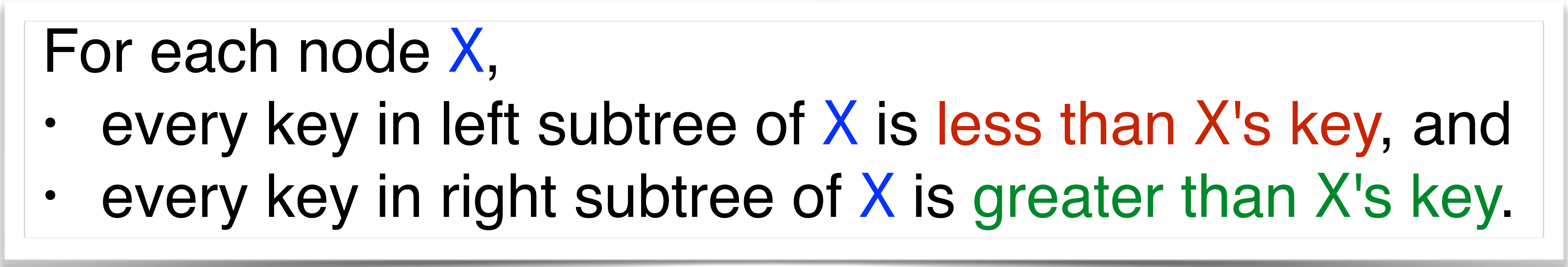
Binary Search Trees

Binary Search Tree Property



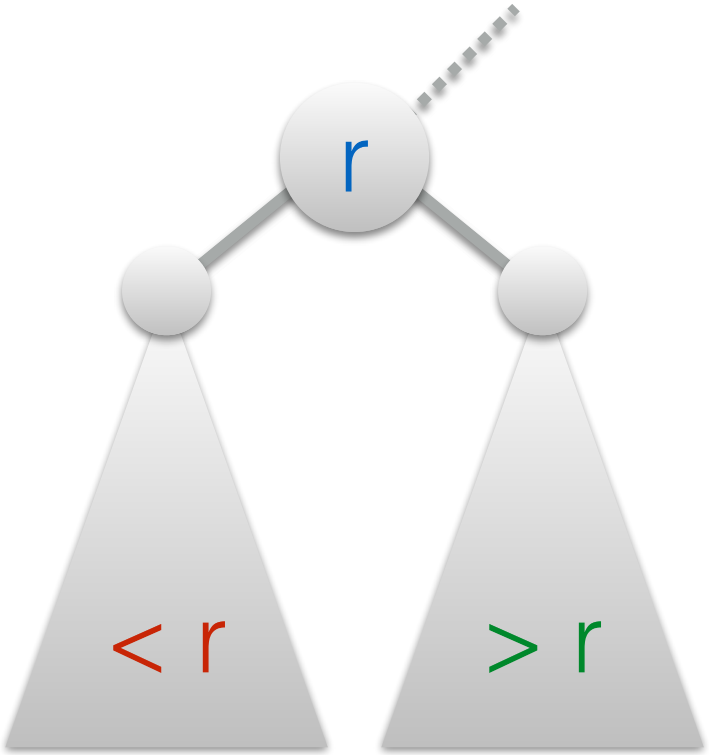
For each node X,

- every key in left subtree of X is less than X's key, and
- every key in right subtree of X is greater than X's key.

Binary Search Tree Property

For each node X,

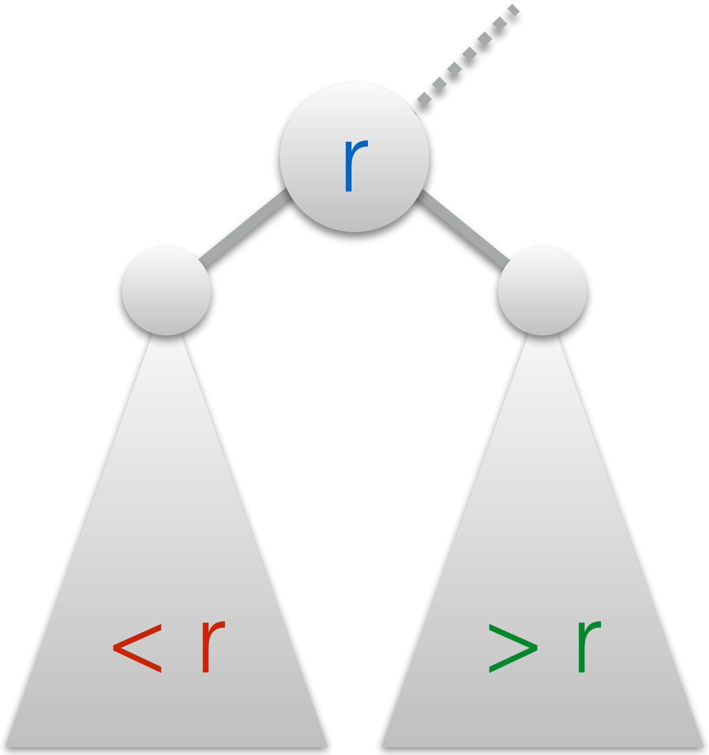
- every key in left subtree of X is less than X's key, and
- every key in right subtree of X is greater than X's key.

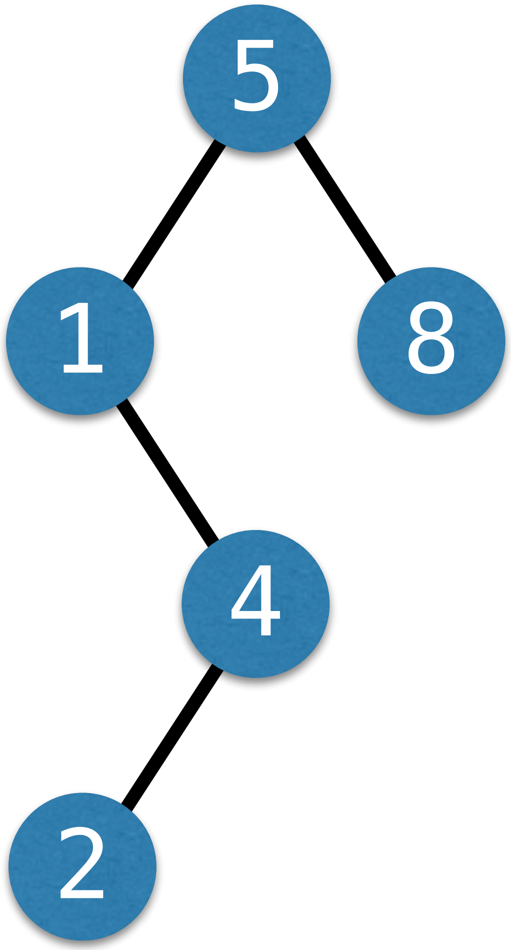


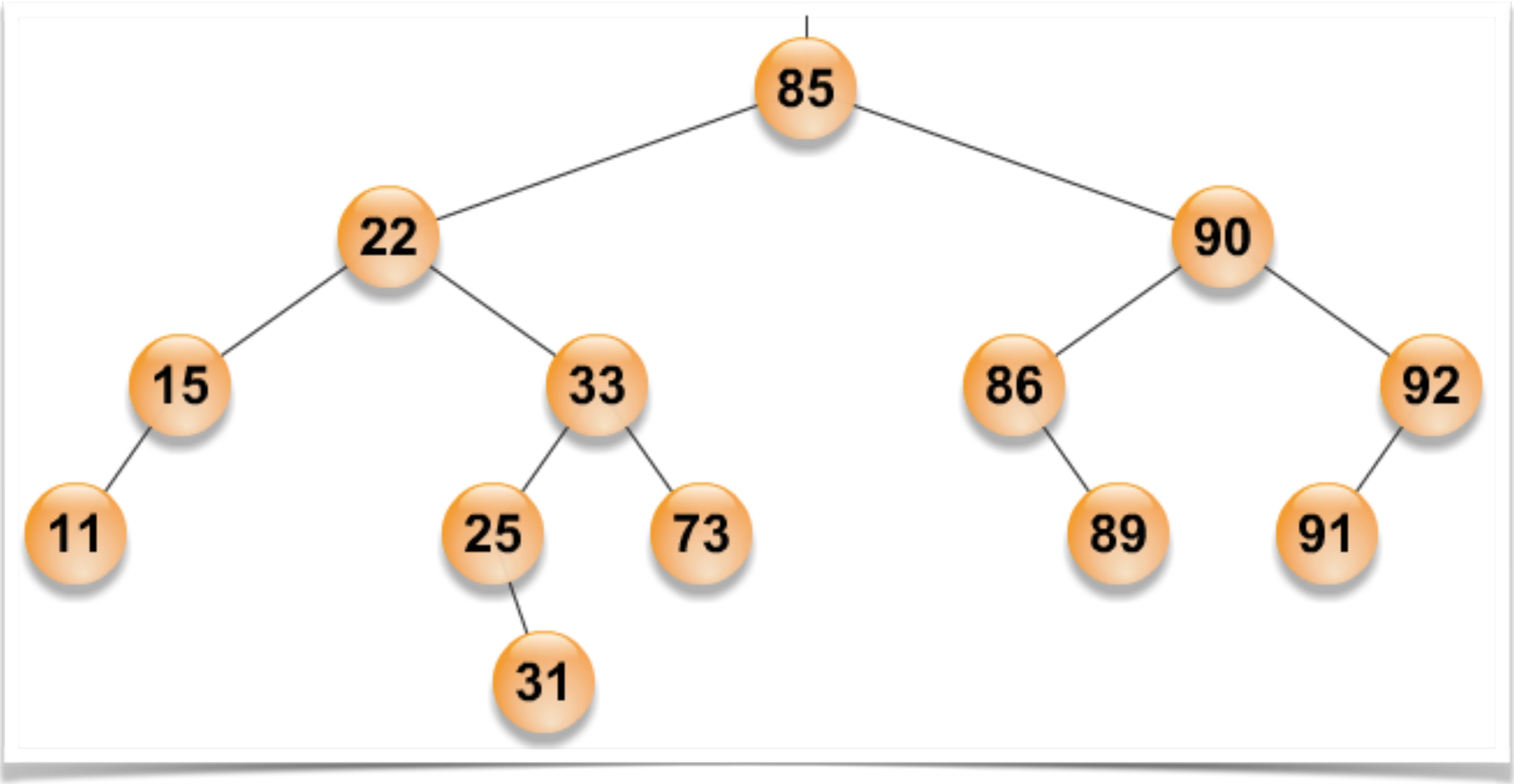
Binary Search Tree Property

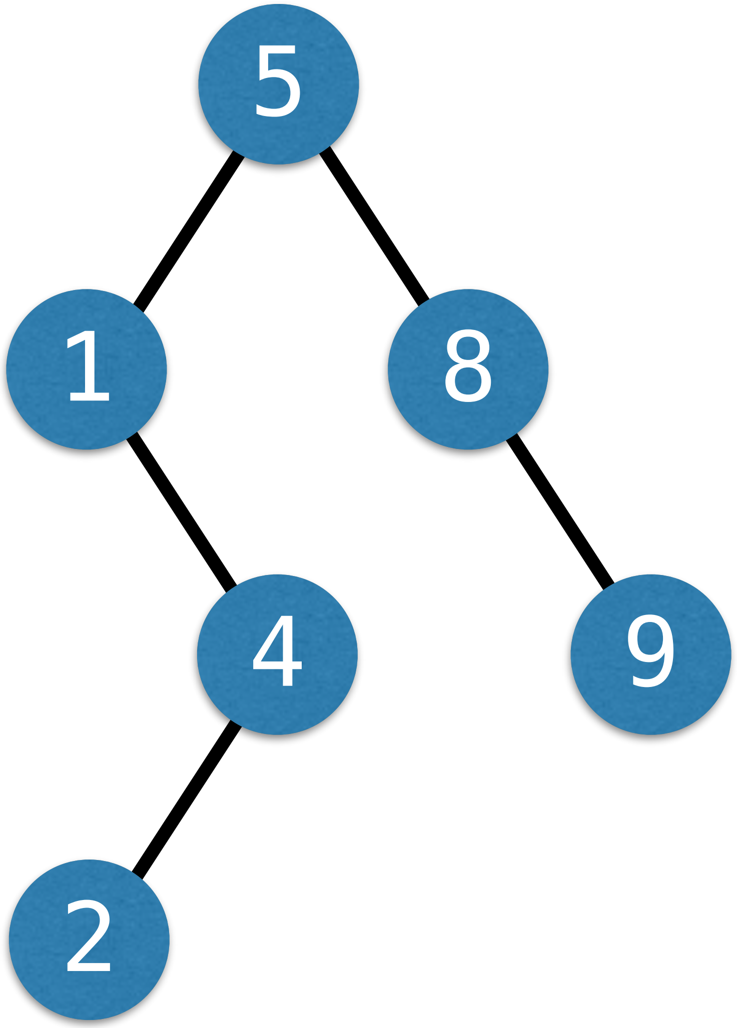
For each node X,

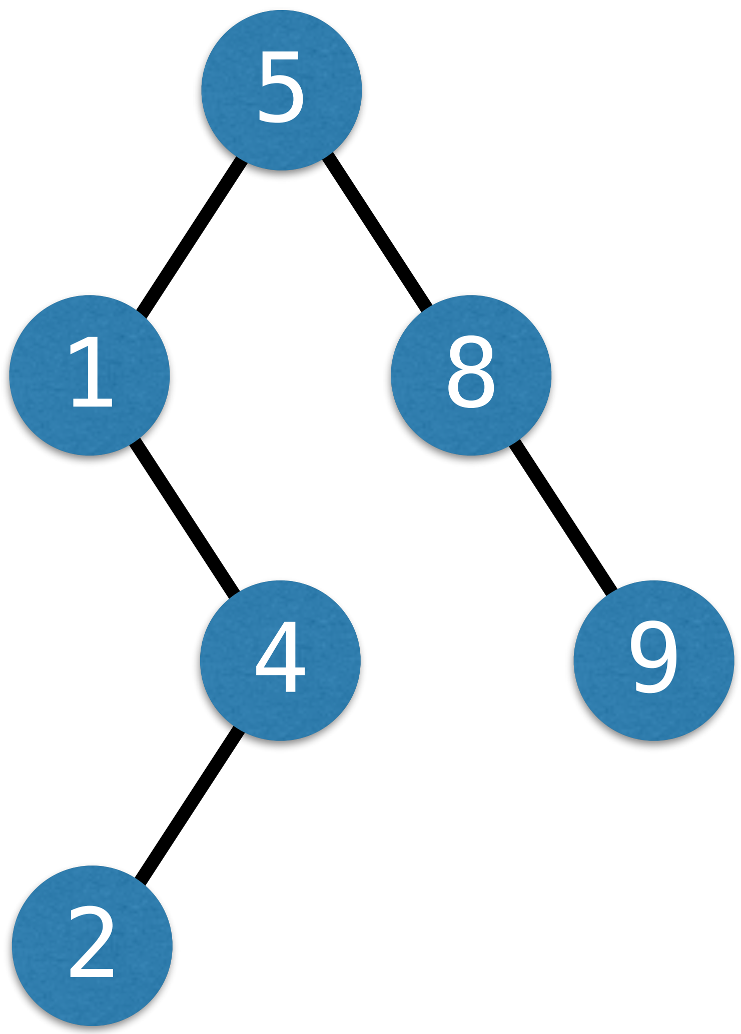
- every key in left subtree of X is less than X's key, and
- every key in right subtree of X is greater than X's key.

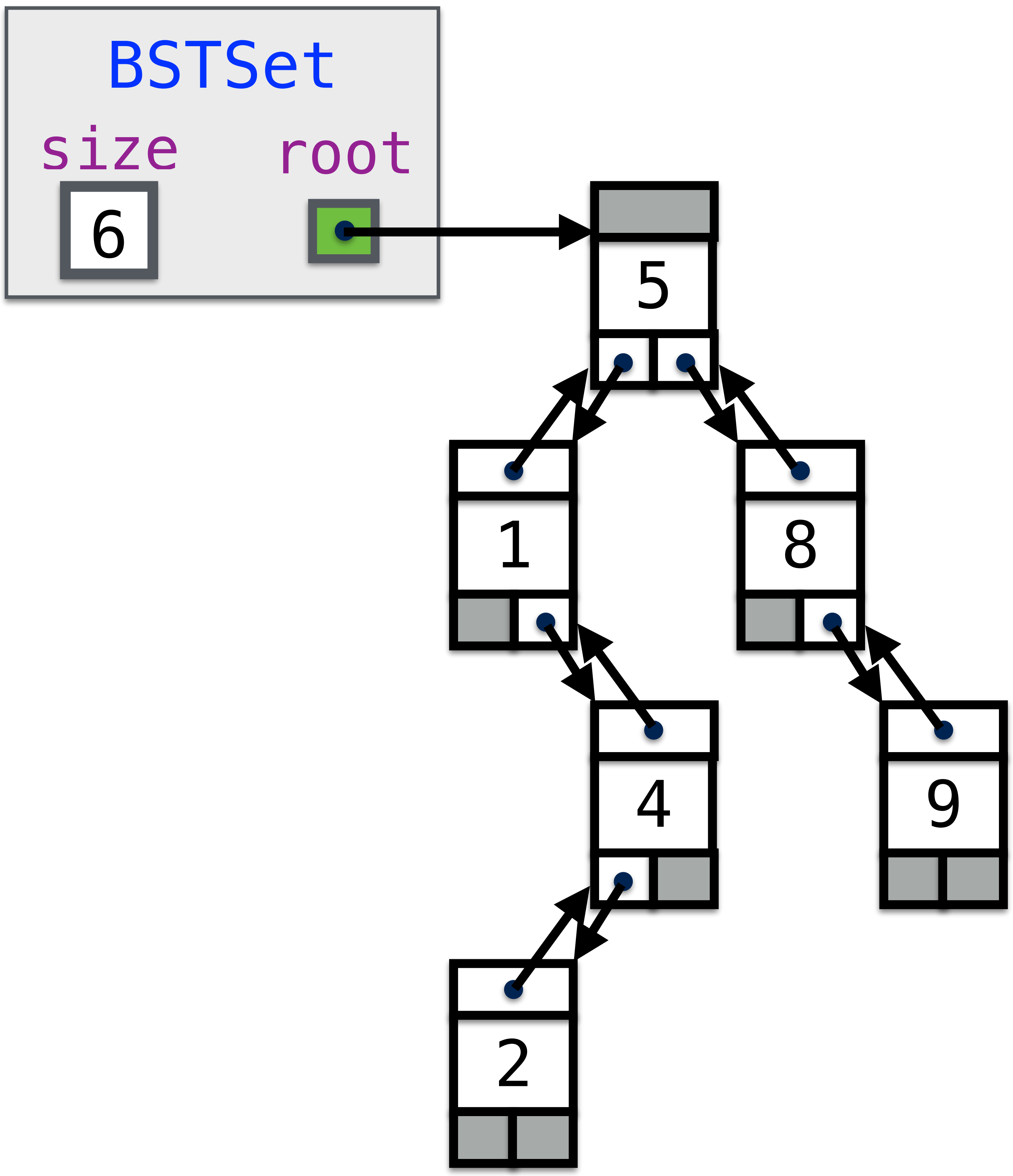


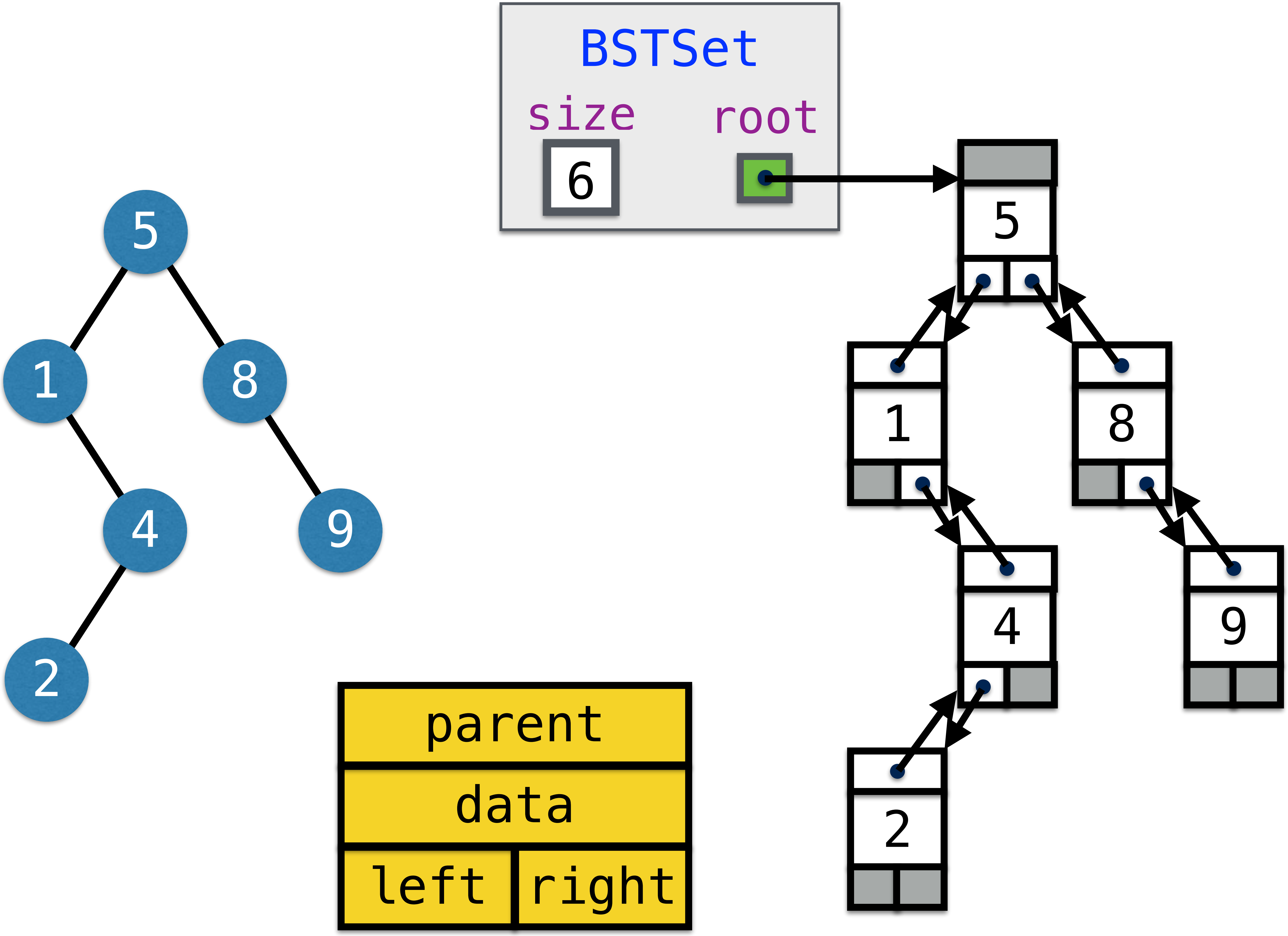


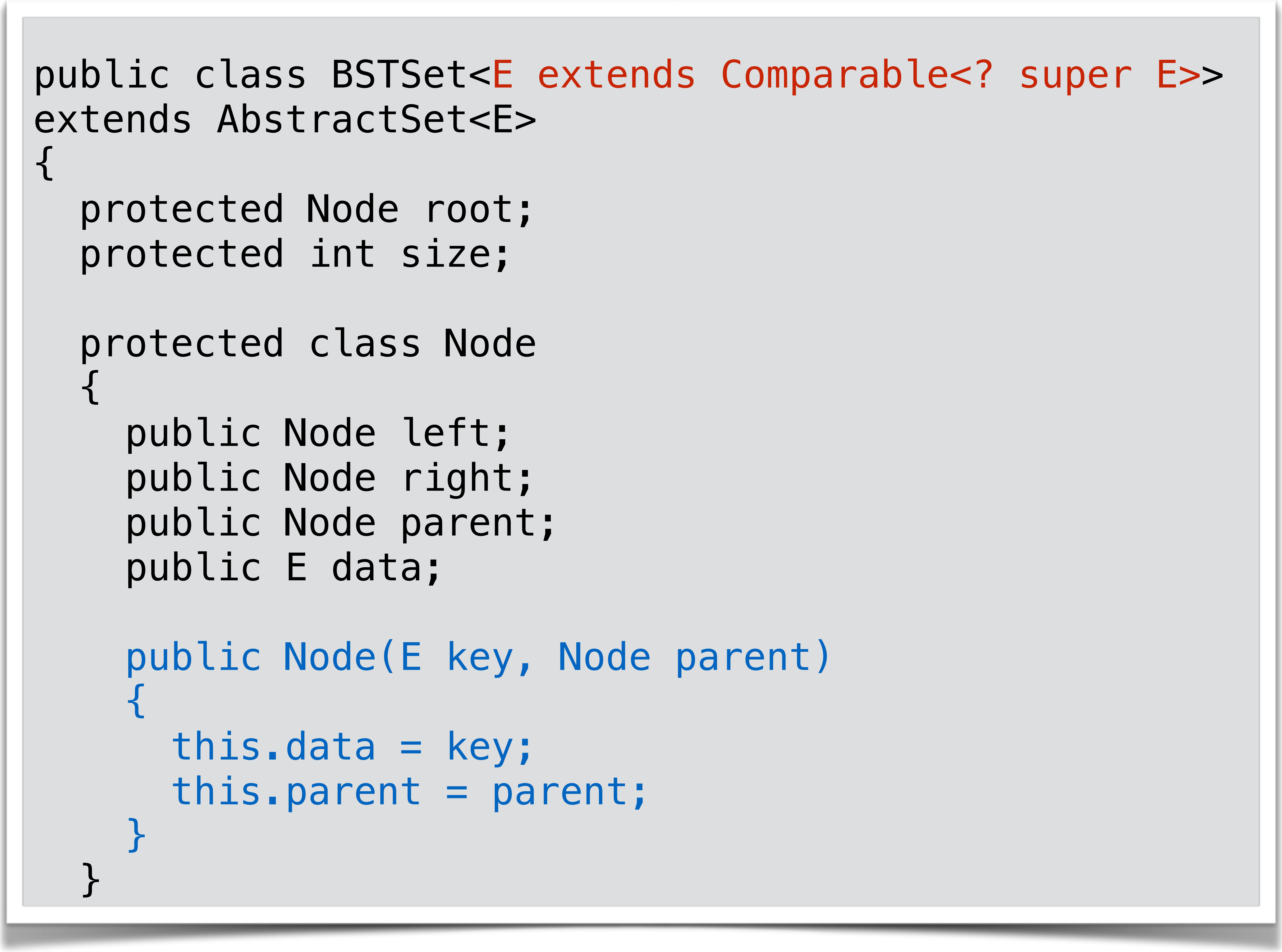

85
22
90
15
33
86
92
11
25
73
89
91
31


5
1
8
4
9
2


5
1
8
4
9
2


BSTSet
size
root
6
5
1
8
4
9
2


5
1
8
4
9
2
BSTSet
size
root
6
5
1
8
4
9
2
parent
data
left
right



```
public class BSTSet<E extends Comparable<? super E>>
extends AbstractSet<E>
{
  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;
    public Node right;
    public Node parent;
    public E data;

    public Node(E key, Node parent)
    {
      this.data = key;
      this.parent = parent;
    }
  }
```

```
public class BSTSet<E extends Comparable<? super E>>
extends AbstractSet<E>
{
  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;
    public Node right;
    public Node parent;
    public E data;

    public Node(E key, Node parent)
    {
      this.data = key;
      this.parent = parent;
    }
  }
```

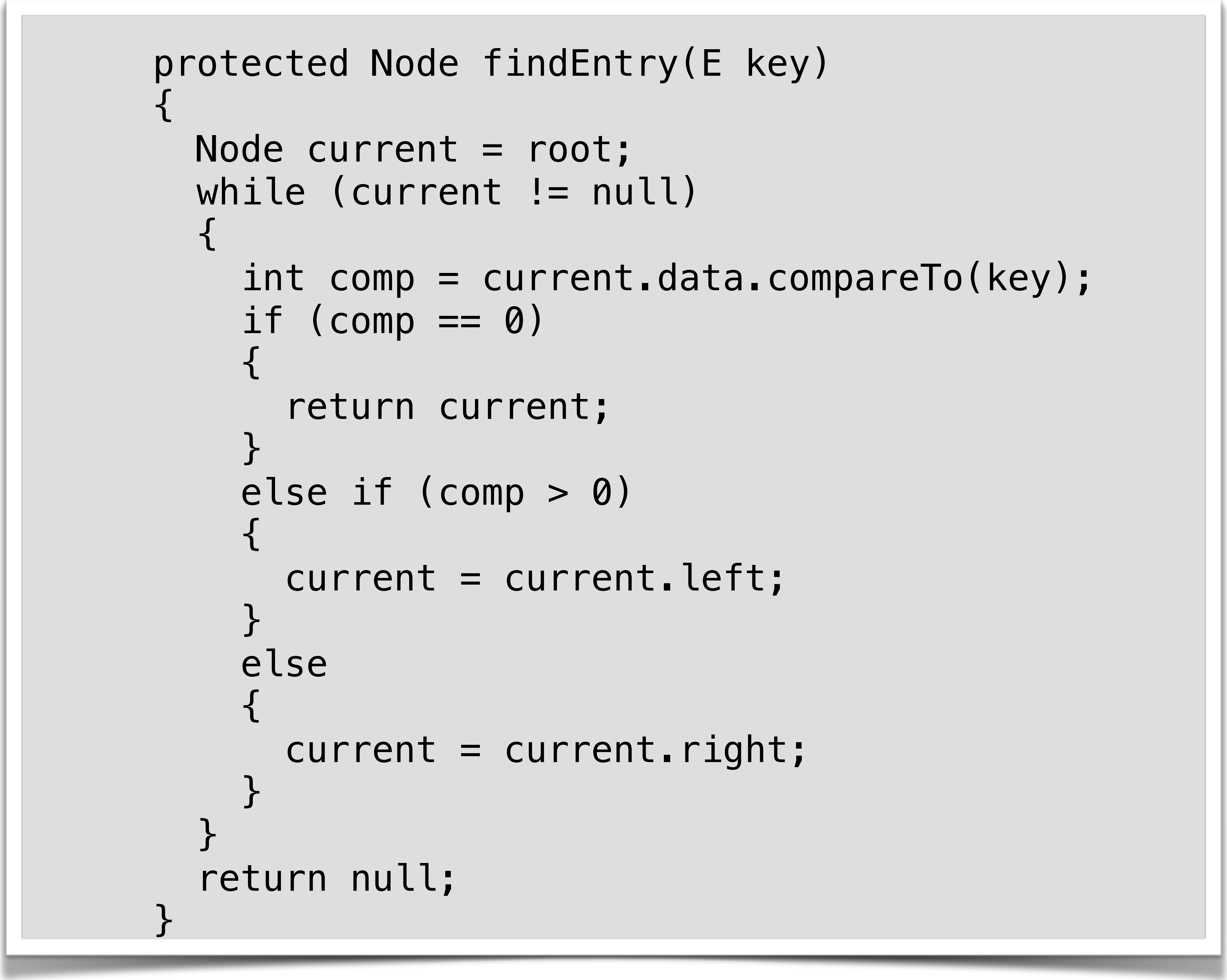
Elements have a natural ordering.

```
public class BSTSet<E extends Comparable<? super E>>
extends AbstractSet<E>
{
  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;
    public Node right;
    public Node parent;
    public E data;

    public Node(E key, Node parent)
    {
      this.data = key;
      this.parent = parent;
    }
  }
```

Elements have a natural ordering.

Sets `data` and `parent` fields

Searching



```
protected Node findEntry(E key)
{
  Node current = root;
  while (current != null)
  {
    int comp = current.data.compareTo(key);
    if (comp == 0)
    {
      return current;
    }
    else if (comp > 0)
    {
      current = current.left;
    }
    else
    {
      current = current.right;
    }
  }
  return null;
}
```

Insertion

```
public boolean add(E key)
{
  if (root == null)
  {
    root = new Node(key, null);
    ++size;
    return true;
  }

  Node current = root;

  while (true)
  {
    int comp = current.data.compareTo(key);

    if (comp == 0)
    {
      return false;
    }
```

```
public boolean add(E key)
{
   if (root == null)
   {
      root = new Node(key, null);
      ++size;
      return true;
   }

   Node current = root;

   while (true)
   {
      int comp = current.data.compareTo(key);

      if (comp == 0)
      {
         return false;
      }
```

key is in tree: do not add it

key precedes current.data: go left.

```
else if (comp > 0)
{
  if (current.left != null)
  {
    current = current.left;
  }
  else
  {
    current.left = new Node(key, current);
    ++size;
    return true;
  }
}
```

`key` precedes `current.data`: go left.

```
else if (comp > 0)
{
   if (current.left != null)
   {
      current = current.left;
   }
   else
   {
      current.left = new Node(key, current);
      ++size;
      return true;
   }
}
```

Left subtree non-empty: keep going.

key precedes current.data: go left.

```
else if (comp > 0)
{
   if (current.left != null)
   {
      current = current.left;
   }
   else
   {
      current.left = new Node(key, current);
      ++size;
      return true;
   }
}
```

Left subtree non-empty: keep going.

key not found: insert it.

`key` succeeds `current.data`:
go right.

```
    else
    {
      if (current.right != null)
      {
        current = current.right;
      }
      else
      {
        current.right = new Node(key, current);
        ++size;
        return true;
      }
    }
  }
}
```

`key` succeeds `current.data`:
go right.

```
    else
    {
      if (current.right != null)
      {
        current = current.right;
      }
      else
      {
        current.right = new Node(key, current);
        ++size;
        return true;
      }
    }
  }
}
```

Right subtree non-empty: keep going.

key succeeds current.data: go right.

```
    else
    {
      if (current.right != null)
      {
        current = current.right;
      }
      else
      {
        current.right = new Node(key, current);
        ++size;
        return true;
      }
    }
  }
}
```

Right subtree non-empty: keep going.

key not found: insert it.