Computational Complexity - Searching

Week 16

Required Activities

- Check Announcements regularly (every 2-3 days)
- Read FA book: Chapter 8
- Keep working on Assignment 5 (**due Feb 15**)

Searching Problems

- Normally limited to finding a record and returning it based on some key value
- For computational complexity we want to analyze and determine time complexity is as good or lower than lower bounds
- Need to consider whether the data to be searches is already sorted and how it is distributed (e.g. is probability the same)
- A number of algorithms discussed this week assume data is sorted (e.g. binary search tree, interpolation search, etc.)
- To determine lower bounds, need to find the longest path to find the key
  - For example, binary search would have d+1 number of comparisons where d is the depth of the tree

Static vs dynamic search

- **Static searching**: create records with all keys upfront and no change to the records
  - Can use an array
  - Can use binary search

- **Dynamic searching**: records are frequently added and deleted from the tree (e.g. airline reservation system)
  - Cannot use an array (would need to keep moving records)
  - Binary search cannot be used because need an array to find middle record (no efficient way to search linked list)
  - Can use tree structure

Binary Search Tree

- Can usually keep the average search time low, while being able to add and delete keys quickly
- Can quickly retrieve the keys in sorted sequence by doing an **in-order traversal** of the tree (left sub-tree, root, right sub-tree)
- Drawback that when keys are dynamically added and deleted, often end up with skewed tree (simply a linked list) and search becomes linear
- If the keys are added at random, the resulting tree will be closer to a balanced tree than it will be closer to a linked list
- The search time would usually be efficient O(log n)

B-Trees

- In many applications, performance can be degraded by a linear search time
  - e.g. keys for records in large databases cannot all fit into RAM so disk access is needed to search (**external search**)
  - When all the keys are simultaneously in memory, it is called an **internal search**
  - A linear-time search could prove to be unacceptable
- One solution is to write a balancing program that takes as its input an existing binary tree and outputs a balanced binary search tree containing the same keys
- In a very dynamic environment, it would be better if the tree never became unbalanced in the first place
- Algorithms for adding and deleting nodes while maintaining a balanced binary tree were developed by Adel'son-Velskii and Landis. Thus, balanced binary trees are often called **AVL trees**
- Bayer and McCreight developed an improvement over binary search trees called **B-trees**
  - When keys are added to or deleted from a B-tree, all leaves are guaranteed to remain at the same level, which is even better than maintaining balance

3-2 Tree

- B-trees actually represent a class of trees, of which the simplest is a **3–2 tree**
- A **3–2 tree** is a tree with the following properties:
  - Each node contains one or two keys
  - If a non-leaf contains one key, it has two children, whereas if it contains two keys, it has three children
  - The keys in the left subtree of a given node are less than or equal to the key stored at that node
  - The keys in the right subtree of a given node are greater than or equal to the key stored at that node
  - If a node contains two keys, the keys in the middle subtree of the node are greater than or equal to the left key and less than or equal to the right key.
  - All leaves are at the same level.
- The tree remains balances because the tree grows in depth at the root instead of at the leaves
- To delete a node, the tree shrinks in depth at the root (all leaves remain at the same level)
- Search, addition, and deletion times are guaranteed to be O (log *n*)
- In-order traversal retrieves the keys in sorted sequence
- B-trees are used most in modern database management systems

Hashing

- **hash function:** function that transforms a key to an index (hash function to a particular key is called “hashing the key”)
- e.g. social security numbers could have hash function ***h* (*key*) = *key* % 100** where 123-45-6789 would be index 89 (123456789/100 = xx.89) so basically last two digits of the key

- If a particular key hashes to i, we store the key and its record at *S* [*i*] (e.g. S[89])
- Not store the keys in sorted sequence so cannot retrieve the records efficiently in sorted sequence
- When two keys hash to the same index it is called a collision or hash clash and the more leys the more collisions

- **Open hashing :** method to handle collisions
  - create a bucket for each possible hash value and place all the keys that hash to a value in the bucket associates with that value
  - usually implemented through linked lists
  - number of buckets need not equal the number of keys but bucket stores only a pointer so not much space is wasted
  - when searching for a key, it is necessary to do a sequential search through the bucket (linked list) containing the key
  - if all the keys hash into the same bucket, the search becomes a sequential search (not likely)
  - when keys uniformly distributed in the buckets there is very small search time

Questions ?

- Post in the discussions
- Send email to RMcFadden@HarrisburgU.edu
- Respond usually within 48 hours