Cursor Implementation of Linked Lists

- **Reason 1**: Many Languages do not support pointers (e.g. Basic, Fortran).
  - If linked lists are required and pointers are not available, then an alternate implementation must be used.
  - The alternate method we will describe here is known as a **cursor implementation**.
- **Reason 2**: If data max length is **known**, using Array is **faster**.

Two features present in a pointer implementation of linked lists:

1. The data are stored in **array** are nodes, each array element (node) contains **data** and a **pointer** to the next node.
2. A new node can be obtained from the system's global memory by a call to ***malloc*** (**m**emory **alloc**ation) and released by a call to ***free*** methods.

Our cursor implementation must be able to simulate these two features:

- The logical way to satisfy 1st feature is to have a global array of nodes. For any cell in the array, its array index can be used in place of an address. The following gives the type declarations for a cursor implementation of linked lists:

```
public class Node<T extends Comparable<T>> {
   T data;
   int next;

   public Node(T data, int next) {
      this.data = data;
      this.next = next;
   }

   public void setData(T data) { this.data = data;  }
   public T getData() {  return data;   }
   public int getNext() {  return next;  }
   public void setNext(int next) { this.next = next;  }

   public String toString() { return "["+ data+ " , " + next + "]"; }
}
```

- We must now simulate 2nd feature by allowing the equivalent of ***malloc*** and ***free*** for nodes in the array.
  - To do this, we will keep a list (the **freelist**) of nodes that are not in any list. The list will use **node 0** as a header. The initial configuration is shown in the following figure: →→→→
- A value of **next** is the equivalent of a pointer to next node.
- The following code to create an array of free nodes:

```
Node<T>[ ]  cursorArray  =  new Node[11];
```

i	data	next
0	null	1
1	null	2
2	null	3
3	null	4
4	null	5
5	null	6
6	null	7
7	null	8
8	null	9
9	null	10
10	null	0

- The initialization of **cursorArray** is a straightforward loop:

```
public int initialization(){
   for(int i=0;i<cursorArray.length-1;i++)
      cursorArray[i] = new Node<>(null, i+1);
   cursorArray[cursorArray.length-1] = new Node<>(null, 0);
   return 0;
}
```

- To perform an ***malloc***, the first element (after the header) is removed from the **freelist**:

```
public int malloc() {
   int p = cursorArray[0].next;
   cursorArray[0].next = cursorArray[p].next;
   return p;
}
```

- To perform a ***free***, we place the cell at the front of the **freelist**:

```
public void free(int p){
   cursorArray[p] = new Node(null, cursorArray[0].next);
   cursorArray[0].next = p;
}
```

- The following are a list of functions to test whether a linked list is **null**, **empty**, or whether a specific node is the **last**:

```
public boolean isNull(int l){
   return cursorArray[l]==null;
}

public boolean isEmpty(int l){
   return cursorArray[l].next == 0;
}

public boolean isLast(int p){
   return cursorArray[p].next == 0;
}
```

- To create a new linked list, first you have to allocate one free node using **malloc** function, then make a new point that next points to **0** as follow:

```
public int createList(){
   int l = malloc();
   if(l==0)
      System.out.println("Error: Out of space!!!");
   else
      cursorArray[l] = new Node("-",0);
   return l;
}
```

- The following code is used to add a new data to a specific linked list:

```
public void insertAtHead(T data, int l){
   if(isNull(l)) // list not created
      return;
   int p = malloc();
   if(p!=0){
      cursorArray[p] = new Node(data, cursorArray[l].next );
      cursorArray[l].next = p;
   }
   else
      System.out.println("Error: Out of space!!!");
}
```

- The following code is used to travers a linked list:

```
public void traversList(int l) {
   System.out.print("list_"+l+"-->");
   while(!isNull(l) && !isEmpty(l)){
      l=cursorArray[l].next;
      System.out.print(cursorArray[l]+"-->");
   }
   System.out.println("null");
}
```

- The following code is used to find a specific data in a linked list:

```
public int find(T data, int l){
   while(!isNull(l) && !isEmpty(l)){
      l=cursorArray[l].next;
      if(cursorArray[l].data.equals(data))
         return l;
   }
   return -1;  // not found
}
```

- Sometimes you need the previous location of a specific data in a linked list:

```
public int findPrevious(T data, int l){
   while(!isNull(l) && !isEmpty(l)){
      if(cursorArray[cursorArray[l].next].data.equals(data))
         return l;
      l=cursorArray[l].next;
   }
   return -1;  // not found
}
```

- The following code is used to delete some data from a linked list:

```
public Node delete(T data, int l){
   int p = findPrevious(data, l);
   if(p!=-1){
      int c = cursorArray[p].next;
      Node temp = cursorArray[c];
      cursorArray[p].next = temp.next;
      free(c);
   }
   return null;
}
```