

Binary Search Trees

Today we consider a specific type of binary tree in which there happens to be an ordering defined on the set of elements at each node. For example, if the elements are numbers or strings, then again there is a natural ordering. For strings, this is the dictionary ordering, also known as “lexicographic ordering”.

A *binary search tree* is a binary tree such that

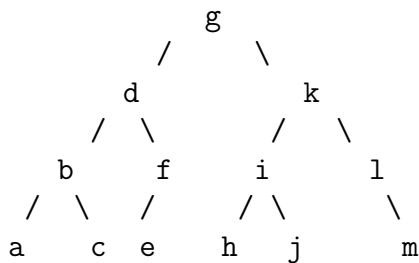
- the elements are comparable, namely there is a strict ordering relation $<$,
- different nodes have different elements
- for any node,
 - all nodes in the left subtree are less than the node
 - all nodes in the right subtree are greater than the node.

We will refer to the element stored at binary search tree as a *key*, for reasons that will be clear later in the course, namely when we discuss “maps”.

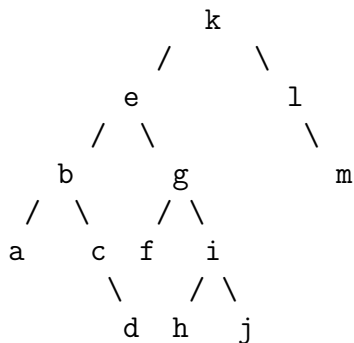
Key property: *an inorder traversal of a binary search tree gives the elements in their correct order.*

Example 1

Here is an example with nodes containing keys `abcdefghijklm`. Verify that an inorder traversal gives the elements in their correct order.

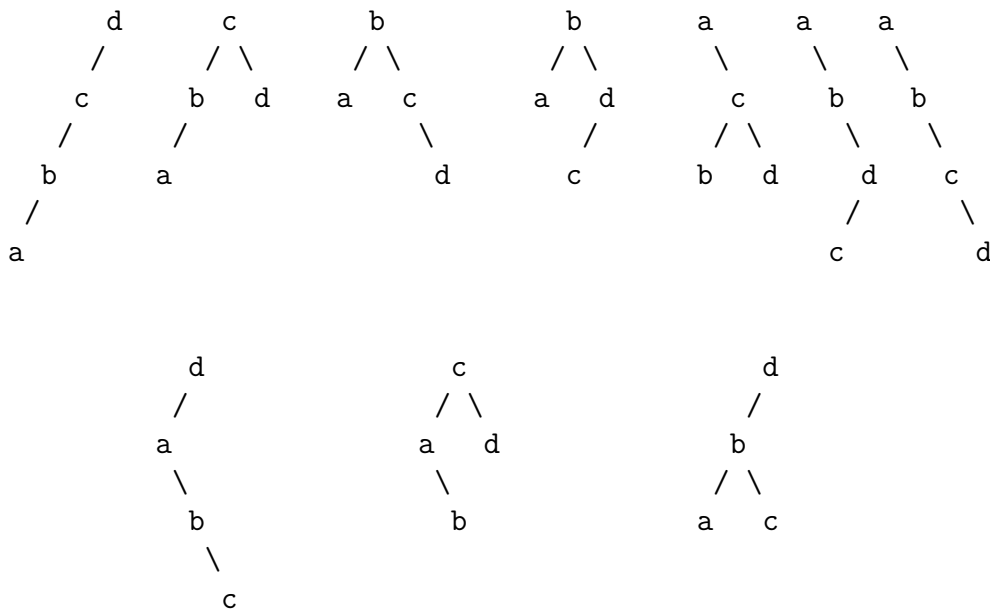


and here is another example with the same set of keys:



Example 2

Here are several examples of binary search trees whose nodes contain the characters a, b, c, d. There are other examples as well (see lecture slides).



Enumerating BSTs with n nodes

Here is a systematic way to enumerate all the binary search trees (BSTs) containing a given set of elements. The idea is to consider all possible binary search trees with each element at the root. If there are n nodes, then for each of the n choices of root node, there are $n - 1$ non-root nodes and these non-root nodes must be partitioned into those that are less than a chosen root and those that are greater than the chosen root.

If node i is the root, then there are $i - 1$ nodes smaller than i and $n - i$ nodes bigger than i . For each of these two sets of nodes, there is a certain number of possible subtrees. Let $t(n)$ be the total number of BSTs with n nodes. The total number of BSTs with i at the root is $t(i - 1) * t(n - i)$. The two terms are multiplied together because the arrangements in the left and right subtrees are independent. That is, for each arrangement in the left tree and for each arrangement in the right tree, you get one BST with i at the root. Summing over i then gives the total number of binary search trees with n nodes,

$$t(n) = \sum_{i=1}^n t(i - 1) t(n - i).$$

The base case is $t(0) = 1$ and $t(1) = 1$, i.e. there is one empty BST and there is one BST with one node.

$$t(2) = t(0)t(1) + t(1)t(0) = 2$$

$$t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 2 + 1 + 2 = 5$$

$$t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 5 + 2 + 2 + 5 = 14$$

BST abstract data type (ADT) and algorithms

One performs several common operations on binary search trees:

- `find(key)`: given a key, return a reference to the node containing that key (or null if key is not in tree)
- `findMinimum()` or `findMaximum()`: find the node containing the smallest or largest key in the tree and return a reference to the node containing that key
- `add(key)`: insert a new node into the tree such that the node contains the key and the node is in its correct position (if the key is already in the tree, then do nothing)
- `remove(key)`: remove from the tree the node containing the key (if it is present)

There are various ways of specifying these operations. Here are a few examples.

```
find(root,key){
  if (root == null)
    return null
  else if (root.key == key))
    return root
  else if (key < root.key)
    return find(root.left, key)
  else
    return find(root.right, key)
}
```

```
findMinimum(root){
  if (root == null)    // only necessary for the first call
    return null
  else if (root.left == null)
    return root
  else
    return findMinimum(root.left)
}
```

Notice that the minimum key is not necessarily a leaf i.e. it can occur if the key has a right child but no left child.

```
findMaximum(root){
  if (root == null)
    return null
  else if (root.right == null))
    return root
  else
    return findMaximum(root.right)
}
```