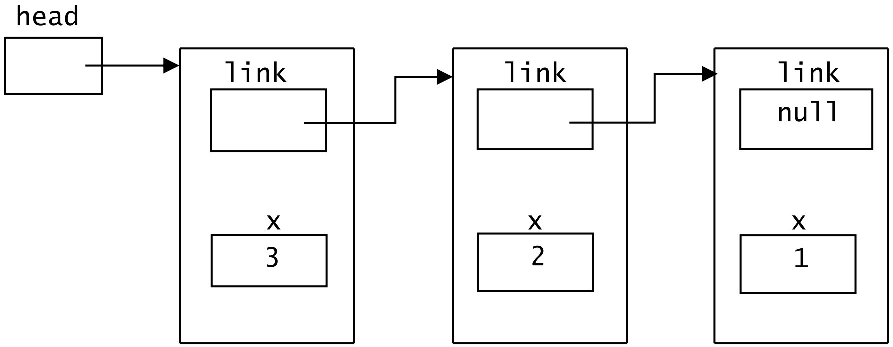
Chapter 15

Linked lists

Advantage of linked lists

Link lists can grow and shrink to the precise size needed.

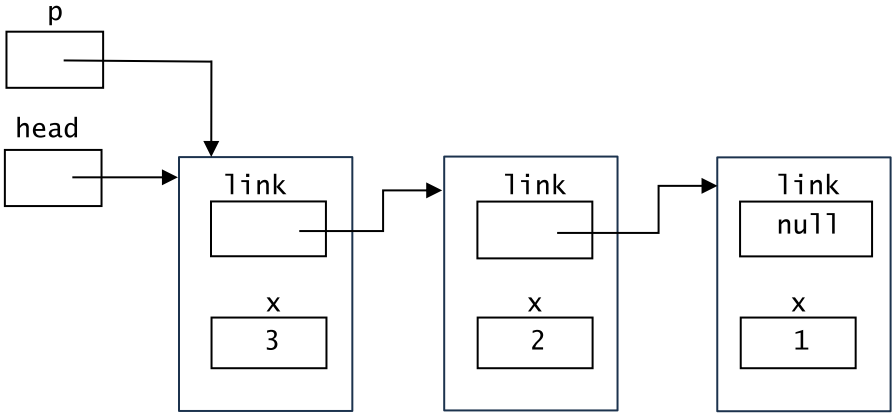
Working with linked lists



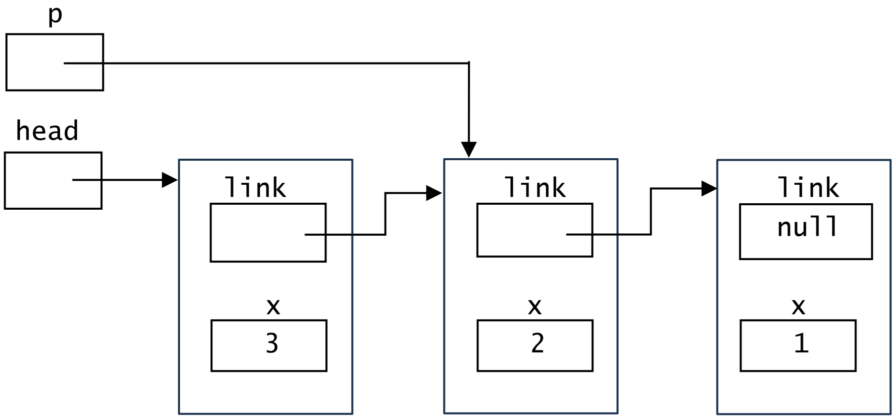
Traversing the list

```
Node p = head;
p = p.link;
```

a)



a)

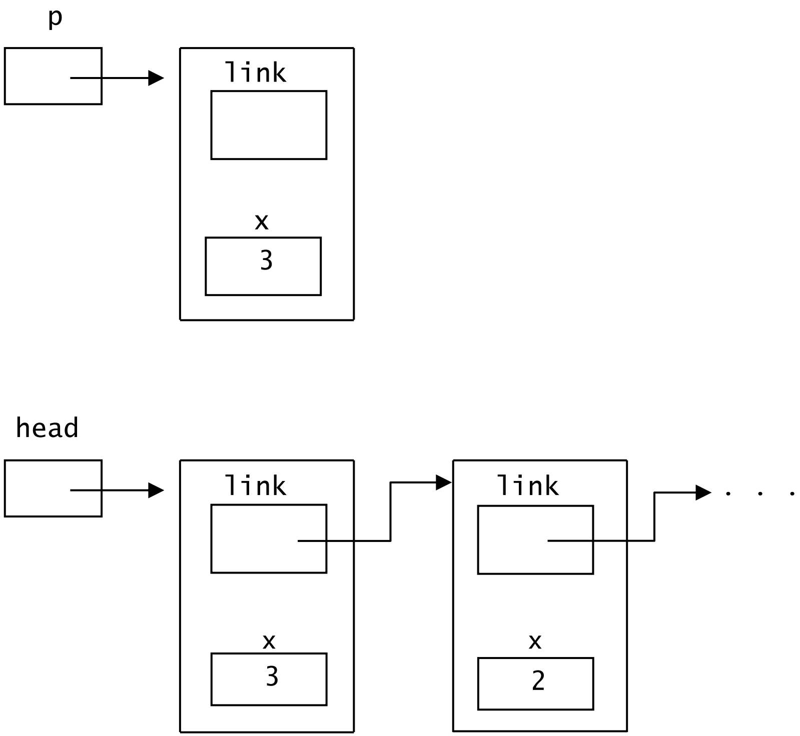


Loop to traverse list

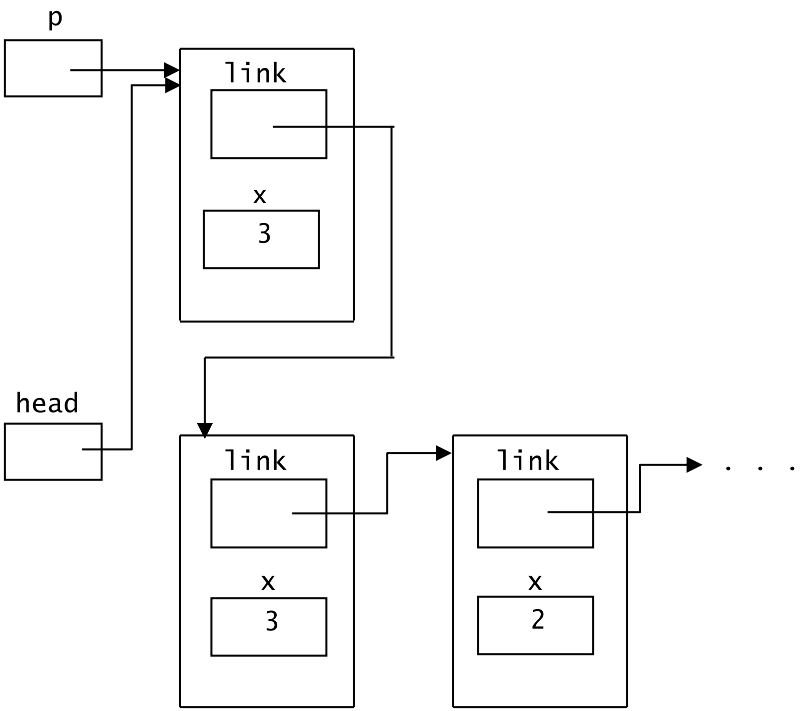
```
Node p = head;
while (p != null)
{
   System.out.println(p.x);
   p = p.link;
}
```

Adding a node to a linked list

a)



b)



Two steps to add node

1) The `link` field of the new node must be set to point to the node that is currently the first node. We accomplish this with

```
p.link = head;
```

2) `head` must be set to point to the new node. We accomplish this with

```
head = p;
```

Simple Linked List

```
class Node      // Node definition
{
   Node link;   // points to next node or is null if at last node in list
   int x;       // data field
}
class SimpleList1     // simple program that creates/traverses list
{
  public static void main(String[] args)
   {
      Node head = null;     // set head to null so list initially empty

      Node p = new Node();  // create new node
      p.link = head;        // make link field point to currrent 1st node
      p.x = 1;              // initialize x field in new node to 1
      head = p;             // make head point to new node so it is 1st

      p = new Node();       // create second node
      p.link = head;        // make link field point to currrent 1st node
      p.x = 2;              // initialize x field in second node to 2
      head = p;             // make head point to new node so it is 1st

      p = new Node();       // create third node
      p.link = head;        // make link field point to currrent 1st node
      p.x = 3;              // initialize x field in third node to 3
      head = p;             // make head point to new node so it is 1st

      p = head;               // assign pointer to 1st node on list to p
      while (p != null)     // use p, not head, so head not corrupted
      {
         System.out.println(p.x);  // display x in node p that points to
         p = p.link;               // move p to next node on list
      }
   }
}
```

```
class SimpleList2
{
   static Node head = null; // static so addNode/traverse can access
   public static void main(String[] args)
   {
      addNode(1);  // add node with 1 to the beginning of list
      addNode(2);  // add node with 2 to the beginning of list
      addNode(3);  // add node with 3 to the beginning of list
      traverse();  // traverse list from first node to last
   }
   public static void addNode(int value)
   {
      Node p = new Node(); // create new node
      p.link = head;       // make new node point to current 1st node
      p.x = value;         // initialize x in new node
      head = p;            // make head to point to new node
   }
   public static void traverse()  // traverse from first to last node
   {
      Node p;
      p = head;           // assign p the pointer to 1st node on list
      while (p != null)   // use p, not head, so head not corrupted
      {
         System.out.println(p.x); // display x in node p points to
         p = p.link;              // move p to next node on list
      }
   }
}
```

```
class SimpleList3
{
   static Node head = null; // static so addNode/traverse can access
   public static void main(String[] args)
   {
      addNode(1);      // add node with 1 to the beginning of list
      addNode(2);      // add node with 2 to the beginning of list
      addNode(3);      // add node with 3 to the beginning of list
      rtraverse(head); // last-to-first traversal
   }
   public static void addNode(int value)
   {
      Node p = new Node(); // create new node
      p.link = head;       // make new node point to current 1st node
      p.x = value;         // initialize x field in this node
      head = p;            // make head point to new node
   }
   public static void rtraverse(Node p)  // traverse from last to 1st
   {
      if (p != null)
      {
         rtraverse(p.link);        // display tail in reverse order
         System.out.println(p.x);  // display data in 1st node
      }
   }
}
```

A linked list class

```
class TestMyLinkedList
{
   public static void main(String[] args)
   {
      MyLinkedList list = new MyLinkedList();
      list.addNode(1);  // add node with 1
      list.addNode(2);  // add node with 2
      list.addNode(3);  // add node with 3
      list.traverse();  // traverse in first node to last
   }
}
class MyLinkedList
{
   private class Node     // Node definition, inner class
   {
      private Node link;  // points to next node or is null if at last
      private int x;      // data
   }
   private Node head = null;  // set head to null so list initially empty
   public void addNode(int value)
   {
      Node p = new Node();   // create new node
      p.link = head;         // make new node point to first node
      p.x = value;           // initialize x field in new node
      head = p;              // set head to point to new node
   }
   public void traverse()          // traverse in natural order
   {
      Node p = head;               // initialize p to pointto first node
      while (p != null)            // execute loop until p goes null
      {
         System.out.println(p.x);  // display x field in node p points to
         p = p.link;               // move p to next node on list
      }
   }
}
```

Problem:

To change iterative `traverse` to a recursive method requires that `traverse` be passed head. But `head` is private (as it should be).

Solution:

Have two `traverse` methods in `MyLinkedList`. One should be public and have no parameters. The other should be private and have one parameter. Then to invoke `traverse` from outside `MyLinkedList`, call the public `traverse`, passing it no arguments. Then this public `traverse` method should call the private `traverse` method, passing it `head`. Thus, `head` remains private, but `traverse` can be called from outside `MyLinkedList`. The public `traverse` method is called an **adapter method.**

MyLinkedList (with recursive traverse)

```
        .
        .
        .

// public version
public void traverse()
{
   traverse(head);// head private
}
//-----------------------------
// private version
private void traverse(Node p)
{
  // recursive implementation
}
```

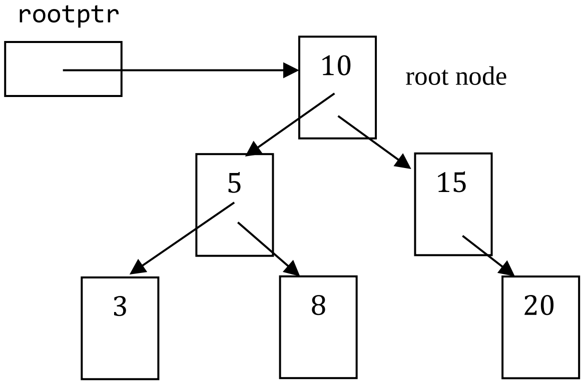
TestMyLinkedList

main

```
MyLinkedListlist list
 = new MyLinkedList();
      ...
      ...
 list.traverse();
      ...
```

Trees (drawn upside down)

```
private class Node
{
   private int x;
   private Node left;
   private Node right;
}
```



root node

Links not shown all `null`

Leaf nodes

Preorder:	10, 5, 3, 8, 15, 20
Inorder:	3, 5, 8, 10, 15, 20
Postorder:	3, 8, 5, 20, 15, 10

```
private void inorder(Node p)
{
   if (p != null)
   {
      inorder(p.left);
      System.out.println(p.x);
      inorder(p.right);
   }
}


public void inorder()     // adapter method
{
   inorder(rootptr);
}
```

```
class C15h21
{
   public static void main(String[] args)
   {
      MyTree t = new MyTree();

      t.insert(5);
      t.insert(10);
      t.insert(2);
      t.insert(9);
      t.insert(3);
      t.insert(1);
      t.inorder();
   }
}
```

```
//====================================================
class MyTree
{
   private class Node
   {
      private int x;
      private Node left;
      private Node right;
   }
   //---------------------------------------
   private Node rootptr = null;
```

```
public void insert(int xx)     // creates binary search tree
 {
    Node n = new Node();
    n.x = xx;
    if (rootptr == null)
       rootptr = n;
    else
    {
       Node trailing = null, leading = rootptr;
       while (leading != null)
       {
          trailing = leading;
          if (xx < leading.x)
             leading = leading.left;
          else
             leading = leading.right;
       }
       if (xx < trailing.x)
          trailing.left = n;
       else
          trailing.right = n;
     }
 }
```

```
  //---------------------------------------
   public void inorder()
   {
      inorder(rootptr);
   }
  //---------------------------------------
   private void inorder(Node p)
   {
      if (p != null)
      {
         inorder(p.left);
         System.out.println(p.x);
         inorder(p.right);
      }
   }
}
```