Q2

-Analysis of the performance of each method theoretically using the most

appropriate asymptotic notation.

List is a LinkedList and iterator is not used case:

```
public void readFile(String file_name)throws IOException {
   File my_file = new File( file_name ); -> constant
   Scanner reader = new Scanner(my_file); -> constant
   String current_line = new String(); -> constant
   while( reader.hasNextLine() ){ -> n lines
      current_line = reader.nextLine(); -> constant
      for( int i = 0; i<current_line.length(); ++i ) -> n chars in each line
         text.add(current_line.charAt(i)); -> constant for doubly linked list as it is added to the tail
      text.add('\n');
   }
}
```

-If we look at it with regards to the number of characters in our text file we can say its notation is o(n), or O(n^2)-because of getting lines, but small o notation is much more correct

```
public void add(String text, int index ){
   System.out.println("size is : " + this.text.size() );
   if( index >= 0 && index < this.text.size() ){
      for( int i = 0; i<text.length(); ++i, ++index ) -> n
         this.text.add( index, text.charAt(i) ); -> n
   }
   else
      throw new IndexOutOfBoundsException();
}
```

-Best notation for this is O(n^2) since we have a nested loop in here where add with index is perfomed on a linked list which has O(n) execution time

```
public int find(String text){
   int index = -1;
   boolean found = false;
   for( int i = 0; i<this.text.size() && !found; ++i ){-> linear
      found = true;
      index = i;
      for( int j = i, k = 0;j < this.text.size() && k < text.length() && found; ++j, ++k ){-> n times( linear )
         if( this.text.get(j) != text.charAt(k) ) {->n(linear) - getting is linear for linked list when we don't have reference to the node
            found = false;
            index = -1;
         }
      }
   }
   return index;
}
```

Since in here we have an inner loop its complexity would be O(n^2) and additional linear operation of hetting is perfomed giving

O(n^3) as this method's complexity

```
public void replace(Character ch, Character ch_new){
   for( int i = 0; i<text.size(); ++i ){ ->linear
      if( text.get(i) == ch )->linear
         text.set(i, ch_new);->linear
   }
}
```

Here we have a nested loop, get and set for linked list are linear so the complexity will be O(n^3)

List is a linkedList and iterator is used

```
public void readFile2(String file_name)throws IOException {
   File my_file = new File( file_name );
   Scanner reader = new Scanner(my_file);
   String current_line = new String();
   ListIterator<Character> itr = text.listIterator();
   while( reader.hasNextLine() ){-> n times( linear )
      current_line = reader.nextLine();
      for( int i = 0; i<current_line.length(); ++i ) -> n times
         itr.add(current_line.charAt(i)); -> constant time
      itr.add('\n');
   }
   reader.close();
}
```

This is the same as when there is not iterator, O(n^2) or o(n). Small o a bit more precise

```
public void add2(String text, int index ){
   System.out.println("size is : " + this.text.size() );
   ListIterator<Character> itr = this.text.listIterator(index);
   if( index >= 0 && index < this.text.size() ){
      for( int i = 0; i<text.length(); ++i, ++index ) { -> linear time, n
         itr.add(text.charAt(i)); -> constant time for the iterator
      }
   }
   else
      throw new IndexOutOfBoundsException();
}
```

This is a bit faster now and the execution time is O(n). Since it is both o(n) and O(n) we can see a tight bound and say this is $\Theta(n)$

```
public int find2(String text){
   int index = -1;
   boolean found = false;
   for( int i = 0; i<this.text.size() && !found; ++i ){ -> linear execution n
      found = true;
      index = i;
```

```
        ListIterator<Character> itr = this.text.listIterator(i);
        for( int k = 0;itr.hasNext() && k < text.length() && found; ++k ){ -> linear execution
            if( itr.next() != text.charAt(k) ) { -> constant
                found = false;
                index = -1;
            }
        }
    }
    return index;
}
```

We have a nested loop and execution for it is O(n^2). We can see it is faster than when it is not implemented with iterators.

```
public void replace2(Character ch, Character ch_new){
    ListIterator<Character> itr = this.text.listIterator();
    while(itr.hasNext()){ -> linear n
        if( itr.next() == ch ) -> constant
            itr.set(ch_new); -> constant
    }
}
```

Again this is faster, and its complexity is obviously always the same, linear $\Theta(n)$

List is an ArrayList and iterator is not used case:

```
public void readFile(String file_name)throws IOException {
    File my_file = new File( file_name ); -> constant
    Scanner reader = new Scanner(my_file); -> constant
    String current_line = new String(); -> constant
    while( reader.hasNextLine() ){ -> n lines
        current_line = reader.nextLine(); -> constant
        for( int i = 0; i<current_line.length(); ++i ) -> n chars in each line
            text.add(current_line.charAt(i)); -> constant for array list, but in worst case scenatrio it
would be linear( for reallocation)

        text.add('\n');
    }
}
```

-If we look at it with regards to the number of characters in our text file we can say its notation is o(n), or O(n^2)-because of getting lines, but small o notation is much more correct. Also here we regard addition as constant operation even though in worst case it would be linear yielding O(n^3), but yet again o(n) is the most correct one

```
public void add(String text, int index ){
    System.out.println("size is : " + this.text.size() );
    if( index >= 0 && index < this.text.size() ){
        for( int i = 0; i<text.length(); ++i, ++index ) -> n
            this.text.add( index, text.charAt(i) ); -> n
    }
    else
```

```
        throw new IndexOutOfBoundsException();
}
```

-Best notation for this is O(n^2) since we have a nested loop in here where add with index is perfomed on a arrayList which has O(n) execution time because of shifting and reallocation sometimes

```
public int find(String text){
    int index = -1;
    boolean found = false;
    for( int i = 0; i<this.text.size() && !found; ++i ){-> linear
        found = true;
        index = i;
        for( int j = i, k = 0;j < this.text.size() && k < text.length() && found; ++j, ++k ){-> n times( linear )
            if( this.text.get(j) != text.charAt(k) ) {->constant for arrays
                found = false;
                index = -1;
            }
        }
    }
    return index;
}
```

Since in here we have an inner loop its complexity would be O(n^2)

O(n^2) as this method's complexity

```
public void replace(Character ch, Character ch_new){
    for( int i = 0; i<text.size(); ++i ){ ->linear
        if( text.get(i) == ch )->constant
            text.set(i, ch_new);->constant
    }
}
```

Get and set for arraylist are constant so the complexity will be $\Theta(n)$

List is an ArrayList and iterator is used

```
public void readFile2(String file_name)throws IOException {
    File my_file = new File( file_name );
    Scanner reader = new Scanner(my_file);
    String current_line = new String();
    ListIterator<Character> itr = text.listIterator();
    while( reader.hasNextLine() ){-> n times( linear )
        current_line = reader.nextLine();
        for( int i = 0; i<current_line.length(); ++i ) -> n times
            itr.add(current_line.charAt(i)); -> constant time
        itr.add('\n');
    }
    reader.close();
}
```

This is the same as when there is no iterator, O(n^2) or o(n). Small o a bit more precise

```
public void add2(String text, int index ){
   System.out.println("size is : " + this.text.size() );
   ListIterator<Character> itr = this.text.listIterator(index);
   if( index >= 0 && index < this.text.size() ){
      for( int i = 0; i<text.length(); ++i, ++index ) { -> linear time, n
         itr.add(text.charAt(i)); -> linear time for the iterator(shifting)
      }
   }
   else
      throw new IndexOutOfBoundsException();
}
```

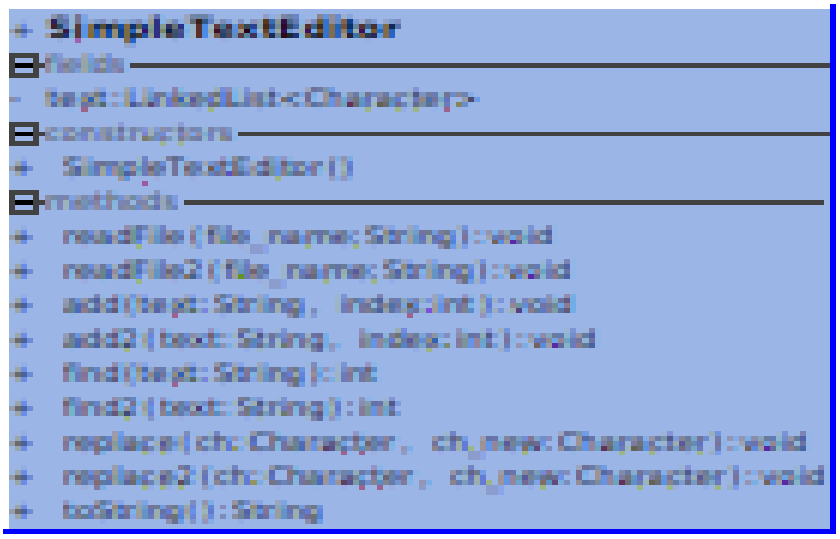
This is a bit faster now and the execution time is O(n^2)

```
public int find2(String text){
   int index = -1;
   boolean found = false;
   for( int i = 0; i<this.text.size() && !found; ++i ){ -> linear execution n
      found = true;
      index = i;
      ListIterator<Character> itr = this.text.listIterator(i);
      for( int k = 0;itr.hasNext() && k < text.length() && found; ++k ){ -> linear execution
         if( itr.next() != text.charAt(k) ) { -> constant
            found = false;
            index = -1;
         }
      }
   }
   return index;
}
```

We have a nested loop and execution for it is O(n^2)

```
public void replace2(Character ch, Character ch_new){
   ListIterator<Character> itr = this.text.listIterator();
   while(itr.hasNext()){ -> linear n
      if( itr.next() == ch ) -> constant
         itr.set(ch_new); -> constant
   }
}
```

Complexity is obviously always the same, linear $\Theta(n)$



Class diagram

Problem solution approach

The way I solved this problem is first wrote these methods for a linked list. There are two methods that do the same thing. The one that have 2 indexed after them are the ones that are using the iterator. Also there are two classes, one SimpleTextEditor uses a linked list and SImpleTextEditor2 uses ArrayList for its data.

Test cases and & running command and results

They can be checked from the command line. Main class is used for testing. Its outputs can be seen by running it

Test scenarios are exactly the same for ArrayList and LinkedList so I just made a table that includes both. Note: Very big text file was used for testing to ensure difference in running times as much as possible

Test Scenario	Expected Results	Actual Results
1.Reading from a file(no iter)	Should be read into our data structure	As expected
2. Aadding Djuro je legendica at 493th index(no iter)	Should be added	As expected
3. Replacing all ‘a’ chars with ‘.’ char	Should be replaced	As expected
4. Finding an element ‘Lorem’	Should find Lorem at the beginning	As expected
5. Finding an element ‘Djuro je legendic.a’	Should not find it	As expected
Doing 1. with iterator	Same	Same

Doing 2. with iterator	Same	Same
Doing 3. with iterator	Same	Same
Doing 4. with iterator	Same	Same
Doing 5. with iterator	Same	Same

Run commands and result output can be seen in the .log file as mentioned in the homework