Kelsey Elliott

11/11/2017

COSC-3351-110

**Comparison of Binary Trees and Hash Tables**

**Binary Trees**

Binary trees are a hierarchical data structure that include a topmost node called the root, and the elements directly under an element are called its children. Those elements that have no children are called leaves and can point to null. Binary trees can only have a maximum of two children per node, while a 2-3-4 tree can have up to four children per parent node. 2-3-4 trees are a self-balancing data structure, and unlike binary trees, all external nodes are at the same depth. 2-3-4 trees can be more difficult to implement than a standard binary tree.

**Hash Tables**

Hash tables only supports a subset of the operations allowed by binary search trees. Tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations like *findMin*, *findMax*, and the printing of the entire table in sorted order in linear time are not supported. Hashing is a technique used for performing insertions, deletions, and finds in constant average time. A hash table can be thought of as an array of some fixed size containing items. This means that it is difficult to expand or decrease the size of a hash table once created. Hash tables support the search, insert, and delete operations at an average O(1) time. In project 6, to compute the hash value for strings I used the ASCII value for each character and summed them together.

**Hash Table vs Binary Trees**

Binary search trees, like hash tables, can be used to implement insert and contains operations. Although the resulting average time bounds are O(log *N*), binary search trees also support routines that require order and thus are more powerful. Using a hash table, it is not possible to find the minimum element or the maximum element, while this could easily be found using a binary tree data structure. It is also not possible to search efficiently for a string unless the exact string is known, while a binary tree could quickly find all items in a certain range. The O(log *N*) bound is not necessarily that much slower than O(1), since no multiplications or divisions are necessary in binary search trees.

The worst-case scenario for hashing generally results from an implementation error, whereas sorted input can make binary trees perform poorly. Balanced search trees, or 2-3-4 trees, can be quite expensive to implement, so if no ordering of information is required and there is any suspicion that the input might already be sorted then using a hash table would be the data structure of choice.

Hash tables seem to beat binary trees in the common operations: search, insert, and delete. However, binary trees offer advantages that hash tables cannot offer. In binary trees, you can get all keys in sorted order by doing an Inorder transversal. After everything is sorted in a binary tree, you would no longer need to waste extra memory or processing time. Order statistics, finding min and max elements, and doing range queries are easier to do with a binary tree data structure. I have found that binary trees are easier to customize, as hash table implementation usually relies on existing libraries. With self-balancing binary trees (2-3-4 trees or red-black trees), all operations are guaranteed to work in O(log *N*) time, but hash table o(1) time is only an average time and some particular operations may be costly especially when a table resizing becomes necessary.

In conclusion, when you are wanting to store data in a sorted manner, then a binary search tree is the better data structure option. However, not all data needs to be sorted and in that case a hash table would be the better data structure option.