

Hashing - 2

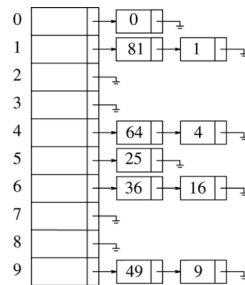
Designing Hash Tables
Sections 5.3, 5.4, 5.4, 5.6

Designing a hash table

- Hash function: establishing a key with an indexed location in a hash table.
 - $\text{Index} = \text{hash}(\text{key}) \% \text{table_size};$
- Resolve conflicts:
 - Need to handle multiple keys that may be mapped to the same index.
 - Two representative solutions
 - Linear probe open addressing (will discuss more later)
 - Chaining with separate lists.

Separate Chaining

- Each table entry stores a list of items
- So we don't need to worry about multiple keys mapped to the same entry.



Separate Chaining (contd.)

```

1  template <typename HashedObj>
2  class HashTable
3  {
4  public:
5      explicit HashTable( int size = 101 );
6
7      bool contains( const HashedObj & x ) const;
8
9      void makeEmpty( );
10     void insert( const HashedObj & x );
11     void remove( const HashedObj & x );
12
13 private:
14     vector<list<HashedObj> > theLists; // The array of Lists
15     int currentSize;
16
17     void rehash( );
18     int myhash( const HashedObj & x ) const;
19 };
20
21 int hash( const string & key );
22 int hash( int key );
    
```

Type Declaration for
Separate Chaining
Hash Table

Separate Chaining (contd.)

```

1  int myhash( const HashedObj & x ) const
2  {
3      int hashVal = hash( x );
4
5      hashVal %= theLists.size( );
6      if( hashVal < 0 )
7          hashVal += theLists.size( );
8
9      return hashVal;
10 }
    
```

Separate Chaining (contd.)

```

1  // Example of an Employee class
2  class Employee
3  {
4  public:
5      const string & getName( ) const
6      { return name; }
7
8      bool operator==( const Employee & rhs ) const
9      { return getName( ) == rhs.getName( ); }
10     bool operator!=( const Employee & rhs ) const
11     { return !(*this == rhs); }
12
13     // Additional public members not shown
14
15 private:
16     string name;
17     double salary;
18     int seniority;
19
20     // Additional private members not shown
21 };
22
23 int hash( const Employee & item )
24 {
25     return hash( item.getName( ) );
26 }
    
```

Separate Chaining (contd.)

```

1 void makeEmpty( )
2 {
3     for( int i = 0; i < theLists.size( ); i++ )
4         theLists[ i ].clear( );
5 }
6
7 bool contains( const HashedObj & x ) const
8 {
9     const list<HashedObj> & whichList = theLists[ myhash( x ) ];
10    return find( whichList.begin( ), whichList.end( ), x ) != whichList.end( );
11 }
12
13 bool remove( const HashedObj & x )
14 {
15     list<HashedObj> & whichList = theLists[ myhash( x ) ];
16     list<HashedObj>::iterator itr = find( whichList.begin( ), whichList.end( ), x );
17     if( itr == whichList.end( ) )
18         return false;
19     whichList.erase( itr );
20     --currentSize;
21     return true;
22 }
23
24

```

Separate Chaining (contd.)

```

1 bool insert( const HashedObj & x )
2 {
3     list<HashedObj> & whichList = theLists[ myhash( x ) ];
4     if( find( whichList.begin( ), whichList.end( ), x ) != whichList.end( ) )
5         return false;
6     whichList.push_back( x );
7
8     // Rehash; see Section 5.5
9     if( ++currentSize > theLists.size( ) )
10        rehash( );
11
12    return true;
13 }

```

Hash Tables Without Chaining

- Try to avoid buckets with separate lists
- How → use **Probing Hash Tables**
 - If collision occurs, try another cell in the hash table.
 - More formally, try cells $h_0(x)$, $h_1(x)$, $h_2(x)$, $h_3(x)$... in succession until a free cell is found.
 - $h_i(x) = (\text{hash}(x) + f(i))$
 - And $f(0) = 0$

Linear Probing: $f(i) = i$

```

Insert(k,x) // assume unique keys
1. index = hash(key) % table_size;
2. if (table[index] == NULL)
    table[index] = new key_value_pair(key, x);
3. Else {
    • index++;
    • index = index % table_size;
    • goto 2;
}

```

Linear Probing: Search

- Search (key)**
 - Index = hash(key) % table_size;
 - If (table[index] == NULL)
 - return -1; // Item not found
 - Else if (table[index].key == key)
 - return index;
 - Else {
 - Index ++;
 - index = index % table_size;
 - goto 2;

Linear Probing Example

Insert 89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Linear Probing: Delete

- Can be tricky ...
- How to maintain the consistency of the hash table
- What is the simplest deletion strategy you can think of?

Quadratic Probing

$$f(i) = i^2$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing

- $f(i) = i * hash_2(x)$
- E.g.: $hash_2(x) = 7 - (x \% 7)$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8		18	18	18	18	18
9		89	89	89	89	89

What if $hash_2(x) == 0$ for some x ?

Rehashing

- Hash Table may get full
 - No more insertions possible
- Hash table may get *too* full
 - Insertions, deletions, search take longer time
- Solution: Rehash
 - Build another table that is twice as big and has a new hash function
 - Move all elements from smaller table to bigger table
- Cost of Rehashing = $O(N)$
 - But happens only when table is close to full
 - Close to full = table is X percent full, where X is a tunable parameter

Rehashing Example

Original Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

0	6
1	15
2	23
3	24
4	
5	
6	13

After Rehashing

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing Implementation

```

1  /**
2  * Rehashing for quadratic probing hash table.
3  */
4  void rehash()
5  {
6      vector<HashEntry> oldArray = array;
7
8      // Create new double-sized, empty table
9      array.resize( nextPrime( 2 * oldArray.size() ) );
10     for( int j = 0; j < array.size(); j++)
11         array[ j ].info = EMPTY;
12
13     // Copy table over
14     currentSize = 0;
15     for( int i = 0; i < oldArray.size(); i++)
16         if( oldArray[ i ].info == ACTIVE )
17             insert( oldArray[ i ].element );
18 }
19
20 /**
21 * Rehashing for separate chaining hash table.
22 */
23 void rehash()
24 {
25     vector<list<HashedObj>> oldLists = theLists;
26
27     // Create new double-sized, empty table
28     theLists.resize( nextPrime( 2 * theLists.size() ) );
29     for( int j = 0; j < theLists.size(); j++)
30         theLists[ j ].clear();
31
32     // Copy table over
33     currentSize = 0;
34     for( int i = 0; i < oldLists.size(); i++)
35     {
36         list<HashedObj>::iterator itr = oldLists[ i ].begin();
37         while( itr != oldLists[ i ].end() )
38             insert( *itr++ );
39     }
40 }

```