3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration
- 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.
Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H
**Binary search tree demo**

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

**insert G**

```
    S
   / 
 X   
/ 
E   R
/ 
A   C
/ 
G   M
```

---

**Play**

---
BST representation in Java

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;  // root of BST

    private class Node {
        /* see previous slide */
    }

    public void put(Key key, Value val) {
        /* see next slide */
    }

    public Value get(Key key) {
        /* see next slide */
    }

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

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

/* see textbook */
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 = 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**

1. **Put.** Associate value with key.
2. **Search.** For key, then two cases:
   - Key in tree ⇒ reset value.
   - Key not in tree ⇒ add new node.
3. **Create.** New node is created.
4. **Reset.** Links are reset on the way up.

**Diagram:***
- Inserting L:
  - Insertion process shown with new node L.
  - Links are reset as node L is inserted.

**Label:**
- Links: Red for new links, black for existing.

**Notes:**
- **BST check:** Ensures binary search tree properties are maintained.
- **Traversal:** In-order, pre-order, post-order.

**Conclusion:** Insertion in a BST preserves its ordered property.
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.

N = 255  
max = 16  
avg = 9.1  
opt = 7.0
Binary search trees: quiz 1

What is the expected number of compares to sort $n$ distinct keys using the following 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. $\sim n \lg n$
B. $\sim n \ln n$
C. $\sim 2n \lg n$
D. $\sim 2n \ln n$
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 into a BST in random order, the expected height is \( \sim 4.311 \ln n \).

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
In which order does `traverse(root)` print the keys in the BST?

A. A C E H M R S X
B. S E A C R H M X
C. C A M H R E X S
D. S E X A R C H M
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.

Property. Inorder traversal of a BST yields keys in ascending order.

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

class Tree {
    Node root;
    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);
    }
}
```
Running time

Property. Inorder traversal of a BST takes linear time.
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
Q2. Given the level-order traversal of a BST, how to (uniquely) reconstruct?

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 $\leq k$?

**Key idea.**
- To compute $\text{floor}(\text{key})$, search for key.
- On search path, must encounter $\text{floor}(\text{key})$ and $\text{ceiling}(\text{key})$. Why?
Computing the floor

public Key floor(Key key)
{  return floor(root, key, null);  }

private Key floor(Node x, Key key, Key best)
{
    if (x == null) return best;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return floor(x.left, key, best);
    else if (cmp > 0) return floor(x.right, key, x.key);
    else if (cmp == 0) return x.key;
}
Rank and select

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

**Select.** Key of rank \(k\).

**Q.** How to implement \(\text{rank}()\) and \(\text{select}()\) 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 < key?

Case 1. [ key < key in node ]
- Keys in left subtree? count
- Key in node? 0
- Keys in right subtree? 0

Case 2. [ key > key in node ]
- Keys in left subtree? all
- Key in node. 1
- Keys in right subtree? count

Case 3. [ key = key in node ]
- Keys in left subtree? count
- Key in node. 0
- Keys in right subtree? 0
Rank

**Rank.** How many keys < \textit{key}?

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></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>$n$</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>

$h = \text{height of BST}$

**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>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

Next week. Guarantee logarithmic performance for all operations.