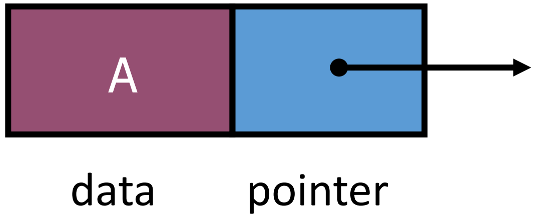
Programming, Data Structures & Algorithms LinkedLists

By

Samira Dayan Jayasekara

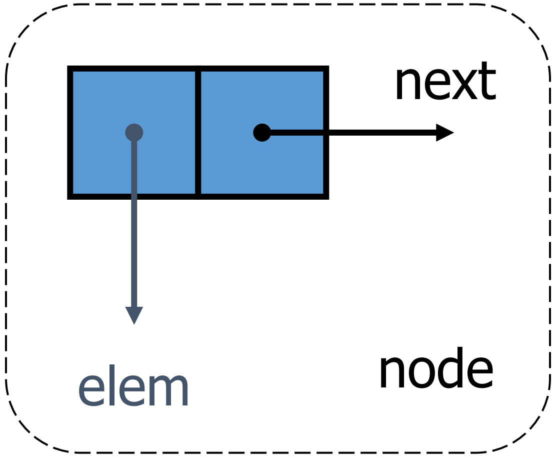
Linked List Types

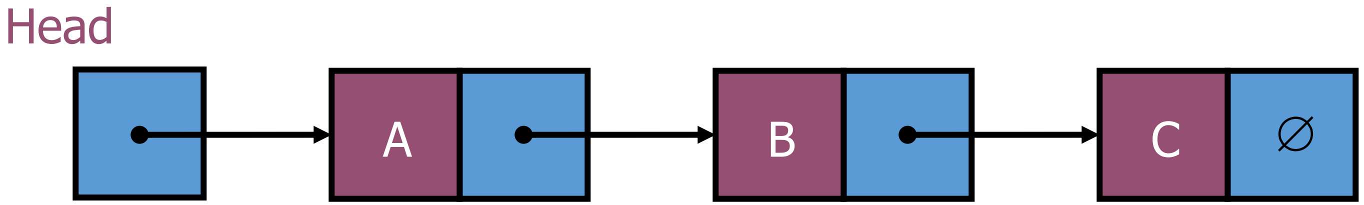
- Singly linked list
- Doubly linked list
- Circularly linked list



Singly Linked List

- A singly linked list is a dynamic data structure consisting of a sequence of ***nodes***, forming a ***linear*** ordering.
- Each node stores
  - element(piece of data)
  - link to the next node
- *Head*: pointer to the first node of the link list



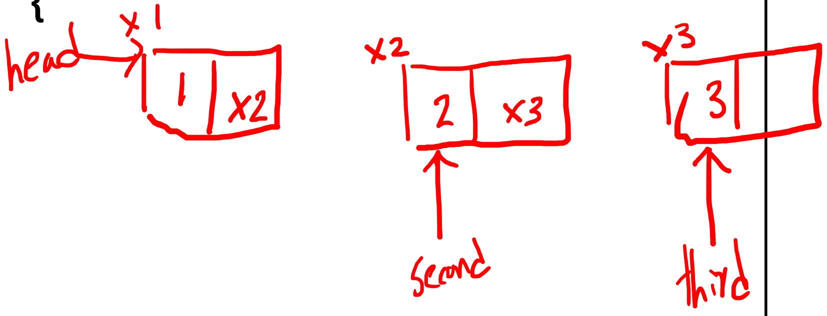


1D-arrays vs. Singly-linked lists

ID-array	**Singly-linked list**
Fixed size: Resizing is expensive (Creation of a new Array with new Size)	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted (Creation of a new Array with new Size)	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Extra storage needed for references; however uses exactly as much memory as it needs
Sequential access is faster because of greater locality of references [Reason: Elements in contiguous memory locations]	Sequential access is slow because of low locality of references [Reason: Elements not in contiguous memory locations]

JAVA Singly Linked List **EXAMPLE 1 - Starting from Head**

```
// A simple Java program to introduce a
linked list
public class MyLinkedList
{
   static Node head;  // head of list

   /* Linked list Node.  This inner class is
made static so that
      main() can access it */
   static class Node {
      int data;
      Node next;
      Node(int d)  { data = d;  next=null; }
 // Constructor
   }

   /* method to create a simple linked list
with 3 nodes*/
   public static void main(String[] args)
   {
```



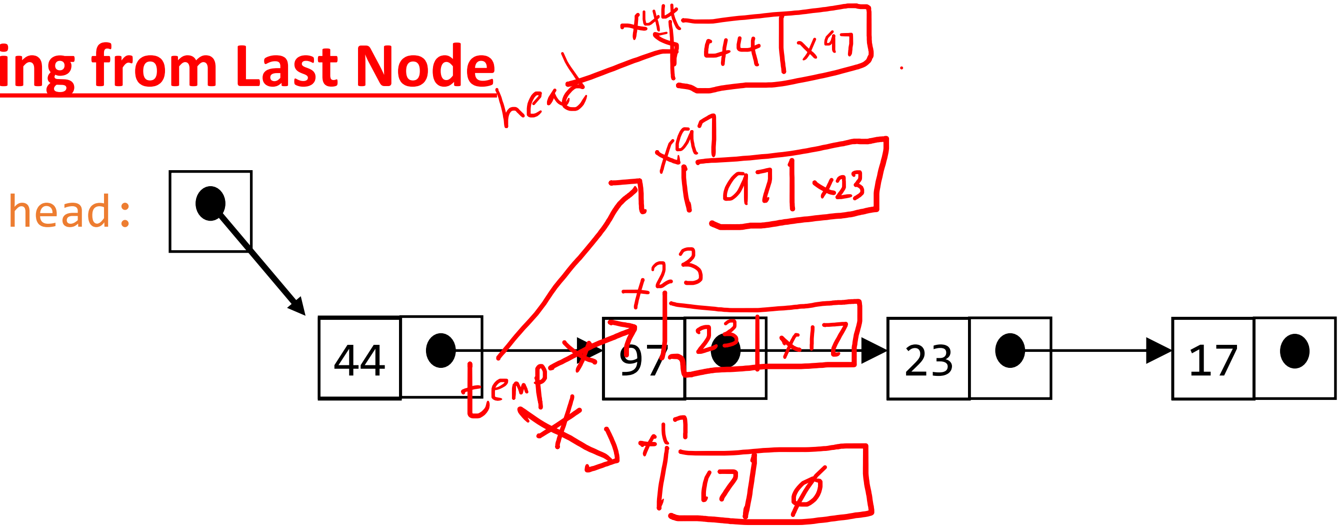
```
      head  = new Node(1);
      Node second = new Node(2);
      Node third  = new Node(3);

      /* Three nodes have been allocated
dynamically.We have refernces to these three
blocks as first,  second and third

           head      second         third
             |          |             |
             |          |             |
         +----+------+  +----+------+  +----+------+
         | 1  | null |  | 2  | null |  |  3 | null |
         +----+------+  +----+------+  +----+------+ */

head.next = second;
// Link first node with the second node

      /*  Now next of first Node refers to second.
So they
         both are linked.

           head       second        third
             |          |             |
             |          |             |
         +----+------+  +----+------+  +----+------+
         | 1  |  o-------->| 2  | null |  |  3 | null |
         +----+------+  +----+------+  +----+------+ */
```

```
second.next = third;
 // Link second node with the third node
      /*  Now next of second Node refers to third.  So
all three
          nodes are linked.

           head      second          third
             |          |              |
             |          |              |
         +----+------+  +----+------+  +----+------+
         | 1  |  o-------->| 2  |  o-------->|  3 | null |
         +----+------+  +----+------+  +----+------+ */
   Display(head);
   }
static void  Display(Node currNode)
   {
     while(currNode != null )
     {
     System.out.println("Value:"+currNode.data );
      currNode=currNode.next;
     }
   }
}
```

```
Output:
Value:1
Value:2
Value:3
```

JAVA Singly Linked List **EXAMPLE 2 - Starting from Last Node**

```
// A simple Java program to introduce a linked list
public class MyLinkedList2
{
   static Node head;  // head of list

   /* Linked list Node.  This inner class is made static so that
     main() can access it */
   static class Node {
      int data;
      Node next;
      Node(int d, Node t)  { data = d;  next=t; }
 // Constructor
   }

   /* method to create a simple linked list with 3 nodes*/
   public static void main(String[] args)
   {

        Node temp = new Node(17, null);

        temp = new Node(23, temp);

        temp = new Node(97, temp);

        head= new Node(44, temp);

        Display(head);

    }
```



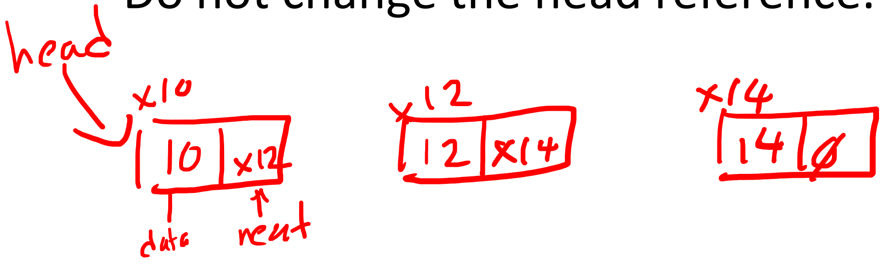
```
static void  Display(Node currNode)
   {
     while(currNode != null )
      {
        System.out.println("Value:" +currNode.data );
        currNode=currNode.next;
      }
   }
}
```

Online Java Complie & Run : https://www.compilejava.net/

JAVA Singly Linked List EXAMPLE -transverse

transverse

- Start with the head and access each node until you reach null.
- Do not change the head reference.

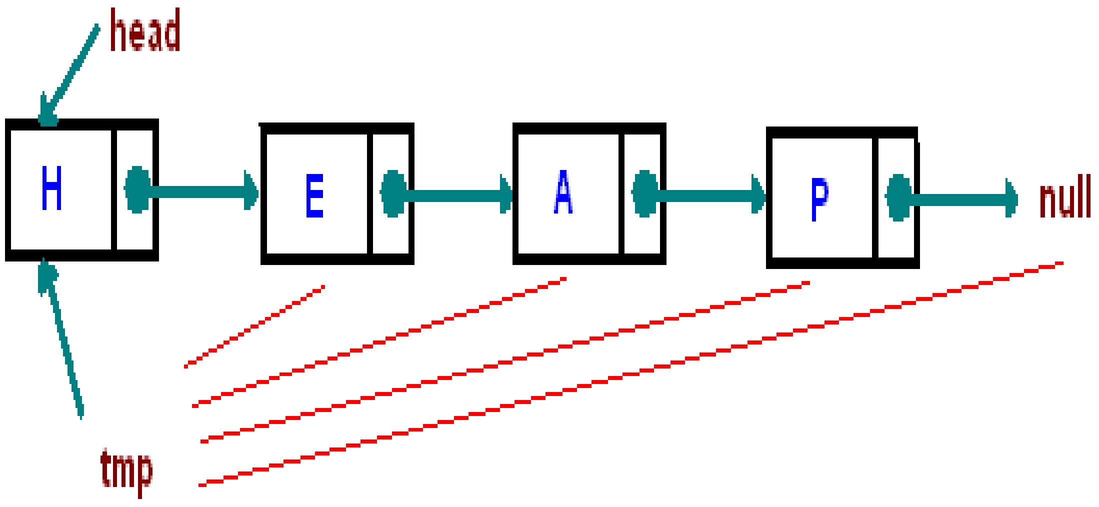


currNode	c1
x10	T
x12	T
x14	T
∅	F

Data : 10

Data : 12

Data : 14

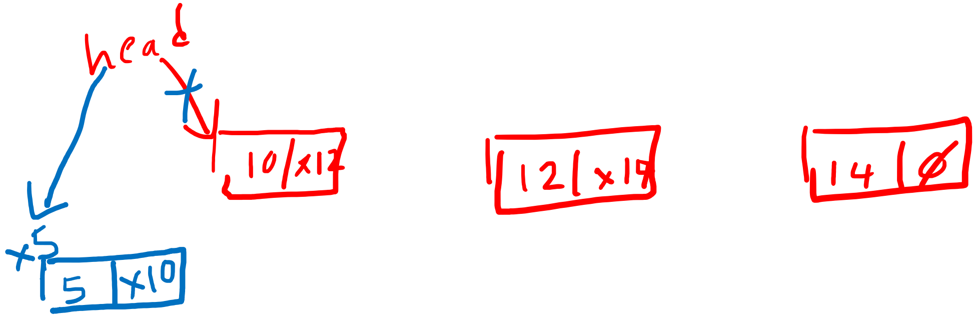


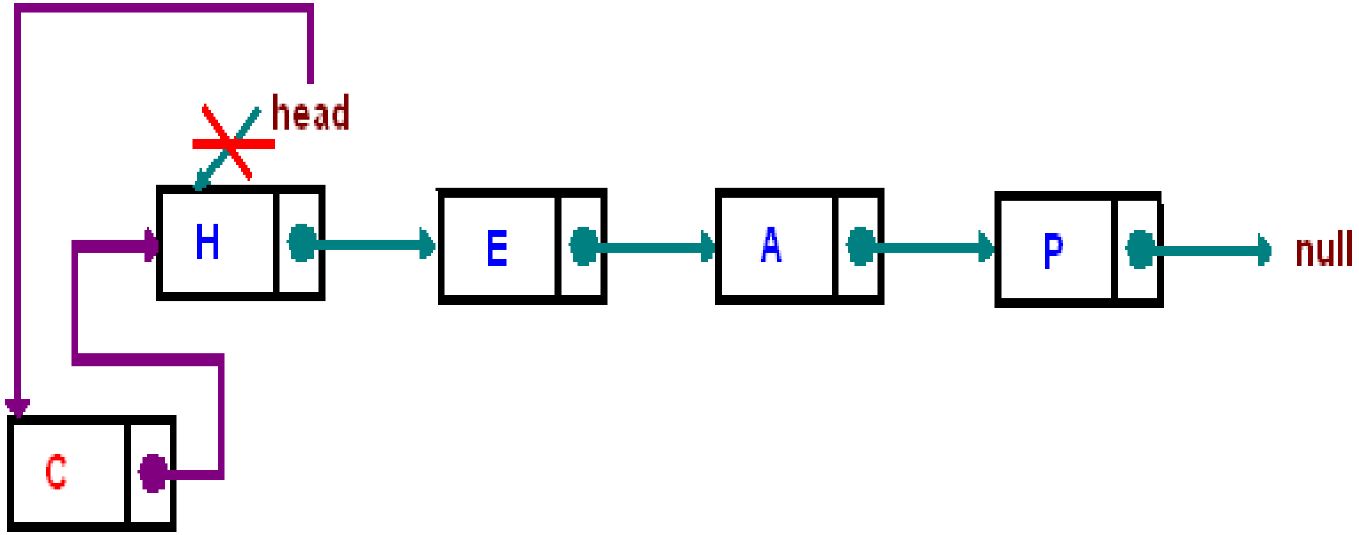
```
static void  Display(Node currNode)
   {
                         c1
     while(currNode != null )
      {
        System.out.println("Data:"+currNode.data );
        currNode=currNode.next;
      }
   }
```

JAVA Singly Linked List **EXAMPLE -addFirst**

addFirst

- The method creates a node and prepends it at the beginning of the list.





1)Create a New Node with
Value=input value
Reference= Current Head Ref
2)Head = new Node Reference

```
public static void addFirst(int item)
  {                      ①
    head = new Node(item, head);
      ②
  }
```

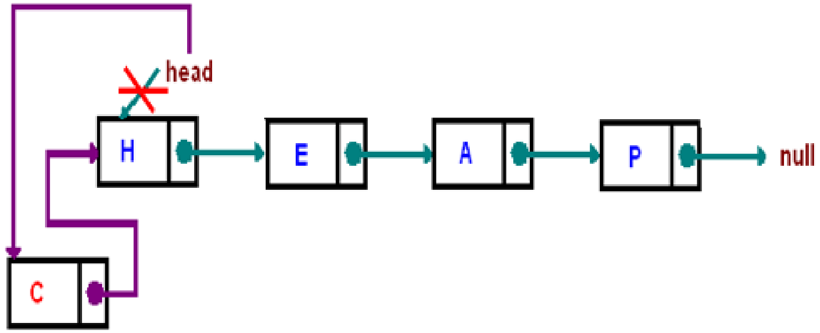
```
// A simple Java program to introduce a linked
list
public class MyLinkedList
{
   static Node head;  // head of list

  static class Node {
       int data;
       Node next;
       Node(int d)  { data = d;  next=null; }
       Node(int d, Node t)  { data = d;  next=t; }

 // Constructor
   }
  public static void main(String[] args)
   {
      head  = new Node(1);
       Node second = new Node(2);
       Node third  = new Node(3);
       head.next = second;
       second.next = third;
       Display(head);
       addFirst(23);
      Display(head);

   }
```

```
   static void  Display(Node currNode)
   {
     while(currNode != null )
      {
      System.out.println("Data:"+currNode.data );
        currNode=currNode.next;
      }
   }

   public static void addFirst(int item)
   {

     head = new Node(item, head);


   }

}
```

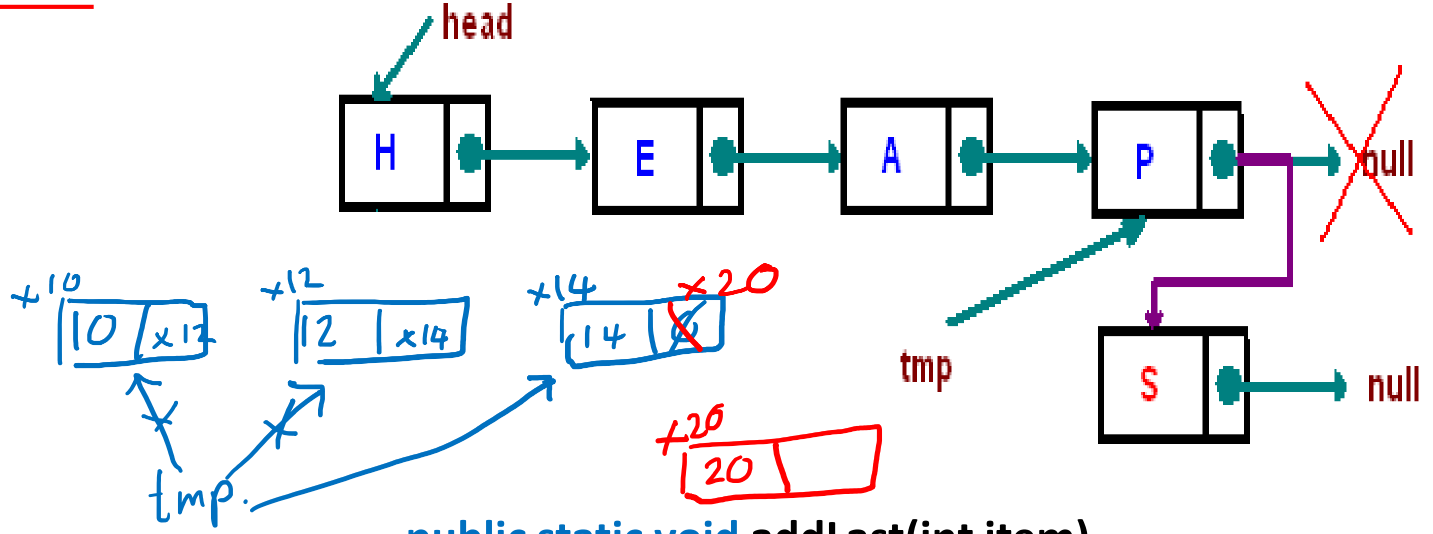
addFirst

- The method creates a node and prepends it at the beginning of the list.

1)Create a New Node with
Value=input value
Reference= Current Head Ref
2)Head = new Node Reference



JAVA Singly Linked List EXAMPLE -addLast



addLast

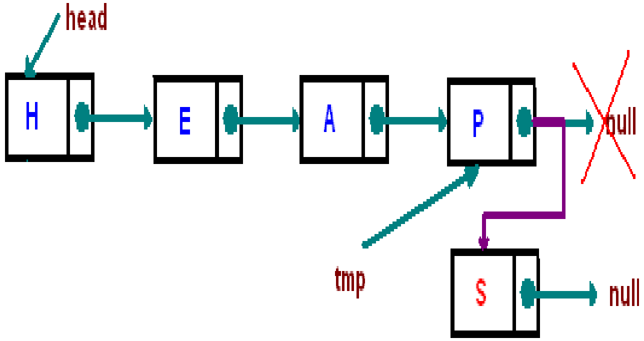
- The method appends the node to the end of the list.
- This requires traversing, but make sure you stop at the last node

```
public static void addLast(int item)
  {
(1) if( head == null)
    {   addFirst(item);  }
    else
    {
  (2) Node tmp = head;
       (3) while(tmp.next != null)
                 { tmp = tmp.next;}
     tmp.next = new Node(item, null);
    }  (5)        (4)
  }
```

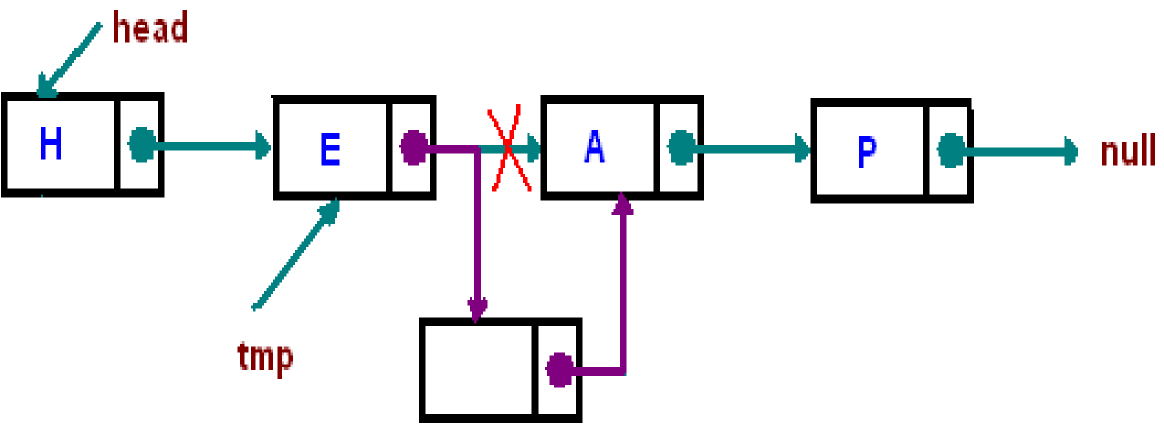
1)Check Whether linklist is empty if empty call addfirst
2)Create tmp node with reference to head
3)Going in a while loop until node’s reference variable becomes null. (That means we are in the last node)
4) Create a new node
Value=input value
Reference= null
5)Set the new node address to tmp’s reference variable

```
// A simple Java program to introduce a linked list
public class MyLinkedList
{
   static Node head;  // head of list

  static class Node {
       int data;
       Node next;
       Node(int d)  { data = d;  next=null; }
      Node(int d, Node t)  { data = d;  next=t; }
 // Constructor
   }
  public static void main(String[] args)
   {
      head  = new Node(1);
       Node second = new Node(2);
       Node third  = new Node(3);
       head.next = second;
       second.next = third;
       Display(head);
       addFirst(23);
       addLast (5);
       Display(head);

   }
    public static void addFirst(int item)
   {
      head = new Node(item, head);
   }
```

```
    static void  Display(Node currNode)
    {
      while(currNode != null )
       {
       System.out.println("Data:"+currNode.data );
         currNode=currNode.next;
       }
    }

   public static void addLast(int item)
    {
      if( head == null)
      {   addFirst(item);  }
      else
      {
        Node tmp = head;
                 while(tmp.next != null)
                        { tmp = tmp.next;
                           }
        tmp.next = new Node(item, null);
      }
    }
 }
```

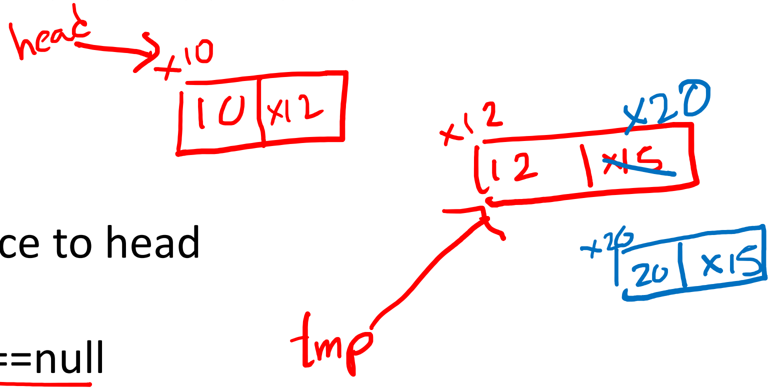


JAVA Singly Linked List **EXAMPLE -insertAfter**



insertAfter

- Find a node containing "key" and insert a new node after it.



x20
20
x15
x15
15
φ

1)Create a tmp node with reference to head

2) Go in a while loop until

tmp node's value == key Or tmp ==null

3)Inside the loop

tmp = tmp node's reference varaible

4) If tmp not equal to null means tmp node will have the key as the value

4.1)Create Node & Set Ref

Value= toInsert

Reference= tmp's Reference variable

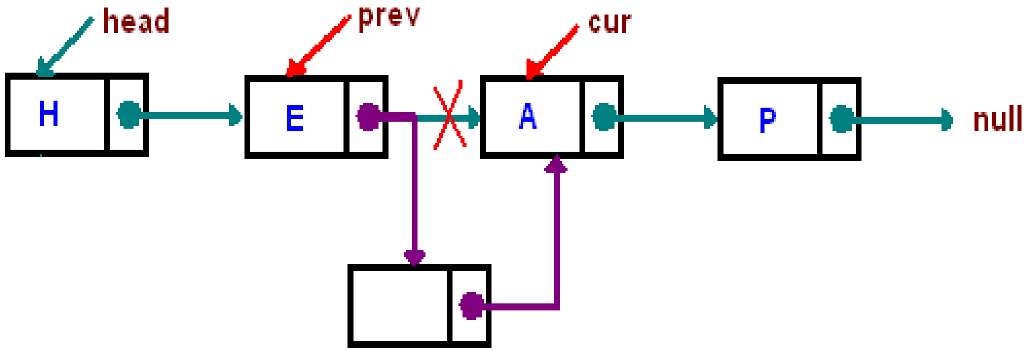
4.2)set tmp's Reference variable with the new node address

```
                                      12          20
public static void insertAfter(int key, int toInsert)
{
1 Node tmp = head;

2 while(tmp != null && tmp.data!=key)
       { tmp = tmp.next; }

4 if(tmp != null)
                                      20          15
     { tmp.next = new Node(toInsert, tmp.next); }
}            4.2                      4.1
```

JAVA Singly Linked List EXAMPLE –insertBefore – Method 1

insertBefore

- Find a node containing "key" and insert a new node before that node.



1)Create a tmp node with reference to head

1.1) Create a prev node & set as null

2) Go in a while loop until

tmp node’s value == key Or tmp ==null

3)Inside the loop

prev node = tmp node

tmp = tmp node's reference varaible

4) If tmp not equal to null means tmp node will have the key as the value

4.1 if tmp is null & prev is null means key is @ 1st Node, so we can call addfirst

4.2)Create Node & Set Ref

Value= toInsert

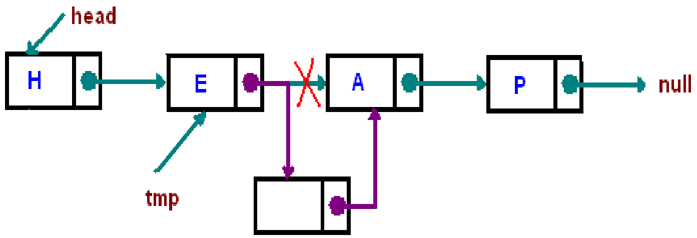
Reference= tmp’s variable

4.3)set prev’s Reference variable with the new node address

```
public static void insertBefore(int key, int toInsert)
 {

   Node tmp = head;
   Node prev =null;
   while(tmp != null && tmp.data !=key)
   {
      prev=tmp;
      tmp = tmp.next;
   }
   if(tmp !=null && prev ==null)
      { addFirst(toInsert); }
   else if(tmp != null )
    {  prev.next = new Node(toInsert, tmp); }
 }
```

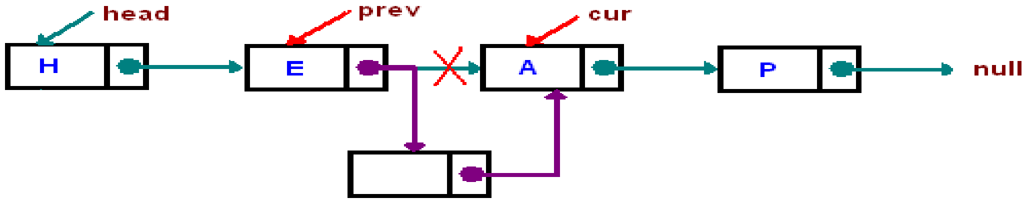
JAVA Singly Linked List EXAMPLE – insertBefore – Method 1

```
public static void insertAfter(int key, int toInsert)
{
  Node tmp = head;

  while(tmp != null && tmp.data!=key)
          { tmp = tmp.next; }

  if(tmp != null)
     { tmp.next = new Node(toInsert, tmp.next); }
}
```



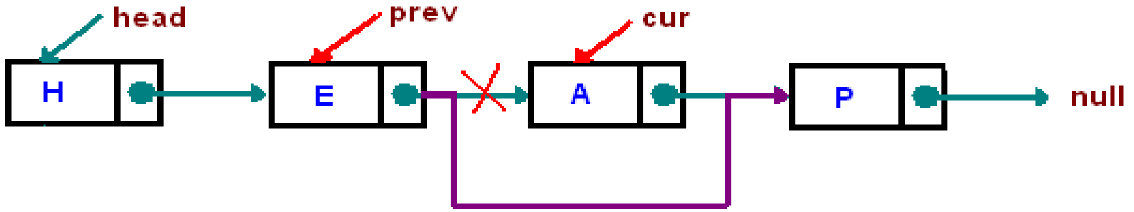
insertBefore

- Find a node containing "key" and insert a new node before that node.



```
public static void insertBefore(int key, int toInsert)
 {
    Node tmp = head;
    Node prev =null;
    while(tmp != null && tmp.data !=key)
    {
       prev=tmp;
       tmp = tmp.next;
    }
    if(tmp !=null && prev ==null)
       { addFirst(toInsert); }
    else if(tmp != null )
      {  prev.next = new Node(toInsert, tmp); }
 }
```

JAVA Singly Linked List EXAMPLE -remove



remove

- Find a node containing "key" and delete it.

(remove the Reference to that object)

1)Create a tmp node with reference to head
2) Go in a while loop until
 curr node’s value == key Or tmp ==null
3)Inside the loop
Prev= tmp address
tmp= reference variable of the tmp node
4) If tmp not equal to null , if it is true, that means tmp node will have the key as the value
4.1) set prev’s Reference variable with tmp’s reference varaible
prev.next=tmp.next;
(remove the Reference to that object that contain the key)

```
public static void remove(int key)
 {
    if(head==null)
    {
       return;
    }
   Node tmp = head;
   Node prev =null;
   while(tmp != null && tmp.data!=key)
   {
      prev=tmp;
      tmp = tmp.next;
   }
  if(tmp =!null && prev ==null) // key @head
     {head=head.next; }
   else if(tmp != null)
     { prev.next = tmp.next; }
}
```

```
public static void insertBefore(int key, int toInsert)
 {
   Node tmp = head;
   Node prev =null;
   while(tmp != null && tmp.data !=key)
   {
      prev=tmp;
      tmp = tmp.next;
   }
   if(tmp !=null && prev ==null)
      { addFirst(toInsert); }
   else if(tmp != null )
     {  prev.next = new Node(toInsert, tmp); }
 }
```

```
public static void remove(int key)
 {
     if(head==null)
     {
        return;
     }
    Node tmp = head;
    Node prev =null;
    while(tmp != null && tmp.data!=key)
    {
       prev=tmp;
       tmp = tmp.next;
    }
   if(tmp =!null && prev ==null) // key @head
       {head=head.next; }
    else if(tmp != null)
      { prev.next = tmp.next; }
}
```

JAVA Singly Linked List EXAMPLE -getLast

getLast

- Returns the last element in the list

```
 public static int getLast()
{
  if(head == null)
   {
             throw new RuntimeException("list is empty");
   }
  Node tmp = head;
  while(tmp.next != null)
           {
                tmp = tmp.next;
           }

  return tmp.data;
}
```

JAVA Singly Linked List EXAMPLE -getFirst

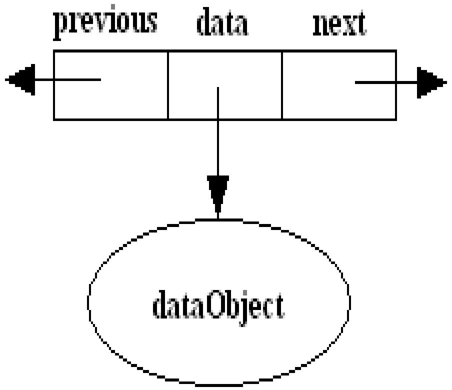
getFirst

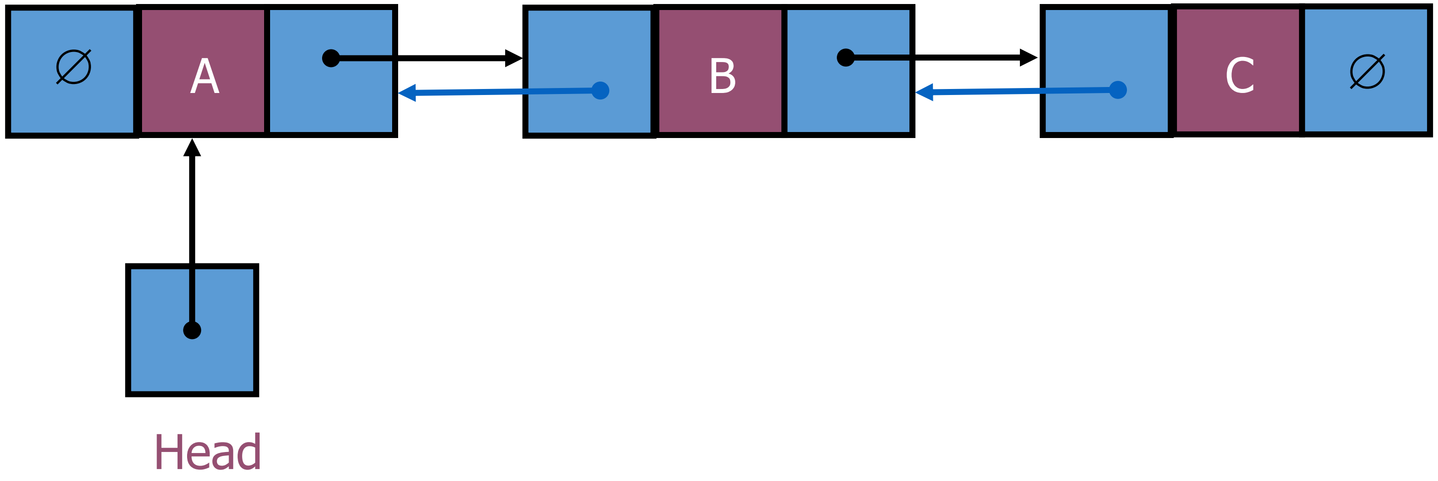
- Returns the first element in the list

```
public static int getFirst()
 {
        if(head == null)
        {
                throw new RuntimeException("list is empty");
        }
    else
      {
                return head.data;
      }
 }
```

Doubly-linked list

- A Doubly linked list is a dynamic data structure consisting of a sequence of *nodes*, forming a *linear* ordering.
- Each node stores
  - Element (data object)
  - Reference (i.e., address) to the next node
  - Reference (i.e., address) to the previous node





```
public class DoublyLinkList {
   static Node head;  // head of list
   static class Node {
      Node prev;
      int data;
      Node next;
      Node(int d) {
         data = d;next = null;prev=null;
      }
     Node(int d,Node p,Node n) {
         data = d; prev=p, next = n;
      }
   }
   public static void main(String[] args) {
      head = new Node(1);
      Node second = new Node(2);
      Node third = new Node(3);
      head.next = second;
      second.prev=head;
      second.next = third;
      third.prev=second;
      Display(head);
   }
```

Output

```
Forward Direction
1
2
3
Backward Direction
3
2
1
```

```
static void Display(Node currNode) {
      Node tail=null;
     System.out.println("Forward Direction:");
     while (currNode != null) {
        System.out.println(currNode.data);
        tail=currNode;
        currNode = currNode.next;
      }
      currNode=tail;
     System.out.println("Backward Direction:");
      while (currNode != null) {
        System.out.println(currNode.data);
        currNode = currNode.prev;
      }
   }
}
```

```
public class DoublyLinkList {
   static Node head;  // head of list
   static class Node {
      Node prev;
      int data;
      Node next;
      Node(int d) {
         data = d;next = null;prev=null;
      }
     Node(int d,Node p,Node n) {
         data = d; prev=p, next = n;
      }
   }
   public static void main(String[] args) {
      head = new Node(1);
      Node second = new Node(2);
      Node third = new Node(3);
      head.next = second;
      second.prev=head;
      second.next = third;
      third.prev=second;
      addFirst(99);
      addLast(70);
      Display(head);
   }
```

```
public static void addFirst(int item)
{
     Node tmp=head;
     head = new Node(item,null, head);
     tmp.prev=head;

}
```

```
public static void addLast(int item)
    {
      if( head == null)
      {   addFirst(item);  }
      else
      {
        Node tmp = head;
        while(tmp.next != null)
                 { tmp = tmp.next;
                  }
        tmp.next = new Node(item,tmp,null);
      }
   }
}
```

```
public class DoublyLinkList {
   static Node head;  // head of list
   static class Node {
      Node prev;
      int data;
      Node next;
      Node(int d) {
         data = d;next = null;prev=null;
      }
     Node(int d,Node p,Node n) {
         data = d; prev=p, next = n;
      }
   }
   public static void main(String[] args) {
      head = new Node(1);
      Node second = new Node(2);
      Node third = new Node(3);
      head.next = second;
      second.prev=head;
      second.next = third;
      third.prev=second;
      addFirst(99);
      addLast(70);
      Display(head);
   }
```

```
public static void addFirst(int item)
{
     Node tmp=head;
     head = new Node(item,null, head);
     tmp.prev=head;

}
```

```
public static void addLast(int item)
   {
     if( head == null)
     {   addFirst(item);  }
     else
     {
      Node tmp = head;
      while(tmp.next != null)
              { tmp = tmp.next;
               }
      tmp.next = new Node(item,tmp,null);
     }
   }
}
```

```
public static void addFirst(int item)
   {
     Node tmp=head;
     head = new Node(item,null, head);
     tmp.prev=head;
   }
```

```
public static void addLast(int item)
    {
      if( head == null)
      {   addFirst(item);  }
      else
      {
        Node tmp = head;
        while(tmp.next != null)
                  { tmp = tmp.next;
                  }
        tmp.next = new Node(item,tmp,null);
      }
    }
```

```
public static void insertAfter(int key, int toInsert)
{
   Node tmp = head;
   while(tmp != null && tmp.value!=key)
              { tmp = tmp.next; }
   if(tmp != null)
       {       Node NextRef = tmp.next;
               tmp.next = new Node(toInsert, tmp, NextRef);
               if(NextRef!= null)
               {
                  NextRef.prev=tmp.next ;
               }
       }
}
```

SLL

```
public static void insertAfter(int key, int toInsert)
  {
    Node tmp = head;

    while(tmp != null && tmp.data!=key)
            { tmp = tmp.next; }

    if(tmp != null)
       { tmp.next = new Node(toInsert, tmp.next); }
  }
```

DLL

```
public static void insertAfter(int key, int toInsert)
  {
    Node tmp = head;
    while(tmp != null && tmp.value!=key)
            { tmp = tmp.next; }
    if(tmp != null)
      {     Node NextRef = tmp.next;
            tmp.next = new Node(toInsert, tmp, NextRef);
            if(NextRef!= null)
            {
              NextRef.prev=tmp.next ;
            }
      }
  }
```

```
public static void insertAfter(int key, int toInsert)
{
   Node tmp = head;
   while(tmp != null && tmp.value!=key)
              { tmp = tmp.next; }
   if(tmp != null)
       {      Node NextRef = tmp.next;
              tmp.next = new Node(toInsert, tmp, NextRef);
              if(NextRef!= null)
              {
                 NextRef.prev=tmp.next ;
              }
       }
}
```

```
public static void insertBefore(int key, int toInsert)
{
   Node tmp = head;
   while(tmp != null && tmp.value!=key)
              { tmp = tmp.next; }

   if(tmp != null)
       {   Node PrevRef= tmp.prev;
              tmp.prev = new Node(toInsert, tmp.prev,tmp);
              if(PrevRef!=null)
               {  PrevRef.next=tmp.prev; }
              else
              {
               head= tmp.prev ;
              }
       }
}
```

```
public static void insertAfter(int key, int toInsert)
{
  Node tmp = head;
  while(tmp != null && tmp.value!=key)
          { tmp = tmp.next; }
  if(tmp != null)
     {     Node NextRef = tmp.next;
           tmp.next = new Node(toInsert, tmp, NextRef);
           if(NextRef!= null)
           {
             NextRef.prev=tmp.next ;
          }
     }
}
```

```
public static void insertBefore(int key, int toInsert)
{
  Node tmp = head;
  while(tmp != null && tmp.value!=key)
          { tmp = tmp.next; }

  if(tmp != null)
     {  Node PrevRef= tmp.prev;
          tmp.prev = new Node(toInsert, tmp.prev,tmp);
          if(PrevRef!=null)
           {  PrevRef.next=tmp.prev; }
          else
          {
           head= tmp.prev ;
          }
      }
}
```

```
public static void insertBefore(int key, int toInsert)
{
   Node tmp = head;
   while(tmp != null && tmp.value!=key)
              { tmp = tmp.next; }

   if(tmp != null && tmp.prev!=null)
       {   Node PrevRef= tmp.prev;
              tmp.prev = new Node(toInsert, tmp.prev,tmp);
               PrevRef.next=tmp.prev;
     }
    else if (tmp != null && tmp.prev==null)
     {  addFirst(toInsert); }
}
```

Java LinkedList class

- Uses doubly linked list to store the elements
- can contain duplicate elements.
- maintains insertion order.
- manipulation is fast because no shifting needs to be occurred.

```
import java.util.*;
public class TestCollection{
 public static void main(String args[]){
  LinkedList<String> al=new LinkedList<>();
```

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element

Java LinkedList class(doubly linked list) EXAMPLE 1

```
import java.util.*;
public class TestCollection7{
 public static void main(String args[]){
        LinkedList<Integer> al=new LinkedList<>()
        al.add(12);
          al.add(34);
          al.add(55);
          al.add(67);

          Iterator<Integer> itr=al.iterator();
          while(itr.hasNext()){
                System.out.println(itr.next());
                }
    }
}
```

Java LinkedList class(doubly linked list) EXAMPLE 2

```
import java.util.*;

class Book {
        int id;
        String name,author;
        public Book(int id, String name, String author)
         {
           this.id = id;
           this.name = name;
           this.author = author;
        }
}
```

```
public class LinkedListExample {
  public static void main(String[] args) {
           //Creating list of Books
           LinkedList<Book> listData=new LinkedList<>();
           //Creating Books
           Book b1=new Book(101,"C",“Saman");
           Book b2=new Book(102,“NTWK",“Kamal");
           Book b3=new Book(103,“OS",“Nimal");
           //Adding Books to list
           listData.add(b1);
           listData.add(b2);
           listData.add(b3);
          //Traversing list  Method 1
          Iterator<Book> itr=listData.iterator();
          while(itr.hasNext()){
                Book b=itr.next();
                System.out.println(b.id+" "+b.name+" "+b.author);
                }
          //Traversing list  Method 2
           for(Book b:listData){
                System.out.println(b.id+" "+b.name+" "+b.author);
           }
   }
}
```

Java LinkedList class(doubly linked list) EXAMPLE 3

```
import java.util.*;
public class LinkedListDemo {
   public static void main(String args[]) {
                 LinkedList ll = new LinkedList();
                 // add elements to the linked list
                 ll.add("F");
                 ll.add("B");
                 ll.add("D");
                 ll.add("E");
                 ll.add("C");
                System.out.println("Original contents of ll: " + ll);
                 ll.addLast("Z");
                System.out.println("After addLast Z contents of ll: " + ll);
                 ll.addFirst("A");
                System.out.println("After addFirst A contents of ll: " + ll);
                 ll.add(1, "A2");
                System.out.println("After add 1,A2 contents of ll: " + ll);
               ll.remove("F");
                System.out.println("After remove F contents of ll: " + ll);
                 ll.remove(2);
                System.out.println("After remove index 2 contents of ll: " + ll);
                 // remove first and last elements
                 ll.removeFirst();
                   System.out.println("After remove First contents of ll: " + ll);
                 ll.removeLast();
                   System.out.println("After remove Last contents of ll: " + ll);
                  Object val = ll.get(2);
                 ll.set(2, (String) val + " Changed");
                 System.out.println("ll after change: " + ll);
   }
}
```

```
OutPut:
Original contents of ll:                [F, B, D, E, C]
After addLast Z contents of ll:         [F, B, D, E, C, Z]
After addFirst A contents of ll:        [A, F, B, D, E, C, Z]
After add 1,A2 contents of ll:          [A, A2, F, B, D, E, C, Z]
After remove F contents of ll:          [A, A2, B, D, E, C, Z]
After remove index 2 contents of ll:    [A, A2, D, E, C, Z]
After remove First contents of ll:      [A2, D, E, C, Z]
After remove Last contents of ll:       [A2, D, E, C]
ll after change:                        [A2, D, E Changed, C]
```

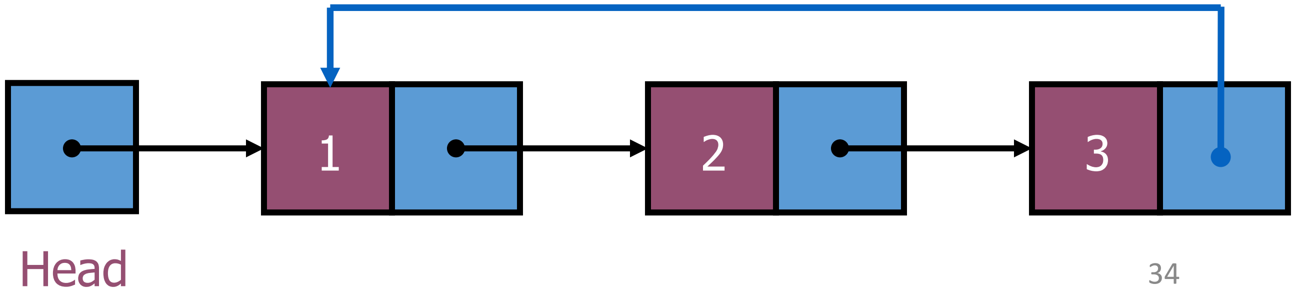
DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier

- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

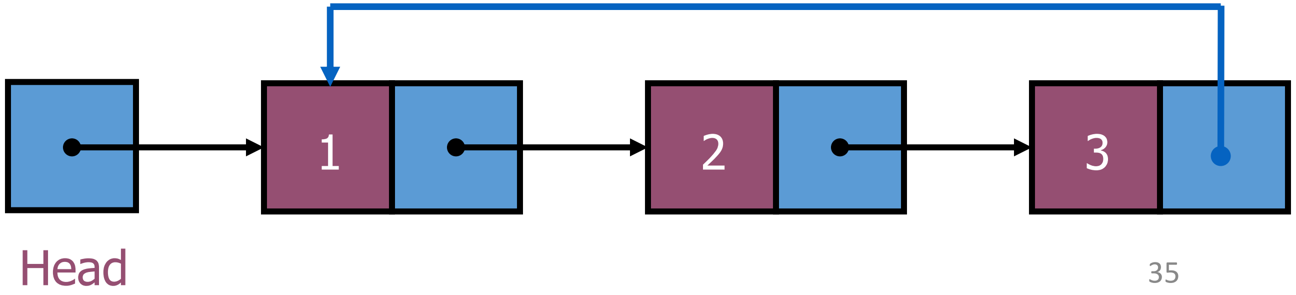
Circular linked lists

- The last node points to the first node of the list
- How do we know when we have finished traversing the list?



Circular linked lists

- The last node points to the first node of the list
- How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)



Java Circular linked lists EXAMPLE 1

```
public class CirLinkList {
          static Node head;  // head of list
           static class Node {
              int data;
              Node next;
                     Node(int d) {
                        data = d;
                        next = null;
                      }
           }
           public static void main(String[] args) {
              head = new Node(1);
              Node second = new Node(2);
              Node third = new Node(3);
              head.next = second;
              second.next = third;
              third.next = head;
              Display(head);
           }
```

```
static void Display(Node currNode) {
      Node temp = currNode;
      if(temp != null)
        {
              do {
                 System.out.println("Data:" + temp.data);
                     temp = temp.next;
              } while (temp!=head) ;
        }
 }

}
```

The last node points to the first node of the list
How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Java Circular linked lists EXAMPLE 1

```
public class CirLinkList {
          static Node head;  // head of list
          static class Node {
             int data;
             Node next;
                    Node(int d) {
                       data = d;
                       next = null;
                     }
          }
          public static void main(String[] args) {
             head = new Node(1);
             Node second = new Node(2);
             Node third = new Node(3);
             head.next = second;
             second.next = third;
             third.next = head;
             Display(head);
          }
```

```
static void Display(Node currNode) {
       Node temp = currNode;
       if(temp != null)
         {
                do {
                   System.out.println("Data:" + temp.data);
                       temp = temp.next;
                } while (temp!=head) ;
         }
 }
```

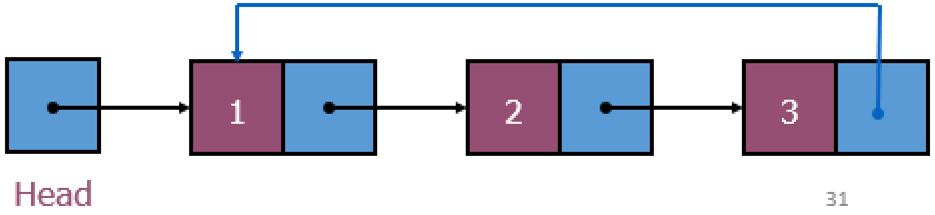
```
}
```

The last node points to the first node of the list
How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Java Circular linked lists EXAMPLE 2

```
class Node{
   int data;
   Node next;
   public Node(int data){
      this.data = data;
   }
}

class CircularLinkedList {
   static int size =0;
   static Node head=null;
   static Node tail=null;
        public void addNodeAtStart(int data){
        ----------
        }
        public void print(){
        ----------
        }
}
```

```
public class CirLLExp {
  public static void main(String[] args) {
     CircularLinkedList c = new CircularLinkedList();
     c.addNodeAtStart(3);
     c.addNodeAtStart(2);
     c.addNodeAtStart(1);
     c.print();
 }
}
```



```
public void addNodeAtStart(int data){
    System.out.println("Adding node " + data + " at start");
      Node n = new Node(data);
      if(size==0){
         head = n;
         tail = n;
         n.next = head;
      }else{
         n.next = head;
         head = n;
         tail.next = head;
      }
      size++;
   }
```

```
public void print(){
      System.out.print("Circular Linked List:");
      Node temp = head;
      if(size<=0){
         System.out.print("List is empty");
      }else{
         do {
            System.out.print(" " + temp.data);
            temp = temp.next;
         }
         while(temp!=head);
      }
      System.out.println();
   }
```

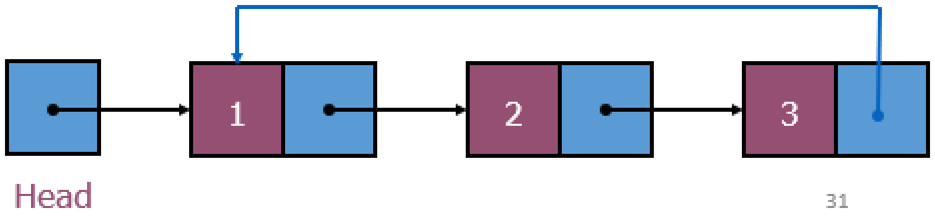
```
public class CirLLExp {
  public static void main(String[] args) {
    CircularLinkedList c = new CircularLinkedList();
    c.addNodeAtStart(3);
    c.addNodeAtStart(2);
    c.addNodeAtStart(1);
    c.print();
 }
}
```

Java Circular linked lists EXAMPLE 2

```
class Node{
   int data;
   Node next;
   public Node(int data){
      this.data = data;
   }
}

class CircularLinkedList {
   static int size =0;
   static Node head=null;
   static Node tail=null;
        public void addNodeAtEnd(int data){
        ----------
        }
        public void addNodeAtStart(int data){
        ----------
        }
        public void deleteNodeAtStart(){
        ----------
        }
        public void print(){
        ----------
        }
}
```

CirLLExp.java

```
public class CirLLExp {
  public static void main(String[] args) {
     CircularLinkedList c = new CircularLinkedList();
     c.addNodeAtStart(3);
     c.addNodeAtStart(2);
     c.addNodeAtStart(1);
     c.print();
     c.addNodeAtEnd(4);
     c.print();
     c.deleteNodeAtStart();
     c.print();
   }
}
```



```
public void addNodeAtStart(int data){
    System.out.println("Adding node " + data + " at start");
      Node n = new Node(data);
      if(size==0){
         head = n;
         tail = n;
         n.next = head;
      }else{
         Node temp = head;
         n.next = temp;
         head = n;
         tail.next = head;
      }
      size++;
   }
```

```
public void addNodeAtEnd(int data){
      if(size==0){
         addNodeAtStart(data);
      }else{
         Node n = new Node(data);
         tail.next =n;
         tail=n;
         tail.next = head;
         size++;
      }
  System.out.println("\nNode " + data + " is added at the end of list");
}
```

```
public void deleteNodeAtStart()
   {
      if(head != null)
      {
          if(size==1)
         { head=null; tail=null; }
         else
          { tail.next=head.next;
         head=head.next; }
          size--;
      }
      else{
         System.out.println("List is Empty");
      }
   }
```

```
public void print(){
      System.out.print("Circular Linked List:");
      Node temp = head;
      if(size<=0){
         System.out.print("List is empty");
      }else{
         do {
            System.out.print(" " + temp.data);
            temp = temp.next;
         }
         while(temp!=head);
      }
      System.out.println();
   }
```

Advantages of Circular Linked Lists:

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

- Useful for implementation of queue. Unlike this implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.