

# Chapter 20

## Hash Tables

### Dictionary

- All elements have a unique key.
- Operations:
  - Insert element with a specified key.
  - Search for element by key.
  - Delete element by key.
- Random vs. sequential access.
- If we have duplicate keys, need rule to resolve ambiguity.
- What ways could you use to implement a dictionary? (Name in class)
  - Ordered array
  - Unordered array
  - Ordered linked list
  - Binary search tree
- Let's use our newly acquired analysis skills to determine which of the following is better. Let's analyze them for two operations – insert and search. This is shown in Table 1.

Method	Insert time	Search Time
Ordered array	$n/2$	$\log n$
Unordered array	1	$n/2$
Ordered linked list	$n/2$	$n/2$
Binary search tree	$\log n$	$\log n$

Table 1 Search time for dictionaries

- Which you pick depends on the relative frequency of insertions and searches.
- What if  $\log n$  is too long?
- A new method we will now consider deals with an attempt to improve on this bound.

### Hashing

- Implementing the dictionary takes two parts:
  - *Hash table* – table where the elements of the dictionary are placed. This table is not sorted, as have other tables previously mentioned.
  - *Hash function* – function that maps a key to a location in the hash table.

### Hash Functions

- How hard is it to find a good hash function?

- Consider the birthday paradox – the chances of two people in a room having the same birthdates?
  - 23 people – 50%
  - 60 people – greater than 99%
- Characteristics of a good hash function.
  - Ideally, insertion and deletion are  $O(1)$  operations.
  - Give some sort of randomization in locations in the table.
- Techniques used in hashing:
  - *Selecting digits* – ignore part of the key.
  - *Folding* – take the exclusive or of two sections of the key and combine. Add digits.
  - *Modular arithmetic* –  $\text{HASH}(\text{key}) = \text{key} \% \text{TABLESIZE}$ . If the table size is prime, we get better spread.
  - *Multiplication* –  $\text{key}^2$  and take middle bits: *mid-square technique*.

### Hash Function 1 – Selecting Digits

- We want to actually manipulate the bits of the key.
- We can treat the characters of the key as numbers. When we add them, we actually get their ASCII value (A = 65, etc.).
- See Figure 1 for the code.
- This is bad because the return value will not get very large, so doesn't try to use all slots in a large table.
- Deletion can be problem – why?
  - Have to use lazy deletion because you can't have empty slots.

```

unsigned int hash(string key)
{unsigned int hashVal = 0;
 for (int i=0; i < key.length(); i++)
   hashVal += key[i];
 return hashVal % TABLESIZE;
}

```

Figure 1 Selecting Digits

### Hash Function 2 – Folding

- See Figure 2 for the code.
- This uses a multiply of 128 to shift the value of hashVal seven positions to the left. (Since this is base two, a multiply by two is a shift of one. A multiply by 128 is a shift of seven.)
- This is better as it uses a broader range of values.

```

string key;
for (i=0; i < key.length(); i++)
  hashVal = (hashVal * 128 +key[i])
            % TABLESIZE;
return hashVal;

```

Figure 2 Folding

### Hash Function 3

- See Figure 3 for the code.
- Faster and takes advantage of allowed overflow.

```

string key;
unsigned int hashVal=0;
for (i=0; i < key.length(); i++)
  hashVal = (hashVal << 5) ^ key[i]
            ^ hashVal;
return hashVal %TABLESIZE;

```

Figure 3 Exclusive or

- Uses 5 instead of 7 as shift, so slows down shifting of early characters.
- The use of the shift operator << may allow the operation to be more efficient.
- The caret (^) is an *exclusive or*. Exclusive ors work at the bit level in the following way:  

```

10110110
11001100
-----
01111010

```
- If corresponding bits are the same, a zero is recorded. If corresponding bits are different, a one is recorded.
- Since we use or to put the original hashVal back in, we aren't in danger of losing the effect of the first characters.

## Collision Resolution

- If  $\text{HASH}(\text{key1}) == \text{HASH}(\text{key2})$ , a collision occurs.
- The text says – collision may not be a problem if there are multiple keys per location.
  - The book talks about overflow, when there is no more room.
  - We are assuming a collision causes overflow (one element per location).
- Thus, we must plan for collisions! If the location is already taken, we must be able to use another.
- *Probes* number of memory locations that must be examined in order to determine the location of a key in the table.

## Linear Probing

- Use a vacant spot in table following  $\text{HASH}(\text{key})$  – “open” addressing means that we find an unused address and use it.
- Evaluation
  - Uses no pointer space
  - Longer search times
  - Clustering gets worse and worse (snowballs).
    - **Primary clustering** is when two keys that hash onto different values compete for same locations in successive hashes.
  - Better if successive probe locations are picked in scrambled order to lessen clustering.
- For example:  $\text{HASH}(\text{key}) = \text{key} \% 7$  with the following data: 5 19 3 108 72

## Quadratic Probing

- Assume the value of the hash function is  $H = \text{HASH}(i)$ . Cells are probed according to  $H + 1^2, H + 2^2, H + 3^2, \dots, H + i^2$ .
- Will not probe every location.
- Can do without multiplication – see the book for more information
- **Secondary clustering** is when different keys that hash to same locations compete for successive hash locations.
- Quadratic probing eliminates primary clustering, but not secondary clustering.

## Double Hashing

- For both linear and quadratic probing, the sequences checked are key independent.
- Have two functions, *Step* gives the increment for *Hash*.
- Let  $\text{Step}(\text{key}) = 1 + \text{key} \% (\text{TABLESIZE} - 2)$  – gives personalized increment.
- **Notice, the location of Step is never used directly as the hash value. It is only an auxiliary function.**
- If TABLESIZE and TABLESIZE – 2 are primes, it works better.
- $\text{Hash}(\text{key}) = \text{key} \% \text{TABLESIZE}$
- If  $\text{Hash}(\text{key})$  is full, successively add increment as specified by Step.
- For example for key = 38 and TABLESIZE = 13
  - $= 1 + 38 \% 11 = 6$  – gives personalized increment.
  - $= 38 \% 13 = 12$
  - $= (12 + 6) \% 13 = 5$
  - $= (5 + 6) \% 13 = 11$
  - $= (11 + 6) \% 13 = 4$
  - $= (4 + 6) \% 13 = 10$
  - $= (10 + 6) \% 13 = 3$
  - $= (3 + 6) \% 13 = 9$
  - $= (9 + 6) \% 13 = 2$
  - $= (2 + 6) \% 13 = 8$
  - $= (8 + 6) \% 13 = 1$
  - $= (1 + 6) \% 13 = 7$
  - $= (7 + 6) \% 13 = 0$
  - $= (0 + 6) \% 13 = 6$
- Notice how hash values jump around over the range of possible values at random.
- Each of the probe sequences visits all the table locations if the size of the table and the size of the increment are relatively prime with respect to each other. This always happens if the table size is prime.

## Buckets

- Another way is to have each entry in the hash table hold more than one (B) item.
- Each entry is called a *bucket*.
- This doesn't really solve the problem. As soon as the bucket fills up, you have to deal with collisions.
- I think it would be better to just have the original table B times as large.

## Separate Chaining

- Each location in the hash table is a pointer to a linked list of things that hashed to that location.
- Advantages
  - Saves space if records are long
  - Collisions are no problem
  - Overflow is solved as space is dynamic
  - Deletion is easy
- Disadvantages

- o Links require space
- For example:  $\text{HASH}(\text{key}) = \text{key} \% 7$  with the data: 5 28 3 108 72 19
- Works well, but we have the overhead of pointers.

## Deletions

- For probing, an empty spot found during a find operation indicates not present.
  - o Solution: use lazy deletion.

## Buckets

- Another way is to have each entry in the hash table hold more than one (B) item.
- Each entry is called a *bucket*.
- This doesn't really solve the problem. As soon as the bucket fills up, you have to deal with collisions.
- I think it would be better to just have the original table B times as large.

## Efficiency of Hashing

- *load factor* = number-of-elements / TABLESIZE
- Should not exceed 2/3.
- With chaining, it is the average size of a chain.

## Summary of Hashing

- Properties of a good hash function
  - o Minimize collisions
  - o Fast to compute
  - o Scatter data (random or non random) evenly through hash table.
  - o Uses all bits of the key – generally helps in scattering non-random data
  - o If mod is used, base should be prime
- Chaining methods are superior to probing, at expense of storage for links.
- Advantages and disadvantages of hashing
  - o Usually lower number of probes than comparison searches (like binary search tree), but could be bad for some sets of data
  - o No order relationship: can't print all items in order
  - o Cannot look for items *close* to given key
- The code below is an example of code for managing a hash table.

```
//Hash.h
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cassert>
using namespace std;
```

```

#ifndef __HashTable
#define __HashTable

/* HashTable expects the type Etype to contain the following functionality:

```

```

    void printIt(ostream &); // Prints itself
    int getHash()           // returns an appropriate hash value
    bool operator!=(Etype) // compares two Etypes for equality

```

Try passing HashTable an Etype which does NOT have these operations to see how the compiler tells you what it requires.

If you wanted to use this hashtable for integers or strings, you would have to replace calls to both printIt and getHash as you cannot add operators to built-in types.

```

*/

```

```

enum KindOfEntry { Active, Empty, Deleted };

```

```

template <class Etype>
class HashEntry
{
public:
    Etype element; // The item
    KindOfEntry kind; // Active, empty, or deleted
    HashEntry(){kind=Empty;}
    HashEntry( Etype & E, KindOfEntry k = Empty ) { element = E; kind = k;}
};

```

```

template <class Etype>
class HashTable
{
public:
    HashTable(ofstream & ,double,int);
    ~HashTable( ) { delete [ ] hashTbl; }
    bool insert( Etype & X ); // Return true if successful
    bool remove( Etype & X ); // Return true if successful
    Etype & find( Etype & X, bool & success); // Return item in table
    void makeEmpty( ); // Clear the hash table
    void Print(char * msg,int size=-1);
        // Print size elements of the hash table preceded by the message
private:
    void reHash(); // Make a new hash table and rehash into it
    int nextPrime( int N ); // find the next prime larger than N
    double fillRatio; // Fraction of arraysize filled before rehashing
    int arraySize; // The maximum size of this array
    int usedSlots; // The number of used items
    int hashFunct(Etype x); // Hash function
    HashEntry<Etype> *hashTbl; // The array of elements
    int findPos( Etype & X , bool any=false );
        // Collision resolution - finds available location
    ofstream & fout;

```

```

};

#endif

// find location where X is located
// any is true if Deleted entries are to be counted as a reasonable location

// This is a REALLY tough method to read - I'm sure you noticed.
// This is why it is so tough. It is trying to be general purpose.
// I can use it to: find where X is located in the array, skipping deleted elements
// or find where X should be located, not skipping deleted elements.
// The "any" flag tells me what my goals are. Any is true means that
// ANY location is good, not just non-deleted ones.

template <class Etype>
int
HashTable<Etype>::
findPos( Etype & X, bool any)
{
    unsigned int probe1 = 0; // The number of failed probes
    unsigned int currentPos = hashFunc( X);

    // This loop handles collision resolution. It says,
    // keep looking until you find an Empty cell.
    // When you find a Deleted cell, stop if "any" is true.
    // Stop when you find the key you were looking for
    // Stop when you get tired (having tried arraySize elements)

    while( (hashTbl[ currentPos ].kind != Empty &&
            !(hashTbl[ currentPos].kind == Deleted && any ))
            && hashTbl[ currentPos ].element != X && probe1 < arraySize)
    {
        //currentPos += 2 * ++probe1 - 1; // Compute ith probe using quadratic probing
        probe1++; // Compute ith probe using linear probing
        currentPos++;
        if( currentPos >= arraySize ) // Implement the mod
            currentPos -= arraySize;
    }

    // If you got out of the loop because you were tired, report that.

    if (probe1 >= arraySize) return -1;
    return currentPos;
}

// returns next prime > N; assume N >= 5

template <class Etype>
int
HashTable<Etype>::
nextPrime( int N )

```

```

{ int i;
  if( N % 2 == 0) N++;
  for( ; ; N += 2 )
  {
    for( i = 3; i * i <= N; i += 2 )
      if( N % i ==0) break;
    if( i * i > N ) return N;
  }
}

// Allocate the hash table array
// size is number of elements in array
// fill is fill-ratio determining when rehash should occur
// f is file to use for output
template <class Etype>
HashTable<Etype>::
HashTable(ofstream & f, double fill = .7,int size=17) : fout(f)
{
  fillRatio = fill;
  arraySize = size;
  hashTbl = new HashEntry<Etype>[ arraySize ];
  usedSlots = 0;
}

// Clear the hash table

template <class Etype>
void
HashTable<Etype>::makeEmpty( )
{
  usedSlots = 0;
  for( int i = 0; i < arraySize; i++ )
    hashTbl[ i ].kind = Empty;
}

// Return item in hash table that matches X
// set success flag appropriately

template <class Etype>
Etype &
HashTable<Etype>::
find( Etype & X, bool &success)
{
  int currentPos = findPos( X);
  success= currentPos >=0 &&hashTbl[ currentPos ].kind == Active;
  return hashTbl[ currentPos ].element;
}

//Use hash function to get location in array

```

```

template <class Etype>
int
HashTable<Etype>::
hashFunct(Etype X)
{
    return X.getHash() %arraySize; // forces Etype to provide the hash function
}

```

```

// Use lazy deletion to delete X.
// Return success flag

```

```

template <class Etype>
bool
HashTable<Etype>::
remove( Etype & X )
{
    int probject;
    int currentPos = findPos( X);
        if( currentPos <0 ||hashTbl[ currentPos ].kind != Active )
            return false;
    hashTbl[ currentPos ].kind = Deleted;
    --usedSlots;
    return true;
}

```

```

// insert X into hash table
// Return false if X is a duplicate; true otherwise
// Rehash automatically as needed

```

```

template <class Etype>
bool
HashTable<Etype>::
insert( Etype & X )
{
    bool success;
    // Don't insert duplicates
    Etype &where = find( X,success );
    if (success)return false;
    int currentPos= findPos( X, true );
    while(currentPos <0)
    {
        reHash();
        currentPos = findPos( X,true );
    }
    assert(hashTbl[ currentPos ].kind != Active );
    // insert X as active
    hashTbl[ currentPos ].element = X;
    hashTbl[currentPos].kind = Active;
    if( ++usedSlots >= arraySize * fillRatio )reHash();
    return true;
}

```

```

// Take everything from the old hash table and put it
// into a hash table which is at least twice as big.

```

```
// Make sure the new hash table size is prime.
```

```
template <class Etype>
void
HashTable<Etype>::reHash()
{
    HashEntry<Etype> *oldArray = hashTbl;
    fout << "rehashing Old size is " << arraySize << " Current size is " << usedSlots << endl;
    //Print("About to Rehash");
    int oldArraySize = arraySize;

    // Create a new empty table
    usedSlots = 0;
    arraySize = nextPrime( oldArraySize *2);
    hashTbl = new HashEntry<Etype>[ arraySize ];
    // Copy table over
    for( int i = 0; i < oldArraySize; i++ )
        if( oldArray[ i ].kind == Active )
            insert( oldArray[ i ].element );
    // Recycle oldArray
    delete [ ] oldArray;
}
```

```
// Print the first num items of the hash table with an informative msg.
```

```
template <class Etype>
void
HashTable<Etype>::Print(char * msg, int num)
{ fout << msg;
  char* Kind[3] = {"Active", "Empty", "Deleted"};
  if (num== -1) num =arraySize;
  fout << " Array size =" << arraySize << "\n";
  fout << " Current Size is " << usedSlots << "\n";
  int activeCt=0;
  for (int i = 0; activeCt < num && i < arraySize;i++)
  {
    if (hashTbl[i].kind != Empty){
      if (activeCt++ <=num)
        { fout << i << setw(6) << " ";
          hashTbl[i].element.printIt(fout);
          fout << Kind[hashTbl[i].kind] << endl;
        }
    }
  }
}
```