A black text on a white background

Description automatically generated

**Data Structures & Algorithms (DSA)**

Year 2 (2024/25), Semester 4

## SCHOOL OF INFOCOMM TECHNOLOGY

Diploma in Information Technology

Diploma in Cybersecurity & Digital Forensics

**COMMON TEST REVISION – SOLUTION DOCUMENT**

INSTRUCTIONS TO CANDIDATES:

1. Write your Student Number, Name and Module Group CLEARLY in the boxes provided below.
2. Provide your answers to the questions in the Common Test paper in this document.
3. Save this file as "DSA Oct 2024 CT – XXX Solution.docx" where XXX is your name.
4. Submit this solution file to BrightSpace.

|  |  |
| --- | --- |
| **Student Number: S10258457** | **Seat Number:** |
| **Student Name: Yeo Jin Rong** | **Module Group: p03** |

**GRADE**

There are 4 questions. Answer ALL questions (100 marks).

Write your solutions to the questions in the space allocated for each question.

Question 1 – Solution (25 marks)

|  |  |
| --- | --- |
| (a) | void List::sortedInsert(ItemType item) {      Node\* newNode = new Node{item, nullptr}; // Create a new node      if (!firstNode || firstNode->item >= item) {          // Insert at the beginning if the list is empty or the item is smaller than the first node          newNode->next = firstNode;          firstNode = newNode;      } else {          // Find the correct position for insertion          Node\* current = firstNode;          while (current->next && current->next->item < item) {              current = current->next;          }          // Insert the new node          newNode->next = current->next;          current->next = newNode;      }      size++; // Increment the size of the list  } |
|  | (10 marks) |
| (b) | void List::sortedInsertR(ItemType item) {      sortedInsertHelper(firstNode, item);      size++; // Increment the size of the list  }  void List::sortedInsertHelper(Node\* current, ItemType item) {      if (!current || current->item >= item) {          // Insert at the current position if end is reached or item is smaller          Node\* newNode = new Node{item, current};          current = newNode;      } else {          // Recursive call for the next node          sortedInsertHelper(current->next, item);      }  } |
|  | (10 marks) |
| (c) | Time complexity for both functions are O(n), as you have to traverse through the list to the desired node, worst case scenario will be O(n) where the desired node is at the end of the list.  Space complexity is O(n) for recursive, O(1) for iterative, this is due to every time the recursive function is called, it is another frame on the stack. |
|  | (5 marks) |

Question 2 – Solution (25 marks)

|  |  |
| --- | --- |
| (a) | void compose(Stack& s) {      cout << "Option 1: Compose" << endl;      cout << "Enter your text (end with Enter): ";      string input;      cin.ignore(); // Clear input buffer      getline(cin, input); // Get user input as a string      for (char c : input) {          if (!s.push(c)) {              cout << "Stack overflow. Could not add: " << c << endl;              break;          }      }      cout << "Text added: " << input << endl;  } |
|  | (10 marks) |
| (b) | void undo(Stack& s) {      cout << "Option 2: Undo" << endl;      if (s.isEmpty()) {          cout << "No text to undo." << endl;          return;      }      char undoneChar;      do {          s.getTop(undoneChar); // Get the character at the top of the stack          cout << "Undoing: " << undoneChar << endl;          s.pop(); // Remove the character from the stack          cout << "Enter 1 to undo again, 0 to quit: ";          int choice;          cin >> choice;          if (choice == 0) break;      } while (!s.isEmpty());      if (s.isEmpty()) {          cout << "The text is now empty." << endl;      } else {          cout << "Remaining text: ";          s.printInOrderOfInsertion(); // Display the remaining text          cout << endl;      }  } |
|  | (10 marks) |
| (c) | void clear(Stack& s) {      cout << "Option 3: Clear" << endl;      while (!s.isEmpty()) {          s.pop(); // Continuously pop until the stack is empty      }      cout << "The text is cleared." << endl;  } |
|  | (5 marks) |

Question 3 – Solution (25 marks)

|  |  |
| --- | --- |
| (a) | void registerCustomer(Queue& serviceQueue, int& queueNumber) {      string customerName;      cout << "Enter customer's name: ";      cin.ignore(); // Clear input buffer      getline(cin, customerName); // Get the customer's name      // Create a new customer object      Customer newCustomer(queueNumber, customerName);      serviceQueue.enqueue(newCustomer); // Add the customer to the queue      cout << "Customer registered successfully: " << customerName           << " (Queue Number: " << queueNumber << ")" << endl;      // Increment the queue number for the next customer      queueNumber++;  } |
|  | (10 marks) |
| (b) | void nextCustomer(Queue& serviceQueue) {      if (serviceQueue.isEmpty()) {          cout << "No customers in the queue." << endl;      } else {          // Get the next customer          Customer next;          serviceQueue.dequeue(next); // Remove the customer from the queue and retrieve their data          cout << "Serving next customer:" << endl;          next.print(); // Display customer details      }  } |
|  | ( 5 marks) |
| (c) | void displayCount(Queue& serviceQueue) {      int count = 0;      // Temporarily store the current queue      Queue tempQueue;      // Traverse the queue to count elements      while (!serviceQueue.isEmpty()) {          Customer tempCustomer;          serviceQueue.dequeue(tempCustomer); // Dequeue each customer          tempQueue.enqueue(tempCustomer);   // Add to a temporary queue          count++;      }      // Restore the original queue      while (!tempQueue.isEmpty()) {          Customer tempCustomer;          tempQueue.dequeue(tempCustomer);          serviceQueue.enqueue(tempCustomer);      }      cout << "Total number of customers in the queue: " << count << endl;  } |
|  | (10 marks) |

Question 4 – Solution (25 marks)

|  |  |
| --- | --- |
| (a) | Hash1  Hash1 computes the hash value using all characters in the string, multiplies current hash value by 31 for each character and adds ASCII value of each character  Hash 2 computes the hash value by skipping over characters, meaning the function only processes a subset of characters |
|  | (6 marks) |
| (b)  (i) | Index 0: 10  Index 1: 36  Index 2: 17  Index 3: 24  Index 4: 19 |
|  | (8 marks) |
| (ii) | It is a prime number, therefore it reduces the possibility of collisions, minimizing the chances of keys hashing to same index when using modulo operations.  However, it is a small size, limits what it can hold without excessive collisions or rehashing |
|  | (6 marks) |
| (iii) | Infinite, assuming separate chaining is used for collision resolution. Separate chaining allows each slot in the hash table to contain a linked list, storing multiple entries. There is no size limit on each chain, the only limiting factor is sufficient memory to support the linked lists and nodes. |
|  | (5 marks) |

**– End of Document –**