Binary Trees – Part I

This lesson is borrowed from the following:

Reference

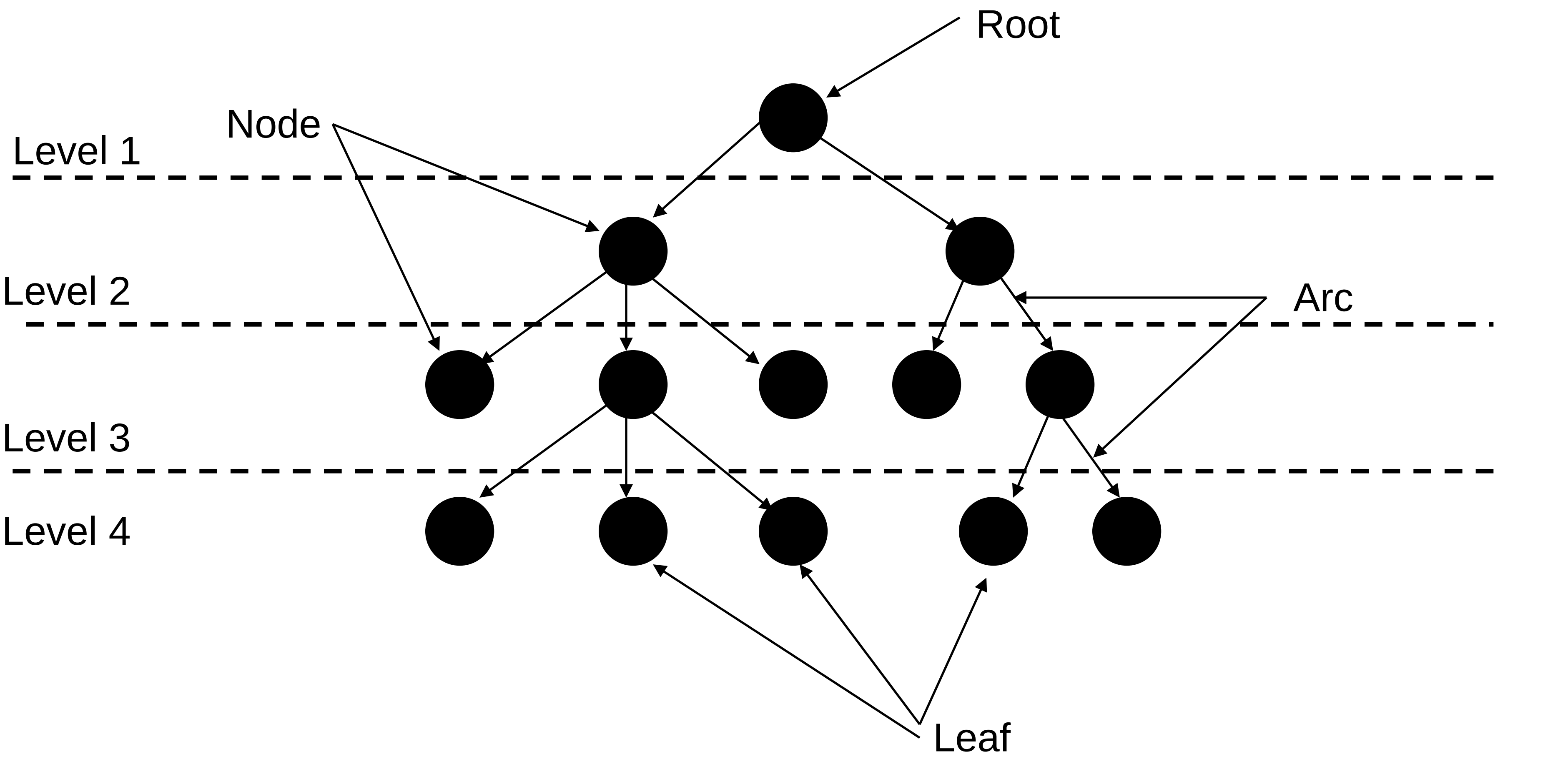
CS 367 – Introduction to Data Structures

http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Trees-I.ppt

Trees

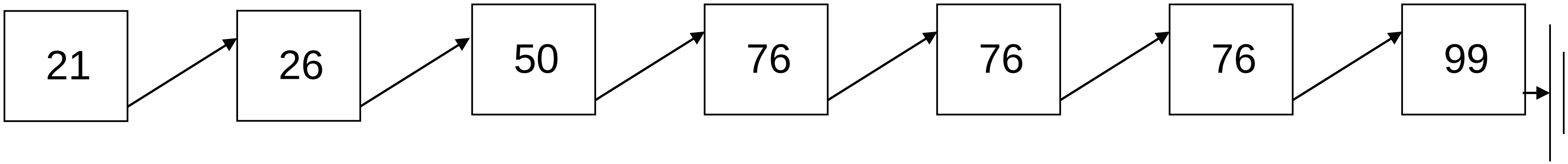
- Nodes
  - element that contains data
- Root
  - node with children and no parent
- Leaves
  - node with a parent and no children
- Arcs
  - connection between a parent and a child
- Depth
  - number of levels in a tree
- A node can have 1 to n children – no limit
- Each node can have only one parent

Tree of Depth 4



Trees vs. Linked Lists

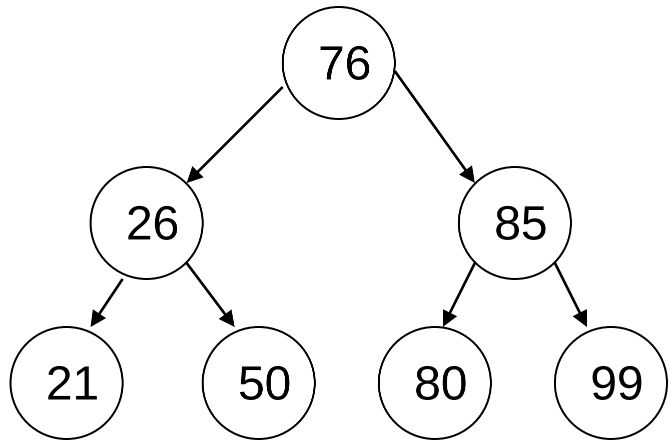
- Imagine the following linked list



- How many nodes must be touched to retrieve the number 99?
  - answer: 7
  - must touch every item that appears in the list before the one wanted can be retrieved
  - random insert, delete, or retrieve is O(n)

Trees vs. Linked Lists

- Now consider the following tree



- How many nodes must now be touched to retrieve the number 99?
  - answer: 3
  - random insert, delete, or retrieve is O(log n)

Trees vs. Linked Lists

- Similarities
  - both can grow to an unlimited size
  - both require the access of a previous node before the next one
- Difference
  - access to a previous node can give access to multiple next nodes
    - if we’re smart (and we will be) we can use this fact to drastically reduce search times

Trees vs. Arrays

- Consider the following array

21	26	50	76	76	76	99

- How many nodes must be touched to retrieve the number 99?
  - answer: 3
    - remember the binary search of a sorted array?
  - searching a sorted array is O(log n)
  - how about inserting or deleting from an array
    - O(n)

Trees vs. Arrays

- Similarities
  - searching the list takes the same time
- Differences
  - inserting or deleting from an array is more time consuming
  - an array is fixed in size

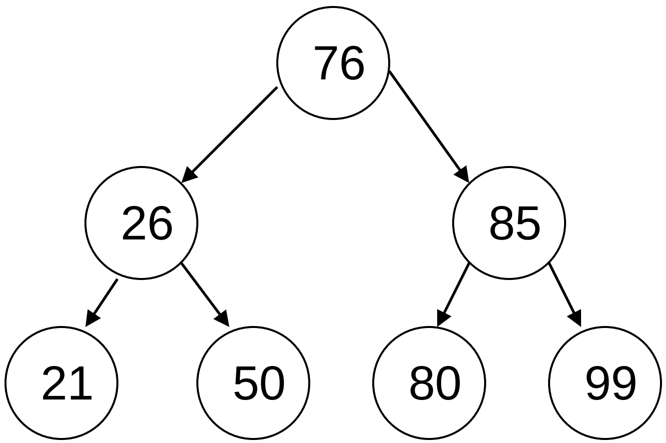
Tree Operations

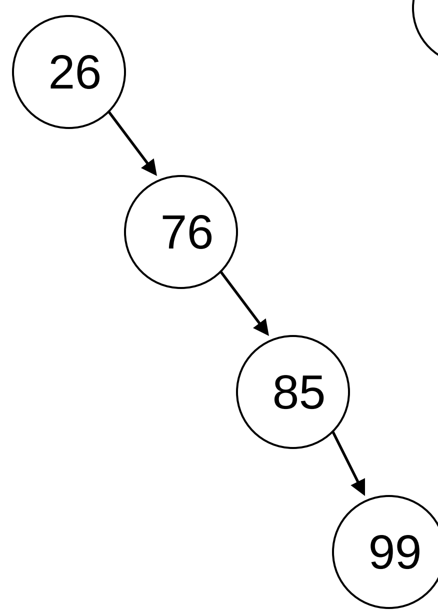
- Sorting
  - way to guarantee the placement of one node with respect to other nodes
- Searching
  - finding a node based on a *key*
- Inserting
  - adding a node in a sorted order to the tree
- Deleting
  - removing a node from the tree in such a way that the tree is still sorted

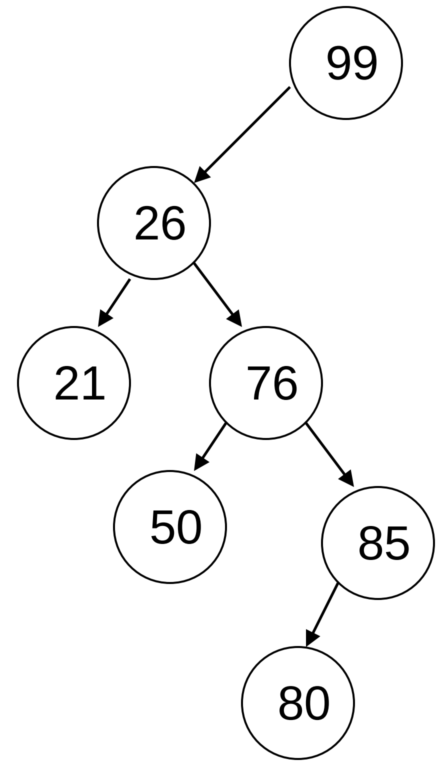
Binary Tree

- One specific type of tree is called a *binary tree*
  - each node has at most 2 children
    - right child and left child
  - sorting the tree is based on the following criteria
    - every item rooted at node N's right child is greater than N
    - every item rooted at node N's left child is less than N

Binary Trees







Implementing Binary Trees

- Each node in the tree must contain a reference to the data, key, right child, and left child

```
class TreeNode {
   public Object key;
   public Object data;
   public TreeNode right;
   public TreeNode left;
   public TreeNode(Object k, Object data) { key=k; data=d; }
}
```

- Tree class only needs reference to root of tree
  - as well as methods for operating on the tree

Implementing Binary Trees

```
class BinaryTreeRec {
    private TreeNode root;
    public BinaryTree() { root = null; }
    public Object search(Object key) { search(key, root); }
    private Object search(Object key, TreeNode node);
    public void insert(Object key, Object data) { insert(key, data, root); }
    private void insert(Object key, Object data, TreeNode node);
    public Object delete(Object key) { delete(key, root, null); }
    private Object delete(Object key, TreeNode cur, TreeNode prev);
}
```

Searching a Binary Tree

- Set reference P equal to the root node
  - if P is equal to the key
    - found it
  - if P is less than key
    - set P equal to the right child node and repeat
  - if P is greater than key
    - set P equal to the left child node and repeat

Recursive Binary Tree Search

```
private Object search(Object key, TreeNode node) {
    if(node == null) { return null; }

    int result = node.key.compareTo(key);
    if(result == 0)
        return node.data;   // found it
    else if(result < 0)
        return search(key, node.right);   // key in right subtree
    else
        return search(key, node.left);   // key in left subtree
}
```

Inserting into Binary Tree

- Set reference P equal to the root node
  - if P is equal to null
    - insert the node at position P
  - if P is less than node to insert
    - set P equal to the right child node and repeat
  - if P is greater than node to insert
    - set P equal to the left child node and repeat

Recursive Binary Tree Insert

```
private void insert(Object key, Object data, TreeNode node) {
    if(node == null) {
        root = new TreeNode(key, data);
        return;
    }

    int result = node.key.compareTo(key);
    if(result < 0) {
        if(node.right == null)
            node.right = new TreeNode(key, data);
        else
            insert(key, data, node.right);
    }
    else {
        if(node.left == null)
            node.left = new TreeNode(key, data);
        else
            insert(key, data, node.left);
    }
}
```

Binary Tree Insert

- One important note from this insert
  - the key cannot already exists
    - if it does, an error should be returned (our code did not do this)
  - all keys in a tree must be unique
    - have to have some way to differentiate different nodes

Binary Tree Delete

- Two possible methods
  - delete by merging
    - discussed in the book
    - problem is that it can lead to a very unbalanced tree
  - delete by copying
    - this is the method we will use
    - can still lead to an unbalanced tree, but not nearly as severe

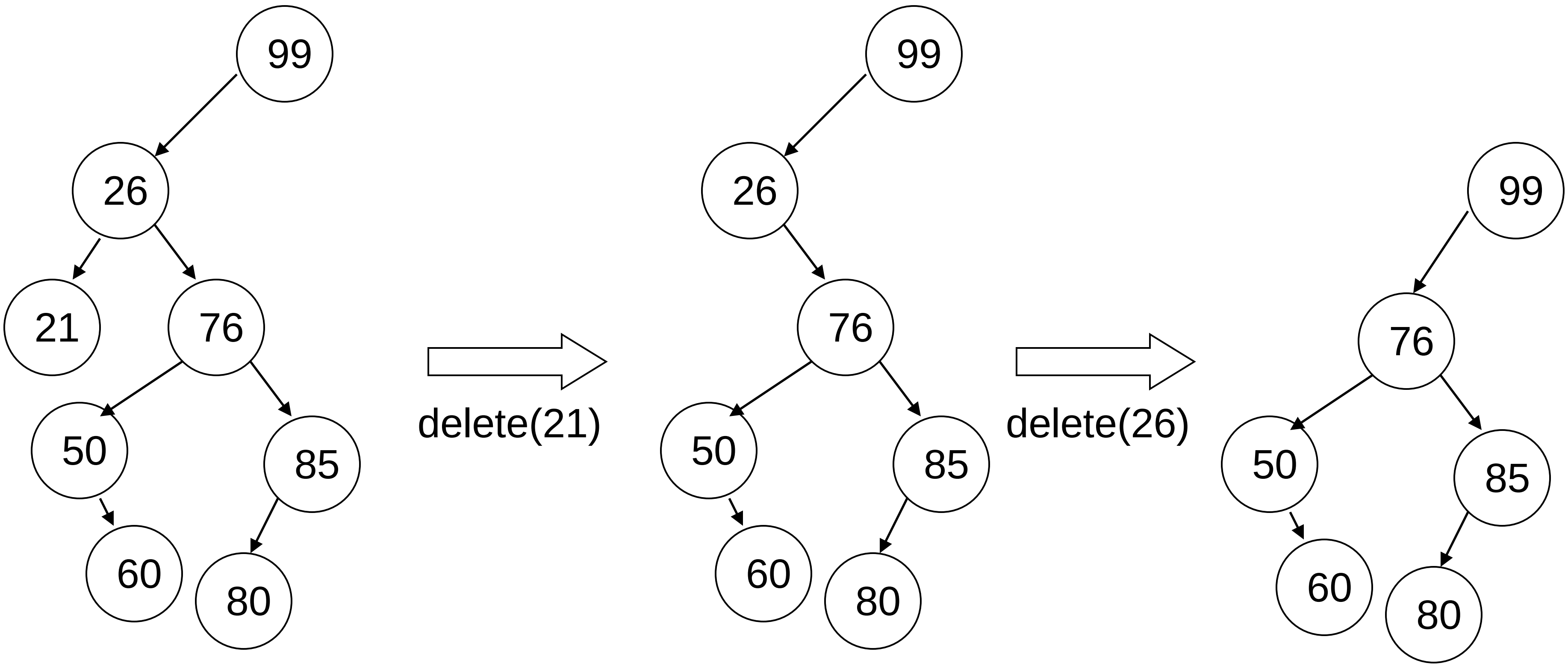
Binary Tree Delete

- Set reference P equal to the root node
  - search the tree to find the desired node
    - if it doesn’t exist, return null
  - once node is found,
    - replace it
    - this is going to require some more explanation

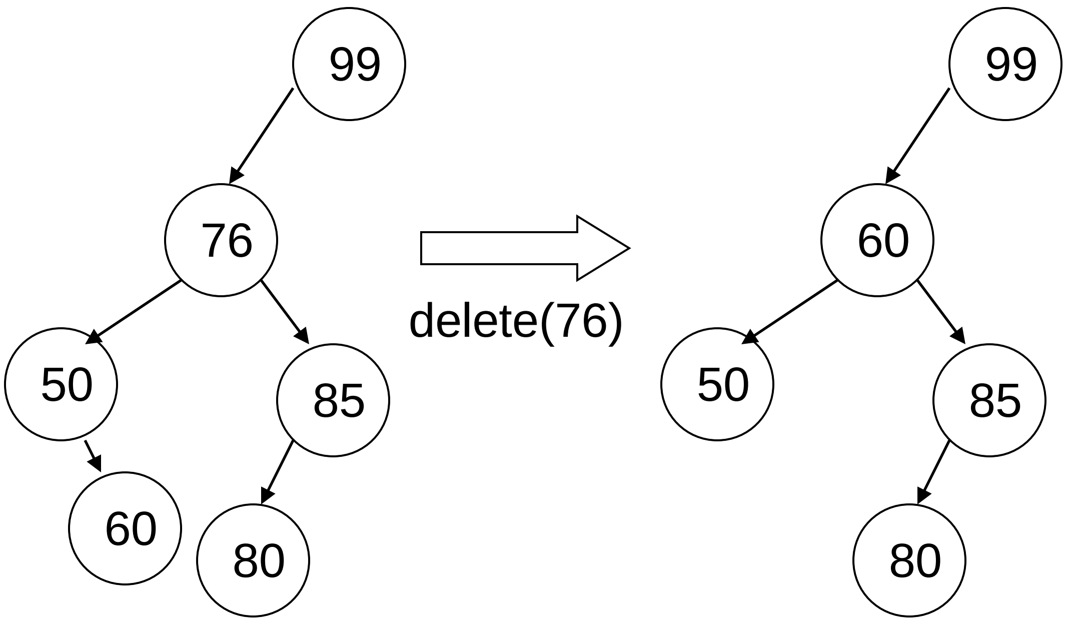
Binary Tree Delete

- Three possible cases for node to delete
  - it is a leaf
    - simple, make it's parent point to null
  - it has only a single child
    - simple, make it's parent point to the child
  - it has two children
    - need to find one of it's descendants to replace it
      - we will pick the largest node in the left subtree
      - could also pick the smallest node in the right subtree
    - this is a fairly complex operation

Simple Cases



Complex Case



Complex Case

- Notice that we must first find the largest value in the left subtree
  - keep moving to the next right child until the right.next pointer is equal to null
    - this is the largest node in a subtree
  - then make this child's parent point to this child's left pointer
  - move this child into the same spot as the deleted node
    - requires the manipulation of a few pointers

Removing a Node

```
public void remove(TreeNode node, TreeNode prev) {
    TreeNode tmp, p;
    if(node.right == null)
      tmp = node.left;
    else if(node.left == null)
      tmp = node.right;
    else {
      tmp = node.left;   p = node;
      while(tmp.right != null) {
            p = tmp;
            tmp = tmp.right;
      }
      if(p == node) { p.left = tmp.left; }
      else { p.right = tmp.left; }
      tmp.right = node.right;
      tmp.left = node.left;
    }
    if(prev == null) { root = tmp; }
    else if(prev.left == node) { prev.left = tmp; }
    else { prev.right = tmp; }
}
```

Recursive Binary Tree Delete

```
private Object delete(Ojbect key, TreeNode cur, TreeNode prev) {
    if(cur == null)
        return null;   // key not in the tree

    int result = cur.key.compareTo(key);
    if(result == 0) {
        remove(cur, prev);
        return cur.data;
    }
    else if(result < 0)
        return delete(key, cur.right, cur);
    else
        return delete(key, cur.left, cur);
}
```

Iterative Solution

- Most operations shown so far can easily be converted into an iterative solution

```
class BinaryTreeIter {
    private TreeNode root;
    public BinaryTree() { root = null; }
    public Object search(Object key);
    public void insert(Object key, Object data);
    public Object delete(Object key);
}
```

Iterative Binary Search

```
public Object search(Object key) {
    TreeNode node = root;
    while(node != null) {
        int result = node.key.compareTo(key);
        if(result == 0)
            return node.data;
        else if(result < 0)
            node = node.right;
        else
            node = node.left;
    }
    return null;
}
```

Iterative Binary Tree Insert

```
public void insert(Object key, Object data) {
    TreeNode cur = root;
    TreeNode prev = null;
    while(cur != null) {
      if(cur.key.compareTo(key) < 0) { prev = cur;   cur = cur.right; }
      else { prev = cur;   cur = cur.left; }
    }

    if(prev == null) { root = new TreeNode(key, data); }
    else if(prev.key.compareTo(key) < 0)
      prev.right = new TreeNode(key, data);
    else
      prev.left = new TreeNode(key, data);
}
```

Iterative Binary Tree Delete

```
public Object delete(Object key) {
    TreeNode cur = root;
    TreeNode prev = null;
    while((cur != null) && (cur.key.compareTo(key) != 0)) {
      prev = cur;
      if(cur.key.compareTo(key) < 0) { cur = cur.right; }
      else { cur = cur.left; }
    }
    if(cur != null) {
      replace(cur, prev);
      return cur.data;
    }
    return null;
}
```