Hashing

CS 202 – Fundamental Structures of Computer Science II

Bilkent University

Computer Engineering Department

Hashing

- We will now see a data structure that will allow the following operations to be done in O(1) time:
  - Insertion
  - Deletion
  - Find
- This data structure,however, is not efficient in operation that require any ordering information among the elements
  - Sort
  - Find minimum
  - Find maximum

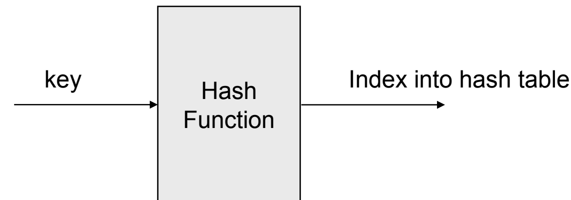
Hashing

- Hashing requires a *Hash Table* to be used.
- There may various methods in *implementing a hash table*, but the most simplest one is using an *array of some fixed size*.
- The maximum size of the array is *TableSize*.
- The items that are stored in the array (hash table) are indexed by values from *0* to *TableSize – 1*.
- A stored item need to have a data member, called *key*, that will be used in computing the index value for the item.
  - For student items, key could be the student ID or student Name
  - For account balance records in a bank, the key could be the account number of the customer, ….

Hashing- General Idea

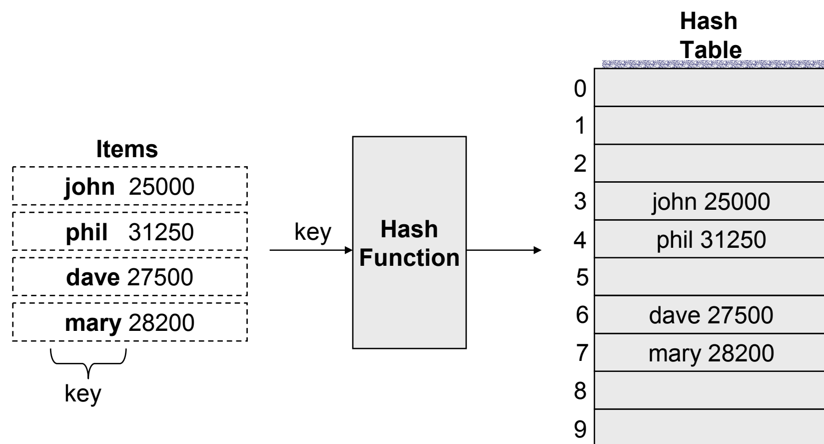
- When an item needs to be inserted into the Hash Table:
  - The *key* of the item is *mapped* into an *index value* (between *0* and *TableSize -1*) using a special function, called Hash Table.
  - The item is then stored in the array at that index value.
  - If more than one key (items) are mapped into the same index value, then we say that we have a collision.
  - Collisions must be resolved. We need to handle collisions.
- When an item needs to be found (searched)
  - The *key* of the item is *mapped* into an *index value* again and the item is tried to be retrieved from that index of the array. If the item could not be stored at the location (at that index), then appropriate error message could be returned.

Hash Function



- Hash function objectives:
  - Should be simple to compute
  - *Theoretically*: Should ensure (or try) that any two distinct keys are mapped into different hash table entries (cells).
  - *Practically*: Should distribute the keys evenly among the cells.

Example



Key could be an *integer*, a *string*, etc. Here it is a string.

Hash Function

- If the keys are integers:
  - *Hash(Key)* = *Key* mod *TableSize*
  - Table size of *prime*.
- If the keys are strings:
  - It is more difficult.
  - The keys need to be well-distributed.
  - Table should be well utilized.
  - We will give 3 functions

Hash Function for Strings

Function 1

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

        for (int i = 0; i < key.length(); i++)
                hashVal += key[i];
        return hashVal % tableSize;
}
```

- Method: Sum up the ascii values of the characters of the key!
- This method causes some part of the hash table to be unused.

- A case for which the functions do not work well:
  - Let say TableSize is 10,007 (a prime number).
  - Assume all keys are 8 characters long.
  - 8 x 127 = 1016 different 8 characters combinations possible.
  - Only the hash values in the range 0..1016 will be generated.
  - The rest of the table will be unused.

Hash function for strings

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

Function 2

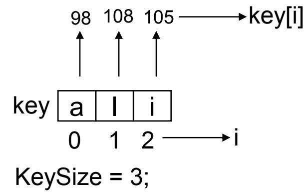
- The functions uses the first 3 characters of the key string.
- Computes the number of English words that can be generated by the first 3 characters.
  - **26** English alphabet character + **1** blank symbol = **27** symbols are represented by a character.
  - Therefore there are **26** * **26** * **26** = **17576** different English words that can be theoretically generated.
  - However, English dictionary contains **2851** three-character valid English words.
- Therefore, this function also does not utilize the entire hash table and does not distribute the keys randomly to cells. Some cells are always empty.

Hash function for strings:

```
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)   /* in case overflows occur in computation
                                 */
                hashVal += tableSize;

        return hashVal;
};
```

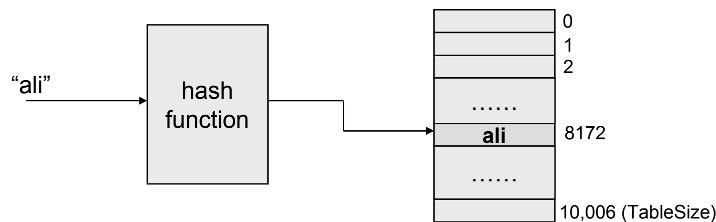
$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^{i}$$

Hash function for strings:



KeySize = 3;

hash("ali") = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172



Resolving Collisions

- When two or more different keys are mapped to the same hash value (index), this is called collision.
  - We have to find some way of storing the items mapped to this hash value.
    - We have a single array cell at the hash value index that can store only a single item
- There are various methods to resolve collisions.

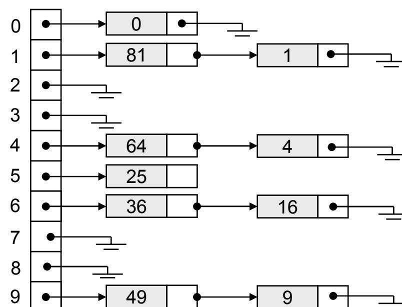
Separate Chaining

- Method is
  - Keep a list of items (elements) that hash to the same value.
  - The array *cell* which has the *hash-value* as the index, will have a pointer to the first element of the list.
  - When a new item is to be inserted into the list, it can be inserted to the front of the list
    - We don't need to traverse to the end of the list.
  - If duplicate items are to be inserted:
    - A *reference count* can be kept at each item node, that points how many items are present at that node.

Separate Chaining – an Example

We have items (keys): 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hask(key) = key % 10.



Hash Table Class for Separate Chaining

```
#include "vector.h"
#include "mystring.h"
#include "LinkedList.h“

template <class HashedObj>
class HashTable
{
  public:
    explicit HashTable( const HashedObj & notFound, int size = 101 );
    HashTable( const HashTable & rhs )
      : ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND ), theLists( rhs.theLists ) { }

    const HashedObj & find( const HashedObj & x ) const;

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

    const HashTable & operator=( const HashTable & rhs );
  private:
    vector<List<HashedObj> > theLists;   // The array of Lists
    const HashedObj ITEM_NOT_FOUND;
};

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


Hash Table Hash() Member function

```
/* A hash routine for string objects. */
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 )
            hashVal += tableSize;

         return hashVal;
      }

      /**  A hash routine for ints */
 int hash( int key, int tableSize )
      {
         if( key < 0 ) key = -key;
         return key % tableSize;
      }
```

Hash Table Constructor And makeEmpty operations

```
/**
 * Construct the hash table.
 */
template <class HashedObj>
HashTable<HashedObj>::HashTable( const HashedObj &
notFound, int size )
  : ITEM_NOT_FOUND( notFound ), theLists( nextPrime( size ) )
{   }

/**
 * Make the hash table logically empty.
 */
template <class HashedObj>
void HashTable<HashedObj>::makeEmpty( )
{
    for( int i = 0; i < theLists.size( ); i++ )
        theLists[ i ].makeEmpty( );
}

}
```


Hash Table Constructor And makeEmpty operations

```
/**
 * Construct the hash table.
 */
template <class HashedObj>
HashTable<HashedObj>::HashTable( const HashedObj &
notFound, int size )
  : ITEM_NOT_FOUND( notFound ), theLists( nextPrime( size ) )
{   }

/**
 * Make the hash table logically empty.
 */
template <class HashedObj>
void HashTable<HashedObj>::makeEmpty( )
{
    for( int i = 0; i < theLists.size( ); i++ )
        theLists[ i ].makeEmpty( );
}

}
```

Hash Table Insert operation

```
/**
 * Insert item x into the hash table. If the item is
 * already present, then do nothing.
 */
template <class HashedObj>
void HashTable<HashedObj>::insert( const HashedObj & x )
{
    List<HashedObj> & whichList = theLists[ hash( x, theLists.size( ) ) ];
    ListItr<HashedObj> itr = whichList.find( x );

    if( itr.isPastEnd( ) )
        whichList.insert( x, whichList.zeroth( ) );
}
```


Hash Table remove operation

```
/**
 * Remove item x from the hash table.
 */
template <class HashedObj>
void HashTable<HashedObj>::remove( const HashedObj & x )
{
   theLists[ hash( x, theLists.size( ) ) ].remove( x );
}
```

Hash Table find operation

```
/**
 * Find item x in the hash table.
 * Return the matching item or ITEM_NOT_FOUND if not found
 */
template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const HashedObj & x ) const
{
    ListItr<HashedObj> itr;
    itr = theLists[ hash( x, theLists.size( ) ) ].find( x );
    if( itr.isPastEnd( ) )
        return ITEM_NOT_FOUND;
    else
        return itr.retrieve( );
}
```


- Linked list solution for collision resolution can be replaces by:
  - Binary search tree
  - An other hash table.
  - ….
- Load factor λ definition:
  - Ratio of number of elements (*N*) in a hash table to the hash *TableSize*.
    - $\lambda = N/TableSize$

- The average length of a list is λ
- Effort required for search
  - Unsuccessful:
    - We have to traverse the list, so we need to examine λ nodes on the average.
  - Successful search:
    - Hash function computation (O(1)) – constant, 1 node processing for the item found, x nodes processing for other items in the list.
    - $x = (N-1) / M = \lambda - 1 / M$
      (N: number of nodes, M: number of lists). M is large.
    - So, $x \sim= \lambda$
    - On the average, we need to check *half* of the *other nodes* while searching for a certain element.
    - So the total cost is: $1 + \lambda/2$

General Rules for separate chaining

- Not the TableSize,but the load factor (λ) is important in defining the *cost* of operations.
- TableSize should be as *large* the number of expected elements in the hash table.
  - To keep load factor around *1*.
- TableSize should be *prime* for even distribution of keys to hash table cells.