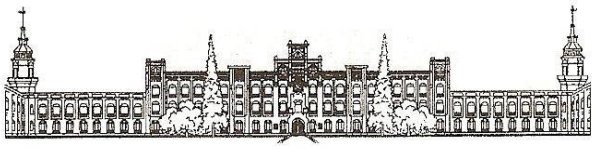

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

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:

```
=== Menu ===
1) Output tree
2) Add node to tree
3) Find node in tree
4) Delete node in tree
5) Exit
Use <m> to display menu
2s332b322j442k992u232k662q11
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
1

Tree:
└k99
        └b32
                └k66
        └s33
                └u23
                        └q11
                └u23
                        └q11
□
```

Створення дерева та вивід

```
Enter identifier:
u23
Data filled successfully
Search path:
(k99)->(s33)->(u23)
```

Пошук у дереві

```
Enter identifier:
s33
Data filled successfully
1

Tree:
└k99
        └b32
                └k66
        └q11
                └u23
                └u23
```

Видалення та вивід з дерева

```
=== Menu ===
1) Output tree
2) Add node to tree
3) Find node in tree
4) Delete node in tree
5) Exit
Use <m> to display menu
2a222a332b342k992z242a232b4421442j99
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
Enter identifier:
Data filled successfully
1

Tree:
└a33
        └a22
                └a23
        └k99
                └b44
                        └b34
                        └j99
                └z24
                        └l44
```

Створення дерева з іншим набором даних

```
Enter identifier:
j99
Data filled successfully
Search path:
(a33)->(k99)->(b44)->(j99)
```

Пошук у дереві

```
Enter identifier:
b44
Data filled successfully
1

Tree:
└b34
        └a33
                └a22
                        └a23
        └k99
                └z24
                        └l44
                └z24
                        └l44
```

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.