Hash Tables

Application

Speed up searching:

Consider the problem of searching an array for a given value.

If the array is not sorted, the search might require examining each and all elements of the array.

If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n).

We could search even faster if we know in advance the index at which that value is located in the array.

With this magic function our search is reduced to just one probe, giving us a constant runtime O(1).

What are hash tables?

- A Hash Table is used to implement a **set**, providing basic operations in constant time:
  - insert
  - remove (optional)
  - find
  - makeEmpty (need not be constant time)

- The table uses a function that maps an object in the set to its location in the table.

- The function is called a **hash function.**

Using a hash function

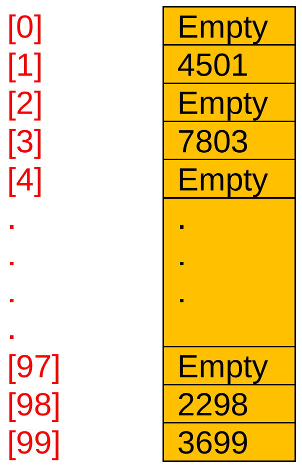
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
.	.
.	.
.	.
.	
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(partNum) = partNum % 100

Placing elements in the array



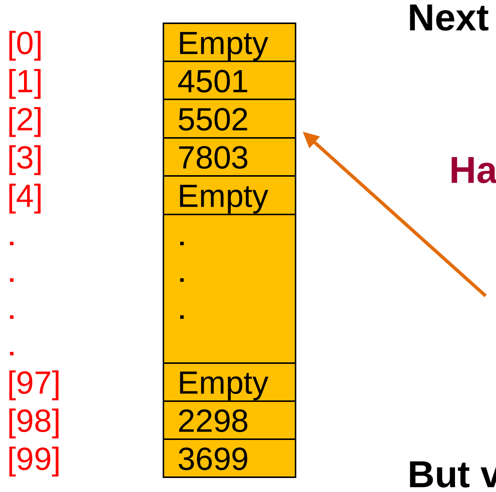
Use the hash function

Hash(partNum) = partNum % 100

to place the element with part number 5502 in the array.

Placing elements in the array

Next , place part number 6702 in the array.



Hash(partNum) = partNum % 100

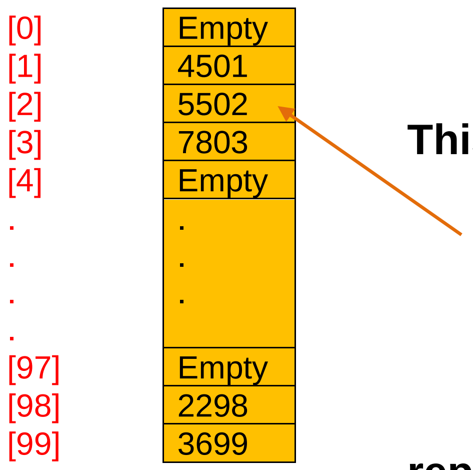
6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to resolve the collision?

One way is by linear probing.

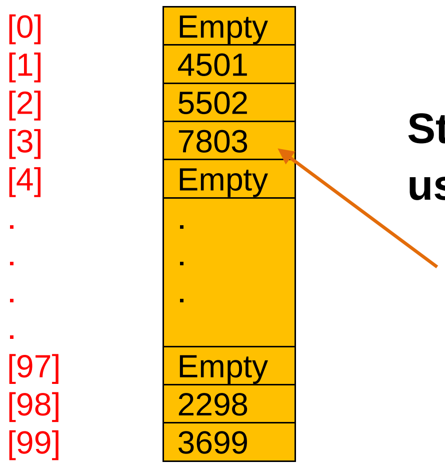


This uses the following function

(HashValue + 1) % 100

repeatedly until an empty location is found for part number 6702.

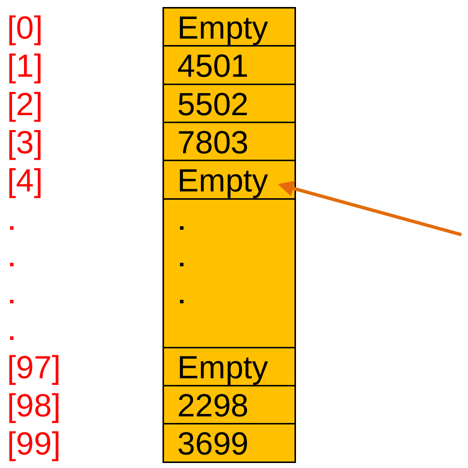
Resolving the collision



Still looking for a place for 6702 using the function

(HashValue + 1) % 100

Collision resolved



Part 6702 can be placed at the

location with index 4.

Collision resolved

[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
.	
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Hashing concepts

- **Hash Table**: where objects are stored by according to their key (**usually an array**)
- **key**: attribute of an object used for **searching / sorting**
  - number of valid keys usually greater than number of slots in the table
  - number of keys in use usually much smaller than table size.
- **Hash function**: maps keys to a Table index
- **Collision**: when two separate keys hash to the same location
- **Collision resolution**: method for finding an open spot in the table for a key that has collided with another key already in the table.
- **Load Factor**: the fraction of the hash table that is full
  - may be given as a percentage: 50%
  - may be given as a fraction in the range from 0 to 1, as in : .5

Hash Function

- **Goals**:
  - computation should be fast
  - should minimize collisions (good distribution)

- Some issues:

  - should depend on ALL of the key (not just the last 2 digits or first 3 characters, which may not themselves be well distributed)

- Final step of hash function is usually : temp % size

  - **temp** is some intermediate result
  - **size** is the hash table size - ensures the value is a valid location in the table

- Picking a value for size:

  - Bad choices:

    - **a power of 2: then the result is only the lowest order bits of temp (not based on whole key)**
    - **a power of 10: result is only lowest order digits of decimal number**

- Good choices: prime numbers

Hash Function: string keys

- If the key is **not** a number, hash function must transform it to a number, to mod by the size
- **Method 1: Add up ascii values**

```
int hash (string key, int tableSize) {
     int hashVal = 0;
     for (int i=0; i  < key.length(); i++)
          hashVal = hashVal + key[i];

     return hashVal % tableSize;
}
```

- Different permutations of same chars have same hash value
- If tableSize is large, **and** key length is not long, may not distribute well:

```
if tableSize is 10007 and keys are 8
characters long:
since ascii values are <= 127,
hash produces values between 0 and 127*8 =
1016

all falling in first 1/10th of the table
```

Hash Function: string keys

- **Method 2: Multiply each char by a power of 128**

```
hash = k[0]*128^3 + k[1]*128^2 + k[2]*128^1 + k[3]*128^0

This equivalent to (which avoids large intermediate
results):

hash = (((k[0]*128 + k[1])*128 + k[2])*128 +
k[3])*128
```

```
int hash (string key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i<key.length(); i++)
        hashVal = (hashVal * 128 + key[i]) % tableSize;

    return hashVal;
}
```

- Now we get really big numbers (overflow)
- Taking mod of intermediate results to reduce overflow
- But mod is expensive
  - could just allow overflow, take mod at the end
  - but repeated multiplying by 128 tends to shift early chars out to the left

Hash Function: string keys

- **Method 3:** **Multiply each char by a power of 37**

```
hash = (((k[0]*37 + k[1])*37 + k[2])*37 + k[3])*37
```

```
int hash (string key, int tableSize) {
     int hashVal = 0;
     for (int i=0; i<key.length(); i++)
          hashVal = hashVal * 37 + key[i];

     return hashVal % tableSize;
}
```

Collision Resolution:

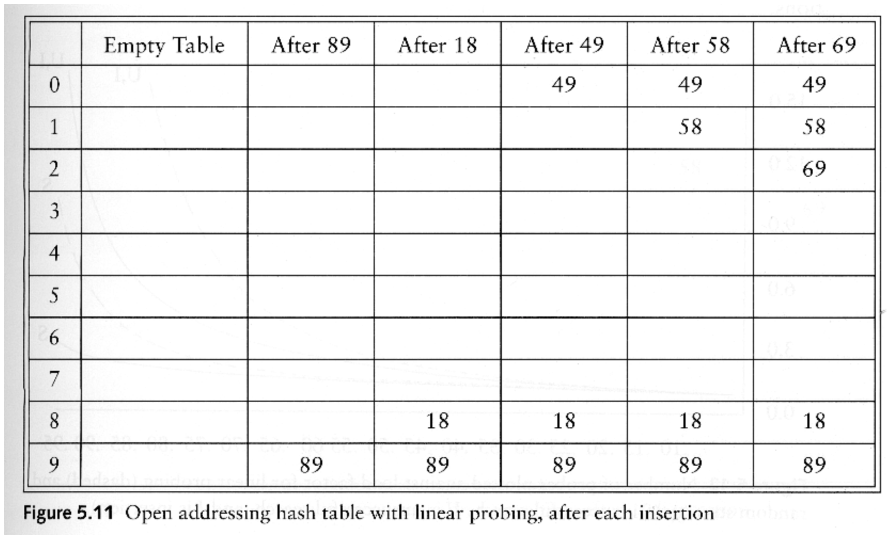
1. Linear Probing

- **Insert**: When there is a collision on, search sequentially for the next available slot
- **Find**: if the key is not at the hashed location, keep searching sequentially for it.
  - If it reaches an empty slot, the key is not found
- Problem: if the table is somewhat full, it may take a long time to find the open slot.
  - May not be O(1) any more
- Problem: Removing an element in the middle of a chain

Linear Probing Example :

Insert: 89, 18, 49, 58, 69, hash(k) = k mod 10

Probing function (attempt i): hi(K) = (hash(K) + i) % tablesize



	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

Figure 5.11 Open addressing hash table with linear probing, after each insertion

49 is in 0 because 9 was full

58 is in 1 because 8, 9, 0 were full

69 is in 2 because 9, 0 were full

Linear Probing: delete problem

[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
.	
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4, after colliding with 5502

Now remove 7803.

Now look for 6702 ===>

(hash(6702)=2):

not at values[2] because 5502 is at values [2] and

values[3] is empty, so not found

Linear Probing: Lazy deletion

- Don't remove the deleted object, just mark as deleted
- During find, marked deletions don't stop the searching
- During insert, the spot may be reused
- If there are a lot of deletions, searching may still take a long time in a "sparse" table

Collision Resolution:

2. Quadratic Probing

- If the hash function returns H, and H is occupied, try H+1, then H+4, then H+9, ...
  - for each attempt i, try $H+i^2$ next.

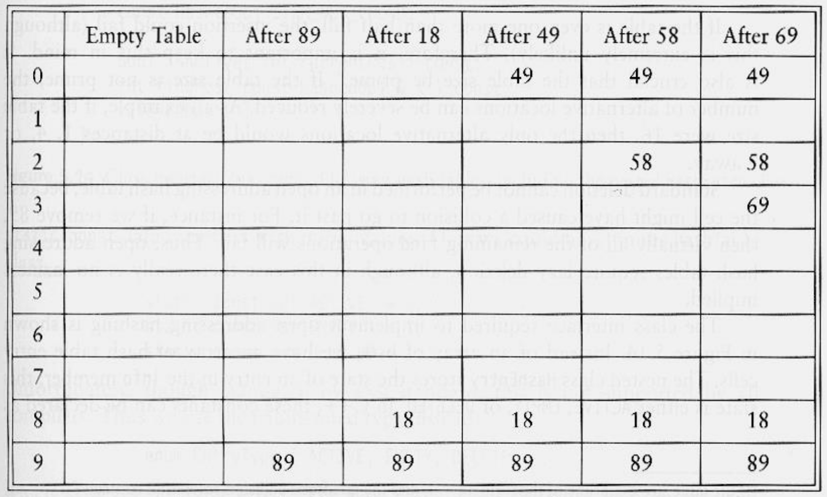
Probing function (attempt i): hi(K) = (hash(K) + i^2) % tablesize

- Is it guaranteed to find an empty slot if there is one (like linear probing)?
  - Yes IF: the table size is prime and the load is <= 50%

Quadratic Probing: Example

- Insert: 89, 18, 49, 58, 69, hash(k) = k mod 10

Probing function (attempt i): hi(K) = (hash(K) + i^2) % tablesize



	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

Figure 5.13 Open addressing hash table with quadratic probing, after each insertion

49 is in 0 because 9 was full

58 is in 2 because 8, 8+1=9 were full, (8+4)%10=2 wasn't

69 is in 3 because 9, (9+1)%10=0 were full, (9+4)%10=3 wasn't

Note: smaller clusters

Quadratic probing: Expansion of Table

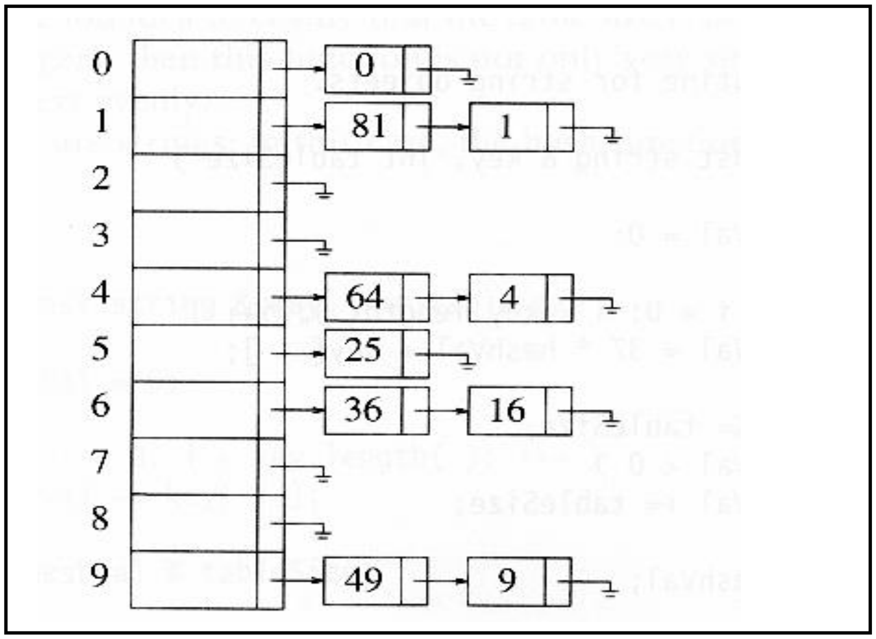
- Since the table should be less than 50% full:
- Can the table be expanded if the load factor gets more than 50%?
- Yes.
  - Find the next prime number greater than 2*tableSize, resize to that.
  - Don't just copy all the elements (new tablesize => new hash function)
  - Scan old table for non-empties, and use insert function to add them to new table.
- This is called **rehashing**.

Collision Resolution:

3. Separate chaining

- Use an **array of linked lists** for the hash table
- Each linked list contains all objects that hashed to that location
    - no collisions

Hash function is still :
h(K) = k % 10



Separate Chaining

- To insert a an object:
  - compute hash(k)
  - insert at front of list at that location (if empty, make first node)

- To find an object:
  - compute hash(k)
  - search the linked list there for the key of the object

- To delete an object:
  - search the linked list there compute hash(k)
  - for the key of the object
  - if found, remove it

- The load can be 1 or more
  - more than 1 node at each location, still O(1) inserts and finds
  - smaller loads do not improve performance
  - moderately larger loads do not hurt performance

- Disadvantages
  - Memory allocation could be expensive
  - too many nodes at one position can slow operations (equivalent to secondary clusters)

- Advantages:
  - deletion is easy
  - don't have to resize/rehash

Example

HashClass.h

```
/*
 *    HashClass.h
 *
 *    Author: Husain Gholoom
 */

#ifndef HASHCLASS_H_
#define HASHCLASS_H_


class hash{
    private:
        int hash_pos;
        int array[40];
    public:
        hash();
        void insert();
        void search();
        int  Hash(int );
        int  reHash(int );
        void Delete();
        void Display();
};

#endif /* HASHCLASS_H_ */
```

HashClassImp.cpp

```
/*
 *    HashClassImp.cpp
 *
 *    Author: Husain Gholoom
 */

#include<iostream>
using namespace std;

#include "HashClass.h"


hash::hash()
{
    for(int i = 0; i < 40; i++){
       array[i] = '*';
    }
}
void hash::insert(){
    int data;
    int count = 0;
    cout<<"Enter the data to insert: ";
    cin>>data;
    hash_pos = Hash(data);
    if(hash_pos >= 40){
        hash_pos = 0;
    }
    while(array[hash_pos] != '*'){
        hash_pos = reHash(hash_pos);
        count++;
        if(count>=40){
            cout<<"Memory Full!!No space is available for storage";
            break;
        }
    }
    if(array[hash_pos] == '*'){
        array[hash_pos] = data;
    }
    cout<<"Data is stored at index "<<hash_pos<<endl;
}
int hash::Hash(int key){
    return key%100;
}
int hash::reHash(int key){
    return (key+1)%100;
}
```

HashClassImp.cpp - (continued)

```
void hash::search(){
    int key,i;
    bool isFound = false;
    cout<<"Enter the key to search: ";
    cin>>key;
    for(i = 0; i < 40; i++){
        if(array[i] == key){
            isFound = true;
            break;
        }
    }
    if(isFound){
        cout<<"The key is found at index "<< i <<endl;
    }else{
        cout<<"No record found!!"<<endl;
    }
}


void hash::Delete(){
    int key,i;
    bool isFound = false;
    cout<<"Enter the key to delete: ";
    cin>>key;
    for(i = 0; i < 40; i++){
        if(array[i] == key){
            isFound = true;
            break;
        }
    }
    if(isFound){
        array[i] = '*';
        cout<<"The key is deleted"<<endl;
    }else{
        cout<<"No key is Found!!";
    }
}

void hash::Display(){

    for(int i = 0; i < 40; i++)
      cout<<array[i]<<"  ";

}
```

HashClassDriver.cpp

```
#include<iostream>
using namespace std;

#include "HashClass.h"

int main(){
    hash h;
    int choice;
    while(1){

        cout<<"\n1. Insert\n2. Search\n3. Delete\n4. Display\n5. Exit\n";
        cout<<"Enter your choice: ";
        cin>>choice;
        switch(choice){
            case 1:
                h.insert();
                break;

            case 2:
                h.search();
                break;

            case 3:
                h.Delete();
                break;

            case 4:
                h.Display();
                break;

            case 5:
                exit(0);

            default:

                cout<<"\nEnter correct option\n";
                break;
        }
    }
    return 0;
}
```

Sample Run

```
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 1
Enter the data to insert: 123
Data is stored at index 23

1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 1
Enter the data to insert: 789
Data is stored at index 0

1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 4
789  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42
42  42  42  42  123  42  42  42  42  42  42  42  42  42  42  42  42  42  42
42  42
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 3
Enter the key to delete: 123
The key is deleted

1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 4
789  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42
42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42  42
42  42
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 5
```