

# **DELHI TECHNOLOGICAL UNIVERSITY**

(Formerly Delhi College of Engineering)  
Shahbad Daultpur, Bawana Road, Delhi- 110042

## **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



### **DATA STRUCTURES LAB FILE**

**CO-201**

**Submitted By:**  
**Sandesh Shrestha**  
**2K21/CO/417**  
**A6 BATCH**

**Submitted To:**  
**Dr. Ashish Girdhar**  
**Department of COE**

## **INDEX: LIST OF PROGRAMS**

<b>S.No.</b>	<b>Program Name</b>	<b>Date of Experiment</b>	<b>Remarks</b>
1)	Write a program to Implement Linear Search in the C programming language	23.08.2022	
2)	Write a program to Implement Binary Search in the C programming language. Assume the list is already sorted.	23.08.2022	
3)	Write a program to insert an element at the mid-position in the One-dimensional array.	23.08.2022	
4)	Write a program to insert & delete a given row in a two-dimensional array.	24.08.2022	
5)	Write a program to implement a stack data structure and perform its operations -	06.09.2022	
6)	Write a program to implement two stacks using a single array.	06.09.2022	
7)	Write a program to implement Queue Data structure with its functions	13.09.2022	
8)	Write a program to find the minimum element of the stack in constant time using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack.	13.09.2022	
9)	Write a program to find the minimum element of the stack without using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack	20.09.2022	
10)	Write a program to reverse the first k elements of a given Queue	27.09.2022	

11)	Write a program to check whether the given string is Palindrome or not, using DEQUEUE	27.09.2022	
12)	Implement Tower of Hanoi Problem using Stack.	11.10.2022	
13)	Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position	18.10.2022	
14)	Write a program to split a given linked list into two sub-list as Front sub-list and back sub-list, if odd number of the element, then add the last element into the front list.	18.10.2022	
15)	Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value.	18.10.2022	
16)	Write a program to implement the Binary Tree using linked list and perform In-order traversal.	25.10.2022	
17)	Write a Program to check whether the given tree is a Binary Search Tree or not.	25.10.2022	
18)	Write a program to implement insertion in the AVL tree.	25.10.2022	
19)	Write an Algorithm to count the number of leaf nodes in an AVL tree	25.10.2022	
20)	Write a program to Delete a key from the AVL tree	25.10.2022	
21)	Write a program to implement Stack Data Structure using Queue.	01.11.2022	
22)	Write a program to implement Queue Data Structure using Stack.	01.11.2022	
23)	Write a program to implement Graph Data Structure and Its traversal BFS and DFS.	01.11.2022	

## Program 1

**Program Objective:** Program to implement linear search in C programming language.

**Program theory:** A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.

**Algorithm:** A simple approach to implement a linear search is:

- Begin with the leftmost element of array and one by one compare x with each element.
- If x matches with an element, then return the index.
- If x does not match with any of the elements, then return -1.

The **time complexity** of a linear search is **O(n) in worst case and O(1) in best case.**

**Program Code:**

```
C pro1_linearsearch.c X
C pro1_linearsearch.c
1  #include<stdio.h>
2  //function for linear search returns index if found
3  int linear_search(int arr[], int n, int key){
4      int i;
5      for( i = 0 ; i < n ; i++ ){
6          if(arr[i] == key)
7              return i;
8      }
9      //returns -1 if element not found
10     return -1;
11 }
12
13 int main(){
14     int arr[5],i,key,index;
15     for(i = 0; i < 5 ; i++){
16         printf("Enter the array elements: ");
17         scanf("%d",&arr[i]);
18     }
19     printf("Enter the key to search: ");
20     scanf("%d",&key);
21
22     index = linear_search(arr , 5 , key);
23     if(index == -1)
24         printf(" element not found.");
25     else
26         printf("The element found at index %d.",index);
27     return 0;
28 }
```

### Program Output:

```
PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Us
h }
Enter the array elements: 1
Enter the array elements: 2
Enter the array elements: 3
Enter the array elements: 4
Enter the array elements: 5
Enter the key to search: 3
The element found at index 2.
PS C:\Users\1226s\Desktop\ds lab ko codes> █
```

## **Program 2**

**Program Objective:** Write a program to Implement Binary Search in the C programming language. Assume the list is already sorted.

**Program theory:** Binary search is a 'divide and conquer' algorithm which requires the initial array to be sorted before searching. It is called binary because it splits the array into two halves as part of the algorithm. Initially, a binary search will look at the item in the middle of the array and compare it to the search terms. Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

**Algorithm:** The basic steps to perform Binary Search are:

- Begin with the mid element of the whole array as a search key.
- If the value of the search key is equal to the item, then return an index of the search key.
- Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty.

The **time complexity** of a binary search is:

<b>Case</b>	<b>Time Complexity</b>
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

**Program Code:**

C pro1\_linearsearch.c

C pro2\_binarysearch.c X

C pro2\_binarysearch.c

```
1  #include<stdio.h>
2  //function for binary search returns index if found
3  int binary_search(int arr[], int n, int key){
4
5      int s = 0 , e = n-1;
6      while(s<=e){
7          int mid = (s+e) / 2;
8          if(arr[mid] == key)
9              return mid;
10         else if(arr[mid] < key)
11             s = mid + 1;
12         else
13             e = mid - 1;
14     }
15     //returns -1 if element not found
16     return -1;
17 }
18
19 int main(){
20     int arr[5],i,key,index;
21     for(i = 0; i < 5 ; i++){
22         printf("Enter the array elements: ");
23         scanf("%d",&arr[i]);
24     }
25     printf("Enter the key to search: ");
26     scanf("%d",&key);
27
28     index = binary_search(arr , 5 , key);
29     if(index == -1)
30         printf(" element not found.");
31     else
32         printf("The element found at index %d.",index);
33     return 0;
34 }
```

### Program Output:

```
PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes"
h }
Enter the array elements: 1
Enter the array elements: 2
Enter the array elements: 3
Enter the array elements: 4
Enter the array elements: 5
Enter the key to search: 2
The element found at index 1.
PS C:\Users\1226s\Desktop\ds lab ko codes> 
```

### Program 3

**Program Objective:** Write a program to insert an element at the mid-position in the One-dimensional array in C programming language.

**Program theory:** Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. In this scenario, we are given the exact location i.e., middle (index) of an array where a new data element (value) needs to be inserted. First, we shall check if the array is full, if it is not, then we shall move all data elements from middle location one step downward. This will make place for a new data element.

**Algorithm:** The basic steps to perform insertion operation are:

- First get the element to be inserted, say x
- Then shift the array elements from middle position to one position forward, and do this for all the other elements next to pos.
- Insert the element x now at the position mid, as this is now empty.

The **time complexity** of a binary search is:

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

### Program Code:

```
C pro1_linearsearch.c X C pro2_binarysearch.c C pro7_getmin_extraspace.c X
C pro7_getmin_extraspace.c
1  /*to find minimum element in constant time but using
2  more space so we create another stack where we store the minimum values as we find them while pushing in stack*/
3
4  #include<stdio.h>
5  #include<stdlib.h>
6  typedef struct stack{
7      int size;
8      int top;
9      int* array;
10 }stack;
11
12 int is_empty(stack* s){
13     if(s->top == -1){
14         return 1;
15     }
16     return 0;
17 }
18
19 int is_full(stack* s){
20     if(s -> top == (s->size -1) ){
21         return 1;
22     }
23     return 0;
24 }
25
```



```

C pro7_getmin_extraspace.c
25
26 static int min = 9999; // high value taken so that first element is always less than 9999
27 void push(stack* s, stack* temp, int value){
28     if(is_full(s))
29         printf("Stack is full. Cannot push %d\n", value);
30     //if value to be stored is greater than minimum then store it in stack normally
31     if(value > min){
32         s->top++;
33         s->array[s->top] = value;
34     }
35     //else if smaller or equal to then store it in both stacks as well as update the minimum element
36     else{
37         s->top++;
38         s->array[s->top] = value;
39         temp->top++;
40         temp->array[temp->top] = value;
41         min = temp->array[temp->top];
42     }
43 }
44

```

```

C pro7_getmin_extraspace.c
44
45
46 void pop(stack* s, stack* temp){
47     //second stack's top element is the minimum element
48     int min = temp->top ;
49     if(is_empty(s)){
50         printf("\nStack is empty. Cannot pop %d \n");
51     }
52     //while popping if both stack ko same element encountered i.e. remove from both stack
53     if(s->array[s->top] <= min ){
54         s->top--;
55         temp->top--;
56         min = temp->array[temp->top];
57     }
58     else{
59         //simple pop from stack 1 if greater than min
60         s->top--;
61     }
62 }
63
64 void display (stack* s){
65     int i;
66     for( i=0; i <= s->top; i++){
67         printf("%d ", s->array[i]);
68     }
69 }
70
71 void get_min(stack* temp){
72     if(is_empty(temp)){
73         printf("Stack is empty.");
74     }
75     else{
76         printf("\n %d ", temp->array[temp->top]);
77     }
78 }
79

```

```

C pro7_getmin_extraspace.c
80  int main(){
81      stack* s = (stack*)malloc(sizeof(stack));
82      s->size = 10;
83      s->top = -1;
84      s->array = (int*)malloc(s->size * sizeof(int));
85      //create new stack to store minimum values
86      stack* min_stack = (stack*)malloc(sizeof(stack));
87      min_stack->size = 10;
88      min_stack->top = -1;
89      min_stack->array = (int*)malloc(min_stack->size * sizeof(int));
90      //push operations
91      push(s,min_stack,5);
92      push(s,min_stack,8);
93      push(s,min_stack,4);
94      push(s,min_stack,6);
95      push(s,min_stack,1);
96      push(s,min_stack,7);
97      printf("The main stack is: ");
98      display(s);
99      printf("\nThe auxiliary stack containing minimum values is ");
100     display(min_stack);
101     printf("\n min value is: ");
102     get_min(min_stack);
103     pop(s,min_stack);
104     pop(s,min_stack);
105     printf("\nAfter popping main stack: ");
106     display(s);
107     printf("\nAfter popping auxiliary stack: ");
108     display(min_stack);
109     printf("\n");
110     printf("\nmin value after popping is: ");
111
112     get_min(min_stack);
113
114     return 0;
115 }

```

## Program Output:

```

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes" & gcc pro7_getmin_extraspace.c & ./a.out
The main stack is: 5 8 4 6 1 7
The auxiliary stack containing minimum values is 5 4 1
min value is:
1
After popping main stack: 5 8 4 6
After popping auxiliary stack: 5 4

min value after popping is:
4
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## **Program 4**

**Program Objective:** Write a program to insert & delete a given row in a two-dimensional array.

**Program theory:** The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

**Algorithm Insert Row:** Create a new array of one more element in row size and another one-dimensional array to store the row to insert. Then there are three cases: If row that we currently are on is less than the row index to insert in, copy everything as it is in new array. If row is equal to row index to insert, then insert the new row. Then after that, copy everything after that row from the input (shifted back one against the current row index).

**Algorithm delete Row:** To delete a row, first we need to get the index of row to be deleted consider it as index. Then, we need to shift elements one index above from the deleted row index and then decrease the row number.

The **time complexity** is:

<b>Case</b>	<b>Time Complexity</b>
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

**Program Code:**

C pro4\_insert\_delete\_row.c

```
1  #include<stdio.h>
2  //display the array
3  void display(int arr[3][4],int m,int n){
4      int i,j;
5      for(i = 0; i<m; i++){
6          for(j=0; j<n; j++){
7              printf("%d ", arr[i][j]);
8          }
9          printf("\n");
10     }
11 }
12 //function to delete row
13 void delete_row(int a[3][4],int r,int c,int row_number){
14     int i,j;
15     for(i = 0; i<r; i++){
16         for(j=0; j<c; j++){
17             a[row_number][j] = a[row_number+1][j];
18         }
19         row_number++;
20     }
21     display(a,--r,c);
22 }
```

C pro4\_insert\_delete\_row.c

```
23 //function to insert row
24 void insert_row(int a[3][4],int r,int c,int index){
25     int in[4]={0,0,0,0};
26     int row = r+1;
27     int arr[row][4];
28     int i,j,k;
29
30     for(i = 0; i<row; i++){
31         for(j=0; j<c; j++){
32             if (i < index){
33                 arr[i][j] = a[i][j];
34             }
35             else if(index == i){
36                 arr[i][j] = in[j];
37             }
38             else{
39                 arr[i][j] = a[i-1][j];
40             }
41         }
42     }
43     display(arr,++r,c);
44 }
45
```

```

C pro4_insert_delete_row.c
46  int main(){
47      int r=3,c=4,row_number,i,j,k,ch;
48      int arr[3][4];
49      //read main array
50      printf("Enter the array elements : ");
51      for(i = 0; i<r; i++){
52          for(j=0; j<c; j++){
53              scanf("%d",&arr[i][j]);
54          }
55      }
56      printf("Array is: \n");
57      display(arr,r,c);
58
59      printf("enter 1 to insert row and 2 to delete row \n");
60      scanf("%d",&ch);
61      switch (ch) {
62      case 1:
63          printf("after inserting the row \n");
64          insert_row(arr,r,c,1);
65          break;
66      case 2:
67          printf("which row index to delete? ");
68          scanf("%d",&row_number);
69          delete_row(arr,r,c,row_number);
70          break;
71      }
72      return 0;
73  }

```

### Program Output:

```

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes"
insert_delete_row }
Enter the array elements : 1 2 3 4 5 6 7 8 9 10 11 12
Array is:
1 2 3 4
5 6 7 8
9 10 11 12
enter 1 to insert row and 2 to delete row
1
after inserting the row
1 2 3 4
0 0 0 0
5 6 7 8
9 10 11 12
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## **Program 5**

**Program Objective:** Write a program to implement a stack data structure and perform its operations -

- i. **Push():** Inserts an element into the stack
- ii. **Pop():** Removes an element from the stack
- iii. **Top():** Returns the top element of the stack
- iv. **Size():** Returns the size of stack

**Program theory:** A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack. The following are some common operations implemented on the stack:

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack.

### **Algorithm:**

The steps involved in the **PUSH** operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e.,  $\text{top} = \text{top} + 1$ , and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.

The steps involved in the **POP** operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top
- Once the pop operation is performed, the top is decremented by 1, i.e.,  $top = top - 1$ .

The **time complexity** is:

OPERATION	BEST TIME COMPLEXITY	WORST TIME COMPLEXITY	AVERAGE TIME COMPLEXITY	SPACE COMPLEXITY
Push	O(1)	O(N)	O(1)	O(1)
Pop	O(1)	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)

### Program Code:

```

C pro1_linearsearch.c X C pro2_binarysearch.c C pro7_getmin_extraspace.c C pro8_get
C pro5_stack.c
1  #include<stdio.h>
2  #include<stdlib.h>
3  typedef struct stack{
4      int size;
5      int top;
6      int* array;
7  }stack;
8
9  int is_empty(stack* s){
10     if(s->top == -1){
11         return 1;
12     }
13     return 0;
14 }
15
16 int is_full(stack* s){
17     if(s -> top == (s->size -1) ){
18         return 1;
19     }
20     return 0;
21 }
22
23 void push(stack* s,int value){
24     if(!is_full(s)){
25         s->top ++;
26         s->array[s->top] = value;
27     }
28     else{
29         printf("Stack is full. Cannot push %d\n",value);
30     }
31 }
32

```

```

C pro5_stack.c
33 void pop(stack* s){
34     if(!is_empty(s)){
35         printf("popped %d. \n ", s->array[s->top]);
36         s->top --;
37     }
38     else{
39         printf("\nStack is empty. Cannot pop %d \n");
40     }
41 }
42 void display (stack* s){
43     int i;
44     for( i=0; i <= s->top; i++){
45         printf("%d ", s->array[i]);
46     }
47 }
48
49 int main(){
50     stack* st = (stack*)malloc(sizeof(stack));
51     st->size = 5;
52     st->top = -1;
53     st->array = (int*)malloc(st->size * sizeof(int));
54     printf("is empty? %d \n",is_empty(st));
55     printf("is full? %d \n",is_full(st));
56     push(st,1);
57     push(st,2);
58     push(st,3);
59     push(st,4);
60     display(st);
61     pop(st);
62     push(st,5);
63     display(st);
64     pop(st);
65     display(st);
66     return 0;
67 }

```

### Program Output:

```

> cd "c:\Users\1226s\Desktop\ds lab ko codes"

is empty? 1
is full? 0
1 2 3 4 popped 4.
1 2 3 5 popped 5.
1 2 3
PS C:\Users\1226s\Desktop\ds lab ko codes>

```



## Program 6

**Program Objective:** Write a program to implement two stacks using a single array.

**Program theory:** The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction. In this case, the stack overflow condition occurs only when  $top1 + 1 = top2$ . This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

**Following functions must be supported by twoStacks:**

push1(int x) → pushes x to first stack

push2(int x) → pushes x to second stack

pop1() → pops an element from first stack and return the popped element

pop2() → pops an element from second stack and return the popped element

**Algorithm:**

Follow the steps below to solve the problem:

- Stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.
- Stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).
- Both stacks grow (or shrink) in opposite directions.
- To check for overflow, all we need to check is for space between top elements of both stacks. In this case, the stack overflow condition occurs only when  $top1 + 1 = top2$ .

The **time complexity** is:

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

## Program Code:

```
C pro6_twostacksinglearray.c
1  #include<stdio.h>
2  #include<stdlib.h>
3  typedef struct stack{
4      int size;
5      int top1;
6      int top2;
7      int* array;
8  }stack;
9
10 int is_empty(stack* s){
11     if(s->top1 == -1 && s->top2 == s->size){
12         return 1;
13     }
14     return 0;
15 }
16
17 int is_full(stack* s){
18     if((s -> top1) +1 == s->top2){
19         return 1;
20     }
21     return 0;
22 }
23
C pro6_twostacksinglearray.c
24 void push1(stack* s,int value){
25     if(!is_full(s)){
26         s->top1 ++;
27         s->array[s->top1] = value;
28     }
29     else{
30         printf("Stack is full. Cannot push %d\n",value);
31     }
32 }
33 void push2(stack* s,int value){
34     if(!is_full(s)){
35         s->top2--;
36         s->array[s->top2] = value;
37     }
38     else{
39         printf("Stack is full. Cannot push %d\n",value);
40     }
41 }
42
43 void pop1(stack* s){
44     if(!is_empty(s)){
45         printf("popped %d. \n ", s->array[s->top1]);
46         s->top1 --;
47     }
48     else{
49         printf("\nStack is empty. Cannot pop %d \n");
50     }
51 }
52
```

C pro6\_twostacksinglearray.c

```
53 void pop2(stack* s){
54     if(!is_empty(s)){
55         printf("popped %d. \n ", s->array[s->top2]);
56         s->top2 ++;
57     }
58     else{
59         printf("\nStack is empty. Cannot pop %d \n");
60     }
61 }
62
63 void display1 (stack* s){
64     int i;
65     for( i=0; i <= s->top1; i++){
66         printf("%d ", s->array[i]);
67     }
68     printf("\n");
69 }
70
71 void display2 (stack* s){
72     int i;
73     for( i=s->size-1; i > s->top2 ; i--){
74         printf("%d ", s->array[i]);
75     }
76     printf("\n");
77 }
78
```

```
int main(){
    stack* st = (stack*)malloc(sizeof(stack));
    st->size = 10;
    st->top1 = -1;
    st->top2 = st->size;
    st->array = (int*)malloc(st->size * sizeof(int));
    push1(st,1);
    push1(st,2);
    push1(st,3);
    push2(st,10);
    push2(st,20);
    push2(st,30);
    display1(st);
    display2(st);
    pop1(st);
    pop2(st);

    return 0;
}
```

### Program Output:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PowerShellLatest

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes\06_twostacksinglearray"
PS C:\Users\1226s\Desktop\ds lab ko codes\06_twostacksinglearray> }
1 2 3
10 20
popped 3.
popped 30.

PS C:\Users\1226s\Desktop\ds lab ko codes>
```

## **Program 7**

**Program Objective:** Write a program to implement Queue Data structure with its functions - Enqueue, Dequeue, IsEmpty, IsFull as below:

- i. **Enqueue():** Inserts an element at the end of the queue i.e. at the rear end
- ii. **Dequeue():** Removes and returns an element that is at the front end of the queue
- iii. **IsEmpty():** Returns 1 if queue is empty & 0 if not
- iv. **IsFull ():** Returns 1 if queue is full & 0 if not

**Program theory:** Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first. A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

A queue is an object (an abstract data structure - ADT) that allows the following operations:

- Enqueue: Add an element to the end of the queue
- Dequeue: Remove an element from the front of the queue
- IsEmpty: Check if the queue is empty
- IsFull: Check if the queue is full
- Peek: Get the value of the front of the queue without removing it

### **Enqueue Operation**

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

### **Dequeue Operation**

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

The complexity of enqueue and dequeue operations in a queue using an array is  $O(1)$ .

## Program Code:

```
C queuedynamicarray.c
1  #include <stdio.h>
2  #include<stdlib.h>
3  typedef struct queue{
4      int size;
5      int f;
6      int r;
7      int* arr;
8  }queue;
9  void display(struct queue* q){
10     for(int i=q->f;i<=q->r;i++){
11         printf("%d ",q->arr[i]);
12     }
13 }
14 int isempty(struct queue* q){
15     //element nai halena or dequeue grda grda rear vanda thulo vayo front
16     if(q->f == -1 || q->f>q->r){
17         return 1;
18     }
19     return 0;
20 }
21
22 int isfull(struct queue* q){
23     //max size samma full vayo
24     if(q->r == q->size-1){
25         return 1;
26     }
27     return 0;
28 }
29
```

```
void enqueue(struct queue* q,int value){
    if(q->r == q->size-1){
        printf("queue is full.");
    }
    else{
        q->f=0;
        q->r++;
        q->arr[q->r] = value;
    }
}

void dequeue(struct queue* q){
    if(q->f == -1){
        printf("queue is empty.");
    }
    else{
        printf("\nthe dequeued element is %d\n ",q->arr[q->f]);
        q->f++;
    }
}
```

```

int main()
{
    struct queue* qq = (struct queue*)malloc(sizeof(struct queue));
    qq->size = 10;
    qq->f = qq->r = -1;
    qq->arr = (int*) malloc (sizeof(int));

    enqueue(qq,1);
    enqueue(qq,2);
    enqueue(qq,3);
    enqueue(qq,4);
    enqueue(qq,5);
    enqueue(qq,6);
    enqueue(qq,7);
    display(qq);
    dequeue(qq);
    dequeue(qq);
    display(qq);
    return 0;
}

```

### Program Output:

```

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users
y }
1 2 3 4 5 6 7
the dequeued element is 1

the dequeued element is 2
3 4 5 6 7
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## Program 8

**Program Objective:** Write a program to find the minimum element of the stack in constant time using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack.

**Program theory:** The correct approach uses two stacks – the main stack to store the actual stack elements and an auxiliary stack to store the determine the minimum value so that we can get\_min in constant time by popping the top of auxiliary stack. This implementation requires few changes in push and pop operations.

### 1. Push operation

The idea is to push the new element into the main stack and push it into the auxiliary stack only if the stack is empty or the new element is less than or equal to the current minimum or top element of the auxiliary stack.

### 2. Pop operation

For pop operation, remove the top element from the main stack and remove it from the auxiliary stack only if it is equal to the current minimum element, i.e., a top element of both the main stack and the auxiliary stack is the same. After the minimum number is popped, the next minimum number appears on the top of the auxiliary stack.

### 3. Min operation

The top of the auxiliary stack always returns the minimum number since we are pushing the minimum number into the auxiliary stack and removing the minimum number from the auxiliary stack only if it is removed from the main stack.

## Program Code:

```
C pro7_getmin_extraspace.c
1  /*to find minimum element in constant time but using
2  more space so we create another stack where we store the minimum values as we find them while pushing in stack*/
3
4  #include<stdio.h>
5  #include<stdlib.h>
6  typedef struct stack{
7      int size;
8      int top;
9      int* array;
10 }stack;
11
12 int is_empty(stack* s){
13     if(s->top == -1){
14         return 1;
15     }
16     return 0;
17 }
18
19 int is_full(stack* s){
20     if(s -> top == (s->size -1) ){
21         return 1;
22     }
23     return 0;
24 }
```



```

C pro7_getmin_extraspace.c
25
26 static int min = 9999; // high value taken so that first element is always less than 9999
27 void push(stack* s, stack* temp, int value){
28     if(is_full(s))
29         printf("Stack is full. Cannot push %d\n", value);
30     //if value to be stored is greater than minimum then store it in stack normally
31     if(value > min){
32         s->top++;
33         s->array[s->top] = value;
34     }
35     //else if smaller or equal to then store it in both stacks as well as update the minimum element
36     else{
37         s->top++;
38         s->array[s->top] = value;
39         temp->top++;
40         temp->array[temp->top] = value;
41         min = temp->array[temp->top];
42     }
43 }
44
45
C pro7_getmin_extraspace.c
46 void pop(stack* s, stack* temp){
47     //second stack's top element is the minimum element
48     int min = temp->top ;
49     if(is_empty(s)){
50         printf("\nStack is empty. Cannot pop %d \n");
51     }
52     //while popping if both stack ko same element encountered i.e. remove from both stack
53     if(s->array[s->top] <= min ){
54         s->top--;
55         temp->top--;
56         min = temp->array[temp->top];
57     }
58     else{
59         //simple pop from stack 1 if greater than min
60         s->top--;
61     }
62 }
63
64 void display (stack* s){
65     int i;
66     for( i=0; i <= s->top; i++){
67         printf("%d ", s->array[i]);
68     }
69 }
70
71 void get_min(stack* temp){
72     if(is_empty(temp)){
73         printf("Stack is empty.");
74     }
75     else{
76         printf("\n %d ", temp->array[temp->top]);
77     }
78 }
79

```

```

C pro7_getmin_extraspace.c
79
80 int main(){
81     stack* s = (stack*)malloc(sizeof(stack));
82     s->size = 10;
83     s->top = -1;
84     s->array = (int*)malloc(s->size * sizeof(int));
85     //create new stack to store minimum values
86     stack* min_stack = (stack*)malloc(sizeof(stack));
87     min_stack->size = 10;
88     min_stack->top = -1;
89     min_stack->array = (int*)malloc(min_stack->size * sizeof(int));
90     //push operations
91     push(s,min_stack,5);
92     push(s,min_stack,8);
93     push(s,min_stack,4);
94     push(s,min_stack,6);
95     push(s,min_stack,1);
96     push(s,min_stack,7);
97     printf("The main stack is: ");
98     display(s);
99     printf("\nThe auxiliary stack containing minimum values is ");
100    display(min_stack);
101    printf("\n min value is: ");
102    get_min(min_stack);
103    pop(s,min_stack);
104    pop(s,min_stack);
105    printf("\nAfter popping main stack: ");
106    display(s);
107    printf("\nAfter popping auxiliary stack: ");
108    display(min_stack);
109    printf("\n");
110    printf("\nmin value after popping is: ");
111
112    get_min(min_stack);
113
114    return 0;
115 }

```

### Program Output:

```

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL
0 > cd "c:\Users\1226s\Desktop\ds lab ko codes"
etmin_extraspace }Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes"
The main stack is: 5 8 4 6 1 7
The auxiliary stack containing minimum values is 5 4 1
min value is:
1
After popping main stack: 5 8 4 6
After popping auxiliary stack: 5 4

min value after popping is:
4
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## Program 9

**Program Objective:** Write a program to find the minimum element of the stack without using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack

**Program theory:** Suppose we will push a number value into a stack with a minimum number, min. If the value is greater than or equal to the min, it is pushed directly into the stack. If it is less than min, push  $2 \times \text{value} - \text{min}$ , and update min as a value since a new minimum number is pushed. To pop, we pop it directly if the top of the stack (it is denoted as top) is greater than or equal to min. Otherwise, the number top is not the real pushed number. The real pushed number is stored as min. After the current minimum number is popped, we need to restore the previous minimum number,  $2 \times \text{min} - \text{top}$ .

### 1. Push operation

The idea is to push the new element into the stack and replace **static** integer minimum which is initially declared as 9999 as the first element in stack. Then after that, if the next element is greater than current min simply push it. Else if it is less than or equal to the minimum then push  $2 * \text{value} - \text{min}$  into stack.

### 2. Pop operation

For pop operation, remove the top element from the stack if it is greater than min. Else if top element is smaller than the min then new min will be  $2 * \text{min} - \text{top element value}$ . Then pop the top element. Done.

### 3. Min operation

The static integer maintained will give the value of min value in stack. Simply print it out.

## Program Code:

```
C pro8_getmin_constant.c
1 //so yo program ma chai constant time ani constant space ma minimum element nikalna xa so euta natra stack rakheko ani minimum element vetyo vane
  encrypt form ma stack ma haldine ani pop grda decrypt grne
2 //https://www.youtube.com/watch?v=QTrNy-00g7E
3 #include<stdio.h>
4 #include<stdlib.h>
5 typedef struct stack{
6     int size;
7     int top;
8     int* array;
9 }stack;
10
11 int is_empty(stack* s){
12     if(s->top == -1){
13         return 1;
14     }
15     return 0;
16 }
17
18 int is_full(stack* s){
19     if(s->top == (s->size -1) ){
20         return 1;
21     }
22     return 0;
23 }
24
```

```

25 static int min = 9999;
26
27 void push(stack* s, int value){
28     //if full
29     if(is_full(s))
30         printf("Stack is full. Cannot push %d\n",value);
31     //if stack empty then insert first element and update minimum value with that first element
32     if(is_empty(s)){
33         s->top++;
34         s->array[s->top] = value;
35         min = value;
36     }
37     //if value to be inserted in stack is less then the minimum value present in stack then that new value is min value so encrypt it and store
    encrypted in stack
38     else if(value < min){
39         s->top++;
40         s->array[s->top] = (2*value) - min;
41         min = value;
42     }
43     //else just store the value normally in stack
44     else{
45         s->top++;
46         s->array[s->top] = value;
47     }
48 }
49

```

```

C pro8_getmin_constant.c
49
50
51 void pop(stack* s){
52     //if empty
53     if(is_empty(s)){
54         printf("\nStack is empty. Cannot pop %d \n");
55     }
56     //while popping if minimum element gets popped then decrypt the element to get next minimum value
57     if(s->array[s->top] <= min){
58         min = ((2*min) - s->array[s->top]);
59         s->top--;
60     }
61     //simple pop if greater than min
62     else{
63         s->top--;
64     }
65 }
66
67 //function to display stack
68 void display (stack* s){
69     int i;
70     for( i=0; i <= s->top; i++){
71         printf("%d ", s->array[i]);
72     }
73 }
74
75 //function for getting minimum
76 void get_min(stack* temp){
77     if(is_empty(temp)){
78         printf("Stack is empty.");
79     }
80     else{
81         printf("\n minimum element : %d ", min);
82     }
83 }
84

```

```

84
85 int main(){
86     stack* s = (stack*)malloc(sizeof(stack));
87     s->size = 10;
88     s->top = -1;
89     s->array = (int*)malloc(s->size * sizeof(int));
90
91     //push operations
92     push(s,5);
93     push(s,8);
94     push(s,4);
95     push(s,6);
96     push(s,1);
97     push(s,7);
98     display(s);
99     printf("\n");
100    get_min(s);
101    pop(s);
102    pop(s);
103    get_min(s);
104    pop(s);
105    pop(s);
106    get_min(s);
107
108    return 0;
109 }

```

### Program Output:

```

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PowerShellLatest

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds lab ko codes"
n_constant }
5 8 3 6 -2 7

minimum element : 1
minimum element : 4
minimum element : 5
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## Program 10

**Program Objective:** Write a program to reverse the first k elements of a given Queue

**Program theory:** The idea is to use an auxiliary stack. Store the first k elements of the queue in a stack and pop it from the queue, then push it back to the queue and perform pop operation for n-k times and again push the popped element.

**Algorithm:**

- Create an empty stack.
- One by one dequeue first K items from given queue and push the dequeued items to stack.
- Enqueue the contents of stack at the back of the queue
- Dequeue (size - k) i.e. remaining elements from the front and enqueue them one by one to the same queue from rear.
- Done.

**Program Code:**

```
C pro1_linearsearch.c  C pro2_binarysearch.c  C pro7_getmin_extraspace.c  reverse_k_using_queue.cpp •
reverse_k_using_queue.cpp
1  #include<iostream>
2  #include<bits/stdc++.h>
3
4  using namespace std;
5  void reverse_after_k(queue<int>& q,int k){
6      stack<int> s;
7      int n = q.size();
8      for(int i = 1 ; i<=k ; i++){
9          s.push(q.front());
10         q.pop();
11     }
12
13     for(int i = 1 ; i<=k ; i++){
14         q.push(s.top());
15         s.pop();
16     }
17
18     for(int i = 1 ; i <= n-k ; i++){
19         q.push(q.front());
20         q.pop();
21     }
22 }
23 void prints(queue<int> q){
24     while (!q.empty())
25     {
26         cout << q.front() << " ";
27         q.pop();
28     }
29     cout << endl;
30 }
31
```

```

31
32  int main(){
33      queue<int> q;
34      int x, i = 0, k;
35      cout<<"Enter the elements in the queue: ";
36      for(; i<5 ; i++){
37          cin>>x;
38          q.push(x);
39      }
40      cout<<"Enter k: ";
41      cin>>k;
42      reverse_after_k(q,k);
43      prints(q);
44      return 0;
45  }
46

```

### Program Output:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://

PS C:\Users\1226s\Desktop\ds lab ko codes> cd "c:\Users\1226s\Desktop\ds
e_k_using_queue }
Enter the elements in the queue: 1 2 3 4 5
Enter k: 3
3 2 1 4 5
PS C:\Users\1226s\Desktop\ds lab ko codes>

```

## Program 11

**Program Objective:** Write a program to check whether the given string is Palindrome or not, using DEQUEUE

**Program theory:** A palindrome is a word, sentence, verse, or even number that reads the same backward or forward.

The solution to this problem will use a deque property. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character. Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.

### Program Code:

```
palindrome_check.cpp
1  #include<iostream>
2  #include<deque>
3  #include<string>
4  using namespace std;
5  int pal_check(string str){
6      deque<char> deq;
7      int length_of_String = str.length();
8      for(int i = 0; i< length_of_String ; i++){
9          deq.push_back(str[i]);
10     }
11     int check = 1;
12     while(deq.size() > 1 && check){
13         char first_letter = deq.front();
14         deq.pop_front();
15         char last_letter = deq.back();
16         deq.pop_back();
17
18         if( first_letter != last_letter){
19             check = 0; //not palindrome
20         }
21     }
22     return check ;
23 }
24 using namespace std;
25 int main(){
26     string a = "madam";
27     cout<< pal_check(a) <<endl;
28     cout<< pal_check("random") <<endl;
29     return 0;
30 }
```

### Program Output:

```
Install the latest Powershell
PS C:\Users\1226s\Desktop>
}
1
0
PS C:\Users\1226s\Desktop>
```



## Program 12

### Program Objective: Implement Tower of Hanoi Problem using Stack.

**Program theory:** Tower of hanoi is a classic problem where you try to move all the disks from one peg to another peg using only three pegs. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time. This problem can be easily solved by Divide & Conquer algorithm.

Let  $T(n)$  be the total time taken to move  $n$  disks from peg A to peg C

- Moving  $n-1$  disks from the first peg to the second peg. This can be done in  $T(n-1)$  steps.
- Moving larger disks from the first peg to the third peg will require first one step.
- Recursively moving  $n-1$  disks from the second peg to the third peg will require again  $T(n-1)$  step.

So, total time taken  $T(n) = T(n-1) + 1 + T(n-1)$

Relation formula for Tower of Hanoi is:  $T(n) = 2T(n-1) + 1$

### Program Code:

```
C > program12.cpp
1  #include<iostream>
2  #include<cmath>
3  #include<climits>
4  using namespace std;
5
6  class Stack{
7  public:
8      int capacity;
9      int top;
10     int arr[1000001];
11
12     Stack(int capacity){
13         this -> capacity = capacity;
14         this -> top = -1;
15     }
16
17     bool isFull(Stack * stack){
18         return (stack -> top == stack -> capacity - 1);
19     }
20
21     bool isEmpty(Stack * stack){
22         return (stack -> top == -1);
23     }
24
25     void push(Stack * stack, int item){
26         if(isFull(stack)){
27             return;
28         }
29         stack -> arr[++stack -> top] = item;
30     }
31
32     int pop(Stack * stack){
33         if(isEmpty(stack)){
34             return INT_MIN;
35         }
36         return stack -> arr[stack -> top--];
37     }
38 }
```

```

38
39 void moveDisks(Stack * src, Stack * dest, char s, char d){
40     int pole1 = pop(src);
41     int pole2 = pop(dest);
42
43     if(pole1 == INT_MIN){
44         push(src, pole2);
45         move(d, s, pole2);
46     }
47
48     else if(pole2 == INT_MIN){
49         push(dest, pole1);
50         move(s, d, pole1);
51     }
52
53     else if(pole1 > pole2){
54         push(src, pole1);
55         push(src, pole2);
56         move(d, s, pole2);
57     }
58
59     else{
60         push(dest, pole2);
61         push(dest, pole1);
62         move(s, d, pole1);
63     }
64 }
65
66 //
67     d = a;
68     a = temp;
69 }
70     total_num_of_moves = (int)(pow(2, num) - 1);
71     for(i = num; i >= 1; i--){
72         push(src, i);
73     }
74
75     for(i = 1; i <= total_num_of_moves; i++){
76         if(i % 3 == 1){
77             moveDisks(src, dest, s, d);
78         }
79         else if(i % 3 == 2){
80             moveDisks(src, aux, s, a);
81         }
82         else if(i % 3 == 0){
83             moveDisks(aux, dest, a, d);
84         }
85     }
86 }
87 };
88
89 int main(){
90     int num = 3;
91     Stack * src, * dest, * aux;
92     src = new Stack(num);
93     dest = new Stack(num);
94     aux = new Stack(num);
95     Stack * sol = new Stack(0);
96     sol -> Iterative(num, src, aux, dest);
97     return 0;
98 }
99
100
101
102
103
104
105
106
107
108

```

## Program Output:

```

PROBLEMS 70 OUTPUT DEBUG CONSOLE TERMINAL
> cd

Move the disk 1 from S to D
Move the disk 2 from S to A
Move the disk 1 from D to A
Move the disk 3 from S to D
Move the disk 1 from A to S
Move the disk 2 from A to D
Move the disk 1 from S to D
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```

### Program 13

**Program Objective:** Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.

**Program theory:** 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.

Array contains following limitations:

- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- 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,

- 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.
- 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.

Data Structure	Time Complexity				Space Complexity				
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

## Program Code:

```
PROGRAM13.cpp > insertAfter(Node *, int)
1  #include<iostream>
2  #include<stdlib.h>
3  using namespace std;
4  struct Node{
5      int data;
6      struct Node* next;
7  };
8  void insertAtBeginning(struct Node** head_ref, int new_data){
9      struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
10     new_node -> data = new_data;
11     new_node -> next = (*head_ref);
12     (*head_ref) = new_node;
13 }
14 void insertAfter(struct Node* prev_node, int new_data){
15     if(prev_node == NULL){
16         cout << "the given previous node cannot be NULL";
17         return;
18     }
19     struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
20     new_node -> data = new_data;
21     new_node -> next = prev_node -> next;
22     prev_node -> next = new_node;
23 }
24 void printList(struct Node* node){
25     while(node != NULL){
26         cout << node->data << " ";
27         node = node->next;
28     }
29 }
30 int main(){
31     struct Node* head = NULL;
32     insertAtBeginning(&head, 100);
33     insertAtBeginning(&head, 200);
34     insertAfter(head->next, 200);
35     insertAtBeginning(&head, 13);
36     insertAtBeginning(&head, 54);
37     insertAfter(head->next, 37);
38     insertAtBeginning(&head, 89);
39     insertAtBeginning(&head, 78);
40     insertAfter(head->next, 46);
41     cout << "Linked list : ";
42     printList(head);
43     cout<<endl;
44 }
```

## Program Output:

```
PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d:\S
AM13 }
Linked list : 78 89 46 54 13 37 200 100 200
PS D:\STUDY MATERIALS\SEM III\DSA\codes>
```

## Program 14

**Program Objective:** Write a program to split a given linked list into two sub-list as Front sub-list and back sub-list, if odd number of the element, then add the last element into the front list.

**Program theory:** 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. Probably the simplest strategy is to compute the length of the list, then use a for loop to hop over the right number of nodes to find the last node of the front half, and then cut the list at that point.

### Program Code:

```
find_min_max_in_bst.cpp  predecessor_successor.cpp  tree_creation.c  split_linkedlist.cpp X
split_linkedlist.cpp > printList(Node *)
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // A Linked List Node
5  struct Node
6  {
7      int data;
8      struct Node* next;
9  };
10
11 // Helper function to print a given linked list
12 void printList(struct Node* head)
13 {
14     struct Node* ptr = head;
15     while (ptr)
16     {
17         printf("%d ", ptr->data);
18         ptr = ptr->next;
19     }
20 }
21
22 // Helper function to insert a new node at the beginning of the linked list
23 void push(struct Node** head, int data)
24 {
25     struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
26     newNode->data = data;
27     newNode->next = *head;
28     *head = newNode;
29 }
30
31 // Return the total number of nodes in a list
32 int findLength(struct Node* head)
33 {
34     int count = 0;
35     struct Node* current = head;
36     while (current != NULL)
37     {
```

```

split_linkedlist.cpp > printList(Node *)
38     count++;
39     current=current->next;
40 }
41 return count;
42 }
43
44 /*
45  Split the given list's nodes into front and back halves
46  and return the two lists using the reference parameters.
47  If the length is odd, the extra node should go in the front list.
48  */
49 void frontBackSplit(struct Node* source, struct Node** frontRef, struct Node** backRef)
50 {
51     int len = findLength(source);
52     if (len < 2)
53     {
54         *frontRef = source;
55         *backRef = NULL;
56         return;
57     }
58
59     struct Node* current = source;
60
61     int hopCount = (len - 1) / 2;    // (figured these with a few drawings)
62     for (int i = 0; i < hopCount; i++) {
63         current = current->next;
64     }
65
66     // Now cut at current
67     *frontRef = source;
68     *backRef = current->next;
69     current->next = NULL;
70 }
71
72 int main(void)
73 {
74     // input keys
75
76     int main(void)
77     {
78         // input keys
79         int keys[] = {6, 3, 4, 8, 2, 9,10};
80         int n = sizeof(keys)/sizeof(keys[0]);
81
82         // points to the head node of the linked list
83         struct Node* head = NULL;
84
85         // construct a linked list
86         for (int i = n-1; i >= 0; i--) {
87             push(&head, keys[i]);
88         }
89
90         struct Node *a = NULL, *b = NULL;
91         frontBackSplit(head, &a, &b);
92
93         // print linked list
94         printf("Front List: ");
95         printList(a);
96
97         printf("\nBack List: ");
98         printList(b);
99
100        return 0;
101    }

```

## Program Output:

```

Install the latest PowerShell for new features

PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d
($?) { .\split_linkedlist }
Front List: 6 3 4 8
Back List: 2 9 10
PS D:\STUDY MATERIALS\SEM III\DSA\codes>

```

## Program 15

**Program Objective:** Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value. Example: Double Linked List 2, 5, 7, 8, 9, 10, 12, 16, 19, 25, and P=35 then pairs will be Pairs will be (10, 25), (16, 19).

**Program theory:** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward). A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

### Program Code:

```
C > G+ program15.cpp
1  #include<iostream>
2  using namespace std;
3
4  struct Node{
5      int data;
6      struct Node *next, *prev;
7  };
8
9  void pair_sum(struct Node *head, int X){
10     struct Node *start = head;
11     struct Node *end = head;
12     while (end -> next != NULL)
13         end = end -> next;
14
15     bool found = false;
16
17     while(start != end && end -> next != start){
18         if((start -> data + end -> data) == X){
19             found = true;
20             cout<<"("&<<start -> data<< " , "<<end -> data<<")"<<endl;
21             start = start -> next;
22
23             end = end -> prev;
24         }
25         else{
26             if((start -> data + end -> data) < X)
27                 start = start -> next;
28             else
29                 end = end -> prev;
30         }
31     }
32 }
```

```

29         end = end -> prev;
30     }
31 }
32
33     if(found == false)
34         cout<<"No pair with given sum exists";
35 }
36
37 void insertAtHead(struct Node **head, int data){
38     struct Node *temp = new Node;
39     temp -> data = data;
40     temp -> next = temp -> prev = NULL;
41     if(!(*head))
42         (*head) = temp;
43     else if(!(*head))
44         (*head) = temp;
45     else{
46         temp -> next = *head;
47         (*head) -> prev = temp;
48         (*head) = temp;
49     }
50 }
51
52 int main(){
53     struct Node *head = NULL;
54     insertAtHead(&head, 12);
55     insertAtHead(&head, 10);
56     insertAtHead(&head, 9);
57     insertAtHead(&head, 7);
58     insertAtHead(&head, 8);
59     insertAtHead(&head, 6);
60     insertAtHead(&head, 5);
61     int X = 19;
62     cout<<"Pair sum is : ";
63     pair_sum(head, X);
64     return 0;
65 }

```

### Program Output:

```

Copyright (c) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and enhancements!
https://aka.ms/powershell

PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d:\S
Pair sum is : (9, 10)
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```



## **Program 16**

**Program Objective:** Write a program to implement the Binary Tree using linked list and perform In-order traversal.

**Program theory:** Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Properties of Binary Tree:

- At each level of  $i$ , the maximum number of nodes is  $2^i$ .
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to  $(1+2+4+8) = 15$ . In general, the maximum number of nodes possible at height  $h$  is  $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$ .
- The minimum number of nodes possible at height  $h$  is equal to  $h+1$ .
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Types of Binary Tree

There are four types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

## Program Code:

```
C > C program16.c
1  #include<stdio.h>
2  #include<stdlib.h>
3  //structure node for tree containing one data and left and right node pointers
4  typedef struct node{
5      int data;
6      struct node* left;
7      struct node* right;
8  }node;
9
10 //function to create new node which takes data as argument and return the created node
11 node* createnode(int value){
12     node* temp = (node*)malloc(sizeof(node));
13     temp->data = value;
14     temp->left = NULL;
15     temp->right = NULL;
16     return temp;
17 }
18 //in-order traversal
19 void inorder(node* root){
20     if(root!=NULL){
21         inorder(root->left);
22         printf("%d ",root->data);
23         inorder(root->right);
24     }
25 }
26 int main(){
27     node* p = createnode(100);
28     node* p2 = createnode(25);
29     node* p3 = createnode(75);
30     node* p4 = createnode(15);
31     node* p5 = createnode(35);
32     node* p6 = createnode(85);
33
34     p->left = p2; //linking p2 to left of p1
35     p->right = p3; // linking p3 to left of p1
36     p2->left = p4;
37     p2->right = p5;
38     p3->right = p6;
39
40
41     inorder(p);
42     printf("\n");
43     return 0;
44 }
45
46
```

## Program Output:

```
PS D:\STUDY MATERIALS\SEM III\DSA\codes>
15 25 35 100 75 85
PS D:\STUDY MATERIALS\SEM III\DSA\codes>
```

## Program 17

**Program Objective:** Write a Program to check whether the given tree is a Binary Search Tree or not.

**Program theory:** A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

### Program Code:

```
BFS_and_DFS.cpp  tree_creation.c  program17.c  program16.c  program20.cpp  program16.exe
C > C program17.c
1  #include<stdio.h>
2  #include<stdlib.h>
3  //structure node for tree containing one data and left and right node pointers
4  typedef struct node{
5      int data;
6      struct node* left;
7      struct node* right;
8  }node;
9
10 //function to create new node which takes data as argument and return the created node
11 node* createnode(int value){
12     node* temp = (node*)malloc(sizeof(node));
13     temp->data = value;
14     temp->left = NULL;
15     temp->right = NULL;
16     return temp;
17 }
18
19 //in-order traversal
20 void inorder(node* root){
21     if(root!=NULL){
22         inorder(root->left);
23         printf("%d ",root->data);
24         inorder(root->right);
25     }
26 }
27
28 //checking if BST or not
29 int is_bst(node* root){
30     static node* prev = NULL; // so that it gets updated again and again in single memory
31
32     //first check if tree is empty or not
33     if(root != NULL){
34         //check for left if left bst haina vane return 0 gardine sidhai
35         if(!is_bst(root->left)){
```

```

C> C program17.c
35 //is_bst(root->left);
36     return 0;
37 }
38 //inorder ma chai agadi ko thulo xa paxadi ko vanda vane nope not a bst
39 //root->data naya naya element paudai grxa ani check grne thulo xa ki nai paila ko vanda
40 if(prev != NULL && root->data <= prev->data){
41     return 0;
42 }
43 //update prev as root
44 prev = root;
45 //check for right
46 return is_bst(root->right);
47 }
48
49
50 else{
51     return 1; // if tree khali then consider bst and return 1
52 }
53 }
54
55
56
57
58 //to insert new node
59 void insert(node* root, int key){
60     struct node* prev = NULL;
61     while(root!=NULL){
62         prev = root; //both pointing at root ani prev lai traverse grne
63         //suru ma check if it already exists or not in tree //if yes then exit
64         if(root->data == key){
65             return;
66         }
67         //if not search left and right sides
68         else if(key < root->data){
69             root=root->left;
70         }

```

```

C > C program17.c
67     //if not search left and right sides
68     else if(key < root->data){
69         root=root->left;
70     }
71     else{
72         root=root->right;
73     }
74 }
75 }
76
77 int main(){
78     node* p = createnode(100);
79     node* p2 = createnode(25);
80     node* p3 = createnode(75);
81     node* p4 = createnode(15);
82     node* p5 = createnode(35);
83     node* p6 = createnode(85);
84
85     p->left = p2; //linking p2 to left of p1
86     p->right = p3; // linking p3 to left of p1
87     p2->left = p4;
88     p2->right = p5;
89     p3->right = p6;
90     inorder(p);
91     printf("\n");
92     printf("%d", is_bst(p)); //1 if yes | 0 if no //
93     return 0;
94 }
95

```

### Program Output:

```

Install the latest PowerShell for new features

PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d:
15 25 35 100 75 85
0
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```

## Program 18

**Program Objective:** Write a program to implement insertion in the AVL tree.

**Program theory:** AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property ( $\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$ ).

- Left Rotation
- Right Rotation


SN	Rotation	Description
1	LL Rotation	The new node is inserted to the left sub-tree of left sub-tree of critical node.
2	RR Rotation	The new node is inserted to the right sub-tree of the right sub-tree of the critical node.
3	LR Rotation	The new node is inserted to the right sub-tree of the left sub-tree of the critical node.
4	RL Rotation	The new node is inserted to the left sub-tree of the right sub-tree of the critical node.

**Program Code:**

```

C > G+ program18.cpp
1  #include<iostream>
2  using namespace std;
3
4  class Node{
5  ∨ public:
6      int key;
7      Node *left;
8      Node *right;
9      int height;
10 };
11
12 ∨ int height(Node *N){
13 ∨     if(N == NULL){
14         return 0;
15     }
16     return N -> height;
17 }
18
19 ∨ int max(int a, int b){
20     return(a > b) ? a : b;
21 }
22
23 ∨ Node* newNode(int key){
24     Node* node = new Node();
25     node -> key = key;
26     node -> left = NULL;
27     node -> right = NULL;
28     node -> height = 1;
29     return(node);
30 }
31
32 ∨ Node *rightRotate(Node *y){
33     Node *x = y -> left;
34     Node *T2 = x -> right;
35     x -> right = y;
36     y -> left = T2;
37     y -> height = max(height(y -> left), height(y -> right)) + 1;
38     x -> height = max(height(x -> left), height(x -> right)) + 1;
39     return x;
40 }
41

```

C >  program18.cpp

```
42 Node *leftRotate(Node *x){
43     Node *y = x -> right;
44     Node *T2 = y -> left;
45     y -> left = x;
46     x -> right = T2;
47     x -> height = max(height(x -> left), height(x -> right)) + 1;
48     y -> height = max(height(y -> left), height(y -> right)) + 1;
49     return y;
50 }
51
52 int getBalance(Node *N){
53     if(N == NULL){
54         return 0;
55     }
56     return height(N -> left) - height(N -> right);
57 }
58
59 Node* insert(Node* node, int key){
60     if(node == NULL){
61         return(newNode(key));
62     }
63
64     if(key < node -> key){
65         node -> left = insert(node -> left, key);
66     }
67     else if(key > node -> key){
68         node -> right = insert(node -> right, key);
69     }
70     else{
71         return node;
72     }
73     node -> height = 1 + max(height(node -> left), height(node -> right));
74     int balance = getBalance(node);
75     if(balance > 1 && key < node -> left -> key){
76         return rightRotate(node);
77     }
78     if(balance < -1 && key > node -> right -> key){
79         return leftRotate(node);
80     }
81     if(balance > 1 && key > node -> left -> key){
82         node -> left = leftRotate(node -> left);
83     }
84     if(balance < -1 && key < node -> right -> key){
85         node -> right = rightRotate(node -> right);
86     }
87     return node;
88 }
```



```

80     }
81     if(balance > 1 && key > node -> left -> key){
82         node -> left = leftRotate(node -> left);
83         return rightRotate(node);
84     }
85     if (balance < -1 && key < node->right->key){
86         node -> right = rightRotate(node -> right);
87         return leftRotate(node);
88     }
89     return node;
90 }
91
92
93 //in-order traversal
94 void inorder(Node* root){
95     if(root!=NULL){
96         inorder(root->left);
97         printf("%d ",root->key);
98         inorder(root->right);
99     }
100 }
101
102 int main(){
103     Node *root = NULL;
104     root = insert(root, 1);
105     root = insert(root, 2);
106     root = insert(root, 3);
107     root = insert(root, 4);
108     root = insert(root, 6);
109     root = insert(root, 5);
110     cout << "Preorder traversal of the constructed AVL tree is : ";
111     inorder(root);
112     cout<<endl;
113     return 0;
114 }

```

### Program Output:

```

Install the latest PowerShell for new features and improvements! https://aka.ms/PowerShellLatest

PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d:\STUDY MATERIALS\SEM III\DSA\codes\"
Preorder traversal of the constructed AVL tree is : 1 2 3 4 5 6
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```

## **Program 19**

**Program Objective:** Write an Algorithm to count the number of leaf nodes in an AVL tree.

**Program theory:** AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a skewed Binary tree. If we make sure that the height of the tree remains  $O(\log(n))$  after every insertion and deletion, then we can guarantee an upper bound of  $O(\log(n))$  for all these operations. The height of an AVL tree is always  $O(\log(n))$  where  $n$  is the number of nodes in the tree.

The logic is the same for the leaf node, any node whose left and right children are null is known as a leaf node in a binary tree. They are the nodes that reside in the last level of a binary tree and they don't have any children. In order to count the total number of leaf nodes in a binary tree, you need to traverse the tree and increase the count variable whenever you see a leaf node.

Here are the actual steps to follow:

- 1) If the node is null return 0, this is also the base case of our recursive algorithm
- 2) If a leaf node is encountered then return 1
- 3) Repeat the process with left and right subtree
- 4) Return the sum of leaf nodes from both left and right subtree

**Program Code:**

```

C > G program19.cpp
1  #include<iostream>
2  using namespace std;
3  struct node{
4      int data;
5      struct node* left;
6      struct node* right;
7  };
8
9  unsigned int getLeafCount(struct node* node){
10     if(node == NULL)
11         return 0;
12     if(node -> left == NULL && node -> right == NULL)
13         return 1;
14     else{
15         return getLeafCount(node -> left) + getLeafCount(node -> right);
16     }
17 }
18
19 struct node* newNode(int data){
20     struct node* node = (struct node*)malloc(sizeof(struct node));
21     node -> data = data;
22     node -> left = NULL;
23     node -> right = NULL;
24     return(node);
25 }
26
27 int main(){
28     struct node *root = newNode(1);
29     root -> left = newNode(2);
30     root -> right = newNode(3);
31     root -> left-> left = newNode(4);
32     root -> left -> right = newNode(5);
33     root -> left -> left -> left = newNode(6);
34     root -> left -> left -> right = newNode(7);
35     cout<<"Leaf count of the tree is : "<<getLeafCount(root) << endl;
36     return 0;
37 }

```

### Program Output:

```

Install the latest PowerShell for new feature
PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "
Leaf count of the tree is : 4
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```

## Program 20

**Program Objective:** Write a program to Delete a key from the AVL tree.

**Program theory:** AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors. AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.


**Balance Factor (k) = height (left(k)) - height (right(k))**

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$


Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore; various types of rotations are used to rebalance the tree.

**Program Code:**

C >  program20.cpp

```
1  #include<iostream>
2  using namespace std;
3
4  class Node{
5  public:
6      int key;
7      Node *left;
8      Node *right;
9      int height;
10 };
11
12 int max(int a, int b);
13
14 int height(Node *N){
15     if(N == NULL){
16         return 0;
17     }
18     return N -> height;
19 }
20
21 int max(int a, int b){
22     return (a > b) ? a : b;
23 }
24
25 Node* newNode(int key){
26     Node* node = new Node();
27     node -> key = key;
28     node -> left = NULL;
29     node -> right = NULL;
30     node -> height = 1;
31     return(node);
32 }
33
34 Node *rightRotate(Node *y){
35     Node *x = y -> left;
36     Node *T2 = x -> right;
37     x -> right = y;
38     y -> left = T2;
39     y -> height = max(height(y -> left), height(y -> right)) + 1;
40     x -> height = max(height(x -> left), height(x -> right)) + 1;
41     return x;
42 }
43
44 Node *leftRotate(Node *x){
45     Node *y = x -> right;
46     Node *T2 = y -> left;
47     y -> left = x;
```

```
47     y -> left = x;
48     x -> right = T2;
49     x -> height = max(height(x -> left), height(x -> right)) + 1;
50     y -> height = max(height(y -> left), height(y -> right)) + 1;
51     return y;
52 }
53
54 int getBalance(Node *N){
55     if(N == NULL){
56         return 0;
57     }
58     return height(N -> left) - height(N->right);
59 }
60
```

C >  program20.cpp

```
60
61 Node* insert(Node* node, int key){
62     if(node == NULL){
63         return(newNode(key));
64     }
65     if(key < node -> key){
66         node -> left = insert(node -> left, key);
67     }
68     else if(key > node -> key){
69         node -> right = insert(node -> right, key);
70     }
71     else{
72         return node;
73     }
74     node -> height = 1 + max(height(node -> left), height(node -> right));
75     int balance = getBalance(node);
76     if(balance > 1 && key < node -> left -> key){
77         return rightRotate(node);
78     }
79     if(balance < -1 && key > node -> right -> key){
80         return leftRotate(node);
81     }
82     if(balance > 1 && key > node -> left -> key){
83         node -> left = leftRotate(node -> left);
84         return rightRotate(node);
85     }
86     if(balance < -1 && key < node -> right -> key){
87         node -> right = rightRotate(node -> right);
88         return leftRotate(node);
89     }
90     return node;
91 }
92
93 Node* minValueNode(Node* node){
94     Node* current = node;
95     while(current -> left != NULL)
96         current = current -> left;
97     return current;
98 }
99
100 Node* deleteNode(Node* root, int key){
101     if(root == NULL){
102         return root;
103     }
104     if(key < root -> key){
105         root -> left = deleteNode(root -> left, key);
106     }
```

C > G+ program20.cpp

```
106     }
107     else if(key > root -> key){
108         root -> right = deleteNode(root -> right, key);
109     }
110     else{
111         if((root -> left == NULL) || (root -> right == NULL)){
112             Node *temp = root -> left ? root->left : root->right;
113             if(temp == NULL){
114                 temp = root;
115                 root = NULL;
116             }
117             else{
118                 *root = *temp;
119             }
120             free(temp);
121         }
122         else{
123             Node* temp = minValueNode(root->right);
124             root->key = temp->key;
125             root->right = deleteNode(root->right,
126                                     temp->key);
127         }
128     }
129     if(root == NULL){
130         return root;
131     }
132     root -> height = 1 + max(height(root -> left), height(root -> right));
133     int balance = getBalance(root);
134     if(balance > 1 && getBalance(root -> left) >= 0){
135         return rightRotate(root);
136     }
137     if(balance > 1 && getBalance(root -> left) < 0){
138         root -> left = leftRotate(root -> left);
139         return rightRotate(root);
140     }
141     if (balance < -1 && getBalance(root -> right) <= 0){
142         return leftRotate(root);
143     }
144     if(balance < -1 && getBalance(root -> right) > 0){
145         root -> right = rightRotate(root -> right);
146         return leftRotate(root);
147     }
148     return root;
149 }
```

```

C > G+ program20.cpp
148     return root;
149 }
150
151 //in-order traversal
152 void inorder(Node* root){
153     if(root!=NULL){
154         inorder(root->left);
155         printf("%d ",root->key);
156         inorder(root->right);
157     }
158 }
159
160 int main(){
161     Node *root = NULL;
162     root = insert(root, 9);
163     root = insert(root, 5);
164     root = insert(root, 10);
165     root = insert(root, 0);
166     root = insert(root, 6);
167     root = insert(root, 11);
168     root = insert(root, -1);
169     root = insert(root, 1);
170     root = insert(root, 2);
171     cout<<"Inorder traversal of the constructed AVL tree is : ";
172     inorder(root);
173     root = deleteNode(root, 10);
174     cout<<"\n Inorder traversal after deletion of 10 : ";
175     inorder(root);
176     cout<<endl;
177     return 0;
178 }

```

## Program Output:

Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>

```

PS D:\STUDY MATERIALS\SEM III\DSA\codes> cd "d:\STUDY MATERIALS\SEM III\DSA\codes\C\" ; if ($?) {
Inorder traversal of the constructed AVL tree is : -1 0 1 2 5 6 9 10 11
Inorder traversal after deletion of 10 : -1 0 1 2 5 6 9 11
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C>

```



## **Program 21**

**Program Objective:** Write a program to implement Stack Data Structure using Queue.

**Program theory:** Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

Given a Queue data structure that supports standard operations like enqueue () and dequeue (). The task is to implement a Stack data structure using only instances of Queue and Queue operations allowed on the instances. A Stack can be implemented using two queues. Let Stack to be implemented be 's' and queues used to implement are 'q1' and 'q2'. Stack 's' can be implemented in two ways:

- Implement Stack using Queues By making push () operation costly: The idea is to keep newly entered element at the front of 'q1' so that pop operation dequeues from 'q1'. 'q2' is used to put every new element in front of 'q1'.
- Implement Stack using Queues by making pop () operation costly: The new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally, the last element is dequeued from q1 and returned.

**Program Code:**

C: > Users > 1226s > Desktop > G+ Q21.cpp

```
1  #include<iostream>
2  #include<queue>
3  using namespace std;
4  class stack{
5  queue<int> q1,q2;
6  public:
7  void push(int element){
8
9      // enqueue in secondary_queue
10     q2.push(element);
11
12     // add elements of primary_queue to secondary_queue
13     while(!q1.empty()){
14         q2.push(q1.front());
15         q1.pop();
16     }
17
18     // swapping the queues
19     queue<int> temp_queue = q1;
20     q1 = q2;
21     q2 = temp_queue;
22 }
23
24 void pop(){
25     if(q1.empty()){
26         return;
27     } else {
28         q1.pop();
29     }
30 }
31
32 int top(){
33     if(q1.empty()){
34         return -1;
35     } else {
36         return q1.front();
37     }
}
```

```

35  } else {
36      return q1.front();
37  }
38  }
39  void printStack()
40  {
41      queue<int> temp_queue = q1;
42
43      while(!temp_queue.empty()){
44          cout<<temp_queue.front()<<" ";
45          temp_queue.pop();
46      }
47      cout<<"\n";
48
49  }
50  };
51  int main(){
52      stack s;
53      s.push(10);
54      s.push(20);
55      s.push(30);
56      s.push(40);
57      s.printStack();
58      cout<<"Top: "<<s.top()<<"\n";
59      s.pop();
60      s.printStack();
61      return 0;
62  }

```

#### Program Output:

```

PS D:\STUDY MATERIALS\SEM III\
40 30 20 10
Top: 40
30 20 10
PS C:\Users\1226s\Desktop>

```

## Program 22

**Program Objective:** Write a program to implement Queue Data Structure using Stack.

**Program theory:** Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

A queue can be implemented using two stacks. Let queue to be implemented be *q* and stacks used to implement *q* be *stack1* and *stack2*. *q* can be implemented in two ways:

- Method 1 (By making enqueue operation costly)
- Method 2 (By making dequeue operation costly)

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enqueue operation, while method 2 (in dequeue operation) moves the elements once and moves elements only if *stack2* empty. So, the amortized complexity of the dequeue operation becomes  $\Theta(1)$ .

**Program Code:**

```
C > G+ program22.cpp
1  #include <iostream>
2  #include<stack>
3  //queue using stacks
4  using namespace std;
5  struct queue {
6      stack < int > s1, s2;
7      void push(int value) {
8          while (!s1.empty()) {
9              s2.push(s1.top());
10             s1.pop();
11         }
12         cout << "The element pushed is " << value << endl;
13         s1.push(value);
14         while (!s2.empty()) {
15             s1.push(s2.top());
16             s2.pop();
17         }
18     }
19     int pop() {
20         if (s1.empty()) {
21             cout << "Stack is empty";
22             exit(0);
23         }
24         int val = s1.top();
```

```

22     exit(0);
23 }
24 int val = s1.top();
25 s1.pop();
26 return val;
27 }
28 // Return the Topmost element from the queue
29 int top() {
30     if (s1.empty()) {
31         cout << "Stack is empty";
32         exit(0);
33     }
34     return s1.top();
35 }
36 };
37

```

```

38 int main() {
39     queue q;
40     q.push(3);
41     q.push(4);
42     q.push(20);
43     std::cout << "The element popped is " << q.pop() << endl;
44     q.push(5);
45     std::cout << "The top of the queue is " << q.top() << endl;
46     return 0;
47 }

```

### Program Output:

PROBLEMS 70 OUTPUT DEBUG CONSOLE TERMINAL

```

The top of the queue is 4
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C> cd "d:\STUDY
The element pushed is 3
The element pushed is 4
The element pushed is 20
The element popped is 3
The element pushed is 5
The top of the queue is 4
PS D:\STUDY MATERIALS\SEM III\DSA\codes\C> 

```

## **Program 23**

**Program Objective:** Write a program to implement Graph Data Structure and Its traversal BFS and DFS.

**Program theory:** A graph data structure is a collection of nodes that have data and are connected to other nodes. A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by  $G(E, V)$ .

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

- DFS (Depth First Search)
- BFS (Breadth First Search)

**BFS traversal of a graph** produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

**Depth-first search** is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So, the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

### Program Code:

```
BFS_and_DFS.cpp X
C > BFS_and_DFS.cpp
1  #include<iostream>
2  #include<queue>
3  using namespace std;
4
5  class search{
6      bool visited1[4]={false,false,false,false}; // unvisited nodes FOR DFS
7      bool visited_BFS[4]={false,false,false,false}; // unvisited nodes FOR BFS
8      //graph in form of adjacency matrix
9      int a[4][4] = {
10         {0,1,0,0},
11         {1,0,1,1},
12         {0,1,0,1},
13         {0,1,1,0}
14     };
15     public:
16     void dfs(int i){
17         visited1[i] = true;
18         cout << i<< " ";
19         for(int j = 0; j < 4 ;j++){
20             if(a[i][j]==1 && visited1[j] == false){
21                 dfs(j);
22             }
23         }
24     }
25
26     void bfs(queue<int> q, int i){
27         q.push(i);
28         visited_BFS[i] = true;
29         while(!q.empty()){
30             int u = q.front();
31             cout<<u<<" ";
32             q.pop();
33             for(int j = 0 ; j<4 ;j++){
34                 if(a[u][j] == 1 && visited_BFS[j] == false){
35                     q.push(j);
```

```

34     if(a[u][j] == 1 && visited_BFS[j] == false){
35         q.push(j);
36         visited_BFS[j] = true;
37     }
38 }
39 }
40 }
41 };
42 int main(){
43     queue<int> q;
44     search s;
45     cout<<"DFS ";
46     s.dfs(0);
47     cout<<endl;
48     cout<<"BFS ";
49     s.bfs(q,3);
50     return 0;
51 }
52

```

### Program Output:

```

Install the latest Pow

PS D:\STUDY MATERIALS\
DFS 0 1 2 3
BFS 3 1 2 0
PS D:\STUDY MATERIALS\

```