

EE 441 Data Structures

Chapter 4: Stacks and Queues

Serkan SARITAŞ

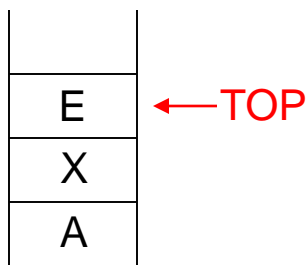
Uğur HALICI

Klaus Werner SCHMIDT

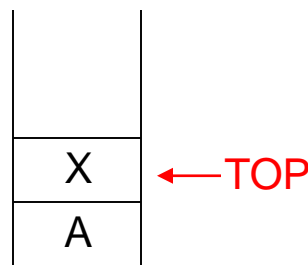
Stacks

- A *stack* is a data structure consisting of a list of items and a *pointer* to the "top" item in the list.
- Pointer: Logical not necessarily implemented as a real pointer
- Items can be inserted or removed from the list only at the top, i.e., the list is ordered in the sequence of entry of items.
- Insertions and removals proceed in the "LIFO" last-in-first-out order.

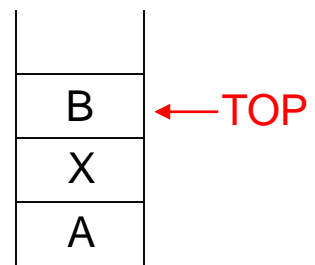
e.g.



remove:
"POP"



insert 'B':
PUSH 'B'



Where do we use stacks?

- *A restricted data structure*
- Stack elements have a natural order:
 - Elements are removed from the stack in the reverse order to the order of their addition
 - The lower elements are those that have been on the stack the longest
- Use of stacks in computer hardware:
 - Allocating and accessing memory.
 - Saving and reloading context registers on procedure calls
- Use of stacks in algorithms:
 - Backtracking: During path finding if you reach a dead-end, you go back in reverse steps as you came until you reach another junction



Stack Class Declaration for storing Characters

```
# include <iostream>
# include <stdlib>
```

```
const int MaxStackSize=50;
```

```
class Stack
{
```

```
    private:
```

```
        char stacklist[MaxStackSize];
        int top;
```

```
    public:
```

```
        Stack(void); // constructor to initialize top
```

```
        //modification operations
```

```
        void Push(const char& item);
```

```
        char Pop(void);
```

```
        void ClearStack(void);
```

```
        //just copy top item without modifying stack contents
```

```
        char Peek(void) const;
```

```
        //check stack state
```

```
        int StackEmpty(void) const;
```

```
        int StackFull(void) const;
```

```
};
```

- This Stack class is specialized for *character stacks*
- If we need to have a Stack class for integers, we need to re-declare it



Generalizing Data Types:

typedef

```
# include <iostream>
# include <stdlib>
```

```
typedef char DataType;
const int MaxStackSize=50;
class Stack
{
    private:
        DataType stacklist[MaxStackSize];
        int top;

    public:
        Stack(void); // constructor to initialize top
        //modification operations
        void Push(const DataType& item);
        DataType Pop(void);
        void ClearStack(void);
        //just copy top item without modifying stack contents
        DataType Peek(void) const;
        //check stack state
        int StackEmpty(void) const;
        int StackFull(void) const;
};
```

- This Stack class is more generalized.
- Replace *DataType* with *char* during compile time to have a *character* stack.
- Replace *DataType* with *Date* during compile time to have a *Date* stack.
- Still only one type (but can be any type) is possible after compilation.



Generalizing Data Types: What do we want?

- We want to use the same class or function definition for different types of items
 - Create different objects within the same class but with different data types
 - Define a general function that works on different data types
 - Link the data type with the object or function call and not with the whole program
 - A single function or class definition gives rise to a family of functions or classes that are compiled from the same source code, but operate on different types.



Generalizing Data Types: Templates

- Examples:

```
Stack<int> S1;
```

```
Stack<float> S2;
```

- `template <class T1, class T2, ... class Tn>`
- T_1, T_2, \dots, T_n are classes that will be used with a specific class upon creation of an object
- Note here that any operation in the template class or function must be defined for any possible data types in the template



Template Functions

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
}
```

```
int main () {
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

<http://www.cplusplus.com/doc/oldtutorial/templates/>





What you learn is useful 😊

[Data Structures](#) [Algorithms](#) [Interview Preparation](#) [Topic-wise Practice](#) [C++](#) [Java](#) [Python](#) [Competitive Programming](#) [Machine Learning](#)

[Amazon Interview | Set 18](#)
[Amazon Interview | Set 19](#)
[Amazon Interview | Set 20](#)
[Amazon Interview | Set 21](#)
[Amazon Interview | Set 22](#)
[Amazon Interview Questions](#)
[Program for Fibonacci numbers](#)
[Program for nth Catalan Number](#)
[Bell Numbers \(Number of ways to Partition a Set\)](#)
[Binomial Coefficient | DP-9](#)

Amazon Interview Questions

Difficulty Level : Easy • Last Updated : 07 Jul, 2021

[Read](#) [Discuss](#)  

Most Asked Questions

1. [K largest elements from a big file or array.](#)
2. [Find a triplet a, b, c such that \$a^2 = b^2 + c^2\$.](#) Variations of this problem like find a triplet with sum equal to 0. Find a pair with given sum. All such questions are efficiently solved using hashing. – Practice [here](#)
3. Binary [tree traversal questions](#) like left view, right view, top view, bottom view, maximum of a level, minimum of a level, children sum property, diameter etc.
4. [Convert a Binary tree to DLL](#) – Practice [here](#)
5. [Lowest Common ancestor in a Binary Search Tree and Binary Tree.](#)
6. [Implement a stack with push\(\), pop\(\) and min\(\) in O\(1\) time.](#)
7. [Reverse a linked list in groups of size k](#) – Practice [here](#)



A general Stack Class Declaration

```
const int MaxStackSize=50;
template <class T>
class Stack
{
    private:
        T stacklist[MaxStackSize];
        int top;

    public:
        Stack(void); // constructor to initialize top
        //modification operations
        void Push(const T& item);
        T Pop(void);
        void ClearStack(void);
        //just copy top item without modifying stack contents
        T Peek(void) const;
        //check stack state returns top element value without removal
        int StackEmpty(void) const;
        //returns true if the stack is empty
        int StackFull(void) const;
        //returns true if the stack is full
};
```



General Stack Class Implementation

```
template <class T>
Stack<T>::Stack(void):top(-1)
{}

template <class T>
//Push
void Stack<T>::Push(const T& item)//
{
    //can not push if stack has exceeded its limits
    if (top==MaxStackSize-1)
    {
        cerr<<"Stack overflow"<<endl;
        exit(1);
    }
    // increment top and copy item into list
    top++;
    stacklist[top] = item;
}
```

```
template <class T>
//pop
T Stack<T>::Pop(void)
{
    T temp;
    // is stack empty nothing to pop
    if (top== -1)
    {
        cerr<<"Stack empty"<<endl;
        exit(1);
    }
    //record the top element
    temp=stacklist[top] ;
    //decrement top and return the earlier top element
    top--;
    return temp;
}
```



General Stack Class Implementation

```
template <class T>
//Peek is the same as Pop, except top is not moved
T Stack<T>::Peek(void) const
{ //just copy top item without modifying stack contents
    T temp;
    // is stack empty nothing to peek
    if (top==-1)
    {
        cerr<<"Stack empty"<<endl;
        exit(1);
    }
    //record the top element
    temp=stacklist[top] ;

    return temp;
}
```

```
template <class T>
//clear stack
void Stack<T>::ClearStack(void)
{
    top = -1;
}

template <class T>
//StackEmpty: returns true if the stack is empty
int Stack<T>::StackEmpty(void) const
{
    return top==-1;
}

template <class T>
//StackFull: returns true if the stack is full
int Stack<T>::StackFull(void) const
{
    return top==(MaxStackSize-1);
}
```



Example

- Example: Write a program that uses a stack to test a given character string and decides if the string is a palindrome (i.e. reads the same backwards and forwards, e.g. "kabak", " a man a plan a canal panama", etc.)
- Algorithm:
 - first get rid of all blanks in the string
 - push the whole string character-wise, into a stack
 - pop out characters one by one, comparing with the characters of the original de-blanked string
 - Example: top spot



Solution

NULL: Pointer does not point at a valid object.
Replaced by \0.
For strings: used as termination character

Assuming that the stack declaration and implementation are in "Stack.hpp"

```
#include "Stack.hpp"
#include <iostream>
using namespace std;

void Deblank(char *s, char *t)
{
    while (*s!='\0')
    {
        if (*s!=' ')
        {
            *t=*s;
            t++;
        }
        s++;
    }
    *t='\0'; //append NULL to new string
}
```

```
int main()
{
    // create stack object to store string in reverse order.

    Stack<char> S;
    char palstring[80], deblankedstring[80], c;
    int i=0; // string pointer
    bool ispalindrome=true; //we'll stop if false

    // get input
    cout << "Enter a string to check if it is a palindrome: " << endl;
    cin.getline(palstring,80,'\n');
    //remove blanks
    Deblank(palstring,deblankedstring);
    i=0;
    while(deblankedstring[i] != '\0')
    {
        S.Push(deblankedstring[i]);
        i++;
    }
```



Solution

//now pop one-by-one comparing with original

```
i=0;
while (!S.StackEmpty())
{
    c=S.Pop();
    //get out of loop when first nonmatch
    if (c!=deblankedstring[i])
    {
        ispalindrome=false;
        break;
    }
    // continue until the end of string
    i++;
}

//operation finished. Printout result
if (ispalindrome)
    cout<<"\\ "<<palstring<<"\\ "<<"is a palindrome"<<endl;
else
    cout<<"\\ "<<palstring<<"\\ "<<"is not a palindrome"<<endl;

return 0;
}
```

```
Enter a string to check if it is a palindrome:
a man a plan a canal panama
\a man a plan a canal panama\is a palindrome
```

```
Enter a string to check if it is a palindrome:
ee441
\ee441\is not a palindrome
```



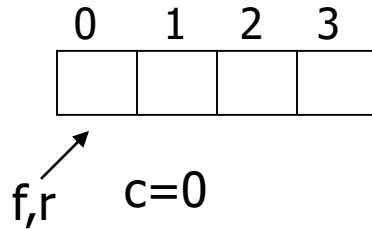
Queues

- A queue is a data structure that stores elements in a list and permits data access only at the two ends of the list.
- An element is inserted only at the “rear” end of the list and deleted from only the “front” end of the list.
- Elements are removed in the same order in which they are stored and hence a queue provides FIFO(first-in/first-out) or FCFS(first-come/first-served) ordering

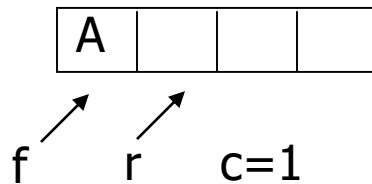


Queues

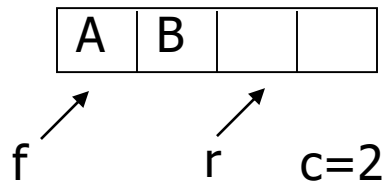
1. initially: count=0, front=0, rear=0



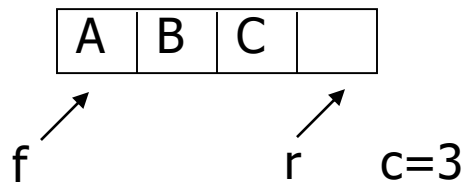
2. insert 'A':



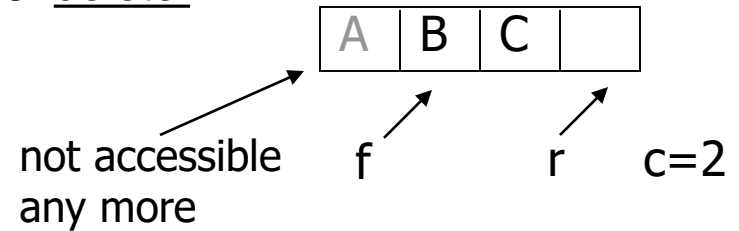
3. insert 'B':



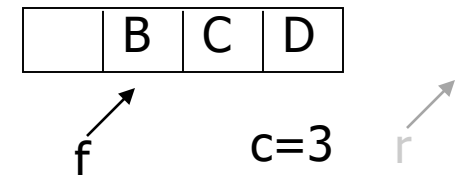
4. insert 'C':



5. delete:



6. insert 'D':



We have more space in the linear storage but we cannot use it



Queues

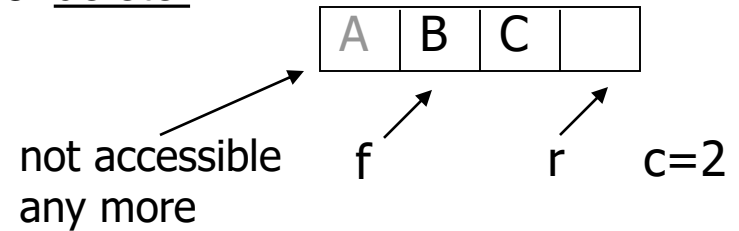
CIRCULAR OPERATION

i.e., move rear & front forward:

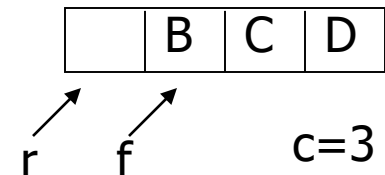
$\text{rear} = (\text{rear} + 1) \% \text{MaxQSize}$

$\text{front} = (\text{front} + 1) \% \text{MaxQSize}$ etc.

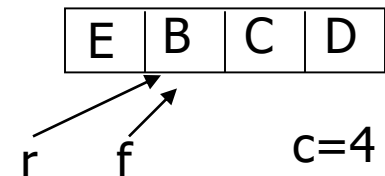
5. delete:



6. insert 'D':



6. insert 'E':



How to check if the queue is full?

Where do we use queues?

- *A restricted data structure*
- Queue elements have a natural order:
 - Elements are removed from the queue in the order of their addition
 - The elements closer to the front are those that have been in the queue the longest
- Use of queues in computers:
 - Managing access to limited resources:
 - Operating system processes → access to hardware resources
 - Computer networks → access to bandwidth



A general Queue Class Declaration

```
# include <iostream>
# include <stdlib>

const int MaxQSize=50;

template <class T>
class Queue
{
private:
    // queue array and its parameters
    int front, rear, count;
    T qlist[MaxQSize];

public:
    //constructor
    Queue(void); // initialize data members

    //queue modification operations
    void QInsert(const T& item);
    T QDelete(void);
    void ClearQueue(void);

    // queue access
    T QFront(void) const;

    // queue test methods
    int QLength(void) const;
    int QEmpty(void) const;
    int QFull(void) const;
};
```



A general Queue Class Implementation (circular)

```
//initialize queue front, rear, count
template <class T>
Queue<T>::Queue(void): front(0), rear (0), count(0)
{}

// QInsert: insert item into the queue
template <class T>
void Queue<T>::QInsert(const T& item)
{
    // terminate if queue is full
    if (count==MaxQSize)
    {
        cerr<<"Queue overflow!" <<endl;
        exit(1)
    }
    //increment count, assign item to qlist and update rear
    count++;
    qlist[rear] =item;
    rear=(rear+1)%MaxQSize;
}
```

```
//QDelete : delete element from the queue and return its value
template <class T>
T Queue<T>::QDelete(void)
{
    T temp;
    // if qlist is empty, terminate the program
    if (count==0)
    {
        cerr<<"Deleting from an empty queue!"<<endl;
    }
    //record value at the front of the queue
    temp=qlist[front];
    //decrement count, advance front and return former front
    count--;
    front=(front+1) % MaxQSize;
    return temp;
}

//clear queue
template <class T>
void Queue<T>::ClearQueue(void)
{
    front=0;rear=0;count=0;
}
```



A general Queue Class Implementation (circular)

```
//Queue Access
template <class T>
T Queue<T>::QFront(void) const
{
    T temp;
    // if qlist is empty, terminate the program
    if (count==0)
    {
        cerr<<"Accessing to an empty queue!"<<endl;
    }
    //record value at the front of the queue
    temp=qlist[front];
    return temp;
}
```

//Queue Test Methods

```
template <class T>
int Queue<T>::QLength(void) const
{
    return count;
}

template <class T>
int Queue<T>::QEmpty(void) const
{
    return count==0;
}

template <class T>
int Queue<T>::QFull(void) const
{
    return count==MaxQSize;
}
```



Example

- Write a function

`void findinQ (int key, Queue _____MyQ)`

which searches a given key in a Queue and deletes it if it exists. If the key does not exist the function ends after the search without performing an operation on the queue. You can only use stacks and at most one temporary variable in this function



Example: Solution

- Pass by reference

```
void findinQ (int key, Queue <int>&MyQ)
```

- Idea: We want to check each item in the queue
 1. Take them out one by one to look at them
 2. Store them in some stack not to lose them
 3. Put them back in the queue
- Problem: Storing in the stack changes the order of items



Example: Solution

```
void findinQ(int key, Queue <int>& MyQ)
{
    Stack<int> S1;
    Stack<int> S2;
    int temp;
    while(!MyQ.QEmpty())
    {
        temp=MyQ.QDelete();
        if(temp!=key)
            S1.Push(temp);
    }
    while(!S1.StackEmpty())
        S2.Push(S1.Pop());

    while(!S2.StackEmpty())
        MyQ.QInsert(S2.Pop());
}
```



```
Enter the # of integers you will provide as queue: 7
Enter the integer #1: 4
Enter the integer #2: 6
Enter the integer #3: 3
Enter the integer #4: 1
Enter the integer #5: 5
Enter the integer #6: 8
Enter the integer #7: 9
BEGINNING
The queue content is: 4 6 3 1 5 8 9
Enter the key which will be searched and deleted if it exists: 7
BEFORE SEARCH/DELETE
The queue content is: 4 6 3 1 5 8 9
AFTER SEARCH/DELETE
The queue content is: 4 6 3 1 5 8 9
```

```
Enter the # of integers you will provide as queue: 7
Enter the integer #1: 4
Enter the integer #2: 6
Enter the integer #3: 3
Enter the integer #4: 1
Enter the integer #5: 5
Enter the integer #6: 8
Enter the integer #7: 9
BEGINNING
The queue content is: 4 6 3 1 5 8 9
Enter the key which will be searched and deleted if it exists: 3
BEFORE SEARCH/DELETE
The queue content is: 4 6 3 1 5 8 9
AFTER SEARCH/DELETE
The queue content is: 4 6 1 5 8 9
```



Some Specific C++ notations:

Compound assignment

```
arraySum(int ia[ ], int sz)
{
    int sum=0;
    for (int i=0; i<sz; ++i)
        sum+=ia[i];
    return sum;
}
```

sum+=ia[i];

is the same as

sum=sum+ia[i];



Some Specific C++ notations:

Compound assignment

- General syntax:

`a op = b;`

meaning

`a = a op b;`

- Examples:

- `i*=j;` is the same as `i=i*j;`

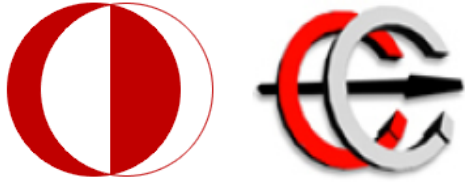
- `ia[i]*=(b+c/d);` is the same as `ia[i]=ia[i]*(b+c/d);`



Increment and decrement operators in the prefix and postfix forms

- Prefix: **++i**
 - Example: `array[++i]=some_value;`
 - prefix form increments `i` before using that value as index
- Postfix: **i++**
 - Example: `array [i++]=some_value;`
 - postfix form increments `i` after using that value as index
- **i++** is the same as **++i** on its own





EE 441 Data Structures

Lecture 4: Stacks and Queues
