Data Structures in Java

Lecture 9: Binary Search Trees.

10/7/2015

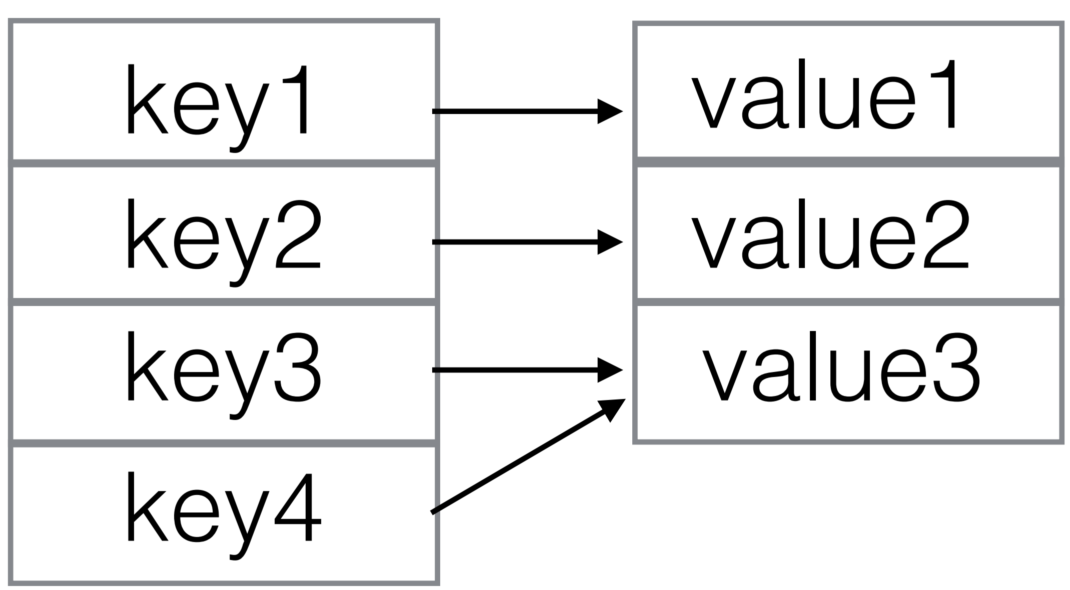
Daniel Bauer

Contents

1. **Binary Search Trees**
2. Implementing Maps with BSTs

Map ADT

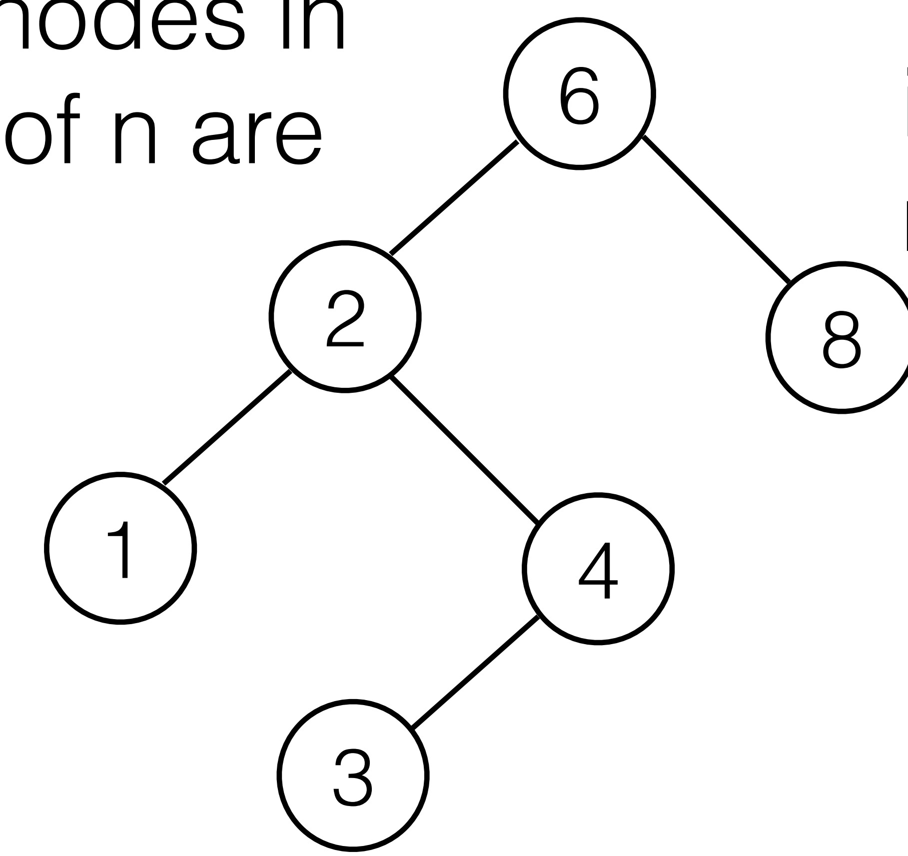
- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be.
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)



How do we implement map operations efficiently?

Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)
- For every node n with value x
- the value of all nodes in the left subtree of n are smaller than x.



- The value of all nodes in the right subtree of n are larger than x.

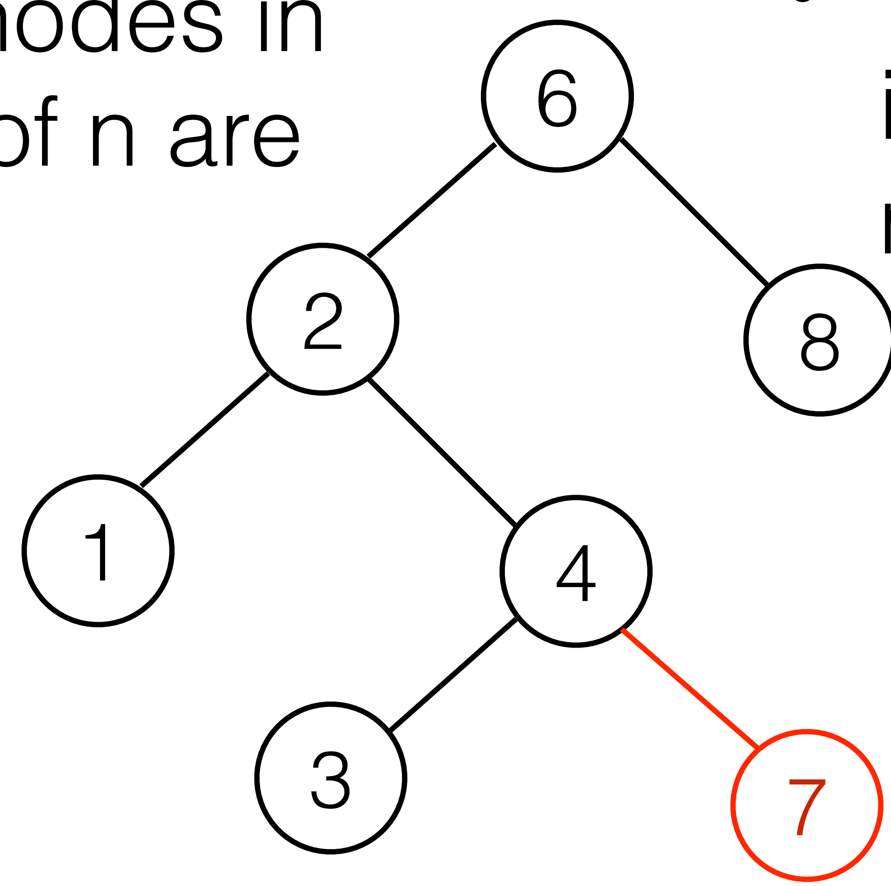
Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)

- For every node n with value x

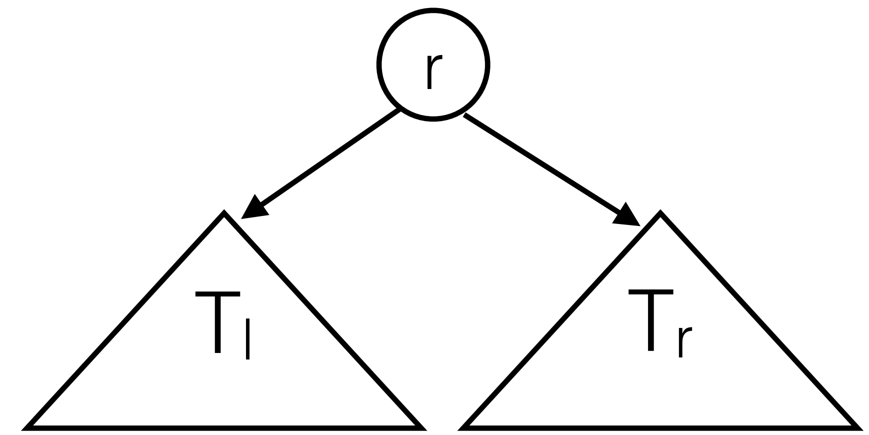
- the value of all nodes in the left subtree of n are smaller than x.

- The value of all nodes in the right subtree of n are larger than x.



This is not a search tree

Binary Search Tree (BST) ADT



- A *Binary Search Tree T* consists of
  - A root node *r* with value r_{item}
  - *A*t most two non-empty subtrees T_l and T_r, connected by a directed edge from r.

- T_l and T_r satisfy the BST property:
  - For all nodes *s* in T_l, $s_{item} < r_{item}$.
  - *F*or all nodes *t* in T_l, $t_{item} > r_{item}$.

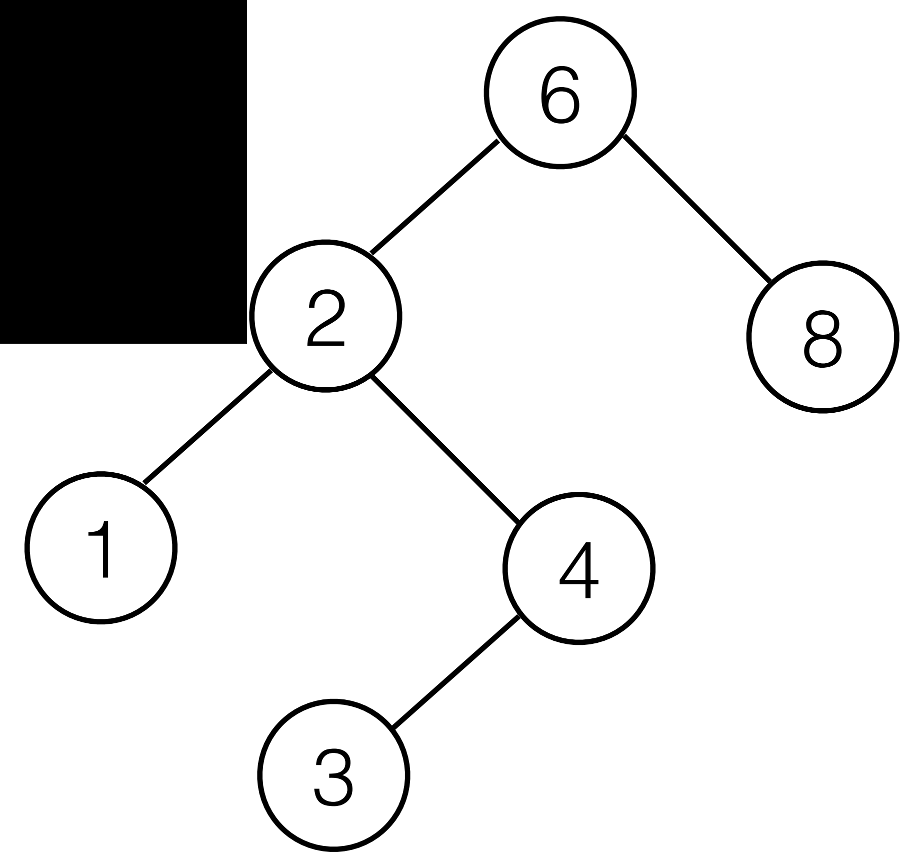
- No value appears more than once in the BST.

BST operations

- `insert(x)` - add value x to T.
- `contains(x)` - check if value x is in T.
- `findMin()` - find smallest value in T.
- `findMax()` -find largest value in T.
- `remove(x)` -remove an item from T.

BST operations: `contains`

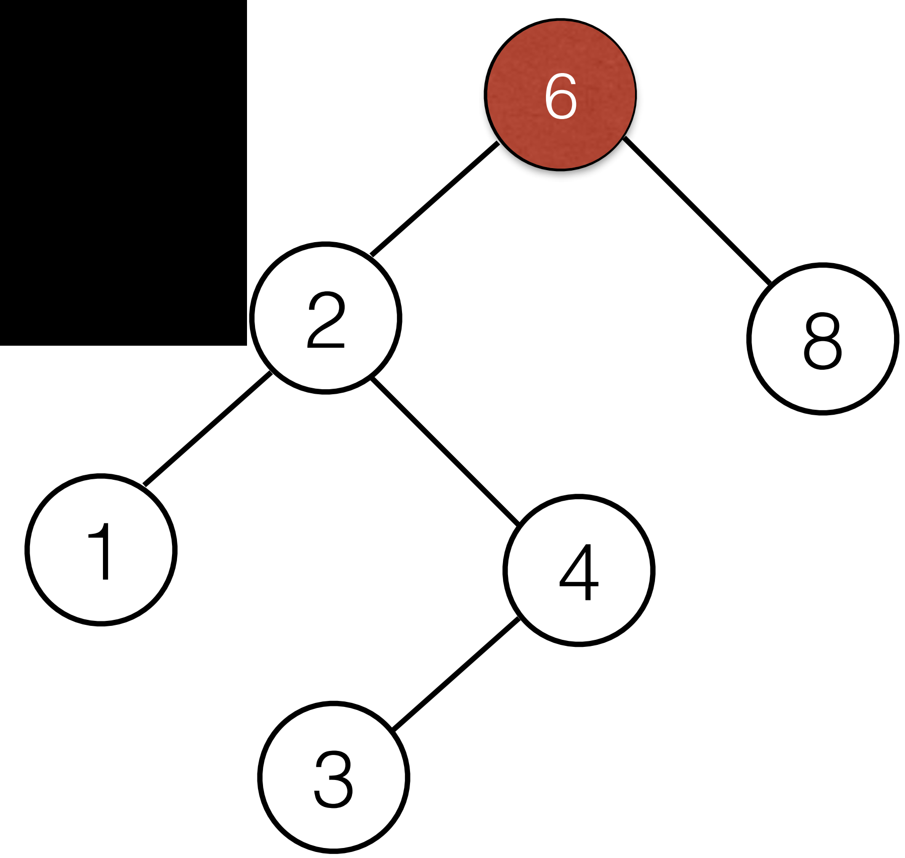
```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```



BST operations: `contains`

```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

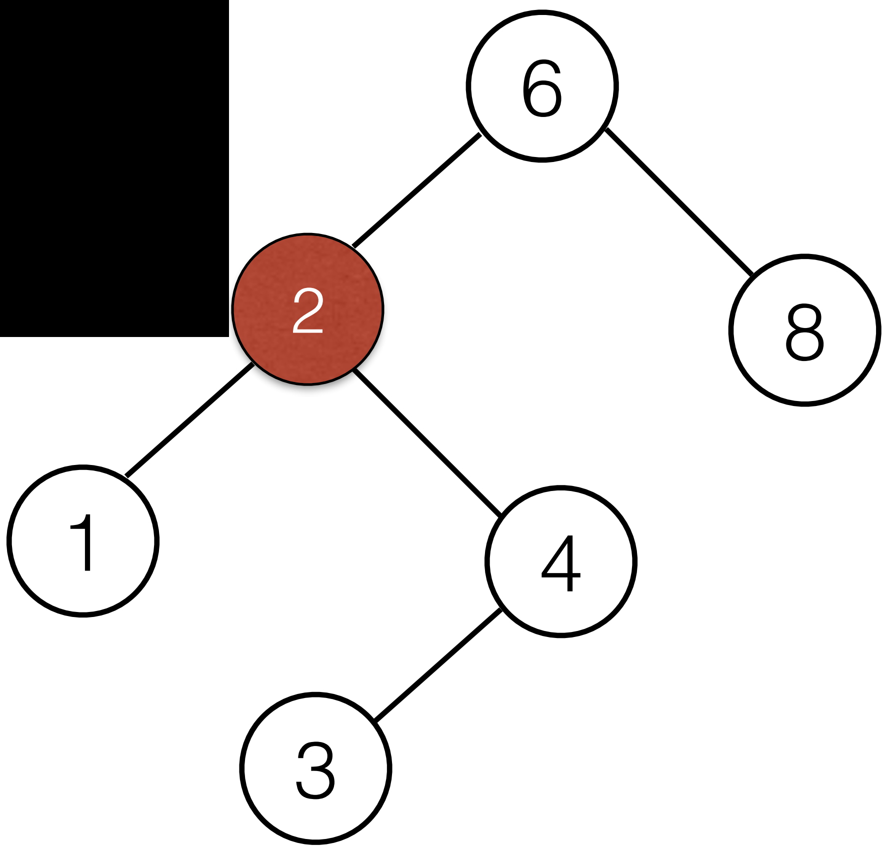
`contains(3)`



BST operations: contains

```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

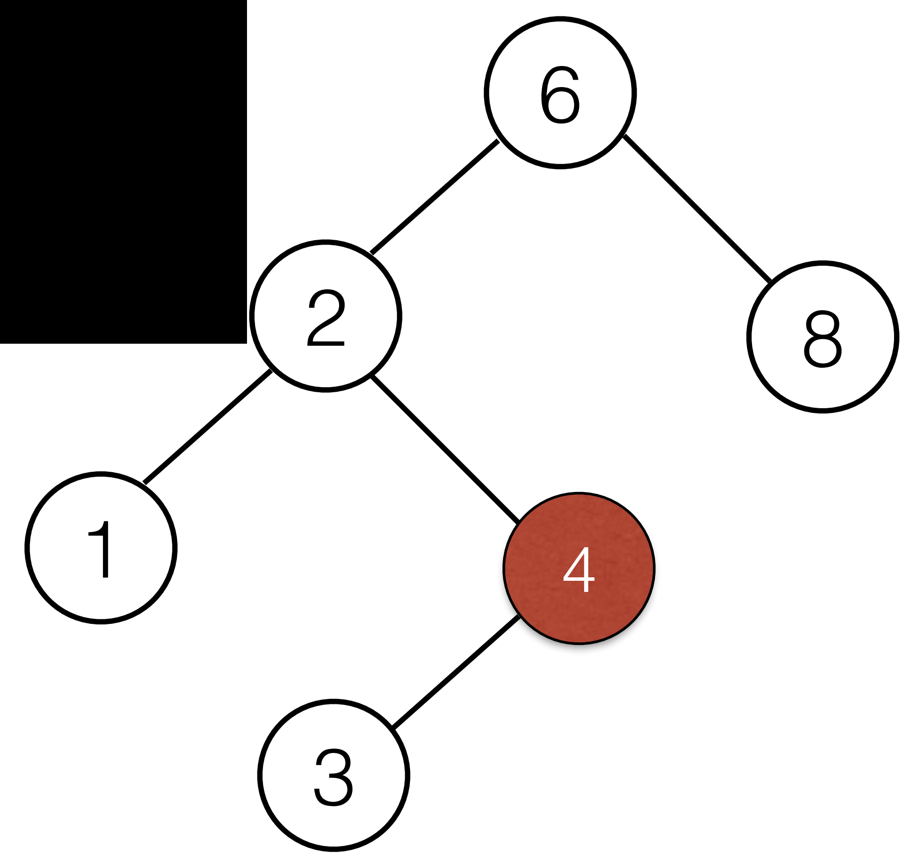
contains(3)



BST operations: `contains`

```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

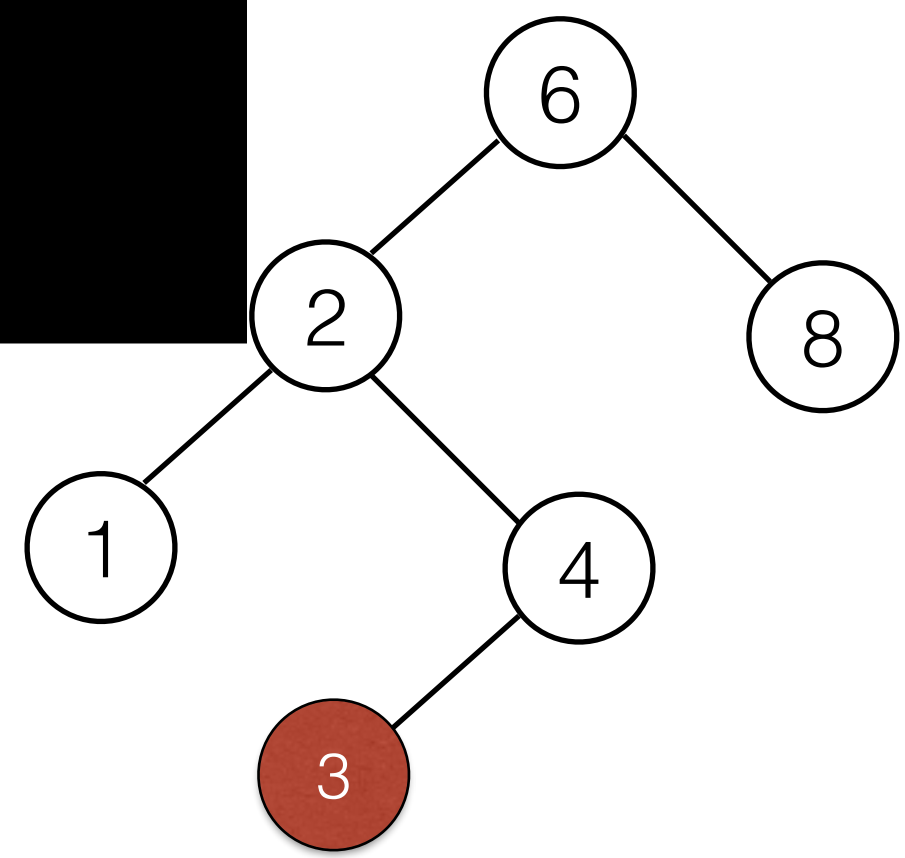
`contains(3)`



BST operations: `contains`

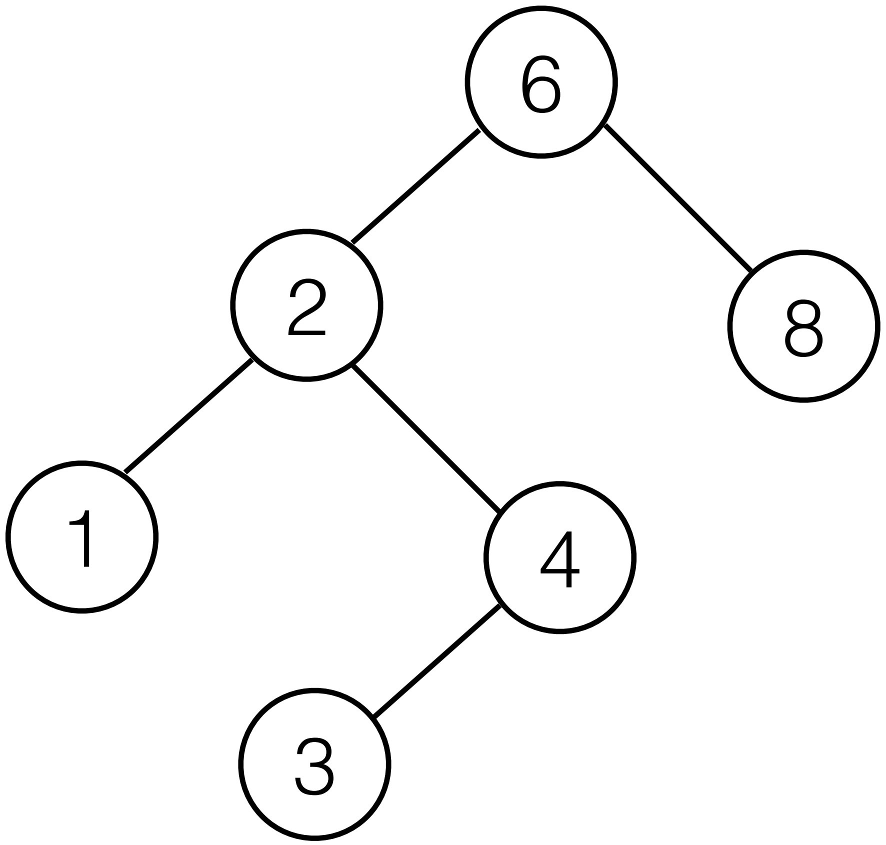
```
private boolean contains( Integer x, BinaryNode t ) {
      if( t == null )
        return false;

     if( x < t.data )
         return contains( x, t.left );
     else if( t.data < x )
         return contains( x, t.right );
     else
         return true;     // Match
}
```

`contains(3)`



BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

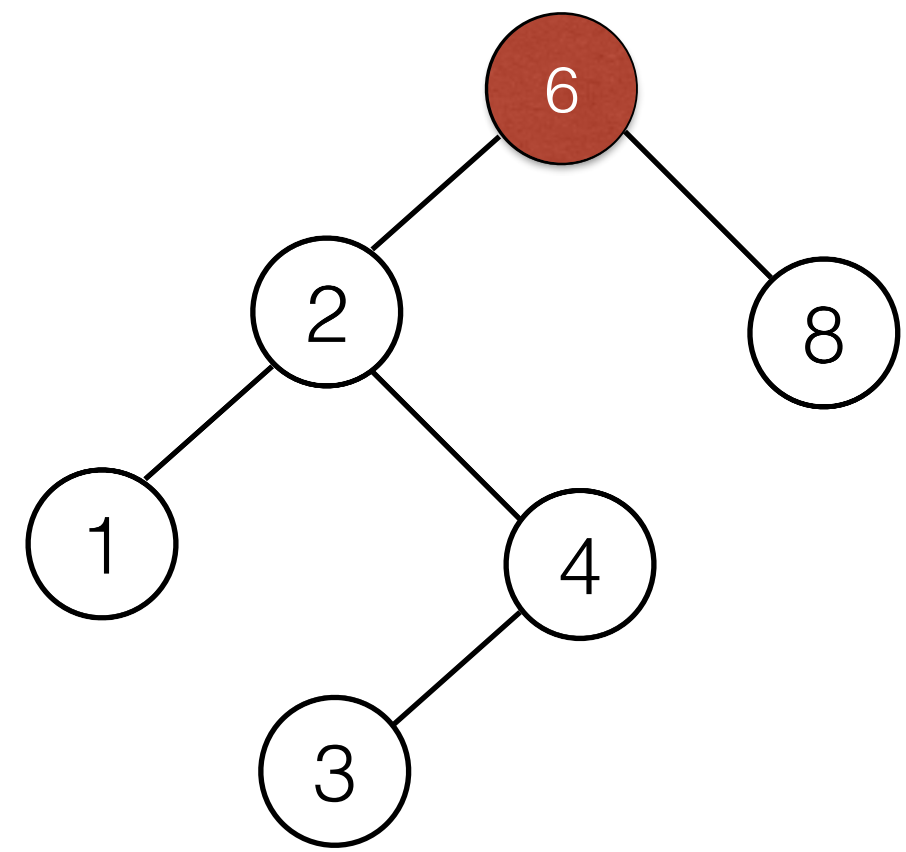


findMax is equivalent.

BST operations: `findMin`

```java
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

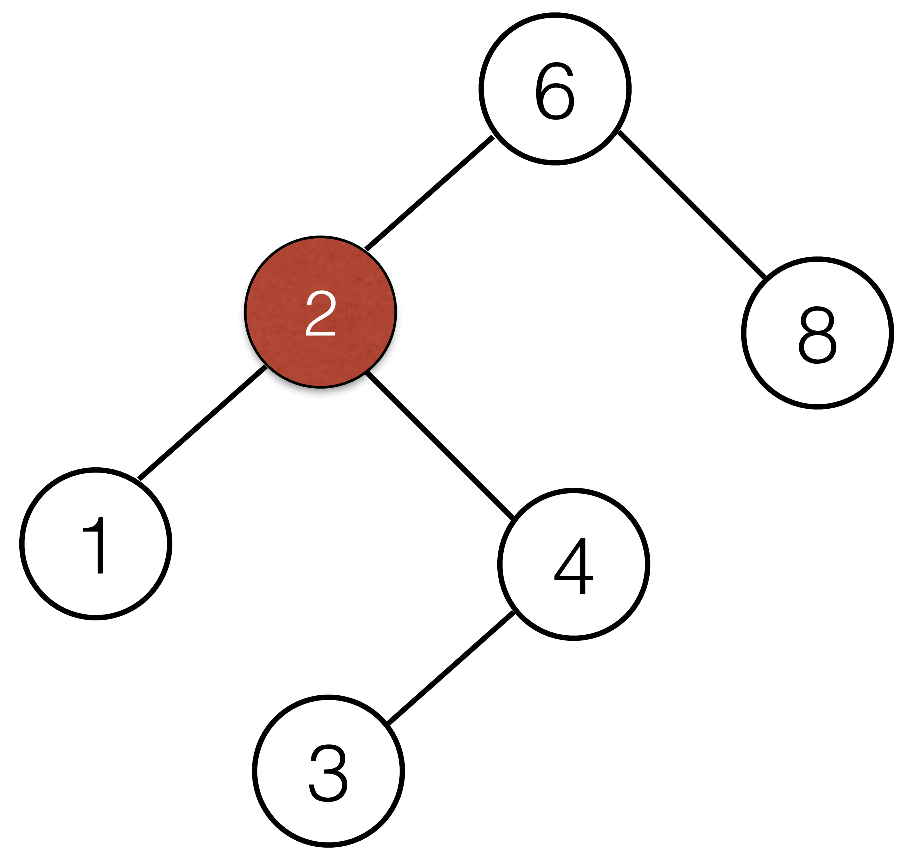


findMax is equivalent.

BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

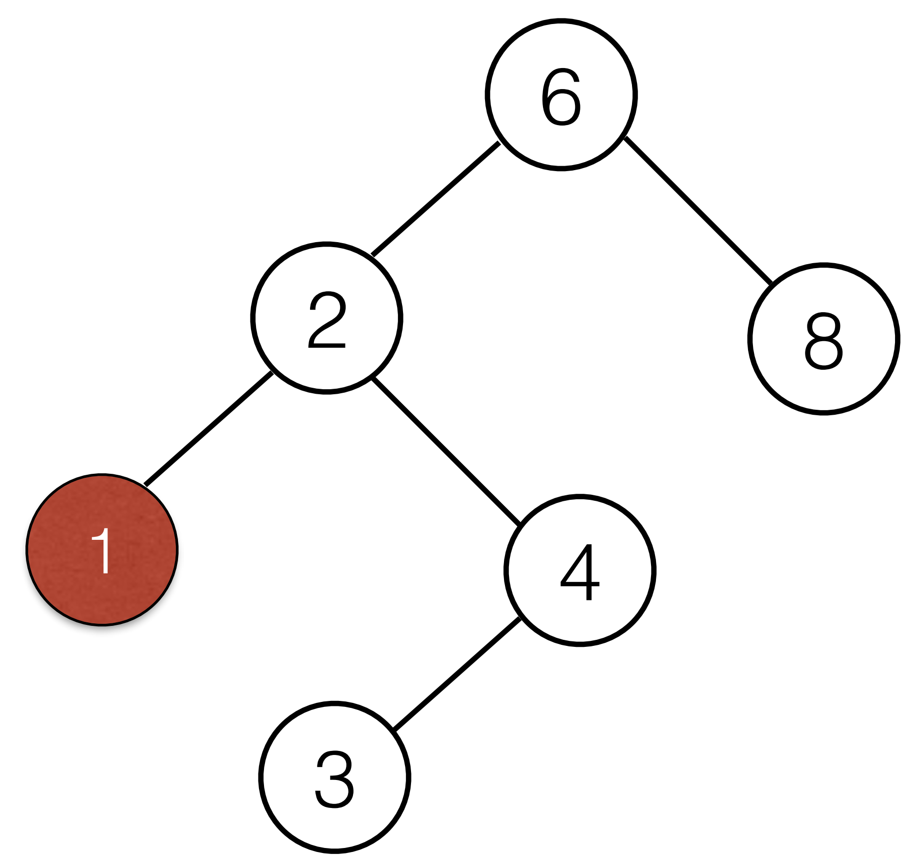


findMax is equivalent.

BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

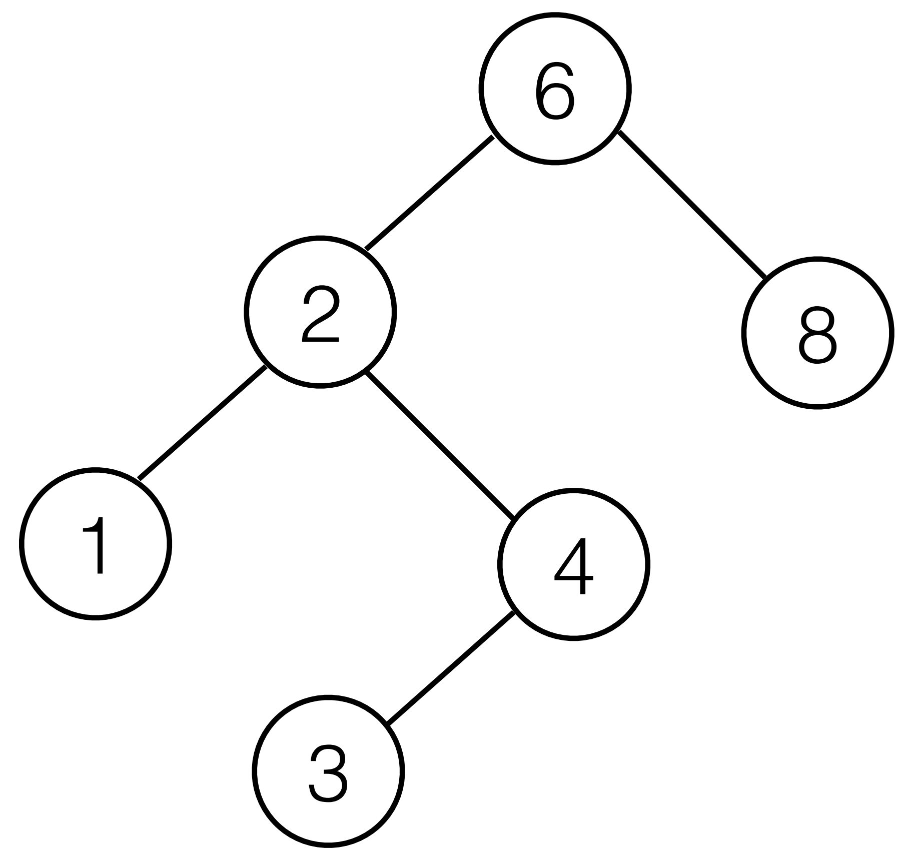


findMax is equivalent.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, contains stopped at node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
        return new BinaryNode( x, null, null );

  if( x < t.data   )
        t.left = insert( x, t.left );
  else if( t.data < x )
        t.right = insert( x, t.right );

  return t;
}
```



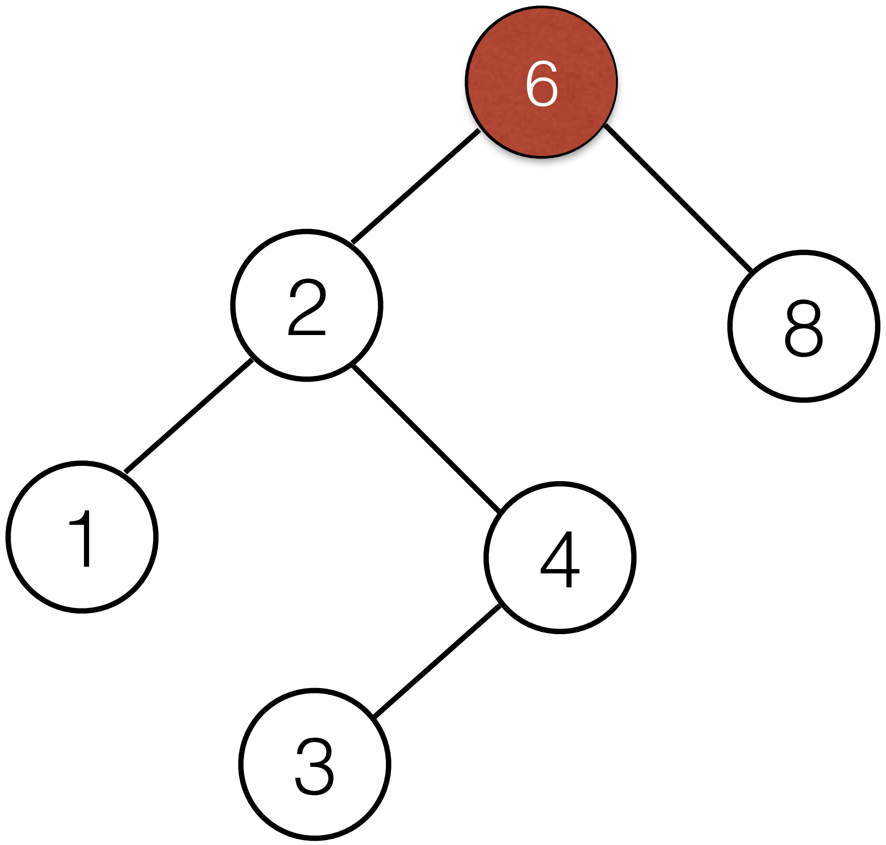
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, contains stopped at node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
        return new BinaryNode( x, null, null );

  if( x < t.data   )
        t.left = insert( x, t.left );
  else if( t.data < x )
        t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



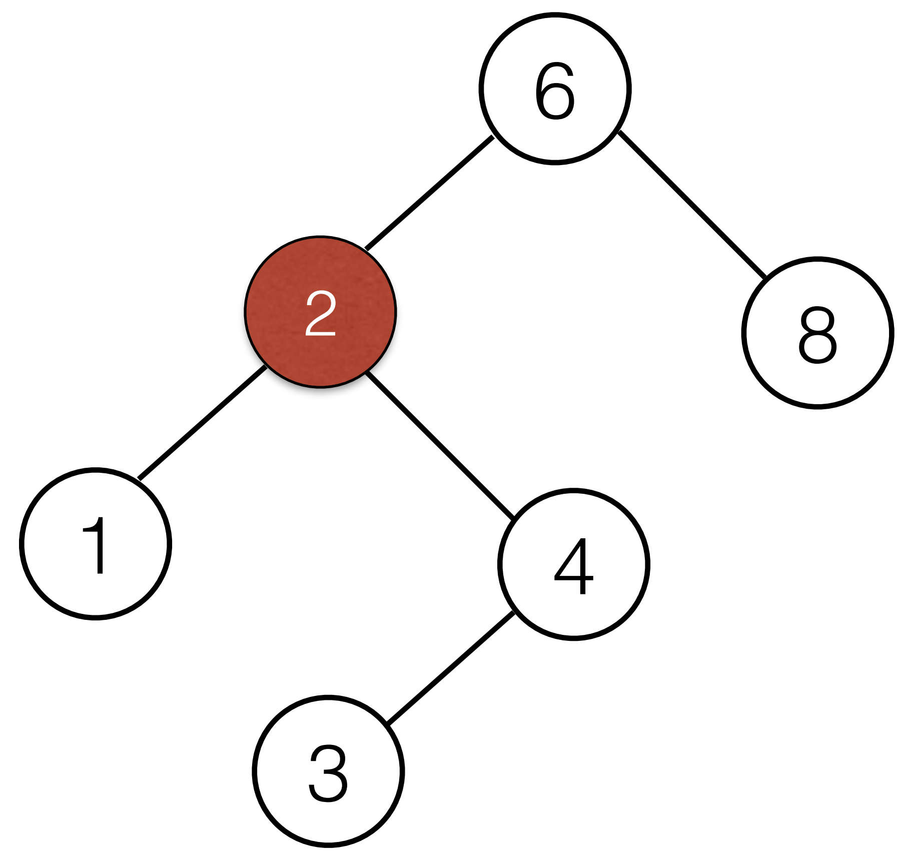
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, contains stopped at node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
        return new BinaryNode( x, null, null );

  if( x < t.data   )
        t.left = insert( x, t.left );
  else if( t.data < x )
        t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



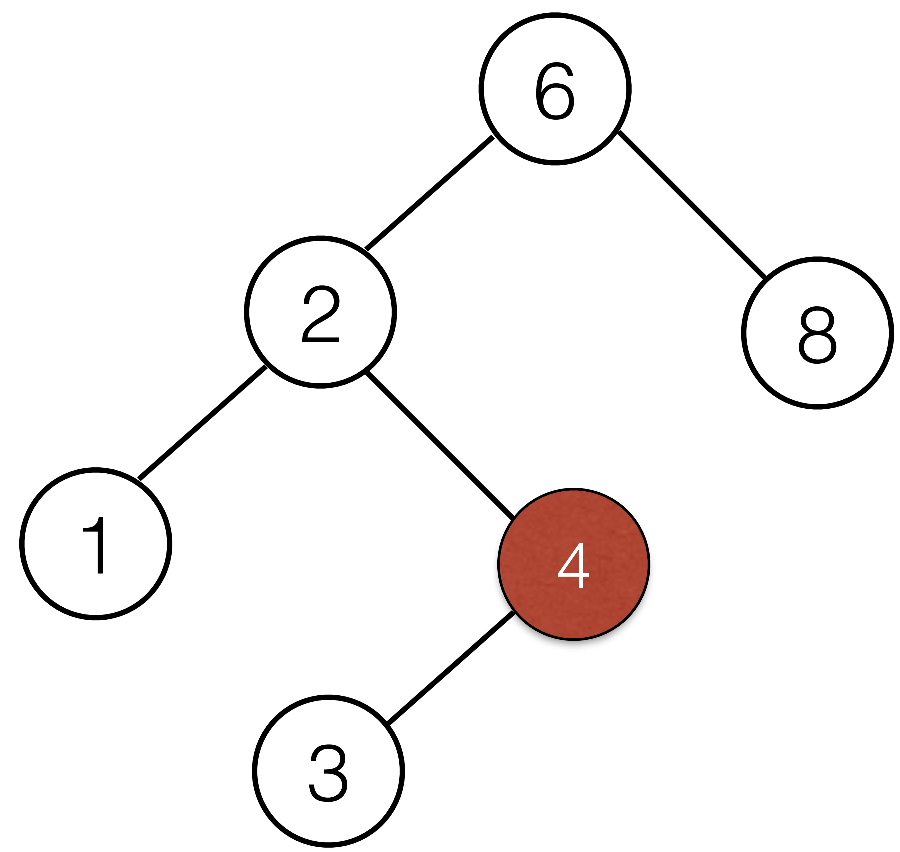
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, contains stopped at node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
        return new BinaryNode( x, null, null );

  if( x < t.data   )
        t.left = insert( x, t.left );
  else if( t.data < x )
        t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



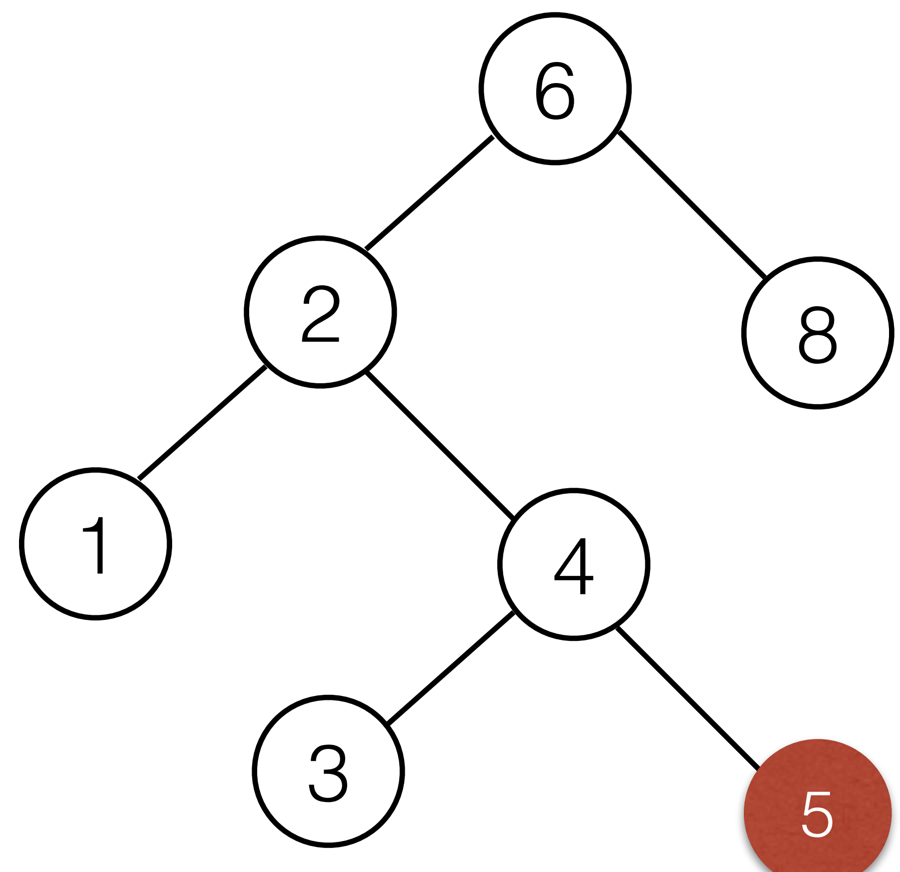
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, contains stopped at node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
        return new BinaryNode( x, null, null );

  if( x < t.data   )
        t.left = insert( x, t.left );
  else if( t.data < x )
        t.right = insert( x, t.right );

  return t;
}
```

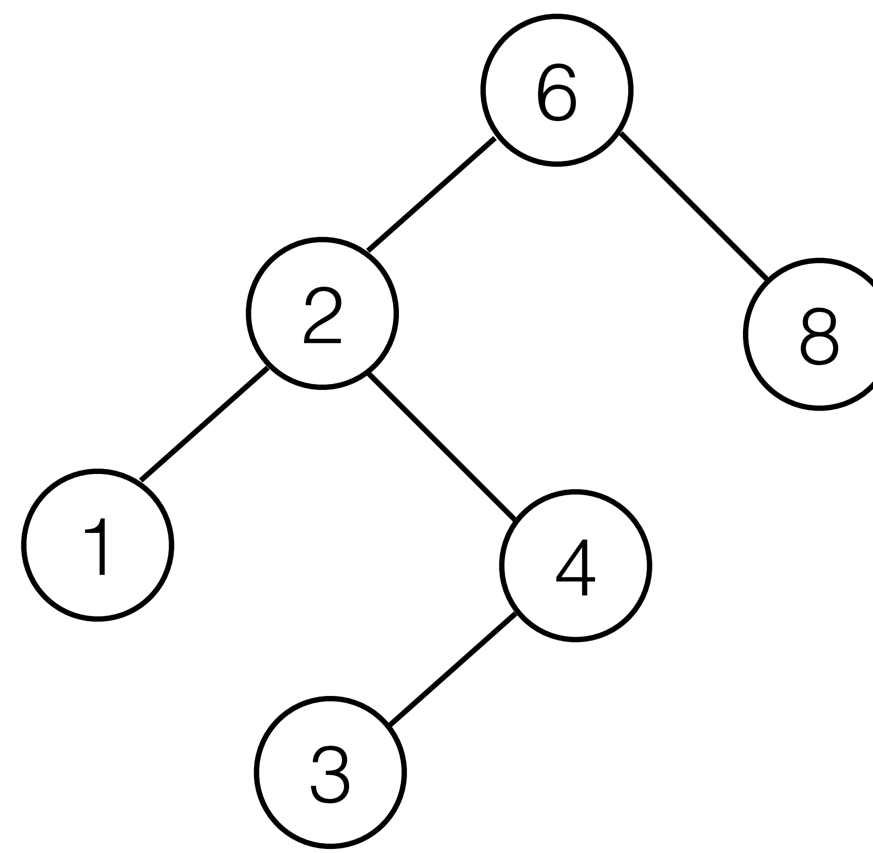
`insert(5)`



Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

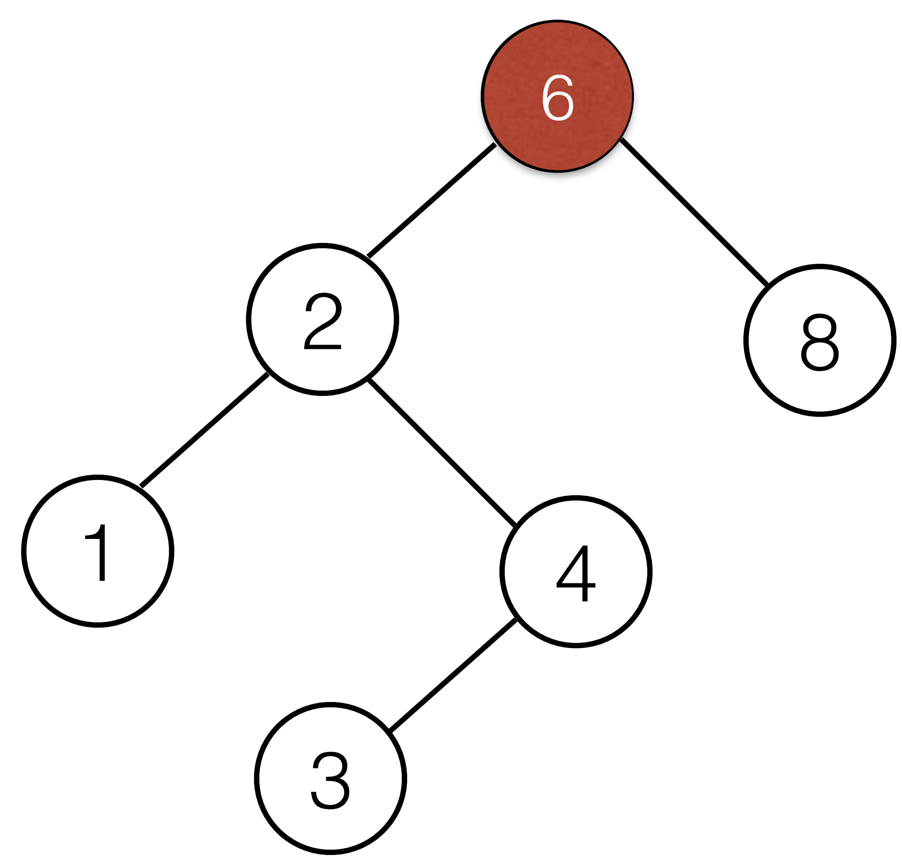


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

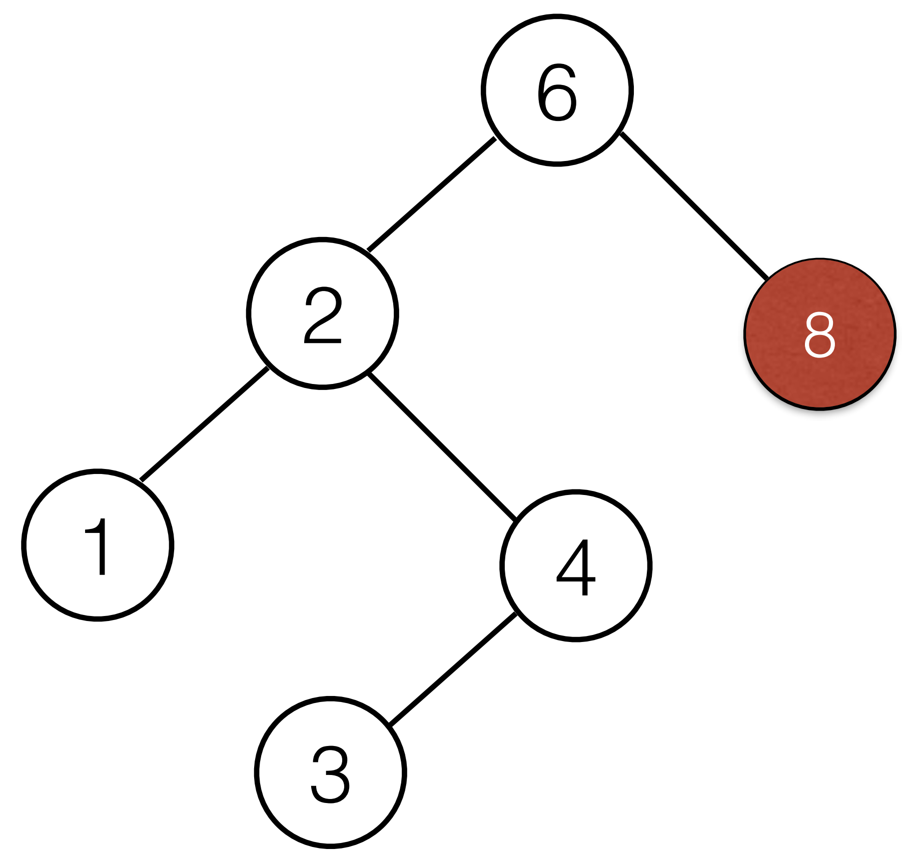


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

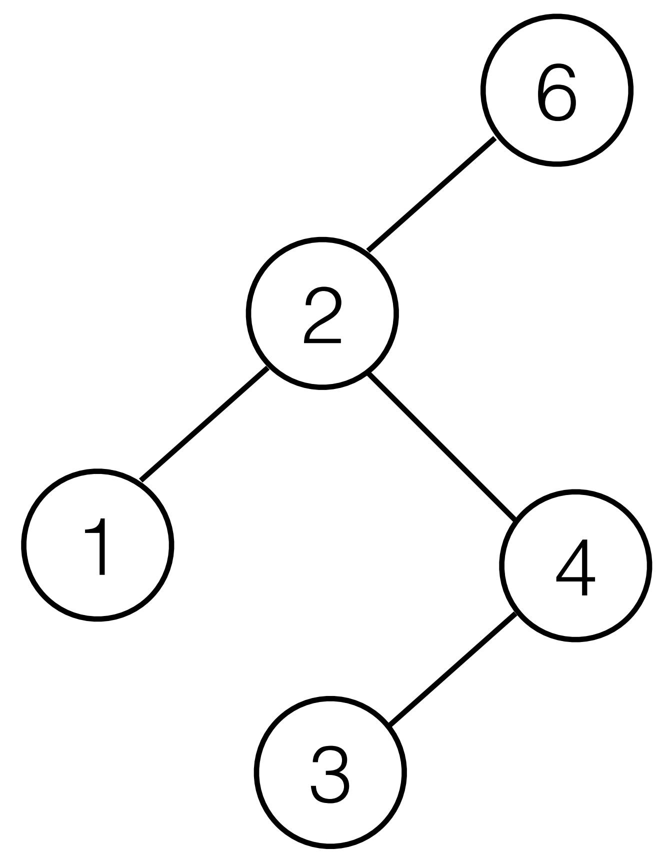


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X).`
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

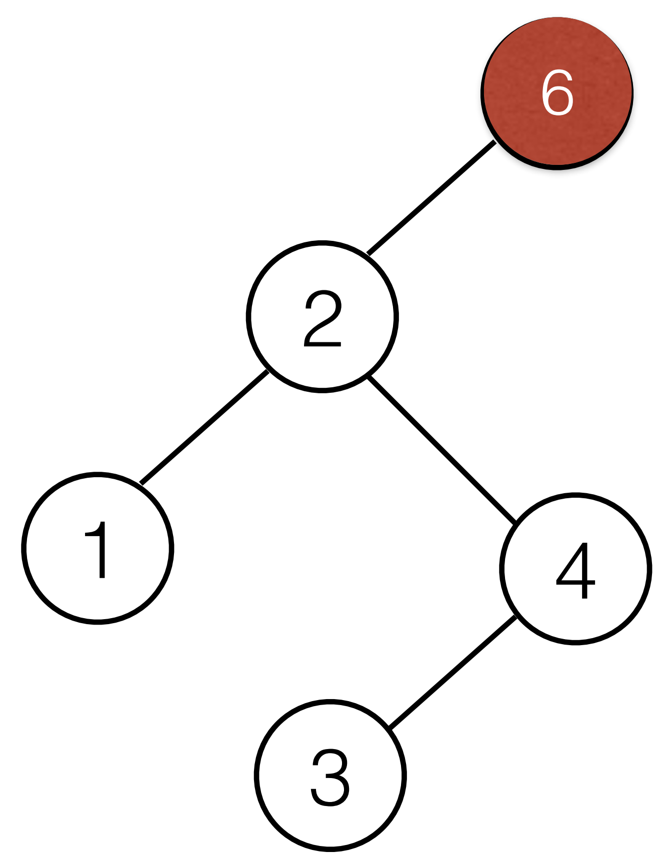


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

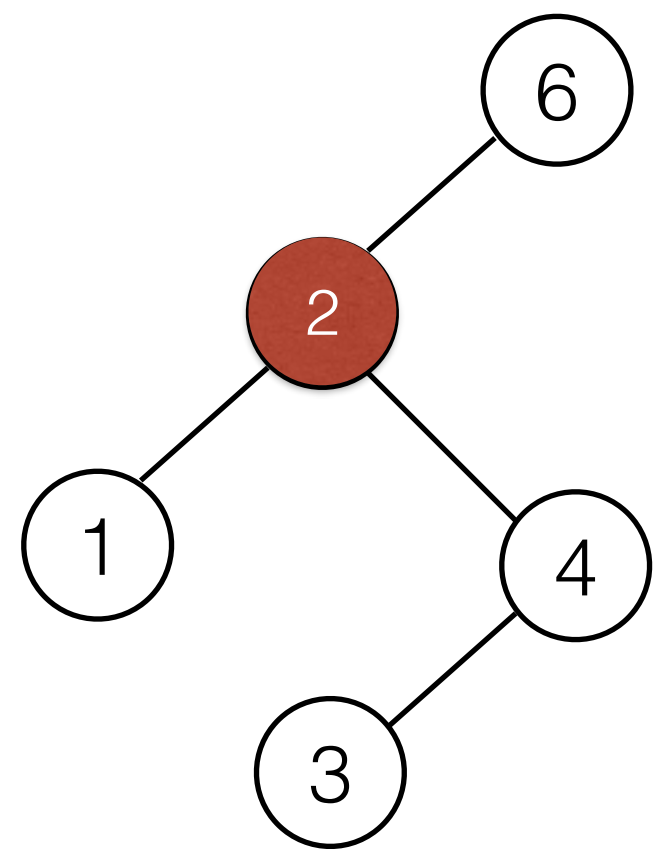


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

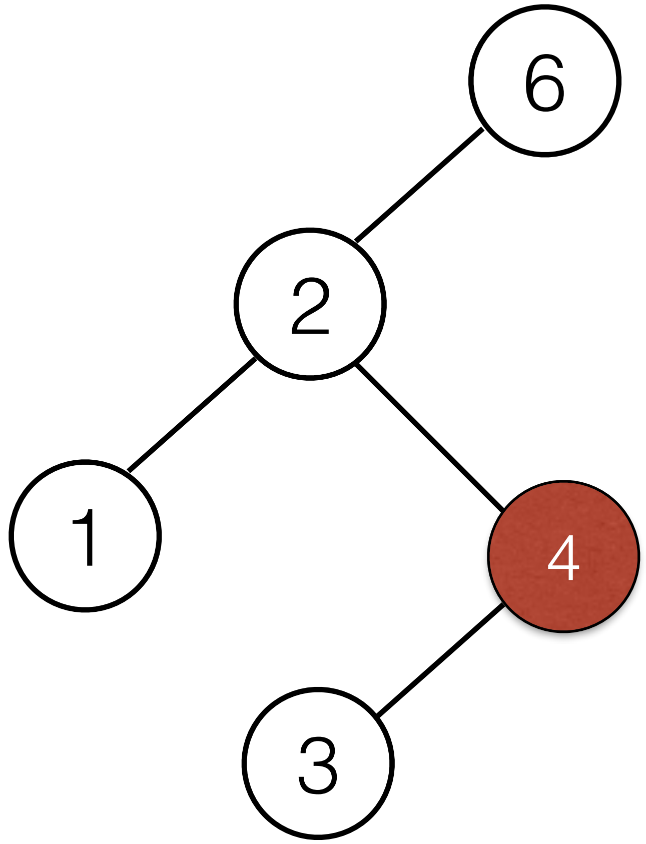


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

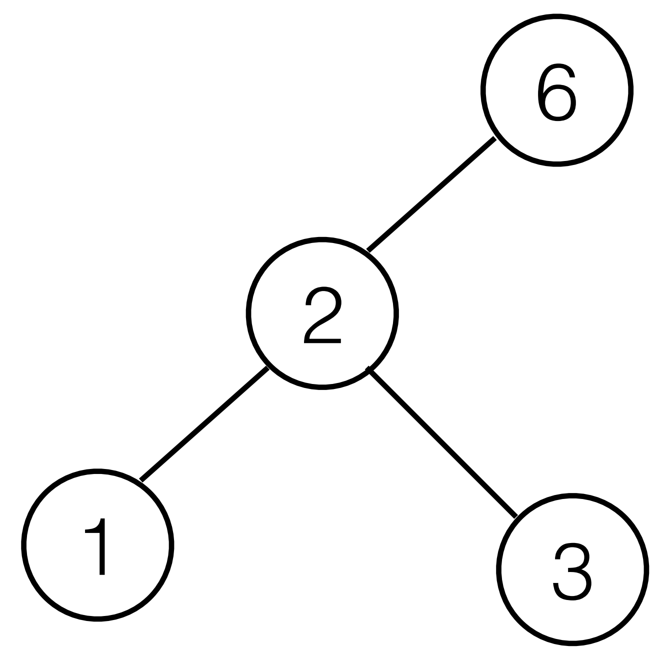


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

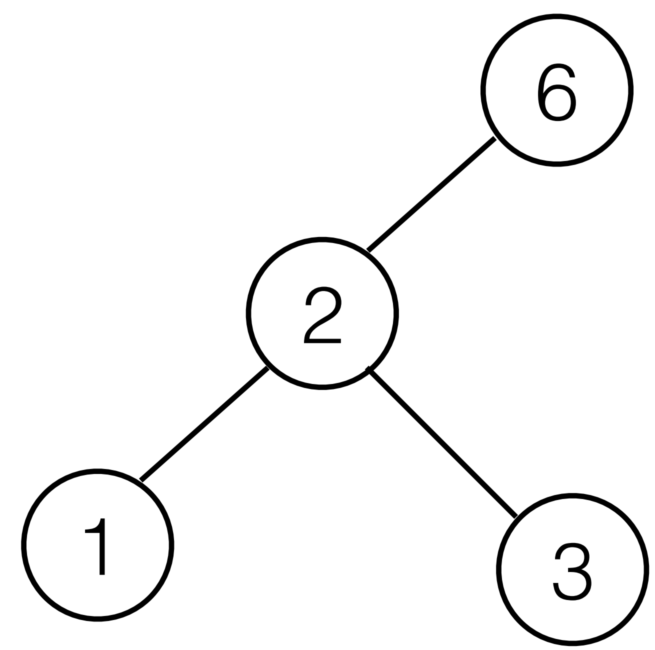


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.
  - what if s has two children?

`remove(4)`



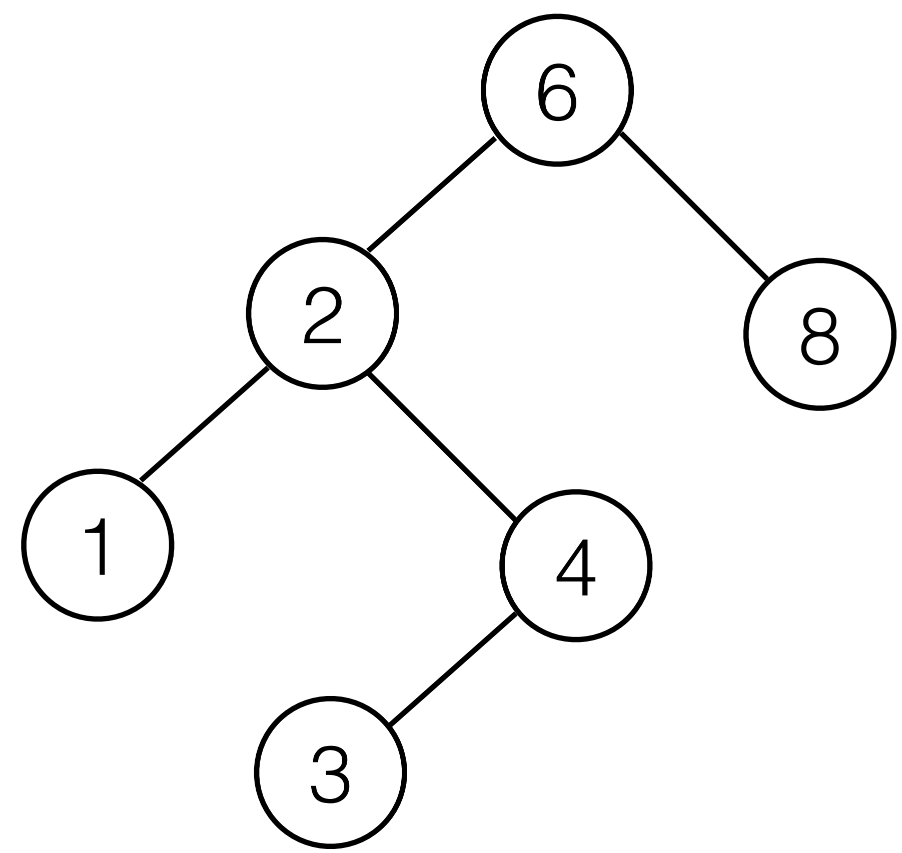
Maintains the BST property.

BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node to replace s needs to be

- larger than any node in the left subtree
- but smaller than any node in the right subtree.

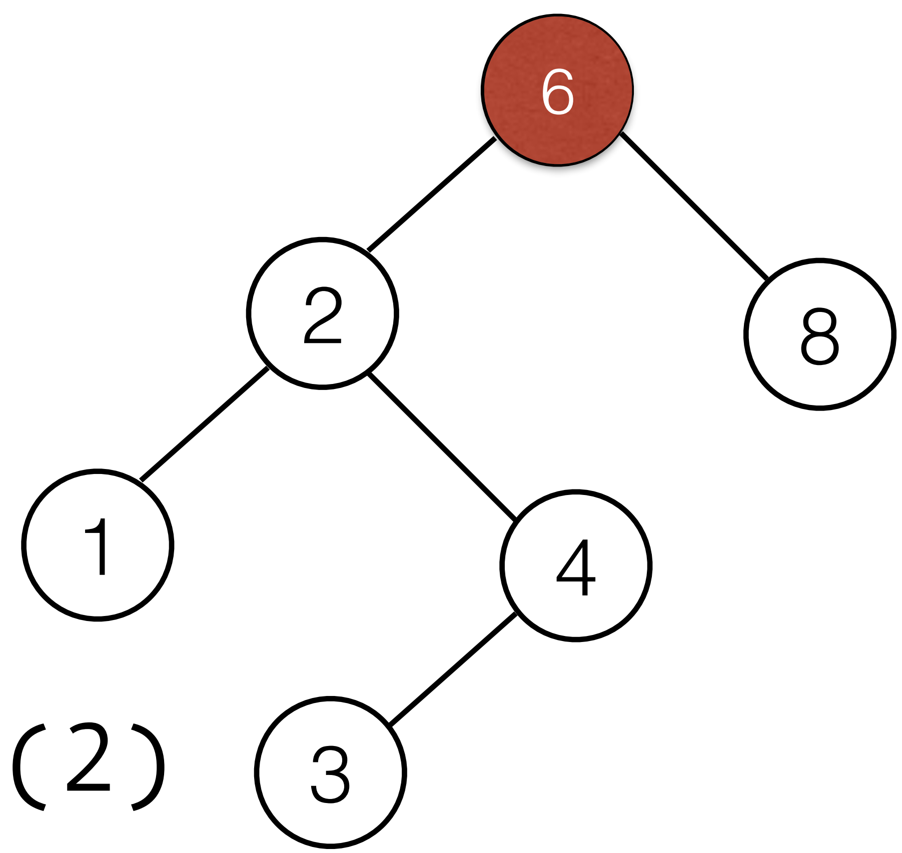


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node to replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

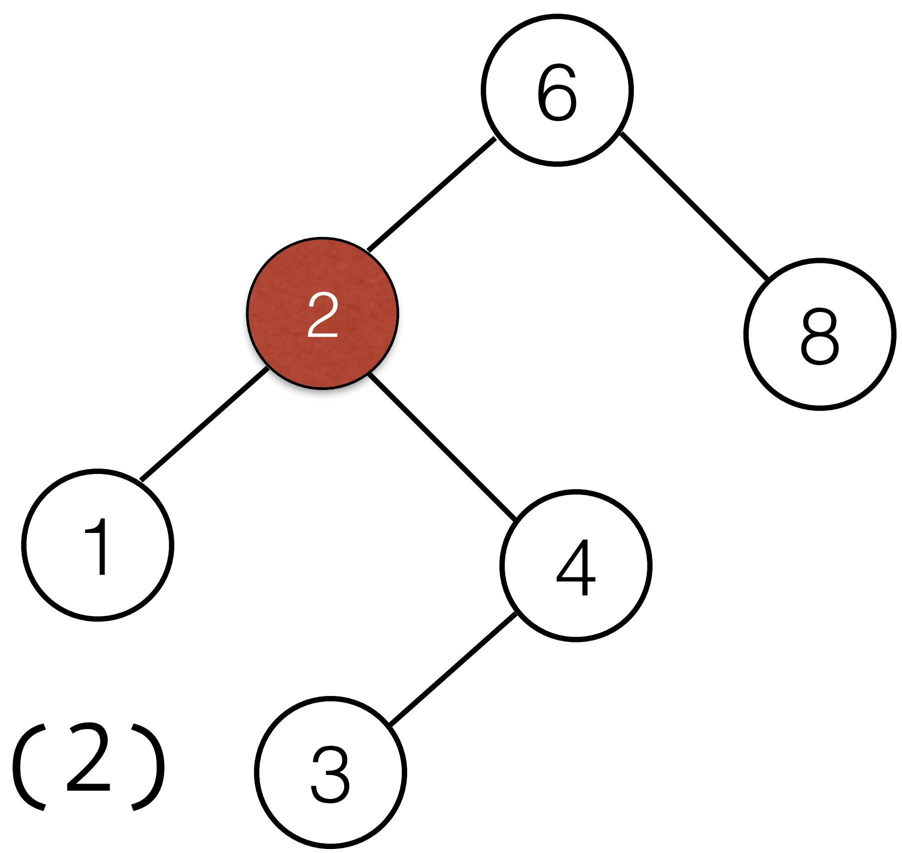


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node to replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`



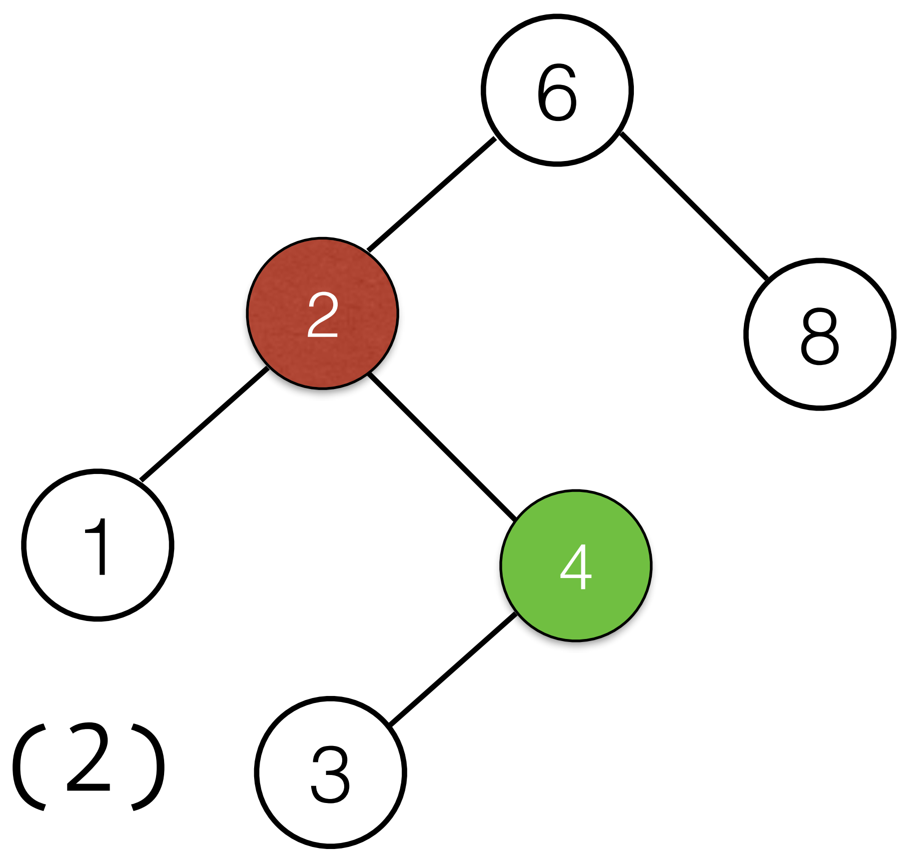
BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
    - Find the smallest node *u* in the subtree rooted in t_{right}.
    - replace value of s with value of u.
    - recursively remove u.

To maintain the BST property, the node to replace s needs to be

- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`



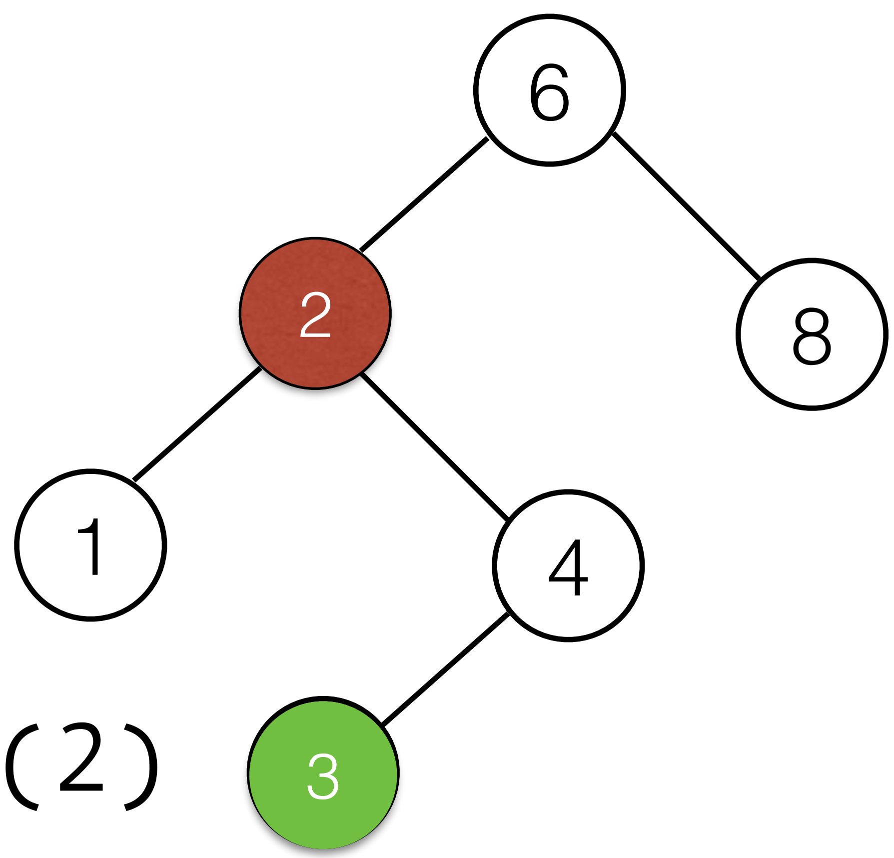
BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node to replace s needs to be

- larger than any node in the left subtree
- but smaller than any node in the right subtree.

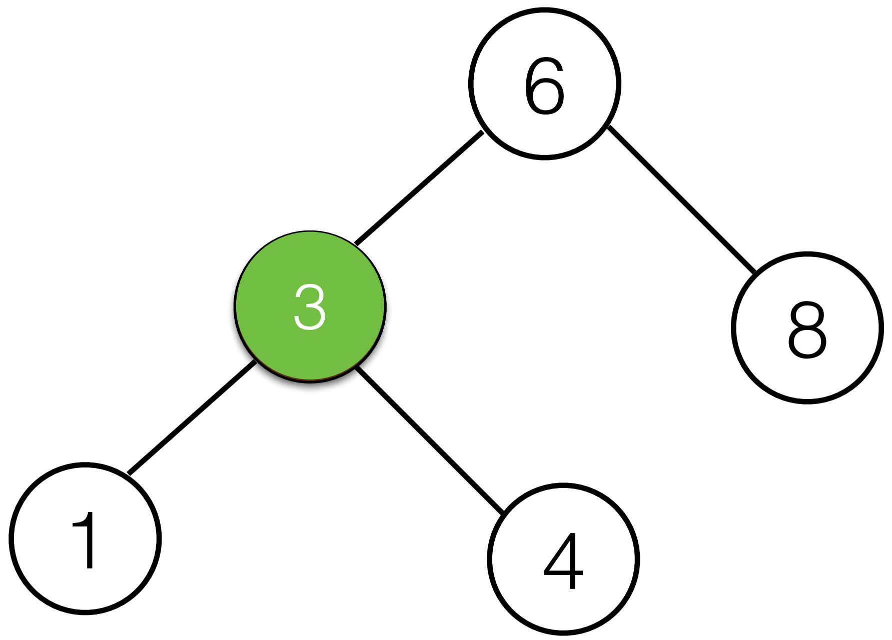
`remove(2)`



BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

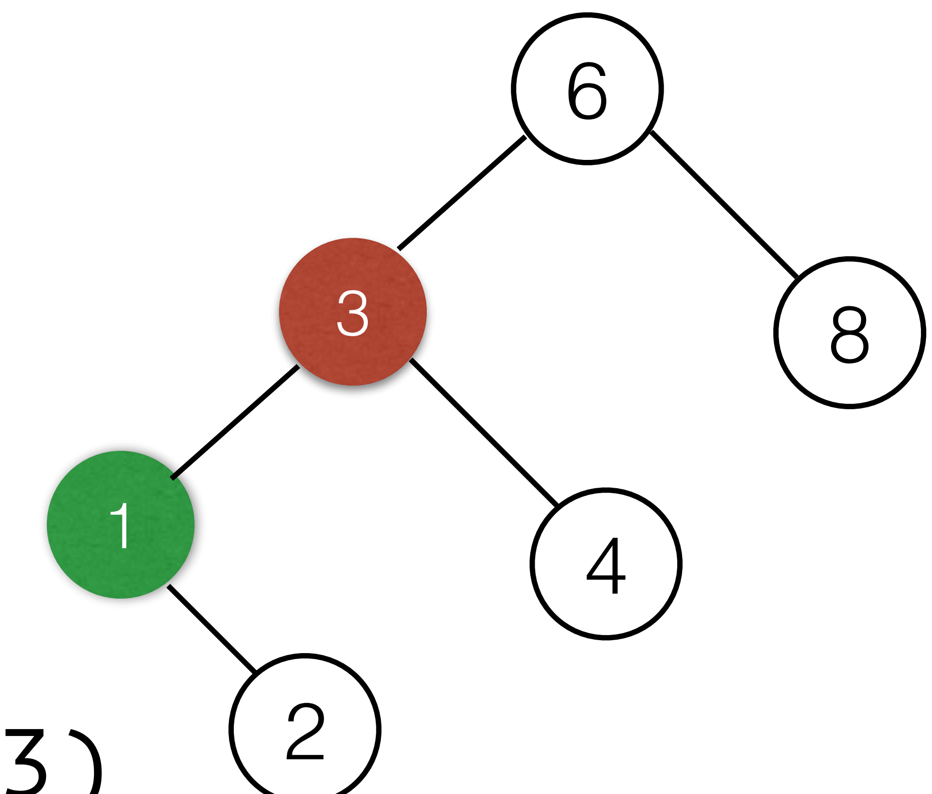
To maintain the BST property, the node to replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.



`remove(2)`

BST operations: `remove`

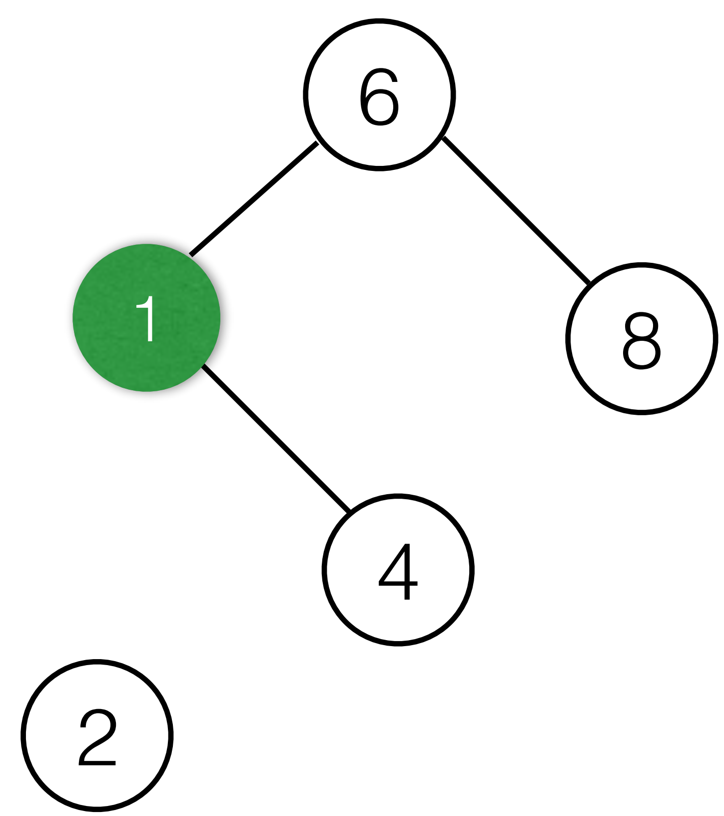
- Why not just replace s with the root of t_{left}?



`remove(3)`

BST operations: `remove`

- Why not just replace s with the root of t_{left}?



`remove(3)`

Implementing `remove`

```
private BinaryNode remove( Integer x, BinaryNode t ){
  if( t == null )
    return t; // Item not found; do nothing

  if (x < t.data )
    t.left = remove( x, t.left );
  else if(t.data < x )
    t.right = remove( x, t.right );

  else //found x
    if( t.left != null && t.right != null ) { // 2 children
      t.element = findMin( t.right ).element;
      t.right = remove( t.element, t.right );
    } else
      if (t.left != null) // 1 or 0 children.
        return t.left;
      else
        return t.right;
}
```

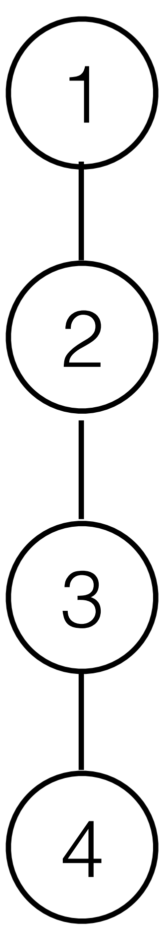
Running Time Analysis

- How long do the BST operations take?
- Given a BST *T*, we need a single pass down the tree to access some node *s* in *depth(s)* steps.
- What is the best/expected/worst-case depth of a node in any BST?

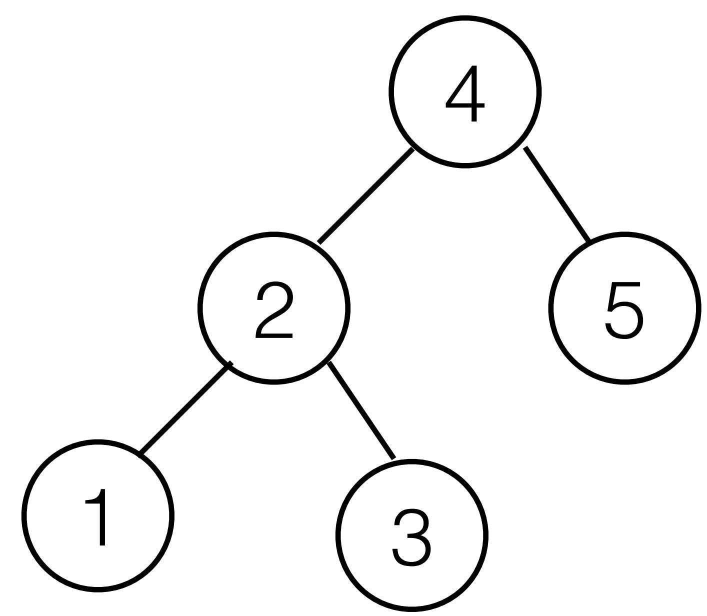
Worst and Best Case Height of a BST

- Assume we have a BST with N nodes.

- Worst case: T does not branch *height(T)=N*



- Best case: *height(T)=log N*



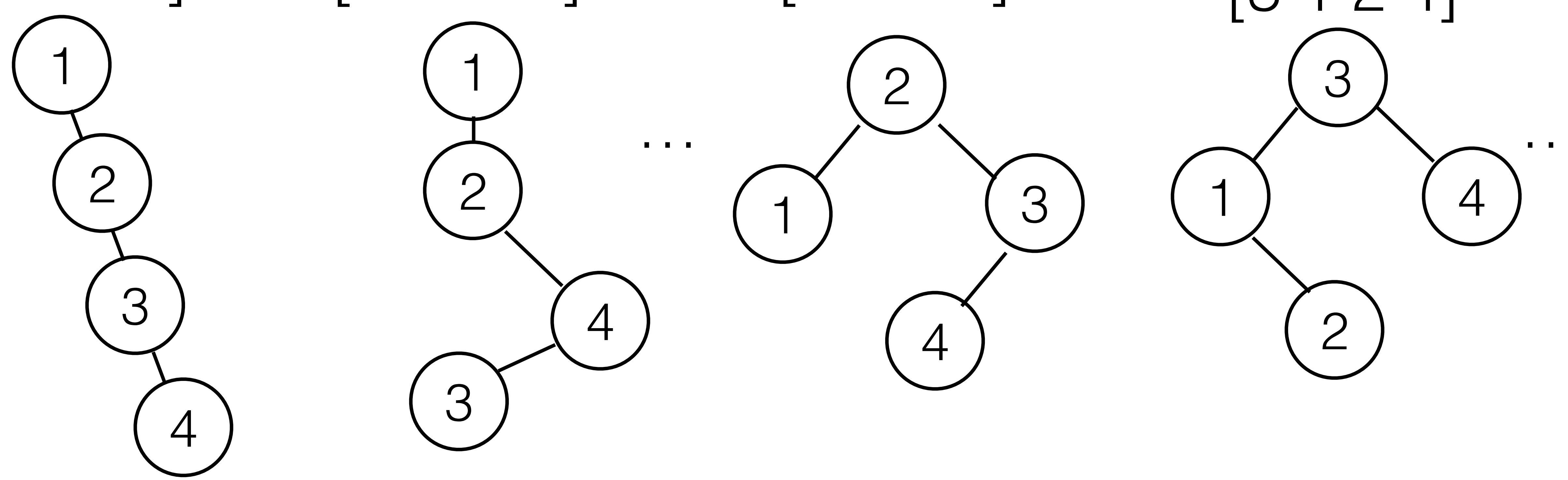
complete binary tree.

Random BSTs

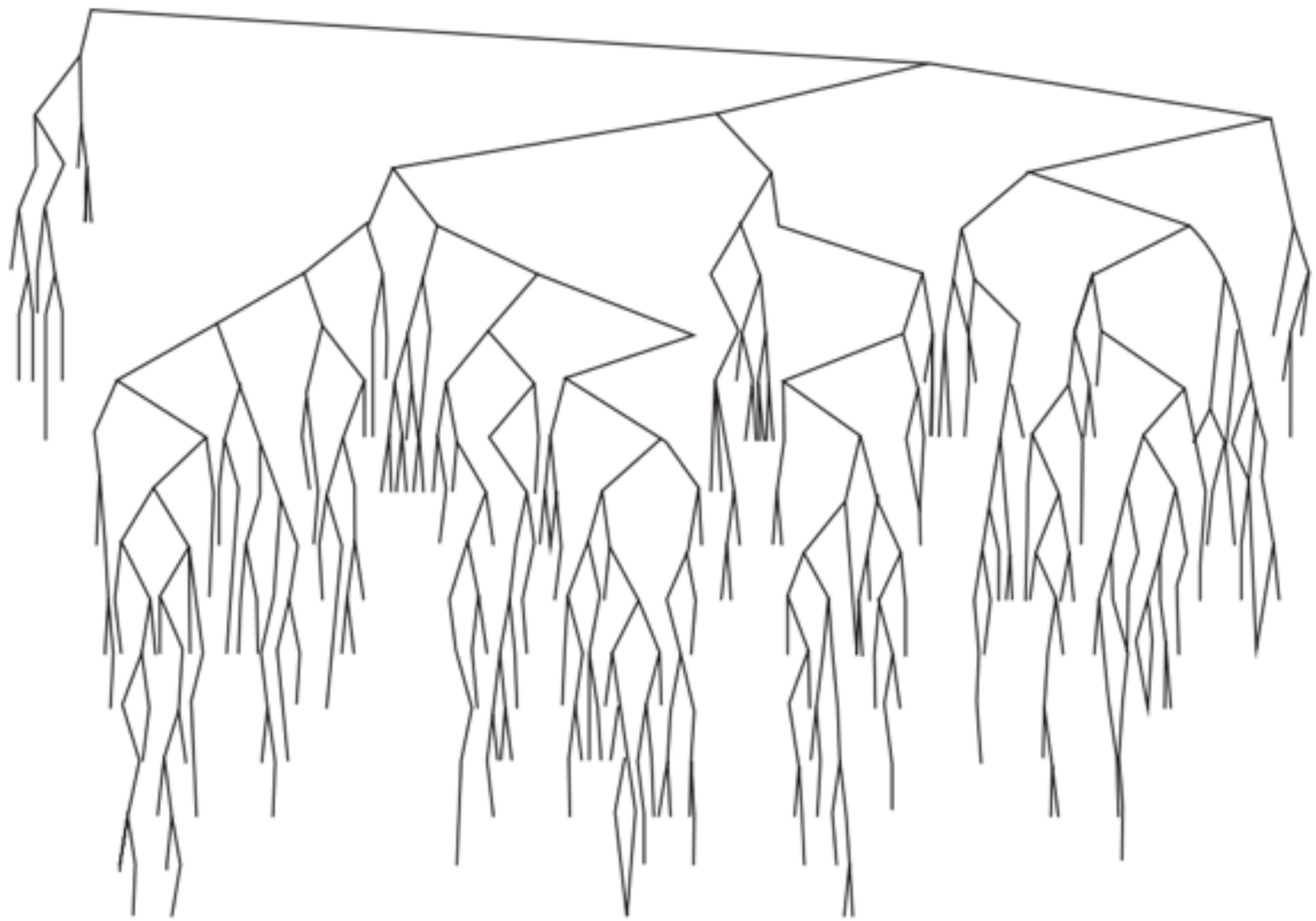
- Assume we have *N* elements. All *N!* permutations of these elements are equally likely.

- We insert items in the order of any permutation into an initially empty BST. What is the average depth of a node?

[1 2 3 4] [1 2 4 3] [2 3 4 1] [3 1 2 4]



Randomly generated BST N=500, average depth = 9.89



Theoretical analysis of random BSTs: Average depth of a node in a random BST of N nodes is about

$$2 \log N = O(\log N)$$

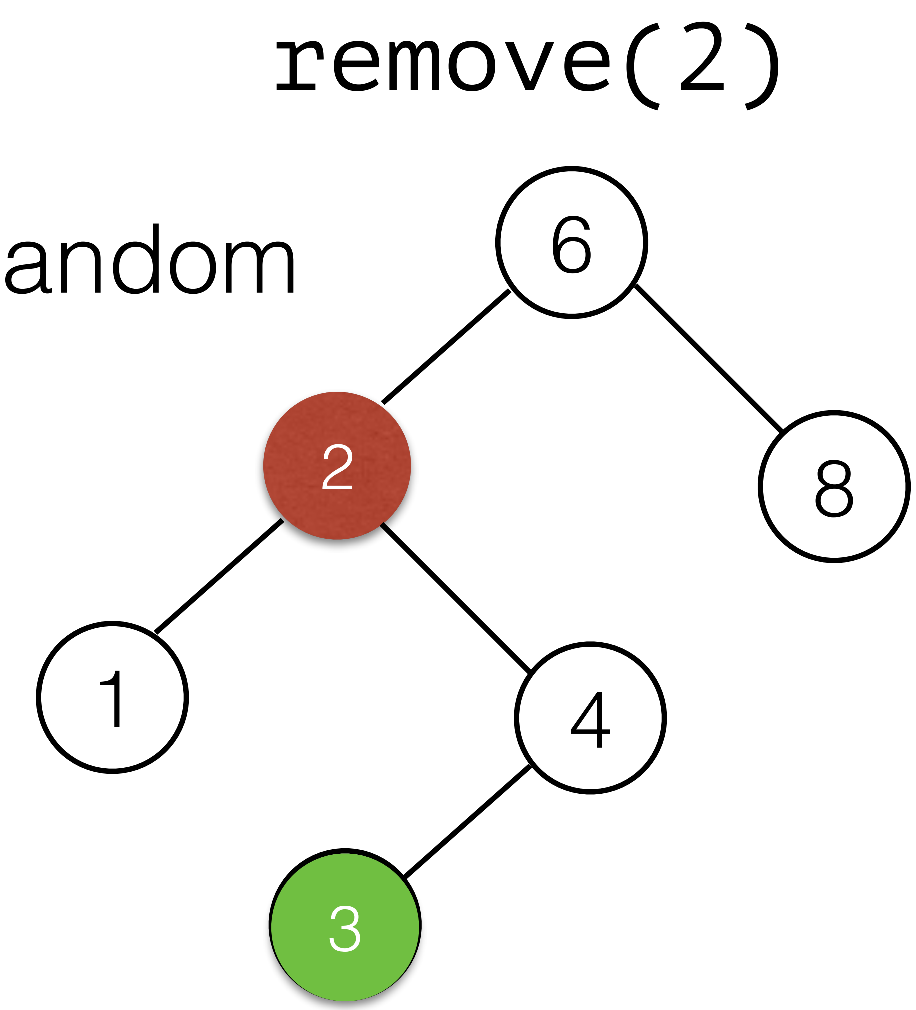
Source: Weiss, Data Structures and Algorithm Analysis in Java, 3rd Edition

What about Different Sequences of Operations?

- The expected depth of a random BST (insertions of a random permutation of elements) is $O(\log N)$.

- But what happens if there are also random deletions?

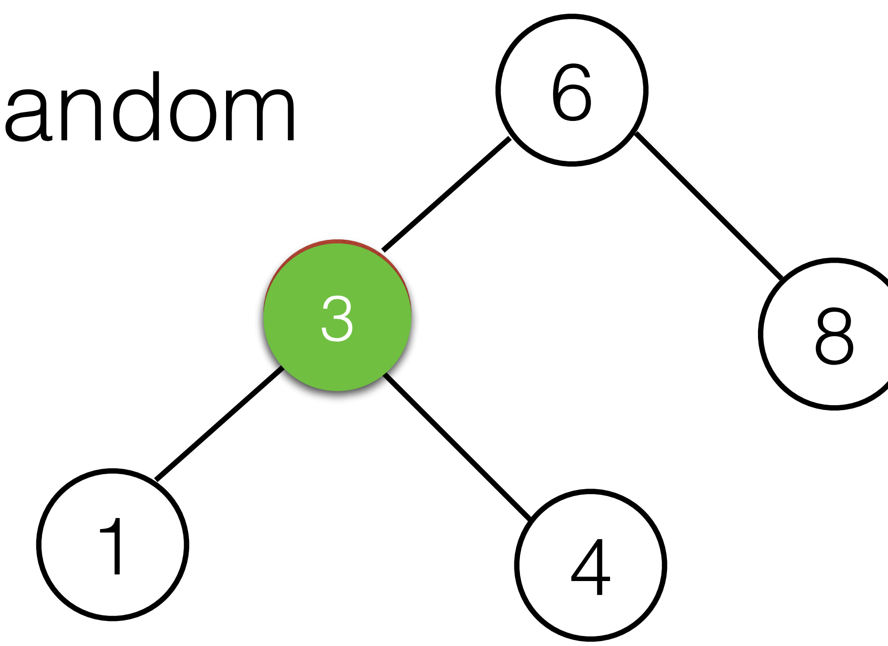
- Deletion produces shorter right subtrees.



What about Different Sequences of Operations?

- The expected depth of a random BST (insertions of a random permutation of elements) is $O(\log N)$.

`remove(2)`

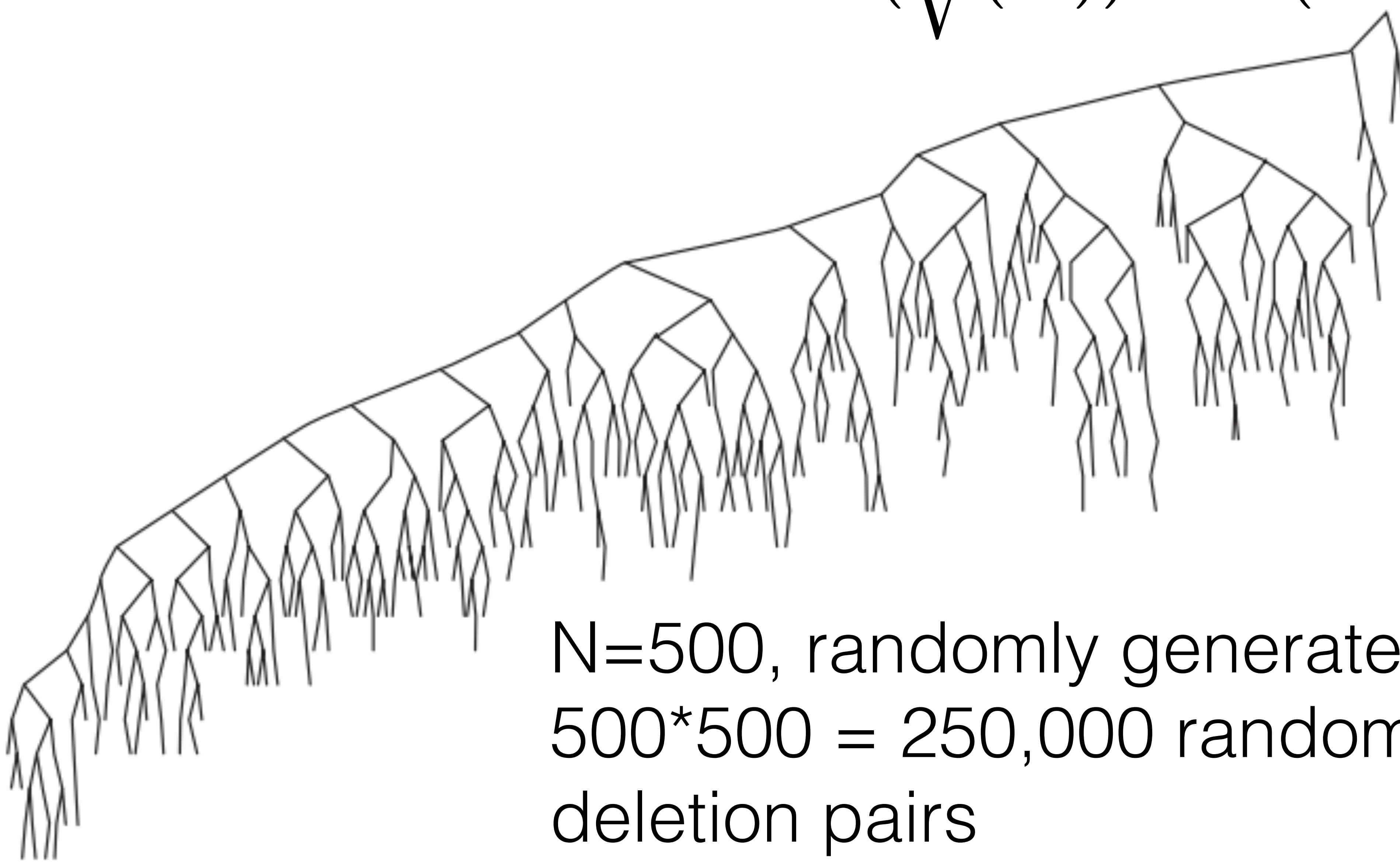


- But what happens if there are also random deletions?

- Deletion produces shorter right subtrees.

Random Insertions and Deletions

- After $\Theta(N^2)$ alternating insertion/deletion pairs, the expected depth is $\Theta(\sqrt{(N)}) = \Theta(N^{1/2})$



N=500, randomly generated as before.
500*500 = 250,000 random insertion/deletion pairs

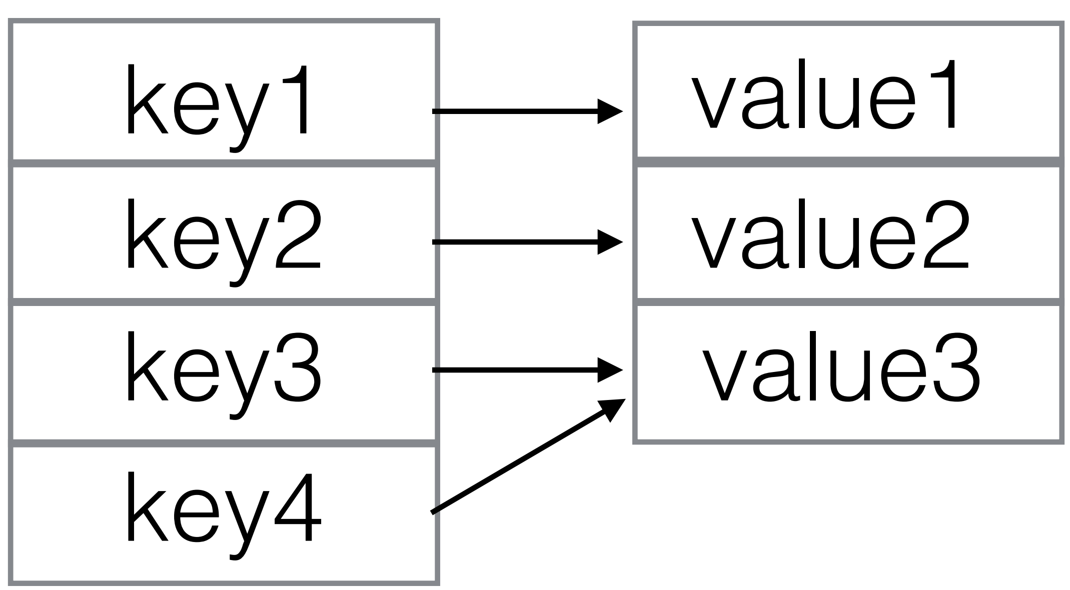
Source: Weiss, Data Structures and Algorithm Analysis in Java, 3rd Edition

Contents

1. Binary Search Trees
2. **Implementing Maps with BSTs**

Map ADT

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be.
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)



How do we implement map operations efficiently?

Arrays as Maps

- When keys are integers, arrays provide a convenient way of implementing maps.

- Time for `get` and `put` is O(1).

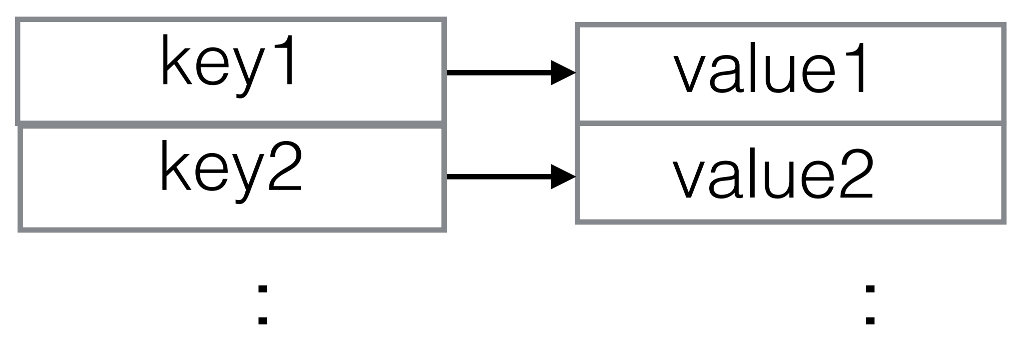
A		B		D	C	
0	1	2	3	4	5	6

- What if we don’t have integer keys?

Comparing Complex Items

- So far, our BSTs contained `Integers`.

- One Goal of BSTs: Implement efficient lookup for Map keys and sorted Sets.



- We can implement generic BSTs that can contain any kind of element, including (key,value) pairs.

- But we must be able to *sort* the elements, i.e. compare them using <, >, and =. The (key, value) pair class should implement Comparable.

Example (key/value) Pair Implementation

```
private class Pair<K extends Comparable<K>,V>
                      implements Comparable<Pair<K,?>>{
    public K key;
    public V value;

    public Pair(K theKey, V theValue) {
       key = theKey; value = theValue;
    }

    @Override
    public int compareTo(Pair<K,?> other) {
        return key.compareTo(other.key);
    }
}
```

Implementing Maps with BSTs

(see example code)