

INF1008
Data Structures & Algorithms

Group 10

CONTENTS

QUESTION 1	PAGE 4
1.1 Problem Statement	4
1.2 Requirements/Specification	4
1.3 User Guide	4
1.4 Structure/Design	4
Node — 5 • QueueADT — 5 • StackADT — 6	
1.5 Limitations	7
1.6 Testing	7
Option 1 – Automatic Generation — 7 • Option 2 – Manual Testing — 8	
1.7 Listings	9
QueueADT.cpp — 9 • StackADT.cpp — 11 • Source.cpp — 12	
QUESTION 2	PAGE 19
2.1 Problem Statement	19
2.2 Requirements/Specification	19
2.3 User Guide	19
2.4 Structure/Design	19
2.5 Limitations	20
2.6 Testing	20
Invalid Inputs — 21	
2.7 Listings	22
Algorithm — 22 • Testing — 22	
QUESTION 3	PAGE 24
3.1 Problem Statement	24
3.2 Requirements/Specification	24
3.3 User Guide	24
3.4 Structure/Design	24
3.5 Limitations	26
3.6 Testing	26
3.7 Listings	27
QUESTION 4	PAGE 30
4.1 Problem Statement	30
4.2 Requirements/Specification	30

4.3	User Guide	30
4.4	Structure/Design	30
4.5	Limitations	31
4.6	Testing	31
4.7	Listings	31
	Algorithm — 31 • Testing — 32	

QUESTION 5

5.1	Problem Statement	34
5.2	Requirements/Specification	34
5.3	User Guide	34
5.4	Structure/Design	35
5.5	Limitations	35
5.6	Testing	35
5.7	Listings	37

LIST OF FIGURES

QUESTION 1	PAGE
1.1 Node Class Overview	5
1.2 Queue Class Overview	5
1.3 Stack Class Overview	6
1.4 Command-line Interface prompt	7
1.5 Option 1 prompt	8
1.6 Option 1 testing	8
1.7 Option 2 testing with invalid inputs	8
QUESTION 2	PAGE
2.1 Test output	21
QUESTION 3	PAGE
3.1 Initialising the array	24
3.2 Insertion	25
3.3 Removal of nodes	25
3.4 List access example (accessing index 4)	26
3.5 insertItem() testing	26
3.6 removeItem() testing	26
3.7 get() testing	26
3.8 Overflow testing	27
3.9 Empty array removeItem() testing	27
QUESTION 4	PAGE
4.1 Valid and invalid inputs	31
4.2 Running the testing script	31
QUESTION 5	PAGE
5.1 Outputs of stable and unstable variants of the sorting algorithms	36

Question 1

1.1 Problem Statement

Implement the queue ADT using a Doubly linked list using any programming language. After the implementation, make use of the standard queue operations to implement the stack ADT. Clearly demonstrate the validity of your implementation through enough test cases. Evaluate the time complexity of the push, pop, empty and full operations.

1.2 Requirements/Specification

Users will have access to a container with behaviour akin to a stack, however in the background, the processes are done using a queue and its functions built upon a linked list structure. Users will have access to the standard stack functions - `push`, `pop`, `isEmpty` and `isFull` operations.

1.3 User Guide

An executable is provided, users can run the executable directly and follow the on-screen prompts, more in-depth details are provided in the Testing section.

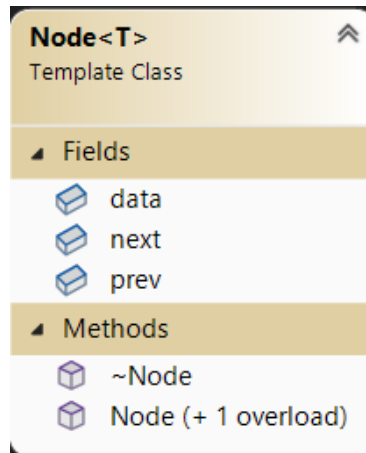
1.4 Structure/Design

The stack as well as the underlying queue containers both have generic types built into the system. Thus, the same code can be re-used to store different types of data, including class objects, should the programmer wish to do that.

There are 3 classes utilised in this program, “Node”, “QueueADT” and “StackADT”. All 3 classes are template classes using typename `T`.

1.4.1 Node

Figure 1.1: Node Class Overview



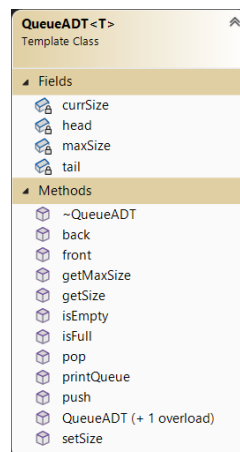
- Data Members

1. `T data` – Used to store the inputted data
2. `Node<T>* next` – Used to store the address of the next node, for use in a linked list
3. `Node<T>* prev` – Used to store the address of the previous node, for use in a linked list

- Methods

1. `Node()` – Default constructor
2. `~Node()` – Default destructor
3. `Node(T data)` – An overloaded constructor, takes in a variable to initialise the data on construction.

Figure 1.2: Queue Class Overview



1.4.2 QueueADT

- Data Members

1. `int maxSize` – Used to limit the size of the container
2. `int currSize` – Denotes the amount of data stored in the container, will be updated as the data is added and removed from the container

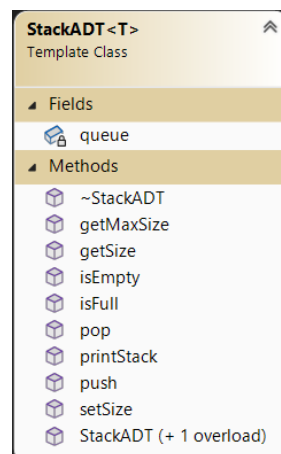
3. `Node<T>* head` – A pointer to the head node of the queue
4. `Node<T>* tail` – A pointer to the tail node of the queue

- Methods

1. `QueueADT()` – Default constructor
2. `~QueueADT()` – Default destructor, will iterate through the linked list and delete all nodes when called
3. `QueueADT(int size)` – An overloaded constructor, takes an `int` argument to initialise `maxSize`
4. `front()` – Returns a pointer to the head node of the queue
5. `back()` – Returns a pointer to the tail node of the queue
6. `setMaxSize(int size)` – Passes in an integer to change the container's max allotted size
7. `getMaxSize()` – Returns `maxSize`
8. `getSize()` – Returns `currSize`
9. `isEmpty()` – Returns the truth value of if `currSize` equal to 0 and if `head` or `tail` are `NULL` as an added precaution, has a time complexity of $O(1)$ as `currSize` is updated together with the container, with the greater than or equal to comparator being used as an added precaution
10. `isFull()` – Returns the truth value of if `currSize` is greater than or equal to `maxSize`, has a time complexity of $O(1)$ as `currSize` is updated together with the container, with the greater than or equal to comparator being used as an added precaution.
11. `push(T data)` – `data` is passed in, and a `Node` is constructed using its overloaded constructor to be added to the linked list, as well as link the nodes between each other, has a time complexity of $O(1)$, as the pointer to the `tail` is stored, new elements can simply be added directly behind `tail` as no comparisons are required. Updates `currSize`, `head`, and `tail` members accordingly
12. `pop()` – Removes the element at the front of the queue, and returns a pointer to it to have its data extracted or deleted accordingly, has a time complexity of $O(1)$, as a pointer to `head` is stored, thus the front element can be removed directly with no comparisons. Updates `currSize`, `head`, and `tail` members accordingly
13. `printQueue()` – Prints the entire linked list, has a time complexity of $O(n)$ as iteration through the entire list is required

1.4.3 StackADT

Figure 1.3: Stack Class Overview



- Data Members

1. `QueueADT<T>* queue` – A pointer to the queue container used to store data

- Methods

1. `StackADT()` – Default constructor
2. `~StackADT()` – Default destructor, invokes the queue destructor, and deletes the pointer to the queue
3. `StackADT(int size)` – Overloaded constructor, initialises the `QueueADT` member with a `maxSize` of the value passed in
4. `setMaxSize(int size)` – Invokes the `QueueADT` member's `setMaxSize(int size)`, passing the same value in
5. `getMaxSize()` – Invokes the `QueueADT` member's `getMaxSize()`
6. `getSize()` – Invokes the `QueueADT` member's `getSize()`
7. `isEmpty()` – Invokes the `QueueADT` member's `isEmpty()`, therefore time complexity of $O(1)$
8. `isFull()` – Invokes the `QueueADT` member's `isFull()`, therefore time complexity of $O(1)$
9. `push(T data)` – Invokes the `QueueADT` member's `push(T data)`, after pushing the initial data in, all the other members in front of it are popped, and re-pushed into the queue to simulate a stack. This causes the function to have a time complexity of $O(n)$, as every element in the container has to be reorganised
10. `pop()` – Invokes the `QueueADT` member's `pop()`, therefore time complexity of $O(1)$

1.5 Limitations

The data type specified on creation is final and is unable to be changed unless the stack is completely re-initialised. A consideration was made to utilise a class holding multiple data types - `int`(4 bytes), `float`(4 bytes), `double`(8 bytes), `char`(1 byte), `std::string`(> 24 bytes) - and construct a container with the type of that class. It would allow the container to store any data that has shares the property of being a standard data type. However, upon further deliberation, a class with that many types will cause memory bloat, as an example if the majority of the data entered was integers, for every 4 bytes used, > 37 bytes are reserved but unused by the system.

1.6 Testing

Users will be prompted (option 1 or option 2) to either allow the program to automatically generate test cases, or test the system out themselves. Users can also input a “\$” to terminate the program.

Figure 1.4: Command-line Interface prompt

```
This is a demo of the stack ADT with queue ADT as the base. What would you like to do?
1) Auto generate 10 stacks of random length(1-10), random amount of data(1-10) and data.
2) Manually input test cases.
Enter $ if you wish to stop the process.
```

1.6.1 Option 1 – Automatic Generation

The program will prompt the user for the type of data they wish to store.

Figure 1.5: Option 1 prompt

```
What data type would you like to use?
1) int
2) float
3) double
4) char
5) string
```

The program will then generate 10 stacks of random length and random amount of data corresponding to the data type specified by the user.

Figure 1.6: Option 1 testing

```
Int Stack 0, Max Stack Size : 9, Number of data : 6
Data stored : 25311, 369, 30400, 13807, 19493, 17619

Int Stack 1, Max Stack Size : 7, Number of data : 9
List is full, unable to push in 2700
List is full, unable to push in 6808
Data stored : 13915, 24739, 2441, 23750, 30711, 29002, 12431
```

1.6.2 Option 2 – Manual Testing

The program will similarly prompt the user for the type of data they wish to store. Additionally, it will enquire on the size they wish for the container to be.

The user will then be able to enter data fitting to the data type into the stack, any data not fitting the data type will be rejected, discarding it and looping back to where the user can continue inputting data. Similarly, if the user tries to add more data into the stack when it is full, the program will also reject the entry. At any time the user can choose to enter “\$” and the process will end, and loop back to the start of the program.

Figure 1.7: Option 2 testing with invalid inputs

```
Data stored : 4, 7, 8, 9, 0, 5, 4, 3, 2, 1
Int Stack Testing, Max Stack Size : 10, Number of data : 10
Enter $ if you wish to stop the process.
1
List is full, unable to push in 1
Data stored : 4, 7, 8, 9, 0, 5, 4, 3, 2, 1
Int Stack Testing, Max Stack Size : 10, Number of data : 10
Enter $ if you wish to stop the process.
j
Please enter a valid integer
```

1.7 Listings

1.7.1 QueueADT.cpp

```
1  #pragma once
2  #include "QueueADT.h"
3
4
5  template <class T>
6  QueueADT<T>::QueueADT() {
7      maxSize = 0;
8      currSize = 0;
9      head = tail = NULL;
10 }
11
12 template <class T>
13 QueueADT<T>::QueueADT(int size) {
14     maxSize = size;
15     currSize = 0;
16     head = tail = NULL;
17 }
18
19 //delete, could do the same thing as empty?
20 template <class T>
21 QueueADT<T>::~~QueueADT() {
22     Node<T>* iter = head;
23     //while iter exists, delete it and move next
24     while (iter) {
25         head = iter;
26         iter = iter->next;
27         delete head;
28         head = NULL;
29     }
30     //safety
31     head = tail = iter = NULL;
32 }
33 template <class T>
34 void QueueADT<T>::setMaxSize(int size) {
35     maxSize = size;
36 }
37
38 template <class T>
39 int QueueADT<T>::getSize() {
40     return currSize;
41 }
42
43 template <class T>
44 int QueueADT<T>::getMaxSize() {
45     return maxSize;
46 }
47
48 //insert element at the end of the Queue
49 template <class T>
50 void QueueADT<T>::push(T data) {
51
52     // if list is full return out
53     if (isFull()) {
54         std::cout << "Container full, failed to push " << data << std::endl;
55         return;
56     }
57
58     Node<T>* newNode = new Node<T>(data);
59
60     //check if list is empty, if yes assign head and tail to same node
61     if (isEmpty()) {
62         head = tail = newNode;
63         ++currSize;
64         return;
65     }
66 }
```

```

66
67 //getting last object and linking newNode to list
68 Node<T>* iter = tail;
69 iter->next = newNode;
70 newNode->prev = iter;
71 tail = newNode;
72 ++currSize; //add to counter
73 }
74
75 //remove and return 1st element
76 template <class T>
77 Node<T>* QueueADT<T>::pop() {
78     if (isEmpty()) {
79         return NULL;
80     }
81     Node<T>* iter = head;
82     if (head->next != NULL) {
83         head = head->next; //move head pointer back
84     }
85     head->prev = NULL; //break prev link to old head node
86     iter->next = NULL; //break next link from old head node
87     --currSize; //minus counter
88     return iter;
89 }
90
91 //checks if container is empty
92 template <class T>
93 bool QueueADT<T>::isEmpty() {
94     //returns if head and tail are both NULL
95     return ((head == NULL || tail == NULL) && currSize == 0);
96 }
97
98 //returns if the Queue is full / at capacity
99 template <class T>
100 bool QueueADT<T>::isFull() {
101     return currSize >= maxSize;
102 }
103
104 template <class T>
105 void QueueADT<T>::printQueue() {
106     //if head or tail is not init, something wrong return out
107     if (!head || !tail)
108         return;
109
110     std::cout << "Data stored : ";
111     //printing data in list
112     Node<T>* iter = head;
113     while (iter != NULL) {
114         //printf("%.6f", iter->data); //printf here to achieve format
115         std::cout << iter->data;
116         //formatting
117         if (iter->next != NULL)
118             std::cout << ", ";
119         iter = iter->next;
120     }
121     std::cout << std::endl;
122 }
123
124 template<class T>
125 Node<T>* QueueADT<T>::front()
126 {
127     //if head exists return head else NULL
128     if (head)
129         return head;
130
131     return NULL;
132 }
133
134 template<class T>
135 Node<T>* QueueADT<T>::back()
136 {

```

```

137 //if tail exists return head else NULL
138 if (tail)
139     return tail;
140
141     return NULL;
142 }

```

1.7.2 StackADT.cpp

```

1
2 #pragma once
3 #include "StackADT.h"
4
5 template<class T>
6 StackADT<T>::StackADT(){
7     queue = new QueueADT<T>(0);
8 }
9
10 template<class T>
11 StackADT<T>::StackADT(int size){
12     queue = new QueueADT<T>(size);
13 }
14
15 template<class T>
16 StackADT<T>::~~StackADT(){
17     //if queue was init, delete
18     if (queue) {
19         delete queue;
20         queue = NULL;
21     }
22 }
23
24 template<class T>
25 void StackADT<T>::setMaxSize(int size){
26     queue->setMaxSize(size);
27 }
28
29 template<class T>
30 int StackADT<T>::getSize(){
31     return queue->getSize();
32 }
33
34 template<class T>
35 int StackADT<T>::getMaxSize() {
36     return queue->getMaxSize();
37 }
38
39 template<class T>
40 void StackADT<T>::push(T data){
41     // if list is full return out
42     if (isFull()) {
43         std::cout << "List is full, unable to push in " << data << std::endl;
44         return;
45     }
46
47     //push the new data on,
48     queue->push(data);
49     //iterate through pop and re-push everything in to
50     //push the new data to the top
51     if (queue->getSize() > 1) {
52         for (int i = 0; i < queue->getSize()-1; ++i) {
53             Node<T> * oldNode = queue->pop();
54             T oldData = oldNode->data;
55             if(oldNode){
56                 delete oldNode;
57                 oldNode = NULL;
58             }
59             queue->push(oldData);
60         }

```

```

61     }
62 }
63
64 template<class T>
65 Node<T>* StackADT<T>::pop()
66 {
67     //get front of queue / top of stack
68     //Node<T>* iter = queue->pop();
69     return queue->pop();
70 }
71
72 template<class T>
73 bool StackADT<T>::isEmpty()
74 {
75     return queue->isEmpty();
76 }
77
78 template<class T>
79 bool StackADT<T>::isFull()
80 {
81     return queue->isFull();;
82 }
83
84 template<class T>
85 void StackADT<T>::printStack()
86 {
87     queue->printQueue();
88 }

```

1.7.3 Source.cpp

```

1
2 #include <stdio.h>
3 #include <iostream>
4 #include "StackADT.h"
5 #include "StackADT.cpp"
6 #include <string>
7 #include <cstdlib>
8 #include <ctime>
9 #include <iomanip>
10
11 enum TestCase {
12     TC_NOTSELECTED,
13     TC_AUTOGEN,
14     TC_MANUAL,
15 };
16
17 enum DataType {
18     DT_NOTSELECTED,
19     DT_INT,
20     DT_FLOAT,
21     DT_DOUBLE,
22     DT_CHAR,
23     DT_STRING
24 };
25
26 int main(void) {
27
28     //seeding
29     srand(time(0));
30
31     //void* theStack;
32     StackADT<int>* intStack = NULL;
33     StackADT<float>* floatStack = NULL;
34     StackADT<double>* doubleStack = NULL;
35     StackADT<char>* charStack = NULL;
36     StackADT<std::string>* stringStack = NULL;
37     bool mainLoop = true;
38     //theStack = charStack;

```

```

39 while (mainLoop) {
40     TestCase testCase = TC_NOTSELECTED;
41     DataType dataType = DT_NOTSELECTED;
42     std::string option = "";
43     int stackSize = 0; //to determine the size of stack to init
44
45     std::cout << "This is a demo of the stack ADT with queue ADT as the base. What would you
like to do?" << std::endl;
46     std::cout << "1) Auto generate 10 stacks of random length(1-10), random amount of data
(1-10) and data." << std::endl;
47     std::cout << "2) Manually input test cases." << std::endl;
48     std::cout << "Enter $ if you wish to stop the process." << std::endl;
49
50     //giving user options to showcase algorithm*****
51     while (testCase == TC_NOTSELECTED) {
52         std::getline(std::cin, option);
53         if (option == "$") {
54             mainLoop = false;
55             break;
56         }
57
58         //just take the 1st char in the string then set the option
59         switch (option[0]) {
60             case '1':
61                 testCase = TC_AUTOGEN;
62                 break;
63             case '2':
64                 testCase = TC_MANUAL;
65                 break;
66             default:
67                 std::cout << "Please enter a valid option." << std::endl;
68                 testCase = TC_NOTSELECTED; //safety
69                 break;
70         }
71         option = ""; //clear string, safety
72
73     }
74     if (!mainLoop) {
75         break;
76     }
77
78     //data type option *****
79     std::cout << "What data type would you like to use?" << std::endl;
80     std::cout << "1) int" << std::endl;
81     std::cout << "2) float" << std::endl;
82     std::cout << "3) double" << std::endl;
83     std::cout << "4) char" << std::endl;
84     std::cout << "5) string" << std::endl;
85
86     while (dataType == DT_NOTSELECTED) {
87         std::getline(std::cin, option);
88
89         //just take the 1st char in the string then set the option
90         switch (option[0]) {
91             case '1':
92                 dataType = DT_INT;
93                 break;
94             case '2':
95                 dataType = DT_FLOAT;
96                 break;
97             case '3':
98                 dataType = DT_DOUBLE;
99                 break;
100             case '4':
101                 dataType = DT_CHAR;
102                 break;
103             case '5':
104                 dataType = DT_STRING;
105                 break;
106             default:
107                 std::cout << "Invalid input. Please enter a valid option." << std::endl;

```

```

108     dataType = DT_NOTSELECTED; //safety
109     break;
110 }
111 option = ""; //clear string, safety
112
113 }
114
115 //auto generate test case
116 if (testCase == TC_AUTOGEN) {
117
118     int numtest = 10;
119     for (int i = 0; i < numtest; i++) {
120         int stackSize = rand() % 10;
121         int numberOfData = rand() % 10;
122
123         switch (dataType) {
124             case DT_INT:
125                 intStack = new StackADT<int>(stackSize);
126                 std::cout << "Int Stack " + std::to_string(i) + ", Max Stack Size : " + std::
to_string(stackSize)
127                     + ", Number of data : " + std::to_string(numberOfData) << std::endl;
128                 for (int j = 0; j < numberOfData; j++) {
129                     int data = rand();
130                     intStack->push(data);
131                 }
132                 intStack->printStack();
133                 break;
134             case DT_FLOAT:
135                 floatStack = new StackADT<float>(stackSize);
136                 std::cout << "Float Stack " + std::to_string(i) + ", Max Stack Size : " + std::
to_string(stackSize)
137                     + ", Number of data : " + std::to_string(numberOfData) << std::endl;
138                 for (int j = 0; j < numberOfData; j++) {
139                     float data = (float)(rand() % RAND_MAX) / RAND_MAX;
140                     floatStack->push(data);
141                 }
142                 floatStack->printStack();
143                 break;
144             case DT_DOUBLE:
145                 doubleStack = new StackADT<double>(stackSize);
146                 std::cout << "Double Stack " + std::to_string(i) + ", Max Stack Size : " + std::
to_string(stackSize)
147                     + ", Number of data : " + std::to_string(numberOfData) << std::endl;
148                 for (int j = 0; j < numberOfData; j++) {
149                     double data = (double)(rand() % RAND_MAX) / RAND_MAX;
150                     doubleStack->push(data);
151                 }
152                 doubleStack->printStack();
153                 break;
154             case DT_CHAR:
155                 charStack = new StackADT<char>(stackSize);
156                 std::cout << "Char Stack " + std::to_string(i) + ", Max Stack Size : " + std::
to_string(stackSize)
157                     + ", Number of data : " + std::to_string(numberOfData) << std::endl;
158                 for (int j = 0; j < numberOfData; j++) {
159                     int number = rand() % 2;
160                     char data;
161                     if (number == 0) { //generate upper case
162                         number = rand() % 26 + 65;
163                     }
164                     else { //generate lower case
165                         number = rand() % 26 + 97;
166                     }
167                     data = (char)(number);
168                     charStack->push(data);
169                 }
170                 charStack->printStack();
171                 break;
172             case DT_STRING:
173                 stringStack = new StackADT<std::string>(stackSize);
174                 std::cout << "String Stack " + std::to_string(i) + ", Max Stack Size : " + std::

```

```

175 to_string(stackSize)
176     + ", Number of data : " + std::to_string(numberOfData) << std::endl;
177     for (int j = 0; j < numberOfData; j++) {
178         int stringLen = rand() % 10;
179         std::string data = "";
180         //data.append
181         for (int k = 0; k < stringLen; k++) {
182             int number = rand() % 2;
183             char letter;
184             if (number == 0) { //generate upper case
185                 number = rand() % 26 + 65;
186             }
187             else { //generate lower case
188                 number = rand() % 26 + 97;
189             }
190             letter = (char)(number);
191             data += letter;
192         }
193         stringStack->push(data);
194     }
195     stringStack->printStack();
196     break;
197 default:
198     //reset loop?
199     break;
200 }
201 //formatting
202 std::cout << std::endl;
203
204 if (intStack) {
205     delete intStack;
206     intStack = NULL;
207 }
208 if (charStack) {
209     delete charStack;
210     charStack = NULL;
211 }
212 if (floatStack) {
213     delete floatStack;
214     floatStack = NULL;
215 }
216 if (doubleStack) {
217     delete doubleStack;
218     doubleStack = NULL;
219 }
220 if (stringStack) {
221     delete stringStack;
222     stringStack = NULL;
223 }
224 }
225 }
226 else if (testCase == TC_MANUAL) {
227     std::cout << "Please enter the size you wish the list to be." << std::endl;
228     while (true) {
229         //check if user input was an int, if yes stack size determined
230         if (std::cin >> stackSize) {
231             break;
232         }
233         std::cout << "Invalid input. Please enter an integer." << std::endl;
234         //clear stream safety
235         std::cin.clear();
236         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
237     }
238     //user test loop
239     std::string input = "";
240     bool activeLoop = true;
241     bool validInput = false;
242     switch (dataType) {
243     case DT_INT:
244         intStack = new StackADT<int>(stackSize);

```



```

245 //loop that runs to accept user input
246 while (activeLoop) {
247     std::cout << "Int Stack Testing, Max Stack Size : " + std::to_string(stackSize)
248         + ", Number of data : " + std::to_string(intStack->getSize()) << std::endl;
249     std::cout << "Enter $ if you wish to stop the process." << std::endl;
250     int data = 0;
251     //validation, looping till get a valid input
252     validInput = false;
253     while (!validInput) {
254         std::getline(std::cin, input);
255         std::cin.clear();
256         if (input.empty()) //skip empty entries
257             continue;
258         if (input == "$") { //to break out of the loop
259             activeLoop = false;
260             break;
261         }
262         try {
263             data = std::stoi(input);
264             validInput = true;
265         }
266         catch (const std::invalid_argument& e) {
267             std::cout << "Please enter a valid integer" << std::endl;
268             validInput = false;
269         }
270     }
271     //to skip the pushing if the user wants to exit
272     if (!activeLoop) {
273         break;
274     }
275     intStack->push(data);
276     intStack->printStack();
277 }
278 break;
279 case DT_FLOAT:
280     floatStack = new StackADT<float>(stackSize);
281     //loop that runs to accept user input
282     while (activeLoop) {
283         std::cout << "Float Stack Testing, Max Stack Size : " + std::to_string(stackSize)
284             + ", Number of data : " + std::to_string(floatStack->getSize()) << std::endl;
285         std::cout << "Enter $ if you wish to stop the process." << std::endl;
286         float data = 0;
287         validInput = false;
288         //validation, looping till get a valid input
289         while (!validInput) {
290             std::getline(std::cin, input);
291             std::cin.clear();
292             if (input.empty()) //skip empty entries
293                 continue;
294             if (input == "$") {
295                 activeLoop = false;
296                 break;
297             }
298             try {
299                 data = std::stof(input);
300                 validInput = true;
301             }
302             catch (const std::invalid_argument& e) {
303                 std::cout << "Please enter a valid float" << std::endl;
304                 validInput = false;
305             }
306         }
307         //to skip the pushing if the user wants to exit
308         if (!activeLoop) {
309             break;
310         }
311         floatStack->push(data);
312         floatStack->printStack();
313     }
314     break;
315 case DT_DOUBLE:

```

```

316 doubleStack = new StackADT<double>(stackSize);
317 //loop that runs to accept user input
318 while (activeLoop) {
319     std::cout << "Double Stack Testing, Max Stack Size : " + std::to_string(stackSize)
320         + ", Number of data : " + std::to_string(doubleStack->getSize()) << std::endl;
321     std::cout << "Enter $ if you wish to stop the process." << std::endl;
322     double data = 0;
323     validInput = false;
324     //validation, looping till get a valid input
325     while (!validInput) {
326         std::getline(std::cin, input);
327         std::cin.clear();
328         if (input.empty()) //skip empty entries
329             continue;
330         if (input == "$") {
331             activeLoop = false;
332             break;
333         }
334         try {
335             data = std::stod(input);
336             validInput = true;
337         }
338         catch (const std::invalid_argument& e) {
339             std::cout << "Please enter a valid double" << std::endl;
340             validInput = false;
341         }
342     }
343     //to skip the pushing if the user wants to exit
344     if (!activeLoop) {
345         break;
346     }
347     doubleStack->push(data);
348     doubleStack->printStack();
349 }
350 break;
351 case DT_CHAR:
352     charStack = new StackADT<char>(stackSize);
353     //loop that runs to accept user input
354     while (activeLoop) {
355         std::cout << "Char Stack Testing, Max Stack Size : " + std::to_string(stackSize)
356             + ", Number of data : " + std::to_string(charStack->getSize()) << std::endl;
357         std::cout << "Enter $ if you wish to stop the process." << std::endl;
358         char data = ' ';
359         validInput = false;
360         //validation, looping till get a valid input
361         while (!validInput) {
362             std::getline(std::cin, input);
363             std::cin.clear();
364             if (input.empty()) //skip empty entries
365                 continue;
366             if (input == "$") {
367                 activeLoop = false;
368                 break;
369             }
370             try {
371                 data = input[0];
372                 validInput = true;
373             }
374             catch (const std::invalid_argument& e) {
375                 std::cout << "Please enter a valid char" << std::endl;
376                 validInput = false;
377             }
378         }
379         //to skip the pushing if the user wants to exit
380         if (!activeLoop) {
381             break;
382         }
383         charStack->push(data);
384         charStack->printStack();
385     }
386     break;

```

```

387     case DT_STRING:
388         stringStack = new StackADT<std::string>(stackSize);
389         //loop that runs to accept user input
390         while (activeLoop) {
391             std::cout << "String Stack Testing, Max Stack Size : " + std::to_string(stackSize)
392                 + ", Number of data : " + std::to_string(stringStack->getSize()) << std::endl;
393             std::cout << "Enter $ if you wish to stop the process." << std::endl;
394             std::getline(std::cin, input);
395             std::cin.clear();
396             if (input.empty()) //skip empty entries
397                 continue;
398             //to skip the pushing if the user wants to exit
399             if (input == "$") {
400                 activeLoop = false;
401                 break;
402             }
403
404             stringStack->push(input);
405             stringStack->printStack();
406         }
407         break;
408     default:
409         break;
410 }
411
412 }
413
414 if (intStack) {
415     delete intStack;
416     intStack = NULL;
417 }
418 if (charStack) {
419     delete charStack;
420     charStack = NULL;
421 }
422 if (floatStack) {
423     delete floatStack;
424     floatStack = NULL;
425 }
426 if (doubleStack) {
427     delete doubleStack;
428     doubleStack = NULL;
429 }
430 if (stringStack) {
431     delete stringStack;
432     stringStack = NULL;
433 }
434 }
435 if (intStack) {
436     delete intStack;
437     intStack = NULL;
438 }
439 if (charStack) {
440     delete charStack;
441     charStack = NULL;
442 }
443 if (floatStack) {
444     delete floatStack;
445     floatStack = NULL;
446 }
447 if (doubleStack) {
448     delete doubleStack;
449     doubleStack = NULL;
450 }
451 if (stringStack) {
452     delete stringStack;
453     stringStack = NULL;
454 }
455
456 return 0;
457 }

```

Question 2

2.1 Problem Statement

Write a program that reads in a sequence of characters, and determines whether its parentheses, braces, and curly braces are “balanced.” Your program should read one line of input containing what is supposed to be a properly formed expression in algebra and tells whether it is in fact legal. The expression could have several sets of grouping symbols of various kinds, (), [], and {}. Your program needs to make sure that these grouping symbols match up properly. Analyse the efficiency of your implementation and provide a detailed discussion of its time and space complexity.

2.2 Requirements/Specification

Given any algebraic statement, (e.g. $-b \pm \left[\sqrt{\{b^2\} - (4)(a)(c)} \right] / 2(a)$), determine if the braces are balanced; That is, if the number of opening braces match the number of closing braces, and that the first closing brace matches with the last opening brace. The algorithm expects a properly formed expression in algebra as a string and outputs either `True` or `False`.

2.3 User Guide

To run the program, simply type `python main.py` in a terminal window. The program then prompts the user for an algebraic statement. If the statement is balanced, the program returns `True` and vice versa. No external libraries other than the standard `Python 3` libraries are required.

2.4 Structure/Design

The algorithm works by pushing opening brackets to a stack by looping through all bracket characters in the original statement. When encountering a closing bracket, it pops the last element of the stack and compares if they are complementary. If at any point the check fails, the algorithm returns `False` and ends the loop prematurely. At the end of the loop, the algorithm checks that the stack is empty. If it is, it returns `True` and `False` otherwise.

Algorithm 1: Bracket balance checker

Input: str statement

```
1 Function is_balanced(statement: str) is
2   statement  $\leftarrow$  all brackets from statement;
3   bracket_pairings  $\leftarrow$  {opening_bracket : closing_bracket};
4   if len(statement) mod 2  $\neq$  0 then
5     | return False;
6   end
7   stack = [];
8   foreach character in statement do
9     | if character is an opening bracket then
10    |   stack.push(character);
11    | end
12    | else if bracket_pairing[stack.pop()]  $\neq$  character then
13    |   | return False;
14    | end
15  end
16  return len(stack) == 0;
17 end
```

As the algorithm iterates through the input string only once, the time complexity is $O(n)$ for a given input string of length n . The opening brackets are iteratively pushed to and popped from a stack, so the space complexity is $O(n)$ as well.

Note:-

The time complexity of [Regular Expressions](#) and [Stack Operations](#) for insertion and deletion are known to be $O(n)$ and $O(1)$ respectively, so the time and space complexity of the algorithm remains at $O(n)$.

2.5 Limitations

While the algorithm determines perfectly if the brackets within any given algebraic statement are balanced, it does not check if the statement itself is a properly formed algebraic statement. Furthermore, it does not check that brackets on two sides of a given equality are balanced, as it only checks for bracket placement relative to other brackets in the entire input string (i.e. $([0]\{= \}2)$ will be evaluated as balanced).

2.6 Testing

Testing is handled by the `tests.py` file, which generates t input strings of up to length l , of which $n/2$ inputs are valid, and the other half are invalid. To generate valid inputs, the generator randomly selects an opening bracket or a closing bracket that matches the last opening bracket until reaching the halfway point of the string length, at which point it iteratively closes all the remaining open brackets. An example output of running the tests is shown in [Figure 2.1](#).

Figure 2.1: Test output

```
Test 974:      ()): Passed (False)
Test 975:      [{[]}]: Passed (True)
Test 976:      [[][: Passed (False)
Test 977:      (): Passed (True)
Test 978:      {{()}}: Passed (True)
Test 979:      {()}: Passed (True)
Test 980:      ({[{}]): Passed (True)
Test 981:      {((()({: Passed (False)
Test 982:      {: Passed (False)
Test 983:      ({[{}]): Passed (True)
Test 984:      [((: Passed (False)
Test 985:      [(){}: Passed (False)
Test 986:      (((({{}))))) : Passed (True)
Test 987:      ([: Passed (False)
Test 988:      [: Passed (False)
Test 989:      {}: Passed (False)
Test 990:      {{()}}: Passed (True)
Test 991:      {}]: Passed (False)
Test 992:      ()){: Passed (False)
Test 993:      {}){}: Passed (False)
Test 994:      [)]{}): Passed (False)
Test 995:      ([[: Passed (False)
Test 996:      []]: Passed (False)
Test 997:      []{}{}: Passed (False)
Test 998:      {[]()[]]: Passed (True)
Test 999:      {[}][: Passed (False)
Test 1000:     (}}}: Passed (False)
Passed 1000 out of 1000 tests
```

2.6.1 Invalid Inputs

Invalid inputs are a superset of valid inputs, thus the selection of brackets to insert at any given point is expanded to include all invalid closing brackets as well. The generator then selects a random index i to insert random brackets until the max length is reached. Then, the generator checks if the number of opening brackets match the number of closing brackets for each type of bracket. If they match, a random index is selected again to either insert or remove a bracket. This ensures that we also deal with the case that the length of the input string is odd. The generator then provides the input string and whether the string is valid to a checker function, which compares the output of the developed algorithm with the validity of the input string. To run the tests, a user may run `python test.py --tests \{number of tests\} --length \{max length of input strings\}`. The output will show how many tests the algorithm passes, and what the generated input strings were for each test.

2.7 Listings

2.7.1 Algorithm

```
1 import re
2
3 def main(statement:str):
4     return is_balanced(statement)
5
6 def is_balanced(statement:str) -> bool:
7     bracket_pairing = {
8         "{": "}" ,
9         "[": "]" ,
10        "(": ")"
11    }
12    # fast check
13    statement = re.sub(r"[A-Za-z0-9\*\-\+\^\\/\=]", "", statement)
14    if len(statement) % 2 != 0: return False
15    brackets = [bracket for bracket in bracket_pairing.keys()] \
16        + [bracket for bracket in bracket_pairing.values()]
17    stack = [ ]
18    for char in statement:
19        if char in brackets:
20            if char in bracket_pairing.keys():
21                stack.append(char)
22            else:
23                try:
24                    if bracket_pairing[stack.pop()] != char:
25                        return False
26                except IndexError:
27                    return False
28    return len(stack) == 0
29
30 if __name__ == "__main__":
31     res = main(input("Enter an algebraic statement: "))
32     print(res)
```

2.7.2 Testing

```
1 import argparse
2 import random
3 from main import is_balanced
4
5 def test_is_balanced(iters: int, max_length:int=10):
6     """Tests the is_balanced function over a given number of iterations.
7
8     Args:
9         iters (int): number of iterations
10    """
11    results = []
12    for i in range(iters):
13        statement, proper = statement_generator(random.randint(1, max_length))
14        print(f"Test {i+1}: \t{statement}", end=" ")
15        res = is_balanced(statement)
16        if res == proper:
17            print(f"Passed ({res})")
18        else:
19            print(f"Failed: {res} (should be {proper})")
20        results.append(res == proper)
21    print(f"Passed {results.count(True)} out of {iters} tests")
22
23 def statement_generator(length: int):
24     """Generates a random algebraic statement of a given length.
25
26     Args:
27         length (int): length of the statement
28    """
```

```

29 Returns:
30     str: random algebraic statement
31     ""
32     length // 2
33     bracket_pairing = {
34         "{": "}" ,
35         "[": "]" ,
36         "(": ")"
37     }
38     brackets = [bracket for bracket in bracket_pairing.keys()] \
39         + [bracket for bracket in bracket_pairing.values()]
40     ret = ""
41     state = random.choice([True, False])
42     ret += random.choice([bracket for bracket in bracket_pairing.keys()])
43     stack = [ret[0]]
44     for _ in range(length):
45         if state:
46             candidates = [b for b in bracket_pairing.keys()]
47             if ret[-1] in bracket_pairing.keys():
48                 candidates += [bracket_pairing[ret[-1]]]
49             ret += random.choice(candidates)
50             if ret[-1] in bracket_pairing.values():
51                 stack.pop()
52             else:
53                 stack.append(ret[-1])
54         else:
55             ret += random.choice(brackets)
56     for _ in range(len(stack)):
57         if state:
58             ret += bracket_pairing[stack.pop()]
59         else:
60             ret += random.choice(brackets)
61     if not state:
62         n_additions = random.randint(0, length)
63         insertion_index = random.randint(0, len(ret))
64         additions = [random.choice(brackets) for _ in range(n_additions)]
65         ret = ret[:insertion_index-1] + "".join(additions) + ret[insertion_index+1:len(ret)+1-
66         n_additions]
67     # count the number of bracket pairs
68     bracket_counts = {(k, v): 0 for k, v in bracket_pairing.items()}
69     for char in ret:
70         for k, v in bracket_pairing.items():
71             if char == k:
72                 bracket_counts[(k, v)] += 1
73             elif char == v:
74                 bracket_counts[(k, v)] -= 1
75     # if the statement is potentially balanced, either remove or add a random character.
76     if all([count == 0 for count in bracket_counts.values()]):
77         # remove a random character
78         if len(ret) > 2:
79             loc = random.randint(1, len(ret)-1)
80             ret = ret[:loc] + ret[loc+1:]
81         else:
82             ret += random.choice(brackets)
83     return (ret, state)
84
85 def main(tests: int=1000, max_length:int=10):
86     test_is_balanced(tests, max_length)
87
88 if __name__ == "__main__":
89     parser = argparse.ArgumentParser()
90     parser.add_argument("-t", "--tests", type=int, default=1000, help="number of tests to run")
91     parser.add_argument("-l", "--length", type=int, default=10, help="maximum length of the
92     statement")
93     args = parser.parse_args()
94     main(args.tests, args.length)

```


Question 3

3.1 Problem Statement

Write an array-based implementation of the array list ADT that achieves $O(1)$ time for insertion and removals at the front and at the end of the array list. Your implementation should also provide for an $O(1)$ time `get(i)` method. Assume that overflow does not occur. Explain and justify why your implementation has achieved the stated time complexity requirements.

3.2 Requirements/Specification

The program creates an array list with functions to insert and remove items at the front and back of the array as well as a function `get(i)` to get the element in the array at index `i`.

The function `insertItem(int input, int position)` inserts an item into the array. The argument `input` will be the item and the argument `position` will decide where the item will be inserted into the array. If the position is 0, the item will be inserted at the front of the array. If the position is 1, the item will be inserted at the end of the array. For example, `insertItem(15, 1)` will insert the item 15 at the end of the array.

The function `removeItem(int position)` removes an item from the array. The argument `position` will decide whether the item will be removed from the start or from the end of the array. If the position is 0, the item at the start of the array will be removed. If the position is 1, the item at the end of the array will be removed. For example, `removeItem(0)` will remove the item at the start of the array.

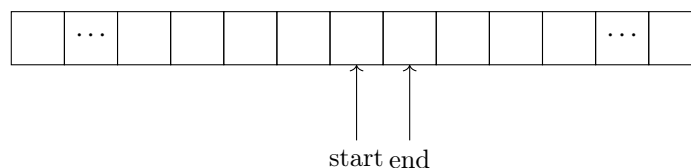
3.3 User Guide

To run the program, a user may compile the source code with their C compiler of choice. Then, simply run the executable in a command line or terminal.

3.4 Structure/Design

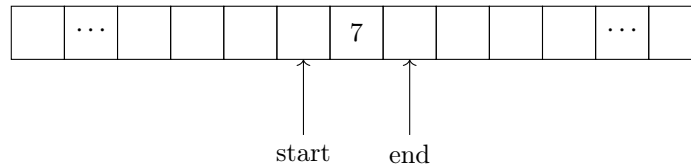
The program starts by creating an array with a large size (e.g. `int arr[100];`) and variables `start` (e.g. `int start = 50;`) and `end` (e.g. `int end = 51;`) to represent the starting and ending index of the array respectively. The array will be initialised somewhere in the middle of the array with the start index and end index next to each other. The array is between the start and end index of the array which will then expand or contract from there depending on where the items are inserted or removed (at the front or at the back of the array).

Figure 3.1: Initialising the array



To add an item at the start of the array, the start position of the array will first take in the value of the item, then the start index will shift left (i.e. `start--`;). Similarly, to add an item at the end of the array, the end position of the array will take in the value of the item, then the end index will shift right (i.e. `end++`;). Since the function does not need to iterate through the array to insert the items, the time complexity for inserting items is $O(1)$.

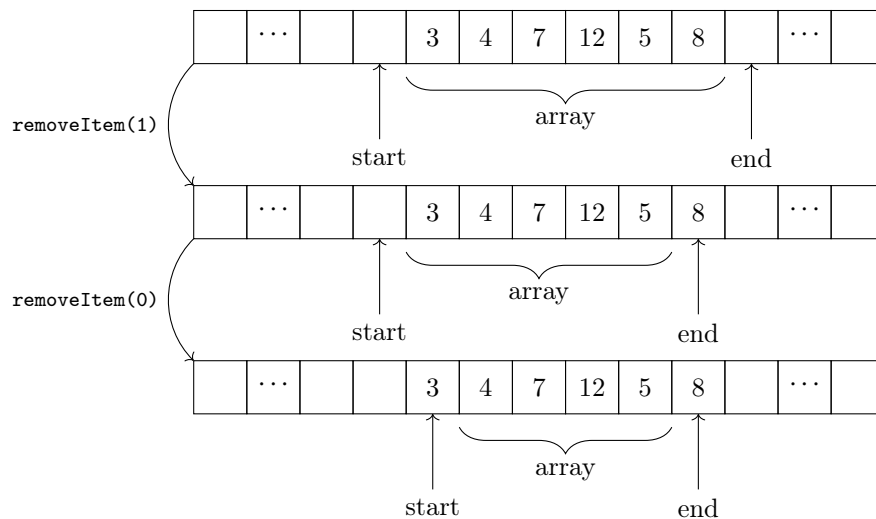
Figure 3.2: Insertion



Even though the problem statement assumes that overflow does not occur, safety measures were put in place to prevent overflowing. If the start index is at the start of the array (i.e. `start == 0`) or the end index is at the end of the array (i.e. `end == 99`), trying to add another item into the array will print “Overflow!”

To remove an item from the start of the array, the start index will simply shift right (i.e. `start++`;), and to remove an item from the end of the array, the end index will simply shift left (i.e. `end--`;). Since the function only shifts the start or end index when removing items and does not iterate through the array, the time complexity for removing items is $O(1)$.

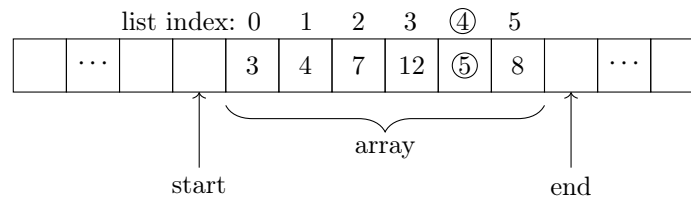
Figure 3.3: Removal of nodes



The array is deemed empty when the start and end index are next to each other. When a function is called to remove another item while the array is empty, it will print “List is empty!”.

To get the item at index `i` of the array, the function starts counting at the start of the array (i.e. `arr[start + 1]`) and returns the item at index `i` of the array (i.e. `arr[start + 1 + i]`). Since this function returns the item at the stated index and does not require iteration through the array, the time complexity for this function is $O(1)$.

Figure 3.4: List access example (accessing index 4)



3.5 Limitations

The limitation of this array list ADT is that it has a maximum size depending on the value used during initialization and has a finite number of items that can be added into the array. The function only stops the list from overflowing, but it does not have a solution to overflowing (e.g. increasing the size of the array).

3.6 Testing

Testing is handled by the `main()` function in the program. Initially, the array is empty.

First, the program tests the `insertItem()` function by adding elements to the start and end of the array. Adding an item to an invalid position will output `“Invalid Input!”`.

Figure 3.5: `insertItem()` testing

```
Initial list: []
After adding 10 items to the start of the list: [19,18,17,16,15,14,13,12,11,10]
After adding 10 items to the end of the list: [19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18]
After adding the item '13' to the start of the list: [13,19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18]
After adding the item '14' to the start of the list: [13,19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18,14]
Adding the item '15' to an invalid position:
Invalid Input!
Current list: [13,19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18,14]
```

Next, the program tests the `removeItem()` function by removing an item at the start and at the end of the array. Removing an item at an invalid location will output `“Invalid Input!”`.

Figure 3.6: `removeItem()` testing

```
After removing item at the start of the list: [19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18,14]
After removing item at the end of the list: [19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18]
Removing item at an invalid position:
Invalid Input!
Current list: [19,18,17,16,15,14,13,12,11,10,0,2,4,6,8,10,12,14,16,18]
```

Thirdly, the program tests the `get()` function at index 5. Using an out of range index will output `“Index Out Of Range!”`.

Figure 3.7: `get()` testing

```
Item at index 5: 14
Getting an item with an index that is out of range (index 20):
Index out of range!
Item at index 20: 0
```

Subsequently, the program will attempt to overflow the array at the start and end. The program will then output `“Overflow!”` when items cannot be added to that respective position any more.

```
Attempting to overflow at the start of the list:  
Overflow!  
Overflow!  
Current list: [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,19,18,17,16,15,14,13,12,1  
1,10,0,2,4,6,8,10,12,14,16,18]  
Attempting to overflow at the end of the list:  
Overflow!  
Overflow!  
Current list: [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,19,18,17,16,15,14,13,12,1  
1,10,0,2,4,6,8,10,12,14,16,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Figure 3.9: Empty array removeItem() testing

```
After removing every items in the list: []
Removing an item from an empty list:
List is empty!
Current list: []
```

```

1 #include <stdio.h>
2
3 int arr[100];
4 int start = 50;
5 int end = 51;
6
7 int insertItem(int input, int position) {    // position: 0 for inserting at the front; 1 for
8     inserting at the end
9     if (position == 0) {
10         if (start == 0) {
11             printf("Overflow!\n");
12         }
13         else {
14             arr[start] = input;
15             start--;
16         }
17     }
18     else if (position == 1) {
19         if (end == 99) {
20             printf("Overflow!\n");
21         }
22         else {
23             arr[end] = input;
24             end++;
25         }
26     }
27     else {
28         printf("Invalid Input!\n");
29     }
30 }
31
32 int removeItem(int position) {                // position: 0 for removing at the front; 1 for
33     removing at the end
34     if (end - start == 1) {
35         printf("List is empty!\n");
36         return 0;
37     }
38     if (position == 0) {
39         start++;
40     }
41     else if (position == 1) {
42         end--;
43     }
44     else {
45         printf("Invalid Input!\n");
46     }
47 }

```

```

44     }
45 }
46
47 int get(int i) {                                // get(i) returns the item in the list at index i
48     if (i >= 0 && i < end - start - 1) {
49         return arr[start + 1 + i];
50     }
51     else {
52         printf("Index out of range!\n");
53         return NULL;
54     }
55 }
56
57
58 int printList() {
59     printf("[");
60     for (int i = 0; i < end - start - 2; i++) {
61         printf("%d,", arr[start + 1 + i]);
62     }
63     if (end - start == 1) {
64         printf("]\n");
65     }
66     else {
67         printf("%d]\n", arr[end - 1]);
68     }
69 }
70
71 int main() {
72     // initial list
73     printf("Initial list: ");
74     printList();
75
76     // inserting 10 items to the start of the list
77     for (int i = 0; i < 10; i++) {
78         insertItem(i + 10, 0);
79     }
80     printf("After adding 10 items to the start of the list: ");
81     printList();
82
83     // inserting 10 items to the end of the list
84     for (int i = 0; i < 10; i++) {
85         insertItem(i * 2, 1);
86     }
87     printf("After adding 10 items to the end of the list: ");
88     printList();
89
90     // inserting the item '13' at the start of the list
91     insertItem(13, 0);
92     printf("After adding the item '13' to the start of the list: ");
93     printList();
94
95     // inserting the item '14' at the end of the list
96     insertItem(14, 1);
97     printf("After adding the item '14' to the start of the list: ");
98     printList();
99
100    // inserting the item '15' at an invalid position
101    printf("Adding the item '15' to an invalid position:\n");
102    insertItem(15, 2);
103    printf("Current list: ");
104    printList();
105
106    // removing the item at the start of the list
107    removeItem(0);
108    printf("After removing item at the start of the list: ");
109    printList();
110
111    // removing the item at the end of the list
112    removeItem(1);
113    printf("After removing item at the end of the list: ");
114    printList();

```

```

115
116 // removing the item at an invalid position
117 printf("Removing item at an invalid position:\n");
118 removeItem(2);
119 printf("Current list: ");
120 printList();
121
122 // get the item at index 5 of the list
123 printf("Item at index 5: %d\n", get(5));
124
125 // get the item at index 20 of the list
126 printf("Getting an item with an index that is out of range (index 20):\n");
127 printf("Item at index 20: %d\n", get(20));
128
129 // attempting to overflow at the start of the list
130 printf("Attempting to overflow at the start of the list:\n");
131 for (int i = 0; i < 42; i++) {
132     insertItem(1, 0);
133 }
134 printf("Current list: ");
135 printList();
136
137 // attempting to overflow at the end of the list
138 printf("Attempting to overflow at the end of the list:\n");
139 for (int i = 0; i < 40; i++) {
140     insertItem(0, 1);
141 }
142 printf("Current list: ");
143 printList();
144
145 // removing all the items in the list
146 for (int i = 0; i < 98; i++) {
147     removeItem(1);
148 }
149 printf("After removing every items in the list: ");
150 printList();
151
152 // attempting to remove an item from an empty list
153 printf("Removing an item from an empty list:\n");
154 removeItem(1);
155 printf("Current list: ");
156 printList();
157 }

```

Question 4

4.1 Problem Statement

Write a recursive algorithm to check that a sentence is a palindrome (ignoring blanks, lower case and upper case differences, and punctuation marks, so that “Madam, I’m Adam” is accepted as a palindrome). Analyse the efficiency of your implementation and provided a detailed discussion of its time complexity.

Example 4.1.1

Please enter a sentence: Madam, I’m Adam Check if “Madam, I’m Adam” is a palindrome: True

4.2 Requirements/Specification

This program is supposed to compare the characters in the sentence. First and last, Second and second last etc. If it matches, it is a palindrome. Some assumptions/conditions would be ignoring blanks, lower case, upper case differences, and punctuation marks. Some assumptions/conditions would be ignoring blanks, lower case, upper case differences, and punctuation marks. Empty strings will be considered as a palindrome too.

4.3 User Guide

1. Click on the “Run” button in the IDE to run the program with python. Alternatively, running `python <filename>.py` will run the program.
2. Input a sentence when prompted in the command line interface.
3. The resulting output will show whether the input sentence was a palindrome.

4.4 Structure/Design

The design of the system is that it will remove all punctuations and black spaces in the string then change all uppercase characters to lowercase characters before passing through the `palindrome` function. After the system has recorded the new string, a function for `palindrome` and `isPalindrome` will run to check the string and determine whether it is a palindrome.

For the `palindrome` function, if there is only one character it will return `true`. If the first and last characters do not match, it will return `false`. For more than 2 characters, it will check the middle substring whether the characters match.

For the `isPalindrome` function, if it is an empty string, it will be considered a palindrome and return `true`.

The operation of removing punctuations and blanks, and converting the string to lowercase is $O(n)$, n being the length of the string. As it iterates through the function `palindrome`, it checks if the first and last characters match, hence the time complexity of this function is $O(n)$. The function `isPalindrome` checks if the string is empty, hence the time complexity of the entire algorithm is $O(n)$.

4.5 Limitations

The algorithm assumes an ASCII input, thus input strings with extended UTF-8 encoding may result in erroneous output.

4.6 Testing

From the `main.py` file, users are able to manually input a string they would like to check and if the string is a palindrome, it will return `True`. However, when the user inputs invalid inputs such as string that is not a palindrome, it will return `False`.

Figure 4.1: Valid and invalid inputs

```
> python main.py
Enter a string: A Man, A Plan, A Canal, Panama
True

> python main.py
Enter a string: 123
False
```

From the `test.py` file, the input strings are generated randomly with 50% being palindromes. The user can set the arguments `-i` and `-l` to set the number of tests and max generated string length (ASCII letters only) respectively. ASCII symbols are generated in between ASCII letters in order to test the filtering capabilities of the algorithm. An example output of running the testing script is shown in [Figure 4.2](#).

Figure 4.2: Running the testing script

```
> python test.py -i 10 -v

Test 1: $a#D.o'T/x*i$N>Y'Y(N:i!x!T+o]D+a: Passed (True)
Test 2: [Y.a'i:z@z"i.a"Y: Passed (True)
Test 3: ]J%M'M]z)u:r/r]u^z,M{M!J: Passed (True)
Test 4: ]p@F}P)E(m&V+q+U;D+i?D~y=k`r: Passed (False)
Test 5: "B_D/x#G#s}t/X$a#a#X:t,s~G*x#D*B: Passed (True)
Test 6: "x(N.j.U"v~b-n[M<j.B@s>i]W: Passed (False)
Test 7: #j@r@P$p%m}r_Q/Q@r`m(p{P*r*j: Passed (True)
Test 8: ;M#P+s:D|O[H)O;p>f^O:d:j.j)d(O/f,p(O^H<O!D<s+P"M: Passed (True)
Test 9: "h#d!S&m-G<l[Q=O>s$Y_V|V>Y*s>O;Q%l@G.m;S,d@h: Passed (True)
Test 10:          #z(d: Passed (False)
Passed 10 out of 10 tests
```

4.7 Listings

4.7.1 Algorithm

```
1 def main():
2     # user input for string
3     input_str = input("Enter a string: ")
4     # Output
5     if isPalindrome(input_str):
6         print("True")
7     else:
8         print("False")
9
10 def palindrome(str2, s, e):
11     # If there is only one character
12     if (s == e):
13         return True
14
15     # If first and last characters do not match
16     if (str2[s] != str2[e]):
```



```

17     return False
18
19     # for > 2 characters, checking for middle substring
20     if (s < e + 1):
21         return palindrome(str2, s + 1, e - 1)
22     return True
23
24 # for empty string, it will be considered as palindrome too
25 def isPalindrome(input_str):
26
27     # remove all punctuations and blank spaces
28     str2 = ''.join(i for i in input_str if i.isalnum())
29     # lower uppercase characters
30     str2 = str2.lower()
31     n = len(str2)
32
33     if n == 0:
34         return True
35     return palindrome(str2, 0, n - 1)
36 if __name__ == "__main__":
37     main()

```

4.7.2 Testing

```

1  import argparse
2  import random
3  import string
4
5  from main import isPalindrome
6
7  def string_generator(length: int, is_palindrome: bool):
8      half_length = length // 2
9      odd = length % 2
10     letter = string.ascii_letters
11     generated_string = ''.join(random.choice(letter) for i in range(length))
12     if is_palindrome:
13         half = generated_string[:half_length + odd][::-1]
14         generated_string = generated_string[:half_length + odd] + half
15     elif generated_string[0].lower() == generated_string[-1].lower():
16         generated_string += random.choice([i for i in letter if i.lower() != generated_string[0].lower()])
17     elif len(generated_string) == 1:
18         generated_string += random.choice([i for i in letter if i.lower() != generated_string[0].lower()])
19     symbols = string.punctuation
20     generated_string = ''.join(random.choice(symbols) + i for i in generated_string)
21     return generated_string
22
23 def test_is_palindrome(iters: int, max_length: int=10, verbose: bool=False, hackerman: bool=False):
24     """Tests the isPalindrome function over a given number of iterations.
25
26     Args:
27         iters (int): number of iterations
28     """
29     results = []
30     if hackerman:
31         end = '\x1b[2K\r'
32     else:
33         end = "\n"
34     i = 0
35     while True:
36         test_state = random.choice([True, False])
37         generated_string = string_generator(random.randint(1, max_length), test_state)
38         res = isPalindrome(generated_string)
39         p = ""
40         if verbose or res != test_state:
41             print(f"{end}Test {i+1}: \t{generated_string}: ", end=" ")
42         if res == test_state:

```

```

43     p = f"Passed ({res})" if verbose else ""
44     else:
45         p = f"Failed: {res} (should be {test_state})"
46     if len(p):
47         print(f"{p:<50}", end="")
48     results.append(res == test_state)
49     i += 1
50     if i == iters:
51         break
52     print(f"{end}Passed {results.count(True)} out of {iters} tests")
53
54 if __name__ == "__main__":
55     parser = argparse.ArgumentParser(description="Test the isPalindrome function")
56     parser.add_argument("-i", "--iters", type=int, default=100000, help="number of iterations")
57     parser.add_argument("-l", "--length", type=int, default=25, help="maximum length of the string")
58     parser.add_argument("-v", "--verbose", action="store_true", help="verbose mode")
59     parser.add_argument("-H", "--hackerman", action="store_true", help="hackerman mode")
60     args = parser.parse_args()
61     test_is_palindrome(args.iters, args.length, args.verbose, args.hackerman)

```

Question 5

5.1 Problem Statement

- (a) Explain the meaning of stability in a sorting algorithm.
- (b) Explain a situation why stability in sorting is desired.
- (c) State which algorithms are stable. Prove it with an implementation of a stable and an unstable sorting algorithm. Discuss in detail your justification.
 - i. Selection sort
 - ii. Insertion sort
 - iii. Quick sort

Solution:

- (a) Stability refers to whether a sorting algorithm respects the relative positions of elements with equal values during the sort. An unstable sorting algorithm would not preserve the order of elements with equal values.
- (b) One situation could be where there is a line of people queueing who need to be ordered by age. If there are two people standing in line who are the same age, a stable sort will ensure that the order of the person who arrived first will stay the same.
- (c) All three algorithms can have stable and unstable implementations.

■

5.2 Requirements/Specification

The program implements python code for selection, insertion and quick sort algorithms. In order to clearly demonstrate the stability of each algorithm, the program will accept an unsorted list of 2-element tuples, where the first element is a unique identifier for the tuple and the second element is the value that is compared against other values for the sorting algorithms. Some example inputs are: `[("a", 3), ("b", 3)]`, `[["a", 3], ["b", 3]]`, and `[[123, 3], [456, 3]]`.

5.3 User Guide

The program can be run from the command line with `python <filename>.py`. Since the assignment requires a proof by implementation, the input data has been pre-defined in the script. If a user would like to input their own data, they may do so by changing the following variables accordingly.

- `data_selection` – for selection sort
- `data_insertion` – for insertion sort
- `data_quick` – for quick sort

5.4 Structure/Design

The code for all 3 sorting algorithms are contained in functions, so for a user to see the sorting processes, they would have to call each sorting algorithm's respective functions. All 3 sorts will print the elements in the array during each iteration of the sort.

5.5 Limitations

As the program uses tuples to demonstrate the stability of the algorithms, the program will not take in any other formats like simple lists as input. For instance, `[26, 54, 56, 32]` and `["a", "b", "c"]` are not valid inputs.

5.6 Testing

The stability of the algorithm may be demonstrated by showing the unsorted input array and how it is iteratively sorted until the algorithm outputs the sorted array. For the arrays used to test the algorithms, two elements are declared with equal value, with the first of the pair having a 'first' identifier, and the second having a 'second' identifier. This will allow users to track the relative positions of these two elements of equal value. The output of the python script is shown in [Figure 5.1](#).

Figure 5.1: Outputs of stable and unstable variants of the sorting algorithms

```
> python .\question_5.py
====Stable Selection Sort====
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('first', 3), ('', 5), ('', 4), ('second', 3)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
====Unstable Selection Sort====
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('second', 3), ('', 5), ('first', 3), ('', 4)]
[('second', 3), ('first', 3), ('', 5), ('', 4)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
====Stable Insertion Sort====
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('first', 3), ('', 4), ('', 5), ('second', 3)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
====Unstable Insertion Sort====
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('', 4), ('', 5), ('first', 3), ('second', 3)]
[('first', 3), ('', 4), ('', 5), ('second', 3)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
====Stable Quick Sort====
[('first', 3), ('', 5), ('', 4), ('second', 3)]
[('first', 3), ('', 5), ('', 4), ('second', 3)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
[('first', 3), ('second', 3), ('', 4), ('', 5)]
====Unstable Quick Sort====
[('first', 3), ('', 5), ('', 4), ('second', 3)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
[('second', 3), ('first', 3), ('', 4), ('', 5)]
```

5.7 Listings

```
1 #Function for selection sort (Stable)
2 def selectionSort(array, size, stable=True):
3     for ind in range(size):
4         min_index = ind
5
6         if stable:
7             for j in range(ind + 1, size):
8                 # select the minimum element in every iteration
9                 if array[j][1] < array[min_index][1]:
10                     min_index = j
11         else:
12             for j in range(size - 1, ind, -1):
13                 if array[j][1] < array[min_index][1]:
14                     min_index = j
15             # swapping the elements to sort the array
16             (array[ind], array[min_index]) = (array[min_index], array[ind])
17             print(array)
18
19 # Function to do insertion sort (Stable)
20 def insertionSort(arr, stable=True):
21     # Traverse through 1 to len(arr)
22     for i in range(1, len(arr)):
23
24         key = arr[i]
25         # Move elements of arr[0..i-1], that are
26         # greater than key, to one position ahead
27         # of their current position
28         j = i-1
29         if stable:
30             while j >= 0 and key[1] < arr[j][1]:
31                 arr[j + 1] = arr[j]
32                 j -= 1
33             arr[j + 1] = key
34         else:
35             while j >= 0 and key[1] <= arr[j][1]:
36                 arr[j + 1] = arr[j]
37                 j -= 1
38             arr[j + 1] = key
39         print(arr)
40     return arr
41
42 # Python program for implementation of Quicksort Sort (Stable)
43 # Function to find the partition position
44 def partition(array, low, high, stable=True):
45
46     # choose the rightmost element as pivot
47     if stable:
48         pivot = array[high][1]
49     else:
50         pivot = array[low][1]
51
52     # pointer for greater element
53     i = low - 1
54
55     # traverse through all elements
56     # compare each element with pivot
57     for j in range(low, high):
58         if (stable and array[j][1] <= pivot) or (not stable and array[j][1] < pivot):
59             # If element smaller than pivot is found
60             # swap it with the greater element pointed by i
61             i += 1
62             # Swapping element at i with element at j
63             (array[i], array[j]) = (array[j], array[i])
64             print(array)
65     # Swap the pivot element with the greater element specified by i
66     (array[i + 1], array[high]) = (array[high], array[i + 1])
67     # Return the position from where partition is done
68     return i + 1
```

```

69
70 # function to perform quicksort
71 def quickSort(array, low, high, stable=True):
72     if low < high:
73
74         # Find pivot element such that
75         # element smaller than pivot are on the left
76         # element greater than pivot are on the right
77         pi = partition(array, low, high, stable)
78
79         # Recursive call on the left of pivot
80         quickSort(array, low, pi - 1)
81
82         # Recursive call on the right of pivot
83         quickSort(array, pi + 1, high)
84
85 data_selection = [("", 4), ("", 5), ("first", 3), ("second", 3)]
86 #print("Unsorted Array: ", data_selection)
87 print("====Stable Selection Sort====")
88 print(data_selection)
89 size = len(data_selection)
90 selectionSort(data_selection, size)
91
92 data_selection = [("", 4), ("", 5), ("first", 3), ("second", 3)]
93 print("====Unstable Selection Sort====")
94 size = len(data_selection)
95 print(data_selection)
96 selectionSort(data_selection, size, False)
97
98 print("====Stable Insertion Sort====")
99 data_insertion = [("", 4), ("", 5), ("first", 3), ("second", 3)]
100 print(data_insertion)
101 print(insertionSort(data_insertion))
102
103 print("====Unstable Insertion Sort====")
104 data_insertion = [("", 4), ("", 5), ("first", 3), ("second", 3)]
105 print(data_insertion)
106 print(insertionSort(data_insertion, False))
107
108 print("====Stable Quick Sort====")
109 data_quick = [("first", 3), ("", 5), ("", 4), ("second", 3)]
110 print(data_quick)
111 quickSort(data_quick, 0, len(data_quick) - 1)
112 print(data_quick)
113
114 print("====Unstable Quick Sort====")
115 data_quick = [("first", 3), ("", 5), ("", 4), ("second", 3)]
116 print(data_quick)
117 quickSort(data_quick, 0, len(data_quick) - 1, False)
118 print(data_quick)

```