Binary Search Trees

BST – A binary tree where a parent's value is greater than all values in the left subtree and less than or equal to all the values in the right subtree

\[ \text{leftTree}(i) < i \leq \text{rightTree}(i) \]

and the left and right children are also binary search trees

Why not?

\[ \text{leftTree}(i) \leq i \leq \text{rightTree}(i) \]

Ambiguous about where elements that are equal would reside

Example

Can be implemented with with references or an array
What else can we conclude?

\[ \text{leftTree}(i) \leq i \leq \text{rightTree}(i) \]

- The smallest element is the leftmost element
- The largest element is the rightmost element

Another example: the solo tree

Another example: the twig

Operations

- \( \text{Search}(T,k) \) – Does value \( k \) exist in tree \( T \)
- \( \text{Insert}(T,k) \) – Insert value \( k \) into tree \( T \)
- \( \text{Delete}(T,x) \) – Delete node \( x \) from tree \( T \)
- \( \text{Minimum}(T) \) – What is the smallest value in the tree?
- \( \text{Maximum}(T) \) – What is the largest value in the tree?
- \( \text{Successor}(T,x) \) – What is the next element in sorted order after \( x \)
- \( \text{Predecessor}(T,x) \) – What is the previous element in sorted order of \( x \)
- \( \text{Median}(T) \) – return the median of the values in tree \( T \)
How do we find an element?

BSTSearch(x, k)
1 if x = null or k = x
2 return x
3 else if k < x
4 return BSTSearch(Left(x), k)
5 else
6 return BSTSearch(Right(x), k)
Finding an element

Search(T, 13)

12

8

14

5

9

20

BSTSearch(x, k)
1. if x = null or k = x
2. return x
3. else if k < x
4. x ← LEFT(x)
5. else
6. x ← RIGHT(x)

Running time of BSTSearch

Worst case?
- O(height of the tree)

Average case?
- O(height of the tree)

Best case?
- O(1)
Height of the tree

Worst case height?
- n-1
- "the twig"

Best case height?
- \( \log n \)
- complete (or near complete) binary tree

Average case height?
- Depends on two things:
  - the data
  - how we build the tree!

Insertion

Search and then insert when you find a "null" spot in the tree

Insert(T, 14)

Inserting duplicates

\( \text{leftTree}(i) < i \leq \text{rightTree}(i) \)
Inserting duplicates

Insert(T, 14)

float T(i) = i ≤ rightTree(i)

Running time

Search and then insert when you find a "null" spot in the tree

O(height of the tree)

Why not Θ(height of the tree)?

Running time

Insert(T, 15)
Height of the tree

Worst case: “the twig” – When will this happen?

Search and then insert when you find a “null” spot in the tree

Best case: “complete” – When will this happen?

Search and then insert when you find a “null” spot in the tree

Average case for random data?

Search and then insert when you find a “null” spot in the tree

Randomly inserting data into a BST generates a tree on average that is $O(\log n)$
Running time of min/max?

BSTMin(x)
1. if Left(x) = null
2. return x
3. else
4. return BSTMin(Left(x))

$O(\text{height of the tree})$

Successor and predecessor

Predecessor(12)? 9

Predecessor in general?
- largest node of all those smaller than this node
- rightmost element of the left subtree

Successor(12)? 13
Successor in general?

- Smallest node of all those larger than this node.
- Leftmost element of the right subtree.

What if the node doesn't have a right subtree?

- Smallest node of all those larger than this node.
- Leftmost element of the right subtree.

What if the node doesn't have a right subtree?

- Node is the largest.
- The successor is the node that has x as a predecessor.

Successor is the node that has x as a predecessor.
successor is the node that has x as a predecessor

keep going up until we're no longer a right child

Successor

Successor

Successor

successor is the node that has x as a predecessor

Successor

Successor

Successor

successor is the node that has x as a predecessor

Successor

Successor

successor is the node that has x as a predecessor

Successor

Successor
**Successor**

If we have a right subtree, return the smallest of the right subtree.

```java
if Right(x) ≠ null
    return BSTMin(Right(x))
else
    y ← Parent(x)
    while y ≠ null and x = Right(y)
        x ← y
        y ← Parent(y)
return y
```

**Successor running time**

$O(\text{height of the tree})$

**Deletion**

Find the node that x is the predecessor of. Keep going up until we're no longer a right child.

```java
if Right(x) ≠ null
    return BSTMin(Right(x))
else
    y ← Parent(x)
    while y ≠ null and x = Right(y)
        x ← y
        y ← Parent(y)
return y
```

Three cases!
Deletion: case 1

No children

Just delete the node

Deletion: case 1

No children

Just delete the node

Deletion: case 2

One child

Splice out the node

Deletion: case 2

One child

Splice out the node
Deletion: case 3

Two children

Replace x with its successor

Height of the tree

Most of the operations take time
$O(\text{height of the tree})$

We said trees built from random data have height $O(\log n)$, which is asymptotically tight

Two problems:
- We can't always insure random data
- What happens when we delete nodes and insert others after building a tree?
Balanced trees

Make sure that the trees remain balanced!
- Red-black trees
- AVL trees
- 2-3-4 trees
- ...

B-trees

Red-black trees: BST (plus some)

1. every node is either red or black
2. root is black
3. leaves (NIL) are black
4. if a node is red, both children are black
5. for every node, all paths from the node to descendant leaves contain the same number of black nodes.

$h(x)$: height of node $x$; number of edges in longest path from $x$ to a leaf
Red-black trees: BST (plus some)

- $h(x)$: height of node $x$: number of edges in longest path from $x$ to a leaf
- $bh(x)$: black height of node $x$: number of black nodes on a path from $x$ to leaf (not including $x$)

1. Every node is either red or black
2. Root is black
3. Leaves (NIL) are black
4. If a node is red, both children are black
5. For every node, all paths from the node to descendant leaves contain the same number of black nodes.

Why don’t we say "path with the most..."?

What is the black height of the root node?
Red-black trees: BST (plus some)

Claim 1: For every node \( x \), \( bh(x) \geq h(x)/2 \)

- every node is either red or black
- root is black
- leaves (NIL) are black
- if a node is red, both children are black
- for every node, all paths from the node to descendant leaves contain the same number of black nodes.

Proof?

Bounding the height

- \( h(x) \): height of node \( x \)
- \( h(x) \): number of edges in longest path from \( x \) to a leaf
- \( bh(x) \): black height of node \( x \)
- \( bh(x) \): number of black nodes on a path from \( x \) to a leaf (not including \( x \))

Claim 1: For every node \( x \), \( bh(x) \geq h(x)/2 \)

Worst case: nodes alternate red/black
- root is black
- leaf is black

In terms of \( h(x) \): How many black nodes are there on this path?
Bounding the height

Claim 1: For every node $x$, $bh(x) \geq h(x)/2$

- Worst case: nodes alternate red/black
  - root is black
  - leaf is black

  $bh(x) = h(x)/2$

  [Note that $bh$ does not include the root]

Worst case: nodes alternate red/black
- root is black
- leaf is black

We can remove red nodes, but that would decrease $h(x)$

Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Proof?

Structural induction

Want to prove something about a recursive structure (e.g., a tree)
Structural induction

Proof by induction:
IH: Assume the property holds for sub-structures (i.e., subtrees)
Show that it holds for the entire tree

Bounding the height

Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Base case:

<table>
<thead>
<tr>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural induction</td>
<td>Structural induction</td>
</tr>
<tr>
<td>Proof by induction:</td>
<td>Base case is often the smallest structure possible (e.g., a leaf)</td>
</tr>
<tr>
<td>IH: Assume the property holds for sub-structures (i.e., subtrees)</td>
<td></td>
</tr>
<tr>
<td>Show that it holds for the entire tree</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>99</td>
</tr>
<tr>
<td>Bounding the height</td>
<td>Bounding the height</td>
</tr>
<tr>
<td>Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes</td>
<td></td>
</tr>
<tr>
<td>Base case: leaf ($h(x) = 0$)</td>
<td></td>
</tr>
<tr>
<td>$bh(x) = 0$</td>
<td>$bh(x)$: black height of node $x$; number of black nodes on a path from $x$ to leaf (not including $x$)</td>
</tr>
<tr>
<td>$2^0 - 1 = 0$</td>
<td></td>
</tr>
</tbody>
</table>
Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Inductive case: $h(x) > 0$

IH: Assume $2^{bh(y)} - 1$ for all $y$ that are subtrees of $x$

What is $bh(child(x))$ wrt $bh(x)$?

$x$ is red: $bh(child(x)) = bh(x) - 1$

$bh(x)$: black height of node $x$; number of black nodes on a path from $x$ to leaf (not including $x$)
Bounding the height

Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Inductive case: $h(x) > 0$

IH: Assume $2^{bh(y)} - 1$ for all $y$ that are subtrees of $x$

- $x$ is red: $bh(child(x)) = bh(x) - 1$
- $x$ is black: $bh(child(x)) = bh(x)$ or $bh(x) - 1$

$bh(child(x)) \geq bh(x) - 1$
Bounding the height

Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Inductive case: $h(x) > 0$

IH: Assume $2^{bh(y)} - 1$ for all $y$ that are subtrees of $x$

$bh(child(x)) \geq bh(x) - 1$

$n = 2^{bh(x)} - 1$

$n \geq 2^{bh(x)} - 1$

$n + 1 \geq 2^{bh(x)}$

$h(x) \leq 2\log(n + 1)$

Claim 1: For every node $x$, $bh(x) \leq \frac{h(x)}{2}$

Claim 2: The subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal (non-leaf) nodes

Inductive case: $h(x) > 0$

IH: Assume $2^{bh(y)} - 1$ for all $y$ that are subtrees of $x$

$bh(child(x)) \geq bh(x) - 1$

$(2^{bh(x)} - 1) + (2^{bh(x)} - 1) + 1 = 2^{bh(x)} - 1$

How does this help us?

What does this mean?
Bounding the height

1. every node is either red or black
2. root is black
3. leaves (NIL) are black
4. if a node is red, both children are black
5. for every node, all paths from the node to descendant leaves contain the same number of black nodes.

If we can maintain these properties: height $O(\log n)$

Search
Insert
Delete
Maximum

Can it be done?

Can we maintain the red-black tree properties without making insertion and deletion more expensive?

![Tree Rotation Image](https://en.wikipedia.org/wiki/Tree_rotation#/media/File:Tree_rotation.png)

A quick example

[Video Link](https://www.youtube.com/watch?v=vDHFF4wjWYU)

Number guessing game

I'm thinking of a number between 1 and n

You are trying to guess the answer

For each guess, I'll tell you “correct”, “higher” or “lower”

Describe an algorithm that minimizes the number of guesses