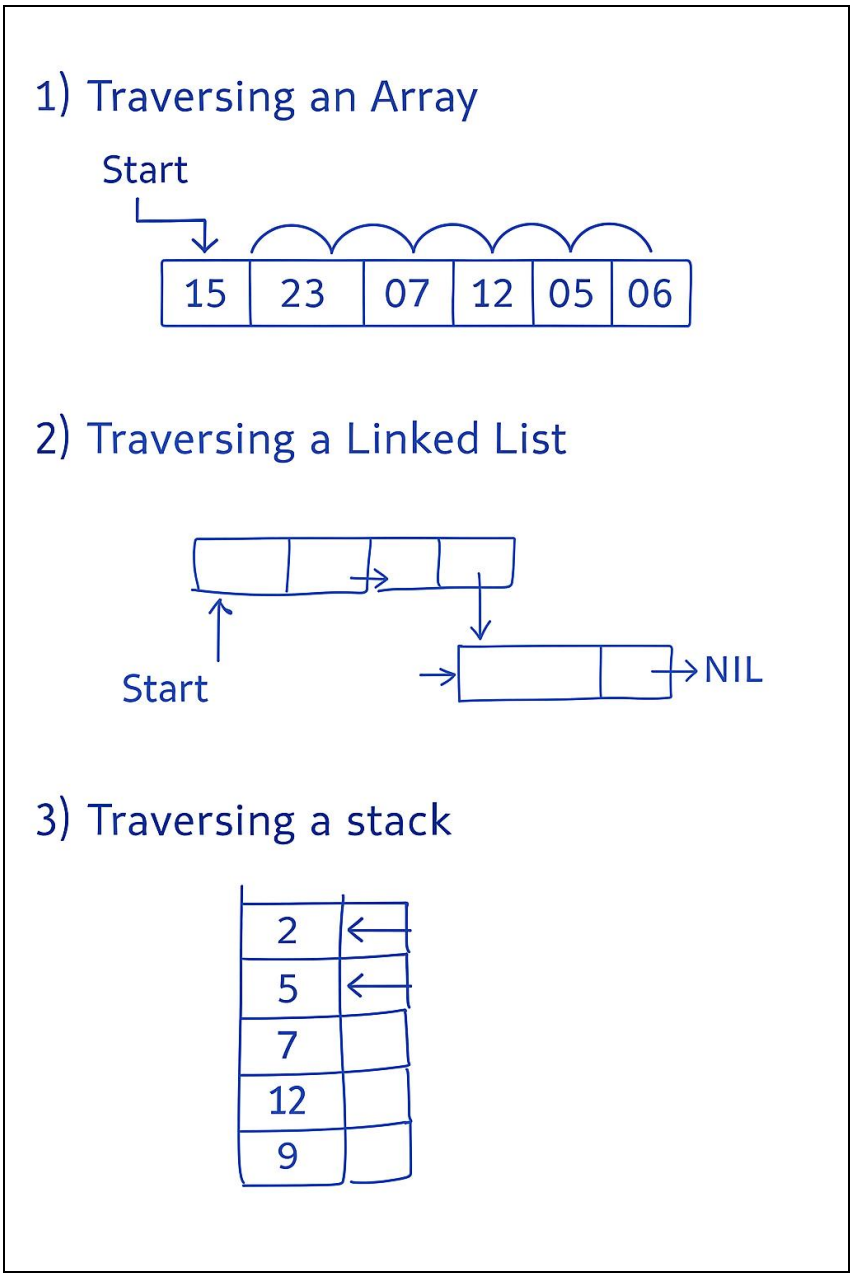


Fig 2.3: Traversing Operation

Activity 1 Understand data operation in a given set

Sr.No.	Data Operation	Effect
1	Insertion	New element is added
2	Deletion	Existing element is removed

3	Traversing	Each data element is accessed

2.5 Searching Operation

Finding an element in the data structure is called as searching. Fig 2.4 shows searching operation in an array.

In the given array we have searched element 78. The element is found at index 4. We need to search from first to last element. If search element is not found then No Match Found message will be generated.

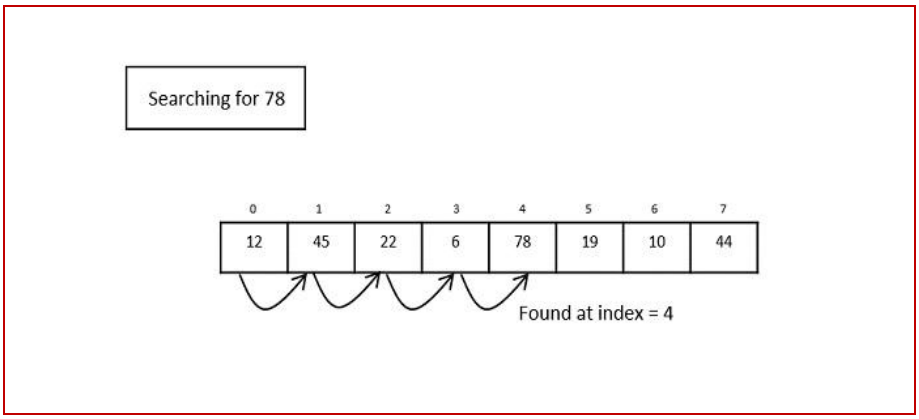


Fig 2.4: Searching in Array

2.6 Merging Operation

Combining two or more similar data structures is called as merging. For example, we can merge two arrays as shown in fig 2.5.

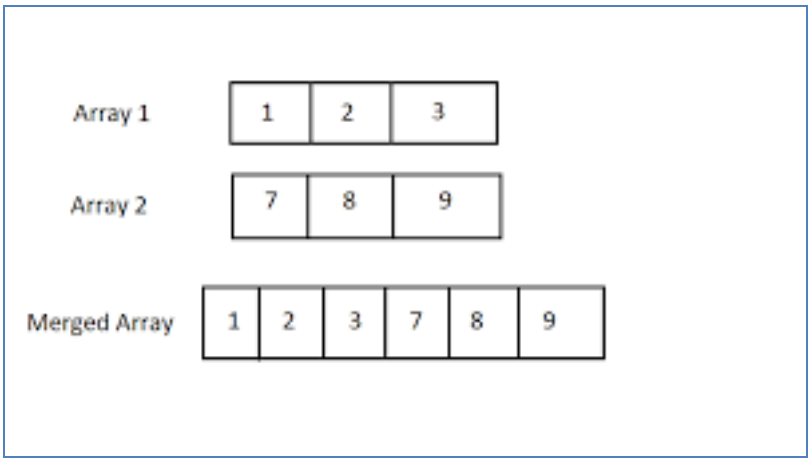


Fig 2.5: Merging of two arrays

When two arrays are merged then we will get only one array. Elements of first array will appear first and then the elements of second array will appear as shown in fig 2.5. Similarly we can merge other data structures.

Activity 2: Identify operation in following example.

Sr.No.	Field Name	Operation
1	Finding student name in list	Searching
2	Combining two class lists together	Merging
3	A new name is added	Insertion
4	A name is removed from list	Deletion

2.7 Sorting Operation

Arranging the elements of data structure in ascending or descending order is called as sorting. Fig 2.6 shows sorting of elements alphabetically. Numbers can be sorted in increasing or decreasing order as shown in fig 2.7.

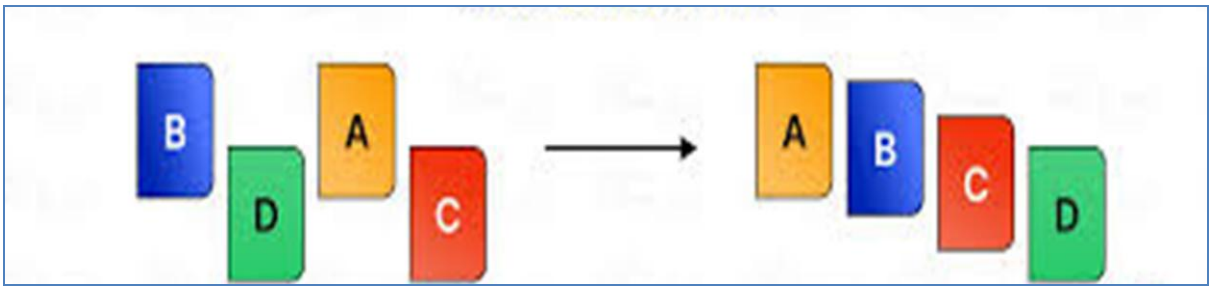


Fig 2.6: Sorting of elements

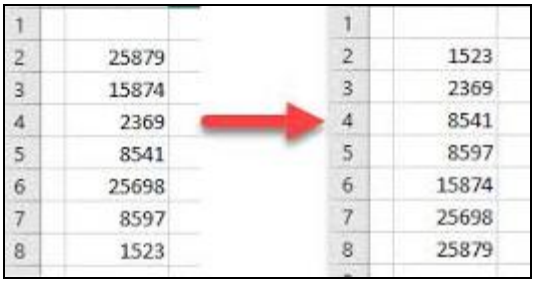


Fig 2.7: Sorting of Numbers

Activity 3 Match the pair

Data Structure Operation	Application
7. Insertion	g. Finding of key
8. Deletion	h. Addition of new element
9. Searching	i. Combination of lists
10. Traversing	j. Ascending order arrangement
11. Sorting	k. Existing element is removed
12. Merging	l. Each element is accessed

(Ans: 1:b, 2:e, 3:a, 4:f, 5:d,6:c)

Summary

Data stored in data structures must be manipulated using specific operations. These data structure operations include methods such as insertion, deletion, traversing, searching, merging, and sorting, which are essential for data management. These fundamental data structure operations help manage, manipulate, and organize data effectively, ensuring efficient storage, retrieval, and processing in various applications.

Questions

1. **Answer the following questions**

   (i) List the different operations that can be performed on data structures.

   (ii) What is insertion and deletion operation?

   (iii) What is searching and sorting operation?

   (iv) What is merging and traversing operation?

2. **Multiple Choice Questions**

   (i) What is the result of an insertion operation on a data structure?
   Options:
   A) Decrease in the number of elements
   B) No change in indices or elements
   C) Addition of a new element at a specific position
   D) Removal of an element
   Answer: C) Addition of a new element at a specific position

   (ii) What happens to the indices of elements after a deletion operation?
   Options:
   A) Indices remain unchanged

B) Indices of elements are updated
C) Indices are deleted along with the element
D) A new index is assigned to all elements
Answer: B) Indices of elements are updated

(iii) What is true about traversing a data structure?
Options:
A) It modifies the elements during the process
B) It accesses each element without making any changes
C) It always starts from the middle of the data structure
D) It deletes elements as they are accessed
Answer: B) It accesses each element without making any changes

(iv) What happens when a search element is not found in a data structure?
Options:
A) An exception is thrown
B) A "No Match Found" message is generated
C) The search operation is repeated
D) The last element is returned as the default
Answer: B) A "No Match Found" message is generated

(v) Which of the following is true about merging two arrays?
Options:
A) The arrays remain separate but are linked
B) The elements of both arrays are combined into one array
C) The elements of the second array overwrite the first array
D) The elements are sorted during the merging process
Answer: B) The elements of both arrays are combined into one array

(vi) What is the purpose of a sorting operation?
Options:
A) To add new elements to a data structure
B) To find the index of a specific element
C) To arrange elements in ascending or descending order
D) To remove duplicate elements from a data structure
Answer: C) To arrange elements in ascending or descending order

(vii) What additional change occurs during an insertion operation in a linked list?
Options:
A) The data structure remains unchanged
B) Pointer values are updated
C) Indices are added to each element
D) Elements are sorted automatically
Answer: B) Pointer values are updated

(viii) What happens to the stack after a deletion operation?
Options:
A) The last element is removed, and the top of the stack is updated
B) Elements are deleted randomly
C) The entire stack is cleared
D) Elements are rearranged in descending order
Answer: A) The last element is removed, and the top of the stack is updated

(ix) What is the main purpose of traversing in a data structure?
Options:

A) To modify data elements
B) To remove duplicate data
C) To visit and access every element
D) To arrange elements in order
Answer: C) To visit and access every element

(x) How does searching operate in an array?
Options:
A) It starts from the last element and moves backward
B) It searches for an element using binary search only
C) It searches sequentially from the first to the last element
D) It skips elements based on the search key
Answer: C) It searches sequentially from the first to the last element

(xi) Which of the following is NOT a result of a sorting operation?
Options:
A) Elements are arranged alphabetically
B) Elements are arranged in numerical order
C) Elements are combined into a single list
D) Elements can be arranged in ascending or descending order
Answer: C) Elements are combined into a single list

(xii) After merging two data structures, the result is:
Options:
A) Two separate data structures remain unchanged
B) A single data structure containing elements from both
C) Only the first data structure is retained
D) A sorted version of both data structures
Answer: B) A single data structure containing elements from both

3. State True or False

(i) Insertion operation in a data structure always occurs at the end of the structure.
Answer: False

(ii) After an insertion operation, the indices of elements in an array may change.
Answer: True

(iii) Deletion operations can occur at any position within a data structure.
Answer: True

(iv) Traversing a data structure modifies the data elements.
Answer: False

(v) During traversing, each element of the data structure is accessed without making any changes.
Answer: True

(vi) Searching in an array requires checking every element sequentially from the start to the end.
Answer: True

(vii) If a searched element is not found, a "Match Found" message is displayed.
Answer: False

(viii) Merging involves combining two or more similar data structures into one.
Answer: True

(ix) After merging two arrays, elements of both arrays retain their original order within the new array.
Answer: True

(x) Sorting arranges elements of a data structure in either ascending or descending order.
Answer: True

(xi) Sorting is used to merge two data structures into one.
Answer: False

(xii) Insertion in linked lists changes pointer values in addition to adding a new element.
Answer: True

(xiii) Deletion in a stack does not affect the top of the stack.
Answer: False

(xiv) Traversing a data structure is required to modify data elements.
Answer: False

(xv) A searching operation can generate a "No Match Found" message if the element is absent.
Answer: True

(xvi) Merging two arrays results in a single array containing elements from both arrays.
Answer: True

(xvii) Sorting is limited to numeric data and cannot be performed on alphabetic data.
Answer: False

(xviii) Insertion and deletion operations do not alter the number of elements in a data structure.
Answer: False

(xix) Merging and sorting are independent operations performed on data structures.
Answer: True

(xx) Sorting always places the smallest element at the beginning, regardless of order preference.
Answer: False

4. Fill in the blanks

(i) Adding a new element into a data structure is called _____.
Answer: insertion

(ii) After an insertion operation, the _____ of elements in an array may change.
Answer: indices

(iii) Removing an element from a data structure is called _____.
Answer: deletion

(iv) During a deletion operation in a linked list, the _____ values are updated.
Answer: pointer

(v) Accessing each element of a data structure without making any changes is called _____.
Answer: traversing

(vi) Finding a specific element in a data structure is referred to as _____.
Answer: searching

(vii) When a searched element is not found in an array, a _____ message is generated.
Answer: "No Match Found"

(viii) Combining two or more similar data structures is called _____.
Answer: merging

(ix) After merging two arrays, the resulting data structure contains elements from both arrays in the _____ order they appeared.
Answer: original

(x) Arranging elements in ascending or descending order is called _____.
Answer: sorting

(xi) Insertion in a linked list updates the _____ values along with adding a new element.
Answer: pointer

(xii) In a stack, the deletion operation updates the _____ of the stack.
Answer: top

(xiii) During a traversing operation, there is _____ change in the data elements.
Answer: no

(xiv) A _____ operation is required to locate an element within a data structure.
Answer: searching

(xv) Sorting can arrange elements _____ or _____ based on the requirement.
Answer: ascending, descending

(xvi) Merging two arrays results in a _____ array containing elements from both.
Answer: single

(xvii) Sorting is used to arrange data either numerically or _____.
Answer: alphabetically

(xviii) Insertion and deletion operations affect the _____ of elements in a data structure.
Answer: number

(xix) Merging combines data structures, whereas _____ organizes the elements within a single structure.
Answer: sorting

(xx) A _____ operation ensures that each element of a data structure is visited.
Answer: traversing

Recommended Books for Further Reading

4. "Data Structures and Algorithms in C++" – Michael T. Goodrich, Roberto Tamassia, David M. Mount
5. "Data Structures Using C++" – D. S. Malik
6. "Data Structures and Algorithm Analysis in C++" – Mark Allen Weiss

8. "Data Structures and Algorithms Made Easy in C++" – Narasimha Karumanchi
9. "Algorithms in C++" – Robert Sedgewick
10. "Programming: Principles and Practice Using C++" – Bjarne Stroustrup (Includes fundamental programming concepts along with data structures)

References to Online Study Resources

GeeksforGeeks - Data Structures in C++
https://www.geeksforgeeks.org/data-structures/

Udemy - Mastering Data Structures & Algorithms using C++ (Abdul Bari) *(Paid Course)*
https://www.udemy.com/course/datastructurescncpp/

Coursera - Data Structures and Algorithms Specialization (UC San Diego & HSE) *(Free/Paid Certificate)*
https://www.coursera.org/specializations/data-structures-algorithms

EdX - Data Structures and Algorithms (UC San Diego)
https://www.edx.org/course/data-structures-fundamentals-2

CS50: Introduction to Computer Science (Harvard) *(Free)*
https://cs50.harvard.edu/

Free Online Books & PDFs

Open Data Structures (in C++) – By Pat Morin *(Free PDF Book)*
https://opendatastructures.org/ods-cpp.pdf

MIT OpenCourseWare - Introduction to Algorithms *(Lecture Notes & Videos)*
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

Coding Practice Platforms

LeetCode - Data Structures Practice *(Interview Prep & Problem Solving)*
https://leetcode.com/problemset/all/

HackerRank - Data Structures Challenges *(Practice & Competitions)*
https://www.hackerrank.com/domains/tutorials/10-days-of-data-structures

CodeChef - Data Structures Certification & Problems *(Competitive Programming)*
https://www.codechef.com/certification/data-structures-and-algorithms/prepare

A Text Book on Data Structure Using C++

School of Computer Science, YCMOU, Nashik

Ch-3 Design of an Algorithm

Objectives

By the end of this session, learners should be able to:

1. Understand the concept and significance of algorithm design strategies.
2. Explain the Top-Down approach and how it emphasizes breaking a problem into smaller sub problems.
3. Illustrate the Bottom-Up approach, where smaller sub problems are solved first and combined to form a solution.
4. Define complexity and its importance in evaluating algorithm performance.
5. Differentiate between Time Complexity and Space Complexity in algorithm analysis.

1.1What is an Algorithm?

Let us consider an example to post a letter. The sequence of steps for this would be:

- Step 1. Writing the letter
- Step 2. Folding the letter
- Step 3. Inserting the letter in an envelope

- Step 4. Writing the address
- Step 5. Sticking the stamp
- Step 6. Posting the letter

Step (1), (2) and (3) in the above process should be followed in the given order. Step (4) and (5) can be interchanged. Above process can be called as an algorithm for posting a letter.

Algorithm is a set of steps. It is written to perform certain operation.

1.2Need of Algorithm

Following points illustrate us the need of algorithm.

1. Writing an algorithm is mostly considered as a first step to programming.

2. It is easy to code the program from algorithm in high level programming language.

3. If the algorithm is correct, computer will run the program correctly, every time.

4. The purpose of using an algorithm is to increase the reliability, accuracy and efficiency of obtaining solutions.

3.3 Features of Algorithm

Following are the features associated with an algorithm.

1. Input: Must have zero or more inputs.
2. Output: Must have one or more outputs.
3. Finiteness: The algorithm must terminate after a finite number of steps.
4. Definiteness: Each step must be unambiguously defined.
5. Effectiveness: All operations must be sufficiently basic.
6. Recursion: Algorithms can be iterative and recursive.
7. Effectiveness: All operations must be sufficiently basic.

   Algorithms can be iterative. That is, certain steps of algorithm are repeatedly executed. Algorithms are recursive as well. That is, we can call certain action within action itself.

Activity 1 Understand algorithm in a given set

Sr.No.	**Algorithm Feature**	**Name of Feature**
1	Clear definition of each step	Definiteness
2	Termination after certain steps	Finiteness
3	Call within itself	Recursion

3.4 Design of Algorithm

There are two approaches to design an algorithm:

- Top Down approach
- Bottom Up approach

Let us discuss both of these approaches.

3.4.1 Top Down Approach

In Top Down approach the problem is divided into smaller sub problems. This approach is also called as Divide and Conquer. In this approach the main task is divided into smaller subtasks as shown in fig. 3.1.

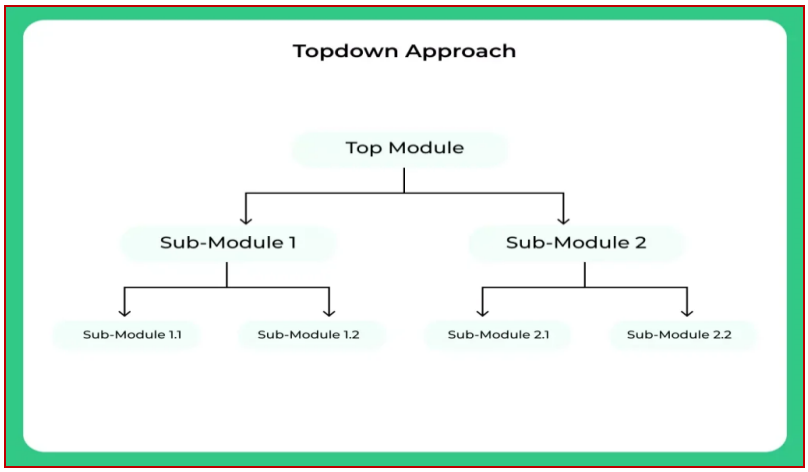


Fig 3.1: Top down Approach

Steps:

1. Identify the problem.
2. Divide it into smaller sub problems.
3. Obtain solution for sub problems.
4. Combine solutions together to get solution of problem.
5. Examples: Searching in a list, Sorting of list, Pay calculation in organization.

3.4.2 Advantages of Top Down Approach

Following are advantages of top down approach. Fig 3.2 shows advantages of top down approach.

1. Easier to Understand: When complex task is sub divided into smaller sub tasks then steps of algorithm becomes easy to understand.

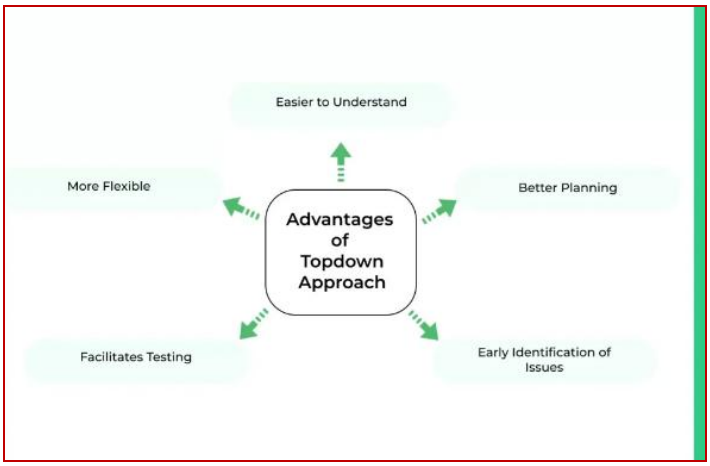


Fig 3.2: Advantages of Top Down Approach

2. Better Planning: With simple steps we can achieve our goal with considerations of obstacles and resources.
3. Early Identification of Issues: If there are any chances of occurrence of problem in obtaining a solution then that is identified at early stage.
4. Facilitates Testing: Test data can be run to check correctness at each level .
5. More Flexible: Change or modification is possible if the conditions are changed.

3.4.3 Bottom Up Approach

In Bottom Up approach we start from simplest sub problem and build the solution to larger problem. This approach is also called as human approach because most of real life problems are solved by humans by attempting simple parts first.

Steps:

1. Identify the components of system
2. Obtain solution to simplest component
3. Combine solutions to solve larger problem
4. Examples: fibonacci sequence, Product design, and program compilation.

Activity 2: Identify algorithm approach in following example.

Sr.No.	Example	Approach
1	Obtain solution to simplest component	Bottom Up
2	Divide it into smaller sub problems	Top Down
3	Fibonacci Sequence	Bottom Up
4	Sorting of list	Top Down

3.4 Complexity of Algorithms

Suppose you want to go from city A to city B as shown in fig 3.3. . You will find that there are multiple ways to travel from city A to B. You will select the way that is best suitable for you.

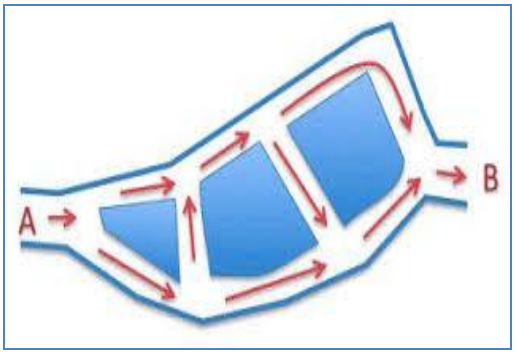


Fig 3.3: City A to B travel

In Computer Science multiple algorithms are available for same problem. Algorithms are compared by using complexity. Every algorithm requires certain resources for it's execution.

Resources can be in the form of memory space and time. Measure of resources for any algorithm is called as complexity of algorithm. An algorithm with less complexity is always better than that of high complexity.

3.4.1 How to Compare Algorithms?

To compare algorithms, there are a few objective measures:

1. Execution times: Not a good measure as execution times are specific to a particular computer
2. Number of statements executed: Not a good measure, since the number of statements varies with the programming language as well as the style of the individual programmer

Ideal solution is:

Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, and programming style.

3.4.2 Rate of Growth

The rate at which the running time increases as a function of input is called rate of growth. Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say buying a car.

This is because the cost of the car is high compared to the cost of the bicycle.

Total Cost=Cost (Car) + Cost (Bicycle) = Cost (Car)

As an example, consider a function: $f = n^4+2n^2+100n+500$.

In above function n^4, $2n^2$, 100n and 500 are the individual costs of some function and approximate to n^4 since n^4 is the highest rate of growth.

3.4.3 Commonly Used Rate of Growth

Some commonly used rate of growth is given in following table 3.1. Here n stands for number of inputs.

. **Table 3.1 Commonly Used Rate of Growth**

Rate of Growth	Name
1	Constant
log(n)	Logarithmic
n	Linear
nlog(n)	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Observe that in above table rate of growth is lowest for constant 1 and highest for exponential 2^n.

3.5 Time Complexity

Every statement in algorithm requires certain amount of time for it's execution. The time requirement of complete execution of algorithm is called time complexity. If some statements are repeatedly executed then time is also taken that many numbers of times. It helps us to understand how algorithm performs. Table 3.2 shows time complexity for input n from constant to factorial.

Table 3.2 Time Complexity

Time Complexity	Name
1	Constant
log(n)	Logarithmic
n	Linear
nlog(n)	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Observe that the best case is 1 and worst case is n!

3.6 Space Complexity

Space complexity refers to memory space requirement of algorithm. It's a function of inputs of algorithm. An algorithm with less space requirement is always a better option. Fig 3.3 shows space complexity for input n. Space requirement is less for 1 and high for 2^n. Observe that big O is notation that is used to specify complexity. We will discuss about notations in next chapter.

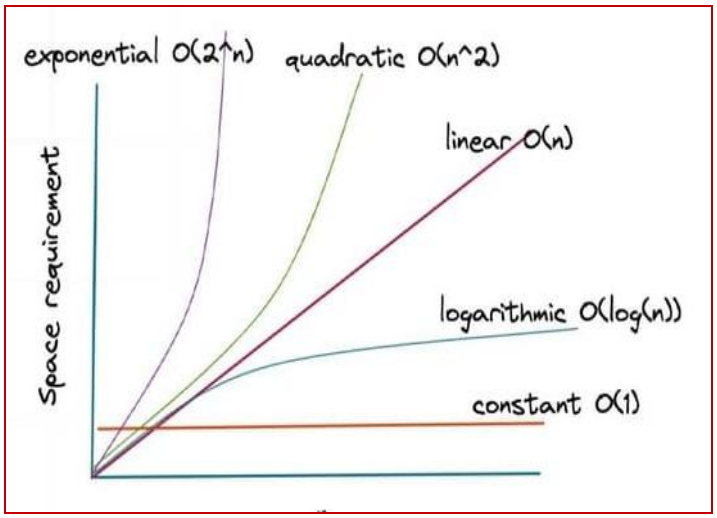


Fig 3.3: Space Complexity

Activity 3 Match the pair

Complexity	**Meaning**
13. 1	m. Logarithmic
14. Space Complexity	n. Constant
15. Log n	o. Factorial
16. Time Complexity	p. Linear
17. O(n)	q. Memory Space Requirement
18. n!	r. Computer Time Requirement

(Ans: 1:b, 2:e, 3:a, 4:f, 5:d,6:c)

Summary

An algorithm is a step-by-step procedure designed to solve a specific problem, much like the process of posting a letter, where steps must be followed in a logical sequence. Algorithms are essential in programming because they simplify coding, ensure accuracy, and improve efficiency. They have key features such as input, output, finiteness, definiteness, effectiveness, and can be iterative or recursive. Algorithms can be designed using two main approaches: Top-Down, which divides a problem into smaller subproblems and Bottom-Up, which starts from solving simpler components to build larger solutions. To evaluate algorithms, their complexity is measured in terms of time (execution steps) and space (memory usage). Complexity is expressed using growth rates like constant, logarithmic, linear,

quadratic, cubic, exponential, and factorial. Time complexity reflects the execution time based on input size, while space complexity refers to the memory required. Efficient algorithms aim for low time and space complexity, ensuring reliable, accurate, and optimal problem-solving.

Questions

1. **Answer the following questions**

   (i) What is an algorithm?

   (ii) What is need of algorithm?

   (iii) What is complexity of algorithm?

   (iv) What is top down and bottom up approach?

   (v) How algorithms are compared?

   (vi) Explain time complexity of algorithms.

   (vii) Explain space complexity of algorithms.

   (viii) What are commonly used rate of growth?

2. **Multiple Choice Questions**

   (i) What is an algorithm?

   Options: a) A programming language used to write code
   b) A set of steps to perform a specific task
   c) A high-level programming language
   d) A tool to debug programs
   Answer: b) A set of steps to perform a specific task

   (ii) Which of the following illustrates the need for algorithms?

   Options: a) To make programming easier
   b) To increase the reliability, accuracy, and efficiency of solutions
   c) To write programs in low-level languages
   d) To avoid using programming languages
   Answer: b) To increase the reliability, accuracy, and efficiency of solutions

   (iii) Which of the following is NOT a feature of an algorithm?

   Options: (a) Must have one or more outputs
   b) Must terminate after an infinite number of steps
   c) Each step must be unambiguously defined

d) Must have zero or more inputs
Answer: b) Must terminate after an infinite number of steps

(iv) The top-down approach in algorithm design is also called:

Options: (a) Human approach
b) Divide and conquer
c) Linear approach
d) Recursive approach
Answer: b) Divide and conquer

(v) Which of the following is an advantage of the top-down approach?

Options: (a) It requires less memory space
b) It facilitates testing at each level
c) It avoids the use of recursion
d) It requires complex steps to understand
Answer: b) It facilitates testing at each level

(vi) In the bottom-up approach, we start solving the problem by:

Options: (a) Breaking it into smaller sub-problems
b) Combining smaller solutions to solve the larger problem
c) Identifying all inputs at once
d) Using a single function for all steps
Answer: b) Combining smaller solutions to solve the larger problem

(vii) What is the measure of resources required by an algorithm called?

Options: a) Resource utilization
b) Complexity
c) Functionality
d) Accuracy
Answer: b) Complexity

(viii) Which of the following is used to compare algorithms in an ideal way?

Options: a) Execution time on a specific computer
b) Number of statements executed
c) Rate of growth as a function of input size
d) Style of programming
Answer: c) Rate of growth as a function of input size

(ix) Time complexity of an algorithm refers to:

Options: a) Memory usage during execution
b) Execution time required for each statement

c) Total time required for complete execution
d) Amount of hardware used during execution
Answer: c) Total time required for complete execution

(x) Space complexity refers to:

Options: a) The execution time of an algorithm
b) The memory space required by an algorithm
c) The resources required for debugging
d) The processing power of the system
Answer: b) The memory space required by an algorithm

3. State True or False

1. An algorithm is a set of steps written to perform a specific operation.

Answer: True

2. Writing an algorithm is not necessary for coding a program.

Answer: False

3. An algorithm must always have at least one input.

Answer: False

4. The top-down approach is also known as the "human approach."

Answer: False

5. Bottom-up approaches solve smaller sub-problems first and combine them to solve the larger problem.

Answer: True

6. Complexity of an algorithm measures the resources it requires, such as time and memory.

Answer: True

7. Execution time on a specific computer is the ideal way to compare algorithms.

Answer: False

8. The rate of growth is the slowest for exponential algorithms.

Answer: False

9. Time complexity is the total time required for the complete execution of an algorithm.

Answer: True

10. Space complexity measures the memory space requirement of an algorithm.

Answer: True

11. Algorithms can be iterative or recursive, or both.

Answer: True

12. The top-down approach is also known as "divide and conquer."

Answer: True

4. Fill in the blanks

(i) An _____ is a set of steps written to perform a specific operation.

Answer: Algorithm

(ii) Writing an algorithm is considered as the _____ step to programming.

Answer: First

(iii). The _____ approach divides a problem into smaller sub-problems and solves each one individually.

Answer: Top-down

(iv) The _____ approach starts from the simplest sub-problem and builds the solution to the larger problem.

Answer: Bottom-up

(v) One of the key features of an algorithm is that it must have zero or more _____.

Answer: Inputs

(vi) An algorithm must have one or more _____ as outputs.

Answer: Outputs

(vii) An algorithm should terminate after a finite number of _____.

Answer: Steps

(viii) _____ refers to the rate at which the running time increases as a function of input size.

Answer: Rate of growth

(ix) The running time of an algorithm can be expressed as a function of _____.

Answer: Input size (n)

(x) _____ is a measure of the time required for the complete execution of an algorithm.

Answer: Time complexity

(xi) _____ refers to the memory space requirement of an algorithm.

Answer: Space complexity

(xii) The time complexity for an algorithm is considered _____ when the time required does not depend on the input size.

Answer: Constant

(xiii) The best case time complexity is denoted by _____.

Answer: 1 (Constant)

(xiv) The _____ notation is used to specify the complexity of an algorithm.

Answer: Big O

A Text Book on Data Structure Using C++

School of Computer Science, YCMOU, Nashik

Ch-4 Algorithmic Notations

Objectives

By the end of this session, learners should be able to:

1. Define complexity and its importance in evaluating algorithm performance.
2. Differentiate between Time Complexity and Space Complexity in algorithm analysis.
3. Analyze Time Complexity by using Big-O notation to measure the efficiency of an algorithm.
4. Evaluate Space Complexity to determine the memory requirements of an algorithm.
5. Compare different algorithms based on their time and space complexities to determine optimal solutions.

In this chapter we will understand the different types of notations that are used for describing complexity of algorithms. First of all we will understand different types of analysis.

4.1 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time (performing worst).

An algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: One for the case where it takes less time (Best Case) and another for the case where it takes more time (Worst Case).

Worst case defines the input for which the algorithm takes a long time. Best case defines the input for which the algorithm takes the least time. Average case provides a prediction about the running time of the algorithm.

4.2 Asymptotic Notations

There are three asymptotic notations that are used for analysis of algorithms. Asymptotic means approaching a given value or condition.

Big O Notation- Upper bound time complexity or worst case time complexity is given by this notation.

Omega (Ω) Notation- Here **t**he execution time serves as a lower bound on the algorithm's time complexity.

Theta (Θ) Notation- It represents the upper and the lower bound of the running time of an algorithm. It is used for analyzing the average-case complexity of an algorithm.

4.3 Big O Notation

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm. It is the most widely used notation for asymptotic analysis. It specifies the upper bound of a function.

This notation gives the tight upper bound of the given function. Generally, it is represented as f(n) = O(g(n))That means, at larger values of n, the upper bound of f(n) is g(n).

O–notation defined as O(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n > n_0$}. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give the smallest rate of growth g(n) which is greater than or equal to the given algorithms' rate of growth f(n).

For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, Then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.

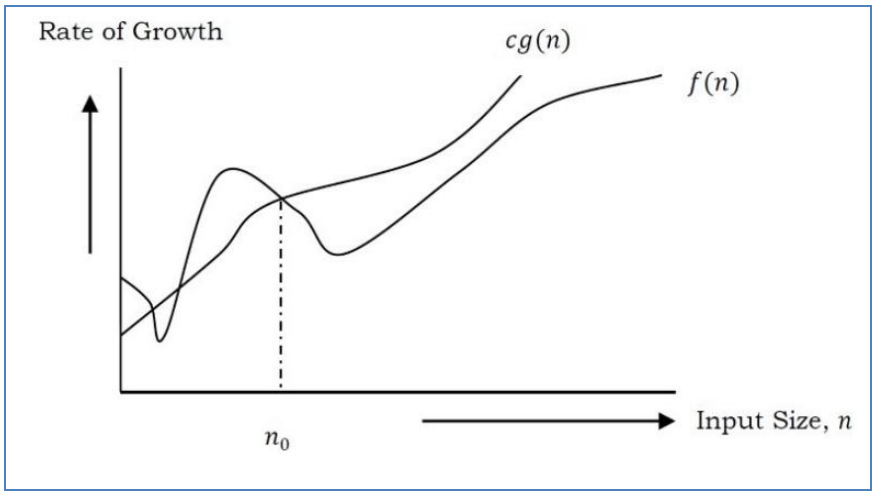


Fig 4.1 Function f(n) and cg(n)

Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0 , the rate of growth could be different. n0 is called threshold for the given function.

The maximum time required by an algorithm or the worst-case time complexity. It returns the highest possible output value (big-O) for a given input. Big-O (Worst Case) is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

Example:

Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1 \therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$.

Activity 1: Identifying Best, Worst, and Average Cases

Given the following algorithms, identify the best, worst, and average cases for each:

Sr. No.	Algorithm	Best Case	Worst Case	Average Case
1	Linear Search	O(1)	O(n)	O(n/2)
2	Binary Search	O(1)	O(log n)	O(log n)

Activity 2: Matching Complexity Expressions

Function	Big O Notation
a) $f(n) = 5n + 20$	(i) $O(n^2)$
b) $f(n) = 2n^3 + 3n + 7$	(ii) O(n)
c) $f(n) = \log(n) + 10$	(iii) $O(n^3)$
d) $f(n) = n^2 + 4n + 1$	(iv) O(log n)

(Ans: a:ii, b:iii, c:iv, d:i)

Activity 3: Graphing Growth Rates

Plot the growth of the following functions for n = 1 to 10:

O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(2^n)$.

Observe that exponential functions grow significantly faster than polynomial functions.

4.4 Omega Notation

Omega (Ω) notation represents the lower bound of the running time of an algorithm. Therefore, it gives the best-case complexity of an algorithm. Similar to the O, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$.

That means, at larger values of n, the tighter lower bound of f(n) is g(n).

For example, if $f(n) = 100n^2 + 10n + 50$, g(n) is $\Omega(n^2)$.

It specifies the lower bound of a function. The minimum time required by an algorithm or the best-case time complexity. It returns the lowest possible output value (Omega) for a given input.

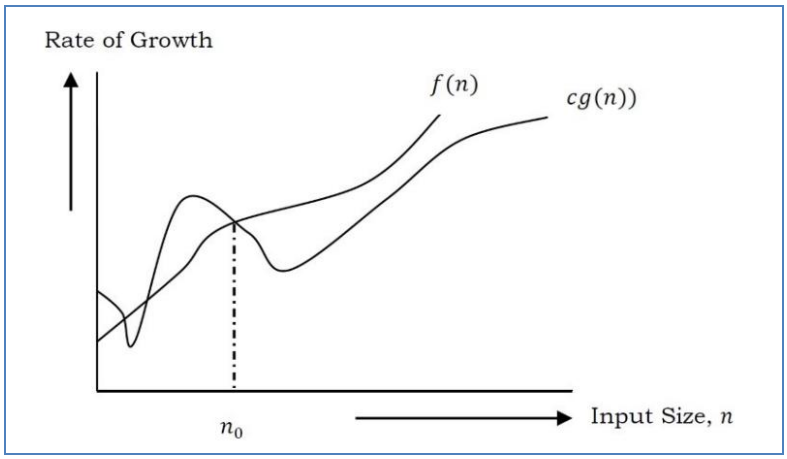


Fig 4.2 Function f(n) and cg(n)

Omega(Best Case) It is defined as the condition that allows an algorithm to complete statement execution in the lowest amount of time possible.

Example: Find lower bound for $f(n) = 5n^2$.

Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 5$ and $n_0 = 1 \therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$.

4.5 Theta (Θ) Notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth.

As an example, let us assume that f(n) = 10n + n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in the best case is g(n) = O(n)

For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same. In this case, we need to consider all possible time complexities and take the average of those.

It is defined as Θ(g(n)) = {f(n): there exist positive constants c1 ,c2 and n0 such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$}. g(n) is an asymptotic tight bound for f(n). Θ(g(n)) is the set of functions with the same order of growth as g(n).

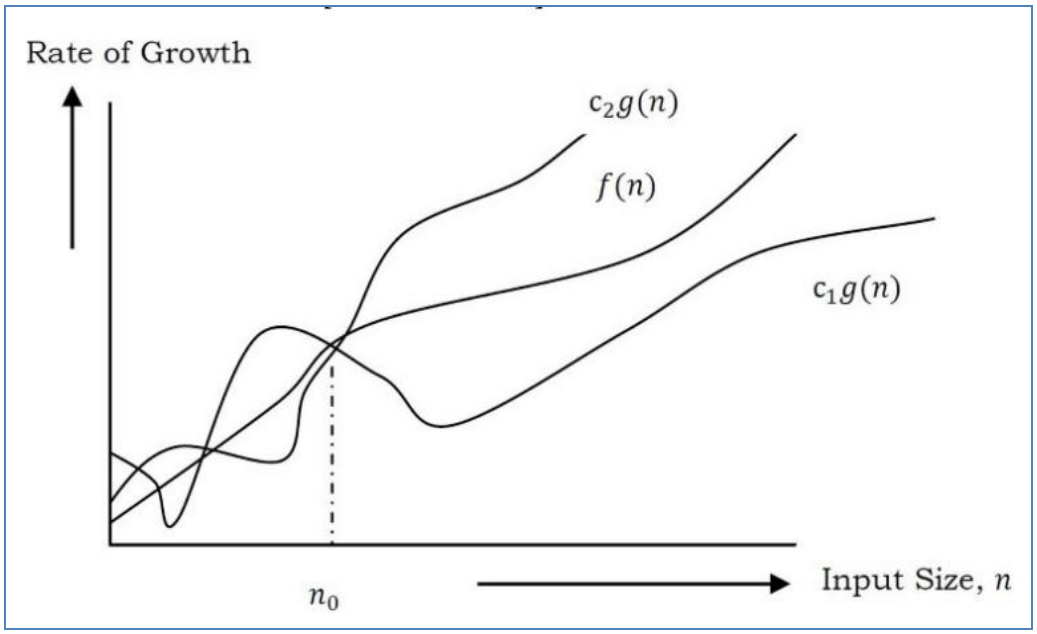


Fig 4.3: Function f(n) and g(n)

Example:

Example 1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$ for all, $n \geq 2$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 2$

Summary

The analysis of algorithms helps determine how an algorithm performs with different inputs, focusing on best case (least time), worst case (longest time), and average case (expected time). To represent these, three asymptotic notations are used: Big O for the upper bound (worst-case complexity), Omega (Ω) for the lower bound (best-case complexity), and Theta (Θ) for the average-case complexity, which lies between the two bounds. Big O is most widely used as it shows the maximum growth rate of an algorithm, Omega highlights the minimum required time, and Theta confirms if both bounds coincide, giving a tight bound for the algorithm's growth. These notations allow comparison of algorithms independent of machine or programming style by focusing on the rate of growth of functions, making them essential for understanding efficiency and scalability.

Questions

1. **Answer the following questions.**

   (i) What are the different types of analysis of algorithms?

   (ii) Write three asymptotic notations used for describing complexity of algorithms.

   (iii) Describe Big O notation.

   (iv) Define and describe omega notation.

   (v) Define and describe theta notation.

2. **Multiple Choice Questions**

1. What does Big O notation represent in terms of algorithm analysis?

Options: a) The average-case time complexity
b) The lower bound of the algorithm's time complexity
c) The upper bound of the algorithm's time complexity
d) The exact time complexity of the algorithm

Answer: c) The upper bound of the algorithm's time complexity

2. Omega (Ω) notation represents which aspect of an algorithm's time complexity?

Options: a) Best-case time complexity
b) Worst-case time complexity
c) Average-case time complexity
d) Both worst-case and best-case time complexity

Answer: a) Best-case time complexity

3. What is the purpose of Theta (Θ) notation in algorithm analysis?

Options: a) It represents the worst-case time complexity
b) It represents both the upper and lower bounds of the running time
c) It is used for comparing two different algorithms
d) It is used only for worst-case analysis

Answer: b) It represents both the upper and lower bounds of the running time

4. The "Big O" notation is most commonly used to represent which of the following?

Options: a) Best-case complexity
b) Worst-case complexity
c) Average-case complexity
d) None of the above

Answer: b) Worst-case complexity

5. Which of the following describes Omega (Ω) notation?

Options: a) It provides an upper bound for an algorithm's running time.
b) It provides a lower bound for an algorithm's running time.
c) It provides both upper and lower bounds for an algorithm's running time.
d) It provides the average-case time complexity.

Answer: b) It provides a lower bound for an algorithm's running time.

6. In the Big O notation, what is the rate of growth for the expression f(n) = n^4 + 100n^2 + 10n + 50 at larger values of n?

Options: a) n^4
b) n^2
c) 100n^2
d) 50

Answer: a) n^4

7. What is the threshold value (n_0) used for in Big O notation?

Options: a) It indicates when we start ignoring lower growth rates in the algorithm.
b) It represents the number of inputs required for an algorithm to start performing efficiently.
c) It provides the worst-case input size for the algorithm.
d) It is used to calculate the exact running time of an algorithm.

Answer: a) It indicates when we start ignoring lower growth rates in the algorithm.

8. In the expression f(n) = 5n^2, the lower bound given by Omega (Ω) notation is:

Options: a) Ω(n^3)
b) Ω(n^2)
c) Ω(n)
d) Ω(1)

Answer: b) Ω(n^2)

9. Which of the following defines Theta (Θ) notation?

Options: a) f(n) = O(g(n)) for some value of g(n)
b) f(n) = Ω(g(n)) for some value of g(n)
c) There exist constants c1, c2, and n0 such that c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0.
d) It gives the best-case time complexity of the algorithm.

Answer: c) There exist constants c1, c2, and n0 such that $c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$.

10. Which of the following statements is true about Theta (Θ) notation?

Options: a) It is used only when the upper and lower bounds of an algorithm are the same.
b) It represents the best-case time complexity of an algorithm.
c) It represents only the average-case time complexity.
d) It is used to describe the average running time between the upper and lower bounds.

Answer: a) It is used only when the upper and lower bounds of an algorithm are the same.

3. State True or False

1. Big O notation represents the lower bound of an algorithm's time complexity.

Answer: False

2. Omega (Ω) notation is used to represent the worst-case time complexity of an algorithm.

Answer: False

3. Theta (Θ) notation represents both the upper and lower bounds of an algorithm's running time.

Answer: True

4. Big O notation is used to analyze the best-case scenario for an algorithm's time complexity.

Answer: False

5. In Big O notation, the rate of growth of an algorithm is always considered for larger values of n.

Answer: True

6. Omega (Ω) notation provides a lower bound for the running time of an algorithm.

Answer: True

7. Theta (Θ) notation is only used when the upper and lower bounds of a function are the same.

Answer: True

8. The threshold value (n_0) in Big O notation determines when we start ignoring lower growth rates of an algorithm.

Answer: True

9. In the function f(n) = n^4 + 100n^2 + 10n + 50, the rate of growth is determined by n^4 at larger values of n.

Answer: True

10. The value of n_0 in Omega (Ω) notation indicates the point from which the lower bound of an algorithm becomes significant.

Answer: True

4. Fill in the blanks.

1. Big O notation represents the _____ bound of an algorithm's time complexity.

Answer: upper

2. Omega (Ω) notation represents the _____ bound of an algorithm's running time.

Answer: lower

3. Theta (Θ) notation is used when the _____ and _____ bounds of an algorithm's running time are the same.

Answer: upper, lower

4. The _____ notation specifies the worst-case time complexity of an algorithm.

Answer: Big O

5. In Big O notation, the maximum rate of growth of a function is represented by _____ for larger values of n.

Answer: g(n)

6. In Omega (Ω) notation, the minimum time required by an algorithm is given by the _____ case time complexity.

Answer: best

7. The value of n_0 in asymptotic notations is called the _____ for a given algorithm.

Answer: threshold

8. If f(n) = $100n^2 + 10n + 50$, the asymptotic tight lower bound is represented as _____ (n^2).

Answer: Ω

9. In the function $f(n) = n^4 + 100n^2 + 10n + 50$, the highest rate of growth is determined by _____.

Answer: n^4

10. Theta (Θ) notation is used to represent the _____ of the running time of an algorithm.

Answer: average-case complexity

Ch-5 Sorting Techniques

Objectives

By the end of this session, learners should be able to:

1. Define sorting and explain its importance in organizing data.
2. Understand how sorting improves efficiency in searching and database algorithms.
3. Identify different types of sorting algorithms and their characteristics.
4. Explain the selection sort process using a step-by-step breakdown.
5. Implement selection sort to sort a given dataset.
6. Explain the bubble sort process using a step-by-step breakdown.
7. Implement bubble sort to sort a given dataset.

Arranging elements in a given data set in ascending (Low to High) or descending (High to Low) form is called as sorting. For example for a given array, we can arrange elements of array in increasing order as shown below.

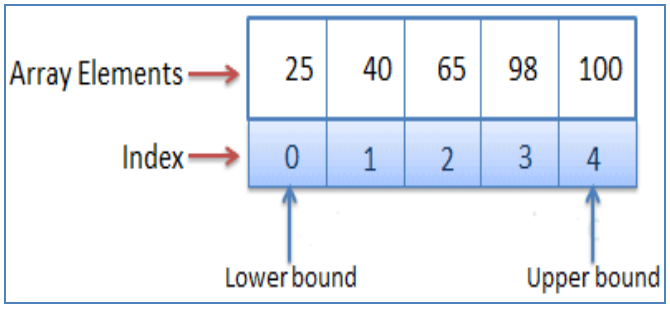


Fig 5.1 Sorted Array

Sorting can significantly reduce the complexity of a problem, and is often used for database algorithms and searches.

5.1 Types of Sorting

There are many types of sorting as given below.

1. Selection Sort: Simple comparison based sorting algorithm. Need no extra memory.
2. Bubble Sort: Repeatedly performs comparison of adjacent elements and swap them. This is simplest and in efficient method.
3. Insertion Sort: This method works on insertion of element at a particular position.
4. Merge Sort: Most efficient algorithm works on principle of divide and conquer.
5. Quick Sort: This method sorts elements faster than other methods.
6. Radix Sort: This method is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm.
7. Shell Sort: This method is a variation of insertion sort.

In this chapter we will discuss selection and bubble sort methods. Rest of the methods will be discussed in later chapters.

5.2 Selection Sort

In selection sort we divide the array into two parts: sorted and unsorted. The left part is sorted sub-array and the right part is unsorted sub-array. Figure 5.2 shows selection sort example.

Initially, sorted sub-array is empty and unsorted array is the complete given array.

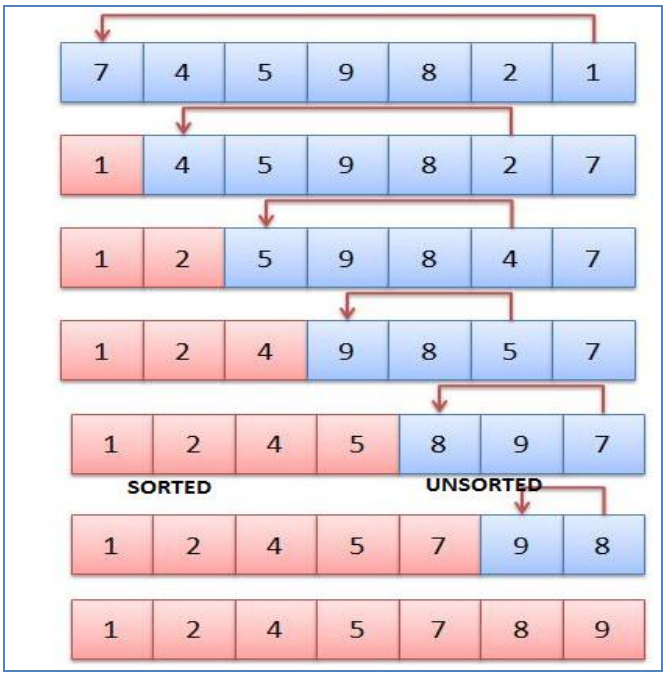


Fig 5.2: Selection Sort

We perform the steps given below until the unsorted sub-array becomes empty:

- Pick the minimum element from the unsorted sub-array.
- Swap it with the leftmost element of the unsorted sub-array.
- Now the leftmost element of unsorted sub-array becomes a part (rightmost) of sorted sub-array and will not be a part of unsorted sub-array.

5.2.1 Algorithm for Selection Sort

```
Input  : Array A of n elements.
Result : Permutation of A such that
         A[1] ≤ A[2] ≤ A[3] ≤ ... ≤ A[n].

procedure SelectionSort(A[ ],n)
1 if (n ≤ 1) then
2 |  return;
3 else
4 |  for i ← 1 to n − 1 do
5 |  |  if (A[i] > A[n]) then Swap(A[i], A[n]);
6 |  end
7 |  SelectionSort(A[ ],n − 1);
8 end
```

5.2.2 Complexity of Selection Sort

The complexity of Selection Sort is as follows:

Time Complexity

1. Best Case (Already Sorted):
   Even if the array is sorted, Selection Sort still performs comparisons to find the minimum element for each position.
   Time Complexity: $O(n^2)$
2. Worst Case (Reverse Sorted):
   The algorithm performs the same number of comparisons regardless of the input order.
   Time Complexity: $O(n^2)$
3. Average Case:
   The number of comparisons is always the same as it depends only on the size of the array.
   Time Complexity: $O(n^2)$

Space Complexity

Selection Sort is an in-place sorting algorithm, meaning it doesn't require extra space for sorting other than a constant amount for variables.
Space Complexity: O(1)

5.2.3 C++ Program for Selection Sort

```
// C++ program for selection sort

#include <bits/stdc++.h>

using namespace std;

//Swap function

void swap(int *xp, int *yp)

{	int temp = *xp;

	*xp = *yp;

	*yp = temp; }

void selectionSort(int arr[], int n)

{	int i, j, min_idx;

	// One by one move boundary of  unsorted subarray

	for (i = 0; i < n-1; i++)

	{

		// Find the minimum element in

		// unsorted array

		min_idx = i;

		for (j = i+1; j < n; j++)

		if (arr[j] < arr[min_idx])

			min_idx = j;

		// Swap the found minimum element

		// with the first element

		swap(&arr[min_idx], &arr[i]);
```

```
        }
}
//Function to print an array
void printArray(int arr[], int size)
{       int i;
        for (i=0; i < size; i++)
                cout << arr[i] << " ";
        cout << endl;
}
// Main program to test above functions
int main()
{       int arr[] = {64, 25, 12, 22, 11};
        int n = sizeof(arr)/sizeof(arr[0]);
        selectionSort(arr, n);
        cout << "Sorted array: ";
        printArray(arr, n);
        return 0;
}
```

Output:

Sorted array: 11 12 22 25 64

5.2.4 Program Explanation

The statement #include <bits/stdc++.h> using namespace std includes all C++ standard libraries and allows the use of names from the standard library for variables and objects. The function swap() is used for interchanging values of variables. The function selectionSort() implements selection sort algorithm. The function printArray() is used for displaying content of an array. The function main() calls sorting function and display sorted array.

5.3 Bubble Sort

In Bubble sort, we compare adjacent elements and see if their order is wrong. That is, a[i] > a[j] for 1 <= i < j <= size of array; if array is to be in ascending order, and vice-versa. If yes, then swap them. Fig 5.3 shows bubble sorting example.

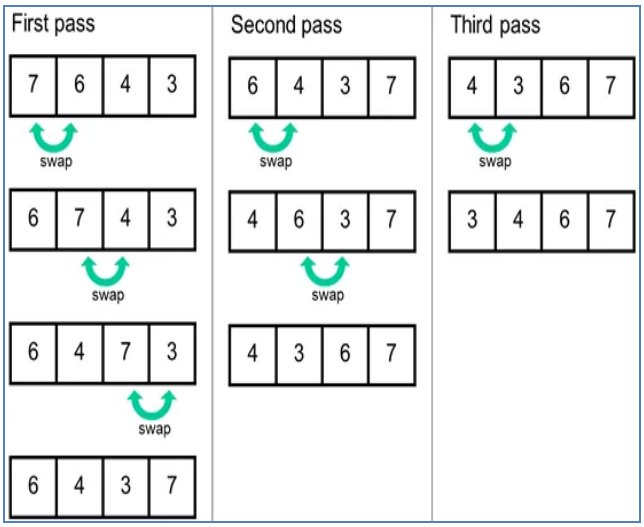
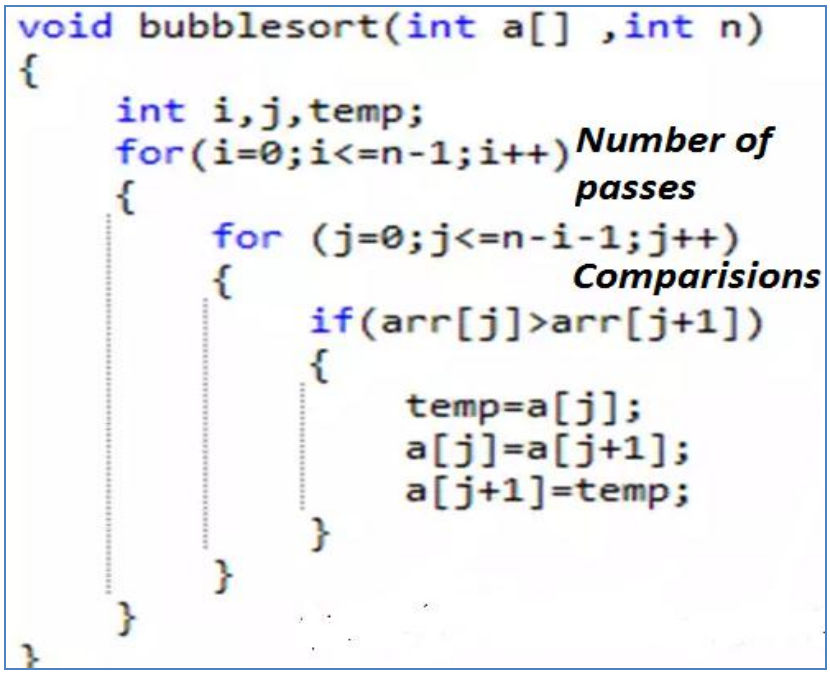


Fig 5.3: Bubble Sort

Suppose we have an array of length n. To sort this array we do the swapping step n - 1 passes. In simple terms, first, the largest element goes at its extreme right place then, second largest to the last by one place, and so on. In the i^{th} pass, the i^{th} largest element goes at its right place in the array by swapping.

In real life, bubble sort can be visualized when people in a queue wanting to be standing in a height wise sorted manner.

5.3.1 Algorithm for Bubble Sort



5.3.2 Complexity of Bubble Sort Algorithm

The complexity of Bubble Sort is as follows:

Time Complexity

1. Best Case (Already Sorted):
   If the array is already sorted, Bubble Sort can detect it early with the help of a flag variable. In this case, only one pass is required.
   Time Complexity: O(n)

2. Worst Case (Reverse Sorted):
   The algorithm performs the maximum number of comparisons and swaps for a reverse-sorted array.
   Time Complexity: $O(n^2)$

3. Average Case (Random Order):
   On average, the algorithm performs $O(n^2)$ comparisons and swaps.
   Time Complexity: $O(n^2)$

Space Complexity

Bubble Sort is an in-place sorting algorithm, meaning it does not require extra space apart from a few variables for swapping.
Space Complexity: O(1)

5.3.3 C ++ Program for Bubble Sort

```
// C++ program for the Bubble sort
#include <bits/stdc++.h>
using namespace std;
void bubbleSort(vector<int>& v) {
   int n = v.size();
   // Outer loop that corresponds to the number of elements to be sorted
   for (int i = 0; i < n - 1; i++) {
      // Last i elements are already in place
      for (int j = 0; j < n - i - 1; j++) {
                  // Comparing adjacent elements
         if (v[j] > v[j + 1])
                  // Swapping if in the wrong order
            swap(v[j], v[j + 1]);
      }
   }
}
int main() {
   vector<int> v = {5, 1, 4, 2, 8};
   // Sorting the vector v
   bubbleSort(v);
   for (auto i : v)
      cout << i << " ";
   return 0;
}
```

Output:

1 2 4 5 8

5.3.4 Program Explanation

The statement #include <bits/stdc++.h> using namespace std includes all C++ standard libraries and allows the use of names from the standard library for variables and objects. The using namespace std; allows the use of standard library functions without the std:: prefix. The function bubbleSort() implements bubble sort algorithm. This function takes a vector v (list of integers) as input. The n = v.size(); stores the size of the vector. The function main() calls sorting function and display sorted array.

5.3.5 Limitations of Bubble Sort

1. Bubble Sort is easy to implement but inefficient for large datasets.

2. It repeatedly swaps adjacent elements to arrange them in order.

3. More efficient sorting algorithms like Quick Sort or Merge Sort are preferred for large data sets.

Summary

Sorting is the process of arranging elements in a dataset in either ascending or descending order. It plays a crucial role in optimizing search operations and database algorithms. There are several sorting techniques, each with its own advantages and limitations. Selection Sort selects the smallest element and swaps it with the leftmost unsorted element. It has a time complexity of $O(n^2)$ in all cases and requires no extra space (O(1)). Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. It has a best-case time complexity of O(n) (if already sorted) but generally performs $O(n^2)$ comparisons. Other efficient sorting methods include Insertion Sort, Merge Sort, Quick Sort, Radix Sort, and Shell Sort.

Questions

1. **Answer the following questions**

(i) What is sorting technique?

(ii) State different types of sorting.

(iii) What is selection sort? Write algorithm for selection sort.

(iv) What is bubble sort? Write algorithm for bubble sort.

(v) What is complexity of selection sort algorithm?

(vi) What is complexity of bubble sort algorithm?

(vii) Write and execute program to implement selection sort using C++.

(viii) Write and execute program to implement bubble sort using C++.

2. Multiple Choice Questions

(i) What is sorting?

a) Arranging elements in random order
b) Arranging elements in ascending or descending order
c) Removing elements from a data set
d) Combining two arrays

Answer: b) Arranging elements in ascending or descending order

(ii) Which sorting algorithm is based on the principle of divide and conquer?

a) Bubble Sort
b) Selection Sort
c) Merge Sort
d) Insertion Sort

Answer: c) Merge Sort

(iii) Which of the following sorting algorithms is the simplest but least efficient?

a) Quick Sort
b) Bubble Sort
c) Merge Sort
d) Radix Sort

Answer: b) Bubble Sort

(iv) In Selection Sort, the sorted sub-array is initially:

a) The entire array
b) Empty
c) Half the array
d) All even-indexed elements

Answer: b) Empty

(v) What is the time complexity of Selection Sort in the best case?

a) O(n)
b) $O(n^2)$
c) O(log n)
d) O(nlog n)

Answer: b) $O(n^2)$

(vi) Bubble Sort detects an already sorted array in:
a) $O(n^2)$ time
b) O(n) time
c) O(log n) time
d) O(1) time

Answer: b) O(n) time

(vii) Which of the following is true about Bubble Sort?
a) It is not an in-place algorithm
b) It uses a flag variable to detect early sorting
c) It divides the array into halves
d) It requires additional space of O(n)O(n)O(n)

Answer: b) It uses a flag variable to detect early sorting

(viii) What is the output of the given Selection Sort program?
Input: {64, 25, 12, 22, 11}
a) 64 25 12 22 11
b) 11 12 22 25 64
c) 22 25 11 12 64
d) 12 22 11 64 25

Answer: b) 11 12 22 25 64

(ix) Which of the following is a stable sorting algorithm?
a) Selection Sort
b) Bubble Sort
c) Quick Sort
d) Shell Sort

Answer: b) Bubble Sort

(x) In Bubble Sort, the number of passes required to sort an array of size n is:
a) n
b) n−1
c) n+1
d) n/2

Answer: b) n−1

3. State True or False

(i) Sorting refers to arranging elements in ascending or descending order.
Ans: True
(ii) Bubble Sort is the most efficient sorting algorithm discussed in this chapter.
Ans: False

(iii) Merge Sort is based on the divide and conquer principle.
Ans: True

(iv) Selection Sort divides the array into a sorted and unsorted sub-array.
Ans: True

(v) In Selection Sort, the sorted sub-array is initially the entire array.
Ans: False

(vi) Bubble Sort requires additional space of O(n) for sorting.
Ans: False

(vii) In the best case, Bubble Sort has a time complexity of O(n).
Ans: True

(viii) Selection Sort has a time complexity of $O(n^2)$ in all cases.
Ans: True

(ix) Bubble Sort uses a flag variable to detect if the array is already sorted.
Ans: True

(x) The output of the given Selection Sort program can be 11,12,22,25,64.
Ans: True

(xi) Shell Sort is a variation of Merge Sort.
Ans: False

(xii) Bubble Sort involves comparing adjacent elements and swapping them if they are in the wrong order.
Ans: True

(xiii) In Bubble Sort, the largest element moves to its correct position in each pass.
Ans: True

(xiv) Selection Sort is a stable sorting algorithm.
Ans: False

4. Fill in the blanks

(i) Sorting refers to arranging elements in _____ or _____ order.
Ans: ascending, descending

(ii) Bubble Sort repeatedly performs comparison of _____ elements and swaps them if necessary.
Ans: adjacent

(iii) Merge Sort works on the principle of _____ and _____.
Ans: divide, conquer

(iv) In Selection Sort, the array is divided into _____ and _____ sub-arrays.
Ans: sorted, unsorted

(v) In Selection Sort, the _____ element from the unsorted sub-array is swapped with the leftmost element.
Ans: minimum

(vi) The time complexity of Selection Sort in the worst-case scenario is _____.
Ans:$(O(n^2))$

(vii) Bubble Sort has a best-case time complexity of _____ when the array is already sorted.
Ans: (O(n))

(viii) Selection Sort does not require extra memory, making its space complexity _____.
Ans: (O(1))

(ix) In Bubble Sort, the _____ element moves to its correct position after each pass.
Ans: (largest)

(x) In the Selection Sort algorithm, the function _____ is used to interchange the values of variables.
Ans: swap

(xi) Shell Sort is a variation of _____ Sort.
Ans: insertion

(xii)The function bubbleSort() uses _____ loops to sort the array.
Ans:nested

(xiii) The standard library header included in the provided C++ programs is _____.
Ans: #include <bits/stdc++.h>

(xiv) The space complexity of Bubble Sort is _____ as it sorts the array in place.
Ans: O(1)

(xv)In Selection Sort, the point from which the sorted and unsorted sub-arrays are divided moves _____ after each iteration.
Ans: right

Ch-6 Insertion and Merge Sorting Techniques

Objectives

By the end of this session, learners should be able to:

1. Understand the Concept how insertion sort works by comparing, shifting, and inserting elements into their appropriate positions.
2. Analyze the time complexity in different cases (Best: O(n), Worst: $O(n^2)$, Average: $O(n^2)$) and space complexity (O(1)).
3. Identify the simplicity, efficiency for small datasets, and stability of insertion sort.
4. Understand why insertion sort is inefficient for large, unsorted datasets due to its quadratic time complexity.

6.1 Insertion Sort

In insertion sort the first step involves the comparison of the element in question with its adjacent element. If at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.

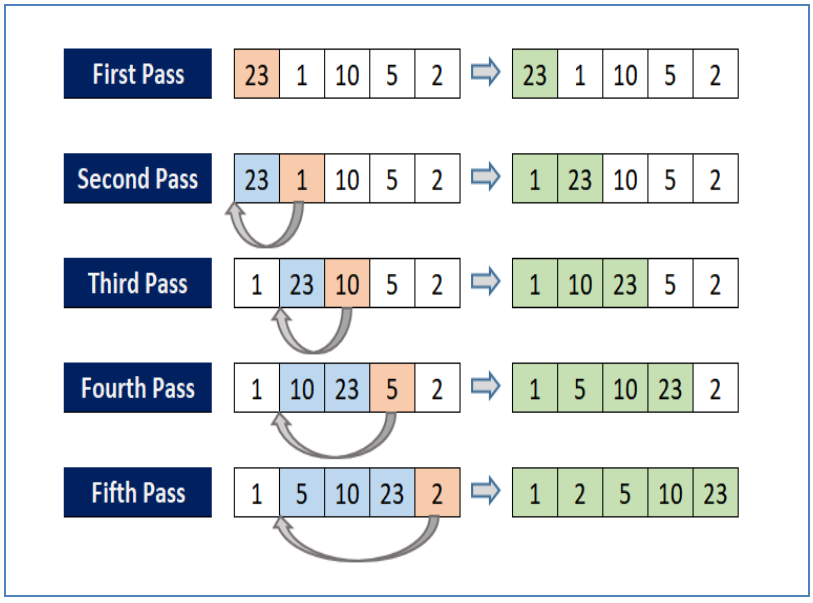


Fig 6.1: Insertion Sort

The above procedure is repeated until all the element in the array is at their appropriate position. Insertion sort is efficient, still, its best case time complexity a linear function of n.

Wherein for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for n x n, i.e.,n^2 comparisons.

6.2 Algorithm for Insertion Sort

```
void insertionSort(Item a[], int n)
{
    for (int i = 1; i < n; i++)
    {
        Item tmp = a[i]; Test element

        for (int j=i; j>0 && tmp < a[j-1]; j--)
            a[j] = a[j-1]; Move element
        a[j] = tmp;
    }
}
```

6.2.1 Complexity of Algorithm

Time Complexity

1. Best Case O(n):

When the array is already sorted, the inner loop only runs once for each element. The algorithm performs n−1 comparisons and no shifts are needed. Time Complexity = O(n).

2. Worst Case O(n^2):

When the array is sorted in reverse order, each element must be compared with all previous elements and shifted to the beginning. For the ithi^{th}ith element, iii comparisons and iii shifts are performed, resulting in ∑ I = (n(n−1)/2. Time Complexity = O(n^2).

3. Average Case O(n^2):

On average, for each element, it is assumed to be compared with half of the previously sorted elements. This still results in a quadratic complexity. Time Complexity = O(n^2).

Space Complexity

Space Complexity = O(1):

Insertion sort is an in-place sorting algorithm, meaning it does not require additional memory for temporary arrays or data structures.

6.3 Advantages of Insertion Sort

Following are advantages of insertion sort.

1. Simple to implement.
2. Efficient for small or nearly sorted data sets.
3. Stable (does not change the relative order of equal elements).

6.4 Disadvantages of Insertion Sort

Inefficient for large or random data sets due to its $O(n^2)$ time complexity in the worst case.

6.5 C++ Program for Insertion Sort

```
// C++ program for insertion sort

#include <bits/stdc++.h>

using namespace std;

// Function to sort an array using insertion sort

void insertionSort(int arr[], int n)

{

        int i, key, j;

        for (i = 1; i < n; i++) {

                key = arr[i];

                j = i - 1;

                // Move elements of arr[0..i-1],

                // that are greater than key, to one position ahead of their current position

                while (j >= 0 && arr[j] > key) {

                        arr[j + 1] = arr[j];

                        j = j - 1;
```

```
                }
                arr[j + 1] = key;
        }
}
// A utility function to print an array of size n
void printArray(int arr[], int n)
{
        int i;
        for (i = 0; i < n; i++)
                cout << arr[i] << " ";
        cout << endl;
}
int main()
{
        int arr[] = { 12, 11, 13, 5, 6 };
        int N = sizeof(arr) / sizeof(arr[0]);
        insertionSort(arr, N);
        printArray(arr, N);
        return 0;
}
```

Output:

```
1 2 4 5 8
```

6.5.1 Program Explanation

Initially all header files are included in program by using #include statement. The using namespace std; allows direct use of standard functions like cout without prefixing std::. The function insertionSort() is used for sorting of elements. arr[] is the array to be sorted. n is the number of elements in the array. key

is the current element being inserted into the sorted portion. i and j are loop variables. The function printArray() is used to print content of array.

6.6 Merge Sort

Merge Sort is a comparison-based sorting algorithm that uses divide and conquer paradigm to sort the given dataset. It divides the dataset into two halves, calls itself for these two halves, and then it merges the two sorted halves.

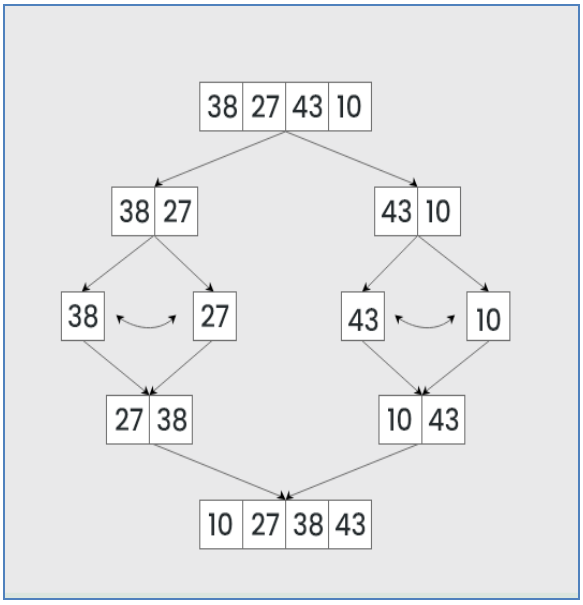


Fig 6.2: Merge Sort

6.7 Algorithm for Merge Sort

Merge sort algorithm divides the given array into smaller sub arrays and then uses the merge() process for merging two halves. It is the key process of merge sort that assumes that given two arrays are sorted and merges the two sorted sub-arrays into one sorted sub array.

Steps:

1. First calculate the midpoint of the array.

2. Then divide the whole array into equal halves using this midpoint.

3. For both the sub arrays, calculate the mid-point again.

4. Further divide these two sub arrays into further halves, until the atomic (single element) sub array is reached, and further division is not possible.

5. Start combining the elements again using merge process.

In merge process, compare the element for each list and then combine them into another list in a sorted manner.

6.8 Complexity of Merge Sort

6.8.1 Time Complexity

1. Best Case O(nlog n):

Even in the best case (when the array is already sorted), Merge Sort still divides the array and performs the merging process. The merging step always requires comparing and combining elements. Time Complexity = O(nlog n).

2. Worst Case O(nlog n):

The algorithm divides the array into two halves recursively until each subarray contains a single element (log n levels of recursion). Merging two sorted sub arrays requires O(n) time for each level. Time Complexity = O(nlog n).

3. Average Case O(nlog n):

The recursive division and merging process remain the same regardless of the initial ordering of the elements. Time Complexity = O(nlog n).

6.8.2 Space Complexity

Space Complexity = O(n):

Merge Sort requires additional memory for temporary arrays during the merge operation. The space complexity depends on the implementation. Top-down implementation requires O(n) auxiliary space for temporary arrays. Bottom-up implementation also requires O(n) space.

6.9 Advantages of Merge Sort

1. Efficient for large datasets.
2. Stable sorting algorithm, that is, preserves the relative order of equal elements.
3. Guarantees O(n log n) time complexity in all cases.

6.10 Disadvantages of Merge Sort

1. Requires additional O(n) memory, which can be a limitation for memory-constrained systems.
2. Slower than in-place algorithms such as Quick Sort for smaller arrays due to the overhead of recursive calls and merging.

6.11 C++ Program for Merge Sort

```
// C++ program for the merge sort
#include <iostream>
#include <vector>
using namespace std;
// Merges two subarrays of vec. First subarray is vec[left..mid]
// Second subarray is vec[mid+1..right]
void merge(vector<int>& vec, int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;
   // Create temporary vectors
   vector<int> leftVec(n1), rightVec(n2);
   // Copy data to temporary vectors
   for (i = 0; i < n1; i++)
      leftVec[i] = vec[left + i];
   for (j = 0; j < n2; j++)
      rightVec[j] = vec[mid + 1 + j];
   // Merge the temporary vectors back into vec[left..right]
   i = 0;
   j = 0;
   k = left;
   while (i < n1 && j < n2) {
      if (leftVec[i] <= rightVec[j]) {
         vec[k] = leftVec[i];
```

```
            i++;
        } else {
            vec[k] = rightVec[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of leftVec[], if any
    while (i < n1) {
        vec[k] = leftVec[i];
        i++;
        k++;
    }
    // Copy the remaining elements of rightVec[], if any
    while (j < n2) {
        vec[k] = rightVec[j];
        j++;
        k++;
    }
}
// The subarray to be sorted is in the index range [left..right]
void mergeSort(vector<int>& vec, int left, int right) {
    if (left < right) {
        // Calculate the midpoint
        int mid = left + (right - left) / 2;
```

```
        // Sort first and second halves
        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);
        // Merge the sorted halves
        merge(vec, left, mid, right);
    }
}
int main() {
    vector<int> vec = {12, 11, 13, 5, 6, 7};
    int n = vec.size();
    // Sorting vec using mergesort
    mergeSort(vec, 0, n - 1);
    for (auto i: vec)
        cout << i << " ";
    return 0;
}
```

Output:

```
5  6  7  11  12  13
```

6.11.1 Program Explanation

Initially all header files are included in program by using #include statement. The <iostream> is used for input-output operations. <vector> is used to store and manipulate dynamic arrays. The using namespace std; allows direct use of standard library functions without prefixing std::.The function merge() is used for merging of two arrays. The function mergeSort() is used for sorting of array.

6.12 Step-by-Step Execution Example

Initial Array:

[12, 11, 13, 5, 6, 7]

Splitting Phase:

Split [12, 11, 13, 5, 6, 7] into [12, 11, 13] and [5, 6, 7]

Split [12, 11, 13] into [12] and [11, 13]

Split [11, 13] into [11] and [13] (already sorted)

Merge [11] and [13] → [11, 13]

Merge [12] and [11, 13] → [11, 12, 13]

Split [5, 6, 7] into [5] and [6, 7]

Split [6, 7] into [6] and [7] (already sorted)

Merge [6] and [7] → [6, 7]

Merge [5] and [6, 7] → [5, 6, 7]

Merging Phase:

Merge [11, 12, 13] and [5, 6, 7] → [5, 6, 7, 11, 12, 13]

Summary

Insertion Sort is a simple sorting algorithm that works by comparing an element with its adjacent values and inserting it in the correct position by shifting other elements as needed. It is efficient for small or nearly sorted datasets but performs poorly for large, random datasets due to its $O(n^2)$ worst-case time complexity. The algorithm operates in-place with a space complexity of O(1), making it memory efficient. Its main advantages include ease of implementation, stability (preserving the order of equal elements), and effectiveness for small inputs. However, its inefficiency for large datasets limits its practical use compared to more advanced algorithms like Merge Sort.

Questions

1. **Answer the following questions.**

    (i) What is sorting of an array?

    (ii) What is insertion sort?

    (iii) Write algorithm for insertion sort.

    (iv) State time and space complexity of insertion sort.

(v) What is merge sort?

(vi) Write algorithm for merge sort.

(vii) State time and space complexity of merge sort.

(viii) State advantages and disadvantages of insertion sort.

(ix) State advantages and disadvantages of merge sort.

2. Multiple Choice Questions

1. What is the primary operation performed during the first step of the insertion sort algorithm?

a) Merging elements
b) Comparing the element in question with its adjacent element
c) Dividing the array into two halves
d) Swapping all elements in the array

Answer: b) Comparing the element in question with its adjacent element

2. How does insertion sort handle placing an element at its correct position?

a) By swapping it with its adjacent elements until it reaches its position
b) By creating space by shifting other elements to the right
c) By merging it into the appropriate sub-array
d) By moving it directly to its position

Answer: b) By creating space by shifting other elements to the right

3. What is the time complexity of insertion sort in the best case?

a) O(1)
b) O(n)
c) $O(n^2)$
d) O(log n)

Answer: b) O(n)

4. In the worst case, how many comparisons does insertion sort perform for an array with nnn elements?

a) n
b) n−1
c) n^2
d) (n(n−1))/2

Answer: d) (n(n−1))/2

5. Why is insertion sort considered a stable sorting algorithm?

a) It uses extra memory for stability
b) It preserves the relative order of equal elements
c) It minimizes comparisons between elements
d) It avoids shifting elements unnecessarily

Answer: b) It preserves the relative order of equal elements

6. What type of datasets is insertion sort most efficient for?

a) Large, random datasets
b) Nearly sorted or small datasets
c) Datasets with many duplicate elements
d) Completely reversed datasets

Answer: b) Nearly sorted or small datasets

7. What is the space complexity of insertion sort?

a) O(1)
b) O(n)
c) O(log n)
d) $O(n^2)$

Answer: a) O(1)

8. Which of the following is a disadvantage of insertion sort?

a) It requires additional memory for temporary arrays
b) It is not stable
c) It is inefficient for large datasets due to $O(n^2)$ time complexity
d) It is difficult to implement

Answer: c) It is inefficient for large datasets due to $O(n^2)$ time complexity

9. During the insertion sort process, what happens in the worst-case scenario?

a) Each element is shifted to the rightmost position
b) Each element is compared with all previous elements and shifted to the beginning
c) The array is divided and merged recursively
d) No shifts or comparisons are required

Answer: b) Each element is compared with all previous elements and shifted to the beginning

10. What is the key feature that makes insertion sort an in-place algorithm?

a) It does not modify the original array
b) It sorts without using additional memory for temporary arrays
c) It merges elements during the process
d) It divides the array into sub-arrays

Answer: b) It sorts without using additional memory for temporary arrays

11. What paradigm does the merge sort algorithm follow?

a) Dynamic programming
b) Divide and conquer
c) Greedy algorithm
d) Backtracking

Answer: b) Divide and conquer

12. How does merge sort process the dataset?

a) It compares each element with all other elements
b) It divides the dataset into halves, sorts them, and merges the sorted halves
c) It swaps adjacent elements until the array is sorted
d) It shifts elements to their correct positions

Answer: b) It divides the dataset into halves, sorts them, and merges the sorted halves

13. What is the key step in merge sort that combines two halves of the dataset?

a) Sorting
b) Splitting
c) Merging
d) Partitioning

Answer: c) Merging

14. What happens when an array is divided during merge sort?

a) It is split into three equal parts
b) It is split into two equal halves until single elements remain
c) It is rearranged in descending order
d) It is sorted immediately

Answer: b) It is split into two equal halves until single elements remain

15. What is the time complexity of merge sort in the best case?

a) O(1)
b) O(n)
c) O(n log n)
d) $O(n^2)$

Answer: c) O(n log n)

16. Why does merge sort require additional memory?

a) To store intermediate sorted arrays during merging
b) To track the midpoint of the array
c) To swap adjacent elements
d) To store indices of sorted elements

Answer: a) To store intermediate sorted arrays during merging

17. What is the space complexity of merge sort?

a) O(1)
b) O(log n)
c) O(n)
d) $O(n^2)$

Answer: c) O(n)

18. Which of the following is an advantage of merge sort?

a) It is in-place and uses no extra memory
b) It is stable and guarantees O(nlog n)O(n \log n)O(nlogn) time complexity
c) It is faster for small arrays compared to other algorithms
d) It avoids recursion entirely

Answer: b) It is stable and guarantees O(nlog n)O(n \log n)O(nlogn) time complexity

19. What is a disadvantage of merge sort?

a) It cannot handle large datasets efficiently
b) It is unstable for equal elements
c) It requires additional memory for temporary arrays
d) It has a time complexity of $O(n^2)$ in the worst case

Answer: c) It requires additional memory for temporary arrays

20. In merge sort, when is the merge() function called?

a) After the dataset is divided into two halves
b) After two sorted sub-arrays are ready to be combined

c) During the initial division of the array
d) When the array is completely sorted

Answer: b) After two sorted sub-arrays are ready to be combined

3. State True or False

1. In insertion sort, each element is compared with its adjacent element to determine its correct position.

Answer: True

2. During the insertion process, elements are shifted one position to the left to create space for the element being inserted.

Answer: False (Elements are shifted one position to the right.)

3. Insertion sort is efficient for small or nearly sorted datasets.

Answer: True

4. In the best-case scenario, the time complexity of insertion sort is $O(n^2)$.

Answer: False (The best-case time complexity is O(n).

5. In the worst-case scenario, insertion sort requires $O(n^2)$ comparisons.

Answer: True

6. Insertion sort is an in-place sorting algorithm, meaning it does not require additional memory for temporary arrays.

Answer: True

7. Insertion sort is an unstable algorithm as it changes the relative order of equal elements.

Answer: False (Insertion sort is stable.)

8. For a reverse-sorted array, insertion sort performs O(n) comparisons.

Answer: False (For a reverse-sorted array, it performs $O(n^2)$ comparisons.)

9. The space complexity of insertion sort is O(1).

Answer: True

10. Insertion sort is faster than other algorithms for large and random datasets due to its simplicity.

Answer: False (Insertion sort is inefficient for large or random datasets due to its $O(n^2)$ time complexity in the worst case.)

11. Merge Sort is a comparison-based sorting algorithm that follows the divide and conquer paradigm.

Answer: True

12. The merge process in Merge Sort assumes that the two arrays being merged are already sorted.

Answer: True

13. In Merge Sort, the array is divided into halves until each sub-array contains at least three elements.

Answer: False (Each sub array is divided until it contains a single element.)

14. Merge Sort guarantees O(n log n) time complexity in the best, worst, and average cases.

Answer: True

15. Merge Sort is an in-place sorting algorithm as it does not require extra memory.

Answer: False (Merge Sort requires additional memory for temporary arrays, with space complexity O(n).)

16. The top-down implementation of Merge Sort requires O(1) auxiliary space.

Answer: False (Top-down implementation requires O(n) auxiliary space.)

17. Merge Sort is a stable sorting algorithm that preserves the relative order of equal elements.

Answer: True

18. The merging process in Merge Sort involves comparing elements from both sub-arrays and combining them into a new sorted array.

Answer: True

19. Merge Sort is more efficient than in-place algorithms like Quick Sort for smaller datasets.

Answer: False (Merge Sort is slower for smaller datasets due to recursive overhead.)

20. Merge Sort is highly efficient for sorting large datasets due to its O(n log n) time complexity.

Answer: True

4. Fill in the blanks

1. Merge Sort is a _____-based sorting algorithm that follows the divide and conquer paradigm.
   Answer: comparison

2. The merge process in Merge Sort assumes that the two arrays being merged are already _____.
   Answer: sorted

3. In Merge Sort, the array is divided into halves until each sub-array contains a _____ element.
   Answer: single

4. Merge Sort guarantees O(_____) time complexity in the best, worst, and average cases.
   Answer: nlogn

5. Merge Sort requires additional memory for _____ arrays during the merge operation.
   Answer: temporary

6. The space complexity of the top-down implementation of Merge Sort is O(_____).
   Answer: n

7. Merge Sort is a _____ sorting algorithm, meaning it preserves the relative order of equal elements.
   Answer: stable

8. The merging process in Merge Sort involves _____ elements from both subarrays and combining them into a new sorted array.
   Answer: comparing

9. Merge Sort is _____ than in-place algorithms like Quick Sort for smaller datasets.
   Answer: slower

10. Merge Sort is highly efficient for sorting large datasets due to its O(_____) time complexity.
    Answer: nlogn
11. In insertion sort, the first step involves the _____ of the element in question with its adjacent element.
    Answer: comparison
12. If at every comparison, the element in question can be inserted at a particular position, then space is created for it by _____ the other elements one position to the right.
    Answer: shifting
13. The above procedure is repeated until all the elements in the array are at their _____ position.
    Answer: appropriate
14. The best-case time complexity of insertion sort is a _____ function of n.
    Answer: linear
15. For an unsorted array, it takes n×n, i.e., n^2 _____ in the worst case.
    Answer: comparisons
16. Insertion sort is an _____ sorting algorithm, meaning it does not require additional memory for temporary arrays.
    Answer: in-place

17. In the best case O(n), the algorithm performs n−1 _____ and no shifts are needed.
    Answer: comparisons
18. In the worst case O(n^2), when the array is sorted in reverse order, each element must be _____ to the beginning.
    Answer: shifted
19. The time complexity in the average case for insertion sort is O(_____).
    Answer: n^2
20. Insertion sort is _____ for small or nearly sorted datasets.
    Answer: efficient
21. Insertion sort is _____, meaning it does not change the relative order of equal elements.
    Answer: stable
22. Insertion sort is inefficient for _____ datasets due to its O(n^2) time complexity in the worst case.
    Answer: large

Ch-7 Radix, Shell and Quick Sorting Techniques

Objectives

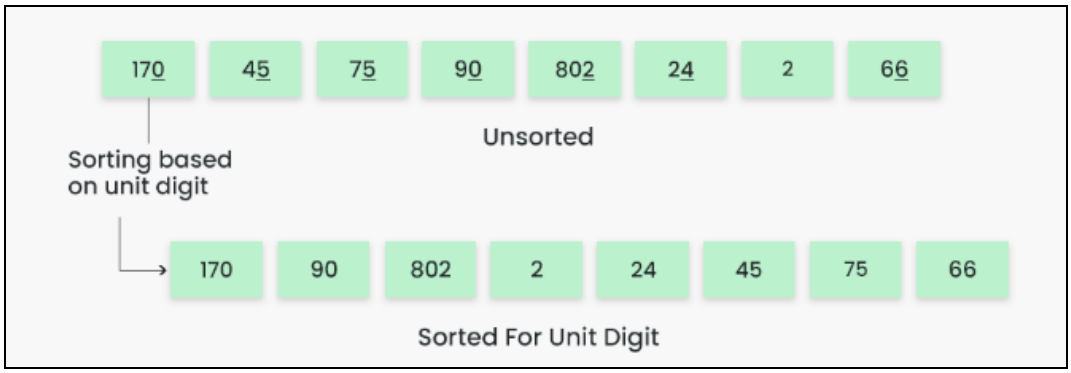
By the end of this session, learners should be able to:

1. Understand the Concept of Radix Sort
2. Understand concept of shell sort
3. Learn shell sort algorithm
4. Learn the Radix Sort Algorithm
5. Understand concept of quick sort
6. Learn quick sort algorithm
7. Identify Strengths and Limitations of radix sort, shell sort and quick sort

7.1 Radix Sort

It is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order. Fig 7.1 shows working of Radix sort.



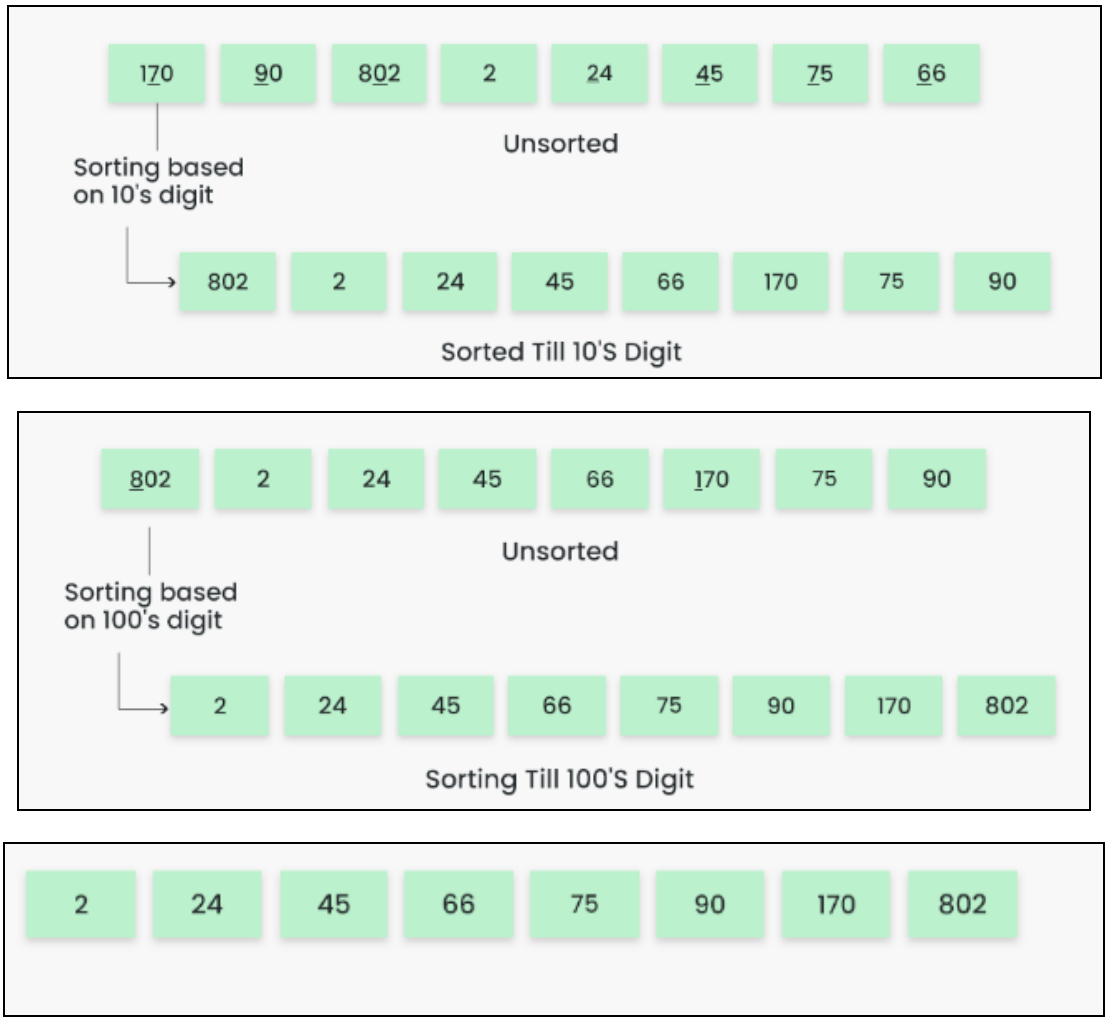


Fig 7.1: Radix Sort

7.1.1 Algorithm for Radix Sort

Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.

Step 2: Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.

It's important to understand that the default implementation of counting sort is unstable i.e. same keys can be in a different order than the input array. To solve this problem, we can iterate the input array in reverse order to build the output array. This strategy helps us to keep the same keys in the same order as they appear in the input array.

7.1.2 Complexity of Radix Sort

7.1.2.1 Time Complexity

Radix Sort processes each digit of the numbers being sorted. The time complexity depends on:

- **n**: The number of elements in the array.
- **k**: The number of digits (or bits) in the largest number.
- **b**: The base used for grouping (e.g., base 10 for decimal digits, base 2 for binary digits).

The key steps involve:

1. **Digit Extraction**: For each digit or group of digits, the algorithm distributes the numbers into buckets.
2. **Stable Sorting**: Each iteration requires a stable sorting algorithm like Counting Sort, which operates in O(n).

Time Complexity Formula: $O(d \cdot (n+b))$

Where:

- d: Number of passes or digits (related to k, the maximum number of digits in the largest number, and the base b).
- n: Number of elements to sort.
- b: Base used for grouping digits.

Special Cases:

- If b (the base) is chosen optimally, such as close to n, the complexity becomes approximately O(n).
- For integers with k bits and base b=2, $d=k/\log_2 b$

7.1.2.2 Space Complexity

Radix Sort requires additional space for:

- Buckets or arrays to store grouped elements.
- Auxiliary data structures during the sorting process.

The **space complexity** is O(n+b), where n is for storing the input and b is the number of buckets.

7.1.2.3 Comparison with Other Algorithms

Radix Sort is faster than comparison-based sorts (like Merge Sort or Quick Sort) when the number of digits d is small relative to n. It is most efficient for fixed-length integers or strings. Radix Sort is not an in-place algorithm, as it requires additional memory.

7.2 Shell Sort

It is mainly a variation of insertion sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved.

The idea of Shell Sort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sub lists of every h'th element are sorted. Fig 7.2 shows working of Shell sort.

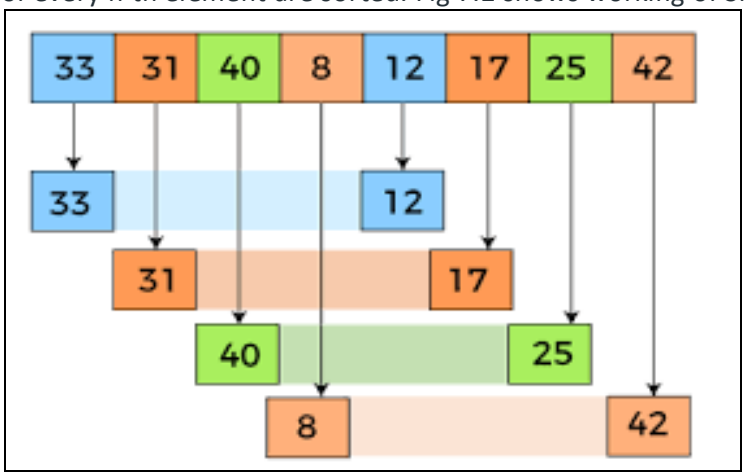


Fig 7.2: Shell Sort

7.2.1 Algorithm for Shell Sort

1. Initialize the value of gap size, say h.
2. Divide the list into smaller sub-part. Each must have equal intervals to h.
3. Sort these sub-lists using insertion sort.
4. Repeat this step 2 until the list is sorted.
5. Print a sorted list.
6. Stop.

7.2.2 Complexity of Algorithm

7.2.2.1 Time Complexity

The time complexity of Shell Sort is influenced by the number of elements n and the gap sequence used.

Worst-Case Time Complexity: The worst-case time complexity depends heavily on the gap sequence: For Shell's original sequence (n/2,n/4,…,1n/2, n/4, \dots, 1n/2,n/4,…,1): $O(n^2)$.

Average-Case Time Complexity: The average-case performance also depends on the gap sequence but is generally better than $O(n^2)$.

The best case occurs when the array is already sorted or nearly sorted: O(n log n) for efficient gap sequences.

7.2.2.2 Space Complexity

Shell Sort is an in-place sorting algorithm, so it requires no additional memory apart from the input array. Space Complexity: O(1).

Shell Sort is generally not stable because elements can be moved far apart during gap sorting, disrupting their relative order.

7.3 Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot. It partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

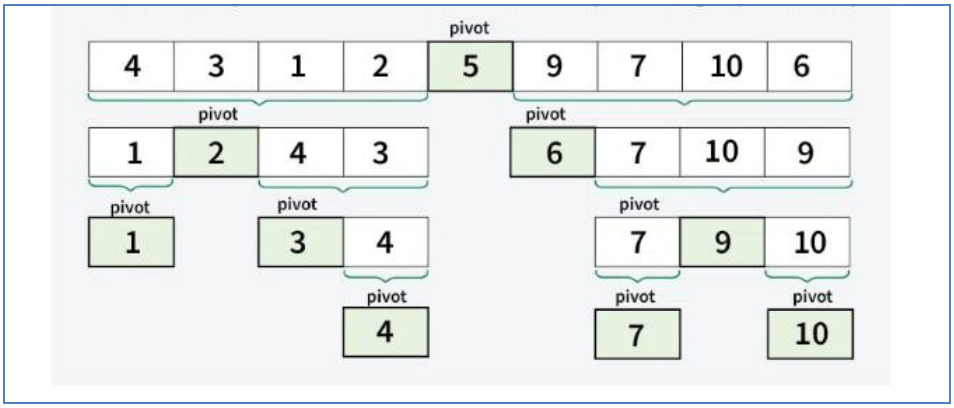


Fig 7.: Quick Sort

7.3.1 Algorithm for Quick Sort

There are mainly three steps in the algorithm:

1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

7.3.2 Complexity of Quick Sort

7.3.2.1 Time Complexity

Best Case

Best Case Occurs when the pivot divides the array into two nearly equal parts at every step. Each partitioning step takes O(n), and the depth of recursion is O(log n) because the array size halves with each step. Time Complexity (Best Case):O(nlog n)

Worst Case

Worst case Occurs when the pivot is the smallest or largest element, resulting in highly unbalanced partitions (e.g., one side has n−1 elements, and the other has 0). In this case, the recursion depth is O(n), and each step still takes O(n).

Time Complexity (Worst Case): $O(n^2)$

Average Case

Assuming the pivot divides the array into reasonably balanced parts, the average number of comparisons is proportional to O(n log n). Time Complexity (Average Case): O(n log n).

7.3.2.2 Space Complexity

Quick Sort is an in-place algorithm**,** so it doesn't require extra space for the array, but it does need space for the recursion stack**.** Best and Average Case: O(log n) for the recursion stack (logarithmic depth of recursion). Worst Case: O(n) for the recursion stack in the case of highly unbalanced partitions.

Summary

Radix Sort is a linear sorting algorithm that processes elements digit by digit, distributing them into buckets based on each digit's value. It efficiently sorts integers and fixed-size keys without direct comparisons. The algorithm involves multiple passes, sorting elements based on their least to most significant digits using a stable sorting method like Counting Sort. Its time complexity depends on the number of digits and base used, typically O(d * (n + b)), making it efficient when d (number of digits) is small. Radix Sort requires additional memory for bucket storage, leading to a space complexity of O(n + b). Compared to comparison-based sorting algorithms like Merge Sort and Quick Sort, it performs better for fixed-length numbers but is not in-place. While it ensures stable sorting and linear time complexity under optimal conditions, it may not be suitable for small datasets or large numbers with high digit counts due to its extra space requirements.

Questions

1. **Answer the following questions**.

    (i) What is radix sort?

(ii) Write algorithm for radix sort.

(iii) Compute time and space complexity of radix sort.

(iv) What is shell sort?

(v) Write algorithm for shell sort.

(vi) Compute time and space complexity of shell sort.

(vii) What is quick sort?

(viii) Write algorithm for quick sort.

(ix) Compute time and space complexity of quick sort.

2. Multiple Choice Questions

1. Radix Sort processes elements by:

(i) Comparing elements directly.

(ii) Distributing elements into buckets based on digit values.

(iii) Dividing the array into smaller sub-arrays.

(iv) Using the divide-and-conquer approach.

Answer:
(ii) Distributing elements into buckets based on digit values.

2. Which of the following is true about Radix Sort?

(i) It is a comparison-based sorting algorithm.

(ii) It is efficient for sorting integers or fixed-size strings.

(iii) It sorts elements in descending order by default.

(iv) It requires O(1) additional memory.

Answer:
(ii) It is efficient for sorting integers or fixed-size strings.

3. The time complexity of Radix Sort is:

(i) $O(n^2)$

(ii) O(n log n)

(iii) O(d·(n+b))

(iv) O(n)

Answer:
(iii) O(d·(n+b))

4. What does the variable d represent in Radix Sort's time complexity formula O(d·(n+b))?

(i) The number of elements in the array.

(ii) The number of digits in the largest number.

(iii) The base used for grouping digits.

(iv) The number of buckets used for sorting.

Answer:
(ii) The number of digits in the largest number.

5. The space complexity of Radix Sort is:

1. O(1)

2. O(n)

3. O(n+b)

4. O(log n)

Answer:
(iii) O(n+b)

6. Radix Sort is not an in-place algorithm because:

(i) It uses recursive calls.

(ii) It requires additional memory for buckets and auxiliary structures.

(iii) It performs comparisons between elements.

(iv) It sorts elements by swapping them in the array.

Answer:
(ii) It requires additional memory for buckets and auxiliary structures.

7. Shell Sort is a variation of which sorting algorithm?

(i) Bubble Sort

(ii) Merge Sort

(iii) Insertion Sort

(iv) Quick Sort

Answer:
(iii). Insertion Sort

8. What is the main idea behind Shell Sort?

(i) To compare adjacent elements and swap them.

(ii) To allow the exchange of far items in the array.

(iii) To divide the array into two equal halves and sort them.

(iv) To sort the array by repeatedly merging sub-arrays.

Answer:
(ii). To allow the exchange of far items in the array.

9. What does it mean for an array to be h-sorted in Shell Sort?

(i) The array is fully sorted.

(ii) All sub-lists of every h th element are sorted.

(iii) Elements within a sub-array are sorted.

(iv) The array is divided into h smaller arrays.

Answer:
(ii) All sub-lists of every h th element are sorted.

10. Which step is repeated until the array is sorted in Shell Sort?

(i) Adjusting the pivot position.

(ii) Dividing the array into two halves.

(iii) Reducing the value of the gap size hhh.

(iv) Merging sub-arrays.

Answer:
(iii). Reducing the value of the gap size h**.**

11. The time complexity of Shell Sort in the worst case depends on:

(i) The size of the input array.

(ii) The choice of the pivot element.

(iii) The gap sequence used.

(iv) The number of recursion levels.

Answer:
(iii). The gap sequence used.

12. What is the time complexity of Shell Sort for Shell's original gap sequence in the worst case?

1. O(n)
2. O(n log n)
3. $O(n^2)$
4. O(log n)

Answer:
(iii) $O(n^2)$

13. What is the space complexity of Shell Sort?

(i) O(1)

(ii) O(n)

(iii) O(log n)

(iv) $O(n^2)$

Answer:
(i). O(1)

14. Quick Sort is based on which algorithmic paradigm?

(i) Greedy

(ii) Divide and Conquer

(iii) Dynamic Programming

(iv) Backtracking

Answer:
(ii). Divide and Conquer

15. In Quick Sort, the pivot element:

(i) Is always the smallest element in the array.

(ii) Divides the array into two parts, with smaller elements on the left and larger elements on the right.

(iii) Is the largest element in the array.

(iv) Must be the middle element of the array.

Answer:
(ii). Divides the array into two parts, with smaller elements on the left and larger elements on the right**.**

16. Which of the following can be chosen as the pivot in Quick Sort?

(i) The first element of the array.

(ii) The last element of the array.

(iii) A randomly chosen element.

(iv) All of the above.

Answer:
(iv). All of the above**.**

17. What happens to the array after partitioning in Quick Sort?

(i) The array is divided into two parts, with the pivot placed in its correct sorted position.

(ii) The array is divided into three equal parts.

(iii) All elements are swapped to reverse the array.

(iv) The array is merged back into one sorted list.

Answer:
(i). The array is divided into two parts, with the pivot placed in its correct sorted position.

18. The base case for the recursive calls in Quick Sort is:

(i) When there are no elements in the array.

(ii) When the pivot is greater than all other elements.

(iii) When the sub-array contains only one element.

(iv) When the pivot is less than all other elements.

Answer:
(iii). When the sub-array contains only one element.

19. What is the time complexity of Quick Sort in the best case?

(i) O(n)

(ii) $O(n^2)$

(iii) O(nlog n)

(iv) O(log n)

Answer:
(iii). O(n log n)

3. State True or False

1. Radix Sort is a linear sorting algorithm that processes elements digit by digit.
   Ans: True
2. Radix Sort works by directly comparing elements to determine their order.
   Ans: False
3. The time complexity of Radix Sort is dependent on the number of elements in the array and the number of digits in the largest element.
   Ans: True
4. Radix Sort can be used for both integers and strings with fixed-size keys.
   Ans: True
5. The space complexity of Radix Sort is O(n+b), where n is the number of elements and b is the base used for grouping digits.
   Ans: True
6. Radix Sort is more efficient than comparison-based sorting algorithms when the number of digits is small compared to the number of elements.
   Ans: True
7. Radix Sort is an in-place algorithm because it does not require additional memory for sorting.
   Ans: False
8. Shell Sort is a variation of Bubble Sort.
   Ans: False
9. The main idea of Shell Sort is to allow the exchange of far apart elements.
   Ans: True

10. An array is considered h-sorted if all sublists of every h'th element are sorted.
    Ans: True
11. Shell Sort begins by making the array 1-sorted and progressively increases the gap size.
    Ans: False
12. The Shell Sort algorithm requires the list to be divided into sub-lists with equal intervals to the current gap size (h).
    Ans: True
13. The best-case time complexity of Shell Sort with efficient gap sequences is O(n log n).
    Ans: True
14. Quick Sort is based on the Divide and Conquer strategy.
    Ans: True
15. In Quick Sort, the pivot element is always chosen as the median of the array.
    Ans: False
16. After partitioning in Quick Sort, all elements smaller than the pivot are placed to its left, and all greater elements to its right.
    Ans: True
17. The base case of Quick Sort occurs when there are two elements left in the sub-array.
    Ans: False
18. The best-case time complexity of Quick Sort is O(n log n).
    Ans: True
19. The worst-case time complexity of Quick Sort is $O(n^2)$, which occurs when the pivot results in highly unbalanced partitions.
    Ans: True
20. The average-case time complexity of Quick Sort is $O(n^2)$.
    Ans: False

4. Fill in the blanks.

1. Quick Sort is a sorting algorithm based on the _____ and _____ approach that picks an element as a pivot.
   (Answer: Divide and Conquer)
2. The pivot in Quick Sort can be chosen as the _____, _____, _____, or the _____ of the array.
   (Answer: first element, last element, random element, median)
3. After partitioning, all elements smaller than the pivot are placed to its _____, and all greater elements to its _____.
   (Answer: left, right)
4. The base case for Quick Sort occurs when the sub-array contains _____ element(s), as it is already sorted.
   (Answer: one)
5. The best-case time complexity of Quick Sort occurs when the pivot divides the array into _____ parts at every step, resulting in a time complexity of _____.
   (Answer: nearly equal, O(n log n))
6. The worst-case time complexity occurs when the pivot is the _____ or the _____ element, leading to highly unbalanced partitions and a time complexity of _____.
   (Answer: smallest, largest, $O(n^2)$)

7. The average-case time complexity of Quick Sort is _____, assuming reasonably balanced partitions.
   (Answer: O(n log n))
8. Quick Sort is an _____ algorithm, meaning it requires no additional memory for the array itself.
   (Answer: in-place)
9. Shell Sort is mainly a variation of _____ sort.
   (Answer: insertion)
10. In insertion sort, we move elements only _____ position ahead, which can involve many movements when an element needs to be moved far.
    (Answer: one)
11. The idea of Shell Sort is to allow the exchange of _____ items.
    (Answer: far)
12. In Shell Sort, the array is made _____-sorted for a large value of h.
    (Answer: h)
13. The value of h is reduced until it becomes _____.
    (Answer: 1)
14. An array is said to be h-sorted if all sub-lists of every _____ element are sorted.
    (Answer: h'th)
15. The gap size, denoted as _____, is initialized at the beginning of the algorithm.
    (Answer: h)
16. The time complexity of Shell Sort heavily depends on the _____ sequence used.
    (Answer: gap)
17. The worst-case time complexity of Shell Sort with Shell's original gap sequence is _____.
    (Answer: $O(n^2)$)
18. Radix Sort is a _____ sorting algorithm that processes elements digit by digit.
    (Answer: linear)
19. Radix Sort is efficient for _____ or _____ with fixed-size keys.
    (Answer: integers, strings)
20. Rather than comparing elements directly, Radix Sort distributes the elements into _____ based on each digit's value.
    (Answer: buckets)

A Text Book on Data Structure Using C++

School of Computer Science, YCMOU, Nashik

Ch-8 Searching Techniques

Objectives

By the end of this session, learners should be able to:

1. Learn about different searching methods: Linear Search and Binary Search.
2. Understand their applications and efficiency in various scenarios.
3. Evaluate the performance of searching algorithms based on the number of comparisons required.
4. Compare the time complexity of Linear Search (O(n)) and Binary Search (O(log n)).

Finding an element in an array is called as searching. There are two types of searching

- Linear Search: Simplest type of search
- Binary search: Most popular type of search

Complexity of search algorithm is measured in terms of number of comparisons to be made to find the element. Searching operation need to be performed several times in computer programming.

8.1 Linear Search

In linear search, search element is compared with each element of array one by one starting from first element. Linear search is also called as sequential search.

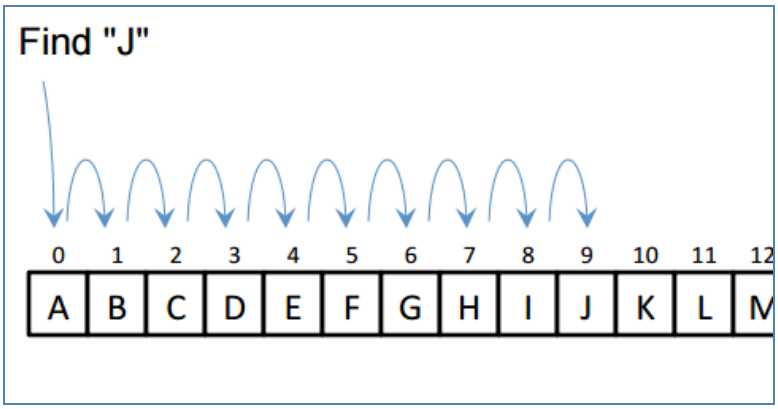


Fig 8.1: Linear Search

8.1.1 Algorithm for Linear Search

```
Linear Search ( Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to
step 8
Step 7: Print element not found
Step 8: Exit
```

8.1.2 Complexity of Linear Search

8.1.2.1 Time Complexity

1. **Best Case**:

The target element is the first element in the list. Time Complexity is O(1).

2. **Worst Case**:

The target element is the last element in the list, or it is not present in the list. Time Complexity: O(n), where **n** is the number of elements in the list.

3. **Average Case**:

The target element is located randomly in the list. On average, half of the elements need to be checked. Time Complexity is O(n).

8.1.2.2 Space Complexity

- Linear search requires no extra space aside from the input list and a few variables for iteration and comparison.
- **Space Complexity**: **O(1)**

8.1.2.3 Advantages of Linear Serach

1. Linear search works on both sorted and unsorted lists.
2. It is less efficient compared to other searching algorithms like binary search for large datasets. However, it is simple to implement and works well for small lists or unsorted data.

8.1.3 C++ Program for Linear Search

```
#include <iostream>

using namespace std;

 int linearSearch(int arr[], int n, int target) {

   for (int i = 0; i < n; i++) {

      if (arr[i] == target) {

         return i; // Element found, return its index

      }

   }

   return -1; // Element not found

}

 int main() {

   int data[] = {12, 45, 78, 23, 56, 89, 67, 34, 90};

   int n = sizeof(data) / sizeof(data[0]);

   int target = 67;

    int result = linearSearch(data, n, target);

    if (result != -1) {

      cout << "Element found at index " << result << endl;
```

```
    } else {
        cout << "Element not found in the array." << endl;
    }
     return 0;
}
```

Output:

Element found at index 6

8.1.4 Program Explanation

Initially #include statement is used to include all required header files in the program. The <iostream> is included for input-output operations. The using namespace std; allows us to use standard functions like cout without needing std::. The function linearSearch() is used to perform search operation. In the main program it is called.

8.2 Binary Search

In Binary Search a sorted array is repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half otherwise; narrow it to the upper half.

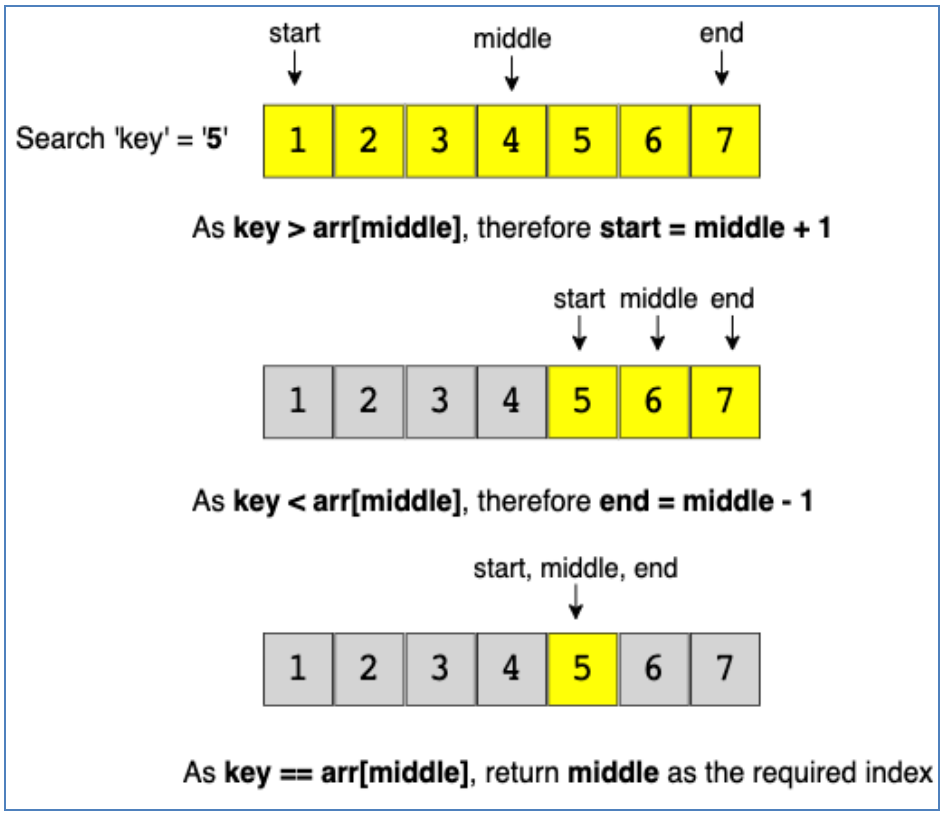


Fig 8.2: Binary Search Example

Repeatedly check until the value is found or the interval is empty. This search method has very low number of comparisons. Complexity of this algorithm is O(log n). This method is much faster than linear search.

8.2.1 Algorithm for Binary Search

1. Start with a sorted array.
2. Find the middle element:
    - If it matches the target, return its index.
    - If the target is smaller, search the left half.
    - If the target is larger, search the right half.
3. Repeat the process until the element is found or the search range is empty.

8.2.2 Complexity of Binary Search

8.2.2.1 Time Complexity

1. Best Case:

The target element is found in the middle of the array on the first comparison. Time Complexity: is O(1)

2. Worst Case:

The search space is halved repeatedly until only one element is left. For a list of size n, the number of comparisons is proportional to log n. Time Complexity is O(log n).

3. Average Case:

The target element is equally likely to be anywhere in the sorted array. The number of comparisons is still proportional to log n. Time Complexity: O(log n).

8.2.2.2 Space Complexity

1. Iterative Version:

Binary search performed iteratively does not require extra space beyond a few variables for indices and comparisons. Space Complexity is O(1).

2. Recursive Version:

The recursive version of binary search requires space for the function call stack.In the worst case, the depth of the recursion tree is $\log_2 n$. Space Complexity is O(log n)

8.2.3 Advantages of Binary Search

1. Binary search works only on sorted arrays or lists.
2. It is much faster than linear search for large datasets, especially when the array is sorted.
3. Iterative implementation is generally preferred for space efficiency, as it avoids the overhead of recursion.

8.2.4 C++ Program for Binary Search

```
// Binary Search in C++

#include <iostream>

using namespace std;

int binarySearch(int array[], int x, int low, int high) {

 // Repeat until the pointers low and high meet each other

  while (low <= high) {

    int mid = low + (high - low) / 2;

    if (x == array[mid])
```

```
    return mid;
   if (x > array[mid])
    low = mid + 1;
   else
    high = mid - 1;
  }
  return -1;
}
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int x = 4;
  int n = sizeof(array) / sizeof(array[0]);
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
   printf("Not found");
  else
   printf("Element is found at index %d", result);
}
```

Output:

Element is found at index 1

8.2.5 Program Explanation

Initially #include statement is used to include all required header files in the program. The <iostream> is included for input-output operations. The using namespace std; allows us to use cout and cin without std:: The function binarySearch() is used to perform search operation. In the main program it is called.

Summary

Searching is the process of finding an element in an array. The two main types of searching are Linear Search and Binary Search. Linear Search is a simple technique where each element is checked one by one. It works on both sorted and unsorted lists but is inefficient for large datasets with a worst-case time complexity of O(n). Binary Search is a faster method that works on sorted arrays by repeatedly dividing the search space in half. It has a time complexity of O(log n) and is more efficient for large datasets.

Questions

1. **Answer the following questions**.

    (i) What is searching operation?

    (ii) Write algorithm for linear search.

    (iii) Compute complexity of linear search algorithm.

    (iv) Write C++ program to implement linear search.

    (v) Write algorithm for binary search.

    (vi) Compute complexity of binary search algorithm.

    (vii) Write C++ program to implement binary search.

2. Multiple Choice Questions

1. What is the process of finding an element in an array called?

A. Sorting
B. Searching
C. Inserting
D. Traversing
Answer: B. Searching

2. Which of the following is NOT a type of searching?

A. Linear Search
B. Binary Search
C. Sequential Search
D. Quick Search
Answer: D. Quick Search

3. Linear search is also known as _____.

A. Random Search
B. Sequential Search

C. Binary Search
D. Fast Search
Answer: B. Sequential Search

4. What is the time complexity of linear search in the best case?

A. O(n)
B. O(log n)
C. O(1)
D. $O(n^2)$
Answer: C. O(1)

5. What is the space complexity of linear search?

A. O(n)
B. O(1)
C. O(log n)
D. $O(n^2)$
Answer: B. O(1)

6. In linear search, how many elements are checked in the worst case for an array of size nnn?

A. 1
B. n/2
C. n
D. 2n
Answer: C. n

7. Which of the following is an advantage of linear search?

A. Works only on sorted lists
B. Requires additional memory
C. Simple to implement for small lists
D. Faster than binary search for large datasets
Answer: C. Simple to implement for small lists

8. In Binary Search, the search interval is repeatedly divided in _____.

A. One-third
B. Half
C. One-fourth
D. One-fifth
Answer: B. Half

9. What is the time complexity of Binary Search in the worst case?

A. O(n)
B. O(log n)
C. $O(n^2)$
D. O(1)
Answer: B. O(log n)

10. Which of the following is the time complexity of Binary Search in the best case?

A. O(n)
B. O(log n)
C. O(1)
D. $O(n^2)$
Answer: C. O(1)

11. Which search method has a lower number of comparisons than Linear Search?

A. Sequential Search
B. Binary Search
C. Depth-First Search
D. Breadth-First Search
Answer: B. Binary Search

12. What is the space complexity of the iterative version of Binary Search?

A. O(n)
B. O(log n)
C. O(1)
D. $O(n^2)$
Answer: C. O(1)

13. What is the space complexity of the recursive version of Binary Search?

A. O(n)
B. O(log n)
C. O(1)
D. $O(n^2)$
Answer: B. O(log n)

14. Which of the following is a feature of Binary Search?

A. It works only on unsorted arrays.
B. It is faster than Linear Search for large datasets, especially when the array is unsorted.
C. It requires more space than Linear Search.
D. It works only on sorted arrays.
Answer: D. It works only on sorted arrays.

15. What is the average-case time complexity of Binary Search?

A. O(n)
B. O(1)
C. O(log n)
D. O(n^2)

3. State True or False

1. Linear search works only on sorted arrays.
   Answer**:** False

2. Binary search is considered the most popular type of search.
   Answer**:** True

3. In linear search, elements are compared one by one starting from the first element.
   Answer**:** True

4. The best-case time complexity of linear search is O(n).
   Answer**:** False (It is O(1).)

5. The worst-case time complexity of linear search is O(n).
   Answer**:** True

6. The space complexity of linear search is O(1) because it requires no extra memory apart from the input array.
   Answer**:** True

7. Linear search is less efficient than binary search for large datasets.
   Answer**:** True

8. In linear search, the average-case time complexity is O(log n).
   Answer: False (It is O(n).)

9. Binary Search repeatedly divides the search interval in half.

   Answer: True

10. The time complexity of Binary Search in the worst case is O(n).

    Answer: False (The correct time complexity is O(log n).)

11. The time complexity of Binary Search in the best case is O(log n).

    Answer: False (The correct time complexity is O(1).)

12. Binary Search is slower than Linear Search for large datasets, especially when the array is sorted.

Answer: False (Binary Search is much faster than Linear Search for large sorted datasets.)

13. Binary Search works only on unsorted arrays.

    Answer: False (Binary Search works only on sorted arrays.)

4. Fill in the blanks

1. Finding an element in an array is called as _____.
   Answer**:** searching
2. There are two types of searching: _____ and _____.
   Answer**:** Linear Search, Binary Search
3. Linear search is also called as _____.
   Answer: sequential search
4. In linear search, the search element is compared with each element of the array one by one, starting from the _____ element.
   Answer: first
5. The best-case time complexity of linear search is _____.
   Answer: O(1)
6. The worst-case time complexity of linear search is _____, where nnn is the number of elements in the list.
   Answer**:** O(n)
7. The space complexity of linear search is _____ because it requires no extra memory aside from the input list and a few variables.
   Answer: O(1)
8. Linear search works on both sorted and _____ lists.
   Answer: unsorted
9. Linear search is less efficient compared to other searching algorithms like _____ for large datasets.
   Answer: binary search
10. In Binary Search, the search interval is repeatedly divided in _____.

    Answer: half

11. The time complexity of Binary Search in the worst case is _____.

Answer: O(log n)

12. The time complexity of Binary Search in the best case is _____.

    Answer: O(1)

13. Binary Search is much _____ than Linear Search for large datasets, especially when the array is sorted.

    Answer: faster

14. Binary Search works only on _____ arrays.

    Answer: sorted

15. . In Binary Search, the search _____ when the element is found.

    Answer: stops

16. The _____ version of Binary Search is generally preferred for space efficiency.

    Answer: iterative

Ch-9 Stack

Objectives

By the end of this session, learners should be able to:

1. Understand Stack Concept
2. Identify Stack Operations
3. Analyze Array Representation of Stack
4. Understand Stack Complexity
5. Explore Stack Applications

9.1 Introduction to Stack

A linear list that allows insertion and deletion operation only at one end is called as Stack. Stack is like a container where first inserted element will go to bottom. It is also called as Last In First Out (LIFO) list.

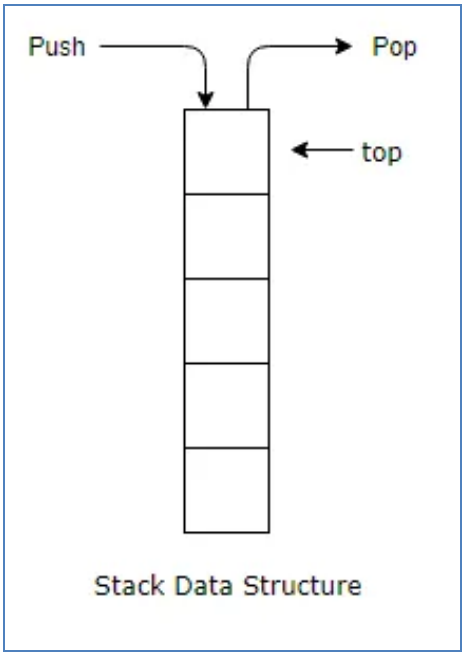


Fig 9.1 Stack

Element that is inserted in end will be taken out first. Insertion operation is called as Push and deletion operation is called as Pop. A pointer Top keep track of top element of stack.

Stack is used in many applications such as Recursion, keeping track of function calls, evaluation of expression, servicing hardware interrupts and solving problems using backtracking.

9.1.1 Array Representation of Stack

Stack data structure is not inherently supported by programming languages. Stack can be implemented by using Arrays or Linked Lists.

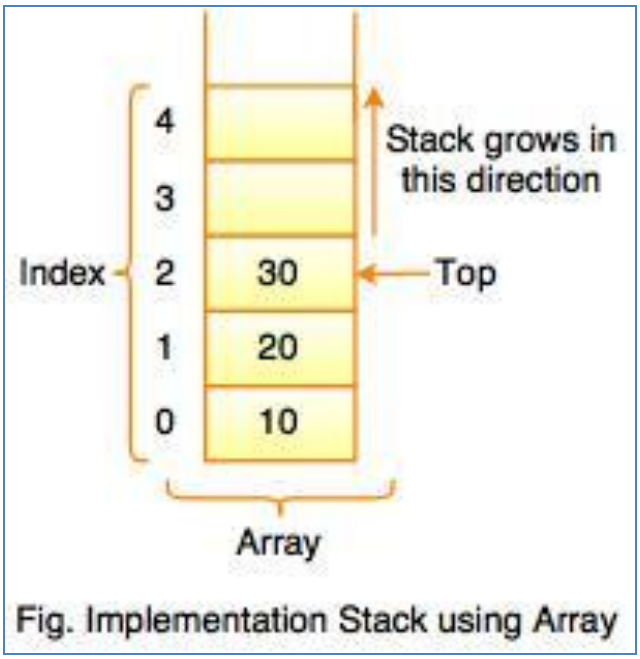


Fig 9.2: Stack Using Array

Value of index variable of an array can be assigned to Top pointer. When we insert element into stack, Top is incremented by one. When we delete element then Top is decremented by one. By using array, we can create a stack of fixed size.

9.1.2 Operations on Stack

There are two commonly performed operations on stack: Push and Pop.

9.1.2.1 Push Operation on Stack

Insertion or addition of data element into stack is called as Push Operation.

Steps to perform Push operation:

1. Check if stack is full. If yes then output an error and exit
2. If not, increment top to point to next empty space
3. Add data element to the stack location pointed by Top
4. Return success

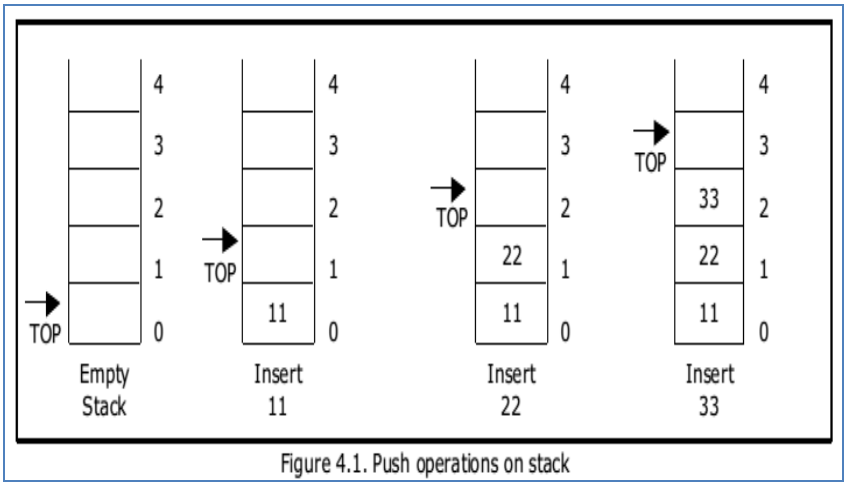


Figure 4.1. Push operations on stack

Fig 9.3: Push Operation on Stack

9.1.2.2 Pop Operation on Stack

Deletion or removing of data element from stack is called as Pop Operation.

Steps to perform Pop operation:

1. Check if stack is empty. If yes then output an error and exit
2. If not, access data element at which top is pointing
3. Decrease the value of Top by 1
4. Return success

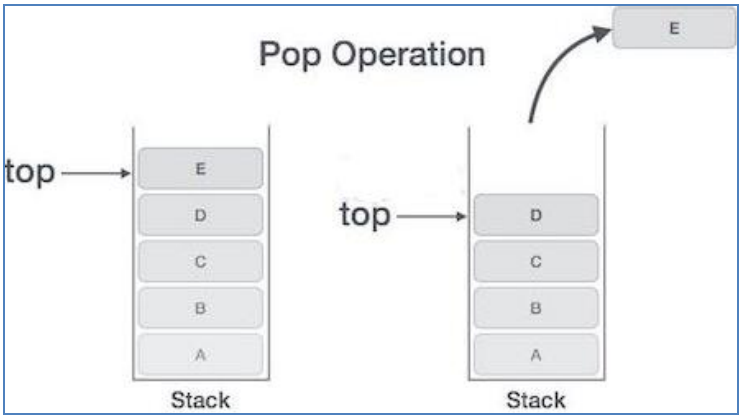


Fig 9.4: Pop Operation on Stack

9.1.3 C++ Program to Implement Stack using Array

```
/* C++ program to implement basic stack operations */

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack {

  int top;

public:

   int a[MAX]; // Maximum size of Stack

   Stack() { top = -1; }

   bool push(int x);

   int pop();

   int peek();

   bool isEmpty();

};

bool Stack::push(int x)

{

   if (top >= (MAX - 1)) {
```

```
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
```

```
    }
    else {
        int x = a[top];
        return x;
    }
}
bool Stack::isEmpty()
{
    return (top < 0);
}
int main() //Main Program
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";
        //print top element of stack after popping
    cout << "Top element is : " << s.peek() << endl;
        //print all elements in stack :
    cout <<"Elements present in stack : ";
    while(!s.isEmpty())
    {
        // print top element in stack
        cout << s.peek() <<" ";
```

```
        // remove top element in stack
        s.pop();
    }
    return 0;
}
```

Output

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is : 20
Elements present in stack : 20 10
```

Observe that in above program the #include bits/stdc++.h is a header file in C++ that includes all standard libraries. The #include keyword instructs the C++ compiler to process the contents of the specified header file during compilation.

9.2 Applications of Stack

Stack can be used in several applications such as reversing a list, polish notations, conversion of infix to postfix expression, evaluation of postfix expression, converting an infix into prefix expression, evaluation of prefix expression and recursion. In this chapter we will discuss reversing a list and other applications will be discussed later.

9.2.1 Reversing a List

We can reverse a stack in C++ using another set or any other sequential data container. We just have to reverse the order of the elements present in the stack. The below approach shows how to reverse stack using another stack.

Approach

1. Declare a temporary stack named tempStack.
2. Till the current stack is empty. Do the following:

1. Push the top of currentStack to tempStack.

2. Pop the top of the currentStack.

3. The order of the elements will be reversed in the tempStack

4. Assign the tempStack to currentStack.

9.2.2 C++ Program to reverse a list

```
// C++ Program to illustrate how we can reverse a stack
#include <iostream>
#include <stack>
using namespace std;
// utility function to print stack
void printStack(stack<int> st)
{
   while (!st.empty()) {
      cout << st.top() << " ";
      st.pop();
   }
   cout << endl;
}
int main()
{
   // Initialize a stack
   stack<int> currentStack;
   // Push some elements into the stack
   currentStack.push(1);
   currentStack.push(2);
   currentStack.push(3);
```

```
    currentStack.push(4);
    currentStack.push(5);
    // Print the original stack
    cout << "Original stack: ";
    printStack(currentStack);
    // Reverse the stack
    stack<int> tempStack;
    while (!currentStack.empty()) {
        tempStack.push(currentStack.top());
        currentStack.pop();
    }
    currentStack = tempStack;
    // Print the reversed stack
    cout << "Reversed stack: ";
    printStack(currentStack);
    return 0;
}
```

Output:

Original stack: 5 4 3 2 1

Reversed stack: 1 2 3 4 5

Observe that the content of stack is reversed.

9.2.3 Explanation of C++ Program

The code #include<iostream> allows us to use input/output functions in C++.

The code #include <stack> using namespace std; in C++ means that you can use the stack functions and classes without calling them, and you can use names from the standard library for variables and objects.

The function printStack() is used to print content of stack. In the main program elements 1,2,3,4,5 are pushed into stack. This original stack is printed. Content of stack are reversed by using tempStack. Further the reversed content is printed.

Summary

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning that the last element added is the first to be removed. The two fundamental operations of a stack are Push (insertion) and Pop (deletion). Stacks are commonly implemented using arrays or linked lists, with a Top pointer keeping track of the last inserted element. The C++ implementation of a stack involves defining a class with functions for push, pop, peek, and checking if the stack is empty. Stacks are widely used in applications like recursion, function call management, expression evaluation, and backtracking.

Questions

1. **Answer the following**

   (i) What is stack?

   (ii) How to implement stack using arrays?

   (iii) How to perform push and pop operations on stack?

   (iv) State applications of stack.

   (v) Write C++ program for stack implementation.

   (vi) Write C++ program for push and pop operation on stack.

   (vii) Write C++ program to reverse a list.

2. **Multiple Choice Questions**

1. What is a Stack?

A. A linear list where insertion and deletion are performed at both ends
B. A linear list where insertion and deletion are performed only at one end
C. A linear list where elements are arranged in sorted order
D. A non-linear data structure

Answer: B

2. What is another name for a Stack?

A. FIFO (First In First Out)
B. LIFO (Last In First Out)
C. FILO (First In Last Out)
D. None of the above

Answer: B LIFO

3. What is the operation called when an element is added to a Stack?

A. Insert
B. Enqueue
C. Push
D. Pop

Answer: C Push

4. Which operation removes the top element from the Stack?

A. Delete
B. Dequeue
C. Push
D. Pop

Answer: D Pop

5. What is the role of the Top pointer in a Stack?

A. Points to the bottom element of the stack
B. Points to the middle element of the stack
C. Points to the topmost element of the stack
D. Points to the next element to be inserted

Answer: C

6. How can a Stack be implemented in programming?

A. Using only arrays
B. Using only linked lists
C. Using arrays or linked lists
D. Using queues

Answer: C

7. What happens to the Top pointer during a Push operation?

A. It remains unchanged
B. It is decremented by one

C. It is incremented by one
D. It is set to zero

Answer: C

8. What should be checked before performing a Push operation on a Stack?

A. If the stack is empty
B. If the stack is full
C. If the stack is sorted
D. If the stack has a middle element

Answer: B

9. What is the first step in the Pop operation?

A. Decrease the value of the Top pointer by one
B. Access the element at the current Top pointer
C. Check if the stack is empty
D. Add data to the stack

Answer: C

10. Which of the following is NOT an application of a Stack?

A. Recursion
B. Evaluation of expressions
C. Reversing a list
D. Sorting data using merge sort

Answer: D

11. In reversing a stack, what happens after all elements are pushed to the temporary stack?

A. The elements are sorted
B. The temporary stack becomes the reversed stack
C. The elements are moved back to the original stack in reverse order
D. The stack is emptied

Answer: C

3. State True or False

1. A stack allows insertion and deletion operations only at one end.
   Ans: True
2. A stack follows the FIFO (First In First Out) principle.
   Ans: False (It follows LIFO: Last In First Out)

3. The operation of adding an element to a stack is called Push.
   Ans: True
4. The Top pointer in a stack points to the bottommost element.
   Ans: False (It points to the topmost element)
5. A stack can be implemented using either arrays or linked lists.
   Ans: True
6. During a Push operation, the Top pointer is decremented by one.
   Ans: False (It is incremented by one)
7. Before performing a Pop operation, we must check if the stack is empty.
   Ans: True
8. A stack can be used for evaluating expressions and reversing a list.
   Ans: True
9. The #include <bits/stdc++.h> header file in C++ includes all standard libraries.
   Ans: True
10. In a Pop operation, if the stack is empty, an error message is displayed.
    Ans: True

5. **Fill in the blanks**.

1. A stack is a linear list that allows insertion and deletion operations only at _____ end.
   Answer: one
2. A stack follows the _____ principle.
   Answer: Last In First Out (LIFO)
3. The operation of adding an element to a stack is called _____, and the operation of removing an element is called _____.
   Answer: Push, Pop
4. The pointer _____ keeps track of the top element in a stack.
   Answer: Top
5. A stack can be implemented using _____ or _____.
   Answer: arrays, linked lists
6. During a Push operation, the Top pointer is _____ by one, and during a Pop operation, it is _____ by one.
   Answer: incremented, decremented
7. Before performing a Push operation, we must check if the stack is _____, and before performing a Pop operation, we must check if the stack is _____.
   Answer: full, empty
8. The operation that returns the value of the top element without removing it is called _____.
   Answer: Peek
9. Stacks are used in applications such as recursion, evaluation of expressions, and _____ a list.
   Answer: reversing
10. If a stack is full and a Push operation is attempted, it results in _____.
    Answer: Stack Overflow

11. If a stack is empty and a Pop operation is attempted, it results in _____.
    Answer: Stack Underflow

Ch-10 Polish Notations Using Stack

Objectives

By the end of this session, learners should be able to:

1. Understand Expression Notations
2. Conversion from Infix to Postfix
3. Evaluate Postfix Expressions

10.1 Arithmetic Expression Evaluation

The stack organization is very effective in evaluating arithmetic expressions.
Expressions are usually represented in what is known as Infix notation, in which each operator is written between two operands (i.e., A - B).
With this notation, we must distinguish between (A - B)*C and A - (B * C) by using either parentheses or some operator-precedence convention.
Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. Polish notation (prefix notation) –
It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, that is, A+B is written as +AB.

2. Reverse Polish notation (postfix notation) –
It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, that is, A+B is written as AB+.

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

Expression	Prefix Notation	Postfix Notation
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +

10.2 Infix to Postfix Conversion

The steps to implement the infix to postfix conversion are given below:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
    - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
    - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

10.2.1 C++ Program

```
#include <bits/stdc++.h>

using namespace std;

// Function to return precedence of operators

int prec(char c) {

  if (c == '^')

    return 3;

  else if (c == '/' || c == '*')

    return 2;

  else if (c == '+' || c == '-')
```

```
        return 1;
    else
        return -1;
}
// The main function to convert infix expression to postfix expression
void infixToPostfix(string s) {
    stack<char> st;
    string result;
    for (int i = 0; i < s.length(); i++) {
        char c = s[i];
        // If the scanned character is an operand, add it to the output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;
        // If the scanned character is an '(', push it to the stack.
        else if (c == '(')
            st.push('(');
        // If the scanned character is an ')', pop and add to the output string from the stack
        // until an '(' is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }
```

```
        // If an operator is scanned
        else {
            while (!st.empty() && prec(c) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }
    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }
    cout << result << endl;
}
int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output:

```
abcd^e-fgh*+^*+i-
```

10.2.2 Program Explanation

The code #include bits/stdc++.h using namespace std means that the program includes all standard libraries and allows the use of names for variables and objects from the standard library. The function

prec() is used to return precedence of operators. For exponentiation operator it returns 3, for / and * it returns 2 and for + and – it returns 1. For all other symbols it returns -1.

Program further scan the infix expression from left to right. If the scanned character is an operand, put it in the postfix expression. Otherwise, if the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.

10.3 Evaluation of postfix expression

Follow the steps mentioned below to evaluate postfix expression using stack:

1. Create a stack to store operands or values.
2. Scan the given expression from left to right and do the following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
3. When the expression is ended, the number in the stack is the final answer.

10.3.1 C++ Program for Evaluation of Postfix Expression

```
// C++ program to evaluate value of a postfix expression

#include <bits/stdc++.h>

using namespace std;

// The main function that returns value of a given postfix expression

int evaluatePostfix(string exp)

{

        // Create a stack of capacity equal to expression size

        stack<int> st;

        // Scan all characters one by one

        for (int i = 0; i < exp.size(); ++i) {

        // If the scanned character is an operand (number here), push it to the stack.

                if (isdigit(exp[i]))
```

```
                st.push(exp[i] - '0');
// If the scanned character is an operator,  pop two elements from stack apply the Operator
            else {
                int val1 = st.top();
                st.pop();
                int val2 = st.top();
                st.pop();
                switch (exp[i]) {
                case '+':
                    st.push(val2 + val1);
                    break;
                case '-':
                    st.push(val2 - val1);
                    break;
                case '*':
                    st.push(val2 * val1);
                    break;
                case '/':
                    st.push(val2 / val1);
                    break;
                }
            }
        }
        return st.top();
}
```

```
// Main Code

int main()

{

        string exp = "231*+9-";

        // Function call

        cout << "postfix evaluation: " << evaluatePostfix(exp);

        return 0;

}
```

Output:

postfix evaluation: -4

10.3.2 Program Explanation

In the program first of all the required header files are included. Further a function is written to evaluate postfix expression to which a postfix expression string is passed as an argument. An expression size stack is created and elements are pushed into stack. If element is operator then two elements are popped off stack and operation is performed. When the expression ends, the number on top of stack is answer. In above example, our expression is 231*+9-. It is evaluated as: (2+3*1)-9 = -4.

Summary

Arithmetic expressions can be represented in infix, prefix (Polish notation), or postfix (Reverse Polish notation). Infix notation requires operator precedence rules or parentheses to determine the correct order of operations, while prefix and postfix notations eliminate this ambiguity. Stack-based evaluation is particularly effective for postfix expressions. To convert an infix expression to postfix, a stack is used to handle operator precedence and parentheses. The process involves scanning the expression, pushing operands directly into the output, and managing operators based on their precedence. For postfix expression evaluation, a stack stores operands. Operators pop values from the stack, perform calculations, and push the result back. The final value on the stack is the evaluated result.

Questions

1. **Answer the following**

    (i) What are polish notations?

    (ii) How these notations are useful?

    (iii) Write prefix and postfix form of following expressions:

a. ((23+45)*4)+67

b. 55+(78*5)-56

(iv) Write steps for conversion of infix expression to postfix form.

(v) Give steps for evaluation of post fix expression.

(vi) Write C++ program for conversion of infix to postfix expression.

(vii) Write C++ program for evaluation of postfix expression.

2. **Multiple Choice Questions**.

1. What is the main advantage of postfix (reverse Polish) notation over infix notation?

A) It is easier to read.
B) It eliminates the need for parentheses to define operator precedence.
C) It requires fewer operators.
D) It allows for more operands.
Answer: B

2. In prefix (Polish) notation, the expression **(A + B) * C** is written as:

A) AB + C *
B) * + A B C
C) A B C + *
D) + * A B C
Answer: B

3. Which of the following is NOT a step in infix-to-postfix conversion?

A) Scan the infix expression from left to right.
B) Push operators directly into the postfix expression.
C) Use a stack to manage operators and parentheses.
D) Pop operators from the stack based on precedence rules.
Answer: B

4. What happens when a closing parenthesis **)** is encountered during infix-to-postfix conversion?

A) It is added directly to the postfix expression.
B) All operators are popped from the stack until an opening parenthesis (is encountered.
C) It is ignored and scanning continues.
D) It is pushed onto the stack.
Answer: B

5. What is the postfix notation for the expression **A + B * C - D**?

A) A B C * + D -
B) A + B * C - D
C) A B + C * D -
D) A B C D + * -
Answer: A

6. Which of the following expressions represents **(A + B) * (C + D)** in postfix notation?

A) A B + C D + *
B) A B C D + + *
C) A B * + C D * +
D) A + B * C + D
Answer: A

7. In the context of stacks and arithmetic expression evaluation, what is the purpose of operator precedence?

A) To determine the order of scanning an expression.
B) To establish the order in which operators are evaluated.
C) To decide which operands to use for an operation.
D) To manage the size of the stack.
Answer: B

8. Which of the following is the correct sequence of steps to evaluate a postfix expression using a stack?

A) Push all operands onto the stack; pop and apply operators.
B) Scan the expression left to right; push operands onto the stack; pop operands when an operator is encountered and apply the operation.
C) Convert the expression to infix; evaluate using operator precedence.
D) Push operators and operands alternately onto the stack and evaluate as they are encountered.
Answer: B

9. Which of the following operators has the highest precedence in arithmetic expressions?

A) Addition (+)
B) Multiplication (*)
C) Parentheses (())
D) Subtraction (-)
Answer: C

10. The infix expression **A * (B + C) / D** is converted to postfix notation as:

A) A B C + * D /
B) A * B + C / D

C) A B C * + D /
D) A B + C D * /
Answer: A

11. What is the first step in evaluating a postfix expression?

A) Scan the expression from right to left.
B) Convert it to infix notation.
C) Create a stack to store operands or values.
D) Push all operators onto the stack.
Answer: C

12. When scanning a postfix expression, what is done if the scanned element is a number?

A) It is popped from the stack.
B) It is evaluated with the top of the stack.
C) It is pushed into the stack.
D) It is ignored.
Answer: C

13. What happens when an operator is encountered during the evaluation of a postfix expression?

A) It is pushed onto the stack.
B) Operands are popped from the stack, the operation is evaluated, and the result is pushed back.
C) It is ignored until the end of the expression.
D) It is converted to infix format.
Answer: B

14. At the end of evaluating a postfix expression, what does the stack contain?

A) The final answer as the only remaining element.
B) The operators in the original expression.
C) All operands in reverse order.
D) An empty stack.
Answer: A

15. How many operands are popped from the stack when a binary operator (e.g., +, -, *, /) is encountered?

A) 1
B) 2
C) 3
D) None
Answer: B

16. Which of the following postfix expressions evaluates to 14?

A) 5 1 2 + 4 * +
B) 7 3 - 2 *
C) 10 4 2 + *
D) 6 2 3 + *
Answer: B

17. What data structure is used to evaluate postfix expressions?

A) Queue
B) Linked List
C) Stack
D) Array
Answer: C

18. For the postfix expression 5 6 2 + *, what is the result after evaluating it?

A) 30
B) 40
C) 50
D) 60
Answer: D

19. If a postfix expression has n numbers and m operators, how many elements will the stack contain at the start of the evaluation?

A) 0
B) n
C) m
D) n + m
Answer: A

20. What would happen if an invalid postfix expression is evaluated (e.g., an operator without enough operands)?

A) The program would crash.
B) An error would occur due to stack underflow.
C) The evaluation would proceed without the operator.
D) The stack would overflow.
Answer: B

3. State True or False

1. Expressions in infix notation require parentheses or operator-precedence conventions to define the order of operations.
Answer: True

2. In prefix notation, the operator is placed after its operands.
Answer: False (It is placed before its operands.)

3. In postfix notation, no parentheses are required to determine the order of operations.
Answer: True

4. Stack-organized computers are better suited for infix notation than postfix notation.
Answer: False (They are better suited for postfix notation.)

5. The precedence of operators must be considered when converting an infix expression to postfix notation.
Answer: True

6. If a scanned character in an infix expression is an operand, it is added directly to the postfix expression.
Answer: True

7. When a closing parenthesis is encountered during conversion, operators are popped from the stack until an opening parenthesis is encountered.
Answer: True

8. After scanning the infix expression, any remaining operators in the stack are discarded.
Answer: False (They are popped and added to the postfix expression.)

9. Parentheses are not required in infix notation to maintain operator precedence.
Answer: False (Parentheses are required to explicitly define precedence.)

10. The order of operands and operators in infix notation uniquely determines the evaluation order.
Answer: False (Parentheses or operator-precedence conventions are required.)

11. A stack is used to store operands or values during the evaluation of a postfix expression.
Answer: True

12. The postfix expression is scanned from right to left.
Answer: False (It is scanned from left to right.)

13. If the scanned element is a number, it is pushed into the stack.
Answer: True

14. When an operator is encountered, operands are pushed onto the stack.
Answer: False (Operands are popped from the stack, the operation is evaluated, and the result is pushed back.)

15. At the end of the evaluation, the stack contains multiple results corresponding to different parts of the expression.
Answer: False (The stack contains only the final answer.)

16. Only one operand is popped from the stack for each binary operator during evaluation.
Answer: False (Two operands are popped for each binary operator.)

17. The result of evaluating a postfix expression is stored in the stack as the only remaining element after the process is complete.
Answer: True

18. The process of evaluating a postfix expression requires converting it to infix notation first.
Answer: False (Postfix expressions are evaluated directly without conversion.)

19. An operator without sufficient operands in the stack will result in a stack underflow error.
Answer: True

20. Postfix expression evaluation can be implemented efficiently using a queue instead of a stack.
Answer: False (It is implemented using a stack.)

4. Fill in the blanks

1. The stack organization is very effective in evaluating _____ expressions.
   Answer: arithmetic

2. Expressions in _____ notation place operators between their two operands.
   Answer: infix

3. Polish notation, also known as _____ notation, places the operator before its operands.
   Answer: prefix

4. In postfix notation, the operator is placed _____ its operands.
   Answer: after

5. Stack-organized computers are better suited for _____ notation than traditional infix notation.
   Answer: postfix

6. During the conversion from infix to postfix notation, operands are added _____ to the postfix expression.
   Answer: directly

7. If a scanned operator has _____ precedence than the operator on top of the stack, it is pushed onto the stack.
   Answer: higher

8. When a closing parenthesis is encountered, operators are popped from the stack until an _____ is encountered.
   Answer: opening parenthesis

9. Once the scanning of the infix expression is complete, the remaining operators in the stack are _____ and added to the postfix expression.
   Answer: popped

10. Parentheses or _____ conventions are used to distinguish the order of operations in infix notation.
    Answer: operator-precedence

11. A _____ is used to store operands or values during the evaluation of a postfix expression.
    Answer: stack
12. The postfix expression is scanned from _____ to _____.
    Answer: left; right
13. If the scanned element is a _____, it is pushed into the stack.
    Answer: number
14. When an _____ is encountered, operands are popped from the stack, the operation is evaluated, and the result is pushed back into the stack.
    Answer: operator
15. At the end of the evaluation, the stack contains the _____ as the only remaining element.
    Answer: final answer
16. For each binary operator, _____ operands are popped from the stack for evaluation.
    Answer: two
17. If the stack does not have enough operands for an operator, it results in a _____ error.
    Answer: stack underflow
18. Postfix expressions can be evaluated _____, without converting them to infix notation.
    Answer: directly
19. The evaluation of postfix expressions is based on the use of a _____ to store intermediate results.
    Answer: stack
20. A postfix expression is processed element by element until the _____ of the expression is reached.
    Answer: end

A Text Book on Data Structure Using C++

School of Computer Science, YCMOU, Nashik

Ch-11 Infix to Prefix Using Stack

Objectives

By the end of this session, learners should be able to:

1. Learn the Conversion Process of infix to prefix.
2. Implement Infix to Prefix Conversion Using C++.
3. Evaluate Prefix Expressions Efficiently.

11.1 Infix to Prefix Conversion

To convert an infix expression to a prefix expression, we can use the stack data structure. The idea is as follows:

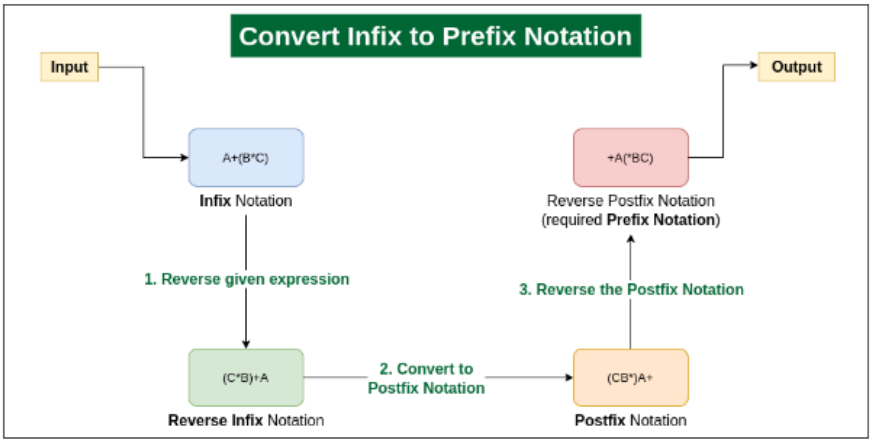
Step 1: Reverse the infix expression. Note while reversing each opening bracket '(' will become closing bracket ')' and each ')' becomes '('.

Step 2: Convert the reversed infix expression to "nearly" postfix expression.

While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.

Step 3: Reverse the postfix expression.

The stack is used to convert infix expression to postfix form.



11.1.1 C++ Program

```
// C++ program to convert infix to prefix
#include <bits/stdc++.h>
using namespace std;
// Function to check if the character is an operator
bool isOperator(char c)
{
	return (!isalpha(c) && !isdigit(c));
}
// Function to get the priority of operators
int getPriority(char C)
{
	if (C == '-' || C == '+')
		return 1;
	else if (C == '*' || C == '/')
		return 2;
	else if (C == '^')
		return 3;
	return 0;
}
// Function to convert the infix expression to postfix
string infixToPostfix(string infix)
{
	infix = '(' + infix + ')';
```

```
	int l = infix.size();
	stack<char> char_stack;
	string output;
	for (int i = 0; i < l; i++) {
// If the scanned character is an operand, add it to output.
	if (isalpha(infix[i]) || isdigit(infix[i]))
			output += infix[i];
// If the scanned character is an  ‘(‘, push it to the stack.
		else if (infix[i] == '(')
			char_stack.push('(');

	// If the scanned character is an  ‘)’, pop and output from the stack
		// until an ‘(‘ is encountered.
		else if (infix[i] == ')') {
			while (char_stack.top() != '(') {
				output += char_stack.top();
				char_stack.pop();
			}
			// Remove '(' from the stack
			char_stack.pop();
		}
		// Operator found
		else {
			if (isOperator(char_stack.top())) {
				if (infix[i] == '^') {
```

```
                                        while (
                                                getPriority(infix[i])
                                                <= getPriority(char_stack.top())) {
                                                output += char_stack.top();
                                                char_stack.pop();
                                        }
                                }
                                else {
                                        while (
                                                getPriority(infix[i])
                                                < getPriority(char_stack.top())) {
                                                output += char_stack.top();
                                                char_stack.pop();
                                        }
                                }
                                // Push current Operator on stack
                                char_stack.push(infix[i]);
                        }
                }
        }
        while (!char_stack.empty()) {
                output += char_stack.top();
                char_stack.pop();
        }
        return output;
```

```
}
// Function to convert infix to prefix notation
string infixToPrefix(string infix)
{
// Reverse String and replace ( with ) and vice versa  Get Postfix  Reverse Postfix
	int l = infix.size();
	// Reverse infix
	reverse(infix.begin(), infix.end());
	// Replace ( with ) and vice versa
	for (int i = 0; i < l; i++) {

		if (infix[i] == '(') {
			infix[i] = ')';
		}
		else if (infix[i] == ')') {
			infix[i] = '(';
		}
	}
	string prefix = infixToPostfix(infix);
	// Reverse postfix
	reverse(prefix.begin(), prefix.end());
	return prefix;
}
int main()
{
```

```
    string s = ("x+y*z/w+u");

    // Function call

    cout << infixToPrefix(s) << std::endl;

    return 0;

}
```

Output:

++x/*yzwu

11.1.2 Program Explanation

First of all the required header files are included in the program. The function isOperator is used to check if input character is operator. Further operator priorities are fixed by using function getPriority. A string function infixToPostfix is used to convert infix string to postfix by using above method.

When input string is: x+y*z/w+u then output string is: ++x/*yzwu.

11.2 Evaluation of Prefix Expression

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions. As long as we can guarantee that a valid prefix or postfix expression is used, it can be evaluated with correctness**.**

Algorithm

Step 1: Put a pointer P at the end of the string
Step 2: If character at P is an operand push it to Stack
Step 3: If the character at P is an operator pop two
    elements from the Stack. Operate on these elements
    according to the operator, and push the result
    back to the Stack
Step 4: Decrement P by 1 and go to Step 2 as long as there
    are characters left to be scanned in the expression.
Step 5: The Result is stored at the top of the Stack,
    return it
Step 6: End

Example:

Expression: +9*26

Sequence of Actions:

Character Scanned	Stack (Front to Back)	Explanation
6	6	6 is an operand, Push to stack
2	6 2	2 is an operand, Push to stack
*	12 (6*2)	* is an operator, pop 6 and 2 multiply them and push result to stack
9	12 9	9 is an operand, Push to stack
+	21 (12+9)	+ is an operator, pop 12 and 9, add them and push result to Stack

Result: 21

11.2.1 C++ Program

```
// C++ program to evaluate a prefix expression.
#include <bits/stdc++.h>
using namespace std;
bool isOperand(char c)
{
   // If the character is a digit then it must
   // be an operand
   return isdigit(c);
}

double evaluatePrefix(string exprsn)
{
   stack<double> Stack;
```

```
for (int j = exprsn.size() - 1; j >= 0; j--) {
   // Push operand to Stack  To convert exprsn[j] to digit subtract '0' from exprsn[j].
   if (isOperand(exprsn[j]))
      Stack.push(exprsn[j] - '0');
   else {
      // Operator encountered Pop two elements from Stack
      double o1 = Stack.top();
      Stack.pop();
      double o2 = Stack.top();
      Stack.pop();
      // Use switch case to operate on o1
      // and o2 and perform o1 Or o2.
      switch (exprsn[j]) {
      case '+':
         Stack.push(o1 + o2);
         break;
      case '-':
         Stack.push(o1 - o2);
         break;
      case '*':
         Stack.push(o1 * o2);
         break;
      case '/':
         Stack.push(o1 / o2);
         break;
```

```
        }
      }
    }
    return Stack.top();
}
int main()
{
    string exprsn = "+9*26";
    cout << evaluatePrefix(exprsn) << endl;
    return 0;
}
```

Output:

21

11.2.3 Program Explanation

All required header files are included in program by using #include statement. The function isOperand() is used to check if input character is operand. The function evaluatePrefix is used for prefix expression evaluation.

Summary

The conversion of infix expressions to prefix notation is an essential concept in computer science, particularly for efficient expression evaluation. Prefix notation eliminates the need for parentheses and operator precedence rules, making computations more straightforward. The conversion process involves reversing the infix expression, converting it into a modified postfix expression, and then reversing the result to obtain the prefix form. A stack-based approach is used to manage operators and operands efficiently during the conversion. A C++ program can be implemented to automate this process by considering operator precedence and using appropriate stack operations.

Questions

1. **Answer the following**

    (i) Write an algorithm to convert infix to prefix.

(ii) Write a program in C++ to convert infix expression to prefix form.

(iii) Give the steps for evaluation of prefix expression.

(iv) Write a program in C++ for evaluation of prefix expression.

2. Multiple Choice Questions

1. What is the first step in converting an infix expression to a prefix expression using a stack?
   A) Reverse the infix expression and replace brackets appropriately
   B) Convert the infix expression directly to postfix
   C) Push all operators to a stack
   D) Reverse the infix expression without changing the brackets

Answer: A

2. In step 1 of the conversion process, what happens to the brackets when reversing the infix expression?
   A) The brackets remain unchanged
   B) Opening brackets '(' become closing brackets ')' and vice versa
   C) Only opening brackets are removed
   D) Brackets are ignored during this step

Answer: B

3. Which of the following statements is **true** about step 2 in the conversion process?
   A) Operators with equal precedence are always popped from the stack
   B) Only operators with greater precedence are popped from the stack
   C) Parentheses are ignored in this step
   D) The postfix expression is directly reversed

Answer: B

4. What is the purpose of reversing the postfix expression obtained in step 3?
   A) To obtain the final prefix expression
   B) To validate the precedence of operators
   C) To check for mismatched parentheses
   D) To rearrange operators and operands

Answer: A

5. Why is a stack used in the conversion process from infix to prefix?
   A) To store the operators and operands temporarily
   B) To validate the infix expression for correctness
   C) To simplify handling operator precedence and parentheses
   D) Both A and C

Answer: D

6. Which of the following is **not** a step in the infix-to-prefix conversion process?
   A) Reverse the infix expression
   B) Convert the reversed infix expression to a "nearly postfix" form
   C) Reverse the postfix expression
   D) Directly convert infix to prefix without modification

Answer: D

7. In step 2, how does the process of converting to postfix differ from the standard postfix conversion?
   A) Operators with equal precedence are treated differently
   B) Only operators with greater precedence are popped
   C) Parentheses are not used
   D) Operands are pushed to the stack

Answer: B

8. If the infix expression is A + B * C, what would the intermediate "nearly postfix" expression look like during the process?
   A) +A*BC
   B) A B C * +
   C) * + A B C
   D) + * A B C

Answer: B

9. After reversing the postfix expression obtained from the infix expression A * (B + C), what would the final prefix expression be?
   A) *A+BC
   B) +A*BC
   C) A+BC*
   D) *+ABC

Answer: D

10. What is the role of operator precedence in the infix-to-prefix conversion process?
    A) To decide the order of operands
    B) To determine when operators are pushed or popped from the stack
    C) To reverse the expression correctly
    D) To check for balanced parentheses

Answer: B

11. Why are prefix and postfix expressions faster to evaluate compared to infix expressions?
    A) They don’t require processing brackets or operator precedence rules

B) They are always shorter than infix expressions
C) They require fewer operands
D) They have a fixed number of operators

Answer: A

12. In prefix and postfix expressions, which operator is evaluated first?
    A) The operator with the highest precedence
    B) The operator that comes first in the expression
    C) The operator with the lowest precedence
    D) The operator closest to the brackets

Answer: B

13. What is the first step in evaluating a prefix expression?
    A) Push all operands to a stack
    B) Initialize a pointer at the end of the string
    C) Check the precedence of operators
    D) Convert the expression to postfix

Answer: B

14. What should be done when the character at pointer PPP is an operand?
    A) Ignore it
    B) Push it to the stack
    C) Pop two elements from the stack
    D) Decrement the pointer

Answer: B

15. When the character at pointer P is an operator, what action should be performed?
    A) Push it to the stack
    B) Pop two elements from the stack, perform the operation, and push the result back
    C) Pop one element from the stack and push the operator back
    D) Skip the operator and move the pointer

Answer: B

16. What happens after processing a character in the prefix expression?
    A) The result is displayed immediately
    B) The pointer PPP is incremented by 1
    C) The pointer PPP is decremented by 1
    D) The stack is cleared

Answer: C

17. Where is the final result of the prefix evaluation stored?
    A) At the bottom of the stack
    B) At the top of the stack
    C) In a separate result variable
    D) In the original expression

Answer: B

18. In the example provided, what is the result of evaluating the prefix expression +9*26?
    A) 15
    B) 18
    C) 21
    D) 27

Answer: C

19. What action is taken when an operator is encountered in a prefix expression?
    A) It is pushed to the stack
    B) It is ignored if there are no operands
    C) Two elements are popped from the stack, the operation is performed, and the result is pushed back
    D) The expression is invalidated

Answer: C

3. State True or False

1. In the first step of converting an infix expression to a prefix expression, the expression is reversed, and the brackets are swapped.
   True
2. During the second step, operators with equal or higher precedence are always popped from the stack.
   False (Only operators with greater precedence are popped.)
3. The final step of the process is to reverse the postfix expression to obtain the prefix expression.
   True
4. The stack data structure is used to convert infix expressions directly to prefix without intermediate steps.
   False
5. Opening brackets '(' are replaced with closing brackets ')' during the reversal of the infix expression.
   True

6. The purpose of the stack is to manage operator precedence and handle parentheses effectively during conversion.
   True
7. In step 2, the process is the same as standard infix-to-postfix conversion.
   False (It differs by only popping operators with greater precedence.)
8. The process of reversing the postfix expression guarantees that operator precedence is preserved in the prefix expression.
   True
9. The intermediate "nearly postfix" expression is obtained by directly converting the original infix expression without any changes.
   False (The reversed infix expression is used to obtain the "nearly postfix" form.)
10. The final prefix expression is obtained by reversing the intermediate postfix expression.
    True
11. Prefix and postfix expressions are faster to evaluate because they do not require processing brackets or operator precedence rules.
    True
12. In prefix and postfix expressions, operators are evaluated based on their precedence.
    False (They are evaluated in the order they appear.)
13. The first step in evaluating a prefix expression is to initialize a pointer at the start of the string.
    False (The pointer is initialized at the end of the string.)
14. If the character at the pointer is an operand, it is pushed to the stack.
    True
15. If the character at the pointer is an operator, two elements are popped from the stack, the operation is performed, and the result is pushed back to the stack.
    True
16. After processing each character in the prefix expression, the pointer is decremented by 1.
    True
17. The final result of evaluating a prefix expression is stored at the bottom of the stack.
    False (It is stored at the top of the stack.)
18. In the given example, the result of evaluating the prefix expression +9*26 is 21.
    True
19. The stack is used to temporarily store operators during prefix expression evaluation.
    False (The stack stores operands and intermediate results.)
20. Prefix expressions do not contain brackets, making them easier to evaluate.
    True

4. Fill in the blanks

1. The first step in converting an infix expression to a prefix expression is to _____ the infix expression and swap the brackets.
   Answer: reverse

2. While reversing the infix expression, each opening bracket '(' becomes a _____ bracket ')', and vice versa.
   Answer: closing
3. The second step involves converting the reversed infix expression to a "nearly" _____ expression.
   Answer: postfix
4. During the postfix conversion step, only operators with _____ precedence are popped from the stack.
   Answer: greater
5. In the final step, the _____ expression obtained in step 2 is reversed to get the prefix expression.
   Answer: postfix
6. The _____ data structure is used to convert infix expressions to postfix and eventually to prefix.
   Answer: stack
7. The stack is primarily used to manage operator _____ and handle parentheses during the conversion process.
   Answer: precedence
8. Reversing the postfix expression ensures that the correct order of operators and operands is maintained in the _____ expression.
   Answer: prefix
9. The reversed infix expression is used to generate a "nearly" postfix expression, not the _____ infix expression.
   Answer: original
10. To convert infix to prefix, the process involves three steps: reverse the infix, convert to a nearly postfix expression, and finally _____ the postfix expression.
    Answer: reverse
11. Prefix and postfix expressions are faster to evaluate because they do not require processing _____ or operator precedence rules.
    Answer: brackets
12. In prefix and postfix expressions, the operator that comes _____ is evaluated first, regardless of its precedence.
    Answer: first
13. To evaluate a prefix expression, the pointer PPP is initialized at the _____ of the string.
    Answer: end
14. If the character at the pointer is an _____, it is pushed to the stack.
    Answer: operand
15. If the character at the pointer is an operator, _____ elements are popped from the stack, the operation is performed, and the result is pushed back to the stack.
    Answer: two
16. After processing each character, the pointer is _____ by 1.
    Answer: decremented

17. The final result of evaluating a prefix expression is stored at the _____ of the stack.
    Answer: top
18. The example prefix expression +9*26 evaluates to _____.
    Answer: 21
19. In the given example, after encountering the operator *, the operands 6 and 2 are _____ together.
    Answer: multiplied
20. Prefix expressions do not contain _____, which simplifies their evaluation.
    Answer: brackets

Ch-12 Recursion

Objectives

By the end of this session, learners should be able to:

1. Differentiate between recursive and iterative approaches to problem-solving.
2. Demonstrate recursion through examples like factorial calculation and Fibonacci sequence.
3. Implement recursion in programming languages such as C++.

12.1 Introduction

There are certain problems for which solution depends upon smaller instances of the same problem. For example, when we want to compute factorial of 5, then it can be computed by obtaining factorial of 4 as, 5 ! = 5 x 4 ! and so on.

Such problems can be solved by making use of technique called as Recursion. Recursion is a technique in which a function or procedure call itself to find solution of the problem.

To avoid recursive function to go into infinite loop, there must be at least one base criteria or condition that will stop recursion. Also, during the process of recursive calls, it must come closer to base condition. Recursion can be implemented by using stack which can hold all information needed during recursion process.

There are different types of recursion such as Linear Recursion, Binary Recursion and Multiple Recursion.

Recursion is a programming concept where a function calls itself to solve a problem. It's a powerful tool often used to solve problems that can be broken down into smaller, similar sub-problems.

12.1.1 Key Components of Recursion

1. Base Case:
    - This is the condition that stops the recursion.
    - Without a base case, the function would call itself indefinitely, leading to a stack overflow error.
2. Recursive Case:

- This is the part of the function where it calls itself to work on a smaller version of the problem.

12.1.2 How Recursion Works

When a recursive function is called:

1. It checks the base case. If the base case is met, it stops calling itself and starts returning values.
2. If the base case is not met, the function calls itself with a modified input, moving closer to the base case.

12.2 Example: Factorial Calculation

The factorial of a number nnn (written as n!n!n!) is the product of all positive integers from 1 to nnn.

Mathematical Definition:

n!=n×(n−1)!with 0!=1n! = n \times (n-1)! \quad \text{with } 0! = 1n!=n×(n−1)!with 0!=1

Recursive Code (in C++):

12.2.1 C++ Program

```
#include<iostream>

using namespace std;

int factorial(int n);

//Main Program

int main() {

 int n;

 cout << "Enter a positive integer: ";

 cin >> n;

 cout << "Factorial of " << n << " = " << factorial(n);

 return 0;

}
```

```
//Recursive Function

int factorial(int n) {

 if(n > 1)

   return n * factorial(n - 1);//Observe function is called in the same function

 else

   return 1;

}
```

For n=4:

- factorial(4)=4×factorial(3)
- factorial(3)=3×factorial(2)
- factorial(2)=2×factorial(1)
- factorial(1)=1×factorial(0)
- factorial(0)=1 (Base case)

Result:

4!=4×3×2×1=24

12.3 Applications of Recursion

Recursion technique can be used in a variety of applications as given below:

1. Mathematical Problems: Computation of Factorials, Fibonacci sequence, and Greatest Common Divisor (GCD) can be performed by using recursion.
2. Tree Traversals: The inorder, preorder, and postorder traversal of binary trees can be performed by using recursion.
3. Divide and Conquer Algorithms: The QuickSort, MergeSort, and Binary Search operations can be performed by using recursion.
4. Dynamic Programming: Solving overlapping sub problems using recursion with memorization, that is, cache results of function call.
5. Combinatorial Problems: Generating permutations, combinations, and solving puzzles like the Tower of Hanoi.

12.4 Advantages of Recursion

1. Simplifies code for problems that have repetitive substructures.
2. Naturally fits problems like tree traversal and divide-and-conquer algorithms.

12.5 Challenges with Recursion

1. Stack Overflow:
    - If the recursion depth exceeds the system's stack size, it can result in a stack overflow error.
    - Example: A recursive function without a base case.
2. Efficiency:
    - Recursive functions may have overlapping sub-problems, leading to redundant computations.
    - Solution: Use memoization or convert to an iterative approach when possible.

Tips for Writing Recursive Functions

1. Always define a base case to avoid infinite recursion.
2. Ensure the recursive case moves toward the base case.
3. For problems with overlapping sub-problems, consider dynamic programming.
4. Test with small inputs first to understand how recursion progresses.

Summary

Recursion is a programming technique where a function calls itself to solve problems that can be broken down into smaller subproblems. To prevent infinite loops, a base case is defined to stop recursion, while a recursive case ensures that the function progresses toward the base case. Recursion is widely used in solving mathematical problems such as factorial computation, Fibonacci sequence, and Greatest Common Divisor (GCD). It is also applied in tree traversals (inorder, preorder, postorder), divide-and-conquer algorithms (QuickSort, MergeSort, Binary Search), dynamic programming, and combinatorial problems like permutations and the Tower of Hanoi. While recursion simplifies complex problems, it comes with challenges like stack overflow and redundant computations. These issues can be mitigated using memoization or converting recursion to an iterative approach when necessary.

Questions

1. **Answer the following questions.**

(i) What is recursion?

(ii) How recursion works?

(iii) State applications of recursion?

(iv) State advantages of recursion.

(v) Write C++ program for generation of Fibonacci sequence.

2. **Multiple Choice Questions**

1. What is recursion in programming?

A. A technique where a function calls another function to solve a problem
B. A method to store data in a linear structure
C. A technique where a function calls itself to solve a problem
D. A way to iterate through loops efficiently

Answer: C

2. What is the role of the base case in recursion?

A. To optimize the recursion for faster execution
B. To stop the recursion and avoid infinite loops
C. To repeat the function until the problem is solved
D. To provide a stack memory for recursive calls

Answer: B

3. Which of the following is an example of a recursive formula for factorial?

A. $n!=n\times(n-1)!$ with $0!=1$
B. $n!=n+(n-1)!$ with $1!=1$
C. $n!=n\times n!= n$ with $1!=1$
D. $n!=n\div(n-1)$ with $0!=1$

Answer: A

4. Which of the following problems is best solved using recursion?

A. Finding the largest element in an array
B. Computing the Fibonacci sequence
C. Sorting an array using bubble sort
D. Calculating the average of a set of numbers

Answer: B

5. How does recursion work internally in a program?

A. It uses queues to store intermediate results
B. It uses stacks to keep track of function calls
C. It creates new memory for each function call without any structure
D. It works directly on the output without intermediate storage

Answer: B

6. What happens if a recursive function does not have a base case?

A. The program will compile successfully but produce incorrect results
B. The program will execute infinitely and eventually cause a stack overflow
C. The program will terminate immediately with no output
D. The program will optimize the recursion into a loop

Answer: B

7. Which of the following is NOT a type of recursion?

A. Linear recursion
B. Binary recursion
C. Multiple recursion
D. Dynamic recursion

Answer: D

8. What does the term "recursive case" refer to?

A. The part of the function that stops further recursion
B. The initial function call to begin recursion
C. The condition under which the function calls itself with modified input
D. A condition that prevents recursion from entering infinite loops

Answer: C

9. Which of the following statements about recursion is TRUE?

A. Recursion cannot be implemented using loops.
B. Every recursive function must have at least one base case.

C. Recursive solutions are always faster than iterative solutions.
D. Recursive functions cannot be used for mathematical problems.

Answer: B

10. What is the key advantage of using recursion?

A. It always uses less memory than iteration.
B. It simplifies solving problems that have a repetitive substructure.
C. It eliminates the need for base cases in mathematical problems.
D. It ensures faster execution for all types of problems.

Answer: B

3. State True or False

1. Recursion is a technique where a function calls itself to solve a problem.
   Answer: True
2. A base case is not necessary for recursion to work properly.
   Answer: False
3. Without a base case, a recursive function can result in a stack overflow error.
   Answer: True
4. During recursion, the function must always move closer to the base condition with each call.
   Answer: True
5. Recursion can be implemented using stacks to hold information during the recursion process.
   Answer: True
6. Binary recursion refers to problems that involve two recursive calls in each step.
   Answer: True
7. Recursive functions are always more efficient than iterative solutions.
   Answer: False
8. Linear recursion involves multiple recursive calls being made at each step.
   Answer: False
9. Recursive functions work by calling themselves indefinitely until the program terminates.
   Answer: False
10. Recursion is well-suited for problems that can be broken into smaller, similar sub-problems.
    Answer: True
11. Every recursive function must have both a base case and a recursive case.
    Answer: True
12. Recursion and iteration are completely unrelated concepts in programming.
    Answer: False
13. Factorial calculation is an example of a problem that can be solved using recursion.
    Answer: True

4. Fill in the blanks

1. Recursion is a technique where a function or procedure _____ itself to solve a problem.
   Answer: calls

2. To avoid a recursive function from going into an infinite loop, there must be at least one _____ condition.
   Answer: base
3. During the recursion process, the function must come closer to the _____ condition.
   Answer: base
4. Recursion can be implemented using a _____, which holds all the information needed during the recursion process.
   Answer: stack
5. Different types of recursion include Linear Recursion, _____ Recursion, and Multiple Recursion.
   Answer: Binary
6. The _____ case is the condition that stops the recursion.
   Answer: base
7. Without a base case, the function would call itself indefinitely, leading to a _____ error.
   Answer: stack overflow
8. The _____ case is the part of the function where it calls itself with modified input.
   Answer: recursive
9. When a recursive function is called, it first checks the _____ case to determine if it should stop.
   Answer: base
10. If the base case is not met, the function calls itself with a _____ input, moving closer to the base case.
    Answer: modified
11. The factorial of a number nnn is the product of all positive integers from _____ to nnn.
    Answer: 1
12. The mathematical formula for factorial is n!=n×(n−1)!n! = n \times (n - 1)!n!=n×(n−1)!, with _____ equal to 1.
    Answer: 0!0!0!
13. Recursion is often used to solve problems that can be broken into smaller, _____ sub-problems.
    Answer: similar
14. In recursion, the _____ case ensures that the problem reduces in size with each recursive call.
    Answer: recursive
15. Without a _____ case, a recursive function would not terminate properly.
    Answer: base

Ch-13 Queue

Objectives

By the end of this session, learners should be able to:

1. To understand the concept, properties, and applications of queues
2. To study different types of queues
3. To learn and implement basic queue operations

Queue is a linear data structure in which insertion takes place at one end and deletion takes place at other end. A new element is inserted at Rear End (Back End) and existing element is deleted from Front End.

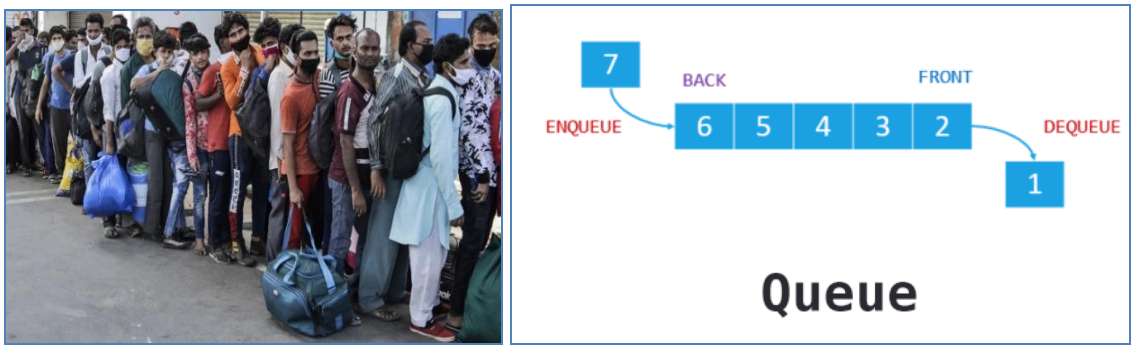


Fig 13.1: Queue

Queue is also called as First In First Out (FIFO) list. Two operations that are commonly performed on Queue:

1. Enqueue: Insertion of element at rear end
2. Dequeue: Deletion of element from front end

Queue is a very useful data structure that is used when resource is shared among multiple consumers such as CPU scheduling and Disk scheduling. It is also used in printer sharing, in OS and in networks.

13.1 Representation of Queue as an Array

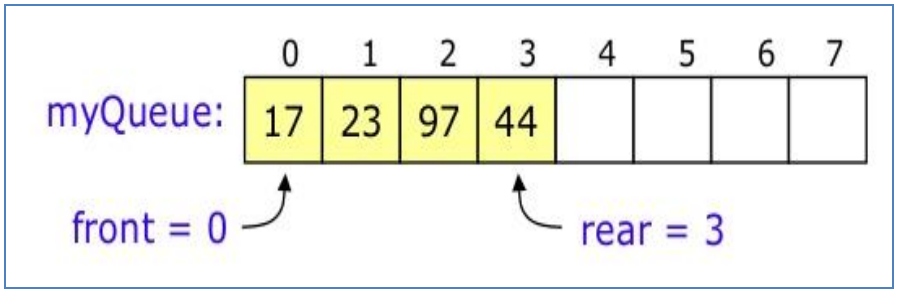


Fig 13.2: Representation of Queue as an Array

An array can be used to represent queue in memory. Front points to the element at the front of the queue (the first element to be dequeued). Rear points to the last element in the queue (the most recently enqueued element). Enqueue is used to add an element to the rear of the queue. Dequeue is used to emove an element from the front of the queue. Fig 13.2 shows representation of queue as an array

13.2 Steps for Implementation

1. **Initialize the Array**

Define an array with a fixed size capacity\text{capacity}capacity. Use front and rear pointers to track the queue's state.

2. **Enqueue**

Check if the queue is full, that is, rear is at the last index. Add the new element at the rear position and increment the rear.

3. **Dequeue**

Check if the queue is empty (front is the same as rear for an empty queue). Remove the element at the front position and increment the front.

13.2.1 C++ Program

```
// C++ Program to implement a queue using array

#include <iostream>

using namespace std;

// defining the max size of the queue
```

```
#define MAX_SIZE 100
// Implement the queue data structure
class Queue {
public:
   int front;
   int rear;
   int arr[MAX_SIZE];
   // initializing pointers in the constructor
   Queue(): front(-1), rear(-1) {}
   // Function to check if the queue is empty or not
   bool isEmpty() { return front == -1 || front > rear; }
   // Function to check if the queue is full or not
   bool isFull() { return rear == MAX_SIZE - 1; }
   // Function to get the front element of the queue
   int getFront()
   {
      if (isEmpty()) {
         cout << "Queue is empty" << endl;
         return -1;
      }
      return arr[front];
   }
   // Function to get the rear element of the queue
   int getRear()
   {
```

```
    if (isEmpty()) {
        cout << "Queue is empty" << endl;
        return -1;
    }
    return arr[rear];
}
// Function to enqueue elements from the queue
void enqueue(int val)
{
    // Check overflow condition
    if (isFull()) {
        cout << "Queue is full" << endl;
        return;
    }
    // if queue is empty, set front to 0
    if (isEmpty())
        front = 0;
    rear++;
    arr[rear] = val;
}
// Function to dequeue elements from the queue
int dequeue()
{
    // Check underflow condition
    if (isEmpty()) {
```

```
        cout << "Queue is empty" << endl;
        return -1;
    }
    int ans = arr[front];
    front++;
    // if queue becomes empty, reset both pointers
    if (isEmpty())
        front = rear = -1;
    return ans;
}

// Display function to print the queue
void display()
{
    if (isEmpty()) {
        cout << "Queue is empty" << endl;
        return;
    }
    cout << "Queue:  ";
    for (int i = front; i <= rear; i++) {
        cout << arr[i] << " ";
    }

    cout << endl;
}
```

```
};
int main()
{
    // Created Queue of size 5
    Queue q;
    // Enqueueing elements
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    // Displaying status of the queue after enqueuing
    cout << "\nAfter Enqueueing:" << endl;
    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;
    q.display();
    // Enqueueing more elements
    q.enqueue(4);
    q.enqueue(5);
    // Displaying the updated queue
    q.display();
    // Enqueueing one more element to demonstrate overflow
    // condition
    q.enqueue(6);
    // Dequeueing elements
    cout << "\nDequeueing elements:" << endl;
    cout << "Dequeued element: " << q.dequeue() << endl;
```

```
    cout << "Dequeued element: " << q.dequeue() << endl;

    // Displaying status of the queue after dequeueing

    cout << "\nAfter Dequeueing:" << endl;

    cout << "Front element: " << q.getFront() << endl;

    cout << "Rear element: " << q.getRear() << endl;

    q.display();

    return 0;

}
```

Output:

After Dequeueing:

Front element: 3

Rear element: 6

Queue: 3 4 5 6

13.2.3 Program Explanation

In above program maximum size of queue is 100 and initial front and rear pointers are set to -1. The funjction isEmpty() is used to check if queue is empty. The funjction isFull() is used to check if queue is full. The function enqueue() is used to insert element. The function dequeue() is used to delete element.

13.3 Types of Queue

There are various types of queues as given below:

Linear queue

Also known as a simple queue, this is the most basic queue structure. It follows the First-In-First-Out (FIFO) principle, where elements are added to the rear and removed from the front.

Circular queue

Also known as a circular buffer, this queue structure connects the last element to the first, forming a circular structure. When the queue is full, new elements overwrite the oldest elements in a circular fashion. Fig 13.3 shows circular queue.

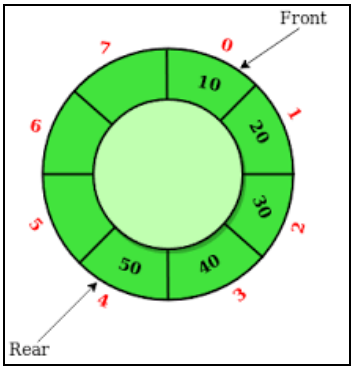


Fig 13.3: Circular Queue

Priority queue

This queue structure assigns a priority value to each element in the queue. Elements with higher priority are dequeued before elements with lower priority. Fig 13.4 shows priority queue.

Fig 13.4: Priority Queue

Double-ended queue

Also known as a deque, this queue structure supports insertion and deletion operations at both ends (front and rear). This means that it can be used as both a stack and a queue. Fig 13.4 shows dqueue.

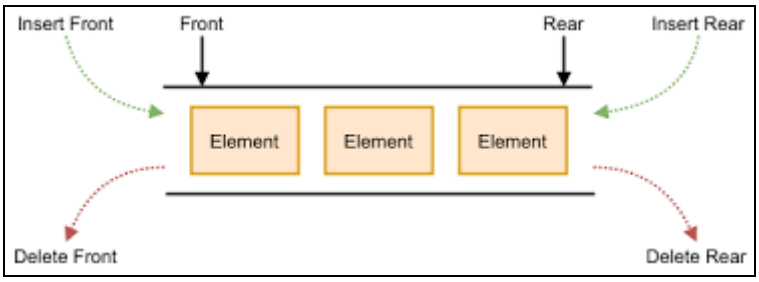


Fig 13.4: Dqueue

13.4 Applications of Queue

Queues are a fundamental data structure in computer science with a wide range of applications across various domains. Below are some common applications of queues:

1. CPU Scheduling and Task Management

Queues are used in operating systems to manage tasks like CPU scheduling, where processes are stored in a queue and executed one at a time in order of their priority or arrival.

Examples include Round-Robin Scheduling and First-Come-First-Served (FCFS).

2. I/O Buffers

Queues are used to manage input/output buffers in devices like printers, where tasks are queued and processed sequentially.

3. Data Transmission

In communication systems, queues manage data packets in networks. For example, in routers and switches, queues ensure that packets are transmitted in the correct order.

4. Breadth-First Search (BFS)

In graph traversal algorithms, such as Breadth-First Search, queues are used to explore nodes layer by layer.

5. Resource Management

Queues are used in managing resources such as printers, disk drives, or other shared resources where jobs wait in line until the resource is available.

6. Asynchronous Data Processing

Queues are integral to asynchronous communication between different parts of a system, like message queues in distributed systems and task queues in parallel programming.

7. Real-Time Systems

Real-time applications like call centers use queues to manage incoming calls and distribute them to available agents.

8. Simulation of Processes

Queues are used in simulations to model real-world systems such as queues at a bank, customer service desks, or traffic management systems.

9. Multimedia Streaming

In multimedia systems, queues help buffer audio and video streams to ensure smooth playback without interruptions.

10. Print Spooling

Printers often use queues to hold print jobs until they can be processed sequentially.

11. Scheduling in Distributed Systems

In distributed systems, task queues are used to schedule jobs across multiple processors or systems for load balancing.

12. Priority Queue Applications

A variation of queues, called priority queues, is used in scenarios like Dijkstra's algorithm for shortest path calculation and event-driven simulation.

13. Real-World Queues

Queues are also used to simulate real-world situations like customer lines at a supermarket or ticket counters.

14. Caching and Memory Management

Queues like circular queues are used in memory management systems for managing buffers and caching data.

Summary

A queue is a linear data structure based on the First-In-First-Out (FIFO) principle, where new elements are inserted at the rear and existing elements are removed from the front. The two main operations are enqueue (insertion at rear) and dequeue (deletion from front). Queues can be represented using arrays with front and rear pointers to track positions, and implemented in C++ with functions to check overflow/underflow, insert, delete, and display elements. Different types of queues include linear queues, circular queues, priority queues, and double-ended queues (deque), each serving specific purposes. Queues are widely used in applications such as CPU scheduling, disk and printer management, I/O buffering, data transmission in networks, graph traversal (BFS), resource management, asynchronous communication, real-time systems, simulations, multimedia streaming, and distributed scheduling. Their versatility makes them one of the most important data structures in computer science.

Questions

1. Answer the following questions

(i) What is queue data structure?

(ii) Explain implementation of queue using an array.

(iii) Write C++ program to implement queue.

(iv) Explain different types of queue.

(v) State applications of queue.

2. Multiple Choice Questions

1. What is a queue in data structures?

A. A linear data structure where insertion and deletion occur at the same end
B. A linear data structure where insertion takes place at one end and deletion at the other end
C. A non-linear data structure where elements are stored hierarchically
D. A data structure used only for searching algorithms

Answer: B

2. What is the key characteristic of a queue?

A. Last In First Out (LIFO)
B. First In Last Out (FILO)
C. First In First Out (FIFO)
D. Random Access

Answer: C

3. What is the process of inserting an element into a queue called?

A. Dequeue
B. Enqueue
C. Push
D. Pop

Answer: B

4. Which operation is used to remove an element from the front of the queue?

A. Enqueue
B. Dequeue
C. Insert
D. Delete

Answer: B

5. In a queue represented as an array, what does the "rear" pointer represent?

A. The first element in the queue
B. The position where the next element will be dequeued
C. The position where the next element will be enqueued
D. The last element that was dequeued

Answer: C

6. What happens during the dequeue operation in a queue?

A. An element is removed from the front of the queue
B. An element is added to the rear of the queue
C. The queue's size is doubled
D. All elements are cleared

Answer: A

7. What condition indicates that a queue is empty in an array representation?

A. Rear equals the array size
B. Front equals Rear
C. Front equals 0
D. Rear equals -1

Answer: B

8. What condition indicates that a queue is full in an array representation?

A. Rear equals the array size - 1
B. Front equals Rear
C. Rear equals 0
D. Front equals -1

Answer: A

9. Which of the following is NOT an application of a queue?

A. CPU scheduling
B. Disk scheduling
C. Depth-First Search (DFS)
D. Printer sharing

Answer: C

10. What are the two pointers used in a queue's array representation?

A. Start and End
B. Front and Rear

C. Top and Bottom
D. Head and Tail

Answer: B

11. In a queue, if the current value of Rear is 5 and an element is enqueued, what will the new value of Rear be?

A. 4
B. 5
C. 6
D. 0

Answer: C

12. What happens if you try to enqueue an element into a full queue?

A. The queue is reset to an empty state
B. An overflow error occurs
C. The element is automatically added to the front
D. The last element in the queue is replaced

Answer: B

13. How is the queue implemented in memory using an array?

A. Using two pointers: one for the middle and one for the end
B. Using two pointers: Front for deletion and Rear for insertion
C. Using one pointer to manage both insertion and deletion
D. Using three pointers for efficient management

Answer: B

3. State True or False

1. In a queue, insertion takes place at one end, and deletion takes place at the other end.
   Answer: True
2. A queue follows the Last In First Out (LIFO) principle.
   Answer: False
3. The operation of inserting an element into a queue is called Enqueue.
   Answer: True
4. Dequeue is the process of removing an element from the rear of the queue.
   Answer: False
5. In a queue represented using an array, the rear pointer indicates the position where the next element will be enqueued.
   Answer: True

6. If the front pointer equals the rear pointer, the queue is empty.
   Answer: True
7. A queue is a non-linear data structure.
   Answer: False
8. The queue is commonly used in CPU scheduling and disk scheduling.
   Answer: True
9. A queue cannot be implemented using an array.
   Answer: False
10. The base condition for a full queue in an array is when the rear pointer equals the array size minus one.
    Answer: True
11. Enqueue and Dequeue are the two primary operations performed on a queue.
    Answer: True
12. In a queue, the front pointer points to the last element in the queue.
    Answer: False
13. Printer sharing in an operating system is an example of queue application.
    Answer: True
14. During the Enqueue operation, if the rear pointer is at the last index, the queue is full.
    Answer: True
15. The first element in a queue is the last one to be removed.
    Answer: False

4. Fill in the blanks

1. Queue is a linear data structure in which _____ takes place at one end and _____ takes place at the other end.
   Answer: insertion, deletion
2. A new element is inserted at the _____ end, and an existing element is deleted from the _____ end.
   Answer: rear, front
3. Queue follows the _____ principle, where the first element inserted is the first one to be removed.
   Answer: First In First Out (FIFO)
4. The operation of inserting an element into a queue is called _____.
   Answer: Enqueue
5. The operation of removing an element from a queue is called _____.
   Answer: Dequeue
6. Queues are used in _____ scheduling and _____ scheduling.
   Answer: CPU, disk
7. Printer sharing in an operating system is an application of _____.
   Answer: queues
8. In an array representation of a queue, the _____ pointer points to the first element in the queue.
   Answer: front
9. In an array representation of a queue, the _____ pointer indicates the position where the next element will be added.
   Answer: rear

10. If the front pointer equals the rear pointer, the queue is considered _____.
    Answer: empty
11. If the rear pointer equals the array size minus one, the queue is considered _____.
    Answer: full
12. To implement a queue in memory, an _____ is defined with a fixed size capacity.
    Answer: array
13. During the Enqueue operation, if the queue is full, _____ cannot occur.
    Answer: insertion
14. During the Dequeue operation, if the queue is empty, _____ cannot occur.
    Answer: deletion
15. The process of adding an element at the rear of the queue is called _____, and removing an element from the front is called _____.
    Answer: Enqueue, Dequeue

Ch-14 Linked List

Objectives

By the end of this session, learners should be able to:

4. To understand the concept and structure of linked lists
5. To learn different types of linked lists
6. To develop the ability to represent linked lists in memory
7. To enhance problem-solving skills using dynamic data structures

Arrays are stored in successive memory locations that requires large memory space. Size of array is fixed and insertion and deletion in array is difficult. A linked list is a linear collection of data elements that are not stored in successive memory locations. Every data element in the linked list points to the next element. Fig 14.1 shows a linked list.

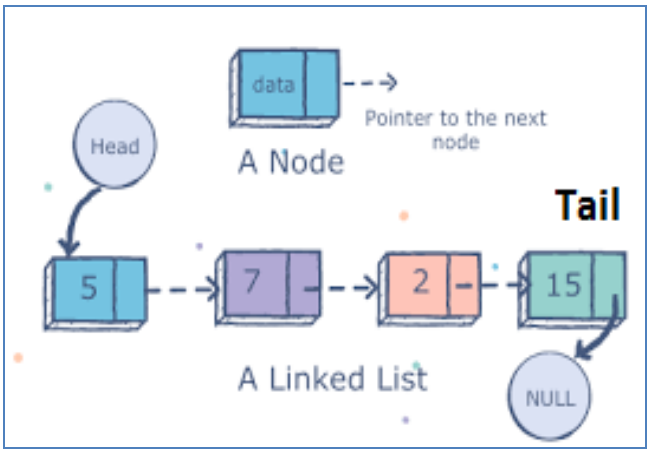


Fig 14.1: Linked List

A linked list is a collection of nodes where every node consists of data and pointer. First element is called as Head and last element is called as Tail of which pointer value is NULL. Accessing elements of linked list is much easier as each element itself contain address of next element. Memory requirement of linked list is larger than arrays. Linked list is a dynamic data structure as the list can grow during program execution.

14.1 Types of Linked List

There are different types of linked list:

1. Singly Linked List: This is a basic linked list in which each node contains data and pointer to next element. Last element pointer has NULL value. In this linked list we can traverse only in forward direction. Fig 14.2 shows a singly linked list.

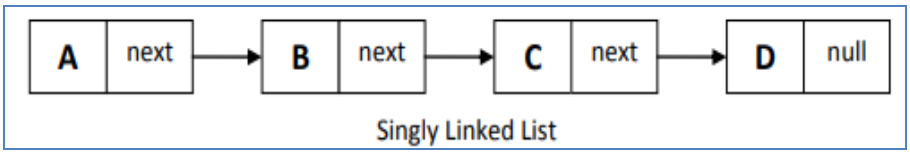


Fig 14.2: Singly Linked List

2. Circular Linked List: This is a singly linked list where last node points to first node. Such lists can be used in time sharing problems. Fig 14.3 shows a circular linked list.

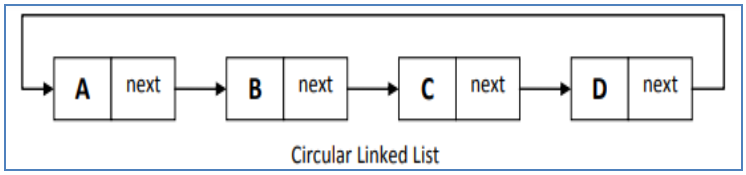


Fig 14.3: Circular Linked List

3. Doubly Linked List: In this list each node contains two pointers LPTR and RPTR. We can traverse this list in forward and reverse direction. Fig 14.4 shows a doubly linked list.

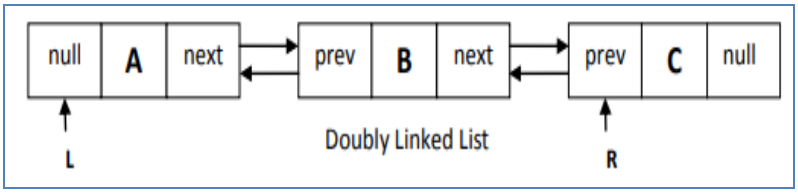
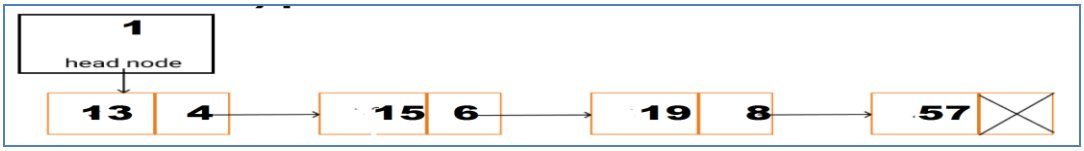


Fig 14.4: Doubly Linked List

14.2 Representations in Memory

Elements of Linked list do not require successive memory locations. They can be stored anywhere in memory where space is available. To maintain a linked list in memory, two arrays are required- Data and Next. First array Data will hold values and second array Next hold memory address of next element.

Consider a linked list:



It can be represented in memory as shown below in fig 14.5.

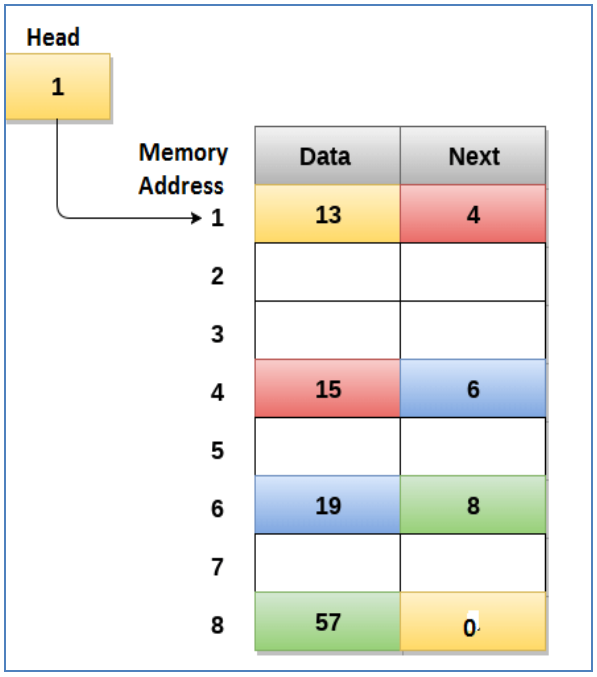


Fig 14.5: Linked List representation

14.3 Operations on Linked List

Operations that can be performed on linked list are:

- Traversing: Visiting each element of list
- Insertion and deletion: Adding and removing element from list
- Searching: Finding an element in list

14.3.1 Traversing a Linked List

Traversal of Singly Linked List is one of the fundamental operations, where we traverse or visit each node of the linked list

Algorithm

Steps:

1. We will initialize a temporary pointer to the head node of the singly linked list.

2. After that, we will check if that pointer is null or not null, if it is null, then return.
3. While the pointer is not null, we will access and print the data of the current node, then we move the pointer to next node.

14.3.2 Searching in a Linked List

Given a linked list and a **key**, the task of searching is to check if **key** is present in the linked list or not.

Algorithm

1. Initialize a node pointer, **curr = head**
2. Do following while current is not NULL
    - If the current value (i.e., **curr->key**) is equal to the key being searched return true.
    - Otherwise, move to the next node (**curr = curr->next**).
3. If the key is not found, return false

14.3.3 Insertion into a Linked List

Given a Linked List, the task is to insert a new node in this given Linked List at the following positions: At the front of the linked list, before a given node, after a given node, at a specific position or at the end of the linked list.

To insert a new node at the front, we create a new node and point its next reference to the current head of the linked list. Then, we update the head to be this new node. This operation is efficient because it only requires adjusting a few pointers.

Algorithm

1. Make the first node of Linked List linked to the new node
2. Remove the head from the original first node of Linked List
3. Make the new node as the Head of the Linked List

14.3.4 Deletion in the Linked List

Deleting a node in a Linked List is an important operation and can be done in three main ways: removing the first node, removing a node in the middle, or removing the last node.

Deletion at the Beginning of Linked List

1. Deletion at the Beginning operation involves removing the first node of the linked list.

2. To perform the deletion at the beginning of Linked List, we need to change the **head** pointer to point to the second node.

3. If the list is empty, there's nothing to delete.

Deletion at Specific Position of Linked List

1. Deletion at a specified position in a linked list involves removing a node from a specific index/position, which can be the first, middle, or last node.

2. To perform the deletion, If the position is 1, we update the head to point to the next node and delete the current head.

3. For other positions, we traverse the list to reach the node just before the specified position.

4. If the target node exists, we adjust the next of this previous node to point to next of next nodes, which will result in skipping the target node.

Deletion at the End of Linked List

Deletion at the end operation involves removing the last node of the linked list.

1. To perform the deletion at the end of Linked List, we need to traverse the list to find the second last node, then set its next pointer to null.

2. If the list is empty then there is no node to delete or has only one node then point head to null.

Summary

A linked list is a dynamic linear data structure where elements (nodes) are stored in non-contiguous memory locations, each containing data and a pointer to the next node. The first node is called the head and the last node (tail) points to NULL. Unlike arrays, linked lists allow easy insertion and deletion but require more memory due to pointers. Types include singly linked list (one-way traversal), circular linked list (last node connects to the first), and doubly linked list (two-way traversal with two pointers). Operations such as traversing, searching, insertion, and deletion are performed by manipulating node pointers, making linked lists flexible for dynamic memory management and efficient modifications compared to arrays.

Questions

1. **Answer the following questions**.

(i) What is linked list? Describe various terms associated with linked list.

(ii) Explain different types of linked lists.

(iii) Write algorithm for traversing a linked list.

(iv) Write algorithm for insertion into a linked list.

(v) Write algorithm for searching a linked list.

(vi) Write algorithm for deletion of node in a linked list.

2. Multiple Choice Questions.

1. What is a linked list?

A. A data structure where elements are stored in contiguous memory locations
B. A linear collection of data elements stored in non-successive memory locations
C. A hierarchical data structure with a parent-child relationship
D. A data structure with fixed size and static memory allocation

Answer: B

2. Which part of a node in a linked list contains the address of the next node?

A. Data
B. Pointer
C. Head
D. Tail

Answer: B

3. What is the first element of a linked list called?

A. Tail
B. Head
C. Pointer
D. Root

Answer: B

4. In a singly linked list, how many pointers does each node contain?

A. None
B. One
C. Two
D. Three

Answer: B

5. What is unique about a circular linked list compared to a singly linked list?

A. Each node contains an additional pointer
B. The last node points to the first node
C. It requires contiguous memory allocation
D. Nodes can only be traversed in reverse order

Answer: B

6. Which type of linked list allows traversal in both forward and reverse directions?

A. Singly linked list
B. Circular linked list
C. Doubly linked list
D. Static linked list

Answer: C

7. What is required to maintain a linked list in memory?

A. A single array to store all elements
B. Two arrays: one for data and another for the address of the next node
C. A stack and a queue
D. Contiguous memory allocation

Answer: B

8. Which of the following is NOT an operation on a linked list?

A. Traversing
B. Sorting
C. Insertion
D. Deletion

Answer: B

9. In a linked list, what does the "head" pointer represent?

A. The address of the last node
B. The address of the first node
C. The number of nodes in the list
D. The value of the first node

Answer: B

10. In the traversal of a singly linked list, when does the algorithm terminate?

A. When the pointer becomes NULL
B. When the pointer reaches the tail node

C. When the pointer reaches the head node
D. When all nodes are visited twice

Answer: A

11. What is the condition for searching a key in a linked list to be unsuccessful?

A. The key matches the first node
B. The key matches the last node
C. The pointer reaches NULL without finding the key
D. The pointer points to the head node

Answer: C

12. How can a new node be inserted at the beginning of a linked list?

A. Update the tail to point to the new node
B. Update the new node's pointer to point to the current head
C. Update the head to point to the last node
D. Update the new node's pointer to NULL

Answer: B

13. During the deletion of the first node in a linked list, what needs to be updated?

A. The pointer of the tail node
B. The head pointer to point to the second node
C. The next pointer of the last node
D. The data of the head node

Answer: B

14. What happens when a node is deleted at the end of the linked list?

A. The head pointer is updated
B. The last node's next pointer is set to NULL
C. The list size is doubled
D. The list becomes circular

Answer: B

15. Which type of linked list is best suited for time-sharing problems?

A. Singly linked list
B. Circular linked list
C. Doubly linked list
D. Static linked list

Answer: B

3. **State True or False**

1. Linked lists store elements in contiguous memory locations.
   Answer: False
2. In a linked list, each node contains data and a pointer to the next node.
   Answer: True
3. The first element of a linked list is called the Tail.
   Answer: False
4. The pointer of the last element in a singly linked list is NULL.
   Answer: True
5. Accessing elements in a linked list is easier because each node contains the address of the next element.
   Answer: True
6. Linked lists require less memory than arrays.
   Answer: False
7. A linked list is a dynamic data structure because its size can grow during program execution.
   Answer: True
8. In a singly linked list, nodes can be traversed in both forward and backward directions.
   Answer: False
9. A circular linked list is a type of singly linked list where the last node points to the first node.
   Answer: True
10. Circular linked lists are commonly used in time-sharing problems.
    Answer: True
11. In a doubly linked list, each node contains two pointers: LPTR and RPTR.
    Answer: True
12. A doubly linked list allows traversal in both forward and reverse directions.
    Answer: True
13. Elements of a linked list do not require successive memory locations.
    Answer: True
14. To represent a linked list in memory, only one array is sufficient.
    Answer: False
15. The operation of traversing a linked list involves visiting each node.
    Answer: True
16. Searching in a linked list involves checking if a key is present in any node of the list.
    Answer: True
17. To insert a new node at the front of a linked list, the new node's pointer is updated to point to the current head.
    Answer: True

18. Deletion at the beginning of a linked list requires updating the head pointer to point to the second node.
    Answer: True
19. Deleting a node at a specific position in a linked list requires updating the next pointer of the previous node to skip the target node.
    Answer: True
20. Deleting the last node in a linked list involves setting the last node's next pointer to NULL.
    Answer: True
21. If a linked list is empty, no node can be deleted.
    Answer: True
22. A singly linked list can have nodes arranged in a circular structure.
    Answer: True
23. The insertion operation at the front of a linked list is inefficient because it requires modifying all node pointers.
    Answer: False
24. Traversal of a singly linked list stops when the pointer becomes NULL.
    Answer: True

4. Fill in the blanks

1. In a linked list, elements are not stored in _____ memory locations.
   Answer: contiguous
2. Each node in a linked list contains _____ and a _____ to the next node.
   Answer: data, pointer
3. The first element of a linked list is called the _____, and the last element is called the _____.
   Answer: Head, Tail
4. The pointer of the last element in a singly linked list is _____.
   Answer: NULL
5. Accessing elements in a linked list is easier because each node contains the _____ of the next node.
   Answer: address
6. Linked lists require _____ memory compared to arrays because of the storage for pointers.
   Answer: larger
7. A linked list is a _____ data structure because its size can grow during program execution.
   Answer: dynamic
8. In a _____ linked list, nodes can only be traversed in the forward direction.
   Answer: singly
9. A _____ linked list is a type of singly linked list where the last node points to the first node.
   Answer: circular
10. Circular linked lists are commonly used in _____ problems.
    Answer: time-sharing

11. In a _____ linked list, each node contains two pointers: LPTR and RPTR.
    Answer: doubly
12. A doubly linked list allows traversal in both _____ and _____ directions.
    Answer: forward, reverse
13. Elements of a linked list do not require _____ memory locations and can be stored anywhere in memory.
    Answer: successive
14. To maintain a linked list in memory, two arrays are required: one for _____ and another for _____.
    Answer: data, addresses
15. The operation of visiting each element in a linked list is called _____.
    Answer: traversing
16. Searching in a linked list involves checking if a specific _____ is present in any node.
    Answer: key
17. To insert a new node at the front of a linked list, the new node's pointer is updated to point to the current _____.
    Answer: head
18. Deleting the first node of a linked list requires updating the head pointer to point to the _____ node.
    Answer: second
19. Deleting a node at a specific position requires updating the _____ pointer of the previous node to skip the target node.
    Answer: next
20. To delete the last node in a linked list, the second last node's _____ pointer is set to NULL.
    Answer: next
21. Traversal of a singly linked list stops when the pointer becomes _____.
    Answer: NULL
22. If a linked list is _____, no node can be deleted.
    Answer: empty
23. A singly linked list can have nodes arranged in a _____ structure.
    Answer: circular
24. Insertion at the front of a linked list is _____ because it only requires adjusting a few pointers.
    Answer: efficient

Ch-15 Trees

Objectives

By the end of this session, learners should be able to:

1. To understand the concept of trees as a non-linear data structure
2. To learn key tree terminologies and types
3. To study tree representations in memory
4. To apply different types of trees and their operations

A tree is a non-linear data structure. This data structure is mainly used to represent data containing hierarchical relationship between its elements. Records or table of contents can be represented in the form of trees. Fig 15.1 shows hierarchical data structure.

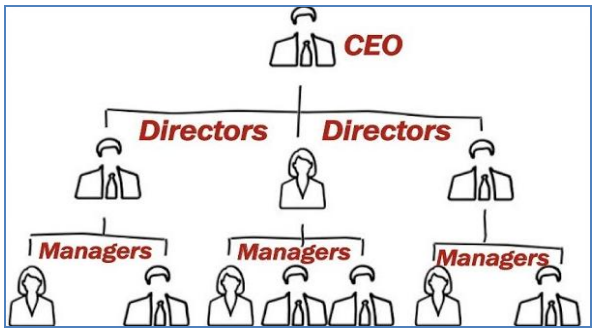


Fig 15.1: Hierarchical Data Structure

A tree is a collection of zero or more nodes with one node called as root and zero or one or more subtrees. Nodes are connected to each other by edges. Node with child is called as Parent node and node with no child is called as Leaf node. Subtree is a part of tree. There are different types of trees such as General tree, Forests, Binary tree, Binary Search tree, and Expression tree. Fig 15.2 shows structure of tree.

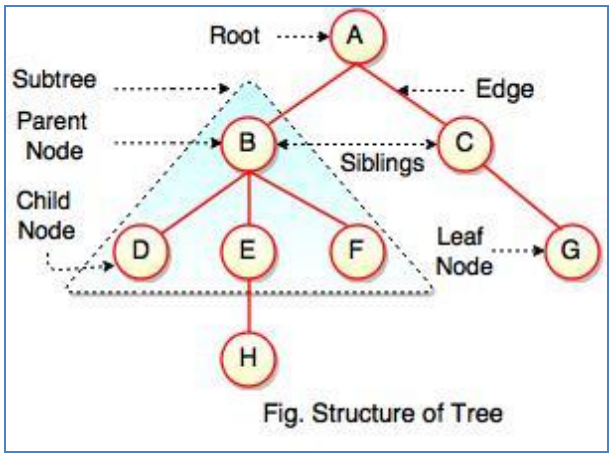


Fig 15.2: Structure of Tree

15.1 Tree Terminology

1. Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
2. Parent − Any node except the root node has one edge upward to a node called parent.
3. Child − The node below a given node connected by its edge downward is called its child node.
4. Leaf − The node which does not have any child node is called the leaf node
5. Sub-tree − Sub-tree represents the descendants of a node.
6. Traversing − Traversing means passing through nodes in a specific order.
7. Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
8. Keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

15.2 Binary Tree

A binary tree is a collection of nodes where each node has at most two children. Children are named as Left and Right child. A binary tree consists of three parts: Data, pointer to left child and pointer to right child. A binary tree with n nodes has n-1 edges. Fig 15.3 shows binary tree.

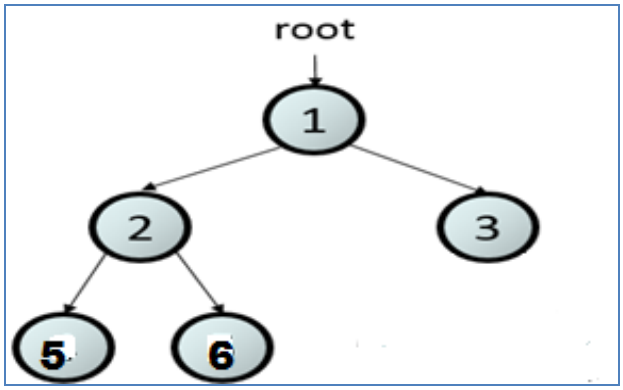


Fig 15.3: Binary Tree

Every node has one parent and two nodes are connected by single path. For height K, number of nodes are: $2^{K+1} – 1$ for K >= 0. Algebraic expression can be represented by binary tree. Binary tree can be represented in memory by linked list or array.

15.3 Representation of Binary Tree

Binary tree can be represented by linked list which is a collection of nodes. Each node consist of data, left pointer and right pointer. Left pointer holds address of left child. Right pointer holds address of right child. If any subtree of a node is empty then corresponding pointer of node will hold Null value. Fig 15.4 shows binary tree representation in linked list.

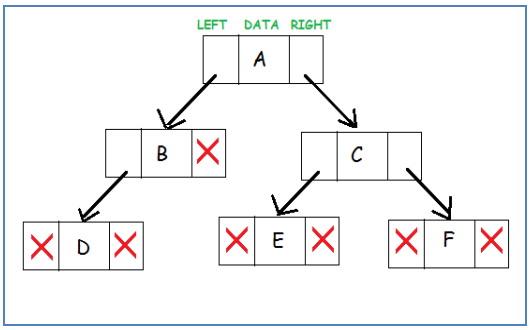


Fig 15.4: Binary Tree Representation

Binary tree can be represented by array. Root is stored at index value 0 and then in subsequent locations left and right child value is stored. For height K, size of array required is: $2^{K+1} – 1$ for K >= 0. Fig 15.5 shows binary tree representation in array.

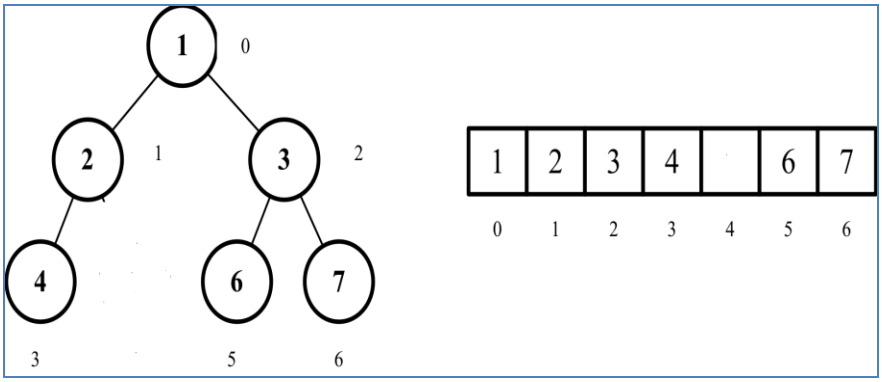


Fig 15.5: Binary Tree Representation

15.4 Types of Binary Tree

There are five types of binary tree:

1. Full Binary tree: Every node has 0 or 2 children
2. Complete Binary tree: All levels are completely filled except last level. In last level all nodes must be as left as possible.
3. Perfect Binary tree: All internal nodes has two children and all leafs are at same level.
4. Balanced Binary tree: Both right and left subtree differ by 1 level. Height of tree is O(log n). Ex: Red Black tree.
5. Degenerate Binary tree: Every internal has one child. Fig 15.6 shows different types of binary tree.

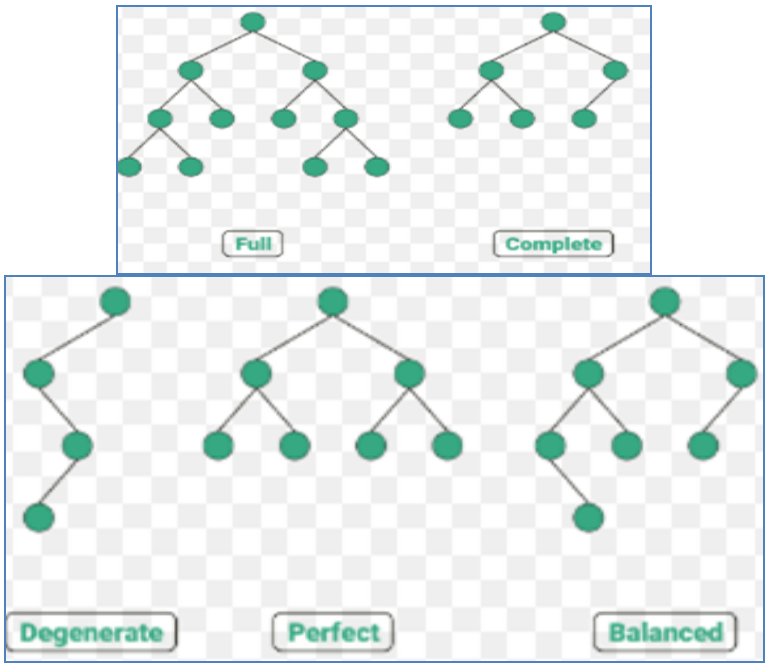


Fig 15.6: Types of Binary Tree

15.5 Traversing Operation on Binary Tree

Visiting each element of binary tree is called as traversing operation. There are three ways to perform traversing: Preorder, Postorder and Inorder.

Preorder: (NLR)

Steps:

- 1. Process root R
- 2. Traverse left subtree of R in preorder. Preorder(LST)
- 3. Traverse right subtree of R in preorder. Preorder(RST)

Example:

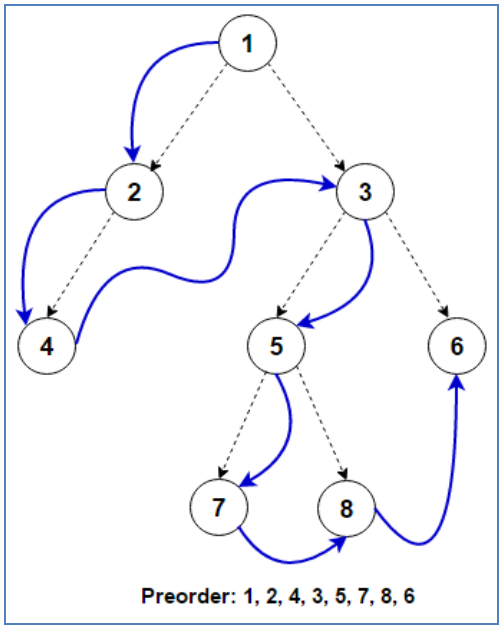


Fig 15.7: Preorder Traversal

Inorder: (LNR)

Steps:

- 1. Traverse left subtree of R in inorder.
- 2. Process root R
- 3. Traverse right subtree of R in inorder. Inorder(RST)

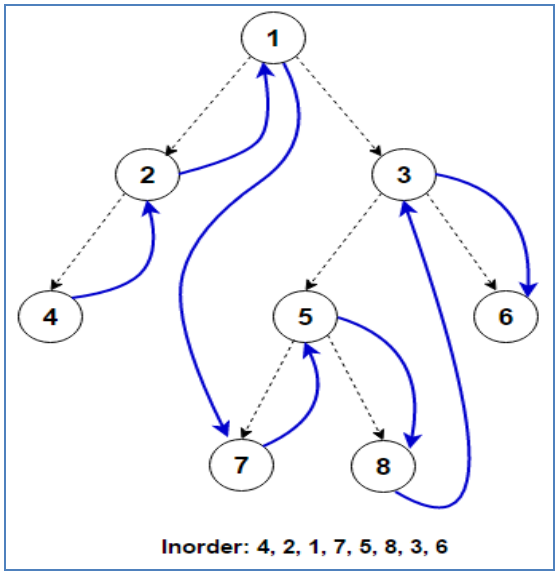


Fig 15.8: Inorder Traversal

Postorder: (LRN)

1. Traverse left subtree of R in postorder. Postorder(LST).
2. Traverse right subtree of R in postorder. Postorder(RST).
3. Process root R.

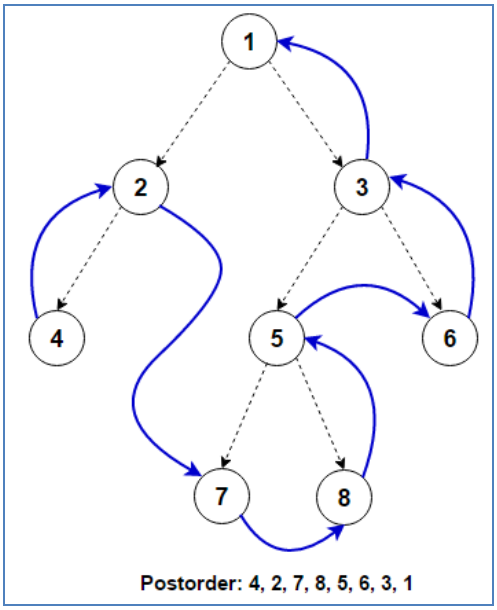


Fig 15.9: Postorder Travesal

All above algorithms are recursively defined and we need to make use of stack to implement them.

15.6 Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator. Each leaf node corresponds to the operand. For example expression tree for 3 + ((5+9)*2) would be:

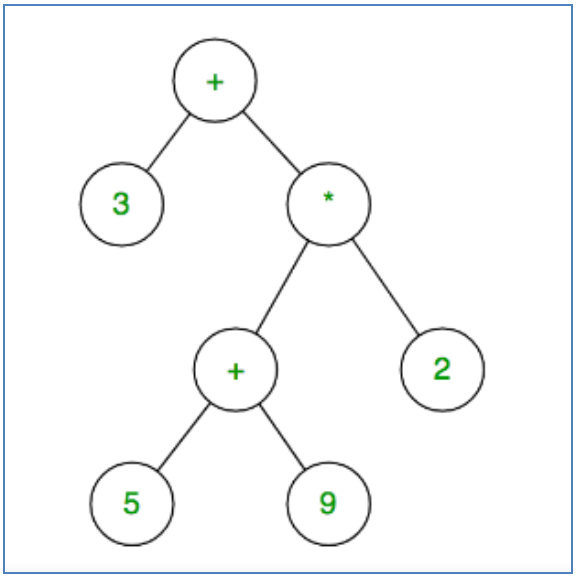


Fig 15.10: Expression Tree

Construction of Expression Tree

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.
3. In the end, the only element of the stack will be the root of an expression tree.

Summary

A tree is a non-linear hierarchical data structure consisting of nodes connected by edges, where the topmost node is called the root and the nodes with no children are called leaves. Each node may have child nodes, forming parent-child relationships that represent hierarchy. Trees are widely used to organize data for efficient searching, insertion, and deletion. A binary tree is a special type where each node has at most two children, while a binary search tree (BST) maintains elements in sorted order for faster search. Trees can be represented in memory using arrays or linked structures, and can be traversed using methods such as preorder, inorder, and postorder traversal. They are also applied in expression evaluation, decision-making, and hierarchical data representation, making them fundamental in computer science.

Questions

1. **Answer the following:**

(i) What is tree?

(ii) State different types of trees

(iii) How a tree is represented in memory.

(iv) What is a binary tree?

(v) Explain different types of binary tree.

(vi) Explain traversing operation in binary tree.

(vii) What is an expression tree?

2. Multiple choice questions

1. What type of data structure is a tree?

A) Linear
B) Non-linear
C) Sequential
D) Tabular
Answer: B) Non-linear

2. What is the topmost node of a tree called?

A) Parent
B) Child
C) Root
D) Leaf
Answer: C) Root

3. What is a node with no children called?

A) Root
B) Parent
C) Leaf
D) Sub-tree
Answer: C) Leaf

4. Which of the following is NOT a type of tree?

A) General tree
B) Binary tree
C) Sequential tree
D) Expression tree
Answer: C) Sequential tree

5. How many children can a node in a binary tree have at most?

A) 1
B) 2
C) 3
D) Unlimited
Answer: B) 2

6. Which of the following is a property of a binary search tree (BST)?

A) Left child has a value smaller than the parent
B) Right child has a value greater than the parent
C) Both A and B
D) None of the above
Answer: C) Both A and B

7. What is the formula for the number of nodes in a binary tree of height K?

A) 2K + 1
B) 2K - 1
C) 2^(K+1) - 1
D) K^2
Answer: C) 2^(K+1) - 1

8. How is a binary tree represented in memory?

A) Stack
B) Array or Linked List
C) Queue
D) Hash Table
Answer: B) Array or Linked List

9. Which type of binary tree has all internal nodes with exactly two children and all leaves at the same level?

A) Full Binary Tree
B) Perfect Binary Tree
C) Complete Binary Tree
D) Balanced Binary Tree
Answer: B) Perfect Binary Tree

10. In which type of binary tree is every internal node connected to only one child?

A) Balanced Binary Tree
B) Degenerate Binary Tree
C) Complete Binary Tree

D) Full Binary Tree
Answer: B) Degenerate Binary Tree

11. What is the order of traversal in Preorder traversal?

A) Left - Right - Root
B) Root - Left - Right
C) Left - Root - Right
D) Right - Left - Root
Answer: B) Root - Left - Right

12. Which traversal method processes the root node last?

A) Preorder
B) Inorder
C) Postorder
D) Level-order
Answer: C) Postorder

13. Which data structure is used for implementing tree traversal?

A) Queue
B) Stack
C) Heap
D) Hash Table
Answer: B) Stack

14. What is the first step when constructing an expression tree?

A) Push operands into the stack
B) Push operators into the stack
C) Pop all elements from the stack
D) Merge all nodes at once
Answer: A) Push operands into the stack

15. Which of the following operations does an expression tree NOT perform?

A) Representing mathematical expressions
B) Sorting data
C) Evaluating expressions
D) Constructing trees using operators
Answer: B) Sorting data

3. State True or False

1. A tree is a linear data structure.
   False

2. The root node is the topmost node in a tree.
   True

3. A leaf node has at least one child.
   False

4. A binary tree can have more than two children per node.
   False

5. A parent node is connected to its child node by an edge.
   True

6. A binary tree with ‘n’ nodes always has ‘n+1’ edges.
   False (It has **n-1** edges)

7. A binary search tree (BST) follows the rule that the left child contains values smaller than the root and the right child contains values greater than the root.
   True

8. A complete binary tree has all levels completely filled, except possibly the last level, which is filled from the left.
   True

9. A degenerate binary tree has all internal nodes with exactly two children.
   False (It has only one child per internal node)

10. A full binary tree has every node with either 0 or 2 children.
    True

11. In preorder traversal, the root is visited before its children.
    True

12. In postorder traversal, the root is visited first.
    False (It is visited **last**)

13. The inorder traversal sequence is: Left → Root → Right.
    True

14. A binary tree can be represented using only a stack.
    False (It can be represented using a **linked list or an array**)

15. An expression tree is a type of binary tree where internal nodes are operators and leaf nodes are operands.
    True

16. A stack is used in constructing an expression tree.
    True

17. A sub-tree consists of the descendants of a node.
    True

18. A balanced binary tree is one where the left and right subtrees differ in height by at most 2 levels.
    False (The difference is at most **1** level)

19. In a binary tree, the root node is always at level 1.
    False (It is at **level 0**)

20. The size of an array needed to represent a binary tree of height K is $\mathbf{2^{K+1} - 1}$.
    True

4. Fill in the blanks:

1. A tree is a _____ data structure.
Answer: non-linear

2. The topmost node in a tree is called the _____.
Answer: root

3. A node with no child nodes is called a _____.
Answer: leaf

4. A _____ is a part of a tree that consists of a node and its descendants.
Answer: subtree

5. In a binary tree, each node can have at most _____ children.
Answer: two

6. Nodes in a tree are connected by _____.
Answer: edges

7. A binary tree with ‘n’ nodes has _____ edges.
Answer: n - 1

8. A _____ binary tree has all levels completely filled except possibly the last level, which is filled from left to right.
Answer: complete

9. A _____ binary tree has every node with either 0 or 2 children.
Answer: full

10. A _____ binary tree has all internal nodes with exactly two children and all leaf nodes at the same level.
Answer: perfect

12. A degenerate binary tree has all internal nodes with only _____ child.
Answer: one

13. The order of traversal in **Preorder traversal** is _____ → Left → Right.
Answer: Root

14. In **Postorder traversal**, the root node is processed _____.
Answer: last

15. The order of traversal in **Inorder traversal** is _____ → Root → Right.
Answer: Left

16. Binary trees can be represented in memory using a _____ or an array.
Answer: linked list

17. The data structure used to implement **tree traversal** is a _____.
Answer: stack

18. A tree is a collection of one or more _____.
Answer: nodes

20. A node except the root has one edge upward to a node called the _____.
Answer: parent

21. A _____ is a tree where the left and right subtrees differ in height by at most one level.
Answer: balanced binary tree

22. An expression tree is a binary tree in which each internal node corresponds to an _____.
Answer: operator

23. The leaf nodes of an **expression tree** represent _____.
Answer: operands

24. To construct an expression tree, a _____ data structure is used.
Answer: stack

25. In a binary tree, the root is stored at index _____ when using an array representation.
Answer: 0

26. The level of a node represents its _____ in the tree.
Answer: generation

27. The size of an array required to represent a binary tree of height K is _____.
Answer: 2^(K+1) - 1

Ch-16 Graphs

Objectives

By the end of this session, learners should be able to:

1. To understand the concept, terminology, and types of graph
2. To learn graph representations in memory
3. To study and apply graph traversal techniques
4. To explore the applications of graphs in real-world problems

A graph is a non-linear data structure consisting of nodes and edges. Nodes are also called as vertices and edges are called as lines or arcs. A graph G = (V, E) where V is set of vertices and E is set of edges. For Ex: set of vertices can be V = {0,1,2,3,4} and set of edges E = {01,12,23,34,04,14,13}. Fig 16.1 shows Graph.

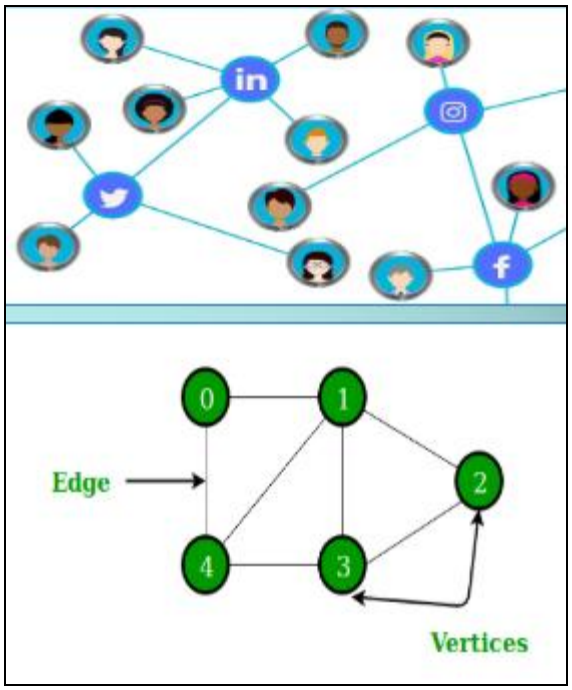


Fig 16.1: Graph

Graphs are used to solve many real life problems. They are used to represent network such as telecommunication network, circuit network. Graphs are used in social networks such as LinkedIn, Face book where each person is represented by a vertex or node. Each node is a structure containing information like id, name, gender, address etc. Google map makes use of graph to find shortest distance where locations are represented as vertices.

16.2 Graph Terminology

1. Vertex: Each node in the graph is called as vertex.
2. Edge: Path between two vertices is called as edge and it is represented by adjacent vertices.
3. Adjacency: Two vertices are adjacent if they are connected to each other by edge.

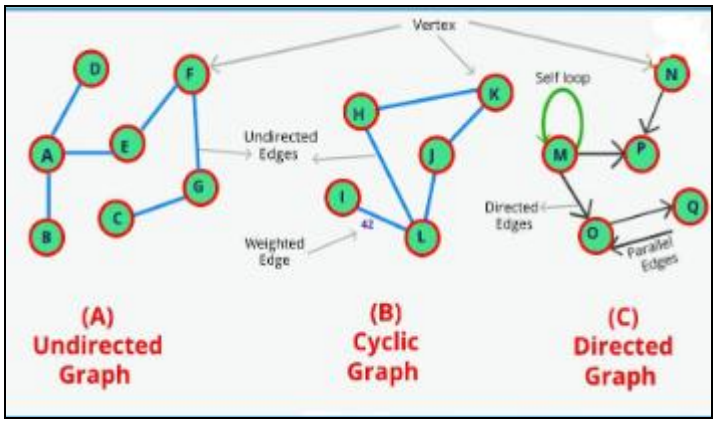


Fig 16.2: Graph Types

4. Loop: An edge of the form (u,u) is called as loop.
5. Path: A sequence of edges between two vertices is called as path.
6. Connected Graph: When there is a path from any vertex to every other vertex then it is called as connected graph.
7. Degree of Vertex: Number of incident edges is called as degree of vertex.
8. In-degree: The number of edges directed toward a vertex.
9. Out-degree: The number of edges directed away from a vertex.
10. Isolated vertex has degree 0, pendent vertex has degree 1.
11. Relation: Relations are often represented using graphs, where vertices represent elements, and edges represent relationships between them.

12. Weight: Weighted graph is a graph where each edge has an associated numerical value called a weight or cost. This weight can represent distances, costs, time, or any other measurable quantity.

13. Length: The length of a path in a graph refers to the sum of the weights of the edges along that path.

A graph can be undirected, directed and cyclic graph as shown in fog 3.

16.3 Representation of Graph

A graph can be represented in memory by two ways. In sequential representation it is represented by adjacency matrix A. In linked representation it is represented by linked list.

The adjacency matrix A for a graph G = (V, E) with n vertices is a nxn matrix, such that Aij = 1, if there is an edge from Vi to Vj and Aij = 0, if there is no edge. In linked representation graph can be represented by adjacency list. Fig 16.3 shows representation of graph.

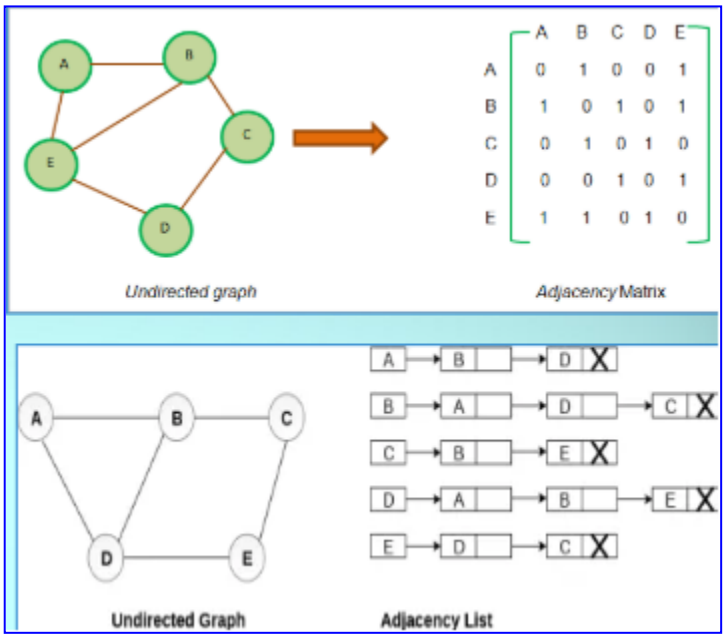


Fig 16.3: Representation of Graph

16.4 Operations on Graph

Operations that can be performed on graph are:

1. Traversing: Visiting each node of graph

2. Searching: Finding location of node N in a graph

3. Insertion: Adding node N in a graph
4. Deletion: Removing node from a graph
5. Merging of Vertices: Two or more vertices can be merged together.
6. Splitting of Vertices: One vertex can be spliced into two or more vertices.
7. Edge contraction: Edge is contracted.

16.5 Graph Traversal BFS

Breadth First Search (BFS) is a fundamental graph traversal algorithm.

It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed.

Algorithm

1. Initialization: Enqueue the given source vertex into a queue and mark it as visited.
2. Exploration: While the queue is not empty:
3. Dequeue a node from the queue and visit it (e.g., print its value).
4. For each unvisited neighbor of the dequeued node:

    Enqueue the neighbor into the queue.

    Mark the neighbor as visited.

5. Termination: Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node. Fig 16.4 shows BFS.

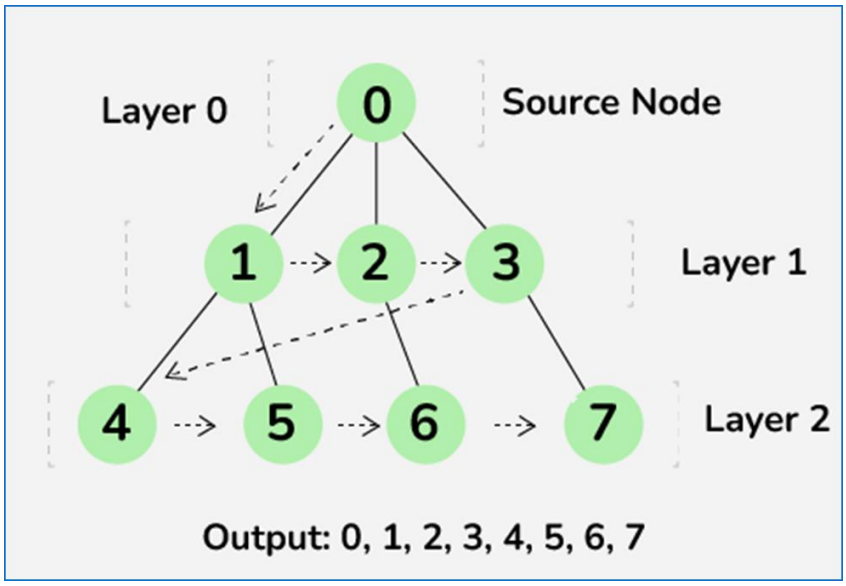


Fig 16.4: BFS

16.6 Graph Traversal DFS

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one.

When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex.

This is similar to a tree, where we first completely traverse the left sub-tree and then move to the right sub-tree.

The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array. Fig 16.5 shows DFS.

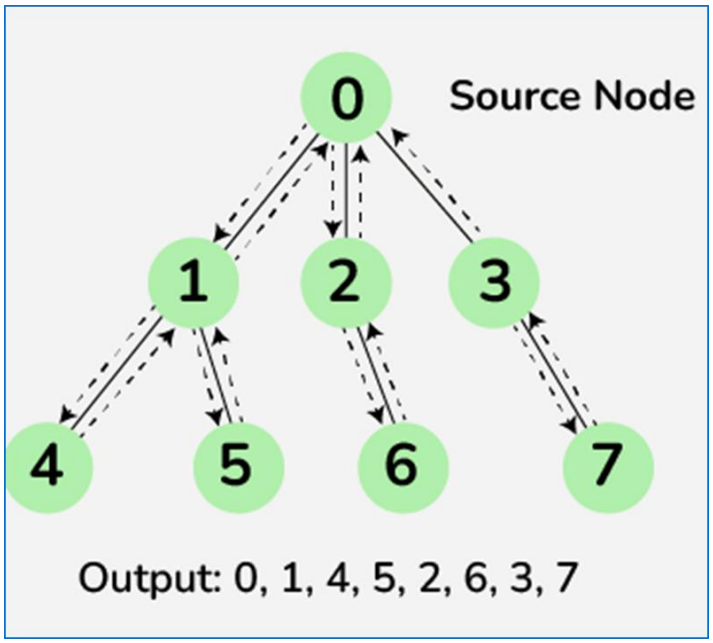


Fig 16.5: DFS

16.7 Applications of Graphs

Graphs are used in various applications as given below:

Computer Science

In Networks graphs are used in representing computer networks, where nodes are devices and edges are connections. They are also used in internet modeling, including routing protocols like Dijkstra's algorithm for shortest paths.

In data Structures graphs are used for dependency graphs in compilers for managing tasks like topological sorting.

In Artificial Intelligence they are used in state-space exploration in path finding algorithms.

In database relationships graphs are used. They are used in social Networks modeling social interactions, where nodes represent individuals; edges represent relationships such as, friendship, following.

In transportation graphs are used in route planning, shortest path algorithms in road networks, GPS systems, and public transit. They are also used in airline routes. Airports are represented as nodes, flights as edges, enabling efficient scheduling and route optimization. In traffic management graphs are used.

Summary

A graph is a non-linear data structure consisting of vertices (nodes) and edges (arcs or lines) that represent relationships between elements. Graphs can be directed, undirected, weighted, or cyclic, and are used in real-life applications such as social networks, telecommunication systems, Google Maps, and transportation networks. Graph terminology includes concepts like degree, in-degree, out-degree, loops, paths, and connectivity. Graphs can be represented in memory using adjacency matrices or adjacency lists. Fundamental operations include traversal, searching, insertion, deletion, merging, and splitting of vertices. Two primary traversal algorithms are Breadth First Search (BFS), which explores all adjacent vertices before moving deeper, and Depth First Search (DFS), which explores as far as possible along a branch before backtracking. Graphs are essential in computer science, AI, databases, and networking due to their ability to efficiently model and solve complex problems.

Questions

1. Answer the following

(i) What is graph?

(ii) Explain following terms: Vertex, edge, in degree, out degree

(iii) How graphs are represented by arrays?

(iv) How graphs are represented by linked lists?

(v) Explain DFS traversal of graph.

(vi) Explain BFS traversal of graph.

(vii) State applications of graph.

2. Multiple Choice Questions

1. What is a graph in data structures?

   a) A linear data structure consisting of nodes and edges
   b) A non-linear data structure consisting of nodes and edges
   c) A tree-based data structure
   d) A sequential data structure
   Answer: b) A non-linear data structure consisting of nodes and edges

2. In a graph, what are the nodes also called?

   a) Edges
   b) Arcs
   c) Vertices
   d) Loops
   Answer: c) Vertices

3. Which of the following is an example of a real-life application of graphs?

   a) Social networks like Facebook and LinkedIn
   b) Google Maps for shortest distance calculation
   c) Telecommunication networks
   d) All of the above
   Answer: d) All of the above

4. If a vertex has no edges connected to it, what is it called?

   a) Pendent vertex
   b) Isolated vertex
   c) Loop vertex
   d) Weighted vertex
   Answer: b) Isolated vertex

5. What is the sum of weights of the edges along a path in a graph called?

   a) Edge degree
   b) Path length
   c) Vertex weight
   d) Loop length
   Answer: b) Path length

6. Which of the following is a property of a connected graph?

   a) Every vertex is connected to every other vertex by a path
   b) There are no edges in the graph
   c) There is at least one isolated vertex
   d) No two vertices are adjacent
   Answer: a) Every vertex is connected to every other vertex by a path

7. What is an adjacency matrix used for?

   a) Representing trees in sequential form
   b) Representing graphs using a matrix
   c) Storing shortest path calculations
   d) Sorting vertices in a graph
   Answer: b) Representing graphs using a matrix

8. In which graph traversal method are all adjacent vertices visited before moving deeper?

   a) Depth First Search (DFS)
   b) Breadth First Search (BFS)
   c) Dijkstra's Algorithm
   d) Kruskal's Algorithm
   Answer: b) Breadth First Search (BFS)

9. Which data structure is used in DFS traversal of a graph?

    a) Queue
    b) Stack
    c) Heap
    d) Array
    Answer: b) Stack

10. In a weighted graph, what does the weight of an edge represent?

    a) The number of vertices connected by the edge
    b) The distance, cost, time, or any measurable quantity between vertices
    c) The degree of the edge
    d) The number of adjacent edges
    Answer: b) The distance, cost, time, or any measurable quantity between vertices

11. Which operation is not commonly performed on a graph?

    a) Traversing
    b) Searching
    c) Sorting
    d) Merging of vertices
    Answer: c) Sorting

12. What does an adjacency list store in a graph?

    a) A matrix representation of edges
    b) A list of adjacent vertices for each vertex
    c) A set of all paths in the graph
    d) A stack of vertices
    Answer: b) A list of adjacent vertices for each vertex

13. What is a loop in a graph?

    a) A sequence of edges between two different vertices
    b) An edge that connects a vertex to itself
    c) A cycle with an even number of edges
    d) A disconnected subgraph
    Answer: b) An edge that connects a vertex to itself

14. What is the main difference between DFS and BFS?

    a) DFS uses a queue, BFS uses a stack
    b) DFS explores deeper before visiting adjacent nodes, BFS explores all adjacent nodes first
    c) BFS works only on directed graphs
    d) DFS always finds the shortest path

Answer: b) DFS explores deeper before visiting adjacent nodes, BFS explores all adjacent nodes first

3. State True or False

1. A graph is a linear data structure consisting of nodes and edges.
   False

2. In a graph, nodes are also called vertices.
   True

3. Graphs cannot be used to represent social networks.
   False

4. An isolated vertex is a vertex with no edges connected to it.
   True

5. The length of a path in a weighted graph is the sum of the edge weights along that path.
   True

6. A connected graph has at least one path between every pair of vertices.
   True

7. An adjacency matrix is not used to represent graphs.
   False

8. In Depth First Search (DFS), adjacent vertices are visited before going deeper.
   False

9. A queue is used in Depth First Search (DFS).
   False

10. In a weighted graph, edge weights can represent distances, costs, or other measurable values.
    True

11. Sorting is a common operation performed on graphs.
    False

12. An adjacency list stores a list of adjacent vertices for each vertex.
    True

13. A loop in a graph is an edge that connects a vertex to itself.
    True

14. Breadth First Search (BFS) explores all adjacent vertices before going deeper.
    True

4. Fill in the blanks

1. A graph is a _____ data structure consisting of nodes and edges.
2. Nodes in a graph are also called _____.
3. The edges in a graph are also called _____ or arcs.
4. A graph is represented as G = (V, E), where V is the set of _____ and E is the set of _____.
5. An _____ vertex has no edges connected to it.
6. The sum of the edge weights along a path in a weighted graph is called the _____ of the path.
7. A _____ graph has a path between every pair of vertices.
8. An adjacency _____ is used to represent graphs in a matrix format.
9. _____ First Search (BFS) explores all adjacent vertices before going deeper.
10. In Depth First Search (DFS), _____ is used as the primary data structure.
11. In a weighted graph, each edge has an associated numerical value called a _____.
12. A _____ is an edge that connects a vertex to itself.
13. The adjacency _____ representation of a graph stores a list of adjacent vertices for each vertex.
14. Graphs are widely used in applications such as _____ networks, _____ planning, and social networks.

(Answers: non-linear, vertices, lines or arcs, vertices, edges, isolated, length, connected, matrix, Breadth, stack, weight, loop, list, computer, route)

Ch-17 Hashing

Objectives

1 To understand the concept of hash functions, their properties, and importance in computer science.

2 To study the applications of hashing in data structures, cryptography, networking, file systems, and compilers.

3 To analyze the problem of collisions in hashing and learn different collision resolution techniques.

4 To explore how hashing enhances efficiency, security, and integrity in real-world applications like databases, blockchain, and load balancing.

A hash function is a mathematical function that takes an input (or "key") and produces a fixed-size string of bytes. It is typically represented as a sequence of characters. The output, known as the hash value, is a representation of the input data. It is often used in various computer science applications. Fig 17.1 shows hash function.

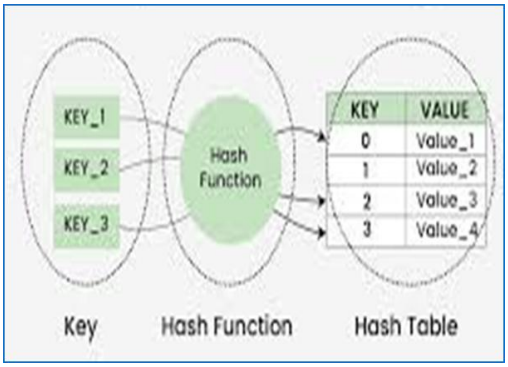


Fig 17.1: Hash Function

17.2 Properties of a Good Hash Function

Following are properties of hash function.

1. Deterministic: The same input always produces the same hash value
2. Fast Computation: It should compute the hash value quickly for any input
3. Uniform Distribution: Outputs should be uniformly distributed to minimize collisions
4. Minimizing Collisions: Different inputs should produce different hash values as much as possible

5. Irreversibility: Given a hash value, it should be computationally infeasible to determine the original input (for cryptographic hash functions)

17.3 Applications of Hash Function

Data Structures

Hash Tables are used for fast data retrieval. A hash function maps keys to indices in an array, enabling quick lookups.

Example: Storing and accessing data in dictionaries or maps.

Cryptography

Password Hashing: Passwords are hashed before storing to enhance security. When a user logs in, the hash of the entered password is compared to the stored hash.

Digital Signatures: Ensures the integrity and authenticity of data by hashing the content.

Data Integrity

Checksums: Hash functions verify data integrity during transmission or storage by checking if the hash value of the received data matches the expected hash.

Blockchain and Cryptocurrencies

Used to link blocks in a blockchain securely and ensure data immutability.

Searching and Indexing

Database Indexing: Hashes are used to quickly locate records in a database.

Search Engines: Hashing helps index web pages for efficient retrieval.

File Systems

Deduplication: Identifying duplicate files by comparing hash values.

File Names: Hashes are used to generate unique file identifiers.

Networking

Load Balancing: Distributing requests evenly across servers using consistent hashing.

Routing Protocols: Hash functions help determine packet paths in network protocols.

Compilers

Hash functions are used for symbol tables to efficiently store and retrieve variable names, function names, etc.

Image and Video Processing

Hashing is used in perceptual hashing to identify similar images or videos.

17.4 Collision Resolution

A collision occurs when two different inputs produce the same hash value. While collisions are inevitable due to the pigeonhole principle, a good hash function minimizes their likelihood. Collision resolution is essential in hash tables when two different keys hash to the same index. They are called as collision resolution techniques.

Open Addressing

In open addressing, all elements are stored within the hash table itself, and collisions are resolved by probing (searching) for the next available slot.

Separate Chaining

In separate chaining, each slot in the hash table contains a pointer to a data structure (usually a linked list) that stores all keys hashing to that slot

Coalesced Hashing

Coalesced hashing combines open addressing with separate chaining. Each slot in the hash table has a pointer to the next slot in case of a collision, forming a linked list within the table itself.

Summary

Hashing is a powerful technique in data structures and computer science that uses a hash function to map input data (keys) into fixed-size hash values for efficient data retrieval and storage. A good hash function must be deterministic, fast, uniformly distributed, collision-minimizing, and irreversible (in cryptographic contexts). Hashing has wide applications such as hash tables for quick lookups, password hashing and digital signatures in cryptography, checksums for data integrity, blockchain security, database indexing, file deduplication, networking load balancing, and symbol tables in compilers. However, collisions—where different keys map to the same hash value—are unavoidable and must be resolved using techniques like open addressing, separate chaining, or coalesced hashing. Thus, hashing forms the foundation of efficient searching, indexing, and security in computer systems.

Questions

1. **Answer the following**

(i) What is hash function?

(ii) What are properties of hash function?

(iii) State different applications of hash function.

(iv) What is collision?

(v) How collision can be resolved?

2. **Multiple Choice Questions**.

1. What is the primary purpose of a hash function?

A) To store data in sequential order

B) To compress large data into smaller sizes

C) To generate a fixed-size representation of input data

D) To randomly distribute data across memory

Answer: C) To generate a fixed-size representation of input data

2. Which of the following is NOT a property of a good hash function?

A) Deterministic behavior

B) Fast computation

C) Producing the same hash for different inputs

D) Uniform distribution of hash values

Answer: C) Producing the same hash for different inputs

3. In which of the following areas are hash functions NOT commonly used?

A) Data retrieval in hash tables

B) Sorting algorithms

C) Cryptographic security

D) Database indexing

Answer: B) Sorting algorithms

4. Why is irreversibility an important property in cryptographic hash functions?

A) To ensure that no two inputs produce the same hash

B) To prevent attackers from deriving the original input from the hash

C) To make hash computation faster

D) To allow retrieval of input data from the hash

Answer: B) To prevent attackers from deriving the original input from the hash

5. Which application of hashing is used to verify the integrity of transmitted data?

A) Password hashing

B) Digital signatures

C) Checksums

D) Load balancing

Answer: C) Checksums

6. How does separate chaining handle hash collisions?

A) By probing for the next available slot

B) By using a linked list to store multiple keys at the same index

C) By replacing the existing entry with the new one

D) By hashing the key again until a unique value is found

Answer: B) By using a linked list to store multiple keys at the same index

7. What is the main advantage of open addressing in collision resolution?

A) It requires additional linked list structures

B) All elements are stored within the hash table itself

C) It prevents collisions completely

D) It increases the chances of hash values being identical

Answer: B) All elements are stored within the hash table itself

8. Which hashing technique combines both open addressing and separate chaining?

A) Double hashing

B) Coalesced hashing

C) Linear probing

D) Direct mapping

Answer: B) Coalesced hashing

9. How does hashing help in load balancing?

A) By storing data in an array for quick access

B) By distributing incoming requests evenly across multiple servers

C) By verifying data authenticity in blockchain

D) By encrypting passwords before storage

Answer: B) By distributing incoming requests evenly across multiple servers

10. In blockchain technology, what is the role of hashing?

A) To encrypt transactions for security

B) To create unique digital signatures

C) To securely link blocks and maintain immutability

D) To store transaction history in a compressed form

Answer: C) To securely link blocks and maintain immutability

3. State True or False

1. A hash function always produces a variable-length output based on the input.

   False (It produces a fixed-size output.)

2. A good hash function should be deterministic, meaning the same input always results in the same hash value.

   True

3. Password hashing is used to store passwords securely by encrypting them.

   False (Hashing is different from encryption; hashing is one-way, while encryption is reversible.)

4. Hash functions are commonly used in data structures such as hash tables for fast data retrieval.

   True

5. A well-designed hash function should produce hash values that are evenly distributed to minimize collisions.

   True

6. Digital signatures use hashing to ensure the authenticity and integrity of data.

   True

7. In open addressing, hash collisions are handled by using a linked list at each index of the hash table.

   False (Open addressing resolves collisions by searching for the next available slot, not by using linked lists.)

8. Separate chaining resolves collisions by using a linked list to store multiple keys at the same index.

   True

9. Blockchain technology does not use hashing for security purposes.

   False (Hashing is a crucial component of blockchain security.)

10. Checksums use hash functions to verify data integrity during transmission or storage.

    True

11. Coalesced hashing is a collision resolution technique that combines elements of both open addressing and separate chaining.

    True

12. In database indexing, hashing is used to quickly locate records.

    True

13. Hashing cannot be used in networking applications such as load balancing or routing.

    False (Hashing is used in networking for load balancing and routing.)

14. Hash functions ensure that different inputs always produce different hash values without any chance of collisions.

False (Collisions can occur due to the pigeonhole principle, but a good hash function minimizes them.)

15. Hashing helps in file deduplication by comparing file contents directly instead of using hash values.

    False (Hashing helps by comparing hash values, not the actual file contents.)

4. Fill in the blanks.

1. A _____ is a mathematical function that converts an input into a fixed-size string of characters.

   (Hash function)

2. The output of a hash function is called a _____.

   (Hash value)

3. A good hash function should have _____ distribution to minimize collisions.

   (Uniform)

4. In _____, passwords are transformed into hash values before being stored for security purposes.

   (Password hashing)

5. _____ are used to verify the integrity of transmitted or stored data.

   (Checksums)

6. Hash tables use a _____ to map keys to indices in an array for fast data retrieval.

   (Hash function)

7. _____ hashing is a technique that combines open addressing and separate chaining to resolve collisions.

   (Coalesced)

8. In _____, all elements are stored within the hash table itself, and collisions are resolved by probing for the next available slot.

   (Open addressing)

9. _____ ensures that the same input always results in the same hash value.

(Deterministic property)

10. Digital signatures use hashing to ensure the _____ and authenticity of data.

    (Integrity)

11. Hash functions play a crucial role in _____ technology to securely link blocks.

    (Blockchain)

12. In _____, each slot in the hash table contains a pointer to a linked list storing multiple keys hashing to the same slot.

    (Separate chaining)

13. Hashing is used in _____ engines to index web pages for efficient retrieval.

    (Search)

14. _____ is a property of cryptographic hash functions that makes it computationally infeasible to determine the original input from its hash value.

    (Irreversibility)

15. In networking, hashing is used for _____, which distributes traffic evenly across multiple servers.

    (Load balancing)