Linear ADTs

- Linear ADTs deal with linearly arranged data and information, where there is only one data item at the start and only one data item at the end. Except start and end data items, each data item is associated with only one data item before it and only one data item after it.

- The data item that comes before an other data item is called its predecessor.

- The data item that appears after another data item is called its successor.

- Except start and end items, each item has unique predecessor and unique successor.

- Start item has only unique successor and end item has only unique predecessor.

- Linear ADTs
  - LIST
  - STACK
  - QUEUE
  - PRIORITY QUEUE
  - RING
List ADT

List ADT arranges data linearly and provides random access i.e. you can access a data element anywhere and you can insert a data element anywhere.

**Specification of List ADT**

**Elements:** Any valid data type.

**Structure:** Elements are linearly arranged which allow random access i.e. there is a first element that has a unique successor and a last element that has a unique predecessor, every other element has a unique predecessor and a unique successor.

**Domain:** The number of elements is bounded.

Ahmed
Shoaib
Nabeel
Akram
Salman
Arsalan
## Specifiation of List ADT

### Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Specification</th>
</tr>
</thead>
</table>
| boolean `empty()` | **Precondition/Requires:** none  
**Processing/Results:** returns true if the list is empty otherwise false. |
| boolean `full()` | **Precondition/Requires:** none  
**Processing/Results:** returns true if no more elements can be inserted into the list otherwise false. |
| boolean `last()`  | **Precondition/Requires:** list is not empty  
**Processing/Results:** returns true if the current element is the last element in the list otherwise false. |
| void `findfirst()` | **Precondition/Requires:** list is not empty  
**Processing/Results:** the first element of the list is set as the current element. |
## Specification of List ADT

### Operations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Precondition/Requires:</th>
<th>Processing/Results:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void findnext()</code></td>
<td>list is not empty</td>
<td>the element following the current element is made the current element. If the current element is the last element, then the first element is made current.</td>
</tr>
<tr>
<td><code>Type retrieve()</code></td>
<td>list is not empty</td>
<td>returns the data of the current element.</td>
</tr>
<tr>
<td><code>void update(Type)</code></td>
<td>list is not empty</td>
<td>updates the data of the current element.</td>
</tr>
<tr>
<td><code>void insert(Type)</code></td>
<td>none</td>
<td>a new element containing given data is created and inserted after the current element in the list and current is made to point to the new element.</td>
</tr>
</tbody>
</table>
**Specification of List ADT**

**Operations:**

<table>
<thead>
<tr>
<th>void remove( )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition/Requires:</strong> list is not empty</td>
</tr>
<tr>
<td><strong>Processing/Results:</strong> the current element is removed from the list and if the resulting list is not empty and the successor of the deleted element exists, it is made the current element, otherwise the first element in the list is made the current element.</td>
</tr>
</tbody>
</table>
Representation of List ADT

List ADT can be represented as
- Array
- Linked List

Array

```
0
1
2
3
```

Linked List

```
Null
Linked List
```
public class ArrayList<T> {
    // Data Members
    private int maxSize;
    private int size;
    private int current;
    private T[] nodes;
    // Operations
    public ArrayList(int n)
    public boolean full()
    public boolean empty()
    public boolean last()
    public void findfirst()
    public void findnext()
    public T retrieve()
    public void update(T val)
    public void insert(T val)
    public void remove()
}

maxsize: The total number of elements which can be stored
size: The number of elements which have been stored
public ArrayList(int n) {
    maxsize = n;
    size = 0;
    current = -1;
    nodes = (T[]) new Object[n];
}

public boolean full() {
    return size == maxsize;
}

public boolean empty() {
    return size == 0;
}
Implementation of Operations

```java
public boolean last()
{
    return current == size;
}
```

```java
public void findfirst()
{
    if(empty()) return;
    current = 0;
}
```

```java
public void findnext()
{
    if(empty()) return;
    current++;
}
```
Implementation of Operations

```java
public T retrieve()
{
    if(empty())
        return nodes[current];
}

public void update(T val)
{
    if(empty()) return;
    nodes[current] = val;
}
```
Implementation of Operations

```java
public void insert(T val) {
    for (int i = size-1; i > current; --i) {
        nodes[i+1] = nodes[i];
    }
    current++;
    nodes[current] = val;
    size++;
}
```
public void remove()
{
    for (int i = current + 1; i < size; i++)
    {
        nodes[i-1] = nodes[i];
    }
    size--;
    if (size == 0) current = -1;
    else if (current == size) current = 0;
}
Performance Analysis

- It does not permit efficient insertion and removal of elements.
- It stores elements in consecutive memory locations. Because of this, it has to move its elements to remove or insert an element.
- Time complexity of insert and remove is $O(n)$ in the worst case.
- Time complexity of all other operations is $O(1)$. 
Linked List

It is a list storage structure that allows elements to be scattered throughout memory. The link between elements in the list is established by assigning each element the reference to its adjacent element. There are three popular structures of Linked list:

- Singly linked list
- Doubly linked list
- Circular Linked List

Singly Linked List

- In singly linked list, each element has reference only to next adjacent element.
- A reference to the first element of the list, called head, is used to keep track of the list.
Doubly Linked List

- In doubly linked list, each element has a reference to its successor as well as a reference to its predecessor.

- A pointer to the first element of the list, called **head**, is used to keep track of the list.

Circular Linked List

- It may be singly linked list or doubly linked list but the last node has reference to first node.
Linked List

- Singly linked list allows only forward traversal whereas doubly linked list allows both forward and backward traversal.
- Implementation of some operations may be more efficient in doubly linked list as compared to singly linked list but doubly linked list consumes more memory space because each node contains two references.
- We will implement List ADT using singly linked list.
Linked List Implementation of List ADT

```java
public class Node<T> {
    public T data;
    public Node<T> next;
    public Node() {
        data = null;
        next = null;
    }
    public Node(T val) {
        data = val;
        next = null;
    }
}

public class LinkList<T> {
    // Data Members
    private Node<T> head;
    private Node<T> current;
    // Operations
    public LinkList() {
        head = null;
        current = null;
    }
    public boolean empty() {
        return head == null;
    }
    public boolean last() {
        return current == null;
    }
    public boolean full() {
        return false;
    }
    public void findfirst() {
        current = head;
    }
    public void findnext() {
        current = current.next;
    }
    public T retrieve() {
        return current.data;
    }
    public void update(T val) {
        current.data = val;
    }
    public void insert(T val) {
        Node<T> newNode = new Node<T>(val);
        newNode.next = head;
        head = newNode;
    }
    public void remove() {
        if (head != null) {
            head = head.next;
        }
    }
}
```
Implementation of Operations

```java
public LinkList()
{
    head = current = null;
}
```

```java
public boolean full()
{
    return false;
}
```

```java
public boolean empty()
{
    return head == null;
}
```
Implementation of Operations

```java
public boolean last()
{
    if (empty()) return;
    return current.next == null;
}

public void findfirst()
{
    if (empty()) return;
    current = head;
}

public void findnext()
{
    if (empty()) return;
    current = current.next;
}
```
Implementation of Operations

public T retrieve()
{
    if(!empty())
        return current.data;
}

public void update(T val)
{
    if(empty()) return;
    current.data = val;
}
Implementation of Operations

```java
public void insert(T val) {
    Node<T> tmp;
    if (empty()) {
        current = head = new Node<T> (val);
    } else {
        tmp = current.next;
        current.next = new Node<T> (val);
        current = current.next;
        current.next = tmp;
    }
}
```
Implementation of Operations

```java
public void remove()
{
    if (current == head){
        head = head.next;
    }
    else {
        Node<T> tmp = head;
        while (tmp.next != current) tmp = tmp.next;
        tmp.next = current.next;
    }
    if (current.next == null) {
        current = head;
    }
    else {
        current = current.next;
    }
}
```
Performance Analysis

- It allows efficient insertion but removal is still not efficient. Removal can be made efficient if doubly linked list is used.
- Depending on the application, it can consume more memory as compared to array because each node also stores reference to next node.
- Time complexity of remove is $O(n)$ in the worst case.
- Time complexity of all other operations is $O(1)$. 