Module 6 : BINARY SEARCH TREE

Course Learning Outcomes:

1. Learn on how to build a Binary Tree.
2. Learn to delete nodes in a Binary tree.
3. Know how to check binary tree into binary search tree.

Binary Search Tree

In computer science, binary search trees, sometimes called ordered or sorted binary trees, are a particular type of container. A data structure that stores "items" in memory.

Binary search tree is a node- based binary tree data structure which has the following properties : **The left subtree** of a node contains only nodes with keys lesser than the node's key. **The right subtree** of a node contains only nodes with keys greater than the node's key.

Searching with arrays or lists

As we have already seen in Chapter 3, many computer science applications involve searching for a particular item in a collection of data. If the data is stored as an unsorted array or list, then to find the item in question, one obviously has to check each entry in turn until the correct one is found, or the collection is exhausted. On average, if there are n items, this will take n/2 checks, and in the worst case, all n items will have to be checked. If the collection is large, such as all items accessible via the internet, that will take too much time. We also saw that if the items are sorted before storing in an array, one can perform binary search which only requires log2 n checks in the average and worst cases. However, that involves an overhead of sorting the array in the first place, or maintaining a sorted array if items are inserted or deleted over time. The idea here is that, with the help of binary trees, we can speed up the storing and search process without needing to maintain a sorted array.

Search keys

If the items to be searched are labelled by comparable keys, one can order them and store them in such a way that they are sorted already. Being 'sorted' may mean different things for different keys, and which key to choose is an important design decision.

In our examples, the search keys will, for simplicity, usually be integer numbers (such as student ID numbers), but other choices occur in practice. For example, the comparable keys could be words. In that case, comparability usually refers to the alphabetical order. If w and

t are words, we write w < t to mean that w precedes t in the alphabetical order. If w = bed and t = sky then the relation w < t holds, but this is not the case if w = bed and t = abacus. A classic example of a collection to be searched is a dictionary. Each entry of the dictionary is a pair consisting of a word and a definition. The definition is a sequence of words and punctuation symbols. The search key, in this example, is the word (to which a definition is attached in the dictionary entry). Thus, abstractly, a dictionary is a sequence of entries, where an entry is a pair consisting of a word and its definition. This is what matters from the point of view of the search algorithms we are going to consider. In what follows, we shall concentrate on the search keys, but should always bear in mind that there is usually a more substantial data entry associated with it.

BINARY SEARCH TREE

The solution to our search problem is to store the collection of data to be searched using a binary tree in such a way that searching for a particular item takes minimal effort. The underlying idea is simple: At each tree node, we want the value of that node to either tell us that we have found the required item, or tell us which of its two subtrees we should search for it in. For the moment, we shall assume that all the items in the data collection are distinct, with different search keys, so each possible node value occurs at most once, but we shall see later that it is easy to relax this assumption. Hence we define :

Definition. A *binary search tree* is a binary tree that is either empty or satisfies the following conditions:

- All values occurring in the left subtree are smaller than that of the root.
- All values occurring in the right subtree are larger than that of the root.
- The left and right subtrees are themselves binary search trees.

So this is just a particular type of binary tree, with node values that are the search keys. This means we can *inherit* many of the operators and algorithms we defined for general binary trees. In particular, the primitive operators `MakeTree(v,l,r)`, `root(t)`, `left(t)`, `right(t)` and `isEmpty(t)` are the same – we just have to maintain the additional node value ordering.

Building Binary Search tree

When building a binary search tree, one naturally starts with the root and then adds further new nodes as needed. So, to insert a new value v, the following cases arise:

• If the given tree is empty, then simply assign the new value v to the root, and leave the left and right subtrees empty.

• If the given tree is non-empty, then insert a node with value v as follows:

– If v is smaller than the value of the root: insert v into the left sub-tree.

– If v is larger than the value of the root: insert v into the right sub-tree.

– If v is equal to the value of the root: report a violated assumption.

Thus, using the primitive binary tree operators, we have the procedure:

```
insert(v,bst) {
   if ( isEmpty(bst) )
      return MakeTree(v, EmptyTree, EmptyTree)
   elseif ( v < root(bst) )
      return MakeTree(root(bst), insert(v,left(bst)), right(bst))
   elseif ( v > root(bst) )
      return MakeTree(root(bst), left(bst), insert(v,right(bst)))
   else error('Error: violated assumption in procedure insert.')
}
```

added is always a leaf. The resulting tree is once again a binary search tree. This can be proved rigorously via an inductive argument.

Note that this procedure creates a new tree out of a given tree **bst** and new value **v**, with the new value inserted at the right position. The original tree **bst** is not modified, it is merely inspected. However, when the tree represents a large database, it would clearly be more efficient to modify the given tree, rather than to construct a whole new tree. That can easily be done by using pointers, similar to the way we set up linked lists. For the moment, though, we shall not concern ourselves with such implementational details.

Searching a binary search tree

Searching a binary search tree is not dissimilar to the process performed when inserting a new item. We simply have to compare the item being looked for with the root, and then keep 'pushing' the comparison down into the left or right subtree depending on the result of each root comparison, until a match is found or a leaf is reached.

Algorithms can be expressed in many ways. Here is a concise description in words of the search algorithm that we have just outlined:

> In order to search for a value `v` in a binary search tree `t`, proceed as follows. If `t` is empty, then `v` does not occur in `t`, and hence we stop with `false`. Otherwise, if `v` is equal to the root of `t`, then `v` does occur in `t`, and hence we stop returning `true`. If, on the other hand, `v` is smaller than the root, then, by definition of a binary search tree, it is enough to search the left sub-tree of `t`. Hence replace `t` by its left sub-tree and carry on in the same way. Similarly, if `v` is bigger than the root, replace `t` by its right sub-tree and carry on in the same way.

Notice that such a description of an algorithm embodies both the steps that need to be carried out and the reason why this gives a correct solution to the problem. This way of describing algorithms is very common when we do not intend to run them on a computer.

When we do want to run them, we need to provide a more precise specification, and would normally write the algorithm in pseudocode, such as the following recursive procedure:

```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

Each recursion restricts the search to either the left or right subtree as appropriate, reducing the search tree height by one, so the algorithm is guaranteed to terminate eventually.

In this case, the recursion can easily be transformed into a while-loop:

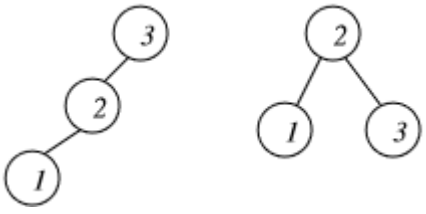
```
isIn(value v, tree t) {
   while ( (not isEmpty(t)) and (v != root(t)) )
      if (v < root(t) )
         t = left(t)
      else
         t = right(t)
   return ( not isEmpty(t) )
}
```

Here, each iteration of the while-loop restricts the search to either the left or right subtree as appropriate. The only way to leave the loop is to have found the required value, or to only have an empty tree remaining, so the procedure only needs to return whether or not the final tree is empty.

Time complexity of insertion and search

As always, it is important to understand the time complexity of our algorithms. Both item insertion and search in a binary search tree will take at most as many comparisons as the height of the tree plus one. At worst, this will be the number of nodes in the tree. But how many comparisons are required on average? To answer this question, we need to know the average height of a binary search tree. This can be calculated by taking all possible binary search trees of a given size n and measuring each of their heights, which is by no means an easy task. The trouble is that there are many ways of building the same binary search tree by successive insertions.

As we have seen above, perfectly balanced trees achieve minimal height for a given number of nodes, and it turns out that the more balanced a tree, the more ways there are of building it. This is demonstrated in the figure below:



The only way of getting the tree on the left hand side is by inserting 3, 2, 1 into the empty tree in that order. The tree on the right, however, can be reached in two ways: Inserting in the order 2, 1, 3 or in the order 2, 3, 1. Ideally, of course, one would only use well-balanced trees to keep the height minimal, but they do not have to be perfectly balanced to perform better than binary search trees without restrictions.

Carrying out exact tree height calculations is not straightforward, so we will not do that here. However, if we assume that all the possible orders in which a set of n nodes might be inserted into a binary search tree are equally likely, then the average height of a binary search tree turns out to be O(log2 n). It follows that the average number of comparisons needed to search a binary search tree is O(log2 n), which is the same complexity we found for binary search of a sorted array. However, inserting a new node into a binary search tree also depends on the tree height and requires O(log2 n) steps, which is better than the O(n) complexity of inserting an item into the appropriate point of a sorted array.

Interestingly, the average height of a binary search tree is quite a bit better than the average height of a general binary tree consisting of the same n nodes that have not been built into a binary search tree. The average height of a general binary tree is actually O($\sqrt{n}$). The reason for that is that there is a relatively large proportion of high binary trees that are not valid binary search trees.

Deleting nodes from a binary search tree

Suppose, for some reason, an item needs to be removed or deleted from a binary search tree. It would obviously be rather inefficient if we had to rebuild the remaining search tree again from scratch. For n items that would require n steps of O(log2 n) complexity, and hence have overall time complexity of O(nlog2 n). By comparison, deleting an item from a sorted array would only have time complexity O(n), and we certainly want to do better than that. Instead, we need an algorithm that produces an updated binary search tree more efficiently. This is more complicated than one might assume at first sight, but it turns out that the following algorithm works as desired:

- If the node in question is a leaf, just remove it.
- If only one of the node's subtrees is non-empty, 'move up' the remaining subtree.
- If the node has two non-empty sub-trees, find the 'left-most' node occurring in the right sub-tree (this is the smallest item in the right subtree). Use this node to overwrite the one that is to be deleted. Replace the left-most node by its right subtree, if this exists; otherwise just delete it.

The last part works because the left-most node in the right sub-tree is guaranteed to be bigger than all nodes in the left sub-tree, smaller than all the other nodes in the right sub-tree, and have no left sub-tree itself. For instance, if we delete the node with value 11 from the tree in Figure 6.1, we get the tree displayed in Figure 7.1.

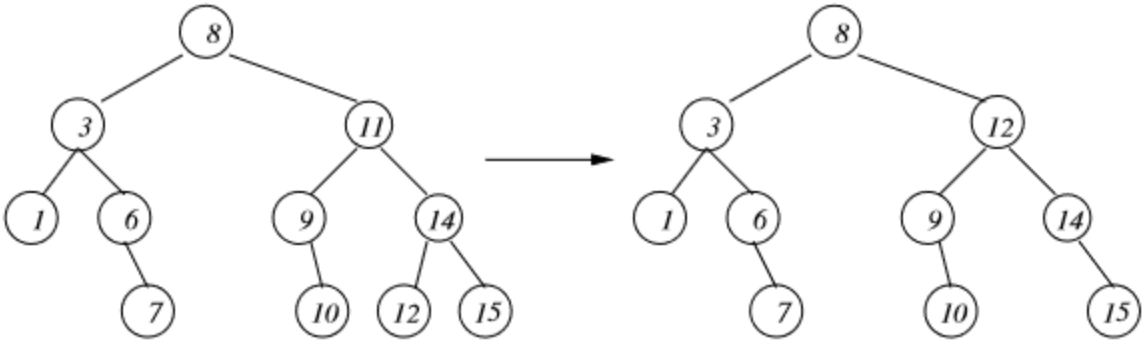


Figure 7.1: Example of node deletion in a binary search tree.

In practice, we need to turn the above algorithm (specified in words) into a more detailed algorithm specified using the primitive binary tree operators:

```
delete(value v, tree t) {
  if ( isEmpty(t) )
     error('Error: given item is not in given tree')
  else
     if ( v < root(t) )    // delete from left sub-tree
        return MakeTree(root(t), delete(v,left(t)), right(t));
     else if ( v > root(t) )     // delete from right sub-tree
        return MakeTree(root(t), left(t), delete(v,right(t)));
     else    // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
           return right(t)
        elseif ( isEmpty(right(t)) )
           return left(t)
        else    // difficult case with both subtrees non-empty
           return MakeTree(smallestNode(right(t)), left(t),
                                          removeSmallestNode(right(t))
}
```

If the empty tree condition is met, it means the search item is not in the tree, and an appropriate error message should be returned.

The delete procedure uses two sub-algorithms to find and remove the smallest item of a given sub-tree. Since the relevant sub-trees will always be non-empty, these sub-algorithms can be written with that precondition. However, it is always the responsibility of the programmer to ensure that any preconditions are met whenever a given procedure is used, so it is important to say explicitly what the preconditions are. It is often safest to start each procedure with a check to determine whether the preconditions are satisfied, with an appropriate error message produced when they are not, but that may have a significant time cost if the procedure is called many times. First, to find the smallest node, we have:

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}
```

which uses the fact that, by the definition of a binary search tree, the smallest node of t is the left-most node. It recursively looks in the left sub-tree till it reaches an empty tree, at which point it can return the root. The second sub-algorithm uses the same idea:

```
removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

except that the remaining tree is returned rather than the smallest node.

These procedures are further examples of recursive algorithms. In each case, the recursion is guaranteed to terminate, because every recursive call involves a smaller tree, which means that we will eventually find what we are looking for or reach an empty tree.

It is clear from the algorithm that the deletion of a node requires the same number of steps as searching for a node, or inserting a new node, i.e. the average height of the binary search tree, or O(log2 n) where n is the total number of nodes on the tree.

Checking whether a binary tree is a binary search tree

Building and using binary search trees as discussed above is usually enough. However, another thing we sometimes need to do is check whether or not a given binary tree is a binary search tree, so we need an algorithm to do that. We know that an empty tree is a (trivial) binary search tree, and also that all nodes in the left sub-tree must be smaller than the root and themselves form a binary search tree, and all nodes in the right sub-tree must be greater than the root and themselves form a binary search tree. Thus the obvious algorithm is:

```
isbst(tree t) {
   if ( isEmpty(t) )
      return true
   else
      return ( allsmaller(left(t),root(t)) and isbst(left(t))
                 and allbigger(right(t),root(t)) and isbst(right(t)) )
}
```

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and allsmaller(left(t),v)
                                        and allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) > v) and allbigger(left(t),v)
                                        and allbigger(right(t),v) )
}
```

However, the simplest or most obvious algorithm is not always the most efficient. Exercise: identify what is inefficient about this algorithm, and formulate a more efficient algorithm.

References and Supplementary Materials

Books and Journals

1. John Bullinaria ; March 2019 ; Data Structures and Algorithms ; Birmingham, UK;
2. Michael T Goodrich ; Roberto Tamassia ; David Mount ; Second Edition ; Data Structures and Algorithm in C++
3. www.google.com
4. Merriam Webster Dictionary