Data Structures in Java

Lecture 10: Proofs by Structural Induction
Binary Search Trees

2/21/22

Daniel Bauer

Structural Induction over Trees

Proofs by Induction

- We are proving a theorem **T.** (“this property holds for all cases.”).
- Step 1: Base case. We know that **T** holds (trivially) for some small value.
- Step 2: Inductive step:
    - Inductive Hypothesis: Assume **T** holds for all cases up to some limit *k.*
    - Show that **T** also holds for *k+1.*
- This proves that **T** holds for any *k.*

Strong Induction Example

- Statement: Any integer $n \geq 2$ has a factorization into prime numbers.
- Base case: Clearly true for $n = 2$ because 2 is itself prime.
- Inductive step:
  - Hypothesis: Assume the statement holds **for all integers**, $2 \leq n \leq k$
  - Need to show that it also holds for $k+1$.

Strong Induction Example

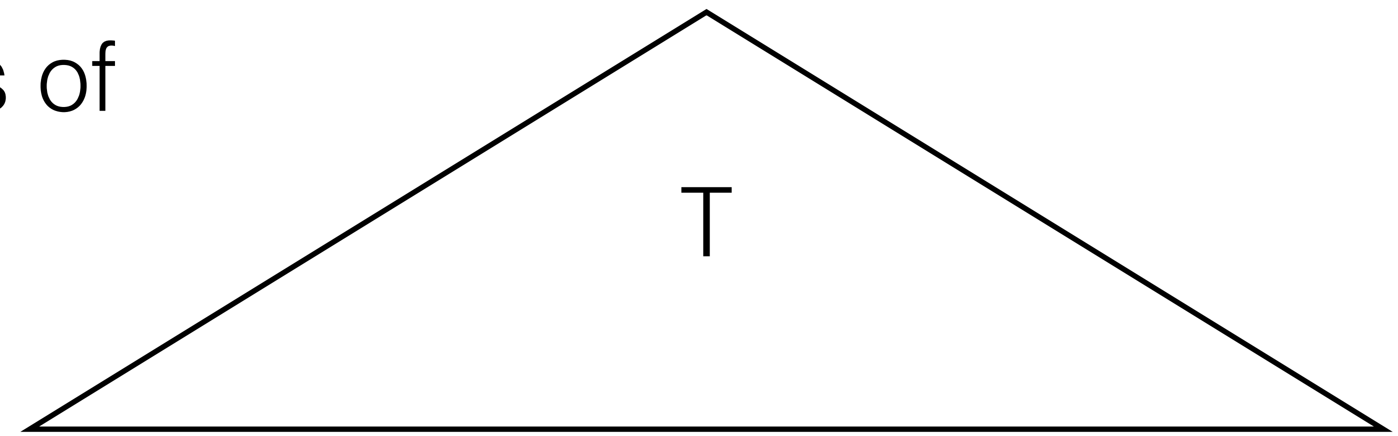
- Inductive step continued (show that $k+1$ can be factorized).
  - There are 2 cases.
    - Case 1: k+1 is a prime number.
      Then k+1 can be factored into itself.
    - Case 2: k+1 is not a prime number, so
      $\boldsymbol{k+1=p\cdot q}$ where $\boldsymbol{p\leq k}$ and $\boldsymbol{q\leq k}$.
      By the inductive hypothesis p and q have factorizations, so k+1 must have a factorization.

Proof by Structural Induction

- We often want to prove properties over tree structures.
- Can use induction:
  - Assume that the property holds for some smaller structure and show that it also needs to hold for larger structures.
  - Structural induction over trees comes in two forms: induction over the height, and induction over the number of nodes.

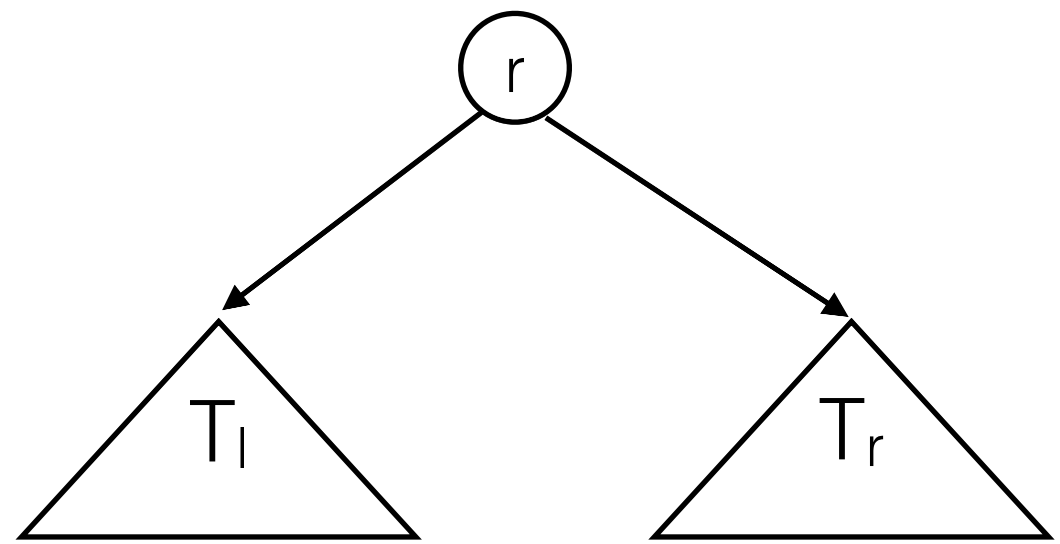
Binary Tree

- A binary tree *T* consists of
  - A root node *r*.
  - zero, one, or two subtrees.



Binary Tree

- A binary tree *T* consists of
  - A root node *r*.
  - zero, one, or two subtrees.

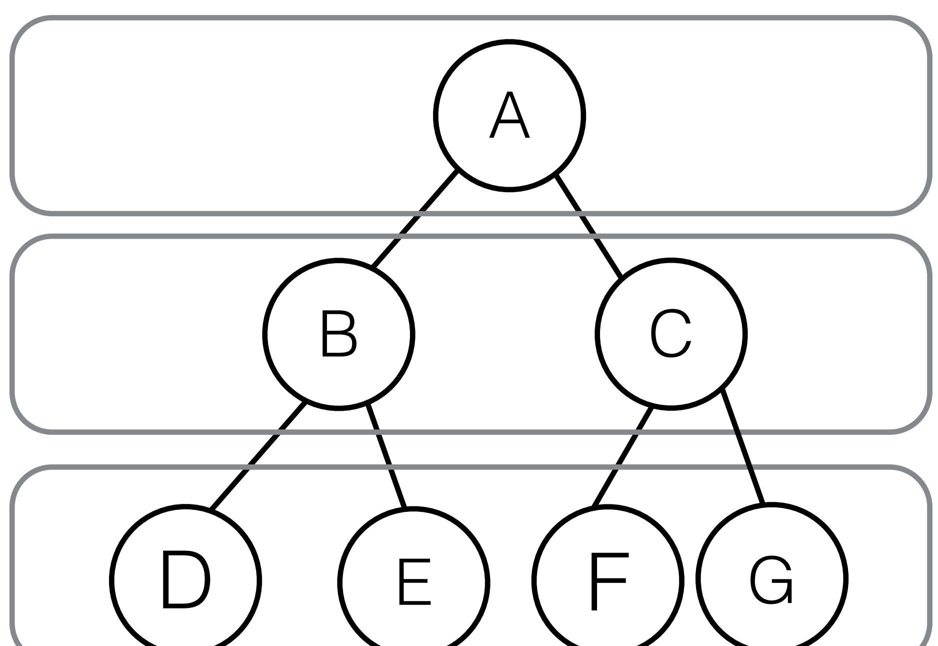


Structural Induction for Binary Trees - Induction over the height

- We want to show that a property holds for all binary trees.
- Base case: the property holds for a single node.
- Inductive step:
  - Assume the property holds for each possible subtree.
  - Show that it holds for all trees by combining the subtrees with a parent.

Perfect BinaryTree

- A perfect binary tree is one in which all levels are completely filled (including the leaf level).
  - Note that there are binary trees that are both full and complete, but not perfect.



Example Proof (Induction over height)

- Want to show:
  A *perfect* binary tree of height h has $2^{h+1} - 1$ nodes.

- Base case:
  A tree of height 0 has $2^{0+1} - 1 = 1$ nodes.

- Inductive step:

  - Hypothesis: Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

  - Show that any perfect binary tree of height $k + 1$ has $2^{k+2} - 1$ nodes. (next slide)

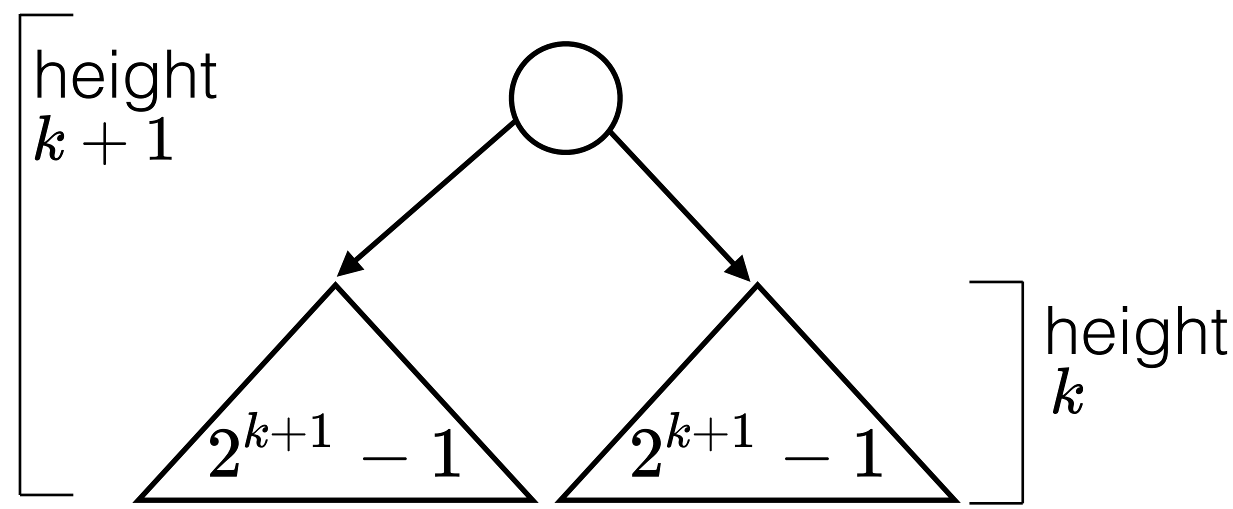
Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

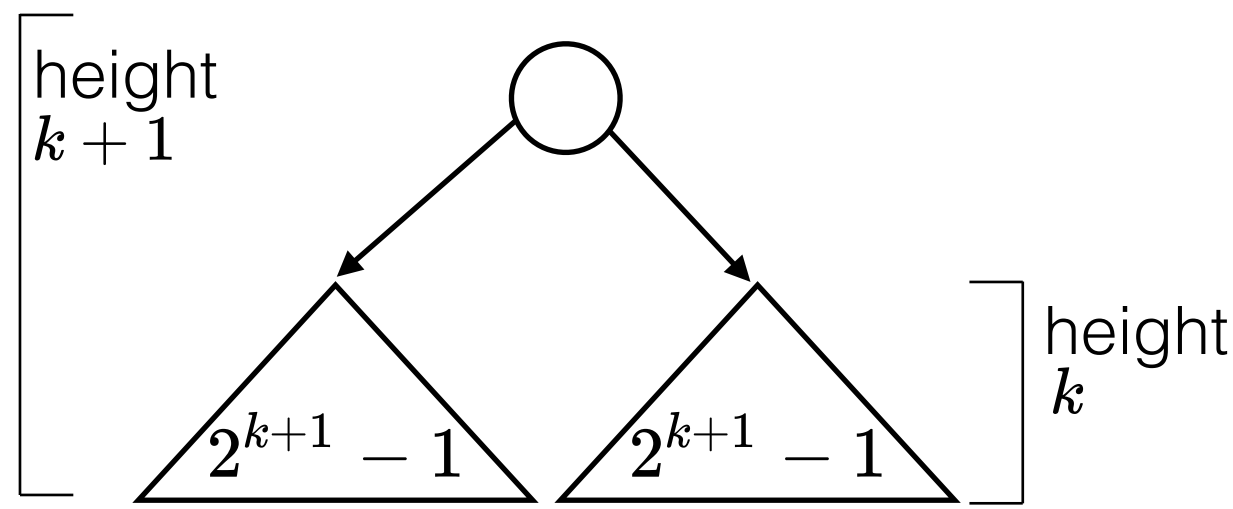
Recursive construction:



Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

Recursive construction:

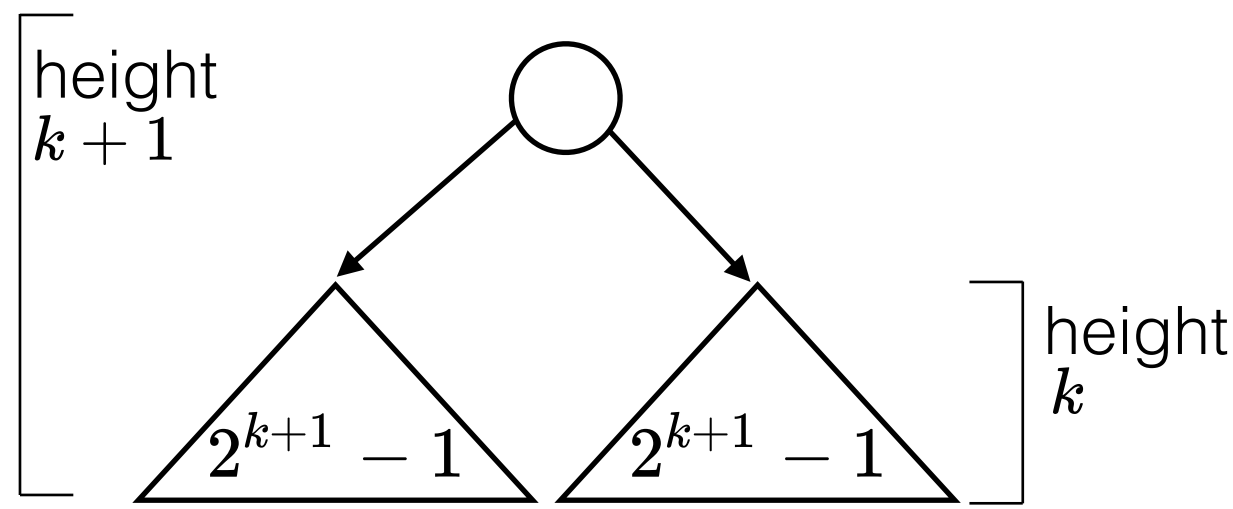


- The number of nodes of a perfect binary tree of height $k+1$ is

Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

Recursive construction:



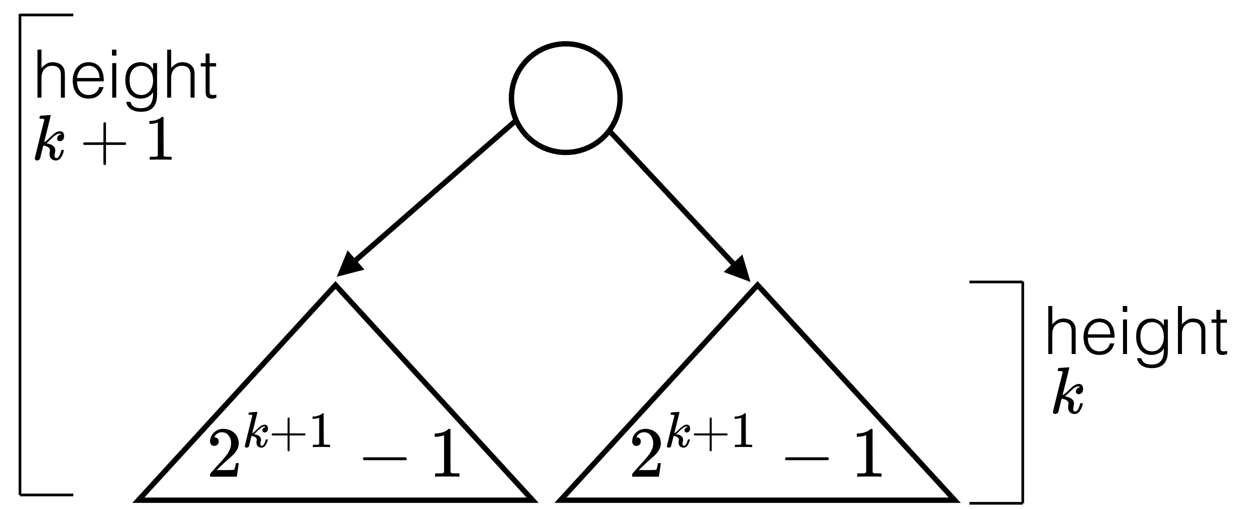
- The number of nodes of a perfect binary tree of height $k + 1$ is

$$2 \cdot (2^{k+1} - 1) + 1$$

Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

Recursive construction:



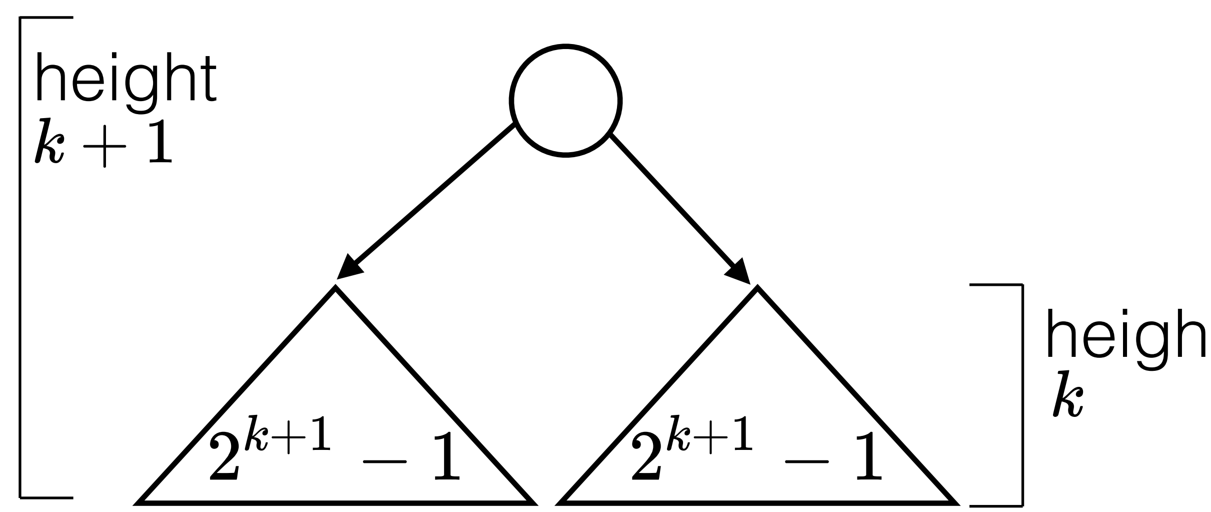
- The number of nodes of a perfect binary tree of height $k+1$ is

$$2 \cdot (2^{k+1} - 1) + 1 = 2 \cdot 2^{k+1} - 2 + 1$$

Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1} - 1$ nodes .

Recursive construction:



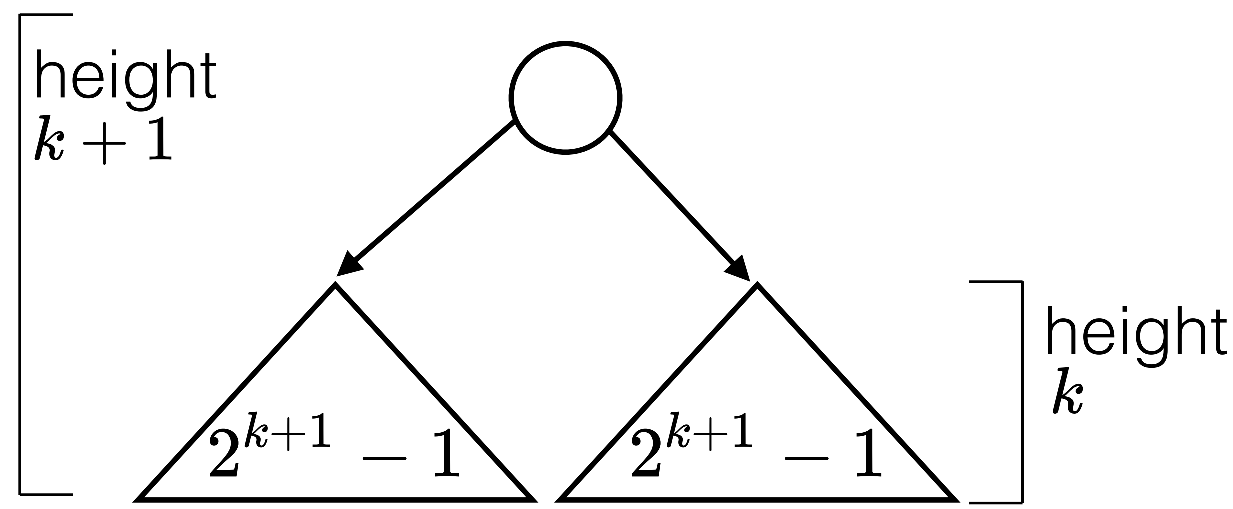
- The number of nodes of a perfect binary tree of height $k + 1$ is

$$
\begin{aligned}
2 \cdot (2^{k+1} - 1) + 1 &= 2 \cdot 2^{k+1} - 2 + 1 \\
&= 2^{k+2} - 2 + 1
\end{aligned}
$$

Example Proof (Inductive Step)

- Assume any perfect binary tree tree of height k has $2^{k+1}-1$ nodes .

Recursive construction:



- The number of nodes of a perfect binary tree of height $k+1$ is

$$\begin{aligned} 2\cdot(2^{k+1}-1)+1 &= 2\cdot 2^{k+1}-2+1 \\ &= 2^{k+2}-2+1 \\ &= 2^{k+2}-1 \qquad \blacksquare \end{aligned}$$

Minimum Height of a Binary Tree

- A binary tree of height h has *at most* $n = 2^{h+1} - 1$ *nodes.*
  (perfect tree is an upper bound)

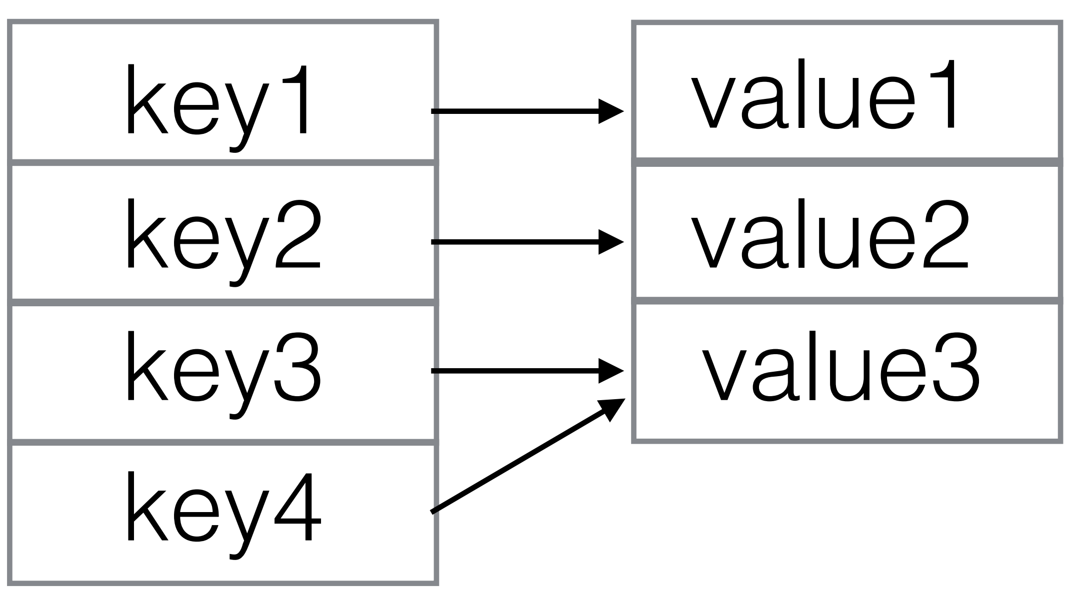
- Therefore, a binary tree of n nodes has at least height $log(n+1) - 1.$

$$n + 1 = 2^{h+1}$$
$$log(n+1) = h + 1$$
$$log(n+1) - 1 = h$$

Binary Search Trees

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)

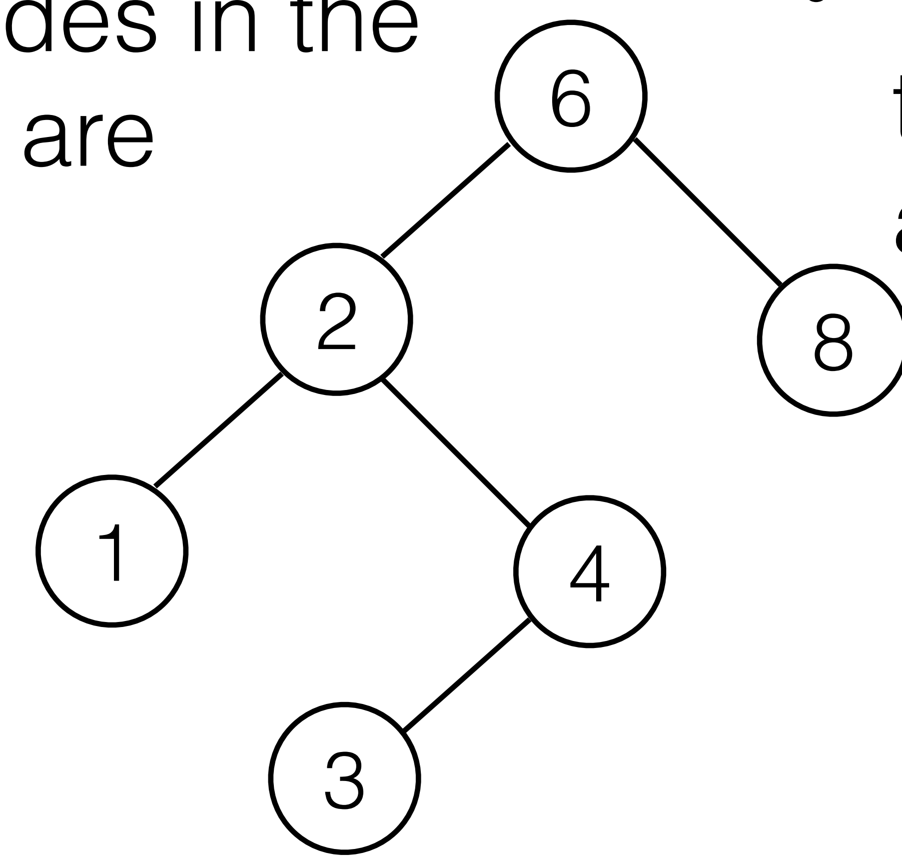


How do we implement map operations efficiently?

Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)
- For every node n with key x

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

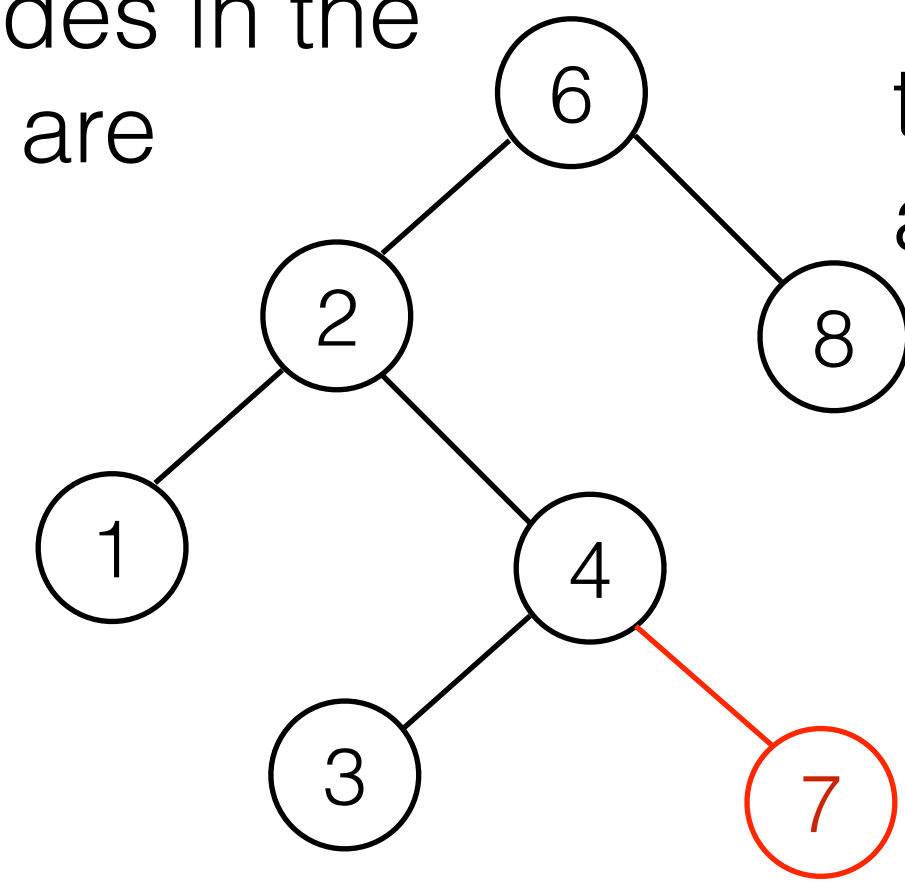


- The key of all nodes in the right subtree of n are larger than x.

Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)
- For every node n with key x

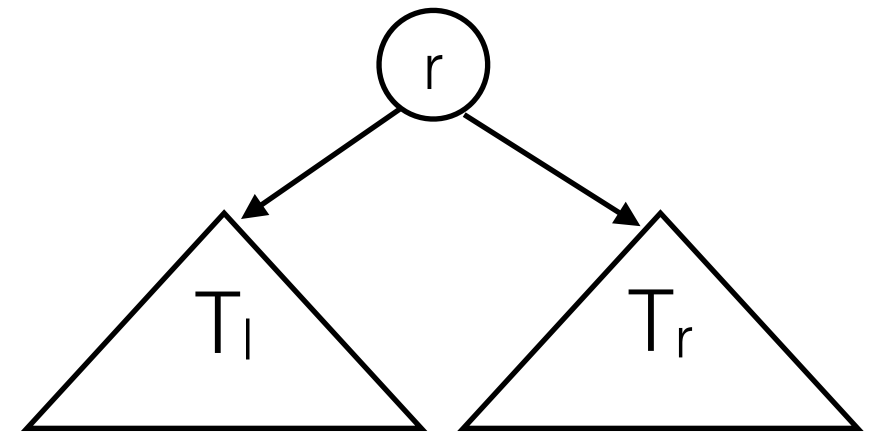
- the key of all nodes in the left subtree of n are smaller than x.



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

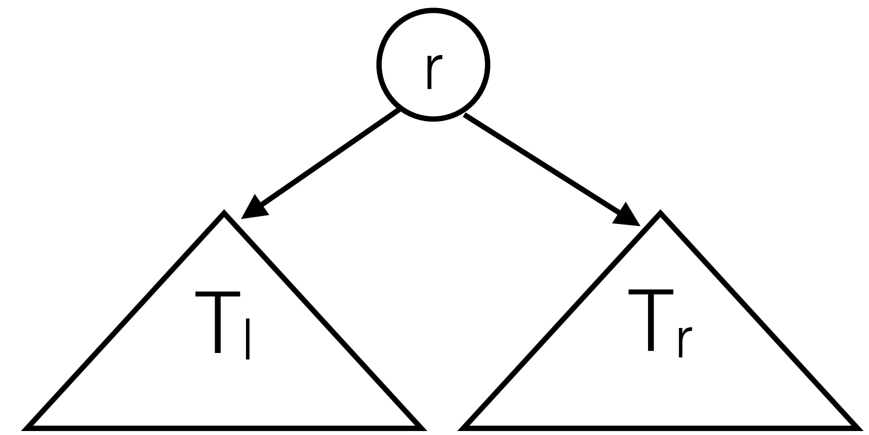
This is not a search tree

Binary Search Tree (BST) ADT



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

Binary Search Tree (BST) ADT



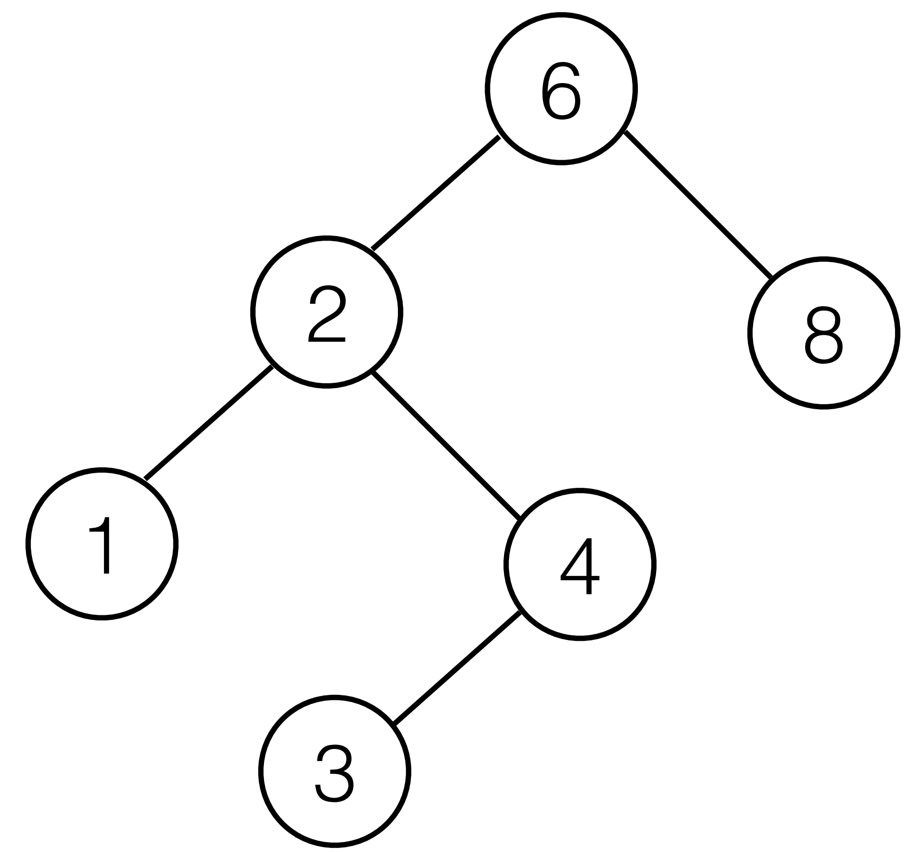
- A *Binary Search Tree T* consists of
  - A root node *r* with key r_{item}
  - *A*t most two non-empty subtrees T_l and T_r, connected by a directed edge from r.
- T_l and T_r satisfy the BST property:
  - For all nodes *s* in T_l, $s_{item} < r_{item}$.
  - *F*or all nodes *t* in T_r, $t_{item} > r_{item}$.
- No key appears more than once in the BST.

BST operations

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

BST operations: `contains`

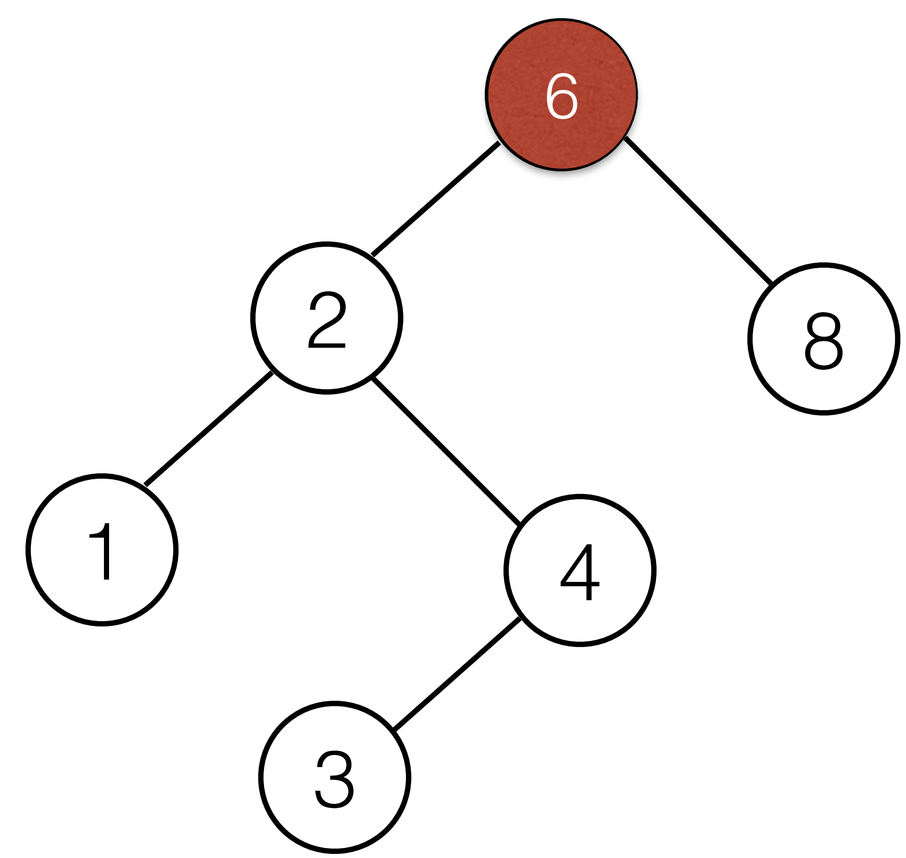
```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    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
}
```

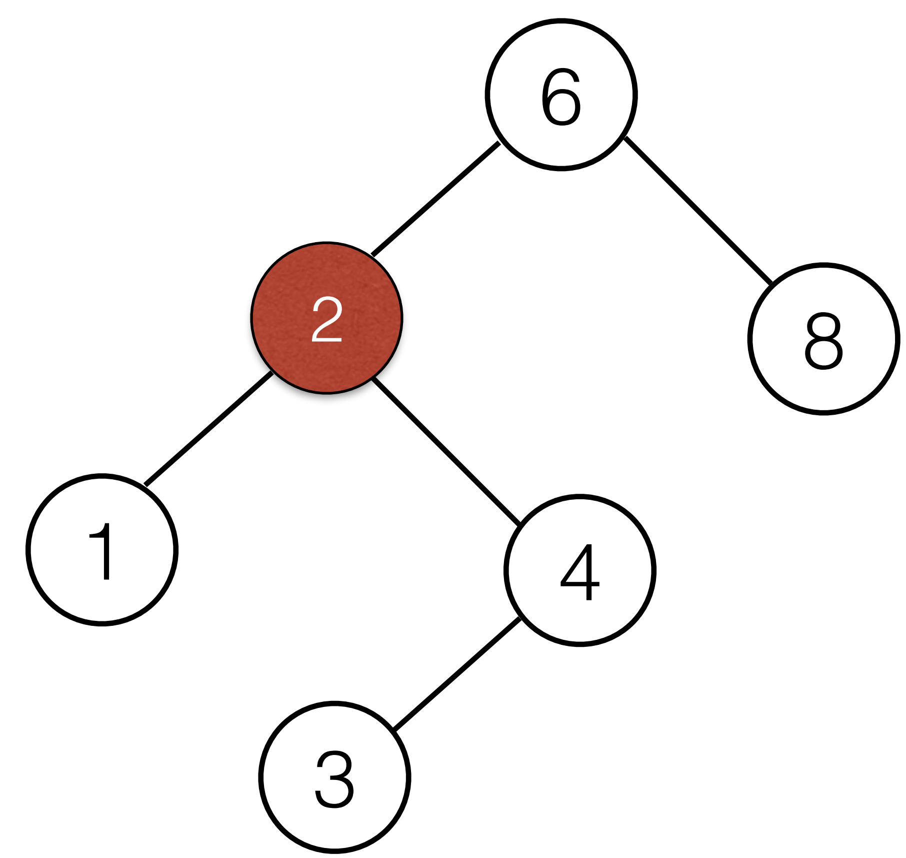
`contains(3)`



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
}
```

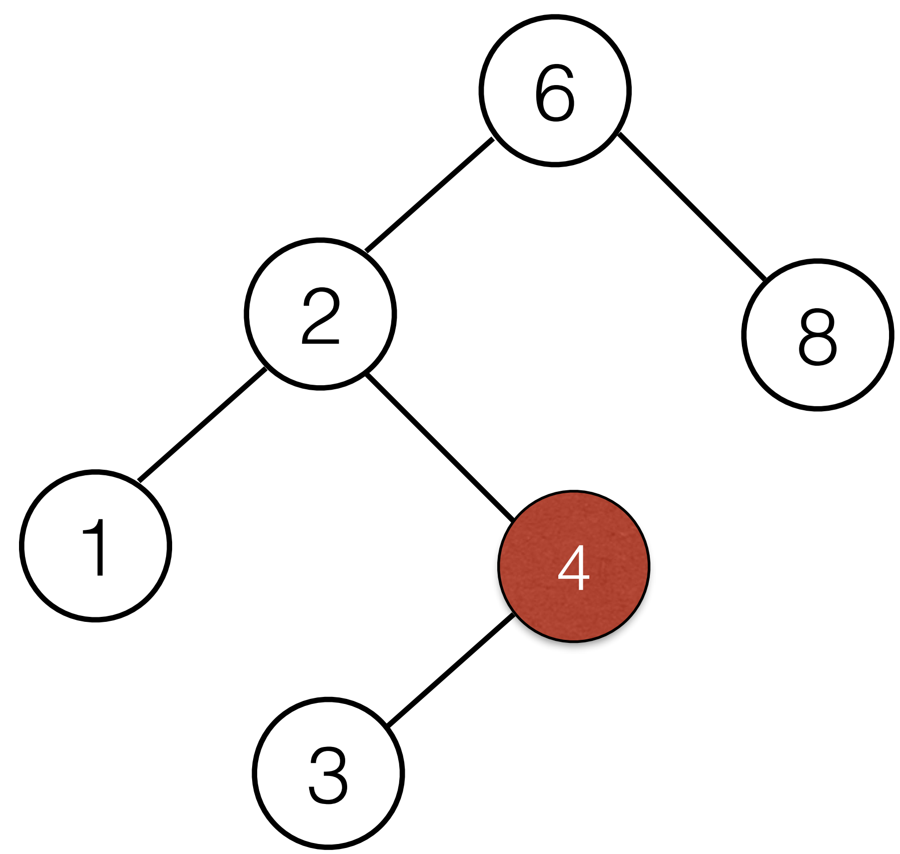
`contains(3)`



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
}
```

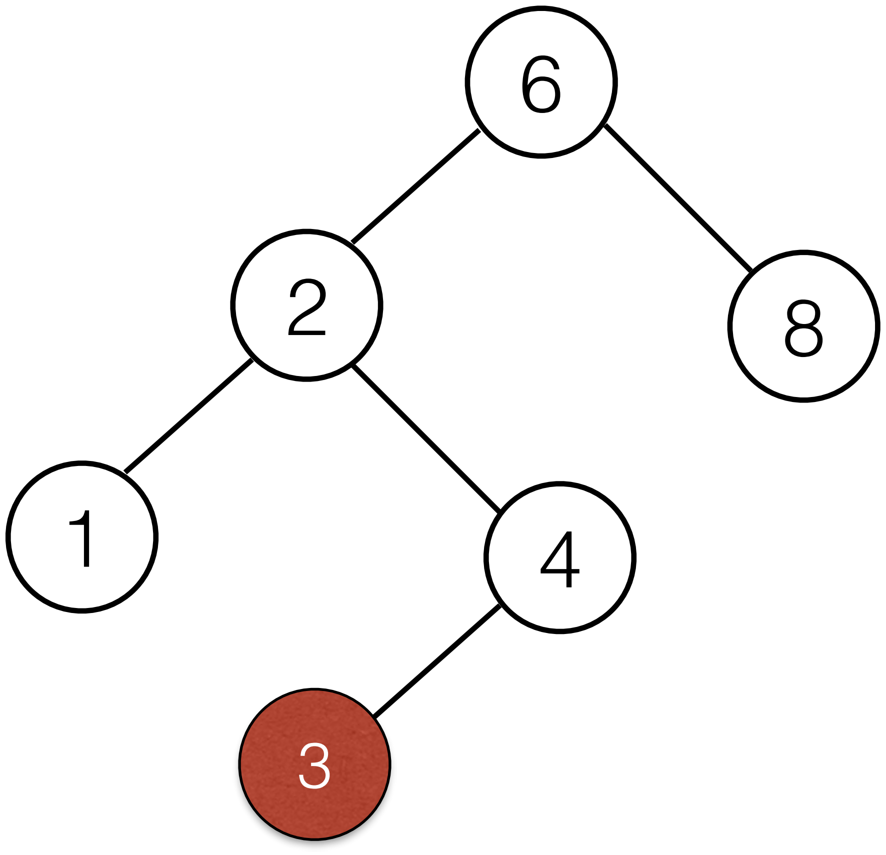
`contains(3)`



BST operations: `contains`

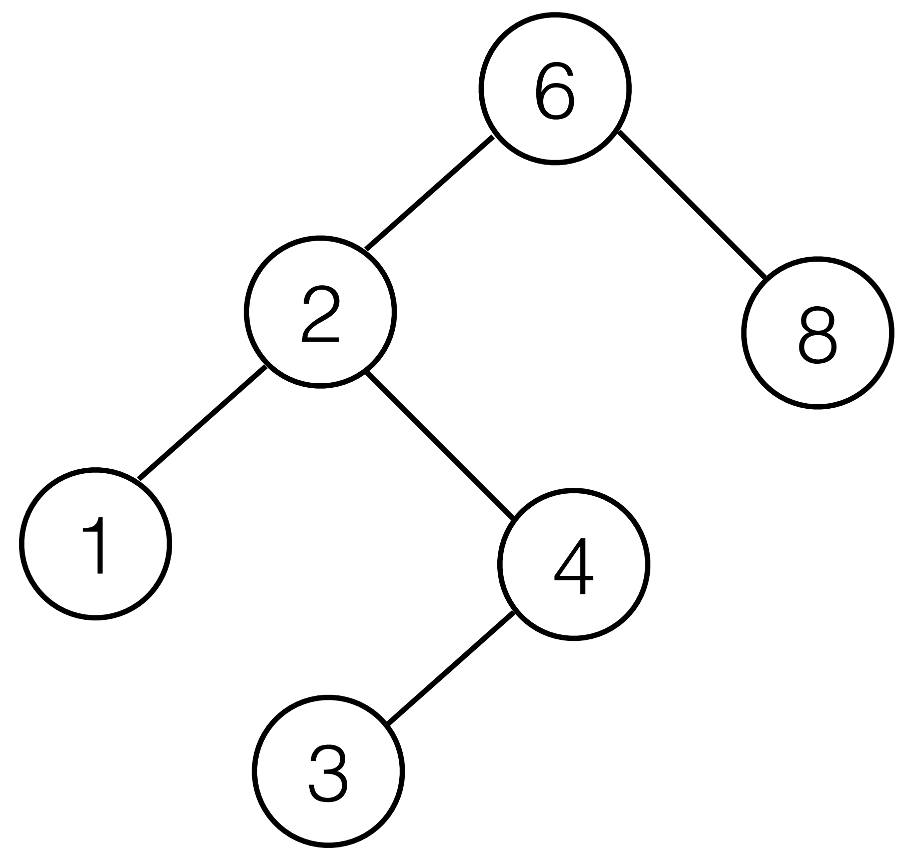
```
private boolean contains( Integer x, BinaryNode t ) {
    if( t == null )
      return false;

   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 );
}
```

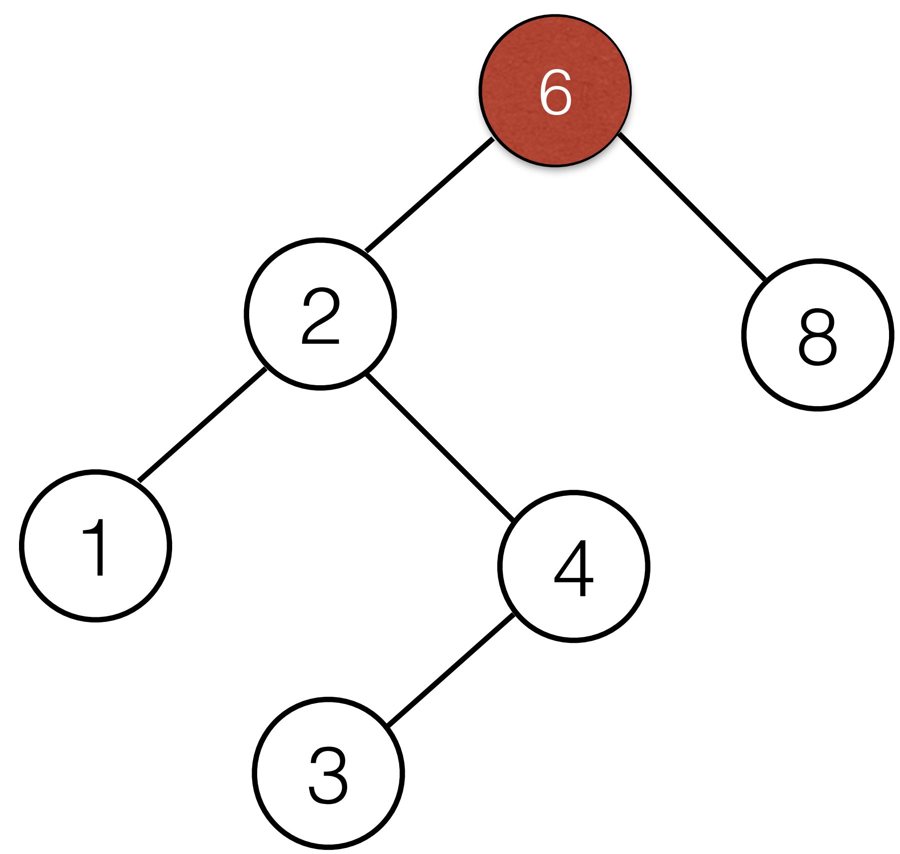


findMax is equivalent.

BST operations: `findMin`

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

`findMin()`

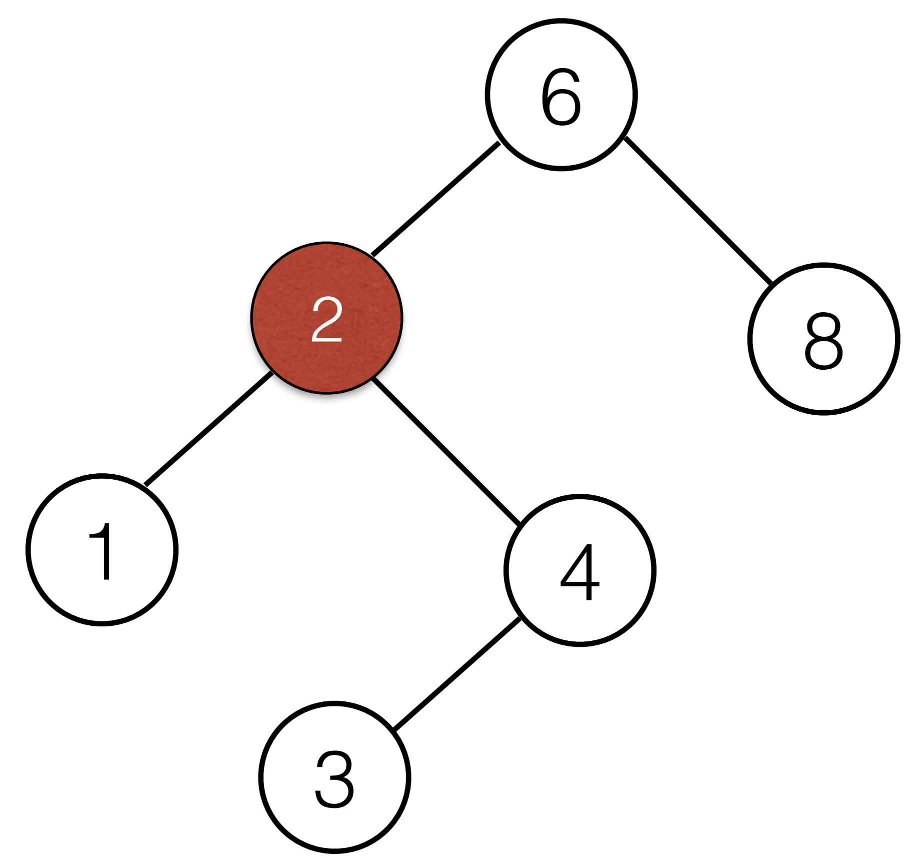


findMax is equivalent.

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()`

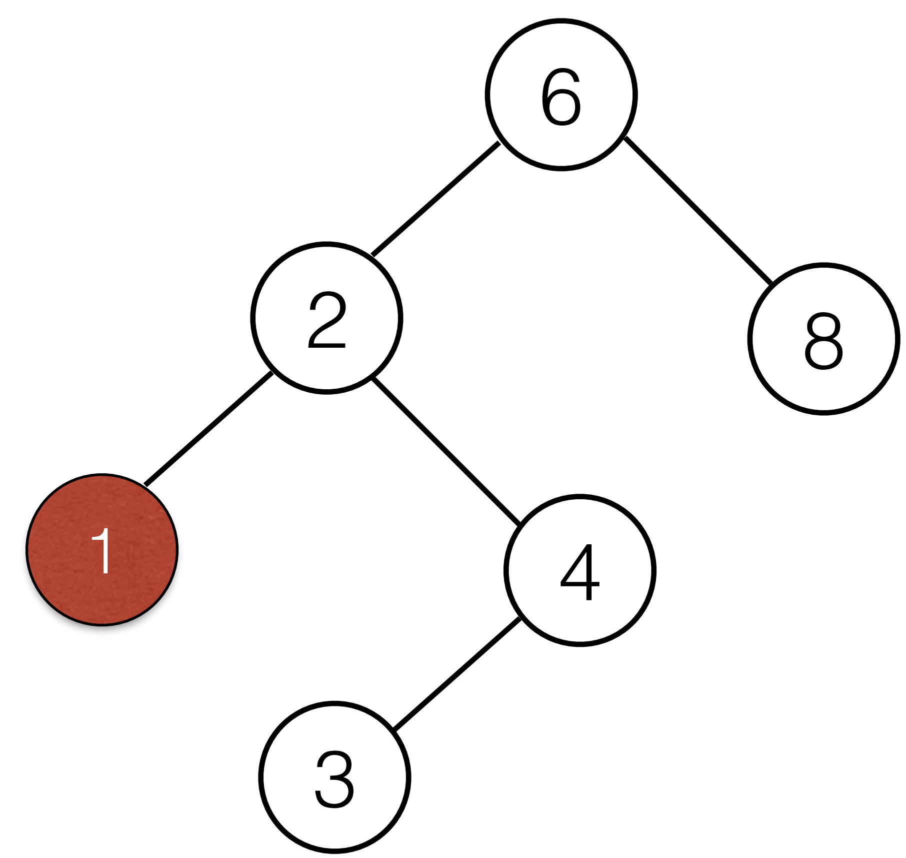


findMax is equivalent.

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()`

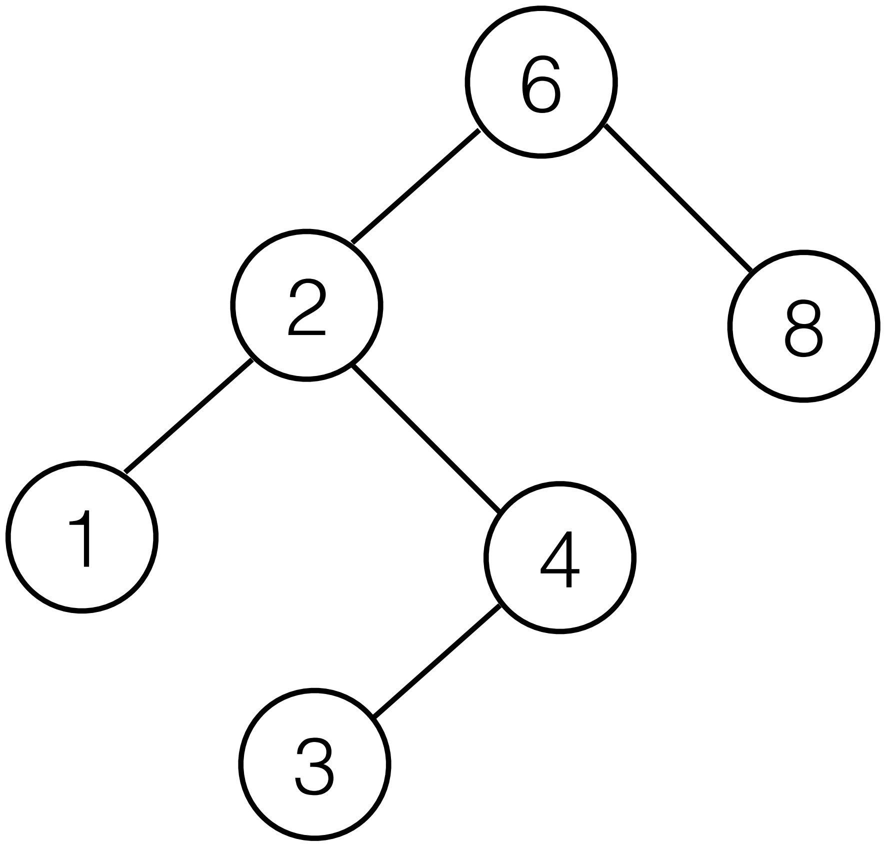


findMax is equivalent.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf 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;
}
```



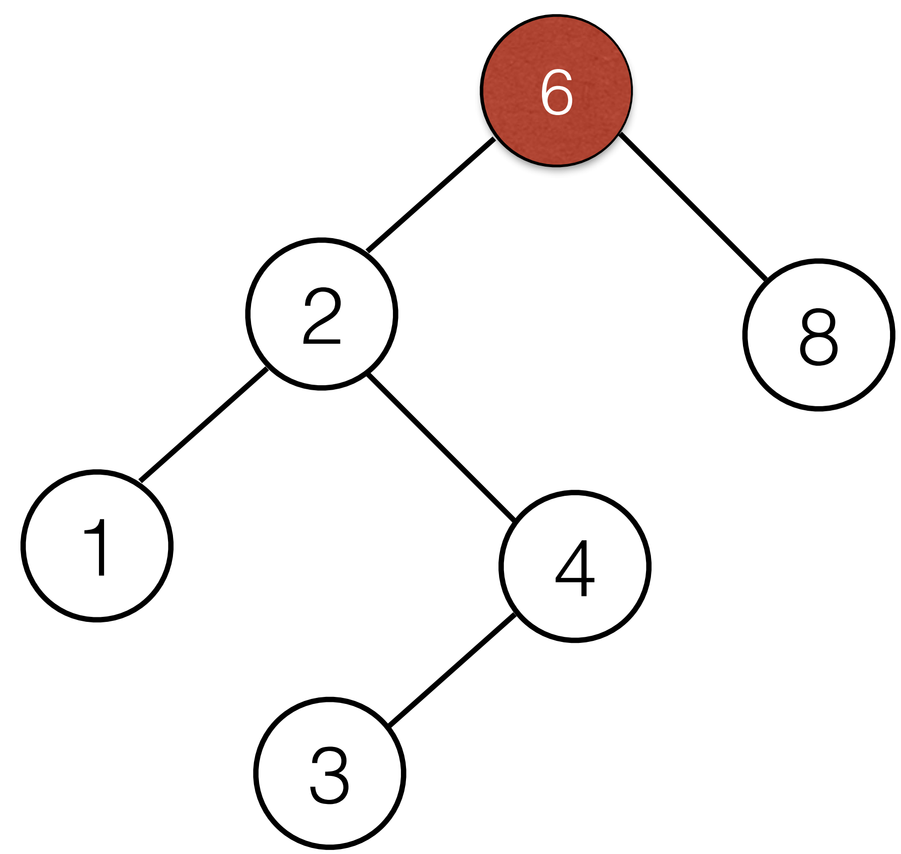
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf 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)`



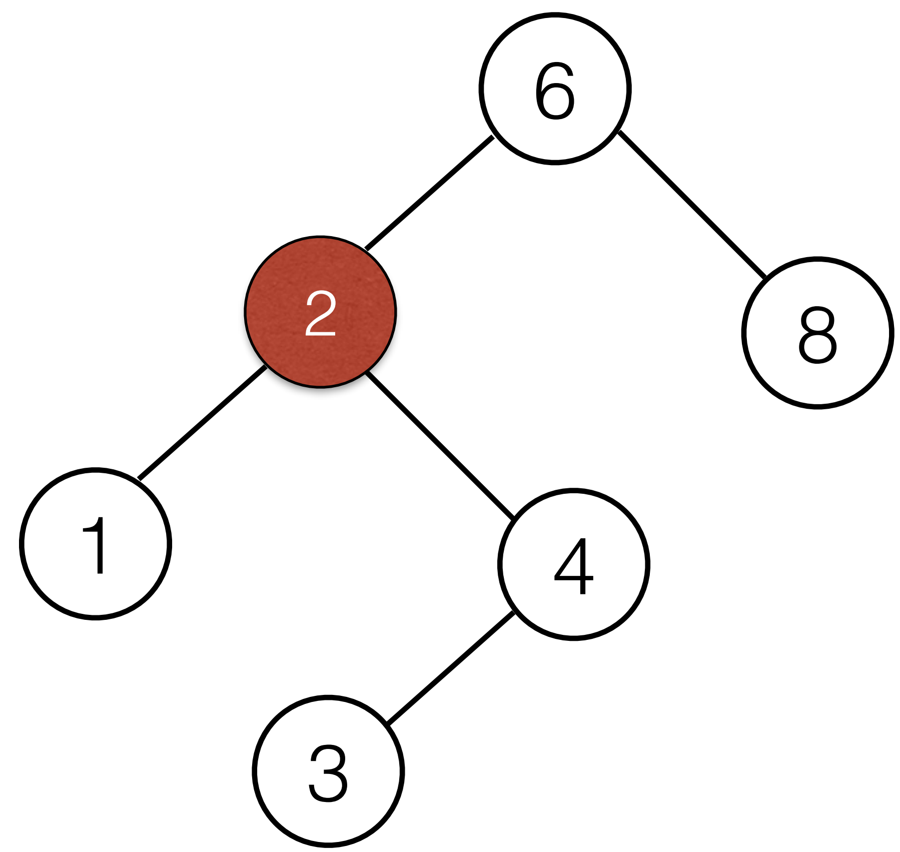
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf 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)`



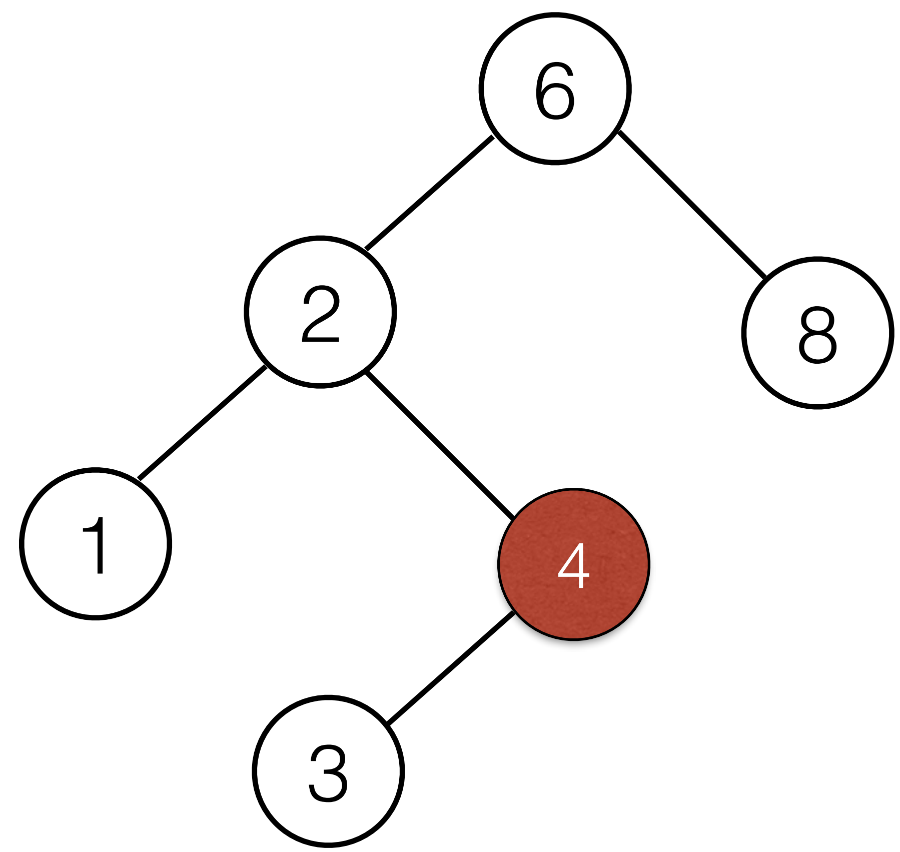
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf 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)`



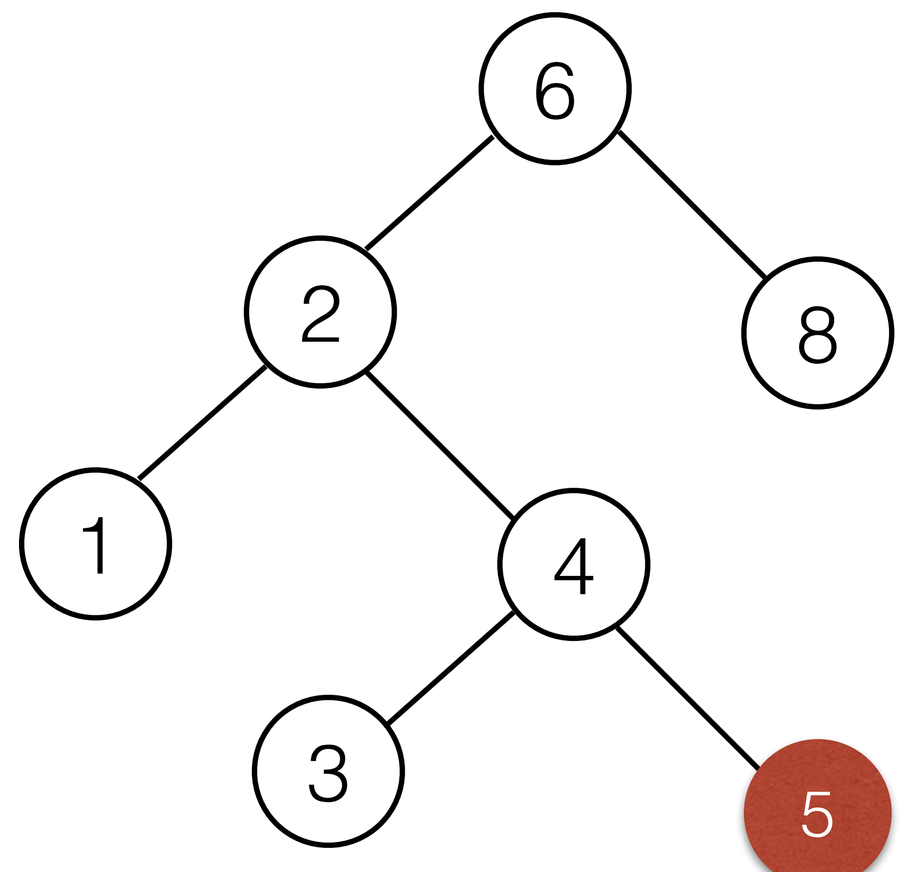
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf 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;
}
```

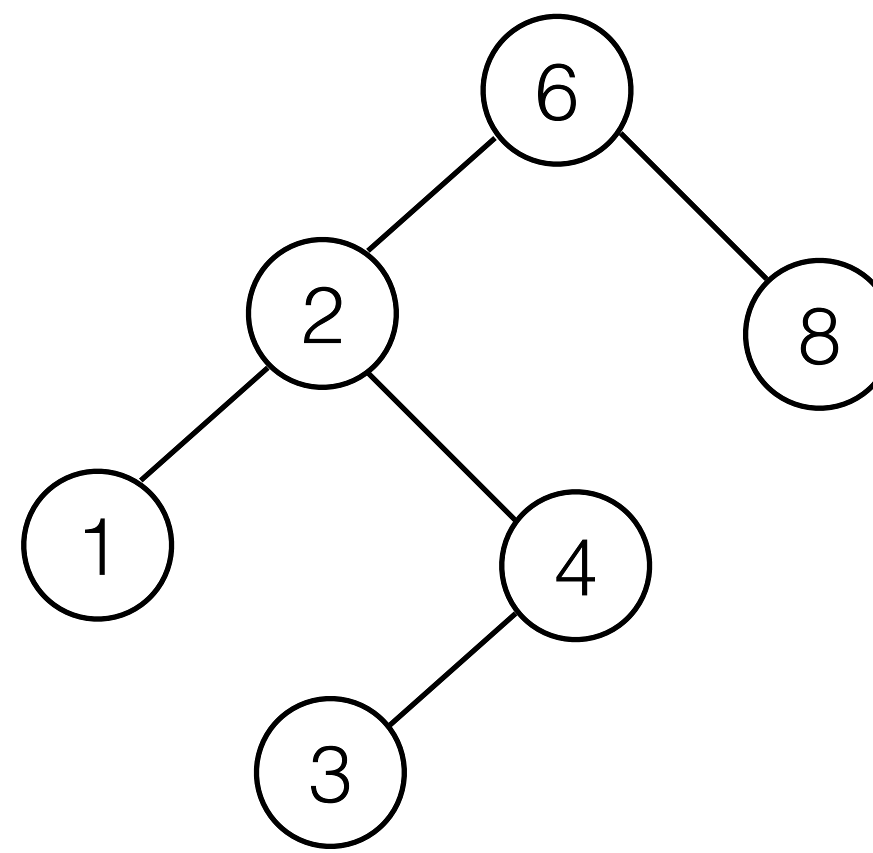
`insert(5)`



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.

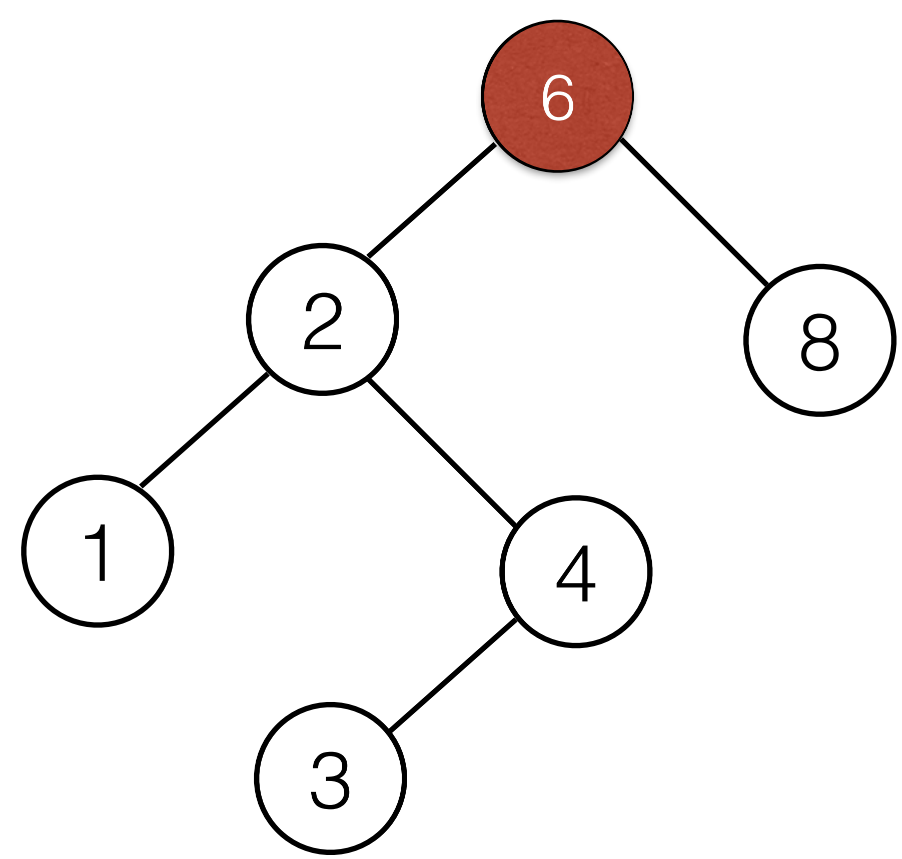


Maintains the BST property.

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)`

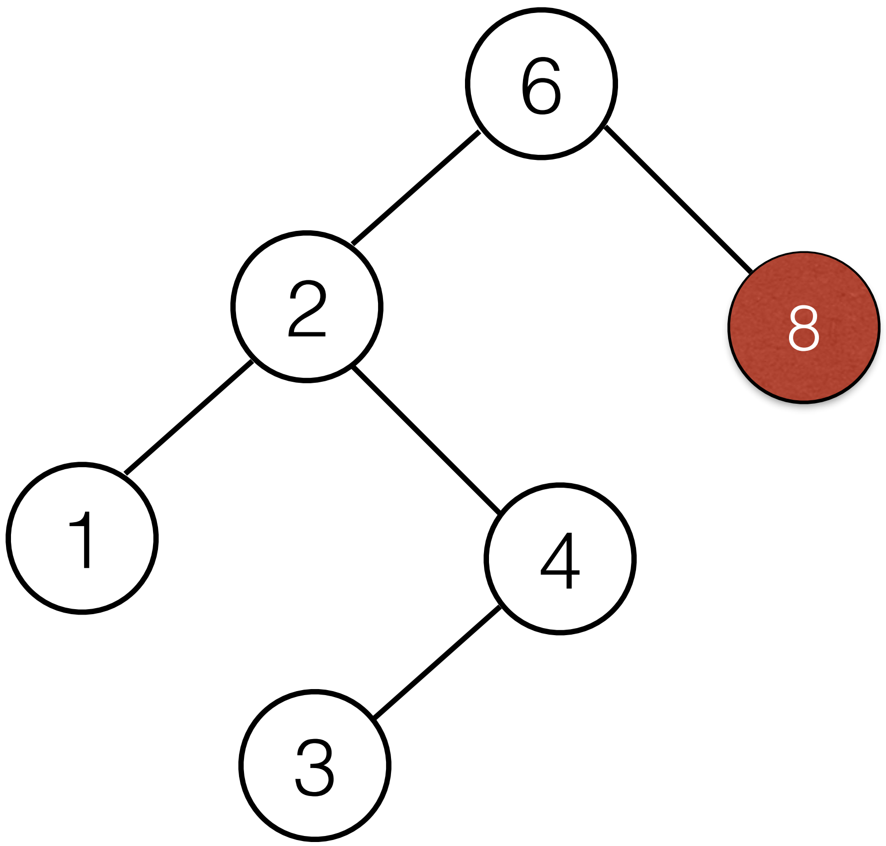


Maintains the BST property.

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)`

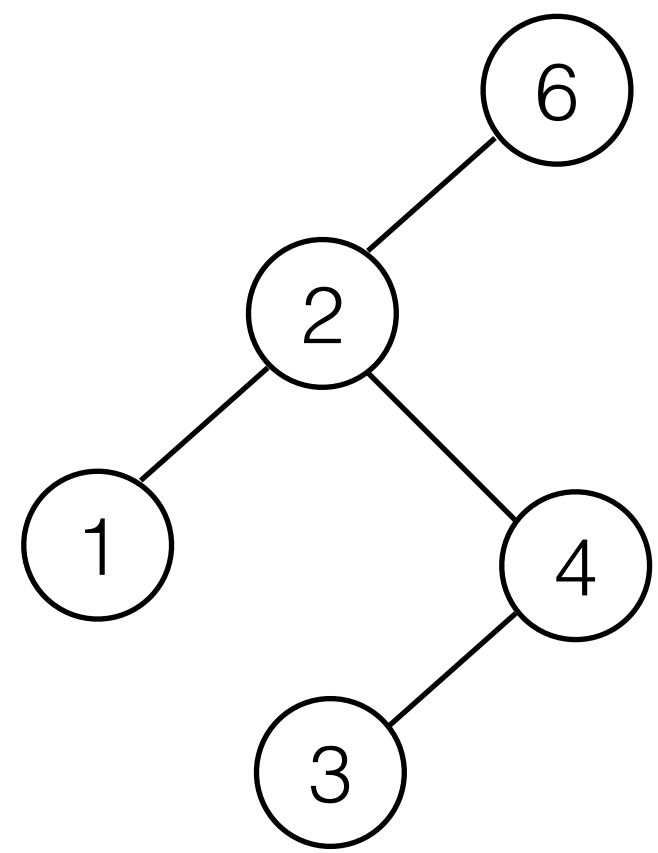


Maintains the BST property.

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)`

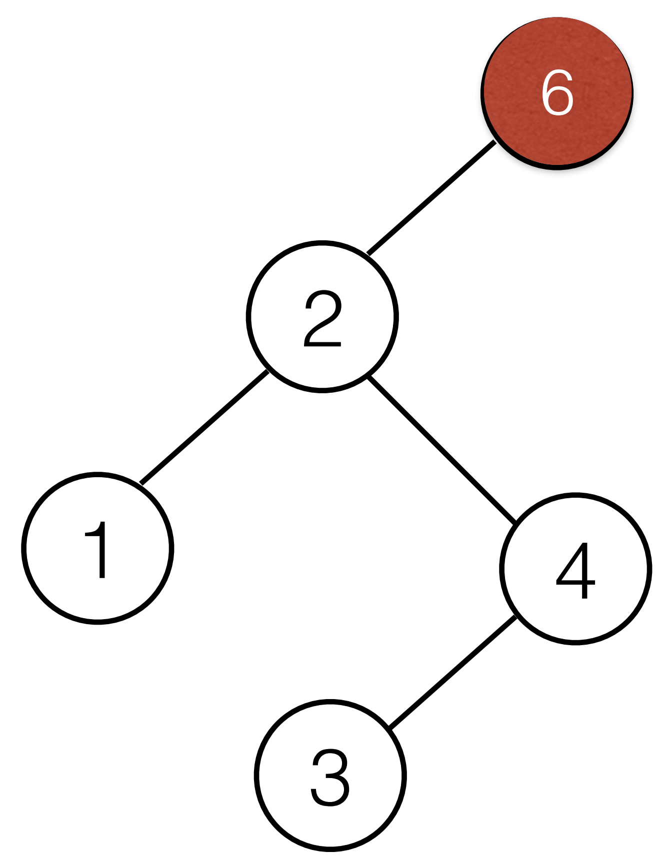


Maintains the BST property.

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)`

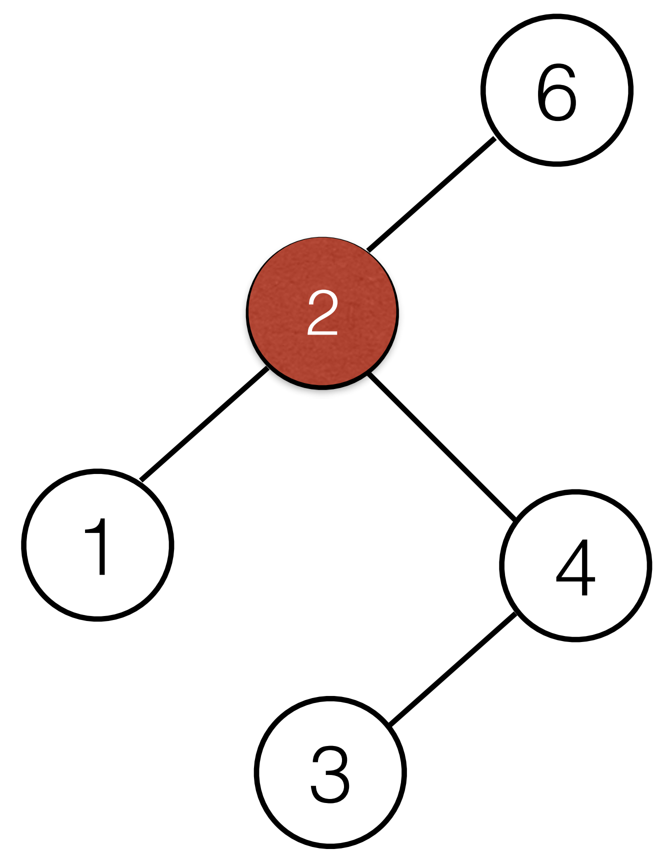


Maintains the BST property.

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)`

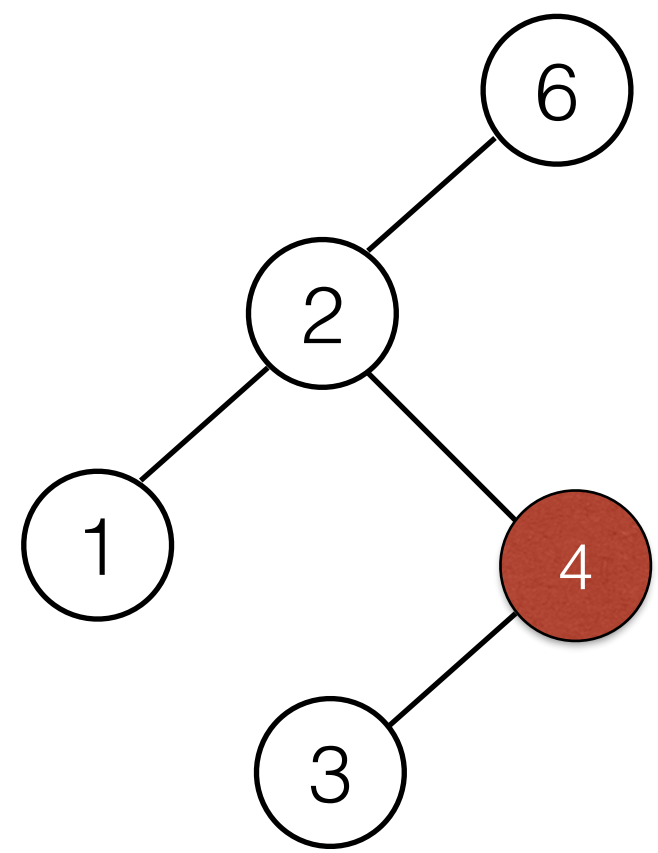


Maintains the BST property.

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)`

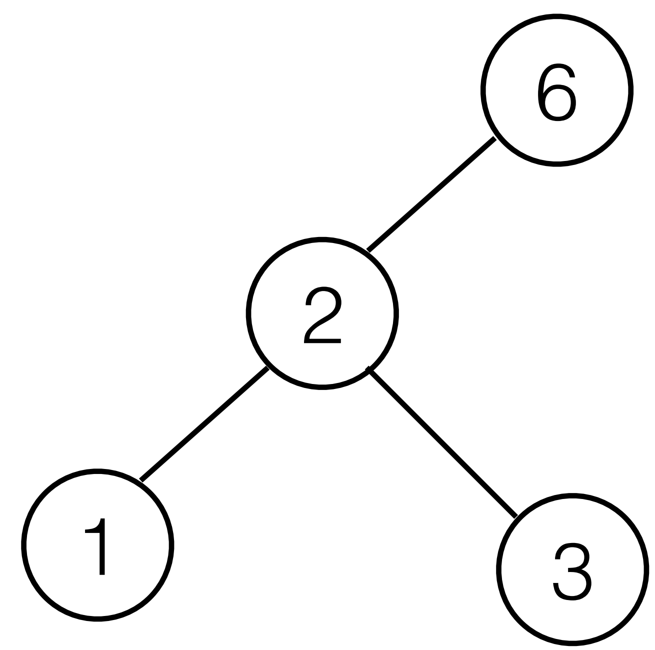


Maintains the BST property.

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)`

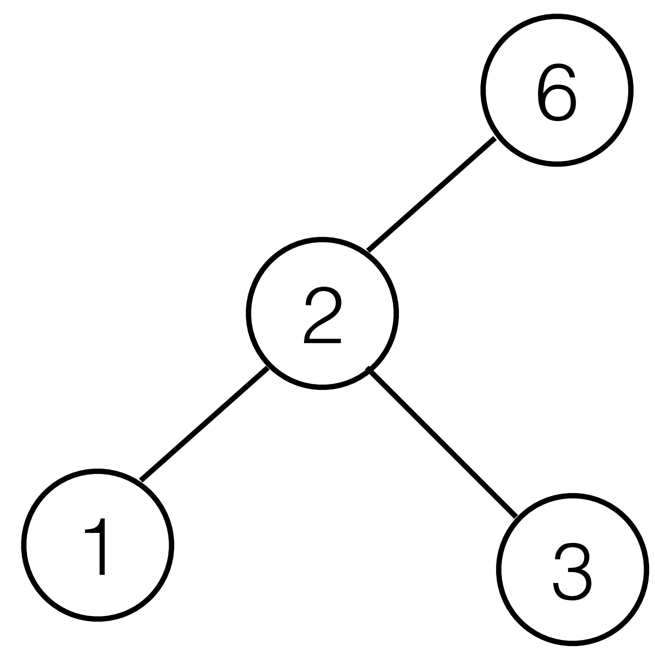


Maintains the BST property.

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)`

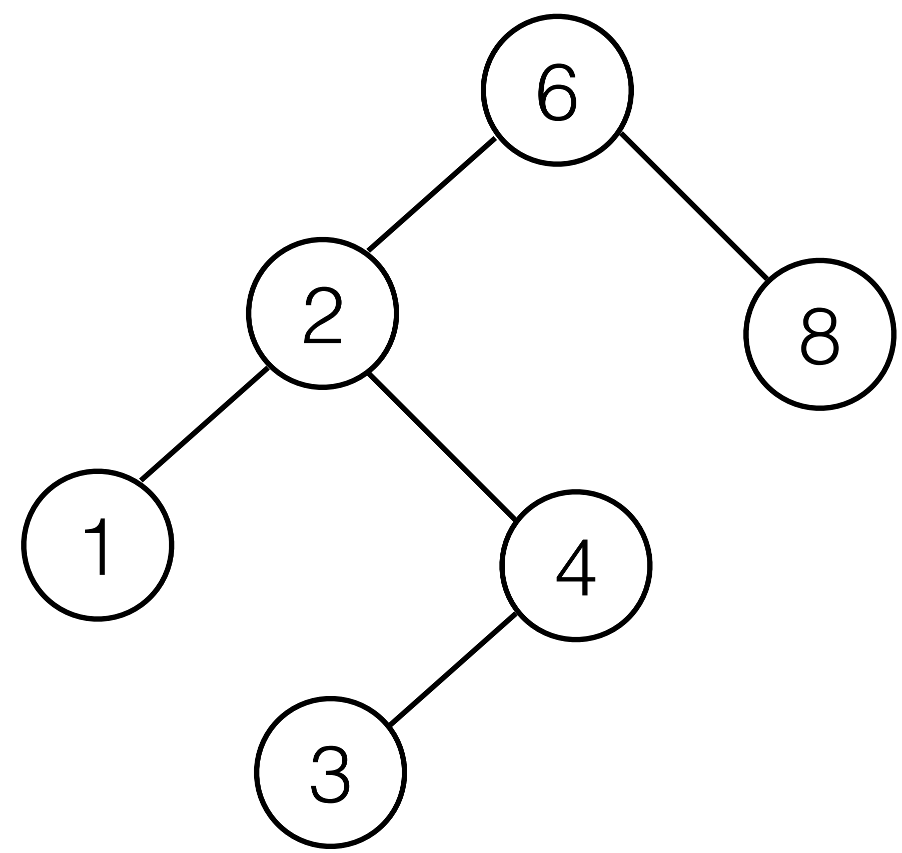


Maintains the BST property.

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 that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

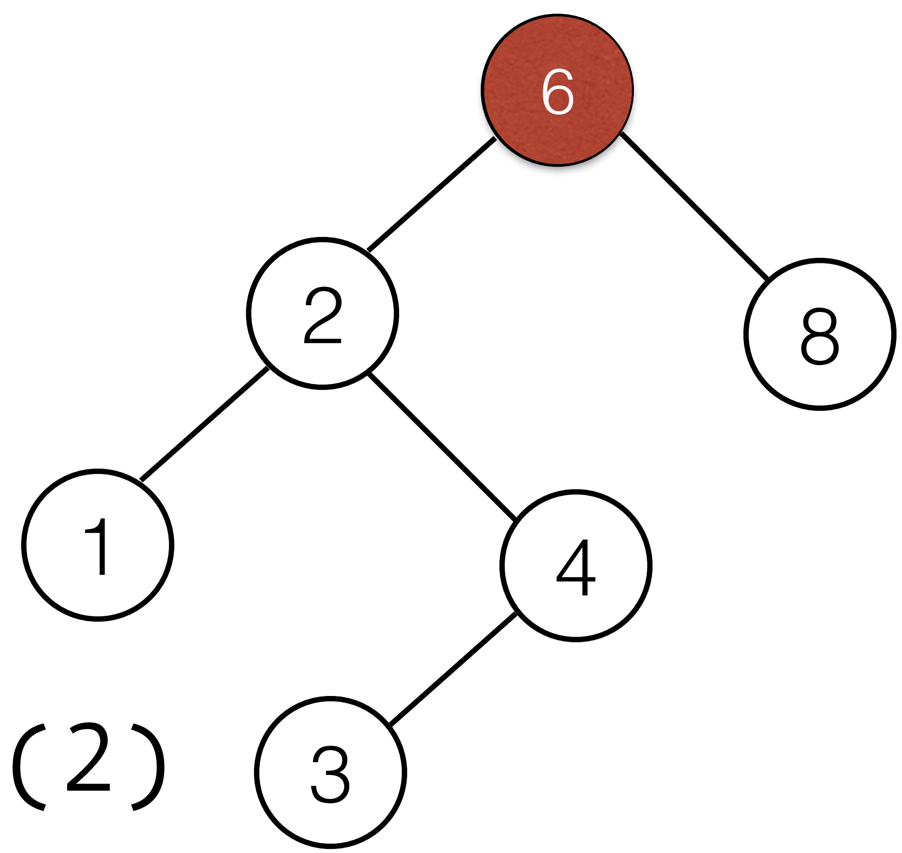


BST operations: `remove`

- 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 that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

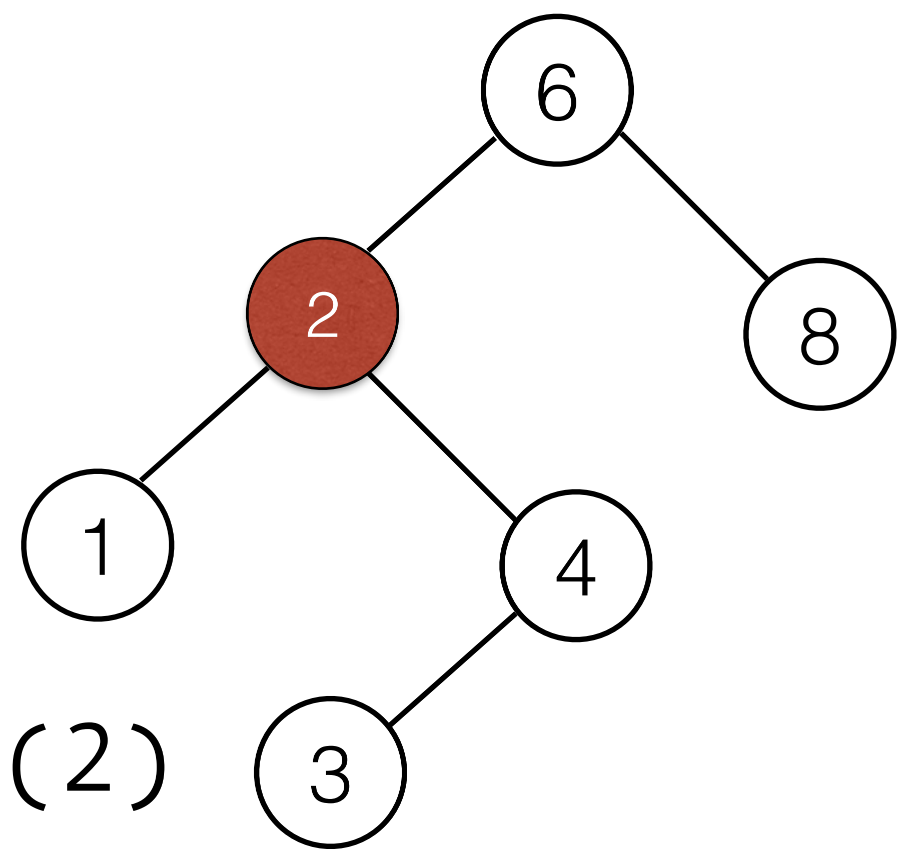


BST operations: `remove`

- 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 that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

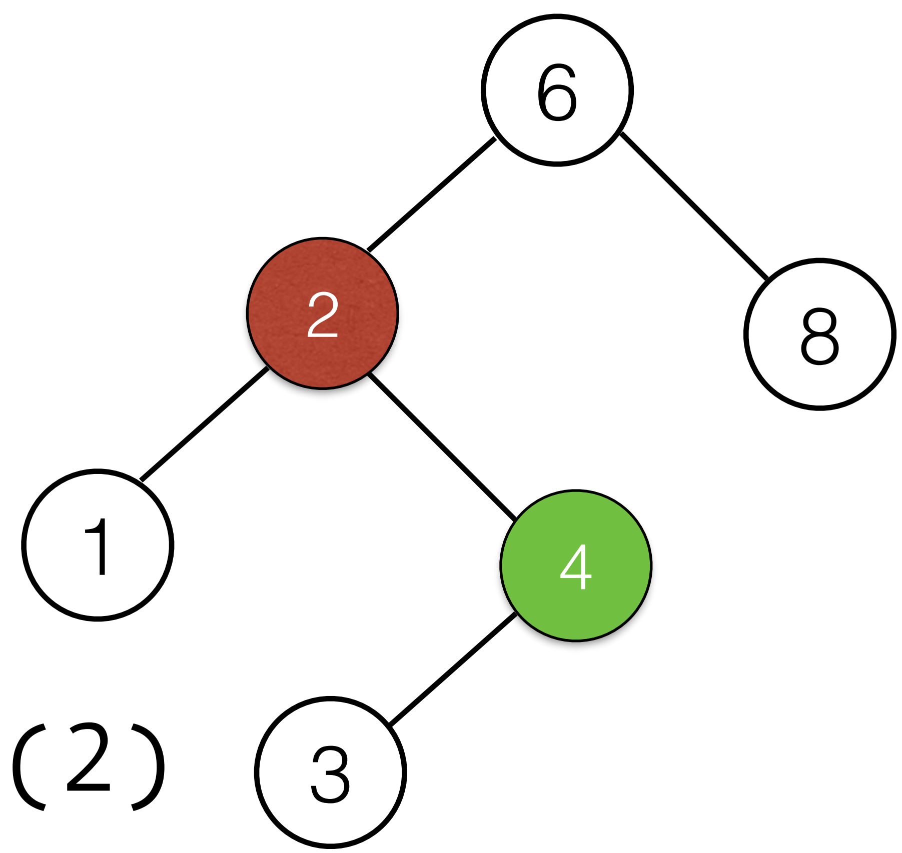


BST operations: `remove`

- 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 that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

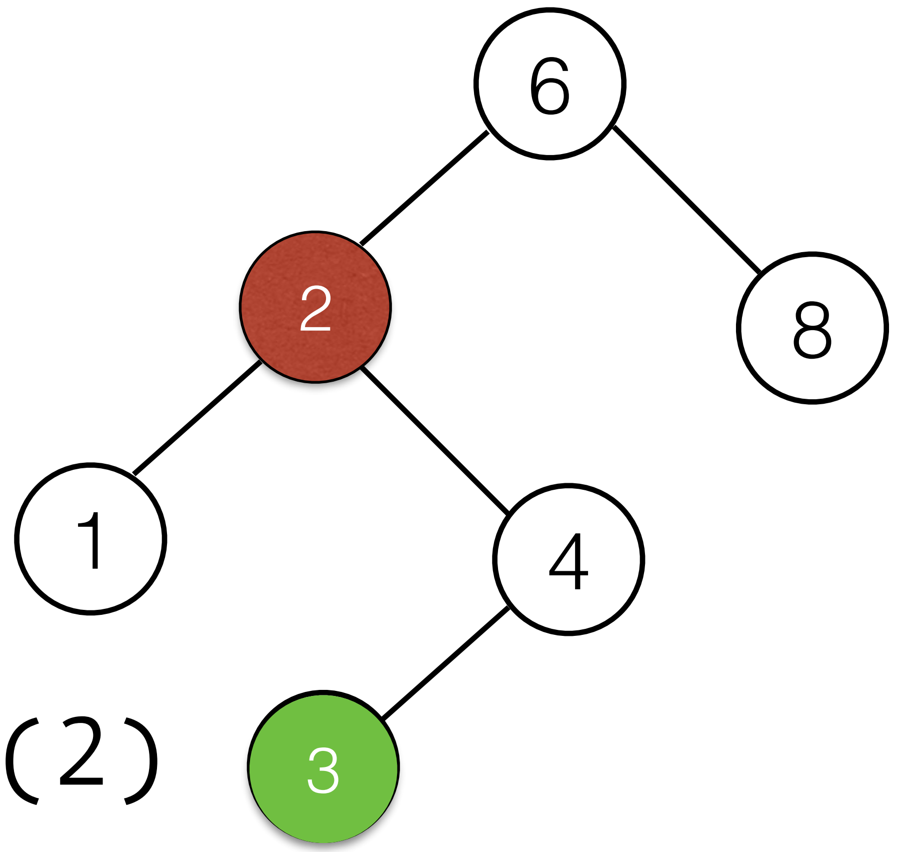


BST operations: `remove`

- 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 that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

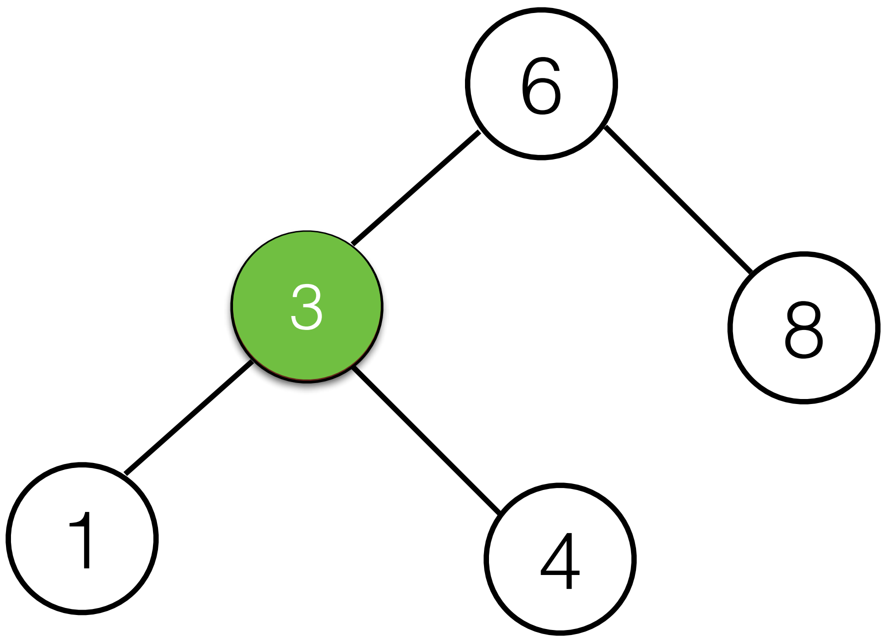
`remove(2)`



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.

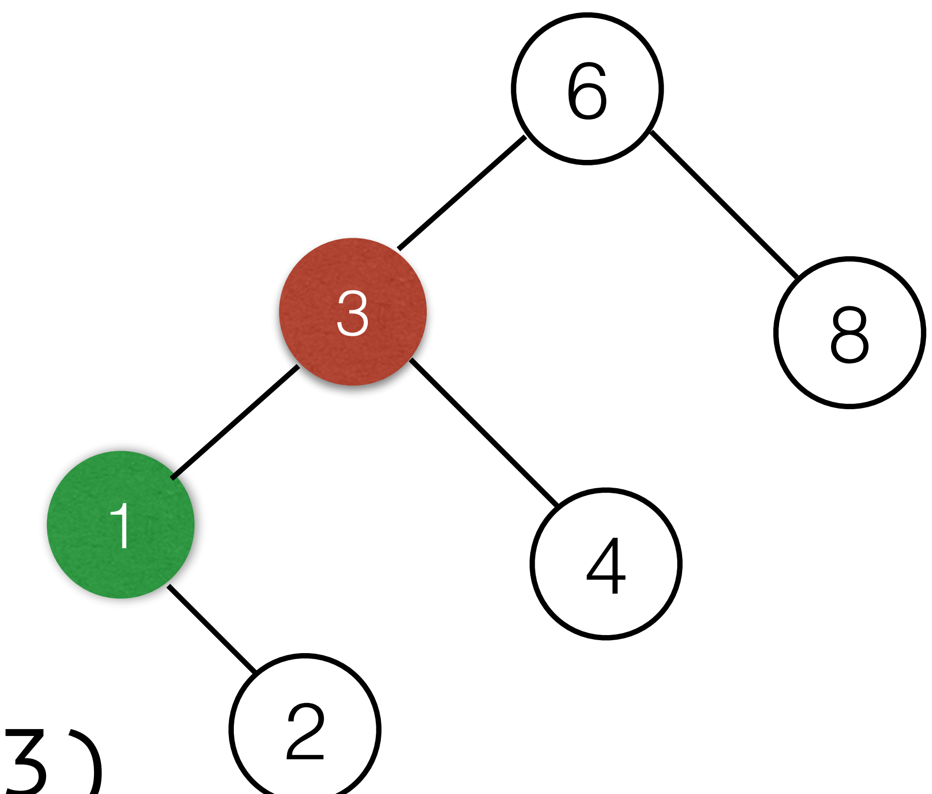
To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.



`remove(2)`

BST operations: `remove`

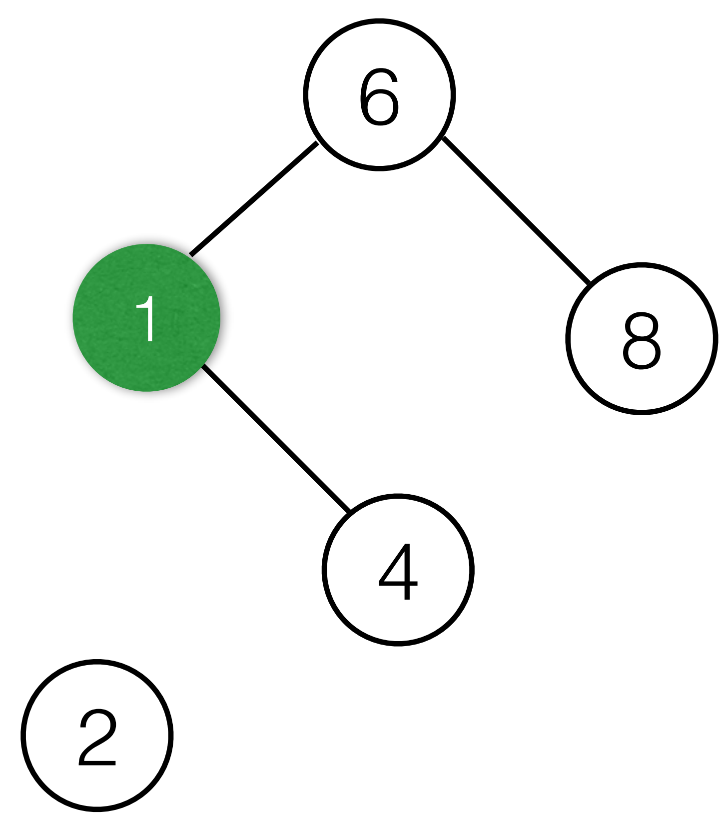
- Why not just replace s with the root of t_{left}?



`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;
}
```

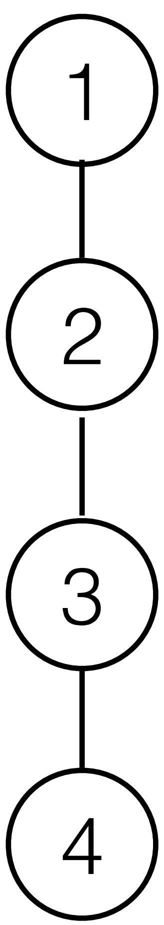
Running Time Analysis for BST Operations

- 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.

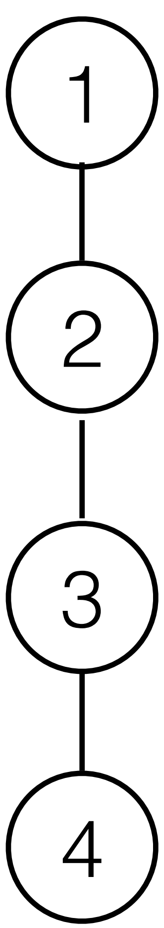
- Worst case: T does not branch *height(T)=O(N)*



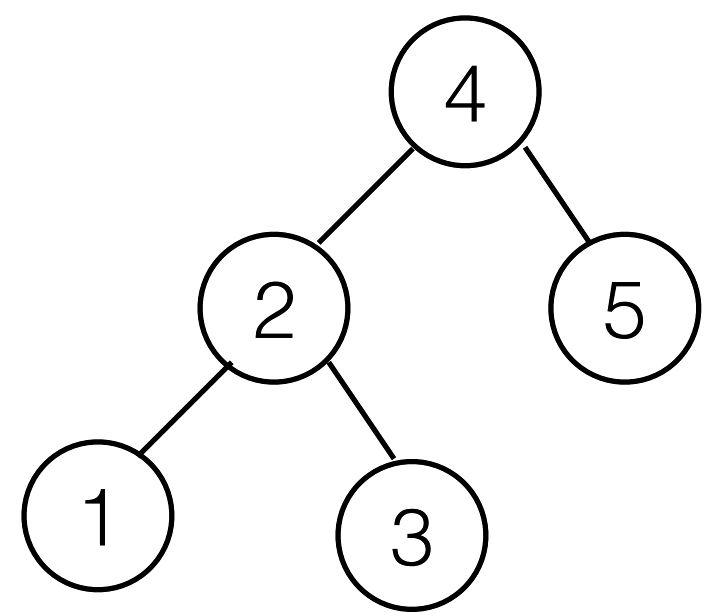
Worst and Best Case Height of a BST

- Assume we have a BST with N nodes.

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



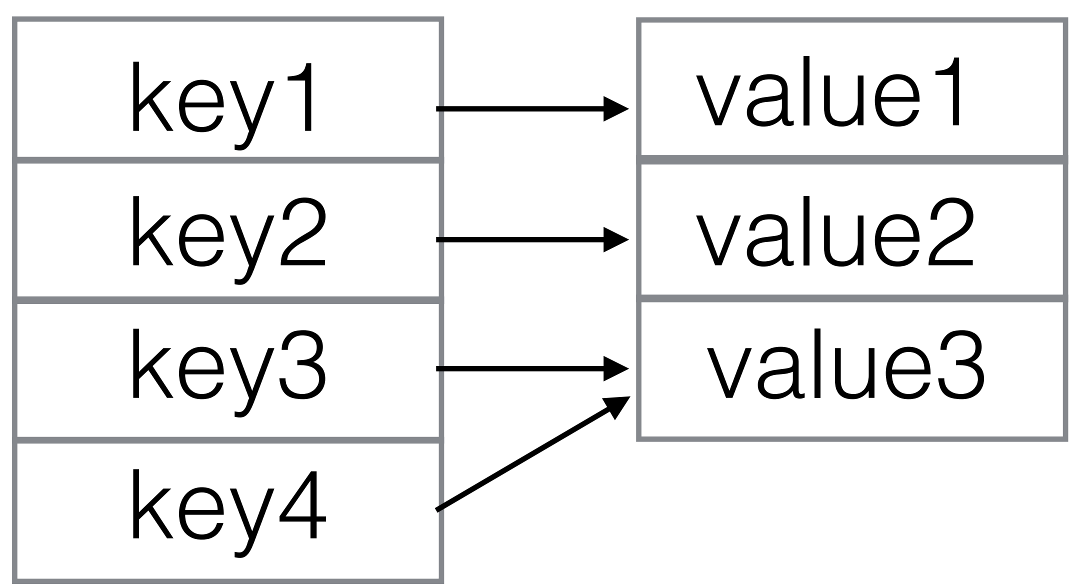
- Best case: *height(T)=O(log N)*



complete binary tree.

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)

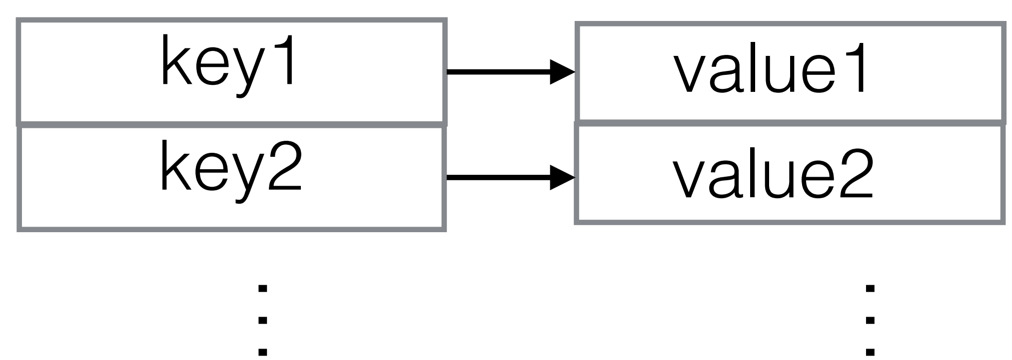


How do we implement map operations efficiently?

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);
  }
}
```