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)

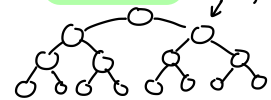
find(Key x)
- returns a reference to associated value v, or **null** if not there.

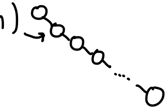
Search: Given a set of n **entries** each associated with **key** x_i and **value** v_i
- Store for quick access + updates
- **Ordered:** Assume that keys are totally ordered: <, >, ==

Binary Search Trees I
- Basic definitions
- Finding keys

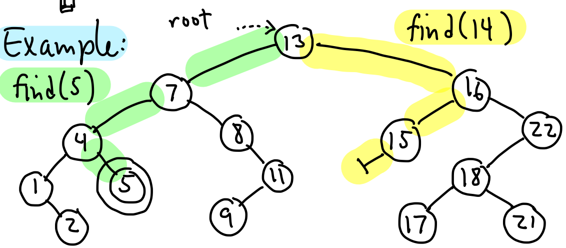
Efficiency: Depends on tree's height

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





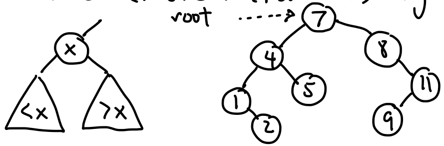
Example: **find(5)**



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



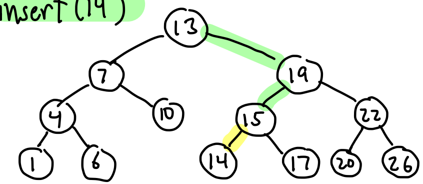
Find: How to find a key in the tree?
- Start at root p ← root
- if (x < p.key) search left
- if (x > p.key) search right
- if (x == p.key) found it!
- if (p == null) not there!

```
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
}
```

Binary Search Trees II
- insertion
- deletion

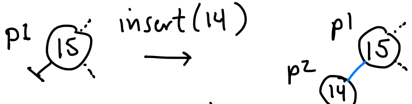
Insert (Key x, Value v)
- find x in tree
- if found ⇒ error! duplicate key
- else: create new node where we "fell out"

insert(14)



```
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)?



p1.left = insert(14, v, p1.left)

p2 = new BSTNode
return p2

Be sure you understand this!

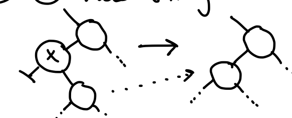
Delete (Key x)
- find x
- if not found → error
- else: remove this node + restore BST structure

How?

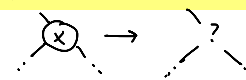
3 cases:

① ⓧ is a leaf

② ⓧ has single child

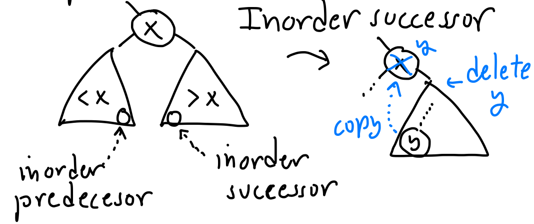


3. ⓧ has two children



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

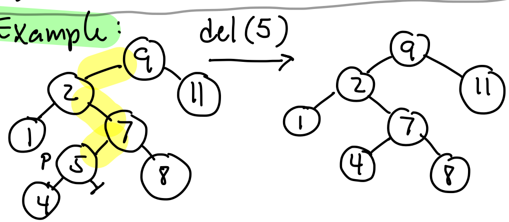
Replacement Node?



Binary Search Trees III
- deletion
- analysis
- Java

```
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
}
```

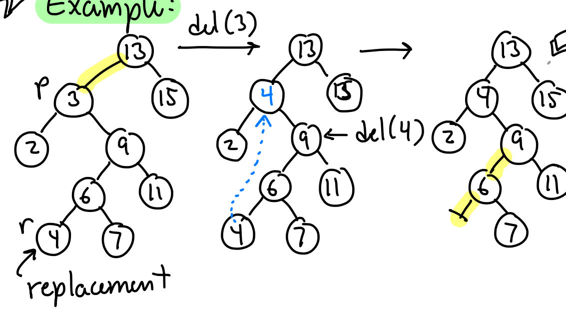
Example:



Find Replacement Node

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

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