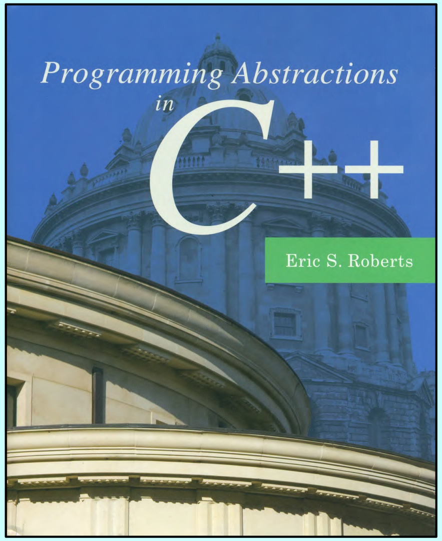
CHAPTER 16

Trees

I like trees because they seem more resigned to the way they have to live than other things do.

—Willa Cather, *O Pioneers!*, 1913



Flashback

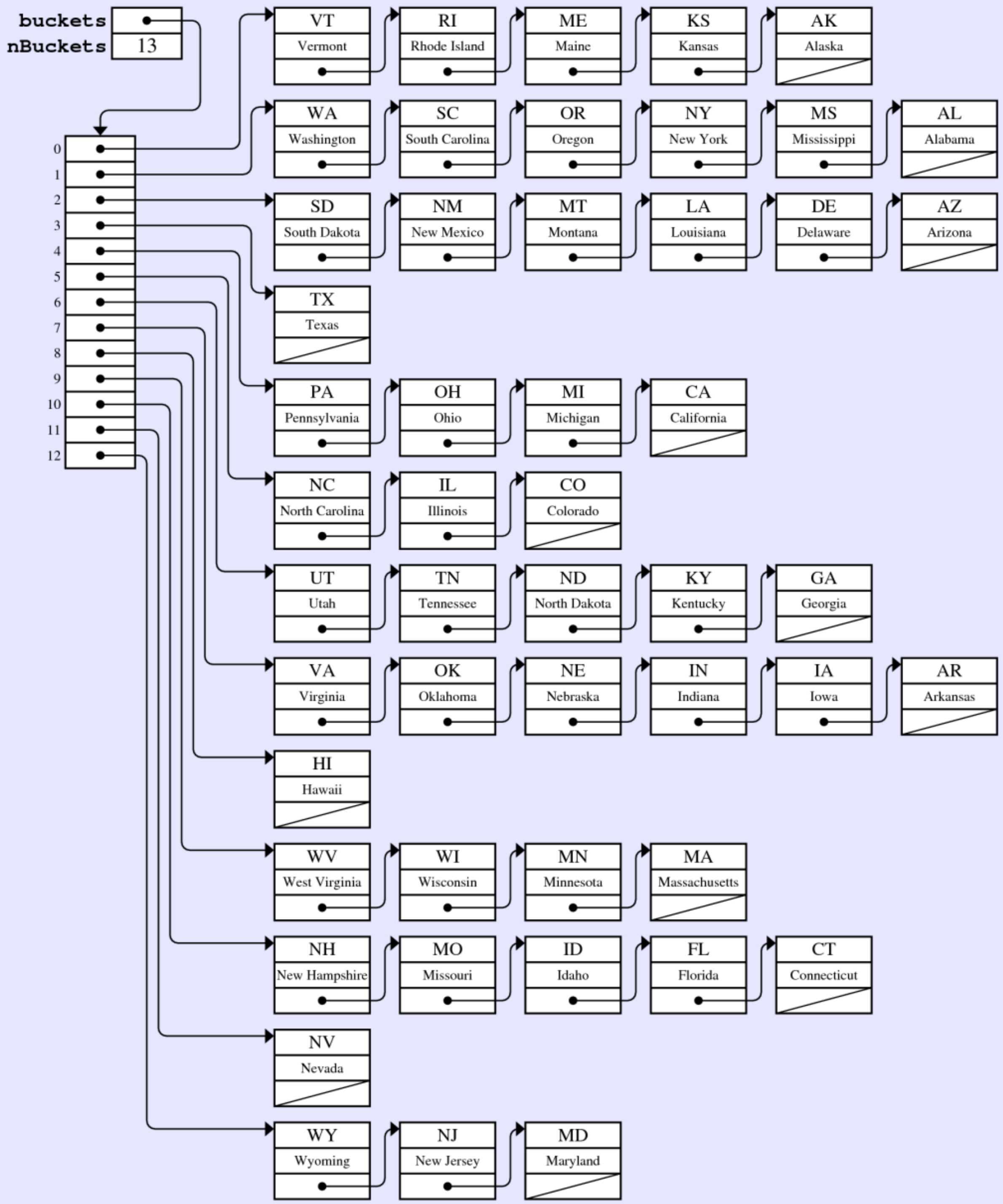
Implementation Strategies for Maps

- There are several strategies you might choose to implement the map operations **`get`** and **`put`**. Those strategies include:
  1. ***Linear search***. Keep track of all the key/value pairs in a vector. In this model, both the **`get`** and **`put`** operations run in $O(N)$ time.
  2. ***Binary search***. If you keep the vector sorted by the key, you can use binary search to find the key. Using this strategy improves the performance of **`get`** to $O(\log N)$ and **`put`** is still $O(N)$.
  3. ***Table lookup in a grid***. In the two-character code example, you can store the state names in a 26 x 26 **`Grid<string>`** in which the first and second indices correspond to the two letters in the code. Because you can now find any code in a single step, this strategy is $O(1)$, although this performance comes at a cost in memory space (only 50/676 occupied).
  4. ***Hashing***. Use a preferably $O(1)$ time hash function to transform keys into as uniformly as possible distributed integer hash codes, which tell the implementation where it should look for a particular key, thereby reducing the search time dramatically.

Limitations of Hashing – Order Sequence

- In the last chapter, We have seen how hashing makes it possible to implement the **`get`** and **`put`** operations for a map in $O(1)$ time.
- Despite its extraordinary efficiency, hashing is not always the best strategy for implementing maps, because of the following limitations:
  - Hash tables depend on being able to compute a hash function on some key. Expanding the hash-function idea so that it applies to types other than strings is subtle.
  - Using the range-based **`for`** on hash tables does not deliver the keys in any sensible order. Even when the keys have a natural order (such as the lexicographic order used with strings), the hash table code for iteration cannot take advantage of that fact.
- The goal now is to explore another representation that supports iterating through the elements in order.

FIGURE 15-9 Hash table containing the state abbreviations



Flashback

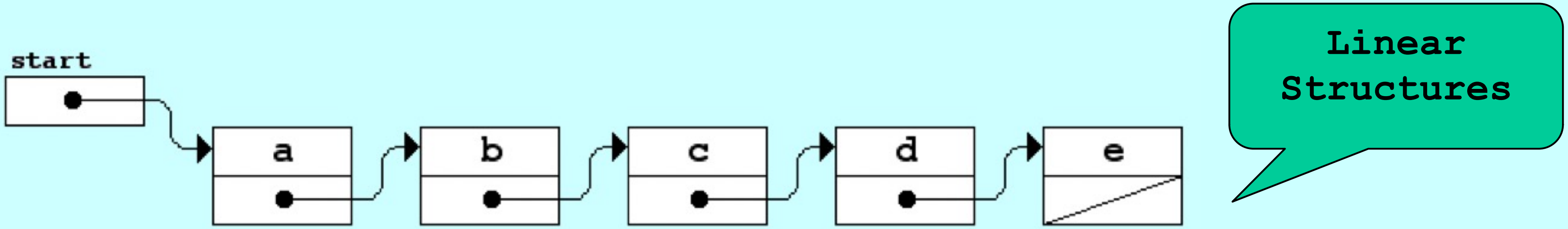
Iterator Order

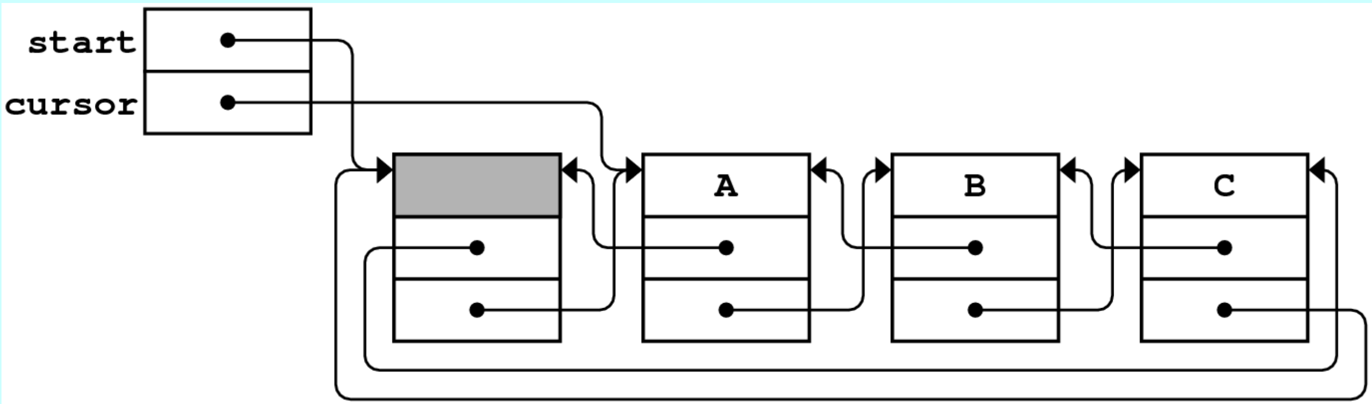
- When you look at the documentation for an iterator, one of the important things to determine is whether the collection class specifies the order in which elements are generated. The Stanford libraries make the following guarantees:
  - Iterators for the **`Vector`** class operate in index order.
  - Iterators for the **`Grid`** class operate in ***row-major order***, which means that the iterator runs through every element in row 0, then every element in row 1, and so on.
  - Iterators for the **`Map`** class deliver the keys in the order imposed by the standard comparison function for the key type; iterators for the **`HashMap`** class return keys in a seemingly random order.
  - Iterators for the **`Set`** class deliver the elements in the order imposed by the standard comparison function for the value type; the **`HashSet`** class is unordered.
  - ~~Iterators for the **`Lexicon`** class always deliver words in alphabetical order.~~

Limitation – Arrays and Linear Linked List

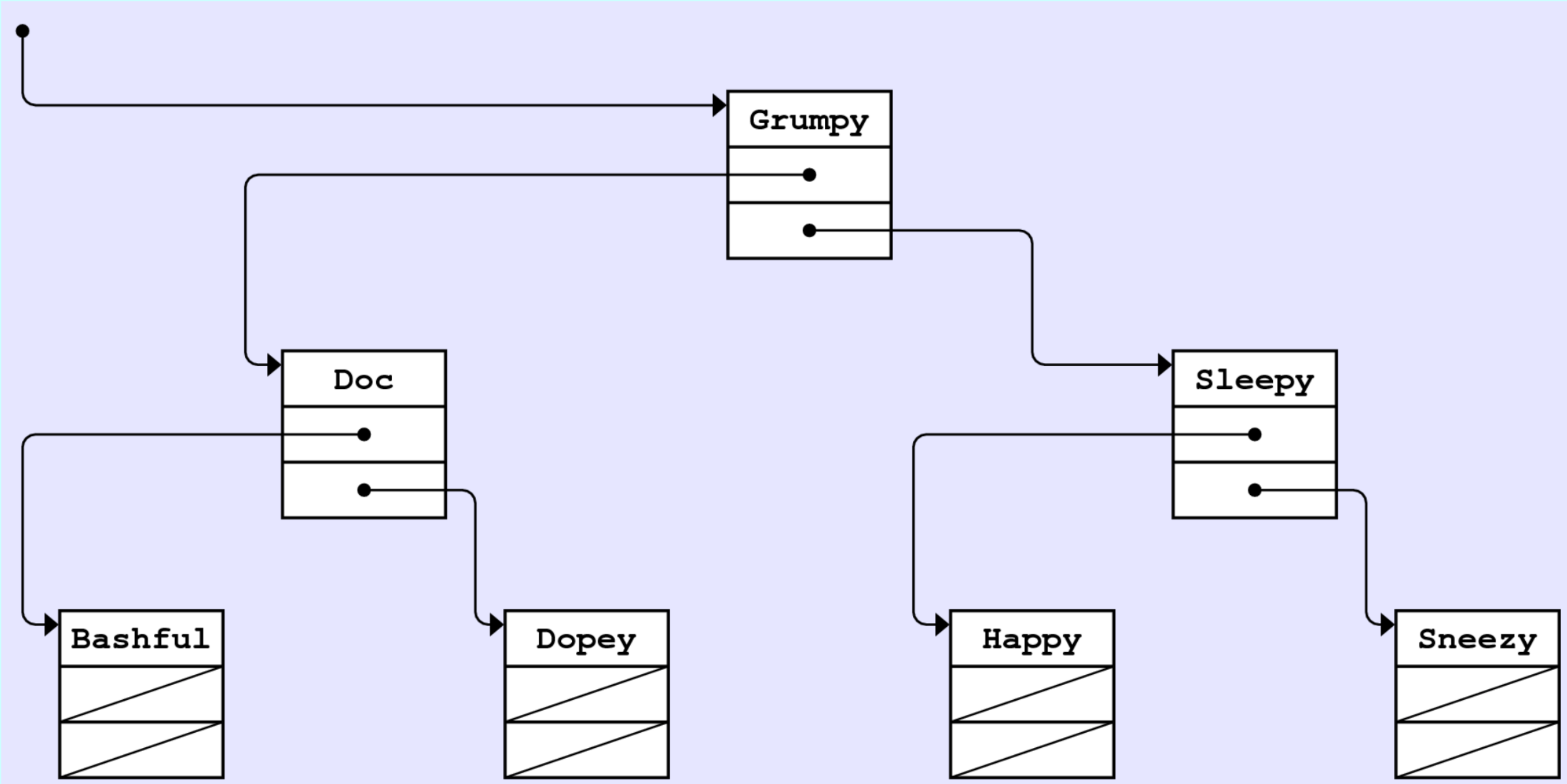
- One of the strategies outlined previously for implementing a map was to use a sorted array to hold the key-value pairs. Given that representation, binary search made it possible to find a key in $O(\log N)$ time.
- The problem with the sorted array strategy was that inserting a new key required $O(N)$ time to ***maintain the order***.
- In the editor buffer, linked lists solved the insertion problem. Unfortunately, turning a sorted array into a linked list makes it impossible to apply binary search because there is no way to go to the middle element in $O(1)$ time.
- But what if you could point to the middle element in a linked list? That question gives rise to a new structure called a ***tree***, which provides the key to implementing a map with $O(\log N)$ performance for both the `get` and `put` operations.

Pointers as ordering relationship



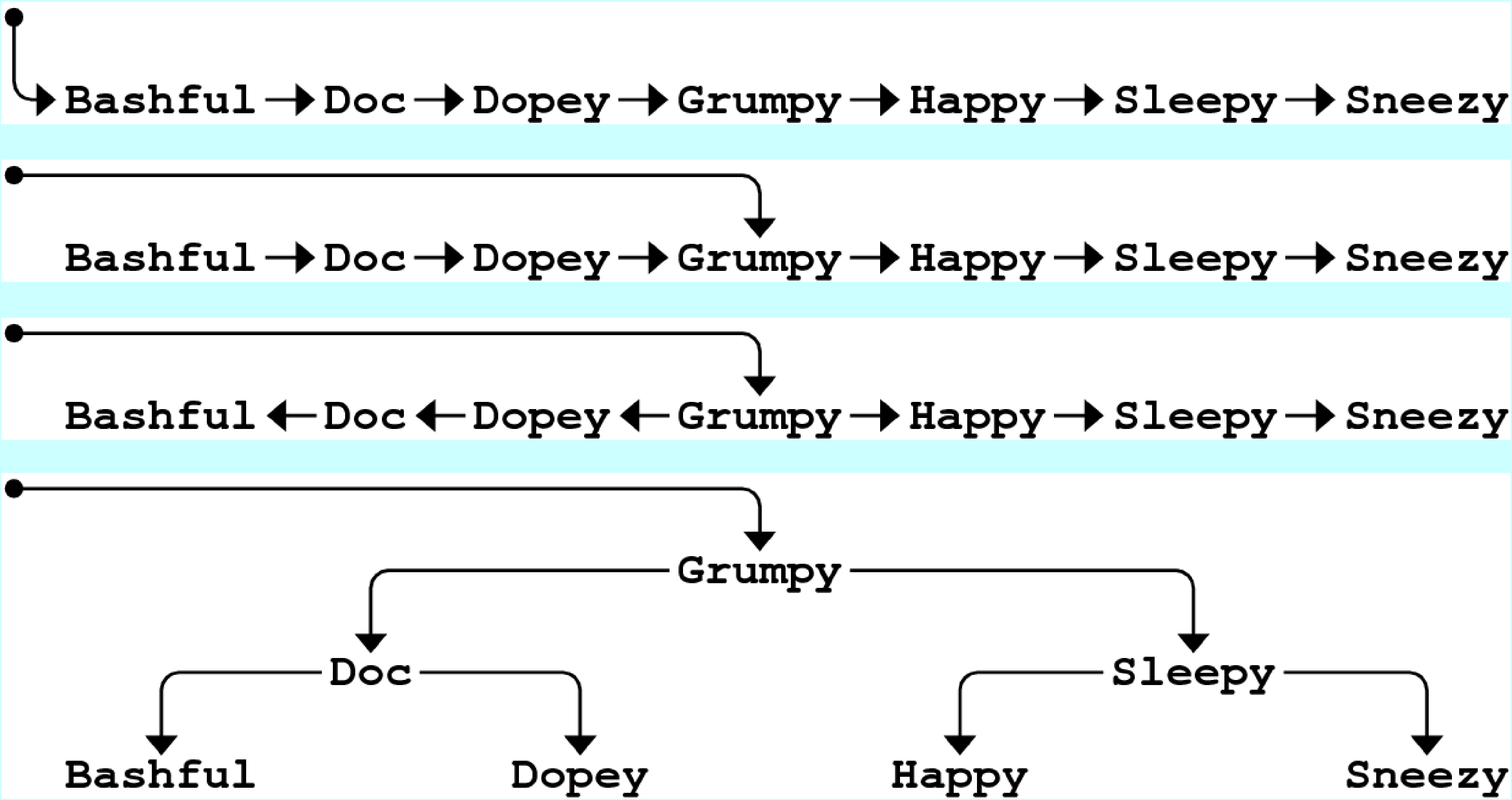


Pointers – Non-Linear Structure



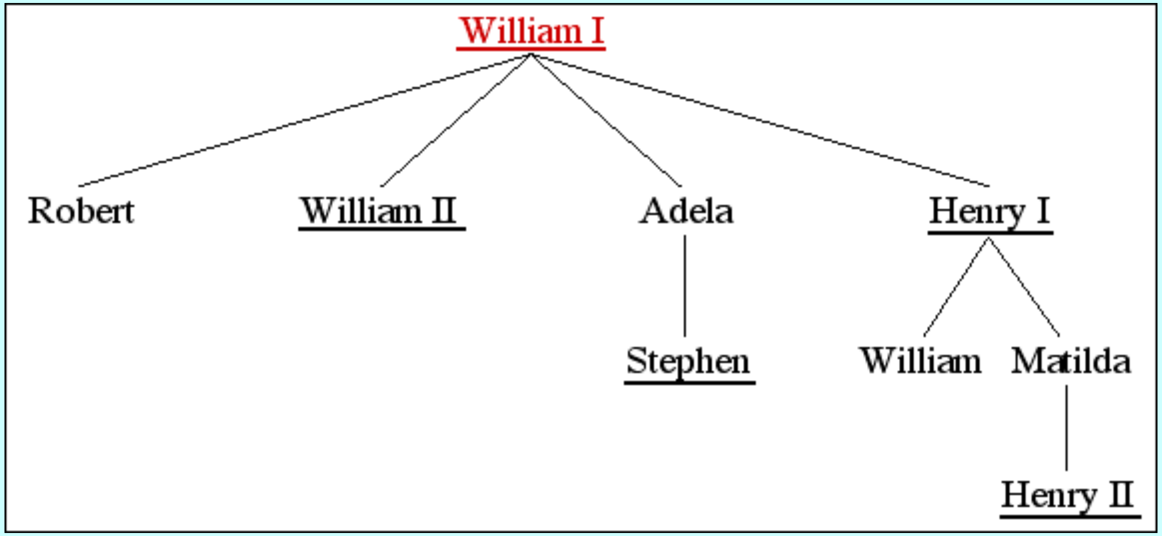
Motivation - Binary Search Trees

- The idea is to combine the linked list data structure (for easier insertion/deletion) and the binary search algorithm (for faster search).



Trees

- In the text, the first example used to illustrate tree structures is the royal family tree of the House of Normandy:



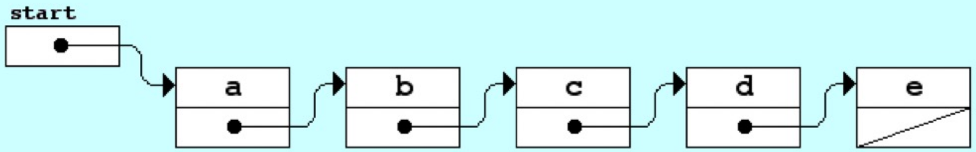
- This example is useful for defining terminology:
  – William I is the ***root*** of the tree.
  – Adela is a ***child*** of William I and the ***parent*** of Stephen.
  – Robert, William II, Adela, and Henry I are ***siblings***.
  – Henry II is a ***descendant*** of William I, Henry I, and Matilda.
  – William I is an ***ancestor*** of everyone else in this tree.

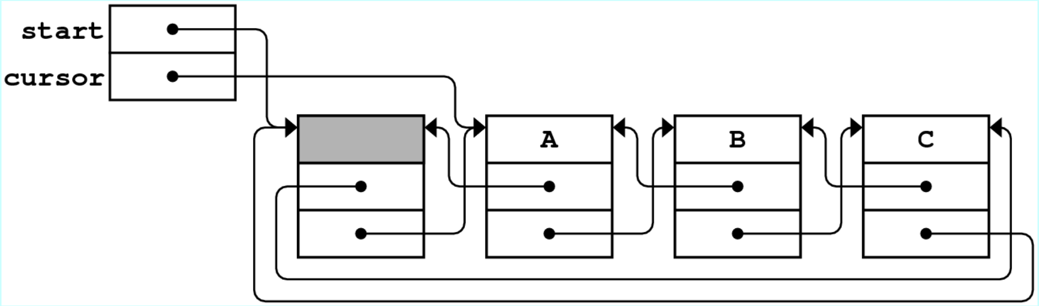
Trees as a Recursive Data Structure

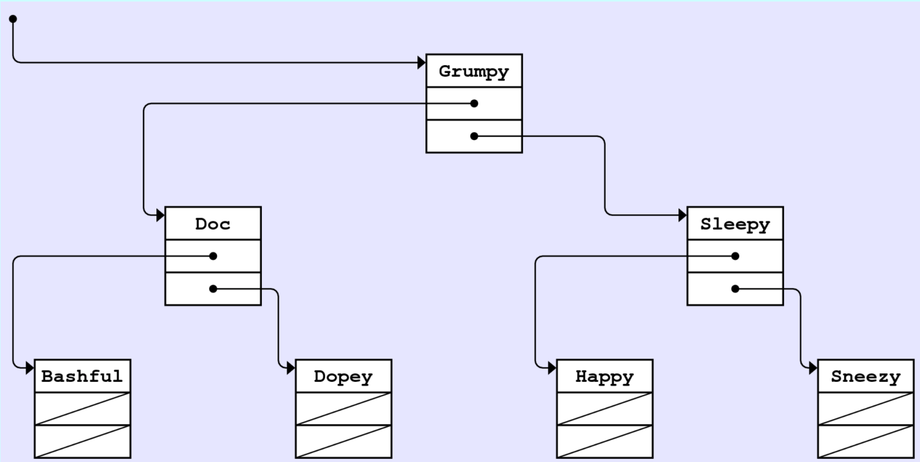
- If you think about trees as a programmer, the following definition is extremely useful:
  - A ***tree*** is a pointer to a node.
  - A ***node*** is a structure that contains some number of trees.
- Although this definition is clearly circular, it is not necessarily infinite, either because
  - Tree pointers can be **`NULL`** indicating an empty tree.
  - Nodes can contain an empty list of children.
- In C++, programmers typically define a structure or object type to represent a node and then use an explicit pointer type to represent the tree.

```
struct TreeNode {
   string key;
   Vector<TreeNode *> children;
};
```

Pointers to constuct complex structures

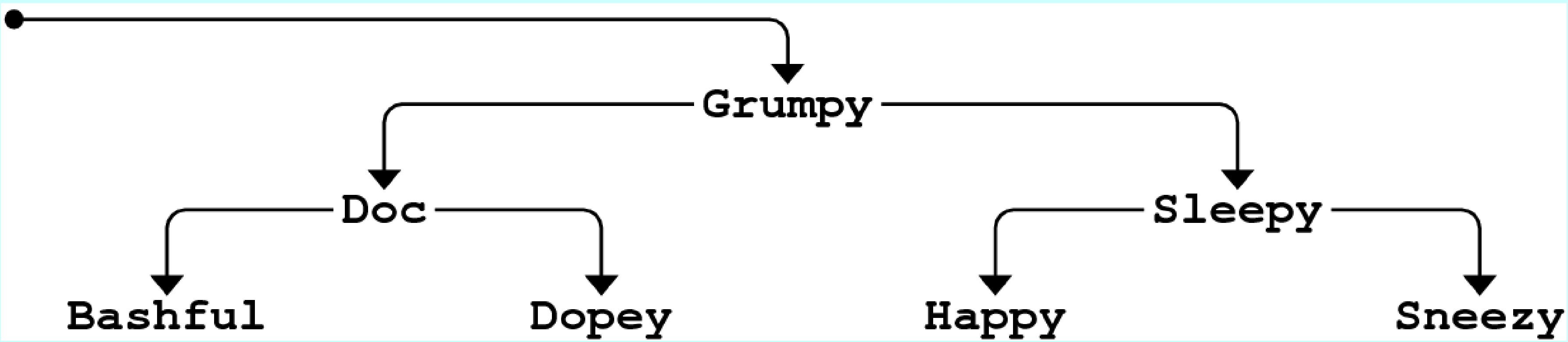






Binary Search Trees

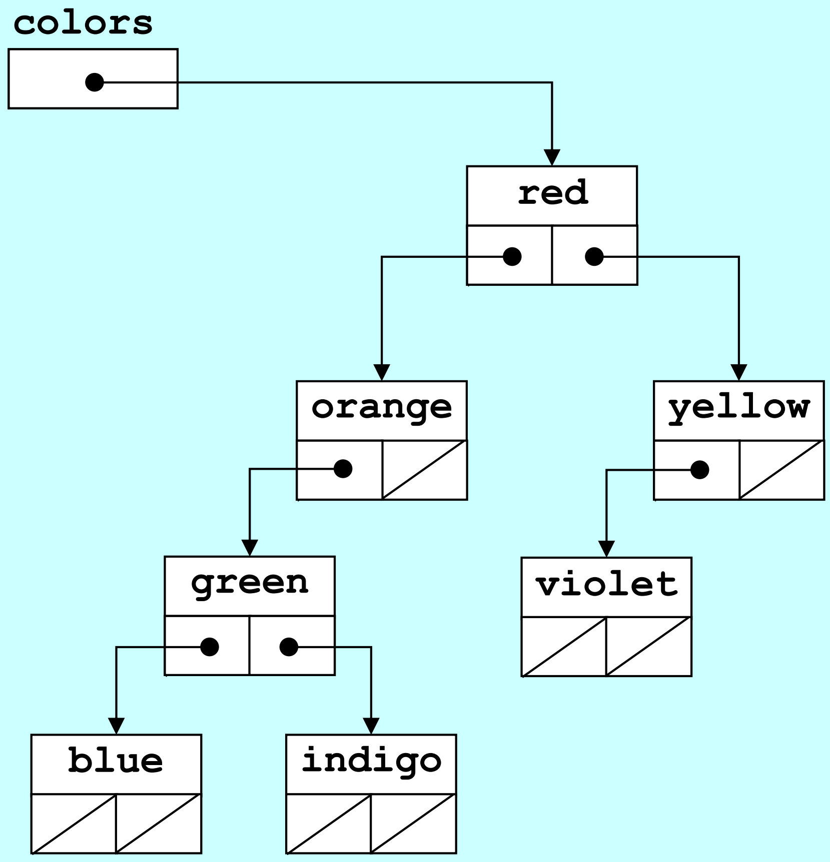
- The tree that supports the implementation of the **`Map`** class is called a ***binary search tree*** (or ***BST*** for short). Each node in a BST has exactly two subtrees: a ***left subtree*** that contains all the nodes that come before the current node and a ***right subtree*** that contains all the nodes that come after it. Either or both of these subtrees may be **`NULL`**.
- The classic example of a binary search tree uses the names from Walt Disney's *Snow White and the Seven Dwarves*:



Exercise: Building a Binary Search Tree

Diagram the BST that results from executing the following code:

```
Node *colors = NULL;
insertNode(colors, "red");
insertNode(colors, "orange");
insertNode(colors, "yellow");
insertNode(colors, "green");
insertNode(colors, "blue");
insertNode(colors, "indigo");
insertNode(colors, "violet");
```



A Simple BST Implementation

- To get a sense of how binary search trees work, it is useful to start with a simple design in which keys are always strings.
- Each node in the tree is then a structure containing a key and two subtrees, each of which is either **`NULL`** or a pointer to some other node. This design suggests the following definition:

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

- The code for finding a node in a tree begins by comparing the desired key with the key in the root node. If the strings match, you've found the correct node; if not, you simply call yourself recursively on the left or right subtree depending on whether the key you want comes before or after the current one.

A Simple BST Implementation

```
/*
 * Type: Node
 * ----------
 * This type represents a node in the binary search tree.
 */

struct Node {
   string key;
   Node *left, *right;
};

/*
 * Function: findNode
 * Usage: Node* node = findNode(t, key);
 * -------------------------------------
 * Returns a pointer to the node in the binary search tree t than contains
 * a matching key.  If no such node exists, findNode returns NULL.
 */

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

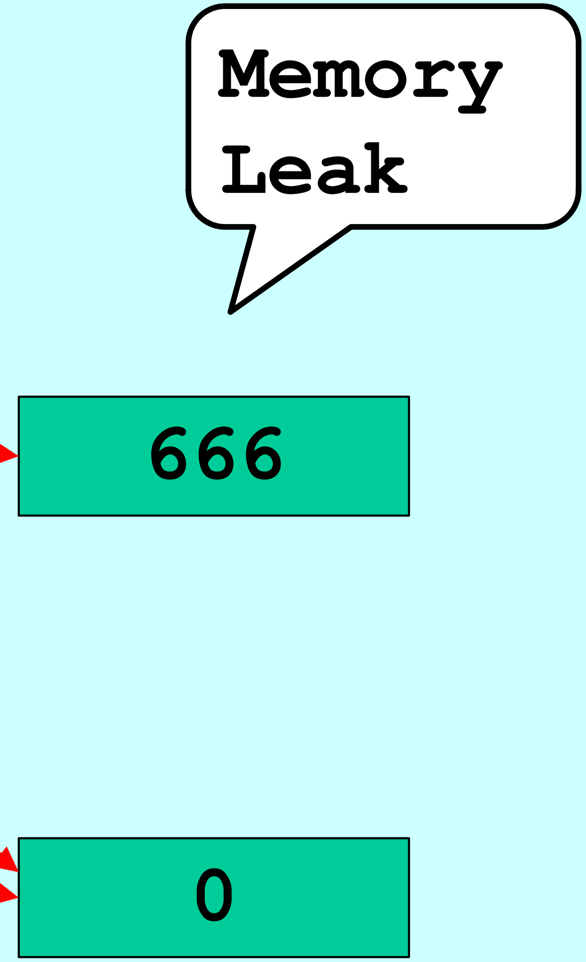
A Simple BST Implementation

```
/*
 * Function: insertNode
 * Usage: insertNode(t, key);
 * --------------------------
 * Inserts the specified key at the appropriate location in the
 * binary search tree rooted at t.  Note that t must be passed
 * by reference, since it is possible to change the root.
 */

void insertNode(Node* & t, string key) {
   if (t == NULL) {
      t = new Node;
      t->key = key;
      t->left = t->right = NULL;
      return;
   }
   if (key == t->key) return;
   if (key < t->key) {
      insertNode(t->left, key);
   } else {
      insertNode(t->right, key);
   }
}
```

Call a pointer by reference. Can you use call by pointer here?

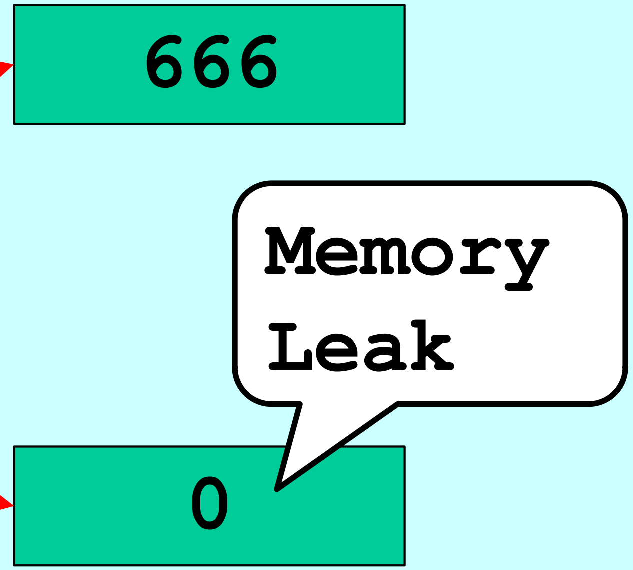
foo1() ?

```
void test1(int *p) {
    p = new int {666};
}

void foo1() {
    int *p = new int {0};

    test1(p);

    cout << *p << endl;

}
```



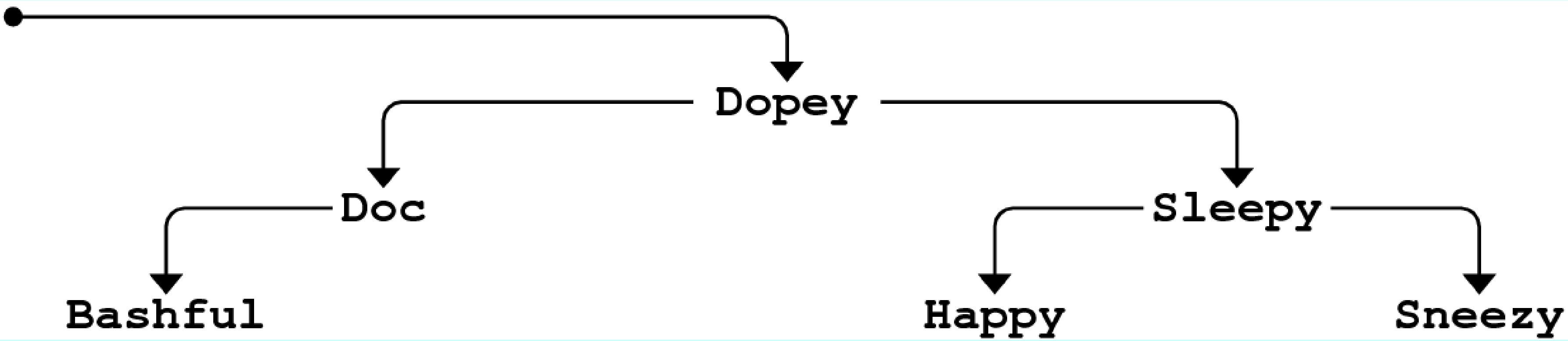
foo2() ?

```
void test2(int * & p) {
    p = new int {666};
}

void foo2() {

    int *p = new int {0};

    test2(p);

    cout << *p << endl;
}
```



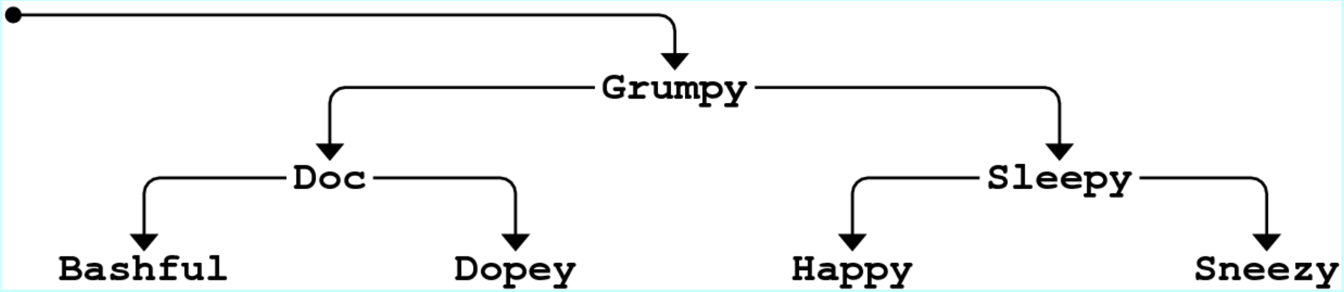
Removing Nodes in Binary Search Trees

- Sneezy - For a leaf node (with no children), replace the pointer to the node with a NULL pointer;
- Sleepy - If either child of the node you want to remove is NULL, replace it with its non-NULL child;
- Grumpy - If you try to remove a node with both a left and a right child, replace it with the rightmost node in the left subtree (predecessor) or the leftmost node in the right subtree (successor). Additional operations might be needed.



A Question of Balance

- Ideally, a binary search tree containing the names of Disney's seven dwarves would look like this:

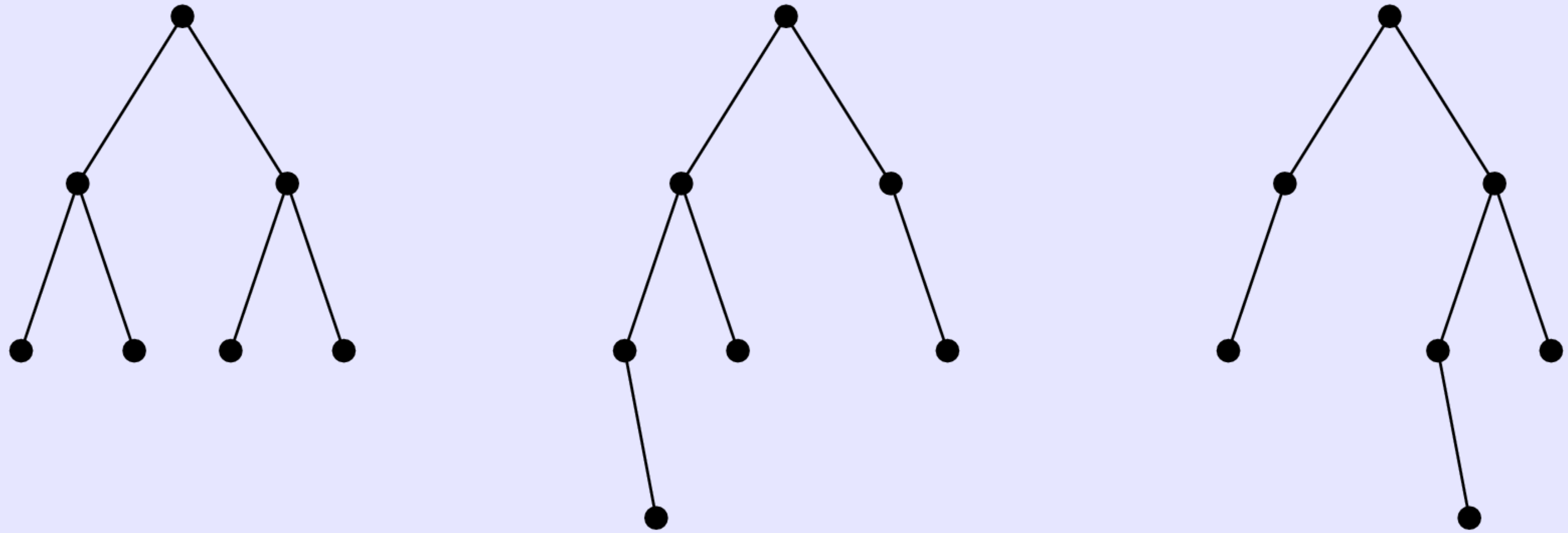


- If, however, you happened to enter the names in alphabetical order, this tree would end up being a simple linked list in which all the left subtrees were **NULL** and the right links formed a simple chain. Algorithms on that tree would run in $O(N)$ time instead of $O(\log N)$ time.

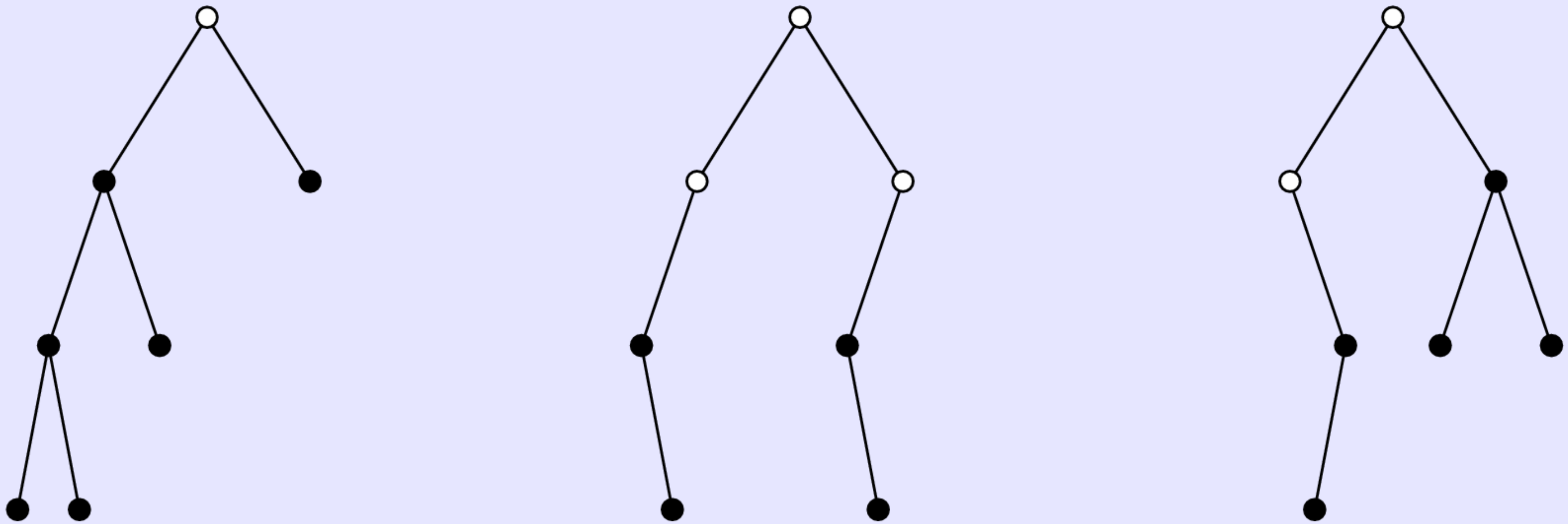
- A binary search tree is ***balanced*** if the height of its left and right subtrees differ by at most one and if both of those subtrees are themselves balanced.

FIGURE 16-5 **Examples of balanced and unbalanced binary trees**

Balanced trees:

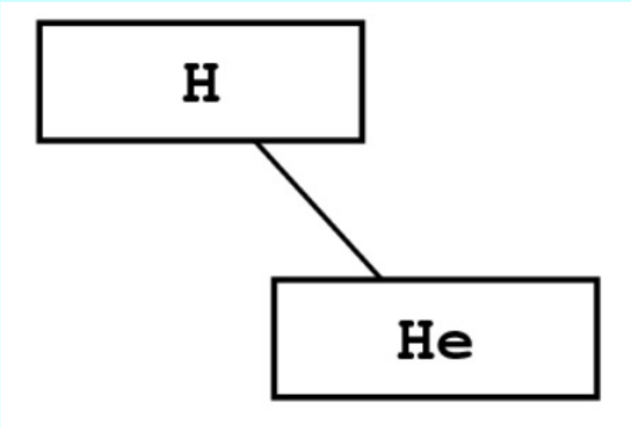


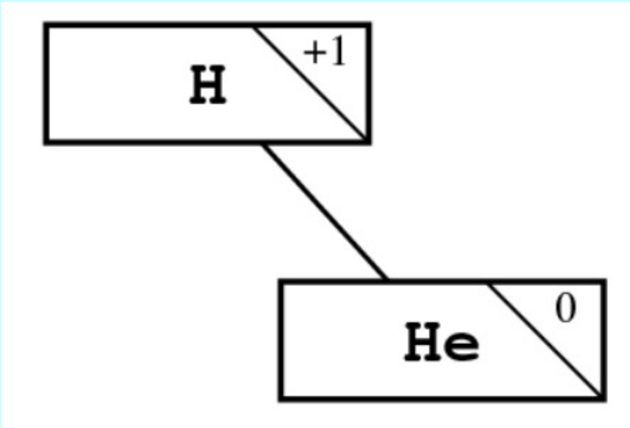
Unbalanced trees:



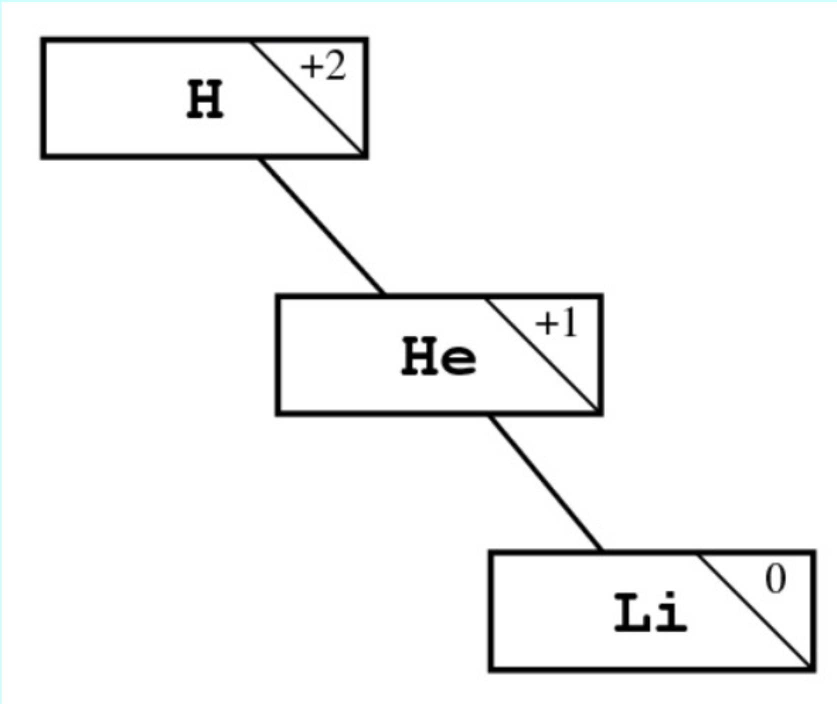
balance factor

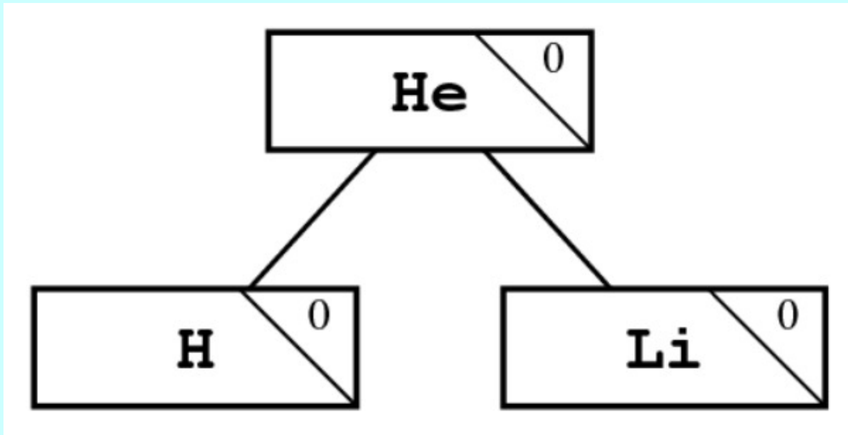
- height of the right subtree minus the height of the left subtree





balance factor - examples





balance factor - example

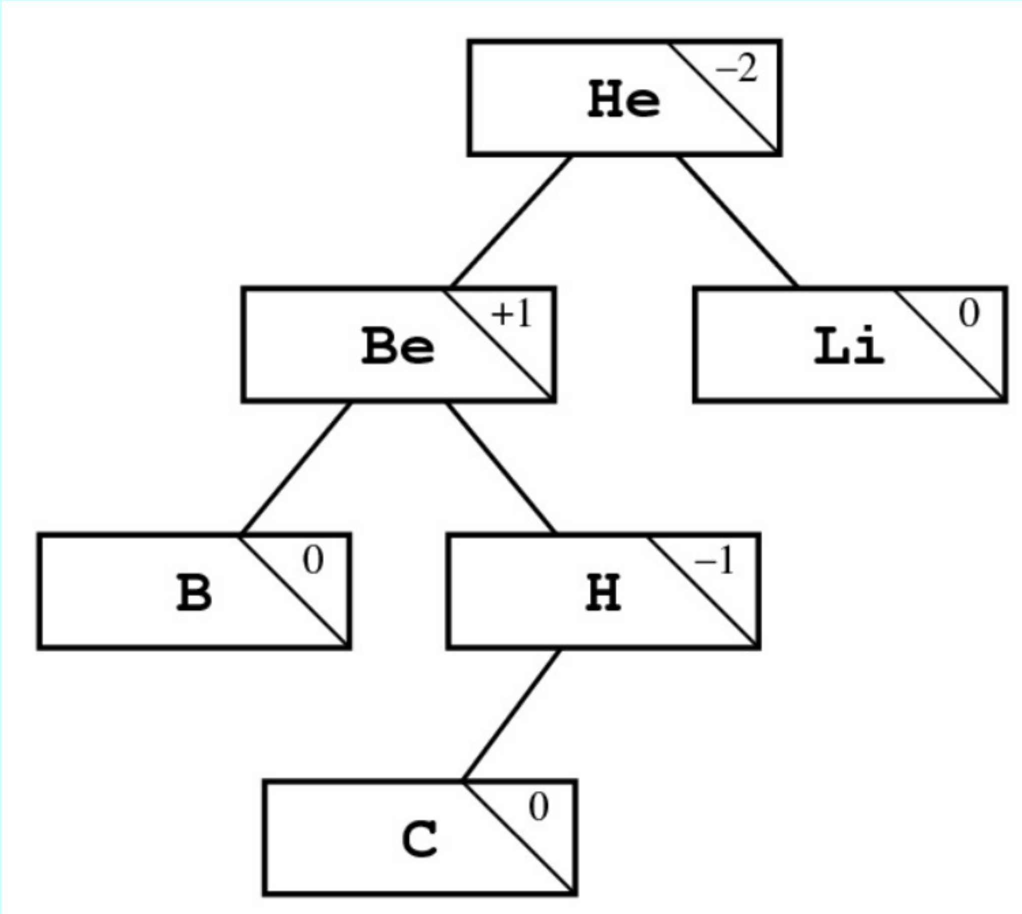
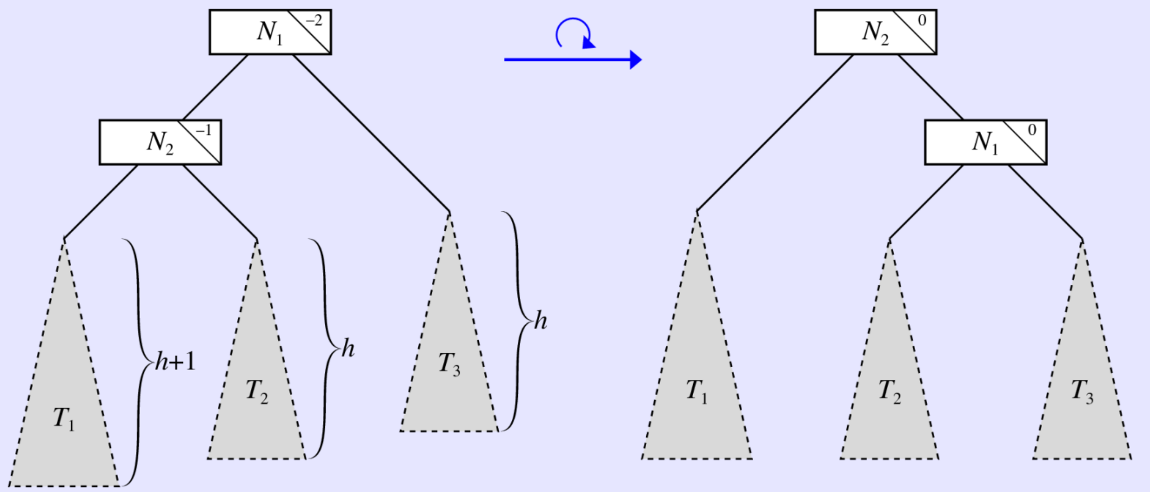
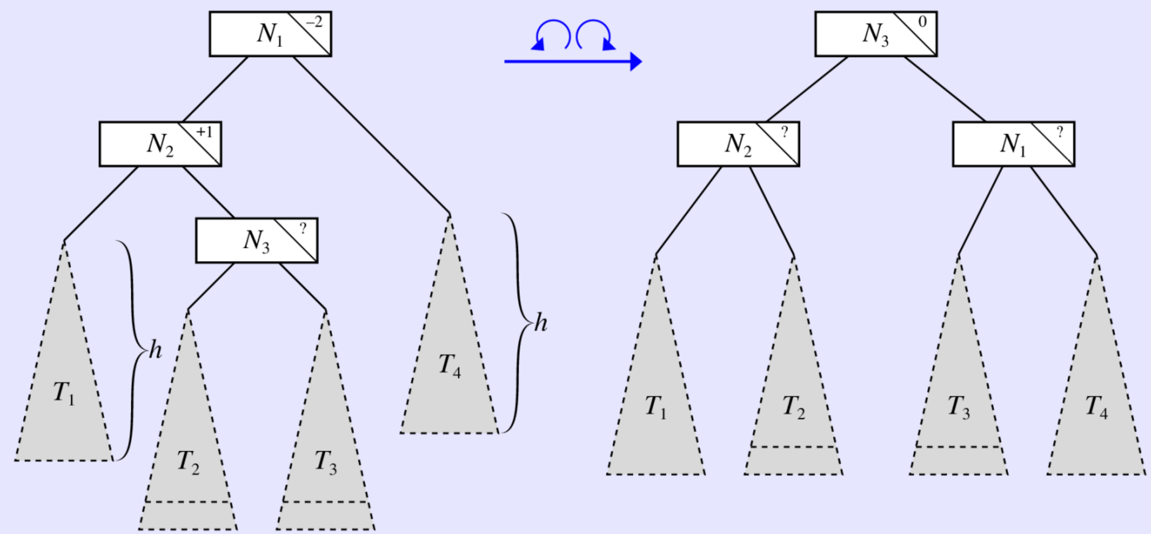


FIGURE 16-7 **Rotation operations in an AVL tree**

Single rotation

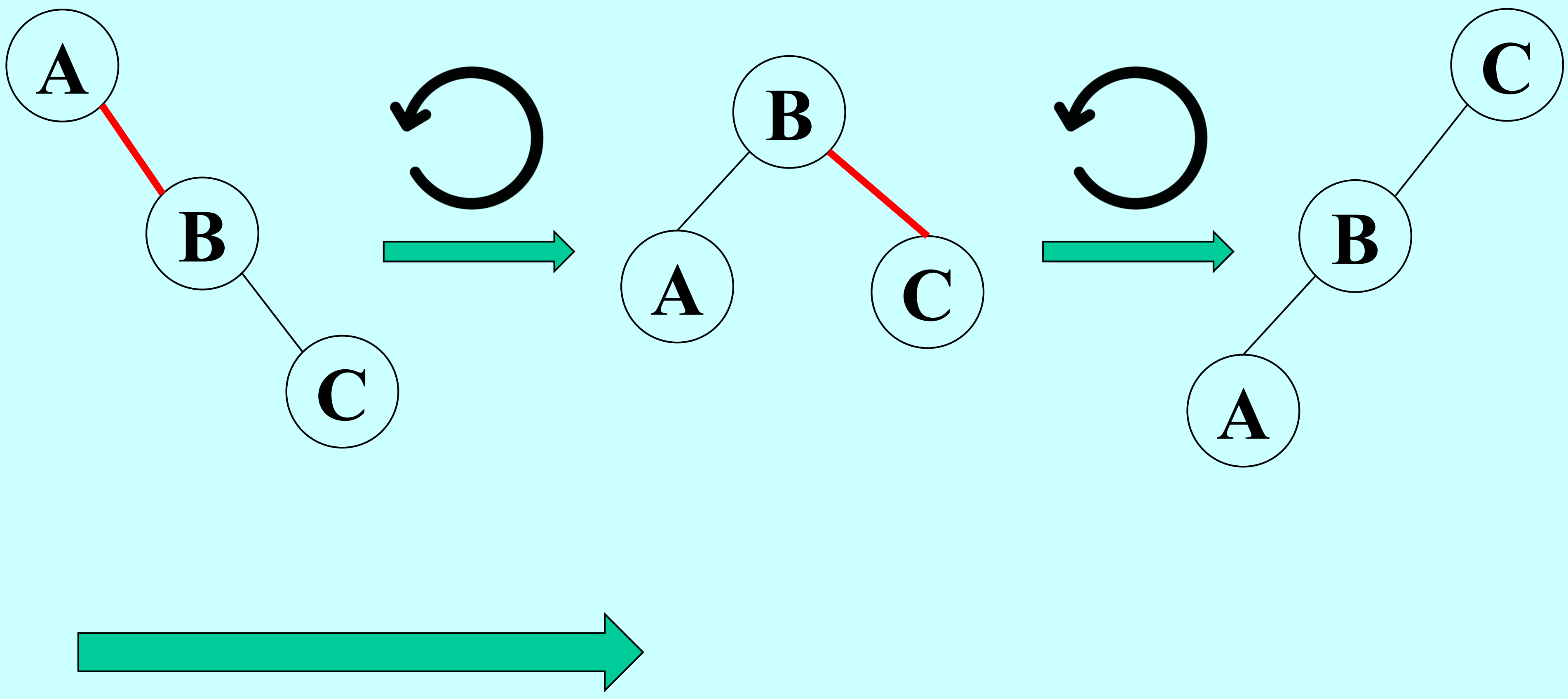


Double rotation

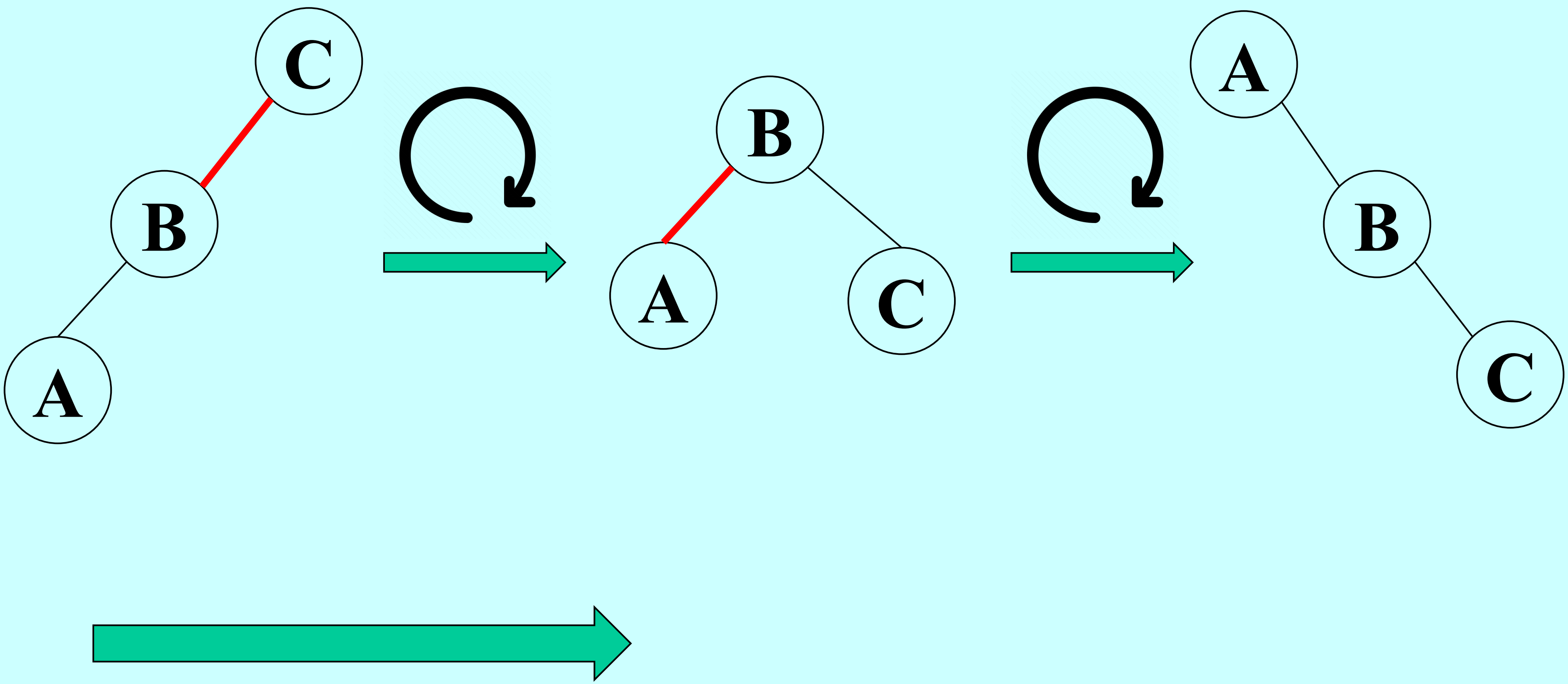


Note: At least one of the subtrees T_2 and T_3 must have height h, the other can have height h or h–1. The balance factors in the final nodes will need to be adjusted to take account of any difference in height.

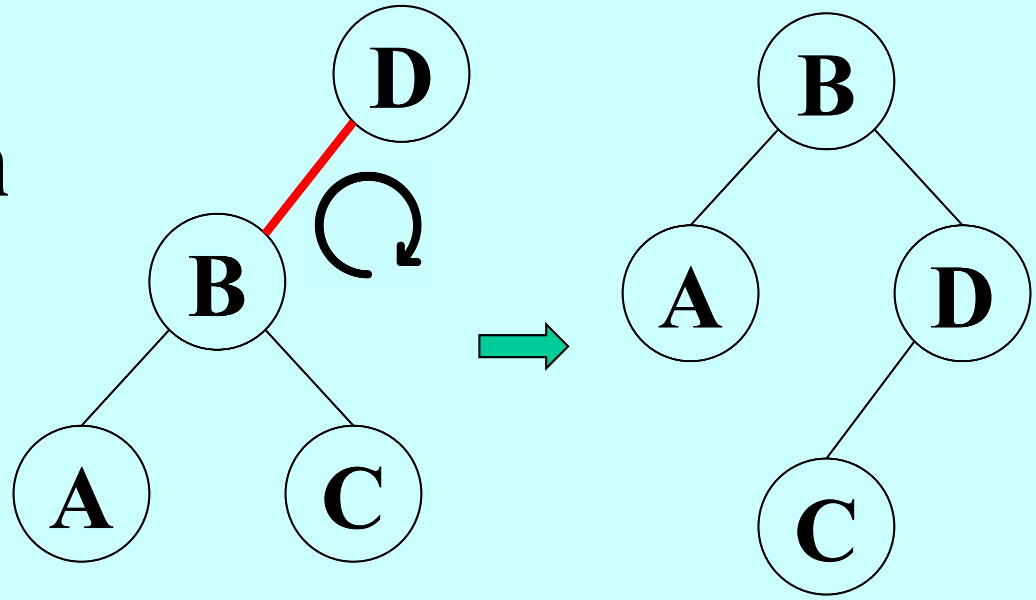
Tree Rotation – Left

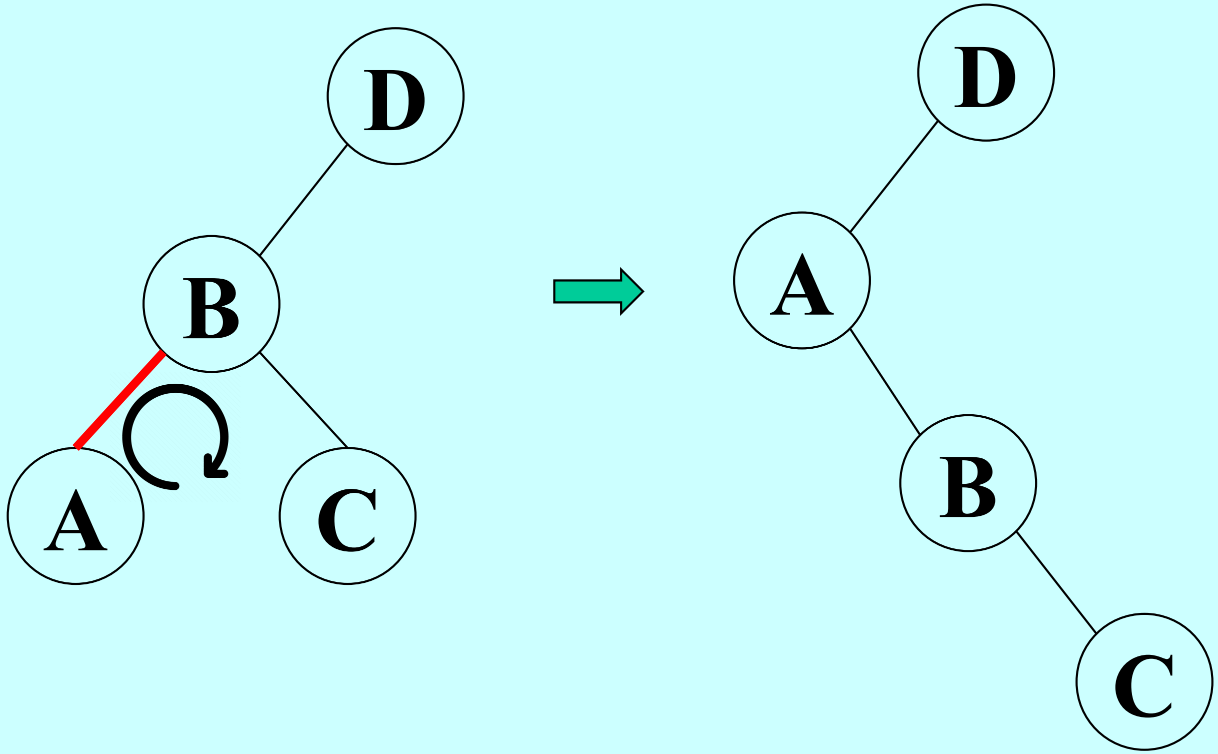


Tree Rotation – Right

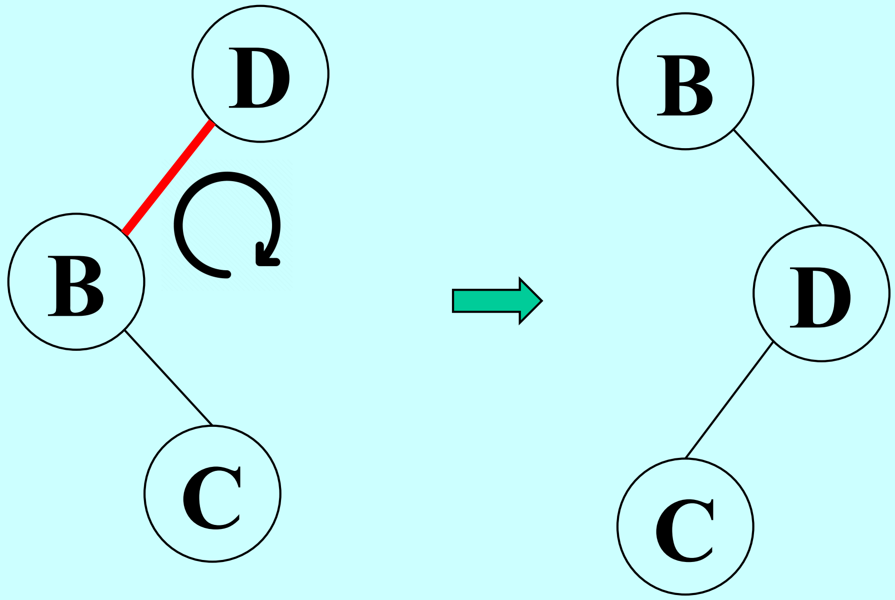


Right Rotation

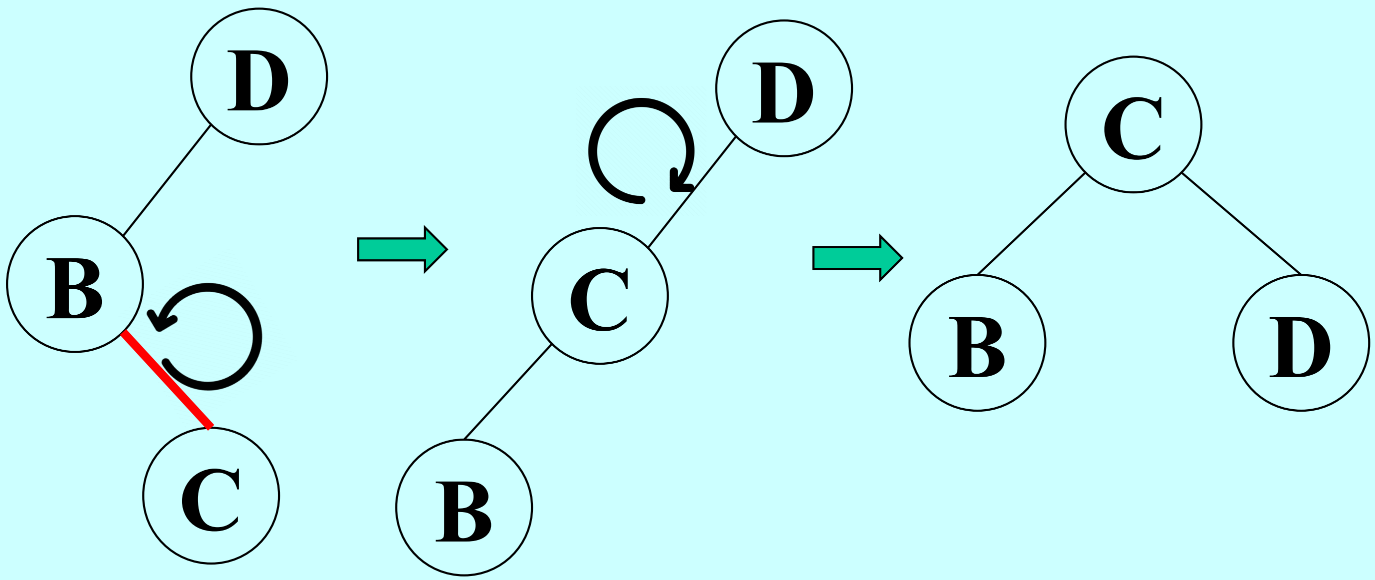




Right Rotation



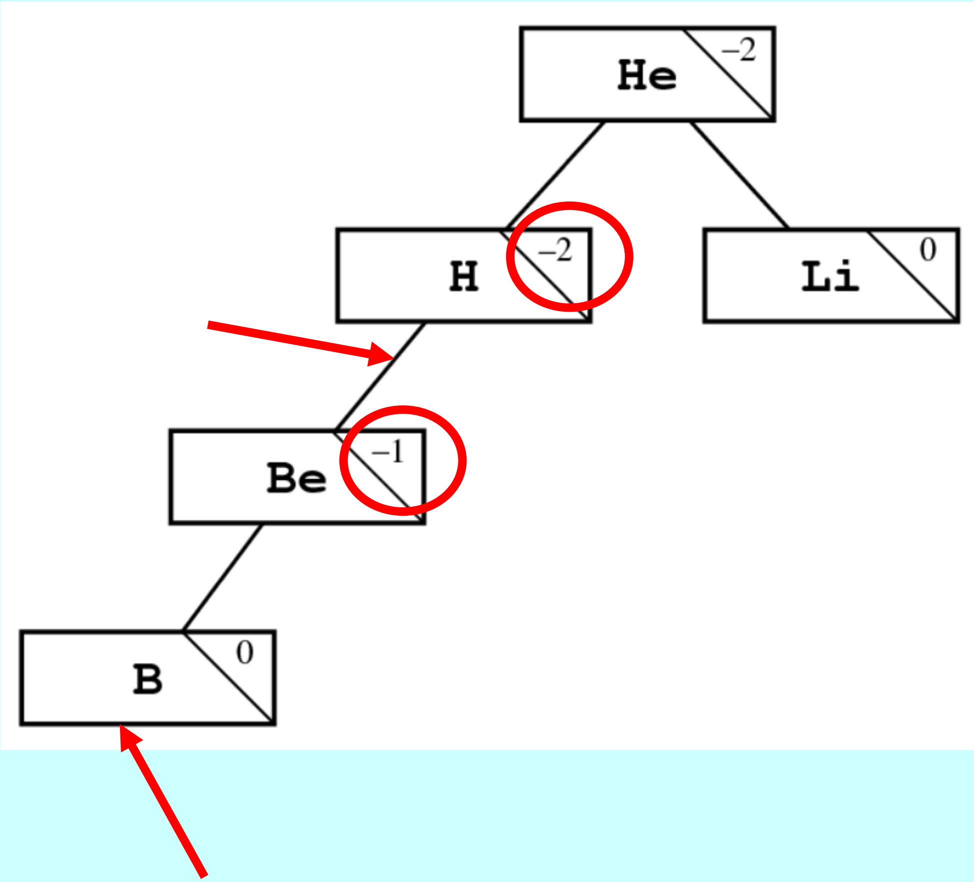
Double Rotation



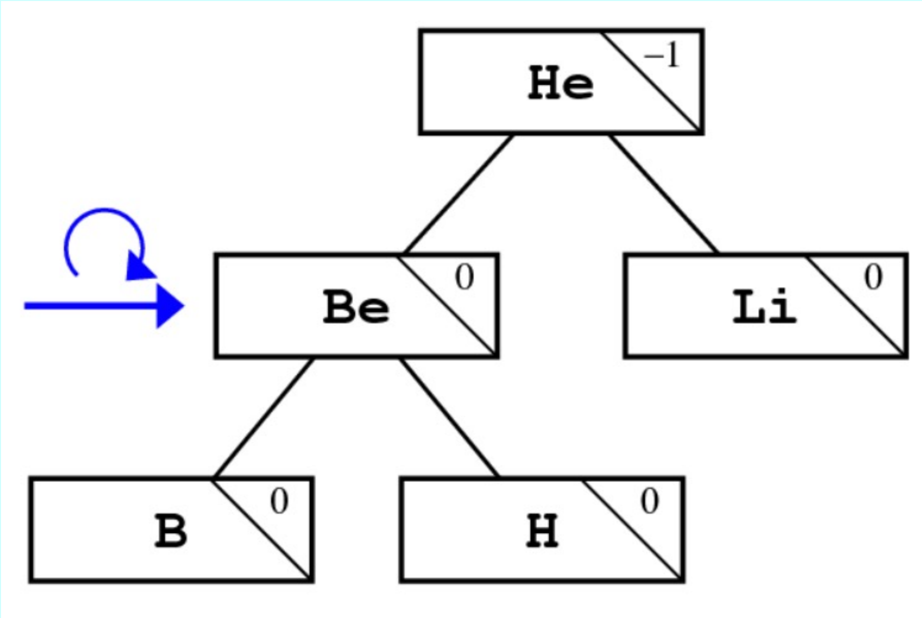
When to Single or Double ?

1. Locate the branch (axis) of the subtree that becomes out of balance (balance factor > 1) after inserting a new node
2. Compare signs of balance factors of the 2 nodes connected by the branch
3. Single Rotation = same sign (-,-) or (+,+)
4. Double Rotation = opp. sign (+,-) or (-,+)

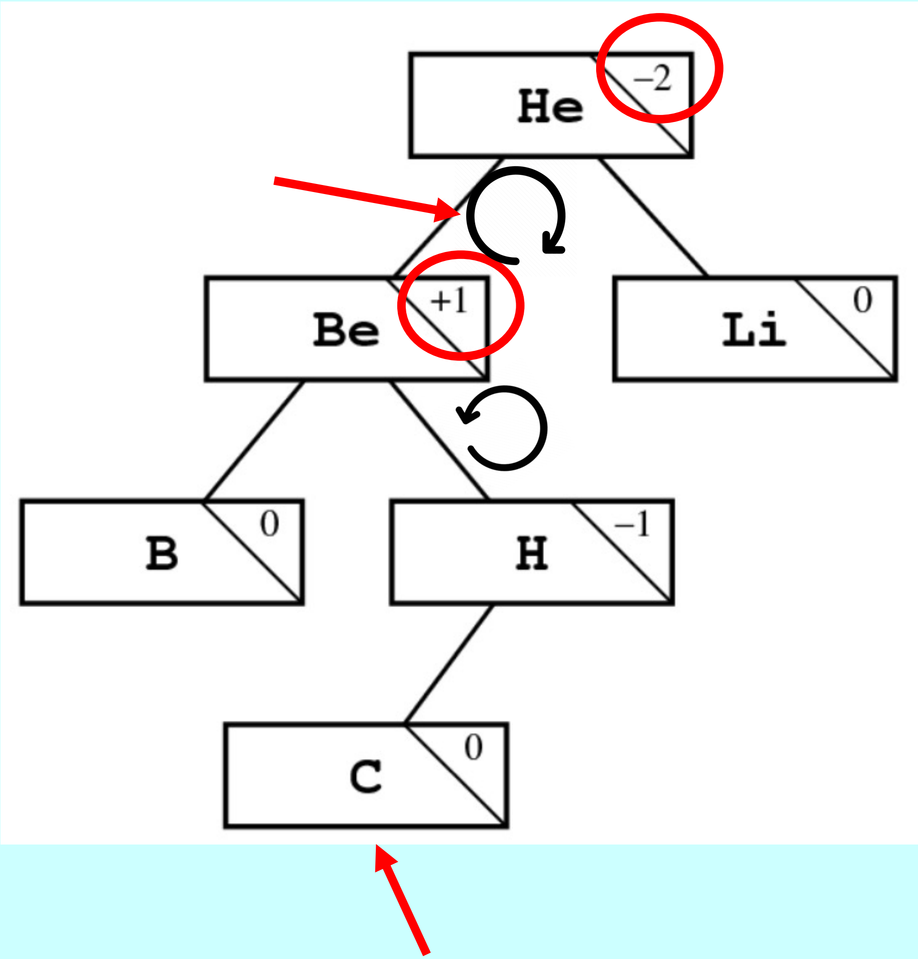
Identify the branch



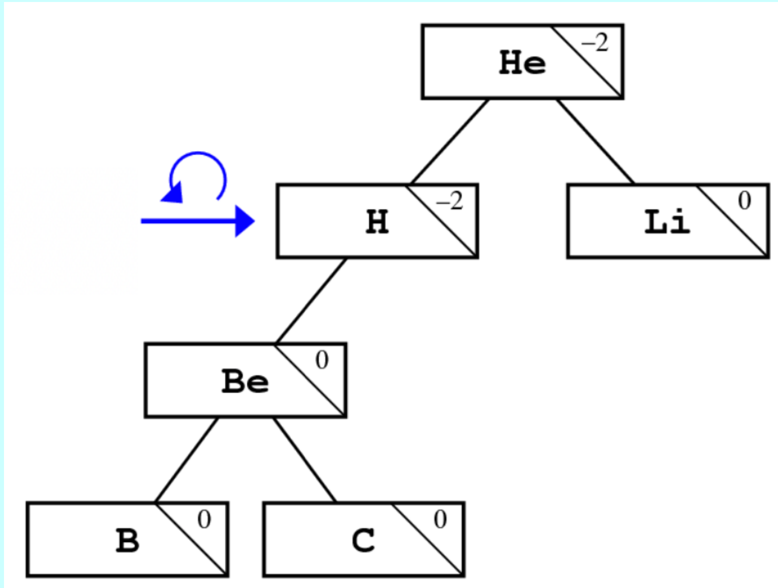
New Node

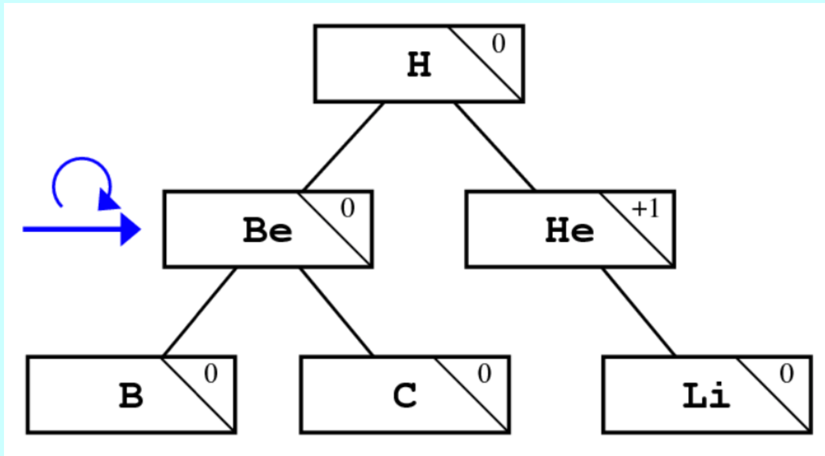


Identify the branch



New Node





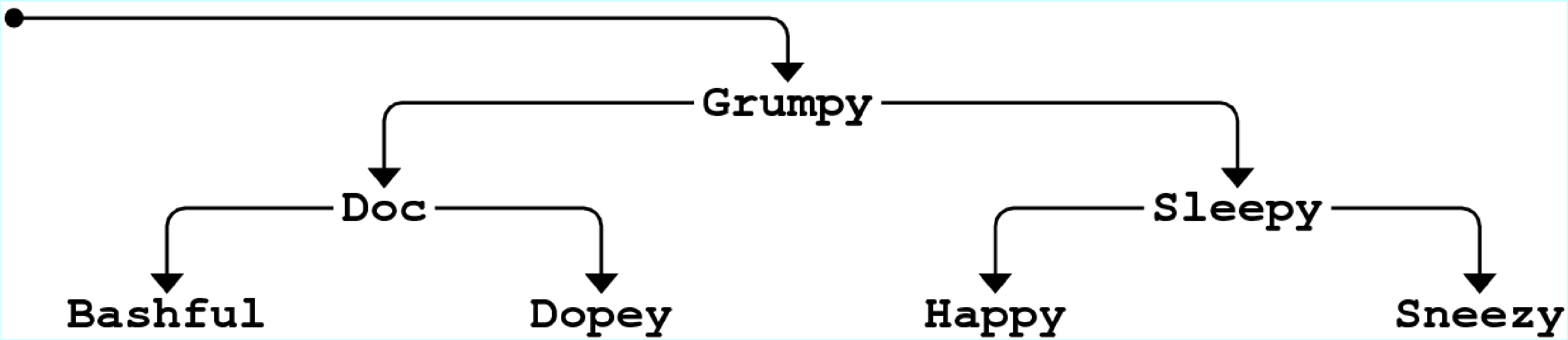
Tree-Balancing Algorithms

- The AVL algorithm was the first tree-balancing strategy and has been superseded by newer algorithms that are more effective in practice. These algorithms include:
  - Red-black trees
  - 2-3 trees
  - AA trees
  - Fibonacci trees
  - Splay trees
- At this point in your study of computer science, the important thing to know is that it is *possible* to keep a binary tree balanced as you insert nodes, thereby ensuring that lookup operations run in $O(\log N)$ time. If you get really excited about this kind of algorithms, you'll have the opportunity to study them in more detail in the later data structures and algorithms course.

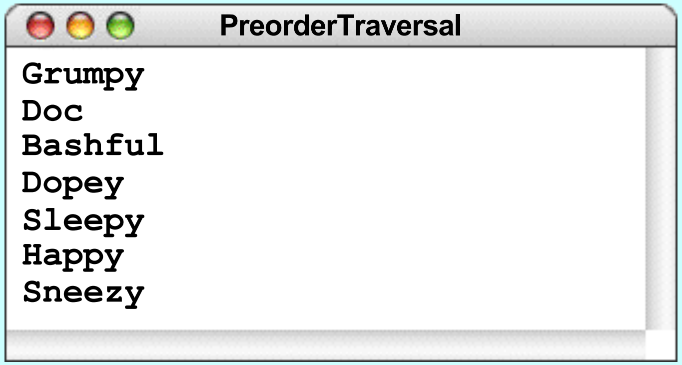
Tree Traversal Strategies

- It is easy to write a function that performs some operation for every key in a binary search tree, because recursion makes it simple to apply that operation to each of the subtrees.
- The order in which keys are processed depends on when you process the current node with respect to the recursive calls:
  - If you process the current node before either recursive call, the result is a ***preorder traversal***.
  - If you process the current node after the recursive call on the left subtree but before the recursive call on the right subtree, the result is an ***inorder traversal***. In the case of the simple BST implementation that uses strings as keys, the keys will appear in lexicographic order.
  - If you process the current node after completing both recursive calls, the result is a ***postorder traversal***. Postorder traversals are particularly useful if you are trying to free all the nodes in a tree.

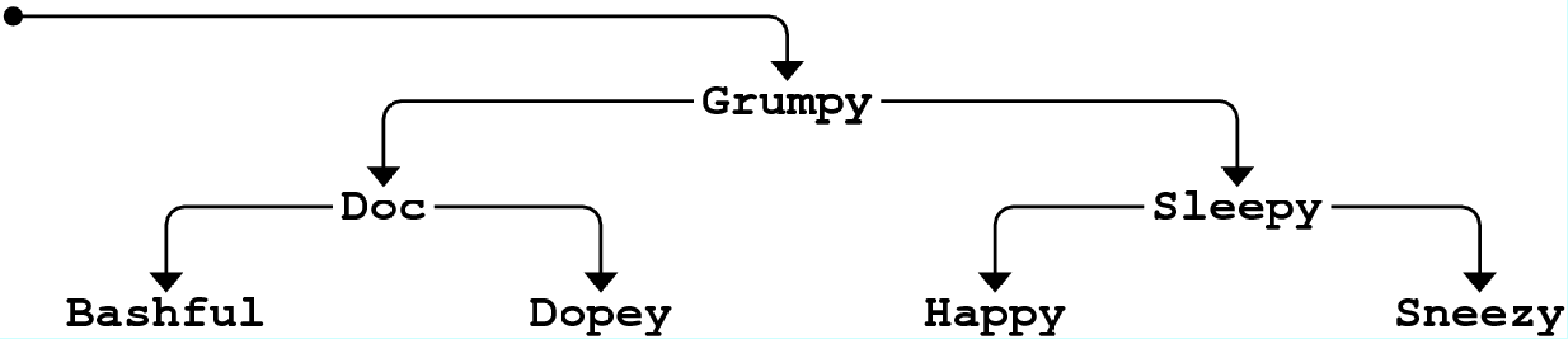
Exercise: Preorder Traversal



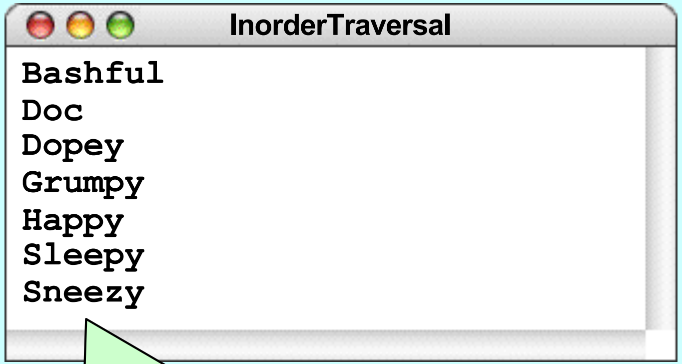

```
void preorderTraversal(Node *t) {
   if (t != null) {
      cout << t->key << endl;
      preorderTraversal(t->left);
      preorderTraversal(t->right);
   }
}
```



Exercise: Inorder Traversal

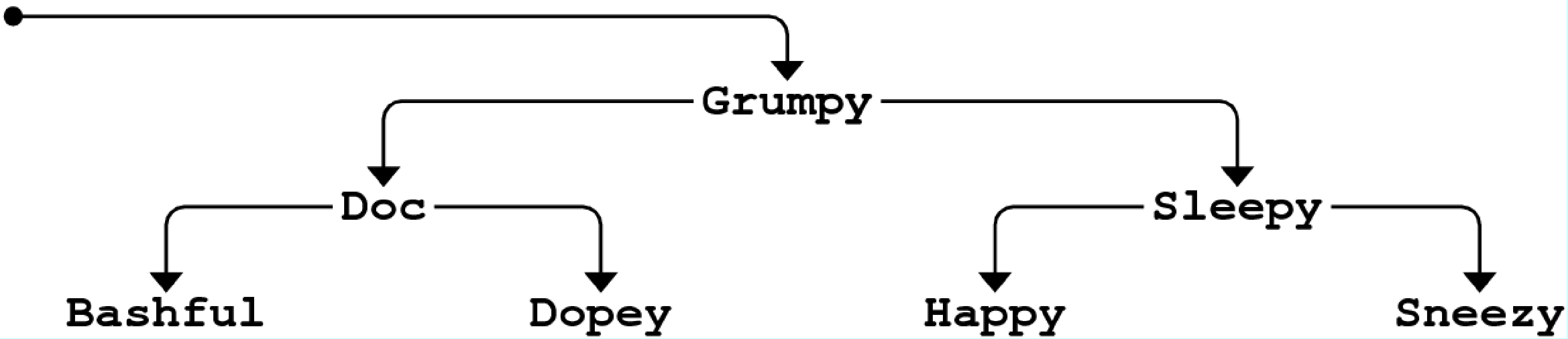



```
void inorderTraversal(Node *t) {
   if (t != null) {
      inorderTraversal(t->left);
      cout << t->key << endl;
      inorderTraversal(t->right);
   }
}
```

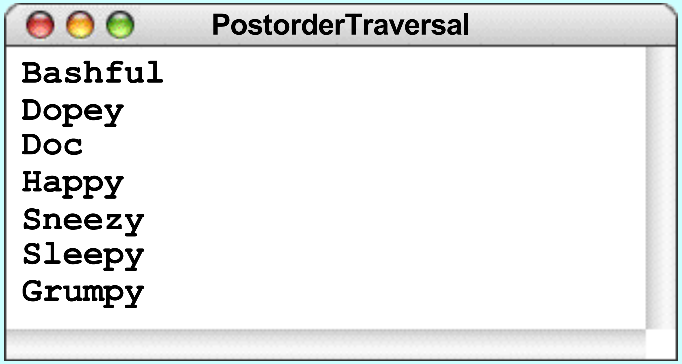


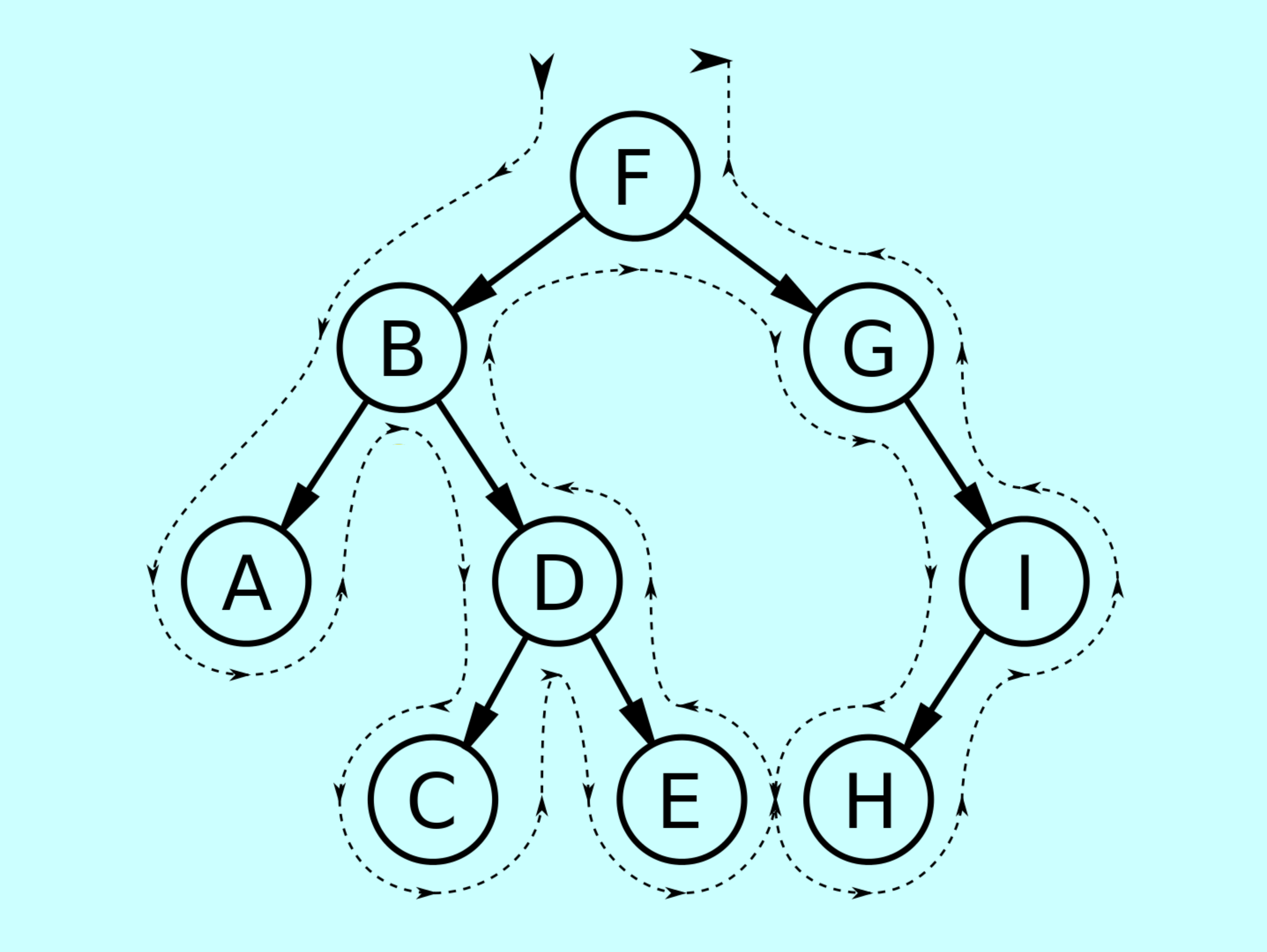
An inorder traversal of a BST will always process the nodes in alphabetical order. This is one of the reasons to choose tree-based map over hash map.

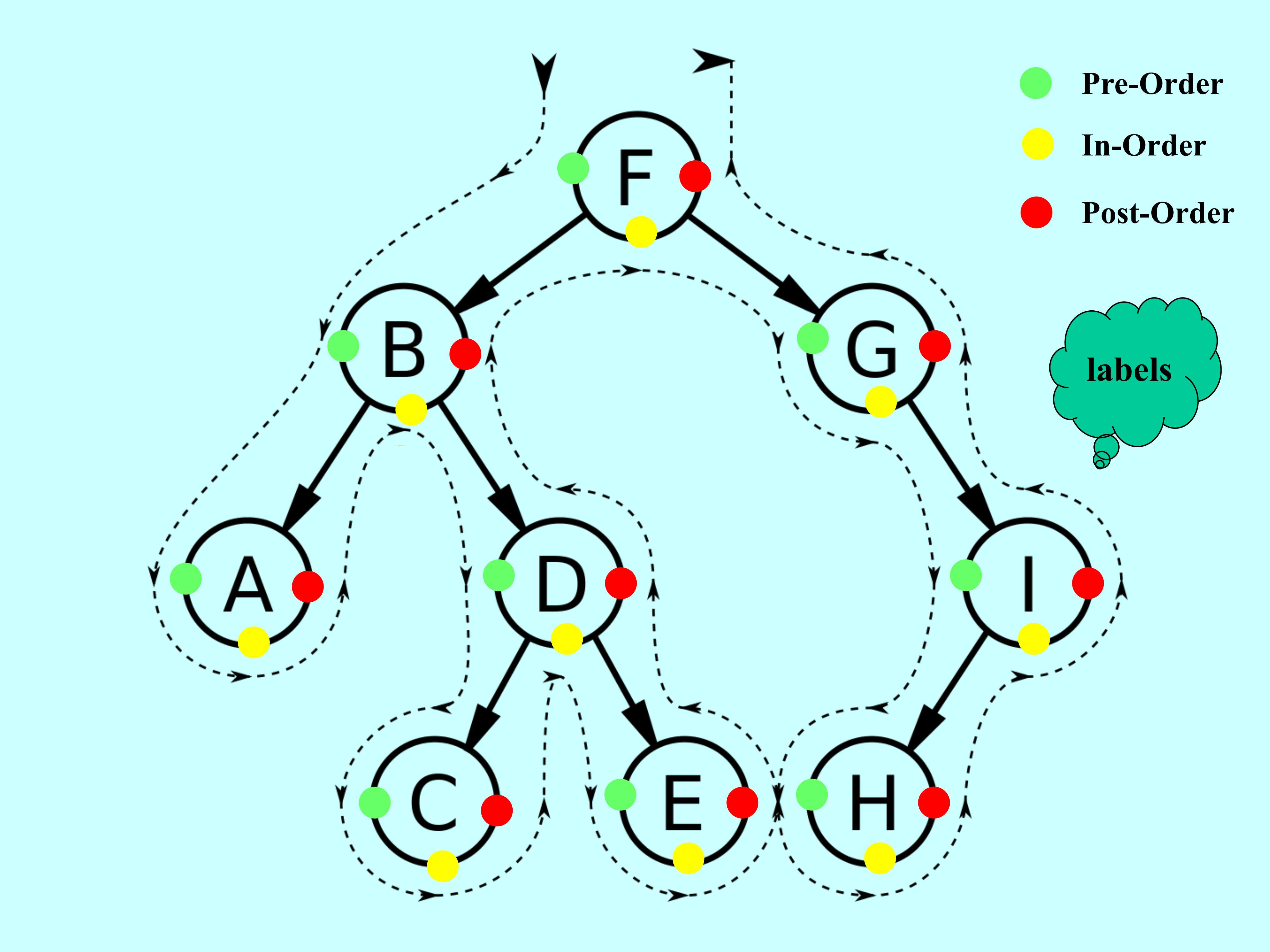
Exercise: Postorder Traversal




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

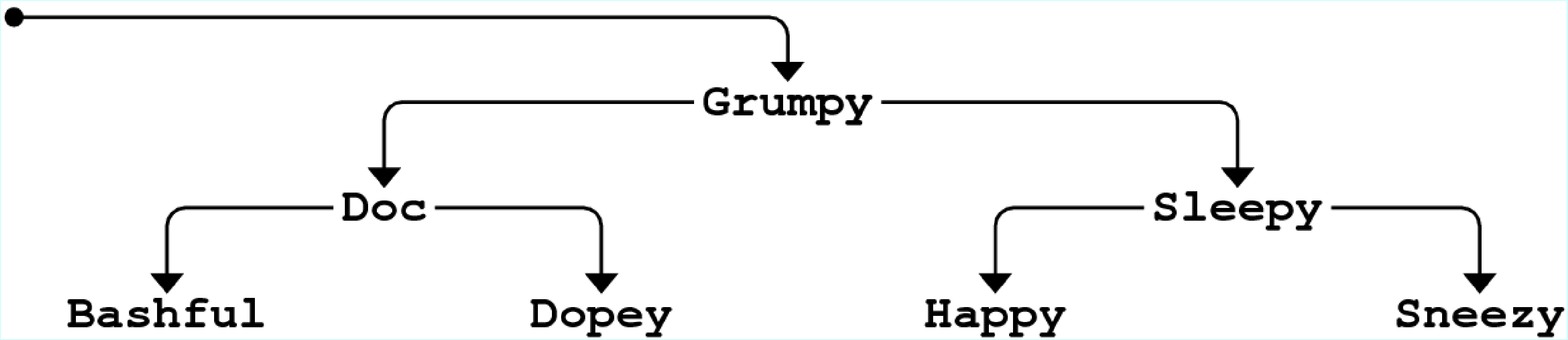


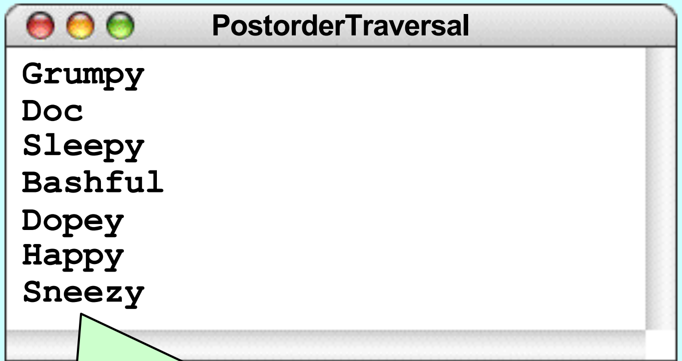

F
B
G
A
D
I
C
E
H



F
B
G
A
D
I
C
E
H
Pre-Order
In-Order
Post-Order
labels

Exercise: Level-order Traversal





Although the previous traversals process the nodes in different orders, they all reach the nodes in the same order (depth first). Level-order traversal, however, reaches the nodes in a different order (breadth first).

Partially Ordered Trees

- A partially ordered tree is a special class of binary tree (but not a binary search tree) with these additional properties:
  - The tree is ***complete***, i.e., it is completely balanced, and each level of the tree is filled as far to the left as possible.
  - The root node of the tree has higher priority than the root of either of its subtrees, which are also partially ordered trees.
- A ***heap*** is an array-based data structure that simulates the operation of a partially ordered tree. (The heap data structure bears no relationship to the pool of unused memory "heap" available for dynamic allocation.)
- The heap stores the nodes in a partially ordered tree simply by counting off the nodes, level by level, from left to right, making it simple to implement tree operations:

`parentIndex(n)`	*is always given by*	**`(n - 1) / 2`**
`leftChildIndex(n)`	*is always given by*	**`2 * n + 1`**
`rightChildIndex(n)`	*is always given by*	**`2 * n + 2`**

Exercise: Partially Ordered Trees

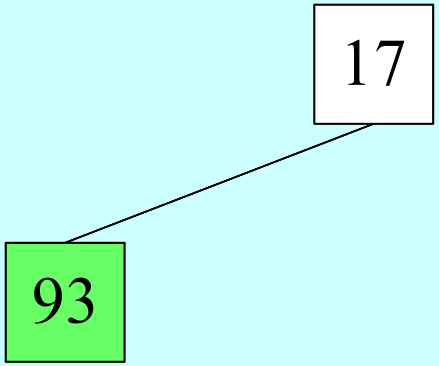
Insert in order: 17 93 20 42 68 11

The heap:

17					

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

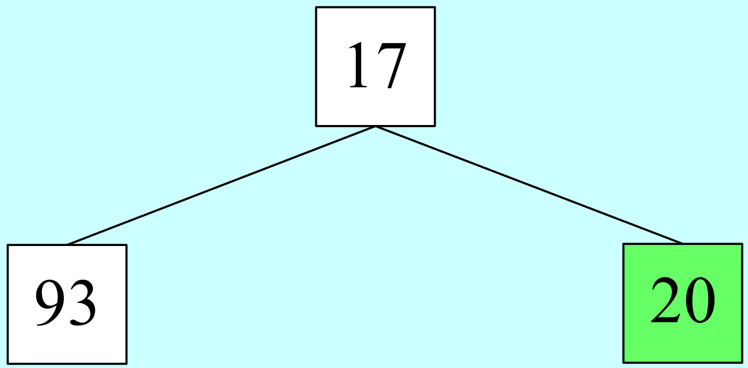


The heap:

17	93				

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

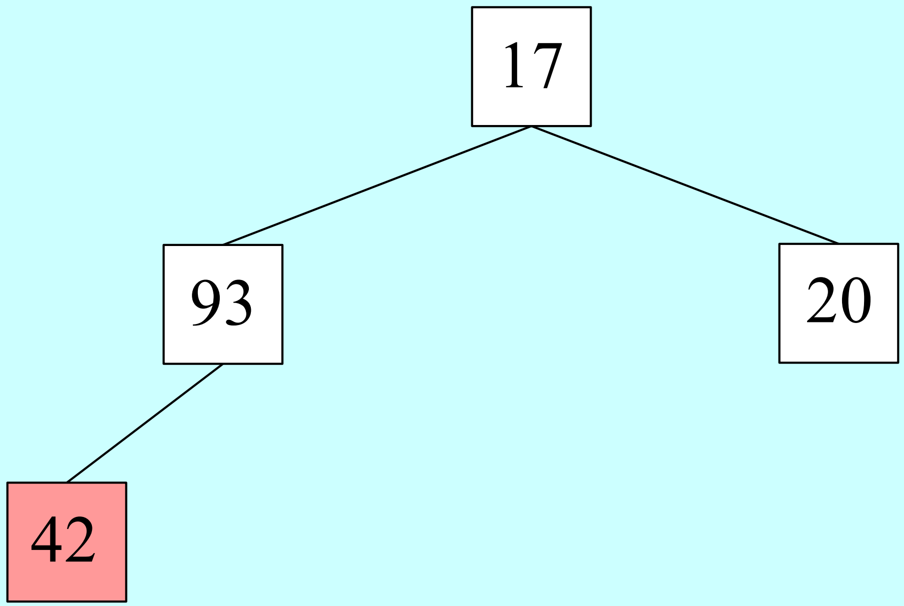


The heap:

17	93	20			

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

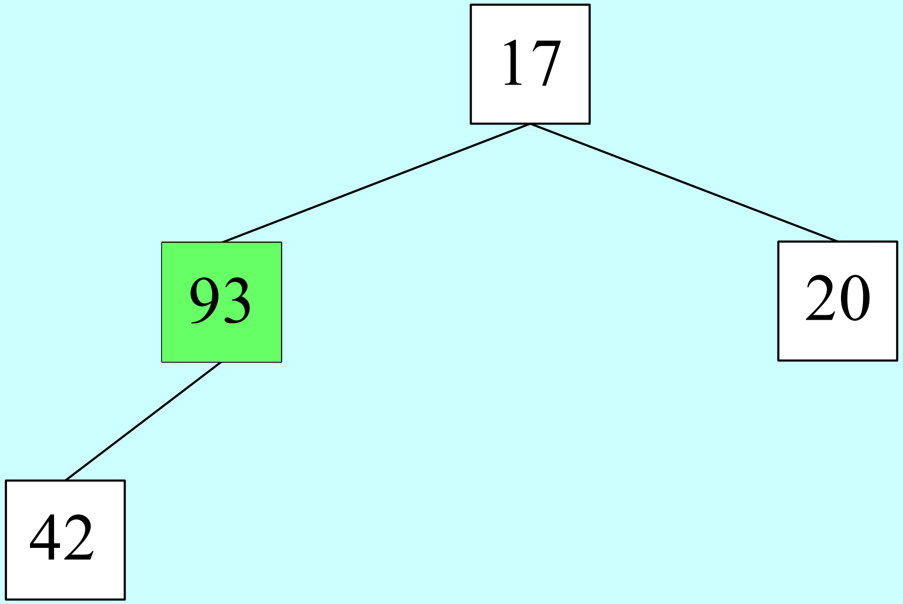


The heap:

17	93	20	42		

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

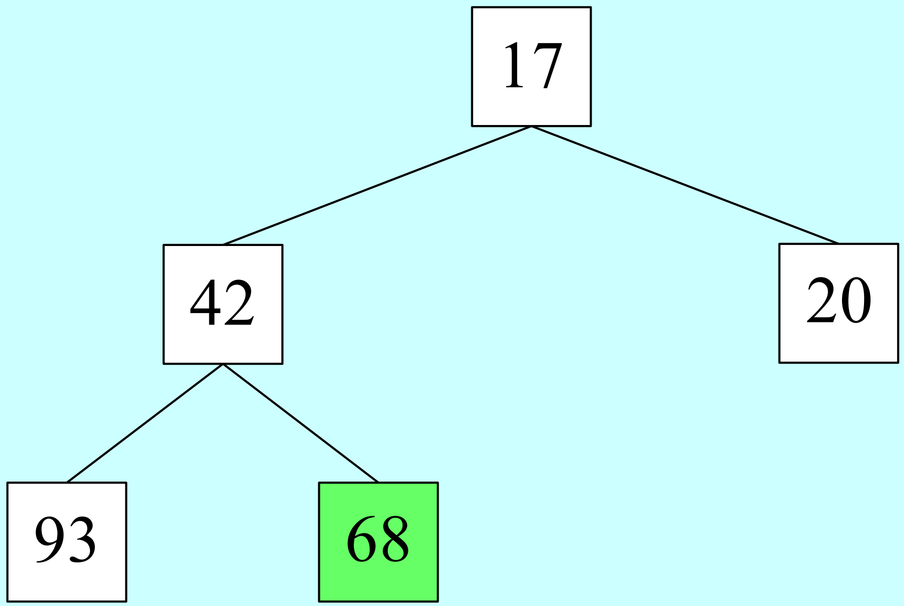


The heap:

17	42	20	93		

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

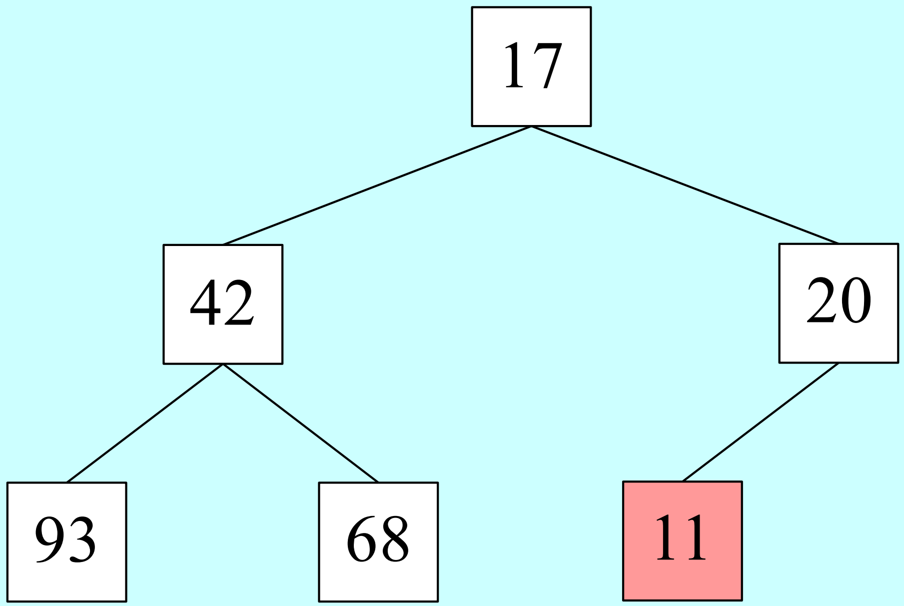


The heap:

17	42	20	93	68	

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

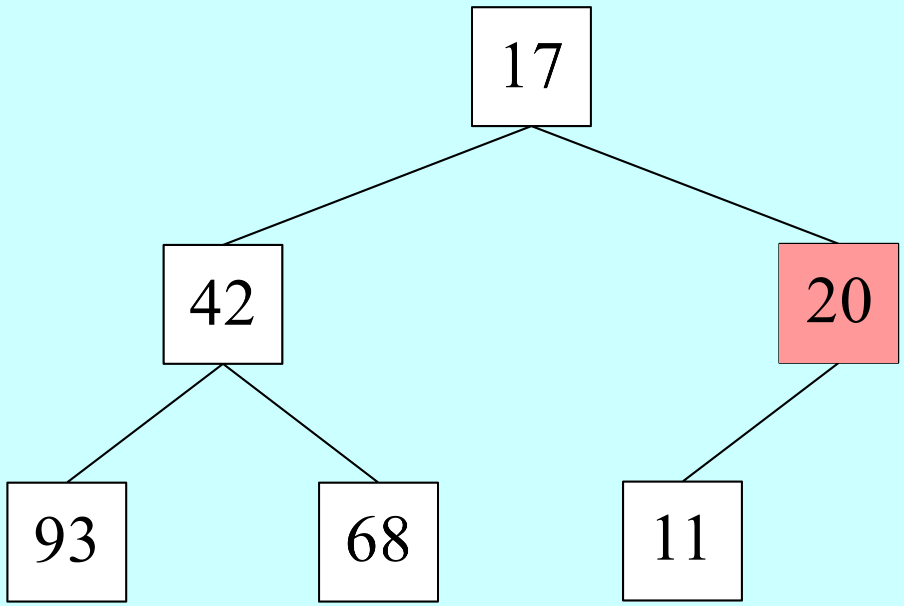


The heap:

17	42	20	93	68	11

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

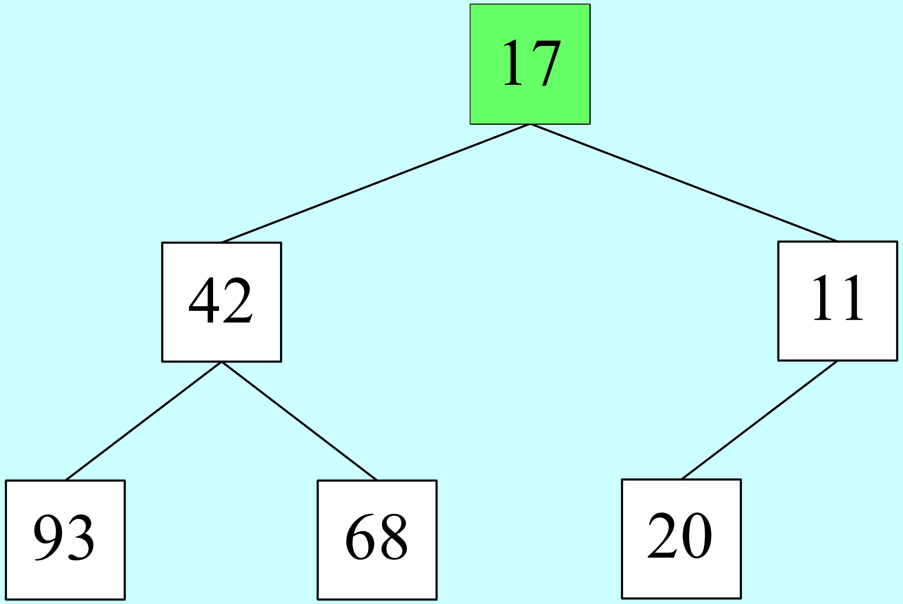


The heap:

17	42	11	93	68	20

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11



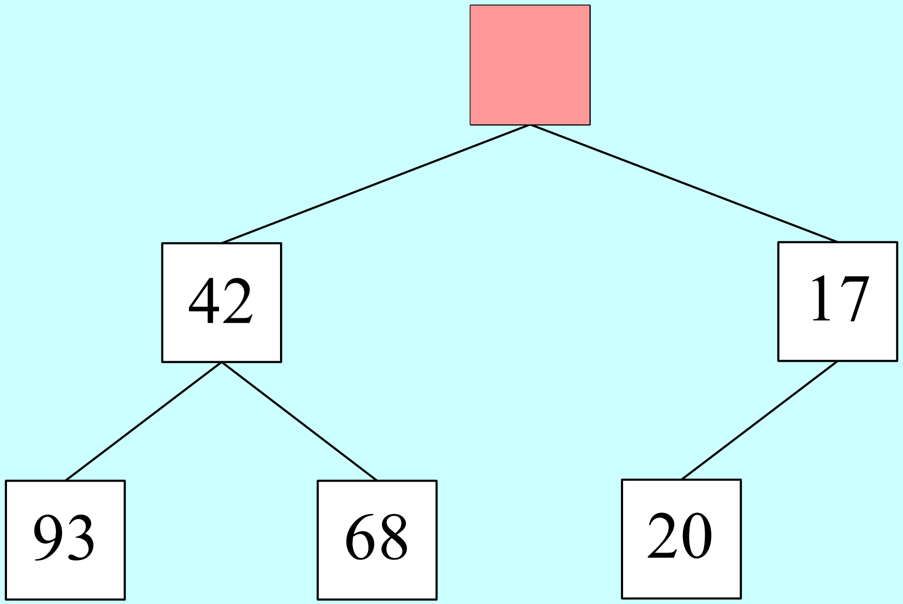
The heap:

11	42	17	93	68	20

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

Dequeue the top priority element $\longrightarrow$ 11



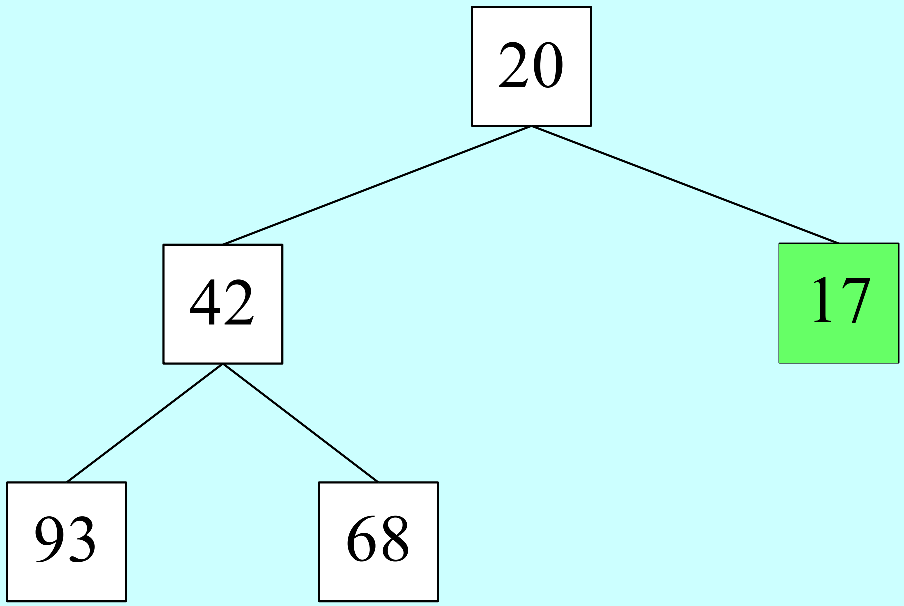
The heap:

11 20	42	17	93	68	20

Exercise: Partially Ordered Trees

Insert in order: 17 93 20 42 68 11

Dequeue the top priority element $\longrightarrow$ 11



The heap:

17	42	20	93	68	

The End