|  |
| --- |
| **Telpon Kontak Hashing dan AVL Tree**  LAPORAN PROYEK AKHIR  MATA KULIAH COMP6362004 – DATA STRUCTURES  KELAS BF20      Oleh :  Calvin Farrellino Kurniawan – 2602085883 Jerry Effendi – 2602149533 Valentino Marcell - 2602084104        Semester [~~Ganjil,~~ Genap] B2026    MALANG |
| I. Overview:  Pemilihan topik ini didasarkan pada kebutuhan untuk mengorganisir dan mengelola kontak telepon dalam sebuah program. Struktur data yang dipilih untuk program ini adalah AVL Tree dan Hashing. AVL Tree digunakan untuk menyimpan dan mengelola data kontak telepon dengan efisien, sedangkan Hashing digunakan untuk memetakan data ke dalam tabel hash yang terorganisir.  Metode AVL Tree dipilih karena memiliki kemampuan untuk menyimpan data secara terurut dan memungkinkan operasi pencarian, penambahan, dan penghapusan data dengan kompleksitas waktu yang relatif cepat. Ini sangat berguna dalam program ini karena memungkinkan pengurutan dan pencarian cepat kontak telepon berdasarkan nama atau nomor telepon.  Selain itu, metode Hashing digunakan untuk memetakan data ke dalam tabel hash yang dapat mempercepat akses ke data dengan kompleksitas waktu konstan. Dalam konteks program ini, Hashing digunakan untuk memetakan kontak telepon ke dalam tabel hash berdasarkan kunci yang dihasilkan dari nomor telepon.  II. Literature Review:  Teori Struktur Data Terkait:  AVL Tree: AVL Tree adalah jenis pohon pencarian biner yang seimbang, di mana setiap simpul memiliki faktor keseimbangan yang memenuhi aturan tertentu. Dengan menggunakan rotasi khusus, AVL Tree menjaga tinggi pohon tetap seimbang sehingga operasi pencarian, penambahan, dan penghapusan dapat dilakukan dalam waktu O(log n).  Hashing: Hashing adalah teknik yang digunakan untuk memetakan data ke dalam struktur data hash. Ini melibatkan penggunaan fungsi hash untuk menghasilkan nilai kunci unik dari data yang akan disimpan. Data kemudian ditempatkan di dalam tabel hash berdasarkan nilai kunci ini, memungkinkan akses cepat ke data dengan kompleksitas waktu konstan.  Umum:  Dalam konteks program ini, menggunakan AVL Tree dan Hashing sebagai struktur data sangat relevan dan bermanfaat. Penggunaan AVL Tree memungkinkan penyimpanan dan pengelolaan data kontak telepon dengan efisien, sementara Hashing mempercepat akses ke data dengan memetakan data ke dalam tabel hash yang terorganisir.  III. Program Definition   * Pseudocode and flowchart of the program     #define MAX\_PHONE 22  #define MAX\_NAME 32  #define TABLE\_SIZE 64  typedef struct Node :  int height  string phone  string name  struct Node \*left, \*right  typedef struct Hashmap :  Node \*root  Hashmap  static const Node EmptyStruct = {0}  Hashmap table[TABLE\_SIZE] = {0}  int getKey(Node temp) :  int key = 0  for (int i = 0; temp.phone[i] != '\0'; i++) :  key += (int)temp.phone[i]  return key  int countBits(int number):  int i = 0  while (number > 0):  i++  number >>= 1  return i  int middlesquare(int key) :  long long int square = key \* key  int shift = (countBits(square) - countBits(key)) / 2  int mask = TABLE\_SIZE - 1  square = (square >> shift) & mask  return square  int max(int a, int b) :  return (a > b) ? a : b  int getHeight(Node \*current) :  if (current != NULL) :  return current->height    return 0  int getBalance(Node \*current) :  if (current == NULL) :  return 0    return getHeight(current->left) - getHeight(current->right)  Node \*leftRotation(Node \*current) :  Node \*T1 = current->right  Node \*T2 = T1->left  T1->left = current  current->right = T2  current->height = 1 + max(getHeight(current->left), getHeight(current->right))  T1->height = 1 + max(getHeight(T1->left), getHeight(T1->right))  return T1  Node \*rightRotation(Node \*current) :  Node \*T1 = current->left  Node \*T2 = T1->right  T1->right = current  current->left = T2  current->height = 1 + max(getHeight(current->left), getHeight(current->right))  T1->height = 1 + max(getHeight(T1->left), getHeight(T1->right))  return T1  Node \*createNode(Node temp) :  Node \*new\_node = (Node \*)malloc(sizeof(Node))  new\_node->height = 1  strcpy(new\_node->phone, temp.phone)  strcpy(new\_node->name, temp.name)  new\_node->left = NULL  new\_node->right = NULL  return new\_node  void writeNodeToFile(FILE \*file, Node \*current) :  if (current != NULL) :  writeNodeToFile(file, current->left)  fprintf(file, "%s,%s\n", current->phone, current->name)  writeNodeToFile(file, current->right)  void writeToFile(const char \*filename) :  FILE \*file = fopen(filename, "w")  if (file == NULL) :  printf("Failed to open the file.\n")  return  for (int i = 0; i < TABLE\_SIZE; i++) :  if (table[i].root != NULL) :  writeNodeToFile(file, table[i].root)    fclose(file)  Node \*push(Node \*current, Node temp) :  int cmp = 0  if (current != NULL) :  cmp = strcmp(current->phone, temp.phone    if (current == NULL) :  return createNode(temp)  else if (cmp < 0) :  current->left = push(current->left, temp)  else if (cmp > 0) :  current->right = push(current->right, temp)    else :  return current    current->height = 1 + max(getHeight(current->left), getHeight(current->right))  int balance = getBalance(current)  int cmpl = 0, cmpr = 0  if (current->left != NULL) :  cmpl = strcmp(current->left->phone, temp.phone)    if (current->right != NULL) :  cmpr = strcmp(current->right->phone, temp.phone)    if (balance > 1 && cmpl < 0) :  return rightRotation(current)    if (balance < -1 && cmpr > 0) :  return leftRotation(current)    if (balance > 1 && cmpl > 0) :  current->left = leftRotation(current->left)  return rightRotation(current)    if (balance < -1 && cmpr < 0) :  current->right = rightRotation(current->right)  return leftRotation(current)    return current  void insertData(Node temp) :  int hashKey = middlesquare(getKey(temp))  if (table[hashKey].root == NULL) :  table[hashKey].root = createNode(temp)    else :  table[hashKey].root = push(table[hashKey].root, temp)    void readFromFile(const char \*filename) :  FILE \*file = fopen(filename, "r")  if (file == NULL) :  fclose(file)  file = fopen(filename,"w")  fclose(file)  return    Node temp = {0}  while (fscanf(file, "%[^,],%[^\n]\n", temp.phone, temp.name) == 2) :  insertData(temp)    fclose(file)  Node \*search(Node \*current, Node temp) :  int cmp = 0  if (current != NULL) :  cmp = strcmp(current->phone, temp.phone)    if (current == NULL) :  return NULL    else if (cmp == 0) :  return current    else if (cmp < 0) :  search(current->left, temp)    else if (cmp > 0) :  search(current->right, temp)    Node \*minValueNode(Node \*node) :  Node \*current = node  while (current->left != NULL) :  current = current->left    return current  Node \*pop(Node \*current, Node temp) :  int cmp = 0  if (current != NULL) :  cmp = strcmp(current->phone, temp.phone)    if (current == NULL) :  return current    else if (cmp < 0) :  current->left = pop(current->left, temp)    else if (cmp > 0) :  current->right = pop(current->right, temp)    else :  if (current->right == NULL || current->left == NULL) :  Node \*temp = (current->left) ? current->left : current->right  if (temp == NULL) :  temp = current  current = NULL    else :  \*current = \*temp    free(temp)    else :  Node \*delete = minValueNode(current->right)  Node temps = \*current  \*current = \*delete  current->left = temps.left  current->right = temps.right  current->right = pop(current->right, \*delete)      if (current == NULL) :  return current    current->height = 1 + max(getHeight(current->left), getHeight(current->right))  int balance = getBalance(current)  int cmpl = 0, cmpr = 0  if (current->left != NULL) :  cmpl = strcmp(current->left->phone, temp.phone)    if (current->right != NULL) :  cmpr = strcmp(current->right->phone, temp.phone)    if (balance > 1 && cmpl < 0) :  return rightRotation(current)    if (balance < -1 && cmpr > 0) :  return leftRotation(current)    if (balance > 1 && cmpl > 0) :  current->left = leftRotation(current->left)  return rightRotation(current)    if (balance < -1 && cmpr < 0) :  current->right = rightRotation(current->right)  return leftRotation(current)    return current  void updateData(Node updated, Node old) :  int hashKey = middlesquare(getKey(old))  table[hashKey].root = pop(table[hashKey].root, old)  insertData(updated)  void inorderDisplay(Node \*current) :  if (current != NULL) :  inorderDisplay(current->left)  printf("| %-30s | %-20s |\n", current->name, current->phone)  inorderDisplay(current->right)    void display() :  for (int i = 0; i < TABLE\_SIZE; i++) :  if (table[i].root != NULL) :  inorderDisplay(table[i].root)      int main() :  readFromFile("output.txt")  int menu  Node temp  do :  temp = EmptyStruc  int namelen = 0, phonelen = 0, hashKey = 0  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")  printf("\* Contact List \*\n")  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")  printf("\* 1. Insert \*\n")  printf("\* 2. View All \*\n")  printf("\* 3. Search \*\n")  printf("\* 4. Update \*\n")  printf("\* 5. Delete \*\n")  printf("\* 6. Exit \*\n")  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")  printf(">> ")  scanf("%d", &menu)  getchar()  switch (menu) :  case 1:  do :  printf("Input Name [string|max:30]: ")  fgets(temp.name, MAX\_NAME, stdin)  fflush(stdin)  temp.name[MAX\_NAME - 1] = '\0'  temp.name[strcspn(temp.name, "\n")] = '\0'  namelen = strlen(temp.name)    while (namelen > 30 || namelen < 0)  do :  printf("Input Phone Number [string|max:20]: "  fgets(temp.phone, MAX\_PHONE, stdin)  fflush(stdin)  temp.phone[MAX\_PHONE - 1] = '\0'  temp.phone[strcspn(temp.phone, "\n")] = '\0'  phonelen = strlen(temp.phone)    while (phonelen > 20 || phonelen < 0)  insertData(temp)  printf("Successfully added data!\n")  break  case 2:  printf("---------------------------------------------------------\n")  printf("| %-30s | %-20s |\n", "Name", "Phone Number")  printf("---------------------------------------------------------\n")  display()  printf("---------------------------------------------------------\n")  break  case 3:  do :  printf("Input Phone Number [string|max:20]: ")  fgets(temp.phone, MAX\_PHONE, stdin)  fflush(stdin)  temp.phone[MAX\_PHONE - 1] = '\0'  temp.phone[strcspn(temp.phone, "\n")] = '\0'  phonelen = strlen(temp.phone)  while (phonelen > 20 || phonelen < 0  hashKey = middlesquare(getKey(temp))  if (table[hashKey].root != NULL) :  Node \*src = search(table[hashKey].root, temp)  if (src == NULL) :  printf("Data not in the table")  else :  printf("Name: %s\n", src->name)  printf("Phone Number: %s\n", src->phone)    else :  printf("Data not in the table")    break  case 4:  do :  printf("Input Phone Number [string|max:20]: ")  fgets(temp.phone, MAX\_PHONE, stdin)  fflush(stdin)  temp.phone[MAX\_PHONE - 1] = '\0'  temp.phone[strcspn(temp.phone, "\n")] = '\0'  phonelen = strlen(temp.phone)  while (phonelen > 20 || phonelen < 0)  hashKey = middlesquare(getKey(temp))  if (table[hashKey].root == NULL) :  printf("Phone number not found!\n")  break    Node \*src = search(table[hashKey].root, temp)  if (src == NULL) :  printf("Phone number not found!\n")  break    temp = \*src  int updateMenu  do :  printf("Update: \n")  printf("1. Name [Current: %s]: \n", temp.name)  printf("2. Phone Number [Current: %s]: \n", temp.phone)  printf("3. Save Change(s)\n")  do :  printf(">> ")  scanf("%d", &updateMenu)  getchar()  while (updateMenu > 3 || updateMenu < 0)  if (updateMenu == 1) :  do :  printf("Input Name [string|max:30]: ")  fgets(temp.name, MAX\_NAME, stdin)  fflush(stdin)  temp.name[MAX\_NAME - 1] = '\0'  temp.name[strcspn(temp.name, "\n")] = '\0'  namelen = strlen(temp.name)  while (namelen > 30 || namelen < 0)    else if (updateMenu == 2) :  do :  printf("Input Phone Number [string|max:20]: ")  fgets(temp.phone, MAX\_PHONE, stdin)  fflush(stdin)  temp.phone[MAX\_PHONE - 1] = '\0'  temp.phone[strcspn(temp.phone, "\n")] = '\0'  phonelen = strlen(temp.phone)  while (phonelen > 20 || phonelen < 0)    else :  if (strcmp(temp.name, src->name) == 0 && strcmp(temp.phone, src->phone) == 0) :  printf("No change(s) detected\n"  else :  updateData(temp, \*src)      while (updateMenu != 3)  break  case 5:  do :  printf("Input Phone Number [string|max:20]: ")  fgets(temp.phone, MAX\_PHONE, stdin)  fflush(stdin)  temp.phone[MAX\_PHONE - 1] = '\0'  temp.phone[strcspn(temp.phone, "\n")] = '\0'  phonelen = strlen(temp.phone)    while (phonelen > 20 || phonelen < 0)  hashKey = middlesquare(getKey(temp))  if (table[hashKey].root == NULL) :  printf("Phone number not found!\n")  break    else :  Node \*src = search(table[hashKey].root, temp)  if (src == NULL) :  printf("Phone number not found!\n")  break    table[hashKey].root = pop(table[hashKey].root, temp)    break  case 6:  break  default:  printf("Invalid Input\n")  break      while (menu != 6)  writeToFile("output.txt")  return 0   * Program Overview   The program aims to implement a hashing mechanism to securely store and retrieve contact information. By using a hash the function to this transform contact details into a fixed-size hash value. This process allows for efficient storage and retrieval of contacts while protecting sensitive information.   * Program Description   The program for hashing contact information is designed to securely store and manage contact details while ensuring data integrity and confidentiality. It employs hashing techniques to transform contact information into a fixed-size hash value, allowing for efficient storage, retrieval, and searching of contacts   * Layout design * Program Features   we have such as add data, view data, update data, delete data.    IV. Result  • Program Screenshot (program layout) and its explanation  Main Menu    If press insert  They will ask your name and your phone number so the preview gonna be like this    After we press enter the program will say successfully added data!  If we press view all it gonna open the phone book what we already insert or its gonna open the data from we save before if we already use this before    And the next is menu number 3 is for searching people using number phone so its gonna be like this    The program will ask you the number phone and then the program will search the name of the user phone  Next is update menu in the menu number 4 the program will ask you to put the number    And then the program will ask you. what do you want to change maybe the name or the number phone after we change the name or number phone we need to save change(s)  So if we want change the name to calvin the preview its gonna be like this      And the menu number 5 is for delete  The program will ask you to enter the phone that we want to delete it  After we put the number phone the program will delete it. This is the preview after we done deleting      And the last is menu number 6 is exit its gonna exit your program    • Program Code (is presented in written form, not screenshot form)  #include <math.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_PHONE 22  #define MAX\_NAME 32  #define TABLE\_SIZE 64 // 2^n  typedef struct Node {  int height;  char phone[MAX\_PHONE]; // max 20; primary key  char name[MAX\_NAME]; // max 30  struct Node \*left, \*right;  } Node;  typedef struct Hashmap {  Node \*root;  } Hashmap;  static const Node EmptyStruct = {0};  Hashmap table[TABLE\_SIZE] = {0};  int getKey(Node temp) {  int key = 0;  for (int i = 0; temp.phone[i] != '\0'; i++) {  key += (int)temp.phone[i];  }  return key;  }  int countBits(int number)  {  int i = 0;  while (number > 0)  {  i++;  number >>= 1;  }  return i;  }  int middlesquare(int key) {  // Get the square of the key  long long int square = key \* key;  // Get how many shifts need to be done  int shift = (countBits(square) - countBits(key)) / 2;  // The mask used to only get the number inside the TABLE\_SIZE range  int mask = TABLE\_SIZE - 1;  // Shift the key and mask the key  square = (square >> shift) & mask;  return square;  }  int max(int a, int b) {  return (a > b) ? a : b;  }  int getHeight(Node \*current) {  if (current != NULL) {  return current->height;  }  return 0;  }  int getBalance(Node \*current) {  if (current == NULL) {  return 0;  }  return getHeight(current->left) - getHeight(current->right);  }  // Left rotation  Node \*leftRotation(Node \*current) {  // right child of parrent node  Node \*T1 = current->right;  // Left child of child node  Node \*T2 = T1->left;  // Rotation  T1->left = current;  current->right = T2;  // Update height  current->height = 1 + max(getHeight(current->left), getHeight(current->right));  T1->height = 1 + max(getHeight(T1->left), getHeight(T1->right));  return T1;  }  // Right rotation  Node \*rightRotation(Node \*current) {  // Left child of parrent node  Node \*T1 = current->left;  // Right child of child node  Node \*T2 = T1->right;  // Rotation  T1->right = current;  current->left = T2;  // Update height  current->height = 1 + max(getHeight(current->left), getHeight(current->right));  T1->height = 1 + max(getHeight(T1->left), getHeight(T1->right));  return T1;  }  // Create new node  Node \*createNode(Node temp) {  Node \*new\_node = (Node \*)malloc(sizeof(Node));  new\_node->height = 1;  strcpy(new\_node->phone, temp.phone);  strcpy(new\_node->name, temp.name);  new\_node->left = NULL;  new\_node->right = NULL;  return new\_node;  }  void writeNodeToFile(FILE \*file, Node \*current) {  if (current != NULL) {  writeNodeToFile(file, current->left);  fprintf(file, "%s,%s\n", current->phone, current->name);  writeNodeToFile(file, current->right);  }  }  void writeToFile(const char \*filename) {  FILE \*file = fopen(filename, "w");  if (file == NULL) {  printf("Failed to open the file.\n");  return;  }  for (int i = 0; i < TABLE\_SIZE; i++) {  if (table[i].root != NULL) {  writeNodeToFile(file, table[i].root);  }  }  fclose(file);  }  // Push into AVL Tree  Node \*push(Node \*current, Node temp) {  int cmp = 0;  if (current != NULL) {  cmp = strcmp(current->phone, temp.phone);  }  if (current == NULL) {  return createNode(temp);  } else if (cmp < 0) {  current->left = push(current->left, temp);  } else if (cmp > 0) {  current->right = push(current->right, temp);  }  // if duplicate key exist, return the current node  else {  return current;  }  // Update the current node height  current->height = 1 + max(getHeight(current->left), getHeight(current->right));  int balance = getBalance(current);  // Check if there a violation  int cmpl = 0, cmpr = 0;  if (current->left != NULL) {  cmpl = strcmp(current->left->phone, temp.phone);  }  if (current->right != NULL) {  cmpr = strcmp(current->right->phone, temp.phone);  }  // Left Left case  if (balance > 1 && cmpl < 0) {  return rightRotation(current);  }  // Right Right case  if (balance < -1 && cmpr > 0) {  return leftRotation(current);  }  // Left Right case  if (balance > 1 && cmpl > 0) {  current->left = leftRotation(current->left);  return rightRotation(current);  }  // Right Left case  if (balance < -1 && cmpr < 0) {  current->right = rightRotation(current->right);  return leftRotation(current);  }  // No modification  return current;  }  // Insert data to the table  void insertData(Node temp) {  int hashKey = middlesquare(getKey(temp));  // if the table is empty, create node at the root  if (table[hashKey].root == NULL) {  table[hashKey].root = createNode(temp);  }  // if the table is occupied, use avl to deal with the collision  else {  table[hashKey].root = push(table[hashKey].root, temp);  }  }  void readFromFile(const char \*filename) {  FILE \*file = fopen(filename, "r");  if (file == NULL) {  fclose(file)  file = fopen(filename,"w")  fclose(file)  return;  }  Node temp = {0};  while (fscanf(file, "%[^,],%[^\n]\n", temp.phone, temp.name) == 2) {  insertData(temp);  }  fclose(file);  }  Node \*search(Node \*current, Node temp) {  int cmp = 0;  if (current != NULL) {  cmp = strcmp(current->phone, temp.phone);  }  if (current == NULL) {  return NULL;  } else if (cmp == 0) {  return current;  } else if (cmp < 0) {  search(current->left, temp);  } else if (cmp > 0) {  search(current->right, temp);  }  }  Node \*minValueNode(Node \*node) {  Node \*current = node;  while (current->left != NULL) {  current = current->left;  }  return current;  }  Node \*pop(Node \*current, Node temp) {  int cmp = 0;  if (current != NULL) {  cmp = strcmp(current->phone, temp.phone);  }  if (current == NULL) {  return current;  } else if (cmp < 0) {  current->left = pop(current->left, temp);  } else if (cmp > 0) {  current->right = pop(current->right, temp);  } else {  // check left and right child  if (current->right == NULL || current->left == NULL) {  Node \*temp = (current->left) ? current->left : current->right;  // leaf  if (temp == NULL) {  temp = current;  current = NULL;  }  // 1 child  else {  \*current = \*temp;  }  free(temp);  } else {  Node \*delete = minValueNode(current->right);  Node temps = \*current;  \*current = \*delete;  current->left = temps.left;  current->right = temps.right;  current->right = pop(current->right, \*delete);  }  }  // If the tree had only one node then return  if (current == NULL) {  return current;  }  // Update tree  current->height = 1 + max(getHeight(current->left), getHeight(current->right));  int balance = getBalance(current);  // Check if there a violation  int cmpl = 0, cmpr = 0;  if (current->left != NULL) {  cmpl = strcmp(current->left->phone, temp.phone);  }  if (current->right != NULL) {  cmpr = strcmp(current->right->phone, temp.phone);  }  // Left Left case  if (balance > 1 && cmpl < 0) {  return rightRotation(current);  }  // Right Right case  if (balance < -1 && cmpr > 0) {  return leftRotation(current);  }  // Left Right case  if (balance > 1 && cmpl > 0) {  current->left = leftRotation(current->left);  return rightRotation(current);  }  // Right Left case  if (balance < -1 && cmpr < 0) {  current->right = rightRotation(current->right);  return leftRotation(current);  }  // No modification  return current;  }  void updateData(Node updated, Node old) {  // Remove old data  int hashKey = middlesquare(getKey(old));  table[hashKey].root = pop(table[hashKey].root, old);  // Insert new data  insertData(updated);  }  // print the data inorder  void inorderDisplay(Node \*current) {  if (current != NULL) {  inorderDisplay(current->left);  printf("| %-30s | %-20s |\n", current->name, current->phone);  inorderDisplay(current->right);  }  }  // display the data from the table  void display() {  for (int i = 0; i < TABLE\_SIZE; i++) {  if (table[i].root != NULL) {  inorderDisplay(table[i].root);  }  }  }  int main() {  readFromFile("output.txt");  int menu;  Node temp;  do {  temp = EmptyStruct;  int namelen = 0, phonelen = 0, hashKey = 0;  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\* Contact List \*\n");  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("\* 1. Insert \*\n");  printf("\* 2. View All \*\n");  printf("\* 3. Search \*\n");  printf("\* 4. Update \*\n");  printf("\* 5. Delete \*\n");  printf("\* 6. Exit \*\n");  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf(">> ");  scanf("%d", &menu);  getchar();  switch (menu) {  case 1:  // Check input  do {  printf("Input Name [string|max:30]: ");  fgets(temp.name, MAX\_NAME, stdin);  fflush(stdin);  temp.name[MAX\_NAME - 1] = '\0';  temp.name[strcspn(temp.name, "\n")] = '\0';  namelen = strlen(temp.name);  } while (namelen > 30 || namelen < 0);  do {  printf("Input Phone Number [string|max:20]: ");  fgets(temp.phone, MAX\_PHONE, stdin);  fflush(stdin);  temp.phone[MAX\_PHONE - 1] = '\0';  temp.phone[strcspn(temp.phone, "\n")] = '\0';  phonelen = strlen(temp.phone);  } while (phonelen > 20 || phonelen < 0);  insertData(temp);  printf("Successfully added data!\n");  break;  case 2:  printf("---------------------------------------------------------\n");  printf("| %-30s | %-20s |\n", "Name", "Phone Number");  printf("---------------------------------------------------------\n");  display();  printf("---------------------------------------------------------\n");  break;  case 3:  do {  printf("Input Phone Number [string|max:20]: ");  fgets(temp.phone, MAX\_PHONE, stdin);  fflush(stdin);  temp.phone[MAX\_PHONE - 1] = '\0';  temp.phone[strcspn(temp.phone, "\n")] = '\0';  phonelen = strlen(temp.phone);  } while (phonelen > 20 || phonelen < 0);  hashKey = middlesquare(getKey(temp));  if (table[hashKey].root != NULL) {  Node \*src = search(table[hashKey].root, temp);  if (src == NULL) {  printf("Data not in the table");  } else {  printf("Name: %s\n", src->name);  printf("Phone Number: %s\n", src->phone);  }  } else {  printf("Data not in the table");  }  break;  case 4:  // Phone number to be updated  do {  printf("Input Phone Number [string|max:20]: ");  fgets(temp.phone, MAX\_PHONE, stdin);  fflush(stdin);  temp.phone[MAX\_PHONE - 1] = '\0';  temp.phone[strcspn(temp.phone, "\n")] = '\0';  phonelen = strlen(temp.phone);  } while (phonelen > 20 || phonelen < 0);  hashKey = middlesquare(getKey(temp));  if (table[hashKey].root == NULL) {  printf("Phone number not found!\n");  break;  }  Node \*src = search(table[hashKey].root, temp);  if (src == NULL) {  printf("Phone number not found!\n");  break;  }  // Copy src to temp  temp = \*src;  // What to update  int updateMenu;  do {  printf("Update: \n");  printf("1. Name [Current: %s]: \n", temp.name);  printf("2. Phone Number [Current: %s]: \n", temp.phone);  printf("3. Save Change(s)\n");  do {  printf(">>");  scanf("%d", &updateMenu);  getchar();  } while (updateMenu > 3 || updateMenu < 0);  // Update name  if (updateMenu == 1) {  do {  printf("Input Name [string|max:30]: ");  fgets(temp.name, MAX\_NAME, stdin);  fflush(stdin);  temp.name[MAX\_NAME - 1] = '\0';  temp.name[strcspn(temp.name, "\n")] = '\0';  namelen = strlen(temp.name);  } while (namelen > 30 || namelen < 0);  }  // Update phone number  else if (updateMenu == 2) {  do {  printf("Input Phone Number [string|max:20]: ");  fgets(temp.phone, MAX\_PHONE, stdin);  fflush(stdin);  temp.phone[MAX\_PHONE - 1] = '\0';  temp.phone[strcspn(temp.phone, "\n")] = '\0';  phonelen = strlen(temp.phone);  } while (phonelen > 20 || phonelen < 0);  }  // Save Changes  else {  // Check if anything change  if (strcmp(temp.name, src->name) == 0 && strcmp(temp.phone, src->phone) == 0) {  printf("No change(s) detected\n");  } else {  updateData(temp, \*src);  }  }  } while (updateMenu != 3);  break;  case 5:  do {  printf("Input Phone Number [string|max:20]: ");  fgets(temp.phone, MAX\_PHONE, stdin);  fflush(stdin);  temp.phone[MAX\_PHONE - 1] = '\0';  temp.phone[strcspn(temp.phone, "\n")] = '\0';  phonelen = strlen(temp.phone);  } while (phonelen > 20 || phonelen < 0);  hashKey = middleesquare(getKey(temp));  if (table[hashKey].root == NULL) {  printf("Phone number not found!\n");  break;  } else {  Node \*src = search(table[hashKey].root, temp);  if (src == NULL) {  printf("Phone number not found!\n");  break;  }  table[hashKey].root = pop(table[hashKey].root, temp);  }  break;  case 6:  break;  default:  printf("Invalid Input\n");  break;  }  } while (menu != 6);  writeToFile("output.txt");  return 0;  }  V. References (all source that used in report and program)   * <https://www.geeksforgeeks.org/deletion-in-an-avl-tree/> * https://book.huihoo.com/data-structures-and-algorithms-with-object-oriented-design-patterns-in-java/html/page214.html * "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.   Chapter 12: Binary Search Trees  Chapter 11: Hash Tables |
| **Pengerjaan**   |  |  |  | | --- | --- | --- | | Jerry | Calvin | Marcell | | Menu & File Processing | Insert , Delete & Display | Update & Search |   **Nilai**   |  |  |  |  | | --- | --- | --- | --- | | Nilai | 100 | 100 | 100 | | Nama | Jerry | Calvin | Marcell | |