BYJU'S EXAM PREP

Vision 2023

A Course for GATE & PSUs

Computer Science Engineering

Programming and Data structures

CHAPTER 4

Stacks & Queues

CHAPTER 4 PROGRAMMING & DATA STRUCTURES

STACKS & QUEUES

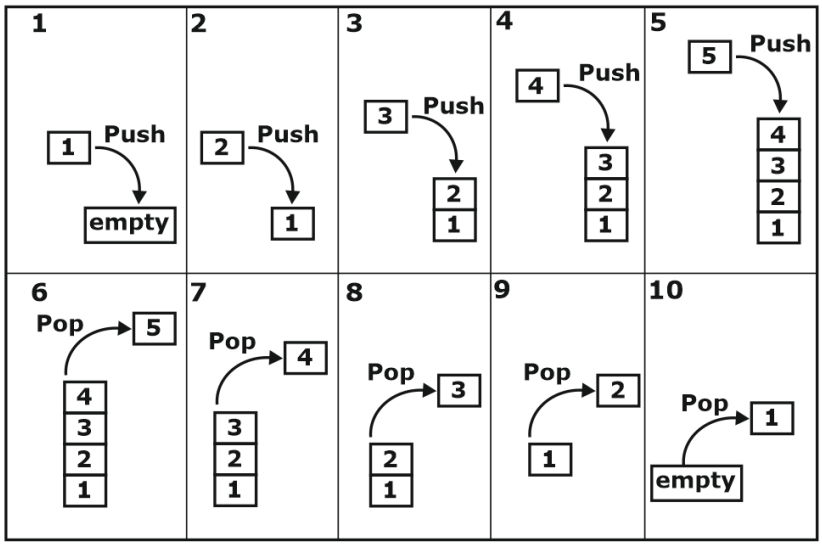
1. STACKS

- A stack is a last in first out (LIFO) abstract data type and data structure.
- There are basically three operations that can be performed on stacks. They are:

a)PUSH: Inserting an item into a stack

b)POP: Deleting an item from the stack.

c)PEEK: Displaying the contents of the stack.



The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if the stack is already full.
6. Before popping, we check if the stack is already empty.

Time Complexities of operations on stack:

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

2. IMPLEMENTATION OF STACK

2.1. Array Implementation: Array implementation aims to create an array where the first element inserted is placed at stack[0] and it will be deleted last. In the array implementation, track of the element inserted at the top must be kept.

Implementing PUSH:

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

Implementing POP:

```
int pop ()
{
if(top == -1)
   {
printf("Underflow");
      return 0;
   }
   else
   {
      return stack[top - - ];
   }
}
```

2.2. Linked List Implementation: It is also called as dynamic implementation as the stack size can grow and shrink as the elements are added or removed respectively from the stack.

- The PUSH operation on stack is same as inserting a node in a Singly linked list.
- The POP operation is same as delete a node in a Singly linked List.

Implementing PUSH:

```
void push ()
{
   int val;
   struct node *ptr=(struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
   {
printf("not able to push the element");
   }
   else
   {
printf("Enter the value");
scanf("%d",&val);
      if(head==NULL)
      {
ptr->val = val;
ptr -> next = NULL;
         head=ptr;
      }
      else
      {
ptr->val = val;
ptr->next = head;
         head=ptr;

      }
printf("Item pushed");

   }
}
```

Implementing POP:

```
void pop()
{
   int item;
   struct node *ptr;
   if (head == NULL)
   {
printf("Underflow");
   }
   else
   {
      item = head->val;
ptr = head;
      head = head->next;
      free(ptr);
printf("Item popped");
   }
}
```

2.3. Queue Implementation: A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented as:

Newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

a)push(s, x) operation's step are described below:

- Enqueue x to q2
- One by one dequeue everything from q1 and enqueue to q2.
- Swap the names of q1 and q2

b)pop(s) operation's function are described below:

- Dequeue an item from q1 and return it.

3. EXPRESSION NOTATION

3.1 Infix Expression: Here, the binary operator comes between the operands.

- The traditional method of our writing of mathematical expressions is called the infix expressions.
- It is of the form <operand><operator><operand>.
- As the name suggests, here the operator is fixed inside between the operands. e.g. A+B here the plus operator is placed inside between the two operators, (A*B)/Q.
- Such expressions are easy to understand and evaluate for human beings. However the computer finds it difficult to parse - Information is needed about operator precedence and associativity rules, and brackets which override these rules.
- Hence we have postfix and prefix notations which make the computer take less effort to solve the problem.

  Eg: a+b

3.2 Postfix Expression: Here, the binary operator comes after both the operands. It is also known as *Polish Notation.*

- The postfix expression as the name suggests has the operator placed right after the two operands.
- It is of the form <operand><operand><operator>
- In the infix expressions, it is difficult to keep track of the operator precedence whereas here the postfix expression itself determines the precedence of operators (which is done by the placement of operators)i.e the operator which occurs first operates on the operand.
- E.g. PQ-C/, here – operation is done on P and Q and then / is applied on C and the previous result.
- A postfix expression is a parenthesis-free expression. For evaluation, we evaluate it from left-to-right.
- Eg: ab+

Infix expression	Postfix expression
(P+Q)*(M-N)	PQ+MN-*
(P+Q) / (M-N) - (A*B)	PQ+MN-/AB*-

3.3 Prefix Expression: Here, the binary operator comes before both the operands. . It is also known as *Reverse Polish Notation.*

- The prefix expression as the name suggests has the operator placed before the operand is specified.
- It is of the form **< operator > < operand > < operand >**.
- It works entirely in the same manner as the postfix expression.
- While evaluating a prefix expression, the operators are applied to the operands immediately on the right of the operator.
- For evaluation, we evaluate it from left-to-right. Prefix expressions are also called polish notation.

Infix expression	Postfix expression
(P+Q)*(M-N)	*+PQ-MN
(P+Q) / (M-N) - (A*B)	-/+PQ-MN*AB

Eg: +ab

4. APPLICATIONS OF STACK

4.1. Infix to Postfix:

Operator stack is used for infix to postfix conversion.

Step 1: Scan the Infix string from left to right.

Step 2: Initialize an empty stack.

Step 3: If the scanned character is an operand, add it to the Postfix string.

Step 4: If the scanned character is an operator and if the stack is empty push the character to stack.

Step 5: If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.

Step 6: If top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.

Step 7: Repeat 4 and 5 steps till all the characters are scanned.

Step 8: After all characters are scanned, we have to add any character that the stack may have to the Postfix string.

If stack is not empty add top Stack to Postfix string and Pop the stack.

Repeat this step as long as the stack is not empty.

EXAMPLE:

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

Stack	**Input**	**Output**
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

4.2. Postfix Evaluation:

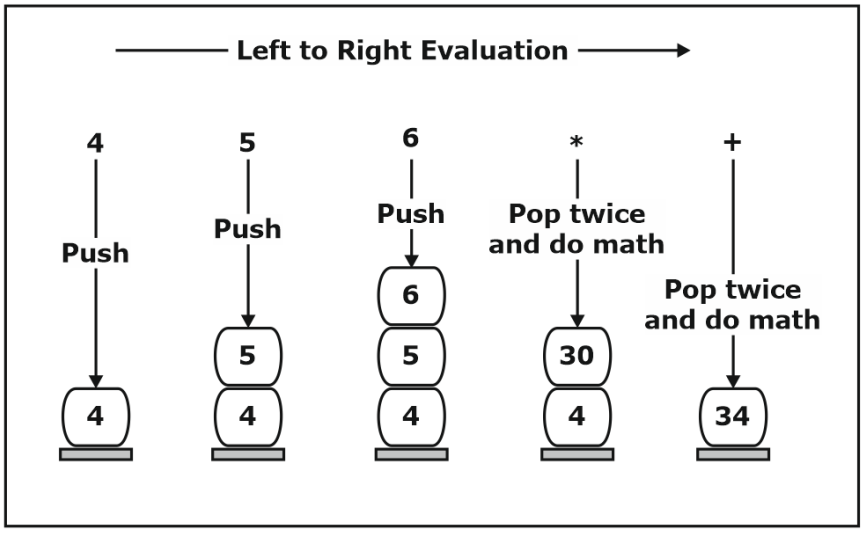
- Operand stack is used for evaluation. Scan the postfix expression.
- When an operand encounters while scanning, push onto stack.
- While scanning postfix expression, if operator found then

  I. POP top two operands on those

  II. Perform the operation on those two operands.

  III.PUSH result on top of stack.
- Finally, the stack contains only one value, which represents the result of the expression.

Example: 456*+



postfix evolutions through stack in C

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop (2 elements & Evaluate)	4	5*6 = 30
5.		Push result (30)	4,30	
6.	+	Pop (2 elements & Evaluate)	Empty	4 + 30 = 34
7.		Push result (34)	34	
8.		No-more elements (pop)	Empty	34 (Result)

For eg: 82/ will evaluate to 8/2=4

138*+ will evaluate to (1+3*8)= 25

4.3. Prefix Evaluation

- Accept a prefix string from the user.
  Say (-*+ABCD), let A = 4, B = 3, C = 2, D = 5
  i.e. (-*+4325) is the input prefix string.
- Start scanning the string from the right one character at a time.
- If it is an operand, push it in the stack.
- If it is an operator, pop opnd1, opnd2 and perform the operation, specified by the operator. Push the result in the stack.
- Repeat these steps until all of the input prefix string ends.

Example: -*+4325

Symbol	Opnd1	Opnd2	Value	Opndstack
5				5
2				5, 2
3				5, 2, 3
4				5, 2, 3, 4
+	4	3	7	5, 2
				5, 2, 7
*	7	2	14	5
				5, 14
-	14	5	9	Empty
				9

4.4. Prefix to Postfix:

Step 1: Iterate the given expression from right to left, one character at a time

Step 2: If the character is operand, push it to the stack.

Step 3: If the character is operator,

a.Pop an operand from the stack, say it's s1.

b.Pop an operand from the stack, say it's s2.

c.perform (s1 s2 operator) and push it to stack.

Step 4: Once the expression iteration is completed, initialize the result string and pop out from the stack and add it to the result.

Step 5: Return the result.

(+-435) ☐ (43-5+)

Symbol	Opnd1	Opnd2	Value	Opndstack
5				5
3				5, 3
4				5, 3, 4
-	4	3	43-	5
				5, 43-
+	43-	5	43-5+	
				43-5+

Equivalent Post fix notation: 43-5+

4.5. Infix to Prefix:

Steps to convert infix to prefix expression:

1. Scan the valid infix expression from right to left.
2. Initialize the empty character stack and empty prefix string.
3. If the scanned character is an operand, add it to the prefix string.
4. If the scanned character is closing parenthesis, push it to the stack.
5. If the scanned character is opening parenthesis, pop all the characters of the stack and concatenate to the end of the prefix string until closing parenthesis occurs.
6. Pop the closing parenthesis.
7. If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack. If the top element of the Stack has higher precedence over the scanned character, pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.
8. Repeat steps 3 to 7 until all characters are scanned.
9. If the stack is empty, reverse and return the prefix string.
10. Else, pop the elements of the stack and add it to the prefix string, and then reverse and return the prefix string.

Example 1: Convert A * (B + C) * D to postfix notation.

a. ABC+*D

b. AB+C*D

c. ABC+D*

d. None

Answer: A

Move	Current Token	Stack	Output
1	A	empty	A
2	*	*	A
3	(	(*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C
7	)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10		empty	

Example 2: What is the result of the given postfix expression? abc*+ where a=1, b=2, c=3.

a) 4

b) 5

c) 6

d) 7

Answer: D

The infix expression is a+b*c. Evaluating it, we get 1+2*3=7

Example 3: +9*26

a. 22

b. 23

c. 21

d. 20

Character Scanned	Stack (Front to Back)	Explanation
6	6	6 is an operand, push to Stack
2	62	2 is an operand, push to Stack
*	12 (6*2)	* is an operator, pop 6 and 2, multiply them and push result to Stack
9	12 9	9 is an operand, push to Stack
+	21 (12 + 9)	+ is an operator, pop 12 and 9 add them and push the result to Stack.

Result: 21

Example 4: Convert *-A/BC-/AKL

a. ABC/-AK/L-*

b. ABC*-/AK/L-*

c. ABC-/AKL-*

d. ABC/-A/K-*

Sol: Option A

Token	Action	Stack	Notes
L	Push **L** to stack	[L]	
K	Push **K** to stack	[L, K]	
A	Push **A** to stack	[L, K, A]	
/	Pop **A** from stack	[L, K]	Pop two operands from stack, A and K. Perform AK/and push AK/to stack
	Pop **K** from stack	[L]	
	Push **AK/**to stack	[L, AK/]	
-	Pop **AK/**from stack	[L]	Pop two operands from stack, AK/and L. Perform AK/L- and push AK/L- to stack
	Pop **L** from stack	[]	
	Push **AK/**L-to stack	[AK/L-]	
C	Push **C** to stack	[AK/L-, C]	
B	Pop **B** to stack	[AK/L-, C, B]	
/	Pop **B** from stack	[AK/L-, C]	Pop two operands from stack, B and C. Perform BC/ and push BC/to stack
	Pop **C** from stack	[AK/L-]	
	Push **BC/** to stack	[AK/L-, BC/]	
A	Push **A** to stack	[AK/L-, BC/,A]	
-	Pop **A** from stack	[AK/L-, BC/]	Pop two operands from stack, A and BC. Perform ABC/- and push ABC/-to stack
	Pop **BC/** from stack	[AK/L-]	
	Push **ABC/-** to stack	[AK/L-, ABC/-]	
*	Pop **ABC/-** from stack	[AK/L-]	Pop two operands from stack, ABC/- and AK/L-. Perform ABC/-AK/L-* and push ABC/-AK/L-* to stack
	Pop **AK/L-** from stack	[]	
	Push **ABC/-AK/L-*** to stack	[ABC/-AK/L-*]	
Postfix Expression: ABC/-AK/L-*			

Example 5: Convert the following infix expression to prefix expression:

(a+ (b*c)/(d-e)) = +a/*bc-de

Scan the infix in reverse order i.e. right to left:

Symbol	Prefix	Stack
)	Empty	)
)	Empty	))
e	e	))
-	e	))-
d	de	))-
(	-de	)
/	-de	)/
)	-de	)/)
c	c-de	)/)
*	c-de	)/)*
b	bc-de	)/)*
(	*bc-de	)/
+	/*bc-de	)+
a	a/*bc-de	)+
(	+a/*bc-de	Empty

4.6. Recursion using Stacks:

- "Recursion" is the technique of solving any problem by calling the same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number
- In recursion the last function called needs to be completed first.
- As Stack is a LIFO data structure i.e. (Last In First Out) and hence it is used to implement recursion.
- The High level Programming languages, such as Pascal , C etc. that provide support for recursion use stack for bookkeeping.
- In each recursive call, there is need to save the

  1.current values of parameters,

  2.local variables and

  3.the return address (the address where the control has to return from the call).
- Also, as a function calls another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

Types of recursion-

Recursion are mainly of two types depending on whether a function calls itself from within itself or more than one function call one another mutually. The first one is called direct recursion and another one is called indirect recursion. Thus, the two types of recursion are:

Direct Recursion: These can be further categorized into four types:

- Tail Recursion: If a recursive function calling itself and that recursive call is the last statement in the function then it's known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

Example:

```
// Code Showing Tail Recursion
#include <stdio.h>
// Recursion function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);

        // Last statement in the
function
        fun(n - 1);
    }
}

// Driver Code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Output:

3 2 1

- **Head Recursion**: If a recursive function calling itself and that recursive call is the first statement in the function then it's known as Head Recursion**.** There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

Example

```
// C program showing Head Recursion
#include <stdio.h>
// Recursive function
void fun(int n)
{
    if (n > 0) {

        // First statement in the function
        fun(n - 1);

        printf("%d ", n);
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Output:

1 2 3

- **Tree Recursion**: To understand Tree Recursion let's first understand Linear Recursion. If a recursive function calling itself for one time then it's known as Linear Recursion. Otherwise if a recursive function calling itself for more than one time then it's known as Tree Recursion.

Example:

Pseudo Code for linear recursion

```
fun(n)
{
   if(n>0)
{

fun(n-1); // Calling itself only once
{

}
```

Program for tree recursion

```
// C program to show Tree Recursion

#include <stdio.h>

// Recursive function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);

        // Calling once
        fun(n - 1);

        // Calling twice
        fun(n - 1);
    }
}

// Driver code
int main()
{
    fun(3);
    return 0;
}
```

Output:

3 2 1 1 2 1 1

- **Nested Recursion**: In this recursion, a recursive function will pass the parameter as a recursive call.That means **"recursion inside recursion".** Let's see an example to understand this recursion.

Example:

```
// C program to show Nested Recursion
#include <stdio.h>
int fun(int n)
{
    if (n > 100)
        return n - 10;
```

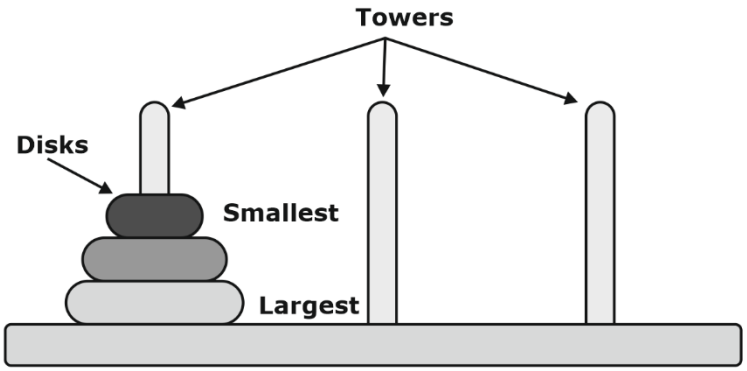
```
    // A recursive function passing
parameter
    // as a recursive call or recursion
    // inside the recursion
    return fun(fun(n + 11));
}
// Driver code
int main()
{
    int r;
    r = fun(95);
    printf("%d\n", r);
    return 0;
}
```

Output:

91

Tower of Hanoi-

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some other tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.

Tower of Hanoi puzzle with n disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken 2^3 - 1 = 7 steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser number of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to the destination peg.
- And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine applying the same in a recursive way for all given set of disks.

The steps to follow are –

Step 1 – Move n-1 disks from source to aux

Step 2 – Move n^{th} disk from source to dest

Step 3 – Move n-1 disks from aux to dest

A recursive algorithm for Tower of Hanoi can be driven as follows –

START

Procedure Hanoi(disk, source, dest, aux)

```
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)     // Step 1
    move disk from source to dest          // Step 2
    Hanoi(disk - 1, aux, dest, source)     // Step 3
  END IF
```

END Procedure

STOP

5. QUEUES

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends.

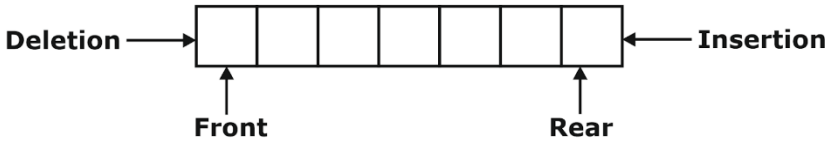
One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

Queues follow the **First In First Out (FIFO)** i.e. the first element that is added to the queue is the first one to be removed.

Elements are always added to the back and removed from the front.

5.1. TYPES OF QUEUE

5.1.1. Simple Queue



A simple queue performs the operations simply. i.e., the insertion and deletions are performed likewise. Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.

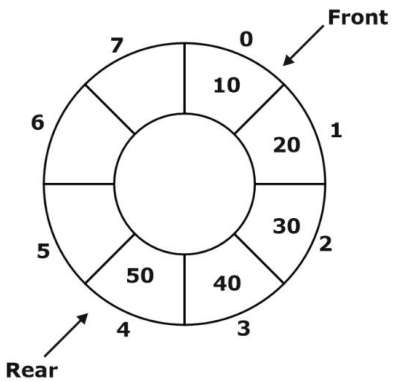
All nodes are connected to each other in a sequential manner. The pointer of the first node points to the value of the second and so on.

The first node has no pointer pointing towards it whereas the last node has no pointer pointing out from it.

Adding an element will be performed after checking whether the queue is full or not. If rear < n which indicates that the array is not full then store the element at arr[rear] and increment rear by 1 but if rear == n then it is said to be an Overflow condition as the array is full.

An element can only be deleted when there is at least an element to delete i.e. rear > 0. Now, elements at arr[front] can be deleted but all the remaining elements have to be shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.

5.1.2. Circular Queue

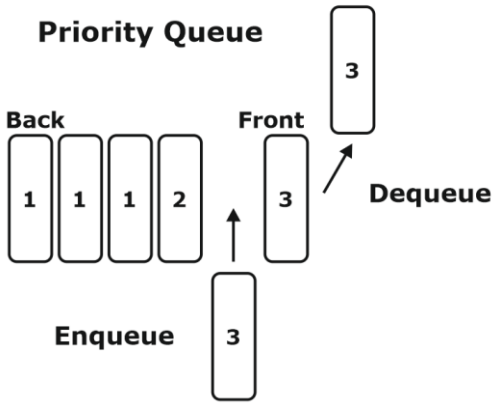


Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node's pointer is also connected to the first node's address. Hence, the last node and the first node also get connected making a circular link overall.

A circular queue overcomes the problem of un-utilized space in linear queues implemented as arrays. We can make following assumptions for circular queue.

- If : (Rear+1) % n == Front, then queue is Full
- If Front = Rear, the queue will be empty.
- Each time a new element is inserted into the queue, the Rear is incremented by 1. Rear = (Rear + 1)) % n
- Each time, an element is deleted from the queue, the value of Front is incremented by one. Front = (Front + 1) % n

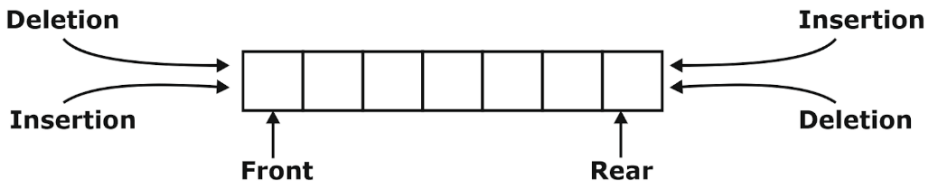
5.1.3. Priority Queue



Priority queue makes data retrieval possible only through a pre-determined priority number assigned to the data items.

While the deletion is performed in accordance with the priority number (the data item with highest priority is removed first), insertion is performed only in the order.

5.1.4. Doubly Ended Queue (Deque)



The doubly ended queue or deque allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them.

5.2 OPERATIONS ON QUEUE

In the queue, we always dequeue (or access) data, pointed by the front pointer and while enqueuing (or storing) data in the queue we take help of rear pointer.

5.2.1. Enqueue:

Add (store) an item to the queue.

If the queue is not full, this function adds an element to the back of the queue, else it prints "OverFlow".

Step 1: Check if the queue is full.

Step 2: If the queue is full, produce overflow error and exit.

Step 3: If the queue is not full, increment **rear** pointer to point the next empty space.

Step 4: Add data element to the queue location, where the rear is pointing.

Step 5: return success.

procedure enqueue(data)

Pseudo Code of Enqueue:

```
procedure enqueue(data)
   if queue is full
       return overflow
   endif
   rear ← rear + 1
   queue[rear] ← data
   return true
end procedure
```

Implementation of enqueue() in C program:

```
void enqueue(int queue[], int element, int& rear, int arraySize) {
if(rear == arraySize)            // Queue is full
printf("OverFlow\n");
else{
        queue[rear] = element;    // Add the element to the back
        rear++;    }
}
```

5.2.2. Dequeue:

Remove (access) an item from the queue.Accessing data from the queue is a process of two tasks – access the data where front is pointing and remove the data after access.

If the queue is not empty, this function removes the element from the front of the queue, else it prints "UnderFlow".

Step 1: Check if the queue is empty.

Step 2: If the queue is empty, produce an underflow error and exit.

Step 3: If the queue is not empty, access the data where **front** is pointing.

Step 4: Increment **front** pointer to point to the next available data element.

Step 5: Return success.

Pseudo Code of Dequeue:

```
procedure dequeue
   if queue is empty
      return underflow
   end if
   data = queue[front]
   front ← front + 1
   return true
end procedure
```

1

Implementation of enqueue() in C program:

```
void dequeue(int queue[], int& front, int rear) {
if(front == rear)            // Queue is empty
printf("UnderFlow\n");
   else {
      queue[front] = 0;       // Delete the front element
      front++;
   }
}
```

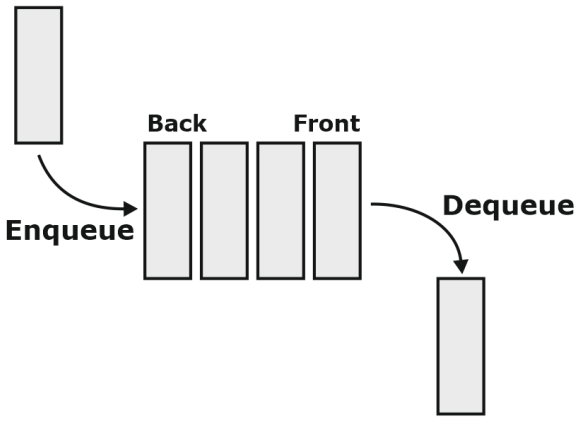
Time Complexities:

- queue insertion i.e. Enqueue takes O(1).
- queue deletion i.e. Dequeue takes O(1).
- Access time is O(n).

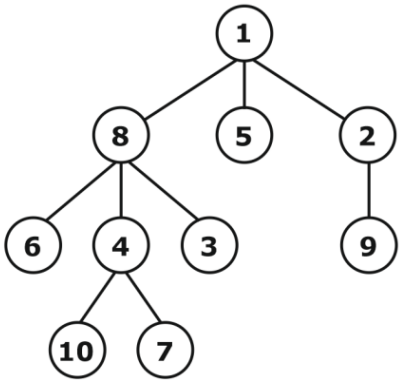
6. APPLICATIONS OF QUEUE

6.1. BFS using queue:

- Queues operate on a first in, first out (FIFO) principle of item access. It is like a standard line at the grocery store. Anyone that enters the line will be served in the order in which they entered.
- When enqueue is performed, element is pushing the element onto the array (think of this as getting in line at the store).
- When an element is dequeued, it is removed from the queue, and the next element is then ready to be dequeued (ie. you check out of the grocery line, and the next customer is ready to be served at the register).



One area where this comes into play is breadth first search. Let's say you have a tree of nodes:



The root node (1), would have a data property of 1, with a children array of nodes 8, 5, and 2. The node with the data property of 8 would have children array of nodes 6, 4, and 3, and so on.

6.2. Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.

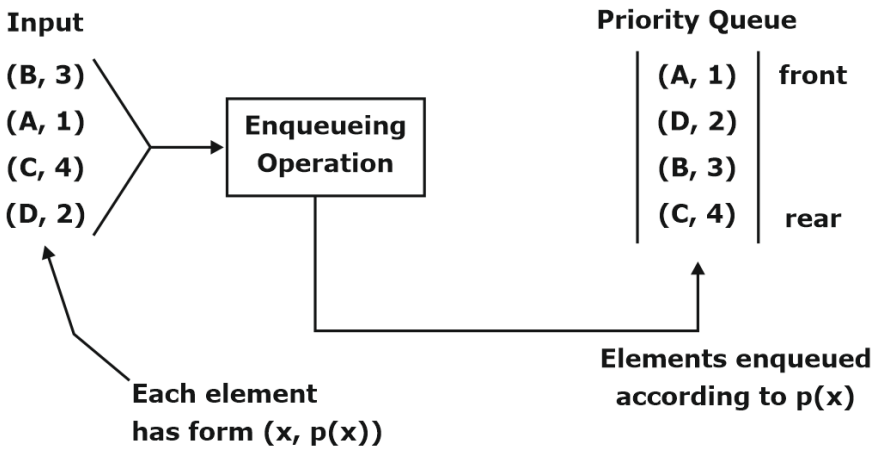
Generally, the value of the element itself is considered for assigning the priority.

For example: The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priority according to our needs.

There are two types of priority queue:

a)Ascending Priority Queue: It is a collection of items in which items can be inserted arbitrarily and from which only the smallest items can be removed.

b)Descending Priority Queue: It is a collection of items in which items can be inserted arbitrarily and from which only the largest items can be removed.



6.2.1. Priority Queue Applications

Priority queues have many applications and below are few of them:

- Data compression: Huffman Coding algorithm
- Shortest path algorithms: Dijkstra's algorithm
- Minimum spanning tree algorithms: Prim's algorithm
- Event-driven simulation: customers in a line
- Selection problem: Finding k^{th} -smallest element

6.3. IMPLEMENTING QUEUE USING STACK

The following algorithm will implement a queue using two stacks.

Step 1: When calling the enqueue method, simply push the elements into the stack 1.

Step 2: If the dequeue method is called, push all the elements from stack 1 into stack 2, which reverses the order of the elements. Now pop from stack 2.

For example: Suppose we push "a", "b", "c" to a stack. If we are trying to implement a queue and we call the dequeue method 3 times, we actually want the elements to come out in the order: "a", "b", "c", which is in the opposite order they would come out if we popped from the stack. So, basically, we need to access the elements in the reverse order that they exist in the stack.

6.4 APPLICATIONS OF QUEUE

- Breadth first Search can be implemented.
- CPU Scheduling
- Handling of interrupts in real-time systems
- Routing Algorithms
- Computation of shortest paths
- Computation a cycle in the graph

****