Trees are ubiquitous in CS.

They are the most used data structure out there, so vectors/arrays, lists, and trees show up frequently. And a tree is a very important application and incredibly useful.

Trees are a collection of nodes and connected by directed arcs. A tree has a single node. By convention the root node is usually drawn at the top. A node that points to (one or more) other nodes is the parent of those node, whiule notes pointed too are the children. Every node except the root, has exactly one parent.

Nodes with no children are leaf nodes. Nodes with children are interior nodes. Nodes that have the same parent are siblings. The descendents of a node consist of its children and their children. There is some relation to family trees. And all those nodes have the same parent. They fall down.

A subtree rooted at a node is rooted in that node.

Trees have unique paths.

There is a single, unique path from the root to any node. Arcs don’t join together. A path’s length is equal to the number of arcs traversed. A node’s height is equal to maximum path length from that node to a leaf node. And we looked at the sub arcs.

If we look at the length of a path, we look at the root node, we count the number of arcs traversed.

A path’s length is equal to the number of arcs traversed, and a depth is equal to the path length from the root of that node. A depth is measure from the node down. The descendent of the root node, ABCD,

Descendent and it is a parent of A, not a true. Trees have a One To One.

Binary Trees, have a special kind of tree. A full binary tree, will have a height of h which is 2^h+1-1 nodes. Height of H will have 2^h leaves.

Complete binary Tree, full except for the bottom level which is filled from left to right. Time complexity is equivalent. The max path length is the height of the tree.

It’s important to note this for complexity as we characterize algorithms. We will come back to that later.

Complete the binary search tree has a specific structure that is efficiently store… the binary tree is here and there.

It’s stored at 2i+1, and 2i+2. Parent of node I is at floor((i-1)/2)

If the tree is not complete, the particular implementation has a problem. We have a problem, and if we created a very incomplete tree, and if it’s unbalanced, the dyn arr will be full of holes. We want trees to be complete because they need to be filled left to right. It has a ton of issues in terms of not being complete and store it the way we stored for the complete binary tree.

We gotta ask questions to do with internal questions regarding the nodes. We use it to play things like the animal game. (See slide)

Start at root. If internal node, go with cat, etc.

Computer looks at stuff. Goes down the no path. We build decision trees when we use no. Games can be used to figure out what kids are thinking and how to find something.

And we can get 1,000,000 outcomes from 20 questions.

BST – Concepts – greater than is the right subtree. Less than or equal to all descendents is the left sub tree. If tree is reasonably full, searching for it should be O(log n). And why? If we look at it, we essentially search down that tree like we look at a phone book. Unless we call it left and right by the computer perspective? Check on this.

Reminder: Binary Search Tree are binary trees, where every nodes object value is: Greater than all its descendents in the left subtree, and less than or equal to all its descendent in the right subtree. In order traversal returns elements in sorted order. And if the tree is full (well balanced), it will be O(log n).

Remember, with Binary Search Trees, we are looking at greater than, less than or equal.

Struct BST tree {

Struct Node \*root

Int cnt;

}

Struct Node {

Type Val;

Struct Node \*left;

Struct Node \*rght;

};

Valid binary search tree, and add value 12. Add pseudocode then we move it along, with tree\_5, 12.

12 > 5. If we look at the root and the node, it returns as true and the recursion will occur later.

GENERIC Structures – look at the data structure, and we can write functions that make use of #defines. Val. And in this case, the developer, it is the LT. GT, and EQ functions. It helps protect and make the ease of change – and if you do it… you gotta recompile if there is any change.

We write our structure to hold void \* pointers. And this is a generic pointer type, indicates absence of a type, any pointer can be cast to void\* and back again without loss of information

Require that the user tell us how toe ‘compare’ the elements that they are actually storing and compare works on void \* and casts them to actual types.

It pays for trees to be BALANCED. If things are roughly balanced, the operation will be logn.

If we had a million entries, balanced, will be 20 time units.

If we have an unbalanced tree, things will be O(n). Linked lists are standard linear lists. A complete binary search tree – filled at all levels, except the last. A perfectly balanced will have full levels, but that’s a moot point. Last levels are filled L->R.

If we are to take alice, and put it over on thje right it would be incomplete (see the slide show).

The longest path, is log(n). It will not be the ceiling. The height difference between the left and right subtrees is at most one. And the annotated nodes, and the leaf nodes are zero. The height of a node, is a maximum pathlink from that link to node.

Height is bottom up, depth is top down(?)

Height Balanced?

1. Is height balanced, as no nodes have a greater height.
2. Height balanced. None of the children have a height balance greater than 1, so it’s balanced.
3. If we have greater than 1, it is not balanced, but yes, that tree was.
4. Node 150 is out of balance. Absolute value is higher than 1, so it is not balanced.
5. 100 is unbalanced, and their difference is 2. So, the node at 100 is unbalanced.

All of our algorithms are logn. See the slide. Mathematically the longest path, is going to be 44% longer than log n, but this is still trivial in differenet, ans this is still a height balanced logarithm.

Height balance is far easier than completeness.

AVL. – named for GM Adelson-Velskii, EM Landis. And this maintains the height property of a balanced tree through a series of rotations. And then qwe use rotations.

When the unbalanced tree performs a rotation top balance the tree, we look at the balance. TRhe Node is highly unbalanced.

WE perform the rotations. +

AVL TREES II: We are going to discuss AVL Concepts of the rotation and balancing of the tree. Work through AVL trees, examples and we do that by looking at lots of examples and rotation, balance, and work through the AVL Tree examples.

So, let’s look at CASE 1 (see slides)

HEAP TREES:

Recall that a complete binary tree can be efficienty represented by an array or a vector.

The parent of a node at index *i* is at index floor((i-1 / 2), and the children of a node at index *I* at indices 2i+1 and 2i+2.

Root node are at 0. 2i+2, … IT MAKES FUCKING SENSE NOW.

Left child = 2i +1, 2i+2, and that’s where the node goes. Level order traversal. Left to right, and in the next two locations and the process is continued for the next level. And this is quite easy to take a heap and represent it by an array or vector.

Void addHeao(struct DynArr \*heap, TYPE val)

{

Int parent;

Int pos = sizeDynArr(heap); -- this is the size of the heap, and the underlying dynamic array. Int pos, and add the dynArr, the heap is – before we add four, the size of the heap is 11.

Twice the index plus 1, plus index 2.

Else, return – okay, we do need to have a return and it jumps us out of the function. Very good.

reMoveMin(cont) … last = sizeDynArr(heap)-1

putDynArr… read the worksheet, and we compare the root node to the smallest of its two children and we make adjustments. \_adjustheat, upto, start…);

\_adjust(heap,last,0)

If we are going to implement our priority queues. And the main purpose is to maintain a relevenat element. – ReserveSortedVector and SortedList:

Insertsions: n \* n = n^2

Removals: n \* 1 = n

Total Time: n^2 + n2 = O(n^2)

Insertions in Heap:

Insert: n\*log n

Removals n\*logN

2Nlog N = O(n log n); this is less than ideal, but it works.

Complete worksheet #33.

BuildHeap and HeapSort. We build an arbitray heap, and the values therein. How do we turn an array into a proper heap.

Array values – and there are complete values, so… yea.

All leaves in a heap, are a proper heap. Size one, and they are nodes. Proper heaps. Complete. And they adhere to the heap order property. First non-leaf node, and the array from right to left, is the 3 node. That’s a parent (interior node). The first non-leaf… size = 11, size/2 -1 =4. Remember, we floor it.

The last non-leaf node, I (going from left to right) adjust heap from it to the max

And decrement and repeat the process until we get to the root.

When we get the value 0, and we percolate from the top of the heap, down to the bottom, proper heap.

BuildHeap and \_AdjustHeap, are the keys to an efficient, in place sorting algorithm.

In-place means that we don’t require any extra storage for the algorithm and for the algorithm with the array itself.

And build heap.

1. Build Heap – turns an arbitrary array into a heap
2. 2. Swap first and last elements
3. 3. Adjust heap (0 to the last and not inclusive!)
4. Repeat 2-3 but decrement last each time through.

Max = 6, i=Max/2-1 = 2

I=2, adjust heapt. 6, 2… Well, we figure out what the max is and what the comparison. And we swap those two.

We always compare by the smallest child, swap the first and last element. And then we adjust down. And from 0 to the last.

I = 3.

Build Heap:

n/2 calls to \_adjustHeap = O(nlog n);

Heap sort: n calls to swap, and adjust = O(n log n)

Total = n log n

Hashing is somewhat like a magic machine, we put the hash function in – and that key can be anything. And it can be any value that can be stored and it needs to take the key and turns it into an integer index. And that is how it works.

We have to turn the key into an integer, look at it from the word, and if it is spelled correctly, then it won’t be flagged. Otherwise, they will put a bite over the word to let us know if it is wrong.

Hash functions are great because they are fast. Maps, part of the key to an integer.

Folding: key partitioned intoi parts, that are combined using efficient operatings.

Eat = 5+1+20, = 26.

EAT will become a three integer value word.

ATE AND TEA

We don’t just fold it, then we shift. We add things. Shift left twice is two two times.

Void put(ky – vtGet key

Int containsKey(KT Key)

Struct Association {

KT key;

VT value;

They do not have to be the same. The key type – string, value – number, for phone number.

You can use this in a lot of different formats. If statement…

We are looking at our compare function – dereference, we compare them.

When we do hash tables, we wil have collisions. And when that happens, we need to find a way to resolve it.

This is open-address hashing – if the spot is full, we look for the next spot. Open address hashing – hash value is used to find initial index to try. And then the hash value to try.

If that position is filled, the next position is examined, then the next, and so on until an empty position is found.

This process is called probing, and specifically is linear probing. And this is great illustrated with the examples. We look at the table and how its hashed.

These names have hashed to a particular area. We find that Amina is filled, and we then put an empty index with the new member.

The location is probed and we find it in the array. Hash to find initial index. Probe forward until, value is found or return 1. Empty location is found (return 0)

Notice that search time is not uniform. In this, we are looking for Andy and d is the integer 3. That is the index.

We find it immediately and it’s true. Search time is NOT uniform. CANNOT EMPHASIZE ENOUGH. Anne releases a function for determinant.

What do we do when we need to remove things? Sometimes it’s reasonable not to. And sometimes, if you are using a hash table, collection of words, and build a hash function for words.

The best solution is a “tombstone” – can be replaced – doesn’t halt the search during a contains or remove.

Can be replaced when adding new entry. Special value that marks deleted entry.

\_TS\_ - alan, andy, alessia, Alfred, aime… only halts on empty. Alan hashes, finds TS, then find Alan.

Otherwise, you might get a remove 0.

A tombstone is highly valuable.

The load factor is the total number of elements, n, divided by the state m.

Represents the portion of the buckets that are filled, for open address hashing, load factor, is between 0 and 1 – and often between 0.5 to 0.75 and this helps us avoid collisions.

Clustering – uniform distribution of hash values, what is the probability the next value we want to hash into this hash table is 1, 2, 6? Well, it depends where we start.

6 is the 4/8, because uniform distribution. Probing.

2 is 1/8, and 1 is 3/8 – for the hash starting at 2, and hash starting at either 7, 0, or 1.

Assumptions – the hash thing is constant. Best case, uniformly distributes, and worst case is when it goes to the same thing.

The worst case for open addressing is O(n) – because we have to do a for/while/do-while statement and that’s probing.

Best Case Open Addressing is O(1). The average case is different ii It is 1(1 – lambda) – lambda is a load factor.

Hash like sorting, and they create very fast sort programs. They are not general purpose, but very efficient for certain situations and only works on previous integers fin a particular range.

Quickly sort positive integer values from a limited range – count (tally) theoccurrences of each value, recreates the sorted values, according to tally. Example, sort 1,000 integers, elements with values between 0 and 19. Count tally, the occurrences of each value. Recreated to the sorted values according to tally. Whatever is biggest in the running time is the dominant – and it’s very efficient.

Radix Sort – punch cards programmed computers. Twelve punch locations in each column. Programs were written. We used to punch out in cards, and each columns has a series of holes. Each of those cards provided a digit or character.

It was a disaster waiting to happen. There was too much chance of dropping your cards and things getting spilled out. Used that to encode a sequence, and we’d look at the sequence and the number 0.

Let’s use a magic sorting machive… least significant digit – the 1s digit.

0 –

We are going to use chaining-buckets. Keep a collection, no matter what. We create a bad, this is influenced by the bag ADT. We implement a collection based on a linked list. Has table using chaining and buckets. And we do it with size five and there are two at hash 0.

Bucketing, lets us create tables and resize them, then rehash them. Then we need to look – normally chaining is O(1), but if we have tables, we get chains with O(1)+.

Worst case for open addressing – O(n)

Worst case for chaining will likely be O(n), give or take, depending on what happens.

Best case for opening O(1)

Best case for chaining O(1)

Average case of all operations O(lambda)

He use hash tables to store HASP MAPS – keys hash with return values – and rather than store linked lists, build the linked list directly. Lists directly – link \*\*hashtable;