Sets & Hash Tables

Week 13

Weiss: chapter 20

CS 5301
Spring 2015

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 a Set containing all elements of the two Sets, no duplications.
- Intersection(Set1,Set2): Set
  - returns a Set containing all elements common to both sets.

Set Operations

- Difference(Set1,Set2): Set
  - returns a Set containing all elements of the first set except for the elements that are in common with the second set.
- Subset(Set1,Set2): boolean
  - True, if Set2 is a subset of Set1 (if 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 1
  - Hand class represented a set of Card objects
  - addCard (insert) does not add duplicates
  - operator+ implemented the union operation
- Exercise: implement all of the set operations for this set

Implementation

- Boolean array implementation
  - size of the array must be equal to number of all possible elements (the universe).

```
//This array will represent a set of days of the week
//  (Sunday, Monday, Tuesday, . . .)

bool daysOfWeek[7] = {false};  //sets all elements to false
```

  - Here is the set {Monday, Wednesday, Friday}:

FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
0	1	2	3	4	5	6

  - if `daysOfWeek[1]` is true, then Monday is in the Set.

Implementation

- Boolean array implementation
  - need a mapping function to convert an element of the universe to a position in the array

```
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 `daysOfWeek[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)
       return daysOfWeek[pos];
    return false;
}
```

- Boolean array implementation: union

```
// c will be the union of a and b:
void union(bool a[], bool b[], bool c[]) {
    for (int pos=0; pos<7; pos++)
       // if either a or b is true for pos, make c true for pos
       c[pos] = (a[pos] || b[pos]);
}
```

  - 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 (no loops/recursion):
  - 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

Use the hash function 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?

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

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.

Collision resolved

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

6702 + 1 % 100 = 3

But values[3] is already occupied.

6702 + 2 % 100 = 4

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

Collision resolved

	values
[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**: (usually an array) where objects are stored according to their key
  - **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 a key to a Table index
- **Collision**: when two separate keys hash to the same location

Implementation

- Standard array implementation
  - keys are ints:

```
class HashTable {
private:
    enum EntryType { OPEN, OCCUPIED, DELETED };
    struct Entry {
        int value;
        EntryType status;  // OPEN, OCCUPIED, or DELETED
    };
    Entry *array;          // array of elements
    int size;              // size of array
    int hash (int  key) ;  // maps key to position in array
public:
    HashTable (int);  //initialize elements status to OPEN
    ~HashTable();

    bool find(int);       //return true if int in table
    void insert (int);    //add int to table
    void display();       //show elements in table
    void remove(int)      //remove int from table
};
```

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

Collision Resolution: Linear Probing

- Insert: When there is a collision, search sequentially for the next OPEN slot
  - Put the value in the table, set status to OCCUPIED
- Find: if the key is not at the hashed location, keep searching sequentially for it.
  - if it reaches an OPEN slot, the key is not found
- Remove: if the key is not at the hashed location, keep searching sequentially for it.
  - if the key is found, set the status to OPEN
- Problem: Removing an element in the middle of a chain. The Find method needs to know to keep searching to the end of the chain.

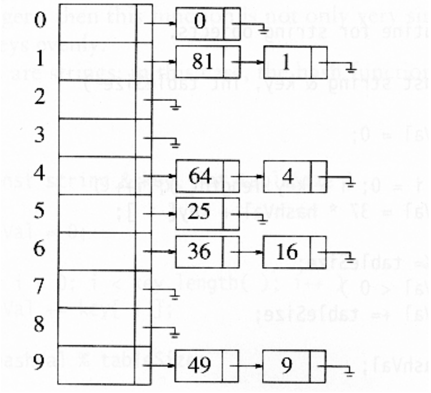
Remove Solution

- Remove: if the key is not at the hashed location, keep searching sequentially for it.
  - skip DELETED and OCCUPIED items not matching the key
  - if the key is found, mark it as DELETED.
- Find: if the key is not at the hashed location, keep searching sequentially for it.
  - skip DELETED and OCCUPIED items not matching the key
- Insert: Use Find, if the key is NOT in the list: Start the search again:
  - skip OCCUPIED items not matching the key
  - this way we can reuse the DELETED spots

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



Implementation

- Array of linked lists implementation
  - The data structure:

```
class HashTable  {
private:
  static const int SIZE = 100;
  struct Node {
    int key;
    node *nextNode;
  };
  Node* pTable[SIZE];  //array of pointers to Nodes
  . . .
};

// constr should init all pointers in the array to NULL
```

Separate Chaining

- To insert a an object:
  - compute hash(k)
  - if the object is not already in the list at that location, insert the object into the list.
- 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

Sample Problems

Union operation: For each of the four set implementations, define the set operation. Given two sets, return a third set that is the union of the two sets. Assume the insert operation is already defined (or define it first):

- Set of floats: Use an array of **double** to store the set elements.
- Set of letters: Use an array of **boolean** to store the set. Assume the set can contain only uppercase letters (char)
- Set of ints: Use a hash table of size 100 to store the set. The keys will be 4 digit integers.
- Set of ints: Use a hash table with Separate Chaining.