Section Solutions 8: Function Pointers and Hashing

Problem 1: Function Pointer Quiz

The statement

```
FunctionOne(IntFunction());
```

calls the function `IntFunction`, and then calls `FunctionOne` with the integer returned by `IntFunction`. The parentheses after `IntFunction` signal that a call to the function `IntFunction` is being made.

On the other hand, the statement

```
FunctionTwo(IntFunction);
```

calls the function `FunctionTwo` with a pointer to the function `IntFunction`. `IntFunction` is not called by this statement, but is simply passed to `FunctionTwo` by means of its address. The prototypes for the two functions are

```
typedef int (*intfn)(void);

void FunctionOne(int i);
void FunctionTwo(intfn fn);
```

Problem 2: `void *` Practice

a) When the tree has no ordering we can't make any assumptions about where the maximum value will be found. We have to search both subtrees and compare their maximums to the current node's value and choose the largest of the three to propagate upwards. This algorithm must visit every node in the tree to find the absolute largest and is a O(n) algorithm.

```
void *MaxPtr(void *x, void *y, cmpfn cmp)
{
	if (x == NULL) return (y);
	if (y == NULL) return (x);
	if (cmp(x, y) < 0) {
		return (y);
	} else {
		return (x);
	}
}

void *MaxInTree(Node *t, cmpfn cmp)
{
	void *leftMax, *rightMax;
```

```
        if (t == NULL) return NULL;

        leftMax = MaxInTree(t->left, cmp);
        rightMax = MaxInTree(t->right, cmp);
        return (MaxPtr(t->value, MaxPtr(leftMax, rightMax, cmp), cmp));
}
```

b)

```
int StudentCompare(void *e1, void *e2)
{
        Student *s1, *s2;
        int cmp;

        s1 = (Student *) e1;
        s2 = (Student *) e2;
        cmp = StringCompare(s1->lastName, s2->lastName);
        if (cmp == 0) cmp = StringCompare(s1->firstName, s2->firstName);
        if (cmp == 0) cmp = s1->middleInitial - s2->middleInitial;
        return (cmp);
}
```

Now finding the alphabetically last student is a mere problem of calling our Max function using the correct function pointer argument. Easy!

```
s = (Student *) MaxInTree(studentTree, StudentCompare);
```

Problem 3 : Hashing performance under collision

Typically if two items hash into the same bucket they are chained into a unordered linked list. In the worst case, where all items hash to the same bucket, you have a list of length N to search and the Lookup function would take O(N) time. However, if the hash function is doing its job, you expect the items to be scattered evenly among the buckets. Each list will average N/B elements and Lookup will run in time O(N/B). If the number of buckets is chosen to be about the number of items, you achieve a O(1) Lookup for the average case.

Problem 4 : Hash function Analysis

a) The table has 2048 buckets and yet this hash function will only place things in 26 of those buckets. And even within those 26 buckets, we will tend to get clustering around the more popular choices for the first letter of peoples' names.

b) Like above, this hash function will only use part of the available buckets, because the product of three digits only produces values is the range 0 to 729. Further, clustering will occur around some values such as 0 and no primes greater than 7 will get used. Notice our range only has 1000 values and our table has 1000 buckets, why not just map each number to the bucket with that same index? Simpler and very efficient.

c) One important feature of a hash function is that it must be stable. A given value must hash to the same result each time. Using a random number generator in this hash function means a value can hash to something different each time it is hashed. This is

completely unacceptable for a hash function, because we will not be able to reliably find the bucket under which we previously stored an item!

Problem 5 : Symbol Table Implementation Choices

a) Either the hash table or the binary tree could support dumping the contents in any order.

b) Only the binary tree gives us easy access to dumping the contents in sorted order.

c) Again, only the binary tree has a notion of ordering, so it would be the better choice for finding the alphabetically last key.

d) The binary tree degenerates into a linked-list (its worst case performance) when you insert items in sorted order. The hash table has no such poor behavior on this case, and thus would make the better choice.

Problem 6: Mapping functions, Iterators, and Symbol Tables

(a) Using a mapping function:

```
string FindLongestKey(symtabADT table)
{
    string longest;

    longest = "";
    MapSymbolTable(ProcessKey, table, &longest);
    return (longest);
}

static void ProcessKey(string key, void *value, void *clientData)
{
    string *pLongest = clientData;

    if (StringLength(key) > StringLength(*pLongest))
      {
        *pLongest = key;
      }
}
```

Note that this approach requires the implementation of `FindLongestKey` to use the `clientData` pointer to keep track of the longest key encountered so far.

(b) Using an iterator:

```
string FindLongestKey(symtabADT table)
{
    string key, longest;
    iteratorADT iterator;

    longest = "";
    iterator = NewIterator(table);
    while (StepIterator(iterator, &key))
      {
        if (StringLength(key) > StringLength(longest))
          {
            longest = key;
          }
      }
    FreeIterator(iterator);
    return (longest);
}
```

(c) It's a matter of personal taste but most programmers would consider the iterator method more intuitive in this case. To use the mapping function, we had to use the client data in sort of a strange way by always changing this value that we're going to eventually return. With an iterator, there was no need to pass around void *'s! In general, mapping functions work best when you want to do the same thing to every element in the table (like increment every value or print every value). Iterators work best when you need to actually access each element (like for comparison). The difference is subtle but as you work with both iterators and mapping functions, you'll start to develop an intuition for which is appropriate.