COSC160: Data Structures
Binary Trees

Jeremy Bolton, PhD

Assistant Teaching Professor

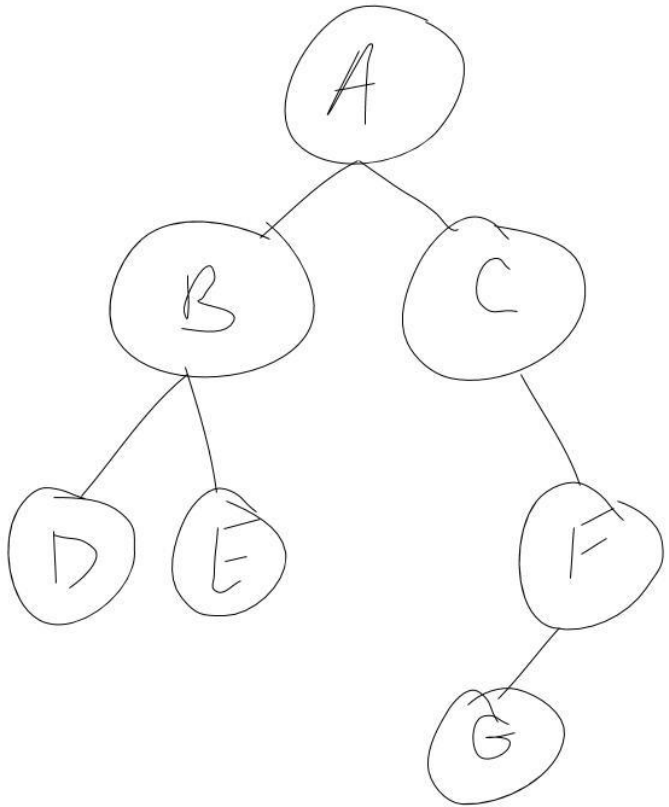
GEORGETOWN UNIVERSITY

Outline

I. Binary Trees
   I. Implementations
      I. Memory Management
II. Binary Search Tree
   I. Operations

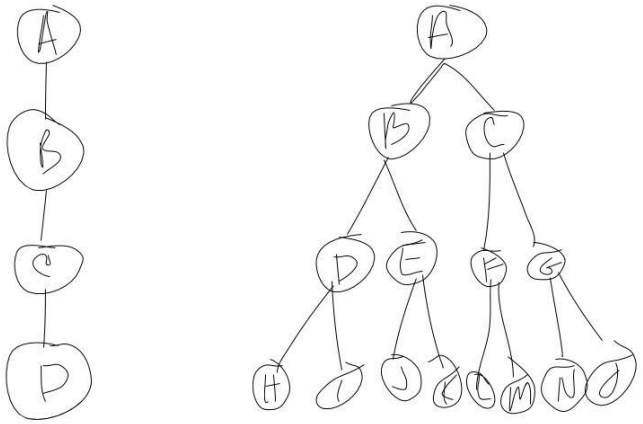
Binary Trees

- A binary tree is a tree where every node has at most 2 children.
  - By standard: leftChild and rightChild



Binary Trees

- Observations: number of children constrained
  - Mitigates allocation issues related to unconstrained numbers of children
    - Permits “simple” implementations
  - The maximum number of nodes is of a binary tree with height h is constrained: $maxNumNodes = \sum_{\{depth=0\}}^{h} 2^{depth} = 2^{h+1} - 1$
    - This constraint is not as helpful as one might think; if fact, it would be much more relevant if we could constrain the height of a tree in terms of the height of its subtrees. ***More on this later!***
    - Further note: given tree height h. the maximum number of nodes is $2^{h+1} - 1$ the minimum number of nodes is h+1.



DF Traversal of Binary Tree

- Traversals are similar to unconstrained trees, but only children.

```
// iterative
function DFS(root)
    stack.push(root)
    while( stack is not empty )
        thisNode := stack.pop( )
        //Process thisNode here (preorder)
        stack.push(rightChild)
        stack.push(leftChild)
```

```
function DFS_preorder(root)
        //Process root here (preorder)
        if(root is Null),return
        DFS(leftChild)
        DFS(rightChild)
```

```
function DFS_inorder(root)
        if(root is Null),return
        DFS(leftChild)
        //Process root here (inorder)
        DFS(rightChild)
```

```
function DFS_postorder(root)
        if(root is Null),return
        DFS(leftChild)
        DFS(rightChild)
        //Process root here (post order)
```

GEORGETOWN UNIVERSITY

BF (level-order) Traversal of Binary Tree

- Traversals are similar to unconstrained trees, but only children.

```
function BFS(root)
        queue.add(root)
        while( queue is not empty )
                thisNode := queue.dequeue(  )
                //Process thisNode here
                if leftChild is not null, queue.add(leftChild)
                if rightChild is not null, queue.add(rightChild)
```

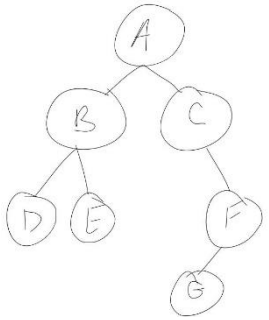
```
//recursive (not practical)
function BFS(root )
        queue := new queue
        queue.add(root)
        BFS_helper(queue)

function BFS_helper(queue)
        thisNode := queue.dequeue(  )
        //Process thisNode here
        if c is not null, then queue.add(leftChild)
        if c is not null, then queue.add(rightChild)
        BFS(queue)
```

GEORGETOWN UNIVERSITY

Implementation of Binary Tree

- Chaining:
  - It is intuitive to implement a binary tree using a node entity that has attributes: data, leftChild, rightChild.

- Array implementation
  - If the *height of the tree is known*, the maximum number of nodes is known and thus we can allocate contiguous space in memory (an array). We can index each node in the binary given a *breadth first* scan of the tree, and use this index to store each node into an array.
    - Not efficient if the number of nodes is not near maximum
    - If the height of the tree can change dynamically, a dynamic array implementation would be necessary (and possibly inefficient).
    - Exercise:
      - Implement Breadth First Traversal
      - Implement Depth First Traversal



Operations on a Binary Tree

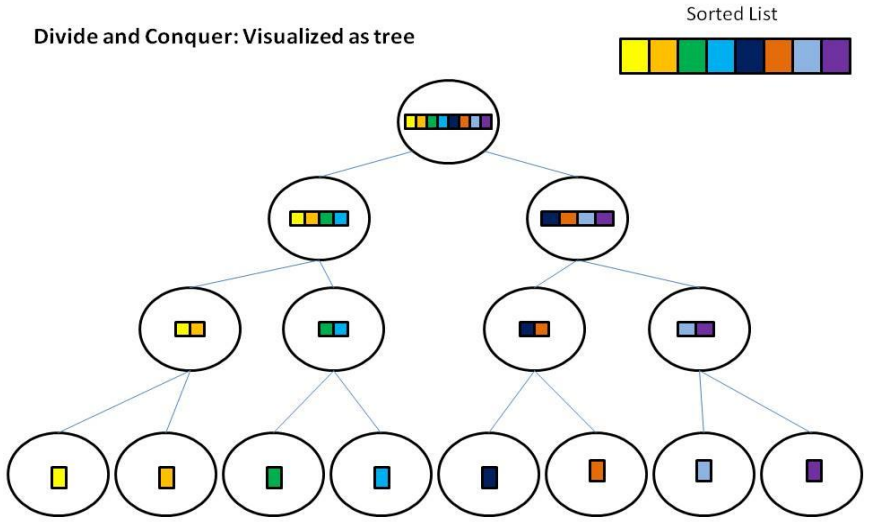
- Common Operations – will depend on application
  - Make copy or delete (discussed during generic trees)
  - Determine height
  - Count number of elements
  - Insert item
  - Search for item (traversals)

- **Important: for all data structures, it is important to identify the "valid" state of the structure and maintain that state through each operation.**

- Application example:
  - Binary search tree

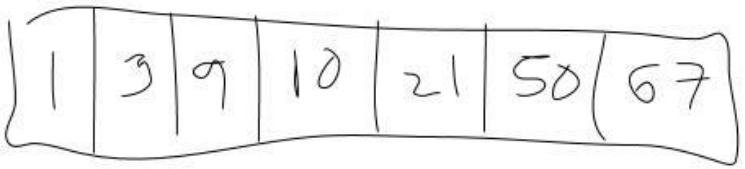
Example: Binary Search Tree

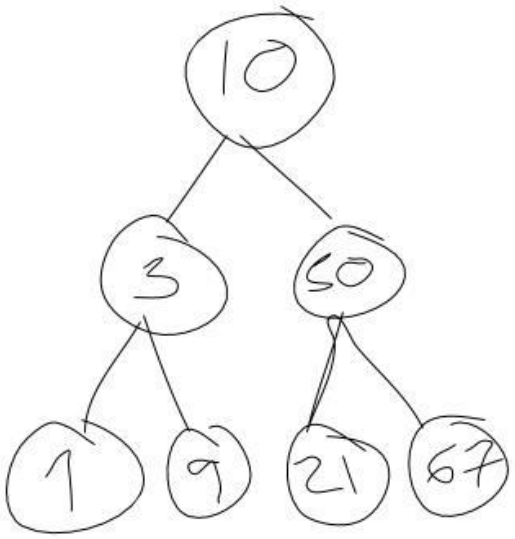
- When searching for data in a list
  - Performing a sequential search yielded a worst case complexity of O(n)
  - Note: when employing binary search we were able to achieve a significant reduction O(log n)
    - REMEMBER: This was achieved by employing a divide and conquer scheme – this process can be visualized as a decision tree
    - We can achieve a similar result when explicitly using a tree structure as well! (in some cases)



Binary Search Tree: Binary Search using a Tree

- The data stored in each node is referred to as the key.
- Binary Search Tree is a binary tree where, for each node n all the keys in the left subtree are less than the key of n, and all keys in the right subtree are greater than the key at n.
- **Note** that we get a *similar traversal by using an explicit BST or a sorted array using binary search*



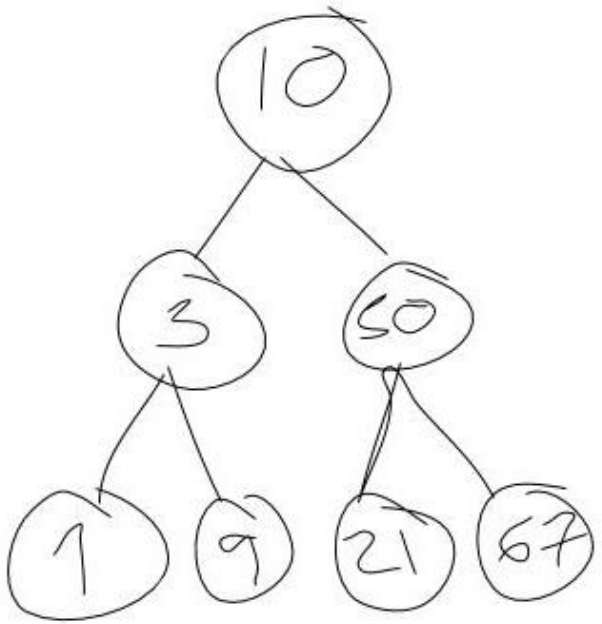


GEORGETOWN UNIVERSITY

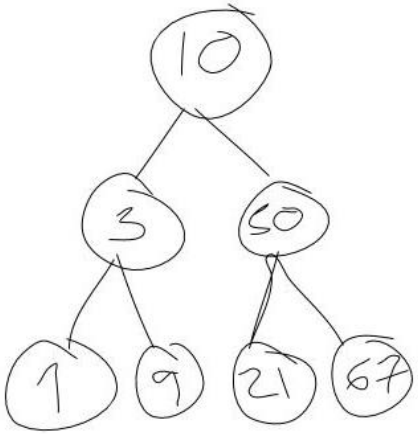
Binary Search Tree

- Search Binary Search Tree for key i.
  - How should we traverse the tree: DF-like or BF-like?
    - Note, there is no need to “backtrack” so we do not need a stack

```
function search(root, i)
        thisNode := root
        while( thisNode != NULL)
                if(i > thisNode.key) // go to right subtree
                        thisNode := thisNode.rightChild
                elseif(i < thisNode.key)  // go to left subtree
                        thisNode := thisNode.leftChild
                else        // i == thisNode.key
                        return thisNode
        return NULL; // not found
```



BST: Insert



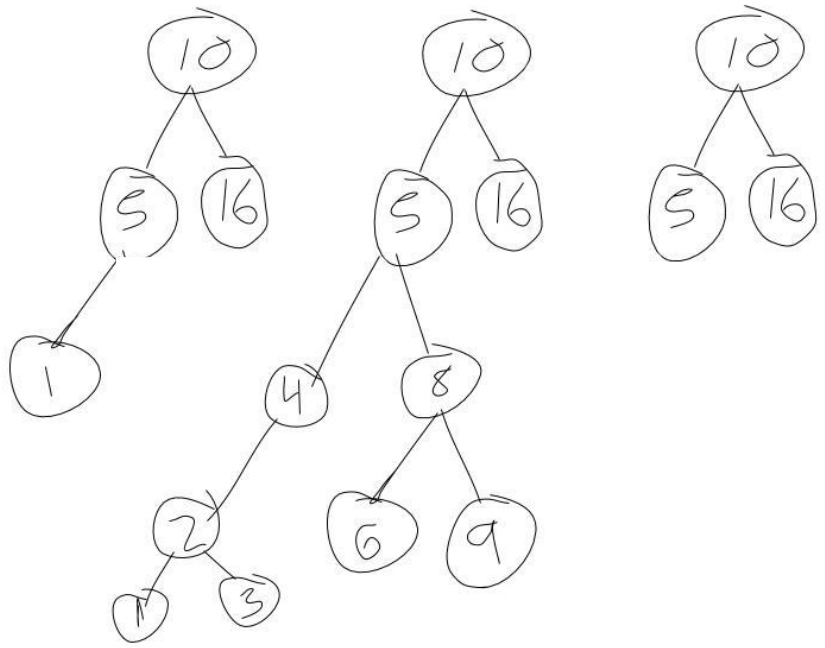

```
function insert(root, i)
        thisNode := root
        while( thisNode != NULL)
                if(i > thisNode.key) // go to right subtree
                        if rightChild not null, thisNode := thisNode.rightChild
                        else, thisNode.rightChild := new Node(i)
                elseif(i < thisNode.key)        // go to left subtree
                        if leftChild not null, thisNode := thisNode.leftChild
                        else, thisNode.leftChild := new Node(i)
                else        // assume unique items: do not add
```

- Worse Case Time Complexity: O(h), where h is the height of the tree.
  - … but what is this in terms of the number of nodes n?
  - This depends on the structure of the tree. (More discussion later.)

BST: Remove

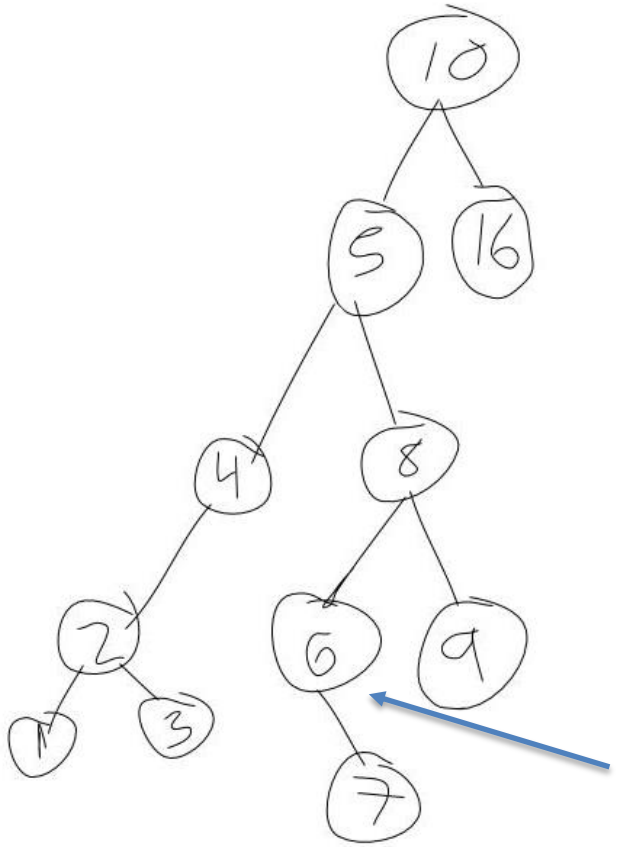
- Removing Nodes from a BST is a bit more difficult. We must assure the resulting structure a valid BST.
  - Removing a node may create a “disconnected” tree – not valid!
  - The node we remove may have 0,1 or 2 children.
  - Cases 0 and 1 are fairly simple.
    - Case 0: deletion requires no further action
    - Case 1: connect child to deleted nodes parent
    - Case 2: ?
- Time Complexity: O(h), where h is the height of the tree. In terms of the size of the tree (the number of nodes in the tree), what is the worst case complexity? What is the best case complexity?

Example: Remove(5).
Note there are 3 cases.



BST: Remove

- ***Example Remove 5.***
  - Note this is case 2. Removed node has two children
  - Which node should replace 5?
    - Max value is left subtree or min value in right subtree.
    - Note we cannot simply remove the min value from the right subtree as this may also create a disconnect. How can we create a general solution.
      - **Observe**: if node m is the min value in the right subtree, it will not have a left child (removal of this item is simple: case 1)
      - First, we define a method to remove the min value from a tree (below), then we can define a method to remove any node from a tree



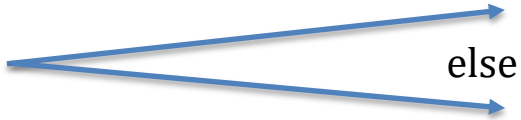

```
function removeMin(parent, node)
        // removes node, but does not delete, returns ptr instead;
        if( node! = NULL)
                if(node.leftChild ! = NULL) // go to leftmost child in right subtree
                        return removeMin(node, node.leftChild)
                else            // min val
                        parent.leftChild = node.rightChild // removes root from tree (case 1)
                        return node
        else // subtree is empty: incorrect use of function
        return NULL
```

BST: Remove

```
function remove( &root, item)
    if( root! = NULL)
        if (root.key < item ) // go to right subtree
            remove(root.rightChild, item)
        else if(root.key > item )        // go to left subtree
            remove(root.left, item)
        else if (root.leftChild ! = NULL  &&root.rightChild ! = NULL)
            // node found – two children (case 2)
            // remove min of right subtree and copy its value into root
            Node minRightSubtree := removeMin(root, root.rightChild)
            root.key := minRightSubtree.key // copy other data as needed
            delete minRightSubtree
        else  // node to remove only has 0 or 1 child
            trash := root
            if (root.leftChild ! = NULL)
                root := root.leftChild // copy other data as needed
            else
                root := root.rightChild // copy other data as needed
            delete trash
    else // subtree is empty: item not found?
```

- Note:
  - Must account for all cases. Uses helper function removeMin to find min value in right subtree.
- Time Complexity?
  - How many recursive function calls (iterations) will occur in the
    - Best case?
    - Worst case?

- Note
  - Root is passed in by reference



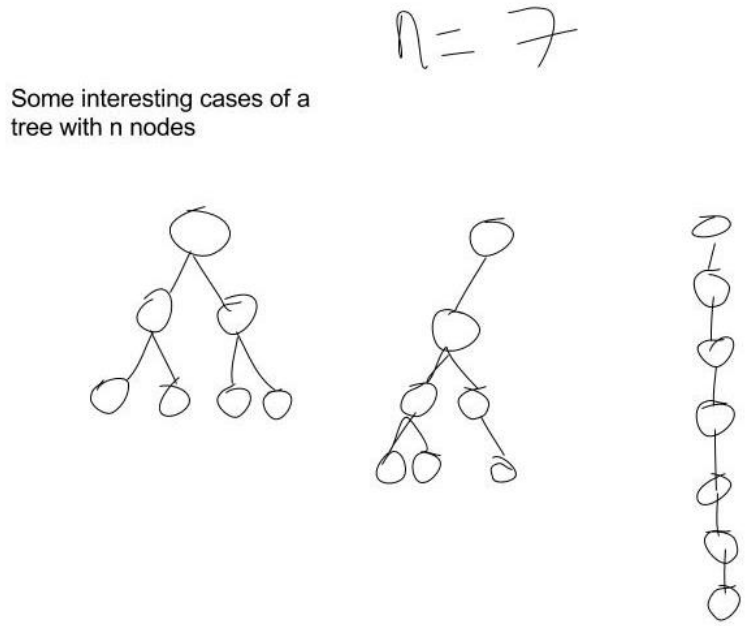
GEORGETOWN UNIVERSITY

BST: Complexity Analysis

- For insert and remove, where n = number of nodes:
  - Best case: O(1)
    - Item searched for is at root
  - Worst case: O(n)
    - Worst Case: Item is at leaf node.
    - Different worst cases for different tree structures
      - If n = h. Worst case O(n)
      - If n = log(h). Worst case is O(log n)

- Observe the worst case is still O(n) which is not necessarily an improvement over using a standard list. In fact, the worst case binary tree is a list.

- Is complexity reduced on average when using a tree? What is the average case complexity?
  - For a tree with n nodes, there are many different cases that will affect search complexity. These different cases correspond to the *different possible tree structures given n nodes*. We can attempt to compute the approx. computation steps for all cases and take the average … but how?



BST search: Average Case with proof

- Given a tree structure, How do we compute the average search time?
  - Two major factors that result in different step counts:
    1. Structure of tree
    2. Depth of searched item in tree
  - Assume T_n is the set of all possible tree structures with n nodes and $t \in T_n$ is one such structure.
  - Assume N is the set of all nodes in the tree.
    - Average step count for tree structure t with n nodes

$$averageStepCount_t = \frac{\sum_{n_i \in N} stepCount_t(n_i)}{n}$$

Using IPL to Compute Average Case

- Average step count (assuming each tree structure is equally likely) is

$$averageStepCount = \frac{\sum_{t \in T_n} averageStepCount_t}{|T_n|}$$

- However computing this value for all possible tree structures iteratively, may be tedious. If we can define this in terms of a recurrence for any tree structure, we can simplify the computation.

- To simplify the computation, we define the **Internal Path Length (IPL)** of a binary tree t with n nodes:
  - Conceptually this is simply the sum of the depths of all of its nodes, which is proportional to step count for a search operation.
  - Observe, the IPL can be computed given a tree structure t

$$averageIPL = \frac{\sum_{t \in T_n} IPL_t}{|T_n|}$$

GEORGETOWN UNIVERSITY

Define recurrence to compute averageIPL

- Determine a recurrence
  - Define function D(n) to be the *averageIPL* for a tree of n nodes.
    - An n-node tree has a root and two subtrees: one with i nodes (left) and one (right) with n - i -1 nodes.
    - Here we use i to constrain the tree structure at each stage of the recurrence. And we assume each tree structure is equally likely, and thus each i chosen is equally likely
      - For a given i, D(i) is the average internal path length for the left subtree (for a tree of i nodes).
      - Big Picture: If we average over i, we effectively average over tree configurations
  - $D(n) = n - 1 + D(i) + D(n - i - 1)$
    - The root contributes 1 to the path length of each of the remaining n-1 nodes (for a total of n-1).
  - Average over all possible i, $0 \leq i \leq n - 1$ (over subtree configurations)
    - $D(n) = n - 1 + 2 * \frac{D(0)+D(1)+D(2)+\cdots+D(n-1)}{n} = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} D(i)$
  - Note: This recurrence is difficult to evaluate since D(n) is in terms of all of ALL of its predecessors in the recurrence.

GEORGETOWN UNIVERSITY

Solving averageIPL recurrence

$$nD(n) = n(n-1) + 2 * \frac{D(0)+D(1)+D(2)+\cdots+D(n-1)}{1}$$

$$(n-1)D(n-1) = (n-1)(n-2) + 2 * \frac{D(0)+D(1)+D(2)+\cdots+D(n-2)}{1}$$

Recurrence is difficult to compute since the n^{th} term contains ALL previous terms

Try to manipulate equation to solve for 1^{st} order recurrence (only 1 previous term).

Start by multiplying both sides by n.

Next, derive equation for $(n-1)^{st}$ term and subtract. (common "trick")

Dividing both side by n(n+1) , by experience, we know this will simplify the next step by clearly flushing out a harmonic series

Subtracting equations we arrive at the following equation:

$$nD(n) - (n-1)D(n-1) = 2n + 2 + 2 * D(n-1)$$

$nD(n) = 2n + (n+1)D(n-1)$, rearranging terms, and ignoring constants

$$\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2}{n+1}$$

Telescoping these terms we have

$$\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2}{n+1}$$
$$\frac{D(n-1)}{n} = \frac{D(n-2)}{n-1} + \frac{2}{n}$$
$$\frac{D(n-2)}{n-1} = \frac{D(n-3)}{n-2} + \frac{2}{n-1}$$
$$\cdots$$
$$\frac{D(2)}{3} = \frac{D(1)}{2} + \frac{2}{3}$$
$$\frac{D(1)}{2} = \frac{D(0)}{1} + \frac{2}{2}$$

D(0) = 0

Next, Solve recurrence by summing telescoping terms. Observe there are many cancellations and we get a closed form for D(n).

GEORGETOWN UNIVERSITY

BST search: Average case using averageIPL recurrence

We can now compute the value of each term in the recurrence, via sequential substitution. *The result is a sum including previous terms.*

$\frac{D(n)}{n+1} = \frac{D(1)}{2} + 2 * (\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1})$, we add and subtract (2) to the RHS to rearrange the terms in the sum to match a harmonic series

$\frac{D(n)}{n+1} = 1 \ + 2 * \left(1 + \frac{1}{2} + \ \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) - 2$

$\frac{D(n)}{n+1} = 2 * \left(1 + \frac{1}{2} + \ \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) - 1$, using harmonic series inequality we have

$\frac{D(n)}{n+1} = 2 * \left(1 + \frac{1}{2} + \ \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n+1}\right) - 1 \leq 2 * \log(n) + 1$

$\mathrm{D(n)} \leq (n+1)(2\log(n) + 1)$

Thus D(n) is $\mathrm{O}(n\ log(n))$

Use Fact: Harmonic Series

$$\sum_{i=1}^{n} \frac{1}{i} \leq \log(n) + 1$$

GEORGETOWN UNIVERSITY

BST search: Average Case

- Final Step:
  - Thus D(n) is $O(n \log(n))$
  - D(n) is the sum of the depths for all nodes. Thus the average depth (proportional to the average search/traversal time) for a tree with n nodes is $O(n \log(n))$/n which is $O(\log(n))$

Why Use Binary Trees?

- What is gained with the use of a tree?
  - The average case for standard operations is O(log n)
  - BUT the worst case is still O(n)

- Can we do better?
  - Yes! The worst case and others like it are O(n). Why? Because the height of the tree is approximately equal to n, the number of nodes (the tree is list-like).
  - If we add constraints that limit the depth of the tree we can eliminate these cases.

- Balanced Trees …