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.**

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

Node class of Singly Linked List:

Fig: A Node

```
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;
        }
}
```

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 indispensable 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 data fields are called self-referential objects.

Figure: Singly linked list

Insertion of a Node in a Singly Linked List

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 non-existing node, which leads to throwing the NULLPointerException.

Singly Linked List (SLL):

Insert a node at the beginning of SLL:

```
void addToHead(int el)
{
        Node xyz = new Node(el);
        if(isEmpty())
        {
                head = tail = xyz;
        }
        else
        {
                xyz.next = head;
                head = xyz;
        }
}
```

Insert a node at any position of SLL:

```
public void addToAny(int el, int n)
{
        Node xyz = new Node(el);
        Node temp = head;
        int count=1, position=n;
        while (count < position - 1)
        {
                temp = temp.next;
                count++;
        }
        Node current = temp.next;
        xyz.next = current;
        temp.next = xyz;
}
```

Insert a node at the end of SLL:

```
public void addToTail(int el)
{
        Node newnode = new Node(el);
        Node current = head;
        if(isEmpty())
        {
                head = tail = newnode;
        }
        else
        {
                While (current.next != null)
                        current = current.next;
                current.next = newnode;
                newnode.next = null;
        }
}
```

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.**

Delete a node from the beginning of SLL:

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

Delete a node from any position of SLL:

```
public void deleteAtAny(int n)
{
        Node previous = head;
        int count=1, position=n;
        while(count < position - 1)
        {
                previous = previous.next;
                count++;
        }
        Node current = previous.next;
        previous.next = current.next;
        current.next = null;
}
```

Delete a node from the end of SLL:

```
public void deleteAtTail()
{
        Node temp = head;
        Node current = null;
        while(temp.next != null)
        {
                current = temp;
                temp = temp.next;
        }
        current.next = null;
}
```

Search

- **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 being search is not in the list.
- That is, if temp is not null and data is found, then search is discontinued. That is why isInList () returns data is 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_el(int key)
{
        Node temp = head;
        while(temp != null)
        {
                if(temp.data != key)
                {
                        temp = temp.next;
                }
                else
                {
                        System.out.println(key+" is found.");
                        return;
                }
        }
        System.out.println(key+" is not found.");
}
```

Traversing a Linked List: visiting each node of a linked list exactly once.

Traversing a Linked List:

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

Method to display the data stored in each node in a linked list:

```
Node temp = head;
while (temp != null)
{
        System.out.println(temp.info + " ");
        temp = temp.next;
}
```

Doubly Linked Lists (DLL)

(a) A Node

(b) A Doubly Linked List

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.

The Node class of Doubly Linked List:

```
class Node
{
        public int info;
        public Node next;
        public Node prev;
        public Node()
        {
                next = null; prev = null;
        }
        public Node(int el)
        {
                info = el;  next = null;  prev = null;
        }
        public Node(int el, Node n, Node p)
        {
                info = el;  next = n;   prev = p;
        }
}
```

Insert a node at the beginning of DLL:

```
void insertAtFirst (el)
{
        Node abc = new Node (el);
        if (isEmpty())              //if list is empty
        {
                head = tail = abc;
        }
        else
        {
                head.previous =  abc;
                abc.next = head;
                head = abc;
        }
}
```

Insert a node at the end of DLL:

```
void addToLast(el)
{
        Node abc = new Node (el);
        Node current = tail;
        if (isEmpty())
        {
                head = tail = abc;
        }
        else
        {
                abc.next = null;
                tail = abc;
                abc.previous = current;
                current.next = abc;
        }
}
```

Delete a node at the beginning of DLL:

```
void delete_first()
{
        Node temp = head;
        if(head == tail)
                head = tail =null;
        else
        {
                head = head.next;
                temp.next = null;
                head.previous = null;
        }
}
```

Delete a node at the end of DLL:

```
void delete_last()
{
        Node temp = tail;
        if(head == tail)
                head = tail =null;
        else
        {
                tail = tail.previous;
                tail.next = null;
                temp.previous = null;
        }
}
```

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.

Question 1: Write function to Insert a node at any particular position of a Doubly Linked List.

Question 2: Write function to delete a node from any particular position of a Doubly Linked List.

// Insert at any particular position of DLL

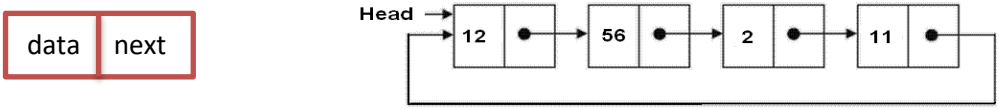
```
void addToAny(int el, int n)
{
        Node newNode = new Node(el);
        Node temp = head;
        int count = 1, position = n;
        if(isEmpty())
        {
                head = tail =newNode;
                newNode.next = null;
                newNode.previous = null;
        }
        else
        {
                while(count < position-1)
                {
                        temp = temp.next;
                        count++;
                }
                newNode.next = temp.next;
                temp.next.previous = newNode;
                temp.next = newNode;
                newNode.previous = temp;
        }
}
```

// Delete a node from any position

```
void delete_Any(int n)
{
        Node temp = head;
        int count = 1, position = n;
        if(head == tail)
                head = tail = null;
        else
        {
                while(count < position-1)
                {
                        temp = temp.next;
                        count++;
                }
                Node current = temp.next;
                temp.next = current.next;
                current.next.previous = temp;
        }
}
```

Circular Linked List

(i) Circular Singly Linked List:



(a) A Node in CSLL (b) A Circular Singly Linked List (CSLL)

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.

1. Circular Singly Linked List(CSLL):

The Node class of Circular Singly Liked List:

```
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 n)
        {
                info = el;  next = n;
        }
}
```

Insert a node at the beginning of CSLL:

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

Insert a node at the last position of CSLL:

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

Insert a node at any position of CSLL:

```
void addToAny(int el, int n)
{
        Node abc = new Node(el);
        Node temp = head;
        int count=1, position=n;
        while(count < position - 1)
        {
                temp = temp.next;
                count++;
        }
        Node current = temp.next;
        abc.next = current;
        temp.next = abc;
}
```

Delete a node at the beginning of CLL:

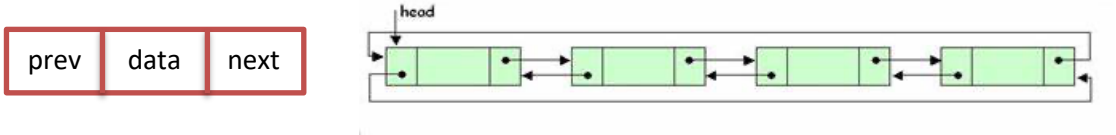
```
public void deleteAtHead()
{
        if(head==tail)
                head = tail = null;
        else
        {
                head = head.next;
                tail.next = head;
        }
}
```

Delete a node at the end of CLL:

```
void deleteAtLast ()
{
        Node temp = head;
        if (head == tail)
        {
                head = tail = null;
        }
        else
        {
                while (temp.next != tail)
                {
                        temp = temp.next;
                }
                tail = temp;
                tail.next = head;
        }
}
```

Question: Write function to delete a node from any particular position of a Circular Linked List.

Circular Doubly Linked List(CDLL):



(a) A Node in CDLL (b) A Circular Doubly Linked List

⇒ Circular Doubly Linked List is a Doubly Linked List where the next link of last node points to the first node and previous link of first node points to last node of the list.

⇒ The main objective of Circular Doubly Linked List is to simplify the insertion and deletion operations performed on Doubly Linked List.

The Node class of Doubly Circular Liked List:

```
class Node
{
        public int info;
        public Node next;
        public Node prev;
        public Node()
        {
                next = null; prev = null;
        }
        public Node(int el)
        {
                info = el; next = null; prev = null;
        }
        public Node(int el, Node n, Node p)
        {
                info = el;  next = n; prev = p;
        }
}
```

Insert a node at the beginning of DCLL:

```
public void addToHead(int el)
{
      Node abc = new Node(el);
      if(isEmpty())
      {
         head = tail = abc;
         head.prev = abc;
         tail.next =abc;
      }
      else
      {
         abc.next = head;
         head.prev = abc;
         tail.next = abc;
         abc.prev = tail;
         head = abc;
      }
}
```

Insert a node at the ent of DCLL:

```
public void addToTail(int el)
{
      Node abc = new Node(el);
      if(isEmpty())
      {
         head = tail = abc;
         head.prev = abc;
         tail.next =abc;
      }
      else
      {
         abc.next = head;
         head.prev = abc;
         tail.next = abc;
         abc.prev = tail;
         tail = abc;
      }
}
```

Delete a node from the beginning of DCLL:

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

Delete a node from the end of DCLL:

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

Skip Lists:

- **A skip list is an interesting variant of the ordered linked list that makes such a nonsequential search possible.**
- 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.
- In a skip list of n nodes, for each k and i such that $1 \le k \le \log n$ and $\lfloor 1 \le i \le n/2^{k-1} \rfloor$, 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 = $\lfloor \log n \rfloor + 1$.

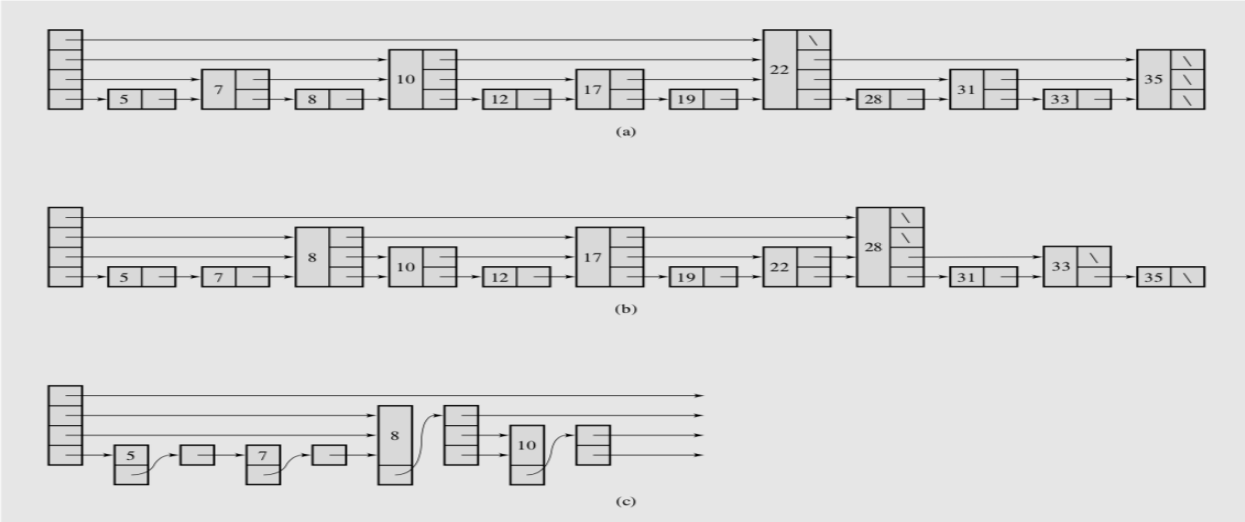
Searching for an element **el** consists of following the references on the highest level until an element is found that finishes the search successful. In the case of reaching the end of the list or encountering an element key that is greater than el, the search is restarted from the node preceding 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.

Skip List Algorithm:

```
find (element el)
        p = the nonnull list on the highest level i;
        while el not found and i >= 0
                if p.key < el
                        keep moving forward on the same level;
                else if p.key > el
                        if p is the last node on level i
                                move one level down and continue search;
                        else
                                p= p. next;
```

FIGURE 3.17 A skip list with (a) evenly and (b) unevenly spaced nodes of different levels; (c) the skip list with reference nodes clearly shown.



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.**