CSE 374 Final Exam Sample Solution 6/10/10

Question 1. (8 points) (testing) Suppose we are testing a list data structure implementation similar to a Java `ArrayList`. The list uses a fixed-size *array* as the underlying data structure to store the list elements. The available operations include `initialize()`, to create an empty list; `add(x)`, which adds `x` to the end of the list; `get(index)`, which returns the item at a given position; `set(index, x)`, which replaces the item at a given position; `size()`, which returns the number of items currently in the list; `capacity()`, which returns the maximum size the list can hold; `contains(x)`, which reports whether `x` appears in the list; and `remove(index)`, which removes the item at the given position.

(a) Describe three different **black-box** tests that you could use to test this data structure. The tests must check different things – you can't just describe the same test three times with different data.

A good test describes the inputs, operations to be done, and expected results to be verified. It's also good to have tests that only check a few things at a time, so it is easier to tell what's wrong if a test fails. Here are several black-box tests:

- **Initialize a new list and verify that its size() is 0.**
- **Initialize a new list and verify that contains(x) is false for some item x.**
- **Add an item to a list and verify that size() increases by 1.**
- **Add an item x and verify that contains(x) is true.**
- **Add an item x to the end of the list and verify that get(size()-1) returns x.**
- **Add some items to a list and verify that contains(y) is false for some item not added to the list.**
- **Add several items to the list and verify that get(k) returns the correct values.**
- **Use set(k,x) to change an existing value; verify that size() is unchanged, that get(k) returns x, and that get(i) returns its previous values for i≠k.**
- **Add items until the capacity() of the list is reached; verify that the items are present and size()==capacity().**
- **Etc. etc. etc.**

(b) Describe three different **white-box** tests you could use to test this data structure. As before, the different tests must check different things.

White-box tests use knowledge of internal implementation details. Examples:

- **Initialize() a list and verify that the internal size variable is 0.**
- **Add three items to the list; verify that they are stored in locations 0, 1, and 2 of the underlying array and that the internal size variable is 3.**
- **Add 5 items to the list; execute set(3,x); verify that the array contents are as expected and that the size did not change.**
- **If delete() is supported, delete an item in the middle of the list and verify that the items in the array are shifted to close up the gap.**
- **If the implementation is garbage-collected, as is true of Java, verify that unused slots in the array are null to avoid spurious pointers to objects that might otherwise be recycled by the garbage collector.**

Question 2. (6 points) (test coverage) Two of the ways we can measure the thoroughness of a set of tests are *statement coverage* and *path coverage*. How do these differ?

Path coverage is the percentage of all possible execution paths through the code that are exercised by the tests. Statement coverage is the percentage of all statements in the code that are executed. Even if statement coverage is 100%, path coverage of the same set of tests may be much less because only some of the possible paths were executed to reach all statements.

Question 3. (6 points) (debugging) You are trying to fix a C program that contains a single function `main`. During execution the program gets stuck in an infinite loop somewhere. From looking at the source code we can see that it contains 4 simple loops.

Describe how you would use a debugger like `gdb` to figure out where in the program the problem lies. You may not modify the source code (i.e., you can't insert print statements or anything like that – you have to work with the source code as it exists).

- **Be sure the code is compiled and linked with `-g` to ensure that debugging information is present; recompile if needed.**
- **Open the program with gdb and set breakpoints in the code at the beginning of each loop. Run the program and continue when each breakpoint is encountered. We can tell which loop has the problem once it starts executing and we don't regain control in the debugger.**
- **Interrupt the program or kill it and restart; this time stopping each time around the infinite loop or otherwise using the debugger commands to diagnose the trouble.**

Question 4. (8 points) (a little preprocessor mischief) Suppose we have the following three C header and source files:

File `x.h`:

```
#ifndef X_H
#define X_H

#include "other.h"
#define XYZZY 42

#endif
```

File `other.h`:

```
#ifndef OTHER_H
#define OTHER_H

#define XYZZY 17

#endif
```

File `test.c`:

```
#include <stdio.h>
#include "x.h"
#include "other.h"

#define F(a,b) a*b

int main() {
   int a, b, c, d;
   a = 2;
   b = 3;
   c = 5;
   d = F(a+b,c);
   printf("xyzzy is %d\n", XYZZY);
   printf("d is %d\n", d);
   return 0;
}
```

What output is produced when we compile and execute `test.c`? (Assume that all three files are in the same directory. This program compiles and executes successfully.)

```
xyzzy is 42
d is 17
```

Question 5. (12 points) (strings and memory) The following program is supposed to read lines from `stdin`, store them in an array, then print the stored strings and delete them. It only needs to work as long as the input lines are under 500 characters. But even if the input lines are not too long, it produces the wrong output and tends to crash.

Without altering the basic organization of program and the specification of the functions in it, identify the problems and fix them. Write your corrections on the code below

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LINES 100
#define MAX_LINE_LENGTH 500
char **text_lines;  /* input stored in text_lines[0..nlines-1] */
int    nlines;

/* initialize data structure */
void init_lines() {
   text_lines = (char**)malloc(MAX_LINES*sizeof(char*));
   nlines = 0;
}

/* store line in next entry in text_lines if room, else ignore */
void store_line(char * line) {
   if (nlines >= MAX_LINES) return;
   ~~text_lines[nlines] = line;~~     // bug – copies pointers not strings
   text_lines[nlines] = (char *)malloc((strlen(line)+1)*sizeof(char));
   strcpy(text_lines[nlines], line);
   nlines++;
}

/* delete data structure */
void delete_lines() {
   int k;
   ~~free(text_lines);~~   // bug – don't free until individual lines freed
   for (k = 0; k < ~~MAX_LINES~~ nlines; k++) { // bug – frees uninit. ptrs
      free(text_lines[k]);
   }
   free(text_lines);
}
/* read stdin and store lines, print them, then delete */
int main() {
   char input_line[MAX_LINE_LENGTH];
   int k;

   init_lines();
   while (fgets(input_line, MAX_LINE_LENGTH, stdin)) {
      store_line(input_line);
   }
   for (k = 0; k < nlines; k++) {
      printf("%s", text_lines[k]);
   }
   delete_lines();
   return 0;
}
```

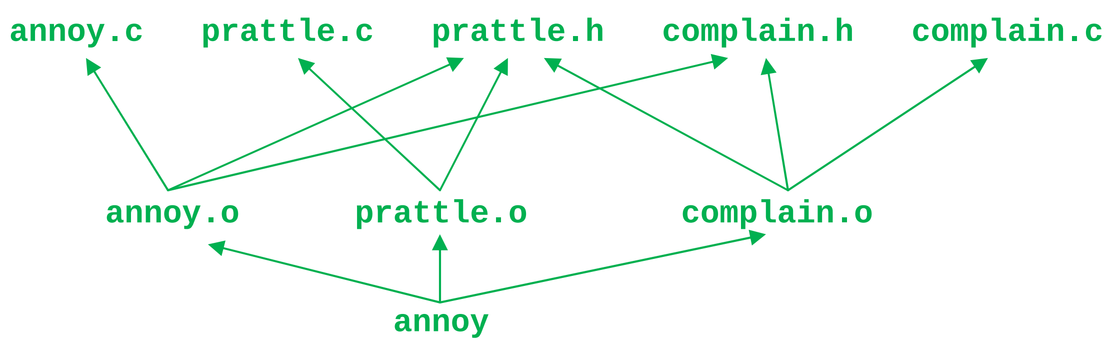
Question 6. (15 points) (`make` and `svn`) Suppose we have the following `Makefile` in the current directory.

(answers for part (c) added here)

```
prattle.o:  prattle.h  prattle.c  irritate.h

   gcc -Wall -g -c prattle.c

complain.o:  complain.h  complain.c  prattle.h

   gcc -Wall -g -c complain.c

annoy: annoy.o  prattle.o  complain.o  irritate.o

   gcc –Wall –g –o annoy annoy.o prattle.o complain.o irritate.o

annoy.o:  annoy.c  prattle.h  complain.h  irritate.h

   gcc -Wall -g -c annoy.c

irritate.o:  irritate.c  irritate.h

   gcc –Wall –g –c irritate.c
```

$$\begin{tikzcd}
\texttt{fie} \\
\texttt{fum} \arrow[u] \\
\texttt{foo} \arrow[u]
\end{tikzcd}$$

(a) In the space below, draw a dependency graph (diagram) showing the dependencies between the files as specified by the above `Makefile`. Your drawing should have an arrow from each file to the files that it depends on, as in the diagram to the left, where `foo` depends on `fum`, which depends in turn on `fie`.



Note: Remember that even if a `.c` file `#includes` a `.h` file, the `.c` file does not depend on it in the `Makefile`. It's the `.o` file that becomes out of date and needs to be recompiled if the `.h` file is changed.

(continued next page)

Question 6. (cont.) (b) If we just type "`make`" in the directory containing this `Makefile`, what happens? Assume that all of the source files (`.c` and `.h`) are present in the directory along with the `Makefile`, but that none of the other files (`.o`, etc.) named in the `Makefile` are present when we run `make`.

`prattle.c` is compiled to generate `prattle.o`.

(Remember that the first rule in the `Makefile` is the default target.)

Now suppose we've been working on this program and want to add a new source file `irritate.c` along with an associated header file `irritate.h`. File `irritate.c` only includes the `irritate.h` header; it does not use any of the others. Files `annoy.c` and `prattle.c` have been modified to use the new functions defined in these new files, with the appropriate `#includes` and calls to the new functions.

(c) On the previous page, write in the modifications needed to the `Makefile` so these new files are included in the program and are correctly compiled (and recompiled) as needed when the program is built.

(See previous page)

(d) Once the new `irritate.h` and `irritate.c` files have been tested and the `Makefile` updated in our working copy of the program, we need to be sure to update the `svn` repository. Write the necessary `svn` commands below to update the repository to reflect these additions and changes.

```
svn add irritate.h irritate.c
svn commit –m "we're done with this irritating question"
```

(It's good practice to run `svn update` first and clean up any conflicts discovered, but it's not strictly necessary if no conflicting changes are present.

Question 7. (12 points) (memory management) One of the things that can go wrong with the memory manager is if the length (size) field in a block on the free list has a wrong value and is so big that the address of a free block plus its size gives an address that is is greater than the address of the next block on the free list. For this problem, complete the following function so that it returns 1 (true) if a free list block overlaps its successor and 0 (false) if overlap is not detected. You should assume that the following `struct` defines the layout of header part of each free list node.

```
struct free_node {
   int size;              /* number of bytes in this node, */
                          /*   including this header       */
   struct free_node *next; /* next node on the free list,  */
                          /*   or NULL if no more nodes.   */
};
```

Complete the following function to check for overlapping free-list blocks.

```
/* Return 1 (true) if a node on free_list overlaps its      */
/* successor (block address+size > successor address).     */
/* Return 0 if no overlap is detected.                      */
/* pre: assume freelist blocks have increasing addresses.   */

int overlap_found(struct free_node * free_list) {
   struct free_node *p = free_list;
   while (p != NULL && p->next != NULL) {
      if ((int)p + p->size > (int)p->next) {
         return 1;
      }
      p = p->next;
   }
   return 0;
}
```

Note that the condition in the `while` loop depends on the fact that the right operand of && is not evaluated if the left operand is false. If the operands were reversed or if both operands were evaluated always, the `p->next` reference would likely produce a segfault at the end of the list.

Question 8. (13 points) (another trieing problem) Once the dictionary has been read by the text messaging program, all of the words and their synonyms are stored in the trie. For this problem, write a function that returns the length of the longest word (or words) in the trie – i.e., the longest string of digits that makes up a word. This is basically the height of the trie, except that links in the trie for synonyms (‘#’ links) should be ignored.

Assume that a trie node is defined as follows:

```
struct tnode {                // one node in the trie:
   char * word;               //  C-string if this node has a
                              //  word attached, otherwise NULL
   struct tnode *next[10];    //  Subtrees.  next[2] to next[9]
                              //  are subtrees for digits 2-9;
                              //  next[0] is the synonym (‘#’)
};                            //  link; next[i]==NULL if no child
```

(Note that you may not need to use all of the information attached to a node.)

Complete the following C function so it returns the length of the longest word (sequence of digits) in the trie with root r.

```
/* Return the length of the longest word in the trie r. */
int max_length(struct tnode * r) {
`  int k, len, max_len;
   if (r == NULL) {
      return 0;
   }
   max_len = 0;
   for (k = 2; k <= 9; k++) {
      if (r->next[k] == NULL) {
         len = 0;
      } else {
         len = 1 + max_length(r->next[k]);
      }
      if (len > max_len) {
         max_len = len;
      }
   }
   return max_len;
}
```

Question 9. (12 points) The ~~dreaded~~ ~~traditional~~ annoying C++ “what does this print” question. What output is produced when this program is executed? It does compile and execute with no warnings or errors using g++.

```
#include <iostream>
using namespace std;

class One {
 public:
   virtual void p() { q(); cout << "One::p" << endl; }
           void q() { r(); cout << "One::q" << endl; }
           void r() {      cout << "One::r" << endl; }
};

class Two: public One {
 public:
           void p() { q(); cout << "Two::p" << endl; }
           void r() {      cout << "Two::r" << endl; }
};

int main() {
   Two * creature = new Two();
   creature->q();
   cout << "----" << endl;
   creature->r();
   cout << "----" << endl;

   One * thing = creature;
   thing->p();
   cout << "----" << endl;
   thing->r();

   return 0;
}
```

Output:

```
   One::r
   One::q
   ----
   Two::r
   ----
   One::r
   One::q
   Two::p
   ----
   One::r
```

Question 10. (8 points) (concurrency) (a) Why would anyone want to write a multi-threaded program?

A primary reason is performance. For instance, if one thread is blocked for I/O, other threads can continue. In a multiprocessor or multi-core system it may be possible to perform more than one computation in parallel and shorten the overall time needed to finish the job.

Another reason is program structure. It may be clearer to structure a program as a set of cooperating threads rather than as a single thread that has complex code in it to keep track of multiple things that are happening at the same time.

(b) Give an example of a problem that can occur in a multi-threaded program that cannot occur in a program with a single thread.

The main problem is if two or more threads have access to shared data and at least one of the threads updates the data. Once that happens there's potential trouble if references to the shared data are not properly synchronized and one thread can see an incorrect or partially updated value of data being changed by another.