INF1008
Data Structures & Algorithms

Group 10

Contents

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.

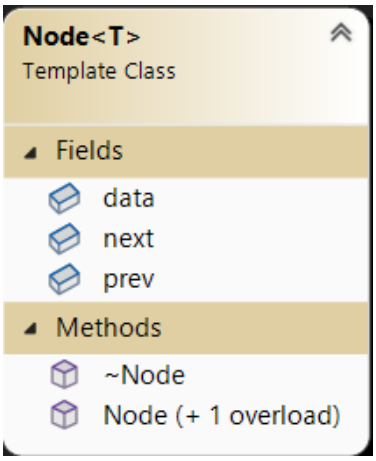
1.4 Structure/Design

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`.

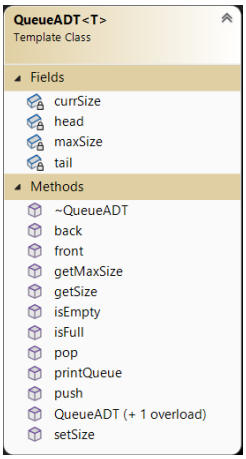
1.4.1 Node

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

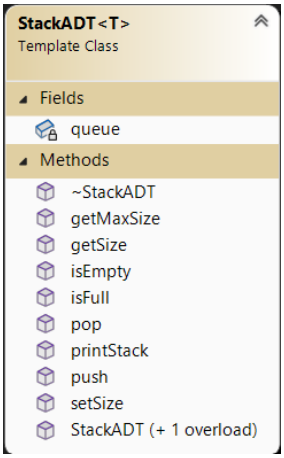
3. `Node<T>* head` – A pointer to the head node of the queue
4. `Node<T>* tail` – A pointer to the tail node of the queue

- 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 argumnet 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 alloted 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, the greater than or equal to comparator is used for added precaution
10. `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
11. `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
12. `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)` – Invbokes 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 progam.

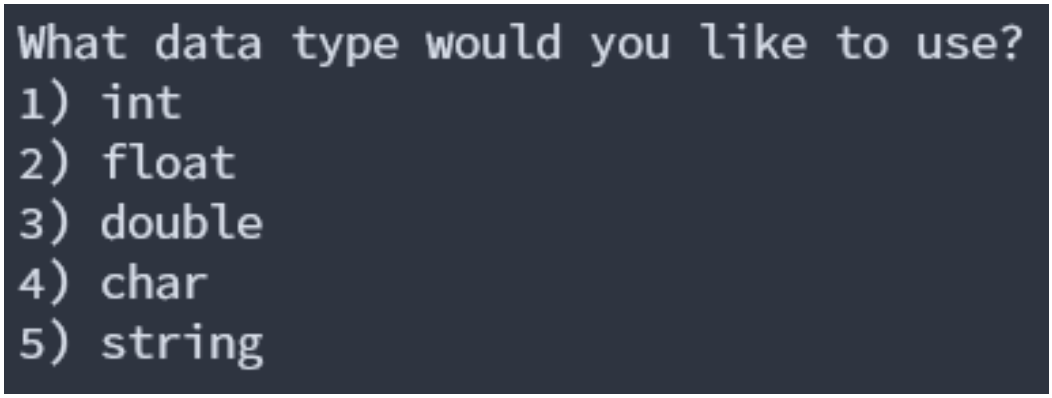
Figure 1.4: Command-line Interface prompt

```
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

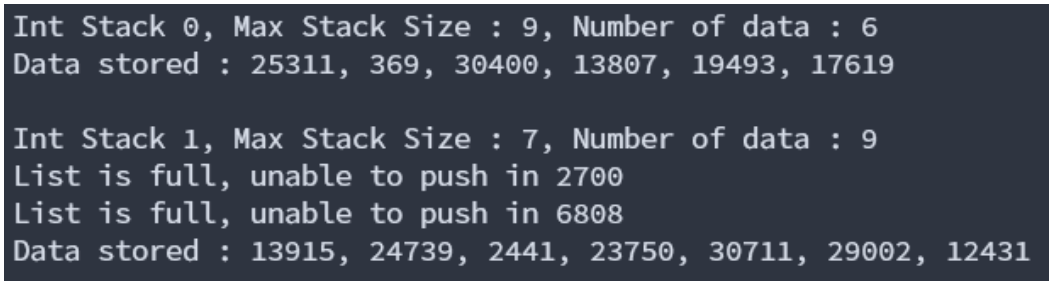
The program will prompt the user for the type of data they wish to store.

Figure 1.5: Option 1 prompt



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

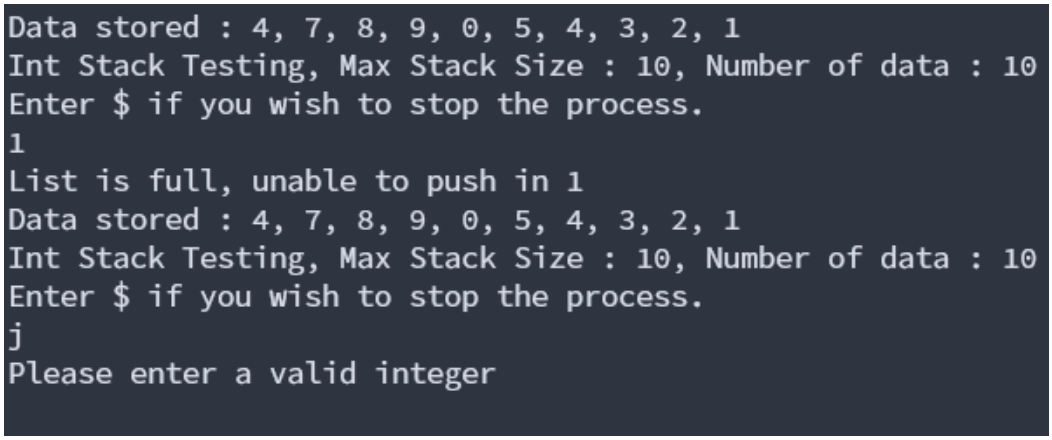


1.6.2 Option 2 – Manual Testing

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



1.7 Listings

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 ← all brackets from statement;
3      bracket_pairings ← {opening_bracket : closing_bracket};
4      if len(statement)  mod 2 ≠ 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()] ≠ 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 $O(n)$.

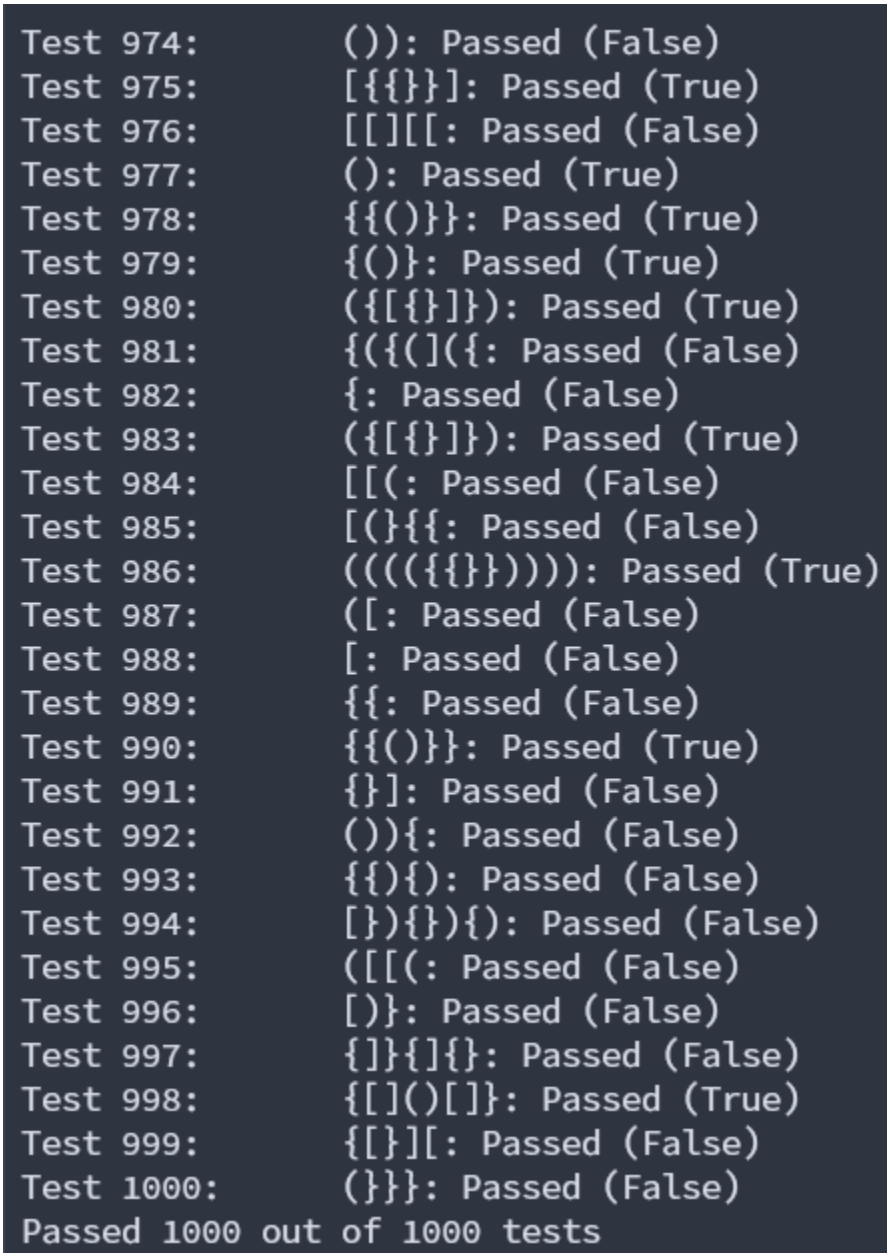
2.5 Limitations

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



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

```python
import re

def main(statement:str):
  return is_balanced(statement)

def is_balanced(statement:str) -> bool:
  bracket_pairing = {
    "{": "}",
    "[": "]",
    "(": ")"
  }
  # fast check
  statement = re.sub(r"[A-Za-z0-9\*\-\+\^\/\=]", "", statement)
  if len(statement) % 2 != 0: return False
  brackets = [bracket for bracket in bracket_pairing.keys()]\
    + [bracket for bracket in bracket_pairing.values()]
  stack = [ ]
  for char in statement:
    if char in brackets:
      if char in bracket_pairing.keys():
        stack.append(char)
      else:
        try:
          if bracket_pairing[stack.pop()] != char:
            return False
        except IndexError:
          return False
  return len(stack) == 0

if __name__ == "__main__":
  res = main(input("Enter an algebraic statement: "))
  print(res)
```

2.7.2 Testing

```python
import argparse
import random
from main import is_balanced

def test_is_balanced(iters: int, max_length:int=10):
  """Tests the is_balanced function over a given number of iterations.

  Args:
    iters (int): number of iterations
  """
  results = []
  for i in range(iters):
    statement, proper = statement_generator(random.randint(1, max_length))
    print(f"Test {i+1}:\t{statement}:", end=" ")
    res = is_balanced(statement)
    if res == proper:
      print(f"Passed ({res})")
    else:
      print(f"Failed: {res} (should be {proper})")
    results.append(res == proper)
  print(f"Passed {results.count(True)} out of {iters} tests")

def statement_generator(length: int):
  """Generates a random algebraic statement of a given length.

  Args:
    length (int): length of the statement

```

```
    Returns:
      str: random algebraic statement
    """
    length //= 2
    bracket_pairing = {
      "{": "}",
      "[": "]",
      "(": ")"
    }
    brackets = [bracket for bracket in bracket_pairing.keys()]\
      + [bracket for bracket in bracket_pairing.values()]
    ret = ""
    state = random.choice([True, False])
    ret += random.choice([bracket for bracket in bracket_pairing.keys()])
    stack = [ret[0]]
    for _ in range(length):
      if state:
        candidates = [b for b in bracket_pairing.keys()]
        if ret[-1] in bracket_pairing.keys():
          candidates += [bracket_pairing[ret[-1]]]
        ret += random.choice(candidates)
        if ret[-1] in bracket_pairing.values():
          stack.pop()
        else:
          stack.append(ret[-1])
      else:
        ret += random.choice(brackets)
    for _ in range(len(stack)):
      if state:
        ret += bracket_pairing[stack.pop()]
      else:
        ret += random.choice(brackets)
    if not state:
      n_additions = random.randint(0, length)
      insertion_index = random.randint(0, len(ret))
      additions = [random.choice(brackets) for _ in range(n_additions)]
      ret = ret[:insertion_index-1] + "".join(additions) + ret[insertion_index+1:len(ret)+1-
    n_additions]
      # count the number of bracket pairs
      bracket_counts = {(k, v): 0 for k, v in bracket_pairing.items()}
      for char in ret:
        for k, v in bracket_pairing.items():
          if char == k:
            bracket_counts[(k, v)] += 1
          elif char == v:
            bracket_counts[(k, v)] -= 1
      # if the statement is potentially balanced, either remove or add a random character.
      if all([count == 0 for count in bracket_counts.values()]):
        # remove a random character
        if len(ret) > 2:
          loc = random.randint(1, len(ret)-1)
          ret = ret[:loc] + ret[loc+1:]
        else:
          ret += random.choice(brackets)
    return (ret, state)

  def main(tests: int=1000, max_length:int=10):
    test_is_balanced(tests, max_length)

  if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-t", "--tests", type=int, default=1000, help="number of tests to run")
    parser.add_argument("-l", "--length", type=int, default=10, help="maximum length of the
    statement")
    args = parser.parse_args()
    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 a $O(1)$ 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 list as well as a function `get(i)` to get the element in the list at index `i`.

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

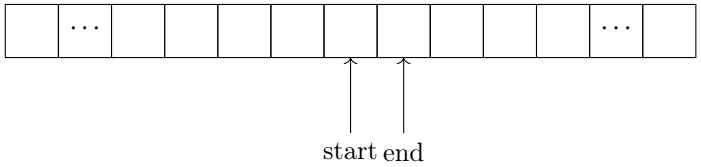
The function `removeItem(int position)` removes an item from the list. The argument position will decide whether the item will be removed from the start or from the end of the list. If the position is 0, the item at the start of the list will be removed. If the position is 1, the item at the end of the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

3.3 User Guide

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 list respectively. The list will be initialised somewhere in the middle of the array with the start index and end index next to each other. The list 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 list).

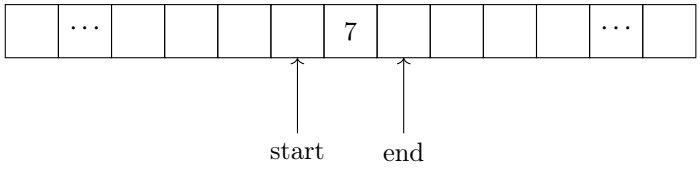
Figure 3.1: Initialising the array



To add an item at the start of the list, 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 list, 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 list to insert the items, the time complexity for inserting items is $O(1)$.

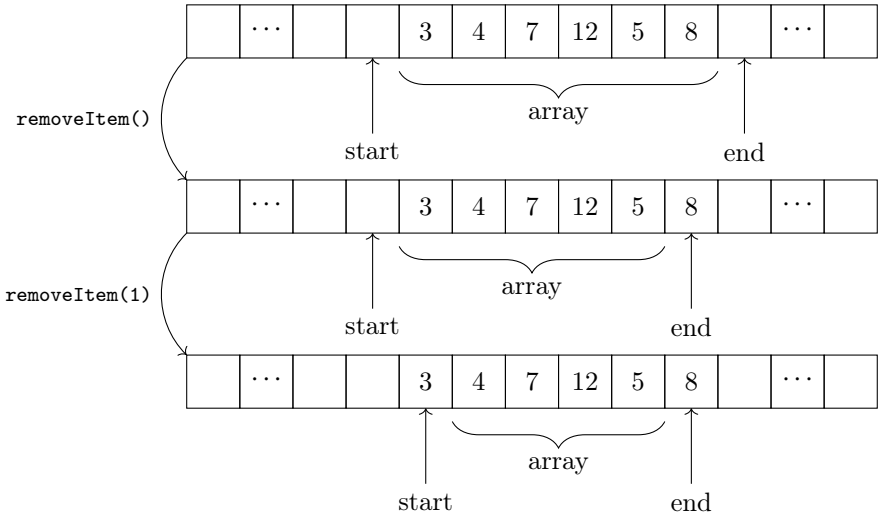
Figure 3.2: Insertion



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 list will print "Overflow!"

To remove an item from the start of the list, the start index will simply shift right (i.e. `start++;`), and to remove an item from the end of the list, 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 list, the time complexity for removing items is $O(1)$.

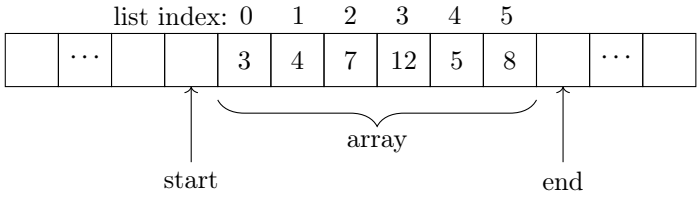
Figure 3.3: Removal of nodes



The list 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 list is empty, it will print "List is empty!".

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

Figure 3.4: List access



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 list. 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

3.7 Listings

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. 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 file.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 such that after it recieves an input from the user, it will remove all punctuation and whitespaces in the string, then change all uppercase characters to lowercase characters. The algorithm will then check the first and last characters in the string to see if they match. If they do, the function is recursively called on the manipulated string without the first and last characters. The time complexity of this algorithm is $O(n)$.

4.5 Limitations

The algorithm expects only the ASCII character set as input, and may not work on sentences including UTF-8 characters beyond the ASCII character set.

4.6 Testing

4.7 Listings

Question 5

5.1 Problem Statement

5.2 Requirements/Specification

5.3 User Guide

5.4 Structure/Design

5.5 Limitations

5.6 Testing

5.7 Listings