3.2 Binary Search Trees

Definition. A BST is a binary tree in symmetric order.

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

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.

Java definition. 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.

Key and Value are generic types; Key is Comparable.

BST implementation (skeleton)

```java
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 search**

**Get.** Return value corresponding to given key, or null if no such key.

#### BST search: Java implementation

```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;
}
```

**Running time.** Proportional to depth of node.

#### BST insert

**Put.** Associate value with key.

Search for key, then two cases:
- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.

#### BST 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;
}
```

**Running time.** Proportional to depth of node.
BST trace: standard indexing client

Tree shape

- Many BSTs correspond to same set of keys.
- Cost of search/insert is proportional to depth of node.

Remark. Tree shape depends on order of insertion.

Observation. If keys inserted in random order, tree stays relatively flat.

Ex. Insert keys in random order.

BST insertion: random order visualization
Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1-1 if no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

But... Worst-case for search/insert/height is $N$. (exponentially small chance when keys are inserted in random order)

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>(unordered list)</td>
<td>$N$</td>
<td>$N/2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td>$N$</td>
<td>$N/2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$1.39 \lg N$</td>
<td>?</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Costs for java FrequencyCounter 8 < tale.txt using BST
Minimum and maximum

Minimum. Smallest key in table.
Maximum. Largest key in table.

Q. How to find the min / max.

Floor and ceiling

Floor. Largest key ≤ to a given key.
Ceiling. Smallest key ≥ to a given key.

Q. How to find the floor / ceiling.

Computing the floor

Case 1. [k equals the key at root]
The floor of k is k.

Case 2. [k is less than the key at root]
The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]
The floor of k is in the right subtree (if there is any key ≤ k in right subtree); otherwise it is the key in the root.

Computing the floor

```java
public Key floor(Key key) {
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0)  return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}
```
In each node, we store the number of nodes in the subtree rooted at that node.

To implement `size()`, return the count at the root.

Remark. This facilitates efficient implementation of `rank()` and `select()`.

```
public int size() {  return size(root);  }
private int size(Node x) {  if (x == null) return 0;  return x.N;  }
```

```
private class Node
{   private Key key;   private Value val;   private Node left;   private Node right;   private int N;   }
```

```
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;    x.N = 1 + size(x.left) + size(x.right);    return x;  }
```

```
public int rank(Key key) {  return rank(key, root);  }
private int rank(Key key, Node x) {    if (x == null) return 0;    int cmp = key.compareTo(x.key);    if (cmp < 0) return rank(key, x.left);    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);    else              return size(x.left);  }
```

```
public Iterable<Key> keys() {     Queue<Key> q = new Queue<Key>();     inorder(root, queue);     return q;}
private void inorder(Node x, Queue<Key> q) {    if (x == null) return;    inorder(x.left, q);    q.enqueue(x.key);    inorder(x.right, q);     }
```

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

Property. Inorder traversal of a BST yields keys in ascending order.
Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```plaintext
inorder(S)  inorder(E)    inorder(A)      enqueue A      inorder(C)        enqueue C    enqueue E    inorder(R)      inorder(H)        enqueue H        inorder(M)          enqueue M      print R  enqueue S  inorder(X)    enqueue X
```

BST: ordered symbol table 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>

worst-case running time of ordered symbol table operations

h = height of BST (proportional to log N if keys inserted in random order)

ST implementations: 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>search</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>delete</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>search (linked list)</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>insert</td>
<td>lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>delete</td>
<td>lg N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.39 lg N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Next. Deletion in BSTs.
BST deletion: lazy approach

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).

Cost. $O(\log N’)$ per insert, search, and delete (if keys in random order), where $N’$ is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone overload.

Deleting the minimum

To delete the minimum key:
- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
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;
}
```

Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

**Case 0.** [0 children] Delete $t$ by setting parent link to null.

```
node to delete
```

```
replace with null link
available for garbage collection
```

```
deleting C
```

```
update counts after recursive calls
```

```
deleting R
```

**Case 1.** [1 child] Delete $t$ by replacing parent link.

```
node to delete
replace with child link
available for garbage collection
update counts after recursive calls
```
To delete a node with key $k$: search for node $t$ containing key $k$.

**Case 2.** [2 children]
- Find successor $x$ of $t$.
- Delete the minimum in $t$’s right subtree.
- Put $x$ in $t$’s spot.

Hibbard deletion

**Hibbard deletion: Java implementation**

```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;
                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;
    }
```

Hibbard deletion: analysis

**Unsatisfactory solution. Not symmetric.**

**Surprising consequence. Trees not random (!) ⇒ sqrt(N) per op.**

Longstanding open problem. Simple and efficient delete for BSTs.

**ST implementations: 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>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>$\sqrt{N}$</td>
</tr>
</tbody>
</table>

Next lecture. Guarantee logarithmic performance for all operations.