KALASALINGAM UNIVERSITY

(**K**alasalingam **A**cademy of **R**esearch and **E**ducation)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CLASS NOTES

DATA STRUCTURES (CSE255)

Prepared by
M.RAJA
kingraaja@gmail.com

CSE 255	DATA STRUCTURES	L	T	P	C
		3	0	0	3

PROBLEM SOLVING

Problem solving – Top-down Design – Implementation – Verification – Efficiency – Analysis – Sample algorithms.

LISTS, STACKS AND QUEUES

Abstract Data Type (ADT) – The List ADT – The Stack ADT – The Queue ADT

TREES

Preliminaries – Binary Trees – The Search Tree ADT – Binary Search Trees – AVL Trees – Tree Traversals – Hashing – General Idea – Hash Function – Separate Chaining – Open Addressing – Linear Probing – Priority Queues (Heaps) – Model – Simple implementations – Binary Heap

SORTING

Preliminaries – Insertion Sort – Shellsort – Heapsort – Mergesort – Quicksort – External Sorting

GRAPHS

Definitions – Topological Sort – Shortest - path Algorithms – Unweighted shortest paths – Dijkstra's Algorithm – Minimum Spanning Tree – Prim's Algorithm – Applications of Depth–First Search – Undirected Graphs – Biconnectivity – Introduction to NP-Completeness

TEXT BOOK

1. Dromey R. G., How to Solve it by Computer, PHI, 2002.

REFERENCES

1. Langsam Y., Augenstein M. J., Tenenbaum A. M., Data Structures using C,
2. Pearson Education Asia, 2004
3. Richard F. Gilberg, Behrouz A. Forouzan, Data Structures – A Pseudocode Approach with C, Thomson Brooks, 1998.
4. Aho. et.al., Data Structures and Algorithms, Pearson Education Asia, 1983.

LESSON PLAN:

Topic no	Topic Name	References	No. of Periods	Cum. No of periods
UNIT – I PROBLEM SOLVING				
1.	Problem Solving	T1(1-7)	2	2
2.	Top-down Design	T1(7-13)	1	3
3.	Implementation	T1(14-19)	1	4
4.	Verification	T1(19-29)	1	5
5.	Efficiency	T1(29-33)	1	6
6.	Analysis**(Student Seminar)**	T1(33-39)	1	7
7.	Sample Algorithms	T1(42-84)	3	10
8.	Tutorial		2	12
UNIT – II LISTS,STACKS AND QUEUES				
9.	Abstract Data Type (ADT)	T2(57-58)	1	13
10.	The List ADT	T2(58-78)	3	16
11.	The Stack ADT	T2(78-95)	2	18
12.	The Queue ADT**(Student Seminar)**	T2(95-101)	2	20
13.	Tutorial		2	22
UNIT – III TREES				
14.	Preliminaries	T2(105-111)	1	23
15.	Binary Trees	T2(111-116)	1	24
16.	The Search Tree ADT – Binary Search Trees	T2(116-126)	2	26
17.	AVL Trees	T2(126-139)	2	28
18.	Tree Traversals**(Student Seminar)**	T2(148-149)	1	29
19.	Hashing – General Idea – Hash Function	T2(165-168)	1	30
20.	Separate Chaining – Open Addressing – Linear Probing	T2(168-175)	1	31
21.	Priority Queues (Heaps) – Model	T2(193-194)	1	32
22.	Simple Implementation – Binary Heap	T2(194-205)	2	34
23.	Tutorial		2	36
UNIT – IV SORTING				
24.	Preliminaries	T2(235-236)	1	37

25.	Insertion Sort	T2(236-237)	1	38
26.	Shell Sort	T2(238-242)	1	39
27.	Heap Sort	T2(242-246)	2	41
28.	Merge Sort(**Student Seminar)**	T2(246-251)	1	42
29.	Quick Sort	T2(251-263)	2	44
30.	External Sorting	T2(266-272)	2	46
31.	Tutorial		2	48
UNIT – V GRAPHS				
32.	Definitions – Topological Sort	T2(299-306)	2	50
33.	Shortest-path Algorithms – Un weighted Shortest paths	T2(306-311)	1	51
34.	Dijkstra's Algorithm	T2(311-320)	1	52
35.	Minimum Spanning Tree – Prim's Algorithm	T2(329-332)	1	53
36.	Applications of Depth-First Search(**Student Seminar)**	T2(335-336)	1	54
37.	Undirected Graphs – Bi Connectivity	T2(336-342)	2	56
38.	Introduction to NP-Completeness	T2(348-353)	2	58
39.	Tutorial		2	60

Introduction:

Abstract Data Type(ADT):

An Abstract Data Type is a set of operations. Abstract data types are mathematical abstractions. Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do abstract data types. For the set ADT, we might have such operations as union, intersection, size, and complement.

The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision.

ADT Types are

- **Stack ADT**
- **Queue ADT**
- **Linked List ADT**

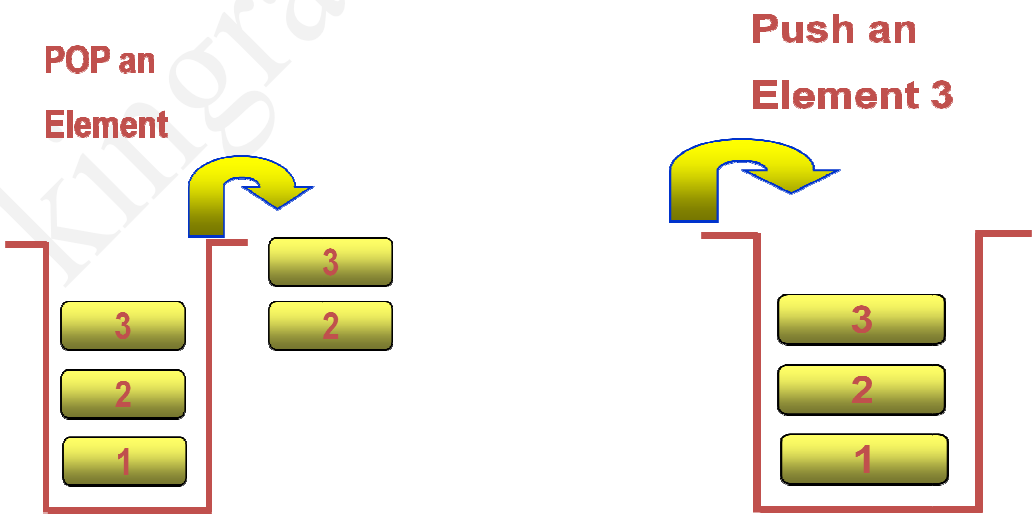
STACK:

- A stack is a collection of data items that can be accessed at only one end, called **top**.
- Items can be inserted and deleted in a stack only at the top.
- The last item which was inserted in a stack will be the first one to be deleted.
- Therefore, a stack is called a **Last-In-First-Out (LIFO)** data structure.

There are two basic operations that are performed on stacks:

- PUSH
- POP

- **PUSH**: It is the process of inserting a new element on the top of a stack.
- **POP**: It is the process of deleting an element from the top of a stack.



Stacks can be implemented in two ways

i) Using an arrays

ii) Linked List.

Firstly we will discuss about the implementation of Stack using Array:

- A stack is similar to a list in which insertion and deletion is allowed only at one end.
- Therefore, similar to a list, stack can be implemented using both arrays and linked lists.
- To implement a stack using an array:
  - Consider size of the Stack is 5. i.e we can hold only 5 elements in the stack.

Declare an array:

```
int Stack[5];                    // Maximum size needs to be specified in advance //
```

Declare a variable, top to hold the index of the topmost element in the stacks:

```
int top;          /* stack elements has to be referred using a variable TOP */
```

Initially, when the stack is empty, set:

```
top = –1                                   /* now top= -1 so stack is empty*/
```

PUSH:

Let us now write an algorithm for the PUSH operation.

Before we PUSH an element into the Stack we must ensure that whether the stack contains overflow or not. (Overflow means the array is filled (size 5)). If yes, PUSH operation cannot be performed.

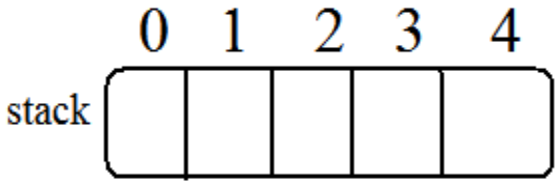
- To avoid the stack overflow, you need to check for the stack full condition before pushing an element into the stack.

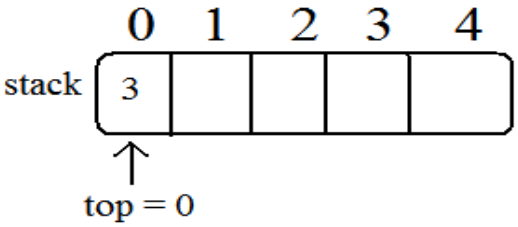
Algorithm to PUSH:

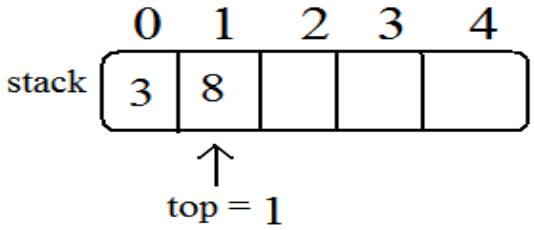
1. If top = MAX – 1:
   a. Display "Stack Full"
   b. Exit
2. Increment top by 1.
3. Store the value to be pushed at index top in the array.

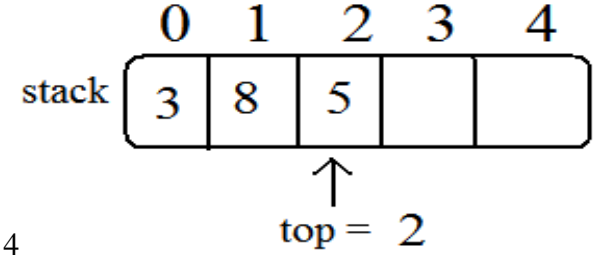
Top now contains the index of the topmost element.

Stack representation:

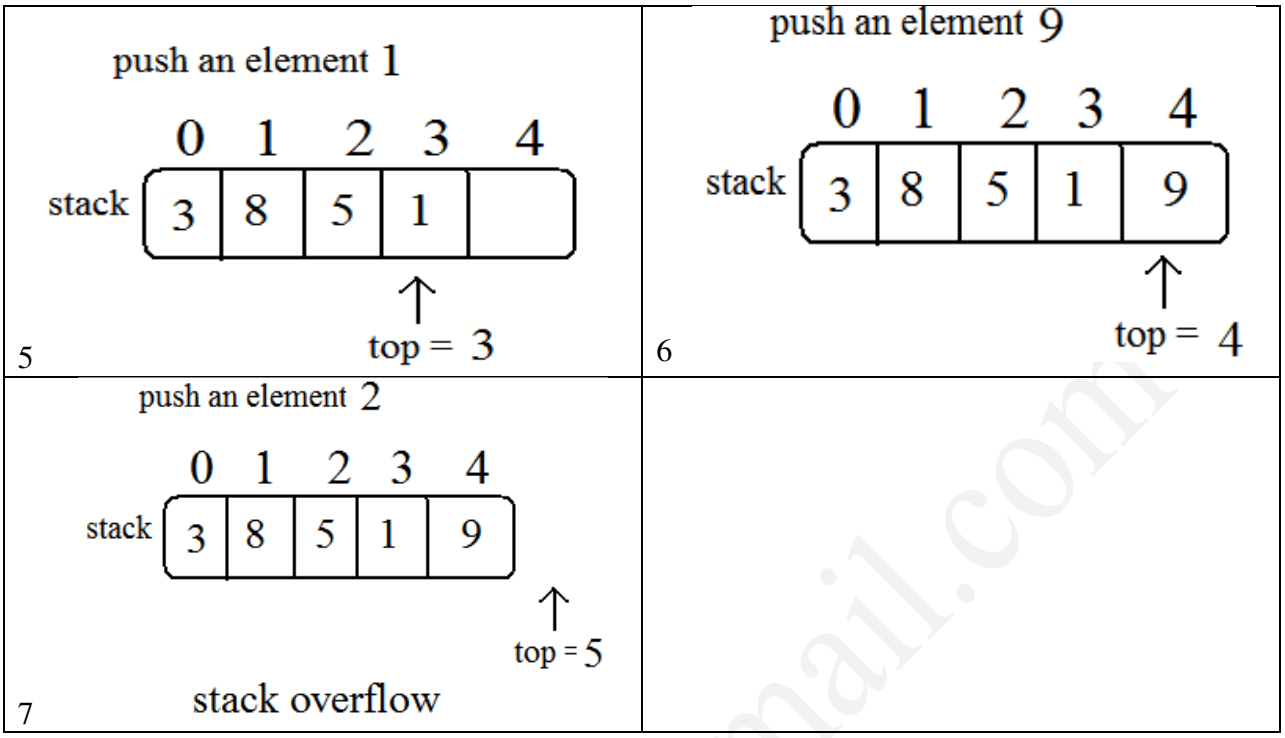
<table>
<tr>
<td>Initially: Top = -1
push an element 3
 

1</td>
<td>Now Top = 0
push an element 3
 

2</td>
</tr>
<tr>
<td>push an element 8
 

3</td>
<td>push an element 5
 

4</td>
</tr>
</table>



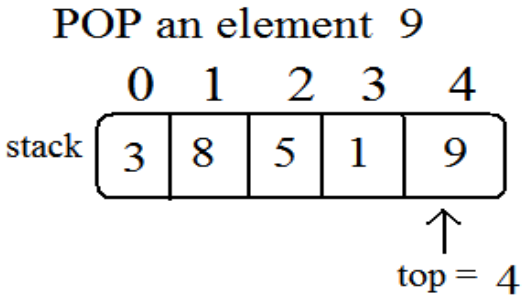
POP:

- The algorithm for POP operation is as follows:
  1. Make a variable/pointer temp point to the topmost node.
  2. Retrieve the value contained in the topmost node.
  3. Make top point to the next node in sequence.
  4. Release memory allocated to the node marked by temp.

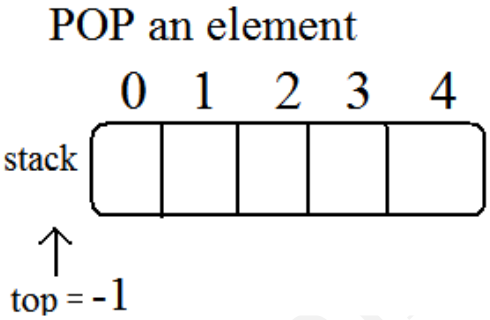
Algorithm for POP operation:

1. If top = – 1:
   a. Display "Stack Empty"
   b. Exit
2. Retrieve the value stored at index top
3. Decrement top by 1

Now the below fig contains 5 elements. If you do POP the topmost element will be removed from the stack. When all the elements are popped no more elements are there to remove. That condition is called 'underflow' condition.



When you keep on removing the elements all the elements are removed now. Now the stack is empty.



STACK USING ARRAY

```
/******    Program to Implement Stack using Array    ******/

#include <stdio.h>
#define MAX 50
void push();
void pop();
void display();

int stack[MAX], top=-1, item;
main()
{
  int ch;
  do
  {
    printf("\n\n\n\n1.\tPush\n2.\tPop\n3.\tDisplay\n4.\tExit\n");
    printf("\nEnter your choice: ");
```

```
    scanf("%d", &ch);

    switch(ch)
    {
     case 1:
                push();
                break;
     case 2:
                pop();
                break;
     case 3:
                display();
                break;
     case 4:
                exit(0);
      default:
                printf("\n\nInvalid entry. Please try again...\n");
    }
  } while(ch!=4);
  getch();
}
void push()
{
  if(top == MAX-1)
            printf("\n\nStack is full.");
  else
     {
      printf("\n\nEnter ITEM: ");
      scanf("%d", &item);

      top++;
      stack[top] = item;
      printf("\n\nITEM inserted = %d", item);
     }
}

void pop()
```

```
{
 if(top == -1)
        printf("\n\nStack is empty.");
 else
        {
           item = stack[top];
           top--;
           printf("\n\nITEM deleted = %d", item);
         }
}

void display()
{
 int i;
   if(top == -1)
           printf("\n\nStack is empty.");
 else
        {
          for(i=top; i>=0; i--)
          printf("\n%d", stack[i]);
          }
}
```

STACK USING LINKED LIST

When a stack is implemented as a linked list, there is no upper bound limit on the size of the stack. Therefore, there will be **no stack full condition** in this case.

A Linked list is a chain of structs or records called Nodes. Each node has at least two members, one of which points to the next node in the list and the other holds the data. These are defined as Single linked Lists because they can point to the next Node in the list.

```
struct node
{
   int data;
   struct node *link;          //to maintain the link other nodes
};
struct node *top=NULL,*temp;
```

We use above structure for a Node in our example. Variable Data holds the data in the Node while pointer of the type *struct node next* holds the address to the next node in the list. Top is a pointer of type *struct node* which acts as the top of the list.

Initially we set 'top' as NULL which means list is empty.

Procedure to create a list:

i) Create a new empty node top.

ii) Read the stack element and store it in top's data area.

iii) Assign top's link part as NULL (i.e. top->link=NULL).

iv) Assign temp as top (i.e. temp=top).

/* create function create the head node */

```
void create( )
{
printf("\nENTER THE FIRST ELEMENT: ");
top=(struct node *)malloc(sizeof(struct node));
scanf("%d",&top->data);
top->link=NULL;
temp=top;
}
```

Procedure to PUSH an element into the List:

a) Check Main memory for node creation.

b) Create a new node 'top'.

c) Read the stack element and store it in top's data area.

d) Assign top's link part as temp (i.e. top->link=temp).

e) Assign temp as top (i.e. temp=top).

Syntax:

```
void push()
        {
        printf("\nENTER THE NEXT ELEMENT: ");
        temp=(struct node *)malloc(sizeof(struct node));
        scanf("%d",&temp->data);
        temp->link=top;
        top=temp;
        }
```

Procedure to POP an element from the list:

a) If top is NULL then display stack is empty.

b) Otherwise assign top as temp (i.e. top=temp, bring the top to top position)

c) Assign temp as temp's link. (i.e. temp=temp->link, bring the temp to top's previous position).

d) Delete top from memory.

Syntax:

```
void pop()
{
        if(top==NULL)
        {
        printf("\nSTACK IS EMPTY\n");
        }
else
        {
```

```
            temp=top;
            printf("\nDELETED ELEMENT IS %d\n",top->data);
            top=top->link;
            free(temp);
            }
    }
```

Traversing the List

If it is traverse(visiting all the nodes), then process the following steps

a) Bring the top to stack's top position(i.e. top=temp)

b) Repeat until top becomes NULL

i) Display the top's data.

ii) Assign top as top's link (top=top->link).

Syntax:

```
/* display function visit the linked list from top to end */

void display()
    {
    top=temp;       // bring the top to top position
    printf("\n");
    while(top!=NULL)
        {
        printf("%d\n",top->data);
        top=top->link;   // Now top points the previous node in the list
        }
    }
```

IMPLEMENTATION OF STACK USING LINKED LIST

Program

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<stdlib.h>

/* Node decleration */
struct node
{
int data;
struct node *link; //to maintain the link other nodes
};
struct node *top,*temp;

void create();
void push();
void pop();
void display();
/* create function create the head node */
void create()
{
printf("\nENTER THE FIRST ELEMENT: ");
temp=(struct node *)malloc(sizeof(struct node));
scanf("%d",&temp->data);
temp->link=NULL;
top=temp;
}

/* display function visit the linked list from top to end */
void display()
{
temp=top;       // bring the top to top position
printf("\n");
while(temp!=NULL)
{
printf("%d\n",temp->data);
```

```
temp=temp->link;   // Now top points the previous node in the list
}
}

void push()
{
printf("\nENTER THE NEXT ELEMENT: ");
temp=(struct node *)malloc(sizeof(struct node));
scanf("%d",&temp->data);
temp->link=top;
top=temp;
}

void pop()
{
if(top==NULL)
{
printf("\nSTACK IS EMPTY\n");
}
else
{
temp=top;
printf("\nDELETED ELEMENT IS %d\n",top->data);
top=top->link;
free(temp);
}
}

void main()
{
int ch;
clrscr();
while(1)
{
printf("\n\n 1.CREATE \n 2.PUSH \n 3.POP \n 4.EXIT \n");
printf("\n ENTER YOUR CHOICE : ");
scanf("%d",&ch);
```

```
switch(ch)
{
case 1:
        create();
        display();
        break;
case 2:
        push();
        display();
        break;
case 3:
        pop();
        display();
        break;
case 4:
        exit(0);
}
}
}
```

SAMPLE INPUT AND OUTPUT:

STACK
1. CREATE
2. PUSH
3. POP
4. EXIT

ENTER YOUR CHOICE : 1
ENTER THE FIRST ELEMENT : 10
10
STACK
1. CREATE
2. PUSH
3. POP
4. EXIT

ENTER YOUR CHOICE: 2
ENTER THE NEXT ELEMENT: 30
10

30
STACK
1. CREATE
2. PUSH
3. POP
4. EXIT
ENTER YOUR CHOICE: 3
DELETED ELEMENT IS 30
STACK
1. CREATE
2. PUSH
3. POP
4. EXIT
ENTER YOUR CHOICE: 3
DELETED ELEMENT IS 10
STACK
1. CREATE
2. PUSH
3. POP
4. EXIT
ENTER YOUR CHOICE: 3
STACK IS EMPTY.

Applications of Stack:

1. Direct Applications

- Page visited history in a Web browser
- Undo sequence in a text editor

2. Some of the indirect applications of stacks are:
    1. Balancing Symbols
    2. Function Calls
    3. Postfix Evaluation
    4. Infix to Postfix Conversion

Balancing Symbols

Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.

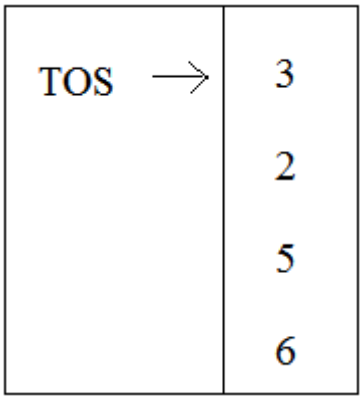
A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to their left counterparts. The sequence [()] is legal, but [(]) is wrong.

The simple algorithm uses a stack and is as follows:

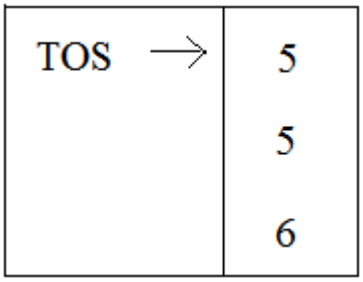
Make an empty stack. Read characters until end of file. If the character is an open anything, push it onto the stack. If it is a close anything, then if the stack is empty report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At end of file, if the stack is not empty report an error.

6 5 2 3 + 8 * + 3 + *

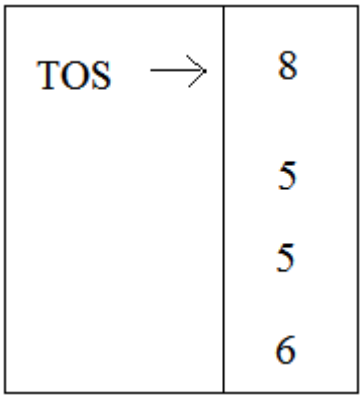
is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is



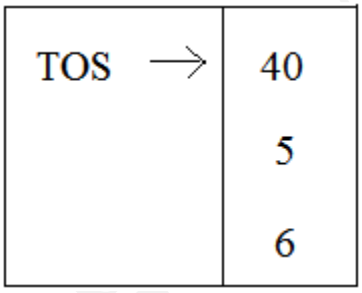
Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.



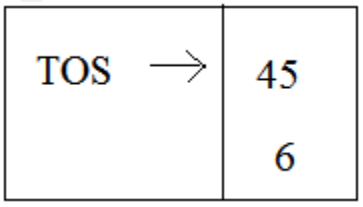
Next 8 is pushed.



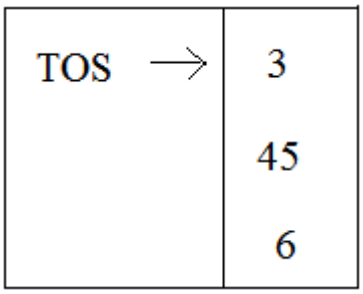
Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed.



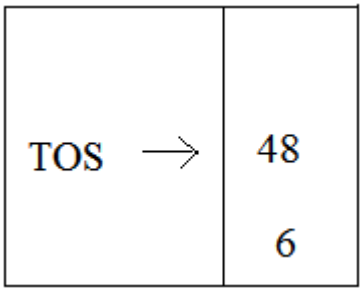
Next a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed.



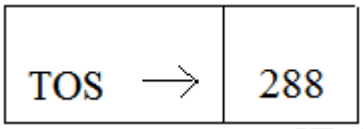
Now, 3 is pushed.



Next '+' pops 3 and 45 and pushes 45 + 3 = 48.



Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed.



When an expression is given in postfix notation, there is no need to know any precedence rules; this is an obvious advantage

INFIX to POSTFIX CONVERSION

Introduction:

In the last post I have explained about the infix, prefix, postfix expression. In this post we are going to know about the conversion of infix to postfix expression.

Infix to Postfix Conversion:

- Consider an infix string x + y * c. Each character of the string is called tokens.
- The algorithm for converting an infix expression to postfix expression is given below.

- We need a stack to solve this problem

Algorithm for converting infix expression to postfix expression:

- Initialize an empty stack and a Postfix String S.
- Read the tokens from the infix string one at a time from left to right
- If the token is an operand, add it to the Postfix string S.
- If the token is a left parentheses, Push it on to the stack
- If the token is a right parentheses
    1. Repeatedly Pop the stack and add the poped out element in to the string S until a left parenthesis is encountered;
    2. Remove the left parenthesis from the stack and ignore it
- If the token is an operator
    1. If the stack is empty push the operator in to the stack
    2. If the stack is not empty compare the precedence of the operator with the element on top of the stack
        - If the operator in the stack has higher precedence than the token Pop the stack and add the poped out element in to the string S. else Push the operator in to the stack
        - Repeat this step until the stack is not empty or the operator in the stack has higher precedence than the token
- Repeat this step till all the characters are read.
- After all the characters are read and the stack is not empty then Pop the stack and add the tokens to the string S
- Return the Postfix string S

Example for Converting infix to Postfix Expression:

Example 1:

Infix String----->A*B+C/D

Solution:

1. Initially the stack is empty and the Postfix String S has no characters
2. Read the token A.A is a operand so it is added to the string S. Now Stack->empty and S->A
3. Read the token *.* is an operator. Stack is empty so push * in to the stack. now Stack->* and S->A
4. Read the token B.B is an operand so it is added to the string S. now Stack->* and S->AB
5. Read the token +.+is an operator. Compare +and *.* has higher precedence than +. so pop * from the stack and add it to the string S. Now Stack -> empty and S->AB*.now the stack is empty so push the + in to the stack. now Stack -> + and S->AB*
6. Read the token C.C is a operand so it is added to the string S. Now Stack->+ and S->AB*C
7. Read the token /./ is an operator. compare /and +./ has higher precedence than +. so push / in to the stack . now Stack->+/ and S->AB*C
8. Read the token D.D is a operand so it is added to the string S. now Stack->+/ and S->AB*CD
9. All the characters are read but the stack is not empty so pop the stack and add the characters to the String. now S->AB*CD/+

So the Postfix String is AB*CD/+

Example 2:

Infix String----->A*(B+C)/D

Solution:

1. Initially the stack is empty and the Postfix String S has no character.
2. Read the token A.A is a operand so it is added to the string S. Now Stack->empty and S->A

3. Read the token *.* is an operator. stack is empty so push * in to the stack. now Stack->* and S->A
4. Read the token (.push the left parentheses in to the stack.
5. Read the token B.B is an operand so it is added to the string S. now Stack->* and S->AB
6. Read the token +. + is an operator. compare +and *.+ has lower precedence than *.So push + in to the stack. now Stack->*+ and S->AB
7. Read the token C.C is an operand so it is added to the string S. now Stack->*+ and S->ABC
8. Read the token).) is a right parentheses ,Pop the stack and add the + in to the string S. now Stack->* and S->ABC+
9. Read the token /./ is an operator. compare /and *./ and * have equal precedence but * comes before /so pop the stack and add * in to the string S. now stack is empty so push / in to the stack . now Stack->/ and S->ABC+*
10. Read the token D.D is an operand so it is added to the string S. now Stack->/ and S->ABC+*D
11. Now the input string is empty but the stack is not empty so pop the stack and add the characters to the String. now Stack->Empty and S->ABC+*D/

So the Postfix String is ABC+*D/

Function call

As processor executes a program, when a function call is made, the called function must know how to return back to the program, so the current address of program execution is pushed on to stack. Once the function is finished the address that was saved is removed from the stack and execution of the program resumes. If a series of function calls occur the successive return values are pushed on to the stack in LIFO order so that each function can return back to calling program. Stacks support recursive function calls.

Balancing Symbols

While compiling any program the compilers check for syntax errors. if one symbol(missing brace or comment starter)is missing then the compiler will stop compiling. so we need to checks whether everything is balanced.(ie) every right brace, bracket and parenthesis must correspond to their left counterparts. Using a stack we read all the characters until end of file if the character is an open anything push it on to the stack. If it is close anything then if the stack is empty report an error. otherwise pop the stack. If the symbol popped is not the corresponding opening symbol then report an error. It is the fast way to find an error.

- **Expression evaluation**.
- **Reversing a string**--Simply push each letter of the string on to the stack then pop them back. now the string is reversed

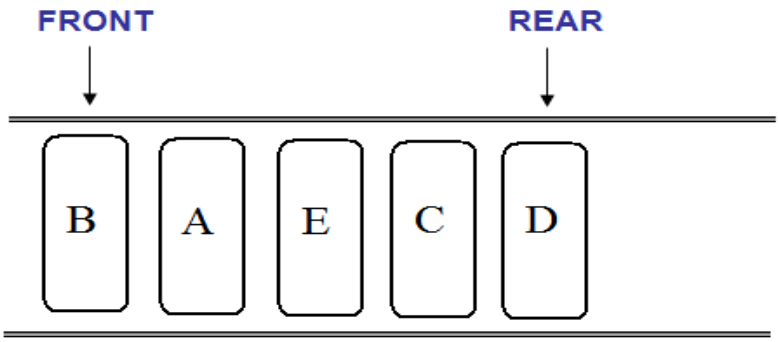
Example: STACK---->KCATS

QUEUE

Introduction:

- Consider a situation where you have to create an application with the following set of requirements:
  - Application should serve the requests of multiple users.
  - At a time, only one request can be processed.
  - The request, which came first should be given priority.
- However, the rate at which the requests are received is much faster than the rate at which they are processed.
  - Therefore, you need to store the request somewhere until they are processed.
- How can you solve this problem?
  - You can solve this problem by storing the requests in such a manner so that they are retrieved in the order of their arrival.
  - A data structure called **queue** stores and retrieves data in the order of its arrival.
  - A queue is also called a FIFO (First In First Out) list.
- Queue is a list of elements in which an element is inserted at one end and deleted from the other end of the queue.

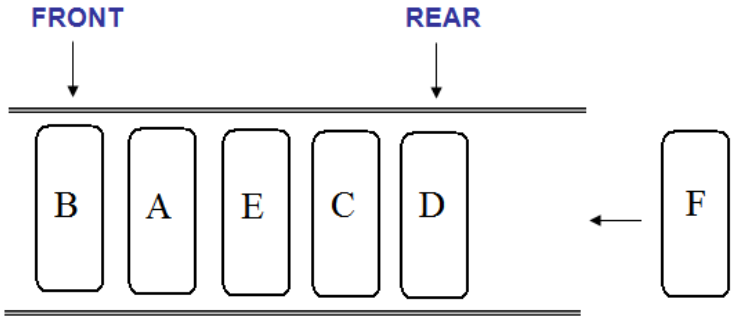
  Elements are inserted at one end called REAR end and deleted at the other end called FRONT end.
- Various operations are implemented on a queue. But the most important are:
  - Enqueue which is called inserting an element
  - Dequeue which is called deleting an element.



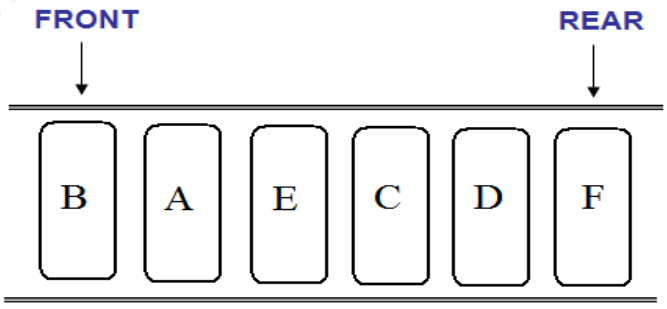
Enqueue(Insert):

It refers to the addition of an item in the queue.

- Suppose you want to add an item F in the following queue.
- Since the items are inserted at the rear end, therefore, F is inserted after D.
- Now F becomes the rear end.



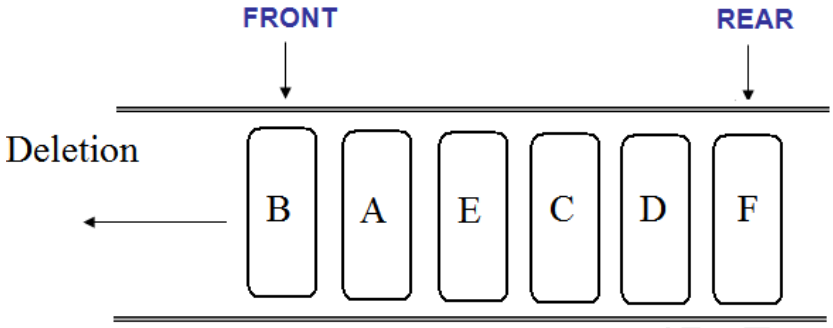
Now the REAR will move to the next position to point out the newly inserted element F. after insertion, the REAR points out F.



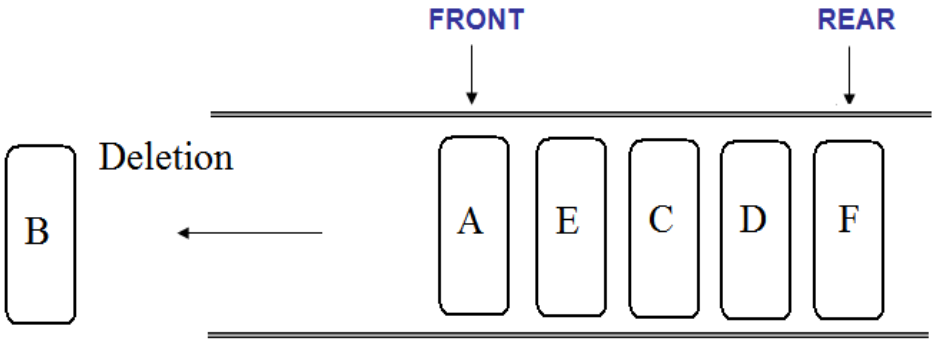
Dequeue(Delete):

It refers to the deletion of an item from the queue.

- Since the items are deleted from the front end, therefore, item B is removed from the queue.
- Now A becomes the front end of the queue.



In this Dequeue operation the FRONT end element will be removed. So the element B is removed and the value of FRONT is incremented to point out the next element A. now in this new Queue, FRONT points out A.



IMPLEMENTING A QUEUE USING AN ARRAY:

Let us implement a Queue using an array that stores the elements in the order of their arrival.

- To keep track of the rear and front positions, you need to declare two integer variables, REAR and FRONT.
- If the queue is empty, REAR and FRONT are set to –1.

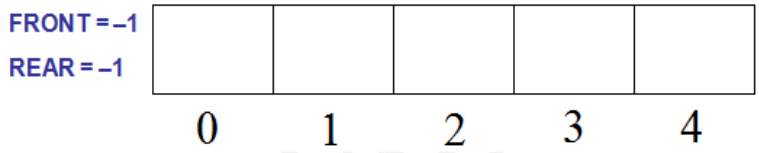
ENQUEUE:

- To insert an element, you need to perform the following steps:
- Increment the value of REAR by 1.
- Insert the element at index position REAR in the array.

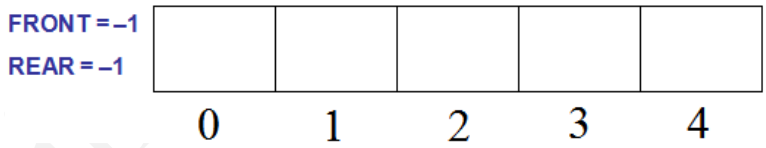
Algorithm to insert (enqueue) an element in a queue:

1. If the queue is empty:
    a. Set FRONT = 0.
2. Increment REAR by 1.
3. Store the element at index position REAR in the array.

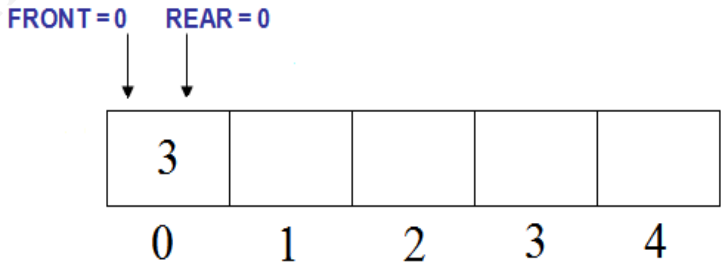
When the stack is empty both FRONT = -1 & REAR = -1



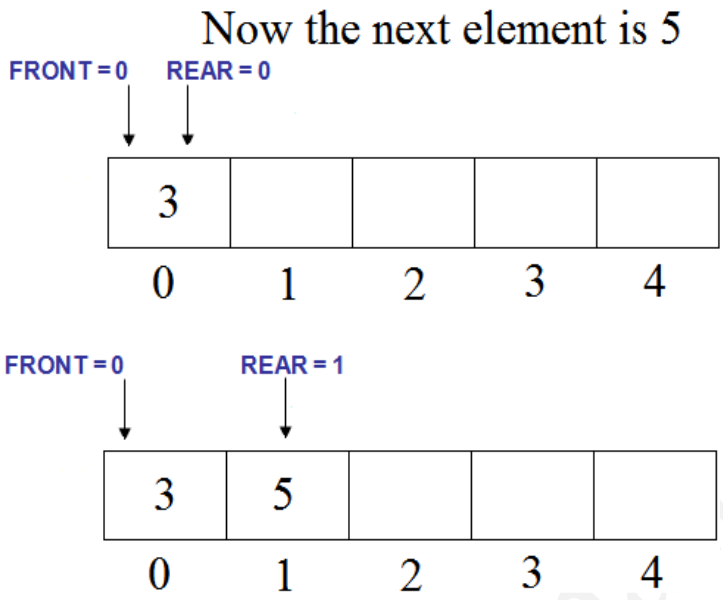
The First Element be 3



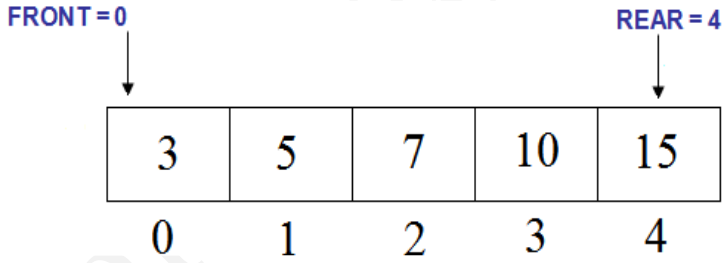
If the first element is inserted, FRONT = 0 & REAR = 0.



When the next elements are inserted only the REAR will be inserted.



Subsequent elements will also be inserted. When the last element is inserted REAR becomes 4.

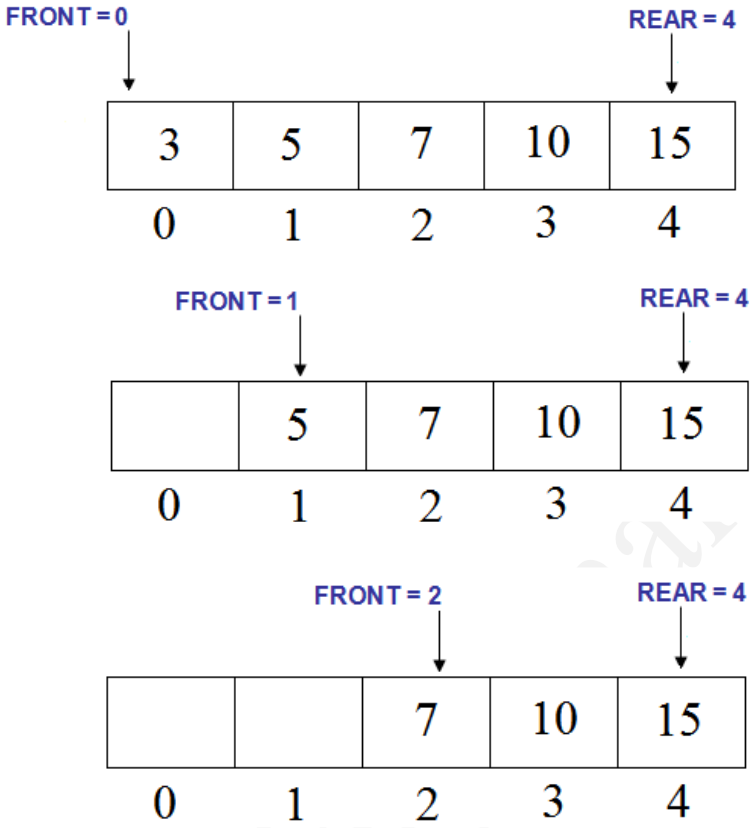


After inserting all the elements in a queue using an array, no more elements can be inserted, because in array the size is fixed (static).

DEQUEUE(delete):

- While insertion is performed in REAR end, Deletion will be done at the FRONT end.
- Algorithm to delete an element for the QUEUE;
    1. Retrieve the element at index FRONT.
    2. Increment FRONT by 1.

Let us see how elements are deleted from the queue once they get processed. When the dequeue is performed, FRONT will be incremented.



To implement an insert or delete operation, you need to increment the values of REAR or FRONT by 1 respectively.

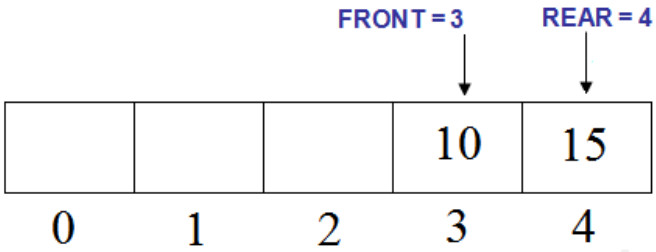
However these values are never decremented.

- As you delete elements from the queue, the queue moves down the array.
- The disadvantage of this approach is that the storage space in the beginning is discarded and never used again.

Consider the next fig.

- REAR is at the last index position.
- Therefore, you cannot insert elements in this queue, even though there is space for them.
- This means that all the initial vacant positions go waste.

If you implement a queue in the form of a linear array, you can add elements only in the successive index positions. However, when you reach the end of the queue, you cannot start inserting elements from the beginning, even if there is space for them at the beginning. You can overcome this disadvantage by implementing a queue in the form of a circular array. In this case, you can keep inserting elements till all the index positions are filled. Hence, it solves the problem of unutilized space.



- In this approach(circular array), if REAR is at the last index position and if there is space in the beginning of an array, then you can set the value of REAR to zero and start inserting elements from the beginning.(we are not touching this topic here).

- What is the **disadvantage** of implementing a queue as an array?
- To implement a queue using an array, you must know the maximum number of elements in the queue in advance.
- To solve this problem, you should implement the queue in the form of a linked list.

QUEUE USING ARRAY

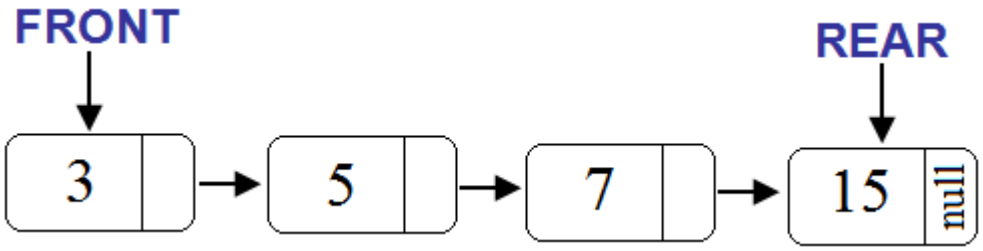
```
/******    Program to Implement Queue using Array    ******/

#include <stdio.h>
#define MAX 50
void insert();
void delet();
void display();
int queue[MAX], rear=-1, front=-1, item;
main()
{
  int ch;
  do
  {
    printf("\n\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("\nEnter your choice: ");
    scanf("%d", &ch);

    switch(ch)
    {
     case 1:
               insert();
               break;
     case 2:
               delete();
               break;
    case 3:
               display();
               break;

    case 4:
                exit(0);

    default:
                printf("\n\nInvalid entry. Please try again...\n");
    }
  } while(1);
  getch();
}

  void insert()
```

```
{
 if(rear == MAX-1)
            printf("\n\nQueue is full.");
 else
 {
  printf("\n\nEnter ITEM: ");
  scanf("%d", &item);

       if (rear == -1 && front == -1)
          {
           rear = 0;
           front = 0;
           }
       else
          rear++;

      queue[rear] = item;
  printf("\n\nItem inserted: %d", item);
 }
}
void delet()
{
 if(front == -1)
  printf("\n\nQueue is empty.");
 else
 {
  item = queue[front];

       if (front == rear)
         {
          front = -1;
          rear = -1;
         }
       else
  front++;

  printf("\n\nItem deleted: %d", item);
 }
}

void display()
{
 int i;
```

```
if(front == -1)
      printf("\n\nQueue is empty.");
else
     {
      printf("\n\n");

      for(i=front; i<=rear; i++)
      printf(" %d", queue[i]);
     }
}
```

IMPLEMENTATION OF QUEUE USING LINKED LIST:

- To keep track of the rear and front positions, you need to declare two variables/pointers, REAR and FRONT, that will always point to the rear and front end of the queue respectively.
- If the queue is empty, REAR and FRONT point to NULL.

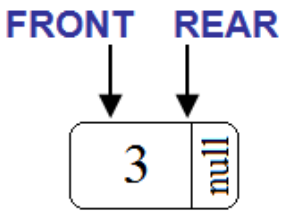


Algorithm to insert an element in QUEUE:

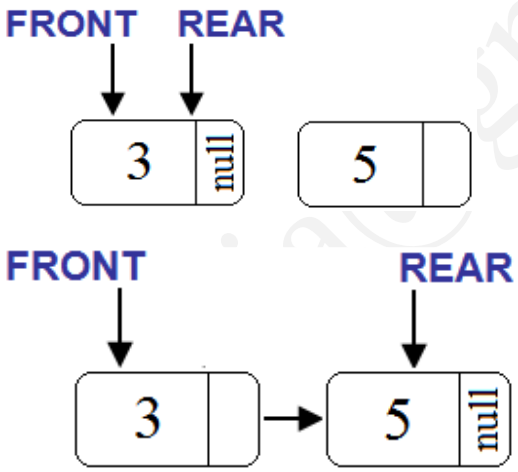
1. Allocate memory for the new node. (Using malloc)
2. Assign value to the data field of the new node.
3. Make the next field of the new node point to NULL.
4. If the queue is empty, execute the following steps:
    a. Make FRONT point to the new node
    b. Make REAR point to the new node
    c. Exit

5. Make the next field of REAR point to the new node.
6. Make REAR point to the new node.

Once the node is created using malloc value is assigned in the data field and the next filed is pointing to NULL.



When the next element is inserted, the link will be created by changing the address field of the previous node to the newly created node.

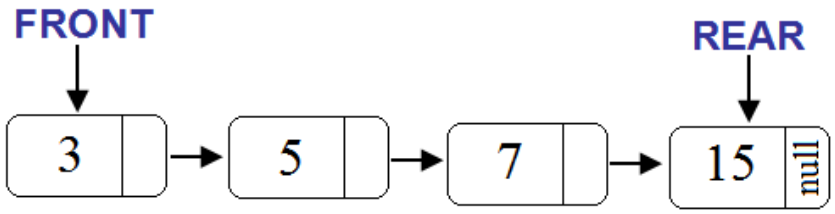


DELETION:

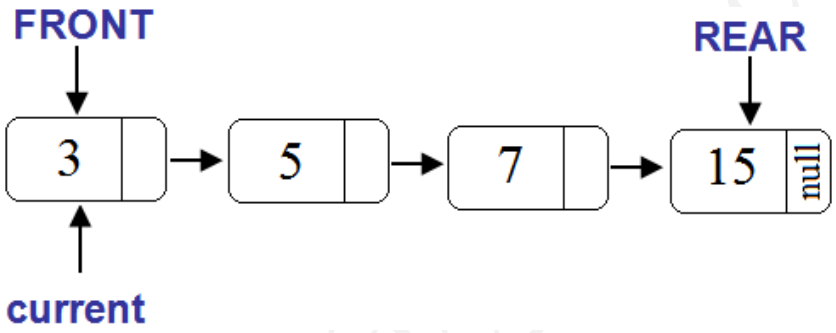
Algorithm to DELETE an element in a QUEUE:

1. If the queue is empty: // FRONT = NULL
    a. Display "Queue empty"
    b. Exit
    c. Mark the node marked FRONT as current

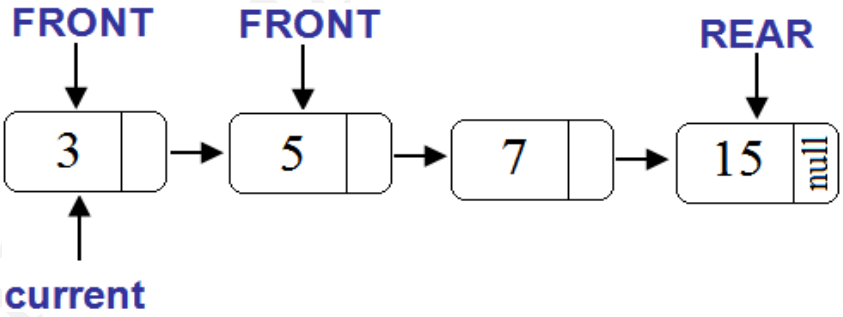
2. Make FRONT point to the next node in its sequence
3. Release the memory for the node marked as current



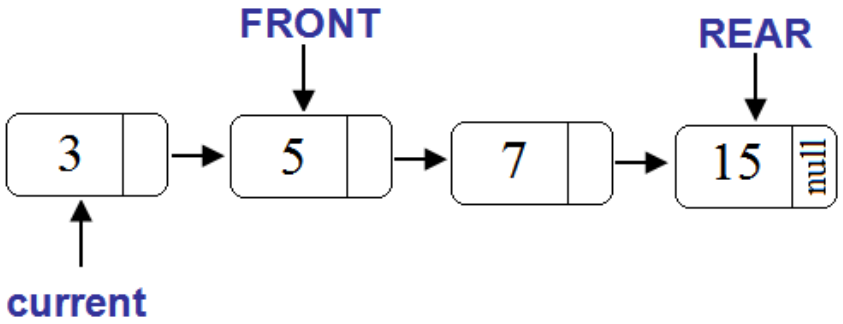
- Mark the node marked FRONT as current



- Make FRONT point to the next node in its sequence



- Release the memory for the node marked as current



In a linked list, you can insert and delete elements anywhere in the list. However, in a linked queue, insertion and deletion takes place only from the ends. More specifically, insertion takes place at the rear end and deletion takes place at the front end of the queue.

❖ Queues offer a lot of practical applications. Some of them are:

- Printer Spooling
- CPU Scheduling
- Mail Service
- Keyboard Buffering

Printer Spooling

❖ A printer may receive multiple print requests in a short span of time.

❖ The rate at which these requests are received is much faster than the rate at which they are processed.

❖ Therefore, a temporary storage mechanism is required to store these requests in the order of their arrival.

❖ A queue is the best choice in this case, which stores the print requests in such a manner so that they are processed on a first-come-first-served basis.

CPU Scheduling

❖ A CPU can process one request at a time.

- The rate at which the CPU receives requests is usually much greater than the rate at which the CPU processes the requests.
- Therefore, the requests are temporarily stored in a queue in the order of their arrival. Whenever CPU becomes free, it obtains the requests from the queue.
- Once a request is processed, its reference is deleted from the queue. The CPU then obtains the next request in sequence and the process continues.
- In a time sharing system, CPU is allocated to each request for a fixed time period.
- All these requests are temporarily stored in a queue.
- CPU processes each request one by one for a fixed time period.
- If the request is processed within that time period, its reference is deleted from the queue.
- If the request is not processed within that specified time period, the request is shifted to the end of the queue.
- CPU then processes the next request in the queue and the process continues.

Mail Service

- In various organizations, many transactions are conducted through mails.
- If the mail server goes down, and someone sends you a mail, the mail is bounced back to the sender.
- To avoid any such situation, many organizations implement a mail backup service.
- Whenever there is some problem with the mail server because of which the messages are not delivered, the mail is routed to the mail's backup server.
- The backup server stores the mails temporarily in a queue.
- Whenever the mail server is up, all the mails are transferred to the recipient in the order in which they arrived.

Keyboard Buffering

- Queues are used for storing the keystrokes as you type through the keyboard.

- Sometimes the data, which you type through the keyboard is not immediately displayed on the screen.
- This is because during that time, the processor might be busy doing some other task.
- In this situation, the data is temporarily stored in a queue, till the processor reads it.
- Once the processor is free, all the keystrokes are read in the sequence of their arrival and displayed on the screen.

LINKED LIST

- We cannot use an array to store a set of elements if you do not know the total number of elements in advance. By having some way in which you can allocate memory as and when it is required.
- When you declare an array, a contiguous block of memory is allocated.
- If you know the address of the first element in the array, you can calculate the address of the next elements.

Linked list:

- A linked list is a data structure which contains a list of elements.
- The elements of the list can be placed anywhere in memory and these elements are linked with each other using an explicit link field (ie) by storing the address of the next element.
- Linked list are used when the quantity of data is not known prior to execution.
- Data is stored in the form of nodes and at run time memory is allocated for creating nodes.
- Data is accessed using the starting pointer of the list.
- Linked list is a dynamic data structure which Allows memory to be allocated as and when it is required.

Types of Linked List:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

1. Singly Linked List:

- Singly Linked List is normally called as Linked list.
- Each node has a data field and a link field.
- Data field contains data and the link field contains the address of the next node.
- The last elements link field points to NULL.
- Inserting a node in to a singly linked list.

  To insert a new node (say X) after the node (say N),
- Create the node X (Allocate memory for the new node X, update the node's data field properly and set the link-field for the node to NULL)
- Set the link-field of node X to the same as that of node N's link-field
- Set the link-field of node N to point to X
- Deleting a specified node in a Singly Linked List:

  To delete a node(say X)
- Determine the previous node of the node X to be deleted by traversing the linked list
- Set the link field of the previous node to the same as the node X's link field

Problems with Singly Linked List

- Singly Linked list allows traversal of the list in only one direction.
- Deleting a node from a list requires keeping track of the previous node (ie) the node whose link points to the node to be deleted.
- If the link in any node gets corrupted, the remaining nodes of the list become unusable.

❖ The problems of singly linked lists can be overcome by adding one more link to each node, which points to the previous node. This type of linked list is called a Doubly Linked List.

2.Doubly Linked List:

- ❖ A Doubly Linked List is a linked list in which every node contain two links, called left link and right link
- ❖ The left link of the node points to the previous node and the right link points to the next node
- ❖ The left link of the first node and the right link of the last node will be NULL.
- ❖ Application of Doubly Linked List:
- ❖ In memory management, a doubly linked list is used to maintain both the list of allocated blocks and the list of free blocks by the memory manager of the operating system.
- ❖ A doubly linked list can be traversed in both directions.
- ❖ Having 2 pointers in a doubly linked list provides safety, because even if one of the pointers get corrupted the node still remains linked.
- ❖ Deleting a particular node from a list does not require keeping track of the previous node.

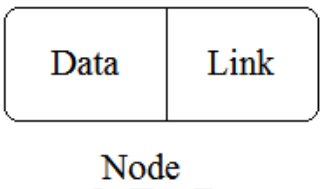
3.Circular Linked List:

- ❖ A Circular Linked list is a list in which the link field of the last node is made to point to the start/first node of the list.
- ❖ In Circular Linked List all nodes are linked in a continuous circle without using NULL.
- ❖ Circular Linked List can be either singly or doubly linked list.
- ❖ The list can be traversed fully beginning at any given node

- Application of Circular Linked List:
- Circular linked list are used to represent arrays that are naturally circular.
- for a pool of buffers that are used and released in FIFO order
- for a set of processes that should be time-shared in round robin order
- Applications that require access to both ends of the list(eg implementation of a queue)

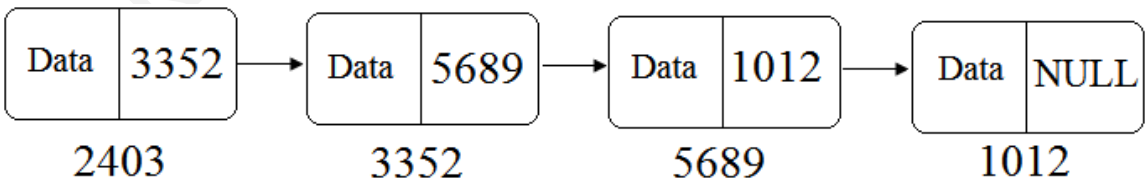
- Singly linked list consists of a chain of elements, in which each element is referred to as a node.
- A node is the basic building block of a linked list.
- A node consists of two parts:

  - **Data:** Refers to the information held by the node
  - **Link:** Holds the address of the next node in the list

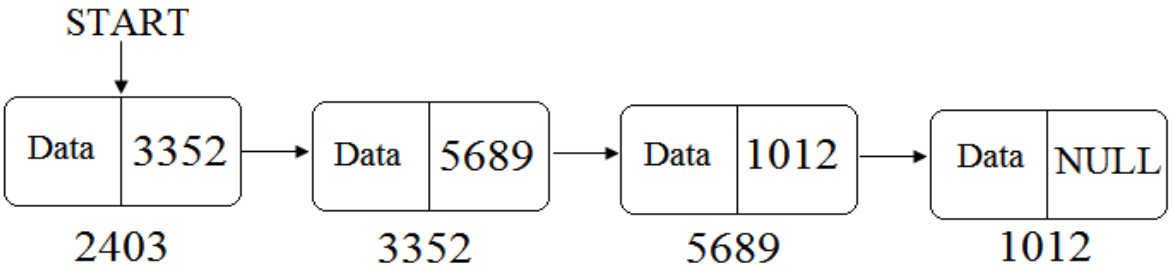


Node

- All the nodes in a linked list are present at arbitrary memory locations. Therefore every node in a linked list that stores the address of the next node in sequence.
- The last node in a linked list does not point to any other node. Therefore, it points to NULL.



- To keep track of the first node, declare a variable/pointer, START, which always points to the first node.
- When the list is empty, START contains null.

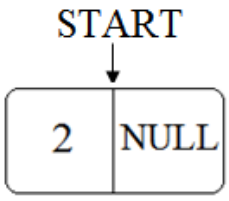


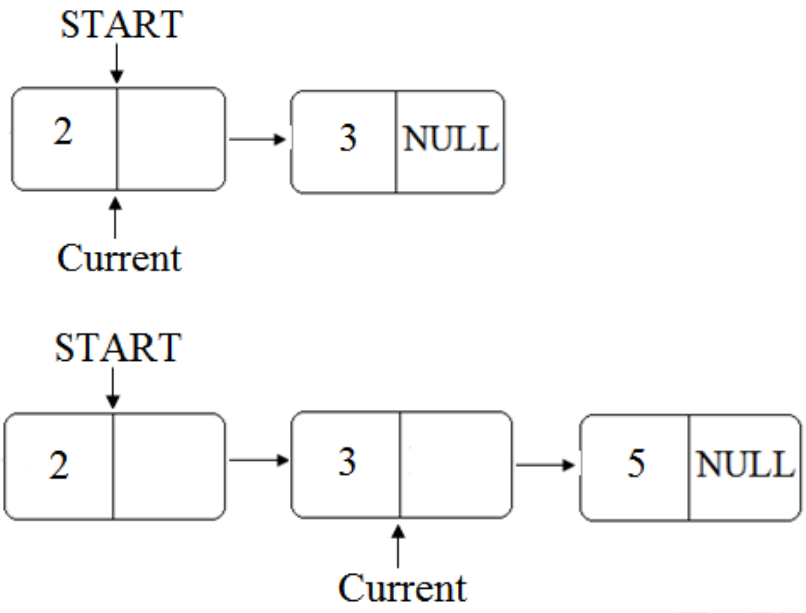
Algorithm to insert a node in a linked list.

1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. If START is NULL, then:
    a. Make START point to the new node.
    b. Go to step 6.
4. Locate the last node in the list, and mark it as current Node. To locate the last node in the list, execute the following steps:
    a. Mark the first node as current Node.
    b. Repeat step c until the successor of current Node becomes NULL.
    c. Make current Node point to the next node in sequence.
5. Make the next field of current Node point to the new node.
6. Make the next field of the new node point to NULL.

Consider that the list is initially.

START = NULL





*****Implementation of Linked List *****

```
#include <stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
void insert_first();
void display();
void insert_last();
void insert_specific();
void delete_last();
void delete_first();
void delete_specific();
struct node
{
  int data;
  struct node *next;
 } *start=NULL;

int item;
void main()
{
     int ch;
     clrscr();
```

```
    do
    {
        printf("\n\n\n1. Insert First\n2. insert last\n3. insert specific\n4. Delete first\n 5.
Delete last\n 6. Delete specific\n 7.Exit\n\n\n");
        printf("\nEnter your choice: ");
        scanf("%d",&ch);

         switch(ch)
                    {
          case 1:
                insert_first();
                display();
                break;

            case 2:
                insert_last();
                display();
                break;

            case 3:
                insert_specific();
                display();
                break;
            case 4:
                delete_first();
                display();
                break;

            case 5:
                delete_last();
                display();
                break;

            case 6:
                delete_specific();
                display();
                break;

            case 7:
                exit(0);

            default:
                printf("\n\nInvalid choice. Please try again.\n");
```

```
            }
        } while (1);
}

void insert_first()
{
  struct node *ptr;

  printf("\n\nEnter item: ");
  scanf("%d", &item);

  if(start == NULL)
  {
    start = (struct node *)malloc(sizeof(struct node));
    start->data = item;
    start->next = NULL;
  }
  else
  {
    ptr= start;
    start = (struct node *)malloc(sizeof(struct node));
    start->data = item;
    start->next = ptr;
  }

  printf("\nItem inserted: %d\n", item);
}


void insert_last()
{
  struct node *ptr;
  printf("\n\nEnter item: ");
  scanf("%d", &item);

  if(start == NULL)
  {
    start = (struct node *)malloc(sizeof(struct node));
    start->data = item;
    start->next = NULL;
  }
  else
  {
```

```
            ptr = start;
            while (ptr->next != NULL)
            ptr = ptr-> next;

            ptr-> next = (struct node *)malloc(sizeof(struct node));
            ptr = ptr-> next;
            ptr-> data = item;
            ptr-> next = NULL;
  }

  printf("\nItem inserted: %d\n", item);
}

void insert_specific()
{
    int n;
    struct node *nw, *ptr;

    if (start == NULL)
          printf("\n\nNexted list is empty. It must have at least one node.\n");
    else
    {
          printf("\n\nEnter DATA after which new node is to be inserted: ");
          scanf("%d", &n);
          printf("\n\nEnter ITEM: ");
          scanf("%d", &item);

          ptr = start;
          nw = start;

          while (ptr != NULL)
          {
              if (ptr->data == n)
              {
                    nw = (struct node *)malloc(sizeof(struct node));
                nw->data = item;
                nw->next = ptr->next;
                    ptr->next = nw;
                printf("\n\nItem inserted: %d", item);
                return;
             }
             else
                    ptr = ptr->next;
```

```
        }
    }
}

void display()
{
 struct node *ptr = start;
 int i=1;

 if (ptr == NULL)
  printf("\nNextlist is empty.\n");
 else
 {
    printf("************************************");
    printf("\nSr. No.\t\tAddress\t\tData\t\tNext\n");
    while(ptr != NULL)
         {
           printf("\n%d. \t\t%d  \t\t%d  \t\t%d \n", i, ptr, ptr->data, ptr->next);
         ptr = ptr->next;
         i++;
     }
     printf("************************************");
 }
}

void delete_first()
{
    struct node *ptr;

    if (start == NULL)
        printf("\n\nNexted list is empty.\n");
    else
    {
        ptr = start;
        item = start->data;
        start = start->next;
        free(ptr);
        printf("\n\nItem deleted: %d", item);
    }
}

void delete_last()
{
```

```
    struct node *ptr, *prev;

    if (start == NULL)
        printf("\n\nNexted list is empty.\n");
    else
    {
      ptr = start;
        prev = start;
        while (ptr->next != NULL)
        {
            prev = ptr;
            ptr = ptr->next;
        }

        item = ptr->data;

        if (start->next == NULL)
            start = NULL;
        else
            prev->next = NULL;

        prev->next = NULL;
        free(ptr);
        printf("\n\nItem deleted: %d", item);
    }
}

void delete_specific()
{
    struct node *ptr, *prev;

    printf("\n\nEnter ITEM which is to be deleted: ");
    scanf("%d", &item);


    if (start == NULL)
      printf("\n\nNexted list is empty.\n");
else if (start->data == item)
 {
     ptr = start;
     start = start->next;
     free(ptr);
 }
```

```
else
{
    ptr = start;
    prev = start;

    while (ptr != NULL)
    {
        if (ptr->data == item)
        {
            prev->next = ptr->next;
            free(ptr);
        }
        else
        {
            prev = ptr;
            ptr = ptr->next;
        }
    }
    printf("\n\nItem deleted: %d", item);
   }
}
```