Hashing

- Average case for operations on a search tree: O(log N)
- Question: Can we do better at least for some of the operations?
- Goal: insertion, deletion and find in O(1)
- Solution: Hash table
- Hashing does not support operations that require ordering information: findMin, findMax and print in sorted order

Hash Table

- Hash Table: an array of fixed size (TableSize) containing the items

  A[0], A[1], ..., A[TableSize – 1]
- Item = key + data member.
- Search operations are performed on the keys.
- Each key is mapped into an integer between 0 and TableSize-1 and placed in the appropriate cell.
- Mapping is done using a hash function.
- Ideal hash function:
  - Easy to compute
  - Maps two distinct keys into different cells (impossible !).
- Good hash function: distributes the keys evenly among the cells.

Hash Table

- Example:

  h(John) = 3

  h(Phil) = 4

  h(Dave) = 6

  h(Mary) = 7

- Problems:
  - Choosing the hash function.
  - Resolving collisions (when two keys hash to the same value).
  - Deciding the table size.

0	
1	
2	
3	John 25000
4	Phil 31250
5	
6	Dave 27500
7	Mary 28200
8	
9	

Hash Function

- Example:

  h(key) = (key) mod (TableSize)

  - Good if keys are uniformly distributed.
  - If TableSize = 10 and all keys ending in zero => bad choice
- Solution: TableSize = a prime number
- If key is a string => use the sum of ASCII values of the characters as input to h().

Hash Function

Example 1:

```
int hash( const string & key, int tableSize )
{
        int hashVal = 0;

        for( int i = 0; i < key.length(); i++)
            hashVal += key[ i ];

        return hashVal % tableSize;
}
```

- If table size is large => keys are not distributed well.
- Example: TableSize = 10,007 (prime number)
  8 characters keys
  ASCII value of a character < 127

=> key values between 0 and 127*8 = 1,016

=> not an equitable distribution

Hash Function

Example 2:

```
int hash( const string & key, int tableSize )
{
        return ( key[ 0 ] + 27* key[ 1 ] + 729 * key [ 2 ]) %
            tableSize;
}
```

- Assumes key length >= 3
- Example: TableSize = 10,007 (prime number)

  Number of combinations of 3 letters => 26^3 = 17,576

  English is not random => the number of different combination of 3 letters is 2,851

  If no collision => only 28 % of the table is used

  => not appropriate as a hash function if the size of the table is large

Hash Function

Example 3:

```
int hash( const string & key, int tableSize )
{
        int hashVal = 0;

        for( int i = 0; i < key.length(); i++)
          hashVal = 37 * hashVal + key[ i ];

        hashVal %= tableSize;
        if(  hashVal < 0 )      //overflow may introduce a negative number
          hashVal += tableSize;

        return hashVal;
}
```

- Involves all characters in a key
- h(Key) = Key[KeySize – 1] + Key[KeySize – 2]*37 + …+ Key[0]*$37^{\text{KeySize} - 1}$
- It is based on Horner's rule for evaluating polynomials:

  $H = k_0 + 37*k_1 + 37^2 * k_2 = ((k_2)*37 + k_1)*37 + k_0$
- Good distribution, but slow for large keys.

Hash Function

```
hashVal %= tableSize;
if(  hashVal < 0 )      //overflow may introduce a negative number
         hashVal += tableSize;
```

Complement of 2 representation:

4 bit signed integer: 1 sign bit | 3 bits

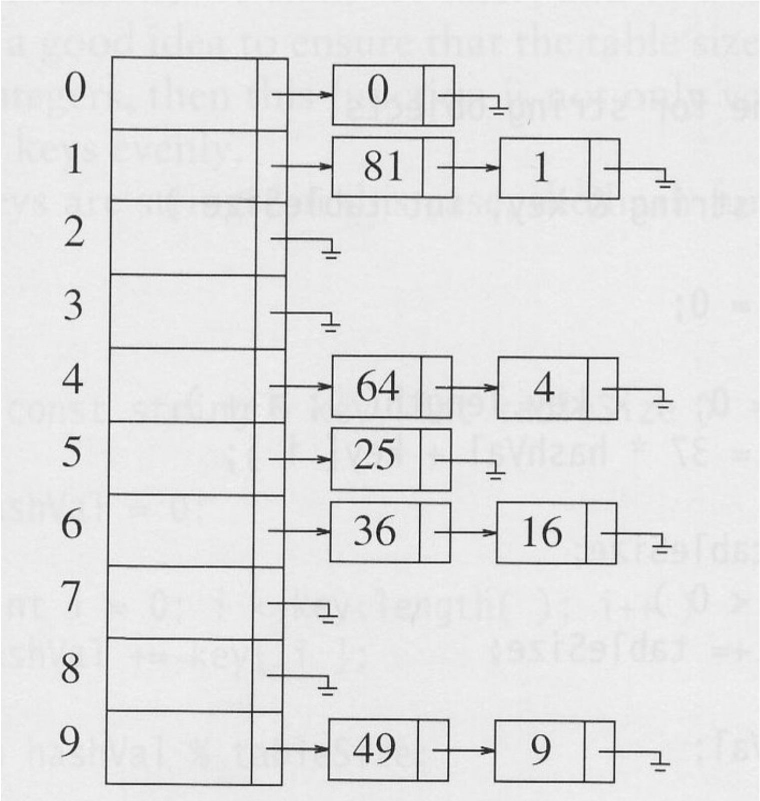
CC1:	CC2:	
+0: 0000	+ 0: 0000	invert the bits then add 1
.	.	
.	.	+7: 0111 → 1000 → 1001 → -7
+7: 0111	+7: 0111	
-7: 1000	-8: 1000	
.	.	
.	.	
-0: 1111	-1: 1111	

Collision Resolution

- Collision: given two keys k_1 and k_2 we have a collision at slot x if $h(k_1) = h(k_2) = x$
- How to deal with collisions?
- Solutions:
  - separate chaining
  - open addressing
- Procedure for finding an item with a key value K:
  - Compute the table location h(K).
  - Starting with slot h(K) locate the item containing key K using (if necessary) a collision resolution policy.

Separate Chaining

- **Idea:** keep a list of all elements that hash to the same value.
- **Assumption:** h(x) = x mod 10 and the keys are the first ten perfect squares.
- **Find(k):**
  - H(k) = x
  - Search list x.
- **Insert(k):**
  - H(k) = x
  - Search list x, if element is not on the list insert the element at the front.



Separate Chaining

```
template <typename HashedObj>
class HashTable
{
        public:
            explicit HashTable( int size = 101 );

            bool contains( const HashedObj & x ) const;

            void makeEmpty( );
            void insert( const HashedObj & x );
            void remove( const HashedObj & x );

        private:
            vector<list<HashedObj> > theLists; // The array of Lists
            int currentSize;

            void rehash( );
            int myhash( const hashedobj & x ) const;
};

int hash( const string & key );
int hash( int key );
```

myhash & makeEmpty

```
int myhash( const HashedObj & x ) const
{
        int hashVal = hash( x );

    hashVal %= theLists.size();
    if( hashVal < 0)
        hashVal += theLists.size ();

    return hashVal;
}


void makeEmpty( )
{
    for( int i = 0; i < theLists.size( ); i++ )
            theLists[ i ].clear( );
}
```

contains & remove

```
bool contains( const HashedObj & x ) const
{
        const list<HashedObj> & whichList = theLists[ myhash( x ) ];
        return find( whichList.begin( ), whichList.end( ), x) != whichList.end( );

}



bool remove( const HashedObj & x )
{
        list<HashedObj> & whichList = theLists[ myhash( x ) ];
        list<HashedObj>::iterator itr = find( whichList.begin( ), whichList.end( ),
x );

        if( itr == whichList.end( ) )
            return false;

        whichList.erase( itr );
        --currentSize;
        return true;
}
```

Insert

```
bool insert( const HashedObj & x )
{
        list<HashedObj> & whichList = theLists[ myhash( x ) ];
        if( find( whichList.begin(), whichList.end(), x ) != whichList.end() )
            return false;

        whichList.push_back( x );

        // Rehash
        if( ++currentSize > theLists.size( ) )
            rehash( );
        return true;
}
```

Running Time Analysis

- What is the running time for find?
- Find:
  - Time to evaluate the hash => O(1)
  - Time to traverse the list => O(?)
- Load factor $\lambda = N/M$

  N = number of items; M = hash table size
- Unsuccessful search: T(N) is given by the number of average elements in the lists => $O(N/M)$ => $O(\lambda)$
- Successful search: $T(N) = 1 + \lambda/2$ => $O(\lambda)$
- Find, insert and remove => $O(\lambda)$
- Size of the table is not important only the load factor is.
- Suggestions:
  - Make the size of the table as large as the number of elements => $\lambda = 1$
  - Table size = prime number