**CSED 2014**

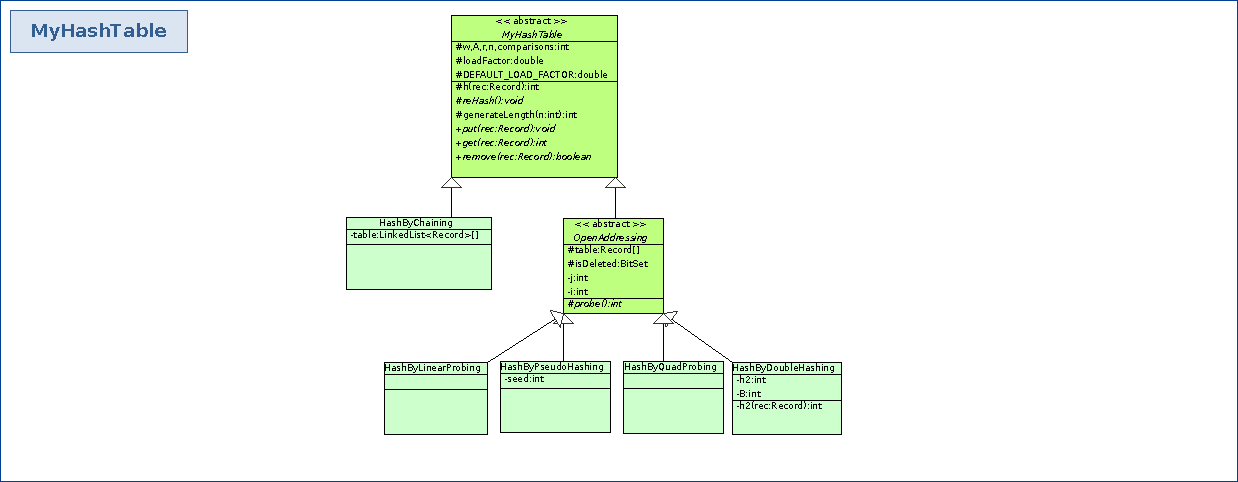
12

**Hashing Equivalent Classes**

**Ashraf Saleh Mohamed Aly (20)**

**Ashraf Saleh**

# UML :



# Algotithms :

## Hashing algorithms:

h1(Record rec) //primary hashing function based on name field

k <- rec.getName().hashCode();

//A\*k mod 2^w >> (w-r)

//taking the mod to 2^w = taking the w bits from the number

return A\*k >> (w-r);

h2(Record rec) //secondary hashing used in DoubleHashing

k <- rec.getName().hashCode();

h2 = (A\*k + B)% (1<<r);

//where A and B are Primes

return h2;

Algorithm : ReHash()

make a new table with double size

re-put all entries in the old table

## Hash By Chaning

Algorithm : put (Record rec)

index <- h(rec);

add record to table at index ;

comparisons <- 0;

loadFactor <- ++n / (1<<r) ;

if (loadFactor > DEFAULT\_LOAD\_FACTOR)

reHash() ;

Algorithm : get(Record rec)

comparisons <- 0;

index <- h(rec);

for (all entries at index in table){

comparisons++;

if (rec == entry)return index;

}

//if didn’t return at the for loop it means that rec is not found in the table;

return comparisons =-1 ;

Algorithm : Remove(Record rec)

index = h(rec);

comparisons = 0;

for (all entries at index in table){

if (rec== entry)

comparisons++;

remove rec from table;

return true;

}

//if didn’t return in the for loop then rec was not found in the table

comparisons = -1;

return false ;

## Open Addressing Class

Algorithm : put(Record rec)

h <- h(rec);

index <- h;

j = comparisons <- 0 ; // j is the number of collisions

while(table[index]!=null && record at index is not marked for deletion){

index <- probe(h,j++);

comparisons++;

}

table[index] <- rec ;

loadFactor <- ++n / (1<<r) ;

if (loadFactor > DEFAULT\_LOAD\_FACTOR)

reHash() ;

Algorithms : get(Record rec)

// To improve, when an element is searched and found in the table,

// the element is relocated to the first location marked for deletion

// that was probed during the search.

h = h(rec);

int i <- h;

j == comparisons <- 0 ;

int firstDeleted <- -1;

boolean found = false ;

//don't have to check that # of probes < length as i rehash when the load factor > 0.7

// for sure i'll find an empty slot

while(table[i] != null){

comparisons++;

if (record was marked for deletion)

if (firstDeleted == -1)

firstDeleted = i;

i <- probe(h,j++);

else

found = rec.equals(table[i]);

// update the table

if (found && firstDeleted != -1)

table[firstDeleted] = rec ;

break;

}

if (found)

return i;

else

return comparisons = -1;

Algorithm : remove(Record rec)

index <- get(rec);

if (found)

return false;

mark entry as deleted

n--;

return true;

## Linear Probing Class

Algorithm : probe(int i, int j)

return (i+j)%(1<<r);

## Quad Probing Class

Algorithm : probe(int i, int j)

return (i+j\*j)%(1<<r);

## Double Hashig Class

Algorithm : probe(int i, int j)

return h+j\*h2;

## pseudo Random Probing Class

Algorithm : probe(int i, int j)

return (seed\*(h+j) + j\*7549 )%(1<<r);

# DataStrucures :

BitSet : to mark entries in the table as deleted (in open addressing classes)

# Order comparison

hashing by external Chaining

memory : n records + n links (int size links)

order : \theta (1 + \alpha ) == O(m)

where m is the table size

hashing by open Addressing

memory : n records

order : \theta (1/(1-\alpha))