3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration

https://algs4.cs.princeton.edu
3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration

https://algs4.cs.princeton.edu
Definition. A BST is a binary tree in symmetric order.

A binary tree is either:
- Empty.
- A node with links to two disjoint binary trees—the left subtree and the right subtree.

Symmetric order. Each node has a key; a node's key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.
- [Duplicate keys not permitted.]
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.
**Binary search tree demo**

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

**successful search for H**

```
  S
 /   \
E     X
 |     |
A     R
 |     |
C     H
 |     |
M
```
Binary search tree demo

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

```

insert G
```

```

```

```
**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 */
    }

}
**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 return x.val;
    }
    return null;
}
```

**Cost.** Number of compares = 1 + depth of node.
**BST insert**

**Put.** Associate value with key.
- Search for key in BST.
- Case 1: Key in BST ⇒ reset value.
- Case 2: Key not in BST ⇒ add new node.

```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 x.val = val;

    return x;
}
```

⚠️ Warning: concise but tricky code; read carefully!

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

---

**inserting L**

**search for L ends at this null link**

**create new node → L**

**reset links on the way up**

**insertion into a BST**

**inserting L**
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

Example. Insert 255 keys in random order.

$n = 255$
height = 13
average depth = 7.3
$\lg n = 8.0$
Binary search trees: quiz 2

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

A. $\sim \log_2 n$
B. $\sim 2 \ln n$
C. $\sim 4.31107 \ln n$
D. $\sim n$
### 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 (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 $\Theta (\log n)$ time?
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
Binary search trees: quiz 3

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

```
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. 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.

```java
public Iterable<Key> keys() {
    Queue<Key> queue = new Queue<Key>();
    inorder(root, queue);
    return queue;
}

private void inorder(Node x, Queue<Key> queue) {
    if (x == null) return;
    inorder(x.left, queue);
    queue.enqueue(x.key);
    inorder(x.right, queue);
}
```

Property. Inorder traversal of a BST yields keys in ascending order.
Property. Inorder traversal of a binary tree with \( n \) nodes takes \( \Theta(n) \) time.

Pf. \( \Theta(1) \) time per node in BST.
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. How to compute level-order traversal of a binary tree in $\Theta(n)$ time?
Q2. Given the level-order traversal of a BST, how to (uniquely) reconstruct?

Ex. $S E T A R C H M$
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
Minimum and maximum

**Minimum.** Smallest key in BST.

**Maximum.** Largest key in BST.

**Q.** How to find the min / max?

**A.** Go down left / right spine.

Running time proportional to depth of node in BST
Floor and ceiling

**Floor.** Largest key in BST $\leq$ query key.

**Ceiling.** Smallest key in BST $\geq$ query key.
Computing the floor

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

**Ceiling.** Smallest key in BST \( \geq \) query key.

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


**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 return x.key;
}
```

Examples of BST order queries

- min()
- max()

Key in node x is too large (floor can’t be in right subtree of x)

Key in node x is better candidate than champ

Key in BST champ must be floor

Key in node x is a candidate for floor (floor can’t be in left subtree of 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><strong>search</strong></td>
<td>$n$</td>
<td>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$n$</td>
<td>$n$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>min / max</strong></td>
<td>$n$</td>
<td>$1$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>floor / ceiling</strong></td>
<td>$n$</td>
<td>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>rank</strong></td>
<td>$n$</td>
<td>$\log n$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>select</strong></td>
<td>$n$</td>
<td>$1$</td>
<td>$h$</td>
</tr>
<tr>
<td><strong>ordered iteration</strong></td>
<td>$n \log n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

$h = \text{height of BST}$

order of growth of worst-case running time of ordered symbol table operations
## ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst case</th>
<th>Ordered ops?</th>
<th>Key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td></td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (sorted array)</td>
<td>$\log n$</td>
<td>$n$</td>
<td>✔️ compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>✔️ compareTo()</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>✔️ compareTo()</td>
</tr>
</tbody>
</table>

*next lecture: BST whose height is guarantee to be $\Theta(\log n)$*
A final thought

Dad, dad, look at that tree!!

It's just a tree, son...
But is a binary tree...

Daniel Stori (turnoff.us)