3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
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.
**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.
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 demo

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

---

successful search for H

```
        S
       /\   /
      E  X
     / \ / \/
    A  R  H
   / \      /
  C   M    
   
```
Binary search tree demo

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

insert G
BST search:  Java implementation

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

```java
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 diagram](image-url)

Insertion into a BST
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.

Remark. Tree shape depends on order of insertion.
BST insertion: random order visualization

Ex. Insert keys in random order.

N = 255
max = 16
avg = 9.1
opt = 7.0
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 $\sim 4.311 \ln N$.

**How Tall is a Tree?**

Bruce Reed  
CNRS, Paris, France  
reed@moka.ccr.jussieu.fr

**ABSTRACT**  
Let $H_n$ be the height of a random binary search tree on $n$ nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathbb{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\text{Var}(H_n) = O(1)$.

**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>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>
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
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. \([k \text{ equals the key at root}]\)
The floor of \(k\) is \(k\).

Case 2. \([k \text{ is less than the key at root}]\)
The floor of \(k\) is in the left subtree.

Case 3. \([k \text{ is greater than the key at root}]\)
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 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 size(), return the count at the root.

Remark. This facilitates efficient implementation of rank() and select().
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 size() {
    return size(root);
}

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

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
**Rank**

**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.
### 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>

*order of growth of running time of ordered symbol table operations*

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

- BSTs
- ordered operations
- deletion
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 iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</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>

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;
}
```
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.
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**

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.

![Tree diagram with statistics]

- \( N = 150 \)
- \( \text{max} = 16 \)
- \( \text{avg} = 9.3 \)
- \( \text{opt} = 6.4 \)

**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 (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td></td>
<td>search hit</td>
<td>insert</td>
<td>delete</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>no</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>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td></td>
<td>search hit</td>
<td>insert</td>
<td>delete</td>
<td>N/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td>search hit</td>
<td>insert</td>
<td>delete</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Other operations also become √N if deletions allowed.

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

- BSTs
- ordered operations
- deletion
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion