

Lecture

NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Linked List

Linked List is an Abstract Data Type (ADT) that holds a collection of **Nodes**, the nodes can be accessed in a sequential way. Linked List doesn't provide a random access to a **Node**.

Usually, those **Nodes** are connected to the next node and/or with the previous one, this gives the **linked** effect. When the Nodes are connected with only the **next** pointer the list is called Singly Linked List and when it's connected by the **next** and **previous** the list is called Doubly Linked List.

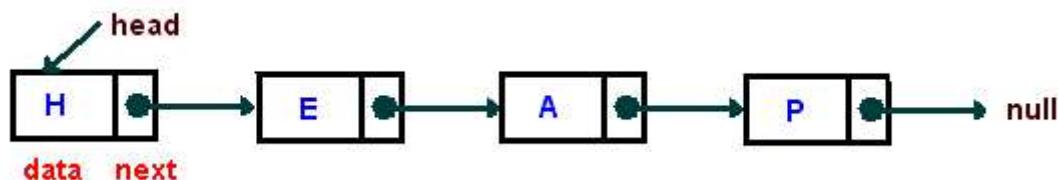


Figure 1: Singly Linked-List

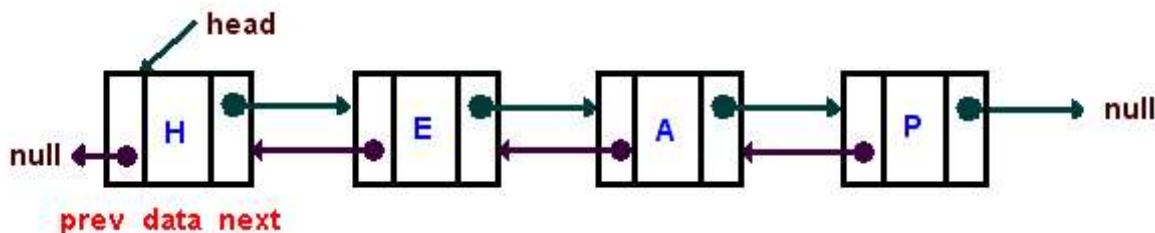


Figure 2: Doubly Linked-List

The **Nodes** stored in a Linked List can be anything from **primitive types** such as integers to more complex types like **instances of classes**.

One **disadvantage** of a linked list against an array is that it does not allow direct access (random access) to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another **disadvantage** is that a linked list uses more memory compare with an array. We have to assign extra memory to store a reference to the next node.

Structure of Node(s)

Each element in the list is called a **node** and each node comprised of two items:

- The **data** and
- A **reference** to the **next node** or both **next and previous**

The entry point into a linked list is called the **head** of the list. It should be noted that **head** is not a separate node, but the reference to the first node. If the list is empty then the head is a **null** reference.



A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

ADT — Interface

The Linked List interface can be implemented in different ways, is important to have operations to insert a new node and to remove a Node:

Operations	Our Operations	Description
<code>prepend(value)</code>	<code>insertAtStart(value)</code>	Add a node in the beginning
<code>append(value)</code>	<code>insert(value)</code>	Add a node in the end
<code>pop()</code>	<code>delete()</code>	Remove a node from the end
<code>popFirst()</code>	<code>deleteAtStart()</code>	Remove a node from the beginning
<code>head()</code>	<code>head</code>	Return the first node
<code>tail()</code>	-	Return the last node
<code>remove(Node)</code>	<code>DeleteAfter(value)</code>	Remove Node from the list

Since abstract data types don't specify an implementation, this means it's also incorrect to talk about the time complexity of a given abstract data type.

Insertion and Deletion Operations

As we already discussed the insertion and deletion operation of linked list in previous lecture one thing that we need to know is those operations had the complexity cost of **$O(n)$** as we have to first go to the last node by traversing the entire Linked list and then perform an insertion and deletion. Today we will try to make these operations with the computational cost of **$O(1)$** .

Traversal and Search Operations

Traversal is the process of accessing all the element of list in a sequential manner. We discussed this traversal in the **`print`** method where we print all the elements of the Linked list.

Search is another method where we find out whether the specified item exists in the Linked list or not. We also discussed this when we insert a value ***after specific node***. First we **`search`** the specific node with specific value and then we insert a new node.

Code for Singly Linked List

In Today's Lecture we discuss the implementation of new Singly Linked List with insertion and deletion cost of **$O(1)$** . We will implement the Linked List as per the operations of ADT-Interface discussed above.



```
public class SinglyLinkedList
{
    class Node
    {
        int data;
        Node next;
    }

    Node head = null;
    Node tail = null;
    Node tail_1;

    public void insertAtStart(int value)
    {
        Node n = new Node();
        n.data = value;

        if(head == null)
        {
            n.next = null;
            head = n;
            tail = n;
        }
        else
        {
            if(head == tail)
            {
                tail_1 = n;
            }
            n.next = head;
            head = n;
        }
    }

    public void insert(int value)
    {
        Node n = new Node();
        n.data = value;
        n.next = null;

        if(head==null)
        {
            head = n;
            tail = n;
        }
        else
        {
            tail_1 = tail;
            tail.next = n;
            tail = n;
        }
    }
}
```



```
public int delete()
{
    if(head == null)
    {
        System.out.println("List is Empty");
        return -9999;
    }
    else
    {
        int value = tail.data;
        tail = tail_1;
        tail.next = null;
        tail_1 = findTail_1();
        return value;
    }
}

public Node findTail_1()
{
    if (head == tail)
        return null;
    else
    {
        Node t = head;
        while(t.next != tail)
            t = t.next;
        return t;
    }
}

public int deleteAtStart()
{
    if(head == null)
    {
        System.out.println("List is Empty");
        return -9999;
    }
    else
    {
        if(head == tail)
        {
            tail = null;
            tail_1 = null;
        }
        if(head.next == tail)
        {
            tail_1 = null;
        }

        int value = head.data;
        head = head.next;
    }
}
```



```
        return value;
    }
}

public void print()
{
    Node t = head;
    System.out.print("LIST: ");
    while(t != null)
    {
        System.out.print(t.data+ " ");
        t = t.next;
    }
    System.out.println("\n");
}
}
```

Code above is the code for SinglyLinkedList class. However, to test the SinglyLinkedList class we will use the following code in the main class.

```
public class SLL2Demo
{
    public static void main(String[] args)
    {
        SinglyLinkedList sll = new SinglyLinkedList();
        sll.delete();
        sll.insert(10);
        sll.print();
        sll.insertAtStart(20);
        sll.print();
        sll.insert(30);
        sll.print();
        System.out.println("Deleted = " + sll.delete());
        sll.print();
        System.out.println("Deleted = " + sll.deleteAtStart());
        sll.print();
        sll.insertAtStart(40);
        sll.print();
        System.out.println("Deleted = " + sll.deleteAtStart());
        sll.print();
        System.out.println("Deleted = " + sll.deleteAtStart());
        sll.print();
        sll.delete();

    }
}
```

Output:

```
List is Empty
LIST: 10
```



LIST: 20 10

LIST: 20 10 30

Deleted = 30

LIST: 20 10

Deleted = 20

LIST: 10

LIST: 40 10

Deleted = 40

LIST: 10

Deleted = 10

LIST:

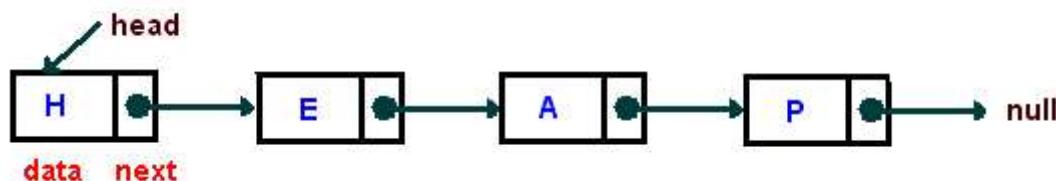
List is Empty

Lecture

NOTE: FOR FURTHER DETAILS AND MORE COMPREHENSIVE STUDY, PLEASE SEE RECOMMENDED BOOKS OR INTERNET.

Linked List

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this Lecture we consider another data structure called Linked Lists that addresses some of the limitations of arrays. A linked list is a linear data structure where each element is a separate object.



Each element (**node**) of a list comprised of two items:

- The data and
- A reference to the next node

The last node has a reference to **null**.

The entry point into a linked list is called the **head** of the list. It should be noted that **head** is not a separate node, but the reference to the first node. If the list is empty then the head is a **null** reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

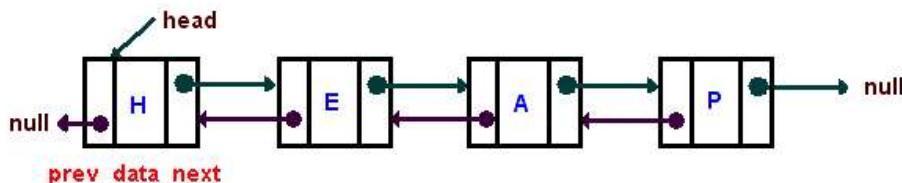
One **disadvantage** of a linked list against an array is that it does not allow direct access (random access) to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another **disadvantage** is that a linked list uses more memory compare with an array. We have to assign extra memory to store a reference to the next node.

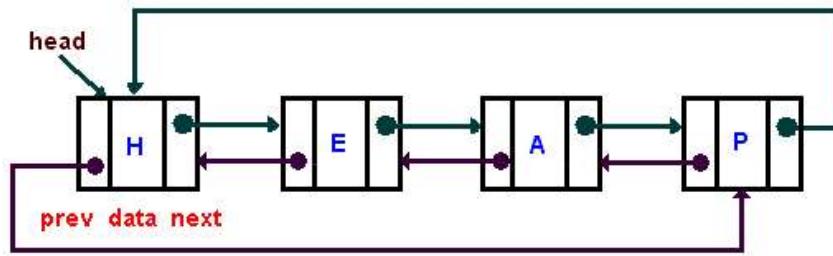
Types of Linked Lists

A **singly linked list** is described above

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.



The Node class

In Java you are allowed to define a class (say **B**) inside of another class (say **A**). The class **A** is called the outer class, and the class **B** is called the inner class. The purpose of inner classes is purely to be used internally as helper classes. In Linked List we will use this inner class to define the structure of a node and the class will be called **Node**.

An inner class is a member of its enclosing (outer) class and has access to other members (including private) of the outer class, and vice versa. The outer class can have a direct access to all members of the inner class. An inner class can be declared private, public, protected, or package private. There are two kind of inner classes: **static** and **non-static**. A static inner class cannot refer directly to instance variables or methods defined in its outer class: it can use them only through an object reference. We can use both type of inner classes for our purpose of creating a node.

Representing a Linked List Using Arrays

How can you represent a linked list in array? (Coming Assignment)

Code for a Singly Linked List

```
public class SinglyLinkedList
{
    class Node
    {
        char data;
        Node next;
    }

    Node head = null;

    public void insert(char value)
    {
        Node n = new Node();
        n.data = value;
```



```
n.next = null;

if(head==null)
    head = n;
else
{
    Node _t = head;
    while(_t.next != null)
        _t = _t.next;

    _t.next = n;
}

public char delete()
{
    if(head == null)
    {
        System.out.println("List is Empty");
        return '\0'; //null Character
    }
    else
    {
        Node _t = head;
        Node _t1 = null;
        while(_t.next != null)
        {
            _t1 = _t;
            _t = _t.next;
        }

        if(_t == head)
        {
            head = null;
            return _t.data;
        }
        else
        {
            char value = _t.data;
            _t1.next = null;
            return value;
        }
    }
}

public void print()
{
    Node t = head;
    System.out.print("LIST: ");
    while(t.next != null)
    {
        System.out.print(t.data+" ");
    }
}
```



```
        t = t.next;
    }
    System.out.println(t.data + "\n");
}

public void insertAtStart(char value)
{
    Node n = new Node();
    n.data = value;
    if(head == null)
    {
        n.next = null;
        head = n;
    }
    else
    {
        n.next = head;
        head = n;
    }
}

public char deleteAtStart()
{
    if(head == null)
    {
        System.out.println("List is Empty");
        return '\0'; //null Character
    }
    else
    {
        char value = head.data;
        head = head.next;
        return value;
    }
}

public void insertAfter(char ch, char value)
{
    Node t = head;
    while(t != null)
    {
        if(t.data == ch)
        {
            Node n = new Node();
            n.data = value;
            n.next = t.next;
            t.next = n;
            return;
        }
        t = t.next;
    }
    System.out.println(ch + " not Found");
}
```



```
}

public char deleteAfter(char ch)
{
    Node t = head;
    while(t != null)
    {
        if(t.data == ch)
        {
            if(t.next != null)
            {
                Node n = t.next; //Node to be deleted
                t.next = n.next;
                return n.data;
            }
            else
            {
                System.out.println("No Node After "+ch);
                return '\0'; //null Character
            }
        }
        t = t.next;
    }
    System.out.println(ch + " not Found");
    return '\0'; //null Character
}
}
```

Code above is the code for SinglyLinkedList class. However, to test the SinglyLinkedList class we will use the following code in the main class.

```
public class SinglyLinkListDemo
{

    public static void main(String[] args)
    {
        SinglyLinkedList sll = new SinglyLinkedList();
        sll.delete();
        sll.insert('A');
        sll.print();
        sll.insert('B');
        sll.print();
        sll.insert('C');
        sll.print();
        System.out.println("Deleted = " + sll.delete());
        sll.print();
        System.out.println("Deleted = " + sll.delete());
        sll.print();
        sll.insertAtStart('D');
        sll.print();
        System.out.println("Deleted = " + sll.deleteAtStart());
        sll.print();
    }
}
```



```
sll.insertAfter('A', 'E');
sll.print();
sll.insertAfter('A', 'F');
sll.print();
sll.insertAfter('C', 'G');
sll.insertAfter('F', 'G');
sll.print();
sll.deleteAfter('F');
sll.print();
sll.deleteAfter('E');

}

}
```

Output:

List is Empty

LIST: A

LIST: A B

LIST: A B C

Deleted = C

LIST: A B

Deleted = B

LIST: A

LIST: D A

Deleted = D

LIST: A

LIST: A E

LIST: A F E

C not Found

LIST: A F G E

LIST: A F E

No Node After E

LIST: A F E