CSCI 104
Binary Search Trees and AVL Trees
Mark Redekopp
David Kempe
Properties, Insertion and Removal

BINARY SEARCH TREES
Binary Search Tree

- Binary search tree = binary tree where all nodes meet the property that:
  - All values of nodes in left subtree are less-than or equal than the parent’s value
  - All values of nodes in right subtree are greater-than or equal than the parent’s value

If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?
BST Insertion

• Important: To be efficient (useful) we need to keep the binary search tree balanced

• Practice: Build a BST from the data values below
  – To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value

Insertion Order: 25, 18, 47, 7, 20, 32, 56

Insertion Order: 7, 18, 20, 25, 32, 47, 56
BST Insertion

• Important: To be efficient (useful) we need to keep the binary search tree balanced

• Practice: Build a BST from the data values below
  – To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value

• https://www.cs.usfca.edu/~galles/visualization/BST.html

A major topic we will talk about is algorithms to keep a BST balanced as we do insertions/removals
Successors & Predecessors

• Let's take a quick tangent that will help us understand how to do **BST Removal**

• Given a node in a BST
  – Its predecessor is defined as the next smallest value in the tree
  – Its successor is defined as the next biggest value in the tree

• Where would you expect to find a node's successor?
• Where would find a node's predecessor?
Predecessors

• If left child exists, predecessor is the right most node of the left subtree

• Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent...that parent is the predecessor)
  – If you get to the root w/o finding a node who is a right child, there is no predecessor
Predecessors

• If left child exists, predecessor is the right most node of the left subtree

• Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent...that parent is the predecessor)
  – If you get to the root w/o finding a node who is a right child, there is no predecessor
Successors

- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent...that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor
Successors

- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent...that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor

Succ(20) = 25
Succ(30) = 50
BST Removal

• To remove a value from a BST...
  – First find the value to remove by walking the tree
  – If the value is in a leaf node, simply remove that leaf node
  – If the value is in a non-leaf node, swap the value with its in-order successor or predecessor and then remove the value
    • A non-leaf node's successor or predecessor is guaranteed to be a leaf node (which we can remove) or have 1 child which can be promoted
    • We can maintain the BST properties by putting a value's successor or predecessor in its place
Worst Case BST Efficiency

• Insertion
  – Balanced: __________
  – Unbalanced: __________

• Removal
  – Balanced: __________
  – Unbalanced: __________

• Find/Search
  – Balanced: __________
  – Unbalanced: __________

```c++
#include<iostream>
using namespace std;

// Bin. Search Tree
template <typename T>
class BST
{
  public:
    BTree();
    ~BTree();
    virtual bool empty() = 0;
    virtual void insert(const T& v) = 0;
    virtual void remove(const T& v) = 0;
    virtual T* find(const T& v) = 0;
};
```
BST Efficiency

• Insertion
  – Balanced: $O(\log n)$
  – Unbalanced: $O(n)$

• Removal
  – Balanced : $O(\log n)$
  – Unbalanced: $O(n)$

• Find/Search
  – Balanced : $O(\log n)$
  – Unbalanced: $O(n)$

```cpp
#include <iostream>
using namespace std;

// Bin. Search Tree
template <typename T>
class BST
{
  public:
    BTree();
    ~BTree();
    virtual bool empty() = 0;
    virtual void insert(const T& v) = 0;
    virtual void remove(const T& v) = 0;
    virtual T* find(const T& v) = 0;
};
```
Trees & Maps/Sets

- C++ STL "maps" and "sets" use binary search trees internally to store their keys (and values) that can grow or contract as needed
- This allows $O(\log n)$ time to find/check membership
  - BUT ONLY if we keep the tree balanced!

Map::find("Greg")
Returns iterator to corresponding pair<string, Student>

Map::find("Mark")
Returns iterator to end() [i.e. NULL]
The key to balancing...

TREE ROTATIONS
BST Subtree Ranges
• Consider a binary search tree, what range of values could be in the subtree rooted at each node
  – At the root, any value could be in the "subtree"
  – At the first left child?
  – At the first right child?

What values might be in the subtree rooted here

(-inf, inf)

(-inf, inf)

What values might be in the subtree rooted here

(-inf, inf)

(-inf, inf)
BST Subtree Ranges

• Consider a binary search tree, what range of values could be in the subtree rooted at each node
  – At the root, any value could be in the "subtree"
  – At the first left child?
  – At the first right child?
Right Rotation

• Define a right rotation as taking a left child, making it the parent and making the original parent the new right child

• Where do subtrees a, b, c and d belong?
  – Use their ranges to reason about it...

![Diagram of a right rotation]

- Subtrees a, b, c and d are positioned as follows:
  - a: (-inf, x)
  - b: (x, y)
  - c: (-inf, y)
  - d: (y, z)

- The range of z is (-inf, inf)

- The range of y is (z, inf)

- The range of x is (-inf, z)

- The range of z is (inf, z)

- Right rotate of z
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child

- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it...

```
\( z \)
\( \text{(-inf, inf)} \)
\( y \)
\( \text{(-inf, z)} \)
\( d \)
\( \text{(z, inf)} \)
\( x \)
\( \text{(-inf, y)} \)
\( c \)
\( \text{(y,z)} \)
\( \text{(-inf, x)} \)
\( a \)
\( \text{(x,y)} \)
\( b \)
```

```
\( y \)
\( \text{(z, inf)} \)
\( \text{Right rotate of z} \)
\( x \)
\( \text{(-inf, x)} \)
\( a \)
\( \text{(x,y)} \)
\( b \)
\( \text{(y,z)} \)
\( c \)
\( \text{(z, inf)} \)
\( d \)
```
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child.

- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it...

```
- Left rotate of x

(y)

(x)  

(z)  

(x, y)

(y, z)

(z, inf)

(-inf, x)

(-inf, x)

(x, inf)

(-inf, inf)

(a)

(b)

(c)

(d)
```
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child.

- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it...

```
  y
 /   \
x     z
 a    b
(-inf, x) (x, y)
 c    d
(y, z) (z, inf)
```

```
  x
 (-inf, x)
  a
 (x, inf)
 b
(x, y)
(y, inf)
```

```
  y
  (x, y)
  (y, inf)
  b
(x, inf)
```

```
  z
 (y, z)
  (z, inf)
  c
 d
```
Rotations

• Define a right rotation as taking a left child, making it the parent and making the original parent the new right child

• Where do subtrees a, b, and c belong?
  – Use their ranges to reason about it...

(-inf, inf)

(-inf, y)

(-inf, x)

(-inf, x)

(-inf, x)

(y, inf)

(x, inf)

(x, y)

(x, y)

(x, inf)

(y, inf)

(a, b)

(x, (y, inf))

(a, x)

(y, (inf, x))

(b, c)

Right rotate of y

Left rotate of x
Rotation's Effect on Height

• When we rotate, it serves to re-balance the tree

Let's always specify the parent node involved in a rotation (i.e. the node that is going to move DOWN).
Self-balancing tree proposed by Adelson-Velsky and Landis

AVL TREES
AVL Trees

- A binary search tree where the **height difference** between left and right subtrees of a node is **at most 1**
  - Binary Search Tree (BST): Left subtree keys are less than the root and right subtree keys are greater

- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - Balance: Define $b(n)$ as the balance of a node = Right – Left Subtree Height
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - *Let's use balance for this lecture.*

#### AVL Tree storing Heights

![AVL Tree storing Heights](image)

#### AVL Tree storing balances

![AVL Tree storing balances](image)
Adding a New Node

• Once a new node is added, can its parent be out of balance?
  – No, assuming the tree is "in-balance" when we start.
  – Thus, our parent has to have
    • A balance of 0
    • A balance of 1 if we are a new left child or -1 if a new right child
  – Otherwise it would not be our parent or the parent would have been out of balance already
Losing Balance

• If our parent is not out of balance, is it possible our grandparent is out of balance?
• Sure, so we need a way to re-balance it
To Zig or Zag

• The rotation(s) required to balance a tree is/are dependent on the grandparent, parent, child relationships
• We can refer to these as the zig-zig case and zig-zag case
• Zig-zig requires 1 rotation
• Zig-zag requires 2 rotations (first converts to zig-zig)
Disclaimer

• There are many ways to structure an implementation of an AVL tree...the following slides represent just 1
  – Focus on the bigger picture ideas as that will allow you to more easily understand other implementations
Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p, n)
Insert-fix(p, n)

- Precondition: p and n are balanced: {-1,0,-1}
- Postcondition: g, p, and n are balanced: {-1,0,-1}
- If p is null or parent(p) is null, return
- Let g = parent(p)
- Assume p is left child of g [For right child swap left/right, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - if g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!
Insertion
• Insert 10, 20, 30, 15, 25, 12, 5, 3, 8
Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

Insert 5

Insert 3

Zig-zig =>
\[ b(g) = b(p) = 0 \]

Insert 8

Zig-zag & \[ b(n) = -1 \] =>
\[ b(g) = 1 \] & \[ b(p) = b(n) = 0 \]
Insertion Exercise 1

- Insert key=28

```
-1 20
  0 10
    0 5
      0 3
    0 12
      0 8
      0 25
    0 15

-1 30
```
Insertion Exercise 2

- Insert key=17

```
  20
 /   \
10    30
 / \
5   12 25
 / \
3   8  15
```


Insertion Exercise 3

• Insert key=2
Remove Operation

• Remove operations may also require rebalancing via rotations
• The key idea is to update the balance of the nodes on the ancestor pathway
• If an ancestor gets out of balance then perform rotations to rebalance
  – Unlike insert, performing rotations does not mean you are done, but need to continue
• There are slightly more cases to worry about but not too many more
Remove

- Let n = node to remove (perform BST find)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- Let p = parent(n)
- If n is not in the root position (i.e. p is not NULL) determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and "patch" the tree
- removeFix(p, diff);
RemoveFix(n, diff)

• If n is null, return

• Compute next recursive call's arguments now before we alter the tree
  – Let p = parent(n) and if p is not NULL let ndiff = +1 if n is a left child and -1 otherwise

• Assume diff = -1 and follow the remainder of this approach, mirroring if diff = +1

• If (balance(n) + diff == -2)
  – [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  – Let c = left(n), the taller of the children
  – If balance(c) == -1 or 0 (zig-zig case)
    • rotateRight(n)
    • if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    • if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  – else if balance(c) == 1 (zig-zag case)
    • Let g = right(c)
    • rotateLeft(c) then rotateRight(n)
    • If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    • If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    • If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    • removeFix(p, ndiff);

• else if (balance(n) + diff == -1) then balance(n) = -1, done!
• else (if balance(n) + diff == 0) balance(n) = 0, removeFix(p, ndiff)
Remove Examples

Remove 15

Remove 3
Remove Examples

Remove 25

Zig-zig & b(c) = -1 => b(n) = b(c) = 0
Remove Examples

Remove 20

Zig-zag & b(g) = -1 =>
b(n) = +1, b(c) = 0, b(g) = 0
Remove Example 1

Remove 8
Remove Example 1

Remove 8

Zig-zag & b(g) = 0 =>
b(n) = -1, b(c) = 0
Remove Example 2

Remove 10

```
-1 20

-1 10 -1 30

-1 8 -1 15 1 25 0 35

0 5 -1 12 0 17 0 28

0 11
```
Remove Example 2

Remove 10
Remove Example 3

Remove 30

Remove 30
else if \( b(c) == 1 \) (zig-zag case)

- rotateLeft(c) then rotateRight(n)
- Let \( g = \text{right}(c) \), \( b(g) = 0 \)
- If \( b(g) == +1 \) then \( b(n) = 0 \), \( b(c) = -1 \), \( b(g) = 0 \)
- If \( b(g) == 0 \) then \( b(n) = b(c) = 0 \), \( b(g) = 0 \)
- If \( b(g) == -1 \) then \( b(n) = +1 \), \( b(c) = 0 \), \( b(g) = 0 \)
- removeFix(parent(p), ndiff);
Remove Example 3 (cont)

Remove 30 (cont.)

```
else if b(c) == 1  (zig-zag case)
  • rotateLeft(c) then rotateRight(n)
  • Let g = right(c), b(g) = 0
  • If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
  • If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
  • If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
  • removeFix(parent(p), ndiff);
```
Online Tool

Distribute these 4 to students

FOR PRINT
Insert(n)

- If empty tree => set as root, \( b(n) = 0 \), done!
- Insert \( n \) (by walking the tree to a leaf, \( p \), and inserting the new node as its child), set balance to 0, and look at its parent, \( p \)
  - If \( b(p) = -1 \), then \( b(p) = 0 \). Done!
  - If \( b(p) = +1 \), then \( b(p) = 0 \). Done!
  - If \( b(p) = 0 \), then update \( b(p) \) and call insert-fix(\( p, n \))
Insert-fix(p, n)

• Precondition: p and n are balanced: {-1,0,-1}
• Postcondition: g, p, and n are balanced: {-1,0,-1}
• If p is null or parent(p) is null, return
• Let g = parent(p)
• Assume p is left child of g [For right child swap left/right, +/-]
  – g.balance += -1
  – if g.balance == 0, return
  – if g.balance == -1, insertFix(g, p)
  – If g.balance == -2
    • If zig-zig then rotateRight(g); p.balance = g.balance = 0
    • If zig-zag then rotateLeft(p); rotateRight(g);
      – if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      – if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      – if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!
Remove

- Let $n = \text{node to remove}$ (perform BST find) and $p = \text{parent}(n)$
- If $n$ has 2 children, swap positions with in-order successor and perform the next step
  - Now $n$ has 0 or 1 child guaranteed
- If $n$ is not in the root position determine its relationship with its parent
  - If $n$ is a left child, let $\text{diff} = +1$
  - if $n$ is a right child, let $\text{diff} = -1$
- Delete $n$ and update tree, including the root if necessary
- $\text{removeFix}(p, \text{diff})$;
RemoveFix(n, diff)

• If n is null, return
• Let ndiff = +1 if n is a left child and -1 otherwise
• Let p = parent(n). Use this value of p when you recurse.
• If balance of n would be -2 (i.e. balance(n) + diff == -2)
  – [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  – Let c = left(n), the taller of the children
  – If balance(c) == -1 or 0 (zig-zig case)
    • rotateRight(n)
    • if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    • if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  – else if balance(c) == 1 (zig-zag case)
    • rotateLeft(c) then rotateRight(n)
    • Let g = right(c)
    • If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    • If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    • If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    • removeFix(\text{parent}(p), ndiff);
  • else if balance(n) == 0 then balance(n) += diff, done!
  • else balance(n) = 0, removeFix(p, ndiff)
OLD ALTERNATE METHOD
Insert

• Root => set balance, done!
• Insert, v, and look at its parent, p
  – If b(p) = -1, then b(p) = 0. Done!
  – If b(p) = +1, then b(p) = 0. Done!
  – If b(p) = 0, then update b(p) and call insert-fix(p)
Insert-Fix

• For input node, v
  – If v is root, done.
  – Invariant: \( b(v) = \{-1, +1\} \)

• Find \( p = \text{parent}(v) \) and assume \( v = \text{left}(p) \) [i.e. left child]
  – If \( b(p) = 1 \), then \( b(p) = 0 \). Done!
  – If \( b(p) = 0 \), then \( b(p) = -1 \). Insert-fix(p).
  – If \( b(p) = -1 \) and \( b(v) = -1 \) (zig-zig), then \( b(p) = 0 \), \( b(v) = 0 \), rightRotate(p) Done!
  – If \( b(p) = -1 \) and \( b(v) = 1 \) (zig-zag), then
    • \( u = \text{right}(v) \), \( b(u) = 0 \), leftRotate(n), rightRotate(p)
    • If \( b(u) = -1 \), then \( b(v) = 0 \), \( b(p) = 1 \)
    • If \( b(u) = 1 \), then \( b(v) = -1 \), \( b(p) = 0 \)
    • Done!