CS302 Data Structures

Spring 2010 – Dr. George Bebis

Homework 4 - Solutions

1. (a) void MergeLists(SortedType, SortedType, SortedType&);

(b)
```
void MergeLists(SortedType list1, SortedType list2, SortedType& list)
{
ItemType item;
bool found;

list.MakeEmpty();

list1.ResetList();
while(!list1.IsLastItem()) {
   list1.GetNextItem(item);
   list.InsertItem(item);
}

list2.ResetList();
while (!list2.IsLastItem() && (list.LengthIs() < MAX_ITEMS-1)) {
   list2.GetNextItem(item);
   list.RetrieveItem(item, found);
   if(!found)
     list.InsertItem(item)
 }

 if((list.Length()== MAX_ITEMS-1) && (!list2.IsLastItem()))
    cout << “Not enough space to merge the lists” << endl;
}
```

(c) The client version of the array-based and linked are the same

(d) Suppose list1 contains N_1 elements and list2 contains N_2 elements. The first while loop is executed $O(N_1)$ times. The body of this loop takes $O(N_1)$ since GetNextItem takes O(1) and InsertItem takes $O(N_1)$. So, the complexity of the first loop is $O(N_1{}^x N_1)$. The second while loop is executed $O(N_2)$ times. The body of that loop takes $O(N_1+N_2)$ time since GetNextItem takes O(1), RetrieveItem takes $O(\log(N_1+N_2))$ time (i.e., using BinarySearch) and InsertItem takes $O(N_1+N_2)$ time. So, the complexity of the second loop is $O(N_2 \, x(N_1+N_2))$. Overall, the running time of MergeList is the max between the two loops: $O(\max(N_1 \times N_1, N_2 \times (N_1+N_2))$.

2. Exercise 12 (page 407)

(a) Doubly linked (b) Circular (c) List with header and trailer

3. Exercise 8 (pages 464)

(a) `-1` (b) `120` (c) `1`

4. Exercise12 (page 466)

(a)This answer is incorrect. The value 0 is returned; the recursive case is never reached. This solution gets half credit, because it correctly calculates the base case (even if it doesn't reach it).
(b)This solution correctly caluculates the sum of squares but gets no credit because it is not a *recursive* solution.
(c)This answer is correct and gets full credit.
(d)This answer is functionally equivalent to (c); it just avoids the last recursive all (to an empty list) by returning the sum of the last sequares as the base case. This answer runs into problems if the list is empty, but the specification states that the list is not empty. This answer gets full credit.
(e)This solution is incorrect. The general case does not correctly calculate the sum of the squares. Quarter credit is given for using the correct control structure and for getting the base case correct

5. Exercise14 (page 467)

(a) The parameter values should be positive real numbers.

(b) Recursive version.

```
#include <math.h>
float SqrRoot(float number, float approx, float tol)
{
   if fabs(approx*approx - number) <= tol
       return approx;
   else
       return SqrRoot(number, (approx*approx + number/(2*approx),
tol);
}
```

(c) Non-recursive version.

```
float SqrRoot(float number, float approx, float tol)
{
    while (fabs(approx*approx - number) > tol)
        approx = (approx*approx + number)/(approx + approx);
    return approx;
}
```

(d) Driver to test the recursive and iterative versions of function **SqrRoot**.

```
#include <iostream.h>
int SqrRoot(int number);
int main()
{
    int number;
    cout << "Input the number you wish the square root of." << endl;
          << "Input a negative number to quit."  << endl;
    cin  >> number;
    while (number >= 0)
    {
        cout  << "The square root of "  << number  << " is "
              << SqrRoot(number)  << endl;
        cout << "Input the number you wish the square root of." <<
endl;
             << "Input a negative number to quit."  << endl;
        cin  >> number;
    }
    return 0;
}
```