

**Data Structures & Algorithms (DSA)**

Year 2/3 (2023/24), Semester 3/5

## SCHOOL OF INFOCOMM TECHNOLOGY

Diploma in Cybersecurity & Digital Forensics

Diploma in Information Technology

**COMMON TEST**

Date: 5 Jun 2023

Time: 1:30pm to 3:00pm

Weightage: 20%

INSTRUCTIONS TO CANDIDATES:

1. This paper consists of 7 pages including this cover page. Check carefully to make sure your set is complete.
2. There are **5** questions in this paper. Answer ALL questions.
3. Write your answers in the SOLUTION document provided online.
4. Computer laptops/notebooks (with no internet access) are ALLOWED.

There are 5 questions. Answer ALL questions (100 marks).

**Question 1 (18 marks)**

A List ADT and a program that makes use of the ADT are given in folder Q1.

You are required to write a program to reverse the items in a list.

(a) Implement the reverse() and main() functions in the program:

1. The reverse() function reverses the items in a list.

(7 marks)

void reverse(List& charList)

{

    // Implement this function to reverse the items in the list

    // Use list operations to reverse the items in the list

    int length = charList.getLength();

    for (int i = 0; i < length / 2; i++)

    {

        ItemType temp = charList.get(i);

        charList.replace(i, charList.get(length - i - 1));

        charList.replace(length - i - 1, temp);

    }

    // Time Complexity: O(n)

}

1. The main() function adds the characters of a string m into charList, calls the reverse() function and displays the reversed list.

(8 marks)

int main()

{

    List charList;

    string m;

    cout << "Enter a string: ";

    cin >> m;

    for (char c: m)

    {

        charList.add(c);

    }\

    cout << "Original List: ";

    charList.display();

    // reverse the list

    reverse(charList);

    cout << "Reversed List: ";

    charList.display();

    return 0;

}

(b) State and explain the time complexity of the reverse() function.

(3 marks)

It is O(n/2) for the for loop, O(1) for get and replace  
Time complexity overall is O(n)

**Question 2 (22 marks)**

A List ADT with its specifications is given in folder Q2.

1. The following implementation of the removeFirst() function removes and returns the first item in the linked list.

Explain the error(s) and re-write the function, removing the error(s).

|  |  |
| --- | --- |
| 1 | int List::removeFirst() |
| 2 | { |
| 3 | Node\* p = firstNode; |
| 4 | Node\* ans = p->next; |
| 5 | delete p; |
| 6 | return ans->item; |
| 7 | size--; |
| 8 | } |

(4 marks)

int List::removeFirst()

{

    if (firstNode == nullptr) {

        // If the list is empty, we cannot remove an item

        throw runtime\_error("List is empty. Cannot remove the first item.");

    }

    // Store the item to return

    int removedItem = firstNode->item;

    // Point to the node to be deleted

    Node\* nodeToDelete = firstNode;

    // Update the head pointer to the next node

    firstNode = firstNode->next;

    // Free the memory of the removed node

    delete nodeToDelete;

    // Decrement the size of the list

    size--;

    // Return the removed item

    return removedItem;

}

1. Implement the following function for the List ADT:

void List::duplicateList(List &new\_list)

(12 marks)

// duplicate another copy of list and return the new list in �new\_list�

void List::duplicateList(List& new\_list)

{

    if (firstNode == nullptr) {

        // Handle empty list case

        new\_list.firstNode = nullptr;

        new\_list.size = 0;

        return;

    }

    // Copy the first node

    Node\* temp = firstNode;

    Node\* newNode = new Node;

    newNode->item = temp->item;

    newNode->next = nullptr;

    new\_list.firstNode = newNode;

    // Track the last node of the new list

    Node\* temp2 = new\_list.firstNode;

    // Iterate through the rest of the nodes

    while (temp->next != nullptr) {

        temp = temp->next;

        Node\* newNode = new Node;

        newNode->item = temp->item;

        newNode->next = nullptr;

        temp2->next = newNode; // Link the new node to the new list

        temp2 = temp2->next;   // Advance the pointer in the new list

    }

    // Set the size of the new list

    new\_list.size = size;

}

1. (i) State and explain the time complexity of removing items near to the end of a linear linked list.

It is O(n). Traversal requires O(n), while deletion operation is O(1), so overall time complexity is O(n)

In a singly linked list, you are required to traverse throughout the list to the node right before the intended node to be deleted, since u need to update the current->next pointer to point to NULL, this requires starting from the head node and traversing through the list to the desired position

1. Suggest improvements to the list structure with the aid of a diagram and describe how the improvements would improve the time complexity.

Doubly Linked List and Tail Pointer

(6 marks)

**Question 3 (30 marks)**

A Queue ADT can be implemented using two stacks.

Both the Queue ADT and the Stack ADT are given in folder Q3.

1. Discuss how you would use the two stacks to implement the queue.

Utilise Stack1 for enqueue operations, use Stack2 for dequeue operations, when dequeuing, check if s1 is empty, pop from both s1 and s2

(5 marks)

1. Implement the following Queue functions in the Queue ADT that uses the Stack ADT:
   1. bool Queue::enqueue(ItemType item)

bool Queue::enqueue(const ItemType item)

{

    s1.push(item); // Push item to s1

    return true;   // Always succeeds

}

(8 marks)

* 1. bool Queue::dequeue() which must have time complexity of O(1)

bool Queue::dequeue()

{

    if (s2.isEmpty()) {

        if (s1.isEmpty()) {

            return false; // Queue is empty

        }

        // Transfer elements from s1 to s2 to prepare for dequeue

        while (!s1.isEmpty()) {

            s2.push(s1.getTop());

            s1.pop();

        }

    }

    s2.pop(); // Remove the front of the queue

    return true; // Operation successful

}

(4 marks)

* 1. void Queue::getFront(ItemType& item)

void Queue::getFront(ItemType& item)

{

    if (s2.isEmpty()) {

        if (s1.isEmpty()) {

            throw runtime\_error("Queue is empty. No front element available.");

        }

        // Transfer elements from s1 to s2

        while (!s1.isEmpty()) {

            s2.push(s1.getTop());

            s1.pop();

        }

    }

    item = s2.getTop(); // Retrieve the front element

}

(4 marks)

* 1. void Queue::isEmpty() which must have time complexity of O(1)

bool Queue::isEmpty()

{

  return s1.isEmpty() && s2.isEmpty();

}

(4 marks)

1. State and explain the time complexity of the enqueue operation.

O(1), push operation is simply creating a new node and adding it to the ADT

(5 marks)

**Question 4 (10 marks)**

A Queue ADT is implemented using a circular array of size 4.

Assume that the queue is initially empty.

Show the contents of the array and the front and back index values after the following sets of operations:

(a) enqueue(10)

enqueue(20)

enqueue(30)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| index: | 0 | 1 | 2 | 3 |
| array |  |  |  |  |

front index: 0 back index: -1

front index: 0 back index: 0

front index: 0 back index: 1

front index: 0 back index: 2

(b) dequeue()

dequeue()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| index: | 0 | 1 | 2 | 3 |
| array |  |  |  |  |

front index:0 back index: 2

front index: 1 back index: 2

front index: 2 back index: 2

(c) enqueue(40)

enqueue(50)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| index: | 0 | 1 | 2 | 3 |
| array |  |  |  |  |

front index:2 back index: 2

front index:2 back index: 3

front index:2 back index: 0

Write your answers in the solution document.

(10 marks)

**Question 5 (20 marks)**

The hash table implementation of the Dictionary ADT given in folder Q5.

The Dictionary ADT is used to store key-value pairs where the key is the mark and the value is the student name.

1. Explain what the following functions in the Dictionary ADT are doing:

* 1. Constructor

The **constructor** of the Dictionary ADT is responsible for initializing the hash table and setting up the initial state of the dictionary.

* 1. Destructor

The **destructor** of the Dictionary ADT is responsible for cleaning up resources when the dictionary object is destroyed.

(4 marks)

1. Implement the following hash table functions in the Dictionary ADT:
2. int Dictionary::hashFunction(int key)

This function takes a key which represents a mark and returns a hash value that represents the grade as shown below:

|  |  |  |
| --- | --- | --- |
| Key | Hash value | Grade Represented |
| 80<=mark<100 | 0 | A |
| 70<=mark<80 | 1 | B |
| 60<=mark<70 | 2 | C |
| 50<=mark<60 | 3 | D |
| mark<50 | 4 | F |

// returns the hash value of a given key.

int Dictionary::hashFunction(int key)

{

    if (key >= 80 && key < 100) {

        return 0; // Grade A

    } else if (key >= 70 && key < 80) {

        return 1; // Grade B

    } else if (key >= 60 && key < 70) {

        return 2; // Grade C

    } else if (key >= 50 && key < 60) {

        return 3; // Grade D

    } else if (key < 50) {

        return 4; // Grade F

    }

    // Handle invalid marks (e.g., < 0 or >= 100)

    throw invalid\_argument("Invalid mark. Key must be within 0 to 100.");

}

(4 marks)

1. void Dictionary::insert(int key, string value)

This function inserts a new key-value pair to the table.

Collisions are handled using separate chaining.

// inserts a new key-value pair to the table array.

void Dictionary::insert(int key, string value)

{

    // Compute hash value

    int hashValue = hashFunction(key);

    // Create new node to store key-value pair

    KeyValuePair\* newKVP = new KeyValuePair;

    newKVP->key = key;

    newKVP->value = value;

    newKVP->next = nullptr;

    // If the bucket is empty, insert the new node

    if (table[hashValue] == nullptr)

    {

        table[hashValue] = newKVP;

    } else {

        // Collision handling

        KeyValuePair\* current = table[hashValue];

        while (current->next != nullptr)

        {

            if (current->key == key)

            {

                // Update value if key already exists

                current->value = value;

                delete newKVP;

                return;

            }

            current = current->next;

        }

        // Check last node for an existing key

        if (current->key == key)

        {

            current->value = value;

            delete newKVP;

        } else {

            current->next = newKVP;

        }

    }

}

(4 marks)

1. Draw a diagram to show the contents of the hash table after the following key-value pairs are inserted in the exact order (1) to (4), using the hash function implemented in Q5(b)(i).
2. 80, "Jeremy"
3. 65, "Thomas"
4. 47, "Steven"
5. 93, "Mary"

I am lazy to draw

Index 0: (80, "Jeremy") → (93, "Mary") → nullptr

Index 1: nullptr

Index 2: (65, "Thomas") → nullptr

Index 3: nullptr

Index 4: (47, "Steven") → nullptr (4 marks)

1. Complete the main() function by writing codes to:
2. Create a Dictionary object.
3. Insert the key-value pairs given in Q5(d) into the Dictionary object.
4. Print the hash value of mark 52.

int main()

{

    Dictionary dict;

    cout << "Inserting key-value pairs...\n";

    dict.insert(80, "Jeremy");

    dict.insert(65, "Thomas");

    dict.insert(47, "Steven");

    dict.insert(93, "Mary");

    // Print hash value of mark 52

    cout << "Hash value of mark 52: " << dict.hashFunction(52) << "\n";

}

(4 marks)

**– End of Paper –**