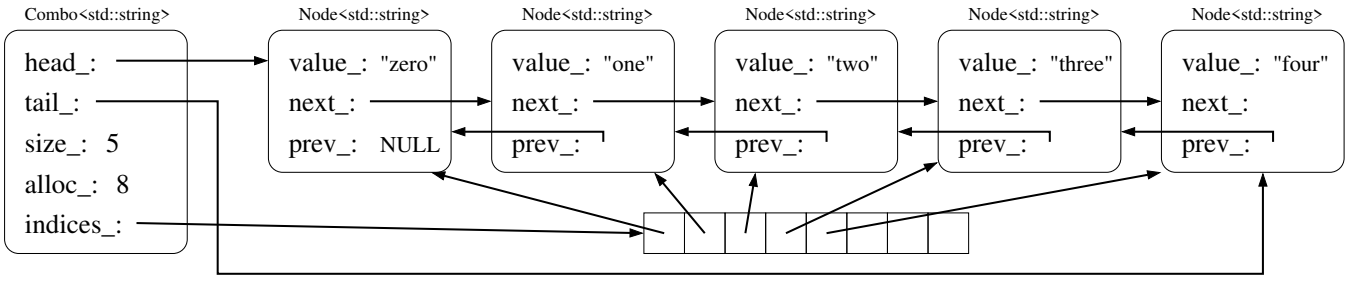
CSCI-1200 Data Structures
Test 2 — Practice Problems

Note: This packet contains selected practice problems from Test 2 from three previous years.
Your test will contain approximately one third to one half as many problems (totalling ∼100 pts).

1 The Combo (`list` + `vector`) Data Structure [26 pts]

After working with the STL `list` data structure, Ben Bitdiddle misses the subscript operator, `[]`, and decides to build a hybrid `Combo` data structure combining the features he likes from both STL `list` and STL `vector`. He sketched the following diagram showing a sequence of five STL `string`s:



1.1 Member Variables of Combo Data Structure [4 pts]

First, complete the data structure representation / member variable section for the `Node` and `Combo` classes as they would appear in the class declaration:

```
template <class T> class Node {
public:
  /* NOTE: CONSTRUCTORS OMITTED */
```

sample solution: 3 lines of code

```
};

template <class T> class Combo {
public:
  /* NOTE: CONSTRUCTORS, ACCESSORS, MODIFIERS, ETC. OMITTED */
private:
```

sample solution: 5 lines of code

```
};
```

1.2 Write `operator[]` for `Combo` Data Structure [3 pts]

Next implement the subscript `operator[]` member function for the `Combo` class as it would appear when implemented inside the `Combo` class declaration.

sample solution: 1 lines of code

1.3 Copy Constructor for `Combo` Data Structure [15 pts]

Now let's implement the copy constructor for the `Combo` data structure. Since this function is more than 1 line of code, we will implement it outside of the class declaration. *NOTE: Don't write or use any helper functions in your implementation.*

sample solution: 21 lines of code

1.4 Disadvantages of `Combo` Data Structure [4 pts]

Unfortunately the `Combo` Data Structure is flawed. Write 2-3 sentences describing one specific disadvantage of this structure compared to STL `vector` and one specific disadvantage when compared to STL `list`.

2 Nested List Sentence [25 pts]

In this problem you will work with a sentence that is represented by an STL `list` of words, where each word is a `list` of `char`. There are no spaces or punctuation in the words or sentence data structure.

2.1 Write `print_sentence` [7 pts]

First, write a function named `print_sentence` that takes in a single argument, `input`, with type described above, and legibly prints the contents of the argument to `std::cout`.

NOTE: Sample output of the `print_sentence` *function is shown below.*

sample solution: 11 lines of code

2.2 Write a `silly` sentence editor [12 pts]

On the next page, we'll write a function named `silly` that takes in the same sentence representation and searches for two (or more) repeated adjacent letters within each word and reduces the repeated adjacent letters to a single instance of that letter. If the word has no repeated adjacent letters it is removed from the sentence. For example, the input sentence:

```
a velvet tress llama did hiss at the aardvark bookkeepers balloon trees crosssection
```

Will be changed to:

```
tres lama his ardvark bokepers balon tres crosection
```

sample solution: 24 lines of code

2.3 Big O Notation [6 pts]

Let's say the input sentence has n words, the longest word has k letters, at most a letters are removed from each word, and r words are removed from the sentence because they have no repeated adjacent letters.

What is the Big O Notation of `print_sentence`?

What is the Big O Notation of `silly`?

If we search and replace STL `list` with STL `vector` in your solution above, will your code still work? What is the Big O Notation of the `vector` version of `silly`? Write 1-2 sentences justifying your answer.

3 Recursive Reach [18 pts]

Let's write a *recursive* function named `recursive_reach` that takes in 4 arguments: `board`, `row`, `col`, and `pathlength`. The `board` is an STL `vector` of STL `string`s that represents a rectangular grid. Some of the locations are marked with the '`#`' symbol, which are "walls" that cannot be crossed. All other positions in the grid are the '`.`' character. The function will *paint* locations that are reachable from the starting point (`row`,`col`) with a path of length less than or equal to `pathlength`. The function paints/edits the `board` with the '`o`' symbol at each location that is reachable. The path length is measured by moving up, down, left, and right. Moving diagonally require 2 moves: horizontal and vertical.

Consider the input board shown below left. We show the output of the program for the same starting location (row=2, col=1) for a variety of different values of `pathlength`.

input	pathlength=1	pathlength=2	pathlength=4	pathlength=6
`...#.....`	`...#.....`	`.o.#.....`	`ooo#.....`	`ooo#oo...`
`...#.....`	`.o.#.....`	`ooo#.....`	`ooo#o....`	`ooo#ooo..`
`.........`	`ooo......`	`oooo.....`	`oooooo...`	`oooooooo.`
`.#####...`	`.#####...`	`o#####...`	`o#####...`	`o#####o..`
`.........`	`.........`	`.........`	`oo.......`	`oooo.....`

3.1 Playing with Examples [6 pts]

For an input board of width w and height h where n grid locations are blocked by a wall, what is the *worst case* longest path length that will be necessary to reach every grid location? (Assuming every location is reachable and not completely blocked by the walls.) Draw a sketch of a worst case input board and neatly label the start location and the longest path. What is the Big O Notation for the path length in terms of w, h, and n? Optionally write 1-2 sentences justifying your answer.

What is the Big O Notation of your code (on the next page)? Write 1-2 sentences justifying your answer.

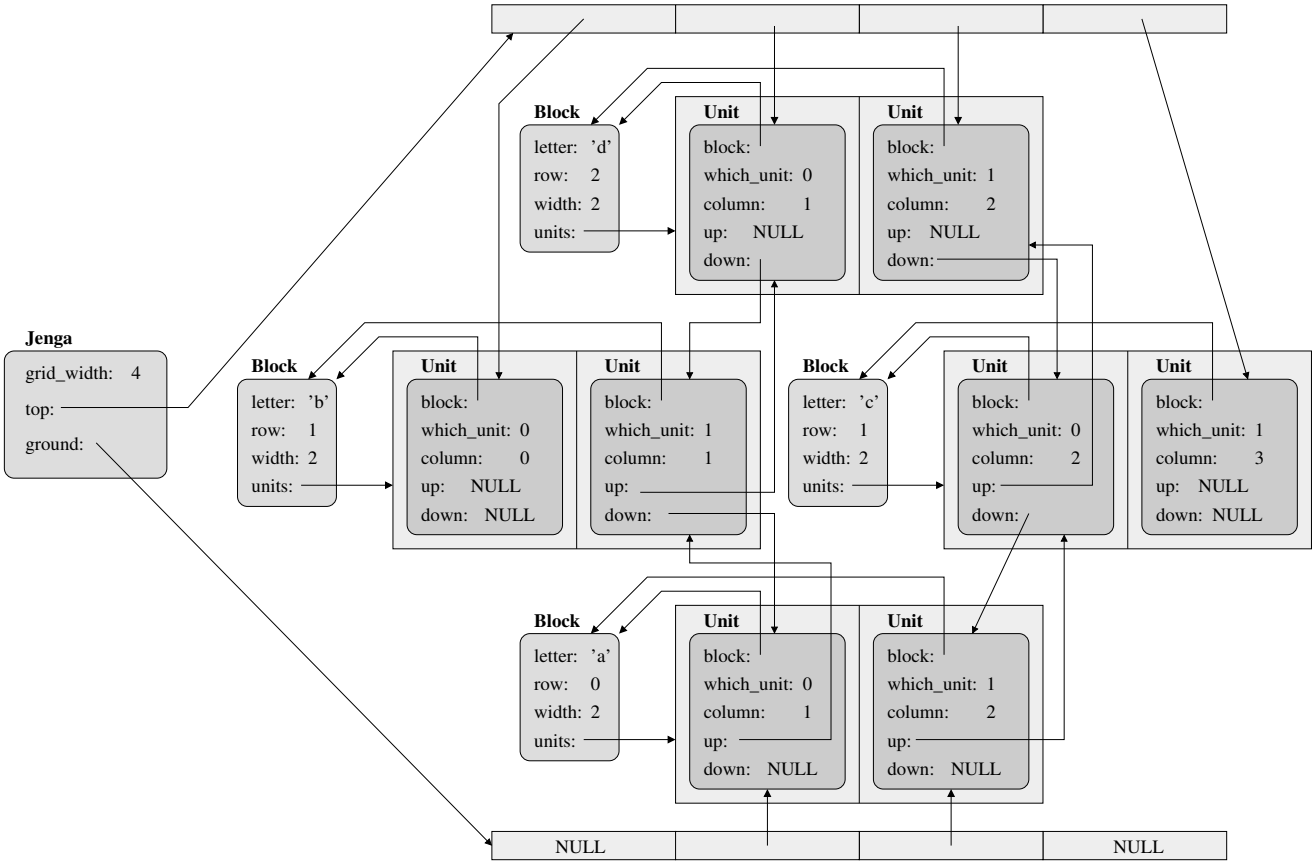
3.2 Painting Recursively [12 pts]

Implement `recursive_reach`.

sample solution: 11 lines of code

4 Jenga Block Merge [28 pts]

Let's implement a *block merge* operation for our Jenga Game Simulation from Homework 5. Below is a diagram of the Jenga data structure showing a tower with four blocks. The merge operation will allow us to "glue together" two blocks in the tower that are on the same row and touching. For example, we could merge blocks `b` and `c` in row 1. After the merge the tower would have the same shape, but would only contain three blocks. Block `b` would now be four units in width, and block `c` would not exist anymore.



Here are the member variables for each class:
NOTE: A couple unnecessary variables from the Jenga class have been omitted.

Unit

```
Unit* up;
Unit* down;
Block* block;
int which_unit;
int column;
```

Block

```
int width;
Unit* units;
int row;
char letter;
```

Jenga

```
int grid_width;
Unit** top;
Unit** ground;
```

4.1 Write Jenga::MergeBlocks [28 pts]

On the next page, write the `Jenga` class `MergeBlocks` member function that takes 2 `Block` pointers and returns true if the blocks were successfully merged. Your function should return false if the blocks are on different rows, or if the blocks are not touching. Also, the second block should be positioned to the right of the first block. For this problem, you can assume all member variables are public (you don't need to use getter / accessor functions).

sample solution: 46 lines of code

5 Mock Interview Practice [28 pts]

Ben Bitdiddle is prepping for a coding interview and asks his Data Structures mentor Jenay for help. *Please read through the entire question before working on any of the subproblems.* Jenay suggests Ben tackle this problem from lecture: "`Write code to remove duplicates from a sequence of words.`"

Ben writes a function `remove_dups_1` that takes in an STL `vector` of STL `string`s and returns the number of words that were removed. For this input:

```
the quick brown fox jumped over the lazy brown dogs and also jumped over the lazy penguin
```

Calling Ben's first draft function returns '6' and the `vector` now contains:

```
also and brown dogs fox jumped lazy over penguin quick the
```

Jenay observes that while sorting the data makes the program run fast, unfortunately the likely intention was to preserve the order within the original data.

5.1 Quick but Flawed [14 pts]

Write code that matches the description of Ben's `remove_dups_1` function:

sample solution: 19 lines of code

If the input has n words and r words are removed, what is the Big O Notation of `remove_dups_1`?

5.2 Preserving the Sequence [14 pts]

For his second draft, `remove_dups_2`, Ben starts over from scratch. The function has the same prototype, but based on Jenay's feedback it now preserves the original order of the data. So for this input:

```
the quick brown fox jumped over the lazy brown dogs and also jumped over the lazy penguin
```

The function returns 6 and the `vector` now contains:

```
the quick brown fox jumped over lazy dogs and also penguin
```

Jenay's feedback about `remove_dups_2` is that he's using the `erase` function, which will negatively impact the performance of this code. Write code that matches the description of Ben's `remove_dups_2` function:

sample solution: 16 lines of code

If the input has n words and r words are removed, what is the Big O Notation of `remove_dups_2`?

Ben copies his code for `remove_dups_2` to a new function. The only difference for `remove_dups_3` is that he search-and-replaces '`vector`' with another STL container. The program produces the same answer but now runs faster. What's the replacement container? What is the Big O Notation for `remove_dups_3`?

6 Clown Car Data Structures [25 pts]

Write a function named `clowncar` that takes in two arguments, one of type `Vec` named `a` and the other of type `dslist` named `b`. *In lecture and lab we talked about the implementations of these two "homemade" versions of our favorite STL containers.* The function should swap the data stored in these two structures so that after the call `a` contains the sequence of values that was in `b` and `b` contains the data that was in `a`.

The `clowncar` function has been appropriately added as a friend function of both `Vec` and `dslist` so that it can directly access the private member variables of both classes. Your implementation of `clowncar` **SHOULD NOT call any other functions (including member functions of Vec or dslist) and it SHOULD NOT use iterators**. We want to see you directly edit the member variables and work with the dynamic memory. As a reminder, here are the private member variables of the relevant classes:

Vec

```
T* m_data;
size_type m_size;
size_type m_alloc;
```

Node

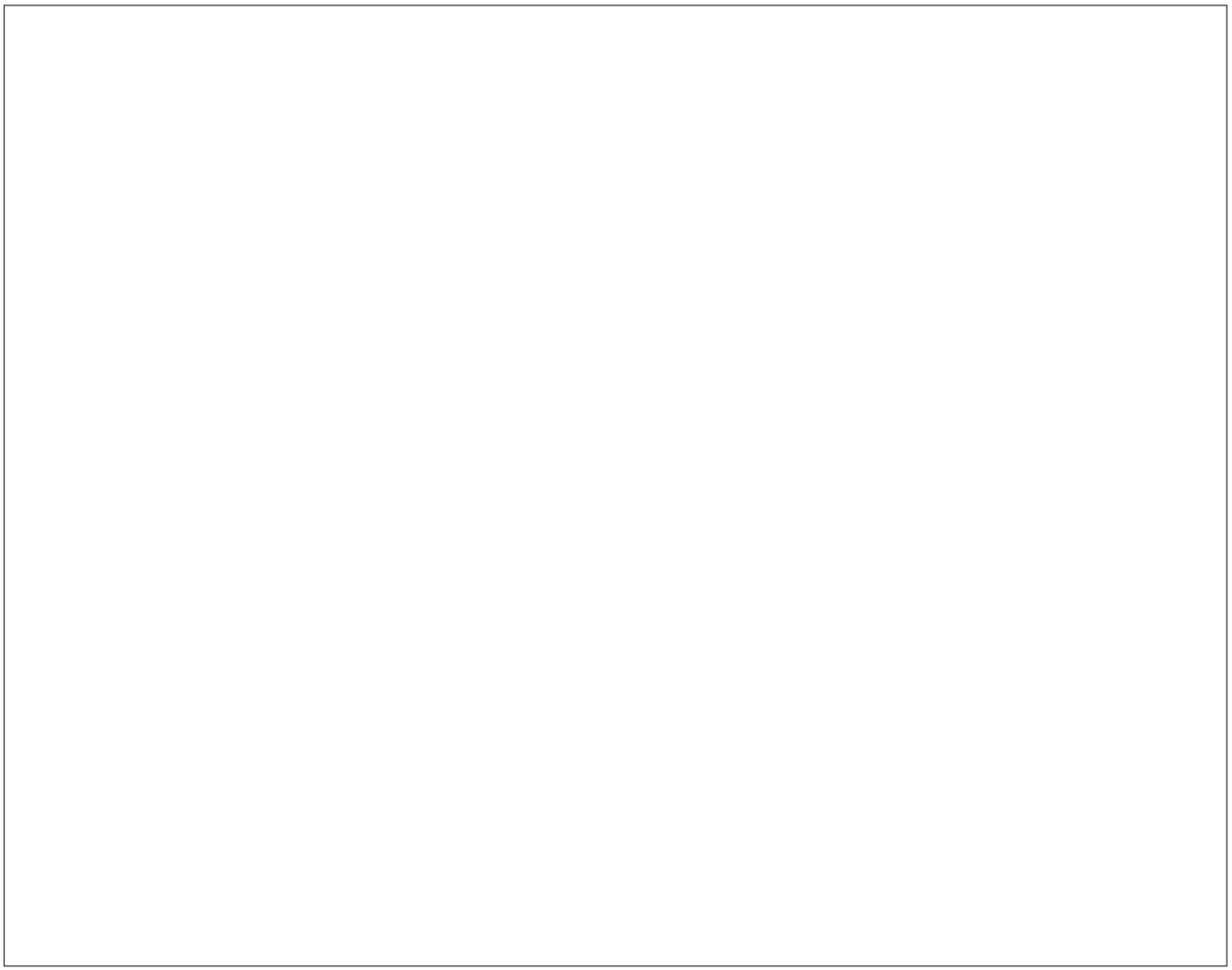
```
T value_;
Node<T>* next_;
Node<T>* prev_;
```

dslist

```
Node<T>* head_;
Node<T>* tail_;
unsigned int size_;
```

6.1 Drawing [9 pts]

First, make a detailed memory diagram of sample input to the function: `a` stores 2 even integers (6 & 8) and `b` stores 3 odd integers (15, 17, & 19). Next, neatly edit this diagram to show what will happen when you call `clowncar`. Instead of erasing, lightly cross out things that are changed (allowing us to legibly grade the diagram both before & after the call). Your diagram should match the code you write on the next page. Be sure to include any temporary variables, and all allocations and deallocations of memory.



6.2 Implementation [16 pts]

Now implement the `clowncar` function. Ensure your function does not have any memory errors or leaks.

sample solution: 29 lines of code

7 "Missing" `dslist` Iterator Operators [14 pts]

Louis B. Reasoner is working on a group project and his teammates are upset that the project code below doesn't compile. They claim something must be wrong with `dslist`!

```
dslist<std::string>::iterator itr = sentence.begin() + 1;
dslist<std::string>::iterator itr2 = itr + 4;
assert (!(itr2 < itr));
while (itr < itr2) {
  std::cout << *itr << " ";
  ++itr;
}
std::cout << std::endl;
```

Louis tries to explain that `dslist` is fine. That this code wouldn't work even if they switched to the STL `list` class. He suggests they modify the lines that do not compile to use these functions:

```
list_iterator<T>& operator++() { ptr_ = ptr_->next_; return *this;  }
bool operator!=(const list_iterator<T>& r) const { return ptr_ != r.ptr_; }
```

Unfortunately, Louis is unable to convince his teammates to change the project code, and with the deadline fast approaching Louis instead modifies the `dslist` implementation to make the project code above work. What two operator member functions does Louis add to the `list_iterator` class? Write these two functions as they would appear in the class declaration. *Note: You may break the course rule discouraging multiple line functions inside the class declaration.*

sample solution: 16 lines of code

8 Debugging Skillz [/ 14]

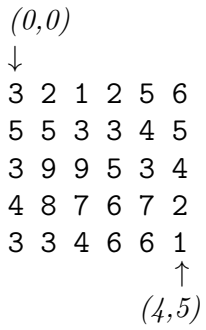
For each program bug description below, write the letter of the most appropriate debugging skill to use to solve the problem. Each letter should be used at most once.

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 3-4 well-written sentences describing of a specific situation where this debugging skill would be useful. You are encouraged to describe a personal anecdote.

9 It's all Downhill from Here! [16 pts]



Write a *recursive* function named `downhill` that takes in 4 arguments: `grid`, `start`, `end`, and `path`. It searches the 2D `grid` of elevations, an STL `vector` of STL `vector`s of integers, for a path from a `start` location to an `end` location. Each step along the path can go up, down, left, or right, but each step must have a lower elevation value than the current position. If it finds a valid downhill path from start to end, the function stores the path (an STL `vector` of locations) and returns true, otherwise it returns false.

```
// A Location on the grid
class loc {
public:
  loc(int r, int c) :
    row(r), col(c) {}
  int row;
  int col;
};
```

For the example shown above right, if start is (3,1) and end is (0,2), then this is a valid downhill path:

```
(3,1) (3,2) (3,3) (2,3) (1,3) (0,3) (0,2)
```

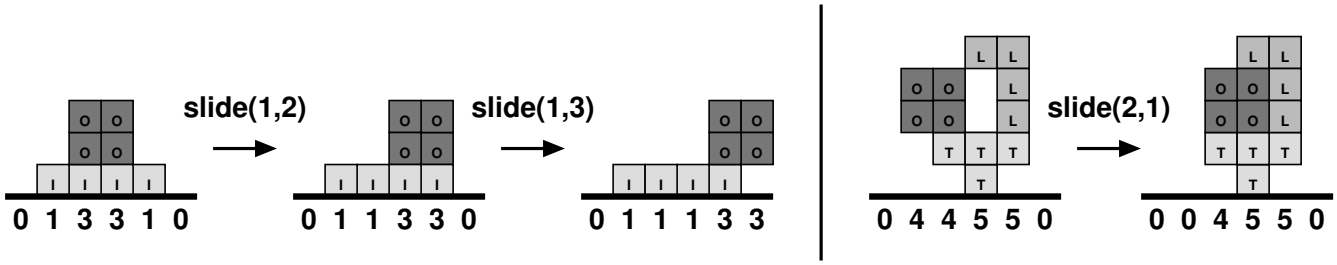
Note: There may be multiple valid downhill paths from start to end, and your function may choose any of these valid paths.

sample solution: 16 lines of code

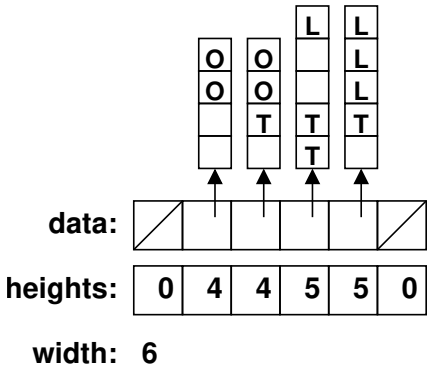
10 The Dynamic Tetris Slide [35 pts]

Our implementation of the Tetris game for Homework 3 only allowed pieces to drop vertically. The full game rules also allow pieces to move horizontally, which can be used by a skilled player to tuck in underneath an "overhang". In this problem we will extend our solution with the `slide` function that allows the square piece, the '`O`' piece, to *slide* one space to the right. *For this problem you don't need to worry about sliding any other piece shape, or about sliding to the left.*

Below are two example Tetris games showing how this function works.



The representation for the `Tetris` class consists of 3 private member variables: `data`, `heights`, and `width`. The memory layout for the 4th diagram above is shown to the right. Remember that we must maintain the arrays to be exactly as long as necessary to store the blocks on the board. The space character is used to represent empty air underneath a block.



The `slide` function takes in 2 integers, the row and column of the lower left corner block of the square '`O`' piece that we want to slide.

We will also implement the `can_slide` function which first tests whether a piece is able to slide to the right. It will return *false* if the 'O' piece at the specified row and column is already at the right edge of the board, e.g., calling `can_slide(1,4)` in the third image above returns false. It will return *false* if the 'O' piece at the specified row and column is blocked by another piece on the board, e.g., calling `can_slide(2,2)` in the 5th image above will return false.

10.1 Algorithm Analysis [5 pts]

Assume that the game board has width w, the height of the tallest column is h, and the number of blocks (total number of piece characters and space characters) is b. What is the Big O Notation for the running time of your `can_slide` and `slide` functions that you have implemented on the next two pages? Write two to three concise and well-written sentences justifying your answers.

```
can_slide:


slide:
```

10.2 `can_slide` Implementation [12 pts]

```
bool Tetris::can_slide(int row, int column) const {

   // First, let's do some error checking on input arguments
   // and the current board state.  This will help when we need
   // to debug this new function.  Write if/else statements
   // and/or assertions to verify your assumptions.
```

sample solution: 8 lines of code

```
// Now, we can do the logic necessary to determine whether this piece
// can slide to the right.
```

sample solution: 6 lines of code

```
}
```

10.3 `slide` Implementation [18 pts]

```
void Tetris::slide(int row, int column) {
    assert (can_slide(row,column) == true);
```

sample solution: 26 lines of code

```
}
```

11 Lightning Round [13 pts]

```
std::vector<std::string> a;
std::list<std::string> b;
// omitted: initialize both containers to hold n = a large number of  words

01  a.push_front("apple");
02  b.push_front("banana");
03  a.push_back("carrot");
04  b.push_back("date");
05  std::vector<std::string>::iterator itr_a = a.begin();
06  std::list<std::string>::iterator itr_b = b.begin();
07  itr_a = a.insert(itr_a,"eggplant");
08  itr_a += 5;
09  itr_a = a.erase(itr_a);
10  itr_b += 5;
11  itr_b = b.insert(itr_b,"eggplant");
12  ++itr_b;
13  itr_b = b.erase(itr_b);
14  a.sort();
15  b.sort();
16  std::sort(a.begin(),a.end());
17  std::sort(b.begin(),b.end());
```

Which lines result in a compilation error?

Which lines cause a segmentation fault?

Which lines have a memory leak?

Which lines run in $O(1)$ time?

Which lines run in $O(n)$ time?

Which lines run in $O(n \log n)$ time?

Which lines run in $O(n^2)$ time?

12 Button Up the Linked Grid [26 pts]

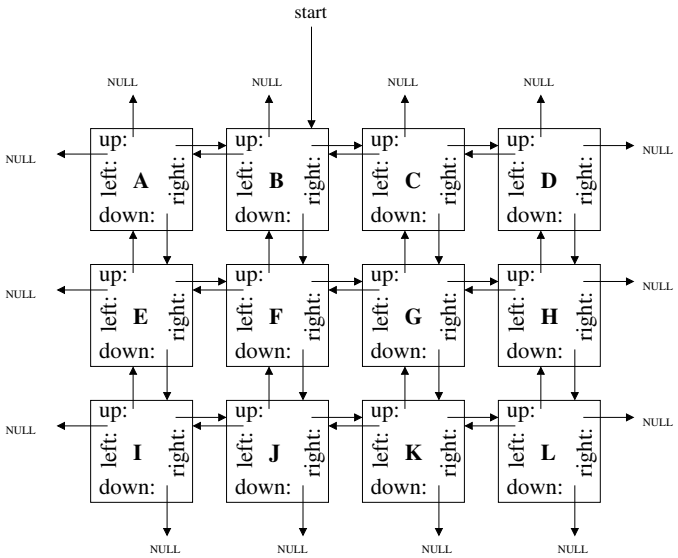
Alyssa P. Hacker and Ben Bitdiddle are working on a team project based on the linked grid of `Node`s data structure from Homework 5. Alyssa suggests they start with the `print_perimeter` function, which takes in a pointer to a `Node` named `start`, and walks around the edge of the grid in a *clockwise direction*. The function should print the value stored in every `Node` visited.

For example, `print_perimeter(start)` for the diagram shown on the right will print this sequence of values to the screen:

```
B C D H L K J I E A
```

Alyssa says it's ok to assume that the grid is at least two rows tall and at least two columns wide and that `start` definitely points to a `Node` somewhere on the edge/perimeter of the grid.

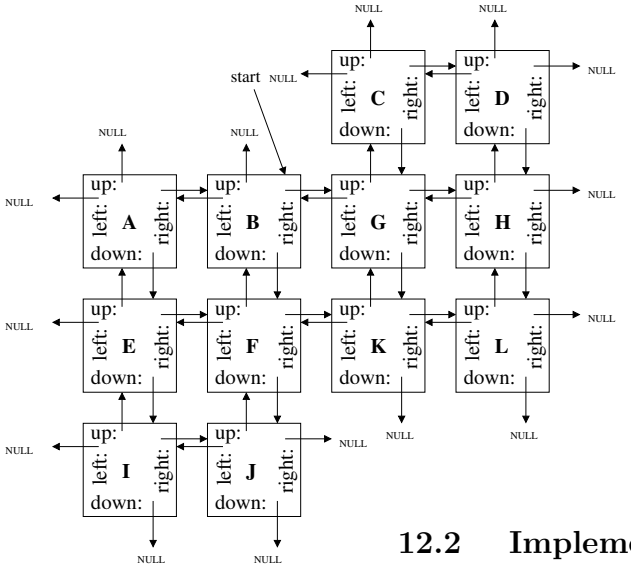
```
template <class T> class Node {
public:
  T value;
  Node<T> *up,*down,*left,*right;
};
```

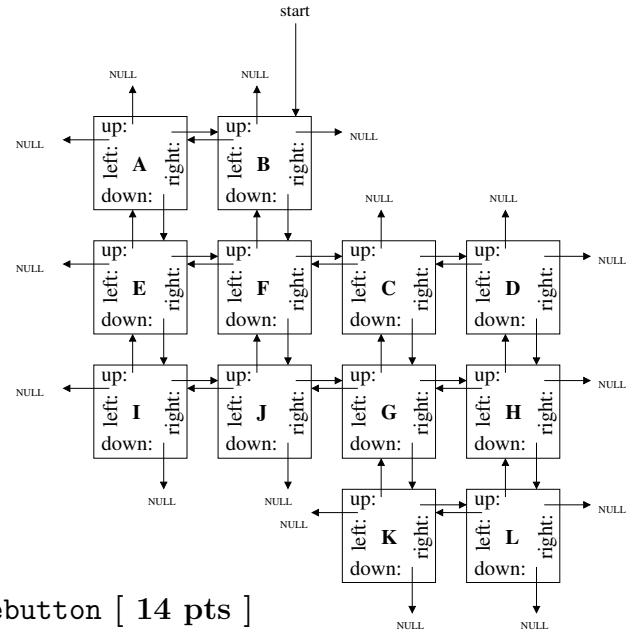


12.1 Implement `print_perimeter` [12 pts]

sample solution: 16 lines of code

Meanwhile, Ben is working on a function named `rebutton`, which takes in 2 arguments: `start`, a pointer to a `Node` on the top edge of the grid and a bool `shift_up`. The function makes a vertical cut to the right of `start` and reconnects the `Node`s on either side of the cut shifted up (below left) or shifted down (below right) one row. Ben claims that calling `rebutton(start,true)` followed by `rebutton(start,false)` will restore the original grid. And vice versa.

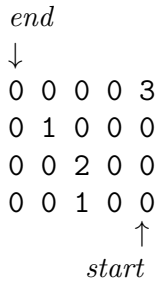




12.2 Implement `rebutton` [14 pts]

sample solution: 28 lines of code

13 Recursive Maximum Coin Path [23 pts]



Write a *recursive* function named `max_coin_path` that searches a 2D `grid` of "coins", an STL `vector` of STL `vector` of non-negative integers, for a path back to the origin (0,0). In walking from the start location (lower right corner of grid) to the origin (upper left corner), the path is only allowed to move up or left one grid space at a time. The goal is to find a path that *maximizes* the sum of the coins along the path. The function should return the maximum sum.

```
class Location {
public:
  Location(int r, int c)
    : row(r),col(c) {}
  int row;
  int col;
};
```

For the example shown above right, the path
`(3,4) (3,3) (3,2) (2,2) (2,1) (1,1) (1,0) (0,0)`
collects coins with values $1 + 2 + 1 = 4$, which is the maximum coin sum that can be achieved on this grid. The path achieving that sum should be stored in the second argument passed to the function, an STL `list` of `Location`s named `path`. Note: there are a few similar paths that have the same sum. Your function may return any of these optimal paths.

13.1 Usage [2 pts]

You will implement the `max_coin_path` on the next page. But first, complete the initial call to the `max_coin_path` function below. Assume `grid` has already been initialized; for example, with the data shown above. What additional information does your function need to get started?

```
std::list<Location> path;
int max_coin_sum = max_coin_path(grid,path,                    );
```

13.2 Algorithm Analysis [5 pts]

Assume that the grid width and height are w and h respectively, the number of non-zero coins in the grid is c, and the value of the maximum coin is m. What is the Big O Notation for the running time of your answer on the next page? Write three to four concise and well-written sentences justifying your answer.

13.3 Implementation [16 pts]

Now implement the `max_coin_path` function. Remember: it should be *recursive*.

sample solution: 27 lines of code