<https://sites.google.com/site/koalalearn/sp2011/cs111c>

Ideal hashing is when the key is also the index where they data can be found. A perfect hash function maps each search key into a different integer that is suitable as an index to the hash table

\*problem: sparse – that is, when only a few of the elements are actually in use (wasted space)

-note: a table is sparse if the potential is greater than the actual (<sqrt(n))

Solution to Ideal Hashing sparse problem: Convert and Compress

-compress the range of possible data values into a size of the number of locations that we want to use

How? We can use modulus:

1. convert the search key into an integer (hash code)

2. compress the hash code into the range of indexes of the hash table

\*problem to Convert and Compress – collisions can occur

-if hash functions returns the same value for different data

Solution to Collisions

1. Open addressing (use another location in the hash table – aka closed buckets)

2. Separate chaining (allow multiple entries per location – aka open buckets)

-note: probing is locating an open location in the hash table

Open Addressing

-open addressing scheme locates an alternate location in the hash table that is available or open

-use this new location to reference the new entry

a. Linear Probing

-if a collision occurs then we look for next available free spot in the array

-if probe sequence reaches the end of the hash table then it continues at the beginning of the table.

-ensures the success of the add operation as long as the hash table is not full

Retrievals – we can’t know exactly which value is where unless we start comparing search key values that were stored with data

Problem with Linear Probing – searches until hits null for values that contained duplicate hashkey calculations (this reverts back to O(n)) 🡪 Clustering

-note: as clusters grow in size they reduce the point of hashing

b. Quadratic Probing

-a solution to collisions AND clustering

-considers the locations that indices k, k+1, k+4, k+9… k+j^2

Problem: clustering still exists (called secondary clustering)

c. Double Hashing

-uses a second hash function to compute these increments in a key-dependent way.

Separate Chaining

-we alter the structure of the hashtable so that each location can represent more than one value

-such a location is called a bucket

-to find a value you hash the search key, locate the bucket, and look through the key-value pairs in it

-each location in the hash table is a head reference to a chain of linked nodes that make up the bucket

-each node contains references to a search key, to the key’s associated value and to the next node in the chain

-adding to the beginning of the chain is the fastest (if you don’t care about duplicates); else you search the chain and add to end if no duplicates are found

How to get a good Hash code?

Horner’s Method – multiplying the Unicode value of each character by a factor based on the character’s position within the string. The hash code is then the sum of these products.

STACKS

Adding to a stack 🡪 place item on top of the stack

Removing from stack 🡪 remove the topmost item (last one added)

AKA: LIFO

stack organizes its entries according to the order in which they were added.

-additions to one end of the stack called the top

-top entry is thus the newest item among the items currently in a stack

-stack restricts access to its entries

-a client can look at or remove only the top entry

-the only way to look at an entry that is not at the top of the stack is to repeatedly remove items from the stack until the desired item reaches the top

-NOTE: if you were to remove items from the stack one by one you would get them in reverse chronological order. ???

Push - add an entry to the stack

Pop – remove an entry from the stack

Peek – retrieves the top entry without removing it