Section 7. Linked Lists

Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.

This week in section you'll practice with linked lists, which are a new way of storing and organizing data that takes advantage of the power of pointers. Linked lists are definitely a tricky subject, but if you draw lots of diagrams and really nail down your pointer fundamentals, you'll be on the road to success. The topics covered in section this week will show up on assignment 6.

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a `.cpp` file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

Starter code

For all the problems in this handout, assume the following structures have been declared:

```
struct Node {
    int data;
    Node *next;
};

struct StringNode {
    string data;
    StringNode *next;
};

struct DoubleNode {
    double data;
    DoubleNode *next;
};
```

You can also assume the following utility functions have been defined as well:

```
/* Prints the contents of a linked list, in order. */
void printList(Node* list) {
    for (Node* cur = list; cur != nullptr; cur = cur->next) {
        cout << cur->data << endl;
    }
}

/* Frees all the memory used by a linked list. */
void deleteList(Node* list) {
    while (list != nullptr) {
        /* Store where to go next, since we're about to blow up our linked
         * list node.
         */
        Node *next = list->next;
        delete list;
        list = next;
    }
}
```

1) Debugging Linked List Code

Take a look back at the **`sorted`** problem from last week's section. You and your friend are reviewing the section handout, and your friend says they wrote the following program:

```
bool isSorted(Node* front) {
    if (front == nullptr) return true;

    Node* current = front;
    while (current != nullptr) {
        if (current->data > current->next->data) {
            return false;
        }
        current = current->next;
    }

    return true;
}
```

Your friend is being met with a segmentation fault error. What is the cause of the issue?

Solution

The **`while`** loop should be checking **`current->next != nullptr`**, and not **`current != nullptr`**. We need to be looking one node ahead here to be able to check the sorted order, so even if **`current != nullptr`**, if **`current->next == nullptr`**, the **`current->next->data`** will cause a dereference of a nullptr. That's a recipe for a segmentation fault!

2) Tracing Pointers by Reference

One of the trickier nuances of linked lists comes up when we start passing around pointers as parameters by reference. To better understand exactly what that's all about, trace through the following code and show what it prints out. Also, identify any memory leaks that occur in the program.

```cpp
void confuse(Node* list) {
    list->data = 137;
}

void befuddle(Node* list) {
    list = new Node;
    list->data = 42;
    list->next = nullptr;
}

void confound(Node* list) {
    list->next = new Node;
    list->next->data = 2718;
    list->next->next = nullptr;
}

void bamboozle(Node*& list) {
    list->data = 42;
}

void mystify(Node*& list) {
    list = new Node;
    list->data = 161;
    list->next = nullptr;
}

int main() {
    Node* list = /* some logic to make the list 1 -> 3 -> 5 -> null */

    confuse(list);
    printList(list); // defined at beginning of section handout

    befuddle(list);
    printList(list);

    confound(list);
    printList(list);

    bamboozle(list);
    printList(list);

    mystify(list);
    printList(list);

    deleteList(list); // defined at beginning of section handout
    return 0;
}
```

Solution

Let's go through this one step at a time.

- The call to **`confuse`** updates the first element of the list to store 137, so the call to **`printList`** will print out 137, 3, 5. Although the argument is passed by value, because both pointers point to the same Node in memory, the original list will also see a different value.
- The call to **`befuddle`** takes its argument by value. That means it's working with a copy of the pointer to the first element of the list, so when we set list to be a new node, it doesn't change where the list variable back in main is pointing. The node created in this function is leaked, and the next call to **`printList`** will print out 137, 3, 5.

- The call to **confound** takes its argument by value. However, when it writes to list->next, it's following the pointer to the first element of the linked list and changing the actual linked list node it finds there. This means that the list is modified by dropping off the 3 and the 5 (that memory gets leaked) and replacing it with a node containing 2718. Therefore, the next call to **printList** will print out 137, 2718.
- The call to **bamboozle** takes its argument by reference, but notice that it never actually reassigns the next pointer. However, it does change the memory in the node at the front of the list to hold 42, so the next call to **printList** will print 42, 2718.
- The call to **mystify** takes its argument by reference, and therefore, when it reassigns **list**, it really is changing where **list** back in main is pointing. This leaks the memory for the nodes containing 42 and 2718. The variable **list** back in main is changed to point at a new node containing 161, so the final call to **printList** prints 161.
- Finally, we free that one-element list. Overall, we've leaked a lot of memory!

3) Inserting into a Linked List (`insert.cpp`)

Write a function named **insert** that accepts a reference to a **StringNode** pointer representing the front of a linked list, along with an index and a string value. Your function should insert the given value into a new node at the specified position of the list. For example, suppose the list passed to your function contains the following sequence of values:

```
{ "Katherine", "Julie", "Kate" }
```

The call of **insert(front, 2, "Mehran")** should change the list to store the following:

```
{ "Katherine", "Julie", "Mehran", "Kate" }
```

The other values in the list should retain the same order as in the original list. You may assume that the index passed is between 0 and the existing size of the list, inclusive.

Constraints: Do not modify the data field of existing nodes; change the list by changing pointers only. Do not use any auxiliary data structures to solve this problem (no array, Vector, Stack, Queue, string, etc).

```
void insert(StringNode*& front, int index, string value)
```

Solution

```
void insert(StringNode*& front, int index, string value) {
    if (index == 0) {
        front = new StringNode{value, front};
    } else {
        StringNode* temp = front;
        for (int i = 0; i < index - 1; i++) {
            temp = temp->next;
        }
        temp->next = new StringNode{value, temp->next};
    }
}
```

4) Remove All Threshold (`threshold.cpp`)

Write a function **removeAllThreshold** that removes all occurrences of a given double value +/- a threshold value from the list. For example, if a list contains the following values:

{ 3.0, 9.0, 4.2, 2.1, 3.3, 2.3, 3.4, 4.0, 2.9, 2.7, 3.1, 18.2}

The call of **`removeAllThreshold(front, 3.0, .3)`** where front denotes a pointer to the front of list, would remove all occurrences of the value 3.0 +/- .3 (corresponding to the underlined values in the above example) from the list, yielding the following list:

{9.0, 4.2, 2.1, 2.3, 3.4, 4.0, 18.2}

If the list is empty or values within the given range don't appear in the list, then the list should not be changed by your function. You should preserve the original order of the list. You should implement your code to match the following prototype

```
void removeAllThreshold(DoubleNode*& front, double value, double threshold)
```

Solution

```
bool valueWithinThreshold (double value, double target, double threshold)
{
    return value >= target - threshold && value <= target + threshold;
}

void removeAllThreshold(DoubleNode*& front, double value, double
threshold) {
    while (front != nullptr
        && valueWithinThreshold(front->data, value, threshold)) {
        DoubleNode* trash = front;
        front = front->next;
        delete trash;
    }

    if (front != nullptr) {
        DoubleNode* current = front;
        while (current->next != nullptr) {
            if (valueWithinThreshold(current->next->data, value,
threshold)) {
                DoubleNode* trash = current->next;
                current->next = current->next->next;
                delete trash;
            } else {
                current = current->next;
            }
        }
    }
}
```

5) Double List (**`double.cpp`**)

Write a function that takes a pointer to the front of a linked list of integers and appends a copy of the original sequence to the end of the list. For example, suppose you're given the following list:

`{1, 3, 2, 7}`

After a call to your function, the list's contents would be:

`{1, 3, 2, 7, 1, 3, 2, 7}`

Do not use any auxiliary data structures to solve this problem. You should only construct one additional node for each element in the original list. Your function should run in **`O(n)`** time where **`n`** is the number of nodes in the original list. You should implement your code to match the following prototype

```
void doubleList(Node*& front)
```

Solution

```
void doubleList(Node*& front) {
    if (front != nullptr) {
        Node *half2 = new Node{front->data, nullptr};
        Node *back = half2;
        Node *current = front;
        while (current->next != nullptr) {
            current = current->next;
            back->next = new Node{current->data, nullptr};
            back = back->next;
        }
        current->next = half2;
    }
}
```

6) Braiding a Linked List (`braid.cpp`)

Now, let's put together what we've done in the last two problems and combine it into the grand finale, braiding a linked list! Write a function braid that takes a linked list and weaves the reverse of that list into the original. (In this case, you will need to create new nodes.) Here are a few examples:

```
           {1, 4, 2} -> {1, 2, 4, 4, 2, 1}
                  {3} -> {3, 3}
{1, 3, 6, 10, 15} -> {1, 15, 3, 10, 6, 6, 10, 3, 15, 1}
```

You should implement your code to match the following prototype

```
void braid(Node*& front)
```

Bonus: This one also has an interesting recursive solution.

Solution

```
void braid(Node*& front) {
    Node *reverse = nullptr;
    for (Node *curr = front; curr != nullptr; curr = curr->next) {
        Node *newNode = new Node{curr->data};
        newNode->next = reverse;
        reverse = newNode;
    }
    // reverse now addresses a memory-independent copy of the original
list,
    // where all of the nodes are in reverse order.
    for (Node *curr = front; curr != nullptr; curr = curr->next->next) {
        Node *next = reverse->next;
        reverse->next = curr->next;
        curr->next = reverse;
        reverse = next;
    }
}
```