CS 3410 – Ch 19 – Binary Search Trees

Sections	Pages	Exercises
19.1, 19.3-19.6	687-697, 702-737	

19

The find operation for a linked list is $O(n)$, which can be prohibitive with a large amount of data. In this chapter we introduce the Binary Search Tree which reduces the complexity to $O(\log n)$ on average, but $O(n)$ in the worst-case. Next, we introduce three ways to provide $O(\log n)$ in the worst case.

19.1 – Binary Search Trees

1. We use a *key* to search for an item. For instance, we may use a social security number as the key when we search for a Person object in a data structure. A *binary search tree* is a data structure that stores items based on a key.

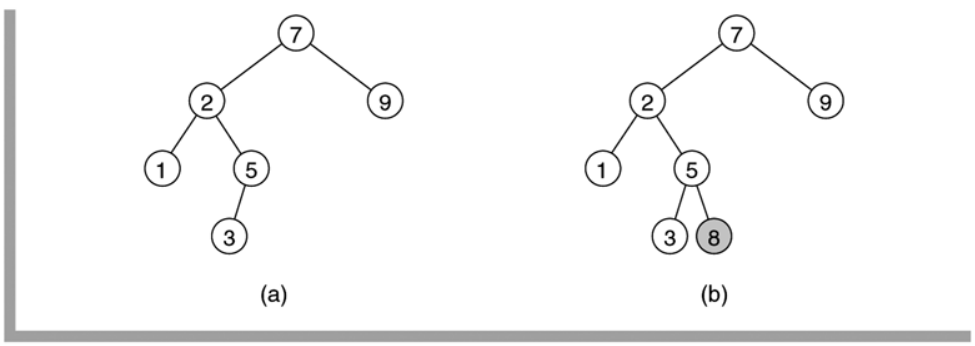
2. A *binary search tree* (BST) is a binary tree that satisfies the *binary search tree order property*:

   For every node, n in a BST:

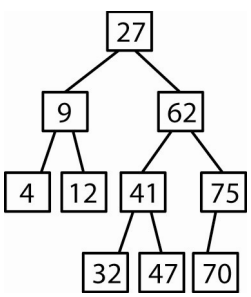
   a. all keys in n's left subtree are less than the key in n
   b. all keys in n's right subtree are greater than the key in n.

3. Examples:

figure 19.1
Two binary trees: (a) a search tree; (b) not a search tree



4. Another Example:



5. There are three important implications of the BST order property:

   a. We get a consistent order. An in-order traversal gives the items in sorted order.
   b. No duplicates are allowed.
   c. The left-most node is the *minimum* and the right-most node is the *maximum*.

6. These are the basic operations we want to be able to perform on a BST:

   find(n), insert(n), findMin(), findMax(), remove(n), removeMin(), removeMax()

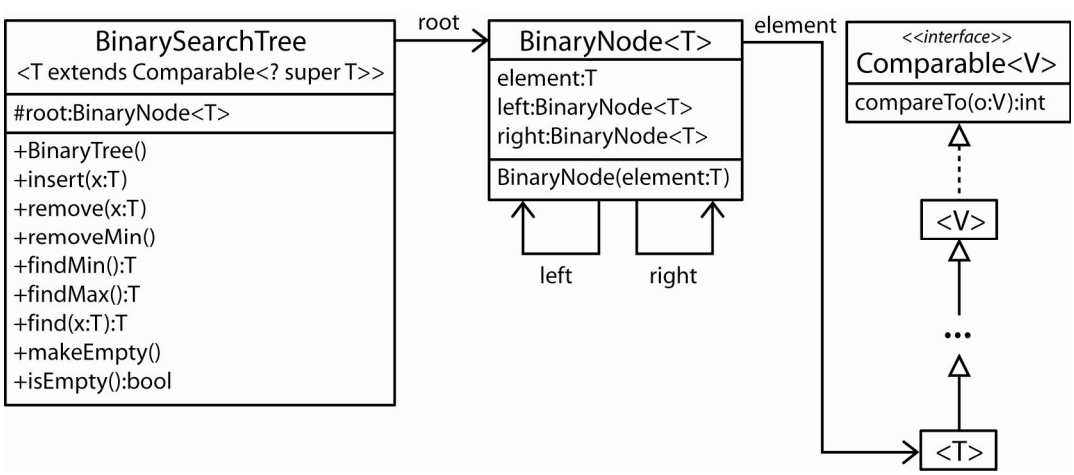
Homework 19.1

1. Build a BST by adding these nodes in this order: 4, 12, 8, 7, 16, 10, 2

19.1.2 – Implementation

1. The BST data structure that we will implement:

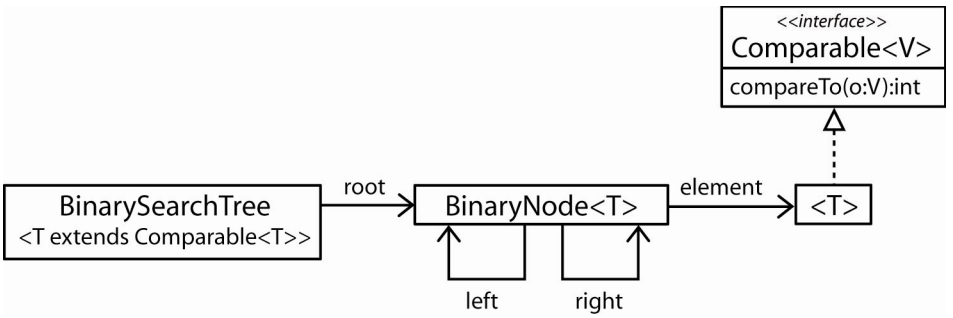
```
public class BinarySearchTree<T extends Comparable<? super T>>
```



Here is a good place to review generics. Do you see what *super* is doing? Suppose we had defined the BST class as:

```
public class BinarySearchTree<T extends Comparable<T>>
```

This would disallow the structure shown above where Comparable is implemented in a super class of T. It would require Comparable to be implemented in T as shown below:



2. The BinaryNode class is shown below. Note that all the variables are defined with package level scope.

```
1  package weiss.nonstandard;
2
3  // Basic node stored in unbalanced binary search trees
4  // Note that this class is not accessible outside
5  // this package.
6
7  class BinaryNode<AnyType>
8  {
9          // Constructor
10     BinaryNode( AnyType theElement )
11     {
12         element = theElement;
13         left = right = null;
14     }
15
16       // Data; accessible by other package routines
17     AnyType             element;  // The data in the node
18     BinaryNode<AnyType> left;     // Left child
19     BinaryNode<AnyType> right;    // Right child
20 }
```

figure 19.5

The `BinaryNode` class for the binary search tree

3. The first part of the BinarySearchTree class is shown below.

```
1  package weiss.nonstandard;
2
3  // BinarySearchTree class
4  //
5  // CONSTRUCTION: with no initializer
6  //
7  // ******************PUBLIC OPERATIONS*********************
8  // void insert( x )       --> Insert x
9  // void remove( x )       --> Remove x
10 // void removeMin( )      --> Remove minimum item
11 // Comparable find( x )   --> Return item that matches x
12 // Comparable findMin( )  --> Return smallest item
13 // Comparable findMax( )  --> Return largest item
14 // boolean isEmpty( )     --> Return true if empty; else false
15 // void makeEmpty( )      --> Remove all items
16 // ******************ERRORS********************************
17 // Exceptions are thrown by insert, remove, and removeMin if warranted
18
19 public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
20 {
21     public BinarySearchTree( )
22       {  root = null; }
23
24     public void insert( AnyType x )
25       { root = insert( x, root ); }
```

figure 19.6a

The `BinarySearchTree` class skeleton (*continues*)

4. The second part of the BinarySearchTree class is shown below. Note that the public interface methods are mirrored by private helper methods whose implementation is shown later. Also note the *elementAt* method which is used to extract the *element* from the BinaryNode.

```
26      public void remove( AnyType x )
27        { root = remove( x, root ); }
28      public void removeMin( )
29        { root = removeMin( root ); }
30      public AnyType findMin( )
31        { return elementAt( findMin( root ) ); }
32      public AnyType findMax( )
33        { return elementAt( findMax( root ) ); }
34      public AnyType find( AnyType x )
35        { return elementAt( find( x, root ) ); }
36      public void makeEmpty( )
37        { root = null; }
38      public boolean isEmpty( )
39        { return root == null; }
40
41      private AnyType elementAt( BinaryNode<AnyType> t )
42        { /* Figure 19.7 */ }
43      private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
44        { /* Figure 19.8 */ }
45      protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
46        { /* Figure 19.9 */ }
47      private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
48        { /* Figure 19.9 */ }
49      protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
50        { /* Figure 19.10 */ }
51      protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
52        { /* Figure 19.11 */ }
53      protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
54        { /* Figure 19.12 */ }
55
56      protected BinaryNode<AnyType> root;
57  }
```

figure 19.6b
The `BinarySearchTree` class skeleton (*continued*)

5. The *elementAt* method is shown below.

```
1       /**
2        * Internal method to get element field.
3        * @param t the node.
4        * @return the element field or null if t is null.
5        */
6       private AnyType elementAt( BinaryNode<AnyType> t )
7       {
8           return t == null ? null : t.element;
9       }
```

figure 19.7
The `elementAt` method

Homework 19.2

1. Draw a UML diagram of a generic BST data structure. Include the major methods.

19.1 – BST – findMin method

1. The algorithm for the *findMin* (*findMax*) method is to branch to the left (right) as far as you can. The stopping point is the minimum (maximum).

Public

```
findMin()
   return findMin( root )
```

```
findMax()
   return findMax( root )
```

Private (recursive)

```
findMin( n )
   while( n.Left != null )
      n = n.Left
   return n
```

```
findMax( n )
   while( n.Right != null )
      n = n.Right
   return n
```

2. The implementation for the protected *findMin* and private *findMax* are shown below. Note that *findMin* is protected because it will be used by the protected *remove* method. Note that either method returns a BinaryNode. The public versions use the *elementAt* method to extract the element and return it to the client.

figure 19.9
The `findMin` and `findMax` methods for binary search trees

```
1   /**
2    * Internal method to find the smallest item in a subtree.
3    * @param t the node that roots the tree.
4    * @return node containing the smallest item.
5    */
6   protected BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
7   {
8       if( t != null )
9           while( t.left != null )
10              t = t.left;
11
12      return t;
13  }
14
15  /**
16   * Internal method to find the largest item in a subtree.
17   * @param t the node that roots the tree.
18   * @return node containing the largest item.
19   */
20  private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
21  {
22      if( t != null )
23          while( t.right != null )
24              t = t.right;
25
26      return t;
27  }
```

Homework 19.3

1. Write a method to find the second smallest item and the corresponding recursive helper. Hint: there are two cases. Hint: see figure for *removeMin* method in notes below.
2. Write the *findMin* and *findMax* methods.

19.1 – BST – find method

1. The algorithm for the *find(x)* method is to continue to branch either left or right, depending on what side x is on until you get a match or don't find it.

Public

```
find( x )
   return find( x, root )
```

Private (recursive)

```
find( x, n )
   while( n != null )
      if( x < n )
         n = n.Left
      else if( x > n )
         n = n.Right
      else
         return n  // found

   return null  // not found
```

2. The implementation for the private *find* method is shown below:

```
1     /**
2      * Internal method to find an item in a subtree.
3      * @param x is item to search for.
4      * @param t the node that roots the tree.
5      * @return node containing the matched item.
6      */
7     private BinaryNode<AnyType> find( AnyType x, BinaryNode<AnyType> t )
8     {
9         while( t != null )
10        {
11            if( x.compareTo( t.element ) < 0 )
12                t = t.left;
13            else if( x.compareTo( t.element ) > 0 )
14                t = t.right;
15            else
16                return t;    // Match
17        }
18
19        return null;         // Not found
20    }
```

figure 19.8
The `find` operation for binary search trees

Homework 19.4

1. Write the *find* method.

19.1 – BST – insert(x) method

1. The algorithm for the *insert(x)* method is to continue to branch either left or right, depending on what side x is on until you get a null. The null is where the item belongs.

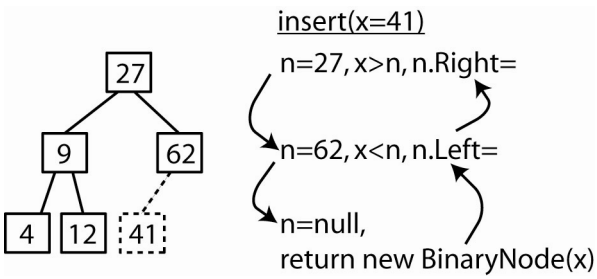
Public

```
insert( x )
   root = insert( x, root )
```

Protected (recursive)

```
insert( x, n )
    if( n == null )
       n = new BinaryNode( x )
    else if( x < n )
       n.Left = insert( x, n.Left )
    else if( x > n )
       n.Right = insert( x, n.Right )

    return n
```

Example:



2. The implementation for the protected *insert* method. Note that if the item already exists an exception is thrown.

```
1     /**
2      * Internal method to insert into a subtree.
3      * @param x the item to insert.
4      * @param t the node that roots the tree.
5      * @return the new root.
6      * @throws DuplicateItemException if x is already present.
7      */
8     protected BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
9     {
10        if( t == null )
11            t = new BinaryNode<AnyType>( x );
12        else if( x.compareTo( t.element ) < 0 )
13            t.left = insert( x, t.left );
14        else if( x.compareTo( t.element ) > 0 )
15            t.right = insert( x, t.right );
16        else
17            throw new DuplicateItemException( x.toString( ) );  // Duplicate
18        return t;
19    }
```

figure 19.10

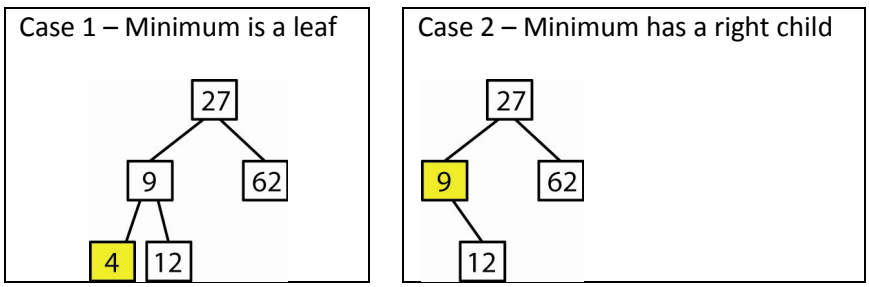
The recursive `insert` for the `BinarySearchTree` class

Homework 19.5

1. Write the *insert(x)* method.
2. Write an *insertMin(x)* method where we assume x is smaller than all the items in the tree. You are not allowed to use *compareTo* as with the general *insert* from problem 1 (or Figure 19.10). You code should be simpler than the general *insert*.

19.1 – BST – removeMin method

1. Consider the *removeMin* method. The minimum is the node furthest to the left. There are two cases.



The algorithm for *removeMin:*

Branch to the left as far as possible
Case 1: set parent's left to null
Case 2: set parent's left to the minimum's right

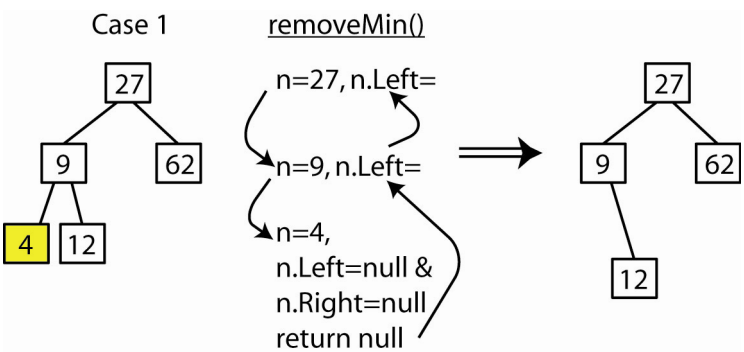
Public

```
removeMin()
    root = removeMin( root )
```

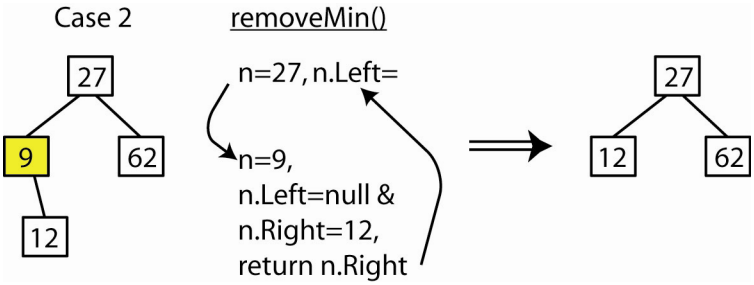
Protected (recursive)

```
removeMin( n )
    if( n.Left != null )
       n.Left = removeMin( n.Left )
       return n
    else if( n.Right == null ) // Case 1
       return null
    else                       // Case 2
       return n.Right
```

Example:



Example:



2. The implementation for the protected *removeMin* method. This method is protected because it will be used by the general *remove*. Note that the code is a little different than the algorithm above. If the left node is null, we simply return the right node. For Case 1 (a leaf), then returning the right node is returning null. For Case 2, then we are returning the non-null right node.

figure 19.11
The `removeMin` method for the `BinarySearchTree` class

```
1     /**
2      * Internal method to remove minimum item from a subtree.
3      * @param t the node that roots the tree.
4      * @return the new root.
5      * @throws ItemNotFoundException if t is empty.
6      */
7     protected BinaryNode<AnyType> removeMin( BinaryNode<AnyType> t )
8     {
9         if( t == null )
10            throw new ItemNotFoundException( );
11        else if( t.left != null )
12        {
13            t.left = removeMin( t.left );
14            return t;
15        }
16        else
17            return t.right;
18    }
```

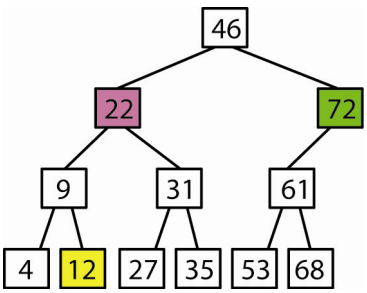
Homework 19.6

1. Write the *removeMin()* method.
2. Write the removeMax() method.
3. Write a *removeSecondSmallest()* method. Hint: I believe there are three cases: (a) min is a leaf and min has a sibling, thus second smallest is parent so remove parent and put in sibling (b) min is a leaf and parent doesn't have a right child, thus parent is second smallest so remove parent and put in min (c) min is not a leaf, thus, second smallest is right child of min, so remove that one.

19.1 – BST – remove method

1. Next, we consider the general *remove(x)* method. A general removal will create a hole in the tree. We must reattach the disconnected parts so that we preserve the BST order property. There are a number of ways we can do this, but we don't want to increase the depth of the tree because that affects run-time. So, we will add the constraint that a remove must not increase the depth of the tree. Thus, the algorithm is to branch left and right according to where x falls until the element is found. Finally, the node to delete can have:v

    1. No children – This case is simple, just remove the node.
       e.g. remove(12)
    2. 1 child – Move the left (or right) subtree in to replace removed node.
       e.g. remove(72)
    3. 2 children – Replace removed node with the minimum node in the right sub-tree.
       e.g. remove(22).

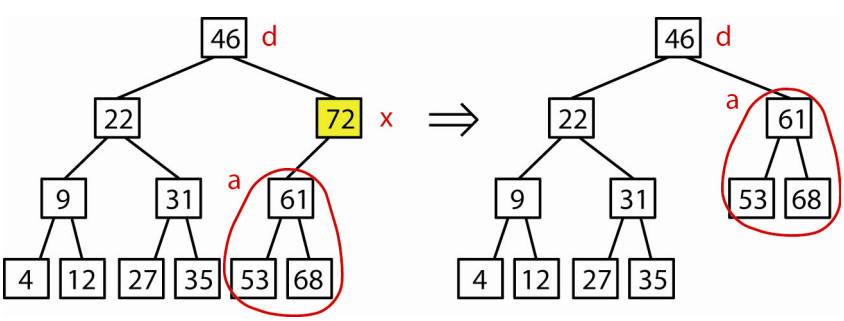


2. Case 2: 1 child

    Let:

    x = node to remove
    d = parent of x
    a = left (or right) subtree of x

    For the example shown at below, since *x* is a right (left) child of *d*, and by definition, every node in *a* is greater (less) than *d*, we can replace *x* with *a*.

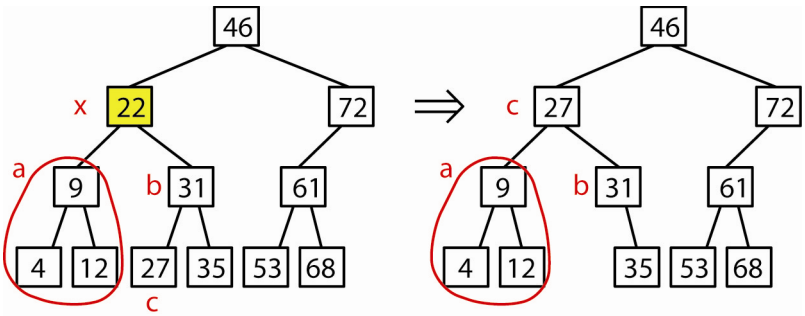


3. Case 3: 2 children

    Let:

    x = node to remove
    a = left subtree of x
    b = right child of x
    c = minimum node in right subtree of x

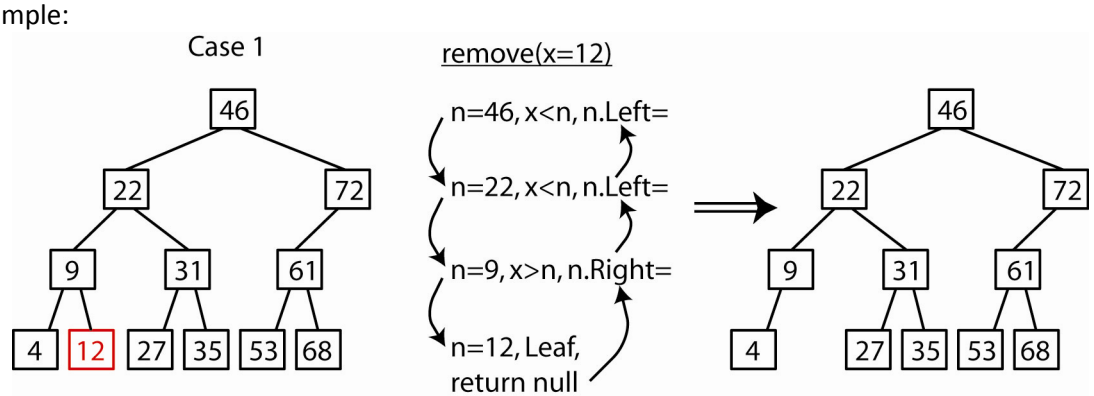
Clearly, since *c* is in the right subtree of *x*, it must be greater than any node in *a* in the left subtree. Also, since *c* is the minimum in the right subtree of *x,* it must be less than *b.* Therefore, *c* can replace *x* and have *a* as its left subtree and *b* as its right child.



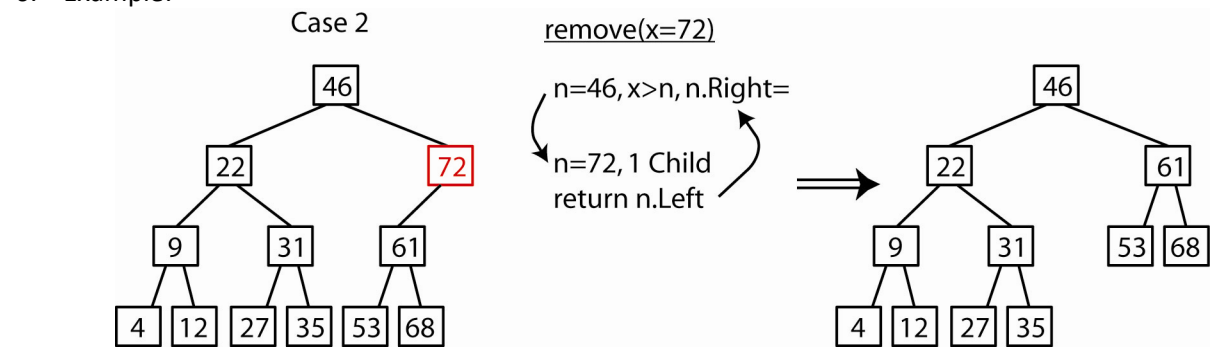
4. The algorithm for the private remove method.

```
if x < n
   n.Left = remove ( x, n.Left )              // Branch left

else if x > n
   n.Right = remove( x, n.Right )             // Branch right

else if n.Left = null and n.Right = null
   n = null                                   // Leaf

else if n.Left != null and n.Right = null
   n = n.Left                                 // Left child only

else if n.Left = null and n.Right != null
   n = n.Right                                // Right child only

else if n.Left != null and n.Right != null
   temp = findMin( n.Right )                  // Two children
   n.element = temp.element
   n.Right = removeMin( n.Right )

return n
```

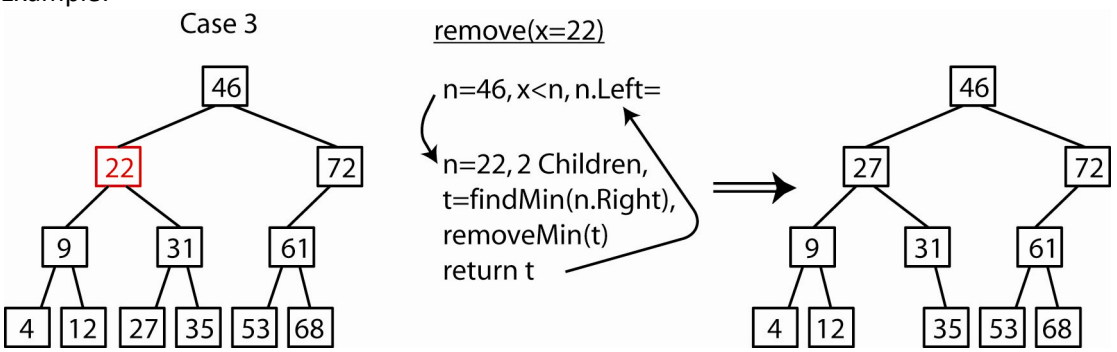
5. Example:



6. Example:



7. Example:



8. The implementation for the protected remove method is shown below. Note that the code is written more succinctly than the algorithm above.

```
1     /**
2      * Internal method to remove from a subtree.
3      * @param x the item to remove.
4      * @param t the node that roots the tree.
5      * @return the new root.
6      * @throws ItemNotFoundException if x is not found.
7      */
8     protected BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
9     {
10        if( t == null )
11            throw new ItemNotFoundException( x.toString( ) );
12        if( x.compareTo( t.element ) < 0 )
13            t.left = remove( x, t.left );
14        else if( x.compareTo( t.element ) > 0 )
15            t.right = remove( x, t.right );
16        else if( t.left != null && t.right != null ) // Two children
17        {
18            t.element = findMin( t.right ).element;
19            t.right = removeMin( t.right );
20        }
21        else
22            t = ( t.left != null ) ? t.left : t.right;
23        return t;
24    }
```

figure 19.12

The `remove` method for the `BinarySearchTree` class

Homework 19.7

1. Write the *remove()* method.

19.3 – Analysis of Binary Search Tree Operations

1. The *cost* of BST operations (insert, remove, find) is proportional to the number of nodes accessed to complete the operations. Thus, the cost of a node, n is: $c(n) = 1 + depth(n)$, remembering that depth is the number of edges to n.

2. In the best case, a BST is *perfectly balanced* (all nodes have two children, except leaves, and all leaves have the same depth). Thus, to find an element in a perfectly balanced BST takes $O(\log n)$ time in the worst case.

3. However, when a BST is not perfectly balanced, then in the worst case the structure can evolve to linked-list in which case the average and worst case complexities to find an element is O(n).

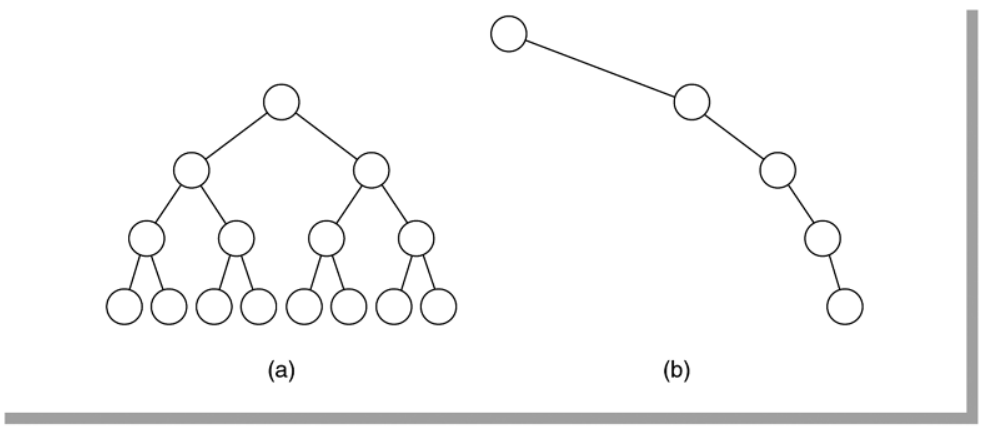


figure 19.19

(a) The balanced tree has a depth of $\lfloor \log N \rfloor$; (b) the unbalanced tree has a depth of $N - 1$.

4. On average, with random input, it can be proved that inserts and finds perform 38% worse than the perfectly balanced tree. Thus, $1.38 \log n = O(\log n)$.

5. The previous statement has not been proved for remove. Consider the remove algorithm. When there are 2 children for the node to be removed, we replace the removed node with the minimum element in the right sub-tree. It has not been proved how this affects the balance of a tree. However, empirical evidence suggests that random inserts and removes do not imbalance the tree in any measurable way.

6. Thus, it is reasonable to assume that all BST operations have an average complexity of $O(\log n)$.

7. The real problem is when the data is sorted, or is non-random in some way. For instance, consider building a BST by inserting nodes in this order: 1,2,3,4,5. The resulting tree is the same as shown in Figure 19.19b above. Thus, degeneration of the tree can result leaving $O(n)$ time for each operation.

8. The solution to this problem is to add a balance constraint to the algorithms for our basic operations (insert and remove). The balance constraint will ensure that the tree stays as close to perfect as possible. The means that worst case performance is $O(\log n)$. However, such algorithms tend to make insert and remove a bit slower (the extra work needed to ensure a balanced tree), but find is faster (because the tree is balanced). Empirical evidence shows that find is about 25% faster.