3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
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.
Q: Which of the following are Binary Search Trees? Why?

(1)

(2)

(3)

(4)

(5)

(6)
## 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 (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$\frac{1}{2} N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$\lg N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$1.39 \lg N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
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 \( \leq \) a given key.

**Ceiling.** Smallest key \( \geq \) a given key.

Q. How to find the floor / ceiling?
Computing the floor

Case 1. [$k$ equals the key in the node]
The floor of $k$ is $k$.

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

Case 3. [$k$ 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.
Computing the floor

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;
}
→ rank(Key k): how many keys less than k?
→ select(int n): what key has rank n?

→ Q: what is the rank of ’S’?
RANK AND SELECT

→ rank(Key k): how many keys less than k?
→ select(int n): what key has rank n?

→ Q: what is the rank of ‘S’?  5 (5 keys less than ‘S’ in the tree)
→ **rank(\text{Key } k)**: how many keys less than \( k \)?
→ **select(int } n)**: what key has rank \( n \)?

![Diagram](image.png)

→ Q: what is the rank of ’S’? \( 5 \) (5 keys less than ’S’ in the tree)
→ Q: what key has rank 4?
→ rank(\text{Key } k): \text{ how many keys less than } k? \\
→ select(int n): \text{ what key has rank } n? \\

→ Q: what is the rank of 'S'? \hspace{1cm} 5 \text{ (5 keys less than 'S' in the tree)} \\
→ Q: what key has rank 4? \hspace{1cm} 'R' \text{ (4 keys less than 'R' in the tree)}
→ **rank(Key k):** how many keys less than k?

→ **select(int n):** what key has rank n?

→ **Q:** what is the rank of ’S’?

→ **Q:** what key has rank 4?

→ **Q:** what is the rank of ’Q’ (’Q’ is not in the tree)?
→ **rank**(Key \(k\)): how many keys less than \(k\)?

→ **select**(int \(n\)): what key has rank \(n\)?

→ **Q**: what is the rank of ‘S’? \(5\) (5 keys less than ‘S’ in the tree)

→ **Q**: what key has rank 4? ‘R’ (4 keys less than ‘R’ in the tree)

→ **Q**: what is the rank of ‘Q’ (‘Q’ is not in the tree)? \(4\) (4 keys in the tree are less than ‘Q’)

```
H
C
A

S
E
R

X
```
→ rank(Key k): how many keys less than k?
→ select(int n): what key has rank n?

→ Q: what is the rank of ‘S’? 5 (5 keys less than ‘S’ in the tree)
→ Q: what key has rank 4? ‘R’ (4 keys less than ‘R’ in the tree)
→ Q: what is the rank of ‘Q’ (‘Q’ is not in the tree)? 4 (4 keys in the tree are less than ‘Q’)
→ Q: what key has rank 7?
→ **rank**(Key k): how many keys less than k?
→ **select**(int n): what key has rank n?

→ **Q**: what is the rank of ’S’? 5 (5 keys less than ’S’ in the tree)
→ **Q**: what key has rank 4? ’R’ (4 keys less than ’R’ in the tree)
→ **Q**: what is the rank of ’Q’ (’Q’ is not in the tree)? 4 (4 keys in the tree are less than ’Q’)
→ **Q**: what key has rank 7? no key in the tree has this rank (no key has 7 keys smaller than it in the tree)
Rank and select

Q. How to implement $\text{rank}()$ and $\text{select}()$ efficiently?

A. In each node, we store the number of nodes in the subtree rooted at that node; to implement $\text{size}()$, return the count at the root.
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;
}
```

ok to call when x is null
initialize subtree count to 1
number of nodes in subtree
Rank

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

Easy recursive algorithm (3 cases!)

```
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);
}
```
**Select.** Find the key with rank $n$.

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

private Node select(Node x, int n) {
    if (x == null) return null;
    int t = size(x.left);
    if (t > n) return select(x.left, n);
    else if (t < n) return select(x.right, n-t-1);
    else return x;
}
```
Task: Process all nodes of the tree.

Purpose: To print all nodes, to add all nodes in a datastructure (e.g. queue), etc.

Three kinds of traversals:

→ **inorder**: for each node:
  1. traverse the left subtree
  2. **process the node**
  3. traverse the right subtree

→ **preorder**: for each node:
  1. **process the node**
  2. traverse the left subtree
  3. traverse the right subtree

→ **postorder**: for each node:
  1. traverse the left subtree
  2. traverse the right subtree
  3. **process the node**
Inorder traversal

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

Property. Inorder traversal of a BST yields keys in ascending order.
**BST: ordered symbol table operations summary**

<table>
<thead>
<tr>
<th></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 $\log N$ if keys inserted in random order)

**Worst case: $h = O(N)$**

**Order of growth of running time of ordered symbol table operations**
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
## 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>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\frac{1}{2} N$</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>$1.39 \log N$</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).

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.
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.

- $x$ 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>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>$\frac{1}{2} N$</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 $\sqrt{N}$ if deletions allowed.

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