**Beteab Gebru**

**Mar 16 2017**

**CSCI 301 - Section 02**

**PROCESSING A LINKED LIST RECURSIVELY**

**Project 6: Program Documentation**

### Introduction

The requirements for this project were to implement recursive versions of the functions for project 4. The functions such as Insert(), Destructor(), Constructor(), Length() were implemented recursively to do the same tasks as ones stated for project4.The project is to build a linked list to store a concordance data of a given text file. A concordance data describes the frequencies of words in text to be used in statistical analysis of written work to help determine authorship of disputed works.

**Design Document**

**Data structures**

An ordered doubly linked list ADT was used to store the concordance information of the given texts. The ADT had the following features;

**Data Members And Functions of the ADT**

class L\_List

{

public:

    struct Node //a single node of the linked list will have the following attributes

    {

        D\_type1 Data;//will store the word of D\_type1 type

        int apearance = 0;//stores number of times a word appears in text

        Node\*next; // pointer that acts as a link to the next word link

        Node\*prev;

    };

private:

    Node\*head;//will be a pointer to head node in the chain

    Node\*tail;//will be a pointer to tail node in the chain

    int Word\_Count=0;//keeps count of distinct words

    L\_List::Node\* L\_List::get\_node(D\_type1 entry);

    void plug(Node\* Current, Node\* parser);

    void give\_Back(Node\*);  //works recursively to deallocate memory deallocate memory

public:

     L\_List(); //    // inline constructor for class L\_List

    ~L\_List();      //list destructor will nulify the list members

    //returns the value of the private datamembers

    Node\* getfirst();

    Node\* getlast();

    int getLength(Node\*);           //returns the legth of the list

    void Data\_Intake(D\_type1 New\_Item);  //inserts new item in a node and chains it to the end

    bool is\_empty();             //returns true if head-> is NULL

    bool Present(Node\* Current, Node \*parser);

    int getCount(Node\* head);       //counts the number of distinct words by counting the nodes in the linked list

//friend functions

    friend std::ostream& operator << (std::ostream& out\_s, L\_List& Chain);  //overloaded <<operator

**Functions**

**The Constructor**

-The constructor in the program simply initializes the Data-members First=NULL, Last =NULL.

**~The Destructor**

The destructor calls the function give\_Back() to efficiently get rid of the data and memory allocation by deleting the nodes one by one.

**Give Back function**

-it is a private function that is called by the destructor to hand over memory back to system. It achieves this by recursively calling itself until the list is empty

**Data Intake function**

-This function gets a parameter passed by a client program and it attaches the new entry into the linked list. -It creates Current node to house the data and initialize the appearance counter

-it asks if !present before it sends the node to the plug function

-if present it simply goes on to take another item to insert

**Present** (); is a function that is called with parameters (Node\* Current, Node \*parser).

-the function takes Current->Data and compares it against the existing nodes

-if it encounters (parser->Data == Curent->Data) it appearance++

-otherwise it returns !found

**Getcount**(); this recursive function calls itself until NULL pointer is encountered

-it returns the number of nodes in the list to get #of distinct concordance words

**Is Empty? Function**

-This function simply returns the truth value of the logical statement (first==NULL)

**The Plug() function**

**-**This function takes two nodes (current and copy of the head pointer ) for parsing

**-**This function recursively calls itself to find right location to plug a new node

**Friend function**

-This is the overloaded output stream function that will be able to take the ordered list ADT object and display its data elements in a chosen format. In my case I have decided to output the elements in a tabular format

**The Main program:**

The main program is used to get the words from the text file input and format them appropriately (all uppercase and max of 8 characters long) and send this separate words individually to the **Data\_Intake()** function as parameters.

The program makes use of three file names stored in an array to test the program. The three input text files are. I Did this to avoid having to type file names during tests. I will be using array index 0,1,2 to use one of the three files

1. "Input\_Normal\_Text.txt"

2. "Input\_No\_Alphabet.txt"

3. "Input\_Empty.txt"

The main function has **Format\_Word()** function that take in a space delimited word and strips the word’s of trailing spaces and punctuations by copying just the alphabet characters and converting them all to upper case characters before they are returned to the main program as a complete word.

We use the main program to print out our list using the overloaded output operator.

### User Document The way to run the program

The program is simply compiled using g++ or run using visual studio. I tested it using g++ but it was crashing during runtime. It runs perfectly fine on my Microsoft visual studio IDE but upon trying to use I discovered run time errors that didn’t appear in the visual studio trial. I have included screen shots of the program running on my visual studio program.

### Testing of the program ::Screen shots from my visual studio IDE

|  |  |
| --- | --- |
| Input for testing | Output screenshot |
| **Normal text input**  My name Is beteab gerbu980.  My name Is beteab gerbu980  My name Is beteab gerbu980  My name Is beteab gerbu980  My name Is beteab gerbu980  My name Is beteab gerbu980  My name Is beteab gerbu980 |  |
| **No text in in file** |  |
| 1276 3817 64866 324841  1276 3817 64866 324841  1276 3817 64866 324841  1276 3817 64866 324841  1276 3817 64866 324841 |  |

### Summary

The program essentially builds concordance data the same as project 4. The implementation of some of the functions in recursive way makes functions like is\_empty(), plug() more efficient and more elegant to look at. The number of comparisons is for most part the same but with fewer lines of code we achieve the aim. The plug function is considerably more efficient since it stops as soon as we hit a true to the condition (current->Data<pointer->data)

**Conclusion**

With this project I have managed to learn how to maintain a linked list efficiently and was able to learn various algorithms to manipulate string datatypes. It was frustrating when a seemingly logical statements don’t work but I was able to discover ways around some errors during run time by using debugging lines that pause **system(“pause”)**the program so I can look at where the bug occurs.