3.2 Binary Search Trees

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

- BSTs
  - iteration
  - ordered operations
  - 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.
## Differences between heaps and binary search trees

<table>
<thead>
<tr>
<th>Supported operations</th>
<th>Heap</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert, delete-max</td>
<td>Insert, delete-max</td>
<td>insert, <strong>search</strong>, delete, ordered operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What is inserted</th>
<th>Keys</th>
<th>Key-value pairs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Underlying data structure</th>
<th>Resizing array</th>
<th>Linked nodes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Tree shape</th>
<th>Fixed shape given $n$ (complete binary tree)</th>
<th>Varies; depends on data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Ordering of keys</th>
<th>Somewhat ordered parent &gt; child for max heap</th>
<th>Totally ordered left child &lt; parent &lt; right child</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Duplicate keys allowed?</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
</table>

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.
A BST contains 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.

Key and Value are generic types; Key is Comparable.
BST implementation (skeleton)

```java
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 slide */
    }

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

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

    public void delete(Key key) {
        /* see textbook */
    }
}
```
BST search (get)

If less, go left; if greater, go right; if equal, search hit; if null node, search miss.

**H: search hit**

**G: search miss**
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 $\Rightarrow$ 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 = 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
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. \( \sim 4.31107 \ln n \)
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 \( \sim 4.31107 \ln n \).

But... Worst-case height is \( n - 1 \).
Unlike quicksort, worst case matters — client may not insert 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</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. G S A 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

```java
private void traverse(Node x)
{
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```
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.
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
Q. Given binary tree, how to compute level-order traversal?

level-order traversal: S E T A R C H M
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 \( \leq \) query key.

Key idea.

- To compute \( \text{floor}(\text{key}) \), search for \text{key}.
- Both \( \text{floor}(\text{key}) \) and \( \text{ceiling}(\text{key}) \) must be on search path. Why?
Computing the floor

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

```java
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;
}
```

- 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 best (x must be in right subtree of node containing best)
Rank and select

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

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

**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;
}

private int count(Node x) {
    if (x == null) return 0;
    return x.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;
}
Rank. How many keys < key?

key < key in node? Recur on left subtree.
key == key in node? Everything in left subtree.
key > key in node? Everything in left subtree + 1 + recursive result from right subtree.

```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);
}
```

Note: use size(x.left) instead of x.left.count to avoid null reference.
### 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.