

Chapter 3

Stacks and Queues

Array Implementation

3.1 Stacks

- A stack is a sequence of items, which can be added and removed from one end only. Stack is a concept known to everyone. We make stacks of books, plates and many other things.
- What goes into a stack first, comes out last and what goes in last comes out first. Hence a stack is often known as LAST-IN-FIRST-OUT (LIFO)
- A stack is very useful and popular data structure. It is used by the operating system and other system programs extensively. Whenever nested function calls are made, the programs keep track of the location to return to by storing it in stacks. Complex arithmetic expressions can also be checked and calculated easily.
- Example:

| | | | |
|-----------------------------------|------------|---------|---------|
| Consider the following situation: | | | |
| - | - | - | 31 |
| 23 | - | 18 | 18 |
| 45 | 45 | 45 | 45 |
| 16 | 16 | 16 | 16 |
| 37 | 37 | 37 | 37 |
| | Remove(23) | Add(18) | Add(31) |

- The processes for adding and removing from a stack are called **PUSHING** and **POPPING** respectively.

- The allowable operations that define a structure as being a stack are

NSH/HAR/RA

1/18

- ✓ Create an empty stack
- ✓ Determine whether the stack is empty or not
- ✓ Determine whether the stack is full or not
- ✓ Find the size of the stack (how many items are in it)
- ✓ PUSH a new entry onto the top of the stack providing it is not full
- ✓ Copy the top entry from the stack providing it is not empty
- ✓ POP the entry off the top of the stack providing it is not empty
- ✓ Clear the stack to make it empty
- ✓ Traverse the stack performing a given operation on each entry

3.2 Stack Implementation

- A stack can be implemented easily using an array and a integer that holds the position of the top element.
- Example:

Consider the following:

| | Element | Value |
|-----------------------|---------|-------|
| | 5 | - |
| | 4 | - |
| Here you need to | 3 | 23 |
| image the memory | 2 | 45 |
| elements are numbered | 1 | 16 |
| from 0 to 5 upwards | 0 | 37 |

- Suppose the array is called “**stack**”. In addition we need something to point to the top of the stack (not a pointer variable). Suppose the pointer we called “**topstack**” then at the present moment topstack would have the value 3 as this is the element number that contains the value at the top of the stack.
- “**topstack**” should have the value -1 initially

- POPing from the stack means assigning the value at the top of the stack to another variable (?) and decrementing the stack pointer.
- Example:

```
? = stack[topstack]
topstack - -
```

- PUSHing would require incrementing the stack pointer and putting an element onto the top of the stack.
- Example:

```
topstack + +
stack[topstack] = ?
```

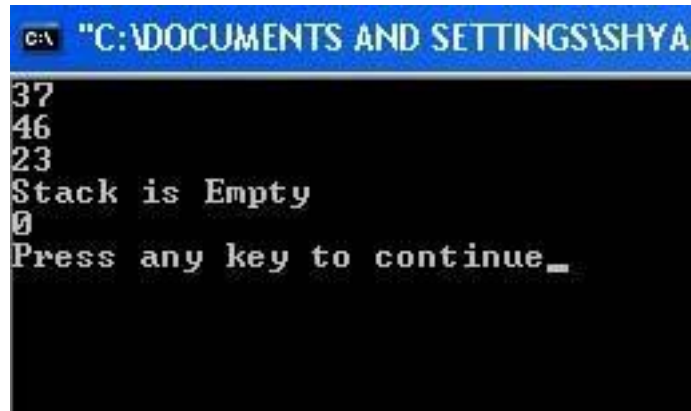
3.3 Declaring a Stack ADT

- It is better to incorporate the requisites from a stack in a single class

```
#include<iostream.h>
class ADTstack
{
private:
    int stack[10];
    int topstack;
public:
    ADTstack()
    {
        topstack = -1;
    }

    int empty()
    {
        if(topstack==-1)
            return 1;
        else
            return 0;
    }
}
```

```
int full()
{
    if(topstack==9)
        return 1;
    else
        return 0;
}
void push(int num)
{
    if (!full())
    {
        topstack++;
        stack[topstack] = num;
    }
    else
    {
        cout<<"Stack is Full"<<endl;
    }
}
int pop()
{
    int num;
    if(!empty())
    {
        num = stack[topstack];
        topstack--;
        return num;
    }
    else
    {
        cout<<"Stack is Empty"<<endl;
        return 0;
    }
}
};
void main()
{
    ADTstack st;
    st.push(23);
    st.push(46);
    st.push(37);
    cout<<st.pop()<<endl;
    cout<<st.pop()<<endl;
    cout<<st.pop()<<endl;
    cout<<st.pop()<<endl;
}
```



```
C:\DOCUMENTS AND SETTINGS\SHYAM
37
46
23
Stack is Empty
0
Press any key to continue_
```

3.4 Stack Application

Parsing and Evaluating Arithmetic Expression

Infix, Postfix and Prefix notation

1. An arithmetic expression can be represented in three different formats: **infix**, **postfix**, **prefix**.
2. In **infix** notation, the operator comes **between** the two operands, the basic format of the algebraic notation we learned in high school. (e.g. : $a + b$)
3. In **postfix** notation, also call **reverse Polish notation (RPN)**, the operator comes after its two operands. (e.g.: $a b +$)
4. In **Prefix** notation, the operator comes before the two operands. (e.g. : $+ a b$)
5. Using postfix and prefix notation eliminates the need for parentheses, because the operator is placed directly after the two operands to which it applies.

Converting Infix To Postfix : Manual Transformation

The rules for manually converting infix to postfix expressions are as follows:

1. Fully parenthesize the expression using any explicit parentheses and the arithmetic precedence- multiply and divide before add and subtract.
2. Change all infix notations in each parenthesis to postfix notation, starting from the innermost expressions. Conversion to postfix notation is done by moving the operator to the location of the expression's closing parenthesis.
3. Remove all parentheses.

For example, for the following infix expression:

$$A + B * C$$

Step 1 results in

$$(A + (B * C))$$

Step 2 removes the multiply operator after C

$$(A + B C *)$$

and then moves the addition operator to between the last two closing parentheses. This change is made because the closing parenthesis for the plus is the last parenthesis. We now have

$$(A (B C *) +)$$

Finally, Step 3 removes the parentheses.

$$A B C * +$$

Let's look at more complex example. This example is not only longer, but it already has one set of parentheses to override the default evaluation order.

$$(A + B) * C + D + E * F - G$$

$$(((((A + B) * C) + D) + (E * F)) - G)$$

Step 2 then removes the operators.

$$(((((A B +) C *) D +) (E F *) +) G -)$$

Step 3 removes the parentheses.

$$A B + C * D E F * + G -$$

Converting Infix To Postfix : Algorithmic Transformation

We have to know the rules of operator precedence in order to convert infix to postfix. The operations + and - have the same precedence. Multiplication and division, which we will represent as * and / also have equal precedence, but both have higher precedence than + and -. These are the same rules you learned in high school.

The expression is processed according to the following rules:

- Variables (in this case letters) are copied to the output
- Left parentheses are always pushed onto the stack
- When a right parenthesis is encountered, the symbol at the top of the stack is popped off the stack and copied to the output. Repeat *until* the symbol at the top

of the stack is a left parenthesis. When that occurs, both parentheses are discarded.

- Otherwise, if the symbol being scanned has a higher precedence than the symbol at the top of the stack, the symbol being scanned is pushed onto the stack.
- If the precedence of the symbol being scanned is lower than or equal to the precedence of the symbol at the top of the stack, one element of the stack is popped to the output. Instead, the symbol being scanned will be compared with the new top element on the stack.
- When the terminating symbol is reached on the input scan, the stack is popped to the output until the terminating symbol is also reached on the stack. Then the algorithm terminates.

Algorithm

```

for (each character ch in the infix expression)
{
    switch (ch)
    {
        case operand:
            postfixExp = postfixExp + ch;
            break;
        case '(':
            aStack.push(ch);
            break;
        case ')':
            while (top of stack is not '(' )
            {
                postfixExp = postfixExp + (top of Stack)
                aStack.pop();
            }
            aStack.pop();

            break;

        case operator:
            while (!aStack.isEmpty() and top of Stack is not
                '(' and precedence (ch) <=
precedence(top of aStack)
            {
                postfixExp = postfixExp + (top of Stack)
                aStack.pop();
            }

            aStack.push(ch)
    }
}

```

```

        break
    } //end of switch
} //end of for

```

Example 1. convert $A + B - C$ to postfix expression

| Ch | Stack (bottom to top) | postfixExp | |
|----|-----------------------|------------|--|
| A | | A | |
| + | + | A | |
| B | + | A B | |
| – | | A B + | precedence (ch) <= precedence(top of aStack) |
| | – | A B + | |
| C | – | A B + C | |
| | | A B + C – | |

Example 2. convert $A - (B + C * D)$ to postfix expression

| Ch | Stack (bottom to top) | postfixExp | |
|----|-----------------------|------------|--|
| A | | A | |
| – | – | A | |
| (| (– | A | |
| B | (– | A B | |
| + | + (– | A B | |
| C | + (– | A B C | |
| * | * + (– | A B C | |

| | | | |
|----------|--|----------------------|---|
| D | * + (– | A B C D | |
|) | + (– | A B C D * | Move operators from stack to postfixExp until ‘(|
| | (– | A B C D * + | |
| | – | A B C D * + | |
| | | A B C D * + – | |

Note: *If you are familiar with the method of conversion, you can actually skip the above steps and convert directly infix expression to postfix expressions.*

Exercise: Test yourself

Convert $(A + B) / (C - D)$ to postfix expression

Answer: $A B + C D - /$

Evaluating Postfix Notation

Evaluating an expression in postfix notation is trivially easy if you use a stack. The postfix expression to be evaluated is scanned from left to right. Variables or constants are pushed onto the stack. When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.

The expression scanned from left to right. The digits in the expression are pushed onto the stack and the operators are performed.

Algorithm

```

for (each character ch in the string)
{
    if (ch is an operand)
        push value that operand ch represents onto stack

    else // ch is an operator named op
    {
        operand2 = top of stack
        pop the stack

        operand1 = top of stack
        pop the stack

        result = operand1 + operand 2
        push result onto stack
    } //end of if
} //end of for

```

Example 3. postfix expression: 2 3 4 + *

| Key entered | Action | After stack operation Stack (bottom to top) |
|-------------|--------|---|
| 2 | Push 2 | 2 |
| 3 | Push 3 | 3 2 |
| 4 | Push 4 | 4 3 2 |
| | | |

| | | |
|---|--|-----------|
| + | Operand2 = pop stack (4) | 3 |
| | | 2 |
| | Operand1 = pop stack (3) | 2 |
| | Result = operand1 + operand2 (7) | 2 |
| | Push result | 7 |
| | | 2 |
| | | |
| * | Operand2 = pop stack (7) | 2 |
| | Operand1 = pop stack (2) | |
| | Result = operand1 * operand2 (14) | |
| | Push result | 14 |

Exercise: Test Yourself

Evaluate the following postfix expression:

- i) 2 4 6 + * (answer: 20)
 ii) 9 7 + 5 3 - / (answer: 8)

3.5 Queues

- A queue is a sequence of items, to which new items can be added at one end (tail) and only removed from the other end (head).
- Queue is a concept known to everyone. We queue up in banks. Shops and many other places. What goes into the queue first comes out first, and what goes in last
- Example:

Consider the following queue:

| | | | |
|------|----|----|------|
| 23 | 46 | 18 | 54 |
| Head | | | Tail |

- Addition to the queue would result in:

| | | | | |
|------|----|----|----|-----------|
| 23 | 46 | 18 | 54 | 26 |
| Head | | | | Tail |

- Serving the queue would result in:

| | | | |
|------|----|----|------|
| 46 | 18 | 54 | 26 |
| Head | | | Tail |

- The allowable operations that can be performed on a queue are:
 - ✓ Create an empty queue leaving it empty
 - ✓ Determine whether the queue is empty or not
 - ✓ Determine whether the queue is full or not
 - ✓ Find the size of the queue (how many items are in it)
 - ✓ Append an entry to the tail of the queue providing it is not full
 - ✓ Retrieve (but do not remove) the front entry of the queue providing it is not empty
 - ✓ Serve (and remove) the front entry of the queue providing it is not empty
 - ✓ Clear the queue to make it empty
 - ✓ Traverse the queue performing a given operation on each entry

3.6 Queue Implementation

- A queue can be implemented easily using an array and two integers to indicate the position of the tail and the head of the queue.
- Example:

| | | | | | | |
|-------------------------|----|----|----|----|---|---|
| Consider the following: | | | | | | |
| Element | 0 | 1 | 2 | 3 | 4 | 5 |
| Value | 23 | 46 | 18 | 54 | | |

- Suppose the array is called “**queue**”. We need a pointer to the head and a pointer to the tail. Suppose these were called “**head**” and “**tail**” respectively.
- “**tail**” is initialised to -1 and “**head**” is initialised to 0.
- With respect the example above, to append an element, means to go into element 4

```
tail + +  
queue[tail] = ?
```

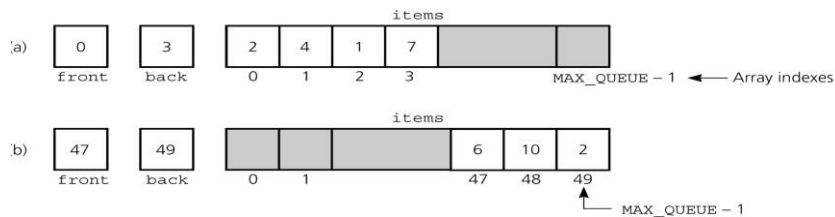
- With respect the example above, to serve would mean removing element from element 0

```
? = queue[head] head  
+ +
```

- **WARNING!!! THIS QUEUE WOULD NOT LAST LONG!!!** As the queue moves down, the storage space at the beginning of the array is discarded and never used again.
- A better implementation, but more expensive time wise, would be to move all the elements up one when one element has been served. In this way the head would always be in element 0.
- Appending is still as before but serving would be

```
? = queue[head]  
for(i=0; i<tail; i++)  
{ queue[i] = queue[i+1]  
}
```

3.7 Queue : An Array Based Implementation



1) Figure

- (a) A naive array-based implementation of a queue;
- (b) rightward drift can cause the queue to appear full

Here *front* and *back* are the indexes of the front and back items. *front* = 0 and *back* = -1.

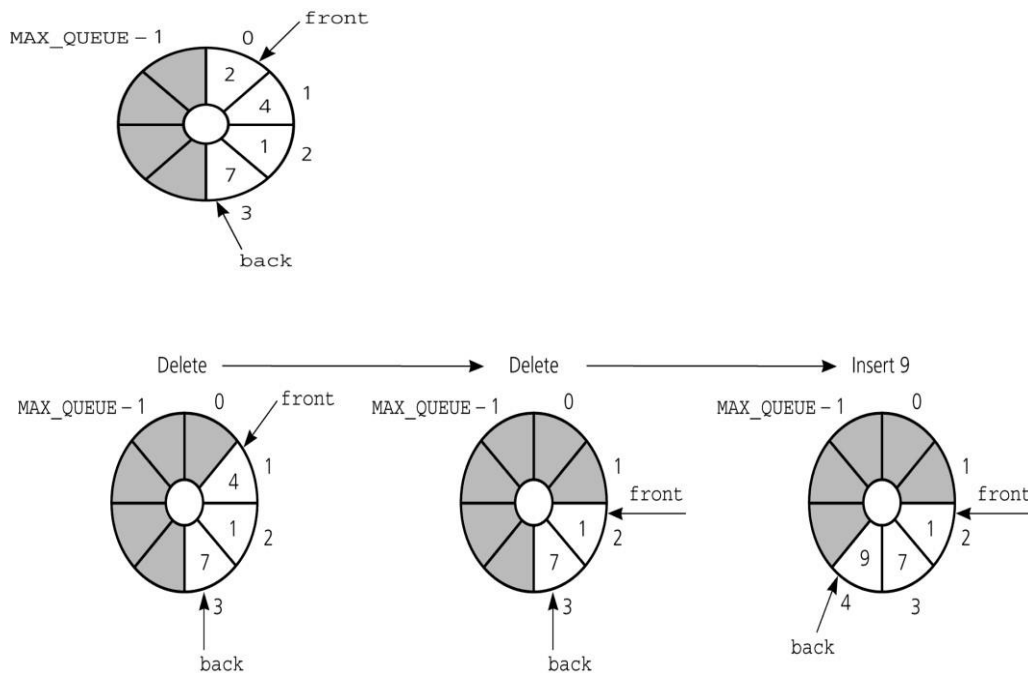
- i) insert a new item into the queue, you increment *back* and place the item in *items[back]*.
- ii) delete an item, you simply increment *front*.

The queue is empty whenever *back* is less than *front*. The queue is full when *back* equals *MAX_QUEUE - 1*.

2) The problem with this strategy is *rightward drift* – after a sequence of additions and removals, the items in the queue will drift toward the end of the array, and *back* could equal *MAX_QUEUE - 1* even when the queue contains only a few items. Figure (b) illustrates this situation.

3) One possible solution to this problem is to shift array elements to the left, either after each deletion or whenever *back* equals *MAX_QUEUE - 1*. This solution guarantees that the queue can always contain up to *MAX_QUEUE* items. Shifting is not really satisfactory, however, as it would dominate the cost of the implementation.

4) A much more elegant solution is possible by viewing the array as circular. Advance the queue indexes *front* (to delete an item) and *back* (to insert an item) by moving them clockwise around the array



5) Figure above illustrate the effect of a sequence of three queue operations on front , back , and the array. When either front or back advances past $\text{MAX_QUEUE} - 1$, it wraps around to 0. This wraparound eliminates the problem of rightward drift, which occurred in the previous implementation, because here the circular array has no end.

The only difficulty with this scheme involves detecting the queue-empty and queue-full conditions.

6) It seems reasonable to select as the queue-empty condition front is one slot ahead of back since this appears to indicate that front “passes” back when the queue becomes empty, as Figure (a) depicts. However, it is also possible that this condition signals a full queue: Because the queue is circular, back might in fact “catch up” with front as the queue becomes full; Figure (b) illustrates this situation.

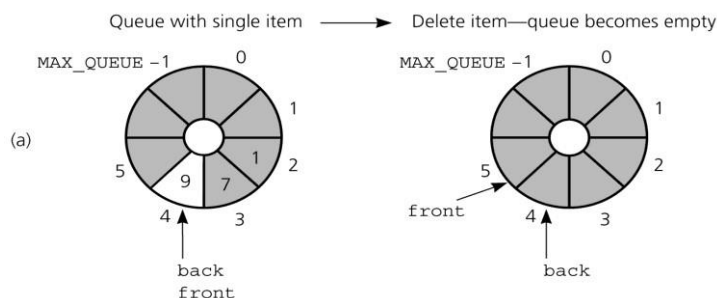


Figure (a) front passes back when the queue becomes empty

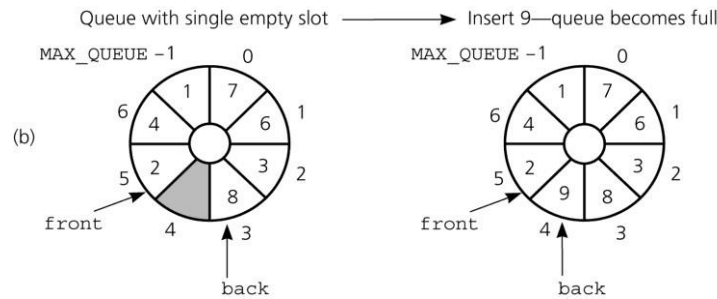


Figure (b) back catches up to front when the queue becomes full

7) Obviously, you need a way to distinguish between the two situations. One such way is to keep a count of the number of items in the queue.

i) Before inserting into the queue, you check to see if the count is equal to *MAX_QUEUE*: if it is, the queue is full.

ii) Before deleting an item from the queue, you check to see if the count is *zero*; if it is, the queue is empty.

8) To initialize the queue, you set *front* to 0, *back* to *MAX_QUEUE - 1*, and *count* to 0. You obtain the wraparound effect of a circular queue by using modulo arithmetic (that is, the C++ % operator) when incrementing *front* and *back*.

For example, you can insert *newItem* into the queue by using the statements

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

Notice that if *back* equaled *MAX_QUEUE - 1* before the insertion of *newItem*, the first statement, *back = (back+1) % MAX_QUEUE*, would have the effect of wrapping *back* around to location 0.

Similarly, you can delete the item at the front of the queue by using the statements

```
front = (front+1) % MAX_QUEUE;
--count;
```


3.8 Declaring a Queue ADT

Again it is best to incorporate all the requisites for a queue into a class

```
#include<iostream.h>
class ADTqueue
{
    private:
        int queue[10];
        int head,tail;

    public:
        ADTqueue()
        {
            tail = -1;
            head = 0;
        }

        int empty()
        {
            if(head == tail+1)
                return 1;
        }
}
```

```
        else
            return 0;
        }

        int full()
        {
            if(tail == 9)
                return 1;
            else
                return 0;
        }

        void append(int num)
        {
            if (!full())
            {
                tail++;
                queue[tail] = num;
            }
            else
            {
                cout<<"Queue is
Full"<<endl;
            }
        }

        int serve()
        {
            int num;
            if(!empty())
            {
                num = queue[head];
                head++;
            }
            return num;
        }
    }
```

```
        else
        {
            cout<<"Queue is Empty"<<endl;
return 0;
        }
```

```
    }  
}; void  
main()  
{  
    ADTQueue q;  
    q.append(23);  
    q.append (46);  
    q.append (37);  
    cout<<q.serve()<<endl;  
    cout<<q.serve()<<endl;  
    cout<<q.serve()<<endl;  
    cout<<q.serve()<<endl;  
}
```



```
C:\> "C:\DOCUMENTS AND SETTINGS\SHYAMA"  
23  
46  
37  
Queue is Empty  
Press any key to continue_
```