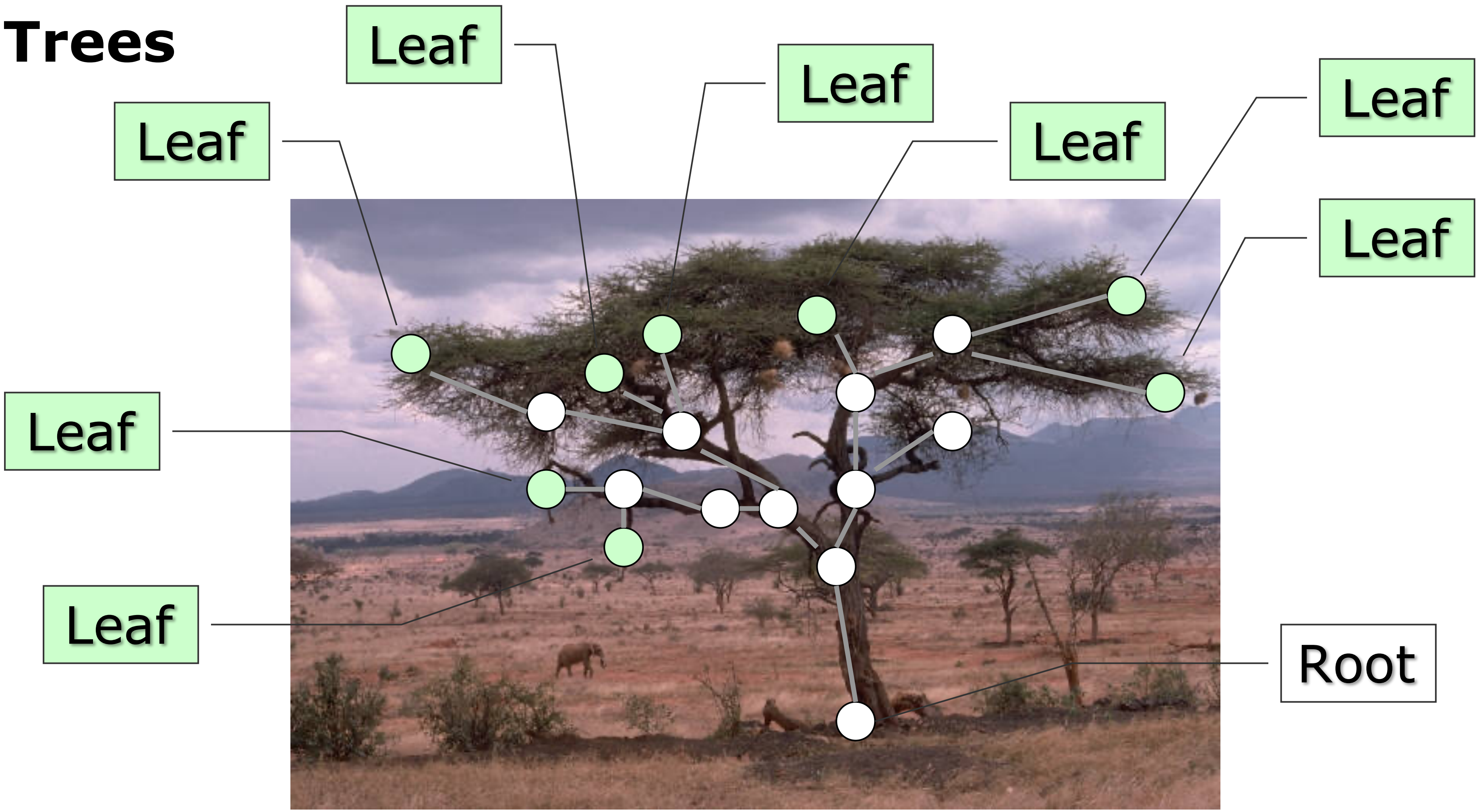
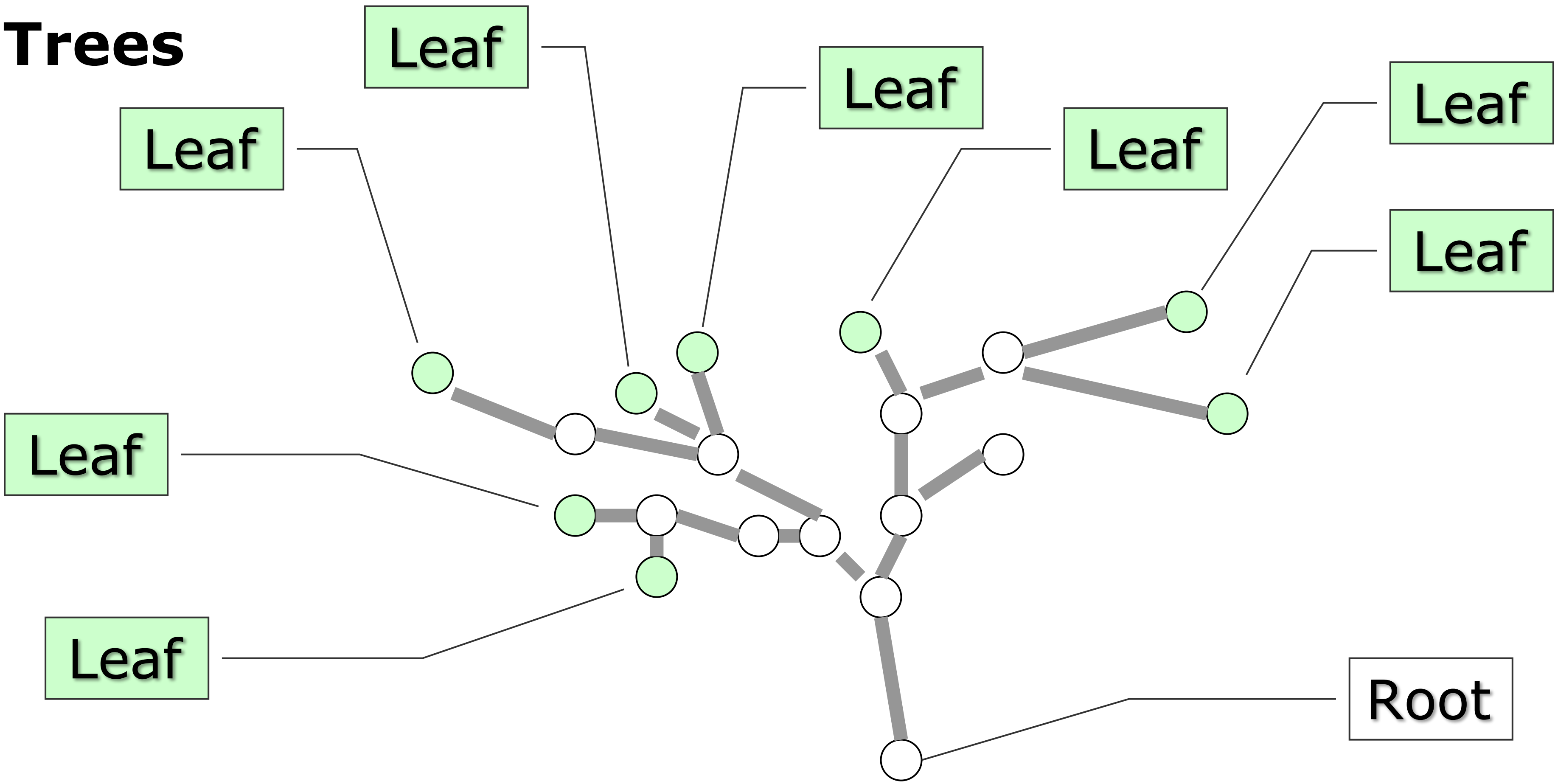
Trees

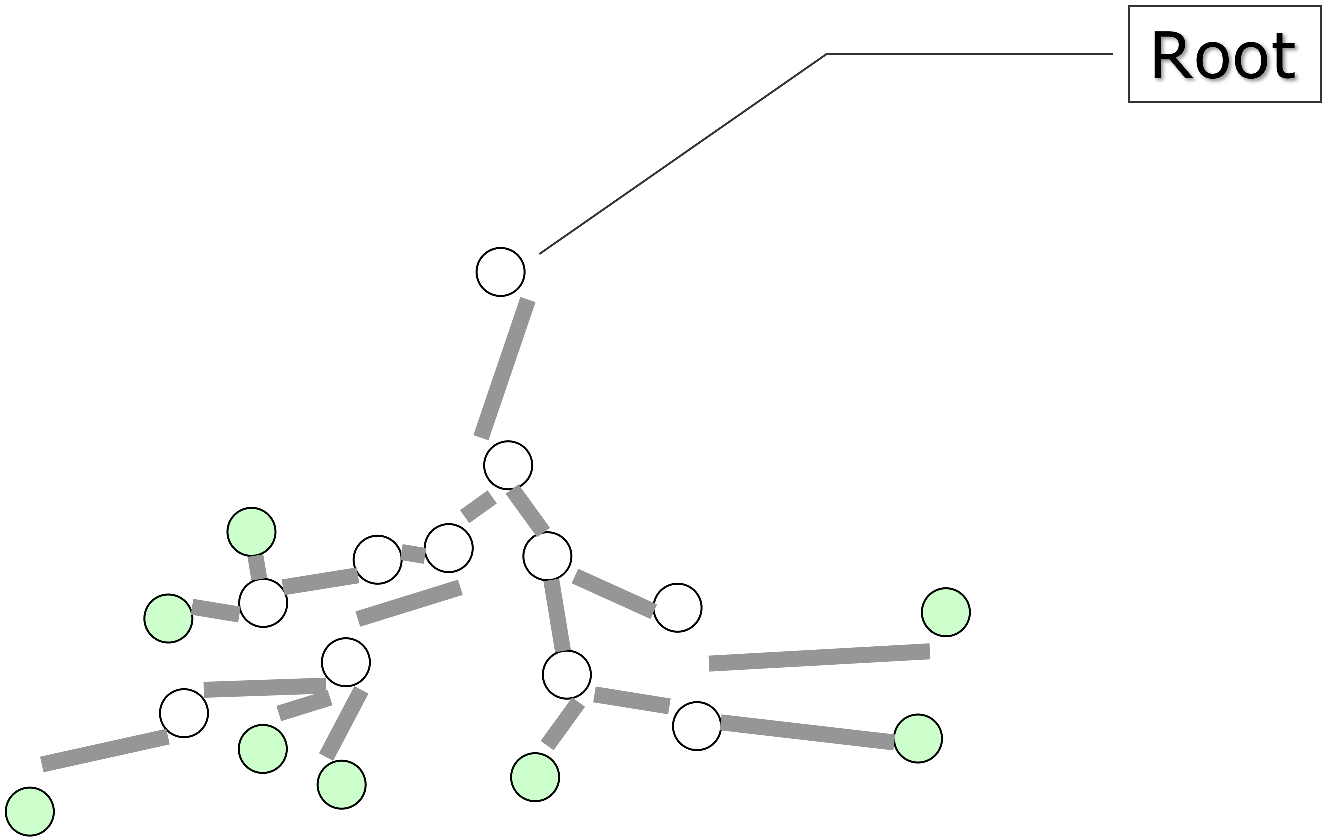
Trees



Trees

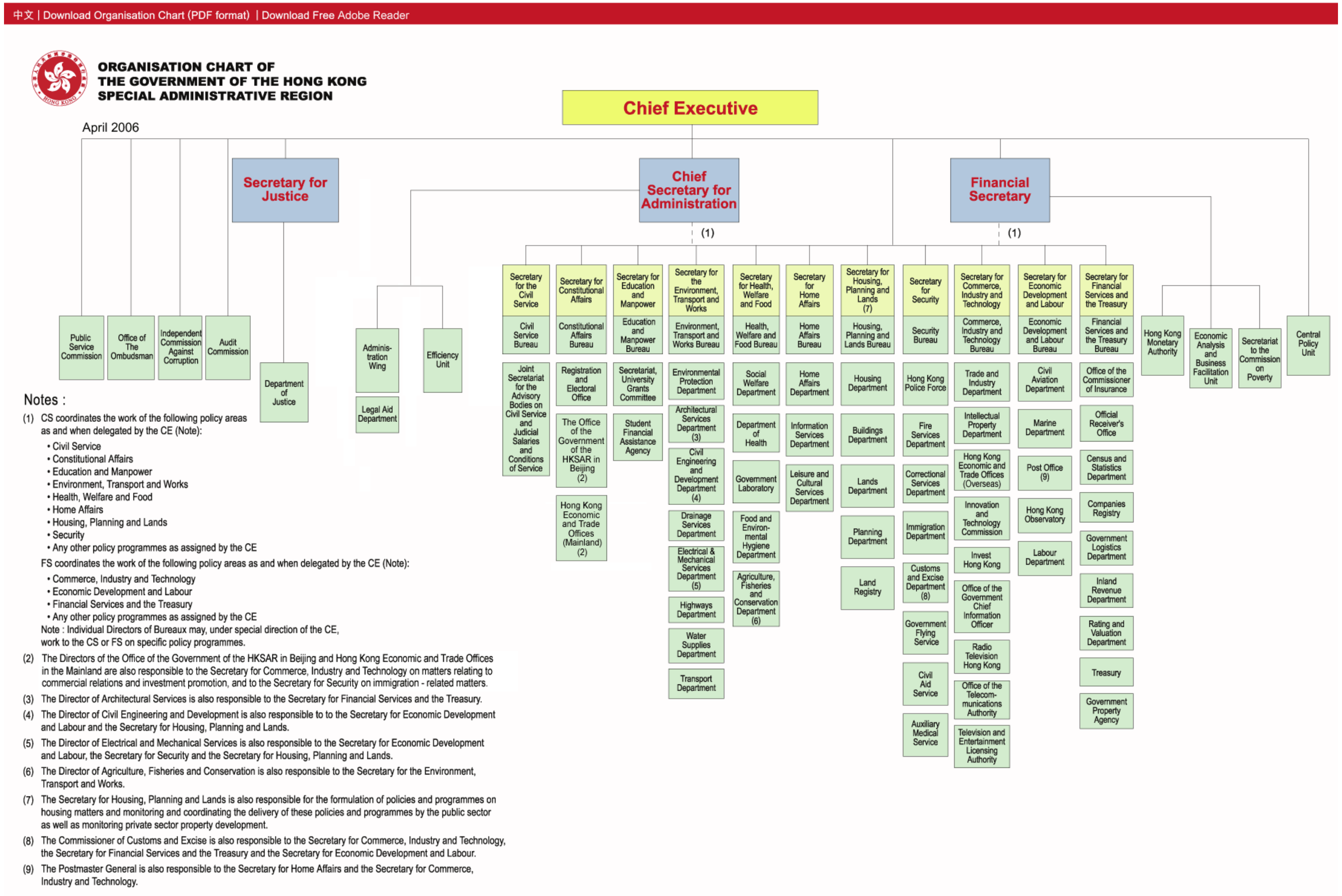


Trees



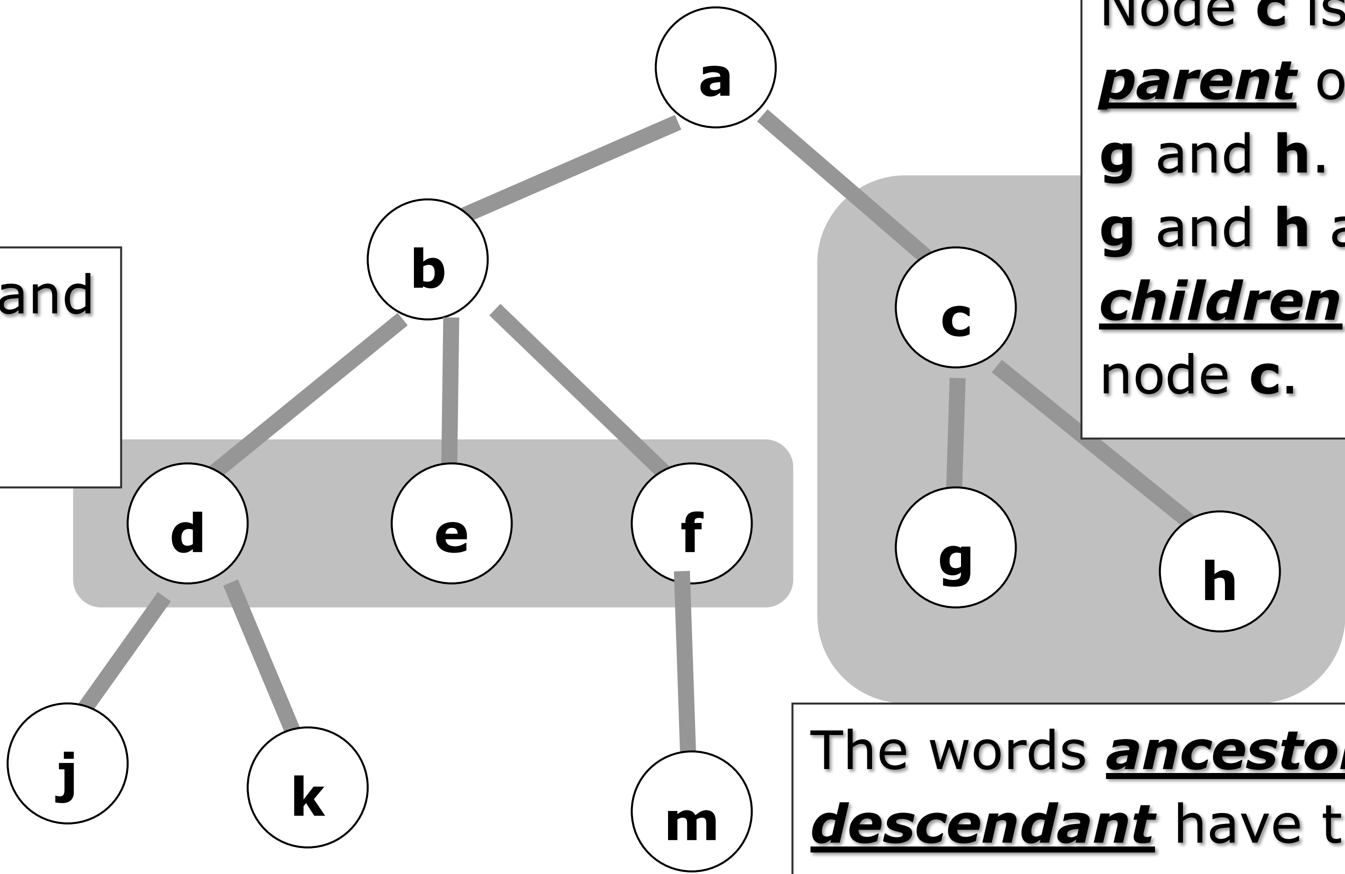
This is pretty abstract…

Are there any concrete examples?



Source: www.info.gov.hk/graphics/cht_e.gif

Terminologies

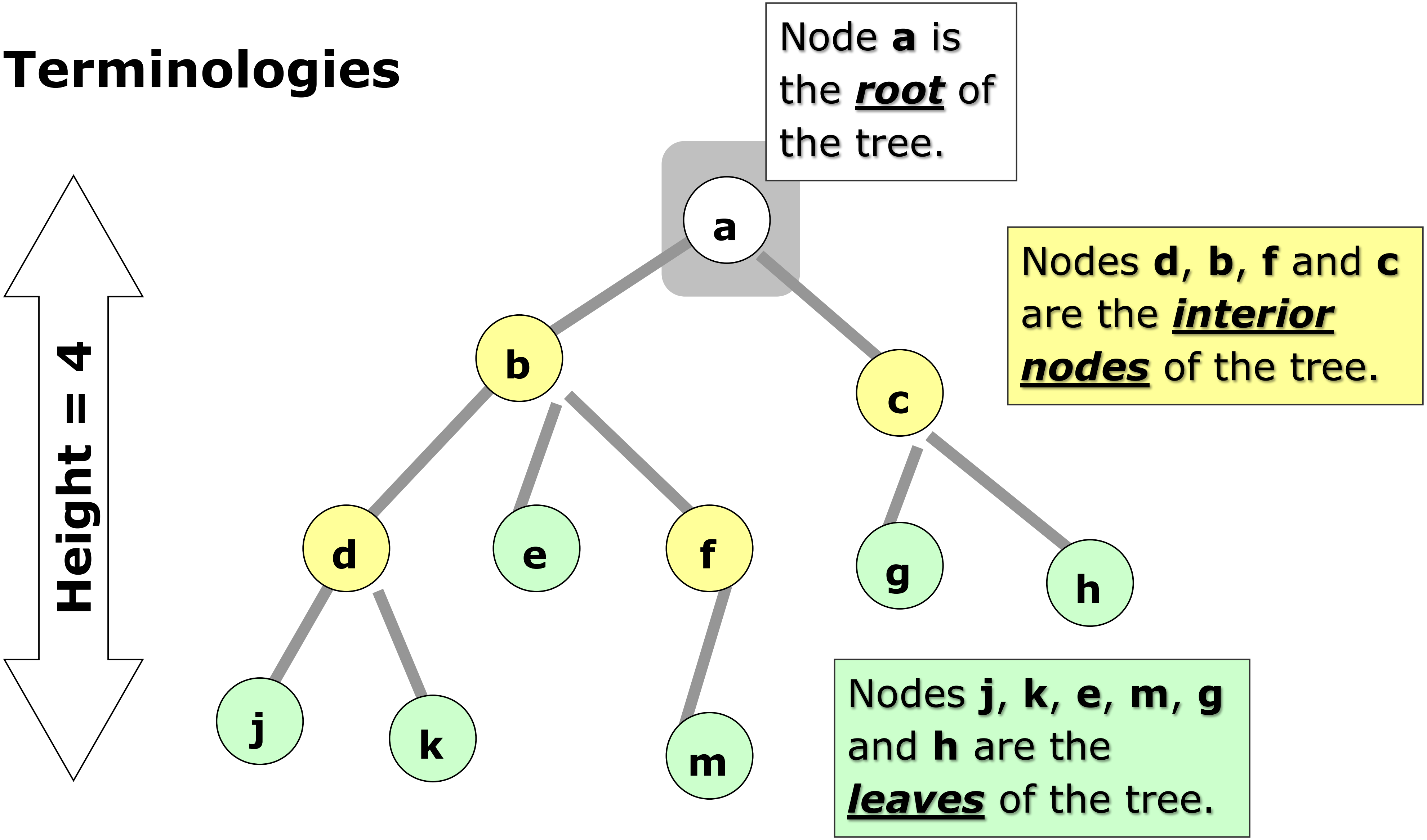


Node **c** is the ***parent*** of nodes **g** and **h**. Nodes **g** and **h** are the ***children*** of node **c**.

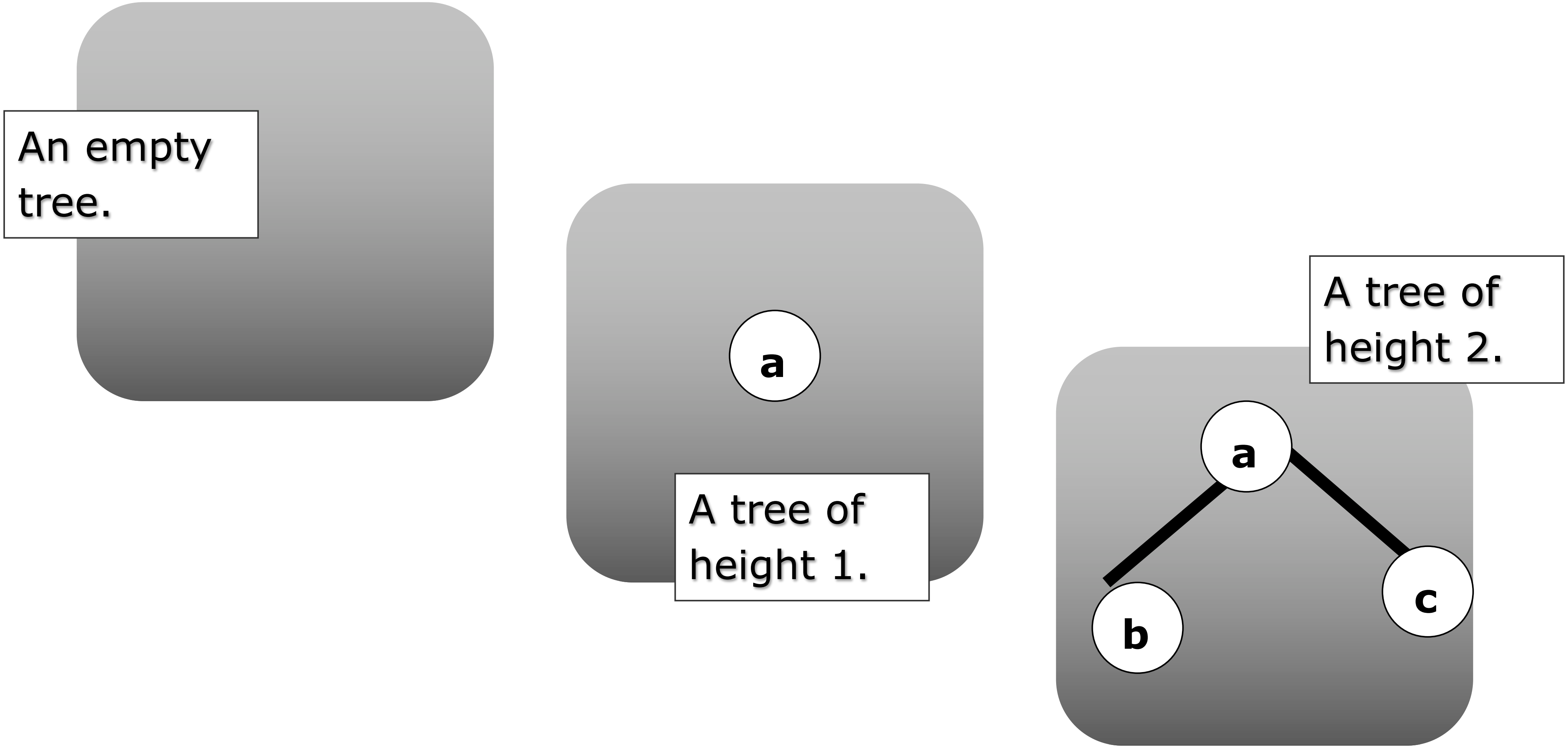
Nodes **d**, **e** and **f** are the ***siblings***.

The words ***ancestor*** and ***descendant*** have the same meaning as they do in English.

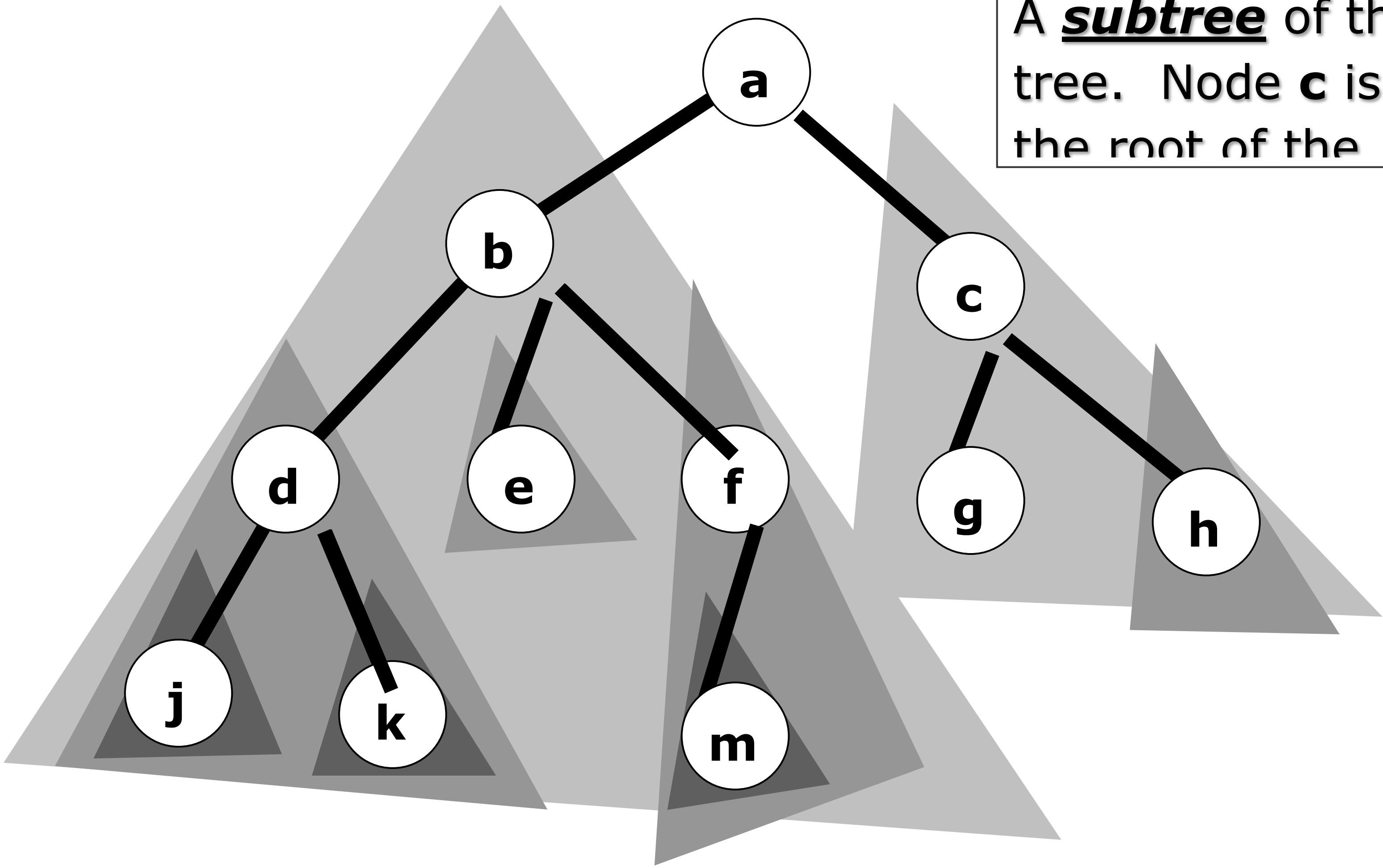
Terminologies



Terminologies



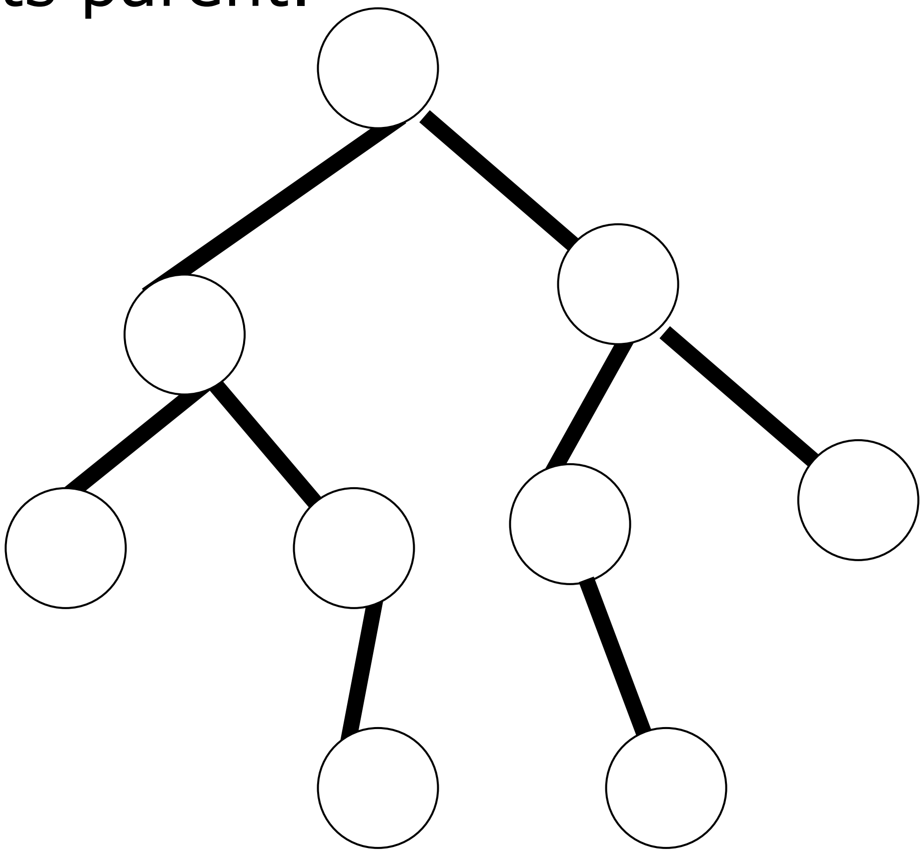
Terminologies



A ***subtree*** of the tree. Node **c** is the root of the

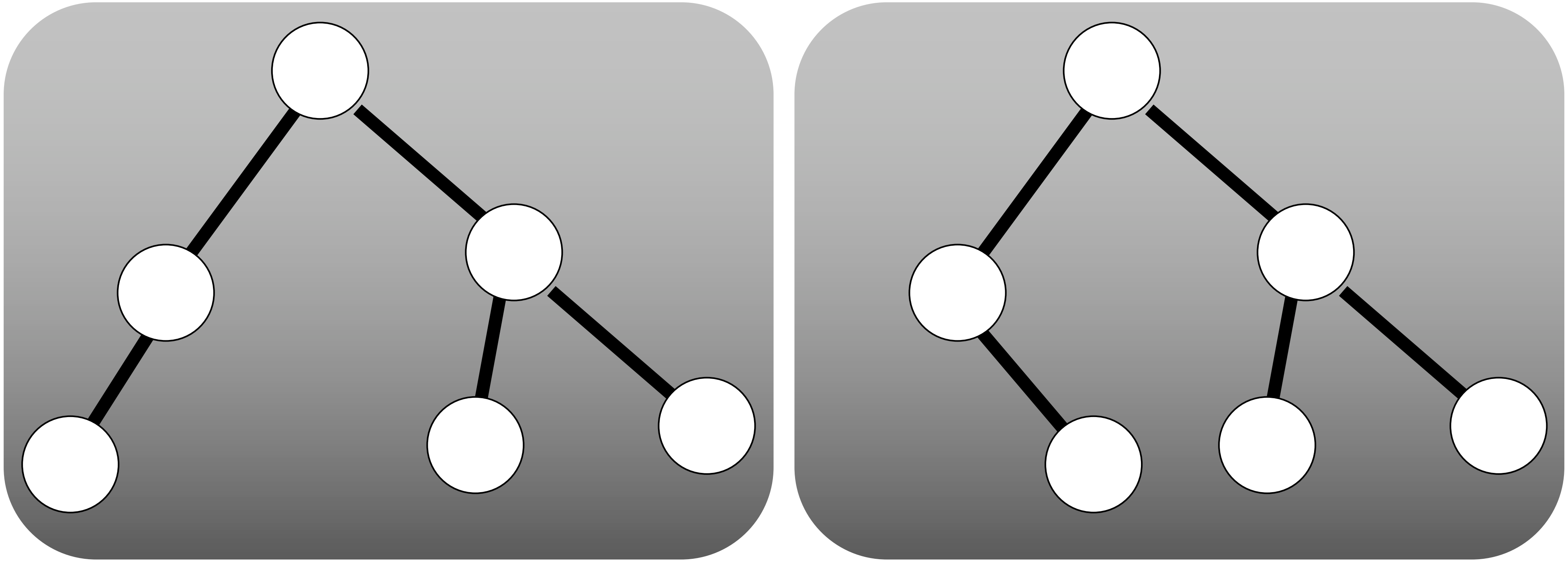
Binary Trees

- Each node in the tree has 0, 1, or 2 children.
- Every node except the root is said to be either a *left child* or a *right child* of its parent.



Binary Trees

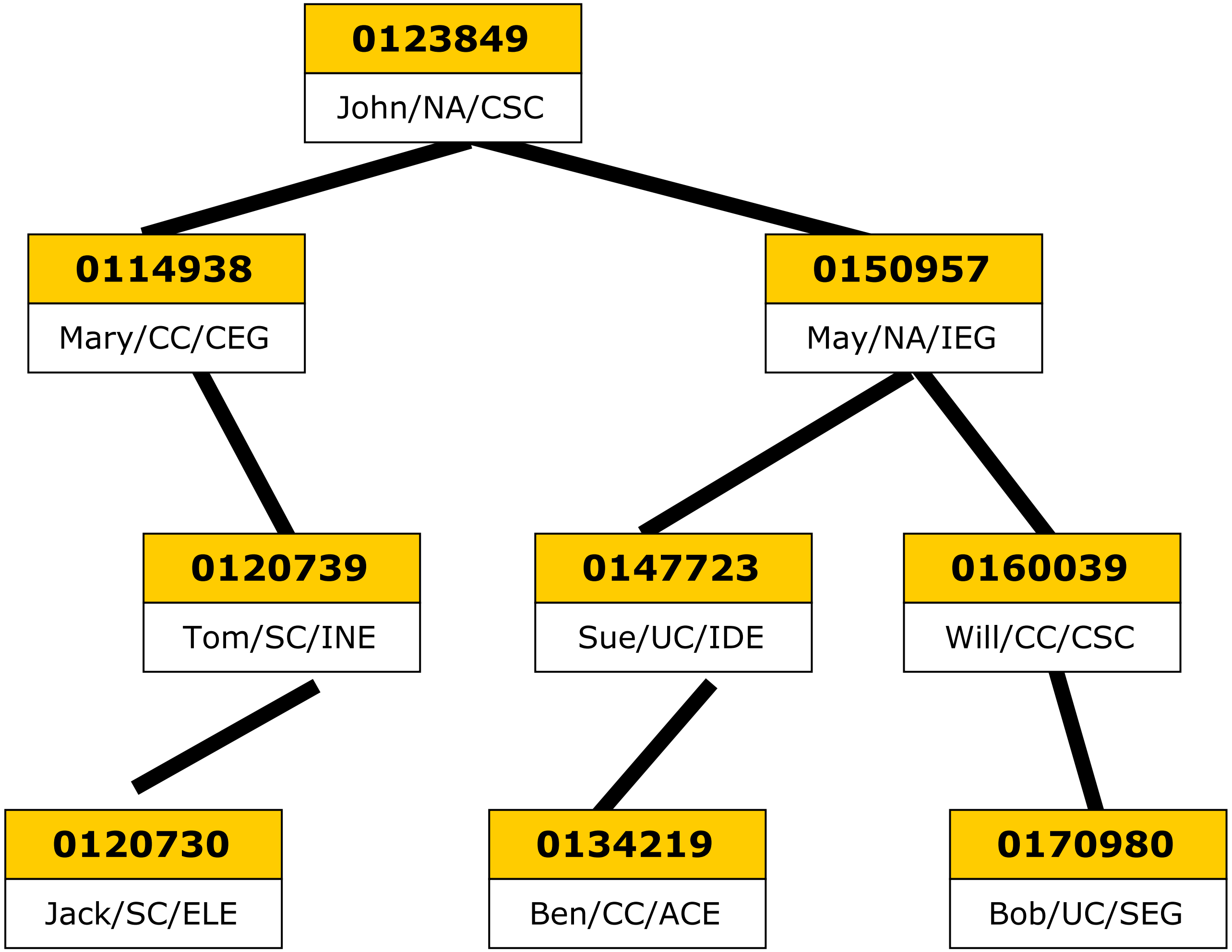
These two binary trees are different:



Binary Search Trees

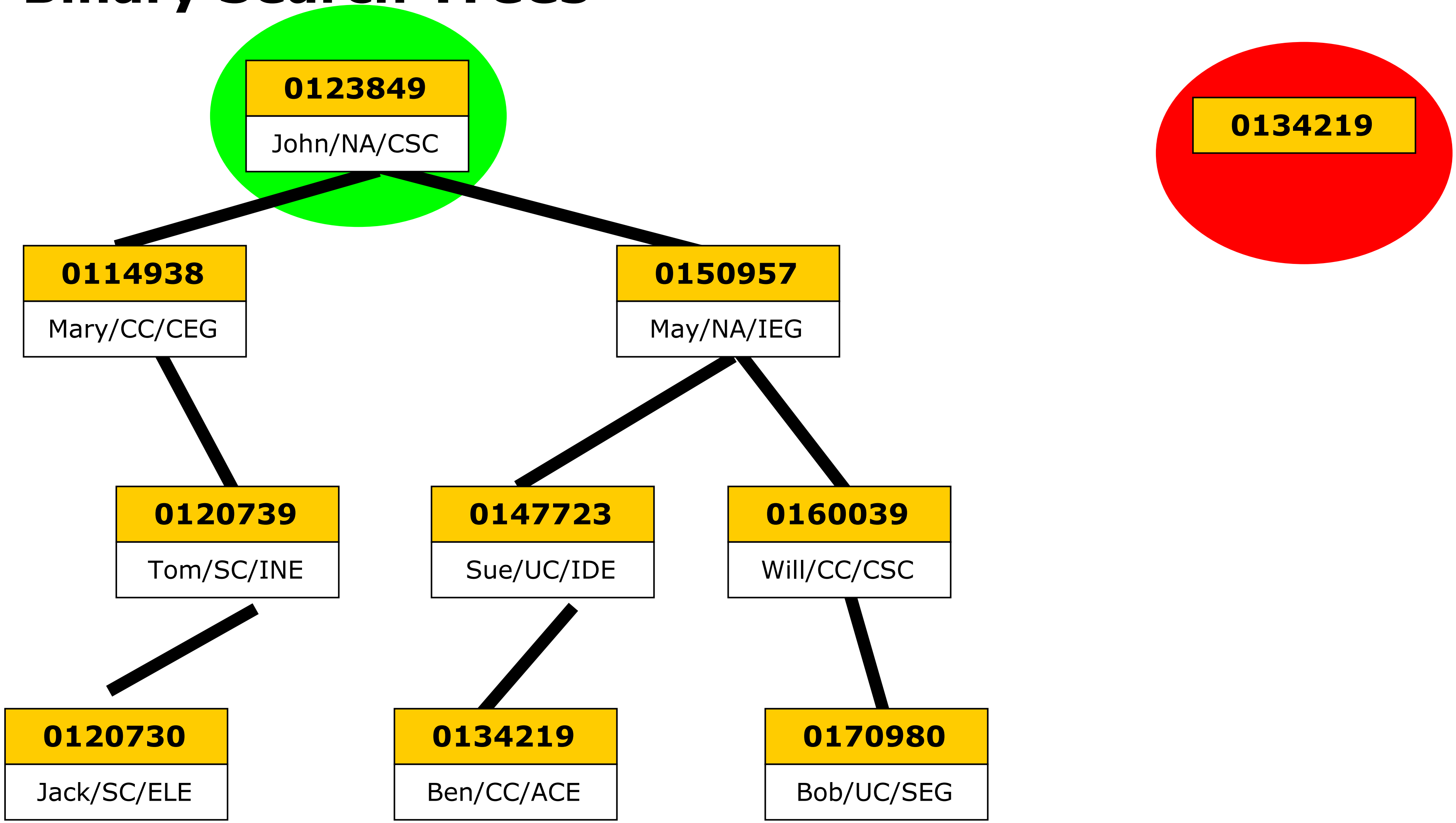
Binary Search Trees are Binary Trees that satisfy two conditions.

Binary Search Trees

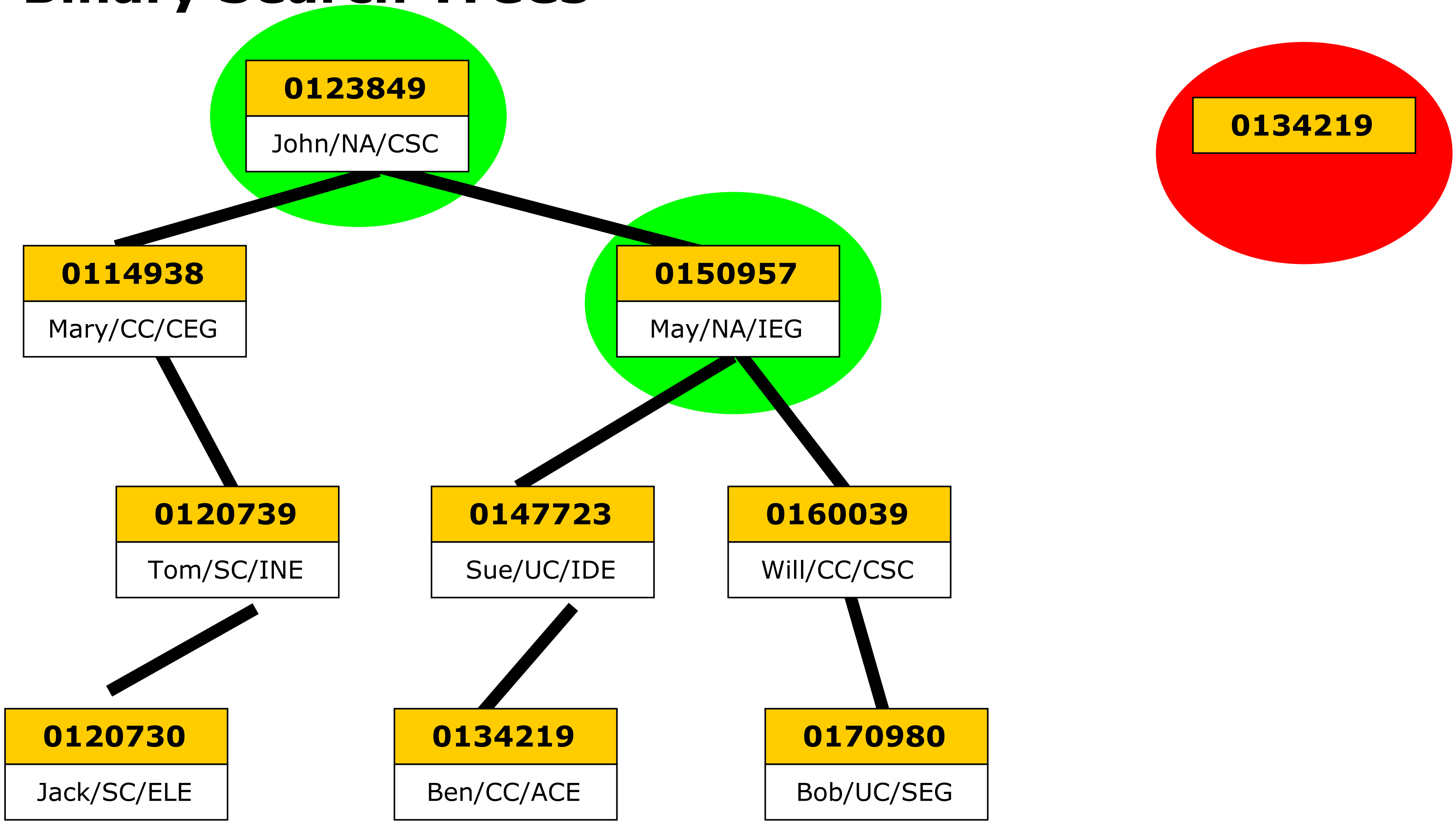


- Key values are unique.
- At every node, the key value must be *greater than* all the keys in the left subtree, and *less than* all the keys in the right subtree.

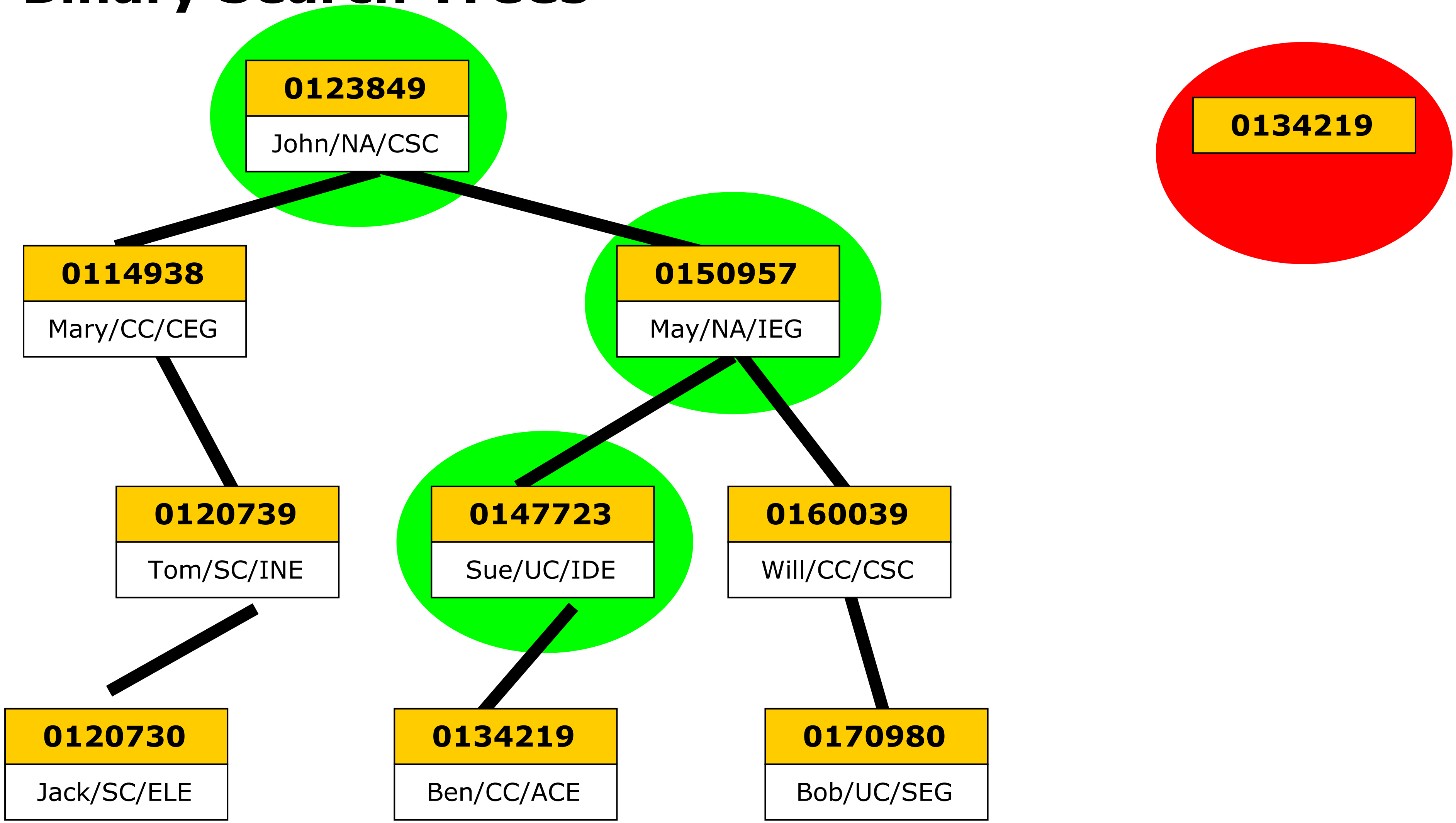
Binary Search Trees



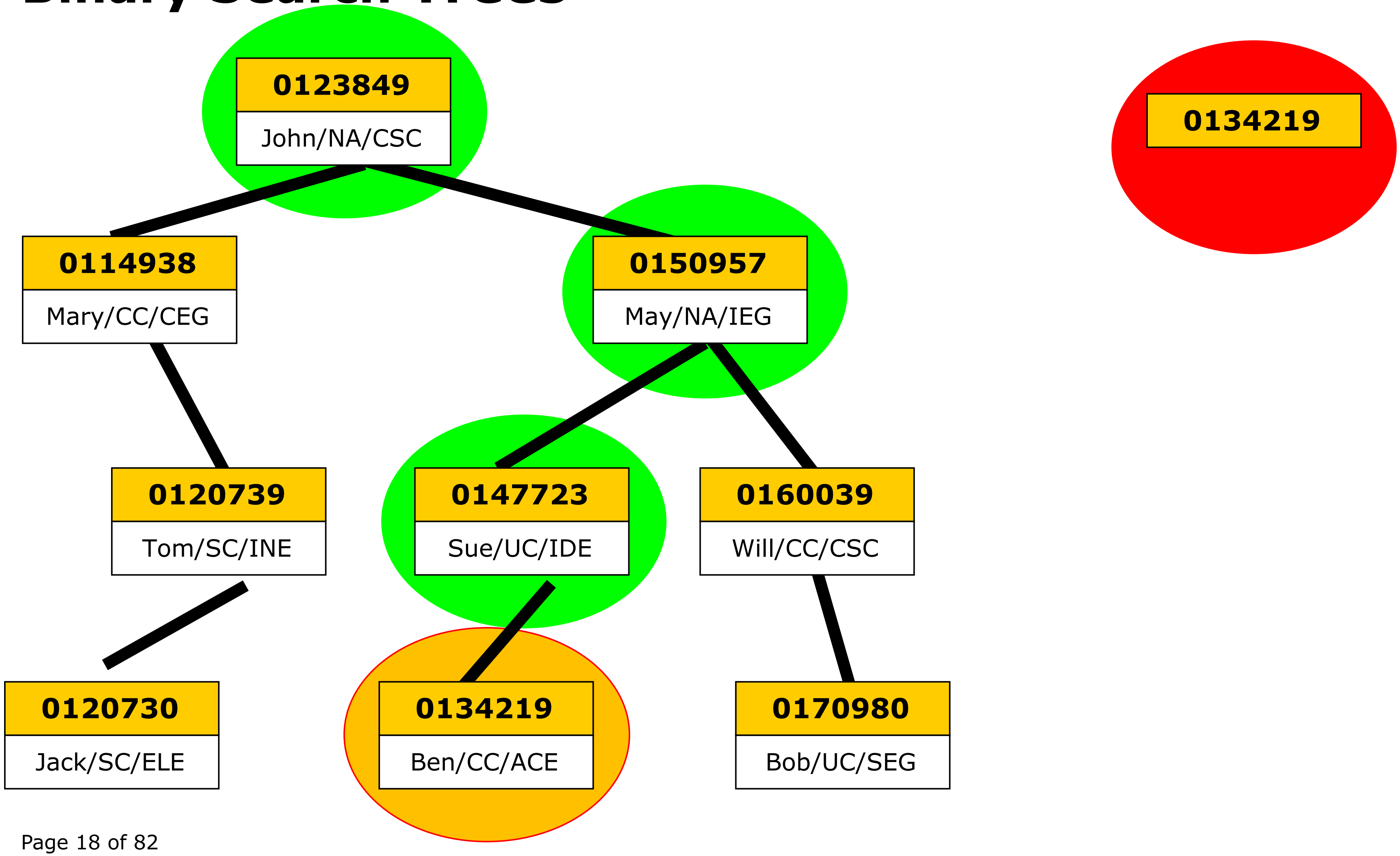
Binary Search Trees



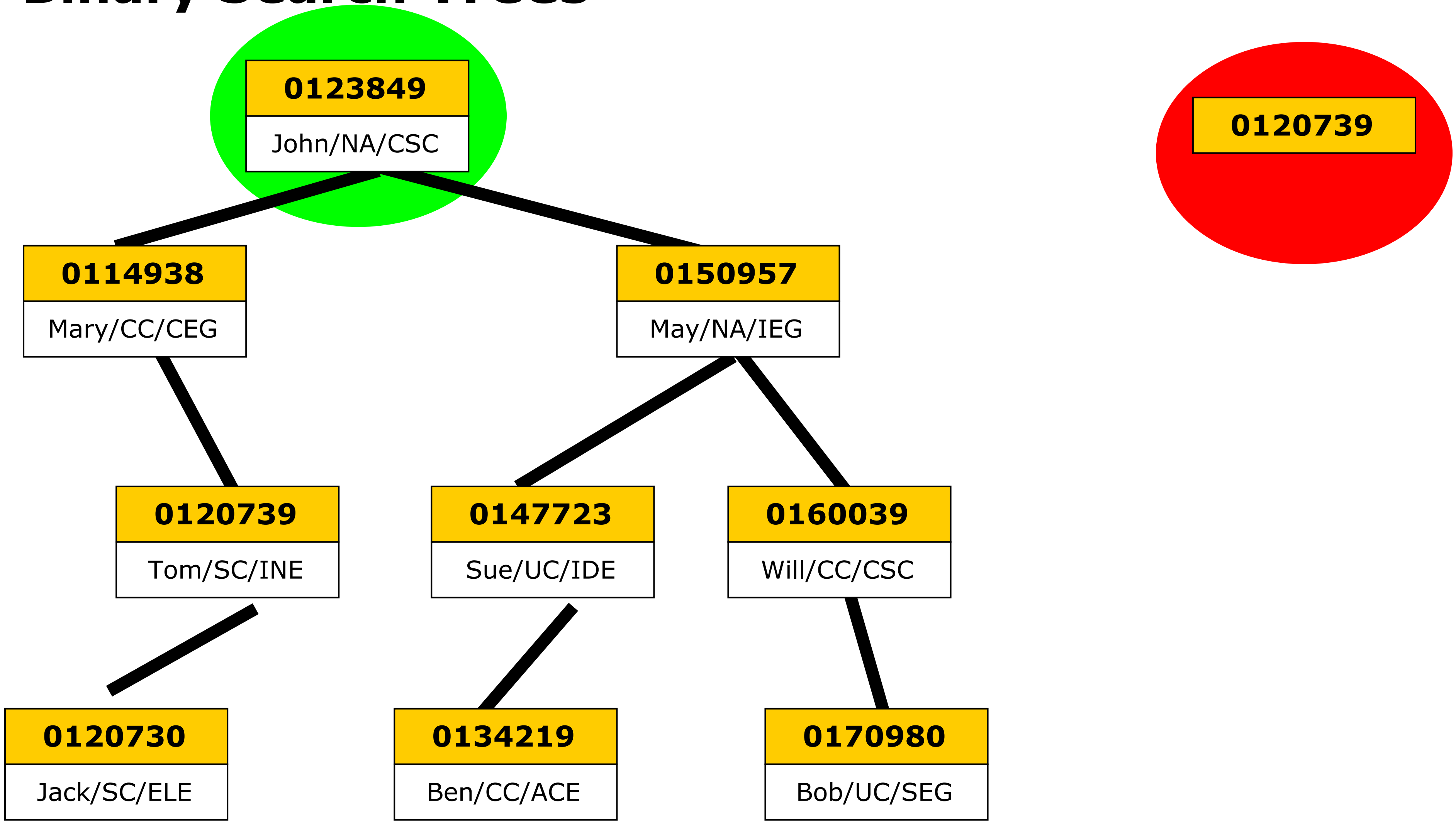
Binary Search Trees



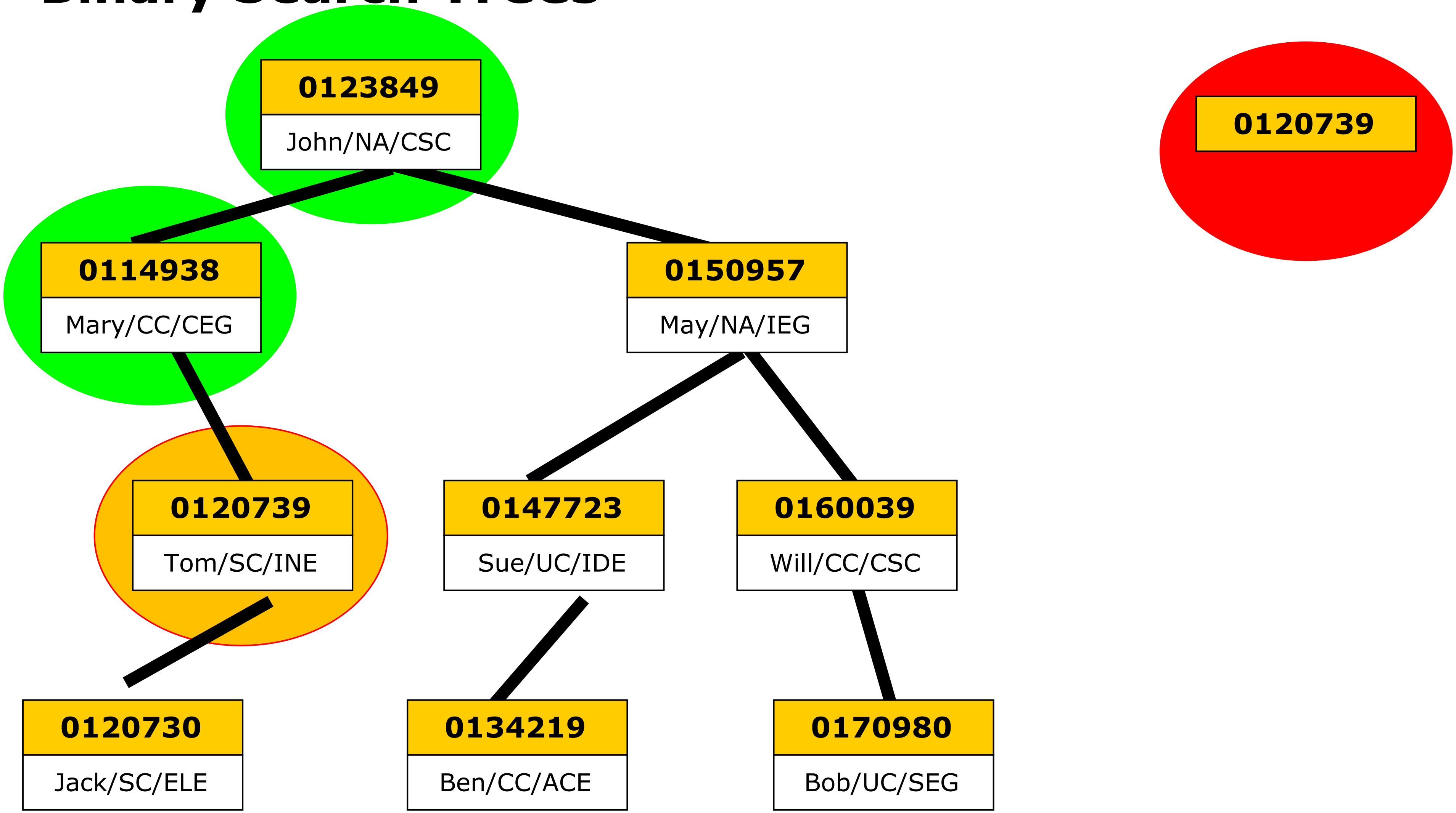
Binary Search Trees



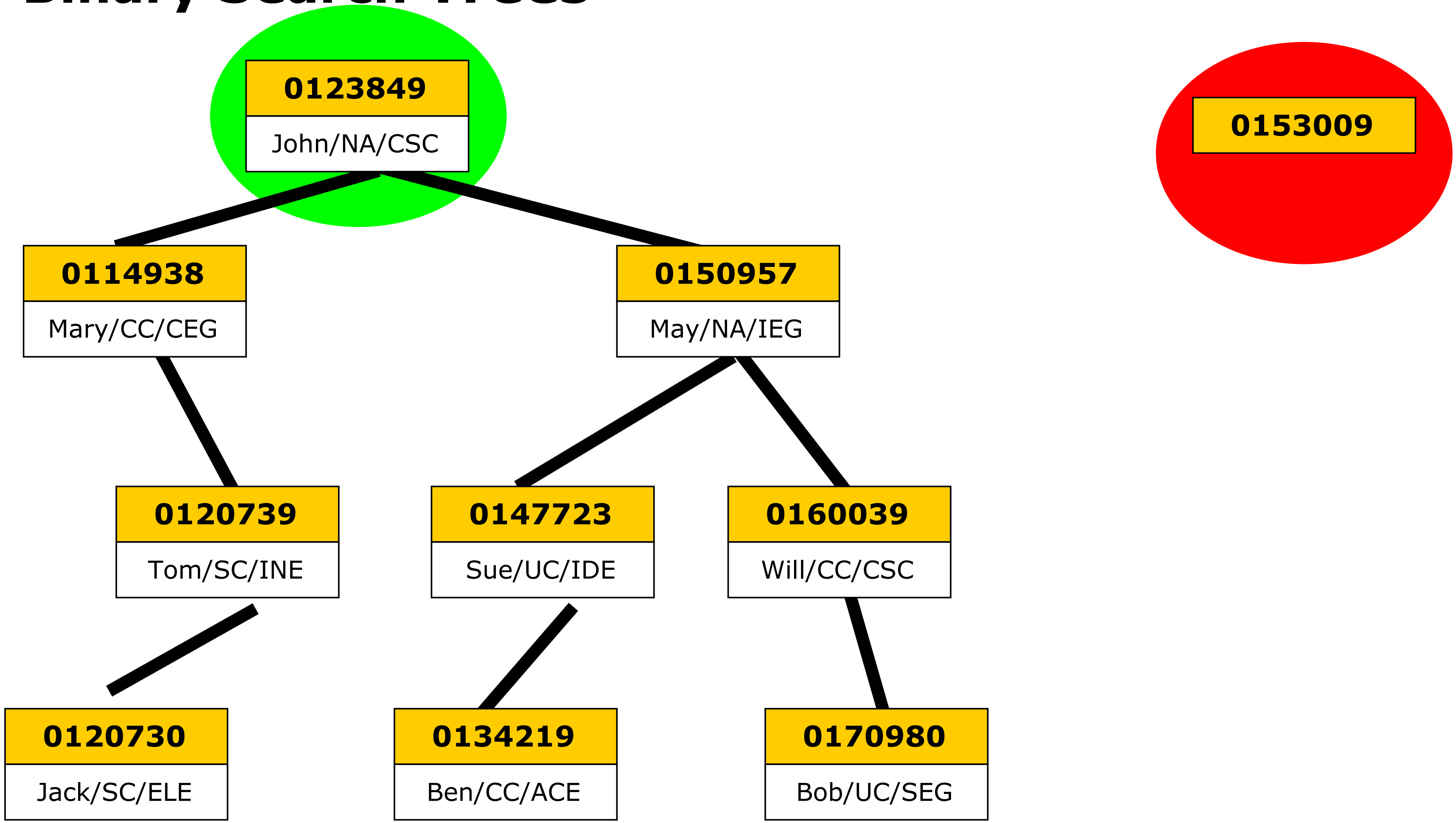
Binary Search Trees



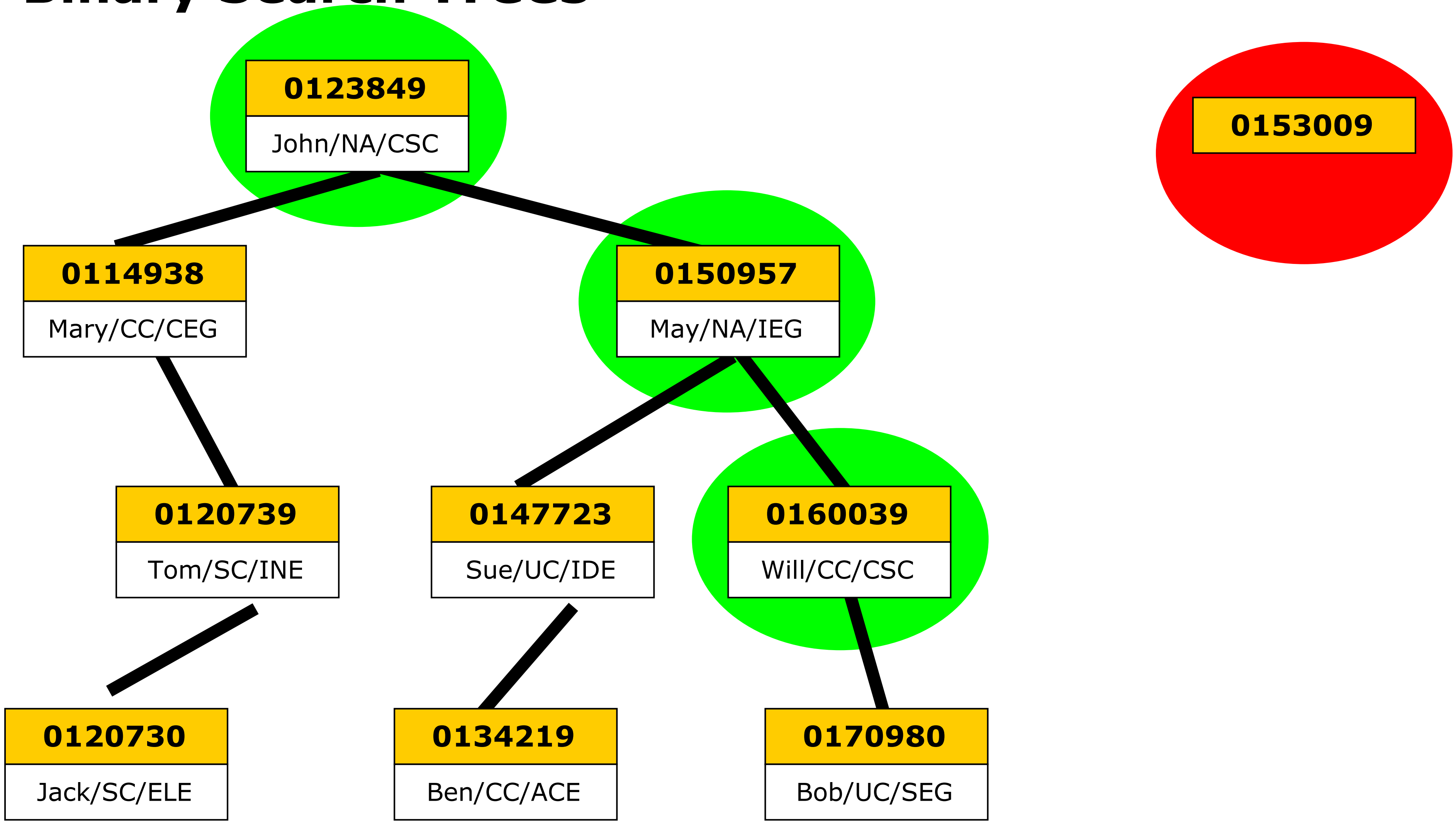
Binary Search Trees



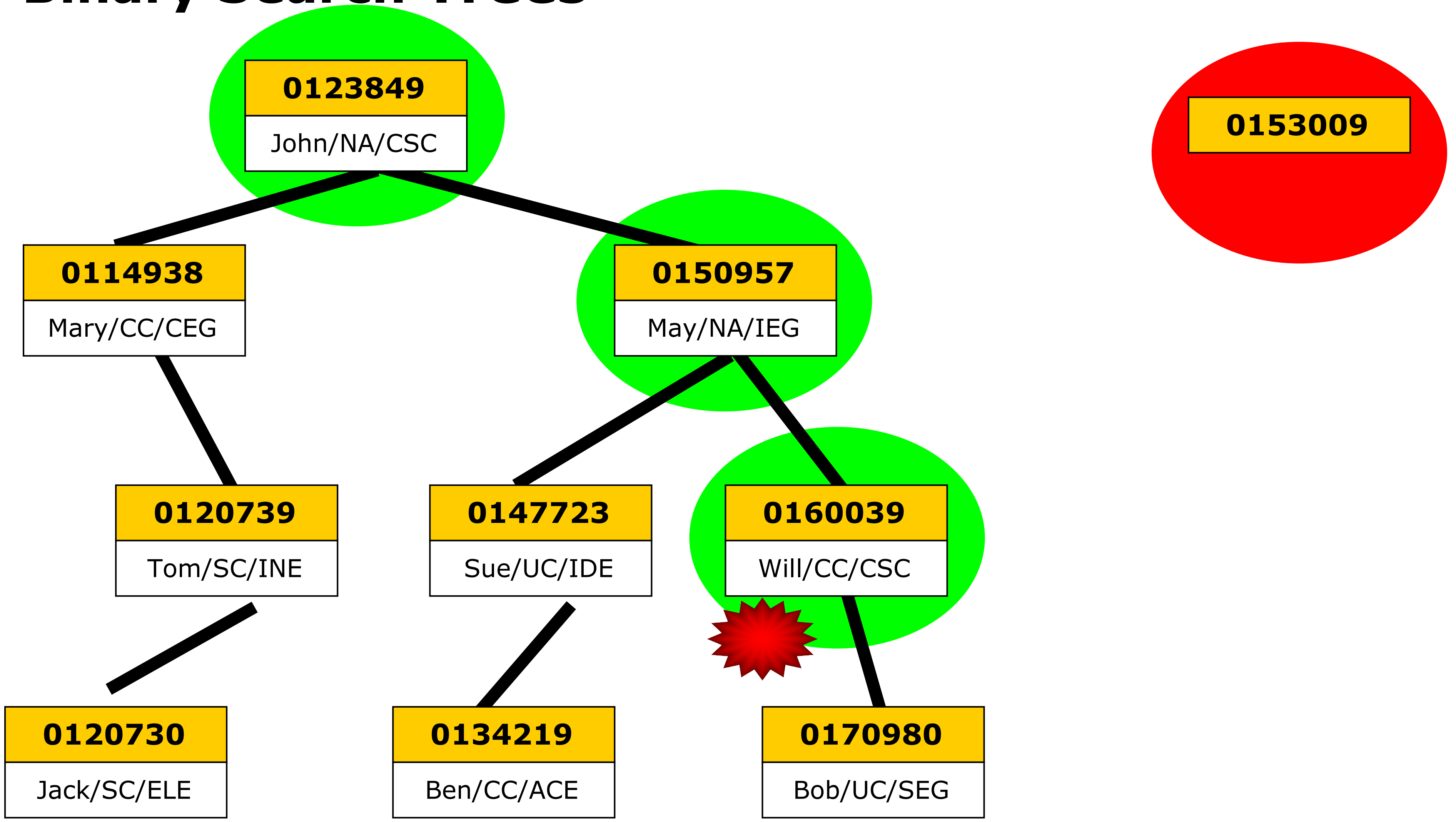
Binary Search Trees



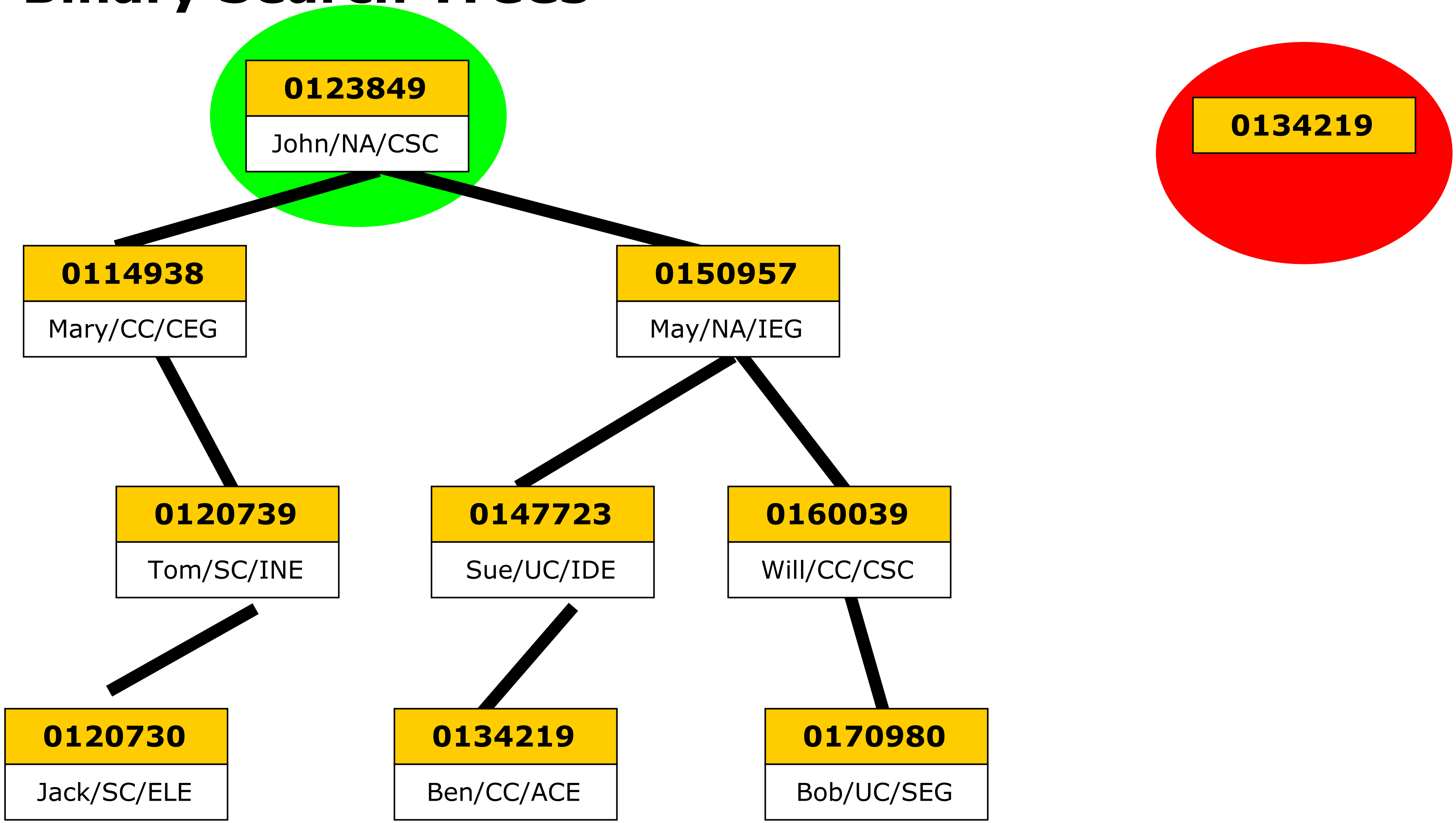
Binary Search Trees



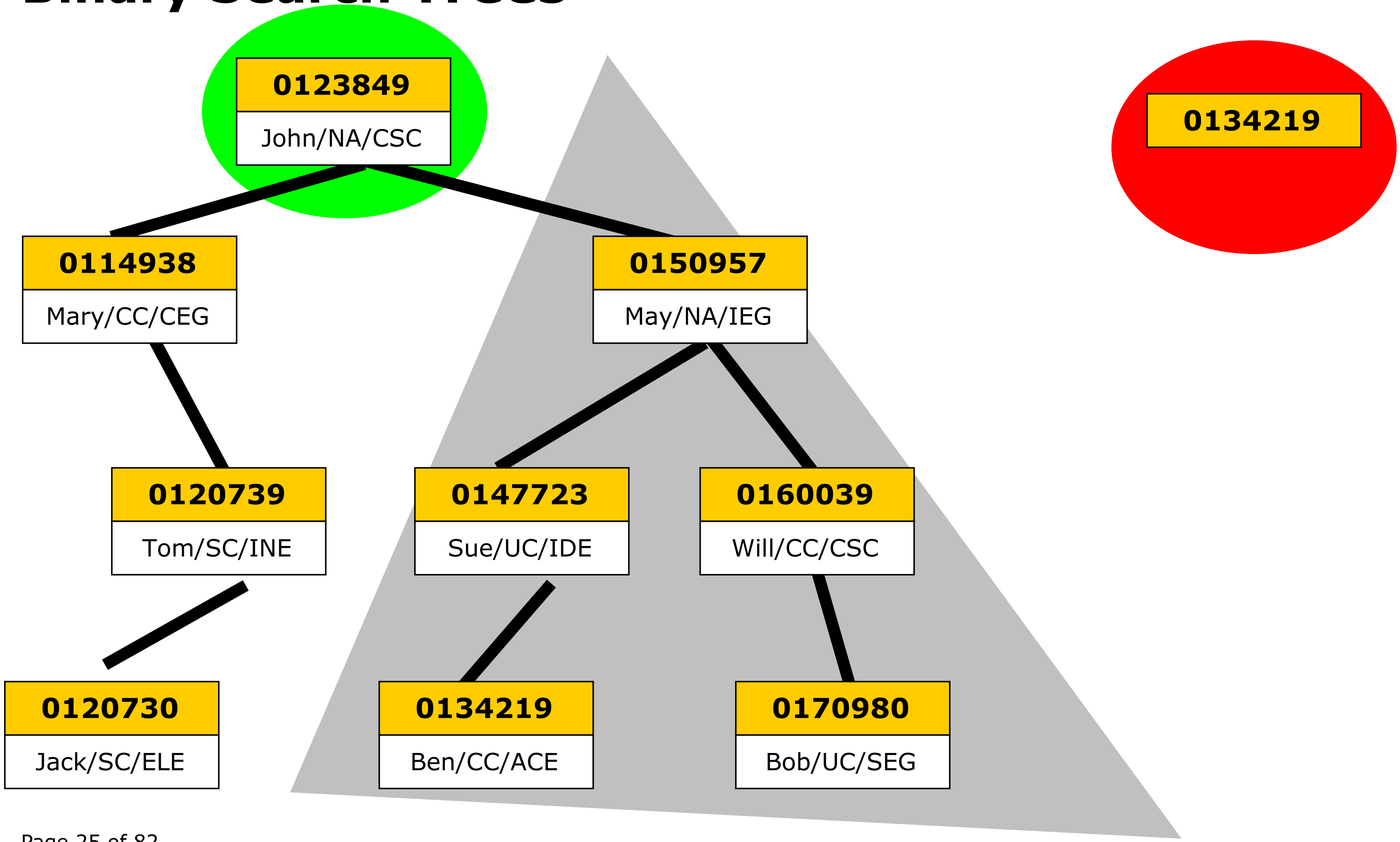
Binary Search Trees



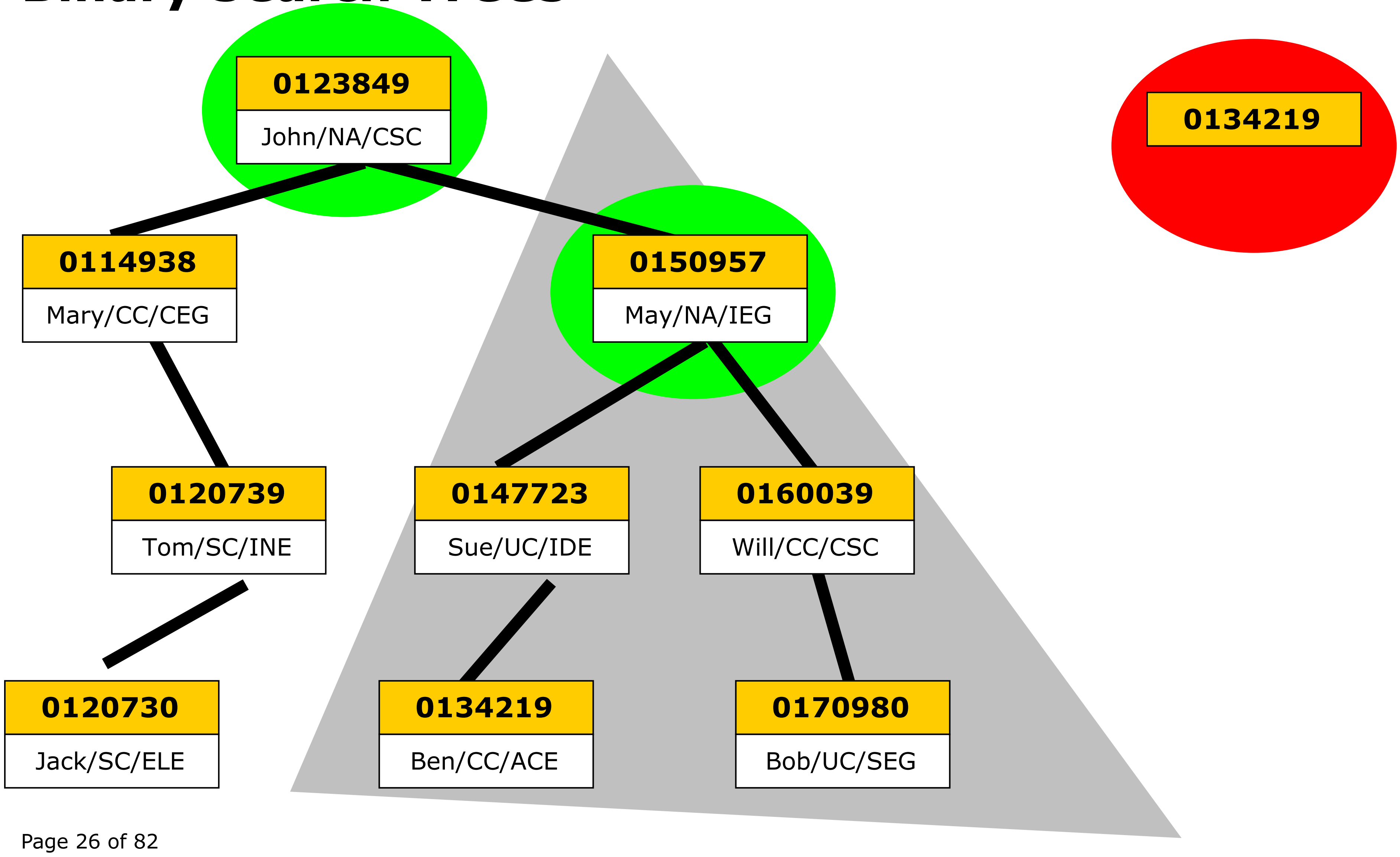
Binary Search Trees



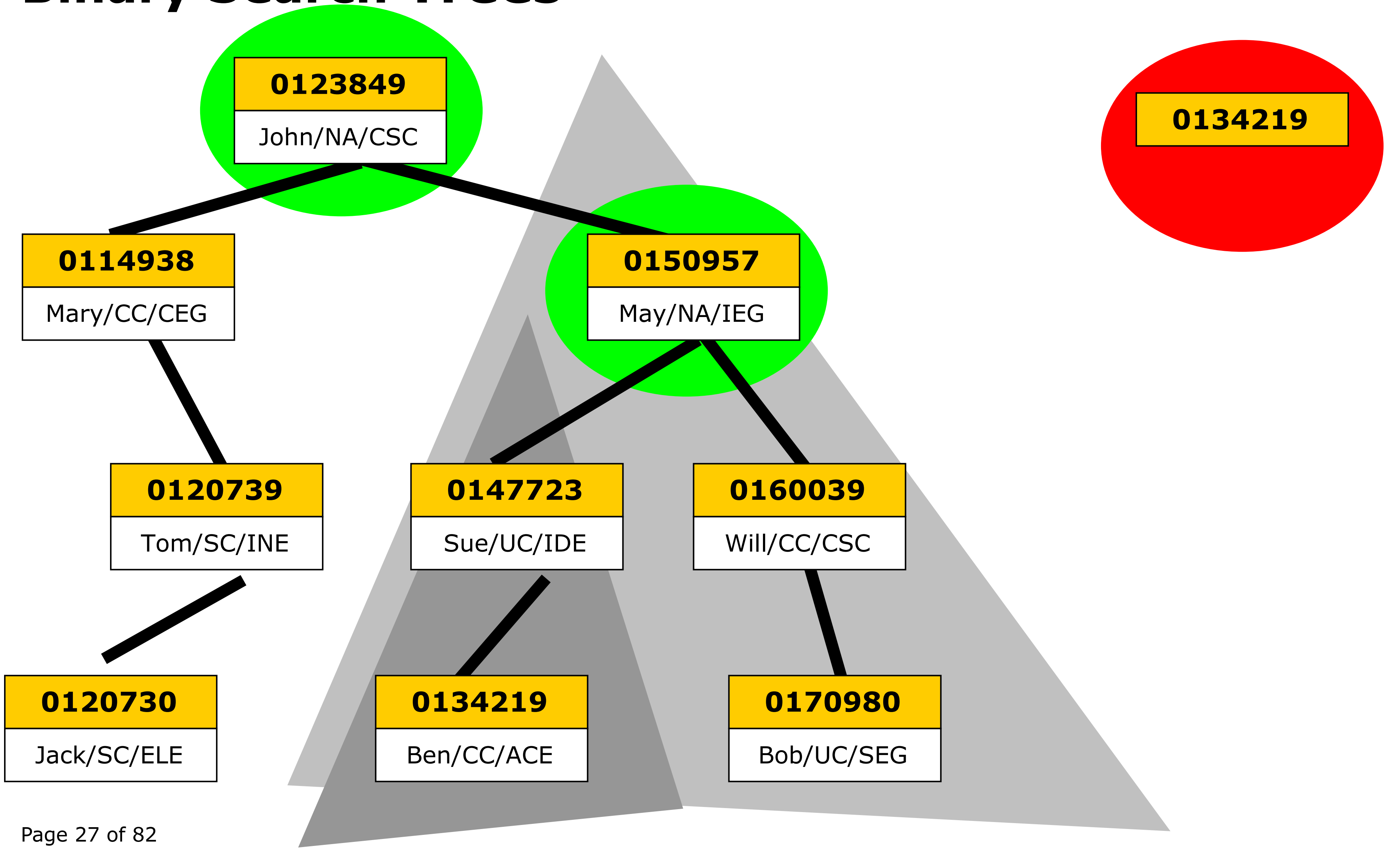
Binary Search Trees



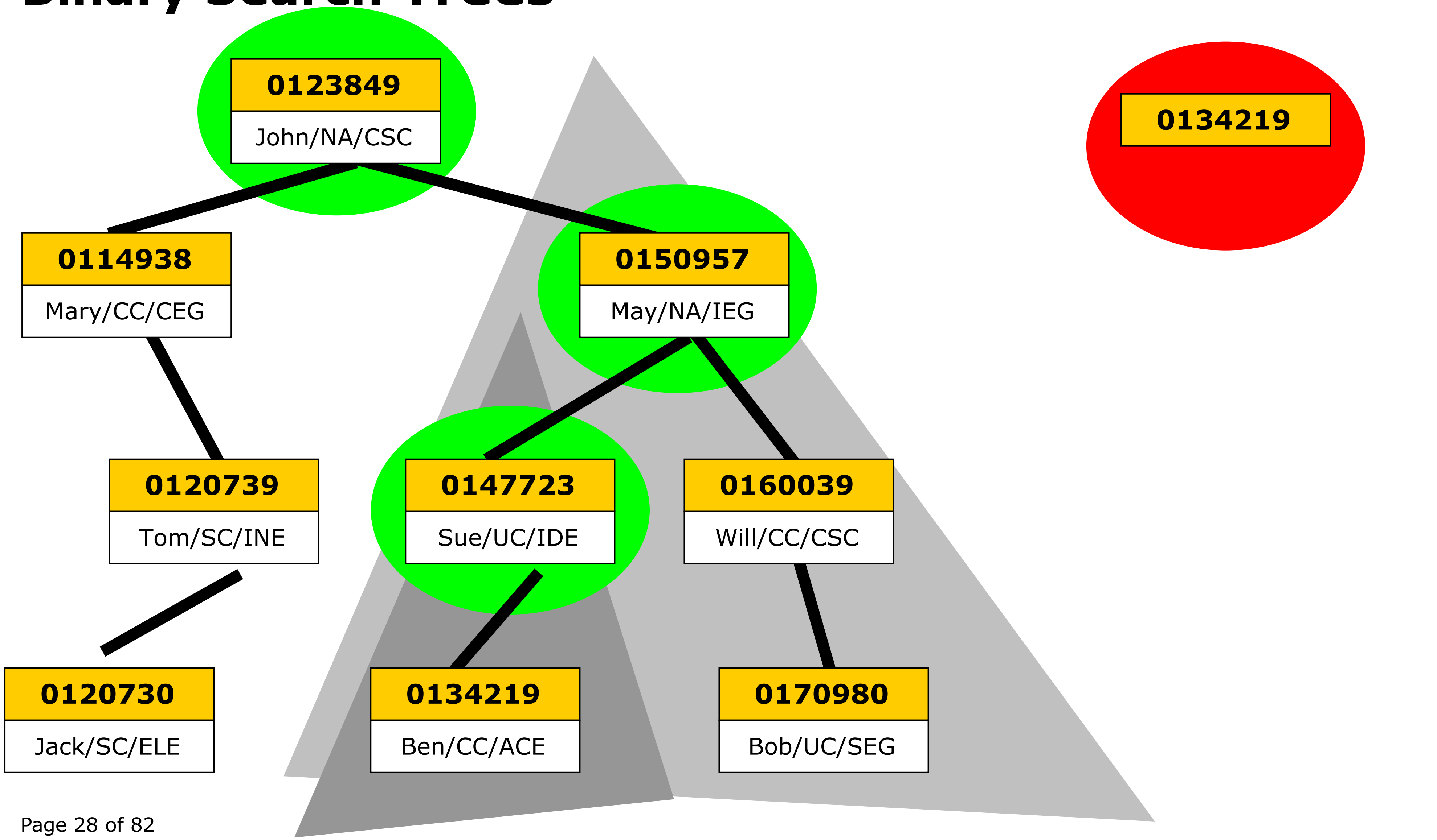
Binary Search Trees



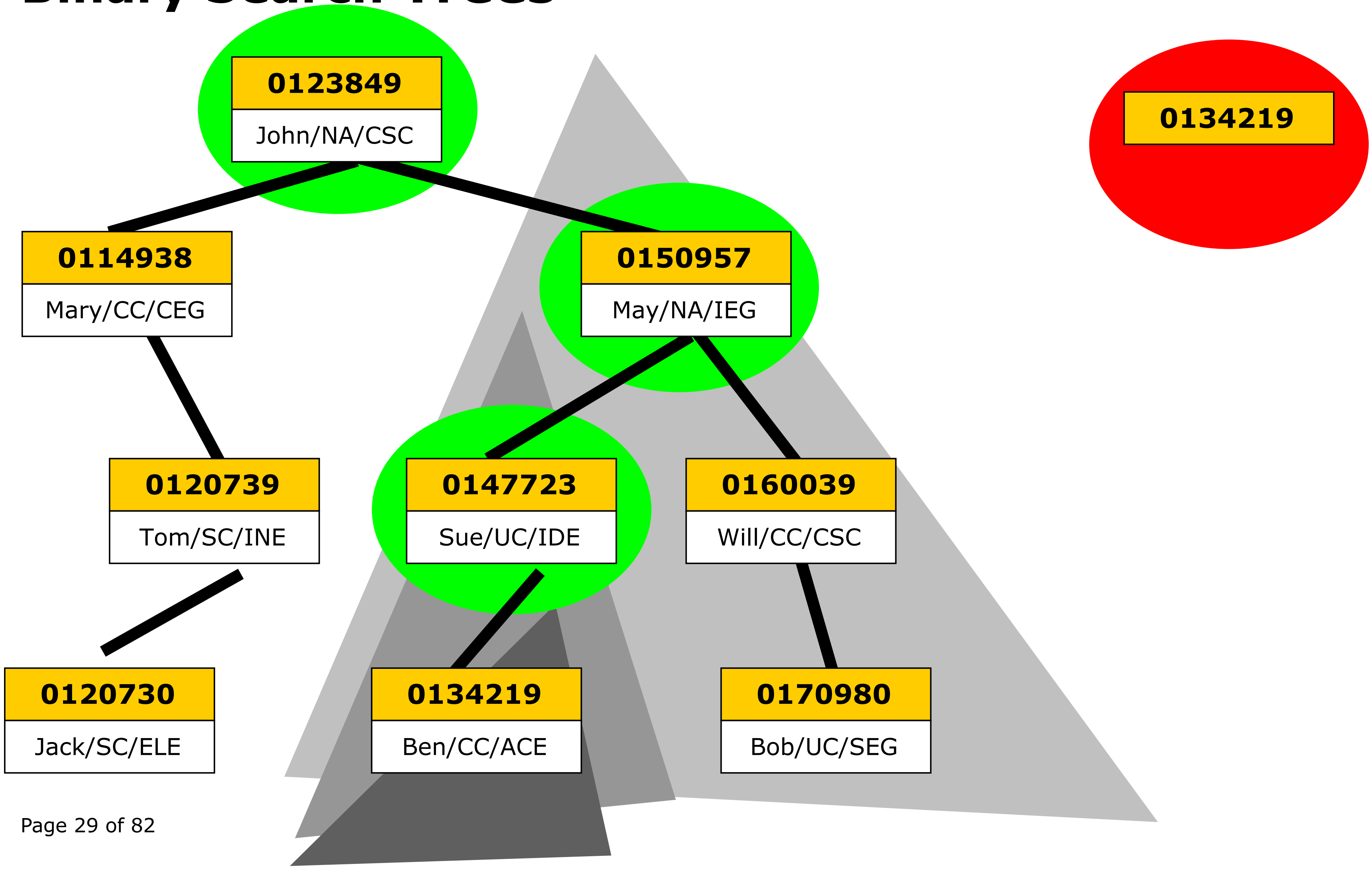
Binary Search Trees



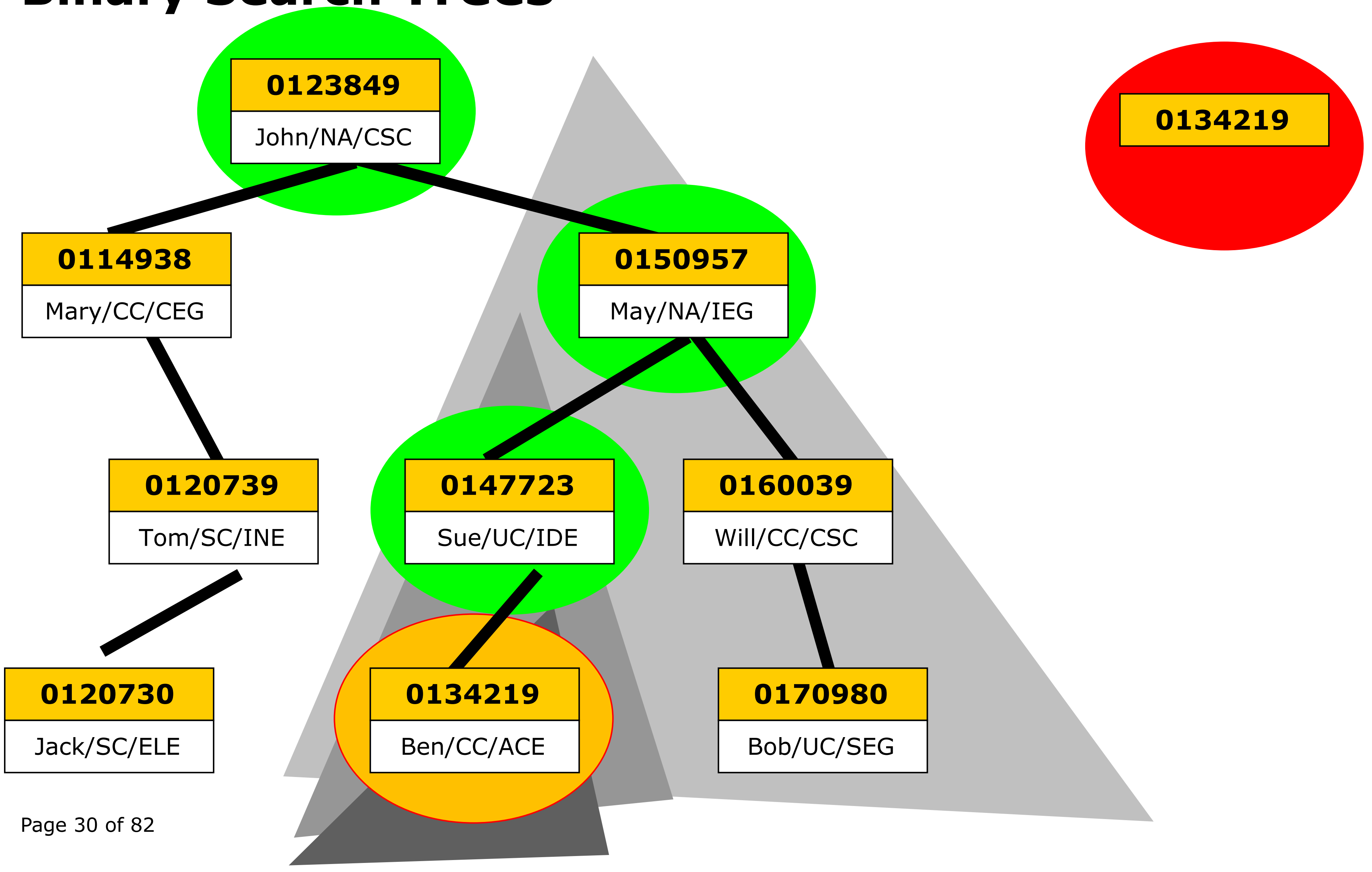
Binary Search Trees



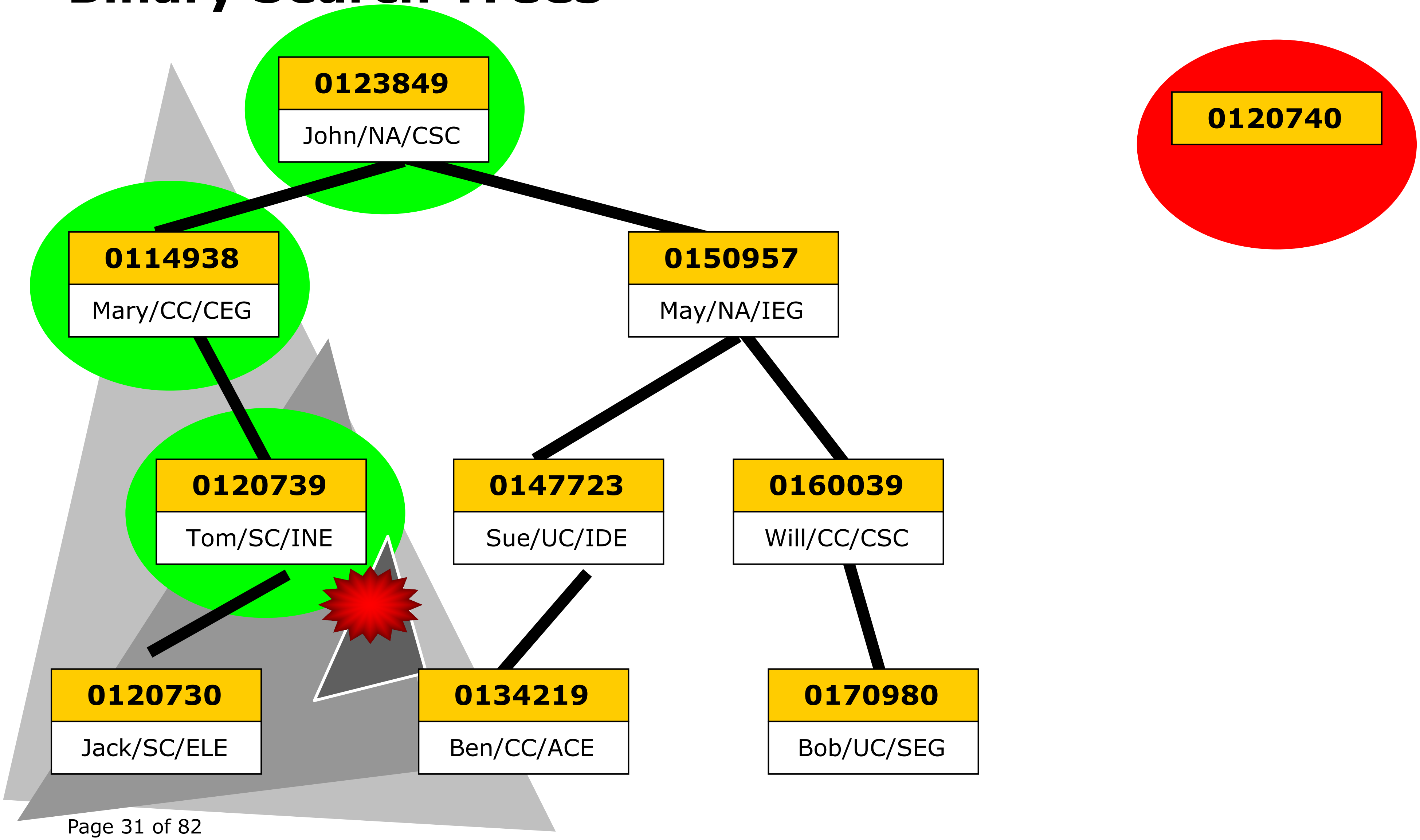
Binary Search Trees



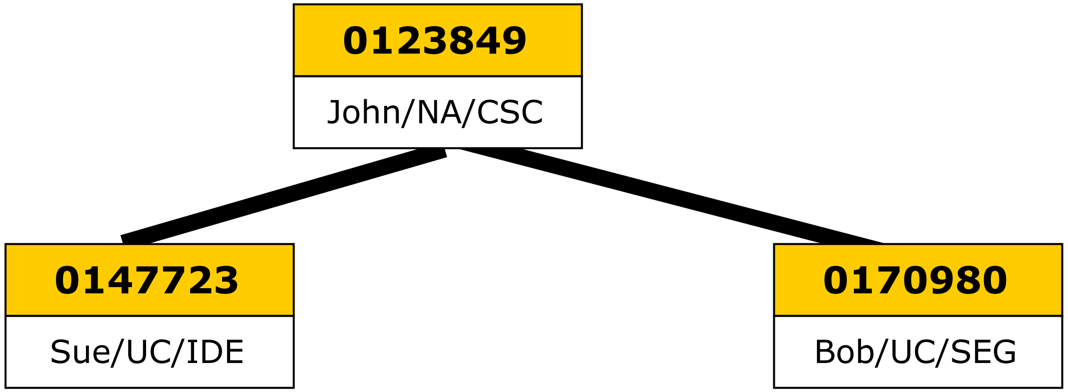
Binary Search Trees



Binary Search Trees

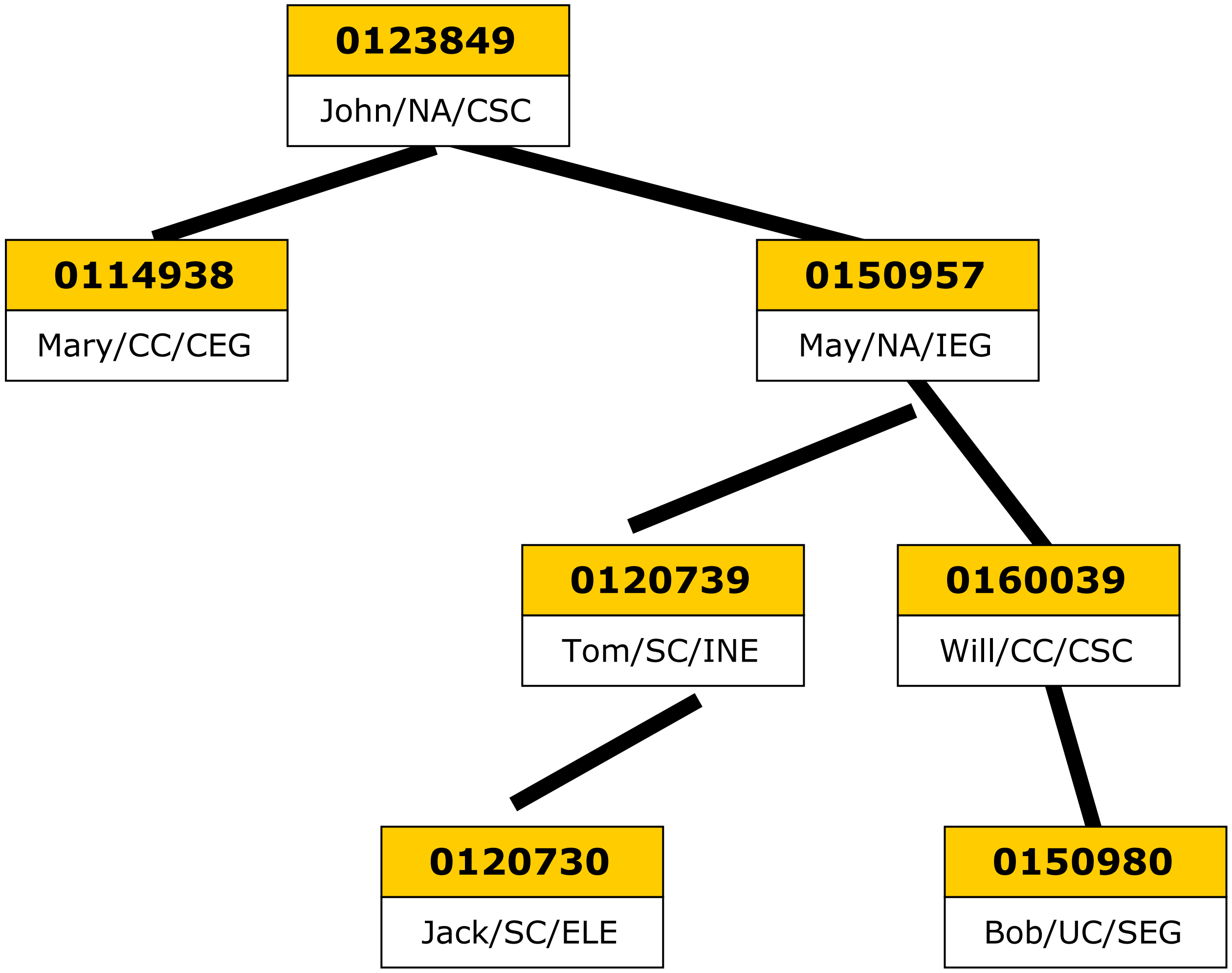


<u>Is This a Binary Search Tree?</u>



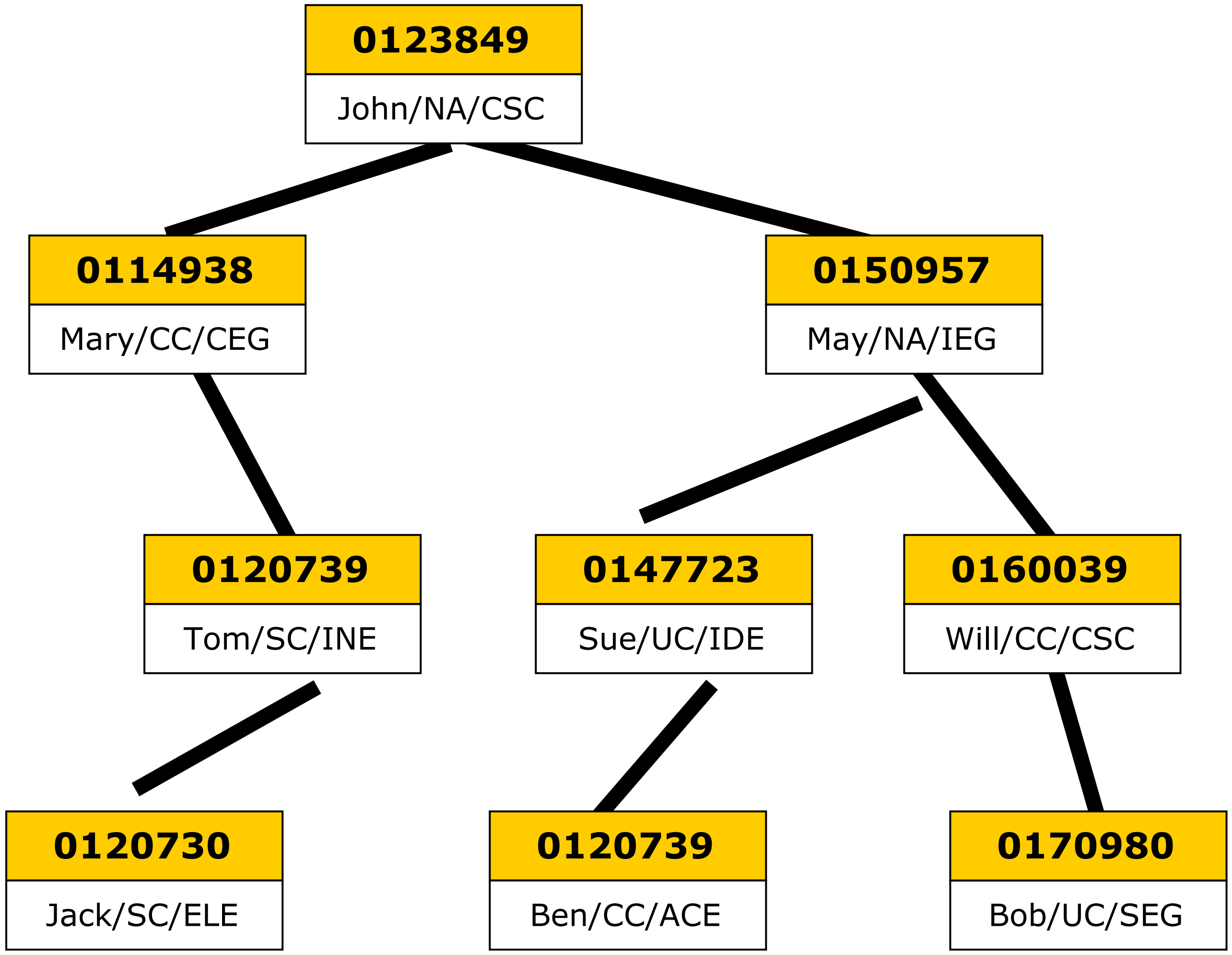
- Key values are unique.
- At every node, the key value must be *greater than* <u>all</u> the keys in the left subtree, and *less than* <u>all</u> the keys in the right subtree.

<u>Is This a Binary Search Tree?</u>



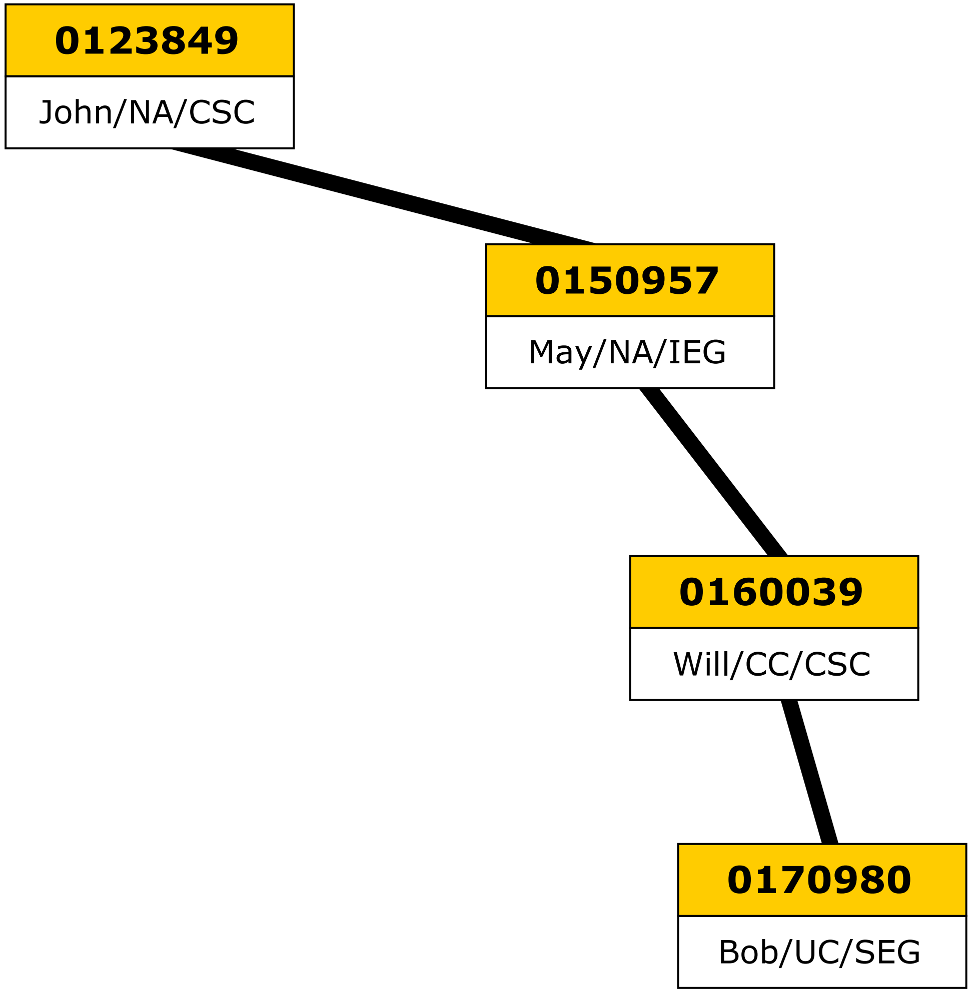
- Key values are unique.
- At every node, the key value must be *greater than* <u>all</u> the keys in the left subtree, and *less than* <u>all</u> the keys in the right subtree.

Is This a Binary Search Tree?



- Key values are unique.
- At every node, the key value must be *greater than* all the keys in the left subtree, and *less than* all the keys in the right subtree.

Is This a Binary Search Tree?

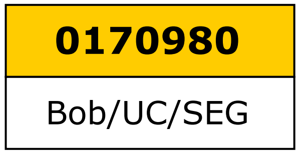


- Key values are unique.
- At every node, the key value must be *greater than* all the keys in the left subtree, and *less than* all the keys in the right subtree.

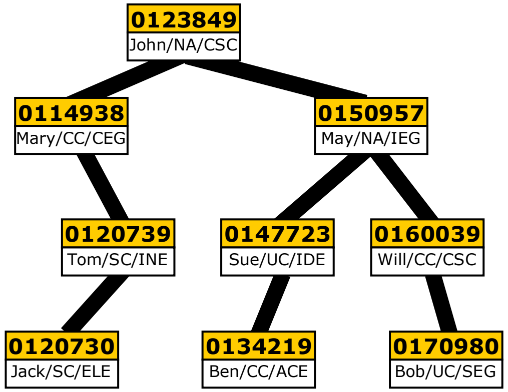
Tree Node ADT and Binary Tree ADT

We define two ADT's:

TreeNode



BinaryTree



```c
/* File: BinaryTree.h */
#include <stdlib.h>

typedef struct BinaryTreeCDT *BinaryTreeADT;
typedef struct TreeNodeCDT *TreeNodeADT;
#define SpecialErrNode (TreeNodeADT) NULL

BinaryTreeADT NonemptyBinaryTree(TreeNodeADT,
   BinaryTreeADT, BinaryTreeADT);
BinaryTreeADT EmptyBinaryTree(void);
BinaryTreeADT LeftSubtree(BinaryTreeADT);
BinaryTreeADT RightSubtree(BinaryTreeADT);
int TreeIsEmpty(BinaryTreeADT);
TreeNodeADT Root(BinaryTreeADT);
char *GetNodeKey(TreeNodeADT);
```

Finding a Node in a Binary Search Tree

char string

```
TreeNodeADT FindNode(BinaryTreeADT t, char *key) {
   TreeNodeADT R; char *k; int sign;

   if (TreeIsEmpty(t)) return (SpecialErrNode);

   R = Root(t); k = GetNodeKey(R);
   sign = strcmp(key, k);
   if (sign == 0) return R;
   if (sign < 0) return FindNode(LeftSubtree(t), key);
   return FindNode(RightSubtree(t), key);
}
```

Note

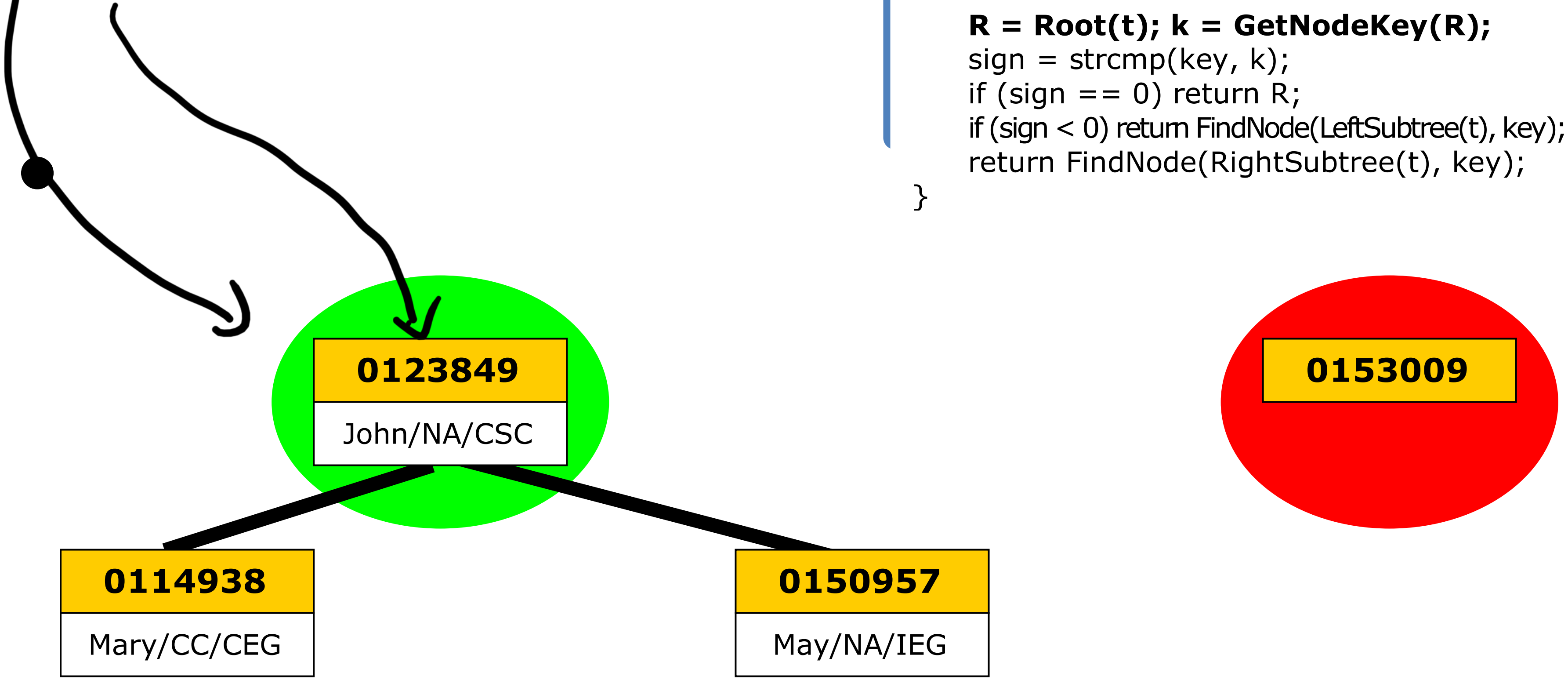
- **if (TreeIsEmpty(t))**
  return(SpecialErrNode);

```
TreeNodeADT FindNode(BinaryTreeADT t, char *key) {
    TreeNodeADT R; char *k; int sign;
    if (TreeIsEmpty(t)) return(SpecialErrNode);

    R = Root(t); k = GetNodeKey(R);
    sign = strcmp(key, k);
    if (sign == 0) return R;
    if (sign < 0) return FindNode(LeftSubtree(t), key);
    return FindNode(RightSubtree(t), key);
}
```

It is an error to search an empty tree. A **special error tree node** is returned to indicate that an error has occurred (No node in the tree is found to have this key).

```
R = Root(t);
k = GetNodeKey(R);
```

```
TreeNodeADT FindNode(BinaryTreeADT t, char *key) {
    TreeNodeADT R; char *k; int sign;
    if (TreeIsEmpty(t)) return(SpecialErrNode);

    R = Root(t); k = GetNodeKey(R);
    sign = strcmp(key, k);
    if (sign == 0) return R;
    if (sign < 0) return FindNode(LeftSubtree(t), key);
    return FindNode(RightSubtree(t), key);
}
```



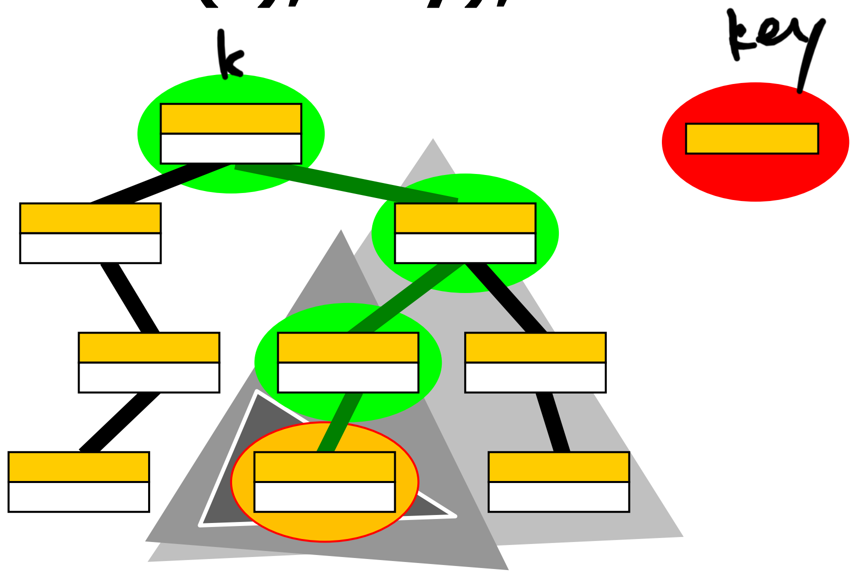
```
sign = strcmp(key, k);


if (sign == 0) return R;
if (sign < 0)
   return FindNode(LeftSubtree(t), key);
return FindNode(RightSubtree(t), key);
```

```
TreeNodeADT FindNode(BinaryTreeADT t, char *key) {
    TreeNodeADT R; char *k; int sign;
    if (TreeIsEmpty(t)) return(SpecialErrNode);

    R = Root(t); k = GetNodeKey(R);
    sign = strcmp(key, k);
    if (sign == 0) return R;
    if (sign < 0) return FindNode(LeftSubtree(t), key);
    return FindNode(RightSubtree(t), key);
}
```

The function strcmp returns −1 if **key** is before **k**, 0 if **key** is the same as **k**, and +1 if **key** is after **k**.

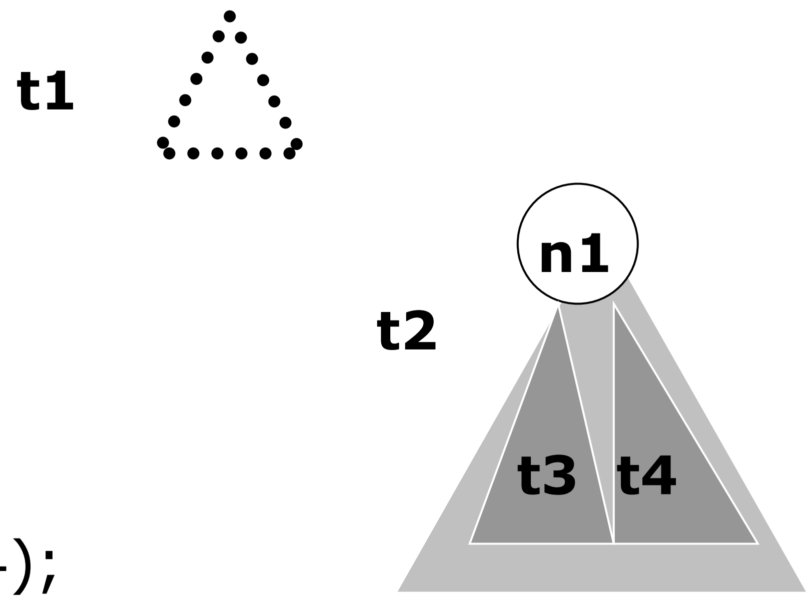
NOTE: #include <_____.h>

.

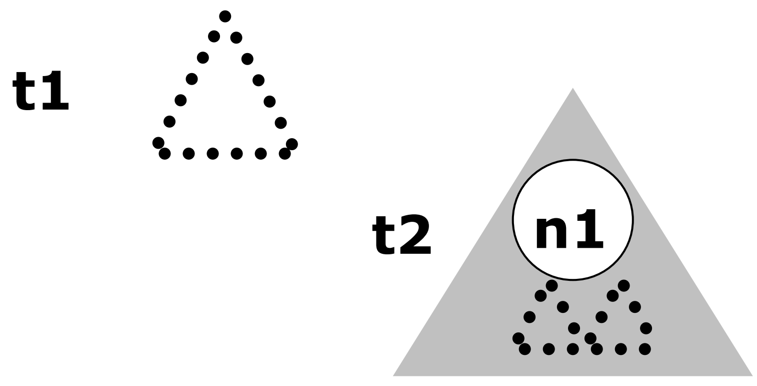


```
/* File: BinaryTree.h */
#include <stdlib.h>

typedef struct BinaryTreeCDT *BinaryTreeADT;
typedef struct TreeNodeCDT *TreeNodeADT;
#define SpecialErrNode (TreeNodeADT) NULL

BinaryTreeADT NonemptyBinaryTree(TreeNodeADT,
   BinaryTreeADT, BinaryTreeADT);
BinaryTreeADT EmptyBinaryTree(void);
BinaryTreeADT LeftSubtree(BinaryTreeADT);
BinaryTreeADT RightSubtree(BinaryTreeADT);
int TreeIsEmpty(BinaryTreeADT);
TreeNodeADT Root(BinaryTreeADT);
char *GetNodeKey(TreeNodeADT);
```

```
#include "BinaryTree.h"

TreeNodeADT n1;
BinaryTreeADT t1, t2, t3, t4;

…
t1 = EmptyBinaryTree();
…
t2 = NonemptyBinaryTree(n1, t3, t4);
```

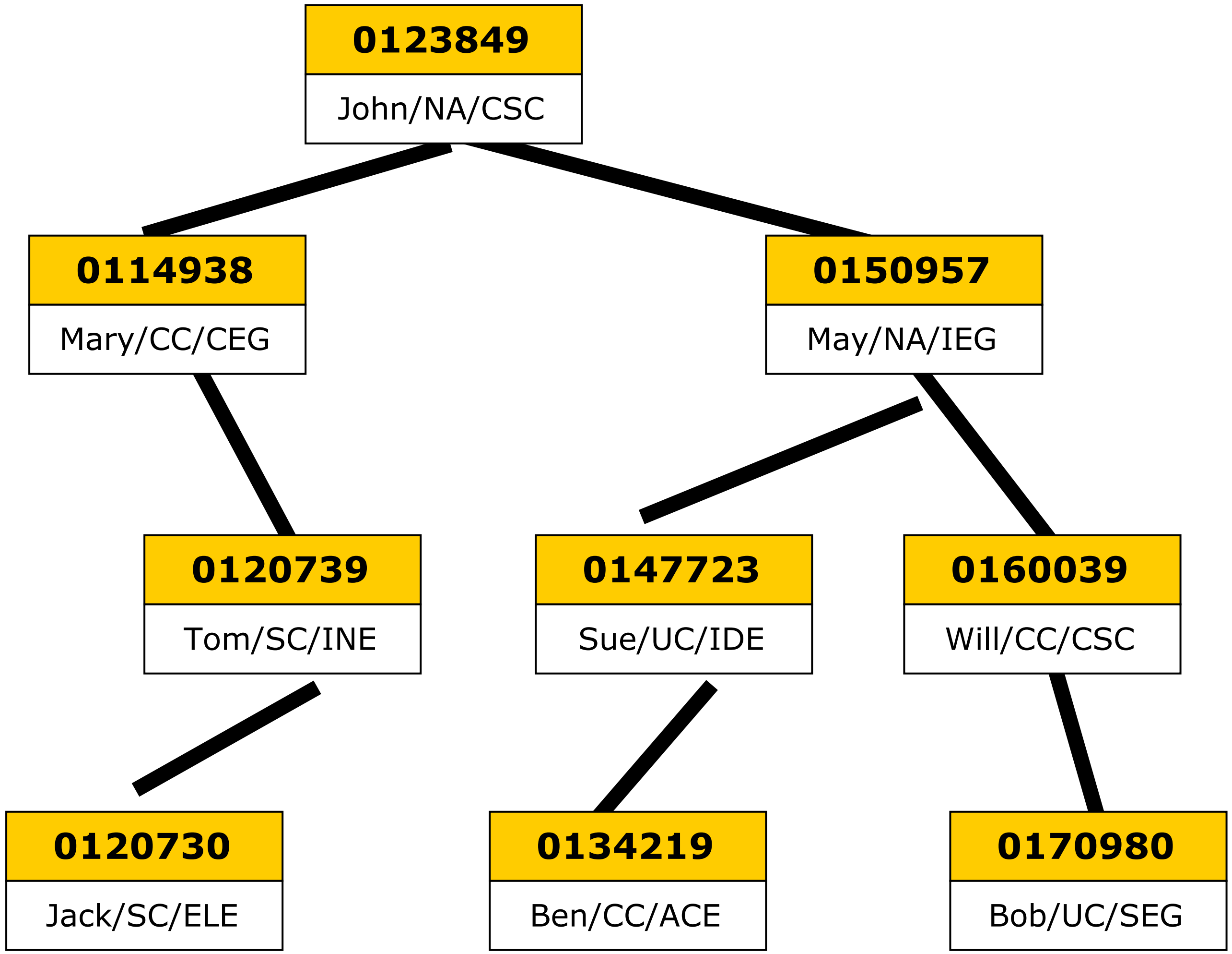


```
#include "BinaryTree.h"

TreeNodeADT n1;
BinaryTreeADT t1, t2;

...
t1 = EmptyBinaryTree();
t2 = NonemptyBinaryTree(n1,
          EmptyBinaryTree(), EmptyBinaryTree());
```



Coffee Break

Binary Search Trees



- Key values are unique.
- At every node, the key value must be *greater than* all the keys in the left subtree, and *less than* all the keys in the right subtree.

```
/* File: BinaryTree.h */
#include <stdlib.h>

typedef struct BinaryTreeCDT *BinaryTreeADT;
typedef struct TreeNodeCDT *TreeNodeADT;
#define SpecialErrNode (TreeNodeADT) NULL

BinaryTreeADT NonemptyBinaryTree(TreeNodeADT,
   BinaryTreeADT, BinaryTreeADT);
BinaryTreeADT EmptyBinaryTree(void);
BinaryTreeADT LeftSubtree(BinaryTreeADT);
BinaryTreeADT RightSubtree(BinaryTreeADT);
int TreeIsEmpty(BinaryTreeADT);
TreeNodeADT Root(BinaryTreeADT);
char *GetNodeKey(TreeNodeADT);
```

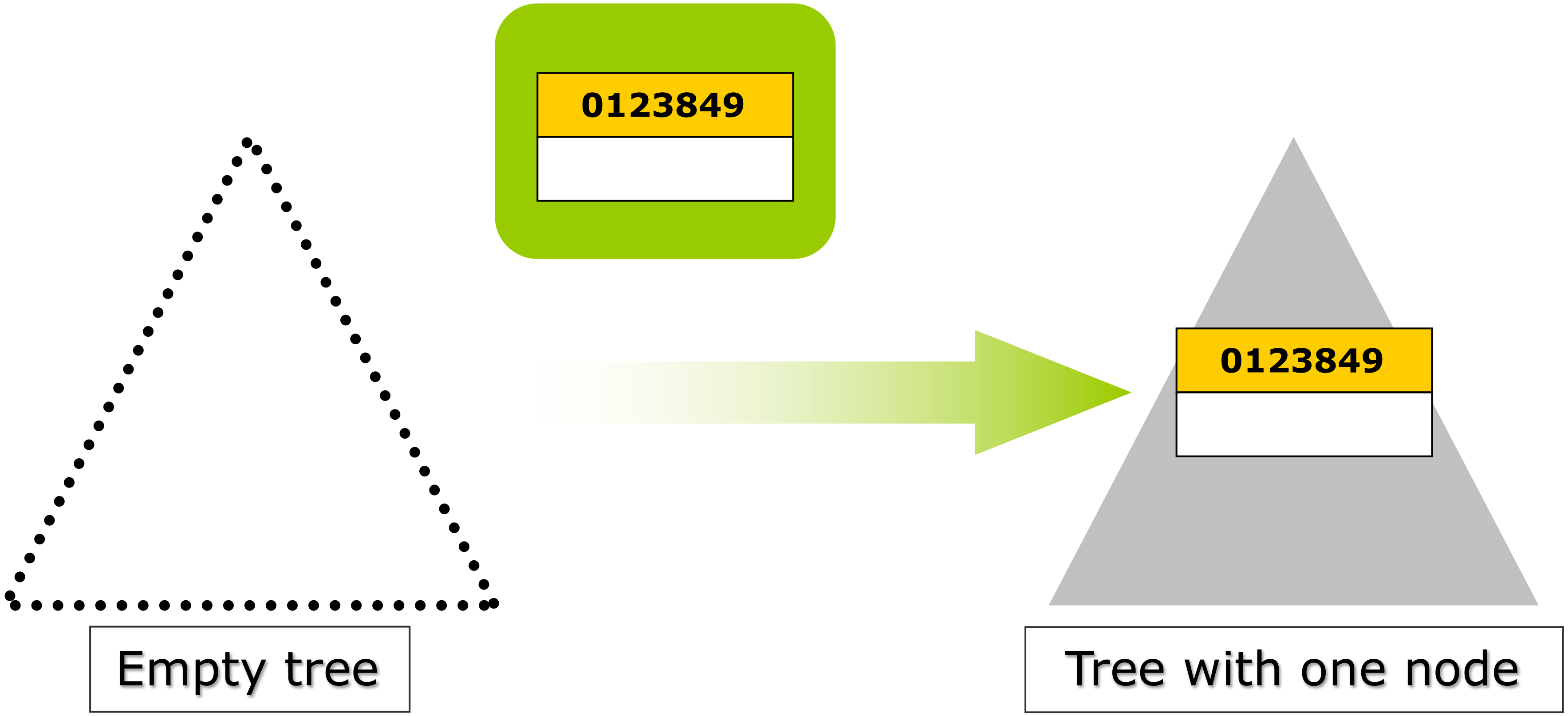
How are Binary Search Trees constructed?

We want to write a function

```
BinaryTreeADT InsertNode(BinaryTreeADT, TreeNodeADT);
```

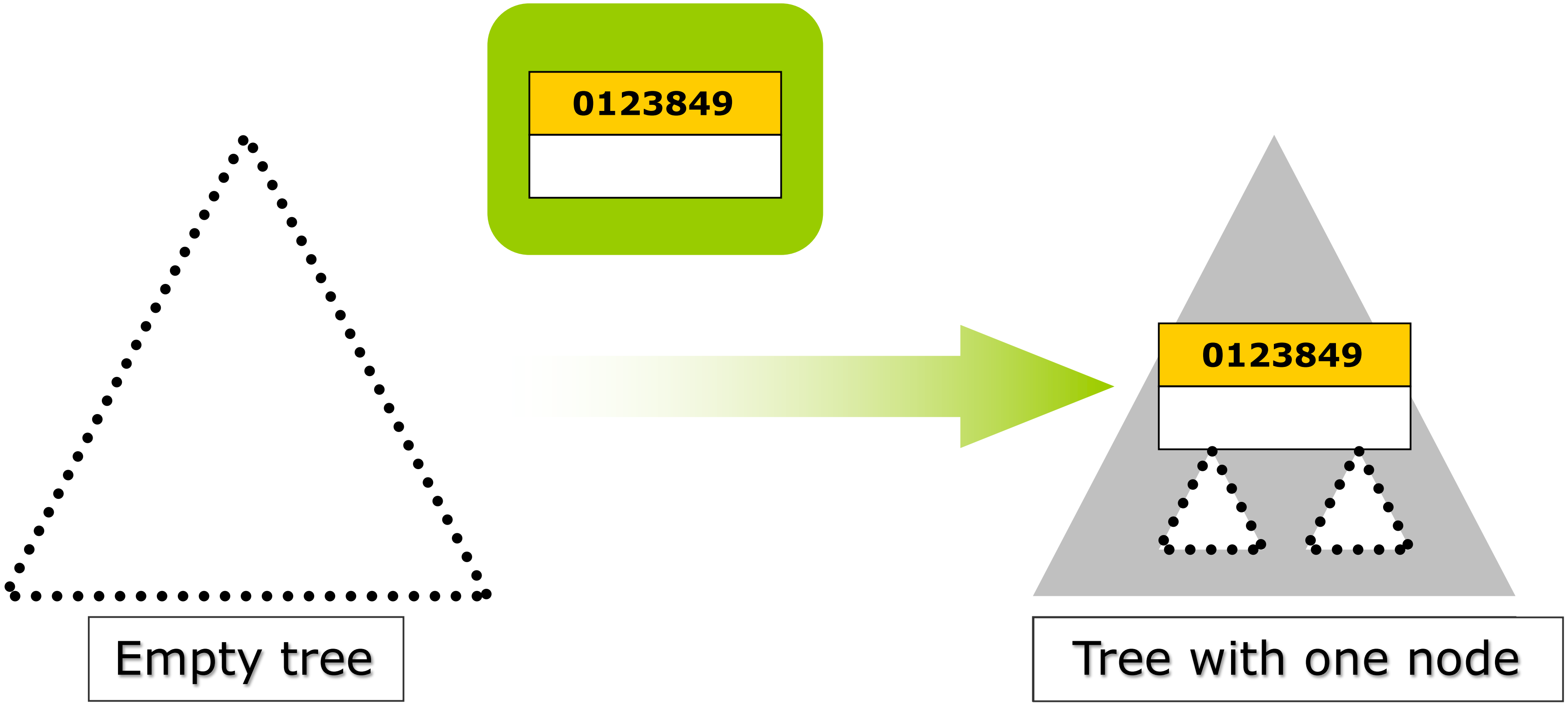
which builds a binary tree from tree nodes.

Constructing a Binary Search Tree



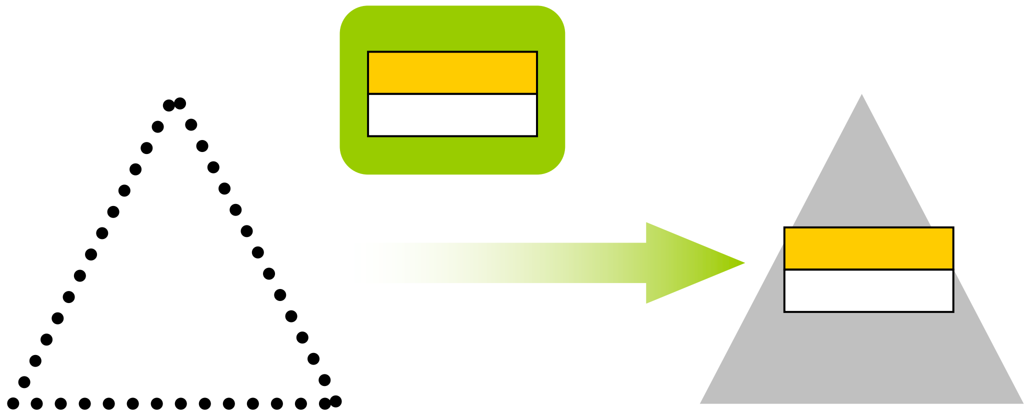
"Obtain a tree with a node inserted into an empty tree"

Constructing a Binary Search Tree



"Obtain a tree with a node inserted into an empty tree"

```
#include "BinaryTree.h"

...
BinaryTreeADT t1, t2;
TreeNodeADT n1;
```

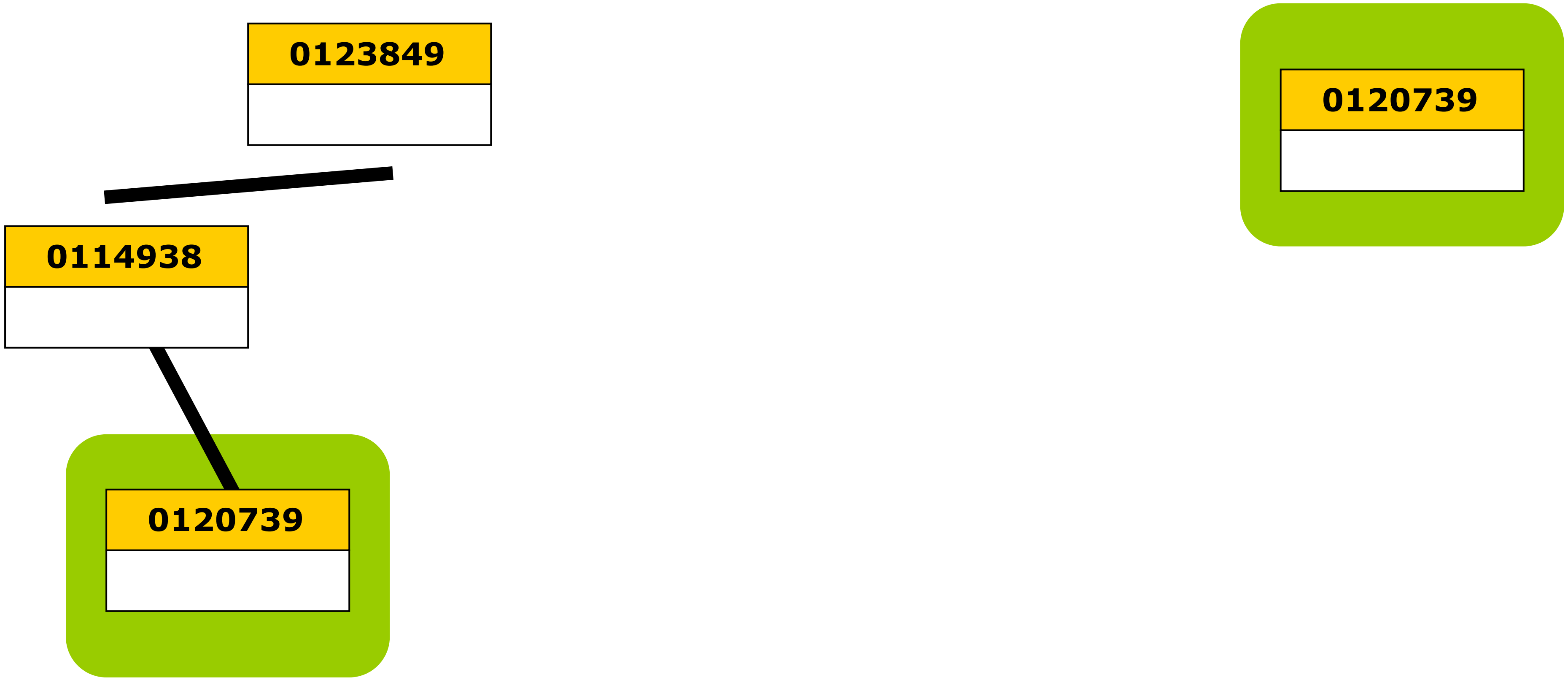


n1 = 

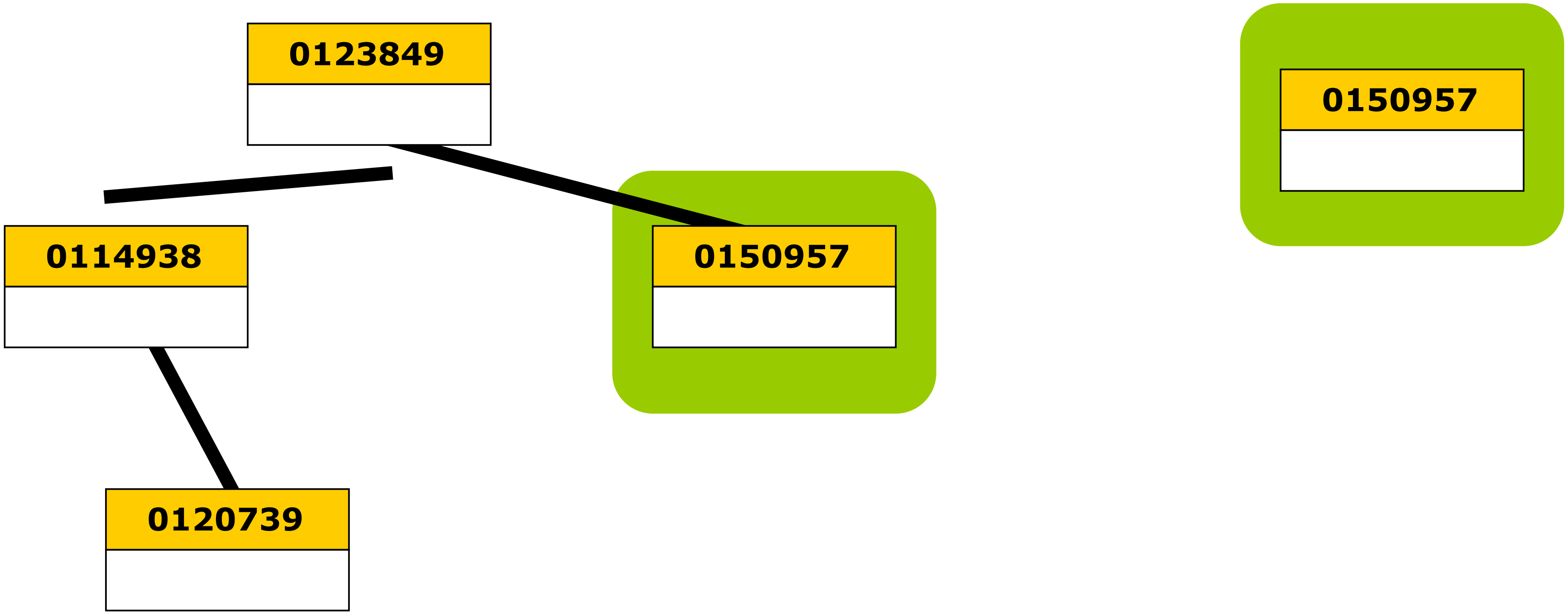
```
t1 = EmptyBinaryTree();
t2 = InsertNode(t1, n1);
...
```

Constructing a Binary Search Tree

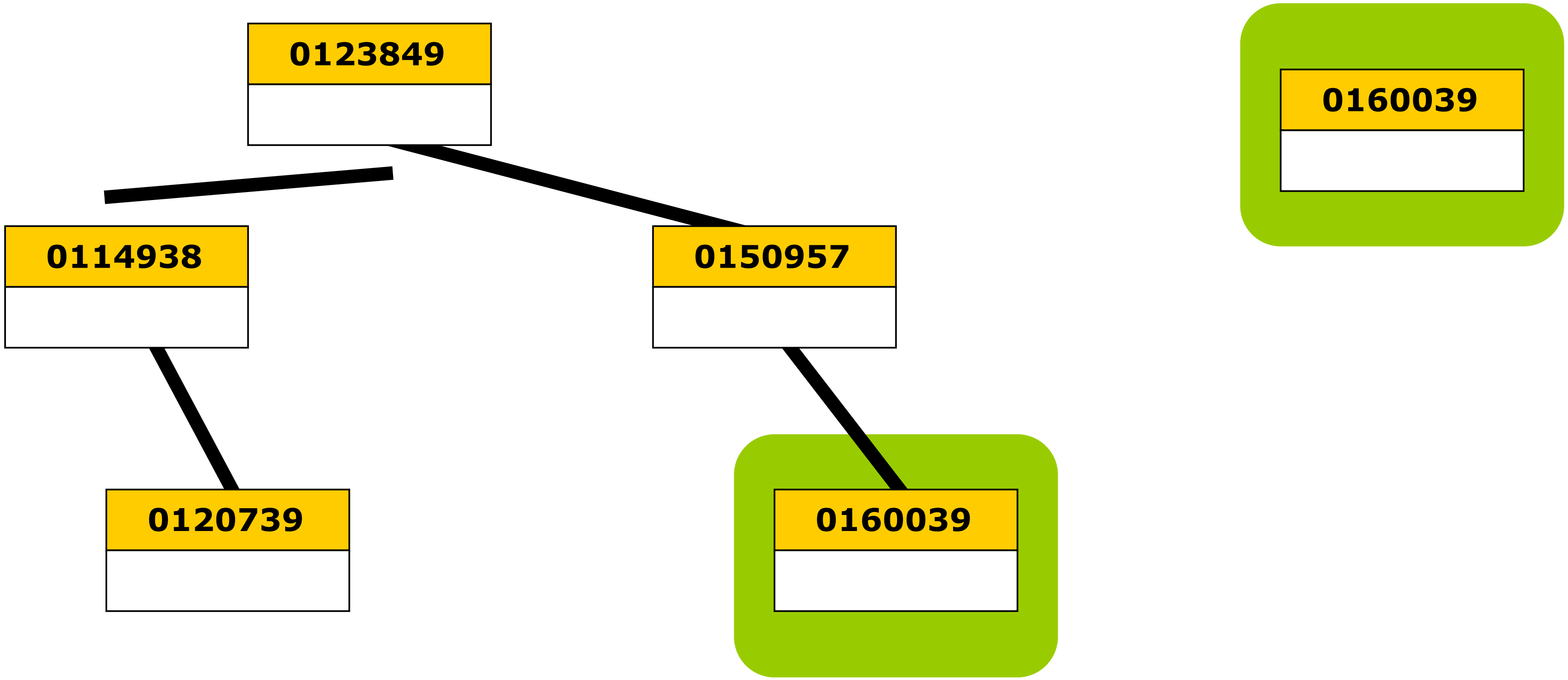
Constructing a Binary Search Tree



Constructing a Binary Search Tree



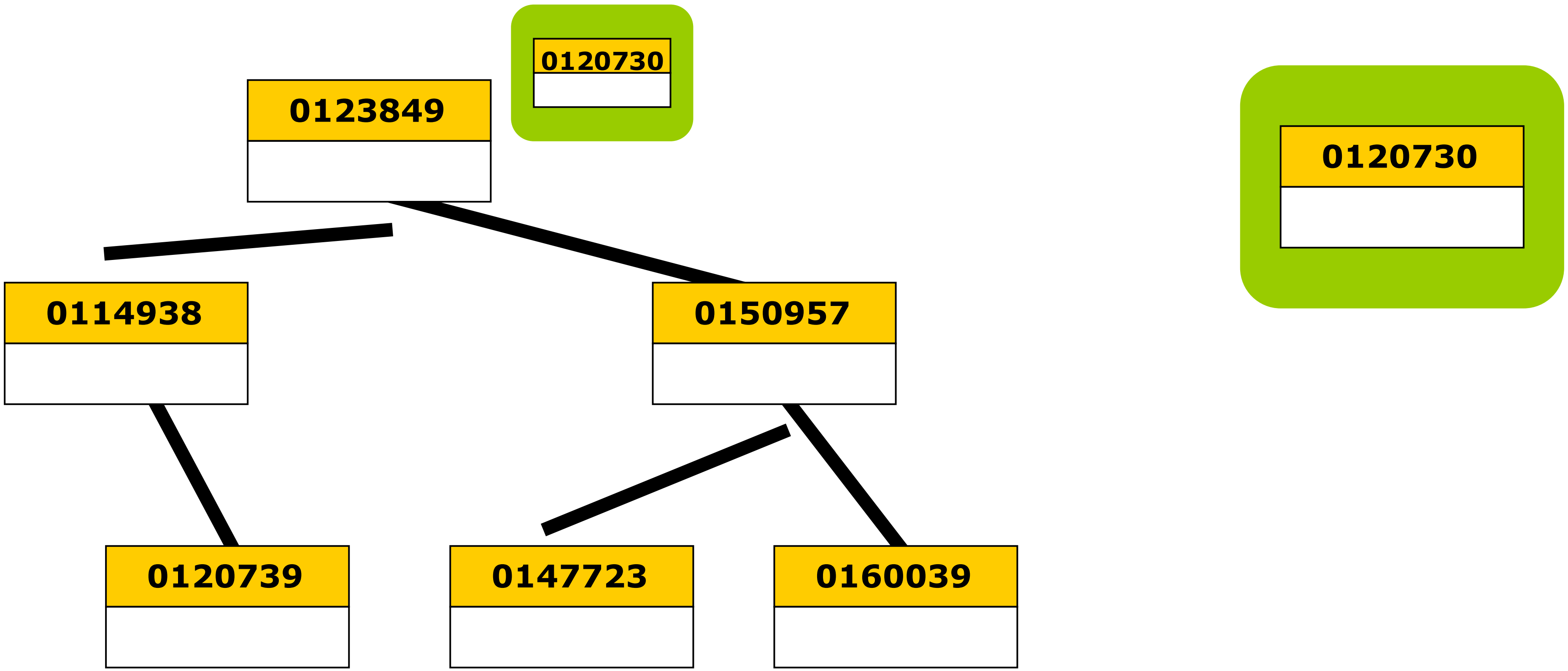
Constructing a Binary Search Tree



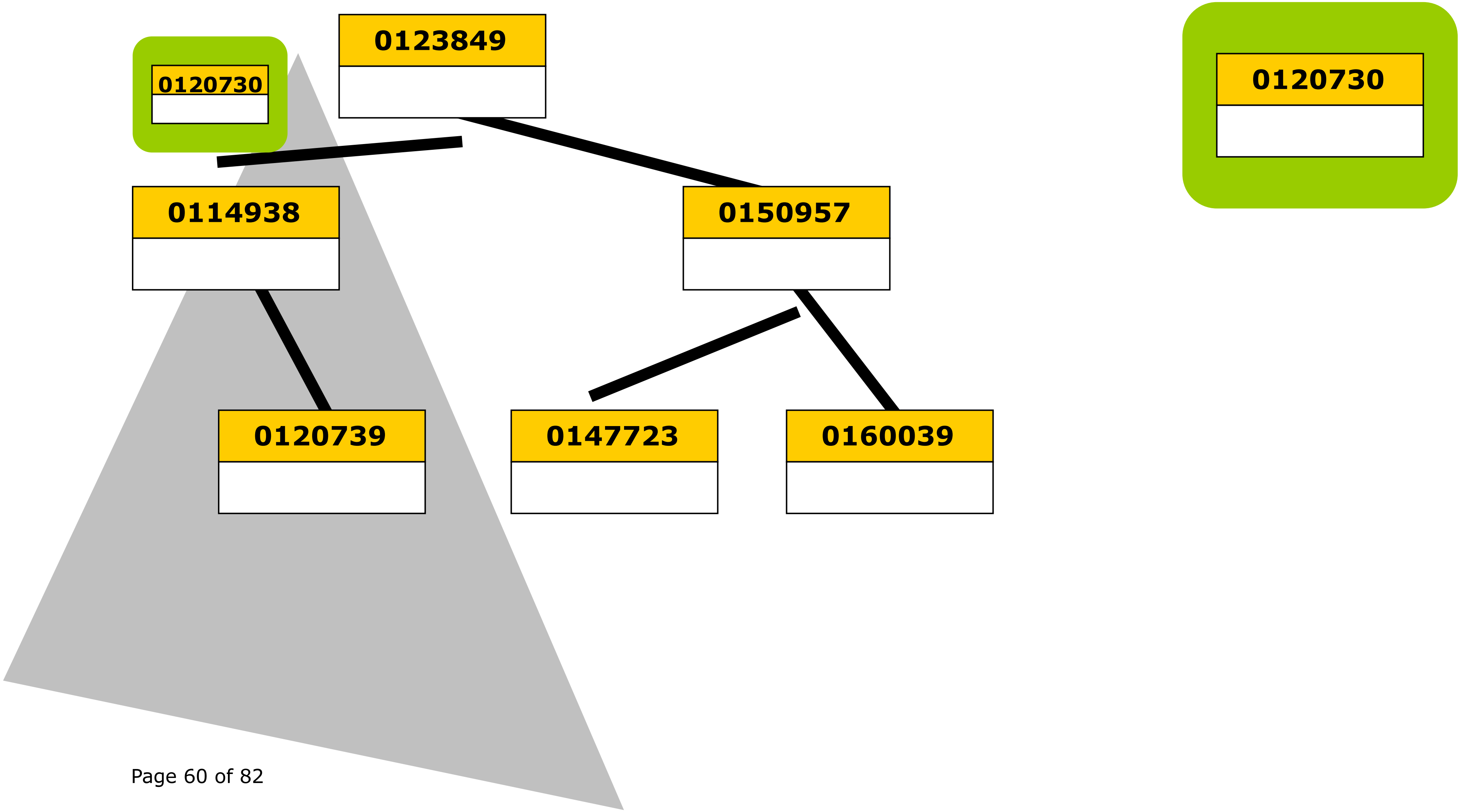
and so on …

Now, another viewpoint of node insertion.

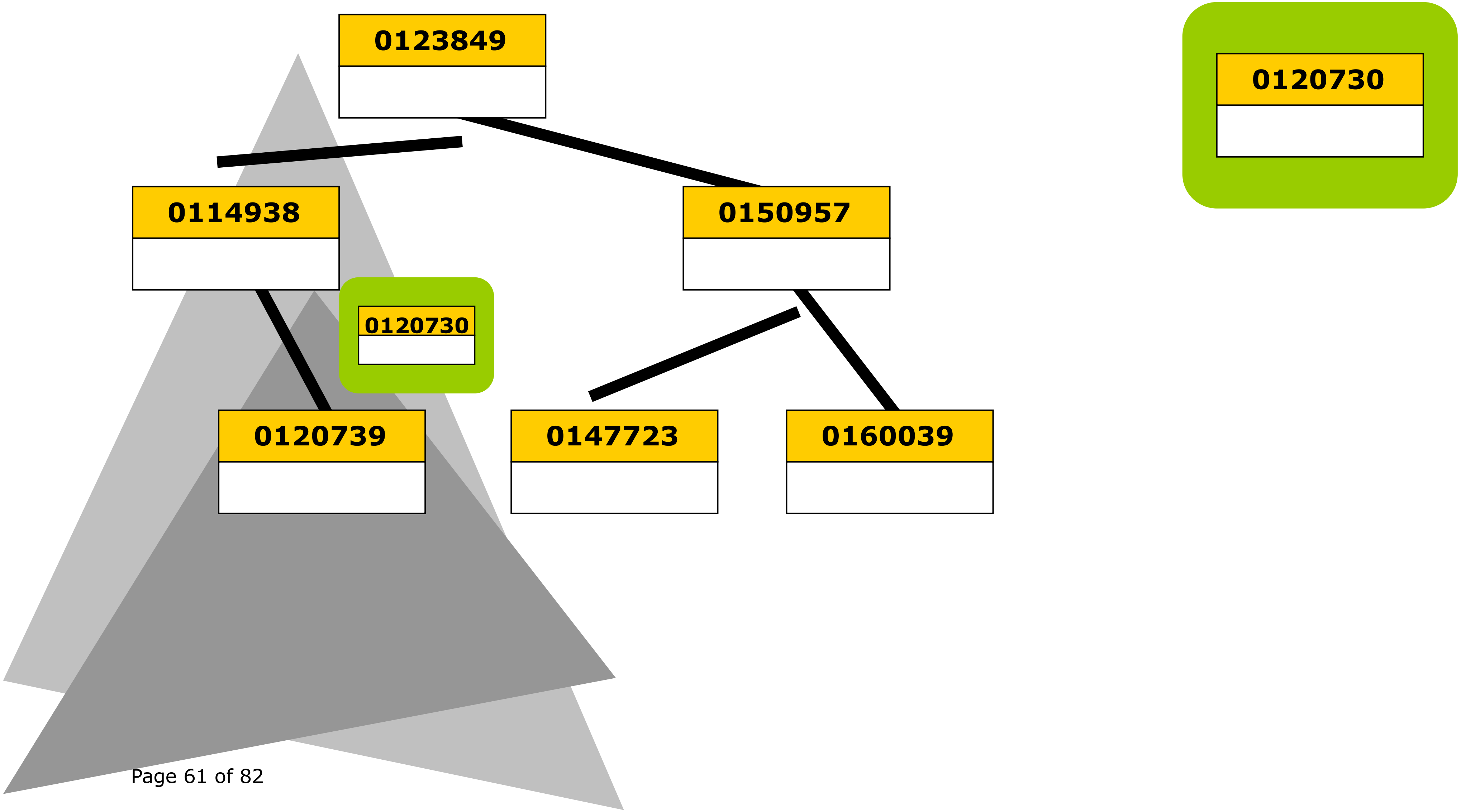
Constructing a Binary Search Tree



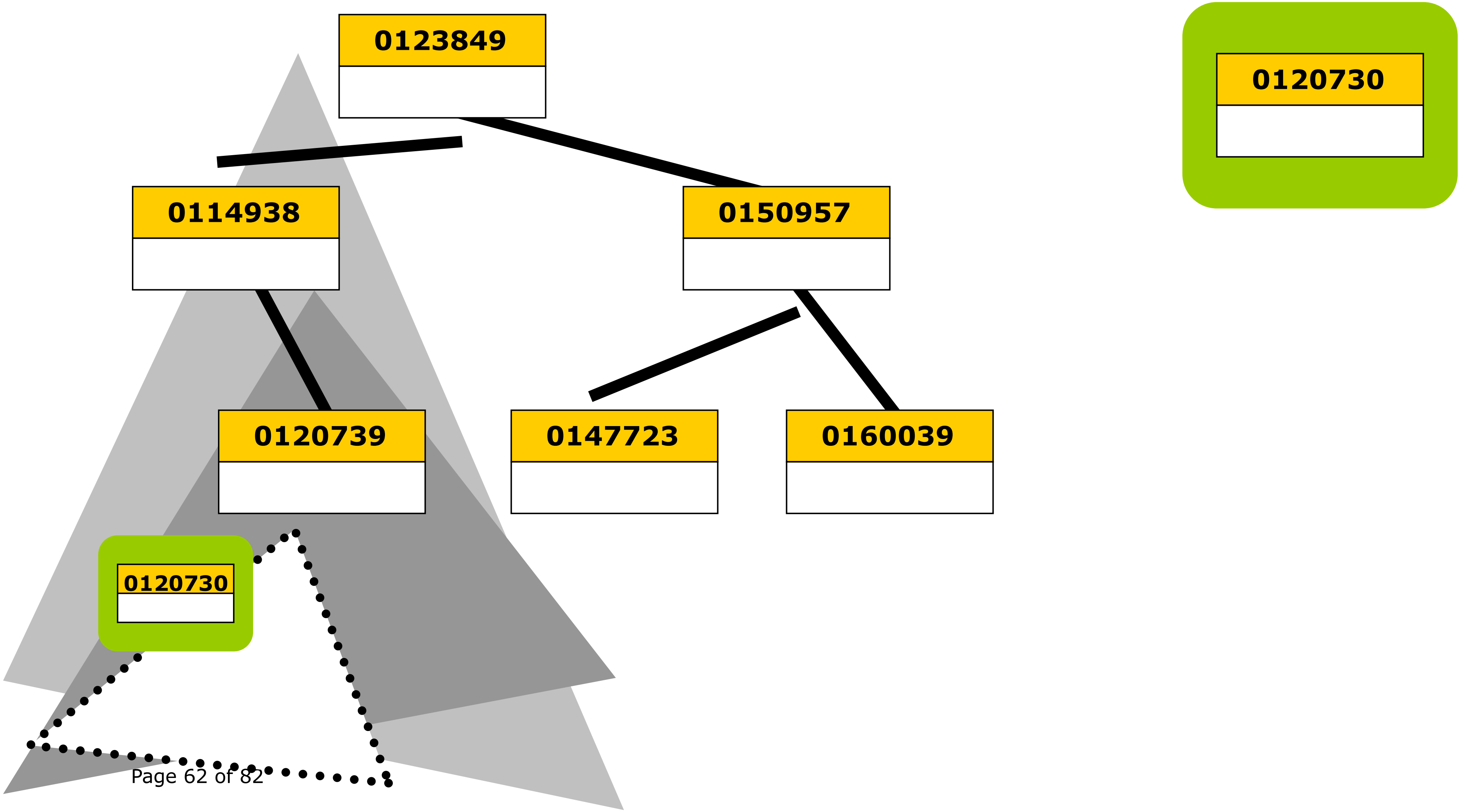
Constructing a Binary Search Tree



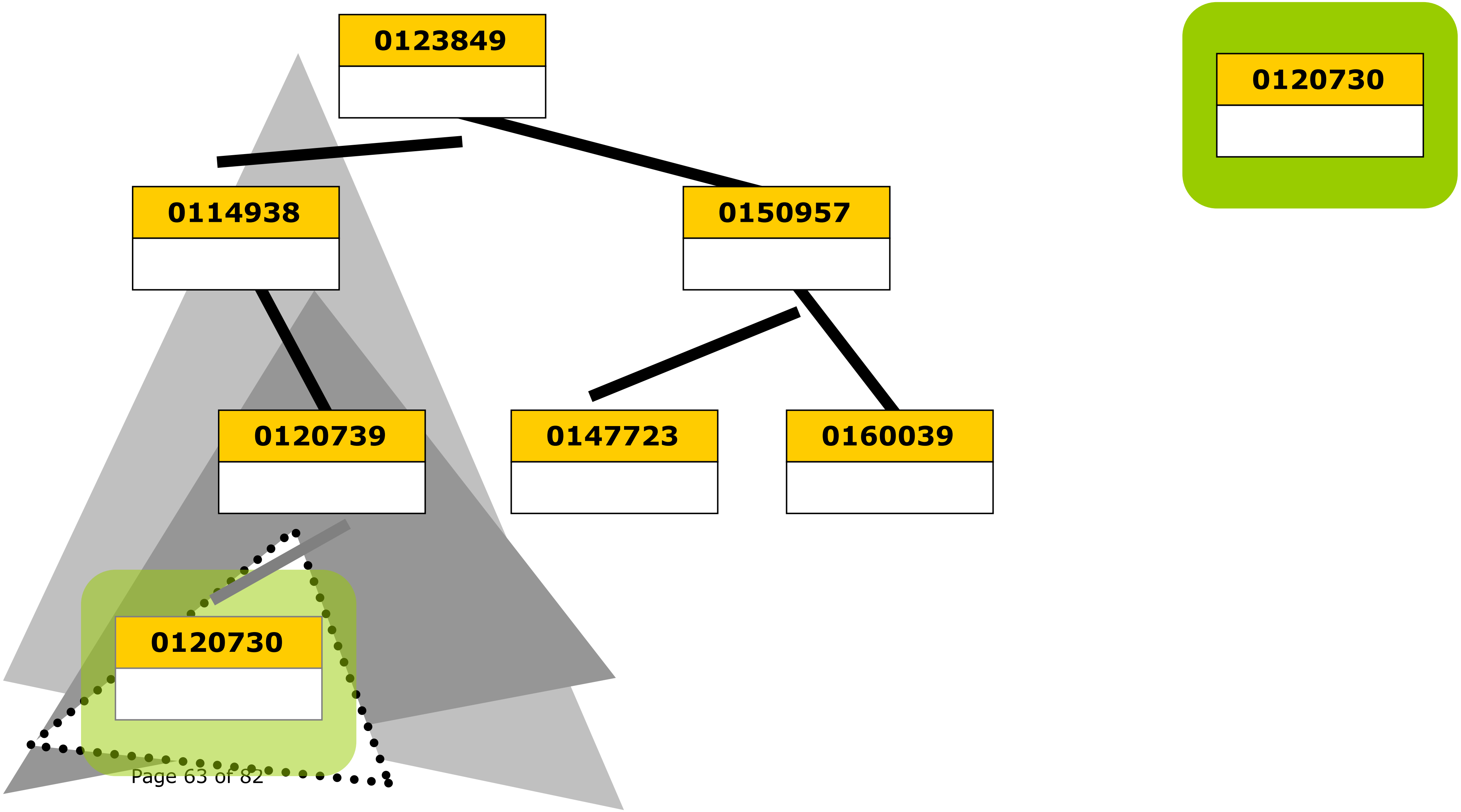
Constructing a Binary Search Tree



Constructing a Binary Search Tree



Constructing a Binary Search Tree

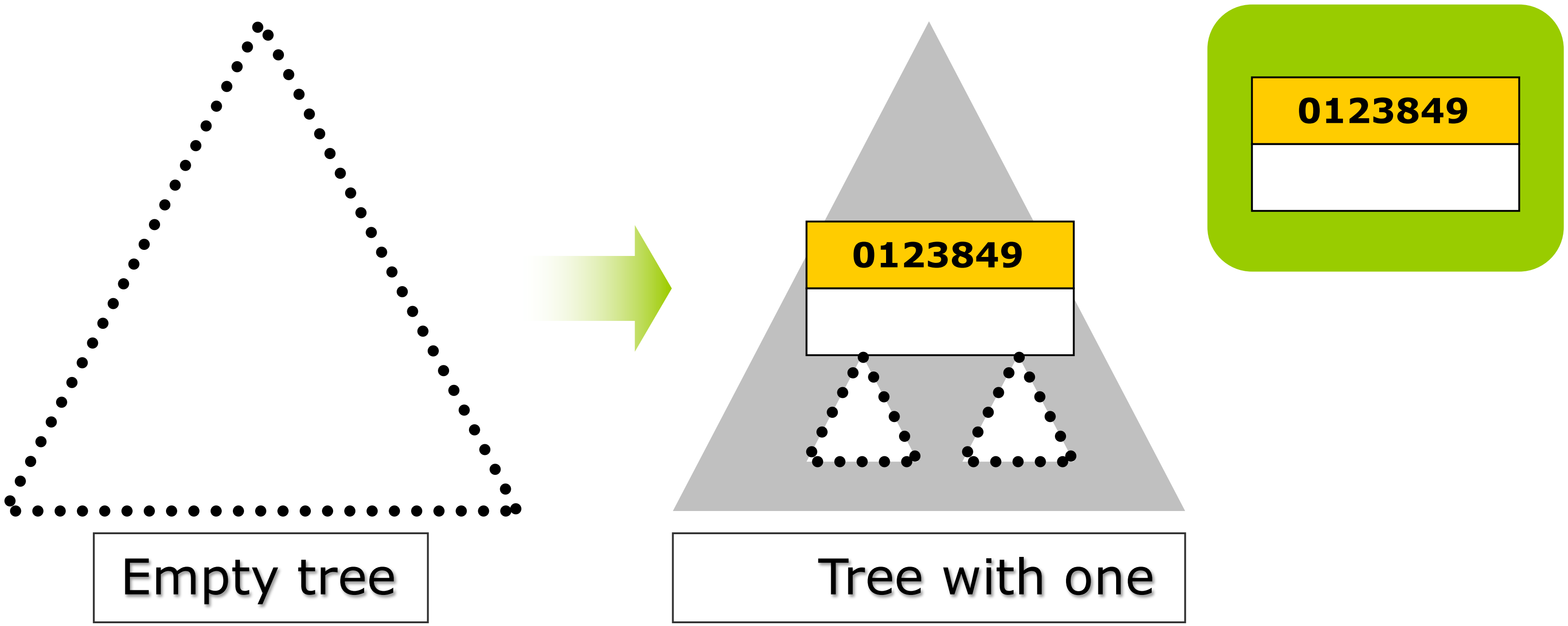


```
BinaryTreeADT InsertNode(BinaryTreeADT t, TreeNodeADT n) {

  if (TreeIsEmpty(t)) return NonemptyBinaryTree(n,
      EmptyBinaryTree(), EmptyBinaryTree());
  else {
    int sign = strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
    if (sign == 0) return NonemptyBinaryTree(n,
        LeftSubtree(t), RightSubtree(t));
    if (sign < 0) return NonemptyBinaryTree(Root(t),
        InsertNode(LeftSubtree(t),n), RightSubtree(t));
    return NonemptyBinaryTree(Root(t),
      LeftSubtree(t), InsertNode(RightSubtree(t), n));
    }
}
```

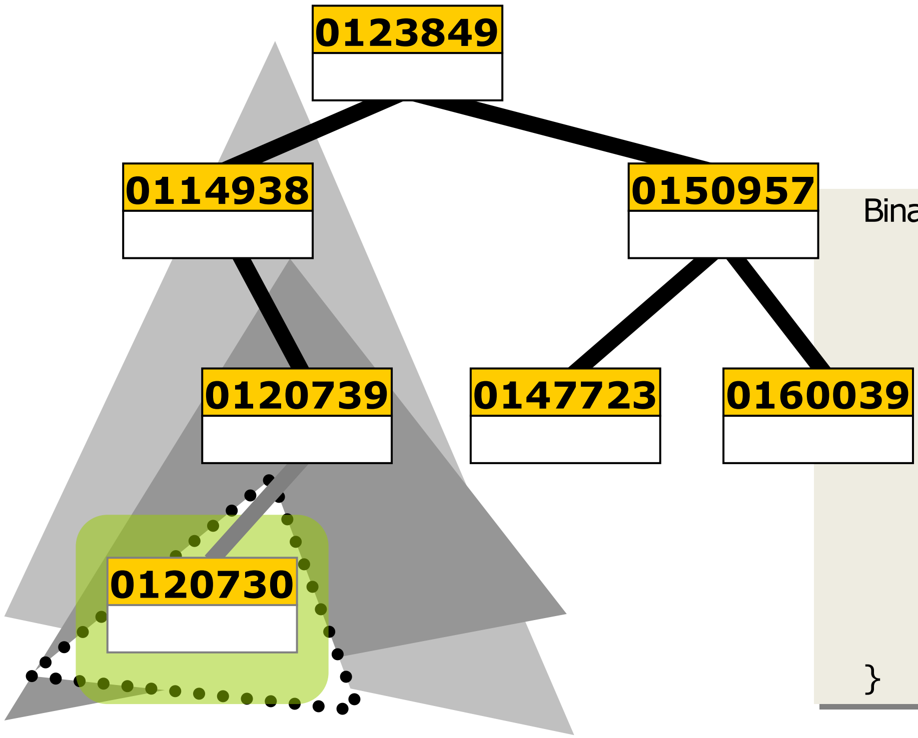
Note

- **if (TreeIsEmpty(t)) return NonemptyBinaryTree(n, EmptyBinaryTree(), EmptyBinaryTree());**

```
BinaryTreeADT InsertNode(BinaryTreeADT t, TreeNodeADT n) {
    if (TreeIsEmpty(t)) return NonemptyBinaryTree(n,
            EmptyBinaryTree(), EmptyBinaryTree());
    else {
        int sign = strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
        if (sign == 0) return NonemptyBinaryTree(n,
                LeftSubtree(t), RightSubtree(t));
        if (sign < 0) return NonemptyBinaryTree(Root(t),
                InsertNode(LeftSubtree(t),n), RightSubtree(t));
        return NonemptyBinaryTree(Root(t),
            LeftSubtree(t), InsertNode(RightSubtree(t), n));
        }
}
```

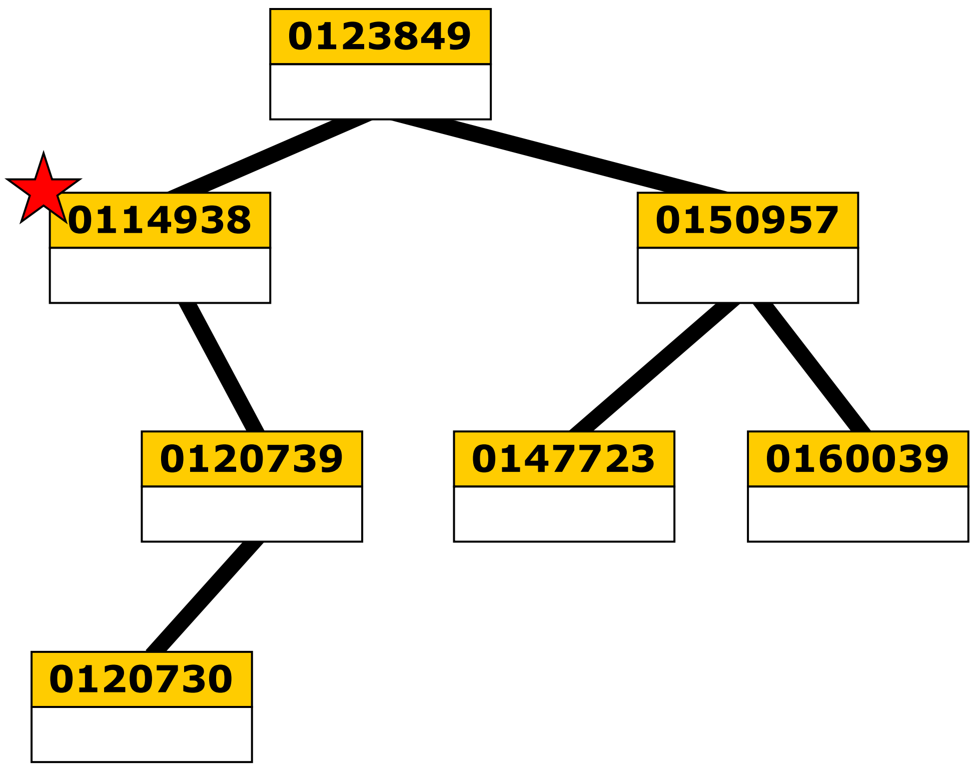


- **int sign =**
  strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
  if (sign == 0) return NonemptyBinaryTree(*n*, L, R));
  if (sign < 0) return NonemptyBinaryTree(r, *L'*, R);
  return NonemptyBinaryTree(r, L, *R'*));




```
BinaryTreeADT InsertNode(BinaryTreeADT t, TreeNodeADT n) {
    if (TreeIsEmpty(t)) return NonemptyBinaryTree(n,
            EmptyBinaryTree(), EmptyBinaryTree());
    else {
        int sign = strcmp(GetNodeKey(n), GetNodeKey(Root(t)));
        if (sign == 0) return NonemptyBinaryTree(n,
                LeftSubtree(t), RightSubtree(t));
        if (sign < 0) return NonemptyBinaryTree(Root(t),
                InsertNode(LeftSubtree(t),n), RightSubtree(t));
        return NonemptyBinaryTree(Root(t),
            LeftSubtree(t), InsertNode(RightSubtree(t), n));
        }
}
```

Binary Tree ADT function: FindMinNode



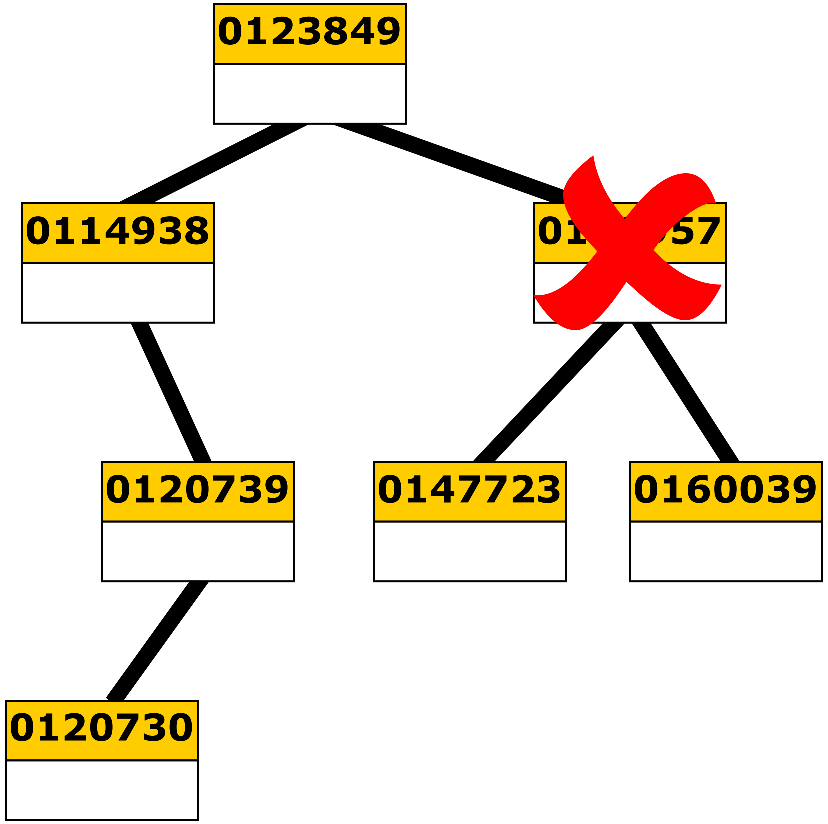
- Tree is empty: **error**.
- Tree is nonempty:
  - *Left subtree does not exist*: root is the min node!
  - *Left subtree exists*: find in the left subtree.

Binary Tree ADT function: FindMinNode

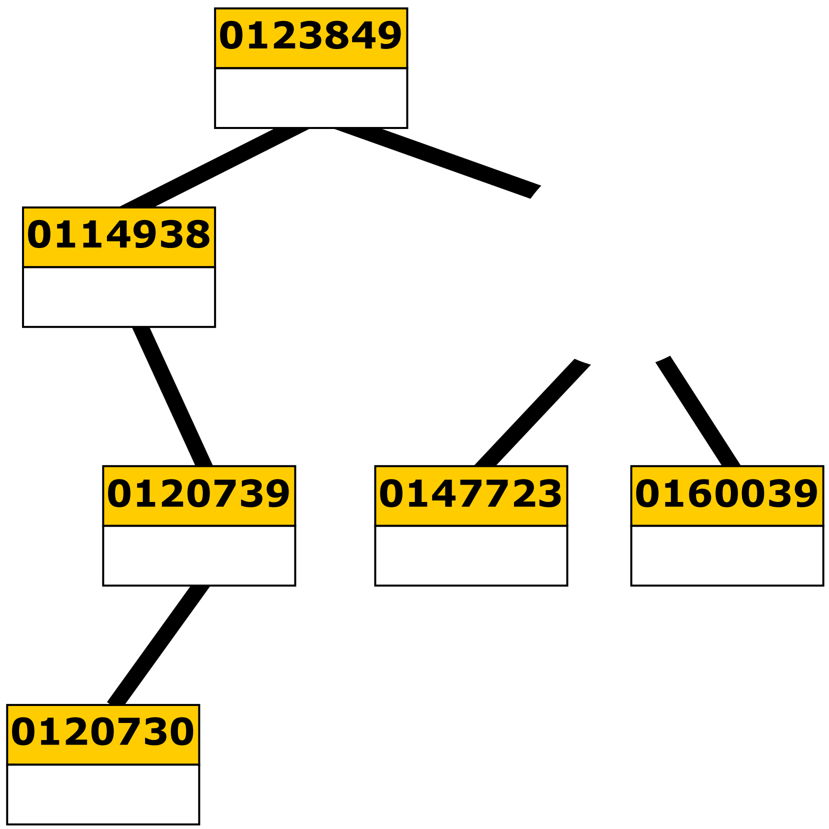
```
TreeNodeADT FindMinNode(BinaryTreeADT t) {

  if (TreeIsEmpty(t))
    return SpecialErrNode;

  if (TreeIsEmpty(LeftSubtree(t)))
    return Root(t);

  return FindMinNode(LeftSubtree(t));

}
```

- Tree is empty: **error**.
- Tree is nonempty:
  - *Left subtree does not exist*: <u>root</u> is the min node!
  - *Left subtree exists*: find in the left subtree.

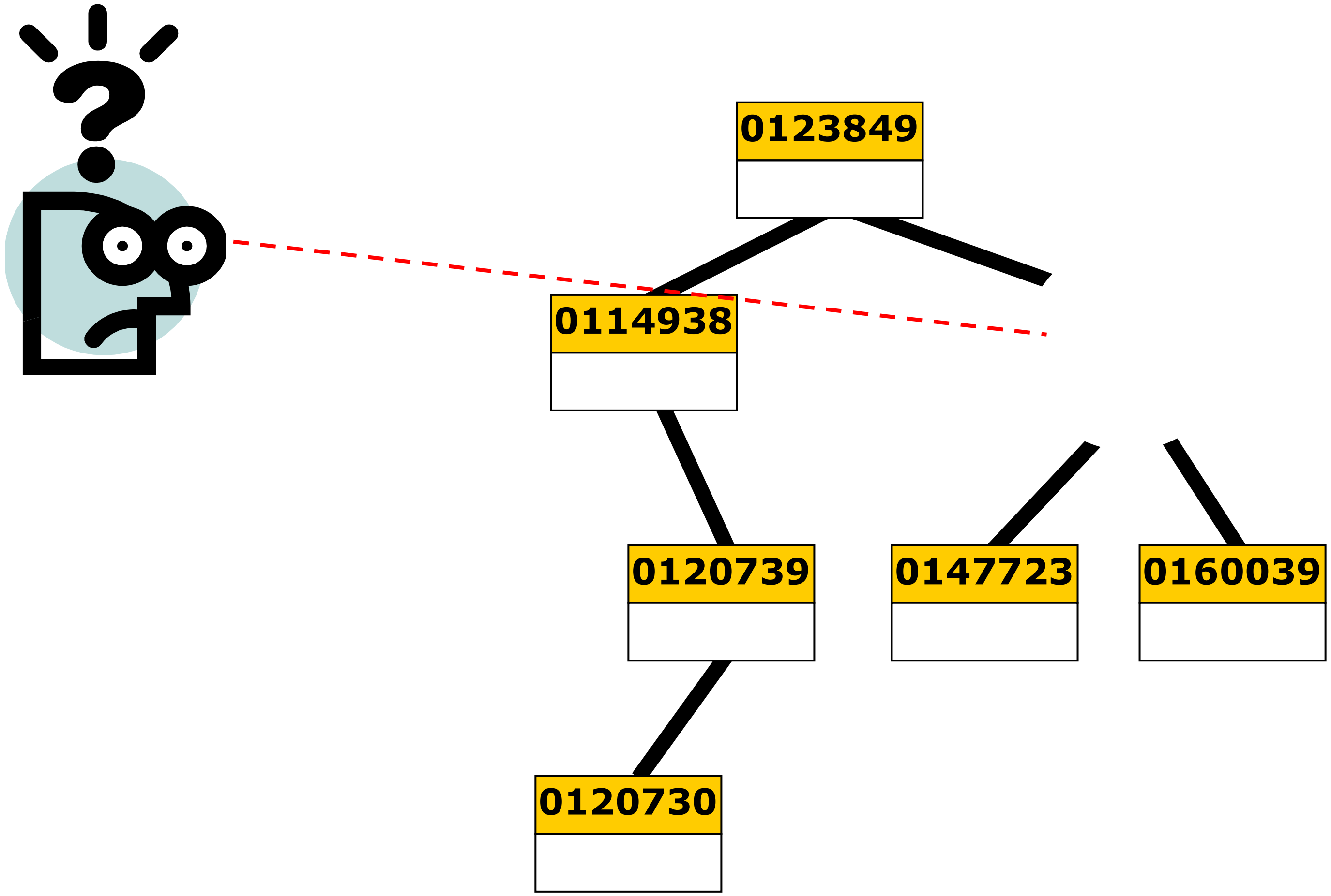
Binary Tree ADT function: Delete Node



Binary Tree ADT function: Delete Node

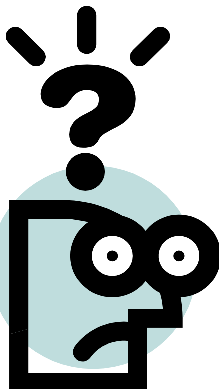


Binary Tree ADT function: Delete Node

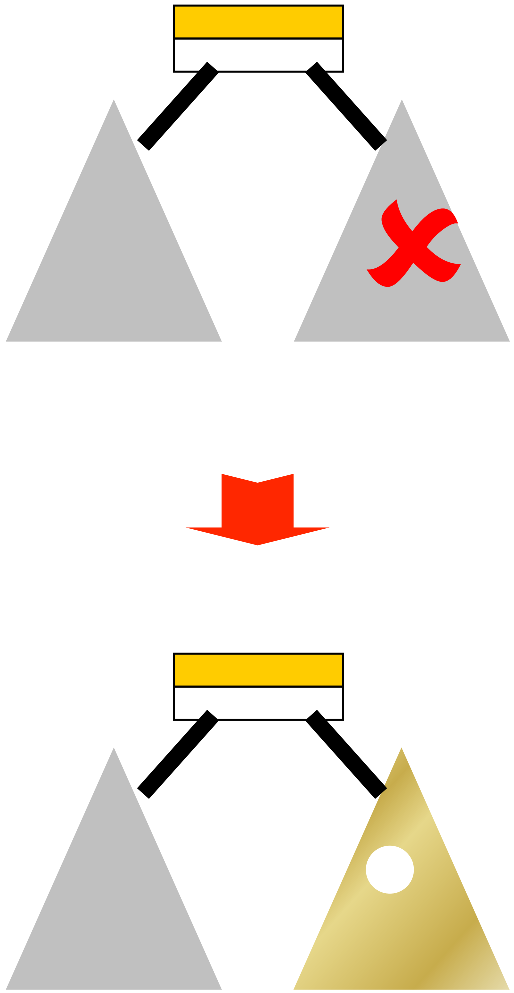


Binary Tree ADT function: Delete Node

Node deletion is not trivial!
Node deletion is hard!



Binary Tree ADT function: Delete Node



CASE 1

If the node to be deleted **is NOT** the root, then it is easy.

1. Delete the node from the left/right subtree.
2. Replace the subtree.

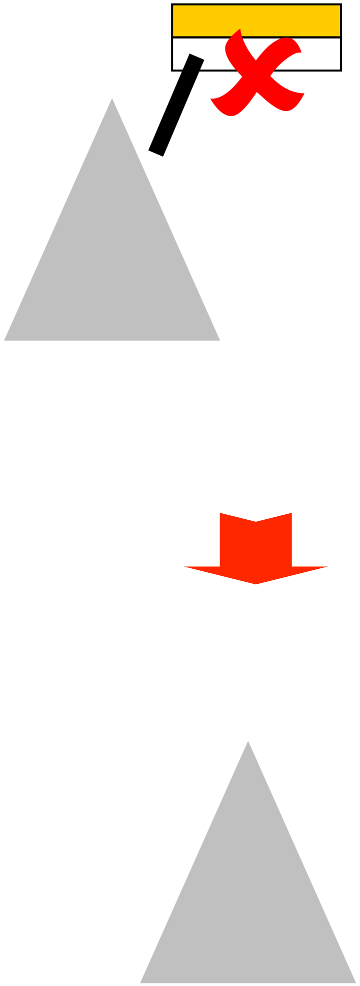
Binary Tree ADT function: Delete Node

CASE 2a

If the node to be deleted **IS** the root, **and** the root has no child, then it is easy.

1. Return an empty binary tree.

Binary Tree ADT function: Delete Node

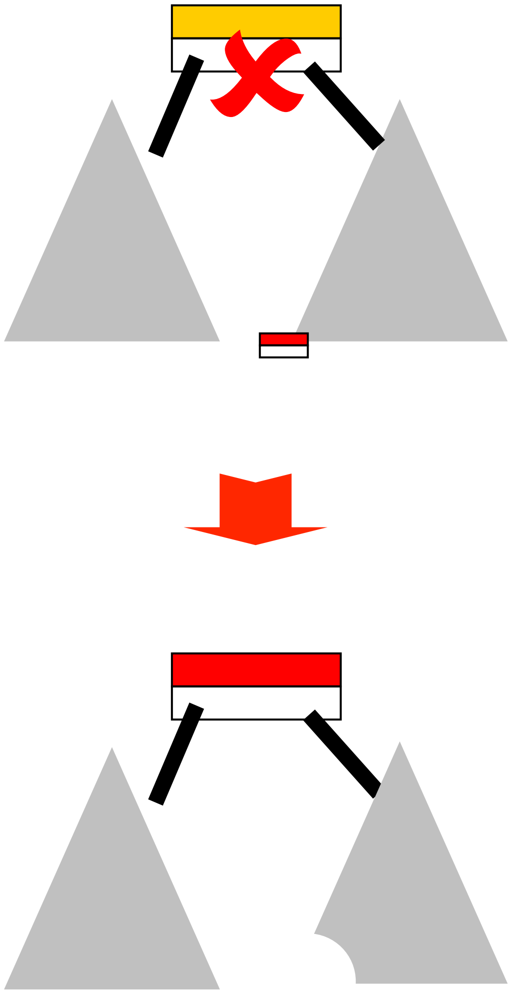


CASE 2b

If the node to be deleted **IS** the root, **and** the root has 1 child, then it is easy.

1. Return the left/right subtree.

Binary Tree ADT function: Delete Node

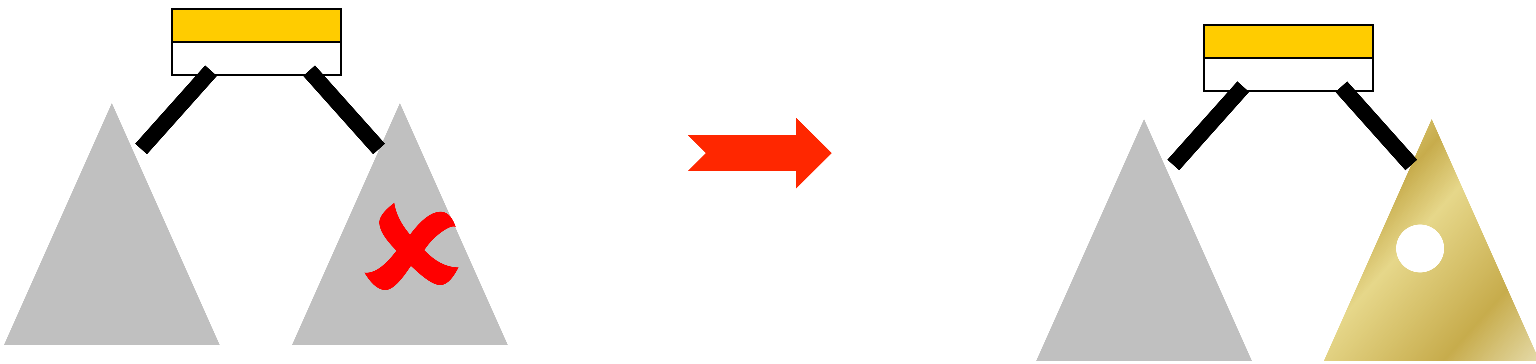


CASE 2c

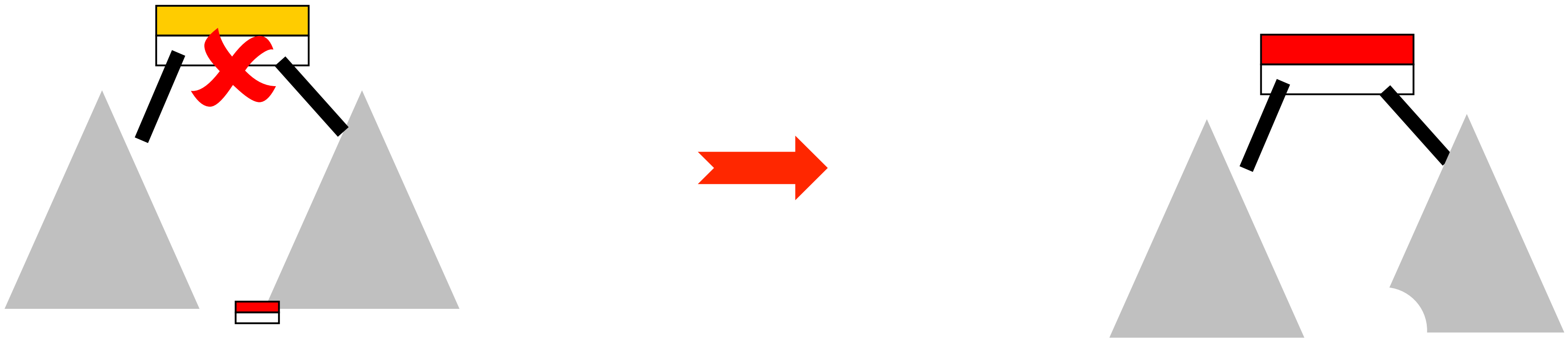
If the node to be deleted **IS** the root, **and** the root has 2 children, then it is easy.

1. Find the node with the smallest key in the right subtree.
2. Delete it from the right subtree.
3. Use it as the root.

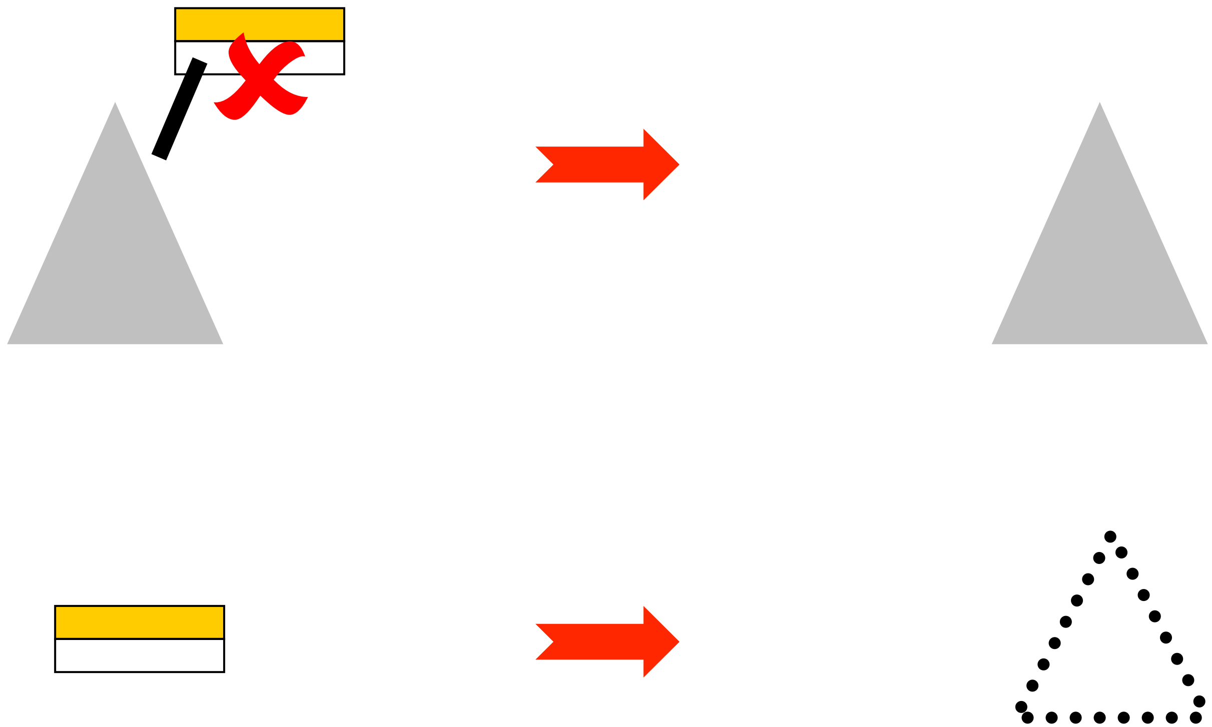
```
BinaryTreeADT DeleteNode(BinaryTreeADT t, char* k) {

   if (TreeIsEmpty(t)) exit(EXIT_FAILURE);
   int sign = strcmp(k, GetNodeKey(Root(t)));

   /* Case 1 */
   if (sign<0) return NonemptyBinaryTree(Root(t),
      DeleteNode(LeftSubtree(t), k), RightSubtree(t));
   /* Case 1 */
   if (sign>0) return NonemptyBinaryTree(Root(t),
      LeftSubtree(t), DeleteNode(RightSubtree(t), k));
```



```
/* Case 2c */
if (!TreeIsEmpty(LeftSubtree(t)) && !TreeIsEmpty(RightSubtree(t))) {
    TreeNodeADT M = FindMinNode(RightSubtree(t));
    return NonemptyBinaryTree(M, LeftSubtree(t),
                DeleteNode(RightSubtree(t),  GetNodeKey(M)));
    };
```



```
/* Cases 2a and 2b */
if (TreeIsEmpty(RightSubtree(t)))
    return LeftSubtree(t);
else
    return RightSubtree(t);
}
```

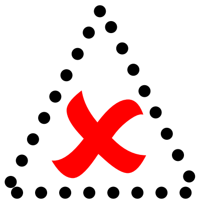


```
    /* Cases 2a and 2b */
    if (TreeIsEmpty(RightSubtree(t)))
        return LeftSubtree(t);
    else
        return RightSubtree(t);
}
```

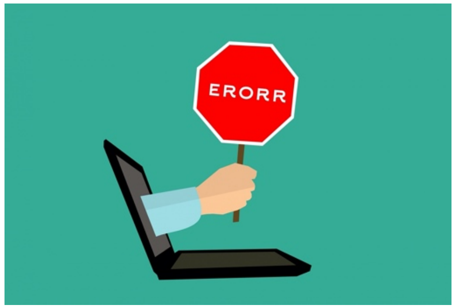
```
return TreeIsEmpty(RightSubtree(t)) ? LeftSubtree(t) : RightSubtree(t);
```

Binary Tree ADT function: Delete Node

CASE 0



If the tree is empty, then this is an error.



```
BinaryTreeADT DeleteNode(BinaryTreeADT t, char* k) {
   if (TreeIsEmpty(t)) exit(EXIT_FAILURE);
   int sign = strcmp(k, GetNodeKey(Root(t)));
/* Case 1 */
   if (sign<0) return NonemptyBinaryTree(Root(t),
      DeleteNode(LeftSubtree(t), k), RightSubtree(t));
   if (sign>0) return NonemptyBinaryTree(Root(t),
      LeftSubtree(t), DeleteNode(RightSubtree(t), k));
/* Case 2c */
   if (!TreeIsEmpty(LeftSubtree(t)) && !TreeIsEmpty(RightSubtree(t))) {
      TreeNodeADT M = FindMinNode(RightSubtree(t));
      return NonemptyBinaryTree(M, LeftSubtree(t),
            DeleteNode(RightSubtree(t), GetNodeKey(M)));
      };
/* Cases 2a and 2b */
   return TreeIsEmpty(RightSubtree(t)) ? LeftSubtree(t) : RightSubtree(t);
}
```