STAT 534

Lecture 4

Data Structures–Binary Tree

04/11/19

mmp@stat.washington.edu

Scribes: Ziyu Jiang, Yikang Shen

1 Binary Trees

1.1 Basic Definitions

In this lecture, we continue our introduction of data structures in addition to lists and stacks, which we covered in the last lecture. Today we are going to talk about binary trees.

Binary tree is a type of graph, in which

- Each node has (at most) two distinguished children, the children on the left is called **left child**, while the one of the right is called **right child**.
- Each node has a **parent** except the top node (which is called **root**).

Notice that not all roots can have two children. If a node does not have (left or right) child, the child is called **NULL**. If a node does not have any child, the node is called **leaf**.

1.2 Data Structure Implementations

Every node contains a fixed number of fields of information:

- A pointer to the parent;
- Pointers to the left child and right child or (NULL);
- Information of the node. **Key** is a particular type of information by which the node is distinguished and it can be viewed as the "value" of a node. Additionally, a node can have other types of information.

2 Possible Operations on Binary Trees

Below operations are useful in relation to binary trees:

- List all nodes (in order)
- Find certain values in the tree (search)

- Insert nodes
- Delete nodes
- Find the node with maximum or minimum key

2.1 Tree Property

The nodes in a binary tree is not placed in an arbitrary order since they have to follow the **tree property** listed as below:

$$\text{key}(z) \leq \text{key}(x) \leq \text{key}(y), \forall z \in \text{left subtree}(x), \forall y \in \text{right subtree}(x) \quad (1)$$

Under the assumption of distinct key:

$$\text{key}(z) < \text{key}(x) < \text{key}(y), \forall z \in \text{left subtree}(x), \forall y \in \text{right subtree}(x) \quad (2)$$

This means that any node in an ordered binary tree should be no greater than any nodes in its right subtree, nor should it be smaller than any nodes in its left subtree.

Remark 1: Notice that there can be different ways to arrange the nodes, depending on the structure of the tree. The largest spanning for a tree is a chain, where each node can have at most one child.

Remark 2: Since the structure of a tree implies the relations of its constituent nodes, some of the operations (e.g. Insert or Delete) may change structure of a tree while others do not.

2.2 Sorting

The following recursive algorithm returns the nodes in a sorted order:

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

The function calls itself, hence the name "recursive".

Remark 3: If we want to output the nodes in a decreasing manner, we can switch the order of searching left and right subtrees.

2.3 Search

Given the pointer to the root node and the key k we would like to search for, the search function returns the pointer to our desired node or returns NULL when k is not contained in the tree. Below is a pseudocode for the algorithm:

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

2.4 Insertion

Notice that under the tree property, insertion only happens at the **leaf node**(which means a node with no children); In light of this, insertion always happens at the bottom of a tree and this is sometimes not the most efficient way. Below is the pseudocode for an algorithm that implements the insertion procedure:

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

Remark 4: The use of y in the pseudocode is to designate the parent of z, i.e. the node we want to insert in order to initialize pointers between parent and child nodes. Lines $9-10$ of the pseudocode correspond to the case where we have no parents, i.e. the tree is NULL before insertion.

3 Complexity of Operations on Binary Trees

3.1 Depth of a Binary Tree

The **depth** of a binary tree is the number of nodes on the longest path between the root and a leaf.

For the same number of nodes, a **chain** is the most inefficient tree (with largest depth), while a **balanced tree** is the most efficient tree (with smallest depth). For a balanced tree with depth d, it can contain $[2^{d-1}, 2^d - 1]$ nodes. Therefore, the depth of a binary tree with n nodes is smaller than n (chain), and

larger than $log_2(n+1)$ (balanced tree). Also, you can always try to rebalance a tree.

3.2 Complexity of Sorting

InOrderTreeWalk will print n times and call 2n times. Therefore, the total number of operations for it is 3n. This number doesn't depend on the structure of the tree.

3.3 Complexity of Search

The total number of operations is always smaller than the depth of the tree. Therefore, this number does depend on the structure of the tree. For the "average" case, $d = O_p(log_2 n)$, so is the number of operations for search.