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`.

----

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 */
    }
}
```

---

Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

**successful search for H**

- **compare H and S** (go left)
- **black nodes could match the search key**

##
Binary search tree operations

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

successful search for \( H \)

\[
\begin{array}{c}
S \\
H \\
E \\
A \\
C \\
R \\
M \\
X \\
\end{array}
\]

compare \( H \) and \( E \)
(go right)

successful search for \( H \)

\[
\begin{array}{c}
S \\
H \\
E \\
A \\
C \\
R \\
M \\
X \\
\end{array}
\]

compare \( H \) and \( R \)
(go left)

successful search for \( H \)

\[
\begin{array}{c}
S \\
H \\
E \\
A \\
C \\
R \\
M \\
X \\
\end{array}
\]
Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

- Successful search for H
- Unsuccessful search for G
Binary search tree operations

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

unsuccessful search for G
Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

---

Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

---

Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

---

Binary search tree operations

**Search.** If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G

---
**Binary search tree operations**

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

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

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

- **Insert.** If less, go left; if greater, go right; if null, insert.
Binary search tree operations

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

Insert G

Binary search tree operations

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

Insert G

Binary search tree operations

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

Insert G

Binary search tree operations

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

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

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

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

Insert. If less, go left; if greater, go right; if null, insert.
**Get.** Return value corresponding to given key, or null if no such key.

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

**BST search**

### Get

- **Successful search for R**
  - Black nodes could match the search key
  - R is less than S so look to the left
  - Found R (search hit)
  - So return value

- **Unsuccessful search for T**
  - Gray nodes cannot match the search key
  - T is greater than S so look to the right
  - Link is null so T is not in tree (search miss)

**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         x.val = val;
   return x;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.
**BST trace: standard indexing client**

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

**Tree shape**
- Remark. Tree shape depends on order of insertion.

**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)

---

**How Tall is a Tree?**

Bj"orn Reed
THI, P"arls, France
reed@friola.stt.jussieu.fr

**ABSTRACT**

Let $H$ be the height of a random binary search tree on $n$ elements. It is known that $H \leq 2 \log_{\log n} n$ with probability $0.999$. From this we derive $\Pr[H \geq n / \log 2] = 0.01$. We also show that $\Pr[H = n] = o(1)$.

---

**ST implementations: summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</td>
<td>$N$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$\lg N$</td>
<td>$N/2$</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>$1.39 \lg N$</td>
<td>$1.39 \lg N$</td>
</tr>
</tbody>
</table>

---

**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 ≤ 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

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;
}
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.

![Subtree counts](image)

**Remark.** This facilitates efficient implementation of \texttt{rank()} and \texttt{select()}.

### BST implementation: subtree counts

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

```java
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;  }
```

Rank

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

Easy recursive algorithm (4 cases!)

```java
public int rank(Key key)
{  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;
```

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;  }
```
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.

---

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>log 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>l</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>l</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 = height of BST (proportional to log N if keys inserted in random order)

---

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 Iteration?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N/2</td>
<td>N2</td>
<td>no equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N/2</td>
<td>N/2</td>
<td>yes compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>yes 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. \( \sim 2 \ln N' \) 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.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.
**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: 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**

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: analysis**

Unsatisfactory solution. Not symmetric.

If we always delete from the same side, the shape of tree will be not random, the right subtrees are trimmed!

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>log N</td>
<td>N</td>
<td>N</td>
<td>log 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 log N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*other operations also become √N if deletions allowed*

**Red-black BST.** Guarantee logarithmic performance for all operations.