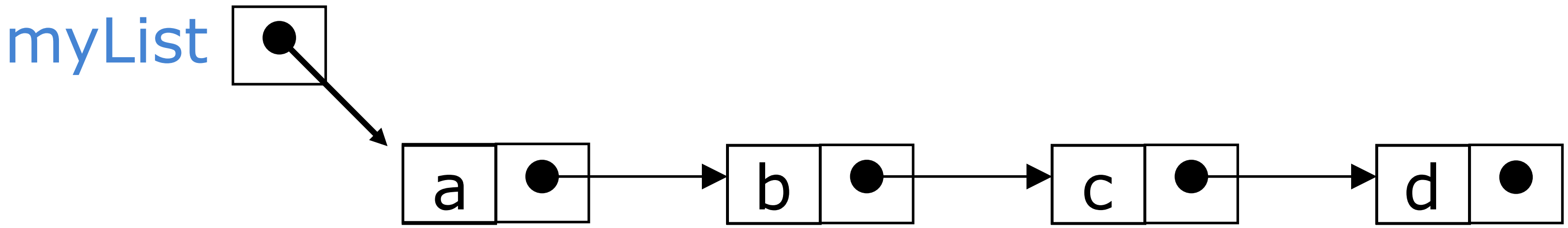
6 - LINKED LISTS

Topics

- Anatomy of Linked Lists
- Simple-Linked List (SLL)
- Double-Ended List
  - Stacks implementation
  - Queues implementation
- Doubly-Linked List (DLL)

Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes (links)



•Each node contains a value and a link (pointer or reference) to some other node

•The last node contains a null link

• The variable myList is called the header.

Link

- In a **linked list**, each data item is embedded in a node ***(link)***.
- A link is an object of a class called something like Link or Node.
- 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.

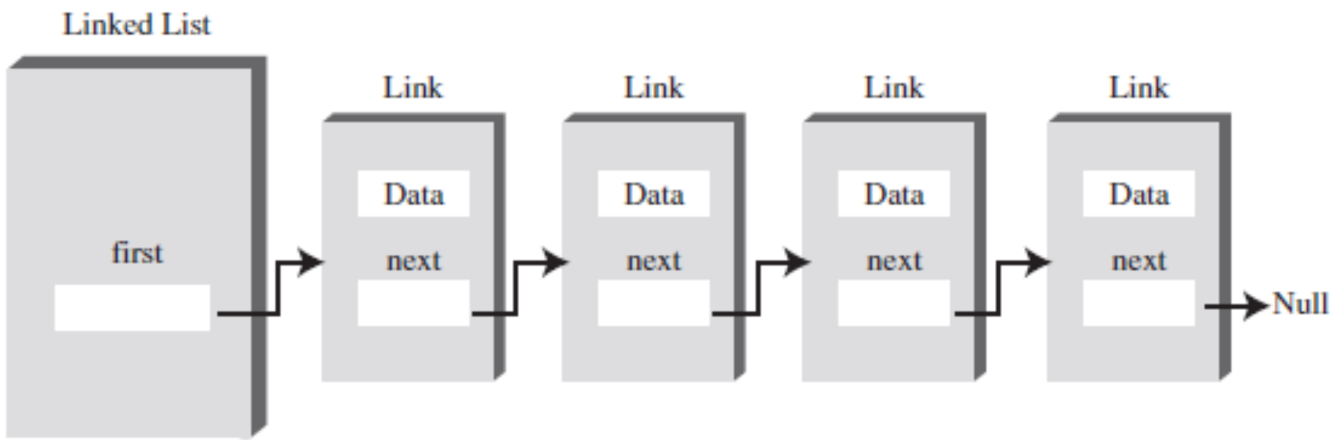


FIGURE 5.1 Links in a list.

Link

- Class Link (node): It contains some data and a reference to the next link:

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

// with objects as datat
class Link
{
    public InventoryItem iI; // object holding data
    public Link next; // reference to next link
}
```

Relationship, Not Position

- Linked lists differ from arrays.
- In an array each item occupies a particular position. This position can be directly accessed using an index number. It's like a row of houses: You can find a particular house using its address.
- In a list the only way to find a particular element is to follow along the chain of elements.

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements). In this case the header is null.

Linked List - illustrated

- See LinkList workshop applet:
  - Insert
    - Insert at the front is the simplest (unsorted)
    - you can insert anywhere (sorted)
    - No limited size like arrays => No overflow
  - Search (sequential)
  - Delete (Manipulation of links – let previous node references the next node)

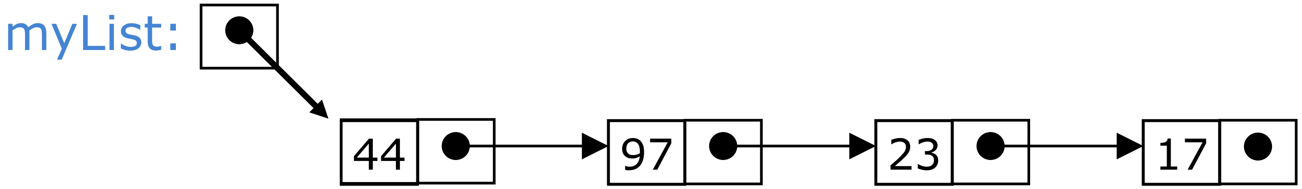
Pointers and references

- In C and C++ we have "pointers," while in Java we have "references"
  - These are essentially the same thing
    - The difference is that C and C++ allow you to modify pointers in arbitrary ways, and to point to anything
  - In Java, a reference is more of a "black box," or ADT
    - Available operations are:
      - dereference ("follow")
      - copy
      - compare for equality
    - There are constraints on what kind of thing is referenced: for example, a reference to an array of int can *only* refer to an array of int

Creating references

- The keyword new creates a new object, but also returns a *reference* to that object
- For example, Person p = new Person("John")
  - new Person("John") creates the object and returns a reference to it
  - We can assign this reference to p, or use it in other ways

Creating links in Java




```
class Cell {
    Object  value;
    Cell next;

     Cell (Object v, Cell n) { // constructor
        value = v;
        next = n;
     }
}
```

- The following instruction construct the list starting from the last element

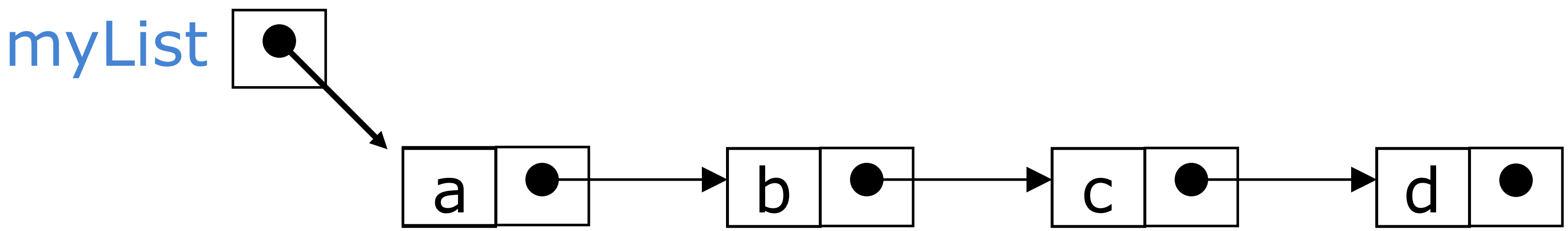
```
Cell temp = new Cell(17, null);
temp = new Cell(23, temp);
temp = new Cell(97, temp);
Cell myList = new Cell(44, temp);
```

- We can also do the following:

```
Cell myList = new Cell(44, new Cell(97, new Cell(23, new Cell(17, null))));
```

Simple-linked lists (SLL)

- Here is a simple-linked list or singly-linked list (SLL):

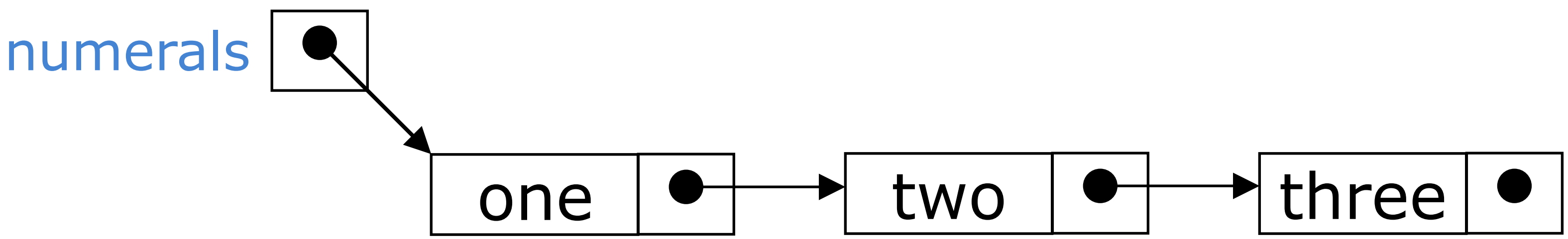


- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)

Creating a simple list

- To create the list ("one", "two", "three"):
- Cell numerals;
- numerals =

```
      new Cell("one",
          new Cell("two",
              new Cell("three", null)));
```



SLL Java Code Implementation

- First Example (see Listing 5.1, linkList.java, page 190) – simple operations:
  - Inserting an item at the beginning of the list ---- (insertFirst())
  - Deleting the item at the beginning of the list ----- (deleteFirst())
  - Iterating through the list to display its contents ----- (displayList())

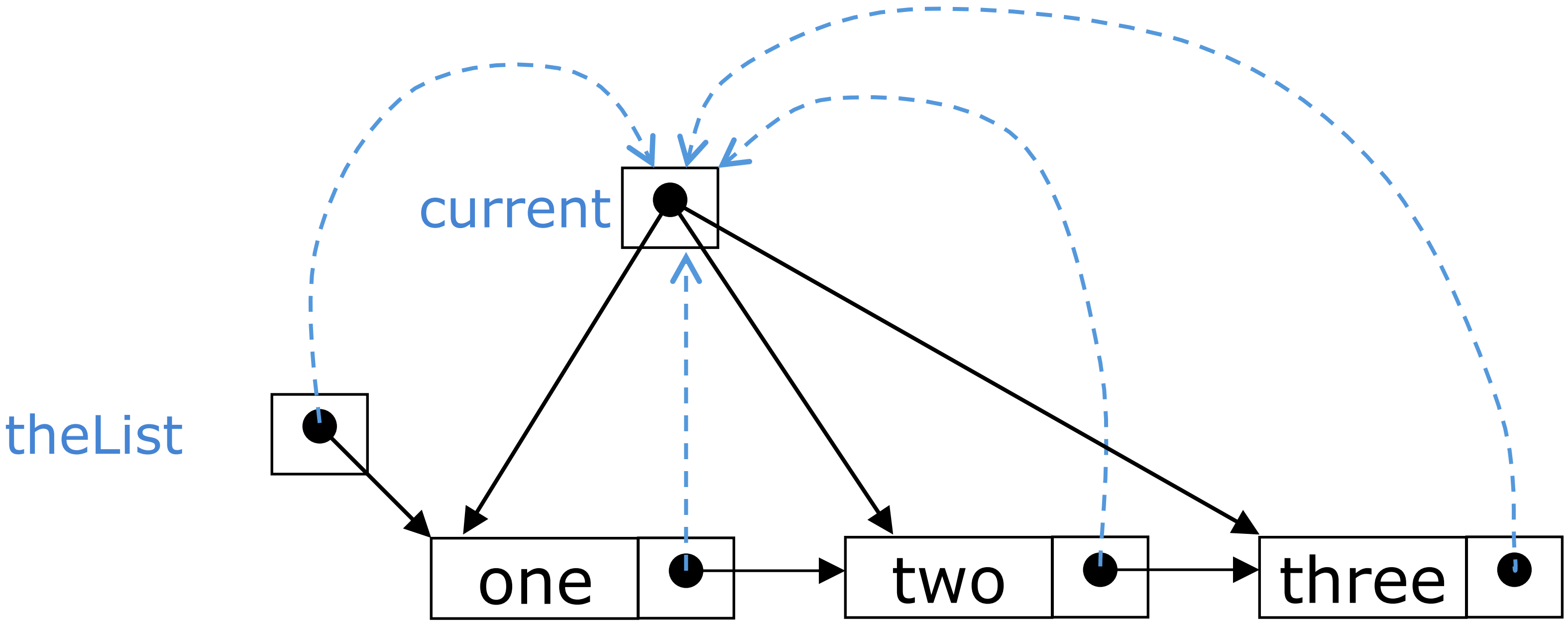
class Link

```
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)
  // -------------------------------------------------------------
  public void displayLink() // display ourself
  { … }
} // end class Link
```

class LinkList

```
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);
  }
  // -------------------------------------------------------------
  // ... other methods go here
}
```

Traversing a SLL (animation)



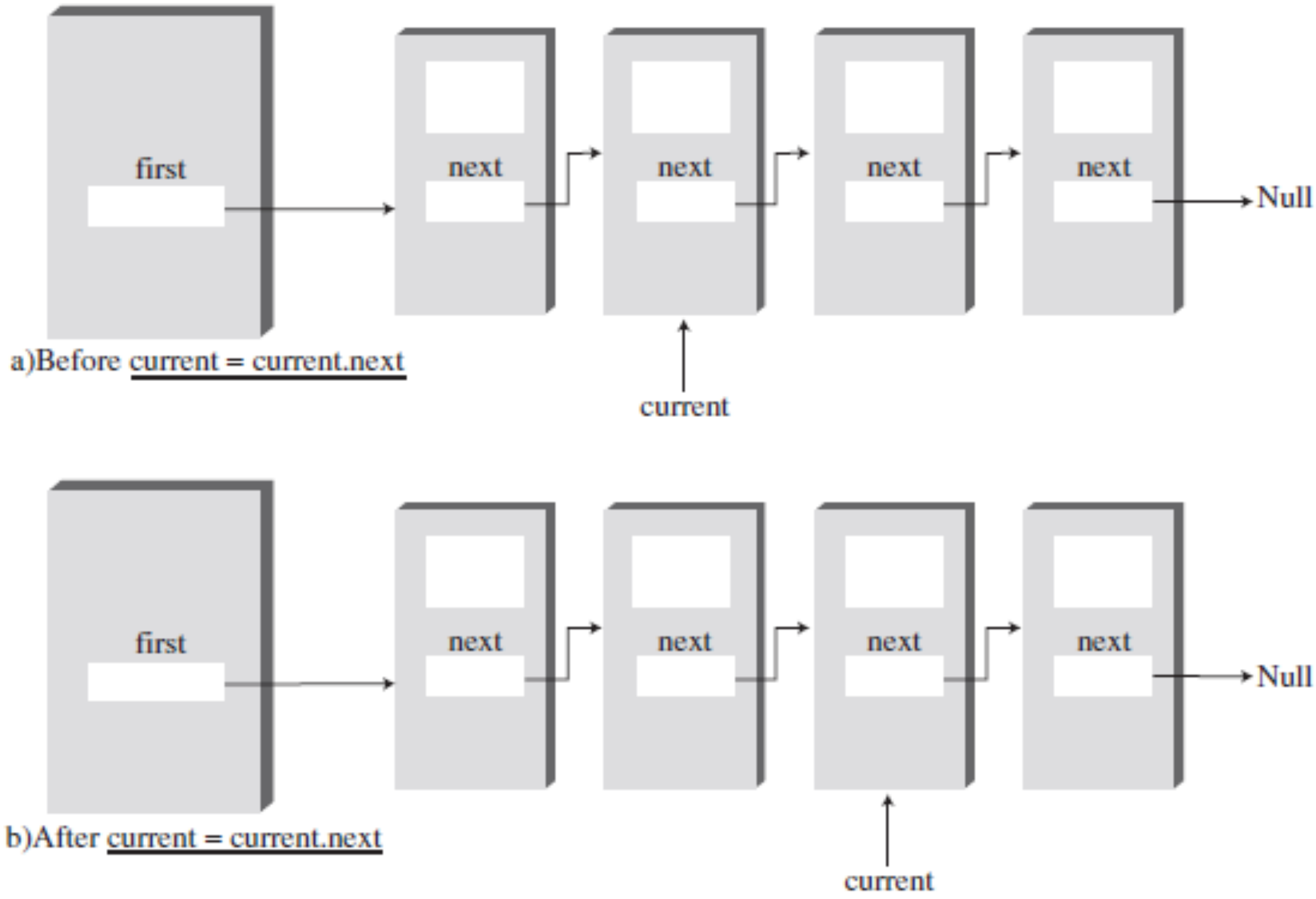


FIGURE 5.7 Stepping along the list.

Traversing a SLL

- The following method traverses a list (and prints its elements):

```
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("");
}
```

Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting as a new first element

- This is probably the easiest method to implement
- In class LinkList:

```
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
    }
```

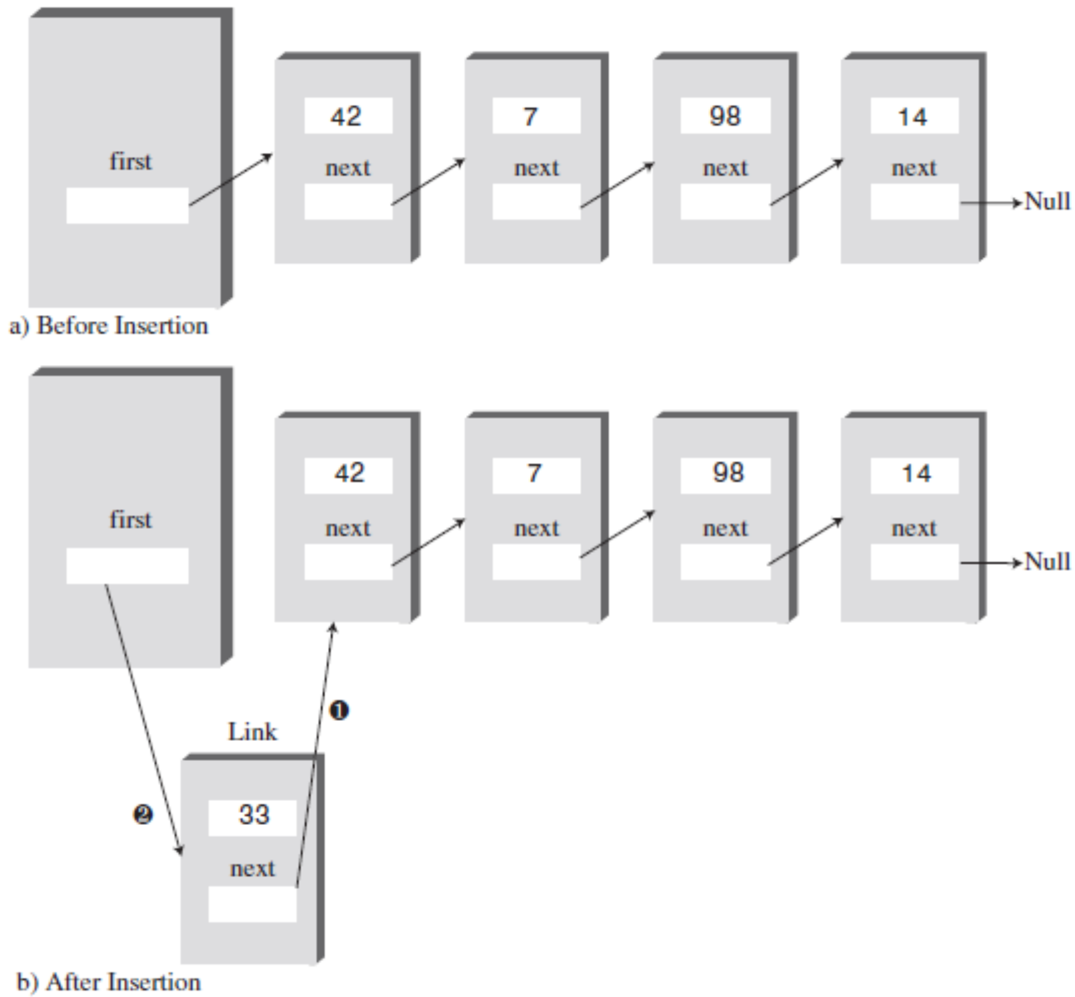
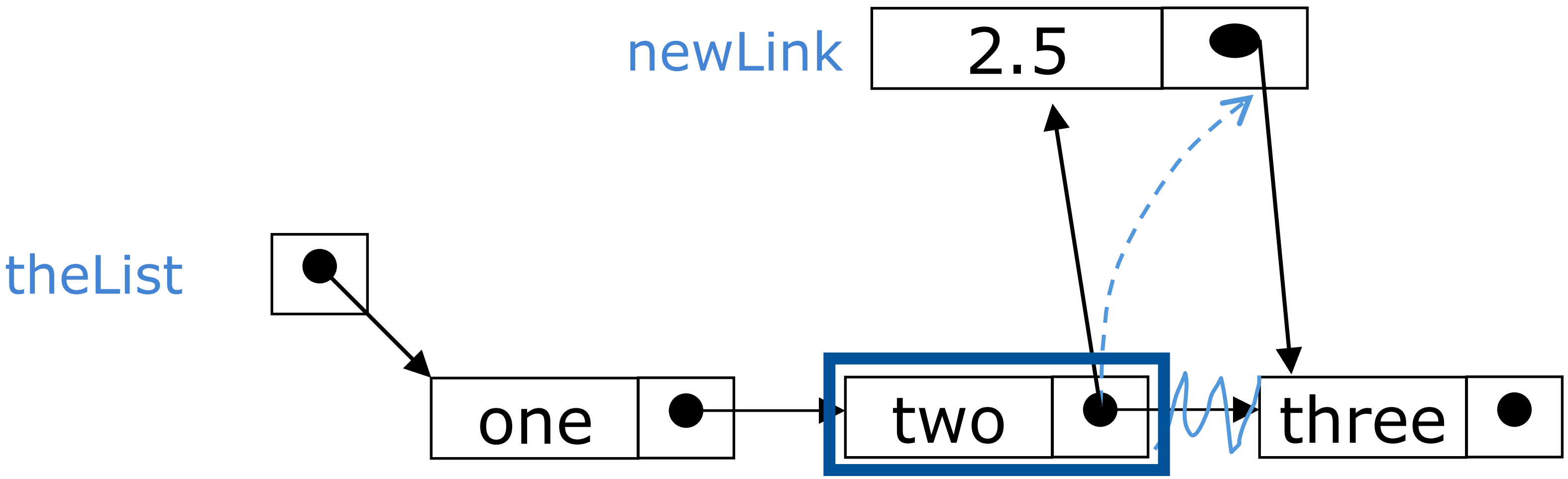


FIGURE 5.5 Inserting a new link.

Inserting after a given value (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

Inserting a node after a given value

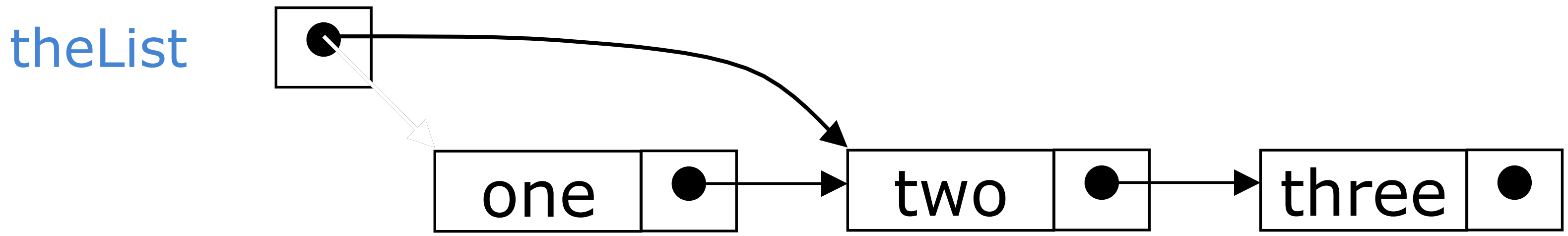
```
void insertAfter(int id, double dd, int value) {
   for (Link current= first; current!= null; current =
   current.next) {
        if (current.iData == value) {
            Link newLink= new Link(id, dd);
            newLink.next = current.next;
            current.next = newLink;
            return;
        }
   }
}
```

Deleting a node from a SLL

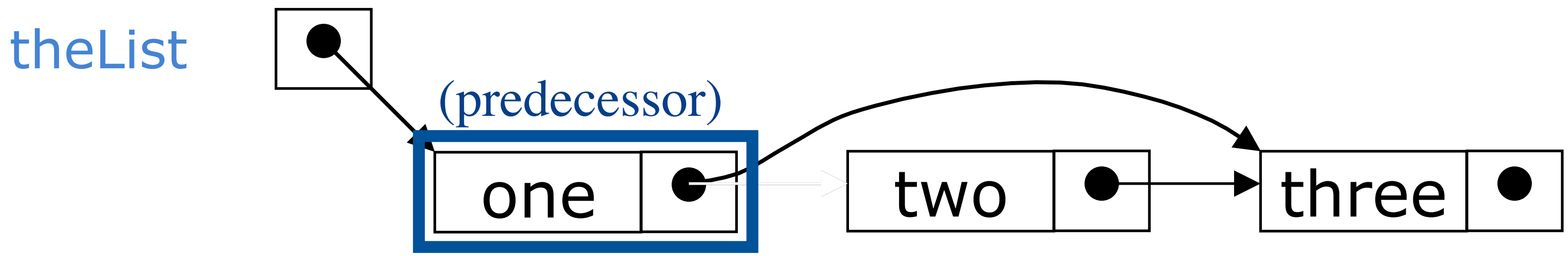
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected

Deleting first node from a SLL (simple)

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

Finding and Deleting a specified node from a SLL

- Listing 5.2, linkList2.java, page 193

```
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
    }
```

Finding and Deleting a specified node from a SLL

```
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;
    }
```

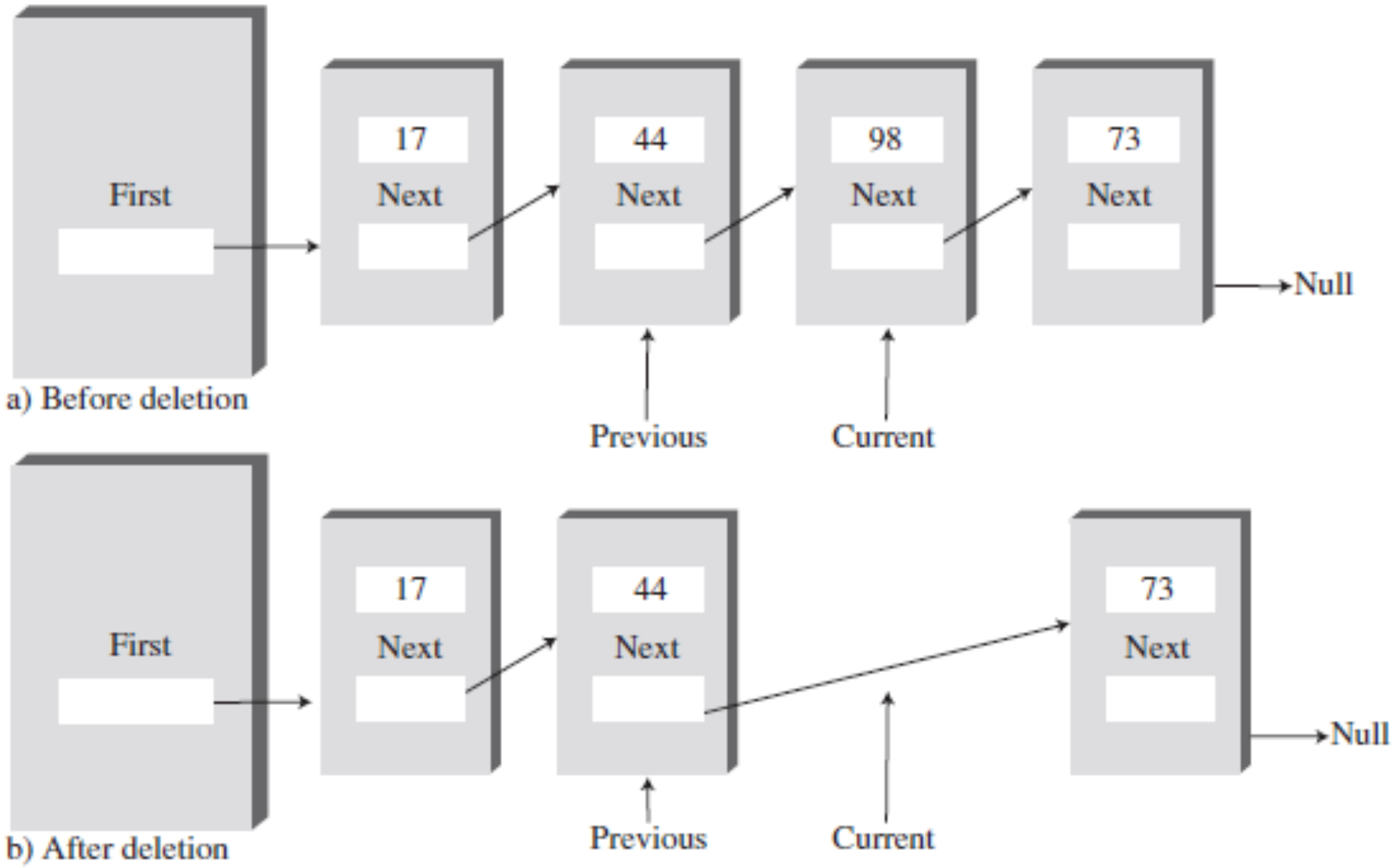


FIGURE 5.8 Deleting a specified link.

Double-Ended list

- A double-ended list is similar to an ordinary linked list, but it has one additional feature: a reference to the last link as well as to the first.
- Double-ended list suitable for certain situations that a single-ended list can't handle efficiently, like implementing a queue. (insertLast method)
- See Listing 5.3, firstLastList.java, page 198.

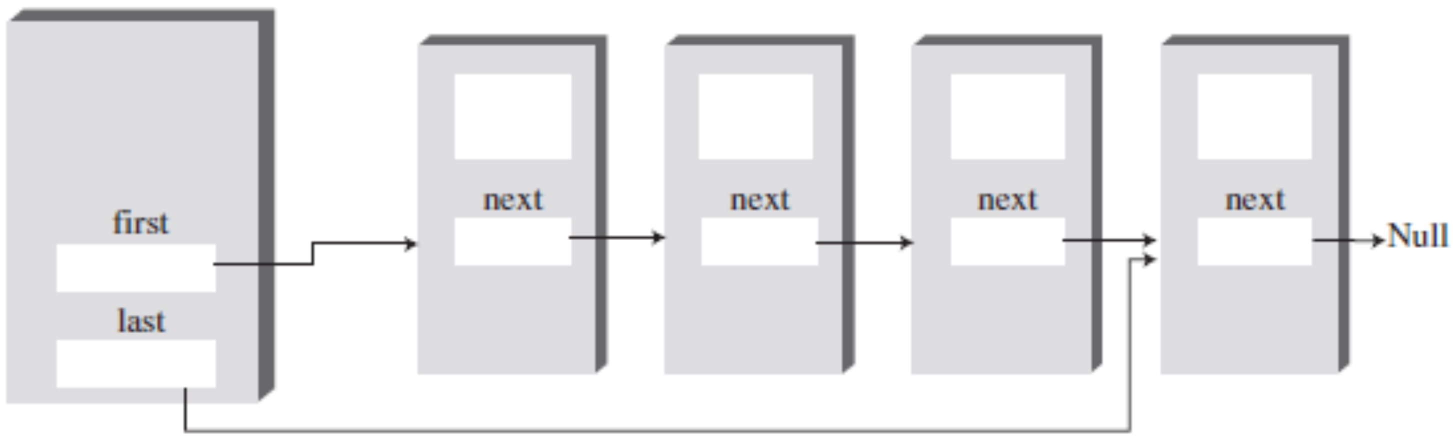


FIGURE 5.9 A double-ended list.

Double-Ended list

```
// insert at end of list
public void insertLast(long dd) {
  Link newLink = new Link(dd); // make new link
  if( isEmpty() )                 // if empty list,
    first = newLink;              // first --> newLink
  else
    last.next = newLink;      // old last --> newLink
  last = newLink;             // newLink <-- last
}
```

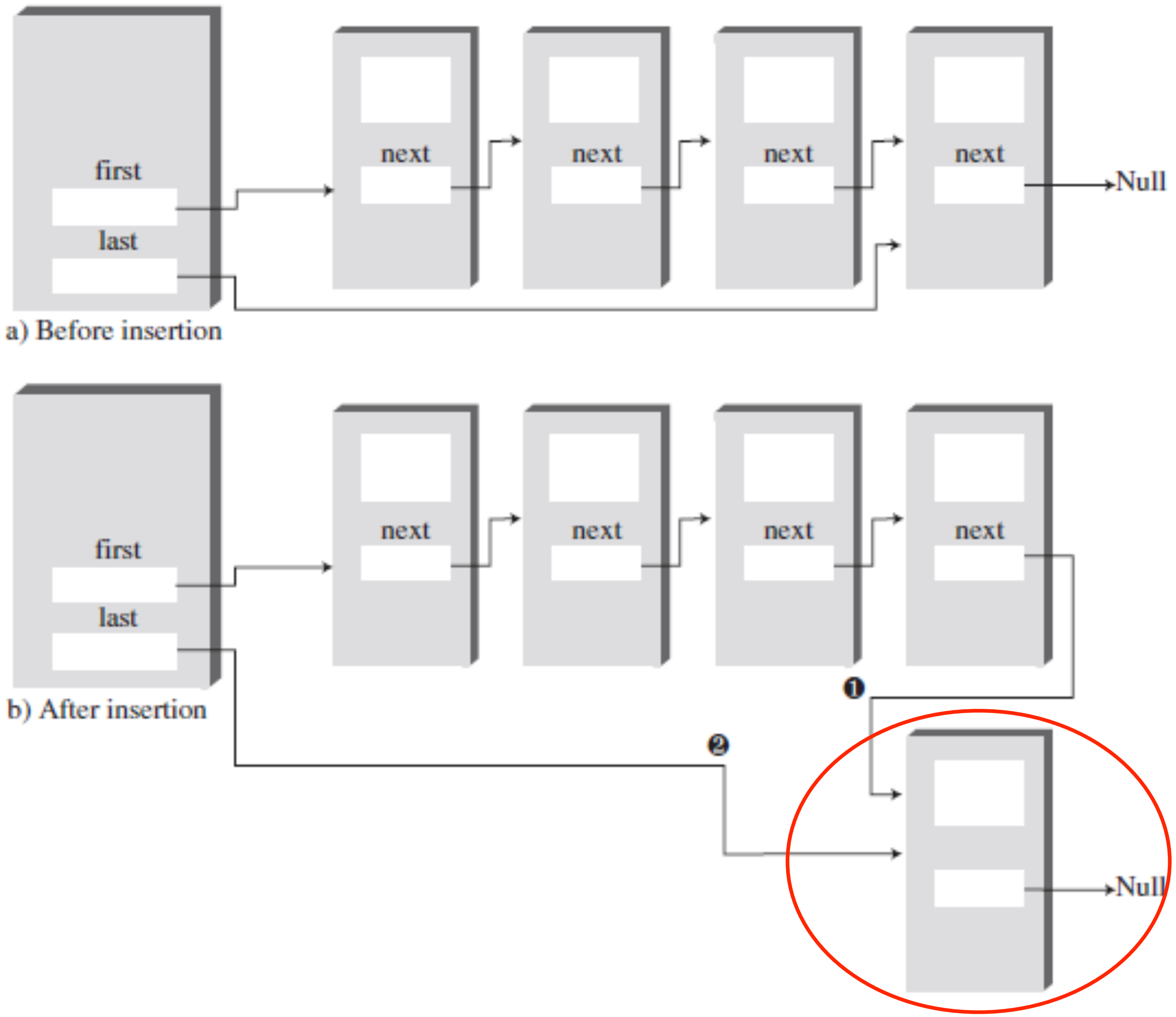


FIGURE 5.10 Insertion at the end of a list.

Linked List Efficiency

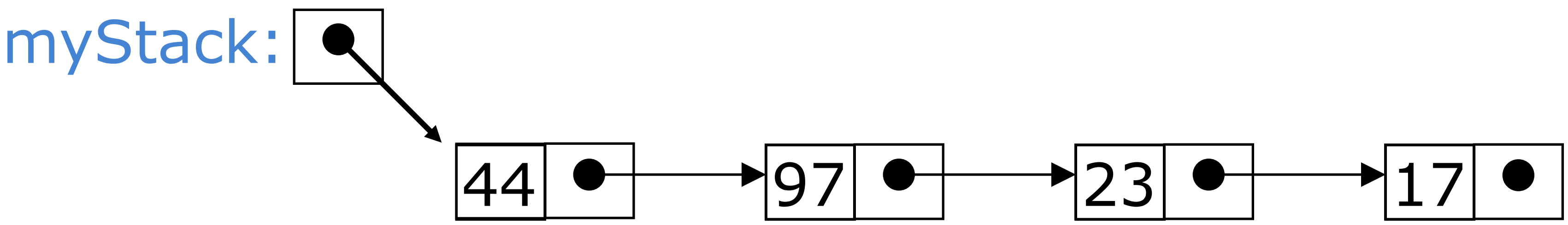
- Insertion and deletion at the beginning of a linked list are very fast. It takes O(1) time.
- Finding, deleting, or inserting next to a specific item requires searching through, on the average, half the items in the list. This requires O(N) comparisons.
- An array is also O(N) for these operations, but the linked list is faster as it does not need to move elements.
- Linked lists is better than arrays in that a linked list uses exactly as much memory as it needs and can expand to fill all of available memory. While size of array is fixed.

Abstract Data Types (ADT)

- It’s a way of looking at a data structure: focusing on what it does and ignoring how it does its job.
- Stacks and Queues are examples of ADTs.
  - Both can be implemented using Arrays or Linked Lists.

Linked-list implementation of stacks

- See Listing 5.4, linkStack.java, page 203
- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
  - theList.insertFirst(data)
- Popping is removing an element from the front of the list
  - data = theList.deleteFirst()

Linked-list implementation of Stacks

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to null
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate

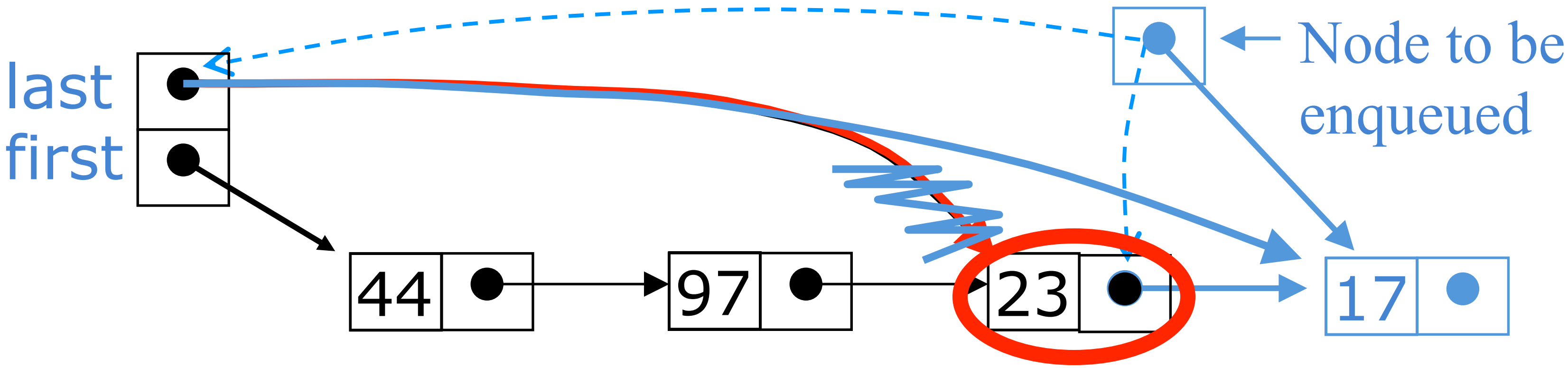
Linked-list implementation of Queues

- See Listing 5.5, linkQueue.java, page 207.
- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a double-ended list to implement both insertions and deletions in O(1) time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list
- Enque (insert) and Deque (remove) operations are implemented by the insertLast() and deleteFirst() methods

SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Enqueueing a node



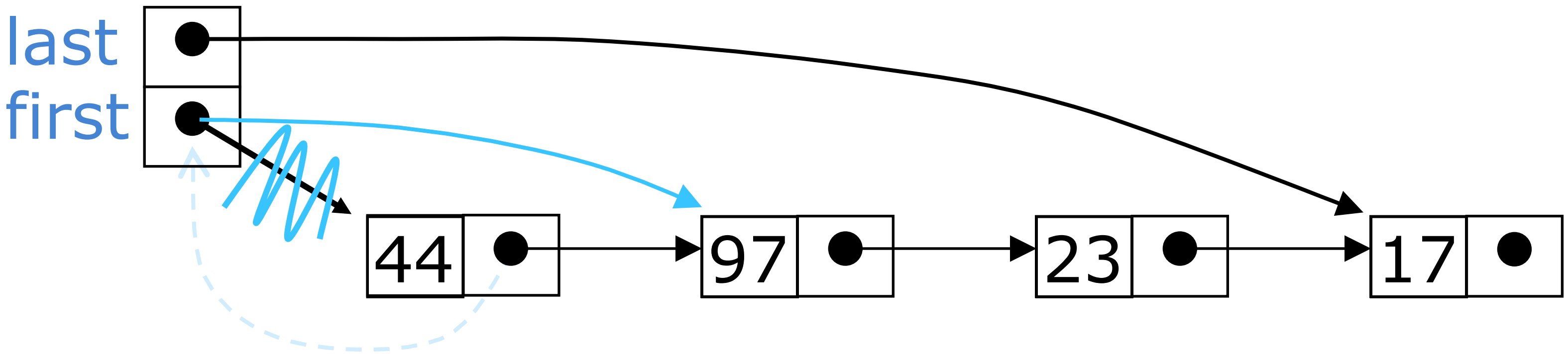
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the last pointer in the list header

Dequeueing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

Queues and Stacks implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to null
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to null

Doubly-linked lists

- Here is a doubly-linked list (DLL):

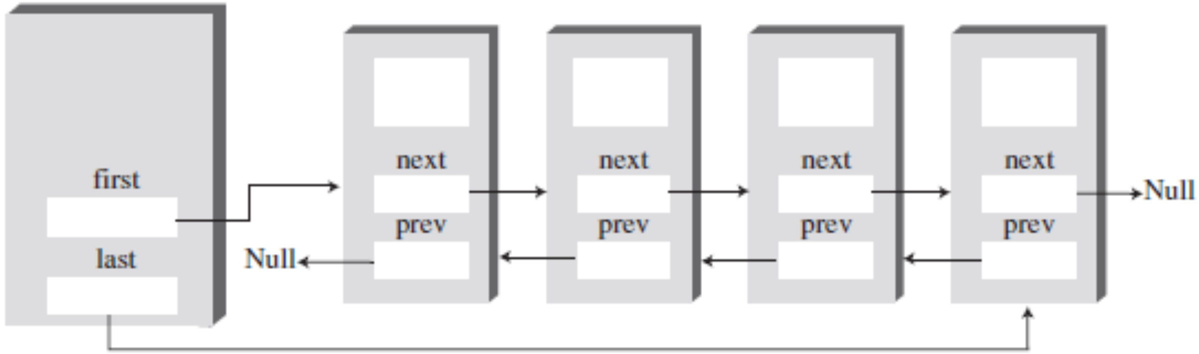


FIGURE 5.13 A doubly linked list.

  - Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
  - The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)
  - Can be traversed backward as well as forward.

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)

Java Code Implementation of DLL

- See Listing 5.8, doublyLinked.java, page 226
- Class of doubly linked list looks like this:

```
class Link
{
   public long dData; // data item
   public Link next; // next link in list
   public link previous; // previous link in list
   ...
}
```

Traversal of a DLL

- Can be traversed in either direction (may be essential for some programs)

- Backward Traversal:

```
Link current = last; // start at end
while(current != null) // until start of list,
      current = current.previous; // move to previous link
```

Insertion at the beginning in a DLL

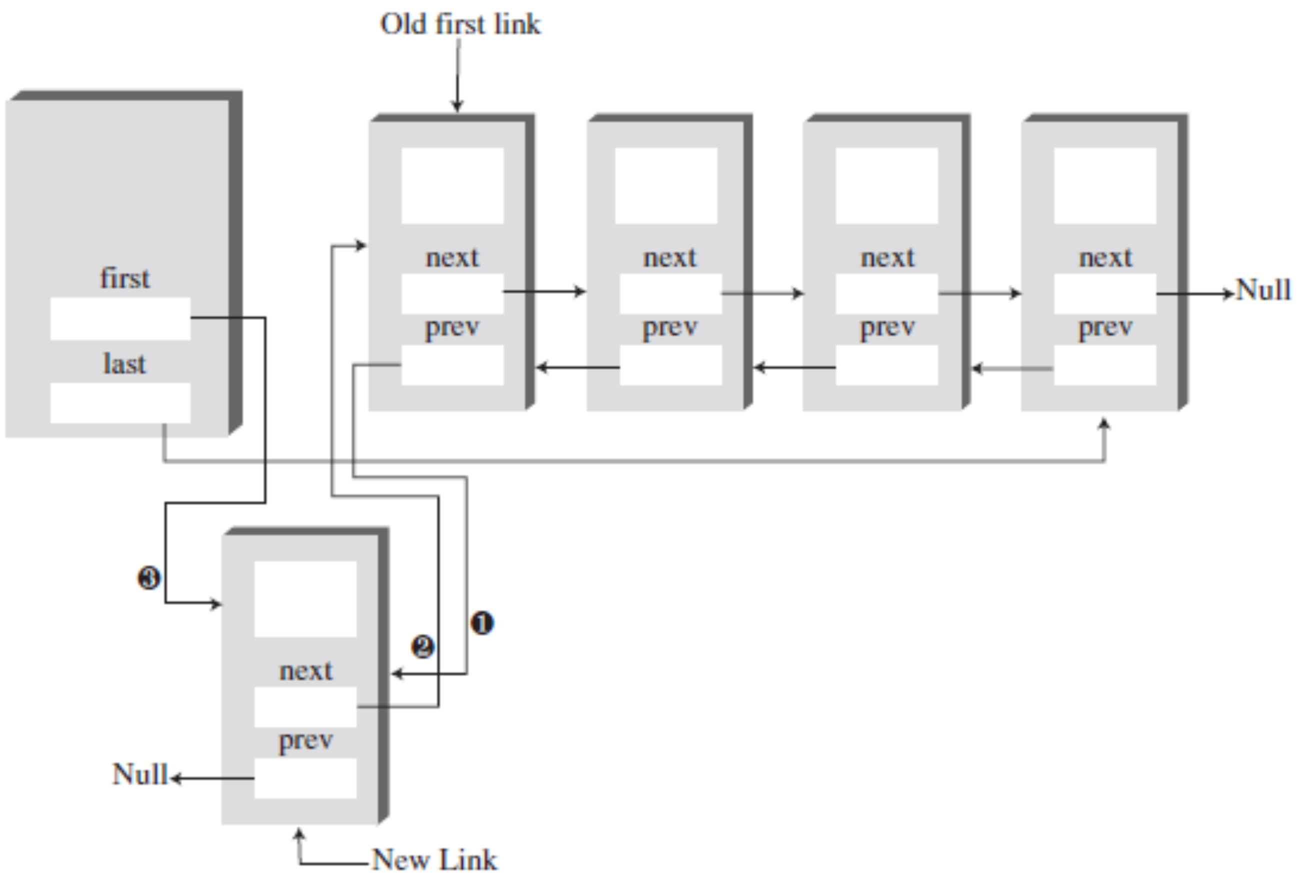


FIGURE 5.14 Insertion at the beginning.

Insertion at the beginning in a DLL

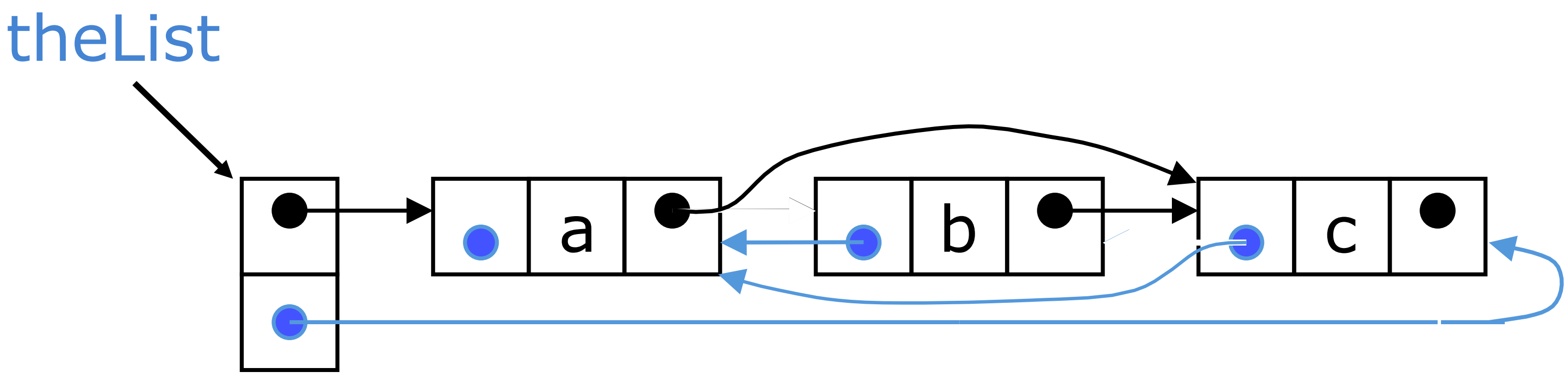
```
if( isEmpty() ) // if empty list,
   last = newLink; // newLink <-- last
else
   first.previous = newLink; // newLink <-- old first

newLink.next = first; // newLink --> old first

first = newLink; // first --> newLink
```

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links
- In this example, we will delete node b



- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

```java
public Link deleteKey(long key) // delete item w/ given key
{ // (assumes non-empty list)
   Link current = first; // start at beginning
   while(current.dData != key) // until match is found,
   {
      current = current.next; // move to next link
      if(current == null)
         return null; // didn't find it
   }
   if(current==first) // found it; first item?
      first = current.next; // first --> old next
   else // not first
      // old previous --> old next
      current.previous.next = current.next;
   if(current==last) // last item?
      last = current.previous; // old previous <-- last
   else // not last
      // old previous <-- old next
      current.next.previous = current.previous;
   return current; // return value
}
```

Other operations on linked lists

- Most “algorithms” on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can’t directly access the n^{th} element—you have to count your way through a lot of other elements

The End