Name: Ayush Garg

Employee ID: TEN/BGD/0003

Batch: June 1

CORE TASK 1

Table of Contents

Chapter - 1

Introduction To Data Structure

Data Structure can be described as the group of records elements that presents an efficient way of storing and organizing statistics in the computer so that they can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are broadly used in nearly every factor of Computer Science i.e., Operating systems, Compiler Design, Artificial intelligence, Graphics, and many more.

Data structures are the main part of many computer technology algorithms as they permit the programmers to address the facts efficiently. It plays an important position in enhancing the performance of the software or a program as the main characteristic of the software program is to keep and retrieve the person's information as rapidly as possible.

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C++ language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.

Types of Data Structures

There are two types of data structures:

1. Primitive data structure: The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.
2. Non-primitive data structure

The non-primitive data structure is divided into two types:

1. Linear data structure
2. Non-linear data structure

Data structures can also be classified as:

Static data structure: It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

Dynamic data structure: It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Advantages of Data structures

The following are the advantages of a data structure:

- Efficiency: If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- Reusability: The data structure provides reusability means that multiple client programs can use the data structure.
- Abstraction: The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Chapter - 2

Arrays

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). The base value is index 0 and the difference between the two indexes is the offset.

Size

In C language, the array has a fixed size meaning once the size is given to it, it cannot be changed i.e., you can't shrink it nor can you expand it. The reason was that for expanding if we change the size, we can't be sure that we get the next memory location to us for free. The shrinking will not work because the array, when declared, gets memory statically allocated, and thus compiler is the only one that can destroy it.

Types of indexing in an array:

- 0 (zero-based indexing): The first element of the array is indexed by a subscript of 0.
- 1 (one-based indexing): The first element of the array is indexed by the subscript of 1.
- N (N-based indexing): The base index of an array can be freely chosen. Usually, programming languages allowing n-based indexing also allow negative index values, and other scalar data types like enumerations, or characters may be used as an array index.

Disadvantages

You can't change the size i.e., once you have declared the array you can't change its size because of static memory allocation. Here Insertion(s) and deletion(s) are difficult as the elements are stored in consecutive memory locations and the shifting operation is costly too.

Applications on Array

- Array stores data elements of the same data type.
- Arrays are used when the size of the data set is known.
- Used in solving matrices problems.
- Applied as a lookup table in computer.
- Databases records are also implemented by the array.
- Helps in implementing sorting algorithm.
- Different variables of the same type can be saved under one name.
- Arrays can be used for CPU scheduling.
- Used to Implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.

Chapter - 3

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list. Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1. The size of the arrays is fixed: So, we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create the room existing elements have to be shifted but in the Linked list if we have the head node then we can traverse to any node through it and insert a new node at the required position.

The various types of linked lists are as follows:

1. Singly Linked List: It is the most basic linked list in which traversal is unidirectional i.e. from the head node to the last node.
2. Doubly Linked List: In this linked list, traversal can be done in both ways, and hence it requires an extra pointer.
3. Circular Linked List: This linked list is unidirectional but, in this list, the last node points to the first i.e., the head node and hence it becomes circular.
4. Circular Doubly Linked List: The circular doubly linked list is a combination of the doubly linked list and the circular linked list. It means that this linked list is bidirectional and contains two pointers and the last pointer points to the first pointer.

Applications

- Linked Lists are used to implement stacks and queues.
- It is used for the various representations of trees and graphs.
- It is used in dynamic memory allocation (linked list of free blocks).
- It is used for representing sparse matrices.
- It is used for the manipulation of polynomials.
- It is also used for performing arithmetic operations on long integers.

Applications in the real world

- The list of songs in the music player is linked to the previous and next songs.
- In a web browser, previous and next web page URLs are linked through the previous and next buttons.
- In the image viewer, the previous and next images are linked with the help of the previous and next buttons.

Advantages

- Insertion and deletion in linked lists are very efficient.
- Linked list can be expanded in constant time.
- For the implementation of stacks and queues and for representation of trees and graphs.
- Linked lists are used for dynamic memory allocation which means effective memory utilization hence, no memory wastage.

Disadvantages

- Use of pointers is more in linked lists hence, complex and requires more memory.
- Searching an element is costly and requires O(n) time complexity.
- Traversing is more time-consuming and reverse traversing is not possible in singly-linked lists.
- Random access is not possible due to dynamic memory allocation.

Drawbacks

1. Random access is not allowed. We have to access elements sequentially starting from the first node (head node). So, we cannot do a binary search with linked lists efficiently with its default implementation. Read about it here.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache-friendly. Since array elements are contiguous locations, there is a locality of reference which is not there in the case of linked lists.

Representation

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

1. data (we can store integers, strings, or any type of data).
2. Pointer (Or Reference) to the next node (connects one node to another)

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

In Java or C++, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

Advantages over arrays

1. Dynamic size
2. Ease of insertion/deletion

Implementation

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

class Node {
public:
	int data;
```

```
		Node* next;
};

int main ()
{
		Node* head = NULL;
		Node* second = NULL;
		Node* third = NULL;

		head = new Node ();
		second = new Node ();
		third = new Node ();

		head->data = 1;
		head->next = second;
		second->data = 2;
		second->next = third;
		third->data = 3;
		third->next = NULL;
		return 0;
}
```

Insertion of Node

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

class node {
public:
		int data;
		node* next;
		node (int value)
		{
				data = value;
				next = NULL;
		}
};

void insertathead(node*& head, int val)
{
		node* n = new node(val);
		n->next = head;
		head = n;
}

void insertafter(node* head, int key, int val)
{
		node* n = new node(val);
		if (key == head->data) {
				n->next = head->next;
				head->next = n;
				return;
		}

		node* temp = head;
		while (temp->data! = key) {
				temp = temp->next;
				if (temp == NULL) {
						return;
				}
		}
		n->next = temp->next;
		temp->next = n;
}
```

```
void insertattail(node*& head, int val)
{
	node* n = new node(val);
	if (head == NULL) {
		head = n;
		return;
	}

	node* temp = head;
	while (temp->next! = NULL) {
		temp = temp->next;
	}
	temp->next = n;
}

void print (node*& head)
{
	node* temp = head;

	while (temp! = NULL) {
		cout << temp->data << " -> ";
		temp = temp->next;
	}
	cout << "NULL" << endl;
}

int main ()
{
	node* head = NULL;
	insertathead(head, 1);
	insertathead(head, 2);
	cout << "After insertion at head: ";
	print(head);
	cout << endl;

	insertattail(head, 4);
	insertattail(head, 5);
	cout << "After insertion at tail: ";
	print(head);
	cout << endl;

	insertafter(head, 1, 2);
	insertafter(head, 5, 6);
	cout << "After insertion at a given position: ";
	print(head);
	cout << endl;
	return 0;
}
```

Deletion of Node

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

struct node {
	int info;
	node* link = NULL;
	node () {}
	node (int a)
		: info(a)
	{
	}
};
```

```
void deleteNode(node*& head, int val)
{
        if (head == NULL) {
                cout << "Element not present in the list\n";
                return;
        }
        if (head->info == val) {
                node* t = head;
                head = head->link;
                delete (t);
                return;
        }
        deleteNode(head->link, val);
}

void push (node*& head, int data)
{
        node* newNode = new node(data);
        newNode->link = head;
        head = newNode;
}

void print (node* head)
{
        if (head == NULL and cout << endl)
                return;
        cout << head->info << ' ';
        print(head->link);
}

int main ()
{
        node* head = NULL;
        push (head, 10);
        push (head, 12);
        push (head, 14);
        push (head, 15);
        print(head);
        deleteNode(head, 20);
        print(head);
        deleteNode(head, 10);
        print(head);
        deleteNode(head, 14);
        print(head);
        return 0;
}
```

Reverse a Linked List (Iterative)

```
#include <iostream>
using namespace std;
struct Node {
        int data;
        struct Node* next;
        Node (int data)
        {
                this->data = data;
                next = NULL;
        }
};

struct LinkedList {
        Node* head;
```

```
	LinkedList () { head = NULL; }
	void reverse ()
	{
		Node* current = head;
		Node *prev = NULL, *next = NULL;

		while (current != NULL) {
			next = current->next;
			current->next = prev;
			prev = current;
			current = next;
		}
		head = prev;
	}

	void print ()
	{
		struct Node* temp = head;
		while (temp != NULL) {
			cout << temp->data << " ";
			temp = temp->next;
		}
	}

	void push (int data)
	{
		Node* temp = new Node(data);
		temp->next = head;
		head = temp;
	}
};

int main ()
{
	LinkedList ll;
	ll.push(20);
	ll.push(4);
	ll.push(15);
	ll.push(85);
	cout << "Given linked list\n";
	ll.print();
	ll.reverse();
	cout << "\nReversed Linked list \n";
	ll.print();
	return 0;
}
```

Reverse a Linked List (Recursive)

```
#include <iostream>
using namespace std;
struct Node {
	int data;
	struct Node* next;
	Node (int data)
	{
		this->data = data;
		next = NULL;
	}
};

struct LinkedList {
	Node* head;
	LinkedList ()
	{
		head = NULL;
	}

	Node* reverse (Node* head)
	{
		if (head == NULL || head->next == NULL)
			return head;
		Node* rest = reverse(head->next);
		head->next->next = head;
		head->next = NULL;
		return rest;
	}

	void print ()
	{
		struct Node* temp = head;
		while (temp != NULL) {
			cout << temp->data << " ";
			temp = temp->next;
		}
	}

	void push (int data)
	{
		Node* temp = new Node(data);
		temp->next = head;
		head = temp;
	}
};

int main ()
{
	LinkedList ll;
	ll.push(20);
	ll.push(4);
	ll.push(15);
	ll.push(85);
	cout << "Given linked list\n";
	ll.print();
	ll.head = ll.reverse(ll.head);
	cout << "\nReversed Linked list \n";
	ll.print();
	return 0;
}
```

Print Linked List Forward

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

class Node {
public:
	int data;
	Node* next;
};

void printList(Node* n)
{
	while (n != NULL) {
		cout << n->data << " ";
		n = n->next;
	}
}
```

```
int main ()
{
	Node* head = NULL;
	Node* second = NULL;
	Node* third = NULL;
	head = new Node ();
	second = new Node ();
	third = new Node ();
	head->data = 1;
	head->next = second;
	second->data = 2;
	second->next = third;
	third->data = 3;
	third->next = NULL;
	printList(head);
	return 0;
}
```

Print Linked List Reverse

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

class Node
{
	public:
	int data;
	Node* next;
};

void printReverse(Node* head)
{
	if (head == NULL)
	return;
	printReverse(head->next);
	cout << head->data << " ";
}

void push (Node** head_ref, char new_data)
{
	Node* new_node = new Node ();
	new_node->data = new_data;
	new_node->next = (*head_ref);
	(*head_ref) = new_node;
}

int main ()
{
	Node* head = NULL;
	push (&head, 4);
	push (&head, 3);
	push (&head, 2);
	push (&head, 1);

	printReverse(head);
	return 0;
}
```

Chapter – 4

Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with i'th next pointer and data which are there in singly linked list.

The following are the advantages/disadvantages of a doubly-linked list over the singly linked list.

Advantages over singly linked list

1. A DLL can be traversed in both forward and backward directions.
2. The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
3. 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 g previous pointer.

Disadvantages over singly linked list

1. 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).
2. 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.

Implementation

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

// A linked list node
class Node
{
	public:
	int data;
	Node* next;
	Node* prev;
};

void push(Node** head_ref, int new_data)
{
	Node* new_node = new Node();
	new_node->data = new_data;
	new_node->next = (*head_ref);
	new_node->prev = NULL;
	if ((*head_ref) != NULL)
		(*head_ref)->prev = new_node;
	(*head_ref) = new_node;
}
```

```
void insertAfter(Node* prev_node, int new_data)
{
        if (prev_node == NULL)
        {
                cout<<"the given previous node cannot be NULL";
                return;
        }
        Node* new_node = new Node();
        new_node->data = new_data;
        new_node->next = prev_node->next;
        prev_node->next = new_node;
        new_node->prev = prev_node;
        if (new_node->next != NULL)
                new_node->next->prev = new_node;
}

void append(Node** head_ref, int new_data)
{
        Node* new_node = new Node();
        Node* last = *head_ref; /* used in step 5*/
        new_node->data = new_data;
        new_node->next = NULL;
        if (*head_ref == NULL)
        {
                new_node->prev = NULL;
                *head_ref = new_node;
                return;
        }
        while (last->next != NULL)
                last = last->next;
        last->next = new_node;
        new_node->prev = last;
        return;
}

void printList(Node* node)
{
        Node* last;
        cout<<"\nTraversal in forward direction \n";
        while (node != NULL)
        {
                cout<<" "<<node->data<<" ";
                last = node;
                node = node->next;
        }
        cout<<"\nTraversal in reverse direction \n";
        while (last != NULL)
        {
                cout<<" "<<last->data<<" ";
                last = last->prev;
        }
}

int main ()
{
        Node* head = NULL;
        append (&head, 6);
        push (&head, 7);
        push (&head, 1);
        append (&head, 4);
        insertAfter(head->next, 8);
        cout << "Created DLL is: ";
        printList(head);
        return 0;
}
```

Chapter – 5

Stack

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last 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. Some of its main operations are: push (), pop (), top (), isEmpty(), size(), etc. To make manipulations in a stack, there are certain operations provided to us. When we want to insert an element into the stack the operation is known as the push operation whereas when we want to remove an element from the stack the operation is known as the pop operation. If we try to pop from an empty stack then it is known as underflow and if we try to push an element in a stack that is already full, then it is known as overflow.

Mainly the following four basic operations are performed in the stack:

- Push: Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.
- 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.
- Peek or Top: Returns the top element of the stack.
- isEmpty: Returns true if the stack is empty, else false.

Advantages

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines like JVM.
- Stacks are more secure and reliable as they do not get corrupted easily.
- Stack cleans up the objects automatically.

Disadvantages

- Requires extra memory due to the involvement of pointers.
- Random access is not possible in stathe ck.
- The total size of the stack must be defined before.
- If the stack falls outside the memory it can lead to abnormal termination.

Implementation (Array)

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000
```

```
class Stack {
        int top;
public:
        int a[MAX]; // Maximum size of Stack
        Stack() { top = -1; }
        bool push(int x);
        int pop();
        int peek();
        bool isEmpty();
};

bool Stack::push(int x)
{
        if (top >= (MAX - 1)) {
                cout << "Stack Overflow";
                return false;
        }
        else {
                a[++top] = x;
                cout << x << " pushed into stack\n";
                return true;
        }
}

int Stack::pop()
{
        if (top < 0) {
                cout << "Stack Underflow";
                return 0;
        }
        else {
                int x = a[top--];
                return x;
        }
}
int Stack::peek()
{
        if (top < 0) {
                cout << "Stack is Empty";
                return 0;
        }
        else {
                int x = a[top];
                return x;
        }
}

bool Stack::isEmpty()
{
        return (top < 0);
}

int main()
{
        class Stack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout << s.pop() << " Popped from stack\n";
        cout << "Top element is : " << s.peek() << endl;
        cout<<"Elements present in stack : ";
        while(!s.isEmpty())
        {
                cout<<s.peek()<<" ";
                s.pop();
```

```
	}
	return 0;
}
```

Implementation (Linked List)

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

class StackNode {
public:
	int data;
	StackNode* next;
};

StackNode* newNode(int data)
{
	StackNode* stackNode = new StackNode();
	stackNode->data = data;
	stackNode->next = NULL;
	return stackNode;
}

int isEmpty(StackNode* root)
{
	return !root;
}

void push(StackNode** root, int data)
{
	StackNode* stackNode = newNode(data);
	stackNode->next = *root;
	*root = stackNode;
	cout << data << " pushed to stack\n";
}

int pop(StackNode** root)
{
	if (isEmpty(*root))
		return INT_MIN;
	StackNode* temp = *root;
	*root = (*root)->next;
	int popped = temp->data;
	free(temp);

	return popped;
}

int peek(StackNode* root)
{
	if (isEmpty(root))
		return INT_MIN;
	return root->data;
}

int main()
{
	StackNode* root = NULL;
	push(&root, 10);
	push(&root, 20);
	push(&root, 30);
	cout << pop(&root) << " popped from stack\n";
	cout << "Top element is " << peek(root) << endl;
	cout<<"Elements present in stack : ";
```

```
        while(!isEmpty(root))
        {
                cout<<peek(root)<<" ";
                pop(&root);
        }

        return 0;
}
```

Reverse String using Stack

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

class Stack
{
public:
    int top;
    unsigned capacity;
    char *array;
};

Stack *createStack(unsigned capacity)
{
    Stack *stack = new Stack();
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = new char[(stack->capacity * sizeof(char))];
    return stack;
}

int isFull(Stack *stack)
{
    return stack->top == stack->capacity - 1;
}

int isEmpty(Stack *stack)
{
    return stack->top == -1;
}

void push(Stack *stack, char item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}

char pop(Stack *stack)
{
    if (isEmpty(stack))
        return -1;
    return stack->array[stack->top--];
}

void reverse(char str[])
{

    int n = strlen(str);
    Stack *stack = createStack(n);

    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

int main()
{
    char str[] = "GeeksQuiz";

    reverse(str);
    cout << "Reversed string is " << str;

    return 0;
}
```

Check for balanced parentheses using stack

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

bool areBracketsBalanced(string expr)
{
    stack<char> temp;
    for (int i = 0; i < expr.length(); i++)
    {
        if (temp.empty())
        {
            temp.push(expr[i]);
        }
        else if ((temp.top() == '(' && expr[i] == ')') || (temp.top() == '{' && expr[i] == '}') || (temp.top() == '[' && expr[i] == ']'))
        {
            temp.pop();
        }
        else
        {
            temp.push(expr[i]);
        }
    }
    if (temp.empty())
    {
        return true;
    }
    return false;
}

int main()
{
    string expr = "{()}[]";
    if (areBracketsBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

Evaluation of Prefix Expression

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

bool isOperand(char c)
{
    return isdigit(c);
}

double evaluatePrefix(string express)
{
    stack<double> Stack;

    for (int j = exprsn.size() - 1; j >= 0; j--)
    {
        if (isOperand(exprsn[j]))
            Stack.push(exprsn[j] - '0');

        else
        {
            double o1 = Stack.top();
            Stack.pop();
            double o2 = Stack.top();
            Stack.pop();
            switch (exprsn[j])
            {
            case '+':
                Stack.push(o1 + o2);
                break;
            case '-':
                Stack.push(o1 - o2);
                break;
            case '*':
                Stack.push(o1 * o2);
                break;
            case '/':
                Stack.push(o1 / o2);
                break;
            }
        }
    }

    return Stack.top();
}

int main()
{
    string exprsn = "+9*26";
    cout << evaluatePrefix(exprsn) << endl;
    return 0;
}
```

Evaluation of Postfix Expression

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

using namespace std;

struct Stack
{
    int top;
    unsigned capacity;
    int *array;
};

struct Stack *createStack(unsigned capacity)
{
    struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int *)malloc(stack->capacity * sizeof(int));

    if (!stack->array)
        return NULL;

    return stack;
}

int isEmpty(struct Stack *stack)
{
    return stack->top == -1;
}

char peek(struct Stack *stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack *stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}

void push(struct Stack *stack, char op)
{
    stack->array[++stack->top] = op;
}

int evaluatePostfix(char *exp)
{
    struct Stack *stack = createStack(strlen(exp));
    int i;
    if (!stack)
        return -1;
    for (i = 0; exp[i]; ++i)
    {
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+':
                push(stack, val2 + val1);
                break;
            case '-':
                push(stack, val2 - val1);
                break;
            case '*':
                push(stack, val2 * val1);
                break;
            case '/':
                push(stack, val2 / val1);
                break;
            }
        }
    }
    return pop(stack);
}

int main()
{
    char exp[] = "231*+9-";
    cout << "postfix evaluation: " << evaluatePostfix(exp);
    return 0;
}
```

Chapter – 6

Queue

Similar to Stack, Queue is a linear data structure that follows a particular order in which the operations are performed for storing data. The order is First In First Out (FIFO). One can imagine a queue as a line of people waiting to receive something in sequential order which starts from the beginning of the line. It is an ordered list in which insertions are done at one end which is known as the rear and deletions are done from the other end known as the front. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Basic Operations on Queue

- void enqueue(int data): Inserts an element at the end of the queue i.e. at the rear end.
- int dequeue(): This operation removes and returns an element that is at the front end of the queue.

Auxiliary Operations on Queue

- int front(): This operation returns the element at the front end without removing it.
- int rear(): This operation returns the element at the rear end without removing it.
- int isEmpty(): This operation indicates whether the queue is empty or not.
- int size(): This operation returns the size of the queue i.e. the total number of elements it contains.

Types of Queues

- Simple Queue: Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.
- Circular Queue: In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is a space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.
- Priority Queue: This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the element with the lowest value gets the highest priority so in turn, it creates a queue with increasing order of values.
- Dequeue: Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the

queue, unlike the other queues in which it can be done only from one end. Because of this propert,y it may not obey the First In First Out property.

Applications

Queue is used when things don’t have to be processed immediately, but have to be processed in First In First Out order like Breadth-First Search. This property of Queue makes it also useful in the following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3) Queue can be used as an essential component in various other data structures.

Implementation (Array)

```
#include <iostream>
using namespace std;
int queue[100], n = 100, front = - 1, rear = - 1;
void Insert() {
  int val;
  if (rear == n - 1)
  cout<<"Queue Overflow"<<endl;
  else {
    if (front == - 1)
    front = 0;
    cout<<"Insert the element in queue : "<<endl;
    cin>>val;
    rear++;
    queue[rear] = val;
  }
}
void Delete() {
  if (front == - 1 || front > rear) {
    cout<<"Queue Underflow ";
    return ;
  } else {
    cout<<"Element deleted from queue is : "<< queue[front] <<endl;
    front++;;
  }
}
void Display() {
  if (front == - 1)
  cout<<"Queue is empty"<<endl;
  else {
    cout<<"Queue elements are : ";
    for (int i = front; i <= rear; i++)
    cout<<queue[i]<<" ";
      cout<<endl;
  }
}
int main() {
  int ch;
  cout<<"1) Insert element to queue"<<endl;
  cout<<"2) Delete element from queue"<<endl;
  cout<<"3) Display all the elements of queue"<<endl;
```

```
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: Insert();
         break;
         case 2: Delete();
         break;
         case 3: Display();
         break;
         case 4: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
   return 0;
}
```

Implementation (Linked List)

```
#include <iostream>
using namespace std;
struct node {
   int data;
   struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert() {
   int val;
   cout<<"Insert the element in queue : "<<endl;
   cin>>val;
   if (rear == NULL) {
      rear = (struct node *)malloc(sizeof(struct node));
      rear->next = NULL;
      rear->data = val;
      front = rear;
   } else {
      temp=(struct node *)malloc(sizeof(struct node));
      rear->next = temp;
      temp->data = val;
      temp->next = NULL;
      rear = temp;
   }
}
void Delete() {
   temp = front;
   if (front == NULL) {
      cout<<"Underflow"<<endl;
      return;
   }
   else
   if (temp->next != NULL) {
      temp = temp->next;
      cout<<"Element deleted from queue is : "<<front->data<<endl;
      free(front);
      front = temp;
   } else {
      cout<<"Element deleted from queue is : "<<front->data<<endl;
      free(front);
      front = NULL;
      rear = NULL;
   }
}
void Display() {
   temp = front;
   if ((front == NULL) && (rear == NULL)) {
      cout<<"Queue is empty"<<endl;
      return;
   }
   cout<<"Queue elements are: ";
   while (temp != NULL) {
      cout<<temp->data<<" ";
      temp = temp->next;
   }
   cout<<endl;
}
int main() {
   int ch;
   cout<<"1) Insert element to queue"<<endl;
   cout<<"2) Delete element from queue"<<endl;
   cout<<"3) Display all the elements of queue"<<endl;
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: Insert();
         break;
         case 2: Delete();
         break;
         case 3: Display();
         break;
         case 4: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
   return 0;
}
```

Chapter – 7

Trees

The data structure is a specialized method to organize and store data in the computer to be used more effectively. There are various types of data structures, such as stack, linked list, and queue, arranged in sequential order. For example, a linked data structure consists of a set of nodes that are linked together by links or points.

Similarly, the tree data structure is a kind of hierarchal data arranged in a tree-like structure. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected.

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the "children").

Basic Terminology in Tree Data Structure:

- Parent Node: The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.
- Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.
- Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- Degree of a Node: The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree. The degree of the node {3} is 3.
- Leaf Node or External Node: The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.
- Ancestor of a Node: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the parent nodes of the node {7}
- Descendant: Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.
- Sibling: Children of the same parent node are called siblings. {8, 9, 10} are called siblings.
- Depth of a node: The count of edges from the root to the node. Depth of node {14} is 3.
- Height of a node: The number of edges on the longest path from that node to a leaf. Height of node {3} is 2.
- Height of a tree: The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is 3.
- Level of a node: The count of edges on the path from the root node to that node. The root node has level 0.
- Internal node: A node with at least one child is called Internal Node.

- Neighbor of a Node: Parent or child nodes of that node are called neighbors of that node.
- Subtree: Any node of the tree along with its descendant

Chapter - 8

Binary Tree

A tree is a popular data structure that is nonlinear Unlike other data structures like array, stack , queue, and linked list that lists linear in nature represents hierarchical structure. The ordering information of tree is not important. A tree contains nodes and 2 pointers. These two pointers are the left child and the right child of the parent node. Let us understand the terms of tree in detail.

- Root: The root of a tree is the top most node of the tree that has no parent node. There is only one root node in every tree.
- Edge: Edge acts as a link between the parent node and the child node.
- Leaf: A node that has no child is known as the leaf node. It is the last node of the tree. There can be multiple leaf nodes in a tree.
- Depth: Depth of the node is the distance from the root node to that particular node.
- Height: The height of the node is the distance from that node to the deepest node of the tree.
- Height of tree: Height of the tree is the maximum height of any node.

Applications

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

Basic Operation on Binary Tree

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing an element. There are three types of traversals in binary tree which will be discussed ahead.

Auxiliary Operation on Binary Tree

- Finding the height of the tree
- Find the level of the tree
- Finding the size of the entire tree.

Applications of Binary Tree

- In compilers, Expression Trees are used which is an application of binary tree.
- Huffman coding trees are used in data compression algorithms.

- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(logn) time complexity.

Binary Tree Traversals

- PreOrder Traversal: Here, the traversal is : root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
- InOrder Traversal: Here, the traversal is : left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.
- PostOrder Traversal: Here, the traversal is : left child – right child – root. It means that the left child is traversed first then the right child and finally its root node.

Chapter - 9

Binary Search Tree

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. There must be no duplicate nodes.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search for a given key.

Implementation

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

class BST
{
    int data;
    BST *left, *right;

public:
    BST();
    BST(int);
    BST *Insert(BST *, int);
    void Inorder(BST *);
};

BST ::BST()
    : data(0), left(NULL), right(NULL)
{
}

BST ::BST(int value)
{
    data = value;
    left = right = NULL;
}

BST *BST ::Insert(BST *root, int value)
{
    if (!root)
    {
        return new BST(value);
    }

    if (value > root->data)
    {
        root->right = Insert(root->right, value);
    }
    else
    {
        root->left = Insert(root->left, value);
    }

    return root;
}

void BST ::Inorder(BST *root)
{
    if (!root)
    {
        return;
    }
    Inorder(root->left);
    cout << root->data << endl;
    Inorder(root->right);
}

int main()
{
    BST b, *root = NULL;
    root = b.Insert(root, 50);
    b.Insert(root, 30);
    b.Insert(root, 20);
    b.Insert(root, 40);
    b.Insert(root, 70);
    b.Insert(root, 60);
    b.Insert(root, 80);
    b.Inorder(root);
    return 0;
}
```

Max and Min in BST

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

class Node
{
public:
    int data;
    Node *left, *right;
    Node(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }
};

int findMax(Node *root)
{
    if (root == NULL)
        return INT_MIN;
    int res = root->data;
    int lres = findMax(root->left);
    int rres = findMax(root->right);
    if (lres > res)
        res = lres;
    if (rres > res)
        res = rres;
    return res;
}
```

```
int main()
{
    Node *NewRoot = NULL;
    Node *root = new Node(2);
    root->left = new Node(7);
    root->right = new Node(5);
    root->left->right = new Node(6);
    root->left->right->left = new Node(1);
    root->left->right->right = new Node(11);
    root->right->right = new Node(9);
    root->right->right->left = new Node(4);
    cout << "Maximum element is " << findMax(root) << endl;
    return 0;
}
```

Height of tree

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

class node
{
public:
    int data;
    node *left;
    node *right;
};

int maxDepth(node *node)
{
    if (node == NULL)
        return 0;
    else
    {
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}

node *newNode(int data)
{
    node *Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;
    return (Node);
}

int main()
{
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << "Height of tree is " << maxDepth(root);
    return 0;
}
```

Breadth First Traversal

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

class node
{
public:
    int data;
    node *left, *right;
};

void printCurrentLevel(node *root, int level);
int height(node *node);
node *newNode(int data);

void printLevelOrder(node *root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++)
        printCurrentLevel(root, i);
}

void printCurrentLevel(node *root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        cout << root->data << " ";
    else if (level > 1)
    {
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}

int height(node *node)
{
    if (node == NULL)
        return 0;
    else
    {
        int lheight = height(node->left);
        int rheight = height(node->right);
        if (lheight > rheight){
            return (lheight + 1);
        }else{
            return (rheight + 1);
        }
    }
}

node *newNode(int data)
{
    node *Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

int main()
{
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}
```

Depth First Traversal

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

struct Node
{
    int data;
    struct Node *left, *right;
};

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

void printPostorder(struct Node *node)
{
    if (node == NULL)
        return;
    printPostorder(node->left);
    printPostorder(node->right);
    cout << node->data << " ";
}

void printInorder(struct Node *node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->data << " ";
    printInorder(node->right);
}

void printPreorder(struct Node *node)
{
    if (node == NULL)
        return;
    cout << node->data << " ";
    printPreorder(node->left);
    printPreorder(node->right);
}

int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);
    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);
    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);

    return 0;
}
```

Check if Binary Tree is BST

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

class node
{
public:
    int data;
    node *left;
    node *right;
    node(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }
};

int isBSTUtil(node *node, int min, int max);

int isBST(node *node)
{
    return (isBSTUtil(node, INT_MIN, INT_MAX));
}

int isBSTUtil(node *node, int min, int max)
{
    if (node == NULL)
        return 1;
    if (node->data < min || node->data > max)
        return 0;
    return isBSTUtil(node->left, min, node->data - 1) &&
           isBSTUtil(node->right, node->data + 1, max);
}

int main()
{
    node *root = new node(4);
    root->left = new node(2);
    root->right = new node(5);
    root->left->left = new node(1);
    root->left->right = new node(3);

    if (isBST(root))
        cout << "Is BST";
    else
        cout << "Not a BST";

    return 0;
}
```

Delete a Node in BST

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

struct Node
{
    int key;
    struct Node *left, *right;
};

Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

void inorder(Node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

Node *insert(Node *node, int key)
{
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

Node *deleteNode(Node *root, int k)
{
    if (root == NULL)
        return root;
    if (root->key > k)
    {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->key < k)
    {
        root->right = deleteNode(root->right, k);
        return root;
    }
    if (root->left == NULL)
    {
        Node *temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL)
    {
        Node *temp = root->left;
        delete root;
        return temp;
    }
    else
    {
        Node *succParent = root;
        Node *succ = root->right;
        while (succ->left != NULL)
        {
            succParent = succ;
            succ = succ->left;
        }
        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;
        root->key = succ->key;
        delete succ;
        return root;
    }
}

int main()
{
    Node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
    printf("Inorder traversal of the given tree \n");
    inorder(root);
    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);
    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);
    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);
    return 0;
}
```

Chapter – 10

Graphs

A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, and locale.

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Edge Lists

The first implementation strategy is called an edge list. An edge list is a list or array of all the edges in a graph. Edge lists are one of the easier representations of a graph.

In this implementation, the underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.

Adjacency Matrix

While an edge list won't end up being the most efficient choice, we can move beyond a list and implement a matrix. For many, a matrix is a significantly better kinesthetic representation for a graph.

An adjacency matrix is a matrix that represents exactly which vertices/nodes in a graph have edges between them. It serves as a lookup table, where a value of 1 represents an edge that exists and a 0 represents an edge that does not exist. The indices of the matrix model the nodes.

Once we've determined the two nodes that we want to find an edge between, we look at the value at the intersection of those two nodes to determine whether there's a link.

Adjacency Lists

Adjacency list is a hybrid between an adjacency matrix and an edge list. An adjacency list is an array of linked lists that serves the purpose of representing a graph. What makes it unique is that its shape also makes it easy to see which vertices are adjacent to any other vertices. Each vertex in a graph can easily reference its neighbors through a linked list.

Due to this, an adjacency list is the most common representation of a graph. Another reason is that graph traversal problems often require us to be able to easily figure out which nodes are the neighbors of another node. In most graph traversal interview problems, we don't really need to build the entire graph. Rather, it's important to know where we can travel (or in other words, who the neighbors of a node are).