3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion
• 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.
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

**Binary search tree**

- BST with smaller keys
- BST with larger keys
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 */
    }

}
3.2 Binary Search Tree Demo

Click to begin demo
**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.
**Put.** Associate value with key.

Search for key, then two cases:
- Key in tree $\Rightarrow$ reset value.
- Key not in tree $\Rightarrow$ 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.

concise, but tricky, recursive code; read carefully!
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.
**Ex.** Insert keys in random order.
**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></td>
<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>
<td>N/2</td>
</tr>
<tr>
<td></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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.** 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.** \([k \text{ equals the key at root}]\)
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

```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;
    else           return x;
}
```

finding floor(G)

- G is less than S so floor(G) must be on the left
- G is greater than E so floor(G) could be on the right
- floor(G) in left subtree is null
- result
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()`.
public int size() {
    return size(root);
}

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

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

Easy recursive algorithm (4 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

**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;
}

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;
    return x;
}
```

Finding `select(3)`

The key of rank 3

Count N = 8

8 keys in left subtree so search for key of rank 3 on the left

2 keys in left subtree so search for key of rank 0 on the right

2 keys in left subtree so search for key of rank 0 on the left

0 keys in left subtree and searching for key of rank 0 so return H
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);
}
```
Inorder traversal

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

```plaintext
function call stack
inorder(S)
inorder(E)
inorder(A)
enqueue A
inorder(C)
enqueue C
enqueue E
inorder(R)
inorder(H)
enqueue H
inorder(M)
enqueue M
enqueue R
enqueue S
inorder(X)
enqueue X
```

```
recursive calls
queue
function call stack
S
S E
S E A
S E A C
S E R
S E R H
S E R H M
S X
```
## 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>1</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>

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

**order of growth of running time of ordered symbol table operations**
› 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>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>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 searches (but don't consider it equal to search key).

![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 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.

---

 deleting R

 node to delete

 replace with child link

 available for garbage collection

 update counts after recursive calls

 5

 7
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.

\[\text{node to delete}\]

\[\text{search for key E}\]

\[\text{t has no left child}\]

\[\text{but don't garbage collect x}\]

\[\text{still a BST}\]
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;  

        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: 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 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>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.

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