DATA STRUCTURE AND ALGORITHMS IN C & C++\

ARRAY

An **array** is a fundamental data structure in computer science that holds a collection of elements, typically of the same data type, stored in contiguous memory locations. Arrays allow you to store multiple items under a single variable name, making it easier to manage and access large sets of related data.

Features of an Array:

1. **Fixed Size**: The size of an array is defined at the time of creation and cannot be changed. This means that you need to know in advance how many elements the array will hold.
2. **Indexed**: Arrays use zero-based indexing (in most programming languages), meaning the first element is accessed using index `0`, the second element with index `1`, and so on.
3. **Homogeneous Elements**: All elements in an array must be of the same data type (e.g., integers, strings, or floats).

Types of Arrays:

- **One-dimensional arrays (1D)**: A simple list of elements.

```
arr = [1, 2, 3, 4, 5]
```

- **Multi-dimensional arrays**: Arrays can have more than one dimension. A two-dimensional array, for example, is like a table or matrix.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Advantages of Arrays:

- **Random Access**: You can access elements in constant time (O(1)) if you know the index.
- **Efficient Memory Usage**: Since arrays use contiguous memory locations, they tend to be memory efficient.
- **Simplicity**: Arrays are simple to understand and easy to implement.

Disadvantages of Arrays:

- **Fixed Size**: Once created, the size of an array cannot be changed (in static arrays).

- **Insertion/Deletion**: Inserting or deleting elements at arbitrary positions can be inefficient because it may require shifting elements.

SPARSE MATRIX

A **sparse matrix** is a matrix in which most of the elements are zero (or a default value). These matrices are used to efficiently represent data structures where the majority of the elements are empty or zero, saving space and improving performance for operations such as matrix multiplication or addition.

Characteristics of a Sparse Matrix:

1. **High Number of Zeros**: The majority of the elements in a sparse matrix are zero (or another default value).
2. **Efficient Storage**: Since most elements are zeros, it is inefficient to store them in a traditional 2D array or matrix, as it would require unnecessary memory for the zero values.
3. **Efficient Operations**: Sparse matrices enable more efficient computation and storage by focusing on the non-zero elements.

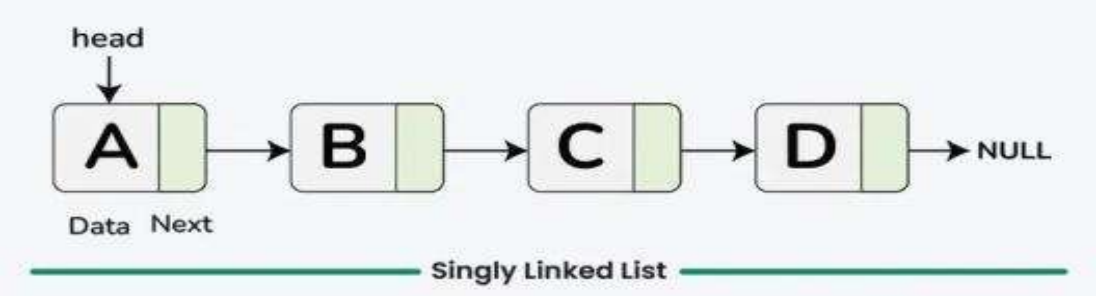
LINKED LIST

linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.

Types of linked list

1. **Singly linked list**

A singly linked list is a fundamental data structure, it consists of nodes where each node contains a data field and a reference to the next node in the linked list. The next of the last node is null, indicating the end of the list. Linked Lists support efficient insertion and deletion operations.



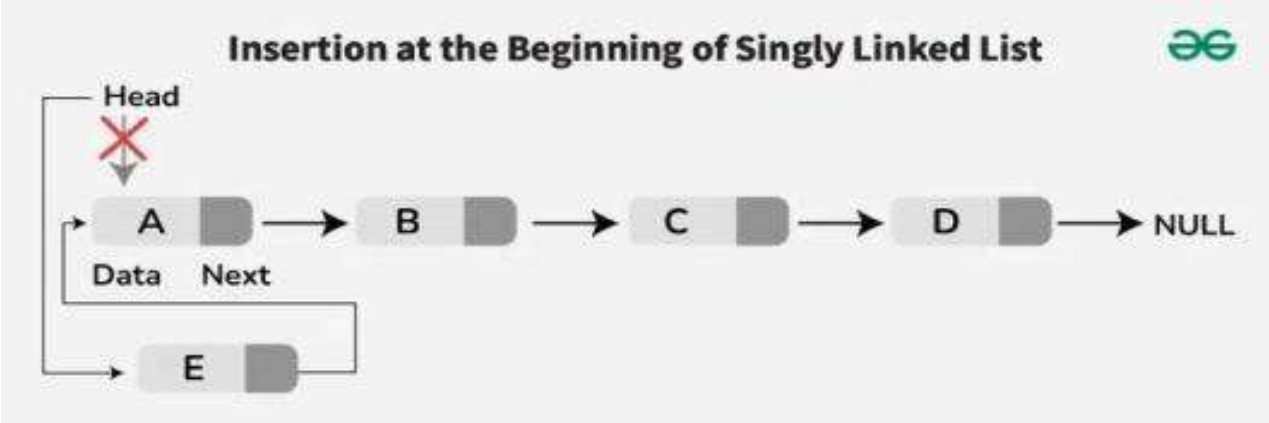
In a singly linked list, each node consists of two parts: data and a pointer to the next node. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.

Insertion in Singly Linked List

Insertion is a fundamental operation in linked lists that involves adding a new node to the list.

There are several scenarios for insertion:

a. Insertion at the Beginning of Singly Linked List:

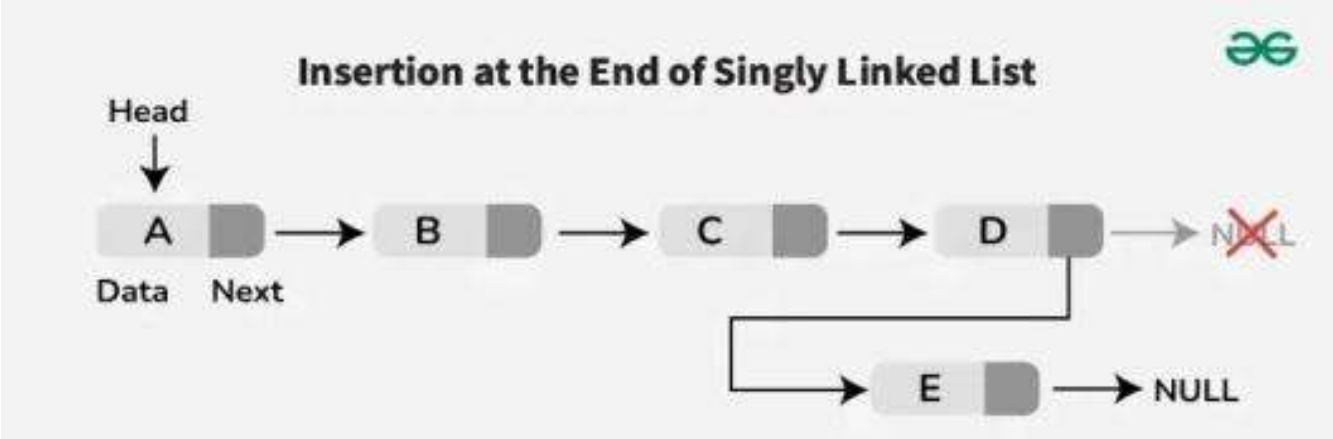


Step-by-step approach:

- Create a new node with the given value.
- Set the next pointer of the new node to the current head.
- Move the head to point to the new node.
- Return the new head of the linked list.

b. Insertion at the End of Singly Linked List:

To insert a node at the end of the list, traverse the list until the last node is reached, and then link the new node to the current last node-

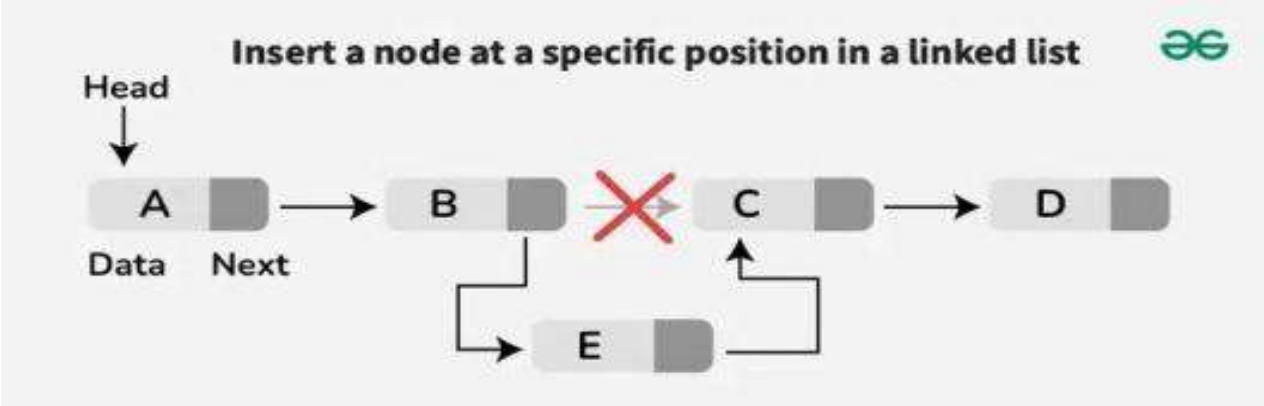


Step-by-step approach:

- Create a new node with the given value.
- Check if the list is empty:
- If it is, make the new node the head and return.
- Traverse the list until the last node is reached.
- Link the new node to the current last node by setting the last node's next pointer to the new node.

c. Insertion at a Specific Position of the Singly Linked List:

To insert a node at a specific position, traverse the list to the desired position, link the new node to the next node, and update the links accordingly.

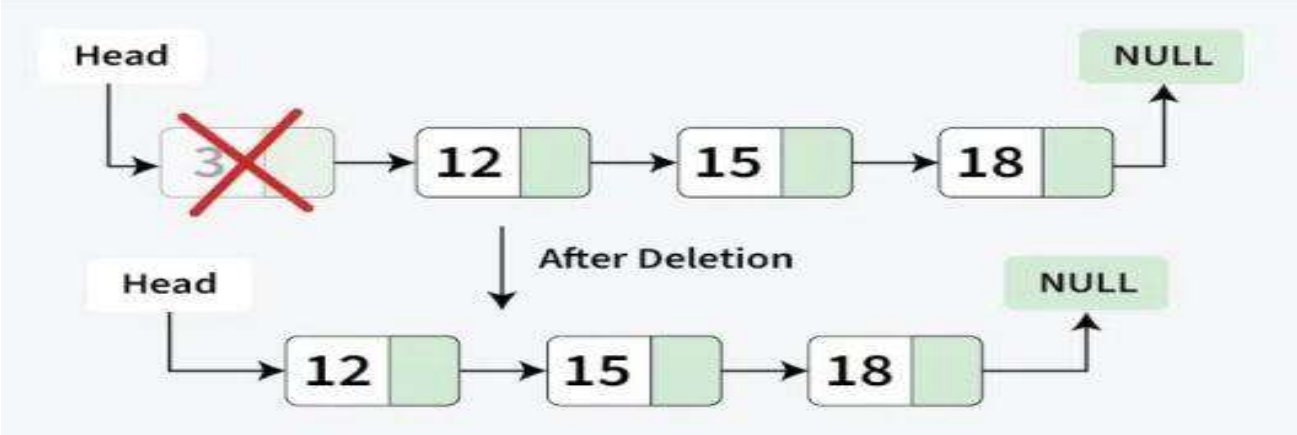


We mainly find the node after which we need to insert the new node. If we encounter a NULL before reaching that node, it means that the given position is invalid.

Deletion in Singly Linked List

Deletion involves removing a node from the linked list. Similar to insertion, there are different scenarios for deletion:

a. Deletion at the Beginning of Singly Linked List:
To delete the first node, update the head to point to the second node in the list.

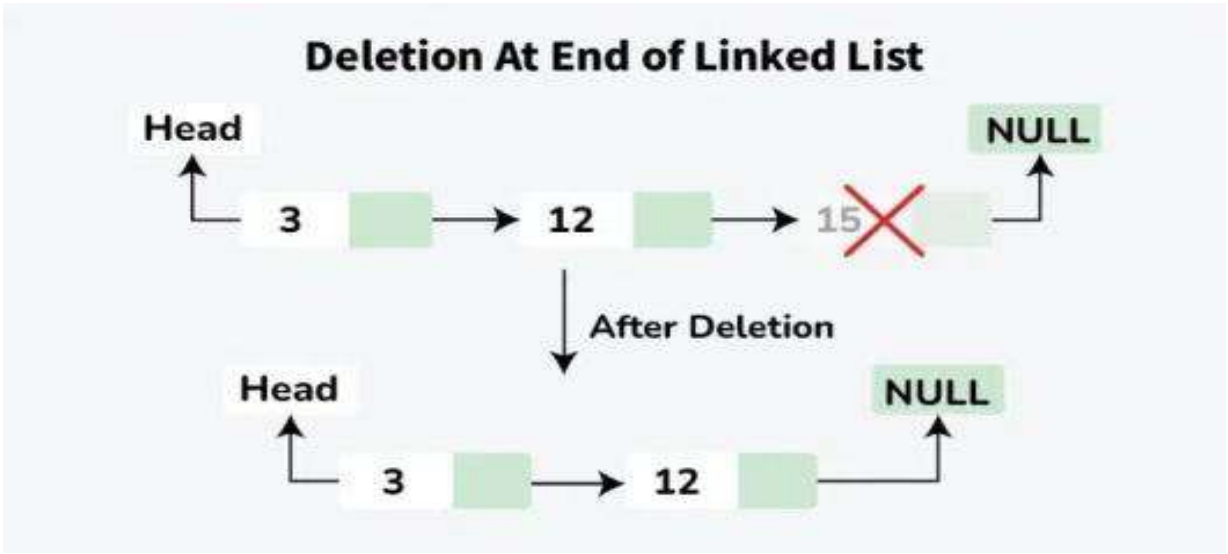


Steps-by-step approach:

- Check if the head is NULL.
- If it is, return NULL (the list is empty).
- Store the current head node in a temporary variable temp.
- Move the head pointer to the next node.
- Delete the temporary node.
- Return the new head of the linked list.

b. Deletion at the End of Singly Linked List:

To delete the last node, traverse the list until the second-to-last node and update its next field to None.



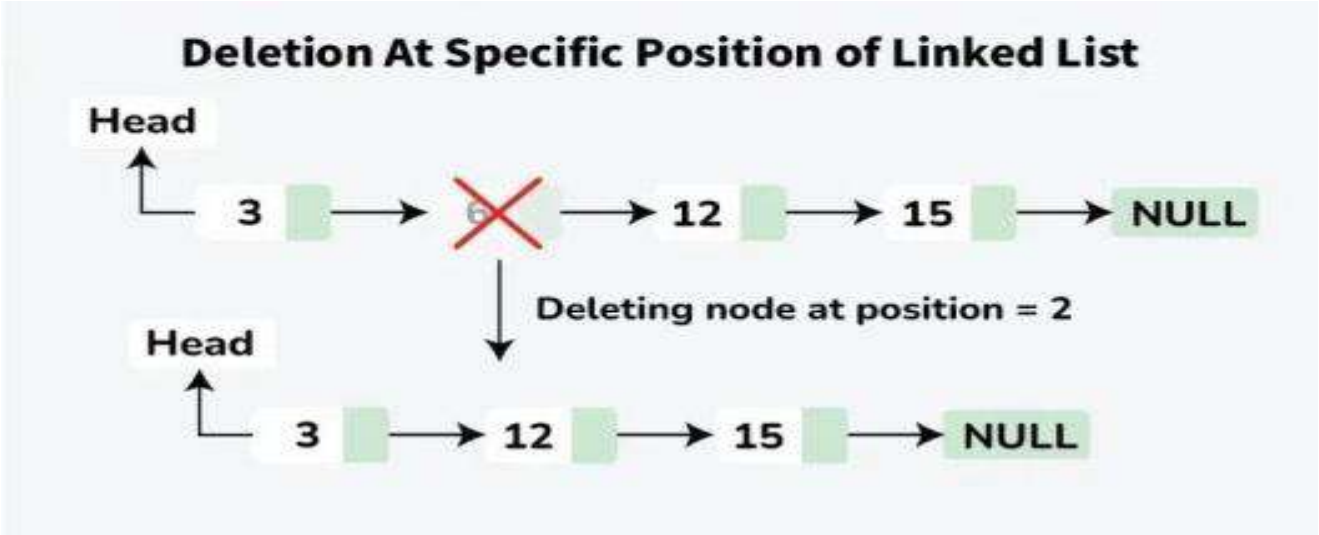
Step-by-step approach:

- Check if the head is NULL.
    - If it is, return NULL (the list is empty).
- Check if the head's next is NULL (only one node in the list).
    - If true, delete the head and return NULL.

- Traverse the list to find the second last node (second_last).
- Delete the last node (the node after second_last).
- Set the next pointer of the second last node to NULL.
- Return the head of the linked list.

c. Deletion at a Specific Position of Singly Linked List:

To delete a node at a specific position, traverse the list to the desired position, update the links to bypass the node to be deleted.

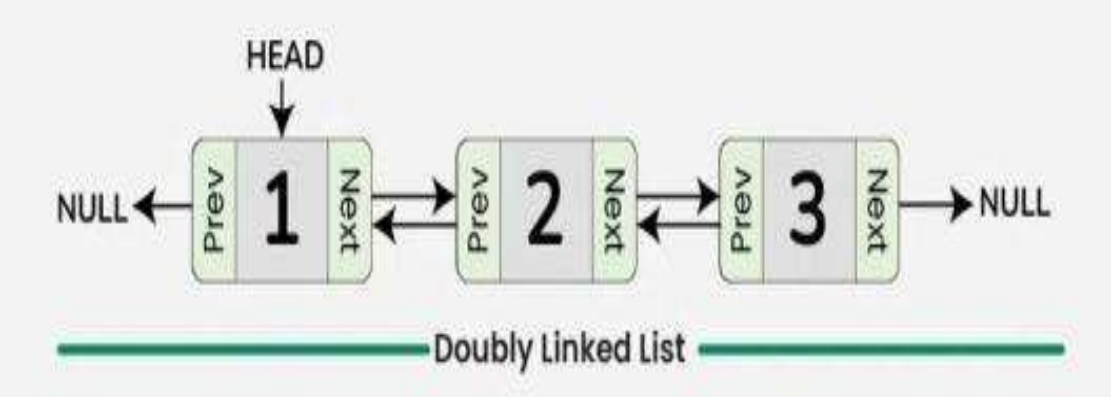


Step-by-step approach:

- Check if the list is empty or the position is invalid, return if so.
- If the head needs to be deleted, update the head and delete the node.
- Traverse to the node before the position to be deleted.
- If the position is out of range, return.
- Store the node to be deleted.
- Update the links to bypass the node.
- Delete the stored node.

2. Doubly Linked List

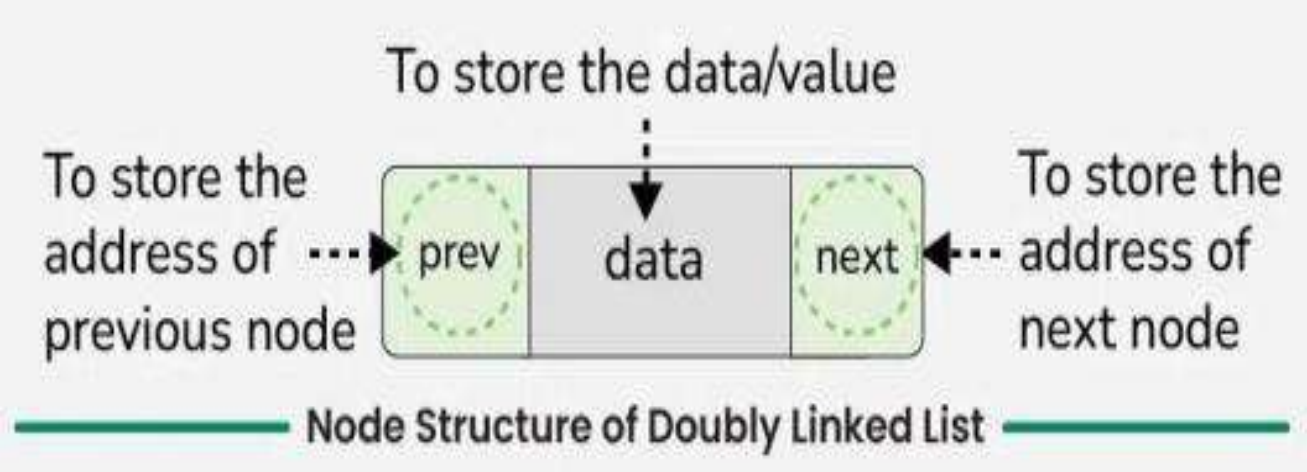
A doubly linked list is a data structure that consists of a set of nodes, each of which contains a value and two pointers, one pointing to the previous node in the list and one pointing to the next node in the list. This allows for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.



Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

- Data
- A pointer to the next node (next)
- A pointer to the previous node (prev)



Node Definition

Each node in a Doubly Linked List contains the data it holds, a pointer to the next node in the list, and a pointer to the previous node in the list. By linking these nodes together through the next and prev pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List.

3. Circular Linked List

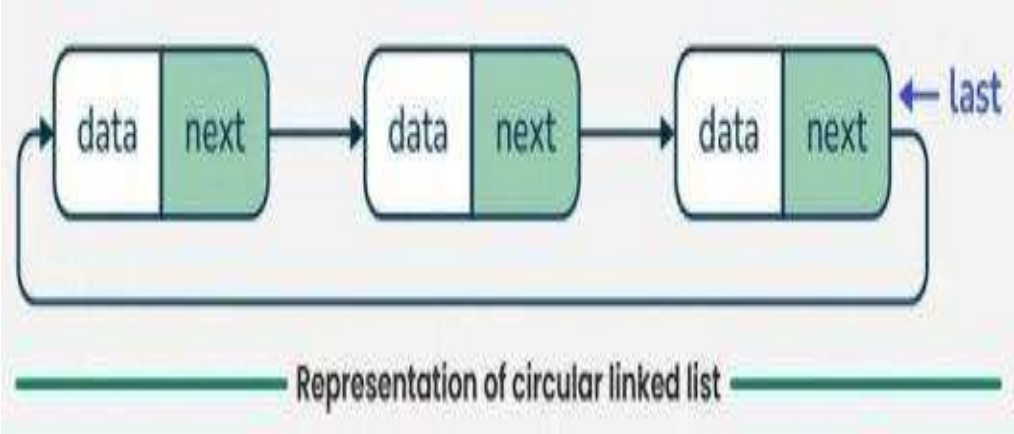
A circular linked list is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to NULL, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a NULL value.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked lists and doubly linked lists. So, circular linked list are basically of two types:

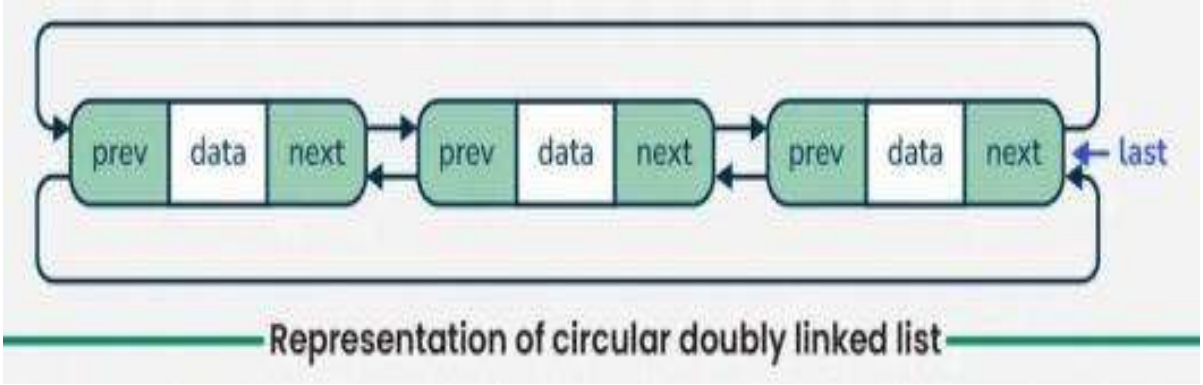
1. Circular Singly Linked List

In Circular Singly Linked List, each node has just one pointer called the “next” pointer. The next pointer of last node points back to the first node and this results in forming a circle. In this type of Linked list we can only move through the list in one direction.



2. Circular Doubly Linked List:

In circular doubly linked list, each node has two pointers prev and next, similar to doubly linked list. The prev pointer points to the previous node and the next points to the next node. Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



STACK

What is Stack Data Structure?

Stack is a **linear data structure** based on LIFO(Last In First Out) principle in which the insertion of a new element and removal of an existing element takes place at the same end represented as the **top** of the stack.
To implement the stack, it is required to maintain the **pointer to the top of the stack** , which is the last element to be inserted because **we can access the elements only on the top of the stack.**

LIFO(Last In First Out) Principle in Stack Data Structure:

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Basic Operations on Stack Data Structure:

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **isFull()** returns true if the stack is full else false.

Push Operation in Stack Data Structure:

Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**
Algorithm for Push Operation:

- Before pushing the element to the stack, we check if the stack is **full** .
- If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.
- Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position** .
- The elements can be pushed into the stack till we reach the **capacity** of the stack.

Pop Operation in Stack Data Structure:

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for Pop Operation:

- Before popping the element from the stack, we check if the stack is **empty** .
- If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.
- Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)** and return the stored top value.

Top or Peek Operation in Stack Data Structure:

Returns the top element of the stack.

Algorithm for Top Operation:

- Before returning the top element from the stack, we check if the stack is empty.
- If the stack is empty (top == -1), we simply print “Stack is empty”.
- Otherwise, we return the element stored at **index = top** .

isEmpty Operation in Stack Data Structure:

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation:

- Check for the value of **top** in stack.
- If **(top == -1)** , then the stack is **empty** so return **true** .
- Otherwise, the stack is not empty so return **false** .

isFull Operation in Stack Data Structure:

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

- Check for the value of **top** in stack.
- If **(top == capacity-1),** then the stack is **full** so return **true**.
- Otherwise, the stack is not full so return **false**.

QUEUE

What is Queue Data Structure?

Queue Data Structure is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.
We define a queue to be a list in which all additions to the list are made at one end (**back of the queue**), and all deletions from the list are made at the other end(**front of the queue**). The element which is first pushed into the order, the delete operation is first performed on that.

FIFO Principle of Queue Data Structure:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue). Similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue.

Basic Operations in Queue Data Structure:

Some of the basic operations for Queue in Data Structure are:

1. **Enqueue:** Adds (or stores) an element to the end of the queue..
2. **Dequeue:** Removal of elements from the queue.
3. **Peek or front:** Acquires the data element available at the front node of the queue without deleting it.
4. **rear:** This operation returns the element at the rear end without removing it.
5. **isFull**: Validates if the queue is full.
6. **isEmpty**: Checks if the queue is empty.

There are a few supporting operations (auxiliary operations):

1. Enqueue Operation in Queue Data Structure:

Enqueue() operation in Queue **adds (or stores) an element to the end of the queue**.
The following steps should be taken to enqueue (insert) data into a queue:

- **Step 1:** Check if the queue is full.
- **Step 2:** If the queue is full, return overflow error and exit.
- **Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.

- **Step 4:** Add the data element to the queue location, where the rear is pointing.
- **Step 5:** return success.

2. Dequeue Operation in Queue Data Structure:

Removes (or access) the first element from the queue.
The following steps are taken to perform the dequeue operation:

- **Step 1:** Check if the queue is empty.
- **Step 2:** If the queue is empty, return the underflow error and exit.
- **Step 3:** If the queue is not empty, access the data where the front is pointing.
- **Step 4:** Increment the front pointer to point to the next available data element.
- **Step 5:** The Return success.

Types of Queue Data Structure:

Queue data structure can be classified into 4 types:
There are different types of queues:

1. **Simple Queue:** Simple Queue simply follows **FIFO** Structure. We can only insert the element at the back and remove the element from the front of the queue.
2. **Double-Ended Queue (Dequeue):** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
   - **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
   - **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. **Circular Queue:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order.
4. **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:
   - **Ascending Priority Queue:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.

- **Descending Priority Queue:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popped first.

LINEAR SEARCH

In Linear Search, we iterate over all the elements of the array and check if it the current element is equal to the target element. If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found. Linear search is also known as **sequential search**

For example:
Consider the array **arr[] = {10, 50, 30, 70, 80, 20, 90, 40}** and **key** = 30

Time and Space Complexity of Linear Search Algorithm:

Time Complexity:

- **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
- **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
- **Average Case:** O(N).

Applications of Linear Search Algorithm:

- **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with
- **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.

- It is a well-suited algorithm for small datasets.

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
- Not suitable for large arrays.

When to use Linear Search Algorithm?

- When we are dealing with a small dataset.
- When you are searching for a dataset stored in contiguous memory.

BINARY SEARCH

Binary Search Algorithm is a searching algorithm used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

What is Binary Search Algorithm?

Binary search is a search algorithm used to find the position of a target value within a **sorted** array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space.

Conditions to apply Binary Search Algorithm in a Data Structure

To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure should take constant time.

Binary Search Algorithm

Below is the step-by-step algorithm for Binary Search:

- Divide the search space into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
    - If the **key** is smaller than the middle element, then the **left** side is used for next search.
    - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

- **Time Complexity:**
    - Best Case: O(1)
    - Average Case: O(log N)
    - Worst Case: O(log N)

Applications of Binary Search Algorithm

- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- It can be used for searching a database.

Advantages of Binary Search

- Binary search is faster than linear search, especially for large arrays.

- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Disadvantages of Binary Search

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

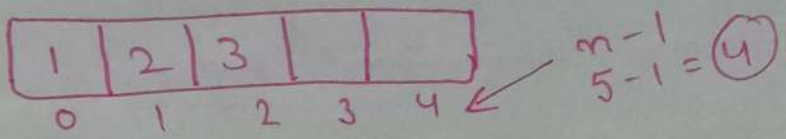
Important Differences

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

① Limitations of arrays -

↳ Fixed size
↳ Homogeneous elements only (lack of Flexibility)
↳ Costly Insertion and deletions
↳ Sequential Access only
↳ Memory waste

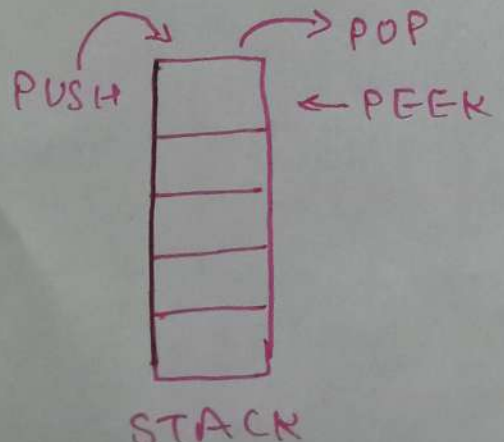
a[5] = { 1, 2, 3 }



② Stack

↳ LIFO (last In First Out)
↳ linear Data Structure
↳ The element w|c is inserted last is the element w|c removed first.
↳ It provides reverse order of operation
↳ There are 3 major operations
↳ PUSH ↳ POP ↳ PEEK
TOP

Example → Stack of plates in table
→ Stack of bangles in hand
↳ Stack of Books in box

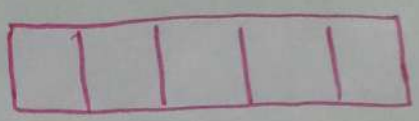


Queue

↳ FIFO (First in First out)
↳ linear data structure
↳ The element w|c is insert first is the element w|c removed first.
↳ Operations →
⇒ Enqueue - ~~Rear~~ end
Insertion at last
⇒ Dequeue - ~~Front~~ End
Delete First
⇒ PEEK
First element access

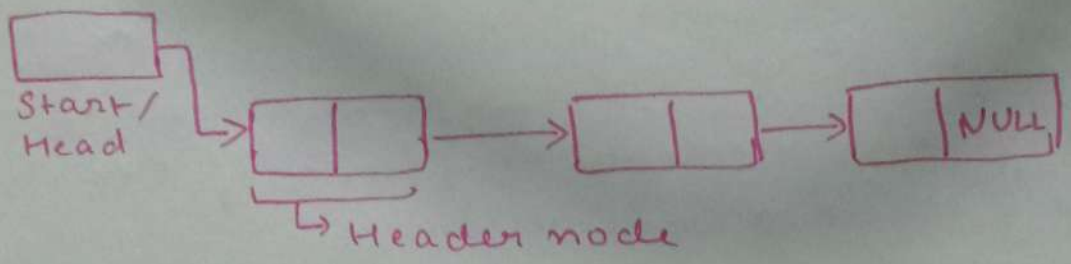
Example →

Queue of lines →
like ticket counter, Examination gate on checking.



QUEUE

③ Advantage of header node in a linked list



↳ Stores the number of nodes
↳ Marker for the beginning of the list
↳ Accesses all node data and types

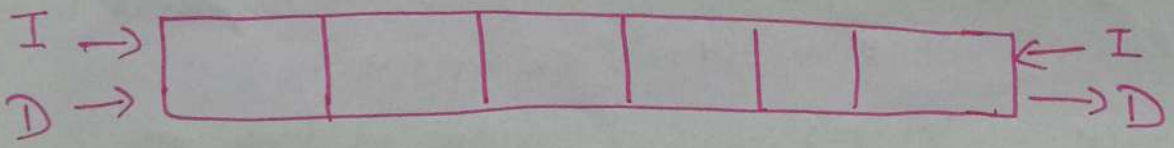
④ Sparse matrix - Represent in memory

A sparse matrix is a matrix with more zero element than non-zero element.

To save space in memory, sparse matrix are represented by storing only the non-zero values.

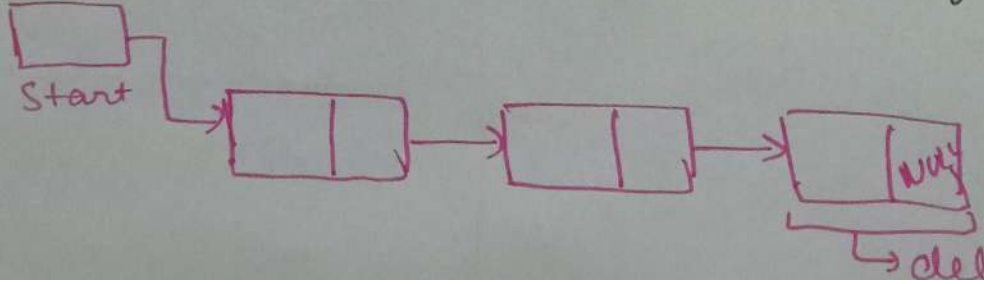
$$\begin{bmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 5 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

⑤ Dequeue → Deque (DS)



↳ Browsing history
↳ Recently visited URLs
↳ Undo operations

⑥ Algorithm to delete last node from a linked list.



Step1→ If Start = NULL

Write Underflow

exit

Step2→ Take prev of node type

Step3→ else

while(current→link != NULL)

prev = current

current = current→link

Step4→ Now prev→link = NULL

Step5→ Free current

Step6→ EXIT (END)

(7) Insert an element - Algoritem

Step1- Input item, loc

Step2- if n == MAX

Write Overflow

Step3- Downward shifting

i = n-1 ; i >= loc ; i--

Step4- a[i+1] = a[i]

Step5- End loop

Step6- Insertion at loc

a[loc] = item

Step7- Increase input data size

n = n+1

Step8- EXIT (END)

Q) Postfix expression evaluate - Algorithm

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.

3, 1, +, 2, ↑, 7, 4, -, 2, *, +, 5, -

Symbol scanned	STACK
3	3
1	3 1
+	4
2	4 2
↑	16
7	16 7
4	16 7 4
-	16 3
2	16 3 2
*	16 6
+	22
5	22 5
-	17

3+1=4

4^2=16

7-4=3

3*2=6

16+6=22

22-5=17

(9) Priority Queue

Stores elements with associated priorities and process them in order of priority

---

(10) Data structures operations

- ↳ Searching
- ↳ Sorting
- ↳ Insertion
- ↳ Deletion
- ↳ Traversal

(11) Stack overflow

occurs in recursion.

Solve → tail recursion
→ iterative solutions
→ memorization
→ dynamic programming
→ divide and conquer.

(12) Prefix & postfix form for

A + B * (C - D) / (E - F)

Prefix - + / - F E * - D C B A

Postfix - A B C D - * E F - / +

(13) Recursive Factorial Function

```
int fact(int n)
{
    if (n==0)
        return(1);
    else
        return(n * fact(n-1));
}
```

(14) Types of sparse matrix

↳ Diagonal sparse matrix

↳ Tridiagonal sparse matrix

↳ Triangular sparse matrix

2-D sparse matrix → Single Dimensional array

Instead of store whole matrix only store non zero elements in array.

Formula of Address calculation

Column major order

loc (a[i][j]) = base(a) + w x (r * (j - lbc) + (i - lbr))

Row major order

loc (a[i][j]) = base(a) + w x ((j - lbc) + c x (i - lbr))

Here base(a) → base address of Matrix

w → No. of bytes words per memory call

r → No. of row

lbc → lower bound of column

c → No. of column

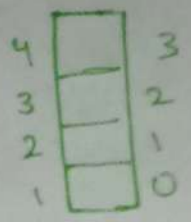
ubr → upper bound of row

Ex- SCORE 25 x 4 Base(SCORE) = 200

w = 4 words per memory

Suppose row major order (RMO)

Find the address of SCORE [12, 3]



↳ Assume lbc is 0

loc (a[12][3])

= 200 + 4 x ((3-0) + 4 x (12-0))

= 200 + 4 x (3 + 48)

= 200 + 4 x 51

= 200 + 204

= 404

Am C

By default lbc is 1

When answer = 304

(15) D-Queue & Priority Queue

↳ DONE ✓

(16) Multiply two matrices A & B

↳ DONE ✓

(17) Create, Insert & delete - Singly linked list

↳ DONE ✓

(18) Circular Queue

↳ DONE ✓

(19) Linear & Binary search difference

↳ DONE ✓

---

(20) deque

↳ ~~DONE~~ ✓

(21) Analysis of algorithm

The process of evaluating the performance of an algorithm.

Usually in terms of time and space complexity.

(22) Principle of recursion

Method of solving a problem it down into smaller, more manageable subproblems, and then solving those subproblems using the same function that called them.

↳ Call itself

↳ Must have a base case

↳ Must change its state and move towards the base case.

(23) Linked list

↳ Done (✓)

Algorithm for insert an element

Step 1- Take newnode type of node.

Step 2- Allocate memory to newnode

Step 3- if NEWNODE == NULL
write Overflow
exit

Step 4- else
NEWNODE->INFO = ITEM
NEWNODE -> LINK = LOC->LINK

Step 5- LOC->LINK = NEWNODE (Insertion)

Step 6- END.

(24) STACK DATA STRUCTURE

↳ DONE (✓)

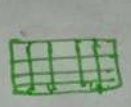
(25) Binary search and linear search

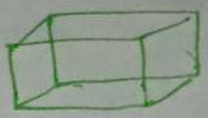
↳ DONE (✓)

(26) Mult Dimensional Array

↳ 2D array → Chess

↳ 3D array → 3D cube





(27) (R|D-X*(G/E*(A+B)+D-E*(Y-F))+S*J)

Symbol Scanned	Stack	Postfix Exp(X)
(	(	
R	(	R
|	(|	R
D	(|	RD
-	(-	RD|
X	(-	RD|X
*	(-*	RD|X
(	(-*(	RD|X
G	(-*(	RD|X G
/	(-*(/	RD|X G

E	(-*(/	RD|XGE
*	(-*(*	RD|XGE/
(	(-*(*(	RD|XGE/
A	(-*(*(	RD|XGE/A
+	(-*(*(+	RD|XGE/A
B	(-*(*(+	RD|XGE/AB
)	(-*(*	RD|XGE/AB+
+	(-*(+	RD|XGE/AB+*
D	(-*(+	RD|XGE/AB+*D
-	(-*(-	RD|XGE/AB+*D+
E	(-*(-	RD|XGE/AB+*D+E
*	(-*(-*	RD|XGE/AB+*D+E
(	(-*(-*	RD|XGE/AB+*D+E
Y	(-*(-*	RD|XGE/AB+*D+EY
-	(-*(-*	RD|XGE/AB+*D+EY
F	(-*(-*	RD|XGE/AB+*D+EYF
)	(-*(-*	RD|XGE/AB+*D+EYF-
)	(-*	RD|XGE/AB+*D+EYF-*-
+	(+	RD|XGE/AB+*D+EYF-*-*-
S	(+	RD|XGE/AB+*D+EYF-*-*-S
*	(+*	RD|XGE/AB+*D+EYF-*-*-S
J	(+*	RD|XGE/AB+*D+EYF-*-*-SJ
)		RD|XGE/AB+*D+EYF-*-*-SJ*+

RD|XGE/AB+*D+EYF-*-*-SJ*+

Step by step → Algo INFIX → POSTFIX

(X + O Y)

1. Push " (" onto STACK, and add ") " to the end of X

(2) Scan X from left to right and REPEAT Step 3 to 7 for each element of X UNTILL the STACK is empty.

(3) If an operand is encountered, add it to Y

(4) If a left paranthesis is encountered, push it onto STACK.

(5) If an operator is encountered, then-

(a) Repeatedly pop from STACK and add to Y each operator (on the Top of STACK) w/c has the same precedence as or higher precedence than operator.

(b) Add operator to STACK

(6) If a right paranthesis is encountered, then:-

(a) Repeatedly pop from STACK and add to Y each operator (on the top of STACK) until a left parenthesis is encountered.

(b) Remove the left paranthesis.
(Do not add the left paranthesis to Y)

(7) Repeated pop from STACK and add to Y each operator until STACK is empty

(8) EXIT.

(28) Doubly linked list is better than linked list

Features	SLL	DLL
Traversal	Forward	Forward and backward
Deletion efficiency	less efficient	more efficient
Use cases	Simple tasks	Complex operations

(29) Two dimensional array is stored in memory (6)

Row major order (RMO)	Column Major order (CMO)
2 | 3 | 4 | 5 | 7 | 8	2 | 4 | 7 | 3 | 5 | 8
00 01 10 11 20 21	00 10 20 01 11 21

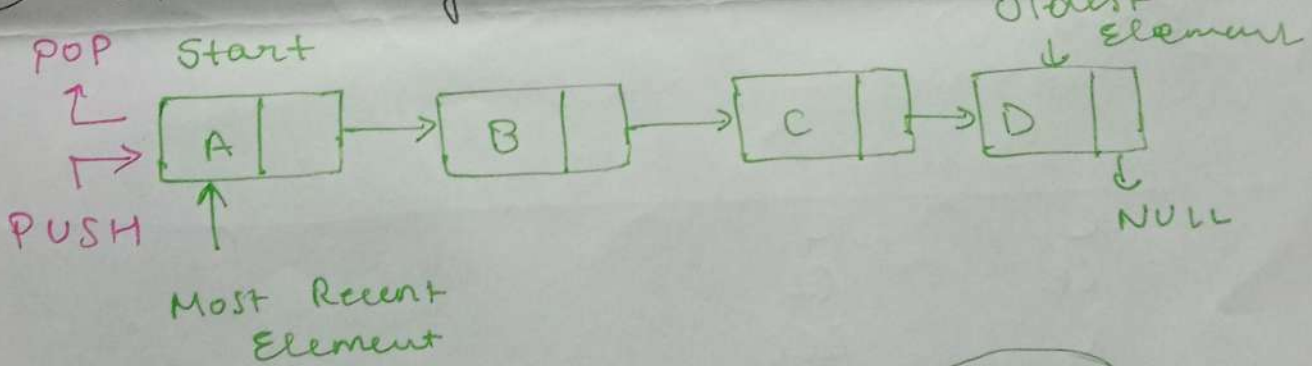
$$\begin{matrix} & C_0 & C_1 \\ r_0 & 2 & 3 \\ r_1 & 4 & 5 \\ r_2 & 7 & 8 \end{matrix}$$

3 × 2

(30) What is priority queue -
↳ DONE (V)

(31) linked list -
↳ DONE (V)

(32) STACK using linked list



(33) Multidimensional array (m x n) (Matrix)
↳ Done (V)

(34)

Structure	Union
↳ Keyword struct	↳ Keyword Union
↳ Size → sum of sizes of its members	↳ Size → size of largest member
↳ Altering the value of member will not affect other member	↳ will alter other members value
↳ Individual member can be accessed at time	↳ Only one member can be accessed at a time
↳ Several members of a structure can initialized at once	↳ only the first member of a union can be initialized

(35) Explain method Deletion linked list
↳ DONE ✓

(36) Binary and linear search
↳ DONE ✓

---

(37) Sparse Array
↳ DONE ✓

(38) Applications of STACK
↳ Expression evaluation
↳ Expression conversion
↳ Syntax parsing
↳ String Reversal

(39) Priority Queue
↳ DONE ✓

(40) Algorithm
↳ DONE ✓

((CA+B)^C)-((CD*C)/F))

Symbol scanned	STACK	Postfix expression
(	(	
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	AB
)	((	AB+
^	((^	AB+
C	((^	AB+C
)	(	AB+C^
-	(-	AB+C^
(	(-(	AB+C^
C	(-((	AB+C^
D	(-(((	AB+C^D

*	C- CC*	AB+C^D
C	C- CC*	AB+C^DC
)	C-C	AB+C^DC*
/	C-C/	AB+C^DC*
F	C-C/	AB+C^DC*F
)	C-	AB+C^DC*F/
)		AB+C^DC*F/-

AB+C^DC*F/-

(41) Linear Search & Binary Search
↳ done ✓

(42) Program - Insertion / Deletion in Simple Queue
↳ DONE ✓

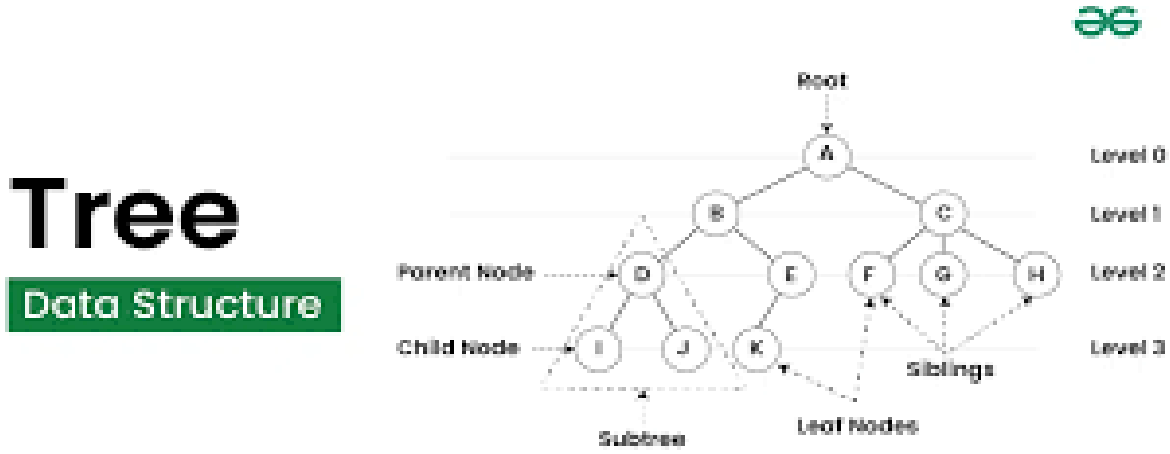
(43) Multiply two matrix 3×3
↳ DONE ✓

(44) Singly list Implementation
↳ DONE ✓

(45) D-Queues
↳ DONE ✓

Introduction to Tree Data Structure

- **Tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
  The topmost node of the tree is called the **root**, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.

- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Different Types of Trees in Data Structure

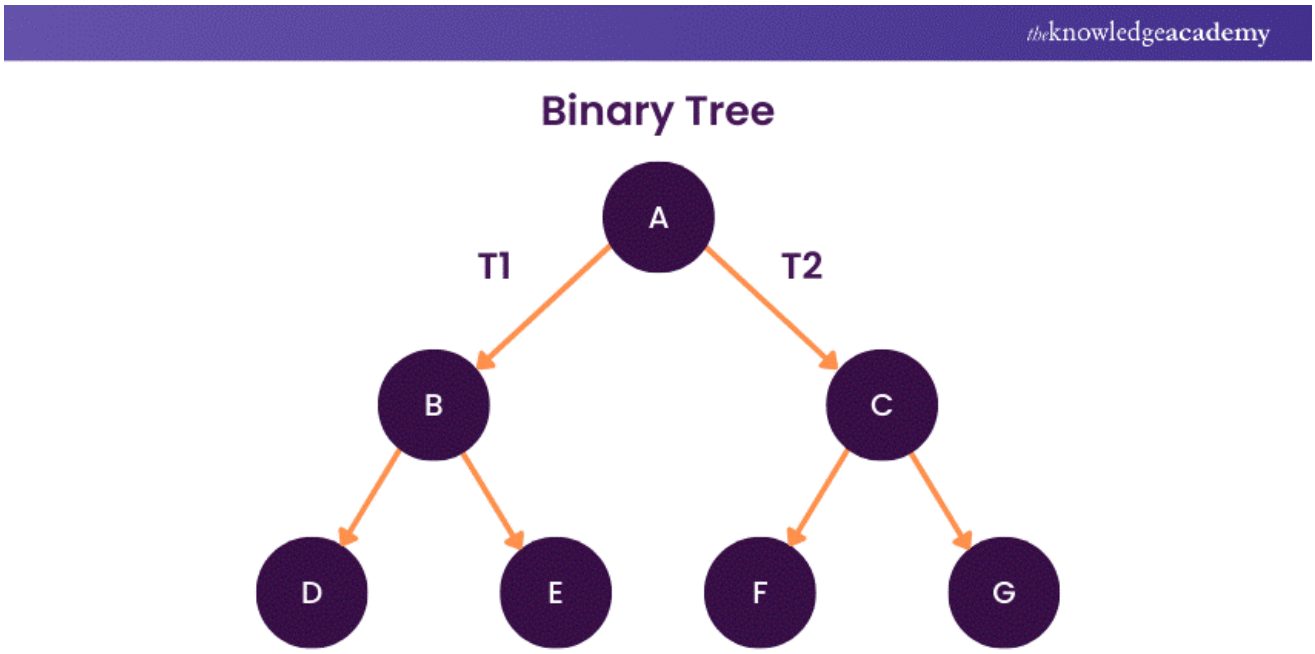
In Data Structures, trees are nothing but hierarchical models pivotal for organising and managing data. Binary trees, including AVL and Red-Black trees, offer simplicity and balance. Multiway trees like B-trees and B+ trees handle large datasets effectively.

Tries, like radix trees, specialise in storing and retrieving dynamic sets of strings. Each type serves distinct applications, contributing to computational efficiency. Below are the various types explained in detail:

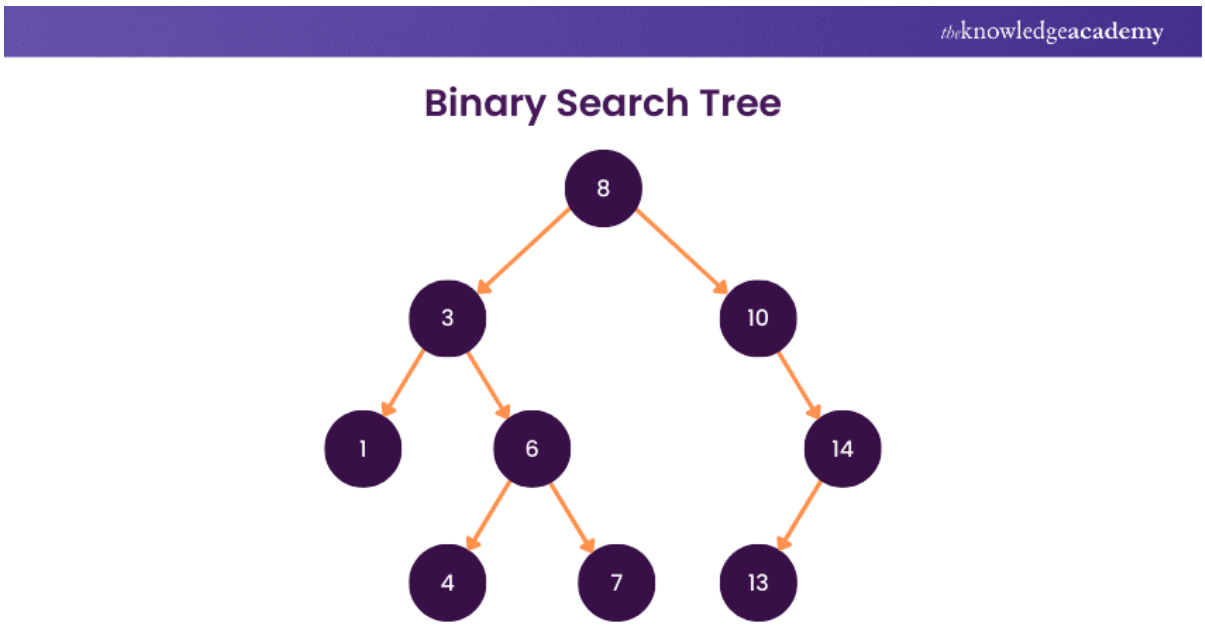
1) Binary Tree

A Binary Tree in Data Structure is a fundamental hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left and right subtrees. The topmost node is the root, and nodes without children are leaves.

More importantly, Binary Trees are efficient for search and retrieval operations, facilitating data organisation that enhances computational efficiency. Their simplicity and versatility make them a cornerstone in various algorithms and applications in Computer Science.



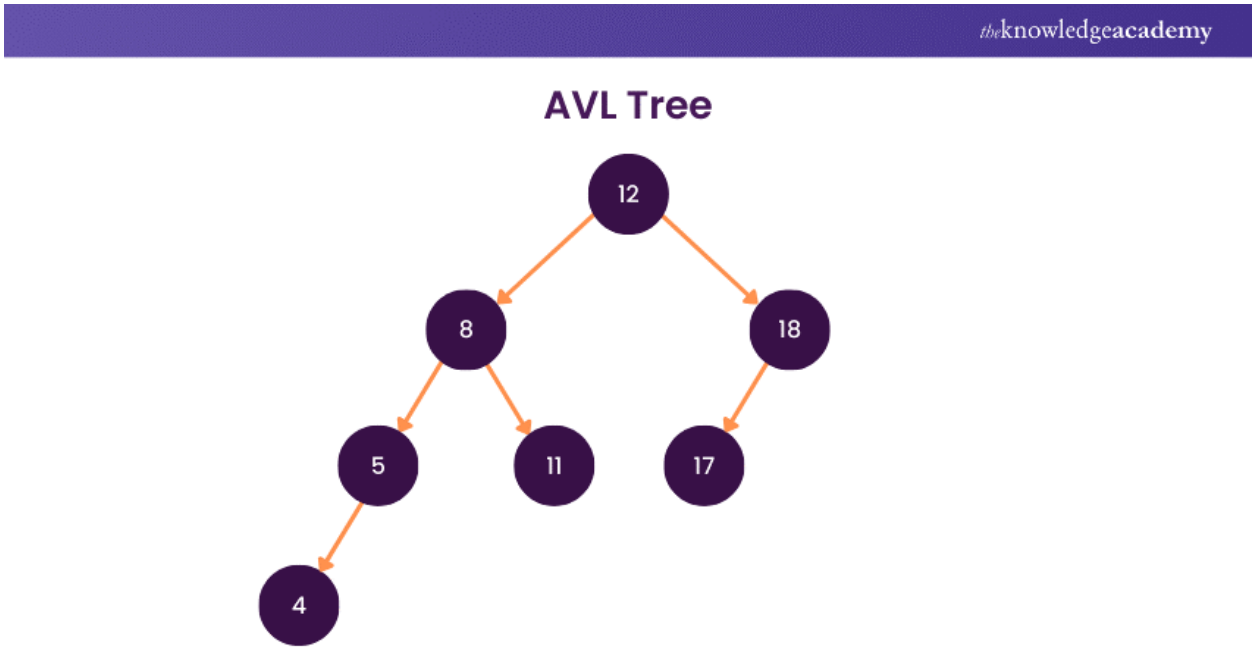
2) Binary Search Tree



A Binary Search Tree (BST) is a specialised Binary Tree where each node has at most two children, and the left child is less than the parent, while the right child is greater. This ordering property enables efficient searching, insertion, and deletion operations, making BSTs valuable in Data Structures.

Moreover, the logarithmic height ensures optimal performance, and the BST's simplicity and versatility make it a key choice for maintaining sorted collections and implementing various algorithms in Computer Science, offering a balance between simplicity and efficiency for storing and retrieving data.

3) AVL Tree



An AVL Tree is a self-balancing binary search tree, ensuring that the height difference between every node's left and right subtrees is at most one. Named after its inventors, Adelson-Velsky and Landis, an AVL Tree automatically adjusts its structure through rotations during insertion and deletion operations, maintaining logarithmic height and efficient search, insertion, and deletion times.

More importantly, this balancing property enhances the performance of operations, making AVL Trees valuable in scenarios where optimal time complexity is critical, such as database indexing.

While the maintenance of balance introduces additional overhead, the benefits of predictable performance make AVL Trees a powerful and widely used data structure in computer science for scenarios requiring fast and reliable search operations. Familiarity with Data Structures Interview Questions can be particularly useful, as it helps in understanding the nuances of AVL Trees and preparing for discussions on their implementation and advantages.

7) B-Tree

A B-Tree, short for Balanced Tree, is a self-balancing search tree data structure renowned for its efficient handling of large datasets and optimal performance for disk storage systems.

Characterised by a variable number of children per node, a B-Tree maintains balance through controlled splits and merges, ensuring a uniform depth. This balance facilitates efficient search,

Tree Traversal Techniques

- **Tree Traversal techniques** include various ways to visit all the nodes of the tree. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. In this article, we will discuss all the tree traversal techniques along with their uses.
  A Tree Data Structure can be traversed in following ways:
    - Inorder Traversal
    - Preorder Traversal
    - Postorder Traversal

Inorder Traversal:

Inorder traversal visits the node in the order: **Left -> Root -> Right**

Algorithm for Inorder Traversal:
Inorder(tree)
- *Traverse the left subtree, i.e., call Inorder(left->subtree)*
- *Visit the root.*
- *Traverse the right subtree, i.e., call Inorder(right->subtree)*

Preorder Traversal:

Preorder traversal visits the node in the order: **Root -> Left -> Right**

Algorithm for Preorder Traversal:
Preorder(tree)
- *Visit the root.*
- *Traverse the left subtree, i.e., call Preorder(left->subtree)*
- *Traverse the right subtree, i.e., call Preorder(right->subtree)*

Postorder Traversal:

Postorder traversal visits the node in the order: **Left -> Right -> Root**

Algorithm for Postorder Traversal:

Algorithm Postorder(tree)

- *Traverse the left subtree, i.e., call Postorder(left->subtree)*
- *Traverse the right subtree, i.e., call Postorder(right->subtree)*
- *Visit the root*

Hash Functions and Types of Hash functions

- **Hash functions** are a fundamental concept in computer science and play a crucial role in various applications such as data storage, retrieval, and cryptography. In data structures and algorithms (DSA), hash functions are primarily used in hash tables, which are essential for efficient data management. This article delves into the intricacies of hash functions, their properties, and the different types of hash functions used in DSA.

What is a Hash Function?

A **hash function** is a function that takes an input (or 'message') and returns a fixed-size string of bytes. The output, typically a number, is called the **hash code** or **hash value**. The main purpose of a hash function is to efficiently map data of arbitrary size to fixed-size values, which are often used as indexes in hash tables.

Key Properties of Hash Functions

- **Deterministic**: A hash function must consistently produce the same output for the same input.
- **Fixed Output Size**: The output of a hash function should have a fixed size, regardless of the size of the input.
- **Efficiency**: The hash function should be able to process input quickly.
- **Uniformity**: The hash function should distribute the hash values uniformly across the output space to avoid clustering.
- **Pre-image Resistance**: It should be computationally infeasible to reverse the hash function, i.e., to find the original input given a hash value.
- **Collision Resistance**: It should be difficult to find two different inputs that produce the same hash value.
- **Avalanche Effect**: A small change in the input should produce a significantly different hash value.
-

Applications of Hash Functions

- **Hash Tables**: The most common use of hash functions in DSA is in hash tables, which provide an efficient way to store and retrieve data.
- **Data Integrity**: Hash functions are used to ensure the integrity of data by generating checksums.
- **Cryptography**: In cryptographic applications, hash functions are used to create secure hash algorithms like SHA-256.
- **Data Structures**: Hash functions are utilized in various data structures such as Bloom filters and hash sets.

Types of Hash Functions

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. Division Method.
2. Multiplication Method
3. Mid-Square Method
4. Folding Method
5. Cryptographic Hash Functions
6. Universal Hashing
7. Perfect Hashing

1. Division Method

The division method involves dividing the key by a prime number and using the remainder as the hash value.

$h(k) = k \bmod m$

Where k is the key and m is a prime number.

Advantages:

- Simple to implement.
- Works well when m is a prime number.

Disadvantages:

- Poor distribution if m is not chosen wisely.

2. Multiplication Method

In the multiplication method, a constant A ($0 < A < 1$) is used to multiply the key. The fractional part of the product is then multiplied by m to get the hash value.

$h(k) = \lfloor m(kA \bmod 1) \rfloor$

Where $\lfloor \rfloor$ denotes the floor function.

Advantages:

- Less sensitive to the choice of m.

Disadvantages:
- More complex than the division method.

3. Mid-Square Method
In the mid-square method, the key is squared, and the middle digits of the result are taken as the hash value.
Steps:
1. Square the key.
2. Extract the middle digits of the squared value.

Advantages:
- Produces a good distribution of hash values.

Disadvantages:
- May require more computational effort.

4. Folding Method
The folding method involves dividing the key into equal parts, summing the parts, and then taking the modulo with respect to mm.
Steps:
1. Divide the key into parts.
2. Sum the parts.
3. Take the modulo mm of the sum.

Advantages:
- Simple and easy to implement.

Disadvantages:
- Depends on the choice of partitioning scheme.

5. Cryptographic Hash Functions
Cryptographic hash functions are designed to be secure and are used in cryptography. Examples include MD5, SHA-1, and SHA-256.
Characteristics:
- Pre-image resistance.
- Second pre-image resistance.
- Collision resistance.

Advantages:
- High security.

Disadvantages:
- Computationally intensive.

6. Universal Hashing

Universal hashing uses a family of hash functions to minimize the chance of collision for any given set of inputs.
$h(k)=((a \cdot k+b) \bmod p) \bmod m$
Where a and b are randomly chosen constants, p is a prime number greater than m, and k is the key.
Advantages:
- Reduces the probability of collisions.

Disadvantages:
- Requires more computation and storage.

7. Perfect Hashing
Perfect hashing aims to create a collision-free hash function for a static set of keys. It guarantees that no two keys will hash to the same value.
Types:
- Minimal Perfect Hashing: Ensures that the range of the hash function is equal to the number of keys.
- Non-minimal Perfect Hashing: The range may be larger than the number of keys.

Advantages:
- No collisions.

Disadvantages:
- Complex to construct.

Collision in Hashing
In this, the hash function is used to find the index of the array. The hash value is used to create an index for the key in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.

Collision Resolution Techniques
There are two types of collision resolution techniques.
- Separate chaining (open hashing)
- Open addressing (closed hashing)

Separate chaining: This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list. Separate chaining is the term used to describe how this connected list of slots resembles a chain. It is more frequently utilized when we are unsure of the number of keys to add or remove.

Time complexity

- Its worst-case complexity for searching is o(n).
- Its worst-case complexity for deletion is o(n).

Advantages of separate chaining
- It is easy to implement.
- The hash table never fills full, so we can add more elements to the chain.
- It is less sensitive to the function of the hashing.

Disadvantages of separate chaining
- In this, the cache performance of chaining is not good.
- Memory wastage is too much in this method.
- It requires more space for element links.

Open addressing: To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. Additionally known as closed hashing.
The following techniques are used in open addressing:
- Linear probing
- Quadratic probing
- Double hashing

Linear probing: This involves doing a linear probe for the following slot when a collision occurs and continuing to do so until an empty slot is discovered.

Quadratic probing: When a collision happens in this, we probe for the i2-nd slot in the i^{th} iteration, continuing to do so until an empty slot is discovered. In comparison to linear probing, quadratic probing has a worse cache performance. Additionally, clustering is less of a concern with quadratic probing.

Double hashing: In this, you employ a different hashing algorithm, and in the i^{th} iteration, you look for (i * hash 2(x)). The determination of two hash functions requires more time. Although there is no clustering issue, the performance of the cache is relatively poor when using double probing

Sorting Algorithms

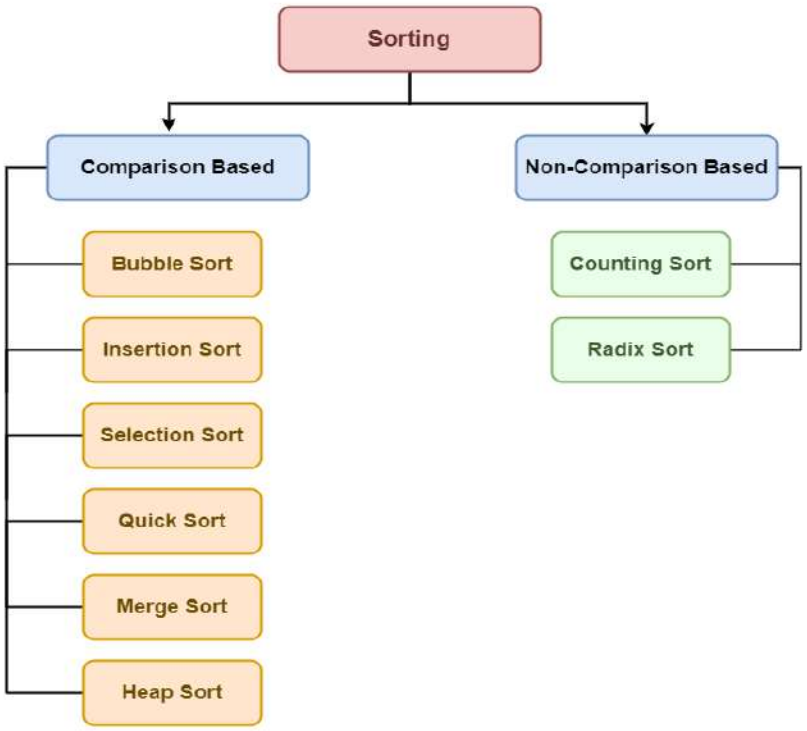
-

A **Sorting Algorithm** is used to rearrange a given array or list of elements in an order. Sorting is provided in library implementation of most of the programming languages.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Selection Sort

-

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.

3. We keep doing this until we get all elements moved to correct position.

Bubble Sort Algorithm

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.
  - We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
  - In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
  - In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Insertion Sort Algorithm

- **Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
  - We start with second element of the array as first element in the array is assumed to be sorted.
  - Compare second element with the first element and check if the second element is smaller then swap them.
  - Move to the third element and compare it with the first two elements and put at its correct position
  - Repeat until the entire array is sorted.

Merge Sort

- **Merge sort** is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

How does Merge Sort work?

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Heap Sort

- **Heap sort** is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

Heap Sort Algorithm

First convert the array into a max heap using **heapify**, Please note that this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1.

- Rearrange array elements so that they form a Max Heap.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.

    - Heapify the remaining elements of the heap.
- Finally we get sorted array.

(1) Binary Tree Traversed

↳ Preorder

↳ Inorder

↳ Postorder

(2) Basic concept of Insertion sort

Put second element and compare with each element in the list and insert in right place. Repeat this with all elements.

At the end list is sorted.

(3) Algorithm insertion and deletion in Binary Tree

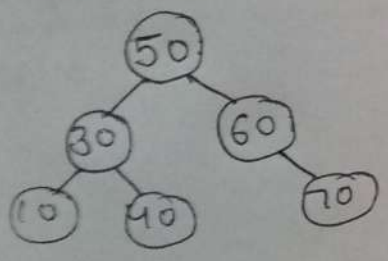
Deletion →

1. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.
4. END.

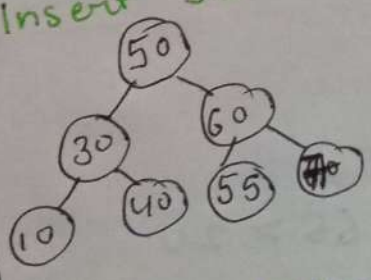
Insertion →

1. If root is NULL, add node and return.
2. else if root > Key, Move left
3. else root < Key, Move right
4. Repeat steps 2 and 3 until reach a leaf node
5. Attach the key as a left or right child based on the comparison with the leaf node's value.
6. END.

Example-

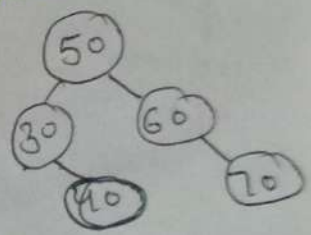


Insert 55



After insertion

Delete 10



After deletion

(4) Hasing and various hasing techniques
↳ DONE (✓)

(5) (b) Heap Sort
↳ DONE

(c) Applications of Binary search tree
↳ left subtree of a node contains only nodes with keys lesser than the node's key
↳ Right subtree of a node contains only nodes with keys greater than the node's key
↳ left and Right subtree each must also be a binary search tree. There must be no duplicates nodes

---

(6) Hashing
↳ DONE (✓)

(7) (c) Quick Sort operation on given numbers
66, 35, 40, 55, 62, 77, 25, 38, 18, 40, 30, 20

Pass (1)

Step1-

66	35	40	55	62	77	25	38	18	40	30	20
0	1	2	3	4	5	6	7	8	9	10	11

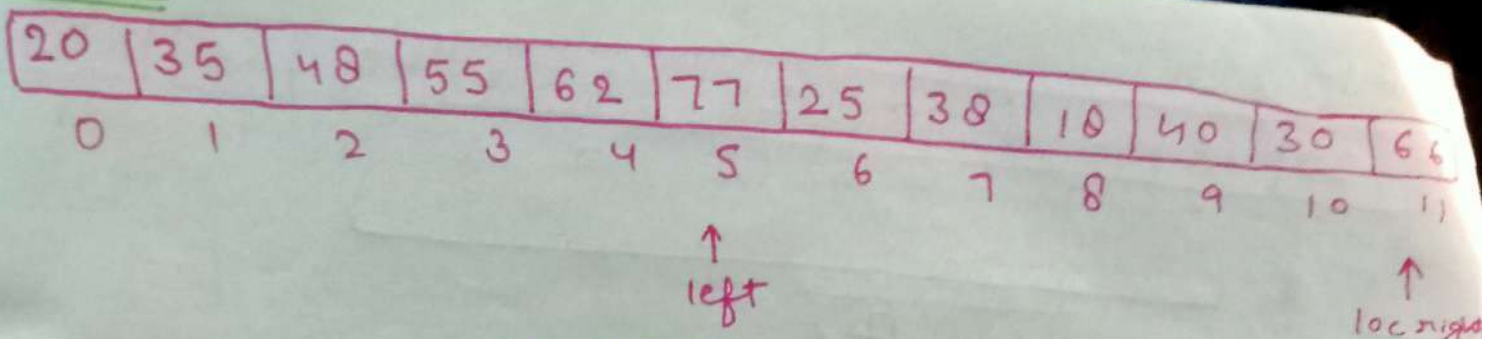
↑ loc, left

↑ right

66 > 20

Interchange a[0] ↔ a[11]

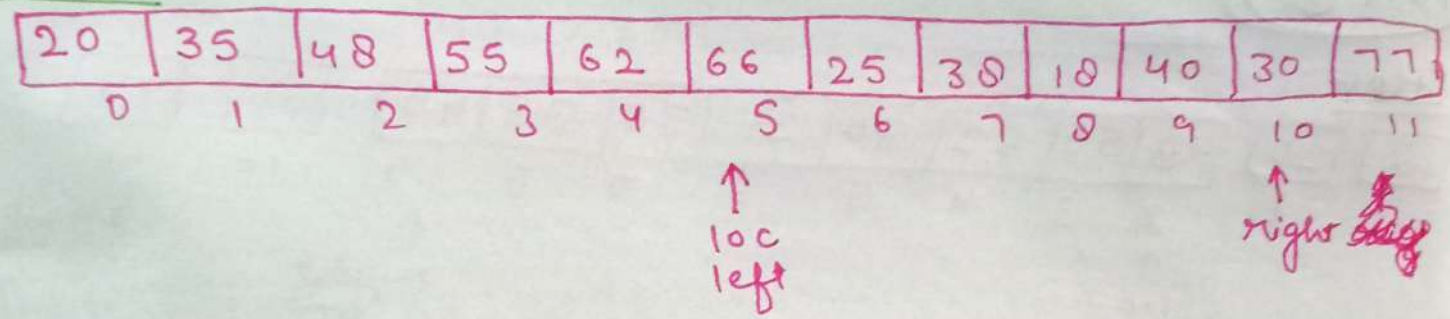
Step 2-



66 < 77

Interchange a[11] ↔ a[5]

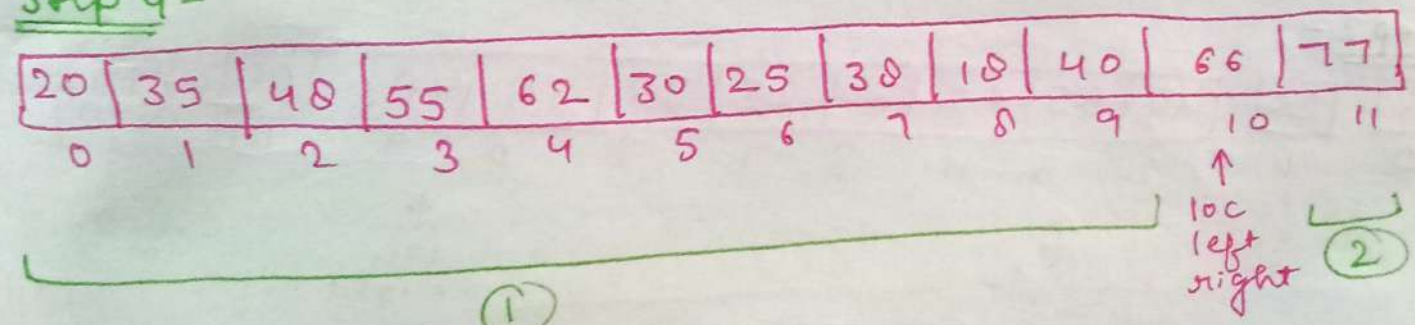
Step 3-



66 > 30

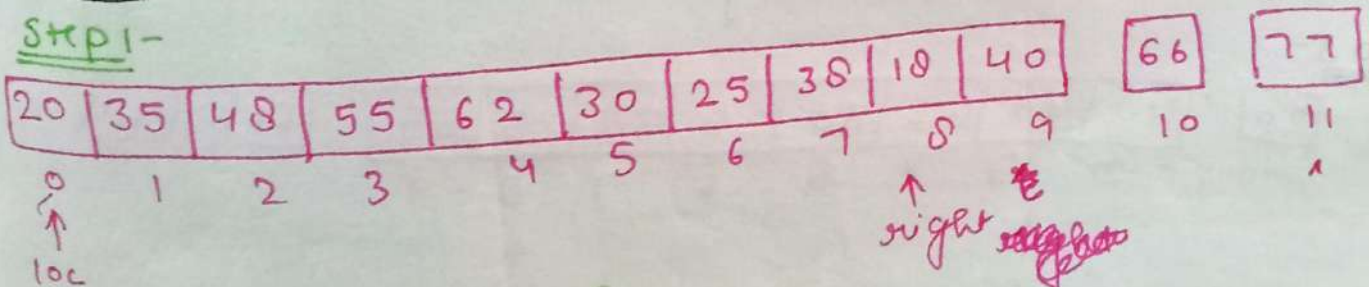
Interchange a[5] ↔ a[10]

Step 4-



Pass ②

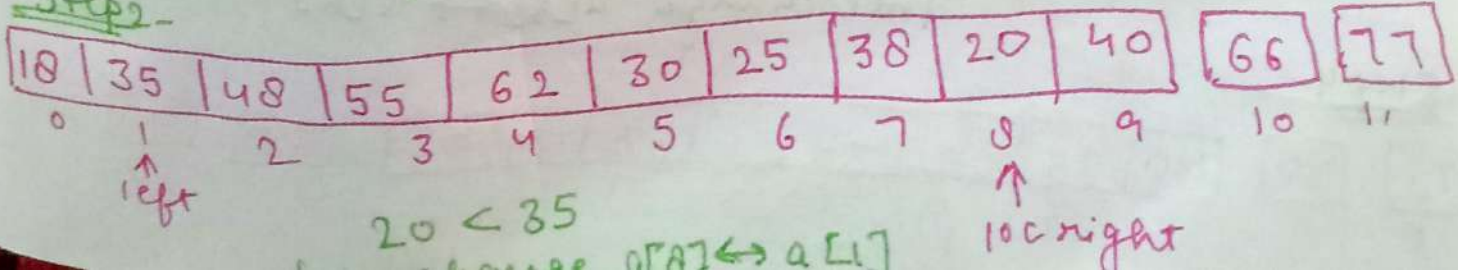
Step 1-



20 > 10

Interchange a[0] ↔ a[9]

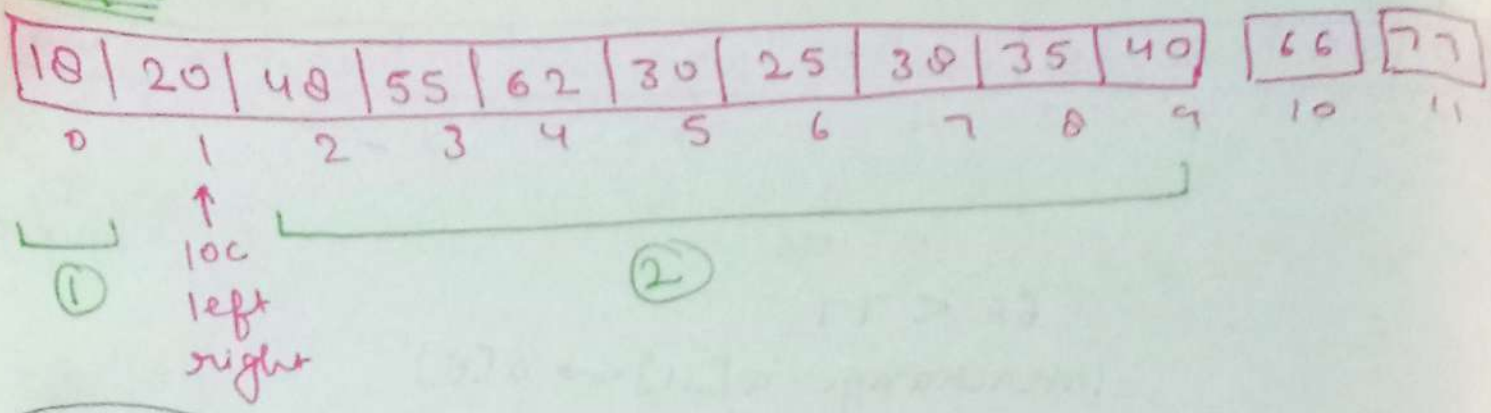
Step 2-



20 < 35

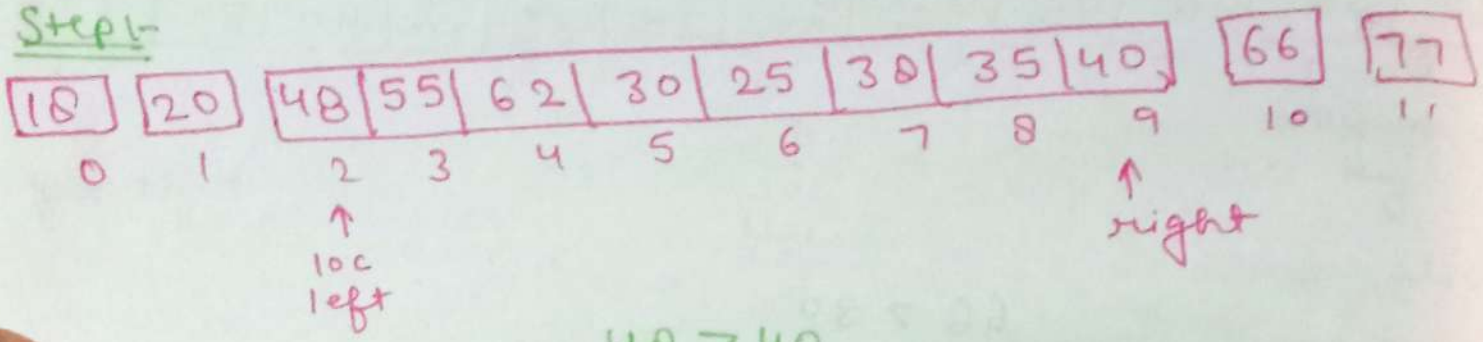
Interchange a[0] ↔ a[1]

Step 3-



Pass ③

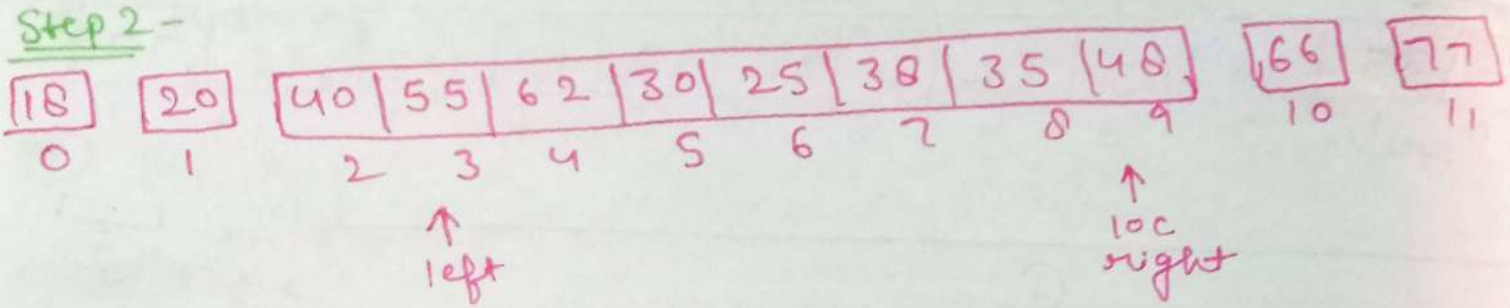
Step 1-



40 > 40

Interchange a[2] ⟷ a[9]

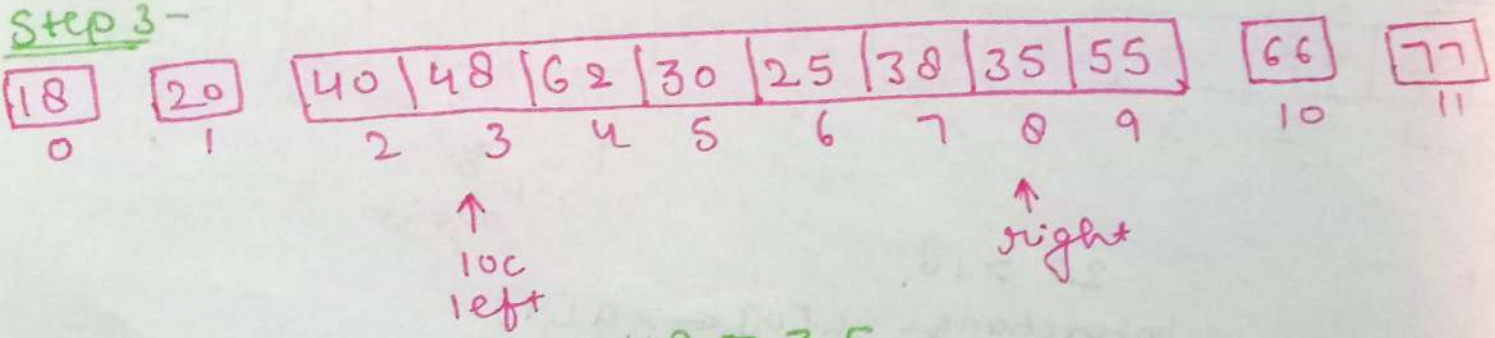
Step 2-



40 < 55

Interchange a[9] ⟷ a[3]

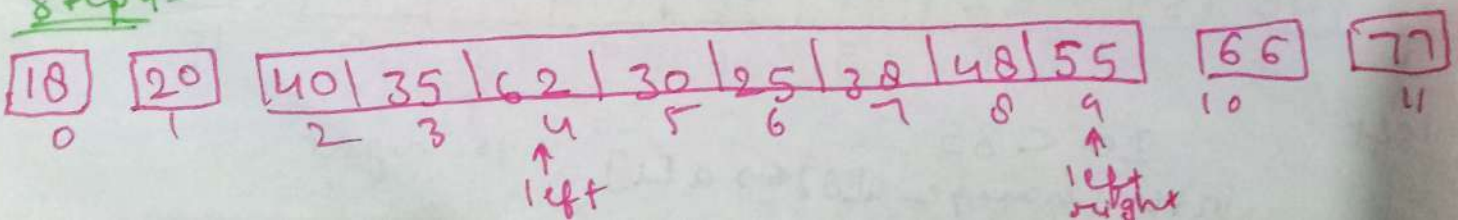
Step 3-



40 > 35

Interchange a[3] ⟷ a[8]

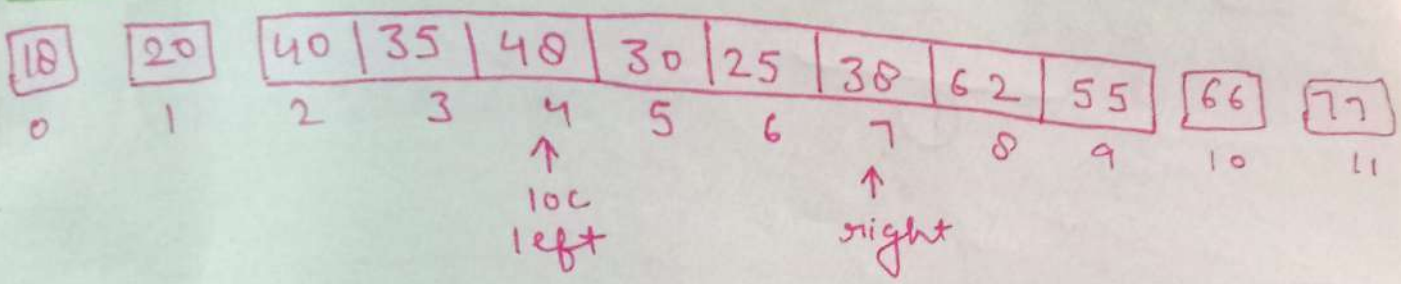
Step 4-



40 < 62

Interchange a[4] ⟷ a[7]

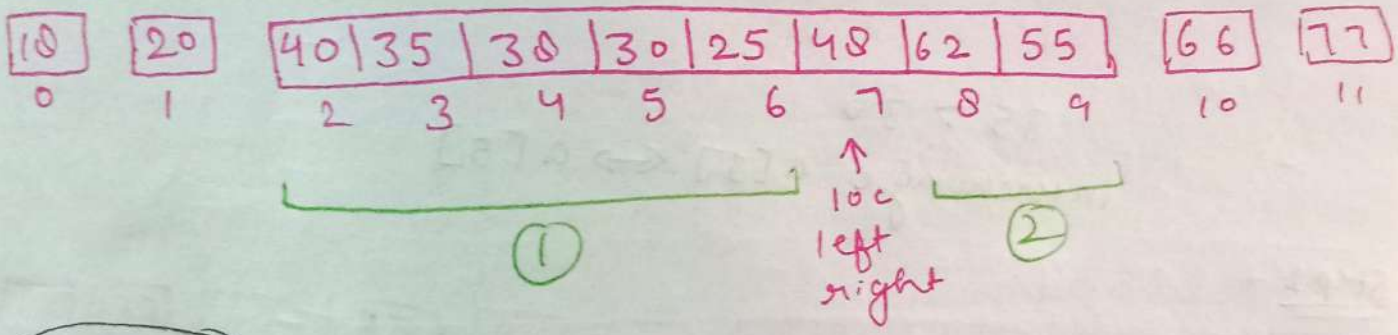
Step 5-



40 > 38

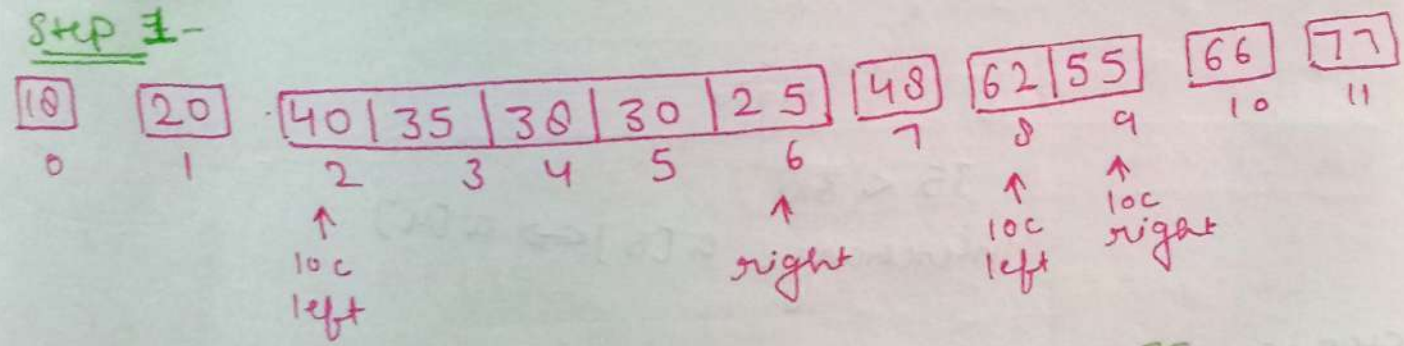
Interchange a[4] ⟷ a[7]

Step 6-



Pass (4)

Step 1-



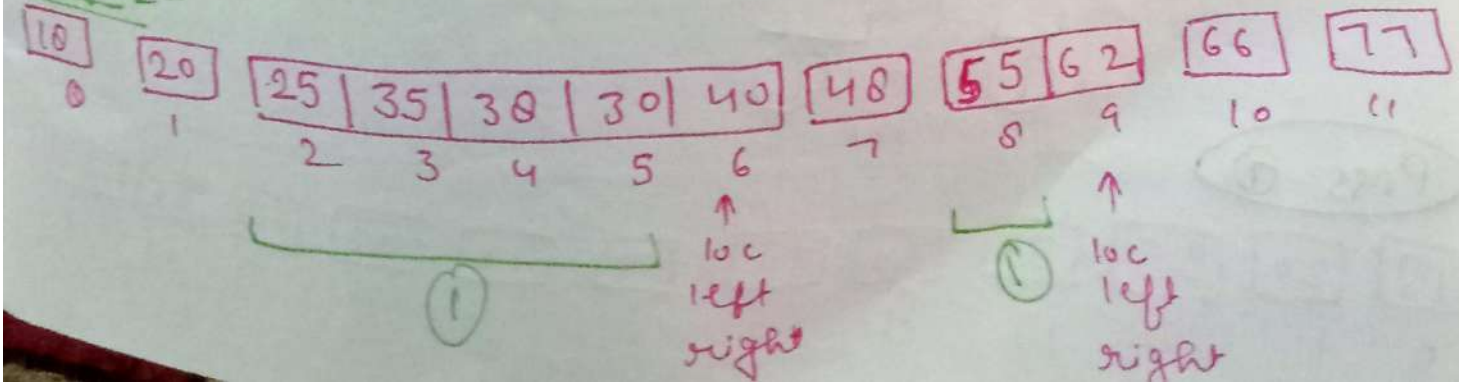
40 > 25

Interchange a[2] ⟷ a[6]

62 > 55

Interchange a[8] ⟷ a[9]

Step 2-



Pass (5)

10	20	25	35	30	30	40	48	55	62	66	77
0	1	2	3	4	5	6	7	8	9	10	11

loc
left
right

(1)

Pass (6)

Step 1 -

10	20	25	35	30	30	40	48	55	62	66	77
0	1	2	3	4	5	6	7	8	9	10	11

loc, left — right

35 > 30

Interchange a[3] ⟷ a[5]

Step 2 -

10	20	25	30	30	35	40	48	55	62	66	77
0	1	2	3	4	5	6	7	8	9	10	11

left — loc, right

35 < 30

Interchange a[5] ⟷ a[4]

Step 3 -

10	20	25	30	35	30	40	48	55	62	66	77
0	1	2	3	4	5	6	7	8	9	10	11

(1) loc, left (1)

right

Pass (7)

10	20	25	30	35	30	40	48	55	62	66	77
0	1	2	3	4	5	6	7	8	9	10	11

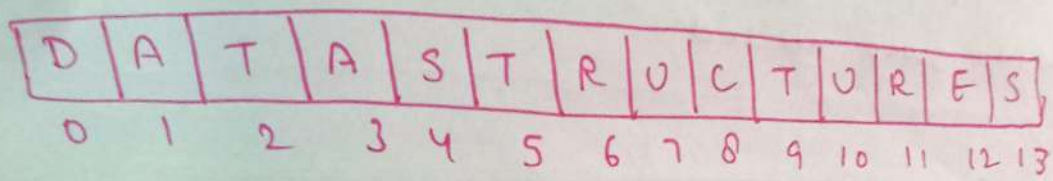
After Quick Sort →

10, 20, 25, 30, 35, 38, 40, 48, 55, 62, 66, 77

(b) Apply Bubble Sort on

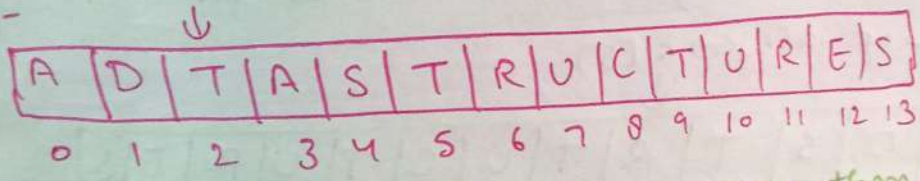
DATA STRUCTURES

Step1-



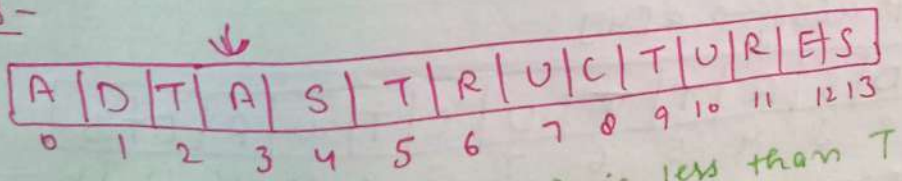
↳ Compare a[1], a[0], A is less than D
↳ Swap a[1] ↔ a[0]

Step2 -

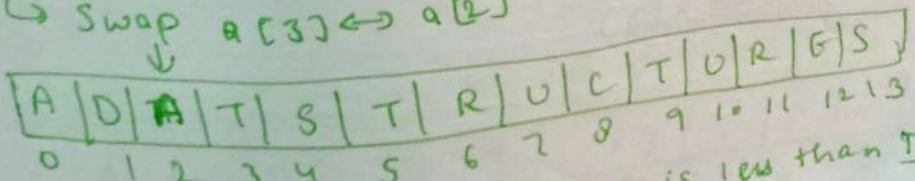


↳ Compare a[2], a[1], T is greater than D
↳ No swap needed.

Step3 -

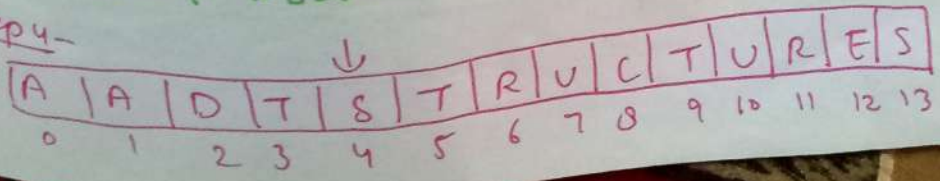


↳ Compare a[3], a[2], A is less than T
↳ Swap a[3] ↔ a[2]



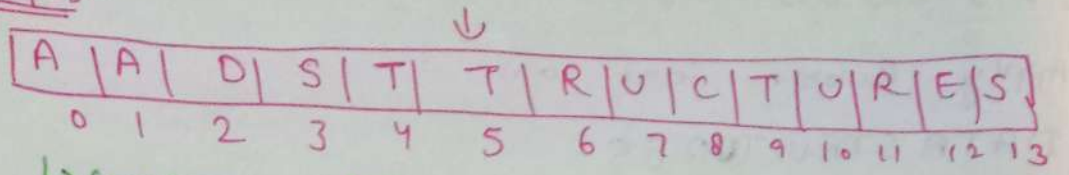
↳ Compare a[2], a[1], A is less than D
↳ Swap a[2] ↔ a[1]

Step4-



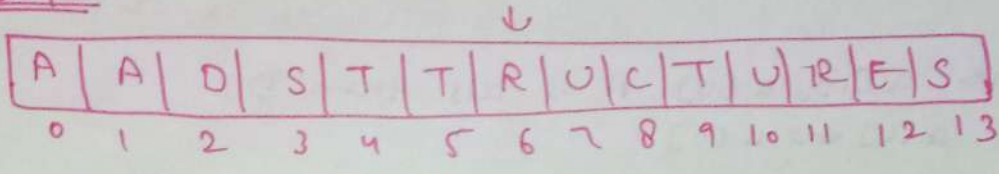
↳ Compare a[4], a[8], S is less than T
↳ Swap a[4] ⇔ a[8]

Step 5 -

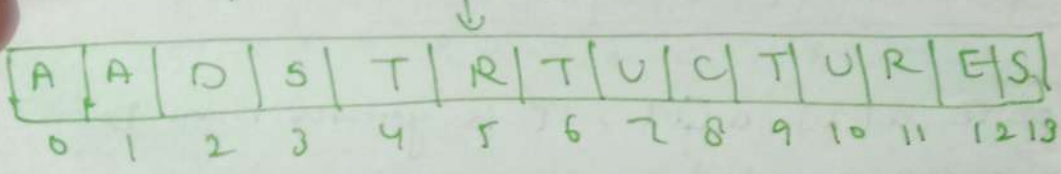


↳ Compare a[5], a[4]
↳ No swap needed

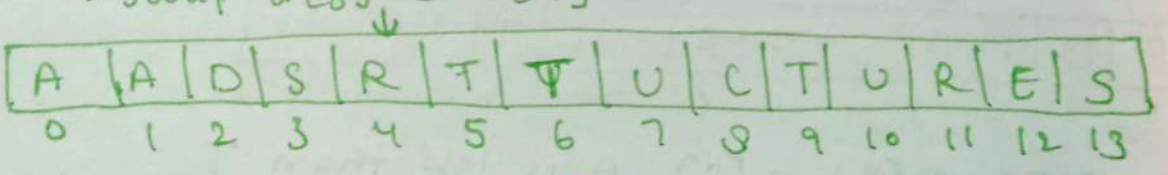
Step 6 -



↳ Compare a[6] ~~&~~ a[5], R is less than T
↳ Swap a[6] ⇔ a[5]

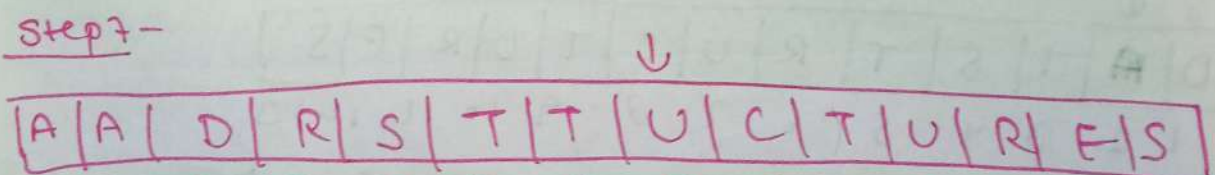


↳ Compare a[5], a[4], R is less than T
↳ Swap a[5] ⇔ a[4]



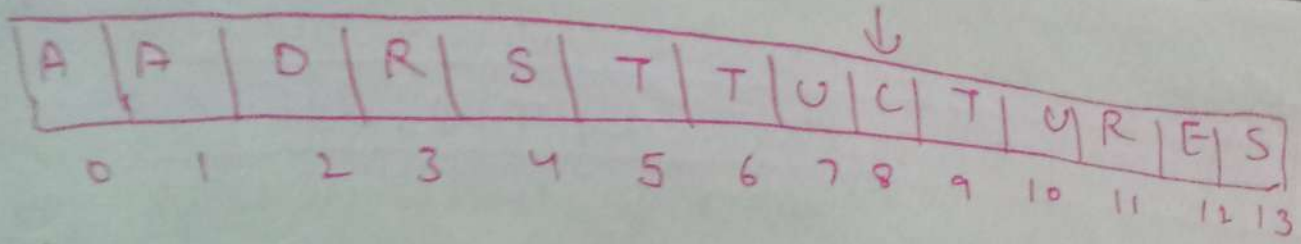
↳ Compare a[4] > a[3], R is less than S
↳ Swap a[4] ⇔ a[8]

Step 7 -



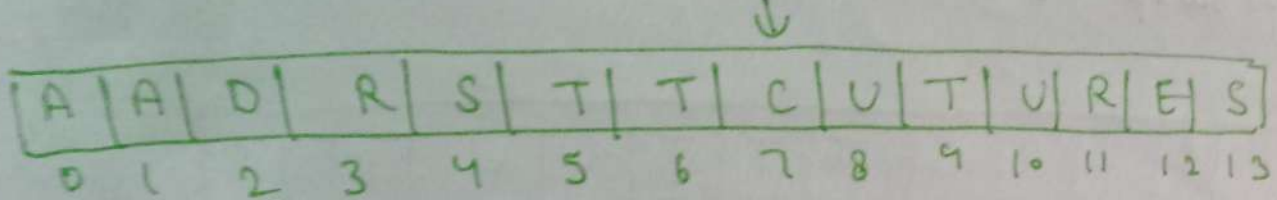
↳ Compare a[7], a[6], U is greater than T
↳ No swap needed

Step 8 -



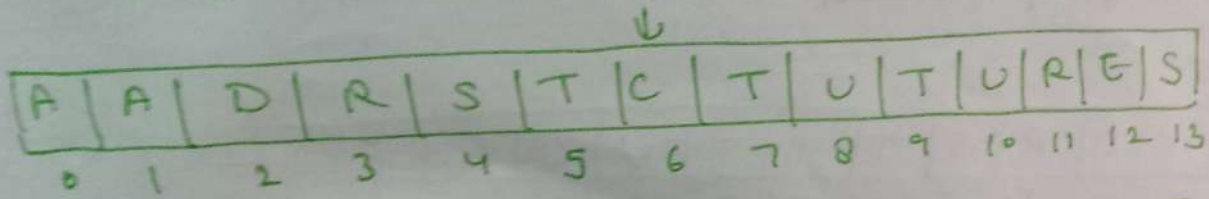
↳ Compare a[8], a[7], C is less than U

↳ Swap a[8] ⟷ a[7]



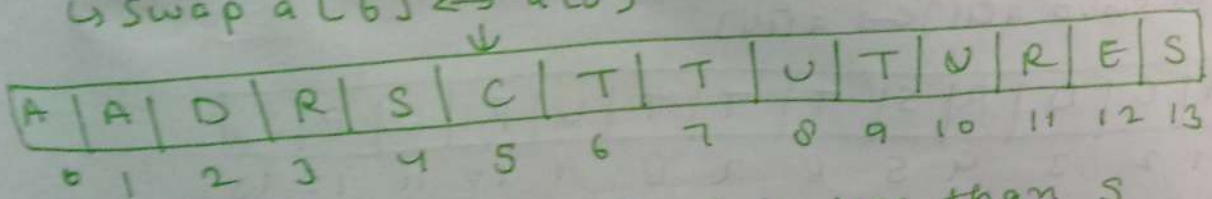
↳ Compare a[7], a[6], C is less than T

↳ Swap a[7] ⟷ a[6]



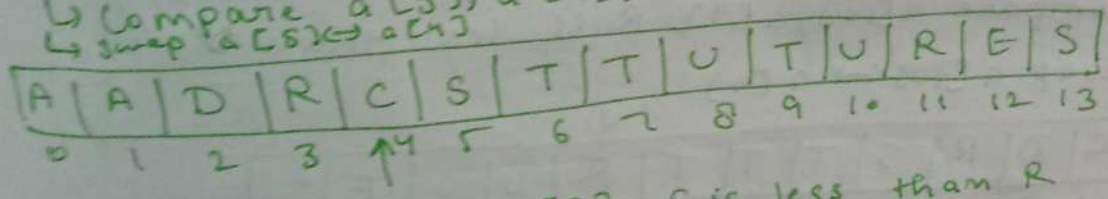
↳ Compare a[6], a[5], C is less than T

↳ Swap a[6] ⟷ a[5]



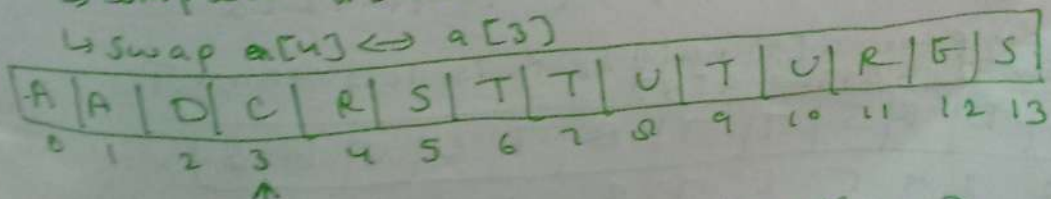
↳ Compare a[5], a[4], C is less than S

↳ Swap a[5] ⟷ a[4]



↳ Compare a[4], a[3], C is less than R

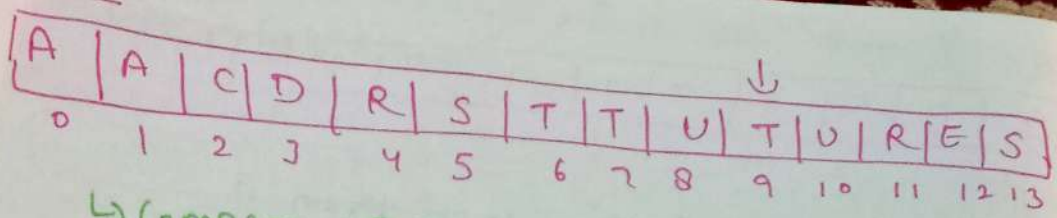
↳ Swap a[4] ⟷ a[3]



↳ Compare a[3], a[2], C is less than D

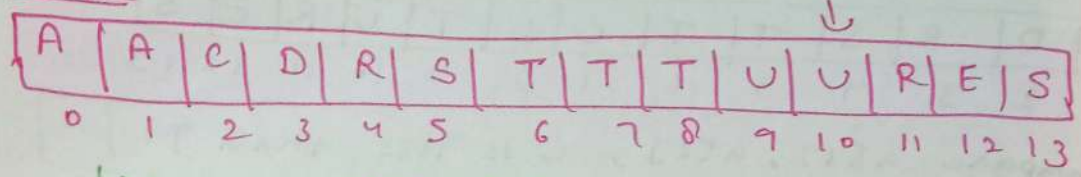
↳ Swap a[3] ⟷ a[2]

Step 9-



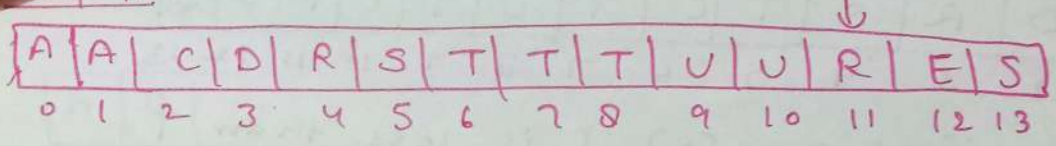
↳ Compare a[9], a[8], T is less than U
↳ swap a[9] ⟷ a[8]

Step 10-

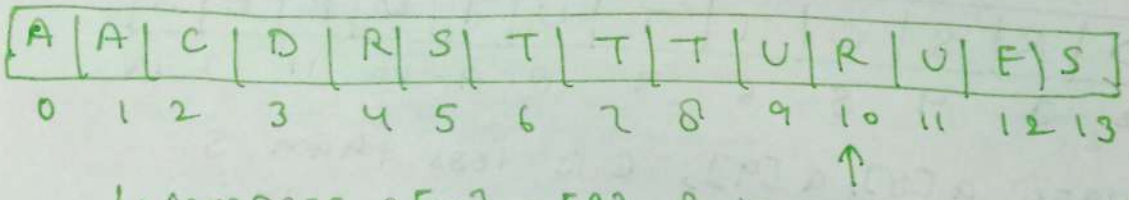


↳ Compare a[10], a[9],
↳ No swap needed

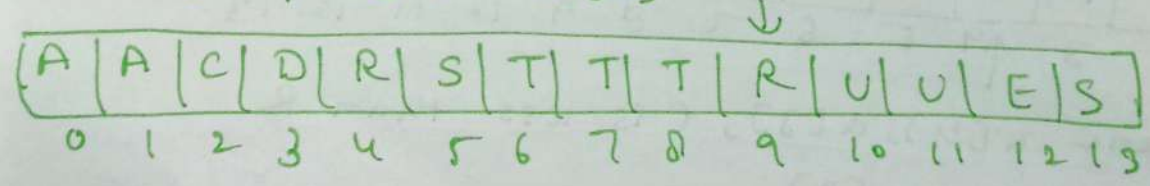
Step 11→



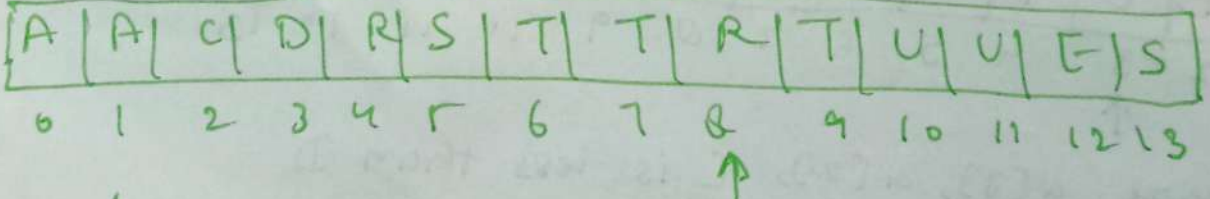
↳ Compare a[11], a[10], R is less than U
↳ swap a[11] ⟷ a[10]



↳ compare a[10], a[9], R is less than U
↳ swap a[10] ⟷ a[9]



↳ Compare a[9], a[8], R is less than T
↳ swap a[9] ⟷ a[8]



↳ Compare a[8], a[7], R is less than T
↳ Swap a[8] ⟷ a[7]

A	A	C	D	R	S	T	R	T	T	U	U	E	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13

↳ Compare a[7], a[6], R is less than T

↳ Swap a[7] ⟷ a[6]

A	A	C	D	R	S	R	T	T	T	U	U	E	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13

↳ Compare a[6], a[5], R is less than S

↳ Swap a[6] ⟷ a[5]

Step 12 -

A	A	C	D	R	R	S	T	T	T	U	U	E	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13

① ↳ Compare a[12], a[11], E is less than U

↳ Swap a[12] ⟷ a[11]

② ↳ Compare a[11], a[10], E is less than U

↳ Swap a[11] ⟷ a[10]

③ ↳ Compare a[10], a[9], E is less than T

↳ Swap a[10] ⟷ a[9]

④ ↳ Compare a[9], a[8], E is less than T

↳ Swap a[9] ⟷ a[8]

⑤ ↳ Compare a[8], a[7], E is less than T

↳ Swap a[8] ⟷ a[7]

⑥ ↳ Compare a[7], a[6], E is less than S

↳ Swap, a[7] ⟷ a[6]

⑦ ↳ Compare a[6], a[5], E is less than R

↳ Swap a[6] ⟷ a[5]

(8) ↳ Compare a[5], a[4], E is less than R
↳ Swap a[5] ⟷ a[4]

Step 13 -

A	A	C	D	E	R	R	S	T	T	T	U	U	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13

(1) ↳ Compare a[13], a[12], S is less than U
↳ Swap a[13] ⟷ a[12]

(2) ↳ Compare a[12], a[11], S is less than U
↳ Swap a[12] ⟷ a[11]

(3) ↳ Compare a[11], a[10], S is less than T
↳ Swap a[11] ⟷ a[10]

(4) ↳ Compare a[10], a[9], S is less than T
↳ Swap a[10] ⟷ a[9]

(5) ↳ Compare a[9], a[8], S is less than T
↳ Swap a[9] ⟷ a[8]

After BUBBLE SORT →

A, A, C, D, E, R, R, S, S, T, T, T, U, U,

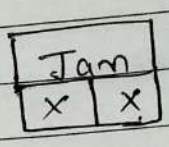
DONE ✓

8. Explain the properties of B-Tree

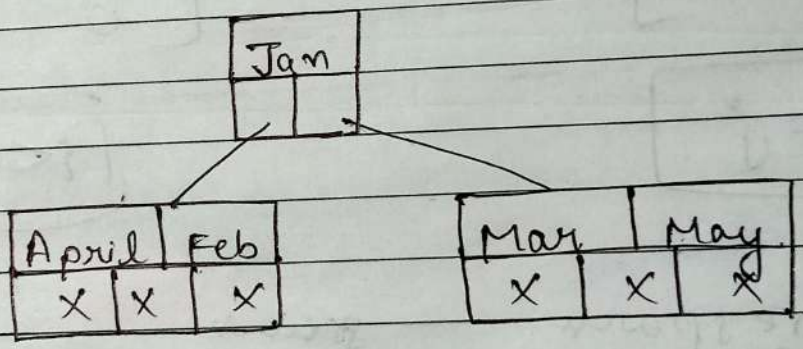
Jan, Feb, Mar, April, May, June, July, Aug, Sep, Oct, Nov, Dec.

B-tree order 3 → Min → 2 children
Max → 3 children

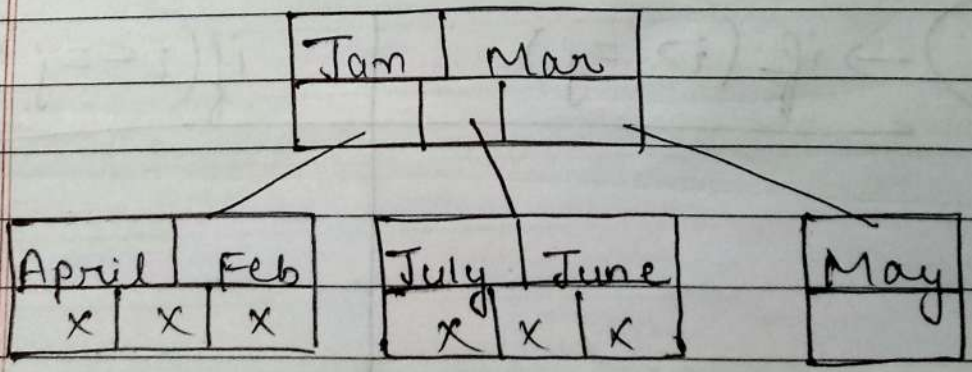
→ (1) Insert Jan



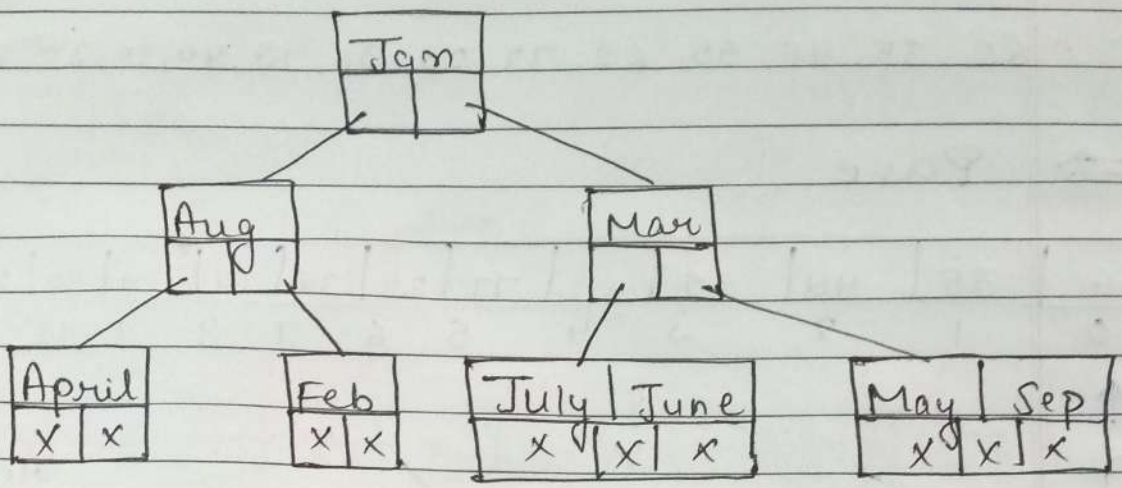
(2) Insert Feb, Mar, April, May



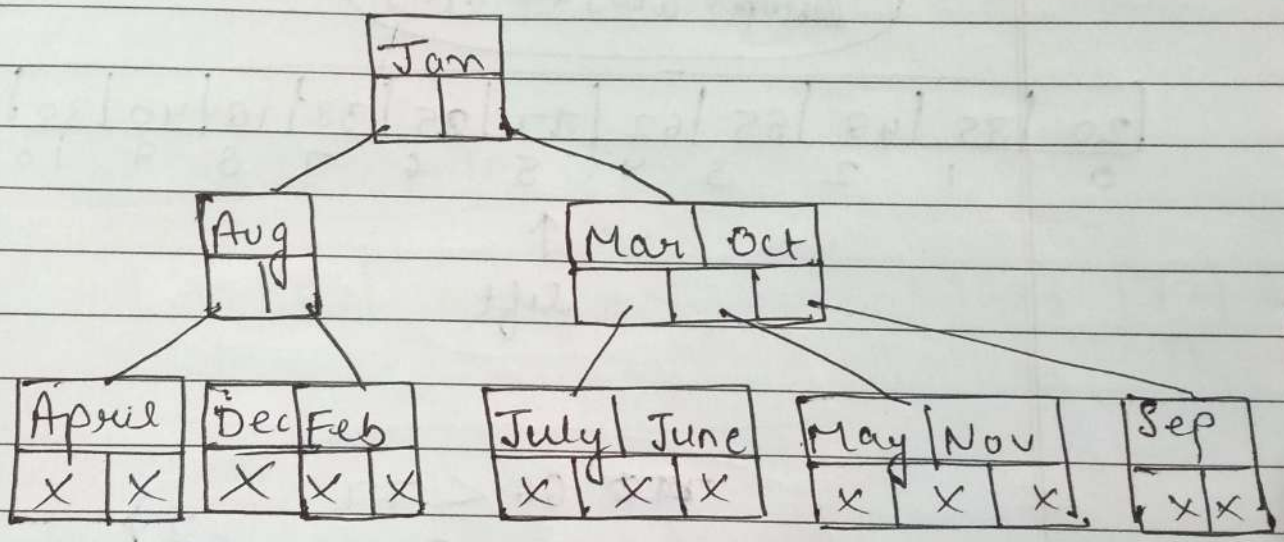
(3) Insert June, July



④ Insert Aug, Sep

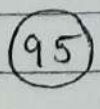


⑤ Insert Oct, Nov, Dec

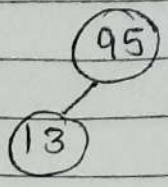


(9) (a) Create Heap tree

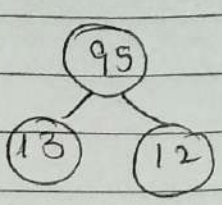
95, 13, 12, 71, 96, 10, 62, 43, 35, 38

(1) 

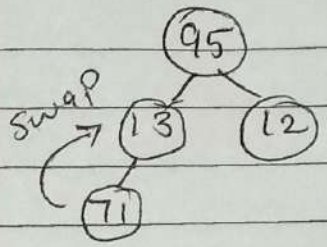
Insert 95

(2) 

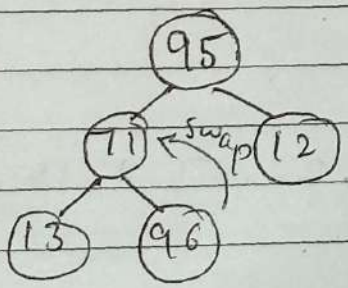
Insert 13

(3) 

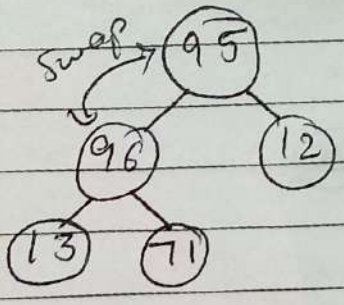
Insert 12

(4) 

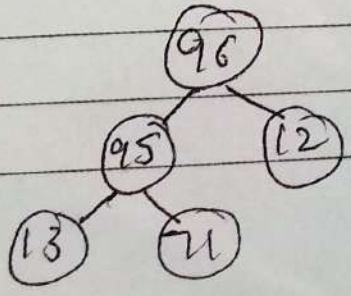
Insert 71
Interchange 71 ↔ 13

(5) 

Insert 96
Interchange 96 ↔ 71

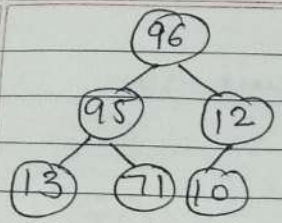
(6) 

Interchange 96 ↔ 95



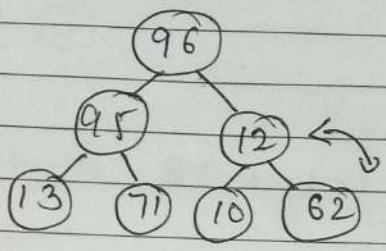
Date:
Page:

(7)

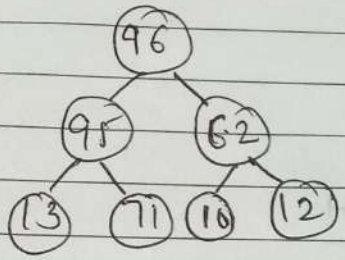


Insert 10

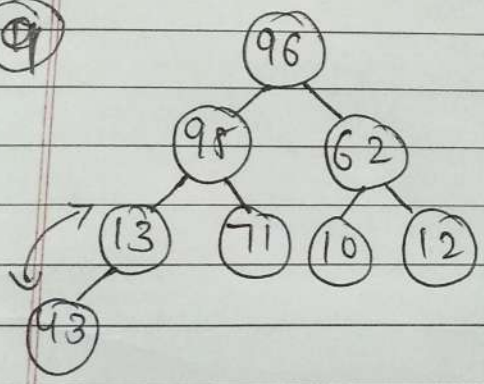
(8)



Insert 62
Interchange 62 ↔ 12

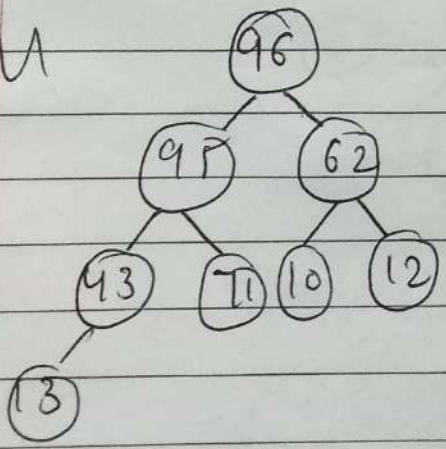


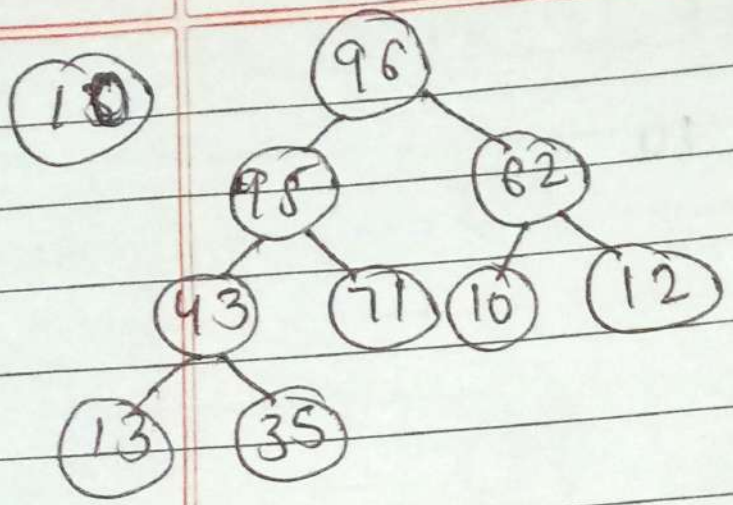
(9)



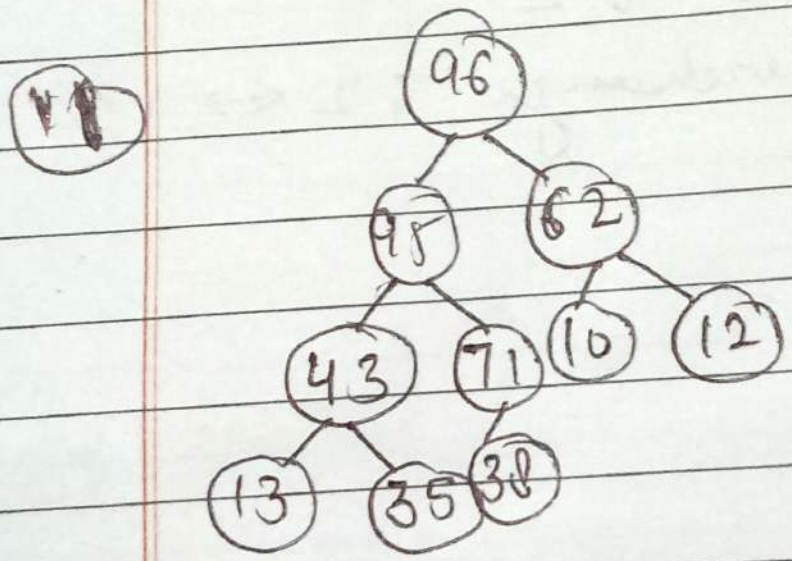
Insert 43
Interchange 43 ↔ 13

(10)





Insert 35

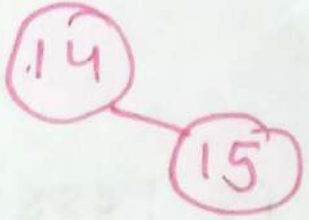


Insert 38

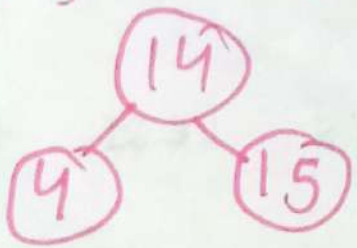
(b) 14, 15, 4, 9, 7, 10, 3, 5, 16, 4, 20, 17, 9, 14, 5

Step1- Insert 14

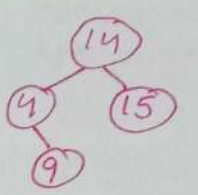
Step2 Insert 15, 15 > 14 → Move right



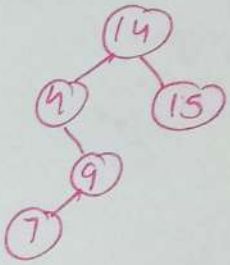
Step3 Insert 4, 4 < 14 → Move left



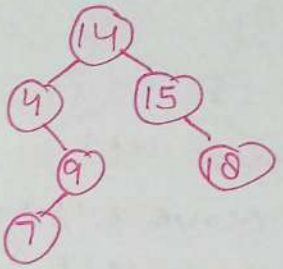
Step 4- Insert 9, 9<14 → Move left
9>4 → Move right



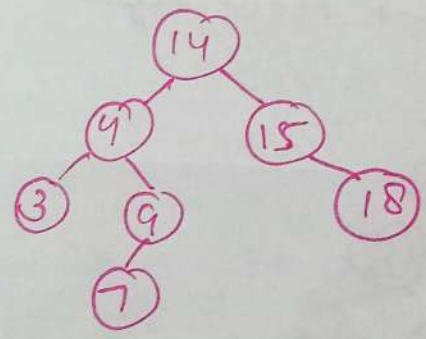
Step 5- Insert 7, 7<14 → Move left
7>4 → Move right
7<9 → Move left



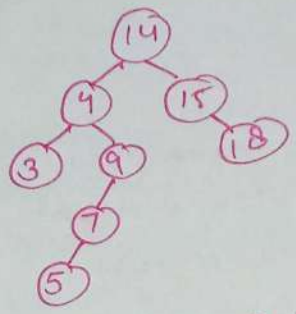
Step 6- Insert 18, 18>14 → Move right
18>15 → Move right



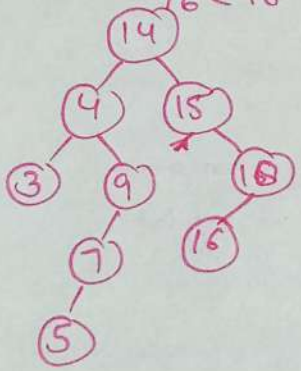
Step 7- Insert 3, 3<14 → Move left
3<4 → Move left



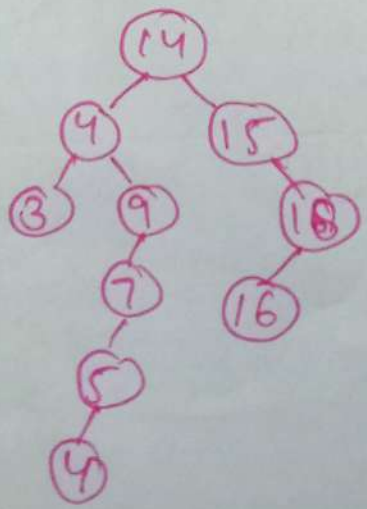
Step 8- Insert 5, 5 < 14 → Move left
5 > 4 → Move right
5 < 9 → Move left
5 < 7 → Move left



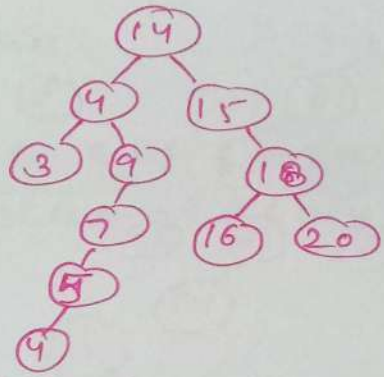
Step 9- Insert 16, 16 > 14 → Move right
16 > 15 → Move right
16 < 18 → Move left



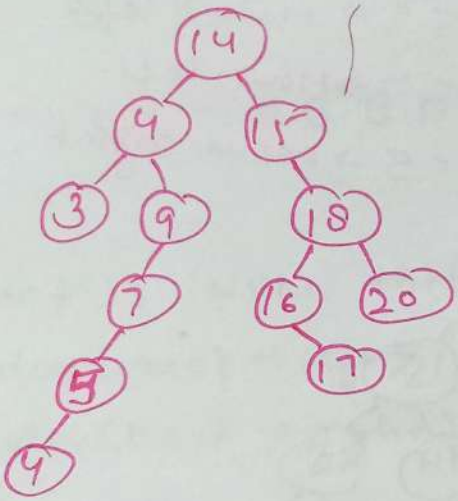
Step 10- Insert 4, 4 < 14 → Move left
4 = 4 → Move right
4 < 9 → Move left
4 < 7 → Move left
4 < 5 → Move left



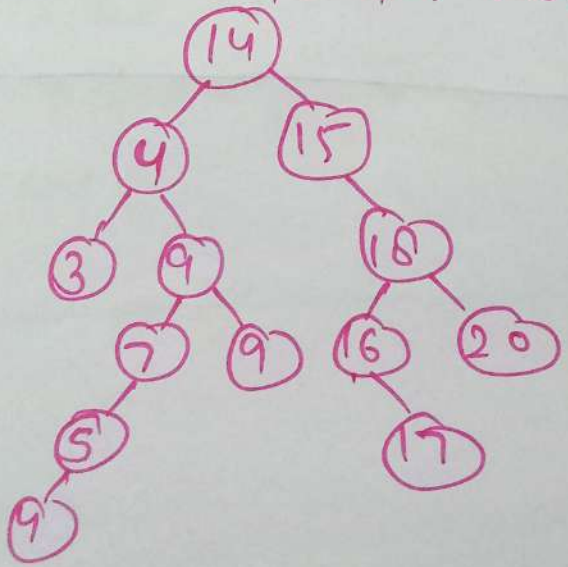
Step 11- Insert 20, 20 > 14 → Move right
20 > 15 → Move right
20 > 16 → Move right



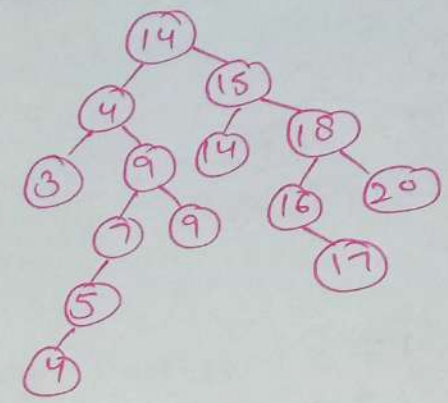
Step 12- Insert 17, ~~17~~ > 14 → Move right
17 > 15 → Move right
~~17~~ < 18 → Move left
17 > 16 → Move right



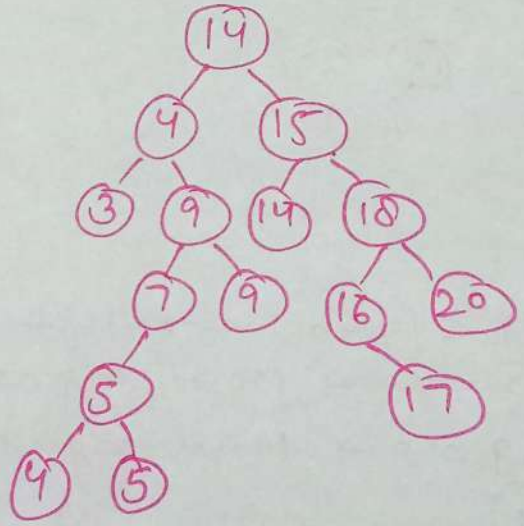
Step 13- Insert 9, 9 < 14 → Move left
9 > 4 → Move right
9 = 9 → Move right



Step 14- Insert 14, 14 = 14 → Move right
14 < 15 → Move left



Step 15- Insert 5, 5 < 14 → Move left
5 > 4 → Move right
5 < 9 → Move left
5 < 7 → Move left
5 = 5 → Move right



---

(10) Hashing.
↳ DONE (V)

(11) Binary Tree
↳ DONE (V)

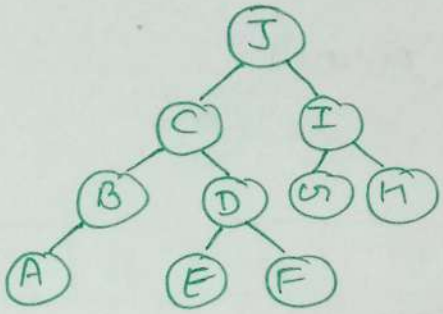
(12)

(13)

(12) Heap sort
↳ DONE (✓)

(13) Binary Tree
↳ DONE (✓)

Inorder - A B C E D F J G I H
Preorder - J C B A D E F I G H



In postorder → A B E F D C G H I J

Algo →

1. If root is NULL then return
2. Postorder (root → left)
3. Postorder (root → right)
4. print root → data

(14) Merge sort
↳ DONE (✓)

(15) Threaded Binary Tree
↳ DONE (✓)

(16) B+ tree
↳ DONE ✓

(17) Hashing
↳ DONE ✓

(18) Selection sort
↳ DONE ✓

(19) (a) Hashing / Hashing function
↳ DONE ✓

(b) Binary tree
↳ DONE ✓

(c) Search in binary tree
↳ DONE ✓

(20) Tree
↳ DONE ✓

Types of Tree
↳ DONE ✓

Traversal of Binary Tree
↳ DONE ✓

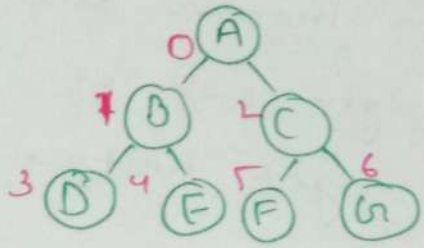
Algorithm
↳ DONE ✓

Example
↳ DONE ✓

(21) Heap sort
↳ DONE (✓)

(22) Properties of B+ tree
↳ DONE (✓)

(23) Represent Binary Tree in memory using array



A	B	C	D	E	F	G
0	1	2	3	4	5	6

The root node of the binary tree is stored at the first position of the array, and its left and right children are stored at the second and third positions, respectively.

The remaining nodes are stored in the same way, with the children of each node stored in consecutive positions in the array.

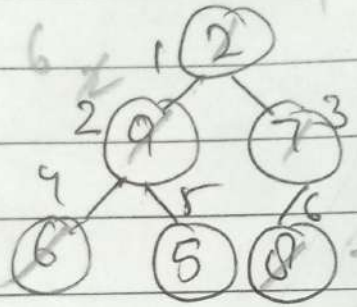
(24) Hashing
↳ DONE (✓)

(25) Perform Heap sort Descending order

K = 2, 9, 7, 6, 5, 8

Step 1 ⇒ Given data 2, 9, 7, 6, 5, 8

Make a binary tree and give indexing

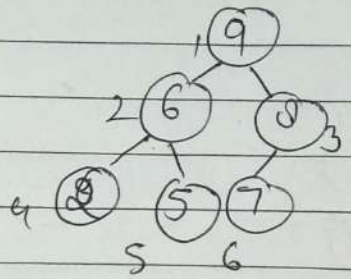


In array form

2	9	7	6	5	8
1	2	3	4	5	6

Step 2- dowheap process up to 6

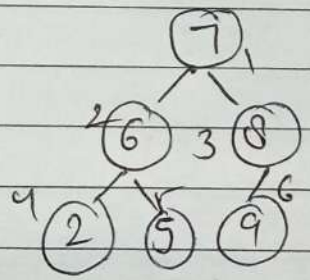
Interchange, 6 ↔ 3 (6 → 7)
Interchange, 2 ↔ 1 (9 → 2)
Interchange, 4 ↔ 2 (6 → 2)



In array

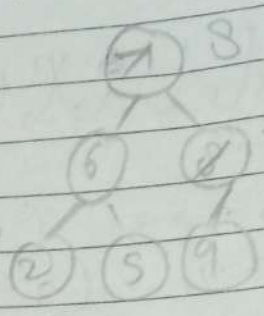
9	6	8	2	5	7
1	2	3	4	5	6

Step 3- Swapping 1 ↔ 6

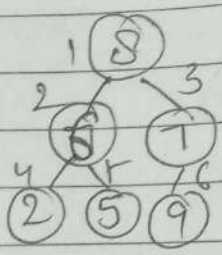


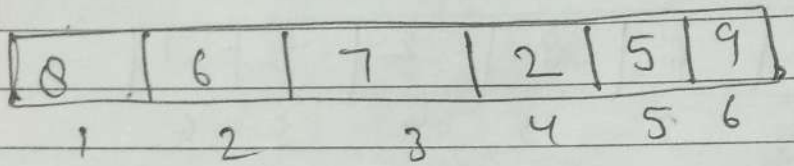
7	6	8	2	5	9
1	2	3	4	5	6

Step 4- Dowheap up to 5

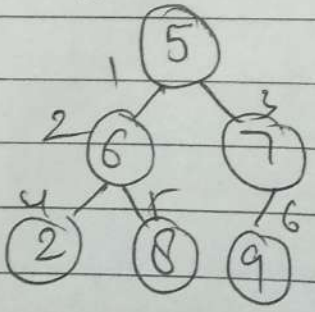


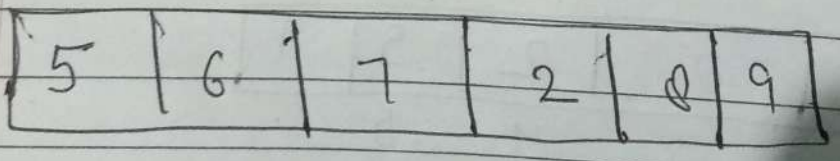
Interchange, 1 ⟷ 3 (7 → 8)



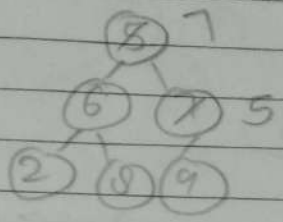


Step 5- Swapping 1 → 5

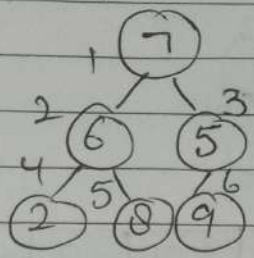




Step 6- Downheap up to 4

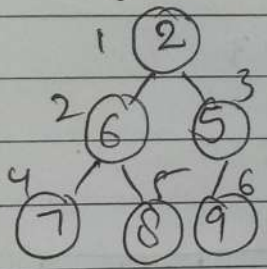


Interchange, 3 ↔ 1 (7 → 5)



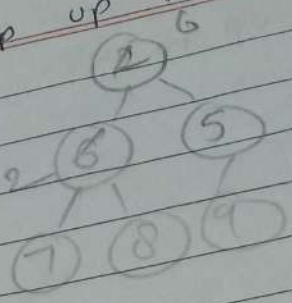
7	6	5	2	8	9
1	2	3	4	5	6

Step 7- Swapping 1 → 4

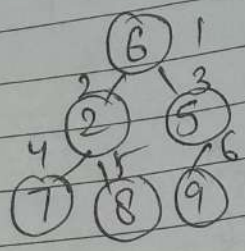


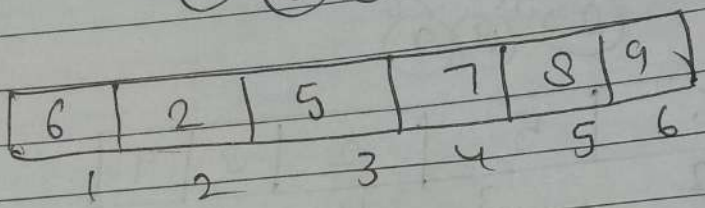
2	6	5	7	8	9
1	2	3	4	5	6

Step 8- Dowheap up to 3

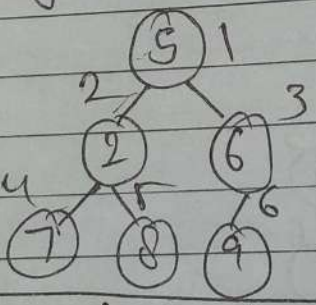


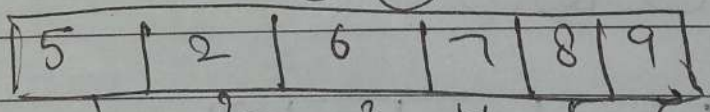
Interchange, 2 ↔ 1 (6 → 2)



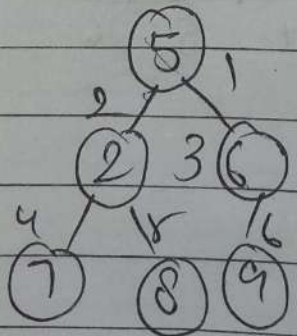


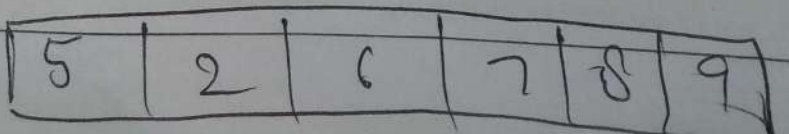
Step 9- Swapping 1 → 3



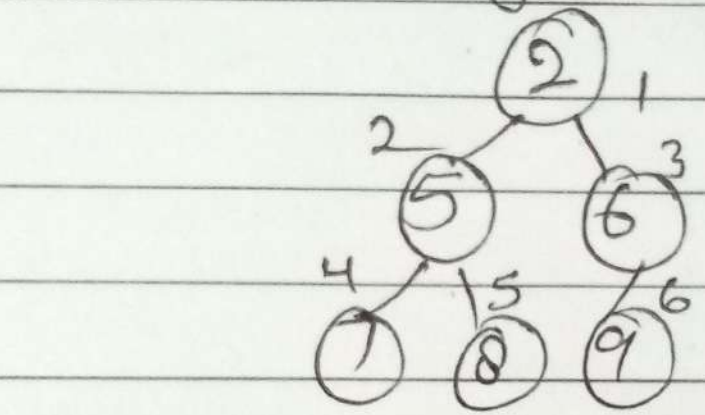


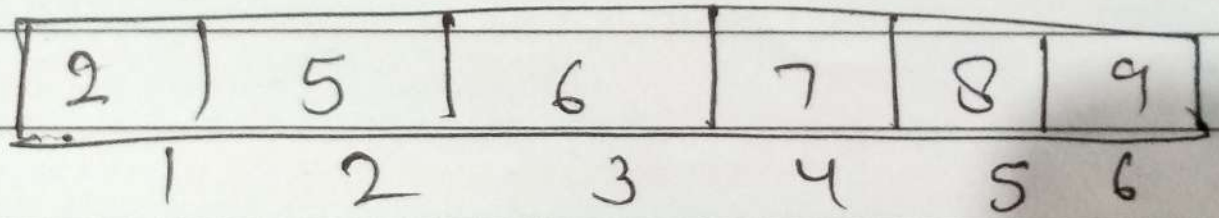
Step 10- dowheap up to 2





Step 11- Swapping





Arrang in descending order

9, 8, 7, 6, 5, 2

(26) Inorder Traversal → D, B, E, A, F, C

Preorder Traversal → A, B, D, E, C, F

Step 1- See first key in preorder that is A insert A as a root node in tree. then see in inorder and divide the left and right side of A as left subtree or right subtree

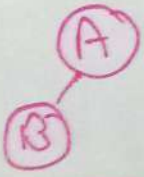
I → D, B, E, A, F, C

Pr → A, B, D, E, C, F

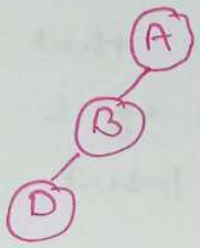
Step 2- Now, See next key in preorder, that is B. then see in inorder w/c side of A belongs to B that is left insert B in left of A root node and divide the left and right side of B as left subtree or right subtree

I → D, B, E, A, F, C

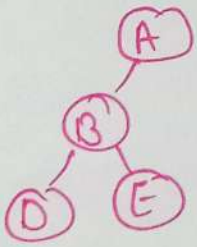
Pr → A, B, D, E, C, F



Step 3- See, next key in preorder, that is D, then see in inorder w/c side of A belongs of D that is left. Now see w/c side of B belongs of D that is also left. Insert D



Step 4- See next key in preorder, that is E, then see in inorder w/c side of A belongs to E that is left. Now see w/c side of B belongs to E that is right. Insert E

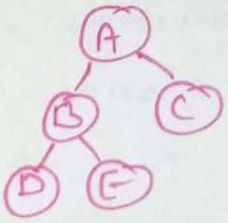


Step 5- See next key in preorder, that is C, then see in inorder w/c side of A belongs to C that is right.

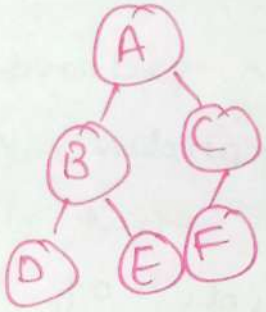
Insert C and divide the left side of C as a leaf node

I = D, B, E, A, F, C

Pr = A, B, D, E, C, F



Step 6- See next key in preorder that is F. Now see in inorder w/c side of A belongs to F that is right. Again see the side of C belongs to F that is left. Insert F



(27) Collision resolution techniques
↳ DONE (✓)

(28) B-Trees
↳ DONE (✓)