**CS2023 - Data Structures and Algorithms**

**In-class Lab Exercise**

Week 6

**Submission By: Sajeev Kugarajah (210554M)**

Github Repo link:

Implementation of Stack using Array

#include <iostream>

#include <chrono>

using namespace std;

// implementing stack using array

class Stack

{

int \*stack;

int top;

int capacity;

public:

Stack(int size)

{

stack = new int[size];

top = 0;

capacity = size;

}

// insert element

void push(int el)

{

if (top == capacity)

{

cout << "Stack Overflow" << endl;

return;

}

stack[top] = el;

top++;

}

// delete element

int pop()

{

if (isEmpty())

{

cout << "Stack Underflow" << endl;

return 0;

}

top--;

return stack[top];

}

bool isEmpty()

{

return top == 0;

}

bool isFull()

{

return top == capacity;

}

int stackTop()

{

if (isEmpty())

{

cout << "Stack is empty" << endl;

return 0;

}

return stack[top - 1];

}

void display()

{

for (int i = top - 1; i >= 0; i--)

{

cout << stack[i] << " ";

}

cout << endl;

}

};

int main()

{

auto start\_time = chrono::steady\_clock::now();

// implementing stack and methods

Stack myStack(10);

myStack.push(10);

myStack.push(20);

myStack.push(30);

myStack.push(40);

myStack.push(50);

myStack.push(60);

myStack.push(70);

myStack.push(80);

myStack.push(90);

myStack.push(100);

myStack.display();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.display();

myStack.push(110);

myStack.push(120);

myStack.push(130);

myStack.push(140);

myStack.display();

auto end\_time = chrono::steady\_clock::now();

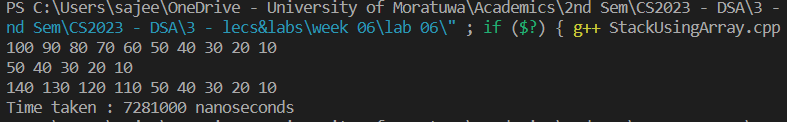
auto time\_taken = chrono::duration\_cast<chrono::nanoseconds>(end\_time - start\_time).count();

cout << "Time taken : " << time\_taken << " nanoseconds" << endl;

return 0;

}

CLI output



Implementing Stack using Linked List

#include <iostream>

#include <chrono>

using namespace std;

// node of linked list

struct Node

{

int data;

Node \*next;

};

// implementing stack using linked list

class Stack

{

Node \*top;

// constructor

public:

Stack()

{

top = NULL;

}

// push to top

void push(int el)

{

Node \*temp = new Node;

temp->data = el;

temp->next = top;

top = temp;

}

// popping from top

int pop()

{

if (isEmpty())

{

cout << "Stack Underflow" << endl;

return 0;

}

Node \*temp = top;

top = top->next;

int el = temp->data;

delete temp;

return el;

}

// checking if the stack is empty

bool isEmpty()

{

return top == NULL;

}

// checking if the stack is full

bool isFull()

{

Node \*temp = new Node;

if (temp == NULL)

{

delete temp;

return true;

}

delete temp;

return false;

}

// finding the top element

int stackTop()

{

if (isEmpty())

{

cout << "Stack is empty" << endl;

return 0;

}

return top->data;

}

// printing the stack

void display()

{

if (isEmpty())

{

cout << "Stack is empty" << endl;

return;

}

Node \*temp = top;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

};

int main()

{

// start clock

auto start\_time = chrono::steady\_clock::now();

// implementing stack and methods

Stack myStack;

myStack.push(10);

myStack.push(20);

myStack.push(30);

myStack.push(40);

myStack.push(50);

myStack.push(60);

myStack.push(70);

myStack.push(80);

myStack.push(90);

myStack.push(100);

myStack.display();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.pop();

myStack.display();

myStack.push(110);

myStack.push(120);

myStack.push(130);

myStack.push(140);

myStack.display();

// end clock

auto end\_time = chrono::steady\_clock::now();

auto time\_taken =

chrono::duration\_cast<chrono::nanoseconds>(end\_time –

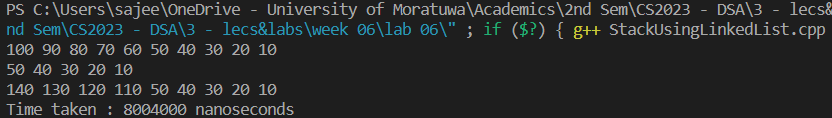
start\_time).count();

cout << "Time taken : " << time\_taken << " nanoseconds" << endl;

return 0;

}

CLI output



Discussion

Implementing stack data structure using array can be simple and efficient in memory, but the size of the stack is fixed at initialization. Unlike array-based stacks, linked list – based stacks are dynamic. We don’t have to define the size of stack at the initialization. But linked lists require more memory to store overhead for storing pointer for each element. And they may have inefficient access of elements.

Since in both type of implementations, the element is pushed and popped from front or beginning, the time complexity for both push() and pop() operations in both implementations are O(1). Hence, it’s proved by the CLI output we get, that the time taken to execute both type of implementations is approximately equal.

But, in addition, we have to consider the memory efficiency, as we said in some cases, the array-based stack will require less time to access elements in stack.

So, the implementation type which has to be implemented is entirely depended on the context. If we can initially define the size of the stack, we can use array-based implementation as it also has faster accessibility to the elements. But if we need a dynamic stack, we can use linked-list-based stack structure.