Eric Roberts
CS 106B

Handout #48
May 20, 2009

Implementing Sets

In Our Last Episode . . .

- I reviewed the basics of mathematical set theory and used that theory to design the interface for a general `Set` class.
- In the first five minutes or so of today's class, I will develop an implementation of the `Set` class based on the generic `Set` class we built last week.
- For the rest of the day, I will talk about optimizations to that implementation along with some general strategies about how to make implementations as efficient as possible.

Contents of the `set.h` Interface

```
template <typename ElemType>
class Set {
public:
   Set(int (*cmpfn)(ElemType, ElemType) = GenericCompare);
   ~Set();
   int size();
   bool isEmpty();
   void clear();
   void add(ElemType elem);
   void remove(ElemType elem);
   bool contains(ElemType elem);
   bool equals(Set & s);
   bool isSubsetOf(Set & s);
   void unionWith(Set & s);
   void intersect(Set & s);
   void subtract(Set & s);
   Iterator iterator();

   template <typename ElemType>
   void mapAll(void (*fn)(ElemType elem, ElemType data),
               ElemType data);

private:
#include "vector.h"
};
#include "setimpl.cpp"
#endif
```

The Easy Implementation

- As is so often the case, the easy way to implement the `Set` class is to build it out of data structures that you already have. In this case, it make sense to build `Set` on top of the `Vector` class.
- The private section looks like this:

```
/*
 * File: vector.h
 * --------------
 * This file contains the private section for the Vector interface.
 */

/* Instance variables */

   VectorElemType *vec;      /* the vector representing the data */
```

The `setimpl.cpp` Implementation

```
template <typename ElemType>
VectorSet<ElemType>::VectorSet(int (*cmp)(ElemType, ElemType)) : Set(cmp) {
   /* empty */
}

template <typename ElemType>
VectorSet<ElemType>::~VectorSet() {
   /* empty */
}

template <typename ElemType>
int VectorSet<ElemType>::size() {
   return vec.size();
}

template <typename ElemType>
bool VectorSet<ElemType>::isEmpty() {
   return vec.isEmpty();
}

template <typename ElemType>
void VectorSet<ElemType>::add(ElemType elem) {
   vec.add(elem);
}

...and more...
```

The `setimpl.cpp` Implementation

```
/*
 * Implementation notes: set operations
 * ------------------------------------
 * The functions isSubsetOf, unionWith, intersect, and subtract
 * are similar in structure.  Each one uses an Iterator to walk over
 * the appropriate sets.
 */

template <typename ElemType>
bool VectorSet<ElemType>::isSubsetOf(Set & s) {
   Iterator iter = iterator();
   while (iter.hasNext()) {
      if (!s.contains(iter.next())) return false;
   }
   return true;
}

template <typename ElemType>
bool VectorSet<ElemType>::equals(Set & s) {
   return isSubsetOf(s) && s.isSubsetOf(*this);
}
```

Initial Versions Should Be Simple

> Premature optimization is the root of all evil.
> —Don Knuth

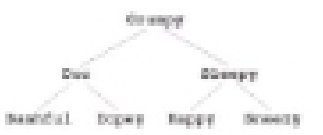
- When you are developing an implementation of a public interface, it is usually best—at least as a first cut—to write the simplest possible code that satisfies the requirements of the interface.
- This approach has several advantages:
  - You can get the package out to clients much more quickly.
  - Simple implementations are much easier to get right.
  - You often won't have any idea what optimizations are needed until you have experiential data from clients of that interface. In terms of overall efficiency, some optimizations are much more important than others.

Optimizing the Binary Search Tree

- If you code the `Set` implementation using the simple "let `BST` do all the work" strategy, it will probably not be surprising to discover that the necessary optimizations are in the `BST` class itself.
- In class last Friday, I didn't go through one of the most important topics in the discussion of binary search trees: the question of whether a tree is balanced. Binary search trees deliver O(log *N*) performance only if the left and right branches of each node are roughly comparable in height. The code as I presented it in lecture does nothing to ensure that property.
- Leaving this issue until today might well mirror the discovery of such problems in the real world. The `BST` class gets far less direct use than the `Set` class, so it is likely that performance problems would show up only when clients began using `Set`s.

A Question of Balance

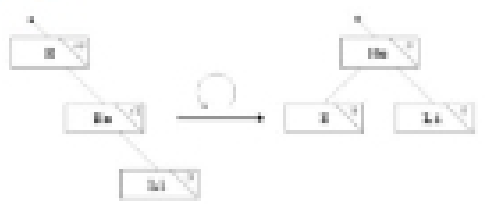
- Ideally, a binary search tree containing the names of Disney's seven dwarves would look like this:



- If, however, you happened to enter the names in alphabetical order, this tree would end up being a simple linked list in which all the left subtrees were `NULL` and the right links formed a simple chain. Algorithms on that tree would run in O(*N*) time instead of O(log *N*) time.
- A binary search tree is **balanced** if the height of its left and right subtrees differ by at most one and if both of those subtrees are themselves balanced.

AVL Trees

- The text presents an algorithm designed by Georgii Adel'son-Vel'skii and Evgenii Landis for keeping a trees in balance. That algorithm is based on operations called **rotations** that try to keep a tree in balance. For example, a simple left rotation looks like this:



- Exercise: How would you write the `RotateLeft` function that performs this transformation? What would the prototype be? What is left out of this diagram that you need to think about?

Sets and Efficiency

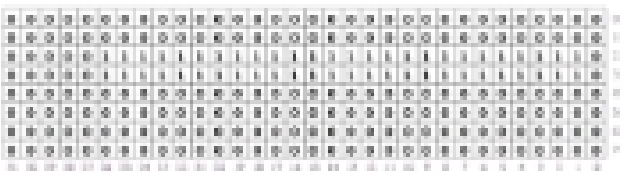
- After you release the set package, you might discover that clients use them often for particular types for which there are much more efficient data structures than binary trees.
- One thing you could do easily is check to see whether the element type was `string` and then use a `Lexicon` instead of a binary search tree. The resulting implementation would be far more efficient. This change, however, would be valuable only if clients used `Set<string>` often enough to make it worth adding the complexity.
- One type of sets that do tend to occur in certain types of programming is `Set<char>`, which comes up, for example, if you want to specify a set of delimiter characters for a scanner. These sets can be made astonishingly efficient as described on the next few slides.

Character Sets

- The key insight needed to make efficient character sets (or, equivalently, sets of small integers) is that you can represent the inclusion or exclusion of a character using a single bit. If the bit is a 1, then that element is in the set; if it is a 0, it is not in the set.
- You can tell what character value you're talking about by creating what is essentially an array of bits, with one bit for each of the ASCII codes. That array is called a **characteristic vector**.
- What makes this representation so efficient is that you can pack the bits for a characteristic vector into a small number of words inside the machine and then operate on the bits in large chunks.
- The efficiency gain is enormous. Using this strategy, a good compiler could perform a set operation in four instructions.

Bit Vectors and Character Sets

- This picture shows a characteristic vector representation for the set containing the upper- and lowercase letters:



Bitwise Operators

- If you know your client is working with sets of characters, you can implement the set operators extremely efficiently by storing the set as an array of bits and then manipulating the bits all at once using C++'s **bitwise operators**.
- The bitwise operators are summarized in the following table and then described in more detail on the next few slides:

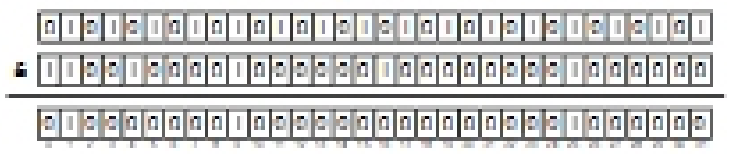
x & *y*	Bitwise AND. The result has a 1 bit wherever both *x* and *y* have 1s.
x | *y*	Bitwise OR. The result has a 1 bit wherever either *x* or *y* have 1s.
x ^ *y*	Exclusive OR. The result has a 1 bit wherever *x* and *y* differ.
~*x*	Bitwise NOT. The result has a 1 bit wherever *x* has a 0.
x << *n*	Left shift. Shift the bits in *x* left *n* positions, shifting in 0s.
x >> *n*	Right shift. Shift *x* right *n* bits (logical shift if *x* is unsigned).

The Bitwise AND Operator

- The bitwise AND operator (&) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which both *x* and *y* have 1 bits. A table for the & operator appears to the right.

&	0	1
0	0	0
1	0	1

- The primary application of the & operator is to select certain bits in an integer, clearing the unwanted bits to 0. This operation is called **masking**.
- In the context of sets, the & operator performs an intersection operation, as in the following calculation of **odds** ∩ **squares**:

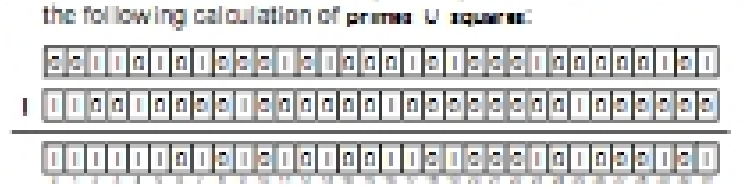


The Bitwise OR Operator

- The bitwise OR operator (|) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position which either *x* or *y* has a 1 bit (or if both do), as shown in the table on the right.

|	0	1
0	0	1
1	1	1

- The primary use of the | operator is to assemble a single integer value from other values, each of which contains a subset of the desired bits.
- In the context of sets, the | operator performs a union, as in the following calculation of **primes** ∪ **squares**:



The Exclusive OR Operator

- The exclusive OR or xOR operator (^) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which *x* and *y* have different bit values, as shown on the right.

^	0	1
0	0	1
1	1	0

- The xOR operator has many applications in programming, most of which are beyond the scope of this text.
- The following example flips all the bits in the rightmost three bytes of a word:

The Bitwise NOT Operator

- The bitwise NOT operator (~) takes a single operand *x* and returns a value that has a 1 wherever *x* has a 0, and vice versa.
- You can use the bitwise NOT operator to create a mask in which you mark the bits you want to eliminate as opposed to the ones you want to preserve.
- The ~ operator creates the **complement** of a set, as shown with the following diagram of ~ **primes**:

The Shift Operators

- Java defines three operators that have the effect of shifting the bits in a word by a given number of bit positions.
- The expression *x* << *n* shifts the bits in the integer *x* leftward *n* positions. Spaces appearing on the right are filled with 0s.
- The expression *x* >> *n* shifts the bits in the integer *x* rightward *n* positions. The question as to what bits are shifted in on the left depend on whether *x* is a signed or unsigned type:
  - If *x* is a signed type, the >> operator performs what computer scientists call an ***arithmetic shift*** in which the leading bit in the value of *x* never changes. Thus, if the first bit is a 1, the >> operator fills in 1s; if it is a 0, those spaces are filled with 0s.
  - If *x* is an unsigned type, the >> operator performs a ***logical shift*** in which missing digits are always filled with 0s.