Name: Gábor Major

Student Number: C00271548

Computing Year 3: 2023-2024

Advanced DSA - Take Home Assignment

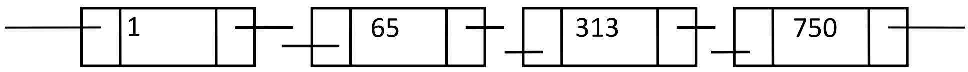
INSTRUCTIONS

- Answer all of the following questions and submit on a **single pdf document** by the due date. This submission document must have your student number in its name. The front page of this document must contain your name and your student number.
- Upload your submission to: HERE
- **Due Date: Sunday November 5th 12 midnight**
- You must demonstrate how you arrive at any answers you get. Correct answers without any explanations/demonstrations will get no credit.
- This assignment has an 50% credit weighting for the Advanced DSA module
- Submissions after November 5th will not be considered and a score of zero will be allocated.
- **You must do this assignment yourself**. Any group of assignment answers that I deem to be too similar for coincidence will be sent to the External Examiner for arbitration. If it is adjudged that plagiarism has occurred then **all parties involved will get a score of zero** on this assignment.
- **All work presented must be your own work.** You may consult class notes, books, online resources for this assignment. If you consult online resources, you are required to submit the **Acknowledge, Describe, Evidence form** along with your submission.

Question 1: (20%)

Write the pseudocode for a non-recursive algorithm to add two numbers represented as two doubly linked lists together and store the result in another doubly linked list.
Represent each number as a doubly linked list with each 3 digits stored as data of a node. (radix 1000)

Example- 1,065,313,750 would be represented as 4 nodes as below.



Demonstrate how your algorithm works on your chosen numbers.

Answer:

The algorithm takes in two linked lists assuming that the first and last nodes point to null for their previous and next respectively. It is also assumed that their is a head and tail pointer for the two lists.
Starting off a new empty list is created, and two pointers are set to the last nodes of the two input linked lists. These two loop through from back to front and add the values of the nodes to a new node that is created each iteration.
This new node is then put into the result linked list that was created.
There is also a rollover variable to make sure each node only has a maximum of 999 for it's value.
The while loop stops when both input linked lists have been traversed, and the result liked list is returned.

number_one = [null, 1, →], [←, 590, →], [←, 920, null]

number_two = [null, 30, →], [←, 400, null]

#	number_one_current.value	number_two_current.value	current_number.value	rollover
1	920	400	1320 → 320	1
2	590	30	621	0
3	1	null	1	0

result = [null, 1, →], [←, 621, →], [←, 320, null]

Pseudocode:

```
def add_numbers(linked_list number_one, linked_list number_two)
{
   linked_list result = null
   number_one_current = number_one.tail
   number_two_current = number_two.tail
   int rollover = 0
```

```
    while (number_one_current != null && number_two_current != null)
    {
        Node current_number = new Node(null, rollover, null)
        rollover = 0
        if (number_one_current != null)
        {
            current_number.value += number_one_current.value
        }
        if (number_two_current != null)
        {
            current_number.value += number_two_current.value
        }
        if (current_number > 1000)
        {
            rollover = current_number // 1000
            current_number %= 1000
        }

        current_number.next = result.head
        result.head = current_number
        current_number.next.previous = current_number
        if (result.tail == null)
        {
            result.tail = current_number
        }
        number_one_current = number_one_current.previous
        number_two_current = number_two_current.previous
    }
    return result
}
```

Question 2: (40%)

(a) Create a Binary Search Tree (BST) with the firstnames of ten of your friends. List the order of entry of the data to the BST.

Answer:

List input: Frank, Olivia, Bob, Dave, Wendy, Ted, Alice, Charlie, Judy, Michael

Tree:

```
                         [Frank]
          [Bob]                              [Olivia]
   [Alice]        [Dave]             [Judy]             [Wendy]
             [Charlie]                  [Michael]     [Ted]
```

(b) Write the algorithm to output this list (from the BST) in alphabetical order.

Demonstrate how your algorithm works on your BST.

Answer:

The algorithm takes in the root node of the whole tree. Each node of the tree is made up of a left child pointer, it's value, and a right child pointer.
The algorithm calls itself on the left child of the root node, then prints out the root's value and finally calls itself again for the right child of the root node, that was passed in.

#	Call trace	Current root	Printed out value
1	Frank	Frank	
2	Frank → Bob	Bob	
3	Frank → Bob → Alice	Alice	Alice
4	Frank → Bob	Bob	Bob
5	Frank → Bob → Dave	Dave	
6	Frank → Bob → Dave → Charlie	Charlie	Charlie
7	Frank → Bob → Dave	Dave	Dave
8	Frank → Bob	Bob	
9	Frank	Frank	Frank
10	Frank → Olivia	Olivia	
11	Frank → Olivia → Judy	Judy	Judy
12	Frank → Olivia → Judy → Michael	Michael	Michael

13	Frank → Olivia → Judy	Judy	
14	Frank → Olivia	Olivia	Olivia
15	Frank → Olivia → Wendy	Wendy	
16	Frank → Olivia → Wendy → Ted	Ted	Ted
17	Frank → Olivia → Wendy	Wendy	Wendy
18	Frank → Olivia	Olivia	
19	Frank	Frank	

Output: Alice, Bob, Charlie, Dave Frank, Judy, Michael, Olivia, Ted, Wendy

```
Pseudocode:
def list_alphabetical(Node root)
{
    if (root.left != null)
    {
        list_alphabetical(root.left)
    }
    print(root.value)
    if (root.right != null)
    {
        list_alphabetical(root.right)
    }
    return
}
```

(c) Write the algorithm that returns the number of names in the tree.

Demonstrate how your algorithm works on your BST.

Answer:

The algorithm takes in the same as the previous one.
It checks if the input node is null, then it returns 0, else it calls itself on the left and right child of the root node and adds 1 to the sum of these outputs.

#	Call trace	Current root	Count returned
1	Frank	Frank	0
2	Frank → Bob	Bob	0
3	Frank → Bob → Alice	Alice	1

4	Frank → Bob	Bob	1
5	Frank → Bob → Dave	Dave	1
6	Frank → Bob → Dave → Charlie	Charlie	2
7	Frank → Bob → Dave	Dave	3
8	Frank → Bob	Bob	4
9	Frank	Frank	4
10	Frank → Olivia	Olivia	4
11	Frank → Olivia → Judy	Judy	4
12	Frank → Olivia → Judy → Michael	Michael	5
13	Frank → Olivia → Judy	Judy	6
14	Frank → Olivia	Olivia	6
15	Frank → Olivia → Wendy	Wendy	6
16	Frank → Olivia → Wendy → Ted	Ted	7
17	Frank → Olivia → Wendy	Wendy	8
18	Frank → Olivia	Olivia	9
19	Frank	Frank	10

Total count: 10

```
Pseudocode:
def count_nodes(Node root)
{
    if (root == null)
    {
        return 0
    }
    else
    {
        return (count_node(root.left) + 1 +
count_nodes(root.right))
    }
}
```

(d) Write the algorithm that output the contents of nodes which have a value greater than a given letter value.

Demonstrate how your algorithm works on your BST.

Answer:

The algorithm takes in the root node like the previous two but this one also takes in a letter as a character.
It checks if the root is null, then returns.
Else it checks if the root value first letter is greater than the letter input,
if yes it calls itself on the left child of the root and then prints out the root value.
Finally it calls itself on the right child of the root node.

Letter: K

#	Call trace	Current root	Printed out value
1	Frank	Frank	
2	Frank → Olivia	Olivia	
3	Frank → Olivia → Judy	Judy	
4	Frank → Olivia → Judy → Michael	Michael	Michael
5	Frank → Olivia → Judy	Judy	
6	Frank → Olivia	Olivia	Olivia
7	Frank → Olivia → Wendy	Wendy	
8	Frank → Olivia → Wendy → Ted	Ted	Ted
9	Frank → Olivia → Wendy	Wendy	Wendy
10	Frank → Olivia	Olivia	
11	Frank	Frank	

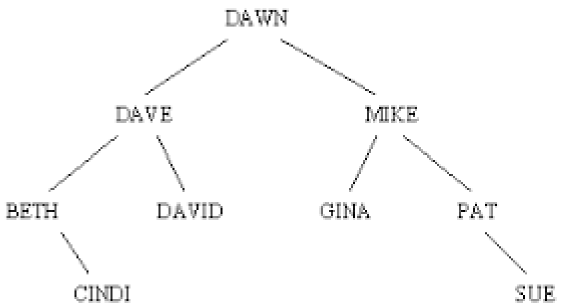
Output: Michael, Olivia, Ted, Wendy

```
Pseudocode:
def output_greater(Node root, char letter)
{
    if (root == null)
    {
        return
```

```
    }
    else
    {
        if (letter < root.value.charat(0))
        {
            output_greater(root.left, letter)
            print(root.value)
        }
        output_greater(root.right, letter)
        return
    }
}
```

(e) Write the algorithm that searches for a given FriendName in the tree. It will return whether the FriendName has been found or not. It will also return the search path used.

For example, given a tree with the following structure:



Searching for DAVID (from root) , would result in left, right, DAVID Found

Searching for AINE (from root) , would result in left, left, left, Not Found

Demonstrate how your algorithm works on your BST.

Answer:

The algorithm takes in the root node, and the name that we are searching for.
It checks if the root node is null, then it prints out that the name was not found.
Otherwise it checks if the root value is the same as the name, and prints out Found.
But if not it checks if it is smaller or bigger and calls itself on the left child or the right child of the root node respectively.

Name: Charlie

#	Call trace	Current root	Printed out value
1	Frank	Frank	left
2	Frank → Bob	Bob	right
3	Frank → Bob → Dave	Dave	left

4	Frank → Bob → Dave → Charlie	Charlie	Charlie Found
5	Frank → Bob → Dave	Dave	
6	Frank → Bob	Bob	
7	Frank	Frank	

Output: left, right, left, Charlie Found

```
Pseudocode:
def name_search(Node root, string name)
{
    if (root == null)
    {
        print("Not Found")
    }
    else
    {
        if (name == root.value)
        {
            print(name + " Found")
        }
        else if (name < root.value)
        {
            print("left, ")
            name_search(root.left, name)
        }
        else if (name > root.value)
        {
            print("right, ")
            name_search(root.right, name)
        }
    }
    return
}
```

(f) List the Big O value for each of your algorithms.

List alphabetical: O(n)
Count nodes: O(n)
Output greater: O(n)
Name search: O(log n)

Question 3: (40%)

Design a Project management application which handles projects and tasks within these projects. Use a linked list(of your choice) to store the project and task details.

Explain the data structure(s) and include a drawing of a node of your structure(s).

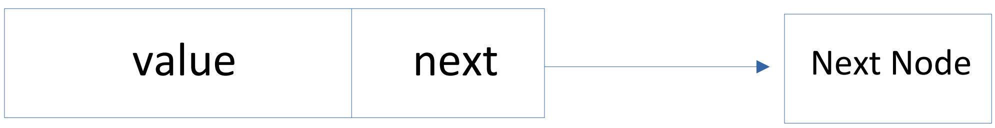
Name and describe eight methods in your app and show how they will work together by writing the pseudocode of each method.

The app should be capable of adding a task to a project, completing a task, list the next task in each project and change the order of tasks within a project (include these as 4 of your 8 methods). Methods like displayAll/viewAll, isEmpty etc.. will not be considered as suitable methods to be counted.

Answer:

Structures:
The project would be the one that holds all of the tasks associated with it. It would have two singly linked lists in it, one for the incomplete tasks and one for the completed tasks. There would be a head and tail pointer for both lists: incomplete_head, incomplete_tail, complete_head, complete_tail
The task would be a node of the linked lists and it's structure would be the following. It would contain a value part, the description of the task, and a next part, a pointer to the next node.

Diagram of node:



Methods for the application.

The add_task method takes in a project, and a description for the new task. It would add on a the newly created task onto the incomplete list in the project.

```
def add_task(project, description)
{
    Task new_task = Task(description, null)
    project.incomplete_tail.next = new_task
    return
}
```

The complete_task takes in a project and a task. It searches through the incomplete list for the task. When found it will remove it from the list and place it into the completed list. A warning message is printed out to the user if the task was not found.

```
def complete_task(project, task)
{
    Task current = project.incomplete_head
    Task previous = null
    while (current != null)
    {
```

```
        if (current == task)
        {
            if (previous == null)
            {
                project.incomplete_head = current.next
            }
            else
            {
                previous.next = current.next
            }
            current.next = null
            project.complete_tail.next = current
            return
        }
        previous = current
        current = current.next
    }
    print("Task not found!")
    return
}
```

The list_next_task takes in just a project. It looks a the incomplete_head and prints out the first task in the linked list it points to.

```
def list_next_task(project)
{
    print(project.incomplete_head.value)
    return
}
```

The change_task_order takes in a project, a task which is already in the incomplete linked list and a new_index for the task to move to. If -1 is passed as the new_index task is put to the back of the linked list.

```
def change_task_order(project, task, new_index)
{
    Task current = project.incomplete_head
    Task previous = null
    while (current != null)
    {
        if (current == task)
        {
            if (previous == null)
            {
                project.incomplete_head = current.next
            }
            else
            {
                previous.next = current.next
            }
            if (new_index == -1)
```

```
            {
                project.incomplete_tail = task
                task.next = null
                return
            }
            current = project.incomplete_head
            previous = null
            int current_index = 0;
            while (current != null)
            {
                if (current_index == new_index)
                {
                    if (previous == null)
                    {
                        project.incomplete_head = task
                        task.next = current
                    }
                    else
                    {
                        previous.next = task
                        task.next = current
                    }
                    return
                }
                previous = current
                current = current.next
                current_index += 1
            }
            project.incomplete_tail = task
            task.next = null
            return
        }
        previous = current
        current = current.next
    }
    print("Task not found!")
    return
}
```

The check_project_status takes in a project. It checks for three possible scenarios, where the complete linked list is empty, then the project hasn't been started yet, if both contain tasks, then it is ongoing, and finally if the incomplete list is empty then it is finished.

```
def complete_project(project)
{
    if (project.complete_head == null)
    {
        print("Project not started")
    }
    else if (project.incomplete_head == null)
    {
```

```
        print(“Project is finished”)
    }
    else
    {
        print(“Project is ongoing”)
    }
    return
}
```

The count_tasks takes in a project. It counts up the total amount of tasks in both the incomplete and complete lists. It also gives a percentage of how many tasks have been completed.

```
def count_tasks(project)
{
    int incomplete_count = 0
    int comlpeted_count = 0
    Task current = project.incomplete_head
    while (current != null)
    {
        incomplete_count += 1
        current = current.next
    }
    current = project.complete_head
    while (current != null)
    {
        complete_count += 1
        current = current.next
    }
    int total_count = incomplete_count + complete_count
    print(“Total amount of tasks: “ + total_count)
    print(“Ratio of completed tasks: “ + (100 * complete_count //
total_count) + “%”)
    print(“Completed tasks: “ + complete_count)
    print(“Incompleted tasks: “ + incomplete_count)
    return
}
```

The delete_task takes in a project and the task to be deleted. It checks both the incomplete and complete linked lists and deletes the task from either list. If the task was not found in either of the lists it prints out a message to the user stating so.

```
def delete_task(project, task)
{
    bool found = False
    Task current = project.incomplete_head
    Task previous = null
    while (current != null)
    {
        if (current == task)
        {
            if (previous == null)
```

```
            {
                project.incomplete_head = current.next
            }
            else
            {
                previous.next = current.next
            }
            found = True
            break
        }
        previous = current
        current = current.next
    }
    current = project.complete_head
    previous = null
    while (current != null)
    {
        if (current == task)
        {
            if (previous == null)
            {
                project.complete_head = current.next
            }
            else
            {
                previous.next = current.next
            }
            found = True
            break
        }
        previous = current
        current = current.next
    }
    if (!found)
    {
        print("Task not found in either list!")
    }
    return
}
```

The reopen_task takes in a project and the task that needs to be reopened. It checks whether the task is in the completed list, and moves it back into the incomplete list. If the task is however not found it prints out a message to the user.

```
def reopen_task(project, task)
{
    Task current = project.complete_head
    Task previous = null
    while (current != null)
    {
        if (current == task)
```

```
        {
            if (previous == null)
            {
                project.complete_head = current.next
            }
            else
            {
                previous.next = current.next
            }
            current.next = null
            project.incomplete_tail.next = current
            return
        }
        previous = current
        current = current.next
    }
    print(“Task not found!”)
    return
}
```