Sets & Hash Tables

Week 13

Weiss: chapter 20

CS 5301
Spring 2018

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 Set1 is a subset of Set2 (if all elements of the Set1 are also elements of Set2).
- Equals(Set1,Set2): boolean
  - True, if both sets contain exactly the same elements.

Implementation

- Array of elements implementation
  - each element of the set will occupy a position in the array.
  - the member (find) operation will be inefficient, must use linear search.

```
class IntSet {
  int count;    //number of elements in the set, set to 0 in constr
  int intSet[100];  //stores the elements in positions 0..count
}
```

  - insert must not add duplicates:

```
void insert(int x) {
   if (!member(x) && count<100) {
      intSet[count] = x;
      count++;
   }
}
```


Implementation

- Array of elements implementation: member

```
bool member(int x) {
   for (int i=0; i<count; i++) {
        if (intSet[i]==x)
            return true;
    }
    return false;
}
```

- Array of elements implementation: union

```
IntSet operator+(IntSet rhs) {
    IntSet newSet;
    for (int i=0; i<count; i++)
        newSet.insert(intSet[i]);
    for (int i=0; i<rhs.count; i++)
        newSet.insert(rhs.intSet[i]);
    return newSet;
}
```

  - Exercise: implement all of the set operations for the IntSet.

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

(Hash(6702) + 1) % 100 = 3

But values[3] is already occupied.

(Hash(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

- Simple array implementation
  - keys are ints, all greater than or equal to 0:

```
class HashTable {
private:
    int *array;              // array of int elements
                             // use -1 to indicate empty slot
    int size;                // size of array
    int hash (int  key) ;    // maps key to position in array
public:
    HashTable (int size);    //initialize all elements to -1
    ~HashTable();

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


Implementation

- Simple array implementation:

```
HashTable::HashTable (int s) {
    size = s;
    array = new int[size];          //dynamic allocation
    for (int i=0; i<size; i++) {    //set all values to -1
        array[i] = -1;
    }
}

int HashTable::hash(int key) {
    return key % size;              //maps keys to array position
}

void HashTable::insert ( int element) {
    int index = hash(element);       //linear probing, if not at index
    while (array[index]!=-1 && array[index] != element) {
        index = (index+1)%size;
    }
    array[index] = element;          //puts element at first open slot
}
```

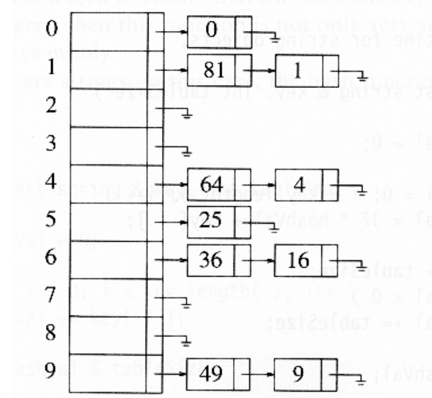
Collision Resolution:
Linear Probing

- Insert: When there is a collision, search sequentially for the next open slot (-1)
  - Put the value in the table at that position
- Find: if the key is not at the hashed location, keep searching sequentially for it.
  - if it reaches an open slot (-1), 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 -1
- 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.

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 ChainedTable  {
private:
  static const int SIZE = 10;
  struct Node {
    int key;
    node *nextNode;
  };
  Node* table[SIZE];  //array of pointers to Nodes
  int hash (int  key) ;  // maps key to position in array

public:
  ChainedTable();   //inits all pointers in array to NULL
  bool find(int);       //return true if int in table
  void insert (int);    //add int to table
   . . .
};
```


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