

Announcements

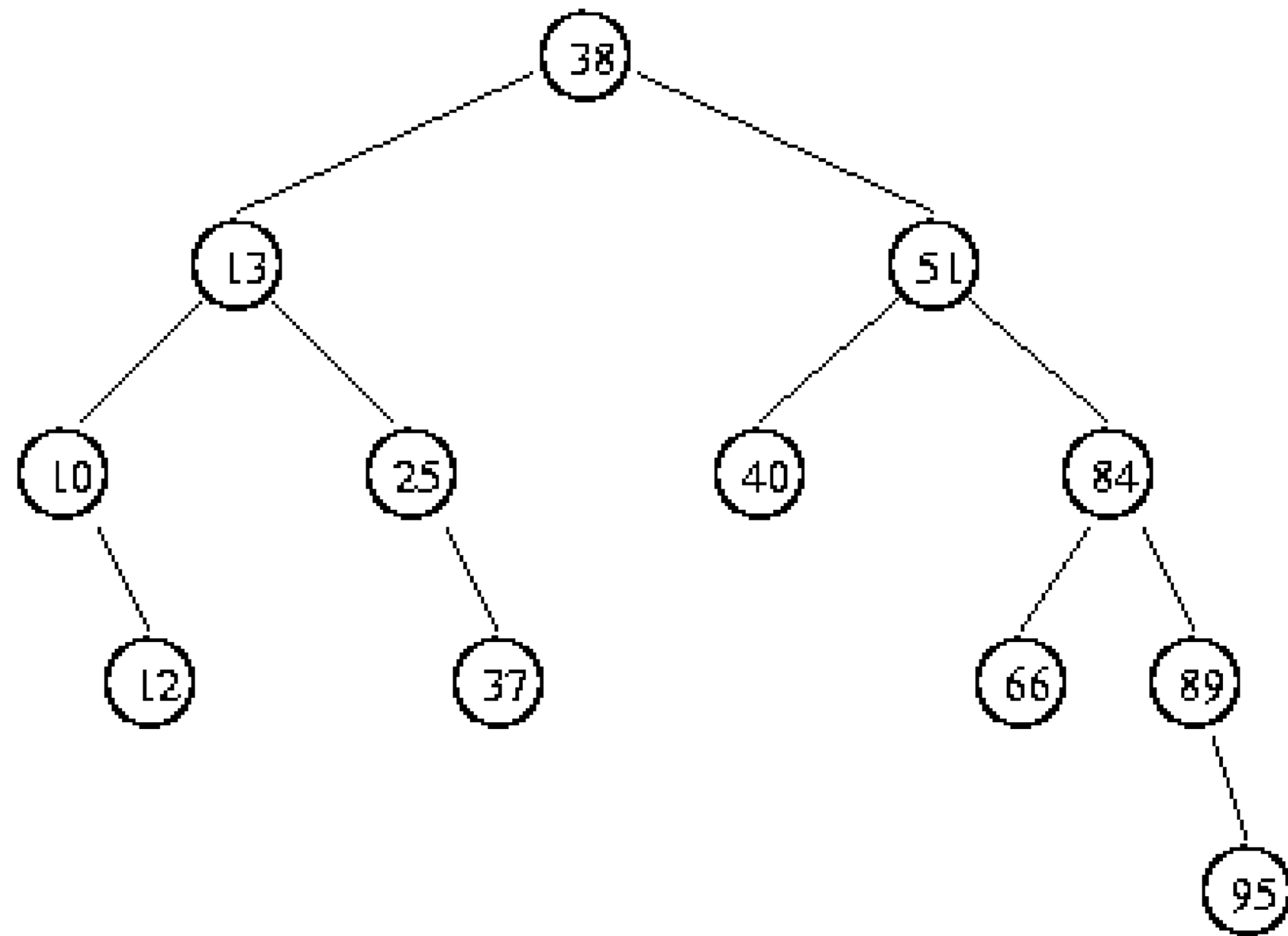
MP5 available, due 10/31, 11:59p. EC due 10/24, 11:59p.

TODAY: BST removal

<http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html>

```
void Dictionary<K,D>::insert(treeNode * & cRoot, const K & key, const D & data){  
    if (cRoot == NULL)  
        cRoot = new treeNode(key,data);  
    else if (key < cRoot->key)  
        insert(cRoot->left, key, data);  
    else if (key > cRoot->key)  
        insert(cRoot->right, key, data);  
}
```


Dictionary ADT: (BST implementation)



insert

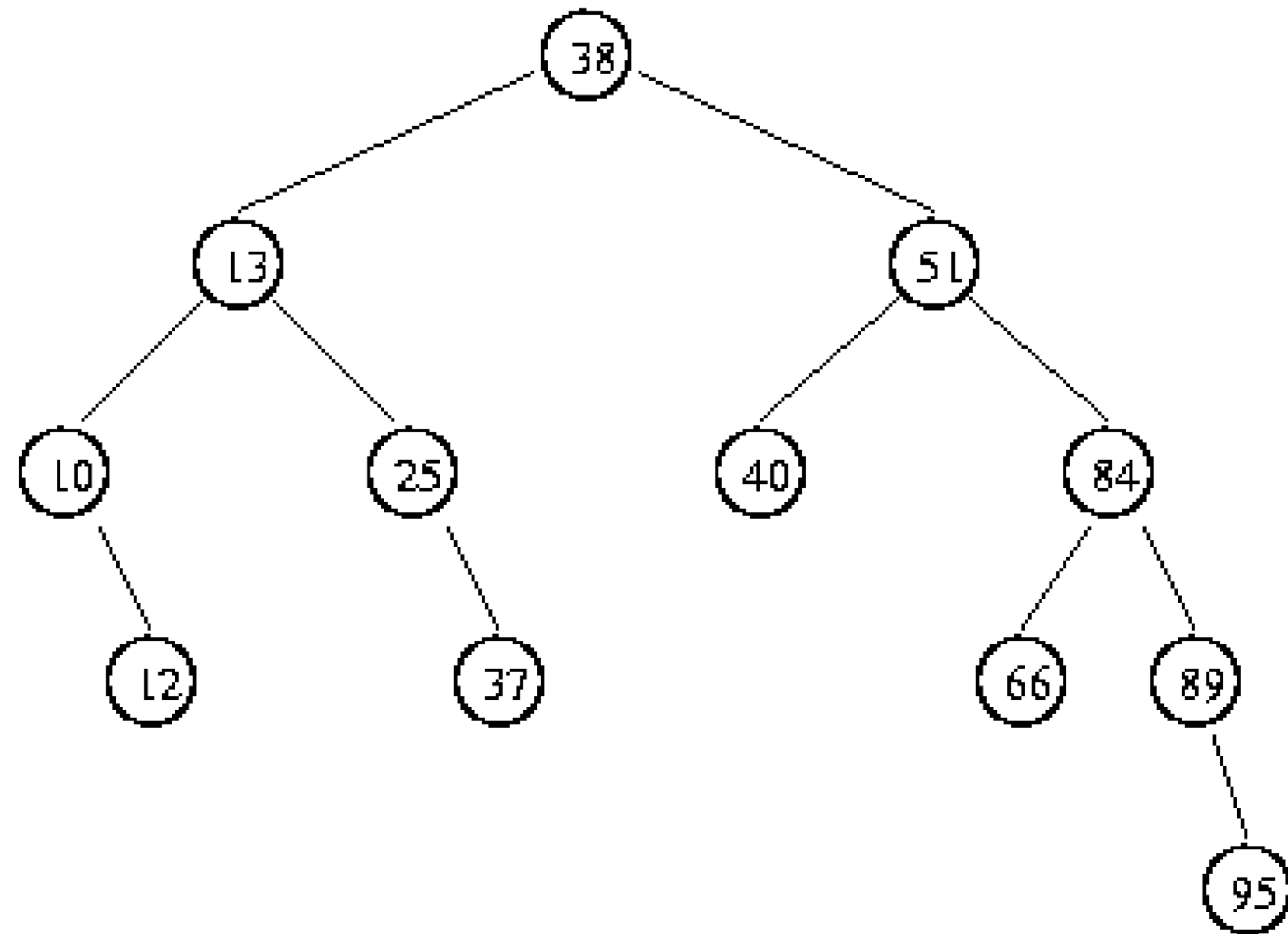
remove

find

traverse

```
template <class K, class D>
class Dictionary{
public:
    // constructor for empty tree.
private:
    struct treeNode{
        D data;
        K key;
        treeNode * left;
        treeNode * right;
    };
    treeNode * root
};
```


Binary Search Tree - Remove

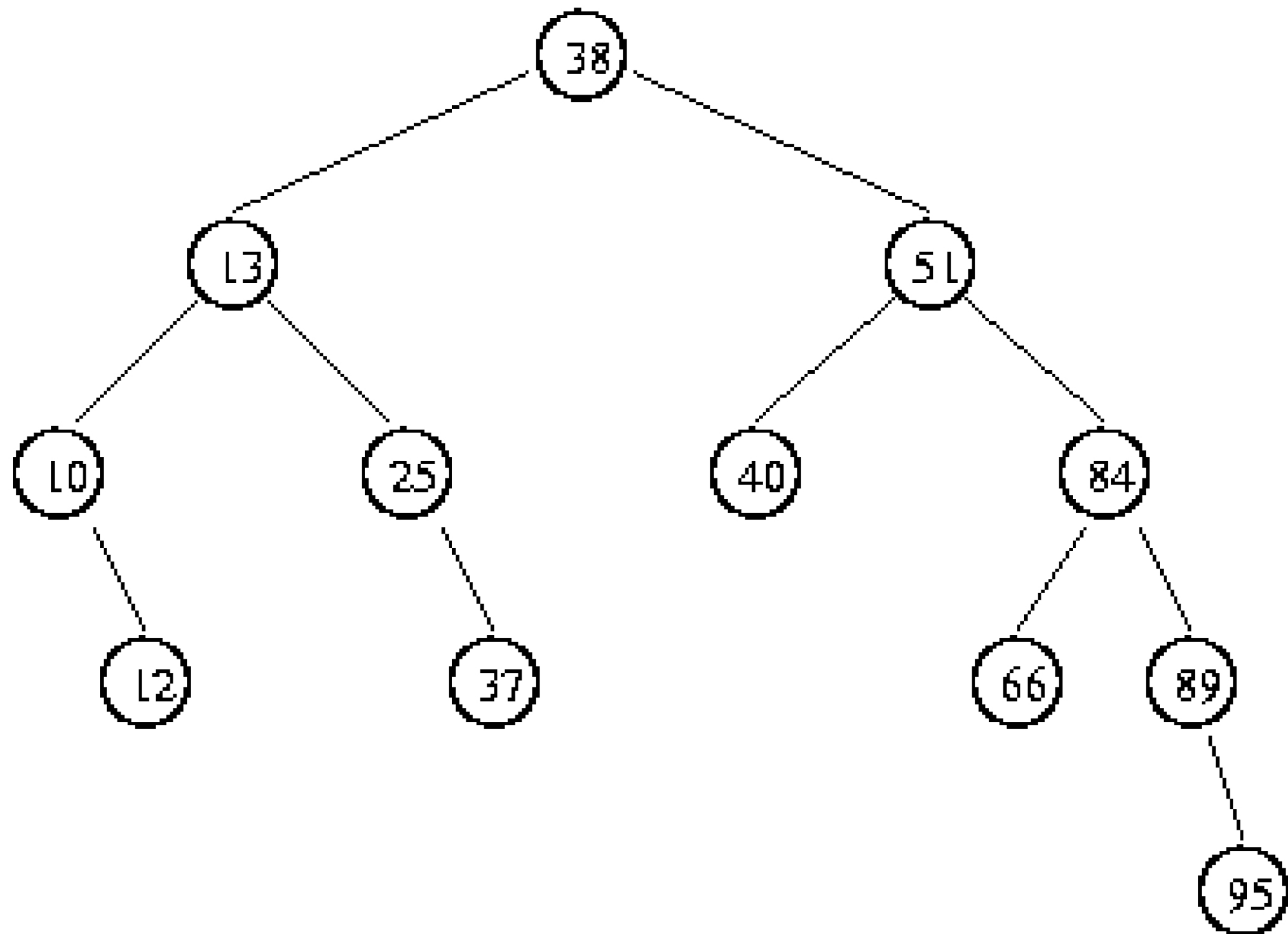


```
void BST<K>::remove(treeNode * & cRoot, const K & k) {  
    if (cRoot != NULL) {  
        if (cRoot->key == k)  
            doRemoval(cRoot);  
        else if (k < cRoot->key)  
            remove(cRoot->left, k);  
        else  
            remove(cRoot->right, k);  
    }  
}
```

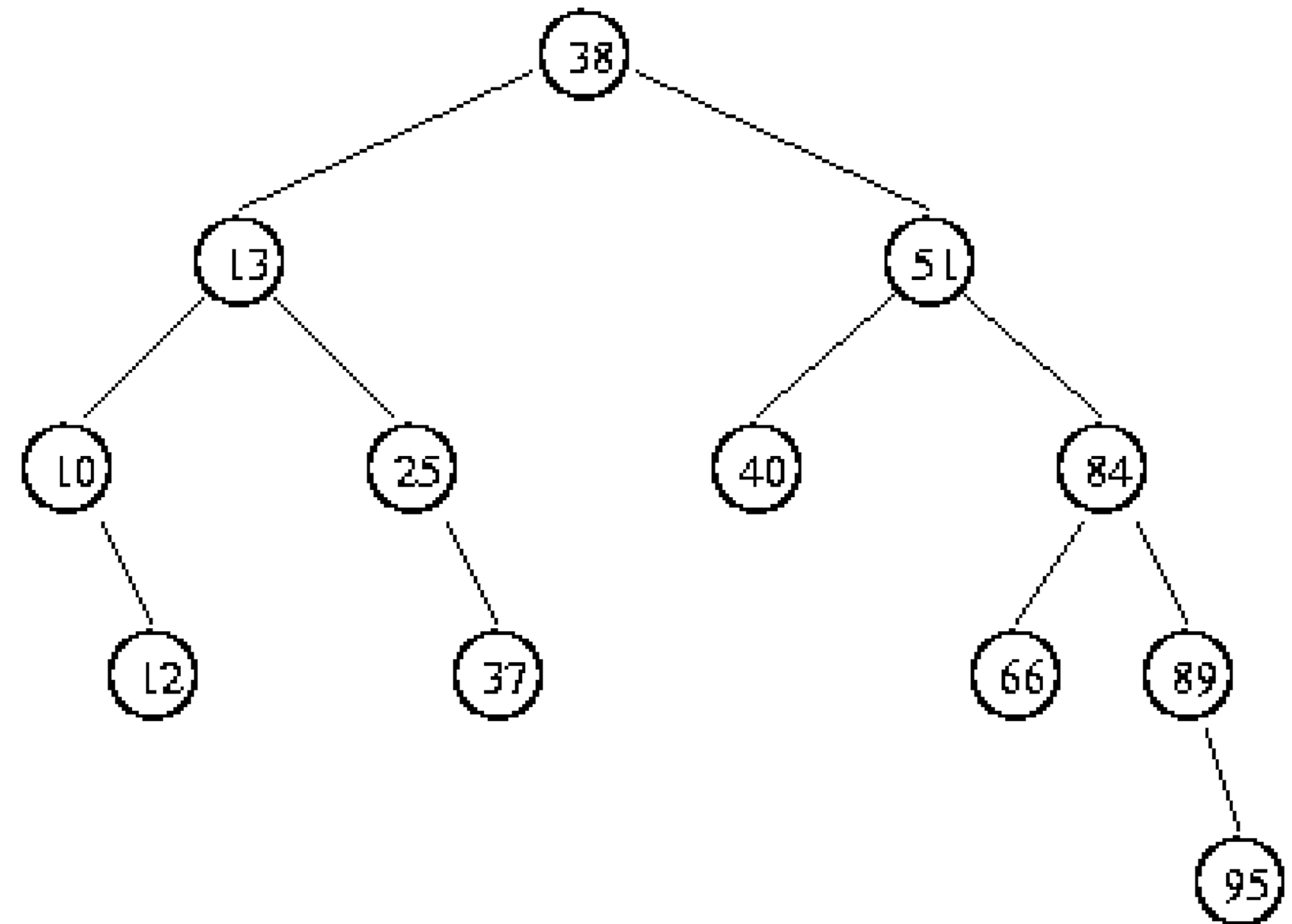

Binary Search Tree - Remove

T.remove(37);

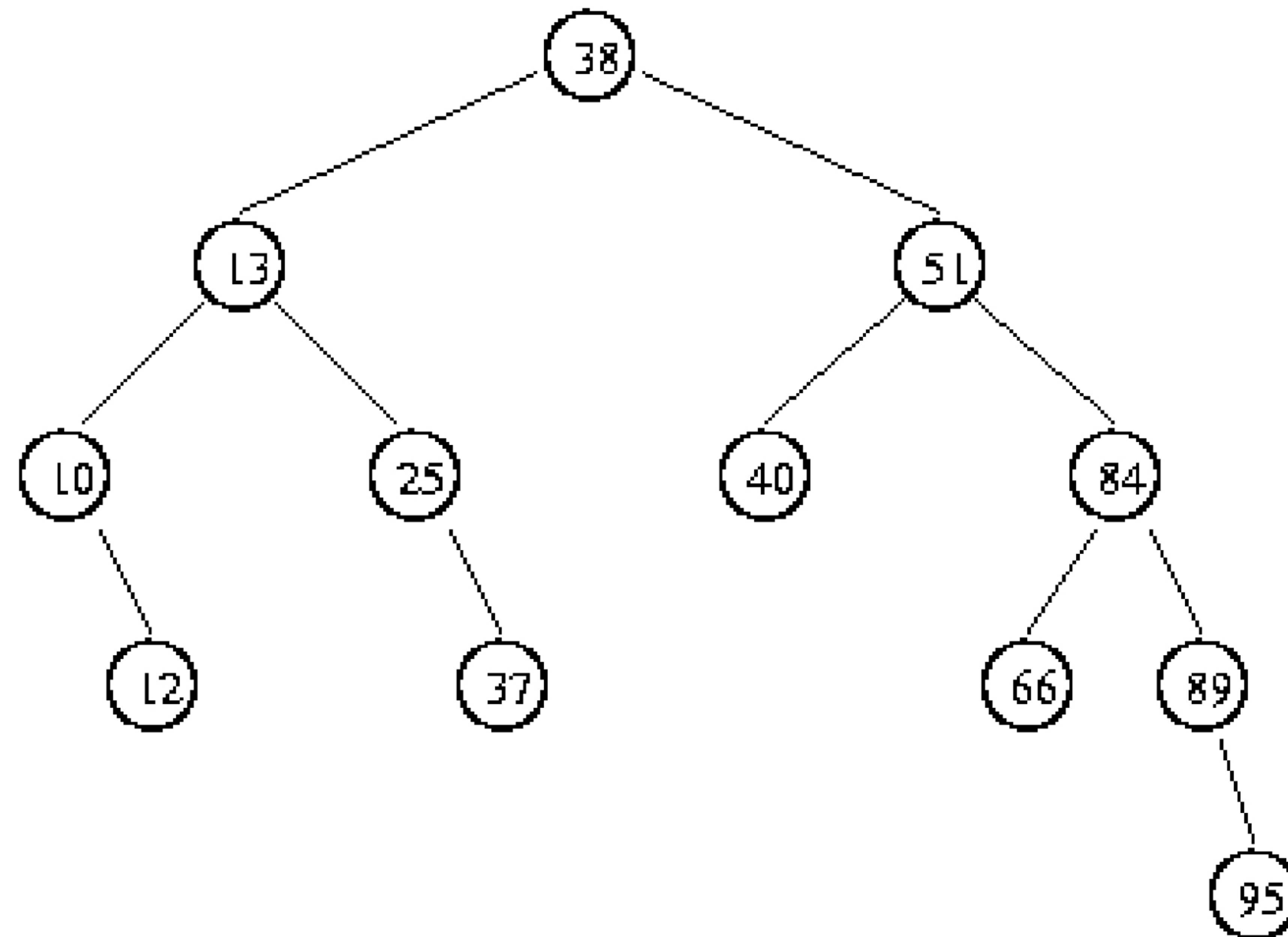
T.remove(10);



T.remove(13);



Binary Search Tree - Remove

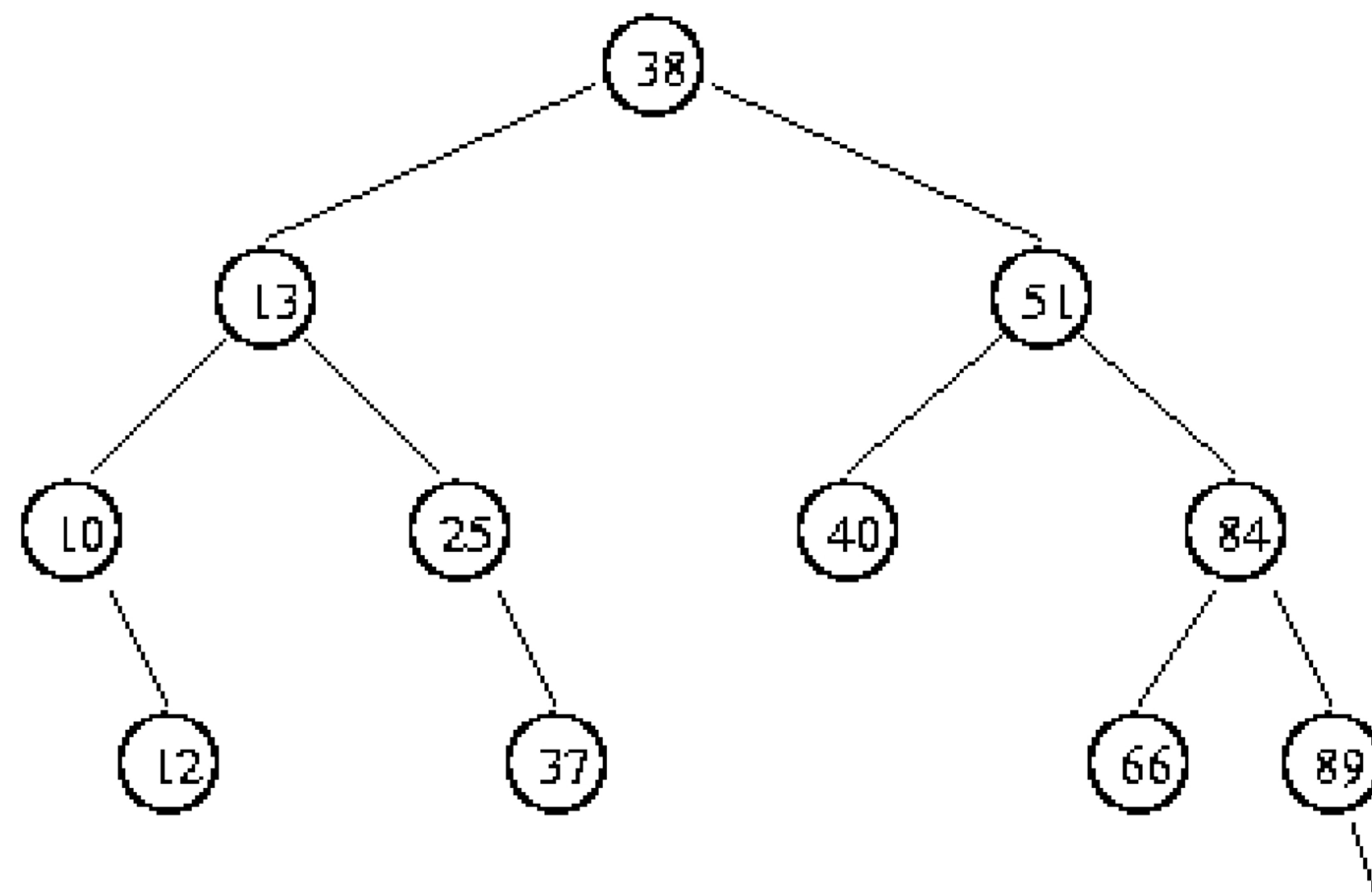


```

void BST<K>::remove(K k) {
    if (cRoot != NULL) {
        if (cRoot->key == k) {
            doRemove(cRoot);
        }
        else if (k < cRoot->key) {
            remove(cRoot->left, k);
        }
        else {
            remove(cRoot->right, k);
        }
    }
}

void BST<K>::doRemoval(treeNode * & cRoot) {
    if ((cRoot->left == NULL) && (cRoot->right == NULL)) {
        noChildRemove(cRoot);
    }
    else if ((cRoot->left != NULL) && (cRoot->right != NULL)) {
        twoChildRemove(cRoot);
    }
    else {
        oneChildRemove(cRoot);
    }
}
  
```


Binary Search Tree - Remove



```

void BST<K>::remove(cRoot) {
    if (cRoot != NULL) {
        if (cRoot->left == NULL) {
            doRemove(cRoot);
        }
        else if (cRoot->right == NULL) {
            remove(cRoot->right);
        }
        else {
            remove(cRoot->left);
        }
    }
}
  
```

```

void BST<K>::doRemove(cRoot) {
    if ((cRoot->left == NULL) && (cRoot->right == NULL)) {
        noChildRemove(cRoot);
    }
    else if ((cRoot->left != NULL) && (cRoot->right != NULL)) {
        twoChildRemove(cRoot);
    }
    else {
        oneChildRemove(cRoot);
    }
}
  
```

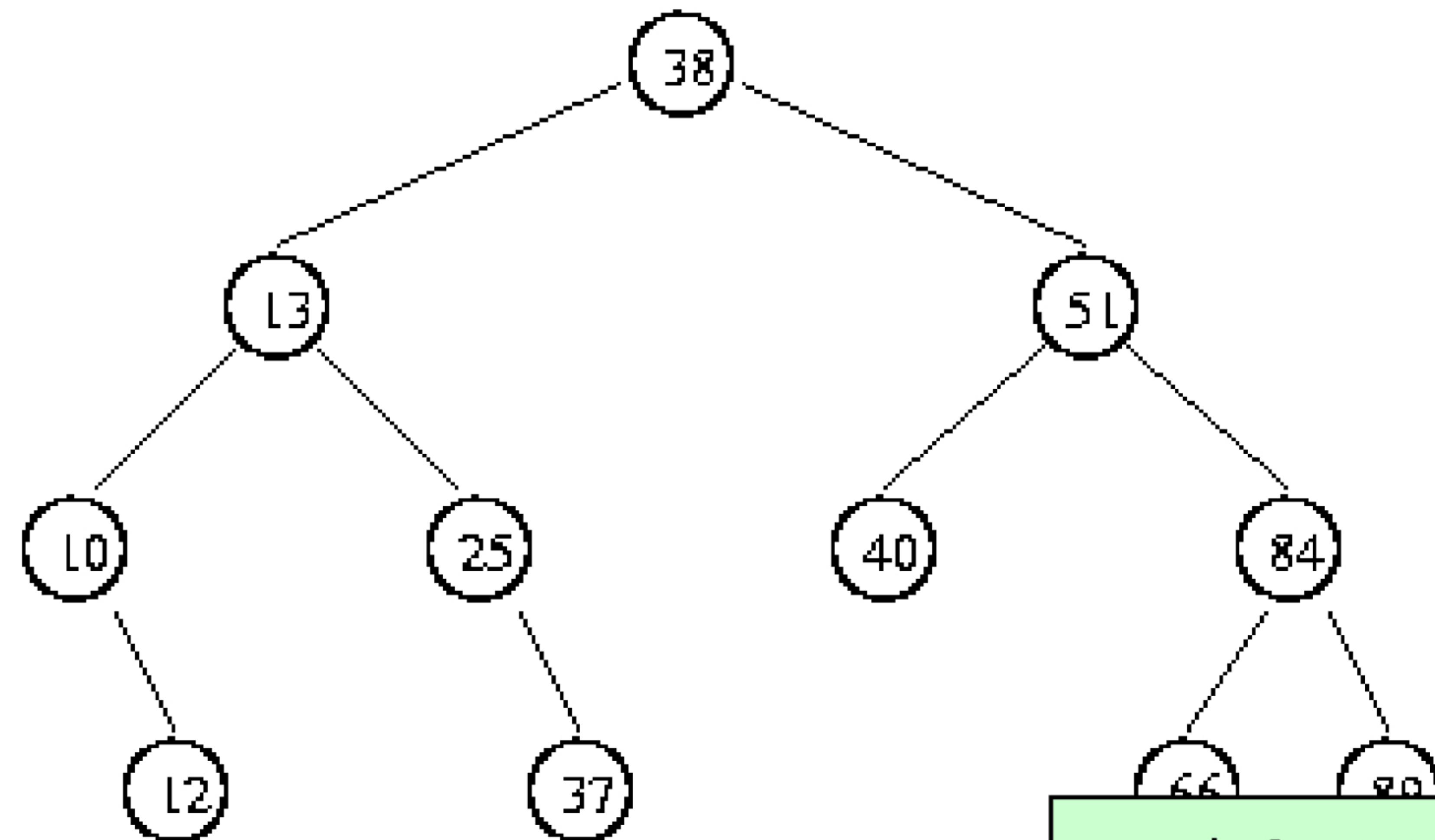
```

void BST<T,S>::noChildRemove(treeNode * & cRoot) {
    treeNode * temp = cRoot;
    cRoot = NULL;
    delete temp;
}
  
```

```

void BST<K>::oneChildRemove(treeNode * & cRoot) {
    treeNode * temp = cRoot;
    if (cRoot->left == NULL) cRoot = cRoot->right;
    else cRoot = cRoot->left;
    delete temp;
}
  
```


Binary Search Tree - Remove



```

void BST<K>::remove(treeNode * & cRoot) {
    if (cRoot != NULL) {
        if (cRoot->key == key) {
            doRemoval(cRoot);
        }
        else if (key < cRoot->key) {
            remove(cRoot->left);
        }
        else {
            remove(cRoot->right);
        }
    }
}
  
```

```

void BST<K>::doRemoval(treeNode * cRoot) {
    if ((cRoot->left == NULL) && (cRoot->right == NULL)) {
        delete cRoot;
        return;
    }
    else if (cRoot->left == NULL) {
        cRoot->left = cRoot->right;
        return;
    }
    else if (cRoot->right == NULL) {
        cRoot->right = cRoot->left;
        return;
    }
    else {
        twoChildRemove(cRoot);
    }
}
  
```

```

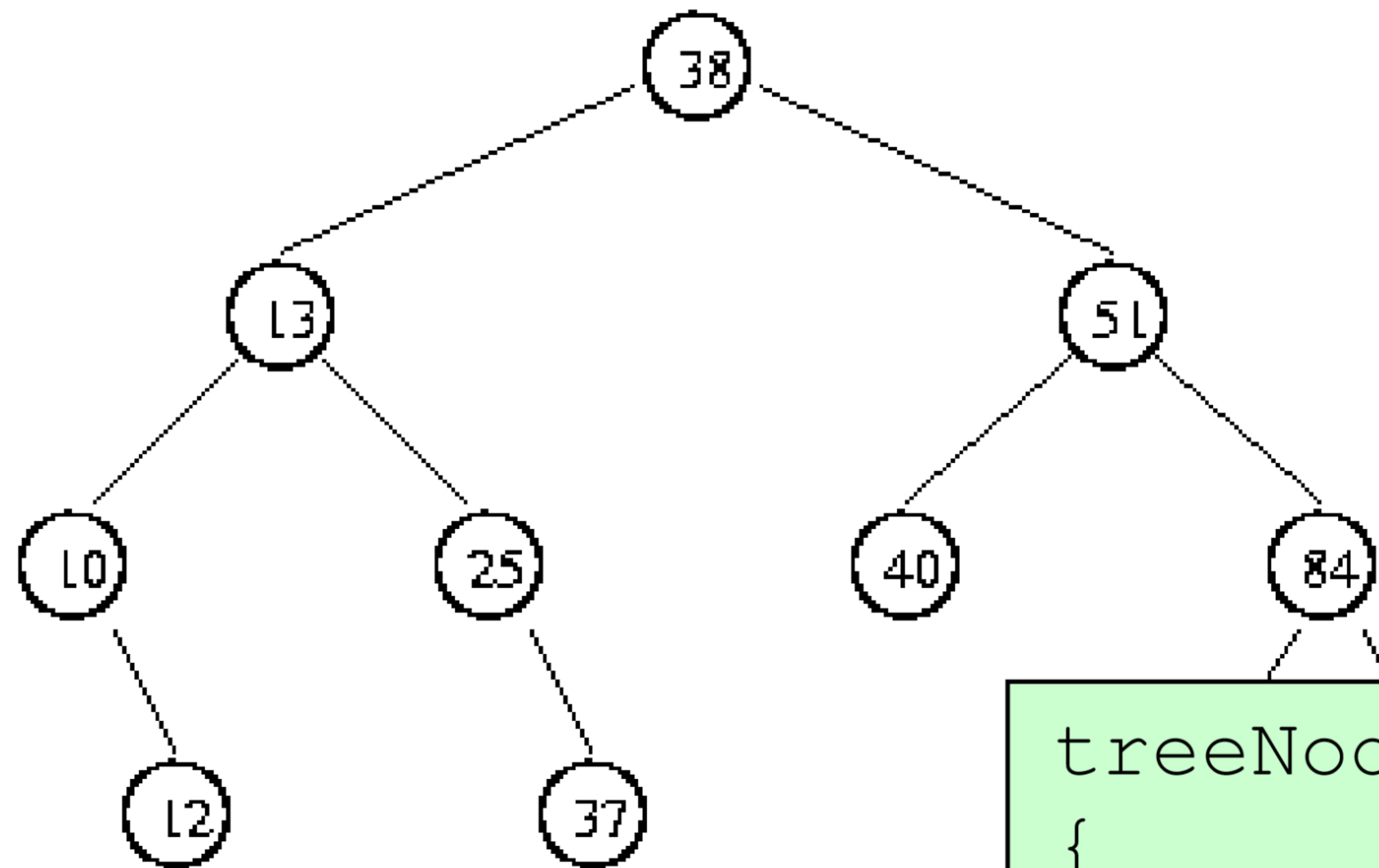
void BST<K>::twoChildRemove(treeNode * & cRoot) {
    treeNode * iop = rightMostChild(cRoot->left);
    cRoot->key = iop->key;
    doRemoval(iop);
}
  
```

doRem

remov

remove(cRoot->right, 81);

Binary Search Tree - Remove



```

void BST<K>::remove(cRoot)
{
    if (cRoot != NULL)
    {
        if (cRoot->right == NULL)
        {
            doRemove(cRoot);
        }
        else if (cRoot->left == NULL)
        {
            remove(cRoot->right, &cRoot);
        }
        else
        {
            remove(cRoot->right, &cRoot);
        }
    }
}
  
```

```

void BST<K>::remove(cRoot)
{
    if ((cRoot->left == NULL) && (cRoot->right == NULL))
    {
        noChildRemove(cRoot);
    }
    else if ((cRoot->left == NULL) && (cRoot->right != NULL))
    {
        oneChildRemove(cRoot);
    }
    else if ((cRoot->left != NULL) && (cRoot->right != NULL))
    {
        twoChildRemove(cRoot);
    }
}
  
```

```

treeNode * & BST<K>::rightMostChild(treeNode * & cRoot)
{
    if (cRoot->right == NULL) return cRoot;
    else return rightMostChild(cRoot->right);
}
  
```

```

}
  
```

```

}
  
```

```

twoChildRemove(cRoot);
  
```

```

else
  
```

```

oneChildRemove(cRoot);
  
```

```

}
  
```

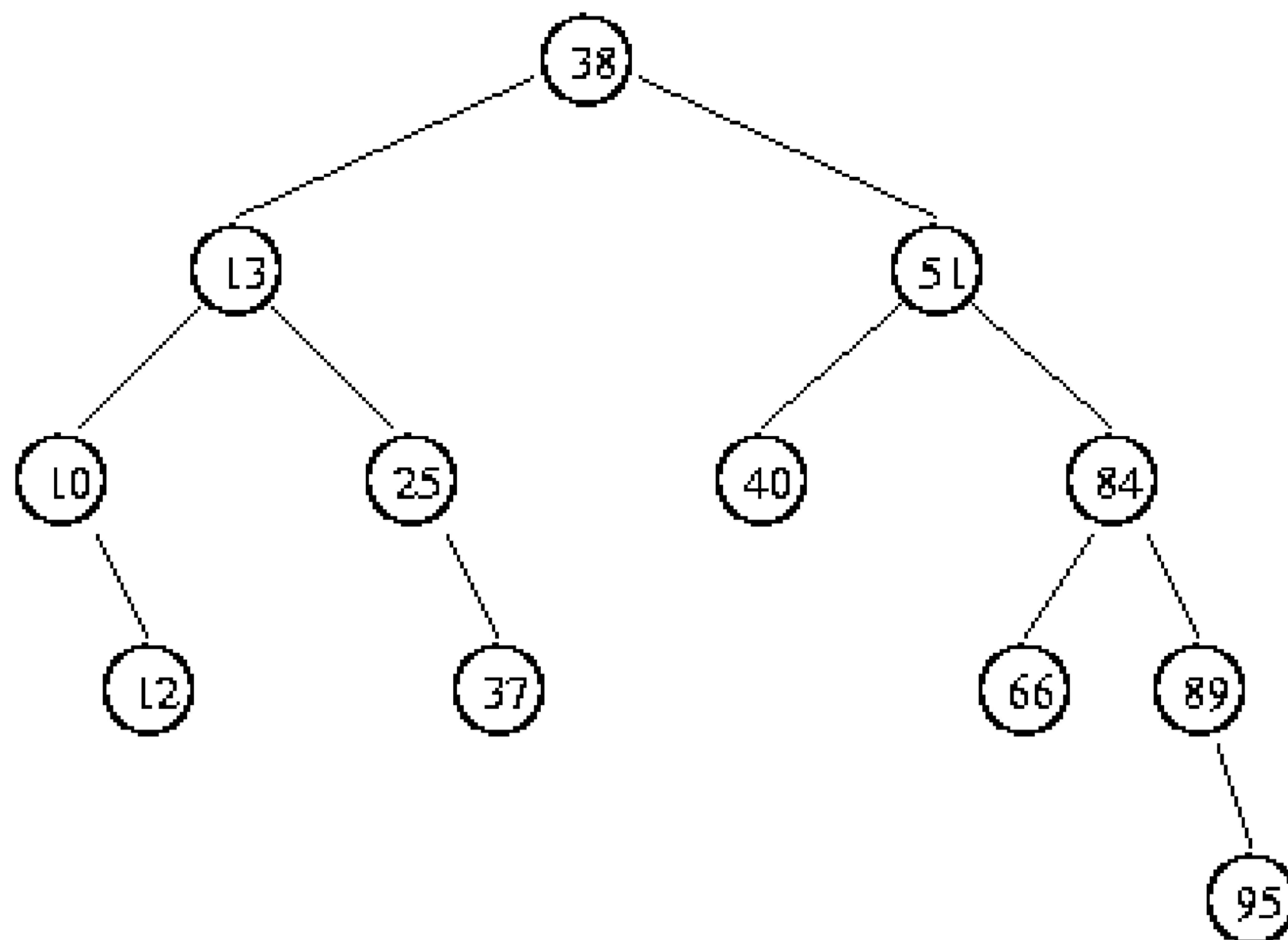
remove(cRoot->right, &cRoot);

Announcements

- MP5 is due on Friday, 04/01 @ 11:59pm, E/C is due on Friday, 03/18 @ 11:59pm
- Exam 3: March 30 - Apr 1. Topics:
- Exam 3 solution party, Monday March 28 @9pm, location TBD, (Siebel 1404)
- MP5.1 will be on the exam!!!
- Labs this week: dict (due march 27) + huffman (due Apr 3)

TODAY: balanced BST (intro)

<http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html>



Running times:

insert

remove

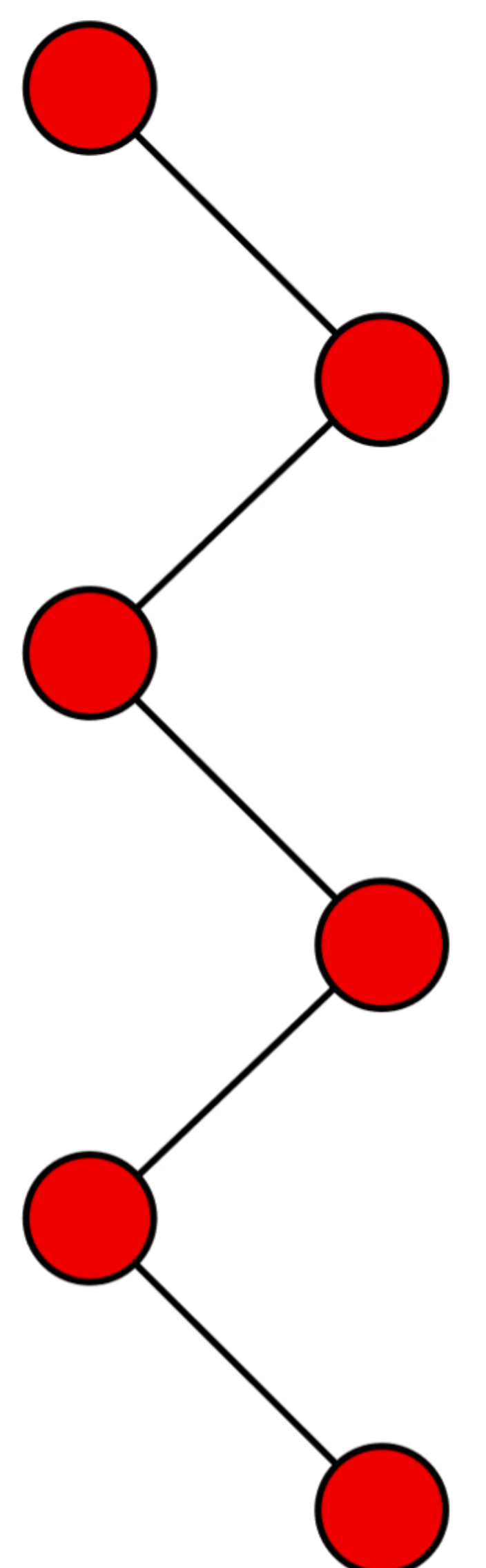
find

traverse

Binary Search Tree - miscellaneous characteristics and analysis

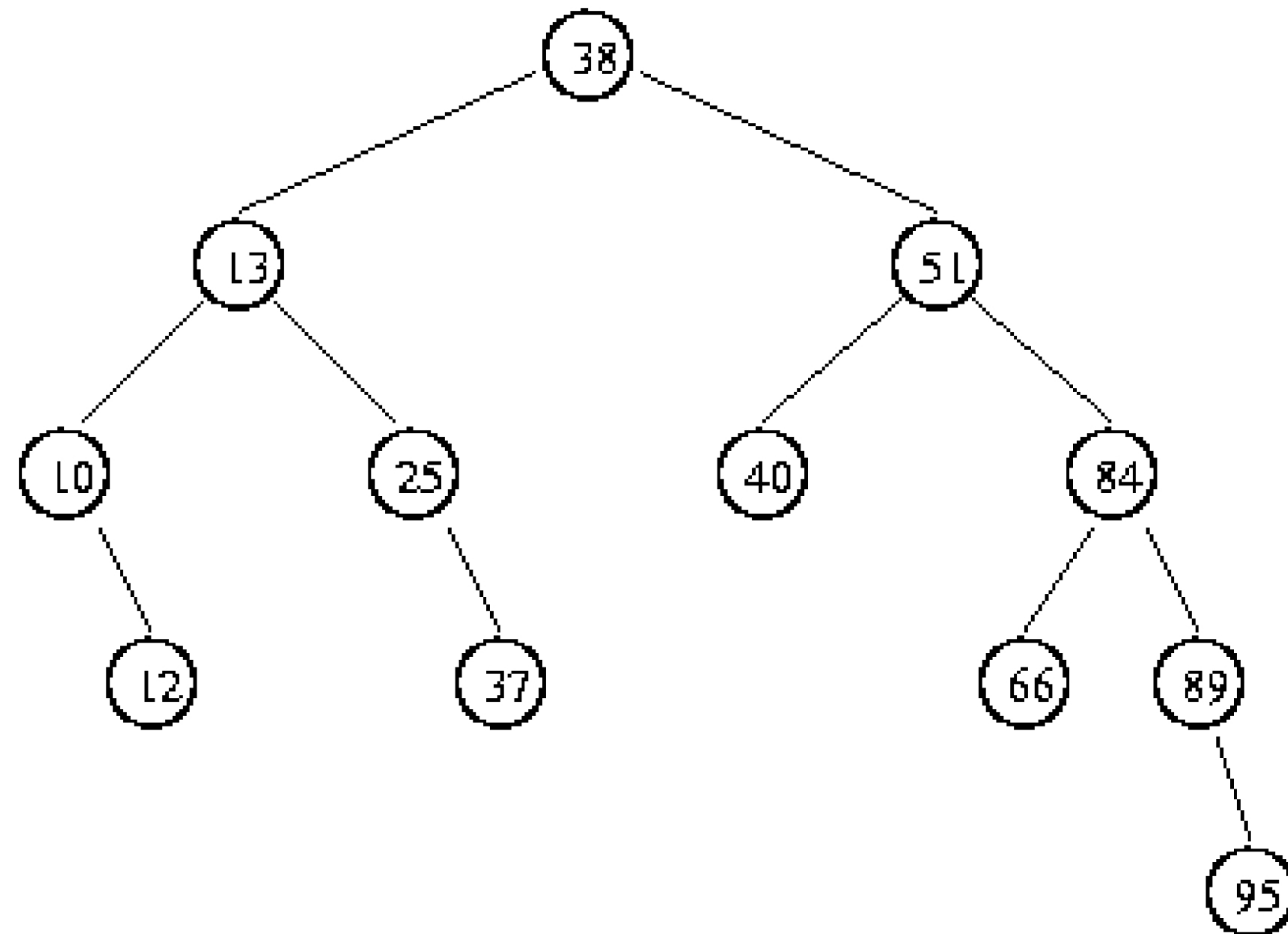
```
BST<int> myT;  
myT.insert(2);  
myT.insert(7);  
myT.insert(15);  
myT.insert(22);  
myT.insert(28);  
...
```

Give a sequence of inserts that result in a tree that looks like:



How many “bad” n-item trees are there?

Binary Tree -



The *algorithms* on BST depend on the height (h) of the tree.

The *analysis* should be in terms of the amt of data (n) the tree contains.

So we need a relationship between h and n .

$$h \geq f(n)$$

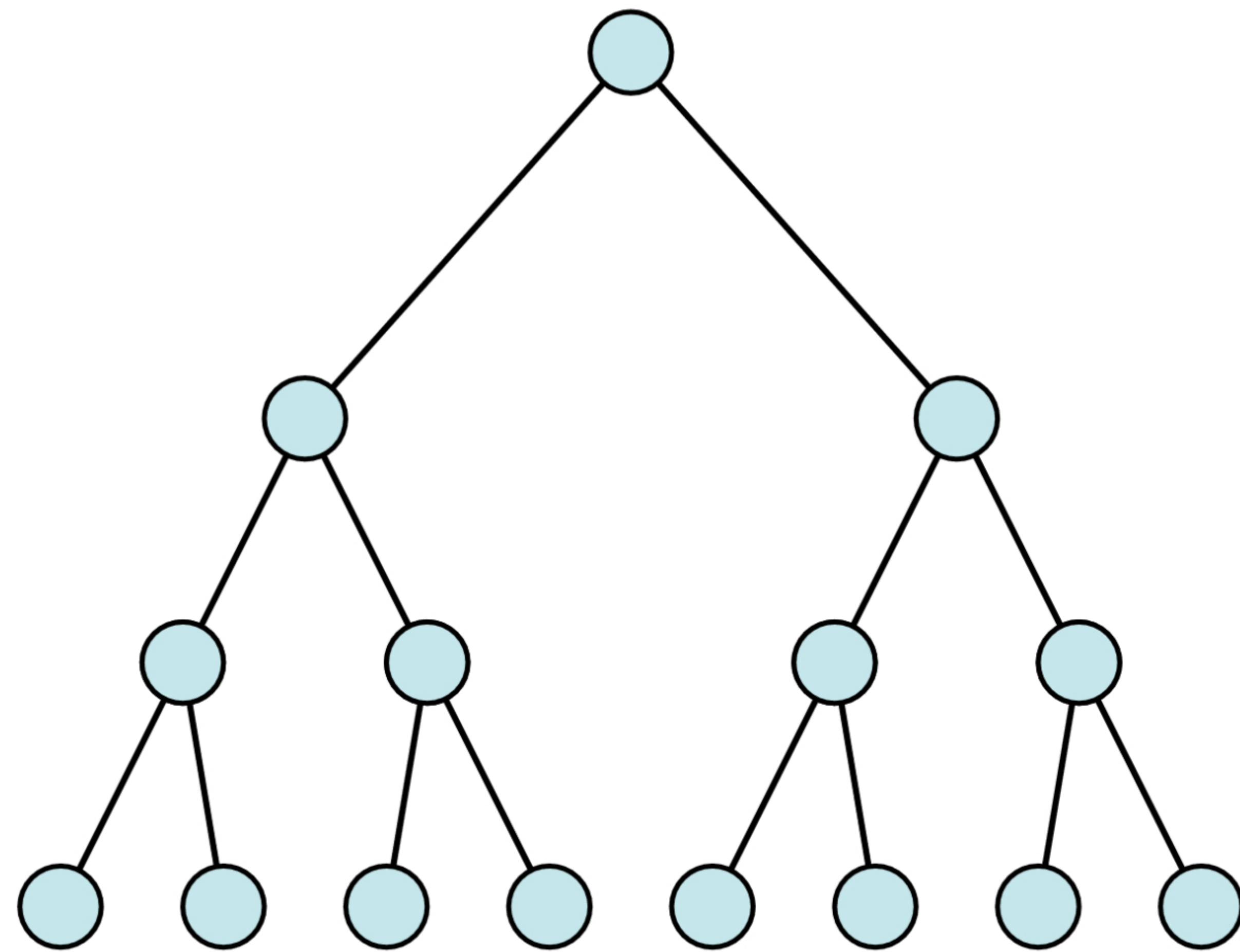
$$h \leq g(n)$$

Reminder:

height(T) is:

- _____ if T is empty
- $1 + \max\{\text{height}(T_L), \text{height}(T_R)\}$, otherwise

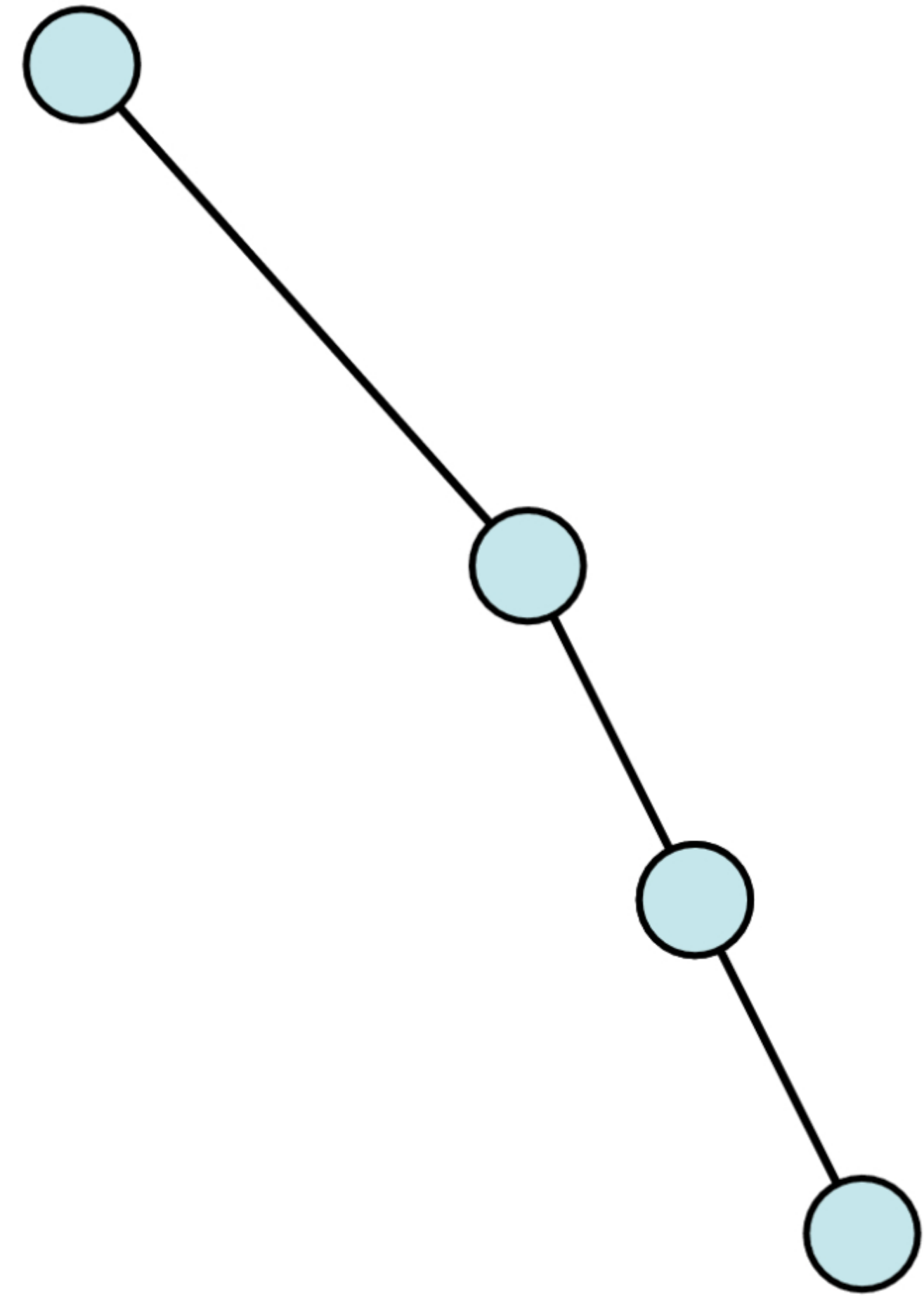
Binary Tree (theory moment #1)



what is maximum number of nodes in a tree of height h ?

what is the least possible height (h) for a tree of n nodes?

Binary Tree (theory moment #2)



what is minimum number of nodes (n) in a tree of height h ?

what is the greatest possible height (h) for a tree of n nodes?

thus: lower bd on ht _____, upper bd on ht _____, good news or bad?

Binary Search Tree -

The height of a BST depends on the order in which the data is inserted into it.

ex. 1 3 2 4 5 7 6 vs. 4 2 3 6 7 1 5

How many different ways are there to insert n keys into a tree?

Avg height, over all arrangements of n keys is _____.

operation	avg case	worst case	sorted array	sorted list
find				
insert				
delete				
traverse				

something new... which tree makes you happiest?



The “height balance” of a tree T is:

$$b = height(T_L) - height(T_R)$$

A tree T is “height balanced” if:

-
-

Binary Search Tree - is this tree “height balanced”?

