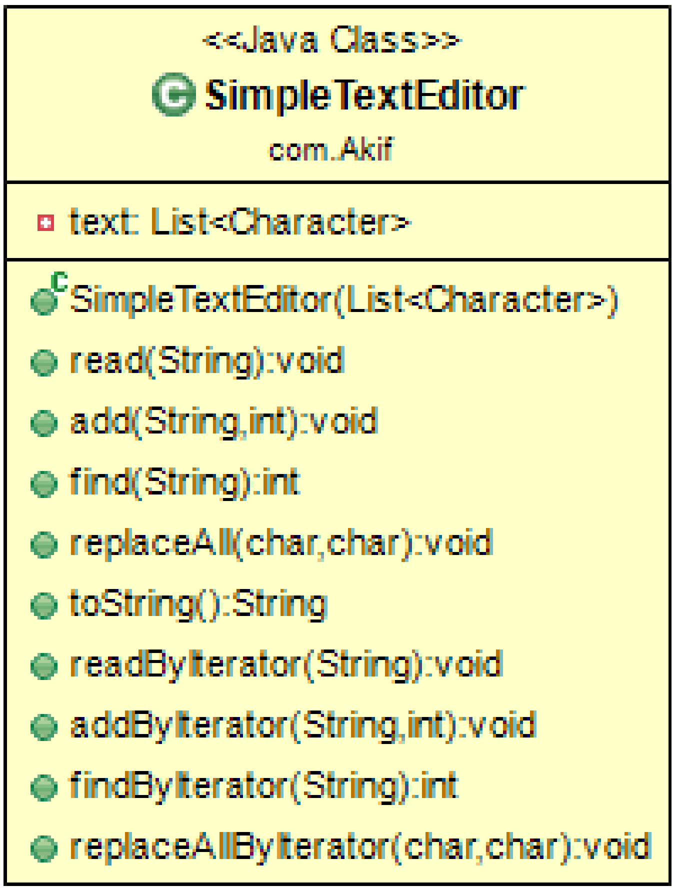
<u>REPORT</u>

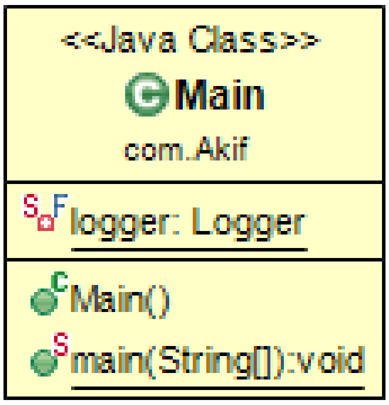
1. SYSTEM REQUIREMENTS

To test this program you need to have 3 file. These are akif.txt , vectors.txt, listInterface.txt in the same directory with source code.

2. CLASS DIAGRAMS



<<Java Class>>
SimpleTextEditor
com.Akif
text: List<Character>
SimpleTextEditor(List<Character>)
read(String):void
add(String,int):void
find(String):int
replaceAll(char,char):void
toString():String
readByIterator(String):void
addByIterator(String,int):void
findByIterator(String):int
replaceAllByIterator(char,char):void



3. PROBLEM SOLUTION APPROACH

My Problem solution steps are;

– Specify the problem requirements
– Analyze the problem
– Design an algorithm and Program
– Implement the algorithm
– Test and verify the program
– Maintain and update the program

1) **Specify the problem requirements :** I understand the problem.

2) **Analyze the problem :** I identify;

   – Input data

   – Output data

   – Additional requirements and constraints

3) **Design an algorithm and Program :** I divide the problem into sub-problems. I listed

   major steps (sub-problems). I break down each step into a more detailed list. To do these We have to divide this big project into small pieces.

   Implement the algorithm : I wrote the algorithm in Java by converting each step into statements of Java (classes ,methods etc.)

   First I started to write SimpleTextEditor class as expected. Then ,In this class I wrote all necessary methods separately and clear.

   To read a file I used file class and file reader class of java. Actually in each method I used java programming language tools such as iterator, add method, get method etc.

   I wrote each method as two types as by using the iterator (ListIterator) to navigate on the List and without using the iterator, using simple for loop with index structure.

4) **Test and verify the program:** To test program I wrote the Main class in this class in main method I created a log file and wrote test results to this file. To test my all method I used 3 different file with different sizes. By using these files I tested all methods and print the results in “test.log” file includes running times.

5) **Maintain and update the program :** I keep the program up-to-date.

4. TEST CASES

Test ID	Test Case	Pre-Condition	Pass/Fail - ArrayList/LinkedList
T1	Test read method without iterator	- There should be a txt file (e.g. akif.txt)	**Pass / Pass**
T2	Test add method without iterator	- List should be ready to add - There should be abstractList.txt file to add	**Pass / Pass**
T3	Test find method without iterator	- List should be ready to find	**Pass / Pass**
T4	Test replace method without iterator	- List should be ready to replace	**Pass / Pass**
T5	Test read method with iterator	- There should be a txt file (e.g. akif.txt)	**Pass / Pass**
T6	Test add method with iterator	- List should be ready to add - There should be abstractList.txt file to add	**Pass / Pass**
T7	Test find method with iterator	- List should be ready to find	**Pass / Pass**
T8	Test replace method with iterator	- List should be ready to replace	**Pass / Pass**

5. RUNNING AND RESULTS

Test ID	Test Result
T1	Text: akif kartal
T2	**Text after add operation:** akiThis class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class. To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int) and size() methods. To implement a modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException). If the list is variable-size the programmer must additionally override the add(int, E) and remove(int) methods. The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification. Unlike the other abstract collection implementations, the programmer does not have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top of the "random access" methods: get(int), set(int, E), add(int, E) and remove(int). The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation. This class is a member of the Java Collections Framework.f kartal
T3	find(ali): -1
T4	**replaceAll(a,k):** kkiThis clkss provides k skeletkl implementktion of the List interfkce to minimize the effort required to implement this interfkce bkcked by k "rkndom kccess" dktk store (such ks kn krrky). For sequentikl kccess dktk (such ks k linked list), AbstrkctSequentiklList should be used in preference to this clkss. To implement kn unmodifikble list, the progrkmmer needs only to extend this clkss knd provide implementktions for the get(int) knd size() methods. To implement k modifikble list, the progrkmmer must kdditionklly override the set(int, E) method (which otherwise throws kn UnsupportedOperktionException). If the list is vkrikble-size the progrkmmer must kdditionklly override the kdd(int, E) knd remove(int) methods.

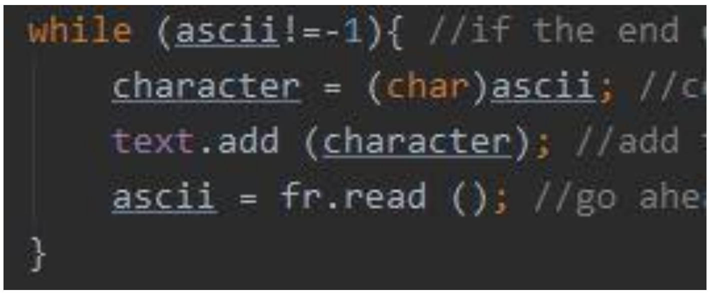
	The progrkmmer should generklly provide k void (no krgument) knd collection constructor, ks per the recommendktion in the Collection interfkce specificktion. Unlike the other kbstrkct collection implementktions, the progrkmmer does not hkve to provide kn iterktor implementktion; the iterktor knd list iterktor kre implemented by this clkss, on top of the "rkndom kccess" methods: get(int), set(int, E), kdd(int, E) knd remove(int). The documentktion for ekch non-kbstrkct method in this clkss describes its implementktion in detkil. Ekch of these methods mky be overridden if the collection being implemented kdmits k more efficient implementktion. This clkss is k member of the Jkvk Collections Frkmework.f kkrtkl
T5	Text: akif kartal
T6	**Text after add operation:** akiThis class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class. To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int) and size() methods. To implement a modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException). If the list is variable-size the programmer must additionally override the add(int, E) and remove(int) methods. The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification. Unlike the other abstract collection implementations, the programmer does not have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top of the "random access" methods: get(int), set(int, E), add(int, E) and remove(int). The documentation for each non-abstract method in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation. This class is a member of the Java Collections Framework.f kartal
T7	find(ali): -1

T8	**replaceAll(a,k):** kkiThis clkss provides k skeletkl implementktion of the List interfkce to minimize the effort required to implement this interfkce bkcked by k "rkndom kccess" dktk store (such ks kn krrky). For sequentikl kccess dktk (such ks k linked list), AbstrkctSequentiklList should be used in preference to this clkss. To implement kn unmodifikble list, the progrkmmer needs only to extend this clkss knd provide implementktions for the get(int) knd size() methods. To implement k modifikble list, the progrkmmer must kdditionklly override the set(int, E) method (which otherwise throws kn UnsupportedOperktionException). If the list is vkrikble-size the progrkmmer must kdditionklly override the kdd(int, E) knd remove(int) methods. The progrkmmer should generklly provide k void (no krgument) knd collection constructor, ks per the recommendktion in the Collection interfkce specificktion. Unlike the other kbstrkct collection implementktions, the progrkmmer does not hkve to provide kn iterktor implementktion; the iterktor knd list iterktor kre implemented by this clkss, on top of the "rkndom kccess" methods: get(int), set(int, E), kdd(int, E) knd remove(int). The documentktion for ekch non-kbstrkct method in this clkss describes its implementktion in detkil. Ekch of these methods mky be overridden if the collection being implemented kdmits k more efficient implementktion. This clkss is k member of the Jkvk Collections Frkmework.f kkrtkl

Analysis of the performance of each method theoretically and experimentally

List is an ArrayList and iterator is not used.

1) read method



```
while (ascii!=-1){ //if the end
    character = (char)ascii; //c
    text.add (character); //add
    ascii = fr.read (); //go ahe
}
```

In this method as in picture we have a while loop to read all file. In while loop we have 3 statements that two of them have constant running time. If we look at the text.add statement this statement has constant running time in average case so according to these information we have three case in this method because of arrayList add method.

So total running time;

$T(n)_{best} = O(n)$ (while loop has O(n), add has constant time)

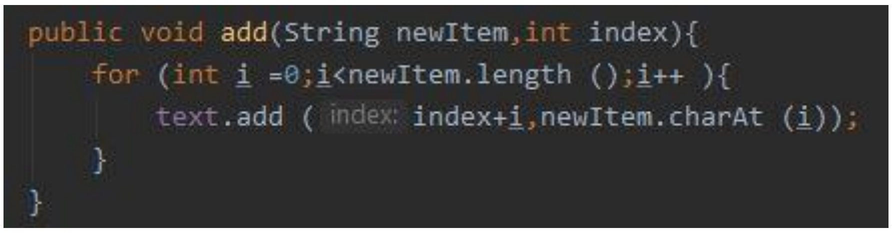
$T(n)_{worst} = O(n^2)$ (while loop*add, add has O(n) time because of reallocate)

$T(n)_{average} = O(n)$ (while loop has O(n), add has constant time in average case)

<u>Experimental running time from “test.log” file;</u>

read method running time(millisecond): 47

2) add method



```java
public void add(String newItem,int index){
    for (int i =0;i<newItem.length ();i++ ){
        text.add ( index: index+i,newItem.charAt (i));
    }
}
```

In this method as in picture we have a for loop to iterate all string which is parameter. In for loop we have 1 statement that it has always O(n) running time because arrayList add(index,item) method has a loop to shift elements.

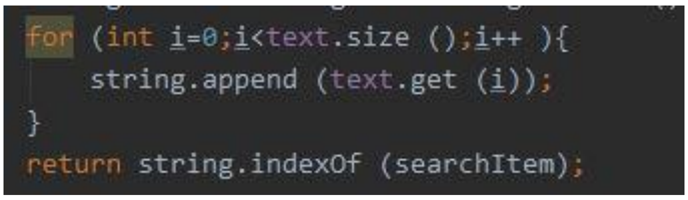
So total running time;

$T(n) = O(n^2)$ (for loop*add, add has O(n) time to shift elements)

<u>Experimental running time from “test.log” file;</u>

add method running time(millisecond): 2

3) find method



```java
for (int i=0;i<text.size ();i++ ){
    string.append (text.get (i));
}
return string.indexOf (searchItem);
```

In this method as in picture we have a for loop to iterate all text that we have. In for loop we have 1 statement. This statement is text.get() statement has ϴ(1) running time because arraylist provides a fast index-based access. In last part we have indexOf method that has O(n) time since it searches the element.

So total running time;

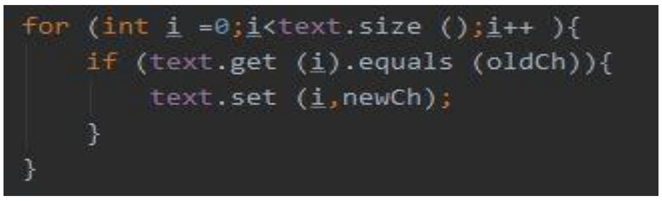
$T(n) = O(n)$ (for loop has O(n), get has O(1) time, append has O(1) time(assumption))

*IndexOf method doesn’t affect our result since it is not in the loop. (O(2n) = O(n))

Experimental running time from "test.log" file;

find method running time(millisecond): 68

4) replace method



```
for (int i =0;i<text.size ();i++ ){
    if (text.get (i).equals (oldCh)){
        text.set (i,newCh);
    }
}
```

In this method as in picture we have a for loop to iterate all text that we have. In for loop we have 1 statement. This statement is text.set() has Ɵ(1) running time because arraylist provides a fast index-based access.

So total running time;

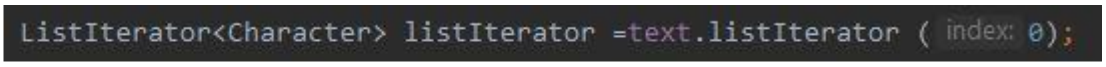
T(n) = O(n) (for loop has O(n), set has O(1) time, if statement doesn't affect the case)

Experimental running time from "test.log" file;

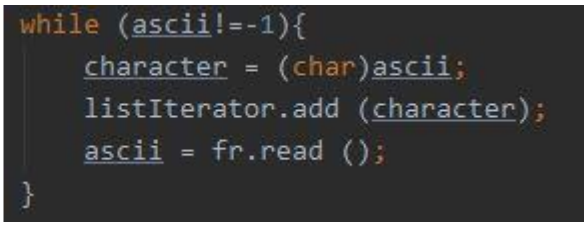
replace method running time(millisecond): 2

List is an ArrayList and iterator is used

1) read method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
```



```
while (ascii!=-1){
    character = (char)ascii;
    listIterator.add (character);
    ascii = fr.read ();
}
```

In this method as in first picture we have constructor of iterator since we start with 0th index we have constant time. In second picture we have a while loop to read all file. In while loop we have 3 statements that two of them have constant running time. If we look at the listIterator.add statement this statement has constant running time in average case so according to these information we have three case in this method because of listiterator add method

So total running time;

$T(n)_{best} = O(n)$ (while loop has O(n), add has constant time)

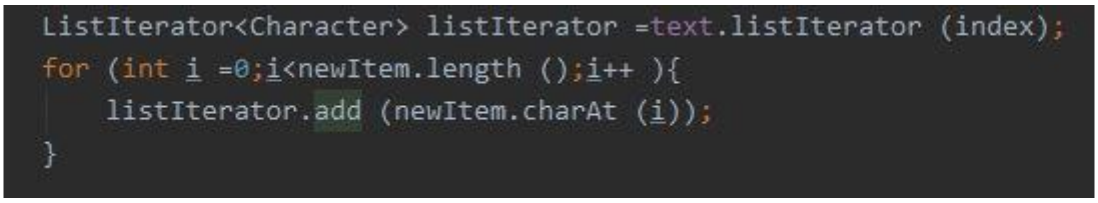
$T(n)_{worst} = O(n^2)$ (while loop*add, add has O(n) time because of reallocate(iterator uses original add method in arraylist)

$T(n)_{average} = O(n)$ (while loop has O(n), add has constant time in average case)

Experimental running time from “test.log” file;

read method running time(millisecond): 56

2) add method



```
ListIterator<Character> listIterator =text.listIterator (index);
for (int i =0;i<newItem.length ();i++ ){
    listIterator.add (newItem.charAt (i));
}
```

In this method as in first part we have constructor of iterator since we start with arbitrary index we have O(n) time to reach that index. In second part we have a for loop to iterate all string which is parameter. In for loop we have 1 statement that it has constant running time because since iterator uses original arrayList add method.

So total running time;

$T(n)_{best} = O(n)$ (for loop has O(n), add has constant time)

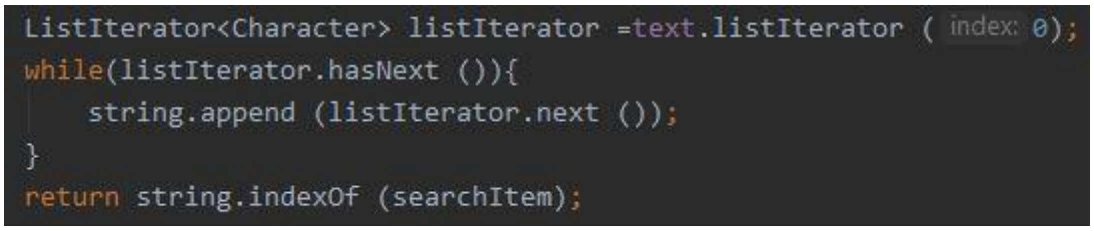
$T(n)_{worst} = O(n^2)$ (for loop*add, add has O(n) time because of reallocate(iterator uses original add method in arraylist)

$T(n)_{average} = O(n)$ (while loop has O(n), add has constant time in average case)

Experimental running time from “test.log” file;

add method running time(millisecond): 3

3) find method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
while(listIterator.hasNext ()){
    string.append (listIterator.next ());
}
return string.indexOf (searchItem);
```

In this method as in first part we have constructor of iterator since we start with 0th index we have O(1) time to reach that index. In second part we have a while loop to iterate all string that we have. In while loop we have 1 statement that it has constant running time. In third part we have indexOf method that has O(n) time since it searches the element.

So total running time;

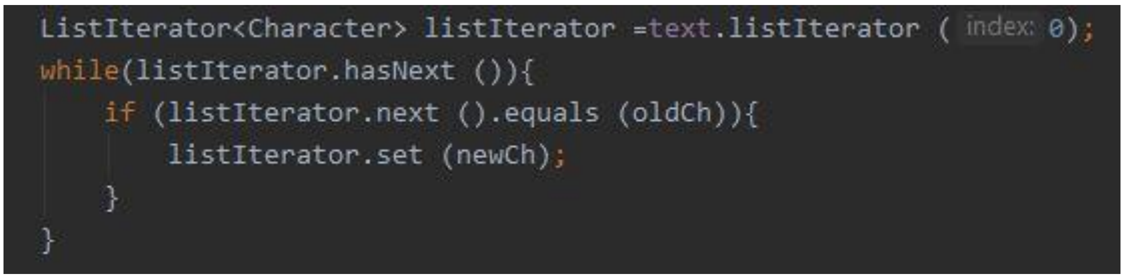
T(n) = O(n) (while loop has O(n) time, next has O(1) time, append has O(1) time(assumption))

*IndexOf method doesn't affect our result since it is not in the loop. (O(2n) = O(n))

Experimental running time from "test.log" file;

find method running time(millisecond): 5

4) replace method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
while(listIterator.hasNext ()){
    if (listIterator.next ().equals (oldCh)){
        listIterator.set (newCh);
    }
}
```

In this method as in first part we have constructor of iterator since we start with 0th index we have O(1) time to reach that index. In second part we have a while loop to iterate all string that we have. In while loop we have 1 statement that it has constant running time

So total running time;

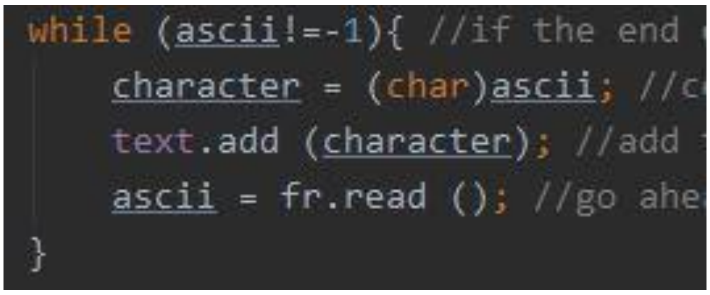
T(n) = O(n) (while loop has O(n) time, set has O(1) time, if statement doesn't affect the case)

Experimental running time from "test.log" file;

replace method running time(millisecond): 1

List is a LinkedList and iterator is not used.

1) read method



```java
while (ascii!=-1){ //if the end
    character = (char)ascii; //c
    text.add (character); //add
    ascii = fr.read (); //go ahe
}
```

In this method as in picture we have a while loop to read all file. In while loop we have 3 statements that two of them have constant running time. If we look at the text.add statement, this statement has always constant running time because LinkedList add has always O(1) running time since it adds end of the list.

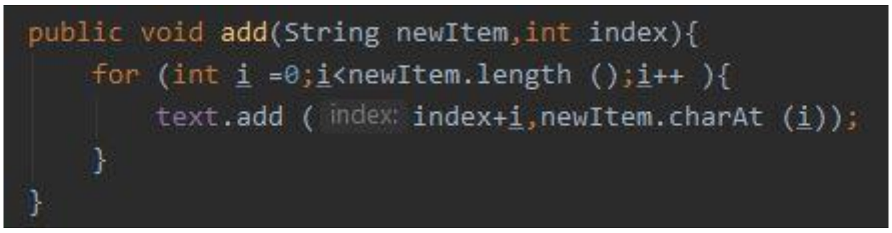
So our running time;

$T(n) = O(n)$ (while loop has O(n), add has O(1) time)

<u>Experimental running time from "test.log" file;</u>

read method running time(millisecond): 4

2) add method



```java
public void add(String newItem,int index){
    for (int i =0;i<newItem.length ();i++ ){
        text.add ( index: index+i,newItem.charAt (i));
    }
}
```

In this method as in picture we have a for loop to iterate all string which is parameter. In for loop we have 1 statement that it has always O(n) running time because linkedList add(index,item) method has a loop to reach that index. LinkedList does only memorize the head and tail index.

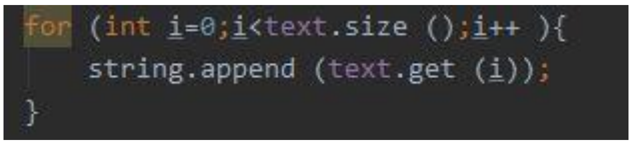
So general running time;

$T(n) = O(n^2)$ (for loop*add, add has O(n) time to reach that index)

<u>Experimental running time from "test.log" file;</u>

add method running time(millisecond): 9

3) find method



```
for (int i=0;i<text.size ();i++ ){
    string.append (text.get (i));
}
```

In this method as in picture we have a for loop to iterate all text that we have. In for loop we have 1 statement. This statement is text.get() statement(we assume string.append() method has constant running time()) has Θ(n) running time for a linked list because LinkedList does only memorize the head and tail. Finding an element in a LinkedList is always slow.

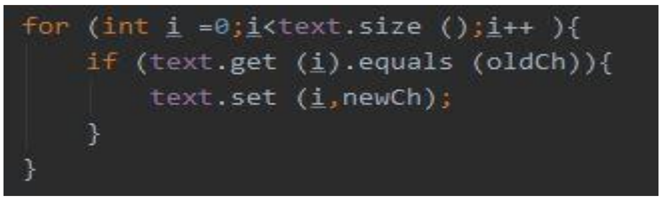
So general running time;

$T(n) = O(n^2)$ (for loop*get(), append has O(1) time(assumption))

<u>Experimental running time from "test.log" file;</u>

find method running time(millisecond): 189

4) replace method



```
for (int i =0;i<text.size ();i++ ){
    if (text.get (i).equals (oldCh)){
        text.set (i,newCh);
    }
}
```

In this method as in picture we have a for loop to iterate all text that we have. In for loop we have 1 statement. This statement is text.set() has Θ(n) running time for a linked list because LinkedList does only memorize the head and tail. Finding an element in a LinkedList is always slow.

So general running time;

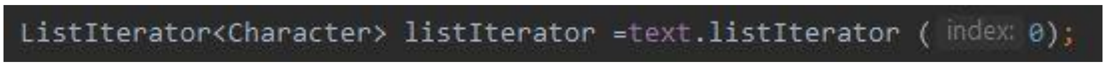
$T(n) = O(n^2)$ (for loop*set, if statement doesn't affect the case)

<u>Experimental running time from "test.log" file;</u>

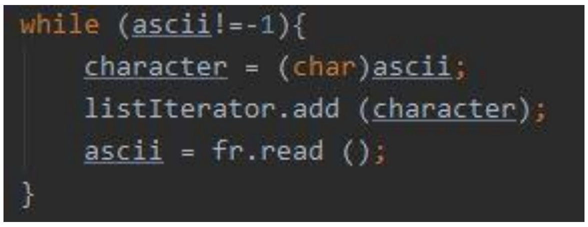
replace method running time(millisecond): 49

List is an LinkedList and iterator is used

1) read method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
```



```
while (ascii!=-1){
    character = (char)ascii;
    listIterator.add (character);
    ascii = fr.read ();
}
```

In this method as in first picture we have constructor of iterator since we start with 0th index we have constant time. In second picture we have a while loop to read all file. In while loop we have 3 statements that two of them have constant running time. If we look at the listIterator.add statement this statement statement has always constant running time because listIterator add has always O(1) running time(to reach end of the list).

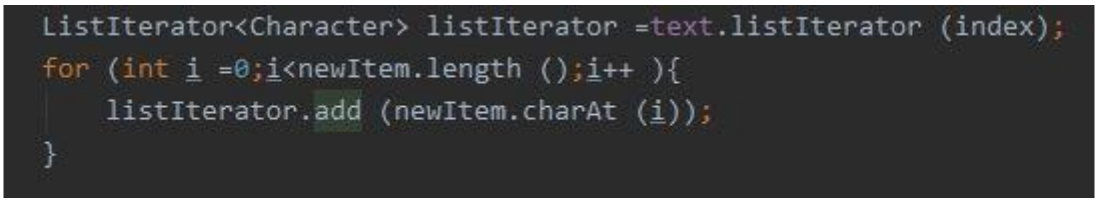
So our running time;

T(n) = O(n) (while loop has O(n), add has O(1) time)

<u>Experimental running time from “test.log” file;</u>

read method running time(millisecond): 36

2) add method



```
ListIterator<Character> listIterator =text.listIterator (index);
for (int i =0;i<newItem.length ();i++ ){
    listIterator.add (newItem.charAt (i));
}
```

In this method as in first part we have constructor of iterator since we start with arbitrary index we have O(n) time to reach that index. In second part we have a for loop to iterate all string which is parameter. In for loop we have 1 statement that it has constant running time since iterator add method has constant time.

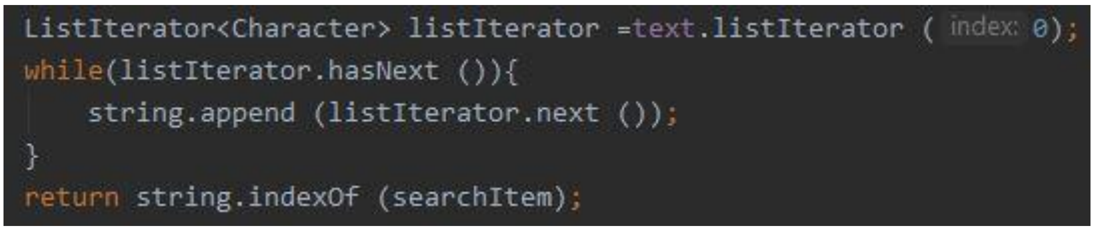
So general running time;

T(n) = O(n) (for loop has O(n), add has O(1) time)

<u>Experimental running time from “test.log” file;</u>

add method running time(millisecond): 0

3) find method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
while(listIterator.hasNext ()){
    string.append (listIterator.next ());
}
return string.indexOf (searchItem);
```

In this method as in first part we have constructor of iterator since we start with 0th index we have O(1) time to reach that index. In second part we have a while loop to iterate all string that we have. In while loop we have 1 statement that it has constant running time. In third part we have indexOf method that has O(n) time since it searches the element.

So total running time;

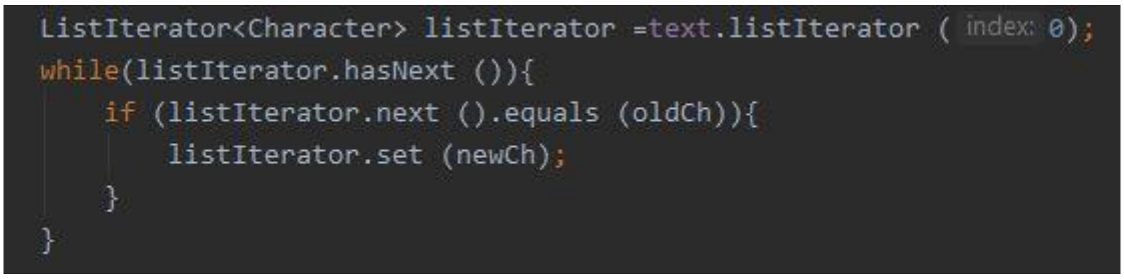
T(n) = O(n) (while loop has O(n), next has O(1) time, append has O(1) time)

*IndexOf method doesn't affect our result since it is not in the loop. (O(2n) = O(n))

<u>Experimental running time from "test.log" file;</u>

find method running time(millisecond): 2

4) replace method



```
ListIterator<Character> listIterator =text.listIterator ( index: 0);
while(listIterator.hasNext ()){
    if (listIterator.next ().equals (oldCh)){
        listIterator.set (newCh);
    }
}
```

In this method as in first part we have constructor of iterator since we start with 0th index we have O(1) time to reach that index. In second part we have a while loop to iterate all string that we have. In while loop we have 1 statement that it has constant running time

So total running time;

T(n) = O(n) (while loop has O(n) time, set has O(1) time)

* if statement doesn't affect the case.

<u>Experimental running time from "test.log" file;</u>

replace method running time(millisecond): 1