A log file (or audit trail) is a dictionary implemented by means of an unsorted sequence

Performance: ◼ insertItem takes O(1)

findValue and removeItem take O(n)

Priority Queue that used an unordered sequence

Effective only for dictionaries of small size

A hash table is an example of a highly efficient implementation of an unordered Dictionary ADT (its operations have expected complexity O(1))

The **hash function** solves the problem of fast table-lookup, i.e., it allows the value associated with each key to be accessed quickly (in O(1) expected time).

Store item (k, o) at index i = h(k) in the table

Collisions occur when different elements are mapped to the same cell

Collisions occur when two different keys hash to the same index i

The average performance of hashing depends on how well the hash function distributes the set of keys (i.e., avoids collisions)

Load factor is n/N where n is the number items in the table and N is the table size

When the load factor goes above .75, the table is resized and the items are rehashed

Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell

Quadratic Probing

Start with the hash value i=h(k),

Then search A[(i + j2 ) mod N] for j = 0, 1, 2, … until an empty slot is found

Disadvantages ◼ Complicates removal even more ◼ Secondary clustering

**Performance of Hashing**

In the worst case, searches, insertions and removals on a hash table take O(n) time

The worst case occurs when all the keys inserted into the dictionary collide

The load factor α = n/N affects the performance of a hash table

Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is 1/(1−α)

The expected running time of all the dictionary ADT operations in a hash table is O(1)

In practice, hashing is very fast provided the load factor is not close to 100% Applications of hash tables: ◼ small databases ◼ compilers ◼ browser caches

**Ordered Dictionaries**

A Lookup Table is an example of an ordered Dictionary ADT allowing elements to be efficiently accessed in order by key. When implemented as an ordered sequence, searching for a key is relatively efficient, O(log n), but insertion and deletion are not, O(n).

**Lookup Tables:** Analogous to the Priority Queue implemented as a sorted Sequence

**Lookup Table:** A dictionary implemented by means of a sorted sequence ◼ store the items of the dictionary in an array-based sequence, sorted by key ◼ uses an external comparator for comparing keys How should we lookup and find the value associated with a given key? ▪ Hint: my favorite Dictionary because it uses my favorite algorithm

◼ findValue takes O(log n)

◼ insertItem takes O(n)

◼ removeItem takes O(n)

A hash table is a very efficient way of implementing an unordered Dictionary ADT; the running time of search, insertion, and deletion is expected O(1) time.

To achieve efficient behavior of the hash table operations takes a careful choice of table size, load factor, hash function, and handling of collisions.

A binary search tree allows users to associate keys to values, then to access or remove those elements by key. **An AVL** tree supports efficient implementation of the binary search tree by maintaining balance and order through restructuring (rebalancing) when key-element pairs are inserted and removed.

A binary search tree is a binary tree with the property that the value at each node is greater than the keys in the nodes of its left subtree (child) and less than the keys in the nodes of its right subtree. When implemented properly, the operations (search, insert, and remove) can be efficiently accomplished.

Fact: The height of an AVL tree storing n keys is O(log n).

Proof: Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

We easily see that n(1) = 1 and n(2) = 2

For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.

That is, n(h) = 1 + n(h-1) + n(h-2)

Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(h-6), … (by induction), n(h) > 2in(h-2i)

Solving the base case ◼ Pick i = ⎡h/2⎤ - 1 since value of the base cases are n(1) = 1 and n(2) = 2 ω i.e., pick i such that 1 < h-2i < 2 ◼ Thus we get: n(h) > 2 h/2-1

Taking logarithms: h < 2log n(h) +2

Thus the height of an AVL tree is O(log n)

AVL RUN TIME

a single restructure is O(1)

find, insert, remove O(log n)

The elimination of the worst case behavior of a binary search tree is accomplished by ensuring that the tree remains balanced, that is, the insert and delete operations do not allow any leaf to become significantly deeper than the other leaves of the tree.

1. In an AVL tree, the heights of the children of each node differ by at most 1; this maintains balance in the tree so search, insert, and delete can be done efficiently in O(log n) time.

2. In order to maintain balance in the tree as a whole, a tri-node restructuring is done whenever the tree gets out of balance, i.e., whenever the heights of the children of a node differ by more than 1.

A Map data structure allows users to assign keys to elements then to access or remove those elements by key. An ordered Map maintains an order relation among keys allowing access to adjacent keys in sorted order while supporting efficient implementation

B-Tree is a balanced multi-way search tree, i.e., all leaves are at the same depth

B-Trees are used to implement a file structure that allows random access by key as well as sequential access of keys in sorted order

Node-Size Property: every internal node has at most four children

Depth Property: all the external nodes have the same depth

(2,4) tree storing n items has height O(log n)

Quick-select is a randomized selection algorithm based on the prune-and-search