Dictionary:

insert (Key x, Value v)
- insert (x, v) in dict. (No duplicates)

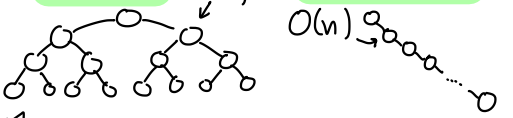
delete (Key x)
- delete x from dict. (Error if x not there)

find (Key x)
- returns a reference to associated value v, or **null** if not there.

Search: Given a set of n entries each associated with **key** x_i and **value** v_i
- Store for quick access + updates
- **Ordered:** Assume that keys are totally ordered: <, >, ==

Efficiency: Depends on tree's height

Balanced: O(log n) **Unbalanced:** O(n)

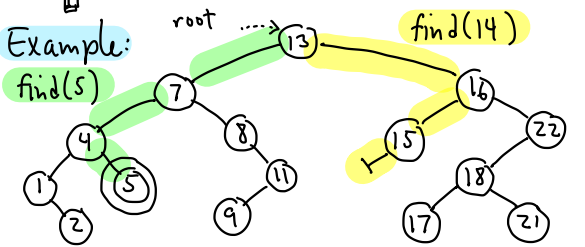


Example:

find(5)

find(14)

root

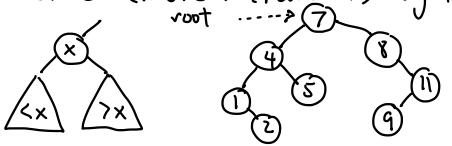


Binary Search Trees I
- Basic definitions
- Finding keys

Sequential Allocation?
- Store in array sorted by key
  - → **Find**: O(log n) by binary search
  - → **Insert/Delete**: O(n) time

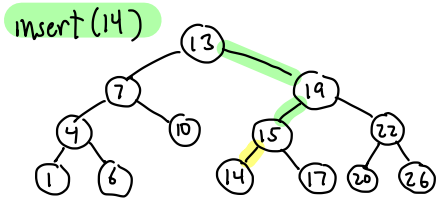
Can we achieve O(log n) time for all ops? ☆ Binary Search Trees ☆

Idea: Store entries in binary tree sorted (inorder traversal) by key



Find: How to find a key in the tree?
- Start at root p ← root
- if (x < p.key) search left
- if (x > p.key) search right
- if (x == p.key) found it!
- if (p == null) not there!

```
Value find(Key x, BSTNode p){
    if (p == null) return null
    else if (x < p.key)
        return find(x, p.left)
    else if (x > p.key)
        return find(x, p.right)
    else return p.value
}
```

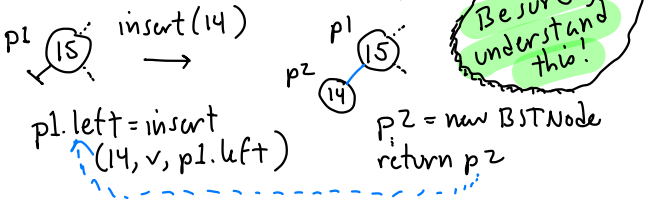


Insert (Key x, Value v)

- find x in tree
- if found ⇒ error! duplicate key
- else: create new node where we "fell out"

```
BSTNode insert(Key x, Value v, BSTNode p){
    if (p == null)
        p = new BSTNode(x,v)
    else if (x < p.key)
        p.left = insert(x,v, p.left)
    else if (x > p.key)
        p.right = insert(x,v, p.right)
    else throw exception → Duplicate!
    return p
}
```

Why did we do:
`p.left = insert(x,v, p.left)` ?



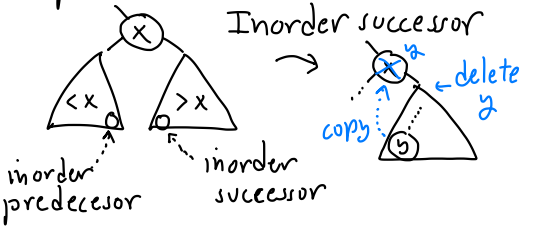
Binary Search Trees II
- insertion
- deletion

Delete (Key x)

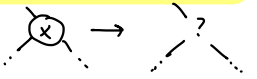
- find x
- if not found → error
- else: remove this node + restore BST structure

How?

Replacement Node?



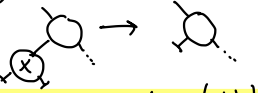
3. (x) has two children



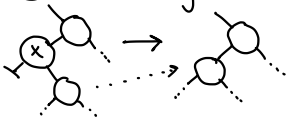
Find replacement node (y), copy to (x), and then delete (y)

3 cases:

① (x) is a leaf



② (x) has single child



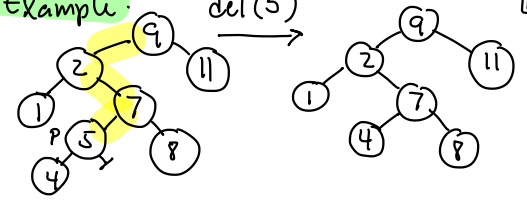
```
BSTNode delete(Key x, BSTNode p){
  if (p == null) Error! Key not found
  else
    if (x < p.key)
      p.left = delete(x, p.left)
    else if (x > p.key)
      p.right = delete(x, p.right)
    else if (either p.left or p.right null)
      if (p.left == null)
        return p.right
      if (p.right == null)
        return p.left
    else
      r = findReplacement(p)
      copy r's contents to p
      p.right = delete(r.key, p.right)
  return p
}
```

Find Replacement Node

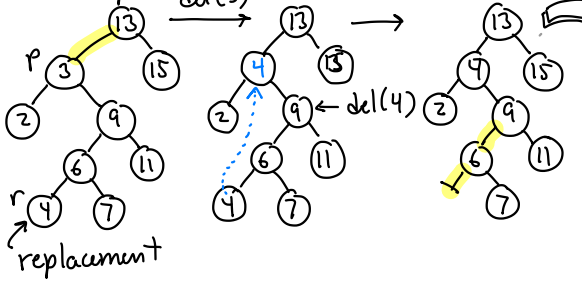
```
BSTNode findReplacement(BSTNode p){
  BSTNode r = p.right
  while (r.left ≠ null)
    r = r.left
  return r
}
```

Binary Search Trees III
- deletion
- analysis
- Java

Example: del(5)



Example:



Java Implementation:

- Parameterize Key + Value types: extends Comparable
  class BinSearchTree<K,V>{...
- BSTNode - inner class
- Private data: BSTNode root
- insert, delete, find: local
- provide public fns insert, delete, find

But height can vary from O(log n) to O(n)...

Expected case is good

Thm: If n keys are inserted in random order, expected height is O(log n).

Analysis:

All operations (find, insert, delete) run in O(h) time, where h = tree's height

Java implementation (see notes for details)

```
public class BSTree<Key extends Comparable, Value> {

    class Node {
        Key key
        Value value
        Node left, right

        .... constructor, toString...
    }

    Value find(Key x, Node p) {...}
    Node insert(Key x, Value v, Node p) {...}
    Node delete(Key x, Node p) {...}

    private Node root;

    public Value find(Key x) {...}
    public void insert(Key x, Value v) {...}
    public void delete(Key x) {...}

}
```

Inner class for node (protected)

Local helpers (private or protected)

Data (private)

Public members (invoke helpers)