CSE 15
Introduction to Data Structures
Midterm 2 Review Problems

Note that we may not reach some of the problems in the group 10-14 before the exam. We'll see how far we get by end of business Monday.

1. Write C code for the following.

   a. A typedef statement that defines a struct called NodeObj suitable for building a singly linked list of integers. NodeObj will have a data field of type int.

   b. A typedef statement that defines a pointer type called Node that points to a NodeObj.

   c. A constructor function with prototype `Node newNode(int x)` that allocates a NodeObj from heap memory, initializes its data field to *x*, then returns a pointer to the allocated memory.

   d. A destructor function with prototype `void freeNode(Node* pN)` that, after checking that its argument is not NULL and does not point to NULL, frees all heap memory associated with its argument, then sets the reference to which its argument points to NULL.

2. Write a *recursive* C function with prototype `void freeAllNodes(Node H)` that, given a head reference to a linked list based on the types defined in the preceding problem, frees all heap memory associated with the Nodes in the list. (This function will *not* alter its argument by setting it to NULL).

3. Write a *recursive* C function with prototype `int product(Node H`) that, given a head reference to a linked list of integers based on the Node and NodeObj types defined in the preceding problem, returns the product of the items in the list. The product of an empty list is defined to be 1.

4. Write functions `void push(int x)` and `int pop()` for a C implementation of an integer stack based on the Node type defined in problem (1). Both functions assume the existence of a global Node reference variable, called `top`, pointing to a linked list data structure regarded as a stack. Function `push()` adds *x* to the top of the stack and deals correctly with the case of an empty stack. Function `pop()` returns and removes the top element, and has as precondition that the stack is not empty (`top!=NULL`). Neither function creates any memory leaks.

5. Write functions `void enqueue(int x)` and `int dequeue()` for a C implementation of an integer Queue based on the Node type defined in problem (1). Both functions assume the existence of global Node reference variables, called `front` and `back` respectively, that point to a linked list data structure regarded as a queue. Function `enqueue()` adds x to the back of the queue and deals correctly with the case of an empty queue. Function `dequeue()` returns and removes the front element, and has as precondition that the queue is not empty (`front!=NULL`). Neither function creates any memory leaks.

6. Write a the implementation (.c) file for an ADT in C called Triple. The states of this ADT consist of ordered triples of the form (a, b, c) where a, b and c are integers. The header file Triple.h below gives prototypes for ADT operations, constructor, destructor and other functions, along with their descriptions. Your file Triple.c should define (using typedef) a struct called TripleObj that encapsulates a single state of the ADT. The number, types and names of the fields in TripleObj are your design decision.

```
//-----------------------------------------------------------------------------
// Triple.h
//-----------------------------------------------------------------------------
#include<stdlib.h>
#include<stdio.h>

#ifndef _Triple_H_INCLUDE_
#define _Triple_H_INCLUDE_


// Exported type --------------------------------------------------------------
typedef struct TripleObj* Triple;

// Constructor & Destructor ---------------------------------------------------
Triple newTriple(int a, int b, int c);
void freeTriple(Triple* pT);

// ADT operations -------------------------------------------------------------
int getFirst(Triple T);          // Returns first member of Triple T
int getSecond(Triple T);         // Returns second member of T
int getThird(Triple T);          // Returns third member of T
int sumOfSquares(Triple T);      // Returns the sum of the squares of
                                 // the members of T
void incrementFirst(Triple T);   // Increases first member of T by 1
void setThird(Triple T, int x);  // Sets third member of T to be x
void rotate(Triple T);  // Sets first to second, second to first and
                        // third to first: (a, b, c) --> (b, c, a)

// Other operations -----------------------------------------------------------
Triple add(Triple T, Triple U); // Returns a new Triple whose members are
                                // the sums of the corresponding members
                                // of T and U, i.e. (a, b, c) + (d, e, f)
                                // --> (a+d, b+e, c+f)
void printTriple(FILE* out, Triple T); // Prints the string "(a, b, c)\n"
                                       // to the file pointed to by out,
                                       // where a, b and c are the members
                                       // of the triple T, in order
int equals(Triple T, Triple U); // Returns true (1) if T has same members
                                // as U, in the same order, and returns
                                // false (0) otherwise

#endif
//-----------------------------------------------------------------------------
```

7. The following C program includes a global variable called `time`. Since it is declared outside of all functions (on line 3), its scope is the entire file. Notice that `time` is incremented before each of the functions `a`, `b`, and `c` return. Show the state of the function call stack when `time=4` (i.e. at the instant `time` becomes equal to `4`). Each stack frame should show the values of all function arguments and local variables, as well as the line to which execution will transfer when the function returns. If a local variable has not yet been assigned a value at `time=4`, indicate that fact by stating its value as `undef`. Assume that the terms in the right hand side of line 19 are evaluated from left to right, i.e. first call `a()` then call `b()`. Also determine the program output, and print it on the line below exactly as it would appear on the screen.

```
1.)  #include<stdio.h>
2.)  #include<stdlib.h>
3.)  int time;
4.)  int a(int x){
5.)     int i;
6.)     i = x*x;
7.)     time++;
8.)     return(i);
9.)  }
10.) int b(int y){
11.)    int j;
12.)    j = y+a(y);
13.)    time++;
14.)    return(j);
15.) }
17.) int c(int z){
18.)    int k;
19.)    k = a(z)+b(z); // first call a() then b()
20.)    time++;
21.)    return(k);
22.) }
23.) int main(void){
24.)    int q, r;
25.)    time = 0;
26.)    q = b(5);
27.)    r = c(2);
28.)    printf("q=%d, r=%d, time=%d\n", q, r, time);
29.)    return(EXIT_SUCCESS);
30.) }
```

Output:

____

State of the function call stack when time=4:

8. Consider the C function below called `wasteTime()`. Your goal is to determine how much time `wastTime()` wastes. The basic (barometer) operation in this algorithm is to waste 1 unit of some unspecified time unit. Determine the total amount of time $T(n)$ wasted as a function of the input n. Also find the asymptotic runtime, i.e. $T(n) = \Theta(\text{some simple function of } n)$.

```
void wasteTime(int n){
   int i, j, k;
   waste 2 units of time;
   for(i=0; i<n; i++){
      waste 5 units of time;
      for(j=0; j<n j++){
         waste 12 units of time;
         for(k=0; k<n; k++){
            waste 3 units of time;
         }
      }
   }
}
```

9. Consider the C function below called `wasteMoreTime()` that again just wastes time. Let $T(n)$ be the number of units of time wasted by the function call `wasteMoreTime(n)`.

```
void wasteMoreTime(int n){
    int i, j;
    for(i=0; i<n; i++){
       for(j=0; j<=i; j++){
          waste 1 unit of time;
       }
    }
}
```

   a. Determine the function $T(n)$.

   b. Change the loop repetition condition in the inner for loop from `j<=i` to `j<i`, and answer the same question.

10. Rewrite the sorting algorithms `BubbleSort()`, `SelectionSort()` and `InsertionSort()` (found on the class webpage in Examples/Lecture/C_Programs/SortingSearching/Sort.c) so that they sort in decreasing instead of increasing order.

11. Adapt the same sorting algorithms from problem 10 so that they operate on arrays of strings (i.e. null `'\0'` terminated char arrays) instead of ints.

12. Write a C function with heading `int countComparisons(int* A, int n, int i)` that takes as input an int array A of length n, and an int i in the range $0 \leq i \leq (n-1)$ specifying an index to A. The function will return an int giving the number of elements in A that are strictly less than $A[i]$.

13. Write another function `int* countAllComparisons(int* A, int n)`, similar to the one in the previous problem, except now you are to count the number of elements in A that are strictly less than $A[i]$, for *all* i in the range $0 \leq i \leq (n-1)$. Instead of returning a single count, your function will return (a pointer to) an int array of length n (allocated from heap memory) storing these counts. Thus if C is the returned array, then $C[i]$ will equal the number of numbers in A that are strictly less than $A[i]$.

14. Observe that if array A is sorted, and contains distinct elements, then the number of numbers in A that are strictly less than $A[i]$, is exactly i. (Verify this with examples if it is not clear.) Write a function with prototype `void comparisonSort(int* A, int n)`, that takes as input an int array A of length n containing distinct integers, then sorts the elements of A by performing the following steps.

    (1) Allocate a new array of length n from heap memory called B.
    (2) Allocate a new array of length n from heap memory called C whose element $C[i]$ is the number of numbers in A that are strictly less than $A[i]$. Do this by calling the function you wrote in the previous problem.
    (3) Place the number $A[i]$ in array B at index $C[i]$, for all i in the range $0 \leq i \leq n-1$.
    (4) Copy the elements of B into A, in order.
    (5) Free arrays B and C.

    As a final step, figure out how to alter comparisonSort() so that it works on arrays of *non*-distinct elements.