TREES

Day 1

Goals: Introduce general idea of trees. Show what they are useful for. Terminology. Introduce binary trees. applications. Terminology. Show recursive def of Bintree. Show array-based BT, linked BT. Traversals (in, pre, postorder). Show applications of these traversals. [No code for any of these, only ideas or high level pseudocode]

INTRO

* So far all of our structures have been linear.
* Recap the structures we’ve seen:
  + ArrayList ADT (implemented with an array)
  + List ADT (implemented with many types of linked lists)
  + Stack/Queue ADT (implemented either as arrays or linked lists)
* But all of these data structures have an inherent linearity to them: for each element, there’s always one element that comes “before” and one element that comes “after.” Before and after mean different things in these different ADTs, but the point is in all of them they can all be drawn as linear structures.
* Today we’re going to begin exploring structures that have an inherent hierarchical nature to them.
* What are some common hierarchical structures we encounter in the real world that we might want to store in a computer?
  + Directory structure of a filesystem
  + URLs
  + Class/object hierarchies (inheritance)
  + Tables of contents for a book.
  + Outlines of documents.
  + Organization charts for companies (who reports to whom).
  + Family tree (genealogy)
  + Military structures
  + Political structures
  + Biological taxonomy (kingdom phylum class order family genus species)
  + Address structures Countries/states/zipcodes or cities/streets/houses
  + Maslow’s hierarchy of human needs
  + Music (composition -> movement -> section/phrase -> measure -> note)
  + sentence diagrams
* Today we’re going to explore how to represent these kind of concepts, where each element in the hierarchy may have a set of items both superior to it (higher) and inferior (lower) than it in the hierarchy.
* In CS, a common way to represent a hierarchy is with a data structure called a TREE.
* Terminology:
  + Trees have a collection of elements that are called NODES.
  + Each node may have zero or more CHILD nodes. This creates a hierarchical structure.
  + The reason we call these structures TREES is from the tree-like diagrams that we use to draw them.
  + SHOW SLIDES
  + Draw this tree:
    - Root 1
      * Children 2 and 3
        + 2 has child 4, and 4 has child 8 & 9
        + 3 has child 5,6,7
  + As a convention, we draw trees in CS so that they grow DOWNWARD, rather than UPWARD.
  + So within that convention, every tree that is non-empty has a special NODE called the ROOT, that is USUALLY drawn at the top.
    - The root never has a PARENT. (no arrows going into it, only going out).
    - Every other node has EXACTLY one PARENT.
    - Every node always has exactly one PATH from the root to that NODE.
  + Nodes that have no CHILDREN are called **LEAVES**. They are necessarily drawn at the BOTTOM, though they might not all be at the same level of the hierarchy.
* **Internal node**
* A node with at least one child.
* **Degree**
* For a given node, its number of children. A leaf is necessarily degree zero.

  + We use a fair amount of genealogical terminology:  
    - AS we said, every node can have zero or more children.
      * In most trees, the ORDER of the children matters. So 5,6,7 is not the same tree as 5,7,6.
      * Often times arrows are left out to show which is parent and which is chid.
    - Nodes with a common parent are called siblings.
    - The **descendants** of a node are the nodes’ children, plus children’s children, plus children children children...etc.
    - The **ancestors** of a node are parent, parent’s parent, etc..
* **Level of a node**
* The level of a node is defined as: 1 + the number of edges between the node and the root.
* **Depth of a node**
* The depth of a node is defined as: the number of edges between the node and the root.
* **Height of node**
* The height of a node is the number of edges on the longest path between that node and a leaf.
* **Height of tree**
* The height of a tree is the height of its root node.
  + - Trees have a naturally recursive structure. For each node in a tree, it has a number of subtrees, which is formed by selecting the structure rooted at any of its children.
* For most of this course, we will restrict ourselves to BINARY TREES, which are trees that have at most two children.
* We call these the left child and the right child, and the subtrees rooted there are called the LEFT subtree and right subtree.
* Binary trees correspond to lots of real world concepts:
  + genealogical tree showing parents, grandparents, etc.
  + Decision tree – each node contains a yes/no question that leads to appropriate subquestions.
    - Used in some of the earliest AI systems .e.g, for medical diagnosis.
  + ASK: Where do you see binary trees EVERY semester? The course picking system.
  + operator tree.
* Linked representation
  + show Node class: consists of data field + Nodes to left & right.
  + Draw example binary tree (expression tree): (3 \* 4) + (6 / 2)
  + Show recursive structure of a binary tree. [in pictures and def'n]
  + A binary tree is either
    - nothing at all (null)
    - or, a binary tree node, with pointers to two sub-binary-trees.
* The code that we write to work with binary trees is often recursive.
  + The reason is because we no longer have a linear structure, so iteration with a for loop or while loop is difficult.
  + In a binary tree, each node has two sub-pieces to process, and each of those pieces may have an additional two pieces, etc. This isn't true for arraylists or linked lists, where each node only has ONE successor.
* First illustrate recursion with review of recursion through processing linked lists recursively.
* Say we have LLNode { int data, LLNode next; }
* we have a function :
  + void func(LLNode n) {
    - if (n != null)
      * print(n.data)
      * func(n.next)
* What if I give this a LL 1->2->3?
  + What does it print? [1 2 3]
* Draw recursion tree.
* Now reverse the order of the print/call.
  + What does it print now? [3 2 1]
  + Draw
* FIRST DO RECURSIVE LL -   
  THEN DO RECURSIVE TREE TRAVERSALS  
  Traversals with parse/expression tree.
* Review traversals:
  + Preorder\_traverse(node):
  + IF NOT NULL:
    - Visit(node.data)
    - Preorder(node.left)
    - Preorder(node.right)
  + Write algs for inorder, postorder.
* traversals with linked list.
* Write function to add up elements of a linked list recursively.
* Write function to add up elements of a tree recursively.
* Write function to evaluate expression tree recursively.
* operator tree
* linked list traversals.
* If time, do binary search trees
  + each value in left tree is <
  + each value in right tree is >

**DAY 2**

* Lightning review:
  + New data structure called TREES. Trees have nodes/vertices, and edges that connect the nodes.
  + Biggest terminology: Draw trees like family trees. So children go below, parents go above. But every node in a tree has only one parent, except the root of the tree, which has ZERO parents.
  + BINARY TREES: Every node has 0, 1, or 2, children.
* Restart **binary search trees**.
* Definition: All the values in the left subtree are < (or <=).
  + and all values in the right subtree >
* Review some examples of BSTs:
  + This is a bst: 50  
     25 75

22 47 70 88  
 44 49 99->100

* Review traversals:
  + Preorder\_traverse(node):
  + IF NOT NULL:
    - Preorder(node.left)
    - Visit(node.data)
    - Preorder(node.right)
  + Write algs for inorder, postorder.
* Trace through explicitly one of these with a small tree🡪 25  
   10 30

20 40

15 22

* Show “triangle shrink wrap” algorithm for these traversals.
* Show operator tree traversals: (shows that these traversals are useful outside of BSTs).
  + Give
* INTRODUCE SET ADT.
  + So what are binary search trees good for?
  + they are a good data structure for implementing two ADTs that we’re going to examine over the next few days.
  + The first one is called the set ADT.
    - Structure: A set consists of a collection of elements with no associated ordering. A set may not contain two identical elements. (Abstraction of a mathematical set).
    - Operations: add. Remove. Contains.
    - Optional: size, clear, union, intersect, set difference.
    - The main difference between a set and a list is that in a LIST ADT, you normally have control over where an item goes. You can put it anywhere you want. With a Set, the user doesn’t have control over it, either the Set imposes some ordering on the objects, or there is no order at all (imagine a set that contains Dog objects. Inherently, dogs have no order. You could fake an order by saying that we put dogs in order by their names, or sizes, or birthdays, or something, which does happen in the real world).
    - So when do you use a set vs a list? When you need to control the position or ordering of the items you are storing, use a list. When there is no order or you don’t care about the order, use a set.
    - Set implementations are often faster for adds/removes than list implementations, so this is a decision to think about in the real world.
* Like the LIST ADT, which can be implemented with a singly-linked list, or doubly (or circular), the SET ADT has multiple implementations as well.
* Let’s look at a few implementations.
  + Suppose we have a SET implemented with an ARRAY.
  + keep #s unsorted.
  + add = O(n) [for expanding]
  + remove = O(n) [for shifting]
  + contains = O(n)
  + SORTED ARRAY
  + add = O(n)
  + remove = O(n)
  + contains = O(log n)
  + UNOSRT LINK LIST
  + add = O(1)
  + remove = O(n)
  + contains = O(n)
  + SORTED LL
  + add = O(n)
  + remove = O(n)
  + contains = O(n)

**DAY 3**

(Last time, we ended with the Set ADT, and talked about the different implementations.)

Review this part.

Goals – introduce a set implemented as a BST. Introduce CONTAINS/INSERT/DELETE for BST.

* **SEARCH/CONTAINS** algorithm.
* First explain in “handwavy” natural way. Start at root. Check if the item at the root is the item we’re searching for. If it is, then we’re done. If the item we’re looking for is less than the root, move to the left subtree and repeat this process. If the item we’re looking for is > than the root, move to the right subtree and repeat this process. If we ever reach NULL, then the item we want isn’t in the tree.
* Now do “real version” with code.
* Review struct/class:
* struct node {  
   int data;  
   node \* left;  
   node \* right;  
  }
* class BST  
   private:   
   node \* root  
    
   public:

bool search(int item)  
bool searchAux(int \*subtreeRoot, int item)

* Start with searchAux function.
  + if subtreeroot == nullptr
    - return false
  + else if (item == subtreroot->data)
    - return true// found it
  + else if (item < subtreeroot->data)
    - return searchAux(subtreeroot->left, item)
  + else if (item > subtreeroot->data)
    - return searchAux(subtreeroot->right, item)

CAN ALSO WRITE SEARCHAUX ITERATIVELY

curr = root

while (curr != nullptr)

{

if curr->data == item return true

else if item < curr->data curr = curr->left

else if item > curr->data curr = curr->right

}

return false// not found

Why can preorder/postorder/inorder traversals not easily be written iteratively?

answer -> because they have two recursive calls. searchAux only has one.

any function with only one recursive call can usually easily be turned into an iterative function. Not so for two or more calls per original call.

* regular search function
* search(int item):
  + return searchAux(root, item)

SEARCH

Boolean search(node, searchkey)

{

if node == null

return false

else if searchKey == node.data

return true

else if searchKey < node.data

return search(node.left, searchkey)

else // searchkey > node.data

return search(node.right, searchkey)

}

Why is this log time?

Draw tree:

Illustrate that every time we add another layer of nodes, we double (roughly) the number of nodes in the tree.

Therefore, the height of the tree is logarithmic in the number of nodes.

n = number of nodes, h = height of tree 🡪 h = log2(n) [roughly] [assuming tree is balanced]

* So search is log time because every time we call ourselves recursively, we are jumping down one level of the tree, which roughly eliminates half of the nodes.
* What is the time of preorder/inorder/postorder?
* Linear.
* Why?
* Here's the big difference : TRAVERSALS MUST VISIT ALL THE NODES. Each recursive call always generates two other recursive calls.
* Search does not---search only has one recursive call.
* So when you're trying to figure out big-oh for a recursive algorithm on a tree (any tree, not just BST). Ask yourself – is each call to this function generating TWO recursive calls and therefore always going down both branches of the tree and visiting all the nodes? If yes, then O(n) [linear]
* Or, is each function generating ONE recursive call, and picking ONE branch of the tree? If so, then this is logarithmic (assuming tree is balanced).

ADD

handwavy version

Same as search algorithm, but whenever we get to null, make a new node at that location.

Implementing this is tricky, because similarly to inserting at the tail of a linked list, we actually need to hold onto the pointer BEFORE it becomes null.

Cover insert code.

REMOVE

Cover handwavy version.

Three cases:

No children.

Easy, just delete.

One child.

also easy, move child (meaning entire tree rooted at the child) to take the place of the parent node.

Two children.

Alg: Replace value to be deleted with its INORDER SUCCESSOR,

then delete the successor.

**DAY 4**

Go through remove code.

Do worksheet.

Talk about bad insertion orders—all sorted forwards or backwards. Best is a “random” order.

Summarize:

In a balanced BST, search, add, remove are all O(log n).

However, if the BST becomes very unbalanced, these can degrade to linear.

So technically, average case O(log n).

Worst case O(n).

Talk about height of a tree. Height is logarithmic in # of nodes in “perfect” case, linear in worst case.

[No] Do recursive T(n) analysis of TRAVERSAL: T(n)=2T(n/2)+1, T(1)=1, [ T(n)=nT(1)+n-1 -> big oh is linear]

and of SEARCH: T(n) = T(n/2)+1, T(1) = 1 [T(n) = T(1) + logn -> big oh is logarithmic]

Write recursive count function.

Write recursive height function. height(node) = max{ height(left), height(right) }, height(null) = 0.

If time, start expanding this from SETs into MAPs.

Suppose we are writing a system for the RAT to keep track of meal swipes.

Let’s assume everyone has an infinite number of meals, or it’s a system where you pay a flat rate for the year and come and go as many times as you want.

In that situation, we just need to store a BST of R#s. Cuz all we need to know is who is in the tree and who is not.

But suppose we need to store not only R#s, but also # of swipes left.

* Tree functions to write:
  + Height of general bintree
  + Height of a specific node in a BST
  + num nodes (size) in a bintree
  + sum of all nodes in a bintree
  + evaluate expression in expression tree
  + contains for general bintree
  + count leaves in general bintree
  + range query (greatest key that is <= searchitem)