Отчет по лабораторной работе N 23 по курсу

"Фундаментальная информатика"

Студент группы: М80-107Б-21, Павлов Иван Дмитриевич
Контакты: pavlov.id.2003@gmail.com
Работа выполнена: 08.04.2022
Преподаватель: Найденов Иван Евгеньевич

1 Тема

Динамические структуры данных. Обработка деревьев.

2 Цель работы

Научиться работать с динамическими структурами данных и обрабатывать деревья.

3 Задание

Составить программу на языке Си для построения и обработки дерева общего вида, содержащего узлы типа *int*. Основные функции работы с деревьями реализовать в виде универсальных процедур или функций. После того, как дерево создано, его обработка должна производиться в режиме текстового меню со следующими действиями:

- *добавление нового узла* (для дерева общего вида должен задаваться *отец* узла);
- *текстовая визуализация дерева* (значение каждого узла выводится в отдельной строке, с отступом, пропорциональным глубине узла, в порядке старшинства узлов);
- *удаление узла* (для дерева общего вида удаляется все поддерево, исходящее из самого узла. Должно быть предусмотрено корректное освобождение памяти);
- Вариант №8: Определить число вершин дерева, степень которых совпадает со значением элемента.

4 Оборудование

Процессор: AMD Ryzen 5 4600H with Radeon Graphics
ОП: 7851 Мб
НМД: 256 Гб
Монитор: 1920x1080

5 Программное обеспечение

Операционная система семейства: linux (ubuntu), версия 20.04.3 LTS
Интерпретатор команд: bash, версия 5.0.17(1)-release.
Система программирования: gcc*, версия 17
Редактор текстов: emacs, версия 25.2.2
Утилиты операционной системы: subl, gcc
Прикладные системы и программы: sublime text, bash
Местонахождение и имена файлов программ и данных на домашнем компьютере: /home/ggame/newlabs/

6 Идея, метод, алгоритм решения задачи

6.1 Структура узла дерева

Листинг 1: struct tree_node

```
1     struct tree_node
2     {
3       int key;
4       struct tree_node *child;
5       struct tree_node *sibling;
6       struct tree_node *parent;
7     };
8     typedef struct tree_node node;
```

Структура узла дерева содержит: значение элемента типа *int* и 3 указателя на структуры узла: сына, брата и отца (используется только для реализации добавления узла);

6.2 Создание узла дерева

Листинг 2: create_tree

```
1     node *create_tree(int value)
2     {
3       node *tree = (node *) malloc(sizeof(node));
4       tree->key = value;
5       tree->child = NULL;
6       tree->sibling = NULL;
7       tree->parent = NULL;
8       return tree;
9     }
```

Данная функция возвращает указатель на узел. Узел создаем с помощью динамического выделения блока памяти размера структуры, значению элемента присваиваем переданный параметр, всем указателям структуры присваиваем значение *NULL*.

6.3 Рекурсивный обход дерева и поиск узла

Листинг 3: create_tree

```
1     node *find_node(node* tree, int value)
2     {
3       if (tree == NULL) {
4         return NULL;
5       }
6       if (tree->key == value) {
7         return tree;
8       }
9       node* cur = NULL;
10      if (tree->child) {
11        cur = find_node(tree->child, value);
12        if (cur != NULL) {
13          return cur;
14        }
15      }
16      if (tree->sibling) {
17        cur = find_node(tree->sibling, value);
18        if (cur != NULL) {
19          return cur;
20        }
21      }
22    }
```

Данная функция возвращает указатель на структуру узла, элемент которого совпадает с введенным параметром. Алгоритм: если дерево пустое, возвращаем *NULL*; если значение элемента узла совпадает с

переданным параметром, возвращаем указатель на этот узел. Далее присваиваем созданной ссылке на узел рекурсивное значение данной функции, вызванной от узла (то же самое делаем и с сиблингами). Так функция в конце концов вернет или нужное значение, или *NULL*.

6.4 Добавление узла

Листинг 4: create_tree

```
1     void add_node(node* tree, int parent, int value)
2     {
3       node *res = find_node(tree, parent);
4       if (res == NULL) {
5         return;
6       }
7       node *res_parent = res;
8       if (res->child == NULL) {
9         res->child = create_tree(value);
10        res->child->parent = res;
11        return;
12      }
13      res = res->child;
14      while (res->sibling != NULL) {
15        res = res->sibling;
16      }
17      res->sibling = create_tree(value);
18      res->sibling->parent = res_parent;
19    }
```

Для создания узла используется следующий алгоритм:

1. создаем ссылку на найденный узел, соответствующий заданному параметру (узел отца);
2. если нет отца - возвращаем пустое значение;
3. создаем ссылку на ссылку на отца;
4. если нет детей, создаем ребенка (необходимый узел с переданным значением элемента), указатель на отца берем равным созданной ссылке;
5. присваиваем ссылке на найденный узел значение ребенка и проходим по ее сиблингам, последнему сиблингу присваиваем указатель на сиблинга на созданный узел (с переданным значением элемента), указатель на отца у которого присваиваем ссылке на ссылку на отца;
6. таким образом мы получили узел, который имеет указатели существующих на отца и брата.

6.5 Удаление узла в дереве общего вида

Так как в дереве общего вида необходимо удалить не только узел, но и все поддерева, то удобно делать это в две функции: удаление узла и удаление его поддерева.

Листинг 5: delete_node

```
1     void delete_node(node* tree, int value)
2     {
3       if (tree != NULL) {
4         if (tree->child != NULL) {
5           if (tree->child->key == value) {
6             node *prev = tree->child;
7             tree->child = tree->child->sibling;
8             free(prev);
9             prev = NULL;
10          } else {
11            delete_node(tree->child, value);
12          }
13        }
14        if (tree->sibling != NULL) {
15          if (tree->sibling->key == value) {
```

```
16              node *prev = tree->sibling;
17              tree->sibling = tree->sibling->sibling;
18              free(prev);
19              prev = NULL;
20            } else {
21              delete_node(tree->sibling, value);
22            }
23          }
24        }
25      }
```

Алгоритм удаления узла в дереве общего вида:

1. это функция типа *void*, значит можно сразу задать условие на то, что дерево не нулевое (в противном случае функция ничего не сделает);
2. рекурсивно проходим по детям (если они есть), если при этом нашелся необходимый элемент, то делаем на него ссылку, затем присваиваем указатель на найденного ребенка его брату, удаляем ссылку (очищаем память) и присваиваем ссылке значение *NULL*;
3. то же самое делаем с сиблингами (удаляем указатель на сиблинга на найденный элемент и присваиваем ему *NULL*);

Листинг 6: delete_undertree

```
1       void delete_undertree(node *tree)
2       {
3         if (tree == NULL) {
4           return;
5         } else {
6           if (tree->child) {
7             tree = tree->child;
8           } else {
9             return;
10          }
11          if (tree->child) {
12            delete_undertree(tree->child);
13          }
14          node *next = tree;
15          node *prev = NULL;under
16          while (next->sibling) {
17            prev = next;
18            next = next->sibling;
19            if (next->child) {
20              delete_undertree(next->child);
21            }
22            free(prev);
23          }
24          free(next);
25        }
26      }
```

Алгоритм удаления поддерева узла в дереве общего вида:

1. ищем детей, если их нет, то возвращаем пустое значение;
2. если дети есть, рекурсивно проходим по ним;
3. создаем 2 ссылки: предыдущий и следующий сиблинг последнего ребенка (листа);
4. циклом while проходим по этим сиблингам и удаляем рекурсивно их детей (если они есть);
5. освобождаем память обеих ссылок.

Таким образом мы очистили всех детей у всех сиблингов.

Листинг 7: delete_tree

```
1     void delete_tree(node *tree, int value)
2     {
3       node *res = find_node(tree, value);
4       delete_undertree(res);
5       delete_node(tree, value);
6     }
```

Сама функция удаления: ищем узел, удаляем его поддерево, затем и сам узел.

6.6 Функция, возвращающая степень узла

Переходим к реализации задания варианта №8. Для его выполнения также необходимы 2 функции, первая из которых возвращает степень узла:

Листинг 8: node_degree

```
1     int node_degree(node *tree)
2     {
3       int res = 0;
4       node *cur = tree;
5       tree = tree->child;
6       if (tree == NULL) {
7         res = 0;
8       } else {
9         if (tree->sibling == NULL) {
10          res = 1;
11        } else {
12          res++;
13          while (tree->sibling != NULL) {
14            res++;
15            tree = tree->sibling;
16          }
17        }
18      }
19      tree = cur;
20      return res;
21    }
```

Алгоритм нахождения степени узла:

1. переходим к сыну узла, если у узла нет сына, то передаем счетчику значение 0;
2. иначе проходим по сиблингам (если их нет, то передаем счетчику значение 1) и считаем их количество.

6.7 Решение задачи

Необходимо написать функцию, которая считает количество узлов, степень которых равна значению элемента.
Причины, по которым я использовал глобальную переменную:

- глобальная переменная изменяется только в одной функции и выводится в int main, поэтому конфликтов, связанных с ней возникнуть не должно;
- для реализации задачи необходимо использовать рекурсию, без которой обход дерева будет неполным;
- так как мы работаем сразу и с типом *int*, и с указателем на структуру, то реализовать что-то наподобие функции find_node будет очень сложно: если мы будем возвращать указатель, то придется реализовывать список найденных узлов, а потом другой функцией уже делать подсчет, что будет куда сложнее; если мы будем возвращать *int*, то функция не поймет, что ей дальше возвращать и произойдет конфликт типов;
- если мы будем передавать в функцию типа *void* счетчик в качестве аргумента (в том числе и по ссылке), то рекурсия в конце обнулит счетчик;

- глобальная область видимости используется для подобных случаев во многих ЯП.

Таким образом, функция, которая обходит дерево и получает необходимый ответ:

Листинг 9: task

```
1     void task(node *tree)
2     {
3       node *cur = tree;
4       if (cur->key == node_degree(tree)) {
5         task_count++;
6       }
7       if (tree->child != NULL) {
8         task(tree->child);
9       }
10      if (tree->sibling != NULL) {
11        task(tree->sibling);
12      }
13    }
```

6.8 Вывод дерева на экран

Листинг 10: task

```
1     void tree_show(node *cur, int deep)
2     {
3       if (cur != NULL) {
4         for (int i = 0; i < deep; i++) {
5           printf("|   ");
6         }
7         printf("%d\n", cur->key);
8         if (cur->child != NULL) {
9           tree_show(cur->child, deep + 1);
10        }
11        tree_show(cur->sibling, deep);
12      }
13    }
```

Здесь:

1. делаем отступ на соответствующее значение глубины узла;
2. рекурсивно выводим детей и сиблингов узла.

6.9 Функция, очищающая память

Просто обходим дерево и очищаем все узлы:

Листинг 11: tree_destroy

```
1     void tree_destroy(node *t) {
2       if (t != NULL) {
3         if (t->child != NULL) {
4           tree_destroy(t->child);
5         }
6         if (t->sibling != NULL) {
7           tree_destroy(t->sibling);
8         }
9       }
10      free(t);
11    }
```

7 Сценарий выполнения работы

7.1 Реализация диалогового окна

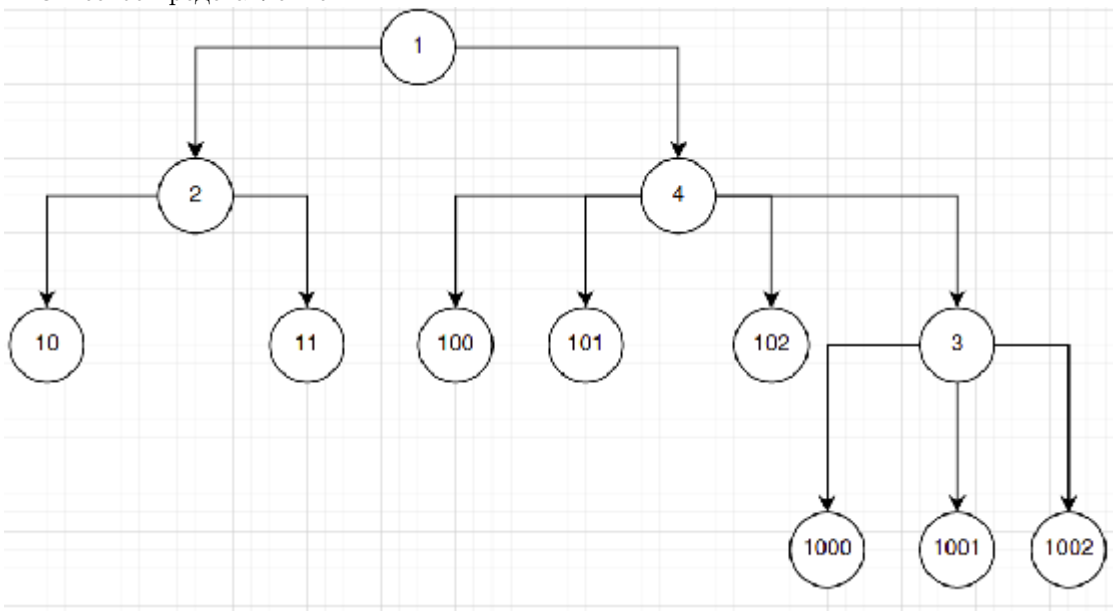
Программа по умолчанию выполняет все функции из задания, поэтому с помощью *switch* реализуем вызов самих функций:

1. создать корень дерева по заданному значению;
2. добавить узел;
3. удалить узел;
4. подсчитать количество узлов, степень которых совпадает со значением элемента;
5. вывод дерева.

И 0, чтобы выйти из программы.

7.2 Тесты

Тест 1:
Ввод: 1 1 4
Вывод: 0
Тест 2: Ввод: 1 0 4
Вывод: 1
Тест 3:
Ввод: 1 1 2 1 2 2 1 4 2 2 10 2 2 11 2 4 100 2 4 101 2 4 102 2 4 3 2 3 1000 2 3 1001 2 3 1002 4
Физическое представление:



Вывод: 3

8 Распечатка протокола

Листинг 12: code

```
#include <stdio.h>
#include <stdlib.h>

struct tree_node
{
  int key;
  struct tree_node *child;
  struct tree_node *sibling;
```

```
9         struct tree_node *parent;
10      };
11      typedef struct tree_node node;
12
13      int task_count = 0;
14
15      node *create_tree(int value)
16      {
17        node *tree = (node *) malloc(sizeof(node));
18        tree->key = value;
19        tree->child = NULL;
20        tree->sibling = NULL;
21        tree->parent = NULL;
22        return tree;
23      }
24
25      node *find_node(node* tree, int value)
26      {
27        if (tree == NULL) {
28          return NULL;
29        }
30        if (tree->key == value) {
31          return tree;
32        }
33        node* cur = NULL;
34        if (tree->child) {
35          cur = find_node(tree->child, value);
36          if (cur != NULL) {
37            return cur;
38          }
39        }
40        if (tree->sibling) {
41          cur = find_node(tree->sibling, value);
42          if (cur != NULL) {
43            return cur;
44          }
45        }
46      }
47
48      void add_node(node* tree, int parent, int value)
49      {
50        node *res = find_node(tree, parent);
51        if (res == NULL) {
52          return;
53        }
54        node *res_parent = res;
55        if (res->child == NULL) {
56          res->child = create_tree(value);
57          res->child->parent = res;
58          return;
59        }
60        res = res->child;
61        while (res->sibling != NULL) {
62          res = res->sibling;
63        }
64        res->sibling = create_tree(value);
65        res->sibling->parent = res_parent;
66      }
67
68      void delete_node(node* tree, int value)
69      {
70        if (tree != NULL) {
71          if (tree->child != NULL) {
72            if (tree->child->key == value) {
73              node *prev = tree->child;
74              tree->child = tree->child->sibling;
```

```
75              free(prev);
76              prev = NULL;
77            } else {
78              delete_node(tree->child, value);
79            }
80          }
81          if (tree->sibling != NULL) {
82            if (tree->sibling->key == value) {
83              node *prev = tree->sibling;
84              tree->sibling = tree->sibling->sibling;
85              free(prev);
86              prev = NULL;
87            } else {
88              delete_node(tree->sibling, value);
89            }
90          }
91        }
92      }
93
94      void delete_undertree(node *tree)
95      {
96        if (tree == NULL) {
97          return;
98        } else {
99          if (tree->child) {
100           tree = tree->child;
101         } else {
102           return;
103         }
104         if (tree->child) {
105           delete_undertree(tree->child);
106         }
107         node *next = tree;
108         node *prev = NULL;
109         while (next->sibling) {
110           prev = next;
111           next = next->sibling;
112           if (next->child) {
113             delete_undertree(next->child);
114           }
115           free(prev);
116         }
117         free(next);
118       }
119     }
120
121     void delete_tree(node *tree, int value)
122     {
123       node *res = find_node(tree, value);
124       delete_undertree(res);
125       delete_node(tree, value);
126     }
127
128     int node_degree(node *tree)
129     {
130       int res = 0;
131       tree = tree->child;
132       if (tree == NULL) {
133         res = 0;
134       } else {
135         if (tree->sibling == NULL) {
136           res = 1;
137         } else {
138           res++;
139           while (tree->sibling != NULL) {
140             res++;
```

```
141             tree = tree->sibling;
142           }
143         }
144       }
145       return res;
146     }
147
148     void task(node *tree)
149     {
150       node *cur = tree;
151       if (cur->key == node_degree(tree)) {
152         task_count++;
153       }
154       if (tree->child != NULL) {
155         task(tree->child);
156       }
157       if (tree->sibling != NULL) {
158         task(tree->sibling);
159       }
160     }
161
162     void tree_show(node *cur, int deep)
163     {
164       if (cur != NULL) {
165         for (int i = 0; i < deep; i++) {
166           printf("|   ");
167         }
168         printf("%d\n", cur->key);
169         if (cur->child != NULL) {
170           tree_show(cur->child, deep + 1);
171         }
172         tree_show(cur->sibling, deep);
173       }
174     }
175
176     void tree_print(node *t)
177     {
178       tree_show(t, 0);
179     }
180
181
182     void tree_destroy(node *t) {
183       if (t != NULL) {
184         if (t->child != NULL) {
185           tree_destroy(t->child);
186         }
187         if (t->sibling != NULL) {
188           tree_destroy(t->sibling);
189         }
190       }
191       free(t);
192     }
193
194     int main(int argc, char *argv[])
195     {
196       int root, flag = 1, cnt = 0;
197       node *t = NULL;
198       while (flag) {
199         printf("1. add root, 2. add node, 3. delete node, 4. print count nodes with
              degree eqial node, 5. print tree, 0. quit\n");
200         int select, value, cur;
201         scanf("%d", &select);
202         switch (select) {
203           case 1:
204           printf("add root: ");
205           scanf("%d", &root);
```

```
206             t = create_tree(root);
207             break;
208             case 2:
209             printf("enter node and value: ");
210             scanf("%d%d", &cur, &value);
211             if (t != NULL) {
212               add_node(t, cur, value);
213             }
214             break;
215             case 3:
216             printf("enter value: ");
217             scanf("%d", &value);
218             if (t != NULL) {
219               delete_tree(t, value);
220             }
221             break;
222             case 4:
223             if (t == NULL) {
224               printf("count nodes %d\n", 0);
225             } else {
226               task(t);
227               printf("count nodes %d\n", task_count);
228               task_count = 0;
229             }
230             break;
231             case 5:
232             printf("
                    +-------------------------------------------------------------------------------
233             -------------------------------------+\n");
234             if (t == NULL) {
235               printf("tree is empty!\n");
236             } else {
237               tree_print(t);
238             }
239             printf("
                    +-------------------------------------------------------------------------------
240             -------------------------------------+\n");
241             break;
242             case 0:
243             flag = 0;
244             break;
245           }
246         }
247         tree_destroy(t);
248         return 0;
249       }
```

9 Вывод

Благодаря данной лабораторной работе я написал основные функции для деревьев, а также узнал как именно реализуются такие структуры данных на языке Си.