Binary Search Trees

Binary Search Trees

Definition

Binary Search Trees are used to store items in memory.

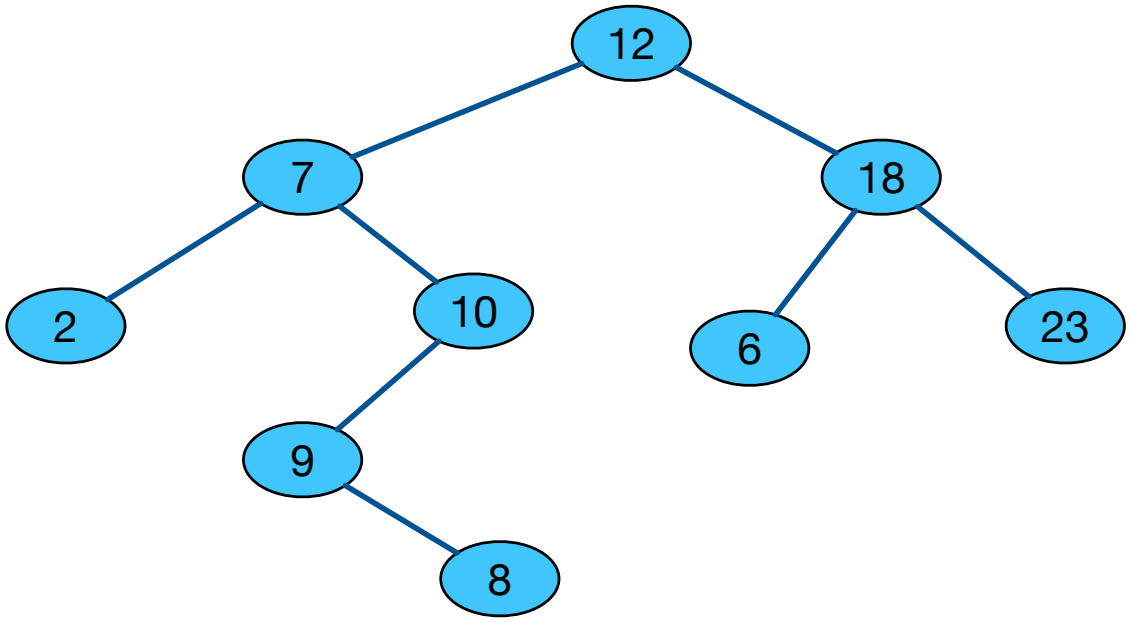
- They allow for fast lookup, insertion, and deletion.
- They keep their keys in sorted order so that lookup and other operations can use the principle of binary search.

Binary Search Trees

- A binary search tree is a binary tree (each vertex, v, has at most two children left, $\ell(v)$, and right, $r(v)$).
- Vertices are organized by the Binary Search Property:
  - Every vertex is ordered by a key.
  - For every vertex, v, the key of $\ell(v)$ (and keys of all vertices in the subtree rooted at $\ell(v)$) is less than the key of v.
  - For every vertex, v, the key of $r(v)$ (and keys of all vertices in the subtree rooted at $r(v)$) is greater than the key of v.

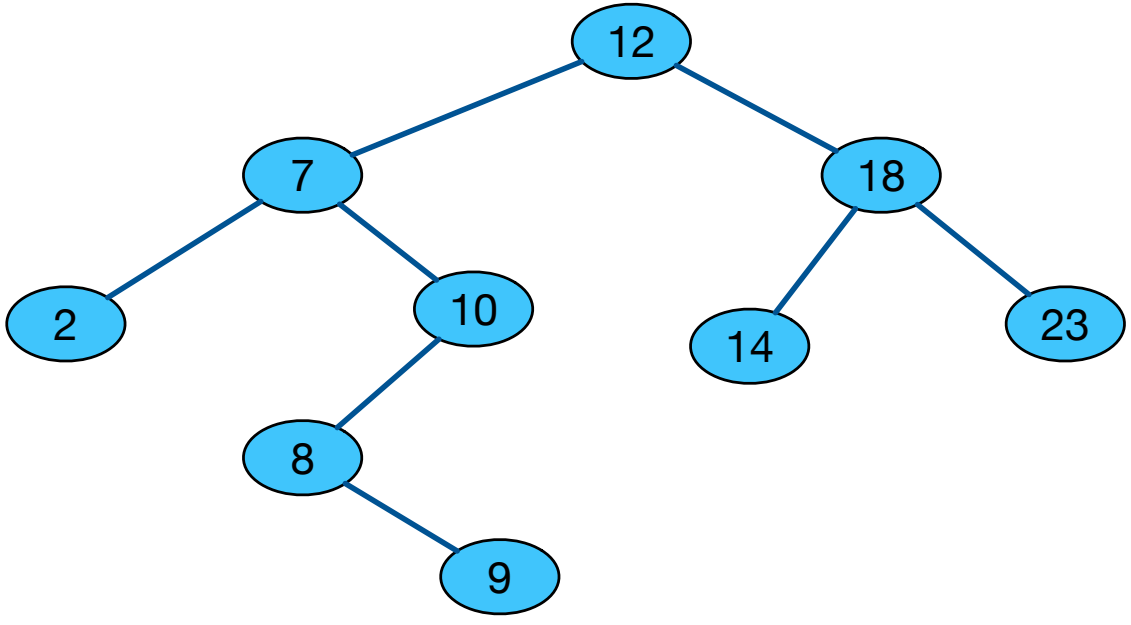
Binary Search Trees

Does the following graph represent a binary search tree?



Binary Search Trees

Does the following graph represent a binary search tree?



Operations on Binary Search Trees

There are three main operations:

- *find:* search the binary search tree for a specific key.
- *insert:* insert an element with a specific key (while maintaining the binary search tree structure).
- *delete:* remove an element (while maintaining the binary search tree structure).

There are additional helpful operations:

- *isEmpty:* check if the binary search tree is empty.
- *findMin:* finds the element with minimum key.
- *findMax:* finds the element with maximum key.

The *isEmpty* Operation

isEmpty(T)

Input: A binary search tree T.
Output: *True* if the tree is empty, *False* otherwise.

```
1: if root(T) = null then
       return True
2: else
       return False
```

The *Find* Operation

We can use a recursive procedure to find a vertex with a specific key, k, in a tree, T:

- Check the root, $root(T)$, if it's key ($key(root(T))$) is k, return the root.
- Otherwise:
  - If $key(root(T)) < k$: recurse on the right subtree.
  - If $key(root(T)) > k$: recurse on the left subtree.
  - There will be no additional case if we assume that all keys are distinct.

The *find* Operation

find(T, k, v)

Input: A binary search tree T, a key, k, and the current vertex, v (initially the root).

Output: Either a statement that there is no vertex with key k or the vertex with key k.

```
1: if isEmpty(T) then
       return "There is no vertex in T with key k."
2: if key(v) = k  then  return v
3: else if k < key(v) then
4:     if v.left = null then
         return "There is no vertex in T with key k."
5:     else  return find(T, k, v.left)
6: else
7:     if v.right = null then
         return "There is no vertex in T with key k."
8:     else  return find(T, k, v.right)
```

The *findMin* Operation

findMin(T, v)

Input: A binary search tree T and the current vertex, v (initially the root).

Output: The vertex with the smallest key.

```
1: if isEmpty(T) then
       return "The tree is empty"
2: else if v.left = null then
       return v
3: else
       return findMin(T, v.left)
```

The *findMax* Operation

findMax(T, v)

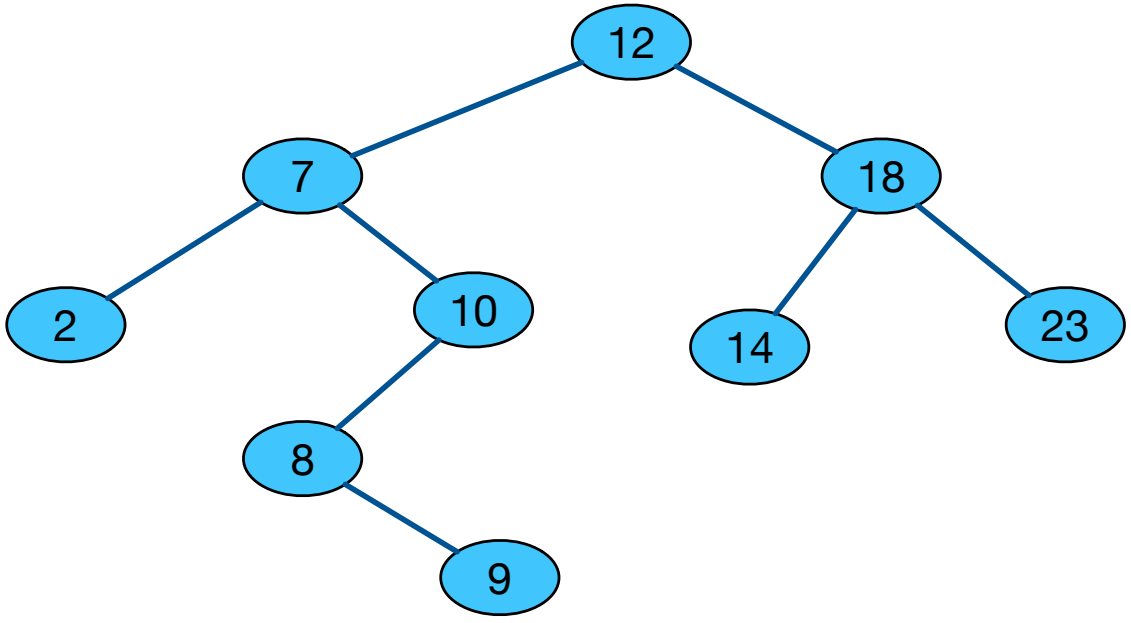
Input: A binary search tree T and the current vertex, v (initially the root).

Output: The vertex with the largest key.

```
1: if isEmpty(T) then
       return "The tree is empty"
2: else if v.right = null then
       return v
3: else
       return findMax(T, v.right)
```

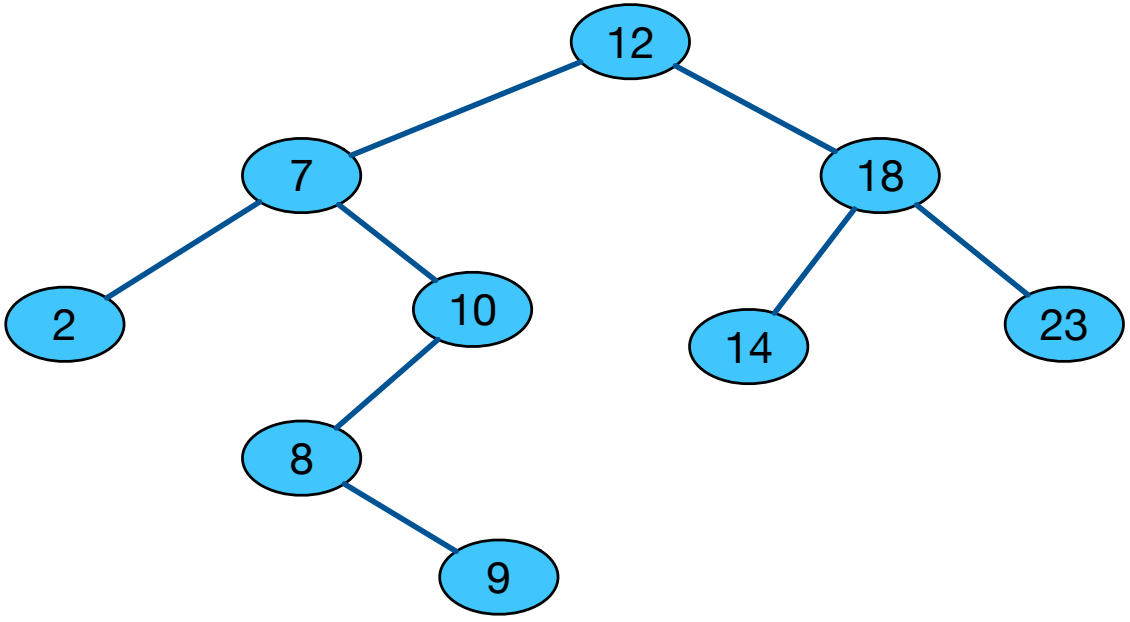
The *insert* Operation

How should we go about inserting a vertex with key 20 into the following binary search tree?



The *insert* Operation

How should we go about inserting a vertex with key 11 into the following binary search tree?



This operation expands on the *find* operation. We insert the new vertex as the appropriate child of the last vertex visited.

The *insert* Operation

insert(T, v, x)

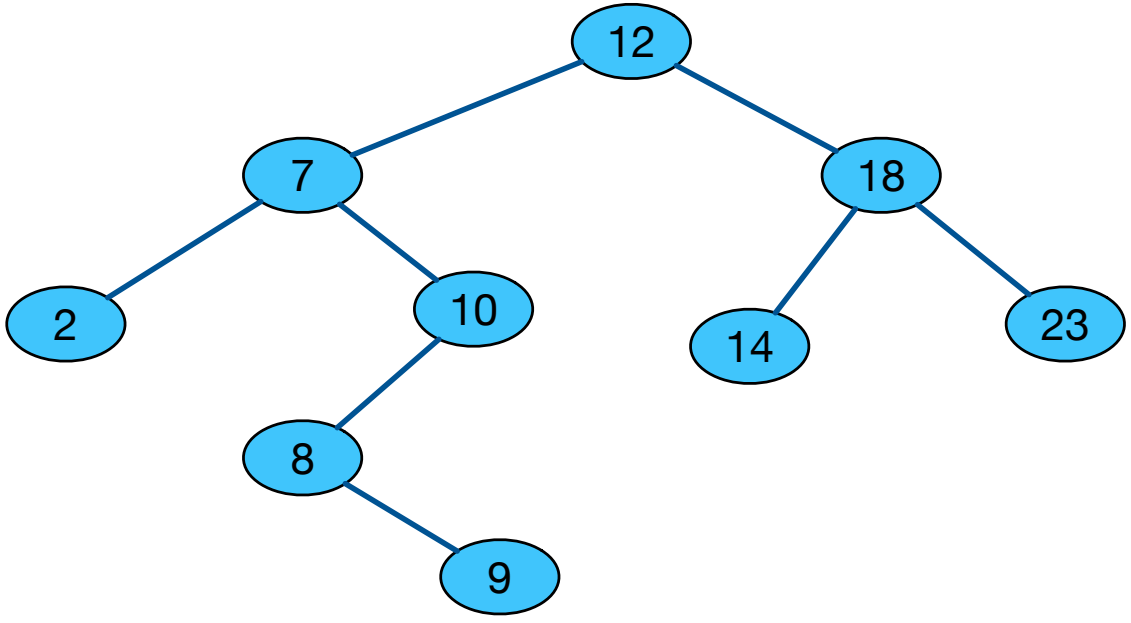
Input: A binary search tree T, the current vertex, v (initially the root), and the vertex to be inserted, x.

Output: The binary search tree, T, including x.

```
 1: if isEmpty(T) then
        return root(T) = x
 2: if key(v) = key(x) then
 3:     Update or discard....
 4: else if key(v) < key(x) then
 5:     if v.right = null then
 6:         v.right = x
 7:     else   return insert(T, v.right, x)
 8: else
 9:     if v.left = null then
10:         v.left = x
11:     else   return insert(T, v.left, x)
```

The *delete* Operation

How should we go about deleting vertex 18?



Vertex 10? Vertex 9?

The *delete* Operation

- If the vertex to be deleted is a leaf, we can immediately delete it.
- If the vertex to be deleted has only one child, the vertex can be deleted and the child takes the place of the deleted vertex.
- If the vertex to be deleted has two children, the idea is to replace this vertex with the vertex with the smallest key in the right subtree and recursively delete this vertex.

Try to write the pseudocode for this operation (recall that we have the operation for *findMin*).

Running Time

What is the running time for the 6 discussed operations?

- *find:* $O(\text{height of tree}) = O(n)$.
  - This is because on average the height of a binary tree is $\log n$. But the worst case height is n.
- *insert:* $O(n)$
- *delete:* $O(n)$
- *isEmpty:* $O(1)$
- *findMin:* $O(n)$
- *findMax:* $O(n)$