Binary Trees Summarized

Definition: A binary tree is a rooted tree where each node has at most two children. It is proper if each internal node has two children.

In an inorder traversal of a proper binary tree, the nodes in the left subtree of a node \( v \) are visited first if they exist, followed by \( v \), followed by the nodes in the right subtree of \( v \) if they exist. Hence, when \( v \) is an internal node, \( \text{inorderBefore}(v) \) is the bottom rightmost node in its left subtree while \( \text{inorderAfter}(v) \) is the bottom leftmost node in its right subtree. On the other hand, when \( v \) is an external node, \( \text{inorderBefore}(v) \) is the lowest ancestor of \( v \) that has \( v \) in its right subtree while \( \text{inorderAfter}(v) \) is the lowest ancestor of \( v \) that has \( v \) in its left subtree. Note that when \( v \) is the first node visited by the inorder traversal, \( \text{inorderBefore}(v) \) does not exist. Similarly, when \( v \) is the last node visited by the inorder traversal, \( \text{inorderAfter}(v) \) does not exist.

Since the bottom nodes of a proper binary tree are external nodes, our above discussion implies that an inorder traversal of a proper binary alternately visits internal and external nodes. That is, it first visits an external node (the bottom leftmost node of the tree), then an internal node, followed by an external node, etc., and finally an external node (the bottom rightmost node of the tree). This observation allows us to prove the following theorem:

**Theorem 2.9:** In a proper binary tree, the number of external nodes is one more than the number of internal nodes.

Binary trees as data structure

Thus far, everything that we have mentioned about a binary tree is structural in nature – that is, it is true whether or not the tree contains any data. As we saw in class though, binary trees are quite useful for storing data. For example, representing arithmetic expressions as shown in Figure 2.17 is one such instance. The numbers in the expression are stored in the external nodes while the arithmetic operators are stored in the internal nodes. A proper binary tree models arithmetic expression naturally because arithmetic operators like +, −, ×, / are binary; that is, they need “left” and “right” numbers to operate.

Another use of binary trees is for implementing ADT’s – specifically Priority Queues and Ordered Dictionary ADT’s. In this case, the data does not naturally form a tree. Instead, we place them in a binary tree with some extra rules so that the methods of the said ADT’s can be implemented quickly. For this application, the book makes two technical requirements – the binary tree is proper and the external nodes do not contain any data. (These two requirements actually go hand-in-hand.
Since the data stored in these ADT’s will not naturally form a proper binary tree, some internal nodes might be missing a child. This is fixed by attaching the missing child to the node where the missing child is an “empty” external node. The book describes this as a “NULL_NODE” object.

In a binary search tree (BST), an implementation of the Ordered Dictionary ADT, the extra rule we use is this: At every node \( v \), the key stored at \( v \) has to be greater than or equal to the keys stored in its left subtree and less than or equal to the keys stored in its right subtree. This rule turns out to be quite powerful. It organizes the keys in such a way that they can be sorted quickly. Here’s how: Run an inorder traversal on the BST. If item (\( k, e \)) is visited before (\( k', e' \)) then \( k \leq k' \). Hence, if we print the keys of the items as they are visited, the keys will be sorted from smallest to largest.

Now, there’s a slight technical issue here – when we do an inorder traversal on a BST, are the empty external nodes visited or not? The book is vague on this. (See for example the second paragraph on page 145, the first time the inorder traversal of a BST is mentioned.) To be consistent, let us assume that the external nodes are visited, but the traversal simply moves on when they are encountered since they do not contain any data.

Here then is a clarification to yesterday’s lecture. Suppose we wish to remove an item (\( k, e \)) stored in node \( w \) with two internal children. To maintain the BST, we have to replace this item with something else. We said that the ideal replacement is either the item visited just before or just after (\( k, e \)) in an inorder traversal of the tree. Where are they located? The item visited just before (\( k, e \)) lies in the bottom rightmost internal node of the left subtree of \( w \) while the item visited just after (\( k, e \)) lies in the bottom leftmost internal node in the right subtree of \( w \).\(^1\)

\(^1\)In class, I mentioned that they lie in \text{inorderBefore}(w) or \text{inorderAfter}(w) respectively. With our assumption, they can’t because the latter two nodes are external nodes and should therefore be empty.