Boxy Storage Solutions

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.

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

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;
};

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

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:

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 rather than the right-hand operand, s. It should return a reference (to avoid unnecessary 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];
    delete [] data;
    delete [] data[0];
}
```

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 {
    public:
        // REPRESENTATION
        T value;
        Node<T> *up;
        Node<T> *down;
        Node<T> *left;
        Node<T> *right;
};
```
2.1 Diagram

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

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

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

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

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++;
    }
}
```
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).