STUDENT OUTLINE

Lesson 36 – Deletion from a Binary Tree

INTRODUCTION: Because a node in a binary tree can have one or two descendants, deleting such a location can cause a lot of problems. After examining the code to delete from a binary tree, you will then solve a mirror image variation of this routine in your lab exercise. The use of diagrams to follow this algorithm is very important.

The key topic for this lesson is:

A. Deletion from a Binary Tree
B. `deleteTargetNode` Method

DISCUSSION:

A. Deletion from a Binary Tree

1. Deleting a node involves two steps:
   a. Locating the node to be deleted.
   b. Eliminate that node from the data structure.

2. After locating the node to be deleted, we must determine the nature of that node.

   a. If it is a leaf, make the parent node point to **`null`**.
   b. If it has one child on the right, make the parent node point to the right child.
   c. If it has one child on the left, make the parent node point to the left child.
   d. If it has two children, the problem becomes much harder to solve.

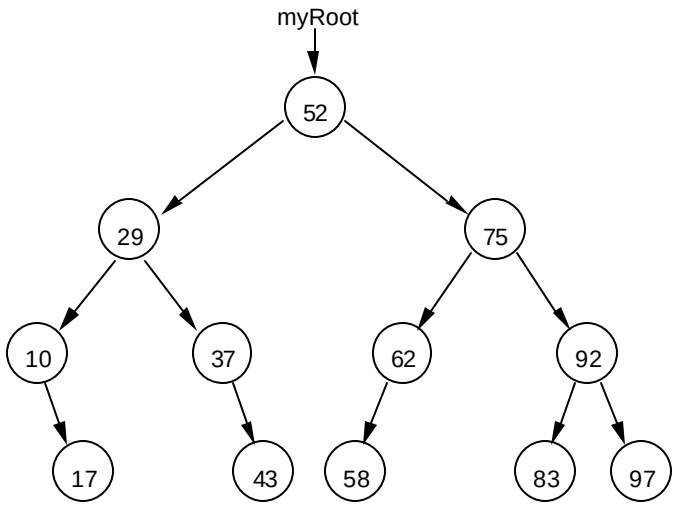

Diagram 36-1

3. A leaf node containing the value 43 will be easy to delete. The parent node of the node containing 43 will change its right pointer to **`null`**.

4. A node with one child on the right, like the node with value 10, will involve rerouting the node from its parent to its single right child.

5. But rerouting around the node with value 29, a node with two children, involves breaking off subtrees and reattaching them at the proper location.

See Handout H.A.36.1, *Deletion from a Binary Tree.*

6. The code to implement deletion from a binary tree is given in Handout, H.A.36.1, *Deletion from a Binary Tree*. The recursive `deleteHelper` method that locates the node to be deleted is given below:

```
public void delete(Comparable target)
{
    myRoot = deleteHelper(myRoot, target);
}

private TreeNode deleteHelper(TreeNode node, Comparable target)
{
  if (node == null)
  {
    throw new NoSuchElementException();
  }
  else if (target.equals(node.getValue()))
  {
     return deleteTargetNode(node);
  }
  else if (target.compareTo(node.getValue()) < 0)
  {
    node.setLeft(deleteHelper(node.getLeft(), target));
    return node;
  }
  else //target.compareTo(root.getValue()) > 0
  {
    node.setRight(deleteHelper(node.getRight(), target));
    return node;
  }
}
```

7. After the value to be deleted is input in `testDelete`, this method calls the `delete` method of the `BinarySearchTree` object the first time and passes a reference to the `Item` object containing the id value to be deleted from the tree. The `delete` method then passes the root of the tree (`myRoot`) and the target item to be located to the `deleteHelper` method. The `deleteHelper` method receives a `TreeNode` reference alias (`node`). The `deleteHelper` method has 4 scenarios:

   a. `node == null`, the value does not exist in the tree, throw a `NoSuchElementException`.

b. We found the correct node (`target.equals(node.getValue())`), call `deleteTargetNode` and pass it `node`.
c. Did not find the node yet, recursively call `deleteHelper` and pass it the internal reference to the left child.
d. Recursively call `deleteHelper` and pass it the internal reference to the right child.

B. `deleteTargetNode` Method

1. The `deleteHelper` method finds the node to be deleted and calls `removeTargetNode`, passing a reference to the `TreeNode` `target` as shown in the following method.

```
private TreeNode deleteTargetNode(TreeNode target)
{
  if (target.getRight() == null)
  {
    return target.getLeft();
  }
  else if (target.getLeft() == null)
  {
    return target.getRight();
  }
  else if (target.getLeft().getRight() == null)
  {
     target.setValue(target.getLeft().getValue());
     target.setLeft(target.getLeft().getLeft());
     return target;
  }
  else // left child has right child
  {
     TreeNode marker = target.getLeft();
     while (marker.getRight().getRight() != null)
       marker = marker.getRight();
     target.setValue(marker.getRight().getValue());
     marker.setRight(marker.getRight().getLeft());
     return target;
  }
}
```

2. The algorithm for deletion employed in the `deleteTargetNode` method is:

   a. Node to be deleted is a leaf. Make the link from the parent **`null`**.

   b. Node to be deleted has no left (or right) subtree (one child). Make the link from the parent refer to the left (or right) subtree.

   c. Node to be deleted has non-empty left and right subtrees (two children). Change the node value to the largest value in the left subtree, and then delete the largest value from the left subtree. (The deletion of the largest value must be either case a or b above.)

3. The leaf and one child cases are handled in `deleteTargetNode` as follows:

```
...
if (target.getRight() == null)
{
  return target.getLeft();
}
else if (target.getLeft() == null)
{
  return target.getRight();
}
...
```

These cases are left for you and your instructor to trace.

4. The two-child case is more difficult and involves changing the node value to the largest value in the left subtree, then deleting the largest value from the left subtree. The rightmost node will be the node with the greatest value in the left subtree.

5. Working with a smaller version of the same binary tree in Diagram 36-1. Suppose in the following diagram we wish to delete the node with value 75

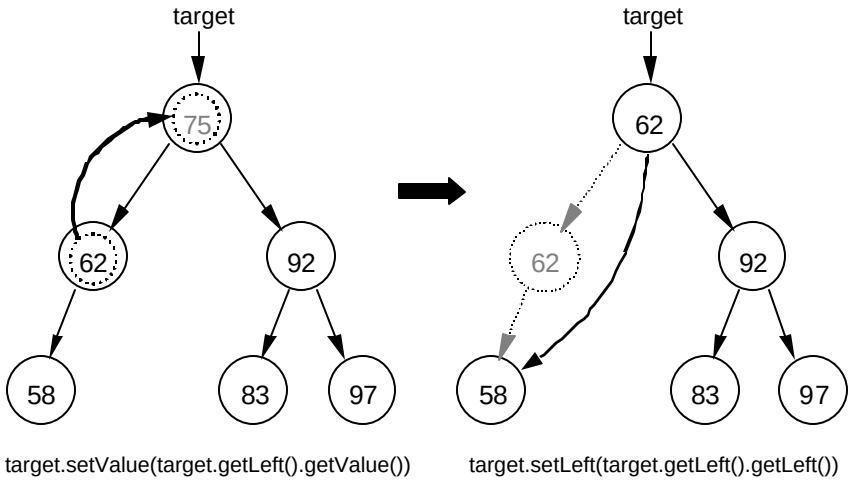

Diagram 36-2

6. Working with a smaller version of the same binary tree in Diagram 36-1. Here are the steps for deleting a node having two children in which the left child has no right.

   a. Copy the contents of the left child of `target` and set it as the current value.

      ```
      target.setValue(target.getLeft().getValue());
      ```

      As show in the diagram above, the value 75 is replace with 62.

   b. Reattach the left subtree to maintain an ordered tree. The left subtree of the node reference by `target` will now point to the node containing the value 58.

      ```
      target.setLeft(target.getLeft().getLeft());
      ```

      As show in the Diagram 36-2 above, since the node that originally contained the value 62 is no longer referenced, it is removed (garbage collected).

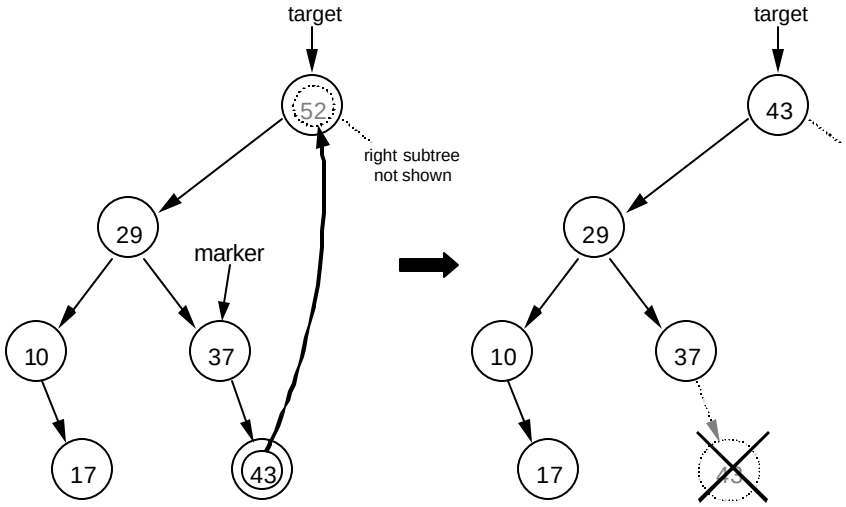

Diagram 36-3

7. Working with the right subtree of our original binary tree (see Diagram 36-1). Here are the steps for deleting a node containing the value 52. In this case the node has two children and the left child has a right child.

   a. Position `marker` to access the node with the largest value in the left subtree. This is the rightmost node in the left subtree.

   ```
   TreeNode marker = target.getLeft();
   while (marker.getRight().getRight() != null)
     marker = marker.getRight();
   ```

   As show in the diagram above, `marker` now references the node pointing to the node with largest value in the left subtree (43).

   b. Copy the contents of the right child of `marker` and set it as the current value.

   ```
   target.setValue(marker.getRight().getValue());
   ```

   As show in the diagram above, the value 52 is replaced with 43.

   c. Delete the largest value from the right subtree. Reattach the right subtree to maintain an ordered tree.

   ```
   marker.setRight(marker.getRight().getLeft());
   ```

   As show in the Diagram 36-3 above, the node containing the value 43 is no longer referenced.

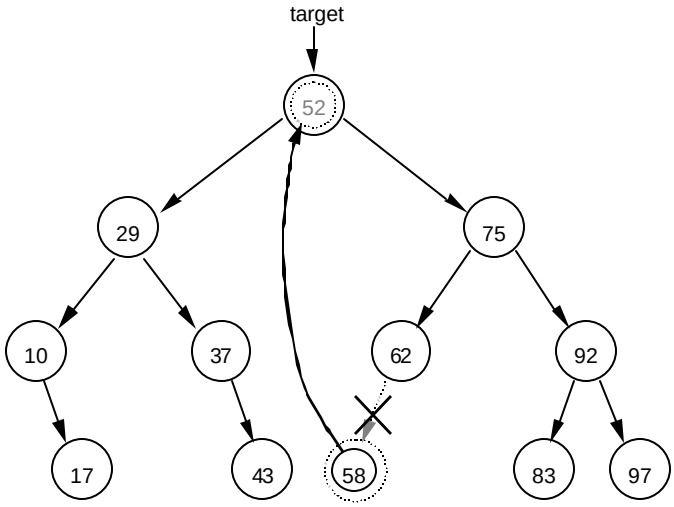

Diagram 36-4

8. This entire process for the two-child case could be directed the other way. Again, suppose the node with value 52 is to be deleted from the original tree. Referring to Diagram 36-4 above, the steps would be:

   a. Access the node with the smallest value in the right subtree. This is the leftmost node in the right subtree.

   b. Copy the contents (58) and set it as the current value.

   c. Delete the smallest value from the left subtree. Reattach the left subtree to maintain an ordered tree.

SUMMARY/ REVIEW: The alternative direction of section B.8. above will be the basis of your lab exercise. There are other deletion algorithms, but this one is efficient and clear. Take some time to review the steps for yourself using the diagrams and code side-by-side.

ASSIGNMENT: Lab Exercise L.A.36.1, *BSTree (Part 3)*

LAB EXERCISE

BSTree (Part 3)

Assignment:

1. Copy the methods presented in Handout H.A.36.1, *Deletion from a Binary Tree*. However, you are required to solve the two-child case as a mirror image of the solution described in Section B.7. of the student outline O.A.36.1. Change the `deleteTargetNode` method to deal with the two-child case as follows:

   a. Position `marker` to access the node with the smallest value in the right subtree. This is the leftmost node in the right subtree.

   b. Copy the contents of the left child of `marker` and set it as the current value.

   c. Delete the smallest value from the left subtree. Reattach the left subtree to maintain an ordered tree.

2. Test your code and solve the following sequence of run output steps:

   a. Load the file from disk (*file20.txt*).
   b. Print the tree.
   c. Print the number of nodes in the tree.
   d. Search for Id values specified by your instructor. Print out the Id and Inv response in column form.
   e. Delete the Id values specified by your instructor.
   f. Print the tree again.
   g. Print the number of nodes in the tree.

Instructions:

1. Turn in your source code for the entire program and the run output described above.

DELETION FROM A BINARY TREE

```
public void delete(Comparable target)
// post: deletes a node with data equal to target, if present,
//       preserving binary search tree property
{
  myRoot = deleteHelper(myRoot, target);
}

private TreeNode deleteHelper(TreeNode node, Comparable target)
// pre : node points to a non-empty binary search tree
// post: deletes a node with data equal to target, if present,
//       preserving binary search tree property
{
  if (node == null)
    throw new NoSuchElementException();

  else if (target.compareTo(node.getValue()) == 0)
  {
     return deleteTargetNode(node);
  }
  else if (target.compareTo(node.getValue()) < 0)
  {
    node.setLeft(deleteHelper(node.getLeft(), target));
    return node;
  }
  else //target.compareTo(root.getValue()) > 0
  {
    node.setRight(deleteHelper(node.getRight(), target));
    return node;
  }
}

private TreeNode deleteTargetNode(TreeNode target)
// pre : target points to node to be deleted
// post: target node is deleted preserving binary search tree property
{
  if (target.getRight() == null)
  {
    return target.getLeft();
  }
  else if (target.getLeft() == null)
  {
    return target.getRight();
  }
  else if (target.getLeft().getRight() == null)
  {
     target.setValue(target.getLeft().getValue());
     target.setLeft(target.getLeft().getLeft());
     return target;
  }
  else // left child has right child
  {
     TreeNode marker = target.getLeft();
```

```
      while (marker.getRight().getRight() != null)
        marker = marker.getRight();

      target.setValue(marker.getRight().getValue());
      marker.setRight(marker.getRight().getLeft());
      return target;
    }
}

// ------------------ testDelete method – add to BSTree.java

public void testDelete(BinarySearchTree temp)
{
  int idToDelete;
  boolean success;

  System.out.println("Testing delete algorithm\n");
  System.out.print("Enter Id value to delete (-1 to quit) --> ");
  idToDelete = console.readInt();

  while (idToDelete >= 0)
  {
    Item dNode = new Item(idToDelete, 0);

    if (temp.find(dNode) == null)
      System.out.println("Id# " + idToDelete + "  No such part in stock");
    else
    {
      temp.delete(dNode);
      System.out.println("     Id #" + idToDelete + " was deleted");
    }
    System.out.println();
    System.out.print("Enter Id value to delete (-1 to quit) --> ");

    idToDelete = console.readInt();
  }
}
```