Chapter 12-2, Lecture Notes

Hashing

1. Binary search trees are impressive with being able to search 10,000 items in only 13 steps.
   1.1 However, that might not be adequate for some situations – like calling 911
      1.1.1 Need something closer to O(1) and not O($\log_2$ n)
      1.1.2 Need an "Address calcualtor" like in fig 12-44

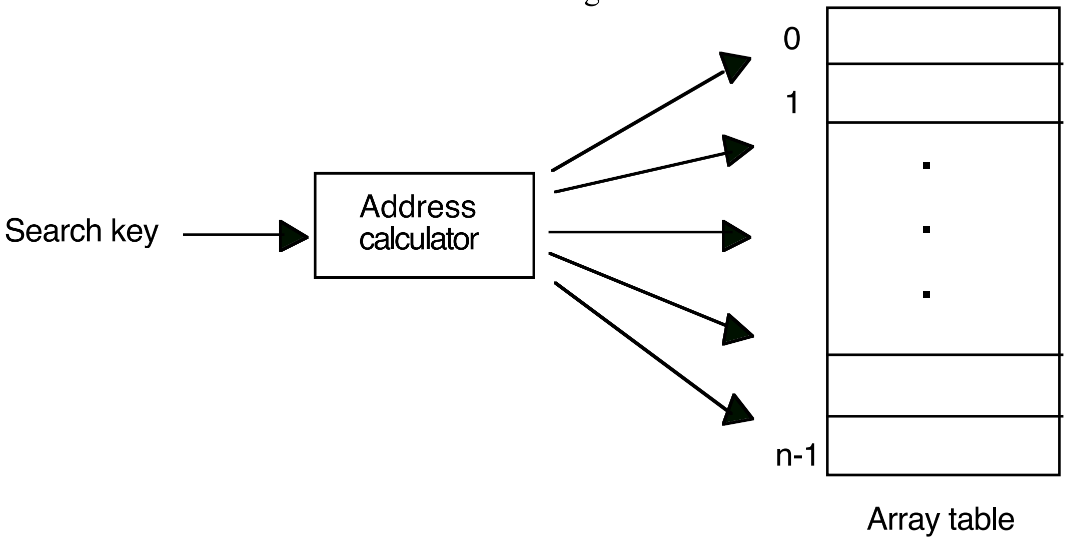


Figure 12-44, Address calculator

1.2 Can do an insertion into the table by:

```
tableInsert(in newItem:TableItemType)

        i = the array index that the address calculator gives you
                for newItem's search key
        table[i] = newItem
```

1.3 The above is O(1) – requires constant time

1.4 You can also use address calculator for tableRetrive and tableDelete, like so

```
tableRetrieve(in searchKey:KeyType, out tableItem:tableItemType)
                throw TableException

        i = the array index that the address calculator gives you for an item whose
                search key equal searchKey
        if (table[i].getKey() != searchKey)
                throw a TableException
        else
                tableItem = table[i]
```

```
tableDelete(in searchKey:KeyType)
            throw TableException

      i = the array index that the address calculator gives you for an item whose
            search key equals searchKey

      if (table[i].getKey() != searchKey)
            throw a TableException
      else
            Delete the item from table[i]
```

1.5 Looks like you can do tableRetrieve, tableInsert, and tableDelete very quickly.

1.5.1 Just let the address calculator point to where the item should go

1.5.2 And the amount of time would depend on how quickly the address calculator can calculate

1.5.3 Need to implement the address calculator with very little work

1.5.3.1 Not that difficult – many exist that approximate the ideal address calculator

1.5.3.2 The address calculator is called a **hash function**

1.5.3.3 The technique is called **hashing**

1.5.3.4 The array table is called the **hash table**

1.6 Let's try an example using the 911 emergency system

1.6.1 You could have a person's telephone number in a search tree but that takes time

1.6.1.1 But you could have a system that stores seven digit number in a table 10 million long

1.6.1.2 For number 1234567 you could directly access the table like table[1234567]

1.6.2 But 911 systems are local and thus you don't need such a huge table.

1.6.2.1 Could only use the right most 4 digits and thus access the table like table[4567]

1.6.2.2 That is only 10,000 records

1.6.2.3 Saves space and this is a simple example of a hash function: dropping the top 3 digits

1.6.2.4 $h(x) = i$, where i is an integer in the range of 0 through 9999

1.6.3 But with a local exchange, the table would be full

1.6.4 That is not always the case with other hash applications

1.6.4.1 Air traffic control system that uses 4 digit flight number

1.6.4.2 That would have a table of 10,000 records

1.6.4.3 However, not that many flights are active (like 50)

1.6.4.4 Would waste a lot of memory

1.6.5 A different hash function would save memory

1.6.5.1 Allow space for a max of 101 flights, index from 0 to 100

1.6.5.2 Function should map into that range

1.6.5.3 Call hash function h

1.6.5.4 There are several that can be used (later)
1.6.5.5 For example:

i = the array index that the address calculator gives you for an item whose search key equals searchKey

or:

i = h(searchKey)

1.6.5.6 The above would be for a searchKey

1.7 Is hashing as good as it looks?
1.7.1 If so then why develop all the other methods?
1.7.2 Hashing is not as simple as it sounds
1.7.2.1 It would have the same problem as other fixed size implementations
1.7.2.2 Hash table has to be big enough
1.7.2.3 But, later, we'll see a more dynamic implementation
1.7.3 Ideally you want hash function to map each x into a unique integer i
1.7.3.1 That would be called a **perfect hash function**
1.7.3.2 It is possible to construct it if you know ALL the possible search keys
1.7.3.3 That would work with 911 since everyone is in database but not with traffic control
1.7.4 In practice, a hash function can map two or more search keys into the same integer.
1.7.4.1 That would be called a **collision**
1.7.4.2 Even with less that 101 items, h could tell you to place more than one item into the same array location
1.7.4.3 For example, two items with search keys 4567 and 7597 and map to the same location like so:
h(4567) = h(7597) = 22
1.7.4.4 Search keys 4567 and 7597 have collided
1.7.5 The only way to avoid collisions is for the hash table to be large enough to hold each possible search key
1.7.5.1 Social security numbers search keys would need a range from 000000000 to 999999999
1.7.5.2 That would be a lot of storage
1.7.5.3 So reserving vast amounts of storage is not practical
1.7.5.4 Collision resolution schemes are needed to make hashing feasible – hopefully to have items placed evenly throughout the has table

1.8 Hash function must:
1.8.1 Be easy and fast to compute
1.8.2 Place items evenly throughout the hash table

1.9 There are several hash function and **collision-resolution schemes**

2 We'll assume that the hash function uses an arbitrary integer as an argument

2.1 If the original value is a string, it can be mapped into an integer which can be used

2.2 There are many ways to convert an arbitrary integer into a range, like 0 to 100

2.2.1 Thus there are many hash functions

2.2.2 But many would not be suitable

2.3 Here are several single hash functions:

2.3.1 **Selecting digits:** if your search key is the nine-digit employee ID number 001364825 you could select the fourth digit and the last digit to get 35 as an index:

h(001364825) = 35 (select the fourth and last digits)

2.3.1.1 Therefore store the item whose search key is 001364825 is table[35]

2.3.1.2 But be careful which digits you choose

2.3.1.3 With social security numbers, if you choose the first 3 digits you could map all of a group into the same location since those digits are based on geographical location

2.3.1.4 Digit selection is simple and fast but generally they do not evenly distribute the items in the hash table

2.3.1.5 Hash function should really use all the digits

2.3.2 **Folding:** one way to improve on the previous method of selecting digits is to add all the digits

2.3.2.1 For example, add all digits in 001364825 to get:

0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29 (add the digits)

2.3.2.2 But if you add the digits, you will only get a range of 0 <= h(search key) <= 81

2.3.2.3 That would only use the table[0] to table[81]

2.3.2.4 One way to increase the range is to add group of digits, for example:

001 + 364 + 825 = 1,190
so the range would be:
0 <= h(search key) <= 3 * 999 = 2,997

2.3.2.5 But if 2,997 is too large, you can alter the groups as you wish

2.3.2.6 You can even use more than one hash function at a time: take the 2,997 then add 29 + 97

2.3.3 **Modulo arithmetic:** that provides a simple and efficient hash function – will continue to use

h(x) = x mod tableSize

2.3.3.1 If the table size is 101, the hash function will map into 0 to 100

2.3.3.2 So h maps 001364825 into 12

2.3.3.3 But h(x) = x mod tableSize would have many x's map into table[0] and many x's map into table[1] and so on

2.3.3.4 But you can reduce collisions by using a prime numbers for the table size – 101 is a prime number (but a bit small for a typical table)

2.4 Converting a character into a string

2.4.1 Convert a string into an integer then use the hash function

2.4.1.1 Take "NOTE" and use the ASCII values of 78, 79, 84, and 69

2.4.1.2 Or could use 1 through 26 for the letters then "NOTE" would be 14, 15, 20, and 5 for N, O, T, and E

2.4.2 If you add all the values you will get an integer but not necessarily a unique one ("TONE" would also have the same value)

2.4.3 You could use binary values then concatenate them like so:

2.4.3.1 N is 14, or 01110 in binary

2.4.3.2 O is 15, or 01111 in binary

2.4.3.3 T is 20, or 10100 in binary

2.4.3.4 E is 5, or 00101 in binary

2.4.3.5 Concatenating the binary would give the binary integer of:

01110011111010000101

2.4.3.6 That would be the integer 474,757 in decimal

2.4.3.7 You can use the hash function on it

2.4.4 Or you could use a formula to evaluate the characters like so:

2.4.4.1 $14 * 32^3 + 15 * 32^2 + 20 * 32^1 + 5 * 32^0$

2.4.4.2 Each character can fit into 5 bits so that 2^5 or can hold 32 values

2.4.4.3 You can factor it and minimize arithmetic values

2.4.4.4 ((14 * 32 + 15) * 32 + 20) * 32 + 5

2.4.4.5 Could have an overflow, depending on the computer

2.4.4.6 But could apply the hash function after computing each parenthesized expression

3 Resolving collisions

3.1 If you have a search key is 4567 then h(x) = x mod 101 would give 22

3.1.1 But if table[22] already has a value then there is a collision

3.1.2 Don't want to abort the insert (can have a collision with just one item in the table)

3.2 Two general approaches to collision resolution are common

3.2.1 One approach allocates another location within the hash table to the new item

3.2.2 Second approach changes the structure of the hash table so that each location can have more than one item

3.3 **Approach 1: Open addressing**

3.3.1 On collision, you probe for another empty or open slot

3.3.2 Figure 12-45 shows the collision

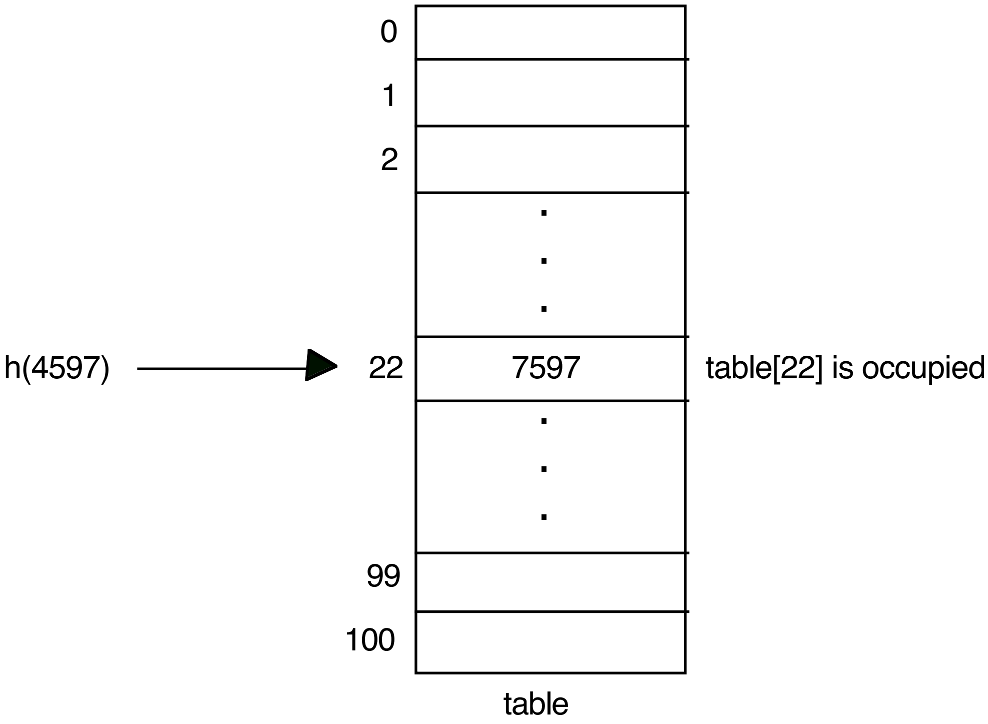


Figure 12-45, A collision

3.3.3 Now probe for another empty or open location

3.3.4 That would be called **probe sequence**

3.3.5 It is also called **open addressing**

3.3.5.1 Have to be able to find a table item after you inserted it

3.3.5.2 tableDelete and tableRetrieve must be able to duplicate the probe

3.4 **Linear probing:** search the table sequentially after a collision

3.4.1 if table[h(searchKey)] is occupied, check table[h(searchKey)+1], then table[h(searchKey)+2] and so on until you find an empty slot

3.4.2 You need to "wrap around" from 0 to 100 back to 0 (use the mod)

3.4.3 Fig 12-46 shows it

3.4.4 For insert and retrieve, it is straightforward

3.4.5 Delete has some complications

3.4.5.1 The finding the item and deleting is straightforward

3.4.5.2 But you need to take care of the gap since retrieve would stop the linear probing with an empty slot

3.4.5.3 You could use three states: occupied, empty (not been used), deleted (had been used)

3.4.5.4 Then retrieve would keep on going with a deleted cell

3.4.5.5 Insert could insert into an empty or deleted slot

3.4.6 There is also a problem with a **cluster**

3.4.6.1 You get items clustered around an item (primary clustering)

3.4.6.2 Clusters can get close to each other

3.4.6.3 Large clusters can get even larger (“rich get richer”)

3.4.6.4 Large clusters increase linear probing and thus decrease efficiency

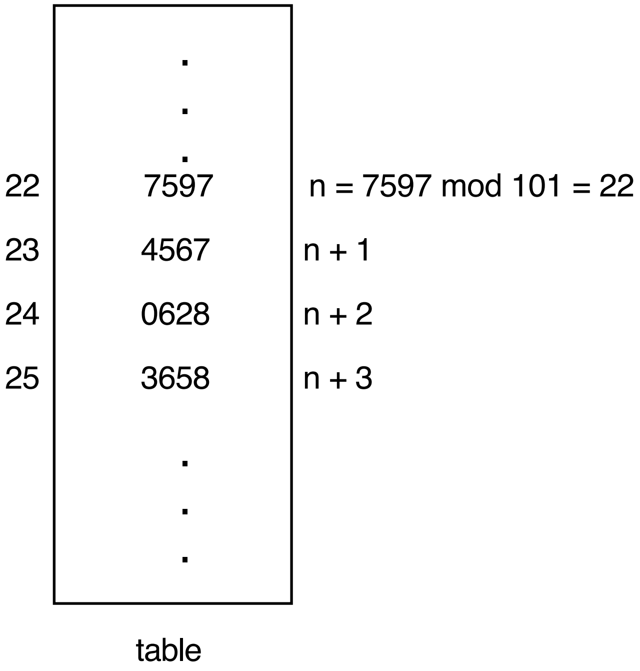


Figure 12-46, Linear probing with h(x) = x mod 101

3.5 Quadratic probing

3.5.1 Eliminate primary clusters by adjusting linear probing to quadratic

3.5.2 Instead of probing consecutive table locations from the original has location table[h(searchKey)] you:

3.5.2.1 Check locations table[h(searchKey) + 1^2], table[h(searchKey) + 2^2], table[h(searchKey) + 3^2], and so on until you find an available location

3.5.2.2 Figure 12-47 shows this opening addressing scheme

3.5.2.3 Unfortunately, when two items hash into the same location, quadratic probing uses the same probe sequence for each item

3.5.2.4 That's called *secondary clustering* – though it doesn't seem to be that much of a problem according to research

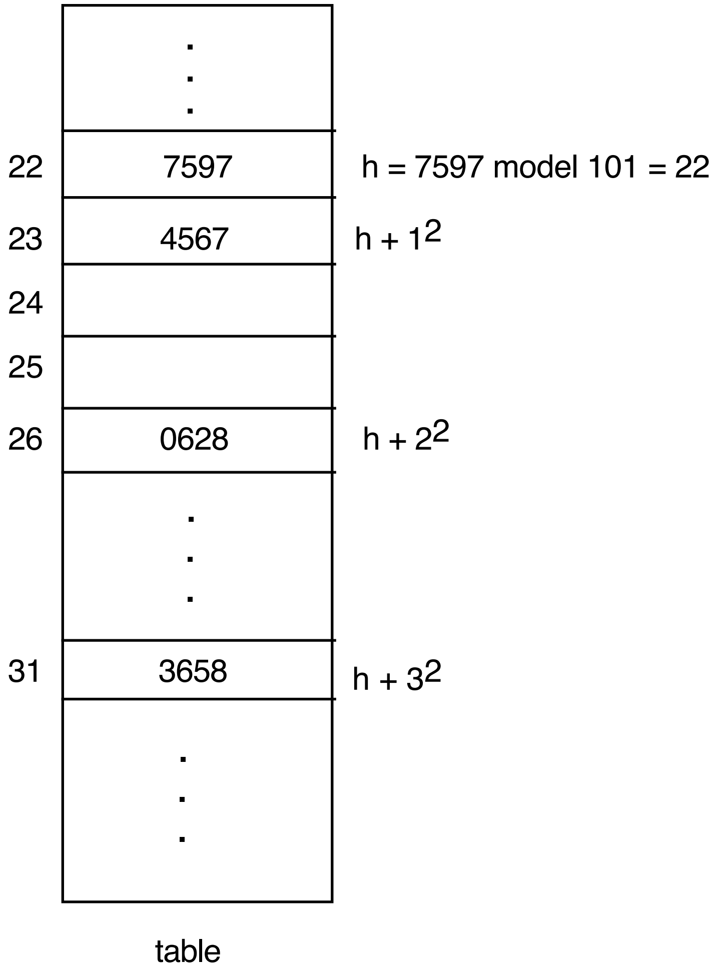


Figure 12-47. Quadratic probing with h(x) = x mod 101

3.6 Double hashing

3.6.1 Another open addressing that drastically reduces clustering

3.6.2 Probe sequences that both linear and quadratic probing use are *key independent*

3.6.2.1 For example, linear probing inspects the table locations sequentially no matter what the hash key is

3.6.2.2 However, double hashing defines *key-dependent* probe sequences

3.6.2.3 This scheme the probe sequence still searches the table in a linear order, starting at location h_1(key), but a second hash function h_2 determines the size of the steps taken

3.6.3 Although you choose h_1 as usual, must follow these guidelines for h_2:

$h_2(key) \neq 0$
$h_2 \neq h_1$

3.6.4 Need a nonzero step size h_2(key) to define the probe sequence – also h_2 must differ from h_1 to avoid clustering

3.6.5 For example, let h_1 and h_2 be the primary and secondary hash functions defined as:

h_1(key) = key mod 11
h_2(key) = 7 – (key mod 7)

3.6.5.1 Where a hash table of only 11 items is assumed so you can see the effect of these functions on hash table

3.6.5.2 If key = 58, h_1 hashes key to table location 3 (58 mod 11) and h_2 indicates that the probe sequence will be 3, 8, 2 (wraps around), 7, 1 (wraps around), 6, 0, 5, 10, 4, 9

3.6.5.3 However, if key = 14, h_1 hashes key to table location 3 (14 mod 11), and h_2 indicates that the probe sequence should take steps of size 7 (7 – 14 mod 7) – the probe sequence would be 3, 10, 6, 2, 9, 5, 1, 8, 4, 0

3.6.6 Each probe sequences visits all the table locations

3.6.6.1 It always occurs if size of table and size of probe step are relatively prime (greatest common divisor is 1)

3.6.6.2 Since the size of the hash table is commonly a prime number, it will be relatively prime to all step sizes

3.6.7 Figure 12-48 illustrates insertion of 58, 14, and 91 into initially empty hash table

3.6.7.1 Since $h_1(58)$ is 3, you place 58 into table[3]

3.6.7.2 Then find that $h_1(14)$ is also 3, so to avoid a collision, step by $h_2(14) = 7$ and place 14 into table[3 + 7] or table [10]

3.6.7.3 Finally $h_1(91)$ is 3 and $h_2(91)$ is 7

3.6.7.4 Since table[3] is occupied, probe table[10] and find that it is also occupied

3.6.7.5 Finally store 91 in table[(10 + 7) % 11] or table [6]

3.6.7.6 Using more than one hash function is called rehashing – though having more than 2 hash functions can be difficult

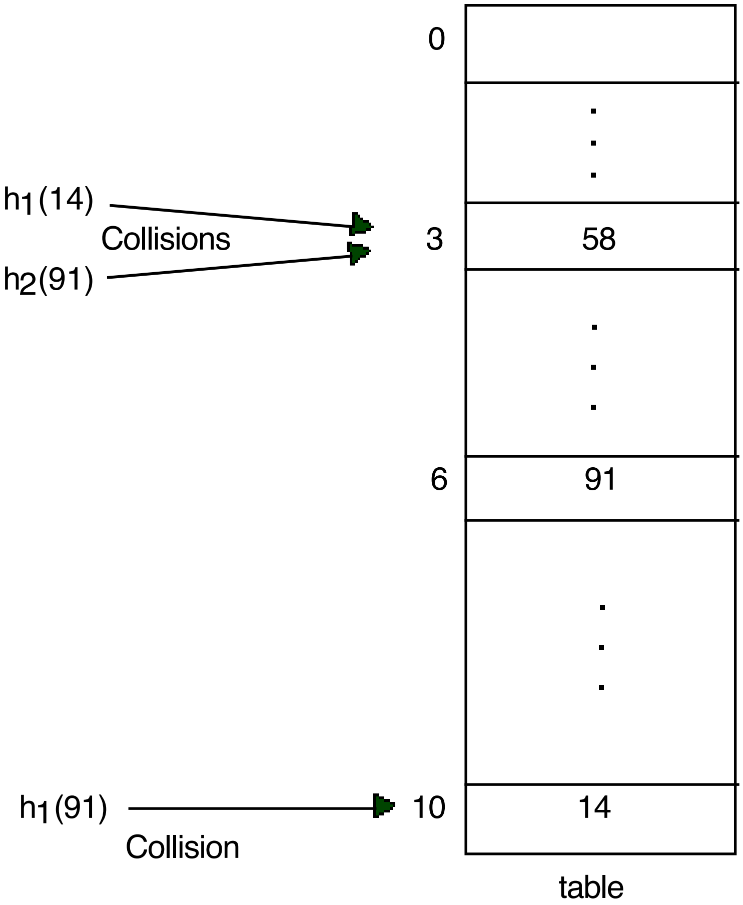


Figure 12-48. Double hashing during insertion of 58, 14, and 91

3.7 Approach 2: restructure the hash table

3.7.1 Another way to resolve collisions is to change the structure of the array table – hash table – so that it can accommodate more than one item in the same location (there are two ways to do that)

3.7.2 First way is to add **buckets**

3.7.2.1 Define the hash table *table* so that each location table[i] is in itself an array called a **bucket**, then you can store items that hash into table[i] in this array

3.7.2.2 The problem is choosing the size B of the bucket.

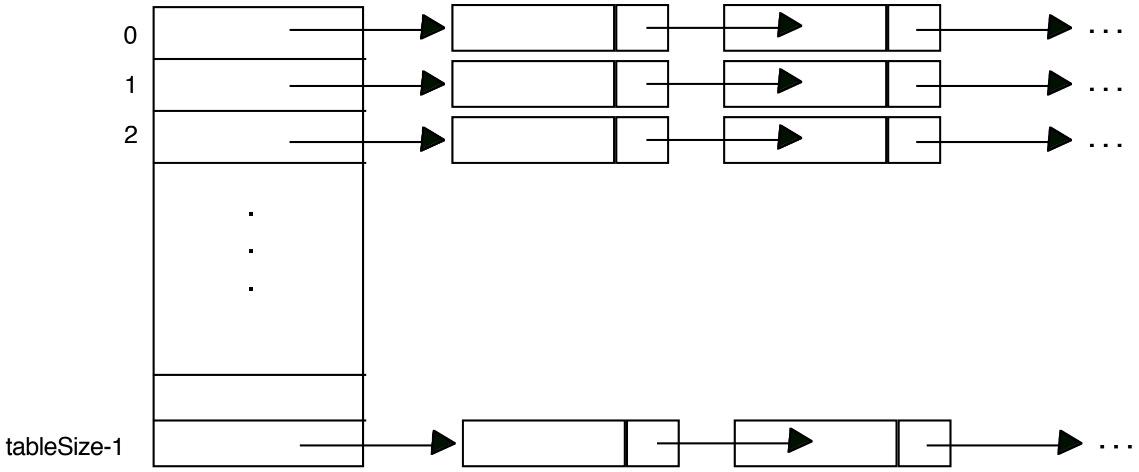
3.7.2.2.1 If too small, you have only postponed the problem of collisions

3.7.2.2.2 If you make it too big, you waste memory

3.7.3 Second way is to do **separate chaining**

3.7.3.1 A better approach is to have the hash table as an array of linked lists

3.7.3.2 This is known as separate chaining, each entry table[i] is a pointer to the linked list – the **chain** – of items that the hash function has mapped into location i, like Figure 12-49 shows:



Each location of the hash table contains a pointer to a linked llist

Figure 12-49, Separate chaining

3.7.3.3 The following classes for the ADT table assume an implementation that uses a hash table

```
/** %file TableH.h */
#include "TableException.h"
#include "ChainNode.h"

typedef KeyedItem TableItemType;

/** ADT table.
 * Hash table implementation.
 * Assumption: A table contains at most one item with a
 * given search key at any time. */

class HashTable
{
public:
// constructors and destructor:
  HashTable ( ) ;
  HashTable(const HashTable& table);
  virtual -HashTable();

// table operations:
  virtual bool tablelsEmpty() const; virtual int tableGetLength() const;
  virtual void tablelnsert(const TableItemType& newItem) throw(TableException);
  virtual void tableDelete(KeyType searchKey) throw(TableException);
  virtual void tableRetrieve(KeyType searchKey, TableItemType& tableItem) const
                                                      throw(TableException);
protected:
  int hashlndex(KeyType searchKey) const;     // Hash function
private:
   static const int HASH_TABLE_SIZE = 101;      // Size of hash
                                                 // table
   typedef ChainNode * HashTableType[HASH_TABLE_SIZE];

   HashTableType table;                          // Hash table

  int   size;                                    // size of ADT table

} ;   // end HashTable

// End of header file.

/** @file KeyedItem.h
 * Provides basis for classes that need a search key value. */
typedef desired-type-of-search-key KeyType;

class KeyedItem
{
public:
   KeyedItem () {}
   KeyedItem(const KeyType& keyValue) : searchKey(keyValue) {}
   KeyType getKey() const            // returns search key
  { return searchKey;
  } // end getKey

private:
   KeyType searchKey;
};  // end KeyedItem


/** @file ChainNode.h.
 * Provides the chain node definition for the hash table. */
#include "KeyedItem.h"

class ChainNode
{
private:
   ChainNode(const KeyedItem & nodeItem, ChainNode *nextNode = NULL)
                      : item(nodeItem), next (nextNode) {}
   KeyedItem item;
   ChainNode *next;

   friend class HashTable;
};  // end ChainNode
```

3.8 The class KeyedItem can be used as the base class for the items that are stored in the table.

3.8.1 The KeyedItem class was first presented in Chapter 10 and provides a data field for the search key.

3.8.2 The search key is used by the hashIndex method in the class HashTable to generate a hash index value.

3.8.3 When you insert a new item into the table, you place it at the beginning of the linked list that the hash function indicates.

3.8.4 The following pseudocode describes the insertion algorithm:

```
tableInsert(in newItem:TableItemType)
          throw TableException

    searchKey = the search key of newItem
    i = hashIndex(searchKey)
    p = pointer to a new node
    Throw TableException according to whether the
            previous memory allocation is successful
    p->item = newItem
    p->next = table[i]
    table[i] = p
```

3.9 When you want to retrieve an item, you search the linked list that the hash function indicates.

3.9.1 The following pseudocode describes the retrieval algorithm:

```
tableRetrieve(in searchKey:KeyType,
              out tableItem:TableItemType)
              throw TableException

    i = hashIndex(searchKey)
    p = table[i]

    while ( (p != NULL) &&
            (p->item.getKey() != searchKey) )
            p = p->next

    if (p == NULL)
            Throw a TableException
    else
            tableItem = p->item
```

3.9.2 The deletion algorithm is very similar to the retrieval algorithm and is left as an exercise. (See Exercise 14.)

3.9.3 Separate chaining is thus a successful approach to resolving collisions.

3.9.3.1 With separate chaining, the size of the ADT table is dynamic and can exceed the size of the hash table, because each linked list can be as long as necessary.

3.9.3.2 As you will see in the next section, the length of these linked lists affects the efficiency of retrievals and deletions.

4 The Efficiency of Hashing

4.1 An analysis of the average-case efficiency of hashing involves the load factor a, which is the ratio of the current number of items in the table to the maximum size of the array table.

4.1.1 That is,

$$\alpha = \frac{\text{Current number of table items}}{\text{tableSize}}$$

4.1.2 α is a measure of how full the hash table table is.

4.1.2.1 As table fills, α increases and the chance of collision increases, so search times increase.

4.1.2.2 Thus, hashing efficiency decreases as a increases.

4.1.3 Unlike the efficiency of earlier table implementations, the efficiency of hashing does not depend solely on the number N of items in the table.

4.1.4 While it is true that for a fixed tableSize, efficiency decreases as N increases, for a given N you can choose tableSize to increase efficiency.

4.1.5 Thus, when determining tableSize, you should estimate the largest possible N and select tableSize so that α is small.

4.1.6 As you will see shortly, α should not exceed 2/3.

4.2 Hashing efficiency for a particular search also depends on whether the search is successful.

4.2.1 An unsuccessful search requires more time in general than a successful search.

4.2.2 The following analyses enable a comparison of collision resolution techniques.

4.2.3 Linear probing.

4.2.3.1 For linear probing, the approximate average number of comparisons that a search requires is

$$\frac{1}{2}\left[1 + \frac{1}{1-\alpha}\right] \quad \text{for a successful search, and}$$

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]^2$$ for a unsuccessful search

4.2.3.2 As collisions increase, the probe sequences increase in length, causing increased search times.

4.2.3.3 For example, for a table that is two-thirds full ($\alpha = 2/3$), an average unsuccessful search might require at most five comparisons, or probes while an average successful search might require at most two comparisons.

4.2.3.4 To maintain efficiency, it is important to prevent the hash table from filling up.

4.3 Quadratic probing and double hashing.

4.3.1 The efficiency of both quadratic probing and double hashing is given by

$$\frac{-\log_e(1-\alpha)}{\alpha}$$ for a successful search, and

$$\frac{1}{1-\alpha}$$ for an unsuccessful search

4.3.2 On average, both techniques require fewer comparisons than linear probing.

4.3.2.1 For example, for a table that is two-thirds full, an average unsuccessful search might require at most three comparisons, or probes, while an average successful search might require at most two comparisons.

4.3.2.2 As a result, you can use a smaller hash table for both quadratic probing and double hashing than you can for linear probing.

4.3.2.3 However, because they are open-addressing schemes, all three approaches suffer when you are unable to predict the number of insertions and deletions that will occur.

4.3.2.4 If your hash table is too small, it will fill up, and search efficiency will decrease.

4.4 Separate chaining.

4.4.1 Because the tableInsert operation places the new item at the beginning of a linked list within the hash table, it is 0(1).

4.4.2 The table Retrieve and tableDelete operations, however, are not as fast.

4.4.3 They each require a search of the linked list of items, so ideally you would like for these linked lists to be short.

4.4.4 For separate chaining, tableSize is the number of linked lists, not the maximum number of table items.

4.4.4.1 Thus, it is entirely possible, and even likely, that the current number of table items N exceeds tableSize.

4.4.4.2 That is, the load factor α, or N/tableSize, can exceed 1.

4.4.4.3 Because tableSize is the number of linked lists, N/tableSize – that is, α – is the average length of each linked list.

4.4.5 Some searches of the hash table are unsuccessful because the relevant linked list is empty.

4.4.5.1 Such searches are virtually instantaneous.

4.4.5.2 For an unsuccessful search of a nonempty linked list, however, tableRetrieve and tableDelete must examine the entire list, or a items in the average case.

4.4.5.3 On the other hand, a successful search must examine a nonempty linked list.

4.4.5.4 In the average case, the search will locate the item in the middle of the list.

4.4.5.5 That is, after determining that the linked list is not empty, the search will examine $\alpha/2$ items.

4.4.5.6 Thus, the efficiency of the retrieval and deletion operations under the separate-chaining approach is

$1 + \alpha/2$ for a successful search, and

α for an unsuccessful search

4.4.6 Even if the linked lists typically are short, you should still estimate the worst case.

4.4.6.1 If you seriously underestimate tableSize or if most of the table items happen to hash into the same location, the number of items in a linked list could be quite large.

4.4.6.2 In fact, in the worst case, all N items in the table could be in the same linked list!

4.4.6.3 Time that a retrieval or deletion operation requires can range from almost nothing – if the linked list to be searched either is empty or has only a couple of items in it – to the time required to search a linked list that contains all the items in the table, if all the items hashed into the same location.

4.5 Comparing techniques.

4.5.1 Figure 12-50 plots the relative efficiency of the collision-resolution schemes just discussed.

4.5.1.1 When the hash table *table* is about half full-that is, when α is 0.5-the techniques are nearly equal in efficiency.

4.5.1.2 As the table fills and a approaches 1, separate chaining is the most efficient.

4.5.1.3 Does this mean that we should discard all other search algorithms in favor of hashing with separate chaining?

4.5.1.4 No. The analyses here are average-case analyses.

4.5.1.4.1 Although an implementation of the ADT table that uses hashing might often be faster than one that uses a search tree, in the worst case it can be much slower.

4.5.1.4.2 If you can afford both an occasional slow search and a large tableSize-that is, a small a-then hashing can be an attractive table implementation.

4.5.1.4.3 But if performing a life-and-death search for your city's poison control center, a search-tree implementation would at least provide you with a guaranteed bound on its worst-case behavior.

4.5.1.4.4 Furthermore, while separate chaining is the most time-efficient collision resolution scheme, you do have the storage overhead of the pointers in the linked list.

4.5.1.4.5 If the data records in the table are small, the pointers add a significant overhead in storage, and you may want to consider a simpler collision-resolution scheme.

4.5.1.4.6 But, if the records are large, the addition of a pointer is insignificant, so separate chaining is a good choice.

4.6 What Constitutes a Good Hash Function?

4.6.1 Before we conclude this introduction to hashing, consider in more detail the issue of choosing a hash function to perform the address calculations for a given application.

4.6.2 This section will present brief summary of the major concerns.

4.6.2.1 • A hash function should be easy and fast to compute.

4.6.2.1.1 If a hashing scheme is to perform table operations almost instantaneously and in constant time, you certainly must be able to calculate the hash function rapidly.

4.6.2.1.2 Most of the common hash functions require only a single division (like the modulo function), a single multiplication, or some kind of "bit level" operation on the internal representation of the search key.

4.6.2.1.3 In these cases, the requirement that the hash function be easy and fast compute is satisfied.

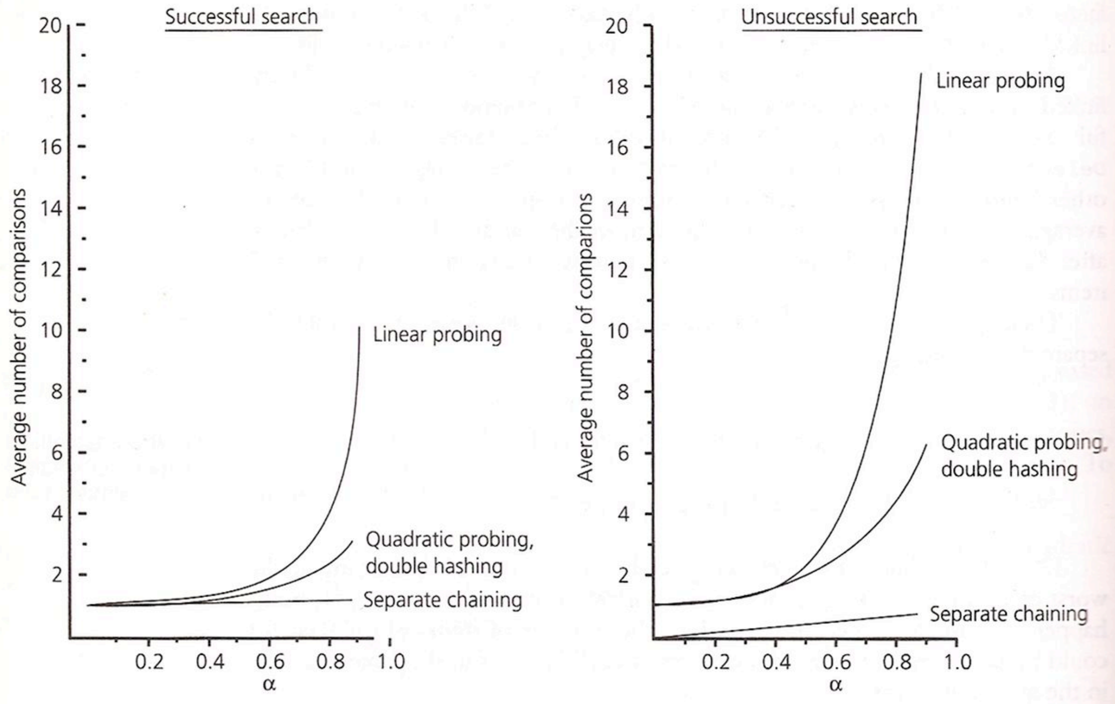


FIGURE 12-50

The relative efficiency of four collision-resolution methods

4.6.2.2 • A hash function should scatter the data evenly throughout the hash table.

4.6.2.2.1 Unless you use a perfect hash function (usually impractical to make) you typically cannot avoid collisions entirely.

4.6.2.2.2 For example, to achieve the best performance from a separate-chaining scheme, each entry *table[i]* should contain approximately the same number of items in its chain; that is, each chain should contain approximately *N/tableSize* items (and thus no chain should contain significantly more than *N/tableSize* items).

4.6.2.2.3 To accomplish this goal, your hash function should scatter the search keys evenly throughout the hash table.

4.6.3 There are two issues to consider with regard to how evenly a hash function scatters the search keys.

4.6.3.1 • How well does the hash function scatter random data?

4.6.3.1.1 If every searchkey value is equally likely, will the hash function scatter the search keys evenly?

4.6.3.1.2 For example, consider the following scheme for hashing nine-digit ID numbers:

table[0 .. 39] is the hash table, and
the hash function is h(x) = (first two digits of x) mod 40

4.6.3.2 Does a given ID number x have equal probability of hashing into anyone of the 40 array locations?

4.6.3.2.1 For this hash function, the answer is no.

4.6.3.2.2 Only ID numbers that start with 19, 59, and 99 map into table[19] , while only ID numbers that start with 20 and 60 map into table [20].

4.6.3.2.3 In general, three different ID prefixes-that is, the first two digits of an ID number-map into each array location 0 through 19) while only two different prefixes map into each array location 20 through 39.

4.6.3.2.4 Because all ID numbers are equally likely-and thus all prefixes 00 through 99 are equally likely-a given ID number is 50 percent more likely to hash into one of the locations 0 through 19 than it is to hash into one of the locations 20 through 39.

4.6.3.2.5 As a result, each array location 0 through 19 would contain, on average, 50 percent more items than each location 20 through 39.

Thus, the hash function

h(x) = (first two digits of x) mod 40

4.6.3.2.6 does not scatter random data evenly throughout the array table [0 •• 39].

4.6.3.2.7 On the other hand, it can be shown that the hash function

h(x) = x mod 101

4.6.3.2.8 does, in fact, scatter random data evenly throughout the array table[0 .. 100].

4.6.3.3 • How well does the hash function scatter nonrandom data?

4.6.3.3.1 Even if a hash function scatters random data evenly, it may have trouble with nonrandom data.

4.6.3.3.2 In general, no matter what hash function you select, it is always possible that the data will have some unlucky pattern that will result in uneven scattering.

4.6.3.3.3 Although there is no way to guarantee that a hash function will scatter all data evenly, you can greatly increase the likelihood of this behavior

As an example, consider the following scheme:

table [0 .. 99] is the hash table, and
the hash function is h(x) = first two digits of x

4.6.3.4 If every ID number is equally likely, h will scatter the search keys evenly throughout the array.

4.6.3.5 But what if every ID number is not equally likely?

4.6.3.6 For instance, a company might assign employee IDs according to department, as follows:

10xxxxx	Sales
20xxxxx	Customer Relations
…	
90xxxxx	Data Processing

4.6.3.7 Under this assignment, only 9 out of the 100 array locations would contain any items at all.

4.6.3.7.1 Further, those locations corresponding to the largest departments (Sales, for example, which corresponds to table[l0] would contain more items than those locations corresponding to the smallest departments.

4.6.3.7.2 This scheme certainly does not scatter the data evenly.

4.6.3.7.3 Research has been done into the types of hash functions that you should use to guard against various types of patterns in the data.

4.6.3.8 The results of this research are really in the province of more advanced courses but two general principles can be noted here:

1. The calculation of the hash function should involve the entire search key. Thus, for example, computing a modulo of the entire ID number is much safer than using only its first two digits.

2. If a hash function uses modulo arithmetic, the base should be prime; that is, if h is of the form

h(x) = x mod tableSize

then tableSize should be a prime number.
This selection of tableSize is a safeguard against many subtle kinds of patterns in the data (for example, search keys whose digits are likely to be multiples of one another).
Although each application can have its own particular kind of patterns and thus should be analyzed on an individual basis, choosing tableSize to be prime is an easy way to safeguard against some common types of patterns in the data.

4.7 Table Traversal: An Inefficient Operation Under Hashing

4.7.1 For many applications, hashing provides the most efficient implementation of the ADT table.

4.7.2 One important table operation-traversal in sorted order-performs poorly when hashing implements the table.

4.7.2.1 A good hash function scatters items as randomly as possible throughout the array, so that no ordering relationship exists between the search keys that hash into table[i] and those that hash into table [i + 1] .

4.7.2.2 So, if you must traverse the table in sorted order, you first would have to sort the items.

4.7.2.3 If sorting were required frequently, hashing would be a far less attractive implementation than a search tree.

4.7.3 Traversing a table in sorted order is really just one example of a whole class of operations that hashing does not support well.

4.7.3.1 Many similar operations that you often wish to perform on a table require that the items be ordered.

4.7.3.2 For example, consider an operation that must find the table item with the smallest or largest value in its search key.

4.7.3.3 If you use a search-tree implementation, these items are in the leftmost and rightmost nodes of the tree, respectively.

4.7.3.4 If you use a hashing implementation, however, you do not know where these items are-you would have to search the entire table.

4.7.3.5 A similar type of operation is a range query, which requires that you retrieve all items whose search keys fall into a given range of values.

4.7.3.6 For example, you might want to retrieve all items whose search keys are in the range 129 to 755.

4.7.3.7 This task is relatively easy to perform by using a search tree (see Exercise 3), but if you use hashing, there is no efficient way to answer the range query.

4.7.4 In general, if an application requires any of these ordered operations, you should probably use a search tree.

4.7.4.1 Although the tableRetrieve, tableInsert, and tableDelete operations are somewhat more efficient when you use hashing to implement the table instead of a balanced search tree, the balanced search tree supports these operations so efficiently itself that, in most contexts, the difference in speed for these operations is negligible

4.7.4.2 (Whereas the advantage of the search tree over hashing for the ordered operations is significant).

4.7.4.3 In the context of external storage, however, the story is different.

4.7.4.4 For data that is stored externally, the difference in speed between hashing's implementation of tableRetrieve and a search tree's implementation may well be significant, as you will see in Chapter 14.

4.7.4.5 In an external setting, it is not uncommon to see a hashing implementation of the tableRetrieve operation and a search-tree implementation of the ordered operations used simultaneously.