LINKED LISTS

Chapter 1

OBJECTIVES

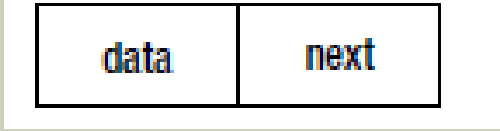
- The Classes NodeData and Node
- 1.2 Building a linked list (Tail)
- 1.3 Insertion
- 1.4 Building a linked list (Head)
- 1.5 Deletion
- 1.6 Sorted lists

  How to build a sorted linked list

CLASS NODEDATA

```
public class NodeData {

    int value;  // int can be replaced by any type
    public NodeData(int v){
        value = v;
    }
    public int compareTo(NodeData nd){
        if (value == nd.value)
            return 0;
        if (value < nd.value)
            return -1;
        return 1;
    }
}
```

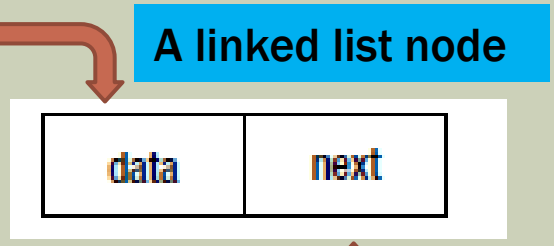
data is of type NodeData



A node contains two fields: data of type NodeData and next of type Node

CLASS NODE

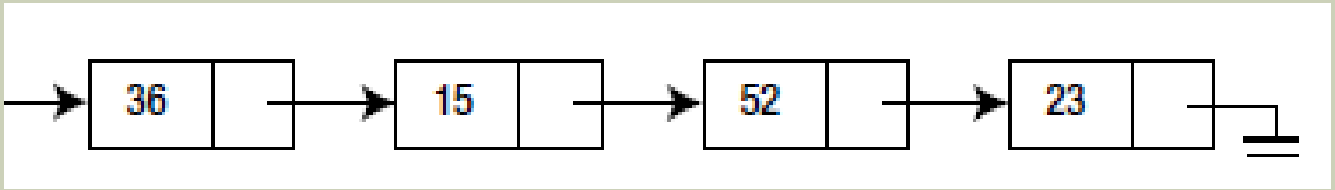
```
public class Node {

    NodeData data;
    Node next;


    public Node(NodeData nd) {
        data = nd;
        next = null;
    }
}
```



1.2 BUILDING A LINKED LIST (TAIL)

- Consider the problem of building a linked list of positive integers in the order in which they are given. Say the incoming numbers are as follows (0 terminates the data): 36 15 52 23 0

- We want to build the following linked list:

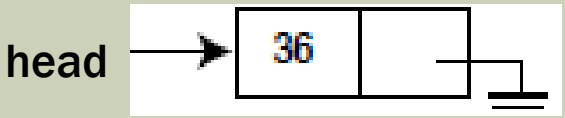


- whenever a new node must be added to the list, storage is allocated for the node, and the appropriate links are set.

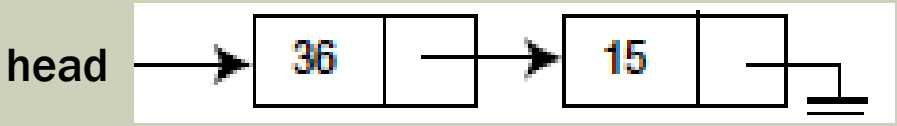
1.2 BUILDING A LINKED LIST (TAIL)

- We start with an empty list (head = null).
- When we read a new number, we must do the following:
  - Create a node containing the number using something like:

    Node newNode = new Node(new NodeData(36));
  - Make the new node the last one in the list
    - If this the first number inserted then the new node will be the head node.



    - For each subsequent number, we must set the next field of the current last node to point to the new node. The new node becomes the last node.

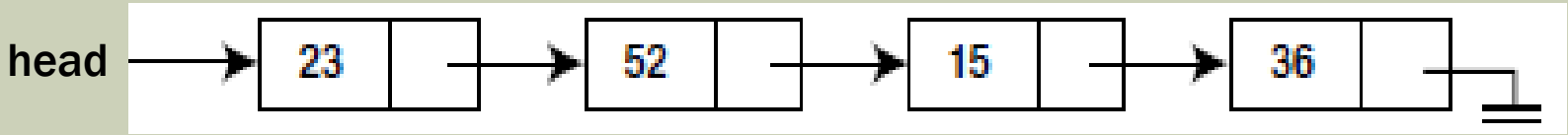


1.2 BUILDING A LINKED LIST (TAIL)

```
// This method adds a node at the end of the list

public void addTail(NodeData item){
    Node newNode = new Node(item);
    if (head == null) // if the list is empty
        head = newNode; //  the new node is the head node
    else
    {
        Node curr = head; // start from the first node
        while (curr.next != null) // go till the last node
            curr = curr.next; // move to the next node
        curr.next = newNode; // set the next field of the current
                             // last node to point to the new node
    }
} // end addTail
```

1.4 BUILDING A LINKED LIST (HEAD)

- Consider again the problem of building a linked list of positive integers but, this time, we insert each new number at the head of the list rather than at the end. The resulting list will have the numbers in reverse order to how they are given. Suppose the incoming numbers are as follows (0 terminates the data): 36 15 52 23 0
- We would like to build the following linked list:



- The program to build the list in reverse order is actually simpler than the previous one. As each new number is read, we set its link to point to the first node, and we set top to point to the new node, making it the (new) first node.

1.4 BUILDING A LINKED LIST (HEAD)

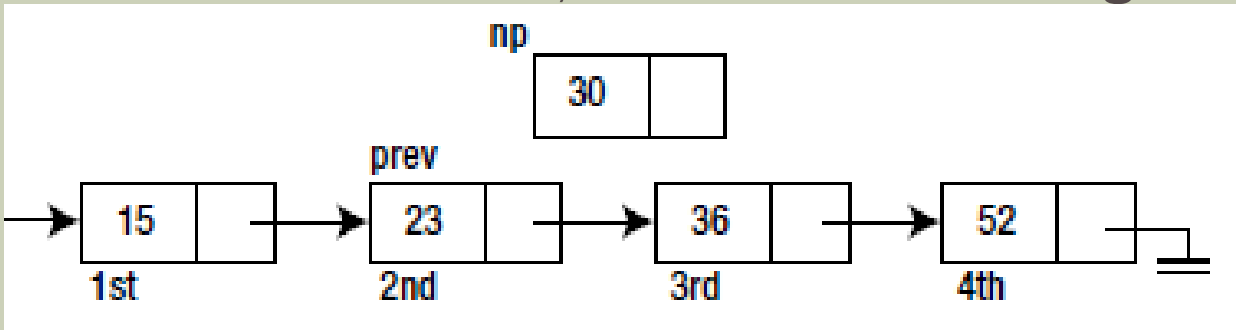
```
// This method adds a node at the beginning of the list

public void addHead(NodeData item) {
    Node newNode = new Node(item);
    if (head == null) // if the list is empty
        head = newNode; //  the new node is the head node
    else
    {
        newNode.next = head; //set the next of the new node to
                                    // point to first node
        head = newNode; //update head to point to new node
    }

} // end addHead
```

1.3 INSERTION

- One important characteristic of a linked list is that access to the nodes is via the “head” reference and the next field in each node. This means that access is restricted to being sequential.
- The only way to get to node 4, say, is via nodes 1, 2, and 3. The great advantage of a linked list is that it allows for easy insertions and deletions anywhere in the list.
- Suppose we want to insert a new node between the second and third nodes. We can view this simply as insertion after the second node.

1.3 INSERTION

- For example, suppose **prev** points to the second node and **np** points to the new node, as shown in the figure below.

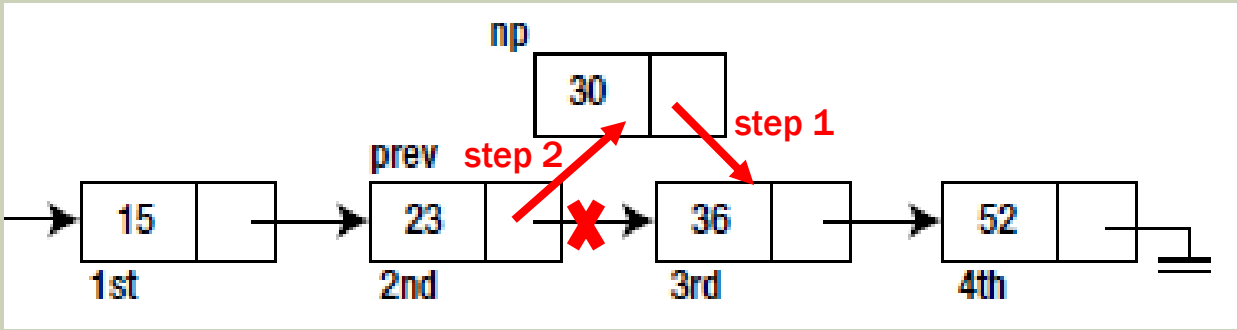


- We can insert the new node by setting its next field to point to the third node and the next field of the second node to point to the new node. Note that all we need to do the insertion is the second node; its next field will give us the third node. The insertion can be done with this:

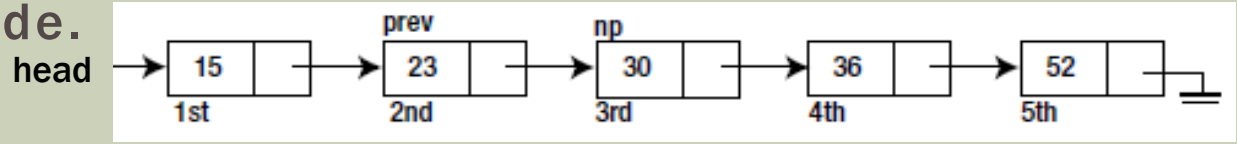
```
np.next= prev.next;
prev.next = np;
```

1.3 INSERTION

```
np.next = prev.next; // step 1
prev.next = np; // step 2
```



- The first statement says, “Let the new node (np) point to whatever the second node (prev) is pointing at, in other words, the third node.” The second statement says, “Let the second node point to the new node.” The net effect is that the new node is inserted between the second and the third. The new node becomes the third node, and the original third node becomes the fourth node.



1.3 INSERTION

```
// This method inserts a node at a given position in the list
public void insert(NodeData item, int position) {
   Node newNode, // to refer to the new node that will contain value
        pred; // to refer to the predecessor of the node at position
   if (position <= 0 || position > countNodes()+1)
        System.out.println("Invalid position. Insert failed.");
   else if (head == null) // Inserting into an empty list
        head = new Node(item);
   else if(position == 1) // Inserting at the beginning
        addHead(item); // Add the item at the head
   else // Inserting in the middle or at the end
   {
        newNode = new Node(item);
        pred = head;
        for (int i=1; i <= position - 2; i++)
             pred = pred.next;
        newNode.next = pred.next;
        pred.next = newNode;
   } //end else
} // end insert
```

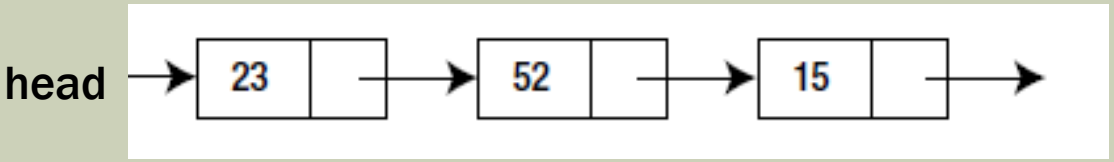
newNode is the same as np in the previous two slides and pred is the same as prev

1.5 DELETION

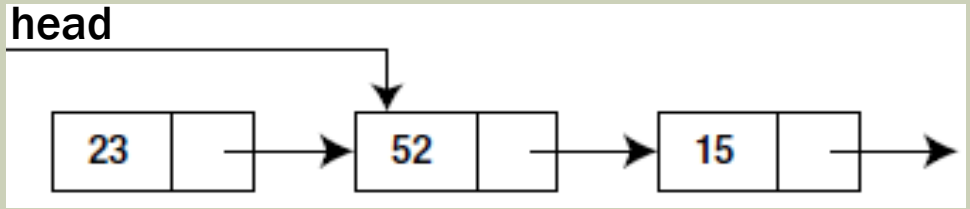
- Deleting a node from the beginning (head) of a linked list is accomplished by this statement:

  head = head.next;

- This says let head refer to whatever the first node was pointing at (that is, the second node, if any). Since head is now pointing at the second node, effectively the first node has been deleted from the list. The statement changes the following:



to this:



1.5 DELETION

- Of course, before we delete, we should check that there *is* something to delete, in other words, that head is not null. If there is only one node in the list, deleting it will result in the empty list; head will become null.
- Deleting an arbitrary node from a linked list requires more information. Suppose **curr** (for "current") points to the node to be deleted. Deleting this node requires that we change the next field of the *previous* node. This means we must know the reference (pointer) to the previous node; suppose it is **prev** (for "previous"). Then deletion of node **curr** can be accomplished by this statement:

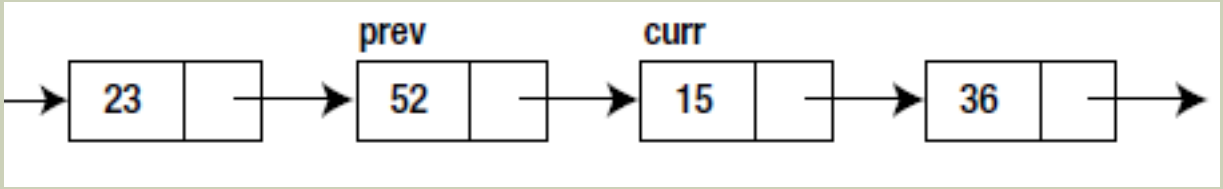
```
prev.next = curr.next;
```

1.5 DELETION

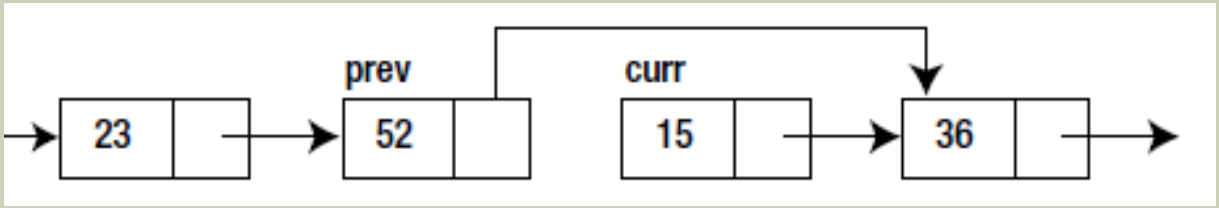
The statement

```
prev.next = curr.next;
```

changes the following:



to this:



Effectively, the node pointed to by curr is no longer in the list — it has been deleted.

1.5 DELETION

```
// This method deletes a node from a list at a given position
public void delete(int position) {
   Node pred, // to refer to the predecessor of the node at position
        curr; // to refer to the node that should be deleted
   if (head == null) // Attempting to delete from an empty list
        System.out.println("Empty list. Delete failed.");
   else if (position <= 0 || position > countNodes())
        System.out.println("Invalid position. Delete failed.");
   else if(position == 1) // Deleting at the beginning
        head = head.next; // set the head to the second node
   else // Deleting in the middle or at the end
   {
        pred = head;
        for (int i=1; i <= position - 2; i++)
             pred = pred.next;
        curr = pred.next;
        pred.next = curr.next;
    } //end else
} // end delete
```

pred is the same as prev in the previous two slides and

1.5 DELETION

Two other useful methods for deleting nodes:

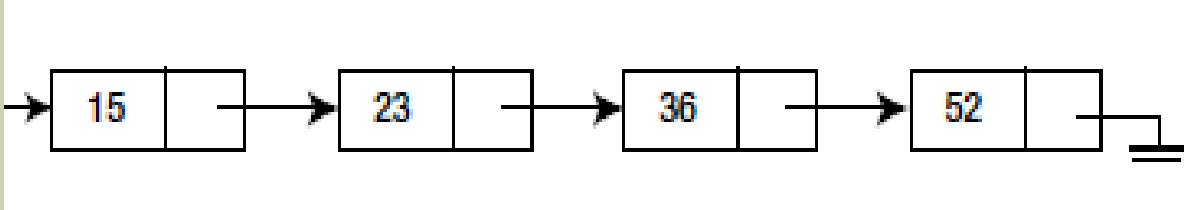
```
// This method deletes all nodes in the list
public void clear() {
   head = null; // set the head to null
} // end clear


// This method deletes the head node
public void deleteHead() {
   if (head != null) // if not empty
       head = head.next; // set the head to the 2nd node
} // end deleteHead
```

1.6 SORTED LISTS

Suppose we want to build the list so that the numbers are always sorted in ascending order.

Suppose the incoming numbers are :36 15 52 23

We would like to build the following linked list:

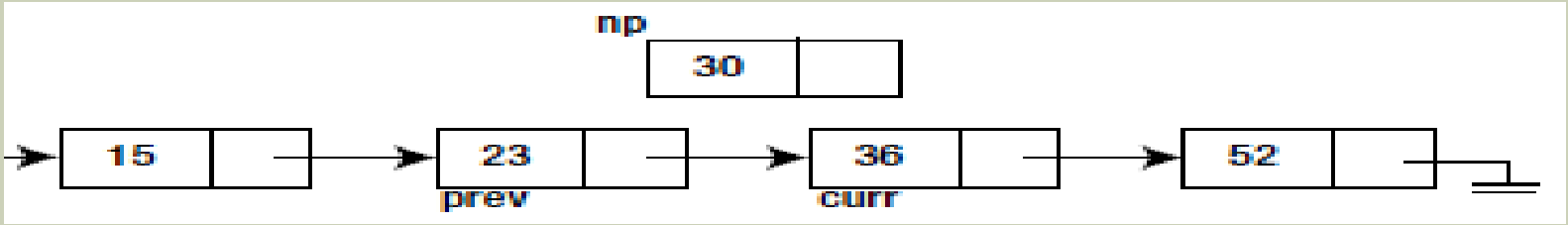


When a new number is read, it is inserted in the existing list (which is initially empty) in its proper place. The first number (i.e. 36) is simply added to the empty list.

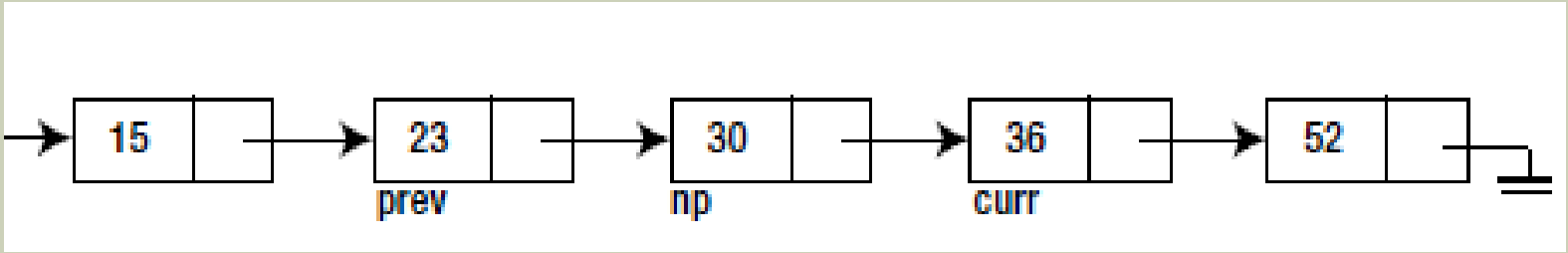
Each subsequent number is compared with the numbers in the existing list. As long as the new number is greater than a number in the list, we move down the list until the new number is smaller than, or equal to, an existing number or we come to the end of the list.

HOW A NEW NODE IS INSERTED INTO A SORTED LIST

Assume we want to insert 30 into the linked list in the previous list then it will be added before 36 since 36 > 30.



Hence the linked list will become



See the code in the next slide. The complete implementation of the LinkedList was emailed to you before and it will be also uploaded to Google Classroom too.

CODE

```
// This method adds an element (nd) to the linked list at the appropriate  place while keeping the list sorted
public void addInPlace(NodeData nd){
      Node newNode, prev = null, curr = head;
      newNode = new Node(nd);
// repeat as long as we did not reach the end of the list and we did  not reach a node with data greater than nd
      while (curr!= null && nd.compareTo(curr.data) > 0)  {
         prev = curr;  // prev is the node that precedes the current node
         curr = curr.next;  // move to the next node
      } // end while
      if (prev == null) { // inserting at the beginning
         head = newNode; // the head refer to the new node
         newNode.next = curr;
      }
      else { // prev != null - inserting in the middle or end
         newNode.next = curr;  // link the new node to the current node
         prev.next = newNode;  // link the previous node to the new node
      } // end else
} // end addInPlace
```