1 Boxy Storage Solutions [ / 25 ]

Eva Lu Ator is working on her capstone project to manage physical storage facilities. She’s mapped out the overall design and started implementation of the two classes.

```cpp
class Box {
public:
    Box(int w, int d, int h) :
        width(w), depth(d), height(h) {}
    int width;
    int depth;
    int height;
};
```

```cpp
class Storage {
public:
    Storage(int w, int d, int h);
    // FILL IN FOR PART 1
    bool add(Box *b, int w, int d, int h);
    int available_space();
private:
    void remove(Box *b, int w, int d, int h);
    Box ***data;
    int width;
    int depth;
    int height;
};
```

```cpp
bool Storage::add (Box *b, int w, int d, int h) {
    for (int i = w; i < w+b->width; i++) {
        if (i >= width) return false;
        for (int j = d; j < d+b->depth; j++) {
            if (j >= depth) return false;
            for (int k = h; k < h+b->height; k++) {
                if (k >= height) return false;
                if (data[i][j][k] != NULL) return false;
            }
        }
    }
    return true;
}
```

```cpp
Storage storage(4,3,2);
assert (storage.available_space() == 24);
Box *a = new Box(2,2,2);
assert (storage.add(a,0,0,0));
Box *b = new Box(3,2,1);
assert (!storage.add(b,2,0,0));
delete b;
Box *b_rotated = new Box(2,3,1);
assert (storage.add(b_rotated,2,0,0));
Box *c = new Box(1,1,1);
assert (storage.add(c,2,0,1));
assert (storage.available_space() == 9);
```

1.1 Missing functions from Storage Class Declaration [ / 5 ]

Her friend Ben Bitdiddle doesn’t remember much from Data Structures, but he reminds her that classes with dynamically-allocated memory need a few key functions. Fill in the missing prototypes for PART 1.

Solution:

```cpp
Storage(const Storage &s);
Storage& operator=(const Storage &s);
~Storage();
```

Note: The convention for C/C++ is that the assignment operator returns an object. It should return this object rather than the right-hand operand, s. It should return a reference (to avoid unneccessary copying). Whether it returns a const or a non-const is debateable. Whoever called this function must have had write access to the this object, so returning a non-const reference is safe.
1.2 Storage Destructor [  / 20 ]

Eva explains to Ben that the private remove member function will be useful in implementing the destructor. First write the remove member function:

Solution:

```cpp
void Storage::remove(Box *b, int w, int d, int h) {
    for (int i = w; i < w+b->width; i++) {
        for (int j = d; j < d+b->depth; j++) {
            for (int k = h; k < h+b->height; k++) {
                assert (data[i][j][k] == b);
                data[i][j][k] = NULL;
            }
        }
    }
    delete b;
}
```

Now write the Storage class destructor:

Solution:

```cpp
Storage::~Storage() {
    for (int w = 0; w < width; w++) {
        for (int d = 0; d < depth; d++) {
            for (int h = 0; h < height; h++) {
                if (data[w][d][h] != NULL) {
                    remove(data[w][d][h],w,d,h);
                }
            }
            delete [] data[w][d];
        }
        delete [] data[w];
    }
    delete [] data;
}
```

2 Transpose Linked Grid [  / 27 ]

Louis B. Reasoner is working on a new member function for our Homework 5 Linked Grid named transpose. This function should mirror or flip the elements along the diagonal. Here’s a sample grid with integer data and how it prints before and after a call to transpose:

```cpp
grid.print();
std::cout << std::endl;
grid.transpose();
grid.print();
```

```cpp
template <class T>
class Node {
    T value;
    Node<T> *up;
    Node<T> *down;
    Node<T> *left;
    Node<T> *right;
};
```
2.1 Diagram [ / 7 ]

First neatly modify the diagram of this smaller grid below to show all of the necessary edits that must be performed by a call to transpose().

Solution:

2.2 Complexity Analysis [ / 5 ]

What is the Big 'O' Notation for the running time of the transpose() member function? Assume the grid width is \( w \) and the height is \( h \). Write 1-2 concise and well-written sentences justifying your answer. You probably want to complete the implementation on the next page before answering.

Solution: We need to update a few variables in the Grid manager class, and we need to visit every node in the structure and modify the links. If we do things in an organized manner we can do so with a small (constant) number of helper variables, and it does not require expensive logic. Number of nodes → Overall: \( O(w \times h) \).

2.3 Implementation [ / 15 ]

Louis has suggested that we first implement a helper non-member function named swap, which will make the implementation of transpose more concise.

Solution:

```cpp
template <class T> void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Now implement transpose, as it would appear outside of the Grid class declaration.

Solution:

```cpp
template <class T> void Grid<T>::transpose() {
    Node<T>* row = upper_left;
    while (row != NULL) {
        Node<T>* next_row = row->down;
        Node<T>* element = row;
        while (element != NULL) {
            Node<T>* next_element = element->right;
            swap(element->up, element->left);
            swap(element->right, element->down);
            element = next_element;
        }
        row = next_row;
    }
    swap(width, height);
    swap(upper_right, lower_left);
}
```
3 Organizing Words [ / 30 ]

Alyssa P. Hacker is working on a program to clean up a dataset of words. The task is to write a function named `organize_words` that takes in an STL `vector` of STL `list`s of words (STL `strings`). The function should organize the words into groups by word length, and ensure that the words are sorted within each group. Many or most of the words will already be in the right place. That is, they will already be in the slot of the vector that matches the length of the word. And the neighboring words in each slot/list will already be mostly alphabetized.

For example, given the data shown on the left, your implementation should move the four misplaced words to produce the data shown on the right.

```
0                      0
1 diamond              1
2                      2
3 gem malachite        3 gem
4 jade opal rock ruby  4 jade opal rock ruby talc
5 geode pearl talc stone topaz 5 geode pearl stone topaz
6 garnet quartz gypsum 6 garnet gypsum quartz
7 amethyst azurite emerald 7 azurite diamond emerald
8 fluorite sapphire    8 amethyst fluorite sapphire
9                      9 malachite
```

To make the problem a little more “fun”, you are NOT ALLOWED to use:

- the STL `vector` subscript/indexing operator, `[]`, or `.at()`,
- the STL `sort` function, or
- any of the `push` or `pop` functions on `vector` or `list`.

You may assume that the initial vector has at least as many slots as the longest word in the structure.

3.1 Complexity Analysis - Big 'O' Notation [ / 6 ]

Once you’ve finished your implementation on the next pages, analyze the running time of your solution. Assume there are `w` total words in the whole structure, `v` slots in the `vector`, a maximum of `m` words per `list`, and `x` words are misplaced and need to be moved. Write 2-3 concise and well-written sentences justifying your answer.

Solution: We need to walk through all of the slots and all of the elements in every slot/list to find all of the misplaced words. The walk is \( O(w) = O(v \times m) \). Deciding if a word is misplaced is constant (calculating the length of the word, and deciding if it greater\(^1\) than the element before and less than the element after). Removing the word is constant. All misplaced words will trigger a walk of the vector \( O(v) \), plus a walk of the single list to find the appropriate insertion point, \( O(m) \). Inserting the word is constant. Overall: \( O(v \times m + x \times (v + m)) \). If \( x \) is large, and \( v \) is small (thus \( x \) and \( m \) are close to \( w \)), then the running time is \( O(w^2) \) – that is, an inefficient insertion sort algorithm. But if \( x \) is quite small (as described in the problem instructions), then the running time is closer to \( O(m \times v) = O(w) \).

\(^1\)Note: comparing two very long words to determine which comes first alphabetically is actually linear \( O(v) \). So the `organize` walk is actually \( O(w \times v) = O(v^2 \times m) \). And the `place` walk is actually \( O(m \times v) \). Overall \( O(v^2 \times m + x \times (v + m \times v)) \). Simplified: \( O(w^2) \) for large \( x \), small \( v \). And \( O(w \times v) \) for small \( x \).
3.2 Helper Function Implementation [ / 12 ]

Alyssa suggests writing a helper function named `place` that will place a word in the correct location in the structure. Work within the provided framework below. Do not add any additional `for` or `while` loops.

Solution:
```cpp
void place(std::vector<std::list<std::string> > &words, const std::string& word) {
    int count = 0;
    std::vector<std::list<std::string> >::iterator itr = words.begin();
    while (itr != words.end()) {
        if (word.size() == count) {
            std::list<std::string>::iterator itr2 = (*itr).begin();
            std::string last = "";
            while (itr2 != (*itr).end()) {
                if (word < *itr2) {
                    (*itr).insert(itr2,word);
                    return;
                }
                itr2++;
            }
            (*itr).insert(itr2,word);
            return;
        }
        itr++;
        count++;
    }
}
```

3.3 Organize Implementation [ / 12 ]

And now write the `organize` function, which calls the `place` function. Again, work within the provided framework below and do not add any additional `for` or `while` loops.

Solution:
```cpp
void organize_words(std::vector<std::list<std::string> > &words) {
    int count = 0;
    std::vector<std::list<std::string> >::iterator itr = words.begin();
    while (itr != words.end()) {
        std::list<std::string>::iterator itr2 = (*itr).begin();
        std::string last = "";
        while (itr2 != (*itr).end()) {
            std::string word = *itr2;
            if (word.size() != count || (last != "" && word < last)) {
                itr2 = (*itr).erase(itr2);
                place(words,word);
            } else {
                last = *itr2;
            }
            itr2++;
        }
        itr++;
        count++;
    }
}
```
4 Merge-Spiration: Recursive Interval Computation

Ben Bitdiddle was inspired by the recursive merge sort example from Data Structures lecture and proposes it as a guide to compute the smallest interval that contains a collection of floating point numbers (e.g., the minimum and maximum). Implement Ben’s idea, a recursive function named `compute_interval` that takes in an STL vector of floats and returns an `Interval` object.

For example:

```
6.2 4.3 10.4 2.5 8.4 1.5 3.7 → [1.5, 10.4]
```

Solution:

```cpp
class Interval {
    public:
        Interval(float i, float j)
            : min(i), max(j) {}
        float min;
        float max;
};

Interval compute_interval(const std::vector<float> &data, int i, int j) {
    // cannot compute an interval for no values
    assert (i <= j);
    if (i == j) return Interval(data[i], data[i]);
    int mid = (i+j)/2;
    Interval low = compute_interval(data, i, mid);
    Interval high = compute_interval(data, mid+1, j);
    if (low.min > high.min) low.min = high.min;
    if (low.max < high.max) low.max = high.max;
    return low;
}

Interval compute_interval(const std::vector<float> &data) {
    return compute_interval(data, 0, data.size()-1);
}
```

Without resorting to personal insults, explain in two or three concise and well-written sentences why Ben’s idea isn’t going to result in significant performance improvements. Be technical.

Solution: Calculating the max and min of an (unsorted) sequence of numbers requires only a linear scan/visit of the elements, comparing each element to the current min & max. \( n \) elements and \( 2n \) comparisons, so overall = \( O(n) \). Ben’s algorithm will also visit each element once, and at each recursive call it will do 2 comparisons. If we draw out the tree we see that we have \( n \) recursive calls. So the algorithms are basically equivalent in Big O Notation for performance / running time. However, function calls are expensive (more expensive than a simple loop), so in practice the running time of Ben’s recursive algorithm will probably be slower (but it’s not terrible).

5 How many DS students to change a lightbulb?

In this problem you will complete the implementation of two new classes named `Bulb` and `Lamp`. We begin with an example of how these classes are used.

First, we create a new lamp that will hold 3 bulbs and make a note of the manufacturer’s recommended bulb: a 60 watt bulb with an estimated lifetime of 300 hours from Phillips. Note that initially this lamp has no bulbs installed.

```
Lamp floorlamp(Bulb(60,300,"Phillips"),3);
bool success;
success = floorlamp.install(); assert(success);
floorlamp.On(50);
assert (floorlamp.getTotalWattage() == 60);
```

Next, we attempt to install 3 bulbs, another of the manufacturer’s recommended bulbs, and then two other brands of bulbs. The installation of the 3rd bulb made by Sylvania fails because there are no available sockets slots in the lamp and no bulbs are burnt out and need replacement.

```
success = floorlamp.install(); assert(success);
success = floorlamp.install(Bulb(40,120,"GE")); assert(success);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(!success);
```

We then use the lamp for another 100 hours. Once the wattage drops (due to a burnt out bulb), we again try to install the Sylvania bulb and it is successful.

```
floorlamp.On(100);
assert (floorlamp.getTotalWattage() == 160);
floorlamp.On(50);
```
assert (floorlamp.getTotalWattage() == 120);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(success);
assert (floorlamp.getTotalWattage() == 240);

Finally, we create a duplicate lamp. Note that when we do this, we match the bulbs currently installed in the original lamp, but the bulbs installed in the new lamp are brand new (and unused).

Lamp another(floorlamp);
assert (floorlamp.getTotalWattage() == another.getTotalWattage());
for (int i = 0; i < 10; i++) {
    floorlamp.On(50);
    another.On(50);
    std::cout << "compare " << floorlamp.getTotalWattage() << " "
               << another.getTotalWattage() << std::endl;
}

Which results in this output:

cmpare 240 240
cmpare 240 240
cmpare 180 240
cmpare 120 240
cmpare 120 240
cmpare 120 240
cmpare 120 240
cmpare 120 120
cmpare 120 120
cmpare 120 120
5.1 Bulb Class Declaration [ / 14 ]

The Bulb class is missing only one function. *You will need to read the rest of the problem to determine what’s missing.* Fill in the missing function – implement the function right here, within the class declaration.

```cpp
class Bulb {
public:
    // constructors
    Bulb(int w, int l, const std::string &b) :
        wattage(w), lifetime(l), hours_used(0), brand(b) {}
    Bulb(const Bulb& b) :
        wattage(b.wattage), lifetime(b.lifetime), hours_used(0), brand(b.brand) {}    

    // accessors
    int getWattage() const { return wattage; }
    bool burntOut() const { return hours_used > lifetime; }
    const std::string& getBrand() const { return brand; }
    // modifier
    void On(int h) { hours_used += h; }
private:
    // representation
    int wattage;
    int lifetime;
    int hours_used;
    std::string brand;
};
```

5.2 Lamp Class Declaration [ / 14 ]

The Lamp class has a few more missing pieces. *Read through the rest of the problem before attempting to fill this in.* Write the prototypes (not the implementation!) for the four missing functions. You will implement some of these missing functions later. Also, fill in the member variables for the Lamp representation. Important: You may not use STL vector on this problem.

```cpp
class Lamp {
public:
    // constructors, assignment operator, destructor
    Lamp(const Bulb& b, int num);
    Lamp(const Lamp &l);
    const Lamp& operator=(const Lamp &l);
    ~Lamp();

    // accessor
    int getTotalWattage() const;
    // modifiers
    bool install(const Bulb &b = Bulb(0,0,""));
    void On(int h);
private:
    // representation
    Bulb recommended;
    Bulb** installed;
    int max_bulbs;
};
```

Lamp Class Implementation

Here’s the implementation of one of the key member functions of the Lamp class.
bool Lamp::install(const Bulb &b) {
    // first, let's figure out where to install the bulb
    int which = -1;
    for (int i = 0; i < max_bulbs; i++) {
        // check for an empty socket
        if (installed[i] == NULL) {
            which = i;
            break;
        }
        // or a socket that contains a burnt out bulb
        if (installed[i]->burntOut()) {
            which = i;
            delete installed[i];
            break;
        }
    }
    // return false if we cannot install this bulb
    if (which == -1) return false;
    if (b.getWattage() == 0) {
        // install the manufacturer's recommended bulb type
        installed[which] = new Bulb(recommended);
    } else {
        // install the specified bulb
        installed[which] = new Bulb(b);
    }
    return true;
}

On the last two pages of this problem you will implement three important functions for the Lamp class, as they would appear outside of the class declaration (in the lamp.cpp file) because their implementations are > 1 line of code.

5.3 Lamp Constructor [ / 9 ]

Solution:
Lamp::Lamp(const Bulb& b, int num) : recommended(b) {
    installed = new Bulb*[num];
    for (int i = 0; i < num; i++) {
        installed[i] = NULL;
    }
    max_bulbs = num;
}

5.4 Lamp Destructor [ / 5 ]

Solution:
Lamp::~Lamp() {
    for (int i = 0; i < max_bulbs; i++) {
        // note: this check not necessary, ok to call delete on a NULL ptr
        if (installed[i] != NULL) {
            delete installed[i];
        }
    }
    delete [] installed;
}

5.5 Lamp Assignment Operator [ / 9 ]

Solution:
const Lamp& Lamp::operator=(const Lamp &l) {
    if (this != &l) {
        for (int i = 0; i < max_bulbs; i++) {
            if (installed[i] != NULL) {
                delete installed[i];
            }
        }
    }
    return *this;
}
6  Singly Linked List Subsequence Sum [    / 18 ]

Write a recursive function named `FindSumStart` that takes the head Node of a singly-linked list storing positive numbers. The function should return a pointer to the Node that begins a subsequence of numbers that ends in the sum of that subsequence. For example, given this sequence: 5 1 4 2 3 9 6 7 the function should return a pointer to the Node storing 4, because 4 + 2 + 3 = 9.

Solution:

```cpp
template <class T> Node<T>** FindSumStart(Node<T>* n) {
    if (n == NULL) {
        return NULL;
    }
    int total = 0;
    Node<T>* tmp = n;
    while (tmp != NULL) {
        if (total == tmp->value) {
            return n;
        }
        total += tmp->value;
        tmp = tmp->next;
    }
    return FindSumStart(n->next);
}
```

Assuming the sequence has n numbers, what is the order notation for the running time of your function?

Solution: $O(n^2)$

7  Reverse Splice [       / 20 ]

Write a function named `reverse_splice` that takes in 3 arguments: an STL list named `data` and two iterators `i` and `j`. The function should reverse the order of the data between those iterators. For example, if `data` initially stores this sequence: 1 2 3 4 5 6 7 8 9 and `i` refers to 3 and `j` refers to 7, then after the call `reverse_splice(data, i, j)`, `data` will contain: 1 2 7 6 5 4 3 8 9, `i` will refer to element 7, and `j` will refer to element 3. Your function should return true if the operation was successful, and false if the request is invalid. Note: Your function may only use a constant amount of additional memory.

Solution:

```cpp
template <class T>
bool reverse_splice(std::list<T> &data,
                    typename std::list<T>::iterator &i,
                    typename std::list<T>::iterator &j) {
    // checking that i comes before j within data
```
if (j == data.end()) {
    return false;
}

// slide the splice end iterator forward (off of the last element)
++j;
typename std::list<T>::iterator k;
for (k = i; k != j; k++) {
    if (k == data.end()) return false;
}

// use a helper iterator to keep track as we walk between the
// endpoints of the splice
k = j;
while (k != i) {
    // move one element
    data.insert(k,*i);
    k--;
    i = data.erase(i);
}

// back the splice end iterator back onto the last element
j--;
return true;

8     Doubly Linked Factorization

class Node {
public:
    Node(int v) :
        value(v),
        next(NULL),
        prev(NULL) {}
    int value;
    Node* next;
    Node* prev;
};

Write a recursive function named Factor that takes in two arguments, pointers to the head and tail Nodes of a doubly linked list. This function should look for a non-prime number in the linked list structure, break the Node into two Nodes storing two of its factors, and then return true. If all elements are prime the function returns false. For example, if we start with a 3 element list containing 35 30 28 and repeatedly call Factor:

PrintNodes(head);
while (Factor(head,tail)) { PrintNodes(head); }

This is the output:

35 30 28
5 7 30 28
5 7 2 15 28
5 7 2 3 5 28
5 7 2 3 5 2 14
5 7 2 3 5 2 7
5 7 2 3 5 2 7

You may write a helper function. You do not need to write the PrintNodes function.
Solution:

```cpp
bool Factor(Node* &head, Node* &tail, Node* n) {
    // base case
    if (n == NULL) return false;
    // see if this element has any factors
    for (int i = 2; i < n->value; i++) {
        if (n->value % i == 0) {
            // create a new node in front of this one
            Node* tmp = new Node(i);
            // change all of the links
            tmp->prev = n->prev;
            if (n->prev != NULL) {
                tmp->prev->next = tmp;
            }
            tmp->next = n;
            n->prev = tmp;
            n->value = n->value / i;
            // handle the special case of the first node
            if (n == head) head = tmp;
            return true;
        }
    }
    // recurse if we couldn't split this element
    return Factor(head,tail,n->next);
}
```

```cpp
// driver function
bool Factor(Node* &head, Node* &tail) {
    return Factor(head,tail,head);
}
```

9 Dynamically Allocated & Templated Stairs

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');
```

9.1 Stairs Class Declaration

First, fill in the blanks in the class declaration:

Solution:

```cpp
template <class T> class Stairs {
public:
    // constructor
    Stairs(int s, const T& val);
    // destructor
    ~Stairs();
    // prototypes of 2 other important functions related to the constructor & destructor
    Stairs(const Stairs& s);
    Stairs& operator=(const Stairs& s);
}
```
// modifier
void set(int i, int j, const T& val) { data[i][j] = val; }

/* NOTE: other Stair functions omitted */
private:
    // representation
    int size;
    T** data;
};

9.2 Stairs Constructor [ / 9 ]

Now write the constructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```cpp
template <class T> Stairs<T>::Stairs(int s, const T& val) {
    size = s;
    data = new T*[s];
    for (int i = 0; i < s; i++) {
        data[i] = new T[i+1];
        for (int j = 0; j <= i; j++) {
            data[i][j] = val;
        }
    }
}
```

9.3 Stairs Destructor [ / 5 ]

Now write the destructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```cpp
template <class T> Stairs<T>::~Stairs() {
    for (int i = 0; i < size; i++) {
        delete [] data[i];
    }
    delete [] data;
}
```

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

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.

10.1 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.

Solution:

```cpp
Node* search(Node* head, const std::string& word) {
    if (head == NULL) return NULL;
    if (head->word == word) {
        return head;
    }
    return search(head->next, word);
}
```
If the Node chain contains \( n \) elements, what is the running time of the search function?

Solution: O(n)

10.2 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 Nodes named sentence contains:

the quick brown fox jumps over the lazy dog

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

```c++
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.

Solution:
```c++
bool compare(Node *a, Node *b) {
    if (a == NULL) return false;
    if (b == NULL) return false;
    if (a == b) return false;
    if (a->next == b) return true;
    return compare(a->next,b);
}
```

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

Solution: O(n)

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.

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

Here’s Ben’s new compare function:
```c++
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!
10.3 Implementing `insert_between` and Maintaining Fast Comparisons

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.

Solution:

```cpp
bool insert_between(Node *head, const std::string& before, const std::string& after, const std::string& word) {
    if (head == NULL) return false;
    if (head->next == NULL) return false;
    if (head->word == before && head->next->word == after) {
        Node* tmp = new Node;
        tmp->word = word;
        tmp->prev = head;
        tmp->next = head->next;
        tmp->next->prev = tmp;
        tmp->prev->next = tmp;
        tmp->distance = (tmp->next->distance + tmp->prev->distance) / 2.0;
        return true;
    } else {
        return insert_between(head->next,before,after,word);
    }
}
```

11 Erase Middles

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
```

Solution:

```cpp
template <class T>
int erase_middles(std::list<T>& data, const T& val) {
    bool found_first = false;
    typename std::list<T>::iterator prev = data.end();
    typename std::list<T>::iterator itr = data.begin();
    int count = 0;
    while (itr != data.end()) {
        if (*itr == val) {
            if (prev != data.end()) {
```

12 Debugging Skillz

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

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

Solution: G  
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.

Solution: A  
I'm unsure where the program is crashing.

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

Solution: D  
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.

Solution: B) Once you've found the general area of the problem, it can be helpful to add a breakpoint shortly before the crash, so you can examine the situation more closely. C) Once you've decided the state of the program is reasonable, you can advance the program one line at a time using next or step into a helper function that may be causing problems. Rebooting your computer is unlikely to fix a bug in your own code.