Even Happier Trees

CSE 122 ~ Algorithms & Data Structures

Binary Tree

- Definition
  - Each node in the tree has at most two children
  - Every node except the root is designated as either a left child or a right child of its parents
- Left and right subtrees are pointed to by pointers
- Node definition

```
typedef char * T;
typedef struct node{
    T key;
    struct node *left;
    struct node *right;
}Node;

Typedef Node *Tree; // pointer to a Node
```

Traversal of Binary Trees

- Three principle ways
  - Preorder
  - Inorder
  - Postorder
- Traversal is defined recursively
- Trivial case: binary tree is empty ~ do nothing
- Otherwise traversal is a three step procedure

Preorder Traversal

- ➔ Visit the root
- ➔ Traverse the left subtree
- ➔ Traverse the right subtree

Inorder Traversal

- ➔ Traverse the left subtree
- ➔ Visit the root
- ➔ Traverse the right subtree

Postorder Traversal

- ➔ Traverse the left subtree
- ➔ Traverse the right subtree
- ➔ Visit the root

Stack-Based Implementations of Traversal

- In recursive definition the call stack remembers the current node so after completing the left subtree can go to the right subtree

Stack-based Pre-order traversal

- Process the current nod
- Before going left store the current node on the stack
- After completing the left subtree processing, pop the stack and go to its right subtree
- Continue until the stack is empty

Stack-based Inorder Traversal

➔ Similar to preorder, except:

- After popping process current node
- Left node and current node processed
- Process right subtree

Stack-based Postorder Traversal

- In preorder and inorder you did not need to visit the same vertex again. But in postorder, each node is visited twice.
- This means after visiting the left subtree we visit the current node and also after processing the right subtree we will visit the current node.
- Need to process the current node **after** we visit the right subtree

Stack-based Postorder Traversal

- Problem: how can you tell if you are returning from a left or right subtree?

Stack-based Postorder Traversal

- Problem: how can you tell if you are returning from a left or right subtree?
- Trick: after you pop an element from the stack, check whether that element and the right of the top of the stack are the same.
  - If they are the same you are done processing the right and left subtree. In this case, pop the stack and print out the data

Level Order Traversal

➔ Level order traversal is defined as:

- Visit the root
- While traversing level L, keep all elements at level L + 1 in queue
- Go to the next level and visit all the nodes at that level
- Repeat until all levels are visited.

Binary Search Tree

- ➔ Important ~ Not all binary trees are binary search trees!
- ➔ Properties of BST:
  - Each node contains contains a special value called a *key* that defines the order of the nodes. Nodes may contain other data as well.
  - Key values are *unique*, in the sense no key can appear more than once in the tree
  - At every node in the tree, the key value must be greater than all keys in the subtree rooted at its left child and less than all the keys in the subtree rooted at its right child

Binary Search Tree ~ Example

- Assume you have the following linked list:
  - Bashful -> Doc -> Dopey -> Grumpy -> Happy -> Sleepy -> Sneezy
- Can't apply binary search algorithm to this list because you have to *search* the list for the **middle**
- So, let's keep track of the middle of the list
  - Grumpy
- Now, apply the same idea to the sub-lists
  - Bashful -> Doc -> Dopey : middle is Doc
  - Happy -> Sleepy -> Sneezy : middle is Sleepy
- Draw the resulting tree

Creating a Binary Search Tree

- At each node, insert compares the new key to the key in the current node
- If the key precedes the existing one, the new key belongs in the left subtree
- If the new key is greater than the current key, it belongs in the right subtree
- Eventually will encounter a NULL subtree ~ that is where the new node needs to be added

Binary Tree Insertions: To Note

- A few things to note about binary search trees
  - The order you insert nodes matters! Can insert so it becomes a linked list.
  - Tree becomes unbalanced unless you are careful how you insert

Deletions from a Binary Search Tree

- ➔ Deleting a node with no children
  - Straight forward ~ find the node and delete
- ➔ Deleting a node with one child
  - Node's child needs to be the child of the node's parent
  - i.e. replace the node with its child
  - If the node is a left child:
    - The node's child becomes the left child of the node's parent
  - If the node is the right child:
    - The node's child becomes the right child of the node's parent

Deleting a node with Two Children

➔ Replace the node's value with its
- in-order predecessor (rightmost child of the left subtree)
- In-order successor (leftmost child of the right subtree)

Other useful tree operations

- Determine the height of a tree
- Determine the number of nodes
- Determine the number of internal nodes
- Determine the external nodes
- Get a mirror image of the tree
- Remove a tree
- Find the smallest node
- Find the largest node