Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 3 Study Guide and Deliverables

Topics:	• Lists • Trees
Readings:	• Module 3 online content • Textbook: Chapter 7, Chapter 8
Assignments:	• Assignment 3 due **Tuesday, April 5 at 6:00 AM ET**
Assessments:	• Quiz 3 due **Tuesday, April 5 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, March 29 from 8:00-9:30 PM** • **Thursday, March 31 from 8:00-10:00 PM** • Another one-hour live office hour session led by your facilitator: TBD

Module 3 Learning Objectives

In this section, we discuss lists and trees, which are two fundamental data structures.

After successfully completing this module, you will be able to:

1. Implement dynamic arrays.
2. Implement positional lists.
3. Use an iterator to process elements in a collection.
4. Implement insertion-sort using a positional list.
5. Implement a binary tree using an array.
6. Implement a binary tree using a linked structure.
7. Implement a general tree using a linked structure.
8. Write programs implementing tree traversal algorithms.

Section 3.1 Lists

Section 3.1. Lists

Overview

In Module 2, we discussed data structures which represent a linearly ordered sequence of objects, such as stacks, queues, and deques. In this section, we discuss more general list structure ADT's that support adding and removing elements at arbitrary positions.

Section 3.1.1. List ADT

In this section, we define an ADT, which specifies a general list data structure. In the specified list, the location of an element is determined by an *index*. The index of an element *e* is the number of elements before *e* in the list. So, the index of the first element is 0, and that of the last element is *n* – 1, assuming that there are *n* elements in the list.

The ADT is presented as a set of following operations:

- *size*( )—Returns the number of elements currently in the list.
- *isEmpty*( )—Returns *true* if the list is empty. Returns *false* otherwise.
- *get*(*i*)—Returns the element of which the index is *i*. An error occurs if *i* is not in the range [0 … *size*( ) – 1].
- *set*(*i*, *e*)—The element at index *i* is replaced with a new element *e*, and the old, replaced element is returned. An error occurs if *i* is not in the range [0 … *size*( ) – 1].
- *add*(*i*, *e*)—Inserts a new element *e* at the location with index *i*. The element that is currently at index *i* and subsequent elements are moved to one index later in the list. An error occurs if *i* is not in the range [0 … *size*( )].
- *remove*(*i*)—Removes and returns the element at index *i*. The elements that are currently in [*i*+1 … *size*( ) – 1] are moved to one index earlier in the list. An error occurs if *i* is not in the range [0 … *size*( ) – 1].

The following table illustrates the ADT operations. It is assumed that the type of element stored in the list is *integer*. Initially, the list is empty. The table shows the resulting list after each operation.

Figure 3.1. Demonstration of List Operations

Operation	**Return Value**	**List Contents**
add(0, 25)	none	(25)
add(0, 32)	none	(32, 25)
add(3, 15)	"error"	(32, 25)
add(2, 15)	none	(32, 25, 15)
add(1, 12)	none	(32, 12, 25, 15)
add(1, 57)	none	(32, 57, 12, 25, 15)
get(2)	12	(32, 57, 12, 25, 15)
get(5)	"error"	(32, 57, 12, 25, 15)
get(4)	15	(32, 57, 12, 25, 15)
size( )	5	(32, 57, 12, 25, 15)
remove(5)	"error"	(32, 57, 12, 25, 15)
remove(1)	57	(32, 12, 25, 15)
size( )	4	(32, 12, 25, 15)
get(1)	12	(32, 12, 25, 15)
set(0, 10)	32	(10, 12, 25, 15)
size( )	4	(10, 12, 25, 15)
set(4, 29)	"error"	(10, 12, 25, 15)

Section 3.1.2. Array Lists

One way of implementing the list ADT is by using an array. An array data structure has the advantage of allowing a direct access to elements using indexes. Adding or removing elements, however, may require restructuring of the array. When an element is added, some other elements may have to be shifted to make a room for it. When an element is removed, some other elements may have to be shifted to fill the empty space it vacates. Another issue is that the capacity of an array is fixed when it is created. If the array is already full, a new element cannot be added to the list unless the capacity of the array is increased. An implementation must address these issues.

In this section, we first discuss an implementation via an array with bounded capacity. Then we discuss a more flexible implementation in which the capacity of an array grows dynamically as more elements are added to the list.

Section 3.1.2.1. Implementation with Bounded Array

Below is a part of the *ArrayList* class definition in Java, which shows instance variables and constructors. After that, we will show Java implementation of the ADT methods.

Code Segment 3.1

```
public class ArrayList<E> implements List<E> {
  // instance variables
  public static final int CAPACITY=16; // default array capacity
  private E[] data;                    // generic array used for storage
  private int size = 0;                // current number of elements
  // constructors
  public ArrayList() {this(CAPACITY);} // constructs list with default capacity
  public ArrayList(int capacity) {     // constructs list with given capacity
    data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
  }
```

The instance variable *data* is an array that stores elements of a list, and *size* is the number of elements currently in a list.

The *size*( ) and *isEmpty*( ) methods are easy to implement using the value of the instance variable *size*. The methods *get*(*i*) and *set*(*i, e*) are also easy to implement because we can access the specified location directly with the index *i*.

The following are Java codes of the *size*( ), *isEmpty( )*, *get*(*i*), and *set*(*i, e*) methods. Note that the *get* and *set* methods use a helper method, *checkIndex,* to check whether the given index *i* is valid.

Code Segment 3.2

```
public int size() { return size; }

public boolean isEmpty() { return size == 0; }

public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}

public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

Implementation of *add*(*i, e*) and *remove*(*i*), however, requires additional work.

Assume that we have a list of integers, which is implemented using a bounded array. Suppose the capacity of the array is 10, the number of elements currently in the list is 6, and we are applying the *add*(3, 10) operation. The following figure shows the array before and after the application of the operation. First, elements in *data*[3 … 5] are shifted to the right. Then, the new element 10 is added to the specified location at *data*[3].

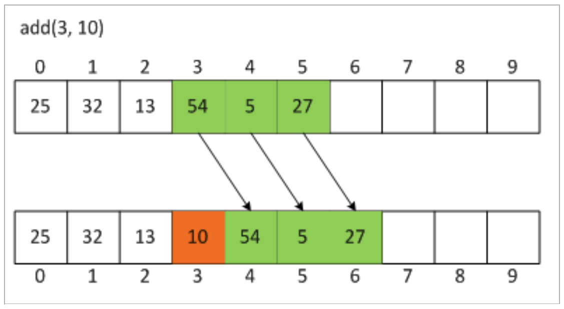


Figure 3.2. Adding an element at index 3.

The following is a Java code for the *add* operation:

Code Segment 3.3

```
1   public void add(int i, E e) throws IndexOutOfBoundsException {
2     checkIndex(i, size + 1);
3     if (size == data.length)        // not enough capacity
4       throw new IllegalStateException("Array is full");
5     for (int k=size-1; k >= i; k--) // start by shifting rightmost
6       data[k+1] = data[k];
7     data[i] = e;                 // ready to place the new element
8     size++;
9   }
```

If the array is already full, an exception is thrown in lines 3 and 4. Otherwise, elements in *data*[*i* … *size*-1] are shifted to the right to make a room for the new element in lines 5 and 6. Then the new element is added to *data*[*i*] and, in line 8, the size is incremented.

The following figure illustrates the *remove*(4) operation. The elements *data*[5] and *data*[6] are shifted to the left. This effectively removes the element in *data*[3]. If the element being removed needs to be returned, then it must be stored in a temporary variable before the shifting.

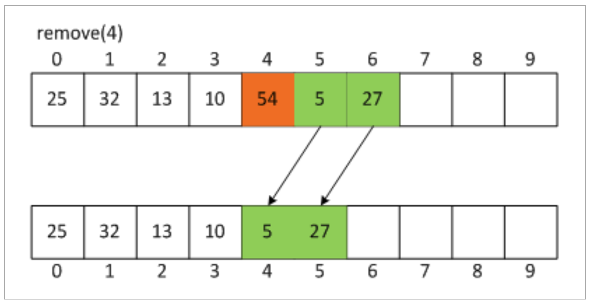


Figure 3.3. Removing an element from index 4.

A Java code implementing the *remove* operation is given below:

Code Segment 3.4

```
1   public E remove(int i) throws IndexOutOfBoundsException {
2     checkIndex(i, size);
3     E temp = data[i];
4     for (int k=i; k < size-1; k++) // shift elements to fill hole
5       data[k] = data[k+1];
6     data[size-1] = null;          // help garbage collection
7     size--;
8     return temp;
9   }
```

In line 3, *data*[*i*], the item to be removed, is stored in a temporary variable *temp*. Lines 4 and 5 shift elements in *data*[*i*+1 … *size*-1] to the left, effectively removing the element in *data*[*i*]. The size is decremented in line 7, and the removed element is returned in line 8.

Running Time Analysis

The *size*( ) and *isEmpty*( ) methods each take $O(1)$ time. The *get* and *set* methods also each take $O(1)$ time each because the specified element is accessed directly with the given index.

The *add* method and the *remove* method involve the shifting of elements. Let *n* be the number of elements in the list (i.e., *n* = *size*( )). The running time of the *add* method depends on where a new element is added. In the best case, a new element is added at index *n* – 1; only the last element needs to be shifted and the running time is $O(1)$. In the worst case, a new element is added at index 0; all *n* elements must be shifted, and the running time is $O(n)$. On average, *n*/2 elements need to be shifted. So, the average running time of the *add* method is $O(n)$. We can do a similar analysis for the *remove* method. In the best case, the last element is removed, and no element needs to be shifted. In the worst case, the element at index 0 is removed, and *n* – 1 elements must be shifted. So, the average running time of the *remove* methods is also $O(n)$.

The average running times of the methods are summarized in the following table.

Figure 3.4. Running Times of *ArrayList* Operations

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
get(*i*)	$O(1)$
set(*i*, *e*)	$O(1)$
add(*i*, *e*)	$O(n)$
remove(*i*)	$O(n)$

Section 3.1.2.2. Implementation with Dynamic Array

In the previous section, we saw how we can implement the list ADT using an array with a bounded capacity. An obvious disadvantage of this implementation is that we cannot add more elements than the capacity of the array allows. This is caused by how memory space is allocated for arrays in programming-language systems.

When an array is created and requires memory space to store its elements, the system allocates consecutive memory locations for it. In other words, all elements of an array are stored in consecutive memory locations. Following this allocation, the array cannot use memory locations before or after the sequence allocated. This is why the size of an array is fixed once it is created with a certain size.

In this section, we describe how the list ADT can be implemented using a dynamic array, the size of which can grow as elements are added. Since the underlying data structure used is still an array, we have the same problem mentioned above: The size of an array is fixed once it is created and memory space is allocated.

To solve this problem, a *dynamic array* is used.

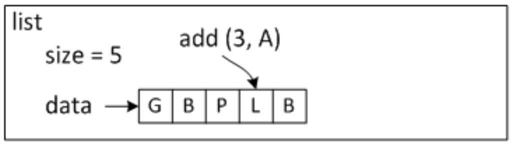
A dynamic array is implemented as follows. When an array of a list becomes full and additional space is required, a new array with a larger capacity is created. Then all elements in the existing array are copied to the new array, which is used for the list.

Suppose that the array *data* is used for a list and is full, but a new element needs to be added to the list. This situation is called an *overflow*. In this case, we first increase the array capacity and then add the new element, as described below:

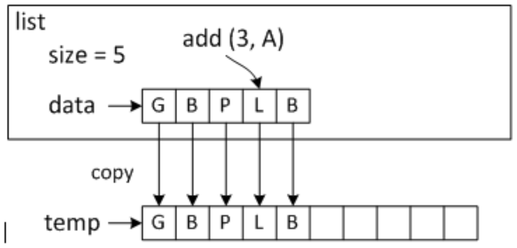
1. Allocate a new array *temp* with larger capacity
2. *temp*[*k*] = *data*[*k*], for *k* = 0, …, *n*-1, where *n* is the number of elements currently in the array
3. *data* = *temp*; now the new array becomes the array storing the elements of the list
4. Add new element to *data* (which is the reference to the new array with larger capacity)

These steps are illustrated in the following figure. We assume the list stores characters.

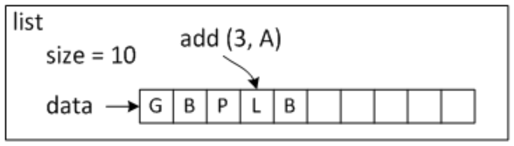
The size of *data* is 5. A new element '*A*' is to be added at *data*[3]. But, since the array is full, *overflow* occurs.



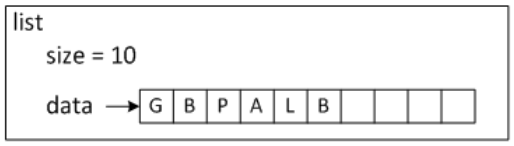
A new array *temp* with capacity 10 is created, and elements in *data* are copied to *temp*.



Make *temp* the array of the list (with the assignment *data* = *temp*).



New element '*A*' is added to *data*[3].



A Java method *resize*, which increases the size of the *data* array of a list, as illustrated above, is given below:

Code Segment 3.5

```
1   protected void resize(int capacity) {
2     E[] temp = (E[]) new Object[capacity]; // safe cast; compiler may give warning
3     for (int k=0; k < size; k++)
4       temp[k] = data[k];
5     data = temp;                           // start using the new array
6   }
```

The caller of this method must pass the new, increased capacity of the new array. In line 2, a new array with the new capacity is created. Lines 3 and 4 copy all elements from *data* to *temp*. Line 5 makes the *temp* array the array of the list.

Usually, the capacity of a new array is twice that of the existing array.

The *add* method is revised to handle overflow cases using the *resize* method, as follows:

Code Segment 3.6

```
1   public void add(int i, E e) throws IndexOutOfBoundsException {
2     checkIndex(i, size + 1);
3     if (size == data.length)        // not enough capacity, overflow
4       resize(2 * data.length);      // increase the capacity
5     for (int k=size-1; k >= i; k--) // start by shifting rightmost
6       data[k+1] = data[k];
7     data[i] = e;                    // ready to place the new element
8     size++;
9   }
```

The code is the same as the one used in the implementation with a bounded array, except that, in line 4, the capacity of the array is doubled when there is an overflow.

The implementation of the list ADT with a dynamic array is the same as the implementation with a bounded array, except that (1) the *resize* method is added and (2) the *add* method is revised.

A complete Java code of this implementation can be found at the *ArrayList.java file*.

Section 3.1.3. Positional Lists

In an index-based sequence, such as an array or a linked list, an index corresponds to the number of elements that come before the element in the sequence. So, if an element is added or removed, the index of an element changes. If the sequence is a linked list, there is another disadvantage. Given the index of an element in a linked list, we cannot directly access the element. Instead, we have to traverse the list from the head.

To alleviate the above-mentioned shortcomings, the notion of *position* is introduced. A *position* is an abstraction that represents the location of an element in a list. The index of an element represents the location of the element in the whole sequence, or in the global context. However, a position is independent of such a notion. Instead, a position focuses on the local aspect of the element's location. With a position, we can perform local operations such as *add before* and *add after*. A position allows a user to refer to any element in a list, regardless of its location, and to perform arbitrary insertions and deletions.

An example of a position is a *cursor* in a text document. A document is considered a sequence of characters, and a cursor refers to a certain character in an arbitrary location in that sequence. We can perform an insertion or deletion operation with regard to the cursor, inserting an element after the cursor or deleting the character referred to by the cursor.

We now develop a new list structure called *positional list*, which uses the *position* abstraction. We first present a position ADT. Then we define a positional list ADT, which uses the position ADT. Finally, we describe how to implement the ADT with a doubly linked list.

Section 3.1.3.1. Position

In the definition of the positional ADT, we use the *position* abstraction to refer to an arbitrary location in the list. A position is defined as an ADT on its own, effectively defining a *position type*. A position ADT has the following single method:

getElement( ): Returns the element stored at this position.

Once the position type is defined, we can use a position variable (of position type) to refer to an arbitrary location in a list and we can pass position variables as parameters of the positional list's methods.

The position *p* of an element *e* does not change when the index of *e* changes due to insertions and deletions. The position of *e* does not change either if the element *e* is replaced with a different element. Then position *p* becomes invalid only if the position and its element are explicitly removed from the list.

3.1.3.2. Positional List ADT

A positional list is viewed as a collection of positions each of which stores an element. The following access operations are defined for the positional list ADT *L*:

- first( )—Returns the position of the first element of *L* (or null if empty)
- last( )—Returns the position of the last element of *L* (or null if empty)
- before(*p*)—Returns the position of *L* immediately before position *p* (or null if *p* is the first position)
- after(*p*)—Returns the position of *L* immediately after position *p* (or null if *p* is the last position)
- isEmpty( )—Returns true if *L* does not have any element.
- Size( )—Returns the number of elements in *L*.

When a position is passed as a parameter, and if it is not a valid position for the list, an error occurs.

To retrieve an element of a position, we can invoke the *getElement* method. For example, *first*().*getElement*() retrieves the first element of the given list.

We can use a position to traverse a positional list. In the following example, *guests* is a positional list that stores names of guests, each of which is a string. The position *cursor* is used to traverse the list and print the names of the guests.

Code Segment 3.7

```
Position<String> cursor = guests.first();
while (cursor != null) {
  System.out.println(cursor.getElement());
  cursor = guests.after(cursor);
}
```

In the above code, as specified in the method definition of the ADT, the *after* method returns *null* when it is called on the last position. So, the *while* loop terminates after it prints the last guest in the list.

The positional list ADT also provides the following update operations:

- addFirst(*e*)—A new element *e* is added at the front of the list, and the position of the new element is returned.
- addLast(*e*)—A new element *e* is added at the back of the list, and the position of the new element is returned.
- addBefore(*p*, *e*)—A new element *e* is added immediately before the position *p*, and the position of the new element is returned.
- addAfter(*p*, *e*)—A new element *e* is added immediately after the position *p*, and the position of the new element is returned.
- set(*p*, *e*)—The element at position *p* is replaced with the new element *e*, and the element that was in that position before the replacement is returned.
- remove(p) —The element at position *p* is removed, and the removed element is returned. The position *p* is invalidated.

Note that we cannot use *addBefore*(*first*( ), *e*) to add a new element to an empty list because *first*( ) returns *null* when a list is empty. That is why we need separate *addFirst* and *addLast* methods.

The following table illustrates various operations applied to a positional list storing integers. Initially, the list is empty. The *List Contents* column shows integer elements stored in the list, as well as the *position* associated with each element, using variables *p*, *q*, *r*, *s*, and so on.

Figure 3.5. Demonstration of Positional-list ADT Operations

Operation	Return Value	List Contents
addFirst(10)	*p*	(10 *p*)
addLast(100)	*q*	(10 *p*, 100 *q*)
first( )	*p*	(10 *p*, 100 *q*)
p.getElement( )	10	(10 *p*, 100 *q*)
addAfter(*p*, 20)	*r*	(10 *p*, 20 *r*, 100 *q*)
addBefore(*p*, 1)	*s*	(1 *s*, 10 *p*, 20 *r*, 100 *q*)
addAfter(*q*, 200)	*t*	(1 *s*, 10 *p*, 20 *r*, 100 *q*, 200 *t*)
after(*s*)	*p*	(1 *s*, 10 *p*, 20 *r*, 100 *q*, 200 *t*)
after(*t*)	null	(1 *s*, 10 *p*, 20 *r*, 100 *q*, 200 *t*)

before(*q*)	*r*	(1 *s*, 10 *p*, 20 *r*, 100 *q*, 200 *t*)
before(*s*)	null	(1 *s*, 10 *p*, 20 *r*, 100 *q*, 200 *t*)
set(*r*, 50)	20	(1 *s*, 10 *p*, 50 *r*, 100 *q*, 200 *t*)
size( )	5	(1 *s*, 10 *p*, 50 *r*, 100 *q*, 200 *t*)
remove(*q*)	100	(1 *s*, 10 *p*, 50 *r*, 200 *t*)
remove(first( ))	1	(10 *p*, 50 *r*, 200 *t*)
remove(*s*)	"error"	

The code of a Java interface that defines the position ADT is shown below. As mentioned earlier, it includes only one method.

Code Segment 3.8

```
public interface Position<E> {
  /*
   * Returns the element stored at this position.
   *
   * @return the stored element
   * @throws IllegalStateException if position no longer valid
   */
   E getElement() throws IllegalStateException;
}
```

The code of a Java interface which defines the positional list ADT can be found at the *PositionalList.java* file.

Section 3.1.3.3. Doubly Linked-List Implementation

In this section, we describe a concrete implementation of the positional list ADT using a doubly linked list.

The Java class *LinkedPositionalList* implements the positional list ADT. Within the *LinkedPositionalList* class definition, the *Node* class is defined as a private nested class. The *Node* class is a concrete implementation of the *Position* ADT. So, in this implementation, nodes are positions.

The *Node* class has three instance variables:

```
private E element;       // reference to the element stored at this node
private Node<E> prev;    // reference to the previous node in the list
private Node<E> next;    // reference to the subsequent node in the list
```

In addition to the *getElement* method, which is defined in the *Position* interface, it includes the following public methods:

```
public Node<E> getPrev() {
  return prev;
}

public Node<E> getNext() {
  return next;
}

public void setElement(E e) {
  element = e;
}

public void setPrev(Node<E> p) {
  prev = p;
}

public void setNext(Node<E> n) {
  next = n;
}
```

The *LinkedPositionalList* class has following instance variables:

```
  private Node<E> header;    // header sentinel
  private Node<E> trailer;   // trailer sentinel
  private int size = 0;      // number of elements in the list
```

Note that a list of *LinkedPostionalList* has two sentinel nodes, which were first discussed in Section 1.3.9 of Module 1. As discussed earlier, they are declared as private so that they may be hidden from the user.

The *LinkedPositionalList* class implements all methods specified in the *PsitionalList* interface. In addition, it implements the following three methods:

```
  private Node<E> validate(Position<E> p) throws IllegalArgumentException
  private Position<E> position(Node<E> node)
  private Position<E> addBetween(E e, Node<E> pred, Node<E> succ)
```

The *validate* method ensures that a *position* is a valid one. A Java code is shown below:

```
  private Node<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
    Node<E> node = (Node<E>) p;     // safe cast
    if (node.getNext() == null)     // convention for defunct node
      throw new IllegalArgumentException("p is no longer in the list");
    return node;
  }
```

The *position* method receives a node as its argument and returns the *position* of the node. If the node is a sentinel (i.e., either a *header* or *trailer*), it returns null. The following is a Java code:

```
  private Position<E> position(Node<E> node) {
    if (node == header || node == trailer)
      return null;   // do not expose user to the sentinels
    return node;
  }
```

The *addBetween* method adds a new element *e* between two specified nodes. The code is shown below:

```
  private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
    // create and link a new node
    Node<E> newest = new Node<>(e, pred, succ);
    pred.setNext(newest);
    succ.setPrev(newest);
    size++;
    return newest;
  }
```

A complete code of *LinkedPositionalList.java* can be found at the *LinkedPositionalList.java* file.

Since the positional list is implemented using a doubly linked list, all operations run in constant time. The following table shows the running times of the methods:

Figure 3.6. Running Times of Operations of Positional List Implemented Using a Doubly Linked List

Method	**Running Time**

size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(*p*), after(*p*)	$O(1)$
addFirst(*e*), addLast(*e*)	$O(1)$
addBefore(*p*, *e*), addAfter(*p*, *e*)	$O(1)$
set(*p*, *e*)	$O(1)$
remove(*p*)	$O(1)$

Section 3.1.4. Iterators

We just discussed a position abstraction and its implementation in Java. A related concept is an *iterator*. An iterator is a software design pattern for traversing a data structure, such as a sequence. Like a position, an iterator is an abstraction that hides the internal details of the data structure from users.

In this section we briefly describe Java's *Iterator* interface and *Iterable* interface.

Java's *Iterator* interface is defined in the *java.util* package, and an *Iterator* object can be used to iterate over a Java collection. The *Iterator* interface includes the following three methods:

- hasNext( )—Returns true if there is at least one additional element in the collection.
- next( )—Returns the next element in the collection.
- remove( )—Removes from the collection the element returned by the most recent call to next( ) (optional operation).

Through these methods, *Iterator* provides a uniform way of traversing collections, regardless of their internal organizations.

To use an iterator to traverse a collection, we first create an iterator object. The *Iterable* interface has the *iterator*( ) method, which creates and returns an *Iterator* object. The Java *Collection* interface extends the *Iterable* interface so that all collection objects can invoke the *iterator*( ) method to create an iterator.

The following example illustrates how an iterator is used to traverse and print elements in an *ArrayList*, which stores strings:

```
ArrayList<String> stringList = new ArrayList<>( );
// population of the list omitted
Iterator<String> stringIterator = stringList.iterator( );
While (stringIterator.hasNext( ))
  System.out.println(stringIterator.next( ));
```

In Java, there is a simpler syntax to iterate over a collection object, which is the *for-each* loop syntax:

```
for (ElementType variable : collection) {
       loopBody
}
```

ElementType is the type of object returned by the iterator of *collection*, and *variable* assumes the returned object in *loopBody*. This syntax is implemented, internally, with an iterator object and is shorthand for the following:

```
Iterator<ElementType> iter = collection.iterator( );
While (iter.hasNext()){
       ElementType variable = iter.next;
       loopBody
}
```

Note that when we use the *for-each* loop syntax, we cannot use the *remove* method of the iterator.

Section 3.1.5. Java Collections Framework

When we write programs, very often we need to organize and manipulate a collection of objects in a certain way. For this, we use a data structure, which is also called a *collection* or a *container*. It allows us to organize and manipulate a group of objects with the same properties effectively. Java provides many data structures (or containers), called the *Java Collections Framework*.

In Java, there are two types of containers:

- *Collection*—Stores a collection of objects.
- *Map*—Stores key/value pairs.

The Java Collections Framework includes many interfaces that specify different containers, and it also includes concrete classes that implement interfaces. The following figure shows the hierarchy of Java's core collection interfaces:

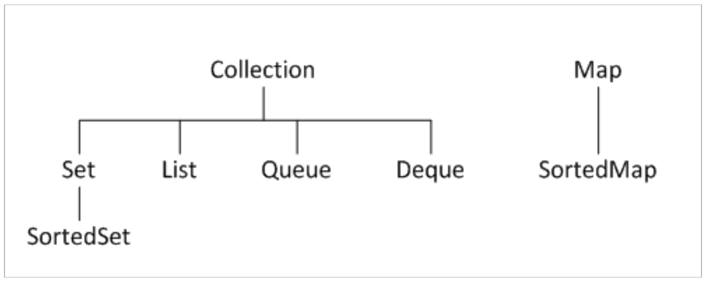


Figure 3.7. Java core collection interfaces.

The *Collection* is the root interface for manipulating collections of objects. It specifies common operations of all of its subinterfaces, which include *Set*, *List*, *Queue*, *Deque*, and their subinterfaces. We already discussed some of these interfaces. We also saw some common operations, including *size*( ), *isEmpty*( ), and *iterator*( ).

The *Map* container will be discussed in Module 4. In this section, we briefly discuss some of features of the *List* container.

3.1.5.1. List Iterators

In Section 3.1.4, we introduced Java's *Iterator* interface. The *ListIterator* interface extends the *Iterator* interface and adds bi-directional traversal of a list. So, a list iterator can move forward and backward. The position of a list iterator is assumed to be before the first element, between two consecutive elements, or after the last element. A list iterator is obtained by invoking the *listIterator*( ) method of a *List* interface. It inherits all operations of *Iterator* and defines additional local update operations. The *ListIterator* interface includes the following methods:

- add(*e*)—Inserts the element *e* at the current position of the iterator.
- hasNext( )—Returns *true* if there is an element after the current iterator position.
- hasPrevious( )—Returns *true* if there is an element before the current iterator position.
- previous( )—Returns the element *e* before the current iterator position and sets the current position to be before *e*.
- next( )—Returns the element *e* after the current iterator position and sets the current position to be after *e*.
- nextIndex( )—Returns the index of the next element.
- previoustIndex( )—Returns the index of the previous element.
- remove( )—Removes the element returned by the most recent *next* or *previous* operation.
- set(*e*)—Replaces the element returned by the most recent *next* or *previous* operation with *e*.

We can create multiple iterators on a list. If all iterators are just traversing the list, there is no problem. However, if one of the iterators modifies the list, then there is a potential problem. Suppose that you created two iterators on a list *L*, and one iterator is modifying *L*. If the other iterator is used to traverse the list *L* (while *L* is being modified), then a *ConcurrentModificationException* is thrown, and the iterator attempting to traverse *L* fails immediately. This feature is called *fail-fast*. Iterators returned by either the *iterator*( ) method (discussed in Section 3.1.4) or the *listIterator*( ) method (of this section) are fail-fast. So, we have to be careful to avoid this situation. This problem can be resolved using *synchronized* collections.

Section 3.1.5.2. Java's List-Based Algorithms

Individual collection interfaces and classes have their operations, some of which we discussed earlier. In addition to these methods, the *Java Collections Framework* has a collection of algorithms that are implemented as static methods in the *Collections* class. Note that the *Collections* class is also a member of the *Java Collections Framework* and should be distinguished from the *Collection* interface.

The *Collections* class has only static methods, which are applied to collections and return collections. Some of the methods defined in the *Collections* class are shown below. Here, *L* denotes a list, and *C* or *D* denotes a collection.

- copy(L_{dest}, L_{src})—Copies all elements of L_{src} list into corresponding indices of the L_{dest} list.
- disjoint(*C*, *D*)—Returns *true* if two collections *C* and *D* are disjointed.
- fill(*L*, *e*)—Replaces each element in *L* with *e*.
- frequency(*C*, *e*)—Returns the number of elements in collection *C* that are equal to *e*.
- replaceAll(*L*, *e*, *f*)—Replaces each occurrence of *e* in *L* with *f*.
- reverse(*L*)—Reverses the ordering of elements in *L*.
- shuffle(*L*)—Pseudorandomly permutes the ordering of elements in *L*.
- sort(*L*)—Sorts the elements in *L*, using their natural ordering.
- swap(*L*, *i*, *j*)—Swaps the elements at indices *i* and *j* in *L*.

Lists and arrays have their own advantages and disadvantages. So, it would be desirable to be able to convert one to the other under certain circumstances.

Java's *Collection* interface has the following two methods, which we can use to convert a collection to an array:

- toArray( )—Returns an array of elements of type *Object*, containing all elements in the collection.
- toArray(*T*[ ] *a*)—Returns an array of elements of type *T*, containing all elements in the collection. The parameter *a* is the array of type *T* into which the elements of the collection are to be stored.

When the collection being converted is a list, then the order of elements in the returned array will be the same as the order by which they appear in the list.

If we want to convert an array to a list, we can use a static method *asList* which is defined in the *java.util.Arrays* class.

- asList(T … a)—Receives an array (or a variable-length list of arguments in general) and returns a list with the same elements.

The following example shows how to convert lists to arrays:

```
Integer[ ] intArray = {1,2,3,4,5};
List<Integer> integerList = Arrays.asList(intArray);
List<String> stringList = Arrays.asList("Boston", "New York", "Chicago");
```

The *integerList* is created from the integer array *intArray*, and the *stringList* is created from an argument consisting of three strings.

Section 3.1.6. Sorting a Positional List

In Section 1.3.2, we described how to sort an array using the *insertion-sort* algorithm. We can use the same algorithm to sort elements in a positional list.

We use three variables: *marker*, *pivot*, and *walk*. At any moment during the sorting process, the given list consists of two parts: one that is already sorted, and one that has elements not yet explored. The *marker* represents the position of the rightmost element in the part of the list that is already sorted. The *pivot* is the position of the element to the immediate right of the list (we assume that the *header* of the list is at the leftmost end and the *trailer* of the list is at the rightmost end). So, the *pivot* represents the first element in the unsorted part of the list. The *walk* is used to traverse the already sorted part of the array.

The algorithm moves the *pivot* element into the sorted part in such a way that all elements in that part are sorted. The *walker* is used in finding the correct position of the *pivot* element in the sorted part. The process is illustrated in the following figure:

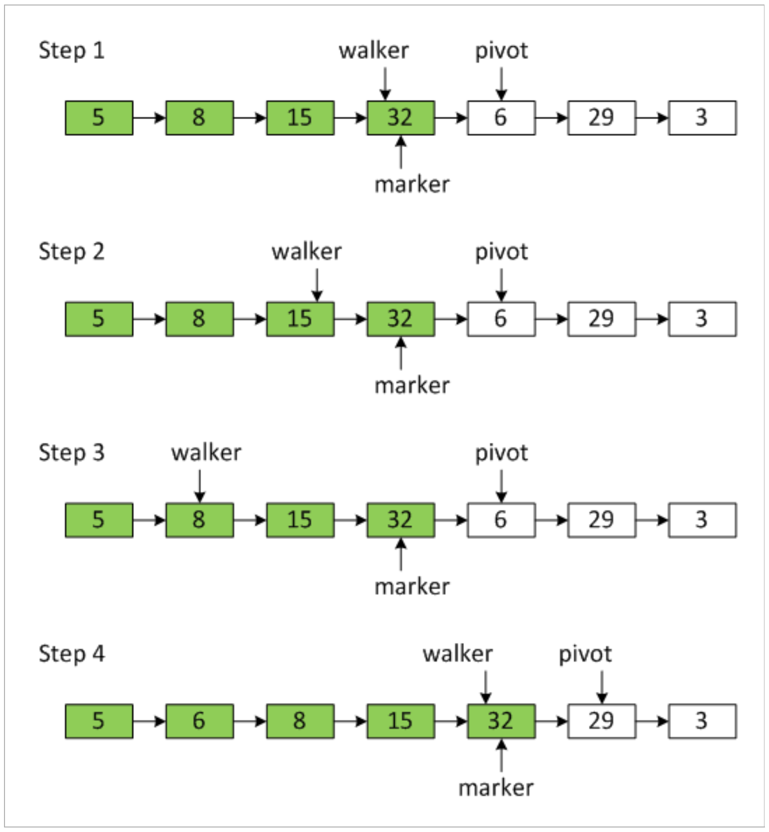


Figure 3.8. Sorting a positional list using insertion-sort.

In this example, we use a singly linked list of integers as underlying storage of a positional list. In Step 1, the sorted part has elements 5, 8, 15, and 32; *marker* is the position of 32, *pivot* is the position of 6, and *walker* references the *marker* element. In Steps 2 and 3, *walker* moves to the left until the element immediately before the *walker* element is smaller than or equal to the *pivot* element. In this example, *walker* stops at the element 8. Then the *pivot* element is removed from its current location and added to the location immediately before *walk*. In Step 4, the same process is repeated to move the new *pivot* element, 29, into the correct position in the sorted part.

A Java code implementing the insertion-sort on a positional list is shown below:

Code Segment 3.9

```
public static void insertionSort(PositionalList<Integer> list) {
  Position<Integer> marker = list.first(); // last position known to be sorted
  while (marker != list.last()) {
    Position<Integer> pivot = list.after(marker);
    int value = pivot.getElement();      // number to be placed
    if (value > marker.getElement())     // pivot is already sorted
      marker = pivot;
    else {                               // must relocate pivot
      Position<Integer> walk = marker;   // find leftmost item greater than value
      while (walk != list.first() && list.before(walk).getElement() > value)
        walk = list.before(walk);
      list.remove(pivot);                // remove pivot entry and
      list.addBefore(walk, value);       // reinsert value in front of walk
    }
  }
}
```

Section 3.1.7. Case Study: Maintaining Access Frequencies

In this case study, we implement a *favorite list*. Suppose that there is a collection of items that are accessed by many users. Your goal is to maintain separate lists of items that are accessed by each user. In each list, you want to keep the items in the order of their access frequency, with the most frequently accessed item at the front of the list. So, a favorite list is an ordered list that stores the favorite items of each user.

This *favorite list* ADT supports the following operations in addition to the *size*( ) and *isEmpty*( ) methods:

- access(*e*)—Accesses the element *e*, adding it to the favorite list if it is not already in the list, and increments its access count.
- remove(*e*)—Removes the element *e* from the favorite list, if present.
- getFavorite(*k*)—Returns the iterable collection of the *k* most accessed elements.

The favorite list ADT can be implemented using a linked list, in which elements are stored in nondecreasing order of their access counts. We can easily return the *k* most accessed elements by taking the first *k* elements from the list. When there is an insertion or a deletion, we can search the item from the head of the list.

When an existing element is accessed, its access count is incremented, and this may require the relocation of this element to the correct position toward the front of the list. This can be achieved using the same technique as in the insertion-sort, discussed in the previous section. We can find the correct position of the element of which the count was incremented in the same way as we find the correct position of the *pivot* item in the sorted part of the list.

We now discuss how to implement the favorite list ADT in Java. We use the positional list, discussed in Section 3.1.3, as underlying storage of a favorite list. The class *FavoriteList* is a generic class implementing the favorite list ADT. In this implementation, an element in a positional list is a *composite pattern* that stores an item and its count. Each element of a positional list is implemented as a protected nested class *Item*. The following Java code segment shows the *Item* class definition:

Code Segment 3.10

```
protected static class Item<E> {
  private E value;
  private int count = 0;
  public Item(E val) { value = val; }
  public int getCount() { return count; }
  public E getValue() { return value; }
  public void increment() { count++; }
}
```

The *Item* class has two instance variables: *value* stores an item, and *count* keeps the number of times the item has been accessed. In addition to a constructor, the class has the *getCount*( ), *getValue*( ), and *increment*( ) methods.

The *FavoriteList* class has nine methods. We now briefly discuss the following three methods: *moveUp*, *access*, and *getFavorite*.

The *moveUp* method is used to relocate an item of which the access count was incremented. This method also uses a position variable *walk*, as was done in the insertion-sort, to find the correct position of the item. So, its code is very similar to that of the insertion-sort. The code is shown below:

Code Segment 3.11

```
1  protected void moveUp(Position<Item<E>> p) {
2    int cnt = count(p);         // revised count of accessed item
3    Position<Item<E>> walk = p;
4    while (walk != list.first() && count(list.before(walk)) < cnt)
5      walk = list.before(walk); // found smaller count ahead of item
6    if (walk != p)
7      list.addBefore(walk, list.remove(p)); // remove/reinsert item
8  }
```

The argument passed to the method is the position *p* of the item of which the count was incremented. The *count* method in line 2 retrieves the incremented count of the element at position *p*. In line 3, *walk* is set to *p*, the position of the item. In the *while* loop of lines 4 and 5, the correct position of the item is found. If the item is already in the correct positions (i.e., the access count of the item at *p* is greater than that of the item immediately before it), then the condition in line 6 becomes false and nothing happens. Otherwise, the item at *p* is removed from the current position and added to the position immediately before *walk* in line 7.

The *access* method first checks whether the element being accessed is already in the list. If it is not in the list, a new *Item* object is created (with the initial count 0) and added to the end of the list, and its count is incremented. If it is already in the list, its count is incremented and the *moveUp* method is invoked. The code is shown below:

Code Segment 3.12

```
public void access(E e) {
  Position<Item<E>> p = findPosition(e); //try to locate existing element
  if (p == null)
    p = list.addLast(new Item<E>(e));   // if new, place at end
  p.getElement().increment();           // always increment count
  moveUp(p);                            // consider moving forward
}
```

The *getFavorite* method retrieves the *k* most accessed items from the list. The code is shown below:

Code Segment 3.13

```
1   public Iterable<E> getFavorites(int k) throws IllegalArgumentException {
2     if (k < 0 || k > size())
3       throw new IllegalArgumentException("Invalid k");
4     PositionalList<E> result = new LinkedPositionalList<>();
5     Iterator<Item<E>> iter = list.iterator();
6     for (int j=0; j < k; j++)
7       result.addLast(iter.next().getValue());
8     return result;
9   }
```

The *k* most accessed items are returned as a positional list. So, in line 4, a variable *result* of type *PositionalList* is created. Line 5 creates an iterator object *iter*, which is used to traverse and collect the first *k* items from the list, in lines 6 and 7.

A complete Java code for this implementation can be found at the *FavoritesList.java* file.

Section 3.2 Trees

Section 3.2. Trees

Overview

A *tree* is organizational structure that represents a collection of items in a hierarchical manner. Examples of trees include family trees, organizational charts, and hierarchical structures of some mathematical expressions. In this section, we discuss trees as data structures used in computer algorithms and computer programs.

Section 3.2.1. General Trees

A *tree* is a data structure that stores a collection of items in a hierarchical manner. In a tree data structure, each element—except the top element, which is referred to as the *root* of the tree—has a *parent* element and zero or more *children* elements. The following is an example of a tree showing the hierarchy of items sold in an electronic store:

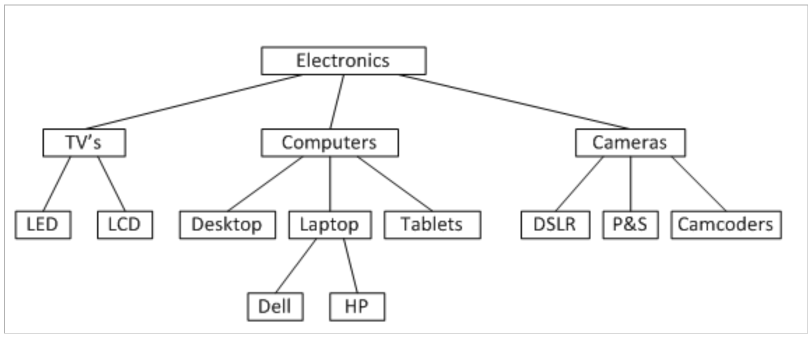


Figure 3.9. A tree representing a hierarchy of items sold in an electronic store.

Usually, the root of a tree is drawn at the top, its children below it, their children below each of them, and so on. In this example, *Electronics* is the root of the tree.

Note that the trees defined in this section are referred to as *rooted trees*. In other literature, a *tree* means a *free tree,* which does not have a root. In our lecture, when we say a *tree*, it refers to a *rooted tree*.

Section 3.2.1.1. Definitions and Properties

A *tree T* is a set of *nodes* storing elements such that the nodes have a *parent-child* relationship that satisfies the following properties:

- If *T* is nonempty, it has a distinguished node, called the *root* of *T*, that has no parent.
- Each *v* of *T*, except the root node, has a unique *parent* node *w*. Every node with parent *w* is a *child* of *w*.

These are some other properties of trees:

- Nodes with the same parent are called *siblings*.
- A node *v* is external if *v* has no children. An external node is also called a *leaf* or a *leaf node*.
- A node *v* is internal if *v* has one or more children.
- A node *u* is an *ancestor* of node *v* if *u* = *v* or *u* is an ancestor of the parent of *v*.
- A node *v* is a descendant of a node *u* if *u* is an ancestor of *v*.
- An *edge* is a pair of nodes (*u*, *v*) such that *u* is the parent of *v*, or vice versa.
- A *path* is a sequence of nodes such that any two consecutive nodes in the sequence constitute an edge. The length of a path is the number of edges in the path.
- There is a unique path from any node to the root.

The following is a part of a file system on a personal computer, represented as a tree:

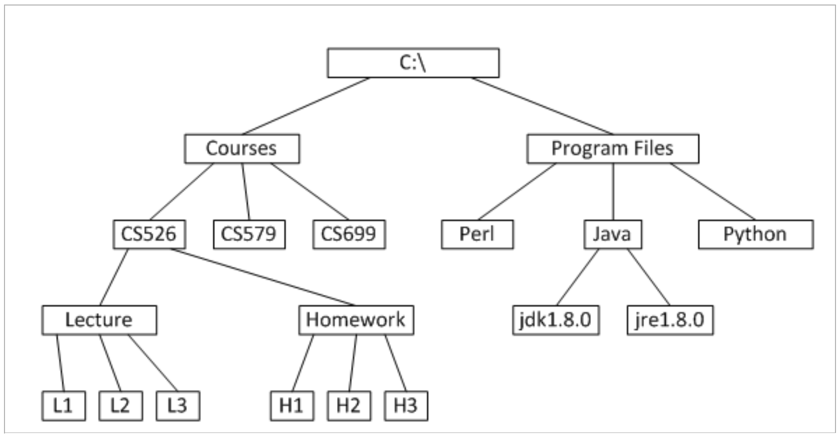


Figure 3.10. A tree representing a file system hierarchy.

The above tree has the following characteristics:

- *C:\* is the root node of the tree.
- *Courses* is a child of the root node.
- *CS526* is the parent of *Lecture* and *Homework*.
- *Perl*, *Java*, and *Python* are children of *Program Files*, and they are siblings.
- *L1*, *L2*, *L3*, *H1*, *H2*, *H3*, *CS579*, *CS699*, *Perl*, *Python*, *jdk1.8.0*, and *jre1.8.0* are leaves (or external nodes) of the tree. All other nodes are internal nodes.
- *C:\, Courses*, *CS526*, and *Lecture* are ancestors of *L1*, *L2*, and *L3*.
- *jdk1.8.0* and *jre1.8.0* are descendants of *Program Files*.

In the above tree, (*Courses*, *CS579*) and (*Java*, *jdk1.8.0*) are edges, (*Courses*, *CS526*, *Lecture*, *L1*) is a path, and the length of the path is 3. The unique path from *jre1.8.0* to the root is (*jre1.8.0*, *Java*, *Program Files*, *C:\*).

A tree is *ordered* if there is a meaningful order among the children of each node. An example of an ordered tree is one that represents the structure of a book. Components of a book form a hierarchy and are naturally ordered. For example, a book begins with a *Preface*, followed by a certain number of *Parts*, and ends with *References*. In each *Part* are *Chapters*, and there is natural order among them. Each *Chapter* has *Sections* and *Sections*, which are ordered by their section numbers. The following figure shows an example of such a tree.

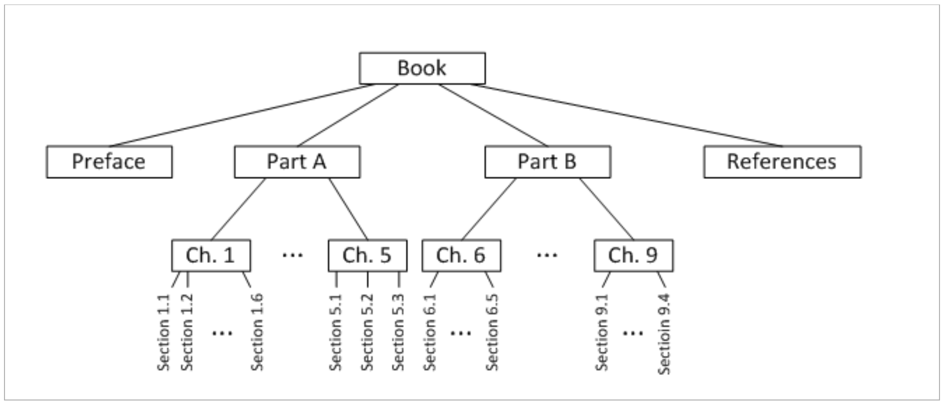


Figure 3.11. An ordered tree. There is ordering among children of a node.

Section 3.2.1.2. Tree ADT

In this section, we define a tree ADT, in which the *position* is used as an abstraction for a tree node.

A position stores each element, and positions satisfy the parent-child relationship. A position object supports the following single method:

getElement( )—Returns the element stored at this position.

The initial design of the tree ADT supports three types of methods: the *accessor*, *query*, and other *general* methods.

Accessor methods allow users to retrieve information about a position at different parts of a tree. Accessor methods include the following:

- root( )—Returns the position of the root of the tree, or *null* if the tree is empty.
- parent(*p*)—Returns the position of the parent of position *p*, or *null* if *p* is the root.
- children(*p*)—Returns the children of position *p*, if any. If the tree is an ordered tree, children are ordered in the result.
- numChildren(*p*)—Returns the number of children of position *p*.

Query methods answer *yes/no* questions about certain properties of positions. They include the following:

- isInternal(*p*)—Returns *true* if position *p* is an internal node.
- isExternal(*p*)—Returns *true* if position *p* is an external node (or a leaf node).
- isRoot(*p*)—Returns *true* if position *p* is the root of the tree.

The following general methods are also defined in the ADT:

- size( ): Returns the number of positions (or the elements) in the tree.
- isEmpty( ): Returns true if the tree does not have any position (or element).
- iterator( ): Returns an iterator for all elements in the tree. So, the tree is *iterable*.
- positions( ): Returns an *iterable* collection of all positions of the tree.

Update methods will be defined later when the tree ADT is implemented.

A Java interface that specifies the tree ADT is given below:

Code Segment 3.14

```java
public interface Tree<E> extends Iterable<E> {

  Position<E> root();
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                              throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size();
  boolean isEmpty();
  Iterator<E> iterator();
  Iterable<Position<E>> positions();
}
```

One difference between an interface and an abstract class, as discussed in Section 1.2.2, is that an abstract class may have concrete implementations of some of its methods. Usually, we define an abstract class when many related classes share a code. The shared code is defined and implemented in the abstract class, and then those related classes can inherit the shared code by extending the abstract class. Note that an abstract class itself cannot be instantiated.

For our tree design, we can define an *abstract tree base class.* This implements some methods that are independent of low-level details, which can be implemented in its subclass(es).

The following abstract tree class implements the *Tree* interface and provides implementations for four simple methods:

Code Segment 3.15

```java
public abstract class AbstractTree<E> implements Tree<E> {
  public boolean isInternal(Position<E> p) {return numChildren(p)>0;}
  public boolean isExternal(Position<E> p) {return numChildren(p)==0;}
  public boolean isRoot(Position<E> p) {return p==root();}
  public boolean isEmpty() {return size()==0;}
}
```

Other methods will be added later in a revised abstract tree class and in concrete subclasses of this abstract class.

Section 3.2.1.3. Depth and Height

In this section, we introduce two more concepts with regard to trees: *depth* and *height*.

Given a position *p*, the depth of *p* is defined to be the number of ancestors of *p*, except *p* itself. Note that this is the same as the length of the path from *p* to the root of the tree (In a tree, there is a unique path from a node to the root of the tree.) So, by definition, the depth of the root of a tree is 0.

The depth of a tree can be alternatively defined recursively, as follows:

- If *p* is the root, the depth of *p* is 0.
- Otherwise, the depth of *p* is one plus the depth of its parent.

We can implement a method that determines the depth of a position using the recursive definition. A Java code of the method is shown below:

Code Segment 3.16

```java
public int depth(Position<E> p) throws IllegalArgumentException {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

The *height* of a tree is the length of the longest path from the root downward to an external node (or a leaf node). This is the same as the maximum of the depths of all positions in the tree. If a tree is empty, then the height is 0. In the personal computer-file system tree shown in Section 3.2.1.1, the height of the tree is 4. In this tree are multiple longest paths from the root to external nodes. They are the paths from the root to individual lectures (e.g., *L1*) or individual homeworks (e.g., *H1*). The lengths of the paths are all the same: 4.

The following Java code implements the method that computes the height of a tree. This method uses the *depth* method shown above. Note that this method is named *heightBad* because it is not very efficient.

Code Segment 3.17

```
1   private int heightBad() {  // works, but quadratic worst-case time
2     int h = 0;
3     for (Position<E> p : positions())
4       if (isExternal(p))     // only consider leaf positions
5         h = Math.max(h, depth(p));
6     return h;
7   }
```

The *for* loop of lines 3 and 4 determines the maximum depth by examining the depths of all positions. However, since the maximum depth occurs only with leaf nodes, the *if* statement in line 4 considers only leaf nodes. The running time of this method is $O(n^2)$.

There is an efficient implementation of the *height* method. First, we define the *height* of a position *p* recursively, as follows:

- If *p* is a leaf, then the height of *p* is 0.
- Otherwise, the height of *p* is one more than the maximum of the heights of *p's* children.

Based on this recursive definition, we can write a method that computes the height of the subtree rooted at a given position. A Java implementation of the method is shown below:

Code Segment 3.18

```
1   public int height(Position<E> p) throws IllegalArgumentException {
2     int h = 0;                            // base case if p is external
3     for (Position<E> c : children(p))
4       h = Math.max(h, 1 + height(c));
5     return h;
6   }
```

Then, to compute the height of a tree, we just need to pass the root node of the tree to this method. The running time of this method is $O(n)$.

The running time of $O(n)$ is obtained in the following way. Initially the *height* method is invoked on p = root of the tree. In line 4, it is recursively invoked on each of its children. Then each of its children invokes the method on its own children, and so on. So, the *height* method is invoked once on each position (or each node) of the tree.

In each activation instance of the *height* method, the maximum of the heights of *p*'s children is computed in the *for* loop of lines 3 and 4. Let c_p be the number of children of a position p. We do not have a concrete implementation of *children*(p) yet. But, we can assume that the time to retrieve the children of p is proportional to the number of its children, c_p, or $O(c_p)$. We can also assume that determining the largest of the heights of c_p children takes $O(c_p)$. So, under these assumptions, we can argue that it takes $O(c_p)$ time to compute the maximum height in the *for* loop. Ignoring other operations that take constant time and also ignoring the recursive execution time of the method itself (in line 4), we can say that each invocation of the *height* method takes $O(c_p)$ time.

In summary, the *height* method is invoked for each position *p* and each invocation takes $O(c_p)$ time. So, the total running time is the summation of $O(c_p)$ over all positions, or

$$\sum_p O(c_p) = O(\sum_p c_p)$$

The term $\sum_p c_p$ represents the number of children of the root node (recall that the method is invoked on the root node initially), and children of them, and so on. So, it is the total number of nodes in the tree except the root node, or *n* – 1. So, the total running time of the *height* method, when invoked on the root node of a tree, is $O(n-1) = O(n)$.

These two methods are added to a revised *AbstractTree* class.

Also added to the revised *Abstract Tree* class is the *positions* method. The *positions* method retrieves all positions in the tree as an iterable collection of positions. This method is relatively easy to implement using a *tree traversal* method (We did not discuss the tree traversal methods yet. They will be discussed in Section 3.2.4). A tree traversal method visits all positions (or nodes) in a given tree in a systematic way. Assuming such a traversal method is available, then the *positions* method can simply collect all positions along the way while the tree traversal method visits all positions in the tree.

A complete code of the *AbstractTree* class, which includes other additional methods, can be found at the *AbstractTree.java* file.

Test Yourself 3.1

Let *T* be a tree with *n* positions. The lowest common ancestor (LCA) of two positions *p* and *q* is defined to be the lowest position in *T* that has both *p* and *q* as descendants (where we allow a position to be a descendant of itself). Write a pseudocode of an efficient algorithm that finds the LCA of *p* and *q*.

Please think carefully, write your program, and then click "Show Answer" to compare yours to the possible algorithm.

Answer - one possible algorithm:

```
Algorithm LCA(Node p, Node q):
  int p_depth = p.depth
  int q_depth = q.depth
  while p_depth > p_depth do
    p = p.parent
  while q_depth > p_depth do
    q = q.parent
  while p ≠ q do
    p = p.parent
    q = q.parent
  return p
```

Section 3.2.2 Binary Trees

A binary tree is an ordered tree with the following properties:

1. Every node has at most two children.
2. Each child node is labeled as being a *left child* or a *right child*.
3. A left child precedes a right child in the order of children of a node.

The subtree rooted at the left or right child of an internal node *v* is called the *left subtree* or the *right subtree*, respectively, of *v*.

A binary tree is *proper* if each node has either zero or two children. Such a binary tree is also referred to as *full binary tree* by some people. So, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is *improper*.

A decision tree, in general, simulates a decision making process. Each internal node *v* represents a test or a question which requires a decision. Each child of the node *v* represents a certain action that is taken based on the outcome of the test or answer to the question. If a child is an internal node, then the action associated with the child is another test or question. If a child is a leaf node, then the action of the child is the *final* decision. If the outcome of a test or the answer to a question is binary, such as yes or no, then the decision tree becomes a binary tree.

The following tree is an example of a binary tree, which is a decision tree:

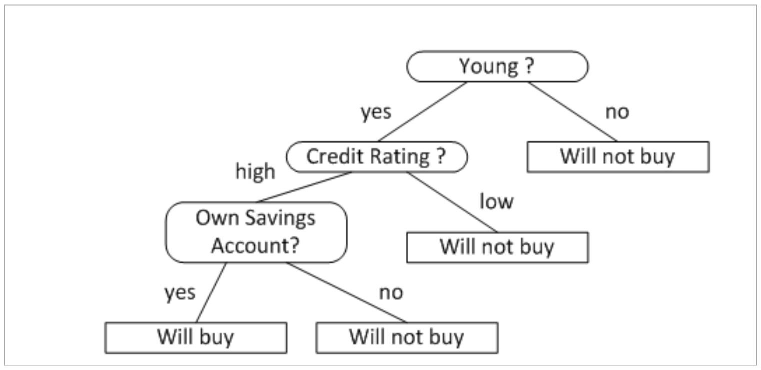


Figure 3.12 A binary tree, which is a decision tree.

This decision tree is used to predict whether a customer would buy or would not buy an expensive sports car. An internal node represents a test on a certain aspect of potential buyers. Based on the outcome of the test on an internal node, we move either to the left or to the right. A leaf node represents the final decision, which in this case is a prediction as to whether a customer would or would not buy a car. This binary tree is a proper binary tree.

As an additional example, we show below a binary tree which represents a mathematical expression. In the tree, an internal node represents an arithmetic operator and a leaf node represents a constant or a variable. Each node is associated with a value, which is determined in the following way:

- If a node is a leaf, then its value is that of its constant or variable.
- If a node is an internal node, then its value is the result of applying the operator of the node to the values of its two children.

If we allow only binary operators, then a mathematical expression tree is a proper binary tree.

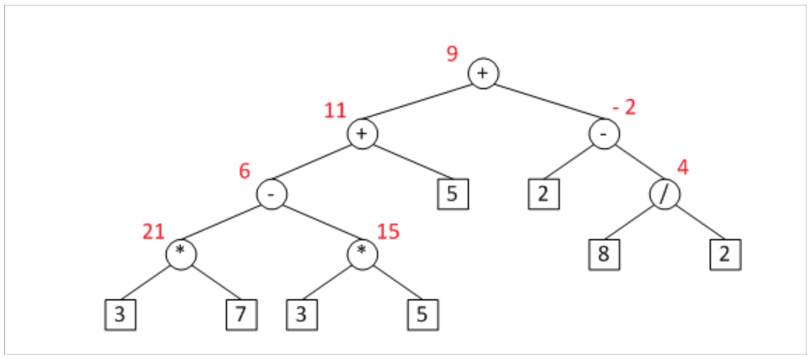


Figure 3.13 A mathematical expression is represented as a binary tree.

The tree corresponds to the expression $((((3 \times 7) - (3 \times 5)) + 5) + (2 - (8/2)))$. The value of each internal node is shown next to the node, and the value of the whole expression is 9.

A binary tree can be recursively defined as follows. A binary tree is either:

- An empty tree, or
- A nonempty tree with a root node *r* and two binary trees that are the left subtree and the right subtree of *r*. One or both of these subtrees can be empty, by definition.

Section 3.2.2.1 Binary Tree Abstract Data Type

The binary tree ADT is a specialization of the *Tree ADT*, which was discussed in Section 3.2.1.2. This ADT supports the following additional accessor methods:

- left(*p*): Returns the position of the left child of *p*. Returns null if *p* has no left child.
- right(*p*): Returns the position of the right child of *p*. Returns null if *p* has no right child.
- sibling(*p*): Returns the position of the sibling of *p*. Returns null if *p* has no sibling.

The following interface defines the binary tree ADT.

Code Segment 3.19

```
public interface BinaryTree<E> extends Tree<E> {

  Position<E> left(Position<E> p) throws IllegalArgumentException;
  Position<E> right(Position<E> p) throws IllegalArgumentException;
  Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

The *BinaryTree* interface extends the *Tree* interface (which was described in Section 3.2.1.2), and defines three more methods to support behaviors that are specific to binary trees.

We now defined the *AbstractBinaryTree* class, which extends the *AbstractTree* class and implements the *BinaryTree* interface. This *AbstractBinaryTree* class serves as an abstract base class for the implementation of a concrete binary tree class, which we will discuss shortly.

Before we discuss the concrete methods implemented in this class, it would be helpful if we visualize the hierarchy of relevant classes and interfaces. The hierarchy diagram is shown below. In the diagram, for each class or interface, the module section where it is described is also indicated.

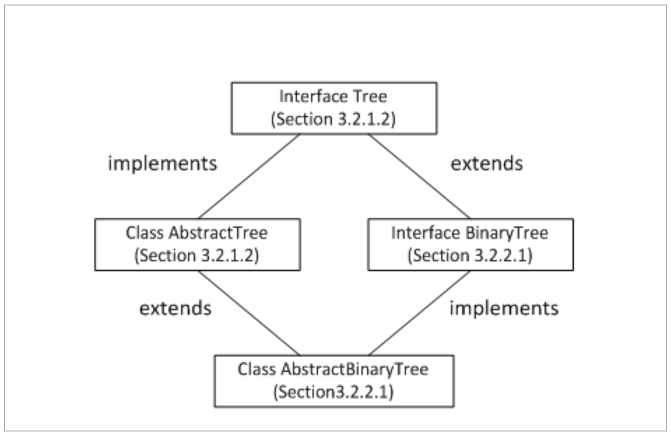

Interface Tree (Section 3.2.1.2)
implements
extends
Class AbstractTree (Section 3.2.1.2)
Interface BinaryTree (Section 3.2.2.1)
extends
implements
Class AbstractBinaryTree (Section3.2.2.1)

Figure 3.14 AbstractBinaryTree inheritance hierarchy.

The *AbstractBinaryTree* class defines three concrete methods – *sibling*, *numChildren*, and *children*. The *sibling* method is a concrete implementation of the *sibling* method which is specified in the *BinaryTree* interface. The *numChildren* is defined in the *AbstractTree* class, and it is overridden and redefined for binary trees. The *children* method is specified in the *Tree* interface but it was not implemented in the *AbstractTree* class. It is implemented in this *AbstractBinaryTree* class. The Java implementations are shown below:

Code Segment 3.20

```
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
                                          implements BinaryTree<E> {
  public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    if (parent == null) return null;  // p must be the root
    if (p == left(parent))            // p is a left child
      return right(parent);           // (right child might be null)
    else                              // p is a right child
      return left(parent);            // (left child might be null)
}

  public int numChildren(Position<E> p) {
    int count=0;
    if (left(p) != null)
      count++;
    if (right(p) != null)
16        count++;
17      return count;
18    }

19    public Iterable<Position<E>> children(Position<E> p) {
20      List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2
21      if (left(p) != null)
22        snapshot.add(left(p));
23      if (right(p) != null)
24        snapshot.add(right(p));
25      return snapshot;
26    }
27  }
```

The implementations of the *sibling* method and the *numChildren* method are straightforward.

The *children* method returns the children of a node in the *ArrayList* object *snapshot.* In line 21 a left child is added first, if any. Then, it checks whether there is a right child. If there is, it is also added to *snapshot*. So, children are ordered in the *snapshot* from left to right.

Section 3.2.2.2 Binary Tree Properties

Let *level d* of a binary tree *T* be the set of nodes at depth *d* of *T*, as shown below:

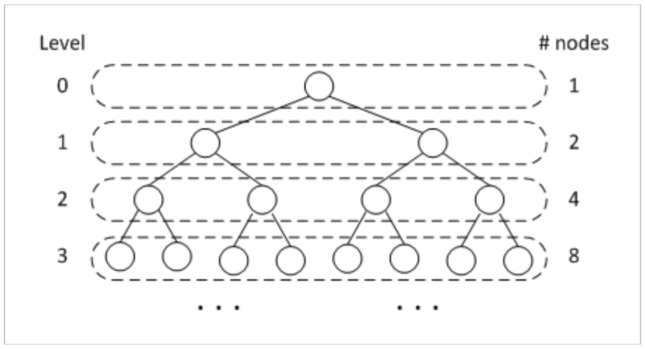


Figure 3.15 Levels of a binary tree.

As we go down a binary tree, the maximum number of nodes at each level doubles. At level d, the maximum number of nodes is 2^d. So, it grows exponentially. We can derive other interesting properties form this observation. First, we define the following notations with regard to a binary tree *T*:

- n: the number of nodes in T
- n_E: the number of external nodes in T
- n_I: the number of internal nodes in T
- h: the height of T

T has the following properties:

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log\,(n + 1) - 1 \leq h \leq n - 1$

If T is proper, it has the following properties:

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$

4. $\log(n+1) - 1 \le h \le (n-1)/2$

There is another property with regard to the number of external nodes and the number of internal nodes. That is:

In a nonempty proper binary tree T, $n_E = n_I + 1$.

We can prove this property in different ways. One simplest way is to use the properties of proper binary trees listed above. We take the second property (about the number of external nodes) and the third property (about the number of internal nodes) and subtract the latter from the former.

$$\begin{array}{rr} & h+1 \le n_E \le 2^h \\ - & h \le n_I \le 2^h - 1 \\ \hline & 1 \le n_E - n_I \le 1 \end{array}$$

This leads to $n_E - n_I = 1$, or $n_E = n_I + 1$. So, in a proper binary tree, the number of external nodes is always one more than the number of internal nodes.

Another proof is shown in our textbook. Students are encouraged to read and understand the proof procedure.

Section 3.2.3 Implementing Trees

We previously defined an abstract base class for general trees (the *AbstractTree* class) and an abstract base class for binary trees (the *AbstractBinaryTree* class). In this section we discuss a concrete implementation of binary trees (note that abstract classes cannot be instantiated). We choose an underlying storage data structure, we add more implementation details, and we add update methods.

Section 3.2.3.1 Linked Structure for Binary Trees

We choose a *linked structure* to implement binary trees. A node, represented by a position p, keeps a reference to the element it stores and references to its parent and children. If p is the root node of a tree, then its parent reference is null. A reference to a left child or a right child is null, if p has no left child or right child, respectively.

The tree itself keeps a reference to the root node and the number of nodes in the tree, *size*.

The node structure and the structure of a binary tree with five nodes are shown below. The elements in the tree are integers.

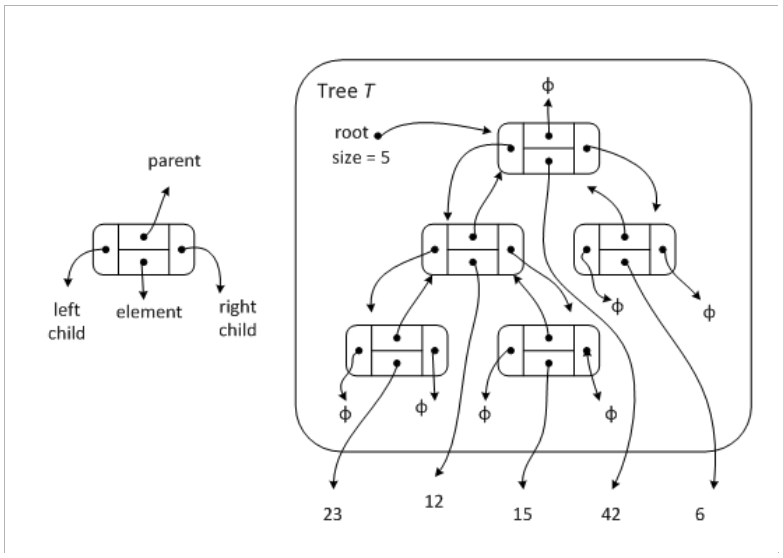


Figure 3.16 Implementation of a binary tree using a linked structure.

This implementation extends *AbstractBinaryTree* abstract class. It adds the following update methods:

- addRoot(e: Creates a new node with element e and make it the root of an empty tree. Returns the position of the root. An error occurs if the tree is not empty.

- addLeft(p, e): Creates a new node with element e and make it a left child of position p. Returns the position of the new node (left child). An error occurs if p already has a left child.
- addRight(p, e): Creates a new node with element e and make it a right child of position p. Returns the position of the new node (right child). An error occurs if p already has a right child.
- set(p, e): Replaces the element of p with element *e*. Returns the previously stored element.
- attach(p, *T*1, *T*2): Attaches internal structure of *T*1 and *T*2 as the left subtree and the right subtree, respectively, of a leaf node position p and resets *T*1 and *T*2 to empty trees. If p is not a leaf node, an error occurs.
- remove(p): Removes the node at position p, replacing it with its child (if any). Returns the element that had been stored at p. An error occurs if p has two children.

All the above methods run in $O(1)$ time.

The *LinkedBinaryTree* class is a Java implementation of this design. The declaration part is shown below:

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {
. . .
```

A node of a tree is implemented as a nested class *Node*, which is shown below:

Code Segment 3.21

```
1   protected static class Node<E> implements Position<E> {
2     private E element;          // an element stored at this node
3     private Node<E> parent;     // a reference to the parent node (if any)
4     private Node<E> left;       // a reference to the left child (if any)
5     private Node<E> right;      // a reference to the right child (if any)

6     public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
7       element = e;
8       parent = above;
9       left = leftChild;
10      right = rightChild;
11    }
    // accessor methods
12    public E getElement() { return element; }
13    public Node<E> getParent() { return parent; }
14    public Node<E> getLeft() { return left; }
15    public Node<E> getRight() { return right; }
    // update methods
16    public void setElement(E e) { element = e; }
17    public void setParent(Node<E> parentNode) { parent = parentNode; }
18    public void setLeft(Node<E> leftChild) { left = leftChild; }
19    public void setRight(Node<E> rightChild) { right = rightChild; }
20  }
```

The *Node* class is declared as *protected*. It has four instance variables – *element*, *parent*, *left*, *and right*. Lines 12 through 15 define accessor methods and lines 16 through 19 defines update methods.

The *LinkedBinaryTree* class itself has the following two instance variables:

```
protected Node<E> root = null;
private int size = 0;
```

It has thirteen methods in addition to a constructor. We briefly *addRoot* method, *addLeft* method, *attach* method, and *remove* method.

The following is a Java code implementing the *addRoot* mthod:

Code Segment 3.22

```
1   public Position<E> addRoot(E e) throws IllegalStateException {
2     if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
3     root = createNode(e, null, null, null);
4     size = 1;
5     return root;
6   }
```

If the tree is not empty, then an *IllegalStateException* is thrown in line 2. In line 3, a new node with element e is created and the root of the tree is set to this new node. The size of the tree is set to 1 in line 5 and the root position is returned in lines 4 and 5.

A Java code implementing the *addLeft* method is shown below:

Code Segment 3.23

```
1  public Position<E> addLeft(Position<E> p, E e)
2                           throws IllegalArgumentException {
3    Node<E> parent = validate(p);
4    if (parent.getLeft() != null)
5      throw new IllegalArgumentException("p already has a left child");
6    Node<E> child = createNode(e, parent, null, null);
7    parent.setLeft(child);
8    size++;
9    return child;
10 }
```

Line 3 checks that the position p is a valid position in the binary tree. Lines 4 and 5 ensure that the position p does not already have a left child. Line 6 creates a new node with element e and line 7 makes the new node the left child of p.

The *attach* method receives three arguments – a position p, a tree *t*1, another tree *t*2. Then, the internal structure of *t*1 becomes the left child of p and the internal structure of *t*2 becomes the right child of p. The following figure shows a high-level description of this method.

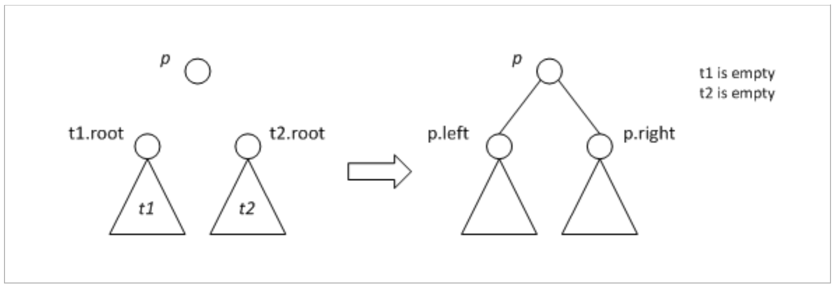


Figure 3.17 Illustration of the attach method.

A Java code is shown below:

Code Segment 3.24

```
public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                  LinkedBinaryTree<E> t2) throws IllegalArgumentException {
  Node<E> node = validate(p);
  if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
  size += t1.size() + t2.size();
  if (!t1.isEmpty()) {                  // attach t1 as left subtree of node
    t1.root.setParent(node);
    node.setLeft(t1.root);
    t1.root = null;
    t1.size = 0;
11    }
12    if (!t2.isEmpty()) {                  // attach t2 as right subtree of node
13      t2.root.setParent(node);
14      node.setRight(t2.root);
15      t2.root = null;
16      t2.size = 0;
17    }
18  }
```

Line 3 validates the position p and stores it in a local variable node. Line 4 checks whether p is a leaf node. Line 5 updates the size of the tree to include all nodes from $t1$ and $t2$. Line 6 makes p (*node* is p) the parent of $t1$ and line 7 makes the root of $t1$ the left child of p. Lines 9 and 10 reset $t1$ to an empty tree. Lines 12 through 16 attaches $t2$ to p as its right child in the same way.

The *remove* method removes a node from a binary tree. Note that this *remove* method is not a general removal method which removes an arbitrary node from a binary tree. This method removes a node only if it has zero or one child. A Java code is shown below:

Code Segment 3.25

```
1   public E remove(Position<E> p) throws IllegalArgumentException {
2     Node<E> node = validate(p);
3     if (numChildren(p) == 2)
4       throw new IllegalArgumentException("p has two children");
5     Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight() );
6     if (child != null)
7       child.setParent(node.getParent()); // child's grandparent becomes its parent
8     if (node == root)
9       root = child;                    // child becomes root
10    else {
11      Node<E> parent = node.getParent();
12      if (node == parent.getLeft())
13        parent.setLeft(child);
14      else
15        parent.setRight(child);
16    }
17    size--;
18    E temp = node.getElement();
19    node.setElement(null);             // help garbage collection
20    node.setLeft(null);
21    node.setRight(null);
22    node.setParent(node);
23    return temp;
24  }
25 }
```

Line 2 validates the position p and lines 3 and 4 make sure that p has only one or no child. The following describes important parts of the code.

Lines 6 and 7: If *child* is not null, i.e., it has either a left child or a right child (Note that, in the code the variable *child* can be null), then we first set the child's parent reference to the parent of p.

Lines 8 and 9: If p is the root of the tree, then *child* becomes the new root of the tree. This code segment works regardless of whether p has a child or not. Suppose p does not have a child, which means that the tree has only the root node, p. Then, the variable *child* is null and the root of the tree is set to null, effectively making the tree an empty tree. If *p* has a child (*child* is not null), then that child becomes the root of the tree.

Lines 10 – 16: This code segment establishes a link between p's parent and p's child. This code segment also works whether p has a child or not. If it has a child, then that child will be the child of p's parent. If it does not have a child, then null will be set. At this time, p is completely removed from the tree.

The remaining part of the code deallocates memory from p and makes it an invalid node.

The complete code of this implementation can be found at the *LinkedBinaryTree.java* file.

The following table summarizes the running times of the methods in this implementation:

Figure 3.18 Running Times of LinkedBinaryTree Operations

Method	**Running Time**
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Section 3.2.3.2 Array-Based Representation of Binary Trees

We can also represent a binary tree with an array. We assign a position number to each node and these numbers are used as indexes in the array. Let $f(p)$ be the integer assigned to a position p in a binary tree T. Then, $f(p)$ is defined as follows:

- If p is the root of T, then $f(p) = 0$.
- If p is the left child of position q, then $f(p) = 2f(q) + 1$.
- If p is the right child of position q, then $f(p) = 2f(q) + 2$.

This numbering scheme is referred to as a *level numbering* of positions in a binary tree T. At each level, nodes are numbered from left to right in increasing order. Note that this numbering method assumes that nodes at each level are *full*, meaning all positions at a level are occupied. At level d, there are 2^d possible positions. So, if the number of nodes at level d is less than 2^d, then not all level numberings of that level are used and some node numbers may not be consecutive.

In the tree (a) of Figure 3.19, all levels are full. So, all level numberings are used at each level and all node numbers are consecutive from 0 to $n - 1$, where n is the number of nodes in the tree. In the tree (b), at level 3, 8 level numberings, 7 to 14, are available. However, there are only four nodes so only four level numberings are used and they are not consecutive.

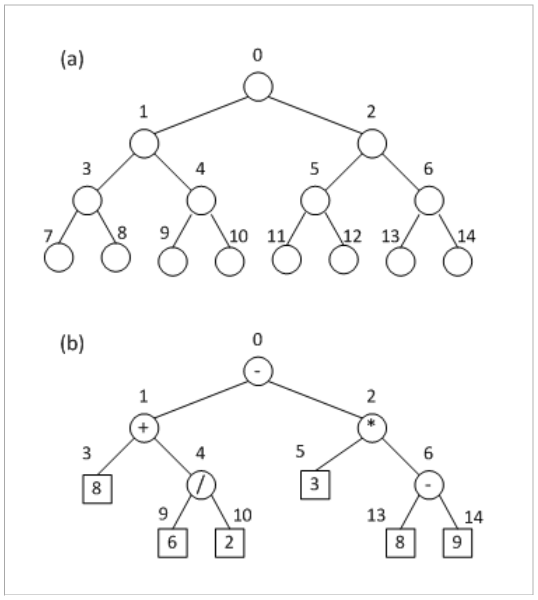


Figure 3.19 Level numbering of positions in a binary tree.

Then, we can store node elements in an array using the node numbers as indexes. The following array is the array representation of the tree (b):

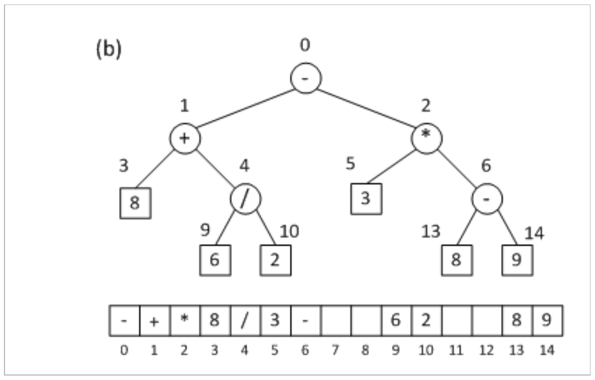


Figure 3.20 Storing binary tree elements in an array using node numbers as indexes.

In the array representation, it becomes easy, and fast, to access the parent and children nodes of a given node. Given the node number of a position p, $f(p)$, the indexes of those nodes can be calculated as follows:

- Index of parent of $p = \lfloor (f(p) - 1)/2 \rfloor$
- Index of left child of $p = 2f(p) + 1$
- Index of right child of $p = 2f(p) + 2$

One disadvantage of an array representation is potential waste of storage space. In general, the storage required for an array representation depends on the shape of a tree. In the worst case, there is only one node at each level, and the size of the array $N = 2^n - 1$, where n is the number of nodes in a tree. For example, suppose that there are four nodes in a tree. The following tree is a worst-case shape of a tree with four nodes:

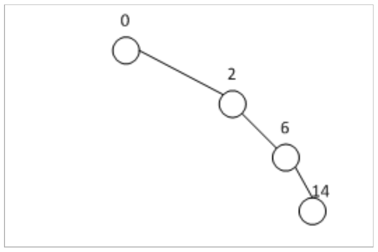


Figure 3.21 Worst-case shape of a binary tree.

To represent this tree, we need an array of size 15, and only 4 out of 15 locations are utilized.

Section 3.2.3.3 Linked Structure for General Trees

In a general tree, the number of children of a node is not fixed. There may be no child, or there may be as many children as required by a particular application where the tree is used. One way of storing children of a node in a general tree is to use a *container*.

A node has three fields – a reference to its parent, a reference to the element stored in the node, and a reference to a container object. The container object stores references to the children of the node. The node structure and an example of a general tree, which stores names of U.S. states, are shown below:

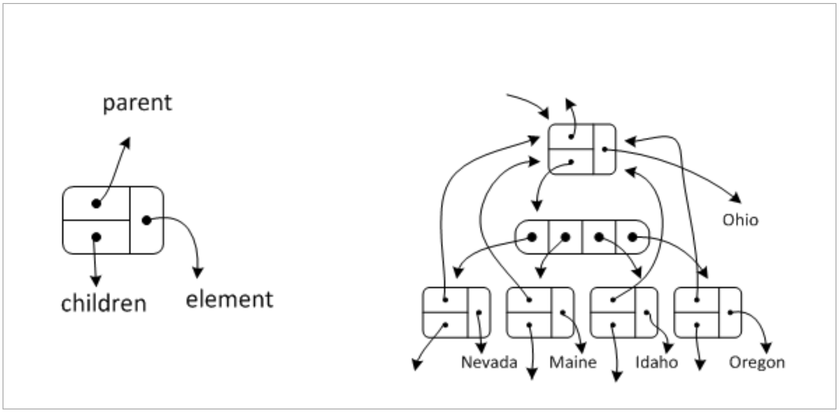


Figure 3.22 Linked structure for general trees.

We can implement the basic accessor methods and query methods of general trees in a similar manner as we implemented linked binary trees. In this implementation, the retrieval of children of a node can be achieved by iterating over the container object which stores the references to children. The running times of some basic methods are shown below:

Figure 3.23 Performance of General Trees Implemented Using Linked Structure.

Method	Running Time
size, isEmpty	$O(1)$
root, parent, isInternal, isExternal, isRoot	$O(1)$
numChildren(p)	$O(1)$
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Section 3.2.4 Tree Traversal

A *traversal* of a tree *T* is a systematic way of visiting all positions in *T*. When a position is *visited*, usually certain processing is performed. It may be a simple operation of incrementing a counter or a complex operation performed on an object stored in that position. In this section we discuss different tree traversal algorithms, implementation of some algorithms, and a few example applications.

Section 3.2.4.1 Preorder and Postorder Tree Traversal of General Trees

A *preorder* tree traversal of a tree *T* visits the root of *T* first and then visits its subtrees recursively. The "pre" of *preorder* denotes that the root is visited *before* its subtrees are visited.

The pseudocode of a preorder tree traversal algorithm is shown below:

```
Algorithm preorder(p)
  visit p
  for each child c in children(p)
    preorder(c)
```

If the tree is an ordered tree, subtrees are visited by the order of the children. The following figure illustrates preorder traversal of a tree:

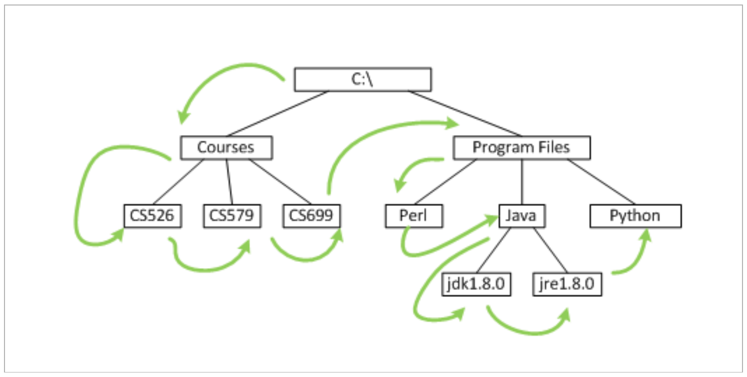


Figure 3.24 Illustration of preorder tree traversal.

In the *postorder* tree traversal, as the "post" implies, the root of a tree is visited *after* all its subtrees have been visited. The pseudocode of a postorder tree traversal algorithm and an illustration is shown below:

```
Algorithm postorder(p)
  for each child c in children(p)
    postorder(c)
  visit p
```

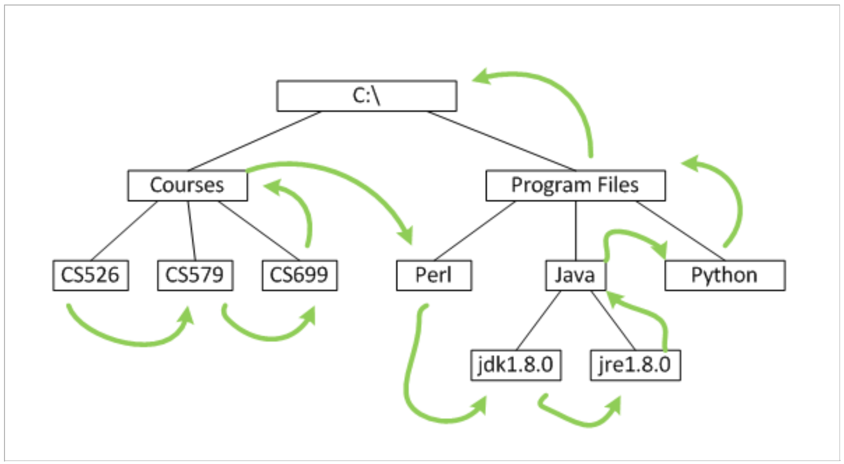


Figure 3.25 Illustration of postorder tree traversal.

The running time of preorder algorithm or postorder traversal algorithm is *O*(*n*). This can be determined in a similar way that the running time of the *height* method was determined (refer to Section 3.2.1.3).

Section 3.2.4.2 Breadth-First Tree Traversal

There is another way of traversing a tree. A *breadth-first tree traversal* visits all nodes at depth *d* before it visits nodes at depth $d + 1$. A breadth-first tree traversal is also referred to as *breadth-first-search*, or *BFS*. We will discuss *BFS* on a general graph in Module 6.

An example of a breadth-first traversal is a decision process in a two-player game, such as a game of tic-tac-toe or a chess game. At a certain point during a game, a plyer would consider many possible moves, and for each of these moves, will examine opponent's multiple responses. Then, for each response, the player would consider, again, many possible counter-moves, and so on. This process can be depicted as a tree, which is called *game tree*. The following figure shows a part of such a game tree with a tic-tac-toe game as an example:

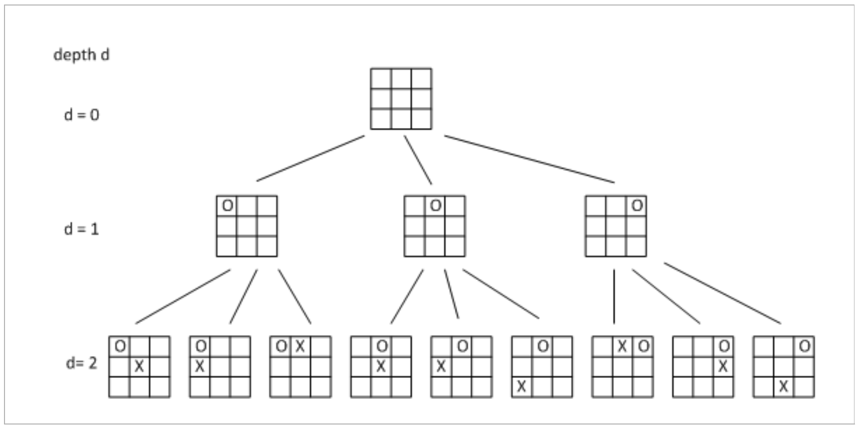


Figure 3.26 A game tree representing a tic-tac-toe game.

A computer program that plays a tic-tac-toe game would consider all options at depth 1 and examine opponent's all moves at depth 2, and so on. A breadth-first traversal is illustrated (with a smaller tree) in the following figure:

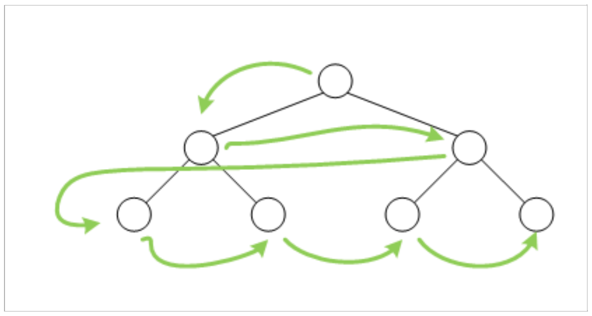


Figure 3.27 Illustration of breadth-first traversal.

The pseudocode of a breadth-first traversal is shown below. This algorithm uses a FIFO queue to order the visits to the nodes in a tree.

```
Algorithm breadthfirst( )
  initialize Q to contain the root of the tree
  while Q is not empty
    p = Q.dequeue( )  // remove the oldest entry in Q
    visit p
    for each child c in children(p)
      Q.enqueue(c)    // add all children of p to the rear of Q
  visit p
```

In the algorithm, each node is enqueued and dequeued once each. So, *enqueuer* is called *n* times and *dequeuer* is also invoked *n* times. Therefore, the running time of this algorithm is $O(n)$.

Section 3.2.4.3 Inorder Traversal of Binary Tree

The three tree traversal methods we discussed in previous sections can be also used for binary trees. In this section, we describe another tree traversal method that applies to binary trees.

In an inorder tree traversal, the left subtree of the root is visited and, after that, the right subtree of the root is visited. Then, the root is visited. The "in" in *inorder* denotes the *in-between* visit to the root. The pseudocode is given below:

```
Algorithm inorder(p)
         if p has a left child lc    // visit left subtree
           inorder(lc)
         visit p
```

```
            if p has a right child rc   // visit right subtree
              inorder(rc)
```

The following figure illustrates an inorder tree traversal of a mathematical expression tree:

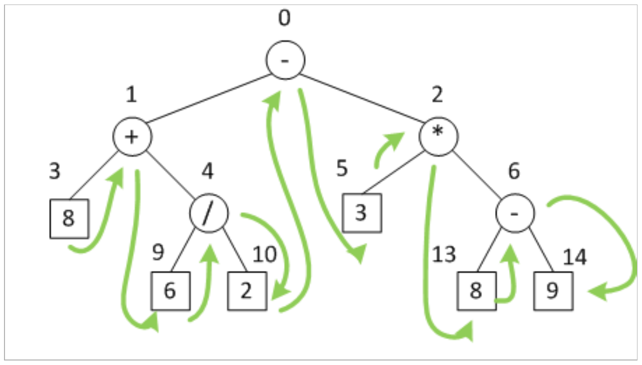


Figure 3.28 Illustration of inorder tree traversal.

If we print the element in each node as we visit the nodes, we will have:

8 + 6 / 2 – 3 * 8 - 9

This is a correct expression corresponding to the tree, except that there are no parentheses.

A binary search tree is a binary tree with additional properties. Let's assume that a binary search tree stores elements from a set *S* and that there is an ordering among all elements in *S*. Then, a binary tree for *S* is a proper binary search tree *T* if, for each position *p* in *T*, the following properties are satisfied:

- Each a position *p* stores an element from *S*, denoted as *e*(*p*).
- All elements in the left subtree of a position *p* (if any) are less than *e*(*p*).
- All elements in the right subtree of a position *p* (if any) are greater than *e*(*p*).

An example of a binary search tree is shown below, which stores integers.

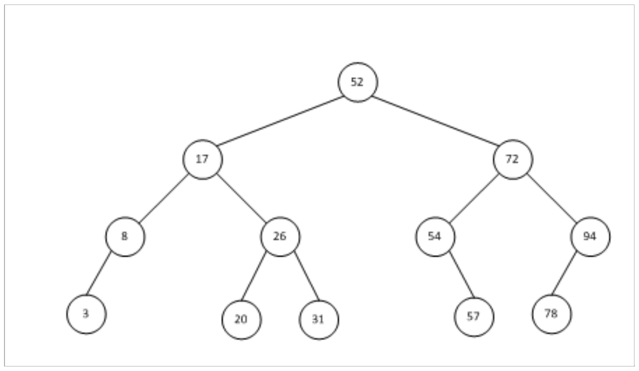


Figure 3.29 An example of a binary search tree.

If we visit the nodes with inorder tree traversal and print the integers stored in each node, we will have the following sequence:

3, 8, 17, 20, 26, 31, 52, 54, 57, 72, 78, 94

As we can see, the nodes are be visited in increasing order of the integers they store in them.

Section 3.2.4.4 Implementing Tree Traversals in Java

In this section we describe how to implement tree traversal algorithm, which were discussed in the previous sections, in Java.

First, we describe implementation of a preorder tree traversal, *preorder* method. This method returns positions of all nodes in the tree as an *ArrayList* where positions are ordered by preorder traversal. Note that an *ArrayList* is an iterable collection. So, an application, or other method, can create an iterator over the

ArrayList object returned by the *preorder* method to perform appropriate operations. In this section, we only discuss how to implement the *preorder* method.

This implementation uses a helper method which allows a parameterized recursion. The helper method is named *preorderSubtree* and it is a direct implementation of the preorder algorithm we discussed in Section 3.2.4.1. The code of the *preorderSubtree* is given below:

Code Segment 3.26

```
1   private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
2     snapshot.add(p);     // add position p before exploring subtrees
3     for (Position<E> c : children(p))
4       preorderSubtree(c, snapshot);
5   }
```

When a node is visited, its position is added to *snapshot*, which is an *ArrayList* of positions. In line 2, the position of p is added and, in the following *for* loop of line 3 and 4, children in its subtree are recursively visited.

Once the *preorderSubtree* is written, then the *preorder* method simply needs to invoke it by passing the root node and an empty *snapshot*. The code of the *preorder* method is shown below:

Code Segment 3.27

```
1   public Iterable<Position<E>> preorder() {
2     List<Position<E>> snapshot = new ArrayList<>();
3     if (!isEmpty())
4       preorderSubtree(root(), snapshot); // fill the snapshot recursively
5     return snapshot;
6   }
```

Line 2 creates a new, empty *ArrayList* named *snapshot*. Then, if the tree is not empty, it calls the *preorderSubtree* method in line 4. In line 5, the *snapshot*, which contains positions all nodes of the tree, is returned.

A postorder traversal can be implemented in the same way. Again, we defined a helper method *postorderSubtree* to implement parameterized recursion. The Java codes of the *postorderSubtree* method and the *postorder* method are shown below:

Code Segment 3.28

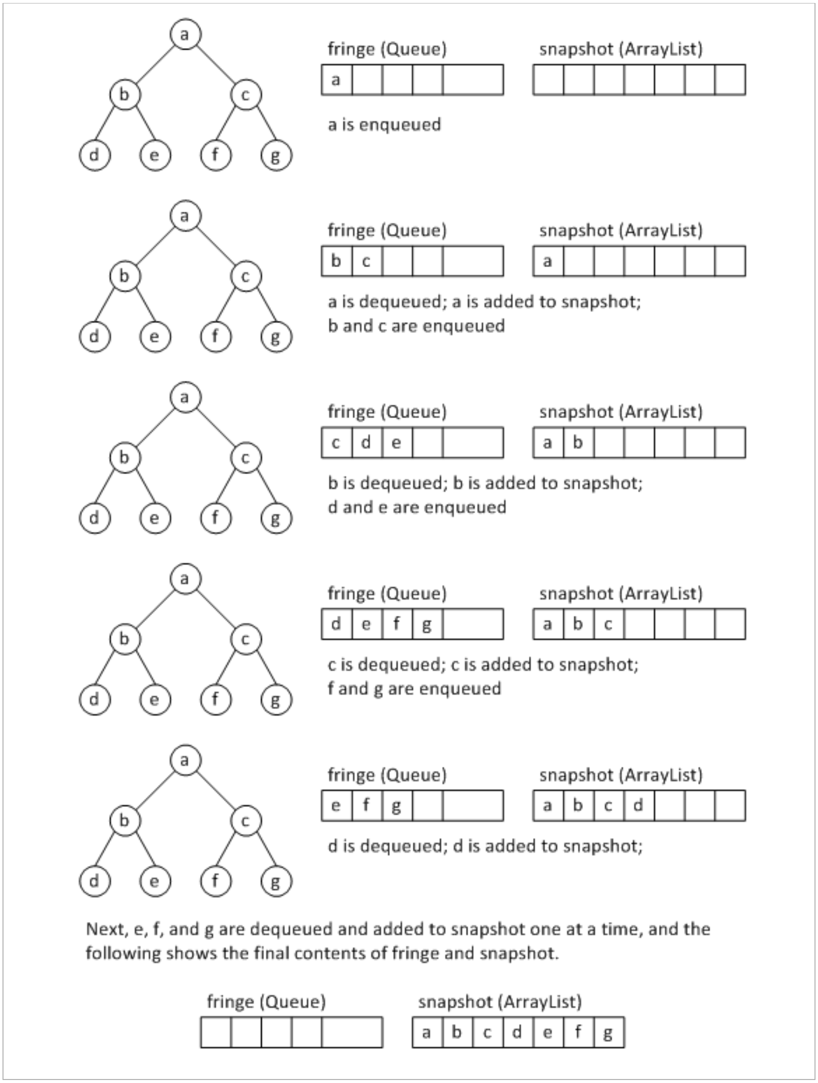
```
1   private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
2     for (Position<E> c : children(p))
3       postorderSubtree(c, snapshot);
4     snapshot.add(p);        // add position p after exploring subtrees
5   }

6   public Iterable<Position<E>> postorder() {
7     List<Position<E>> snapshot = new ArrayList<>();
8     if (!isEmpty())
9       postorderSubtree(root(), snapshot); // fill the snapshot recursively
10    return snapshot;
11  }
```

The breadth-first traversal algorithm discussed in Section 3.2.4.2 uses a queue to properly order the visits to the nodes. We first show a Java code and then we illustrate a traversal of a tree.

Code Segment 3.29

```
1  public Iterable<Position<E>> breadthfirst() {
2    List<Position<E>> snapshot = new ArrayList<>();
3    if (!isEmpty()) {
4      Queue<Position<E>> fringe = new LinkedQueue<>();
5      fringe.enqueue(root());             // start with the root
6      while (!fringe.isEmpty()) {
7        Position<E> p = fringe.dequeue(); // remove from front of the queue
8        snapshot.add(p);                  // report this position
9        for (Position<E> c : children(p))
10          fringe.enqueue(c);             // add children to back of queue
11     }
12    }
13    return snapshot;
14  }
```

Line 2 creates an empty *ArrayList*, which will eventually contain all positions ordered by a breadth-first traversal. In line 4, an empty queue, *fringe*, is created and the root node is enqueued in line 5. From line 6 to line 11, all positions are visited. This process is illustrated in the following figure. In the figure, for the purpose of illustration, nodes are labeled with alphabets.



a is enqueued

a is dequeued; a is added to snapshot; b and c are enqueued

b is dequeued; b is added to snapshot; d and e are enqueued

c is dequeued; c is added to snapshot; f and g are enqueued

d is dequeued; d is added to snapshot;

Next, e, f, and g are dequeued and added to snapshot one at a time, and the following shows the final contents of fringe and snapshot.

Figure 3.30 Illustration of implementation of breadth-first traversal, which uses a queue.

The inorder tree traversal is defined for binary trees because it relies on the explicit notion of a left and a right child of a node. The implementation is similar to those of the preorder traversal and the postorder traversal. Java implementation is shown below:

Code Segment 3.30

```
private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
  if (left(p) != null)
    inorderSubtree(left(p), snapshot);
  snapshot.add(p);
  if (right(p) != null)
    inorderSubtree(right(p), snapshot);
}

public Iterable<Position<E>> inorder() {
  List<Position<E>> snapshot = new ArrayList<>();
  if (!isEmpty())
    inorderSubtree(root(), snapshot);   // fill the snapshot recursively
  return snapshot;
}
```

Test Yourself 3.2

Write a pseudocode of an algorithm that implements the *inorderNext*(*v*) operation on a binary tree *T*. This operation returns the position visited after *v* in an inorder traversal of *T*, and returns null if *v* is the last node visited.

Please think carefully, write your program, and then click "Show Answer" to compare yours to the possible algorithm.

Answer – one possible algorithm:

```
if v has a right child
  p = v’s right child
  while (p has a left child) do
      p = p’s left child
  return p
else
  p = v’s parent
  while (p ≠ and v is not a left child of p) do
      v = p
      p = p’s parent
  return p
```

Section 3.2.4.5 Applications of Tree Traversals

In this section, we show two applications where tree traversals are used.

As the first example, we describe how to print the table of contents of a document which has a hierarchical structure among its components.

Let’s assume that we have a book, and that the book has two parts, each part has two chapters, and each chapter has two sections. The components of the book are shown as a tree below:

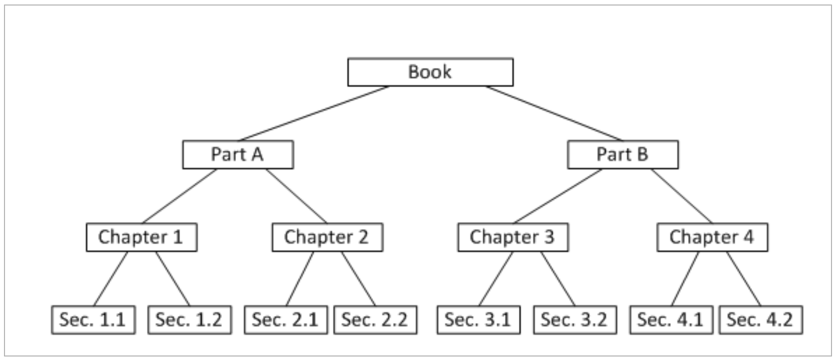


Figure 3.31 A tree representation of book components.

We can print the table of contents of the book using the preorder tree traversal. When we visit a node, we print the name of the book component in that node. If we print each component per line without indentation, it should look like:

```
Part A
Chapter 1
Sec. 1.1
Sec. 1.2
Chapter 1
Sec. 2.1
Sec. 2.2
Part B
Chapter 3
Sec. 3.1
Sec. 3.2
Chapter 4
Sec. 4.1
Sec. 4.2
```

This can be achieved by the following code segment:

```
for (Position<E> p : T.preorder( ))
  System.out.println(p.getElement( ));
```

However, we can print the table of contents in "prettier" way with appropriate indentations, as shown below:

```
Part A
  Chapter 1
    Sec. 1.1
    Sec. 1.2
  Chapter 1
    Sec. 2.1
    Sec. 2.2
Part B
  Chapter 3
    Sec. 3.1
    Sec. 3.2
  Chapter 4
    Sec. 4.1
    Sec. 4.2
```

We can print the table of contents in this way by slightly modifying the above code. As we go down the tree (by recursive calls), we can add appropriate indentations as follow:

```
k = 2;
for (Position<E> p : T.preorder( ))
  System.out.println(spaces(k*T.depth(p), p.getElement( ));
```

Here, the *space*(n) method returns a string of *n* spaces. For example, when a component at depth 1 (such as Chapter 1) is printed, it is preceded by 2 spaces $k \times 1 = 2$, and when a component at depth 2 (such as Sec 1.1) is printed, it is preceded by 4 spaces $k \times 2 = 4$. We can change the amount of indentation by adjusting the value of k.

However, this implementation is not efficient. This is because, in the worst case, the *depth* method is invoked for every position of the tree, which makes the running time $O(n^2)$.

There is an efficient implementation. We can design a recursive algorithm which, as it goes down to lower levels, passes the current depth as a parameter. In this way, we can avoid having to keep invoking the *depth* function for each position. A Java implementation is shown below:

Code Segment 3.31

```
1   public static <E> void printPreorderIndent(Tree<E> T, Position<E> p, int d){
2    System.out.println(spaces(2*d) + p.getElement());  // indent based on d
3     for (Position<E> c : T.children(p))
4       printPreorderIndent(T, c, d+1);                 // child depth is d+1
5   }
```

There are three arguments. The first argument is the tree. The second argument *p* is the position of the node that is to be printed. The third argument *d* is the depth of *p* and it is used to calculate the amount of indentation in line 2. Note that, when the method is invoked recursively on the children of the current node, the depth is incremented in line 4. So, a child component of a current component *v* is always indented by 2 more spaces than *v*.

As the second example, we describe how to compute the disk space usage. Once a file system on a disk is represented as a tree, we can calculate the disk usage by adding up the space used by each node while traversing the tree. When we calculate the disk usage of a directory, for example, we first need to calculate the disk usages of all its subdirectories. So, the postorder traversal is an appropriate choice for this application. The code of a recursive implementation in Java is shown below:

Code Segment 3.32

```
1   public static int diskSpace(Tree<Integer> T, Position<Integer> p) {
2     int subtotal = p.getElement(); // we assume element represents space usage
3     for (Position<Integer> c : T.children(p))
4       subtotal += diskSpace(T, c);
5     return subtotal;
6   }
```

We use the postorder traversal, as mentioned earlier. But, since each invocation on a subdirectory must return the disk usage of the subdirectory, we need to modify the previous postorder traversal implementation (in Section 3.2.4.4).

It is assumed that, to simplify the discussion, the *getElement* method returns the disk space used by the current position p. In line 2, the disk usage of the current position p (or current directory) is stored in the local variable *subtotal.* Then, in the *for* loop of lines 3 and 4, disk usages of all its children (or subdirectories) are computed and then it is added to *subtotal*.

Module 3 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 3.3

This question is with regard to the list ADT described in Section 3.1.1 (it is also described in Section 7.1 of the textbook). Consider the following table. The first column shows a sequence of operations performed on a list of integers. Show the return value and the list content after each operation is performed. Assume that the list is initially empty.

Operation	Return Value	List Contents
add(0, 12)		
add(1, 5)		
add(1, 42)		
add(2, 12)		
get(1)		
size()		
remove(4)		

add(4, 27)		
remove(1)		
get(3)		
add(3)		
set(1, 75)		

Suggested answer:

Test Yourself 3.4

Consider the following method that is specified in the *ArrayList* class, which is described in Section 3.1.2 (the *ArrayList* class is described in Section 7.2 of the textbook).

```
1   public E remove(int i) throws IndexOutOfBoundsException {
2    checkIndex(i, size);
3    E temp = data[i];
4    for (int k=i; k < size-1; k++) // shift elements to fill hole
5      data[k] = data[k+1];
6    data[size-1] = null;           // help garbage collection
7    size--;
8    return temp;
9  }
```

Express the best-case running time, the average-case running time, and the worst-case running time of the method using the *big-oh* notation. You need to justify your answers.

Suggested answer:

Lines 2, 3, 6, 7, and 8 take $O(1)$ time. The running time of the method is determined by the *for* loop of lines 4 and 5. In the best case, $i = size - 1$. In this case, the loop body is not executed and the running time is $O(1)$. In the worst case, $i = 0$, and $n - 1$ elements must be shifted. So, it takes $O(n)$. If we assume that each index is equally likely to be passed as an argument to the method, on average, $n/2$ elements are shifted. So, average-case running time is $O(n)$.

Test Yourself 3.5

This question is with regard to *ArrayList* class (which is implemented as *ArrayList.java*) described in Section 3.1.2 (it is also described in Section 7.2 of the textbook). Suppose that N denotes the capacity of the array that stores the elements of an array list. Write a Java method that reduces the capacity of the array from N to $N/2$ if the number of elements in the list goes below $N/4$.

Suggested answer - one possible solution:

```
protected void resizeToHalf( ) {
  if (size < data.length / 4) {
    E[] temp = (E[]) new Object[data.length / 2]; //create a new array
    for (int k=0; k < size; k++)
      temp[k] = data[k];
    data = temp;      // start using the new array
  }
}
```

Test Yourself 3.6

This question is with regard to the *Positional List* ADT, which is described in Section 3.1.3 (it is also described in Section 7.3.1 of the textbook). Suppose that elements are stored in a list *L*. The *position* of an element does not change even if the index of the element changes in *L* due to insertions or deletions elsewhere in the list. True or False?

Suggested answer: True.

Test Yourself 3.7

This question is with regard to the *Positional List* ADT, which is described in Section 3.1.3 (it is also described in Section 7.3.1 of the textbook). Suppose that *myList* is an instance of a positional list. The method invocation *myList.last*( ) returns the last element in the list. True of False?

True.

Your option is wrong.

False.

Your option is correct. It returns the position of the last element in the list.

Test Yourself 3.8

Section 3.1.3.3 describes a concrete implementation of the positional list ADT using a doubly linked list (it is also described in Section 7.3.3 of the textbook). One of the methods implemented in the *LinkedPositionalList* is the *position* method. It receives a *Node* as its argument and returns a *Position*. What is the purpose of this method?

Suggested answer: The purpose of the method is to not expose the sentinel nodes to a caller, returning a null reference if a sentinel node is passed.

Test Yourself 3.9

An *iterator* is used to scan a sequence of elements, one element at a time. In Java, you can get an iterator by invoking the *iterator*( ) method. This *iterator*( ) method is defined in the *Iterator* interface. True or False?

True.

Your option is wrong.

False.

Your option is correct. It is defined in a class that implements the Iterable interface.

Test Yourself 3.10

Consider the positional ADT operations (methods) described in Section 3.1.3.2 (they are also described in pages 274 and 275 of the textbook). Write a Java code that implements the *addBefore* operation using only methods in {*isEmpty*, *first*, *last*, *before*, *after*, *addFirst*, *addAfter*}.

Suggested answer - one possible solution:

```
public Position<E> addBefore(Position<E> p, E e){
  if (p == first( ))
    return addFirst(e);
  else
    return addAfter(before(p), e);
}
```

Test Yourself 3.11

Suppose you want to add to the positional ADT a new operation *findPosition*(e), which returns the first position containing an element equal to *e* (or null if no such position exists). Write a Java code implementing the operation using only existing operations given in Section 3.1.3.2 (or using only the methods shown in pages 274 and 275 of the textbook).

Suggested answer - one possible solution:

```
public Position<E> findPosition(E e) {
  Position<E> walk = first( );
  while (walk != null && !e.equals(walk.getElement( )))
    walk = after(walk);
  return walk;
}
```

Test Yourself 3.12

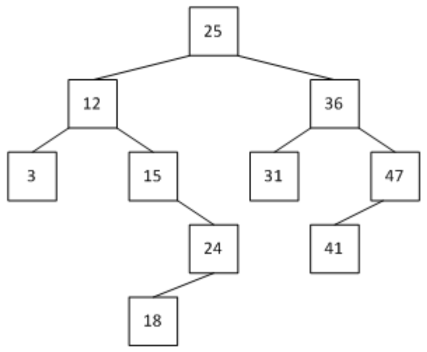
Suppose you want to add to the positional ADT a new operation *positionAtIndex*(*i*), that returns the first position of the element having index *i* (or throws an *IndexOutOfBoundsException*, if necessary). Write a Java code implementing the operation using only existing operations given in Section 3.1.3.2 (or using only the methods shown in pages 274 and 275 of the textbook).

Suggested answer - one possible solution:

```
public Position<E> positionAtIndex(index i) {
if (i < 0 | | i >= size( ))
  throw new IndexOutOfBoundsExcpetion("Invalid index");
  Position<E> walk = first( );
  for (int k=0; k < i; k++)
    walk = after(walk);
  return walk;
}
```

Test Yourself 3.13

Consider the following tree:

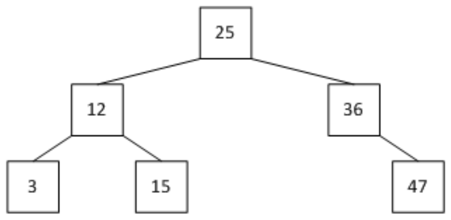


What is the depth of the node with 41? What is the height of the tree?

Suggested answer: Depth of the node with 41 is 3, and the height of the tree is 4.

Test Yourself 3.14

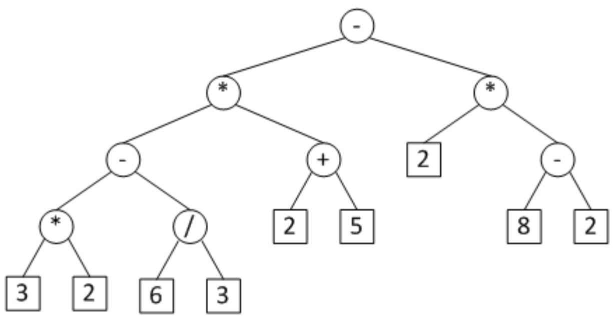
Is the following binary tree a proper binary tree?



Suggested answer: No, because the node with 36 has only one child.

Test Yourself 3.15

Show the mathematical expression that is represented by the following binary tree:



Suggested answer: $((((3 \times 2) - (6/3)) \times (2 + 5)) - (2 \times (8 - 2)))$

Test Yourself 3.16

Let T be a proper binary tree, *h* be the height of T, and n_E be the number of external nodes of T. Prove the following property:

$$h + 1 \leq n_E \leq 2^h$$

Suggested answer:

We prove the property using the mathematical induction on h.

Base case: $h = 1$. A proper binary tree of height $h = 1$ has one root node and two external nodes, i.e., $N_E = 2$. Therefore, the property holds.

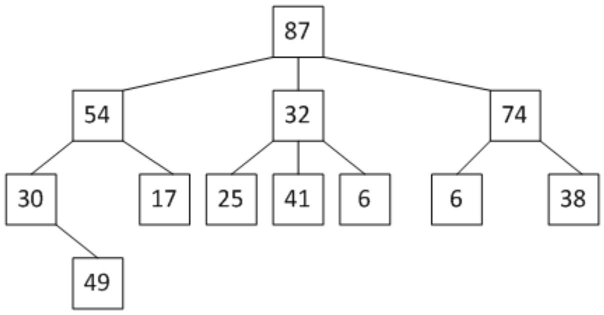
Assume the property is true for $h = k$.

We first show that the minimum number of external nodes for $h = k + 1$ is $k + 2$. Suppose that a proper binary tree T has the minimum number of external nodes $n_E = k + 1$. Growing T by one more level with the minimum number of external nodes can be done by adding two children to an external node e. Then, e now becomes an internal node and the number of external node is decreased by one. Since two new external nodes are created, in total, one more external node is added to T. Therefore, $n_E = k + 2$.

Next, we show that the maximum number of external nodes for $h = k + 1$ is 2^{k+1}. Suppose that a proper binary tree T has the maximum number of external nodes $n_E = 2^k$. We can grow T by one more level with the maximum number of external nodes by adding two children to all external nodes. Then, $2.2^k = 2^{k+1}$ new external nodes are created. Since all existing external nodes have become internal nodes, the total number of external nodes in the new T is 2^{k+1}.

Test Yourself 3.17

The following tree that stores integers in the nodes:



Show the sequence of numbers generated by the postorder traversal of the tree.

Suggested answer: 49, 30 17, 54, 25, 41, 6, 32, 6, 38, 74, 87

Test Yourself 3.18

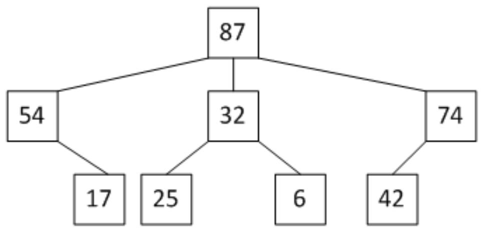
Consider the *breadth-first traversal*, whose pseudocode is shown below:

```
1      Algorithm breadthfirst( )
2        initialize Q to contain the root of the tree
3        while Q is not empty
4          p = Q.dequeue( )  // remove the oldest entry in Q
5          visit p
6          for each child c in children(p)
7            Q.enqueue(c)  // add all children of p to the rear of Q
8        visit p
   }
```

The queue operations are performed in lines 2, 4, and 7, as follows:

Line 2: The root node is enqueued.
Line 4: The node at the front of the queue, *p*, is dequeued.
Line 7: Children of *p* are enqueued, one at a time. Assume children are added from left to right in the tree.

Suppose that the *breadth-first traversal* is performed on the following tree:



After line 2, the queue has {87}.
Show the contents of the queue after each iteration of the while loop. Assume that the front of the queue is on the left.

Suggested answer:

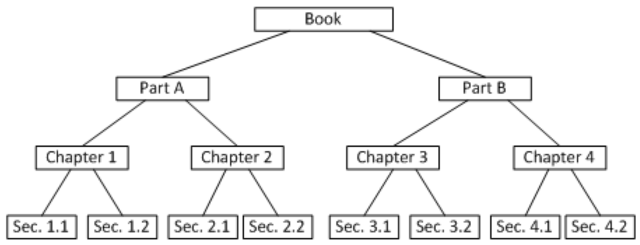
{87}
{54, 32, 74}
{32, 74, 17}
{74, 17, 25, 6}
{17, 25, 6, 42}
{25, 6, 42}
{6, 42}
{42}
{ }

Test Yourself 3.19

Consider the following code segment, which prints indented version of a preorder traversal, which is described in Section 3.2.4.5 (it is also described in Section 8.4.5 of the textbook):

```
public static <E> void printPreorderIndent(Tree<E> T, Position<E> p, int d){
 System.out.println(spaces(2*d) + p.getElement());  // indent based on d
  for (Position<E> c : T.children(p))
    printPreorderIndent(T, c, d+1);                 // child depth is d+1
}
```

Suppose, for example, that the table of contents of a book is represented as the following tree:



Then, the above method will print the table of contents as follows:

```
Part A                      // no indentation
    Chapter 1             // indented by 2 spaces
        Sec. 1.1            // indented by 4 spaces
        Sec. 1.2            // . . .
    Chapter 1
        Sec. 2.1
        Sec. 2.2
Part B
    Chapter 3
        Sec. 3.1
        Sec. 3.2
    Chapter 4
        Sec. 4.1
        Sec. 4.2
```

Question: show the print out of the table of contents when *d* + 1is changed to *d* + 2 in line 4. You need to indicate how many spaces are indented at each level in your answer.

Suggested answer:

```
Part A                                  // no indentation
        Chapter 1                       // indented by 4 spaces
          Sec. 1.1                      // indented by 8 spaces
          Sec. 1.2                      // . . .
        Chapter 1
          Sec. 2.1
          Sec. 2.2
Part B
        Chapter 3
          Sec. 3.1
          Sec. 3.2
        Chapter 4
          Sec. 4.1
          Sec. 4.2
```

Test Yourself 3.20

Show an example of a tree for which a preorder traversal visits the nodes in reverse order of the postorder traversal of the tree.

Suggested answer: A tree with two nodes.

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.) Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.
- Oracle. *The Java Tutorials*.