CS 61BL Lab 10

Ryan Purpura

Announcements

- Please re-sign up for a Design Document Review Meeting for Monday (link on Piazza)
- Design Documents due end of the day tomorrow

Finding an element

Finding an element

- Given an array of items, what is the worst-case runtime of checking if an item exists in that array?

Finding an element

- Given an array of items, what is the worst-case runtime of checking if an item exists in that array?
- Answer: $\Theta(N)$

Finding an element

- Given an array of items, what is the worst-case runtime of checking if an item exists in that array?
- Answer: $\Theta(N)$
- Can we do better?

Binary Search

- If the array is sorted, is there a better algorithm than just iterating through each element?

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Binary Search Algorithm

- Check the center element.
  - If our search key is greater than the center element, we can completely eliminate the left side from our search
  - if our search key is less than the center element, we can completely eliminate the right side from our search
- Perform this process recursively on the resulting sublist!

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length - 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid - 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Search key: 84

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length − 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid − 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Search key: 84

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length - 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid - 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Search key: 84

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length - 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid - 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Search key: 84

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length - 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid - 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

Search key: 84

Pseudocode

```
function binarySearch(array, key):
    return binarySearch(array, key, 0, array.length - 1)

function binarySearch(array, key, low, high):
    if low > high: return false
    mid = (high + low) / 2
    if key < array[mid]:
        return binarySearch(array, key, low, mid - 1)
    else if key > array[mid]:
        return binarySearch(array, key, mid + 1, high)
    else:
        return true
```

1	3	6	10	13	15	18	22	41	84	100
0	1	2	3	4	5	6	7	8	9	10

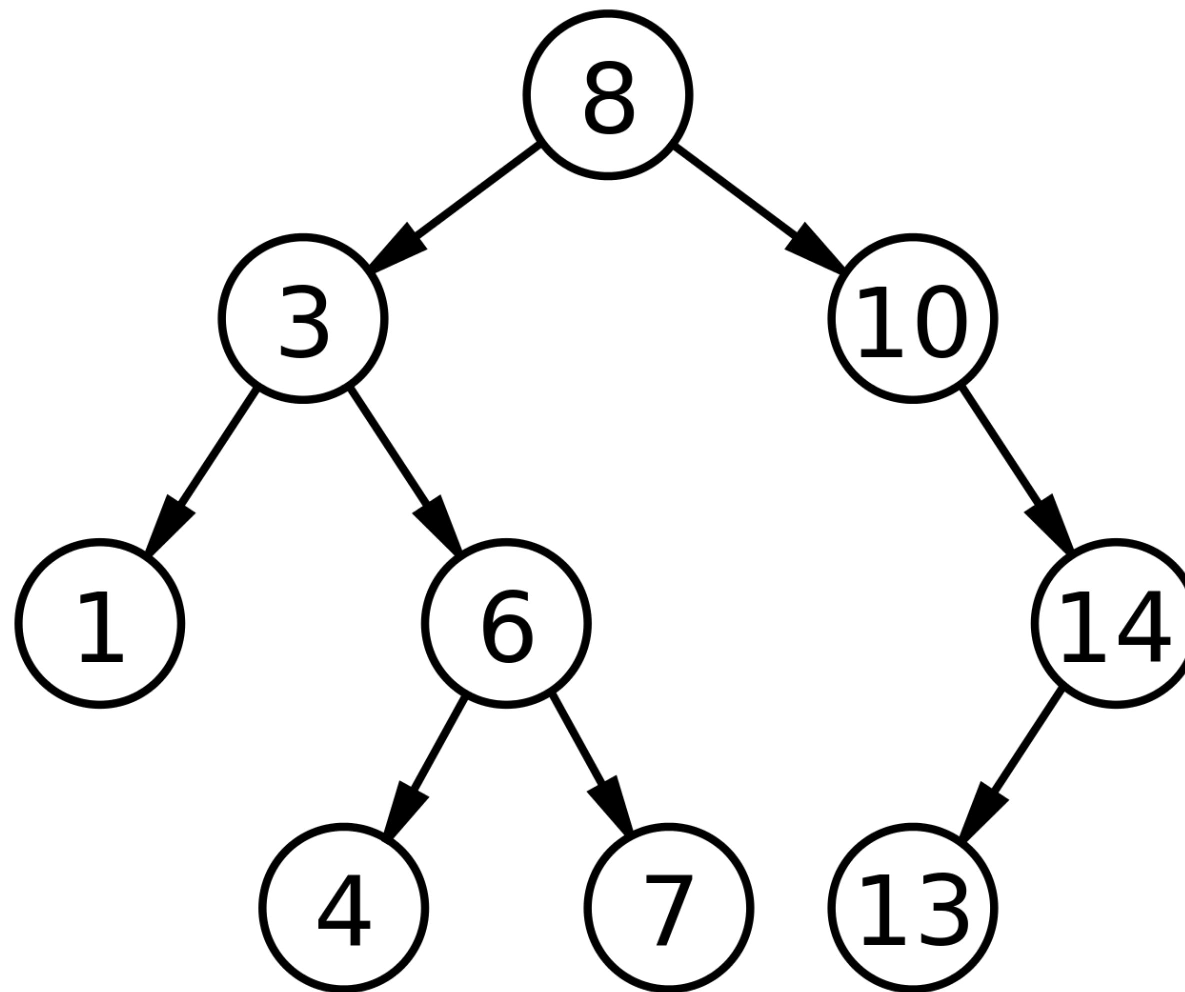
Search key: 84

Runtime

- We keep dividing the array in half, so we visit at most $\log N$ elements, so the runtime is $O(\log N)$

Binary Search Trees

- Let's bring the power of binary search to the tree!
- Idea: For each node, every element in its left subtree must be less than it, and every element in its right subtree must be greater than it

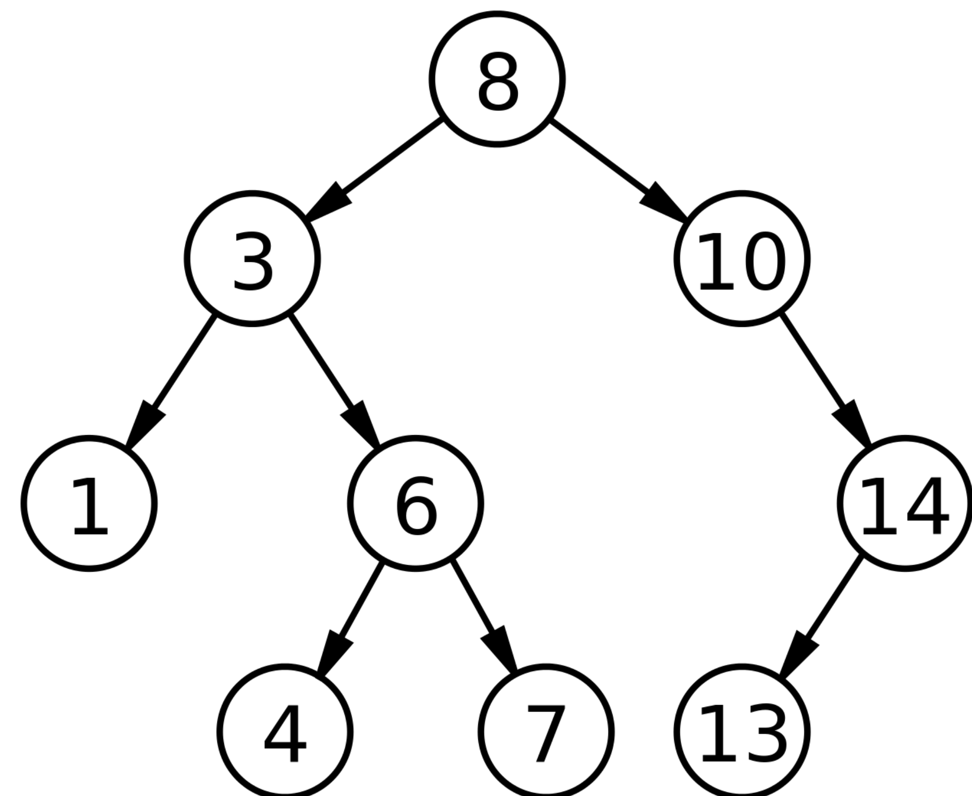


Binary Tree Search

- How to check if an element exists in a binary search tree?

1. If root equals the search key, return true
2. If tree is a leaf, return false
3. If search key is greater than root, go back to step 1 with left subtree
4. Else if search key is less than root, go back to step 1 with right subtree

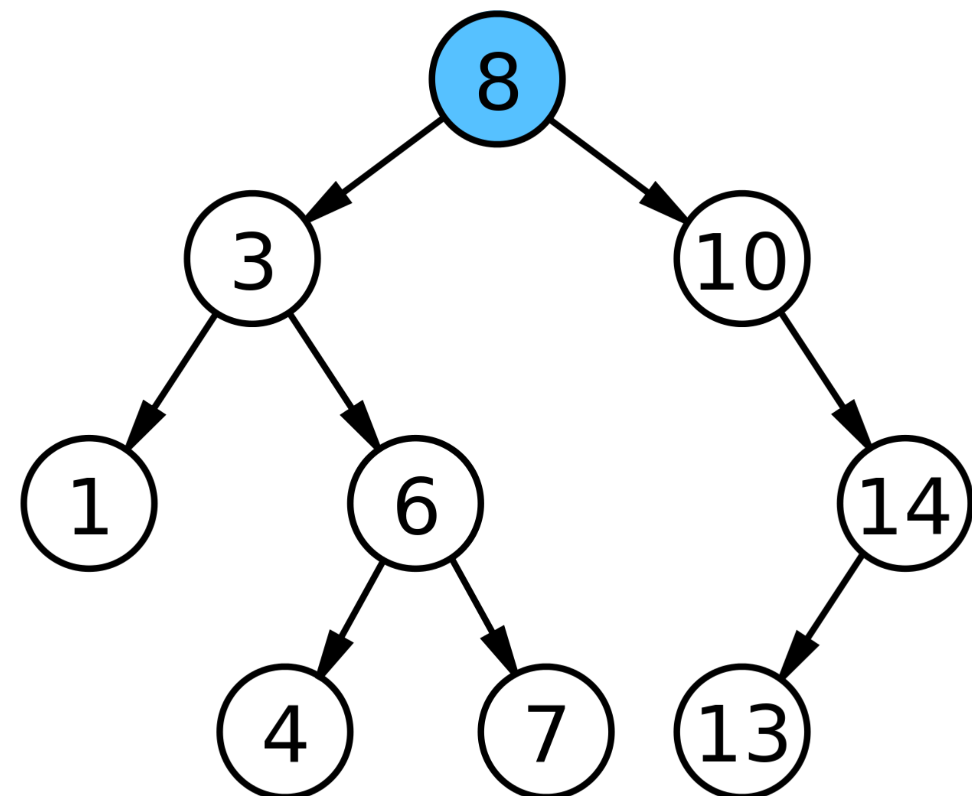
Searching for 4:



Binary Tree Search

- How to check if an element exists in a binary search tree?

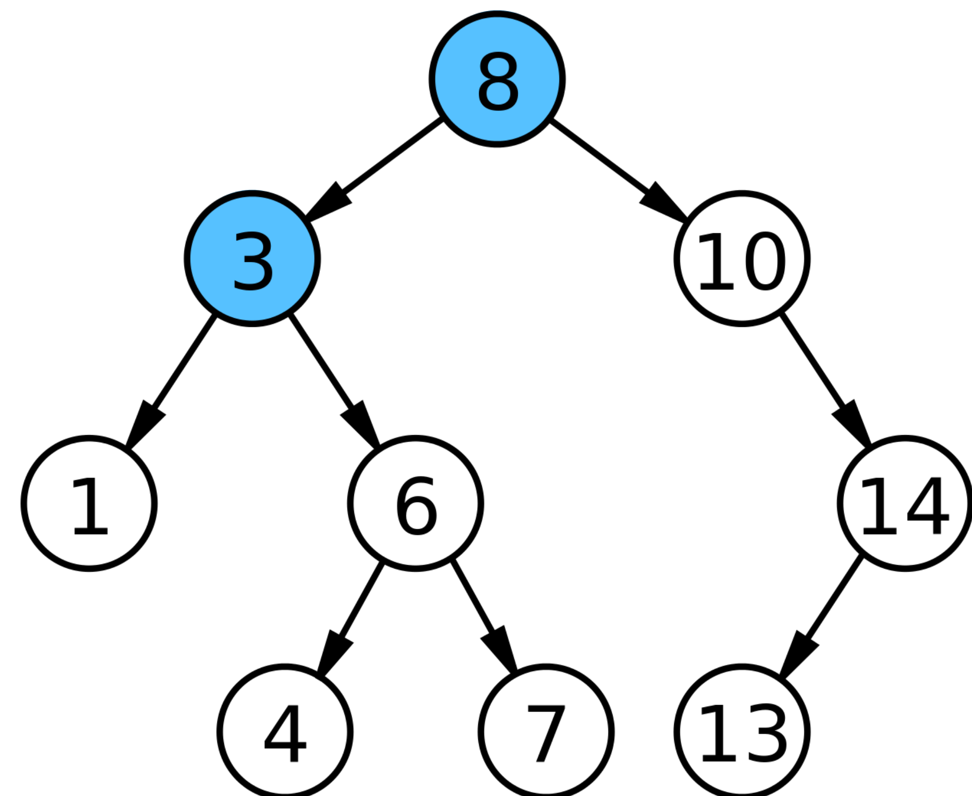
1. If root equals the search key, return true
2. If tree is a leaf, return false
3. If search key is greater than root, go back to step 1 with left subtree
4. Else if search key is less than root, go back to step 1 with right subtree



Binary Tree Search

- How to check if an element exists in a binary search tree?

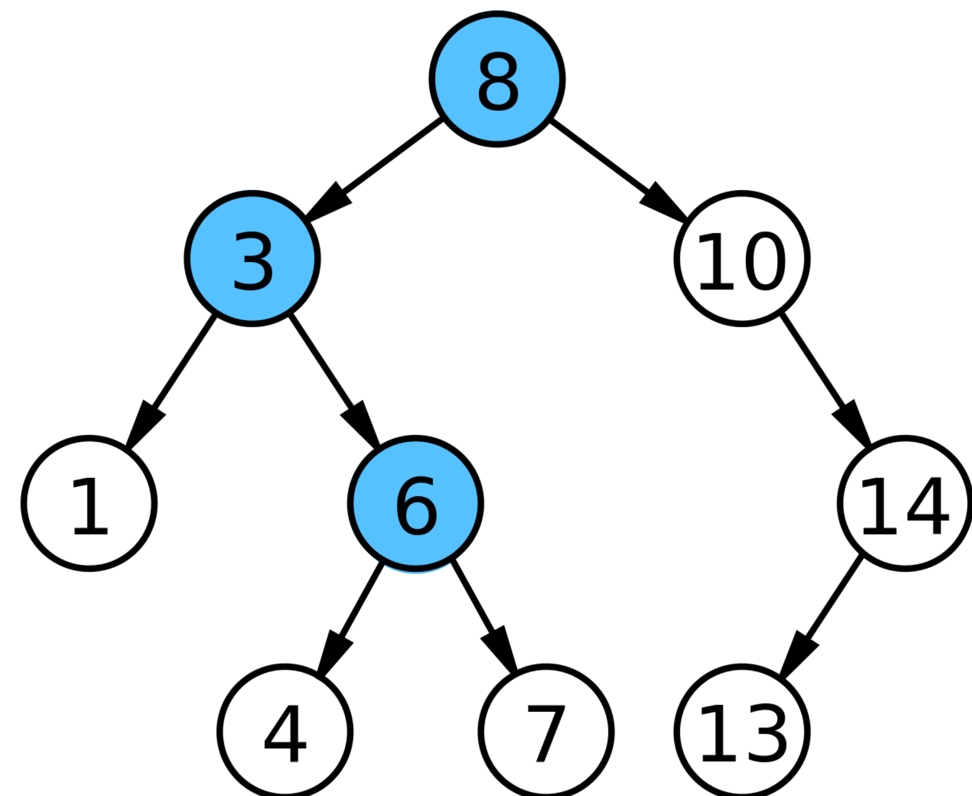
1. If root equals the search key, return true
2. If tree is a leaf, return false
3. If search key is greater than root, go back to step 1 with left subtree
4. Else if search key is less than root, go back to step 1 with right subtree



Binary Tree Search

- How to check if an element exists in a binary search tree?

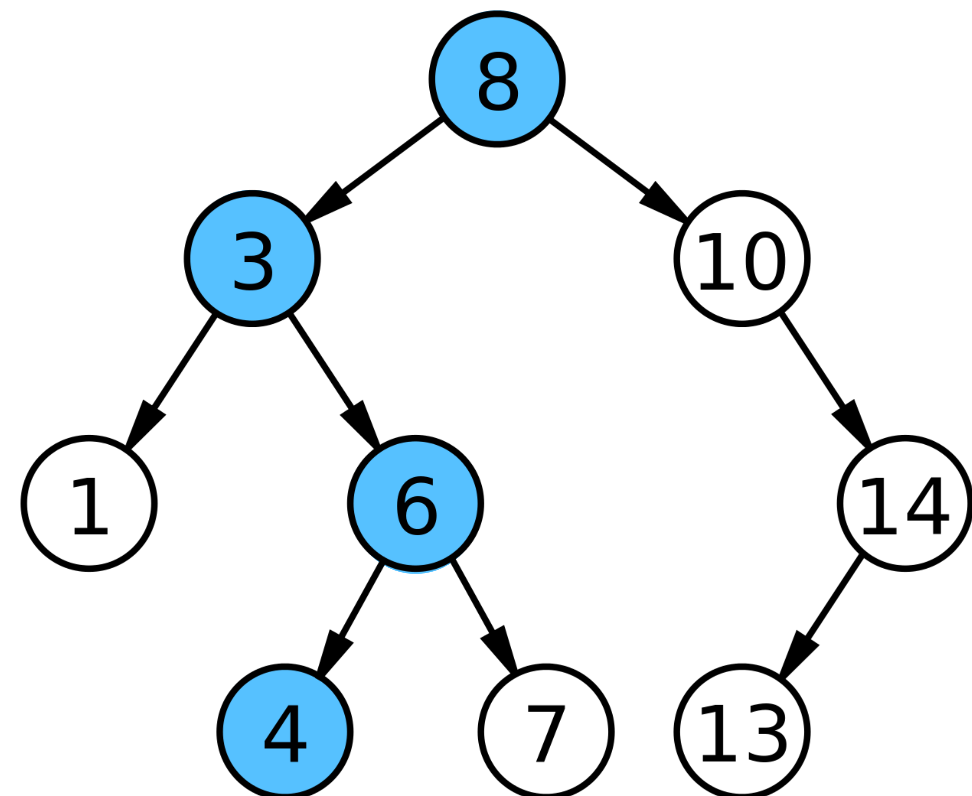
1. If root equals the search key, return true
2. If tree is a leaf, return false
3. If search key is greater than root, go back to step 1 with left subtree
4. Else if search key is less than root, go back to step 1 with right subtree



Binary Tree Search

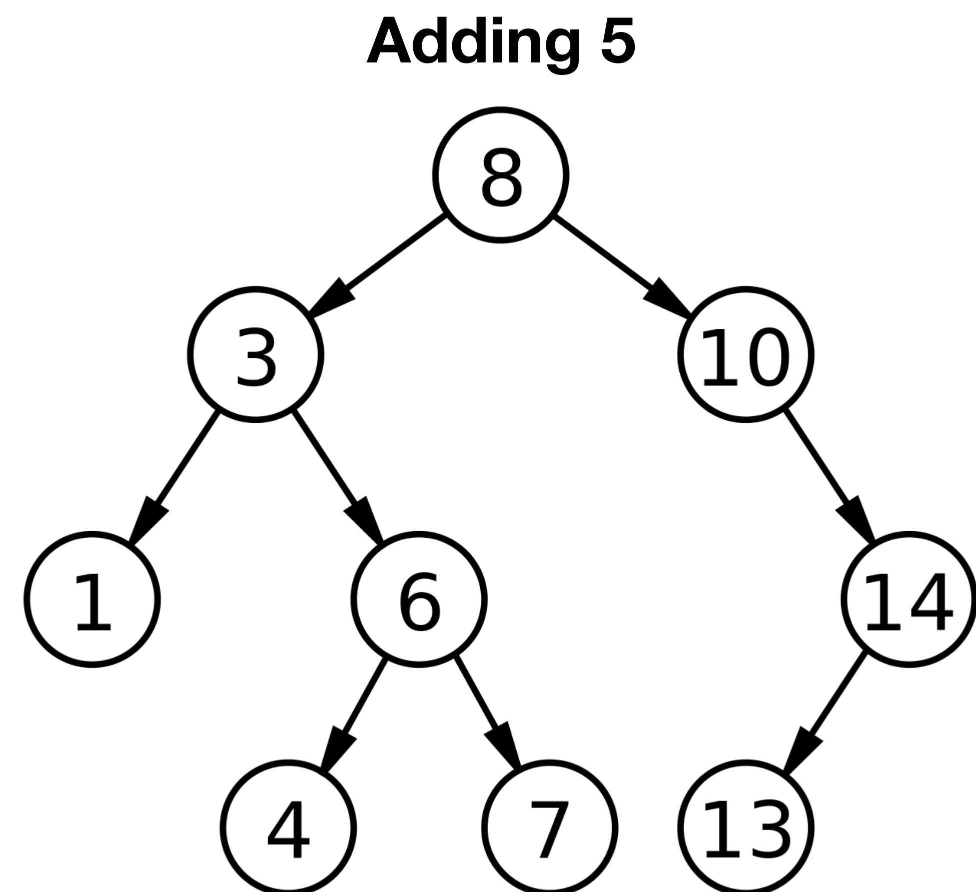
- How to check if an element exists in a binary search tree?

1. If root equals the search key, return true
2. If tree is a leaf, return false
3. If search key is greater than root, go back to step 1 with left subtree
4. Else if search key is less than root, go back to step 1 with right subtree



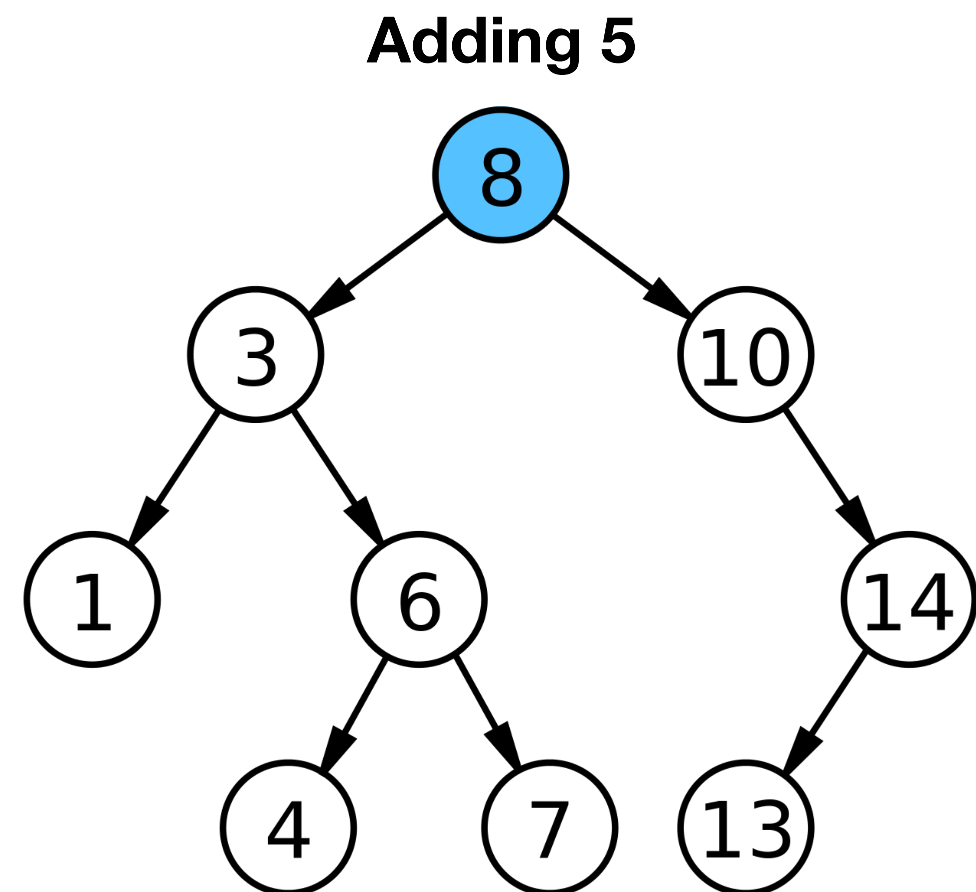
Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



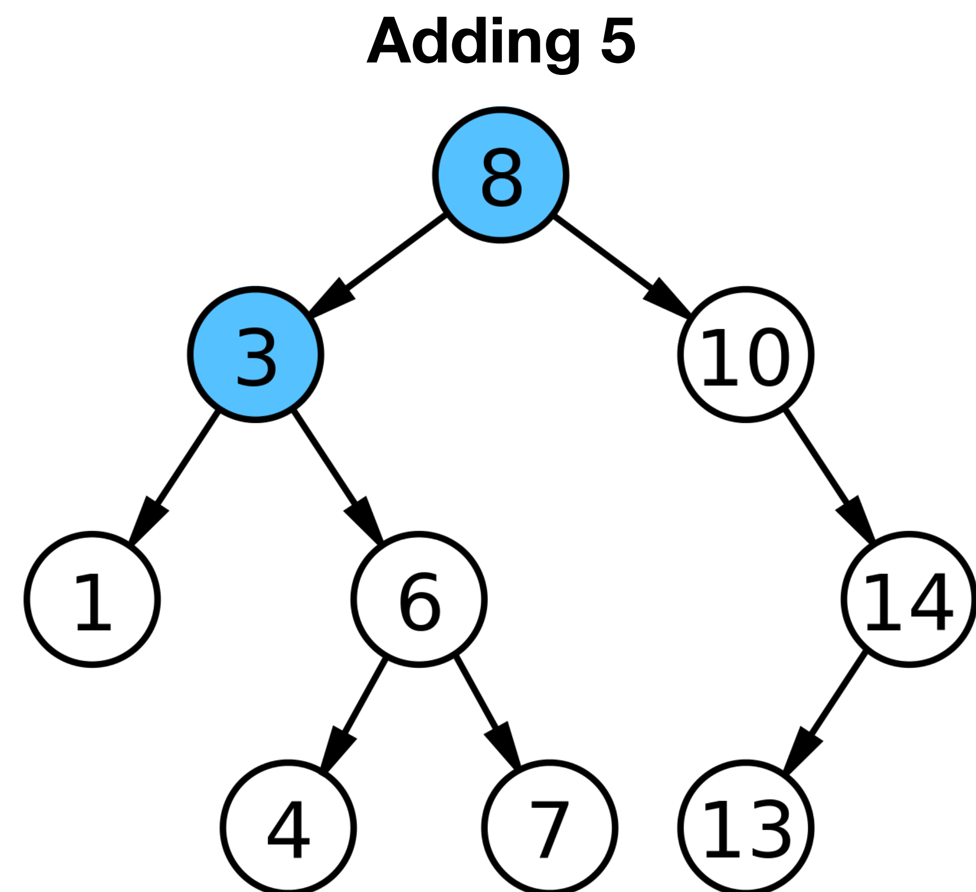
Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



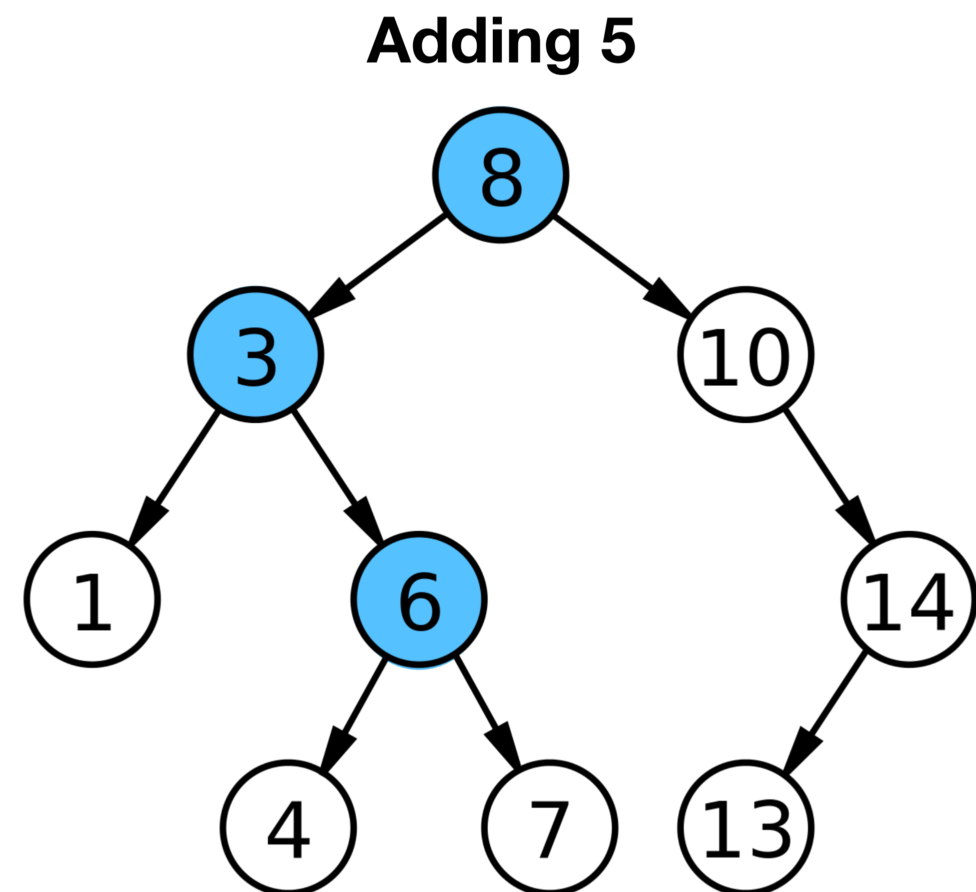
Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



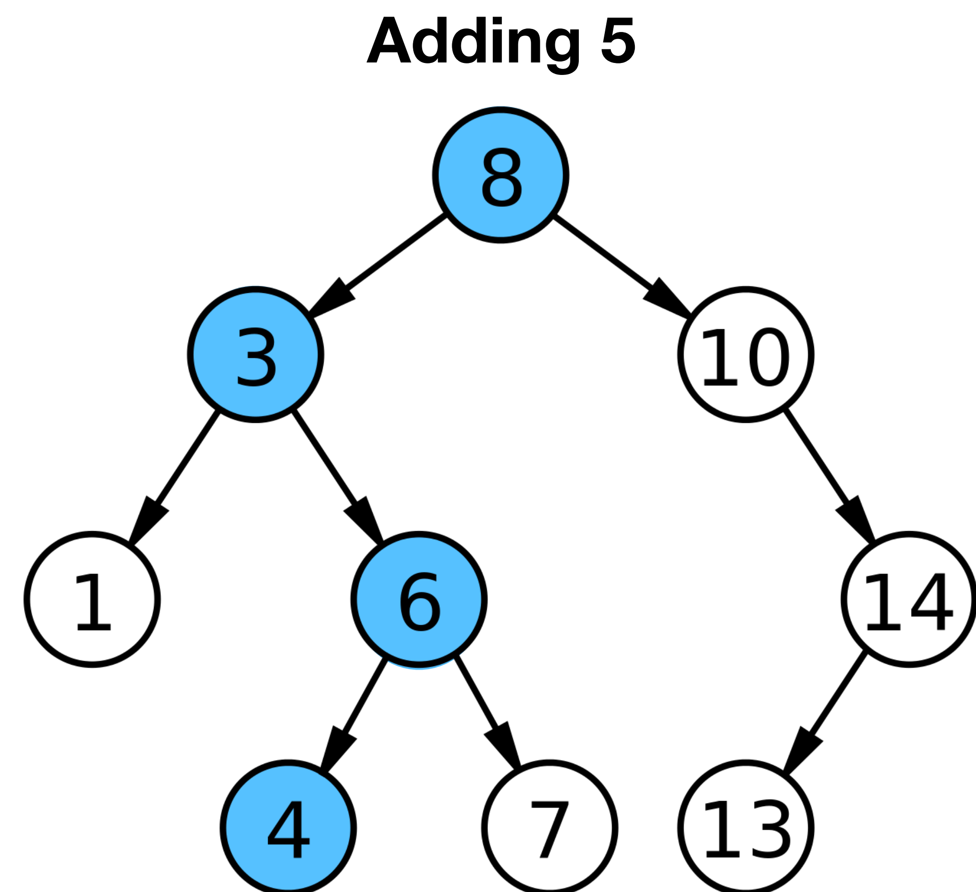
Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



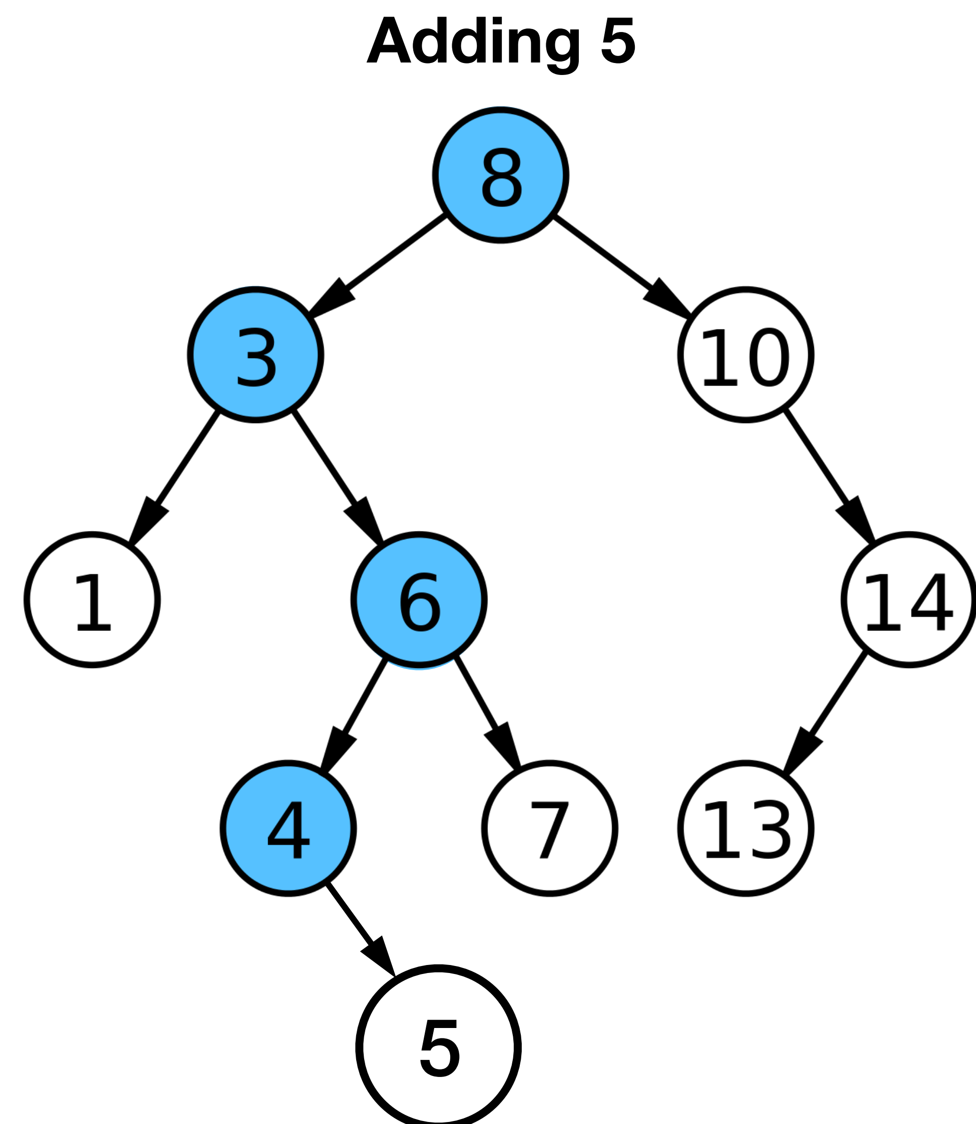
Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



Adding Elements to a Binary Search Tree

- Search for the element you want to add. If it doesn't exist (i.e. when we reach a leaf), promote that leaf to a parent by adding a child node on the appropriate side.



Comparing Objects

- In Java, it's easy to compare numbers using >, <, <=, >=
- But we often want to compare objects, too, and Java doesn't let you use those operators
  - This would let us sort an array of objects, use it in a binary search tree, etc.
- What to do?

Comparable Interface

- The Comparable interface allows us to define a *natural ordering* of objects.

Interface Comparable<T>

Type Parameters:

`T - the type of objects that this object may be compared to`

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Usage

- Let's say we have a Person class:

```
public class Person
{
    String name;
    int id;

    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }







}
```

Usage

- Let's say we have a Person class:

```
public class Person implements Comparable<Person>
{
    String name;
    int id;

    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }








}
```

Usage

- Let's say we have a Person class:

```
public class Person implements Comparable<Person>
{
    String name;
    int id;

    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }

    @Override
    public int compareTo(Person other) {
        return this.id - other.id;
    }


}
```

Usage

- Let's say we have a Person class:

```
public class Person implements Comparable<Person>
{
    String name;
    int id;

    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }

    @Override
    public int compareTo(Person other) {
        return this.id - other.id;
    }


}
```

Now if you have a `Person[]` array, you can sort it with `Arrays.sort(arr)`
Or if you have a `List<Person>` list you can sort it with `Collections.sort(lst)`

Comparators

- However, sometimes you want to order objects in a different way than what compareTo does.
- You can define another class that implements Comparator.

Interface Comparator<T>

Type Parameters:

T - the type of objects that may be compared by this comparator

compare

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```
class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.name.compareTo(o2.name);
    }
}
```

Sorting by Comparators

- Very often, methods that expect a class to implement Comparable will offer an alternative that will take a Comparator instead
- This allows you to sort things in different ways!

`sort(List<T> list)`

Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

`sort(List<T> list, Comparator<? super T> c)`

Sorts the specified list according to the order induced by the specified comparator.