CS 171: Introduction to Computer Science II

Binary Search Trees (cont.)
Binary Search Trees

• Definitions and terminologies
• Search and insert
• Traversal
• Ordered operations
• Delete
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.
BST representation in Java

**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.

```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;
    }
}
```

Key and Value are generic types; Key is Comparable
## 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></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
</tr>
</tbody>
</table>
Traversals

- **In-order**
  - Left subtree, current node, right subtree

- **Pre-order**
  - Current node, left subtree, right subtree

- **Post-order**
  - Left subtree, right subtree, current node
Inorder traversal

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

```java
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    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);
}
```

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

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

- In-order
  A C E H M R S X
- Pre-order?
- Post-order?
Traversal

• In-order
  A C E H M R S X

• Pre-order
  S E A C R H M X

• Post-order
  C A M H R E X S

• How to visit the nodes in descending order?
• What’s the use of pre-order and post-order traversal?
Expression Tree

- Post-order traversal results in postfix notation
- Pre-order traversal results in prefix notation

A tree that represents the expression
\[ 3 \times \frac{(7+1)}{4} + (\times 7 - 5) \]
The upward pointing arrows show how the value of the expression can be computed.
Binary Search Trees

• Definitions and terminologies
• Search and insert
• Traversal

• Ordered operations
  – Minimum and maximum
  – Rank: how many keys < k?
  – Select: key of given rank

• Delete
Minimum and maximum

**Minimum.** Smallest key in table.

**Maximum.** Largest key in table.

Q. How to find the min / max?
Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node. To implement \texttt{size()}, return the count at the root.

\textbf{Remark.} This facilitates efficient implementation of \texttt{rank()} and \texttt{select()}. 
BST implementation: subtree counts

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

```java
public int size() {
    return size(root);
}
```

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

```java
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;
}
```
**Rank**

**Rank.** How many keys < $k$?

Easy recursive algorithm (4 cases!)

```java
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 if (cmp == 0) return size(x.left);
}
```
Selection

Select. Key of given rank.

```java
public Key select(int k)
{
    if (k < 0) return null;
    if (k >= size()) return null;
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k)
{
    if (x == null) return null;
    int t = size(x.left);
    if (t > k)
        return select(x.left, k);
    else if (t < k)
        return select(x.right, k-t-1);
    else if (t == k)
        return x;
}
```
Binary Search Trees

• Definitions and terminologies
• Search and insert
• Traversal
• Ordered operations

• Delete
  – Delete minimum and maximum
  – Delete a given key
Delete minimum
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.

```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.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]
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.
Hibbard deletion

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

Case 1. [1 child]
Hibbard deletion

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

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

[Diagram showing the deletion process]
Hibbard deletion

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

Case 2. [2 children]
Hibbard deletion

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.

---

**Diagram:**

- **Node to delete:**
  - Search for key E.
  - Go right, then go left until reaching null left link.

- **Successor:**
  - $\text{min}(t.\text{right})$.

- **Delete:**
  - $t.\text{left}$.
  - $\text{deleteMin}(t.\text{right})$.
  - Update links and node counts after recursive calls.

- **x has no left child**
- **but don’t garbage collect x**
- **still a BST**
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 (!) $\Rightarrow \sqrt{N}$ 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></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>
</tbody>
</table>

*other operations also become √N if deletions allowed*
Binary Search Trees

- Definitions and terminologies
- Search and insert
- Traversal
- Ordered operations
- Delete

- Balanced search trees (Amy Shannon)