CIS 265 Summer 2023 Notes

The midterm will cover chapters 11 – 13, and 18 – 20.

The final exam covers chapters 1 – 13 and 18 – 29.

Chapter 11: Inheritance and Polymorphism

Inheritance – a key feature of object-oriented programming that allows you to define new classes from existing classes.

- Allows you to avoid redundancy when creating several classes with common features
- A subclass / child class inherits from a superclass / parent class

Syntax for defining a subclass that inherits from a superclass:

```
public class ChildClassName extends ParentClassName {
      // class body
}
```

- Constructor chaining
- A child class constructor can explicitly call one of the constructors of its superclass using the `super()` keyword
- If no explicit call to `super()`, then the no-arg constructor of the superclass is called implicitly by the subclass constructor; the superclass first calls the constructor of its parent class, and so on.

Overriding vs. Overloading Methods

- A subclass can ***override*** a method with the same name in its superclass if it has the same ***signature*** (i.e. same parameter types, in the same order). This allows the subclass to create a different implementation of the method than was defined in its superclass.
- A class cannot override its own method.

- ***Overloading*** is the process of defining multiple methods with the same name, but different signatures.
- Data and methods visibility controlled by access modifier:

Table 1: Access Modifiers

Access modifier	Same class	Same package	Subclass in a different package	Non-subclass in different package
`public`	☑	☑	☑	☑
`protected`	☑	☑	☑	
`default (no modifier)`	☑	☑		
`private`	☑			

Polymorphism – another key feature of object-oriented programming

- You can pass an instance object of a subclass to a parameter of its superclass type, but not vice versa.
- An object of a subclass can be used whenever its superclass object is used.
- A variable of a supertype can refer to a subtype object.

Chapter 12: Exception Handling and Text I/O

Chapter 13: Abstract Classes and Interfaces

Chapter 18: Recursion

- ***Recursive methods*** are methods that invoke themselves.
    - The advantage of recursion is that in some cases it allows you to develop a natural, straightforward, simple solution to an otherwise difficult problem.
    - The disadvantage is that in many cases, recursion is not efficient, requiring more time and more memory than a solution based on loops.
- All recursive methods have the following characteristics (see section 18.4):

  - The method is implemented using an if-else or switch statement leading to different cases.
  - One or more base cases (simplest case) are used to stop recursion; i.e. value is returned without having to call recursive function.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.
- A ***recursive helper method*** recursively solves a problem similar to the original problem. The original problem can be solved by invoking the recursive helper method.
- Any problem that can be solved recursively can be solved non recursively with iterations.
- Recursion bears substantial overhead.
  - Each time the method is called, the system must allocate memory for the method's local variables and parameters.
  - Requires extra time to manage memory.
- In some cases, recursion allows you to specify a clear, simple solution to an inherently recursive problem.
- A recursive method is said to be ***tail recursive*** if there are no pending operations to be performed on return from a recursive call.
  - Tail recursion is desirable, because the method ends when the last recursive call ends. This implies that there is no need to store intermediate calls in the stack.
  - Compilers can optimize tail recursion to reduce stack size.

Code Examples:

Standard recursive factorial method

```
public static long factorial(int n) {
    if (n == 0) // Base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call
  }
```

Recursive factorial with tail recursion. Note that this also includes a helper method.

```
public class ComputeFactorialTailRecursion {
  /** Return the factorial for a specified number */
  public static long factorial(int n) {
    return factorial(n, 1); // Call auxiliary method
  }

  /** Auxiliary tail-recursive method for factorial */
  private static long factorial(int n, int result) {
    if (n == 0)
      return result;
    else
```

```
        return factorial(n - 1, n * result); // Recursive call
    }
}
```

Recursive palindrome method; illustrates helper method.

```
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  public static boolean isPalindrome(String s, int low, int high)
{
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }
```

Recursive Finonacci method defined in text; how many times is it called when parameter n is passed to it?

# of calls to `fib()` method for `fib(n)` = 1 (for fib(n) itself) + # of calls for fib(n-1) + # of calls for fib(n-2)

Examples: fib(0) = fib(1) = 1 (these are the base cases); fib(2) = 1 + #fib(1) + #fib(0) = 3; fib(3) = 1 + #fib(2) + #fib(1) = 5; fib(4) = 1 + #fib(3) + #fib(2) = 9; etc.

Chapter 19: Generics

- Key benefit – enable errors to be detected at compile time, rather than at runtime.
    - Generic class or method allows you to specify allowable types of objects that the class or method can work with.
- Typically, in class or method definitions, a single capital letter **<E>**, or sometimes **<T>**, represents a ***formal generic type***.
- Replacing a generic type with a specific type is called a ***generic instantiation***, or replacing the generic type with a ***concrete type***.

Syntax: To **define a generic type for a class**, place <E> after the class name, for example

```
public class GenericStack<E> {
     // … class body

     public GenericStack() {
          // no-arg constructor for this class.
          // Note that it does not include <E> after the method
          // name.
     }
```

```
}
```

Syntax: To define a generic type for a method, place the generic type <E> after the static keyword, and before the method return type. For example

```
public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
```

Example: creating an object of a generic class type with a concrete type specified

```
ArrayList<String> list = new ArrayList<>();
```

Note: in a class with a generic type, the generic type must always be an object; it can never be a primitive type. For example, the following is incorrect

```
ArrayList<double> double_list = new ArrayList<>(); // incorrect
```

- Use a wrapper class for the primitive type.

Syntax for using a generic static method – prefix method name with concrete type in angle brackets (e.g. the print() method listed above):

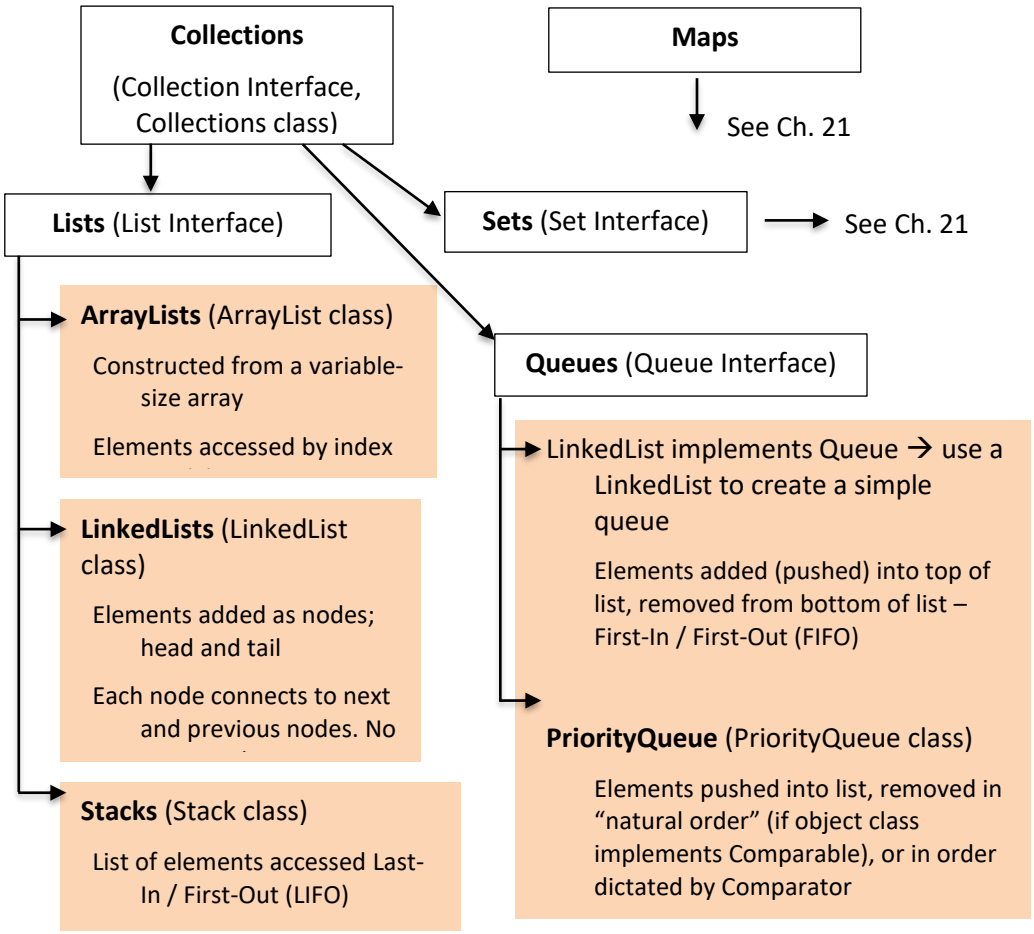
```
ClassName.<ConcreteType>method_name(parameters);

GenericMethodDemo.<Integer>print(list1);
GenericMethodDemo.<String>print(list2);
```

Chapter 20: Lists, Stacks, Queues, and Priority Queues

Java Collections Framework – a set of interfaces, abstract classes, and concrete classes that create a number of key data structures, and methods to operate on them.

Figure 1: Java Collections Framework. Classes Discussed in Ch. 20 are Highlighted



Defines two general types of data structures:

- **Collection** – stores a collection of elements (i.e. objects)
- **Map** – stores key / value pairs (chapter 21)

Collections include

- **Lists** – store an sequentially ordered collection of elements
    - **ArrayList** – sequenced by index; implemented using an array that can be resized as needed.
    - **LinkedList** – sequenced by "previous" and "next" links to the next object in the list
    - **Stacks** – store objects, allowing them to be added / processed on a last-in / first-out order (like a stack of dinner plates in a cafeteria).

- **Sets** – store a group of nonduplicate elements (chapter 21)
- **Queues** – store objects, allowing to be added / processed first-in, first-out (like people waiting in a line)
    - LinkedList implements Queue, so a simple queue can be created with a LinkedList.
    - **PriorityQueues** -- store objects that are processed in order of priority (like an emergency room at a hospital)

Key methods in the `Collection` interface (in almost all cases, usable by every class that implements `Collection`).

- size(): int
- add(o: E)
- addAll(c: Collection<? extends E>) – similar to a set union
- remove(o: Object) – removes one instance of o from collection, if present
- removeAll(c: Collection<?>) – remove all elements of c from this collection, if present. – like a set difference
- retainAll(c: Collection<?>) – removes all elements of this collection except for those that are also in c – like a set intersection
- clear()

Although it is more confusing than it should be, in addition to the `Collection` interface, **there is also a class called `Collections`** that has some useful methods.

- Collections.sort(List<T> listName) or Collections.sort(List<T> listName, Comparator<? super T> c) – these methods **only work with lists**.
- Collections.min/max(Collection<? extends T> listName) or Collections.min/max(Collection<? extends T> listName, Comparator<? super T> c) – return the max or min object from the collection (works with all concrete classes that implement Collection).

Some **key methods in the `List` Interface** – inherited by both `ArrayList` and `LinkedList`

- add(index, element)
- get(index): E
- indexOf(element: Object): int; lastIndexOf(element): int
- set(index, element) – replaces element at index
- remove(index)
- subList(fromIndex, toIndex): List<E> -- returns a sublist [fromIndex, toIndex-1]

Note: both ArrayList and LinkedList have constructors that allow an object of either type to be created from an array, using the **`Arrays.asList()`** method. For example,

```
      String[] colors_array = {"red", "green", "blue"};
ArrayList<String> colors_list = new ArrayList<>(Arrays.asList(colors_array));
```

Some key methods that are **unique to `LinkedList`**

- addFirst(element: E)

- addLast(element: E)
- getFirst(): E
- getLast(): E
- removeFirst(): E
- removeLast(): E

Note that ArrayList<E> includes two overloaded remove() methods:

- remove(index: int) – removes the element at index (this overrides the corresponding method in the `List` interface).
- remove(o: Object) – removes the **first** instance of object o in the array list (if present)

Syntax for creating objects of ArrayList and LinkedList types:

```
ArrayList<AConcreteType> listname = new ArrayList<>();

LinkedList<AConcreteType> listname = new LinkedList<>();
```

Note that, because of polymorphism, you can also define a reference variable of type List and assign it to an object of type ArrayList or LinkedList.

```
List<AConcreteType> listname = new ArrayList<>();
```

Stacks

Syntax for creating an object of `Stack` type:

```
Stack<AConcreteType> stackname = new Stack<>();
```

Key methods in `Stack` class:

- `empty(): boolean` – returns true if the stack is empty, false if not empty
- `peek(): E` – Returns top element of stack, without removing it
- `pop(): E` – Removes top element of stack, and returns it
- `push(o: E)` – Adds new element to top of stack

Simple Queues with LinkedList

Syntax for creating a queue with a LinkedList:

```
Queue<AConcreteType> queuename = new LinkedList<>();
```

Key methods in `Queue` interface:

- `offer(o: E)` – Inserts an element into the queue.
- `poll(): E` – Returns and removes element at head of queue, or `null` if queue is empty.
- `remove(): E` – Returns and removes element at head of queue; throws an exception if queue is empty.

- `peek(): E` – Returns, but does not remove, head element; `null` if queue is empty.
- `element(): E` -- Returns, but does not remove, head element; throws an exception if queue is empty.

PriorityQueues

Syntax for creating a priority queue:

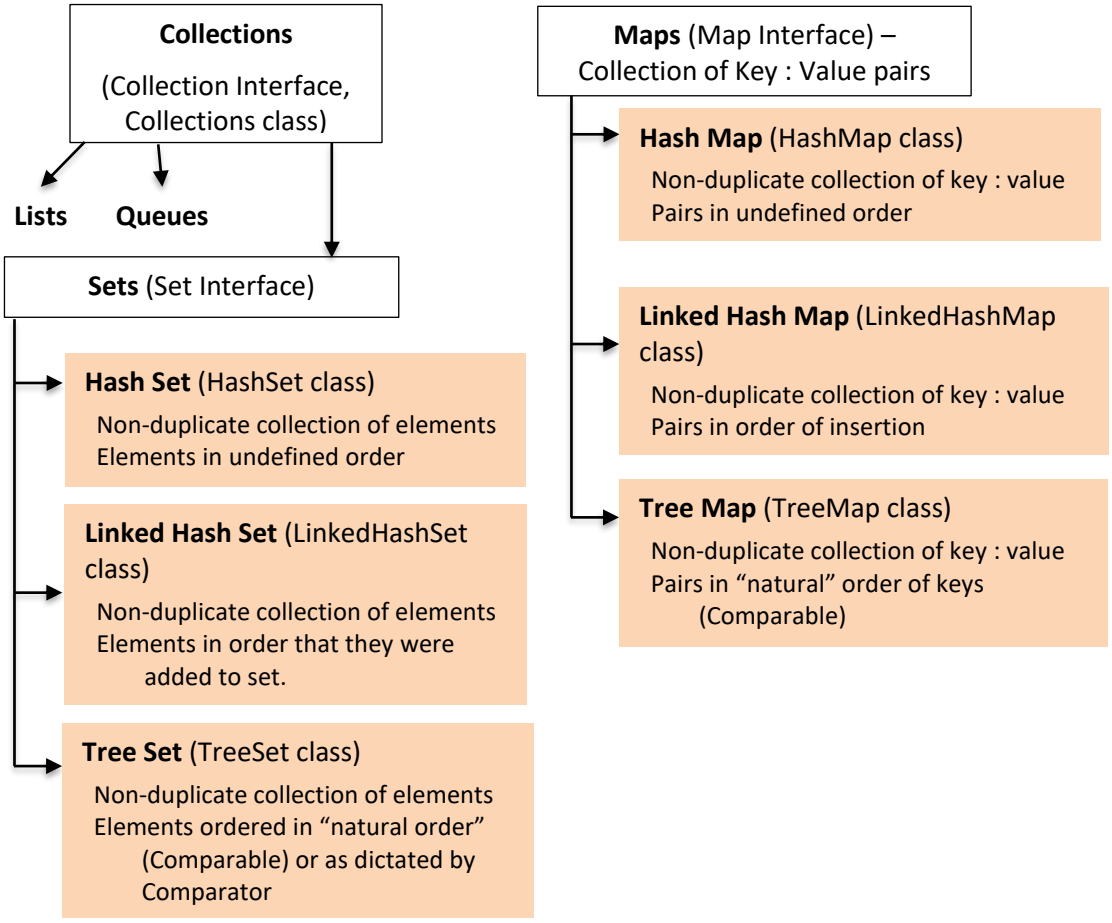
```
PriorityQueue<AConcreteType> queuename = new PriorityQueue<>();
```

The default constructor for a PriorityQueue assumes that the objects that comprise it are of a class that implements `Comparable`. If you want to order the objects in the queue using a Comparator, use the following constructor.

```
PriorityQueue(initialCapacity: int, comparator: Comparator<? super E>);
```

Chapter 21: Sets and Maps

Figure 2: Java Collections Framework. Classes Discussed in Ch. 21 are Highlighted



Syntax for Creating Set Objects

```
HashSet<AConcreteType> setname = new HashSet<>();
LinkedHashSet<AConcreteType> setname = new LinkedHashSet<>();
TreeSet<AConcreteType> setname = new TreeSet<>();
```

Note that all three set classes have constructors that allow them to be created from an existing collection. For example,

```
HashSet<AConcreteType> setname = new HashSet<>(c: Collection<?
extends AConcreteType>);
```

Differences between Set Classes

- HashSet – collection of non-duplicate objects
  - Elements are NOT stored in the set in the order in which they were inserted.
  - Most efficient set structure if you do not care about the order in which the elements are stored.

- LinkedHashSet – collection of non-duplicate objects
  - Elements are stored in the order in which they were inserted.
- TreeSet – collection of non-duplicate objects
  - Elements are sorted as they are inserted
    - Default constructor – sorts objects according to `compareTo()` method (assuming objects implement `Comparable`)
    - You can also use a constructor that takes a comparator as a parameter – sort using `compare()` method of that comparator.

Key Methods for Sets

All inherited from `Set` interface:

- `add(o: E): boolean` – Adds an element to the set, if it's not already present. Returns false if element already existed in set.
- `clear()`: Remove all elements from set.
- `size()`: Returns number of elements in the set.
- `remove(o: E): boolean` – Remove object o from the set, if present.
- `removeAll(Collection<?> c): boolean` -- Removes all elements in c from set, if present – equivalent to a set difference.
- `retainAll(Collection<?> c): boolean` -- Retains only the elements of the set which are also elements of c from set. Equivalent to a set intersection.
- `addAll(Collection<?> c): boolean` -- Adds all elements in c to the set – equivalent to a set union.
- `contains(o: E): boolean` – Returns true if object o is an element of the set.

Maps

Syntax for Creating Map Objects

```
HashMap<KeyType, ValueType> mapname = new HashMap<>();
LinkedHashMap<KeyType, ValueType> mapname = new LinkedHashMap<>();
TreeMap<KeyType, ValueType> mapname = new TreeMap<>();
```

Note that all three set classes have constructors that allow them to be created from an existing map. For example,

```
TreeMap<String, Integer> tmap = new TreeMap<>(hmap);
```

Differences Between Map Classes

- For all maps
  - Map values to keys – keys are like the "indices" that point to values.
  - Keys are not duplicated; values may be duplicated.
- HashMap – No defined order to keys
- LinkedHashMap – Ordered in the order in which the pairs were added to the map.

- TreeMap – Ordered in the “natural order” (i.e. Comparable) of the keys.

Key Methods for Maps

- `put(key, value)` – Adds the pair {key, value} to the map.
- `containsKey(key: Object): boolean` – Returns true if the map has an entry for the specified key.
- `get(key: Object): V` – Returns the value mapped to object key in this map.
- `size()` – Returns the number of {key, value} pairs in the map.

Ch. 22 Developing Efficient Algorithms

This chapter focuses on **algorithm analysis**: predicting the performance of an algorithm in a way that is independent of any specific data set, and independent of hardware or system details. Raw execution time is not a good way to do this:

- Execution time depends on system load.
- Also depends on size and specific details of the input.

Instead, focus on **how fast an algorithm's execution time increases as the input size increases** – i.e. **focus on an algorithm's growth rate**.

Big-O Notation

Suppose that the execution time (i.e. the time required to complete) of a specific algorithm operating on a data set of size ***n*** (for example, sorting an array of n elements, or searching for a specific value in a set of n numbers) is equal to ***T(n)***.

Consider the ratio of execution time for two different data-set sizes – n1 and n2.

$$\frac{T(n2)}{T(n1)}$$

This ratio represents the growth rate of the algorithm as the size of the dataset grows from n1 to n2.

Suppose further that the execution time is a linear function of n:

$$T(n) = an + c$$

where a and c are constants. Then, the growth rate of the execution time is

$$\frac{T(n2)}{T(n1)} = \frac{a \times n2 + c}{a \times n1 + c} \approx \frac{n2}{n1}$$

where the approximation holds true for large enough values of n1 and n2. In this case, the execution time grows at the same rate that the size of the data set grows. That is, if *n* doubles, the execution time $T(n)$ doubles; if *n* increases by a factor of 10, the execution time also increases by a factor of 10. In this case, we would say that **the algorithm growth rate is of the order of n, or $\boldsymbol{O(n)}$.**

Note that **the growth rate is independent of the multiplicative constant, a, and essentially independent of the additive constant c (at least for large enough n).**

Determining *O(n)* for an Algorithm

Useful summations:

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$

$$a^0 + a^1 + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = O(2^n)$$

Useful recurrence relations:

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log(n))$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(\mathrm{n})$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(\mathrm{n})$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n\log(n))$	Merge sort
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

Examples

- Example 1: Time complexity for iteration

```
for (int i = 1; i <= n; i++) {
      k = k + 5;
}
```

There is a constant time, *c*, for executing the addition statement. Since the loop is executed *n* times, the time complexity for the loop is equal to

$$T(n) = c \times n \to O(n)$$

- Example 2: Time complexity for nested loops, each of which executes *n* times.

```
for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
            k = k + i + j;
      }
}
```

The outer loop executes *n* times, and for each time the outer loop executes, the inner loop also executes *n* times. Thus, the time complexity is

$$T(n) = c \times n \times n \to O(n^2)$$

- Example 3:

```
for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= i; j++) {
            k = k + i + j;
      }
}
```

The outer loop iterates n times, but on the i'th iteration, the inner loop iterates i times. Thus, the time complexity of this pair of loops is

$$T(n) = c + 2c + 3c + \cdots + nc = c\frac{n(n+1)}{2} \to O(n^2)$$

Ch. 23 Sorting

Insertion Sort

- Time complexity: $O(n^2)$
- Sorts a list of values by repeatedly inserting a new element into a sorted sublist, until the whole list is sorted.
- Example: Suppose we start with a 3-element array that has been sorted: 2, 5, 9, and then we want to add a 4th element: 4.

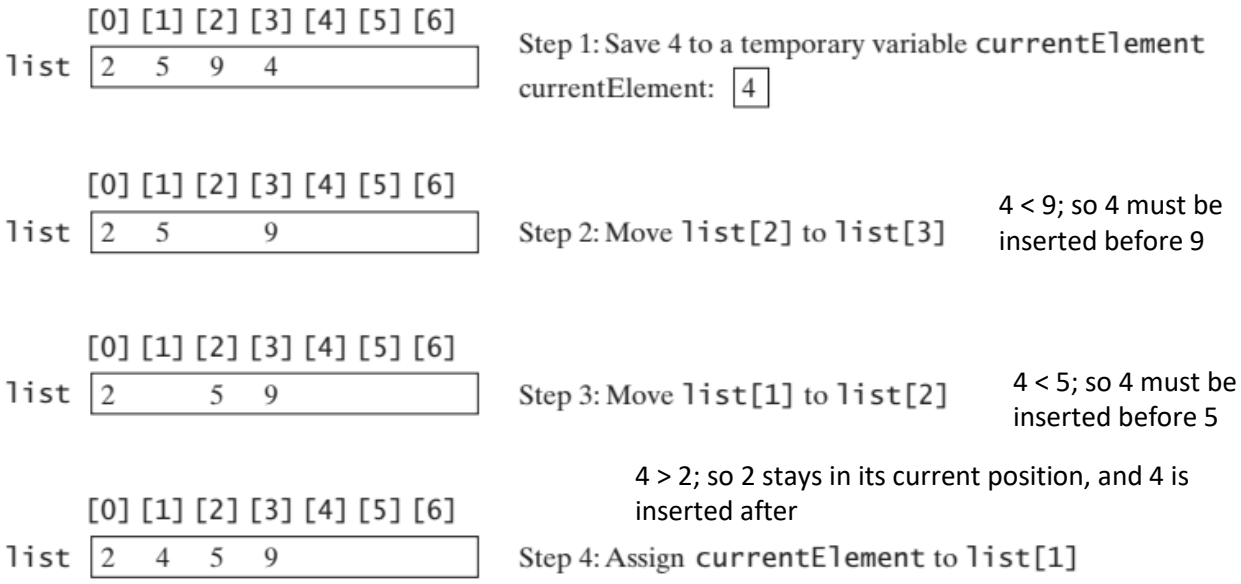


FIGURE 23.2 A new element is inserted into a sorted sublist.

Bubble Sort

- Time complexity: $O(n^2)$
- Sorts the array in multiple passes.
  - Each pass successively swaps neighboring elements if the elements are not in order.
  - At the end of the pass, the largest element in the current sublist ends up at the end of that sublist.
  - The next pass operates on the sublist of the remaining n-1 elements.
- Example: Start with an unsorted array of 6 numbers: 2 9 5 4 8 1

  Pass 1: 2 & 9 do not swap → 9 swaps with 5 → 9 swaps with 4 → 9 swaps with 8 → 9 swaps with 1

  2 5 4 8 1 9 (after pass 1)

  Pass 2: Repeats this process with the sub-array consisting of the lower 5 elements.

Merge Sort

- Time complexity: $O(nlog(n))$
- Divides the array into two halves
- Applies a merge sort on each half recursively.
- After the two halves are sorted, merge them.

Example of merging two sorted sub-lists into one larger sorted list

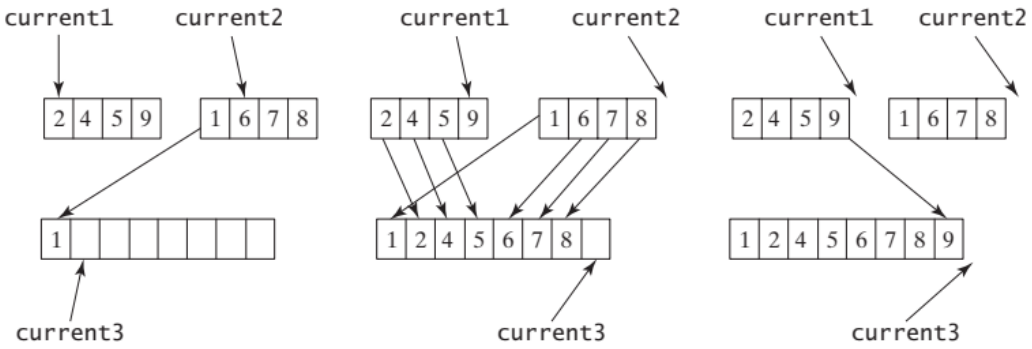


FIGURE 23.5 Two sorted arrays are merged into one sorted array.

Quick Sort

- Selects an element of the array – the pivot.
- It divides the remainder of the array into two parts – first part contains all the elements ≤ the pivot, the second part contains all the elements > the pivot.
- Recursively applies quick sort to the first half and to the second half.
- Partition list[]
    - Pick list[0] as pivot.
    - Set index low = 1, and index high = list[list.length – 1]
    - Continue until high <= low
        - Increment low until you find list[low] > pivot.
        - Decrement high until you find list[high] <= pivot.
        - Swap list[low] and list[high]
    - Decrement high until you find list[high] < pivot. Swap pivot (list[0]) and list[high].
    - Return index high – index in which original pivot value is now sitting.
- Recursively apply quick sort to list[0:high-1] and to list[high+1, list.length – 1]

Heap Sort

Key point: A heap sort uses a binary heap. It first adds all the elements to a heap and then removes the largest elements successively to obtain a sorted list.

Binary tree – a hierarchical structure with the following properties:

- It is either empty or contains an element called the ***root***.
- It also contains two distinct binary subtrees – the left subtree and the right subtree.

The ***length*** of a path is the number of edges in the path.

The ***depth*** of a node is the length of the path from the root to the node.

Complete binary tree – a binary tree in which

- Each level, except perhaps the last, is full.
- In the last level, all of the leaves are placed leftmost.

Binary heap – A binary tree with the following properties:

- It is complete.
- Each node is greater than or equal to any of its children.

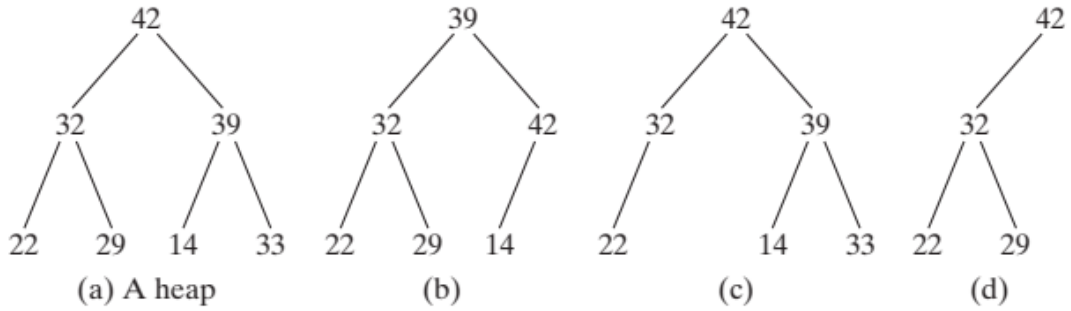


FIGURE 23.9 A binary heap is a special complete binary tree.

a and b are complete; b is not a heap

c and d are not

- A heap can be stored in an ArrayList – list[0] = root, list[1] = left child, list[2] = right child. Continue on, listing all elements of a level, from left to right, before moving on to the next level.
- Add a new node to the heap
  - Add new node to the end of the heap, and then rebuild as follows:
  - While current node > parent
    - Swap current node with parent.
    - Current now points one level up.
- Removing the root (or root of a subtree)
  - Move the last node to replace the root. Current = root.
  - While current < one of its children
    - Swap current with larger of its children.

- Current now points one level down.

Ch. 25 Binary Search Trees

Key Point: A ***binary search tree*** can be implemented using a linked structure.

Binary tree – a hierarchical structure that is either empty, or consists of an element called the root, with two distinct binary trees – the left subtree and the right subtree – either or both of which may be empty.

- ***Length*** of a path from a node to a subnode is equal to the number of edges in the path.
- The ***depth*** of a node is the length of the path from the root to the node.
- The root of a left (right) subtree of a node is called a ***left (right) child*** of the node.
- A node without children is called a ***leaf***.
- The ***height*** of a tree is the length of the path from the root to its deepest leaf.

A ***binary search tree (BST)*** with no duplicate elements is a binary tree which has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node.

The binary trees in the following figure are both BSTs.

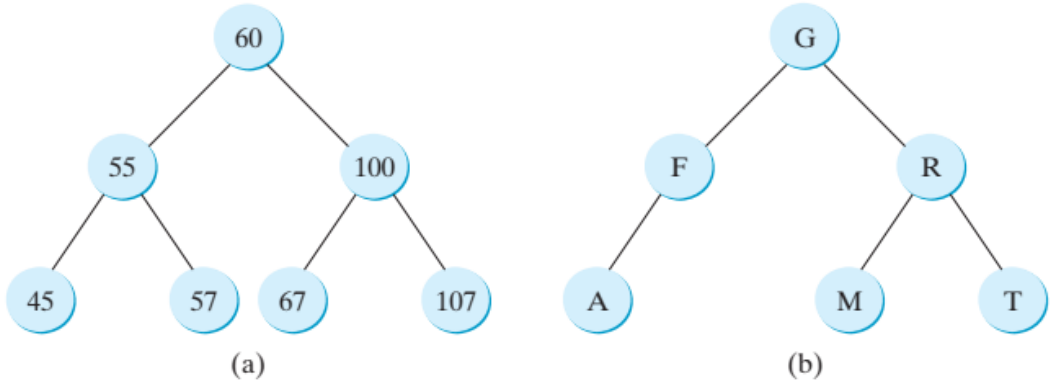


FIGURE 25.1 Each node in a binary tree has zero, one, or two subtrees.

A tree node can be defined as a class, as follows:

```
class TreeNode<E> {
      protected E element;
      protected TreeNode<E> left;
      protected TreeNode<E> right;

      public TreeNode(E e) {
            element = e;
      }
}
```

Searching for an Element in a BST

Let `current` point to the `root`. Repeat the following steps until the element matches `current.element` or `current = null`.

- If `element = current.element`, return `true`.
- If `element < current.element`, assign `current = current.left`.

- If `element > current.element`, assign `current = current.right`.
- If `current = null`, return `false`; the element is not in the tree.

Inserting an Element into a BST

Key idea – locate the parent for the new node, and make the new node one of the children of the parent.

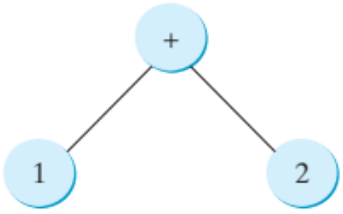
- If tree is empty, insert node as root.
- Set `parent = current = root`.
- while `current != null`
  - If `e < current.element` then
    - `parent = current`
    - `current = current.left`
  - If `e > current.element` then
    - `parent = current`
    - `current = current.right`
  - If `e = current.element`, then return false; this would be a duplicate element, and should not be inserted.

Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once.

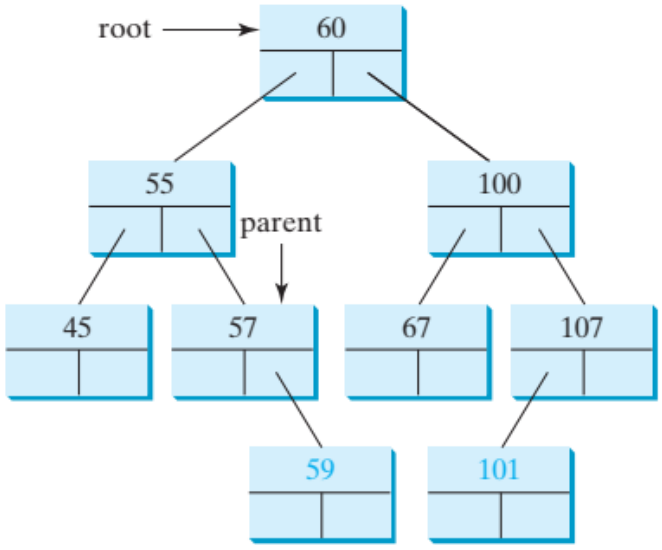
- In-order, pre-order (or depth-first), and post-order traversal.

You can use the following tree to help remember inorder, postorder, and preorder.



The inorder is 1 + 2, the postorder is 1 2 +, and the preorder is + 1 2.

- Breadth-first traversal: Nodes are visited level by level.
  - First visit the root
  - Then visit all children of the root, left to right
  - Then visit the grandchildren of the root, left to right
  - etc.



- In-order traversal: 45, 55, 57, 59, 60, 67, 100, 101, 107
- Post-order traversal: 45, 59, 57, 55, 67, 101, 107, 100, 60
- Pre-order (aka depth-first) traversal: 60, 55, 45, 57, 59, 100, 67, 107, 101
- Breadth-first traversal: 60, 55, 100, 45, 57, 67, 107, 59, 101

25.3 Deleting Elements from a BST

Key point: To delete an element from a BST, first locate it in the tree and then consider two cases – case 1 = element has a left child, or case 2 = element has no left child – before deleting the element and reconnecting the tree.

- Case 1: element has no left child

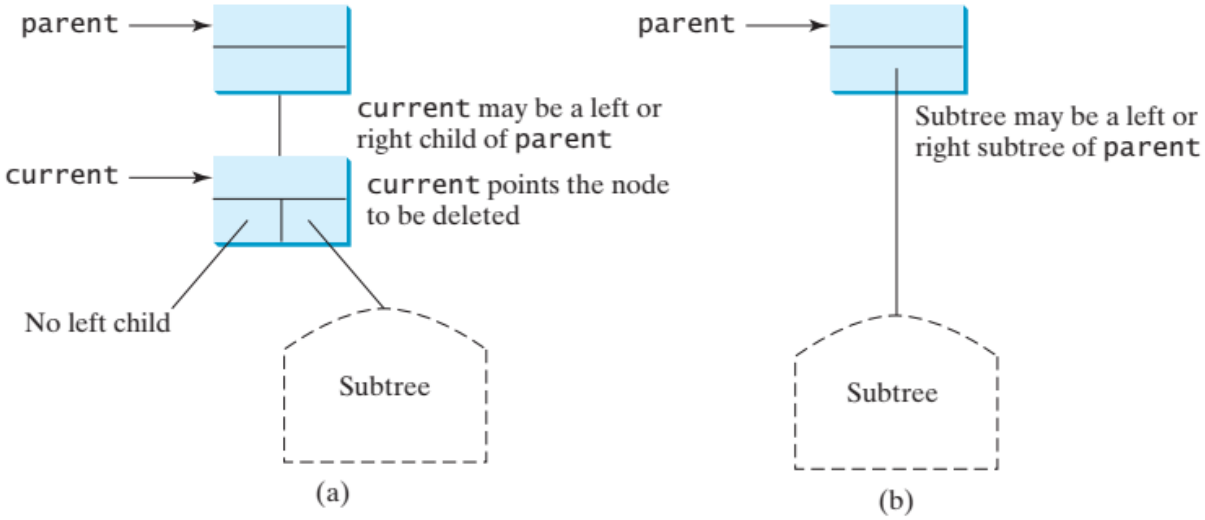


FIGURE 25.10 Case 1: The current node has no left child.

  - If the element to be deleted has no left child, then simply delete the element, and connect its right child to the same point (either left or right) of its parent.

- Case 2: Element to be deleted has a left child

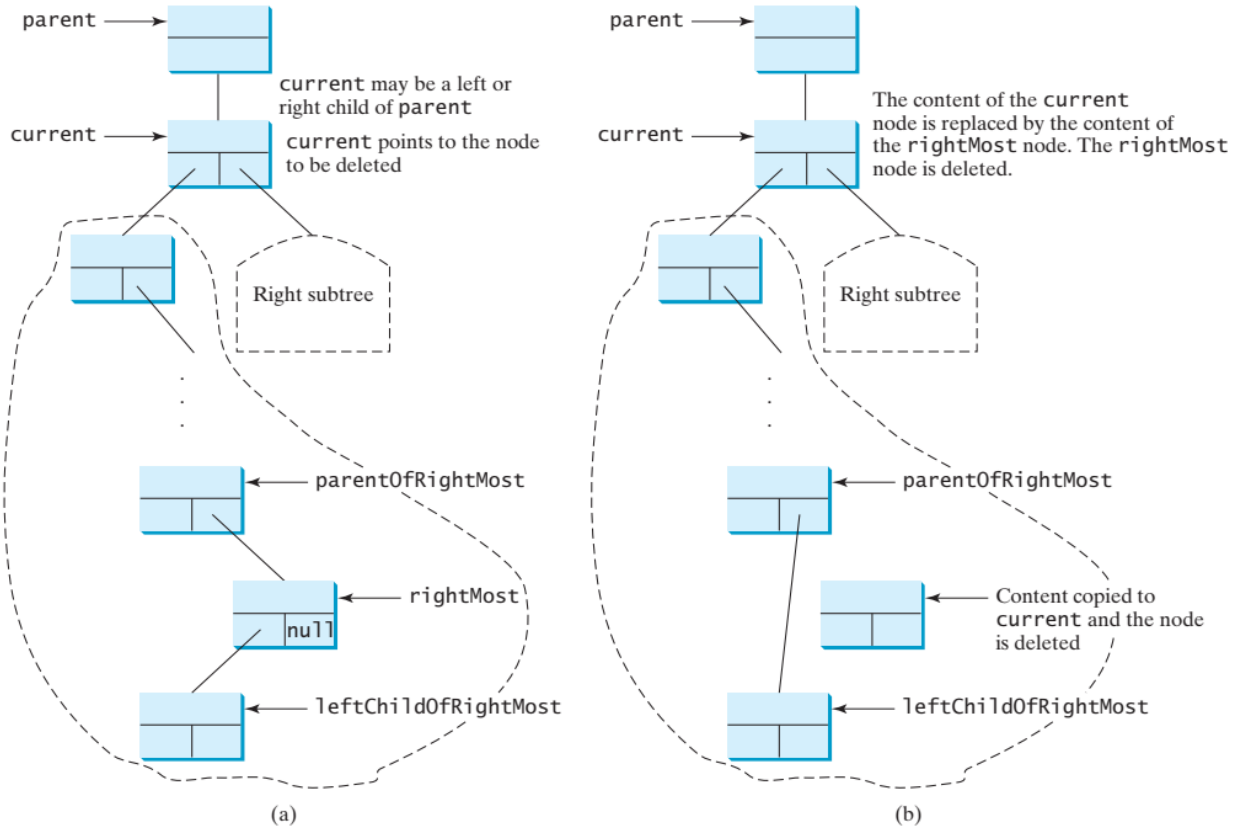


FIGURE 25.12 Case 2: The current node has a left child.

- Let `current` = the element to be deleted
- Let `rightMost` point to the node that contains the largest element of the left subtree of `current`. Also mark the parent of this node as `parentOfRightMost`.
- Replace `current` with `rightMost`, and connect `rightMost`'s left subtree to `parentOfRightMost`.

Ch. 26 AVL Trees

Key point: An ***AVL tree*** is a balanced binary search tree.

- The process for inserting and deleting is the same as in a regular BST, except that you may have to rebalance tree afterwards.
- Definition: ***Balance factor*** of a node – the height of the node's right subtree – height of left subtree.
    - AVL tree is balanced if balance factor = -1, 0, or +1.
    - ***Left-heavy*** if -1; ***Right-heavy*** if +1

Rebalancing Trees

Key point: If a node in the AVL tree becomes unbalanced, perform a rotation operation to rebalance.

- ***LL imbalance*** and rotation:
    - Occurs at a node A which has a balance factor ≤ -2, and a left child B with balance factor -1 or 0.
    - Fix by single right rotation at A.

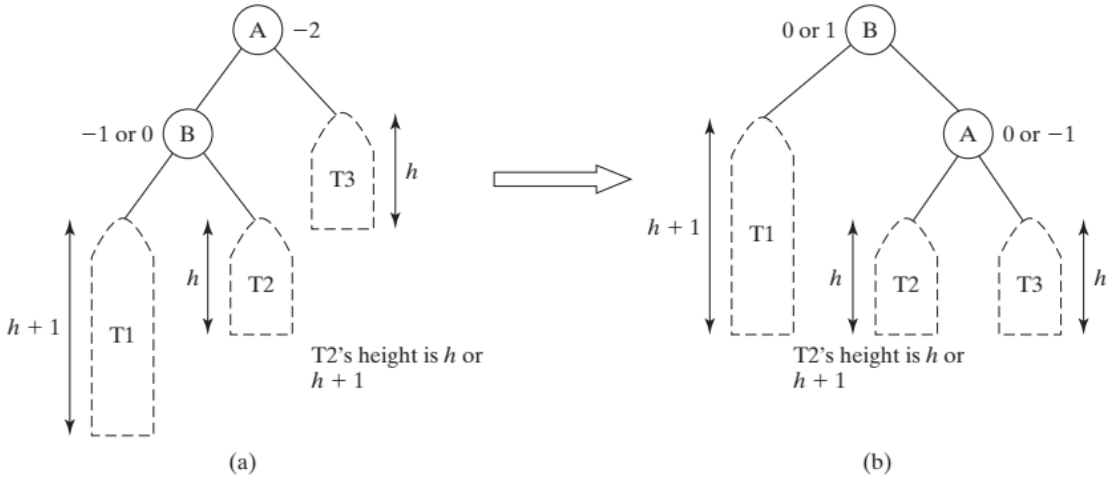


FIGURE 26.2 An LL rotation fixes an LL imbalance.

- ***RR imbalance*** and rotation:
    - Occurs at a node A which has a balance factor ≥ +2, and a right child B with balance factor +1 or 0.
    - Fixed by a single left rotation at A.

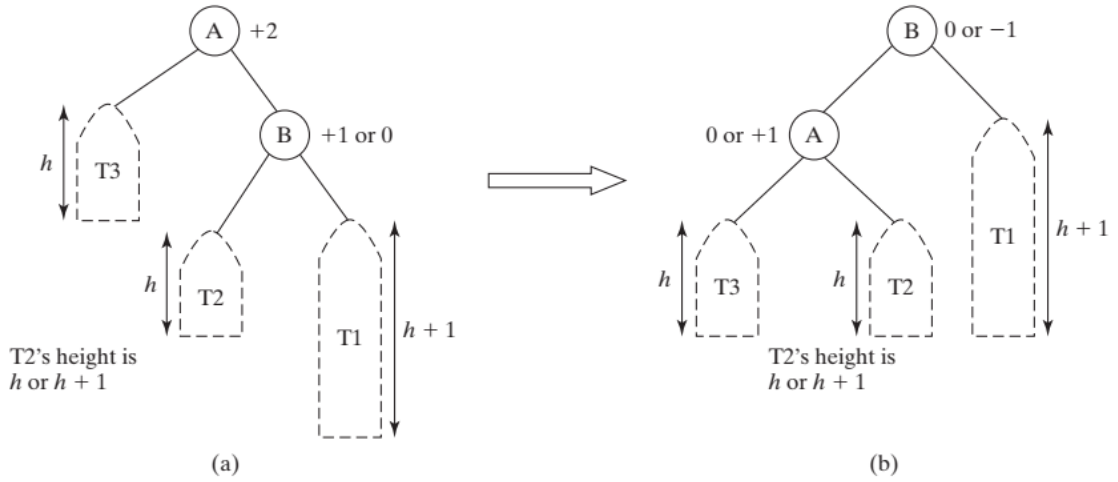


FIGURE 26.3 An RR rotation fixes an RR imbalance.

- ***LR imbalance*** and rotation:
  - Occurs at a node A which has a balance factor of -2 and a left child B with a balance factor of +1.
  - Fixed by a double rotation – left rotation at B followed by a right rotation at A.

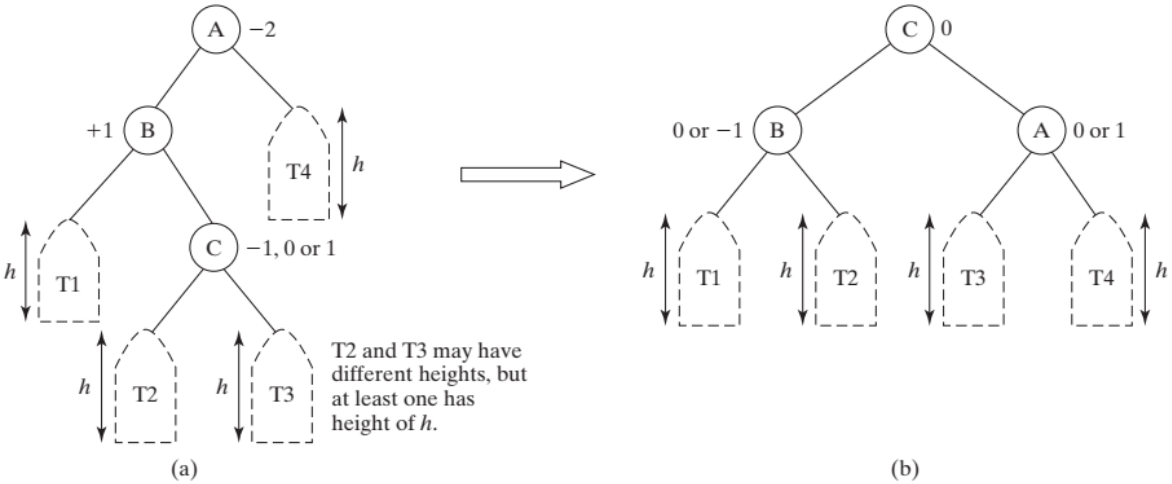


FIGURE 26.4 An LR rotation fixes an LR imbalance.

- ***RL imbalance*** and rotation:
  - Occurs at a node A which has a balance factor of +2 and a right child B with a balance factor of -1.
  - Fixed by a double rotation – right rotation at B followed by left rotation at A.

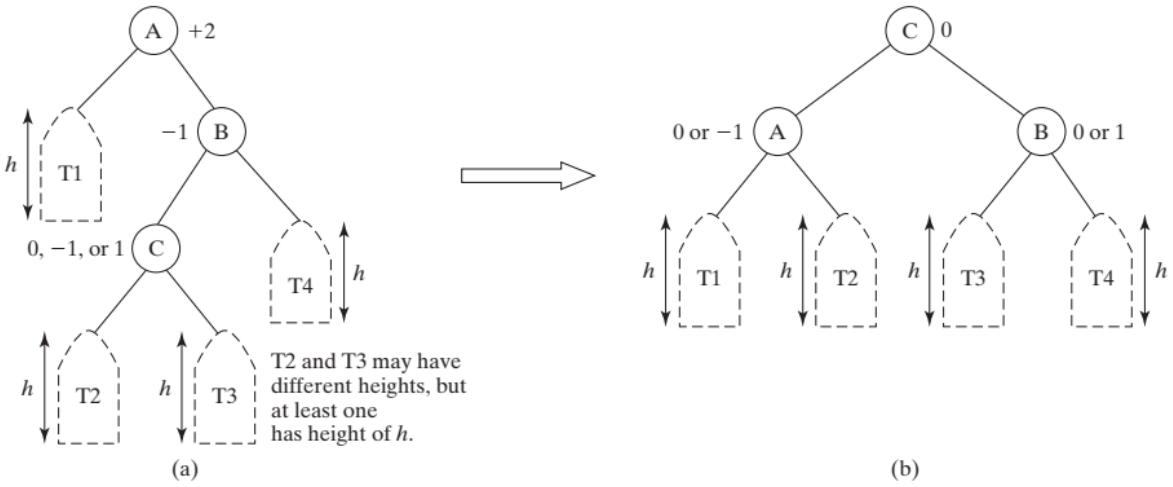


FIGURE 26.5 An RL rotation fixes an RL imbalance.

Designing Classes for AVL Trees

- `AVLTree<E extends Comparable<E>>` extends `BST<E extends Comparable<E>>`
- `AVLTreeNode<E>` extends `TreeNode<E>` and is an inner class of `AVLTree`
- Note that the `createNewNode`() method in `AVLTree` overrides the method in the `BST` class.
- The `insert`() and `delete`() methods in `BST` are overridden in `AVLTree` in order to ensure that the tree is balanced.

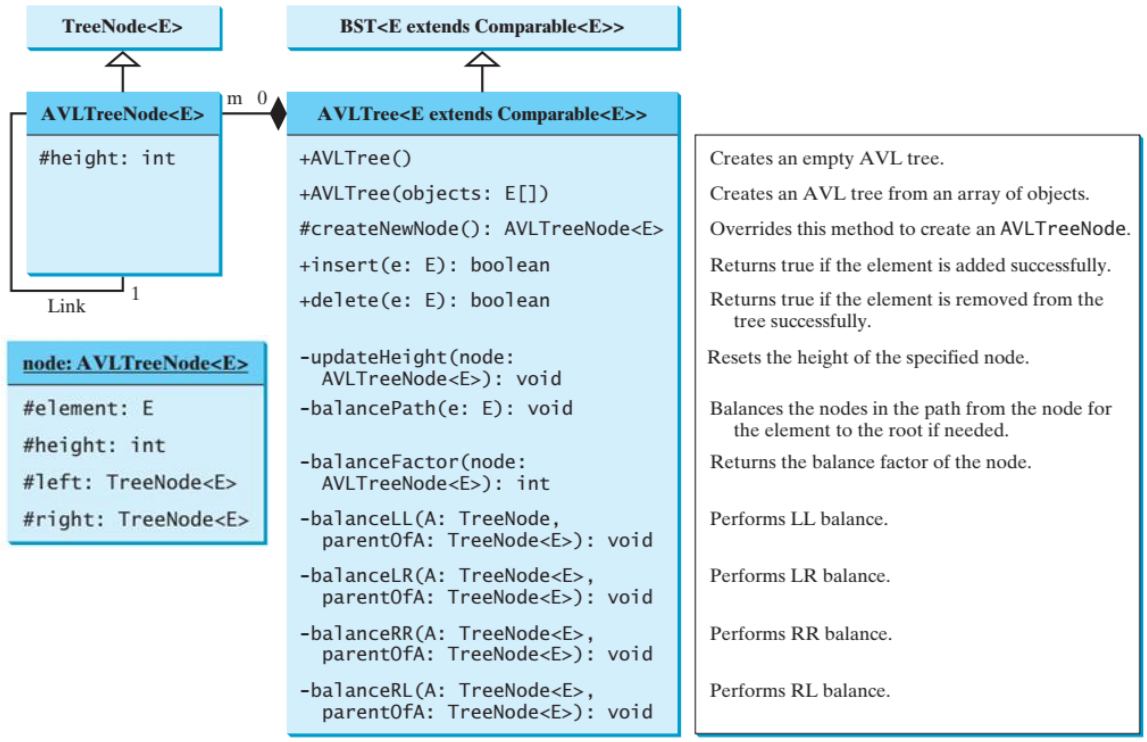


FIGURE 26.7 The `AVLTree` class extends `BST` with new implementations for the `insert` and `delete` methods.

Overriding the insert Method

- A new element is always inserted into an AVL tree (just as into a BST) as a leaf node.
- Perform appropriate rotations using the algorithms described previously.

AVL Tree Time Complexity Analysis

Key point: Since the height of an AVL tree is $O(\log(n))$, the time complexity of `search`, `insert`, and `delete` methods in `AVLTree` is $O(\log(n))$.

Ch. 27 Hashing

Key points:

- ***Hashing*** is superefficient. It takes $O(1)$ time to search, insert, and delete an element.
- It uses a ***hashing function*** to map a key to an index.

Note that if you know the index of an element in an array, you can retrieve the element using the index in $O(1)$ time. We can implement a hash map using an array and a hashing function:

- The array stores the values in what is called a ***hash table***.
- A hash function maps the keys into indices in the hash table.

Ideally, we would implement a perfect hash function to map every search key to a different index, but perfect hash functions are difficult to find.

When two or more keys are mapped to the same hash value, this is called a ***collision***.

Hash Functions and Hash Codes

A typical hash function:

- First converts a search key to an integer value called a ***hash code***.
- The compresses the hash code into an index to the hash table.

Java's `Object` class has the `hashCode` method.

- Returns an integer hash code (by default, the memory address for the object).
- When overriding the `hashCode` method for classes that you create
    - Override `hashCode` whenever `equals` is overridden to ensure that two equal objects return the same hash code.
    - Invoking the `hashCode` method multiple times returns the same integer, provided the object's data has not changed.
    - Two unequal objects may have the same hash code, but you should try to implement the `hashCode` method to minimize these collisions.

Hash Codes for Primitive Types

Primitive Type	Hash Code
`char, byte, short, int`	Cast each of these as `int`; the int value is the hash code
`long`	`int hashCode = (int)(key ^ (key >> 32));` This is a bitwise exclusive OR of the 32 MSBs and 32 LSBs of the long (64-bit) integer. This procedure is called "folding."
`float`	`int hashCode = Float.floatToIntBits(key)` Use the floatToIntBits() function to convert the 4-byte float to a 4-bit int.
`double`	`long bits = Double.doubleToLongBits(key);` `int hashCode = (int)(bits ^ (bits >> 32));`

	Convert double to a long integer, and then use folding to get hash code from that integer
`String`	The hash code is equal to: $$s_0 b^{n-1} + s_1 b^{n-2} + \cdots + s_{n-1}$$ Where `si` is the i'th character in the string. Remember that a character is a 1-byte integer.

Compressing Hash Codes

The hash code functions listed in the previous section return a 4-byte (32-bit) integer.

Since most hash sets or hash maps will contain much less than $2^{32} - 1$ keys, the hash code must be ***compressed*** before using the number to map the key-value pair to an index of an array.

Typical compression functions are:

```
h(hashCode) = hashCode % N;
```

In `java.util.HashMap`, `N` is set to a value that is a power of 2. In this case, the % N operation can be efficiently performed as follows:

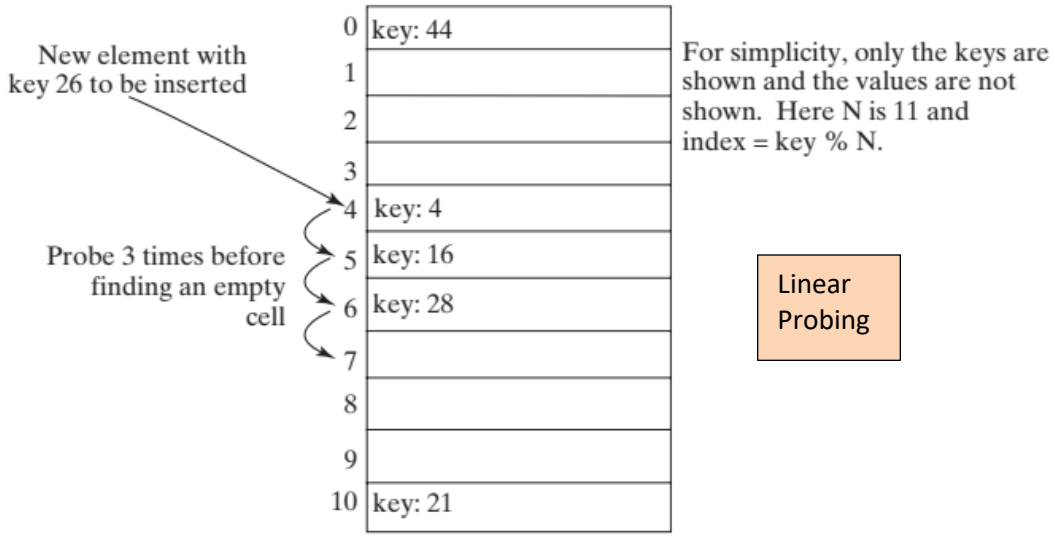
```
h(hashCode) = hashCode % N = hashCode & (N - 1);
```

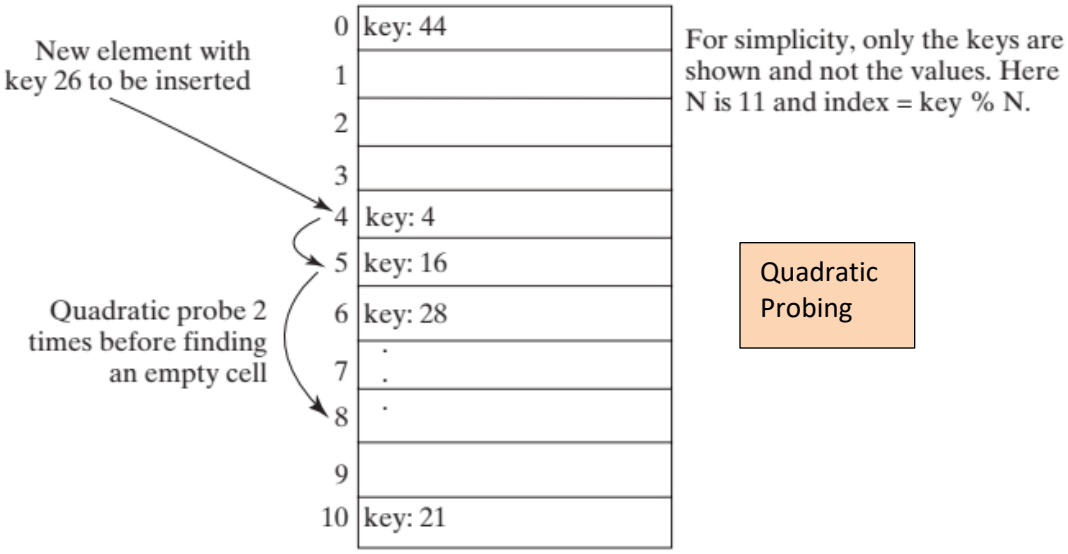
Handling Collisions

- A **collision** occurs when two keys are mapped to the same index in a hash table.
- **Open addressing** – the process of finding an open location in the hash table in the event of a collision – is one way of dealing with these.

Open addressing algorithms:

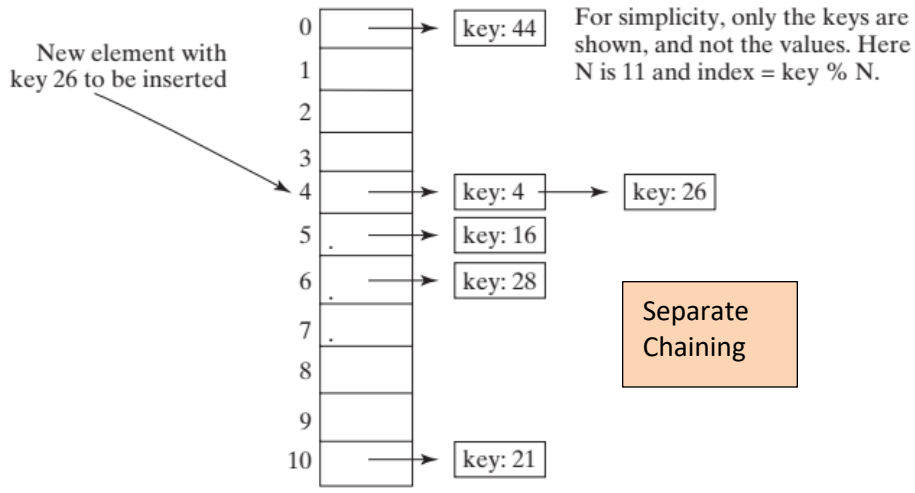
- Linear probing – one step at a time, until you find the next open location in the array
- Quadratic probing – additional steps of 1, 4, 9, 16, etc.
- Double hashing -- uses a secondary hash function `h'(key)` to determine the increments. That is, indexes `(k + j*h'(key)) % N` for `j ≥ 0`.





Handling Collisions Using Separate Chaining

- The separate chaining scheme places all entries with the same hash index in the same location.
- Each location uses a bucket to hold multiple entries.
- You can implement a bucket using an array, ArrayList, or LinkedList.



Load Factor and Rehashing

- The ***load factor*** measures how full a hash table is.

$$\lambda = \frac{n}{N}$$

where n is the number of elements currently in the table, and N is the size of the table.

- If the load factor is exceeded, increase the hash table size, and reload the entries into the larger table – ***rehashing***.

The book recommends

- Keep $\lambda < 0.5$ for any of the open addressing schemes
- Keep $\lambda < 0.9$ for separate chaining.

Ch. 28 – Graphs and Applications

Basic Graph Terminologies

- A ***graph*** consists of ***vertices*** and ***edges*** that connect the vertices.
  - A nonempty set of vertices
  - A set of edges that connect the vertices
- Directed vs. undirected
  - In a ***directed*** graph, each edge has a direction.
  - In an ***undirected*** graph, you can move between vertices through an edge in either direction.
- Edges may be ***weighted*** for ***unweighted***.
- Adjacency
  - **Vertices are *adjacent*** if they are connected by the same edge. Adjacent vertices are also said to be ***neighbors***.
  - **Edges are *adjacent*** if they are connected to the same vertex.
- ***Loop*** – an edge that links a vertex to itself.
- ***Simple graph*** – one that has no loops for parallel edges between vertices.
- ***Complete graph*** – a graph that includes an edge between every pair of vertices.
- ***Connected*** – a graph is connected if there exists a path between every pair of vertices.
- ***Cycle*** (or ***circuit***) – a closed path in a graph that starts and ends at the same vertex.
- ***Tree*** – a connected graph that has no cycles.
- ***Spanning tree*** – a subgraph of a graph G that is a tree, and that includes all vertices.

Representing Graphs

- The vertices can be stored in an array or a list.
- Can be objects of any type (e.g. city object containing the name, population, and mayor of the city).

Outline of Topics by Chapter

Ch. 22 Developing Efficient Algorithms

- Big-O notation refers to algorithm growth rate.
- How is it related to execution time, and why do we use big-O notation to specify algorithm efficiency rather than execution time?
- Summations that are useful in analyzing algorithm growth rate

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2} = O(n^2)$$

$$a^0 + a^1 + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = O(2^n)$$

- Useful recurrence relations:

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log(n))$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(\mathrm{n})$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(\mathrm{n})$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n\log(n))$	Merge sort
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

- Examples of time-complexity analysis: https://www.geeksforgeeks.org/miscellaneous-problems-of-time-complexity/

Ch. 23 Sorting

- Make sure you can sort a small list using each of the following methods
  - Insertion sort -- $O(n^2)$
  - Bubble sort -- $O(n^2)$
  - Merge sort -- $O(nlog(n))$
  - Quick sort -- $O(nlog(n))$

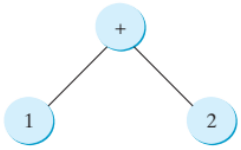
    - Heap sort – build a binary heap (see below), and then iterate as follows, until there are no more elements left in the heap:
        - Remove the root – this is the current largest remaining element in the heap.
        - Rebuild the heap using the process described below.

- What is a heap? A binary tree that satisfies the following properties:
    - It is complete (every level but the last is fully populated; the last level is populated from the left side).
    - Each parent node is greater than each of its children.
- How to add a new node to the heap?
- Add a node to the heap – add the element at the next open slot in the last row, then rebuild:
    - While current node > parent
        - Swap current node with parent.
        - Current now points one level up.
- Removing the root (or root of a subtree)
    - Move the last node added to the last row to replace the root. Current = root.
    - While current < one of its children, swap current with larger of its children.
    - Current now points one level down.

Ch. 25 – Binary Search Trees

- What is a Binary Search Tree (BST)? A binary tree with no duplicate elements, which has the property that for every node in the tree
    - The value of any node in its left subtree is less than the value of the node.
    - The value of any node in its right subtree is greater than the value of the node.
- How to insert a new node into the BST
- How to delete an element from a BST – two cases
    - The element to be deleted has no left child
    - The element has a left child
- Traversing the tree
    - in-order
    - pre-order
    - post-order

You can use the following tree to help remember inorder, postorder, and preorder.



The inorder is 1 + 2, the postorder is 1 2 +, and the preorder is + 1 2.

- Breadth-first traversal

Ch. 26 AVL Trees

- For any BST, the **balance factor** of any node = height of right subtree – height of left subtree
- An **AVL Tree** is a balanced BST – every node has a balance factor of -1, 0, or +1.
- After adding a new node to the AVL tree, if any node becomes imbalanced (i.e. has a balance factor outside the range of [-1, 1]), then execute the proper rotation to rebalance
  - LL imbalance; LL rotation
  - LR imbalance; LR rotation
  - RL imbalance; RL rotation
  - RR imbalance; RR rotation
- Insertion and deletion of elements from AVL – same process as BST, followed by rebalancing, as necessary.
- Since the height of an AVL tree is $O(\log(n))$, the time complexity of `search`, `insert`, and `delete` methods in `AVLTree` is $O(\log(n))$.

Ch. 27 Hashing

- ***Hashing*** enables operations with a HashSet or HashMap to be superefficient. It takes $O(1)$ time to search, insert, and delete an element.
- The array stores the values in what is called a ***hash table***.
- A hash function maps the keys into indices in the hash table.
  - First converts a search key to an integer value called a ***hash code***.
  - The compresses the hash code into an index to the hash table. Typical compression function:

```
h(hashCode) = hashCode % N;
```

- When two or more keys are mapped to the same hash value, this is called a ***collision***.

Java's `Object` class has the `hashCode` method. The method returns an `int`.

Primitive Type	Hash Code

`char, byte, short, int`	Cast each of these as `int`; the int value is the hash code
`long`	`int hashCode = (int)(key ^ (key >> 32));` This is a bitwise exclusive OR of the 32 MSBs and 32 LSBs of the long (64-bit) integer. This procedure is called "folding."
`float`	`int hashCode = Float.floatToIntBits(key)` Use the floatToIntBits() function to convert the 4-byte float to a 4-bit int.

`double`	`long bits = Double.doubleToLongBits(key);`
	`int hashCode = (int)(bits ^ (bits >> 32));`
	Convert double to a long integer, and then use folding to get hash code from that integer

`String`	The hash code is equal to: $$s_0 b^{n-1} + s_1 b^{n-2} + \cdots + s_{n-1}$$ Where s_i is the i'th character in the string. Remember that a character is a 1-byte integer.

- Collisions are handled through:
    - Open addressing
        - Linear probing – one step at a time, until you find the next open location in the array
        - Quadratic probing – additional steps of 1, 4, 9, 16, etc.
        - Double hashing -- uses a secondary hash function `h'(key)` to determine the increments.
    - Separate chaining
        - The separate chaining scheme places all entries with the same hash index in the same location.
        - Each location uses a bucket to hold multiple entries.
- When hash table becomes too full, table size is increased (typically doubled), and all elements / keys are rehashed.