CMPSC 250

Analysis of algorithms

Dr. Aravind Mohan

Allegheny College
Binary Search Trees

A BST is a binary tree in a symmetric order

A binary tree is either:
- Empty
- Two disjoint binary trees (left and right)
Anatomy of a binary tree

Anatomy of a binary search tree
BST in Java

- In Java a BST is a reference to a root Node
- A Node is comprised of four fields:
  - A Key and a Value
  - A reference to the left and right subtree.
BST in Java

![BST Diagram]

- BST (Binary Search Tree)
- Node
  - key
  - val
  - left
  - right
  - BST with smaller keys
  - BST with larger keys

Binary search tree
Number of compares?
\[ \text{depth of the tree} + 1 \]
Insert: Java Implementation

```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;
}
```

Number of compares?
depth of the tree + 1
Possibilities for BST

best case

```
   H
  / \  
 C   S
 /   /  
A   E   R   X
```

typical case

```
   S
  /  
 E   X
 /   /  
A   R   C
```

worst case

```
   S
  /  
 E   X
 /   /  
A   R   C
```

```
Search Analysis

- Search hits built from $N$ random keys require $\sim 2\ln N$ compares on average.
- Or about $1.39 \log_2 N$ compares on average.
Ordered Operations: Min, Max

- **Minimum**: smallest key in the table
- **Maximum**: largest key in the table
Ordered Operations: Floor, Ceiling

- **Floor**: largest key in the table $\leq$ the given key
- **Ceiling**: smallest key in the table $\geq$ the given key
### Ordered ST Operations Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential Search</th>
<th>Binary Search</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$N$</td>
<td>$\lg N$</td>
<td>$h$</td>
</tr>
<tr>
<td>insert</td>
<td>1</td>
<td>$N$</td>
<td>$h$</td>
</tr>
<tr>
<td>min / max</td>
<td>$N$</td>
<td>1</td>
<td>$h$</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>$N$</td>
<td>$\lg N$</td>
<td>$h$</td>
</tr>
<tr>
<td>rank</td>
<td>$N$</td>
<td>$\lg N$</td>
<td>$h$</td>
</tr>
<tr>
<td>select</td>
<td>$N$</td>
<td>1</td>
<td>$h$</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>$N \log N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

$h = \text{height of BST (proportional to log N if keys inserted in random order)}$

Order of growth of running time of ordered symbol table operations
Deletion: lazy

- To remove a node with a given key:
  - Set its value to null
  - Leave key in tree to guide searches (but don’t consider it equal to search key)
Delete Minimum Key

- Go left until reaching null left link.
- Return that node's right link.
- Available for garbage collection.
- Update links and node counts after recursive calls.
public void deleteMin()
{
    root = deleteMin(root);
}

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
Delete a Node with Key $k$: Case 1
Delete a Node with Key $k$: Case 2
Delete a Node with Key $k$: Case 3

1. Search for key $E$.
2. If $E$ is a leaf, replace it with its successor $H$.
3. If $E$ has a right child, replace it with its successor $H$ and update the links.
4. If $E$ has a left child, replace it with its successor $H$ and update the links.
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;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
## ST Implementation Summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>(linked list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other operations also become ✓N if deletions allowed.
Chapter 03, Section 3.2, 396 - 424
Class Activity

1. Exercise 3.2.1 - Draw the BST that results when you insert the key E A S Y Q U E S T I O N in that order (associating the value i with the ith key) into an initially empty tree. How many comparisons are needed to build the tree.

2. Exercise 3.2.17 - Draw the sequence of BSTS’s that results when you delete the keys from the tree you created in Exercise 3.2.1, one by one in the order they were inserted.