Project 1

Zhecheng Li, 35688296

1. Linear Hashing (lazy delete)
   1. Theoretical predictions (from slides)

Average search time is constant, but it depends on load factor α. Efficiency of linear hashing degrades badly when α gets close to 1.

Expected time is for successful search, while for unsuccessful search.

Typical worst case search time is log n.

* 1. Description of algorithm
     1. Constructor

*Allocating an ArrayList with size of capacity of the hash table. Each element in the ArrayList is a Pair(key, value).*

* + 1. Hashing function

*Key % capacity*

* + 1. Set(key,value):

*Index = hashing(key)*

*Count = 0*

*While count < capacity AND element at index is not null AND key of element at index is not null:*

*Index = (index + 1) % capacity*

*Count++*

*If count is not capacity:*

*Put (key,value) at index*

* + 1. Search(key):

*Index = hashing(key)*

*Count = 0*

*While count < capacity AND list at index is not null AND key of element at index is not key:*

*Index = (index + 1) % capacity*

*Count++*

*If count < capacity AND element at index is not null:*

*Result = value of element at index*

* + 1. Delete(key)

*Index = hashing(key)*

*Count = 0*

*While count < capacity AND element at index is not null AND (key of element at index is null OR key of element at index not equals key):*

*Index = (index + 1) % capacity*

*Count++*

*If count < capacity AND element of index is not null:*

*Set key of element at index to null*

* 1. Test results

The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The time cost is tested every 50 operations (set, get, delete). It is tested for 20 times to get the average.

1. Chained Hashing
   1. Theoretical predictions (from slides)

time/operation = O(1 + length(H[h(k)])

E[time/operation] = O(1 + α)

Typical worst case search time is

* 1. Description of algorithm
     1. Constructor

*Allocating an ArrayList with size of the capacity of hash table. Each element of the ArrayList is a LinkedList. Eech element of the LinkedList is a Pair(key, value).*

* + 1. Hashing function

*Key % capacity*

* + 1. Set

*Index = hashing(key)*

*If element at index is null:*

*New a LinkedList*

*Add a Pair(key, value) into the LinkedList*

*Put the LinkedList at index*

*Else:*

*Add the Pair(key, value) to the LinkedList at index*

* + 1. Search

*Index = hashing(key)*

*If element at index is not null:*

*For pair in the LinkedList:*

*If key of the pair equals key:*

*Return the pair*

* + 1. Delete

*Index = hashing(key)*

*If element at index is not null:*

*For pair in the LinkedList:*

*If key of the pair equals key:*

*delete the pair*

* 1. Test results

The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The time cost is tested every 50 operations (set, get, delete). It is tested for 20 times to get the average.

1. Cuckoo Hashing
   1. Theoretical predictions (from slides)

Guaranteed O(1) search

The expected time for the sequence of operations is O(n)

* 1. Description of algorithm
     1. Constructor

*Allocating two ArrayList with size of capacity of the hash table. Elemnt of an ArrayList is a Pair(key, value).*

* + 1. Hashing function

*One function: key % capacity*

*Another function: (key % capacity) % capapcity*

* + 1. findPlace(toPutPair) function

*t = 0*

*count = 0*

*index = hashing(t, key of toPutPair)*

*while element of ArrayList t at index is not null AND count < capacity:*

*if key of element of ArrayList t at index equals key:*

*break*

*put toPutPair to current position*

*update toPutPair with Pair at current position*

*t = 1 – t*

*index = hashing(t, key of toPutPair)*

*count++*

*if count >= capacity:*

*return -1 (to resize) and toPutPair*

*else:*

*return position and toPutPair*

* + 1. resize(toPutPair)

*backupList = this.list*

*noConflict = true*

*do*

*noConflict = true*

*capacity \*= 2*

*allocate new ArrayList with double size*

*put Pair of oldList into new ArrayList, if conflict set nonConflict to false*

*put toPutPair into new ArrayList, if conflict set nonConflict to false*

*while noConflict is false*

* + 1. Set(key, value):

*findPlace(toPutPair)*

*if need resize:*

*resize*

*else:*

*put toPutPair to the empty slot of one of the ArrayLists*

* + 1. Search(key)

*Try to find the key at position (hashing result) in each ArrayList.*

*If found:*

*Return it*

*Else:*

*Return null*

* + 1. Delete(key)

*Try to find the key at position (hashing result) in each ArrayList.*

*If found:*

*Delete it*

* 1. Test results

The capacity of the hash table is 5,000. I test sequence time cost of operations (set, get, delete) from 100 to 4,800, 48 data point in total. I do the test for 300 times to get the average.

1. Quadratic Hashing (lazy delete)
   1. Theoretical predictions (from slides)
   2. Reason for choosing it

Quadratic probing is better since it avoids the clustering problem of linear probing-Wikipedia

* 1. Description of algorithm
     1. Constructor

*Allocating an ArrayList with size of capacity of the hash table. Each element in the ArrayList is a Pair(key, value).*

* + 1. Hashing function

*Key % capacity*

*(hash result should be: key % capacity + , it is completed in each operation function )*

* + 1. Set(key,value):

*Count = 1*

*hashVal = hashing(key)*

*Index = (hashVal + count \* count) % capacity*

*While count < capacity + 1 AND element at index is not null AND key of element at index is not null:*

*Index = (hashVal + count \* count) % capacity*

*Count++*

*If count < capacity + 1:*

*Put (key,value) at index*

* + 1. Search(key):

*Count = 1*

*hashVal = hashing(key)*

*Index = (hashVal + count \* count) % capacity*

*While count < capacity + 1 AND element at index is not null AND (key of element at index is null OR key of element at index not equals key):*

*Index = (hashVal + count \* count) % capacity*

*Count++*

*If count < capacity + 1 AND element at index is not null:*

*Return value of element at index*

* + 1. Delete

Count = 1

*hashVal = hashing(key)*

*Index = (hashVal + count \* count) % capacity*

*While count < capacity + 1 AND element at index is not null AND (key of element at index is null OR key of element at index not equals key):*

*Index = (hashVal + count \* count) % capacity*

*Count++*

*If count < capacity + 1 AND element at index is not null:*

*Set key of element at index to null*

* 1. Test results

The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The time cost is tested every 50 operations (set, get, delete). It is tested for 20 times to get the average.