7:58

| chenz11@rpi.edu | Zhengneng Chen | lab section: 6 |
|---|---|---|
| room: DARRIN 324 | zone: TEAL row: 9 seat: 3L | 6-7:50pm |

# CSCI-1200 Data Structures — Spring 2018
# Exam 2 — Monday, March 5th, 6-7:50pm

| | | |
|---|---|---|
| | / 3 | |
| 1 | / 16 | |
| 2 | / 20 | |
| 3 | / 20 | |
| 4 | / 20 | |
| 5 | / 21 | |
| Total | / 100 | |

**Name your undergraduate lab mentors grouped by hair color:**

Black: Samuel

- **This exam has 14 pages: 11 pages of questions, 1 blank page, and the 2 pages of notes you uploaded to Submitty. Let us know if you are missing a page. DO NOT SEPARATE YOUR EXAM.**
- This test is closed-book and closed-notes *except for 2 sheets of notes on 8.5x11 inch paper (front & back) that are pre-printed.* Put away all other papers and books. Computers, cell phones, calculators, PDAs, MP3 players, etc. are not permitted and **must be turned off and placed under your desk.**
- Many of the problems require you to write a C++ function. If the function declaration is not given, part of your grade will depend on specifying the parameters correctly. In solving any problem, you may write additional functions to better structure your solution.
- Write your answer in the box provided below each question. Be sure to write neatly. If we can't read your solution, we won't be able to give you full credit for your work. Please state clearly any assumptions that you made in interpreting a question.
- You can assume appropriate STL `#include` statements and `using namespace std`. You are allowed to use `std::` if you prefer.

Zhengneng Chen chenz11@rpi.edu

## 1 Time Complexity [ / 20]

For each of the functions, write the time complexity assuming there are $n$ elements stored in the container. If there is a difference between C++98 and C++11, you should assume C++11. For the singly-linked list, assume that we only have the *head_* pointer and no other member variables. For the singly-linked list, assume that *erase()* is given a pointer to the node we want to erase **and** a pointer to the node before the one we want to erase.

| | STL vector or Vec<T> | Singly-linked List | STL list or dslist<T> |
|---|---|---|---|
| size() | O(1) | O(n) | O(1) |
| push_back() | O(1) | O(n) | O(1) |
| erase() | O(n) | O(1) | O(1) |
| insert() | O(n) | O(1) | O(1) |
| pop_back() | O(1) | O(n) | O(1) |

Write 2-3 complete sentences about one of the above methods which is more efficient for STL `lists` than for STL `vectors` and why this is the case.

Erase and insert in the middle of the STL list is very efficient, independent of the size of the list. Because both operations are implemented by rearranging pointers between the small blocks of memory.

## 2 Pokémon Battles [ / 16]

Players, known as trainers, collect teams of Pokémon. Trainers then have battles against each other where they use one Pokémon at a time from their team to try and defeat their opponent's team. Each face-off between two Pokémon is considered a "fight". A series of fights between two trainers is called a "battle". When a Pokémon loses a fight, it cannot be used again in the same battle. A battle is not over until one of the trainers has no more usable Pokémon, at which point their opponent wins.

Each monster is represented by an instance of the class `Pokemon`. You do not need to know the details of the class. To determine which Pokémon will win in a fight, the following function is used - it returns a negative number if *p1* wins, and a positive number if *p2* wins. There are no ties.

```
int pokemonFight(const Pokemon & p1, const Pokemon & p2);
```

In this problem you will be completing an implementation for the recursive function *TrainerOneWins()*. The function takes in two Node pointers which represent two lists of Pokémon, *trainer1* represents the Pokémon belonging to Trainer 1, and *trainer2* represents the Pokémon belonging to Trainer 2. The function returns `true` if Trainer 1 wins, and `false` if Trainer 2 wins. A trainer wins if they still have usable Pokémon but their opponent does not. If a trainer's Pokémon loses, the trainer will use the next Pokémon in their list.

In this problem, the Node class is defined as follows:

```
template <class T> class Node {
  T data;
  Node* next;
};
```

As an example consider the following case. You do not need to know anything about specific Pokémon to solve this problem.

*Node<Pokemon>* * *list1* has Bulbasuar, Ivysaur, Geodude

*Node<Pokemon>* * *list2* has Squirtle, Charmander

Running the following code:

```
if(TrainerOneWins(list1,list2)){
  std::cout << "Trainer 1 wins." << std::endl;
}
else{
  std::cout << "Trainer 2 wins." << std::endl;
}
```

A possible run using might look something like this. The output is handled in *pokemonFight()*, you should not write any output statements:

```
Bulbasuar wins against Squirtle
Charmander wins against Bulbasaur
Charmander wins against Ivysaur
Geodude wins against Charmander
Trainer 1 wins.
```

Assume that TrainerOneWins() is called with at least one non_empty list.

### 2.1 TrainerOneWins Implementation [ /12]

Fill in the blanks to finish *TrainerOneWins()*:

```
bool TrainerOneWins( [const Node<Pokemon>* &] trainer1, [const Node<Pokemon>* &] trainer2)
{
  if(trainer1 == NULL || trainer2 == NULL)
  {
    return [trainer2 == NULL];
  }

  int fight_result = pokemonFight( [trainer1->data, trainer2->data] );

  if(fight_result < 0){

    return [TrainerOneWins(trainer1, trainer2->next)];
  }
  else{

    return [TrainerOneWins(trainer1->next, trainer2)];

  }
}
```

### 2.2 TrainerOneWins Complexity [ /4]

If *pokemonFight()* is O(1), and there are m pokemon in Trainer 1's list and n pokemon in Trainer 2's list, what is the time complexity for *TrainerOneWins()*?

O(m+n)

## 3 Print-and-Modify Functions [ / 20]

In this problem you must infer the behavior of two functions from the code sample and output provided below. You should not hard-code any values. *Hint: Write out the relationship between the numbers before and after AddByPositionAndPrint().*

Running this code:

```
int main(){
  std::list<int> counts = std::list<int>(3,0);
  AddOneAndPrint(counts);
  AddOneAndPrint(counts);
  counts.push_back(9);
  counts.push_front(11);
  AddOneAndPrint(counts);
  std::list<int> add_amounts;
  add_amounts.push_back(4); add_amounts.push_back(1); add_amounts.push_back(-3);
  add_amounts.push_back(0); add_amounts.push_back(4);
  AddByPositionAndPrint(counts,add_amounts);
  AddOneAndPrint(add_amounts);
  return 0;
}
```

Produces this output, with one line coming from each non-STL function call:

```
Elements after updates: 1 1 1
Elements after updates: 2 2 2
Elements after updates: 12 3 3 3 10
Elements after updates: 16 4 0 3 14
Elements after updates: 5 2 -2 1 5
```

**Your answers should go in the boxes on the next page.**

### 3.1 AddByPositionAndPrint [ / 14]

Start by implementing *AddByPositionAndPrint*. Assume that the two arguments have the same size.

```
void AddByPositionAndPrint(const std::list<int>& a,
                           const std::list<int>& b) {
    std::cout << "Elements after update:";
    for (std::list<int>::const_iterator itr1 = a.begin(), itr2=b.begin();
         itr1 != a.end(); ++itr1, ++itr2) {
        std::cout << " " << *itr1 + *itr2;
    }
    std::cout << std::endl;
}
```

*sample solution: 10 line(s) of code*

### 3.2 AddOneAndPrint [ / 6] (AddByPositionAndPrint)

Now implement *AddOneAndPrint*. Do not write any duplicate code.

```
void AddOneAndPrint(const std::list<int>& list) {
    std::list<int> one(list.size(), 1);
    AddByPositionAndPrint(list, one);
}
```

*sample solution: 3 line(s) of code*

## 4 Mystery List Function [ / 20]

This problem focuses on a couple similar functions that are used to manipulate a collection of doubly linked nodes:

```
template <class T>
class Node{
public:
  Node(const T& v) : value(v), next(NULL), prev(NULL) {}
  T value;
  Node<T> *next, *prev;
};
```

We also define a function for printing nodes:

```
template <class T>
void PrintList(Node<T>* head){
  std::cout << "List:";
  while(head){
    std::cout << " " << head->value;
    head = head->next;
  }
  std::cout << std::endl;
}
```

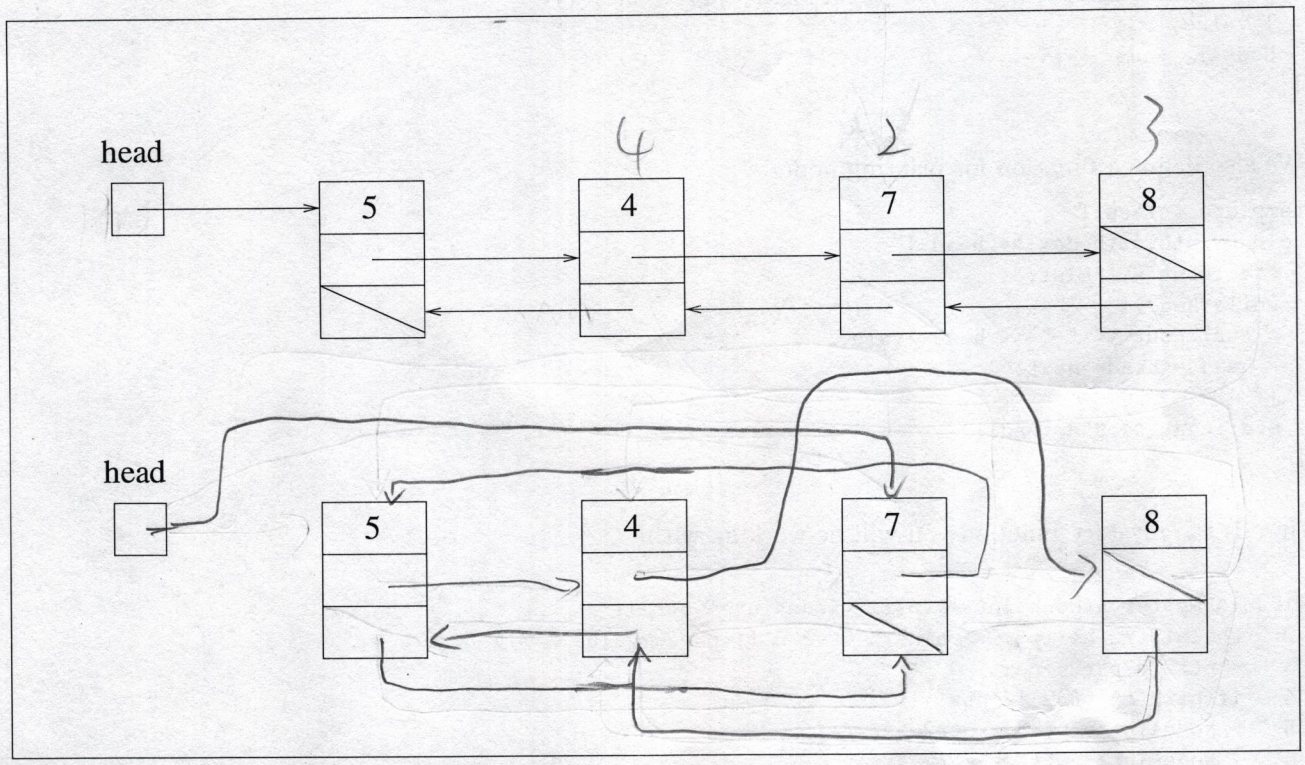
Here is the mystery function you will be working with:

```
 1 void MysteryA(Node<int>*& ptr1, Node<int>*& ptr2){
 2   while(ptr2 && (ptr2->value % 2 == 0 || ptr2->value % 7))
 3     ptr2 = ptr2->next;
 4   if(ptr2 && ptr2 != ptr1){
 5     Node<int>* ptr3 = ptr2->next;
 6     Node<int>* ptr4 = ptr2->prev;
 7     ptr2->next = ptr1;
 8     ptr1->prev = ptr2;
 9     ptr1 = ptr2;
10     if(ptr3)
11       ptr3->next = ptr4;
12     if(ptr4)
13       ptr4->prev = ptr3;
14     ptr1->prev = NULL;
15   }
16 }
```

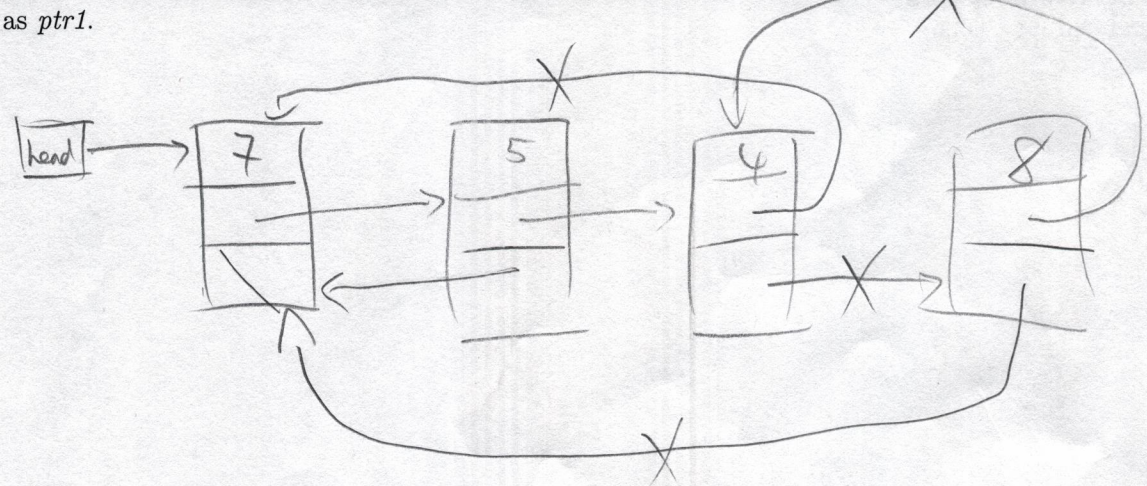
% 7 != 0

## 4.1 Visualizing *MysteryA* [ / 14]

In the box below, a list is shown. Below it is a copy of the nodes with none of the pointers added. Update the bottom list to illustrate how the list has changed after a call to *MysteryA(head,head)*.




The function is supposed to find the first odd multiple of 7 by using *ptr2* and if it's not already at the head, the function should make *ptr2* the new head by moving it to the front and updating links accordingly. It also has to update the pointer passed in. Assume that *ptr1* is a list head and that *ptr2* belongs to the same list as *ptr1*.



### 4.2 Fixing MysteryA [ / 6]

Consider the original list given in 4.1 and the following code fragment:

```
PrintList(head);
MysteryA(head,head);
PrintList(head);
```

Explain what will happen when this code runs. If there are bugs, explain which lines of code need to be changed and what the corrected lines of code should be. Use the line numbers provided in the left margin of the code.

First call of PrintList(head) would print list normally. However, after executing MysteryA(), the second PrintList() would be trapped in an infinite loop. Line 11 & 13's ptr3 -> next and ptr4 -> prev need to be changed. Correction is shown below:

11 ptr3 -> prev = ptr4;

13 ptr4 -> next = ptr3;

## 5 Acrostic Printing [ / 21]

An "acrostic" is a word that is found by a column reading top-to-bottom when looking at several words stacked on top of each other. For example, if our input is a vector of 4 strings:

```
Had
a
Nice
Day
```

then we could look at the first letter in each word to see that they spell a new word: "HaND". Similarly, we could look at the second letter in each word and see they spell: "aia". Your task is to write a function, *acrostics()*, which takes a vector of strings (each string contains exactly one word) and returns an array of C-style strings where each string is one column of the acrostic. For the input shown above, the return array should contain the following strings:

```
HaND
aia
dcy
e
```

You can ignore the null-terminating character, '\0' for simplicity. We have given you a partial implementation, you will need to fill in the rest of it. You should not use any arrays besides *ret*, and you should not call the `delete` keyword. You cannot declare any new vectors/lists in your code. Finally, keep in mind that you should be memory efficient and not allocate more space than you need in the return array.

**Your answer should go in the box on the next page.**

### 5.1 *acrostic* Implementation [ / 16]

```
char** acrostics(const std::vector<std::string>& v){
  unsigned int max_return_length = 0;

  for(unsigned int i=0; i<v.size(); i++){
    max_return_length = std::max(max_return_length, (unsigned int)v[i].size());
  }

  std::vector<int> characters_per_string(max_return_length,0);
  for(unsigned int i=0; i<v.size(); i++){
    for(unsigned int j=0; j<v[i].size(); j++){
      characters_per_string[j]++;
    }
  }
```

```
char ** ret = new char * [max_return_length];
for (unsigned int i=0; i<max_return_length; ++i) {
    ret[i] = new char [characters_per_string[i]];
}

for (unsigned int i=0; i<max_return_length; ++i) {
    for (unsigned int j=0; j<characters_per_string[i]; ++j) {
        for (unsigned int k=0; k<v.size(); ++k) {
            if (v[k].size >= i+1) ret[i][j] = v[k][i];
        }
    }
}
```

```
  return ret;
}
```

### 5.2 *acrostic* Performance [ / 5 ]

If there are $m$ input strings, the length of the longest input string is $n$, and there are $c$ characters in all input strings combined, what is the time complexity and the space (memory) complexity of *acrostic*?

$O(m \cdot n \cdot c)$

(blank page)

# ata structure cheat sheet
# Exam 1

Zhengneng Chen chenz11@rpi.edu

## Lecture 7 — Order Notation & Basic Recursion

• O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
• O($\log n$), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.
• O(n), a.k.a. LINEAR. e.g., sum up a list.
• O($n \log n$), e.g., sorting
• O($n^2$), O($n^3$),O($n^k$), a.k.a. POLYNOMIAL. e.g., find closest pair of points.
• O($2^n$),O($k^n$), a.k.a. EXPONENTIAL.e.g., Fibonacci, playing chess.
For a given fixed size array, we might want to know:
– The fewest number of operations (best case) that might occur.
– The average number of operations (average case) that will occur.
– The maximum number of operations (worst case) that can occur.

The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, iterative functions are generallyfaster than their corresponding recursive functions. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called tail call optimization.
1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use wishful thinking, i.e., if someone else solves the problem of fact(4) I can extend that solution to solve fact(5). This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

## Lecture 8 — Templated Classes & Vector Implementation

• In terms of the layout of the code in vec.h, the biggest difference is that this is a templated class. The keyword template and the template type name must appear before the class declaration: template <class T> class Vec
• Within the class declaration, T is used as a type and all member functions are said to be "templated over type T". In the actual text of the code files, templated member functions are often defined (written) inside the class declaration.
• The templated functions defined outside the template class declaration must be preceded by the phrase: template <class T> and then when Vec is referred to it must be as Vec<T>. For example, for member function create (two versions), we write: template <class T> void Vec<T>::create(...

•Templated classes and templated member functions are not created/compiled/instantiated until they are needed. Compilation of the class declaration is triggered by a line of the form: Vec<int> v1; with int replacing T. This also compiles the default constructor for Vec<int> because it is used here. Other member functions are not compiled unless they are used.
• When a different type is used with Vec, for example in the declaration: Vec<double> z; the template class declaration is compiled again, this time with double replacing T instead of int. Again, however, only the member functions used are compiled.
• This is very different from ordinary classes, which are usually compiled separately and all functions are compiled regardless of whether or not they are needed.
• The templated class declaration and the code for all used member functions must be provided where they are used. As a result, member functions definitions are often included within the class declaration or defined outside of the class declaration but still in the .h file. If member function definitions are placed in a separate .cpp file, this file must be #included, just like the .h file, because the compiler needs to see it in order to generate code. (Normally we don't #include .cpp files!) See also diagram on page 7 of this handout. Note: Including function definitions in the .h for ordinary non-templated classes may lead to compilation errors about functions being "multiply defined". Some of you have already seen these errors.

• For Vec (and other classes with dynamically-allocated memory) to work correctly, each object must do its own dynamic memory allocation and deallocation. We must be careful to keep the memory of each object instance separate from all others.
• All dynamically-allocated memory for an object should be released when the object is finished with it or when the object itself goes out of scope (through what's called a destructor).
• To prevent the creation and use of default versions of these operations, we must write our own:
– Copy constructor
– Assignment operator
– Destructor

• All class objects have a special pointer defined called this which simply points to the current class object, and it may not be changed.
• The expression *this is a reference to the class object.
• The this pointer is used in several ways:
– Make it clear when member variables of the current object are being used.
– Check to see when an assignment is self-referencing.
– Return a reference to the current object.

• This constructor must dynamically allocate any memory needed for the object being constructed, copy the contents of the memory of the passed object to this new memory, and set the values of the various member variables appropriately.

• Assignment operators of the form: v1 = v2; are translated by the compiler as: v1.operator=(v2);
• Cascaded assignment operators of the form: v1 = v2 = v3; are translated by the compiler as: v1.operator=(v2.operator=(v3));
• Therefore, the value of the assignment operator (v2 = v3) must be suitable for input to a second assignment operator. This in turn means the result of an assignment operator ought to be a reference to an object.
• The implementation of an assignment operator usually takes on the same form for every class:
– Do no real work if there is a self-assignment.
– Otherwise, destroy the contents of the current object then copy the passed object, just as done by the copy constructor. In fact, it often makes sense to write a private helper function used by both the copy constructor and the assignment operator.
– Return a reference to the (copied) current object, using the this pointer.

• The destructor is called implicitly when an automatically-allocated object goes out of scope or a dynamically-allocated object is deleted. It can never be called explicitly!
• The destructor is responsible for deleting the dynamic memory "owned" by the class.
• The syntax of the function definition is a bit weird. The has been used as a logic negation in other contexts.

## Lecture 9 — Iterators & STL Lists

• We have seen how push backadds a value to the end of a vector, increasing the size of the vector by 1. There is a corresponding function called pop back, which removes the last item in a vector, reducing the size by 1.
• There are also vector functions called front and back which denote (and thereby provide access to) the first and last item in the vector, allowing them to be changed.

• Here's the definition (from Koenig & Moo). An iterator:
– identifies a container and a specific element stored in the container,
– lets us examine (and change, except for const iterators) the value stored at that element of the container,
– provides operations for moving (the iterators) between elements in the container,
– restricts the available operations in ways that correspond to what the container can handle efficiently.
• As we will see, iterators for different container classes have many operations in common. This often makes the switch between containers fairly straightforward from the programer's viewpoint.
• Iterators in many ways are generalizations of pointers: many operators / operations defined for pointers are defined for iterators. You should use

this to guide your beginning understanding and use of iterators.
• The dereference operator is combined with dot operator for accessing the member variables and member functions of elements stored in containers. Here's an example (*i).compute_averages(0.45);
Notes:
– This operation would be illegal if i had been defined as a const iterator because compute_averages is a non-const member function.
– The parentheses on the *i are required (because of operator precedence).
• Just like pointers, iterators can be incremented and decremented using the ++ and – operators to move to the next or previous element of any container.
• Iterators can be compared using the == and != operators.
• Iterators can be assigned, just like any other variable.
• Vector iterators have several additional operations:
– Integer values may be added to them or subtracted from them. This leads to statements like enrolled.erase(enrolled.begin() + 5);
– Vector iterators may be compared using operators like <, <=, etc.
– For most containers (other than vectors), these "random access" iterator operations are not legal and therefore prevented by the compiler. The reasons will become clear as we look at their implementations.

• Although the interface (functions called) of lists and vectors and their iterators are quite similar, their implementations are VERY different. Clues to these differences can be seen in the operations that are NOT in common, such as:
– STL vectors / arrays allow "random-access" / indexing / [] subscripting. We can immediately jump to an arbitrary location within the vector / array.
– STL lists have no subscripting operation (we can't use [] to access data). The only way to get to the middle of a list is to follow pointers one link at a time.
– Lists have push front and pop front functions in addition to the push back and pop back functions of vectors.
– erase and insert in the middle of the STL list is very efficient, independent of the size of the list. Both are implemented by rearranging pointers between the small blocks of memeory. (We'll see this when we discuss the implementation details next week).
– We can't use the same STL sort function we used for vector; we must use a special sort function defined by the STL list type. std::vector<int> my_vec; std::list<int> my_lst; // ... put some data in my_vec & my_lst std::sort(my_vec.begin(),my_vec.end(),optional_compare_function); my_lst.sort(optional_compare_function);
Note: STL list sort member function is just as efficient, O($n \log n$), and will also take the same optional compare function as STL vector.
– Several operations invalidate the values of vector iterators, but not list iterators: erase invalidates all iterators after the point of erasure in vectors; push back and resize invalidate ALL iterators in a vector The value of any associated vector iterator must be re-assigned / re-initialized after these operations.

## Lecture 10 — Vector Iterators & Linked Lists

• Goal: place a new node at the end of the list.
• We must step to the end of the linked list, remembering the pointer to the last node.
– This is an O(n) operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
• We must create a new node and attach it to the end.
• We must remember to update the head pointer variable's value if the linked list is initially empty.
– Hence, in writing the function, we must pass the pointer variable by reference.

## Lecture 11 — Doubly-Linked Lists

Here is a summary of common mistakes. Read these carefully, and read them again when you have problem that you need to solve.
• Allocating a new node to step through the linked list; only a pointer variable is needed.
• Confusing the . and the -> operators.
• Not setting the pointer from the last node to NULL.
• Not considering special cases of inserting / removing at the beginning or the end of the linked list.
• Applying the delete operator to a node (calling the operator on a pointer to the node) before it is appropriately disconnected from the list. Delete should be done after all pointer manipulations are completed.
• Pointer manipulations that are out of order. These can ruin the structure of the linked list.

• Trying to use STL iterators to visit elements of a "home made" linked chain of nodes. (And the reverse... trying to use ->next and ->prev with STL list iterators.)

Zhengfeng Chen chenz11@rpi.edu

• We can only move through it in one direction
• We need a pointer to the node before the spot where we want to insert and a pointer to the node before the node that needs to be deleted.
• Appending a value at the end requires that we step through the entire list to reach the end.

• Three common generalizations (can be used separately or in combination):
– Doubly-linked: allows forward and backward movement through the nodes
– Circularly linked: simplifies access to the tail, when doubly-linked
– Dummy header node: simplifies special-case checks
• Today we will explore and implement a doubly-linked structure.