Module 2

Stacks and Queues: Abstract Data Types

A **stack** is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the *top* of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called ***Last-in-First-out (LIFO)***. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.

PRIMITIVE STACK OPERATIONS: PUSH AND POP

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

ARRAY AND LINKED IMPLEMENTATION OF STACK

Stack can be implemented in two ways:
1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (*i.e.*,increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (*i.e.*, memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity. The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:
(*a*) Display “The stack is in overflow condition”

(*b*) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

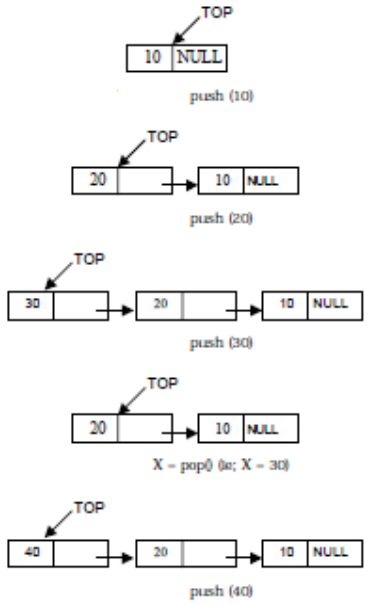
Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then
(*a*) Display "The Stack is empty"
(*b*) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit

STACK USING LINKED LIST

Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.



ALGORITHM FOR PUSH OPERATION

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode -> DATA = DATA
4. NewNode – > Next = TOP
5. TOP = NewNode
6. Exit

ALGORITHM FOR POP OPERATION

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
(*a*) Display "The stack is empty"
2. Else
(*a*) TEMP = TOP
(*b*) Display "The popped element TOP –> DATA"
(*c*) TOP = TOP –> Next
(*d*) TEMP –> Next = NULL
(*e*) Free the TEMP node
3. Exit

APPLICATIONS OF STACKS

There are a number of applications of stacks; three of them are discussed briefly. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

1) PREFIX AND POSTFIX EXPRESSIONS

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)
1. Infix notation
2. Prefix notation
3. Postfix notation

The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example : The expression to add two numbers A and B is written in infix notation as:
A + B
Note that the operator '+' is written in between the operands A and B.

The *prefix notation* is a notation in which the operator(s) is written before the operands, it is also called polish notation in the honor of the polish mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:
+ A B
As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:
A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:
add(A, B). Note that the operator *add* (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression(due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor).

Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. In the preceding sections we will study the conversion of the expression from one notation to other.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in *infix* notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS, and associativity. Using infix notation, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence
Exponential operator ^ Highest precedence
Multiplication/Division *, / Next precedence
Addition/Subtraction +, - Least precedence

CONVERTING INFIX TO POSTFIX EXPRESSION

The method of converting infix expression A + B * C to postfix form is:
A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:
1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression
B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Give postfix form for A + [(B + C) + (D + E) * F] / G
Solution. Evaluation order is
A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE + F *] / G}
A + { [(BC + (DE + F * +] / G} .

A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Algorithm

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Push "(" onto stack, and add")" to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator is encountered, then:
(*a*) Repeatedly pop from stack and add P each operator (on the top of stack),
which has the same precedence as, or higher precedence than .
(*b*) Add to stack.
6. If a right parenthesis is encountered, then:
(*a*) Repeatedly pop from stack and add to P (on the top of stack until a left
parenthesis is encountered.
(*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.

EVALUATION OF POSTFIX EXPRESSION

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator is encountered, then:
(*a*) Remove the two top elements of STACK, where A is the top element and B is
the next-to-top element.
(*b*) Evaluate B A.
(*c*) Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.

2) RECURSION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.
2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1
(initiate a recursive call).
3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.

Each time a function call to itself is executed, the prologue of the function saves necessary information required for its proper functioning. The Last-in-First-Out characteristics of a recursive function points that the stack is the most obvious data structure to implement the recursive function. Programs compiled in modern

high-level languages (even C) make use of a stack for the procedure or function invocation in memory. When any procedure or function is called, a number of words (such as variables, return address and other arguments and its data(s) for future use) are pushed onto the program stack. When the procedure or function returns, this frame of data is popped off the stack.
As a function calls a (may be or may not be another) function, its arguments, return address and local variables are pushed onto the stack. Since each function runs in its own environment or context, it becomes possible for a function to call itself — a technique known as *recursion*. This capability is extremely useful and extensively used — because many problems are elegantly specified or solved in a recursive way. The stack is a region of main memory within which programs temporarily store data as they execute. For example, when a program sends parameters to a function, the parameters are placed on the stack. When the function completes its execution these parameters are popped off from the stack. When a function calls other function the current contents (ie., variables) of the caller function are pushed onto the stack with the address of the instruction just next to the call instruction, this is done so after the execution of called function, the compiler can backtrack (or remember) the path from where it is sent to the called function.

Disadvantages of Recursion

1. It consumes more storage space because the recursive calls along with automatic variables are stored on the stack.
2. The computer may run out of memory if the recursive calls are not checked.
3. It is not more efficient in terms of speed and execution time.
4. According to some computer professionals, recursion does not offer any concrete advantage over non-recursive procedures/functions.
5. If proper precautions are not taken, recursion may result in non-terminating iterations.
6. Recursion is not advocated when the problem can be through iteration. Recursion may be treated as a software tool to be applied carefully and selectively.

3) TOWER OF HANOI PROBLEM

So far we have discussed the comparative definition and disadvantages of recursion with examples. Now let us look at the Tower of Hanoi problem and see how we can use recursive technique to produce a logical and elegant solution. The initial setup of the problem is shown below. Here three pegs (or towers) X, Y and Z exists. There will be four different sized disks, say A, B, C and D. Each disk has a hole in the center so that it can be stacked on any of the pegs. At the beginning, the disks are stacked on the X peg, that is the largest sized disk on the bottom and the smallest sized disk on top as shown in Fig. below

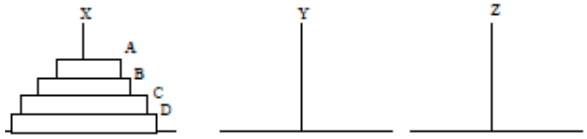
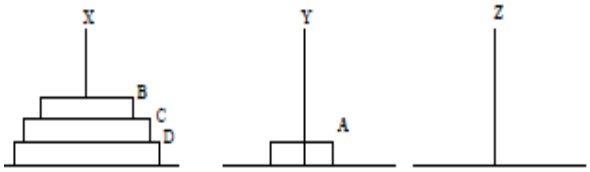


Fig: Initial Position of Towers of Hanoi

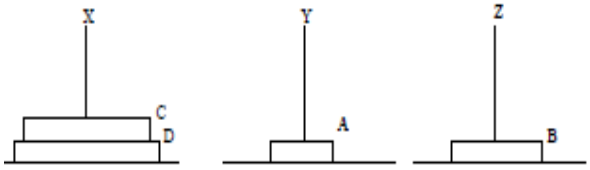
Here we have to transfer all the disks from source peg X to the destination peg Z by using an intermediate peg Y. Following are the rules to be followed during transfer :

1. Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
2. Only one disk may be moved at a time.
3. Each disk must be stacked on any one of the pegs.

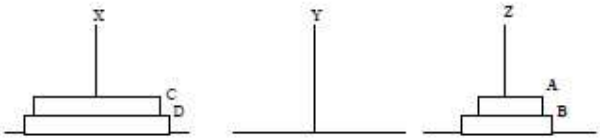
Now Tower of Hanoi problem can be solved as shown below :



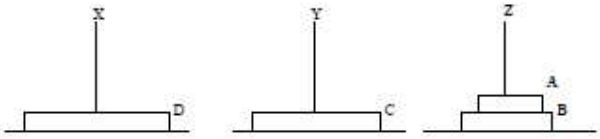
Move disk A from the peg X to peg Y



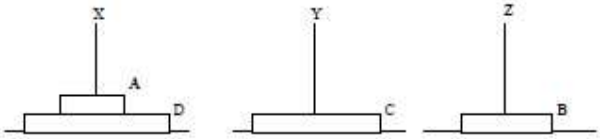
Move disk B from the peg X to peg Z



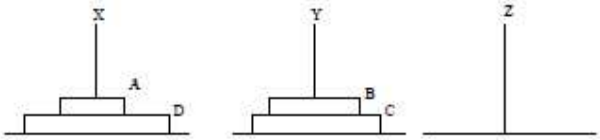
Move disk A from the peg Y to peg Z



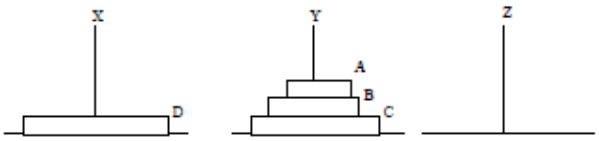
Move disk C from the peg X to peg Y



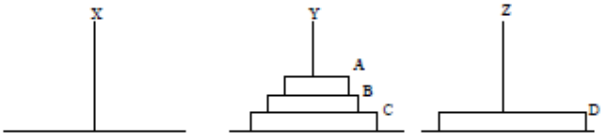
Move disk A from the peg Z to peg X



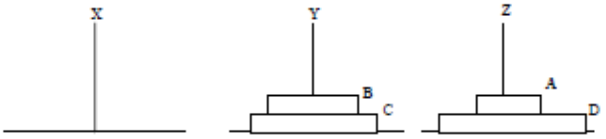
Move disk B from the peg Z to peg Y



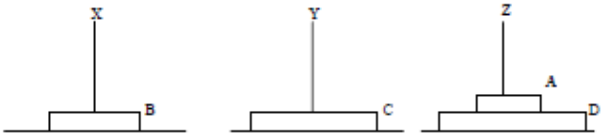
Move disk A from the peg X to peg Y



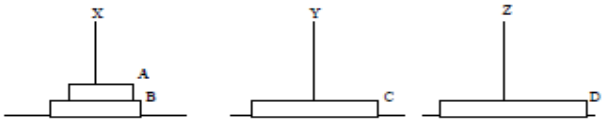
Move disk D from the peg X to peg Z



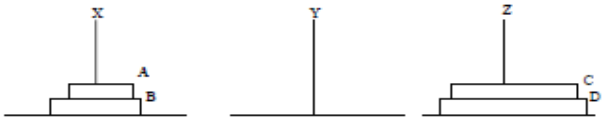
Move disk A from the peg Y to peg Z



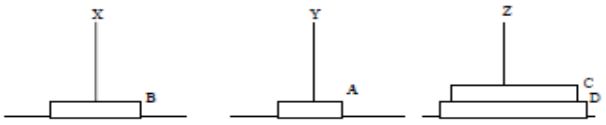
Move disk B from the peg Y to peg X



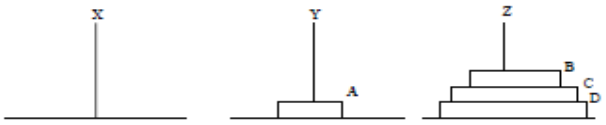
Move disk A from the peg Z to peg X



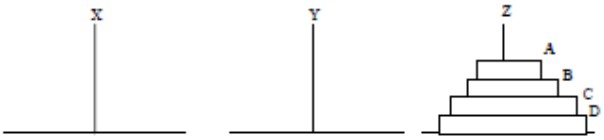
Move disk C from the peg Y to peg Z



Move disk A from the peg X to peg Y



Move disk B from the peg X to peg Z



Move disk A from the tower Y to tower Z

We can generalize the solution to the Tower of Hanoi problem recursively as follows :

To move *n* disks from peg X to peg Z, using Y as auxiliary peg:
1. If $n = 1$, move the single disk from X to Z and stop.
2. Move the top($n - 1$) disks from the peg X to the peg Y, using Z as auxiliary.
3. Move *n*th disk to peg Z.
4. Now move $n - 1$ disk from Y to Z, using Z as auxiliary.

SIMULATING RECURSION

Simulating recursion means to simulate the recursive mechanism by using non-recursive techniques. This is important as many languages like COBOL, FORTRAN and many compilers do not support recursion. So recursive programs can be programmed into non recursive techniques by simulating recursive techniques. To create a correct solution in a non-recursive language the conversion from recursive to non-recursive must be correct. In C language that supports recursion, a recursive

solution is more expensive than a non-recursive one in terms of time as well as space. So a program can be designed in recursive solution and then simulated to the non-recursive solution to use in actual practice.

REMOVAL OF RECURSION

Recursive function can be changed into iterative with the help of looping.

- Convert recursive function to tail recursive
- Convert tail recursive function to iterative.

TAIL RECURSION

Recursive procedures call themselves to work towards a solution to a problem. In simple implementations this balloons the stack as the nesting gets deeper and deeper, reaches the solution, then returns through all of the stack frames. This waste is a common complaint about recursive programming in general. A function call is said to be tail recursive if there is nothing to do after the function returns except return its value. Since the current recursive instance is done executing at that point, saving its stack frame is a waste. Specifically, creating a new stack frame on top of the current, finished, frame is a waste. A compiler is said to implement Tail Recursion if it recognizes this case and replaces the caller in place with the callee, so that instead of nesting the stack deeper, the current stack frame is reused. This is equivalent in effect to a "GoTo", and lets a programmer write recursive definitions without worrying about space inefficiency (from this cause) during execution. TailRecursion is then as efficient as iteration normally is.
The term TailCallOptimization is sometimes used to describe the generalization of this transformation to non-recursive TailCalls. The best-known example of a language that does this is the SchemeLanguage, which is required to support ProperTailCalls. Recursion is the basic iteration mechanism in Scheme.

Consider this recursive definition of the factorial function in C:

```
factorial(n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}
```

This definition is *not* tail-recursive since the recursive call to factorial is not the last thing in the function (its result has to be multiplied by n). But watch this:

```
factorial1(n, accumulator) {
  if (n == 0) return accumulator;
  return factorial1(n - 1, n * accumulator);
}

factorial(n) {
  return factorial1(n, 1);
```

```
}
```

The tail-recursion of factorial1 can be equivalently defined in terms of goto:

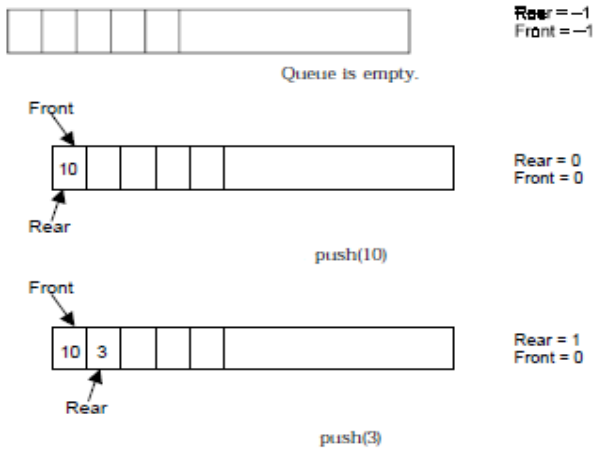
```
factorial1(n, accumulator) {
beginning:
  if (n == 0) return accumulator;
  else {
    accumulator *= n;
    n -= 1;
    goto beginning;
  }
}
```

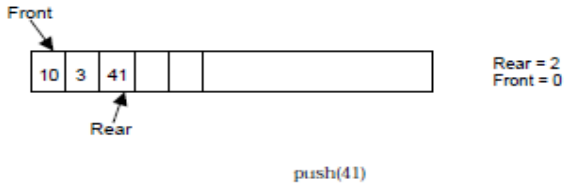
QUEUES

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre. It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*. The basic operations that can be performed on queue are:

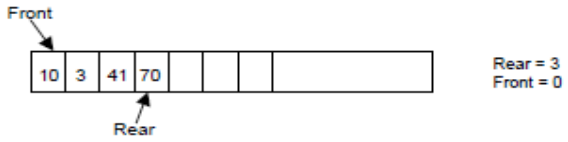
1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

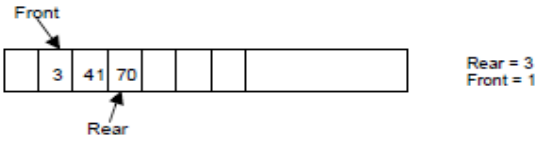




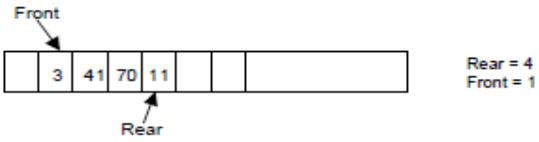
push(41)



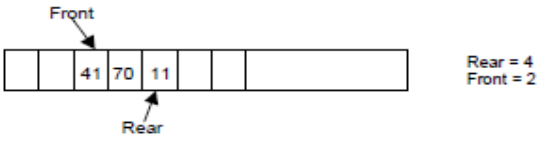
push(70)



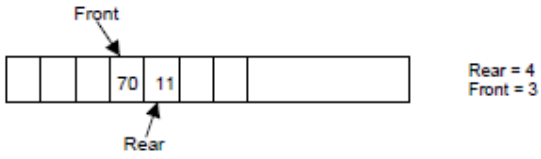
x = pop() (*i.e.*; x = 10)



push(11)



x = pop() (*i.e.*; x = 3)



x = pop() (*i.e.*, x = 41)

ARRAY AND LINKED IMPLEMENTATION OF QUEUES

Queue can be implemented in two ways:
1. Using arrays (static)
2. Using pointers (dynamic)

Let us discuss underflow and overflow conditions when a queue is implemented using arrays. If we try to pop (or delete or remove) an element from queue when it is empty, underflow occurs. It is not possible to delete (or take out) any element when there is no element in the queue. Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements

ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

INSERTING AN ELEMENT INTO THE QUEUE

1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable "data"
3. If (rear >= SIZE)
(*a*) Display "Queue overflow"
(*b*) Exit
4. Else
(*a*) Rear = rear +1
5. Q[rear] = data
6. Exit

DELETING AN ELEMENT FROM QUEUE

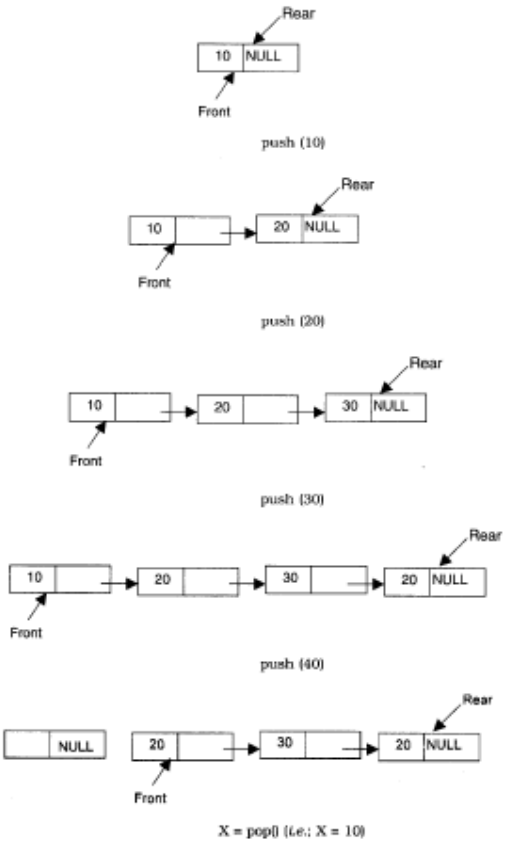
1. If (rear< front)
(*a*) Front = 0, rear = –1
(*b*) Display "The queue is empty"
(*c*) Exit
2. Else
(*a*) Data = Q[front]
3. Front = front +1
4. Exit

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.

QUEUE USING LINKED LIST

Queue is a First In First Out [FIFO] data structure. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in the following figures.



ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode –> DATA = DATA
4. NewNode –>Next = NULL
5. If(REAR not equal to NULL)

(*a*) REAR –>next = NewNode;

6. REAR =NewNode;
7. Exit

ALGORITHM FOR POPPING AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT is equal to NULL)

(*a*) Display "The Queue is empty"

2. Else

(*a*) Display "The popped element is FRONT DATA"

(*b*) If(FRONT is not equal to REAR)

(*i*) FRONT = FRONT –>Next

(*c*) Else

(*d*) FRONT = NULL;

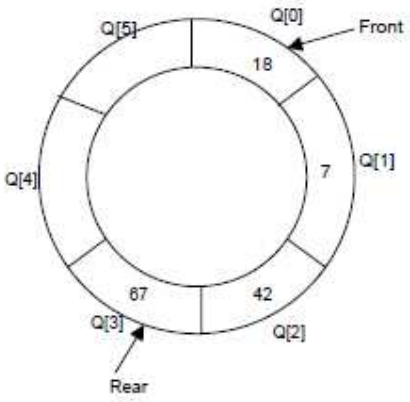
3. Exit

OTHER QUEUES

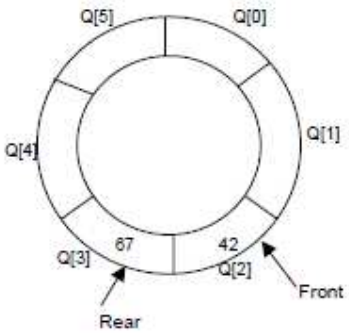
There are three major variations in a simple queue. They are
1. Circular queue
2. Double ended queue (de-queue)
3. Priority queue
Priority queue is generally implemented using linked list.

CIRCULAR QUEUE

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full. Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.

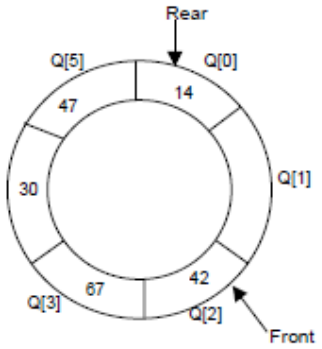


A circular queue after inserting 18, 7, 42, 67.



A circular queue after popping 18, 7

After inserting an element at last location Q[5], the next element will be inserted at the very first location (*i.e.*, Q[0]) that is circular queue is one in which the first element comes just after the last element.



A circular queue after pushing 30, 47, 14

At any time the position of the element to be inserted will be calculated by the relation Rear = (Rear + 1) % SIZE After deleting an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE . After locating the position of the new element to be inserted, *rear*, compare it with *front*. If (rear = front), the queue is full and cannot be inserted anymore.

ALGORITHMS

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Inserting an element to circular Queue

1. Initialize FRONT = – 1; REAR = 1
2. REAR = (REAR + 1) % SIZE
3. If (FRONT is equal to REAR)
(*a*) Display "Queue is full"
(*b*) Exit
4. Else
(*a*) Input the value to be inserted and assign to variable "DATA"
5. If (FRONT is equal to – 1)
(*a*) FRONT = 0
(*b*) REAR = 0
6. Q[REAR] = DATA
7. Repeat steps 2 to 5 if we want to insert more elements
8. Exit

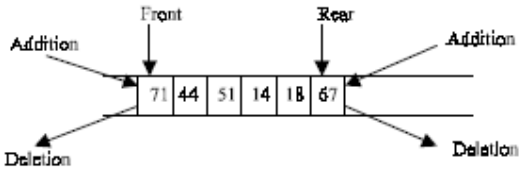
Deleting an element from a circular queue

1. If (FRONT is equal to – 1)
(*a*) Display "Queue is empty"
(*b*) Exit
2. Else
(*a*) DATA = Q[FRONT]
3. If (REAR is equal to FRONT)
(*a*) FRONT = –1
(*b*) REAR = –1
4. Else
(*a*) FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit

DEQUE AND PRIORITY QUEUES

DEQUE

A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.



A deque

There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque
2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists. An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation performed on deque is
1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

ALGORITHMS FOR INSERTING AN ELEMENT

Let Q be the array of MAX elements. *front* (or left) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue *left* and *right* pointer will point to the – 1.

INSERT AN ELEMENT AT THE RIGHT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right + 1))
(*a*) Display “Queue Overflow”
(*b*) Exit
3. If (left == –1)
(*a*) left = 0
(*b*) right = 0
4. Else
(*a*) if (right == MAX –1)
(*i*) left = 0
(*b*) else
(*i*) right = right+1
5. Q[right] = DATA
6. Exit

INSERT AN ELEMENT AT THE LEFT SIDE OF THE DE-QUEUE

1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right+1))
(*a*) Display “Queue Overflow”
(*b*) Exit

3. If (left == – 1)
(*a*) Left = 0
(*b*) Right = 0
4. Else
(*a*) if (left == 0)
(*i*) left = MAX – 1
(*b*) else
(*i*) left = left – 1
5. Q[left] = DATA
6. Exit

ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

DELETE AN ELEMENT FROM THE RIGHT SIDE OF THE DE-QUEUE

1. If (left == – 1)
(*a*) Display “Queue Underflow”
(*b*) Exit
2. DATA = Q [right]
3. If (left == right)
(*a*) left = – 1
(*b*) right = – 1
4. Else
(*a*) if(right == 0)
(*i*) right = MAX-1
(*b*) else
(*i*) right = right-1
5. Exit

DELETE AN ELEMENT FROM THE LEFT SIDE OF THE DE-QUEUE

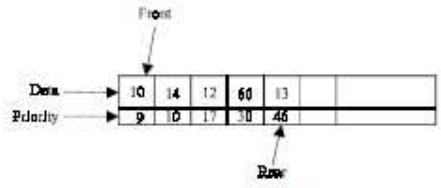
1. If (left == – 1)
(*a*) Display “Queue Underflow”
(*b*) Exit
2. DATA = Q [left]
3. If(left == right)
(*a*) left = – 1
(*b*) right = – 1
4. Else
(*a*) if (left == MAX-1)
(*i*) left = 0
(*b*) Else
(*i*) left = left +1
5. Exit

PRIORITY QUEUES

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

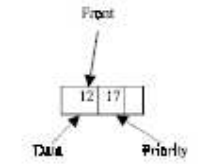
1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.



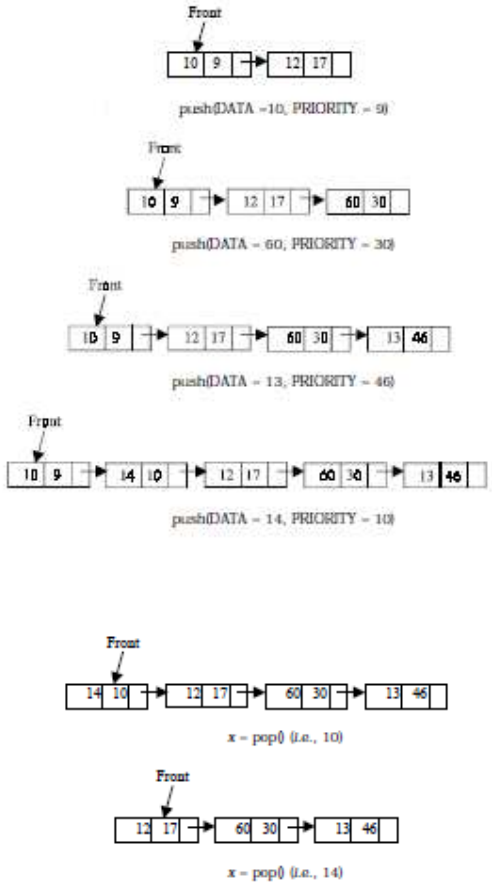
Priority queue representation using arrays

Above Fig. gives a pictorial representation of priority queue using arrays after adding 5 elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is O(n), which is much higher than the other queue (ie; other queues takes only O(1)) for inserting an element. So it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list - which is discussed in this section. A node in the priority queue will contain DATA, PRIORITY and NEXT field. DATA field will store the actual information; PRIORITY field will store its corresponding priority of the DATA and NEXT will store the address of the next node. Fig below shows the linked list representation of the node when a DATA (*i.e.*, 12) and PRIORITY (*i.e.*, 17) is inserted in a priority queue.



Linked list representation of priority queue

When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end. Following figures will illustrate the push and pop operation of priority queue using linked list.



APPLICATIONS OF QUEUE

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.

QUEUE USING TWO STACKS

A queue can be implemented using two stacks. Suppose STACK1 and STACK2 are the two stacks. When an element is pushed on to the queue, push the same on STACK1. When an element is popped from the queue, pop all elements of STACK1 and push the same on STACK2. Then pop the topmost element of STACK2; which is the first (front) element to be popped from the queue. Then pop all elements of STACK2 and push the same on STACK1 for next operation (*i.e., push or pop*).