DADSA COURSEWORK B DOCUMENTATION

Pseudocode and Design Diagrams

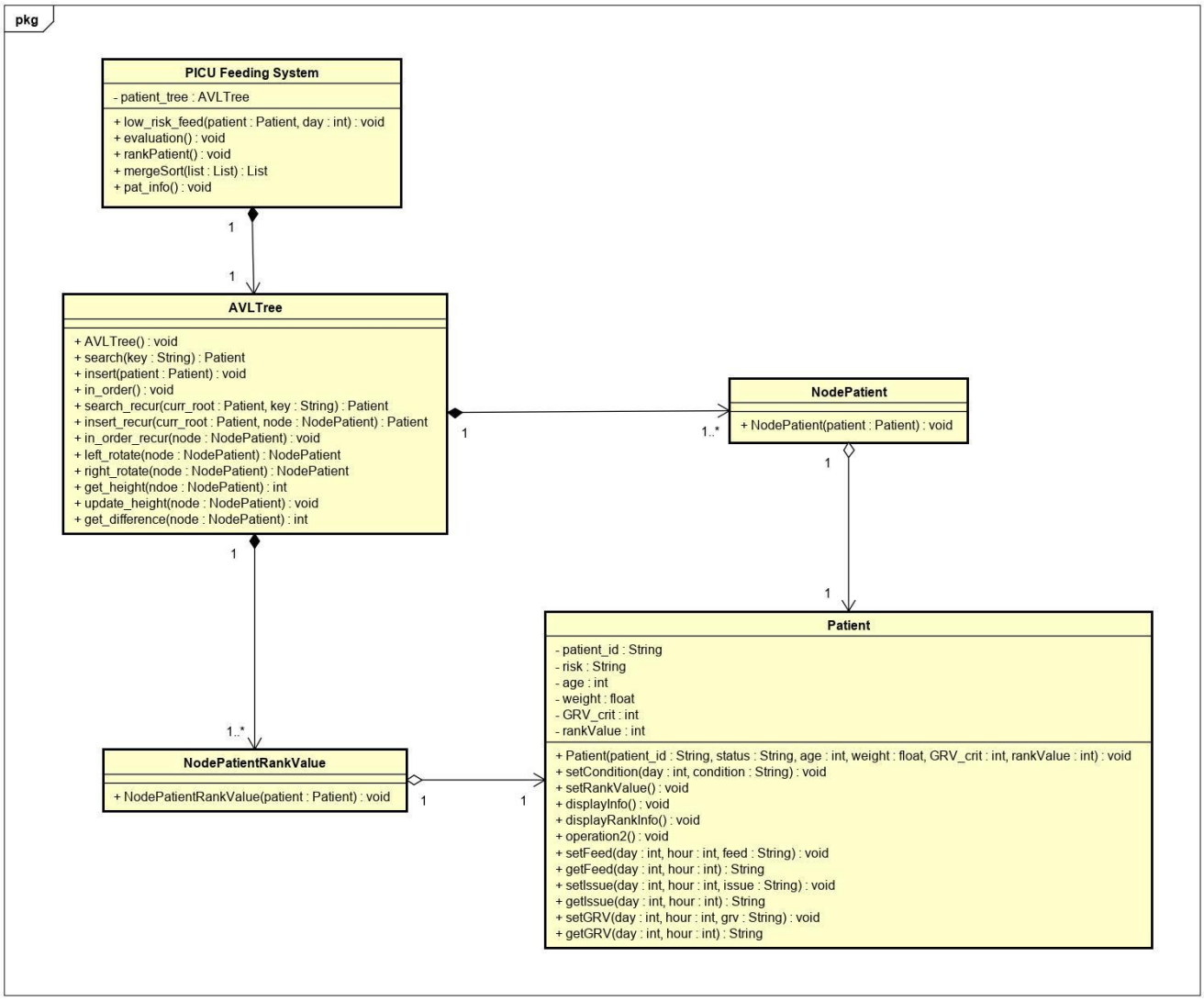
23rd April 2020

Hue Nguyen
Student Id: 19035298
hue2.nguyen@live.uwe.ac.uk

Contents

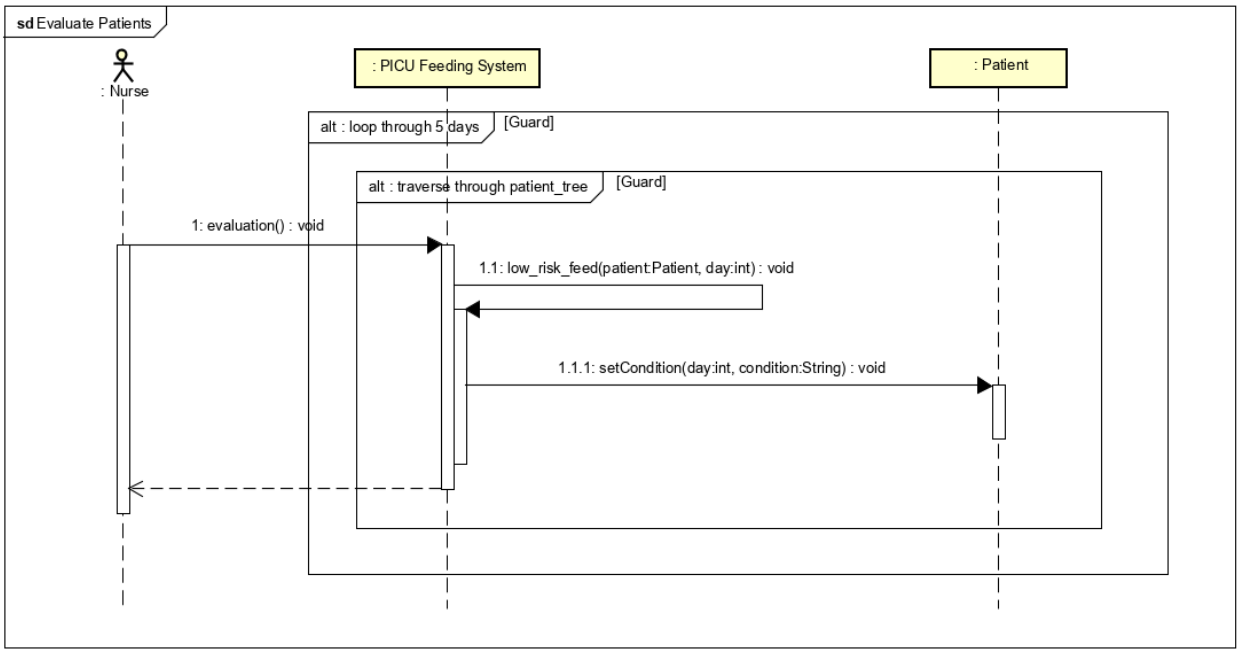
I. Design Diagrams

A. Class Diagram

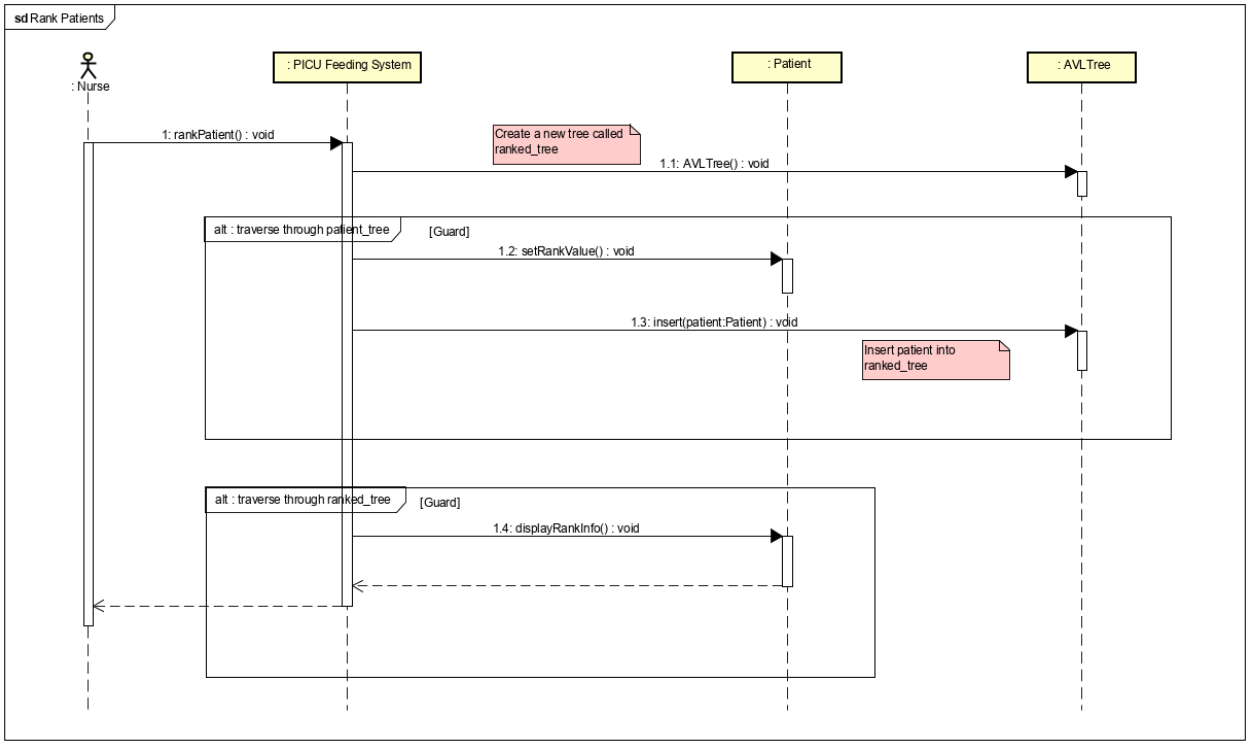


B. Sequence Diagram

1. Evaluate Patients



2. Rank Patients



II. Pseudocode and Code Explanation

A. restrict.py

- the functions in this file will check if the parameters are of the correct type.

```
function getter_setter_gen(attribute, type) {
    function getter() {
        return attribute
    }

    function setter(value)
        if value is not an instance of type then
            raise a TypeError
        else
            set the value to the attribute

    return a class property with getter and setter
}

function check_attribute(class) {
    attribute_dictionary = empty dictionary
    for every key-value pair in the buitin dictionary of the class do
        if value is an instance of type then
            value = call function getter_setter_gen(key, value)
        assign value to key in attribute_dictionary

     return a new class, replacing current class deictionary with the
     modified dictionary
}
```

B. Patient class

```
class Patient {
    attribute patient_id of type String
    attribute risk       of type String
    attribute age        of type int
    attribute weight     of type float
    attribute GRV_crit   of type int
    attribute rankValue  of type int

    constructor (patient_id, status, age, weight, feeding_chart)
        attribute patient_id      = patient_id
        attribute risk            = status
        attribute age             = age
        attribute weight          = weight
        attribute GRV_crit        = weight*5 if weight <= 40, else 250
        attribute feeding_chart   = feeding_chart
        attribute none_count              = 0
        attribute feeding_stopped_count   = 0
        attribute dietitian_count         = 0
        attribute condition_list  = ['N/A', 'N/A', 'N/A', 'N/A', 'N/A']
        attribute continuous_stop = 0
        attribute rankValue       = 0
```

```
    function display_info() {
        display patient's information
    }

    function set_condition(day, condition){
        set condition_list[day] = condition
    }

    function setFeed(day, hour, feed) {
        feeding_chart[day][hour][2] = feed
    }

    function getFeed(day, hour) {
    return feeding_chart[day][hour][2]
    }

    function setGRV(day, hour, grv) {
    feeding_chart[day][hour][3] = feed
    }

    function getGRV(day, hour) {
    return feeding_chart[day][hour][3]
    }

    function setIssue(day, hour, issue) {
    feeding_chart[day][hour][4] = issue
    }

    function getIssue(day, hour) {
    return feeding_chart[day][hour][4]
    }

    function setRankValue(){
        set rankValue = none_count * 10 - feeding_stopped_count -
                        dietitian_count * 3
    }
}
```

C. AVLTree

1. Main Methods

a. Insertion

The insertion process requires the following functions:

- `insert(node_object)`: this function calls the recursive function `insert_recur(root, node)` to find the correct position and balance the tree.
- `insert_recur(current_root, node)`: this function recursively goes through the tree to find the correct position for the `node`, then perform rotations to rebalance the tree.
- `right_rotate(node)`: performs right rotation around the `node`, see details below.
- `left_rotate(node)`: performs left rotation around the `node`, see details below.
- `get_height(node)`: returns the height of the `node` if it exists.
- `get_difference(node)`: returns the height difference between two branches of the `node`.

b. Search

The searching process requires the following functions:

- `search(key, object_type)`: this function calls the recursive function `search_recur(root, key)` to find the node with the passed in `key`.
- `search(current_root, key)`: this function recursively goes through the tree to find the node with the `key`.

2. Implementation

```
class NodePatient{
    constructor(patient){
        attribute patient = patient
        attribute key     = patient's id
        attribute left    = None
        attribute right   = None
        attribute height  = 1
    }
}

class NodePatientRankValue{
    constructor(patient){
        attribute patient = patient
        attribute key     = patient's rankValue
        attribute left    = None
        attribute right   = None
        attribute height  = 1
    }
}

class AVLTree {
    constructor(){
        attribute root = None
    }

    // function to search for patient using their id, this function calls
       the search_recur function to perform recursive search
    function search(key){
        node = call search_recur(self.root, key)

        if node is found then
            return patient
        else
            return a string saying patient is not found
    }

    function search_recur(current_root, key) {
        if current_root is None or key is at root then
            return current_root
        if key > the current_root's id then
            return search_recur(current_root's right node, key)
        if key < the current_root's id then
            return search_recur(current_root's left node, key)
    }
```

```
// function to rotate left around a node
function left_rotate(node) {
    R = node's right child

    assign left branch of R to right of node
    assign node to left of R

    update height of node
    update height of R
    return R
}

// function to rotate right around a node
function right_rotate(node) {
    L = node's left child

    assign right brach of L to left of node
    assign node to right of L

    update height of node
    update height of L
    return L
}

// function to insert an object into tree, it calls recursive function
   insert_recur to find the correct position and rebalance the tree
function insert(node_object){
    root = call insert_recur(root, node_object)
}

function insert_recur(current_root, node) {
    if current_root does not exist then
       return node
    else if node's key > current_root's key then
       current_root's right = insert_recur(current_root's right, node)
    else if node's key < current_root's key then
       current_root's left = insert_recur(current_root's left, node)

    update height of current_root
    difference = get height difference between current's root children

    // Perform rotation to rebalance the tree
    if (current_root's left child exists)
           and (node's key < current_root's left child's key)
           and (height difference > 1) then
       return rotate right around current_root

    if (current_root's right child exists)
           and (node's key > current_root's right child's key)
           and (height difference < -1) then
       return rotate left around current_root

    if (current_root's left child exists)
           and (node's key > current_root's left child's key)
           and (height difference > 1) then
```

```
            current_root's left = rotate left around current_root's left child
            return rotate right around current_root

        if (current_root's right child exists)
                and (node's key < current_root's right child's key)
                and (height difference < -1) then
            current_root's right = rotate right around current_root's right child
            return rotate right around current_root

        return current_root
    }

    // function to traverse through the tree in alphabetical order and print
       patient information, it calls the recursive funtion in_order_recur
    function in_order(){
        call recursive function in_order_recur(root)
    }

    function in_order_recur(node) {
        if node is None then
            stops the recursion

        call in_order_recur(for node's left child)
        display patient's information
        call in_order_recur(for node's right child)
    }

    // function to get height of a node
    function get_height(node) {
        if node exists then
            return node's height
        else
            return 0
    }

    // function to update height of a node
    function update_height(node) {
        node's height = the bigger of two node's children's heights + 1
    }

    // function to get height difference between node's left and right children
    function get_difference(node) {
        if node exists then
            return node's left child's height - node's right child's height
        else
            return 0
    }
}
```

D. database.py

- feeding_chart is a 3-dimensional list, with 1st dimension denoting days, 2nd denoting hours, and 3rd denoting a corresponding row similar to that in the csv files.

```
Patient tree = new empty AVLTree
patient_ids list includes "A1", "A2", "A3", "B1", "B2", "B3", "B4", "B5",
"B6", "B7"
patient_csv list includes "PATIENT DATA - PATIENT " combined with each
element from patient_ids

for every file in patient_csv list do
   open file
     reader        = file reader
     data          = reader as list
     risk_type     = data[0][1]
     age           = take the number in data[0][2] cell
     weight        = take the number in data[0][3] cell
     feeding_chart = new empty list

     for j from 0 to 4 do
         append data of 24 hours of each day into feeding_chart

     patient = create an instance of Patient with patient's id, risk_type,
               age, weight, feeding chart
     insert a NodePatient instance of patient into Patient tree
```

E. PICU_Feeding_Sytem.py

1. Low Risk Feeding

```
function low_risk_feed (patient, day) {
   weight     = patient's weight
   grv_crit   = patient's critical GRV
   from_HR    = True if patient's id starts with 'A', False otherwise

   if it's day 1 or (patient was HR and it's day 4) then
       feed = "5ML/2HRS" if weight <=40, else "20ML/2HRS"
       FEED at hour 0 and 2 = feed
       ISSUE at hour 0 and 2 = "NONE"

   // loop through hours
   for i from 0 to 11 do:
       hr = 2i if patient was not from HR, else 2i+1
       hr_grv = getGRV(day, hr)
       if hr_grv has reading then
           if hr_grv <= grv_cirt then
               feed = "10ML/2HRS" if weight <= 40 else "30ML/2HRS"
               FEED at hr = feed

               if hr + 2 < 24 then
                   FEED at hr + 2 =  feed
               ISSUE at hr i= "NONE"
               patient's continuous feeding_stop counter set to 0
           else
               FEED at hr = "NO FEEDING"
               patient's continuous feeding_stop increases by 1
               if continuous feeding_stop >= 3 then
                   ISSUE at hr = "REFER DIETITIAN"
               else
                   ISSUE at = "FEEDING STOPPED"
```

```
    // decide condition at the end of every day
    for j from 23 to 0 do
        last_issue = ISSUE at hour j
            if last_issue is not blank then
                ISSUE at hour 23 = last_issue
                call set_condition to set condition on day for patient

                if last_issue is "NONE" then
                increment patient's none counter by 1
                else if last_issue is "FEEDING STOPPED" then
                increment patient's feeding_stopped counter by 1
                else if last_issue is "REFER DIETITIAN" then
                    increment patient's dietitian counter by 1
                break
}
```

2. Evaluate Each Patient Through 5 Days

```
funtion evaluation {
    function lr_feed_in_order(node, day) {
        if node does NOT exist then
          stop recursion
        call lr_feed_in_order(node's left child, day)
        call low_risk_feed(node's patient, day)
        call lr_feed_in_order(node's right child, day)
    }
    for day from 0 to 4 do:
        display "DAY" + (day+1)
        call lr_feed_in_order(patient_tree's root, day)
}
```

3. Rank Patients

- Firstly, the patients are sorted in *descending* order of NONE counter, then sorted in *ascending* order of REFER DIETITIAN counter, and lastly sorted in *ascending* order of FEEDING STOPPED counter.
- However, instead of sorting 3 times, using the point system of rankVlaue, we only need to sort the patients once in the *descending* order of rankValue.
- In the rankValue point system, each NONE worth 10 points; for every FEEDING STOPPED, total point is reduced by 1; each REFER DIETITIAN equals 3 FEEDING STOP, so for every REFER DIETITIAN, total point is reduced by 3.
- This ranking algorithm allows patients with higher NONE count to be at the top, with each level of NONE count, patients with higher REFER DIETITIAN count w.ill be at the bottom.

```
function rank_patient {
    ranked_tree = new AVL tree

    // function to recursively set patients' rank points and add patients to
       ranked_tree
    funtion in_order_recur(node) {
        if node does not exist then
            stop recursion
        call function in_order_recur(node's left child)
        call setRankValue()
```

```
        add node's patient to ranked_tree
        call function in_order_recur(node's right child)
    }

    funtion in_order_recur_rank(node) {
        if node does not exist then
            stop recursion
        call function in_order_recur(node's left child)
        call displayRankInfo() for node's patient
        call function in_order_recur(node's right child)
    }

    call in_order_recur_rank(patient_tree's root)
    call in_order_recur(ranked_tree's root)
}
```

4. Display a Patient's Information

```
function pat_info(patient_id) {
    patient = search for patient_id in patient_tree
    if patient is found then
        call displayInfo() for patient
}
```