// ------- Base\_AVL.c -------

struct accountCell{

struct accountCell \*next, \*prev;

char accName[30];

char username[30];

char password[30];

};

struct treeNode{

struct treeNode \*left, \*right, \*prev;

struct accountCell \*head;

char name[30];

unsigned int hashVal;

int level, AVL;

};

struct Session{

treeNode \*hashRoot[10];

treeNode \*root;

char name[30];

};

declare struct Session session as global variable

//node -> for binary search tree

//cell -> for linked list

functions session\_destroy(){

Reset every data on session

}

function session\_change\_root(char ch){

session.root = session.hashRoot[ch % 10];

}

function traverse(struct Tree \*root, unsigned int hashValue){ //For binary tree inside of hash table

if root equals NULL, return NULL

else if hashValue is smaller than root->value, traverse left

else if hashValue is greater than root->value, traverse right

else if hashValue equals root->value, return root

}

function generate\_password(char \*str){

Declare i = 0;

repeat 30 times{

generate random number between 0 and 63

if number is between 0 and 25, str[i] is between 'A' and 'Z'

if number is between 26 and 51, str[i] is between 'a' and 'z'

if number is between 52 and 61, str[i] is between '0' and '9'

if number is 62, str[i] is '#'

if number is 63, str[i] is '$'

}

}

function dataCheck(){

Count how many data on session.hashRoot array is not NULL

return the result

}

//------- AVL Functions -------

function AVL\_L\_rotate(struct treeNode \*node, struct treeNode \*root){

Declare struct treeNode \*parent with value of node->prev

node->prev = parent->prev

parent->prev = node

parent->left = node->right

node->right = parent

Declare struct treeNode \*temp with value of node->prev

if temp equals NULL, root = node

else{

if temp->left equals parent, temp->left = node

else temp->right = node

}

temp = parent->left

if temp doesn't equals NULL, temp->prev = parent

}

function AVL\_R\_rotate(struct treeNode \*node, struct treeNode \*root){

Declare struct treeNode \*parent with value of node->prev

node->prev = parent->prev

parent->prev = node

parent->right = node->left

node->left = parent

Declare struct treeNode \*temp with value of node->prev

if temp equals NULL, root = node

else{

if temp->right equals parent, temp->right = node

else temp->left = node

}

temp = parent->right

if temp doesn't equals NULL, temp->prev = parent

}

declare AVL\_Violation(struct treeNode \*node, int type, struct treeNode \*root)

function AVL\_check(struct treeNode \*node, struct treeNode \*root){

if left sub tree and right sub tree of node doesn't exist{

node->level = 1

node->AVL = 0

}

else{

Declare treeNode \*left with value of node->left

Declare treeNode \*right with value of node->right

if sub left tree of node exists, AVL\_check(left, root)

if sub right tree of node exists, AVL\_check(right, root)

if left sub tree of node doesn't exist{

node->level = right->level + 1

node->AVL = - right->level

}

if right sub tree of node doesn't exist{

node->level = left->level + 1

node->AVL = - left->level

}

else{ // if both sub tree of node exists

if left->level is greater than right->level, node->level = left->level + 1

else node->level = right->level + 1

node->AVL = left->level - right->level

}

if node->AVL is greater than 1, AVL\_violation(node, 0, root)

else if node->AVL is less than -1, AVL\_violation(node, 1, root)

}

}

function AVL\_violation(struct treeNode \*node, int type, struct treeNode \*root){

Declare treeNode \*child

if type equals 0{

child = node->left

if child->AVL is positive{

AVL\_L\_rotate(child, root)

}

else{

AVL\_R\_rotate(child->right, root)

AVL\_L\_rotate(child->prev, root)

}

}

else{

child = node->right

if child->AVL is positive{

AVL\_L\_rotate(child->right, root)

AVL\_R\_rotate(child->prev, root)

}

else{

AVL\_R\_rotate(child, root)

}

}

Check again for AVL violation

}

// ------- CRUD\_Functions.c -------

//------- CRD for tree -------

function createData(struct treeNode \*newNode, struct treeNode \*node, int mode){

if session.root is NULL{

session.root = newNode

newNode->prev = NULL

session.hashRoot[newNode->name[0] % 10] = session.root;

}

else{

treeNode \*node = session.root;

while(1){

ifnewNode->name < node->name lexicographically,{

if node->left is not NULL, traverse left;

else {

node->left = newNode

newNode->prev = node

break

}

}

else if newNode->name > node->name lexicographically, {

if node->right is not NULL, traverse right;

else {

node->right = newNode

newNode->prev = node

break

}

}

else{

print "Error"

break

}

}

Check for AVL violation

}

}

function readData(struct treeNode \*node){ // In Order

if node->left exist, readData(node->left)

print data inside node

if node->right exist, readData(node->right)

}

function freeList(treeNode \*node){

free every cell of linked list with head of node->head

}

function deleteData(struct treeNode \*node){

declare struct treeNode \*parent, \*child

if left sub tree of node don't exist{

if right sub tree of node also don't exist{

if node is root, session.root = NULL

else if node is at parent->left, parent->left = NULL

else parent->right = NULL

freeList(node)

free(node)

}

else{

child = node->right

if node is root, session.root = child

else if node is at parent->left, parent->left = node->right

else parent->right = node->right

child->prev = parent

freeList(node)

free(node)

}

}

else if right sub tree of node don't exist{

child = node->right

if node is root, session.root = child

if node is at parent->left, parent->left = node->left

else parent->right = node->left

child->prev = parent

freeList(node)

free(node)

}

else{ // Both left sub tree and right sub of node tree exist

Find rightmost left sub tree, let it named as rep

replace the values of node with the values of rep

Delete rep recursively

}

Update session.hashRoot value according to this change

check for AVL violation

}

//------- CRUD for data accounts -------

function createDataAccount(struct treeNode \*node, struct accountCell \*newCell){

find the right data at the binary search tree

let node->head known as \*cell

if cell is NULL, node->head = newCell

else{

traverse until cell->next is NULL

cell->next = newCell

newCell->prev = cell

}

}

function readDataAccount(struct treeNode \*node){

print every data inside that node

return node

}

function deleteDataAccount(struct accountCell \*cell){

let cell->prev known as before

let cell->next known as after

before->next = after

after->prev = before

free(cell)

}

// ------- Menus.c -------

//------- Menus -------

function viewExtra(struct treeNode \*node){

loop until user wants to go back{

readDataAccount(node)

print menu to create new account, modify account, delete account, delete this data, and go back

if user choose to create new account{

Declare struct accountCell \*newCell and allocate memory

Fill newCell

generate password

createDataAccount(node, newCell)

notify user that operation is successful

}

if user choose to modify account{

Find the cell which user wants to modify, let it known as cell

Declare struct accountChain modified

fill modified

confirmation from user to change the account data

updateDataAccount(cell, modified)

notify user that operation is successful

}

if user choose to delete account{

Find the cell which user wants to delete, let it known as cell

confirmation from user to delete this data

deleteDataAccount(cell)

}

if user choose to delete this data{

confirmation from user to delete this data

deleteData(node)

}

}

}

function add(){

Loop until user wants to go back{

print menu to add new data, add new account, and go back

if user choose to create new account{

Declare struct accountCell \*newCell and allocate memory

Fill newCell

generate password

createDataAccount(node, newCell)

notify user that operation is successful

return to main menu

}

if user choose to create new data{

Declare char name[30]

scan char

session.root = session.hashRoot[ASCII of char[0] % 10]

if name already exist inside tree root[hashed % 10]{

notify the user that this data exists

viewExtra(node)

}

else{

find the right place to add this new node, let it known as node

Declare struct treeNode \*newNode and allocate memory

newNode->name = name

Declare struct accountCell \*newCell and allocate memory

Fill newCell with data

generate password

createData(newNode, node)

notify user that operation is successful

return to main menu

}

}

if user choos to go back{

return to main menu

}

}

}

function view(){

loop until user wants to go back{

print the menu to view all, view a certain data, and go back

if user choose to view all{

loop x from 0 to 9{

readData(session.root[x])

}

}

if user choose to view a certain data{

declare char str[30]

scan str

session.root = session.hashRoot[ASCII of str[0] % 10]

find the node where the data belongs to, let it known as node

viewExtra(node)

}

if user choos to go back{

return to main menu

}

}

}

function delete(){

loop until user wants to go back{

print the menu to delete one data, delete one account, and go back

if user choose to delete one data{

declare char str[30]

scan str

session.root = session.hashRoot[ASCII of str[0] % 10]

find the node where the data belongs to, let it known as node

Confirmation from user to delete this data

if yes, deleteData(node)

}

if user choose to delete one account{

declare char str[30]

scan str

session.root = session.hashRoot[ASCII of str[0] % 10]

find the node where the data belongs to, let it known as node

print the data inside that node

user choose which data wants to be deleted

Confirmation from user to delete this account

if yes, deleteData(node)

}

if user choos to go back{

return to main menu

}

}

}

function mainMenu(){

loop until user wants to log out{

print the user's name

print the menu to change password, add, view, delete, and log out

if user choose to change password{

change\_password()

}

if user choose to add new data{

add()

}

if user choose to view{

view()

}

if user choose to delete{

delete()

}

if user choose to log off{

return to login menu

}

}

}