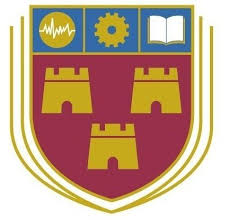
Computing 3: 2020-2021

DSA II – Assignment 1



Lecturer: Áine Byrne

Student Name: Rachel Doogue

Student Number: C00237335

Due Date: 11th December



**Description of Problem**



This is a report on how to write a menu-driven application to implement a Text Editing Management (TEM) application and its operations. The TEM application allows a user options to add words, one by one into a text document. Update and Undo options are also provided for entered words. Update will allow option to change an already entered word and the Undo option will erase the last word you have typed in. Therefore, the words are erased in reverse order of their typing.

TEM needs to be able to perform the following operations implemented using a linked structure:

## addWord() : adds most recent typed-in word into the document

## Undo() : removes the most recently typed word

## lastTypedWord () : returns the last word that was typed in

## Update(): replaces an existing word with a new word

## size() : returns the number of words entered

## isEmpty () : checks if the document is empty

**clear() :** removes all the words from the document

Along with the listed operations above 2 more operations which would be suitable and useful for TEM application must be included.



**Data Structures Used**

To create a TEM I used a singly linked list. This is list of nodes. Each contains data and the address of the next node. Each node points to one list element. Singly linked lists are unidirectional.



A picture containing diagram

Description automatically generated

**Pseudocode for Algorithms used**

* initialiseAtHead()

{

create newnode

cin string

newnode->data = string

newnode->next = head



head = newnode

return head

}

* addWord()

{

cin string

node current = head

create newnode

while (current->next != NULL)

{

Current = current->next



}

cin string

newnode = string

current->next = newnode

}

* undo()

{

create secondlast = head

while (secondlast->next->next != NULL)

{

secondlast = secondlast->next

}



delete secondlast->next

secondlast->next = NULL

}

* lastTypedWord()

{

create secondlast = head

while (secondlast->next->next != NULL)

{



secondlast = secondlast->next

}

cout secondlast->next->data

}

* update()

{

cin stringFind



cin stringReplacement



node current head



while (current->next != NULL)



{



current = current->next

if (current->data == stringFind)



{

current->data = stringReplacement

}

}

}

* size()

{

node current = head

int count

while (current != NULL)

{

current = current->next



count++

}

cout count

}

* isEmpty()

{

node current = head

if (current == NULL)

{



cout Empty

}

else

{

cout Not Empty

}

}

* clear()

{

node current = head

if (current == NULL)

{

cout Already Clear

}

else



{

delete current

cout Clear

}

}

* print()

{

node current = head

while (current != NULL)

{

cout current->data



current = current->next

}

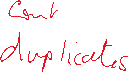
}

* repeat()

{

cin string

int count



node current = head



while (current->next != NULL)

{



if (current->data == string)

{



count++

}

Current = current->next

}

count count

}

**Extra Operations**

For the 2 extra operations I choose to have a function that would print the data that is in each node. For the second function you choose a word that you wish to find the number of times it repeats itself.



**Copy of Code**

//Student Name: Rachel Doogue

//Student Number: C00237335

//Discrete Structures Program 1

#include <iostream>

#include <string>

class node

{

public:

std::string data;

node\* next;

node(std::string input)

{

data = input; next = NULL;

}

};

void DeleteFirst(node\*);

node\* initialiseAtHead(node\*);

node\* addWord(node\*);

node\* undo(node\*);

node\* lastTypedWord(node\*);

node\* update(node\*);

node\* size(node\*);

node\* isEmpty(node\*);

node\* clear(node\*);

node\* print(node\*);



node\* repeat(node\*);

int main(int argc, char\* argv[])

{

node\* current;

node\* head = NULL;

int response = -1;

head = initialiseAtHead(head);

while (response != 0)

{

std::cout << "\n";

std::cout << "Press 1 to add most recent typed-in word into the document\n";

std::cout << "Press 2 to remove the most recently typed word\n";

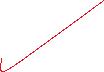
std::cout << "Press 3 to return the last word that was typed in \n";

std::cout << "Press 4 to replaces an existing word with a new word\n";

std::cout << "Press 5 to return the number of words entered\n";

std::cout << "Press 6 to check if the document is empty\n";

std::cout << "Press 7 to remove all the words from the document\n";



std::cout << "Press 8 to print all words\n";

std::cout << "Press 9 to find the number of times a word repeats\n";

std::cout << "Press 10 to exit\n";

std::cin >> response;

switch (response)

{

case 1:

head = addWord(head);

system("CLS");

break;

case 2:

head = undo(head);

system("CLS");

break;

case 3:

head = lastTypedWord(head);

break;

case 4:

head = update(head);

system("CLS");

break;

case 5:

head = size(head);

break;

case 6:

head = isEmpty(head);

break;

case 7:

head = clear(head);

system("CLS");

break;

case 8:

head = print(head);

break;

case 9:

head = repeat(head);

break;

case 10:

std::cout << "\n exit \n ";

break;

}

}

return 0;

}

//Initialise linked list

node\* initialiseAtHead(node\* inHead)

{

std::string firstWord;

node\* new\_node;

std::cout << ("Enter your first word \n");

std::cin >> firstWord;

new\_node = new node(firstWord);

new\_node->next = inHead;

inHead = new\_node;

return inHead;

}

//adds most recent typed word

node\* addWord(node\* inHead)

{

std::string word;

node\* current;

node\* new\_node;

current = inHead;

while (current->next != NULL)

{

current = current->next;

}

std::cin >> word;

new\_node = new node(word);

current->next = new\_node;

return inHead;

}

//removes the most recently typed word

node\* undo(node\* inHead)

{

node\* second\_last = inHead;

while (second\_last->next->next != NULL)

{

second\_last = second\_last->next;

}

delete (second\_last->next);

second\_last->next = NULL;

return inHead;

}

//returns the last word that was typed in

node\* lastTypedWord(node\* inHead)

{

node\* second\_last = inHead;

while (second\_last->next->next != NULL)

{

second\_last = second\_last->next;

}

std::cout << "\nLast word typed: " << (second\_last->next->data) << "\n";

return inHead;

}

//replaces an existing word with a new word

node\* update(node\* inHead)

{

std::string find;

std::string replacement;

node\* current = inHead;

std::cout << "\n What word do you want to replace? \n";

std::cin >> find;

std::cout << "\n What word do you want to replace it with? \n";

std::cin >> replacement;

while (current->next != NULL)

{

current = current->next;

if (current->data == find)

{

current->data = replacement;

}

}

return inHead;

}

//returns the number of words entered

node\* size(node\* inHead) // returns the number of words entered

{

node\* current = inHead;

int count = 0;

while (current != NULL)

{

current = current->next;

count++;

}

std::cout << "\nThere are " << count << " words \n";

return inHead;

}

//checks if the document is empty

node\* isEmpty(node\* inHead)

{

node\* current = inHead;

if (current == NULL)

{

std::cout << ("\n Empty \n");

}

else

{

std::cout << ("\n Is not empty \n");

}

return inHead;

}

//removes all the words

node\* clear(node\* inHead)

{

node\* current = inHead;

if (current == NULL)

{

std::cout << ("\n Already clear \n");

}

else

{

delete (current);

std::cout << ("\n Clear \n");

}

return inHead;

}

//prints all words

node\* print(node\* inHead)

{

node\* current = inHead;

std::cout << ("\n All elements in list \n");

while (current != NULL)

{

std::cout << current->data;

std::cout << (" \n");

current = current->next;

}

return inHead;

}

//counts the number of times a word was repeated

node\* repeat(node\* inHead)



{

std::string find;

int count = 0;

node\* current = inHead;

std::cout << "\n What word do you want to find? \n";

std::cin >> find;

while (current->next != NULL)

{

if (current->data == find)

{

count++;

}

current = current->next;

}

std::cout << "\nThe word " << find << " repeats " << count << " times. \n";

return inHead;

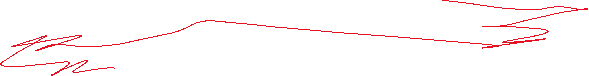
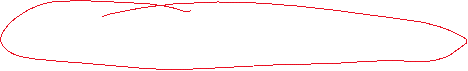
}



**Description of all Methods**



initialiseAtHead() is used to put the first word into the linked list. Without this the program would shut down when trying to access the menu options. It has the user enter a word for the list. This word is then entered into the new node that had been created. This new node is the head of the list.



addWord() lets the user at any word to the list. The node current is used to go through the list until the last node is found. When this is found a new node is created and the word that the used input is inserted into the list.



undo() removes the last node in the list. It will start at the head of the list. As it is trying to find the second last node it will look for the node that is then followed by a node equal to null. When this is found it is indicated that the next node after the second last is to be deleted.



lastTypedWord() follows the same principles undo(). It will start at the head and then will search for a node that is followed by a node equal to null. Unlike undo() this time nothing will be deleted. Instead, this time the data from the node next to the second last node will be output to the screen.

update() has the user input two strings at the start. The first is the string they wish to replace. The second is the string that they wish to put in the linked list, replacing another. While the next node is not equal to null the function will go through each node to find the word the user is looking for until the end. When it is found the word is then replaced. Because the loop continues until the very end of the linked list if the word appears multiple times it will be replaced each time.

size() used the same while loop to go through the entire linked list. Each time it finds a node with next not equal to null a counter will be incremented by 1. Once the loop is finished the number of words counted will be output.

isEmpty() starts at the head of the linked list. A new node current is created to see if the head is equal to null. If it is then the list contains no other nodes. If not then this means that there are nodes with data in them in the list. A message is output to the user to indicate which is true.

clear() uses the same structure as isEmpty(). The only difference is that instead when a linked list is empty it tells a user that it is already clear. When the list is not already empty the function will just delete the head.

print() use current to start at the head of the linked list. Using a while loop it will go through the entire linked list. Each time it will output the data that is contained in the that node. It will then move current to indicate the next node as to access it.

repeat() asks the user to enter a word that they would like to see how many times it repeats itself. A while loop is used to travel through until the end of the linked list. As it does this it checks if the data in that node is equal to the word entered by the user. If it does the counter is incremented by 1. If not, nothing happens. Current moves onto the next node to check until next node is equal to null. Finally, a message is output to the user to let them know how many times that word repeated.

**Test Data**

* addWord()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| hi my | hi my | hi my |
| name | hi my name | hi my name |
| is | hi my name is | hi my name is |
| Rachel | hi my name is Rachel | hi my name is Rachel |

Text

Description automatically generated

**Text

Description automatically generated**



Text

Description automatically generated



* undo()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| hi my name is Rachel | hi my name is Rachel | hi my name is Rachel |
| hi my name is | hi my name is | hi my name is |
| hi my name | hi my name | hi my name |
| hi my | hi my | hi my |

Text

Description automatically generated



Text

Description automatically generated



* lastTypedWord()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| Hi my | my | my |
| Hello world | world | world |
| Hello | Error, program ends | Hello |
| Time to run the numbers | numbers | numbers |

Text

Description automatically generated



Text

Description automatically generated



* update() – Hello world hello world hello world my name is Rachel



|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| hello – time | Hello world time world time world my name is Rachel | Hello world time world time world my name is Rachel |
| world - World | Hello World time World time World my name is Rachel | Hello World time World time World my name is Rachel |
| Name - Rachel | Hello World time World time World my name is Rachel | Hello World time World time World my name is Rachel |
| dog - fish | Hello World time World time World my name is Rachel | Hello World time World time World my name is Rachel |

Text

Description automatically generated



Text

Description automatically generated



Text

Description automatically generated

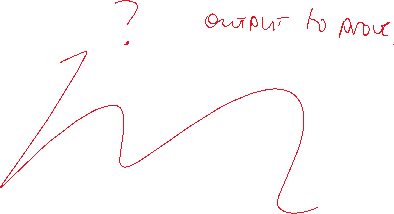


* size()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| Hi my name is Rachel | 5 | 5 |
| Hello world | 2 | 2 |
| What time is it ? | 5 | 5 |
| Testing 1 2 3 | 4 | 4 |

Text

Description automatically generated

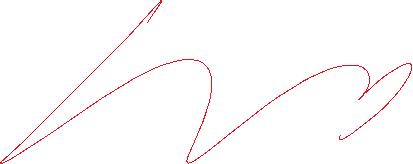


* isEmpty()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
|  | Program shuts down when empty | Empty |
| Hello | Is not empty | Is not empty |
| My name is Rachel | Is not empty | Is not empty |
| The dog is gone | Is not empty | Is not empty |

Text

Description automatically generated



* clear()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| Hello world | Empty list | Empty list |
| My name is Rachel | Empty list | Empty list |
| What time is it ? | Empty list | Empty list |
| Woof | Empty list | Empty list |

Note: if the list is empty and you use print the program shuts down

Text

Description automatically generated



Text

Description automatically generated



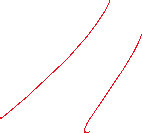
* print()

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| Hello world | Hello world | Hello world |
| My name is Rachel | My name is Rachel | My name is Rachel |
| What time is it ? | What time is it ? | What time is it ? |
| It is raining | It is raining | It is raining |

Text

Description automatically generatedText

Description automatically generated



* repeat() - Hello world hello world hello world my name is Rachel

|  |  |  |
| --- | --- | --- |
| Input | Output | Expected Output |
| hello | 2 | 2 |
| world | 3 | 3 |
| Rachel | 0 | 1 |
| time | 0 | 0 |

Text

Description automatically generatedText

Description automatically generated