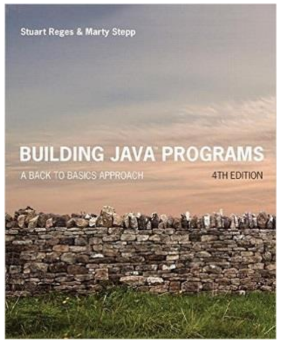
Abstract Data Type

Reading Assignment: Read chapter 16

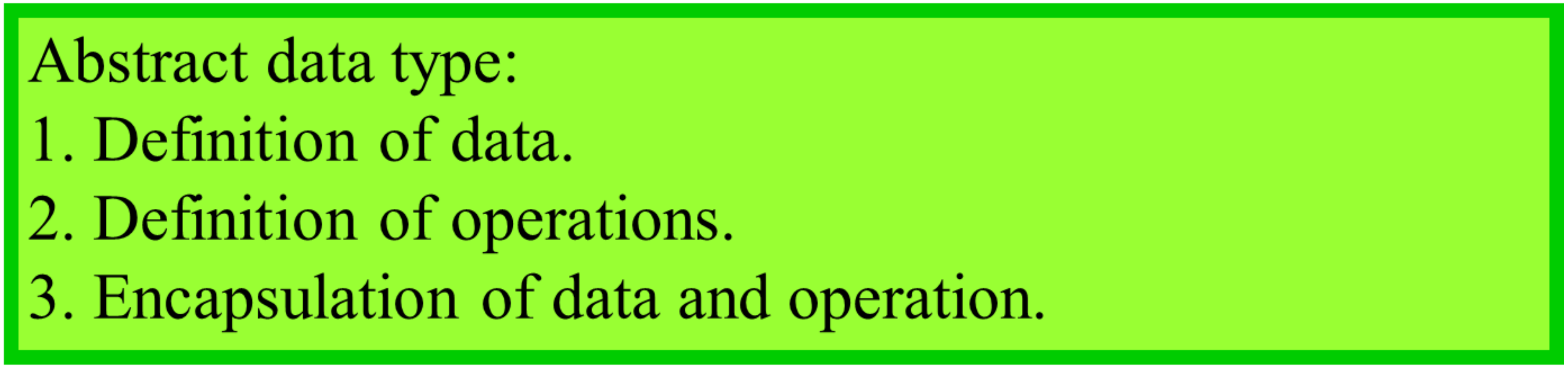
Building Java Programs (Stuart Reges and Marty Stepp)



Also: Credits https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

What is an ADT ?

- An abstract data type is a **data type** packaged with the **operations** that are meaningful for the data type. We then encapsulate the data and the operations on the data and hide them from the user.



Abstract data type:

1. Definition of data.
2. Definition of operations.
3. Encapsulation of data and operation.

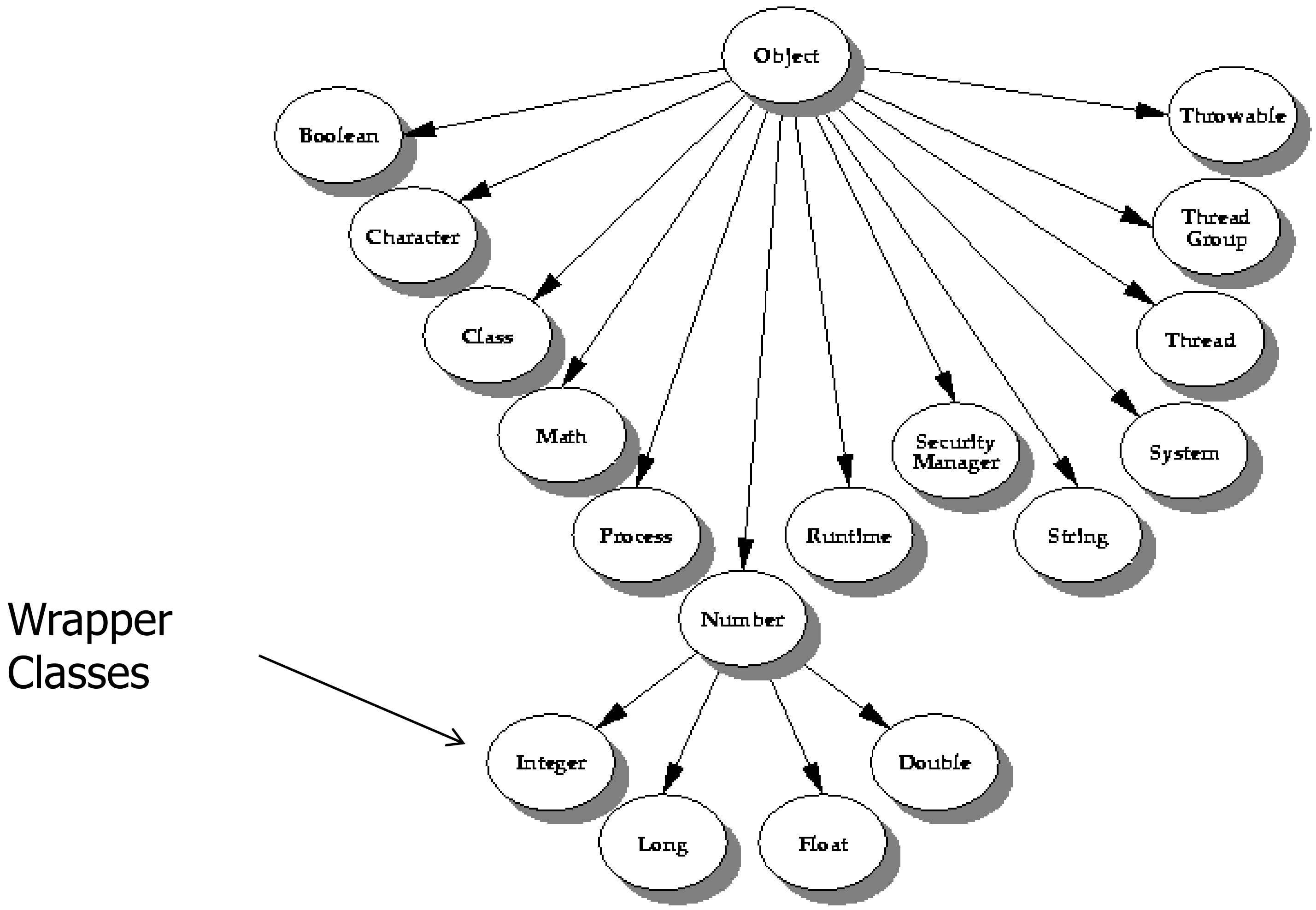
Why ADT

- To abstract is to leave out unimportant parts of the information keeping the more important parts.
- In an ADT we leave out details of implementation. So, programmers can concentrate more on the problem solving.
- An ADT should be implemented such that the programmers who use the type do not need to know the details of <u>how the values are represented and how the operations are implemented</u>.
- High level languages often provide built in ADTs.
  - the C++ STL (Standard Template Library)
  - the Java standard library

A Few Useful Packages (Standard Library)

Package	Description
java.lang	Provides classes that are fundamental to the design of the Java programming language (imported automatically)
java.util	Contains the collections framework (all your data structures you learned in school), date & time, etc
java.io	Provides for system input and output through data streams, serialization and the file system
java.math	Provides classes for performing arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal)
java.net	Provides the classes for implementing networking applications. (sockets, connections, etc)
javax.swing	For gui (graphic user interface)

Superclass



The java.lang package contains the collection of base types (language types) that are always imported into any given compilation unit. This is where you'll find the declarations of Object (the root of the class hierarchy) and Class, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.

Source: http://www.oracle.com/technetwork/java/libraries-140433.html

Object: The Cosmic Superclass

- **The *Object* class is the highest super class (ie. *root* class) of Java.**
- **All classes defined without an explicit extends clause automatically extend *Object* .**
- **Most useful methods:**
  - **String toString()**
  - **boolean equals(Object otherObject)**
  - **Object clone() // Create copies**
- **Good idea to override these methods in your classes**

Wapper classes

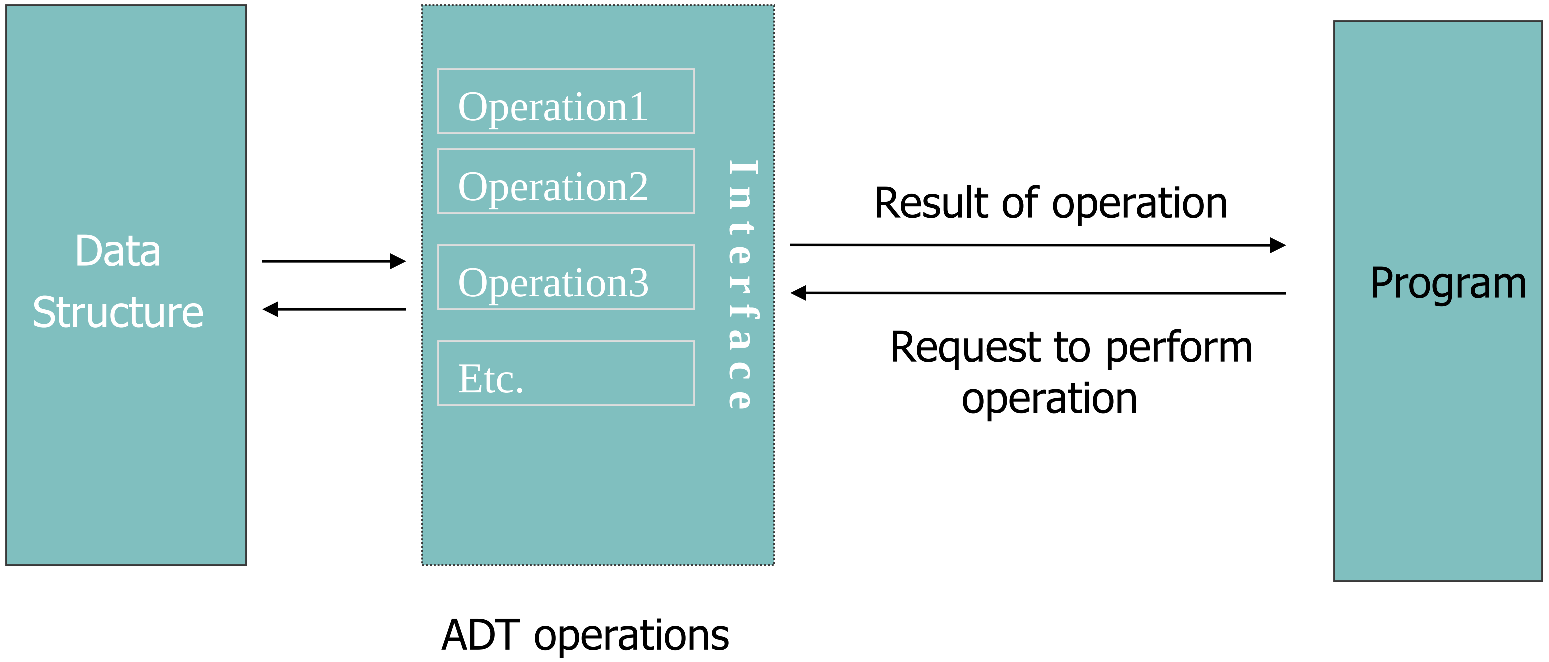
Type wrappers are classes used to enclose a simple value of a primitive type into an object.

1. Integer
2. Long
3. Boolean
4. Character
5. Double
6. Float

Notes:

- All wrapper objects are immutable. Once created the contained value can't be changed.
- Wrapper classes are final and can't be subclassed

Data structure, ADT, and Program

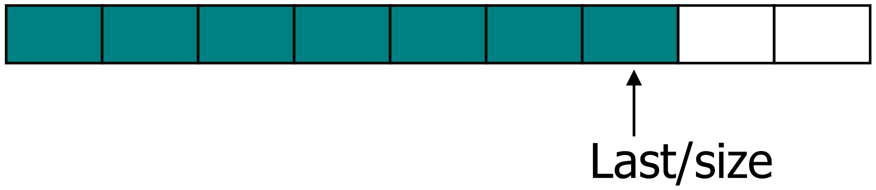


Methods for representing values of data structures

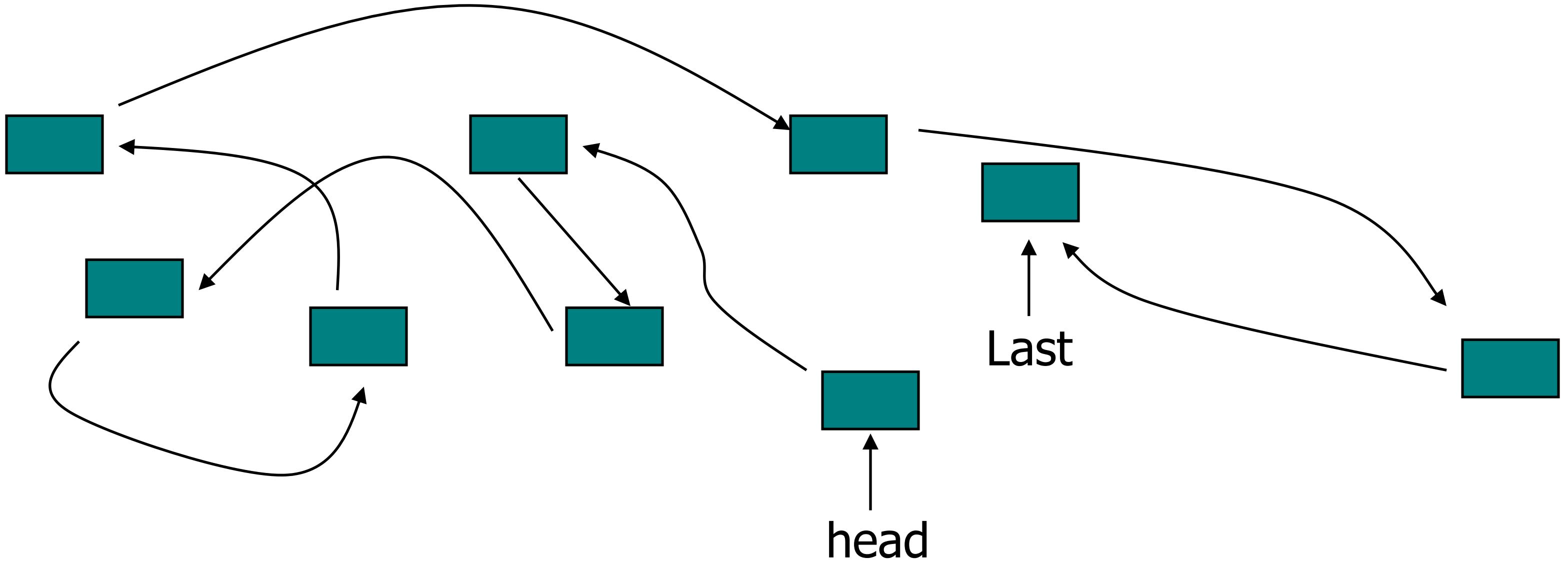
1. Array based
   - There are better algorithms for sorting and searching.
2. Reference based/pointer based/Linked/Chaining
   - Insertion and removal of elements can be done faster.

Array based VS Reference based

- Array based

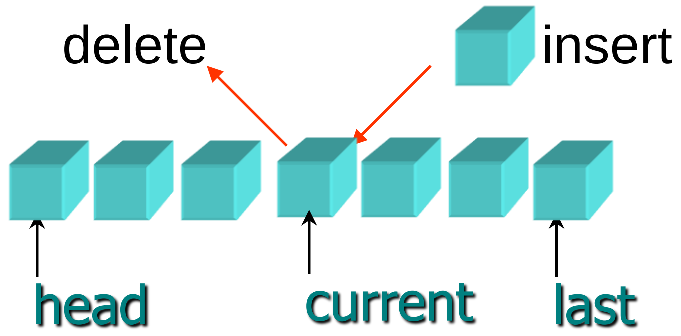


- Reference based



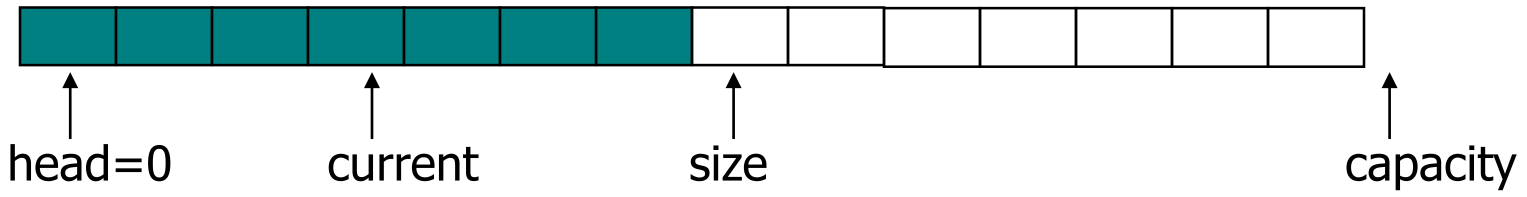
List ADT

- A collection of objects in a specific order and having the same type.



- Primary operations:
  - insertBefore(e) – Add an element e before the current position.
  - insertAfter(e) – Add an element e after the current position.
  - remove(e) – Element e is removed from the list.
  - current() – Returns the current element.
  - size() – Returns the number of elements on the list.
  - forward() – Move the current position forward one position.
  - backward() - Move the current position backward one position.
  - resetCurrent – Reset the current position at the head element.

Array Implementation List ADT

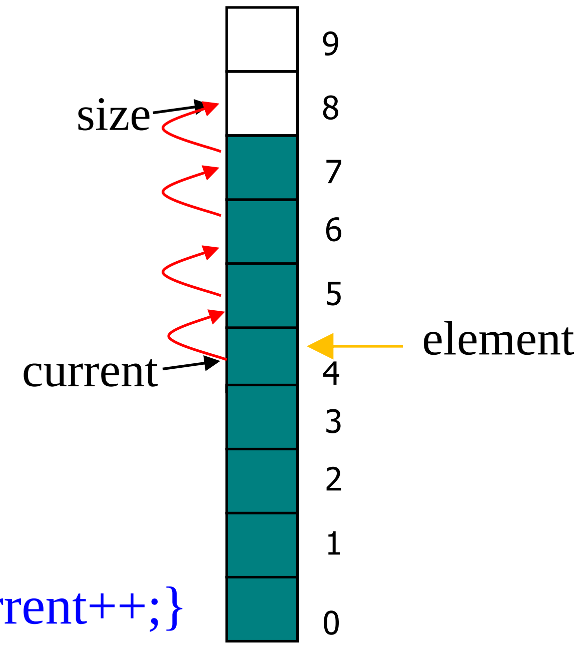


Representation:

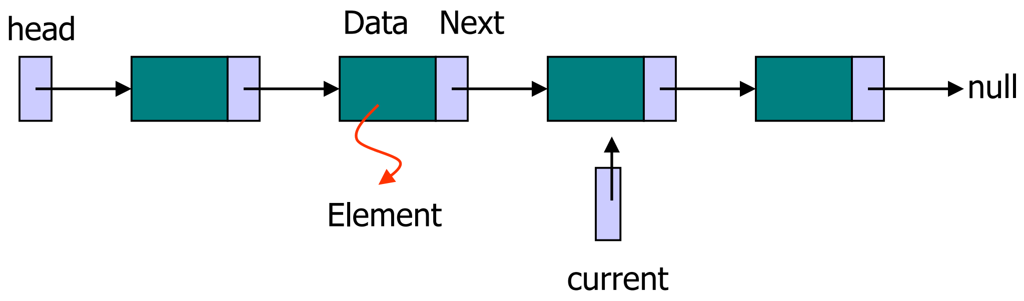
```
public class List {
    private int size=0, current=-1, capacity=default ;
    private Object L[];
    public List(int maxSize) {
        capacity = maxSize;
        L = new Object[capacity];
    }
}
```

Array Implementation List ADT conti.

```
public int size() { return size; }
public Object current() { if (current>=0) return L[current]; else return null;}
public void insertBefore(Object element) {
    if (size==capacity) return;
    if (size>0)
        for (int i=size-1; i>=current;--i) L[i+1] = L[i];
    else current = 0;
    L[current] = element;
    size++;
}
public void forward() { if (size>0 && current<size-1) current++;}
```



Linked Implementation List ADT

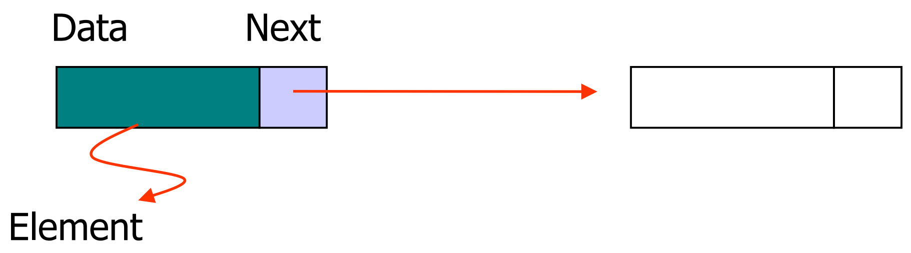


Representation: Singly linked list.

Drawback: Can only process elements in one direction.

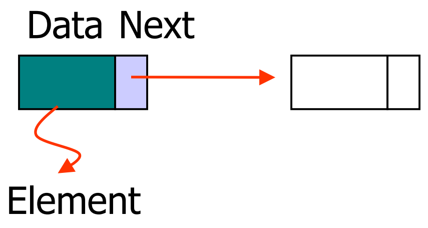
```
public class List {
    private int size=0;
    private Node current, head ;
}
```

Representation of nodes



Representation of nodes

```
public class Node {
        private Object Data = null;
        private Node Next = null;
        public Node(Object Element) { Data = Element; }
        public void setNext(Node N) { Next = N; }
        public Node getNext() {  return Next; }
        public Object getElement() {  return Data; }
        public void setElement(Object Element) { Data = Element; }
 }
```

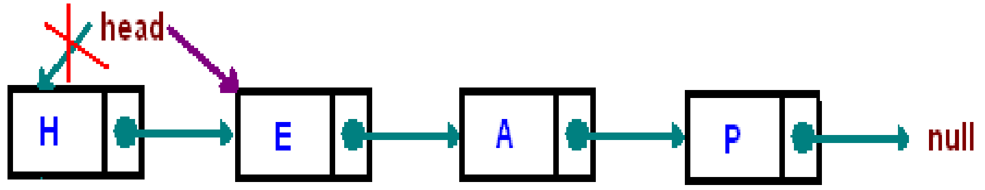


Linked Implementation List ADT conti.

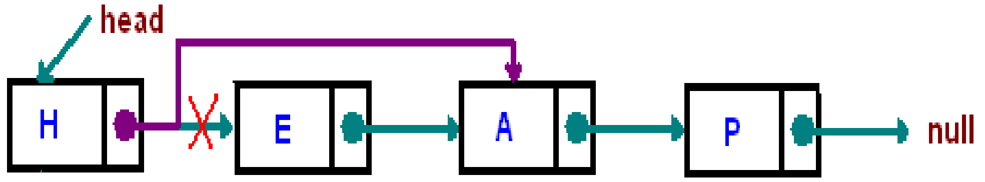
```
public int size() { return size; }
public Object current() {
    if (current!=null) return current.getElement(); else return null;
}
public void forward() {
    Node tmp = current.getNext();
    if (tmp!=null) current = tmp;
}
```

More Examples

```
head = head.next;
```



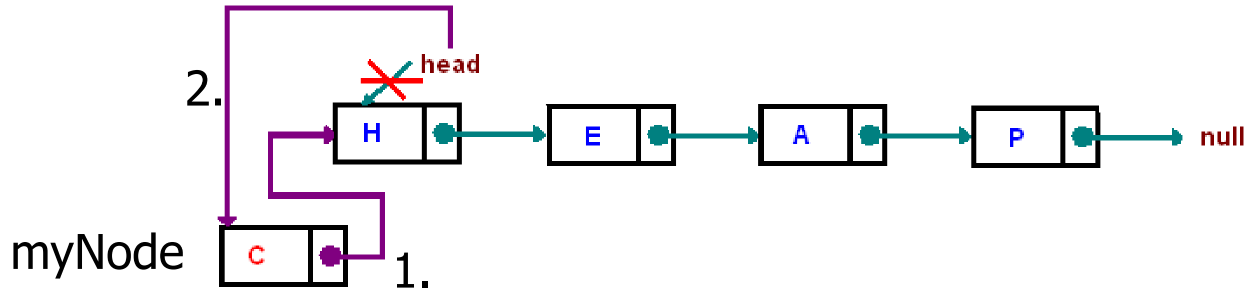
```
head.next = head.next.next;
```



addFirst - public void addFirst(Object o)

addFirst

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



1. Create a new node. Have its next points to node head.
2. update head to point to the new node.

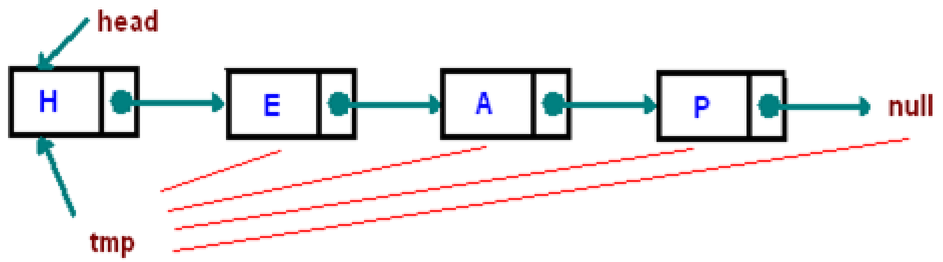
Do not forget: (3) Initialization of current node for first node added. (4) Increase size.

```
//Initialize a new Node with Object o
and a null next pointer
Node tmp = new Node(o, head);

// head = tmp;
//If current is currently pointing at null,
point to the head, otherwise stay where it is
current = current==null?head:current;
size++;
```

Traversing

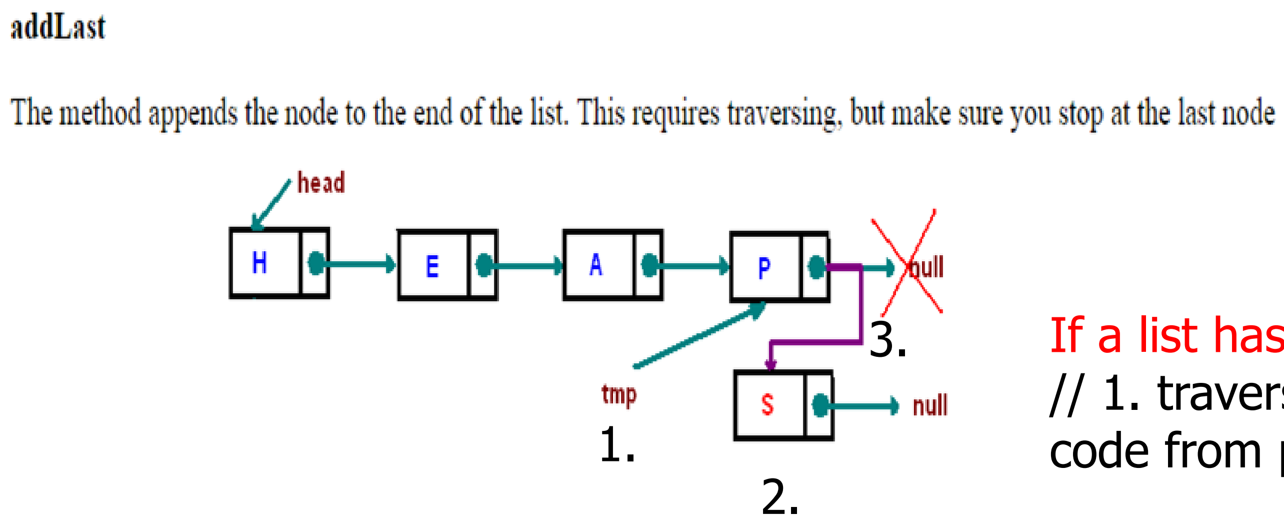
Traversing

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



```
// Initialize tmp to start from head
Node tmp = head;
// Loop till run into null at the end:
while(tmp.Next != null)
          tmp = tmp.Next;
```

addLast public void addLast(Object o)



addLast

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

1. traverse to the last node.
2. create a new node. Have its next points to null.
3. have the last node points to the new node.

Note: (4) Initialization of cur for first node added. (5) Increase size

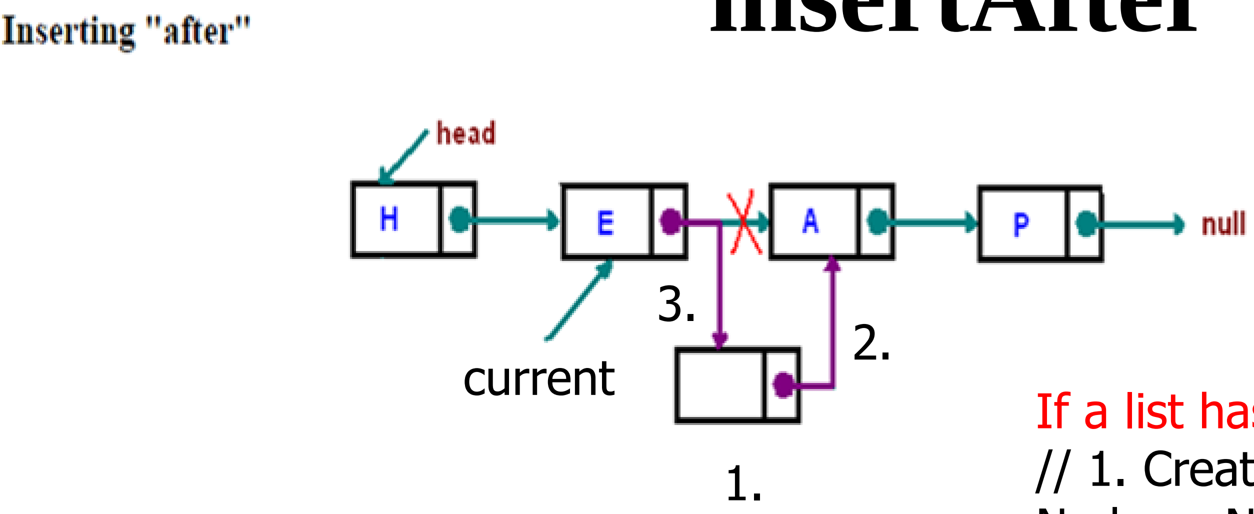
If a list has at least one element
// 1. traverse to the last node. Use code from previous slide (slide 21).

// 2.
Node myNode = new Node(o,null);

// 3. tmp is computed from step 1
tmp.setNext(myNode);

// 4. Increase size
size += 1;
else
// add first element on the list
   addFirst(o); // size increased here

insertAfter



1. create new node.
2. have new node point to where current node points to.
3. have current node points to new node.

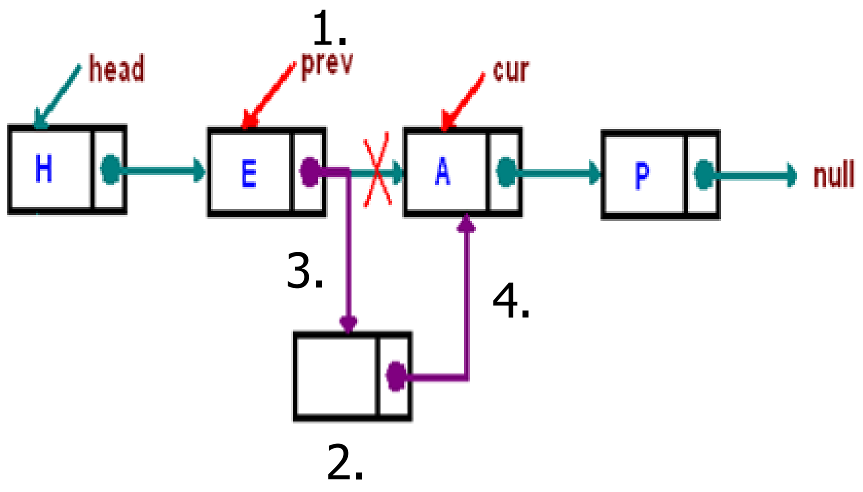
If a list has at least one element

```
// 1. Create new node
Node myNode = new Node(o,null);

// 2.  new node points to where current pointed to
myNode.seNext (current.getNext());

// 3. current node points to new node
current.setNext(myNode);

size += 1;
```

Else return null.

Pre-condition: list must have one element to insert after

insertBefore



1. find a previous node using current node.
2. create a new node.
3. have previous node points to new node.
4. have the new node points to current node.

```
If a list is null, return
Else if a list has one element
     if(head == current) {
        addFirst(o);
 Else // list has two or more elements
// 1. get previous node
      backward();
      prev = current;
      forward();
// 2. create new node
       Node myNode = new Node(o,null);
// 3. previous node points to new node
        prev.seNext(myNode);
// 4. new node point to current node
        myNode.setNext(current);
 size += 1;
```

Pre-condition: list must have one element to insert before

forWard method

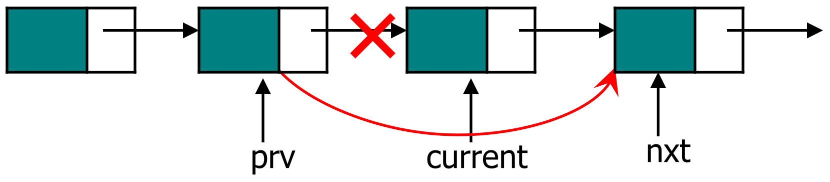
```
 // Move the current position forward one position

public void forward() {
    Node tmp = current.getNext();
    if (tmp!=null)
        current = tmp;
}
```

backWard

```
// Move the current position backward one position
public void backwards()
 {
    if(head !=current)
    {
       //Create a node to traverse the list in order to
       // find the Node before the current one
       Node tmp = head;
       // While the next node for tmp is not the current one
       // and not the end of the list, step forward one node
       while ((tmp.getNext()!=current)&&(tmp.getNext()!=null))
       {
          tmp = tmp.getNext();
       }
       current = tmp;
    }
 }
```

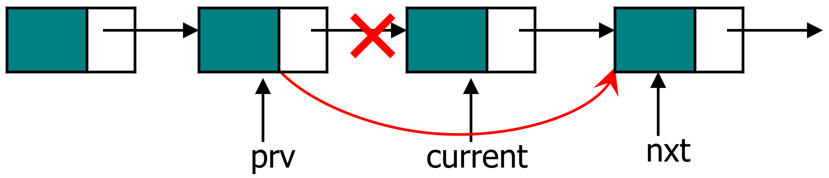
Linked Implementation List ADT conti.

```
public void remove(Object o) {
 if (size!=0)       {
        //Node prev will point to the Node just before the node being removed
        Node prev = null;  Node tmp = head;
        while (tmp.getNext()!=null && tmp.getElement()!=o) {
           prev = tmp;
           tmp = tmp.getNext();
        }
        if (tmp.getElement()==o)   {
            current = current==tmp ? prev:current;
            prev.setNext(tmp.getNext()); // General condition
            size--;
        }
     }
}
```

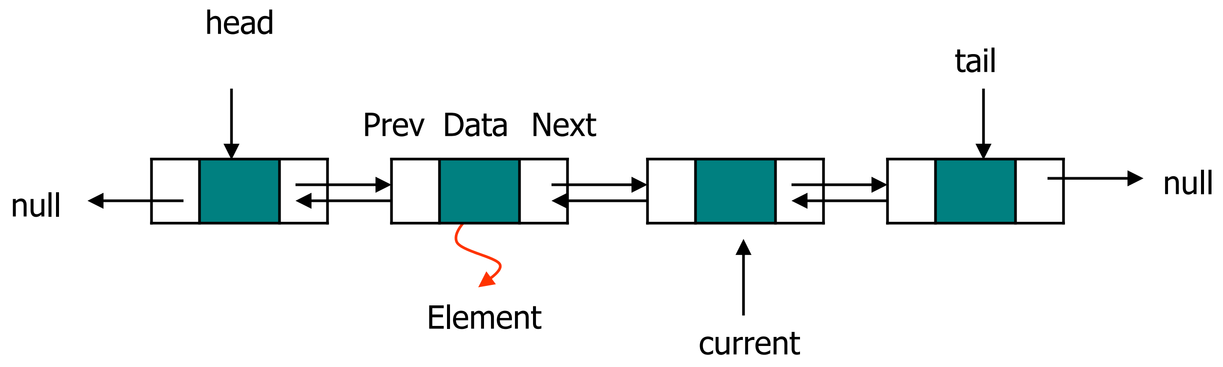


Linked Implementation List ADT conti.

```
public void remove(Object o) {
 if (size!=0)      {
        //Node prev will point to the Node just before the node being removed
        Node prev = null;  Node tmp = head;
        while (tmp.getNext()!=null && tmp.getElement()!=o) {
          prev = tmp;
          tmp = tmp.getNext();
        }
        if (tmp.getElement()==o)   {
          current = current==tmp ? prev:current;
          if (prev != null)  prev.setNext(tmp.getNext());
          else  head = head.getNext(); // Still: Not complete yet!
          size--;
        }
     }
}
```



Doubly Linked Implementation List ADT



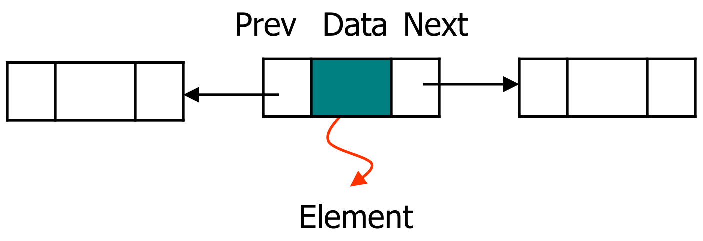
Representation: Doubly linked list.

Drawback: Waste more memory space on node references.

```
public class List {
    private int size=0,;
    private DLNode current, Head, Tail ;
}
```

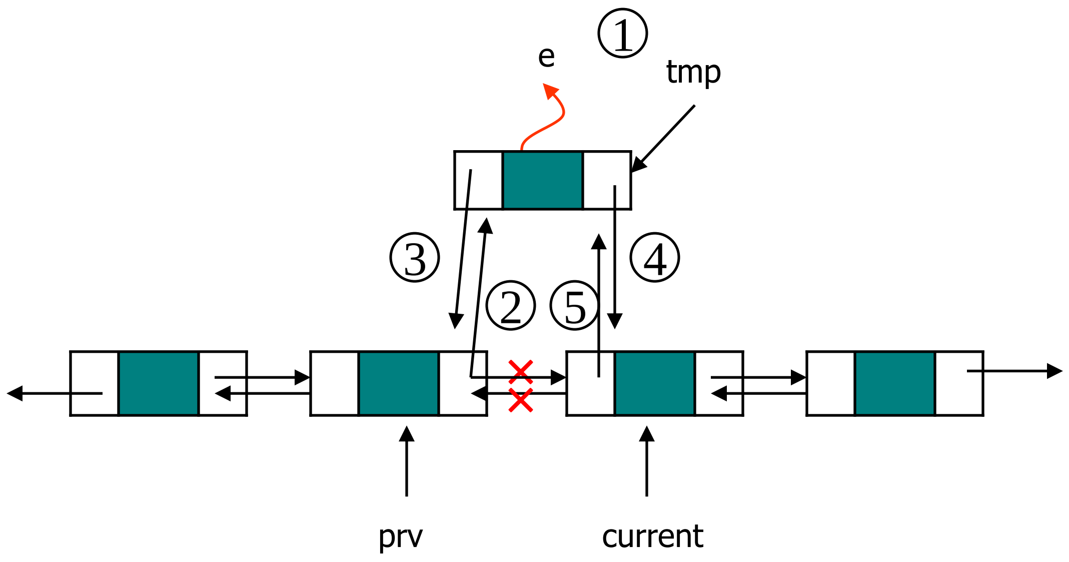
Representation of doubly linked nodes

```
public class DLNode {
        public Object Data = null;
        public DLNode Prev= null, Next=null;
 }
```



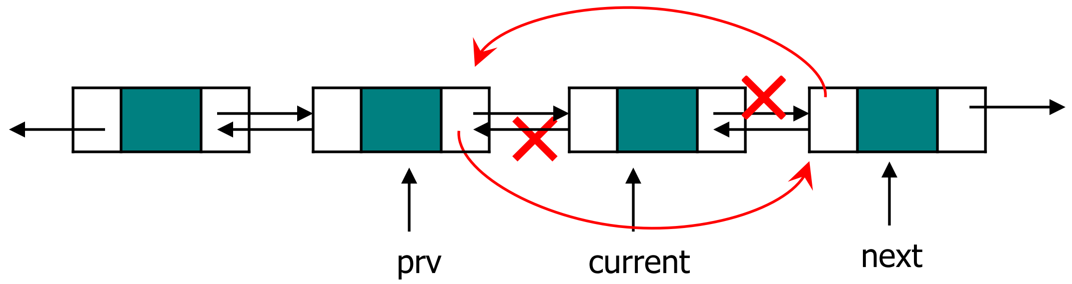
Doubly Linked Implementation List ADT conti.

```
public void insertBefore(Object e) {
    DLNode prv = current==null ? Tail : current.Prev;
①   DLNode tmp = new DLNode(); tmp.Data = e;
②   if (prv!=null) prv.Next = tmp; else Head = tmp;
③   tmp.Prev = prv;
④   tmp.Next = current;
⑤   if (current!=null) current.Prev = tmp; else Tail = tmp;
    size++;
}
```



DoublyLinked Implementation List ADT conti.

```
public void remove() {
    if (current==null) return;
    DLNode prv = current.Prev;
    DLNode nxt = current.Next;
    if (prv!=null) prv.Next = nxt; else head = nxt;
    if (nxt!=null) nxt.Prev = prv; else tail = prv;
    current = nxt;
    if (current==null && prv!=null) current = prv;
    size--;
}
```



Programming by Contract

PreConditions and PostConditions

- PreCondition
  - what you assume to be true before an operation or process
- PostCondition
  - what you assert to be true after an operation or process
- Contract: If Preconditions hold before, then Postconditions will hold afterwards

Assertion Violations

- **What happens if a precondition or a postcondition fails (i.e., evaluates to false)**
  - **The assertions can be checked (i.e., monitored) dynamically at run-time to debug the software**
  - **A *precondition violation* would indicate a bug at the *caller***
  - **A *postcondition violation* would indicate a bug at the *callee***

- **Our goal is to prevent assertion violations from happening**
  - **The pre and postconditions are not supposed to fail if the software is correct**
    - **hence, they differ from exceptions and exception handling**
  - **By writing the contracts explicitly, we are trying to avoid contract violations, (i.e, failed pre and postconditions)**

Meaning of Pre and Post conditions

- **Precondition: If preconditions are not obeyed by the client of the class’s method , the service provider will deny it service.**

- **Postcondition: If any postconditions is violated, it uncovers a problem on the service provider side.**

Simple example

```
/**
 * Precondition: num2 is not zero.
 * Postcondition: Returns the quotient of num1 and num2.
 */
```

-
  ```
  public double divide(double num1, double num2)
  {
     return num1 / num2;
  }
  ```

Simple example (in real life - safety)

```
/**
 * Precondition: num2 is not zero.
 * Postcondition: Returns the quotient of num1 and num2. If
num2 is zero, zero is returned.
 */
```

-
  ```
  public double divide(double num1, double num2)
  {
     if (num2 != 0)
         return num1 / num2;
     return 0;
  }
  ```