CSCI-1200 Data Structures
Test 2 — Practice Problems

Note: This packet contains practice problems from three previous exams. Your exam will contain approximately one third as many problems.

1 Dynamically Allocated & Templated Stairs [ / 28 ]

In this problem you will write a simple class to build a staircase-shaped storage shelf. Here’s an example usage of the class, which constructs the diagram on the right.

```cpp
Stairs<char> s(4,'_');
s.set(0,0,'A');
s.set(1,1,'B');
s.set(2,2,'C');
s.set(3,3,'D');
s.set(2,1,'U');
s.set(3,1,'S');
```

1.1 Stairs Class Declaration [ / 14 ]

First, fill in the blanks in the class declaration:

```cpp
class Stairs {
public:
    // constructor
    sample solution: 1 line(s) of code
    // destructor
    sample solution: 1 line(s) of code
    // prototypes of 2 other important functions related to the constructor & destructor
    sample solution: 2 line(s) of code
    // modifier
    void set(int i, int j, val) { data[i][j] = val; }
    // NOTE: other Stair functions omitted */
private:
    // representation
    sample solution: 2 line(s) of code
};
```
1.2 Stairs Constructor [ / 9 ]
Now write the constructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

sample solution: 10 line(s) of code

1.3 Stairs Destructor [ / 5 ]
Now write the destructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

sample solution: 6 line(s) of code
Ben Bitdiddle is working on a software project for essay writing using a doubly-linked chain of nodes. His initial Node class is on the right.

One of the features of his software allows a user to compare the location of two words within the document and say which word appears first. Ben plans to implement this using two helper functions: search and compare.

### 2.1 Searching for a Word

First, let's write the search function, which takes in two arguments: a pointer to the first Node in the document (word chain) and the specific word we’re looking to find. The function returns a pointer to the first Node containing that word. Use recursion to implement this function.

```cpp
class Node {
public:
   std::string word;
   Node* next;
   Node* prev;
};
```

*sample solution: 7 line(s) of code*

If the Node chain contains $n$ elements, what is the running time of the search function?

### 2.2 Comparing Positions within the Node Chain

Next, let’s implement the compare function. This function takes in two Node pointers and returns true if the first argument appears closer to the front of the list than the second argument. For example, let’s say a chain of word Nodes named sentence contains:

```
the quick brown fox jumps over the lazy dog
```

Here’s an example using the search and compare functions:

```cpp
Node* over = search(sentence, "over");
Node* quick = search(sentence, "quick");
Node* lazy = search(sentence, "lazy");

assert (compare(quick, over) == true);
assert (compare(over, quick) == false);
assert (compare(quick, lazy) == true);
assert (compare(lazy, over) == false);
```
Again using recursion, implement the compare function.

sample solution: 7 line(s) of code

If the Node chain contains \( n \) elements, what is the running time of the compare function?

Improving Word Position Comparison Performance

Alyssa P. Hacker stops by to help, and suggests that Ben switch to a different data structure if he is frequently comparing word positions within a long essay.

But Ben’s a stubborn guy. Instead of switching to a different data structure, he has a plan to augment his list structure to improve the running time of compare. Ben explains that the new distance member variable in each node will indicate how far away the node is from the front of the list.

Here’s Ben’s new compare function:

```cpp
bool compare_fast(Node *a, Node *b) {
    return a->distance < b->distance;
}
```

Ben reassures Alyssa that he’ll add some error checking to this code. 

*SIDE NOTE: Hopefully your implementation of the original compare function has some error checking!*

But Alyssa is more concerned about how this addition to the data structure will impact performance when the essay or sentence is edited. She says he can’t afford to change the distance in all or many Nodes in the data structure any time a small edit is made to the document.

Ben explains that the push_back function will assign the distance of the new Node to be the distance of the last Node in the chain plus 10.0. And similarly, push_front will assign the new Node to be the distance from the first Node minus 10.0. *BTW, negative distance values are ok.* Finally, Ben says the insert_between function (on the next page) can similarly be implemented without editing the distance value in any existing Node!
Continuing with the previous example, here’s a quick demonstration of how this function works:

```cpp
bool success = insert_between(sentence, "the", "lazy", "VERY");
assert (success);
Node* VERY = search(sentence, "VERY");
assert (compare(VERY, lazy) == true);
assert (compare(quick, VERY) == true);
assert (compare_fast(VERY, lazy) == true);
assert (compare_fast(quick, VERY) == true);
success = insert_between(sentence, "quick", "fox", "RED");
assert (!success);
```

And here’s the contents of the sentence variable after the above fragment of code:

```
the quick brown fox jumps over the VERY lazy dog
```

Implement insert_between. And yes, use recursion.
3    Erase Middles [ / 20 ]

Write a function named erase_middles that takes in 2 arguments: an STL list named data and a value. The function should remove all instances of value from data, except the first and the last instances. The function returns the number of removed elements. For example, if data initially contains:

5 2 5 2 3 4 3 2 5 2 3 2 3 4 2 5

A call to erase_middles(data,5) will return 2 and now data contains:

5 2 2 3 4 3 2 2 3 2 3 4 2 5

And then a call to erase_middles(data,2) will return 4 and data contains:

5 2 3 4 3 3 3 4 2 5

sample solution: 22 line(s) of code
For each program bug description below, write the letter of the most appropriate debugging skill to use to solve the problem. Each letter should be used at most once.

A) get a backtrace
B) add a breakpoint
C) use step or next
D) add a watchpoint
E) examine different frames of the stack
F) reboot your computer
G) use Dr Memory or Valgrind to locate the leak
H) examine variable values in gdb or lldb

A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.

The program always gets the right answer, but when I test it with a complex input dataset that takes a long time to process, my whole computer slows down.

I'm unsure where the program is crashing.

I've got some tricky math formulas and I suspect I've got an order-of-operations error or a divide-by-zero error.

I'm implementing software for a bank, and the value of a customer's bank account is changing in the middle of the month. Interest is only supposed to be added at the end of the month.

Select one of the letters you did not use above, and write a concise and well-written 3-4 sentence description of a specific situation where this debugging skill would be useful.
Flipping & Sorting Words

Finish the implementation of the function `FlipWords` that takes in an *alphabetically sorted* STL list of STL `string`s named `words` and modifies the list. The function should remove all palindromes (words that are the same forwards & backwards). The function should insert the flipped (reversed) version of all other words into the list, *in sorted order*. For example this input list:

```
bard civic diva flow pots racecar stop warts
```

Should be changed to contain:

```
avid bard diva drab flow pots stop straw warts wolf
```

You may not use STL `sort`. You may assume the input list does not contain any duplicates. And after calling the `FlipWords` function the list should not contain any duplicates.

```cpp
std::string reverse(std::string &word) {
    std::string answer(word.size(),' ');
    for (int i = 0; i < word.size(); i++) { answer[i] = word[word.size()-1-i]; }
    return answer;
}

void FlipWords(std::list<std::string> &words) {
    sample solution: 1 line(s) of code
    while (current != words.end()) {
        std::string flip = reverse(*current);
        if (flip == *current) {
            sample solution: ≤3 line(s) of code
        } else {
            sample solution: ≤8 line(s) of code
        }
    }
}
```
Ben Bitdiddle thinks he has stumbled on a brilliant idea to make each Node of a doubly linked list “smart” and store global information about the list. Each Node will have a pointer to the head and tail Nodes of the overall list.

Help him by finishing the implementation of PushFront to add a new element to the list. Note: You should not change the value inside of any existing Nodes.

```cpp
void PushFront(Node* head, Node* tail, int v) {
    Node* tmp = new Node;
    tmp->value = v;
    if (head == NULL) {
        // sample solution: 4 line(s) of code
    } else {
        // sample solution: 9 line(s) of code
    }
}
```
Alyssa P. Hacker has joined the Rensselaer Center for Open Source Software and is working on a program to help students manage their schedules over their time at RPI. She will use a two dimensional array to store courses taken each term. The declaration for two key classes is shown on the right:

Alyssa's program assumes that all undergraduate RPI degree programs require students to take 32 4-credit courses. She also assumes that each specific student takes the same number of courses per term throughout their time at RPI.

Your task is to implement the critical functions for this class with dynamically allocated memory, as they would appear in the `Student.cpp` file. Make sure to use the private helper functions as appropriate so your code is concise.

A few examples of usage are shown below.

```cpp
class Course {
public:
    Course(const std::string &p="XXXX", int n=1000)
        : prefix(p), num(n) {}
    /* member functions omitted */
private:
    std::string prefix;
    int num;
};

class Student {
public:
    Student();
    Student(int courses_per_term_);
    Student(const Student& s);
    ~Student();
    const Student& operator=(const Student& s);
    int numTerms() const { return num_terms; }
    const Course& getCourse(int t, int c) const
        { return data[t][c]; }
    /* additional member functions omitted */
private:
    void initialize();
    void copy(const Student& s);
    void destroy();
    int num_terms;
    int courses_per_term;
    Course** data;
};
```

// a typical student takes 4 courses per term for 8 terms
Student regular; assert (regular.numTerms() == 8);
// if a student takes 5 courses per term, they can finish in 3.5 years
Student overachiever(5); assert (overachiever.numTerms() == 7);
// students who take 3 courses per term will require 5.5 years
Student supersenior(3); assert (supersenior.numTerms() == 11);
/* details of how courses are scheduled omitted */

```cpp
Student::Student() {
    sample solution: 3 line(s) of code
}

Student::Student(int courses_per_term_) {
    sample solution: 3 line(s) of code
}
```
Student::Student(const Student& s) {
    // sample solution: 1 line(s) of code
}

const Student& Student::operator=(const Student& s) {
    // sample solution: 5 line(s) of code
}

Student::Student() {
    // sample solution: 1 line(s) of code
}

void Student::initialize() {
    // sample solution: 4 line(s) of code
}

void Student::copy(const Student& s) {
    // sample solution: 8 line(s) of code
}

void Student::destroy() {
    // sample solution: 4 line(s) of code
}
8 Reverse Iterators [ / 10 ]

Complete the function below named `reverse` that takes in an STL `list` as its only argument and returns an STL `vector` that contains the same list except in reverse order. You should use a `reverse iterator` and you may not use `push_back`.

```cpp
reverse( my_list ) {
    sample solution: 3 line(s) of code
    while (itr != my_list.rend()) {
        sample solution: 3 line(s) of code
    }
    return answer;
}
```

9 Order Notation [ / 5 ]

Rank these 6 order notation formula from fastest(1) to slowest(6).

\[
\begin{align*}
O(8 \cdot s \cdot w \cdot h) & & O(w \cdot h \cdot 8^s) \\
O((s \cdot w \cdot h)^8) & & O((s + w \cdot h)^8) \\
O((8 \cdot w \cdot h)^s) & & O(w \cdot h \cdot s^8)
\end{align*}
\]
10 Dynamic Tetris Arrays [ /26]

10.1 HW3 Tetris Implementation Order Notation [ /6]

Match up the Tetris class member functions from HW3 with the appropriate order notation, where \( w \) is the width of the board and \( h \) is the maximum height of any column. Assume the solution is efficient, but uses only the 3 member variables specified in the original assignment (data, heights, and width).

*Note: Some letters may be used more than once or not at all.*

- `void add_piece(char piece, int rotation, int position);`
  - a) \( O(1) \)
- `int get_width();`
  - b) \( O(w) \)
- `int remove_full_rows();`
  - c) \( O(h) \)
- `int get_max_height();`
  - d) \( O(w + h) \)
- `void destroy();`
  - e) \( O(w \times h) \)

10.2 Tetris Representation Conversion [ /20]

Now let’s revisit the details of the dynamic memory representation for the game of Tetris. Your task is to convert a Tetris board from the column representation we used for HW3 to a row representation. In addition to the three member variables in our HW3 Tetris class: data, heights, and width, we add 2 additional member variables: widths and height. In the column representation we don’t need the widths variable, so it is set to NULL. Each time the board is modified to add Tetris pieces or score full rows the height variable is updated as necessary to store the maximum height of any column.

The diagram on the left shows an example Tetris board first in column representation and then in row representation — the “before” and “after” diagrams for a call to the new Tetris class member function convert_to_row_representation. Note that once in row representation the heights variable isn’t needed and we set it to NULL. The convert_to_row_representation function takes no arguments.
Now write the **Tetris** class member function `convert_to_row_representation` as it would appear in the `tetris.cpp` implementation file. You may assume that before the call the board is in the column representation and the member variables are all set correctly. Make sure your code properly allocates new memory as needed and does not have memory leaks.

*sample solution: 23 line(s) of code*
11 Mystery Recursion

For each function or pair of functions below, choose the letter that best describes the program purpose or behavior.

A ) infinite loop  E ) function is not recursive  I ) reverse the digits
B ) factorial  F ) sum of the digits  J ) multiplication
C ) integer power  G ) syntax error  K ) greatest common divisor
D ) the answer is 42  H ) modulo 2  L ) other

```c
int mysteryONE(int x, int y) {
    if (y == 0)
        return x;
    else
        return mysteryONE(y, x % y);
}
```

```c
int mysteryFOUR(int x, int y) {
    if (x == 0)
        return 0;
    else
        return y + mysteryFOUR(x-1, y);
}
```

```c
int mysteryTWO(int x) {
    if (x == 0)
        return 0;
    else
        return mysteryTWO(x/10) + x%10;
}
```

```c
int mysteryFIVEa(int x, int y) {
    if (x == 0)
        return y;
    else
        return mysteryFIVEa(x/10, y*10 + x%10);
}
```

```c
int mysteryFIVEb(int x) {
    return mysteryFIVEa(x, 0);
}
```

```c
int mysterySIX(int x) {
    if (x == 0)
        return 1;
    else
        return x * mysterySIX(x-1);
}
```
Write a function named Collect that takes in two alphabetically sorted STL lists of STL strings named threes and candidates. The function searches through the second list and removes all three letter words and places them in the first list in alphabetical order. For example, given these lists as input:

```
threes: cup dog fox map
candidates: ant banana egg goat horse ice jar key lion net
```

After the call to `Collect(threes, candidates)` the lists will contain:

```
threes: ant cup dog egg fox ice jar key map net
candidates: banana goat horse lion
```

If there are \( n \) and \( m \) words in the input lists, the order notation of your solution should be \( O(n + m) \).
The expected output of the program below is:

chris is a sophomore, his/her favorite color is blue, and he/she has used 1 late day(s).

However, there are a number of small but problematic errors in the DSStudent class code. Hint: This problem’s title is relevant! Only one completely new line may be added (line 6), and the 7 other lines require one or more small changes. These lines are tagged with an asterisk, *. Your task is to rewrite each incorrect or missing line in the appropriately numbered box. Please write the entire new line in the box.

```cpp
1 class DSStudent {
  2 public:
* 3     DSStudent(std::string n, int y) : name(n) {
  4       int entryYear = y;
* 5     }
* 6
  7 }
  8     std::string& getName() const {
  9       return name;
 10   }
*11 const std::string& getYear() {
 12       if (entryYear == 2014) {
 13           return "freshman";
 14       } else if (entryYear == 2013) {
 15           return "sophomore";
 16       } else if (entryYear == 2012) {
 17           return "junior";
 18       } else {
 19           return "senior";
 20     }
 21   }
 22     void incrLateDaysUsed() const {
 23       days++;
 24   }
 25     int& getLateDaysUsed() const {
 26       return days;
 27   }
 28     std::string FavoriteColor() {
 29       return color;
 30   }
 31   private:
 32     std::string name;
 33     std::string color;
 34     int entryYear;
 35     int days;
 36   };
 37
 38 int main() {
 39     DSStudent s("chris", 2013);
 40     s.FavoriteColor() = "blue";
 41     s.incrLateDaysUsed();
 42     std::cout << s.getName() << " is a " << s.getYear() << ", his/her favorite color is " << s.FavoriteColor() << "", and he/she has used " << s.getLateDaysUsed() << " late day(s)." << std::endl;
```
Write a recursive function named `occurrences` that takes in a sorted STL vector of STL strings named `data`, and an STL string named `element`. The function returns an integer, the number of times that `element` appears in `data`. Your function should have order notation $O(\log n)$, where $n$ is the size of `data`.

Sample solution: 21 line(s) of code
15 Short Answer [ / 8 ]

15.1 What’s Wrong? [ / 4 ]

Write 1-2 complete and concise sentences describing the problem with this code fragment:

```cpp
std::vector<std::string> people;
people.push_back("sally");
people.push_back("brian");
people.push_back("monica");
people.push_back("fred");
std::vector<std::string>::iterator mom = people.begin() + 2;
std::vector<std::string>::iterator dad = people.begin() + 1;
people.push_back("paula");
std::cout << "My parents are " << *mom << " and " << *dad << std::endl;
```

15.2 Fear of Recursion [ / 4 ]

Rewrite this function without recursion:

```cpp
class Node {
public:
    std::string value;
    Node* next;
};

void printer (Node* n) {
    if (n->next == NULL) {
        std::cout << n->value;
    } else {
        std::cout << "(" << n->value << "+"
        printer (n->next);
        std::cout << ");";
    }
}
```

Sample solution: 13 line(s) of code
Ben Bitdiddle is working on a project that stores data with two different data structures: our Vec and dslist classes. Occasionally he needs to convert data from one format to the other format. Alyssa P. Hacker suggests that he write a copy-constructor-like function for each class that takes in a single argument, the original format of the data. For example, here’s how to convert data in Vec format to dslist format:

// create a Vec object with 4 numbers
Vec<int> v; v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
// create a dslist object that initially stores the same data as the Vec object
dslist<int> my_lst(v);

Here are the relevant portions of the two class declarations (and the Node helper class):

```
template <class T> class Node {
    public:
        Node(const T& v):
            value_(v), next_(NULL), prev_(NULL) {}  // constructor
        T value_;  // value of the node
        Node<T>* next_;  // pointer to the next node
        Node<T>* prev_;  // pointer to the previous node
};

template <class T> class Vec {
    public:
        // conversion constructor
        Vec(const dslist<T>& lst);
        /* other functions omitted */
        // representation
        T* m_data;
        unsigned int m_size;
        unsigned int m_alloc;
};

template <class T> class dslist {
    public:
        // conversion constructor
        dslist(const Vec<T>& vec);
        /* other functions omitted */
        // representation
        Node<T>* head_;  // pointer to the head node
        Node<T>* tail_;  // pointer to the tail node
        unsigned int size_;  // size of the list
};
```

Ben asks about access to the private member variables of one class from a member function of the other. Alyssa says he can write the functions assuming he has full access to the private member variables. (She promises to teach him how to use the friend keyword to make that work after Test 2.)

### 16.1 Diagrams

First, draw the detailed internal memory representations for a Vec object and a dslist object, each storing the numbers: 1 2 3 4.

<table>
<thead>
<tr>
<th>m_data:</th>
<th>m_alloc:</th>
<th>m_size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail_:</td>
<td>head_:</td>
<td>size_:</td>
</tr>
</tbody>
</table>
16.2 Implementing the Conversion Constructors

Now write the two conversion constructors. You may not use push_back, push_front, insert or iterators in your answer. Instead, demonstrate that you know how to construct and manipulate the low level memory representation.

template <class T> Vec<T>::Vec(const dslist<T>& lst) {

sample solution: 13 line(s) of code

} // Vec<T>()

template <class T> dslist<T>::dslist(const Vec<T>& v) {

sample solution: 13 line(s) of code

} // dslist<T>()