CS 61B Data Structures and Programming Methodology

July 21, 2008

David Sun

Hash Code and Compression Function

- How do we design a “good” hash code and compression function?
  - Unfortunately it’s a bit of a black art.
  - Ideally, hash code and compression function maps each key to a *uniformly distributed random* bucket from zero to *N-1* for any input.
  - Note: random does not mean that the hash code gives a random value each time. Hash code on the same object should return the same value each time!

A Bad Compression Function

- Consider integers:
  - Try *hashCode(i) = i*.
  - Then *h(hashCode) = hashCode mod N* where *N* is 10000.
  - What's wrong with this?
- Consider an application that ***only*** generates integer divisible by 4:
  - Any integer divisible by 4 mod 10000 is divisible by 4.
  - Three quarters of the buckets are wasted!

A Better Compression Function

- Setting the number of buckets to a prime number:
  - The same compression function is much better if N is prime.
  - Even if the hash codes are always divisible by 4, numbers larger than N often hash to buckets not divisible by 4, so all the buckets can be used.

Good Hash Code: Strings

Only 32 bits

Large enough prime

```
private static int hashCode(String key) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++) {
        hashVal = (127 * hashVal + key.charAt(i)) % 16908799;
    }
    return hashVal;
}
```

- Treat a string as a base 127 number and convert it to an internal integer.
- By multiplying the hash code by 127 before adding in each new character to ensure that each character has a different effect on the final result.
- The "%" operator with a prime number tends to "mix up the bits" of the hash code. The prime is chosen to be large, but not so large that *127 * hashVal + key.charAt(i)* will cause an overflow of int.

Bad Hash Code: Strings

- Sum up the ASCII values of the characters.
  - The sum will rarely exceed 500.
  - Most of the entries will be bunched up in a few hundred buckets.
  - Anagrams like "pat," "tap," and "apt" will collide. Ordering information is not considered
- Use the first three letters of a word, in a table with 26^3 buckets.
  - Systematic bias in the English language, e.g., words beginning with "pre" are much more common than words beginning with "xzq", and the former will be bunched up in one long list.
- Consider the good hash function. Suppose the prime modulus is 127 instead of 16908799.
  - The return value is just the last character of the word, because (127 * hashVal) % 127 = 0.
  - That's why 127 and 16908799 were chosen to have no common factors.

Hashing Lists

- Lists (ArrayList, LinkedList, etc.) are analagous to strings. Sum the weighted hash values of each item in the list, weigh the item's the hashcode by it's position in the list.

```
hashCode = 1; Iterator i = list.iterator();
  while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31*hashCode
          + (obj==null ? 0 : obj.hashCode());
}
```

- Can limit time spent computing hash function by not looking at entire list. For example: look only at first few items.

Hashing Trees

- Recursively defined data structures) recursively defined hash code.
- For example, on a binary tree, one can use something like

```
int hashCode(T) {
  if (T == null)
   return 0;
  else return someHashFunction (T.key ())
   + 255 * hash(T.left ())
   + 255*255 * hash(T.right ());
}
```

Hashing in Java

- In class `Object`, is function `hashCode()`
  - Inherited by every class.
  - By default, returns address of `this`.
  - Can override it for your particular type.
- Rule:
  - If `x.equals(y)` then `x.hashCode() == y.hashCode()`.
  - The inverse need not to be true.
- The types `Hashtable`, `HashSet`, and `HashMap` use hashCode to give you fast look-up of objects.

```
HashMap<KeyType,ValueType> map =
          new HashMap<KeyType,ValueType> (approximate size,
                                          load fac-tor);
map.put (key, value);
// VALUE last mapped to by SOMEKEY.
map.get (someKey)
// VALUE last mapped to by SOMEKEY.
map.containsKey (someKey)
// Is SOMEKEY mapped?
map.keySet () // All keys in MAP (a Set)
```

Rehashing

- To get constant-time lookup, need to keep #buckets within constant factor of #items.
- Sometimes we can't predict in advance how many entries we'll need to store.
  - If the load factor n/N (entries per bucket) gets too large, we are in danger of losing constant-time performance.
- Enlarge the hash table when the load factor gets higher than some limit (typically larger than 0.75).
  - Allocate a new array (typically at least twice as long as the old)
  - Walk through all the entries in the old array and rehash them into the new.
  - This operation costs constant time per item.

Stacks

- A list that you can only put and take elements from one end.
  - Think about a stack of papers, or plates in a restaurant, or boxes in a garage or closet.
- Main Operations
  - Push a new item onto the top of the stack;
  - Pop the top item off the stack;
  - Peek at the top item of the stack.
- In any reasonable implementation, all these operations run in *O(1)* time, e.g., using a linked list (see the text book for an implementation).

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

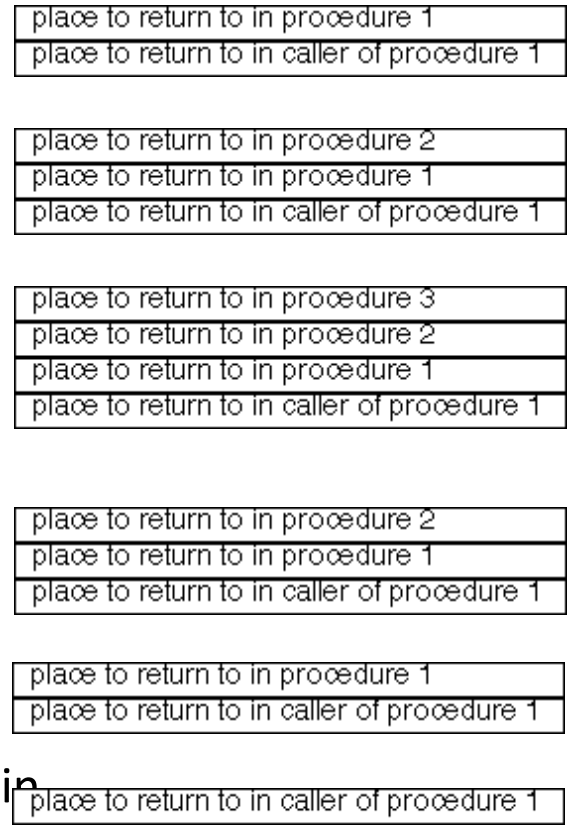
Sample Application

- When parsing an arithmetic expression, the compiler needs to verifying matched parentheses like "`{ [ ( ) { [ ] } ] ( ) }`"
  1. Scan through the String, character by character.
  2. When you encounter a lefty '{', '[', or '(' push it onto the stack.
  3. When you encounter a righty, pop its counterpart from top the stack, and check that they match.
  4. 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.

Sample Application

```
public void procedure1() {
    . . .
    procedure2();
    . . .
}
public void procedure2() {
    . . .
    procedure3();
    . . .
}

public void procedure3() {
    . . .
    procedure4();
    . . .
}
```



Procedure 1 calls procedure 2

Procedure 2 calls procedure 3

Procedure 3 calls procedure 4

Procedure 4 returns

Procedure 3 returns

Procedure 2 returns

- Each procedure call pushes the place in the program to return to on a stack maintained by the operating system.
- Each procedure return pops the stack to find out where to return to.

Queues

- A queue is a list where you can only insert and delete items from the front and the end, not the middle.
- Main operations:
  - Enqueue an item at the back of the queue;
  - Dequeue the item at the front;
  - Examine the "front" item.
- In any reasonable implementation, all these methods run in O(1) time, e.g., using a linked list with a tail pointer.

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

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.

Tree Iterator

- Returning the elements of a tree one by one, using an iterator.

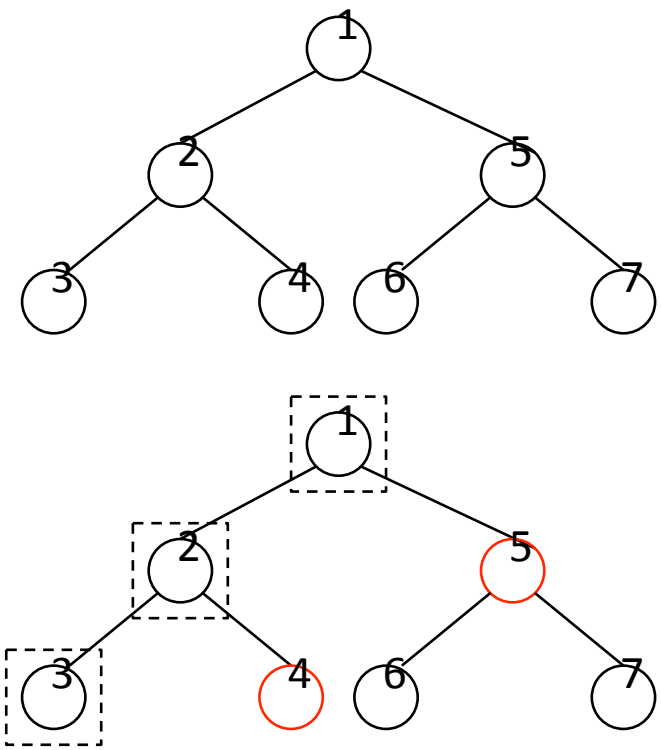
```
// Are there more tree elements yet to be returned?
boolean hasMoreElements ( );
// Return the next element.
// Precondition: hasMoreElements ( );
// throws NoSuchElementException when
// that precondition is not met.
Object nextElement ( );
```

- Must decide what information to maintain to let us find the next tree element to return.

Example

- The first element to be returned is the one labeled "1". We need to somewhere keep track of the fact that we have to return to element "5".
- Similarly, once we return element "2", we have to remember that element "4" is yet to return.
- The state-saving information must include a *collection* of "bookmarks" to nodes we've passed along the way.

Do a pre-order walk on the tree



Maintaining the Fringe

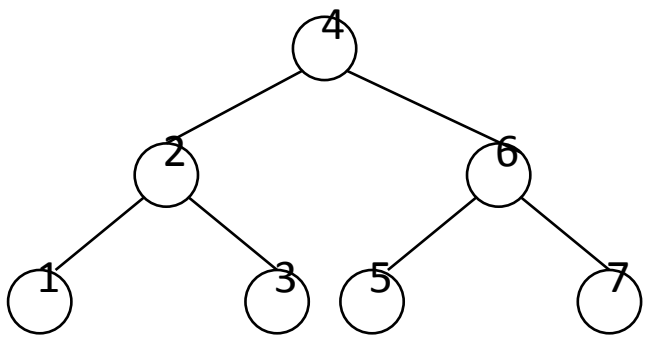
- Maintain a *fringe* or *frontier* of all the nodes in the tree that are candidates for returning next.
  - The `nextElement` method will choose one of the elements of the fringe as the one to return, then add its children to the fringe as candidates for the next element to return.
  - `hasMoreElements` is true when the fringe isn't empty.
- The iteration sequence will then depend on the order we take nodes out of the fringe.
  - Depth-first iteration results from storing the fringe elements in a *stack*, a last-in first-out structure.
  - Breath-first iteration results from storing the fringe elements in a queue.

Tree Iterator

```
public class DepthFirstIterator implements Enumeration { private
   Stack fringe = new Stack ( );
   public DepthFirstIterator ( ) {
     if (myRoot != null) { fringe.push (myRoot); }
   }
   public boolean hasMoreElements ( ) {
     return !fringe.empty ( );
   }
   public Object nextElement ( ) {
     if (!hasMoreElements ( )) {
          throw new NoSuchElementException ("tree ran out of
   elements");
     }
     TreeNode node = (TreeNode) fringe.pop ( );
     if (node.myRight != null) {fringe.push (node.myRight);}
     if (node.myLeft != null) {fringe.push (node.myLeft);}
     return node;
     }
   }
```

Quiz

- What is the maximum number of nodes that the stack will contain during a depth-first traversal of the tree above, and where in the traversal does the stack contain the maximum number of nodes?
- What is the maximum number of nodes that the queue will contain during a breadth-first traversal of the tree above, and where in the traversal does the queue contain the maximum number of nodes?



Reading

- Objects, Abstraction, Data Structures and Design using Java 5.0
  - Chapter 5: pp257 – 277,
  - Chapter 6: pp313 - 317,