Binary Search Trees

Instructors: Sam McCauley and Dan Barowy
April 20, 2022
Admin

- Sign up to be a TA! Deadline Friday
  - End of the form asks to list professors; pretty much anyone you’ve had is probably fine
  - If you want we can have a brief conversation where I say I’m OK with you putting my name down

- Lab 8 tomorrow: please read over the lab and create a design document before your lab
  - We’ll actually collect them this week
  - Very important to get a head start on the lab

- We’ll briefly discuss course registration Friday

- Any questions?
Tree Iterators
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don’t want to traverse the tree, store all tree values, and then dispense them one by one
  - Instead: each call to `next()` should go to the next node in the tree we want to output
- Challenge: implementing a recursive traversal piece-by-piece
- To think about: what data structure helps with recursion?
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child

```
18
   9  24
  5 12 22 30
     29 35
```
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
- What happens when we hit a leaf?
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
- Instead: maintain nodes to visit on a stack!
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we *still need to traverse*

- So `next()`:
  - `pops` the top item off the stack
  - `Stores` its value to be returned
  - `Pushes` its right child onto the stack if nonempty
  - `Pushes` its left child onto the stack if nonempty

- `hasNext()`?
  - `Just` returns if the stack is empty
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse
- So next():
  - pops the top item off the stack
Pre-order traversal

- Stack maintains the non-empty `BinaryTree<E>` objects that we *still need to traverse*

- So `next()`:
  - *pops* the top item off the stack
  - Stores its value to be returned
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we *still need to traverse*

- So next():
  - **pops** the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we *still need to traverse*

- So `next()`:
  - *pops* the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse

- So next():
  - pops the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty

- hasNext()?
Pre-order traversal

- Stack maintains the non-empty `BinaryTree<E>` objects that we *still need to traverse*

- So `next()`:
  - *pops* the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty

- `hasNext()`?
  - Just returns if the stack is empty
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side.
- In other words: want to output the root after the left child has been completely traversed.
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side.
- In other words: want to output the root after the left child has been completely traversed.
- Seems like we want the root at the very bottom of the stack. We’ll keep it at the bottom of the stack as we traverse the left subtree; then when we pop the root off we’ll output its value and traverse the right child.
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side
- In other words: want to output the root after the left child has been completely traversed
- Seems like we want the root at the very bottom of the stack. We’ll keep it at the bottom of the stack as we traverse the left subtree; then when we pop the root off we’ll output its value and traverse the right child
- Nice idea, but it takes some care. Let’s be a bit more specific
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to next():
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on
- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on

- `hasNext()`: return if the stack is nonempty
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on

- `hasNext()`: return if the stack is nonempty

- Let’s look at the code
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

**Stack:** 18 9 5
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

Stack: 18 9
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type `BinaryTree`.

Stack: 18
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`

**Stack:** 24 22
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

Stack: 24
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Stack: 30 29
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type *BinaryTree*.

**Stack:** 30
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`

**Stack:** 35
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.
Post-order traversal

- Same idea as in-order traversal
Post-order traversal

- Same idea as in-order traversal
- Output the node when popping from the stack
Post-order traversal

- Same idea as in-order traversal
- Output the node when popping from the stack
- If you pop a node, and it’s the left child of its parent, push the parent’s right child (and leftmost descendants) onto the stack
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order traversal

- Level-order traversal is not recursive!
Level-order traversal

- Level-order traversal is not recursive!

- How do we keep track of what nodes to visit next?
Level-order traversal

• Level-order traversal is not recursive!

• How do we keep track of what nodes to visit next?

• Key insight: the order we visit nodes at a given “level” is the same order we visited their parents
Level-order traversal

- Level-order traversal is not recursive!

- How do we keep track of what nodes to visit next?

- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents

- So the *first* parents to be visited have the *first* children that are visited
Level-order traversal

- Level-order traversal is not recursive!

- How do we keep track of what nodes to visit next?

- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents

- So the \textit{first} parents to be visited have the \textit{first} children that are visited

- ...Can we use a queue?
Level-order iterator

- To begin: push root onto the queue
Level-order iterator

- To begin: push root onto the queue

- next():
Level-order iterator

- To begin: push root onto the queue

- `next()`:
  - Dequeue node off the queue; store its value to be returned

- `hasNext()`:
  - Return if queue is empty
Level-order iterator

- To begin: push root onto the queue

- next():
  - Dequeue node off the queue; store its value to be returned
  - Enqueue its non-empty children onto the queue
Level-order iterator

- To begin: push root onto the queue

- next():
  - Dequeue node off the queue; store its value to be returned
  - Enqueue its non-empty children onto the queue

- hasNext(): return if queue is empty
Level-order iterator

- To begin: push root onto the queue

- next():
  - Dequeue node off the queue; store its value to be returned
  - Enqueue its non-empty children onto the queue

- hasNext(): return if queue is empty

- Let's look at the code
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type *BinaryTree*.

**Queue**: 9 24
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type `BinaryTree`.

Queue: 24 5 12
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type BinaryTree

Queue: 5 12 22 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

Queue: 12 22 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 22 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type `BinaryTree`.

Queue: 30
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type BinaryTree

Queue: 29 35
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 35
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`.

Queue:
Binary Search Trees
Finding Items Using Trees

- Goal: store items in a tree such that we can implement methods like `add()` and `contains()` efficiently

Finding Items Using Trees

- Goal: store items in a tree such that we can implement methods like `add()` and `contains()` efficiently.
- Don’t want to traverse the entire tree.
Finding Items Using Trees

- Goal: store items in a tree such that we can implement methods like `add()` and `contains()` efficiently

- Don’t want to traverse the entire tree

- In an OrderedVector we store items in order to allow for efficient binary search
Finding Items Using Trees

- Goal: store items in a tree such that we can implement methods like `add()` and `contains()` efficiently.
- Don’t want to traverse the entire tree.
- In an `OrderedVector` we store items in order to allow for efficient binary search.
  - Though `add()` is still slow.
Finding Items Using Trees

- Goal: store items in a tree such that we can implement methods like `add()` and `contains()` efficiently

- Don’t want to traverse the entire tree

- In an `OrderedVector` we store items in order to allow for efficient binary search
  - Though `add()` is still slow

- How can we do something similar for trees?
Binary Search Tree Invariant

- For every node $n$ in a binary search tree with value $v$:
Binary Search Tree Invariant

• For every node \( n \) in a binary search tree with value \( v \):

  • All values \( v_\ell \) of nodes that are descendants of the left child have values \( v_\ell \leq v \)
Binary Search Tree Invariant

- For every node \( n \) in a binary search tree with value \( v \):
  - All values \( v_\ell \) of nodes that are descendants of the left child have values \( v_\ell \leq v \)
  - All values \( v_r \) of nodes that are descendants of the right child have values \( v_r > v \)
Binary Search Tree Examples

Is this a binary search tree?
Is this a binary search tree? (It has the same elements!)
Is this a binary search tree?
Is this a binary search tree?

No: note that *all* right descendants must be greater than the node.
Finding an element in a binary search tree

• How can I search for an element (say 14)?
Finding an element in a binary search tree

• How can I search for an element (say 14)?
• Recursively!
Finding an element in a binary search tree

• How can I search for an element (say 14)?
• Recursively!
• Idea: we can look at a node and know immediately if the element we’re searching for is a descendant of the left child, or of the right child
Finding an element in a binary search tree

- How can I search for an element (say 14)?
- Recursively!
- Idea: we can look at a node and know immediately if the element we’re searching for is a descendant of the left child, or of the right child
- Recurse on the appropriate node
Finding an element in a binary search tree

• How can I search for an element (say 14)?
• Recursively!
• Idea: we can look at a node and know immediately if the element we’re searching for is a descendant of the left child, or of the right child
• Recurse on the appropriate node
• If we find the element, or if we hit an empty node, we’re done
Adding an element to a binary search tree

- How can I add an element (say 23)?

```plaintext
   18
  /   \
 9     24
 /     /  \
5     22   30
        /    /
       29    35
```
How can I add an element (say 23)?

Recursively!
Adding an element to a binary search tree

- How can I add an element (say 23)?
- Recursively!
- Idea: we can look at a node and know immediately if the element we’re adding should be a descendant of the left child, or of the right child
Adding an element to a binary search tree

- How can I add an element (say 23)?
- Recursively!
- Idea: we can look at a node and know immediately if the element we’re adding should be a descendant of the left child, or of the right child
- Recurse on the appropriate node
Adding an element to a binary search tree

• How can I add an element (say 23)?
• Recursively!
• Idea: we can look at a node and know immediately if the element we’re adding should be a descendant of the left child, or of the right child
• Recurse on the appropriate node
• If we hit an empty node, replace it with the element we want to add
Adding an element to a binary search tree

- How can I add an element (say 23)?
- Recursively!
- Idea: we can look at a node and know immediately if the element we’re adding should be a descendant of the left child, or of the right child
- Recurse on the appropriate node
- If we hit an empty node, replace it with the element we want to add
Adding an element to a binary search tree: caveat!

• How can I add something to a BST that’s already in the tree
How can I add something to a BST that’s already in the tree
For example: add 9 to this tree
Adding an element to a binary search tree: caveat!

- How can I add something to a BST that’s already in the tree?
- For example: add 9 to this tree.
- Idea: first, find the element. Then, find an empty leaf where the new element can go.
Adding an element to a binary search tree: caveat!

• How can I add something to a BST that’s already in the tree
• For example: add 9 to this tree
• Idea: first, find the element. Then, find an empty leaf where the new element can go
• Rightmost descendant of left child
Adding an element to a binary search tree: caveat!

- How can I add something to a BST that’s already in the tree?
- For example: add 9 to this tree.
- Idea: first, find the element. Then, find an empty leaf where the new element can go.
- Rightmost descendant of left child.
Implementing a Binary Search Tree
Comparing Elements

- Need some kind of way to compare elements
Comparing Elements

- Need some kind of way to compare elements
- What are our options?
Comparing Elements

- Need some kind of way to compare elements
- What are our options?
  - Store Comparable items, or use a Comparator
Comparing Elements

- Need some kind of way to compare elements

- What are our options?
  - Store Comparable items, or use a Comparator
  - The structure5 BinarySearchTree<E> assumes comparable items, but also allows a Comparator to be used...how?
Natural Comparator

- Let’s say we have an item of type \( E \) that implements \( \text{Comparable}<E> \)
Natural Comparator

- Let’s say we have an item of type E that implements Comparable<E>
- That means we can already compare items of type E
Let’s say we have an item of type E that implements Comparable<E>

That means we can already compare items of type E

But, we want the flexibility to compare them other ways using a Comparator<E>
Natural Comparator

- Let’s say we have an item of type E that implements Comparable<E>
- That means we can already compare items of type E
- But, we want the flexibility to compare them other ways using a Comparator<E>
- The NaturalComparator<E> implements Comparator<E>, and compares items using their compareTo() method
Let’s say we have an item of type E that implements Comparable<E>

That means we can already compare items of type E

But, we want the flexibility to compare them other ways using a Comparator<E>

The NaturalComparator<E> implements Comparator<E>, and compares items using their compareTo() method

That way, we can write code assuming we always have a comparator; if we want we can replace it with a different comparator
Natural Comparator

- Let’s say we have an item of type E that implements Comparable<E>
- That means we can already compare items of type E
- But, we want the flexibility to compare them other ways using a Comparator<E>
- The NaturalComparator<E> implements Comparator<E>, and compares items using their compareTo() method
- That way, we can write code assuming we always have a comparator; if we want we can replace it with a different comparator
- Let’s look at the code
Binary Search Tree: Comparisons

- We'll assume our items are comparable. But, another constructor takes a Comparator to allow us to compare the items.
Binary Search Tree: Comparisons

- We’ll assume our items are comparable. But, another constructor takes a Comparator to allow us to compare the items.

- Let’s look at how these constructors work.
Building up the BST

- The BinaryTree class was recursive
Building up the BST

- The BinaryTree class was recursive

- On the other hand, BinarySearchTree is made up of BinaryTree's
Building up the BST

- The BinaryTree class was recursive

- On the other hand, BinarySearchTree is made up of BinaryTree

- Allows us to keep track of the number of items, a comparator, etc.
Building up the BST

- The BinaryTree class was recursive
- On the other hand, BinarySearchTree is made up of BinaryTree
- Allows us to keep track of the number of items, a comparator, etc.
- Now: let’s look at the code to locate an item, or to add it to the tree
Finding an element in a binary search tree

- Idea: we can look at a node and know immediately if the element we’re searching for is a descendant of the left child, or of the right child
- Recurse on the appropriate node
- If we find the element, or if we hit an empty node, we’re done
- Let’s look at the code
Adding an element to a binary search tree

- Idea: we can look at a node and know immediately if the element we’re adding should be a descendant of the left child, or of the right child
- Recurse on the appropriate node
- If we hit an empty node, replace it with the element we want to add
- If adding a duplicate element, find rightmost descendant of left child of current location
Tree Vocabulary

• Descendant: A node $n'$ is a descendant of node $n$ if there exists a sequence of nodes $n = n_1, n_2, \ldots, n_i = n'$ such that for all $1 \leq j < i$, $n_j$ is a child of $n_{j+1}$.

(Ancestor is the opposite)

• Siblings: Two nodes are siblings if they share the same parent

• Subtree: A subset of the nodes in a tree that themselves form a tree (possibly with a different root node)

• Interior node: a node that is not a leaf
Tree Vocabulary

- **Descendant**: A node $n'$ is a descendant of node $n$ if there exists a sequence of nodes $n = n_1, n_2, \ldots, n_i = n'$ such that for all $1 \leq j < i$, $n_j$ is a child of $n_{j+1}$. *(Ancestor* is the opposite)
Tree Vocabulary

- **Descendant**: A node $n'$ is a descendant of node $n$ if there exists a sequence of nodes $n = n_1, n_2, \ldots, n_i = n'$ such that for all $1 \leq j < i$, $n_j$ is a child of $n_{j+1}$. *(Ancestor is the opposite)*

- **Siblings**: Two nodes are siblings if they share the same parent
Tree Vocabulary

• **Descendant**: A node $n'$ is a descendant of node $n$ if there exists a sequence of nodes $n = n_1, n_2, \ldots, n_i = n'$ such that for all $1 \leq j < i$, $n_j$ is a child of $n_{j+1}$. (*Ancestor* is the opposite)

• **Siblings**: Two nodes are siblings if they share the same parent

• **Subtree**: A subset of the nodes in a tree that themselves form a tree (possibly with a different root node)
Tree Vocabulary

- **Descendant**: A node \( n' \) is a descendant of node \( n \) if there exists a sequence of nodes \( n = n_1, n_2, \ldots, n_i = n' \) such that for all \( 1 \leq j < i \), \( n_j \) is a child of \( n_{j+1} \). *(Ancestor is the opposite)*

- **Siblings**: Two nodes are siblings if they share the same parent

- **Subtree**: A subset of the nodes in a tree that themselves form a tree (possibly with a different root node)

- **Interior node**: a node that is not a leaf
Tree Vocabulary

- **Path**: the unique shortest sequence of edges between two nodes $n_1$ and $n_2$. Each successive edge in the path must share one of its nodes with the previous edge.
Tree Vocabulary

- **Path**: the unique shortest sequence of edges between two nodes $n_1$ and $n_2$. Each successive edge in the path must share one of its nodes with the previous edge.

- **Full Tree**: A tree where every leaf has the same depth $h$, and every internal node has exactly two children.
Tree Vocabulary

- **Path**: the unique shortest sequence of edges between two nodes $n_1$ and $n_2$. Each successive edge in the path must share one of its nodes with the previous edge.

- **Full Tree**: A tree where every leaf has the same depth $h$, and every internal node has exactly two children.

- **Complete Tree**: A full tree with $\emptyset$ or more of the rightmost leaves of depth $h$ removed.
Binary Search Tree Analysis

- How much time does a call to `locate()` take?

  - **Worst case**
    - Definitely not worse than $O(n)$ (we never look at a node multiple times)
  - Is there a tree where it's actually $O(n)$? Yes; let's try to create an example on the board
  - Let's say we have a tree of height $h$. How long does a call to `locate()` take in terms of $h$?
    - Each time we call the method the height of the node increases by one, so $O(h)$
  - If we have time: how can we prove this by induction?
Binary Search Tree Analysis

- How much time does a call to `locate()` take?
  - Worst case
Binary Search Tree Analysis

• How much time does a call to `locate()` take?
  
  • Worst case
  
  • Definitely not worse than $O(n)$ (we never look at a node multiple times)
Binary Search Tree Analysis

• How much time does a call to `locate()` take?
  • Worst case
  • Definitely not worse than $O(n)$ (we never look at a node multiple times)
  • Is there a tree where it’s actually $O(n)$? Yes; let’s try to create an example on the board
Binary Search Tree Analysis

- How much time does a call to `locate()` take?
  - Worst case
  - Definitely not worse than $O(n)$ (we never look at a node multiple times)
  - Is there a tree where it’s actually $O(n)$? Yes; let’s try to create an example on the board

- Let’s say we have a tree of height $h$. How long does a call to `locate()` take in terms of $h$?
Binary Search Tree Analysis

• How much time does a call to `locate()` take?

  • Worst case
    • Definitely not worse than $O(n)$ (we never look at a node multiple times)
    • Is there a tree where it’s actually $O(n)$? Yes; let’s try to create an example on the board

• Let’s say we have a tree of height $h$. How long does a call to `locate()` take in terms of $h$?
  
    • Each time we call the method the height of the node increases by one, so $O(h)$
Binary Search Tree Analysis

- How much time does a call to `locate()` take?
  - Worst case
  - Definitely not worse than $O(n)$ (we never look at a node multiple times)
  - Is there a tree where it’s actually $O(n)$? Yes; let’s try to create an example on the board

- Let’s say we have a tree of height $h$. How long does a call to `locate()` take in terms of $h$?
  - Each time we call the method the height of the node increases by one, so $O(h)$
  - If we have time: how can we prove this by induction?
Binary Search Tree Analysis

- How much time does a call to `add()` take?
Binary Search Tree Analysis

- How much time does a call to `add()` take?
  - $O(n)$ in a tree of size $n$
Binary Search Tree Analysis

- How much time does a call to `add()` take?
  - $O(n)$ in a tree of size $n$
  - $O(h)$ in a tree of height $h$
Tree Discussion

- How many nodes are in a full tree of depth $h$?
Tree Discussion

- How many nodes are in a full tree of depth $h$?

- How can we sort using a Binary Search Tree?
Tree Discussion

- How many nodes are in a full tree of depth $h$?

- How can we sort using a Binary Search Tree?

- How much time does this take?
Making Binary Search Trees More Efficient

• Goal: ensure that our BST has small height
Making Binary Search Trees More Efficient

- Goal: ensure that our BST has small height

- What should our goal be for height?
Making Binary Search Trees More Efficient

• Goal: ensure that our BST has small height

• What should our goal be for height?

• Complete trees are optimal; what is their height?
Making Binary Search Trees More Efficient

- Goal: ensure that our BST has small height

- What should our goal be for height?

- Complete trees are optimal; what is their height?

- $O(\log n)$
Making Binary Search Trees More Efficient

- Goal: ensure that our BST has small height

- What should our goal be for height?

- Complete trees are optimal; what is their height?

- $O(\log n)$

- Can we design our Binary Search Tree so that it maintains height $O(\log n)$?