KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

DATA STRUCTURES –ITT 201 (S3 IT)

MODULE -2

	Module 2: Linked lists	**10 hrs**
2.1	**Linked lists**: static and dynamic representation, Classification -Singly linked list- Doubly linked list- Circular linked list, array and linked list. **Singly linked list**: Operations on Singly linked list- Traversal-Insertion-deletion, copying -searching - Merging.	2
2.2	**Doubly linked list**: Operations on doubly linked list- Insertion-deletion.	2
2.3	**Circular Linked list** : Operations on circular linked list-Insertion and deletion	2
2.4	**Applications of linked list**: Polynomial representation and manipulation (addition)- Dynamic Memory management.	2
2.5	**Dynamic Memory management**: Fixed sized and variable sized memory allocation and de-allocation. First-fit, best-fit and worst-fit allocation schemes and problems.	2

LINKED LISTS

* A linked list is an ordered collection of finite,homogenous data elements called nodes ,where the linear order is maintained by means of links or pointers.
* Linked lists is called dynamic data structures where amount of memory required can be varied during its use
* In linked lists ,adjacency between the element are maintained by means of links or pointers.
* A link or pointer actually is the address of the subsequent elements.
* A node consists of two fields;

  DATA -- to store the actual information

  LINK – to point to the next node

* linked list can be classified into three major groups

  ->single linked list

  ->double linked list

  ->circular linked list

Advantages

Linked list have many advantages.They are:

- Linked lists are dynamic data structure :- that is ,they can grow or shrink during the execution of a program

- Efficient memory utilization :- here,memory is not preallocated.Memory is allocated whenever it is required and it is deallocated when it is no longer needed.
- Insertion and deletion are easier and efficient :- linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked list.

Disadvantage

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Single linked list

In a single linked list each node contains only one link which points the subsequent node in the list.

In a single linked list one can move from left to right only this is why a single linked list is also alternatively termed as one way list.

Operations on a single linked list

1. *INSERTION – a new node may be inserted*

   ->at the beginning of the linked list

   -> at the end of the linked list

   -> at the specific position of the linked list

2. DELETION – to delete an element

   -> bedinning of the linked list

   -> end of the linked list

   -> at the specific position of the linked list

3. DISPLAY – this operation is used to print each and every node's information.

4. TRAVERSING – it is a process of going through all the nodes of a linked list from one end to the other end .

TO CREATE A NODE:

```
Struct node
{
        int data;
        struct node*Next;
};
```

TO TRAVERSE IN A SINGLY LINKED LIST:

```
{
        struct node*temp;
if(start==NULL)
printf("\nList is empty");
        else{           temp=start;
while(temp!=NULL){
printf("data=%d\n",temp->info);
                        temp=temp->link;
                }
        }
}
```

TO ADD A NODE IN THE BEGINNING:

```
void insertatfront()
{
        int data;       struct node*temp;
temp=malloc(sizeof(struct node));
printf("enter the number to be inserted:");
scanf("%d",&data);          temp->info=data;
temp->link=start;   start=temp;
}
```

TO INSERT A NODE AT THE END:


```
void insertatend()
{
        int data;       struct node*temp,*head;
temp=malloc(sizeof(struct node));
printf("enter the number to be inserted:");
scanf("%d",&data);          temp->link=0;
temp->info=data;   head=start;
        while(head->link!=NULL){
head=head->link;
        }
        head->link=temp;
}
```

TO INSERT A NODE BETWEEN TWO NODES:

```
void insertatposition()
{
        struct node*temp,*newnode;   int
pos,data,i=1;
newnode=malloc(sizeof(struct node));
printf("enter the position and data:");
scanf("%d%d",&pos,&data);      temp=start;
newnode->info=data;      newnode->link=0;
while(i<pos-1){
                temp=temp->link;
                i++;
        }
        newnode->link=temp->link;      temp-
>link=newnode;

}
```

TO DELETE A NODE FROM BEGINNING

```
if(head==NULL) //List is empty
{
printf(Deletion is not possible\n”);
}
```

```
else if(head==tail) //List contains only one node
{
 free(head); head=tail=NULL;
}
```

TO DELETE A NODE AT THE END

```
struct node *p ;
if(head!=NULL){ if(head-
>next==NULL){ head = NULL;
}else{ p=head; while(p-
>next->next!=NULL){ p= p-
>next;
}
p->next =NULL;
}
```


DOUBLE LINKED LIST

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either

forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

Link – Each link of a linked list can store a data called an element.

Next – Each link of a linked list contains a link to the next link called Next.

Prev – Each link of a linked list contains a link to the previous link called Prev.

LinkedList – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation

Doubly Linked List

As per the above illustration, following are the important points to be considered.

Doubly Linked List contains a link element called first and last.

Each link carries a data field(s) and two link fields called next and prev.

Each link is linked with its next link using its next link.

Each link is linked with its previous link using its previous link.

The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

Insertion – Adds an element at the beginning of the list.

Deletion – Deletes an element at the beginning of the list.

Insert Last – Adds an element at the end of the list.

Delete Last – Deletes an element from the end of the list.

Insert After – Adds an element after an item of the list.

Delete – Deletes an element from the list using the key.

Display forward – Displays the complete list in a forward manner.

Display backward – Displays the complete list in a backward manner.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
Void insertFirst(int key, int data) {
 //create a link
 Struct node *link = (struct node*) malloc(sizeof(struct node));
 Link->key = key;
 Link->data = data;
 If(isEmpty()) {
 //make it the last link
 Last = link;
 } else {
 //update first prev link
 Head->prev = link;
 }
 //point it to old first link
 Link->next = head;
 //point first to new first link
 Head = link;
}
```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
```

```
Struct node* deleteFirst() {
 //save reference to first link
 Struct node *tempLink = head;
 //if only one link
 If(head->next == NULL) {
 Last = NULL;
 } else {
 Head->next->prev = NULL;
 }
 Head = head->next;
 //return the deleted link
 Return tempLink;
}
```


Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

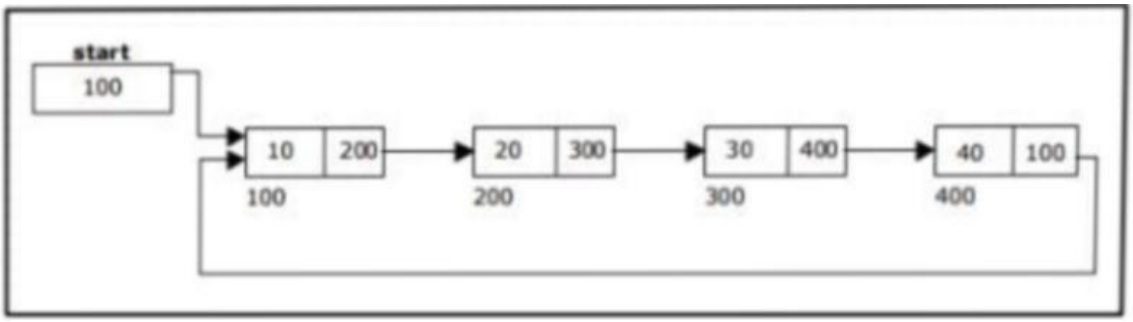
```
//insert link at the last location
Void insertLast(int key, int data) {
 //create a link
 Struct node *link = (struct node*) malloc(sizeof(struct node));
 Link->key = key;
 Link->data = data;
 If(isEmpty()) {
```

```
//make it the last link
Last = link;
} else {
//make link a new last link
Last->next = link;

//mark old last node as prev of new link
Link->prev = last;
}
//point last to new last node
Last = link;
}
```


Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

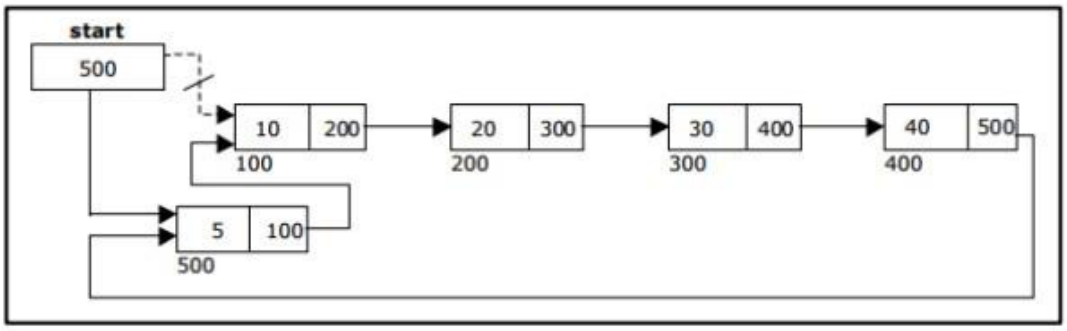
Creating a circular single Linked List with 'n' number of nodes



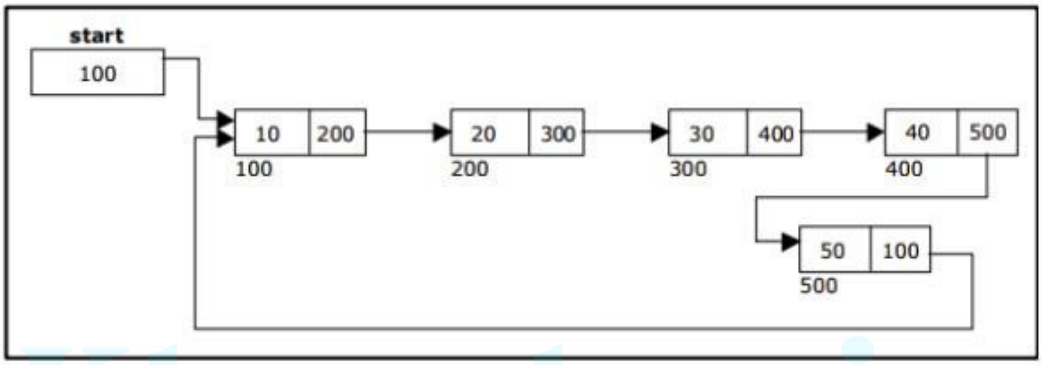
The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing

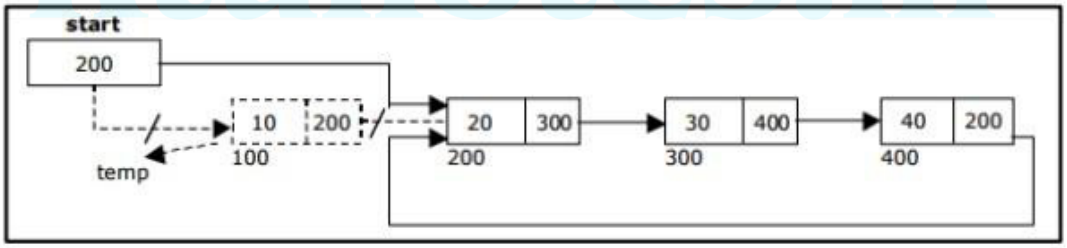
Inserting a node at the beginning:



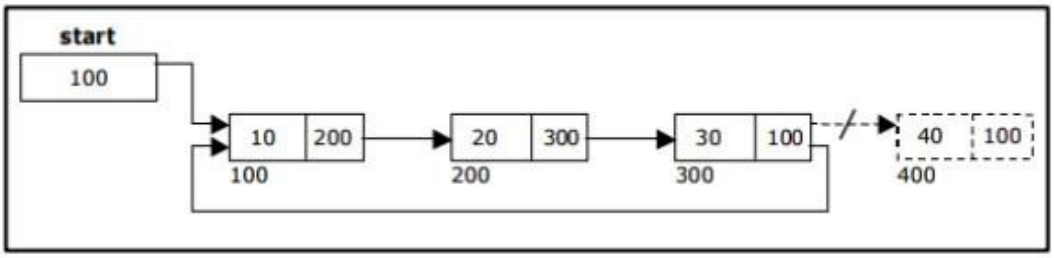
Inserting a node at the end:



Deleting a node at the beginning:



Deleting a node at the end:



Source Code for Circular Single Linked List:

// C code to perform circular linked list operations

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

struct Node {
  int data;
  struct Node* next;
};

struct Node* addToEmpty(struct Node* last, int data) {
  if (last != NULL) return last;

  // allocate memory to the new node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to the new node
  newNode->data = data;

  // assign last to newNode
  last = newNode;

  // create link to iteself
  last->next = last;

  return last;
}

// add node to the front
struct Node* addFront(struct Node* last, int data) {
  // check if the list is empty
  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // add data to the node
  newNode->data = data;

  // store the address of the current first node in the newNode
  newNode->next = last->next;
```

```
  // make newNode as head
  last->next = newNode;

  return last;
}

// add node to the end
struct Node* addEnd(struct Node* last, int data) {
  // check if the node is empty
  if (last == NULL) return addToEmpty(last, data);

  // allocate memory to the new node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // add data to the node
  newNode->data = data;

  // store the address of the head node to next of newNode
  newNode->next = last->next;

  // point the current last node to the newNode
  last->next = newNode;

  // make newNode as the last node
  last = newNode;

  return last;
}

// insert node after a specific node
struct Node* addAfter(struct Node* last, int data, int item) {
  // check if the list is empty
  if (last == NULL) return NULL;

  struct Node *newNode, *p;

  p = last->next;
  do {
  // if the item is found, place newNode after it
  if (p->data == item) {
    // allocate memory to the new node
```

```
    newNode = (struct Node*)malloc(sizeof(struct Node));

    // add data to the node
    newNode->data = data;

    // make the next of the current node as the next of newNode
    newNode->next = p->next;

    // put newNode to the next of p
    p->next = newNode;

    // if p is the last node, make newNode as the last node
    if (p == last) last = newNode;
    return last;
  }

  p = p->next;
  } while (p != last->next);

  printf("\nThe given node is not present in the list");
  return last;
}

// delete a node
void deleteNode(struct Node** last, int key) {
  // if linked list is empty
  if (*last == NULL) return;

  // if the list contains only a single node
  if ((*last)->data == key && (*last)->next == *last) {
  free(*last);
  *last = NULL;
  return;
  }

  struct Node *temp = *last, *d;

  // if last is to be deleted
  if ((*last)->data == key) {
  // find the node before the last node
  while (temp->next != *last) temp = temp->next;
```

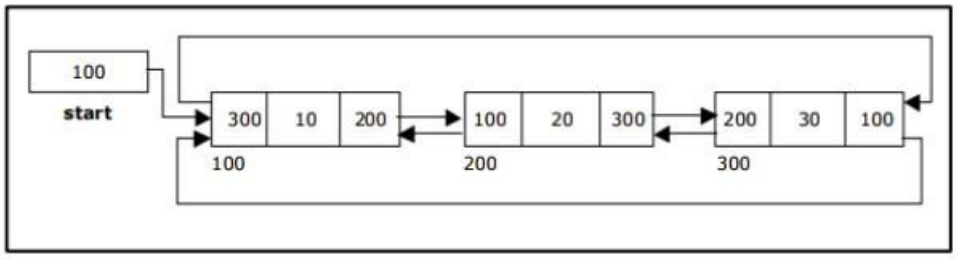
```
  // point temp node to the next of last i.e. first node
  temp->next = (*last)->next;
  free(*last);
  *last = temp->next;
  }

  // travel to the node to be deleted
  while (temp->next != *last && temp->next->data != key) {
  temp = temp->next;
  }

  // if node to be deleted was found
  if (temp->next->data == key) {
  d = temp->next;
  temp->next = d->next;
  free(d);
  }
}

void traverse(struct Node* last) {
  struct Node* p;

  if (last == NULL) {
  printf("The list is empty");
  return;
  }

  p = last->next;

  do {
  printf("%d ", p->data);
  p = p->next;

  } while (p != last->next);
}

int main() {
  struct Node* last = NULL;

  last = addToEmpty(last, 6);
```

```
  last = addEnd(last, 8);
  last = addFront(last, 2);

  last = addAfter(last, 10, 2);

  traverse(last);

  deleteNode(&last, 8);

  printf("\n");

  traverse(last);

  return 0;
}
```

Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the right link of the right most mode points. Hack to the start node and left link of the first node points to the last node.

A circular double linked list is shown in figure

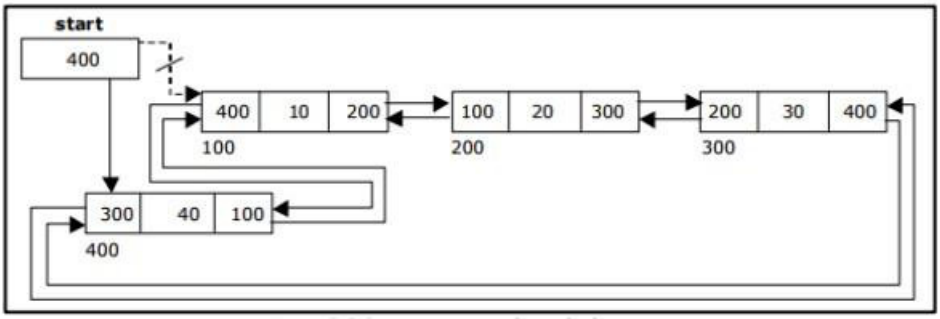


Creating a Circular Double Linked List with 'n' number of nodes The basic operations in a circular double linked list are:

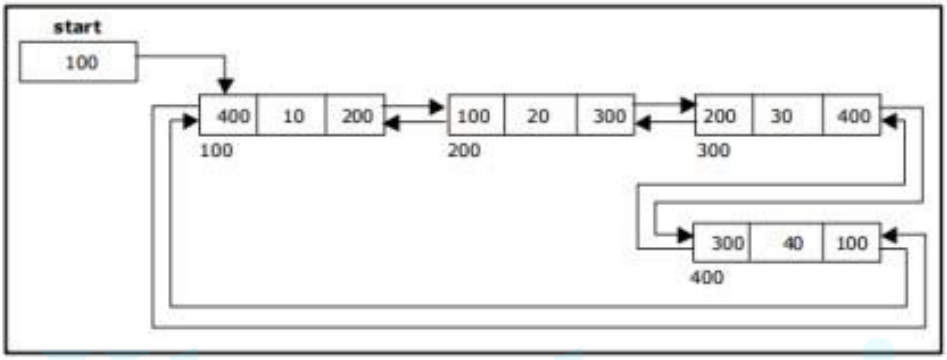
The basic operations in a circular double linked list are:

- Creation.
- Insertion
- Deletion.
- Traversing

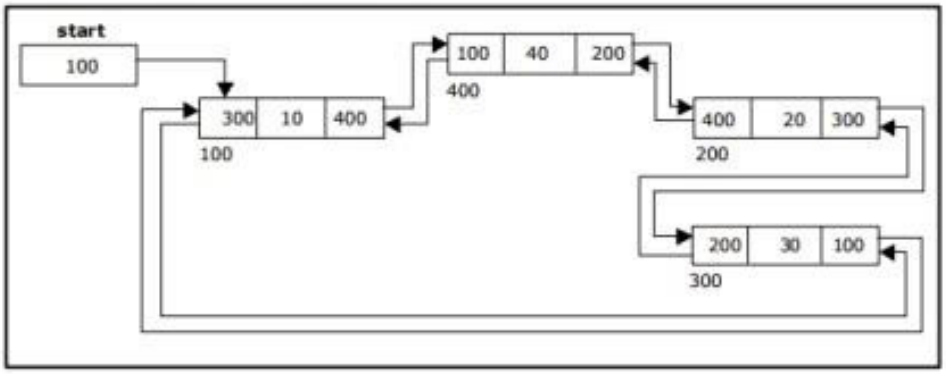
Inserting a node at the beginning:



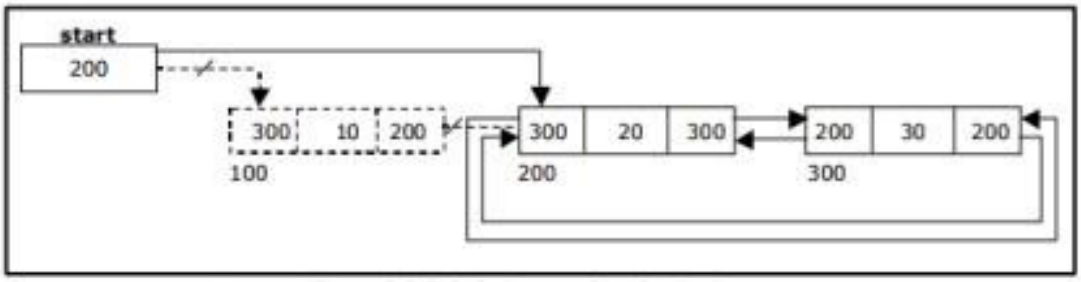
Inserting a node at the end:



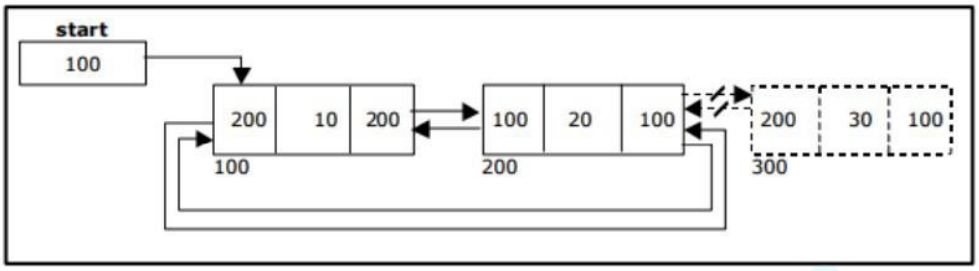
Inserting a node at an intermediate position:



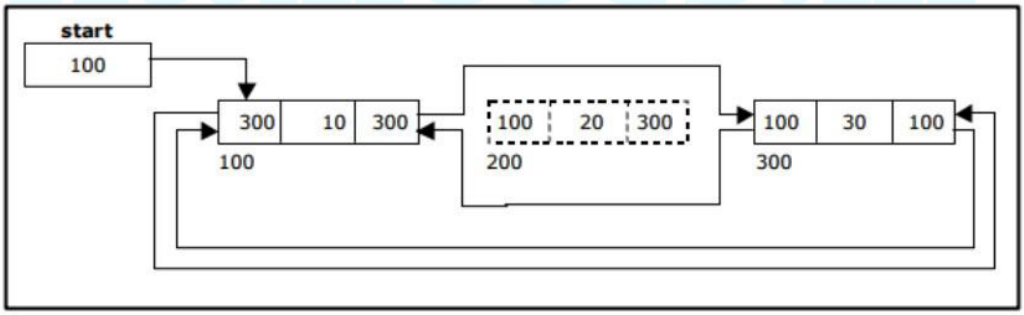
Deleting a node at the beginning:



Deleting a node at the end:



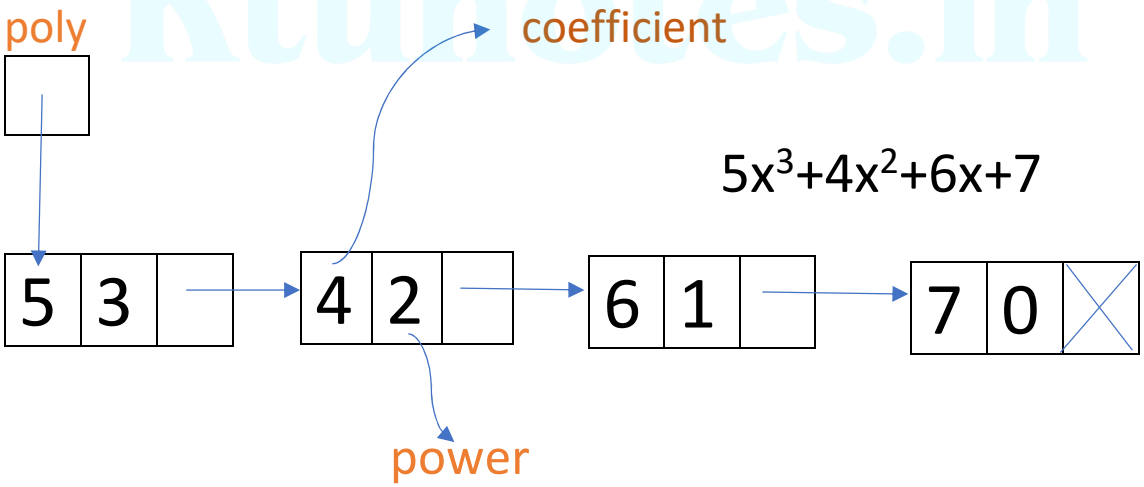
Deleting a node at Intermediate position:



APPLICATIONS OF LINKED LISTS

1. Implementation of stacks and queues
2. Implementation of graphs
3. Dynamic memory allocation
4. Maintaining a directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in node of linked list
7. Representing sparse matrices

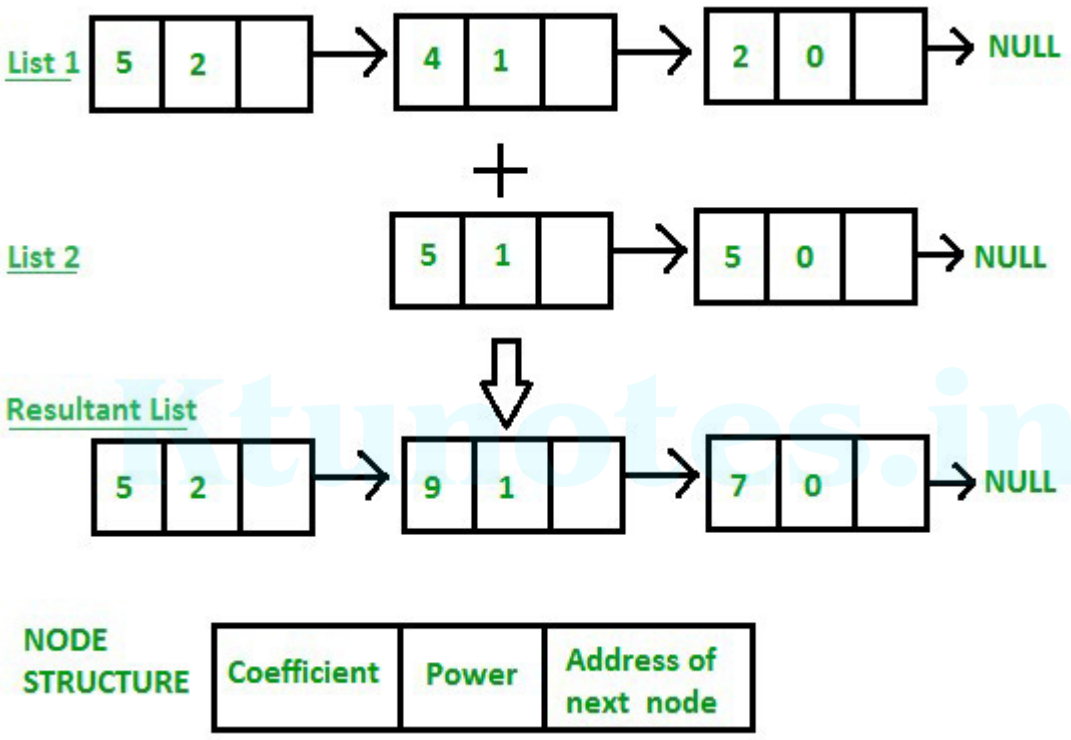
POLYNOMIAL REPRESENTATION USING LINKED LISTS




```
struct pnode{
        int coeff;
        int pow;
        struct node*next;
};
Struct pnode*poly=NULL;
```

POLYNOMIAL ADDITION USING LINKED LISTS

List 1 = $5x^2+4x+2$
List 2 = $5x+5$
Resultant list = $5x^2+9x+7$



Dynamic Memory Management in C

When a C program is compiled, the compiler allocates memory to store different data elements such as constants, variables (including pointer variables), arrays and structures. This is referred to as compile-time or static memory allocation. There are several limitations in such static memory allocation:

1. This allocation is done in memory exclusively allocated to a program, which is often limited in size.

2. A static array has fixed size. We cannot increase its size to handle situations requiring more elements. As a result, we will tend to declare larger arrays than required, leading to wastage of memory. Also, when fewer array elements are required, we cannot reduce array size to save memory.

3. It is not possible (or efficient) to create advanced data structures such as linked lists, trees and graphs, which are essential in most real-life programming situations.

The C language provides a very simple solution to overcome these limitations: dynamic memory allocation in which the memory is allocated at run-time, i. e., during the execution of a program. Dynamic memory management involves the use of pointers and four standard library functions, namely, malloc, calloc, realloc and free. The first three functions are used to allocate memory, whereas the last function is used to return memory to the system (also called freeing/deallocating memory). The pointers are used to point to the blocks of memory allocated dynamically.

When called, a memory allocation function allocates a contiguous block of memory of specified size from the heap (the main memory of a computer) and returns its address. This address is stored in a pointer variable so as to access that memory block.

The heap is managed by the operating system and is allocated (on demand) to running programs. The heap is much larger than the program's local memory used to hold program data. Thus, we can create much larger arrays and other data structures in the heap. Note that the allocation on the heap is not in a sequence, i. e., the memory blocks can be allocated anywhere. As a result, a heap memory is usually fragmented. The memory manager in the operating system decides the optimal location for allocation of a particular memory block.

Standard library functions for dynamic memory management

The C language provides four functions for dynamic memory management, namely, malloc, calloc, realloc and free. These functions are declared in stdli.b .h header file. The malloc, calloc and realloc functions allocate a contiguous block of memory from heap. If memory allocation is successful, they return a pointer to allocated block (i. e., starting address of the block) as a void (i. e., a typeless) pointer; otherwise, they return a NULL pointer.

The malloc function

The malloc (memory allocate) function is used to allocate a contiguous block of memory from heap. If the memory allocation is successful, it returns a pointer to the allocated block as a void pointer; otherwise, it returns a NULL pointer. The prototype of this function is given below.

// void *malloc(size_t size);

The malloc function has only one argument, the size (in bytes) of the memory block to be allocated. Note that size_t is a type, also defined in stdlib.h, which is used to declare the sizes of memory

objects and repeat counts. We should be careful while using the malloc function as the allocated memory block is uninitialized, i. e., it contains garbage values.

As different C implementations may have differences in data type sizes, it is a good idea to use the sizeof operator to determine the size of the desired data type and hence the size of the memory block to be allocated. Further, we should typecast the void pointer returned by these functions to a pointer of appropriate type. Thus, we can dynamically allocate a memory block to store 50 integers as shown below.

```
// int *pa; /* pointer to the memory block to be allocated */

   pa= (int*) malloc(50 * sizeof(int)); /* alloc memory*/
```

This is often sufficient to allocate a memory block, particularly in small toy-like programs. However, in coding applications, it is a good idea to ensure, before continuing with program execution, that the memory allocation was successful, i. e., the pointer value returned is not NULL as shown below.

```
// pa= (int*) malloc(50 * sizeof(int)); /* alloc memory*/

if (pa == NULL) {

 printf("Error: Out of memory ...\n'');

 exit(l); /*Error code 1 is used here to indicate

 out of memory situation */

}

/* continue to use pa as an array here onwards */
```


Note that once memory is allocated, this dynamically allocated array can be used as a regular array. Thus, the I th element of array pa in the above example can be accessed as pa [i].

Note that we can combine the memory allocation statement with the NULL test to make the code concise (sacrificing readability) as shown below.

```
if ((pa= (int*) malloc(50 * sizeof(int))) == Null) {

 printf("Error: Out of memory ...\n");

 exit(1);

}
```

If we use this approach to create concise programs, we are likely to make mistakes at least in the beginning. Hence, observe carefully the use of parentheses in the if statement. You will soon realize that it is not as difficult as it appears to be. Since the assignment statement has lower precedence than the equality operator, the memory allocation statement is first included in a pair of parentheses and then compared with the NULL value within the parentheses of if statement as illustrated below.

The calloc function

The calloc function is similar to malloc. However, it has two parameters that specify the number of items to be allocated and the size of each item as shown below.

//void *callee (size_t n_items, size_t size) ;

Another difference is that the calloc initializes the allocated memory block to zero. This is useful in several situations where we require arrays initialized to zero. It is also useful while allocating a pointer array to ensure that the pointers will not have garbage values.

The realloc function

The realloc (reallocate) function is used to adjust the size of a dynamically allocated memory block. If required, the block is reallocated to a new location and the existing contents are copied to it. Its prototype is given below.

// void *realloc(void *block; size_t size);

Here, block points to a memory block already allocated using one of the memory allocation functions (malloc, calloc or realloc). If successful, this function returns the address of the reallocated block; or NULL otherwise.

The free function

The free function is used to deallocate a memory block allocated using one of the memory allocation functions (malloc, calloc or realloc). The deallocation actually returns that memory block to the system heap. The prototype of free function is given below.

// void free (void *block);

when a dynamically allocated block is no longer required in the program, we must return its memory to the system.

Module - II

Arrays & Searching

Polynomial Representation using Arrays :-

Arrays

→ An array is a consecutive set of memory locations.
→ An array is a set of pairs ie index & values
→ For each index which is defined, there is a value associated with the index.
→ In mathematical term we call this a correspondance or a mapping.
→ Array can be defined as

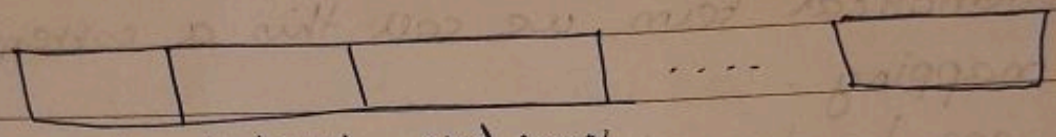
Structure - ARRAY (value, index)
Declare - CREATE() → array
RETRIEVE (array, index) → value
STORE (array, index, value) → array.

Representation of array.

→ one dimensional array
1D array can be represented as
A[lowerbound : Upperbound]
Eg A[3:4]
→ The address of A[i] can be calculated by
Base address + (i - lowerbound) = $\alpha + (i - L)$
Here Base address is the starting address
→ Total number of elements can be calculated by
UB - LB + 1
→ Two dimensional array
It can be represented as
$A[L_1 \ldots U_1, L_2 \ldots U_2]$
Row - $U_1 - L_1 + 1$
Column - $U_2 - L_2 + 1$ Total no: of element = Row * Column

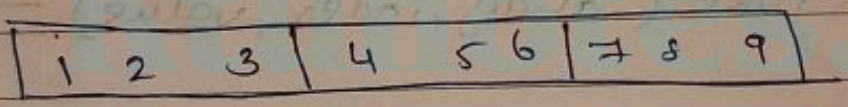
→ Two common ways to represent multidimensional arrays
Row major order
Column major order

→ Row major order
It stores multidimensional arrays by rows
$A[L_1 \ldots U_1, L_2 \ldots U_2]$
where U_1 - row representation
U_2 - column representation



Eg: 1 2 3
4 5 6
7 8 9

A[3,3] in C A[3][3]

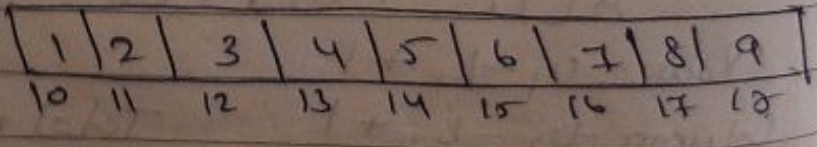


Address calculation of A[i][j]
Base address + $(i - L_1) U_2 + (j - L_2)$
where U_2 - column

Eg: A[5:7, 2:4] find the address of A[6,3] BA = 10
m (row) = 7 - 5 + 1 = 3
n (column) = 4 - 2 + 1 = 3

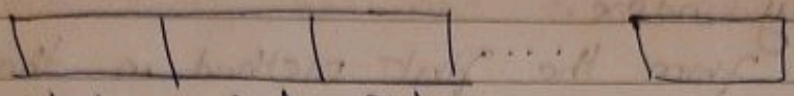
(5,2)	(5,3)	(5,4)
1	2	3
(6,2)	(6,3)	(6,4)
4	5	6
(7,2)	(7,3)	(7,4)
7	8	9

A [6,3]
BA + $(i - L_1) U_2 + (j - L_2)$
10 + (6 - 5) 3 + (3 - 2)
10 + 3 + 1
14



→ Column Major Order

It stores multidimensional arrays by columns



1st column 2nd column 3rd column

1	4	7	2	5	8	3	6	9

Address calculation

$$BA + (i - L_1) + (j - L_2)\,m$$

$m \rightarrow$ row

Eg: A[6:9, 3:6] find the address of A[8,5]

BA = 10

$m = 9 - 6 + 1 = 4$

$n = 6 - 3 + 1 = 4$

A[8,5]

$10 + (8 - 6) + (5 - 3)\,4$

$10 + 2 + 8$

$= 20$

Representation of Polynomial using array.

(1)

→ 1D array is defined and coefficient is added to the array and exponent is indicated by the index value

Eg: $2x^2 + 3x + 1$

$2x^2$	$3x^1$	$1x^0$
2	3	1
a[2]	a[1]	a[0]

Here array size is fixed. Usually it will be larger than the degree of polynomial.

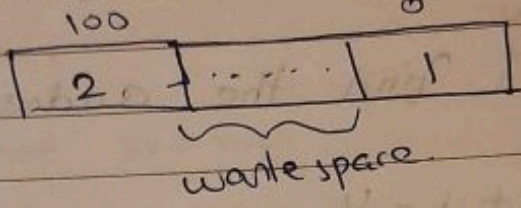
(2) Here a 1D array is defined and the coefficient is added to the array and the exponent is represented by the array index.

⇒ It differs from the first method in the "size of array".

⇒ Here size of the array is defined as the largest degree of the polynomial

⇒ Disadvantage - waste of space

Eg: $2x^{100} + 1$



(3) ⇒ Here coefficient and exponent is stored in the array

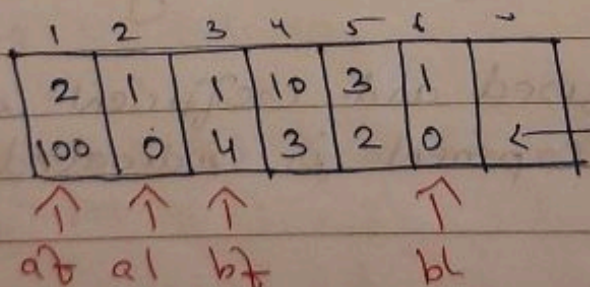
⇒ Two pointers are used to indicate the beginning and end of the polynomial.

⇒ Zero coefficient term is not used

⇒ There is no fixed size allocation is needed

Eg: $2x^{100} + 1 = A(x)$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$



⇒ To find the last term of the polynomial we use

$$el = ef + (n-1)$$

el = last term of the polynomial

n = length " " "

ef = first " " " "

Eg: B(x) $el = 3 + (4-1) = 6$

⇒ A polynomial is a sum of terms where each term has the form ax^e where x is the variable, a is the coefficient and e is the exponent.

Eg: $10x^2 + 3x - 10$

⇒ If it is a n^{th} degree polynomial, it will contain maximum of $(n+1)$ terms.

```
struct poly
  {
    int coeff;
    int exp;
  }
  p1[100];
```

⇒ A polynomial of a single variable $A(x)$ can be written as $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ where $a_n \neq 0$ and degree of $A(x) = n$

	0	1	2		n-1	n
Poly	a_0	a_1	a_2	...	a_{n-1}	a_n

$B(x) = 2x^4 + 3x^3 + 5x^2 + 8x + 15$

Coeff	2	3	5	8	15
Exp	4	3	2	1	0

where the structures are (2,4) (3,3), (5,2) (8,1) (15,0)

Polynomial Addition

⇒ The algorithm works by comparing terms from the two polynomials until one or both of the polynomials become empty.

⇒ The comparison is performed and the proper term is added to the resulting sum polynomial

→ If one of the polynomial becomes empty, we copy the remaining terms from the non-empty polynomial into the resulting polynomial

Algorithm PolyAdd(start A, Finish A, start B, Finish B, * start D, * Finish D)

```
1  start
2  * StartD = avail
3  while( start A <= FinishA AND start B <= Finish B)
4;  if (terms[start A]. expon < terms[start B]. expon)
5       attach( terms[start B]. coef, terms[start B. expon)
6       start B++
7  elseif (term [Start A]. expon > term [start B]. expon)
8       attach (terms[start A]. coef, terms[start A. expon)
9       start A++
10 else coefficient = term[startA]. coef + terms[start B]. coef
11 if (coefficient) attach (coefficient, terms[start A]. expon
                                                    end if
12  start A++, start B++
13 end if
14 end while
15 while (start A <= finish A)
16 attach (terms[start A]. coef, terms[start A] expon)
17   start A++
18 end while
19 while (start B <= Finish B)
20 attach (term[start B]. coef, terms[start B] . expon)
21   start B++
22 end while
23 * Finish D = avail - 1
24 stop
```

Algorithm Attach (coefficient, exponent)

1. Start
2. if (avail >= MAX_TERMS)
3. print " Too many terms in the polynomial"
4. exit
5. end if
6. term[avail].coef = coefficient
7. term[avail++].expon = exponent
8. stop

Analysis of algorithm

O(m+n) where m and n is the degree of the polynomial.

Ex. $A_1 = 3x^1 + 5x^2 + 7x^3$

$B_2 = 10x^0 + 3x^1 + 5x^2$

$A_1 = \underline{0x^0} + 3x^1 + 5x^2 + 7x^3$

$B_2 = 10x^0 + 3x^1 + 5x^2 + \underline{0x^3}$

i=0 | 0 | 3 | 5 | 7 | j=0 | 10 | 3 | 5 | 0 |

$a_0 = 0 + 10$ $a_1 = 3 + 3$ $a_2 = 5 + 5$

$C3 = 10x^0 + \cdots$ $C3 = 10x^0 + 6x^1 + \cdots$ $C_3 = 10x^0 + 6x^1 + 10x^2 + \cdots$

$a_3 = 7 + 0$

$C_5 = 10x^0 + 6x^1 + 10x^2 + 7x^3$

| 10 | 6 | 10 | 7 |

Ex: $x1 = 7x^4 + 5x^2 + 3x^1$ $\quad x2 = 5x^3 + 3x^1 - 8x^0$

i=0 $\qquad$ j=0

coeff	7	5	3	coeff	5	3	-8
expon	4	2	1	expon	3	1	0

x3 Coeff
Expon

Case I if i > j

```
if (x1[i].expo > x2[j].expo)
{
x3[k].coeff = x1[i].coeff;
x3[k].expo = x1[i].expo;
i = i+1;
k = k+1;
}
```

x3 Coeff 7
expon 4

Case II if j > i

```
if (x1[i].expo < x2[j].expo)
{
x3[k].coeff = x2[j].coeff;
x3[k].expo = x2[j].expo;
j = j+1;
k = k+1;
}
```

x3 coeff 7 5
expo 4 3

x3 coeff 7 5 5 6
expo 4 3 2 0

Case II if i = j

```
if (x1[i].expo == x2[j].expo)
{
x3[k].coeff = x1[i].coeff + x2[j].coeff
```

x3 coeff 7 5 5 6
expo 4 3 2 1

```
x3[k].expo = x1[i].expo;
i=i+1;
j=j+1;
k=k+1;
}
```

x3 coeff 7 5 5 6 -8
expo 4 3 2 1 0

```
while (j<n) do
{
x3[k].coeff = x2[j].coeff;
x3[k].expo = x2[j].expo;
j=j+1;
k=k+1;
}
```


Sparse Matrix

→ It is a matrix in which most of the elements are zeros

→ Using a 2D array to store a sparse matrix result in wastage of memory and hence an efficient representation is required.

→ Ex: Consides a matrix of size 100 x 100 containing only 10 non zero elements

→ Any element within a matrix can be uniquely represented by the triple <row, column, value>

→ Hence we can use an array of triples to represent a sparse matrix

Triplet Representation (Array representation) of Sparse Matrix

→ In this representation we consides only non zero values along with their row, column and index values.

→ The triples should be organised such that the row indices are in ascending order.

→ And all the triples for any row should be stored such that the column indices are also in ascending order

→ The number of rows, columns and number of non zero elements are to be stored.

→ The 0th row stores number of rows, columns and non-zero elements in the sparse matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

$$\begin{bmatrix} 0 & 0 & 9 & 0 \\ 0 & 1 & 0 & 0 \\ 5 & 0 & 9 & 0 \\ 8 & 0 & 0 & 6 \end{bmatrix}$$

4	4	6
0	2	9
1	1	1
2	0	5
2	2	9
3	0	8
3	3	6

Advantages of Sparse Matrix

→ Storage - There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements

→ Computing time - It can saved by logically designing a data structure traversing only the non-zero elements.

Transposing a matrix

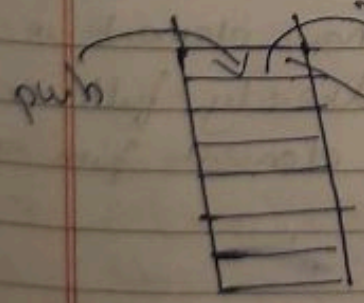
	row	column	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

transpose →

	row	colum	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Stack

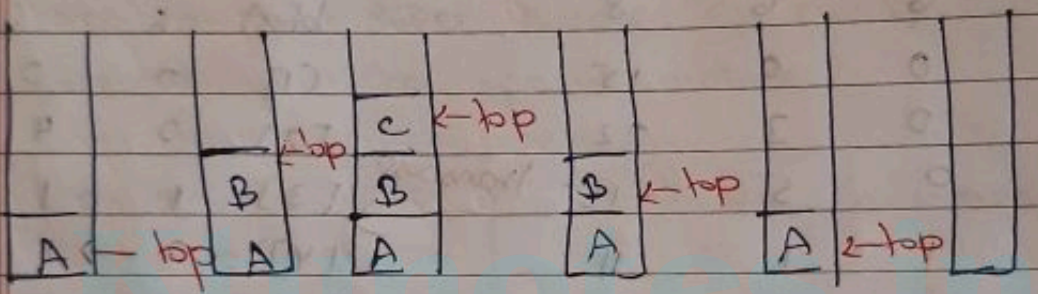
→ An ordered list in which insertions (also called push) and deletions (also called pop) are made at one end called the top



Eg: Stack of Books
Stack of Plates

→ Two basic operations of stack
→ Push Insert an element at the top of the stack
→ Pop Delet an element from the top of the stack

→ LIFO
In stack elements are arranged in Last In First Out manner So it is called LIFO lists
→ An element in the stack is termed as ITEM
→ The maximum no: of elements that a stack can accomodate is called its SIZE



Array Representation of Stack.

→ Stack can be represented as a linear array
→ TOP – It is a variable to store address of the topmost element of the stack
→ MAX – It is the maximum number of elements that the stack can hold
→ If TOP = -1 ⇒ stack is empty
If TOP = MAX-1 ⇒ Stack is full
→ Overflow – If we try to insert a new element in the stack top which is already full
Underflow: If we try to delet an element from an empty stack.

Basic Operations
Push() –
pop() –

peek() get the top data element of the stack, without removing it

isFull()

isEmpty()

```
* int peek()
  {
   return stack[top];
  }
```

```
* bool isfull()
  {
   if(top==MAX-1)
     return TRUE;
     else
     return FALSE;
  }
```

```
* bool isempty
  {
    if (top==-1)
      return TRUE;
      else
      return FALSE;
  }
```

Algorithm Push

Algorithm Push(A, item)

1. Start
2. If top == MAX-1 then
3. print "Stack is full"
4. exit
5. End if
6. top++
7. A[top] = item
8. Stop

Algorithm Pop

Algorithm Pop(A)

1. Start
2. If top < 0 then
3. print "Stack Underflow"
4. exit
5. End if
6. item = A[top]
7. top--
8. return top;
9. Stop

Complexity of Stack operations

Push	O(1)
Pop	O(1)
Traversal	O(n)
IsEmpty	O(1)
Top()	O(1)

Applications of stack

→ When a function call occurs, the return address is stored in stack
→ Number s/m conversion
→ Sorting
→ Expression evaluation
→ Expression conversion
→ String reversal
→ Used for implementing subroutines in general programming language.

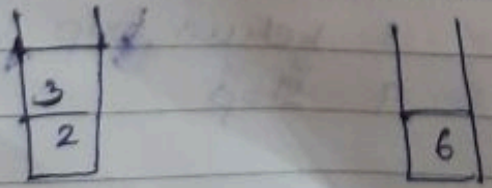
Expression Notation

* Infix Notation → A+B
* Prefix Notation → +AB Polish Expression
* Postfix Notation → AB+ Reverse Polish Expression

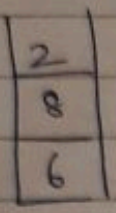
Evaluation of Postfix Expression

Eg: 23 * 82 / 2 * + 6 2 * - 4 + # (# — end of expression)

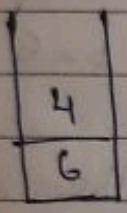
→ Push 2 and 3 into the stack
→ Pop 3 and 2 and perform the operation 3*2 = 6
Push 6 into the stack.



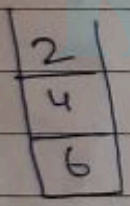
Now ; push 8 and 2 into the stack



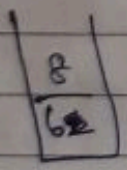
Pop 2 and 8 and perform division operation
ie $8/2 = 4$



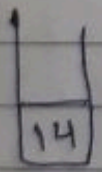
Push 2 into the stack.



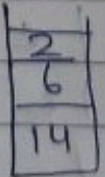
Pop 2 and 4 $\Rightarrow 2 \times 4 = 8$



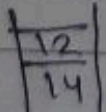
Pop 8 and 6 perform addition and push result into the stack



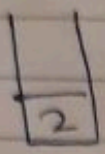
Push 6 and 2



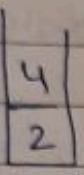
Pop 6 and 2 perform multiplication $6 \times 2 = 12$



Pop 12 and 14 perform subtraction and push the result into stack.



Push 4



Pop 2 and 4 and perform addition 4+2=6

Time Complexity = O(n) where n is the number of tokens in the expressions

Infix to Postfix Conversion.

1) 2+3*4 → (2+(3*4))
2+(34*)
234*+

2) a*b+5 → (a*b)+5
(ab*)+5
ab*5+

3) (1+2)*7 → ((1+2)*7)
(12+)*7
12+7*

4) a*b/c → (a*b)/c
(ab*)/c
ab*c/

Rules

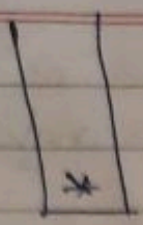
Scan the infix expression from left to right. For each symbol

* If the symbol is an operand, output it immediately.
* If it is a left paranthesis, push it on the stack
* If it is a right paranthesis, continually pop the stack and output the operator until the corresponding left paranthesis is poped.
* otherwise pop and output operators from the stack as long as they have a priority higher than or equal to the current operator.
  But never pop a left paranthesis.
* At the end of the expression, pop and output operators until the stack is empty

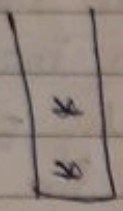
  \+ and - have low priority
  * and / have high priority

Ex: A * (B+C) *D

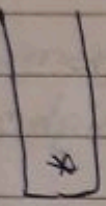
Stack	output
	A
C *	AB
*	A
+ C *	AB
C *	A
+ C *	ABC



ABC+



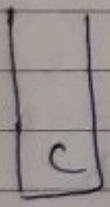
ABC+*

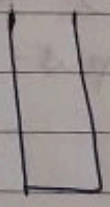


ABC+*D

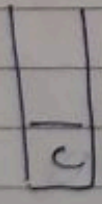
ABC+*D*

(2) Infix -> (5+3) * (8-2)

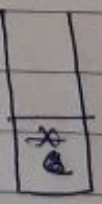




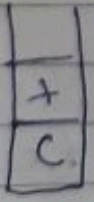
53+



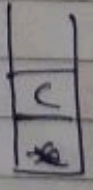
5



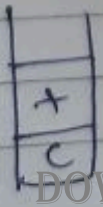
53+



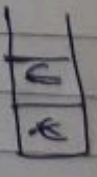
5



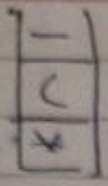
53+8



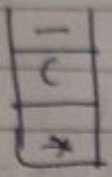
53



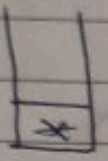
53+8



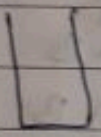
53+8



53+82



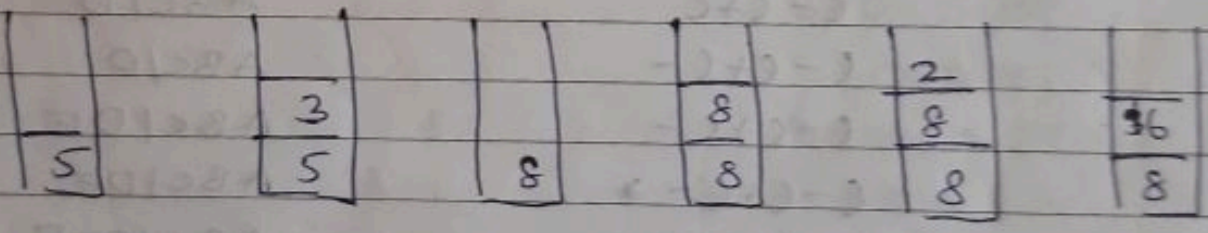
53+82-

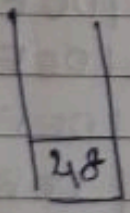


53+82-*

Evaluate the infix expression

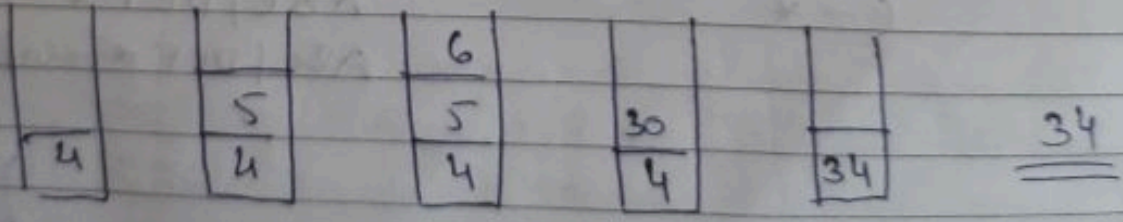
1) (5+3)*(8-2) → 53+82-*





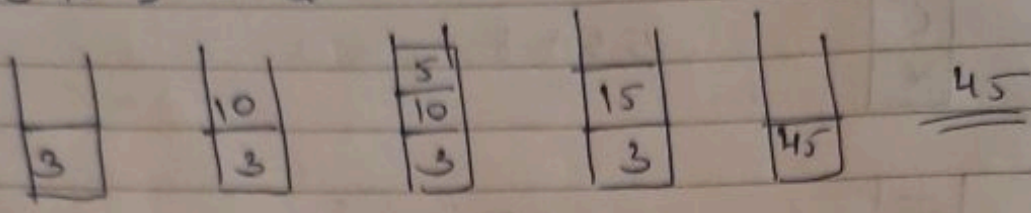
48

2) 456*+



34

3) 3 10 5 + *



45

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

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
/	(+ (/	A B
C	(+ (/	A B C
+	(+ (+	A B C /
(	(+ (+ (	A B C /
D	(+ (+ (	A B C / D
-	(+ (+ (-	A B C / D
E	(+ (+ (-	A B C / D E
*	(+ (+ (- *	A B C / D E
F	(+ (+ (- *	A B C / D E F
)	(+ (+	A B C / D E F * -
/	(+ (+ /	A B C / D E F * -
G	(+ (+ /	A B C / D E F * - G
)	(+	A B C / D E F * - G / +
*	(+ *	A B C / D E F * - G / +
H	(+ *	A B C / D E F * - G / + H
		A B C / D E F * - G / + H * +

5) (A + B) * C / D + E ^ F / G

Scanned symbol	stack	Postfix Expression
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
/	(*/	AB+C*
D	(/	AB+C*D
+	(+	AB+C*D/
E	(+	AB+C*D/E
^	(+^	AB+C*D/E
F	(+^	AB+C*D/EF
/	(+/	AB+C*D/EF^
G	(+/	AB+C*D/EF^G
		AB+C*D/EF^G/+

* Evaluate the postfix notation
P: 5, 6, 2, +, *, 12, 4, /, -

Symbol Scanned	stack
5	5
5, 6	5, 6
2	5 6 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
	37

* 2 3 1 x + 9 −

Scanned Symbol	Stack
2	2
3	2 3
1	2 3 1
x	2 3
+	5
9	5, 9
−	−4

* 5 3 + 6 2 / * 3 5 * +

Scanned Symbol	Stack
5	5
3	5 3
+	8
6	8 6
2	8 6 2
/	8 3
*	24
3	24 3
5	24 3 5
*	24 15
+	39

Note: Priorities of operators ^

* /

\+ −

No two operators of same priority can stay together in the stack column.

Lowest priority cannot be placed before higher-priority.

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

Symbol	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
-	(-	ABC/DE+*+
F	(-	ABC/DE+*+F
)		ABC/DE+*+F-

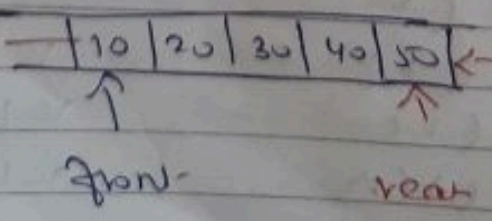
K+L-M*N+(O^P)*W/U/V*T+Q

Symbol	Stack	Postfix
K	(	K
+	(+	K
L	(+	KL
-	(-	KL+
M	(-	KL+M
*	(-*	KL+M
N	(-*	KL+MN
+	(+	KL+MN*-
(	(+(	KL+MN*-
O	(+(	KL+MN*-O
^	(+(^	KL+MN*-O
P	(+(^	KL+MN*-OP
)	(+	KL+MN*-OP^

*	(+*	KL+MN*-OP^
W	(+*	KL+MN*-OP^W
/	(+/	KL+MN*-OP^W*
U	(+/	KL+MN*-OP^W*U
/	(+/	KL+MN*-OP^W*U/
V	(+/	KL+MN*-OP^W*U/V
*	(+*	KL+MN*-OP^W*U/V/
T	(+*	KL+MN*-OP^W*U/V/T
+	(+	KL+MN*-OP^W*U/V/T*+
Q	(+	KL+MN*-OP^W*U/V/T*+Q
		KL+MN*-OP^W*U/V/T*+Q+

Queue

- Queue is an ordered list.
- The end at which new elements are added is called the rear
- From which the elements are deleted is called front
- It is also called FIFO list (First In First Out)
- It can be implemented using either an array or a linked list
- Insertion is called enqueue
- Deletion is called dequeue
- Length of the queue = rear - front
- Condition for empty queue = front == rear = -1
- If rear == n-1 then queue is full

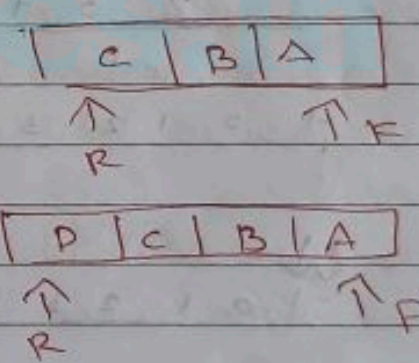


A	AB	ABC	ABCD	ABCDE	BCDE
f,r	f r	f r	f r	f r	f r
add	add	add	add	delete	

Array Implementation

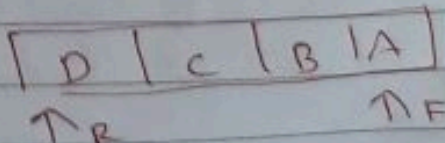
Algorithm insert(x)

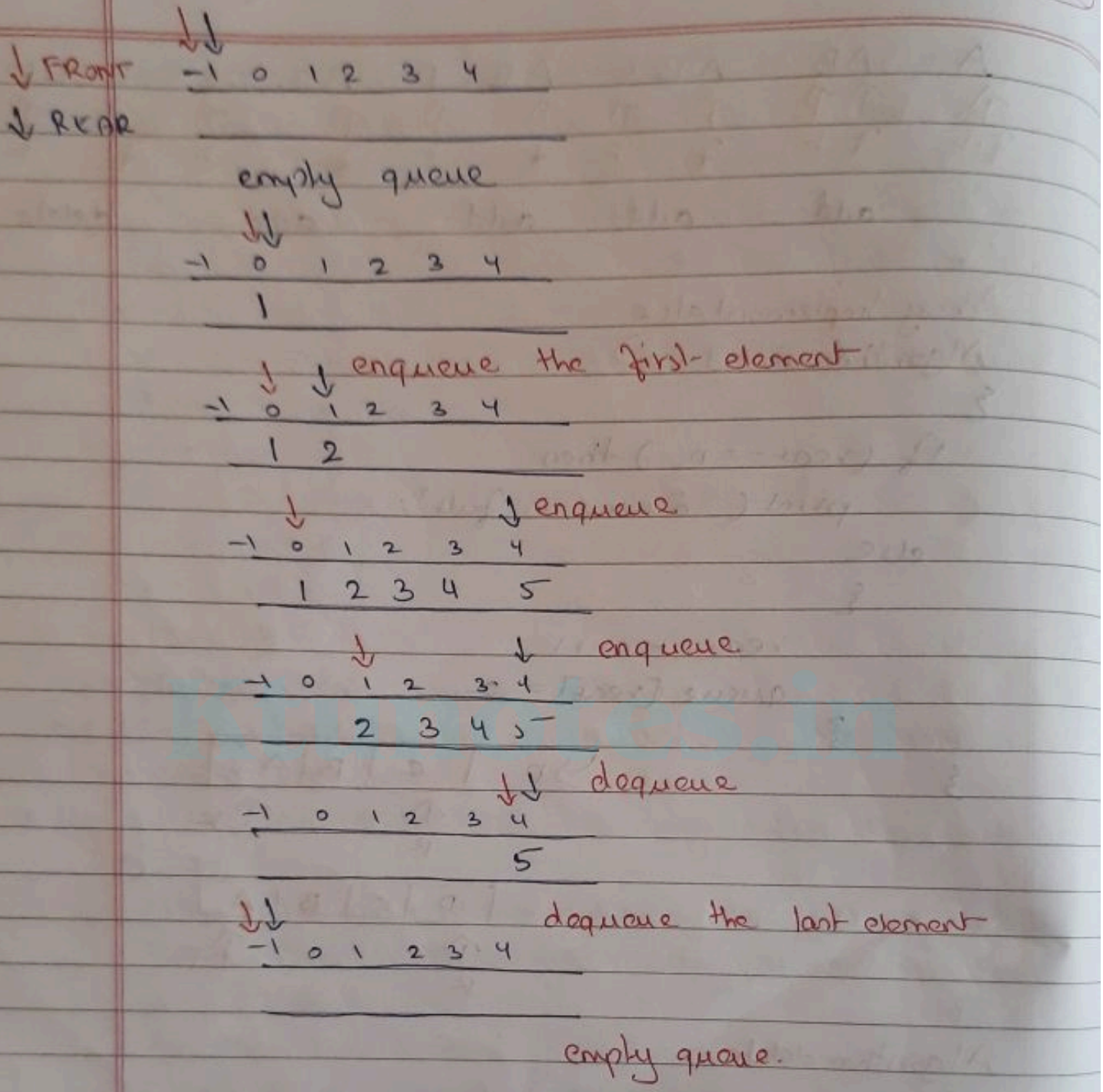
```
{
  if (rear==n ) then
     print("Queue is full";
  else
     {
        rear = rear+1
        queue[rear] = x;
     }
}
```



Algorithm delete

```
{
  if (front==rear) then
     print("Queue is empty");
  else
     { front = front+1;
       item = queue[front];
     }
}
```





Queue is empty front = rear = -1
Queue is full front = 0 rear = n-1
Queue contains elements ≥1 front <= rear
No. of element - Rear - Front + 1

Algorithm Enqueue

Algorithm Enqueue(Q, item)

1. Start
2. If rear = size - 1
3. Print "Queue is full"
4. exit
5. end if
6. if front == -1 and rear == -1 then
7. front = 0
8. end if
9. rear = rear + 1
10. Q[rear] = item
11. Stop

Algorithm Dequeue

Algorithm Dequeue (Q)

1. Start
2. If front == -1 then
3. Print "Empty queue"
4. exit
5. end if
6. item = Q[front]
7. if (front == rear)
8. rear = -1, front = -1
9. else
10. front = front + 1
11. end if
12. stop

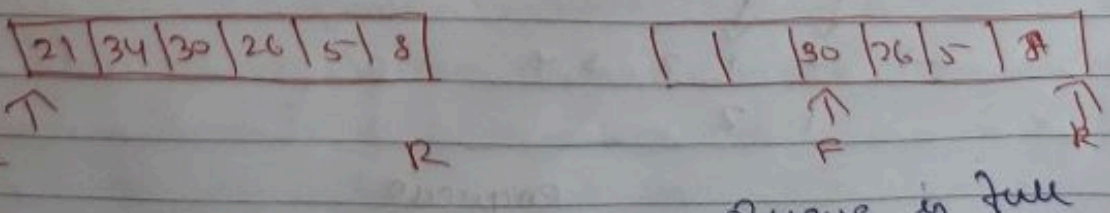
Complexity of Queue Operations

Enqueue	O(1)
Dequeue	O(1)
Traversal	O(n)

Disadvantage of Linear queue

⇒ In a linear queue, once the queue is completely full, it's not possible to insert more elements.

⇒ Even when we dequeue some elements from the queue, we cannot insert new elements because the rear pointer is still at the end of the queue

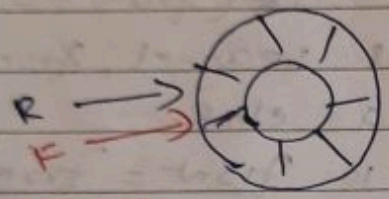


Circular Queue

⇒ It is also a linear data structure, which follows the principle of FIFO.

→ But instead of ending the queue at the last position, it again starts from the first position after the last.

→ Hence making the queue behave like a circular data structure.



→ Once the data is added, rear pointer is incremented to the point to the next available location

→ When dequeue is executed, only the front pointer is incremented.

→ The queue data is the only data blw front and rear.

→ The front and rear pointers will get reinitialized to every time when they reach the end of the queue

↓FRONT
↓REAR

-1	0	1	2	3	4

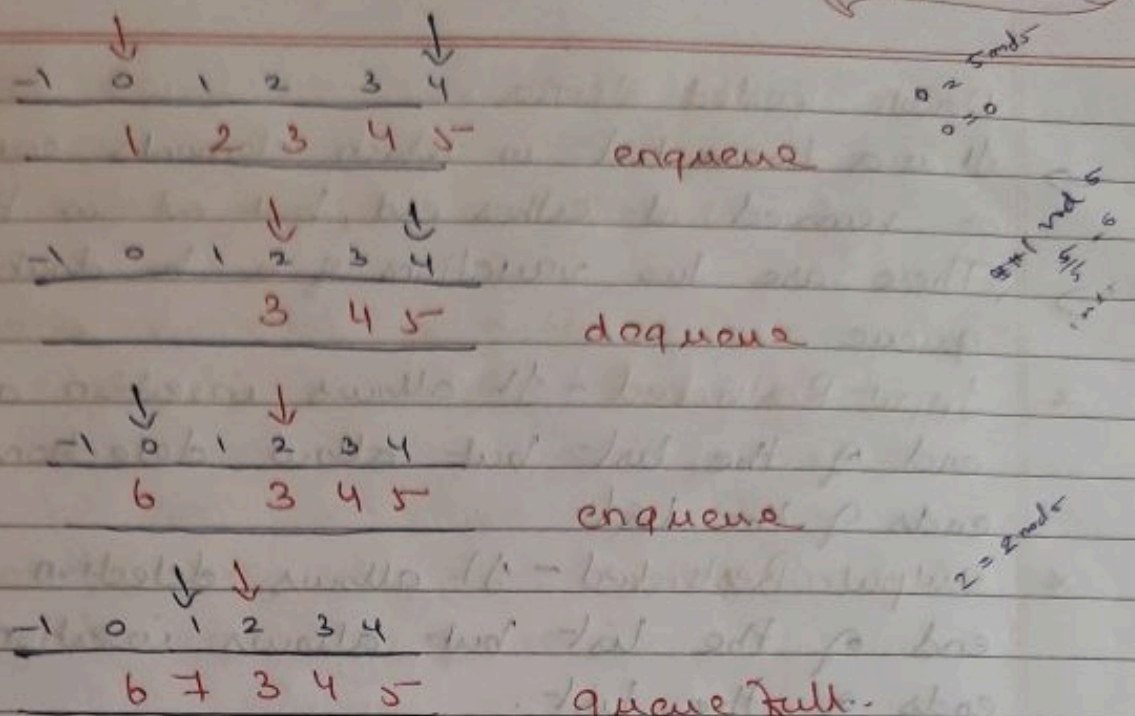
empty queue

-1	0	1	2	3	4
	1				

enqueue first element

-1	0	1	2	3	4
	1	2			

enqueue



Algorithm to insert an element into circular queue

```
{
    rear = (rear+1) mod n;
    if (front == rear) then
        print ("Queue is full");
    else
        queue[rear] = x;
}
```

Algorithm to delete an element from circular queue

```
{
    if (front == rear) then
        print ("Queue is empty");
    else
    {
        front = (front + 1) mod n;
        item = queue[front];
    }
}
```

Double Ended Queue

→ It is a linear list in which elements can be added or removed at either end, but not in the middle

→ There are two variations of a double ended queue.

* Input Restricted - It allows insertion at only one end of the list but allow deletions at both ends of the list
* Output Restricted - It allows deletion at only one end of the list but allows insertions at both ends of the list.

Priority Queue

→ It is a collection of elements such as each element has been assigned a priority

→ The order in which elements are deleted and processed comes from the following rules

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

→ Priority queue can be implemented in two ways
* Using array
* using linked list

Array Representation of priority queue

→ Use separate queue for each level of priority

Linked List Representation of priority queue

a) Each node in the list will contain 3 items of information

1. An information field INFO
2. A priority number PRN
3. A link number LINK

b. A node x precedes a node y in the list-
* When x has higher priority than y or
* When both have same priority, but x was added to the list before y. This means the order in the one way list corresponds to the order of the priority queue.

Application of Queue.

1) In a multiuser s/m, task from a queue waiting to be executed one after another.
2) In computer n/w, the packets the arrive from various lines are kept in the queue.
3) For performing Breadth First Search (BFS) of a graph, queue is used
4) Queues are generally used for ordering events on FIFO basis

Linear Search.

Algorithm LinearSearch(a, n)

```
{
    k=1, loc=0;
    while(loc==0 and k<=n)
    {
        if (item == a[k] then
            loc=k;
            k=k+1;
    }
    if (loc==0)
```

print("Item is not found");
else
print("Item is in location - loc");
}

n = 5
item = 1

3	2	0	1	4
a[1]	a[2]	a[3]	a[0]	a[5]

Best case time complexity = O(1)
Worst case time complexity = O(n)
Average case time complexity -

Binary Search

- The average time complexity and worst-case time complexity can be reduced.
- The operation is performed on small list.
- ⇒ The list must be a sorted one.
- ⇒ Mid value of the sorted list is

$$(L+U)/2$$

L

3	8	11	17	20

U

- ⇒ Let x is the searching element.
- ⇒ If x < mid value, we consider only the left-portion of the list and we next consider that list.
- ⇒ Then we will find the mid value of that list.
- ⇒ Here time complexity can be reduced

Algorithm BinarySearch()

```
{
    l=0
    u=n-1;
while(l<=u)
    mid=(l+u)2
    if(x== A[mid]
        return mid
else if (x> A[mid]
        l=mid+1
    else
        u=mid-1
```

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

key=31 mid = (0+9)/2 = 4

10	14	19	26	27	31	33	35	42	44

↑ mid

27 < 31 => l = 31

mid = l+u/2 = (5+9)/2 = 7

10	14	19	26	27	31	33	35	42	44

↑ mid

31 < 35 => u = 33

mid = (5+6)/2 = 5

31 = 31

=> The best case is O(1) because the searching element may be the mid value.

=> The total list is reduced when divide each time

=> That is the search space is reduced.

1st bived- = n/2

2nd bived- n/2²

ith bived- n/2^i

Finally i- becomes 1

if n/2^i = 1, then O(log n)

n = 2^i Taking log on both sides

log n = i log 2

i = log n / log 2 => log₂ n

log^a / log^b = log_b a

Binary search is efficient- than linear search

Circular Queue - Enqueue

Insertion Algorithm (Enqueue)

if (front == -1 & rear == -1)

set front = 0 and rear = 0

a[rear] = item

else if (front = (rear + 1) % n then

print over flow

else

set rear = (rear + 1) % n

a[rear] = item

Deletion Algorithm (Dequeue)

if (front == -1 & rear == -1)

print under flow

exit

else if front == rear

item = a[front]

set front = -1 & rear = -1

else

set item = a[front]

front = (front + 1) % n

if ((front == 0 && rear == size - 1) || (front == rear + 1)) Overflow

Doubly Ended Queue

→ Insertion at rear end

Start

If rear = MAX - 1 then
print overflow
else
rear = rear + 1
set A[rear] = item
exit

→ Insertion at front end

Start

if front = 0 then
print overflow exit
else
set front = front - 1
A[front] = item.

→ Deletion at front end

Start

if front = 0 and rear = -1 then
print underflow exit
set item = A[front]
if front = rear then
set front = -1 and rear = -1
else set front = front + 1
exit

→ Deletion at rear end.

Start

if front = 0 and rear = -1
print underflow exit
set item = A[rear]
if front = rear then
set front = -1 and rear = -1
rear = rear - 1
exit