Linked Lists

An array is a very useful data structure provided in programming languages. However, it has at least two limitations:

1. Changing the size of the array requires creating a new array and then copying all data from the array with the old size to the array with the new size.
2. The data in the array are next to each other sequentially in memory, which means that inserting an item inside the array requires shifting some other items in the array.

This limitation can be overcome by using linked structures. A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in the memory, and passing from one node to another is accomplished by storing the reference to other node in the structure.

Singly Linked List

If a node contains a data field that is a reference to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a **linked list,** which is a data structure composed of nodes, each node holding some information and a reference to another node in the list. If a node has a link only to its successor in this sequence, the list is called a **singly linked list.**

Insert a node at the beginning of SLL:

```
void insertAtFirst ()
{
        Node newNode = new Node (10);
        newNode.next = head;
        head = newNode;
}
```

Insert a node at the end of SLL:

```
void insertAtEnd ()
{
        Node newNode = new Node (10);
        Node current = head;
        while(current.next != null)
        {
                current = current.next;
        }
        current.next = newNode;
}
```

Insert a node after a given node in SLL:

```
void insertAtEnd ()
{
        Node newNode = new Node (10);
        newNode.next = previous.next;
        previous.next = newNode;
}
```

Delete first node from a SLL:

```
void deleteFirst()
{
        Node temp = head;
        head = head.next;
        temp.next = null;
}
```

Delete last node from a SLL:

```
void deleteLast()
{
        Node last = head;
        Node previousToLast = null;
        while(last.next != null)
        {
                previousToLast = last;
                last = last.next;
        }
        previousToLast.next = null;
}
```

Delete a node at a given position in a SLL:

```
void deleteAny()
{
        Node previous = head;
        Int count = 1;
        while(count <  position - 1)
        {
                previous = previous.next;
                count++;
        }
        Node current = previous.next;
        previous.next = current.next;
        current.next = null;
}
```

Java Implementation of singly linked list

This is a Java Program to implement a Singly Linked List. A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence. In a singly linked list each node has only one link which points to the next node in the list

The complete Java program to implement singly linked list is as follows:

```
class Node {
   public int info;
   public Node next;
   public Node(){
      next = null;
   }
   public Node(int el){
      info = el;
      next = null;
   }
   public Node(int el, Node ptr){
      info =el;
      next = ptr;
   }
}
class LinkList
{
   public Node head;
   public Node tail;
   public LinkList (){
      head = null;
      tail  = null;
   }
   public boolean isEmpty (){
      return head == null;
   }
   public void addToHead (int el) {
      head = new Node (el, head);
      if (tail == null)
      tail = head;
   }
   public void addToTail (int el) {
      if (! isEmpty ()) {
         tail.next = new Node(el);
         tail = tail.next;
```

```
        }
        else
            head = tail = new Node(el);
    }
    public void printAll () {
        Node temp;
        for (temp = head; temp! =null; temp = temp.next)
        System.out.print (temp.info+" ");
    }
    public int deleteFromHead () {
        if (isEmpty ())
            return 0;
        int el = head.info;
        if(head==tail)
            head = tail = null;
        else
            head = head.next;
        return el;
    }
    public int deleteFromTail (){
        if(isEmpty())
            return 0;
        int el = tail.info;
        if (head == tail)
            head = tail = null;
        else {
            Node temp;
            for (temp = head; temp.next! =tail; temp = temp.next);
            tail = temp;
            tail.next = null;
    }
        return el;
}
    public boolean isInList (int el){
        Node temp;
        for (temp = head; temp!=null && temp.info!=el; temp = temp.next);
        return temp! = null;
    }
    public void delete(int el){
        if(!isEmpty())
            if(head == tail&&el == head.info)
                head = tail=null;
```

```
        else if(el==head.info)
           head = head.next;
        else{
           Node pred,temp;
           for(pred = head,temp = head.next; temp!=null&&temp.info!=el;pred = pred.next,temp = temp.next);
           if(temp!=null){
           pred.next = temp.next;
           if(temp == tail)
           tail = pred;
           }
           }
 }
}
class LinkedList{
 public static void main(String[] args) {
      LinkList list = new LinkList();
      list.addToHead(20);
      list.addToHead(10);
      list.addToTail(30);
      System.out.println("Linked List ");
      list.printAll();
      list.delete(30);
      System.out.println ("\nLinked List after deletion");
      list.printAll();
      System.out.println ("\n30 is in list "+list.isInList(30));
   }
}
```

A node includes two fields: **info** and **next**. The i**nfo** field is used to store information. The next filed is used to join together nodes to form a linked list. It is an auxiliary filed used to maintain the linked list. It is indispensible for implementation of the linked list. The link field is declared as Node. It is a reference to a node of the same type that is being declared. Objects that include such a data fields are called self referential objects.

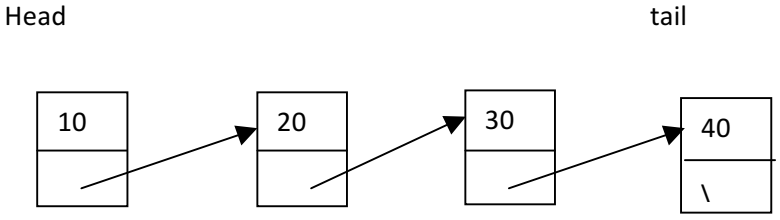

Figure: Singly linked list

Insertion

Adding a node at the beginning of a linked list is performed in four steps.

1. An empty node is created. It is empty in the sense that program performing insertion does not assign any values to the fields of the node.
2. The node's data filed is initialized to a particular integer.
3. Because the node is being included at the front of the list, the next filed becomes a reference to the first node on the list; that is, the current value of head.
4. The new node preceded all the nodes on the list, but this fact has to be reflected in the value of head; otherwise, the new node is accessible. Therefore, head is updated to become the reference to the new node.

The four steps are executed by the method addToHead (). The method executes the first three steps indirectly by calling the constructor Node (el, head). The last step is executed directly in the method by assigning the address (reference) of the newly created node to head.

The method addToHead () singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both head and tail are null; therefore, both become references to the only node of the new list. When inserting in a nonempty list, both head needs to be updated.

Adding a new node to the end of the list has five steps.

1. An empty node is created.
2. The node's data field is initialized to some value.
3. Because the node is being included at the end of the list, the next filed is set to null.
4. The node is now included in the list by making the next field of the last node of the list a reference to the newly created node.
5. The new node follows all the nodes of the list, but this fact has to be reflected in the value of tail or end, which now becomes the reference to the new node.

All these steps are executed in the if clause of the addToList () method. The else cause of this method is executed only if the linked list is empty. If this case were not included, the program would crash because in the if clause we make the assignment to the next filed of the node referred by tail. In the case of an empty linked list, it is a reference to a nonexisting node, which leads to throwing the NULLPointerException.

Deletion

One deletion operation consists of deleting a node at the beginning of the list and returning the value stored in it. In this operation the information from the first node is temporarily stored in some local variable and then head is reset so what was the second node becomes the first node. In this way, the former node is abandoned to be processed later by garbage collector. Because the head node is immediately accessible, deleteFromHead () takes constant time O(1) to perform its task.

The second deletion operation consists of deleting a node from the end of the list. After deletion, the last node has to be moved backward by one node.

The deletion operation can be summarized in the following points.

1. An attempt to remove a node from an empty list, in which case the method is immediately exited.
2. Deleting the only node from a one node linked list. Both head and tail are set to null.

3. Removing the first node of the list with at least two heads, which requires updating head.
4. Removing the last node of the list with at least two nodes, leading to the update of tail.
5. An attempt to delete a node with a number that is not in the list: Do nothing.

It is noted that the best case for delete () is when the head node is to be deleted, which takes O(1) time to accomplish. The worst case is when the last node needs to be deleted, which reduces deleteFromTail () to O(n) performance.

Search

The insertion and deletion operations modify linked lists. The searching operation scans an existing list to learn whether a number is in it. We implement this operation with the Boolean method isInList(). The method uses a temporary variable temp to through the list starting from the head node. The number stored in each node is compared to the number being sought, and if the two numbers are equal, the loop is exited; otherwise, temp is updated to temp.link so that the next node can be investigated. After reaching the last node and executing the assignment temp = temp.link, temp becomes null, which is used as an indication that the number el is not in the list. That is, if temp is not null, the search is discontinued somewhere inside the list because el was found. That is why isInList () returns the result of comparison temp! =null. If temp is not null, el was found and true is returned. If temp is null, the search was unsuccessful and false is returned.

With reasoning very similar to that used to determine the efficiency of delete (), isInList () takes O(1) time in best case and O(n) in the worst and average case.

```
void search()
{
        Node current = head;
        while(current != null)
        {
                if(current.data == searchKey)
                {
                        return true;
                }
                current = current.next;
        }

        return false;

}
```

Doubly Linked Lists

A doubly linked list (DLL) is similar to singly linked list and it has two links one pointing to next node and other pointing to previous node.

One problem in singly linked list is that a node in the list has no knowledge about its previous node. We cannot go back to previous node from the current node. This problem is overcome is doubly linked list.

Methods for processing doubly linked lists are slightly more complicated than their singly linked counterparts because there is one more reference filed to be maintained.

To add a node to a list, the node has to be created, its fields properly initialized and then the node needs to be incorporated into the list. To insert a node at the end of the list we perform following five steps

- A new node is created and then its three fields are initialized.
- The info filed field to the number el being inserted.
- The next filed to null.
- The prev field to the value of tail so that this field refers to the last node in the list. But now, the new node should become the last node; therefore, tail is set to reference the new node. But the new node is not yet accessible from its predecessor, to rectify this.
- The next field of the predecessor is set to reference the new node.

Insert a node at the beginning of DLL:

```
void insertAtFirst ()
{
        Node newNode = new Node (10);
        if (isEmpty)              //if list is empty
        {
                last = newNode;
        }
        else
        {
                head.previous =  newNode;
                newNode.next = head;
                head = newNode;
        }
}
```

Insert a node after n position of DLL:

```
void insertAtAny ()
{
        Node newNode = new Node (10);
        if (current = last)
        {
                newNode.next = null;
                last = newNode;
        }
        else
```

```
        {
                newNode.next = current.next;
                current.next.previous = newNode;
        }
        newNode.previous = current;
        current.next = newNode;
}
```

The following program illustrates the concept of doubly linked list

```
class DLLNode
{
   public int info;
   public DLLNode next;
   public DLLNode prev;
   public DLLNode()
   {
      info = 0;
     next = null;
     prev = null;
   }
   public DLLNode(int el)
   {
     info = el;
     next = null;
     prev = null;
   }
   public DLLNode(int el,DLLNode n,DLLNode p)
   {
     info = el;
     next = n;   prev = p;
   }
}
class DoublyLinkedList{
   private DLLNode head, tail;
   public DoublyLinkedList()
   {
      head = null;
      tail = null;
   }
   public boolean isEmpty()
   {
      return head ==null;
```

```
}
public void setToNull()
{
   head =null;
   tail =null;
}
public int firstElement()
{
   if(head!=null)
      return head.info;
   else
      return 0;
}
public void addToHead(int el)
{
   if(head!=null)
   {
      DLLNode temp= new DLLNode(el);
      temp.next = head;
      temp.prev = null;
      head = temp;
   }
   else
      head = tail = new DLLNode(el);
}
public void addToTail(int el)
{
   if(tail!=null)
   {
      DLLNode temp = new DLLNode(el);
      tail.next = temp;
      temp.prev = tail;
      tail = temp;
   }
   else
      head = tail = new DLLNode(el);
}
public int deleteFromHead()
{
   if(isEmpty())
      return 0;
   int el = head.info;
```

```
        if(head==tail)
            head = tail = null;
        else
        {
            head = head.next;
            head.prev = null;
        }
    }
    public int deleteFromTail()
    {
        if(isEmpty()){
            return 0;
        }
        int el = tail.info;
        if(head==tail)
        head = tail = null;
        else{
            tail = tail.prev;
            tail.next = null;
        }
    }
    public boolean find(int el)
    {
        DLLNode temp;
        for(temp = head;temp!=null&&temp.info!=el;temp = temp.next);
        return temp!=null;
    }
    public void printALL()
    {
        DLLNode temp;
        for(temp = head;temp!=null;temp = temp.next)
            System.out.print(temp.info+ " ");
    }
}
public class DoublyLinkedListDemo {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        list.addToHead(30);
        list.addToHead(20);
        list.addToHead(10);
        list.addToHead(0);
        list.addToTail(40);
```

```
        System.out.println("Doubly Linked List");
        list.printALL();
        System.out.println(" 0 is in list "+list.find(10));
    }
}
```

Deleting the last node from the doubly linked list is straightforward because there is direct access from the last node to its predecessor, and no loop is needed to remove the last node.

Circular Linked List

In some situations, a circular list is needed in which nodes form a ring. The list is finite and each node has a successor. An example of such a situation is when several processes are using the same resource for the same amount of time and we have to assure that each process has a fair share of the resources. Therefore, all processes –let their number be 6 , 5, 8 and 10 are put a circular list accessible through current. After one node in the list is accessed and the process number is retrieved from the node to activate this process, current moves to the next node so that the next node to activate this process, current moves to the next node so that the next process can be activated the next time.

In an implementation of a circular linked list, we can use only one permanent reference, tail, to the list even though operations on the list require access to the tail and its successor, the head.

The implementation has some problem. A method for deletion of the tail node requires a loop so that tail can be set to its predecessor after deleting the node. This makes this method delete the tail node in O(n) time. Moreover, processing data in the reverse order is not very efficient. To avoid the problem and still be able to insert and delete nodes at the front and at the end of the list without using a loop, a doubly linked circular list can be used. The list forms two rings: one going forward through next fields and one going backward through prev fields. Deleting the node from the end of the list can be done easily because there is a direct access to the next to last node that needs to be updated in case of such a deletion. In this list, both insertion and deletion of the tail node can be node in O(1) time.

Insert a node at the beginning of CLL:

```
void addToFirst (int el)
{
        Node newNode = new Node (el);
        newNode.next = head;
        if(head == null)
        {
                head = newNode;
                tail = head;
        }
        else
        {
                tail.next = newNode;
                head = newNode;
        }
}
```

Insert a node after n position of CLL:

```
void addTotail (int el)
{
        Node newNode = new Node(el)
        newNode.next = head;
        if (head == null)
        {
                head = newNode;
                newNode.next = head;
                tail = newNode;
        }
        else
        {
                tail.next = newNode;
                tail = newNode;
        }
}
```

Delete a node at the beginning of CLL:

```
void deleteAtFirst ()
{
        newNode.next = head;
        if (head == tail)
        {
                head = tail = null;
        }
        else
        {
                head = head.next;
                tail.next = head;
        }
}
```

Delete a node at the end of CLL:

```
void deleteAtLast ()
{
        if (head == tail)
        {
                head = tail = null;
        }
        else
        {
```

```
                temp = head;
                while (temp.next != tail)
                {
                        temp = temp.next;
                }
                tail = temp;
                tail.next = head;
        }
}
```

The following program implements the singly circular liked list.

```
class CNode {
   public int info;
   public CNode next;
   public CNode(){
      info =0;
      next = null;
   }
   public CNode(int el){
      info = el;
      next = null;
   }
   public CNode(int el,CNode n){
      info = el;
      next = n;
   }
}
class CLinkList
{
   public CNode head;
   public CNode tail;
   public CLinkList(){
     head =  tail = null;
   }
   public boolean isEmpty(){
            return tail == null;
   }
   public void addToHead(int el)
   {
      CNode temp = new CNode(el);
      temp.next = head;
```

```
    if(head == null)
    {
       head = temp;
       //temp.next = head;
       tail = head;
    }
    else
    {
       tail.next = temp;
       head = temp;
    }
 }
 public void addToTail(int el){
 CNode temp = new CNode(el);
 temp.next = head;
 if(head==null){
    head = temp;
    temp.next = head;
    tail = temp;
 }
 else
 {
    tail.next =temp;
    tail =temp;
 }
}
 public void deleteAtHead(){
    CNode temp;
    if(head==tail)
       head = tail = null;
    else
    {
       head = head.next;
       tail.next = head;

    }
 }
 public void deleteAtTail(){
    CNode temp;
    if(head==tail)
       head = tail = null;
    else
```

```
      {
         for(temp = head;temp.next!=tail;temp = temp.next);
         tail = temp;
         tail.next = head;
      }

   }
      public void printAll(){
      CNode temp;
      for(temp = head;temp.next!=head;temp = temp.next)
      System.out.print(temp.info+"->");
      System.out.println(temp.info);
   }
}
class CicularLinkedList{
 public static void main(String[] args) {
      CLinkList list = new CLinkList();
      list.addToTail(50);
      list.addToHead(40);
      list.addToHead(30);
      list.addToHead(20);
      list.addToHead(10);
      list.addToTail(60);
      list.addToTail(70);
      list.addToTail(80);
      list.addToTail(90);
      list.addToTail(100);
      System.out.println("Circular Linked before deletion ");
      list.printAll();
      list.deleteAtHead();
      list.deleteAtTail();
      System.out.println("Circular Linked List after deletion at both head and tail ");
      list.printAll();
      }
}
```

The output of the above program is:

Circular Linked before deletion

10->20->30->40->50->60->70->80->90->100

Circular Linked List after deletion at both head and tail

20->30->40->50->60->70->80->90

The doubly circular linked List can be implemented in java as follows:

```
class DCNode {
```

```
    public int info;
    public DCNode next;
    public DCNode prev;
    public DCNode(){
        info =0;
        next = null;
        prev = null;
    }
    public DCNode(int el){
        info = el;
        next = null;
        prev = null;
    }
}
class DCLinkList
{
    public DCNode head;
    public DCNode tail;
    public DCLinkList(){
      head =  tail = null;
    }
    public boolean isEmpty(){
                return head == null;
    }
    public void addToHead(int el)
    {
        DCNode temp = new DCNode(el);
        temp.next = head;
        if(head == null)
        {
            head = temp;
            head.prev = temp;
            tail = temp;
            tail.next =temp;
        }
        else
        {
            head.prev = temp;
            tail.next = temp;
            temp.prev = tail;
            head = temp;
        }
```

```
}
public void addToTail(int el){
DCNode temp = new DCNode(el);
temp.next = head;
if(head==null){
   head = temp;
   head.prev = temp;
   tail = temp;
   tail.next = temp;
}
else
{
  head.prev = temp;
  tail.next =temp;
  temp.prev = tail;
  tail =temp;
}
}
public void deleteAtHead(){
   if(head==tail)
      head = tail = null;
   else
   {
      head = head.next;
      head.prev = tail;
      tail.next = head;

   }
}
public void deleteAtTail(){
   if(head==tail)
      head = tail = null;
   else
   {
      tail = tail.prev;
      head.prev = tail;
      tail.next = head;
   }

}
public boolean find(int el){
   DCNode temp;
```

```
        boolean t= false;
        for(temp=head;temp.next!=head;temp=temp.next)
        {
           if(temp.info==el)
           {
              t=true;
              break;
           }
        }
        return t;
    }
        public void printAll(){
        DCNode temp;
        for(temp = tail;temp!=head;temp = temp.prev)
        System.out.print(temp.info+"->");
        System.out.println(temp.info);
    }
 }
class DoublyCicularLinkedList{
public static void main(String[] args) {
        DCLinkList list = new DCLinkList();
        list.addToHead(4);
        list.addToHead(3);
        list.addToHead(2);
        list.addToHead(1);
        list.addToTail(5);
        list.addToTail(6);
        list.addToTail(7);
        list.addToTail(8);
        list.addToTail(9);
        list.addToTail(10);
        System.out.println("Circular Linked before deletion ");
        list.printAll();
        list.deleteAtHead();
        list.deleteAtTail();
        System.out.println("Circular Linked after deletion at both ends ");
        list.printAll();
        System.out.println("The data 7 is in list "+list.find(17));
    }
}
```

The output of the above program is:

Doubly Circular Linked before deletion

10->9->8->7->6->5->4->3->2->1
Doubly Circular Linked after deletion at both ends
9->8->7->6->5->4->3->2
The data 7 is in list false

Skip Lists

Linked lists have one serious drawback. They require sequential scanning to locate a searched for element. The search starts from the beginning of the list and stops when either a searched for element is found or the end of the list is reached without finding this element. Ordering elements on the list can speed up searching, but a sequential searching is still required. Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing. A skip list is an interesting variant of the ordered linked list that makes such a non sequential search possible.

In a skip list of n nodes, for each k and i such that $1 \le k \le \log n$ and $[1 \le i \le n/2^{k-1}]$, the node in position $2^{k-1}.i$ points to the node in position $2^{k-1}.(i+1)$. This means that every second node points to the node two positions ahead, every fourth node points to the node four positions ahead, and so on. This is accomplished by having different numbers of reference fields in nodes on the list. Half of the nodes have just one reference field, one fourth of the nodes have two reference fields, one eighth of the nodes have three reference fields, and so on. The number of reference fields indicates the level of each node, and the number of levels is maxLevel = [logn]+1.

Searching for an element **el** consists of following the references on the highest level until an element is found that finishes the search successful. In the case of reaching the end of the list or encountering an element key that is greater than el, the search is restarted from the node preceding the one containing key, but this time starting from a reference on a lower level than before. The search continues until el is found, or the first level references are followed to reach the end of the list or find an element greater than el.

Searching appears to be efficient, however, the design of skip lists can lead to very inefficient insertion and deletion procedures. To insert a new element, all nodes following the node just inserted have to be restructured; the number of reference fields and value of references have to be changed. In order to retain some of the advantages that skip lists offers with respect to searching and to avoid problems with restructuring the lists when inserting and deleting nodes, the requirement on the positions of nodes of different levels is now abandoned and only the requirement on the number of nodes of different levels is kept. The new least is searched exactly the same way as the original list. Inserting does not require list restructuring, and nodes are generated so that the distribution of the nodes on different levels is kept adequate.

Efficiency of Skip List

In the ideal situation, the search time O(logn). In the worst situation, when all lists are on the same level, the skip list turns into a regular singly linked list, and the search time is O(n). However, the latter situation is unlikely to occur; in the random skip list, the search is of the same order as the best case; that is, O(logn). This is an improvement over the efficiency of searching regular linked list.

The following illustrates the skip list

```
import java.util.Random;
class SkipListNode{
   public int key;
   public SkipListNode[] next;
   SkipListNode(int i, int n){
      key = i;
      next = new SkipListNode[n];
      for(int j =0;j<n;j++)
      next[j] = null;
   }
}
class SkipList{
   private int maxLevel;
   private SkipListNode[] root;
   private int[] powers;
   private Random rd = new Random();
   public SkipList(){}
   public SkipList(int i){
      maxLevel = i;
      root = new SkipListNode[maxLevel];
      powers = new int[maxLevel];
      for(int j=0;j<maxLevel;j++)
      root[j] = null;
      choosePowers();
   }
   public boolean isEmpty(){
   return root[0]==null;
   }
   public void choosePowers(){
      powers[maxLevel-1] = (2<<(maxLevel-1))-1;
      for(int i = maxLevel-2,j=0;i>=0;i--,j++)
      powers[i] = powers[i+1]-(2<<j);
   }
   public int chooseLevel(){
      int i,r = Math.abs(rd.nextInt())%powers[maxLevel-1]+1;
      for(i=1;i<maxLevel;i++)
         if(r<powers[i])
            return i-1;
      return i-1;
   }
   public int search(int key){
      int lvl;
```

```
    SkipListNode prev,curr;
    for(lvl = maxLevel-1;lvl>=0&&root[lvl]==null;lvl--);
    prev = curr = root[lvl];
    while(true){
      if(key==curr.key)
        return curr.key;
      else if(key<curr.key){
        if(lvl == 0)
          return 0;
        else if(curr == root[lvl])
          curr = root[--lvl];
        else curr = prev.next[--lvl];
      }
      else{
        prev = curr;
        if(curr.next[lvl]!=null)
          curr = curr.next[lvl];
        else
        {
          for(lvl--;lvl>=0&&curr.next[lvl]==null;lvl--);
          if(lvl>=0)
            curr = curr.next[lvl];
          else return 0;
        }
      }
    }
  }
  public void insert(int key){
    SkipListNode [] curr = new SkipListNode[maxLevel];
    SkipListNode [] prev = new SkipListNode[maxLevel];
    SkipListNode newNode;
    int lvl,i;
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = null;
    for(lvl = maxLevel-1;lvl>=0;lvl--){
      while(curr[lvl]!=null&&curr[lvl].key<key){
        prev[lvl] = curr[lvl];
        curr[lvl] = curr[lvl].next[lvl];
      }
      if(curr[lvl]!=null&&curr[lvl].key==key)
        return;
      if(lvl>0)
```

```
            if(prev[lvl]==null){
               curr[lvl-1] = root[lvl-1];
               prev[lvl-1] = null;
            }
            else{
               curr[lvl-1] = prev[lvl].next[lvl-1];
               prev[lvl-1] = prev[lvl];
            }
      }
      lvl = chooseLevel();
      newNode = new SkipListNode(key,lvl+1);
      for(i=0;i<=lvl;i++){
      newNode.next[i] = curr[i];
      if(prev[i]==null)
         root[i] = newNode;
      else prev[i].next[i] = newNode;
      }
   }
}
public class SkipListDemo {
   public static void main(String[] args) {
      SkipList list = new SkipList(4);
      list.insert(30);
      list.insert(20);
      list.insert(10);
      list.insert(0);
      list.insert(40);
      list.insert(50);
      list.insert(60);
      list.insert(70);
      System.out.println("The number 10 is in list "+list.search(100));
   }
}
```

Self organizing List

The introduction of skip lists was motivated by the need to speed up the searching process. Although singly and doubly linked lists require sequential search to locate an element to see that it is not in the list, we can improve the efficiency of the searched by dynamically organizing the list in a certain manner. The organization depends on the configuration of data; thus, the stream of data requires re-organizing the nodes already on the list. There are many ways to organize the lists as below:

1. Move to front method: After the desired element is located, put it at the beginning of the list.

2. Transpose method: After the desired element is located, swap it with its predecessor unless it is at the head of the list.
3. Count method: Order the list by the number of times elements are being accessed.
4. Ordering method: Ordering the list using certain criteria natural for the information under scrutiny.

In the first three methods, new information is stored in a node added to the end of the list. In the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list. An example of searching for an element in a list organized by these different methods is shown in the figure below:

S.N	Element searched For	Plain	Move to Front	Transpose	Count	Ordering
1	A	A	A	A	A	A
2	C	C	AC	AC	AC	AC
3	B	ACB	ACB	ACB	ACB	ABC
4	C	ACB	CAB	CAB	CAB	ABC
5	D	ACBD	CABD	CABD	CABD	ABCD
6	A	ACBD	ACBD	ACBD	CABD	ABCD
7	A	ACBD	DACB	ACBD	DCAB	ABCD
8	D	ACBD	ADCB	CADB	ADCB	ABCD
9	A	ACBD	CADB	ACDB	CADB	ABCD
10	C	ACBD	ACDB	CADB	ACDB	ABCD
11	A	ACBD	CADB	ACDB	ACDB	ABCD
12	C	ACBD	CADB	CADB	CADB	ABCD
13	C	ACBD	CADB	CADB	CADB	ABCD
14	E	ACBDE	CADBE	CADBE	CADBE	ABCDE
15	E	ACBDE	CADBA	CADEB	CAEDB	ABCDE

With the first three methods, we try to place the elements most likely to be looked for near the beginning of the list, most explicitly with the move to front method and most cautiously with the transpose method. The ordering method uses some properties inherent to the information stored in the list. For example, if we are storing nodes pertaining to people, then the list can be organized alphabetically by the name of the person, or city or in ascending or descending order using say, birthday or salary. This is particularly advantageous when searching for information that is not in the list, because the search can terminate without scanning the entire list. Searching all the nodes of the list, however, is necessary in such cases using the other three methods. The count method can be subsumed in the category of the ordering methods if frequency is part of the information.

Analysis of the efficiency of these methods customarily compare their efficiency to that of optimal static ordering.

Sparse Tables

In many applications, the choice of a table seems to be the most natural one, but space considerations may preclude this choice. This is particularly true if only a small fraction of the table is actually used. A table of this type is called a sparse table because the table is populated sparsely by data and most of its cells are empty. In this case, the table is replaced by a system of linked lists.

As an example, consider the problem of storing grades of storing for all students in a university for a certain semester. Assume that there are 8000 students and 300 classes. A natural implementation is two dimensional array grades where student numbers are indexes of the columns and class numbers are indexes of rows. Any association of student names and numbers is represented by the one dimensional array **students** and an association of class names and numbers is represented by the array **classes.**

Each cell of grades stores a grade obtained by each student after finishing a class.

The entire table occupies 8000 students.300 classes. Byte = 2..4 million bytes.

This table is very large but is sparsely populated by grades. Assuming that, on the average, students take four classes a semester, each column of the table has only four cells occupied by grades, and the rest of the cells, 296 cells or 98.7 are unoccupied and wasted.

Students

0	Sheaver Geo
1	Weaver Henry
2	Shelton Mary
3	……..
…..	……..
404	Crawford William
405	Lawson Earl
….	
…..	
5206	Fulton Jenny
5207	Craft Donald
5208	Oates Key
……	

Classes

0	Anatomy/Physiology
1	Introduction to Microbiology
2	…….
30	Advanced Writing
31	Chaucer
….	…….
…..	…….
115	Data Structures
116	Cryptology
117	Computer Ethics
…..	……

GradesCodes

0	A
1	A-
2	B+
3	B
4	B-
5	C+
6	C
7	C-
8	D
9	F

Student

Class

Index	0	1	2	….	404	405	….	5206	5207	5208	…..	7999
0										3		
1	1		4		7			1				
…												
….												
30		5										
31	0					5						
….												
115			0		4				5			
116			3									
117												

….												
299												