CS 261 – Data Structures

Trees
Binary Tree

• Nodes have no more than two children:
  – Children are generally referred to as “left” and “right”

• Complete Binary Tree:
  except for the bottom level, filled from left to right
Binary Tree

• Full Binary Tree: every leaf is at the same depth
  – Every internal node has 2 children
  – Height of $n$ will have $2^{n+1} - 1$ nodes
  – Height of $n$ will have $2^n$ leaves
Complete Tree: **Height and Node Count**

- What is the height of a complete binary tree that has $n$ nodes?
Dynamic Array Implementation

- Children of node $i$ are stored at $2i + 1$ and $2i + 2$
- Parent of node $i$ is at $\text{floor}((i - 1) / 2)$
Dynamic Array Implementation

- Children of node \( i \) are stored at \( 2i + 1 \) and \( 2i + 2 \)
- Parent of node \( i \) is at \( \text{floor}((i - 1) / 2) \)

Why is this a bad idea if tree is not complete?
Dynamic Array Implementation (cont.)

If the tree is not complete

(it is thin, unbalanced, etc.),

the **DynArr** implementation

will be full of holes

Big gaps where the level is not filled!
Dynamic Memory Implementation

```c
struct Node {
    TYPE val;
    struct Node *left; /* Left child. */
    struct Node *right; /* Right child. */
};
```

Like the **Link** structure in a linked list
Binary Tree Traversals

• How to examine every node in a tree

• A list is a simple linear structure: can be traversed either forward or backward – but usually forward

• What order do we visit nodes in a tree?

• Most common traversal orders:
  – Pre-order
  – In-order
  – Post-order
Binary Tree Traversals (cont.)

• All traversal algorithms have to:
  – Process node
  – Process left subtree
  – Process right subtree

Traversal order is determined by the order these operations are done
Six possible traversal orders:

1. Node, left, right \(\rightarrow\) Pre-order
2. Left, node, right \(\rightarrow\) In-order
3. Left, right, node \(\rightarrow\) Post-order

4. Node, right, left
5. Right, node, left
6. Right, left, node

Most common

Subtrees are *not* usually analyzed from right to left.
Pre-Order Traversal

- Process order → Node, Left subtree, Right subtree

```c
void preorder(struct Node *node) {
    if (node != 0) {
        process (node->val);
        preorder(node->left);
        preorder(node->right);
    }
}
```

Example result: p s a m a e l r t e e
Post-Order Traversal

• Process order → Left subtree, Right subtree, Node

```c
void postorder(struct Node *node) {
    if (node != 0){
        postorder(node->left);
        postorder(node->right);
        process(node->val);
    }
}
```

Example result: a a m s l t e e r e p
In-Order Traversal

• Process order → Left subtree, Node, Right subtree

```c
void inorder(struct Node *node) {
    if (node != 0){
        inorder(node->left);
        process(node->val);
        inorder(node->right);
    }
}
```

Example result: a sample tree
Traversals

• Computational complexity:
  – Each traversal requires constant work at each node (not including recursive calls)
  – Order not important
  – Iterating over all $n$ elements in a tree requires $O(n)$ time
Traversals

• Problems with traversal code:
  – If internal: ties (task dependent) process method to tree implementation
  – If external: exposes internal structure (access to nodes) → Not good information hiding
  – Recursive function can’t return single element at a time
  – Solution → Iterator (more on this later)
Binary Search Trees
Binary Search Tree

• Binary search trees are binary trees where every node’s object value is:
  – Greater than all its descendants in the left subtree
  – Less than or equal to all its descendants in the right subtree
Binary Search Tree

• In-order traversal returns elements in sorted order

• If tree is reasonably full (well balanced), searching for an element is $O(\log n)$
Binary Search Tree: Example

- Alex
  - Abner
    - Abigail
    - Adam
  - Adela
    - Agnes
  - Angela
    - Alice
    - Allen
    - Audrey
  - Audrey
    - Arthur
Bag Implementation: **Contains**

- Start at root

- At each node, compare value to node value:
  - Return true if match
  - If value is $<$ node value, go to left child; Repeat
  - If value is $>$ node value, go to right child; Repeat
  - If node is null, return false
Bag Implementation: **Contains**

- Traverses a path from the root to the leaf
- Therefore, if tree is *reasonably full* *(an important if)*, execution time is \( O(??) \)
BST: Contains/Find Example

Object to find → Agnes
Bag Implementation: **Add**

- Do the same type of traversal from root to leaf
- When you find a null value, create a new node
BST: Add Example

After first call to `add`
Bag Implementation: **Remove**

- Remove is most complex ➞ Leaves a “hole”
- What value should fill the hole?
BST: Remove

How would you remove Abigail? Audrey? Angela?
Who fills the hole?

- Answer: **the leftmost child of the right child** (smallest element in right subtree)

- Try this on a few values
BST: Remove Example

Before call to \texttt{remove}
BST: Remove Example

After call to remove
Special Case

• What if you don’t have a right child?

• Try removing “Audrey”
  – Think about it
  – Can just return left child
Complexity

• Running down a path from root to leaf

• What is the complexity? $O(??)$