**MSCF**

**Financial Computing I**

**Homework 2, for Lectures 4, 5, and 6**

***Due At 12:30 pm Tuesday, Sept. 11, Before the Quiz Session***

***You will lose 1 point every 10 minutes after that time***

1. **Linked Lists (35 points)**
2. Create a new, empty project named **hw2.1**. ***Either*** add the existing item **hw2.1.cpp** available online, ***or*** add a new, empty source file named **hw2.1.cpp**, and copy-and-paste the code below, from our lecture notes, into **hw2.1.cpp**. Compile and test.

**// File: hw2.1.cpp**

**// Author(s):**

**#include <iostream>**

**#include <iomanip>**

**using namespace std;**

**// clist structure definition**

**struct cl\_node {**

**char data;**

**cl\_node \*next;**

**};**

**// clist creation functions**

**cl\_node \*mk\_clist\_from\_Cstring(const char \*s)**

**{**

**cl\_node \*first(nullptr);**

**cl\_node \*last(nullptr);**

**for ( ; \*s; ++s) {**

**cl\_node \*pnew = new cl\_node{\*s, nullptr};**

**if (!last) {**

**last = first = pnew;**

**}**

**else {**

**last = last->next = pnew;**

**}**

**}**

**return first;**

**}**

**// clist access functions**

**size\_t clist\_size(const cl\_node \*cl)**

**{**

**size\_t size(0);**

**for ( ; cl; cl = cl->next)**

**size += 1;**

**return size;**

**}**

**void clist\_display\_to\_cout(const cl\_node \*cl)**

**{**

**for ( ; cl; cl = cl->next)**

**cout << cl->data;**

**}**

**// clist element modification functions**

**void clist\_to\_uppercase(cl\_node \*cl)**

**{**

**// assumes ASCII character set**

**for ( ; cl; cl = cl->next) {**

**if ('a' <= cl->data && cl->data <= 'z') {**

**cl->data -= 'a' - 'A';**

**}**

**}**

**}**

**// clist deletion function (a shape modifying function)**

**void clist\_delete(cl\_node \*\*p\_cl)**

**{**

**while (\*p\_cl) {**

**cl\_node \*tmp = \*p\_cl;**

**\*p\_cl = (\*p\_cl)->next;**

**delete tmp;**

**}**

**}**

**int main()**

**{**

**cl\_node \*bob = mk\_clist\_from\_Cstring("Bob");**

**clist\_display\_to\_cout(bob); // Bob**

**cout << '\n';**

**cout << "list is " << clist\_size(bob) << " chars long\n";**

**clist\_to\_uppercase(bob);**

**clist\_display\_to\_cout(bob); // BOB**

**cout << '\n';**

**clist\_delete(&bob);**

**clist\_display\_to\_cout(bob); //** *(empty -- no memory leak)*

**}**

1. Define a function, **mk\_copy\_clist**, that takes a *pointer-to-***cl\_node** as an argument, creates a copy of the pointed-to list, and returns a pointer to the first **cl\_node** of the copy. ***HINT: DRAW A PICTURE!!!*** (You don’t need to turn in the picture.) Add this code to **main** (either uncomment the code if you used the online **hw2.1.cpp**, or copy-and-paste this code):

**cl\_node \*hello = mk\_clist\_from\_Cstring("hello");**

**clist\_display\_to\_cout(hello); cout << '\n'; // hello**

**cl\_node \*hello2 = mk\_copy\_clist(hello);**

**clist\_display\_to\_cout(hello2); cout << '\n'; // hello**

**clist\_to\_uppercase(hello2);**

**clist\_display\_to\_cout(hello); cout << '\n'; // hello**

**clist\_display\_to\_cout(hello2); cout << '\n'; // HELLO**

**clist\_delete(&hello);**

**clist\_delete(&hello2);**

**clist\_display\_to\_cout(hello); //** *(empty)*

**clist\_display\_to\_cout(hello2); //** *(empty)*

Compile and test.

1. Define a function, **mk\_clist\_from\_string**, that takes a **string** as an argument, constructs a cl\_**node** list from the elements in the **string**, and returns a pointer to the first cl\_**node**. ***PICTURE!*** Add this code to **main**:

**string s("howdy");**

**cl\_node \*howdy = mk\_clist\_from\_string(s);**

**clist\_display\_to\_cout(howdy); cout << '\n'; // howdy**

**clist\_delete(&howdy);**

**clist\_display\_to\_cout(howdy); //** *(empty)*

Compile and test.

1. Define a function, **clist\_append\_char**, that appends a single **char** to the end of a clist. ***PICTURE!*** Add this code to **main**:

**cl\_node \*cl = mk\_clist\_from\_Cstring("Bell");**

**clist\_display\_to\_cout(cl); cout << '\n'; // Bell**

**clist\_append\_char(&cl, 'a');**

**clist\_display\_to\_cout(cl); cout << '\n'; // Bella**

**clist\_delete(&cl);**

**clist\_display\_to\_cout(cl); //** *(empty)*

**clist\_append\_char(&cl, 'X');**

**clist\_display\_to\_cout(cl); cout << '\n'; // X**

**clist\_delete(&cl);**

**clist\_display\_to\_cout(cl); //** *(empty)*

Compile and test.

1. Define a function, **clist\_count\_char**, that returns the number of occurrences of a **char** in a clist. Add this code to **main**:

**cl\_node \*cl2 = mk\_clist\_from\_Cstring("Mississippi");**

**clist\_display\_to\_cout(cl2); cout << '\n'; // Mississippi**

**cout << clist\_count\_char(cl2, 'i') << '\n'; // 4**

Compile and test.

1. Define a function, **clist\_delete\_dups**, that deletes duplicate characters from a clist. Add this code to **main**:

**clist\_display\_to\_cout(cl2); cout << '\n'; // Mississippi**

**clist\_delete\_dups(&cl2);**

**clist\_display\_to\_cout(cl2); cout << '\n'; // Misisipi**

**clist\_delete(&cl2);**

**clist\_display\_to\_cout(cl2); cout << '\n'; //** *(empty)*

Compile and test.

1. Define a function, **clist\_insert\_clist**, that takes a pointer to a clist (*pointer-to-pointer-to-***cl\_node**), a clist (*pointer-to-***cl\_node**), and a position (a **size\_t**) as arguments, and that inserts a *copy* of the second clist into the first at the specified location. The start of the clist is location **0**. Add this code to **main**:

**cl\_node \*cl3 = mk\_clist\_from\_Cstring("national");**

**cl\_node \*cl4 = mk\_clist\_from\_Cstring("ive fic");**

**clist\_insert\_clist(&cl3, cl4, 2);**

**clist\_display\_to\_cout(cl3); cout << '\n'; // naive fictional**

**clist\_display\_to\_cout(cl4); cout << '\n'; // ive fic**

**clist\_delete(&cl3);**

**clist\_delete(&cl4);**

Compile and test.

1. Define a function, **mk\_empty\_clist**, that takes no argument and returns an empty clist (*pointer-to-***cl\_node**). Add this code to **main**:

**cl\_node \*cl5 = mk\_empty\_clist();**

**clist\_display\_to\_cout(cl5); cout << '\n'; //** *(empty)*

**clist\_append\_char(&cl5, 'h');**

**clist\_append\_char(&cl5, 'i');**

**clist\_display\_to\_cout(cl5); cout << '\n'; // hi**

**clist\_delete(&cl5);**

Compile and test.

1. **const and Type Casts (5 points)**
2. Create a new, empty project named **hw2.2**. Download and add the existing **hw2.2.cpp** file, ***or*** add a new, empty source file named **hw2.2.cpp** and copy-and-paste the code below. Compile and test. This code should work fine: we have an initialized **const double** named **pi**, and a *pointer-to-***double**-**const** named **ppi**, initialized with the address of **pi**.

**// File: hw2.2.cpp**

**// Author(s):**

**#include <iostream>**

**#include <iomanip>**

**using namespace std;**

**int main()**

**{**

**const double pi(3.1415926535897932);**

**const double \*ppi{&pi};**

**cout << fixed << setprecision(15);**

**cout << "pi: " << pi << '\n';**

**cout << "&pi: " << &pi << '\n';**

**cout << "ppi: " << ppi << '\n';**

**cout << "\*ppi: " << \*ppi << '\n';**

**return 0;**

**}**

1. At the end of **main** (but above **return 0;**) add this statement:

**const double e;**

Should this work? Compile and test.

1. The statement in part (b) will not compile, because a **const** must be initialized. Add initialization for **e** and display its value with **cout**. Compile and test.

**const double e(2.7182818284590452);**

**cout << "e: " << e << '\n';**

1. Declare and initialize a pointer-to-double to point to e:

**double \*pe(&e); // pe points to e**

Should this work? Compile and test.

1. The statement in part (d) will not compile, because a *pointer-to-***double** (that is, a pointer to a variable) cannot point to a **const double**.

Use a *type cast* to force the compiler to treat the address of **e** as a *pointer-to-***double** rather than a *pointer-to-***double** **const**:

**double \*pe((double \*)&e); // force pe to point to e**

**cout << "pe: " << pe << '\n'; // address of e**

**cout << "\*pe: " << \*pe << '\n'; // value of e**

Compile and test. This should work fine.

1. Since **pe** is a *pointer-to-***double**, not a *pointer-to-***double** **const**, we can assign a value to the location that **pe** is pointing to. Add this code:

**\*pe = pi; // change e to pi?**

**cout << "\*pe: " << \*pe << '\n'; // is e now pi?**

Compile and test. Does this change **e** to **pi**’s value?

1. Display the value of **e** and the address of **e**, as well as the value of **pe** and the value of **\*pe**:

**cout << "e: " << e << '\n'; // is e now pi?**

**cout << "&e: " << &e << '\n'; // address of e**

**cout << "pe: " << pe << '\n'; // same as address of e?**

**cout << "\*pe: " << \*pe << '\n'; // value of e?**

Compile and test. Did your program change the value of **e**, or not?

The behavior is *undefined*: There are two conflicting things going on here. First, you cannot change the value of a **const**. Second, using a cast you can force the compiler to do what you want. Here is the output that I get. Notice that it is *different* in GNU C++ and Visual Studio!

**GNU C++ Visual Studio**

pi: 3.141592653589793 3.141592653589793

&pi: 0xffffcbd8 00F8F96C

ppi: 0xffffcbd8 00F8F96C

\*ppi: 3.141592653589793 3.141592653589793

e: 2.718281828459045 2.718281828459045

pe: 0xffffcbd0 00F8F950

\*pe: 2.718281828459045 2.718281828459045

\*pe: 3.141592653589793 3.141592653589793

e: 3.141592653589793 2.718281828459045

&e: 0xffffcbd0 00F8F950

pe: 0xffffcbd0 00F8F950

\*pe: 3.141592653589793 3.141592653589793

1. **Recursion (20 points)**
2. Create a new, empty project named **hw2.3**. Download and add the existing **hw2.3.cpp** file, ***or*** add a new, empty source file named **hw2.3.cpp**, and copy-and-paste the code below, from our lecture notes.

**// File: hw2.3.cpp**

**// Author(s):**

**#include <iostream>**

**using namespace std;**

**void put\_int\_bits(unsigned i)**

**{**

**if (i == 0) { // stopping condition**

**cout << 0; // leading 0 bit**

**}**

**else {**

**put\_int\_bits(i / 2); // hard task**

**cout << i % 2; // easy task**

**}**

**}**

**int main()**

**{**

**put\_int\_bits(0); // 0**

**cout << "\n";**

**put\_int\_bits(7); // 0111**

**cout << "\n";**

**put\_int\_bits(1234); // 010011010010**

**cout << "\n";**

**put\_int\_bits(-1); // 011111111111111111111111111111111**

**cout << "\n";**

**return 0;**

**}**

Compile and test.

1. The **put\_int\_bits** function is not very satisfactory, for at least a couple of reasons: (i) we would usually like to see all the bits that make up the **int** representation (32 bits, on our systems); and, (ii) we don’t want to see a leading 0 in front of the representation of a negative value: the first bit is the *sign bit*, which should be 1 for a negative value. Here is an improved version, **put\_int\_bits\_v2**, that uses a *helper function* for the low-level recursive work:

**void put\_int\_n\_bits(unsigned i, unsigned n)**

**{**

**if (n) {**

**put\_int\_n\_bits(i / 2, n - 1); // hard task**

**cout << i % 2; // easy task**

**}**

**}**

**void put\_int\_bits\_v2(unsigned i)**

**{**

**// assume 8 bits per byte**

**const size\_t bits\_per\_int = sizeof (int) \* 8;**

**put\_int\_n\_bits(i, bits\_per\_int);**

**}**

1. Add this code above the **return 0;** in **main**:

**put\_int\_with\_commas(0); cout << '\n';**

**put\_int\_with\_commas(1); cout << '\n';**

**put\_int\_with\_commas(12); cout << '\n';**

**put\_int\_with\_commas(123); cout << '\n';**

**put\_int\_with\_commas(1234); cout << '\n';**

**put\_int\_with\_commas(12345); cout << '\n';**

**put\_int\_with\_commas(123456); cout << '\n';**

**put\_int\_with\_commas(1234567); cout << '\n';**

**put\_int\_with\_commas(12345678); cout << '\n';**

**put\_int\_with\_commas(123456789); cout << '\n';**

**put\_int\_with\_commas(1234567890); cout << '\n';**

**put\_int\_with\_commas(INT\_MAX); cout << '\n';**

**put\_int\_with\_commas(-1); cout << '\n';**

**put\_int\_with\_commas(-12); cout << '\n';**

**put\_int\_with\_commas(-123); cout << '\n';**

**put\_int\_with\_commas(-1234); cout << '\n';**

**put\_int\_with\_commas(-12345); cout << '\n';**

**put\_int\_with\_commas(-123456); cout << '\n';**

**put\_int\_with\_commas(-1234567); cout << '\n';**

**put\_int\_with\_commas(-12345678); cout << '\n';**

**put\_int\_with\_commas(-123456789); cout << '\n';**

**put\_int\_with\_commas(-1234567890); cout << '\n';**

**put\_int\_with\_commas(INT\_MIN); cout << '\n';**

Include the header needed for **INT\_MAX** and **INT\_MIN**.

Define the **put\_int\_with\_commas** function and any needed helper(s) so that the output produced by these statements in **main** is:

**0**

**1**

**12**

**123**

**1,234**

**12,345**

**123,456**

**1,234,567**

**12,345,678**

**123,456,789**

**1,234,567,890**

**2,147,483,647**

**-1**

**-12**

**-123**

**-1,234**

**-12,345**

**-123,456**

**-1,234,567**

**-12,345,678**

**-123,456,789**

**-1,234,567,890**

**-2,147,483,648**

Compile and test.

1. Copy-and-paste these function definitions (from the lecture notes) above **main**:

**void put\_rev\_str\_it(char \*s) { // iterative version**

**char \*end(s); // start at beginning,**

**while (\*end++) // then find the end**

**;**

**int num = s – end; // how many chars?**

**while(num)**

**cout << s[num--]; // display in reverse**

**}**

**void put\_rev\_str\_rc(char \*s) { // recursive version**

**if (\*s) { // if there is a char**

**put\_rev\_str\_rc(s+1); // put trailing chars**

**cout << \*s; // then current char**

**}**

**}**

Above **return 0;** in **main**, copy-and-paste this code:

**char hello[] = "Hello World!";**

**put\_rev\_str\_it(hello);**

**cout << "\n";**

**put\_rev\_str\_rc(hello);**

**cout << "\n";**

Compile and test. You will discover that the iterative function **put\_rev\_str\_it** is badly broken. Either fix the bugs in this **put\_rev\_str\_it** function, or write a working **put\_rev\_str\_it** function of your own. Would you say that the iterative function is easier to understand than the recursive function, or vice versa? (There is not a “correct” answer to this question.)

1. **Binary Tree (40 points)**
2. Create a new, empty project named **hw2.4**. Download and add the existing **hw2.4.cpp** file (the binary search tree code from the lecture notes). Compile and test. Be sure the output makes sense to you.

1. Above **main**, define a function **put\_binary\_tree\_rev** that displays the values in the binary tree in reverse order, that is, from highest value to lowest. Add this code above **binary\_tree\_delete(&top);** in **main**:

**cout << "binary tree values in reverse:\n";**

**put\_binary\_tree\_rev(top);**

**cout << '\n';**

Compile and test.

1. Above **main**, define a function **binary\_tree\_contains** with this interface:

**bool binary\_tree\_contains(const bt\_node \*top, int val)**

This function returns **true** if the binary tree contains **val**, otherwise **false**. Add this test code above **binary\_tree\_delete(&top);** in **main**:

**cout << boolalpha;**

**cout << "binary tree contains 13: "**

**<< binary\_tree\_contains(top, 13) << '\n';**

**cout << "binary tree contains -7: "**

**<< binary\_tree\_contains(top, -7) << '\n';**

Compile and test.

1. Define a function **put\_binary\_tree\_pretty** that displays the binary tree “on its side” in an indented fashion: the top value should be at the left margin, the second-level values indented by one tab, the third-level values indented by two tabs, etc. Greater values should display above lesser values, so that the tree seems to have been “tipped over” on its left side. Add this call of **put\_binary\_tree\_pretty** above **binary\_tree\_delete(&top);** in **main**:

**cout << "pretty binary tree:\n";**

**put\_binary\_tree\_pretty(top);**

The output should look like:

**pretty binary tree:**

**29**

**13**

**12**

**9**

**7**

**-4**

This is a “tipped on its left side” representation of this tree:

**12**

**7**

**13**

**0**

**-4**

**0 0**

**9**

**0 0**

**29**

**0 0**

Compile and test.

1. The **binary\_tree\_insert** function allows the same value to be inserted into the tree multiple times. Confirm this by adding this code above **binary\_tree\_delete(&top);** in **main**:

**cout << "three more 7s:\n";**

**binary\_tree\_insert(&top, 7);**

**binary\_tree\_insert(&top, 7);**

**binary\_tree\_insert(&top, 7);**

**put\_binary\_tree\_pretty(top);**

Compile and test. The output for this part should look like:

**three more 7s:**

**29**

**13**

**12**

**9**

**7**

**7**

**7**

**7**

**-4**

Then, define a function **binary\_tree\_insert\_unique** that inserts a value into the tree only if the value does not already exist. Add this code above **binary\_tree\_delete(&top);** in **main**:

**cout << "new tree, unique values:\n";**

**bt\_node \*top2 = nullptr;**

**binary\_tree\_insert\_unique(&top2, 7);**

**binary\_tree\_insert\_unique(&top2, 4);**

**binary\_tree\_insert\_unique(&top2, 12);**

**binary\_tree\_insert\_unique(&top2, 7);**

**binary\_tree\_insert\_unique(&top2, 7);**

**binary\_tree\_insert\_unique(&top2, 9);**

**binary\_tree\_insert\_unique(&top2, 9);**

**binary\_tree\_insert\_unique(&top2, 12);**

**binary\_tree\_insert\_unique(&top2, 12);**

**binary\_tree\_insert\_unique(&top2, 4);**

**binary\_tree\_insert\_unique(&top2, 2);**

**binary\_tree\_insert\_unique(&top2, 2);**

**put\_binary\_tree\_pretty(top2);**

**binary\_tree\_delete(&top2);**

Compile and test. The output for this part should look like:

**new tree, unique values:**

**12**

**9**

**7**

**4**

**2**

1. Define a function **binary\_tree\_size** that returnhhlls the number of nodes in the tree. Add this code above **binary\_tree\_delete(&top2);** in **main**:

**bt\_node \*top3 = nullptr;**

**cout << "top size: " << binary\_tree\_size(top) << "\n";**

**cout << "top2 size: " << binary\_tree\_size(top2) << "\n";**

**cout << "top3 size: " << binary\_tree\_size(top3) << "\n";**

Compile and test.

1. Define a function **binary\_tree\_sum** that returns the sum of the values in the tree. If the tree is empty, return 0. Add this code above **binary\_tree\_delete(&top2);** in **main**:

**cout << "top sum: " << binary\_tree\_sum(top) << "\n";**

**cout << "top2 sum: " << binary\_tree\_sum(top2) << "\n";**

**cout << "top3 sum: " << binary\_tree\_sum(top3) << "\n";**

Compile and test.

1. Define a function **binary\_tree\_mean** that returns the mean of the values in the tree. If the tree is empty, return **DBL\_MIN**. Add this code above **binary\_tree\_delete(&top2);** in **main**:

**cout << "top mean: " << binary\_tree\_mean(top) << "\n";**

**cout << "top2 mean: " << binary\_tree\_mean(top2) << "\n";**

**cout << "top3 mean: " << binary\_tree\_mean(top3) << "\n";**

Compile and test.

1. Define a function with this interface:

**binary\_tree\_delete\_value(bt\_node \*\*ptop, int val)**

This function should delete any node(s) where the value is **val**, while retaining the

rule about the structure of the binary tree: at each node, values less than the

node’s value are in the left subtree, and values greater than (or equal to) the

node’s value are in the right subtree. ***Hint:*** If you delete a node that has left and

right subtrees, one way of maintaining the ordering rule is to attach the right

subtree at the right-most leaf of the left subtree, and move the left subtree up in

place of the deleted node. Draw pictures!

Add this code above **binary\_tree\_delete(&top2);** in **main**:

**binary\_tree\_delete\_value(&top, 7);**

**binary\_tree\_delete\_value(&top2, 7);**

**cout << "top with 7 deleted:\n";**

**put\_binary\_tree\_pretty(top);**

**cout << "top2 with 7 deleted:\n";**

**put\_binary\_tree\_pretty(top2);**

**binary\_tree\_delete(&top3); // properly deallocate top3**

Compile and test. The output for this part *could* look like:

**top with 7 deleted:**

**29**

**13**

**12**

**9**

**-4**

**top2 with 7 deleted:**

**12**

**9**

**4**

**2**

Your trees might be shaped differently, if you followed a different implementation strategy, but these should be the values in your trees, and the ordering rule must be maintained.

***REMEMBER*** to put all authors’ names into each of your source code files.Put your **hw2.1.cpp, hw2.2.cpp**, **hw2.3.cpp**, and **hw2.4.cpp** code files into a **.zip** archive and upload to the course web site.