Computer Programming II (CS-141)

====== Chapter 9 ======

Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- *All methods in an interface type are **abstract**; they don't have an implementation*
- *All methods in an interface type are automatically public*
- *An interface type does not have instance fields*

Declaring an Interface

- *Interfaces are tagged with a "stereotype" indicator «**interface***
  - EX: Public class interface car {
- Use implements word to indicate that a class implements an interface type
  - EX: public class bank implements car {
- A class can implement more than one interface type

Inner Classes

Classes can be declared inside method or inside other classes. If inner class is declared inside an enclosing class, but outside its methods, it is available to all methods of enclosing class. Compiler turns an inner class into a regular class file.

An *inner class is a class that is defined inside another class. Why would you want to do that?*

- There are four reasons:
  - An object of an inner class can access the implementation of the object that created
  - it—including data that would otherwise be private.
  - Inner classes can be hidden from other classes in the same package.
  - *Anonymous inner classes are handy when you want to define callbacks on the fly.*
  - Inner classes are very convenient when you are writing event-driven programs.

A **mock object** provides the same services as another object, but in a simplified manner.

====== Chapter 10 ======

Inheritance (OOP):
Superclass: more general class
Subclass: more specialized class that inherits from the superclass.
A subclass has no access to private instance variables of its superclass
Inheritance is a mechanism for extending existing classes by adding instance variables and methods.

Method overriding allows a subclass or child class to provide a specific implementation of a **method** that is already provided by one of its super classes or parent classes.

Method Overloading allows a class to have two or more methods having same name, if their argument lists are different.
Use the super reserved word to call a method of the superclass.
To call the superclass constructor, use the super reserved word in the first statement of the subclass constructor.

EX: class SavingsAccount extends BankAccount

Polymorphism: Ability to treat objects with differences in behavior in a uniform way.
Protected features can be accessed by all subclasses and by all classes in the same package. Protected data can be accessed by all methods of classes in the same package. It is best to leave all data private and provide accessor methods for the data.

All classes defined without an explicit extends clause automatically extend *Object*. Most useful methods:

```
String toString()
boolean equals(Object otherObject)
```

====== Chapter 12 ======

The Software Life Cycle: Encompasses all activities from initial analysis until obsolescence.

Formal process for software development describes phases of the development process and gives guidelines for how to carry out the phases.

Development process consists of following five phases.

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment

The waterfall model:

- **Sequential process of analysis, design, implementation, testing, and deployment**

The spiral model:

- Breaks development process down into multiple phases
- Early phases focus on the construction of *prototypes*
- Lessons learned from development of one prototype can be applied to the next iteration
- Problem: Can lead to many iterations, and process can take too long to complete

Advantages of Spiral model:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.

Disadvantages of Spiral model:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project’s success is highly dependent on the risk analysis phase.
- Doesn’t work well for smaller projects.

When to use Spiral model:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Users are unsure of their needs
- Requirements are complex

Relationships Between Classes

Inheritance

- *Is-a* relationship
- Every savings account is a bank account

Aggregation

- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Every car has a tire (in fact, it has four)

Dependency

- *Uses* relationship
- Example: Many of our applications depend on the Scanner class to read input

====== Chapter 13 ======

RECURSION

- A recursive computation solves a problem by using the solution of the same problem with simpler values
- recursion is a method that is calling it's self until a condition is met
- For recursion to terminate, there must be special cases for the simplest inputs

Iteration

- • iteration like for, while, do...while loops and they are control structures and they might be part of a method

====== CHAPTER 14 ======

SELECTION SORT

- Sorts an array by repeatedly finding the smallest element of
  the unsorted tail region and moving it to the front
- Slow when run on large data sets
- Example: sorting an array of integers
- Doubling the size of the array more than doubles the time needed to sort it

PERFORMANCE OF SELECTION SORT

- The number of visits is of the order n^2
- Using big-Oh notation: The number of visits is $O(n^2)$
- Multiplying the number of elements in an array by **2** multiplies the processing time by **4**
- Big-Oh notation "$f(n) = O(g(n))$"
  expresses that f grows no faster than g
- To convert to big-Oh notation: Locate fastest-growing term, and ignore constant coefficient

QUESTION: If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?
Answer: It takes about 100 times longer.

MERGE SORT

- Sorts an array by
  - *Cutting the array in half*
  - *Recursively sorting each half*
  - *Merging the sorted halves*
- Dramatically faster than the selection sort

- Merge the two sorted arrays into a single sorted array

SEARCHING

- **Linear search: also called sequential search**
- **Examines all values in an array until it finds a match or reaches the end**
- **Number of visits for a linear search of an array of *n* elements:**
    - ***The average search visits n/2 elements***
    - ***The maximum visits is n***
- **A linear search locates a value in an array in O(*n*) steps**

BINARY SEARCH

- **Locates a value in a sorted array by**
    - ***Determining whether the value occurs in the first or second half***
    - ***Then repeating the search in one of the halves***

Question: Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?
Answer: You would search about 20. (The binary log of 1,024 is 10.)

Stack

- Collection of items *last in, first out* or LIFO order
- New items are added to the top of the stack
- Items are removed at the top of the stack
- Addition and removal operations are called push and pop

Queue

- Queues store items in a *first in, first out* or FIFO fashion
- Add items to one end of the queue (the tail)
- Remove items from the other end of the queue (the head)
- Items are removed in the same order in which they have been added
- Addition and removal operations are called add and remove
- peek to get the head element of the queue without removing it

Stacks and Queues: Uses in Computer Science

- Queue
  - Event queue of all events
  - Queue of print jobs
- Stack
  - Run-time stack that a processor or virtual machine keeps to organize the variables of nested methods

Linked Lists

- A linked list consists of a number of nodes, each of which has a reference to the next node
- Adding and removing elements in the middle of a linked list is efficient
- Visiting the elements of a linked list in sequential order is efficient
- Random access is not efficient

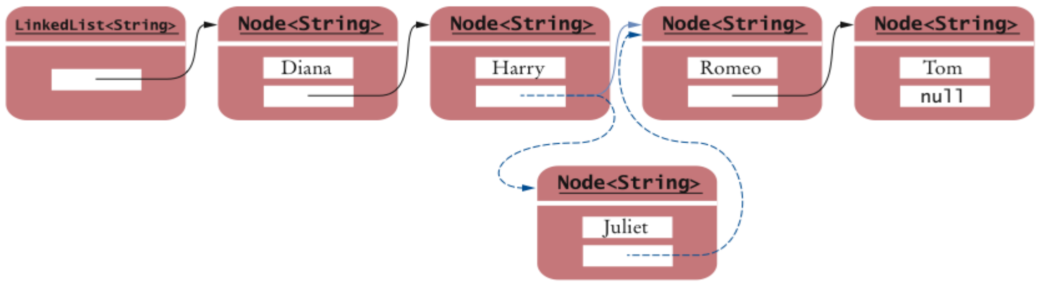


Figure 1 Inserting an Element into a Linked List

Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements (see Table 2 on page 634).

List Iterator

- ListIterator type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

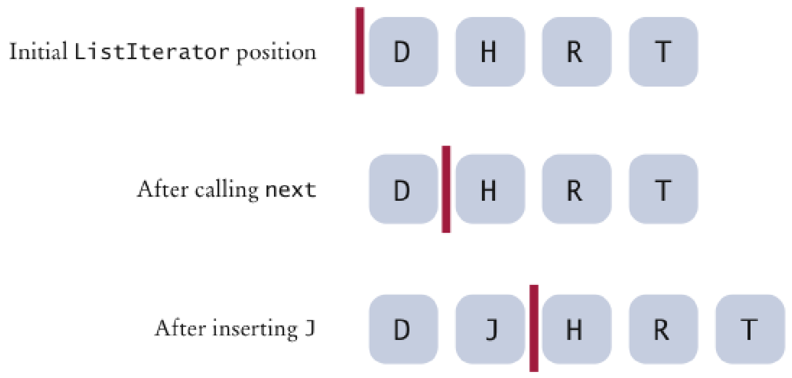


Figure 3 A Conceptual View of the List Iterator

<u>The add method:</u>

- *Adds an object after the iterator*
- *Moves the iterator position past the new element:*

<u>The remove method</u>

- *Removes and returns the object that was returned by the last call to next or previous*

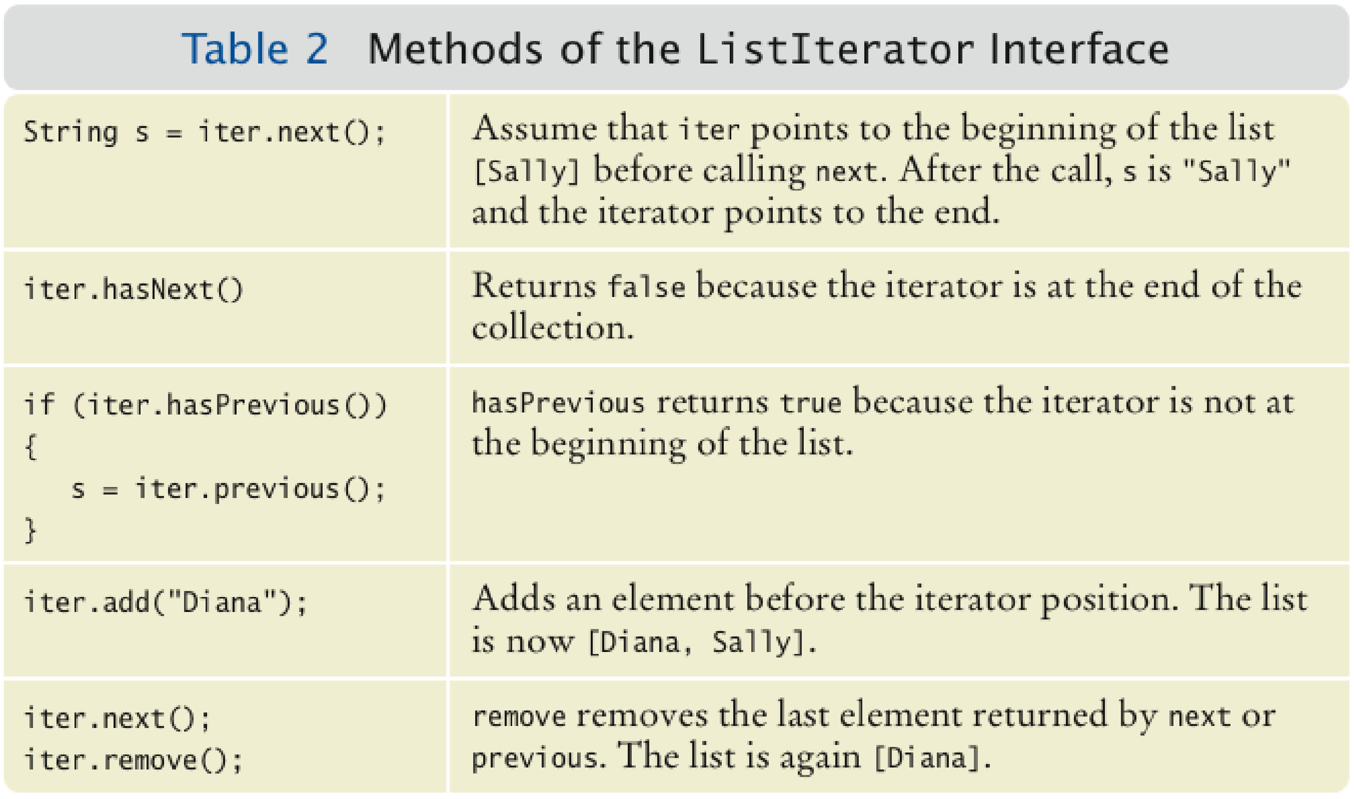


Table 2 Methods of the ListIterator Interface

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`   s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.

`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.

`iter.next();`
`iter.remove();`
	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

Exercise

Do linked lists take more storage space than arrays of the same size?

Answer

Yes, for two reasons. You need to store the node references, and each node is a separate object. (There is a fixed overhead to store each object in the virtual machine.)

There are two ways of looking at a linked list

- *To think of the concrete implementation of such a list*
  - Sequence of node objects with links between them
- *Think of the abstract concept of the linked list*
  - Ordered sequence of data items that can be traversed with an iterator

There are two ways of looking at an array list

- Concrete implementation: A partially filled array of object references
- Abstract view: Ordered sequence of data items, each of which can be accessed by an integer index

Abstract Data Types

- ArrayList: Combines the interfaces of an array and a list
- Both ArrayList and LinkedList implement an interface called List
  - List *defines operations for random access and for sequential access*

Abstract list

- *Ordered sequence of items that can be traversed sequentially*
- *Allows for insertion and removal of elements at any position*

Abstract array

- *Ordered sequence of items with random access via an integer index*

Exercise

What is the advantage of viewing a type abstractly?

Answer

You can focus on the essential characteristics of the data type without being distracted by implementation details.

Exercise

How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

Answer

To locate the middle element takes *n* / 2 steps. To locate the middle of the subinterval to the left or right takes another *n*/4 steps. The next lookup takes *n* / 8 steps. Thus, we expect almost *n* steps to locate an element. At this point, you are better off just making a linear search that, on average, takes *n* / 2 steps.

Set

- Unordered collection of distinct elements
- Elements can be added, located, and removed
- Sets don't have duplicates

Operations

- Adding an element
  - *Adding an element has no effect if the element is already in the set*
- Removing an element
  - *Attempting to remove an element that isn't in the set is silently ignored*
- Containment testing (Does the set contain a given object?)
- Listing all elements (in arbitrary order)

We could use a linked list to implement a set

- *Adding, removing, and containment testing would be relatively slow*
- There are data structures that can handle these operations much more quickly
  - ***Hash tables***
  - ***Trees***
- Standard Java library provides set implementations based on both data structures
  - ***HashSet***
  - ***TreeSet***
- Both of these data structures implement the Set interface

- As a rule of thumb, use a hash set unless you want to visit the set elements in sorted order
- **TreeSet: listed in sorted order**
- **HashSet: listed in unsorted order**

Iterator

- Use an iterator to visit all elements in a set
- A set iterator does not visit the elements in the order in which they were inserted
- An element cannot be added to a set at an iterator position
- A set element can be removed at an iterator position

Exercise

Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?

Answer

Efficient set implementations can quickly test whether a given element is a member of the set.

Exercise

Why are set iterators different from list iterators?

Answer:

Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backwards.

Exercise

When would you choose a tree set over a hash set?

Answer

When it is desirable to visit the set elements in sorted order.

Maps

- A map keeps associations between key and value objects
- Mathematically speaking, a map is a function from one set, the *key set*, to another set, the *value set*
- Every key in a map has a unique value
- A value may be associated with several keys
- Classes that implement the Map interface
  - ***HashMap***
  - ***TreeMap***

- Both of these data structures implement the Map interface

Exercise

What is the difference between a set and a map?

Answer:

A set stores elements. A map stores associations between keys and values.

Exercise

Why is the collection of the keys of a map a set?

Answer

The ordering does not matter, and you cannot have duplicates.

- **Hashing** can used to find elements in a data structure quickly without making a linear search
- A **hash table** can be used to implement sets and maps
- A **hash function** computes an integer value (called the **hash code**) from an object
- A good hash function minimizes *collisions* — identical hash codes for different objects
- To compute the hash code of object x: **int h = x.hashCode();**

Problems with Simplistic Implementation

- It is not possible to allocate an array that is large enough to hold all possible integer index positions
- It is possible for two different objects to have the same hash code

Solution

- Pick a reasonable array size and reduce the hash codes to fall inside the array

```
int h = x.hashCode();
if (h < 0) h = -h;
position = h % buckets.length;
```

- When elements have the same hash code:
  - *Use a node sequence to store multiple objects in the same array position*
  - *These node sequences are called buckets*

Algorithm for Finding an Object x in a Hash Table

1. Get the index h into the hash table
   - *Compute the hash code*
   - *Reduce it modulo the table size*
2. Iterate through the elements of the bucket at position h

  - *For each element of the bucket, check whether it is equal to x*

3. If a match is found among the elements of that bucket, then x is in the set
    - *Otherwise, x is not in the set*

<u>Hashtables</u>

- A hash table can be implemented as an array of buckets
- Buckets are sequences of nodes that hold elements with the same hash code
- If there are few collisions, then adding, locating, and removing hash table elements takes constant time
    - *Big-Oh notation: O(1)*
- For this algorithm to be effective, the bucket sizes must be small
- The table size should be a prime number larger than the expected number of elements
    - *An excess capacity of 30% is typically recommended*

<u>Adding an element: Simple extension of the algorithm for finding an object</u>

- *Compute the hash code to locate the bucket in which the element should be inserted*
- *Try finding the object in that bucket*
- *If it is already present, do nothing; otherwise, insert it*

<u>Removing an element is equally simple</u>

- *Compute the hash code to locate the bucket in which the element should be inserted*
- *Try finding the object in that bucket*
- *If it is present, remove it; otherwise, do nothing*

<u>Exercise</u>

If a hash function returns 0 for all values, will the HashSet work correctly?

<u>Answer</u>

Yes, the hash set will work correctly. All elements will be inserted into a single bucket.

<u>Exercise</u>

What does the hasNext method of the HashSetIterator do when it has reached the end of a bucket?

Answer

It locates the next bucket in the bucket array and points to its first element.

Binary search trees

- Binary search trees allow for fast insertion and removal of elements
- They are specially designed for fast searching
- All nodes in **a binary search tree** fulfill the property that:
    - *Descendants to the left have smaller data values than the node data value*
    - *Descendants to the right have larger data values than the node data value*

A binary tree consists of two nodes, each of which has two child nodes

Balanced tree: each node has approximately as many descendants on the left as on the right

- If a binary search tree is balanced, then adding an element takes $O(\log(n))$ time
- If the tree is unbalanced, insertion can be slow
    - *Perhaps as slow as insertion into a linked list*

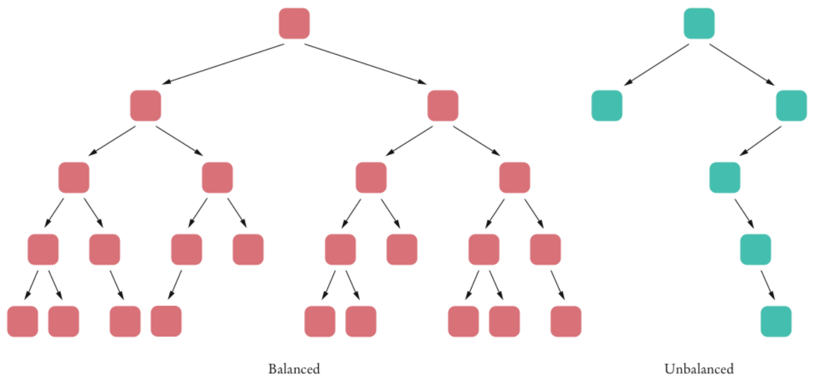


Figure 13 Balanced and Unbalanced Trees

Exercise

What is the difference between a tree, a binary tree, and a balanced binary tree?

Answer

In a tree, each node can have any number of children. In a binary tree, a node has at most two children. In a balanced binary tree, all nodes have approximately as many descendants to the left as to the right.

Binary Tree Traversal

- Print the tree elements in sorted order:
    1. *Print the left subtree*
    2. *Print the data*
    3. *Print the right subtree*

Tree traversal schemes include

- *Preorder traversal*
- *Inorder traversal*
- *Postorder traversal*

Preorder Tree traversal

- Visit the root
- Visit the left subtree
- Visit the right subtree

Inorder Tree traversal

- Visit the left subtree
- Visit the root
- Visit the right subtree

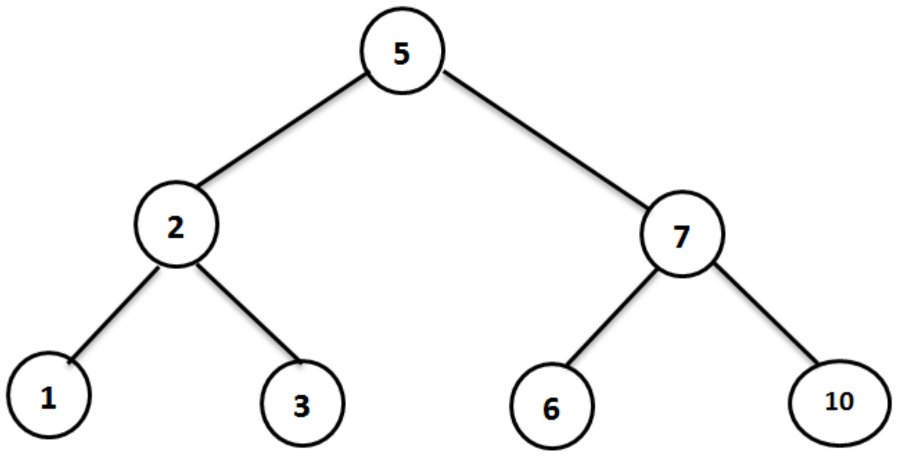
Postorder Tree traversal

- Visit the left subtree
- Visit the right subtree
- Visit the root

Example

{5,2,3,7,1,10,6}

BST



PreOrder ➔root,left subtree , right subtree

[5, 2, 1, 3, 7, 6, 10]

InOrder ➔Left subtree, root , right subtree

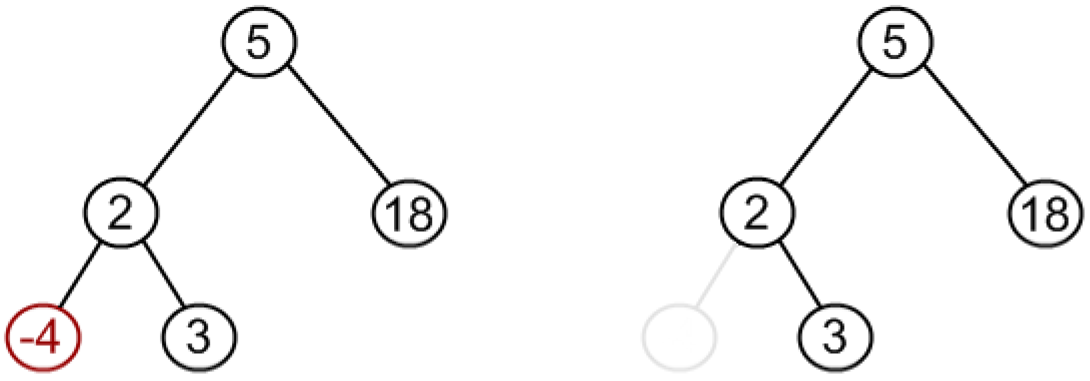
[1 , 2 , 3 , 5 , 6 , 7 , 10]

PostOrder➔Left subtree, right subtree, root

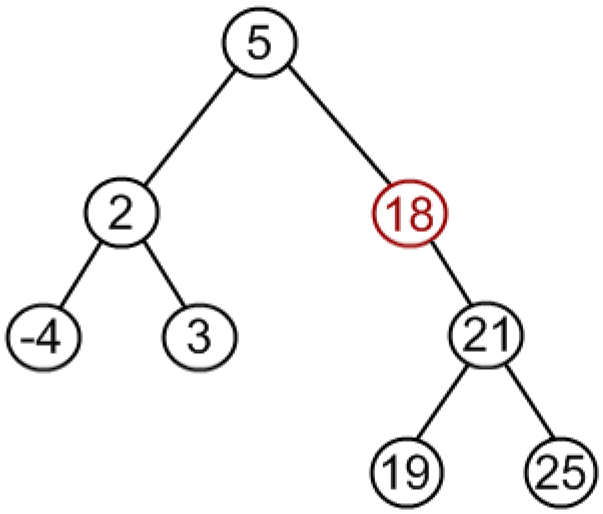
[1 , 3, 2 , 6 , 10, 7 , 5]

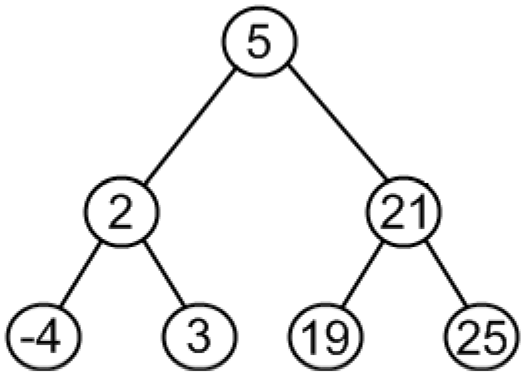
<u>Remove items from BST</u>

Example. Remove -4 from a BST.

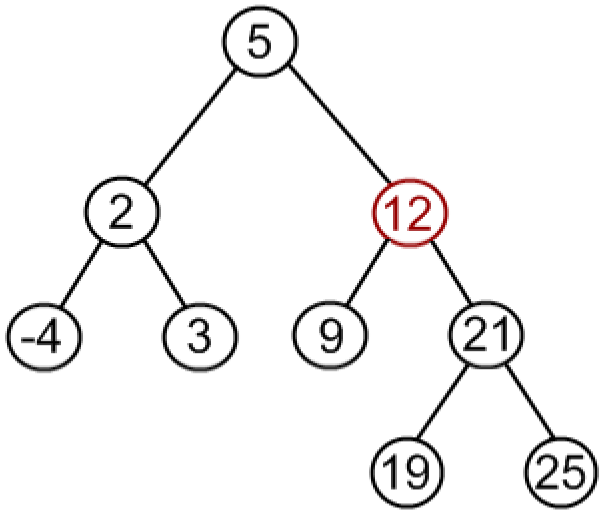


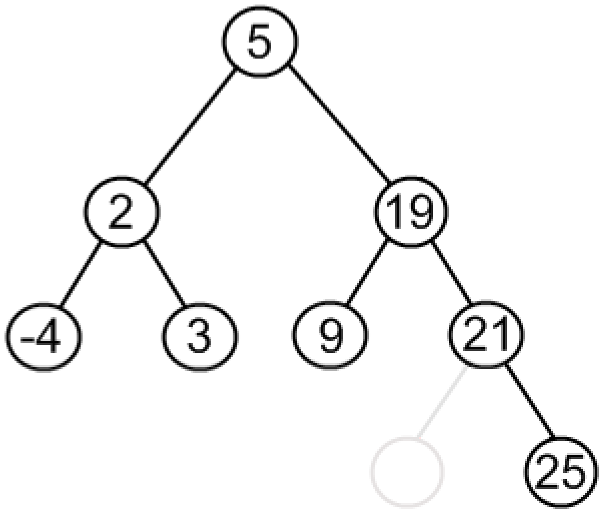
Example. Remove 18 from a BST.





Example. Remove 12 from a BST.





A priority queue

- collects elements, each of which has a priority
- Example: Collection of work requests, some of which may be more urgent than others
- When removing an element, element with highest priority is retrieved
    - *Customary to give low values to high priorities, with priority 1 denoting the highest priority*
- Standard Java library supplies a PriorityQueue class
- A data structure called *heap* is very suitable for implementing priority queues
- When calling q.remove() for the first time, the work order with priority 1 is removed
- Next call to q.remove() removes the order with priority 2

Heaps

- A **heap** (or, a *min-heap*) is a binary tree with two special properties
    1. *It is almost complete*
        - All nodes are filled in, except the last level may have some nodes missing toward the right
    2. *The tree fulfills the heap property*
        - All nodes store values that are at most as large as the values stored in their descendants
- Heap property ensures that the smallest element is stored in the root

Differences of a Heap with a Binary Search Tree

- The shape of a heap is very regular

  - *Binary search trees can have arbitrary shapes*
- In a heap, the left and right subtrees both store elements that are larger than the root element
  - *In a binary search tree, smaller elements are stored in the left subtree and larger elements are stored in the right subtree*

Heap Efficiency

- Insertion and removal operations visit at most *h* nodes
- *h*: Height of the tree
- insertion and removal operations take *O*(log(*n*)) steps
- Heap's regular layout makes it possible to store heap nodes efficiently in an array

Exercise

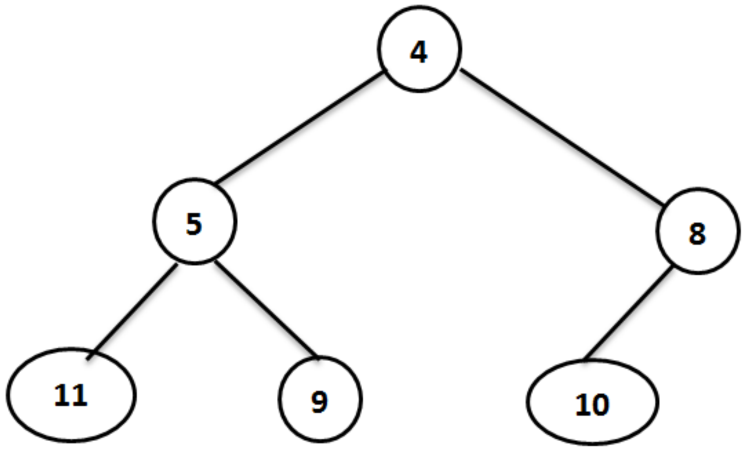
The software that controls the events in a user interface keeps the events in a data structure. Whenever an event such as a mouse move or repaint request occurs, the event is added. Events are retrieved according to their importance. What abstract data type is appropriate for this application?

Answer

A priority queue is appropriate because we want to get the important events first, even if they have been inserted later.

Draw heap by this array?

[null,4,5,8,11,9,10]



Inserting

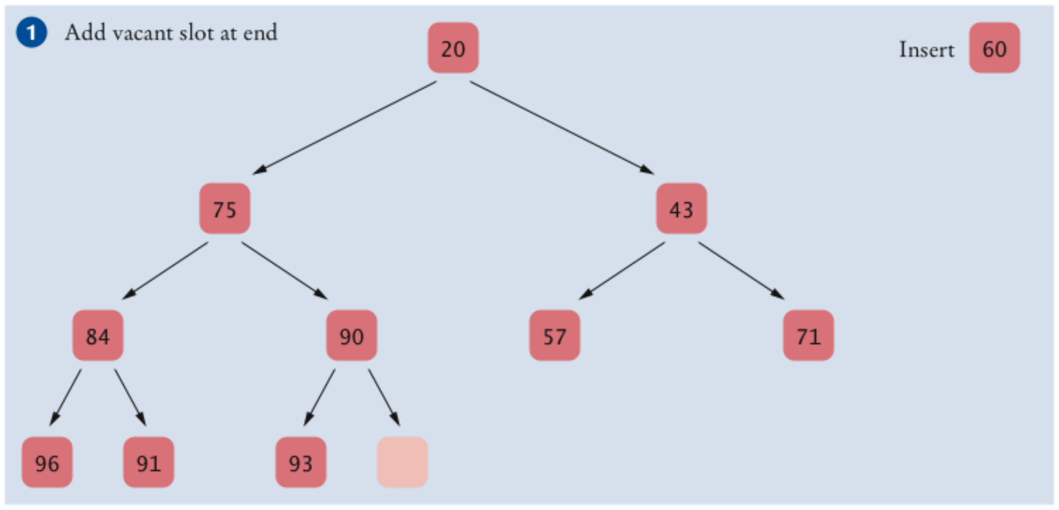
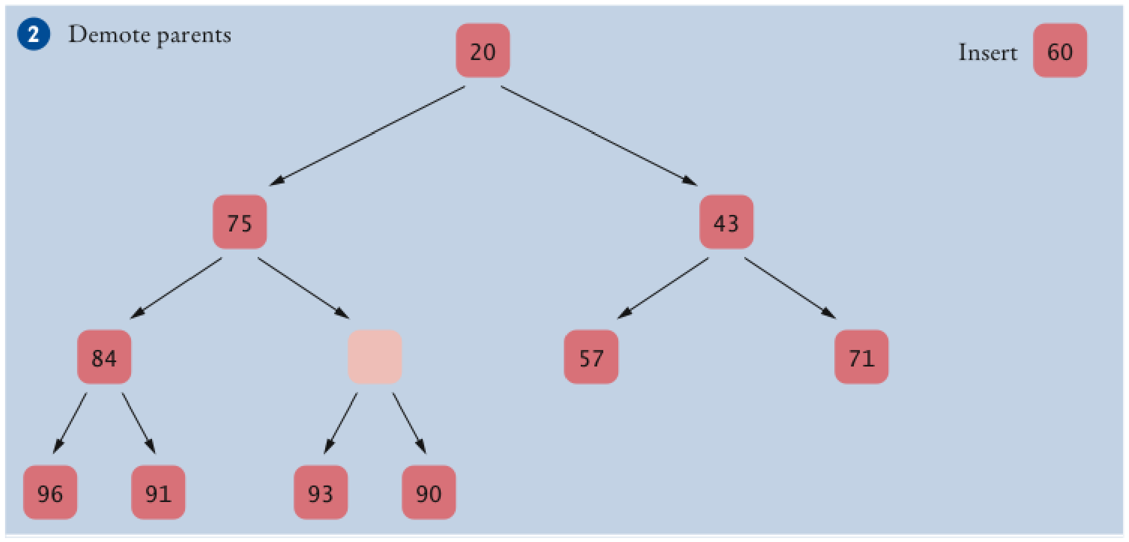
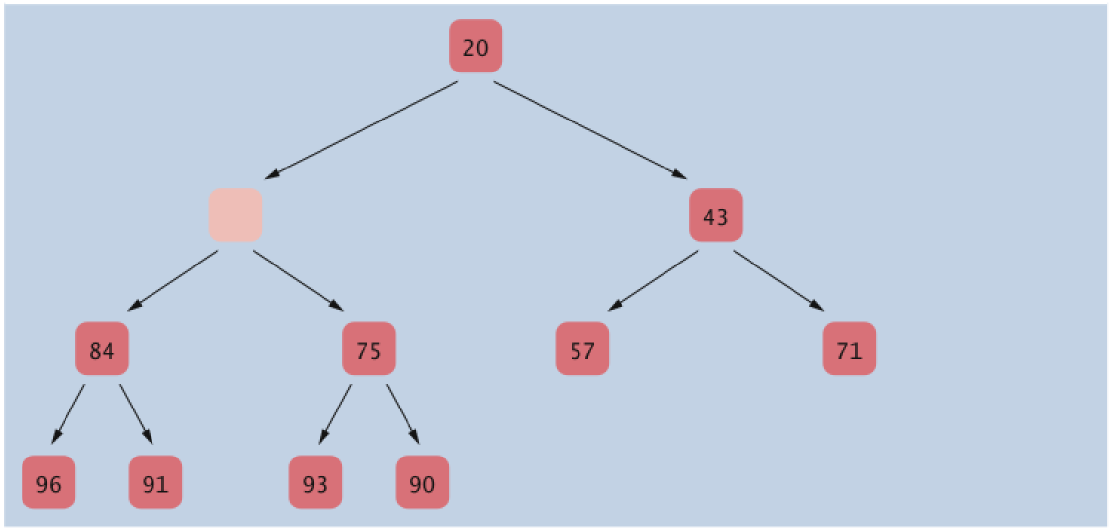


Figure 17 Inserting an Element into a Heap





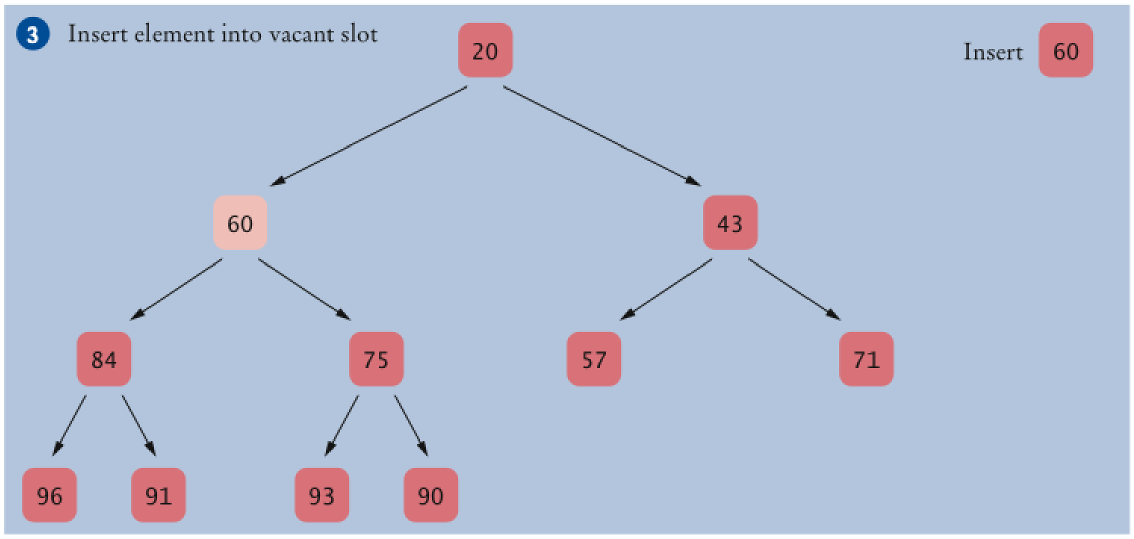
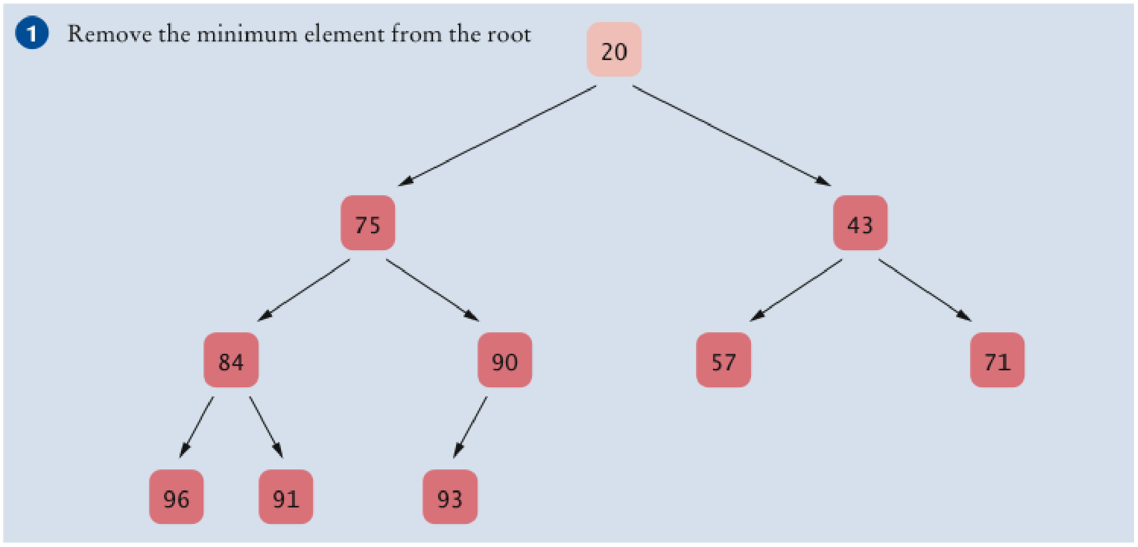
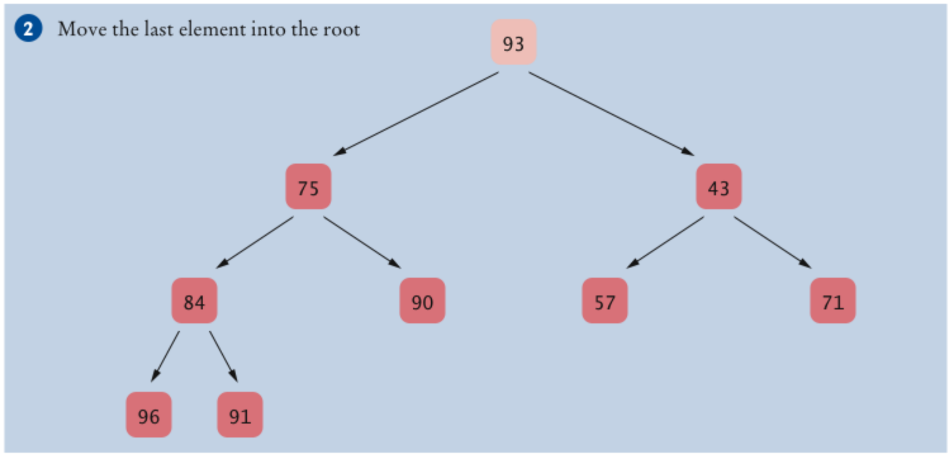
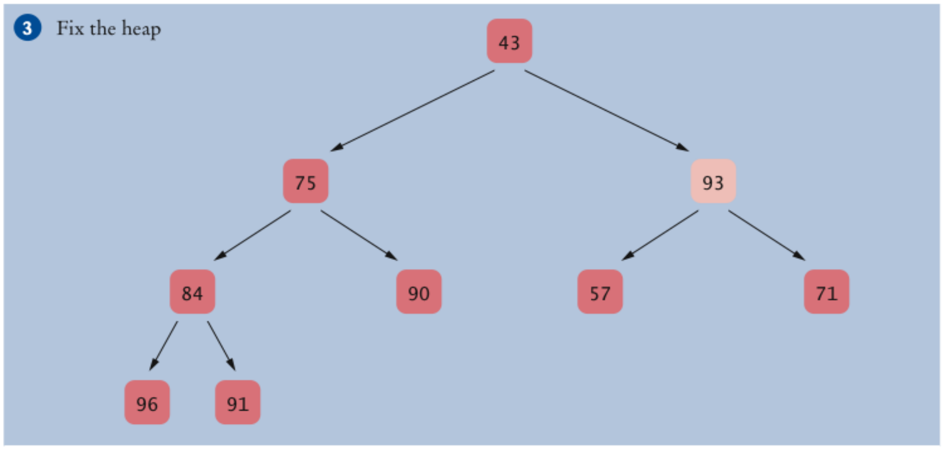


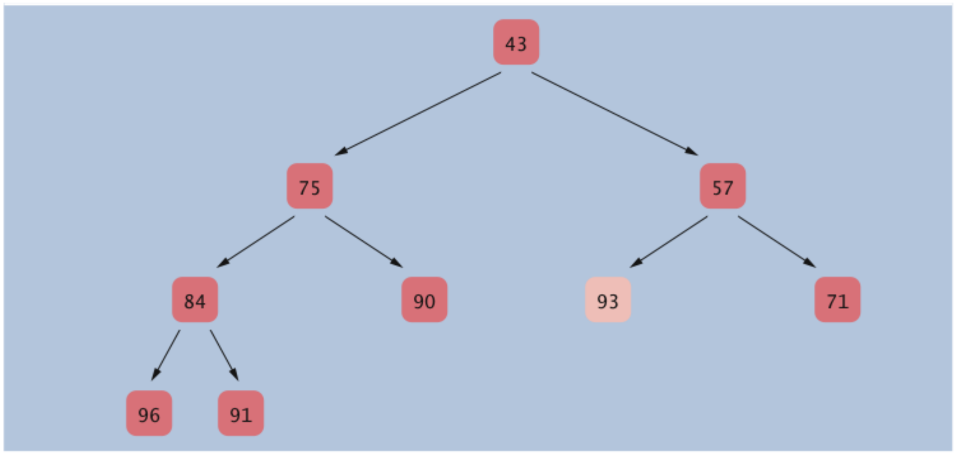
Figure 17 (continued) Inserting an Element into a Heap

remove process


1
Remove the minimum element from the root
20
75
43
84
90
57
71
96
91
93


2
Move the last element into the root
93
75
43
84
90
57
71
96
91


3
Fix the heap
43
75
93
84
90
57
71
96
91



Heapsort algorithm

- Based on inserting elements into a heap and removing them in sorted order
- This algorithm is an $O(n \log(n))$ algorithm:
    - *Each insertion and removal is O(log(n))*
    - *These steps are repeated n times, once for each element in the sequence that is to be sorted*

Exercise

Which algorithm requires less storage, heapsort or mergesort?

Answer

Heapsort requires less storage because it doesn't need an auxiliary array.

Heapsort:

store a binary search tree and Heap in an array:

children(L):2*index+1

children(R):2*index+2

Parent:1/2 -1