Binary Search Trees

Useful for dynamic sets
Operations: Search, Min, Max, Predecessor, Successor, Insert, Delete

**Performance of Some of These Operations**

\( \Theta(h) \), where \( h \) = height of tree

\( \Theta(lgn) \), is \__________, where \( n \) is number of nodes in tree, this is true in the case of a full binary tree

\( \Theta(n) \) is \__________, this is true in the case of a linear chain

---

**Binary Search Tree Property**

\[ n = \text{node in BST} \]
\[ l = \text{node in left subtree of } n \]
\[ r = \text{node in right subtree of } n \]

For a binary search tree, \( \text{key}[l] \ _____ \text{key}[n] _____ \text{key}[r] \). Different from a heap, in which the left-right ordering of values does not matter.
Example

BST Traversals

Inorder(x): visit left(x), then x, then right(x)
   PreOrder(x): visit x, then left(x), then right(x)
   PostOrder(x): visit left(x), then right(x), then x

InOrder

InOrder(x) ; prints elements in sorted order
   if x <> NIL
      then InOrder(left(x))
         print(key(x))
         InOrder(right(x))

InOrder traversal order: 1 2 4 5 8 10
PreOrder

PreOrder(x)
if x <> NIL
then print(key(x))
    PreOrder(left(x))
    PreOrder(right(x))

PreOrder traversal order: 5 2 1 4 10 8

PostOrder

PostOrder(x)
if x <> NIL
then PostOrder(left(x))
    PostOrder(right(x))
    print(key(x))

PostOrder traversal order: 1 4 2 8 10 5

Analysis

\[ T(n) = \begin{cases} 
  \Theta(1) & n = 0 \\
  T(k) + T(n - k - 1) + \Theta(1) & n > 0 
\end{cases} \]

\[ k = (n-1)/2: \]
\[ T(n) = T((n-1)/2) + T(n - (n-1)/2 - 1) + \Theta(1) \]
\[ = T((n-1)/2) + T(n/2 + 1/2 - 1) + \Theta(1) \]
\[ = T((n-1)/2) + T((n-1)/2) + \Theta(1) \]
= 2T((n-1)/2) + \Theta(1) \\
\leq 2T(n/2) + \Theta(1) \\
= \Theta(n)

k = 0: \\
T(n) = T(0) + T(n-1) + \Theta(1) \\
= T(n-1) + \Theta(1) \\
= \Theta(n)

---

**Searching**

Search(n, k) ; n is a pointer to a node, not the tree

  if n=NIL or k=key(n) ; Initially n points to the root node  
  then return n  
  if k < key(n)  
  then return Search(left(n), k)  
  else return Search(right(n), k)

---

**Example: **Search(Root(T), 14)

---

**Analysis**

Could write this code iteratively:

  if k < key(n)  
  then n = left(n)  
  else n = right(n)  

LOOP
• This code is clearly \( \Theta(h) \).
• Not necessarily \( \Theta(lgn) \).
• Remember, we are not assured that the tree is balanced

\[
\text{Min}(n) \quad ; \quad \text{leftmost leaf of tree rooted at } n \\
\text{while left}(n) <> \text{NIL} \\
\quad n = \text{left}(n) \\
\quad \text{return } n \\
\]

Min is ______

\[
\text{Max}(n) \quad ; \quad \text{rightmost leaf of tree rooted at } n \\
\text{while right}(n) <> \text{NIL} \\
\quad n = \text{right}(n) \\
\quad \text{return } n \\
\]

Max is ______

\[
\text{Successor}(n) \\
\]

Case I:
Try \( n=4 \)
right(\( n \)) \( \neq \) NIL
return(\( \text{Min}(\text{right}(n)) \))
Try n=7
Try n=5

**Successor(n)**

right(n) = NIL
Case II: Find s such that left(s) is n or an ancestor of n

Try n=5

Case III: No such ancestor exists; thus, no successor
Try n=7
Successor(n)

Successor(n)
    if right(n) <> NIL
    then return Min(right(n))
    p = Parent(n)
    while p <> NIL and n = right(p)
        n = p
        p = Parent(p)
    return(p)

Successor is _____

Predecessor(n)

Predecessor(n)
    if left(n) <> NIL
    then return Max(left(n))
    p = Parent(n)
    while p <> NIL and n = left(p)
        n = p
        p = Parent(p)
    return(p)

Predecessor is _____

Insertion

1. Go search for key until run off the end of the tree
2. Put new key there
Insert(T, x) ; x = pointer to new node
    p = NIL
    n = root(T) ; Search for key
    while n <> NIL
        p = n
        if key(x) < key(n)
            then n = left(n)
        else n = right(n)
        Parent(x) = p ; Insert new key
    if p = NIL
        then root(T) = x
    else if key(x) < key(p)
    then left(p) = x
    else right(p) = x

Insert is ______

Examples

Insert(T, 6)
For a sequence of integers in the range 1...10, which insertion order will yield the tallest tree? ________________________________

Which insertion order will yield the shortest tree? ________________________________

Delete

Case I: Node is a leaf. ________________________________

Delete

Case II: Node has only one child. ________________________________

Delete

Case III: Node has two children. Delete __________ of node and replace node’s key with successor’s key. Note that successor always satisfies ________________________________

Pseudocode

```
Delete(T,x)
    if left(x) = NIL or right(x) = NIL
        then d = x
        else d = Successor(x)
        if left(d) <> NIL
            then c = left(d)
    ; Returns deleted node

    ; Case I and II
    ; Case III
    ; Get child of node to be deleted
```
else c = right(d)
if c <> NIL
then Parent(c) = Parent(d) ; Remove node d
if Parent(d) = NIL
then root(T) = c
else if d = left(parent(d))
    then left(parent(d)) = c
    else right(parent(d)) = c
if d <> x ; Case III
then key(x) = key(d)
return(d)

Randomly Built BSTs

Theorem 13.6: The average height of a randomly built BST on n distinct keys is _________ By randomly built we mean that keys are inserted in random order.

Red-Black Trees

Binary Search Tree operations are ______ not ____________________
If we can keep the tree balanced, the operations will be _______
This is the goal of Red-Black trees.
- Internal Node

NIL children are leaves

• Internal Node

NIL children are leaves

• Internal Node

NIL children are leaves

- Leaf Node: NIL

Leaf nodes are sentinels (dummy objects), and the color is always Black.

By constraining colors of nodes along paths from root to leaf, RB trees ensure no path is more than twice as long as any other (the tree is balanced).

Properties of RB Trees

1. Every node is either ____ or ________.

2. Every leaf is ______.

3. If a node is Red, then both its children are ______.

4. Every path from some node to a leaf contains the ______________ of Black nodes.
Properties of RB Trees

**Definition:** The ________ of a node n, denoted bh(n), is the number of black nodes (excluding n) on the path from n to a leaf, including the leaf.

\[
bh(\text{root}) = \text{Black Height of the tree}
\]

By property 4, bh(n) is the same regardless of the path.

---

**Lemma 14.1**

A RB tree with n internal nodes has height at most ________.

Thus the dynamic set operations on RB trees are all ________.

**Proof:**

1. First show that subtree rooted at \(x\) contains at least \(2^{bh(x)} - 1\) internal nodes.
Proof by induction.

**Initial condition:** if height(x) = 0, then x is a leaf whose subtree contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

**Inductive Step:** Consider internal node x. Each child has black-height bh(x) (if the child is Red) or bh(x)-1 (if the child is Black).

By the Inductive Hypothesis, the child has at least $2^{bh(x)-1} - 1$ internal nodes.

Therefore the subtree rooted at x has at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$ internal nodes, or $2^{bh(x)} - 1$ internal nodes.

2. Next, by property 3, a tree of height $h$ has a black-height of at least $h/2$.

   \[
   bh(T) \geq h/2 \\
   n \geq 2^{bh(T)} - 1 \\
   n \geq 2^{h/2} - 1, \ n+1 \geq 2^{h/2} \\
   \lg(n+1) \geq \lg(2^{h/2}) \\
   \lg(n+1) \geq h/2 \\
   h \leq 2\lg(n+1)
   \]

---

**Properties of RB Trees**

As you can tell, Insert and Delete need some work to maintain the RB tree properties.

**Question:** Is a RB tree with a Red root still a RB tree if we change the root color to Black?

**Answer:** ___
Rotations

Redistribute nodes in the tree.

Rotations

Left-Rotate(T,x)

\[
y = \text{right}(x) \quad ; \text{assume} \; \text{right}(x) \not\to \text{NIL} \\
\text{right}(x) = \text{left}(y) \quad ; \text{move} \; y\'s \; \text{child \; over} \\
\text{if} \; \text{left}(y) \not\to \text{NIL} \\
\text{then} \; \text{parent}(\text{left}(y)) = x \\
\text{parent}(y) = \text{parent}(x) \quad ; \text{move} \; y \; \text{up \; to} \; x\'s \; \text{position} \\
\text{if} \; \text{parent}(x) = \text{NIL} \\
\text{then} \; \text{root}(T) = y \\
\text{else if} \; x = \text{left}(\text{parent}(x)) \\
\quad \text{then} \; \text{left}(\text{parent}(x)) = y \\
\quad \text{else} \; \text{right}(\text{parent}(x)) = y \\
\text{left}(y) = x \quad ; \text{move} \; x \; \text{down} \\
\text{parent}(x) = y
\]

Rotations

Right-Rotate(T,y)

\[
x = \text{left}(y) \quad ; \text{assume} \; \text{left}(y) \not\to \text{NIL} \\
\text{left}(y) = \text{right}(x) \\
\text{if} \; \text{right}(x) \not\to \text{NIL} \\
\text{then} \; \text{parent}(\text{right}(x)) = y
\]
parent(x) = parent(y)
if parent(y) = NIL
then root(T) = x
else if y = left(parent(y))
    then left(parent(y)) = x
    else right(parent(y)) = x
right(x) = y
parent(y) = x

---

Insertion

1. Insert node into tree using BST Insert(T,x) and color node Red
2. Fix violated RBT properties
3. Color root Black

Which properties might be violated?

1. _____, new node is Red; previous nodes are already colored
2. _____, new node inserted with NIL (Black) leaves
3. ____________, parent may also be Red
4. _____, replacing Black node with a Red and Black node
• Tree was balanced before insert
• If colored Black, may violate property 4

RB Trees

If parent node ‘a’ was Black, then no changes are necessary.
If not, then there are three cases to consider for each of the orientations below.

3 cases to consider for each of these

Move up the tree until there are no violations or we are at the root.
In the following discussion we will assume the parent is a left child (if the parent is a right child perform the same steps swapping “right” and “left”)

16
RB-Insert(T,x)

Case I: x’s uncle is Red

if parent(x) = left(parent(parent(x)))
then uncle(x) = right(parent(parent(x)))
ext else uncle(x) = left(parent(parent(x)))

• Change x’s grandparent to Red
• Change x’s uncle and parent to Black
• Change x to x’s grandparent

Case II: x’s uncle is Black, x is the right child of its parent

• Change x to x’s parent
• Rotate x’s parent (now x) left to make Case III
• Case II is now Case III
Click mouse to advance to next frame.

Case III: x’s uncle is Black, x is the left child of its parent

• Set x’s parent to Black
• Set x’s grandparent to Red
• Rotate x’s grandparent right
Click mouse to advance to next frame.
Pseudocode

RB-Insert(T,x)
   Insert(T,x)
   color(x) = Red
   while x <> root(T) and color(parent(x)) = Red
      if parent(x) = left(parent(parent(x)))
         then uncle = right(parent(parent(x)))
            if color(uncle) = Red
               then color(parent(x)) = Black
                     color(uncle) = Black
                     color(parent(parent(x))) = Red
               x = parent(parent(x))
      else if x = right(parent(x))
         then x = parent(x)
            Left-Rotate(T,x)
            color(parent(x)) = Black
            color(parent(parent(x))) = Red
            Right-Rotate(T, parent(parent(x)))
   else
      ... ; same as then with "right" and "left" swapped
   color(root(x)) = Black

The performance of this algorithm is O(lgn), with \leq 2 rotations

Deletion

1. Delete node from tree using an algorithm similar to BST Delete(T,x)
2. Fix violated properties
Which properties might be violated?

- If node deleted was Red, _____
- If node deleted was Black, then property __ will be violated
- Property __ is also violated but is immediately fixed

To correct the violations, look at the violation from another perspective: Assume the child of the deleted node is colored “double-Black”, violating property 1, and we want to give half of the “double-Black” to another Red node or push half of the Black out the top of the tree.

Sort of BST Delete

1. Use of sentinel nil(T) for NIL leaves
2. Call to RB-Delete-Fixup

RB-Delete(R, z) ; return deleted node
    if left(z) = nil(T) or right(z) = nil(T)
        then d = z
    else d = Successor(z)
        if left(d) <> nil(T)
            then c = left(d)
        else c = right(d)
        parent(c) = parent(d) ; no test for NIL with sentinel
        if parent(d) = nil(T)
            then root(T) = c
        else if d = left(parent(d))
            then left(parent(d))) = c
        else right(parent(d))) = c
if d <> z
then key(z) = key(d)
if color(d) = Black
then RB-Delete-Fixup(T,c) ; c is now "Double-Black"
return d

RB-Delete-Fixup(T,x)

First, if color(x) = Red, then color x Black; done!

Note that x always has a sibling s. This is because if x is Black and is
the child of a deleted Black node, then there is a sibling s because the tree
was previously balanced.

There are four cases to consider for each orientation of x (whether x is
a left child or a right child). In each case we need to maintain the number
of Black nodes.

For these examples we will assume that x = left(parent(x)) (x is a left
child).

Case I:

x’s sibling is Red, s has two Black children
• Switch colors of s and parent(x) (color(s) = Black, color(parent(x)) = Red)
• Rotate parent(x) left
• Reset sibling s
Case I is now Case II, Case III, or Case IV

---

**Case II:**

Sibling is black, sibling’s children are both Black

- Change s to Red
- Add extra Black to parent(x)
- Repeat while loop with parent(x)
- If entered Case II from Case I, will then terminate (parent(x) = Red)

---

**Case III:**

x’s sibling is Black, s’s left child is Red, s’s right child is Black

- Switch colors of s (Red) and left(s) (Black)
- Rotate s right
- Reset sibling s
- Case III is now Case IV

---

**Case IV:**

x’s sibling is black, s’s right child is Red

- Change color(s) to color of parent(x)
• Change color of parent to Black
• Change color of sibling’s right child (Red) to Black
• Rotate parent(x) left
• All done!

Pseudocode

RB-Delete-Fixup(T,x)
while x <> root(T) and color(x) = Black
    if x = left(parent(x))
        then s = right(parent(x)) ; Get x’s sibling
            if color(s) = Red
                then color(s) = Black ; Case I
                color(parent(x)) = Red
                Left-Rotate(T, parent(x))
                s = right(parent(x))
            if color(left(s)) = Black and color(right(s)) = Black
                then color(s) = Red ; Case II
                x = parent(x)
            else if color(right(s)) = Black
                then color(left(s)) = Black ; Case III
                color(s) = Red
                Right-Rotate(T,s)
                s = right(parent(x))
                color(s) = color(parent(x)) ; Case IV
        color(parent(x)) = Black
        color(right(s)) = Black
Left-Rotate(T, parent(x))
    x = root(T)
else
    ... ; Same as then with right and left swap
    color(x) = Black

The performance of this algorithm is $O(lgn)$, with $\leq 3$ rotations
Thus we see that RB trees maintain $O(lgn)$ time for dynamic-set operations.

Applications