3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration
- deletion (see book)
3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration
- 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.
**Binary search tree demo**

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

**successful search for H**

![Binary search tree with successful search for H](image-url)
**Binary search tree demo**

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

**insert G**
java definition. A BST is a reference to a root Node.

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

```
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 Iterable<Key> iterator()
    { /* see slides in next section */ }

    public void delete(Key key)
    { /* see textbook */ }
}

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 = 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.

**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;
}
```

*Warning: concise but tricky code; read carefully!*

**Cost.** Number of compares = 1 + depth of node.
Tree shape

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

**Bottom line.** Tree shape depends on order of insertion.
BST insertion: random order visualization

**Ex.** Insert keys in random order.

![Diagram of a binary search tree with metrics: N = 255, max = 16, avg = 9.1, opt = 7.0.](image)
Binary search trees: quiz 1

Given $N$ distinct keys, what is the name of this sorting algorithm?

1. **Shuffle** the keys.
2. **Insert** the keys into a BST, one at a time.
3. Do an **inorder traversal** of the BST.

**A.** Insertion sort.

**B.** Mergesort.

**C.** Quicksort.

**D.** *None of the above.*

**E.** *I don't know.*
Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1–1 if array has no duplicate keys.
**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 into a BST in random order, the expected height 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 $E(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\text{Var}(H_n) = O(1)$.

---

**But...** Worst-case height is $N - 1$.

[ 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>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>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log $N$</td>
<td>$N$</td>
<td>log $N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>log $N$</td>
</tr>
</tbody>
</table>

**Why not shuffle to ensure a (probabilistic) guarantee of log $N$?**
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- deletion
Binary search trees: quiz 2

In what order does the `traverse(root)` code print out the keys in the BST?

```java
private void traverse(Node x) {
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```

A. A C E H M R S X
B. A C E R H M X S
C. S E A C R H M X
D. C A M H R E X S
E. I don't know.
Inorder traversal

```plaintext
inorder(S)
  inorder(E)
    inorder(A)
      print A
    inorder(C)
      print C
done C
done A
print E
inorder(R)
inorder(H)
  print H
inorder(M)
  print M
done M
done H
print R
done R
done E
print S
inorder(X)
  print X
done X
done S
```

output: A C E H M R S X
Inorder traversal

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

```java
class Node {
    int key;
    Node left, right;
}
```

```java
class BinarySearchTree {
    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.
Level-order traversal of a binary tree.

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...

level order traversal:  S E T A R C H M
Q1. Given binary tree, how to compute level-order traversal?

```java
queue.enqueue(root);
while (!queue.isEmpty()) {
    Node x = queue.dequeue();
    if (x == null) continue;
    StdOut.println(x.item);
    queue.enqueue(x.left);
    queue.enqueue(x.right);
}
```

level order traversal: S E T A R C H M
Q2. Given level-order traversal of a BST, how to (uniquely) reconstruct BST?

Ex. SETARCHM
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?
**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 `rank()` and `select()` efficiently for BSTs?

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

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;
}
Computing the rank

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

**Case 1.** \([ k < \text{key in node} ]\)
No key in right subtree < \( k \);
some keys in left subtree < \( k \).

**Case 2.** \([ k > \text{key in node} ]\)
All keys in left subtree < \( k \);
the key in the node is < \( k \);
some keys in right subtree may be < \( k \).

**Case 3.** \([ k = \text{key in node} ]\)
All keys in left subtree < \( k \);
no key in right subtree < \( k \).
**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);
}
```
## BST: Ordered Symbol Table Operations Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential Search</th>
<th>Binary Search</th>
<th>BST (h = height of BST, proportional to log N if keys inserted in random order)</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>

**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><strong>log $N$</strong></td>
<td><strong>log $N$</strong></td>
<td>log $N$</td>
<td>log $N$</td>
</tr>
</tbody>
</table>

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