EECE 2560:
Fundamentals of Engineering Algorithms

Review (1)

Bag with Arrays

A Bag is an abstract data type (ADT) that contains items of the same type. Items in the bag can be unordered and possibly duplicated.

a) The following is a partial definition of the `ArrayBag` class (as part of the array-based implementation of the bag). Write the C++ code to implement the shown `add` and `clear` methods.

```
template<class ItemType>
class ArrayBag {
 private:
   static const int DEFAULT_BAG_SIZE = 6;
   ItemType items[DEFAULT_BAG_SIZE]; // array of bag items
   int itemCount;                    // current count of bag items
   int maxItems;                     // max capacity of the bag
public:
  ……
  bool add(const ItemType& newEntry);
  void clear(); // remove all items in the bag
  ……
};  // end Bag
```

Bag with Linked List

a) The following is the definition of `Node` class and a partial definition of the `LinkedBag` class (as part of the linked-based implementation of the bag). Write the C++ code to implement the shown `add` and `clear` methods (no need to implement any of the methods of the `Node` class).

```
template<class ItemType>
class Node
{
private:
   ItemType        item; // A data item
   Node<ItemType>* next; // Pointer to next node
public:
   Node();
   void setItem(const ItemType& anItem);
   void setNext(Node<ItemType>* nextNodePtr);
   ItemType getItem() const ;
   Node<ItemType>* getNext() const ;
}; // end Node
```

```
template<class ItemType>
class LinkedBag {
private:
   Node<ItemType>* headPtr; // Pointer to first node
   int itemCount;           // Current count of bag items
public:
   ……
   bool add(const ItemType& newEntry);
   void clear(); // remove all items in the bag
   ……
}; // end LinkedBag
```

Towers of Hanoi

In the Towers of Hanoi problem, you are given *n* disks and three poles: A, B, and C. The disks were of different sizes. Any disk can be placed only on top of disks larger than it. Initially, all the disks are on pole A. The problem is to move the disks, one by one, from pole A to pole B. You can use pole C during the transfer process.

The following pseudocode describes a recursive function that solves the problem:

```
solveTowers(count, source, destination, spare)
  if (count is 1)
          Move a disk directly from source to destination
  else
  {
          solveTowers(count - 1, source, spare, destination)
          solveTowers(1, source, destination, spare)
          solveTowers(count - 1, spare, destination, source)
  }
```

Use mathematical induction to prove the following:

a) The correctness of the above code in solving the problem.

b) If we start with N disks, then the number of moves the above code makes to solve the problem is $2^N - 1$.

Merge and Quick Sort

1. Explain, without mathematical proof, why the worst-case running time of the Merge Sort is O(n log n) while the worst-case running time of the Quick Sort is O(n^2) even though both algorithms have average-case running time of O(n log n).

2. Do the Quick Sort and/or the Merge Sort algorithms sort in place? Explain why

Stable Merge

The following is the code that implements the merge process as part of the Merge Sort algorithm implementation. What change is needed, if any, to make Merge Sort a stable sorting algorithm? Explain your answer.

```
// While both subarrays are not empty, copy the smaller item into the temporary array
   int index = first1;      // Next available location in tempArray
   while ((first1 <= last1) && (first2 <= last2))
{
   // At this point, tempArray[first..index-1] is in order
   if (theArray[first1] < theArray[first2]) {
      tempArray[index] = theArray[first1];
      first1++; }
   else {
      tempArray[index] = theArray[first2];
      first2++; }   // end if
   index++;
}  // end while
```

Quick Sort Partition

The following is a partial C++ implementation of the `Partition` function needed by the Quick Sort algorithm. Complete the function by writing the missing C++ in the shown `while` loop.

```
int Partition(int theArray[], int first, int last)
{
         // Choose pivot using median-of-three selection
      int medianIndex = sortFirstMiddleLast(theArray, first, last);
         //Reposition pivot so it is last in the array
          swap(theArray[medianIndex], theArray[last - 1]);
          int pivotIndex = last - 1;
          int pivot = theArray[pivotIndex];
         // Determine the regions S1 and S2
         int indexFromLeft = first + 1;
         int indexFromRight = last - 2;
         bool done = false;
         while (!done)
         {
                  ??????????????????????
         }  // end while

         // Place pivot in proper position between S1 and S2
         swap(theArray[pivotIndex], theArray[indexFromLeft]);
         pivotIndex = indexFromLeft;

         return pivotIndex;
}  // end partition
```

Running Time

For each of the following code, state the big *O* asymptotic notation of its running time as a function of *n*:

a)
```
m=1;
for (j=1; j<=n; j++)  m*=m;
for (k=1; k<=m; k++) cout<<k;
```

b)
```
m=1;
for (j=1; j<=n; j++) m*=2;
for (k=1; k<=m; k++) cout<<k;
```

c)
```
m=2;
for (j=1; j<=n; j++) m*=m;
for (k=1; k<=m; k++) cout<<k;
```

Recurrence Equations

Solve the following recurrence equations to find the big *O* asymptotic notation of the running time as a function of *n*:

a) T(n) = T(n/2) + c, and T(1) = c
   (assume that $n = 2^k$ for a positive integer *k*)

b) T(n) = T(n/2) + n, and T(1) = 1
   (assume that $n = 2^k$ for a positive integer *k*)