Binary Search Trees

Throughout this course, we will use a wide variety of trees to solve problems with the algorithms we study. One of the more commonly used tree structures is the **binary search tree**. A binary search tree is a binary tree where the keys are rooted in the tree in such a way to facilitate easy search. For this to happen, the binary search tree imposes a particular property on the orientation of the key values in the tree.

Def: The binary search tree property is a property imposed on every vertex of a binary search tree such that the following conditions hold. Let v_i and v_j be two vertices in the tree. For the sake of discussion, assume all key values are distinct (although, in general, this is not required).

If v_j is located in the left subtree of v_i , then $key[v_j] < key[v_i]$.

On the other hand, if v_j is located in the right subtree of v_i, then $key[v_j] > key[v_i]$.

Any given binary tree can be traversed in a number of interesting ways. Tree traversal corresponds to visiting all of the vertices in a tree in some order. Three approaches that yield interesting and useful results are:

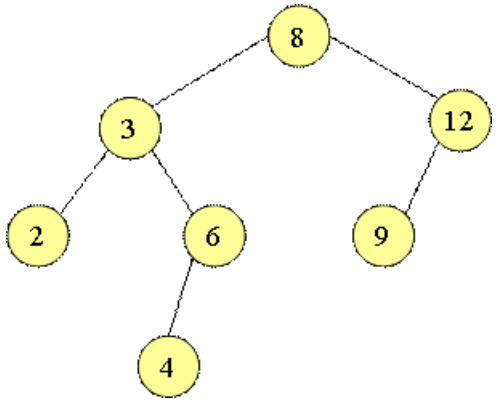
- Inorder traversal
- Preorder traversal
- Postorder traversal

Each of these traversals can be performed in $\Theta(n)$ time where $n = |\mathbf{T}|$, and we can represent these choices using the following single algorithm:

Algorithm 1 Tree Traversal

```
TreeTraverse(root, type)
  if root = NIL then return
  if type = "preorder" then
    print(key[root])
  TreeTraverse(root.left, type)
  if type = "inorder" then
    print(key[root])
  TreeTraverse(root.right, type)
  if type = "postorder" then
    print(key[root])
end
```

Interpreting this algorithm, we note that **root** corresponds to a pointer to the root of a subtree, and **type** specifies the type of traversal from the set of values {**preorder**, **inorder**, **postorder**}. From this, we see that a preorder traversal involves "visiting" the root of the tree first (indicated by printing that nodes key value; however, any operation can be performed during a "visit"), then traversing left subtree and traversing right subtree. Consider the following tree:



In this tree, a preorder traversal yields the sequence 8–3–2–6–4–12–9. Before looking at the inorder traversal, let's consider postorder. In a postorder traversal, from a root node, we traverse the left subtree, then traverse the right subtree, and then visit the root. Again, using the tree above, we see that postorder traversal yields the sequence 2–4–6–3–9–12–8.

Finally, let's consider the inorder traversal. For this traversal, we begin by traversing the left subtree from the root, then we visit the root node, and finally we traverse the right subtree. The reason we saved this for last is that, when applying an inorder traversal to a binary search tree, something interesting happens.

For example, consider the tree above. An inorder traversal of this tree yields the sequence 2–3–4–6–8–9–12. In other words, an inorder traversal of any binary search tree will visit the vertices of the tree in non-decreasing order of the key values stored in the tree.