Stack Data Structure (Introduction and Program)

- Difficulty Level : **Easy**
- Last Updated : 19 Jul, 2021

Stack is a linear data structure which 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).

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

- Push: Adds an item in 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 top element of stack.
- isEmpty: Returns true if stack is empty, else false.

How to understand a stack practically?
There are many real-life examples of a stack. Consider the simple example ofplates 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 LIFO/FILO order.
Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

- **Balancing of symbols**
- **Infix to Postfix** /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, **stock span problem**, **histogram problem**.
- Backtracking is one of the algorithm designing technique .Some example of back tracking are Knight-Tour problem,N-Queen problem,find your way through maze and game like chess or checkers in all this 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 current state we need to store the previous state for that purpose we need stack.
- In Graph Algorithms like **Topological Sorting** and **Strongly Connected Components**
- In Memory management any modern computer uses stack as the primary-management for a running purpose.Each program that is running in a computer system has its own memory allocations
- String reversal is also a another application of stack.Here one by one each character get inserted into the stack.So the first character of string is on the bottom of the stack and the last element of string is on the top of stack. After Performing the pop operations on stack we get string in reverse order .

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Implementing Stack using Arrays

- C++

- C
- Java
- Python
- C#

```
/* C++ program to implement basic stack
   operations */
#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);
}

// Driver program to test above functions
int main()
{
        class Stack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout << s.pop() << " Popped from stack\n";
        //print all elements in stack :
        cout<<"Elements present in stack : ";
        while(!s.isEmpty())
        {
                // print top element in stack
                cout<<s.peek()<<" ";
                // remove top element in stack
                s.pop();
        }

        return 0;
}
```

Output :

10 pushed into stack

20 pushed into stack

30 pushed into stack

30 Popped from stack

Top element is : 20

Elements present in stack : 20 10

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

Implementing Stack using Linked List:

- C++

- C
- Java
- Python
- C#

```
// C++ program for linked list implementation of stack
#include <bits/stdc++.h>
using namespace std;

// A structure to represent a stack
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;
}

// Driver code
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 : ";
          //print all elements in stack :
        while(!isEmpty(root))
        {
                // print top element in stack
                cout<<peek(root)<<" ";
                // remove top element in stack
                pop(&root);
        }

        return 0;
}
```

// This is code is contributed by rathbhupendra

Output:

10 pushed to stack

20 pushed to stack

30 pushed to stack

30 popped from stack

Top element is 20

Elements present in stack : 20 10

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

Stack | Set 2 (Infix to Postfix)

- Difficulty Level : **Medium**
- Last Updated : 09 Jul, 2021

Prerequisite – **Stack | Set 1 (Introduction)**

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or

prefix) form before evaluation.

The corresponding expression in postfix form is abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,

    1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.

    2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Recommended: Please solve it on "*PRACTICE* " first, before moving on to the solution.

Following is the implementation of the above algorithm

- C++

- C
- Java
- Python
- C#

```
/* C++ implementation to convert
infix expression to postfix*/

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

//Function to return precedence of operators
int prec(char c) {
	if(c == '^')
		return 3;
	else if(c == '/' || c=='*')
		return 2;
	else if(c == '+' || c == '-')
		return 1;
	else
		return -1;
}

// The main function to convert infix expression
//to postfix expression
void infixToPostfix(string s) {

	stack<char> st; //For stack operations, we are using C++ built in stack
	string result;

	for(int i = 0; i < s.length(); i++) {
		char c = s[i];

		// If the scanned character is
		// an operand, add it to output string.
		if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
			result += c;

		// If the scanned character is an
		// ‘(‘, push it to the stack.
		else if(c == '(')
			st.push('(');

		// If the scanned character is an ‘)’,
		// pop and to output string from the stack
		// until an ‘(‘ is encountered.
		else if(c == ')') {
			while(st.top() != '(')
			{
				result += st.top();
				st.pop();
			}
			st.pop();
		}
```

```
            //If an operator is scanned
            else {
                while(!st.empty() && prec(s[i]) <= prec(st.top())) {
                    result += st.top();
                    st.pop();
                }
                st.push(c);
            }
        }

        // Pop all the remaining elements from the stack
        while(!st.empty()) {
            result += st.top();
            st.pop();
        }

        cout << result << endl;
}

//Driver program to test above functions
int main() {
        string exp = "a+b*(c^d-e)^(f+g*h)-i";
        infixToPostfix(exp);
        return 0;
}
```

Output

abcd^e-fgh*+^*+i-

Stack | Set 3 (Reverse a string using stack)

- Difficulty Level : **Easy**
- Last Updated : 24 Jun, 2021

Given a string, reverse it using stack. For example “GeeksQuiz” should be converted to

“ziuQskeeG”.

Following is simple algorithm to reverse a string using stack.

1) Create an empty stack.

2) One by one push all characters of string to stack.

3) One by one pop all characters from stack and put

them back to string.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Following programs implements above algorithm.

- C++

- C
- Java
- Python
- C#
- Javascript

```
// C++ program to reverse a string using stack
#include <bits/stdc++.h>
using namespace std;

// A structure to represent a stack
class Stack
{
	public:
	int top;
	unsigned capacity;
	char* array;
};

// function to create a stack of given
// capacity. It initializes size of stack as 0
Stack* createStack(unsigned capacity)
{
	Stack* stack = new Stack();
	stack->capacity = capacity;
	stack->top = -1;
	stack->array = new char[(stack->capacity * sizeof(char))];
	return stack;
}

// Stack is full when top is equal to the last index
int isFull(Stack* stack)
{ return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(Stack* stack)
{ return stack->top == -1; }

// Function to add an item to stack.
// It increases top by 1
```

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

// Function to remove an item from stack.
// It decreases top by 1
char pop(Stack* stack)
{
	if (isEmpty(stack))
		return -1;
	return stack->array[stack->top--];
}

// A stack based function to reverse a string
void reverse(char str[])
{
	// Create a stack of capacity
	//equal to length of string
	int n = strlen(str);
	Stack* stack = createStack(n);

	// Push all characters of string to stack
	int i;
	for (i = 0; i < n; i++)
		push(stack, str[i]);

	// Pop all characters of string and
	// put them back to str
	for (i = 0; i < n; i++)
		str[i] = pop(stack);
}

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

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

	return 0;
}

// This code is contributed by rathbhupendra
```

Output:

Reversed string is ziuQskeeG

Time Complexity: O(n) where n is number of characters in stack.

Auxiliary Space: O(n) for stack.

A string can also be reversed without using any auxiliary space. Following C, Java, C# and Python programs to implement reverse without using stack.

- C++

- C
- Java
- Python
- C#
- Javascript

```
// C++ program to reverse a string without using stack
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two characters
void swap(char *a, char *b)
{
	char temp = *a;
	*a = *b;
	*b = temp;
}

// A stack based function to reverse a string
void reverse(char str[])
{
	// get size of string
	int n = strlen(str), i;

	for (i = 0; i < n/2; i++)
		swap(&str[i], &str[n-i-1]);
}

// Driver program to test above functions
int main()
{
	char str[] = "abc";

	reverse(str);
```

```
        cout<<"Reversed string is "<< str;

        return 0;
}

//This is code is contributed by rathbhupendra
```

Output:

Reversed string is cba

The Stock Span Problem

- Difficulty Level : **Medium**
- Last Updated : 22 Jul, 2021

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}

Recommended: Please solve it on "*PRACTICE* " first, before moving on to the solution.

A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on the left of it and increment the span value of it while elements on the left side are smaller.

Following is the implementation of this method:

- C++

- C
- Java
- Python3
- C#
- PHP
- Javascript

```cpp
// C++ program for brute force method
// to calculate stock span values
#include <bits/stdc++.h>
using namespace std;

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
	// Span value of first day is always 1
	S[0] = 1;

	// Calculate span value of remaining days
	// by linearly checking previous days
	for (int i = 1; i < n; i++)
	{
		S[i] = 1; // Initialize span value

		// Traverse left while the next element
		// on left is smaller than price[i]
		for (int j = i - 1; (j >= 0) &&
					(price[i] >= price[j]); j--)
			S[i]++;
	}
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
	for (int i = 0; i < n; i++)
		cout << arr[i] << " ";
}

// Driver code
int main()
{
	int price[] = { 10, 4, 5, 90, 120, 80 };
	int n = sizeof(price) / sizeof(price[0]);
	int S[n];
```

```
        // Fill the span values in array S[]
        calculateSpan(price, n, S);

        // print the calculated span values
        printArray(S, n);

        return 0;
}

// This is code is contributed by rathbhupendra
```

Output

1 1 2 4 5 1

The Time Complexity of the above method is O(n^2). We can calculate stock span values in O(n) time.

A Linear-Time Complexity Method

We see that S[i] on the day i can be easily computed if we know the closest day preceding i, such that the price is greater than on that day than the price on the day i. If such a day exists, let's call it h(i), otherwise, we define h(i) = -1.

The span is now computed as S[i] = i – h(i). See the following diagram.

To implement this logic, we use a stack as an abstract data type to store the days i, h(i), h(h(i)), and so on. When we go from day i-1 to i, we pop the days when the price of the stock was less than or equal to price[i] and then push the value of day i back into the stack.

Following is the implementation of this method.

- C++
- Java
- Python3
- C#

```
// C++ linear time solution for stock span problem
#include <iostream>
```

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

// A stack based efficient method to calculate
// stock span values
void calculateSpan(int price[], int n, int S[])
{
	// Create a stack and push index of first
	// element to it
	stack<int> st;
	st.push(0);

	// Span value of first element is always 1
	S[0] = 1;

	// Calculate span values for rest of the elements
	for (int i = 1; i < n; i++) {
		// Pop elements from stack while stack is not
		// empty and top of stack is smaller than
		// price[i]
		while (!st.empty() && price[st.top()] <= price[i])
			st.pop();

		// If stack becomes empty, then price[i] is
		// greater than all elements on left of it,
		// i.e., price[0], price[1], ..price[i-1]. Else
		// price[i] is greater than elements after
		// top of stack
		S[i] = (st.empty()) ? (i + 1) : (i - st.top());

		// Push this element to stack
		st.push(i);
	}
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
	for (int i = 0; i < n; i++)
		cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
	int price[] = { 10, 4, 5, 90, 120, 80 };
	int n = sizeof(price) / sizeof(price[0]);
	int S[n];

	// Fill the span values in array S[]
```

```
	calculateSpan(price, n, S);

	// print the calculated span values
	printArray(S, n);

	return 0;
}
```

Output

1 1 2 4 5 1

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of the array is added and removed from the stack at most once. So there are total 2n operations at most. Assuming that a stack operation takes O(1) time, we can say that the time complexity is O(n).

Auxiliary Space: O(n) in worst case when all elements are sorted in decreasing order.

Another approach: (without using stack)

- C++
- Java
- Python3
- C#

```
// C++ program for a linear time solution for stock
// span problem without using stack
#include <iostream>
#include <stack>
using namespace std;

// An efficient method to calculate stock span values
// implementing the same idea without using stack
void calculateSpan(int A[], int n, int ans[])
{
	// Span value of first element is always 1
	ans[0] = 1;

	// Calculate span values for rest of the elements
	for (int i = 1; i < n; i++) {
		int counter = 1;
		while ((i - counter) >= 0 && A[i] >= A[i - counter]) {
			counter += ans[i - counter];
```

```
            }
            ans[i] = counter;
        }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
        for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
        int price[] = { 10, 4, 5, 90, 120, 80 };
        int n = sizeof(price) / sizeof(price[0]);
        int S[n];

        // Fill the span values in array S[]
        calculateSpan(price, n, S);

        // print the calculated span values
        printArray(S, n);

        return 0;
}
```

Output

1 1 2 4 5 1

A Stack Based approach :

1. In this approach, I have used the data structure stack to implement this task.
2. Here, two stacks are used. One stack stores the actual stock prices whereas, the other stack is a temporary stack.
3. The stock span problem is solved using only the Push and Pop functions of Stack.
4. Just to take input values, I have taken array ‘price’ and to store output, used array ‘span’.

Below is the implementation of the above approach:

- C

```
// C program for the above approach
#include <limits.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 7

// change size of  stack from here
// change this char to int if
// you want to create stack of
// int. rest all program will work fine
typedef int stackentry;

typedef struct stack {
	stackentry entry[SIZE];
	int top;
} STACK;

// stack is initialized by setting top pointer = -1.
void initialiseStack(STACK* s) { s->top = -1; }

// to check if stack is full.
int IsStackfull(STACK s)
{
	if (s.top == SIZE - 1) {
		return (1);
	}
	return (0);
}

// to check if stack is empty.
int IsStackempty(STACK s)
{
	if (s.top == -1) {
		return (1);
	}
	else {
		return (0);
	}
}

// to push elements into the stack.
void push(stackentry d, STACK* s)
{
	if (!IsStackfull(*s)) {

		s->entry[(s->top) + 1] = d;
		s->top = s->top + 1;
	}
}

// to pop element from stack.
stackentry pop(STACK* s)
{
```

```
	stackentry ans;
	if (!IsStackempty(*s)) {
		ans = s->entry[s->top];
		s->top = s->top - 1;
	}
	else {

		// '\0'  will be returned if
		// stack is empty and of
		// char type.
		if (sizeof(stackentry) == 1)
			ans = '\0';
		else
			// INT_MIN  will be returned
			// if stack is empty
			// and of int type.
			ans = INT_MIN;
	}
	return (ans);
}

// The code for implementing stock
// span problem is written
// here in main function.
int main()
{
	// Just to store prices on 7 adjacent days
	int price[7] = { 100, 80, 60, 70, 60, 75, 85 };

	// in span array , span of each day will be stored.
	int span[7] = { 0 };
	int i;

	// stack 's' will store stock values of each
	// day. stack 'temp' is temporary stack
	STACK s, temp;

	// setting top pointer to -1.
	initialiseStack(&s);
	initialiseStack(&temp);

	// count basically signifies span of
	// particular day.
	int count = 1;

	// since first day span is 1 only.
	span[0] = 1;
	push(price[0], &s);
```

```
        // calculate span of remaining days.
        for (i = 1; i < 7; i++) {

                // count will be span of that particular day.
                count = 1;

                // if current day stock is larger than previous day
                // span, then it will be popped out into temp stack.
                // popping will be carried out till span gets over
                // and count will be incremented .

                while (!IsStackempty(s)
                                && s.entry[s.top] <= price[i]) {
                        push(pop(&s), &temp);
                        count++;
                }

                // now, one by one all stocks from temp will be
                // poped and pushed back to s.
                while (!IsStackempty(temp)) {
                        push(pop(&temp), &s);
                }

                // pushing current stock
                push(price[i], &s);

                // appending span of that particular
                // day into output array.
                span[i] = count;
        }

        // printing the output.
        for (i = 0; i < 7; i++)
                printf("%d ", span[i]);
}
```

Output

1 1 1 2 1 4 6

Design a stack with operations on middle element

- Difficulty Level : **Medium**
- Last Updated : 31 May, 2021

How to implement a stack which will support following operations in O(1) time complexity?

1) push() which adds an element to the top of stack.

2) pop() which removes an element from top of stack.

3) findMiddle() which will return middle element of the stack.

4) deleteMiddle() which will delete the middle element.

Push and pop are standard stack operations.

The important question is, whether to use a linked list or array for implementation of stack?

Please note that, we need to find and delete middle element. Deleting an element from middle is not O(1) for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

The idea is to use Doubly Linked List (DLL). We can delete middle element in O(1) time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

Following is implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the second middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 3.

- C++

- C
- Java

- Python3
- C#

```
/* C++ Program to implement a stack
that supports findMiddle() and
deleteMiddle in O(1) time */
#include <bits/stdc++.h>
using namespace std;

/* A Doubly Linked List Node */
class DLLNode {
public:
	DLLNode* prev;
	int data;
	DLLNode* next;
};

/* Representation of the stack data structure
that supports findMiddle() in O(1) time.
The Stack is implemented using Doubly Linked List.
It maintains pointer to head node, pointer to
middle node and count of nodes */
class myStack {
public:
	DLLNode* head;
	DLLNode* mid;
	int count;
};

/* Function to create the stack data structure */
myStack* createMyStack()
{
	myStack* ms = new myStack();
	ms->count = 0;
	return ms;
};

/* Function to push an element to the stack */
void push(myStack* ms, int new_data)
{
	/* allocate DLLNode and put in data */
	DLLNode* new_DLLNode = new DLLNode();
	new_DLLNode->data = new_data;

	/* Since we are adding at the beginning,
	prev is always NULL */
	new_DLLNode->prev = NULL;
```

```
        /* link the old list off the new DLLNode */
        new_DLLNode->next = ms->head;

        /* Increment count of items in stack */
        ms->count += 1;

        /* Change mid pointer in two cases
        1) Linked List is empty
        2) Number of nodes in linked list is odd */
        if (ms->count == 1) {
                ms->mid = new_DLLNode;
        }
        else {
                ms->head->prev = new_DLLNode;

                if (!(ms->count
                        & 1)) // Update mid if ms->count is even
                    ms->mid = ms->mid->prev;
        }

        /* move head to point to the new DLLNode */
        ms->head = new_DLLNode;
}

/* Function to pop an element from stack */
int pop(myStack* ms)
{
        /* Stack underflow */
        if (ms->count == 0) {
                cout << "Stack is empty\n";
                return -1;
        }

        DLLNode* head = ms->head;
        int item = head->data;
        ms->head = head->next;

        // If linked list doesn't
        // become empty, update prev
        // of new head as NULL
        if (ms->head != NULL)
                ms->head->prev = NULL;

        ms->count -= 1;

        // update the mid pointer when
        // we have odd number of
        // elements in the stack, i,e
        // move down the mid pointer.
```

```
        if ((ms->count) & 1)
                ms->mid = ms->mid->next;

        free(head);

        return item;
}

// Function for finding middle of the stack
int findMiddle(myStack* ms)
{
        if (ms->count == 0) {
                cout << "Stack is empty now\n";
                return -1;
        }

        return ms->mid->data;

}
// Function for deleting middle of the stack
int  deletemiddle(myStack* ms)   // IMPROVED BY Sohaib Ayub
          {
                    int temp=ms->mid->data;
                ms->mid->prev->next=ms->mid->next;
                ms->mid->next->prev=ms->mid->prev->next;

                delete ms->mid;
                    ms->mid = ms->mid->next; //So that mid does not contain garbage value
                return temp;
          }




// Driver code
int main()
{
        /* Let us create a stack using push() operation*/
        myStack* ms = createMyStack();
        push(ms, 11);
        push(ms, 22);
        push(ms, 33);
        push(ms, 44);
        push(ms, 55);
        push(ms, 66);
        push(ms, 77);

        cout << "Item popped is " << pop(ms) << endl;
```

```
        cout << "Item popped is " << pop(ms) << endl;
        cout << "Item popped is " << pop(ms) << endl;
        cout << "Middle Element is " << findMiddle(ms) << endl;
        cout << "Deleted Middle Element is "<<deletemiddle(ms)<<endl;
        cout << "Middle Element is " << findMiddle(ms) << endl;
        return 0;
}

// This code is contributed by rathbhupendra
```

Output

Item popped is 77

Item popped is 66

Item popped is 55

Middle Element is 33

Deleted Middle Element is 33

Middle Element is 22

Design and Implement Special Stack Data Structure | Added Space Optimized Version

- Difficulty Level : **Medium**
- Last Updated : 07 Jun, 2021

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, . etc.

Example:

Consider the following SpecialStack

16 --> TOP

15

29

19

18

When getMin() is called it should

return 15, which is the minimum

element in the current stack.

If we do pop two times on stack,

the stack becomes

29 --> TOP

19

18

When getMin() is called, it should

return 18 which is the minimum in

the current stack.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Solution: Use two stacks: one to store actual stack elements and the other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of the auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

1) push x to the first stack (the stack with actual elements)

2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.

…..a) If x is smaller than y then push x to the auxiliary stack.

…..b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

1) pop the top element from the auxiliary stack.

2) pop the top element from the actual stack and return it.

Step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

1) Return the top element of the auxiliary stack.

We can see that all the above operations are O(1).

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15, and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack

18 <--- top

Auxiliary Stack

18 <---- top

When 19 is inserted, both stacks change to following.

Actual Stack

19 <--- top

18

Auxiliary Stack

18 <---- top

18

When 29 is inserted, both stacks change to following.

Actual Stack

29 <--- top

19

18

Auxiliary Stack

18 <---- top

18

18

When 15 is inserted, both stacks change to following.

Actual Stack

15 <--- top

29

19

18

Auxiliary Stack

15 <---- top

18

18

18

When 16 is inserted, both stacks change to following.

Actual Stack

16 <--- top

15

29

19

18

Auxiliary Stack

15 <---- top

15

18

18

18

The following is the implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object *min* which works as an auxiliary stack.

- C++
- Java
- Python3

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

```
using namespace std;

/* A simple stack class with
basic stack funtionalities */
class Stack {
private:
	static const int max = 100;
	int arr[max];
	int top;

public:
	Stack() { top = -1; }
	bool isEmpty();
	bool isFull();
	int pop();
	void push(int x);
};

/* Stack's member method to check
if the stack is empty */
bool Stack::isEmpty()
{
	if (top == -1)
		return true;
	return false;
}

/* Stack's member method to check
if the stack is full */
bool Stack::isFull()
{
	if (top == max - 1)
		return true;
	return false;
}

/* Stack's member method to remove
an element from it */
int Stack::pop()
{
	if (isEmpty()) {
		cout << "Stack Underflow";
		abort();
	}
	int x = arr[top];
	top--;
	return x;
}

/* Stack's member method to insert
an element to it */
```

```
void Stack::push(int x)
{
        if (isFull()) {
                cout << "Stack Overflow";
                abort();
        }
        top++;
        arr[top] = x;
}

/* A class that supports all the stack
operations and one additional
    operation getMin() that returns the
minimum element from stack at
    any time.  This class inherits from
the stack class and uses an
    auxiliary stack that holds minimum
elements */
class SpecialStack : public Stack {
        Stack min;

public:
        int pop();
        void push(int x);
        int getMin();
};

/* SpecialStack's member method to insert
  an element to it. This method
      makes sure that the min stack is also
updated with appropriate minimum
      values */
void SpecialStack::push(int x)
{
        if (isEmpty() == true) {
                Stack::push(x);
                min.push(x);
        }
        else {
                Stack::push(x);
                int y = min.pop();
                min.push(y);
                if (x < y)
                        min.push(x);
                else
                        min.push(y);
        }
}

/* SpecialStack's member method to
```

```
remove an element from it. This method
      removes top element from min stack also. */
int SpecialStack::pop()
{
        int x = Stack::pop();
        min.pop();
        return x;
}

/* SpecialStack's member method to get
  minimum element from it. */
int SpecialStack::getMin()
{
        int x = min.pop();
        min.push(x);
        return x;
}

/* Driver program to test SpecialStack
  methods */
int main()
{
        SpecialStack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout << s.getMin() << endl;
        s.push(5);
        cout << s.getMin();
        return 0;
}
```

Output

10

5

Complexity Analysis:

- Time Complexity:
    1. For insert operation: O(1) (As insertion ‘push’ in a stack takes constant time)
    2. For delete operation: O(1) (As deletion ‘pop’ in a stack takes constant time)
    3. For ‘Get Min’ operation: O(1) (As we have used an auxiliary stack which has it’s top as the minimum element)

- Auxiliary Space: O(n).

Use of auxiliary stack for storing values.

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in the auxiliary stack. We can push only when the incoming element of the main stack is smaller than or equal to the top of the auxiliary stack. Similarly during pop, if the pop-off element equal to the top of the auxiliary stack, remove the top element of the auxiliary stack. Following is the modified implementation of push() and pop().

- C++

- Java

```
/* SpecialStack's member method to
       insert an element to it. This method
       makes sure that the min stack is
       also updated with appropriate minimum
       values */
void SpecialStack::push(int x)
{
        if (isEmpty() == true) {
                Stack::push(x);
                min.push(x);
        }
        else {
                Stack::push(x);
                int y = min.pop();
                min.push(y);

                /* push only when the incoming element
                      of main stack is smaller
                than or equal to top of auxiliary stack */
                if (x <= y)
                        min.push(x);
        }
}

/* SpecialStack's member method to
       remove an element from it. This method
       removes top element from min stack also. */
int SpecialStack::pop()
{
        int x = Stack::pop();
```

```
        int y = min.pop();

        /* Push the popped element y back
           only if it is not equal to x */
        if (y != x)
            min.push(y);

        return x;
}
```

Complexity Analysis:

- Time Complexity:
  1. For Insert operation: O(1) (As insertion ‘push’ in a stack takes constant time)
  2. For Delete operation: O(1) (As deletion ‘pop’ in a stack takes constant time)
  3. For ‘Get Min’ operation: O(1) (As we have used an auxiliary which has it’s top as the minimum element)
- Auxiliary Space: O(n).

The complexity in the worst case is the same as above but in other cases, it will take slightly less space than the above approach as repetition is neglected.

Further optimized O(1) time complexity and O(1) space complexity solution :

The above approach can be optimized further and the solution can be made to work in O(1) time complexity and O(1) space complexity. The idea is to store min element found till current insertion) along with all the elements as a reminder of a DUMMY_VALUE, and the actual element as a multiple of the DUMMY_VALUE.

For example, while pushing an element ‘e’ into the stack, store it as (e * DUMMY_VALUE + minFoundSoFar), this way we know what was the minimum value present in the stack at the time ‘e’ was being inserted.

To pop the actual value just return e/DUMMY_VALUE and set the new minimum as (minFoundSoFar % DUMMY_VALUE).

Note: Following method will fail if we try to insert DUMMY_VALUE in the stack, so we have to make our selection of DUMMY_VALUE carefully.

Let’s say the following elements are being inserted in the stack – 3 2 6 1 8 5

d is dummy value.

s is wrapper stack

top is top element of the stack

min is the minimum value at that instant when the elements were inserted/removed

The following steps shows the current state of the above variables at any instant –

1. *s.push(3);*

min=3 //updated min as stack here is empty

*s = {3*d + 3}*

*top = (3*d + 3)/d = 3*

2. *s.push(2);*

min = 2 //updated min as min > current element

*s = {3*d + 3-> 2*d + 2}*

*top = (2*d + 2)/d = 2*

3. *s.push(6);*

min = 2

*s = {3*d + 3-> 2*d + 2-> 6*d + 2}*

*top = (6*d + 2)/d = 6*

4. *s.push(1);*

min = 1 //updated min as min > current element

*s = {3*d + 3-> 2*d + 2-> 6*d + 2 -> 1*d + 1}*

*top = (1*d + 1)/d = 1*

5. *s.push(8);*

min = 1

*s = {3*d + 3-> 2*d + 2-> 6*d + 2 -> 1*d + 1 -> 8*d + 1}*

*top = (8*d + 1)/d = 8*

6. *s.push(5);*

min = 1

*s = {3*d + 3-> 2*d + 2-> 6*d + 2 -> 1*d + 1 -> 8*d + 1 -> 5*d + 1}*

*top = (5*d + 1)/d = 5*

7. *s.pop();*

*s = {3*d + 3 -> 2*d + 2 -> 6*d + 2 -> 1*d + 1 -> 8*d + 1 -> 5*d + 1}*

*top = (5*d + 1)/d = 5*

*min = (8*d + 1)%d = 1 // min is always remainder of the second top element in stack.*

8. s.pop();

*s = {3*d + 3 -> 2*d + 2-> 6*d + 2 -> 1*d + 1 -> 8*d + 1}*

*top = (8*d + 1)/d = 8*

*min = (1*d + 1)%d = 1*

9. s.pop()

*s = {3*d + 3 -> 2*d + 2-> 6*d + 2 -> 1*d + 1}*

*top = (1*d + 1)/d = 1*

*min = (6*d + 2)%d = 2*

10. s.pop()

*s = {3*d + 3-> 2*d + 2-> 6*d + 2}*

*top = (6*d + 2)/d = 6*

*min = (2*d + 2)%d = 2*

11. s.pop()

*s = {3*d + 3-> 2*d + 2}*

*top = (2*d + 2)/d = 2*

*min = (3*d + 3)%d = 3*

12. s.pop()

*s = {3*d + 3}*

*top = (3*d + 3)/d = 3*

min = -1 // since stack is now empty

- C++
- Java

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

/* A special stack having peek() pop() and
  * push() along with additional getMin() that
  * returns minimum value in a stack without
  * using extra space and all operations in O(1)
  * time.. ???? */
class SpecialStack
{

        // Sentinel value for min
        int min = -1;

        // DEMO_VALUE
        static const int demoVal = 9999;
        stack<int> st;

public:

        void getMin()
        {
                cout << "min is: " << min << endl;
        }

        void push(int val)
        {

                // If stack is empty OR current element
                // is less than min, update min.
```

```
        if (st.empty() || val < min)
        {
            min = val;
        }

        // Encode the current value with
        // demoVal, combine with min and
        // insert into stack
        st.push(val * demoVal + min);
        cout << "pushed: " << val << endl;
    }

    int pop()
    {
        // if stack is empty return -1;
        if ( st.empty() ) {
            cout << "stack underflow" << endl ;
            return -1;
        }

        int val = st.top();
        st.pop();

        // If stack is empty, there would
        // be no min value present, so
        // make min as -1
        if (!st.empty())
            min = st.top() % demoVal;
        else
            min = -1;

        cout << "popped: " << val / demoVal << endl;

        // Decode actual value from
        // encoded value
        return val / demoVal;
    }

    int peek()
    {

        // Decode actual value
        // from encoded value
        return st.top() / demoVal;
    }
};

// Driver Code
```

```
int main()
{
	SpecialStack s;

	vector<int> arr = { 3, 2, 6, 1, 8, 5, 5, 5, 5 };

	for(int i = 0; i < arr.size(); i++)
	{
		s.push(arr[i]);
		s.getMin();
	}

	cout << endl;
	for(int i = 0; i < arr.size(); i++)
	{
		s.pop();
		s.getMin();
	}
	return 0;
}

// This code is contributed by scisaif
```

Output

```
pushed: 3

min is: 3

pushed: 2

min is: 2

pushed: 6

min is: 2

pushed: 1

min is: 1

pushed: 8

min is: 1

pushed: 5
```

min is: 1

pushed: 5

min is: 1

pushed: 5

min is: 1

pushed: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 5

min is: 1

popped: 8

min is: 1

popped: 1

min is: 2

popped: 6

min is: 2

popped: 2

min is: 3

popped: 3

min is: -1

PROJECT

Note: This assignment is used to assess the required outcomes for the course, as outlined in the course syllabus. These outcomes are:

i. use of arrays and pointers in the solution of programming problems using C++
ii. create and use classes within the C++ programming language
iii. create, compile, and execute C++ programs within the Unix environment, using the Object-Oriented design model
iv. program using C++ techniques: composition of objects, operator overloads, dynamic memory allocation, inheritance and polymorphism, and file I/O
v. program using C++ techniques: composition of objects, templates, preprocessor directives, and basic data structures.

These will be assessed using the following rubric:

Rubric for Outcome v.		I	E	H	
v.	Templates, Basic Data Structures, Preprocessor Directives, and Composition	-	-	-	Key: I = ineffective E = effective H = highly effective

In order to earn a course grade of C- or better, the assessment must result in Effective or Highly Effective for each outcome.

Educational Objectives: After completing this assignment the student should have the following knowledge, ability, and skills:

- Define the concept of Generic Container
- Define the ADT Stack with elements of type T and maximum size N
- Give examples of the use of Stack in computing
- Describe an implementation plan for generic Stack as a class template `Stack<T,N>` based on an array of `T` objects
- Code the implementation for `Stack<T,N>` using the plan
- Define the ADT Queue with elements of type T
- Give examples of the use of Queue in computing
- Describe an implementation plan for generic Queue as a class template `Queue<T>` based on dynamically allocated links containing `T` objects
- Code the implementation for `Stack<T,N>` using the plan
- Describe an implementation plan for `Stack<T,N>` based on dynamically allocated links containing `T` objects

- Explain why it is impractical to implement `Queue<T>` using an array

Operational Objectives: Create two generic container classes `fsu::TStack<T,N>` and `fsu::TQueue<T>` that satisfy the interface requirements given below, along with appropriate test harnesses for these classes.

Deliverables: Four files `tstack.h`, `tqueue.h`, `ftstack.cpp`, and `ftqueue.cpp`.

Extra Credit: Enable copy for TStack and TQueue (copy constructor and assignment operator) to receive up to 20 points extra credit on the assignments side of the course.

Abstract Data Types

An *abstract data type*, abbreviated ADT, consists of three things:

1. A set of elements of some type `T`
2. Operations that may modify the set or return values associated with the set
3. Rules of behavior, or axioms, that determine how the operations interact

The operations and axioms together should determine a unique character for the ADT, so that any two implementations should be essentially equivalent. (The word *isomorphic* is used to give precision to "essentially equivalent". We'll look at this in the next course.)

Stacks and Queues

The *stack* and *queue* are ADTs that are used in many applications and have roots that pre-date the invention of high-level languages. Conceptually, stack and queue are sets of data that can be expanded, contracted, and accessed using very specific operations. The stack models the "LIFO", or last-in, first-out, rule, and the queue models the "FIFO", or first-in, first-out rule. The actual names for the stack and queue operations may vary somewhat from one description to another, but the behavior of the abstract stack and queue operations is well known and unambiguously understood throughout computer science. Stacks and queues are important in many aspects of computing, ranging from hardware design and I/O to algorithm control structures.

Typical uses of ADT Stack are (1) runtime environment for modern programming languages (facilitating recursive function calls, among other things), (2) control of the depth first search and backtracking search algorithms, (3) hardware evaluation of postfix expressions, and (4) various compiler operations, such as converting expressions from infix to postfix.

Typical uses of ADT Queue are (1) buffers, without which computer communication would be impossible, (2) control of algorithms such as breadth first search, and (3) simulation modelling of systems as diverse as manufacturing facilities, customer service, and computer operating systems.

Abstract Stack Interface

The stack abstraction has the following operations and behavior:

- **`Push(t)`** Inserts the element **`t`** into the stack
- **`Pop()`** Removes the last-inserted element; undefined when stack is empty
- **`Top()`** Returns the last-inserted element; undefined when stack is empty
- **`Empty()`** Returns true iff the stack has no elements
- **`Size()`** Returns the number of elements in the stack

Abstract Queue Interface

The queue abstraction has the following operations and behavior:

- **`Push(t)`** Inserts the element **`t`** into the queue
- **`Pop()`** Removes the first-inserted element; undefined when queue is empty
- **`Front()`** Returns the first-inserted element; undefined when queue is empty
- **`Empty()`** Returns true iff the queue has no elements
- **`Size()`** Returns the number of elements in the queue

Application: Converting Infix to Postfix Notation

As one example of the use of stacks and queues in computing, consider the following function that illustrates an algorithm for converting arithmetic expressions from infix to postfix notation:

```
...
#include <tqueue.h>
#include <tstack.h>
...
typedef fsu::TQueue < Token > TokenQueue;
typedef fsu::TStack < Token > TokenStack;
// a Token is either an operand, an operator, or a left or right
parenthesis
...
bool i2p (TokenQueue & Q1, TokenQueue & Q2)
// converts infix expression in Q1 to postfix expression in Q2
// returns true on success, false if syntax error is encountered
{
   Token L( '(' ), R( ')' );  // left and right parentheses
   TokenStack S;              // algorithm control stack
   Q2.Clear();                // make sure ouput queue is empty
   while (!Q1.Empty())
   {
      if (Q1.Front() == L) // next Token is '('
      // push '(' to mark beginning of a parenthesized expression
      {
         S.Push(Q1.Front());
         Q1.Pop();
      }
      else if (Q1.Front().IsOperator()) // next Token is an operator
      {
         // pop previous operators of equal or higher precedence to
output
         while (!S.Empty() && S.Top() >= Q1.Front())
         {
            Q2.Push(S.Top());
            S.Pop();
         }
         // then push new operator onto stack
         S.Push(Q1.Front());
         Q1.Pop();
      }
      else if (Q1.Front() == R) // next Token is ')'
      // regurgitate operators for the parenthesized expression
      {
         while (!S.Empty() && !(S.Top() == L))
         {
            Q2.Push(S.Top());
            S.Pop();
         }
         if (S.Empty())      // unbalanced parentheses
         {
            std::cout << "** error: too many right parens\n";
            return false;
         }
         S.Pop();            // discard '('
         Q1.Pop();           // discard ')'
      }
      else                   // next Token should be an operand
      // send operand directly to output
```

```
        {
            Q2.Push(Q1.Front());
            Q1.Pop();
        }
    }  // end while()
    // regurgitate remaining operators
    while (!S.Empty())
    {
        if (S.Top() == L)      // unbalanced parentheses
        {
            std::cout << "** error: too many left parens\n";
            return false;
        }
        Q2.Push(S.Top());
        S.Pop();
    }
    return true;
}  // end i2p()
```

This is a complex algorithm, but not beyond your capability to understand. Notice how the algorithm takes into account the different levels of precedence among operators and the possibility of parenthetical sub-expressions. Things to make special note of are:

- A typedef statement is used to define the types `TokenQueue` and `TokenStack` as a queue or stack of tokens; this serves both programmer convenience and readability of the program.
- Function arguments of type `TokenQueue` are used as buffers to pass an expression in to and out from the function.
- A locally declared variable of type `TokenStack` is used as the principal control structure for the function.

The stack is used to store the operators of parenthetical subexpressions as well as operators of one precedence level pending discovery of an operator of lower precedence. This function is distributed as part of the file `in2post.cpp`.

Use the distributed executable `in2post.x` to experiment so that you understand the roles Stack and Queue play in the algorithm.

TStack Implementation Plan

We will implement the stack abstraction as a C++ class template

```
template < typename T , size_t N >
TStack;
```

with the following public methods:

```
// TStack < T , N > API
void     Push     (const T& t); // push t onto stack; error if full
T        Pop      ();           // pop stack and return removed element;
error if stack is empty
T&       Top      ();           // return top element of stack; error if
stack is empty
const T& Top      () const;     // const version
size_t   Size     () const;     // return number of elements in stack
size_t   Capacity () const;     // return storage capacity [maximum size]
of stack
int      Empty    () const;     // return 1/true if stack is empty,
0/false if not empty
void     Clear    ();           // make the stack empty
void     Display  (std::ostream& os, char ofc = '\0') const; // output
contents through os
```

The element and size data will be maintained in private variables:

```
const size_t capacity_;  // = N = size of array   - fixed during life of
stack
T            data_[N];   // array of T objects    - where T objects are
stored
size_t       size_;      // current size of stack - dynamic during life
of stack
```

The class constructor will have responsibility for initializing variables. Note that `capacity_` is a constant, so it must be initialized by the constructor in the initialization list and it cannot be changed during the life of a stack object; `capacity_` should be given the value passed in as the second template argument `N`. Because all class variables are declared at compile time, the destructor will have no responsibilities. Values stored in the `data_` array and the `size_` variable will be correctly maintained by the push and pop operations, using the "upper index" end of the data as the top of the stack. The data in the stack should always be the array elements in the range `[0..size_)`, and the element `data_[size_ - 1]` is the top of the stack (assuming `size_ > 0`). Copy will be disabled.

Please note that the `data_` array is automatically allocated at compile time and remains allocated during the lifetime of the object. It is implicitly de-allocated just like any statically declared array, when it goes out of scope. Thus the underlying "footprint" of the stack object remains fixed as the size changes, even when the size is changed to zero. There should be no calls to operators `new` or `delete` in this implementation.

This implementation will have the requirement on clients that the maximum size required for the stack is known in advance and determined by the second template argument - see requirements below.

TQueue Implementation Plan

We will implement the queue abstraction as a C++ class template `TQueue` with the following public methods:

```
// TQueue < T > API
void      Push    (const T& t); // push t onto queue
T         Pop     ();           // pop queue and return removed element;
error if queue is empty
T&        Front   ();           // return front element of queue; error
if queue is empty
const T&  Front   () const;     // const version
size_t    Size    () const;     // return number of elements in queue
int       Empty   () const;     // return 1 if queue is empty, 0 if not
empty
void      Clear   ();           // make the queue empty
void      Display (std::ostream& os, char ofc = '\0') const; // output
contents through os
```

Unlike Stack, Queue requires access to data at both the front and back, which makes an array implementation impractical. We will use a linked list implementation using a link class defined as follows:

```
class Link
{
  Link ( const T& t );  // 1-parameter constructor
  T      element_;
  Link * nextLink_;
  friend class TQueue<T>;
};
```

Note that all members of class `Link` are private, which means a `Link` object can be created or accessed only inside an implementation of its friend class `TQueue<T>`. The only method for class `Link` is its constructor, whose implementation should just initialize the two variables. (This can be done inside the class definition, as shown below.)

The private `TQueue` variables for this implementation will be exactly two pointers to links, the first and last links created. Including the definition of the helper class `Link`, the private section of the class definition should be like the following (with implementor-chosen private variable names):

```
template < typename T >
class TQueue
{
  public:
  ...

  private:
    class Link
    {
      Link ( const T& t ) : element_(t), nextLink_(0) {}
      T      element_;
      Link * nextLink_;
      friend class TQueue<T>;
    };
  Link * firstLink_;
  Link * lastLink_;
};
```

The class constructor will have responsibility for initializing the two variables to zero. These two pointers will be zero exactly when there is no data in the queue (the queue is empty). Links for data will be allocated as needed by `Push()` and de-allocated by `Pop()`. These operations will also need to re-direct appropriate link pointers in the dynamically allocated links, and maintain the class variables `firstLink_` and `lastLink_` correctly pointing to the links containing the first and last elements of the queue. The destructor should de-allocate all remaining dynamically allocated links in the queue. The Size() method will have to loop through the links to count them, whereas the Empty() method can do a simple check for emptiness of the queue. Copy will be disabled.

Because this implementation relies on dynamically allocated memory, the container makes no restrictions on the client program as to anticipated maximum size of the queue.

Procedural Requirements

1. Create and work within a separate subdirectory `cop3330/proj2`. Review the COP 3330 rules found in Introduction/Work Rules.

2. Begin by copying the following files from the course directory `[LIB]` into your `proj2` directory:

```
proj2/in2post.cpp
proj2/proj2submit.sh
area51/in2post_s.x
area51/in2post_i.x
area51/ftstack_i.x
area51/ftstack_s.x
area51/ftqueue_i.x
area51/ftqueue_s.x
```

The naming of these files uses the convention that `_s` are compiled for Sun/Solaris and `_i` are compiled for Intel/Linux. Use one of the sample client executables to experiment to get an understanding of how your program should behave.

3. Define and implement the class template `fsu::TStack<T,N>` in the file `tstack.h`

4. Devise a test client for `TStack<T,N>` that exercises the Stack interface for at least one native type and one user-defined type `T`. Repair your code as necessary. Put this test client in the file `ftstack.cpp`.

5. Define and implement the class template `fsu::TQueue<T>` in the file `tqueue.h`

6. Devise a test client for `TQueue<T>` that exercises the Queue interface for at least one native type and one user-defined type `T`. Put this test client in the file `ftqueue.cpp`.

7. Test `TStack` and `TQueue` using the supplied application `in2post.cpp`. Again, make sure behavior is appropriate and make corrections if necessary.

8. Turn in `tstack.h`, `tqueue.h`, `ftstack.cpp`, and `ftqueue.cpp` using the `proj2submit.sh` submit script.

Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

Code Requirements and Specifications

1. Both `TStack` and `TQueue` should be a proper type, with copy protection. That is, they should have (1) a public default constructor, (2) a public destructor, (3) a private copy constructor and (4) a private assignment operator. The copy constructor and assignment operator should have prototypes in the class definition but should not be implemented.

2. The `TStack` constructor should create a stack that is empty but has the capacity to hold `N` elements, where `N` is the second template parameter with type `size_t`. Note that this parameter should be given the default value of 100. This has the effect of making a declaration such as

```
fsu::TStack<int> s;
```

legal and create a stack with capacity 100.

3. The `TQueue` constructor should create an empty queue with no dynamic memory allocation.

4. The `TQueue<T>::Push(t)` operation must dynamically allocate memory required for storing `t` in the queue. Similarly, the `TQueue<T>::Pop()` operation must de-allocate memory used to store the element being removed from the queue.

5. As always, the class destructors should de-allocate all dynamic memory still owned by the object. The stack and queue implementations will be very different.

6. Use the implementation plans discussed above. No methods or variables should be added to these classes, beyond those specified above and in the implementation plans.

7. The `Display(os, ofc)` methods are intended to regurgitate the contents out through the `std::ostream` object `os`. The second parameter `ofc` is a single output formatting character that has the default value `'\0'`. (The other three popular choices for `ofc` are `' '`, `'\t'` and `'\n'`.) The implementation of `Display` must recognize two cases:

   i. `ofc == '\0'`: send the contents to output with nothing between them
   ii. `ofc != '\0'`: send the contents to output with `ofc` separating them

Thus, for example, `S.Display(std::cout)` would send the contents of `S` to standard output.

8. The output operator should be overloaded as follows:

```
template < typename T , size_t N >
std::ostream& operator << (std::ostream& os, const TStack<T,N>& S)
{
  S.Display (os, '\0');
  return os;
}

template < typename T >
std::ostream& operator << (std::ostream& os, const TQueue<T>& S)
{
  S.Display (os, '\0');
  return os;
}
```

The overload of `operator <<()` should be placed in your stack/queue header file immediately following the class definition.

9. The classes `TStack` amd `TQueue` should be in the `fsu` namespace.

10. The files `tstack.h` and `tqueue.h` should be protected against multiple reads using the `#ifndef ... #define ... #endif` mechanism.

11. The test client programs `ftstack.cpp` and `ftqueue.cpp` should adequately test the functionality of stack and queue, respectively, including the output operator. It is your responsibility to create these test programs and to use them for actual testing of your stack and queue data structures.

Hints

- Your test clients can be modelled on the harness `fstring.cpp` distributed as part of a previous assignment. Testing stack and queue will be much simpler, however, because:

  i. We have disabled copying objects. Therefore, you need only one object in the functionality test (where `fstring.cpp` used three).
  ii. The API for TStack and TQueue is simpler than that of String.

- It is recommended that you carry the stack portion of the project through to completion, including implementation and testing, before starting on the queue portion. The implementation plan for `TStack` uses design and methodology that you already have experience with. The implementation plan for `TQueue` uses design and methodology that is very different and new.

- Keep in mind that the implementations of class template methods are in themselves template functions. For example, the implementation of the `TStack` method `Pop()` would look something like this:

```
template < typename T , size_t N >
T TStack<T,N>::Pop()
{
  // yada dada
  return ??;
}
```

and the `TQueue` method `Pop()` would look something like this:

```
template < typename T >
T TQueue<T>::Pop()
{
  // yada dada
  return ??;
}
```

- We will test your implementations using (1) your supplied test clients, (2) `in2post`, and (3) test clients of our own design.

- There are two versions of `TStack::Top()` and `TQueue::Front()`. These are distunguished by "`const`" modifiers for one of the versions. The implementation code is identical for each version. The main point is that "`Top()`" can be called on a constant stack, but the returned reference may not be used to modify the top element. This nuance will be tested in our assessment. You can test it with two functions such as:

```
char ShowTop(const fsu::TStack<char>& s)
{
  return s.Top();
}

void ChangeTop(fsu::TStack<char>& s, char newTop)
{
  s.Top() = newTop;
}
```

Note that `ShowTop` has a `const` reference to a stack, so would be able to call the const version of `Top()` but not the non-const version, but that suffices. `ChangeTop` would need to call the non-const version in order to change the value at the top of the stack. A simple test named "`constTest.cpp`" is posted in the distribution directory.