Sets & Hash Tables

Week 13

Weiss: 20
Main & Savitch: 3, 12.2-3

CS 5301
Spring 2014

Jill Seaman

What are sets?

- A set is a collection of objects of the same type that has the following two properties:
  - there are no duplicates in the collection
  - the order of the objects in the collection is irrelevant.
- {6,9,11,-5} and {11,9,6,-5} are equivalent.
- There is no first element, and no successor of 9.

Set Operations

- Set construction
  - the empty set (0 elements in the set)
- isEmpty()
  - True, if the set is empty; false, otherwise.
- Insert(element)
  - If element is already in the set, do nothing; otherwise add it to the set
- Delete(element)
  - If element is not a member of the set, do nothing; otherwise remove it from the set.

Set Operations

- Member(element): boolean
  - True, if element is a member of the set; false, otherwise
- Union(Set1,Set2): Set
  - returns all elements of two Sets, no duplications.
- Intersection(Set1,Set2): Set
  - returns all elements common to both sets.
- Difference(Set1,Set2): Set
  - returns all elements of the first set except for the elements that are in common with the second set.

Set Operations

- Subset(Set1,Set2): boolean
  - True, if Set2 is a subset of Set1. All elements of the Set2 are also elements of Set1.

Implementation

- Array of elements implementation
  - each element of the set will occupy an element of the array.
  - the member (find) operation will be inefficient, must use linear search.
- see Lab 6, exercise 2
  - represented a set of integers
  - class contained a pointer to a dynamically allocated array of ints
- Exercise: implement all of the set operations for this set

Implementation

- Boolean array implementation
  - size is equal to number of all possible elements (the universe).
  - need a mapping function to convert an element of the universe to a position in the array

```
bool array[7] = {false};  //sets all elements to false
int map(string x) {
    if (x=="Sunday") return 0;
    if (x=="Monday") return 1;
    if (x=="Tuesday") return 2;
    if (x=="Wednesday") return 3;
    if (x=="Thursday") return 4;
    if (x=="Friday") return 5;
    if (x=="Saturday") return 6;
}
```

  - if `array[map(“Monday”)]` is true, then Monday is in the Set.

Implementation

- Boolean array implementation: member

```
bool member(string x) {
    int pos = map(x);
    if (0<=pos && pos<7 && array[pos])
       return true;
    return false;
}
```

  - Exercise: implement all of the set operations for the set implemented as a boolean array

What are hash tables?

- A Hash Table is used to implement a **set** (or a **search table**), providing basic operations in constant time:
  - insert
  - delete (optional)
  - find (also called “member”)
  - makeEmpty (need not be constant time)
- It uses a function that maps an object in the set (a key) to its location in the table.
- The function is called a **hash function**.

Using a hash function

	values
[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

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

Use the hash function

Hash(partNum) = partNum % 100

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

Placing elements in the array

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

Next place part number 6702 in the array.

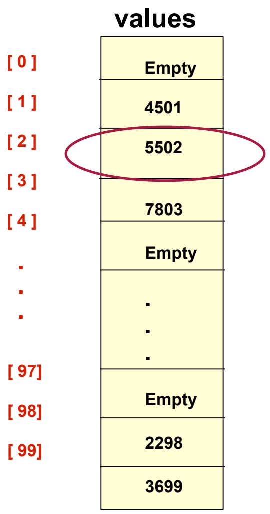
Hash(partNum) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to resolve the collision?

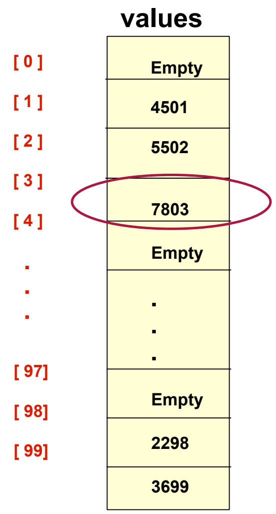


One way is by linear probing.
This uses the following function

(HashValue + 1) % 100

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

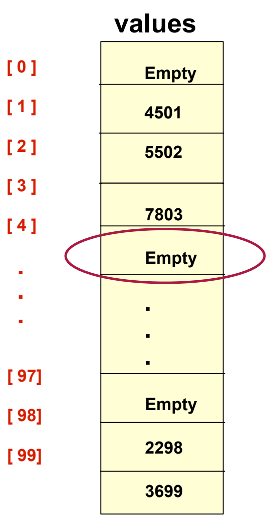
Resolving the collision



Still looking for a place for 6702 using the function

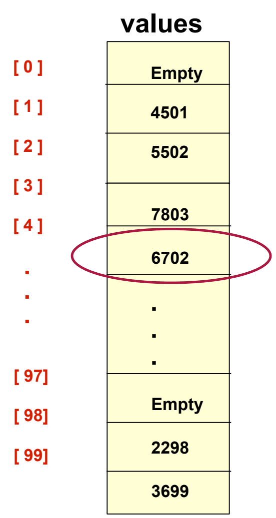
(HashValue + 1) % 100

Collision resolved



Part 6702 can be placed at the location with index 4.

Collision resolved



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

Hashing concepts

- **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)

Hash Function

- 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

Collision Resolution:
Linear Probing

- Insert: When there is a collision, 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 the table is somewhat full, it may take a long time to find the open slot.
- 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): $h_i(K)$ = (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

49 is in 0 because 9 was full

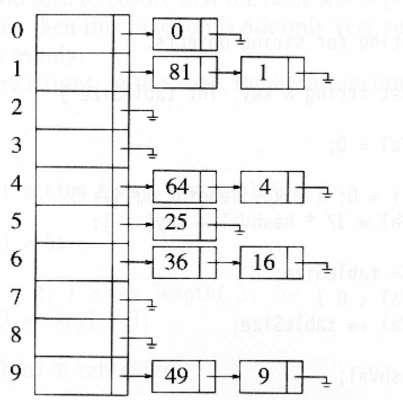
58 is in 1 because 8, 9, 0 were full

69 is in 1 because 9, 0 were full

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