

Data Structures and Algorithms

COMP1002

## **Project Report**

**Assignment Semester 2 2022**

Sean Anain

20324861

## Contents

User Guide .....	1
Purpose .....	1
Interactive Mode .....	1
Exit Program .....	1
Load Keyboard File .....	1
Node Operations .....	2
Edge Operations .....	2
Display Graph .....	3
Display Graph Information .....	3
Enter a String for Finding a Path .....	3
Generate Paths .....	3
Display Paths .....	3
Save Keyboard .....	3
Silent Mode .....	3
Description of Classes .....	4
DSALinkedList .....	4
DSAListNode .....	4
DSALinkedListIterator .....	4
DSAQueue .....	4
DSAStruct .....	4
DSAGraph .....	4
DSAGraphVertex .....	4
DSAGraphEdge .....	4
Justification of Decisions .....	5
Use of DSAGraph ADT .....	5
Use of DSALinkedList ADT .....	5
Use of DSAQueue ADT .....	5
Use of DSAStruct ADT .....	5
Depth-First Search Methods .....	5
Breadth-First Search Methods .....	5
String Altering .....	5
Switch Keyboard .....	6
UML Class Diagram .....	7
Traceability Matrix .....	8
Showcase .....	10
Introduction .....	10
Scenario 1: Same String on Different Keyboards .....	8
Keyboard: netflix.al .....	8
Keyboard: stan.al .....	9
Keyboard: switch.al .....	10
Discussion .....	10
Scenario 2: Use of Capitals and Punctuation .....	11
Scenario 3: Effect of Wrapping .....	13
Wrapping On .....	13
Wrapping Off .....	13
Conclusion .....	14

## User Guide

Ensure all source code and all supporting code and files are all within the same directory. Compile all necessary files as shown below:

- `javac DSAGraph.java`
- `javac DSALinkedList.java`
- `javac DSAQueue.java`
- `javac DSABackStack.java`
- `javac keyMeUp.java`

When starting program, there are 3 starting options:

- No command line arguments: provides usage information
- `"-i"`: interactive testing environment (`java keyMeUp -i`)
- `"-s"`: silent mode (`java keyMeUp -s keyboardFile stringFile pathFile`)

Note that vertices refer to the keys themselves and edges refer to the adjacency of two respective "keys".

## Purpose

Purpose of this program to represent a keyboard in the form of a graph which allows the user to enter a string to which the program can determine the shortest path to "type" the entered string using different graph traversal methods.

## Interactive Mode

When starting in interactive mode, the following menu is presented. Each feature can be accessed by entering the respective integer associated with each menu choice into the terminal.

0. Exit program
1. Load keyboard file
2. Node operations
  0. Return to main menu
  1. Find
  2. Insert
  3. Delete
  4. Update
3. Edge operations
  0. Return to main menu
  1. Find
  2. Add
  3. Remove
  4. Update
4. Display graph
5. Display graph information
6. Enter string for finding path
7. Generate paths
8. Display path(s)
  - y. Save results to file
9. Save keyboard

## Exit Program

This option promptly ends the program.

## Load Keyboard File

Selecting this option will prompt the user to enter a file name in the format `<filename.fileType>`. This file will then be read and converted into a graph to be used in the program.

## Node Operations

After selecting this option, the user will be presented a second menu to which they will be prompted to enter an integer associated with the desired option

0. Return to main menu
1. Find
2. Insert
3. Delete
4. Update

### *Return to main menu*

Selecting this option returns user to main menu.

### *Find*

The user will be prompted to with a request to enter a label. This label will be associated with the vertex the user is trying to find. Enter a desired label into the terminal and the user will be informed if the associated vertex exists in the current graph or not.

### *Add*

The user will be prompted to enter a label for a new vertex. They will then be requested to enter a value for that vertex. This vertex will then be added to the current graph.

### *Delete*

The user will again be prompted to enter a label of the vertex they intend to remove. They will then be informed whether the associated vertex for that label exists. If the vertex exists, it will be removed from the current graph.

### *Update*

The user will be prompted to enter the label of vertex they intend to edit. If such vertex exists, they are then prompted to enter a new label for the vertex. This vertex will now be referenced by that new label throughout the entire graph

## Edge Operations

After selecting this option, the user will be presented a second menu to which they will be prompted to enter an integer associated with the desired option

0. Return to main menu
1. Find
2. Add
3. Remove
4. Update

### *Return to main menu*

Selecting this option returns user to main menu.

### *Find*

The user will be prompted to with a request to enter two labels. These labels will be associated with the vertices containing the edge the user is trying to find. Enter a desired label into the terminal and the user will be informed if the associated vertex exists in the current graph or not. Note that the order of vertices entered will change whether a edge will be located or not.

### *Add*

The user will be prompted to enter two labels associated with two existing vertices. If both vertices exist and an edge does not already exist between them in the direction chosen by the user, a edge will be added between the two vertices

### Delete

The user will be prompted to enter two labels associated with two existing vertices. If both vertices exist and an edge does exist between them, this edge will be removed from the current graph

### Update

The user will be prompted to enter three labels associated with three existing vertices. If all three exist, edge between first and second will be removed, and edge between first and third will be inserted.

### Display Graph

Selecting this option will display the graph as both an adjacency matrix and list of adjacencies.

### Display Graph Information

Selecting this option will display relevant information regarding the current graph such a number of vertices and edges.

### Enter a String for Finding a Path

The user will be prompted to enter a string they intend to find a path for.

### Generate Paths

Selecting this option will internally generate the paths for the inputted string.

### Display Paths

Selecting this option will display the two paths generated by the program. Each path will be printed on several lines, with each line conveying the path between two “keys” within the respective string. Following this, the number of moves will also be displayed, allowing the user to easily rank the two paths against one another. Following this, the user will be prompted to enter ‘y’ or ‘n’ regarding whether they would like the paths saved. Should the user select ‘y’, the paths will be printed to “results.txt” in the same format it was displayed on the terminal.

### Save Keyboard

Selecting this option will the save the keyboard to “output.al” in the same format as the rest of the existing keyboard files.

### Silent Mode

When running the program in silent mode, all interaction with the user will be done in the command line. When running the program, run with the following command line arguments:

```
Java keyMeUp -s keyboardFile stringFile pathFile
```

Ensure all files are entered in the format <filename.fileType>. The paths can then be viewed by the user by opening the file entered for “pathFile”.

## Description of Classes

### DSALinkedList

The purpose of this class is to create the data structure that is the double ended, doubly linked, linked list. This class gives ability to store data as a linked list and allows for the ability to insert first, insert last, remove first, remove last, peek first, peek last, and to display. This class is very useful for the “keyMeUp” program as it allows data such as the generated paths to be stored as a linked list. This class is also the foundation for the data structures queues and stacks.

### DSAListNode

This class has an composition relationship with “DSALinkedList”. This class is responsible for the creation of the nodes used in the linked list such as the head and tail node. It also gives the ability to retrieve the value of a specific node within the list, set the value of a node, get the next node from the current, get the previous node from the current, and to set the next and previous nodes.

### DSALinkedListIterator

This class also has an composition relationship with “DSALinkedList”. This class has the purpose of creating an iterator to allow a program to iterate through the linked list. This class implements the default java class “Iterator”.

### DSAQueue

This class has the purpose of creating the data structure, queue. This class has an aggregation relationship with “DSALinkedList”. This class allows the linked list to be treated as queue by having methods that allow program to queue a value, dequeue a value, peek the front of the queue, check if the queue is empty, and the ability to iterate through the queue.

### DSABack

This class has the purpose of creating the data structure, stack. This class has an aggregation relationship with “DSALinkedList”. This class allows the linked list to be treated as a stack by having methods that allow the program to push values onto the stack, pop values off of the stack, view the top of the stack, check if the stack is empty, and the ability to iterate through the stack.

### DSAGraph

This class has the purpose of creating the data structure, graph. This class allows the creation of several vertices (nodes) and to form edges between them. This class also has an aggregation relationship with “DSALinkedList”. This class allows programs to add vertices, remove vertices, edit vertices, add edges, remove edges, edit edges, check if a label has a vertex, get number of vertices, get number of edges, retrieve vertices using labels, return linked lists containing the adjacent vertices of one vertex, check if two vertices are adjacent, display the graph as an adjacency list, display graph as adjacency matrix, and perform breadth-first and depth-first search.

### DSAGraphVertex

This class has the purpose of creating the vertices for the DSAGraph class. This class has a composition relationship with “DSAGraphVertex”. The fields for this class include the DSALinkedList, links, Boolean variable visited, objects label and value. This class allows programs to get a vertex’s value, get all adjacent vertices, add edges, set as visited, get visited state, and clear visited.

### DSAGraphEdge

This class has the purpose of creating the vertices for the DSAGraph class. This class has a composition relationship with “DSAGraphEdge”. The fields for this class include two DSAGraphVertex “from” and “to”, objects label and value. This class allows programs to get the edge’s label, value, vertex it is coming from, vertex it is going towards, and its state of direction.

## Justification of Decisions

### Use of DSAGraph ADT

The abstract data type, DSAGraph, was implemented into the code in order to represent the “keyboard”. The underlying data structure is graph from DSAGraph. The production code uses 1 main DSAGraph variable, which represents the entire keyboard which is then passed onto all methods within the code. Within the DSAGraph.java class, two private classes create two more underlying ADTs, DSAGraphVertex, and DSAGraphEdge. These ADTs are called upon within the DSAGraph class in order to create vertices and the edges between them. This entire class was implemented with the purpose of representing a virtual keyboard using a graph.

### Use of DSALinkedList ADT

The abstract data type, DSALinkedList, was implemented into the production code to allow for efficient data storage. The underlying data structure is a linked list from the class DSALinkedList.java. This abstract data type is called upon several times within the program and in its supporting classes as it is a convenient data structure to store data such as the structure of queue, stack, or the path generated by the program, hence why it was necessary to implement.

### Use of DSAQueue ADT

The abstract data type, DSAQueue, was implemented to allow the storage and handling of data as a queue. The ADT, DSAQueue, is created using the DSAQueue.java class. The main purpose of this ADT within the program is its use within the methods involving depth first search and breadth first search. These searching algorithms rely on the use of the queue data type hence why it was implemented into the program.

### Use of DSABack ADT

The abstract data type, DSABack, was implemented to allow the storage and handling of data as a queue. The ADT, DSABack, is created using the DSABack.java class. The main purpose of this ADT within the program is its use within the methods involving depth first search. This searching algorithm relies on the use of the stack data type hence why it was implemented into the program.

### Depth-First Search Methods

In order to generate a path from one key to another, the system must need a way of traversing through the keyboard graph starting at one key to another. One method of doing so would be to use a searching algorithm such as depth-first search. Depth first search traverses a graph by going as far down a path before traversing through the next. Hence why this algorithm, was implemented into the program. This allows the system to search for a key starting at a specific key. This then allows whole string to be typed out on the keyboard. However, this produces a very messy path that sometimes makes unnecessary moves or branches off elsewhere. This why another method is implemented to optimised the outputted path from depth first search. This method removes all unnecessary moves a return an optimised path between the two keys using depth first search.

### Breadth-First Search Methods

As mentioned, to generate a path from one key to another, the system must need a way of traversing through the keyboard graph starting at one key to another. One method of doing so would be to use a searching algorithm such as depth-first search. Breadth first search traverses a graph by visiting all adjacent keys before moving onto the next. Hence why this algorithm, was implemented into the program. This allows the system to search for a key starting at a specific key. This then allows whole string to be typed out on the keyboard. However, this also produces a very messy path that sometimes makes unnecessary moves or branches off elsewhere. This why another method is implemented to optimised the outputted path from breadth first search. This method removes all unnecessary moves a return an optimised path between the two keys using depth first search.

### String Altering

String altering has been implemented into the program in order to change the string to include keys that are necessary but not included in the string itself. This includes the shift key and the space bar. The provided string is analysed one character at a time and added into a linked list. If the character is a whitespace, it is substituted for a

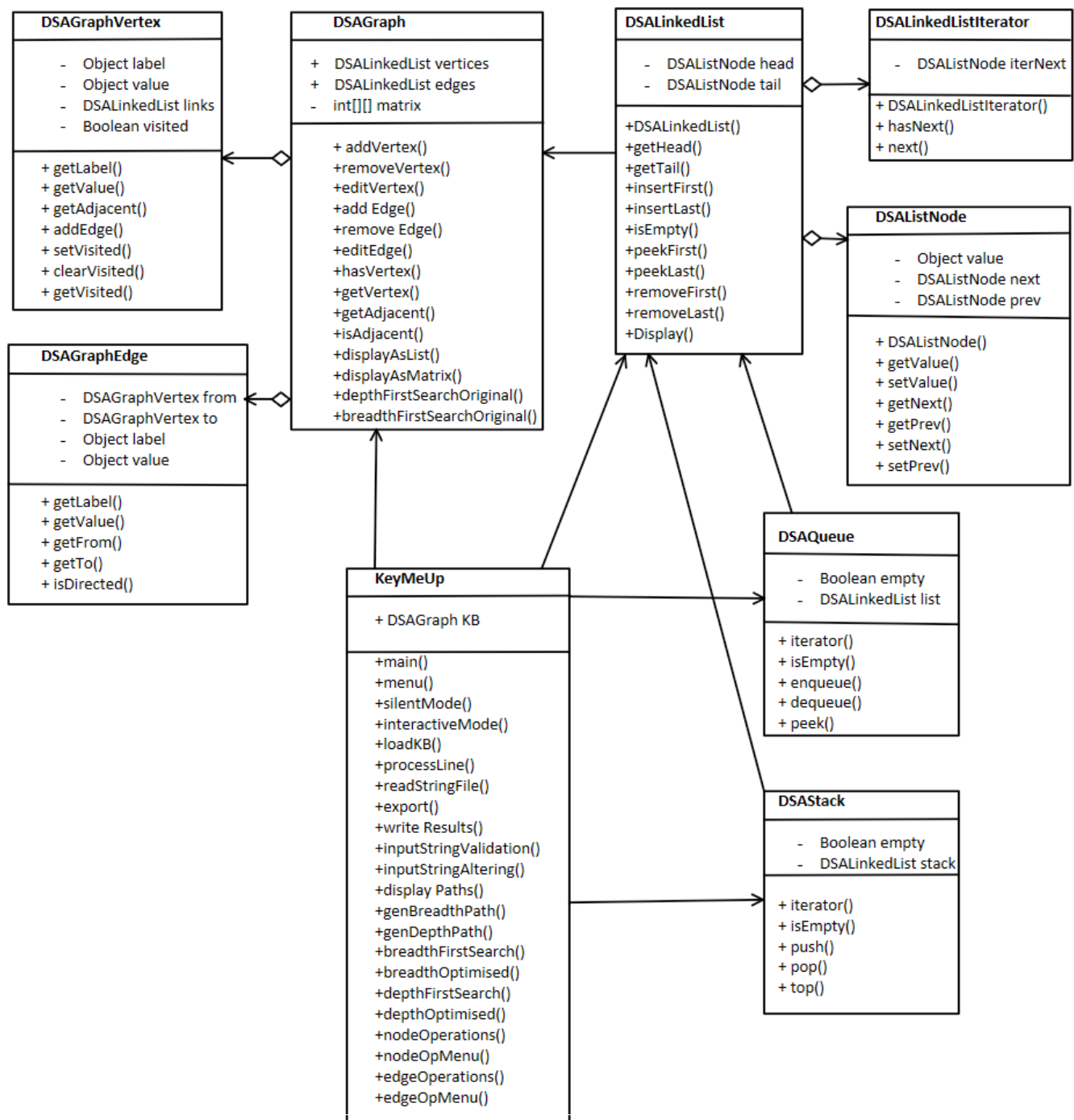
space key within the string array. If a character is a capital letter, the shift key is added last to linked list then followed the respective character but in lower case. This linked list is then converted to a string array and given to the path generation methods. This system is vital for the program's ability to convert strings into correct paths on a keyboard as those extra keys are needed to be pressed even though they are not apart of the original string. This is also important as for most keyboards, the keys are all in lower case, and providing a capitalised character to the path generators would cause issues as a capital version of a vertex does not exist.

### Switch Keyboard

The implementation of a switch keyboards builds off the string altering mentioned above but requires a justification of its own. A switch keyboards contains a key such as the shift key or punctuation key that will toggle to an alternate keyboard to the original such as capitalised keyboard or punctuation keyboard. In order to implement this feature, the inputted string would have to be altered to simulate the existence of the alternate keyboards. The entire system is still using the first keyboard and never changes the graph. When the string is being altered, the system detects if the string consists of any special characters found within the string and places the punctuation key into the string before and after the special character. The special character is then also substituted for the equivalent key in its position on the original keyboard. After the paths have been generated, when displaying or printing the results, the path must also be altered. The system detects if the path presses on the punctuation key and printed all following keys as the equivalent key in that position but on the alternate punctuation keyboard. This is then toggled off when the system presses the punctuation key again. This system allows the program to create paths using a switch keyboard effectively.



## UML Class Diagram



## Traceability Matrix

	Feature	Requirements	Design/Code	Test
1	Driver/Menu & Modes	1.1 System displays usage if called without arguments.	keyMeUp   main()   lines 11-40	Manual Test   Ran program with no command line arguments. [PASSED]
		1.2 System displays interactive menu with “-i” argument.	keyMeUp   main()   lines 21-23	Manual Test   Ran program with “-i” as command line argument [PASSED]
		1.3 In interactive mode, user enter command and system responds.	keyMeUp   main()   lines 11-40	Manual Test   All menu options accessible using terminal commands [PASSED]
		1.4 In interactive mode, user enters exit command and system closes.	keyMeUp   interactiveMode ()   lines 90-189	Manual Test   Enter exit command “0”. [PASSED]
		1.5 System enters silent mode with “-s” argument.	keyMeUp   main()   lines 25-35	Manual Test   Ran program with “-s keyFile strFile pathFile ” as command line arguments [PASSED]
		1.6 System validates command line arguments from user in silent mode.	keyMeUp   main()   lines 25-35	Manual Test   Ran program with “-s keyFile strFile pathFile ” as command line arguments [PASSED]
		1.7 System displays node operations menu when called from main menu.	keyMeUp   interactiveMode ()   lines 106-108	Manual Test   Enter node operations terminal command [PASSED]
		1.8 System displays edge operations menu when called from main menu.	keyMeUp   interactiveMode ()   lines 110-112	Manual Test     Enter edge operations terminal command [PASSED]
		1.9 System returns appropriate error messages from wrong user input.		Manual Test   Enter a character in any menu [PASSED]
2	File I/O	2.1 In interactive mode, system prompts user for file name for graph.	keyMeUp   interactiveMode ()   lines 100-101	Manual Test   Enter terminal command for load keyboard file [PASSED]
		2.2 In silent mode, system reads and writes to all files correctly.	keyMeUp   silentMode ()   lines 63-72	Manual Test   Inputted same keyboard and string in Interactive and Silent mode and compare results [PASSED]
		2.3 System reads graph file and produces correct graph.	keyMeUp   loadKB()   lines 194-233	UnitTestKeyMeUp   File I/O Test   Test 1 [PASSED]
		2.4 System reads string file and exports correct string.	keyMeUp   readStringFile ()   lines 264-302	UnitTestKeyMeUp   File I/O Test   Test 3 [PASSED]
		2.5 System exports graph to “output.al” correctly.	keyMeUp   exportKB ()   lines 304-334	UnitTestKeyMeUp   File I/O Test   Test 4 [PASSED]
		2.6 System exports results to “results.txt” correctly.	keyMeUp   writeResults ()   lines 336-703	UnitTestKeyMeUp   File I/O Test   Test 5 [PASSED]
3	Node Operations	3.1 For all options, system prompts user to enter a label/s.	keyMeUp   nodeOperations ()   lines 1481-1567	Manual Test   Enter all four node operation options using terminal command [PASSED]
		3.2 “Find” option has system return if entered label has an associated node.	keyMeUp   nodeOperations ()   lines 1499-1510	Manual Test   Enter a valid or invalid label. [PASSED]
		3.3 “Insert” option has system create new node correctly.	DSAGraph   addVertex()   lines 112-121	UnitTestDSAGraph   testAddVertex() [PASSED]
		3.4 “Delete” option has system delete associated node correctly.	DSAGraph   removeVertex()   lines 122-174	UnitTestDSAGraph   testRemoveVertex() [PASSED]

		3.5 "Update" option has system change label of associated node successfully.	DSAGraph   editVertex()   lines 175-211	UnitTestDSAGraph   testEditVertex() [PASSED]
4	Edge Operations	4.1 For all options, system prompts user to enter at least 2 labels associated with existing nodes.	keyMeUp   edgeOperations ()   lines 1581-1674	Manual Test   Enter all four edge operation options using terminal command [PASSED]
		4.2 "Find" option has system return if the two nodes associated with the labels has an existing edge between them/	keyMeUp   edgeOperations ()   lines 1598-1612	Manual Test   Enter 2 labels associated with 2 nodes. [PASSED]
		4.3 "Add" option has system create new edge between the two nodes correctly if edge does not already exist.	DSAGraph   addEdge()   lines 212-221	UnitTestDSAGraph   testAddEdge() [PASSED]
		4.4 "Remove" option has system delete associated edge between the two nodes correctly.	DSAGraph   removeEdge()   lines 222-263	UnitTestDSAGraph   testRemoveEdge() [PASSED]
		4.5 "Update" option has system change label of associated node successfully.	DSAGraph   editEdge()   lines 264-276	UnitTestDSAGraph   testEditEdge() [PASSED]
5	Graph Representation	5.1 System displays graph as list of adjacencies and as adjacency matrix correctly	keyMeUp   interactiveMode()   lines 114-120	Manual Test   Enter terminal command to display graph [PASSED]
		5.2 System displays number of nodes/vertices and number of edges within the current graph correctly	keyMeUp   interactiveMode()   lines 122-126	Manual Test   Enter terminal command to display graph information. [PASSED]
6	String Methods	6.1 System prompts user to enter a string	keyMeUp   interactiveMode()   lines 128-143	Manual Test   Enter terminal command to enter string for path. [PASSED]
		6.2 System validates if all characters within the string are associated with a node in the current graph correctly	keyMeUp   inputStringValidation()   lines 705-727	UnitTestKeyMeUp   String Methods Test   Test 1 [PASSED]
		6.3 System alters string for capital letters correctly	keyMeUp   inputStringAltering()   lines 729-743	UnitTestKeyMeUp   String Methods Test   Test 2   Test2 [PASSED]
		6.4 System alters string for punctuation correctly	keyMeUp   inputStringAltering()   lines 729-743	UnitTestKeyMeUp   String Methods Test   Test 2   Test3 & Test4 [PASSED]
7	Path Generation	7.1 System generates a path between all characters in the string using breadth first search correctly.	keyMeUp   genBreadthPath()   lines 1302-1319	UnitTestKeyMeUp   Path Generation Test   Test 5 [PASSED]
		7.2 System generates a path between all characters in the string using depth first search correctly.	keyMeUp   genDepthPath()   lines 1321-1340	UnitTestKeyMeUp   Path Generation Test   Test 6 [PASSED]
		7.3 System displays paths and ranks correctly.	keyMeUp   displayPaths()   lines 945-1299	UnitTestKeyMeUp   File I/O Test   Test 5 [PASSED]
		7.4 User is given a prompt to save results.	keyMeUp   interactiveMode()   lines 166-171	Manual Test   Enter terminal command to display paths. [PASSED]

# Showcase

## Introduction

In order to showcase the extensive functionality of the keyMeUp program, several replicable scenarios will be presented. The major feature of the program will be its ability to generate paths for a given string. This system has the intent of representing a virtual keyboard as a graph, with each node representing a key on the graph and edges and representing the connection between adjacent keys. Note that a key can only have an edge on its left, right, above, and below it. Several types of keyboards can be loaded into the program, with some having the ability to “wrap” over certain sides, some having a space bar or a shift key, and even some with an alternate keyboard for punctuation. The system will then take in a string from the user, which then validates if the string can be “written” using the graph and alters the string to include all keys necessary to generate the respective string. This includes including the shift key, space bar, or punctuation key. The system then generates a path between each key in the string, using an optimised version of breadth first search and depth first search.

Following will be some scenarios showcasing the previously mentioned functionality. They will be exploring the effect on the path created by factors such as using different keyboards for the same string, the effect of using capital letters or punctuation in a string, and the effect of wrapping.

For all scenarios, the following files are necessary:

- |                      |                   |                 |
|----------------------|-------------------|-----------------|
| • DSAGraph.java      | • keyMeUp.java    | • stan.al       |
| • DSALinkedList.java | • results.txt     | • switch.al     |
| • DSAQueue.java      | • inputString.txt | • iview.al      |
| • DSASTack.java      | • netflix.al      | • stanNoWrap.al |

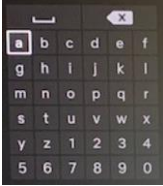
When asked to compile all necessary files, compile the following files:

- javac DSAGraph.java
- javac DSALinkedList.java
- javac DSAQueue.java
- javac DSASTack.java
- javac keyMeUp.java

## Scenario 1: Same String on Different Keyboards

This scenario aims to explore the differences in the paths created by different keyboard given the same string. This scenario also aims to showcase the silent mode feature of the program and its ability to generate two shortest paths to type out of given string using breadth first search and depth first. The keyboards being used in this scenario are listed below:

- netflix.al



- stan.al



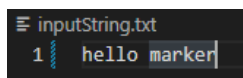
- switch.al



Both “netflix.al” and “stan.al” have the same alphabetical order layout whilst “switch.al” has the more common QWERTY layout. Note that for “netflix.al” there is no wrapping, for “stan.al” there is left and right wrapping, and for “switch.al” there is wrapping on both sides. The string to be used to compare the three keyboards will not be containing special characters as that will be explored in a later scenario. The string will, however, spaces as all three keyboards have the appropriate keys to do so.

The string to be used to compare the three is: “hello marker”

This scenario will be take place using the keyMeUp silent mode. Ensure the string is entered into inputString.txt without quotation marks as shown below.



Compile necessary files and call the program with the following command line arguments for each respective scenario.

```
java keyMeUp -s netflix.al inputString.txt results.txt
```

```
java keyMeUp -s stan.al inputString.txt results.txt
```

```
java keyMeUp -s switch.al inputString.txt results.txt
```

Keyboard: netflix.al

Following is a screenshot of results.txt following the completion of the program.

```

1 | Input string: hello marker
2 |
3 | Breadth First Search
4 |
5 | h b c d e
6 | e k l
7 | l f l
8 | l k j i o
9 | o i c SPACE
10 | SPACE a g m
11 | m g a
12 | a SPACE BACKSPACE d j k l r
13 | r l k
14 | k e
15 | e k l r
16 | Total moves: 33
17 |
18 | Depth First Search
19 |
20 | h i o u 1 2 3 4 x r q k e
21 | e k l
22 | l f l
23 | l k q w 3 9 8 7 1 u o
24 | o p q k e BACKSPACE SPACE
25 | SPACE BACKSPACE d j k q w 3 9 8 7 1 u o n m
26 | m g a
27 | a b h n t z 1 2 3 4 x r
28 | r l k
29 | k e
30 | e k l r
31 | Total moves: 66
32 |
33 | Ranking
34 | =====
35 | 1. Breadth First Search: 33 moves.
36 | 2. Depth First Search: 66 moves.
37 |

```

From the results, it can be concluded that more effective searching algorithm was breadth first search with 33 moves needed to complete the string in comparison to 66 for depth first search. From observation of the paths, breadth first search definitely took the more optimal route with depth first taking very long routes to get to certain keys, often traversing almost the whole length of the keyboard. The space in the string is also taken into account as the path goes to press the space bar.

Keyboard: [stan.al](#)

Following is a screenshot of results.txt following the completion of the program.

```

1 | Input string: hello marker
2 |
3 | Breadth First Search
4 |
5 | h g a f e
6 | e k l
7 | l f l
8 | l k j i o
9 | o i c SPACE
10 | SPACE a g m
11 | m g a
12 | a g m r
13 | r l k
14 | k e
15 | e k l r
16 | Total moves: 29
17 |
18 | Depth First Search
19 |
20 | h g m r q k e
21 | e k l
22 | l f l
23 | l g h n o
24 | o p q k e f a SPACE
25 | SPACE BACKSPACE d j k l r m
26 | m g a
27 | a g l r
28 | r l k
29 | k e
30 | e k l r
31 | Total moves: 39
32 |
33 | Ranking
34 | =====
35 | 1. Breadth First Search: 29 moves.
36 | 2. Depth First Search: 39 moves.
37 |

```

From the results, it can be concluded that more effective searching algorithm was breadth first search with 29 moves needed to complete the string in comparison to 39 for depth first search. In comparison to the Netflix keyboard, both searching algorithms performed better, especially depth first search. This is relevant due to the same layout of keys between netflix.al and stan.al. This improvement can then be sourced from the left and right wrapping on stan.al. The effect of this wrapping will be further explored in a following scenario.

Keyboard: switch.al

Following is a screenshot of results.txt following the completion of the program.

```
1 Input string: hello marker
2
3 Breadth First Search
4
5 h g f d e
6 e w q RETURN + p o l
7 l k l
8 l o
9 o 9 SPACE
10 SPACE v b n m
11 m SPACE OK z a
12 a q w e r
13 r t y u i k
14 k , SPACE #+= 3 e
15 e r
16 Total moves: 39
17
18 Depth First Search
19
20 h n SPACE #+= 3 e
21 e r f g h j k l
22 l k l
23 l o
24 o 9 SPACE
25 SPACE 4 5 t g h n m
26 m SPACE OK RETURN q a
27 a z OK / SPACE 4 r
28 r 4 SPACE OK / : _ l k
29 k , SPACE #+= 3 e
30 e r
31 Total moves: 49
32
33 Ranking
34 =====
35 1. Breadth First Search: 39 moves.
36 2. Depth First Search: 49 moves.
37
```

From the results, it can be concluded that more effective searching algorithm was breadth first search with 39 moves needed to complete the string in comparison to 49 for depth first search. Again, breadth first search prevails as the superior searching algorithm with 10 less moves in comparison to depth first search. In comparison to the other two keyboards, this keyboard performed worse in typing out the same string. The shortest path found was 39 which is greater than the latter. This suggests that the traditional QWERTY layout may be inferior to a basic alphabetical order layout.

## Discussion

Overall, the program succeeded in exploring the different paths generated on different keyboard given the same string. The program successfully generated two paths, using both breadth first search and depth first search, and displayed them in “results.txt” in a concise manner, while also ranking the two paths against one another based on number of moves. The paths are printed by displaying the path between each key in the string on a new line.

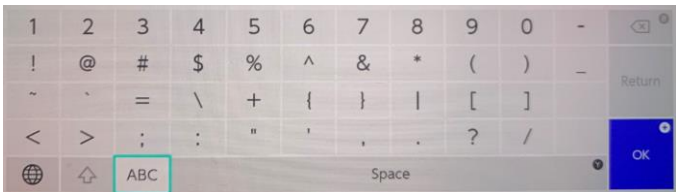
From the results, it can be concluded that alphabetically ordered layout can be more effective in producing a shorter path in comparison to a QWERTY layout given the same string.

## Scenario 2: Use of Capitals and Punctuation

This scenario will be showcasing the use of a switch keyboard within the program. A switch keyboard is one that has shift key to enter an alternate keyboard where all keys are capitalised and a punctuation key to enter a different alternate keyboard with punctuation keys instead of letters.

To showcase this, the keyboard that will be used is “switch.al”.

This keyboard and its two alternate layouts are shown below:



The string to be used to showcase this feature will include spaces, capital letters, and punctuation in order to fully showcase the ability of the program. This string will be: “{“h1 M4rK3r”}”

First, compile necessary files. This scenario will be using keyMeUp in interactive mode. Run the program with the “-i” flag as shown below.

```
java keyMeUp -i
```

The main menu will then be displayed. Enter “1” into the terminal to load a keyboard file. When prompted, enter “switch.al”, with no quotation marks.

```
Main Menu
=====
0. Exit
1. Load keyboard file
2. Node operations (find, insert, delete, update)
3. Edge operations (find, add, remove, update)
4. Display graph
5. Display graph information
6. Enter string for finding path
7. Generate paths
8. Display path(s) (ranked, option to save)
9. Save keyboard
1
Enter file name
switch.al
```

Then open the option to enter a string by entering “6”. Then enter the chosen string into the terminal.



```

Main Menu
=====
0. Exit
1. Load keyboard file
2. Node operations (find, insert, delete, update)
3. Edge operations (find, add, remove, update)
4. Display graph
5. Display graph information
6. Enter string for finding path
7. Generate paths
8. Display path(s) (ranked, option to save)
9. Save keyboard
6
Enter a string
{"h1 M4rk3r"}

```

Then generate the paths by entering "7". These paths will be displayed by entering "8". Following is the results.

```

Input string: {"h1 M4rk3r"}

Breadth First Search

#+= ; = \ + {
{ ' SPACE #+=
#+= SHIFT #+=
#+= ; v "
" SPACE #+=
#+= c d f g h
h n SPACE #+= SHIFT WORLD 1
1 BACKSPACE @ SPACE
SPACE #+= SHIFT
SHIFT X C V B N M
m SPACE 4
4 r
r e w 2 SHIFT
SHIFT WORLD OK / - . , K
k , SPACE #+= 3
3 4 r
r e 3 #+=
#+= ; v "
" SPACE #+=
#+= SHIFT #+=
#+= ; = \ + { }
} m SPACE #+=
Total moves: 76

Depth First Search

#+= SPACE 4 $ \ + {
{ ' SPACE #+=
#+= SHIFT #+=
#+= SPACE 4 $ \ + "
" SPACE #+=
#+= SPACE 4 r f g h
h n SPACE OK z WORLD 1
1 BACKSPACE OK / SPACE
SPACE #+= SHIFT
SHIFT #+= 3 E D F G H J M
m SPACE 4
4 r
r e d c #+= SHIFT
SHIFT #+= 3 E D F G H J K
k , SPACE #+= 3
3 4 r
r e d c #+=
#+= SPACE 4 $ \ + "
" SPACE #+=
#+= SHIFT #+=
#+= SPACE 4 $ \ + { }
} m SPACE #+=
Total moves: 93

Ranking
=====
1. Breadth First Search: 76 moves.
2. Depth First Search: 93 moves.
Save results? y/n

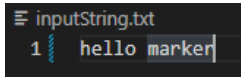
```

From the results, it can be concluded that breadth first search is again superior to depth first search with 76 moves to 93. Again, depth first search is seen taking a much longer path between keys in comparison to breadth first search. From the path, the ability to use the switch keyboard is successfully showcased. It successfully enters the alternate punctuation keyboard using the punctuation key and traverses to the necessary key and returns back to the punctuation key to return to the default keyboard. The program also successfully presses the shift key in order to access the capital letter keyboard and traverses to the necessary key, and the keyboard automatically returns back to the default keyboard.

### Scenario 3: Effect of Wrapping

This scenario will explore the effect of wrapping on generating paths on the same keyboard. This scenario also aims to showcase the program's ability to allow keyboard wrapping. This will be done by using the same keyboard with wrapping on and wrapping off with the same string.

The keyboard to be used in this scenario is "stan.al" and "stanNoWrap.al". This scenario will be taken place using the keyMeUp silent mode. Ensure the string is entered into inputString.txt without quotation marks as shown below.



```
1 hello marker
```

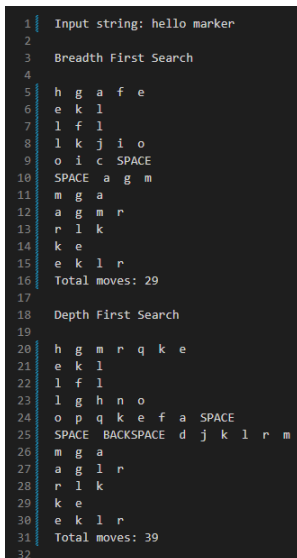
Compile all necessary files and call the program with the following command line arguments for each respective scenario.

```
java keyMeUp -s stan.al inputString.txt results.txt
```

```
java keyMeUp -s stanNoWrap.al inputString.txt results.txt
```

### Wrapping On

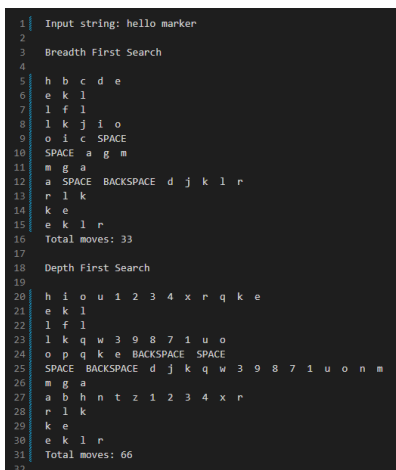
Following is a screenshot of results.txt following the completion of the program.



```
1 Input string: hello marker
2
3 Breadth First Search
4
5 h g a f e
6 e k l
7 l f l
8 l k j i o
9 o i c SPACE
10 SPACE a g m
11 m g a
12 a g m r
13 r l k
14 k e
15 e k l r
16 Total moves: 29
17
18 Depth First Search
19
20 h g m r q k e
21 e k l
22 l f l
23 l g h n o
24 o p q k e f a SPACE
25 SPACE BACKSPACE d j k l r m
26 m g a
27 a g l r
28 r l k
29 k e
30 e k l r
31 Total moves: 39
32
```

### Wrapping Off

Following is a screenshot of results.txt following the completion of the program.



```
1 Input string: hello marker
2
3 Breadth First Search
4
5 h b c d e
6 e k l
7 l f l
8 l k j i o
9 o i c SPACE
10 SPACE a g m
11 m g a
12 a SPACE BACKSPACE d j k l r
13 r l k
14 k e
15 e k l r
16 Total moves: 33
17
18 Depth First Search
19
20 h i o u 1 2 3 4 x r q k e
21 e k l
22 l f l
23 l k q w 3 9 8 7 1 u o
24 o p q k e BACKSPACE SPACE
25 SPACE BACKSPACE d j k q w 3 9 8 7 1 u o n m
26 m g a
27 a b h n t z 1 2 3 4 x r
28 r l k
29 k e
30 e k l r
31 Total moves: 66
32
```

Overall, this scenario is successful in showcasing the wrapping ability of the program. As shown in the results with wrapping on, as the program traverses from “h” to “e”, it utilises the wrapping ability from “a” to “f”, whilst the results with wrapping off show that the program had to traverse the whole width of the keyboard to get to “f”. This efficiency is shown by the significantly less average number of moves with wrapping on compared to wrapping off. Both have depth first search producing the shortest path, with wrapping on having 29 moves and wrapping off with 33. For breadth first search, wrapping on had a path length of 39 whilst the latter had a massive 66. This all suggests that wrapping allows the program to be more efficient as it produces a much shorter path.

## Conclusion

Ultimately, the “keyMeUp.java” program is successful in providing all necessary features given in this assignment. A portion of these features are displayed in the showcase section of this report. The main feature of this program, which is to simulate a virtual keyboard and to find the shortest path to type a string on it, was heavily showcased, with the program having the ability to use keyboards with wrapping, capital letters, spaces, and special characters. In the showcase section of this report, these abilities are explored. It was shown that keyboards with wrapping produced a shorter path due to more available pathways. This program also showed its ability to generate these paths using two searching algorithms, breadth first search and depth first search. The effectiveness of these two algorithms were explored in the showcase section of the report. From the results, the clearly superior algorithm was breadth first search, providing a shorter path every time. The poor performance of depth first search would be the fact that the algorithm is designed to go down a path as far as possible before exploring another. This meant that algorithm would continue along the first set of keys it chooses until it reaches the intended destination, which often involves taking much longer paths. Overall, this assignment thoroughly explored the concepts of data structures such as graphs, linked lists, stacks, and queues, as well as algorithms such as breadth first search and depth first search. This project could be brought even further by exploring different searching algorithms or creating a visual representation of the graph.

## Curtin University – Department of Computing

# Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	ANAIN	Student ID:	20324861
Other name(s):	SEAN		
Unit name:	Data Structures and Algorithms	Unit ID:	COMP1002
Lecturer / unit coordinator:	Valerie Maxville		
Assessment:	Assignment		

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Sean Anain	Date of signature: 10/10/2022
-----------------------	-------------------------------

*(By submitting this form, you indicate that you agree with all the above text.)*