3.2 Binary Search Trees

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

- BSTs
  - ordered operations
  - iteration
  - deletion (see book or videos)
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.
Which of the following properties hold?

A. If a binary tree is heap ordered, then it is symmetrically ordered.
B. If a binary tree is symmetrically ordered, then it is heap ordered.
C. Both A and B.
D. Neither A nor B.
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
 /   \
E     X
|     |
A     R
|     |
C     H
|     |
G     M
```
**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 slide in this section */
  }

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

  public Iterable<Key> keys() {
    /* 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.
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.
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 idea. 1–1 correspondence with quicksort partitioning.
Binary search trees: quiz 2

Suppose that you insert $n$ keys in random order into a BST. What is the expected height of the resulting BST?

A. $\sim \lg n$

B. $\sim \ln n$

C. $\sim 2 \lg n$

D. $\sim 2 \ln n$

E. $\sim 4.31107 \ln n$
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</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>log $n$</td>
<td>$n$</td>
<td>log $n$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></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

https://algs4.cs.princeton.edu
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

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.
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.
- ...
Q1. Given binary tree, how to compute level-order traversal?

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. $SEFTRACHM$
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- deletion

https://algs4.cs.princeton.edu
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 ≤ query key.

**Ceiling.** Smallest key in BST ≥ query key.
**Computing the floor**

**Floor.** Largest key in BST \( \leq \) query key.

**Key idea.**
- To compute \( \text{floor}(\text{key}) \), search for key.
- Both \( \text{floor}(\text{key}) \) and \( \text{ceiling}(\text{key}) \) must be on search path.
- Moreover, as you go down search path, any candidates get better and better.

![Diagram of BST with floor(G) and floor(l) highlighted]
Computing the floor: Java implementation

**Invariant 1.** The floor is either champ or in subtree rooted at x.
**Invariant 2.** Node x is in the right subtree of node containing champ.

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

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

- champ must be floor
- key in node is too large (floor can’t be in right subtree)
- key in node is a candidate for floor (floor can’t be in left subtree)
- key in node is better candidate than champ
Rank and select

**Rank.** How many keys $< key$?

**Select.** Key of rank $k$.

**Q.** How to implement $\text{rank}(\cdot)$ and $\text{select}(\cdot)$ efficiently for BSTs?

**A.** In each node, store the number of nodes in its subtree.
private class Node
{
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int 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;
}

public int size()
{
    return size(root);
}

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

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

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.