

# Binary search trees

Lecturer: Arran Stewart

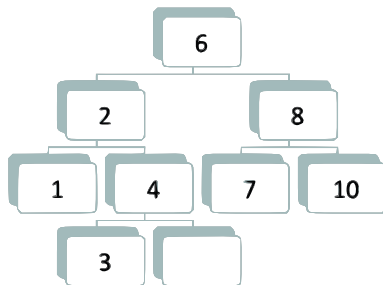
# Outline

We discuss:

- ▶ What is a binary search tree?
- ▶ How are they used?
- ▶ How fast are they to use?

# Binary search trees

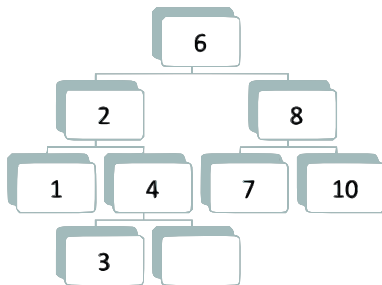
- ▶ A binary search tree is a special type of binary tree, with its nodes in a particular order, which allows us to look up values quickly.
- ▶ Notice that for a tree, every node below the root of a tree is the root of another tree, called a *subtree*.



# Binary search trees

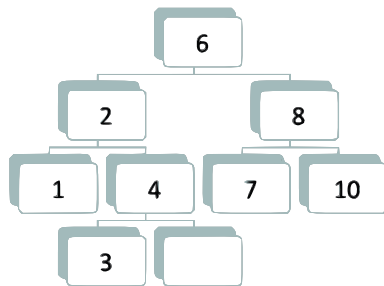
In this tree:

- ▶ the root of the tree is 6
  - ▶ its left child, 2, is the root of a subtree containing 1, 4 and 3.
  - ▶ its right child, 8, is the root of a subtree containing 8, 7 and 10.



## Binary search tree requirements

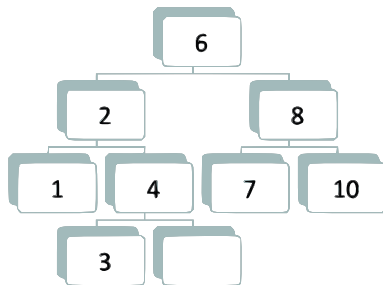
- ▶ To be a binary search tree, a binary tree must satisfy the following property:  
for all nodes, the values in the node's right subtree are greater than the node's value.



- ▶ For example:
  - ▶ 2,1,4 are all less than 6.
  - ▶ 8,2,10 are all greater than 6.
- ▶ At the next level, 1 is less than 2 and 4 and 3 are greater than 2.
- ▶ In the right subtree 7 is less than 8 and 10 is greater than 8.

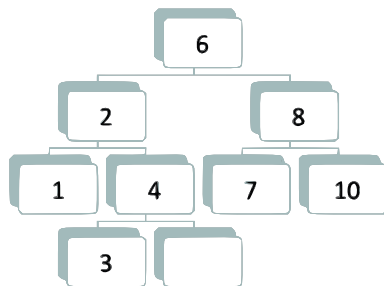
## Duplicate values

- ▶ If we need to handle duplicate values, we can allow identical values to fall either on the left or right, but we should be consistent about it throughout.



## Using a binary search tree

- ▶ The primary use of a binary search tree is to search for values and see if they are in the tree.
- ▶ As an example: How do we check to see if 7 is in the tree?

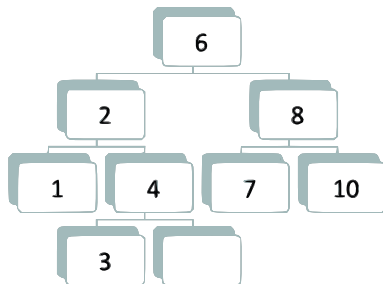


- ▶ Start at the root.
- ▶ 7 is greater than 6, so if it's in the tree, it must be to the right.
- ▶ Then, 7 is less than 8, so it must be left.
- ▶ And we then find 7 in the tree.

## Using a binary search tree

Now, let's search for 5.

- ▶ Start at the root.
- ▶ Five is less than 6, so it must be to the left.
- ▶ Five is greater than 2, so it must be right.



- ▶ Five is also greater than 4, so it must be right again.
- ▶ However, there is no child here. The pointer is null. This means that 5 is not in our tree



# Binary search tree algorithm

We can search the binary tree with the following algorithm:

- ▶ At every node, check to see if we have found the value we are looking for.
- ▶ If we do not find it, determine if it should be on the left or right and check that subtree.
- ▶ Continue down the tree until there is no child node on either the left or right.

# Binary search tree algorithm

In pseudocode:

```
method binarySearch(aNode, key):  
    if aNode == null:  
        return false  
    if aNode.value == key:  
        return true  
    if key > aNode.value:  
        binarySearch(aNode.right)  
    else:  
        binarySearch(aNode.left)
```

## Binary search tree Java code

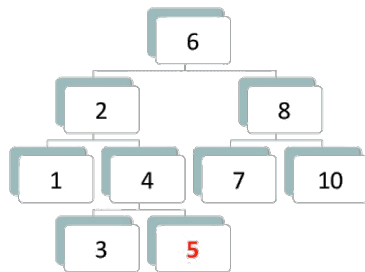
Here is a simplified version of the find method from BinarySearchTree.java implementing this algorithm. We assume that our tree only stores ints.

```
boolean find(int key, BinaryTreeNode n) {  
    if (n == null) { return false; }  
    if (n.value == key) {  
        return true;  
    } else if (key < n.value) {  
        return ( find(key, n.left) );  
    } else {  
        return ( find(key, n.right) );  
    }  
}
```

The complete method in BinarySearchTree.java uses Java generics, so it can store any type of data.

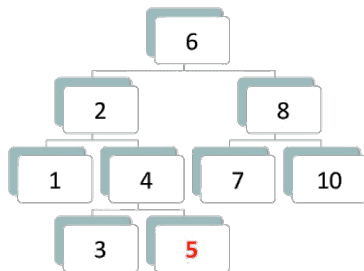
# Inserting new values

- ▶ We can insert new values into the tree, but they must go into the correct place so that the tree still obeys the rules for being a binary search tree.
- ▶ Recall that 5 was not in the tree. How do we insert it?
- ▶ The process looks similar to search.



# Inserting new values

- ▶ We start at the root
- ▶ We compare the value to be inserted with the value at this node –  
five is less than six, so we go left to 2
- ▶ Then we go right to 4
- ▶ and right again
- ▶ but 4 has no child on this side.
- ▶ This will be the new position for 5.
  - ▶ it will start with no children

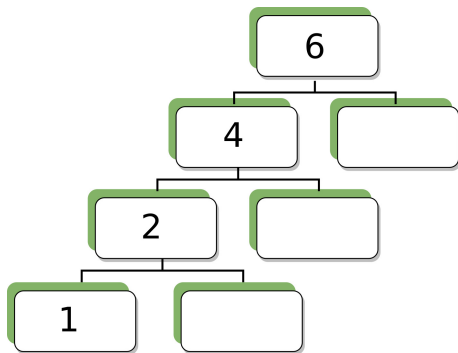


# Complexity of binary search tree operations

- ▶ We would like to know how fast the operations on a binary search tree are.
- ▶ Recall that “Big ‘O’ ” notation seeks to provide an upper bound.

## Degenerate binary search trees

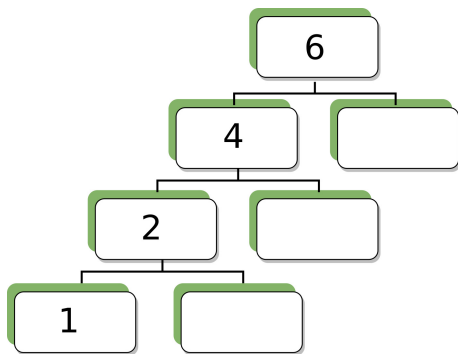
- ▶ The tree on the right *is* a binary search tree – it obeys the rules all such trees have to satisfy.
- ▶ But, searching it will not be quick at all – it will be no better than searching a linked list.



- ▶ (And also, we are wasting space on extra node pointers we don't need.)
- ▶ So in the worst case, our tree could simply be a linked list, and search performance would be  $O(n)$ .
- ▶ The tree here is called a *degenerate* tree.

# Degenerate binary search trees

- ▶ For a degenerate tree, insertion, deletion, and search could take time proportional to the number of nodes in the tree
  - ▶ So all of them are  $O(n)$ .





## Balanced binary search trees

- ▶ Ideally, we would want every node of our binary search tree to have exactly 2 children.
- ▶ This way, insertion, deletion and search would take, at worst,  $O(\log n)$  time.
- ▶ Why?  
Because in such a tree, each layer will half *half* the nodes in the layer below it (until we get to the root layer, which has one node).
- ▶ The most number of “steps” we would have to take to find a node, is therefore the same as the number of times we would have to *halve* the number of nodes in the bottom layer, to get to 1.
- ▶ And we know that the complexity of this is  $O(\log n)$ .

## Balanced binary search trees

- ▶ How much faster than  $O(n)$  is  $O(\log n)$ ?
- ▶ To get a feel for this – suppose we have some operation to perform that takes 1 second, and we need to apply it to a data-set containing one million items.
- ▶ If we had to perform the operation  $\log_2 n$  times, then this would mean the time to process the whole data-set would be

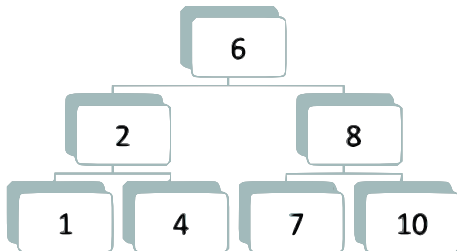
$$\begin{aligned} & \log_2 1\,000\,000 \\ & \approx 19.93 \text{ seconds} \end{aligned}$$

- ▶ Whereas if we had to perform the operation  $n$  times, that would be

$$\begin{aligned} & 1\,000\,000 \text{ seconds} \\ & \approx 16667 \text{ minutes} \\ & \approx 278 \text{ hours} \\ & \approx 11.6 \text{ days.} \end{aligned}$$

## Balanced binary search trees

- ▶ The tree on the right is *balanced*, because every node has exactly 2 children.
- ▶ A tree like this has the *least depth* possible, for its number of nodes.
- ▶ (Whereas a degenerate tree has the *greatest* depth possible.)



# Balanced binary search trees

- ▶ Looking for a value in a balanced binary search tree is similar to binary search on a sorted array.
  - ▶ If we don't need to insert or delete items, they behave exactly the same way.
- ▶ However, a tree structure is better for handling insertions and deletions.
- ▶ Maintaining a balanced tree may need the tree to be re-structured when you insert and delete.
- ▶ The code for this is tricky and it won't be covered here. But you can read about it in the text book.

# Use of balanced binary search trees

- ▶ Are balanced binary search trees commonly used?
- ▶ They are. They are very frequently found in the standard libraries for programming languages, because they provide a convenient way of:
  - ▶ searching a collection of items efficiently
  - ▶ adding items to the collection
  - ▶ removing items from the collection
- ▶ The Java class `java.util.TreeMap`, from the Java standard library, is frequently used to store elements you wish to search, and is implemented using a balanced binary search tree.

## Use for abstract data types

- ▶ Linked lists and binary search trees are both *concrete* data structures.
- ▶ They are defined by their structure – e.g. linked list nodes always have one pointer to another node, and contain one value.
- ▶ However, they can be used to implement *abstract* data types.

## Use for abstract data types

- ▶ For instance, suppose we wanted to define a **Set** abstract data type
  - ▶ A set is used for storing a collection of items, but never contains multiple instances of an item.
- ▶ The operations we would want are:
  - ▶ insert an item
  - ▶ delete an item
  - ▶ search to see whether some item is in the set.

## Use for abstract data types

There are many ways we could implement the Set abstract data type ...

- ▶ We could implement a Set using an array (as we did for stacks and queues).
- ▶ We will discuss briefly how to do this.
- ▶ However, this has the disadvantage that the set will have a maximum capacity – as we have to declare the size of an array when it is created
- ▶ Also, unless we can keep the array sorted, search will have  $O(n)$  complexity.
- ▶ Inserting or deleting an item at the beginning of the array will also have poor performance – both have  $O(n)$  complexity.
  - ▶ They also are slow, compared to linked list operations, because many items must be copied when this happens.



## Use for abstract data types

- ▶ We could implement a Set using a linked list (again, as we did for stacks and queues).
- ▶ Again, we will discuss briefly how to do this.
- ▶ This means we no longer have the restriction that our Set must have a maximum capacity.
- ▶ However, search, insertion and deletion still have  $O(n)$  complexity (though insertion and deletion will be faster than for an array).

## Use for abstract data types

- ▶ Or, we could implement a Set using a binary search tree.
- ▶ We will discuss how this could be done.
  
- ▶ Similar to linked lists, there is no restriction on our Set's maximum capacity.
- ▶ Furthermore, insertion, deletion and search now have  $O(\log n)$  complexity – a big improvement.