**3.4.1** Insert the keys E A S Y Q U T I O N in that order into an initially empty table of *M* = 5 lists, using separate chaining. Use the hash function 11 k % M to transform the *k*th letter of the alphabet into a table index.

|  |  |
| --- | --- |
| 0 | E Y T O |
| 1 | A U |
| 2 | Q |
| 3 |  |
| 4 | S I N |

**3.4.2** Develop an alternate implementation of SeparateChainingHashST that directly uses the linked-list code from SequentialSearchST.

**3.4.3** Modify your implementation of the previous exercise to include an integer field for each key-value pair that is set to the number of entries in the table at the time that pair is inserted. Then implement a method that deletes all keys (and associated values) for which the field is greater than a given integer k. *Note* : This extra functionality is useful in implementing the symbol table for a compiler.

**public** **void** deleteAllKeysWithIndexLargerThan(**int** k)

{

**if**( k < 0 ) **return**;

**for**(**int** i = 0 ; i < *M* ; i++)

{

Node n = buckets[i];

**if**(n != **null**)

{

//1. delete all leading nodes with index > k

**while**(n != **null** && n.Index > k)

{

n = n.Next;

N--;

}

buckets[i] = n;

//2. now n is the first node with index <= k,

//search for the end node with this property.

**while**(n != **null**)

{

**while**(n.Next != **null** && n.Next.Index <= k)

n = n.Next;

//3. delete all nodes with index > k

Node p = n.Next;

**while**(p != **null** && p.Index > k)

{

p = p.Next;

N--;

}

n.Next = p;

n = p;

}

}

}

}

**3.4.4** Write a program to find values of a and M, with M as small as possible, such that the hash function (a \* k) % M for transforming the *k*th letter of the alphabet into a table index produces distinct values (no collisions) for the keys S E A R C H X M P L.

The result is known as a *perfect hash function*.

**public** **static** **int**[] getPerfectHashingParameter(**char**[] keys)

{

**int**[] ret = **new** **int**[2];

**for**(**int** M = keys.length ; ; M++)

{

**boolean**[] marker = **new** **boolean**[M];

**boolean** hasCollision = **false**;

**int** a = 1;

**for**( ; a < M ; a++)

{

hasCollision = **false**;

*clearMarker*(marker);

**for**(**int** i = 0 ; i < keys.length && !hasCollision ; i++)

{

**int** hash = *hash*(keys[i], a, M);

**if**(!marker[hash])

marker[hash] = **true**;

**else**

hasCollision = **true**;

}

**if**(!hasCollision)

{

ret[0] = a;

ret[1] = M;

**return** ret;

}

}

}

}

**private** **static** **int** hash(**char** c, **int** a, **int** M)

{

**return** ((a \* (c - 'A')) % M);

}

**private** **static** **void** clearMarker(**boolean**[] marker)

{

**for**(**int** i = 0 ; i < marker.length ; i++)

marker[i] = **false**;

}

**3.4.5** Is the following implementation of hashCode() legal?

public int hashCode()

{ return 17; }

If so, describe the effect of using it. If not, explain why.

It’s legal. But this hash function will make the hash table equals to the linear search (for separate chaining) and worse than linear (for linear probing).

**3.4.6** Suppose that keys are *t*-bit integers. For a modular hash function with prime *M*, prove that each key bit has the property that there exist two keys differing only in that bit that have different hash values.

1. When M >= 2 ^ t – 1, for any 2 keys with 1 bit difference, suppose their values are k1, k2 (k1 != k2), their hash values would be k1, k2 as well because k1, k2 <= M, k1 % M = k1, k2 % M = k2. Their hash values are different.

2. When M = 2, the minimum prime. The last 1 bit will be the key bit. For 2 integers represented by t-bit have different parity they will have 2 different hash value because their last bits are different.

3. Let’s say mth bit (rightmost bit is the 1th bit) is a key bit and A1, A2 are 2 keys differing in that bit. Suppose A1 > A2, then A1 =A2 + 2^(m-1).

H2 = A2 % M

H1 = A1 % M = (H2 + 2^(m-1) % M) % M

Because M is a prime > 2, 2^(m-1) % M != 0.

Let c = 2^(m-1) % M, clearly c < M.

* H1 = (H2 + c) % M = H2 + c - floor((H2 + c) / M) \* M
* H1 != H2, because if they equal, we will have c = floor((H2+c)/M) \* M => c/M = floor((H2+ c) / M). Left hand side in range (0, 1) while right hand side is an integer.

All these 3 cases prove the question.

**3.4.7** Consider the idea of implementing modular hashing for integer keys with the code (a \* k) % M, where a is an arbitrary fixed prime. Does this change mix up the bits sufficiently well that you can use nonprime M?

No. By using a nonprime M, the hash function favors last a few bits of all bits whereas multiplication does not permute those limited bits by all bits but only themselves. However (k % a) % M where M is nonprime and ‘a’ is a prime seems to be an option, as Q&A section shows.

**3.4.8** How many empty lists do you expect to see when you insert *N* keys into a hash table with SeparateChainingHashST, for *N*=10, 102, 103, 104, 105, and 106? *Hint* : See Exercise 2.5.31.

**3.4.9** Implement an eager delete() method for SeparateChainingHashST.

**3.4.10** Insert the keys E A S Y Q U T I O N in that order into an initially empty table of size *M* =16 using linear probing. Use the hash function 11 k % M to transform the *k*th letter of the alphabet into a table index. Redo this exercise for *M* = 10.

E – 5 55 U – 21 231

A – 1 11 T – 20 220

S – 19 209 I – 9 99

Y – 25 275 O – 15 165

Q – 17 187 N – 14 154

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|  | S |  | Y | I | O |  | E | U |  | N | A | Q | T |  |  |

Probes:

E – 1 Q – 2 O - 1

A - 1 U – 2 N – 1

S – 1 T - 2

Y – 1 I – 2

Total = 14

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| T | A | U | I | N | E | Y | Q | O | S |

Probes:

E – 1 U - 2

A – 1 T - 1

S – 1 I - 5

Y – 2 O - 4

Q – 1 N - 1

Total – 19

**3.4.11** Give the contents of a linear-probing hash table that results when you insert the keys E A S Y Q U T I O N in that order into an initially empty table of initial size *M* = 4 that is expanded with doubling whenever half full. Use the hash function 11 k % M to transform the *k*th letter of the alphabet into a table index.

**3.4.12** Suppose that the keys A through G, with the hash values given below, are inserted in some order into an initially empty table of size 7 using a linear-probing table (with no resizing for this problem). Which of the following could not possibly result from inserting these keys?

key: A B C D E F G

hash: 3 5 3 4 5 6 3

*a.* E F G A C B D

*b.* C E B G F D A

*c.* B D F A C E G

*d.* C G B A D E F

*e.* F G B D A C E

*f.* G E C A D B F

Give the minimum and the maximum number of probes that could be required to build a table of size 7 with these keys, and an insertion order that justifies your answer.

1. ABCDEFG
2. No, because F is before A
3. ACEGBDF
4. ADEFCGB
5. No, because A is after D.
6. ADBFGEC

Maximum number of probes = Minimum number = 20 and any insertion sequence will get this result.

**3.4.13** Which of the following scenarios leads to expected *linear* running time for a random search hit in a linear-probing hash table?

*a.* All keys hash to the same index.

*b.* All keys hash to different indices.

*c.* All keys hash to an even-numbered index.

*d.* All keys hash to different even-numbered indices.

A and C

A ~ (N+1) / 2

B ~ 1

C ~ (N+1) / 2

D ~ 1

**3.4.14** Answer the previous question for search *miss*, assuming the search key is equally likely to hash to each table position.

A ~ 1+ (N+1) \* N / 2M

B ~ 1 + N/2M + 1/2M \* SUM (i from 1 to k) L(i)^2 where k is the number of clusters.

C = A

D ~ N/4M

A and C

**3.4.15** How many compares could it take, in the worst case, to insert *N* keys into an initially empty table, using linear probing with array resizing?

Every resizing operation costs re-hash of all items in the table.

So there’re 2 items need to be considered:

1. Hash N keys into the table in the worst case, which equals to N(N+1)/2;
2. Re-hash all existing keys in the table when resizing which equals to SUM(k = 1 to 1) k + SUM (k = 1 to 2) k + SUM(k = 1 to 3) k + SUM(k = 1 to 4) k + SUM(k = 1 to 8) k + SUM(k = 1 to 16) k … +SUM(k = 1 to N) (here we suppose N = 2^m which is the worst case because the last insertion causes double-resize as well).

Result = 10 + SUM(k = 2 to m) 2^k\*(2^k+1)/2 + N(N+1)/2

= 10 + (1/2) \* (SUM(4^k) + SUM(2^k)) + N(N+1)/2

= 10 + (1/2) \*((4\*N^2 - 16)/3 + 2N – 4) + N(N+1)/2

= 8 + (4\*N^2 -16)/6 + N^2/2 + 3N/2

= (7N^2 + 9N – 16)/6 + 8

**3.4.16** Suppose that a linear-probing table of size 106 is half full, with occupied positions chosen at random. Estimate the probability that all positions with indices divisible by 100 are occupied.

Let Xi be the r.v. denotes item i of the table is occupied. Since the table is half full, we have

P(Xi = occupied) = 1/2 where i = 0, 1… 10^6 – 1.

Positions with indices divisible by 100 = 10^6 / 100 = 10^4

P(X0 = occupied & X100 = occupied & X200 = occupied … & X999900 = occupied) = 1/ (2^10000)

**3.4.17** Show the result of using the delete() method on page 471 to delete C from the table resulting from using LinearProbingHashST with our standard indexing client (shown on page 469).

**3.4.18** Add a constructor to SeparateChainingHashST that gives the client the ability to specify the average number of probes to be tolerated for searches. Use array resizing to keep the average list size less than the specified value, and use the technique described on page 478 to ensure that the modulus for hash() is prime.

**3.4.19** Implement keys() for SeparateChainingHashST and LinearProbingHashST.

**public** Iterable<TKey> keys() {

Queue<TKey> ret = **new** LinkedList<TKey>();

**for**(Node first : slots)

**while**(first != **null**)

{

ret.add(first.Key);

first = first.Next;

}

**return** ret;

}

**public** Iterable<TKey> keys() {

Queue<TKey> q = **new** LinkedList<TKey>();

**for**(**int** i = 0, j = 0 ; i < N ; j++)

**if**(keys[j] != **null**)

{

q.add(keys[j]);

i++;

}

**return** q;

}

**3.4.20** Add a method to LinearProbingHashST that computes the average cost of a search hit in the table, assuming that each key in the table is equally likely to be sought.

**public** **void** put(TKey key, TValue value) {

**if**(N == M / 2)

resize(M \* 2);

**int** h = hash(key);

**int** i = h;

**int** potentialProbe = 0;

**while**(keys[i] != **null** )

{

**if**(keys[i].equals(key))

{

values[i] = value;

**return**;

}

**else**

{

potentialProbe++;

i = (i + 1) % M;

}

}

//insert;

totalInsertionProbes += potentialProbe + 1;

keys[i] = key;

values[i] = value;

N++;

}

**public** **void** delete(TKey key) {

**int** h = hash(key);

**int** i = h;

**while**(keys[i] != **null** && !keys[i].equals(key))

i = (i+1) % M;

**if**(keys[i] == **null**) //not found

**return**;

totalInsertionProbes -= i - h + 1;

keys[i] = **null**;

values[i] = **null**;

N--;

//fix following keys

**for**(i = (i + 1) % M; keys[i] != **null** ; i = (i + 1) % M)

{

TKey k = keys[i];

TValue v = values[i];

h = hash(k);

totalInsertionProbes -= i - h + 1;

keys[i] = **null**;

values[i] = **null**;

N--;

put(k,v);

}

**if**(N <= M / 8) resize(M/2);

}

**public** **double** avgSearchHitProbes()

{

**return** N != 0 ? 1.0 \* totalInsertionProbes / N : 0;

}

**public** **double** avgSearchHitPrb()

{

**return** 0.5 \* (1 + 1 / (1-loadFactor()));

}

**public** **double** avgSearchHitPrbAlt()

{

**double** alpha = loadFactor();

**return** (-1.0 / alpha) \* Math.*log*(1 - alpha);

}

**3.4.21** Add a method to LinearProbingHashST that computes the average cost of a search *miss* in the table, assuming a random hash function. *Note* : You do not have to compute any hash functions to solve this problem.

**public** **double** avgSearchMissProbes()

{

**int** sumOfClusterSqr = 0;

//find the start point of last clustering (if any)

**int** e = M - 1;

**while**(keys[e] != **null**)

e--;

**for**(**int** i = (e + 1) % M ; i < e ; i = (i + 1) % M)

{

**int** clusteringLen = 0;

**for**(**int** j = i ; i < e && keys[j] != **null** ; j = (j+1) % M, i = j)

clusteringLen++;

sumOfClusterSqr += clusteringLen \* clusteringLen;

}

**return** 1 + loadFactor() / 2 + 1.0 \* sumOfClusterSqr / (2\*M);

}

**public** **double** avgSearchHitPrbAlt()

{

**double** alpha = loadFactor();

**return** (-1.0 / alpha) \* Math.*log*(1 - alpha);

}

**public** **double** avgSearchMissPrb()

{

**return** 0.5 \* (1 + 1 / Math.*pow*((1-loadFactor()),2));

}

**3.4.22** Implement hashCode() for various types: Point2D, Interval, Interval2D, and Date.

Point2D

@Override

**public** **int** hashCode()

{

**return** (**int**) ((Double.*doubleToLongBits*(x) ^ Double.*doubleToLongBits*(y)) % Integer.*MAX\_VALUE*);

}

Interval

@Override

**public** **int** hashCode()

{

**return** (**int**) ((Double.*doubleToLongBits*(low) ^ Double.*doubleToLongBits*(high)) % Integer.*MAX\_VALUE*);

}

Interval2D

@Override

**public** **int** hashCode()

{

**return** x.hashCode() ^ y.hashCode();

}

Date

@Override

**public** **int** hashCode()

{

**return** (31 \* year + 17 \* month + day) % Integer.*MAX\_VALUE*;

}

**3.4.23** Consider modular hashing for string keys with R = 256 and M = 255. Show that this is a bad choice because any permutation of letters within a string hashes to the same value.

Suppose we have a string with length N: S = C1C2C3…Cn

For H(S) = SUM (k from 1 to n) H(Ck) \* R^(k-1) % M

When R = 256 and M = 255 => R^k %M = 1 for any k > 0.

So H(S) = SUM (k from 1 to n) H(Ck) % M. By the commutative law of addition, any permutation of the string will have same hash value.

**3.4.24** Analyze the space usage of separate chaining, linear probing, and BSTs for double keys. Present your results in a table like the one on page 476.

|  |  |
| --- | --- |
| Method | Memory usage for double key type |
| Separate chaining | ~48N +32M |
| Linear probing | ~32N to ~128N |
| BST | ~56N |

Separate chaining:

Every slot is a referent to LinearSearchST which uses 8 bytes;

Every LinearSearchST 16 bytes(overhead) + 8 bytes(Node first);

Then total 32M.

For each node:

Overhead = 16 bytes

Key = 8 bytes (double)

Value = 8 bytes (ref)

Next = 8 bytes (ref)

Outer ‘this’ ref = 8 bytes (ref)

48N.

Linear probing:

Suppose there’re N items in table, then the table size is [2N, 8N]

For each item, it needs a Key ref and a Value ref but no overhead because they’re represented by array directly. So totally [32N, 128N]

BST:

For each node:

Overhead = 16 bytes

Key = 8 bytes (double)

Value = 8 bytes (ref)

Left = 8 bytes (ref)

Right = 8 bytes (ref)

Size = 4 bytes (int)

Padding = 4 bytes

Total 56N

**3.4.25** *Hash cache.* Modify Transaction on page 462 to maintain an instance variable hash, so that hashCode() can save the hash value the first time it is called for each object and does not have to recompute it on subsequent calls. *Note* : This idea works only for immutable types.

**private** **int** hashCode = Integer.*MAX\_VALUE*;

@Override

**public** **int** hashCode()

{

**if**(hashCode == Integer.*MAX\_VALUE*)

{

hashCode = (**int**)(

(

(31 \* who.hashCode() + 17 \* when.hashCode())

^ Double.*doubleToLongBits*(amount))

% Integer.*MAX\_VALUE*

);

}

**return** hashCode;

}

**3.4.26** *Lazy delete for linear probing.* Add to LinearProbingHashST a delete() method that deletes a key-value pair by setting the value to null (but not removing the key) and later removing the pair from the table in resize(). Your primary challenge is to decide when to call resize(). *Note* : You should overwrite the null value if a subsequent put() operation associates a new value with the key. Make sure that your program takes into account the number of such *tombstone* items, as well as the number of empty positions, in making the decision whether to expand or contract the table.

**public** **void** put(TKey key, TValue value) {

**if**(N + tombStones == M / 2)

resize(M \* 2);

**int** h = hash(key);

**int** i = h;

**while**(keys[i] != **null** )

{

**if**(keys[i].equals(key))

{

**if**(values[i] == **null**) tombStones--;

values[i] = value;

**return**;

}

**else**

{

i = (i + 1) % M;

}

}

//insert;

keys[i] = key;

values[i] = value;

N++;

}

**public** **void** delete(TKey key) {

**int** h = hash(key);

**int** i = h;

**while**(keys[i] != **null** && !keys[i].equals(key))

i = (i+1) % M;

**if**(keys[i] == **null**) //not found

**return**;

values[i] = **null**;

N--;

tombStones++;

**if**(N <= M / 8) resize(M/2);

}

**private** **void** resize(**int** newSize)

{

TKey[] oldKeys = keys;

TValue[] oldValues = values;

keys = (TKey[])**new** Object[newSize];

values = (TValue[])**new** Object[newSize];

**int** n = N;

**int** m = M;

M = newSize;

N = 0;

tombStones = 0;

**for**(**int** i = 0, k = 0 ; i < n && k < m ; k++)

{

**if**(oldKeys[k] != **null** && oldValues[k] !=**null**)

{

put(oldKeys[k], oldValues[k]);

i++;

}

}

}

**3.4.27** *Double probing.* Modify SeparateChainingHashST to use a second hash function and pick the shorter of the two lists. Give a trace of the process of inserting the keys E A S Y Q U T I O N in that order into an initially empty table of size *M* =3 using the function 11 k % M (for the *k*th letter) as the first hash function and the function 17 k % M (for the *k*th letter) as the second hash function. Give the average number of probes for random search hit and search miss in this table.

|  |  |
| --- | --- |
| 0 |  |
| 1 |  |
| 2 |  |

**3.4.28** *Double hashing.* Modify LinearProbingHashST to use a second hash function to define the probe sequence. Specifically, replace (i + 1) % M (both occurrences) by (i + k) % M where k is a nonzero key-dependent integer that is relatively prime to M. *Note* : You may meet the last condition by assuming that M is prime. Give a trace of the process of inserting the keys E A S Y Q U T I O N in that order into an initially empty table of size *M* =11, using the hash functions described in the previous exercise. Give the average number of probes for random search hit and search miss in this table.

**3.4.29** *Deletion.* Implement an eager delete() method for the methods described in each of the previous two exercises.

**3.4.30** *Chi-square statistic.* Add a method to SeparateChainingST to compute the \_ 2 statistic for the hash table. With *N* keys and table size *M*, this number is defined by the equation

\_ 2*=* (*M/N*) ( (*f*0 \_ *N/M*)2 *+* (*f*1 \_ *N/M*)2 \_ *. . .* (*fM* \_ 1\_ *N/M*)2 ) here *fi* is the number of keys with hash value *i*. This statistic is one way of checking our assumption that the hash function produces random values. If so, this statistic, for *N > cM*, should be between *M* \_ \_ *M* and *M +* \_ *M* with probability 1 \_ 1/*c*.

**public** **double** kaiSquareStat()

{

**double** ret = 0;

**double** loadFactor = (1.0 \* size()) / M;

**for**(**int** i = 0 ; i < M ; i++)

**if**(slots[i] != **null**)

ret += Math.*pow*(slots[i].size() - loadFactor, 2);

**return** ret / loadFactor;

}

**3.4.31** *Cuckoo hashing.* Develop a symbol-table implementation that maintains two hash tables and two hash functions. Any given key is in one of the tables, but not both. When inserting a new key, hash to one of the tables; if the table position is occupied, replace that key with the new key and hash the old key into the other table (again kicking out a key that might reside there). If this process cycles, restart. Keep the tables less than half full. This method uses a constant number of equality tests in the worst case for search (trivial) and amortized constant time for insert.

**3.4.32** *Hash attack.* Find 2*N* strings, each of length 2*N*, that have the same hashCode() value, supposing that the hashCode() implementation for String is the following:

public int hashCode()

{

int hash = 0;

for (int i = 0; i < length(); i ++)

hash = (hash \* 31) + charAt(i);

return hash;

}

*Strong hint* : Aa and BB have the same value.

**public** **static** String[] stringsWithSameHash(**int** n)

{

String[] ret = **new** String[2 \* n];

**for**(**int** i = 0 ; i < 2\*n ; i++)

{

StringBuilder sb = **new** StringBuilder();

String bs = Integer.*toBinaryString*(i);

**for**(**int** k = 0 ; k < bs.length() ; k++)

{

**if**(bs.charAt(k) == '0') sb.append("Aa");

**else** sb.append("BB");

}

**while**(sb.length() / 2 < Math.*floor*(Math.*log*(2 \* n)/Math.*log*(2)) + 1)

sb.insert(0, "Aa");

ret[i] = sb.toString();

}

**return** ret;

}

**3.4.33** *Bad hash function.* Consider the following hashCode() implementation for String, which as used in early versions of Java:

public int hashCode()

{

int hash = 0;

int skip = Math.max(1, length()/8);

for (int i = 0; i < length(); i += skip)

hash = (hash \* 37) + charAt(i);

return hash;

}

Explain why you think the designers chose this implementation and then why you think it was abandoned in favor of the one in the previous exercise.

This hash function runs faster (n/8 faster) than previous one for it only considers partial amount of the whole string. But this performance gain leads to much more collisions.