ALGORITHMS AND DATA STRUCTURES

BINARY TREES

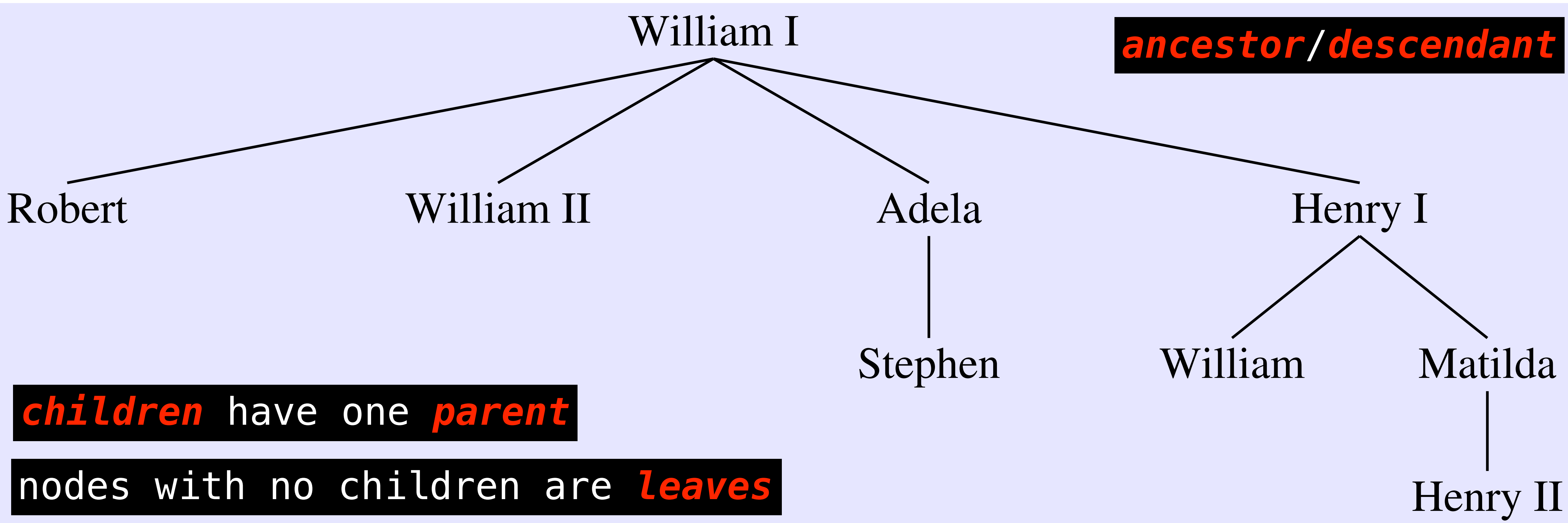
TREES

- are data structures that uses pointers hierarchically
- are collections of individual nodes with the properties:
  - there is a node called the **root** at the top of the hierarchy
  - **other** nodes are connected to the root via a single line path
- **Examples**
  - biological classifications
  - directory hierarchies

TREES: TERMINOLOGY

- Family trees

`height`: number of nodes in longest path from root to leaf



`ancestor/descendant`

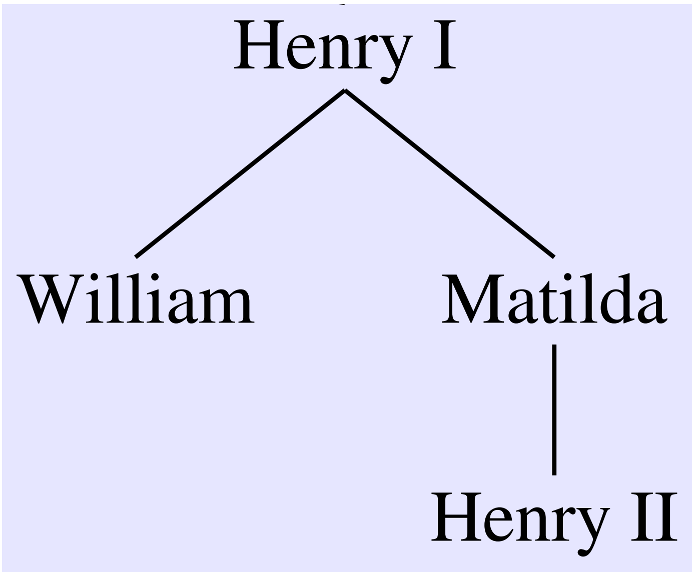
`children` have one `parent`

nodes with no children are `leaves`

nodes with same parent are `siblings`

TREES ARE RECURSIVE

- Trees have subtrees
  - *Subtree*: a node with all its descendants
  - pattern repeats at every level of the tree
- A subtree is a tree: recursive nature
  - a tree is then a node with a collection of children
    - children are themselves trees

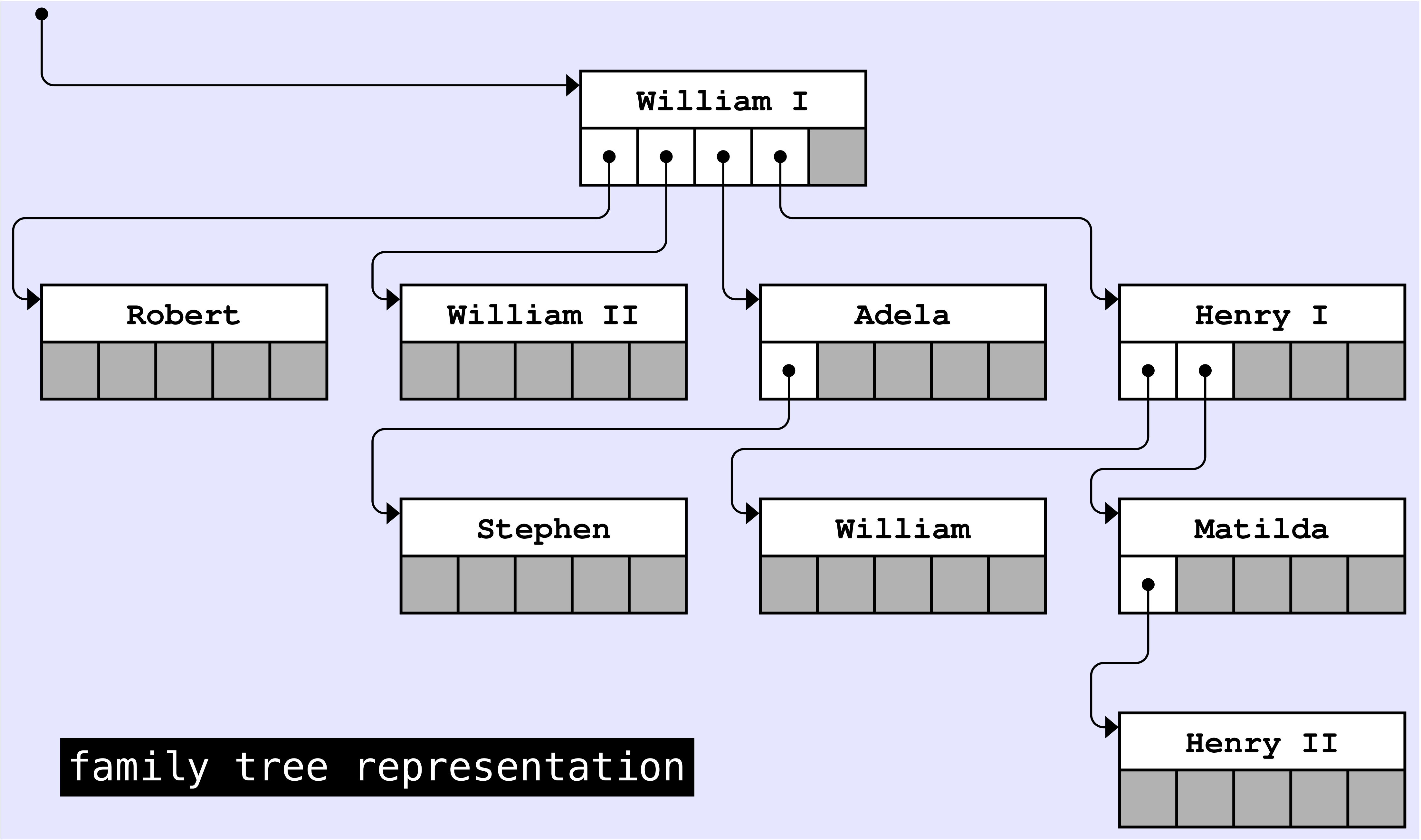


TREES: REPRESENTATION

- How to model hierarchies in trees?
- Use the following definition
  - trees are pointers to nodes
  - nodes contain trees
- In C++ terms

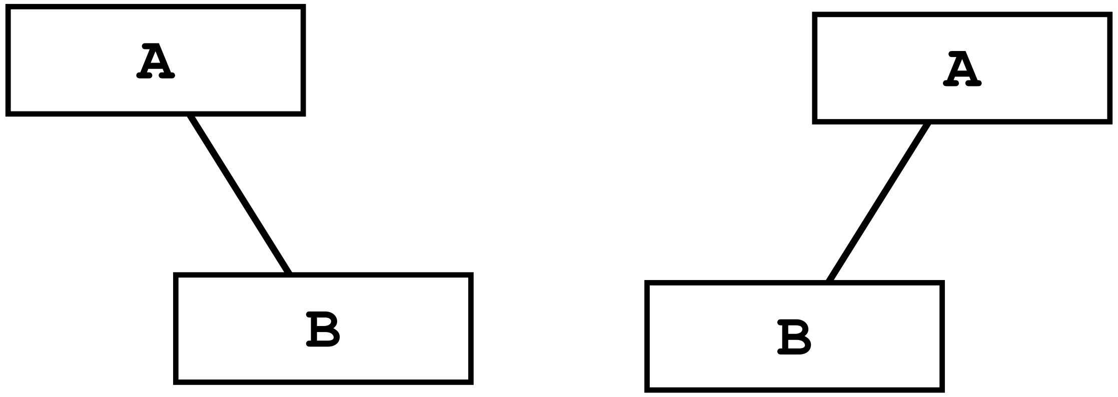
```
struct FamilyTreeNode {
    string name;
    Vector<FamilyTreeNode *> children;
};
```

TREES: REPRESENTATION



BINARY SEARCH TREES

- **Binary tree:** a tree with the following additional properties
    - nodes have at most two children
    - every node, except the root, is dubbed either ***left child*** or ***right child***
- **<u>Example</u>**: Are these trees the same?



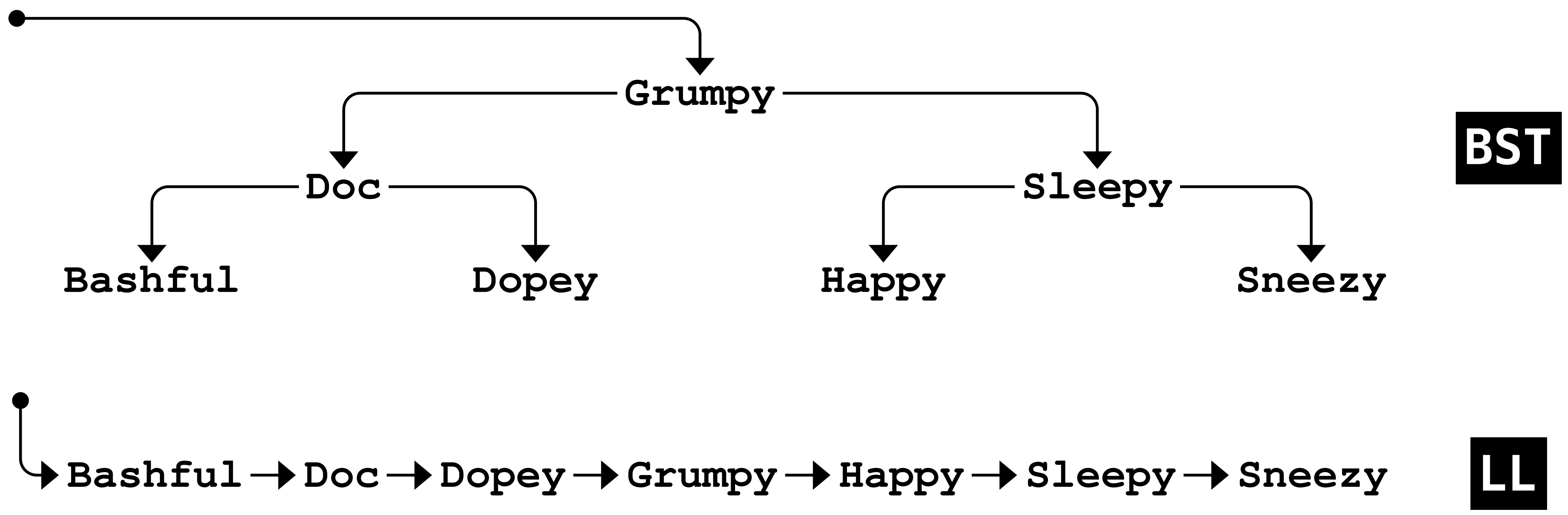
BINARY SEARCH TREES

- Most common application of a binary tree is a binary search tree
- **Binary search tree** have the following properties
  - nodes contain a value called a *key* that defines their order
  - key values are unique—no key can appear more than once
  - at every node in the tree, key must be greater than all the keys in the subtree rooted at its left child and less than all the keys in the subtree rooted at its right child

BINARY SEARCH TREE PROPERTY

BINARY SEARCH TREES

- Extraction of an element
  - $O(\lg N)$ time in a binary search tree
  - What about a linked list?



BINARY SEARCH TREES: FIND

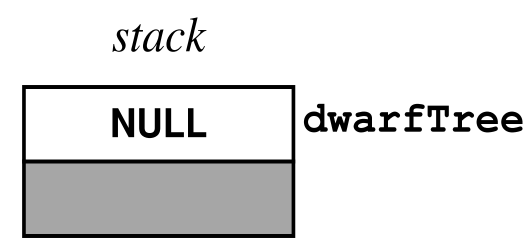
```
struct BSTNode {
    string key;
    BSTNode *left, *right;
};

BSTNode *findNode(BSTNode *t, string key) {
    if (t == nullptr) return nullptr;
    if (key == t->key) return t;
    if (key < t->key) {
        return findNode(t->left, key);
    } else {
        return findNode(t->right, key);
    }
}
```

BINARY SEARCH TREES: INSERTION

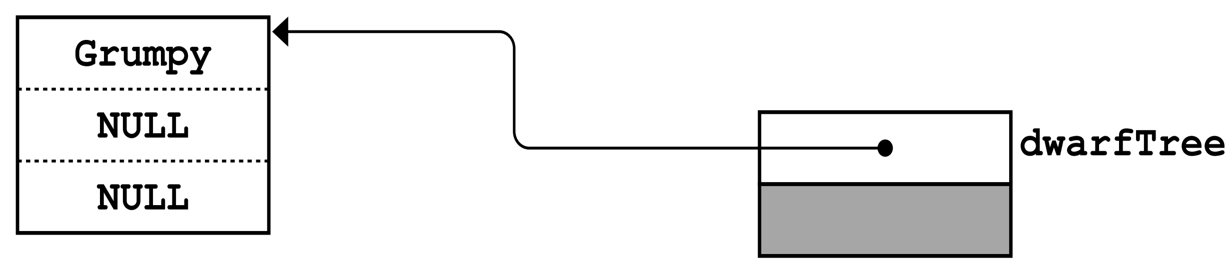
```
void insertNode(BSTNode * & t, string key) {
    if (t == nullptr) {
        t = new BSTNode;
        t->key = key;
        t->left = t->right = nullptr;
    } else {
        if (key != t->key) {
            if (key < t->key) {
                insertNode(t->left, key);
            } else {
                insertNode(t->right, key);
            }
        }
    }
}
```

BINARY SEARCH TREES: INSERTION

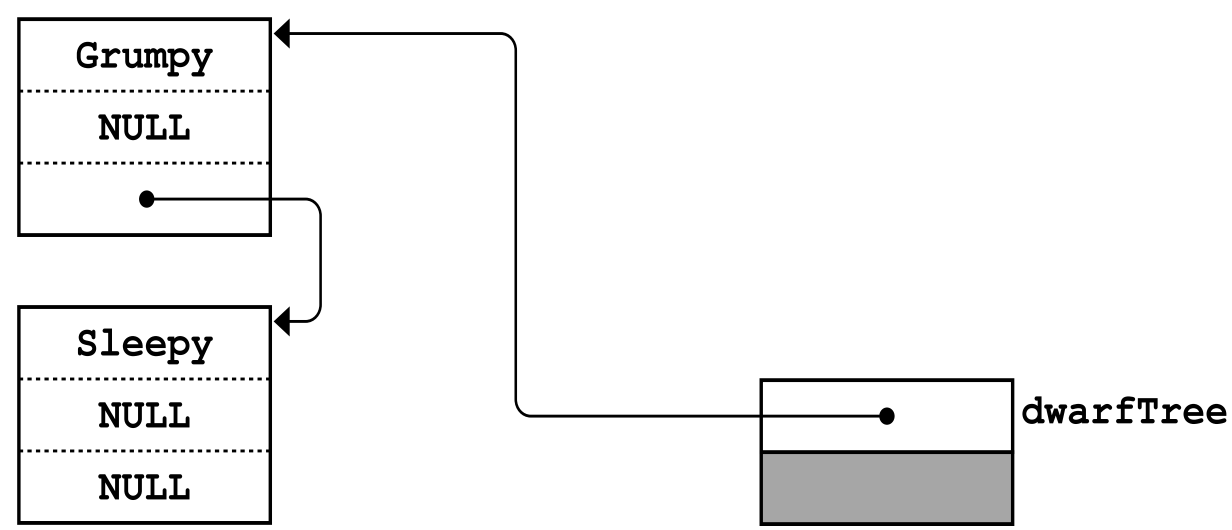
```
BSTNode *dwarfTree = nullptr;
```



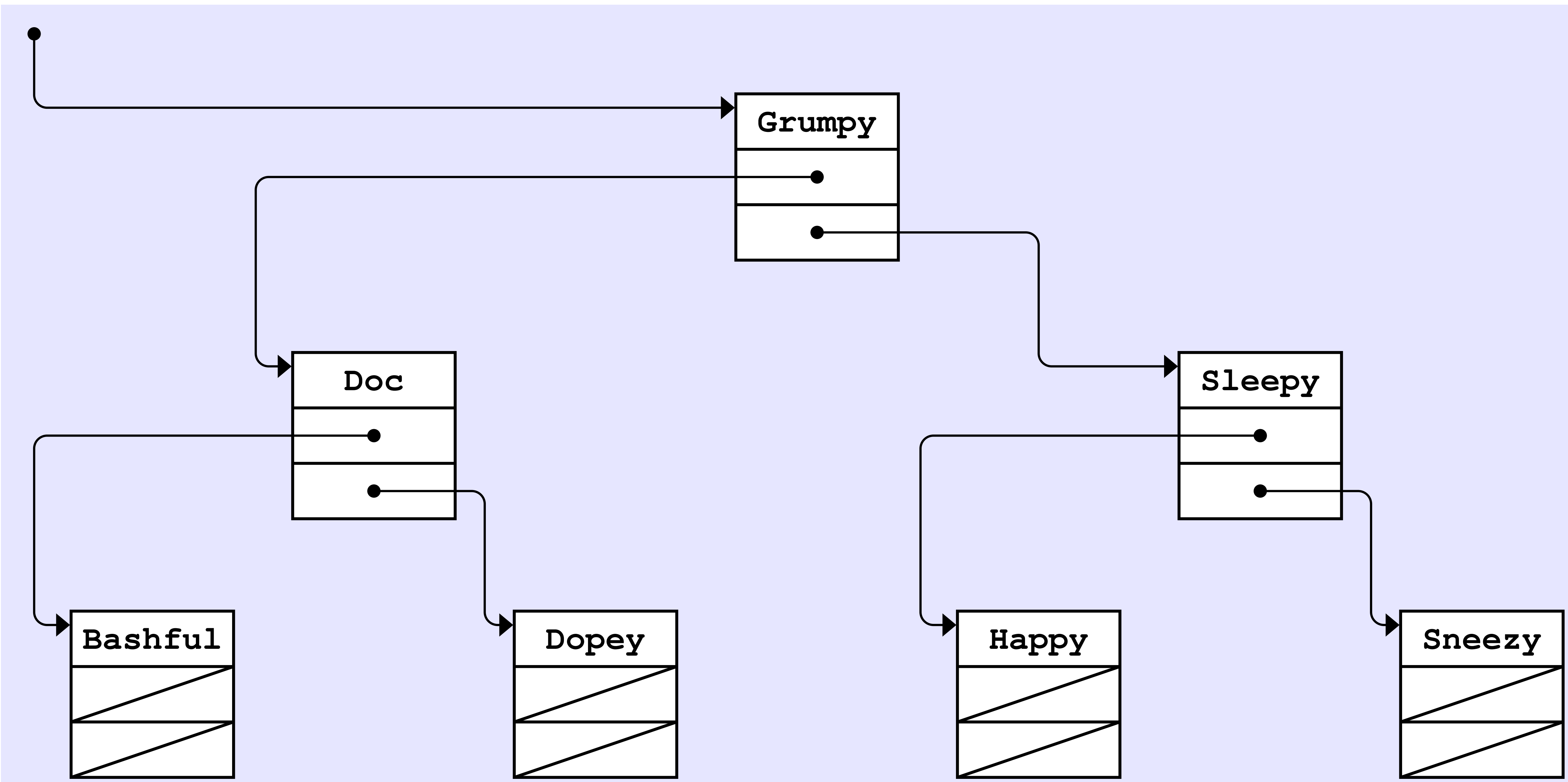

```
insertNode(dwarfTree, "Grumpy");
```



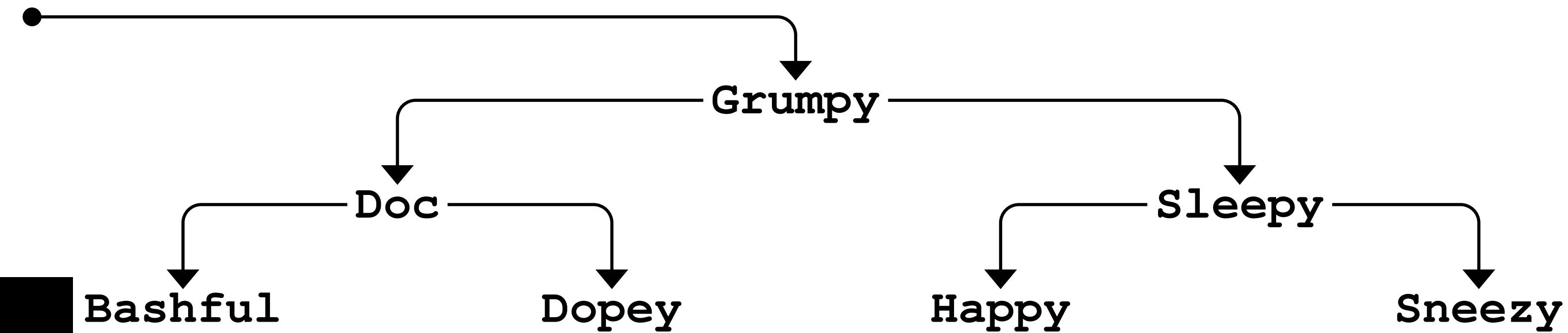

```
insertNode(dwarfTree, "Sleepy");
```



BINARY SEARCH TREES: INSERTION

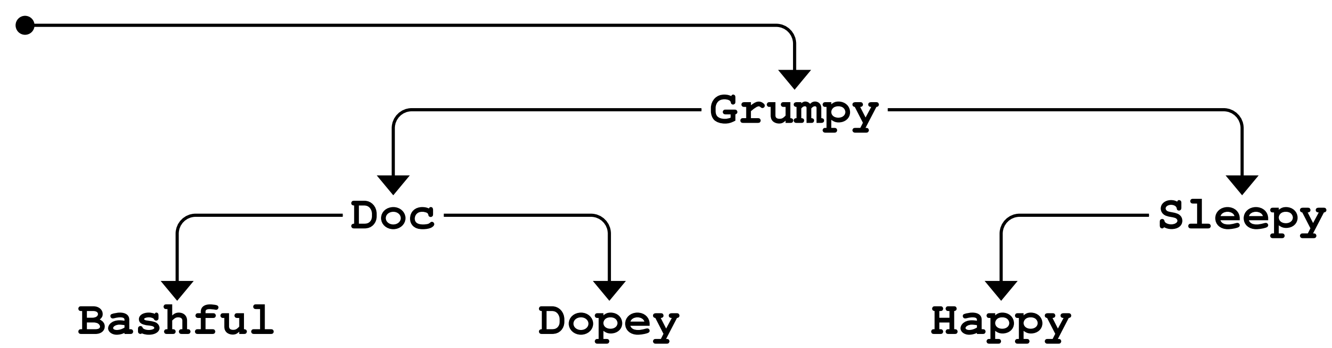


BINARY SEARCH TREES: REMOVE



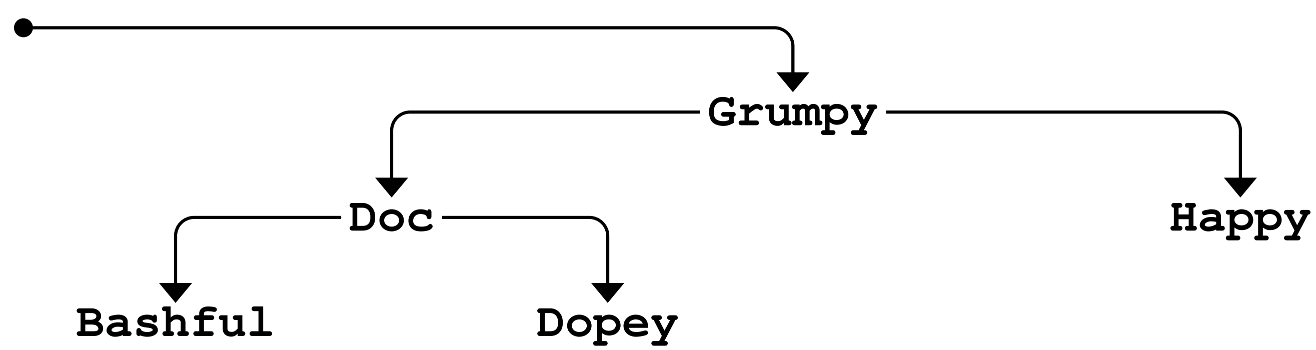
remove Sneezy:

replace Sneezy with nullptr



remove Sleepy:

replace Sleepy with either non-nullptr child



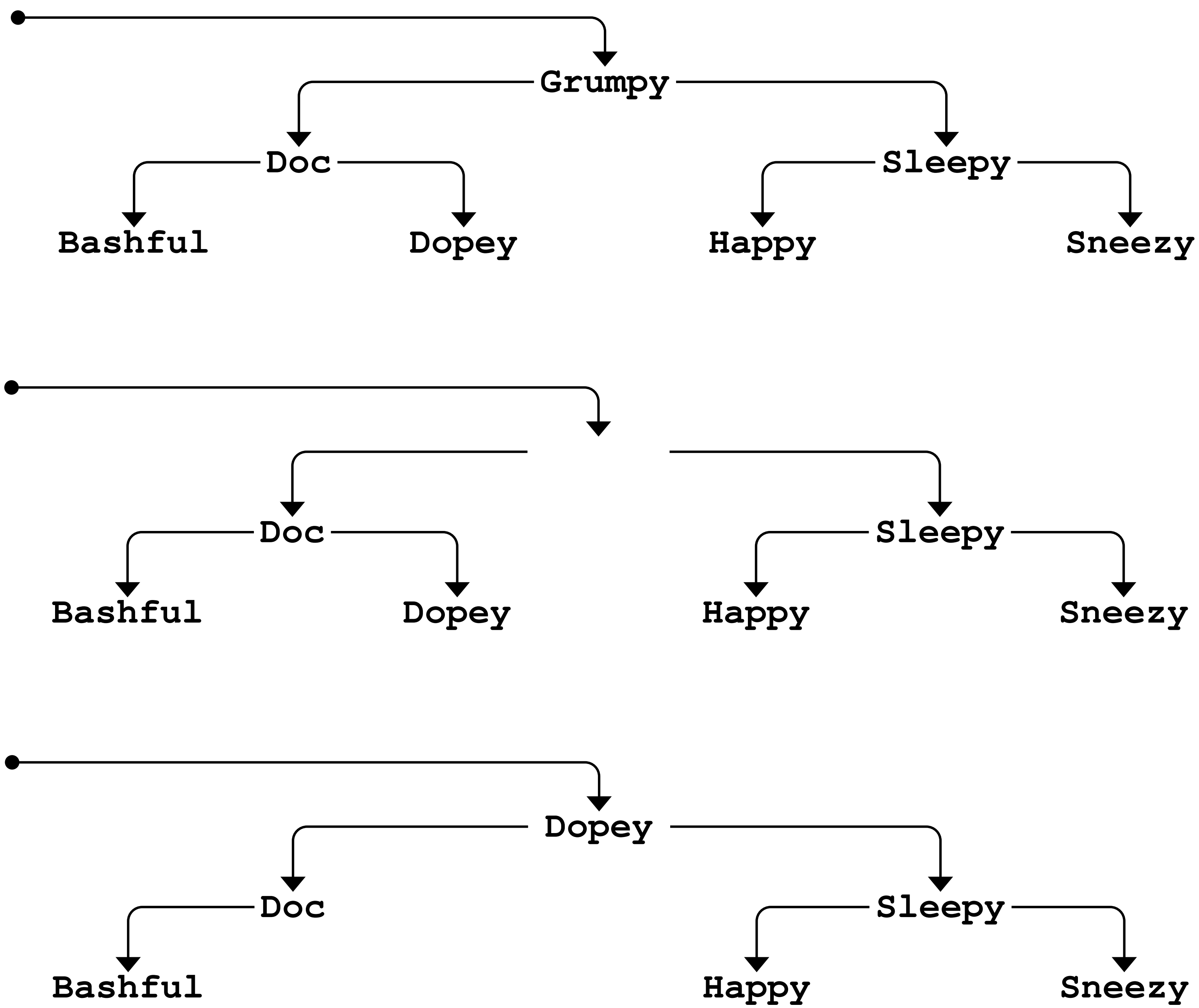
BINARY SEARCH TREES: REMOVE

remove Grumpy:
What to do next?

remove Grumpy:
use rightmost node in left subtree or leftmost node in right subtree

replace Grumpy with either of such nodes (e.g. Dopey)

replace Dopey with its left child



BINARY SEARCH TREES: TRAVERSALS

- To display a binary search tree

```
void displayTree(BSTNode *t) {
    if (t != nullptr) {
        displayTree(t->left);
        cout << t->key << endl;
        displayTree(t->right);
    }
}
```

```
Bashful
Doc
Dopey
Grumpy
Happy
Sleepy
Sneezy
```

- ***traversing or walking the tree***: process of going through the nodes and performing some operation on each one of them

BINARY SEARCH TREES: TRAVERSALS

- <u>**Theorem**</u>:

  If `x` is the root of an n-node binary tree, then the call `displayTree(x)` takes $\Theta(n)$ time.

- How to prove it?

BINARY SEARCH TREES: TRAVERSALS

- Types of traversals or walkings
  - ***inorder***: consists of processing the current node between the recursive calls to the left and right subtrees
  - ***preorder***: current node is processed before traversing either of its subtrees
  - ***postorder***: subtrees are processed first, followed by the current node

BINARY SEARCH TREES: TRAVERSALS

```
void preorderTraversal(BSTNode *t) {
    if (t != nullptr) {
        cout << t->key << endl;
        preorderTraversal(t->left);
        preorderTraversal(t->right);
    }
}

void postorderTraversal(BSTNode *t) {
    if (t != nullptr) {
        postorderTraversal(t->left);
        postorderTraversal(t->right);
        cout << t->key << endl;
    }
}
```

Grumpy
Doc
Bashful
Dopey
Sleepy
Happy
Sneezy

Bashful
Dopey
Doc
Happy
Sneezy
Sleepy
Grumpy

BINARY SEARCH TREES: REMOVE (REVISITED)

- There are three cases to consider
  - ***node has no children***: remove it by making its ancestor corresponding child a `nullptr` node
  - ***node has one child***: replace node by its child by modifying node's ancestor's child with node's child
  - ***node has two children***: find node's successor and have it take node's position in the tree. Node's <u>*right(left)*</u> subtree becomes successor's new <u>*right(left)*</u> subtree

BINARY SEARCH TREES: REMOVE (REVISITED)