SUMMER TRAINING REPORT

On

Logic Building, Programming and Data Structures: Learn How to Think, & Program

Submitted by

Name – Nikhil Kumar

Reg No. – 12212136

Program Name: Bachelor of Technology

Under the Guidance of

Dr. Chirag Sharma

School of Computer Science & Engineering Lovely professional University, Phagwara

(June-July, 2024)

DECLARATION

I hereby declare that I have completed my summer training of DSA from June 2024 to July 2024 under the guidance of Dr.Chirag sharma. I have declared that I have worked with full dedication duringthese days of training and my learning outcomes fulfil the requirements of training for the award of degree of Bachelor of Technology, Lovely Professional university, Phagwara.

Name of Student: Nikhil Kumar

Registration no: 12212136

Acknowledgement

It is with sense of gratitude; I acknowledge the efforts of entire hosts of well-wishers who have in some way or other contributed in their own special ways to the success and completion of the Summer Training.

Successfully completion of any type technology requires helps from a number of people. I have also taken help from different people for the preparation of the report. Now, there is little efforts to show my deep gratitude to those helpful people.

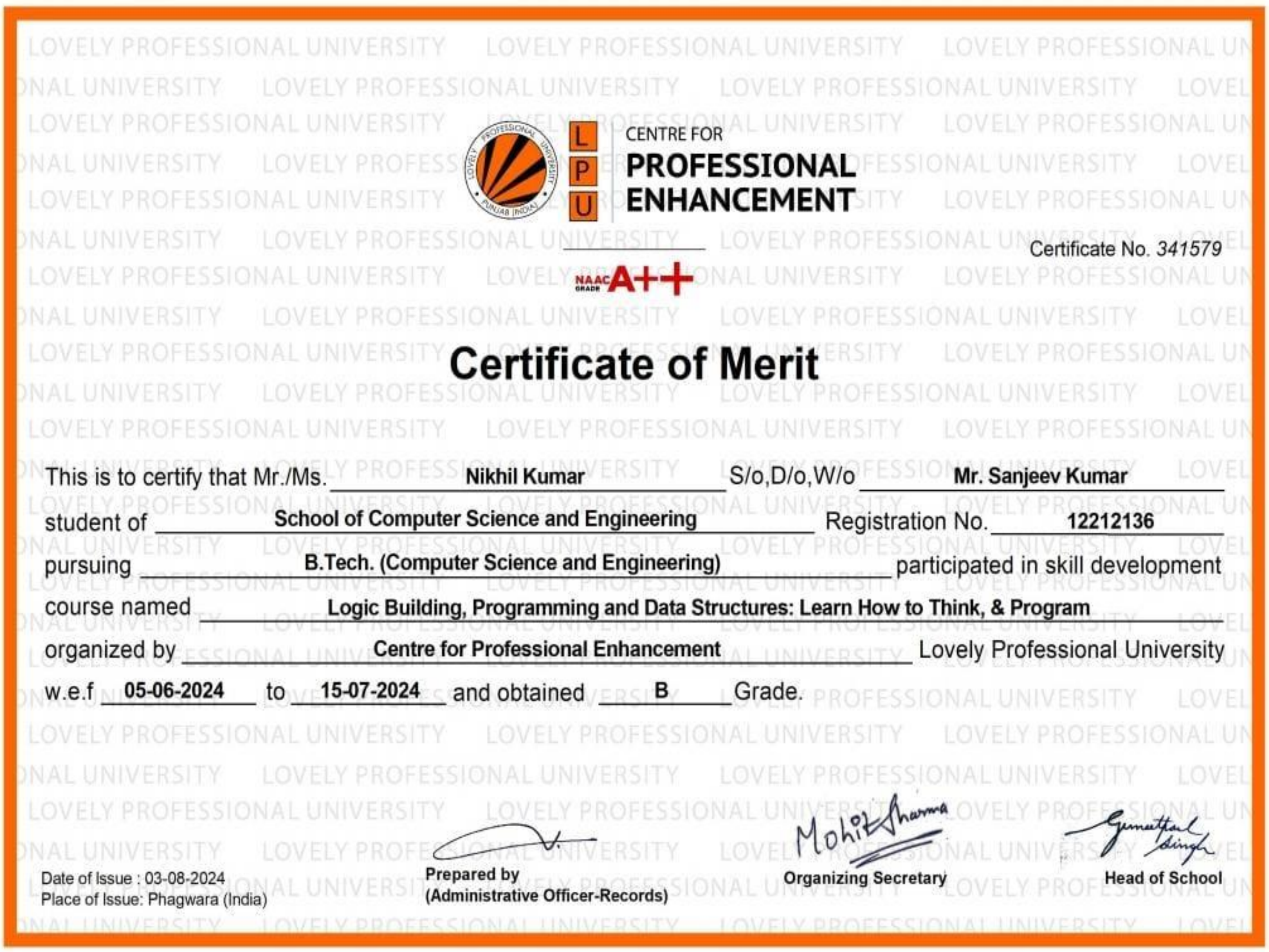
I would like to also thank my own college Lovely Professional University for offering such a course which not only improve my programming skill but also taught me other new technology.

Then I would like to thank my parents and friends who have helped me with their valuable suggestions and guidance for choosing this course.

Then I would like to thank my lecturer and Lovely Professional University which provided me knowledge of thefield with the best of their capabalities.

Last but not least I would like to thank my all classmates who have helped me a lot.

Training certificate from organization



CENTRE FOR
PROFESSIONAL ENHANCEMENT

NAAC GRADE A++

Certificate No. 341579

Certificate of Merit

This is to certify that Mr./Ms. **Nikhil Kumar** S/o,D/o,W/o **Mr. Sanjeev Kumar** student of **School of Computer Science and Engineering** Registration No. **12212136** pursuing **B.Tech. (Computer Science and Engineering)** participated in skill development course named **Logic Building, Programming and Data Structures: Learn How to Think, & Program** organized by **Centre for Professional Enhancement** Lovely Professional University w.e.f **05-06-2024** to **15-07-2024** and obtained **B** Grade.

Date of Issue : 03-08-2024
Place of Issue: Phagwara (India)

Prepared by
(Administrative Officer-Records)

Mohit Sharma
Organizing Secretary

[illegible]
Head of School

CONTENT

- INPROVING LINEAR SEARCH
- BINARY SEARCH
- BINARY SEARCH ALGORITHIM
- ANALYSIS OF BINARY SEARCH
- GET ()/SET ()/AVG () FUNCTIONS ON ARRAY
- REVERSE AND SHIFT AN ARRAY
- CHECK IF ARRAY IS SORTED
- MERGING ARRAYS
- SET OPERATOR ON ARRAY -: UNION, INTERSECTION, DIFFERNCE

6) STRINGS

- INTRODUCTION TO STRINGS
- FINDING LENGTH OF A STRING
- CHANGING CASE OF A STRING
- COUNTING WORDS AND VOWELS ON A STRING
- VALIDATING A STRING
- REVERSING A STRING
- COMPARING STRINGS CHECKING PLAINDROME
- FINDING DUPLICATES IN A STRING
- FINDING FUPLICATES IN A SGTRING USING BITWISE OPERATIONS
- CHECKING IF 2 STRINGS ARE ANAGRAM

7) MATRICES

- INTRODUCTION
- DIAGONAL MATRIX
- C++ CLASS FOR DIAGONAL MATRIX
- LOWER TRIANGULAR MATRIX ROW-MAJOR MAPPING
- UPPER TRIANGULAR MATRIX ROW-MAJOR MAPPING
- UPPER TRIANGULAR MATRIX COLIMN-MAJOR MAPPING
- SYMMETRIC MATRIX
- TRI-DIGONAL AND TRI-BAND MATRIX
  TOEPLITZ MATRIX

MENU DRIVEN PROGRAMS FOR MATRICES

8) LINKED LIST

- WHY WE NEED DYNAMIC DATASTRUCTURES
- ABOUT LINKED LIST
- DISPLAY LINKED LIST
- RECURSIVE DISPLAY FOR LINKED LIST
- COUNTING NODES IN LINKED LIST
- SUM OF ALL ELEMENTS IN A LINKED LIST
- SEARCHING IN A LINKED LIST • INSERTING IN A LINKED LIST
- CREATING A LINKED LIST USING INSERT
- CREATING A LINKED LIST BY INSERTING AT LAST
- INSERTING IN A SORTED LINKED LIST
- DELETING FROM A LINKED LIST • CHECK IF A LINKED LIST IS SORTED
- REMOVE DUPLICATES FROM A SORTED LINKED LIST
- REVERSING A LINKED LIST
- REVERSING USING SLIDING POINTERS
- CONCATENATING 2 LINKED LIST • MERGING 2 LINKED LISTS • CHECK FOR LOOP IN LINKED LIST
- CIRCULAR LINKED LIST
- DISPLAY CIRCULAR LINKED LIST
- INSERTING IN A CIRCULAR LINKED LIST
- DELETING FROM A CIRCULAR LINKED LIST
- DOUBLY LINKED LIST
- INSERTING IN A BOUBLY LINKED LIST
- FELETING FROM A DOUBLY LINKED LIST
- REVERSE A DOUBLY LINKED LIST • CIRCULAR DOUBLY LINKED LIST
- COMPARISON OF ARRAY WITH LINKED LIST

9) STACK

- INTRODUCTION TO STACK
- STACK USING ARRAY
- IMPLEMENTATION OF STACK USING ARRAY
- STACK USING LINKED LIST
- STACK OPERTIONS USING LINKED LIST
- PARENTHSIS MATCHING

<u>INTRODUCTION</u>

a) STACK VS HEAP MEMORY:

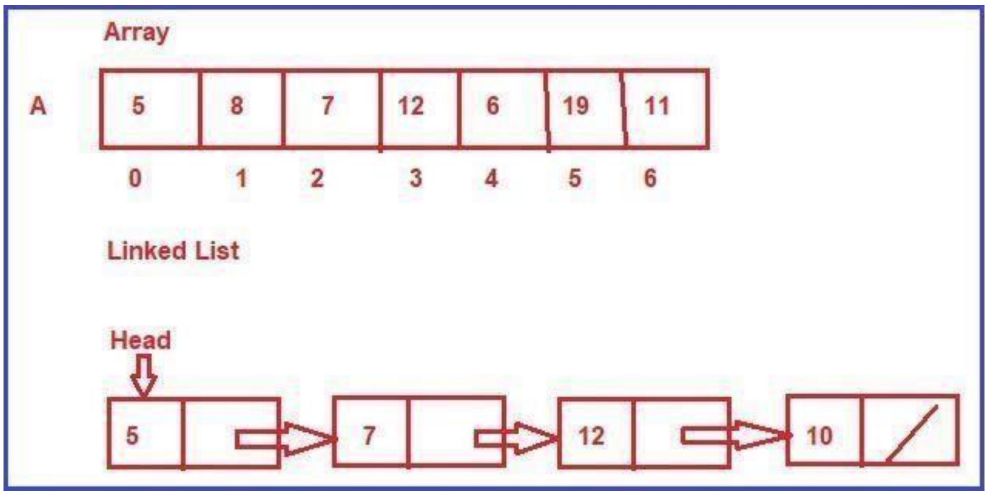
Stack Allocation: The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de- allocated. This all happens using some predefined routines in the compiler.

Heap Allocation: The memory is allocated during the execution of instructions written by programmers. Note that the name heap has nothing to do with the heap data structure. It is called heap because it is a pile of memory space available to programmers to allocate and de- allocate. Every time when we made an object it always creates in Heap- space and the referencing information to these objects are always stored in Stack-memory.

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and de-allocation are automatically done by the compiler whereas in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than the handling of the stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack have a small region of memory and is cache-friendly, but in case of heap frames which are dispersed throughout the memory so it causes more cache misses.

b) PHYSICAL VS LOGICAL DATA STRUCTURES



The Array and Linked List are the two physical data structures. We can have more physical data structures by taking the combination of these two data structures i.e. array and linked list.

Logical Data Structure:

The following is a list of logical data structures.

1. **Stack**
2. **Queue**
3. **Tree**
4. **Graph**
5. **Hash Table**

Stack works on the discipline of LIFO i.e. Last in First Out. Queue works on the discipline of FIFO i.e. First in First Out. The trees are a non-linear data structure and they will be organized in a hierarchy. The graph is a collection of nodes and the links between the nodes. These data structures are actually used in applications and algorithms.

The most important point that you need to remember is for implementing the logical data structures (Stack, Queue, Trees, Graphs, Hash Tables) we either use an array or linked list or a combination of array and linked list physical data structures. So, that is all we have given the introduction of various types of data structures. This was just the introduction to give us awareness.

c) ADT (ABSTRACT DATA TYPE)

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword <Abstract= is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Let us see some operations of those mentioned ADT –

- Stack –

o isFull(), This is used to check whether stack is full or not o isEmpry(), This is used to check whether stack is empty or not

o push(x), This is used to push x into the stack ○ pop(), This is used to delete one element from top of the stack o peek(), This is used to get the top most element of the stack o size(), this function is used to get number of elements present into the stack

- Queue − ○ isFull(), This is used to check whether queue is full or not

o isEmpry(), This is used to check whether queue is empty or not o insert(x), This is used to add x into the queue at the rear end o delete(), This is used to delete one element from the front end of the queue

o size(), this function is used to get number of elements present into the queue

- List − o size(), this function is used to get number of elements present into the list

o insert(x), this function is used to insert one element into the list o remove(x), this function is used to remove given element from the list

o get(i), this function is used to get element at position i ○ replace(x, y), this function is used to replace x with y value

d) TIME AND SPACE COMPLEXITY: Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity S(p) of any algorithm p is S(p) = A + Sp(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

RECURSION

a) HOW RECURSION WORKS:

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

b) TAIL RECURSION:

Tail recursion *is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.*

```
   // An example of tail recursive function void
print(int n)
 { if (n < 0) return;
      cout << " " <<
      n;
      // The last executed statement is recursive call print(n
      - 1);
 }
```

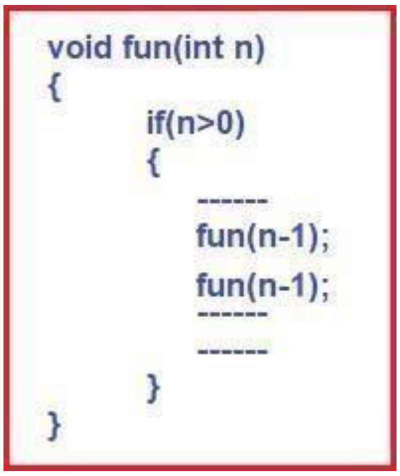
c) HEAD RECURSION:

If a recursive function calling itself and that recursive call is the first statement in the function then it9s known as **Head Recursion.** There9s no statement, no operation before the call. The function doesn9t have to process or perform any operation at the time of calling and all operations are done at returning time. **Example:**

```
// C++ program showing Head Recursion

#include <bits/stdc++.h>
using namespace std;
// Recursive function void fun(int
n)
{ if (n > 0) {

        // First statement in the function fun(n
        - 1);
cout << " "<< n;
      }
  }
  // Driver code int main()
  {  int  x  =  3;
      fun(x);
      return 0;
  }
```

d) TREE RECURSION:

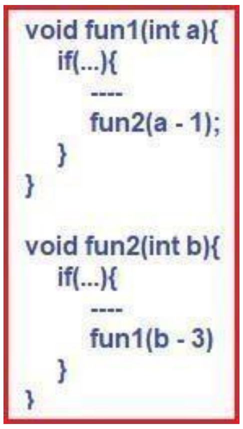
A function that calls itself, is a recursive function. If it is calling itself more than one time, then it is a tree recursion. Please have a look at the below example which is an example of a tree recursive function. As you can see fun is calling itself 2 times (more than 1 time). In our example, we have shown the recursive call like 2 times, but it can also be more than 2 times.



```
void fun(int n)
{
        if(n>0)
        {
              ------
              fun(n-1);
              fun(n-1);
              ------
              ------
        }
}
```

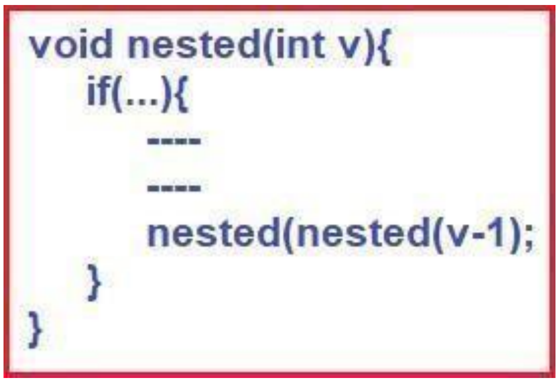
e) INDIRECT RECURSION:

In Indirect recursion, there may be more than one function and they are calling one another in a circular fashion. For example, if the first function calls the second function, the second function calls the third function, and again the third function calls back the first function, then it becomes a cycle which is also a recursion i.e. Indirect Recursion.


```
void fun1(int a){
    if(...){
        ----
        fun2(a - 1);
    }
}

void fun2(int b){
    if(...){
        ----
        fun1(b - 3)
    }
}
```

f) <u>NESTED RECURSION:</u>

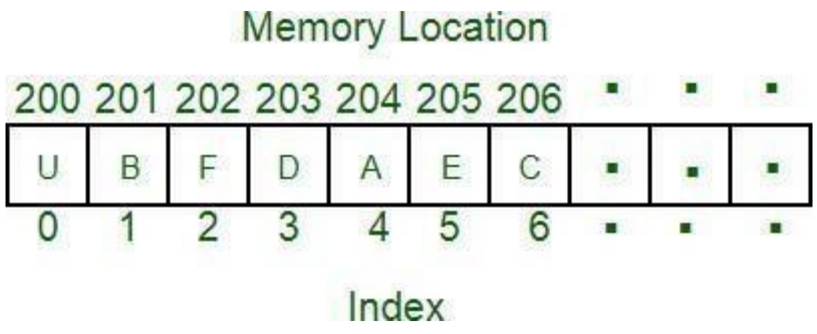
In Nested recursion, the recursive function will pass the parameter as a recursive call. For better understanding, please have a look at the below image. In the below image nested is a recursive function which calls itself. But the parameter itself is a recursive call i.e. nested(v-1). This is called nested recursion.


```
void nested(int v){
    if(...){
        ----
        ----
        nested(nested(v-1);
    }
}
```

<u>ARRAY REPRESENTATION:</u>

) <u>INRODUCTION TO ARRAY:</u>

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



) STATIC VS DYNAMIC ARRAYS:

Dynamic Arrays:

Dynamic arrays differ from static arrays in that they don’t have a fixed size. However, this is at the cost of having quick appends. How a dynamic array works is that it resizes itself whenever it runs out of space. This is usually by doubling itself. For example, if its size defaulted to 10 indices then it would double to 20. Let’s take a closer look at how it does this:

1. When you allocate a dynamic array your language of choice will make a static array with a set size. Let’s say that this size is 10.

2. Let’s say you go to append an 11th item to your array. This will make your array run out of space causing it to create a bigger array that’s double its size (20).

Static Arrays:

When you allocate an array in a low-level language like C++ or Java, you have to specify upfront how many indices you want your array to have like so:

This makes the array static because it only has five indices that you can use. This makes it impossible to append items when all five indices are filled with values. The upside to this is that if you know that you are only going to have five elements, then appending to this array becomes worst case O(1) time.

) 2D ARRAYS:

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

Declaration of two dimensional Array in C

The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

) HOW TO INCREASE SIZE OF ARRAY:

If you want to change the size, you need to **create a new array of the desired size, and then copy elements from the old array to the new array, and use the new array**. In our example, arr can only hold int values. Arrays can hold primitive values, unlike ArrayList, which can only hold object values.

ARRAY ADT:

a) ARRAY ADYT:

ADT indicates for Abstract Data Type.

Arrays are defined as ADT’s because they are capable of holding contiguous elements in the same order. And they permit access for the specific element via index or position.

They are abstract because they can be String, int or Person

Advantages

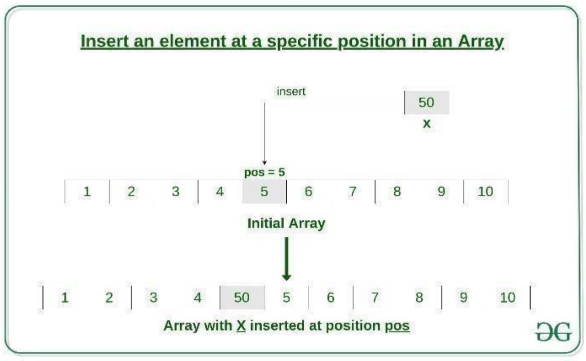
- Fast, random access of items or elements.
- Very memory efficient, very little memory is needed other than that needed to store the contents.

Disadvantages

- Slow insertion and deletion of elements
- Array size must be known when the array is created and is fixed (static)

b) Insertion in array:

Given an array arr of size **n**, this article tells how to insert an element **x** in this array arr at a specific position **pos**.


Insert an element at a specific position in an Array
insert
50
x
pos = 5
1 2 3 4 5 6 7 8 9 10
Initial Array
1 2 3 4 50 5 6 7 8 9 10
Array with X inserted at position pos

```
#include <stdio.h>
int main()
{ int arr[100] = { 0 }; int
    i, x, pos, n = 10;
    // initial array of size 10
    for (i = 0; i < 10; i++)
    arr[i] = i + 1;

    // print the original array for
    (i = 0; i < n; i++) printf("%d
    ", arr[i]); printf("\n");
    // element to be inserted x
    = 50;
```

```
     // position at which element
     // is to be inserted pos = 5;

     // increase the size by 1 n++;

     // shift elements forward for
     (i = n - 1; i >= pos; i--)
     arr[i] = arr[i - 1];

     // insert x at pos arr[pos
     - 1] = x;

     // print the updated array
     for  (i  =  0;  i  <  n;  i++)
     printf("%d   ",   arr[i]);
     printf("\n");
return 0; }
```

c) DELETING FROM ARRAY:

Given an array and a number 8x9, write a function to delete 8x9 from the given array. We assume that array maintains two things with it, capacity and size. So when we remove an item, capacity does not change, only size changes.

```
        #include<bits/stdc++.h> using
        namespace std;
        // This function removes an element x from arr[] and
        // returns new size after removal (size is reduced only
        // when x is present in arr[]
        int deleteElement(int arr[], int n, int x) {

        // Search x in array
int i; for (i=0; i<n; i++) if
(arr[i] == x) break;

        // If x found in array if (i
        < n)
        {
            // reduce size of array and move all
            // elements on space ahead n = n - 1;
            for (int j=i; j<n; j++) arr[j] =
            arr[j+1];
        } return
        n;

        }
```

d) LINEAR SEARCH:

Follow the given steps to solve the problem:

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn9t match with any of the elements, return -1.

```
#include <stdio.h>
int search(int arr[], int N, int x)
{ int i; for (i = 0; i < N;
    i++) if (arr[i] == x)
        return i;
    return -1; }

// Driver’s code int main(void)
{ int arr[] = { 2, 3, 4, 10, 40 }; int
    x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);
    // Function call
    int result = search(arr, N, x);
    (result == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result); return
0; }
```

e) BINARY SEARCH:

Binary Search 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.

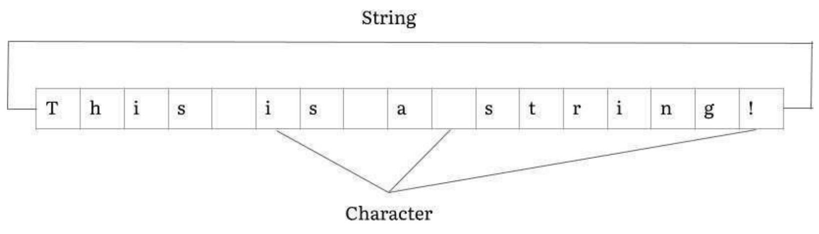
1) ITERATIVE METHOD:

```
 binarySearch(arr,  x,  low,  high)
     repeat till low = high mid =
     (low  +  high)/2  if  (x  ==
     arr[mid]) return mid


                  else if (x > arr[mid]) // x is on the right side
                       low = mid + 1


                  else // x is on the left side high = mid – 1 2)
   RECURSIVE METHOD:
binarySearch(arr, x, low, high)
              if low > high return
                   False

              else mid = (low + high) /

                   2 if x == arr[mid]

                   return mid
else if x > arr[mid] // x is on the right return

binarySearch(arr, x, mid + 1, high)
side

                   else                              // x is on the right
return binarySearch(arr, x, low, mid - 1) side
```

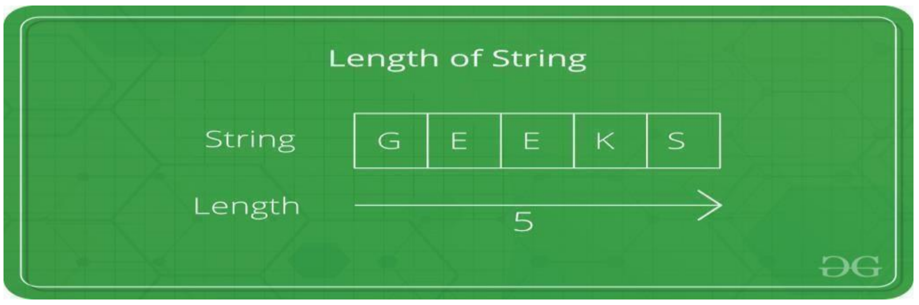
 STRING:

a) INTRODUCTION TO STRINGS:

A string is generally considered as a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. String may also denote more general arrays or other sequence (or list) data types and structures.



b) FINDING LENGTH OF A STRING:



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

<string.h>

int main()
{ char Str[1000]; int
    i;

    printf("Enter the String: "); scanf("%s",
    Str); for (i = 0; Str[i] !=
'\0'; ++i); printf("Length of Str
is %d", i); return 0;
}
```

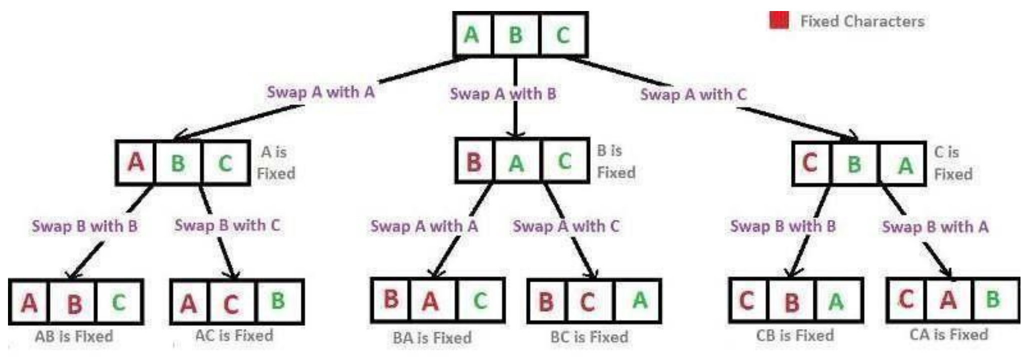
c) REVERSING A STRING:

Different Methods to Reverse a String in C++ are:

- Making our own reverse function ▯
- Using 8inbuilt9 reverse function ▯
- Using Constructor ▯
- Using a *temp* file ▯

```
 #include <bits/stdc++.h>
using namespace std;
 // Function to reverse a string void reverseStr(string&
 str)
 { int n = str.length();
      // Swap character starting from two
      // corners
      for (int i = 0; i < n / 2; i++)
 swap(str[i], str[n - i - 1]); }
 // Driver program int main()
 {    string    str    =    "geeksforgeeks";
 reverseStr(str); cout << str;
 return 0; }
```

d) PERMUTATION OF A STRING:



Recursion Tree for Permutations of String "ABC"

```
#include <bits/stdc++.h>
using namespace std;


// Function to print permutations of string // This
function takes three parameters:
// 1. String
// 2. Starting index of the string //
3. Ending index of the string. void
permute(string a, int l, int r)
```

```
{ // Base case if (l
    == r)
    cout<<a<<endl;
    else
    {
        // Permutations made for (int i
        = l; i <= r; i++)
        {

            // Swapping done swap(a[l],
            a[i]);

      // Recursion called permute(a, l+1,
                         r);
            //backtrack swap(a[l], a[i]);
        }
    }
}
// Driver Code int
main()
{ string str = "ABC"; int n
        =     str.size();
    permute(str, 0, n-1); return
    0;
}
```

MATRICES:

a) INTRODUCTION:

What is Matíix in Data Stíuctuíe?

A matrix represents a collection of numbers arranged in an order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

EXAMPLE:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

b) DIAGONAL MATRIX:

For example, consider the following 4 X 4 input matrix.

```
A00 A01 A02 A03

A10 A11 A12 A13

A20 A21 A22 A23
A30 A31 A32 A33
```

- The primary diagonal is formed by the elements A00, A11, A22, A33. Condition for Principal Diagonal:

```
#include <bits/stdc++.h>
using namespace std;
const int MAX = 100;
//   Function   to   print   the   Principal   Diagonal   void
printPrincipalDiagonal(int mat[][MAX], int n)
{ cout << "Principal Diagonal: "; for
(int i = 0; i < n; i++) { for
    (int j = 0; j < n; j++) {

            // Condition for principal diagonal if
            (i == j) cout << mat[i][j] <<
                ", ";
        } } cout
    << endl;
```

C) SYMMETRIC MATRIX:

A **Simple solution** is to do following.

1. Create transpose of given matrix.
2. Check if transpose and given matrices are same or not

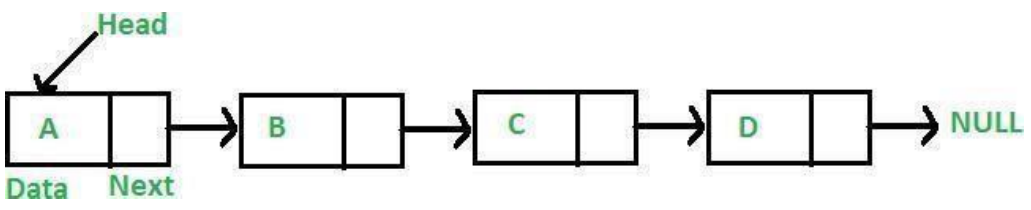
```
#include  <iostream>
using namespace std;
const int MAX = 100;
```

```
// Returns true if mat[N][N] is symmetric, else false bool
isSymmetric(int mat[][MAX], int N)
{ for (int i = 0; i < N; i++) for (int j
    = 0; j < N; j++) if (mat[i][j] !=
    mat[j][i]) return false;
    return true; }

// Driver code int main()
{
    int mat[][MAX] = { { 1, 3, 5 }, {
                       3, 2, 4 }, { 5,
                       4, 1 } };
if (isSymmetric(mat, 3)) cout
    << "Yes";
    else cout <<
        "No";
    return 0;
}
```

LINKED LIST:

a) ABOUT LINKED LIST:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations



b) Display linked list:

Algorithm

1. Create a class Node which has two attributes: data and next. Next is a pointer to the next node.
2. Create another class which has two attributes: head and tail.
3. addNode() will add a new node to the list:
   . Create a new node.

   . It first checks, whether the head is equal to null which means the list is empty.
   . If the list is empty, both head and tail will point to the newly added node.
   . If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.
4. display() will display the nodes present in the list:

. Define a node current which initially points to the head of the list.

   . Traverse through the list till current points to null.
   . Display each node by making current to point to node next to it in each iteration.

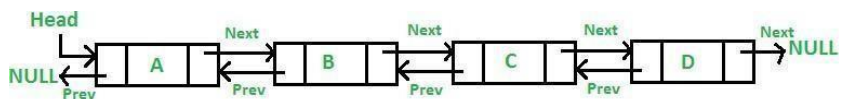
```
#include <stdio.h>
 #include <stdlib.h>
  //Represent a node of singly linked list
  struct node{
  int data;
  struct node *next;
  };

//Represent the head and tail of the singly linked list
struct node *head, *tail = NULL;

  //addNode() will add a new node to the list
  void addNode(int data) {
  //Create a new node
  struct node *newNode = (struct node*)malloc(sizeof(struct node));
  newNode->data = data;
  newNode->next = NULL;
```

```

    //Checks if the list is empty
    if(head == NULL) {
    //If list is empty, both head and tail will point to new node
    head = newNode;
    tail = newNode;
    }
    else {
    //newNode will be added after tail such that tail’s next will point to newN
    ode
    tail->next = newNode;
    //newNode will become new tail of the list
    tail = newNode;
    }
    }

//display() will display all the nodes present in the list
void display() {
//Node current will point to head
struct node *current = head;

    if(head == NULL) {
    printf("List is empty\n");
    return;
    }
    printf("Nodes of singly linked list: \n");
    while(current != NULL) {
    //Prints each node by incrementing pointer
    printf("%d ", current->data);
    current = current->next;
    }
    printf("\n");
    }
```

```
int main()
{
//Add nodes to the list
addNode(1);
addNode(2);
  addNode(3);
  addNode(4);

//Displays the nodes present in the list
display();
```

c) DOUBLY LINKED LIST:

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.



Advantages of DLL over the singly linked list:

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

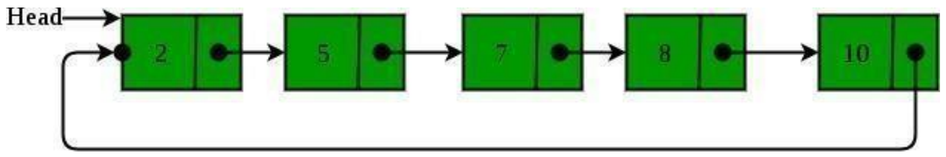
Disadvantages of DLL over the singly linked list:

- Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See this and this).
- All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

```
        / Node of a doubly linked list

class Node { public:
     int data;
     // Pointer to next node in DLL Node*
next;
     // Pointer to previous node in DLL Node* prev;
};
```

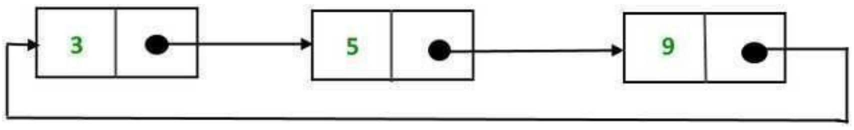
c) CIRCULAR LINKED LIST:

______*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*

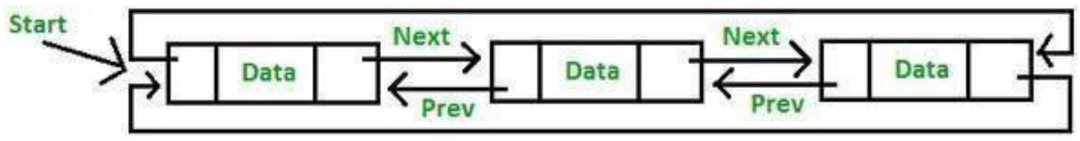


There are generally two types of circular linked lists:

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Representation of circular linked list:

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

```
// Class Node, similar to the linked list class
Node{ int value;

  // Points to the next node.
    Node next; }
```

d) SEARCHING IN A LINKED LIST:

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

Follow the below steps to solve the problem:

- Initialize a node pointer, **current** = **head**.
- Do following while current is not NULL
- If the current value (i.e., **current->key**) is equal to the key being searched return true.
- Otherwise, move to the next node (**current** = **current- >next**).
- If the key is not found, return false

```
#include <bits/stdc++.h>
using namespace std; /*
Link list node */ class
Node { public: int key;
    Node* next; };

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front of
the list. */ void push(Node** head_ref, int new_key)
{
    /* allocate node */ Node* new_node =
    new Node();

    /* put in the key */ new_node->key
    = new_key;
    /* link the old list off the new node */ new_node->next =
    (*head_ref);
    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
/* Checks whether the value x is present in linked list */ bool
search(Node* head, int x)
{
    Node* current = head; // Initialize current while
    (current != NULL) { if (current->key
    == x) return true; current =
        current->next;
    }       return
    false;
}
/* Driver code*/ int main()
{
    /* Start with the empty list */ Node* head
    = NULL;
    int x = 21;
    /* Use push() to construct below list

    14->21->11->30->10 */
    push(&head,      10);
    push(&head,      30);
    push(&head,      11);
    push(&head,      21);
    push(&head, 14);

      // Function call search(head, 21) ? cout <<
    "Yes" : cout << "No"; return 0;
```

e) DELETED FROM LINK LIST:

Delete from a Linked List:-

You can delete an element in a list from:

- Beginning
- End
- Middle

1) Delete from Beginning:

```
Point head to the next node i.e. second node temp
    = head head = head->next


Make sure to free unused memory free(temp); or
    delete temp;
```

2) Delete from End:

```
Point head to the previous element i.e. last second element
    Change next pointer to null struct node *end = head; struct
    node *prev = NULL; while(end->next)
    { prev = end; end = end->next;
    }
    prev->next = NULL;

Make sure to free unused memory free(end); or
    delete end;
```

3) Delete from Middle:

```
Keeps track of pointer before node to delete and pointer to node to
delete temp = head; prev = head; for(int i = 0; i < position; i++)
    { if(i == 0 && position == 1)
        head = head->next; free(temp)
        else
        { if (i == position - 1 && temp)
            {
prev->next = temp->next;

                free(temp);

            }



            else



            {
                 prev =
                temp;
                if(prev == NULL) // position was greater than
number of nodes in the list
                    break;
                temp = temp->next;
            }
        }
    }
```

STACK:

a) INTRODUCTION TO STACK:

Stack

It is a linear data structure that follows a particular order in which the operations are performed.

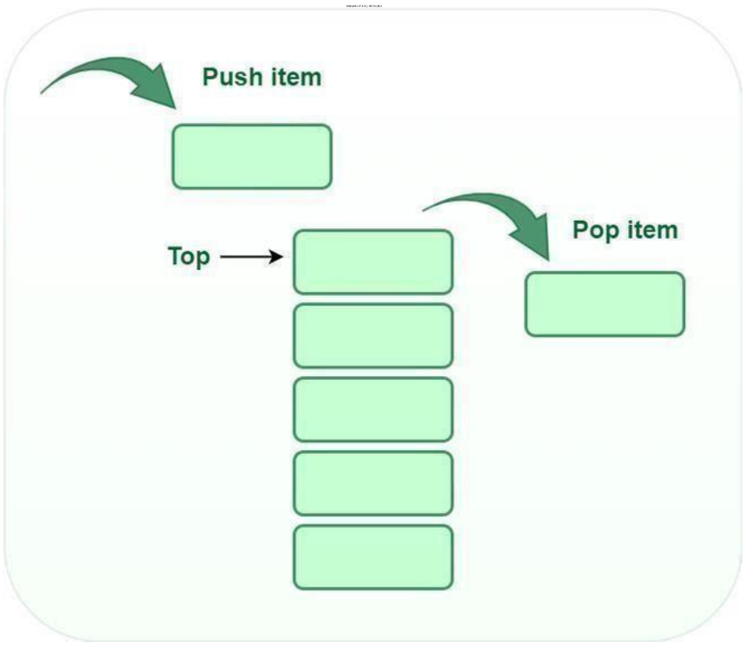
LIFO(Last In First Out):

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

Basic Operations on Stack

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

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



Push:

Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

Algorithm for push:
```
begin
 if stack is full return
 endif else
increment            top
stack[top] assign value
end else end procedure
```

Pop:

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:
```
begin if
 stack  is  empty  return
 endif
else
 store value of stack[top]
 decrement top     return value
```

```
end else end
```

procedure **Top:**

Returns the top element of the stack.

Algorithm for Top: `begin`

```
return   stack[top]   end
```

procedure **isEmpty:**

Returns true if the stack is empty, else false.

Algorithm for isEmpty: `begin`

```
 if top < 1 return true
 else   return false
   end procedure
```

Understanding stack practically:

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow the LIFO/FILO order.

Complexity Analysis:

- ~~Time Complexity~~ **Operations Complexity**

push()	O(1)
pop()	O(1)
isEmpty()	O(1)
size()	O(1)

Types of Stacks:

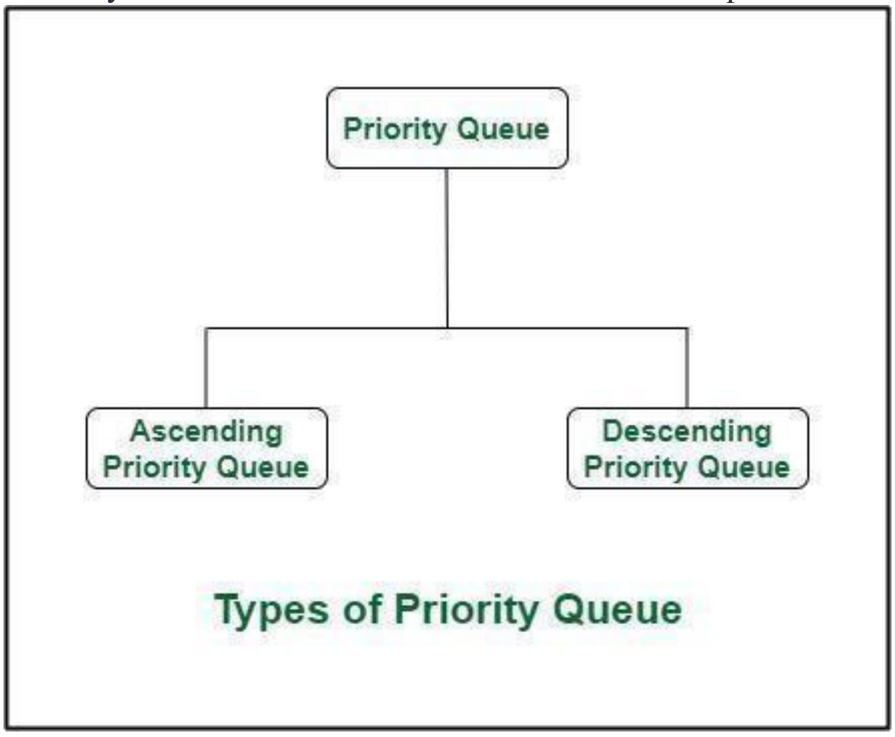
- **Register Stack:** This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
- **Memory Stack:** This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

Applications of the stack:

- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward features in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problems, and histogram problems.
- Backtracking is one of the algorithm designing techniques. Some examples of backtracking are the Knight-Tour problem, N-Queen problem, find your way through a maze, and game-like chess or checkers in all these problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from a current state we need to store the previous state for that purpose we need a stack.
- In Graph Algorithms like Topological Sorting and Strongly

2) Descending order Priority Queue

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.



Types of Priority Queues

Difference between Priority Queue and Normal Queue?

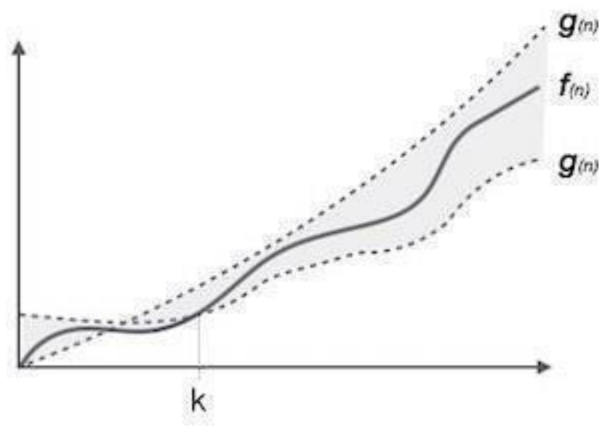
There is no priority attached to elements in a queue, the rule of first-in-first- out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

- Arrays
- Linked list
- Heap data structure
- Binary search tree

TREES:



CONCLUSION:

PROJECT

Project name – *To do application*

Introduction

This project is a simple Task Management System implemented in C++. The program allows users to manage their daily tasks by adding, viewing, marking tasks as completed, deleting tasks, and filtering tasks by category. The program makes use of object-oriented principles, particularly classes, to manage task data efficiently. The key components and features of this project are outlined below:

Key Features

1. Task Class:

- The `Task` class encapsulates the details of a task. Each task has a description, category, due date, priority, and status (completed or pending).
- The constructor initializes a task with a description, category, priority, due date, and sets the task status to "not completed" by default.

2. Task Storage:

- A `vector` named `NotesAdded` is used to store tasks dynamically. This allows easy addition and deletion of tasks as needed.

- A predefined `vector` called `categories` is used to store available categories for tasks such as "Work," "Personal," "Study," etc.

3. Menu and User Interface:

- The program features a simple text-based menu interface that provides users with the following options:

1. Add Task: Allows users to add a new task by providing a description, choosing a category from a predefined list, and setting a priority (HIGH/LOW/MEDIUM).
2. View Task: Displays all the tasks along with their details like description, category, priority, due date, and status.
3. Mark Task as Completed: Allows users to mark a task as completed by entering the task number.
4. Delete Task: Allows users to delete a specific task by entering the task number.
5. Filter Task By Category: Although the function is defined, this feature is currently not implemented. This would ideally allow users to view tasks filtered by a specific category.
6. Exit: Exits the program.

4. Task Management Functions:

- `addTask()`: Prompts the user for task details and adds a new task to the list.
- `displayTasks()`: Displays the list of tasks along with their details.
- `markTaskAsCompleted()`: Marks a specified task as completed.
- `deleteTask()`: Deletes a specified task.
- `filterTaskByCategory()`: Intended to filter tasks by category (yet to be implemented).

#codes

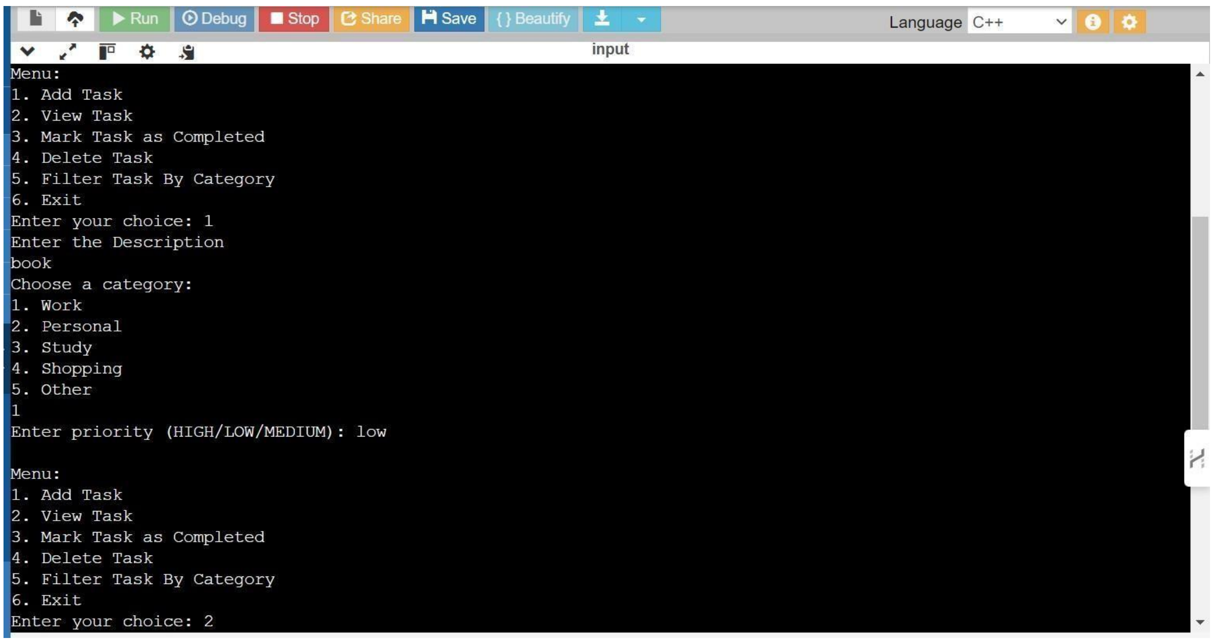
```
#to do application
#include
#include
#include
#include <bits/stdc++.h>
#include using
namespace std; class
Task{ public:
string description;
string category; bool
completed; string
dueDate; // TAKEN
FROM SYSTEM
TODAY(); string
priority; // LOW
MEDIUM
Task(string description , string category,string priority,string
dueDate){ this->description = description; this->category = category;
this->dueDate = dueDate; this->priority = priority; this->completed =
false;
} };
vector NotesAdded; void displayTasks(){
if(NotesAdded.empty()){ cout<<"Please add some
task first to View";
}
else{ cout<<"\nTasks:\n"; cout<<" "<<"Task"<<"
"<<"Category"<<" "<<"Priority"<<"
"<<"DueDate"<<" "<< "Status"<<"\n";
for(int i=0;i<NotesAdded.size();i++){
cout<<i+1<<". "
<<NotesAdded[i].description<<" "
<<NotesAdded[i].category<<" "
<<NotesAdded[i].priority<<" "
<<NotesAdded[i].dueDate<<" "
<<(NotesAdded[i].completed ? "Completed" : "Pending")<<"\n";
}
}
}
```

```
void markTaskAsCompleted() { int taskNumber; cin>>taskNumber;
if(taskNumber>=0 && taskNumber<=NotesAdded.size()){
NotesAdded[taskNumber - 1].completed = 1;
}else{ cout<<"Invalid Task
Number"<<"\n";
} };
void deleteTask(){ //index of
that task // if that task is not
present show him that put
the task first int
taskNumber;
cin>>taskNumber;
if(taskNumber>=0 && taskNumber<=NotesAdded.size()){
NotesAdded.erase(NotesAdded.begin()+taskNumber-1);
}else{ cout<<"Invalid Task
Number"<<"\n";
}
}
void filterTaskByCategory(){
} // Map to store the leaderboard (player name and score) // Function to play
the game void addtask() { string description; string category; string priority;
cout<<"Enter the Description\n"; cin.ignore(); getline(cin,description);
cout<<"Enter the Category you want\n"; getline(cin,category); cout<<"Enter
Task Priority\n"; getline(cin,priority); auto now = time(nullptr); tm* current =
localtime(&now); char buffer[80]; strftime(buffer,sizeof(buffer),"%Y-%m-
%d",current); string dueDate(buffer);
Task task(description,category,priority,dueDate);
NotesAdded.push_back(task);
}
// Function to display the menu and handle user choices void
displayMenu() { int choice; // Variable to store the user's menu
choice do { cout << "\nMenu:\n" << "1. Add Task\n"
<< "2. View Task\n"
<< "3. Mark Task as Completed\n"
<< "4. Delete Task\n"
<< "5. Filter Task By Category\n"
<< "6. Exit\n"
<< "Enter your choice: "; cin >> choice; //
Read the user's choice switch (choice) {
case 1: addtask(); break; case 2:
displayTasks(); break; case 3:
markTaskAsCompleted();
break; case 4:
deleteTask(); case 5:
filterTaskByCategory();
break; case 6:
cout<<"Goodbye !Please do the tasks on time";
}
} while (choice != 6); // Loop until the user chooses to exit
}
int main() { displayMenu(); // Display the menu to the
user return 0; // End of the program
}
```

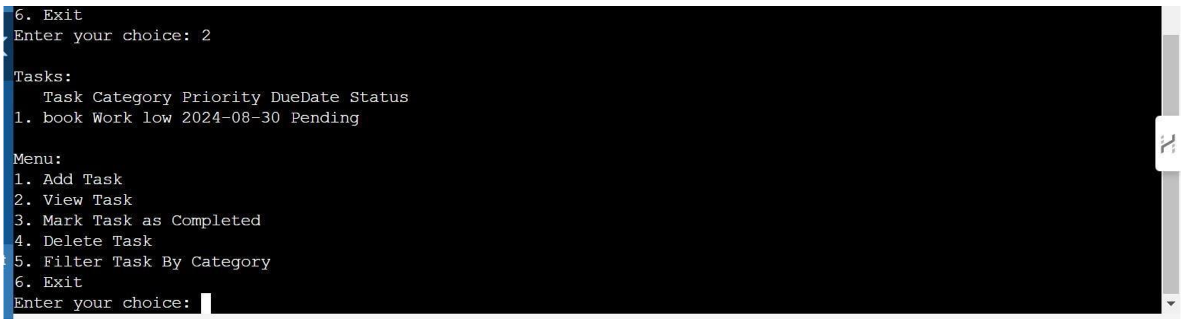
Conclusion:

The Task Management System project demonstrates fundamental concepts of object-oriented programming in C++, such as encapsulation, dynamic memory management using vectors, and the implementation of basic CRUD (Create, Read, Update, Delete) operations. The program is designed with simplicity and usability in mind, providing users with essential task management functionalities like adding, viewing, marking, and deleting tasks.

Photo



```
Menu:
1. Add Task
2. View Task
3. Mark Task as Completed
4. Delete Task
5. Filter Task By Category
6. Exit
Enter your choice: 1
Enter the Description
book
Choose a category:
1. Work
2. Personal
3. Study
4. Shopping
5. Other
1
Enter priority (HIGH/LOW/MEDIUM): low

Menu:
1. Add Task
2. View Task
3. Mark Task as Completed
4. Delete Task
5. Filter Task By Category
6. Exit
Enter your choice: 2
```



```
6. Exit
Enter your choice: 2

Tasks:
   Task Category Priority DueDate Status
1. book Work low 2024-08-30 Pending

Menu:
1. Add Task
2. View Task
3. Mark Task as Completed
4. Delete Task
5. Filter Task By Category
6. Exit
Enter your choice:
```

Learning Outcomes

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are used to solve the problem using that data.

Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use each one of them. It also teaches you the science of evaluating the efficiency of an algorithm. This enables you to choose the best of various choices.

For example, you want to search your roll number in 30000 pages of documents, for that you have choices like Linear search, Binary search, etc. So, the more efficient way will be Binary search for searching something in a huge number of data.

So, if you know the DSA, you can solve any problem efficiently. The main use of DSA is to make your code scalable because

• Time is precious

• Memory is expensive