**UNIT-3: Linked Lists:**

**Single Linked List: Introduction, Representation, Operations, Applications.**

**Double Linked Lists: Introduction, Representation, Operations**

**Circular Linked Lists: Introduction, Representation, Operations**

One of the drawbacks of an array is that it is a static data structure, that is, the maximum capacity of an array should be known before the compilation process. Therefore, we must explicitly define its size before compilation there by reduces effective space utilization.

Another drawback of arrays is that the elements in an array are stored a fixed distance apart, and the insertion and deletion of elements in between require a lot of data movement.

The linked list is the solution to overcome all these problems. A linked list using dynamic memory management follows this principle—allocate and use memory when you need it and release it (free or deallocate)

A linked list is a very effective and efficient dynamic data structure for linear lists. Items may be added or deleted from it at any position more easily as compared to arrays. A programmer does not need to worry about how many data items a program will have to store. This enables the programmer to make effective use of the memory.

**Differences between arrays and linked lists**

| **Basis for Comparison** | **Array** | **Linked list** |
| --- | --- | --- |
| Basic | It is a consistent set of a fixed number of data items. | It is an ordered set comprising a variable number of data items. |
| Size | Specified during declaration. | No need to specify; grow and shrink during execution. |
| Storage Allocation | Element location is allocated during compile time. | Element position is assigned during run time. |
| Order of the elements | Stored consecutively | Stored randomly |
| Accessing the element | Direct or randomly accessed, i.e., Specify the array index or subscript. | Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer. |
| Insertion and deletion of element | Slow relatively as shifting is required. | Easier, fast and efficient. |
| Searching | Binary search and linear search | linear search |
| Memory required | less | More |
| Memory Utilization | Ineffective | Efficient |

**Linked Lists:**

Similar to arrays, **LinkedList is a linear data structure**. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointer or a link. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList. Each element in the LinkedList is called the **Node**. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

data next

Node

linked lists do not necessarily contain consecutive memory locations. These data items can be stored anywhere in the memory in a scattered manner.

To maintain the specific sequence of these data items, we need to maintain link(s) with a successor (and/or a predecessor). It is called as a *linked list* as each node is linked with its successor (and/or predecessor).

A *linked list* is an ordered collection of data in which each element (node) contains a minimum of two values, *data* and *link*(*s*) to its successor (and/or predecessor). A list with one link field using which every element is associated to its successor is known as a *singlylinked list* (SLL). A link is made from each item to the next item in the list

data 1 next

data 2 next

data3 NULL

HEAD Node1 Node2 Node3/TAIL

memory for a node is dynamically allocated. Therefore, the number of elements that may be added to a list is limited only by the amount of memory available.

Declaration of the node is,

class Node

{

Element\_type data;

Node next;

}

Some of the basic operations that can be performed on a list are

1. Create a list
2. Search for an element in a list
3. Delete an element from a list ( at beg, at end, at given location)
4. Add an element at a specified location of a list ( at beg, at end, at given location).
5. Sort a list
6. Print a list
7. Determine the size or no of elements in a list

**Implementation of linked list**

import java.util.\*;

class MyNode

{

int data;

MyNode next;

}

class MyLinkedList

{

MyNode head=null;

MyNode last=null;

Scanner s=new Scanner(System.in);

void create(int n)

{

MyNode temp;

for(int i=0;i<n;i++)

{

temp=new MyNode();

System.out.println("enter the data");

temp.data=s.nextInt();

temp.next=null;

if(head==null)

head=last=temp;

else

{

last.next=temp;

last=temp;

}

}

}

void display()

{

MyNode temp;

for(temp=head;temp!=null;temp=temp.next)

System.out.print(temp.data+"->");

System.out.println("NULL");

}

void insertB()

{

MyNode temp=new MyNode();

System.out.println("enter the element to be inserted");

temp.data=s.nextInt();

temp.next=head;

head=temp;

}

void insertE()

{

MyNode temp=new MyNode();

System.out.println("enter the element to be inserted");

temp.data=s.nextInt();

temp.next=null;

MyNode temp1;

for(temp1=head;temp1.next!=null;temp1=temp1.next);

temp1.next=temp;

//last.next=temp;

//last=temp;

}

void insertP(int p)

{

MyNode temp=new MyNode();

System.out.println("enter the element to be inserted");

temp.data=s.nextInt();

MyNode temp1;

int i;

for(i=0,temp1=head;i<p-1;i++,temp1=temp1.next);

temp.next=temp1.next;

temp1.next=temp;

}

void deleteB()

{

MyNode temp=head;

head=temp.next;

temp=null;

}

void deleteE()

{

MyNode temp,temp1;

temp=head;

temp1=null;

while(temp.next!=null)

{

temp1=temp;

temp=temp.next;

}

temp1.next=null;

temp=null;

last=temp1;

}

void deleteP(int p)

{

MyNode temp,temp1;

int i;

for(temp=head,i=0;i<p-1;i++,temp=temp.next);

temp1=temp.next;

temp.next=temp1.next;

temp1=null;

}

}

class LinkedListDemo

{

public static void main(String[] args)

{

int ch;

Scanner s=new Scanner(System.in);

MyLinkedList l=new MyLinkedList();

System.out.println("enter the size of the list");

l.create(s.nextInt());

while(true)

{

System.out.println("enter your choice");

ch=s.nextInt();

switch(ch)

{

case 1: l.insertB();

break;

case 2: l.insertE();

break;

case 3:System.out.println("enter the position");

l.insertP(s.nextInt());

break;

case 4:l.deleteB();

break;

case 5:l.deleteE();

break;

case 6: System.out.println("emter the position");

l.deleteP(s.nextInt());

break;

case 7:l.display();

break;

default:return;

}

}

}

}

**Advantages of linked lists:**

* Efficient implementation of insertion and deletion operations(they are performed at the complexity of O(1) where as in arrays O(n)).
* Efficient use of memory.
* Can dispose nodes after its use.
* Can expand the list in constant time without memory overhead

**Disadvantages:**

* Can’t access the nodes randomly.

**Applications of linked lists:**

**Polynomial Representation**

polynomial we want to represent using a linked list be *A*(*x*). It is expanded as,

*A*(*x*) = *k*1*xm* + … + *kn*-1 *x*2 + *knx*1

where *ki* is a non-zero coefficient with exponent *m* such that *m* *m* 1... 2 1 ≥0.

A node of the linked list will represent each term. A node will have 3 fields, which represent the coefficient and exponent of a term and a pointer to the next term

****

For instance, the polynomial, say *A* 6*x*7 3*x*5 4*x*3 12 would be stored as

****

The polynomial *B* 8*x*5 9*x*4 2*x*2 10 would be stored as

**Program for polynomial representation**

import java.util.\*;

class MyNode

{

int c,e;

MyNode next;

}

class MyLinkedList

{

Scanner s=new Scanner(System.in);

MyNode head,last;

MyLinkedList()

{

head=last=null;

}

void create(int n)

{

MyNode temp;

for(int i=0;i<n;i++)

{

temp=new MyNode();

System.out.println("enter coefficient and exponent");

temp.c=s.nextInt();

temp.e=s.nextInt();

temp.next=null;

if(head==null)

head=last=temp;

else

{

last.next=temp;

last=temp;

}

}

}

void add(MyLinkedList p1, MyLinkedList p2)

{

MyNode temp1=p1.head;

MyNode temp2=p2.head;

MyNode temp3;

while(temp1!=null && temp2!=null)

{

temp3=new MyNode();

if(temp1.e==temp2.e)

{

temp3.c=temp1.c+temp2.c;

temp3.e=temp1.e;

temp3.next=null;

temp1=temp1.next;

temp2=temp2.next;

}

else if(temp1.e>temp2.e)

{

temp3.c=temp1.c;

temp3.e=temp1.e;

temp3.next=null;

temp1=temp1.next;

}

else

{

temp3.c=temp2.c;

temp3.e=temp2.e;

temp3.next=null;

temp2=temp2.next;

}

if(head==null)

head=last=temp3;

else

{

last.next=temp3;

last=temp3;

}

}

while(temp1!=null)

{

temp3=new MyNode();

temp3.c=temp1.c;

temp3.e=temp1.e;

temp3.next=null;

last.next=temp3;

last=temp3;

temp1=temp1.next;

}

while(temp2!=null)

{

temp3=new MyNode();

temp3.c=temp2.c;

temp3.e=temp2.e;

temp3.next=null;

last.next=temp3;

last=temp3;

temp2=temp2.next;

}

}

void mul(MyLinkedList p1, MyLinkedList p2)

{

MyNode temp1=p1.head;

MyNode temp2;

MyNode temp3;

while(temp1!=null)

{

temp2=p2.head;

while(temp2!=null)

{

temp3=new MyNode();

temp3.c=temp1.c\*temp2.c;

temp3.e=temp1.e+temp2.e;

temp3.next=null;

if(head==null)

head=last=temp3;

else

{

last.next=temp3;

last=temp3;

}

temp2=temp2.next;

}

temp1=temp1.next;

}

temp1=head;

while(temp1!=null)

{

temp2=temp1;

temp3=temp1.next;

while(temp3!=null)

{

if(temp1.e==temp3.e)

{

temp1.c=temp1.c+temp3.c;

temp2.next=temp3.next;

temp3=temp2.next;

}

else

{

temp2=temp3;

temp3=temp3.next;

}

}

temp1=temp1.next;

}

}

void display()

{

MyNode temp;

for(temp=head;temp!=null;temp=temp.next)

{

if(temp.next!=null)

System.out.print(temp.c+"x^"+temp.e+"+");

else

System.out.println(temp.c+"x^"+temp.e);

}

}

}

class Poly

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

MyLinkedList p1=new MyLinkedList();

MyLinkedList p2=new MyLinkedList();

MyLinkedList p3=new MyLinkedList();

MyLinkedList p4=new MyLinkedList();

System.out.println("enter no.of terms in the polynomial equation1");

p1.create(s.nextInt());

System.out.println("enter no.of terms in the polynomial equation2");

p2.create(s.nextInt());

p1.display();

p2.display();

p3.add(p1,p2);

p3.display();

p4.mul(p1,p2);

p4.display();

}

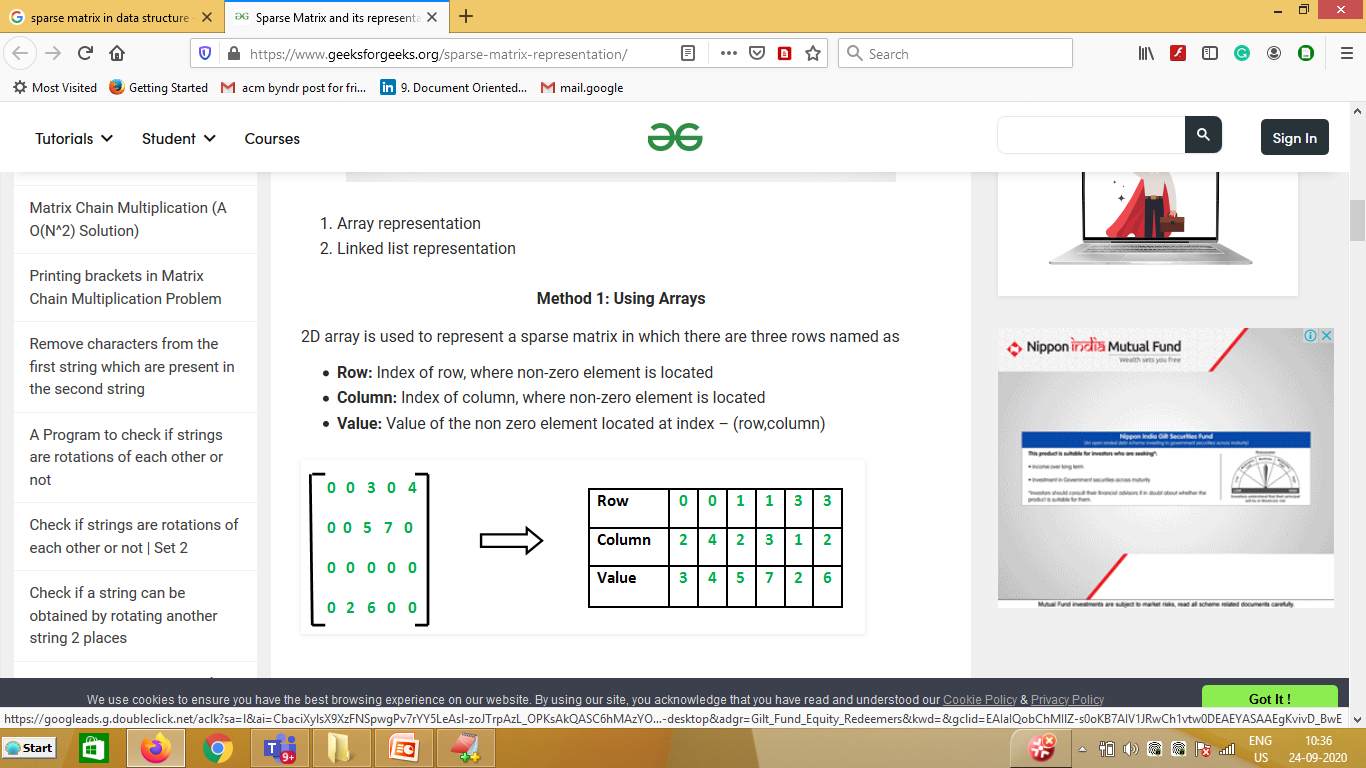
}

**Sparse Matrix Representation**

**Representation of sparse matrix using linked list:**

Sparse matrix is a matrix in which most of the elements are zeros. A matrix with a maximum of 1/3 non-zero elements. We can efficiently perform the operations with matrix representation. But this representation uses more space for storing zeros.

We have studied the sparse matrix representation using arrays, which is a sequential allocation scheme. Representing a sparse matrix sequentially allows faster execution of matrix operations, and it is more storage efficient than linked allocation schemes. However, it has many shortcomings. The insertion and deletion of elements need the movement of many other elements. In applications with frequent insertions and deletions, a linked representation can be adopted.



**Implementation o sparse matrix representation:**

class MyNode

{

int r,c,e;

MyNode next;

}

class SparseList

{

public static void main(String[] args)

{

int s[][]={{0,0,2},{0,0,0},{3,0,0}};

MyNode head=null;

MyNode last=null;

MyNode temp;

for(int i=0;i<3;i++)

{

for(int j=0;j<3;j++)

{

if(s[i][j]!=0)

{

temp=new MyNode();

temp.r=i;

temp.c=j;

temp.e=s[i][j];

temp.next=null;

if(head==null)

head=last=temp;

else

{

last.next=temp;

last=temp;

}

}

}

}

for(temp=head;temp!=null;temp=temp.next)

System.out.print(temp.r+"|"+temp.c+"|"+temp.e+"->");

System.out.println("null");

}

}

**Doubly linked list:**

Doubly linked list  is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The cost of this is an extra link, which adds to the space requirements and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand it simplifies deletion, because you no longer have to refer to a key by using pointers to the previous cell.

Declaration of the node is,

class Node

{

Node previous;

Element\_type data;

Node next;

}

A Sample Double Linked List



**Implementation of Double Linked List:**

import java.util.Scanner;

class MyNode

{

MyNode prev;

int data;

MyNode next;

}

class MyLinkedList

{

Scanner s=new Scanner(System.in);

MyNode head,last;

MyLinkedList()

{

head=null;

last=null;

}

void create(int n)

{

MyNode temp;

for(int i=0;i<n;i++)

{

temp=new MyNode();

System.out.println("enter data");

temp.data=s.nextInt();

temp.next=null;

if(head==null)

{

head=last=temp;

head.prev=null;

}

else

{

last.next=temp;

temp.prev=last;

last=temp;

}

}

}

void insertB()

{

MyNode temp=new MyNode();

System.out.println("enter data");

temp.data=s.nextInt();

temp.prev=null;

temp.next=head;

head.prev=temp;

head=temp;

}

void insertE()

{

MyNode temp=new MyNode();

MyNode temp1;

System.out.println("enter data");

temp.data=s.nextInt();

temp.next=null;

for(temp1=head;temp1.next!=null;temp1=temp1.next);

temp.prev=temp1;

temp1.next=temp;

last=temp;

}

void insertP(int p)

{

MyNode temp=new MyNode();

MyNode temp1;

int i;

System.out.println("enter data");

temp.data=s.nextInt();

for(i=0,temp1=head;i<p-1;temp1=temp1.next,i++);

temp.next=temp1.next;

temp.prev=temp1;

temp1.next.prev=temp;

temp1.next=temp;

}

void deleteB()

{

MyNode temp;

temp=head;

head=temp.next;

head.prev=null;

temp=null;

}

void deleteE()

{

MyNode temp,temp1;

for(temp=head;temp.next!=null;temp=temp.next);

temp1=temp.prev;

temp1.next=null;

temp=null;

last=temp1;

}

void deleteP(int p)

{

MyNode temp,temp1;

int i;

for(temp=head,i=0;i<p-1;i++,temp=temp.next);

temp1=temp.next;

temp.next=temp1.next;

temp1.next.prev=temp;

temp1=null;

}

void displayF()

{

MyNode temp;

for(temp=head;temp!=null;temp=temp.next)

System.out.print(temp.data+"->");

System.out.println("null");

}

void displayB()

{

MyNode temp;

for(temp=last;temp!=null;temp=temp.prev)

System.out.print(temp.data+"->");

System.out.println("null");

}

}

class DoubleLinkedListDemo

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

MyLinkedList l=new MyLinkedList();

System.out.println("enter the size o the list");

int ch;

l.create(s.nextInt());

while(true)

{

System.out.println("enter your choice");

ch=s.nextInt();

switch(ch)

{

case 1: l.insertB();

break;

case 2: l.insertE();

break;

case 3:System.out.println("enter the position");

l.insertP(s.nextInt());

break;

case 4:l.deleteB();

break;

case 5:l.deleteE();

break;

case 6: System.out.println("emter the position");

l.deleteP(s.nextInt());

break;

case 7:l.displayF();

break;

case 8:l.displayB();

break;

default:return;

}

}

}

}

**Advantages over singly linked list**  
**1)** A DLL can be traversed in both forward and backward direction.   
**2)** The delete operation in DLL is more efficient if pointer to the node to be deleted is given.  
**3)**We can quickly insert a new node before a given node.

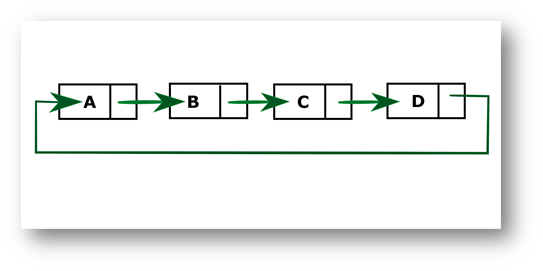
**Disadvantages over singly linked list**  
**1)** Every node of DLL Require extra space for an previous pointer.

**2)** All operations require an extra pointer previous to be maintained.

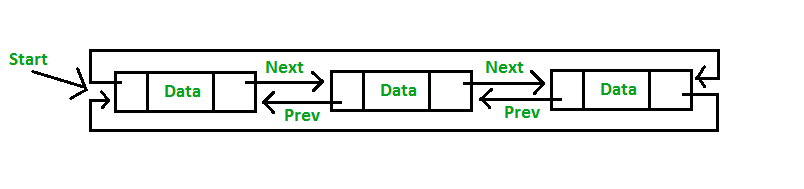
**Circular linked list**

Circular linked list is a linked list in which the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

Circular single linked list



Circular double linked list



**Implementation of circular linked list:**

import java.util.Scanner;

class MyNode

{

int data;

MyNode next;

}

class CircularLinkedList

{

Scanner s=new Scanner(System.in);

MyNode head;

CircularLinkedList()

{

head=null;

}

void create(int n)

{

MyNode temp,last=null;

for(int i=0;i<n;i++)

{

temp=new MyNode();

temp.data=s.nextInt();

if(head==null)

{

head=last=temp;

head.next=head;

}

else

{

last.next=temp;

temp.next=head;

last=temp;

}

}

}

void display()

{

MyNode temp;

for(temp=head;temp.next!=head;temp=temp.next)

System.out.print(temp.data+"->");

System.out.println(temp.data);

}

}

class CircularLinkedListDemo

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

CircularLinkedList c=new CircularLinkedList();

c.create(s.nextInt());

c.display();

}

}

**Linked Stacks:**

we have implemented stacks using arrays. However, an array implementation has certain limitations. One of the limitations is that such a stack cannot grow or shrink dynamically. This drawback can be overcome by using linked implementation.

A stack implemented using a linked list is also called *linked stack*. Each element of the stack will be represented as a node of the list. The addition and deletion of a node will be only at one end. The last element given is at the top of the stack, and it will be pointed to by a pointer called top. The first element is at the bottom of the stack, and its link field is set to Null. An empty stack will have Top = Null.



**Implementation of Stack using linked lists:**

import java.util.Scanner;

class MyNode

{

int data;

MyNode next;

}

class MyStack

{

MyNode top;

MyStack()

{

top=null;

}

void push(int e)

{

MyNode temp=new MyNode();

temp.data=e;

if(top==null)

{

top=temp;

top.next=null;

}

else

{

temp.next=top;

top=temp;

}

}

void pop()

{

if(top==null)

System.out.println("stack underflow");

else

{

System.out.println("the element to be deleted is "+top.data);

if(top.next!=null)

top=top.next;

else

top=null;

}

}

int peek()

{

if(top==null)

return -1;

else

return top.data;

}

void display()

{

MyNode temp;

for(temp=top;temp!=null;temp=temp.next)

System.out.print(temp.data+"<-");

System.out.println("null");

}

}

class LinkedStackDemo

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

MyStack stk=new MyStack();

int ch;

while(true)

{

System.out.println("enter your choice");

ch=s.nextInt();

switch(ch)

{

case 1:stk.push(s.nextInt());

break;

case 2:stk.pop();

break;

case 3:System.out.println(stk.peek());

break;

case 4:stk.display();

break;

default:return;

}

}

}

}

**Linked Queues**

we have a circular queue of fixed size. However, there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major drawback is the updating of front and rear.



**Implementation of Queue using linked lists**

import java.util.Scanner;

class MyNode

{

int data;

MyNode next;

}

class MyQueue

{

MyNode front,rear;

MyQueue()

{

front=rear=null;

}

void enQueue(int e)

{

MyNode temp=new MyNode();

temp.data=e;

temp.next=null;

if(front==null)

front=rear=temp;

else

{

rear.next=temp;

rear=temp;

}

}

void deQueue()

{

if(front!=null)

{

System.out.println("the element to be deleted is "+front.data);

if(front.next!=null)

front=front.next;

else

front=rear=null;

}

else

System.out.println("Queue underflow");

}

void display()

{

MyNode temp;

for(temp=front;temp!=null;temp=temp.next)

System.out.print(temp.data+"->");

System.out.println("null");

}

}

class LinkedQueueDemo

{

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

MyQueue q=new MyQueue();

int ch;

while(true)

{

System.out.println("enter your choice ");

ch=s.nextInt();

switch(ch)

{

case 1:q.enQueue(s.nextInt());

break;

case 2:q.deQueue();

break;

case 3:q.display();

break;

default:return;

}

}

}

}