Temasek Junior College
2023 JC2 H2 Computing
Data Structures 7: Binary Search Trees

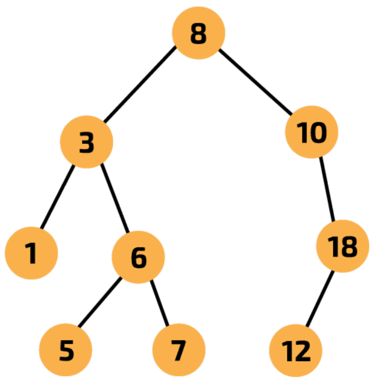
1 Introduction to Binary Search Trees

A binary search tree is a special type of binary tree, where the nodes of this rooted tree are ordered to optimise searching.

If the order is ascending (low to high):
- nodes of the left subtree have values that are lower than the root,
- nodes of the right subtree have values that are higher than the root.

This property is **also true for any parent node** of the tree;
- nodes of the left subtree will have values that are lower than the parent,
- nodes of the right subtree will have values that are higher.

Consider the following binary search tree:



Imagine that you are searching for the number 5:
- Start at the root 8.
- Compare 5 with 8. Since 5 is lower, check the left child.
- Compare 5 with 3. Since 5 is higher, check the right child.
- Compare 5 with 6. Since 5 is lower, check the left child.
- Compare 5 with 5. Successfully found!

You have found the item with four comparisons.

Observe that at each comparison, you are sent either left or right, and therefore the values in the other subtree do not need to be searched. This is similar to the approach taken in a binary search. However, to be as efficient as a binary search, the tree must be **balanced**.

In a **balanced** tree, every leaf (a node at the end of a branch) is around the same distance away from the root as any other leaf. In this case, the binary search tree algorithm is as efficient as the binary search.

In an **unbalanced** tree, one or more leaves is much further away from the root than another leaf. In this case, the binary search tree algorithm is less efficient than the binary search.

Notice that for a tree to be balanced, it does not have to have the same number of nodes in the left and right subtrees. The important thing is that each branch (from root to leaf) is roughly the same **height**. The height of a branch will determine the maximum number of comparisons that could be carried out when searching that branch of the tree.

In the example above, there are four leaf nodes. There are three edges between the root and each of the nodes with values 5, 7 and 12. Although the remaining branch has only two edges (between the root and the node with value 1), this tree can be considered balanced. Trees can get unbalanced when nodes are added or deleted and so will need to be rebalanced to optimise search performance.

2 Binary Tree Search Algorithm

The pseudocode below gives a typical example of a binary tree search algorithm.

```
FUNCTION binary_tree_search(node, search_item)

    IF search_item == node.data THEN
        RETURN True

    ELSEIF search_item > node.data AND node.right != Null THEN
        RETURN binary_tree_search(node.right, search_item)

    ELSEIF search_item < node.data AND node.left != Null THEN
        RETURN binary_tree_search(node.left, search_item)

    ENDIF

    RETURN False

ENDFUNCTION
```

The binary tree search algorithm returns `True` if the value is found and `False` if the value is not found. It is a **recursive** function which will take two arguments:

- `node` which is the current node being examined,
- `search_item` which is the item being searched for.

As the data is stored in a tree, we refer to each element as a node rather than an item (as in a list). The node is a structure that contains:

- `data` which in this case is a simple value attached to the node,
- `right` which is a reference to the **node to the right** of the node,
- `left` which is a reference to the **node to the left** of the node.

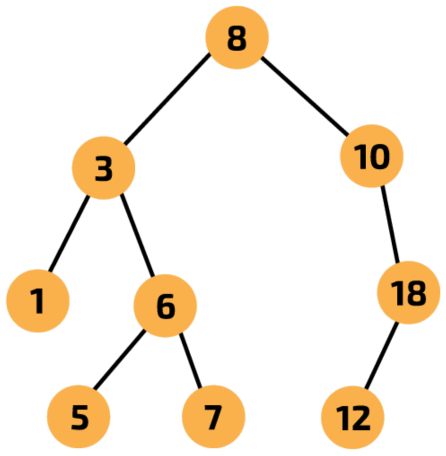
For example, if we examined the root node:

- `node.data` would be 8,
- `node.right` would return a reference to the node that contained the data value 10,
- `node.left` would return a reference to the node that contained the data value 3.

If there is no node to the right, `node.right` will contain a `Null` value; if there is no node to the left, `node.left` will contain a `Null` value.

3 Tracing the Binary Tree Search Algorithm

Let us attempt to trace the algorithm using a trace table.



Imagine that you are searching for the number 6 in the above binary search tree.

When tracing a recursive algorithm, you must remember that there will be more than one version of the subroutine active (i.e. on the call stack) at one time, and each will have its own local data. There may also be shared data. In this example, the tree data and the value of `search_item` will remain constant throughout, but the value of `node` will change.

When tracing an algorithm, it is really useful to number the lines of code. This will allow you to keep track of where you need to return to as each version of the subroutine completes.

- The **first** time that the subroutine is called:
  - `node` points to the item whose data value is 8,
  - this is **higher** than `search_item` and a left node exists, so the subroutine is called again from line 7 with the pointer for the left node.

- On the **second** call:
  - `node` points to the item whose data value is 3,
  - this is **lower** than `search_item` and a right node exists, so the subroutine is called again from line 5 with the pointer for the right node.

- On the **third** call:
  - `node` points to the item whose data value is 6,
  - this matches the `search_item` and a right node exists, so the value `True` is returned from line 3
  - the third version of the subroutine has completed and will be removed from the call stack.

- The **second** call will resume on line 5:
  - where it receives the returned value `True`
  - and returns `True` to the still opened first call
  - the second version of the subroutine has completed and will be removed from the call stack.

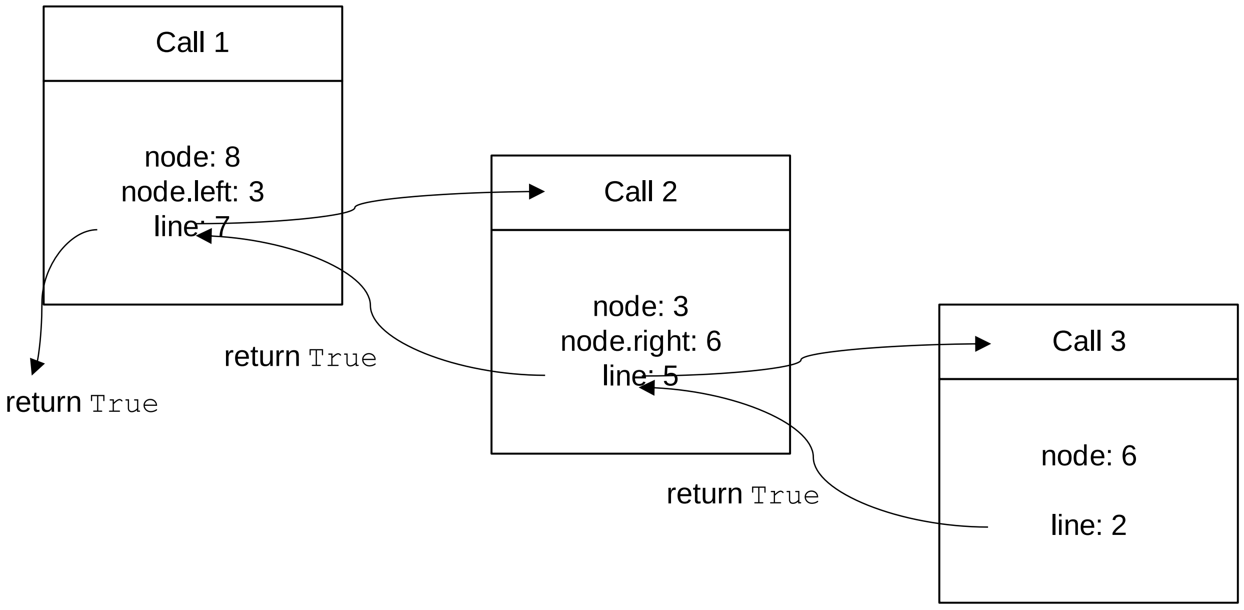
- The **first** call will resume on line 7:
  - where it receives the returned value `True`

- and returns `True` (as the “final answer”)
- the first version of the subroutine has completed and will be removed from the call stack.

A simple trace table of the search process can be constructed as follows:

Call	**`node`**	**Return value**
1	8	
2	3	
3	6	`True`
2		`True`
1		`True`

The same process can be represented using a recursive trace diagram as follows:



4 Time Complexity of the Binary Tree Search Algorithm

To determine the time complexity of the binary tree search algorithm, you must consider the **main operation**. The main operation here is checking each node in the tree to see whether its value matches the item being searched for.

With a binary tree structure, the maximum number of comparisons will depend on the **balance** of the tree.

Balanced Tree

Recall that a **balanced tree** is a tree where every leaf is about the same distance away from the root as any other leaf. In this case, the branches of the tree have largely the same length.

In a completely balanced tree, the nodes are evenly distributed between the left and right subtrees. As such, for a completely balanced tree with height *h*, the total number of nodes is

$$n = 1 + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h$$

which is a geometric progression of h + 1 terms, where the 1st term is 1 and common ratio is h.

As, such

$$n = \frac{1\,(2^{h+1} - 1)}{2 - 1} \quad = \ (2^{h+1} - 1)$$

Since the time needed for the search depends directly on h, we can make h the subject as follows:

$$2^{h+1} = n + 1$$

$$(h + 1) \log_2 2 = \log_2 (n + 1)$$

$$h + 1 = \log_2 (n + 1)$$

$$h = \log_2 (n + 1) - 1$$

Applying big-O notation, the time complexity will be

$$O \log (n)$$

Hence

- In the **best case**, the time complexity will be O(1). This will be achieved if the item you are looking for is found at the root.
- In the **worst case**, the maximum number of comparisons needed will be O(log n) where n is the number of nodes in the tree). This has the same worst case time complexity as a binary search.

<u>**Unbalanced Tree**</u>

Recall that an **unbalanced tree** is a tree where every leaf is much further away from the root than any other leaf. In this case, the branches of the tree have widely differing lengths. In a completely unbalanced tree, all of the nodes are contained in a single branch.

- In the **best case**, the time complexity will be O(1). This will be achieved if the item you are looking for is found at the root of the single branch.
- In the **worst case**, the maximum number of comparisons needed will be O(n) (where n is the number of nodes in the tree). This will happen if the item you are looking for is at the last node of the single branch or it is not found in the tree.

5 Binary Search Trees vs. Binary Search

Searching using a binary search tree works the same way as the binary search in a linear sorted array.

The algorithm for binary search also has time complexity of O(log n) since the search scope is always reduced by half for every iteration, making it one of the most time efficient algorithms.