SEARCHING

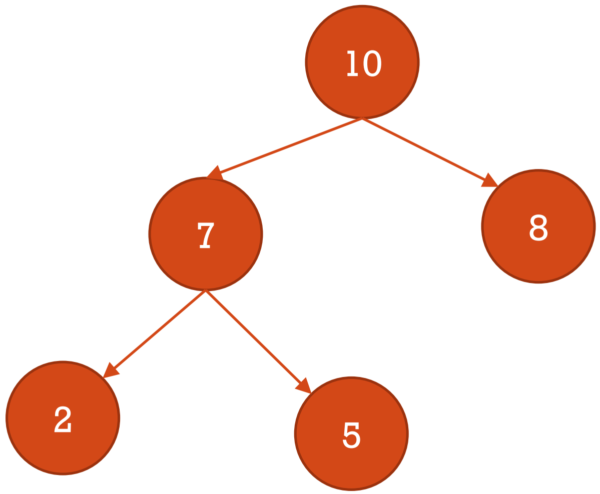
BINARY SEARCH TREES

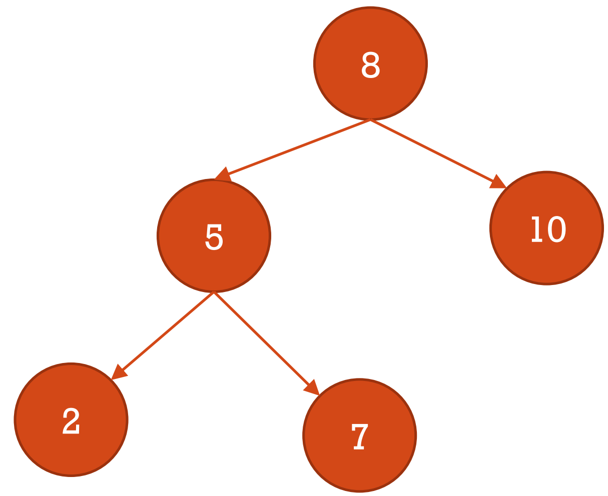
Ruben Acuña

Fall 2018

2 INTRODUCTION

HEAPS TO BINARY SEARCH TREES



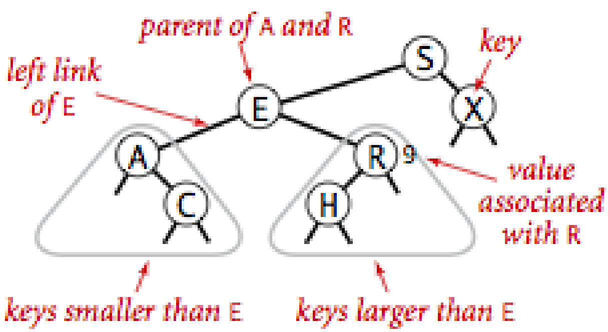


- In the last chapter, we discussed heaps as a way to structure data in a priority queue.
- Now we'll use BSTs which are a little more restrictive.
- Are heaps BSTs? Vice versa?

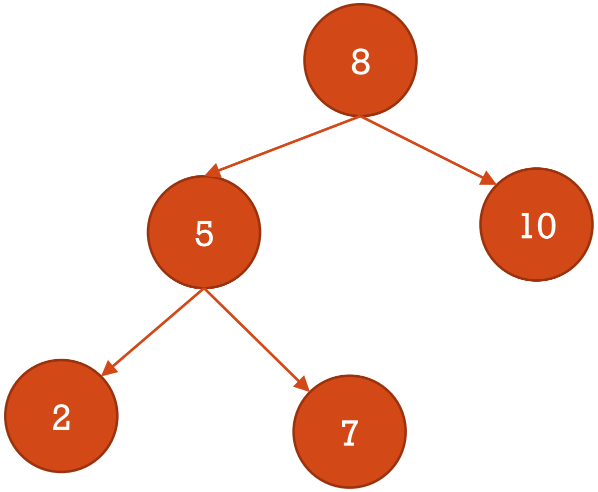
- Side note: one requirement of symbol tables is that each key maps to exactly one value. This useful in our implementations since we can assume the elements are distinct.

THE CONCEPT

- **Definition**: A *binary tree* is a tree where each node has at most two children.
- **Definition**: A binary search tree is a binary tree where each node's left child has a key less than the parent, and the right child has a key greater than the parent.



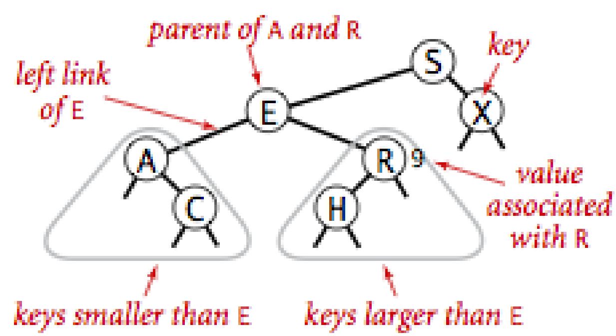
Anatomy of a binary search tree



THE CONCEPT

- **Recursive Definition**: a node is the root of a BST if:
  - A left child has a key less than than the parent, and the child is the root of a BST.
  - A right child has a key greater than the parent , and the child is the root of a BST.

- So… what?



Anatomy of a binary search tree

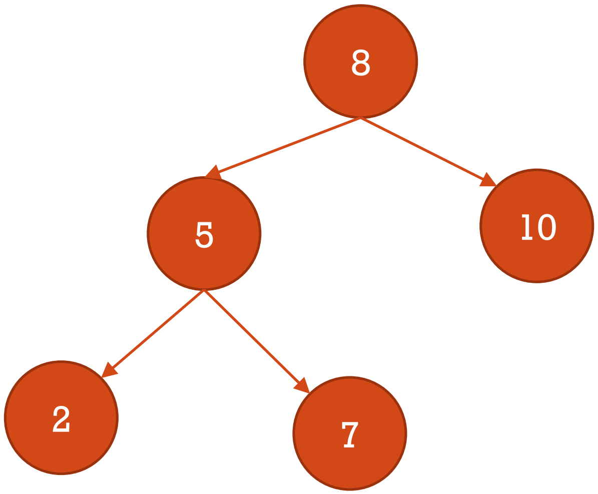
TRAVERSING TREES

- A common operation performed on trees is to visit all of their nodes. To explore a non-linear structure, we need rules to systematically explore the structure and be sure we don't miss any nodes.
  - How does this compare to exploring a linked list?

- There are three main algorithms for doing this:
  - **Preorder(root):**
    visit root, preorder(root.left), preorder(root.right)
  - **Inorder(root):**
    inorder(root.left), visit root, inorder(root.right)
  - **Postorder(root):**
    postorder(root.left), postorder(root.right), visit root

TRAVERSING TREES

- **Preorder(root):**
  visit root, preorder(root.left), preorder(root.right)



8 IMPLEMENTATION

ORDERED SYMBOL TABLE

We're going to implement the whole thing – a ordered symbol table:

public class ST<Key extends Comparable<Key>, Value>

	ST()	*create an ordered symbol table*
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is null)*
Value	get(Key key)	*value paired with* key *(null if* key *is absent)*
void	delete(Key key)	*remove* key *(and its value) from table*
boolean	contains(Key key)	*is there a value paired with* key*?*
boolean	isEmpty()	*is the table empty?*
int	size()	*number of key-value pairs*
Key	min()	*smallest key*
Key	max()	*largest key*
Key	floor(Key key)	*largest key less than or equal to* key
Key	ceiling(Key key)	*smallest key greater than or equal to* key
int	rank(Key key)	*number of keys less than* key
Key	select(int k)	*key of rank* k
void	deleteMin()	*delete smallest key*
void	deleteMax()	*delete largest key*
int	size(Key lo, Key hi)	*number of keys in* [lo..hi]
Iterable<Key>	keys(Key lo, Key hi)	*keys in* [lo..hi]*, in sorted order*
Iterable<Key>	keys()	*all keys in the table, in sorted order*

API for a generic ordered symbol table

Image from Sedgewick and Wayne.

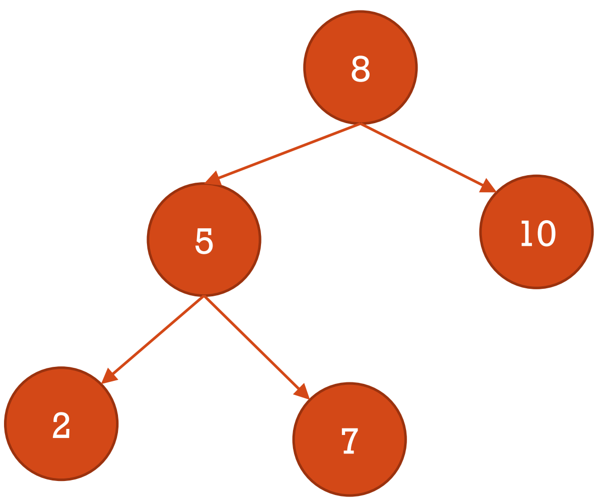
BST REPRESENTATION

- In the vernacular of the discussion we had during priority queues: we will represent our symbol table as a binary search tree and structure it as a binary tree.
- Each node will be a linked node with three references, a value, and a size.

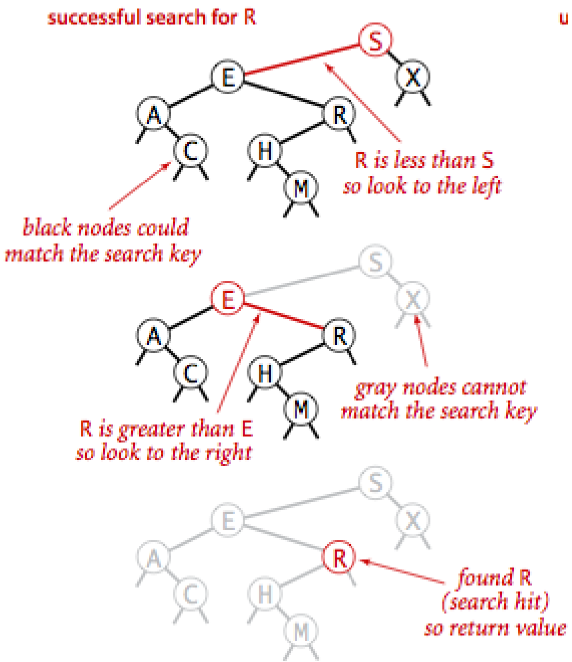
```
private class Node {
    private final Key key;
    private Value val;
    private Node left, right;
    private int N;

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

GET() IN BSTS

- Per the API, we need to get "value paired with key".



GET() IMPLEMENTATION

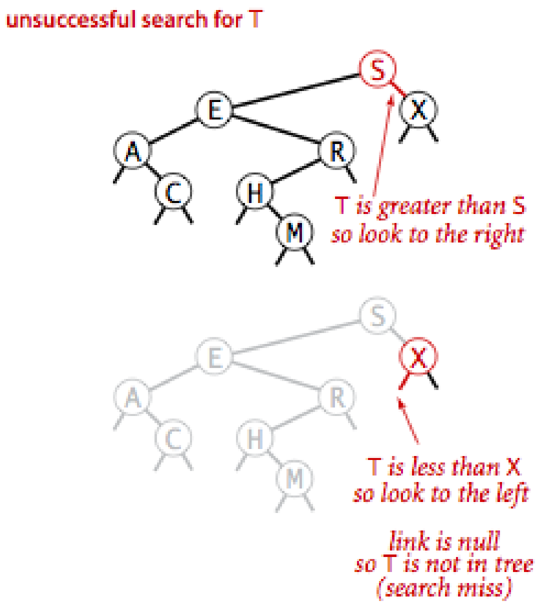


sucessful search

```
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;
    }
```


Image from Sedgewick and Wayne.

GET() IMPLEMENTATION



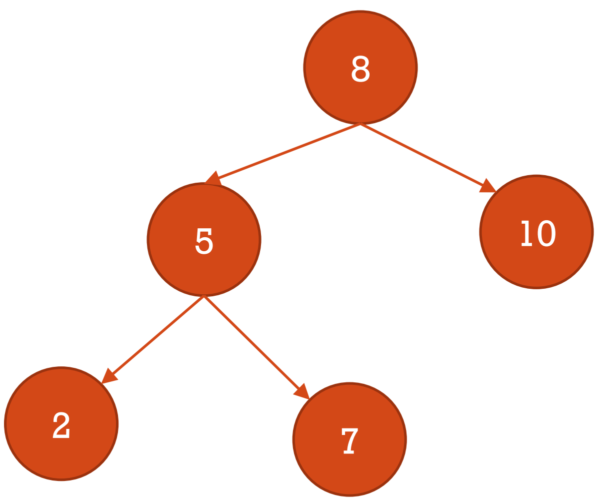
unsucessful search

```
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;
    }
```

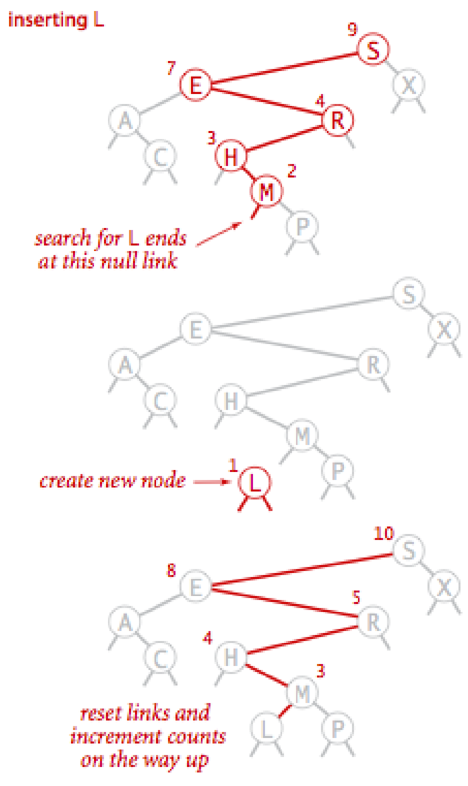

Image from Sedgewick and Wayne.

PUT() IN BSTS

- Per the API, we need to "put a key-value pair into the table".



PUT() IMPLEMENTATION

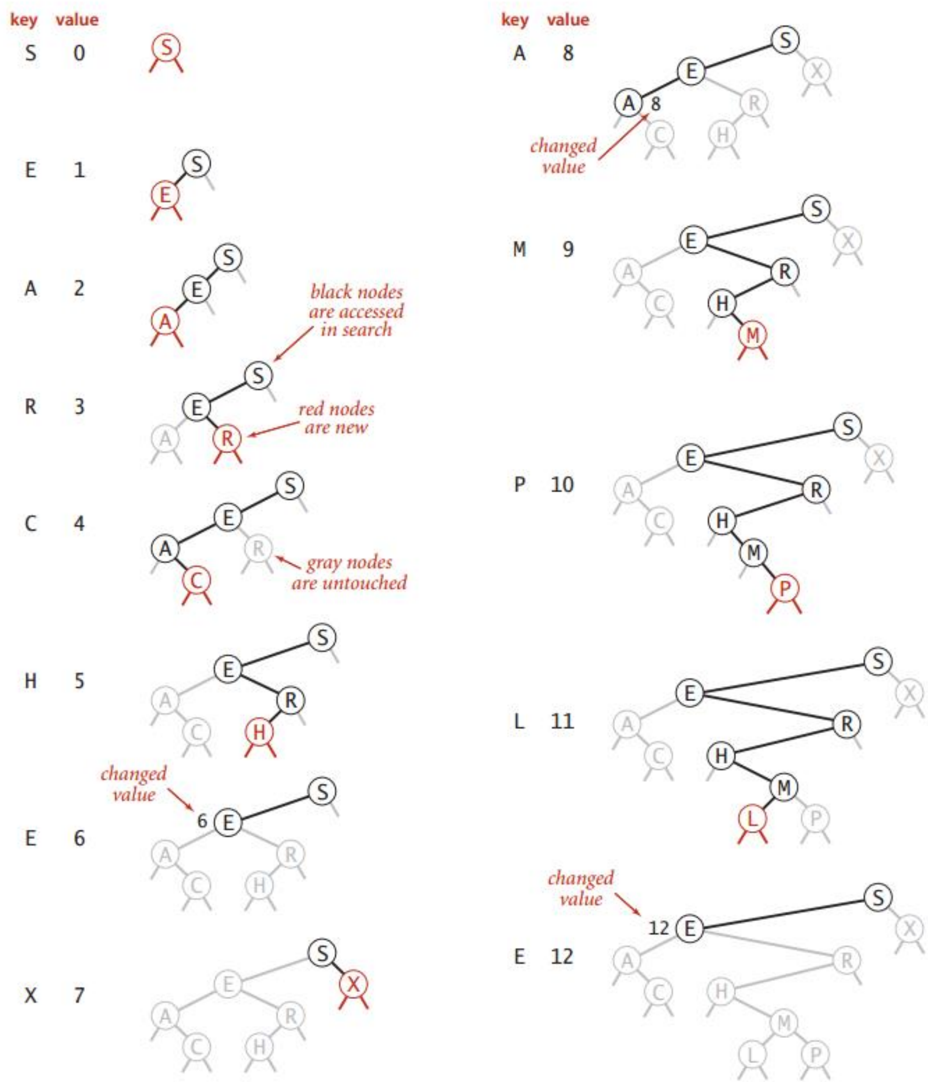


Insertion into a BST

```
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.N = size(x.left) + size(x.right) + 1;

    return x;
}
```


Image from Sedgewick and Wayne.

BST TRACE



One general concern: how will the resultant tree look?

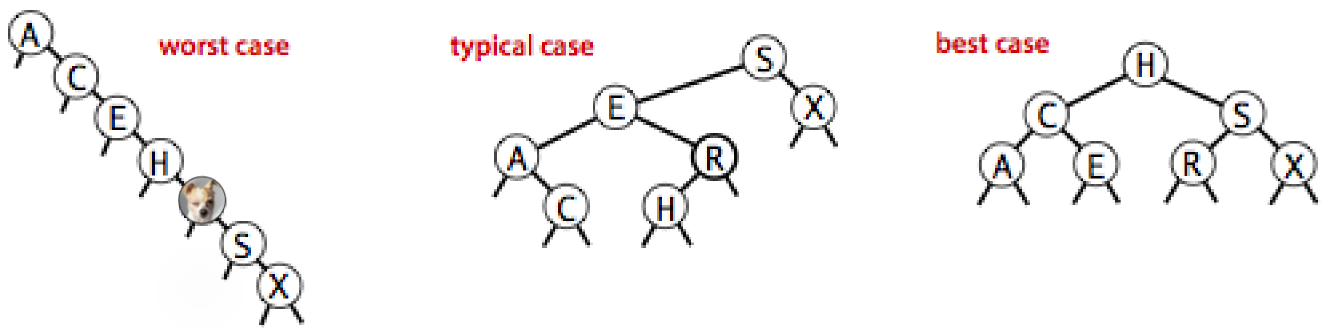
BST LAYOUT

There three cases for layout of a tree:

Worse case: a stilted tree.

Typical case: in between the other two cases.

Best case: a balanced tree.



We will focus on the typical (average) case for a successful search where we count comparisons.

- Where do unsuccessful searches and inserts fit in?
- Any guesses for the worse or best case Big-Oh right now?

The average case will be defined as a tree where the domain of keys is randomly inserted.

AVERAGE ANALYSIS FOR BST

Proof:

The sum of the depth of all nodes is the *internal path length*, call it C_N.

The average cost of a search hit is then:

- $1 + \frac{C_N}{N}$

As initial values, we have $C_0 = C_1 = 0$. We now want to write C_N:

- $N - 1$ (every node other than the root has to cross the root's edge)
- $((C_0 + C_{N-1}) + (C_1 + C_{N-2}) + \cdots (C_{N-1} + C_0))/N$ (take the average of possible subtrees)

The textbook rewrites as:

- $C_N = N - 1 + (C_0 + C_{N-1})/N + (C_1 + C_{N-2})/N + \cdots (C_{N-1} + C_0)/N$

a complete expression that looks like the quicksort recurrence.

We'll skip the algebra on this one:

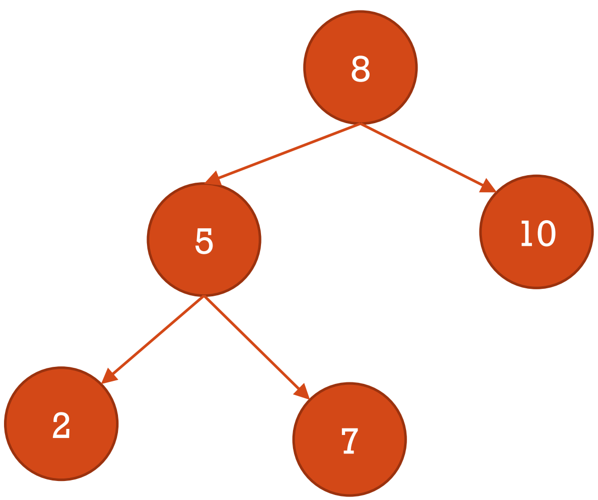
- $C_N \sim 2NlnN$

We can now evaluate our original expression, $1 + \frac{C_N}{N}$:

- $1 + \frac{2Nln}{N} \sim 2lnN \approx 1.39lgN$

MIN() IN BSTS

- Per the API, we need to find the "smallest key".



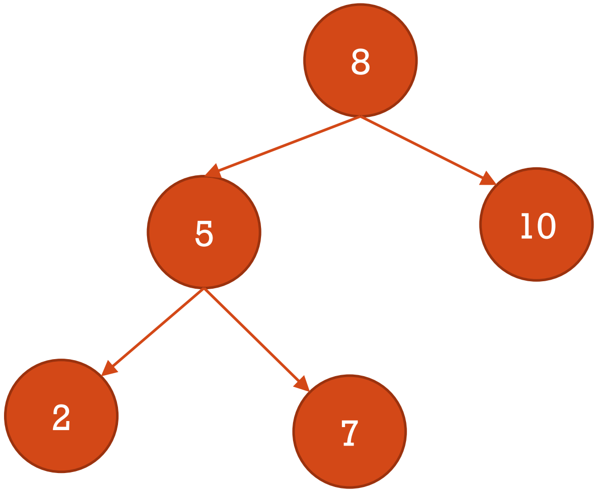
MIN() IMPLEMENTATION

- Finding the min is simple, just move left until you cannot do so.

```
public Key min() {
        return min(root).key;
    }

    private Node min(Node x) {
        if (x.left == null)
            return x;
        return min(x.left);
    }
```

DELETEMIN() IN BSTS

- Per the API, we need to “delete smallest key”.

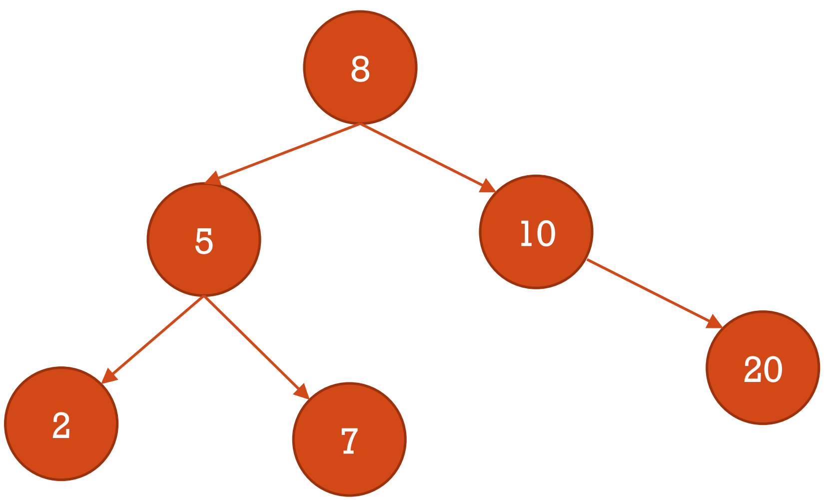


DELETEMIN() IMPLEMENTATION

```
public void deleteMin() {
        root = deleteMin(root);
    }

    private Node deleteMin(Node x) {
        if (x.left == null) return x.right;
        x.left = deleteMin(x.left);
        x.N = size(x.left) + size(x.right) + 1;
        return x;
    }
```

DELETE() IN BSTS

- Per the API, we need to "remove key (and value) from table".
- This will be our most complicated method – there are actually three different cases to handle (explicitly or implicitly).

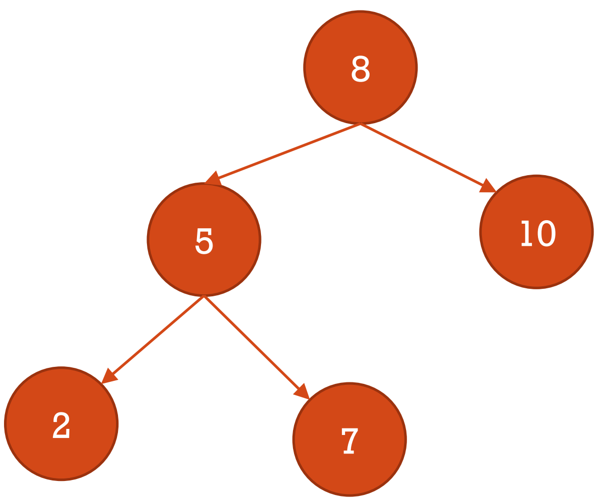


DELETE() IMPLEMENTATION

```
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;
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
      }
      x.N = size(x.left) + size(x.right) + 1;
      return x;
   }
```

FLOOR() IN BSTS

- Per the API, we need to "largest key less than or equal to key".



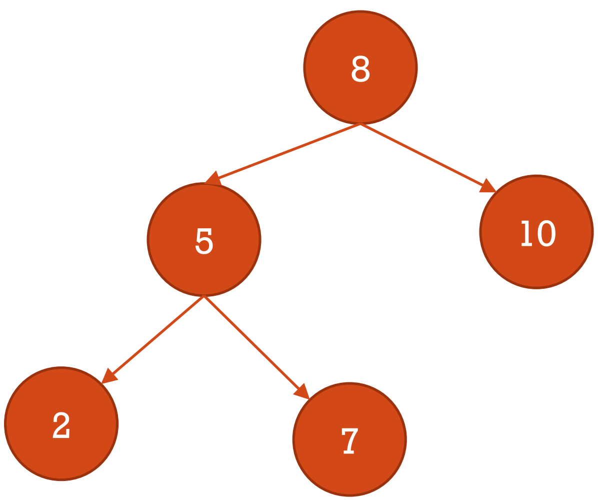
IMPLEMENTATION: FLOOR

- Floor is a little more tricky…

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

KEYS() IN BSTS

- Per the API, we need to find all the "keys in [lo..hi]".



KEYS() IMPLEMENTATION

```
public Iterable<Key> keys(Key lo, Key hi) {
    Queue<Key> queue = new LinkedList<>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
    if (x == null) return;
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) keys(x.left, queue, lo, hi);
    if (cmplo <= 0 && cmphi >= 0) queue.add(x.key);
    if (cmphi > 0) keys(x.right, queue, lo, hi);
}
```

PERFORMANCE SUMMARY

Algorithm (data structure)	avg: search hit	avg: insert	Efficiently support ordered operations?
sequential search (unordered linked list)	N/2	N	No
binary search (ordered array)	lg N	N	Yes
binary search trees	1.39lg N	1.39lg N	Yes