

CS 61B: Lecture 22
Friday, March 13, 2009

Today's reading: Goodrich & Tamassia, Chapter 5.

BASIC DATA STRUCTURES

=====

Stacks

A `_stack_` is a crippled list. You may manipulate only the item at the top of the stack. The main operations: you may "push" a new item onto the top of the stack; you may "pop" the top item off the stack; you may examine the "top" item of the stack. A stack can grow arbitrarily large.

```

| | -pop()-> | | -push(c)-> | | -size()-> 2 |d| -top()-> d | |
|a|         |a|         |a| -push(d)--> |a| --pop() x 3--> | | -top()--
|---|         |---|         |---|         |---|         |---|
v             v             v             v             v
b             b             b             b             b
EmptyStackException

```

```

public interface Stack {
    public int size();
    public boolean isEmpty();
    public void push(Object item);
    public Object pop() throws EmptyStackException;
    public Object top() throws EmptyStackException;
}

```

In any reasonable implementation, all these methods run in $O(1)$ time.

A stack is easily implemented as a singly-linked list, using just the `front()`, `insertFront()`, and `removeFront()` methods.

Why talk about Stacks when we already have Lists? Mainly so you can carry on discussions with other computer programmers. If somebody tells you that an algorithm uses a stack, the limitations of a stack give you a hint how the algorithm works.

Sample application: Verifying matched parentheses in a String like

"{[(){}]}()". Scan through the String, character by character.

- o When you encounter a lefty--'{' , '[' , or '('--push it onto the stack.
- o When you encounter a righty, pop its counterpart from atop the stack, and check that they match.

If there's a mismatch or exception, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.

Queues

A `_queue_` is also a crippled list. You may read or remove only the item at the front of the queue, and you may add an item only to the back of the queue. The main operations: you may "enqueue" an item at the back of the queue; you may "dequeue" the item at the front; you may examine the "front" item. Don't be fooled by the diagram; a queue can grow arbitrarily long.

```

===             ===             ===             === -front()-> b
ab. -dequeue()-> b.. -enqueue(c)-> bc. -enqueue(d)-> bcd
===             ===             ===             === -dequeue() x 3--> ===
|               |               |               |
v               v               v               v
a               a               a               a
EmptyQueueException <-front()-- ===

```

Sample Application: Printer queues. When you submit a job to be printed at a selected printer, your job goes into a queue. When the printer finishes printing a job, it dequeues the next job and prints it.

```

public interface Queue {
    public int size();
    public boolean isEmpty();
    public void enqueue(Object item);
    public Object dequeue() throws EmptyQueueException;
    public Object front() throws EmptyQueueException;
}

```

In any reasonable implementation, all these methods run in $O(1)$ time. A queue is easily implemented as a singly-linked list with a tail pointer.

Dequeues

A `_deque_` (pronounced "deck") is a Double-Ended QUEUE. You can insert and remove items at both ends. You can easily build a fast deque using a doubly-linked list. You just have to add `removeFront()` and `removeBack()` methods (Goodrich and Tamassia call them `removeFirst()` and `removeLast()`), and deny applications direct access to list nodes. Obviously, dequeues are less powerful than lists whose list nodes are accessible.

DICTIONARIES
=====

Suppose you have a set of two-letter words and their definitions. You want to be able to look up the definition of any word, very quickly. The two-letter word is the `_key_` that addresses the definition.

Since there are 26 English letters, there are $26 * 26 = 676$ possible two-letter words. To implement a dictionary, we declare an array of 676 references, all initially set to null. To insert a Definition into the dictionary, we define a function `hashCode()` that maps each two-letter word (key) to a unique integer between 0 and 675. We use this integer as an index into the array, and make the corresponding bucket (array position) point to the Definition object.

```
public class Word {
    public static final int LETTERS = 26, WORDS = LETTERS * LETTERS;
    public String word;

    public int hashCode() {
        return LETTERS * (word.charAt(0) - 'a') + (word.charAt(1) - 'a');
    }
}

public class WordDictionary {
    private Definition[] defTable = new Definition[Word.WORDS];

    public void insert(Word w, Definition d) {
        defTable[w.hashCode()] = d;
    }

    Definition find(Word w) {
        return defTable[w.hashCode()];
    }
}
```

Returning to our dictionary: what if we want to store every English word, not just the two-letter words? The table "defTable" must be long enough to accommodate pneumonoultramicroscopicvolcanoconiosis, 45 letters long. Unfortunately, declaring an array of length 26^{45} is out of the question. English has fewer than one million words, so we should be able to do better.

Hash Tables (a fast implementation of dictionaries)

Suppose n is the number of keys (words) whose definitions we want to store, and suppose we use a table of N buckets, where N is perhaps a bit larger than n , but much smaller than the number of `_possible_` keys. A hash table maps a huge set of possible keys into N buckets by applying a `_compression_function_` to each hash code. The obvious compression function is

$$h(\text{hashCode}) = \text{hashCode} \bmod N.$$

Hash codes are often negative, so remember that mod is not the same as Java's "%" operator. If you compute `hashCode % N`, check if the result is negative, and add N if it is.

With this compression function, no matter how long and variegated the keys are, we can map them into a table whose size is not much greater than the actual number of entries we want to store. However, we've created a new problem: several keys are hashed to the same bucket in the table if $h(\text{hashCode1}) = h(\text{hashCode2})$. This circumstance is called a `_collision_`.

How do we handle collisions without losing entries? We'll discuss that next lecture.