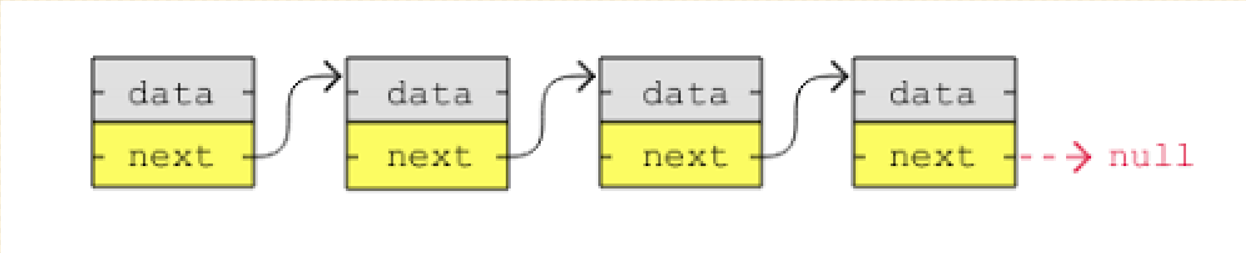
# linked list



# Linked list

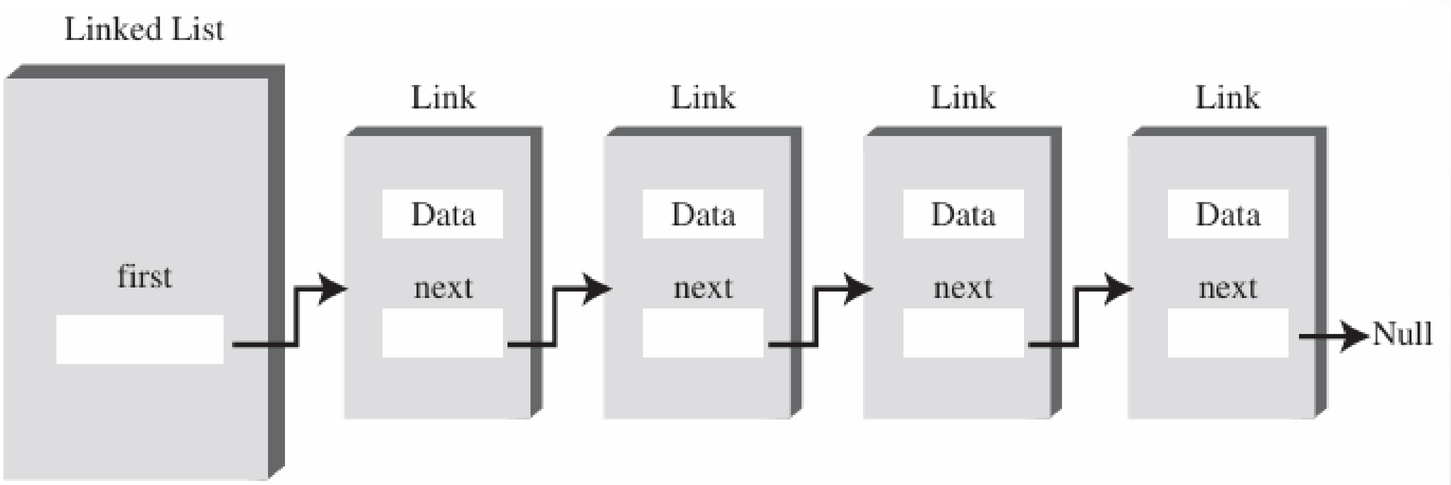
- **Linked list Data Structure**

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node.


Linked list Data Structure

# Links

- In a linked list, each data item is embedded in a link.
- A link is an object of a class called Link.
- Each Link object contains a reference (usually called next) to the next link in the list. A field in the list itself contains a reference to the first link. This relationship is shown below




Links in a list.

# Link List in Java

```
class Link
   {
   public int iData;        // data
   public double dData;     // data
   public Link next;        // reference to next link
   }
```

- **This kind of class definition is sometimes called self-referential because it contains a field—called next in this case—of the same type as itself.**
- **an object of a class can be used instead of the items**

```
class Link
   {
   public inventoryItem iI;  // object holding data
   public Link next;         // reference to next link
   }
```

# A Simple Linked List

- **The operations allowed in this version of a list are :**
  - **Inserting an item at the beginning of the list**
  - **Deleting the item at the beginning of the list**
  - **Iterating through the list to display its contents**
  - **Finding a Specified Links**
  - **Deleting Specified Links**

**these operations are all you need to use a linked list as the basis for a stack**

# A Simple Linked List

```
class Link
   {
   public int iData;              // data item
   public double dData;           // data item
   public Link next;              // next link in list
// ------------------------------------------------------------
   public Link(int id, double dd) // constructor
      {
      iData = id;                 // initialize data
      dData = dd;                 // ('next' is automatically
      }                           //  set to null)
```

**There's no need to initialize the next field because it's automatically set to null when it's created. (However, you could set it to null explicitly, for clarity.) The null value means it doesn't refer to anything, which is the situation until the link is connected to other links.**

# A Simple Linked List

```
public void displayLink()      // display ourself
   {
   System.out.print("{" + iData + ", " + dData + "} ");
   }
}  // end class Link
```

# A Simple Linked List

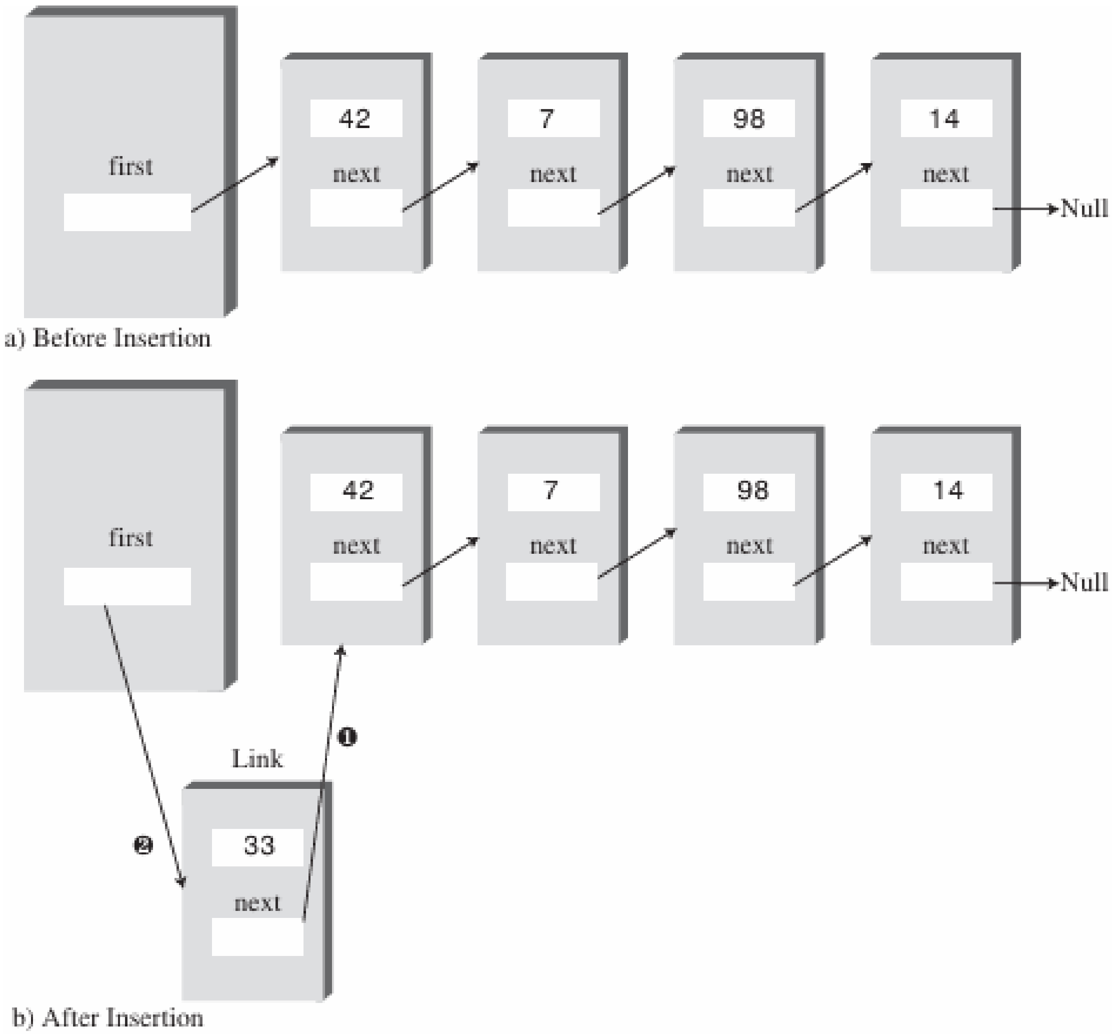
```
class LinkList
   {
   private Link first;             // ref to first link on list

// -------------------------------------------------------------
   public void LinkList()          // constructor
      {
      first = null;                // no items on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()        // true if list is empty
      {
      return (first==null);
      }
```

**The LinkList class contains only one data item: a reference to the first link on the list**

# A Simple Linked List

## The insertFirst() Method




To insert the new link, we need only set the next field in the newly created link to point to the old first link and then change first so it points to the newly created link.

# A Simple Linked List

## The insertFirst() Method

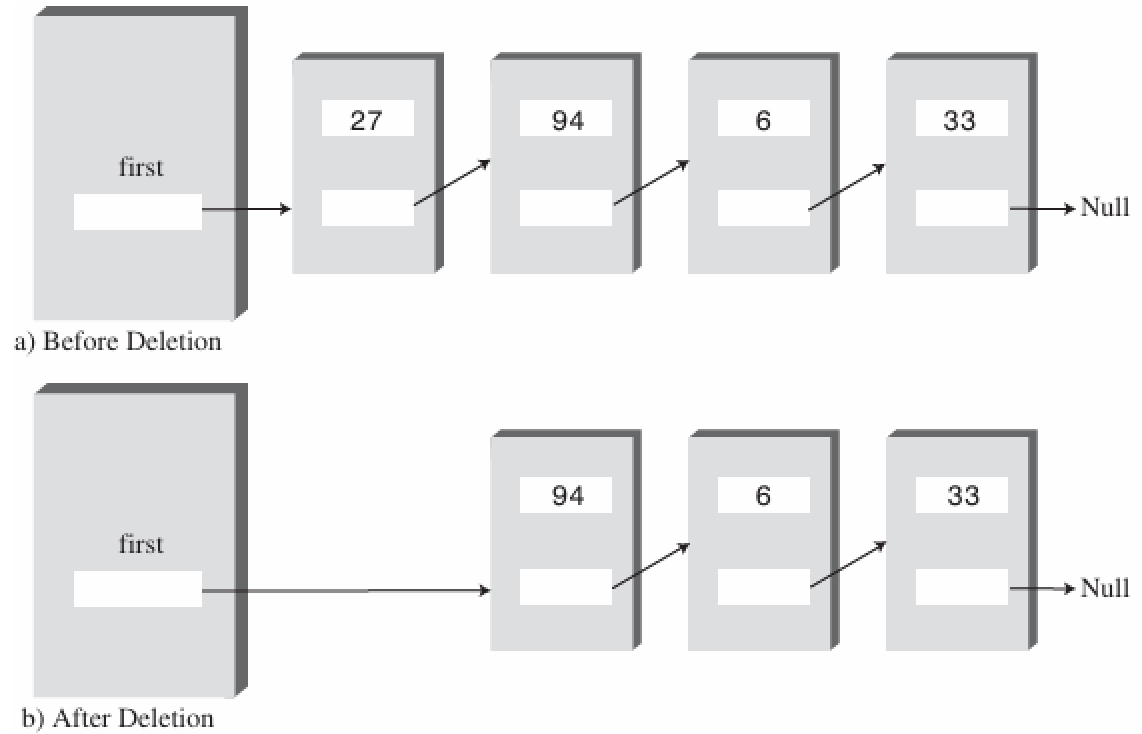
```
public void insertFirst(int id, double dd)
   {                                // make new link
   Link newLink = new Link(id, dd);
   newLink.next = first;            // newLink --> old first
   first = newLink;                 // first --> newLink
   }
```

In insertFirst() we begin by creating the new link using the data passed as arguments. Then we change the link references as following :

1. set the next field in the newly created link to point to the old first link
2. change first *(the first link in the linkList)*so it points to the newly created link.

# A Simple Linked List

## The deleteFirst() Method




**The deleteFirst() method disconnects the first link by rerouting first to point to the second link. This second link is found by looking at the next field in the first link:**

# A Simple Linked List

## The deleteFirst() Method

```
public Link deleteFirst()          // delete first item
   {                               // (assumes list not empty)
   Link temp = first;              // save reference to link
   first = first.next;             // delete it: first-->old next
   return temp;                    // return deleted link
   }
```

Notice that the deleteFirst() method assumes the list is not empty. Before calling it, your program should verify this fact with the isEmpty() method.

# A Simple Linked List

## The displayList() Method

```
public void displayList()
   {
   System.out.print("List (first-->last): ");
   Link current = first;       // start at beginning of list
   while(current != null)      // until end of list,
      {
      current.displayLink();   // print data
      current = current.next;  // move to next link
      }
   System.out.println("");
   }
```

1. start at first and follow the chain of references from link to link.
2. A variable current points to each link in turn.
3. It starts off pointing to first, which holds a reference to the first link.
4. The statement current = current.next; changes current to point to the next link

# A Simple Linked List

## Finding a Specified Links

```
public Link find(int key)       // find link with given key
   {                            // (assumes non-empty list)
   Link current = first;              // start at 'first'
   while(current.iData != key)        // while no match,
      {
      if(current.next == null)        // if end of list,
         return null;                 // didn't find it
      else                            // not end of list,
         current = current.next;      // go to next link
      }
   return current;                    // found it
   }
```

# A Simple Linked List

## Deleting a Specified Link

```
public Link delete(int key)    // delete link with given key
   {                           // (assumes non-empty list)
   Link current = first;              // search for link
   Link previous = first;
   while(current.iData != key)
      {
      if(current.next == null)
         return null;                 // didn't find it
      else
         {
         previous = current;          // go to next link
         current = current.next;
         }
      }                               // found it
   if(current == first)               // if first link,
      first = first.next;             //    change first
   else                               // otherwise,
      previous.next = current.next;   //    bypass it
   return current;
   }
```

# A Simple Linked List App (1)

```
class LinkListApp
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // make new list

      theList.insertFirst(22, 2.99);      // insert four items
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // display list

      while( !theList.isEmpty() )         // until it's empty,
         {
         Link aLink = theList.deleteFirst();   // delete link
         System.out.print("Deleted ");         // display it
         aLink.displayLink();
         System.out.println("");
         }
      theList.displayList();              // display list
      }  // end main()
   }  // end class LinkListApp
```

# A Simple Linked List App(2)

```
class LinkList2App
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // make list

      theList.insertFirst(22, 2.99);      // insert 4 items
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // display list

      Link f = theList.find(44);          // find item
      if( f != null)
         System.out.println("Found link with key " + f.iData);
      else
         System.out.println("Can't find link");

      Link d = theList.delete(66);        // delete item
      if( d != null )
         System.out.println("Deleted link with key " + d.iData);
      else
         System.out.println("Can't delete link");

      theList.displayList();              // display list
      }  // end main()
   }  // end class LinkList2App
```

# Linked-List Efficiency

- Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time.
- Finding, inserting or deleting item requires searching through, on the average, half the items in the list. This requires O(N) comparisons, An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted. The increased effi ciency can be significant, especially if a copy takes much longer than a comparison.
- important advantage of linked lists over arrays is that a linked list uses exactly as much memory as it needs and can expanded while the size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or too small.