CS302ES: DATA STRUCTURES

B.TECH II Year I Sem. **L T P C**
3 1 0 4

Prerequisites: A course on "Programming for Problem Solving".

Course Objectives:

- Exploring basic data structures such as stacks and queues.
- Introduces a variety of data structures such as hash tables, search trees, tries, heaps, graphs.
- Introduces sorting and pattern matching algorithms

Course Outcomes:

- Ability to select the data structures that efficiently model the information in a problem.
- Ability to assess efficiency trade-offs among different data structure implementations or combinations.
- Implement and know the application of algorithms for sorting and pattern matching.
- Design programs using a variety of data structures, including hash tables, binary and general tree structures, search trees, tries, heaps, graphs, and AVL-trees.

UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

UNIT - II

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

UNIT - III

Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

UNIT - IV

Graphs: Graph Implementation Methods. Graph Traversal Methods.

Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort.

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm,
the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

TEXTBOOKS:

1. Fundamentals of Data Structures in C, 2nd Edition, E. Horowitz, S. Sahni and Susan Anderson Freed, *Universities Press*.

2. Data Structures using C – A. S. Tanenbaum, Y. Langsam, and M.J. Augenstein, *PHI/Pearson Education*.

REFERENCE BOOKS:

1. Data Structures: A Pseudocode Approach with C, 2nd Edition, R. F. Gilberg and B.A. Forouzan, Cengage Learning.

Unit I

Introduction to Data Structures:

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure.

The data structure can be subdivided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structures:

A data structure is said to be linear if its elements combine to form any specific order. There are two techniques of representing such linear structure within memory.

- The first way is to provide the linear relationships among all the elements represented using linear memory location. These linear structures are termed as arrays.
- The second technique is to provide a linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of the linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non-Linear Data Structures:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non-Linear Data Structures are listed below:

- Graphs
- Trees

Linear List Data Structures:

Each instance of the data structure linear list is an ordered collection of elements. Each instance is of the form ($e_{0,}e_{1, \dots ,}e_{n-1}$) where n is a fine natural number; the e_i items are the elements of the list; the index of e_i is I; and n is the list length or size. When n=0 the list is

empty. When n>0 , e_0 is the front element and e_{n-1} is the last element of the list. Operations on a linear list

- Create a linear list
- Destroy a linear list.
- Determine whether the list is empty.
- Determine the size of the list.
- Find the element with a given index.
- Find the index of a given element.
- Delete a given element.
- Insert a new element in given index.

Abstract Data Type of *Linear list*:

AbstractDataType *linear list*{

Instances

Ordered finite collection of zero or more elements

Operations

empty(): return true if the list is empty, false otherwise

size(): return the list size

get(index): return the element of given index of the list

indexOf(x): return the index of the first occurrence of x in the lsit, return -1 if x is not in the list

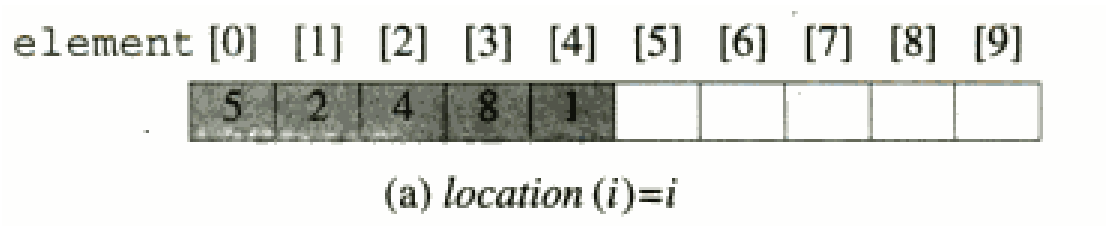
delete(index): remove the indexth element, elements with higher index have their index reduced by 1

insert(index, x): insert x as the indexth element, elements with index>= index have their index increased by 1

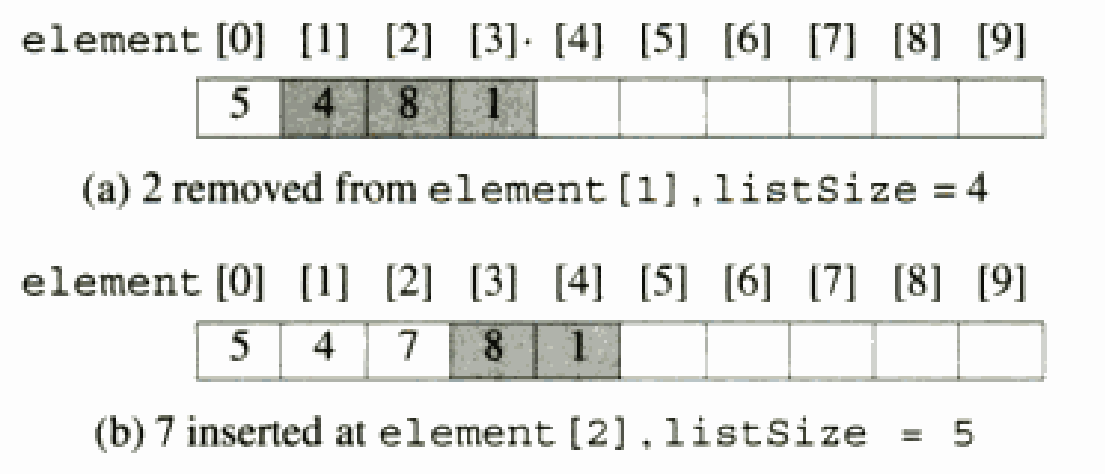
output(): output the list elements form left to right

}

Array Representation of Linear List

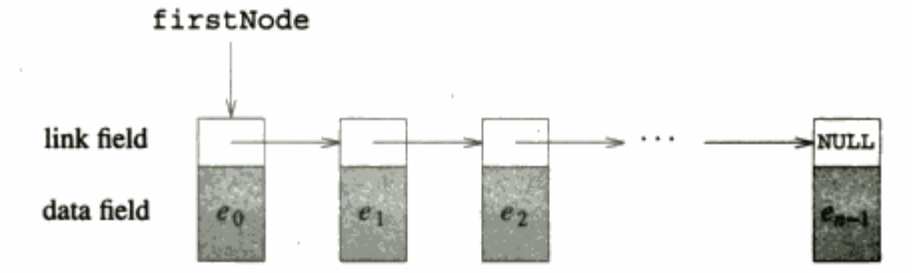


(a) *location* $(i)=i$



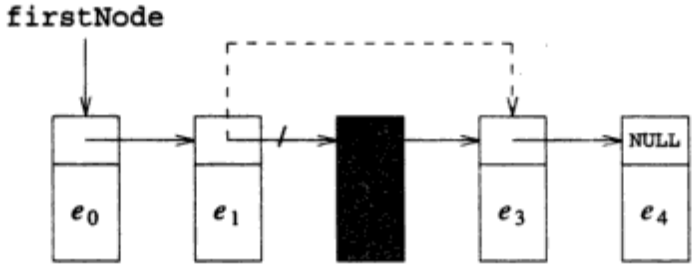
Singly Linked List:

In a linked representation, each element of an instance of data object this represented in a cell or node. The nodes, however, need not be components of an array, and no formula is used to locate individual elements. Instead, each node keeps explicit information about the location of other relevant nodes. This explicit information about the location of another node is called a **link** or **pointer.** The variable **head or firstNode** locates the first node in the representation.



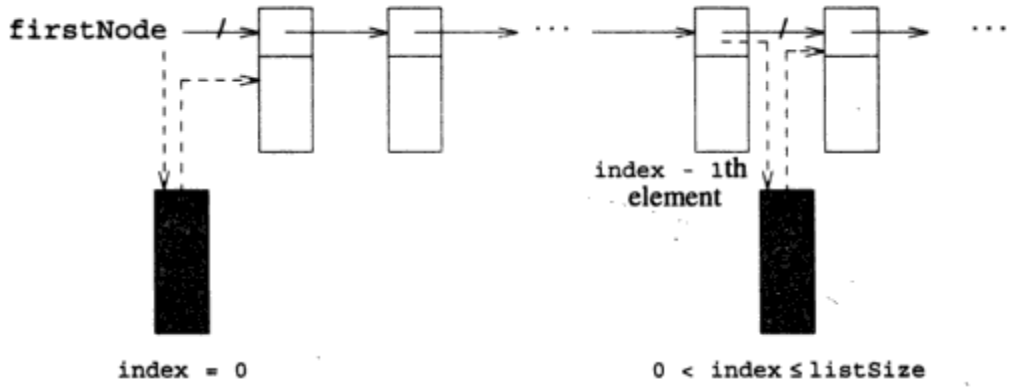
To remove the element e_2 whose index of 2 from the chain, we do the following

- Locate the second node in the chain.
- Link the second node to the fourth node.



Removing e_2 from a 5-node chain

To insert a new element as the indexth element in a chain, we need to first locate the index-1 th element and then insert a new node just after it. Below figure show the lind changes needed for the tow cases index = 0 and 0 < index <= listSize. Solid pointers exist proor to the insert, and those shown as broken (or dashed) lines exist following the inset.




```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct list
{
        int data;
        struct list *link;
};
typedef struct list node;
node *start;
void append();
void display();
void add_beg();
void insert();
void deletenode();

int main()
{
        int ch;
        while(1)
        {

                printf("\n1.append\n2.display\n3.Add beg\n4.insert\n5.delete\n6.exit");
                printf("\nEnter your choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
```

```
                        append();
                        break;
                case 2:
                        display();
                        break;
                case 3:
                        add_beg();
                        break;
                case 4:
                        insert();
                        break;
                case 5:
                        deletenode();
                        break;
                case 6:
                        exit(0);
            }
        }
        return 0;
}

void append()
{
int n;
node *cur;
cur=start;
printf("Enter number :");
scanf("%d",&n);
if(start==NULL)
{
cur=(node*)malloc(sizeof(node));
cur->data=n;
cur->link=NULL;
start=cur;
}
else
{
while(cur->link!=NULL)
cur=cur->link;

cur->link=(node*)malloc(sizeof(node));
cur->link->data=n;
cur->link->link=NULL;
}
```

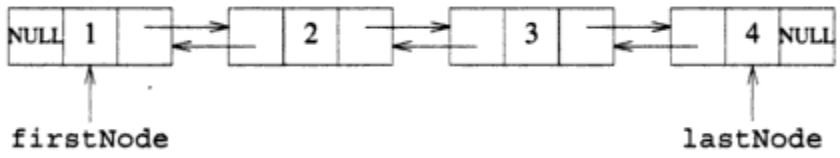
```
}
void display()
{
node *cur;
cur=start;
while(cur!=NULL)
{
printf("%d  ",cur->data);
cur=cur->link;
}
getch();

}

void add_beg()
{
node *cur;
int n;
printf("Enter element ;");
scanf("%d",&n);
cur=(node*)malloc(sizeof(node));
cur->data=n;
cur->link=start;
start=cur;
}

void insert()
{
node *cur,*temp;
int i,loc,n;
cur=start;
int flag=0;
printf("Enter element and location :");
scanf("%d%d",&n,&loc);
for(i=1;i<loc;i++)
{
cur=cur->link;
if(cur==NULL)
{
printf("Location not found ");
getch();
flag=1;
}
}
```

```
if(flag==0){
temp=(node*)malloc(sizeof(node));
temp->link=cur->link;
temp->data=n;
cur->link=temp;
}
}
void deletenode()
{
int n,flage=0;
node *cur,*temp;
cur=start;
temp=cur;
printf("Enter element to delete :");
scanf("%d",&n);
while(temp!=NULL)
{
if(temp->data==n)
{
if(temp==start)
{
start=temp->link;
free(temp);
}
else
{
cur->link=temp->link;
free(temp);
}
flage=1;
break;
}
else
{
cur=temp;
temp=temp->link;
}
}
if(!flage)
{
printf("Element %d is not found ",n);
getch();
}
}
```

Doubly Linked List:

A doubly linked list is an ordered sequence of nodes in which each node has two pointers **next** and **previous**. The **previous** pointer points to the node on the left and the **next** pointer points to the node on the right.

In a doubly linked list first node **previous** pointer contains NULL and last node **next** pointer contains NULL.




```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct list
{
struct list *previous;
int data;
struct list *next;
};
typedef struct list node;
node *start;

void add();
void display();
void count();
void deletenode();
void add_beg();
void insert();
void displayrev();

int main()
{
int ch;

while(1)
{
```

```
printf("\n1.Adding\n2.display\n3.count\n4.add beg\n5.add after\n6.display
reverse\n7.delete\n8.Exit\n");
printf("\nEnter your choice :");
scanf("%d",&ch);
switch(ch)
{
case 1:
add();
break;
case 2:
display();
break;
case 3:
count();
break;
case 4:
add_beg();
break;
case 5:
insert();
break;
case 6:
displayrev();
break;
case 7:
deletenode();
break;
case 8:
exit(0);
}
}
return 0;
}

void add()
{
node *cur;
node *pre;
int n;
printf("Enter element :");
scanf("%d",&n);

if(start==NULL)
{
```

```
cur=(node*)malloc(sizeof(node));
cur->previous=NULL;
cur->data=n;
cur->next=NULL;
start=cur;
}
else
{
cur=start;
while(cur->next!=NULL)
cur=cur->next;

pre=(node*)malloc(sizeof(node));
pre->data=n;
pre->next=NULL;
cur->next=pre;
pre->previous=cur;
}
}
void display()
{
node *cur;
cur=start;
while(cur!=NULL)
{
printf("%d  ",cur->data);
cur=cur->next;
}
printf("---->NULL");
getch();
}

void count()
{
node *cur;
int c=0;
cur=start;

while(cur!=NULL)
{
cur=cur->next;
c++;
}
printf("No of nodes are  %d ",c);
```

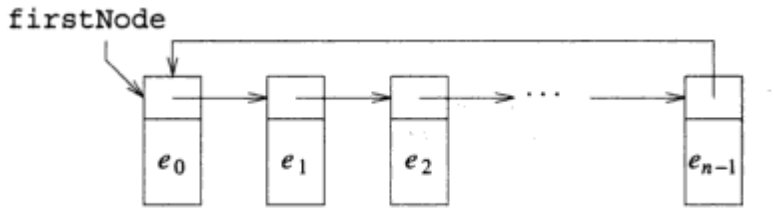
```
getch();
}

void add_beg()
{
node *cur;
int n;
cur=(node*)malloc(sizeof(node));
printf("Enter element to add at the Begining :");
scanf("%d",&n);
cur->data=n;
cur->next=start;
cur->previous=NULL;
start->previous=cur;
start=cur;
}

void insert()
{
node *temp,*cur;
int n,loc,i;
cur=(node*)malloc(sizeof(node));
cur=start;
printf("Enter the element and location :");
scanf("%d%d",&n,&loc);
for(i=1;i<loc;i++)
{
cur=cur->next;
if(cur==NULL)
{
printf("Location not found");
getch();
return;
}
}
temp=(node*)malloc(sizeof(node));
temp->data=n;
temp->next=cur->next;
temp->previous=cur;
temp->next->previous=temp;
cur->next=temp;
}

void displayrev()
```

```
{
node *cur;
cur=start;
while(cur->next!=NULL)
{
cur=cur->next;
}
while(cur!=NULL)
{
printf("%d  ",cur->data);
cur=cur->previous;
}
getch();
}
void deletenode()
{
int n;
node *cur;
cur=start;
printf("Enter element to delete :");
scanf("%d",&n);
while(cur!=NULL)
{
if(cur->data==n)
{
if(cur==start)
{
cur->next->previous=NULL;
start=cur->next;
free(cur);
}
else
if(cur->next==NULL)
{
cur->previous->next=NULL;
free(cur);
}
else
{
cur->previous->next=cur->next;
cur->next->previous=cur->previous;
free(cur);
}
}
```

```
cur=cur->next;
}
}
```

Circular Linked List:

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. In the circular linked list last node link contains the address of the first node. A circular linked list can be a singly circular linked list or doubly circular linked list.



(a) Circular list

Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4) Circular Doubly Linked Lists are used for implementation of advanced data structures like **Fibonacci Heap**.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct list
{
        int data;
        struct list *link;
};
typedef struct list node;
node *start;
void append();
void display();
```

```
void count();
void add_beg();
void insert();
void deletenode();

int main()
{
	int ch;
	while(1)
	{

		printf("1.Append\n2.count\n3.display\n4.insert\n5.add
after\n6.delete\n7.exit");
		printf("\nEnter your choice:");
		scanf("%d",&ch);
		switch(ch)
		{
			case 1:
				append();
				break;
			case 2:
				count();
				break;
			case 3:
				display();
				break;
			case 4:
				add_beg();
				break;
			case 5:
				insert();
				break;
			case 6:
				deletenode();
				break;
			case 7:
				exit(0);
		}
	}
	return 0;
}

void append()
{
	int n;
	node *cur;
	printf("Enter number :");
	scanf("%d",&n);
	if(start==NULL)
	{
```

```
                cur=(node*)malloc(sizeof(node));
                cur->data=n;
                cur->link=cur;
                start=cur;
        }
        else
        {
                cur=start;
                while(cur->link!=start)
                        cur=cur->link;

                cur->link=(node*)malloc(sizeof(node));
                cur->link->data=n;
                cur->link->link=start;
        }
}
void display()
{
        node *cur;
        cur=start;
        do{
                printf("%d  ",cur->data);
                cur=cur->link;
        }while(cur!=start);
        getch();

}
void count()
{
        int c=0;
        node *cur;
        cur=start;
        do{
                cur=cur->link;
                c++;
        }while(cur!=start);
        printf("\nThe no of nodes are %d",c);
        getch();
}
void add_beg()
{
        node *cur;
        node *pre;
        int n;
        printf("Enter element ;");
        scanf("%d",&n);
        cur=(node*)malloc(sizeof(node));
        cur->data=n;
        cur->link=start;
        pre=start;
```

```
	while(pre->link!=start)
		pre=pre->link;

	pre->link=cur;
	start=cur;
}

void insert()
{
	node *cur,*temp;
	int i,loc,n,flag=0;
	cur=start;
	printf("Enter element and location :");
	scanf("%d%d",&n,&loc);
	for(i=1;i<loc;i++)
	{
		cur=cur->link;
		if(cur==start)
		{
			printf("Location not found ");
			flag=1;
			getch();
		}
	}
	if(flag==0)
	{
		temp=(node*)malloc(sizeof(node));
		temp->link=cur->link;
		temp->data=n;
		cur->link=temp;
	}
}
void deletenode()
{
	int n,flag=0;
	node *cur,*temp,*pre;
	cur=pre=start;
	temp=cur;
	printf("Enter element to delete :");
	scanf("%d",&n);
	do{
		if(temp->data==n)
		{
			flag=1;
			if(temp==start)
			{
				while(pre->link!=start)
				pre=pre->link;
				start=temp->link;
```

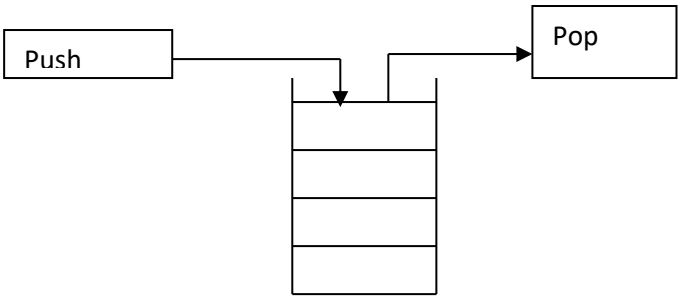
```
                        free(temp);
                        pre->link=start;
                        break;
                }
                else
                {
                        cur->link=temp->link;
                        free(temp);
                        break;
                }

        }
        else
        {
                cur=temp;
                temp=temp->link;
        }
    }while(temp!=start);
    if(!flag)
    {
        printf("Element %d is not found ",n);
        getch();
    }
}
```

Stack:

The stack is one of the most widely used data structures. It plays a very important role in programming language.

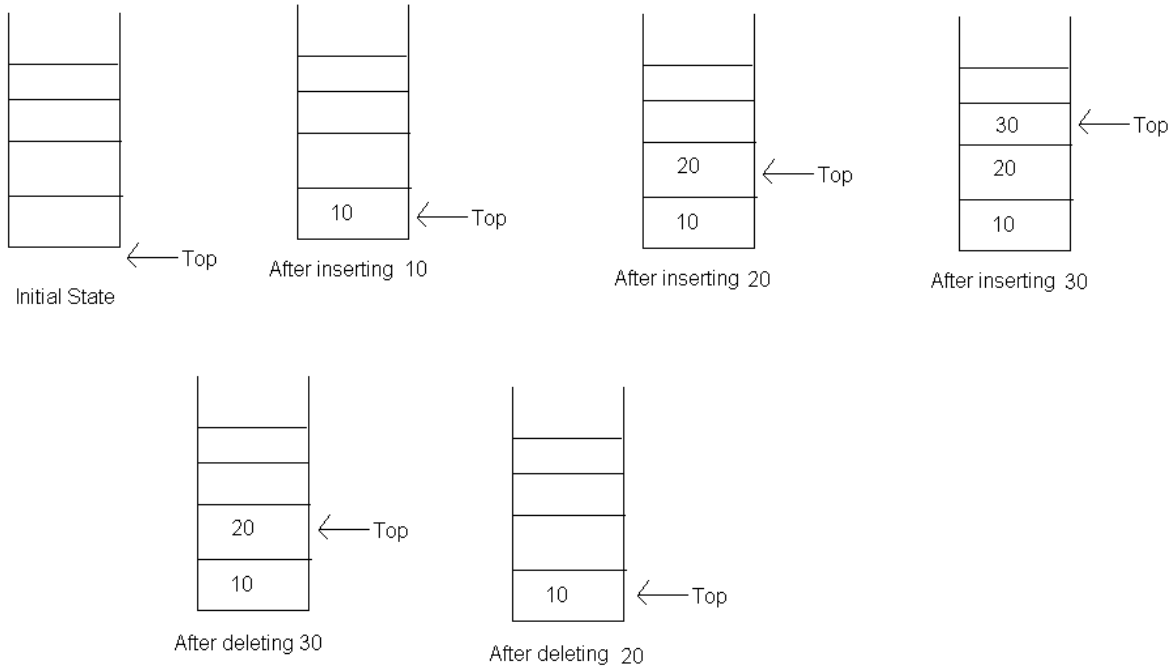
Definition:

A Stack can be defined as an ordered collection of items into which items may be added or deleted at the end only, called the top of the stack. According to the definition a stack is dynamic, constantly changing object where access to items is only restricted to the top we can illustrate stack as follows



Here, if any new items are to added they will be placed on top of the stack, and if any item is

deleted it remove the first one. Such a concept used in stacks is referred to as Last In First Out (LIFO) operation.



The simplest way to represent a stack is by using a one-dimensional Array say *stack*[0 : n-1], where *n* is the maximum number of allowable entries.

Another way to represent a stack is by using links. A *node* is a collection of data and link information. A stack can be represented using nodes with two fields, possibly called *data* and *link.*

AbstractDataType *stack{*

Instances

Linear list of elements; one end is called the *bottom;* the other is the *top*

Operations

empty() : Return **true** if the stack is empty. Return **false** otherwise;
size() : Return the number of elements in the stack;
t*op() :* Return the top element of the stack;
pop() : Remove the top element from the stack;
push(x) : Add element x at the top of the stack;

}

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
# define SIZE 10
```

```
int stack[SIZE];
int top=-1;
void push(int);
int pop();
void display();

int main()
{

char c;
int ch;
int element;
while(1)
{

printf("\nMENU\n");
printf("1.Push\n2.Display\n3.Pop\n4.Exit");
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("Enter element :");
scanf("%d",&element);
push(element);
break;
case 2:
display();
break;
case 3:
printf("\nPoped element :%d",pop());
getch();
break;
case 4:
exit(0);
}
}
return 0;
}
void push(int element)
{
if(top>=SIZE-1)
{
printf("Stack overflow");
```

```
getch();
}
else
{
top+=1;
stack[top]=element;
}
}

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

void display()
{
int i;
printf("Elements in Stack:\n");
for(i=0;i<=top;i++)
printf("%d  ",stack[i]);
getch();
}
```

INFIX, POSTFIX AND PREFIX

This application that illustrates the different types of stacks and various operations and functions defined upon them.

Consider the sum of A and B. We think of applying the **operator "+"** to the **operands** Aand B and white the sum as A+B. this particular representation is called **infix.** There are two alternate notations for expressing the sum of Aand B using the symbols A,B, and +. These are

+AB prefix

AB+ postfix

The prefixes “pre”, post”, and “in” refer to the relative position of the operator with respect to the two operands. In prefix notation the operator precedes the two operand. In postfix notation the perator follows the two operands, and in infix notation the operator is between the two operands. The prefix ad postfix notations are not really as awkward to use as they might at first appears. For example, a C function to return the sum o9f the tow arguments A and B is invoked by *add(A,B)*. The operator *add* precedes the operands A and B.

We consider five binary operations: addition, subtraction, multiplication, division and exponentiation. The first four are available in C and are dented by the usual operators +, -, * and /. The fifth, exponentiation is represented by the operator $. The value of the expression *A$B* is a raised to the B power, so that 3$2 is 9. For these binary operators the following is the order of precedence (highest to lowest)

Exponentiation

Multiplication/division

Addition/subtraction

Examples:

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$	$-*+ABC-DE+FG
A-B/(C*D$E)	ABCDE$*/-	-A/B*C$DE

Infix to Postfix Conversion:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXCOLS 80
void postfix(char*,char*);
int isoperand(char);
void popandtest(struct stack*,char *,int*);
int prcd(char,char);
void push(struct stack*,char);
char pop(struct stack*);
```

```
void main()
{
char infix[MAXCOLS];
char postr[MAXCOLS];
int pos=0;
clrscr();
printf("Enter infix string :\n");
gets(infix);

printf("The original infix expression is %s",infix);
postfix(infix,postr);
printf("\n%s",postr);
getch();
}

struct stack
{
int top;
char items[MAXCOLS];
};

void postfix(char infix[],char postr[])
{
int position,und;
int outpos=0;
char topsymb='+';
char symb;
struct stack opstk;
opstk.top=-1;
for(position=0;(symb=infix[position])!='\0';position++)
if(isoperand(symb))
postr[outpos++]=symb;
else
{
popandtest(&opstk,&topsymb,&und);
while(!und && prcd(topsymb,symb))
{
postr[outpos++]=topsymb;
popandtest(&opstk,&topsymb,&und);
}
if(!und)
push(&opstk,topsymb);
if(und || (symb!=')'))
push(&opstk,symb);
```

```
else
topsymb=pop(&opstk);
}
while(!(opstk.top==-1))
postr[outpos++]=pop(&opstk);
postr[outpos]='\0';
}

void push(struct stack *ps,char x)
{
if(ps->top==MAXCOLS-1)
printf("Stack overflow");
else
ps->items[++(ps->top)]=x;
}

char pop(struct stack *ps)
{
if(ps->top==-1)
{
printf("Stack underflow");
exit(1);
}
else
return (ps->items[ps->top--]);
}

int isoperand(char op)
{
if( op!='+' && op !='-' && op!='*' && op!= '/' && op!='$' && op!='(' && op!=')')
return 1;
else
return 0;
}

int prcd(char op1,char op2)
{
if(op1== '('  || op2=='(')
return 0;
if(op2==')')
return 1;
if(op1==')')
{
printf("Attempt to compare the two indicatres an error");
```

```
exit(1);
}
if(op1=='$' && op2!='$')
return 1;
else
if((op1=='*' || op1=='/') && (op2=='+' || op2=='-'))
return 1;
else
if(op1=='*' || op1=='/')
return 1;
else
if( (op1=='+' || op1=='-') && (op2!='*' && op2!='/') )
return 1;
else

return 0;
}
void popandtest(struct stack *ps,char *px,int *pund)
{
if(ps->top==-1)
{
*pund=1;
return;
}
*pund=0;
*px=ps->items[ps->top--];
return;
}
```

Postfix evaluation:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define MAXCOLS 80

double eval(char[]);
double pop(struct stack *);
void push(struct stack*,double);
double oper(int,double,double);

void main()
{
```

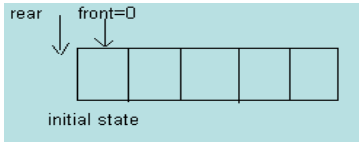
```
char expr[MAXCOLS];
int position=0;
clrscr();
printf("Enter expression :\n");

gets(expr);

printf("%s %s","the original postfix expression is ",expr);
printf("\n Result = %lf",eval(expr));
getch();
}

struct stack{
int top;
double items[MAXCOLS];
};

double eval(char expr[])
{
int c,position;
double opnd1,opnd2,value;
struct stack opndstk;
opndstk.top=-1;
for(position=0;(c=expr[position])!='\0';position++)
if(isdigit(c))
push(&opndstk,(double)(c-'0'));
else
{
opnd2=pop(&opndstk);
opnd1=pop(&opndstk);
value=oper(c,opnd1,opnd2);
push(&opndstk,value);
}
return(pop(&opndstk));
}

double oper(int symb,double op1,double op2)
{
switch(symb)
{
case '+':  return (op1+op2);
case '-':  return (op1-op2);
case '*':  return (op1*op2);
case '/':  return (op1/op2);
```

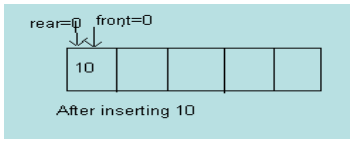
```
case '$':  return (pow(op1,op2));
default: printf("illegal operation");
exit(1);
}
}


void push(struct stack *ps,double x)
{
if(ps->top==MAXCOLS-1)
printf("Stack overflow");
else
ps->items[++(ps->top)]=x;
}

double pop(struct stack *ps)
{
if(ps->top==-1)
{
printf("Stack underflow");
exit(1);
}
else
return (ps->items[ps->top--]);
}
```

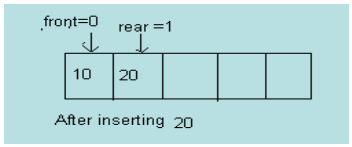
Queue:

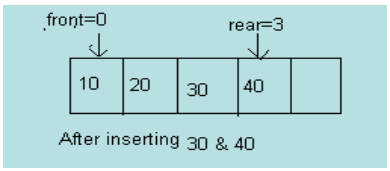
Queues are another part of data structure where will be accept the values and arrange them in array. Whenever we want to display the values the values will be displayed in FIFO (Fist In Fist Out) method and whenever we want to delete the values will be delete in FIFO form methods are use. Queues are used to solve many of the problems in computers. Most common application of queues in computes is for scheduling of jobs.

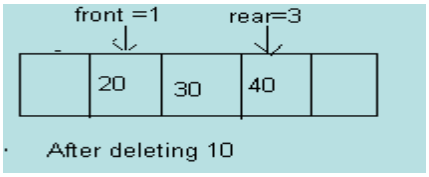
Definition: A Queue can be defined as an ordered collection of items in which the first element added is also the first element to be removed. Hence it is also called as Fist In Fist Out structure.
In Linear Queue initially *front=0* and *rear=-1*. To insert an element increase rear by 1 and insert element in rear position. If rear = *n-1* queue is full. To delete an element increase front by 1. If front > rear Queue is empty.











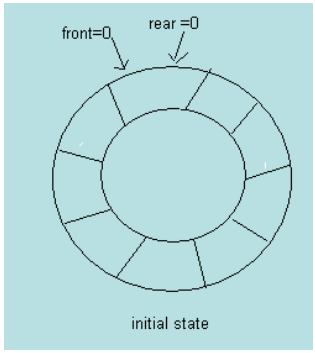

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

#define SIZE 5

int que[SIZE];
int front,rear;
void enqueue(int);
int dequeue();
void display();

int main()
{
int element;
int ch;

front=0;
rear=-1;


while(1)
{

printf("\nMENU\n");
printf("1.insert\n2.delete\n3.display\n4.exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
switch(ch)
{
case 1:
```

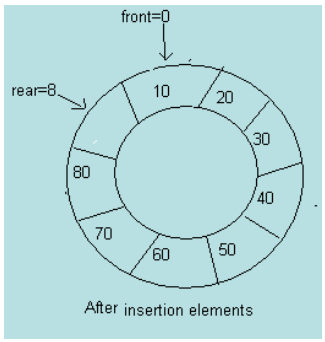
```
printf("Enter element :");
scanf("%d",&element);
enqueue(element);
break;
case 2:
element=dequeue();
printf("Deleted element : %d",element);
getch();
break;
case 3:
display();
break;
case 4:
exit(0);
}
}
return 0;
}
void enqueue(int element)
{
if(rear==SIZE-1)
{
printf("Overflow");
getch();
}
else
{
rear=rear+1;
que[rear]=element;
}
}
int dequeue()
{
if(front>rear)
{
printf("underflow");
return 0;
}
front=front+1;
return que[front-1];
}

void display()
{
```

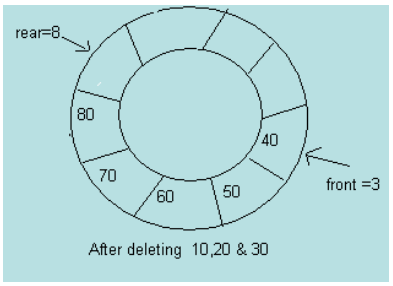
```
int i;
for(i=front;i<=rear;i++)
printf("%d ",que[i]);

getch();
}
```

Circular Queue:

In Circular Queue *front* =0 and *rear* = 0. Insert the element in rear index then increase rear value by 1. To delete the element front value by 1.







Algorithm insert(item){
If (front =0 and rear = size-1) or (rear +1 = front) then
{
Write (“ Queue is full”);
return false;
}
else{
Queue[rear] := item;
rear := (rear +1) mod size;
return true;
}
}

Algorithm remove(){
if (front = rear) then
{
Write(“ Queue is empty”);
}
else{
Item :=queue[front];
front :=front+1;
front =front mod size;

```
return item;
}
}
```

Double Ended Queue

Insertion and deletion can be performed from both sides. To insert an element at rear position increase *rear* value by 1. If *rear=size-1* Queue is full insertion not possible. To insert an element at front decrease *front* value by 1, If *front=0*, insertion not possible.

To delete an element from rear position, decrement the *rear*. To delete an element from front, increase *front, if front > rear* Queue is empty.

Queue implementing using pointer (Linked list)

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

struct qnode{
        int data;
        struct qnode *link;
};
typedef struct qnode node;
node *front, *rear;

void enqueue(int n);
int dequeue();
void display();

int main(){
        int n,ch;
        front=rear=NULL;
        while(1){
                printf("\nMENU\n1.insert\n2.delete\n3.display\n4.exit\n");
                printf("Enter your choice:");
                scanf("%d",&ch);
                switch(ch){
                        case 1:
                                printf("Enter element:");
                                scanf("%d",&n);
                                enqueue(n);
                                break;
                        case 2:
                                printf("Deleted element : %d",dequeue());
                                break;
```

```
                        case 3:
                                display();
                                break;
                        case 4:
                                exit(0);
                }
        }
        return 0;

}

void enqueue(int n){
        node *c;
        c=(node*)malloc(sizeof(node));
        c->data=n;
        c->link=NULL;
        if(rear == NULL){
                front=rear=c;
        }
        else{
                rear->link=c;
                rear=c;
        }
}

int dequeue(){
        node *c;
        int temp;
        if(front==NULL)
                return 0;
        c=front;
        front=front->link;
        temp=c->data;
        free(c);
        return temp;
}

void display(){
        node *c;
        c=front;
        if(c==NULL){
                printf("Queue is empty");
        }
        else{
```

```
                while(c!=NULL){
                        printf("%d  ",c->data);
                        c=c->link;
                }
        }
}
```