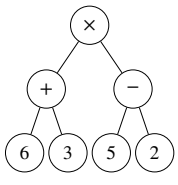
Chapter 18
Trees

18.1 Introduction to Trees

A **tree** is a mathematical abstraction that plays an important role in the design and analysis of many important algorithms. In a tree, information is organized hierarchically, and we can use this hierarchical structure to capture common properties of data.

For example, arithmetic expressions can be expressed using trees. Consider the expression $(6+3)\times(5-2)$. In the figure below, we build a tree where our operands are leaf nodes and our operators are internal (non-leaf) nodes.

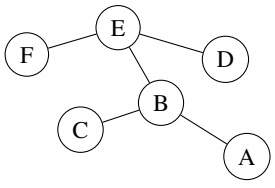


Why would we ever want to turn our math expressions into trees? By translating this mathematical expression into a tree, we are able to elegantly solve the expression using a simple tree traversal algorithm. In fact, many different calculators use these types of trees — known as *expression trees* — to solve human-readable math expressions that adhere to the order of operations.

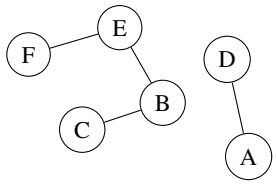
The expression tree is just one example of the versatility of trees. In this chapter, we will analyze the tree structure and explore several tree algorithms that can be used to solve different types of problems.

Remark: Before we can introduce the concept of a tree, we have to introduce the concept of a *graph*. A graph is simply a collection of nodes that are connected by edges. A tree is simply a restricted form of a graph; that is, every tree is a graph, but not every graph is a tree! In this chapter, we will focus primarily on trees — we will explore the broader category of graphs in the next chapter.

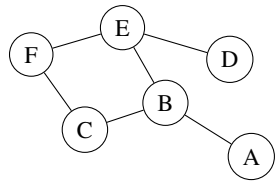
To begin, we will first introduce different types of trees. A **simple tree** is an acyclic, connected graph. A graph is *acyclic* if there are no cycles (i.e., there exist no two points in the graph that can be connected by more than one route). A graph is *connected* if there exists a path from a node to any other node in the graph (i.e., there are no disjoint sets). A few examples are shown below:



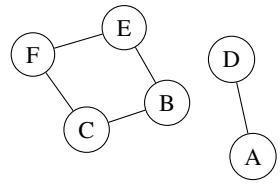
acyclic, connected
(this is a simple tree)



acyclic, not connected
(this is *not* a simple tree)

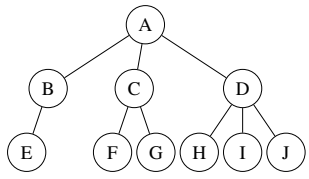


not acyclic, connected
(this is *not* a simple tree)



not acyclic, not connected
(this is *not* a simple tree)

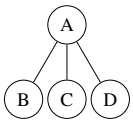
A **rooted tree** is a type of simple tree that contains a special node called the *root*. All edges in a rooted tree branch away from this root node, and any node can be selected as the root. An example of a rooted tree is shown below, where node A is the root:

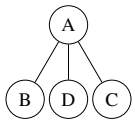


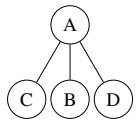
Below lists some general tree terminology (many of these terms are derived from terms used for family relationships):

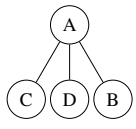
- **Root:** The root of a tree is the "top-most" vertex in the tree, from which all other nodes are directed away from. In the tree above, node A is the root of the tree.
- **Parent/Child:** For every direct connection in the tree, we define the higher node as the parent, and the lower node as the child. In the tree above, node A is the parent of nodes B, C, and D. Node B is the parent of node E. Similarly, nodes B, C, and D are the children of node A, and node E is the child of node B.
- **Sibling:** Two nodes are siblings if they share the same parent. In the tree above, B, C, and D are siblings, since they share the parent A.
- **Descendant:** The descendants of a given node consist of any node that exists along a path from the given node to a leaf node (including the leaf node). In the tree above, nodes B and E are both descendants of node A.
- **Ancestor:** The ancestors of a given node consist of any node that exists along the path from the given node to the root node (including the root node itself). In the tree above, nodes B and A are both ancestors of node E.
- **External (Leaf) Node:** An external node (or leaf node) is a node that has no children. In the tree above, nodes E to J are examples of external (leaf) nodes.
- **Internal Node:** An internal node is a node with children. In the tree above, nodes A to D are examples of internal nodes.
- **Depth:** The depth of a node represents how far it is from the top of the tree. The root has a depth of 1. If a node is one edge away from the root, it has a depth of 2. If a node is two edges away from the root, it has a depth of 3. We can define the depth of a node recursively:
  - depth(empty) = 0
  - depth(node) = depth(parent) + 1
- **Height:** The height of a node represents how far it is from the bottom of the tree. Leaf nodes have a height of 1. If a node has more than one child, its height is equal to one plus the largest height of any of its children. We can define the height of a node recursively:
  - height(empty) = 0
  - height(node) = max(height(children)) + 1
- In a tree, the largest height of any node and the largest depth of any node should be the same value.

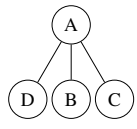
An **ordered tree** is a tree that has a linear ordering for the children of each node. In other words, the children of an ordered tree are defined in some predetermined order (e.g., left child and right child). Consider the following:

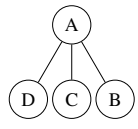






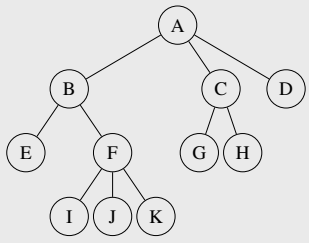






Are these trees identical? If they are *not ordered*, then yes, they are identical, because non-ordered trees do not enforce an ordering for siblings. However, if these trees are *ordered*, then no, they are not identical, since the linear order of children does matter (e.g., a tree where B is to the left of C is different from one where C is to the left of B).

Example 18.1 Consider the following tree. Which node is the root? Which nodes are leaf nodes? Which nodes are internal nodes? What is the maximum depth of the tree? What is the height of node B?



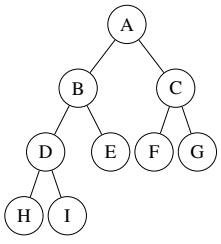
In this tree, node A is the root. Nodes A, B, C, and F are internal nodes because they have children. Nodes D, E, G, H, I, J, and K are leaves because they have no children. The maximum depth of the tree is 4 (this is the number of levels in the tree). The height of B is equal to max(height(E), height(F)) + 1. Since F is the child with the larger height (height(E) = 1, height(F) = 2), the height of B is 2 + 1 = 3.

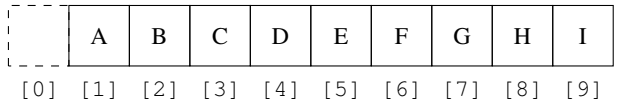
18.2 Binary Trees

A **binary tree** is a common type of ordered tree in which each node can have at most two children (a left child and a right child). For the majority of this chapter, we will be dealing with binary trees. There are two primary ways to represent a binary tree in memory: (1) using an array-based approach, and (2) using a pointer-based approach.

※ 18.2.1 *Array-Based Tree Implementation*

If we implement a binary tree using an array, we essentially flatten out the tree and store its nodes sequentially in memory, level by level. This is identical to how we used an array to represent a binary heap. (For simplicity, we assume 1-indexing as with bef)





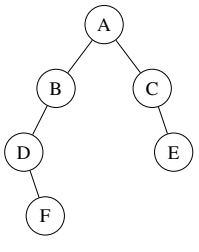
To implement a *binary* tree using an array, we will follow these rules:

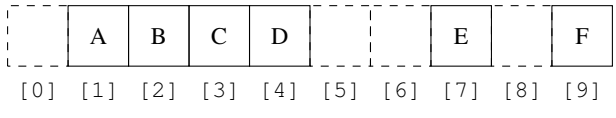
- The root of the binary tree is placed at index 1 of the array.[1]
- The left child of the node at index i should be placed at index $2i$. If node i has no left child, then index $2i$ should be empty.
- The right child of the node at index i should be placed at index $2i+1$. If node i has no right child, then index $2i+1$ should be empty.

The array-based approach worked well when we dealt with binary heaps. This is because binary heaps are **complete** binary trees. Recall that a complete binary tree is a binary tree with depth d such that:

- All nodes at a depth of 1, 2, 3, …, $d-1$, have the maximum number of nodes possible (e.g., all levels but the bottom are filled).
- All nodes at a depth of d are filled from left to right with no gaps.

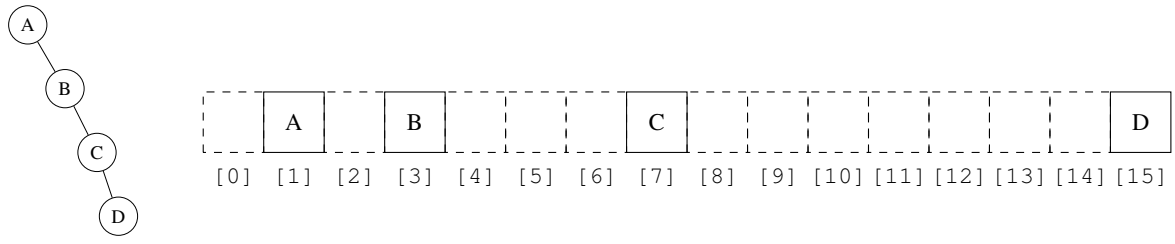
Because binary *heaps* are guaranteed to be complete, we can ensure that our underlying array will have no gaps (except perhaps index 0, if we start our heap at index 1). However, binary *trees* need not be complete, so we cannot avoid potential gaps in our underlying array. For example, if we implemented the following binary tree using an array, indices 5, 6, and 8 would be empty:





[1] Like with heaps, you don't have to physically put the root at index 1. It's perfectly fine to put the root at index 0, as long as you are able to index children correctly. You can do this by using the same formulas as above, but first converting index i into $i-1$ whenever you have to calculate the index of a child, or by using alternative formulas to calculate the indices of the left and right children.

Because indices in the underlying array may be skipped, an array-based approach can be space prohibitive for sparse trees (as memory would be wasted by allocating space for nodes that do not exist). In the worst case, we could get a stick like this, which would require us to allocate a giant array that remains mostly empty:



The following table illustrates the time complexity of operations for an array-based binary tree with n nodes:

Operation	Complexity
Insert Key (Best Case)	$\Theta(1)$
Insert Key (Worst Case)	$\Theta(n)$
Remove Key (Worst Case)	$\Theta(n)$
Find Parent	$\Theta(1)$
Find Child	$\Theta(1)$
Space Required (Best Case)	$\Theta(n)$
Space Required (Worst Case)	$\Theta(2^n)$

Note that we do not have rules for determining where an element goes when it is inserted (this will change when we talk about different types of binary trees, such as the *binary search tree*). For a pure binary tree, we can insert a node anywhere in the tree.

Insert Key

In the best case for insertion, the root node does not have two children, allowing us to identify an open index immediately (e.g., we can directly insert the new node as the left or right child of the root without having to traverse the remaining elements). In the worst case for insertion, we could have to traverse the entire array before we find an open position to insert the new element (and if the array is completely full, we may also have to reallocate the underlying array).

Remove Key

Since there are no rules for how data is stored in a binary tree, removal takes $\Theta(n)$ time in the worst case, since we may have to visit every node in the tree before we find the one we want to remove. Furthermore, if we are asked to remove a nonexistent value, we would have to traverse all the nodes in the tree before concluding the element doesn't exist, which takes $\Theta(n)$ time.

Find Parent and Child

Finding the parent or child of a node takes constant time in an array implementation. Given a node at index i (assuming the root is at index 1), its parent must be at index $i/2$, its left child must be at index $2i$, and its right child must be at index $2i+1$. This is just basic arithmetic, which can be done in $\Theta(1)$ time.

Space Complexity

Regarding the space complexity, the best case occurs when the tree is complete and there are no gaps in the underlying array. In this case, we would only need $\Theta(n)$ memory to store n nodes. The worst case occurs when we have a stick, as shown in the example above. When this happens, only a single node can exist at each depth of the tree, which forces us to use $\Theta(2^n)$ memory. This is because the maximum number of nodes that can exist at any depth d (and thus the number of array positions needed to support that depth) is equal to 2^{d-1}, assuming the root has a depth of 1. If we have a stick, the depth of our tree would be n, and the array size we need to support a depth of d is

$$2^0 + 2^1 + 2^2 + \ldots + 2^{n-2} + 2^{n-1} = 2^n - 1 = \Theta(2^n)$$

In general, the worst-case space complexity of an array-based approach depends on the maximum number of children each node could have. If each node in the tree could have at most 3 children, then the maximum number of nodes that can exist at a depth d (and thus the amount of additional memory we have to allocate to support that depth) is equal to 3^{d-1}. If we have a stick, the depth would still be n, but the array size we need to support a depth of n now becomes

$$3^0 + 3^1 + 3^2 + \ldots + 3^{n-2} + 3^{n-1} = \frac{1}{2}(3^n - 1) = \Theta(3^n)$$

To solve the above equations, the formula for the sum of a geometric series was used (where a represents the starting term, r represents the common ratio, and n represents the number of terms in the sequence):

$$a\left(\frac{1-r^n}{1-r}\right)$$

In general, given a tree with n nodes, where each node can have at most k children (where k is a constant), the worst-case space complexity of implementing this tree using an array-based approach is

$$1\left(\frac{1-k^n}{1-k}\right) = \frac{1}{k-1}(k^n - 1) = \Theta(k^n)$$

Example 18.2 Suppose you are trying to implement an **array**-based tree whose internal nodes can have up to 10 children. What are the best- and worst-case space complexities of implementing this tree, if it has n nodes?

In the best case, there are no gaps in the underlying array, which allows you to store the n elements sequentially, requiring $\Theta(n)$ space. In the worst case, you have a stick where every depth only has a single element. When this happens, you can use the formula for a geometric series to show that the space complexity is $\Theta(10^n)$.

※ 18.2.2 *Pointer-Based Tree Implementation*

An alternative to the array-based approach is the pointer-based approach. In the pointer-based approach, each node of a binary tree stores pointers to its left and right children:

```
1  template <typename T>
2  struct Node {
3    T val;          // value of type T
4    Node* left;     // pointer to left child, nullptr if no left child
5    Node* right;    // pointer to right child, nullptr if no right child
6  };
```

If we implement a binary tree this way, we would get the following time complexities for a binary tree with n nodes:

Operation	Complexity
Insert Key (Best Case)	$\Theta(1)$
Insert Key (Worst Case)	$\Theta(n)$
Remove Key (Worst Case)	$\Theta(n)$
Find Parent	$\Theta(n)$
Find Child	$\Theta(1)$
Space Required (Best Case)	$\Theta(n)$
Space Required (Worst Case)	$\Theta(n)$

Insert Key

The best case for insert happens when the root either has no left child or no right child (e.g., if `left` or `right` is `nullptr`) — when this happens, we can just insert the new node as a child of the root in constant time. The worst case of insert is still $\Theta(n)$ if we get unlucky when searching for a open spot and end up looking at every node before we find one with no left or right child.

Remove Key

Removing a key is still $\Theta(n)$ in the worst case: this happens when we try to remove an element that does not exist (we would need to look at every element before we know that the element is not in the tree), or if we get unlucky and the element we want to remove is among the last ones we visit.

Find Parent and Child

If each node does not store a parent pointer, the worst-case time complexity of finding the parent of any given node is $\Theta(n)$; this is because we will have to visit every node and check if any of its children match the node we are given. However, finding the child of a node can be done in constant time because each node stores pointers to its children.

Space Complexity

Unlike the array-based approach, the space complexity required to build a pointer-based tree with n nodes is always $\Theta(n)$. This is because memory is only allocated for nodes when they are created — there is no need to store additional space for nodes that do not exist, which we needed for an array-based tree.

An alternative would be to store a `parent` pointer within each node, as shown in the `Node` definition below. If the node is the root, the parent pointer is `nullptr`; otherwise, it points to the node's parent. Using this approach, the time complexity of finding a node's parent would be $\Theta(1)$ instead of $\Theta(n)$, since we have direct access to the parent.

```
1  template <typename T>
2  struct Node {
3    T val;          // value of type T
4    Node* parent;   // pointer to parent, nullptr if root
5    Node* left;     // pointer to left child, nullptr if no left child
6    Node* right;    // pointer to right child, nullptr if no right child
7  };
```

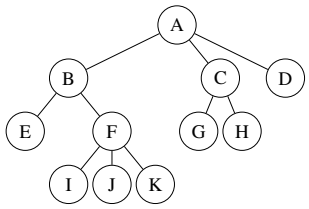
However, this implementation is rarely used because it is memory intensive, and the benefit of a `parent` isn't often worth the extra memory. Usually, we do not need a `parent` pointer, since the behavior of a `parent` pointer can be emulated using recursion (i.e., when a recursive call unrolls, we automatically move from a child to its parent; there's no need to store an explicit pointer).

※ 18.2.3 *Converting a Generic Tree into a Binary Tree*

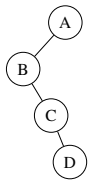
If you are given a generic tree T, where each node can have more than two children, you can turn this generic tree into a *binary* tree T' using the following procedure (starting from the root):

1. If a node v has k children v_1, v_2, v_3, …, v_k, first set v_1 as the left child of v in T'.
2. Then, set v_2, v_3, …, v_k so that they become a chain of right children of v_1 in T'.
3. Repeat these two steps recursively for the remaining nodes in the tree.

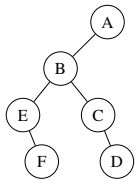
For instance, suppose we are given the following generic tree, and we want to convert it into a binary tree:



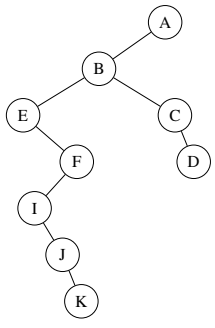
Using the algorithm discussed previously, we first consider the root node A. Node A has three children: B, C, and D, so we set B as the left child of A, and then set C and D as a chain of right children of B:



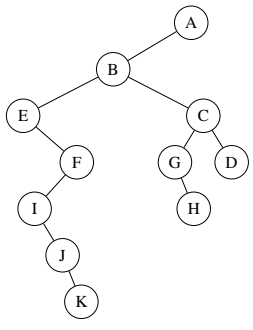
We will repeat these steps recursively. Next, we look at B, which has two children: E and F. We set E as the left child of B, and set F as the right child of E:



Node E has no children, so we do not have to do any additional work for E. Node F, however, has three children: I, J, and K. We will set node I as the left child of node F, and set nodes J and K as a chain of right children of node I.



Node C has two children: G and H. We will set G as the left child of C, and H as the right child of G:



The remaining nodes (D, G, H, I, J, K) do not have any children, so we are done. We have successfully converted our original generic tree into a binary tree. If you paid close attention, you may have noticed that this is very similar to a pairing heap, where the left child of a node represents its child, and the right child of a node represents its sibling. The original tree is on the left, and the binary tree we created is on the right (where the right children are displayed horizontally to highlight the similarities with a pairing heap). Even though a pairing heap can have more than two children, we still used the two-pointer structure of a binary tree to represent it!

Example 18.3 You are given a generic tree, where each node can have more than 2 children. Each node in the generic tree is represented as:

```
template <typename T>
struct GenericNode {
  T val;
  std::vector<GenericNode*> children;  // empty if no children
  GenericNode(T x) : val{ x } {}
};
```

Write a function that takes in a generic tree and constructs a *binary* tree with the same values as the generic tree. Return the root of the newly constructed binary tree. Each node of a binary tree is represented as follows:

```
template <typename T>
struct BinaryNode {
  T val;
  BinaryNode* left;
  BinaryNode* right;
  BinaryNode(T x) : val{ x }, left{ nullptr }, right{ nullptr } {}
};
```

To solve this problem, we will use the algorithm described earlier. First, we will look at the children of the root. If the root has no children, we can just create a copy of the node and attach it to the binary tree we are trying to construct. However, if the root does have children, we have to (1) set the first child as the root's left child and (2) set the remaining children as a chain of right children of the first child.

```
1  template <typename T>
2  BinaryNode<T>* convert_generic_to_binary(GenericNode<T> *generic_tree) {
3    BinaryNode<T> *binary_tree = new BinaryNode<T>{ generic_tree->val };
4    return helper(generic_tree, binary_tree);
5  } // convert_generic_to_binary()
6
7  template <typename T>
8  BinaryNode<T>* helper(GenericNode<T>* generic_node, BinaryNode<T>* binary_node) {
9    if (generic_node->children.empty()) {
10     return binary_node;
11   } // if
12   GenericNode<T> *next_generic = generic_node->children[0];
13   BinaryNode<T> *next_binary = new BinaryNode<T>{ next_generic->val };
14   binary_node->left = helper(next_generic, next_binary);
15   BinaryNode<T> *current = next_binary;
16   for (size_t i = 1; i < generic_node->children.size(); ++i) {
17     current->right = helper(generic_node->children[i],
18                             new BinaryNode<T>{ generic_node->children[i]->val });
19     current = current->right;
20   } // for
21   return binary_node;
22 } // helper()
```

The helper function defined on line 8 takes in a node from the generic tree and its counterpart value in a binary tree, and it recursively constructs the children of the binary node based on the children of the generic node. If a generic node has no children (base case), we can attach its corresponding binary node to the binary tree we are building (this is handled by the recursive call; when we finish processing a node, it gets returned and is thus attached to its parent when the recursive call finishes on line 14). Otherwise, we create a binary node for the first child of the generic node (`next_binary` on line 13), recursively attach the children of this generic node, and then attach this newly constructed binary node as the left child of the binary node on line 14. Then, on lines 15-20, we set the remaining children of the generic node as a chain of right children of the new binary node `next_binary`. The completed binary node is then returned, and it is attached as the left child of its parent as the recursion unrolls.

18.3 Tree Traversals

If you want to process every node in a tree, you will have to complete a **tree traversal**: a systematic method for processing every node in a tree. In this section, we will look at four different tree traversal methods: the preorder, inorder, postorder, and level-order traversals, each of which visit the nodes of a tree in a different order. The first three of these traversals (preorder, inorder, postorder) are recursive, while the level-order traversal is iterative. The names of these traversals actually provide insight into their behavior: the prefix of the traversal name ("pre", "in", and "post") tells us when to process the parent node (i.e., the node you are currently visiting) relative to its children. In a preorder traversal, the parent node is processed *before* its left and right children; in a postorder traversal, the parent node is processed *after* its left and right children; and in an inorder traversal, the parent node is processed *in between* its left and right children.

※ 18.3.1 *Preorder Traversal*

To conduct a **preorder traversal**, you would complete the following steps, in this order:

1. process the parent node (the node the recursion is currently on)
2. recursively process the left subtree
3. recursively process the right subtree

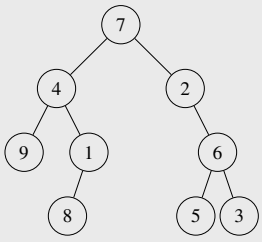
The code for a preorder traversal is shown below:

```
1  void preorder(Node* root) {
2    if (!root) return;
3    process(root->val);      // process the current node
4    preorder(root->left);    // recurse into left child
5    preorder(root->right);   // recurse into right child
6  } // preorder()
```

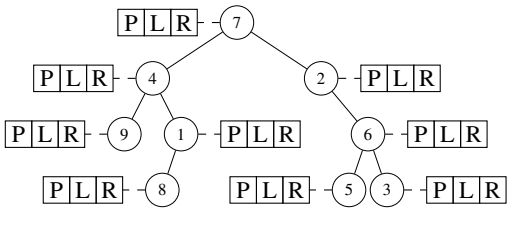
Note that the `process()` function in the code above is just a placeholder for the work that is done on each node. For example, to print out the nodes in a preorder traversal, you would replace `process()` with a print statement:

```
1  void preorder(Node *root) {
2    if (!root) return;
3    std::cout << root->val << '\n';
4    preorder(root->left);
5    preorder(root->right);
6  } // preorder()
```

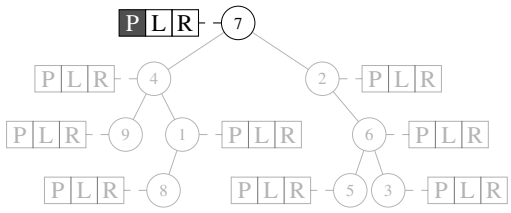
Example 18.4 What is the preorder traversal of the following tree? That is, if you were to conduct a preorder traversal of the tree, in what order would you process the nodes?



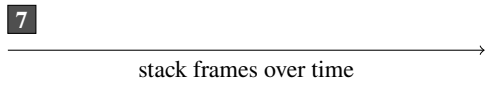
In a preorder traversal, we will always process a node before we recursively process its left and right children. There are three steps we have to complete at each node: process its value, recursively visit its left child, and then recursively visit its right child. To illustrate this process, we will label each node with three letters — *P*, *L*, and *R* — that represent each of these three steps. We will mark each letter as "completed" as soon as we finish each step (i.e., we will mark the *P* step of a node as completed after we finish processing its value, we will mark the *L* step of a node as completed as soon as we finish processing its left subtree, and we will mark the *R* step of a node as completed as soon as we finish processing its right subtree). In a preorder traversal, *P* must be completed before *L*, and *L* must be completed before *R*.



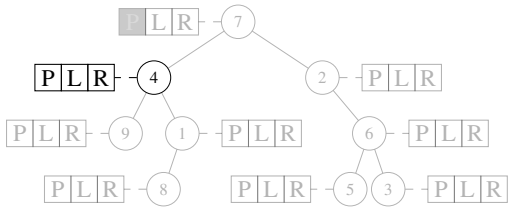
We start at the root (7), which is our current stack frame. The first step is to process the data of the current node, so the first element in our preorder traversal is 7. In fact, the first element of a preorder traversal is always the root, since a node is always processed before its children.



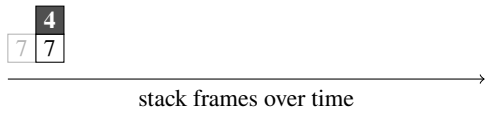
Preorder traversal: 7



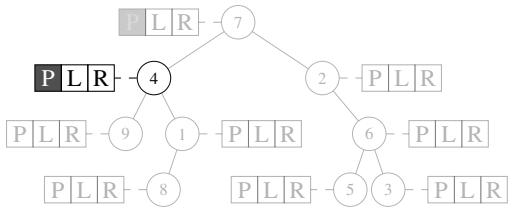
After processing the value of the root node, we mark its *P* step as complete. The next step is to recursively process the left child (*L*), so we make a recursive call on node 4, which becomes the value of our current stack frame.



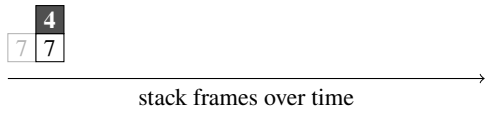
Preorder traversal: 7

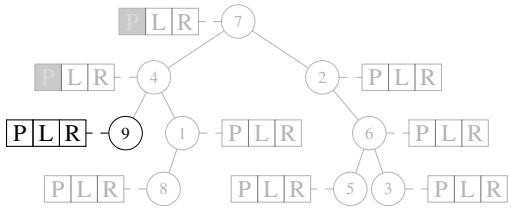


First, we will process the value of our current node, so 4 is the next value in our preorder traversal. We then mark the *P* step of node 4 as completed and make a recursive call on the left child of 4 (node 9), which becomes the value of our current stack frame.

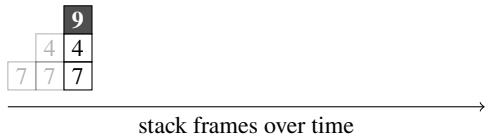


Preorder traversal: 7, 4

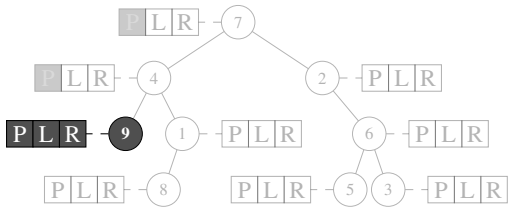




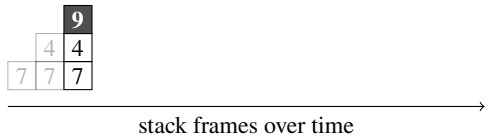
Preorder traversal: 7, 4

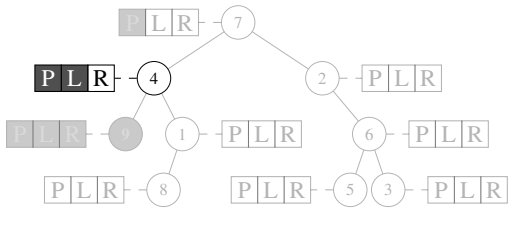


We process the value of our current node, so 9 is next in our preorder traversal. Since 9 has no children, the recursive calls on its left and right children can be completed trivially, and we can mark the *L* and *R* steps of node 9 as complete. Since all the work for node 9 is finished, the recursive call unrolls, and we return to the stack frame of node 4. The *L* step of node 4 is now complete.

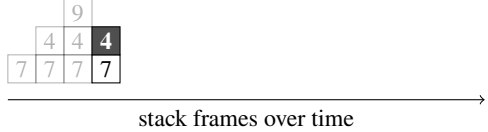


Preorder traversal: 7, 4, 9

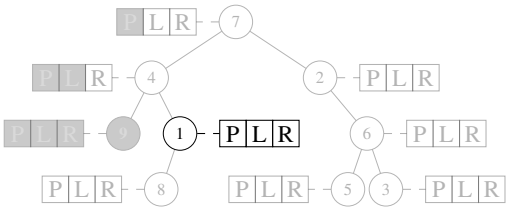




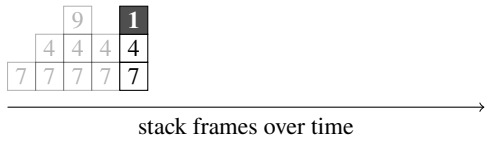
Preorder traversal: 7, 4, 9



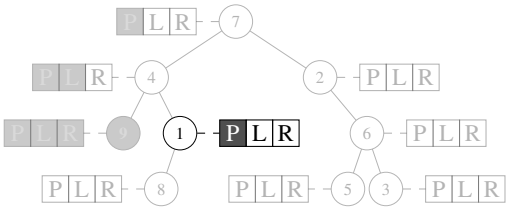
Since the recursive call on the left child is done, the next step is to make a recursive call on the right child of node 4, which is node 1. Node 1 thus becomes the value of our current stack frame.



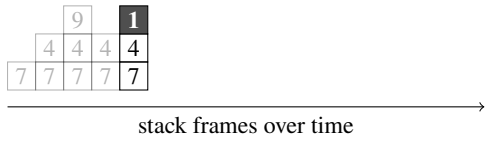
Preorder traversal: 7, 4, 9

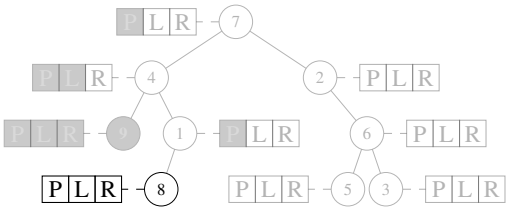


We first process the value of our current node, so 1 is next in our preorder traversal. We then make a recursive call on 1's left child, or node 8. Node 8 then becomes the node on our current stack frame.

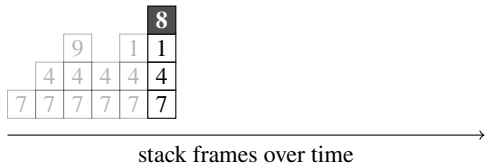


Preorder traversal: 7, 4, 9, 1

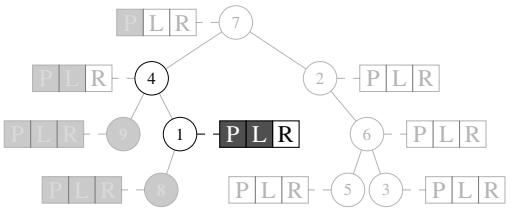




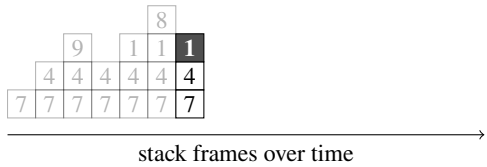
Preorder traversal: 7, 4, 9, 1



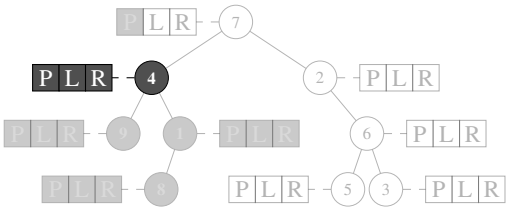
The value of node 8 is processed first, so 8 is next in our preorder traversal. Since node 8 has no left or right children, we do not need to do any more work for this node. The recursion unrolls, and we return to the stack frame of node 1.



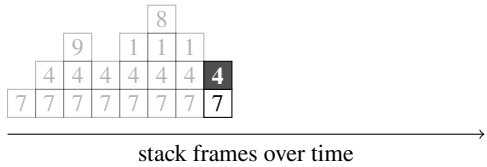
Preorder traversal: 7, 4, 9, 1, 8



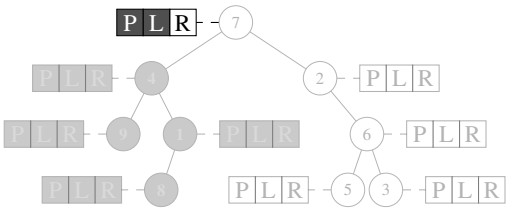
Now, we will make a recursive call on the right child of node 1. However, node 1 has no right child, so this step can be done trivially. All three steps for node 1 are now complete, so the recursion unrolls, and we return to the stack frame of node 4.



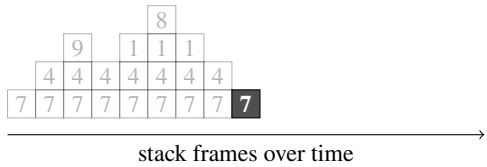
Preorder traversal: 7, 4, 9, 1, 8



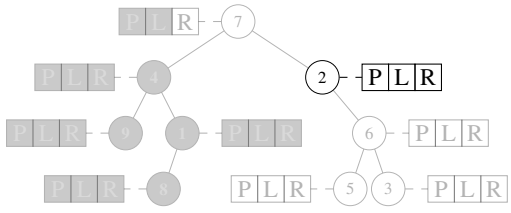
All three steps for node 4 are done, so the recursion unrolls, and we return to the stack frame of node 7. Since the entire left subtree of node 7 has been completely processed, we can mark the L step of the root node as complete.



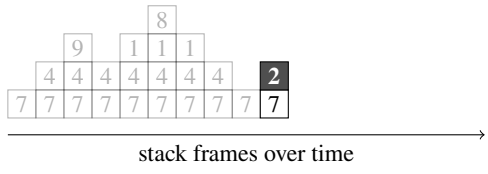
Preorder traversal: 7, 4, 9, 1, 8



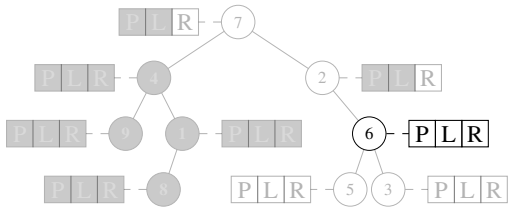
Our next step is to make a recursive call on the right child of 7, or node 2.



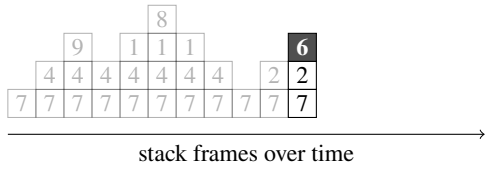
Preorder traversal: 7, 4, 9, 1, 8



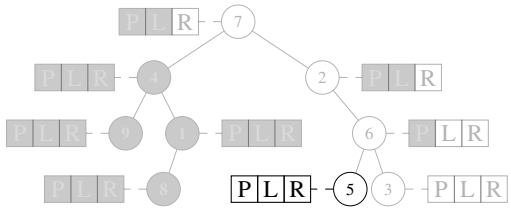
We then process the value of node 2, so 2 is next in our preorder traversal. Since 2 has no left child, the recursive call on its left child can be completed trivially. We then make a recursive call on the right child of 2, or node 6. Node 6 now becomes the node on our current stack frame.



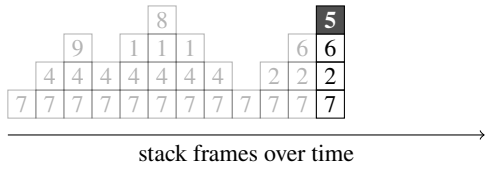
Preorder traversal: 7, 4, 9, 1, 8, 2



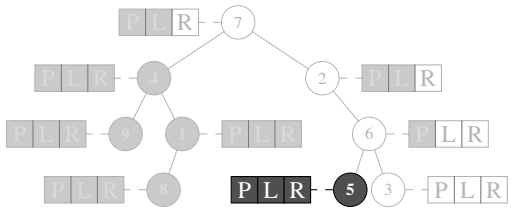
We now process the value of node 6 and add it to our preorder traversal. Then, we make a recursive call on the left child of node 6, or node 5. Node 5 now becomes the node on our current stack frame.



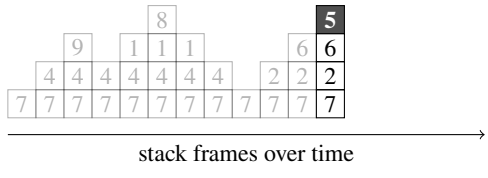
Preorder traversal: 7, 4, 9, 1, 8, 2, 6

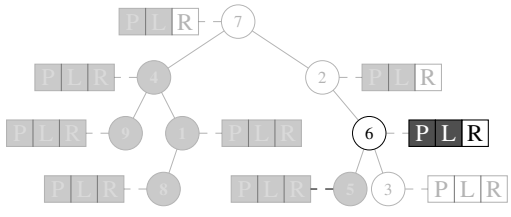


We process the value of node 5 by adding it to our preorder traversal. Since 5 has no left or right children, no more work needs to be done on this node. The recursion unrolls, and we return to the stack frame of node 6.

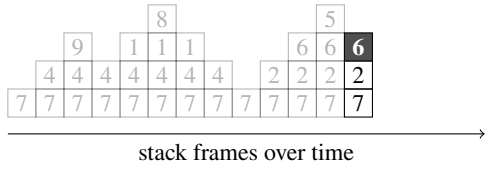


Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5

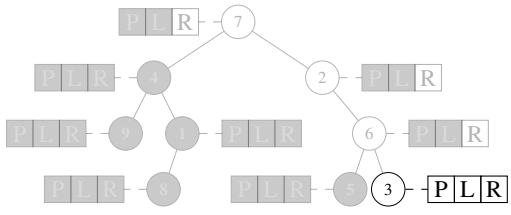




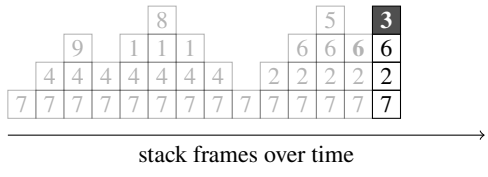
Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5



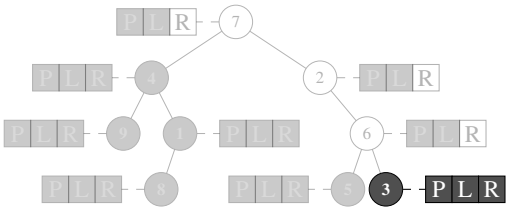
Now, we will make a recursive call on the right child of node 6, or node 3.



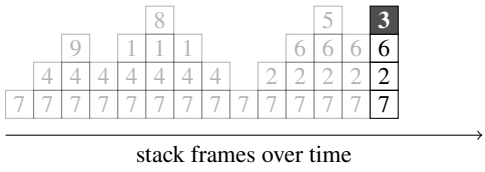
Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5

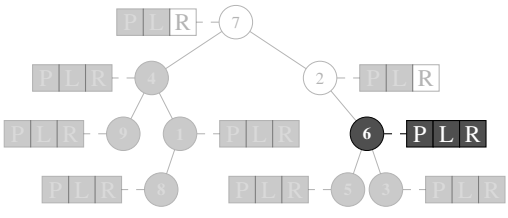


We process the value of node 3 by adding it to our preorder traversal. Since 3 has no left or right children, no more work needs to be done on this node. The recursion unrolls, and we return to the stack frame of node 6.

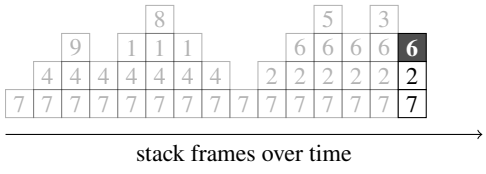


Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5, 3

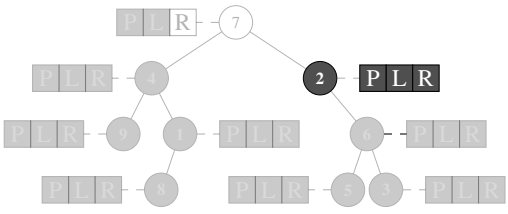




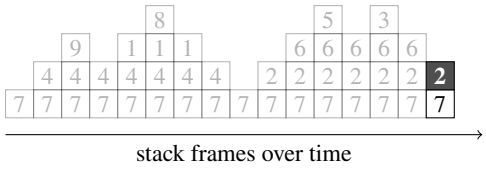
Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5, 3



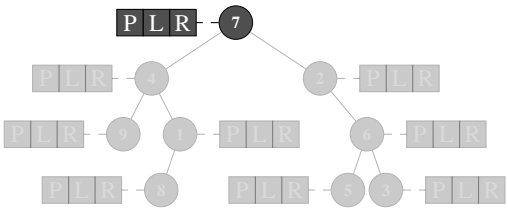
All three steps for node 6 have been completed, so the recursion unrolls, and we return to the stack frame of node 2.



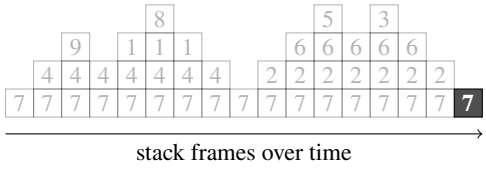
Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5, 3



All three steps for node 2 have also been completed, so the recursion unrolls, and we return to the stack frame of node 7.



Preorder traversal: 7, 4, 9, 1, 8, 2, 6, 5, 3



The function returns, and the traversal is complete. The preorder traversal of this tree is: 7, 4, 9, 1, 8, 2, 6, 5, 3.

※ 18.3.2 *Inorder Traversal*

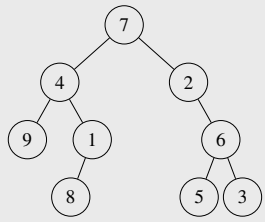
To conduct an **inorder traversal**, you would complete the following steps, in this order:

1. recursively process the left subtree
2. process the parent node (the node the recursion is currently on)
3. recursively process the right subtree

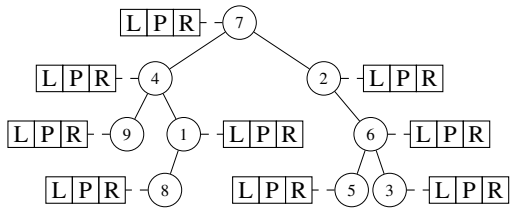
The code for an inorder traversal is shown below:

```
1  void inorder(Node* root) {
2    if (!root) return;
3    inorder(root->left);   // recurse into left child
4    process(root->val);    // process the current node
5    inorder(root->right);  // recurse into right child
6  } // inorder()
```

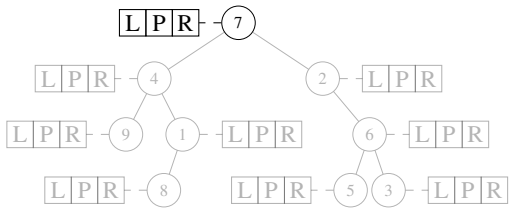
Example 18.5 What is the inorder traversal of the following tree? That is, if you were to conduct a inorder traversal of the tree, in what order would you process the nodes?



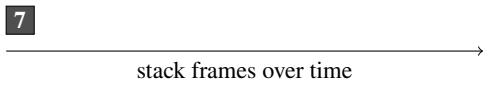
We will use the same procedure as before, but this time we will recursively process a node's left subtree before we process the node's value itself. In other words, L must be completed before P, and P must be completed before R.

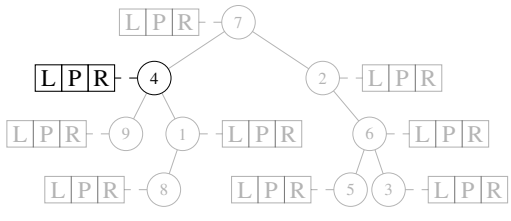


Once again, we will start at the root. The first step is to recursively process the left subtree of the root, so we make a recursive call on node 4 (and 4 becomes the node of our current stack frame).

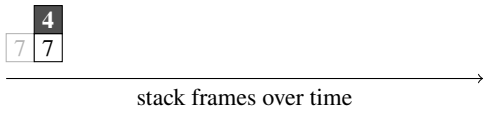


Inorder traversal:

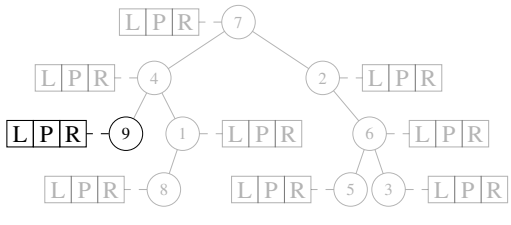




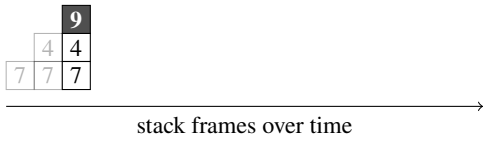
Inorder traversal:



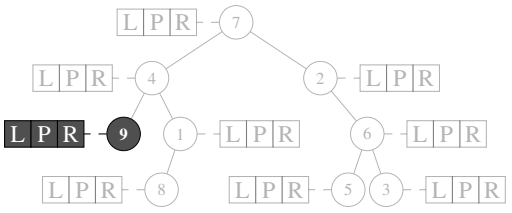
Our current node is node 4. However, before we can add 4 to the inorder traversal, we have to recursively process its left child. Thus, we make a recursive call on 4's left child, or 9. Node 9 now becomes the node on our current stack frame.



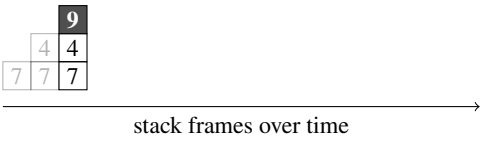
Inorder traversal:

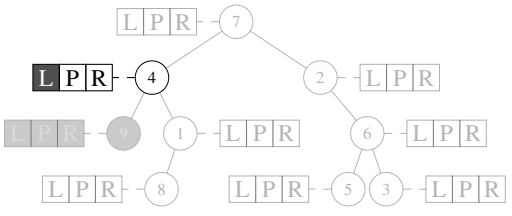


We are now at node 9. We must recursively process 9's left child, then add 9 to our inorder traversal, then recursively process 9's right child. Since 9 has no left or right children, the recursive calls can be completed trivially, and 9 is added to our inorder traversal. The recursion unrolls, and we return to the stack frame of node 4.

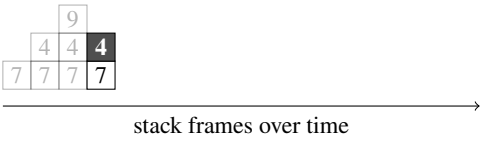


Inorder traversal: 9

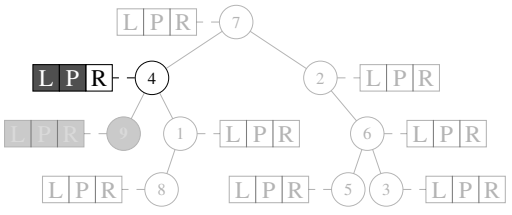




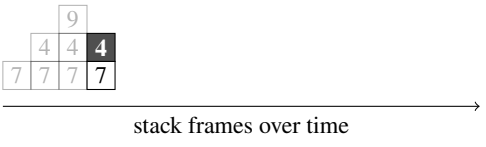
Inorder traversal: 9



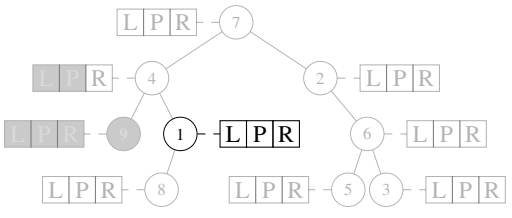
Since the left subtree of node 4 has been completely processed, we can add 4 to our inorder traversal.



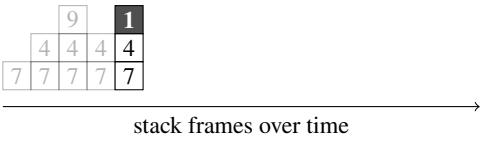
Inorder traversal: 9, 4



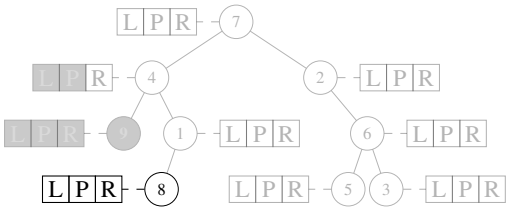
We then make a recursive call to 4's right child, or node 1.



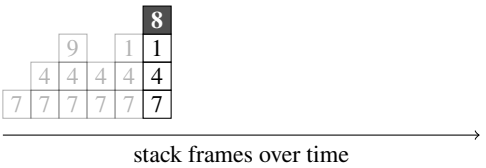
Inorder traversal: 9, 4

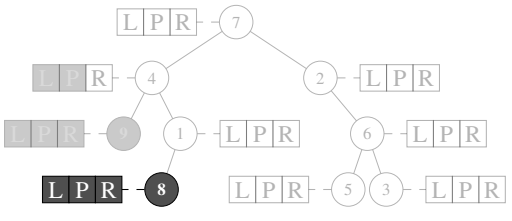


Our current node is now node 1. Before we can add 1 to our traversal, we must make a recursive call on 1's left child, or node 8. Since 8 has no left child, the *L* step can be completed trivially, and 8 gets added to the inorder traversal.

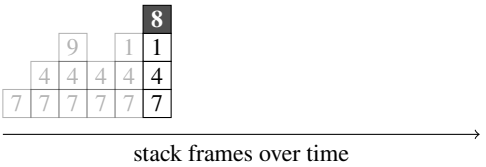


Inorder traversal: 9, 4

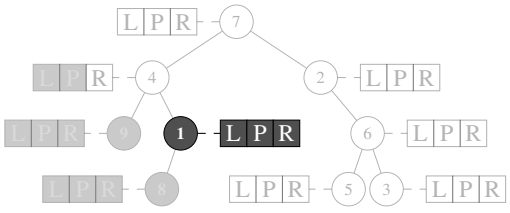




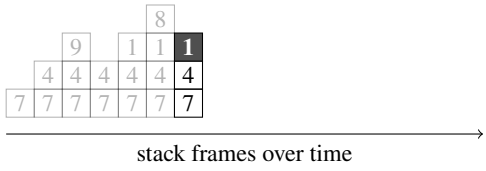
Inorder traversal: 9, 4, 8



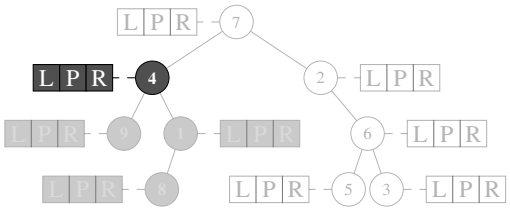
The recursion unrolls, and we return to node 1. Since the left subtree of 1 has been fully processed, we can add 1 to the inorder traversal. Then, we make a recursive call to 1's right child (which can be done trivially since 1 has no right child).



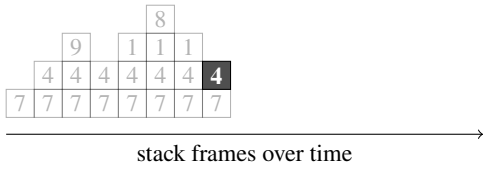
Inorder traversal: 9, 4, 8, 1



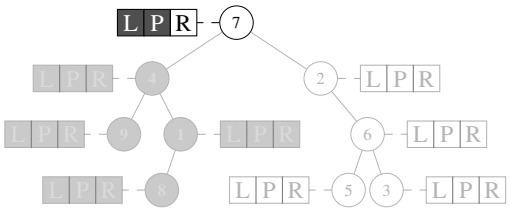
All three steps for node 1 are complete, so the recursion unrolls to node 4.



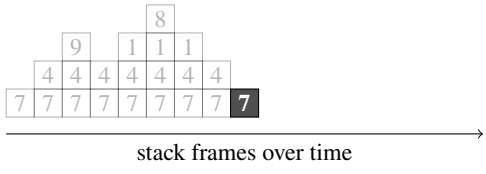
Inorder traversal: 9, 4, 8, 1



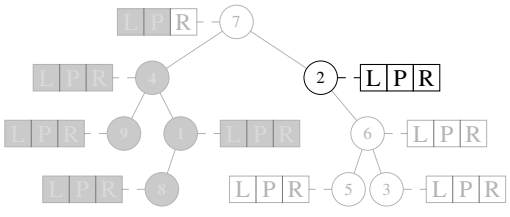
All three steps for node 4 are complete, so the recursion unrolls to node 7. Since the entire left subtree of 7 has been processed, we can now add 7 to the inorder traversal.



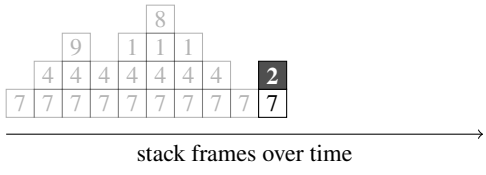
Inorder traversal: 9, 4, 8, 1, 7



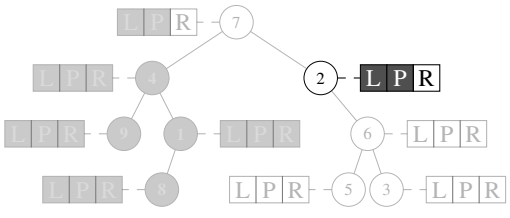
Now, we will make a recursive call on 7's right child, or node 2.



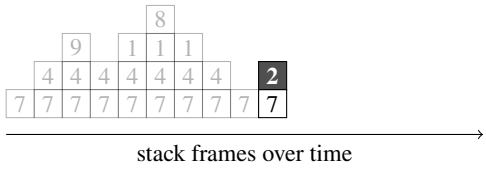
Inorder traversal: 9, 4, 8, 1, 7



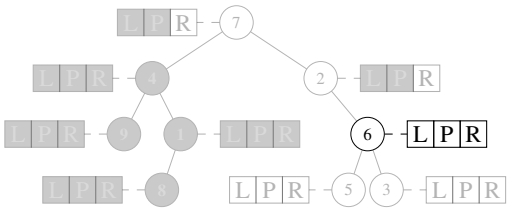
We first make a recursive call on 2's left child (which can be done trivially since 2 has no left child). Then, we can add 2 to our inorder traversal.



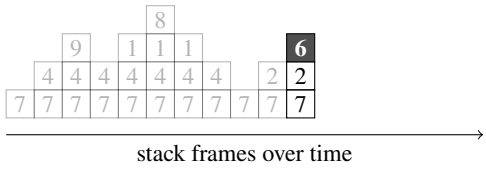
Inorder traversal: 9, 4, 8, 1, 7, 2



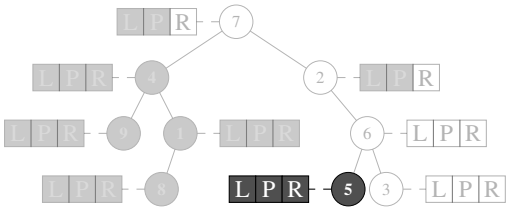
Next, we make a recursive call on 2's right child, or node 6.



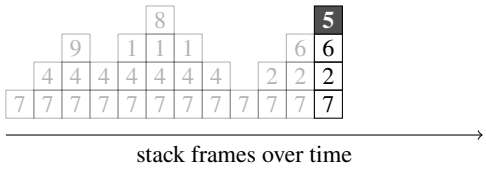
Inorder traversal: 9, 4, 8, 1, 7, 2



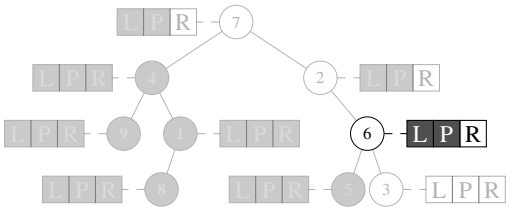
We then make a recursive call on 6's left child, or node 5. Since 5 has no children, we can add 5 to our inorder traversal, as the *L* and *R* steps can be completed trivially.



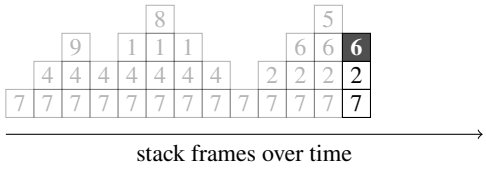
Inorder traversal: 9, 4, 8, 1, 7, 2, 5



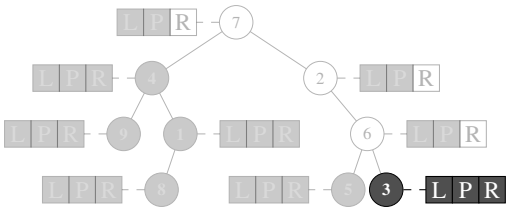
The recursion unrolls, and we return to node 6. Since the left subtree of node 6 has been fully processed, we can add 6 to our inorder traversal.



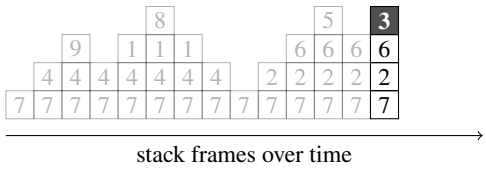
Inorder traversal: 9, 4, 8, 1, 7, 2, 5, 6



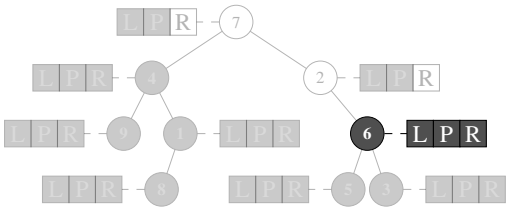
We then make a recursive call on 6's right child, or node 3. Since 3 has no children, we can add 3 to our inorder traversal.



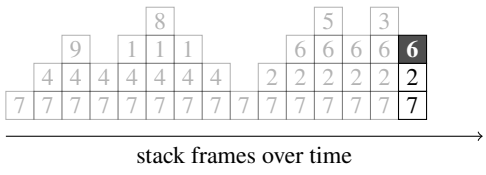
Inorder traversal: 9, 4, 8, 1, 7, 2, 5, 6, 3

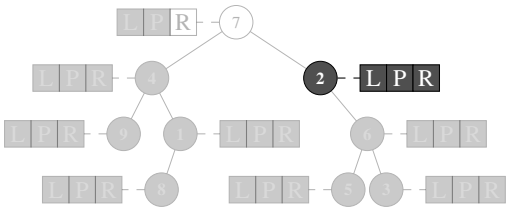


The recursion unrolls, and we return to node 6. All the steps for node 6 are complete, so the recursion unrolls again to node 2. The steps for node 2 are also complete, so we return to the root node. Since all three steps for the root are complete, the function returns, and the traversal is complete. The inorder traversal of this tree is: 9, 4, 8, 1, 7, 2, 5, 6, 3.

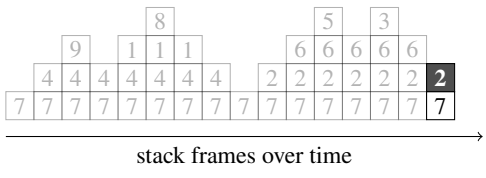


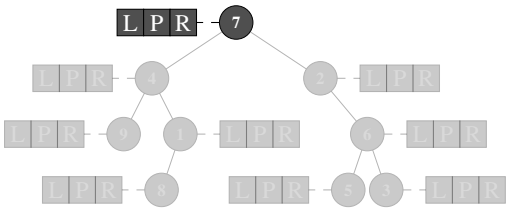
Inorder traversal: 9, 4, 8, 1, 7, 2, 5, 6, 3



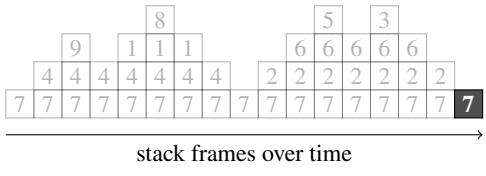


Inorder traversal: 9, 4, 8, 1, 7, 2, 5, 6, 3





Inorder traversal: 9, 4, 8, 1, 7, 2, 5, 6, 3



※ 18.3.3 *Postorder Traversal*

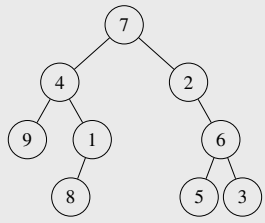
To conduct a **postorder traversal**, you would complete the following steps, in this order:

1. recursively process the left subtree
2. recursively process the right subtree
3. process the parent node (the node the recursion is currently on)

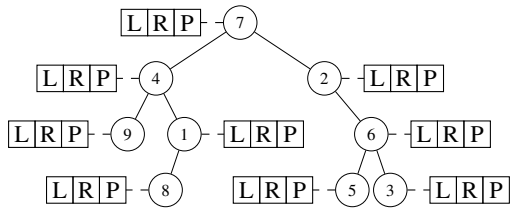
The code for a postorder traversal is shown below:

```
1  void postorder(Node* root) {
2    if (!root) return;
3    postorder(root->left);   // recurse into left child
4    postorder(root->right);  // recurse into right child
5    process(root->val);      // process the current node
6  } // postorder()
```

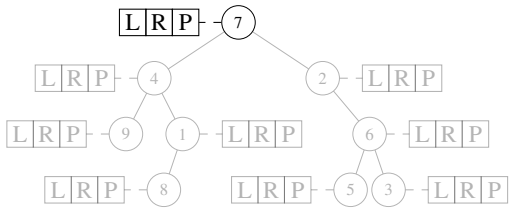
Example 18.6 What is the postorder traversal of the following tree? That is, if you were to conduct a postorder traversal of the tree, in what order would you process the nodes?



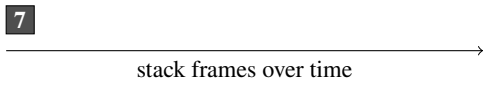
We will use the same procedure as before, but this time we will process both the left and right subtrees of a node before we add its value to the postorder traversal.

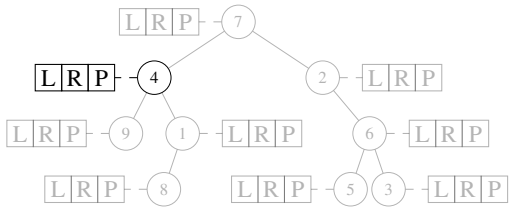


We start at the root node. Since this is a postorder traversal, we first make a recursive call on its left child, or node 4.

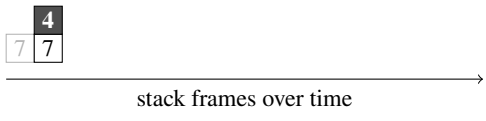


Postorder traversal:

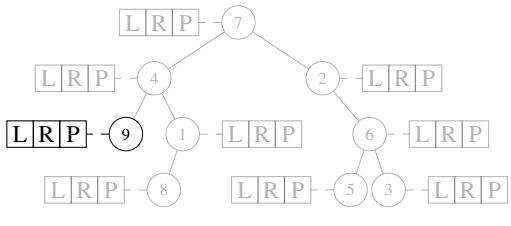




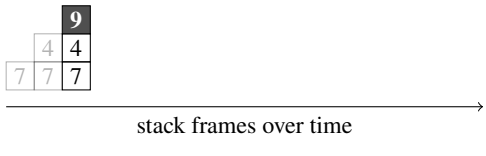
Postorder traversal:



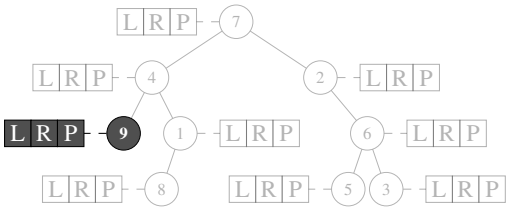
We then make a recursive call on 4's left child, or node 9.



Postorder traversal:

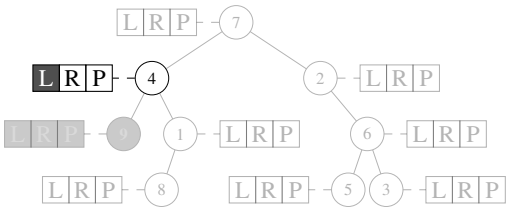


Next, we make recursive calls on 9's left and right children. However, since 9 is a leaf node, these steps can be completed trivially. Since the *L* and *R* steps are done, we can then add 9 to our postorder traversal.

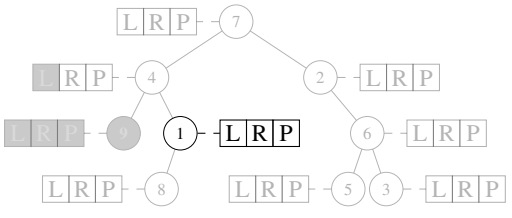


Postorder traversal: 9

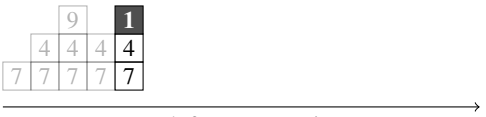
The recursion unrolls, and we return to node 4. Since we have finished processing 4's left child, we will now make a recursive call to 4's right child, or node 1.



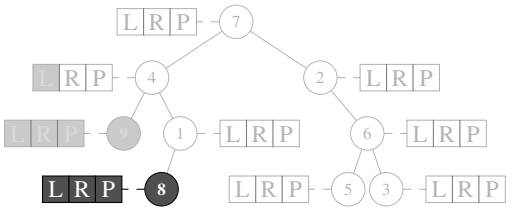
Postorder traversal: 9



Postorder traversal: 9

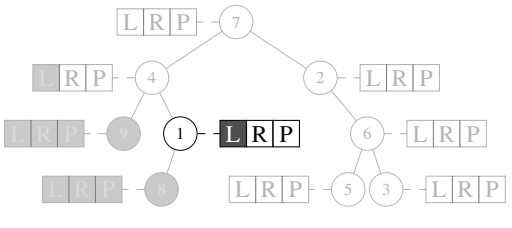


Now, we will make a recursive call to 1's left child, or node 8. Node 8 has no children, so its *L* and *R* steps can be completed trivially. 8 is then added to our postorder traversal.

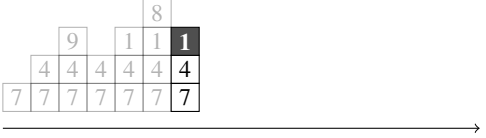


Postorder traversal: 9, 8

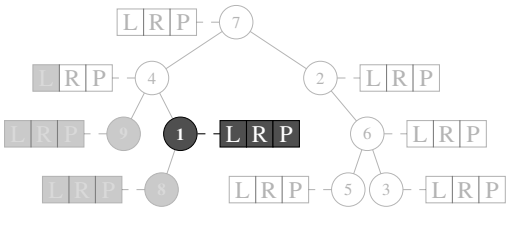
The recursion unrolls, and we return to the stack frame of node 1.



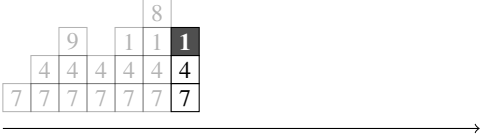
Postorder traversal: 9, 8

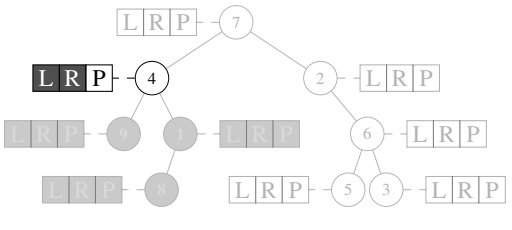


Next, we make a recursive call on 1's right child. However, 1 has no right child, so this step can be completed trivially. We can now add 1 to the postorder traversal, and the recursion unrolls back to node 4.

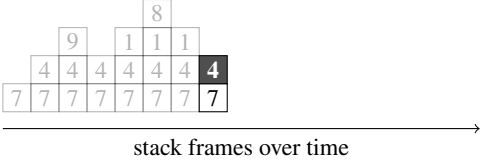


Postorder traversal: 9, 8, 1

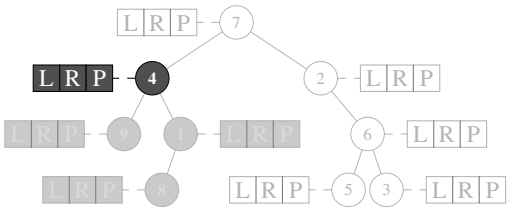




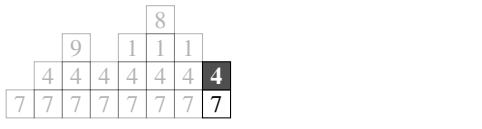
Postorder traversal: 9, 8, 1

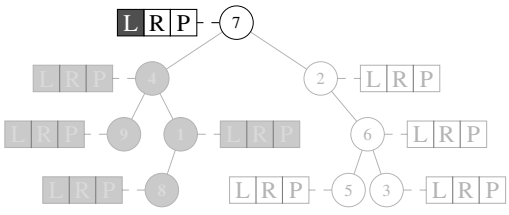


We can now add 4 to the postorder traversal. The recursion then unrolls back to the root.

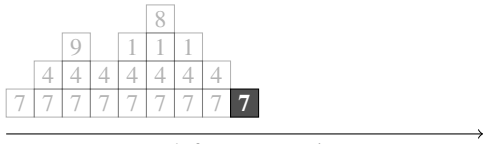


Postorder traversal: 9, 8, 1, 4

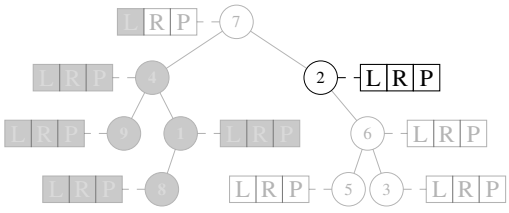




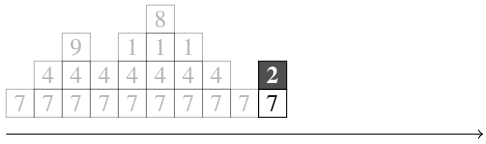
Postorder traversal: 9, 8, 1, 4



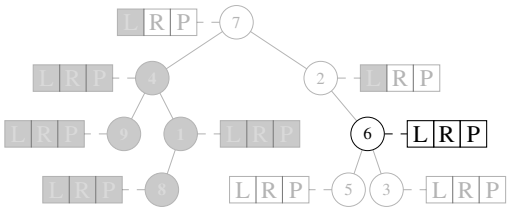
Now, we make a recursive call to the right child of 7, or node 2.



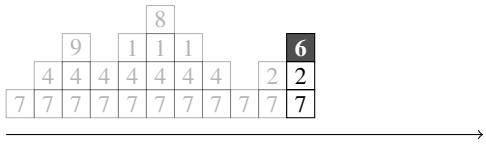
Postorder traversal: 9, 8, 1, 4



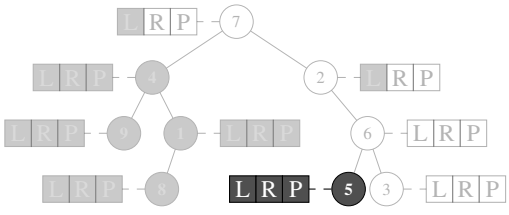
We then make a recursive call to 2's left child. Since 2 has no left child, this step can be completed trivially. We follow with a recursive call to 2's right child, or node 6.



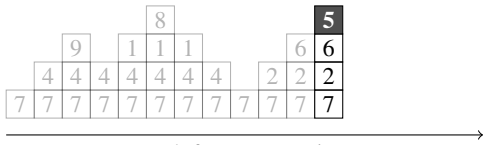
Postorder traversal: 9, 8, 1, 4



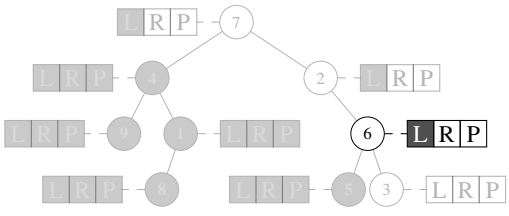
Now, we make a recursive call on the left child of node 6, or node 5. Since 5 has no left or right children, we can immediately add it to our postorder traversal.



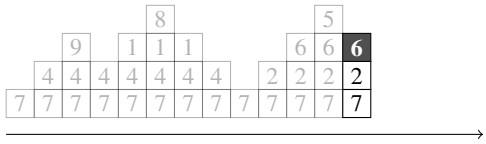
Postorder traversal: 9, 8, 1, 4, 5

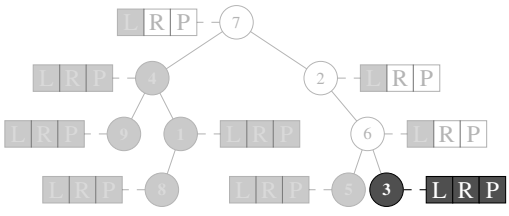


The recursion unrolls, and we return to node 6. Next, we will make a recursive call to 6's right child, or 3. Since 3 has no left or right children, we can immediately add it to our postorder traversal.

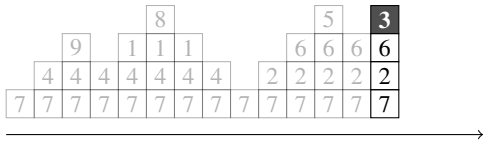


Postorder traversal: 9, 8, 1, 4, 5

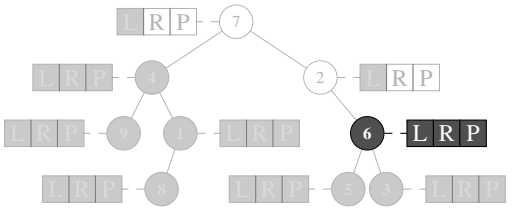




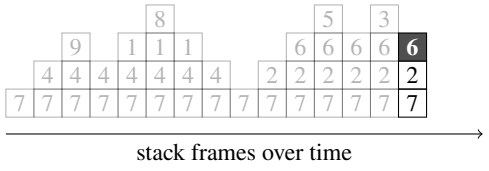
Postorder traversal: 9, 8, 1, 4, 5, 3



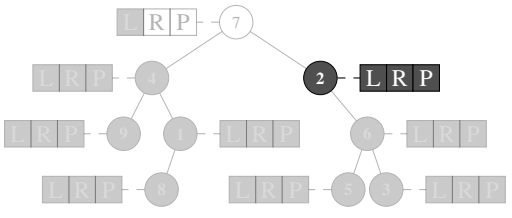
The recursion unrolls again, and we return to node 6. Since the left and right children of 6 have both been processed, we can add 6 to our postorder traversal.



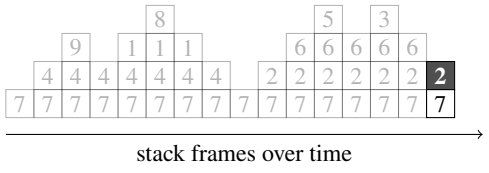
Postorder traversal: 9, 8, 1, 4, 5, 3, 6



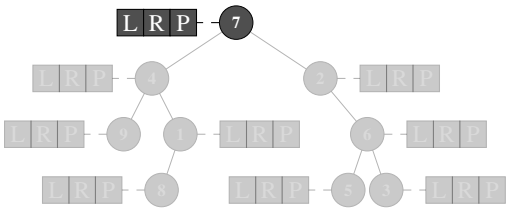
The work for node 6 is done, so the recursion unrolls to node 2. Similarly, because the left and right children of 2 have both been processed, we can add 2 to our postorder traversal.



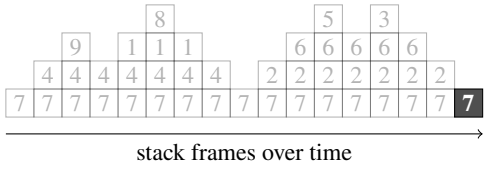
Postorder traversal: 9, 8, 1, 4, 5, 3, 6, 2



The work for node 2 is done, so the recursion unrolls back to the root. We have finished processing both the left and right subtrees of the root, so we can add 7 to the postorder traversal. The function returns, and the traversal is complete.



Postorder traversal: 9, 8, 1, 4, 5, 3, 6, 2, 7



The postorder traversal of the tree is 9, 8, 1, 4, 5, 3, 6, 2, 7.

To summarize, the preorder, inorder, and postorder traversals are three methods you can use to process the nodes in a tree. In a preorder traversal, you first process the value of the current node, then you recursively process its left subtree, then you recursively process its right subtree. In an inorder traversal, you first recursively process the current node's left subtree, then you process the value of the current node, then you recursively process the current node's right subtree. In a postorder traversal, you first recursively process the current node's left subtree, then you recursively process the current node's right subtree, then you process the value of the current node itself.

```
1  void preorder(Node* p) {
2    if (!p) return;
3    process(p->val);
4    preorder(p->left);
5    preorder(p->right);
6  } // preorder()
```

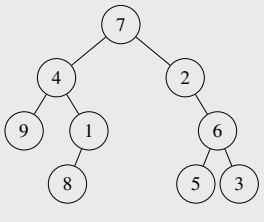
```
1  void inorder(Node* p) {
2    if (!p) return;
3    inorder(p->left);
4    process(p->val);
5    inorder(p->right);
6  } // inorder()
```

```
1  void postorder(Node* p) {
2    if (!p) return;
3    postorder(p->left);
4    postorder(p->right);
5    process(p->val);
6  } // postorder()
```

※ 18.3.4 *Level-Order Traversal*

The **level-order traversal** is another traversal method that can be used to process the nodes of a tree. In a level-order traversal, nodes are processed in order of increasing depth, where nodes on the same level (i.e., nodes that have the same depth) are processed from left to right.

Example 18.7 What is the level-order traversal of the following tree? That is, if you were to conduct a level-order traversal of the tree, in what order would you process the nodes?

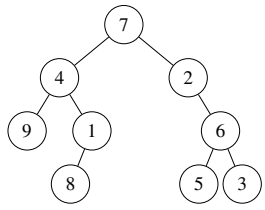


The level-order traversal processes nodes in order of increasing depth, from left to right: 7, 4, 2, 9, 1, 6, 8, 5, 3.

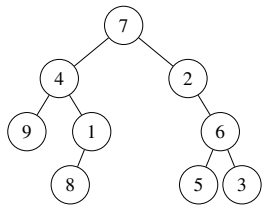
Unlike the other three traversal methods, the level-order traversal is iterative rather than recursive. To conduct a level-order traversal, a queue is used in place of recursive calls (using an algorithm known as a *breadth-first search*, which will be covered in the next chapter). The code for a level-order traversal is shown below:

```
1  void level_order(Node* root) {
2    if (!root) return;
3    std::queue<Node*> bfs;
4    bfs.push(root);  // push root into queue
5    while (!bfs.empty()) {  // while queue not empty
6      Node* curr = bfs.front();
7      bfs.pop();
8      process(curr);
9      // push current node's left and right children into queue
10     if (curr->left) {
11       bfs.push(curr->left);
12     } // if
13     if (curr->right) {
14       bfs.push(curr->right);
15     } // if
16   } // while
17 } // level_order()
```

In the code, nodes are pushed into the queue in level order (i.e., nodes closer to the root are pushed in *before* nodes closer to the leaves). Since queues support first-in, first-out (FIFO) behavior, we are able to process the nodes in the same order that they are pushed into the queue. Consider the tree in the previous example:



First, we push the root into the queue, as shown on line 4:



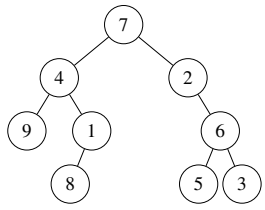
Queue:

Level-order traversal:

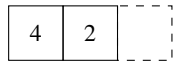
Then, as long as the queue is not empty, we would repeat the following:

1. Take out the node at the front of the queue.
2. Process the node (in this case, print it to the level-order traversal).
3. Push the node's left child into the queue, if one exists.
4. Push the node's right child into the queue, if one exists.

During the first iteration of the `while` loop, we take out the node at the front of the queue (node 7), add it to our level-order traversal, and push 7's children (nodes 4 and 2) into the queue.



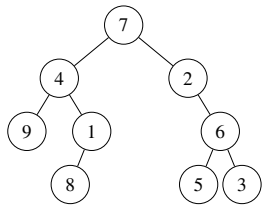
Queue:



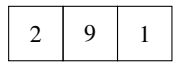
Level-order traversal:

7

During the second iteration, we take out the node at the front of the queue (node 4), add it to our level-order traversal, and push 4's children (nodes 9 and 1) into the queue.



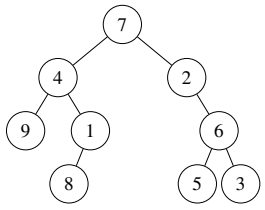
Queue:



Level-order traversal:

7, 4

During the third iteration, we take out node 2, add it to our level-order traversal, and push 2's child (node 6) into the queue.

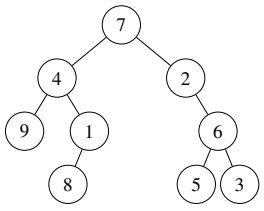


Queue:

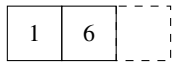
Level-order traversal:

7, 4, 2

During the fourth iteration, we take out node 9 and add it to our level-order traversal. Since 9 has no children, nothing gets pushed into the queue.



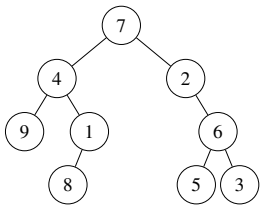
Queue:



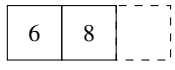
Level-order traversal:

7, 4, 2, 9

During the fifth iteration, we take out node 1, add it to our level-order traversal, and push 1's child (node 8) into the queue.



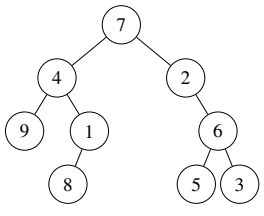
Queue:



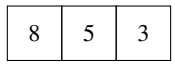
Level-order traversal:

7, 4, 2, 9, 1

During the sixth iteration, we take out node 6, add it to our level-order traversal, and push 6's children (5 and 3) into the queue.



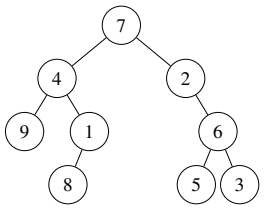
Queue:



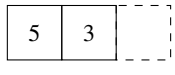
Level-order traversal:

7, 4, 2, 9, 1, 6

During the seventh iteration, we take out node 8 and add it to our level-order traversal. 8 has no children, so nothing gets pushed into the queue.



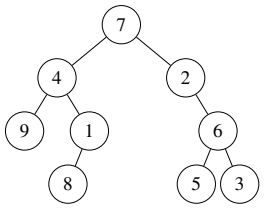
Queue:



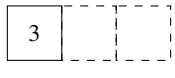
Level-order traversal:

7, 4, 2, 9, 1, 6, 8

During the eighth iteration, we take out node 5 and add it to our level-order traversal. 5 has no children, so nothing gets pushed into the queue.



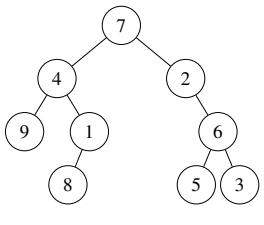
Queue:



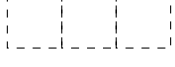
Level-order traversal:

7, 4, 2, 9, 1, 6, 8, 5

During the ninth iteration, we take out node 3 and add it to our level-order traversal. 3 has no children, so nothing gets pushed into the queue. After this iteration, the queue is empty, and our level-order traversal is complete.



Queue:



Level-order traversal:

7, 4, 2, 9, 1, 6, 8, 5, 3

Level-order traversals are often useful for solving tree problems that require you to obtain information about each level of a tree. However, some of these problems may require you to keep track of which nodes belong to each level (e.g., how does your algorithm know that node 2 is on the same level as node 4, but not node 9?).

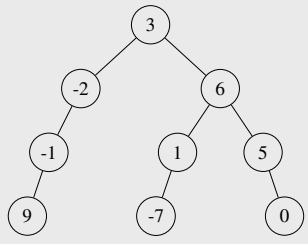
It turns out that we can use the size of the queue to obtain this information. Immediately after we finish processing all the nodes in a level, the size of the queue represents *the number of nodes in the next level of the tree*. For example, we start with the root, 7. We know that node 7 is the only node in its level, since it is the root. Thus, when we finish processing 7, the size of the queue represents the number of nodes on the second level (in this case, the size is 2, since nodes 4 and 2 are in the queue). Similarly, once we finish processing the second level (nodes 4 and 2), the size of the queue is 3 (nodes 9, 1, and 6). Thus, there must exist 3 nodes in the third level of the tree. As a result, if you are asked to solve a problem that requires you to distinguish between nodes at different levels of the tree, you can use the size of the queue to help you process the tree one level at a time. One example of such a problem is provided below:

Example 18.8 You are given the root of a binary tree whose nodes have the following structure:

```
struct Node {
  int32_t val;
  Node* left;
  Node* right;
};
```

Write a function that finds the *maximum level sum* of the binary tree. In other words, find the level of the tree that has the largest cumulative sum and return this value.

Example: Suppose you are given the following binary tree.



The first level has a sum of 3, the second level has a sum of -2 + 6 = 4, the third level has a sum of -1 + 1 + 5 = 5, and the fourth level has a sum of 9 - 7 + 0 = 2. Since 5 is the largest level sum in this tree, you would return 5.

To solve this problem, we will need to sum up all the nodes at each level and keep track of the largest level sum we've encountered so far. A level-order traversal would be ideal for this problem, but our algorithm will need to process each level one at a time. As a result, we can use the size of our traversal queue to determine how many nodes we need to sum at each level.

To do this, we will add a slight adjustment to the original level order traversal so that each iteration of the `while` loop processes an *entire level* of the tree rather than a single node. This can be done by calculating the size of the queue at the beginning of the while loop (which we will denote as *n*), and then running an inner loop that processes *n* nodes at a time. We add up all the nodes in this loop, and if the result is larger than the largest sum we've encountered so far, we update this largest sum. After the entire tree is processed, we return this value.

The code is shown below:

```
int32_t max_level_sum(Node* root) {
  int32_t curr_max = std::numeric_limits<int32_t>::min();  // smallest int
  std::queue<Node*> bfs;
  bfs.push(root);
  while (!bfs.empty()) {
    size_t level_size = bfs.size();  // number of nodes in next level
    int32_t level_sum = 0;
    // loop through all the nodes in the level
    for (size_t i = 0; i < level_size; ++i) {
      Node* curr = bfs.front();
      bfs.pop();
      level_sum += curr->val;        // add element to level sum
      if (curr->left) {
        bfs.push(curr->left);        // push left child if exists
      } // if
      if (curr->right) {
        bfs.push(curr->right);       // push right child if exists
      } // if
    } // for i
    // if level_sum is larger than largest level sum we've seen, update
    if (level_sum > curr_max) {
      curr_max = level_sum;
    } // if
  } // while
  return curr_max;
} // max_level_sum()
```

18.4 Solving Problems Using Trees

Because trees are recursive structures, we can use recursion to elegantly solve many types of tree problems. A common approach toward solving tree problems is to use a *traversal* method to traverse the tree in a certain manner. In this section, we will focus on problems that can be solved using a preorder or postorder traversal. In general, many of these tree problems can be solved using the following four-step approach:

- Identify the base case of the problem. The base case should run if the input allows you to solve the problem trivially.
- Process the left subtree recursively (this solves the problem for the subtree rooted at the left child).
- Process the right subtree recursively (this solves the problem for the subtree rooted at the right child).
- Complete the work needed to solve the problem for the subtree rooted at the current node (e.g., combine the results from the left and right subtrees to obtain the solution for the entire tree).

The ordering of bullets 2, 3, and 4 above depends on the problem you are trying to solve. If you need to operate on a node before you can operate on its children, bullet point 4 would go before bullet points 2 and 3, and the overall approach would be similar to a preorder traversal. On the other hand, if you need to operate on a node's children before you can operate on the node itself, bullet point 4 would go after bullet points 2 and 3, and the overall approach would be similar to a postorder traversal. We will look at a few examples below.

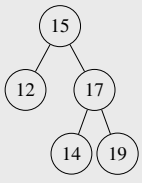
Example 18.9 You are given a binary tree, where each node is represented as follows:

```
1  struct Node {
2    int32_t val;
3    Node* left;
4    Node* right;
5  };
```

Write a function that returns the maximum height of the binary tree. In other words, return the number of nodes in the longest path from the root node to the farthest leaf node. The function header is:

```
int32_t find_max_height(Node* root);
```

Example: Given the following binary tree, you would return 3, since the root node has a height of 3.



When you are presented with a tree problem, it is beneficial to think recursively. Given a node, your function should return the maximum height of the binary tree that is rooted at that node. Would knowing the maximum heights of the node's left and right children help you solve this problem? If so, we can use the steps above to solve the problem recursively.

In this case, knowing the maximum heights of a node's children does help us find the maximum height of the node itself. If a node's left child has a height of L, and its right child has a height of R, the height of the node itself must be $\max(L, R) + 1$. Knowing this, we can implement a solution to this problem using a postorder traversal.

Step 1. Identify the base case of the problem.

In this problem, the base case runs if we are given an empty tree (`root == nullptr`). This is because we can find the height of an empty tree trivially — an empty tree must have a height of 0.

```
1  int32_t find_max_height(Node* root) {
2    // Step 1: Identify the base case (case that can be solved trivially)
3    if (!root) {
4      return 0;   // if root is nullptr, height must be 0
5    } // if
6    ...
7  } // find_max_height()
```

Steps 2 and 3. Process the left and right subtrees recursively.

Now, we want to retrieve the maximum heights of the left and right subtrees. Since trees are recursive structures, we can accomplish this by recursively calling `find_max_height()` on the left and right children (lines 7 and 8). This will allow us to obtain the maximum heights of the left and right children:

```
1  int32_t find_max_height(Node* root) {
2    // Step 1: Identify the base case (case that can be solved trivially)
3    if (!root) {
4      return 0;   // if root is nullptr, height must be 0
5    } // if
6    // Steps 2 and 3: Recursively process the left and right children
7    int32_t height_left_child = find_max_height(root->left);
8    int32_t height_right_child = find_max_height(root->right);
9    ...
10 } // find_max_height()
```

Step 4. Complete the work needed to solve the problem for the subtree rooted at the current node.

Now that we've obtained the maximum height of the left and right subtrees, we have to combine these results to calculate the maximum height of the node we are currently processing. In this problem, we know that the height of a node is equal to one plus the largest height of any of its children. We can implement this as follows:

```
1   int32_t find_max_height(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;   // if root is nullptr, height must be 0
5     } // if
6     // Steps 2 and 3: Recursively process the left and right children
7     int32_t height_left_child = find_max_height(root->left);
8     int32_t height_right_child = find_max_height(root->right);
9     // Step 4: Combine the results from the left and right subtrees
10    return 1 + max(height_left_child, height_right_child);
11  } // find_max_height()
```

Our function is now complete.

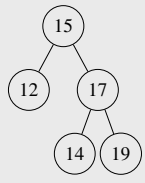
Example 18.10 You are given a binary tree, where each node is represented as follows:

```
1   struct Node {
2     int32_t val;
3     Node* left;
4     Node* right;
5   };
```

Write a function that returns the *minimum* depth of any leaf node in the binary tree. In other words, return the depth of the shallowest leaf node in the tree (where the root has a depth of 1). The function header is:

```
int32_t find_min_depth(Node* root);
```

Example: Given the following binary tree, you would return 2, since the minimum depth of any leaf node is 2 (for the node with value 12).



Once again, this problem can be solved recursively. For any node in the tree, if we know the minimum depth of any node in its left and right subtrees, we can use this information to calculate the minimum depth of the node itself. Thus, we will follow the steps of a postorder traversal to solve this problem.

Step 1. Identify the base case of the problem.

In this problem, the base case runs if we are given an empty tree (`root == nullptr`). This is because we can find the minimum depth of an empty tree trivially — an empty tree must have a minimum depth of 0.

```
1   int32_t find_min_depth(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;   // if root is nullptr, min depth must be 0
5     } // if
6     ...
7   } // find_min_depth()
```

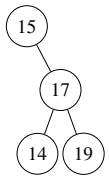
Steps 2 and 3. Process the left and right subtrees recursively.

Now, we want to retrieve the minimum leaf depths of the left and right subtrees. Since trees are recursive structures, we can accomplish this by recursively calling `find_min_depth()` on the left and right children (lines 7 and 8). This will allow us to obtain the minimum leaf depths of the subtrees rooted at each child:

```
1   int32_t find_min_depth(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;   // if root is nullptr, min depth must be 0
5     } // if
6     // Steps 2 and 3: Recursively process the left and right children
7     int32_t depth_left_child = find_min_depth(root->left);
8     int32_t depth_right_child = find_min_depth(root->right);
9     ...
10  } // find_min_depth()
```

Step 4. Complete the work needed to solve the problem for the subtree rooted at the current node.

Now that we've obtained the results for the left and right children, we have to combine them to find the minimum leaf depth of the subtree rooted at the node we are currently processing. Intuitively, if a node's left child has a minimum leaf depth of L, and its right child has a minimum leaf depth of R, the minimum leaf depth of the tree rooted at the node itself should be $\min(L, R) + 1$. However, consider the following tree:



Here, the left child of the root has a minimum depth of 0 (since it is a `nullptr`), and the right child of the root has a minimum depth of 2. However, if we just took the minimum of 0 and 2 and added 1, we would end up concluding that the root node has a minimum depth of 0 + 1 = 1. This would be incorrect! Instead, if either L or R is equal to zero, we want to add one to the *maximum* of L and R so that we aren't including `nullptr` in our calculations. The code is shown below:

```
1   int32_t find_min_depth(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;   // if root is nullptr, min depth must be 0
5     } // if
6     // Steps 2 and 3: Recursively process the left and right children
7     int32_t depth_left_child = find_min_depth(root->left);
8     int32_t depth_right_child = find_min_depth(root->right);
9     // Step 4: Combine the results from the left and right subtrees
10    if (depth_left_child == 0 || depth_right_child == 0) {
11      return 1 + max(depth_left_child, depth_right_child);
12    } // if
13    return 1 + min(depth_left_child, depth_right_child);
14  } // find_min_depth()
```

We can also write lines 10-13 in one line using the ternary/conditional operator:

```
return 1 + (min(depth_left_child, depth_right_child) ?
            min(depth_left_child, depth_right_child) :
            max(depth_left_child, depth_right_child));
```

Our function is now complete.

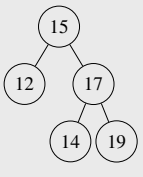
Example 18.11 You are given a binary tree, where each node is represented as follows:

```
1   struct Node {
2     int32_t val;
3     Node* left;
4     Node* right;
5   };
```

Write a function that returns the sum of all the left leaves in a given binary tree. The function header is:

```
int32_t left_leaf_sum(Node* root);
```

Example: Given the following binary tree, you would return 26, since that is the sum of all left leaves (12 + 14).



Like before, this problem can also be solved recursively. For any node in the tree, if we know the total left leaf sum of the subtrees rooted at the left and right children, we can use this information to calculate the total left leaf sum of the tree rooted at the node itself. Thus, we will follow the steps of a postorder traversal to solve this problem.

Step 1. Identify the base case of the problem.

In this problem, the base case runs if we are given an empty tree (`root == nullptr`). This is because the left leaf sum of an empty tree is trivially 0.

```
1   int32_t left_leaf_sum(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;  // if root is nullptr, left leaf sum must be 0
5     } // if
6     ...
7   } // left_leaf_sum()
```

Steps 2 and 3. Process the left and right subtrees recursively.

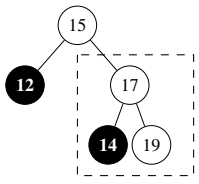
Now, we want to retrieve the left leaf sums of the left and right subtrees. Since trees are recursive structures, we can accomplish this by recursively calling `left_leaf_sum()` on the left and right children (lines 7 and 8). This will allow us to obtain the left leaf sums of the left and right children:

```
1   int32_t left_leaf_sum(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;  // if root is nullptr, left leaf sum must be 0
5     } // if
6     // Steps 2 and 3: Recursively process the left and right children
7     int32_t sum_left_child = left_leaf_sum(root->left);
8     int32_t sum_right_child = left_leaf_sum(root->right);
9     ...
10  } // left_leaf_sum()
```

Step 4. Complete the work needed to solve the problem for the subtree rooted at the current node.

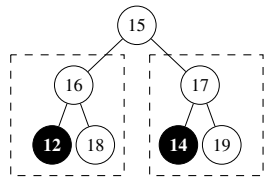
Now we have to combine these results to calculate the left leaf sum of the node we are currently processing. There are two cases that can happen. If the node's left child is a leaf node, we would combine the value of the left child with the left leaf sum of the right subtree. If the node's left child is not a leaf, we would combine the left leaf sum of the left subtree with the left leaf sum of the right subtree.

`root->left->val + left_leaf_sum(root->right)`



recurse into right subtree to get left leaf sum of 14

`left_leaf_sum(root->left) + left_leaf_sum(root->right)`



recurse into left subtree to get left leaf sum of 12

recurse into right subtree to get left leaf sum of 14

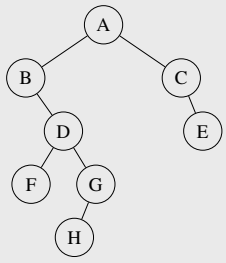
The code from step 4 is shown below:

```
1   int32_t left_leaf_sum(Node* root) {
2     // Step 1: Identify the base case (case that can be solved trivially)
3     if (!root) {
4       return 0;  // if root is nullptr, left leaf sum must be 0
5     } // if
6     // Steps 2 and 3: Recursively process the left and right children
7     int32_t sum_left_child = left_leaf_sum(root->left);
8     int32_t sum_right_child = left_leaf_sum(root->right);
9     // Step 4: Combine the results from the left and right subtrees
10    if (root->left && !root->left->left && !root->left->right) {
11      return root->left->val + sum_right_child;
12    } // if
13    return sum_left_child + sum_right_child;
14  } // left_leaf_sum()
```

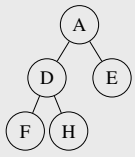
Our function is now complete. Note that we can further optimize the function above by making a recursive call to the left subtree only when the left child is not a leaf.

Example 18.12 A binary tree is defined to be a *full* (or *proper*) binary tree if every node either has zero or two children. In other words, for a binary tree to be full, every internal node must have two children. Given a binary tree, write a function that turns this tree into a full binary tree by removing all nodes that have only one child.

Example: Given the following tree:



you would remove nodes B, C, and G, since they only have one child. The resulting tree would look like this:



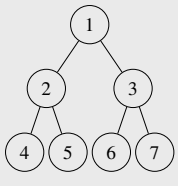
To solve this problem, we will have to traverse the binary tree and identify all the nodes that only have one child. However, the method we use to traverse the tree is important, since we must recursively process the left and right children of a node before we can delete the node itself. Thus, processing the tree in a bottom-up fashion should be the way to go, and a postorder traversal should be used.

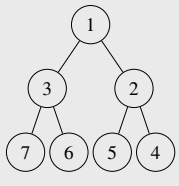
Our function can be written as follows. First, we recursively process the left and right children to remove nodes with only one child in these subtrees. Then, we check if the current node has one child. If it does, we delete the current node and attach its child to its parent. Otherwise, we ignore the node and continue traversing the rest of the tree. The code is shown below:

```
1   Node* remove_single_children(Node* root) {
2     // Base case: empty tree
3     if (!root) {
4       return nullptr;
5     } // if
6     // Recursively process the left and right children
7     root->left = remove_single_children(root->left);
8     root->right = remove_single_children(root->right);
9     // If node has 0 or 2 children, no need to do anything (just return node)
10    if ((!root->left && !root->right) || (root->left && root->right)) {
11      return root;
12    } // if
13    // If node has 1 child, delete node and replace with child
14    Node* child = root->left ? root->left : root->right;
15    delete root;
16    return child;
17  } // remove_single_children()
```

Example 18.13 Write a function that can be used to invert (mirror) a binary tree.

Example: Given the following tree on the left, the function would turn it into the tree on the right:





To solve this problem, we will need to traverse the tree and flip each node's left and right children. However, a node's left and right children must be flipped *before* we can recursively invert the children of that node. As a result, we have to complete the work for the current node (swapping its children) before we make recursive calls to the left and right children. Thus, we should process the tree from top to bottom, and a preorder traversal should be used.

Our function can be written as follows. As we traverse through the tree, we first swap the left subtree of the current node with the right subtree. Then, we recursively invert the left and right subtrees of the current node. The code is shown below:

```
1  void invert_binary_tree(Node* root) {
2    // Base case: empty tree
3    if (!root) {
4      return;
5    } // if
6    // Swap left subtree and right subtree
7    std::swap(root->left, root->right);
8    // Recursively invert left and right subtrees
9    invert_binary_tree(root->left);
10   invert_binary_tree(root->right);
11 } // invert_binary_tree()
```

18.5 Constructing Binary Trees from Preorder, Inorder, and Postorder Traversals

If you are given two traversals of a binary tree: (1) an inorder traversal and (2) either a preorder or postorder traversal, you can use these traversals to construct a unique binary tree. This is because you can use the preorder or postorder traversal to determine the root of each subtree, and the inorder traversal to partition the remaining elements into a left and right subtree. We will look at a few examples in this section.

※ 18.5.1 *Constructing a Binary Tree from Inorder and Preorder Traversals*

Example 18.14 You are given a binary tree with the following **inorder** and **preorder** traversals. Use these traversals to draw out this tree.

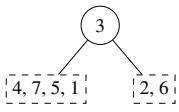
Inorder: 5, 7, 1, 4, 3, 6, 2
Preorder: 3, 4, 7, 5, 1, 2, 6

First, we will use the preorder traversal to determine the root. In a preorder traversal, the root is always visited first, so we know that 3 must be the root of the entire tree.

Inorder: 5, 7, 1, 4, 3, 6, 2
Preorder: **3**, 4, 7, 5, 1, 2, 6

Once we have identified the root, we will use the inorder traversal to partition the remaining elements into left and right subtrees. Since an inorder traversal traverses the tree "in order," all elements in the traversal that come before 3 must be to the left of 3, and all elements in the traversal that come after 3 must be to the right of 3.

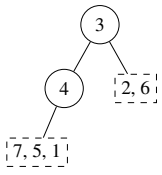
Inorder: 5, 7, 1, 4, 3, 6, 2
Preorder: 3, 4, 7, 5, 1, 2, 6



We now repeat the same procedure for each of the subtrees. We set the element that comes first in the preorder traversal as the root of the subtree, and then we use the inorder traversal to partition the elements into left and right subtrees.

In the example, the elements 4, 7, 5, and 1 make up the left subtree. Of these elements, 4 is first in the preorder traversal, so 4 must be the root of the left subtree. Knowing this, we will now look for the remaining elements in the left subtree (7, 5, and 1) in our inorder traversal. All three of these elements precede 4 in the inorder traversal, so they must all be to the left of 4.

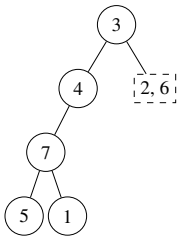
Inorder: 5, 7, 1, 4, 3, 6, 2
Preorder: 3, 4, 7, 5, 1, 2, 6



The elements 7, 5, and 1 make up the left subtree of 4. Since 7 appears first in the preorder traversal, it must be the root of this subtree. If we look at the inorder traversal, 5 comes before 7, and 1 comes after 7. Thus, 5 must be to the left of 7, and 1 must be to the right of 7. The entire left subtree is now complete.

Inorder: 5, [7], 1, 4, 3, 6, 2

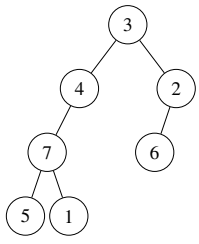
Preorder: 3, 4, 7, 5, 1, 2, 6



We can repeat this process for the right subtree of the root. Since 2 appears before 6 in the preorder traversal, 2 must be the root of the right subtree. Since 6 appears before 2 in the inorder traversal, 6 must be to the left of 2, and thus is the left child of 2. The entire tree is now complete.

Inorder: 5, 7, 1, 4, 3, 6, [2]

Preorder: 3, 4, 7, 5, 1, 2, 6



※ 18.5.2 *Constructing a Binary Tree from Inorder and Postorder Traversals*

Example 18.15 You are given a binary tree with the following **inorder** and **postorder** traversals. Use these traversals to draw out this tree.

Inorder: 3, 6, 5, 4, 2, 1, 7
Postorder: 3, 6, 4, 2, 7, 1, 5

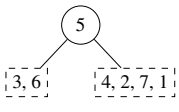
Solution: First, we will use the postorder traversal to determine the root. In a postorder traversal, the root is always visited last, so we know that 5 must be the root of the entire tree.

Inorder: 3, 6, 5, 4, 2, 1, 7
Postorder: 3, 6, 4, 2, 7, 1, **5**

Now that we have identified the root, we will use the inorder traversal to partition the remaining elements into left and right subtrees. All elements in the inorder traversal that come before 5 must be to the left of 5, and all elements that come after 5 must be to the right of 5.

Inorder: 3, 6, [5], 4, 2, 1, 7

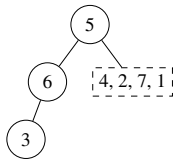
Postorder: 3, 6, 4, 2, 7, 1, 5



The left subtree of 5 includes 3 and 6. Since 6 comes last in the postorder traversal for this subtree, 6 must be the root of the left subtree. Now, we will determine where 3 goes using the inorder traversal. In the inorder traversal, 3 comes before 6, so 3 must be the left child of 6.

Inorder: 3, [6], 5, 4, 2, 1, 7

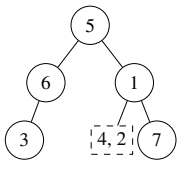
Postorder: 3, 6, 4, 2, 7, 1, 5



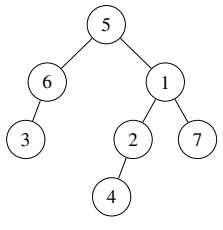
We can repeat this process for the right subtree of 5. Of the four elements in the right subtree, 1 appears last in the postorder traversal, so it must be the root of the right subtree. Now, we can look at the inorder traversal to determine the positions of the other elements. Since 4 and 2 come before 1 in the inorder traversal, they must be in the left subtree of 1. Since 7 comes after 1 in the traversal, 7 must be the right subtree of 1.

Inorder: 3, 6, 5, 4, 2, [1], 7

Postorder: 3, 6, 4, 2, 7, 1, 5



Since 2 comes after 4 in the postorder traversal, 2 must be the root of 1's left subtree. Using the inorder traversal, 4 comes before 2, so 4 must be the left child of 2. Our tree is now complete.

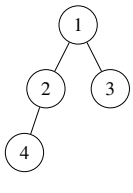


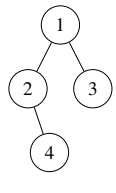
※ 18.5.3 *Constructing a Binary Tree from Preorder and Postorder Traversals*

If you are given either a preorder or postorder traversal, you can build a unique binary tree as long as you are also given the inorder traversal. However, if you are only given a preorder and a postorder traversal, you may not be able to construct a unique binary tree. Without the inorder traversal, there is no way to determine if a node should go to the left or right of the root, which could cause issues if a node only has one child.

Example 18.16 Draw two binary trees that have different structures but share the same preorder and postorder traversals. If this is not possible, explain why.

The easiest way to construct two different trees with the same preorder and postorder traversals is to draw a tree where one of the nodes only has a left child, and then draw the same tree with this left child as the right child. For example, the following two trees share the exact same preorder and postorder traversals (preorder: 1, 2, 4, 3; postorder: 4, 2, 3, 1). This is because, without the inorder traversal, we cannot determine if a child goes to the left or right of its parent.





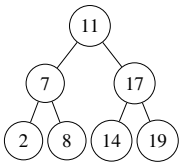
18.6 Binary Search Trees

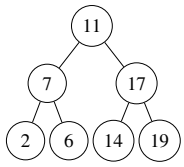
※ 18.6.1 *Binary Search Tree Structure*

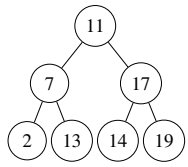
One pitfall of the standard binary tree is that its elements are not ordered in any manner. This can make searching difficult: if you wanted to find a given element, that element could be anywhere in the tree! To address this issue, we will introduce the **binary search tree (BST)**, which is a binary tree whose elements are ordered based on a sorting invariant. A standard non-empty binary search tree exhibits the following properties:

- The key of any node is always greater than all of the keys in its left subtree.
- The key of any node is always less than or equal to all of the keys in its right subtree.
- Both the left and right subtrees of any node are also binary search trees themselves.

For example, consider the following three binary trees:



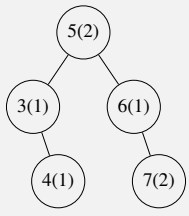




Only the first tree is a binary search tree. The second tree is not a binary search tree because the right child of 7, or 6, is smaller than 7. The third tree is not a binary search tree because 13 is in the left subtree of 11, even though 13 is larger than 11.

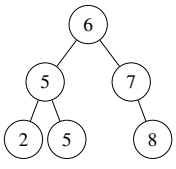
Remark: In the definition above, we specify that equal keys in a binary search tree always go into the right subtree. However, does it matter which side equal values are sent to? As long as you are consistent with where you put duplicates, it shouldn't matter. In our definition, we send duplicates to the right subtree because this allows our tree to be implemented using just `operator<`, the default operator often used to order STL containers (i.e., if `operator<`, send to left subtree, else send to right subtree). However, it is possible for a tree to be defined differently and still be a valid binary search tree.

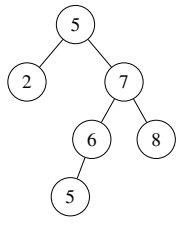
Another convention for managing duplicates would be to store a counter for each element in the tree representing the number of copies that element has in the tree. For example, we can use the following to represent a binary tree that has a duplicate 5 and a duplicate 7 (the number in parentheses is the counter associated with the key):



This approach has its advantages: it makes the height of the BST independent of the number of duplicates, and it may make the balancing process easier for certain types of self-balancing BSTs (the concept of balancing will be covered in the next section). However, we will *not* be using this convention in this class, unless otherwise stated.

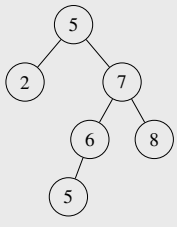
For a tree to be a valid binary search tree, only the rules specified above need to be met. As a result, it is possible for two binary search trees to have different structures, even if they have the same keys. Two examples are shown below:





Since keys in a binary search tree are ordered in a way such that all keys to the left are smaller and all keys to the right are larger, the inorder traversal of a binary search tree will always visit its keys in sorted order.

Example 18.17 Consider the following binary search tree. What is the inorder traversal of this BST?



The inorder traversal of a binary search tree always visits the keys in sorted order. Therefore, the inorder traversal of the tree is: 2, 5, 5, 6, 7, 8.

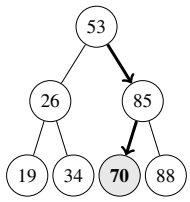
Example 18.18 You are given a binary search tree with n nodes. What is the worst-case time complexity of printing all the keys in this binary search tree in sorted order?

Since the tree is a binary search tree, we know that its inorder traversal would visit the keys in sorted order. Thus, all we need to do is to conduct an inorder traversal of the tree and print each key out along the way. Since an inorder traversal visits each node at most once, the worst-case time complexity of this task is $\Theta(n)$.

※ 18.6.2 *Searching in a Binary Search Tree*

Binary search trees support efficient searching, since you will only need to look down one side of the tree at each level. If you want to search for a target key k, and k is smaller than the current root element you are looking at, then k must be in the left subtree if it exists. Otherwise, if k is larger than the current root element, then k must be in the right subtree if it exists. You can think of this as a "binary search" of the tree's elements, since you are removing half of the search space every time; hence why this data structure is called a *binary search* tree!

For example, if we wanted to search for 70 in the following binary search tree, we would first compare 70 with the root, 53. Since 70 is larger than 53, we would look down the right subtree of 53, rooted at 85. We would then compare 70 with 85; since 70 is smaller, we would look down the left subtree of 85, rooted at 70. We've successfully found 70, so our search is complete.

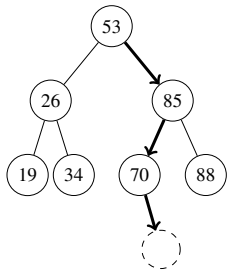


70 > 53, so look down right subtree

70 < 85, so look down left subtree

we found 70, so search is complete

If we ever reach a `nullptr` without finding the key we want to find, then the key must not exist in the binary search tree. For example, if we wanted to search for 71 in the tree, we would attempt to look down the right subtree of 70. However, 70 has no right child, so we can conclude that 71 doesn't exist in the tree.



71 > 53, so look down right subtree

71 < 85, so look down left subtree

71 > 70, so look down right subtree

we hit a `nullptr`, so 71 must not exist

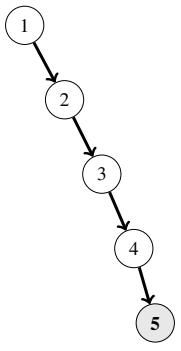
The searching algorithm can be implemented both iteratively and recursively. The following iterative implementation returns a pointer to the node with key k if it exists; otherwise, it returns `nullptr`.

```
1   template <typename T>
2   struct Node {
3     T val;       // assume T supports operator<
4     Node* left;
5     Node* right;
6   };
7
8   template <typename T>
9   Node<T>* tree_search(Node<T>* root, T k) {
10    while (root != nullptr && k != root->val) {
11      if (k < root->val) {
12        root = root->left;
13      } // if
14      else {
15        root = root->right;
16      } // else
17    } // while
18    return root;
19  } // tree_search()
```

The following implementation does the same thing, but with recursion instead of iteration. Note that this implementation is tail recursive, since the recursive call is always done last.

```
1   template <typename T>
2   struct Node {
3     T val;       // assume T supports operator<
4     Node* left;
5     Node* right;
6   };
7
8   template <typename T>
9   Node<T>* tree_search(Node<T>* root, T k) {
10    if (root == nullptr || root->val == k) {
11      return root;
12    } // if
13    if (k < root->val) {
14      return tree_search(root->left, k);
15    } // if
16    return tree_search(root->right, k);
17  } // tree_search()
```

What is the time complexity of searching for a key in a binary search tree? The time complexity of search depends on the height of the tree. In the best case, the key you want to find is in the root, allowing you to find it in $\Theta(1)$ time. However, in the worst-case, the binary search tree could be in the form of a stick, and you may have to search the entire tree to find a given key. This results in $\Theta(n)$ runtime, where n is the number of nodes in the tree. For instance, if you wanted to search for 5 in this tree, you would have to traverse through every node:



5 > 1, so look down right subtree

5 > 2, so look down right subtree

5 > 3, so look down right subtree

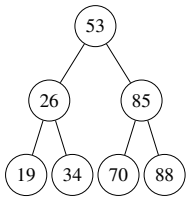
5 > 4, so look down right subtree

we found 5, so search is complete

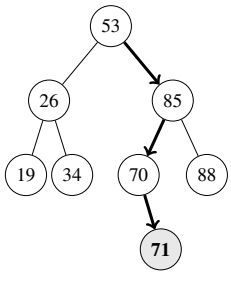
What about the average case? On average, a binary search tree is relatively balanced (i.e., there are about as many items to the left as there are to the right), allowing you to ignore half of the remaining elements at every level. Since we are halving the search space at every iteration, the average-case time complexity of search on a binary search tree is $\Theta(\log(n))$. Note that $\Theta(\log(n))$ is also the height of an average-case binary search tree; as mentioned, the complexity of search depends on a tree's height.

※ 18.6.3 *Inserting into a Binary Search Tree*

To insert a key into a binary search tree, we follow a procedure that is similar to search: we start at the root and trace a path downwards, but this time we want to look for a `nullptr` position to append the node. For example, suppose we wanted to insert 71 into the binary search tree that was introduced earlier:



We look down the tree in a similar manner. However, once we find a `nullptr` position that 71 can go in, we insert the new node there.



71 > 53, so look down right subtree

71 < 85, so look down left subtree

71 > 70, so look down right subtree

we hit a `nullptr`, so insert 71 here

The implementation for BST insertion is shown below. Note that the function is passed in a *pointer by reference* (`*&root`), which allows us to directly assign the new node to its desired position. By passing the pointer by reference, any changes that the `tree_insert()` function makes to the pointer will also be reflected in the calling function. In other words, if `tree_insert()` changes what the pointer is pointing at, the pointer is also changed in the function that called `tree_insert()`.

```
template <typename T>
struct Node {
  T val;       // assume T supports operator<
  Node* left;
  Node* right;
  Node(T k) : val{ k } {}
};

template <typename T>
void tree_insert(Node<T>*& root, T k) {
  if (root == nullptr) {
    root = new Node{k};
  } // if
  else if (k < root->val) {
    tree_insert(root->left, k);
  } // else if
  else {
    tree_insert(root->right, k);
  } // else
} // tree_insert()
```

Similar to search, the time complexity of insert depends on the height of the tree. In the best case, the new node can be directly inserted as either the left or right child of the initial root, which would take $\Theta(1)$ time. In the worst case, the binary search tree could be in the form of a stick, requiring the algorithm to traverse the entire tree before finding the position to insert the new node — this would take $\Theta(n)$ time. On average, however, the binary search tree should be relatively balanced, allowing insertions to be done in $\Theta(\log(n))$ time.

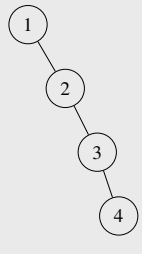
Example 18.19 You are given a binary search tree with n nodes. Write a function to find the node with the smallest key. What are the average-case and worst-case time complexities of this function?

If you are looking at any node in a binary search tree that has a left child, that left child must have a smaller value than the node you are looking at. Thus, to find the minimum value in a binary search tree, keep on recursing on the left child until you reach a node that has no left child at all. The code is shown below:

```
Node* bst_tree_min(Node* root) {
  if (root == nullptr) {
    return nullptr;
  } // if
  while (root->left) {
    root = root->left;
  } // while
  return root;
} // bst_tree_min()
```

In the worst case, you could end up getting a left-facing stick, which could cause this function to visit every node before finding the node with the smallest key, taking $\Theta(n)$ time. However, given an average-case tree that is relatively balanced, the time complexity of finding the smallest key would be $\Theta(\log(n))$.

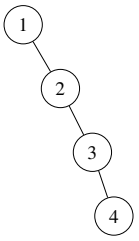
Example 18.20 We define a *stick* as a tree in which none of its nodes have any siblings. When dealing with binary search trees, we often consider sticks to be "worst-case" trees, since they exhibit worst-case $\Theta(n)$ behavior for search, insertion, and removal. For instance, the following binary search tree is a stick:



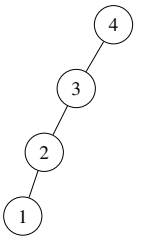
Suppose you wanted to insert the following four keys into a binary search tree: 1, 2, 3, and 4. How many insertion orders would end up creating a stick? What if you wanted to insert n unique keys into a binary search tree instead of 4 — how many insertion orders, in terms of n, would end up creating a stick?

There are two intuitive cases: if you insert the values in ascending or descending order, you will create a rightward or leftward facing stick:

Insertion order: 1, 2, 3, 4

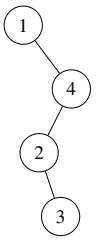


Insertion order: 4, 3, 2, 1

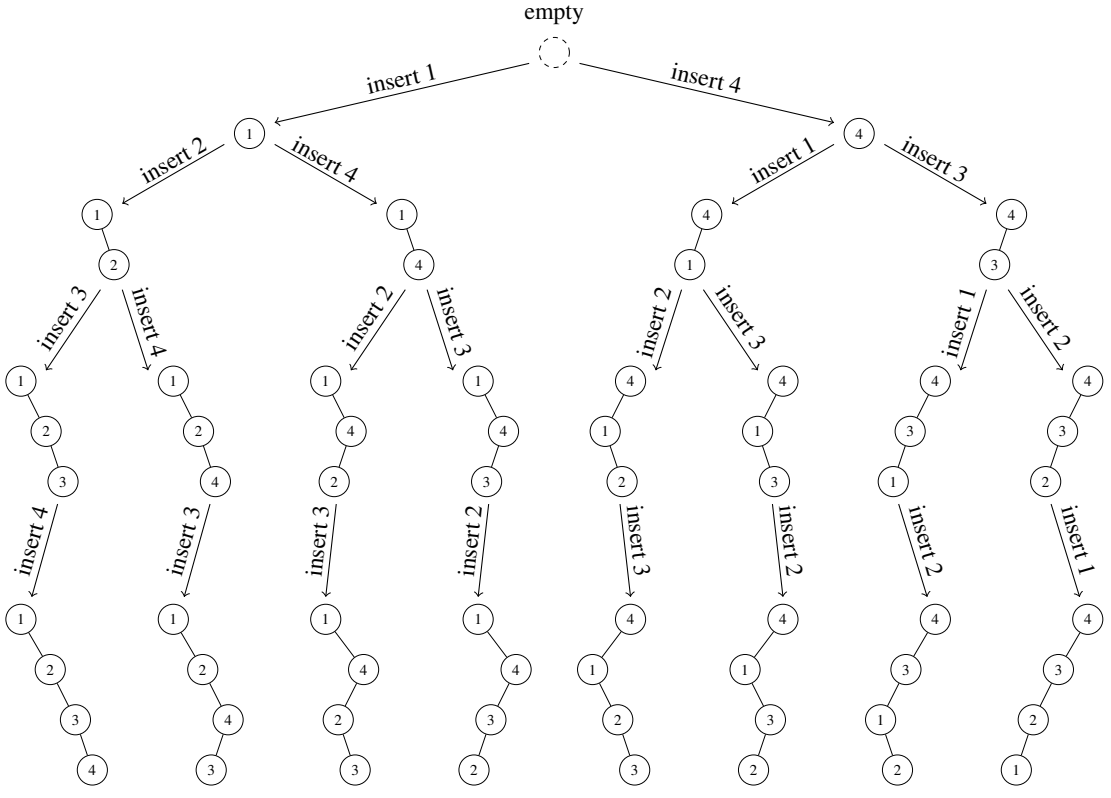


However, these are not the only two situations where you could end up with a stick! A stick does not have to be a straight line — for instance, the following tree would also be considered a stick:

Insertion order: 1, 4, 2, 3



It turns out you can create a stick tree if, at every step, you always insert the smallest or largest of the remaining elements into the tree. For instance, you can either insert 1 or 4 into the tree first if you want to create a stick. If you insert 1 first, the next insertion can either be 2 or 4 (since 2 is the smallest of the remaining elements, and 4 is the largest). Similarly, if you insert 4 first, the next insertion can either be 1 or 3. We can express these options using a decision tree, as shown below:



As shown, there are eight possible stick orientations that can be constructed using the four unique keys. Since you can insert either the smallest or largest of the remaining elements at every step of the process, there are two values you can choose for the first insertion, two values you can choose for the second insertion, and two values you can choose for the third insertion. Once you insert the first three elements, you only have one choice left for the fourth insertion, since there is only one element left. Putting this together, the total number of sticks we can construct using four unique keys is equal to $2 \times 2 \times 2 \times 1 = 8$, as illustrated above.

We can generalize this process for n insertions. If you want to insert n unique keys to create a stick, there are two values you can choose for the first insertion (i.e., smallest or largest), two values you can choose for the second insertion, …, and two values you can choose for the $(n-1)^{\text{th}}$ insertion. Once you insert the $(n-1)^{\text{th}}$ element, you only have one choice left for the last insertion, as there is only one element left. Since you can make two choices for the first $(n-1)$ insertions, and one choice for the n^{th} insertion, there are a total of $2^{n-1} \times 1 = 2^{n-1}$ insertion orders that would yield a stick.

Remark: If you are given n unique keys to insert into a binary search tree, there are a total of $n!$ insertion orders that are possible for these elements (since you have n choices for the first insertion, $n-1$ choices for the second insertion, and so on). This leads to an interesting question: how many different binary search trees can be constructed using n unique elements? The answer here is not $n!$, since multiple insertion orders could produce the same binary search tree (e.g., the insertion orders 2, 1, 3 and 2, 3, 1 produce the same tree).

To answer this question, you would have to enumerate through all the keys and count the number of unique binary search trees that can be constructed with each key as the root. If we denote the smallest key as key 0, the second smallest key as key 1, ..., and the largest key as key $n-1$, we can recursively express the total number of binary search trees that can be built with n unique keys as C_n below:

$$C_n = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \sum_{i=0}^{n-1}(C_i \times C_{n-i-1}), & \text{if } n > 1 \end{cases}$$

Here, $C_i \times C_{n-i-1}$ represents the total number of binary search trees that can be constructed if key i were the root (where i starts at 0). This is because there are i keys to the left of key i, and $n-i-1$ keys to the right of key i. As a result, there are C_i ways to construct the left subtree of key i and C_{n-i-1} ways to construct the right subtree of key i. Thus, by the fundamental counting principal, we can conclude that there are $C_i \times C_{n-i-1}$ binary search trees that can be constructed with key i as the root. The summation $\sum_{i=0}^{n-1}$ is then applied to enumerate through all keys as potential roots of the tree (i.e., the total number of BSTs for n unique keys = number of BSTs if key 0 were the root + number of BSTs if key 1 were the root + ... + number of BSTs if key $n-1$ were the root).

It turns out that this recurrence relation can be rewritten using the following closed-form formula. The proof has been omitted because the mathematics involved is too complex for this class, but there are many resources online that explain the process, if you are curious.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

In mathematics, the sequence of natural numbers C_n for $n = 0, 1, 2, 3, \ldots$ are known as the **Catalan numbers**, and they show up in many different counting problems.

Example 18.21 You are given the following seven integers, which you want to insert into an empty BST:

101, 183, 203, 280, 281, 370, 376

How many distinct BSTs can you construct using these seven elements? What percentage of these trees are sticks? What percentage of these trees are complete trees?

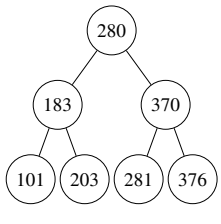
Using the equation above, we can calculate the number of unique binary search trees with seven elements as:

$$C_7 = \frac{14!}{8!7!} = 429$$

Thus, there are a total of 429 unique binary search trees that can be built using seven elements. In the previous example, we showed that there are a total of 2^{n-1} stick formations that can be constructed with n elements. Since $n = 7$, the total number of sticks we can generate is $2^{7-1} = 64$. The percentage of stick trees is therefore equal to

$$\frac{\text{number of sticks with 7 elements}}{\text{number of BSTs with 7 elements}} = \frac{64}{429} = 14.92\%$$

As for completeness, there is only one arrangement that allows the binary search tree to be complete (recall that completeness means that all levels but the bottom level must be completely filled, and the bottom level must be filled from left to right with no gaps). In fact, for any n, there is only one distinct structure to organize the nodes in for the entire tree to be complete. For this example, the configuration is shown below:



Thus, only 1 of the 429 possible BST configurations is complete, and the percentage of seven-node binary search trees that are complete is 1/429 = 0.23%.

Example 18.22 You are given a sorted array of n unique integers. Implement the `array_to_BST()` function, which takes in the sorted array and two indices `left` and `right`, and constructs a binary search tree containing values in the range `[left, right]` with the *minimum* possible height. Your function should run in worst-case $\Theta(n)$ time.

```
struct Node {
  int32_t val;
  Node* left;
  Node* right;
  Node(int32_t x) : val{ x }, left{ nullptr }, right{ nullptr } {}
};

Node* array_to_BST(int32_t arr[], size_t left, size_t right);
```

Since the array is sorted, the element in the middle of the sorted array must be the root if we want our final BST to be as balanced as possible. Furthermore, the element in the middle of the left half of the sorted array should be the left child of the root, and the element in the middle of the right half of the sorted array should be the right child of the root. We can therefore use recursion to write an elegant solution to this problem that runs in linear time.

```
Node* array_to_BST(int32_t arr[], size_t left, size_t right) {
  if (left > right) {
    return nullptr;
  } // if
  size_t mid = left + (right - left) / 2;
  Node* root = new Node(arr[mid]);
  root->left = array_to_BST(arr, left, mid - 1);
  root->right = array_to_BST(arr, mid + 1, right);
  return root;
} // array_to_BST()
```

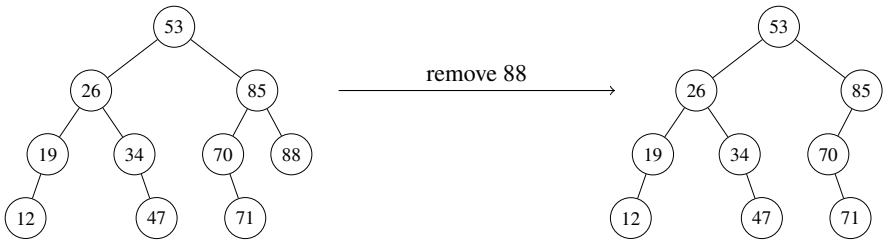
※ 18.6.4 *Removing from a Binary Search Tree*

So far, we have covered search and insertion. However, what if we wanted to remove a key from a binary search tree? Removal is slightly more complicated, since we will have to detach a node while still maintaining the sorted property of a binary search tree. There are four different conditions we have to consider when removing a node from a binary search tree:

1. The node we want to remove has no children.
2. The node we want to remove has no left child.
3. The node we want to remove has no right child.
4. The node we want to remove has two children.

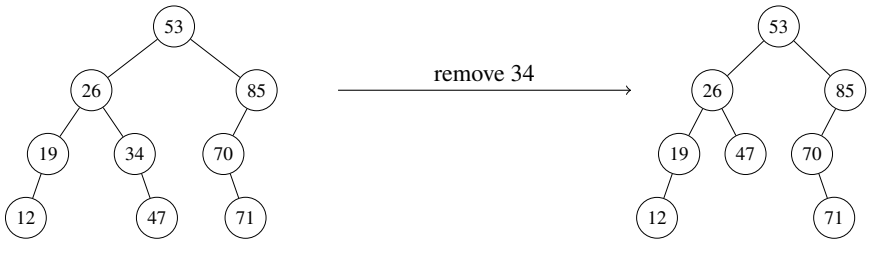
1. The node we want to remove has no children.

This condition is trivial; if we want to remove a leaf node, we can just snip off the leaf.



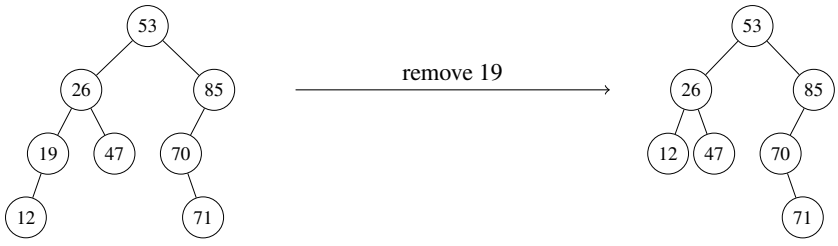
2. The node we want to remove has no left child.

If the node we want to remove has no left child, we can just replace it with its right child. In the example below, 34 has no left child, so we just replace it with its right child, 47.



3. The node we want to remove has no right child.

If the node we want to remove has no right child, we can just replace it with its left child. In the example below, 19 has no right child, so we just replace it with its left child, 12.

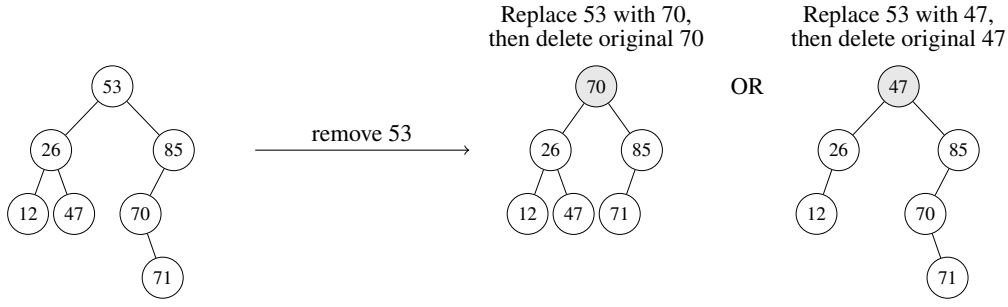


4. The node we want to remove has two children.

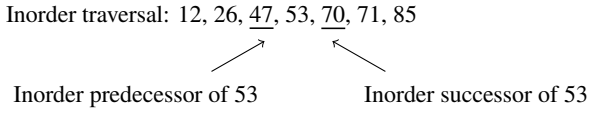
This is a trickier case, since the node to remove points to two things instead of one. As a result, we will need a procedure for replacing the deleted node in a way that preserves the binary search tree property.

A key observation to make here is that any node in the left subtree of the deleted node must be smaller than any node in the right subtree of the deleted node. Thus, we can preserve the binary search tree property by either (1) replacing the deleted node with the smallest node in its right subtree, or (2) replacing the deleted node with the largest node in its left subtree.

For example, suppose we wanted to remove 53 from the following binary search tree. We can either replace 53 with the smallest element in its right subtree (70), or with the largest element in its left subtree (47). Both approaches would maintain the binary search tree property:



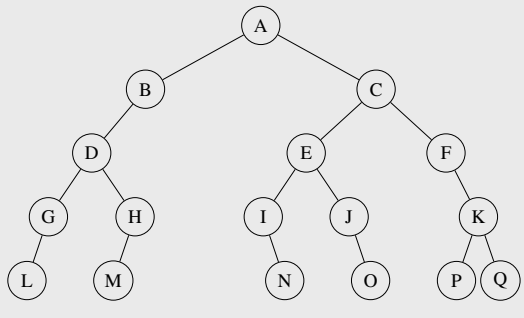
The smallest node in the right subtree of a given node k is known as the **inorder successor** of node k. Similarly, the largest node in the left subtree of a given node k is known as the **inorder predecessor** of node k. This is because the inorder successor of k comes directly after (i.e., succeeds) k in an inorder traversal, and the inorder predecessor of k comes directly before (i.e., precedes) k in an inorder traversal.



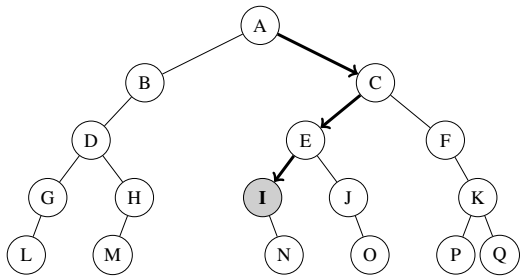
Because the inorder successor and predecessor are directly adjacent to the node we want to delete, it is safe to replace the deleted node with either of these values. For instance, if we replaced 53 with 70 (and then deleted the original 70 in 53's right subtree), the elements of the tree would still be in sorted order. This would also be the case if we replaced 53 with 47 (and then deleted the original 47 in 53's left subtree). Replacing a deleted node with either its inorder successor or predecessor is the simplest way to approach deletion, as it maintains the BST property without requiring you to reconfigure the other elements in the tree.

The process for finding the inorder successor or predecessor of any node is also relatively straightforward. To identify the inorder successor of a node, go right once, then go as far left as possible until you reach a node without a left child. To identify the inorder predecessor of a node, go left once, then go as far right as possible until you reach a node without a right child.

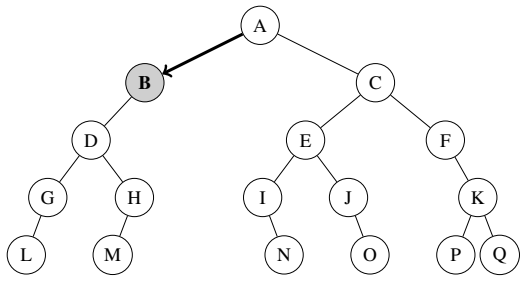
Example 18.23 Consider the following tree. What is the inorder successor and inorder predecessor of node A?



To identify the inorder successor, we go right once, and then go as far left as possible (until we hit a node with no left child). In this case, node I is the the inorder successor of node A:



To identify the inorder predecessor, we go left once, and then go as far right as possible (until we hit a node with no right child). In this case, we go left to B, but B has no right children. Thus, node B is the inorder predecessor of node A:



The code for remove is shown below. In this version, the deleted node is replaced with the inorder successor (the inorder predecessor version is not shown, but it follows the same logic).

```
1   template <typename T>
2   struct Node {
3     T val;       // assume T supports operator<
4     Node* left;
5     Node* right;
6   };
7
8   // Removes the node from the tree with value "val"
9   template <typename T>
10  void tree_remove(Node<T>*& root, const T& val) {
11    Node<T>* node_to_delete = root;
12    Node<T>* inorder_successor;
13    // Recursively finds the node containing the value ("val") to remove
14    if (root == nullptr) {
15      return;
16    } // if
17    else if (val < root->val) {
18      tree_remove(root->left, val);
19    } // else if
20    else if (root->val < val) {
21      tree_remove(root->right, val);
22    } // else if
23    else {
24      // Check for simple cases where at least one subtree is empty
25      if (root->left == nullptr) {
26        root = root->right;
27        delete node_to_delete;
28      } // if
29      else if (root->right == nullptr) {
30        root = root->left;
31        delete node_to_delete;
32      } // else if
33      else {
34        // Node to delete has both left and right subtrees
35        inorder_successor = root->right;
36        while (inorder_successor->left) {
37          inorder_successor = inorder_successor->left;
38        } // while
39        // Replace value with inorder successor's value
40        node_to_delete->val = inorder_successor->val;
41        // Remove the inorder successor from right subtree
42        tree_remove(root->right, inorder_successor->val);
43      } // else
44    } // else
45  } // tree_remove()
```

Like search and insert, the time complexity of remove depends on the height of the tree. In the average case, the tree is relatively balanced, allowing removal to be done in $\Theta(\log(n))$ time, where n is the number of nodes in the tree. This is because finding a node and its inorder successor or predecessor would take $\Theta(\log(n))$ time if the tree were balanced. In the worst case, you could end up with a stick, which could cause removal to run in up to $\Theta(n)$ time (since finding the inorder successor or predecessor of a stick may take $\Theta(n)$ time).

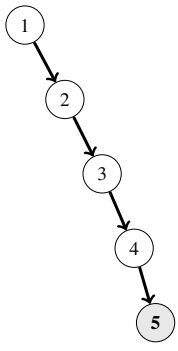
Summary of Binary Search Tree Time Complexities

Operation	Best-Case Time	Average-Case Time	Worst-Case Time
Finding a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Inserting a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Deleting a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$

18.7 AVL Trees

※ 18.7.1 *Properties of AVL Trees*

Binary search trees provide us with average-case $\Theta(\log(n))$ time complexities for search, insertion, and removal. However, they do *not* provide us with this guarantee in the worst case. The worst-case time complexity of search, insertion, and removal in a standard binary search tree with n nodes is $\Theta(n)$, since we may end up with a stick tree that forces us to traverse every node. For example, searching for 5 in the following binary search tree would require us to visit every node, resulting in a linear traversal.



5 > 1, so look down right subtree

5 > 2, so look down right subtree

5 > 3, so look down right subtree

5 > 4, so look down right subtree

we found 5, so search is complete

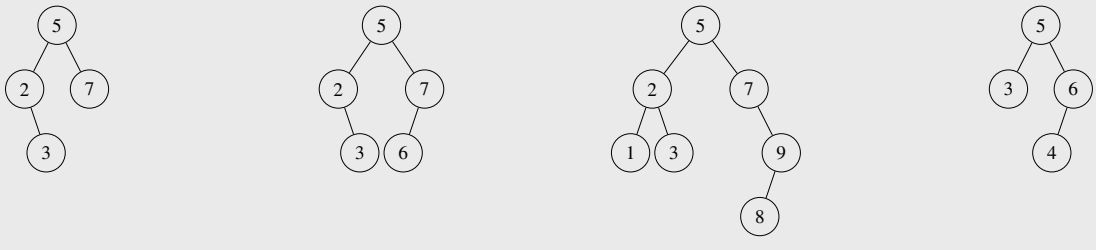
To obtain a better time complexity bound in the worst case, we need to ensure that this worst-case stick tree never happens. One method is to *balance* the tree with every insertion and deletion. By doing so, we can fix the tree whenever it becomes too lopsided, essentially preventing a stick tree from ever happening. If we can guarantee that the size of the left and right subtrees are relatively equal at all times, then the time complexities of search, insertion, and removal will be $\Theta(\log(n))$ in the worst case.

This brings us to the **AVL tree**, a self-balancing binary search tree named after its inventors, Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree that enforces the *height balance property*, which states that the heights of the left and right subtrees of any node in the tree can only differ by at most one. If this property is ever broken after an operation, the AVL tree will automatically use *rotations* to correct its imbalance. This allows an AVL tree to remain balanced at all times, guaranteeing a $\Theta(\log(n))$ time complexity bound on search, insertion, and removal.

To review, the *height* of a node represents how far it is from the bottom of the tree, measured upward from the leaf nodes. Leaf nodes have a height of 1 (by definition), and the height of all other nodes in the tree can be defined recursively:

- height(empty) = 0
- height(node) = max(height(children)) + 1

Example 18.24 Consider the following four trees, which we will denote as trees A, B, C, and D (from left to right). Which of these trees are valid AVL trees? Recall that an AVL tree is a binary search tree that satisfies the height balance property, where the heights of the children of any node differ by at most one.



Only trees A and B are valid AVL trees. Tree C is not a valid AVL tree because node 7 is imbalanced (the height of 7's left child is 0, and the height of 7's right child is 2, resulting in a difference greater than 1). Tree D is not a valid AVL tree because it is not a binary search tree, since 4 should not be in the right subtree of 5.

Example 18.25 Prove that the height of an AVL tree with n nodes is $O(\log(n))$.

To show that the height of an AVL tree cannot exceed $O(\log(n))$, we have to show that, even if try to create the "worst" AVL tree possible (a tree where the height is as large as possible, using the fewest number of nodes), the maximum height is still bounded by $O(\log(n))$. To start, we will first try to determine the minimum number of nodes needed to build an AVL tree of height h (which we will denote as n_h).

We know that the minimum number of nodes needed to construct an AVL tree of height 0 is 0, so $n_0 = 0$. Similarly, we know that the minimum number of nodes needed to construct an AVL tree of height 1 is 1, so $n_1 = 1$. With these base cases, we can recursively define the minimum number of nodes to construct an AVL tree of height h as

$$n_h = 1 + n_{h-1} + n_{h-2}$$

for $h > 1$. This is because an AVL of height h with the minimum number of nodes consists of a root node (1 node), an AVL subtree of height $h-1$ (which has minimum n_{h-1} nodes), and an AVL subtree of height $h-2$ (which has minimum n_{h-2} nodes). Putting this together, an AVL tree of height h must have minimum $1 + n_{h-1} + n_{h-2}$ nodes. Any more, and the tree would no longer be using the minimum number of nodes; any fewer, and the tree would no longer be balanced.

We know that $n_h > n_{h-1}$ and $n_{h-1} > n_{h-2}$; as $n_h = 1 + n_{h-1} + n_{h-2}$, we can then conclude that $n_h > 2n_{h-2}$. Since the equation is defined recursively, $n_h > 2n_{h-2}$ also implies that $n_{h-2} > 2n_{h-4}$, and $n_{h-4} > 2n_{h-6}$, and so on. This allows us to perform the following substitution:

$$n_h > 2n_{h-2} > 2\left(2n_{h-4}\right) > 2\left(2\left(2n_{h-6}\right)\right) > \ldots$$

If we generalize the above substitution, we can conclude that, for any $0 < i < \frac{h}{2}$,

$$n_h > 2^i n_{(h-2i)}$$

We know that the base case $n_1 = 1$, so we can substitute $\frac{h}{2} - 1$ for i to get

$$n_h > 2^{\left(\frac{h}{2}-1\right)} n_{\left(h-2\left(\frac{h}{2}-1\right)\right)}$$

$$n_h > 2^{\left(\frac{h}{2}-1\right)} n_1$$

$$n_h > 2^{\left(\frac{h}{2}-1\right)}$$

Solving for h, we get

$$h < 2\log(n_h) + 2 = O(\log(n))$$

Thus, the height of an AVL tree is $O(\log(n))$.

Example 18.26 Suppose you have an AVL tree of height 9. What is the minimum number of nodes that can be in this AVL tree? What is the maximum number of nodes that can be in this AVL tree?

To calculate the minimum number of nodes in an AVL tree of height 9, we can use the equation in the previous example:

$$n_h = 1 + n_{h-1} + n_{h-2}$$

We know that an AVL tree of height 0 has 0 nodes, and an AVL tree of height 1 has 1 node, so $n_0 = 0$ and $n_1 = 1$. Our goal is to solve for n_9, the minimum number of nodes in an AVL tree of height 9:

- $n_2 = 1 + n_1 + n_0 = 1 + 1 + 0 = 2$
- $n_3 = 1 + n_2 + n_1 = 1 + 2 + 1 = 4$
- $n_4 = 1 + n_3 + n_2 = 1 + 4 + 2 = 7$
- $n_5 = 1 + n_4 + n_3 = 1 + 7 + 4 = 12$
- $n_6 = 1 + n_5 + n_4 = 1 + 12 + 7 = 20$
- $n_7 = 1 + n_6 + n_5 = 1 + 20 + 12 = 33$
- $n_8 = 1 + n_7 + n_6 = 1 + 33 + 20 = 54$
- $n_9 = 1 + n_8 + n_7 = 1 + 54 + 33 = 88$

Since $n_9 = 88$, an AVL tree of height 9 must have at least 88 nodes.

For an AVL tree of height 9 to have the maximum number of nodes possible, every level must be completely filled. In section 18.2, we showed that a binary tree of height h can have at most $2^h - 1$ nodes. Thus, the maximum number of nodes possible in an AVL tree of height 9 is $2^9 - 1 = 511$.

※ 18.7.2 *Balance Factor*

To balance an AVL tree, we will consider its balance factor. The **balance factor** of any node is equal to the difference between the heights of that node's left and right children. In an AVL tree implementation, we typically store the height of a node in its definition, as shown:

```
1   struct Node {
2     int val;
3     int height;
4     Node* left;
5     Node* right;
6     // returns height of left child
7     int left_height() {
8       return left ? left->height : 0;
9     } // left_height()
10    // returns height of right child
11    int right_height() {
12      return right ? right->height : 0;
13    } // right_height()
14  };
```

This allows us to calculate a node's balance factor using the following equation:

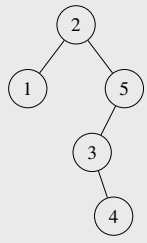
```
balance_factor = left_height() - right_height()
```

For an AVL tree to be balanced, all of its nodes must have a balance factor of -1, 0, or +1.

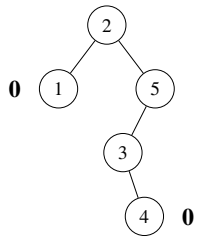
- If `balance_factor(n) == 0`, the left and right subtrees of node `n` have the same height.
- If `balance_factor(n) == +1`, the left subtree of node `n` is taller than the right subtree by one.
- If `balance_factor(n) == -1`, the right subtree of node `n` is taller than the left subtree by one.

If the absolute value of `balance_factor(n)` exceeds 1, that means the node is out of balance, since the difference between the heights of its left and right children is greater than 1. When this happens, the AVL tree will have to self-balance using rotations.

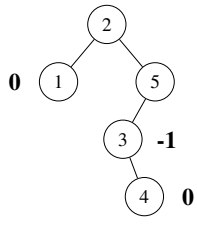
Example 18.27 What is the balance factor of each node in this tree?



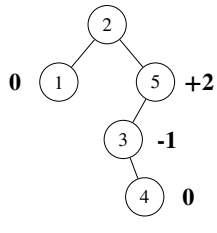
To solve this problem, we will start from the bottom of the tree and move upwards, calculating the balance factor along the way. First, we will look at node 4. Node 4 has no children, so its balance factor is 0. Similarly, node 1 also has no children, so its balance factor is 0 as well.



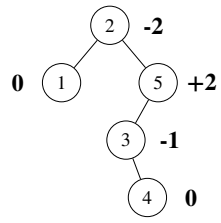
Next, we look at node 3. The height of 3's left child is 0, and the height of 3's right child is 1. Thus, the balance factor of node 3 is 0 - 1 = -1.



Next, we look at node 5. The height of 5's left child is 2, and the height of 5's right child is 0. Thus, the balance factor of node 5 is 2 - 0 = +2.



Lastly, we will look at the root node, 2. The height of 2's left child is 1, and the height of 2's right child is 3. Thus, the balance factor of node 2 is 1 - 3 = -2.



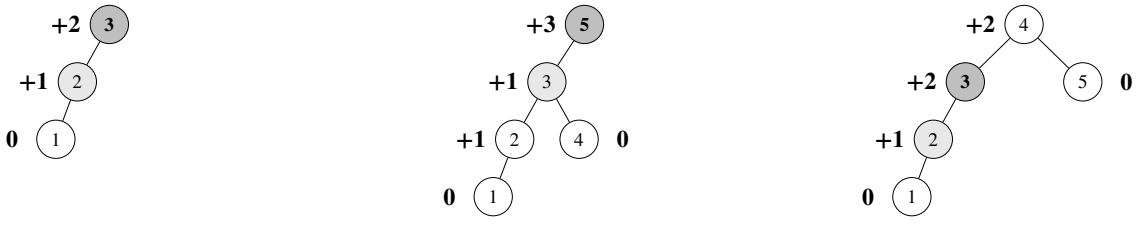
※ 18.7.3 *AVL Tree Rotations*

The tree in the above example is not balanced, since some of its nodes have a balance factor whose absolute value exceeds 1. To balance a tree, we start from the nodes at the bottom of the tree and move upwards toward the root, calculate the balance factor of each node along the way, and perform *rotations* whenever we encounter a node whose balance factor is not -1, 0, or +1. Using this procedure, there are four unique situations that may arise, each of which result in different rotations:

1. An imbalanced node has a balance factor *greater* than +1, and its left child has a balance factor that is either 0 or +1.
   - When this happens, fix the imbalance by conducting a *right rotation* on the subtree rooted at the imbalanced node.
2. An imbalanced node has a balance factor *less* than -1, and its right child has a balance factor that is either 0 or -1.
   - When this happens, fix the imbalance by conducting a *left rotation* on the subtree rooted at the imbalanced node.
3. An imbalanced node has a balance factor *greater* than +1, and its left child has a balance factor of -1.
   - When this happens, fix the imbalance by first conducting a *left rotation* on the subtree rooted at the imbalanced node's left child, and then conducting a *right rotation* on the subtree rooted at the imbalanced node.
4. An imbalanced node has a balance factor *less* than -1, and its right child has a balance factor of +1.
   - When this happens, fix the imbalance by first conducting a *right rotation* on the subtree rooted at the imbalanced node's right child, and then conducting a *left rotation* on the subtree rooted at the imbalanced node.

1. An imbalanced node has a balance factor greater than +1, and its left child has a balance factor that is either 0 or +1.

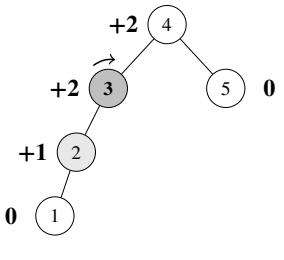
This scenario happens when you encounter an imbalanced node whose left subtree causes the imbalance, and the left side of that subtree is equal in height or taller than the right side of that subtree. Examples of this scenario are shown in the following trees, where the darker shaded node is the first node we encounter with a balance factor greater than +1 (starting from the bottom and moving up), and the lighter shaded node is the left child of the imbalanced node:



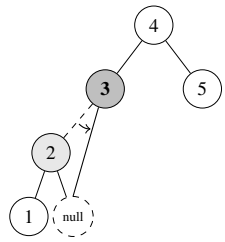
To fix the tree in this situation, we will perform a **right rotation** on the subtree rooted at the node with the balance factor greater than +1 (i.e., the darker shaded node). To conduct a right rotation, complete the following steps:

1. Take the root of the subtree that you want to rotate (the darker shaded node) and set its left child to the right child of its original left child.
2. Take the original left child (the lighter shaded node) and set its right child to its original parent (the darker shaded node).
3. Return the new root of the rotated subtree to the caller that invoked the rotation (which should be the parent of the initial imbalanced node, if there is one). If the node you want to rotate (the darker shaded node) has a parent, reset its parent's child so that it points to the new root of the rotated subtree (which ends up being the lighter shaded node).

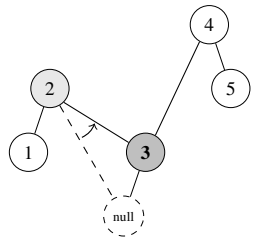
For instance, let's fix the rightmost tree in the previous illustration. If we move from the bottom of the tree upwards, the first imbalanced node we encounter is the node with a value of 3. This node has a balance factor of +2, and its left child has a balance factor of +1, which fits this condition. Thus, to fix this tree, we will conduct a right rotation on the subtree rooted at node 3.



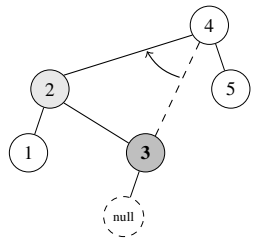
To perform the rotation, we first set 3's left child to the right child of 2. Since 2 has no right child, 3's left child becomes `nullptr`.



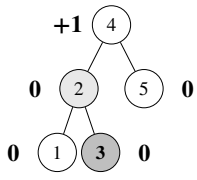
Then, we set 2's right child to 3.



Lastly, we update 4's left child so it points to the new root of the left subtree, node 2.

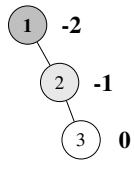


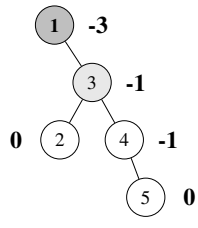
This is now our new tree after the right rotation. Since all of the nodes have a balance factor of -1, 0, or +1, the tree is balanced, and no other rotations need to be completed.

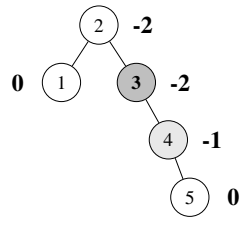


2. An imbalanced node has a balance factor less than -1, and its right child has a balance factor that is either 0 or -1.

This scenario happens when you encounter an imbalanced node whose right subtree causes the imbalance, and the right side of that subtree is equal in height or taller than the left side of that subtree. Examples of this scenario are shown in the following trees, where the darker shaded node is the first node we encounter with a balance factor less than -1 (starting from the bottom and moving up), and the lighter shaded node is the right child of the imbalanced node:



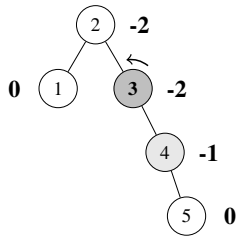




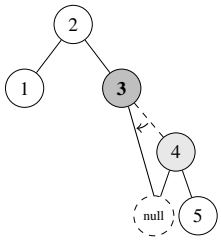
To fix the tree in this situation, we will perform a **left rotation** on the node with the balance factor less than -1 (i.e., the darker shaded node). To conduct a left rotation on a node, complete the following steps:

1. Take the root of the subtree you want to rotate (the darker shaded node) and set its right child to the left child of its original right child.
2. Take the original right child (the lighter shaded node) and set its left child to its original parent (the darker shaded node).
3. Return the new root of the rotated subtree to the caller that invoked the rotation (which should be the parent of the initial imbalanced node, if there is one). If the node you want to rotate (the darker shaded node) has a parent, reset its parent's child so that it points to the new root of the rotated subtree (which ends up being the lighter shaded node).

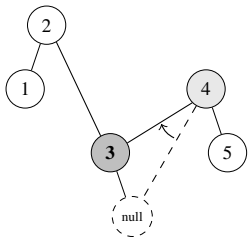
As an example, we will fix the rightmost tree in the previous illustration. If we move from the bottom of the tree upwards, the first imbalanced node we encounter is the node with the value of 3. This node has a balance factor of -2, and its right child has a balance factor of -1, which fits this condition. Thus, to fix this tree, we will conduct a left rotation on the subtree rooted at node 3.



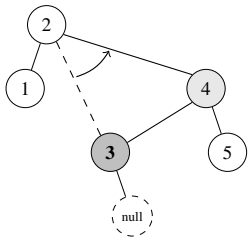
To perform the rotation, we first set 3's right child as the left child of 4. Since 4 has no left child, 3's right child becomes `nullptr`.



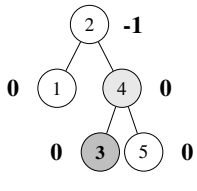
Then, we set 4's left child to 3.



Lastly, we update 2's right child so it points to the new root of the right subtree, node 4.

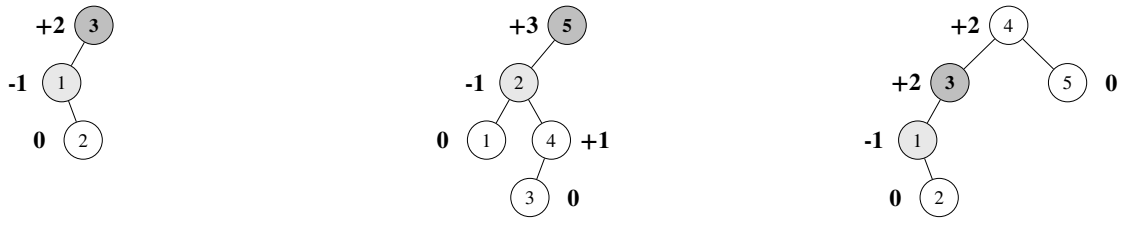


This is now our new tree after the left rotation on node 3. Since all of the nodes have a balance factor of -1, 0, or +1, the tree is balanced, and no other rotations need to be completed.



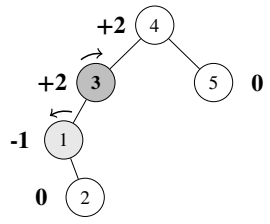
3. An imbalanced node has a balance factor greater than +1, and its left child has a balance factor of -1.

This scenario happens when you encounter an imbalanced node whose left subtree causes the imbalance, and the right side of that subtree is taller than the left side of that subtree. Examples of this scenario are shown in the following trees, where the darker shaded node is the first node we encounter with a balance factor greater than +1 (starting from the bottom and moving up), and the lighter shaded node is the left child of the imbalanced node:

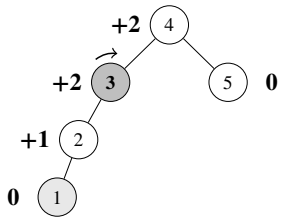


To fix the tree in this situation, we must first perform a **left rotation** on the subtree rooted at the imbalanced node's *left child* (i.e., the lighter shaded node), and then perform a **right rotation** on the subtree rooted at the imbalanced node itself (i.e., the darker shaded node).

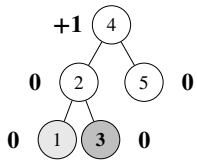
As an example, we will fix the rightmost tree in the previous illustration. If we move from the bottom of the tree upwards, the first imbalanced node we encounter is the node with the value of 3. This node has a balance factor of +2, but its left child has a balance factor of -1 (a sign change). Thus, to fix this tree, we will need to conduct *two* rotations: a left rotation on node 1, followed by a right rotation on node 3.



We start off by conducting a left rotation on node 1. To do so, 1's right child is set to 2's left child (`nullptr`), 2's left child is set to its original parent (node 1), and 3's left child is set to the new root of the subtree (node 2).



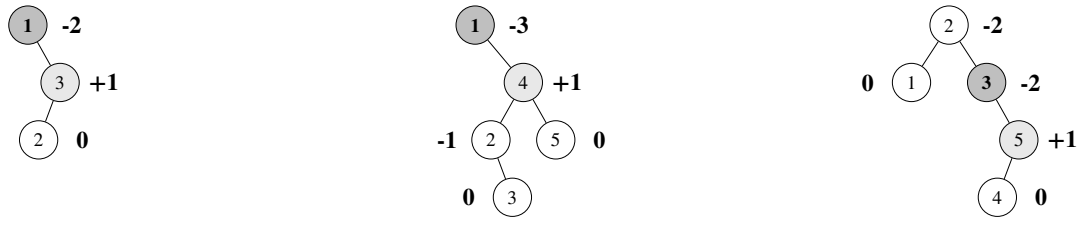
The above tree is the result of performing a left rotation on node 1. After the left rotation is done, we then conduct a right rotation on the original imbalanced node, or node 3. To perform this right rotation, 3's left child is set to 2's right child (`nullptr`), 2's right child is set to its original parent (node 3), and 4's left child is set to the new root of the subtree (node 2).



None of the remaining nodes in the tree are imbalanced, so no other rotations need to be completed.

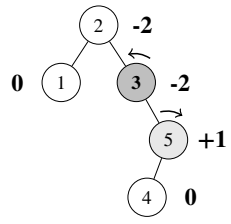
4. An imbalanced node has a balance factor less than -1, and its right child has a balance factor of +1.

This scenario happens when you encounter an imbalanced node whose right subtree causes the imbalance, and the left side of that subtree is taller than the right side of that subtree. Examples of this scenario are shown in the following trees, where the darker shaded node is the first node we encounter with a balance factor less than -1 (starting from the bottom and moving up), and the lighter shaded node is the right child of the imbalanced node:

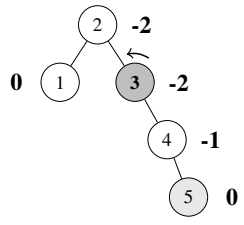


To fix the tree in this situation, we must first perform a **right rotation** on the subtree rooted at the imbalanced node's *right child* (i.e., the lighter shaded node), and then perform a **left rotation** on the subtree rooted at the imbalanced node itself (i.e., the darker shaded node).

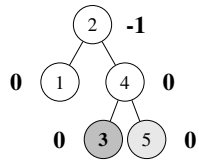
As an example, we will fix the rightmost tree in the previous illustration. If we move from the bottom of the tree upwards, the first imbalanced node we encounter is the node with the value of 3. This node has a balance factor of -2, but its right child has a balance factor of +1 (a sign change). Thus, to fix this tree, we will need to conduct *two* rotations: a right rotation on node 5, followed by a left rotation on node 3.



We start off by conducting a right rotation on node 5. To do so, 5's left child is set to 4's right child (`nullptr`), 4's right child is set to its original parent (node 5), and 3's right child is set to the new root of the subtree (node 4).



Then, we will conduct a left rotation on the original imbalanced node, or node 3. To perform this left rotation, we set 3's right child to 4's left child (`nullptr`), 4's left child to its parent (node 3), and 2's right child to the new root of the subtree (node 4).



None of the remaining nodes in the tree are imbalanced, so no other rotations need to be completed.

Remark: Right rotations are done when a node has a *positive* balance factor > +1, and left rotations are done when a node has a *negative* balance factor < -1. Luckily, this is intuitive, since left is associated with negative, and right is associated with positive. However, you *must* pay attention to the third and fourth conditions above, where there exists a sign change between an imbalanced node and its corresponding child (left child for balance factors > +1, right child for balance factors < -1). In these cases, you will need to rotate the subtree rooted at the imbalanced node's child before you can rotate the subtree rooted at the imbalanced node itself!

The rotation process on an AVL tree can therefore be summarized as follows:

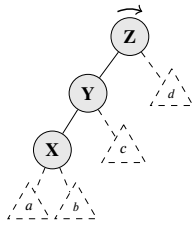
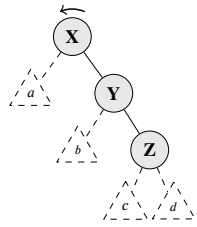
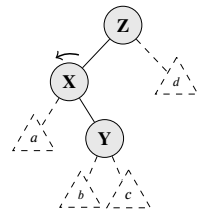
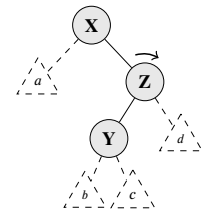
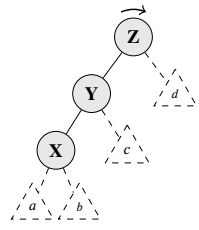
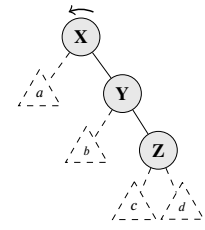
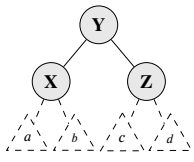
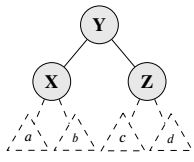
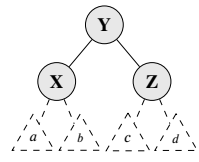
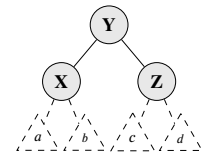
- If the balance factor of a node is greater than +1, check the balance factor of its left child.
  - If the left child has a negative balance factor, perform a left rotation on the subtree rooted at that left child.
  - Perform a right rotation on the subtree rooted at the current parent node (always done regardless of balance factor of left child).
- If the balance factor of a node is less than -1, check the balance factor of its right child.
  - If the right child has a positive balance factor, perform a right rotation on the subtree rooted at the right child.
  - Perform a left rotation on the subtree rooted at the current parent node (always done regardless of balance factor of right child).

The pseudocode is shown below:

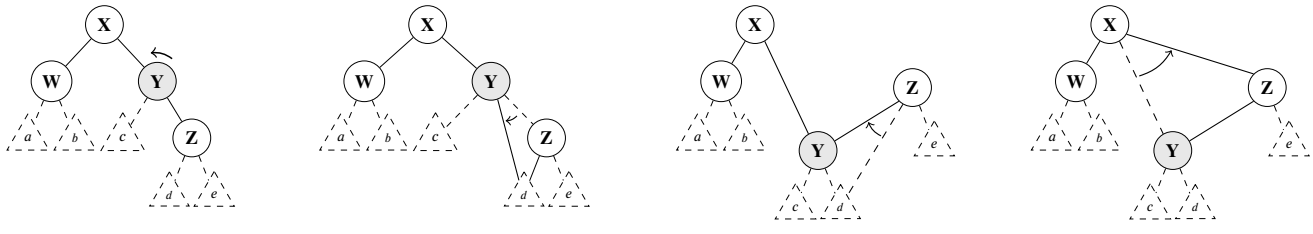
```
1  Algorithm check_and_balance(Node* n):
2    if balance_factor(n) > +1:
3      if balance_factor(n->left) < 0:
4        n->left = rotate_left(n->left)
5      n = rotate_right(n)
6    else if balance_factor(n) < -1:
7      if balance_factor(n->right) > 0:
8        n->right = rotate_right(n->right)
9      n = rotate_left(n)
```

In the pseudocode above, `rotate_left()` and `rotate_right()` return the new root after the rotation, which allows the parent of the rotated node to be easily assigned.

Summary of AVL Tree Rotations

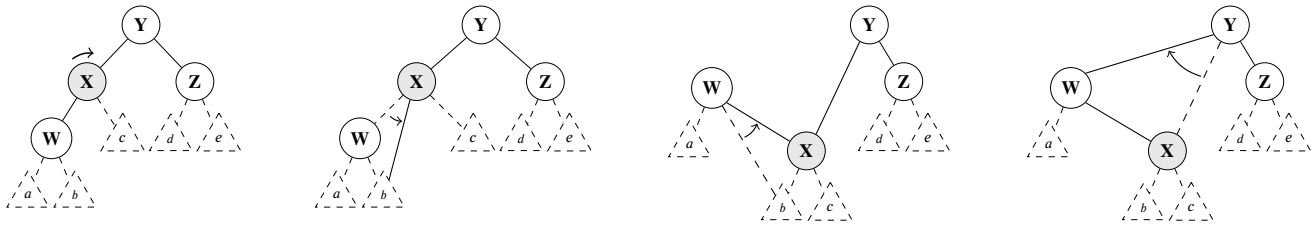
Straight Left-Left Imbalance	Straight Right-Right Imbalance	Zigzag Left-Right Imbalance	Zigzag Right-Left Imbalance
① *rotate right* 	① *rotate left* 	① *rotate left on left child* 	① *rotate right on right child* 
		② *rotate right* 	② *rotate left* 
② 	② 	③ 	③ 
1 Rotation	**1 Rotation**	**2 Rotations**	**2 Rotations**

Left Rotation



rotate left on subtree rooted at node | *set right child of node to left child of right child* | *set left child of original right child to rotated node* | *set right child of original parent to new root of subtree*

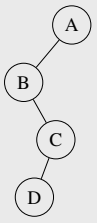
Right Rotation



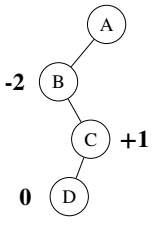
rotate right on subtree rooted at node | *set left child of node to right child of left child* | *set right child of original left child to rotated node* | *set left child of original parent to new root of subtree*

Rotations are very useful in practice because a single rotation always takes constant time (since you are just swapping pointers around), and they ensure a logarithmic tree height. This allows search, insertion, and removal to all take worst-case $\Theta(\log(n))$ time, an improvement over the $\Theta(n)$ worst-case time of a standard non-balancing binary search tree.

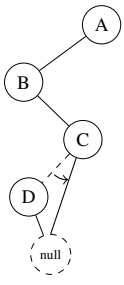
Example 18.28 You are given the following stick tree. Using the rules of rotation above, balance this stick. To do this, start from the bottom node and move upwards toward the root, calculate the balance factor, and rotate whenever necessary.

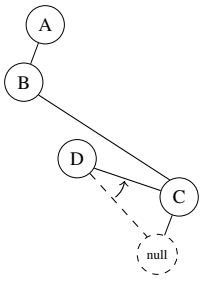


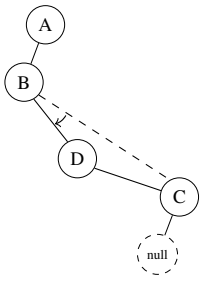
To solve this problem, we will start by calculating the balance factor of all the nodes in this tree, starting from the bottom. Node D has no children, so it has a balance factor of 0. Node C has a left child of height 1 and a right child of height 0, so its balance factor is 1 - 0 = +1. Node B has a left child of height 0 and a right child of height 2, so its balance factor is 0 - 2 = -2. Thus, node B is an imbalanced node.

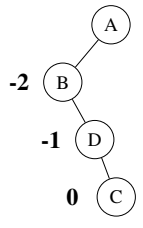


However, before we can rotate node B, we must first check to see if we need to conduct a double rotation! Since node B has a negative imbalance, we look at the balance factor of its right child to see if there is a sign change. In this case, there is: node B has a balance factor of -2, but node C has a balance factor of +1. Thus, we will need to rotate right on node C before we can rotate left on node B! The process of rotating right on node C is shown below, where C's left child is set to D's right child, D's right child is set to C, and B's right child is set to D.

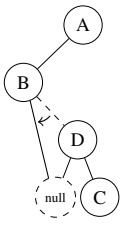


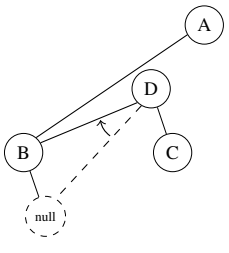


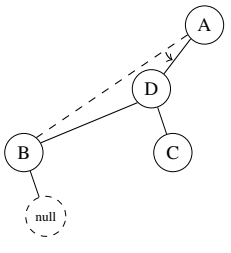


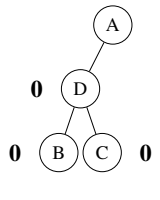


Now, we can rotate left on node B. B's right child is set to D's left child, D's left child is set to B, and A's left child is set to D.

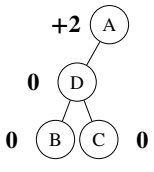




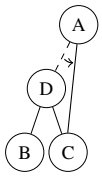


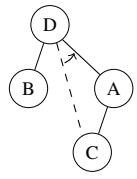


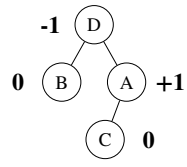
The rotation on B is complete, and none of the nodes in the subtree rooted at B remain imbalanced. We continue up the tree and calculate the balance factor of node A. Since A's left child has a height of 2, and A's right child has a height of 0, the balance factor of A is 2 - 0 = +2.



The left child of node A has a balance factor of 0, so there is no need for a double rotation. Since A's balance factor is positive, we will rotate right on A. This is done by setting A's left child to D's right child (or node C) and D's right child to its original parent (node A). After the rotation, node D becomes the new root of the tree.







The right rotation on node A is now complete, and the tree is balanced.

※ 18.7.4 *Inserting and Removing Elements*

The behaviors of inserting and removing elements from an AVL tree are nearly identical to their corresponding behaviors in a standard binary search tree. However, there is a difference: the insertion or removal of an element in an AVL tree may cause an imbalance that will need to be resolved. As a result, AVL trees require additional checks after each insertion and removal to ensure that the tree remains balanced, and that any imbalanced nodes are corrected using rotations. The rules for insertion and deletion are shown below.

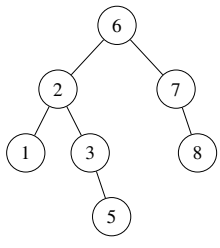
Insertion into an AVL Tree

1. Insert the node to its correct position normally, as if the tree were a standard non-self-balancing binary search tree (without considering imbalances).
2. After the node is inserted, check the balance factors of the ancestors of the new node, starting from the bottom of the tree and moving toward the top. If any of these nodes are imbalanced ($|\text{balance factor}| > 1$), perform the appropriate rotations to balance that node. Once you fix the first imbalanced node you encounter (via either a single or double rotation), you are done, and the AVL tree is guaranteed to be balanced.

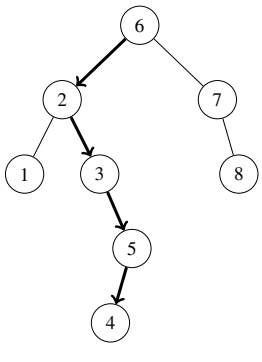
Deletion from an AVL Tree

1. Delete the target node from the tree normally, as if the tree were a standard non-self-balancing binary search tree (using either the inorder successor or inorder predecessor).
2. After the node is removed, travel up the tree from the parent of the removed node. At every imbalanced node encountered, perform the appropriate rotations. Unlike the insertion process, this restructuring step may unbalance multiple ancestors, so you must continue checking and rebalancing up to the root.

An important distinction between insertion and deletion is that insertion only needs to fix the *first* imbalanced node encountered along the path from the newly inserted node to the root (using either a single or double rotation). Once this node is rotated, the AVL tree is guaranteed to be balanced. For example, consider the following AVL tree:



Suppose we want to insert 4 into this AVL tree. We would first insert 4 into the tree as if it were a standard binary search tree, without worrying about imbalances and rotations.



$4 < 6$, so look down left subtree

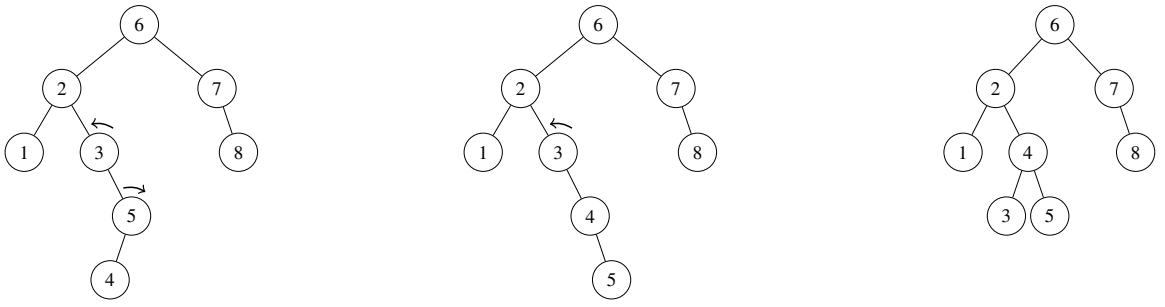
$4 > 2$, so look down right subtree

$4 > 3$, so look down right subtree

$4 < 5$, so look down left subtree

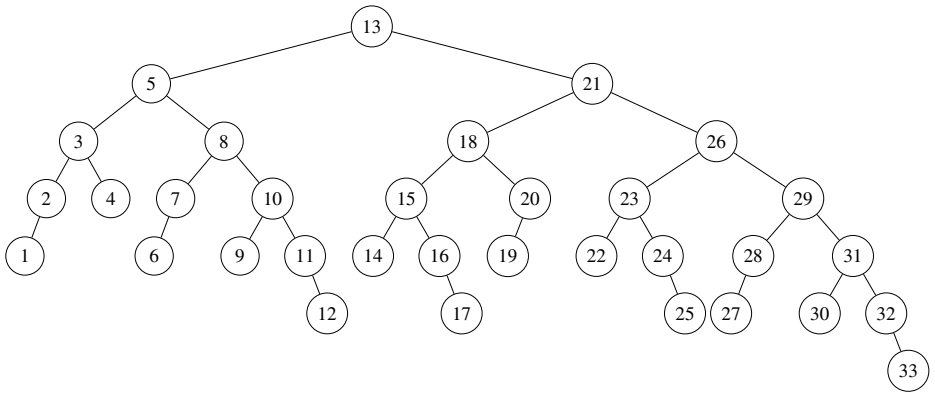
we hit a `nullptr`, so so insert 4 here

Then, look for the first imbalanced node along the path from the newly inserted node to the root, starting from the newly inserted node and moving upwards. This ends up being node 3, which has a balance factor of -2. However, since 5 has a balance factor of +1, we have a sign change (i.e., the right-left zigzag case), so we must rotate right on 5 before we can rotate left on 3.

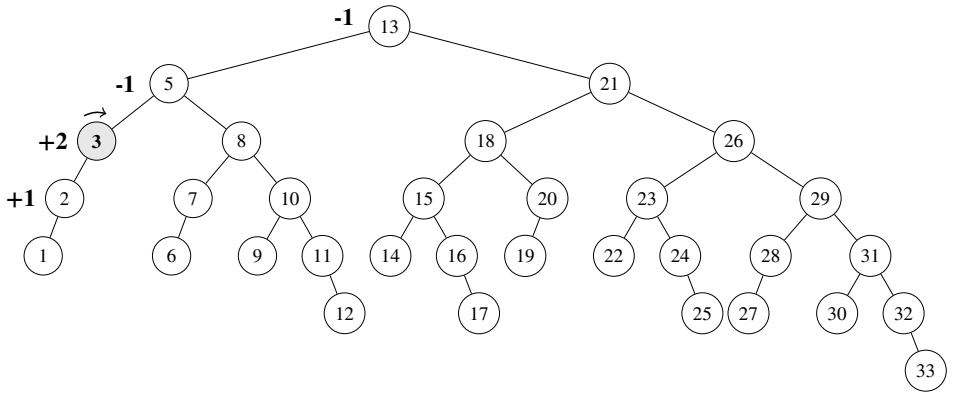


However, once we finish rotating node 3, we do *not* need to check nodes 2 or 6 for a potential imbalance. This is because an AVL tree can always be fixed after an insertion using only a single or a double rotation! Why is this the case? Notice that, after an insertion, the only way an imbalance can happen is if the new node ends up extending a branch way too long (e.g., the insertion of 4 above made the right subtree of 3 too long). However, when you fix the first imbalanced node you encounter using either a single or double rotation, you end up "pulling" the newly inserted node upward, which shortens the height of its branch. Since the new node is pulled upward, any imbalance caused by the insertion is removed, which repairs the entire tree. For example, by fixing node 3 above, we were able to turn a subtree of height 2 into a subtree of height 1, which balances the entire tree.

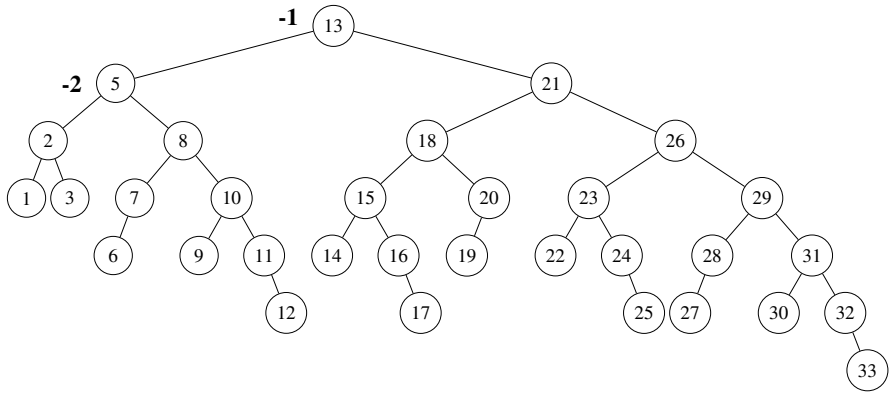
Deletion is a different story. Fixing an AVL tree after an insertion requires at most two rotations, but fixing an AVL tree after a deletion may require up to $\Theta(\log(n))$ rotations (where the $\log(n)$ term comes from the height of the tree). This is because a deleted node can only create an imbalance by making its branch shorter. However, each fix *also* ends up making a subtree of the tree shorter. If a shortened subtree was already shorter than its sibling, a fix will be required at a higher level of the tree, which could result in multiple rotations before the tree returns to a balanced state. For example, consider the following AVL tree:



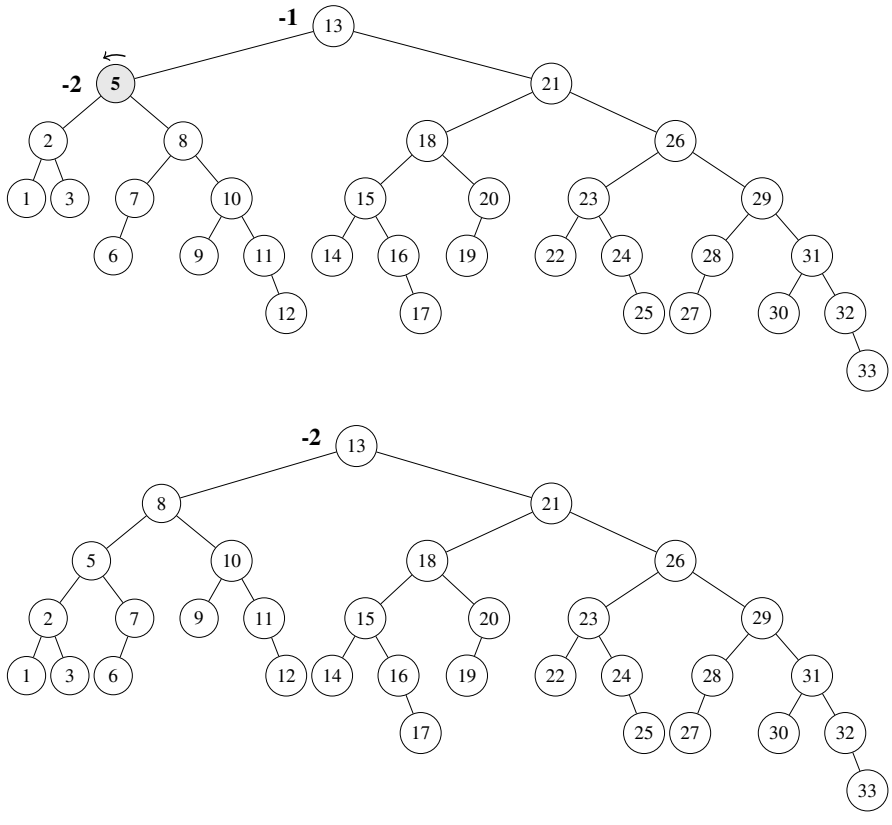
Suppose we remove 4 from this tree. This ends up creating an imbalance at node 3. To fix this, we will rotate right on 3.



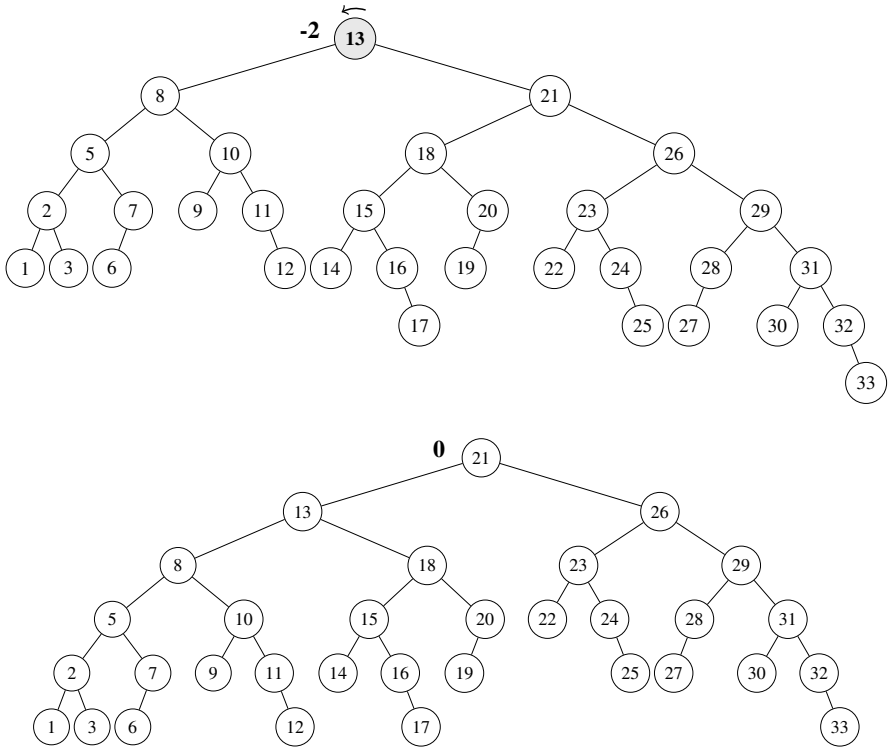
After the right rotation, the tree looks like this:



Notice that the rotation ended up shortening the height of 5's left subtree. This causes an imbalance on node 5, since 5's shorter child was shortened even more! To fix this imbalance, we have to conduct a left rotation on 5.



This rotation ended up shortening the height of 13's left subtree. This ended up creating an imbalance on node 13, since the left subtree of 13 was already shorter than its right subtree. We have to fix this imbalance by conducting a left rotation on 13.



The tree is now fully balanced. The deletion of a single node ended up triggering three rotations!

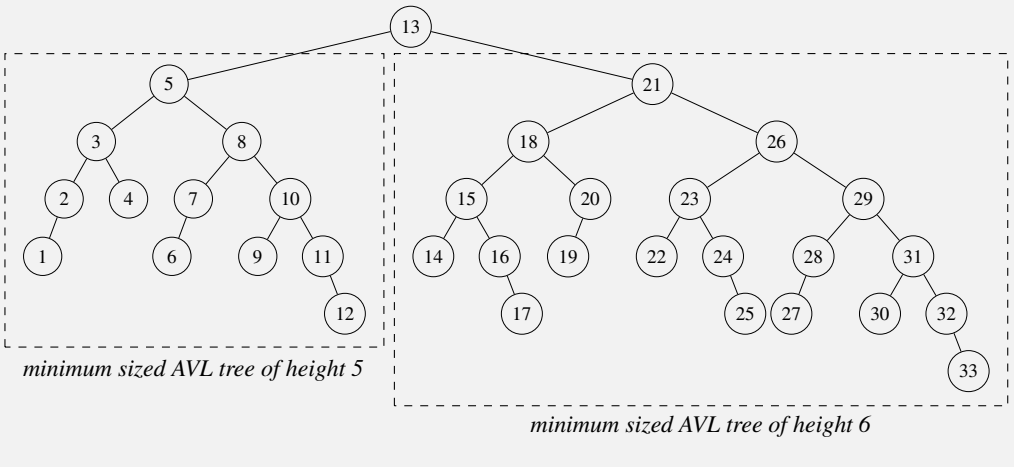
Remark: In the tree above, every imbalance satisfied either the left-left or right-right case, so we were able to correct each imbalance using a single rotation. However, we could have arranged the nodes of the tree in such a way that a *double* rotation is needed to fix every imbalance up the tree. In this case, a total of up to six rotations could have been triggered!

Just how many rotations could result from a single deletion? This depends on the size of the tree. The smallest tree that can trigger up to two rotations (either a single or double rotation) after a single deletion has size 4 — this can be easily proven by drawing some trees out. The smallest tree that can trigger up to four rotations after a single deletion has size 12. The smallest tree that can trigger up to six rotations after a single deletion has size 33 (like the tree above). The smallest tree that can trigger up to eight rotations after a single deletion has size 88.

4, 12, 33, 88, … if you paid close attention to a previous example, this sequence should be somewhat familiar to you. Recall the example where we calculated the minimum number of nodes in an AVL tree of height 9:

- $n_2 = 1 + n_1 + n_0 = 1 + 1 + 0 = 2$
- $n_3 = 1 + n_2 + n_1 = 1 + 2 + 1 = \mathbf{4}$
- $n_4 = 1 + n_3 + n_2 = 1 + 4 + 2 = 7$
- $n_5 = 1 + n_4 + n_3 = 1 + 7 + 4 = \mathbf{12}$
- $n_6 = 1 + n_5 + n_4 = 1 + 12 + 7 = 20$
- $n_7 = 1 + n_6 + n_5 = 1 + 20 + 12 = \mathbf{33}$
- $n_8 = 1 + n_7 + n_6 = 1 + 33 + 20 = 54$
- $n_9 = 1 + n_8 + n_7 = 1 + 54 + 33 = \mathbf{88}$

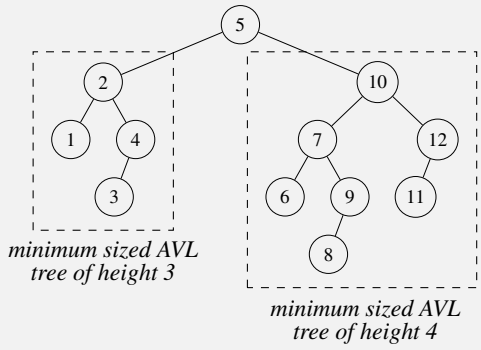
The sequence 4, 12, 33, 88, … represent the minimum number of nodes required for AVL trees of odd heights. Is this some crazy coincidence, or is this all related? Unsurprisingly, it turns out that these two relationships are in fact related. If you look at the previous 33-node tree example, the children of the root are themselves minimum sized trees of heights 5 and 6:



minimum sized AVL tree of height 5

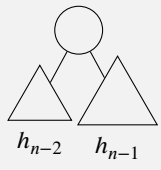
minimum sized AVL tree of height 6

A tree of size 12 is the smallest that can trigger up to four rotations, which occurs if the children of the root are minimum sized AVL trees of heights 3 and 4. For instance, deleting node 1 from the following tree of size 12 would trigger four rotations (RR-4, RL-2, RR-10, RL-5):

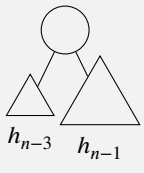


Similarly, an AVL tree of size 88 (which is too big to draw on the page) can trigger up to eight rotations if the children of the root are minimum sized AVL trees of heights 7 and 8.

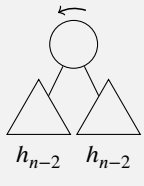
Why is this true? To trigger more than one rotation upon a deletion, we have to delete a node from the *shorter* of a tree's two children. Consider an arbitrary AVL tree of height $n > 3$, where one child has height $n-1$ and the other has height $n-2$:



If we delete a node from the smaller subtree and create an imbalance, rotations will be needed to fix that subtree. Since a rotation can only shorten the height of a subtree, a rebalance of the left subtree above would shorten its height to $n-3$:



The difference in heights between the two subtrees is now greater than 1, so a rotation on the root is needed to balance the entire tree. This can take up to two rotations (in this example, a double rotation is needed if the left child of the right subtree has a greater height than the right child of the right subtree, which results in the zigzag right-left case).



The maximum number of rotations needed to fix a tree after a deletion is thus the maximum number of rotations needed to fix the smaller subtree, plus two rotations for the root. Since the smallest possible height a child in an AVL tree of height n can have is $n-2$, we can define the maximum number of rotations needed after deleting a node from an AVL tree of height n as:

$$\text{max\_rotation}(h_n) = \text{max\_rotation}(h_{n-2}) + 2$$

We know that an AVL of height 3 can only support either a single or double rotation after a deletion, so $\text{max\_rotation}(h_3) = 2$. We also know that an AVL tree of height less than 3 cannot produce any rotations after a deletion, since there is no way for a node to end up with a balance factor greater than 1 or less than -1. Thus, $\text{max\_rotation}(h_2) = 0$. We can use this information to recursively calculate the maximum number of rotations that are possible after a deletion for an AVL tree of any height:

- $\text{max\_rotation}(h_4) = \text{max\_rotation}(h_2) + 2 = 0 + 2 = 2$
- $\text{max\_rotation}(h_5) = \text{max\_rotation}(h_3) + 2 = 2 + 2 = 4$
- $\text{max\_rotation}(h_6) = \text{max\_rotation}(h_4) + 2 = 2 + 2 = 4$
- $\text{max\_rotation}(h_7) = \text{max\_rotation}(h_5) + 2 = 4 + 2 = 6$
- $\text{max\_rotation}(h_8) = \text{max\_rotation}(h_6) + 2 = 4 + 2 = 6$
- $\text{max\_rotation}(h_9) = \text{max\_rotation}(h_7) + 2 = 6 + 2 = 8$
- …

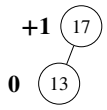
This is precisely why the maximum number of rotations possible after a single deletion mirrors the sequence of odd AVL heights. Every time you reach a new odd height, you can potentially trigger another single or double rotation after a deletion!

Example 18.29 How many rotations are needed to perform the following operations on an initially empty AVL tree? Note that a double rotation operation counts as two rotations.

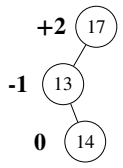
Insert 17, Insert 13, Insert 14, Insert 15, Insert 16, Delete 13

First, we will insert 17 into the AVL tree. This can be done trivially:

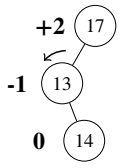
Next, we insert 13. Since 13 is smaller than 17, we add it as the left child of 17.

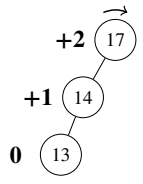


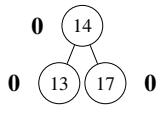
Next, we insert 14. As 14 is smaller than 17, we look at the left child of 17, or 13. Since 14 is larger than 13, we add 14 as the right child of 13.



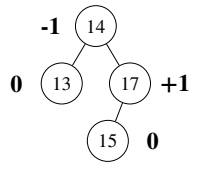
The tree is now imbalanced, so we will need to perform a rotation. Since 17 has a balance factor of +2 and 13 has a balance factor of -1, we must first conduct a left rotation on 13, followed by a right rotation on 17. This results in *two* rotations, as shown.



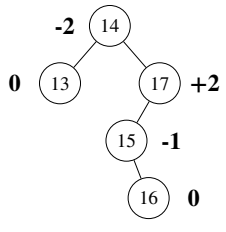




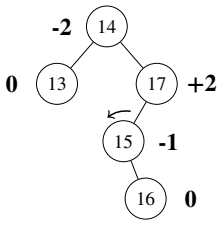
Next, we insert 15. Walking down the tree, 15 ends up getting placed as the left child of 17. This doesn't unbalance the tree in any way, so no rotations are needed.

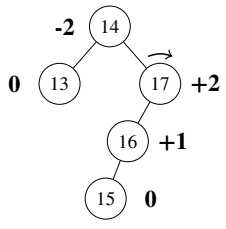


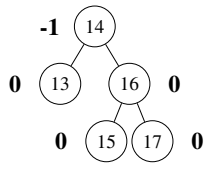
Next, we insert 16, which ends up getting placed as the right child of 15.



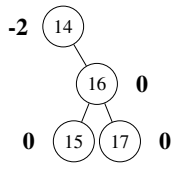
The tree is now imbalanced. To fix the tree, we will have to conduct a left rotation on 15, followed by a right rotation on 17. This requires two rotations. In total, we have completed four rotations so far.



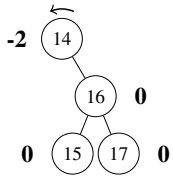


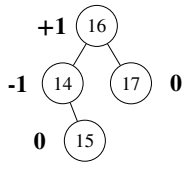


Next, we will delete 13. Since 13 has no children, we can just remove the node without having to deal with the inorder predecessor or successor. However, the tree ends up being imbalanced after the removal of 13.



To fix this, we will perform a single left rotation on 14.





The tree is now balanced, and all operations are complete. A total of 5 rotations were needed to complete the given insertions.

Summary of AVL Tree Time Complexities

Operation	Best-Case Time	Average-Case Time	Worst-Case Time
Finding a value	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Inserting a value	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Deleting a value	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

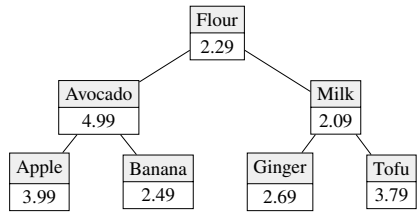
18.8 The STL Map Container

The `std::map<>` in the C++ `<map>` library is an associative container that stores key-value pairs. Each item in a `std::map<>` is represented as a `std::pair<>`, where the key is the `.first` value and the value is the `.second` value. Keys in a `std::map<>` must be unique. Much like an `std::unordered_map<>`, a `std::map<>` supports fast lookup of values associated with each key. However, unlike an `std::unordered_map<>`, the keys in a `std::map<>` are *ordered* based on a sorting criterion. Because maps need to support efficient search, insertion, and removal while also keeping track of the ordering of its keys, they are *not* implemented using hashing (which does not preserve order). Instead, C++ maps are internally implemented using *self-balancing binary search trees*.[2]

For example, let's consider the table of food prices that we introduced in the previous chapter (reproduced below). If we were to insert these food items into a `std::map<>` instead of a `std::unordered_map<>`, we could potentially end up with something like this under the hood (note that this is not exact, as it is just one example of what the map could look like):

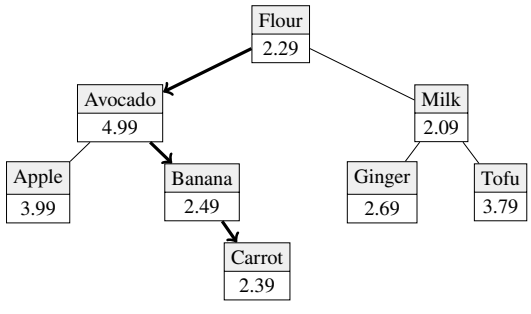
Food Item	Price
Apple	$3.99
Avocado	$4.99
Banana	$2.49
Flour	$2.29
Ginger	$2.69
Milk	$2.09
Tofu	$3.79

`std::map<>` under the hood



Search, insertion, and deletion behave as expected for a self-balancing binary search tree. For instance, if we wanted to add the key-value pair `{Carrot, $2.39}` into this map, we would find and insert `"Carrot"` into the correct sorted position, as shown below:

`std::map<>` under the hood



[2]Maps in the standard library are typically implemented using *red-black trees* instead of AVL trees. Both implementations accomplish the same goal of balancing a tree after a modification, and both have the same worst-case $\Theta(\log(n))$ time complexities for search, insertion, and deletion.

Since maps use a self-balancing binary search tree structure to keep track of the ordering of its keys, the complexities of search, insertion, and deletion are $\Theta(\log(n))$ instead of $\Theta(1)$. This is the tradeoff between a map and an unordered map; if you want to keep track of the ordering of keys, the average-case complexities of these three operations are logarithmic rather than constant.

Operation	`std::unordered_map<>`	`std::map<>`
Search for Key	$\Theta(1)$ average-case time	$\Theta(\log(n))$ average-case time
	$\Theta(n)$ worst-case time	$\Theta(\log(n))$ worst-case time
Insert Key	$\Theta(1)$ average-case time	$\Theta(\log(n))$ average-case time
	$\Theta(n)$ worst-case time	$\Theta(\log(n))$ worst-case time
Delete Key	$\Theta(1)$ average-case time	$\Theta(\log(n))$ average-case time
	$\Theta(n)$ worst-case time	$\Theta(\log(n))$ worst-case time

To instantiate a `std::map<>`, pass in the types of the key and the mapped value (similar to the syntax of an unordered map):

```
std::map<KEY_TYPE, VALUE_TYPE> map_name;
```

For example, to create a map that maps a food name (string) into a price (double), we could declare a map as follows:

```
std::map<std::string, double> food_prices;
```

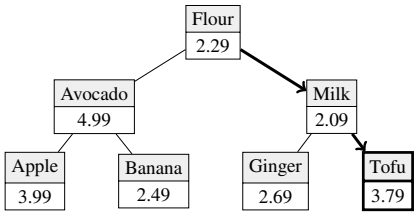
We can then go through and insert our data into the map:

```
food_prices.insert({"Apple", 3.99});
food_prices.insert({"Avocado", 4.99});
food_prices.insert({"Banana", 2.49});
food_prices.insert({"Flour", 2.29});
food_prices.insert({"Ginger", 2.69});
food_prices.insert({"Milk", 2.09});
food_prices.insert({"Tofu", 3.79});
```

Similar to the unordered map, `operator[]` can be used to retrieve the value associated with a key. For instance, if you wanted to retrieve the price of `"Tofu"`, you can just call `food_prices["Tofu"]` to get its price, 3.79.

```
double price_tofu = food_prices["Tofu"];
std::cout << price_tofu << '\n';  // prints 3.79
```

Behind the scenes, calling `operator[]` on a key results in a lookup in the map's underlying binary search tree.



However, if `operator[]` is called on a key that does not exist, it will be automatically inserted into the container (similar to an unordered map). Thus, you have to be careful not to use `operator[]` on a key before you know that it actually exists. To check if a key actually exists in a map, you can use the `.find()` member function:

```
template <typename K, typename V>
iterator std::map<K, V>::find(const K& key);
```

Checks if `key` exists in the container. If it exists, the function returns an iterator to the element with that key. If it does not exist, the function returns an iterator that points one past the end (i.e., the `end` iterator).

Much of the functionality supported by `std::map<>` is also supported by `std::unordered_map<>`. A list of these operations is shown below (these behave similarly to their corresponding operations in an unordered map):

```
template <typename K, typename V>
std::pair<iterator, bool> std::map<K, V>::insert(const std::pair<K, V>& p);
```

Attempts to insert a key-value pair into the container. Since keys in an `std::map<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
template <typename K, typename V, typename... Args>
std::pair<iterator, bool> std::map<K, V>::emplace(Args&&... args);
```

Attempts to insert a key-value pair into the container, constructed in-place using `args`. Since keys in an `std::map<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
template <typename K, typename V, typename... Args>
std::pair<iterator, bool> std::map<K, V>::try_emplace(K&& k, Args&&... args);
std::pair<iterator, bool> std::map<K, V>::try_emplace(const K& k, Args&&... args);
```

Inserts a new element into the container with key `k` and value constructed using `args`, but only if `k` does not exist as a key in the table already. Unlike `insert()` and `emplace()`, `try_emplace()` does not move from rvalue arguments if an insertion does not happen, which makes them useful when working with maps whose values are move-only types such as `std::unique_ptr<>` (covered in chapter 27). Also, unlike `emplace()`, `try_emplace()` treats the arguments of the key and mapped value separately, so you do not need the constructor arguments to directly construct a `std::pair<>` (i.e., `try_emplace(key, value_arg1, value_arg2, ...)` works instead of `emplace(key, value)` which passes in an instance of the value directly, while `value_arg...` can be used to construct `value` in-place). The return value is the same as that of `emplace()`.

```
template <typename K, typename V>
bool std::map<K, V>::empty();
```

Returns a `bool` indicating whether the map is empty.

```
template <typename K, typename V>
size_t std::map<K, V>::size();
```

Returns the number of elements in the container.

```
template <typename K, typename V>
void std::map<K, V>::clear();
```

Erases all elements in the container, dropping size to 0.

```
template <typename K, typename V>
iterator std::map<K, V>::erase(const iterator pos);
```

Erases the element pointed to by `pos` and returns an iterator to the element following the one that was removed.

```
template <typename K, typename V>
iterator std::map<K, V>::erase(const iterator first, const iterator last);
```

Erases all elements in the iterator range `[first, last)` and returns an iterator to the element following the last element removed.

```
template <typename K, typename V>
size_t std::map<K, V>::erase(const K& key);
```

Erases the element with the key equivalent to `key` and returns the number of elements removed (which is always 1 since keys are unique).

```
template <typename K, typename V>
size_t std::map<K, V>::count(const K& key);
```

Returns the number of elements whose key compares equal to `key`. For an `std::map<>`, this function returns 0 if the key does not exist, and 1 if it does.

Because maps store their elements in sorted order, they also support additional functionalities that rely on this invariant. For instance, iterators in an `std::unordered_map<>` are typically only useful for iteration purposes, since there is no guarantee on the order of keys in the container. However, in a `std::map<>`, we can easily access its keys in sorted order. As a result, iterators in a `std::map<>` can be used to identify and visit keys based on their sorted position.

Because of this, one big difference is that maps support reverse iterators, while unordered maps do not. Thus, maps (and other *ordered* associative containers) support bidirectional iterators.

```
template <typename K, typename V>
iterator std::map<K, V>::begin();
```

Returns an iterator pointing to the first element in the `std::map<>` container (`.cbegin()` returns a `const` version of this iterator). By default, this is the position of the smallest key in the map.

```
template <typename K, typename V>
iterator std::map<K, V>::end();
```

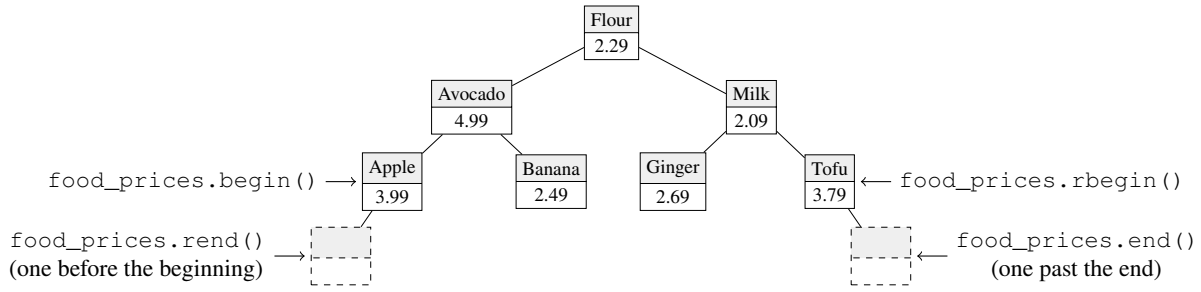
Returns an iterator pointing to one past the last element in the `std::map<>` container (`.cend()` returns a `const` version of this iterator). By default, this is the position one past the largest key in the map.

```
template <typename K, typename V>
iterator std::map<K, V>::rbegin();
```

Returns an iterator pointing to the last element in the `std::map<>` container (`.crbegin()` returns a `const` version of this iterator). By default, this is the position of the largest key in the map.

```
template <typename K, typename V>
iterator std::map<K, V>::rend();
```

Returns an iterator pointing to one before the first element in the `std::map<>` container (`.crend()` returns a `const` version of this iterator). By default, this is the position one before the smallest key in the map.



Incrementing an iterator in a map moves it to the next sorted position. For example, if an iterator were pointing to the key-value pair `{"Banana", 2.49}`, incrementing it would move it to the next key-value pair in alphabetical order, or `{"Flour", 2.29}`. Thus, iterating through a map from the begin iterator to the end iterator is akin to performing an inorder traversal of the tree.

Maps also support `.lower_bound()`, `.upper_bound()`, and `.equal_range()`, which behave similarly to their counterparts in the algorithm library. These operations can be used to obtain the relative position of a key in the map.

`template <typename K, typename V>` `iterator std::map<K, V>::lower_bound(const K& key);`
Returns an iterator pointing to the first key-value pair that is *not* less than `key`. If no such element is found, the end iterator is returned.
`template <typename K, typename V>` `iterator std::map<K, V>::upper_bound(const K& key);`
Returns an iterator pointing to the first key-value pair that is greater than `key`. If no such element is found, the end iterator is returned.
`template <typename K, typename V>` `std::pair<iterator, iterator> std::map<K, V>::equal_range(const K& key);`
Returns a pair of iterators, where the first iterator points to the first element that is not less than `key`, and the second iterator points to the first element greater than `key`. An end iterator is returned if no such elements are found.

For instance, consider the following code, using the food prices above:

```
auto p = food_prices.lower_bound("Mango");  // returns iter to key-value pair
std::cout << p->first << '\n';              // prints "Milk"
std::cout << p->second << '\n';             // prints 2.09
```

Here, `.lower_bound()` returns an iterator to the first key-value pair in the map whose key is not less than `"Mango"`. This ends up being `"Milk"`, which is why `"Milk"` and `"2.09"` are printed in the example code above.

Example 18.30 Design a time-based key value data structure that can be used to store multiple values for the same key based on timestamp. This data structure should be able to retrieve the value of any key at any given timestamp. An outline of this structure is provided below:

```
class TimedMap {
private:
  // TODO: Add any data structures here!
public:
  TimedMap() {}

  // sets the key to a value at a given timestamp
  void set(const std::string& key, const std::string& value, int32_t timestamp) {
    // TODO: Implement code here
  } // set()

  // gets the value of the key at the specified timestamp
  std::string get(const std::string& key, int32_t timestamp) {
    // TODO: Implement code here
  } // get()
};
```

This class supports two members:

- `set()`: Sets the value of a key at a given timestamp.
- `get()`: Gets the value associated with the key at the given timestamp. If there are no values, returns the empty string `""`.

Example: Given the following sequence of key-value sets:

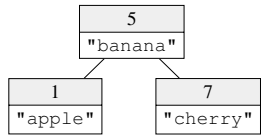
- `set("fruit", "apple", 1)`
- `set("fruit", "banana", 5)`
- `set("fruit", "cherry", 7)`
- `set("vegetable", "carrot", 4)`

Then the following is true:

- `get("fruit", 1)` returns `"apple"`. This is because the value of `"fruit"` was set to `"apple"` at timestamp 1.
- `get("fruit", 6)` returns `"banana"`. This is because the value of `"fruit"` was set to `"banana"` at timestamp 5, but was not changed to `"cherry"` until timestamp 7, so the value of `"fruit"` at timestamp 6 remains `"banana"`.
- `get("vegetable", 3)` returns `""`, since `"vegetable"` did not exist as a key in the map yet at timestamp 3.

Since we may have to retrieve the value of a key at any given timestamp, we will have to store all key-value updates in a separate data structure so that they can be referenced later. This data structure must support efficient search based on a provided timestamp value.

As covered in previous chapters, search is more efficient if the key to search on is stored in *sorted* order, as that allows you to perform a $\Theta(\log(n))$ binary search instead of a $\Theta(n)$ linear search to find the value you want. Thus, it would be prudent to consider a container sorted by timestamp for storing updates. Furthermore, since each timestamp is associated with a value, we can map each timestamp to its corresponding value in a sorted lookup container like a `std::map<>`. For example, we could use the following map to represent the change history of `"fruit"` (where key is the timestamp, and value is the fruit value):



Here, if we wanted to query the value of `"fruit"` at timestamp 6, we could look for the largest key in the map that is less than or equal to 6. To do so, we can take advantage of the `std::map<>`'s `.upper_bound()` member function, which returns an iterator pointing to the first key-value pair that is greater than the given timestamp (in this case, 6). If `.upper_bound()` returns the `.begin()` iterator, we know that all the timestamps in the map must be larger than the key, so no value must exist for the key at that timestamp (and we return an empty string). Otherwise, since `.upper_bound()` returns an iterator to the first timestamp *greater* than the given timestamp, we would return the key-value pair that directly precedes it.

Additionally, our `TimedMap` will need to store the history of multiple keys (e.g., `"fruit"`, `"vegetable"`). Since each key will need to store its own history, we will need a separate lookup container to map each key to its corresponding history map. Since there is no need to access the keys in sorted order, this can be done using an `std::unordered_map<>`. An implementation of this solution is shown below:

```
1   class TimedMap {
2   private:
3     std::unordered_map<std::string, std::map<int32_t, std::string>> time_map;
4   public:
5     TimedMap() {}
6
7     // sets the key to a value at a given timestamp
8     void set(const std::string& key, const std::string& value, int32_t timestamp) {
9       time_map[key].emplace(timestamp, value);
10    } // set()
11
12    // gets the value of the key at the specified timestamp
13    std::string get(const std::string& key, int32_t timestamp) {
14      const auto& map_for_key = time_map[key];
15      auto it = map_for_key.upper_bound(timestamp);
16      return it == map_for_key.begin() ? "" : std::prev(it)->second;
17    } // get()
18  };
```

What are the time complexities of `set()` and `get()`? To set a value, we first look up a key in an `std::unordered_map<>`, which takes constant time. Then, we insert the timestamp into a `std::map<>` associated with the key; insertion into a map takes $\Theta(\log(n))$ time for n timestamps. Thus, the total time complexity of a single call to `set()` is $\Theta(\log(n))$.

To get a value, we have to find the closest timestamp that is smaller; this requires a binary search, which takes $\Theta(\log(n))$ time. Since this is the most expensive operation, the overall time complexity of a single call to `get()` is also $\Theta(\log(n))$.

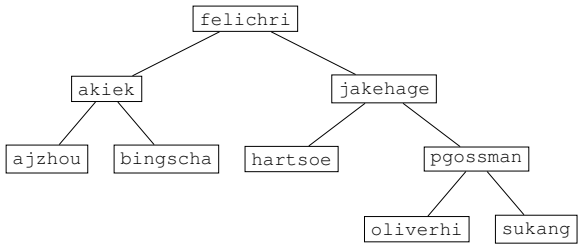
Remark: There is an additional optimization we can make if we can guarantee that the timestamps are set in *increasing* order. If this is the case, then we no longer need to use a `std::map<>` to maintain the timestamps in sorted order; we can just append them to the back of a `std::vector<>` and they will be naturally sorted because of the order in which timestamps are set. With this implementation, the time complexity of `get()` would still be $\Theta(\log(n))$ because we cannot avoid the binary search, but the time complexity of `set()` would become $\Theta(1)$ since appending to a vector takes constant time.

18.9 The STL Set Container

In chapter 13, we introduced the implementation of an ordered set using a sorted vector, which allows elements to be searched in $\Theta(\log(n))$ time. However, the sorted vector implementation is not as ideal for insertions and removals, since its contents may need to be shifted to ensure the contiguity of elements. If we want to remove this inefficiency while still maintaining $\Theta(\log(n))$ time complexities for search, insertion, and deletion, we can use a self-balancing binary search tree instead.

Much like the `std::map<>`, the STL `std::set<>` container, found in the `<set>` library, is implemented using a self-balancing binary search tree. The `std::set<>` is similar to the `std::unordered_set<>`, except that the elements in a `std::set<>` are ordered based on a sorting criterion. Thus, sets are a good container type to consider if you want to keep track of a collection of keys, where the ordering of keys is important to remember.

```
1  std::set<std::string> eecs281staff;
2  eecs281staff.insert("ajzhou");
3  eecs281staff.insert("akiek");
4  eecs281staff.insert("bingscha");
5  eecs281staff.insert("felichri");
6  eecs281staff.insert("hartsoe");
7  eecs281staff.insert("jakehage");
8  eecs281staff.insert("oliverhi");
9  eecs281staff.insert("pgossman");
10 eecs281staff.insert("sukang");
11 ...
```



Because sets store their elements in sorted order, the time complexities of search, insertion, and deletion are $\Theta(\log(n))$. Note that the values in a set are distinct. A few common `std::set<>` operations are summarized below:

```
template <typename K>
std::pair<iterator, bool> std::set<K>::insert(const K& key);
```
Attempts to insert a key into the container. Since keys in an `std::set<>` must be unique, an insertion is done only if the key does not already exist in the container. Returns a pair consisting of an iterator to the inserted element (or the element that prevented insertion) and a `bool` for whether the insertion took place.

```
template <typename K, typename InputIterator>
void std::set<K>::insert(InputIterator first, InputIterator last);
```
Attempts to insert all elements from the range `[first, last)` into the `std::set<>`.

```
template <typename K>
void std::set<K>::insert(std::initializer_list<K> ilist);
```
Attempts to insert all the elements from the initializer list `ilist` into the `std::set<>` (for example, the line `my_set.insert({1, 3, 5, 7})` inserts the elements 1, 3, 5, and 7).

```
template <typename K>
bool std::set<K>::empty();
```
Returns a `bool` indicating whether the set is empty.

```
template <typename K>
size_t std::set<K>::size();
```
Returns the number of elements in the set.

```
template <typename K>
iterator std::set<K>::begin();
```
Returns an iterator pointing to the first element in the `std::set<>` container (`.cbegin()` returns a `const` version of this iterator iterator). By default, this is the position of the smallest value in the set.

```
template <typename K>
iterator std::set<K>::end();
```
Returns an iterator pointing to one past the last element in the `std::set<>` container (`.cend()` returns a `const` version of this iterator). By default, this is the position one past the largest value in the set.

```
template <typename K>
iterator std::set<K>::rbegin();
```
Returns an iterator pointing to the last element in the `std::set<>` container (`.crbegin()` returns a `const` version of this iterator). By default, this is the position of the largest value in the set.

```
template <typename K>
iterator std::set<K>::rend();
```
Returns an iterator pointing to one before the first element in the `std::set<>` container (`.crend()` returns a `const` version of this iterator). By default, this is the position one before the smallest value in the set.

```
template <typename K>
void std::set<K>::clear();
```
Erases all elements in the container, dropping size to 0.

```
template <typename K>
iterator std::set<K>::erase(const iterator pos);
```
Erases the element pointed to by `pos` and returns an iterator to the element following the one that was removed.

```
template <typename K>
iterator std::set<K>::erase(const iterator first, const iterator last);
```
Erases all elements in the iterator range `[first, last)` and returns an iterator to the element following the last element removed.

```
template <typename K>
size_t std::set<K>::erase(const K& key);
```
Erases the element with the key equivalent to `key` and returns the number of elements removed (which is always 1 since keys are unique).

```
template <typename K>
iterator std::set<K>::find(const K& key);
```
Checks if `key` exists in the container. If it exists, the function returns an iterator to the element with that key. If it does not exist, the function returns an iterator that points one past the end (i.e., the `end` iterator).

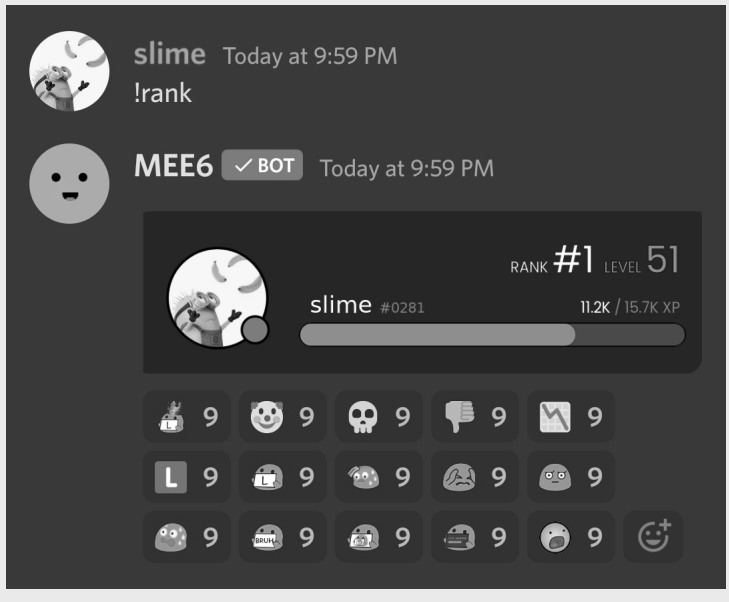
```
template <typename K>
size_t std::set<K>::count(const K& key);
```
Returns the number of elements whose key compares equal to `key`. For a `std::set<>`, this function returns 0 if the key does not exist, and 1 if it does.

```
template <typename K>
iterator std::set<K>::lower_bound(const K& key);
```
Returns an iterator pointing to the first key in the set that is *not* less than `key`. If no such element is found, the end iterator is returned.

```
template <typename K>
iterator std::set<K>::upper_bound(const K& key);
```
Returns an iterator pointing to the first key in the set that is greater than `key`. If no such element is found, the end iterator is returned.

```
template <typename K>
std::pair<iterator, iterator> std::set<K>::equal_range(const K& key);
```
Returns a pair of iterators, where the first iterator points to the first element that is not less than `key`, and the second iterator points to the first element greater than `key`. An end iterator is returned if no such elements are found.

Example 18.31 EECS 281 has a Discord server that is actively used by both staff and students as an additional platform for providing help. Like with any platform, there are some people who use it more than others. Users who contribute more are awarded a higher server rank: ranks that are closer to 1 indicate a higher level of engagement in the server. Those with upper echelon ranks are often respected as valuable members of the EECS 281 Discord community, as shown:



You are implementing a class that can be used to identify who the highest ranked contributors in the Discord server are. For this example, you are more concerned with the quality of messages that are sent (rather than quantity), so the rank of a user will be determined by the user's *total word count*. Words in a message are separated by spaces (e.g., the message "worst-case time complexity" has a word count of 3). Assume that all usernames are unique. An outline of this class is shown below:

```
class DiscordRankManager {
private:
  // TODO: Add any data structures here!
public:
  DiscordRankManager() {}

  // Invoked when a user posts a message in the server
  void post_message(const std::string& user, const std::string& message) {
    // TODO: Implement code here
  } // post_message()

  // Gets the word count of a specific user
  int32_t get_word_count(const std::string& user) {
    // TODO: Implement code here
  } // get_word_count()

  // Gets all users with word count in range [min_word_count, max_word_count]
  std::vector<std::string> get_user_range(int32_t min_word_count, int32_t max_word_count) {
    // TODO: Implement code here
  } // get_word_count_range()

  // Gets the user with the highest word count
  std::string get_highest_ranked_user() {
    // TODO: Implement code here
  } // get_highest_ranked_user()
};
```

The methods of this class are summarized below:

- `post_message()`: Takes in a username and a message, indicates that a message has been sent. Assume the username and message string will both never be empty.
  - For example, a call to `post_message("slime", "@everyone")` means that user `"slime"` sent the message `"@everyone"` in the server.
- `get_word_count()`: Gets the word count of the specified user. If the user has not posted anything, return 0.
  - For example, if `post_message("slime", "@everyone")` is called (and `"slime"` posts no other messages), `get_word_count("slime")` should return 1.
- `get_user_range()`: Returns all users with a word count within the specified range. If there are no users, return an empty vector.
- `get_highest_ranked_user()`: Gets the user with the highest word count. If two users have the same word count, the one with the lexicographically smaller username has the higher rank. If there are no users, return an empty string.
  - For example, if `"slime"` has the highest word count, then `get_highest_ranked_user()` should return `"slime"`.
  - If both `"slime"` and `"doubledelete"` have the same word count, then `"doubledelete"` would have the higher rank since their username is lexicographically smaller (`"d" < "s"`).

Example: Consider the following conversation in the EECS 281 Discord (assume these are the only messages that have been sent):

7/21/2022 9:29:47 PM **Khuldraeseth:** Which is heavier, a pound of feathers or a pound of gold?
7/21/2022 9:30:18 PM **slime:** pound of feathers
7/21/2022 9:30:25 PM **Khuldraeseth:** yes
7/21/2022 9:30:32 PM **doubledelete:** depends. pound as in lb or currency
7/21/2022 9:31:06 PM **Khuldraeseth:** weight
7/21/2022 9:31:52 PM **Khuldraeseth:** Which is heavier, an ounce of feathers or an ounce of gold?
7/21/2022 9:33:43 PM **slime:** going to use my phone a friend
7/21/2022 9:34:07 PM **Khuldraeseth:** what is "friend"? this is CS
7/21/2022 9:34:27 PM **slime:** i don't know it's the last lifeline i have left
7/21/2022 9:36:41 PM **doubledelete:** friend class is the only friend i have
7/21/2022 9:37:32 PM **iamr:** An ounce of gold weighs more on the scales of society
7/21/2022 9:38:02 PM **Khuldraeseth:** also more on the scale in my bathroom
7/21/2022 9:39:39 PM **Khuldraeseth:** The key is that gold is measured in pounds Troy and ounces Troy, which are respectively less and greater than their Avoirdupois counterparts

In this example:

- `get_highest_ranked_user()` would return `"Khuldraeseth"`, since this is the user with the highest word count among these messages.
- `get_word_count("doubledelete")` would return 15, since that is the total number of words sent by user `"doubledelete"` (7 words in first message at 9:30:32 PM, 8 words in second message at 9:36:41 PM).
- `get_user_range(1, 30)` would return `["slime", "doubledelete", "iamr"]` (in any order), since this is the list of users whose total word count falls in the range [1, 30].

There is a lot to digest with this problem, so let us break it down into smaller components. First, we will need a way to keep track of the word count of each user on the server. This can be done using a lookup container that maps each person's username to their total word count. Since the order of usernames does not matter in determining word count, an `std::unordered_map<>` would suffice in storing this data. Every time a new message is posted in the server, we count the number of words in the string and add it to the total word count of its user. This allows us to implement the `get_word_count()` function by simply querying the map, as shown:

```
class DiscordRankManager {
private:
  std::unordered_map<std::string, int32_t> user_to_word_count_map;
public:
  DiscordRankManager() {}

  void post_message(const std::string& user, const std::string& message) {
    // get word count of message
    std::stringstream msg_stream(message);
    std::string word;
    int32_t msg_word_count = 0;
    while (msg_stream >> word) {
      ++msg_word_count;
    } // while
    // add word count to map
    user_to_word_count_map[user] += msg_word_count;
  } // post_message()

  int32_t get_word_count(const std::string& user) {
    auto it = user_to_word_count_map.find(user);
    return it == user_to_word_count_map.end() ? 0 : it->second;
  } // get_word_count()
};
```

However, the `get_user_range()` method makes things a bit trickier. Notice that this method requires us to identify users based on word count values. Since our previous `std::unordered_map<>` maps from username to word count, it will not help us here; instead, we need a way to map from word count back to username. Therefore, we will use another lookup container to support this mapping. In this case, the word count values should be stored in sorted order, as that allows us to more efficiently identify all word counts that fall into a given range. As a result, we will use a `std::map<>` to map from word count to user. A updated solution with `get_user_range()` implemented is shown below; note that we will need to update the `std::map<>` whenever a new message is posted, as the word count for a given user would change.

```
class DiscordRankManager {
private:
  std::unordered_map<std::string, int32_t> user_to_word_count_map;
  std::map<int32_t, std::string> word_count_to_user_map;
public:
  DiscordRankManager() {}

  void post_message(const std::string& user, const std::string& message) {
    // get word count of message
    std::stringstream msg_stream(message);
    std::string word;
    int32_t msg_word_count = 0;
    while (msg_stream >> word) {
      ++msg_word_count;
    } // while
    // get old word count of user, and remove old value from corresponding map
    int32_t& total_word_count = user_to_word_count_map[user];
    word_count_to_user_map.erase(total_word_count);
    // get new word count by adding in the new message, and add it into map
    total_word_count += msg_word_count;
    word_count_to_user_map[total_word_count] = user;
  } // post_message()

  int32_t get_word_count(const std::string& user) {
    auto it = user_to_word_count_map.find(user);
    return it == user_to_word_count_map.end() ? 0 : it->second;
  } // get_word_count()

  std::vector<std::string> get_user_range(int32_t min_word_count, int32_t max_word_count) {
    auto lower_bound = word_count_to_user_map.lower_bound(min_word_count);
    auto upper_bound = word_count_to_user_map.upper_bound(max_word_count);
    // return vector with all contents between lower_bound and upper_bound iterators
    std::vector<std::string> users;
    for (auto it = lower_bound; it != upper_bound; ++it) {
      users.push_back(it->second);
    } // for it
    return users;
  } // get_user_range()
};
```

This implementation is not fully correct, however, as it incorrectly assumes that the word counts of all the users are distinct. However, multiple users may have the same word count! Thus, instead of mapping each word count to a single user, we have to map each word count to a container of users. What type of container should be use? In this problem, the ordering of users with the same word count does matter (as we will see when we implement the `get_highest_ranked_user()` method), so our container should ideally be sorted as well. Since we can safely assume that all usernames are distinct, we can use a `std::set<>` to store all users that share the same word count. This change is reflected below (differences on line 4, 18-23, 26, and 39-42):

```
1  class DiscordRankManager {
2  private:
3    std::unordered_map<std::string, int32_t> user_to_word_count_map;
4    std::map<int32_t, std::set<std::string>> word_count_to_user_map;
5  public:
6    DiscordRankManager() {}
7
8    void post_message(const std::string& user, const std::string& message) {
9      // get word count of message
10     std::stringstream msg_stream(message);
11     std::string word;
12     int32_t msg_word_count = 0;
13     while (msg_stream >> word) {
14       ++msg_word_count;
15     } // while
16     // get old word count of user, and remove user's old entry from map
17     int32_t& total_word_count = user_to_word_count_map[user];
18     std::set<std::string>& users_with_word_count =  word_count_to_user_map[total_word_count];
19     users_with_word_count.erase(user);
20     // if no user has this word count anymore, remove the word count from the map
21     if (users_with_word_count.empty()) {
22       word_count_to_user_map.erase(total_word_count);
23     } // if
24     // get new word count by adding in the new message, and add it into map
25     total_word_count += msg_word_count;
26     word_count_to_user_map[total_word_count].insert(user);
27   } // post_message()
28
29   int32_t get_word_count(const std::string& user) {
30     auto it = user_to_word_count_map.find(user);
31     return it == user_to_word_count_map.end() ? 0 : it->second;
32   } // get_word_count()
33
34   std::vector<std::string> get_user_range(int32_t min_word_count, int32_t max_word_count) {
35     auto lower_bound = word_count_to_user_map.lower_bound(min_word_count);
36     auto upper_bound = word_count_to_user_map.upper_bound(max_word_count);
37     // return vector with all contents between lower_bound and upper_bound iterators
38     std::vector<std::string> users;
39     for (auto it = lower_bound; it != upper_bound; ++it) {
40       const std::set<std::string>& user_set = it->second;
41       users.insert(users.end(), user_set.begin(), user_set.end());
42     } // for it
43     return users;
44   } // get_user_range()
45 };
```

Since the map is sorted by word count, we can identify the highest ranked user by finding the user associated with the largest key in the map. If there are multiple users with this largest word count, we would return the first user in the corresponding set (since it is sorted lexicographically). An implementation of the `get_highest_ranked_user()` method is shown below:

```
1  class DiscordRankManager {
2  private:
3    std::unordered_map<std::string, int32_t> user_to_word_count_map;
4    std::map<int32_t, std::set<std::string>> word_count_to_user_map;
5  public:
6    DiscordRankManager() {}
7
8    /* ALL OTHER METHODS ARE THE SAME AS ABOVE */
9
10   std::string get_highest_ranked_user() {
11     if (user_to_word_count_map.empty()) {
12       return "";
13     } // if
14     auto word_count_it = word_count_to_user_map.rbegin();
15     const std::set<std::string>& user_set = word_count_it->second;
16     return *user_set.begin();
17   } // get_highest_ranked_user()
18 };
```

18.10 The STL Multimap and Multiset Containers (*)

The STL also provides the `std::multimap<>` and `std::multiset<>` containers (located in the `<map>` and `<set>` libraries, respectively), which are maps and sets that support duplicate keys. Both of these containers are implemented internally as self-balancing binary search trees that support duplicate elements. To iterate through all elements in a multimap or multiset that match a given key, you can use the iterator range returned by the `.equal_range()` member function.

```
template <typename K, typename V>
std::pair<iterator, iterator> std::multimap<K, V>::equal_range(const K& key);
std::pair<iterator, iterator> std::multiset<K>::equal_range(const K& key);
```

Returns an iterator range containing all elements in the container with the given key. Like with other STL algorithms, the first iterator returned is inclusive, and the second iterator returned is exclusive. Since multimaps and multisets are sorted, the first iterator points to the first element not less than `key`, and the second iterator points to the first element greater than `key`.

You will not need to know how to use these two containers in this class, but it is good to recognize that they exist. Because multimaps and multisets store their data in sorted order, they support one key feature that is *not* supported by unordered multimaps and unordered multisets: the order of elements with identical keys is determined by the *order of insertion*.

For example, consider the following code:

```
1  std::multimap<std::string, int32_t> classes;
2  classes.insert({"EECS", 280});
3  classes.insert({"EECS", 183});
4  classes.insert({"EECS", 376});
5  classes.insert({"EECS", 281});
6  classes.insert({"EECS", 370});
7  classes.insert({"EECS", 203});
8
9  auto iter_range = classes.equal_range("EECS");
10 for (auto it = iter_range.first; it != iter_range.second; ++it) {
11   std::cout << it->first << " " << it->second << '\n';
12 } // for it
```

The output of this code is:

```
EECS 280
EECS 183
EECS 376
EECS 281
EECS 370
EECS 203
```

Because we inserted the key "EECS" into the multimap with the value 280 first, we also ended up visiting this key-value pair first. The same applies for the remaining elements; the order of elements with identical keys in the multimap matches the insertion order of these elements. This is because duplicate elements are always inserted to the *right* of any existing node with the same key in the underlying binary search tree, and thus will always be visited later than keys that have already been inserted so far.

Although you do not need to know them, multimaps and multisets can still be used to elegantly solve different types of problems. For instance, consider the streaming median problem that we covered at the end of chapter 10. In that problem, we were given a stream of data, and we wanted to calculate the median of all values we have seen so far at any point in the stream. We previously solved this problem using a two-heap approach, where one heap stored all values in the smaller half of values seen, and the other heap stored all values in the larger half of values seen. Although this solution works well, it is actually possible to write a cleaner solution using multisets that exhibits the same asymptotic performance as the two-heap solution.

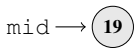
Behind the scenes, a `std::multiset<>` is simply a self-balancing binary search tree that supports duplicate keys. We can use this container to keep track of the streaming median with just one iterator that either points to the median element (if the tree size is odd) or the first of the middle two values (if the tree size is even). The algorithm is summarized as follows:

- First, instantiate a `std::multiset<>` (self-balancing binary search tree) and an iterator (`std::multiset<>::iterator`) that keeps track of the middle element.
- Every time you encounter a new data value `val` in the stream, there are three scenarios that can happen:
  1. The multiset is currently empty. In this case, insert `val` and set the iterator to this new element.
  2. The size of the multiset is currently odd, which means the iterator is currently pointing to the median value.
     - If `val` is smaller than the current value of the iterator, the insertion of `val` would cause the current value of the iterator to become the second of the median values (since the size of the set would become even). We want the iterator to point to the first of the middle two values, so we must decrement the iterator after the insertion.
     - If `val` is larger than (or equal to) the current value of the iterator, the insertion of `val` would cause the iterator to point to the first of the middle two values. We want the iterator to point to the first of the middle two values, so the iterator does not need to be modified after the insertion.
  3. The size of the multiset is currently even, which means the iterator is currently pointing to the first of the middle two values.
     - If `val` is smaller than the current value of the iterator, the insertion of `val` would cause the current value of the iterator to become the new median. Thus, the iterator does not need to be modified after the insertion.
     - If `val` is larger than (or equal to) the current value of the iterator, the insertion of `val` would cause the iterator to point to the value directly before the new median. We want the iterator to point to the median, so we must increment the iterator after the insertion.

By following this algorithm, we can calculate the median of a stream of numbers by either dereferencing the iterator if size is odd, or taking the average of the iterator and the value of its inorder successor if size is even. For example, consider the following stream of values:

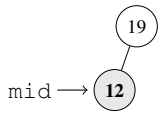
```
19 12 17 16 14 17 ...
```

Using a self-balancing binary search tree and a single iterator, we can calculate the median of the numbers we have encountered at any point in the stream. We start by inserting the first element into the binary search tree, 19, and setting the iterator to this value.



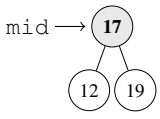
The size of the tree is odd, so `mid` points to the current streaming median.

The next value in the stream is 12, which is smaller than the current value of `mid`. Since 12 < `mid` and the size of the tree is odd before the insertion, we decrement `mid` (since multisets are sorted, `mid` now points to the inorder predecessor of 19).



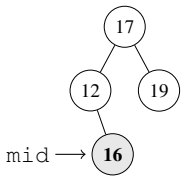
The size of the tree is even, so `mid` points to the first of the middle two values. Thus, the streaming median is now (12 + 19) / 2 = 15.5.

The next value in the stream is 17, which is larger than the current value of `mid`. Since 17 ≥ `mid` and the size of the tree is even before insertion, we increment `mid` from 12 to 17.



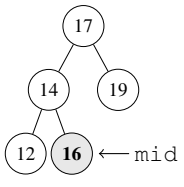
The size of the tree is odd, so `mid` points to the median value. Thus, the streaming median is now 17.

The next value in the stream is 16, which is smaller than the current value of `mid`. Since 16 < `mid` and the size of the tree is odd before insertion, we decrement the iterator from 17 to 16.



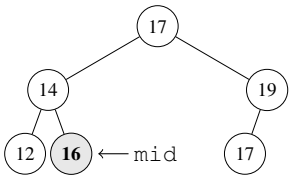
The size of the tree is even, so `mid` points to the first of the middle two values. Thus, the streaming median is now (16 + 17) / 2 = 16.5.

The next value in the stream is 14, which is smaller than the current value of `mid`. Since 14 < `mid` and the size of the tree is even before insertion, we do not need to move `mid`. This is because the value of `mid` (which used to point to the first of the middle two values) has now become the median value after the insertion of a value smaller than it.



The size of the tree is odd, so `mid` points to the median value. Thus, the streaming median is now 16.

The next value in the stream is 17, which is larger than the current value of `mid`. Since 17 ≥ `mid` and the size of the tree is odd before insertion, we do not need to move `mid`. This is because the value of `mid` (which used to point to the median) has now become the first of the middle two values after the insertion of a value larger than it.



The size of the tree is even, so `mid` points to the first of the middle two values. Thus, the streaming median is now (16 + 17) / 2 = 16.5.

This procedure can be repeated for the remaining elements in the stream. Since a `std::multiset<>` is always balanced, the time complexity of inserting an element or moving the `mid` iterator is bounded by $\Theta(\log(n))$ in the worst case. This matches the time complexity of the two-heap solution discussed at the end of chapter 10. An implementation of this algorithm using a multiset is shown below, within a median finder class:

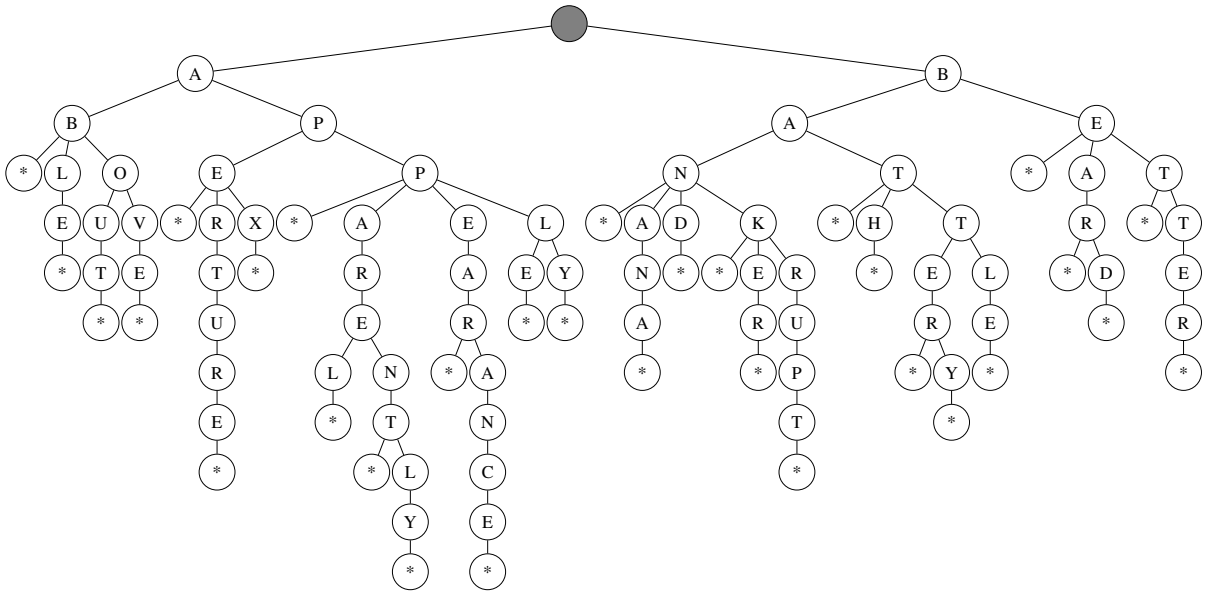
```
class MedianFinder {
private:
  std::multiset<int32_t> data;  // self-balancing BST
  std::multiset<int32_t>::iterator mid;
public:
  MedianFinder() : mid{data.end()} {}

  // insert a new value "val" into the collection of values seen so far
  void add_value(int32_t val) {
    size_t prev_size = data.size();
    data.insert(val);
    // if container was empty before insertion, set mid to this element
    if (prev_size == 0) {
      mid = data.begin();
    } // if
    // if container was odd before insertion and val < *mid, decrement mid
    else if (val < *mid && prev_size % 2 == 1) {
      --mid;
    } // else if
    // if container was even before insertion and val >= *mid, increment mid
    else if (val >= *mid && prev_size % 2 == 0) {
      ++mid;
    } // else if
  } // add_value()

  // returns the median of all values seen so far
  // assumes this does not get called if nothing has been added so far
  double find_median() {
    // if container is odd, mid points to the median
    // if container is even, mid points to the first of the middle two values
    // std::next() returns an iterator pointing to ++mid (see chapter 11)
    return data.size() % 2 == 1 ? *mid : static_cast<double>(*mid + *std::next(mid)) / 2;
  } // find_median()
};
```

18.11 Tries (⁎)

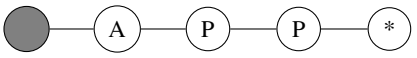
※ 18.11.1 *Trie Structure* (⁎)

In this section, we will discuss the **trie** data structure (short for *re**trie**val tree*, but commonly pronounced as "try" to distinguish it from the word "tree"). Tries are also known as *prefix trees*. Tries aren't required knowledge for this class, but they are included here because they occasionally show up during coding interviews.

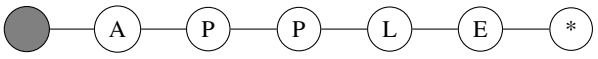
A trie is a tree-like data structure that stores characters at each node. Each branch down the tree can be used to represent a word, where words that share a common prefix share the same ancestor node in the tree. Tries can be used to organize a set of strings and perform fast lookups to determine if a string exists in the set. Tries are also helpful for conducting quick prefix lookups, and they play an important role in algorithms ranging across many different fields, from search engine development to genome analysis. An example of a trie is shown below:



The * nodes are used to indicate a complete word. For example, the branch



represents the word "app". On the other hand, the branch



represents the word "apple". Because "app" and "apple" share a similar prefix, they also share ancestor nodes in the trie.

How can we implement a trie? One implementation is shown below. First, we will define an object representing each node of the trie. If the trie can only store the 26 letters of the alphabet, we can store the children of each node in an array of size 26.

```
constexpr size_t ALPHABET_SIZE = 26;

struct TrieNode {
  TrieNode* children[ALPHABET_SIZE] = {nullptr};
  bool is_end_of_word = false;
  ~TrieNode();
};
```

However, we will implement a trie that can work beyond the 26 letters of the English alphabet. Since a `char` can take on 128 different ASCII values, we will instead store each node's children in an array of size 128. (Note: You can also store the children in an associative container, such as an `std::unordered_map<>` that maps a `char` to a `TrieNode*` associated with that character. However, that approach is more memory intensive since hash tables require more memory than arrays.)

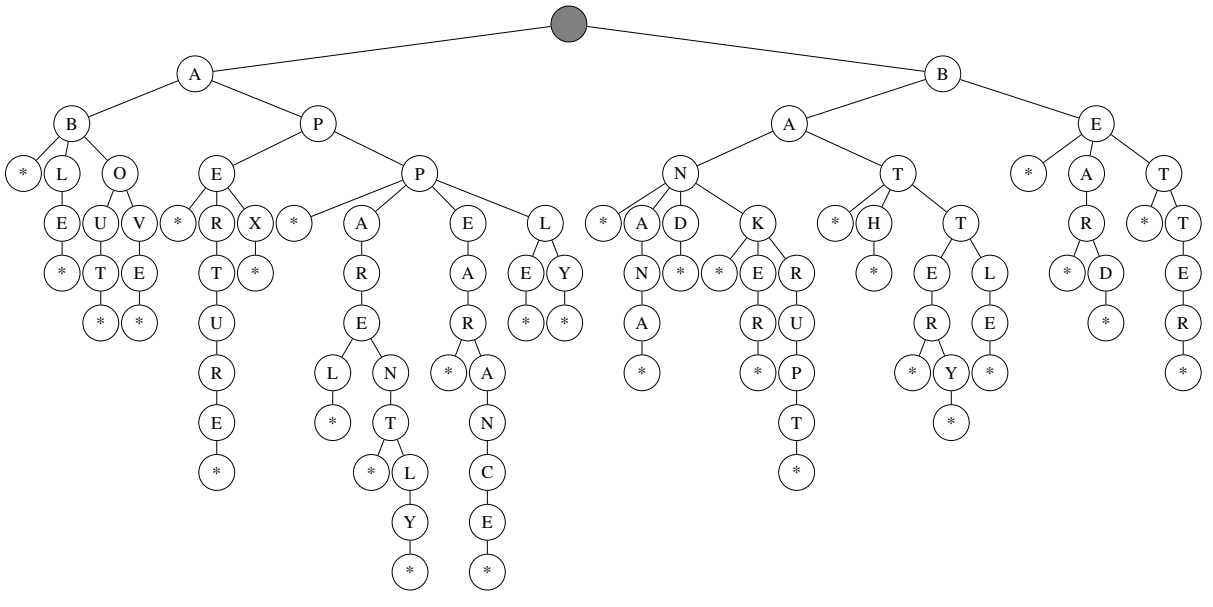
```
constexpr size_t CHAR_SIZE = 128;

struct TrieNode {
  TrieNode* children[CHAR_SIZE] = {nullptr};
  bool is_end_of_word = false;
  ~TrieNode();
};
```

As shown previously, each `TrieNode` also stores an "end of word" Boolean that is set to `true` if the node represents the end of a word (i.e., they perform the functionality of the * nodes in the above illustration). Since our trie will manage dynamic memory, we also define a destructor for each `TrieNode`, which iterates through the map and frees the memory associated with each of the children. The implementation of the destructor is shown below.

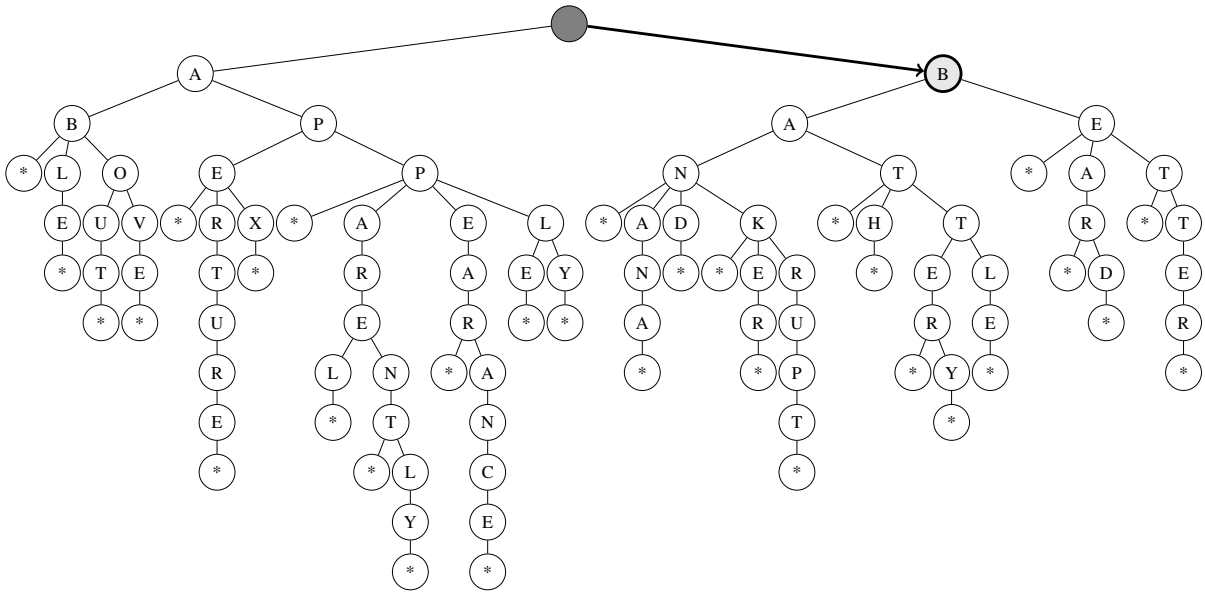
```
TrieNode::~TrieNode() {
  for (TrieNode* child : children) {
    delete child;
    child = nullptr;
  } // for child
} // ~TrieNode()
```

※ 18.11.2 *Inserting into a Trie* (✻)

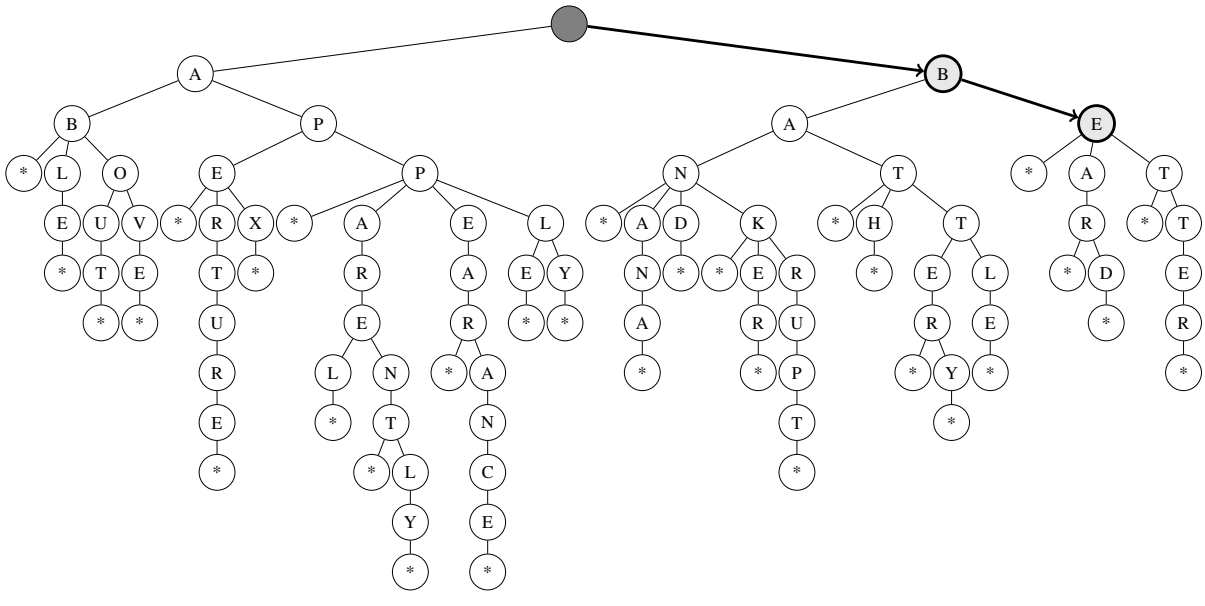
Our trie should be able to support word insertion, search, and removal. To insert a string into the trie, we traverse through each character of the string and walk down the trie until we encounter a letter that is present in the string but not in the corresponding position in the trie. When this happens, we allocate a new trie node at that position and build out the branch for the new string (appending a terminating * node at the end). For example, suppose we wanted to insert the string "bean" into the following trie:



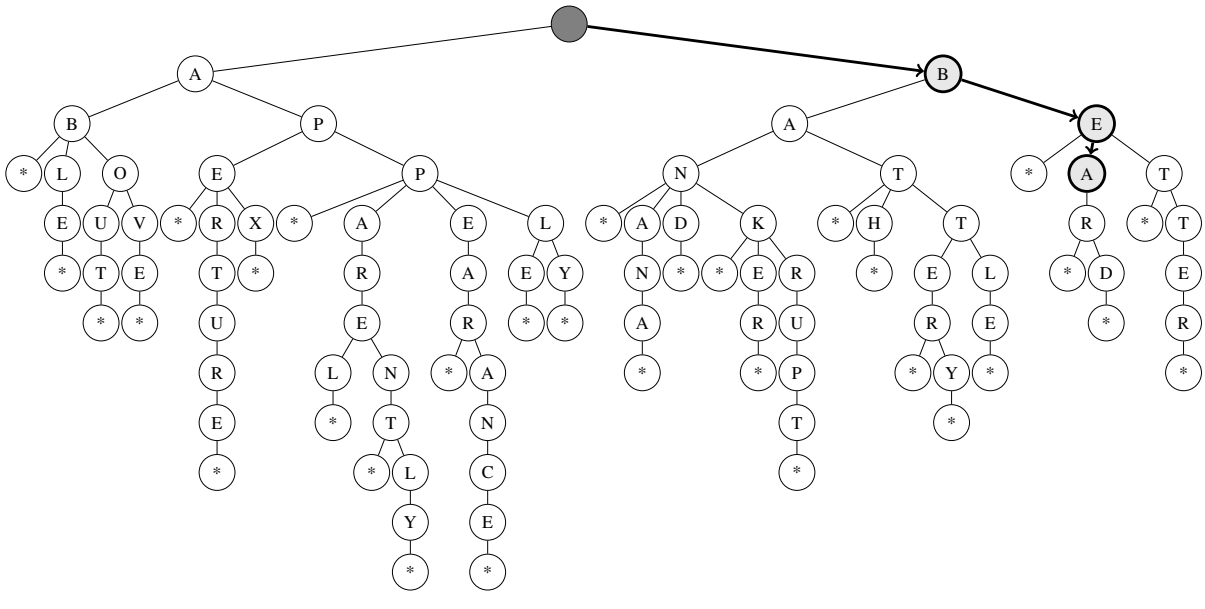
We start off by traversing through the characters of "bean" and walking down the trie. The first character is "B", so we check to see if "B" is a child of the root. It is, so we walk down to this node.



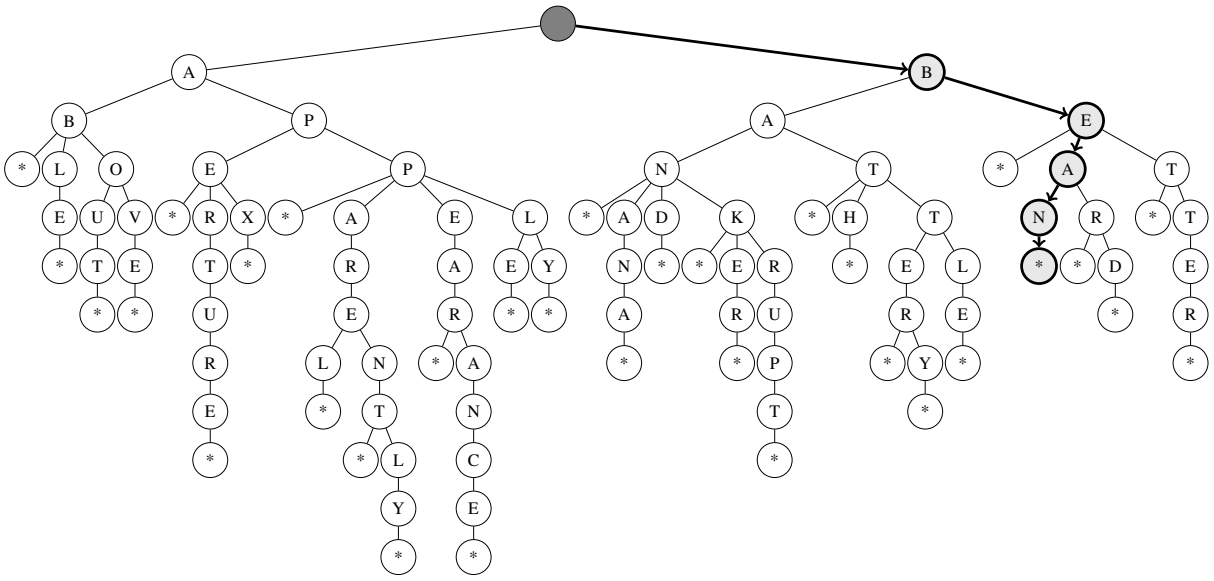
The next character of "bean" is "E", so we check to see if "E" is a child of "B". It is, so we walk down to this node.



The next character of "bean" is "A", so we check to see if "A" is a child of "E". It is, so we walk down to this node.



The next character of "bean" is "N", so we check to see if "N" is a child of "A". It is not, so we will have to construct a new `TrieNode` with the value "N" at this position. To do so, we will allocate a new `TrieNode` for "N" and add it to the `std::unordered_map<>` of the current node "A". We would continue adding new nodes for each additional character in the string. However, "N" is the last letter of the string "bean", so we mark it as the end of a word.



The string has been successfully inserted into the trie. The code for the insert operation is shown below:

```
1   class Trie {
2   public:
3     void insert(const std::string& word);
4     bool contains(const std::string& word) const;
5     bool remove(const std::string& word);
6     void clear();
7     size_t size() const;
8     ~Trie();
9   private:
10    TrieNode* root = nullptr;
11    size_t sz = 0;
12    bool has_valid_children(TrieNode* nodes[], size_t size);
13  };
14
15  void Trie::insert(const std::string& word) {
16    if (root == nullptr) {
17      root = new TrieNode;
18    } // if
19    TrieNode* curr = root;
20    for (size_t i = 0; i < word.size(); ++i) {
21      if (curr->children[word[i]] == nullptr) {
22        curr->children[word[i]] = new TrieNode;
23      } // if
24      curr = curr->children[word[i]];
25    } // for i
26    if (!curr->is_end_of_word) {
27      curr->is_end_of_word = true;
28      ++sz;
29    } // if
30  } // insert()
```

On line 16, we first check to see if the trie is currently empty. If it is, we create the root node on line 17. Then, we iterate through the characters of the string we want to insert (`word`) in the `for` loop on line 20. For each character, we check to see if it already exists as a child of the current node. If the character does not exist as a child, we add it on line 22. Then, we walk down the trie by visiting the node corresponding to the next character on line 24. Lastly, after we visit the last character of the string, we mark its node as the final character by setting its `is_end_of_word` Boolean to `true` on line 27. The overall time complexity of this operation is $\Theta(k)$, where k is the length of the string that is inserted. However, since the length of a string is typically a small constant, we often describe trie insertion as a constant time operation.

※ 18.11.3 *Searching in a Trie* (✻)

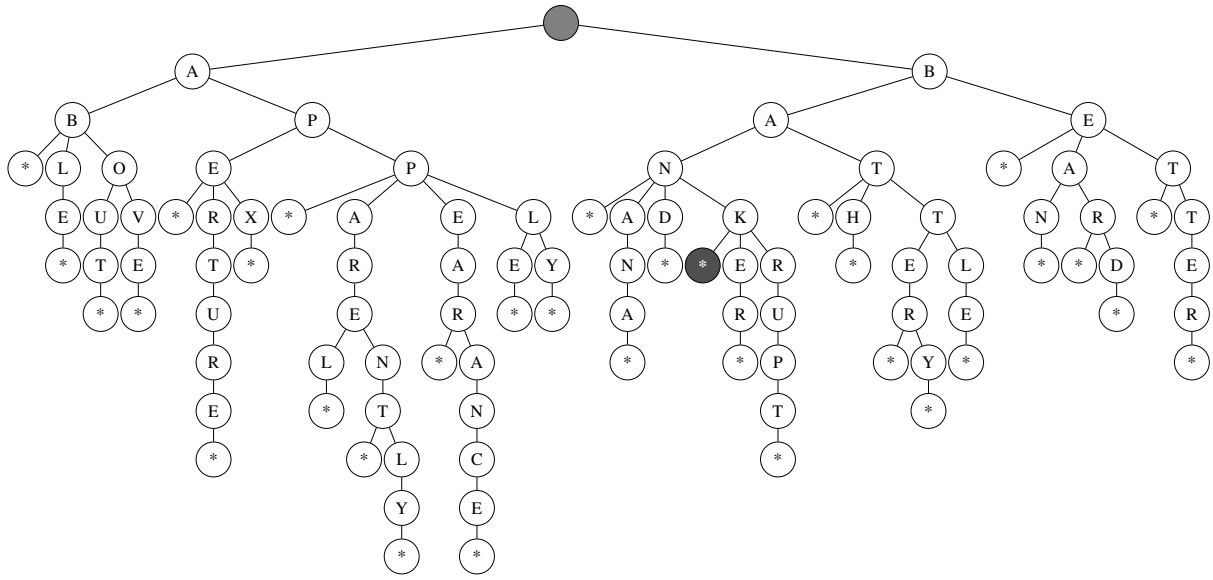
To check if a string exists in a trie, we can follow a similar process. Similar to insert, we would traverse through the given string and walk down the trie. However, if at any point we encounter a character in the string that does not exist at the current position of the trie, we can immediately conclude that the string does not exist in the trie. If we are able to walk down all the characters of the string without encountering any missing letters, we will use the value of the `is_end_of_word` Boolean of the final letter to determine if the string itself actually exists in the trie. The code for determining if a string exists in the trie is shown below:

```
1  bool Trie::contains(const std::string& word) const {
2    TrieNode* curr = root;
3    if (curr == nullptr) {
4      return false;
5    } // if
6    for (size_t i = 0; i < word.size(); ++i) {
7      if (curr->children[word[i]] == nullptr) {
8        return false;
9      } // if
10     curr = curr->children[word[i]];
11   } // for i
12   return curr->is_end_of_word;
13 } // contains()
```

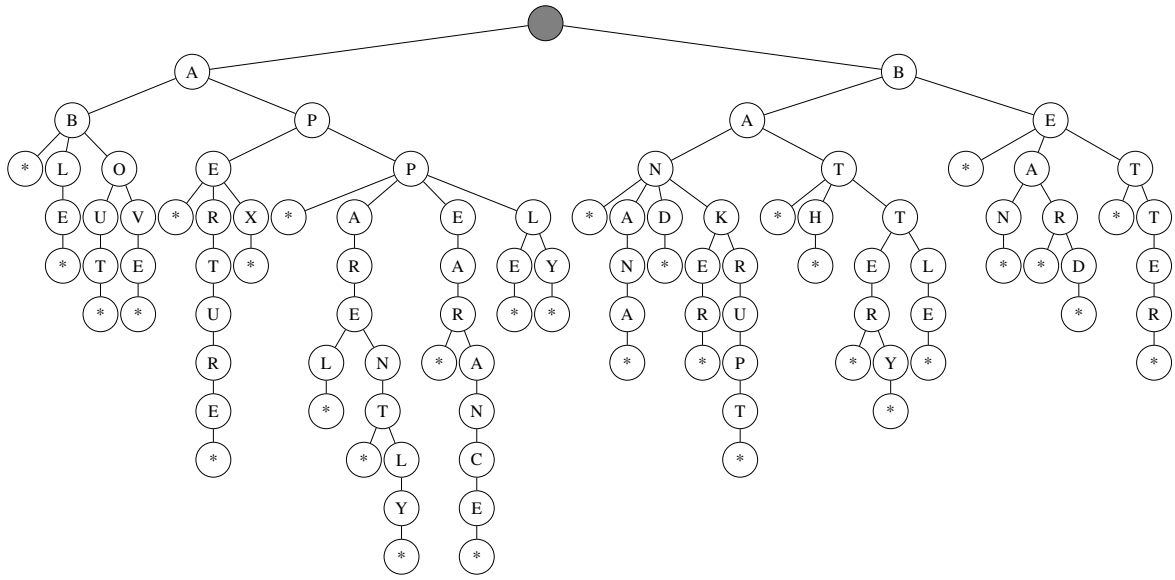
The time complexity of checking whether a string exists in the trie is worst-case $\Theta(k)$, where k is length of the string to check. Once again, assuming that strings have constant size, we can treat the search process as something that takes constant time.

※ 18.11.4 *Removing from a Trie* (✻)

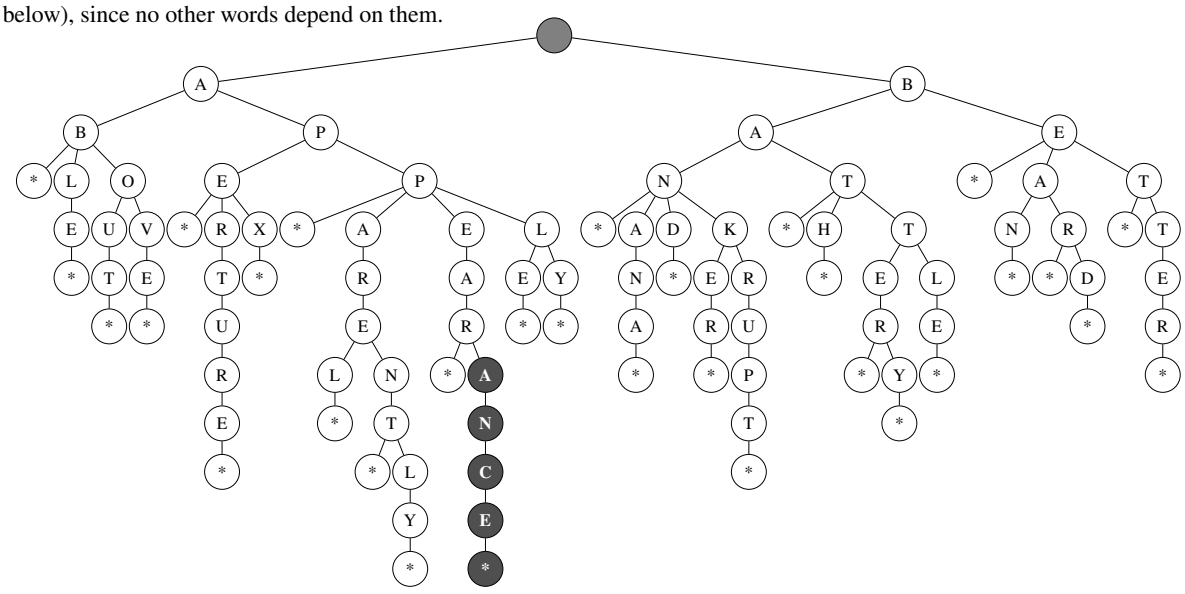
Removing a string from the trie is slightly trickier. For example, suppose we wanted to remove "bank" from the following trie:



We cannot just remove the characters associated with "bank", since other strings in the trie may rely on it (e.g., "banker" and "bankrupt"). In this case, the correct behavior would be to remove the * node associated with the "K" in "bank" (shaded above):



However, if we wanted to remove the word "appearance" from the following trie, we would be able to delete the last four letters from the trie (shaded below), since no other words depend on them.



The basic idea here is that nodes in the trie can only be deallocated if no other words depend on them (i.e., they have no children). Thus, in our deletion algorithm, we would walk down the branch of the word we want to delete (and verify that the word actually exists in the trie), set the `is_end_of_word` value of the final character to `false`, and then delete any nodes along the branch that have no children (starting from the bottom of the trie and moving upward). The code is shown below (the Boolean returned indicates if the removal was successful):

```
1   bool Trie::remove(const std::string& word) {
2     TrieNode* curr = root;
3     if (curr == nullptr) {
4       return false;
5     } // if
6     std::vector<TrieNode*> nodes_of_word(word.size(), nullptr);
7     for (size_t i = 0; i < word.size(); ++i) {
8       TrieNode* child = curr->children[word[i]];
9       if (child == nullptr) {
10        return false;
11      } // if
12      nodes_of_word[i] = child;
13      curr = child;
14    } // for i
15    if (!curr->is_end_of_word) {
16      return false;
17    } // if
18    curr->is_end_of_word = false;
19    --sz;
20    TrieNode* node_ptr = nullptr;
21    for (size_t i = word.size() - 1; i > 0; --i) {
22      node_ptr = nodes_of_word[i];
23      if (!node_ptr->is_end_of_word && !has_valid_children(node_ptr->children, CHAR_SIZE)) {
24        delete node_ptr;
25        node_ptr = nullptr;
26      } // if
27      else {
28        return false;
29      } // else
30      nodes_of_word[i - 1]->children[word[i]] = nullptr;
31    } // for i
32    node_ptr = nodes_of_word[0];
33    if (!node_ptr->is_end_of_word && !has_valid_children(node_ptr->children, CHAR_SIZE)) {
34      delete node_ptr;
35      node_ptr = nullptr;
36      root->children[word[0]] = nullptr;
37    } // if
38    return true;
39  } // remove()
40
41  bool Trie::has_valid_children(TrieNode* nodes[], size_t size) {
42    for (size_t i = 0; i < size; ++i) {
43      if (nodes[i] != nullptr) {
44        return true;
45      } // if
46    } // for i
47    return false;
48  } // has_valid_children()
```

In the code for deletion, we store a vector of parent pointers so that we can walk back up the trie and deallocate memory for nodes that are no longer needed. On lines 11 and 17, we exit early if the string we want to delete does not exist in the trie. If the string does exist, we set its `is_end_of_word` Boolean to `false` (line 19) and decrement the size counter (line 20). Then, we walk back up the tree and deallocate nodes that have no children and are not the end of an existing word (lines 21 to 40). Similar to search, the time complexity of removing a string from a trie is worst-case $\Theta(k)$, where k is length of the string to check. If we assume that strings have constant size, we can treat the removal process as a constant time operation as well.

※ 18.11.5 *Implementing the Trie Destructor* (✻)

The following code implements the remaining functionality of the trie. The `.clear()` function recursively clears the trie in postorder fashion, deallocating all of its memory and bringing its size to zero (notice that the entire trie is cleaned up on line 2, since the `TrieNode*` destructor recursively deletes all of its children). The trie destructor utilizes the `.clear()` method to clean up the trie after it is destructed. The `.size()` function, as its name suggests, returns the size of the trie.

```
1   void Trie::clear() {
2     delete root;
3     root = nullptr;
4     sz = 0;
5   } // clear()
6
7   size_t Trie::size() const {
8     return sz;
9   } // size()
10
11  Trie::~Trie() {
12    clear();
13  } // ~Trie()
```

Outside of programming interviews, why should we care about the trie data structure? Tries do have extensive practical applications. For instance, they can be used to develop spell checkers or autocomplete functionality (e.g., when you type something into a search engine, you will likely get a bunch of search suggestions that start with the letters you typed in). This data structure also plays an important role in routing systems. Most routers store IP addresses in an optimized version of a trie — this allows routers to conduct fast IP lookups to determine where data packets should be forwarded. As an example, the router in the following illustration uses a trie to look up the destination IP prefix of an incoming data packet, and it uses the result to determine which port to send the data packet out from.

Port	Destination IP Prefix Range
1	11000000 to 11000011
2	11000100 to 11000111
3	11001000 to 11001111
4	11010000 to 11011111