Chapter 4

Objectives

- Arrays versus `ArrayLists`
- Linked list data structure as a concept
- Linked list implementation
- Exception handling
- Insertion sort

Arrays and `ArrayLists`

Arrays

- Random access structure
- Easy to subscript
- Dangerous and deprecated
- Can waste space
- Poor dynamic behavior
- Length returns preallocated length

`ArrayLists`

- Random access structure
- Easy to subscript
- Manages space itself
- Good dynamic behavior
- Length returns the space actually used

Why are arrays evil?
For example:

```
int myArray[100];
int mySubscript = 95;
initalizationLength = System.console.in();
for(int i = 0; i < initializationLength; ++i)
{
  myArray[i] = System.console.in();
}
int n = myArray[mySubscript];
```

Linked Lists (As a Concept)

- Not a random access structure
- Linear list data structure
- Dynamic allocation, de-allocation (at least to the user, when properly implemented)
- Performance advantage in alloc/dealloc over `ArrayList`
- The canonical concept of a *dynamic linear list*
- Often used to keep data in sorted order

Singly Linked List

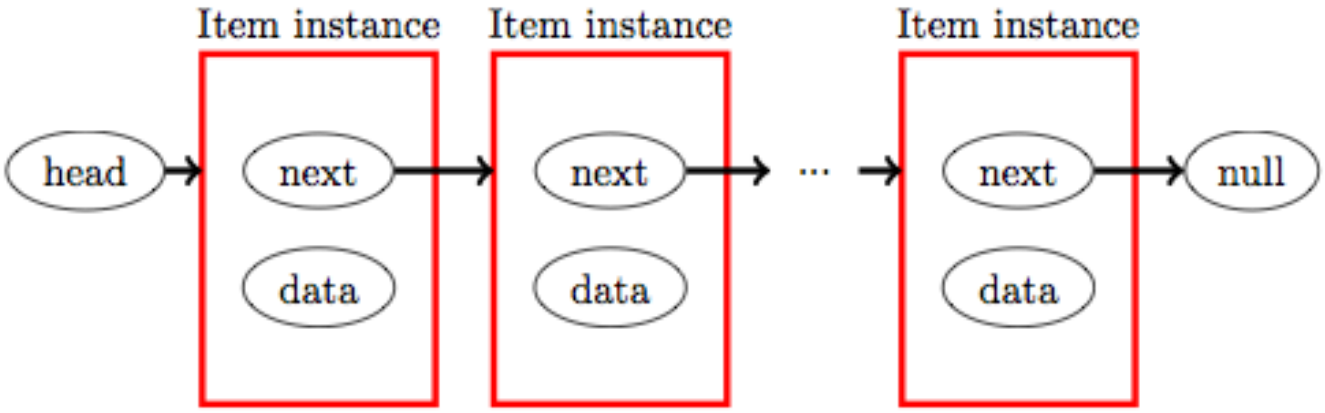

Figure 4.1 A linked list of "items"

Doubly Linked List

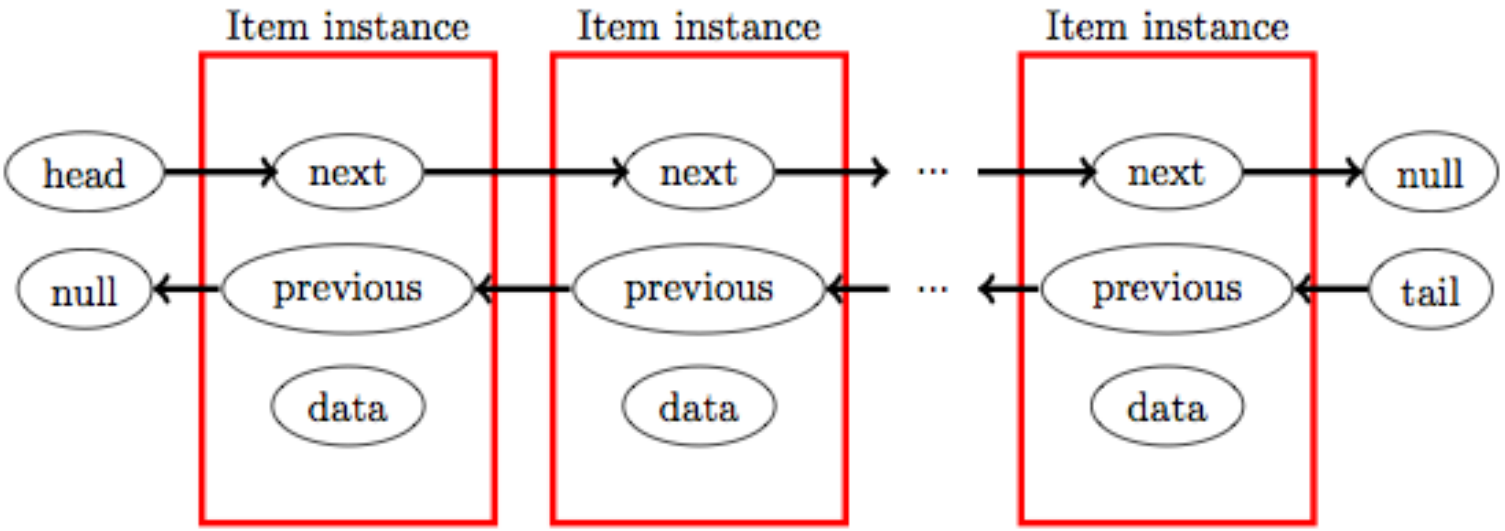

Figure 4.2 A doubly-linked list of "items"

Inserting a Node

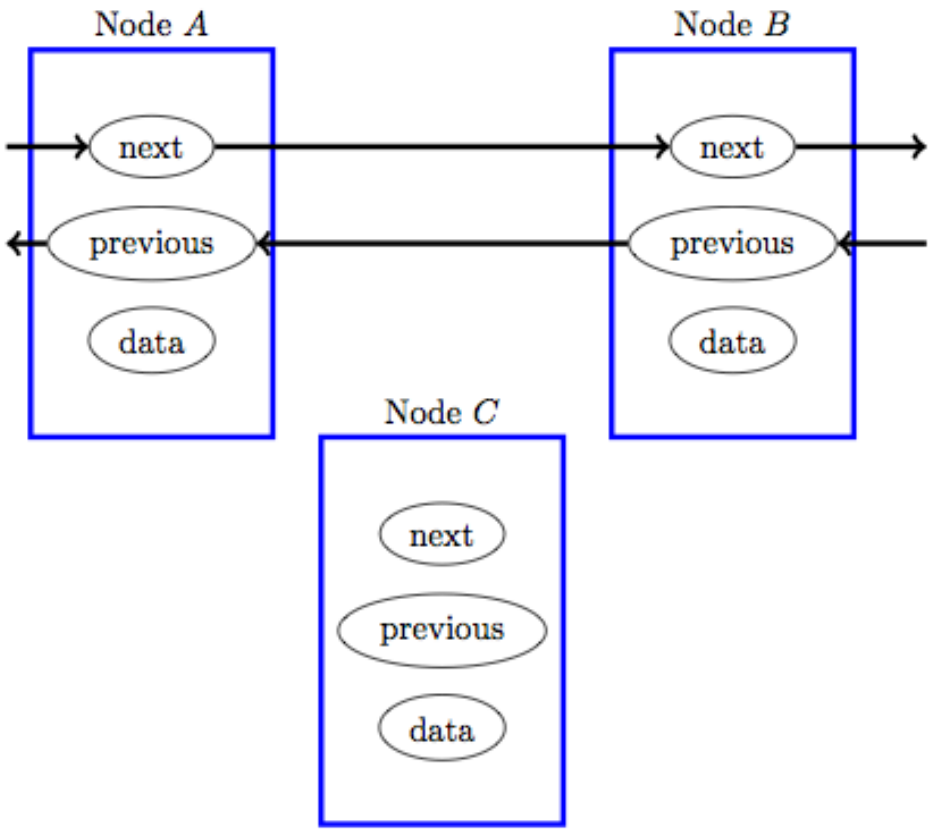

Figure 4.3 Inserting a node C after node A

Inserting a Node

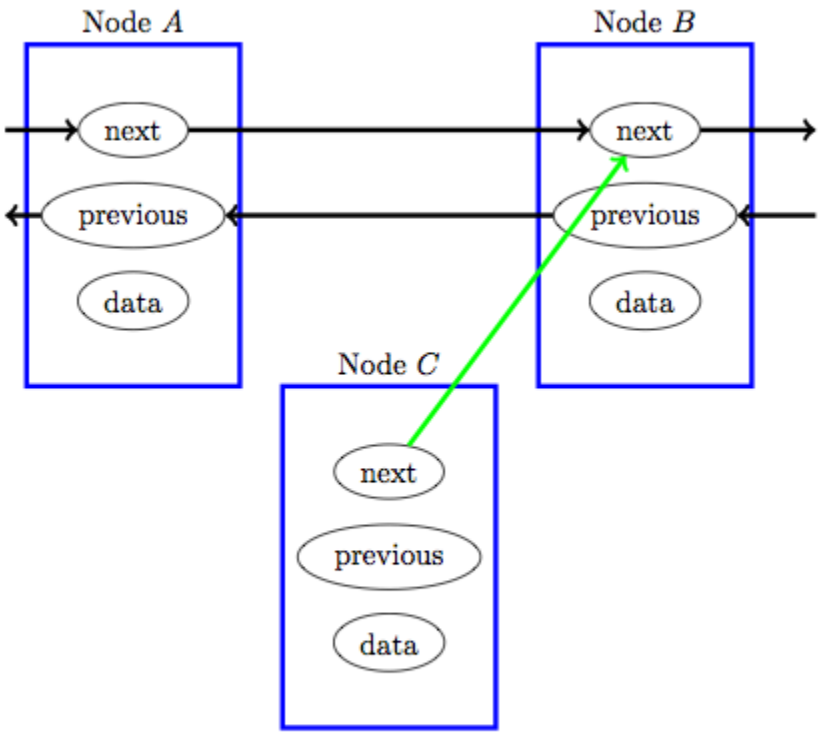

Figure 4.4 Inserting a node, step 1

Inserting a Node

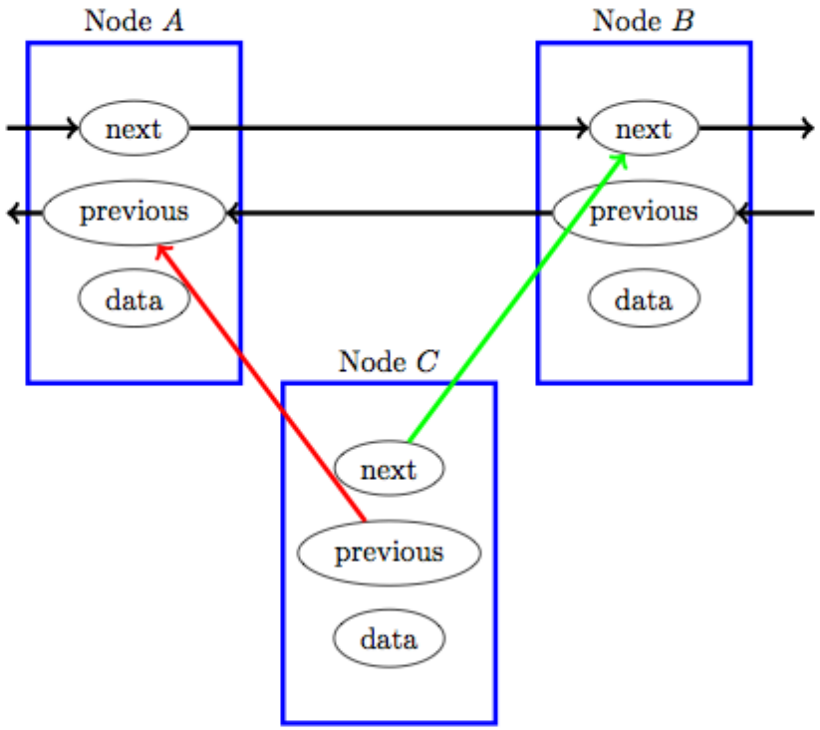

Figure 4.5 Inserting a node, step 2

Inserting a Node

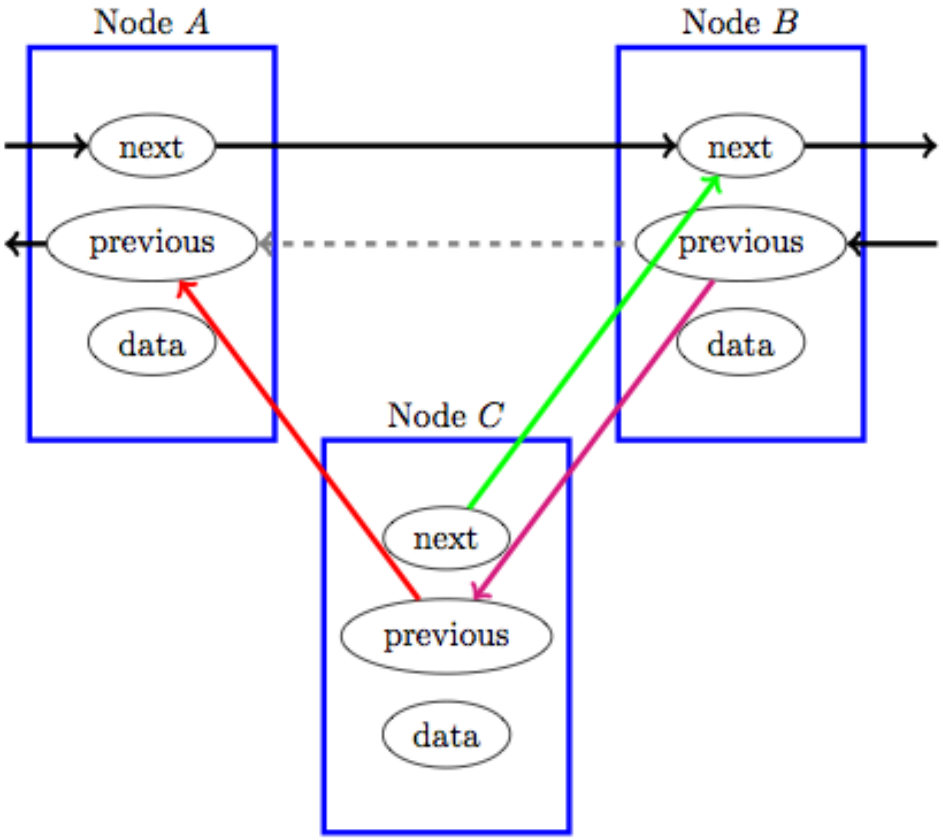

Figure 4.6 Inserting a node, step 3

Inserting a Node

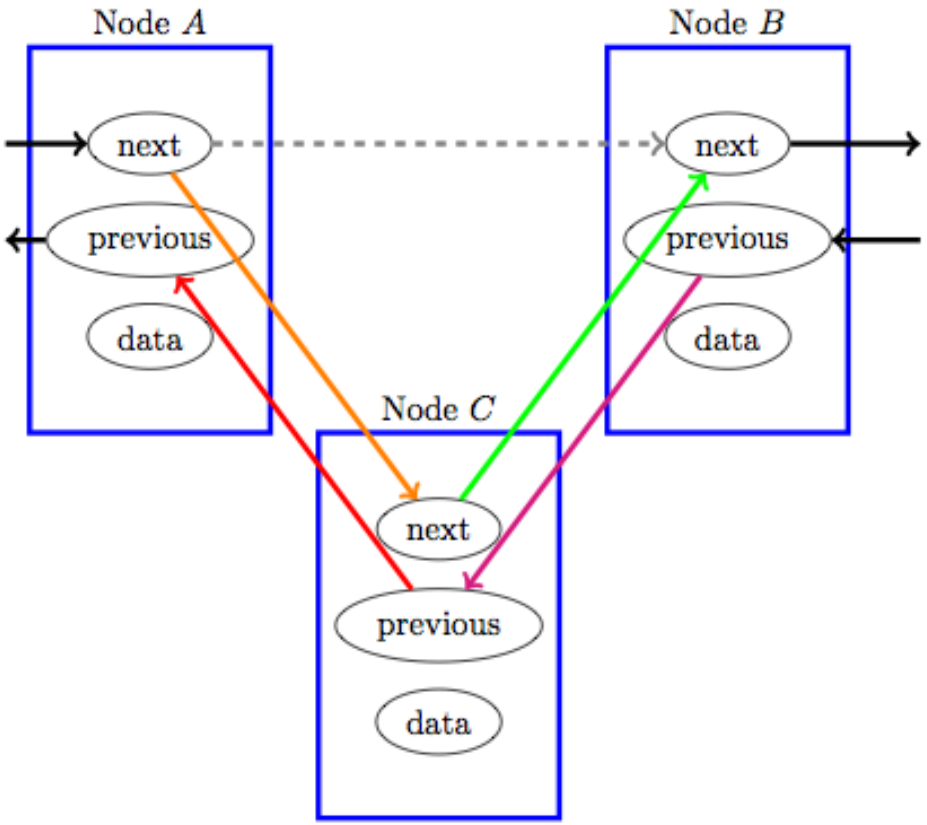

Figure 4.7 Inserting a node, step 4

Inserting a Node

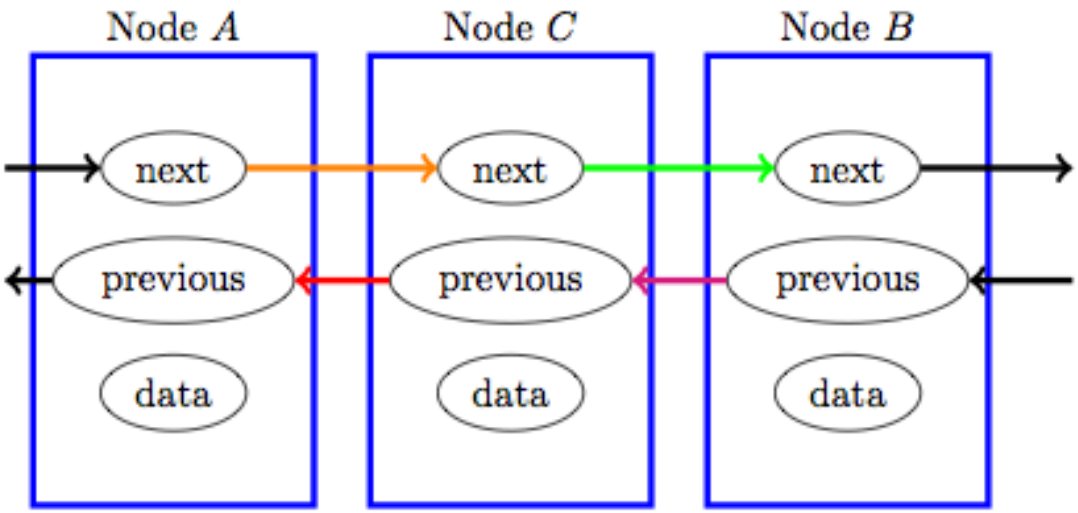

Figure 4.8 Inserting a node, Final Status

Dummy Nodes … or Not?

- Sometimes it's useful to have a dummy head and tail.
- If data is numerically sorted, the head can hold, e.g., `Integer.MIN_VALUE`, and the tail can hold `Integer.MAX_VALUE`, and some of the testing can be cleaner.
- Either with or without is ok, provided that all code works the same way.

Implementation--Code Examples

```
public class DLLNode
{
  private DLLNode next;
  private DLLNode prev;
  private Record nodeData;

  constructor code ...

  public Record getNodeData()
  {
    return this.nodeData;
  }
  public void setNodeData(Record newData)
  {
    this.nodeData = newData;
  }
  public DLLNode getNext()
  {
    return this.next;
  }
  public void setNext(DLLNode newNext)
  {
    this.next = newNext;
  }
  public DLLNode getPrev()
  {
    return this.prev;
  }
  public void setPrev(DLLNode newPrev)
  {
    this.prev = newPrev;
  }

}
```

Figure 4.10 Code fragment for a node

Implementation--Code Examples

```
public class DLL
{
  private int size;
  private DLLNode head;
  private DLLNode tail;

 constructor code ...

  public void add(Record dllData)
  {
    this.addAtHead(dllData);
  }

  private void addAtHead(Record dllData
  {
    DLLNode newNode = null;
    newNode = new DLLNode();
    newNode.setNodeData(dllData);
    this.linkAfter(this.getHead(), newNode);
  }
```

```
  private void linkAfter(DLLNode baseNode, DLLNode newNode)
  {
    newNode.setNext(baseNode.getNext());
    newNode.setPrev(baseNode);
    baseNode.getNext().setPrev(newNode);
    baseNode.setNext(newNode);
    this.incrementSize();
  }

  private void unlink (DLLNode node)
  {
    node.getNext().setPrev(node.getPrev());
    node.getPrev().setNext(node.getNext());
    node.setNext(null);
    node.setPrev(null);
    this.decrementSize();
  }
}
```

Figure 4.11 Code fragment for a doubly linked list

Exception Handling

- What do we do when we try to delete from an already-empty list?
- What do we do when we try to add to a list that is full?

Exception Handling (2)

```
private void unlink (DLLNode node)
{
  if((node.getNext() == null) || (node.getPrev() == null))
  {
    throw new BadNodeException("null value for next or previous");
  }

  node.getNext().setPrev(node.getPrev());
  node.getPrev().setNext(node.getNext());
  node.setNext(null);
  node.setPrev(null);
  this.decrementSize();
}
```

Figure 4.12 A better method for unlink

Exception Handling (3)

```
/*****************************************************************************
 * Copyright (C) 2010 by Duncan A. Buell.  All rights reserved.
 * An exception to be thrown for bad nodes without a valid next or
 * previous pointer.
 *
 * @author Duncan A. Buell
 * @version 1.00 2010-12-24
**/
public class BadNodeException extends RuntimeException
{

/*****************************************************************************
 * Constructor.
**/
  public BadNodeException()
  {
    super();
  } // public BadNodeException()

/*****************************************************************************
 * Constructor with a message to be printed.
 *
 * @param message the message to be printed with the exception.
**/
  public BadNodeException(String errorMessage)
  {
    super(" " + errorMessage);
  } // public BadNodeException(String errorMessage)

} // public class BadNodeException extends RuntimeException
```

Figure 4.13 A class for exception handling

Linked Lists from Arrays

- In ancient days before pointers, arrays were used for everything.
- There was a preallocated *free list* of space not yet used, and a list that was the "actual" linked list.
- Both the free list and the actual LL were linked lists.
- Modern languages use dynamic memory allocation instead.

Linked Lists from Arrays

subscript	0	1	2	3	4	5	6	7	8	9
`myData`	10	9	4	15	23	19	7	1	18	22
`next`	3	0	6	8	null	9	1	2	5	4

subscript	7	2	6	1	0	3	8	5	9	4
`next`	2	6	1	0	3	8	5	9	4	null
`myData`	1	4	7	9	10	15	18	19	22	23

Insertion Sort

- If we have a sorted list, and we have only one entry to add to it, then an insertionsort is appropriate.
- Walk through the list until we find the correct location, and insert into the list at that point.
- This is usually done by adding the element to the end and then pulling it forward to the appropriate location (less data to move).

Insertion Sort (2)

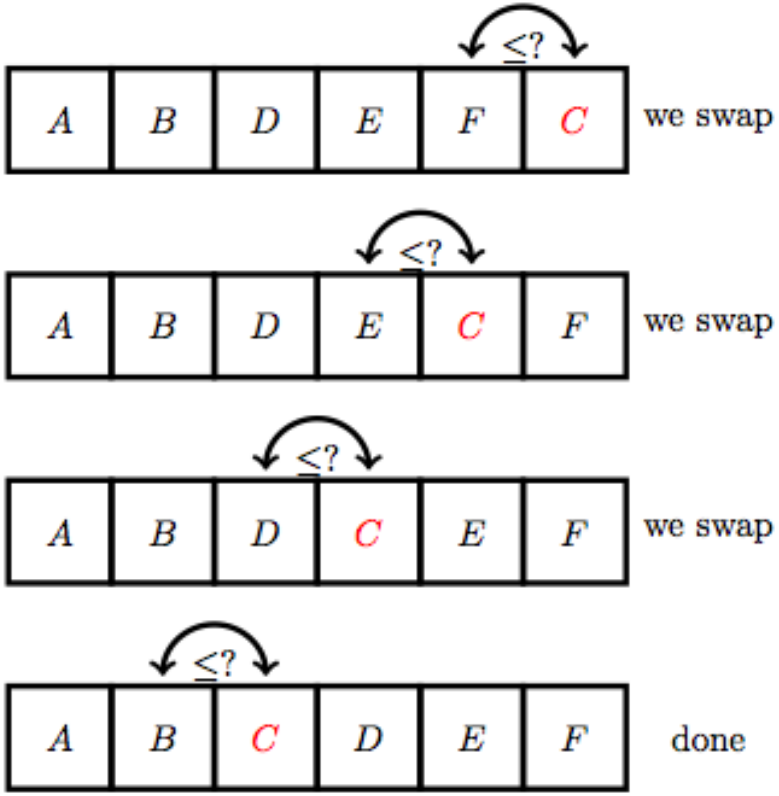

Figure 4.9 Inserting a Node Into an Array

Insertion Sort (3)

```
public void insertAndSort(newElement,insertionSub)
{
  recs[insertionSub] = newElement;
  for(int i = insertionSub-1; i >= 0; --i)
  {
    if(recs[i] > recs[insertionSub])
    {
      swap elements recs[i] and recs[insertionSub]
      insertSub = i;
    }
  }
}
```

Figure 4.14 An insertionsort on an array

Insertion Sort (4)

```
while(newRecordKey < currentRecordKey)
{
  follow pointers from the current record to the next record
}
insert the new record before the current record
```

```
while(newRecordKey < nextRecordKey)
{
  follow pointers from the current record to the next record
}
insert the new record after the current record
```

The End