SUMMER OF SCIENCE 2021

MIDTERM REPORT

"Bad programmers worry about the code. Good programmers worry about data structures and their Relationships."

~Linus Torvalds

Project By:

Shubham Namdev Kamble

180020105

Mentor: Vibhor Jain

Introduction

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. From the data structure point of view, following are some important categories of algorithms

- ☐ Search − Algorithm to search an item in a data structure.
- ☐ Sort − Algorithm to sort items in a certain order.
- ☐ Insert − Algorithm to insert item in a data structure.
- ☐ Update − Algorithm to update an existing item in a data structure.
- ☐ Delete − Algorithm to delete an existing item from a data structure.

Arrays: It is a list of variables of similar type. It is a Data Structure used to store information in contiguous memory allocation. This data can be integers, strings, characters, class objects etc.

Implementation:

Eg. Find minimum and maximum number from array

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

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];}
    int max_elem=INT_MIN;
    int min_elem=INT_MAX;
    }

    for(auto x:arr){
        max_elem=max(max_elem,x);
        min_elem=min(min_elem,x);
    }
    cout<<"Max Elem: "<<max_elem<<endl;
```

```
    cout<<"Min Elem: "<<min_elem<<endl;
    return 0;
}
```

Searching Element in Array:

Linear Search: Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list.

```
int LinearSearch(int arr[], int n, int key){
    for(int i=0;i<n;i++){
        if(arr[i]==key){
            return i;
        }
    }
    return -1;
}
```

Binary Search: Binary search is a search that finds the middle element in the list recursively until the middle element is matched with a searched element.

```
int BinarySearch(int arr[], int n, int key){
    int s=0,e=n;
    while(s<=e){
        int mid=(s+e)/2;

        if(arr[mid]==key){
            return mid;
        }
        else if(arr[mid]>key){
            e=mid-1;
        }
        else{
            s=mid+1;
        }
    }
    return -1;
}
```

Sorting :

Placing elements in ascending order

Selection Sort: Find the minimum element in the unsorted array and swap it with element at the beginning of array

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

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }

    for(int i=0;i<n-1;i++){
        for(int j=i+1;j<n;j++){
            if(arr[j]<arr[i]){
                int temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
    }
    for(int i=0;i<n;i++){
        cout<<arr[i]<<endl;
    }
    return 0;
}
```

Bubble Sort : Repeatedly swap two adjacent elements if they are in wrong order

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

```
int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    int counter=0;
    while(counter< n-1){
        for(int i=0;i<n-1-counter;i++){
            if(arr[i]>arr[i+1]){
                int temp=arr[i];
                arr[i]=arr[i+1];
                arr[i+1]=temp;
            }
        }
        counter++;
    }
    for(int i=0;i<n;i++){
        cout<<arr[i]<<endl;
    }
    return 0;
}
```

Insertion Sort : Insert an element from unsorted array to its correct position in sorted array

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

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }

    for(int i=1;i<n;i++){
        int current=arr[i];
        int j=i-1;
        while(arr[j]<arr[j+1] && j>=0){
            arr[j+1]=arr[j];
```

```
            j--;
        }
        arr[j+1]=current;
    }
    for(int i=0;i<n;i++){
        cout<<arr[i]<<endl;
    }
    return 0;
}
```

Subarrays and Subsequences:

1. **Subarray** : Continuous part of array

   Number of subarrays of an array with n elements = $\frac{n(n+1)}{2}$

2. **Subsequence** : sequence that can be derived an array selecting zero or more elements, without changing the order of the remaining elements

   Number of subsequences of an array with n elements = 2^n

Every subarray is subsequence but every subsequence is not a subarray

Stacks:

Stacks are dynamic data structures that follow the Last In First Out (LIFO) principle. The last item to be inserted into a stack is the first one to be deleted from it.

Features of stacksDynamic data structures

- ☐ Do not have a fixed size
- ☐ Do not consume a fixed amount of memory
- ☐ Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.

Different Operations used in Stack and their Implementations:

1. push(x): Insert element x at the top of a stack

```
void push (int stack[ ] , int x , int n) {
 if ( top == n-1 ) {         //If the top position is the last of position in a
stack, this means that the stack is full
    cout << "Stack is full.Overflow condition!" ;
    }
    else{
        top = top +1 ;            //Incrementing top position
        stack[ top ] = x ;       //Inserting element on incremented position
    }
}
```

2. pop( ): Removes an element from the top of a stack

```
void pop (int stack[ ] ,int n )
    {

        if( isEmpty ( ) )
        {
            cout << "Stack is empty. Underflow condition! " << endl ;
        }
        else
        {
             top = top - 1 ; //Decrementing top's position will detach last
element from stack
        }
    }
```

Queues

Queues are data structures that follow the First In First Out (FIFO) i.e. the first element that is added to the queue is the first one to be removed. Elements are always added to the back and removed from the front.

Different Operations used in Queues and their Implementations:

Enque : If the queue is not full, this function adds an element to the back of the queue, else it prints “OverFlow”.

```
void enqueue(int queue[], int element, int& rear, int arraySize) {
    if(rear == arraySize)            // Queue is full
```

```
            printf("OverFlow\n");
    else{
         queue[rear] = element;    // Add the element to the back
         rear++;
    }
}
```

Deque : If the queue is not empty, this function removes the element from the front of the queue, else it prints “UnderFlow”

```
void dequeue(int queue[], int& front, int rear) {
    if(front == rear)            // Queue is empty
        printf("UnderFlow\n");
    else {
        queue[front] = 0;        // Delete the front element
        front++;
    }
}
```

Linked Lists : A linked list is a way to store a collection of elements. Like an array these can be characters or integers. Each element in a linked list is stored in the form of a node.

A node is a collection of two sub-elements or parts. A data part that stores the element and a next part that stores the link to the next node.

Implementation:

```
#include <iostream>
using namespace std;

struct node
{
    int data;
    node *next;
};

class linked_list
{
private:
    node *head,*tail;
public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }

    void add_node(int n)
    {
        node *tmp = new node;
        tmp->data = n;
        tmp->next = NULL;

        if(head == NULL)
        {
            head = tmp;
            tail = tmp;
        }
        else
        {
            tail->next = tmp;
            tail = tail->next;
        }
```

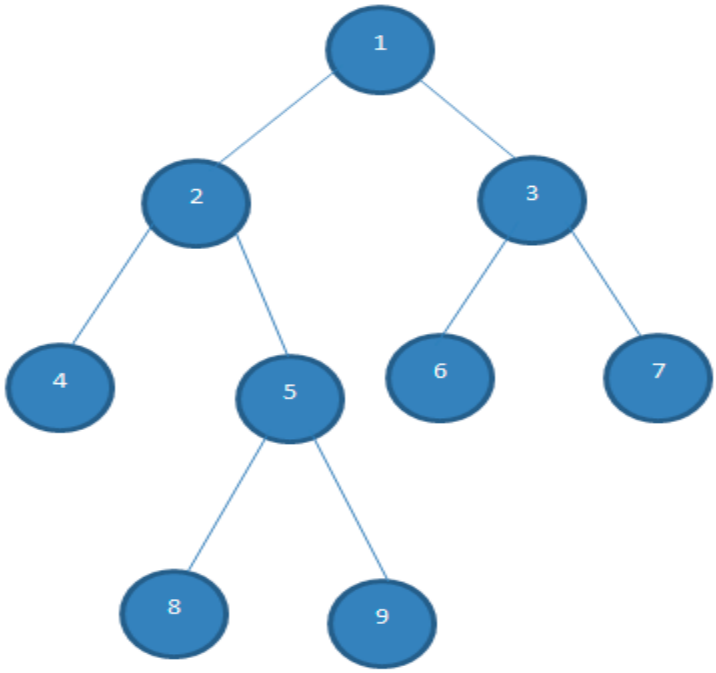
```
    }
};

int main()
{
    linked_list a;
    a.add_node(1);
    a.add_node(2);
    return 0;
}
```

Binary Trees:

A binary tree is a structure comprising nodes, where each node has the following 3 components:

1. Data element: Stores any kind of data in the node
2. Left pointer: Points to the tree on the left side of node
3. Right pointer: Points to the tree on the right side of the node



As the name suggests, the data element stores any kind of data in the node.The left and right pointers point to binary trees on the left and right side of the node respectively.If a tree is empty, it is represented by a null pointer.

Operations on Binary Tree

1. Pre order Traversal
2. Searching a key in the Tree
3. Removing or deleting a node from the tree

Revised Plan of Action

Timeline	Topics
Week 5	**Heaps, Hashing, Binary Trees**
Week 6	**Graphs**
Week 7	**Remaining Graphs + Dynamic Programming**
Week 8	**Dynamic Programming**