Node representation

```
private class Node {
    private Key key;                // sorted by key
    private Value val;              // associated data
    private Node left, right;       // left and right subtrees
    private int size;               // number of nodes in subtree

    public Node(Key key, Value val, int size) {
        this.key = key;
        this.val = val;
        this.size = size;
    }
}
```

Search - iterative implementation

```
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;
}
```

Search - recursive implementation

- ```
  public Value get(Key key) {
       return get(root, key);
  }
  ```
- ```
  private Value get(Node x, Key key) {
       if (x == null)
             return null;
       int cmp = key.compareTo(x.key);
       if (cmp < 0)
           return get(x.left, key);
       else if (cmp > 0)
           return get(x.right, key);
       else
           return x.val;
  }
  ```

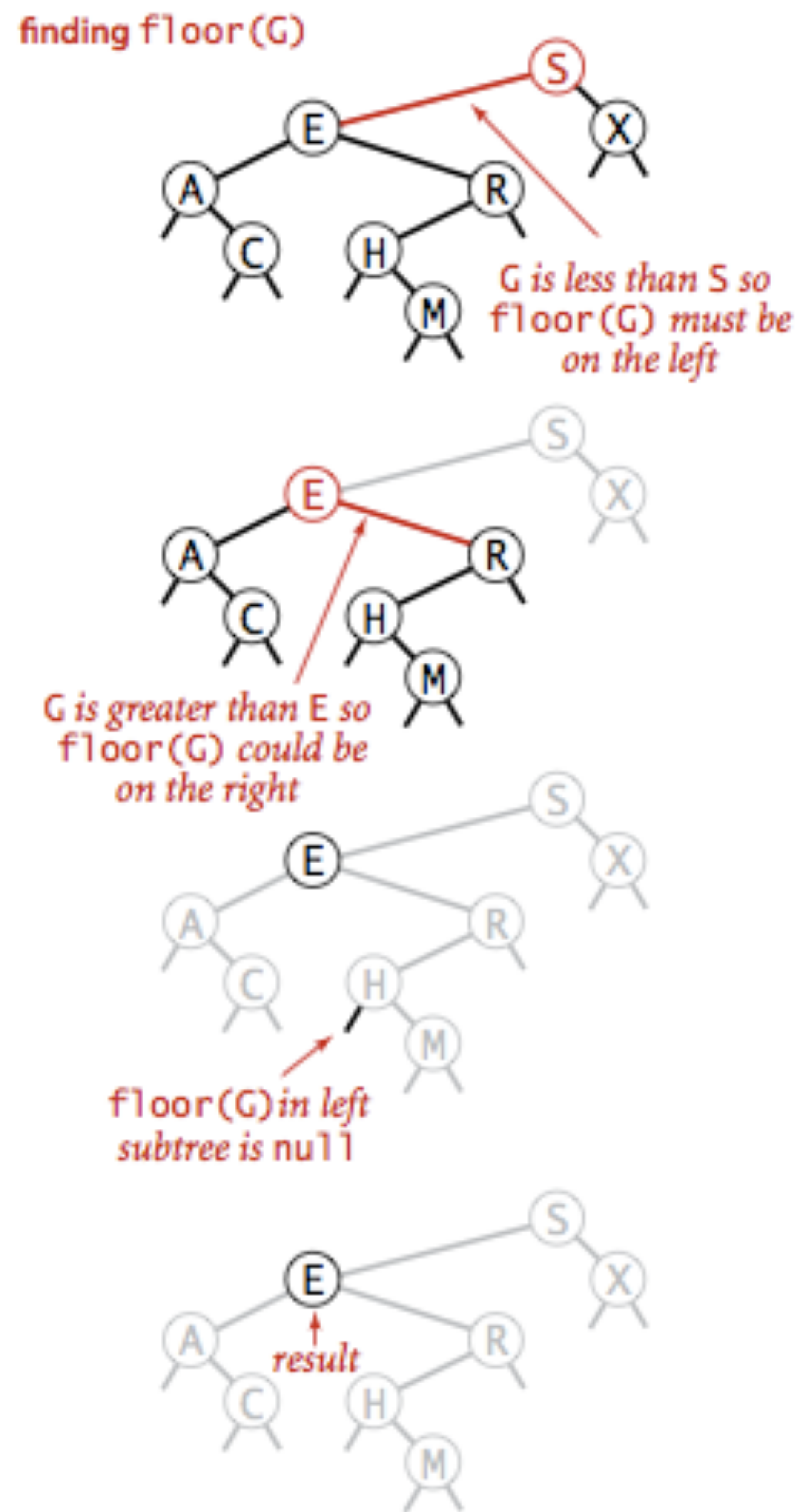
Insert

- ```
  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, 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
           x.val = val;
       x.size = 1 + size(x.left) + size(x.right);
       return x;
  }
  ```
  ```

Floor

- ```
  public Key floor(Key key) {
          Node x = floor(root, key);
          if (x == null)
              return null;
          else
              return x.key;
  }
  ```

- ```
  private Node 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);
          Node t = floor(x.right, key);
          if (t != null)
              return t;
          else
              return x;
      }
  ```



Rank

- Rank: How many keys < query key k.

```
public int rank(Key key) {
   return rank(key, root);
}

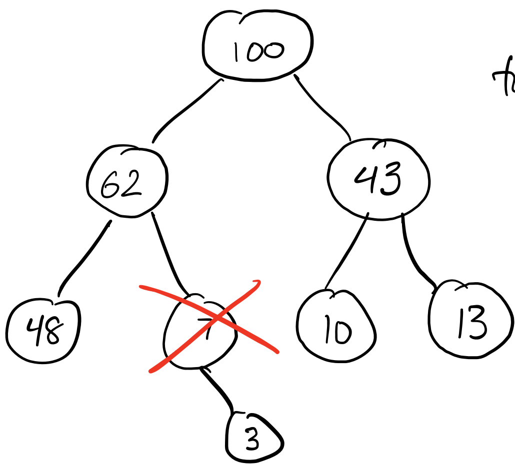
// Number of keys in the subtree less than key.
private int rank(Key key, Node x) {
    if (x == null)
        return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return rank(key, x.left);
    else if (cmp > 0)
        return 1 + size(x.left) + rank(key, x.right);
    else
        return size(x.left);
}
```

```
public void delete(Key key) {
    root = delete(root, key);
}

 private Node delete(Node x, Key key) {
     if (x == null) return null;

     int cmp = key.compareTo(x.key);
     if (cmp < 0)
         x.left  = delete(x.left,  key);
     else if (cmp > 0)
         x.right = delete(x.right, key);
     else {
         if (x.right == null)
             return x.left;
         if (x.left  == null)
             return x.right;
         Node t = x; //replace with successor
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
     }
     x.size = size(x.left) + size(x.right) + 1;
     return x;
 }
```

Tim Randolph
t.randolph@columbia.edu



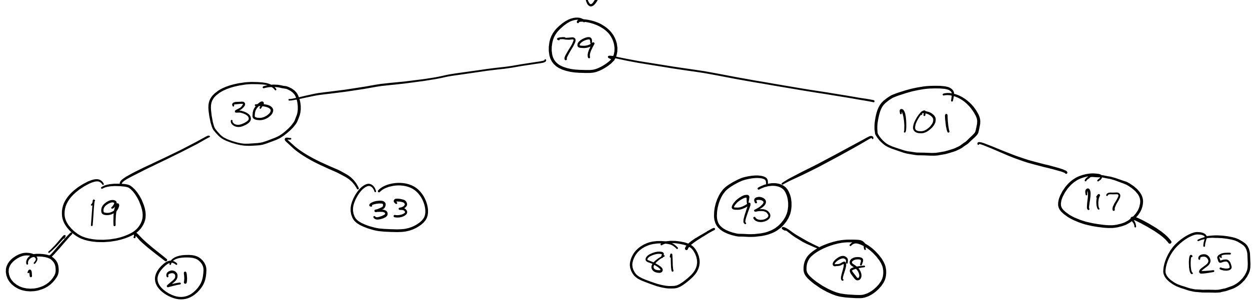
What's the best way to search a heap?
Say, for 10?

Today: Binary Search Trees

1. Basics
2. Search & Insert
3. "Ordered" Operations
4. Deletion

1. BST Basics

A Binary Search Tree (BST) is a binary tree in which every node's left subtree contains smaller keys, right subtree contains larger keys.



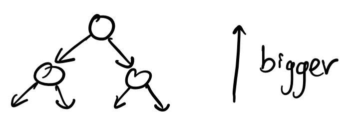
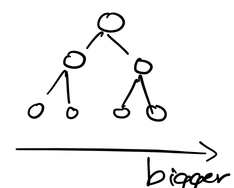
larger keys

- Node keys sorted small to large according to in-order traversal.
- Also known as symmetric order.

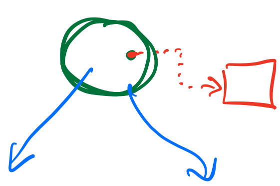
Main feature: very fast binary search!

\# comparisons ≈ tree height, h.

Trees vs. Heaps.

	(Max) Heap	BST
ordering:		
operations:	insert-max delete-max search slow ☹	search, insert, delete min, max, floor, ceiling, rank
tree shape:	flexible insert means tree stays complete	every node has one spot. ☹ ⟹ can lead to weird tree shape.
representation:	array OK	linked nodes. ☹

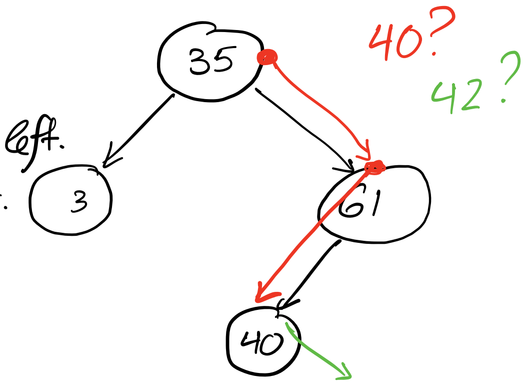
BST Nodes:



- key (comparable)
- value (data)
- left, right children
- size (# of descendants)

BST SEARCH:

1) Start at root.
2) If target key < current node key, go left.
   target key > current node, go right.
3) Repeat step 2 until we find target,
   or reach an empty leaf.



Runtime: height $O(h)$

→ # of comparisons required to locate target (or return null).

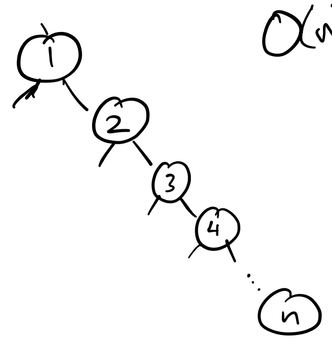
BST INSERT:

1) search for the target key
2) if you reach an empty leaf, add the new node.
   (if target key is already in BST, update value).

Runtime: $O(h)$

What's the real runtime?

- depends on tree shape.
- insert n keys in random order:
  - $2 \ln(n)$ compares/insertion (expected)
  - $4.3 \ln(n)$ tree height.
- worst-case:

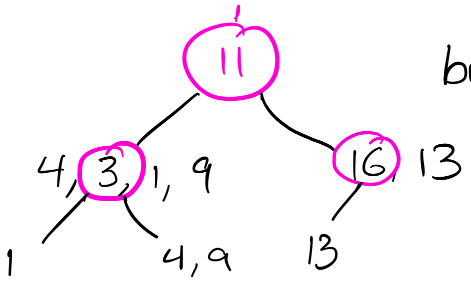


$O(n)$ compares/insert

Inserting BST elements in random order

≈ quicksort

Example: 16, 4, 11, 3, 13, 1, 9



building a BST.

Quicksort # pivots

= Expected BST height $= 2\ln(n)$

3. "Ordered" Operations

- find min, find max.

  (go all the way to the left/right).

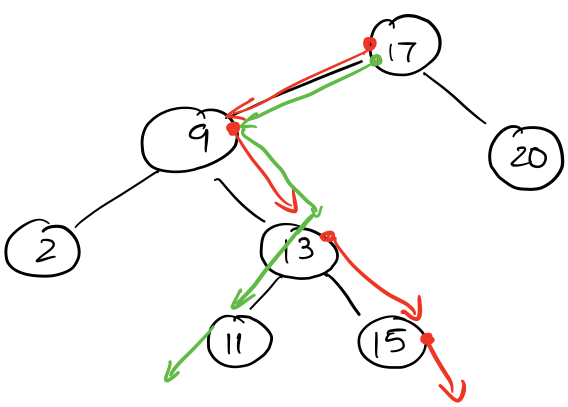
- find floor/ceiling of a key. $\lceil x \rceil$

  floor: what's the largest key of size at most k?

  ceiling: —— " — smallest key —— " — least k?

BST Floor/Ceiling:

1) search for k
2) if k isn't in our BST, find k's predecessor/successor.

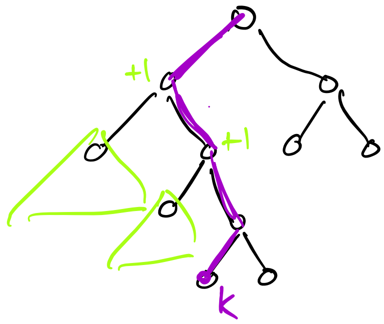


floor (16)

- search
- last parent 15

floor (10)

BST Rank: number of nodes with key smaller than target.

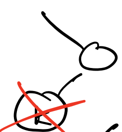


rank(k)

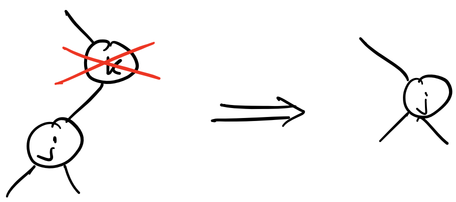
1) search for target k
2) on a "right turn":
   - add +1 to rank (for subtree root)
   - add size of left subtree to rank

BST Deletion:

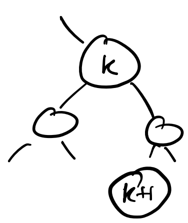
1) Search for k, our target
2) Delete k (3 cases):
   i) k is a leaf: remove.

   ii) k has one child: replace k w/ child.

   iii) k has two children:

   - find k's successor
   - replace k with succesor
   - (recursively) delete k's successor.