

Alif
Rafian
Wais

Text Editor (DSA Final Project)

Problem Description

The functionality that a regular text editor has is usually the ability to edit and store text in a file while it is still being edited, by which the cursor of the text editor has to move around a lot in order to be able to successfully edit text on the go. In order to be able to achieve a higher efficiency for the cursor, we would need a faster data structure for text editing, especially for larger sizes of text.

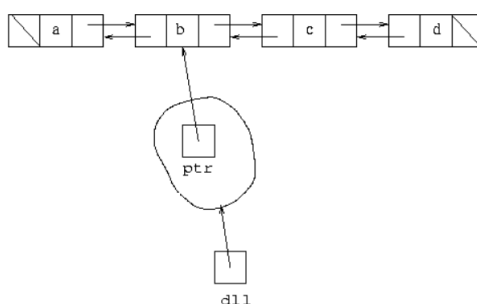
Proposed Data Structure Solution

The solution that we have come up with is creating a text editor that is implemented through a doubly linked list using gap buffers.

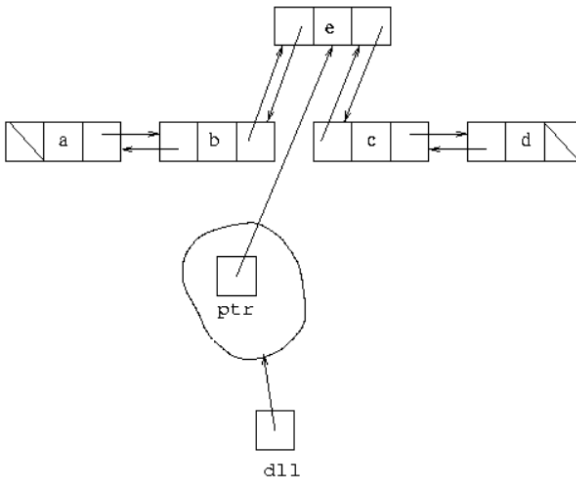
To explain, gap buffers are a data structure that are mainly used to edit and store text that is being edited efficiently. It works similarly to arrays but with gaps in between each index that enables the cursor for the text editor to be able to handle multiple changes. The problem that gap buffers have however, are that they are very inefficient when dealing with larger files, as whenever a change occurs, all the contents in the array must be shifted, which will be a problem if it were a larger file with a larger array.

To solve this problem, we will implement our text editor using a doubly linked list as well, since unlike a normal linked list, the links can go both ways, which means that it will make it easy for the cursor to move around. It can also perform better than double stacks and gap buffers for larger files, where the insertion and deletion for the text is very fast. However, doubly linked lists consume a lot of memory, and may not function as well for smaller text.

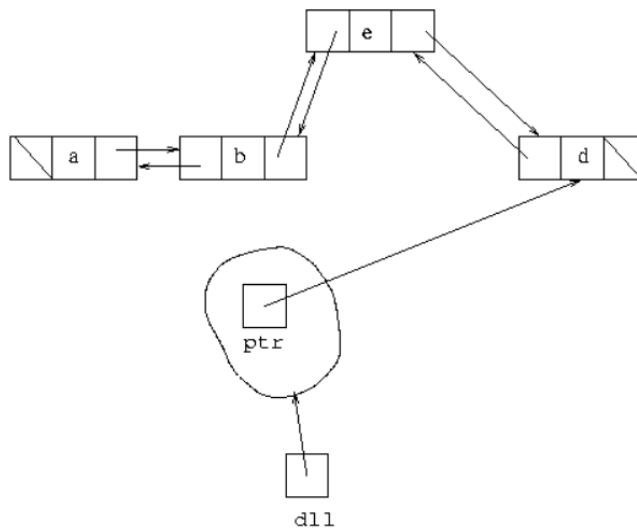
To explain further, this is how it looks like for a doubly linked list with a pointer that shows where the cursor is currently at:



the user can move around back and forth due to it being linked twice, and whenever a new element is inserted, it creates a new cell which rearranges the pointers to point at the new element:

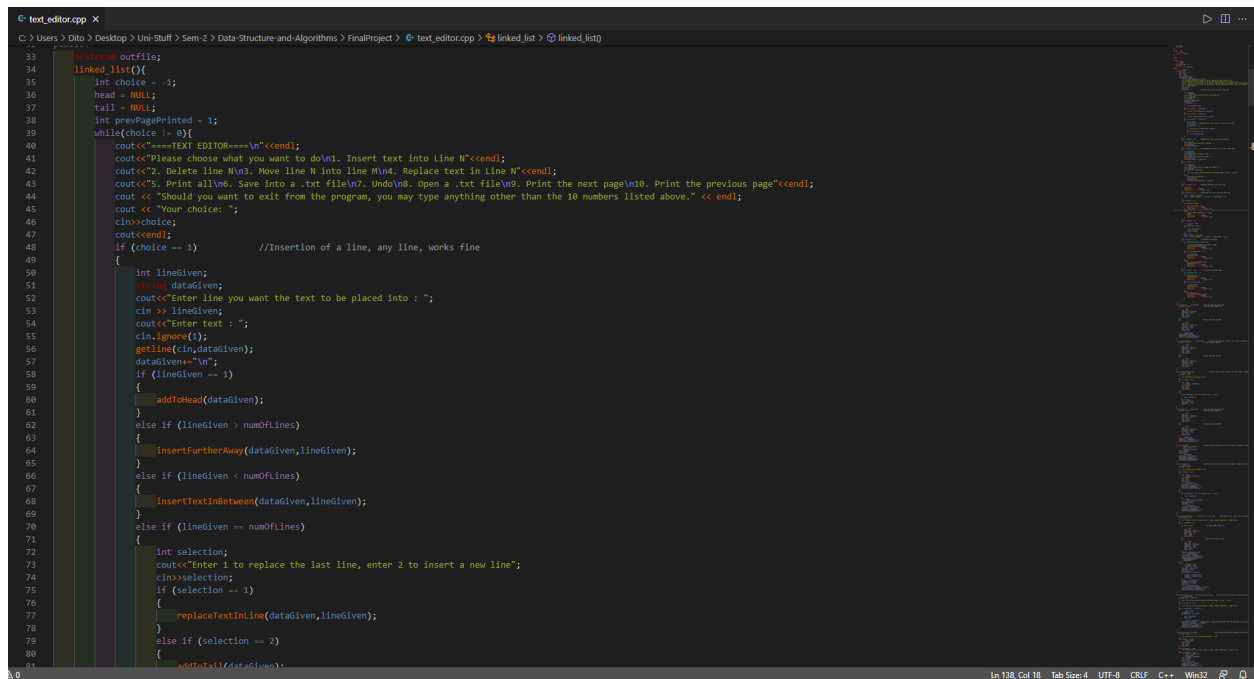


When an element is deleted, the pointer will then move to the nearest element and rearrangement happens:



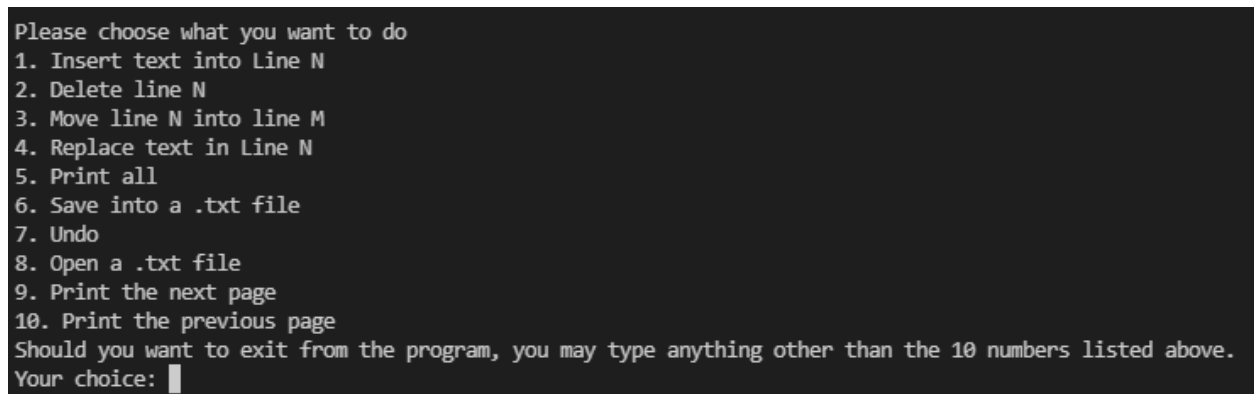
That is the basic gist of how doubly linked lists work in the text editor.

Program Manual



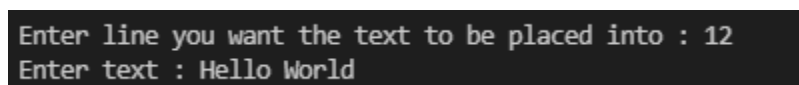
```
33 // Remove outfile;
34 linked_list()
35 {
36     int choice = -1;
37     head = NULL;
38     tail = NULL;
39     int prevPagePrinted = 1;
40     while(choice != 0)
41     {
42         cout<<"====TEXT EDITOR====\n"<<endl;
43         cout<<"Please choose what you want to do\n1. Insert text into Line N"<<endl;
44         cout<<"2. Delete line N\n3. Move line N into line M\n4. Replace text in Line N"<<endl;
45         cout<<"5. Print all\n6. Save into a .txt file\n7. Undo\n8. Open a .txt file\n9. Print the next page\n10. Print the previous page"<<endl;
46         cout<<"Should you want to exit from the program, you may type anything other than the 10 numbers listed above." << endl;
47         cout<<"Your choice: ";
48         cin>>choice;
49         cout<<endl;
50         if(choice == 1) //Insertion of a line, any line, works fine
51         {
52             int lineGiven;
53             string dataGiven;
54             cout<<"Enter line you want the text to be placed into : ";
55             cin>>lineGiven;
56             cout<<"Enter text : ";
57             cin.ignore(1);
58             getline(cin,dataGiven);
59             dataGiven+="\n";
60             if (lineGiven == 1)
61             {
62                 addtoHead(dataGiven);
63             }
64             else if (lineGiven > numOfLines)
65             {
66                 insertFurtherAway(dataGiven,lineGiven);
67             }
68             else if (lineGiven < numOfLines)
69             {
70                 insertTextInBetween(dataGiven,lineGiven);
71             }
72             else if (lineGiven == numOfLines)
73             {
74                 int selection;
75                 cout<<"Enter 1 to replace the last line, enter 2 to insert a new line";
76                 cin>>selection;
77                 if (selection == 1)
78                 {
79                     replaceTextInLine(dataGiven,lineGiven);
80                 }
81                 else if (selection == 2)
82                 {
83                     addDataAtTail(dataGiven);
84                 }
85             }
86         }
87     }
88 }
```

To start, open the text_editor.cpp file with an IDE that supports C++ and run it. Above is an example of opening the code in Visual Studio Code (using a CodeRunner extension).



```
Please choose what you want to do
1. Insert text into Line N
2. Delete line N
3. Move line N into line M
4. Replace text in Line N
5. Print all
6. Save into a .txt file
7. Undo
8. Open a .txt file
9. Print the next page
10. Print the previous page
Should you want to exit from the program, you may type anything other than the 10 numbers listed above.
Your choice: █
```

A menu like the one listed above should appear. Here, we can do multiple things with the program. To do a certain action, simply input the number corresponding to the action to the user input (note: please do not input any string values for the numeric inputs as it will cause the program to loop out of control. If that happens, terminate the command line/terminal immediately).



```
Enter line you want the text to be placed into : 12
Enter text : Hello World
```

Inserting (input number 1) and replacing text (input number 4) into a certain line yields the same set of input. First, input the line and then, enter the text of the line.

```
Enter the line you want to delete : 12
```

To delete lines (input number 2), just input the specific line which will be deleted.

```
Enter line 1 you want to swap : 1
Enter line 2 you want to swap : 2
```

Moving line N to line M (input number 3) will receive 2 separate inputs, that being the two target lines. Just type the line numbers of the target lines and the text in the lines will be swapped.

```
-----Page 1-----
1) Hello world
```

Printing the page (input number 5), printing the next page (input number 9), and printing the previous page (input number 10) will receive no additional input. It will just print the contents of the linked list (for printing the previous page and next page, it will only show particular lines of the linked list). Above is an example of it printing contents after inserting "Hello world" to line 1.

```
Enter the file name : |
```

For both saving (input number 6) and opening (input number 8), it will ask for a file name. Here, there is no need to input the .txt extension as our code will do that automatically. For saving, if the file doesn't exist yet, it will create a new text file. If a non-existent text file is opened, the program will set the contents to nothing.

```
Your choice: 7
Added To head, removing from head...
```

Undo (input number 7) will not receive any additional inputs. If it is activated, the program will state the action that has been done and then undo it.

Explanation of Code

Above is the node structure of our program. It's quite standard, containing a data which in this case is a string and an address of another node which is the next node. Each node in this case will represent one line and the data will represent the contents of that line.

```

struct undoCmd{

    int lineNumber;

    string text;

    int commandNumber;

    int mLine;

    int nLine;

};

```

The undoCmd is a custom data type that stores information about what undo should do. Every function has its own undoCmd object that corresponds to a particular commandNumber. The undo function will do something depending on what commandNumber the undoCmd is set to. The lineNumber stores the line where the text has been inserted/deleted initially. The string text will store whatever has been inserted/deleted so that it may be deleted/brought back.

```

class linked_list
{
private:

    node *head;

    node *tail;

    int numOfLines = 0;

    int next = 1;

    stack<undoCmd> undoStack;

```

Here are the variables on our linked list. Our linked list is a custom linked list specifically designed for this text editor. It contains the code that calls for the user's input and all the

functions for the text editor, meaning that once this linked list has been initiated, it will immediately start up the command line interface. Here, we have two nodes, one being the head and the other being the tail. The head represents the first line of the text file while the tail represents the last. There is also an int called numOfLines which basically tracks how many lines we have in our text file. We also have a stack of undoCmds named undoStack which stores the undoCmd data for undoing actions. A stack is used for this feature because of the stack's inherent last in first out feature, meaning the most recent change is the one that will be undone.

Insertion

```
node *prevNode = head;

node *nextNode = head;

node *temp = new node();

temp->data = dataGiven;

temp->next = NULL;

int iterator = 2;

while(iterator < lineGiven)

{

    prevNode = prevNode->next;

    nextNode = nextNode->next;

    iterator++;

}

nextNode = nextNode->next;

prevNode->next = temp;

temp->next = nextNode;

numOfLines++;
```

```
undoCmd insertedInBetween;  
  
insertedInBetween.lineNumber = lineGiven;  
  
insertedInBetween.commandNumber = 6;  
  
undoStack.push(insertedInBetween);
```

Sample code that inserts text in a given line where the line is located in-between the head and tail.

The code above shows an example of how we insert data to the linked list. Here we make a node called temp to temporarily store the data we want to insert and then set the data of that node to what we want to insert. Then, we call on an int variable called iterator which will iterate through the linked list until the iterator is less than the line given. Then, we simply set the node next to the prevNode (representing the node before the target line) to our temp node and with that, the text has been inserted. The data is then put into an undoCmd with a given commandNumber (depending on what line is deleted) and then that undoCmd object is pushed to the stack.

Deletion

```
undoCmd deletedLine;  
deletedLine.commandNumber = 10;  
node *prevNode = head;  
node *nextNode = head;  
node *temp = head;  
int iterator = 2;  
while(iterator < lineGiven)  
{  
    prevNode = prevNode->next;  
    nextNode = nextNode->next;  
    iterator++;  
}  
nextNode = nextNode->next;  
temp = nextNode;  
nextNode = nextNode->next;  
prevNode->next = nextNode;  
string backup = temp->data;  
delete(temp);  
numOfLines--;
```

```
deletedLine.text = backup;
deletedLine.lineNumber = lineGiven;
undoStack.push(deletedLine);
```

Sample code that deletes the text in a line given that line is in-between the head and tail.

The deletion follows a similar process with using the iterators to get the specific line. However, instead of setting temp to a value, it deletes it entirely using delete() which will free up the memory space that temp takes up. However, before deletion, the string data within temp is stored on a backup string variable so that its contents can be put into an undoCmd.

Undo

```
undoCmd temp = undoStack.top();
```

```
else if (temp.commandNumber == 6) // Corresponds to inserting
a text that is in-between the head and tail
{
    cout<<"Inserted in between, removing that line..."<<endl;
    deleteLine(temp.lineNumber);
    undoStack.pop();
}
```

This is where the undoStack is used. temp here represents the most recent action that the user did and depending on the commandNumber of temp, the undo function will do something different. For instance, commandNumber 6 corresponds to inserting a text in a line given that the line is in-between the head and tail. Knowing this, what the undo function will do is delete that specific line by getting the lineNumber attribute of temp and then after the action is undone, the top of the stack (temp) will be popped.

It's also worth mentioning that the undo function cannot properly undo opening a file because our undoCmd class is not made to store data from multiple lines, hence opening a file with multiple lines cannot be undone. The undo function will only delete the tail (last line) of the text if the top of the undoStack (the last action done) corresponds to opening a file.

Saving

```
ofstream outfile;
```

```
double saveAll() {
    node *temp = head;
    int linePrinted = 1;
    int pagePrinted = 2;
```



```

    string fileName;
    cout<<"Enter the file name : ";
    cin>>fileName;
    fileName+=" .txt";
    outfile.open(fileName, ios_base::app);
    auto start = steady_clock::now();
    while(temp!=NULL)
    {
        outfile<<temp->data;
        temp = temp->next;
        linePrinted++;
    }
    outfile.flush();
    outfile.close();
    auto end = steady_clock::now();

    double elapsedTime =
double(duration_cast<nanoseconds>(end-start).count());
    return elapsedTime;

}

```

Saving an object is done via an ofstream object which allows our program to write in an opened text file. First, a temp node is set up that acts as a copy of our linked list. Then, the user enters the file name. It is then opened by our ofstream object where while the temp variable is not NULL (meaning that there are still lines left to be added) it will put our linked list data line-by-line.

Opening Files

```

double openFile() { //function used to open a
file from the same folder this cpp file is in
    string fileName;
    cout<<"Enter the file name : ";
    cin>>fileName;
    fileName+=" .txt";
    ifstream myfile;
    myfile.open(fileName);
    string s;

```

```

        auto start = steady_clock::now();
        while(getline(myfile,s))
        {
            addToTail(s);
        }
        myfile.close();
        auto end = steady_clock::now();

        double elapsedTime =
double(duration_cast<nanoseconds>(end-start).count());
        return elapsedTime;
    }

```

For this operation, ifstream is used to open the file and read its contents. The getline function is called to get the contents of each line in the .txt file where the line will be inserted to the tail of our linked list.

Both our save function and open function would normally return nothing. However, since we want to calculate the runtime of both operations, we let our functions return a double representing the runtime in nanoseconds of these operations.

Results and Testing

Our tests were done in a laptop with the following specifications:

- CPU: AMD Ryzen 7 4800H (8 cores, 16 threads with 2.9 GHz base clock speed)
- Memory: 16 GB (3200 MHz clock speed)

To minimize the external factors that could affect our testing, there were no applications running other than our program and an excel sheet to note down our results.

We measured the runtimes of opening and saving files because we wanted to see how effective the program is at processing multiple lines. Most of the insertion options in the program can only insert data on one line and therefore, there isn't any significant increase in runtime when more characters are added.

```

Your choice: 8

Enter the file name : hundredthousandchar
Time to execute: 1.7592e+06 nanoseconds
====TEXT EDITOR====

```

When opening or saving the file, it should print the time it took to execute said function like shown above (note: the above image was done with many background applications running so it's not an accurate representation of the runtime of the function).

To measure the runtime of these functions, we used `steady_clock` from `std::chrono` which acts as a timer that can calculate the amount of time it took to execute particular lines of code. We also made it so that `steady_clock` only measures the runtime of specific lines of the function relating to its core processes. This ensures that our runtime is as accurate as possible since these functions also ask for the user's input.

Sample Information

For our sample size, we decided to do a test for opening on saving with 1000 characters (10 lines), 10000 characters (100 lines), 50000 characters (500 lines), and 100000 characters (1000 lines). The reason why we chose that specific sample size is because there already was a significant increase in runtime when going from 10000 characters to 100000 characters, hence why we didn't go for larger sample sizes. Also, putting the sample sizes closer to each other allows for us to create a more accurate graph for our runtimes. For each sample size, five trials are conducted to account for any processing errors that could happen and the average of the trials is used for graphing.

Opening Files

```
auto start = steady_clock::now();
    while(getline(myfile,s))
    {
        addToTail(s);
    }
    myfile.close();
    auto end = steady_clock::now();

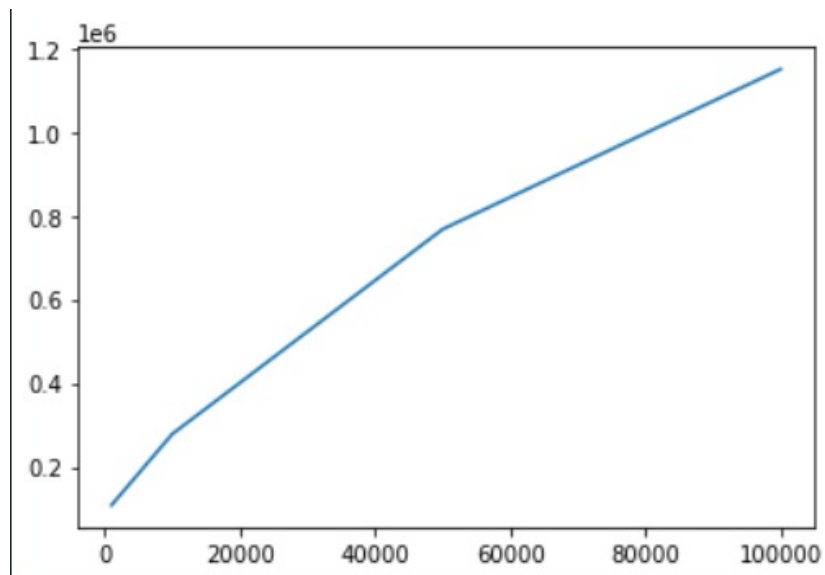
double elapsedTime =
double(duration_cast<nanoseconds>(end-start).count());
    return elapsedTime;
```

_____For opening files, the runtime is measured when the function starts getting lines and putting them to the tail of our linked list one-by-one. The timer ends once the opened file has been closed by the program. This shows how quickly our text editor can read text files.

Opening Files							
		Runtime (nanoseconds)					
Number of characters	Number of Lines	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1000	10	91200	98600	101400	136000	114500	108340
10000	100	200300	208100	364800	225900	393100	278440
50000	500	858800	757500	587600	640300	993800	767600
100000	1000	1,08E+06	1,50E+06	1,02E+06	1,07E+06	1,08E+06	1,15E+06

Excel spreadsheet showing the results we got from our testing.

We can see here that as our sample size gets larger, there is a clear increase in runtime. It is seen here that the runtime is not always the same and can vary, hence why running multiple trials is necessary.



Plotting the number of characters to the average of our trials. The y-axis represents the runtime in nanoseconds while the x-axis represents the number of characters.

Graphing our data, it looks as if the time complexity for opening files is logarithmic, although it looks as if the data is leaning towards a linear curve, especially towards the end of the graph.

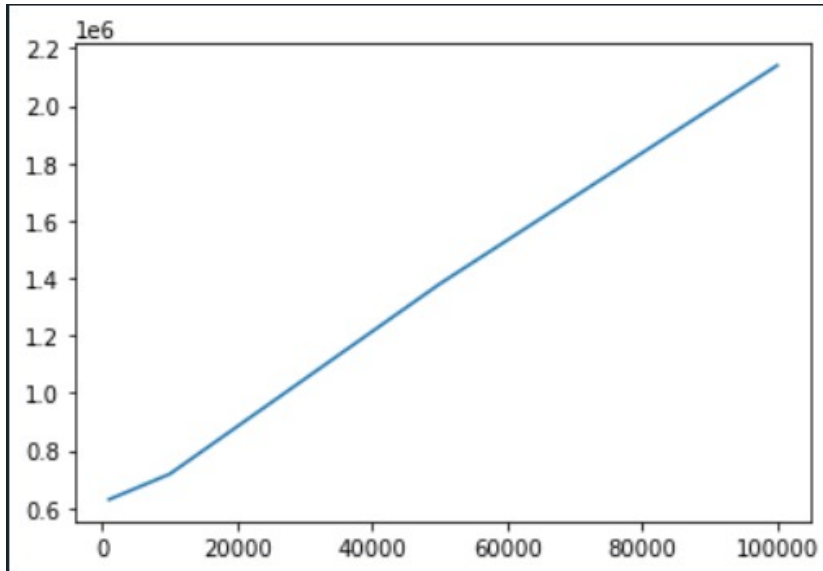
Saving Files

```
auto start = steady_clock::now();
while(temp!=NULL)
{
    outfile<<temp->data;
    temp = temp->next;
    linePrinted++;
}
outfile.flush();
outfile.close();
auto end = steady_clock::now();
```

For saving files, the timer starts when the lines within the .txt files are starting to get inserted one by one and the timer stops when the file is closed by the program. This represents how quick our text editor is at writing data into a text file.

Saving Files							
		Runtime (nanoseconds)					
Number of characters	Number of Lines	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
1000	10	516000	561800	612600	608700	850500	629920
10000	100	679300	778300	754700	660600	719500	718480
50000	500	1,34E+06	1,75E+06	1,31E+06	1,28E+06	1,21E+06	1,38E+06
100000	1000	2,25E+06	2,16E+06	2,17E+06	2,07E+06	2,03E+06	2,14E+06

Here, it is seen that the runtime for saving files is generally larger than that of opening files. This is to be expected as the program has to write the text instead of simply reading it.



Plotting the number of characters to the average of our trials. The y-axis represents the runtime in nanoseconds while the x-axis represents the number of characters.

Here, it is seen that saving files has a linear time complexity. It seems to start with less of a gradient but as the number of characters grows larger, the gradient seems to be constant.

Conclusion

Implementing a text editor using linked lists proves to be sufficiently fast even as the number of characters gets larger but overall, there are still better data structures for implementing text editors. For instance, the rope data structure has a reported time complexity of approximately $\theta(\log n)$ (Kalra, n.d.). This is slower than the text editor used for this project which, from testing the saving function, can insert text to files with a linear time complexity. It is worth mentioning however that the rope data structure struggles with smaller text files due to its logarithmic time complexity (Kalra, n.d.). This means that our text editor is good when the number of characters to be inserted is unknown because it does just fine in both larger and smaller text files, even if it may not be the best of both worlds.

Demo Link

https://drive.google.com/file/d/1faqsMawHcL-Z7c_L9-hMplIBSeSLUMqA/view?usp=sharing

GitHub Link

https://github.com/AlifsyahRS/DSAFinalProject_TextEditor

References

Booker, E.Z. (n.d.). "Data structures used in text editors". *OpenGenus*. Retrieved from <https://iq.opengenus.org/data-structures-used-in-text-editor/>

Huntbach M. (12 March 2004) *Doubly linked lists: the Editor example*. Queen Mary, University of London.

<http://www.eecs.qmul.ac.uk/~mmh/DCS128/2006/resources/doublylinkedlists.html>

Kalra, I.S. (n.d.). "Rope: the Data Structure used by text editors to handle large strings". *OpenGenus*. Retrieved from <https://iq.opengenus.org/rope-data-structure/>