3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

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.

Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.
Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

insert G

BST representation in Java

Java definition. A BST is a reference to a root Node.

A Node is composed of four fields:
- A Key and a Value.
- A reference to the left and right subtree.

smaller keys larger keys

Key and Value are generic types; Key is Comparable

BST implementation (skeleton)

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

    private class Node {
        /* see previous slide */
        private Key key;
        private Value val;
        private Node left, right;
        public Node(Key key, Value val) {
            this.key = key;
            this.val = val;
        }
    }

    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 Iterator<Key> iterator() {
        /* see next slides */
    }
}

BST search: Java implementation

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

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;
}

Cost. Number of compares is equal to 1 + 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**

**Put.** Associate value with key.

```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;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.

**Tree shape**

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.

**BST insertion: random order visualization**

**Ex.** Insert keys in random order.

Bottom line. Tree shape depends on order of insertion.
Sorting with a binary heap

Q. What is this sorting algorithm?

A. It’s not a sorting algorithm (if there are duplicate keys)!

Q. OK, so what if there are no duplicate keys?
Q. What are its properties?

Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1–1 if array has no duplicate keys.

BSTs: mathematical analysis

Proposition. If \( N \) distinct keys are inserted into a BST 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 \( N \) distinct keys are inserted in random order, expected height of tree is \( \sim 4.311 \ln N \).

But... Worst-case 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>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>( N )</td>
<td>( N )</td>
<td>( \frac{1}{2} N )</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td>equals()</td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( N )</td>
<td>( \lg N )</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( N )</td>
<td>1.39 ( \lg N )</td>
</tr>
</tbody>
</table>
### 3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

#### Minimum and maximum

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

#### Floor and ceiling

**Floor.** Largest key \( \leq \) a given key.
**Ceiling.** Smallest key \( \geq \) a given key.

#### Computing the floor

**Case 1.** \([k \text{ equals the key in the node}]

The floor of \( k \) is \( k \).

**Case 2.** \([k \text{ is less than the key in the node}]

The floor of \( k \) is in the left subtree.

**Case 3.** \([k \text{ is greater than the key in the node}]

The floor of \( k \) is in the right subtree (if there is any key \( \leq k \) in right subtree); otherwise it is the key in the node.

Q. How to find the floor / ceiling?
### 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;
    return x;
}
```

### Rank and select

**Q.** How to implement `rank()` and `select()` efficiently?

**A.** 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.

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

private int size(Node x) {
    if (x == null) return 0;
    return x.count + size(x.left) + size(x.right);
}
```

### BST implementation: subtree counts

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

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 x.count;
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
}
```

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

**Easy recursive algorithm (3 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);
}
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.

### 3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

#### 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>$N$</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>

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

#### 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 (linked list)</td>
<td>$N$</td>
<td>$\frac{1}{2}N$</td>
<td>$N$</td>
<td>$\frac{1}{2}N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
<td>$\frac{1}{2}N$</td>
<td>$\frac{1}{2}N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$1.39 \lg N$</td>
<td>$1.39 \lg N$</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 search (but don't consider it equal in search).

Cost. \( \sim 2 \ln N' \) 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 (memory) 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.

```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;
}
```

**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.

**Case 1.** [1 child] Delete \( t \) by replacing parent link.
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.

```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 {
        // $x$ has no left child
        if (x.left == null) return x.right;
        if (x.right == null) return x.left;
        // $x$ is a leaf
        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

*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 ops?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>