Binary Search Trees

Analysis of Algorithms
Binary Search Trees

• A BST is a binary tree in **symmetric order**

• Each node has a key and every node’s key is:
  – larger than all keys in its left subtree
  – smaller than all keys in its right subtree
BST Implementation

```java
private class Node {
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val) {
        this.key = key;
        this.val = val;
    }
}

public class BST<Key extends Comparable<Key>, Value> {
    private Node root;

    private class Node {
        /* see previous slide */
    }

    public void put(Key key, Value val) {
        /* see next slides */
    }

    public Value get(Key key) {
        /* see next slides */
    }

    public void delete(Key key) {
        /* see next slides */
    }

    public Iterable<Key> iterator() {
        /* see next slides */
    }
}
```
BST Put

- Put a new node with the given key in the tree at the proper location
- Number of compares = $1 + \text{depth of node}$

```java
public void put(Key key, Value val)
{   root = put(root, key, val); }

private Node put(Node x, Key key, Value val)
{   if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else if (cmp == 0)
        x.val = val;
    return x;
}
BST Get

- Return value corresponding to the given key, or null if no such key
- Number of compares = 1 + depth of node

```java
public Value get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```
Shape of the Tree

• The order that the keys are inserted into the tree determines the shape of the tree
• Recall that the number of comparisons for search/insert is equal to $1 + \text{depth of node}$
Shape of the Tree

Question: Sketch the binary search tree for the following input sequences:
   a) [20, 17, 25, 27, 13, 30]
   b) [30, 13, 17, 20, 25, 27]

Question: The worst-case running time of the search (or insert) operation in a binary search tree containing $N$ keys is
   a) constant
   b) $\log_2 N$
   c) the height of the tree
   d) $N^2$
Analysis of BST

• Insertion takes *longer* for BST (proportional to the height of the tree, h) than for a regular binary tree (constant time insertion)

• However, finding a particular item is faster if the BST is balanced

• For example, if N = 2048 and the BST is balanced, the maximum number of comparisons that have to be made is 11 (vs. 2048 for a binary tree)
Finding Minimum and Maximum

- **Tree minimum**
  - Follow left subtree to null

- **Tree maximum**
  - Follow right subtree to null

Node min(Node x) {
    if (x.left == null) return x;
    else return min(x.left);
}
Finding a Successor in a BST

• Tree successor (find the key that comes after a given key)
  – Case 1: If the right subtree of a node is not empty, then return the minimum value in the right subtree
  – Case 2: If the right subtree of a node is empty, then go up the tree until we encounter the lowest ancestor whose left child is also an ancestor (note that a node can be its own ancestor)

Case 1: successor of 31 is 35.
Case 2: successor of 25 is 31.
Case 2: successor of 35 is 38.
Case 2: 40 has no successor.
Deleting a Node in a BST

- Tree deletion (Hibbard deletion)
- Which node is actually removed depends on the number of children the node has
  - Case 0 (0 children): just delete the node
  - Case 1 (1 child): splice out the node
  - Case 2 (2 children): splice out the node’s successor, and replace the deleted node with it’s successor
Hibbard Deletion

Case 0 (0 children):

– Delete the node by just eliminating it (find the node’s parent and set parent.left or parent.right to null)
Hibbard Deletion

Case 1 (1 child):

– Delete the node by just splicing it out (link the node’s parent to the node’s child)
Hibbard Deletion

Case 2 (2 children):

– Step 1: Find the node to be deleted
– Step 2: Find the successor of the node to be deleted
  • If a node has two children, then the successor of the node will have no left child (why?)
– Step 3: Replace the node to be deleted with the node’s successor
– Step 4: Update the links and the subtree counts
Hibbard Deletion

Case 2 (2 children):

Step 1. 

Step 2. 

Step 3. 

Step 4. 

node to delete

search for key E

go right, then go left until reaching null left link

successor min(t.right)

t.left
deleatemin(t.right)

update links and node counts after recursive calls
Hibbard Deletion

```java
public void deleteMin()
{   root = deleteMin(root);
}

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

```java
public void delete(Key key)
{   root = delete(root, key);
}

private Node delete(Node x, Key key)
{   if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.count = size(x.left) + size(x.right) + 1;
    return x;
}
```
Hibbard Deletion Analysis

• Problem: repeated insertions and deletions may leave the tree unbalanced
  – Search becomes proportional to $\sqrt{N}$ instead of $\log N$
• This is an unsatisfactory solution for a dynamic application
• Is there a way to insert and delete from a BST that maintains log $N$ height?
  – Yes, if we can re-balance the tree every time an insertion or deletion occurs
Balanced Search Trees

- 2-3 trees
- Red-black trees
- B-trees
- AVL trees
- Splay trees
2-3 Trees

- Allow 1 or 2 keys per node
  - 2-node: one key, two children
  - 3-node: two keys, three children
  - Property: every path from the root to a null link has the same length (guaranteed)
2-3 Trees

- Inserting a key into a 2-node is simple
- For example, insert the key K into the 2-3 tree shown on the left below:
  - Add the new key to the 2-node (L)
  - The 2-node (L) becomes a 3-node with the keys (K L)
2-3 Trees

• Inserting a key into a 3-node is also simple
  – If a parent is a 4-node, split it into three 2-nodes
  – For example, insert key Z into the 2-3 tree shown on the left:
    • Add key Z to the 3-node (S X) to create a temporary 4-node (S X Z)
    • Move the middle key (X) in the 4-node into the parent node (R), creating a new 3-node (R X), and split the 4-node (S X Z) into two 2-nodes (S) and (Z)
    • Repeat up the tree, as necessary
2-3 Trees

• Example: Inserting L into the 2-3 tree shown on the left results in the 2-3 tree shown on the right:

• Question: Show all the steps involved in building a 2-3 tree from the input sequence [S,E,A,R,C,H,X,P,L]
2-3 Trees

- Splitting a 4-node is a *local transformation*; it involves a constant number of operations
Analysis of 2-3 Trees

• 2-3 trees are perfectly balanced: every path from the root to a null link has the same length

• Tree height:
  – Worst case: $\log_2 N$ (all nodes in the tree are 2-nodes)
  – Best case: $\log_3 N \approx 0.631 \log_2 N$ (all nodes in the tree are 3-nodes)
  – When $N = \text{one million}$, the tree height will be between 12 and 20
  – When $N = \text{one billion}$, the tree height will be between 18 and 30

• Guaranteed logarithmic performance for search and insert
2-3 Tree Implementation

• Direct implementation is complicated because:
  – Maintaining multiple node types is cumbersome
  – Need multiple compares (one for 2-nodes and a different one for 3-nodes)
  – Need to move back up the tree to split 4-nodes
  – Large number of cases for splitting
2-3 Tree Implementation

QUESTION: Suppose that you are inserting a new key into a 2-3 tree. Under which one of the following scenarios must the height of the 2-3 tree increase by one?

a) When the number of keys equals one less than a power of two
b) When the number of nodes equals one less than a power of two
c) When the final node on the search path from the root is a 3-node
d) When every node on the search path from the root is a 3-node
Red-Black Trees

- A (left-leaning) red-black tree is a binary search tree with additional properties:
  1. Each link is colored either red or black
  2. Red links lean left
  3. No node has two red links to its children
  4. There cannot be two or more red links in a row along a path
  5. Every path from the root to a null link has the same number of black links
Red-Black Trees

- There is a 1-1 correspondence between 2-3 trees and left-leaning red-black trees.
Implementation of R-B BST

The color of a node is the color of the link to its parent.

```java
private static final boolean RED = true;
private static final boolean BLACK = false;

private class Node {
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
}

private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```
Search in Red-Black Trees

• Search in a red-black BST is the same as for an ordinary BST, but runs faster because the tree is always relatively balanced.
• Most other operations are identical to an ordinary BST.
• To ensure that the BST is relatively balanced, during insertion or deletion always maintain the red-black properties.
Insertion in Red-Black Trees

• By default, initialize a node’s color (the link to its parent) to red before inserting the node in the tree

• Sometimes, after the insertion, a link may be leaning in the “wrong” direction or have the “wrong” color

• It thus may be necessary to re-orient some links by changing just a few pointers
  – Left rotation
  – Right rotation
Rotations in Red-Black Trees

• Left rotation
  – Orient a right-leaning red link to lean left (this will not change the black-height)
  – For example, the figure on the left below has a right-leaning red link, so rotate E left to make the link lean left

```java
private Node rotateLeft(Node h) {
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```
Rotations in Red-Black Trees

• Right rotation
  – Orient a left-leaning red link to lean right (we’ll see why this may be necessary later)
  – For example, the figure on the left below has a left-leaning red link, so rotate S right to make the link lean right
Rotations in Red-Black Trees

Question: What is the in-order traversal of the tree shown below? Sketch the tree that results after a left-rotation of the node containing E. What is the in-order traversal of the resulting tree?
Color-Flip in Red-Black Trees

• After an insertion, we might end up with a node that has two red links to its children (a 4-node)
  – Just re-color the 3 links

```java
private void flipColors(Node h) {
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```
Insertion in Red-Black Trees

• Insertion into a 2-node:
  – Do a standard BST insert, color the new link to the parent red
  – If it is a right link, rotate left

• Insertion into a 3-node: one of three possible situations will occur (see next slide)
  – The inserted key is **larger** than the other keys in the 2-node
  – The inserted key is **smaller** than the other keys in the 2-node
  – The inserted key is **between** the other keys in the 2-node
Insertion in Red-Black Trees

larger: insert c

search ends at this null link

attached new node with red link

colors flipped to black

smaller: insert a

search ends at this null link

attached new node with red link

rotated right

colors flipped to black

between: insert b

search ends at this null link

attached new node with red link

rotated left

rotated right

colors flipped to black
Insertion in Red-Black Trees

Example: insert H
Insertion in Red-Black Trees

Example: insert P (repeat up the tree as needed)
Implementation of Insert R-B

• Same code handles all cases!
  – If the right child is red and the left child is black: rotate left
  – If the left child and the left-left grandchild are both red: rotate right
  – If both children are red: flip colors

```java
private Node put(Node h, Key key, Value val)
{
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if (cmp < 0) h.left = put(h.left, key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val = val;

    if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right)) flipColors(h);

    return h;
}
```

*only a few extra lines of code provides near-perfect balance*
Analysis of Red-Black Trees

• Height of tree is $\leq 2 \log N$ in the worst case
  – Every path from the root to a null link will have the same number of black links
  – A balanced red-black tree will never have two red links in a row

• The height of the tree is $\sim \log N$ in typical applications

• Expert witness testimony in lawsuit: “If implemented properly, the height of a red-black binary search tree with $N$ keys is at most $2 \log N$”