35. Data Structures

Java

Summer 2008

Instructor: Dr. Masoud Yaghini

Outline

- Introduction
- Lists
- Linked Lists
- Stacks
- Queues
- Trees
- References

Introduction

What is a Data Structure?

- **Data Structure**
  - A data structure is a collection of data organized in some fashion.
  - A data structure not only stores data, but also supports the operations for accessing and manipulating data in the structure.

Arrays

- An array is a data structure that holds a collection of data in sequential order.
- You can find the size of the array, and store, retrieve, and modify data in the array.
- Arrays are simple and easy to use, but they have two limitations:
  - (1) once an array is created, its size cannot be altered;
  - (2) an array does not provide adequate support for insertion and deletion operations.

Classic Dynamic Data Structures

- Four classic dynamic data structures are introduced in this chapter:
  - linked lists,
  - stacks,
  - queues, and
  - trees.

Classic Dynamic Data Structures

- **Linked lists**
  - are collections of data items "linked up in a chain"insertions and deletions can be made anywhere in a linked list.
- **Stacks**
  - are important in compilers and operating systems; insertions and deletions are made only at one end of a stackits top.

Classic Dynamic Data Structures

- **Queues**
  - represent waiting lines; insertions are made at the back (also referred to as the **tail**) of a queue and deletions are made from the front (also referred to as the **head**).
- **Trees**
  - is a data structure that supports searching, sorting, inserting, and deleting data efficiently.

Object-Oriented Data Structure

- In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements.
- Some people refer to data structures as container objects or collection objects.
- To define a data structure is essentially to declare a class.
- The class for a data structure should use data fields to store data and provide methods to support such operations as insertion and deletion.

Object-Oriented Data Structure

- To create a data structure is therefore to create an instance from the class.
- You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into the data structure or deleting an element from the data structure.

Lists

Lists

- A list is a popular data structure for storing data in sequential order.
- For example, a list of students, a list of available rooms, a list of cities, and a list of books can all be stored using lists.
- The operations listed below are typical of most lists:
  - Retrieve an element from a list.
  - Insert a new element to a list.
  - Delete an element from a list.
  - Find how many elements are in a list.
  - Find whether an element is in a list.
  - Find whether a list is empty.

Two Ways to Implement Lists

- There are two ways to implement a list.
  - **Using Arrays**
    - Arrays are dynamically created. If the capacity of the array is exceeded, create a new, larger array and copy all the elements from the current array to the new array.
  - **Using linked structures**
    - A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

Array Lists

Array Lists

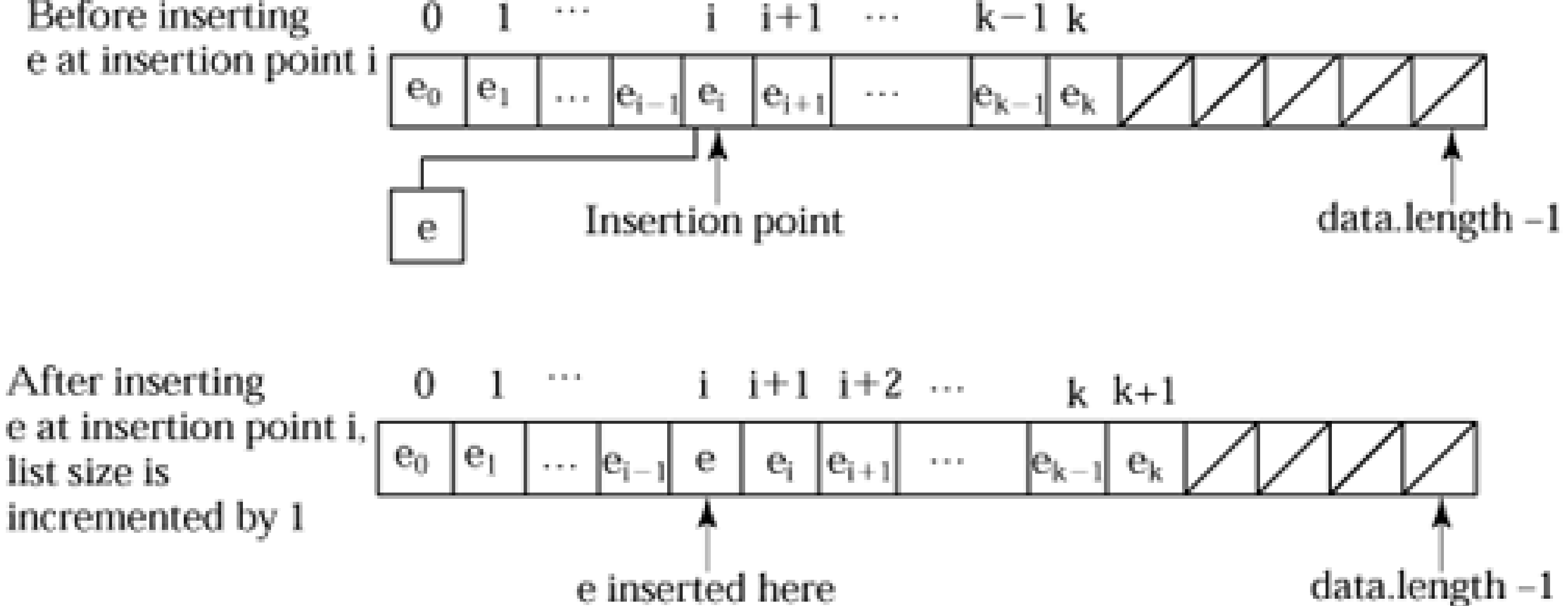
- Array is a fixed-size data structure.
- Once an array is created, its size cannot be changed.
- Nevertheless, you can still use arrays to implement dynamic data structures.
- The trick is to create a larger new array to replace the current array if the current array cannot hold new elements in the list.
- This section shows how to use arrays to implement MyArrayList.

Array Lists

- Initially, an array, say data of Object[ ] type, is created with a default size. Each cell in the array actually stores the reference of an object.
- When inserting a new element into the array, first make sure that there is enough room in the array.
- If not, create a new array twice as large as the current one.
- Copy the elements from the current array to the new array. The new array now becomes the current array.

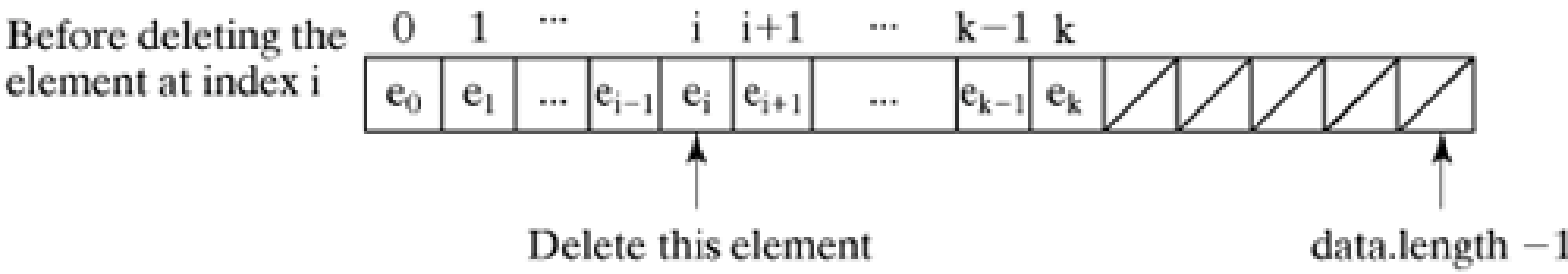
Inserting a new element to the array

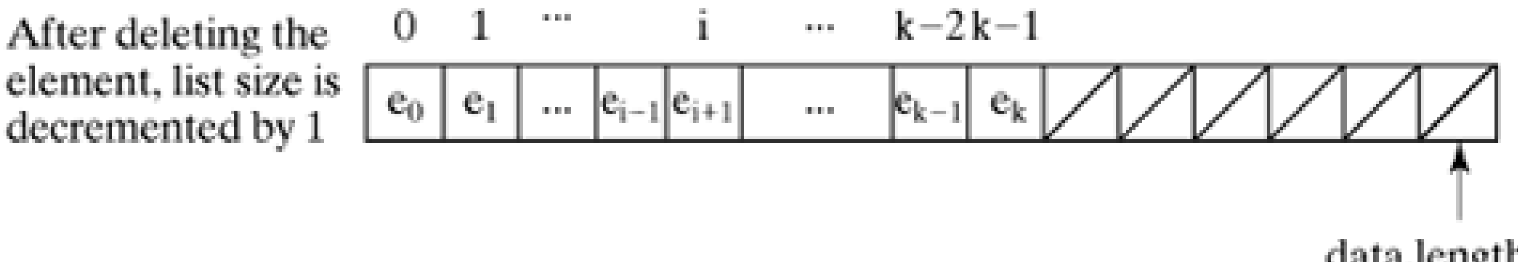
- Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.



Deleting an element from the array

- Deleting an element from the array requires that all the elements after the deletion point be shifted one position to the left and decrease the list size by 1





Array Lists

- Implementing array list
  - MyArrayList
  - TestList

Linked Lists

- Since MyArrayList is implemented using an array, the following methods are efficient:
  - get(int index) for accessing an element
  - set(int index, Object o) for modifying an element through an index, and
  - add(Object o) for adding an element at the end of the list
- However, the methods:
  - add(int index, Object o) and
  - remove(int index)
  - are inefficient because they require shifting a potentially large number of elements.

Linked Lists

Linked Lists

- Linked list
  - Linear collection of nodes
    - Self-referential-class objects connected by reference links
    - Can contain data of any type
  - A program typically accesses a linked list via a reference to the first node in the list
    - A program accesses each subsequent node via the link reference stored in the previous node
    - The link reference in the last node is set to null to mark the end of the list.

Linked Lists

- Linked Lists are dynamic
  - The length of a list can increase or decrease as necessary
  - Become full only when the system has insufficient memory to satisfy dynamic storage allocation requests
- Stacks and queues are also linear data structures and, as we will see, are constrained versions of linked lists.
- Trees are non-linear data structures.

Performance Tip

- An array can be declared to contain more elements than the number of items expected, but this wastes memory.
- Linked lists provide better memory utilization in these situations.
- Linked lists allow the program to adapt to storage needs at runtime.

Performance Tip

- Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point).
- All existing node objects remain at their current locations in memory.

Performance Tip

- Insertion and deletion in a sorted array can be time consuming
- All the elements following the inserted or deleted element must be shifted appropriately.

Linked Lists

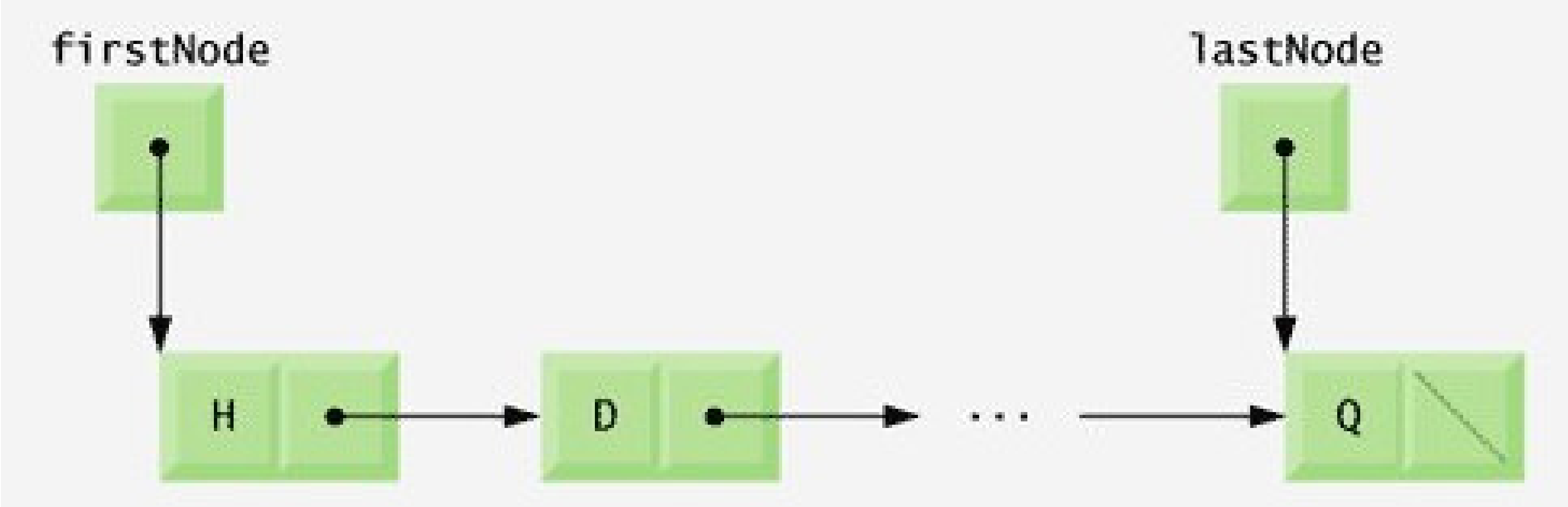
- Singly linked list
  - Each node contains one reference to the next node in the list
- Doubly linked list
  - Each node contains a reference to the next node in the list and a reference to the previous node in the list
  - `java.util`'s `LinkedList` class is a doubly linked list implementation

Performance Tip

- Normally, the elements of an array are contiguous in memory.
- This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array.
- Linked lists do not afford such immediate access to their elements
- An element can be accessed only by traversing the list from the front (or from the back in a doubly linked list).

Linked Lists

- Linked list graphical representation



TestLinkedList.java & List.Java

- The program of TestLinkedList.java uses an object of List class in List.java to manipulate a list of miscellaneous objects.
- List.java consists of two classes ListNode and List .
- Encapsulated in each List object is a linked list of ListNode objects.
  - List.java
  - TestLinkedList.java

TestLinkedLists.Java Output

The list is: 7

The list is: 11 7

The list is: 12 11 7

The list is: 12 11 7 5

12 removed
The list is: 11 7 5

5 removed
The list is: 11 7

Linked Lists

- Class ListNode declares package-access fields data and nextNode.
- The data field is an Object reference, so it can refer to any object.
- ListNode member nextNode stores a reference to the next ListNode object in the linked list (or null if the node is the last one in the list).

Linked Lists

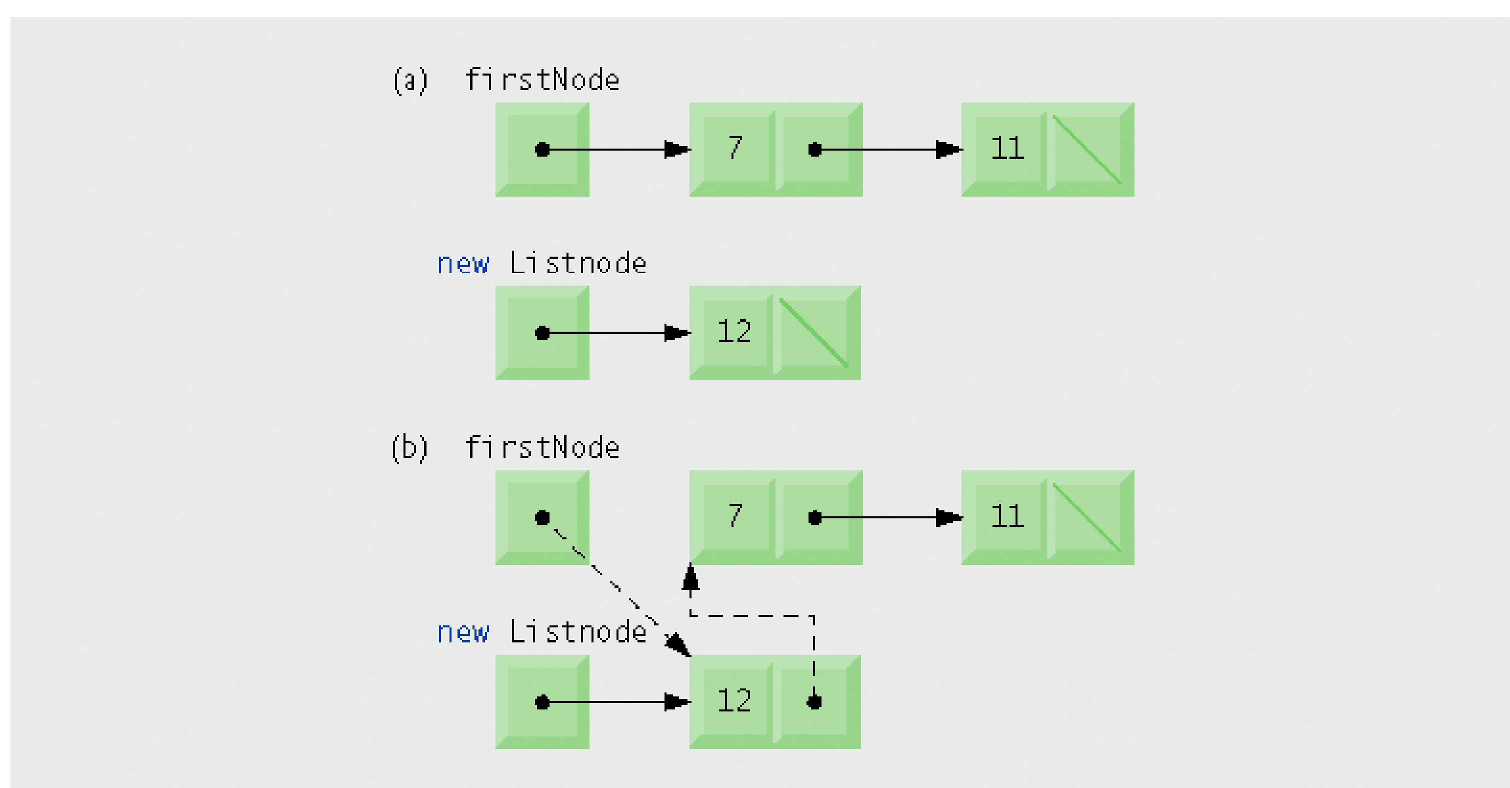
- Method main of class TestLinkedList
  - inserts objects at the beginning of the list using method insertAtFront,
  - inserts objects at the end of the list using method insertAtBack,
  - deletes objects from the front of the list using method removeFromFront
  - deletes objects from the end of the list using method removeFromBack

Linked Lists

- Method insertAtFront's steps
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, set firstNode to a new ListNode object and initialize that object with insertItem and firstNode
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to the ListNode passed as argument, which previously was the first node

Linked Lists

- Graphical representation of operation insertAtFront

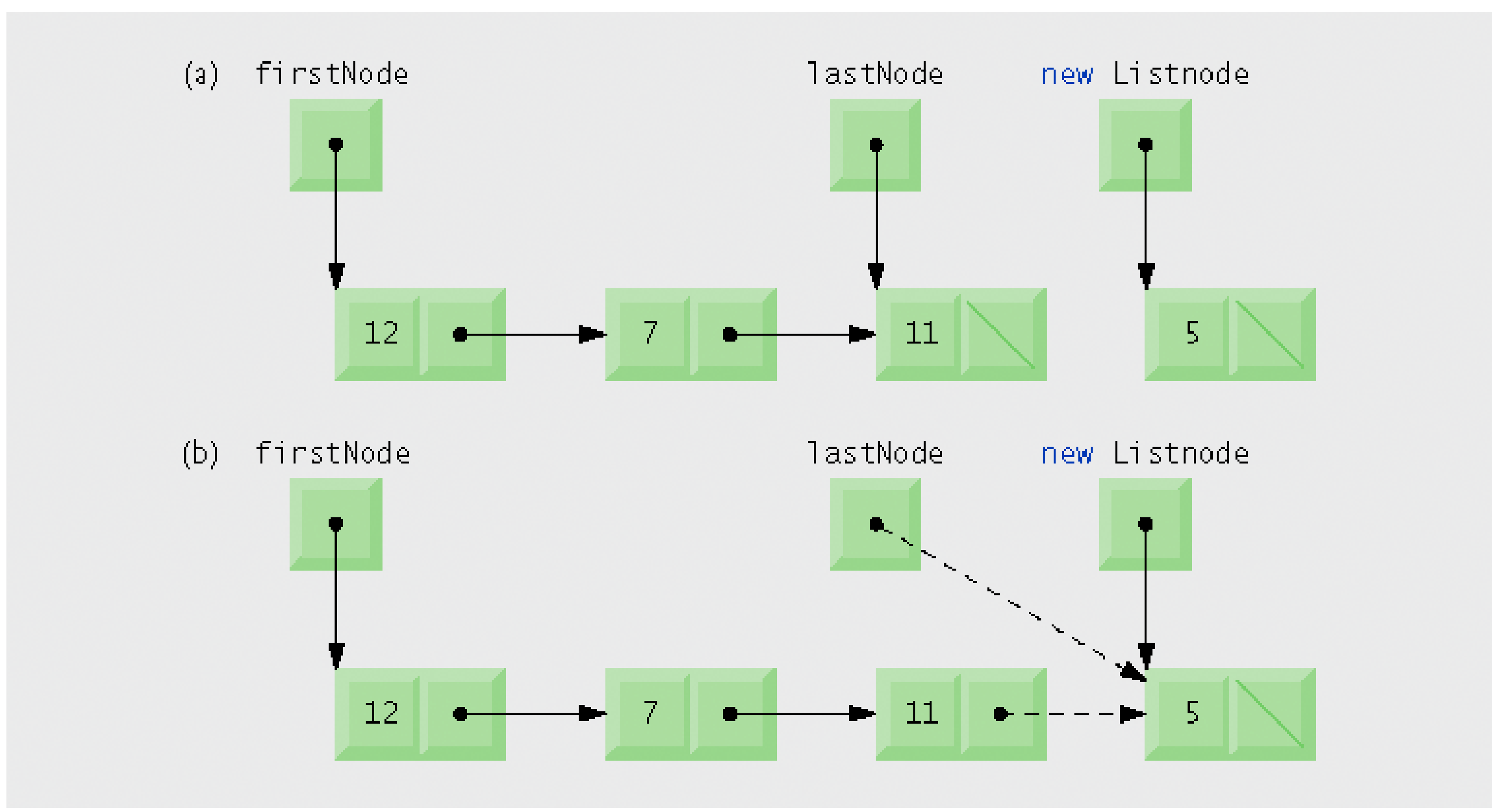


Linked Lists

- Method insertAtBack's steps
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, assign to lastNode and lastNode.nextNode the reference to the new ListNode that was initialized with insertItem
    - The ListNode constructor sets data to refer to the insertItem passed as an argument and sets reference nextNode to null

Linked Lists

- Graphical representation of operation `insertAtBack`

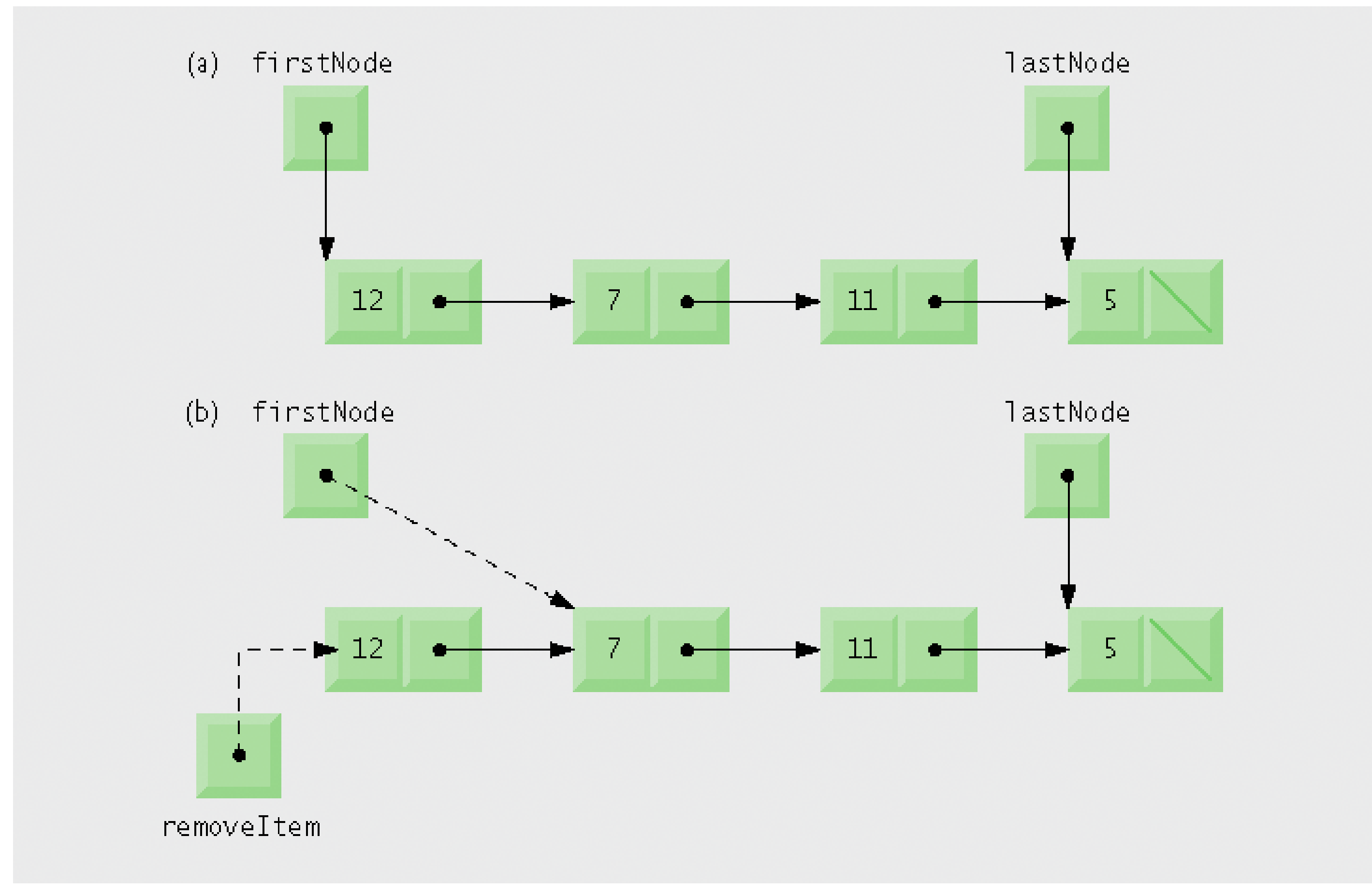


Linked Lists

- Method removeFromFront's steps
  - Throw an EmptyListException if the list is empty
  - Assign firstNode.data to reference removedItem
  - If firstNode and lastNode refer to the same object, it means there is only one node in list, set firstNode and lastNode to null
  - If the list has more than one node, assign the value of firstNode.nextNode to firstNode
  - Return the removedItem reference

Linked Lists

- Graphical representation of operation removeFromFront

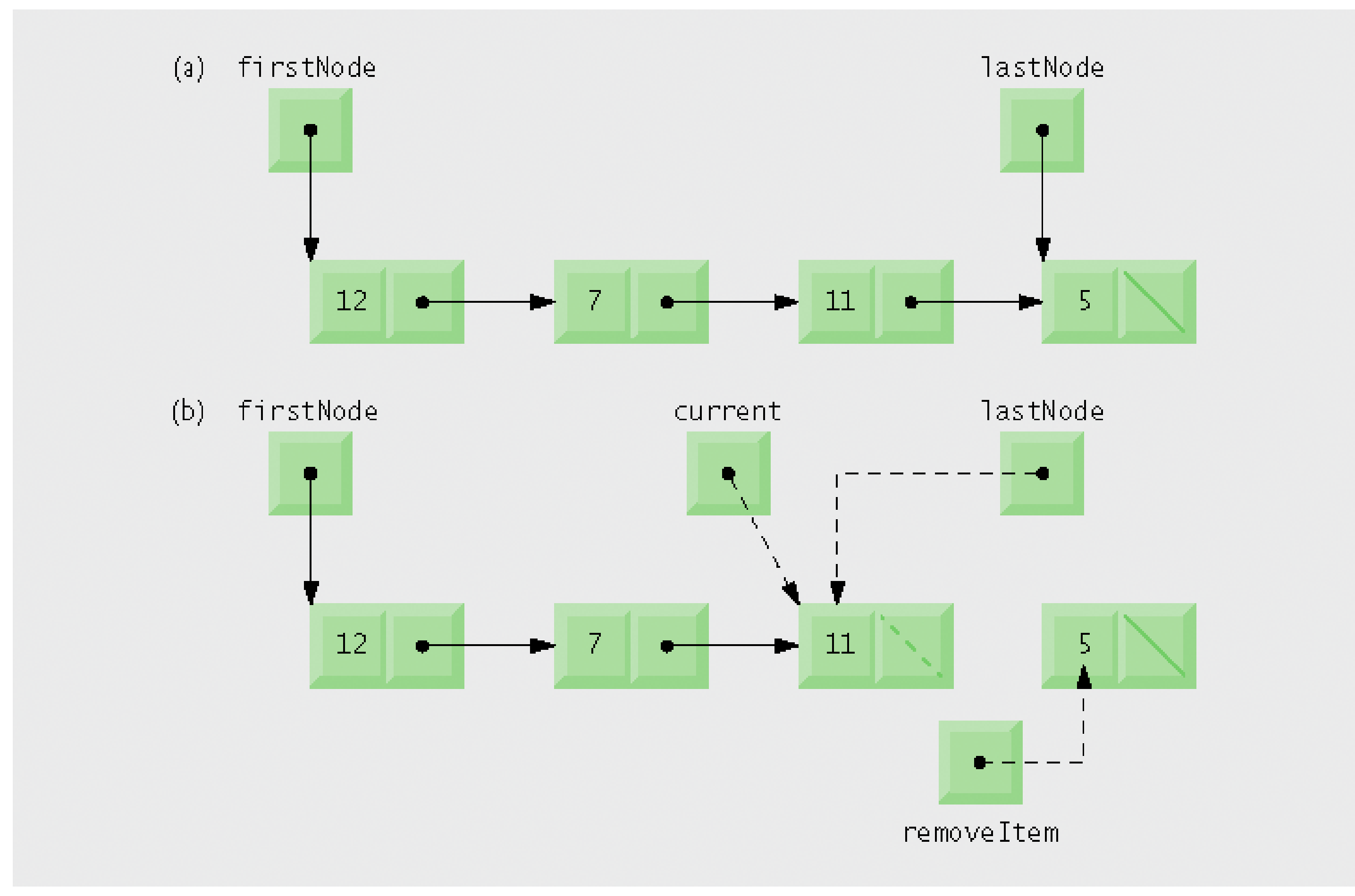


Linked Lists

- Method removeFromBack's steps
  - Assign lastNode.data to removedItem
  - If the firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, create the ListNode reference current and assign it firstNode
  - "Walk the list" with current until it references the node before the last node
    - The while loop assigns current.nextNode to current as long as current.nextNode is not lastNode
  - Assign current to lastNode
  - Set current.nextNode to null
  - Return the removedItem reference

Linked Lists

- Graphical representation of operation `removeFromBack`



Stacks

Stacks

- Stacks
  - A stack is a constrained version of a linked list
  - The link member in the bottom (i.e., last) node of the stack is set to null to indicate the bottom of the stack.
  - **Last-in, first-out (LIFO)** data structure
    - Method push adds a new node to the top of the stack
    - Method pop removes a node from the top of the stack and returns the data from the popped node
  - **Program execution stack**
    - Holds the return addresses of calling methods
    - Also contains the local variables for called methods

Stacks

- The StackInheritance class that inherits from List
  - Stack methods push, pop, isEmpty and print are performed by inherited methods insertAtFront, removeFromFront, isEmpty and print
    - push calls insertAtFront
    - pop calls removeFromFront
    - isEmpty and print can be called as inherited
  - Other List methods are also inherited
    - Including methods that should not be in the stack class's public interface

Stacks

- The programs:
  - StackInheritance.java
  - QueueTest.java

Stacks

- Class StackInheritanceTest's method main creates an object of class StackInheritance called stack.
- The program output:

The stack is: -1

The stack is: 0 -1

The stack is: 1 0 -1

The stack is: 5 1 0 -1

5 popped

The stack is: 1 0 -1

1 popped

The stack is: 0 -1

Queues

Queues

- Queue
  - Similar to a checkout line in a supermarket
  - **First-in, first-out (FIFO)** data structure
    - **Enqueue** inserts nodes at the tail (or end)
    - **Dequeue** removes nodes from the head (or front)
  - Used to support print spooling
    - A spooler program manages the queue of printing jobs

Queues

- Queue class that contains a reference to a List
  - Method enqueue calls List method insertAtBack
  - Method dequeue calls List method removeFromFront
  - Method isEmpty calls List method isEmpty
  - Method print calls List method print

- The programs:
  - QueueInheritance .java
  - QueueInheritanceTest.java

Queues

- The program output:

  The queue is: -1

  The queue is: -1 0

  The queue is: -1 0 1

  The queue is: -1 0 1 5

  -1 dequeued

  The queue is: 0 1 5

  0 dequeued

  The queue is: 1 5

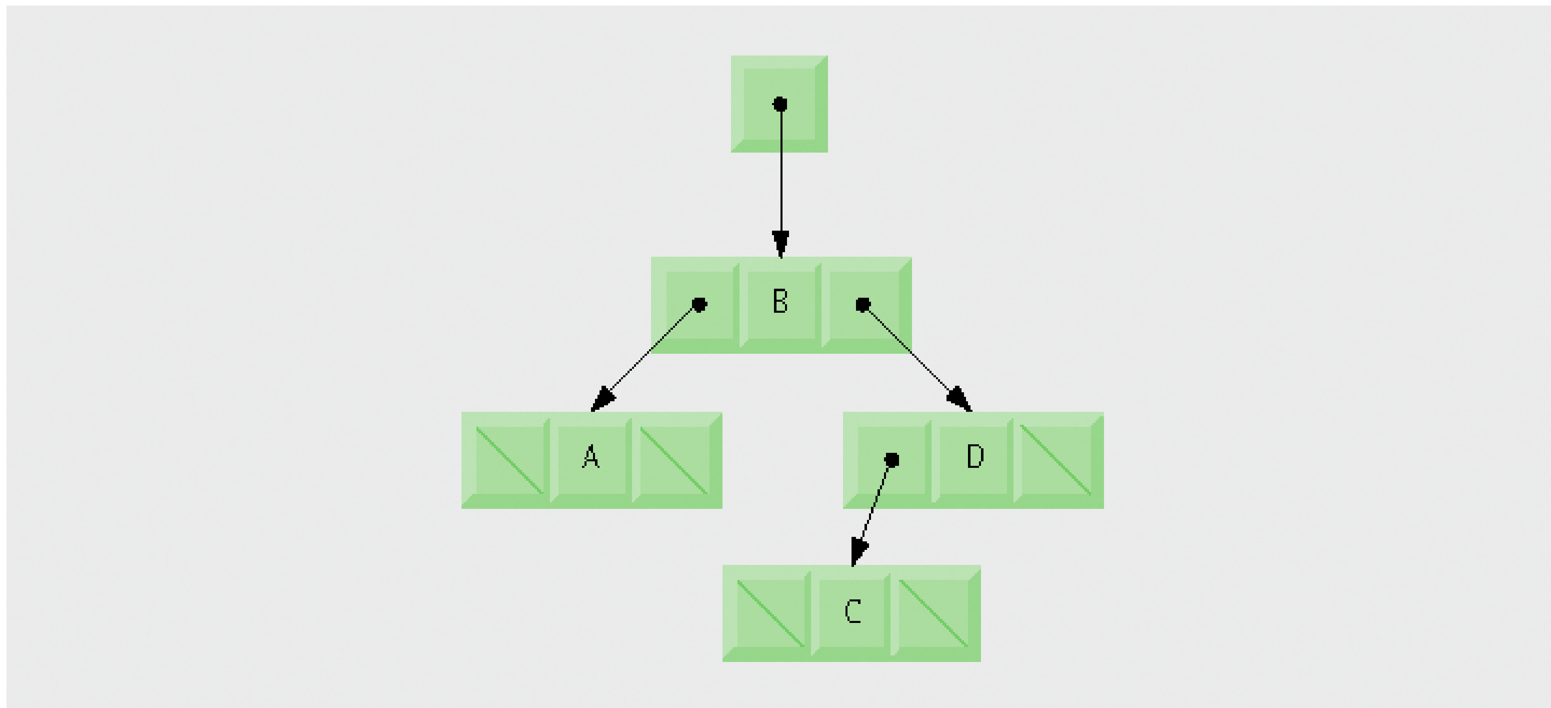
Trees

Trees

- **Trees**
  - The **root node** is the first node in a tree
  - Tree nodes contain two or more links
  - The children of a specific node are called **siblings**.
  - A **leaf node** has no children
- **Binary trees**
  - Trees whose nodes each contain **two links** (one or both of which may be null).
  - Each link refers to a child
    - **Left child** is the root of the **left subtree**
    - **Right child** is the root of the **right subtree**

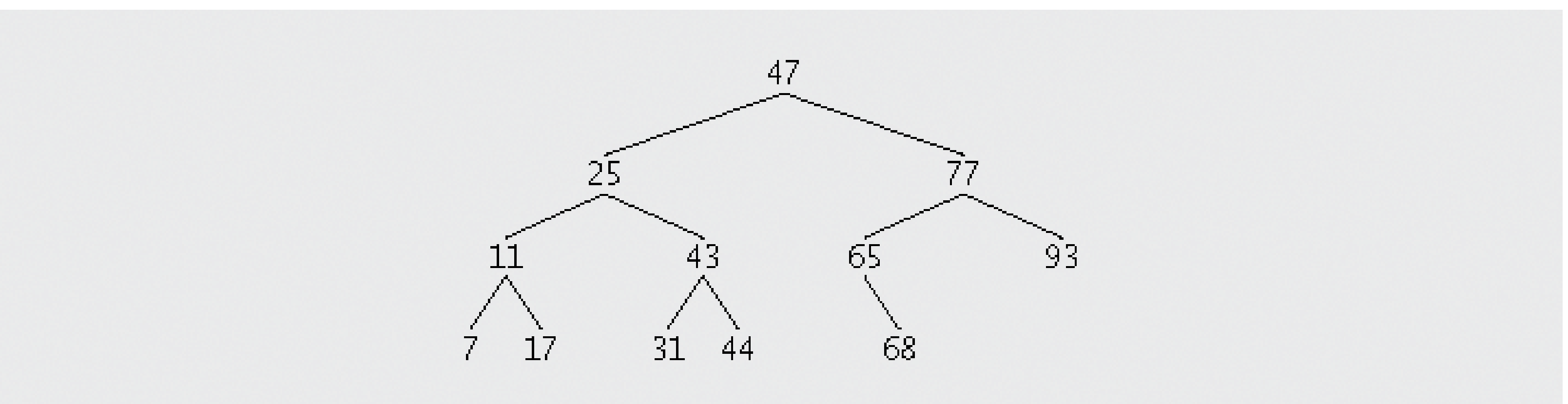
Trees

- Binary tree graphical representation:



Trees

- **Binary search trees**
  - Values in the left subtree are less than the value in that subtree’s parent node and values in the right subtree are greater than the value in that subtree’s parent node



Trees

- The programs:
  - TreeTest.java
  - Tree.java

Trees

- Class Tree's method insertNode first determines whether the tree is empty.
- If so, it allocates a new TreeNode, initializes the node with the integer being inserted in the tree and assigns the new node to reference root.
- If the tree is not empty, it calls TreeNode method insert.
- This method uses recursion to determine the location for the new node in the tree and inserts the node at that location.
- A node can be inserted only as a leaf node in a binary search tree.

Trees

- **Traversing a tree**
  - **Inorder** - traverse left subtree, then process root, then traverse right subtree
  - **Preorder** - process root, then traverse left subtree, then traverse right subtree
  - **Postorder** - traverse left subtree, then traverse right subtree, then process root

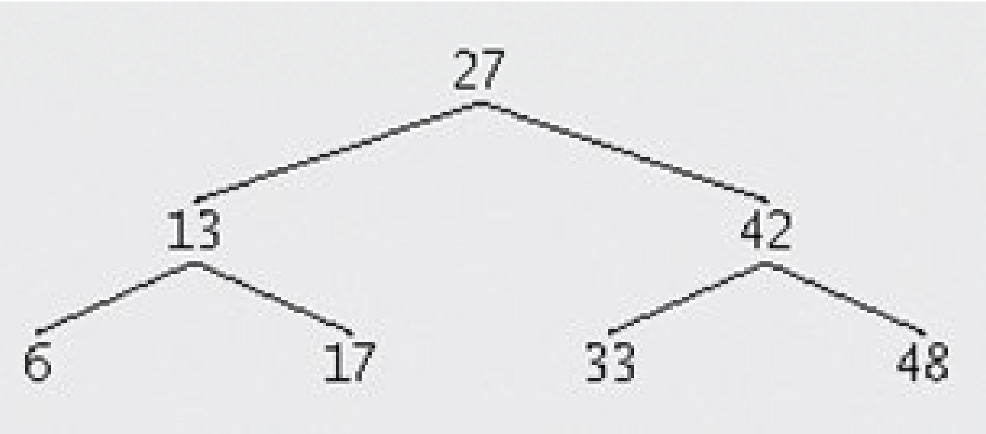
Trees

- Inorder traversal steps:
  - Return immediately if the reference parameter is null
  - Traverse the left subtree with a call to inorderHelper
  - Process the value in the root node
  - Traverse the right subtree with a call to inorderHelper
- **Binary tree sort**:
  - The inorder traversal of a binary search tree prints the node values in ascending order

Trees

- The inorder traversal of the tree:

  6 13 17 27 33 42 48



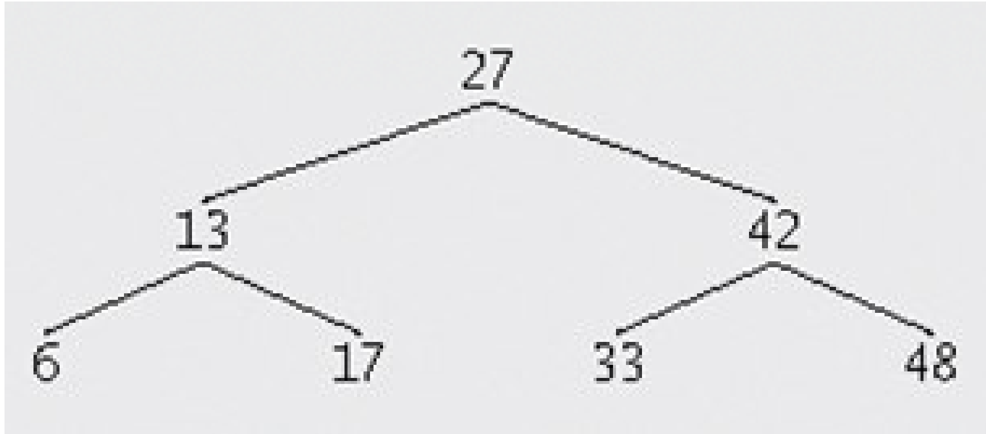
Trees

- Preorder traversal steps
  - Return immediately if the reference parameter is null
  - Process the value in the root node
  - Traverse the left subtree with a call to preorderHelper
  - Traverse the right subtree with a call to preorderHelper

Trees

- The preorder traversal of the tree:

  27 13 6 17 42 33 48



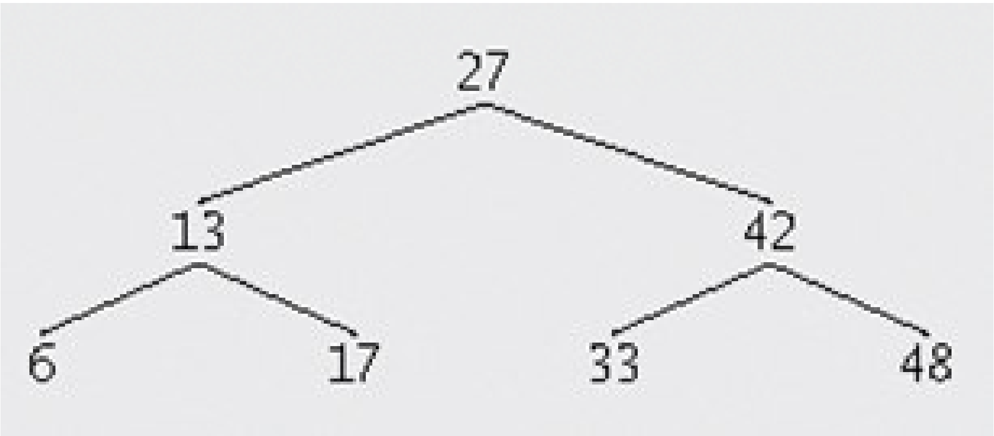
Trees

- Postorder traversal steps
  - Return immediately if the reference parameter is null
  - Traverse the left subtree with a call to postorderHelper
  - Traverse the right subtree with a call to postorderHelper
  - Process the value in the root node

Trees

- The postorderTraversal of the tree:

  6 17 13 33 48 42 27



Trees

- **Duplicate elimination**
  - Because duplicate values follow the same “go left” or “go right” decisions, the insertion operation eventually compares the duplicate with a same-valued node
  - The duplicate can then be ignored
- **Tightly packed (or balanced) trees**
  - Each level contains about twice as many elements as the previous level

References

References

- H. M. Deitel and P. J. Deitel, **Java™ How to Program**, Sixth Edition, Prentice Hall, 2005. (Chapter 17)

The End