WnCC Summer of Code 2022

Intro To Competitive Programming Progress

Name: Rohan Rajesh Kalbag
Undergraduate Sophomore
B.Tech. Electrical Engineering

Mentor: Shivang Tiwari

May - July 2022

This report was created and is being maintained for **MnP Summer of Science 2022** for the topic **Data Structures and Algorithms**. I am mentioning it here as well due to the nature of its interdependence with **Introduction To Competitive Programming** as knowledge of DSA supplements CP

Contents

1 Preliminaries and STL Libraries of C++

In the first week, I went through elements of the STL libraries of C++. I was introduced to useful classes like `vector`, `string`, `map`, `set`. I was also introduced to the concept of `iterators` in C++.

1.1 Vector

Link to Vector Class Documentation
The vector is a dynamic array in C++. It allows indexing operations like the normal array in C++. It additionally allows dynamic resizing, unlike C++ arrays. So from now on we can use `vector` instead of regular array due to the above advantages.

1.2 String

Link to String Class Documentation
The string class allows dynamic insertion of `char` type inputs using `push_back()`. It also allows concatenation using the + operator and also indexing like in arrays. Also easily an entire line of characters from the input can be taken into a string using `getline(cin, s)`.

1.3 Set

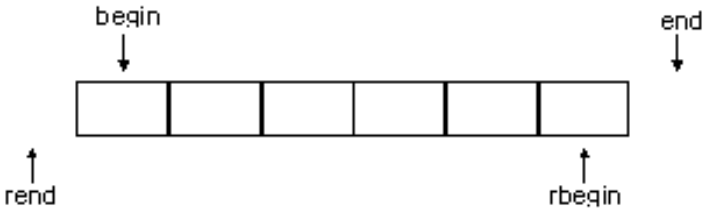
Link to Set Class Documentation
The set class is a very useful class of the STL library. It is implemented internally using a **Binary Search Tree**. It ignores duplicate elements and can be used when there is an application of only getting the distinct values. It also holds the values in ascending order so the smallest and largest element can be easily accessed using iterators. The minimum value is given by `*s.begin()` and the maximum value is given by `*prev(s.end())`. Also operations like insertion and deletion takes O($log(n)$). It cannot be indexed unlike in vectors.

1.4 Map

Link to Map Documentation
The map class is a very useful class of the STL library. It is implemented internally using a **Binary Search Tree** like in Set however each element is a **pair** (has a key : value relationship) like the `dictionary` in `python`. It ignores duplicate keys and can be used when there is an application of distinct keys. It also holds the keys in ascending order so the smallest and largest element can be easily accessed using iterators. Its applications include linking the frequency of occurence of characters in a string with the characters in string problems. It can also be used to link the index of an object with its value among other applications. It can also be used as a hash table and also as look up tables. If `it` is an iterator of the Map, we can access the key and value in the following way. key : `it->first` and value : `it->second`.

1.5 Iterators

Iterators are like pointers to elements of STL class. For example, if we have a `vector`. It can be iterated in the following ways. Suppose a location is referenced by the iterator `it`, then just like dereferencing in pointers we can access the value by `*it`. Also iterators allow simple **reverse traversal** of STL classes without usage of structures like `stacks`. Iterators are very useful when indexing is not possible such as in `set` and `map` classes.



Important Iterators for Standardized STL Container: Link to Image

1.5.1 Examples of Using Iterators

```
//suppose we have a vector
vector<int> a = {1,2,7,5,5,6};
map<int,int> freq;

//forward traversal gives 1 2 7 5 5 6
for(auto it = a.begin();it!=a.end();it++){
    cout<<*it<<endl;
    s[*it]++; //count the frequency of each number
}

//reverse traversal gives 6 5 5 7 2 1
for(auto it = a.rbegin();it!=a.rend();it++){
    cout<<*it<<endl;
}

//traversal of map
for(auto it = freq.rbegin();it!=freq.rend();it++){
    //print keys and values side by side
    cout<<it->first<<"-"<<it->second<<endl;
}

//Expected Output:
/*
        1-1
        2-1
        5-2
        6-1
        7-1
/*
```

2 Complexity and Competitive Programming

In the second week, I was introduced to the notion of Time and Space Complexity and the **Big-Oh** notation to indicate worst case complexity. I was also introduced to practices followed in Competitive Programming

2.1 Time and Space Complexity

We can estimate the complexity constraints of a CP problem using **Big-Oh** notation. It is a function of the input size n.

2.1.1 Important Properties

$$O(cn) = O(n), c > 0 \tag{1}$$

$$O(n^n + n^m) = O(n^m), m > n, m, n > 1 \tag{2}$$

$$O(c + n) = O(n), c > 0 \tag{3}$$

For example, the following block has a time complexity of $O(n^2)$

```
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        cout<<a[i][j]<<endl;
    }
}
```

Also the following block has a space complexity of $O(n)$ as we use additional memory of size linearly related to the input

```
vector<int> arr; //suppose an array of size n
set<int> s;
for(int i=0;i<n;i++){
    s.insert(arr[i]);
}
```

2.2 Estimating the Time Complexity of a CP problem

Suppose there are t testcases and and an array of size n is given and constraints are given $0 \leq t \leq 10^3$ and $0 \leq n \leq 2 \cdot 10^5$ for a particular program. We know that a language such as C++ can perform around 10^9 operations in 1 sec. Also supposing the time limit for the problem is 1 sec.

Thus we can atmost afford an algorithm of time complexity $O(n)$ as $t * n$ is of the order of 10^8 operations. If we had an algorithm of $O(n^2)$, then the number of operations per second would be the order of $t * n^2$ which is 10^{13}, leading to a **Time Limit Exceeded** verdict.

Thus we can estimate the expected time complexity for any Competitive Programming problem in the following way.

Sometimes another indication is given in problems such as `"The sum of n across all testcases` $\leq 10^5$`"`. This means $t * n$ is of the order of 10^5 operations and not of the order of 10^8.

2.3 Creation of a Generalized C++ Code Template

To simplify code writing and reduce time taken we make ourselves a template for Competitive Programming using `typedef` and `#define` commands in C++. For example, this is the template I created for myself.

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

//@author: Rohan Rajesh Kalbag

//type definitions
typedef long long int ll;
typedef unsigned long long int ull;
typedef vector<int> vi;
typedef set<int> si;
typedef map<int,int> mii;
typedef vector<long long int> vll;
typedef priority_queue<int> pqi;

//definitions
#define pb push_back
#define sz size()
#define ff first
#define ss second
#define pi 3.14159265359
#define endl '\n'

void testcase(){
    // for each testcase some code goes here
    int n;
    cin>>n;
    vector<int> a(n);
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);
    int t;
    cin>>t;
    cin.ignore();
    while(t--){
        testcase();
    }
}
```

2.4 Creation of Accounts in Leetcode and Codeforces

To practice problems related to Data Structures and Algorithms. I was suggested to create accounts in Leetcode and was adviced to participate in rated contest of Codeforces to improve exposure to various kinds of CP problems. I started participating in all Div 2, Div 3 contests of Codeforces

3 Arrays, Strings and Data Structures

In Week 3, I started practicing problems of easy and medium difficulty from the topic Arrays and Strings in **GeeksForGeeks** and **Leetcode** as instructed. Also at the same time I started covering the various types of Data Structures using resources online such as the videos of **William Fiset**.

I also created a GitHub Repository where I uploaded the C++ code of my attempts at implementing the various Data Structures encountered by me. The problems solved in Arrays and String can be found here: Problems Solved. The rated contests participated in till now can be found here: Contests.

3.1 Data Structures Covered Till Now

3.1.1 Linked List

The Linked List (Singly and Doubly Linked) are used in the implementation of Queues, Stacks and Abstact Data Types like Circular List. They are also used in Hashtables to prevent Hashing collisions.

Important Terminology for Linked List: Head (The First Node), Tail (The Last Node), Pointer to Next (Present in both SLL and DLL), Pointer to Prev (Present in DLL and not in SLL).

C++ Code of SLL Implementation can be found here
C++ Code of DLL Implementation can be found here

Operation	Singly LL	Doubly LL
Search	O(n)	O(n)
Insert at Head	O(1)	O(1)
Insert at Tail	O(1)	O(1)
Remove at Head	O(1)	O(1)
Remove at Tail	O(n)	O(1)
Remove in Middle	O(n)	O(n)

Time Complexity of Operations in Linked List

3.1.2 Stack

The stack is used in browsers, editors, games to hold the previous instruction in case of undo. It is also used in Depth First Search in Graphs. Tower of Hanoi is solved using Stack. It follows **Last In First Out** terminology.

Important Terminology for Stack: Top (pointer to topmost element in stack)

C++ Code of Stack Implementation can be found here

Operation	Complexity
Search	O(n)
Push	O(1)
Pop	O(1)
Size	O(1)
Peek	O(1)

Time Complexity of Operations in Stack

3.1.3 Queue

To keep track of x most recently added elements. It is also used in Breadth First Search in Graph Traversal. It models real life queues. It follows **First In First Out** terminology.

Important Terminology for Queue: Front (pointer to the first element), End (pointer to the last element)

C++ Code of Queue Implementation can be found here

Operation	Complexity
Contains	O(n)
Removal	O(n)
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)

Time Complexity of Operations in Queue

3.1.4 Heap

A Heap is a Tree based Data Structure that satisfies the **Heap Invariant**.

Remark. *Heap Invariant: If A is the parent node of B in the heap, then* $value(A) \lesseqgtr value(B)$ *$\forall$ nodes A, B. If $\geq$ then it is a maxheap and if $\leq$ then it is a minheap.*

Remark. *A max heap can be converted into a min heap by using the negate comparator (negated comparison operation). Or elements can be negated while insertion and and negated back after removal from the heap.*

A **Binary Heap** is a heap where each node can have atmost two children. It is used to implement Priority Queues

3.1.5 Priority Queue

It is similar to a Queue but each element has a certain priority associated. The priority determines the way in which elements are removed from the PQ. They can only hold comparable datatypes. It follows **First In Most Prioritized Out** terminology. Priority Queues are used in Best First Search in Graph Theory. It is also used for the Minimum Spanning Tree Algorithm.

Important Terminology for Priority Queue: poll - remove the highest priority element, add - insert an element.

C++ Code of Priority Queue Implementation can be found here

Operation	Complexity
Binary Heap Construction	O(n)
Polling	O(log(n))
Peeking	O(1)
Adding	O(log(n))
Naive Removal	O(n)
Removal with Hashing	O(log(n))
Naive Contains	O(n)
Contains with Hashing	O(1)

Time Complexity of Operations in Priority Queue

3.1.6 Union Find

It keeps track of elements which are split into one or more disjoint sets. It is used in Kruskal's Minimum Spanning Tree Algorithm in Graph Theory

Important Terminology for Union Find: union - merges two groups, find - given an element, tells which group it belongs to

C++ Code of Union Find Implementation can be found here

Operation	Complexity
Construction	O(n)
Union	$\alpha(n)$
Find	$\alpha(n)$
Get Component Size	$\alpha(n)$
isConnected	$\alpha(n)$
No of Components	O(1)

$\alpha(n)$: Amortized Constant Time (Almost Constant Time)
Time Complexity of Operations in Union Find

Remark. *We have Amortized Constant Time because of Path Compression. Everytime we get a reference to the root node of a group such as in a* `find()` *call we set the parent of each node in that group to be the root node.*

3.1.7 Trees

A tree is an undirected graph which is an acyclic connected graph (or) A connected graph with N nodes and N-1 vertices (or) A graph in which two vertices are connected by exactly one path

Important Terminology for Tree: root: start node of the tree, child: a node extending from another node which will be called its parent, leaf node: a node with no children

A **Binary Tree** is a tree where each node can have atmost two children.

3.1.8 Binary Search Tree

A Binary Search Tree is a binary tree which satisfies the **BST Invariant**.

Remark. *BST Invariant: The left subtree of a node has smaller elements than it and the right subtree has larger elements*

Binary Search Trees are used for creation of STL classes like **Map** and **Set** that were encountered in Week 1.

C++ Code of Binary Search Tree Implementation can be found here

Operation	Average	Worst Case
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Remove Node	O(log(n))	O(n)
Search	O(log(n))	O(n)

Time Complexity of Operations in Binary Search Tree

3.1.9 Recursive Binary Tree Traversals

Preorder, Postorder and Inorder Traversal are recursively defined.

```
//node has left and right node* pointing to the child nodes

//preorder: print, then left and right
void preorder(node* n){
    if(node==nullptr)
        return;
    print(n->val);
    preorder(n->left);
    preorder(n->right);
}
```

```
//inorder: left, then print and right
void inorder(node* n){
    if(node==nullptr)
        return;
    inorder(n->left);
    print(n->val);
    inorder(n->right);
}


//postorder: left, then right and print
void postorder(node* n){
    if(node==nullptr)
        return;
    postorder(n->left);
    postorder(n->right);
    print(n->val);
}
```

3.1.10 Level Order Traversal

We maintain a queue and add the nodes followed by their left and right children into the queue and follow this until the queue is empty. This is a level by level traversal of the Binary Tree. This algorithm is a Breadth First Search extended to a Tree from Graph Theory.

4 References

- https://cses.fi/book.pdf
- https://www.youtube.com/playlist?list=PLDV1Zeh2NRsB6SWUrDFW2RmDotAfPbeHu