Binary Search Tree

June 26, 2017
A binary search tree is a binary tree:
- If y is a node in the left subtree of x, then $y.key \leq x.key$
- If y is a node in the right subtree of x, then $y.key \geq x.key$

Binary search tree can efficiently maintain a dynamically changing dataset in sorted order. It's easier to add and remove elements.

Support everything you can get from a sorted array

(a) binary search tree; (b) NOT a binary search tree
Some properties

- **Relationship to Quicksort:** We can think of each node \( x \) as a pivot for quicksort. The keys of all the nodes in left subtree are less than \( x \), all nodes in right subtree are greater than \( x \). (assuming no duplicates)

- **Sorting the keys:** We can do an inorder traversal of the tree to recover the nodes in sorted order from left to right

- **Operations**
  - Search
  - Insert
  - Delete
TreeNode search(TreeNode root, int key){
    if (root==null || root.val==key)
        return root;
    if (root.val > key)
        return search(root.left,key);
    else
        return search(root.right, key);
}

- Time complexity: $O(h)$, $h$ is the height of the tree.
- Average $O(\log n)$
- Worst $O(n)$
TreeNode tree_minimum(TreeNode root) {
    if (root == null) return null;
    while (root.left != null)
        root = root.left;
    return root
}
TreeNode tree_maximum(TreeNode root) {
    if (root == null) return null;
    while (root.right != null)
        root = root.right;
    return root
}

● Time complexity: $O(h)$, $h$ is the height of the tree.
As before, we will assume that all keys are distinct. A new node is always inserted at leaf. We walk down from the root, comparing with each node until we get to the place.

```
TREE-INSERT(T, z)
1  y = NIL
2  x = T.root
3  while x ≠ NIL
4     y = x
5     if z.key < x.key
6        x = x.left
7     else x = x.right
8  z.p = y
9  if y == NIL
10     T.root = z    // tree T was empty
11  elseif z.key < y.key
12     y.left = z
13  else y.right = z
```
The *successor* of a node is the next node in Inorder traversal of the binary tree.

If the right subtree of $x$ is nonempty, then the successor of $x$ is the leftmost node in the right subtree.

If the right subtree of $x$ is empty, then $y$ is the lowest ancestor of $x$ whose left child is also an ancestor of $x$.

If $x$ has a right child, then the successor of $x$ does not have a left child.

If $x$ has a left child, then the predecessor of $x$ does not have a right child.
Successor examples

CLRS 12.2

- The successor of the node with key 15 is the node with key 17
- The successor of the node with key 13 is the node with key 15
TreeNode successor(TreeNode x) {
    if (x.right != null) {
        return tree_minimum(x.right);
    }
    TreeNode p = x.parent;
    while (p != null && x == p.right) {
        x = p;
        p = p.parent;
    }
    return p;
}
Deletion

Let us say that we are deleting a node $n$

Case 1: if $n$ has no children, then we simply remove it by modifying its parent to replace $n$ with null as its child.

Case 2: if $n$ has one child, then we elevate that child to take $n$’s position in the tree, by modifying $n$’s parent to replace $n$ by $n$’s child.

Case 3: if $n$ has two children, we need to find $n$’s successor and replace $n$ with it. We know that $n$’s successor has at most 1 child (why?), so we can immediately apply either case 1 or case 2 to it.
Build Binary Search Tree

- Sort the array of integers $O(n \log n)$
- Construct BST from sorted array $O(n)$

TreeNode sortedArray2BST(int[] A) {
    if (A.length == 0)
        return null;
    return sortedArray2BST(A, 0, A.length - 1);
}
TreeNode helper(int[] num, int start, int end) {
    if (start > end)
        return null;
    int mid = start + (end - start)/2;
    TreeNode root = new TreeNode(num[mid]);
    root.left = helper(num, start, mid - 1);
    root.right = helper(num, mid + 1, end);
    return root;
}
Build Binary Search Tree

```java
BuildTree(int[] A) {
    if (A.length == 0)
        return;
    TreeNode root = null;
    for(int i = 0; i < A.length; i++){
        tree_insert(root, A[i]);
    }
}
```

- Average case is $O(n \log n)$
- Worst case is $O(n^2)$, if A is already in sorted order.
Balanced Binary Search Tree

- If the input sequence is sorted, we have linear time cost per operation rather than logarithmic time cost per operation
  - analogous to quicksort

- One solution to this problem is to insist on an extra structural condition called *balance*: No node is allowed to get too deep.

- Balanced BST:
  - AVL Tree
  - Red-Black Tree
  - 2-3 Tree

- A binary search tree that has an additional balance condition.

- Any balance condition must be easy to maintain and ensures that the depth of the tree is $O(\log n)$. 
An AVL tree is a binary search tree with the additional balance property that, for any node in the tree, the height of the left and right subtrees can differ by at most 1. Assuming the height of an empty subtree is 1.

(a) an AVL tree; (b) NOT an AVL tree

Textbook figure 19.21
AVL Tree

- The difficulty is that operations such as insert, delete will change the tree.
- These operations can destroy the balance of several nodes in the tree.
- The balance must then be restored before the operation can be considered complete.
- A key observation is that after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered.
- The node to be rebalanced is \( X \):
  - An insertion in the left subtree of the left child of \( X \) (left - left)
  - An insertion in the right subtree of the left child of \( X \) (left - right)
  - An insertion in the left subtree of the right child of \( X \) (right - left)
  - An insertion in the right subtree of the right child of \( X \) (right - right)
$k_2$ violates the AVL balance property because its left subtree is two levels deeper than its right subtree. right-rotate($k_2$)

Textbook 19.23
Left Left Case (Single Rotation)

```java
static BinaryNode rotateWithLeftChild( BinaryNode k2 )
{
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

(a) Before rotation
(b) After rotation

Textbook 19.24 19.25
Right Right Case (Single Rotation)

- left-rotate($k_1$)

```
static BinaryNode rotateWithRightChild( BinaryNode k1 )
{
    BinaryNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

Textbook 19.26 19.27
Left Right Case (Double Rotation)

- Single rotation does not work, need double rotation
Left Right Case (Double Rotation)

```java
static BinaryNode doubleRotateWithLeftChild( BinaryNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

(a) Before rotation

(b) After rotation

Textbook 19.30 19.32
Right Left Case (Double Rotation)

(a) Before rotation
(b) After rotation

```java
static BinaryNode doubleRotateWithRightChild(BinaryNode k1) {
    k1.right = rotateWithLeftChild(k1.right);
    return rotateWithRightChild(k1);
}
```

Textbook 19.30 19.32
A red-black tree is a binary search tree with one extra bit of storage per node: its **color**.

Red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p.

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. NILs are black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
Red-Black Tree

Textbook Figure 19.34
For some node \( x \), let \( b(x) \) be the black height of \( x \), which is the number of black nodes on a \( x \) to NIL path excluding \( x \).

The number of non-NIL descendants of \( x \) is at least \( 2^{b(x)} - 1 \).

Any valid red-black tree on \( n \) nodes has height at most \( 2 \log(n + 1) = O(\log n) \)

\[ n \geq 2^{b(r)} - 1 \geq 2^{\frac{h}{2}} - 1, \text{ } r \text{ is the root} \]

Why it is balanced?

- by property (5), all paths from root to leaf have length at least \( b(r) \)
- by property (4), the number of red nodes is limited to half of the path, so the length is at most \( 2b(r) \)
Proof by Induction

- The number of non-NIL descendants of \( x \) is at least \( 2^{b(x)} - 1 \)
- Base case: NIL node has \( b(x) = 0 \) and \( 2^0 - 1 = 0 \) non-NIL descendants.
- For inductive step, let \( \text{num}(x) \) be the number of non-NIL descendants of \( x \). Then:

\[
\text{num}(x) = 1 + \text{num}(x.\text{left}) + \text{num}(x.\text{right}) \\
\geq 1 + (2^{b(x)-1} - 1) + (2^{b(x)-1} - 1) \text{ (by induction)} \\
= 2^{b(x)} - 1
\]
Reversible operations

Take constant time
RB-INSERT(T, z)
1  y = T.nil
2  x = T.root
3  while x \neq T.nil
4     y = x
5     if z.key < x.key
6        x = x.left
7     else x = x.right
8  z.p = y
9  if y == T.nil
10    T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
RB-INSERT-FIXUP(T, z)

1. while z.p.color == RED
2.     if z.p == z.p.p.left
3.         y = z.p.p.right
4.     if y.color == RED
5.         z.p.color = BLACK
6.         y.color = BLACK
7.         z.p.p.color = RED
8.         z = z.p.p
         // case 1
9. else if z == z.p.right
10.    z = z.p
        // case 2
11.    LEFT-ROTATE(T, z)
12.    z.p.color = BLACK
13.    z.p.p.color = RED
14.    RIGHT-ROTATE(T, z.p.p)
        // case 3
15. else (same as then clause
       with “right” and “left” exchanged)
16.    T.root.color = BLACK
        // case 3

CLRS
Example

(a)  

(b)  

Case 1

Case 2
Example

CLRS 13.4