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

- **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).

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.

```java
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
}

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

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

```java
//This array will represent a set of days of the week
// (Sunday, Monday, Tuesday, ...)
bool daysOfWeek[7] = {false}; //sets all elements to false

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

Implementation

- **Array of elements implementation: member**

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

- **Array of elements implementation: union**

```java
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.
Implementation

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

```cpp
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.

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

**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

Next place part number 6702 in the array.

Hash(partNum) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Empty</td>
</tr>
<tr>
<td>1: 4501</td>
</tr>
<tr>
<td>2: 5502</td>
</tr>
<tr>
<td>3: 7803</td>
</tr>
<tr>
<td>4: Empty</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>97: Empty</td>
</tr>
<tr>
<td>98: 2298</td>
</tr>
<tr>
<td>99: 3699</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Empty</td>
</tr>
<tr>
<td>1: 4501</td>
</tr>
<tr>
<td>2: 5502</td>
</tr>
<tr>
<td>3: 7803</td>
</tr>
<tr>
<td>4: Empty</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>97: Empty</td>
</tr>
<tr>
<td>98: 2298</td>
</tr>
<tr>
<td>99: 3699</td>
</tr>
</tbody>
</table>

Collision resolved

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.

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Empty</td>
</tr>
<tr>
<td>1: 4501</td>
</tr>
<tr>
<td>2: 5502</td>
</tr>
<tr>
<td>3: 7803</td>
</tr>
<tr>
<td>4: 6702</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>97: Empty</td>
</tr>
<tr>
<td>98: 2298</td>
</tr>
<tr>
<td>99: 3699</td>
</tr>
</tbody>
</table>

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**: (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:
    ```cpp
    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
        // do not implement remove
    };
    ```

Hash Function

- **Goals**:
  - computation should be fast
  - should minimize collisions (good distribution)
- **Final step of hash function is usually**: `temp % size`
  - `temp` is some intermediate result (or initial key value)
  - `size` is the hash table size
  - ensures the value is a valid location in the table
- **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)

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

<table>
<thead>
<tr>
<th>Hash function is still: h(K) = k % 10</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>81</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>9</td>
</tr>
</tbody>
</table>

Implementation

- Array of linked lists implementation
  - The data structure:

```cpp
class ChainedTable {
private:
    static const int SIZE = 101;
    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