02 - Binary Search Trees

***Notes***

**Definitions:**

1. Record - A collection of values for attributes of a single entity instance; a row of a table
2. Collection - a set of records of the same entity type; a table
3. Search Key - A value for an attribute from the entity type
4. Dsf

**Records**

* If each record takes up x bytes of memory, then for n records, we need n\*x bytes of memory

**Contiguously Allocated List**

* All n\*x bytes are allocated as a single chunk of memory. Ex. 6 records are in an array (row)
* Arrays are faster for random access, but slow for inserting anywhere but the end

**Linked Lists**

* Each record needs x bytes + additional space for 1 or 2 memory addresses
* Individual records are linked together in a type of chain using memory addresses
* Linked Lists are faster for inserting anywhere in the list, but slower for random access

**Binary Search**

Input: array of values in sorted order, target value

Output: the location(index) of where target is located or some value indicating target was not found

Best case: target is found at mid; 1 comparison (inside the loop)

Worst case: target is not in the array; log2 n comparisons

**Linear Search**

Best case: target is found at the first element; only 1 comparison

Worst case: target is not in the array; n comparisons

1. An array of tuples (specialVal, rowNumber) sorted by specialVal
   1. We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
   2. But, every insert into the table would be like inserting into a sorted array - slow…
2. A linked list of tuples (specialVal, rowNumber) sorted by specialVal
   1. searching for a specialVal would be slow - linear scan required
   2. But inserting into the table would theoretically be quick to also add to the list.

**Binary Search Tree**

* A binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent

Ex. Creating/inserting into a binary search tree (23,17,20,43,31,50)



**Tree Traversals**

Pre Order, Post Order, In Order, Level Order (HW1)

Level Order: It outputs the order from top to bottom left to right Ex -> 23,17,43,20,31,50

**Double ended queue:** insert and remove from both the front and the back

**Example Code for HW1:**

Class Binary Tree Node (self, value, left=none, right=none)

Value: int

Left: Binary Tree Node

Right: Binary Tree Node

In a function ->

Root = Binary Tree Node (23)

Root.left = Binary Tree Node (17)

Root.right = Binary Tree Node (43)

Root.left.right = Binary Tree Node (20)



Above is an unbalanced tree because it’s a straight diagonal line down. **The purpose of the Binary Search tree is to minimize the height as best as possible. This means the tree is as efficient as possible.**

**AVL Tree:**

* Is an approximate balanced Binary search tree.
* It maintains a balance factor in each node.
* AVL balance property
* Node of imbalance has an equation to restructure the tree to be balanced
* An insertion that causes imbalance can be fixed at a different node



4 Cases of Imbalance:

1. Left Left Case



1. Left Right Case



1. Right Left Case



1. Right Right Case



**Rebalancing each Case:**

Arrange the order of the nodes and trees so that the order still makes sense but they are rearranged so that they are balanced

**Example: Insert 3,2,1,4,5,6,7**



Insert 16,15,14 to tree above



08 - B+ Trees

***Notes***

1. CPU
2. Register
3. L1 Cache
4. L2 Cache
5. Ram
6. SDD/HDD

**B Trees:**

* In memory indexing

**B+ Trees:**

* It is an m-way tree only M -> maximum # of bytes in each
* M + 1 -> max children of each node
* Minimizes disk access for indexing
* Optimal for desk-based industries
* To check if full using code or within data structure
* Fill up node before splitting or moving left or right
* Have 2 different types of node structures

1. Internal nodes (only store key values and pointers (children))
2. Leaf nodes (store keys and data)(Bottom row)

* Take lowest value in leaf and move it to the top root

**Properties:**

* All nodes, except the root, must be ½ full
* Root nodes doesn't have to be ½ full
* Insertions are always done at leaf level
* Leaves are stored as a DLL

**Example: Insert M=3 (42, 29, 81, 99, 35, 2,30,45,100)**

****

****

****