CS211, LECTURE 19  
LINEAR SEARCH STRUCTURES

ANNOUNCEMENTS:

OVERVIEW:
- search structure reminder
- sorted arrays, sorted lists
- sets, sorted sets
- multisets/bags
- dictionary/map
- hashtables

1. Motivation
1.1 ADT Reminder
- ADT: data and operations
- interface gives functionality of ADT
- ADT uses variety of foundational structures

1.2 ADT Categories
- Search Structures
- Sequence Structures

1.3 Search Structures
- large collection of ordered data
- need to find something

1.4 Linear Search Structures
- arrays, lists, dictionaries, hashtables

1.5 Interface to implement

```java
interface SearchStructure {
    void insert(Object o);
    void delete(Object o);
    boolean search(Object o);
    int size();
}
```

2. Sorted Arrays

2.1 Why sorted?
- linear search is $O(n)$: OK
- binary search is $O(\log n)$: better!

2.2 Data
- items in array, current element, count

2.3 Operations
- search: if find item $i$, return true; use binary search
- insert (allowing for repeats):
  - search for item to set cursor (where item should go)
  - make array bigger if more space needed (sadly, $O(n)$)
  - move items higher than $i$ to the right
  - insert item
- delete (for repeated values, $O(n)$):
  - find each occurrence of item $i$
  - shuffle items to the left, overwriting item $i$

2.4 More? Chap 7 in DS&A; 7.5 in DS&SD

2.5 Example

```java
public class TestSortedArray {
    public static void main(String[] args) {
        SortedArray sa =
                new SortedArray(Integer.parseInt(args[0]));
        System.out.println(sa);
        sa.insert("A");
        System.out.println(sa);
        sa.insert("B");
        System.out.println(sa);
        sa.insert("B");
        System.out.println(sa);
        sa.insert("D");
        System.out.println(sa);
        sa.insert("A");
        System.out.println(sa);
        sa.insert("C");
        System.out.println(sa);
        System.out.println("SA size: " + sa.size());
    }
}
```

/* Sample session:
 > java TestSortedArray 3
 [null, null, null]
 [A, null, null]
 [A, B, null]
 [A, B, B]
 [A, B, B, D]
 [A, A, B, B, C, D]
 SA size: 6
*/
import java.util.Arrays;
class SortedArray implements SearchStructure {
    private Comparable[] a;
    private int size; // max size of array
    private int cursor; // position to insert element
    private int count; // current count of elements
    // constructor, getters, setters
    // more methods: isFull, isEmpty, size
    public void sort() { Arrays.sort(a); }
    public String toString() {
        return Arrays.asList(a).toString();
    }
    public boolean search(Object key) {
        int left = 0;
        int right = count - 1;
        int middle = 0;
        while (left <= right) {
            middle = (left + right)/2;
            int test = a[middle].compareTo(key);
            if (test < 0) left = middle+1;
            else if (test > 0) right = middle-1;
            else { cursor = middle; return true; }
        }
        // If we reach here, we didn't find key:
        cursor = right + 1;
        return false;
    }
    public void insert(Object key) {
        search(key);
        int oldcount = count;
        count++;
        if (this.isFull()) {
            Comparable[] aNew = new Comparable[count];
            System.arraycopy(a,0,aNew,0,size);
            a = aNew;
            size = a.length;
        }
        for (int i = oldcount-1; i >= cursor; i--)
            a[i+1] = a[i];
        a[cursor] = (Comparable) key;
    }
    public void delete(Object key) {
        int same=0;
        for (int i = 0; i < size; i++)
            if (a[i].equals(key)) same++;
        if (same > 0) {
            int oldsize = size;
            size = size - same;
            Comparable[] aNew = new Comparable[size];
            int newindex = 0;
            for (int i = 0; i < oldsize; i++) {
                if (!a[i].equals(key))
                    aNew[newindex]=a[i];
                newindex++;
            }
        }
    }
}

3. Lists for Searching
3.1 Sorted?
• no, not needed
• but we’ll delete repeats this time
3.2 Data:
• head of list
• cursors for current and previous nodes
3.3 Operations:
• search: linear search (O(n))
  - starting at head, look at each element
  - set current and previous cursors during search
• insert:
  - check if item already exists
  - if not, make item the head of the list
• delete:
  - if not in list, return false (search sets cursors)
  - if in list and previous is null, remove head
  - else, connect the previous node to the current’s next

public class TestSearchList {
    public static void main(String[] args) {
        SearchList s = new SearchList();
        s.insert(new Integer(5));
        s.insert(new Integer(1));
        s.insert(new Integer(2));
        System.out.println(s);
        s.delete(new Integer(5));
        System.out.println(s);
        System.out.println(s.size());
        System.out.println(s.search(new Integer(1)));
    }
}

/* Output:
 2 1 5
2 1
2
true
*/
class SearchList implements SearchStructure {
    private ListNode head;
    private int size;
    private ListNode current; // set during search
    private ListNode previous; // set during search
    // constructors, inner class, more methods
4. Sets

4.1 What’s a set?
- unordered collection
- no repeats

4.2 ADT?
- searchable (insert, delete, contains, size, …)
- linear

4.3 Implementation?
- could use SearchList or modify SortedArray: be unsorted and not contain repeats, nulls
- see also DS&A, Chap 12

4.4 API
- constants: Collections.EMPTY_SET
- interfaces: Set, SortedSet
- abstract classes: AbstractSet
- concrete classes: BitSet, TreeSet, HashSet, LinkedHashSet

5. Multisets/Bags

5.1 Bag o’ what?
- **bag** (or multiset) is an unordered set with repeats
- so, think set with repeats

5.2 Why?
- shows up in Collections API: “Bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly.”
- See DS&A, 12.3 (wrap an array)
6. Dictionaries

6.1 Definitions
• big book with words and definitions
• modern approach: http://history.cc.ukans.edu/carrie/reference/dictionaries.html

6.2 Motivation by example (word search)
• Find meaning of Muggle
• Linear search: look up every word and compare
• Binary search: sort all the words and then compare
• how many words? lots!
• Big-Oh? Oh no…

6.3 Better process?
• quick way to find word means quick retrieval of definition
• use existing databases (books, online, files, …)
• inspect words? find letters? put words in certain locations? …
• possible to vastly improve lookup time

6.4 Improvements
• use array for constant time (O(1)) find and insert
• need way to convert word to integer for array pos
• example) consider first character of string?

6.5 Key-Value Pair
• more formal defs for “word” and “definition”
• Key: symbol that labels information
• Value: that information
• Operation: Find key in collection and retrieve value(s) associated with that key.
  example) Muggle is a word (a key) that has a definition (a value).

6.6 ADT Definition
• Dictionary: From DADS, “An abstract data type storing items, or keys, associated with values. Basic operations are insert, find, and delete.”
• related terms: map, table, look-up table
• API uses Map (Dictionary is obsolete)

7. Hashtable

7.1 Definition of hashtable:
• Special kind of dictionary: “dictionary in which keys are mapped to array positions by a hash function…”
• process of building:
  - convert keys to integers
  - integers must be indicies of array
  - store keys in array using indicies
• needs:
  - function to convert object (key) to integer
  - array for indexing
  - elements of array to handle collisions (multiple keys hash to same index)

7.2 Collision
• Hash function plays “games” with objects to make integers…
• Having more than one key map to the same position is called a collision
• chained hashing: use linked lists for each array element
7.3 Example

- want to store Student info (Student objects)
- use IDs as keys
- each key has values, like Student name
- need to insert keys by hashing (how?)

```
IDs:
  37123
  37980
  37556
  37406
  37553
  37643
  37778
```

7.4 Rough Implementation of HashTable

```java
abstract class MyHashTable implements SearchStructure {
    private LinkedList[] table; // array of buckets
    private int size; // size of array

    // Create HashTable for input size:
    public MyHashTable(int size) {
        // create table
        createListsInTable();
        // record size
    }

    // Return size of HashTable:
    public int size() { return size; }

    // Methods to implement:
    public void insert(Object key) {
        // hash key to get bucket number
        // append key to list at that bucket
        table[hash(key)].add(key);
    }

    public void delete(Object key) {
        // hash key to get bucket number
        // walk list at that bucket and remove ID
        table[hash(key)].remove(key);
    }

    public boolean search(Object key) {
        // hash key to get bucket number
        // look for key by walking list at bucket
        return table[hash(key)].contains(key);
    }

    public int hash(Object key) {
        // convert key into an integer
        return (((Integer) key).intValue() % 10);
    }

    public String toString() {
        String s = "";
        for (int row = 0; row < size; row++)
            s += "table[row]+"+\n";
        return s;
    }
}
```

7.5 “Complete” Implementation

```java
import java.util.*;

class MyHashTable implements SearchStructure {
    private int size;
    public MyHashTable(int size) {
        table = new LinkedList[size];
        this.size = size;
        for (int row = 0; row < size; row++)
            table[row] = new LinkedList();
    }

    public int size() { return size; }
    public void insert(Object key) {
        table[hash(key)].add(key);
    }
    public void delete(Object key) {
        table[hash(key)].remove(key);
    }
    public boolean search(Object key) {
        return table[hash(key)].contains(key);
    }
    public int hash(Object key) {
        return (((Integer) key).intValue() % 10);
    }
    public String toString() {
        String s = "";
        for (int row = 0; row < size; row++)
            s += row+"+"+\n";
        return s;
    }
}
```

```java
public class FindName {
    public static void main(String[] args) {
        SearchStructure students = new MyHashTable(10);
        students.insert(new Integer(37123));
        students.insert(new Integer(37980));
        students.insert(new Integer(37556));
        students.insert(new Integer(37406));
        students.insert(new Integer(37553));
        students.insert(new Integer(37643));
        students.insert(new Integer(37778));
        System.out.println(students);
    }
}
```

/* Output:
0: [37980]
1: []
2: []
3: [37123, 37553, 37643]
4: []
5: []
6: [37556, 37406]
7: []
8: [37778]
9: []
*/
### 7.6 Storing Objects

- want to store objects to find corresponding values to keys
- **Object** supplies `int hashCode()`, which returns a hash code for the current object
- hash code usually related to an objects address (toString calls hash code…)
- for you to override (see wrapper classes for examples)
- must return same integer for same object
- equal objects (see **equals**) must return same hash code
- unequal objects may or may not return same hash code
- to store/retrieve object, call its `hashCode` method and use that integer to determine the bucket address

### 7.7 Hash Functions

- \( h(x) = x \)
- \( h(x) = x \mod 2 \)
- \( h(x) = |x| \mod \text{primenumber} \)
- and more …

### 7.8 Extremes

- Map each item to its own bucket, with no empty buckets
  - Zero collisions and no wasted space
  - perfect!
- Map all items to the same bucket, no empty buckets
  - Maximal collisions, single bucket
  - what have you gained by hashing in this way? essentially, linked-list
- Map all items to the same bucket, many empty buckets
  - Maximal collisions!
  - as bad as above and you’re wasting space

### 7.9 Real Life

- some non-zero number of collisions
- some empty buckets
  - Collisions aren’t always bad
  - Collisions are usually a tradeoff against wasted space.
- rules of thumb:
  - Reduce number of buckets: increase collisions, reduce space consumption
  - Increase number of buckets: reduce collisions, increase space consumption

### 7.10 Load Factor and Capacity

- **Capacity**: number of buckets
- **Load factor**: how full the hash table is allowed to get
- API: “When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.”
- API: “As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs.”

### 7.11 API

- **Hashtable**
  - not synchronized
  - no **nulls**
  - seems unpopular…
- **HashMap**
  - has **Map** operations
  - ordering not guaranteed
  - allows **null**
  - not synchronized
- **HashSet**
  - has **Set** properties
  - “uses” **HashMap** (extends?)
  - ordering not guaranteed
  - allows **null**
- **LinkedHashMap, LinkedHashSet**
8. API Example

```java
import java.util.HashMap;
public class APIHashMap {
    private static String[] keys = {"A","B","C"};
    private static int[] vals = {1, 2, 3};
    private static HashMap stuff = new HashMap();
    public static void main(String[] args) {
        fillHashMap();
        showKeys();
        showValues();
        retrieveValues();
    }
    private static void fillHashMap() {
        for (int i=0 ; i < keys.length ; i++)
            stuff.put(keys[i],new Integer(vals[i]));
    }
    private static void showKeys() {
        System.out.println(stuff.keySet());
    }
    private static void showValues() {
        System.out.println(stuff.values());
    }
    private static void retrieveValues() {
        for (int i=0 ; i < keys.length ; i++)
            System.out.print(stuff.get(keys[i]) + " ");
        System.out.println();
    }
}
/* Output:
[C, B, A]
[3, 2, 1]
1 2 3*/
```

9. Exercises

- Rewrite `TestSortedArray.java` to remove repeated items.
- Rewrite `TestSortedArray.java` to use a ragged 2D array in which each row has a uniform maximum size. So, after the first row is filled, a new row is created, it becomes the second row, and new items are placed within it.
- Rewrite `TestSortedList.java` for a doubly-linked list.
- Rewrite (or extend or rewrite) `FindName.java` to return a student’s name for its supplied ID. So, the IDs remain the keys, but you need to store key-value pairs inside the table, not just the keys. I suggest one of these options: giving up on implementing `SearchStructure.java` so you can have methods, like `add(Object key, Object value)`; or, using a `Student` object instead of `key` for insert/delete/search. You will still need to have a method to return the result of your search. Use `Object`'s `hashCode` for your hash function. Be sure to test your program with sample data.