3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- deletion
Minimum and maximum

**Minimum.** Smallest key in BST.

**Maximum.** Largest key in BST.

Q. How to find the min / max?
Floor and ceiling

**Floor.** Largest key in BST \( \leq \) query key.

**Ceiling.** Smallest key in BST \( \geq \) query key.

Q. How to find the floor / ceiling?
Floor in a BST demo

**Floor.** Find the largest key in a BST that is $\leq k$?

**floor of G**
**Floor in a BST demo**

**Floor.** Find the largest key in a BST that is \( \leq k \)?
Floor in a BST demo

**Floor.** Find the largest key in a BST that is $\leq k$?

floor of $G$

E

compare $G$ and $E$
(go right)

floor of $G$ can't be in left subtree;
floor is either $E$ or floor of $G$ in right subtree
Floor in a BST demo

Floor. Find the largest key in a BST that is $\leq k$?

floor of $G$

E

all keys in right subtree of $R$ are greater than $G$ 
⇒ compute floor of $G$ in left subtree
**Floor in a BST demo**

**Floor.** Find the largest key in a BST that is $\leq k$?

---

**floor of G**

E

[Diagram of a binary search tree (BST) with nodes A, C, E, H, M, R, S, X. The node G is highlighted with a red arrow pointing to it, indicating the floor of G. The node H is connected to the left of G, and all keys in the right subtree of H are greater than G. A red arrow goes from H to G, indicating to compare G and H (go left). The diagram also notes that all keys in the right subtree of H are greater than G, leading to the conclusion that the floor of G should be computed in the left subtree.]
Floor. Find the largest key in a BST that is $\leq k$?

floor of $G$
Computing the floor

Floor. Largest key in BST \( \leq k \) ?

Case 1. [ key in node \( x = k \) ]
The floor of \( k \) is \( k \).

Case 2. [ key in node \( x > k \) ]
The floor of \( k \) is in the left subtree of \( x \).

Case 3. [ key in node \( x < k \) ]
The floor of \( k \) can't be in left subtree of \( x \): it is either in the right subtree of \( x \) or it is the key in node \( x \).
Computing the floor

public Key floor(Key key)
{  return floor(root, key);  }

private Key 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);

  Key t = floor(x.right, key);
  if (t != null) return t;
  else return x.key;
}
Rank and select

Q. How to implement \texttt{rank()} and \texttt{select()} efficiently for BSTs?

A. In each node, store the number of nodes in its subtree.
BST implementation: subtree counts

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

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

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

```java
private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    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.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

number of nodes in subtree

ok to call when x is null

initialize subtree count to 1
Computing the rank

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

**Case 1.** [$k < \text{key in node}$]
- Keys in left subtree? \textit{count}
- Key in node? 0
- Keys in right subtree? 0

**Case 2.** [$k > \text{key in node}$]
- Keys in left subtree? \textit{all}
- Key in node. 1
- Keys in right subtree? \textit{count}

**Case 3.** [$k = \text{key in node}$]
- Keys in left subtree? \textit{count}
- Key in node. 0
- Keys in right subtree? 0
Rank

**Rank.** How many keys in BST < \( k \) ?

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);
}
```
Selection in a BST demo

**Select.** Find the key in a BST of rank $k$.

$\text{rank}(S, 3)$

![BST Diagram with subtree counts]
Selection in a BST demo

**Select.** Find the key in a BST of rank \( k \).

\[
\text{rank}(S, 3)
\]

compare 3 and 6 (go left)

keys of rank 0–5 are in left subtree  \( \Rightarrow \)  find key of rank 3 in subtree rooted at E
Selection in a BST demo

Select. Find the key in a BST of rank $k$.

$\text{rank}(S, 3)$
$\text{rank}(E, 3)$

compare 3 and 2  
(go right)

keys of rank 0–1 are in left subtree  $\Rightarrow$  
find key of rank 0 in subtree rooted at R
Selection in a BST demo

**Select.** Find the key in a BST of rank $k$.

- $\text{rank}(S, 3)$
- $\text{rank}(E, 3)$
- $\text{rank}(R, 0)$

keys of rank 0–1 are in left subtree ⇒ find key of rank 0 in subtree rooted at H
Selection in a BST demo

*Select.* Find the key in a BST of rank $k$.

- `rank(S, 3)`
- `rank(E, 3)`
- `rank(R, 0)`
- `rank(H, 0)`

0 keys in left subtree $\Rightarrow$
key of rank 0 in subtree rooted at H is H
### 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>$\log N$</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>$\log N$</td>
<td>$h$</td>
</tr>
<tr>
<td>rank</td>
<td>$N$</td>
<td>$\log N$</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>

$h = \text{height of BST}$

(order proportional to $\log N$ if keys inserted in random order)

Order of growth of running time of ordered symbol table operations
**ST implementations: summary**

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</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>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

Next lecture. **Guarantee** logarithmic performance for all operations.
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- deletion
## ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Key Interface</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 (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log $N$</td>
<td>$N$</td>
<td>$N$</td>
<td>log $N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>log $N$</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).

![Diagram of BST deletion]

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 $\tau$ containing key $k$.

**Case 0.** [0 children] Delete $\tau$ 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] 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.

```plaintext
node to delete

search for key E

go right, then go left until reaching null left link

successor min(t.right)

deleteMin(t.right)

t.left

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;
        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

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>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

Other operations also become $\sqrt{N}$ if deletions allowed.

Next lecture. **Guarantee** logarithmic performance for all operations.