Final Exam
Dec 19, 2013

COMS W3136 Essential Data Structures in C/C++
Columbia University
Fall 2013

Instructor: Jae Woo Lee

About this exam:

- There are 3 problems totaling 100 points:

    Problem 1: 24 points
    Problem 2: 40 points
    Problem 3: 36 points

- Assume the following programming environment:

    Platform: Ubuntu Linux 12.04, 64-bit version
    Primitive type sizes: sizeof(int) is 4 and sizeof(int *) is 8

    All library function calls and system calls are successful. For example, you can assume malloc() does not return NULL.

    If this exam refers to certain algorithms or functions without providing code, assume the code provided in class through lecture notes, sample code, skeleton code and solutions.

What to hand in and what to keep:

- At the end of the exam, you will hand in only the answer sheet, which is the last sheet of this exam booklet. The answer sheet is a 2-pager printed double-sided.

- Make sure you write your name & UNI on the answer sheet.

- All other pages (i.e., the rest of this exam booklet and any blue book you used during the exam) are yours to keep.

- Before you hand in your answer sheet, please copy down your answers back onto the exam booklet so that you can verify your grade when the solution is published in the mailing list.

Good luck!

Problem [1]: 24 points

Consider the following CLIC lab session:

```
    ~/cs3136$ git clone /home/jae/cs3136-pub/lab1 lab1
    Initialized empty Git repository in ~/cs3136/lab1/.git/

    ~/cs3136$ cd lab1

    ~/cs3136/lab1$ ls
    main.c  Makefile  myadd.c  myadd.h  README.txt

    ~/cs3136/lab1$ git status
    # On branch master
    nothing to commit (working directory clean)

    ~/cs3136/lab1$ vim  Makefile  myadd.h  multiply.c

    ~/cs3136/lab1$ make
    gcc -g -Wall    -c -o main.o main.c
    gcc -g -Wall    -c -o myadd.o myadd.c
    gcc -g  main.o myadd.o   -o main

    ~/cs3136/lab1$ make
    make: ‘main’ is up to date.

    ~/cs3136/lab1$ touch myadd.h

    ~/cs3136/lab1$ make
    gcc -g -Wall    -c -o main.o main.c
    gcc -g -Wall    -c -o myadd.o myadd.c
    gcc -g  main.o myadd.o   -o main

    ~/cs3136/lab1$ ls
    main    main.o    multiply.c  myadd.h  README.txt
    main.c  Makefile  myadd.c     myadd.o

    ~/cs3136/lab1$ git add myadd.h

    ~/cs3136/lab1$
```

Note that the user issued "vim Makefile myadd.h multiply.c" command to launch his editor, modified some comments in Makefile and myadd.h, and created a new file, multiply.c, to start working on a multiplication routine.

Problem [1] continued

(a) 18 points

At the end of the shell session shown above, there are 9 files in the current directory:

```
~/cs3136/lab1$ ls
main    main.o    multiply.c  myadd.h  README.txt
main.c  Makefile  myadd.c     myadd.o
```

Put each file into one of the following 4 categories from the point of view of the git revision control system.

Untracked:
Tracked, unmodified:
Tracked, modified, but unstaged:
Tracked, modified, and staged:

(b) 6 points

The Makefile in that directory is shown here, with the build rules removed. Fill in the missing lines, specifying targets and dependencies that are consistent with the CLIC lab session shown above.

Fill in only targets and dependencies. That is, do not write the gcc commands that come below the target/dependency lines. Rely on the implicit rules that make knows by default.

```
CC  = gcc
CXX = g++
INCLUDES =
CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)
LDFLAGS = -g
LDLIBS =

# Fill in the missing targets and dependencies here

.PHONY: clean
clean:
        rm -f *.o *~ a.out core main
```

Problem [2]: 40 points (10 parts, 4 points each)

Recall the Big-O notation for classifying computer algorithms according to the time it takes to run them as a function of the size of the data on which the algorithms operates.

For example, O(N^2) algorithms run in time proportional to the square of the input size. O(1) algorithms run in constant time, independent of the input size.

For each of the algorithms or operations below, choose one of the following:

O(1)

O(log N)

O(N)

O(N log N)

O(N ^ 2)

(2.1) Merge sort

(2.2) Selection sort

(2.3) prepend() function in singly linked list

```
/*
 * Create a node that holds the given data x,
 * add the node to the front of the given list,
 * and returns the resulting list.
 *
 * Returns NULL if malloc for the node fails.
 */
struct Node *prepend(struct Node *list, double x)
```

(2.4) remove_front() function in singly linked list

```
/*
 * Removes the first node (deallocating the memory for the node)
 * and returns the rest of the list.
 *
 * Returns NULL if list is NULL.
 */
struct Node *remove_front(struct Node *list)
```

(2.5) find() function in singly linked list

```
/*
 * Returns the first node containing x in the given list.
 * Returns NULL if no node contains x.
 */
struct Node *find(struct Node *list, double x)
```

Problem [2]: continued

For the following algorithms and operations, assume that binary search trees (BSTs) are well-balanced and that graphs are sparse.

(2.6) lookup() function in BST

```
/*
 * Returns the node containing x.
 * Returns NULL if x is not in T.
 */
BST *lookup(int x, BST *T)
```

(2.7) insert() function in BST

```
/*
 * Insert x into the given binary search tree T.
 * If T is not NULL, T is again returned after inserting x.
 * If T is NULL, a new single-node tree is returned.
 */
BST *insert(int x, BST *T)
```

(2.8) height() function in BST

```
/*
 * Returns the given node's height, which is defined as the
 * length of a longest path from the node to a leaf.
 * (All leaves have height 0.)
 */
int height(BST *T)
```

(2.9) Breadth-first search (BFS) in a graph with N vertices

(2.10) push_back() in the vector container class in C++ standard library, considering the average time taken by one push_back() call when we make many push_back() calls.

Problem [3] 36 points

Consider bst.cpp:

```
// #include statements omitted

using namespace std;

struct BST {
    int data;
    BST *left;
    BST *right;

    BST(int x) {
        data = x;
        left = NULL;
        right = NULL;
    }
};

/*
 * Insert x into the given binary search tree T.
 * If T is not NULL, T is again returned after inserting x.
 * If T is NULL, a new single-node tree is returned.
 */
BST *insert(int x, BST *T)
{
    // code is the same as bst-1.cpp in the lecture note
}

/*
 * Deallocate all nodes in the given tree.
 */
void remove_all_nodes(BST *T)
{
    // code is the same as bst-1.cpp in the lecture note
}

/*
 * Draw the BST rotated 90-degree counter-clockwise.
 */
void draw(BST *T, int depth = 0, const char *edge = "--")
{
    // code is the same as bst-1.cpp in the lecture note
}
```

Problem [3] continued

```
/*
 * In-order traversal.
 */
void traverse_inorder(BST *T, void (*f)(BST *))
{
    // (3.1) Implement traverse_inorder() in C/C++.
}

/*
 * Level-order traversal.
 */
void traverse_levelorder(BST *T, void (*f)(BST *))
{
    // (3.2) Briefly describe the algorithm for this function in English

    // (3.3) Implement traverse_levelorder() in C/C++
}

void print_node(BST *node)
{
    cout << node->data << " ";
}

int main()
{
    srandom(time(NULL));

    // read the number of elements from the user.
    int n;
    cin >> n;

    // Initially empty tree.
    BST *T = NULL;

    for (int i = 0; i < n; i++) {
        T = insert(random() % 100, T);
    }

    cout << "\nThe BST:" << endl;
    draw(T);

    cout << "\nAll nodes in-order:" << endl;
    traverse_inorder(T, print_node);
    cout << endl;

    cout << "\nAll nodes level-order:" << endl;
    traverse_levelorder(T, print_node);
    cout << endl;

    remove_all_nodes(T);
    return 0;
}
```

Problem [3] continued

The bst.cpp program takes as a user input the number of random integers to put into a binary search tree (BST), draw the BST rotated 90-degree counter-clockwise, and prints the results of in-order traversal and level-order traversal. Here is an example run:

```
$ ./a.out
10

The BST:
      /- 92
                        /- 66
               /- 55
         \- 50
               \- 46
-- 45
         /- 40
      \- 24
               /- 17
         \- 5

All nodes in-order:
5 17 24 40 45 46 50 55 66 92

All nodes level-order:
45 24 92 5 40 50 17 46 55 66
```

A level-order traversal of a BST visits the nodes in increasing depth. That is, starting from the root node, all the child nodes get visited from left to right, and then all the grandchild nodes get visited from left to right, and so on. The example run above should make this clear.

(3.1) 10 points

Implement traverse_inorder() in C/C++.

(3.2) 10 points

Describe in English how you would go about implementing traverse_levelorder(). Please be succinct. Write no more than 2-3 sentences. All I'm looking for is if you got the key concept of the algorithm. [The sub-problem (3.2) is basically a way for you to get partial credit if you are unable to do (3.3).]

(3.3) 16 points

Implement traverse_levelorder() in C/C++.

COMS W3136, Fall 2013: Final Exam Answer Sheet, page 1 of 2

Name: _____

UNI: _____

[1] (a)

Untracked:

Tracked, unmodified:

Tracked, modified, but unstaged:

Tracked, modified, and staged:

(b) # Fill in the missing targets and dependencies

---

[2]

(2.1) Merge sort _____

(2.2) Selection sort _____

(2.3) prepend() _____

(2.4) remove_front() _____

(2.5) find() in linked list _____

(2.6) lookup() in BST _____

(2.7) insert() in BST _____

(2.8) height() in BST _____

(2.9) Breadth-first search (BFS) _____

(2.10) push_back() in the vector _____

(3.1) Implement traverse_inorder() in C/C++.

(3.2) How would you go about writing traverse_levelorder()?

(3.3) Implement traverse_levelorder() in C/C++.