**Data Structures - Lecture Notes**

**Linked Lists**

**A. Definition:**

The linked list consists of a series of nodes, which are not necessarily adjacent in memory. Each node contains the element and a link to the next node. We need to maintain two pointers for the whole list, the first is the head which points to the first element in the list and the second is the tail which points to the last element.

Since elements are not stored in consecutive spaces in the memory we can't access an element directly. The following is a detailed explanation of the implementation of the Linked List:

**B. Implementation:**

**1. Class Declaration:**

We first need a class for each node containing its value and a pointed to the next node:

|  |
| --- |
| template <class V>  class Node  {    public:  V value;  Node<V> \*next;  Node(void);  Node(V val);  ~Node(void);  }; |

For the Linked List class you need only to maintain the head and tail pointers as well as a counter to the number of elements.

|  |
| --- |
| template <class T>  class LinkedList  {  private:  Node<T>\* head;  Node<T>\* tail;  int size;  public:  int Length();  T At(int pos);  void InsertAt(int pos, T val);  T DeleteAt(int pos);  void Append(T val);  LinkedList(void);  }; |

**2. Linked List Functions:**

***Append***

Append adds an element to the end of the list. Since the element is added to the end you will need first to make the last node point to the new node. You will next need to change the tail of the list to also point at the new node.

Special case: if the list is empty, you will need to set the head also to point at the new node.

|  |
| --- |
| template <class T>  void LinkedList<T>::Append(T val)  {  Node<T>\* tmp = new Node<T>(val);  //in case the list is empty  if(head==NULL)  head=tail=tmp;  else //if the list is not empty  {  tail->next=tmp;  tail=tmp;  }  size++;  } |

***At:***

To display the element at a certain position you need to start at the head and move through the list element by element until you reach the required position.

|  |
| --- |
| template <class T>  T LinkedList<T>::At(int pos)  {  //validate input: check that the position exists  assert (pos<size);  Node<T>\* tmp=head;  int cur=0;  //find the node at the required position  while(cur<pos)  {  tmp=tmp->next;  cur++;  }  return tmp->value;  } |

***InsertAt:***

The insertAt functions takes a position and a value and adds a node with the given value to the list in the same position passed to the function. You have to search the list starting with the head until you reach the node before the required position.

|  |
| --- |
| template <class T>  void LinkedList<T>::InsertAt(int pos, T val)  {  //validate input: check that the position exists  assert (pos<size);  Node<T>\* tmp=head;  int cur=0;  Node<T>\* newNode=new Node<T>(val);    //if you want to insert at the first position //(you need to change the head)  if(pos==0)  {  newNode->next=head;  head=newNode;  }  else //for all other cases  {  //find the node before the position you //want to insert at  while(cur<pos-1)  {  tmp=tmp->next;  cur++;  }  newNode->next=tmp->next;  tmp->next=newNode;  }  size++;  } |

It uses the following steps (the graphs assume you want to insert at position 2):

We will break the code part by part

1. You first need to create the new node:

|  |
| --- |
| Node<T>\* newNode=new Node<T>(val); |

2. Locate the node before the required position

|  |
| --- |
| while(cur<pos-1)  {  tmp=tmp->next;  cur++;  } |

3. Make the new node point at the node currently at position 2

|  |
| --- |
| newNode->next=tmp->next; |

4. make the previous node point at the new node instead of (A2)

|  |
| --- |
| tmp->next=newNode; |

Special Case: If you insert at position 0 (which is the start of the list), you will need to change the head to point at the new node.

***DeleteAt:***

DeleteAt takes a position and removes the node at the specified position.

|  |
| --- |
| template <class T>  void LinkedList<T>::DeleteAt(int pos)  {  //validate input: check that the position exists  assert (pos<size);  T val;  Node<T>\* tmp=head;  int cur=0;  //deleting the first node  if(pos==0)  {  head=head->next;  delete tmp;  }  else //for all other cases  {  while(cur<pos-1)  {  tmp=tmp->next;  cur++;  }  Node<T>\* deletedNode=tmp->next;  if(deletedNode->next==NULL) //if you are deleting the last node  tail=tmp;  else  tmp->next=deletedNode->next;  delete deletedNode;  }  size--;  } |

It uses the following steps (the graphs assume you want to remove the node at position 2):

We will break the code part by part

1. You first need to locate the node before the required position:

|  |
| --- |
| while(cur<pos-1)  {  tmp=tmp->next;  cur++;  } |

2. Make the previous node point to the node after the node you want to delete

|  |
| --- |
| tmp->next=deletedNode->next; |

3. remove the node

Special Cases:

- if you are deleting the first node: you need to change the head to point to the next node.

|  |
| --- |
| if(pos==0)  {  head=head->next;  delete tmp;  } |

- If you are deleting the last node: you need to change the tail to pint at the previous node.

|  |
| --- |
| if(deletedNode->next==NULL)  //if you are deleting the last node  tail=tmp; |

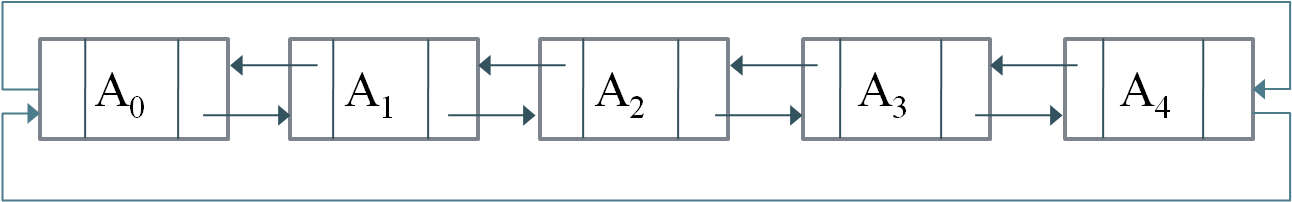
**C. Variations to Linked Lists:**

***1. Doubly Linked List:***

each node contains two pointers, one to the next node and another to the previous node. The previous pointer for the head node is NULL and the next pointer for the tail node is NULL.

***2. Circular Linked List:***

Same as doubly Linked List except that the previous pointer of the head nodes points at the tail and the next pointer of the tail node points at the head.



**D. Array Lists vs. Linked Lists:**

Array Lists and Linked lists are two different implementations for the list ADT. Each of these implementations has its advantages and disadvantages. We will here discuss a few situations and determine which of the two implementations is better in each situation:

***1. The size of data is unknown:***

In array lists you need to declare an array with initial size. This array can be resized later but this is an expensive process so it is better to know the size beforehand. But with linked lists the list can be extended at any time and you do not use any extra space than the one needed. Therefore, linked lists is better in the situation where the size of the data is not known.

***2. Random Access:***

Random access means the need to access any element at any position. In an array list you can access any item at any position directly (O(1)). But in a linked list you do not have access except to the head of the list, therefore you need to start at the head and move element by element until you reach the desired position (O(n)). So in situations where you need random access a lot of the time, array lists are better.

***3. Memory requirements:***

Array lists only occupy the space taken by the values stored. But linked lists there is the extra space needed by pointers so it takes more space for the same number of items. Therefore, if memory is a concern array lists are a better option.