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 Iterating Over a List and Inserting New Points [/21]

In this question you are asked to write two functions that use iterators to traverse and manipulate STL lists.

Don't worry about `#include` or `#define` statements. You may assume `using namespace std`.

1.1 `Lists` Finding Zeros [/6]

First, write a function that will be passed a **const** reference to an STL list of ints and returns a new list of ints containing the positions of zeros in the input list. The function should iterate through the list using a list iterator. It should not use *find()* or similar STL functions.

For example, if the original list contained 1 0 11 16 0 0 50 75 85 90 0, the returned list should contain 1 4 5 10.

sample solution: 12 line(s) of code

1.2 `Lists` Replacing Zeros[/15]

Now, write a second function. This **void** function will be passed a reference to an STL list of ints. In this function each zero in the list should be replaced with the sum of the adjacent numbers. A zero in the first or last position of the list should not be replaced. For example, if the list originally contained 1 0 11 16 0 0 50 75 85 90 0, the returned list will contain 1 12 11 16 16 66 50 75 85 90 0. Iterate through the list from left to right and replace the elements sequentially. Notice how consecutive zeros are handled. The first zero is replaced and the replacement value becomes the adjacent value for the next zero. That is, a list containing x 0 0 0 y will become x x x x+y y, where x and y are integers.

The zeros are to be replaced in the original list. Do not make a copy of the list. Iterate through the list and replace the elements. Do not use *std::replace* or *std::find*.

sample solution: 15 line(s) of code

2 Recursive Lists [/25]

In this question, don't worry about `#include` or `#define` statements. You may assume `using namespace std`.

2.1 `Recursive Lists` Delete a List[/6]

A templated class for Nodes is defined as:

```
template <class T>
class Node {
public:
  T value;
  Node<T>* next;
};
```

First, write a templated **recursive** function to delete a list of $\mathit{Node} < T >$s.

sample solution: 8 line(s) of code

2.2 `Recursive Lists` Recursive Merge [/19]

Merging two or more sorted lists is a common operation. It is a basic part of the merge sort which we recently covered in lecture. The idea behind merging two lists is to travel through the two sorted input lists and produce a third *sorted* list containing all of the elements of the two original lists.

For example, if the first list contains *apple cow rhino tree* and the second list contains *cat dog mongoose zebra*, the merged list should contain *apple cat cow dog mongoose rhino tree zebra.*

Write a templated **recursive** function that takes two pointers to sorted singly linked lists of *Nodes*, defined on the previous page, and a reference to a pointer to a singly linked list of *Nodes.* On return from the function, the third list should contain the merged sorted list. Your merged list must copy the data in the sorted lists.

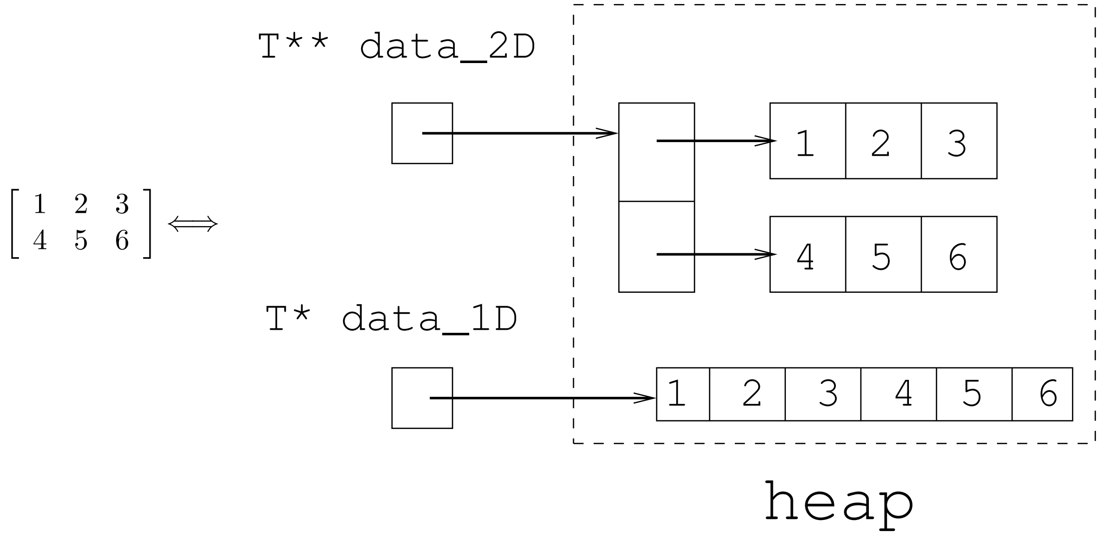
The function must be *recursive.* Do not sort the list. Merge the lists, don't sort. These are not STL lists, use the pointers to iterate through the lists. You may include any helper functions that you find necessary. Do not edit the Node class. You don't have to write a main() function.

sample solution: 20 line(s) of code

3 Templating and Flattened Matrices [/19]

In Homework 3 we explored a matrix based around a 2D data structure. We will not be using the *Matrix* class we designed, but instead will be using a templated *T*** `data_2D` data structure to represent a matrix. The layout should look familiar.

In addition to the 2D representation, we would like to have the option to work on a "flattened" version of the matrix, which is implemented in a *T** `data_1D` structure. Both structures are depicted below. We would like to be able to go between the two data structures.



$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Longleftrightarrow$
T** data_2D
1 2 3
4 5 6
T* data_1D
1 2 3 4 5 6
heap

Below is an example of how the two functions we will write might be used, along with the output from this code fragment.

```
int** data_2D = new int*[2];
data_2D[0] = new int[3];
data_2D[1] = new int[3];
data_2D[0][0] = 1; data_2D[0][1] = 2;
data_2D[0][2] = 3; data_2D[1][0] = 4;
data_2D[1][1] = 5; data_2D[1][2] = 6;

int* data_1D = Flatten(data_2D,2,3);

for(int i=0; i<2; i++){
  for(int j=0; j<3; j++){
    int index;
    int readval = Read1D(data_1D,2,3,i,j,index,-1);
    std::cout << "(" << index << "," << readval << ") ";
  }
}

//Assume and Delete1D/2D are already written
Delete2D(data_2D,2,3);
Delete1D(data_1D);

Output:
(0,1) (1,2) (2,3) (3,4) (4,5) (5,6)
```

3.1 Flattening the Matrix [/10]

Your first task is to implement the templated *flatten* function. *flatten* should take in `data_2D` as shown on the previous page, and two integers `m` and `n`, which are the number of rows and the number of columns in the data respectively. *flatten* should return a pointer to an equivalent 1D data structure.

If either the number of rows or columns is non-positive, the function should return a NULL pointer.

Do not change `data_2D`. **Only** use const and & when needed. Remember that since *flatten* is templated it should work with **any** datatype. Do not leak any memory, any memory still allocated must be reachable from the returned pointer.

sample solution: 14 line(s) of code

If the 2D matrix reprensentation contains m rows and n columns, what is the running time of the *flatten* function?

3.2 Reading From the Flattened Matrix [/9]

Another important task is to be able to read from the data structure. Write a function *Read1D* that takes in a 1D representation of data `data_1D`, two integers `m` and `n` which are the number of rows and columns respectively, two integers `row` and `col` which are the row and column position we are trying to extract data from, a reference to an integer `index`, and a failure_value which will be returned in case of an error.

Just like in Homework 3, we will number the upper left corner of a 2D structure as $(0, 0)$ or `row`$= 0$, `col`$= 0$.

The function should do two things. If the dimensions are legal (i.e. there are a positive number of rows and columns), and the requested position can exist within the given bounds, then the function should return the data stored at that position and set `index` to the index in `data_1D` where the data came from. If the dimensions are illegal or the requested position is out of bounds, the index should be set to -1 and `failure_value` should be returned.

Keep in mind that the same `data_1D` object can be viewed different ways. For example, if there is a 2×3 `data_2D_example` and `data_1D_example` = *flatten*(`data_2D_example`,...), then after calling *Read1D*(`data_1D_example`,1,6,1,1,index,-1), `index` will be -1, because in this example call we specified that `m`$= 1$, `n`$= 6$, and there is no position $(1, 1)$ inside of a 1×6 matrix.

On the other hand, using the same `data_1D_example`, *Read1D*(`data_1D_example`,2,3,1,1,index,-1) would set `index`$= 4$.

Do not call any STL functions. **Only** use const and & when needed. Remember that since *Read1D* is templated it should work with **any** datatype.

sample solution: 11 line(s) of code

4 Memory Errors [/9]

For each function or pair of functions below, choose the letter that best describes the memory error that you would find. You can assume `using namespace std` and any necessary `#include` statements.

A) use of uninitialized memory

B) mismatched new/delete/delete[]

C) memory leak

D) already freed memory

E) no memory error

F) invalid write

□

```
char* a = new char[6];
a[0] = 'B';  a[1] = 'y';
a[2] = 'e';  a[3] = '\0';
cout << a << endl;
delete a;
```

□

```
int a[2];
float** b = new float*[2];
b[0] = new float[1];
a[0] = 5; a[1] = 2;
b[0][0] = a[0]*a[1];
delete [] b[0];
b[0] = new float[0];
delete [] b;
```

□

```
int a[10];
int b[5];
for(int i=10; i>5; i--){
  a[((i-6)*2+1)] = i*2;
  a[((i-6)*2)] = b[i-6];
  cout << a[(i-6)*2] << endl;
}
```

□

```
string* str1 = new string;
string* str2;
string* str3 = new string;
*str1 = "Hello";
str2 = str1;
*str3 = *str1;
delete str1;
delete str3;
delete str2;
```

□

```
bool* is_even = new bool[10];
for(int i=0; i<=10; i++){
  is_even[i] = ((i%2)==0);
}
delete [] is_even;
```

□

```
int x[3];
int* y = new int[3];
for (int i=3; i>=1; i--){
  y[i-1] = i*i;
  x[i-1] = y[i-1]*y[i-1];
}
delete [] y;
```

5 Complexity Code Writing [/ 9]

For each of the problems below, write a function `void complexity(int n)` that satisfies the big-O running time. Assume that the input is size n. You should not use anything that requires a `#include` statement. You should write no more than 7 lines of code per box (including the function prototype).

$O(1)$:

```
void complexity(int n){
```

sample solution: 1 line(s) of code

```
}
```

$O(n^2)$:

```
void complexity(int n){
```

sample solution: 5 line(s) of code

```
}
```

$O(log\ n)$

For this one, do not use any loops, do not use math functions such as *log()* or *log2()*:

```
void complexity(int n){
```

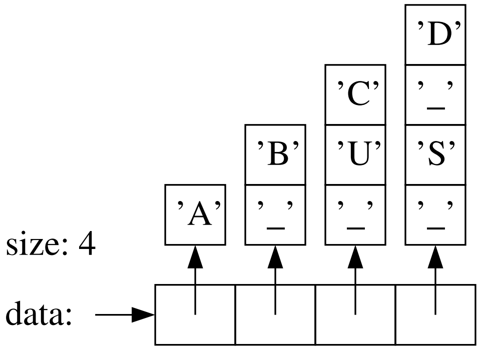
sample solution: 4 line(s) of code

```
}
```

6 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.

```
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');
```



6.1 Stairs Class Declaration [/ 14]

First, fill in the blanks in the class declaration:

```
                                        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
};
```

6.2 `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

6.3 `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

7 Comparing Linked List Pointers w/ Recursion [/ 32]

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

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`.

7.1 Searching for a Word [/ 7]

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.

sample solution: 7 line(s) of code

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

7.2 Comparing Positions within the Node Chain [/ 8]

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 `Node`s named `sentence` contains:

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

Here's an example using the `search` and `compare` functions:

```
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.

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

Here's Ben's new compare function:

```
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 `Node`s 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`!

7.3 Implementing insert_between and Maintaining Fast Comparisons [/ 17]

Continuing with the previous example, here's a quick demonstration of how this function works:

```
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*.

sample solution: 16 line(s) of code

8 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

9 Debugging Skillz [/ 17]

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.

10 Flipping & Sorting Words [/ 18]

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.

```
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 {
```

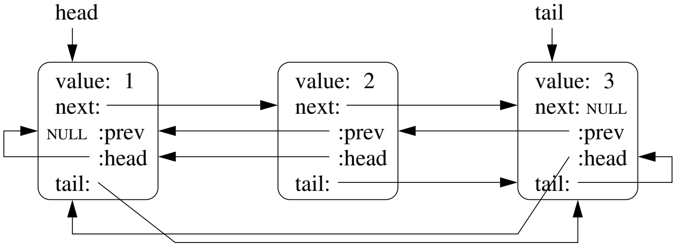
sample solution: ≤8 line(s) of code

```
    }
  }
}
```

11 “Smart” List Nodes [/ 18]

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` `Node`s of the overall list.

```
class Node {
public:
  Node* head;
  Node* tail;
  Node* next;
  Node* prev;
  int value;
};
```



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* `Node`*s.*

```
void PushFront(                    head,                    tail, int v) {

  Node* tmp =                                              ;

  tmp->value = v;
  if (head == NULL) {
```

sample solution: 4 line(s) of code

```
  } else {
```

sample solution: 9 line(s) of code

```
  }
}
```

12 Dynamically Allocated Student Schedules [/ 30]

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.

```
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);
  const Student& operator=(const Student& s);
  ~Student();
  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 */
```

```
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

```
}
```

13 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`.

```
reverse(                    my_list) {
```

sample solution: 3 line(s) of code

```
    while (itr != my_list.rend()) {
```

sample solution: 3 line(s) of code

```
    }
    return answer;
}
```

14 Order Notation [/ 5]

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

□ $O(8 \cdot s \cdot w \cdot h)$

□ $O((s \cdot w \cdot h)^8)$

□ $O((8 \cdot w \cdot h)^s)$

□ $O(w \cdot h \cdot 8^s)$

□ $O((s + w \cdot h)^8)$

□ $O(w \cdot h \cdot s^8)$