

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут»

**Лабораторна робота №4**

***з дисципліни «Введення до операційних систем»***

**«Файлові системи»**

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Київ 2024**

*Загальне завдання*

1. Написати програму, що моделює роботу складових заданої файлової системи згідно варіанта (перелік варіантів представлений нижче).

Вхідні дані студент задає самостійно з урахуванням особливостей індивідуального варіанта завдання.

2. Зробити візуалізацію роботи програми і кінцевих результатів на різних

наборах вхідних даних.

*Індивідуальне завдання за варіантом 23(8)*

Побудувати таблиці ідентифікаторів за методом бінарного дерева

Дерево повинно бути ідеально збалансованим і упорядкованим.

Забезпечити можливість додаткового включення та виключення ідентифікаторів при збереженні збалансованості. Забезпечити можливість пошуку заданого ідентифікатора з роздруківкою шляху пошуку та відображенням вигляду дерева.

Додаткові умови – у якості ідентифікаторів використовувати одну букву та дві цифри.

*Код програми:*

==> main.c <==

#include "tree.h"

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#define prnt(x) printf("%s\n", x)

#define check\_until(x, y)                                                      \

  do {                                                                         \

    if (x != '\n') {                                                           \

      if (!y(x))                                                               \

        prnt("Symbol has incorrect type, use alpha for the first symbol, and " \

             "digit for two others");                                          \

    }                                                                          \

    x = getchar();                                                             \

  } while (!y(x))

#define fill\_data\_basic(data, S, N1, N2)                                       \

  (data).S = S;                                                                \

  (data).N1 = N1 - '0';                                                        \

  (data).N2 = N2 - '0'

#define input\_data                                                             \

  prnt("Enter identifier: ");                                                  \

  check\_until(S, isalpha);                                                     \

  check\_until(N1, isdigit);                                                    \

  check\_until(N2, isdigit);\

  prnt("Data filled successfully")

int main() {

  struct BinaryTree \*tree = NULL;

  struct Data data;

  enum Menu { WAIT = 0, OUT = 1, ADD = 2, FIND = 3, DELETE = 4, EXIT = 5 };

  char \*menu = "=== Menu ===\n"

               "1) Output tree\n"

               "2) Add node to tree\n"

               "3) Find node in tree\n"

               "4) Delete node in tree\n"

               "5) Exit\n"

               "Use <m> to display menu";

  char choice = WAIT;

  prnt(menu);

  do {

    choice = getchar();

    choice -= '0';

    char S = 'a';

    char N1 = '0';

    char N2 = '0';

    switch (choice) {

    case OUT:

      output\_tree(tree, 0, true, true, false);

      printf("\n");

      break;

    case ADD:

      input\_data;

      fill\_data\_basic(data, S, N1, N2);

      tree = add(tree, data);

      break;

    case FIND:

      input\_data;

      fill\_data\_basic(data, S, N1, N2);

      struct SearchData \*search = find(tree, data);

      output\_search(search);

      break;

    case DELETE:

      input\_data;

      fill\_data\_basic(data, S, N1, N2);

      tree = rm(tree, data);

      break;

    case WAIT:

    default:

      if (choice + '0' != '\n' && choice + '0' != 'm')

        prnt("Select another menu choice");

      break;

    case EXIT:

      free\_tree(tree);

    };

    if (choice + '0' == 'm')

      prnt(menu);

  } while (choice != EXIT);

  return 0;

}

==> tree.c <==

#include "tree.h"

#include <stdio.h>

#include <stdlib.h>

struct BinaryTree \*create(struct Data data) {

  struct BinaryTree \*node = malloc\_struct(struct BinaryTree \*, 1);

  fill\_data(node->data, data);

  node->node1 = NULL;

  node->node2 = NULL;

  node->height = 1;

}

struct BinaryTree \*add(struct BinaryTree \*node, struct Data data) {

  if (node == NULL) {

    node = create(data);

    return node;

  }

  if (compare(&data, node->data) ==

      LOWER) // When is LOWER = 1 reference /\*CMP\*/

    node->left = add(node->left, data);

  else if (compare(&data, node->data) == BIGGER)

    node->right = add(node->right, data);

  else {

    printf("Error: same identifier already in this directory");

    return node;

  }

  update\_height(node);

  return rebalance(node);

}

void update\_height(struct BinaryTree \*node) {

  if (node->left > node->right)

    node->height = node->left->height + 1;

  else if (node->right != NULL)

    node->height = node->right->height + 1;

}

size\_t get\_height(struct BinaryTree \*node) {

  if (node == NULL)

    return 0;

  return node->height;

}

struct BinaryTree \*rebalance(struct BinaryTree \*node) {

  signed short int delta = calculate\_delta(node);

  if (delta > 1 && compare(node->data, node->left->data) == BIGGER)

    return rotate\_right(node);

  if (delta < -1 && compare(node->data, node->right->data) == LOWER)

    return rotate\_left(node);

  if (delta > 1 && compare(node->data, node->left->data) == LOWER) {

    node->left = rotate\_left(node->left);

    return rotate\_right(node);

  }

  if (delta < -1 && compare(node->data, node->right->data) == BIGGER) {

    node->right = rotate\_right(node->right);

    return rotate\_left(node);

  }

  return node;

}

struct BinaryTree \*rotate\_left(struct BinaryTree \*node) {

  struct BinaryTree \*y = node->right;

  struct BinaryTree \*T2 = y->left;

  y->left = node;

  node->right = T2;

  update\_height(node);

  update\_height(y);

  return y;

}

struct BinaryTree \*rotate\_right(struct BinaryTree \*node) {

  struct BinaryTree \*x = node->left;

  struct BinaryTree \*T2 = node->right;

  x->right = node;

  node->left = T2;

  update\_height(node);

  update\_height(x);

  return x;

}

void free\_tree(struct BinaryTree \*node) {

  if (get\_height(node) == 1) {

    node->height = 0;

    free(node);

    free(node->data);

  } else {

    if (node->node1 != NULL)

      free\_tree(node->node1);

    if (node->node2 != NULL)

      free\_tree(node->node2);

    node->height = 0;

    free(node);

  }

}

struct SearchData \*find(struct BinaryTree \*node, struct Data data) {

  struct Data \*\*path = malloc\_struct(struct Data \*\*, 1);

  struct SearchData \*search = malloc\_struct(struct SearchData \*, 1);

  // if node null no search value

  if (node == NULL) {

    search->values = NULL;

    search->count = 0;

    return search;

  }

  // if node->id == id

  if (compare(&data, node->data) == SAME) {

    fill\_data(path[0], data);

    search->values = path;

    search->count = 1;

    return search;

  }

  // if node->id != id

  fill\_data\_link(path[0], node->data);

  search->count = 1;

  // also add depth ids

  struct SearchData \*under\_path;

  if (compare(&data, node->data)) // When is LOWER = 1 reference /\*CMP\*/

    under\_path = find(node->left, data);

  else

    under\_path = find(node->right, data);

  for (size\_t i = 0; i < under\_path->count; i++) {

    fill\_data\_link(path[search->count], under\_path->values[i]);

    search->count++;

  }

  search->values = path;

  return search;

}

enum Level compare(struct Data \*data1, struct Data \*data2) {

  if (data1->S < data2->S)

    return LOWER;

  else if (data1->S > data2->S)

    return BIGGER;

  else if (data1->N1 < data2->N1)

    return LOWER;

  else if (data1->N1 > data2->N1)

    return BIGGER;

  else if (data1->N2 < data2->N2)

    return LOWER;

  else if (data1->N2 > data2->N2)

    return BIGGER;

  else

    return SAME;

}

struct BinaryTree \*children\_left(struct BinaryTree \*node) {

  if (node == NULL)

    return NULL;

  while (node->left != NULL)

    node = node->left;

  return node;

}

struct BinaryTree \*children\_right(struct BinaryTree \*node) {

  if (node == NULL)

    return NULL;

  while (node->right != NULL)

    node = node->right;

  return node;

}

signed short int calculate\_delta(struct BinaryTree \*node) {

  if (node == NULL)

    return 0;

  return get\_height(node->left) - get\_height(node->right);

}

struct BinaryTree \*rm(struct BinaryTree \*node, struct Data data) {

  if (node == NULL)

    return node;

  enum Level cmp = compare(&data, node->data);

  if (cmp == LOWER)

    node->left = rm(node->left, data);

  else if (cmp == BIGGER)

    node->right = rm(node->right, data);

  else if (cmp == SAME) {

    if (node->left == NULL || node->right == NULL) {

      struct BinaryTree \*temp = node->left ? node->left : node->right;

      if (temp == NULL) {

        temp = node;

        node = NULL;

      } else {

        \*node = \*temp;

      }

      free(temp);

    } else {

      struct BinaryTree \*temp = children\_left(node->right);

      node->data = temp->data;

      struct Data data = {temp->data->S, temp->data->N1, temp->data->N2};

      node->right = rm(node->right, data);

    }

  }

  if (node == NULL)

    return node;

  update\_height(node);

  return rebalance(node);

}

#define indent printf("\t")

#define newl printf("\n")

#define p\_down\_right printf("↳")

#define p\_up\_right printf("↱")

#define p\_right printf("→")

#define p\_up\_down\_right printf("⇰")

#define p\_out printf("↪")

#define out\_node\_data(x)                                                       \

  printf("%c%d%d", (x)->data->S, (x)->data->N1, (x)->data->N2)

void output\_tree(struct BinaryTree \*node, size\_t depth, bool alone,

                 size\_t first, size\_t i\_am\_right) {

  if (node != NULL) {

    newl;

    if (first) {

      printf("Tree:\n");

      first = false;

      p\_down\_right;

      out\_node\_data(node);

    } else {

      for (size\_t i = 0; i < depth; i++) {

        indent;

      }

      if (!i\_am\_right) {

        if (alone) {

          p\_down\_right;

        } else {

          p\_down\_right;

        }

        out\_node\_data(node);

      } else {

        p\_down\_right;

        out\_node\_data(node);

      }

    }

    alone = (node->node1 == NULL || node->node2 == NULL) ? true : false;

    output\_tree(node->node1, depth + 1, alone, false, false);

    output\_tree(node->node2, depth + 1, alone, false, true);

    if (node->node1 == NULL && node->node2 == NULL) {

      first = true;

      depth = 0;

    }

  } else {

    if (first)

      printf("Tree is empty\n");

  }

}

void output\_search(struct SearchData \*data) {

  printf("Search path: \n");

  for (size\_t i = 0; i < data->count; i++) {

    if (i != 0)

      printf("->");

    printf("(%c%d%d)", data->values[i]->S, data->values[i]->N1,

           data->values[i]->N2);

  }

  printf("\n");

}

==> tree.h <==

#include <stddef.h>

#include <stdbool.h>

#ifndef TREE\_H

#define TREE\_H

#define left node1

#define right node2

#define left\_of\_right node2->node1

#define left\_of\_left node1->node1

#define right\_of\_left node1->node2

#define right\_of\_right node2->node2

#define malloc\_struct(x, k) (x) malloc(sizeof(x) \* k)

#define fill\_data(data\_dest, data\_source)        \

    data\_dest = malloc\_struct(struct Data \*, 1); \

    data\_dest->S = data\_source.S;                \

    data\_dest->N1 = data\_source.N1;              \

    data\_dest->N2 = data\_source.N2

#define fill\_data\_link(data\_dest, data\_source)   \

    data\_dest = malloc\_struct(struct Data \*, 1); \

    data\_dest->S = data\_source->S;               \

    data\_dest->N1 = data\_source->N1;             \

    data\_dest->N2 = data\_source->N2

#define swap(x, y)              \

    do                          \

    {                           \

        void \*temp = (void \*)x; \

        x = y;                  \

        y = (typeof(y))temp;    \

    } while (0)

/\*CMP\*/

enum Level

{

    BIGGER,

    LOWER,

    SAME

};

struct Data

{

    char S;

    char N1;

    char N2;

};

struct SearchData

{

    struct Data \*\*values;

    size\_t count;

};

struct BinaryTree

{

    struct BinaryTree \*node1;

    struct BinaryTree \*node2;

    struct Data \*data;

    size\_t height;

};

struct BinaryTree \*create(struct Data data);

void free\_tree(struct BinaryTree \*node);

struct BinaryTree \*add(struct BinaryTree \*node, struct Data data);

struct BinaryTree \*rm(struct BinaryTree \*node, struct Data data);

struct SearchData \*find(struct BinaryTree \*node, struct Data data);

void update\_height(struct BinaryTree \*node);

struct BinaryTree \*rebalance(struct BinaryTree \*node);

struct BinaryTree \*rotate\_right(struct BinaryTree \*node);

struct BinaryTree \*rotate\_left(struct BinaryTree \*node);

size\_t get\_height(struct BinaryTree \*node);

enum Level compare(struct Data \*data1, struct Data \*data2);

struct BinaryTree \*children\_left(struct BinaryTree \*node);

struct BinaryTree \*children\_right(struct BinaryTree \*node);

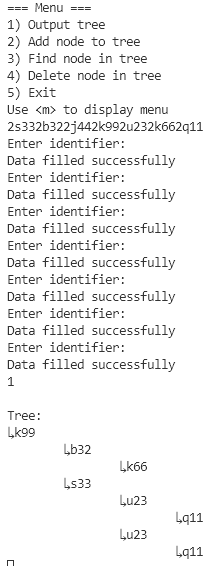
signed short int calculate\_delta(struct BinaryTree \*node);

void output\_tree(struct BinaryTree \*node, size\_t depth, bool alone, size\_t first, size\_t i\_am\_right);

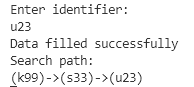
void output\_search(struct SearchData \*data);

#endif

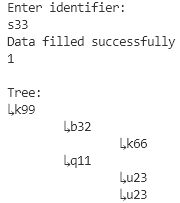
*Скріншоти програми:*

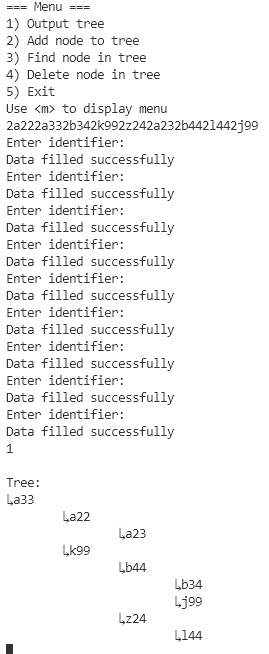
**

*Створення дерева та вивід*

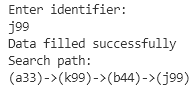
**

*Пошук у дереві*

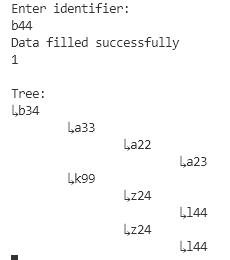
**

*Видалення та вивід з дерева*

*Створення дерева з іншим набором даних*



*Пошук у дереві*



*Видалення та вивід з дерева*

*Висновок:*

Під час виконання лабораторної роботи ознайомились з основними підходами до фізичної організації різноманітних файлових систем, таких як FAT, NTFS, FAT16 та інші, а також з їх реалізацією у файлових системах. Написали програму, що моделює роботу складових файлової системи та зробили візуалізацію роботи програми та кінцевих результатів на різних наборах даних. А саме, побудували таблиці ідентифікаторів за методом бінарного дерева, що було ідеально збалансованим і упорядкованим. Забезпечили можливість додаткового включення та виключення ідентифікаторів при збереженні збалансованості. Забезпечили можливість пошуку ідентифікатора з роздруківкою шляху пошуку та відображенням вигляду дерева. В якості ідентифікатора використовувалась одна буква та дві цифри.

Як виявилося для збереження збалансованості та упорядкованості потрібна була реалізація АВЛ-дерева, що є збалансованим по висоті бінарним деревом пошуку, де для кожної його вершини висота її двох піддерев не відрізняється більше ніжна один.

Під час операцій додавання та видалення ідентифікатора відбувалася операція балансування, яка у разі різниці висот лівого та правого піддерев на два змінює зв’язку предок -нащадок в піддереві даної вершини так, що різниця стає менше або дорівнює одиниці, інакше не робить змін. Результат досягається за рахунок обертання піддерева даної вершини. Розрізняють такі чотири типи обертань: мале ліве обертання, велике ліве обертання, мале праве обертання та велике праве обертання. З метою збереження впорядкованості використовувалася функція порівняння, що й сортувала відповідні вершини, спочатку за першим символом, потім за двома цифрами.