BALANCED BINARY SEARCH TREE

Think of a balanced binary search tree as a dynamic sorted array. That is, you can perform all the operations as with a sorted array (Extract smallest/largest element, provide elements in sorted order, etc.), but you can also handle dynamic insertions and deletions with the balanced binary search tree data structure.

SORTED ARRAYS: SUPPORTED OPERATIONS

GIVEN SORTED ARRAY: | 3 | 6 | 10 | 11 | 17 | 23 | 30 | 36 |

OPERATION	RUNNING TIME
SEARCH	$O(\lg n)$
SELECT (GIVEN ORDER STATISTIC, i)	$O(1)$
MIN/MAX ELEMENT	$O(1)$
PREDECESSOR/SUCCESSOR (GIVEN POINTER TO A KEY)	$O(1)$
RANK (# OF KEYS LESS THAN OR EQUAL TO A GIVEN VALUE)	$O(\lg n)$
OUTPUT IN SORTED ORDER	$O(n)$
INSERTION/DELETION	$O(n)$

BALANCED SEARCH TREE SUPPORTED OPERATIONS

OPERATION	RUNNING TIME	
SEARCH	$O(\lg n)$	
SELECT	$O(\lg n)$	UP FROM $O(1)$, STILL FAST
MIN/MAX	$O(\lg n)$	
PRED/SUCC	$O(\lg n)$	
RANK	$O(\lg n)$	
OUTPUT IN SORTED ORDER	$O(n)$	
INSERTION/DELETION	$O(\lg n)$	DOWN FROM $O(n)$

(con't on next sheet)

BALANCED BINARY SEARCH TREE (con't)

BBST vs. HEAP DATA STRUCTURE

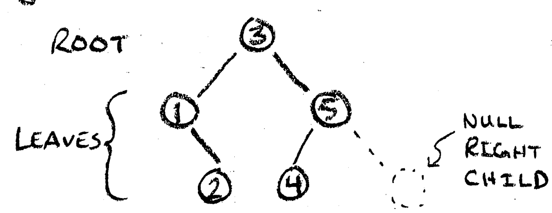
While a Balanced Binary Search Tree supports all the operations a Heap supports, the two are not the same. Think of the Balanced Binary Search Tree as a Swiss Army Knife and the Heap as a filet knife. While you can do everything a filet knife can do with a Swiss Army Knife it will not be as efficient as using the tool optimized for the specific job.

If you don't need any of the other operations a Balanced Binary Search Tree provides other than extracting minimum or maximum values stick with a heap over a Balanced Binary Search Tree. While the asymptotic running time is $O(\lg n)$ for both data structures, heaps have smaller constant factors and will perform better in their specific application. In other words, heaps are designed to find the minimum/maximum value quickly and a binary search tree is designed to search quickly.

BINARY SEARCH TREE BASICS

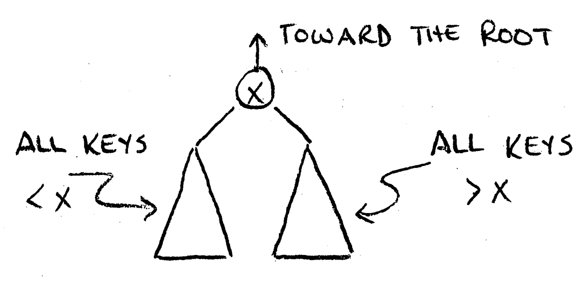
Each node represents one key value. There can be duplicate key values, but each key value gets a node.

Each node has:
- left child pointer
- right child pointer
- parent pointer



SEARCH TREE PROPERTY:

Given an arbitrary node of value x in a binary search tree all keys less than x will reside with the left child branch and all keys greater than x will reside within the right child branch of the tree. This property holds at every node of the search tree.



(con't on next sheet)

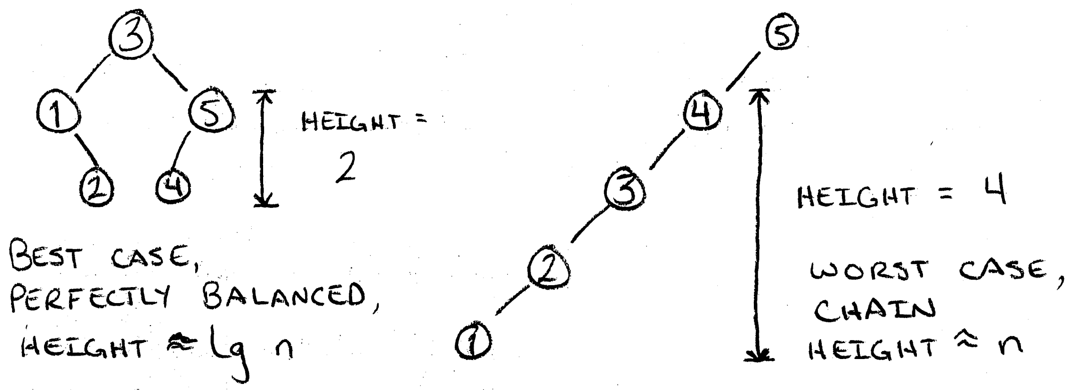
BINARY SEARCH TREE BASICS (con't)

Keys with like values (duplicates) can be implemented by allowing all values less than or equal to x to exist within the left branch of the binary search tree.

HEIGHT OF A BINARY SEARCH TREE

There can be many possible tree layouts for a set of keys. The differing layouts can affect performance.

EXAMPLE



SEARCH A BINARY SEARCH TREE

To search for key, k, in tree, T:

1. Start at the root
2. Traverse left/right child pointers as needed. That is,
   if k < key compared go left
   if k > key compared go right
3. Return node with k or null (k is not in T) as appropriate.

INSERT INTO A BINARY SEARCH TREE

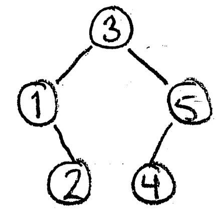
To insert a new key, k, into a tree, T:

1. Search for k in T as above.
2. Replace the final pointer to a null child to a new node with k.

(con't on next sheet)

BINARY SEARCH TREE BASICS (con't)

COMPUTE THE MINIMUM KEY OF A TREE



1. Start at the root
2. Follow left child pointers until you reach a null pointer. Return the last non-null key found.

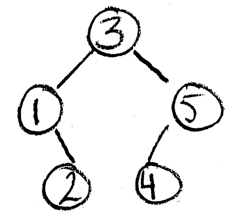
COMPUTE THE MAXIMUM KEY OF A TREE

Same as above except follow right child pointers.

COMPUTE THE PREDECESSOR OF KEY, k

EASY CASE

If k's left subtree is non-empty, return the max key in the left subtree.



For 3, the predecessor is 2, the max value in 3's left subtree

For 5, the predecessor is 4, the max value in 5's left subtree.

OTHERWISE

Follow parent pointers until you get to a key less than k. If you reach the root and still have not found a key less than k, then k has no predecessor in the search tree and is also the minimum key.

This happens the first time you "turn left" as you progress up the binary search tree structure.

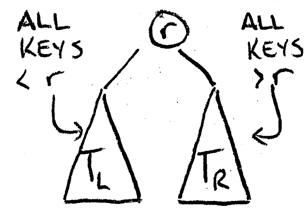
COMPUTE THE SUCCESOR OF KEY, k

Same as computing the predecessor except use right subtree instead of the left subtree.

(con't on next sheet)

Binary Search Tree Basics (con't)

In-Order Traversal (Print Keys in Increasing Order)



1. Let r = root of search tree with subtrees T_L and T_R
2. Recurse on T_L to print out minimum values in increasing order.
3. Print out r's key
4. Recurse on T_R to print out maximum values in increasing order.

Deletion of a Key from the Search Tree

1. Search for k

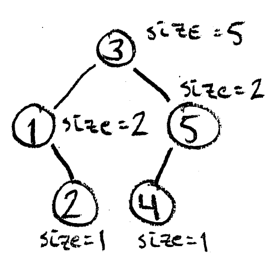
2a) Easy Case (k's node has no children)
Delete k's node from the tree.

2b) Medium Case (k's node has one child)
1) Overwrite k's node with its child
2) Remove the pointer to the child's original position or delete the original child

2c) Difficult Case (k's node has two-children)
1) Compute k's predecessor, p.
2) Swap nodes k and p. In k's new position, k has no right child.
3.) Depending if k has Ø or 1 child follow the case for deletion.

Select and Rank

To accomplish select (Find i^{th} order statistic) and rank (Find number of keys less than or equal to a selected value) we must store some extra information about the tree itself, also known as augmenting the data.



Example Augmentation: size(x) = # of tree nodes in subtree rooted at x.

If x has children y and z, then

$$\underbrace{\text{size}(y)}_{\text{population of left subtree}} \quad \underbrace{\text{size}(z)}_{\text{population of right subtree}}$$

$$\text{size}(x) = \text{size}(y) + \text{size}(z) + 1 \leftarrow x \text{ itself}$$

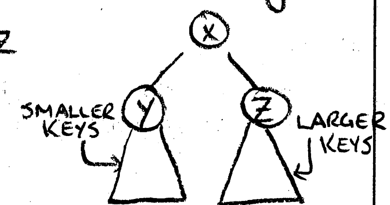
BASICS (con't)

SELECT AND RANK (con't)

Be aware when augmenting the data structure there are tradeoffs, for example, if you perform an insertion or deletion on the search tree you must also update the augmented data. This takes some time so ensure that by augmenting your data structure you are using it to speed up your frequently used operations rather than tracking the information for information's sake.

SELECT i^{th}-ORDER STATISTIC

Having augmented the node data with a node's subtree sizes



1.) Start at root x with children y and z
2.) Let $a = size(y)$, $a = \emptyset$ if no left child
3.) If $a = i-1$ return x's key
4.) If $a \geq i$ recursively compute i^{th}-order statistic of search tree rooted at y
5.) If $a < i-1$ recursively compute $(i-a-1)^{th}$-order statistic of search tree rooted at z. The order statistic changes since you've thrown out a section of the search tree less than the i^{th} order statistic. For example, suppose $x = 12^{th}$ order statistic, therefore, $size(y) = 11$ and you want the 17^{th}-order statistic in a binary search tree of size 25. $size(z) = 13$, since $13 < 17$ if you searched for the 17^{th} order statistic in the subtree rooted at z you would get an undefined answer. After adjustment, however, $(17-11-1) = 5$ the 5^{th} order statistic in the subtree rooted at z does exist.

RUNNING TIME = O(HEIGHT)

See Selection Algorithm notes for more info on i^{th}-order statistic.

SELECT RANK

Find where the value should exist in the search tree, then it's the size of the tree rooted at x with all untraversed nodes removed.

(con't on next sheet)

Balanced Binary Search Tree

Red-Black Tree

This type of binary tree guarantees a height of $\lg_2 n$ where n is the number of nodes in the tree structure ensuring best case runtimes over the basic binary search tree implementation.

There are other types of balanced binary search trees as well:

AVL trees - A self-balancing binary search tree named after inventors, Adelson-Velsky and Landis. This was the first data structure of this type. Often compared to red-black trees as both take $O(\lg n)$ for the supported operations (see page 1), however, AVL trees are better optimized for lookups due to being more strictly balanced.

Splay trees - A self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. A splay tree performs supported operations in amortized $O(\lg n)$ time which means $O(n)$ time can happen, but is both unlikely and can be further prevented using randomization.

B-trees - A self-balancing tree (not binary) data structure. A generalization of the binary search tree in that a B-tree may have more than two children per parent node. Well-suited for storage systems that read and write relatively large blocks of data, such as discs. Commonly used in databases and file systems.

Invariants (How the Height is Controlled)

1. Each node is either red or black
2. The root is always black
3. There are no two red nodes in sequence moving up or down the tree structure. A red node can only have black child nodes. Bear in mind a black node is allowed to have black child nodes.
4. Every path from the root to a null node (like in an unsuccessful search) has the same number of black nodes.

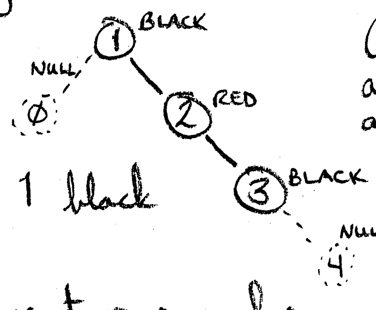
Balanced Binary Search Tree (con't)

Red-Black Tree (con't)

Example: Non-Balanced Search Tree

A chain of length 3 cannot be a red-black tree

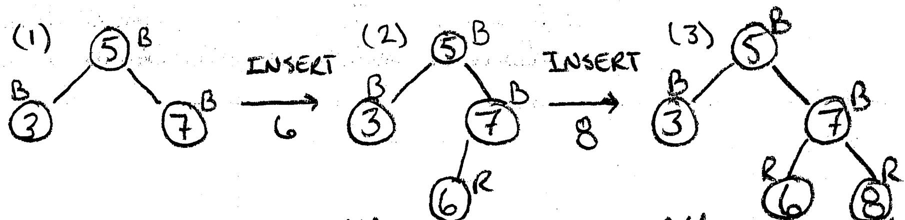
A search for Ø reaches a non-existent, null node and passes through 1 black node.



A search for 4 reaches a non-existent, null node and passes through 2 black nodes.

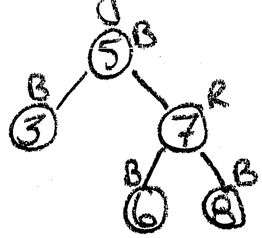
From the above two searches no matter how the nodes are colored every path from the root to a null node does not and cannot pass through the same number of black nodes violating the fourth invariant listed on the previous page. Additionally, a chain of length 3 is not balanced.

Example: Balanced Search Tree



All invariants maintained | All invariants maintained

The last tree (3) above can also be represented by altering node 7. This is done by rotation (covered later) and is an integral part of maintaining the red-black tree invariants as nodes are inserted and deleted from the binary search tree. Maintains balance.



(con't on next sheet)

BALANCED BINARY SEARCH TREE (con't)

RED-BLACK TREE (con't)

HEIGHT GUARANTEE PROOF

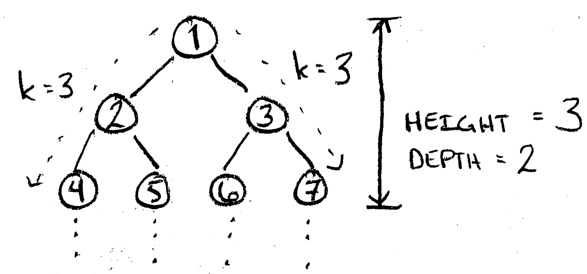
CLAIM: Every red-black tree with n nodes has height $\leq 2\lg(n+1)$

PROOF:

If every root-null path has $\geq k$ nodes, then the tree, at the top, includes a perfectly balanced search tree of depth $k-1$.

Therefore, the size of the tree must be at least $2^k - 1$.

EXAMPLE: $k = 3$, $\text{size}(k) = 2^3 - 1 = 7$ nodes at top



nodes labeled as example, not intended as search tree

We can rewrite $n \geq 2^k - 1$ as $k \leq \lg(n+1)$. So, in a red-black tree with n nodes there exists a root-null path with at most $\lg(n+1)$ black nodes.

From the 4^{th} invariant (page 7) every path from the root to a null node has the same number of black nodes, therefore the statement above is correct that every path, root-null, has $\leq \lg(n+1)$ black nodes.

From the 3^{rd} invariant every root-null path has $\leq 2\lg(n+1)$ nodes since there can be no two red nodes in sequence moving up or down the tree structure.

(con't on next sheet)

BALANCED BINARY SEARCH TREE (con't)

ROTATIONS

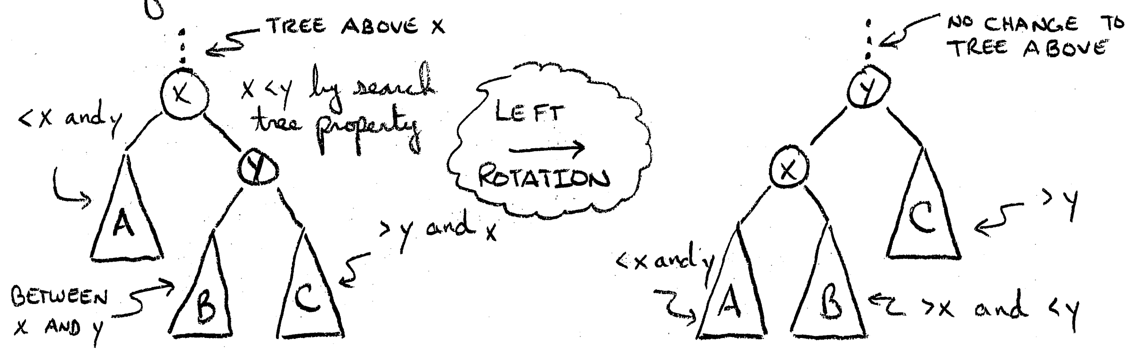
Rotations are a key operation common to all balanced search tree implementations, for example: red-black trees, AVL trees, B-trees, etc.

Rotations locally rebalance subtrees at a node in O(1), constant, time. This is because only parent-child pointers need to be adjusted.

LEFT ROTATION

Let x be a parent and y be x's right child.

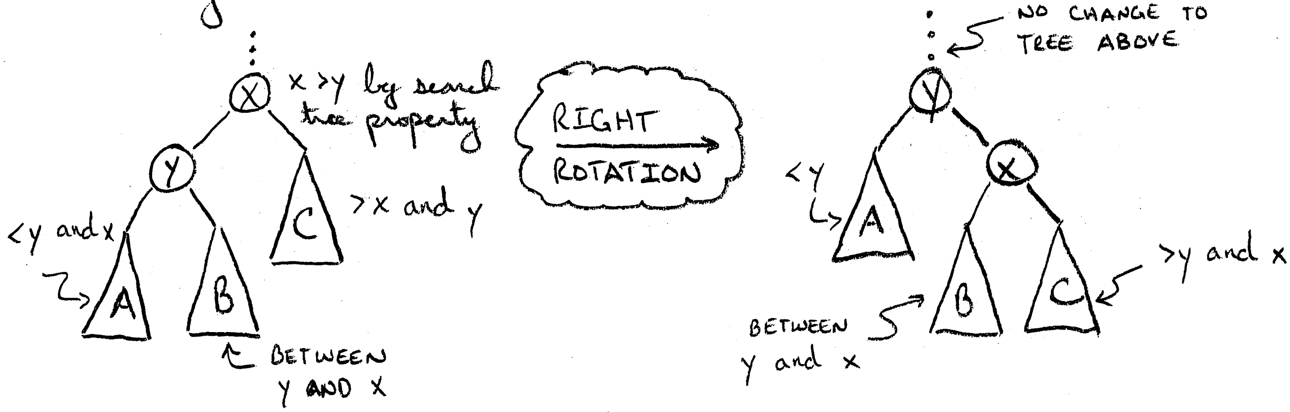
With a left rotation we are adjusting the tree such that y becomes the parent and x becomes y's left child (rotating x left).



RIGHT ROTATION

Let x be a parent and y be x's left child.

With a right rotation we are adjusting the tree such that y becomes the parent and x becomes y's right child (rotating x right).



BALANCED BINARY SEARCH TREE (con't)

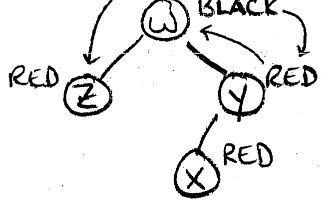
INSERTION IN A RED-BLACK TREE

HIGH LEVEL IMPLEMENTATION

Proceed as in a normal binary search tree for insertion/deletion, then recolor and perform rotations until invariants are restored.

1. Insert x as usual (makes x a leaf), let y be x's parent.
2. Node x will have to be either red or black.
   Recall that coloring x red or black may break invariants that there cannot be two red nodes in a row and every path from root to a null pointer must have the same number of black nodes.
   As such, color x red since the invariant that there can be no two red nodes in sequence as you traverse up or down the tree is a local flaw and can be dealt with on a smaller scale than violating the invariant requiring every path from root to null to have the same number of black nodes which is a flaw affecting the entire search tree structure.
3. Check the color of x's parent, y.
   If y is black, done.
   Else y is red and we violate the invariant requiring that there not be two reds in sequence as we move up and down the tree structure.
4. Since y is red from 3, y has a parent w that must be black.

CASE 1: THE OTHER CHILD OF W IS ALSO RED



In this case, recolor z and y black and color w red.

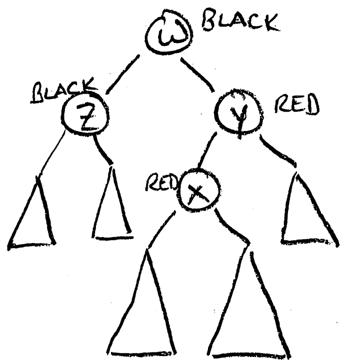
Doing this restores the requirement that all paths from root to null possess the same number of black nodes, but may continue to violate the two red nodes in sequence invariant. As such, continue recoloring nodes on the way through the tree until there are either no two red nodes in sequence or you reach the root. If you get to the point where you have to color the root red, don't. Leave the root black and all properties of a red-black tree hold.

(con't on next sheet)

BALANCED BINARY SEARCH TREE (con't)

INSERTION IN A RED-BLACK TREE (con't)

CASE 2: w EITHER HAS NO OTHER CHILD THAN y OR THE OTHER CHILD OF w, z, IS ALSO BLACK



Possible situation as we propagate red nodes up through the search tree.

Let x and y be the current double-red pair with x being the deeper node.

Let w be x's grandparent.

Suppose w's other child (≠ y) is null or is a black node, z.

This case can be resolved by eliminating the double-red sequence in O(1) time via 2-3 rotations and recolorings, case 1.

EXAMPLE: