1  Linked Tube Repair [ / 33 ]

Alyssa P. Hacker is working on a modified linked list that is both two-dimensional and circular. A small sample with height=3 and circumference=4 is shown below. Each templated Node has pointers to its 4 neighbors. The top and bottom edges of the tube structure have NULL pointers. But the left and right edges wrap around, like a circularly linked list. This cylindrical tube structure may have any number of nodes for its height and its circumference.

1.1 Tube repair Diagram [ / 4 ]

First Alyssa wants to tackle the challenge of repairing a hole in the structure. Assume a single Node is missing from the structure, and we have a pointer \( n \) to the Node immediately to the left of the hole. Modify the diagram below to show all of the necessary edits for a call to \( \text{repair}(n,7) \);

Solution: The diagram should be neatly edited to add a new Node box. The value 7 should be stored in the box. The 4 pointers from the new box should point at the appropriate neighbors and the 4 neighbors should point back at the new Node.

1.2 Thinking about Tube repair Complexity [ / 3 ]

The repair function should have constant running time in most cases. Describe an example structure with a single missing Node that can be repaired, but not in constant time. Write 2-3 concise and well-written sentences. You may want to complete the implementation on the next page before answering.

Solution: The repair can be solved in constant time by walking around the hole. Special case code must be written if the hole is at the top or bottom edge of the tube – but both cases can be solved in constant time. However, if the height of the structure is 1, The only way to get to the right edge of the hole is to walk to the left, through all the nodes in the circumference. \( O(c) \), where \( c = \) the circumference.

```cpp
template <class T>
class Node {
public:
  // REPRESENTATION
  T value;
  Node<T> *up;
  Node<T> *down;
  Node<T> *left;
  Node<T> *right;
};
```
Now, implement `repair`, which takes 2 arguments: a pointer to the `Node` immediately to the left of the hole and the value to be stored in the hole. You may assume a single `Node` is missing from the structure.

**Solution:**

```cpp
template <class T> void repair(Node<T>* n, const T &value) {
    Node<T>* tmp = new Node<T>;
    tmp->value = value;
    // left
    tmp->left = n;
    tmp->left->right = tmp;
    // up
    if (n->up != NULL) {
        tmp->up = n->up->right;
        tmp->up->down = tmp;
        tmp->right = n->up->right->right->down;
    } else {
        // special case, hole at the top edge
        tmp->up = NULL;
    }
    // down
    if (n->down != NULL) {
        tmp->down = n->down->right;
        tmp->down->up = tmp;
        tmp->right = n->down->right->right->up;
    } else {
        // special case, hole at the bottom edge
        tmp->down = NULL;
    }
    // right
    if (n->up == NULL && n->down == NULL) {
        // non-constant time special case: height == 1
        Node<T>* tmp2 = n;
        while (tmp2->left != NULL) tmp2 = tmp2->left;
        tmp->right = tmp2;
    }
    tmp->right->left = tmp;
}
```
Now write `destroy_tube` (and any necessary helper functions) to clean up the heap memory associated with this structure. The function should take a single argument, a pointer to any `Node` in the structure. You may assume the structure has no holes or other errors. You cannot use a `for` or `while` loop.

**Solution:**

```cpp
template <class T> void destroy_row(Node<T>* n) {
    if (n->right != NULL)
        destroy_row(n->right);
    delete n;
}

template <class T> void destroy_rows(Node<T>* n) {
    if (n->down != NULL)
        destroy_rows(n->down);
    n->left->right = NULL;
    destroy_row(n);
}

template <class T> void destroy_tube(Node<T>* n) {
    if (n->up != NULL)
        destroy_tube(n->up);
    else
        destroy_rows(n);
}
```

**Note:** The following attempt at a flood fill solution does not work. It will attempt to free the same Nodes multiple times.

```cpp
template <class T> void destroy_tube_buggy(Node<T>* n) {
    Node<T>* u = n->up;
    Node<T>* d = n->down;
    Node<T>* l = n->left;
    Node<T>* r = n->right;
    if (u != NULL) u->down = NULL;
    if (d != NULL) d->up = NULL;
    if (l != NULL) l->right = NULL;
    if (r != NULL) r->left = NULL;
    delete n;
    if (u != NULL) destroy_tube_buggy(u);
    if (d != NULL) destroy_tube_buggy(d);
    if (l != NULL) destroy_tube_buggy(l);
    if (r != NULL) destroy_tube_buggy(r);
}
```
Complete the Vec assignment operator implementation below, while minimizing wasted heap memory. Assume the allocator is most efficient when all heap allocations are powers of two (1, 2, 4, 8, 16, etc.)

**Solution:**

```cpp
1 template <class T>
2 Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
3 if (this != &v) {
4     delete [] m_data;
5     m_size = v.m_size;
6     m_alloc = pow(2, ceil(log2(m_size)));
7     m_data = new T[m_alloc];
8     for (int i = 0; i < m_size; ++i) {
9         m_data[i] = v.m_data[i];
10     }
11 }
12 return *this;
13 }
```

Add code below to perform a simple test of the assignment operator:

**Solution:**

```cpp
Vec<double> v; v.push_back(3.14159); v.push_back(6.02); v.push_back(2.71828);
Vec<double> v2; v2.push_back(1.0); v2.push_back(2.0);
assert (v2.size() != v.size()) && v2[1] != v[1]);
v2 = v;
assert (v2.size() == v.size()) && v2[1] == v[1]);
```

Is line 12 necessary? Continue your testing code above with a test that would break if line 12 was omitted.

**Solution:** We need the return value if the result of the assignment is stored or used as part of a larger expression.

```cpp
Vec<double> v3; v3.push_back(100.0);
v[1] = 1.414;
v3 = v2 = v;
```

What is the purpose of line 3? Write code for a test that would break if lines 3 and 10 were omitted.

**Solution:** It prevents errors for self-assignment. If we delete the object, we have nothing to use as our example to copy from!

```cpp
v = v;
assert (v.size()) == 3 && v[2] == 2.71828);
```
Write a function `embellish` that modifies its single argument, `sentence` (an STL list of STL strings), adding the word “very” in front of “pretty” and adding “with a wet nose” after “grey puppy”. For example:

```
the pretty kitty sat next to a grey puppy in a pretty garden
```

Should become:

```
the very pretty kitty sat next to a grey puppy with a wet nose in a very pretty garden
```

### Solution:

```cpp
void embellish(std::list<std::string> &sentence) {
    std::list<std::string>::iterator itr = sentence.begin();
    std::string prev = ""
    while (itr != sentence.end()) {
        std::string word = *itr;
        if (word == "pretty") {
            sentence.insert(itr,"very");
            itr++;
        } else if (prev == "grey" && word == "puppy") {
           (itr++;
            sentence.insert(itr,"with");
            sentence.insert(itr,"a");
            sentence.insert(itr,"wet");
            sentence.insert(itr,"nose");
        } else {
            itr++;
        }
        prev = word;
    }
}
```

If there are \(w\) words in the input sentence, what is the worst case Big O Notation for this function? If we switched each STL list to STL vector in the above function, what is the Big O Notation?

### Solution: We do an outer loop over all \(w\) words. If every word is “puppy” we will do \(4\) inserts for every word. STL list insert is an \(O(1)\) function. So for list the overall runtime is \(O(w + w * 4 * 1) = O(w)\). However, STL vector insert is an \(O(w)\) operation, so this function (as written) will be \(O(w + w * 4 * w) = O(w^2)\).
Complete `redundant`, which takes a sentence and 2 phrases and replaces all occurrences of the first phrase with the second, shorter phrase. For example “pouring down rain” is replaced with “pouring rain”:

it is pouring down rain so take an umbrella → it is pouring rain so take an umbrella

Or we can just eliminate the word “that” (the replacement phrase is empty):

I knew that there would be late nights when I decided that CS was the career for me → I knew there would be late nights when I decided CS was the career for me

Solution:

typedef std::list<std::string> words;

void redundant(words &sentence, const words &phrase, const words &replace) {
    assert (phrase.size() > replace.size());
    words::iterator s = sentence.begin();
    while (s != sentence.end()) {
        // see if this word is the start of the phrase
        bool match = true;
        words::iterator s2 = s;
        for (words::const_iterator p = phrase.begin(); p != phrase.end(); p++) {
            if (s2 == sentence.end() || *s2 != *p) {
                match = false;
                break;
            }
            s2++;
        }
        // move to the next word if this isn't a match
        if (!match) { s++; continue; }
        // otherwise, remove an appropriate number of words
        for (int i = 0; i < phrase.size() - replace.size(); i++) {
            s = sentence.erase(s);
        }
        // and overwrite the words with the replacement phrase
        for (words::const_iterator r = replace.begin(); r != replace.end(); r++, s++) {
            *s = *r;
        }
    }
}
5  Don’t Ignore Compilation Warnings!  

Write a useful but buggy segment of code (or function) that will compile with no errors but will produce the indicated compilation warning. Put a star ★ next to the line of code that will trigger the warning. Write a concise and well-written sentence describing the intended vs. actual (buggy) behavior of the code.

**warning:** comparison of integers of different signs: 'int' and 'unsigned int'

**Solution:** This code is attempting to print a vector in reverse order. But it will go into an infinite loop because the condition is always true.

```cpp
int zero = 0;
★ for (unsigned int i = vec.size()-1; i >= zero; i--) {
    std::cout << vec[i] << std::endl;
}
```

**warning:** control reaches / may reach end of non-void function

**Solution:** The function below does not handle the case when a == b. Essentially the return value is uninitialized in this case and could be anything.

```cpp
int larger_value (int a, int b) {
    if (a > b) return a;
    if (b > a) return b;
★ }
```

**warning:** variable is uninitialized when used here / in this function

**Solution:** We’ve forgotten to initialize the sum variable, so unfortunately all of the work is wasted. The final value of sum could be anything.

```cpp
int sum;
for (int i = 0; i < vec.size(); i++) {
★ sum += vec[i];
}
```

**warning:** returning reference to local temporary object / reference to stack memory associated with a local variable returned

**Solution:** The code below reads a file into a vector of strings, and wants to save memory & copying by returning this data by reference, but since this local variable is on the stack it disappears as soon as we go out of scope (leave the function). We cannot safely return this vector by reference.

```cpp
std::vector<std::string>& load_data(std::ifstream &istr) {
    std::vector<std::string> answer;
    std::string s;
    while (istr >> s) { answer.push_back(s); }
★ return answer;
}
```

**warning:** expression result unused / expression has no effect

**Solution:** The code below is attempting to add one to every element in the vector... but nothing changes! Instead of addition, the code should either use the += operator or the ++ operator.

```cpp
for (unsigned int i = 0; i < vec.size(); i++) {
★ vec[i] += 1;
}
```
Solution: We intended to use the count variable to count the number of values we read from the file and divide the sum by this value. But the code is incomplete! Whoops!

```cpp
int compute_average(std::ifstream &istr) {
    int sum = 0;
    int count = 0;
    int x;
    while (istr >> x) {
        sum += x;
    }
    return sum;
}
```

6 Cyber Insecurity

Ben Bitdiddle wrote the following code fragment to manage his personal information.

```cpp
std::ifstream istr("my_information.txt");
std::string s;
std::vector<std::string> data;
while (istr >> s) { data.push_back(s); }
std::vector<std::string>::iterator password = data.begin()+4;
data.push_back("credit_card:");
data.push_back("1234-5678-8765-4321");
data[4] = "qwerty";
std::cout << "my password is: " << *password << std::endl;
```

Write “True” in the box next to each true statement. Leave the boxes next to the false statements empty.

- (false) Lines 2 & 3 will produce an “uninitialized read” error when run under gdb or lldb.
- (false) Line 5 is not a valid way to initialize an iterator.
- TRUE Ben’s credit card information is not saved back to the file.
- TRUE This program might behave differently if re-run on this computer or another computer.
- TRUE A memory debugger might detect an “unaddressable access of freed memory” error on Line 9.
- TRUE If we move lines 6 & 7 after line 9, this code fragment will run without memory errors.
- (false) This code contains memory leaks that can be detected by Dr. Memory or Valgrind.
- TRUE These password choices disqualify Ben from any job in computer security.

Solution: The core problem with the code above is that an iterator is attached to a vector (line 5), then the vector is edited via calls to push_back (lines 6 & 7), and afterwards the iterator is used to access an element in the vector (line 9). Vector iterators may be invalidated (and you should assume they are invalidated) after any edit that might reallocate/relocate the data vector (e.g., push_back, insert, and resize). Note that calls to erase are also dangerous because the data has been shifted and the iterator may point to different data (depending whether it was before or after the erase point).