Tommy Hollenberg
CS162 Assign5
Design Document
06/04/2017

Understanding the Problem

There are two programs in this assignment, both of which make use of a singly linked list implementation. The list struct must have a pointer to its tail as well as its head. The list implementation must have the following functionality:

- an init function that initializes an empty list
- a size function that returns the number of nodes
- a print function that prints the values in the list, starting from the head
- a push_front function that adds a new node to the front of the list
- a push_back function that adds a new node to the back of the list
- a front function that returns the value of the node at the head of the list
- a back function that returns the value of the node at the tail of the list
- a pop_back function that removes the tail node from the list and returns its value
- a remove_front function that removes the head node from the list and returns its value
- an empty function that returns true if the list is empty
- a delete function that removes all nodes from the list

Mini-Compiler

The first program implements one of the functions performed by compilers to ensure code is syntactically valid; namely, checking that all opening parenthesis, brackets, and curly braces are paired with matching closing punctuation, and vice versa. The program will accept a string of input from the user and declare its use of brackets, parentheses, and curly braces as valid or invalid. The key is that the first punctuation to be closed must be the last opened, and at the end of the string, all opened punctuation must be closed. All other characters should be ignored.

Comparing Queueing Systems

The second program simulates the efficiency of two queueing systems for use in a fast-food restaurant environment. The program must accept positive integer command-line arguments with the tags "-n", "-s", "-e", and "-c", representing the number of cashiers (N), the minimum possible time in minutes required to serve a customer (S), the maximum possible time in minutes required to serve a customer (E), and the average time in minutes between new customer arrivals (C),

respectively. The first queueing system consists of one large queue that is serviced by all N cashiers as they become available. The second system consists of N queues each serviced by one individual cashier. The program must randomly simulate both queueing systems and report the number of customers in each queue after 1 hour, 5 hours, and 10 hours.

Devising a Plan

List Implementation

Most of the list functionality was already implemented in this past week's Lab. To avoid redundancy, I will only provide pseudocode for the three functions that are non-obvious and have not previously been implemented in Lab.

```
void push_front(pointer to head of list, value to push onto front)
    Allocate a new node, set its value to the parameter value, and
      its next pointer to the head of the list.
    Set list's head pointer to the new node.
    If the list's tail pointer is NULL, set the tail pointer to the
      new node as well (the list was previously empty).

int remove_front(pointer to head of list)
    Declare a temp struct node pointer (initialize it to the list's
      head) and an integer to hold the return value.
    If the list's head pointer is NULL, return -1 (empty list).
    Set the list's head pointer to the address in its next pointer.
    If the list's head pointer is NULL, set the list's tail pointer to
      NULL (there was only one node in the list, so it is now empty).
    Set the return value integer to the value pointed to by temp.
    Free temp.
    Return the value.

void delete(pointer to head of list)
    Declare a temp struct node pointer and initialize it to the list's
      head.
    While temp isn't NULL
        Set list's head pointer to the address in its next pointer.
        Free temp.
        Set temp to the address in the list's head pointer.
    Set the list's tail pointer to NULL.
```

Mini-Compiler

The key criteria that the compiler must check (that the first punctuation to be closed must be the last opened, and at the end of the string, all opened punctuation must be closed) can be modeled with a LIFO data structure (a stack). The list implementation provides this functionality with the push_back and pop_back functions. Once the list is implemented, this program requires almost no effort. The pseudocode to determine string validity follows (I am using a typedef int bool as my return type, with #define true 1 and #define false 0):

```
bool check_validity(C-string "code" to check)
    Declare int i = 0, bool valid = true, and struct list stack. Call
      init(&stack).
    If code is NULL, return false;
    While(code[i] isn't a null terminator)
        If code[i] is an opening '[', '{', or '(', push_back a 0, 1,
          or 2, respectively.
        Else if code[i] is a closing ']', '}', or ')'
            If pop_back doesn't return 0, 1, or 2, respectively
                valid = false
                Break.
        ++i
    valid = valid && empty(stack)
    delete the stack.
    return valid.
```

Queueing System Comparison

I haven't finished this one yet, but I will do so if I have time this week. Since queue behavior is desired, the push_back and remove_front functions will be used in this program. I have written a function to check the command-line arguments:

```
bool check_args(argc, argv, pointers to n, s, e, and c)
    Declare integer i.
    The variables n, s, e, and c are initialized to 0.
    If argc isn't 9, return false.
    For i = 1, i < 9 (could also be 8), i += 2
        If argv[i] is "-n", "-s", "-e", or "-c", and the corresponding
          variable is zero (it hasn't already been listed)
            Set the corresponding variable to atoi(argv[i + 1])
            If the variable is <= 0
                Return false.
        Else return false.
    Return s <= e (the lower limit must be <= the upper limit).
```

Then, a struct list array of size n will be created, as well as a list for the one-for-all queue system. I need to consider further how I will store the information about when the cashiers are occupied and when they are free. My thought at this point, however, is that I will represent the cashiers with an n-element array of integers, one for each queueing system. When a customer is assigned to a cashier, a random integer will be generated on the interval [s, e], by rand() % (e – s + 1) + s, which will be stored in the corresponding element of the array.

Every minute, the array will be looped through and each element will be decremented (if it is greater than 0). If the element is now 0, (if any are in the queue) a new customer will be popped from the front of the queue and assigned to the cashier associated with that element, and a new number representing their service time will be generated.

The random customer arrivals will be modeled by a Poisson distribution, with a rate parameter of $\lambda = \frac{1}{c}$. I found a simple algorithm on Wikipedia attributed to Donald Knuth that will be suitable for the purposes of this program. The algorithm generates a random sample from a Poisson distribution based on random samples from a uniform distribution on the interval (0, 1] (using the rand() function and RAND_MAX macro from stdlib.h). The exp() function from math.h will also be used. This requires compiling the program with the "-lm" tag to link the math library. The algorithm is as follows:

```
int poisson_knuth(double lambda)
    int k = 0.
    double p = 1.0, L = exp(-lambda).
    do
        ++k.
        p *= the random sample of a uniform distribution on (0, 1].
    loop while (p > L).
    return k - 1.
```

The smaller c is, the smaller $e^{-\frac{1}{c}}$ is, and more iterations of the loop will be required, on average, for p to be <= L. This makes sense because the return value, k-1, is the number of new customers that arrived during that minute. Smaller c means a shorter average interval between customer arrivals, which equates to more customer arrivals per minute, on average.

To ensure a fair comparison, the same customer arrival numbers will be used for both systems, and the customer service times will be calculated for each customer as they arrive, so the sum of customer * time will be the same for both systems. In the multiple queue system, the new customers will be pushed onto the back of to the shortest queue (or one of the shortest queues if there is a tie) at the time of their arrival.

Testing Plan

The list implementation must perform all its required functions without producing any memory leak, including pushing new nodes and popping off existing nodes from the front and back, as well as deleting the entire list.

The compiler must properly classify character strings as having valid or invalid parenthesis, bracket, and curly brace usage. The invalid cases include closing punctuation not immediately preceded by the corresponding opening punctuation (excluding non-bracketing punctuation characters, but including when it is the first bracketing punctuation character encountered), and unclosed punctuation.

The queueing comparison program must only accept valid command-line arguments, consisting of tag and positive integer pairs that can come in any order (i.e. NESC, CNSE, etc.). It must also properly simulate the two systems based on the supplied parameters.