



SAPTHAGIRI NPS
UNIVERSITY

UNMATCHED EXCELLENCE, UNLIMITED POTENTIAL

School of Engineering and Technology

(COMMON TO B.TECH CSE/ AIML/AI&DS/CYBER SECURITY/DATA SCIENCE)

Fundamentals of Data Structures (24BTELY106)

Lab Manual




COURSE CO-ORDINATOR

Dr. SWETHA K B

ASSOCIATE PROFESSOR,

Department of Computer Science and Engineering

Sapthagiri NPS University, Bangalore 560057

	Program	B. Tech Computer Science and Engineering				Program Code		
	Course	Fundamentals of Data Structure				Course Code	24BTELY106	
	Semester	I	Credits	4	Theory	Practical	Total Hours	60
					3	2		
COURSE OBJECTIVES:								
1. To impart knowledge of fundamental data structures and how they are implemented. 2. To comprehend the significance of data structures in the context of creating effective software. 3. To gain the ability to use the right data structures for addressing problems.								

COURSE OUTCOMES:
At the end of Course Students will be, 1. Determine the various C programming language data structures. 2. Evaluate how data structures are used to solve problems. 3. Put data structures into practice with the C programming language. 4. Assess the control structures, algorithmic notations, array representation as a stack, insertion and deletion into a singly linked list, operation on a queue, and tree height balancing.

PART – A

1. Write a menu driven C program to perform the following string operations using pointers (i) String Length (ii) String Concatenation (iii) string reverse
2. Write a program to find the location of a given number using Linear search.
3. Write a C program to search for an element in an array using Binary search.
4. Write a C program to sort a list of N elements using Selection Sort Algorithm.
5. Write a C program to construct a singly linked list and perform insertion at the beginning, deletion at the specified position and display the elements.
6. Write a C program to demonstrate the working of stack using arrays.
7. Write a C program for Towers of Hanoi problem using Recursion.
8. Write a C program to create and traverse a binary search tree.

PART – B

1. Write a C program to demonstrate the following operations in an array
(i) insertion of an element (ii) deletion of an element (iii) traversal
2. Write a C program to sort a list of elements using Bubble sort algorithm
3. Write a C program to sort a list of N elements using Insertion Sort Algorithm.
4. Write a C program to simulate the working of Linear Queue using a linked list.
5. Write a C program to search for a given element in a sorted linked list.
6. Write a C program to implement stack operations using Linked list.
7. Write a C program to create a directed graph using Adjacency Matrix.

1. Write a menu driven C program to perform the following string operations using pointers**(i) String Length (ii) String Concatenation(iii) string reverse****(i)String Length:****Aim:**

To write a C program to find the length of the string.

In C programming language, you can calculate the length of a string using pointers. The idea is to start from the beginning of the string and keep incrementing the pointer until you reach the end of the string, which is marked by the null terminator character ('').

Algorithm:

- 1 Start the program
- 2 Read the Value
- 3 Use the while loop and test the condition that given_string!='\0'
- 4 Use the length value to display the length
- 5 Stop the program

C program to find length of string

```
#include <stdio.h>
```

```
int string_length(char* given_string)
{
    int length = 0;
    while (*given_string != '\0')
    {
        length++;
        given_string++;
    }

    return length;
}
```

```
int main()
```

```
{  
    char given_string[] = &quot;Datastructure&quot;;  
  
    printf(&quot;Length of the String: %d&quot;,,  
        string_length(given_string));  
    return 0;  
}
```

Output:

Length of the string : 13

(ii)Program to concatenate two strings**Aim:**

To write a C program to concatenate two strings.

The concatenation of strings is a process of combining two strings to form a single string. If there are two strings, then the second string is added at the end of the first string.

Algorithm:

1. Start the program
2. Read the strings
3. Use the while loop to get the first sentence and use the second while loop for second sentence and increment the counter str1.
4. Now assign the sentence str2 to str1 until the string reaches the condition str1!=0 and the sentences are concatenated
- 5.Display the result
- 6.Stop the program

```
#include <stdio.h>
```

```
void concatenate(char *str1, char *str2)
```

```
{
```

```
while (*str1)
```

```
{  
  
str1++;  
  
}  
  
while (*str2)  
  
{  
  
*str1 = *str2;  
  
str1++;  
  
str2++;  
  
}  
  
*str1 = '\\0';  
  
}  
  
int main()  
  
{  
  
char string1[100], string2[50];  
  
printf("Enter the first string:\\n");  
  
gets(string1);  
  
printf("Enter the second string:\\n");  
  
gets(string2);  
  
concatenate(string1, string2);  
  
printf("Concatenated string: %s\\n", string1);  
  
return 0;  
  
}
```

Output:

Enter the first string:

Hello

Enter the second string:

World

Concatenated string: HelloWorld

Program to reverse a string

Aim:

To write a C program to reverse a String

We will use here two pointers, one is start pointer and another is end pointer. and by swapping the character we will proceed to achieve, reverse the characters similar to what we have done in the first method.

Algorithm:

1. Start the program
2. Read the Value
3. Declare the reverseString () function.
4. Read the String
5. Use for loop to reverse the string
6. Display the result
7. Stop the program

```
#include <stdio.h>
#include <string.h>
```

```
void reverseString(char* str)
{
    int l, i;
    char *begin_ptr, *end_ptr, ch;
```

```
    l = strlen(str);
    begin_ptr = str;
    end_ptr = str + l - 1;
    for (i = 0; i < (l - 1) / 2; i++)
    {

        ch = *end_ptr;
        *end_ptr = *begin_ptr;
        *begin_ptr = ch;
        begin_ptr++;
        end_ptr--;
    }
}

int main()

{

    char str[100] = "Datastructure";
    printf("Enter a string: %s\n", str);
    reverseString(str);
    printf("Reverse of the string: %s\n", str);
    return 0;
}
```

Output:

Enter a string: Datastructure
Reverse of the string: erutcurtsataD

2. Write a program to find the location of a given number using Linear search.**AIM:**

To write a C program to find the location of a given number using Linear search

Linear Search is a sequential searching algorithm in C that is used to find an element in a list.

- We can implement linear search in C to check if the given element is present in both random access and sequential access data structures such as arrays, linked lists, trees, etc.
- Linear Search compares each element of the list with the key till the element is found or we reach the end of the list.

Algorithm:

1. Start the program
2. Declare the linearSearch() function.
3. Read the value
4. Use for loop to find the size of array
5. If the index== -1 means the number is not found otherwise the number is present
6. Display the result
7. Stop the program

```
#include <stdio.h>
int linearSearch(int* arr, int size, int key)
{
    for (int i = 0; i < size; i++)
    {
        if (arr[i] == key)
        {
            return i;
        }
    }
    return -1;
}

int main()
{
```

```
int arr[10] = { 3, 4, 1, 7, 5, 8, 11, 42, 3, 13 };
int size = sizeof(arr[0]);
int key = 4;
int index = linearSearch(arr, size, key);
if (index == -1)
{
    printf("The element is not present in the arr.");
}
else
{
    printf("The element is present at arr[%d].", index);
}

return 0;
}
```

Output:

The element is present at arr[1].

3. Write a C program to search for an element in an array using Binary search**Aim:**

To write a C program to search for an element in an array using Binary search

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:

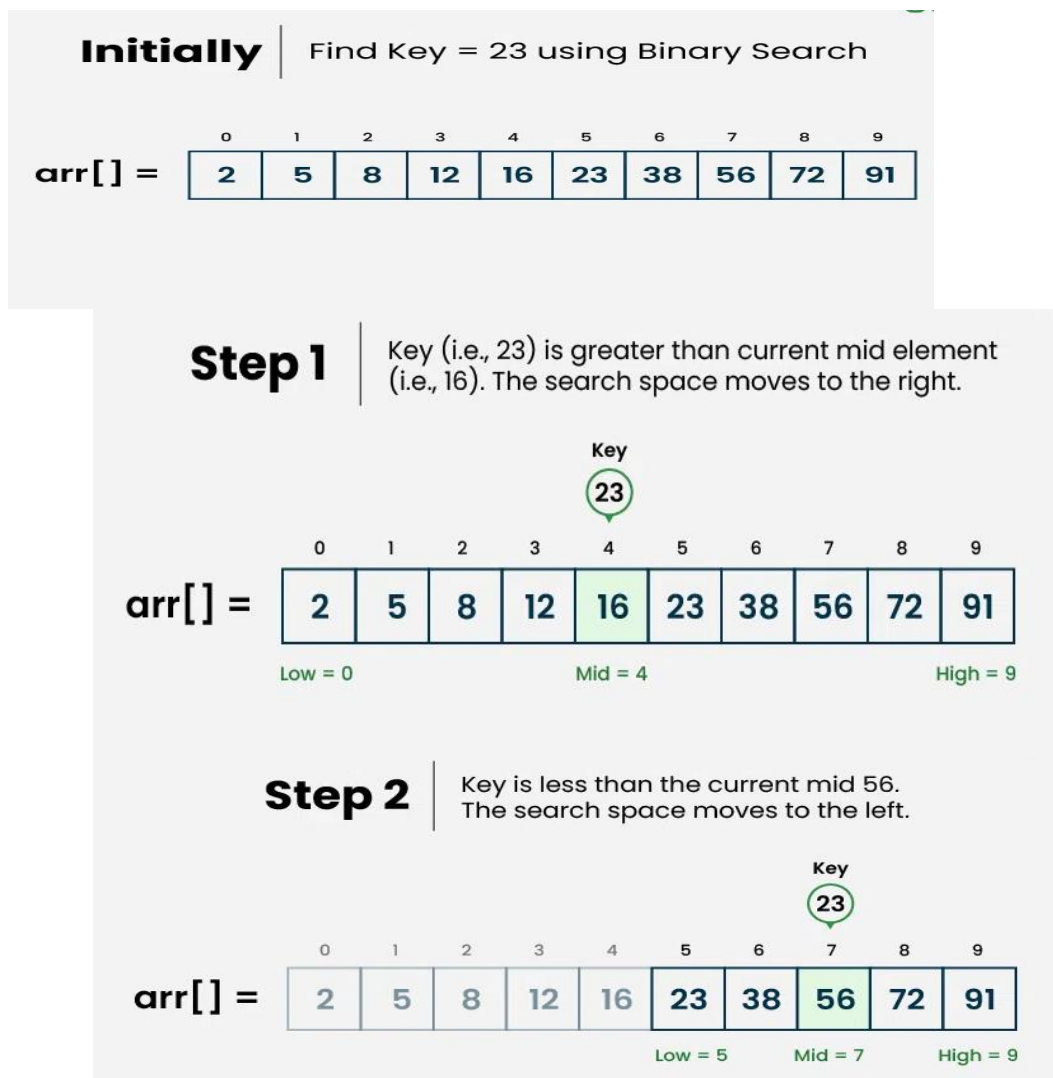
Below is the step-by-step algorithm for Binary Search:

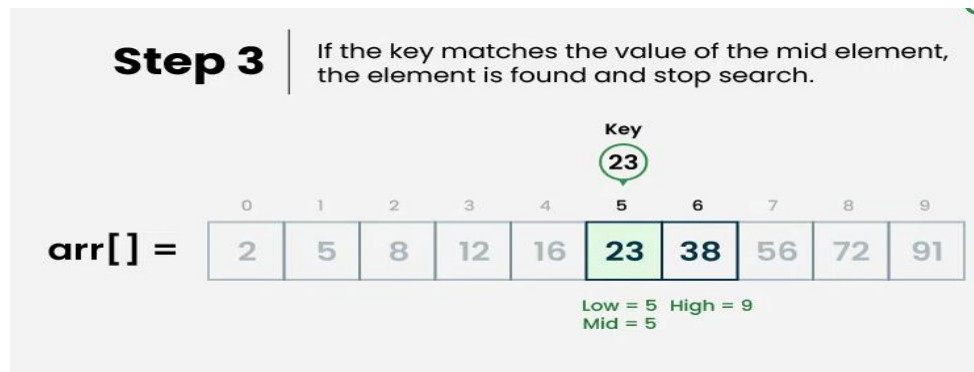
- Divide the search space into two halves by **finding the middle index “mid”**.

- 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.

How does Binary Search Algorithm work?

To understand the working of binary search, consider the following illustration: Consider an array **arr[]** = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the **target** = 23.



**Algorithm:**

1. Start the program
2. Declare the binarySearch() function.
3. Read the value
4. If($r \geq 1$) then find the mid value
5. Mid value is x means return otherwise search the value
6. Display the result
7. Stop the program

```
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }
    }
    return binarySearch(arr, mid + 1, r, x);
}

return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
}
```

```
int size = sizeof(arr[0]);
int x = 10;
int index = binarySearch(arr, 0, size - 1, x);

if (index == -1)
{
    printf("Element is not present in array");
}
else
{
    printf("Element is present at index %d", index);
}

return 0;
}
```

Output:

Element is present at index 3

4. Write a C program to sort a list of N elements using Selection Sort Algorithm.**Aim:**

To write a C program to sort a list of N elements using Selection Sort Algorithm

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

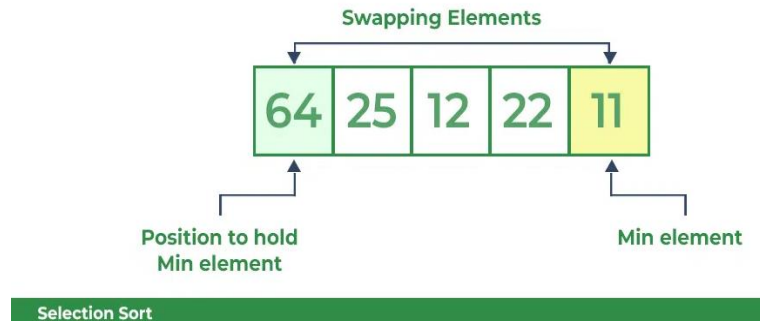
The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

How does Selection Sort Algorithm work?

Lets consider the following array as an example: **arr[] = {64, 25, 12, 22, 11}**

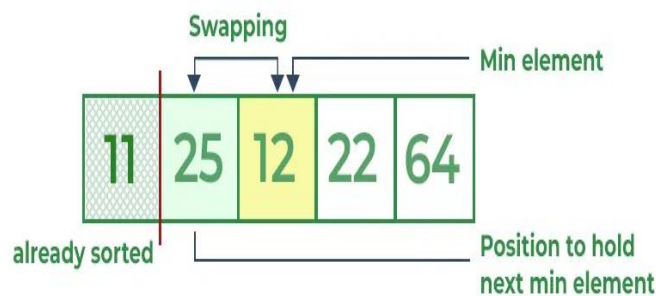
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.
- Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



Third Pass:

- Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.
- While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.



Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As **25** is the 4th lowest value hence, it will place at the fourth position.



Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.



Selection Sort

Algorithm:

1. Start the program
2. Declare the swap function.
3. Read the value and reverse the value using swap function
4. Initialize minimum value **min_idx** to location 0.
5. Traverse the array to find the minimum element in the array.
6. While traversing if any element smaller than **min_idx** is found then swap both values.
7. Then, increment **min_idx** to point to the next element.
8. Repeat until the array is sorted.
9. Display the result
10. Stop the program

```
#include <stdio.h>
```

```
void swap(int* xp, int* yp)
```

```
{  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}
```

```
void selectionSort(int arr[], int n)
```

```
{  
    int i, j, min_idx;  
  
    for (i = 0; i < n - 1; i++)  
    {  
  
        min_idx = i;  
        for (j = i + 1; j < n; j++)  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
  
        swap(&arr[min_idx], &arr[i]);  
    }  
}
```

```
void printArray(int arr[], int size)
```

```
{  
    int i;  
    for (i = 0; i < size; i++)
```



```
        printf("%d ", arr[i]);  
    printf("\n");  
}  
  
int main()  
{  
    int arr[] = { 64, 25, 12, 22, 11 };  
    int n = sizeof(arr) / sizeof(arr[0]);  
    selectionSort(arr, n);  
    printf("Sorted array: \n");  
    printArray(arr, n);  
    return 0;  
}
```

Output :

Sorted array:

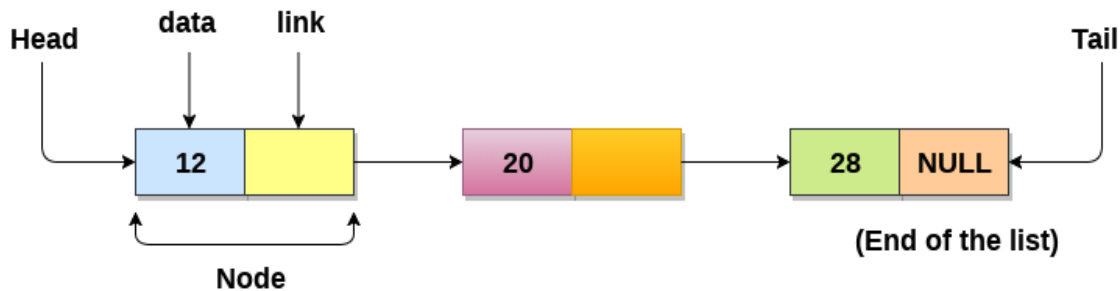
11 12 22 25 64

- 5. Write a C program to construct a singly linked list and perform insertion at the beginning, deletion at the specified position and display the elements.**

Aim:

To write a C program to construct a singly linked list and perform insertion at the beginning, deletion at the specified position and display the elements.

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

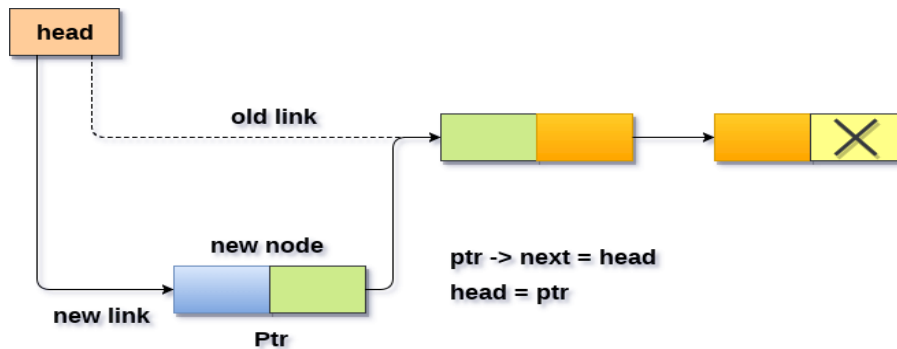
Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

Insertion in singly linked list at beginning

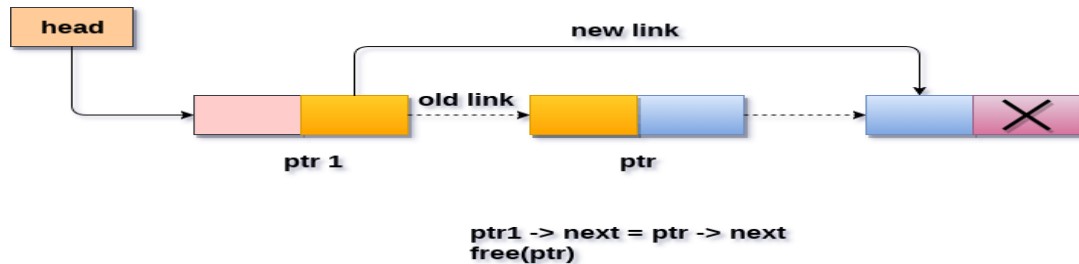
Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to insert a new node in the list at beginning.

- Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.
 1. `ptr = (struct node *) malloc(sizeof(struct node *));`
 2. `ptr → data = item`
 - Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.
 1. `ptr->next = head;`
 - At the last, we need to make the new node as the first node of the list this will be done by using the following statement.
 1. `head = ptr;`



Deletion at the specified position

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.



Deletion a node from specified position

Algorithm:

1. Start the program
2. Define a node structure
3. Function to insert a node at a the beginning of the linked list
4. Allocate memory for new node
5. Assign data to the new node
6. Make next of new node as head
7. Move the head to point to the new node
8. Function to delete a node at a specified position
9. If linked list is empty
10. Store head node. If head needs to be removed, change the head and remove the old head
11. Find previous node of the node to be deleted
12. If position is more than number of nodes
13. Node temp->next is the node to be deleted
14. Store pointer to the next of node to be deleted

15. Unlink the node from the linked list.
16. Unlink the deleted node from list
17. Function to display elements of the linked list
18. Display the result
19. Stop the program

```
#include <stdio.h>
#include <stdlib.h>

// Define a node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a node at the beginning of the linked list
void insertAtBeginning(struct Node** head_ref, int new_data) {
    // Allocate memory for new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    // Assign data to the new node
    new_node->data = new_data;

    // Make next of new node as head
    new_node->next = *head_ref;

    // Move the head to point to the new node
    *head_ref = new_node;
}

// Function to delete a node at a specified position
void deleteNodeAtPosition(struct Node** head_ref, int position) {
    // If linked list is empty
    if (*head_ref == NULL)
        return;

    // Store head node
    struct Node* temp = *head_ref;
```

```
// If head needs to be removed
if (position == 0) {
    *head_ref = temp->next; // Change head
    free(temp); // Free old head
    return;
}

// Find previous node of the node to be deleted
for (int i = 0; temp != NULL && i < position - 1; i++)
    temp = temp->next;

// If position is more than number of nodes
if (temp == NULL || temp->next == NULL)
    return;

// Node temp->next is the node to be deleted
// Store pointer to the next of node to be deleted
struct Node* next = temp->next->next;

// Unlink the node from the linked list
free(temp->next); // Free memory

// Unlink the deleted node from list
temp->next = next;
}

// Function to display elements of the linked list
void displayList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning
```

```
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 1);
insertAtBeginning(&head, 7);
insertAtBeginning(&head, 2);

printf("Linked list after insertion:\n");
displayList(head);

// Delete node at position 2
deleteNodeAtPosition(&head, 2);

printf("Linked list after deletion at position 2:\n");
displayList(head);

return 0;
}
```

Output:

Linked list after insertion:
2 -> 7 -> 1 -> 3 -> NULL
Linked list after deletion at position 2:
2 -> 7 -> 3 -> NULL

6. Write a C program to demonstrate the working of stack using arrays**Aim:**

To Write a C program to demonstrate the working of stack using arrays

Stack is a **linear data structure** which follows **LIFO** principle. In this article, we will learn how to implement Stack using Arrays. In Array-based approach, all stack-related operations are executed using arrays. Let's see how we can implement each operation on the stack utilizing the Array Data Structure.

Step-by-step approach:

1. **Initialize an array** to represent the stack.
2. Use the **end of the array** to represent the **top of the stack**.

3. Implement **push** (add to end), **pop** (remove from the end), and **peek** (check end) operations, ensuring to handle empty and full stack conditions.

Implement Stack Operations using Array:

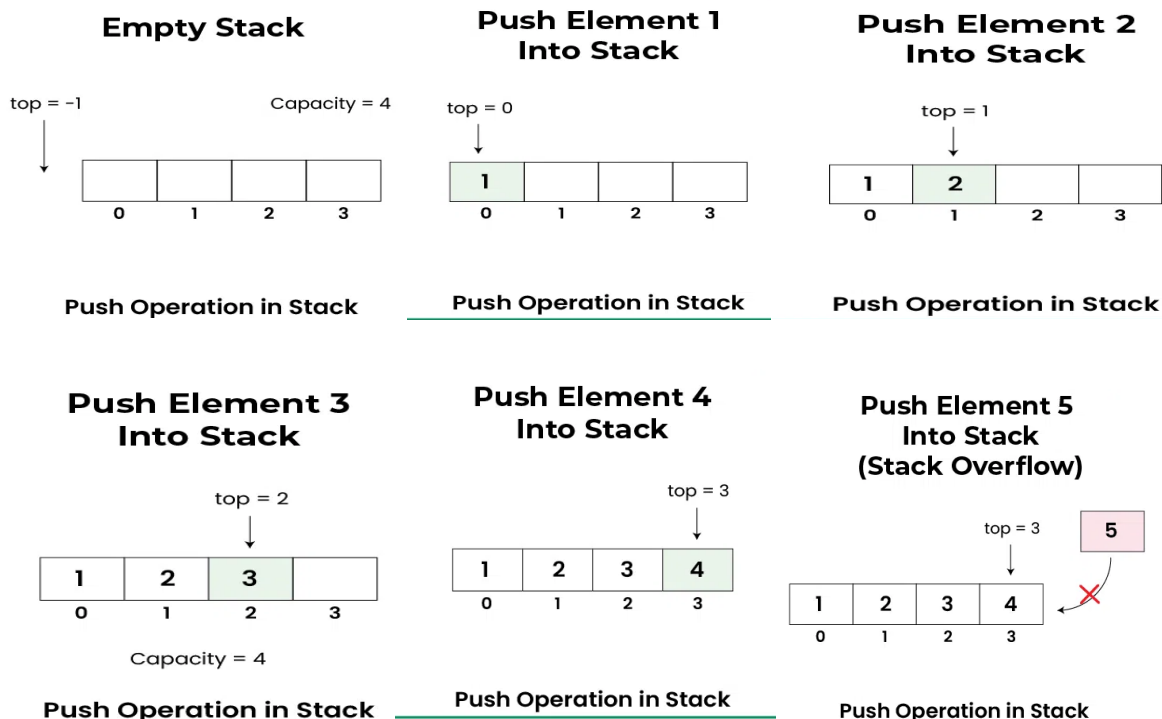
Here are the following operations of implement stack using array:

Push Operation in Stack:

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 ($\text{top} == \text{capacity} - 1$), then **Stack Overflows** and we cannot insert the element to the stack.
- Otherwise, we increment the value of top by 1 ($\text{top} = \text{top} + 1$) 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.

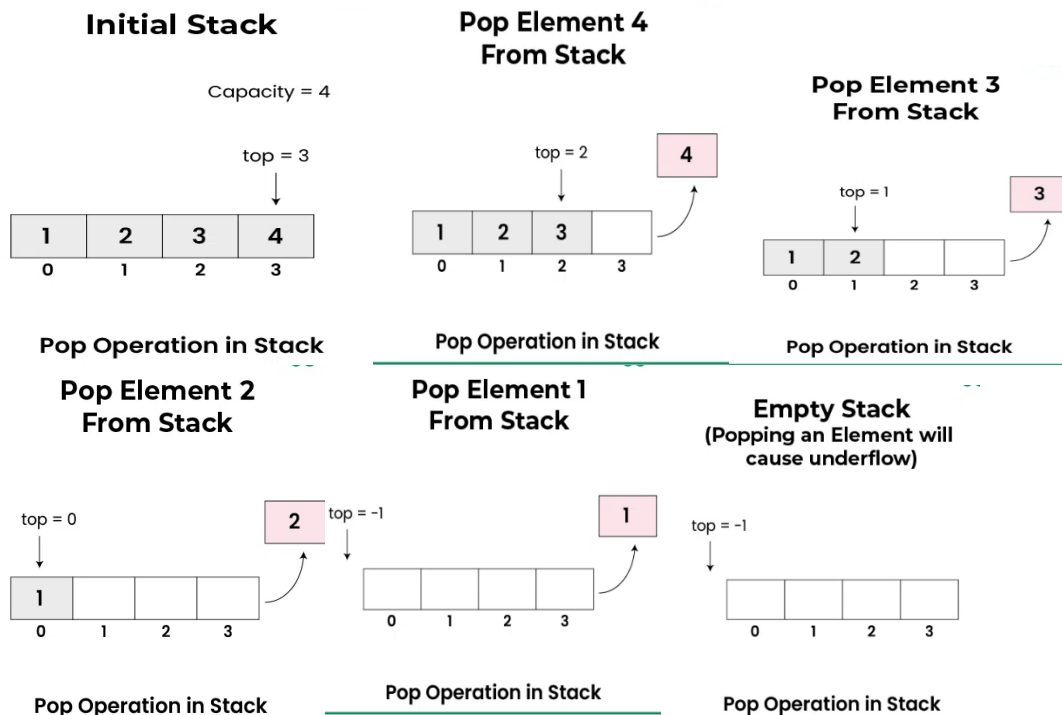


Pop Operation in Stack:

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 ($\text{top} == -1$), 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 ($\text{top} = \text{top} - 1$) and return the stored top value.



Top or Peek Operation in Stack:

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 ($\text{top} == -1$), we simply print "Stack is empty".
- Otherwise, we return the element stored at **index = top**.

isEmpty Operation in Stack:

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation :

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

isFull Operation in Stack :

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

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

Algorithm:

- 1 Start
- 2 Initialize $\text{top} = -1$;

- 3 Push operation increases top by one and writes pushed element to storage[top];
- 4 Pop operation checks that top is not equal to -1 and decreases top variable by 1;
- 5 Display operation checks that top is not equal to -1 and returns storage[top];
- 6 Stop

Program:

```
#include<stdio.h>

int stack[100],choice,n,top,x,i;

void push(void);
void pop(void);
void display(void);

int main()
{
    top=-1;

    printf("\n Enter the size of STACK[MAX=100]:");

    scanf("%d",&n);

    printf("\n\t STACK OPERATIONS USING ARRAY");

    printf("\n\t-----");

    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

    do

    {

        printf("\n Enter the Choice:");

        scanf("%d",&choice);
```

```
switch(choice)
{
    case 1:
    {
        push();
        break;
    }
    case 2:
    {
        pop();
        break;
    }
    case 3:
    {
        display();
        break;
    }
    case 4:
    {
        printf("\n\t EXIT POINT ");
        break;
    }
    default:
    {
        printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
    }
}
```

```
    }

    }
}
while(choice!=4);
return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");

    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
```

```
{
    printf("\n\t Stack is under flow");
}
else
{
    printf("\n\t The popped elements is %d",stack[top]);
    top--;
}
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
}
```

OUTPUT:

Enter the size of STACK[MAX=100]:10

STACK OPERATIONS USING ARRAY

1.PUSH

2.POP

3.DISPLAY

4.EXIT

Enter the Choice:1

Enter a value to be pushed:12

Enter the Choice:1

Enter a value to be pushed:24

Enter the Choice:1

Enter a value to be pushed:98

Enter the Choice:3

The elements in STACK

98

24

12

Press Next Choice

Enter the Choice:2

The popped elements is 98

Enter the Choice:3

The elements in STACK

24

12

Press Next Choice

Enter the Choice:4

EXIT POINT

7. Write a C program for Towers of Hanoi problem using Recursion.

Aim:

To write a C program for Towers of Hanoi problem using Recursion

Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- Shift 'N-1' disks from 'A' to 'B', using C.
- Shift last disk from 'A' to 'C'.
- Shift 'N-1' disks from 'B' to 'C', using A.

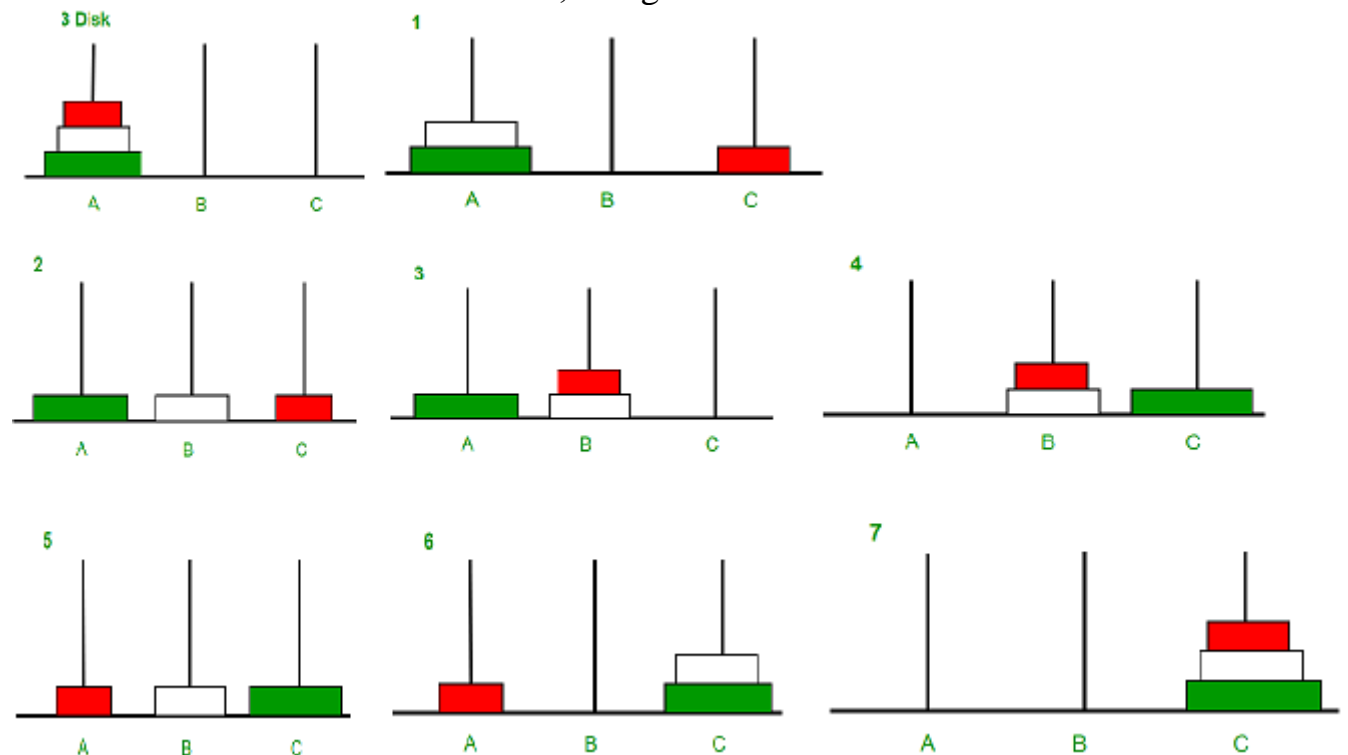


Image illustration for 3 disks

Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod**.
- Make a function call for $N - 1$ th disk.
- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for $N - 1$ th disk.

Algorithm:

- 1 Start the program
- 2 Tower of Hanoi is mathematical puzzle where we have three rods(A,B,C) and N disks
- 3 Initially all the disks are stacked in decreasing value of diameter that is the smallest disk is placed on the top and they are on rod A.
- 4 The objective of the puzzle is to move the entire stack to another rod (here considered C)
- 5 Rules: Only one disk can be moved at a time
- 6 Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack that is a disk can only be moved if it is the uppermost disk on a stack
- 7 No disk may be placed on top of a smaller disk
- 8 Stop

Program:

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
}
```



```
towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
```

```
towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
return 0;
}
```

Output:

Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C

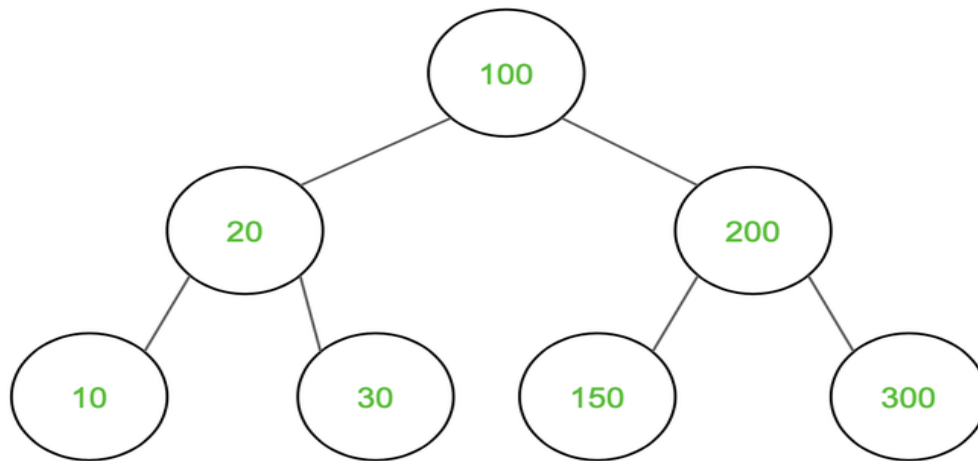
8. Write a C program to create and traverse a binary search tree.

Aim:

To write a C program to create and traverse a binary search tree

a Binary Search Tree, The task is to print the elements in inorder, preorder, and postorder traversal of the Binary Search Tree.

Input:



Binary Search Tree

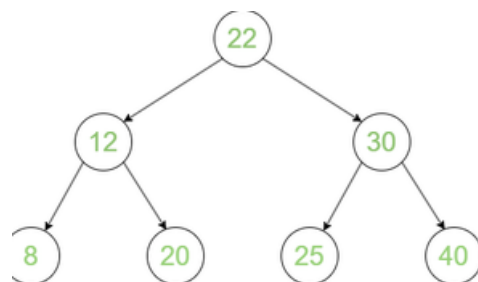
Output:

Inorder Traversal: 10 20 30 100 150 200 300

Preorder Traversal: 100 20 10 30 200 150 300

Postorder Traversal: 10 30 20 150 300 200 100

Input:



Output:

Inorder Traversal: 8 12 20 22 25 30 40

Preorder Traversal: 22 12 8 20 30 25 40

Postorder Traversal: 8 20 12 25 40 30 22

Inorder Traversal:

Below is the idea to solve the problem:

At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**.

Follow the below steps to implement the idea:

- Traverse left subtree
- Visit the root and print the data.
- Traverse the right subtree

The **inorder traversal** of the BST gives the values of the nodes in sorted order.

To get the decreasing order visit the right, root, and left subtree.

Preorder Traversal:

Below is the idea to solve the problem:

At first visit the **root** then traverse **left subtree** and then traverse the **right subtree**.

Follow the below steps to implement the idea:

- Visit the root and print the data.
- Traverse left subtree
- Traverse the right subtree

Postorder Traversal:

Below is the idea to solve the problem:

At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**.

Follow the below steps to implement the idea:

- Traverse left subtree
- Traverse the right subtree
- Visit the root and print the data.

Algorithm:

- 1 Start
- 2 Search operation algorithm:
 - 1 If the root is NULL or the key of the root matches the target then return the current root node
 - 2 If the target is greater than the key of the current root then recursively call searchNode with the right subtree of the current root and the target. Return the result of the recursive call.
 - 3 If the target is smaller than the key of the current root then recursively call searchNode with the left subtree of the current root and target. Return the result of the recursive call.
 - 4 If the target is not found in the current subtree, return NULL.
- 3 Insertion operation algorithm:
 - 1 If the current node is NULL return a new node created with the specified value
 - 2 If the value is less than the key of the current node then recursively call insertNode with the left subtree of the current node and the value. Update the left child of the current node with the result of the recursive call.
 - 3 If the value is greater than the key of the current node then recursively call insertNode with the right subtree of the current node and value. Update the right child of the current node with the result of the recursive call.

- 4 Return the current node(the root of the modified tree)
- 4 Delete operation algorithm:
 - 1 If the root is NULL then Return NULL(base case for recursion)
 - 2 If the value to be deleted (x) is greater than the key of the current root then recursively call delete with the right subtree of the current root and the value x. otherwise update the right child of the current root with the result of the recursive call.
 - 3 If the value to be deleted (x) is smaller than the key of the current root then recursively call delete with the left subtree of the current root and the value x. otherwise update the left child of the current root with the result of the recursive call.
 - 4 If the value to be deleted (x) is equal to the key of the current root:
 - 1.If the current node has no child then free the current node otherwise return NULL.
 - 2.If the current node has one child then set temp to the non null child of the current node. Free the current node then return temp.
 3. If the current node has two children then find the minimum node in the right subtree(temp).Replace the key of the current node with the key of temp. Recursively call delete with the right subtree of the current node and the key of temp.
 - 5 Return the current node(the root of the modified tree).
- 5.Display the value
- 6.Stop

Program:

```
#include <stdio.h>

#include <stdlib.h>

// Define a structure for a binary tree node
struct BinaryTreeNode {
    int key;
```

```
struct BinaryTreeNode *left, *right;
};

// Function to create a new node with a given value
struct BinaryTreeNode* newNodeCreate(int value)
{
    struct BinaryTreeNode* temp
        = (struct BinaryTreeNode*)malloc(
            sizeof(struct BinaryTreeNode));
    temp->key = value;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to insert a node with a specific value in the
// tree
struct BinaryTreeNode*
insertNode(struct BinaryTreeNode* node, int value)
{
    if (node == NULL) {
        return newNodeCreate(value);
    }
    if (value < node->key) {
        node->left = insertNode(node->left, value);
    }
    else if (value > node->key) {
        node->right = insertNode(node->right, value);
    }
}
```

```
    return node;
}

// Function to perform post-order traversal
void postOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf(" %d ", root->key);
    }
}

// Function to perform in-order traversal
void inOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
        inOrder(root->left);
        printf(" %d ", root->key);
        inOrder(root->right);
    }
}

// Function to perform pre-order traversal
void preOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
```

```
    printf(" %d ", root->key);
    preOrder(root->left);
    preOrder(root->right);
}
}

// Function to delete a node from the tree
struct BinaryTreeNode* delete (struct BinaryTreeNode* root,
                               int x)
{
    if (root == NULL)
        return NULL;

    if (x > root->key) {
        root->right = delete (root->right, x);
    }
    else if (x < root->key) {
        root->left = delete (root->left, x);
    }
    else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        else if (root->left == NULL
                 || root->right == NULL) {
            struct BinaryTreeNode* temp;
            if (root->left == NULL) {
```



```
        temp = root->right;
    }

    else {
        temp = root->left;
    }

    free(root);
    return temp;
}

else {
    struct BinaryTreeNode* temp
        = findMin(root->right);
    root->key = temp->key;
    root->right = delete (root->right, temp->key);
}

}

return root;
}
```

```
int main()
{
    // Initialize the root node
    struct BinaryTreeNode* root = NULL;

    // Insert nodes into the binary search tree
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
}
```

```
insertNode(root, 40);
insertNode(root, 70);
insertNode(root, 60);
insertNode(root, 80);

// Search for a node with key 60
if (searchNode(root, 60) != NULL) {
    printf("60 found");
}
else {
    printf("60 not found");
}

printf("\n");

// Perform post-order traversal
postOrder(root);
printf("\n");
// Perform pre-order traversal
preOrder(root);
printf("\n");

// Perform in-order traversal
inOrder(root);
printf("\n");

// Perform delete the node (70)
```

```
struct BinaryTreeNode* temp = delete (root, 70);  
printf("After Delete: \n");  
inOrder(root);  
  
// Free allocated memory (not done in this code, but  
// good practice in real applications)  
  
return 0;  
}
```

OUTPUT:

```
60 found  
20 40 30 60 80 70 50  
50 30 20 40 70 60 80  
20 30 40 50 60 70 80  
After Delete:  
20 30 40 50 60 80
```

PART-B

1. Write a C program to demonstrate the following operations in an array

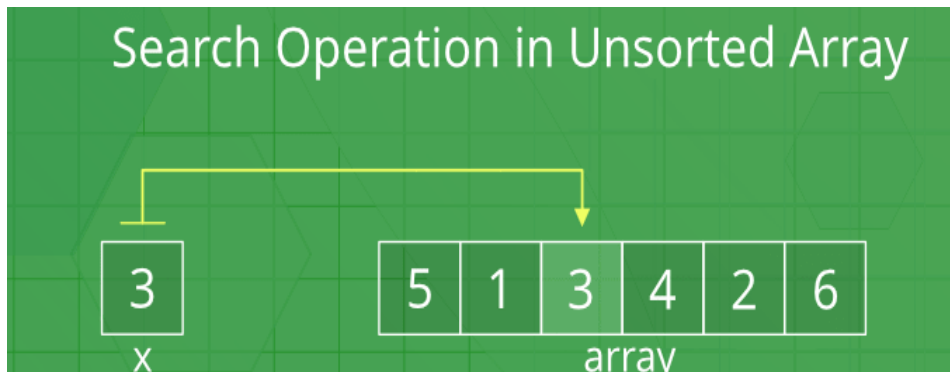
- (i) insertion of an element (ii) deletion of an element (iii) traversal**
- (i) Searching of an element:**

Aim:

To write a C program to demonstrate the searching operation in an array

Search Operation:

In an unsorted array, the search operation can be performed by linear traversal from the first element to the last element.

**Algorithm:**

- 1 Start
- 2 Declare the variable
- 3 Input the array elements, the elements to be searched
- 4 Traverse the array and check if the element is present in the array
- 5 Display “Yes” if it is present in the array, else display “No”
- 6 Stop

Program:

```
// C program to implement linear
// search in unsorted array
#include <stdio.h>

// Function to implement search operation
int findElement(int arr[], int n, int key)
{
```

```
int i;
for (i = 0; i < n; i++)
    if (arr[i] == key)
        return i;

// If the key is not found
return -1;
}

// Driver's Code
int main()
{
    int arr[] = { 12, 34, 10, 6, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Using a last element as search element
    int key = 40;

    // Function call
    int position = findElement(arr, n, key);

    if (position == -1)
        printf("Element not found");
    else
        printf("Element Found at Position: %d",
            position + 1);

    return 0;
```

```
}
```

OUTPUT:

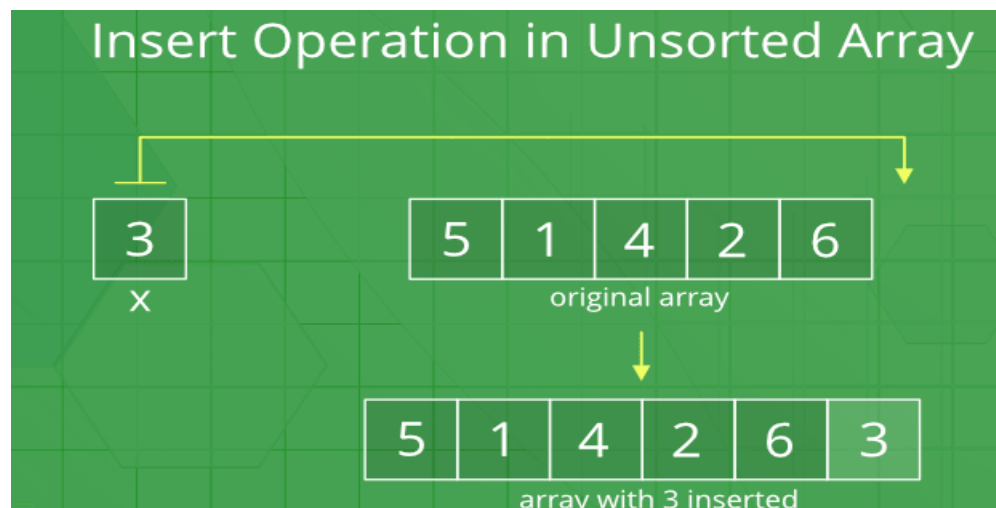
Element Found at Position: 5

(ii) Insertion of an element:**Aim:**

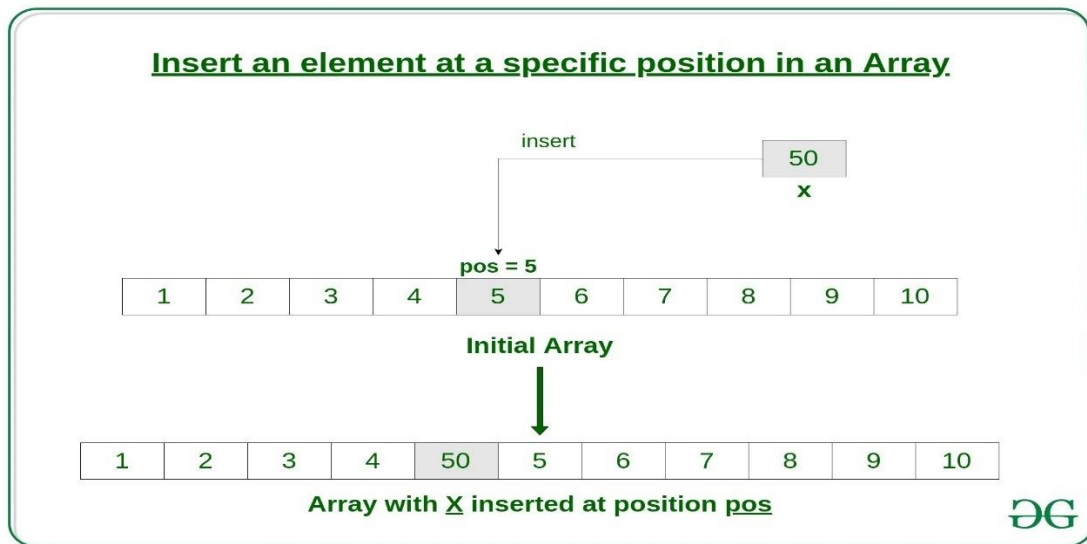
To write a C program to demonstrate the insertion operation in an array

1. Insert at the end:

In an unsorted array, the insert operation is faster as compared to a sorted array because we don't have to care about the position at which the element is to be placed.

**2. Insert at any position**

Insert operation in an array at any position can be performed by shifting elements to the right, which are on the right side of the required position

**Algorithm:**

- 7 Start
- 8 Declare the variable
- 9 Input the array elements, the position of the new element to be inserted and the new element
- 10 Insert the new element at that position and shift the rest of the elements to right by one position
- 11 Display the output
- 12 Stop

// C program to implement insert

// operation in an unsorted array.

```
#include <stdio.h>
```

// Inserts a key in arr[] of given capacity.

// n is current size of arr[]. This

// function returns n + 1 if insertion

// is successful, else n.

```
int insertSorted(int arr[], int n, int key, int capacity)
```

```
{
```

```
// Cannot insert more elements if n is
// already more than or equal to capacity
if (n >= capacity)
    return n;

arr[n] = key;

return (n + 1);
}

// Driver Code
int main()
{
    int arr[20] = { 12, 16, 20, 40, 50, 70 };
    int capacity = sizeof(arr) / sizeof(arr[0]);
    int n = 6;
    int i, key = 26;

    printf("\n Before Insertion: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    // Inserting key
    n = insertSorted(arr, n, key, capacity);

    printf("\n After Insertion: ");
```



```
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}

int i, key = 26;

printf("\n Before Insertion: ");
for (i = 0; i < n; i++)
    printf("%d ", arr[i]);

// Inserting key
n = insertSorted(arr, n, key, capacity);

printf("\n After Insertion: ");
for (i = 0; i < n; i++)
    printf("%d ", arr[i]);

return 0;
}
```

Output

```
Before Insertion: 12 16 20 40 50 70
After Insertion: 12 16 20 40 50 70 26
```

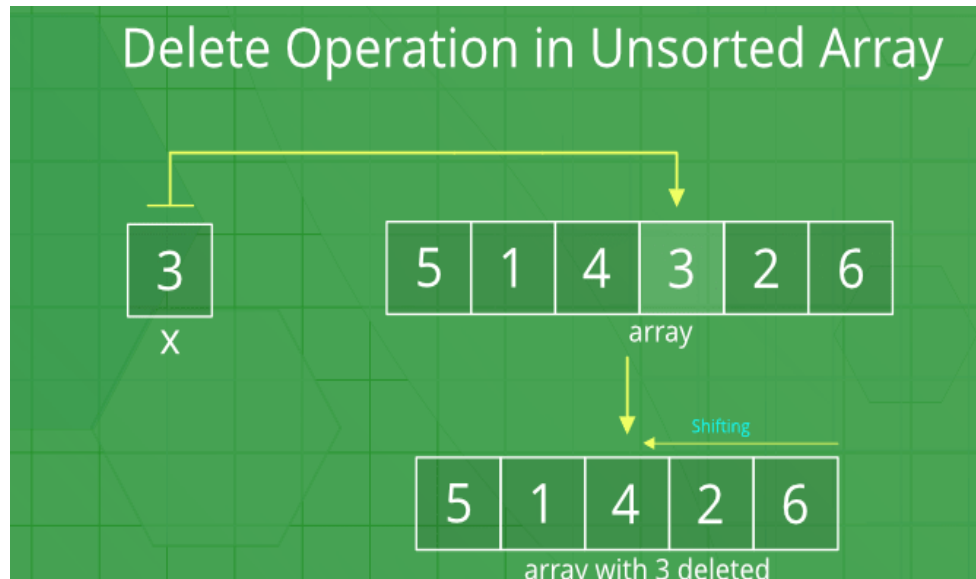
(iii) Deletion of an element:

Aim:

To write a C program to demonstrate the deletion operation in an array

Delete Operation:

In the delete operation, the element to be deleted is searched using the linear search, and then the delete operation is performed followed by shifting the elements.



Algorithm:

1. Start
2. Declare the variable
3. Input the array elements, the position of the new element to be inserted and the new element
4. Delete the element and shift the rest of the elements to left by one position
5. Display the output
6. Stop

// C program to implement delete operation in a

// unsorted array

```
#include <stdio.h>
```

```
// To search a key to be deleted
```

```
int findElement(int arr[], int n, int key);
```

```
// Function to delete an element
```

```
int deleteElement(int arr[], int n, int key)
```

```
{
    // Find position of element to be deleted
    int pos = findElement(arr, n, key);

    if (pos == -1) {
        printf("Element not found");
        return n;
    }

    // Deleting element
    int i;
    for (i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];

    return n - 1;
}

// Function to implement search operation
int findElement(int arr[], int n, int key)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == key)
            return i;

    return -1;
}
```

```
// Driver's code

int main()
{
    int i;
    int arr[] = { 10, 50, 30, 40, 20 };

    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    printf("Array before deletion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    // Function call

    n = deleteElement(arr, n, key);

    printf("\nArray after deletion\n");

    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output

Array before deletion

10 50 30 40 20

Array after deletion

10 50 40 20

2. Write a C program to sort a list of elements using Bubble sort algorithm**Aim:**

To write a C program to sort a list of elements using 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 is quite high.

Bubble Sort Algorithm

In Bubble S

ort algorithm,

- traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

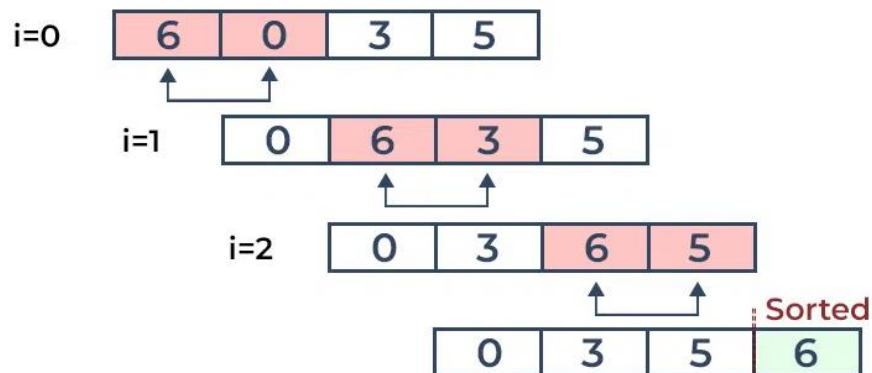
How does Bubble Sort Work?

Let us understand the working of bubble sort with the help of the following illustration:

Input: arr[] = {6, 0, 3, 5}

First Pass:

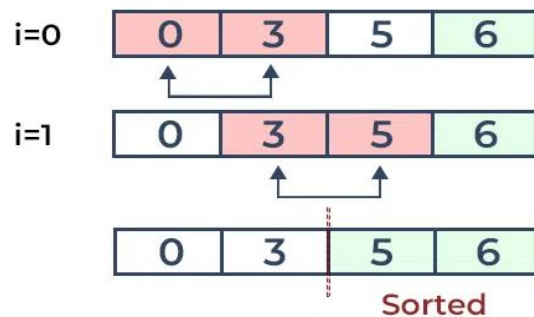
The largest element is placed in its correct position, i.e., the end of the array.

STEP
01Placing the 1st largest element at Correct position

Bubble Sort Algorithm : Placing the largest element at correct position

Second Pass:

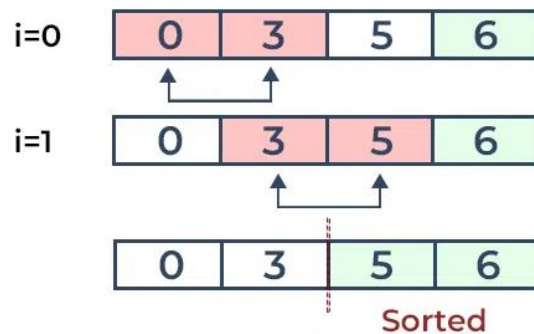
Place the second largest element at correct position

STEP
02Placing 2nd largest element at Correct position

Bubble Sort Algorithm : Placing the second largest element at correct position

Third Pass:

Place the remaining two elements at their correct positions.

STEP
02Placing 2nd largest element at Correct position

Bubble Sort Algorithm : Placing the remaining elements at their correct positions

Algorithm:

1. Start
2. Repeat steps 3 and 4 for $i=1$ to n
3. Set $j=1$
4. Repeat while $j \leq n$
5. If $a[j] < a[j+1]$
Then interchange $a[j]$ and $a[j+1]$
End of if
6. stop

```
// C program for implementation of Bubble sort
#include <stdio.h>
```

```
// Swap function
void swap(int* arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
```

```
int i, j;
for (i = 0; i < n - 1; i++)

    // Last i elements are already
    // in place
    for (j = 0; j < n - i - 1; j++)
        if (arr[j] > arr[j + 1])
            swap(arr, j, j + 1);
}

// Function to print an array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver code
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int N = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, N);
    printf("Sorted array: ");
    printArray(arr, N);
    return 0;
}
```

Output

```
Sorted array:
1 2 4 5 8
```

3. Write a C program to sort a list of N elements using Insertion Sort Algorithm.

Aim:

To write a C program to sort a list of N elements using Insertion Sort Algorithm

Insertion sort is an algorithm used to sort a collection of elements in ascending or descending order. The basic idea behind the algorithm is to divide the list into two parts: a sorted part and an unsorted part.

Initially, the sorted part contains only the first element of the list, while the rest of the list is in the unsorted part. The algorithm then iterates through each element in the unsorted part, picking one at a time, and inserts it into its correct position in the sorted part.

To do this, the algorithm compares the current element with each element in the sorted part, starting from the rightmost element. It continues to move to the left until it finds an element that is smaller (if sorting in ascending order) or larger (if sorting in descending order) than the current element.

Once the correct position has been found, the algorithm shifts all the elements to the right of that position to make room for the current element, and then inserts the current element into its correct position.

This process continues until all the elements in the unsorted part have been inserted into their correct positions in the sorted part, resulting in a fully sorted list.

One of the advantages of insertion sort is that it is an in-place sorting algorithm, which means that it does not require any additional storage space other than the original list. Additionally, it has a time complexity of $O(n^2)$, which makes it suitable for small datasets, but not for large ones.

Overall, insertion sort is a simple, yet effective sorting algorithm that can be used for small datasets or as a part of more complex algorithms.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12	11	13	5	6
----	----	----	---	---

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6
----	----	----	---	---

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6
-----------	-----------	----	---	---

Second Pass:

- Now, move to the next two elements and compare them

11	12	13	5	6
----	-----------	-----------	---	---

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11** and **12**
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6
----	----	-----------	----------	---

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6
----	----	----------	-----------	---

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6
----	----------	-----------	----	---

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6
----------	-----------	----	----	---

- here, it is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5**, **11** and **12**
- Moving to the next two elements 13 and 6

5	11	12	13	6
---	----	----	-----------	----------

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13
---	----	----	----------	-----------

- Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13
---	----	---	----	----

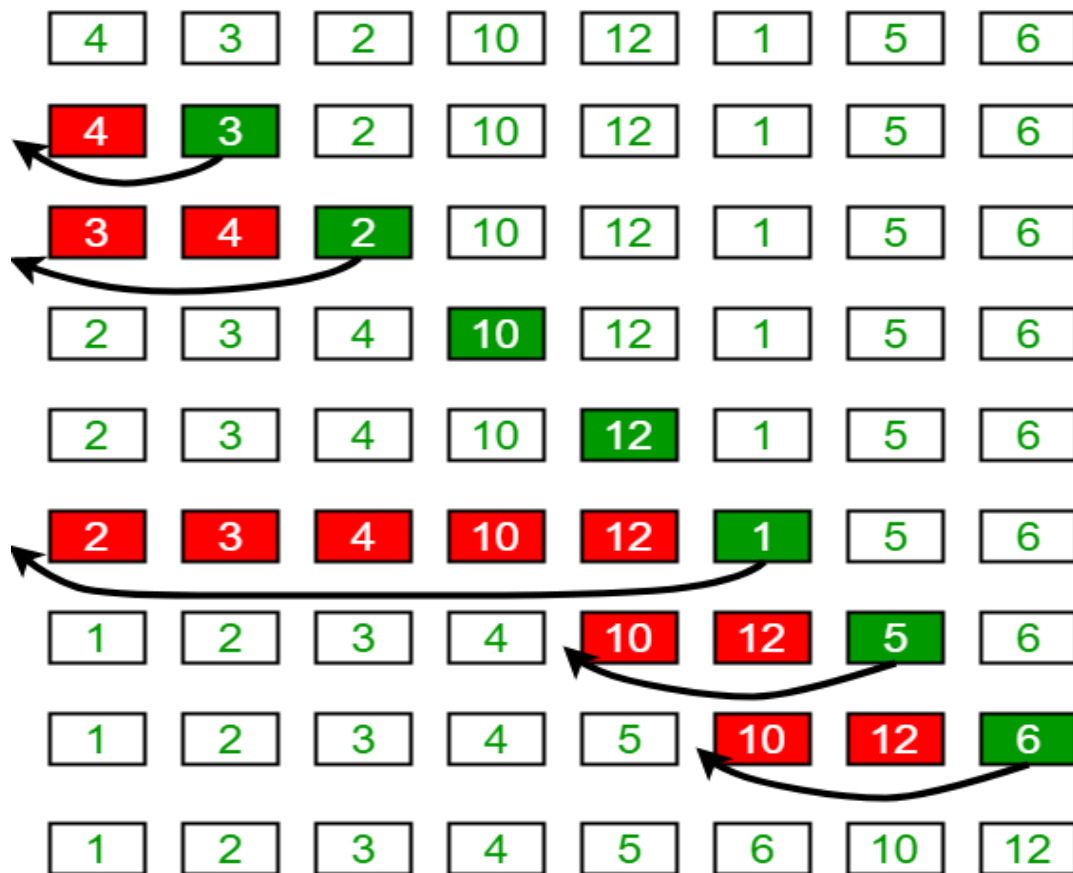
- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13
---	---	----	----	----

- Finally, the array is completely sorted.

Illustrations:

Insertion Sort Execution Example



Algorithm:

1. Start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

4. Stop

Program:

```
# include <stdio.h>
```

```
// Define the maximum size of the array
```

```
#define max 20
```

```
// Main function
```

```
int main() {  
    // Declare variables  
    int arr[max], i, j, temp, len;  
  
    // Input the number of elements  
    printf("Input numbers you want to input: ");  
    scanf("%d", &len);  
  
    // Input array values  
    printf("Input %d values to sort\n", len);  
    for (i = 0; i < len; i++)  
        scanf("%d", &arr[i]);  
  
    // Insertion sort algorithm  
    for (i = 1; i < len; i++) {  
        for (j = i; j > 0; j--) {  
            // Swap if the current element is smaller than the previous  
            one  
            if (arr[j] < arr[j - 1]) {  
                temp = arr[j];  
                arr[j] = arr[j - 1];  
                arr[j - 1] = temp;  
            }  
        }  
    }  
  
    // Display the sorted array in ascending order
```

```
printf("\nThe ascending order of the values:\n");  
for (i = 0; i < len; i++)  
    printf("%d\n", arr[i]);  
  
return 0;  
}
```

Sample Output:

Input numbers you want to input: Input 5 values to sort

The ascending order of the values:

11
13
15
20
25

4. Write a C program to simulate the working of Linear Queue using a linked list

Aim:

To write a C program to simulate the working of Linear Queue using a linked list

the Linked List implementation of the queue data structure is discussed and implemented. Print '-1' if the queue is empty.

Approach: To solve the problem follow the below idea:

we maintain two pointers, **front**, and **rear**. The front points to the first item of the queue and rear points to the last item.

- **enQueue():** This operation adds a new node after the rear and moves the rear to the next node.
- **deQueue():** This operation removes the front node and moves the front to the next node.

Follow the below steps to solve the problem:

- Create a class QNode with data members integer data and QNode* next

- A parameterized constructor that takes an integer x value as a parameter and sets data equal to x and next as NULL
- Create a class Queue with data members QNode front and rear
- Enqueue Operation with parameter x:
 - Initialize QNode* temp with data = x
 - If the rear is set to NULL then set the front and rear to temp and return(Base Case)
 - Else set rear next to temp and then move rear to temp
- Dequeue Operation:
 - If the front is set to NULL return(Base Case)
 - Initialize QNode temp with front and set front to its next
 - If the front is equal to NULL then set the rear to NULL
 - Delete temp from the memory

Algorithm:

1. Start
2. Initialize front=0;rear=-1;
3. Enqueue operation inserts an element at the rear of the list
4. Dequeue operation deletes an element at the front of the list
5. Display operation displays all the element in the list
6. Stop

Program:

```
// implementation of queue

#include <stdio.h>

#include <stdlib.h>

// A linked list (LL) node to store a queue entry

struct QNode {
    int key;
    struct QNode* next;
};
```

// The queue, front stores the front node of LL and rear

// stores the last node of LL

```
struct Queue {  
    struct QNode *front, *rear;  
};
```

// A utility function to create a new linked list node.

```
struct QNode* newNode(int k)  
{  
    struct QNode* temp  
        = (struct QNode*)malloc(sizeof(struct QNode));  
    temp->key = k;  
    temp->next = NULL;  
    return temp;  
}
```

// A utility function to create an empty queue

```
struct Queue* createQueue()  
{  
    struct Queue* q  
        = (struct Queue*)malloc(sizeof(struct Queue));  
    q->front = q->rear = NULL;  
    return q;  
}
```

// The function to add a key k to q

```
void enQueue(struct Queue* q, int k)
```

```
{
    // Create a new LL node
    struct QNode* temp = newNode(k);

    // If queue is empty, then new node is front and rear
    // both
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove a key from given queue q
void deQueue(struct Queue* q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return;

    // Store previous front and move front one node ahead
    struct QNode* temp = q->front;

    q->front = q->front->next;
```



```
// If front becomes NULL, then change rear also as NULL
if (q->front == NULL)
    q->rear = NULL;

free(temp);
}

// Driver code
int main()
{
    struct Queue* q = createQueue();
    enqueue(q, 10);

    enqueue(q, 20);

    dequeue(q);

    dequeue(q);

    enqueue(q, 30);

    enqueue(q, 40);

    enqueue(q, 50);

    dequeue(q);

    printf("Queue Front : %d \n", ((q->front != NULL) ? (q->front)->key : -1));

    printf("Queue Rear : %d", ((q->rear != NULL) ? (q->rear)->key : -1));
```

```
return 0;  
  
}
```

Output

Queue Front : 40

Queue Rear : 50

5. Write a C program to search for a given element in a sorted linked list.

Aim:

To Write a C program to search for a given element in a sorted linked list.

Search an element in a Linked List (Recursive Approach):

Follow the below steps to solve the problem:

- If the head is NULL, return false.
- If the head's key is the same as **X**, return true;
- Else recursively search in the next node.

Algorithm:

1. Start with the first item in the list
2. Compare the current item to the target
3. If the current value matches the target then we declare victory and stop
4. If the current value is less than the target then set the current item to be the next item and repeat from 2
5. Stop

Program:

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
};
```

```
void display (struct Node *node)
```

```
{
```

```
    //as linked list will end when Node is Null
```

```
    while (node != NULL)
```

```
    {
```

```
        printf ("%d ", node->data);
```

```
        node = node->next;
```

```
    }
```

```
    printf ("\n");
```

```
}
```

```
int searchElement (struct Node *head, int item, int index)
```

```
{
```

```
    // Base case
```

```
    if (head == NULL)
```

```
    return -1;

// If data is present in current node, return true
if (head->data == item)
    return index;

// not present here will check for next position
// in next recursive iteration
index++;

// Recur for remaining list
return searchElement (head->next, item, index);
}

int main ()
{
    int item;

    //creating 4 pointers of type struct Node
    //So these can point to address of struct type variable
    struct Node *head = NULL;
    struct Node *node2 = NULL;
    struct Node *node3 = NULL;
    struct Node *node4 = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node *) malloc (sizeof (struct Node));
```

```
node2 = (struct Node *) malloc (sizeof (struct Node));
node3 = (struct Node *) malloc (sizeof (struct Node));
node4 = (struct Node *) malloc (sizeof (struct Node));

head->data = 10;           // data set for head node
head->next = node2;        // next pointer assigned to address of node2

node2->data = 15;
node2->next = node3;

node3->data = 20;
node3->next = node4;

node4->data = 25;
node4->next = NULL;

printf ("Linked List: ");
display (head);

printf ("Enter element to search: ");
scanf ("%d", &item);

int index = searchElement (head, item, 0);

if (index == -1)
    printf ("Item not found");
```

```
else  
    printf ("Item found at position: %d", index + 1);  
  
return 0;  
}
```

Output

Linked List: 10 15 20 25

Enter element to search: 10

Item found at position: 1

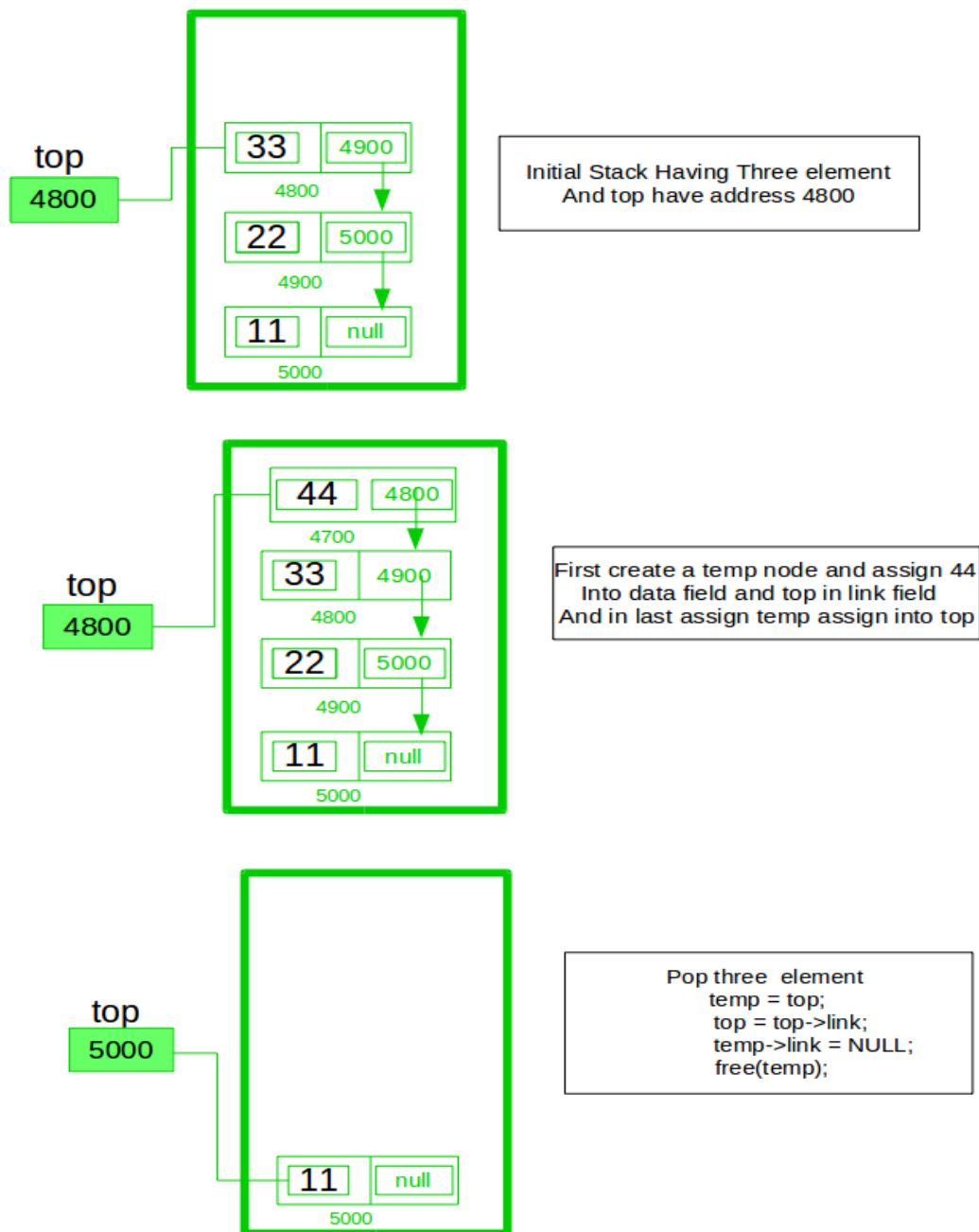
6. write a C program to implement stack operations using Linked list.

Aim:

To write a C program to implement stack operations using Linked list

To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

So we need to follow a simple rule in the implementation of a stack which is **last in first out** and all the operations can be performed with the help of a top variable. Let us learn how to perform **Pop, Push, Peek, and Display** operations in the following article:



In the stack Implementation, a stack contains a top pointer. which is the “head” of the stack where pushing and popping items happens at the head of the list. The first node has a null in the link field and second node-link has the first node address in the link field and so on and the last node address is in the “top” pointer.

The main advantage of using a linked list over arrays is that it is possible to implement a stack that can shrink or grow as much as needed. Using an array will put a restriction on the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated. so overflow is not possible.

Stack Operations:

- **push():** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
- **pop():** Return the top element of the Stack i.e simply delete the first element from the linked list.
- **peek():** Return the top element.
- **display():** Print all elements in Stack.

Push Operation:

- Initialise a node
- Update the value of that node by data i.e. **node->data = data**
- Now link this node to the top of the linked list
- And update top pointer to the current node

Pop Operation:

- First Check whether there is any node present in the linked list or not, if not then return
- Otherwise make pointer let say **temp** to the top node and move forward the top node by 1 step
- Now free this temp node

Peek Operation:

- Check if there is any node present or not, if not then return.
- Otherwise return the value of top node of the linked list

Display Operation:

- Take a **temp** node and initialize it with top pointer
- Now start traversing temp till it encounters NULL
- Simultaneously print the value of the temp node

Algorithm:

1. Start
2. Push operation inserts an element at the front
3. Pop operation deletes an element at th front of the list
4. Display operation displays all the elements in the list
5. Stop

Program:

```
#include<stdio.h>
#include<stdlib.h>
```

```
struct Node
```



```
{
    int data;
    struct Node *next;
};

struct Node *head = NULL;

void push(int val)
{
    //create new node
    struct Node *newNode = malloc(sizeof(struct Node));
    newNode->data = val;

    //make the new node points to the head node
    newNode->next = head;

    //make the new node as head node
    //so that head will always point the last inserted data
    head = newNode;
}

void pop()
{
    //temp is used to free the head node
    struct Node *temp;

    if(head == NULL)
        printf("Stack is Empty\n");
    else
    {
        printf("Poped element = %d\n", head->data);

        //backup the head node
        temp = head;

        //make the head node points to the next node.
        //logically removing the node
        head = head->next;

        //free the popped element's memory
        free(temp);
    }
}
```

```
//print the linked list
void display()
{
    struct Node *temp = head;

    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main()
{
    push(10);
    push(20);
    push(30);
    printf("Linked List\n");
    display();
    pop();
    printf("After the pop, the new linked list\n");
    display();
    pop();
    printf("After the pop, the new linked list\n");
    display();

    return 0;
}
```

Output:

```
Linked List
30->20->10->NULL
Poped element = 30
After the pop, the new linked list
20->10->NULL
Poped element = 20
After the pop, the new linked list
10->NULL
```

7. Write a C program to create a directed graph using Adjacency Matrix

Aim:

To write a C program to create a directed graph using Adjacency Matrix

Adjacency Matrix of a Directed Graph is a square matrix that represents the graph in a matrix form. In a directed graph, the edges have a direction associated with them, meaning the adjacency matrix will not necessarily be symmetric.

In a directed graph, the edges have a direction associated with them, meaning the adjacency matrix will not necessarily be symmetric. The adjacency matrix A of a directed graph is defined as follows:

What is Adjacency matrix of Directed graph?

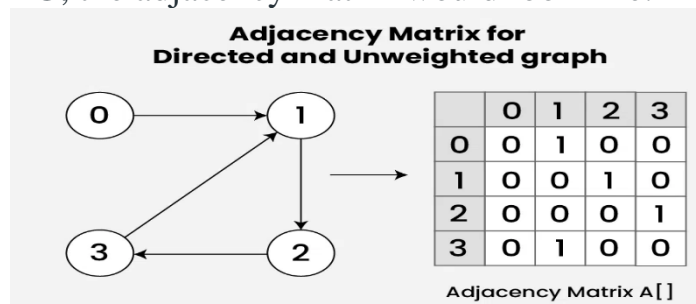
For a graph with N vertices, the adjacency matrix A is an $N \times N$ matrix where:

- $A[i][j]$ is 1 if there is a directed edge from vertex i to vertex j .
- $A[i][j]$ is 0 otherwise.

Adjacency Matrix for Directed and Unweighted graph:

Consider an Directed and Unweighted graph G with 4 vertices and 4 edges.

For the graph G , the adjacency matrix would look like:

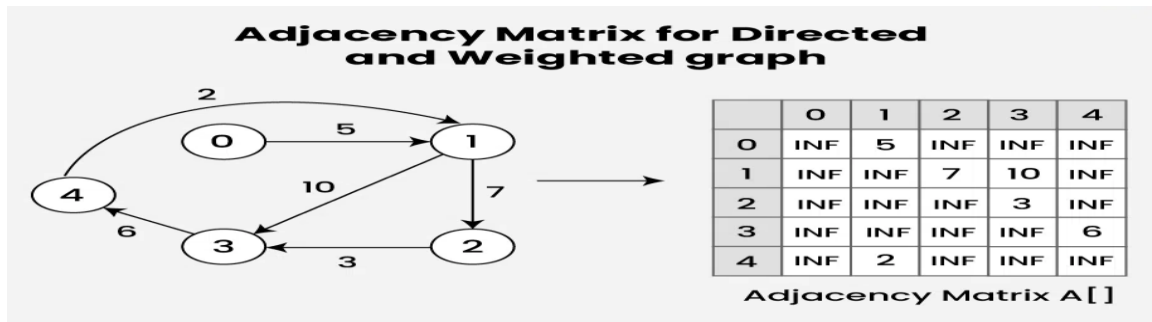


Here's how to interpret the matrix:

- $A[0][1] = 1$, there is an edge between vertex 0 and vertex 1.
- $A[1][2] = 1$, there is an edge between vertex 1 and vertex 2.
- $A[2][3] = 1$, there is an edge between vertex 2 and vertex 3.
- $A[3][1] = 1$, there is an edge between vertex 3 and vertex 1.
- $A[i][i] = 0$, as there are no self loops on the graph.
- All other entries with a value of 0 indicate no edge between the corresponding vertices.

Adjacency Matrix for Directed and Weighted graph:

Consider an Directed and Weighted graph G with 5 vertices and 6 edges. For the graph G , the adjacency matrix would look like:



Here's how to interpret the matrix:

- $A[0][1] = 5$, there is an edge between vertex 0 and vertex 1.
- $A[1][2] = 7$, there is an edge between vertex 1 and vertex 2.
- $A[2][3] = 3$, there is an edge between vertex 2 and vertex 3.
- $A[3][1] = 10$, there is an edge between vertex 3 and vertex 1.
- $A[i][i] = 0$, as there are no self loops on the graph.
- All other entries with a value of 0 indicate no edge between the corresponding vertices.

Properties of Adjacency Matrix of Directed Graph:

1. **Diagonal Entries:** The diagonal entries A_{ii} are usually set to 0, assuming the graph has no self-loops.
2. **Out-degree and In-degree:** The number of 1's in a row i (out-degree) indicates the number of outgoing edges from vertex i , while the number of 1's in a column j (in-degree) indicates the number of incoming edges to vertex j .

Algorithm:

1. Start
2. Create a 2D array(say $Adj[N+1][N+1]$) of size $N \times N$ and initialise all value of this matrix to zero
3. For each edge in $arr[][]$ (say X and Y), update value at $Adj[X][Y]$ and $Adj[Y][X]$ to 1, denotes that there is a edge between X and Y
4. Display the Adjacency Matrix after the above operation for all the pairs in $arr[][]$
5. Stop

Program:

```
// C program for the above approach

#include <stdio.h>


// N vertices and M Edges

int N, M;


// Function to create Adjacency Matrix

void createAdjMatrix(int Adj[][N + 1],

                    int arr[][2])

{

    // Initialise all value to this

    // Adjacency list to zero

    for (int i = 0; i < N + 1; i++) {

        for (int j = 0; j < N + 1; j++) {

            Adj[i][j] = 0;

        }

    }
```

```
}

// Traverse the array of Edges

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

    // Find X and Y of Edges

    int x = arr[i][0];

    int y = arr[i][1];

    // Update value to 1

    Adj[x][y] = 1;

    Adj[y][x] = 1;

}

}

// Function to print the created

// Adjacency Matrix

void printAdjMatrix(int Adj[][N + 1])
```

```
{  
  
    // Traverse the Adj[][]  
  
    for (int i = 1; i < N + 1; i++) {  
  
        for (int j = 1; j < N + 1; j++) {  
  
            // Print the value at Adj[i][j]  
  
            printf("%d ", Adj[i][j]);  
  
        }  
  
        printf("\n");  
  
    }  
}
```

// Driver Code

int main()

{

// Number of vertices

```
N = 5;

// Given Edges

int arr[][2]

    = { { 1, 2 }, { 2, 3 },

        { 4, 5 }, { 1, 5 } };

// Number of Edges

M = sizeof(arr) / sizeof(arr[0]);

// For Adjacency Matrix

int Adj[N + 1][N + 1];

// Function call to create

// Adjacency Matrix

createAdjMatrix(Adj, arr);

// Print Adjacency Matrix
```



```
printAdjMatrix(Adj);  
  
return 0;  
  
}
```

Output:

```
0 1 0 0 1  
1 0 1 0 0  
0 1 0 0 0  
0 0 0 0 1  
1 0 0 1 0
```

DATA STRUCTURES VIVA QUESTIONS

1Q) What is a Data Structure?

Ans) **Data Structure** is a data object together with the relationships that exists among the instances & among the individual elements that compose an instance.

2Q) Types of Data Structures and give examples?

Ans) There are two types of Data Structures:

1. Linear Data Structures:

A data structure is said to be linear if the elements form a sequence. It is sequential and continues in nature i.e. access the data in sequential manner. In linear data structure we can not insert an item in middle place and it maintains a linear relationship between its elements e.g.: Array, Linked list, Stack, Queue, Dequeue etc.

Non Linear Data Structures:

A data structure is said to be non-linear if elements do not form a sequence. (Not sequential).

It does not maintain any linear relationship between their elements. Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.

e.g.: Trees, Graphs.

[A data structure is linear if every item is related with next and previous item and it is nonlinear if it is attached with many of the items in specific ways to reflect relationship.]

3Q) What is a Singly Linked List?

Ans) Singly Linked List is a Sequence of dynamically allocated Storage elements, each element of which contains a pointer to its successor. A pointer to the first element of the list is called as head and a pointer to the last element of the list is called as tail used to keep track of the list elements.

4Q) What is Doubly Linked List?

Ans) In Doubly Linked List each element contains two pointers: One Pointer points to its successor and another to its predecessor (previous element). It is also called as two way linked list (traversing can be done in both directions).

STACKS: (LIFO DATA STRUCTURE)

6Q) What is a Stack? (LIFO Data Structure)

Ans) Stack is an ordered collection of items into which items can be inserted and deleted from

only one end called as “Top” of the

Stack. It is also called as LIFO list.(Last In FirstOut).

7Q) What is Stack Underflow?

Ans) If Stack is empty and POP operation is performed it is not possible to delete the items.This situation is called Stack Underflow.

8Q) What is Stack Overflow?

Ans) If Stack is full and PUSH operation is performed it is not possible to insert or Push thenew items into the stack. This situation is called Stack Overflow.

9Q) What are the Applications of Stack?

Ans) i) Stacks are used to convert Infix expression into Postfix.ii) Stacks are used to Evaluate Postfix Expression.iii) Stacks are used in recursion etc.

10Q) What is the use of Postfix expressions?

Ans) Postfix Expressions are easy to evaluate as postfix expressions does not make use ofoperator precedence not does it require the use of parenthesis

QUEUES: (FIFO DATA STRUCTURE)

11Q) What is a Queue?

Ans) It is an ordered collection of items into which items can be inserted from one end calledas REAR end and items are deleted from other end called as FRONT end of the Queue. Itis also called as FIRST IN FIRST OUT (FIFO) LIST).

12Q) What are the applications of Queues?

Ans) i) Queues are used in Breadth First Traversal of a Tree.ii) Queues are used in implementation of Scheduling algorithms of Operating Systems.

13Q) What is a Circular Queue?

Ans) In Circular Queue, the first position of the array is kept behind the last position of thearray.

14Q) Differentiate Linear Queue and Circular Queue?

Ans) In Linear Queue once the queue is full and the deletion is performed, even if first position is free(vacant) it is not possible to insert the item in that position whereas in Circular Queue it is possible since the first position is kept behind the last position.

15Q) What is Dequeue? (Double Ended Queue)

Ans) In Double Ended Queue insertion and deletion are possible from both the ends.

TREES:

16Q) What is a Tree?

Ans) Tree is a finite non-empty set of nodes with the following properties:i)

A designated node of the set is called as root of the tree and ii)

The remaining nodes are partitioned into $n \geq 0$ subsets, each of which is a tree.

Degree of a node:

The number of sub trees attached to a node is called degree of that node and the maximum degree of any node in a tree is called

degree of that tree.

[Note: In a general tree degree of a node is not fixed]

Nodes that have degree zero are called

Leaf or Terminal Nodes

. Consequently, the other nodes are referred to as

Non-Terminals.

The

Level of a node

is defined by letting the root be at level 0 or 1. The

height or depth of a tree

is defined to be the maximum level of any node in the tree

7Q) What is a Binary Tree?

Ans) A Binary tree T is a finite set of nodes with the following properties:i)

Either the set is empty, $T = \emptyset$ or ii)

The set consists of a root and exactly two distinct binary trees TL and TR , $T = \{r, TL, TR\}$. TL is the left subtree and TR is the right subtree of T .

[Note: Maximum degree of any node in a binary tree is 2. Degree of a node in a Binary Tree be either 0 or 1 or 2]

18Q) What is Tree Traversal?

List different Tree Traversal Techniques?

Ans) Visiting all nodes of a tree exactly once in a systematic way is called Tree Traversal. Different types of tree traversals are i)

Depth First Traversals

: PREORDER (N L R), INORDER (L N R) & POSTORDER (L R N)
Traversals. ii)

Breadth First Traversal

(or)

Level Order Traversal

(Visiting Level by level from left to right)

19Q) What is a Binary Search Tree? Give one example?

Ans) A Binary Search Tree T is a finite set of keys. Either the set is empty $T = \emptyset$, or the set

consists of a root “ r ” and exactly two binary search trees TL and TR , $T = \{r, TL, TR\}$ with

the following properties: i)

All the keys contained in the left subtree are less than the root key. ii)

All the keys contained in the right subtree are larger than the root key.[Note: Duplicates are not allowed in a Binary Search Tree]

20Q) What is the best, average and worst case time complexity of insertion, deletion and

Search operations in a Binary Search Tree?Ans) In Best and Avg case

$O(\log n)$

and in Worst case

$O(n)$.

21Q) What is an AVL Search Tree? What is AVL Balance Condition?

Ans) An AVL Search Tree is a balanced binary search tree. An empty binary tree is AVL balanced. A non

—

empty binary tree, $T = \{r, TL, TR\}$ is AVL balanced if both TL & TR are AVL balanced and $|h_L - h_R| \leq 1$

L

—

h_L

R

$|h_L - h_R| \leq 1$, Where : h

L

is the Height of the left subtree and h_R

R

is the Height of the right subtree

GRAPHS:

29Q) What is a Graph? Name various Graph Representation Techniques?

Ans) A Graph $G = (V, E)$ is Set of Vertices and Edges. A Graph can be represented as an Adjacency Matrix (or) as an Adjacency List.

30Q) What is difference between a Graph and a Tree?

Ans) A Graph contains Cycles (loops) but a Tree does not contain any cycles. [A Tree contains a root node but Graph does not contain the root node.]

31Q) What is a Spanning Tree?

Ans) A Spanning Tree $T = (V', E')$ is a Sub Graph of $G = (V, E)$ with following properties:i)

$V = V'$ [The vertices in a graph and spanning tree are same]ii)

T is Acyclic.iii)

T is connected.

[Note: If Graph has “n” vertices the

Spanning

Tree contains exactly “n

-

1” edges]

32Q) Name the methods to construct a Minimum cost Spanning Tree?

Ans)

Prim’s method and Kruskal’s method.