Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
(СибГУТИ)

02.03.02 Фундаментальная информатика
и информационные технологии
Профиль: Системное программное
обеспечение
(очная форма обучения)

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

в/на кафедре вычислительных систем СибГУТИ
(наименование профильной организации/структурного подразделения СибГУТИ)

СТРУКТУРА ДАННЫХ «SPLAY-TREE»

Выполнил: Наумов А.А.
студент института ИВТ
гр. ИС-142 _____/_____/
«27» мая 2023г. (подпись)

Проверил:
Руководитель от СибГУТИ _____/_____/
«27» мая 2023г. (подпись)

Новосибирск 2023

План-график проведения учебной **практики**

Вид практики

Наумов Алексей Александрович

Фамилия Имя Отчество студента

института Информатика и вычислительная техника, 2 курса, гр. ИС-142

Направление: 02.03.02 Фундаментальная информатика и информационные технологии

Код – Наименование направления (специальности)

Профиль: Системное программное обеспечение

Место прохождения практики _____

Объем практики: **108/3** часов/ЗЕ

Вид практики ***учебная***

Тип практики ***научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)***

Срок практики с "30" января 2023 г.
по "27" мая 2023 г.

Содержание практики*:

Наименование видов деятельности	Дата (начало – окончание)
1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности	30.01.2023–01.02.2023
2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной индивидуальной темы, формирование плана работ	02.02.2023–04.02.2023
3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики	06.02.2023–11.02.2023
4. Выполнение работ в соответствии с составленным планом: – Изучение структуры данных – Разработка программы – Составление отчета	13.02.2023 – 20.05.2023
5. Анализ полученных результатов и произведенной работы Составление отчета по практике, защита отчета	22.05.2023–27.05.2023

*В соответствии с программой практики

Руководитель от СибГУТИ _____/_____/

«28» 01 2023г. (подпись)

ЗАДАНИЕ НА ПРАКТИКУ

Изучить структуру данных скошенное дерево, реализовать и описать алгоритм работы дерева.

ВВЕДЕНИЕ

В практической работе рассматривается структура данных – Splay tree (скошенное дерево). Подобно красно-черным и АВЛ-деревьям, Splay-tree (или *косое дерево*) также является самобалансирующимся бинарным деревом поиска.

В середине восьмидесятых Роберт Тарьян и Даниель Слейтор предложили несколько красивых и эффективных структур данных. Splay-дерево — одна из таких структур.

СТРУКТУРА SPLAY-TREE

- Splay Tree — это самобалансирующееся бинарное дерево поиска. Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным по памяти. После каждого обращения, даже поиска, splay tree меняет свою структуру и, за счет этого, оно позволяет находить быстрее те данные, которые использовались недавно. Splay tree не подходит для данных, которые редко или никогда не меняются, особенно в многопоточной среде. Они наиболее полезны для часто изменяемых структур данных.

Все операции со splay-деревом выполняются в среднем за время порядка O(log n), где n - количество элементов в дереве. Любая отдельная операция в худшем случае может занять время порядка θ(n).

ИДЕЯ SPLAY-TREE

Основная идея splay tree заключается в принципе - move-to-root. После обращения к любой вершине, она поднимается в корень. Подъем реализуется через повороты вершин. За один поворот, можно поменять местами родителя с ребенком, как показано на рисунке ниже. (Рис. 1)

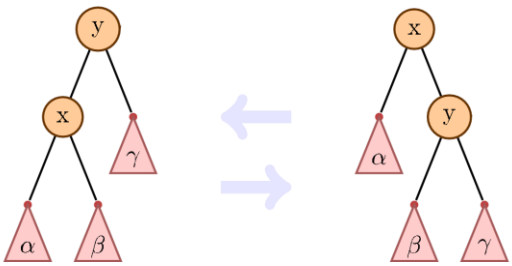


Рисунок 1

Но просто поворачивать вершину, пока она не станет корнем, недостаточно. Хитрость splay-дерева в том, что при продвижении вершины вверх, расстояние до корня сокращается не только для поднимаемой вершины, но и для всех ее потомков в текущих поддеревьях. Для этого используется техника zig-zig и zig-zag поворотов.

Основная идея zig-zig и zig-zag поворотов, рассмотреть путь от дедушки к ребенку. Если путь идет только по левым детям или только по правым, то такая ситуация называется zig-zig. Как ее обрабатывать показано на рисунке ниже. Сначала повернуть родителя, потом ребенка. (Рис 2).

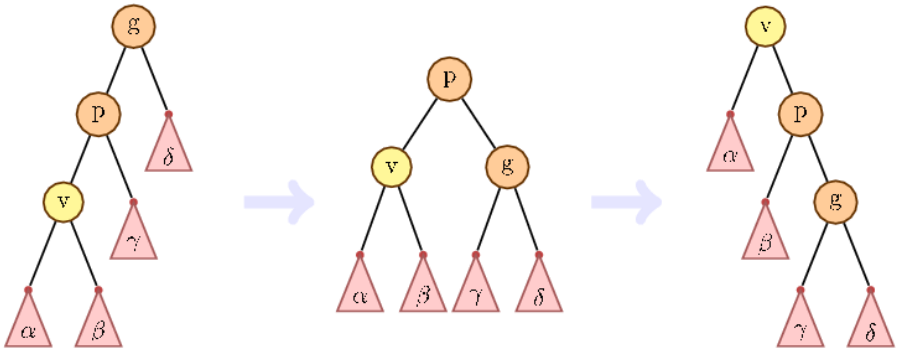


Рисунок 2

В противном случае, мы сначала меняем ребенка с текущим родителем, потом с новым. (Рис 3).

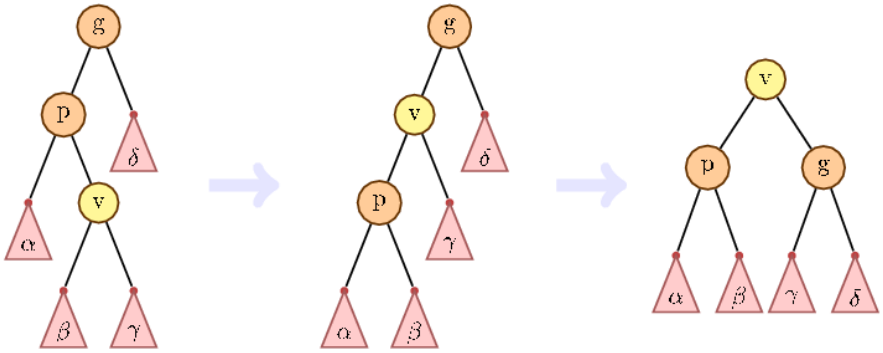


Рисунок 3

Если у вершины дедушки нет, делаем обычный поворот: (Рис 4).

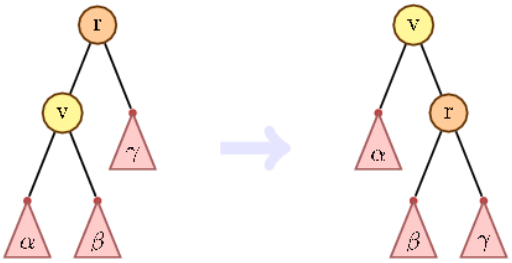


Рисунок 4

Описанная выше процедура поднятия вершины с помощью zig-zig и zig-zag поворотов является ключевой для splay-дерева.

ОПЕРАЦИИ SPLAY-TREE

Операция поиска

Операция поиска в splay-дереве представляет собой стандартный алгоритм поиска в бинарном дереве, после которого дерево выворачивается (искомый узел перемещается в корень — операция splay). Если поиск завершился успехом, то найденный узел поднимается наверх и становится новым корнем. В противном случае корнем становится последний узел, к которому был осуществлен доступ до достижения NULL.

В результате осуществления доступа к узлу возможны следующие случаи:

1. Узел является корневым. Мы просто возвращаем корень, больше ничего не делаем, так как узел, к которому осуществляется доступ, уже является корневым.

2. Zig: узел является дочерним по отношению к корню (у узла нет прародителя). Узел является либо левым потомком корня (мы делаем правый разворот), либо правым потомком своего родителя (мы делаем левый разворот). (Рис 5).

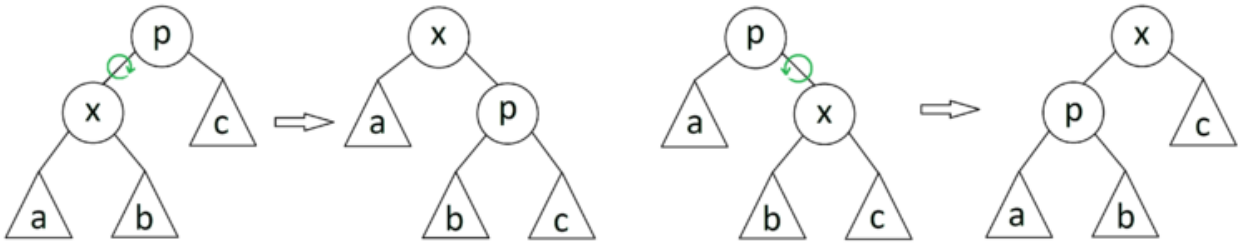


Рисунок 5

3. У *узла есть и родитель, и прародитель*. Возможны следующие варианты:

а) Zig-Zig и Zag-Zag. Узел является левым потомком родительского элемента, и родитель также является левым потомком прародителя (два разворота вправо) ИЛИ узел является правым потомком своего родительского элемента, и родитель также является правым потомком своего прародитель (два разворота влево). (Рис 6.)

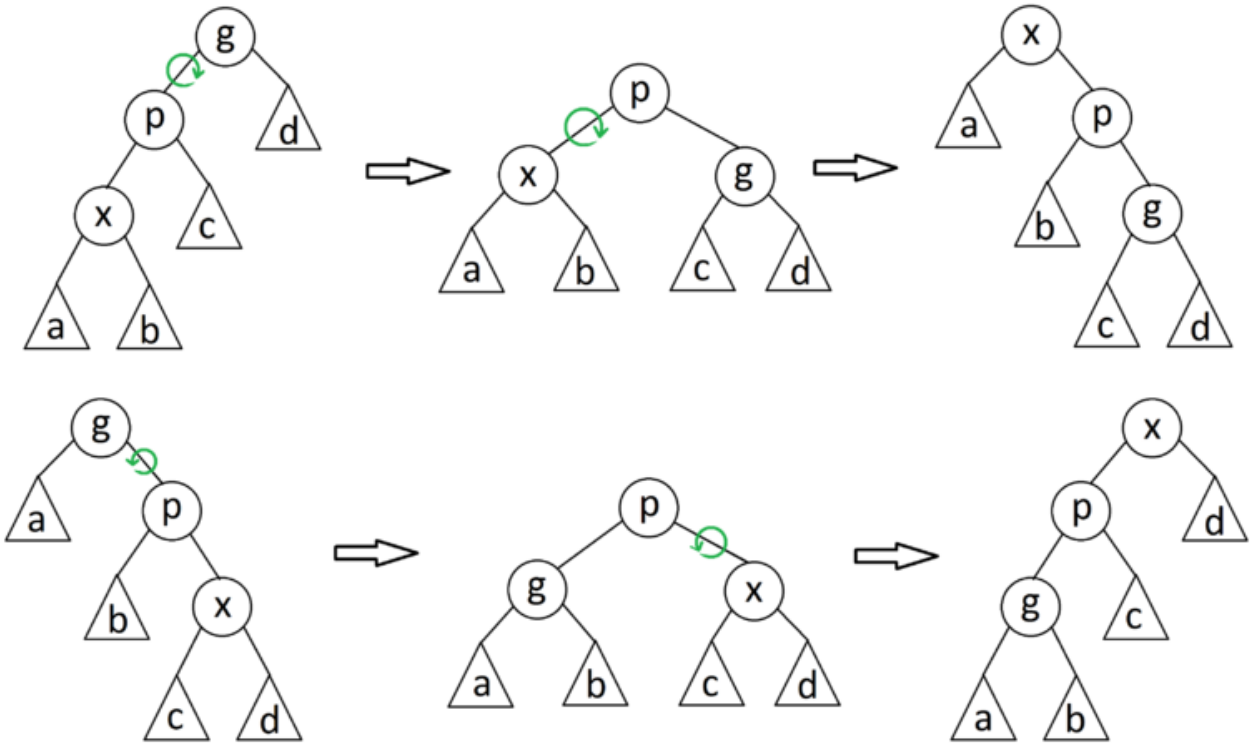


Рисунок 6

б) Zig-Zag и Zag-Zig. Узел является левым потомком по отношению к родительскому элементу, а родитель является правым потомком прародителя (разворот влево с последующим разворотом вправо) ИЛИ узел является правым потомком своего родительского элемента, а родитель является левым потомком прародителя (разворот вправо с последующим разворотом влево). (Рис 7.)

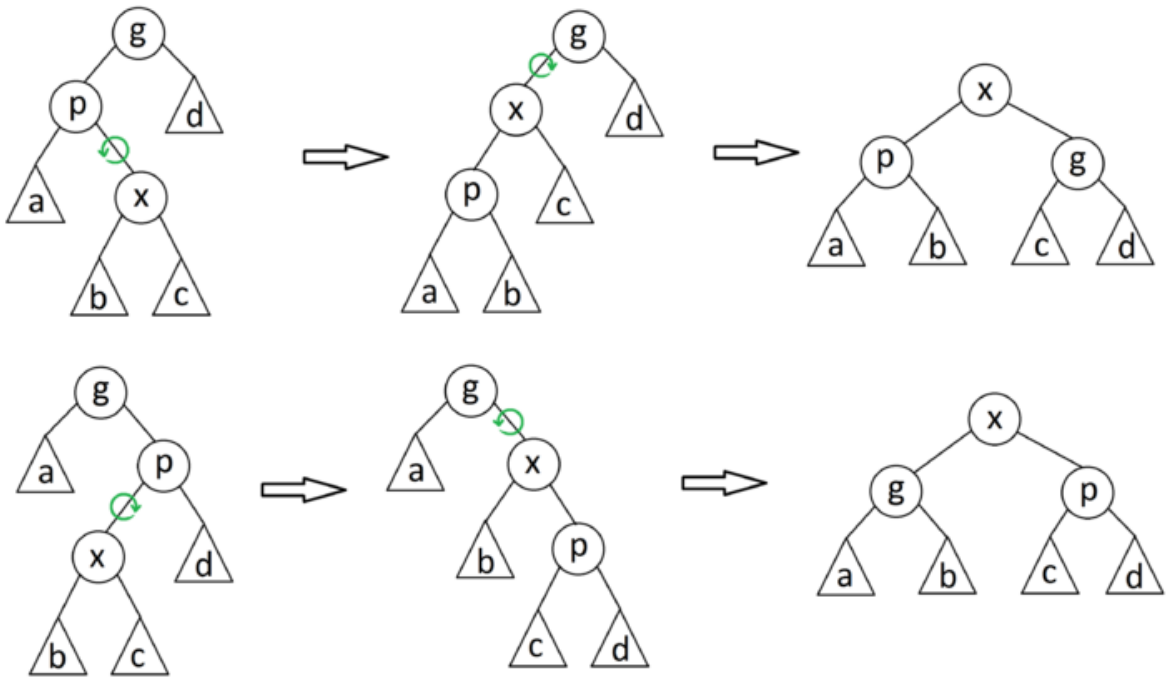


Рисунок 7

Важно отметить, что операция поиска или разворота (splay) не только переносит найденный ключ в корень, но также уравновешивает дерево.

Операция вставки

Операция вставки аналогична вставке в бинарное дерево поиска с несколькими дополнительными шагами, цель которых убедиться, что вновь вставленный ключ становится новым корнем.

Ниже приведены различные случаи при вставке ключа k в Splay-дерево:

1) Корень равен NULL: мы просто создаем новый узел и возвращаем его как корневой.

2) Выполняем операцию Splay над заданный ключом k. Если k уже присутствует, он становится новым корнем. Если он отсутствует, то новым корневым узлом становится последний узел-лист, к которому был осуществлен доступ.

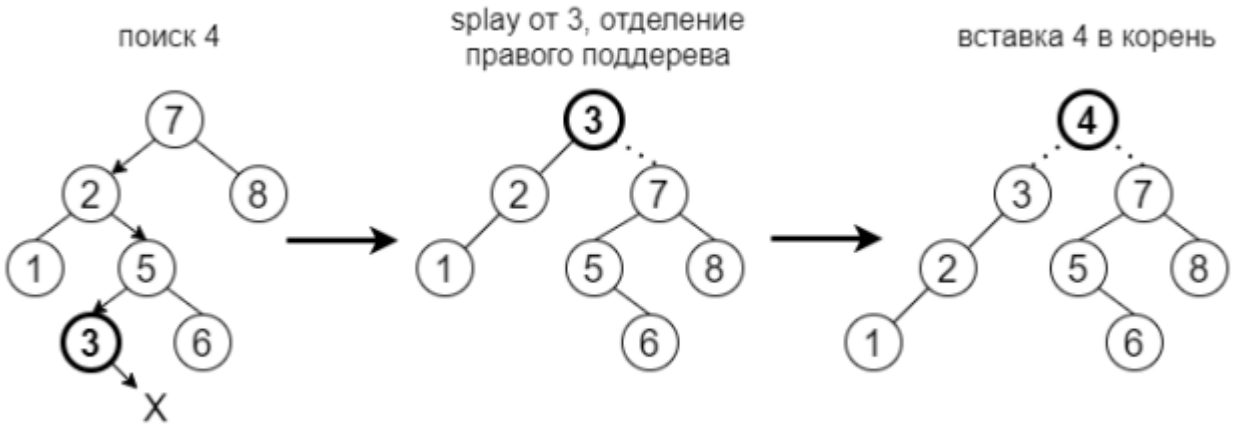
3) Если новый корневой ключ такой же, как k, ничего не делаем, поскольку k уже существует.

4) В противном случае выделяем память для нового узла и сравниваем корневой ключ с k.

4.a) Если k меньше корневого ключа, делаем корень правым дочерним элементом нового узла, копируем левый дочерний элемент корня в качестве левого дочернего элемента нового узла и делаем левый дочерний элемент корня равным NULL.

4.b) Если k больше корневого ключа, делаем корень левым дочерним элементом нового узла, копируем правый дочерний элемент корня в качестве правого дочернего элемента нового узла и делаем правый дочерний элемента корня равным NULL.

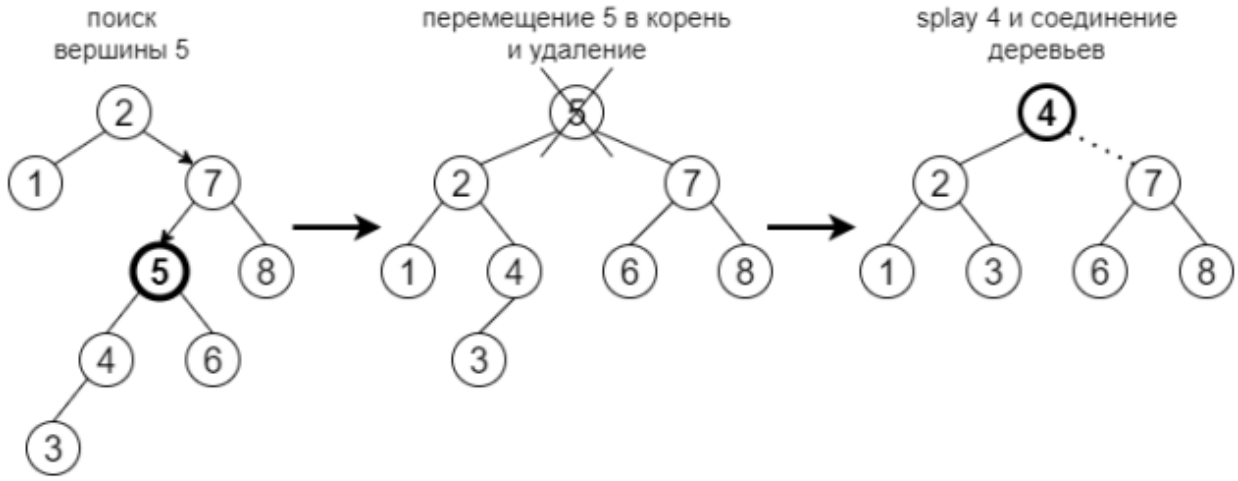
5) Возвращаем новый узел в качестве нового корня дерева.



Вставка ключа 4 в splay tree.

Операция удаления

Чтобы удалить вершину из дерева, нужно найти ее и переместить в корень, используя операцию splay. После удаления корня необходимо соединить его левое и правое поддеревья в корректное splay-дерево. Если левого поддерева нет, корнем становится корень правого поддерева, иначе в левом поддереве ищется вершина с наибольшим ключом (для этого достаточно совершить спуск по дереву из корня, каждый раз спускаясь в правого потомка, пока он есть) и перемещается в его корень с помощью splay. После этого у корня левого поддерева не будет правого потомка, поэтому на его место можно будет подвесить правое поддерево, таким образом соединив их. Ниже представлен пример удаления ключа 5 из скошенного дерева. Сначала находится положение ключа в дереве, после чего выполняется splay от него, делая удаляемую вершину корнем. Корень удаляется, в левом поддереве ищется вершина с наибольшим ключом (4), совершается splay от нее, после чего она становится корнем левого поддерева, у которого нет правого потомка, так как в этом поддереве нет вершин с ключами больше. После этого правое поддерево подвешивается к корню левого.



Удаление ключа 5 в splay tree

Вывод об операциях

Доказано, что выбор между zig, zig-zig и zig-zag во время splay по описанным выше принципам, позволяет осуществлять поиск вставку и удаление ключей в структуре за O(logn).

- Splay-деревья обладают отличным свойством локальности. Часто используемые элементы легко найти. Редкие элементы не мешаются при поиске.
- Все операции со splay-деревом в среднем занимают время порядка O(logn). Можно строго доказать, что Splay-деревья работают в среднем за время порядка O(logn) на

операцию при любой последовательности операций (при условии, что мы начинаем с пустого дерева).

- В отличие от АВЛ-дерева, splay-дерево может изменяться даже при выполнении операций чтения, таких как поиск.

Применение:

Splay-деревья стали наиболее широко используемой базовой структурой данных, изобретенной за последние 30 лет, потому что они являются самым быстрым типом сбалансированного дерева поиска для огромного множества приложений.

Splay-деревья используются в Windows NT (в виртуальной памяти, сети и коде файловой системы), компиляторе gcc и библиотеке GNU C++, редакторе строк sed, сетевых маршрутизаторах Fore Systems, наиболее популярной реализации Unix malloc, загружаемых модулях ядра Linux и во многих других программах

Экспериментальное исследование эффективности алгоритма

Анализ

Ясно, что в таком дереве нельзя гарантировать сложность операций O(log(n)) (вдруг нас попросят найти глубоко залегшую вершину в несбалансированном на данный момент дереве?). Вместо этого, гарантируется сложность операции O(log(n)), то есть любая последовательность из m операций с деревом размера n работает за O((n+m)*log(n)). Более того, splay-дерево обладает некоторыми магическими свойствами, за счет которого оно на практике может оказаться намного эффективнее остальных вариантов. Например, вершины, к которым обращались недавно, оказываются ближе к корню и доступ к ним ускоряется. Более того, доказано что если вероятности обращения к элементам фиксированы, то splay-дерево будет работать асимптотически не медленней любой другой реализации бинарных деревьев. Еще одно преимущество в том, что отсутствует overhead по памяти, так как не нужно хранить никакой дополнительной информации.

В отличие от других вариантов, операция поиска в дереве модифицирует само дерево, поэтому в случае равномерного обращения к элементам splay-дерево будет работать медленней. Однако на практике оно часто дает ощутимый прирост производительности. К сожалению, из-за отсутствия гарантий на производительность отдельных операций, splay-деревья неприминимы в realtime-системах (например в ядре ОС, garbage-collector'ах), а так же в библиотеках общего назначения.

Асимптотическая сложность операций

	В среднем	В худшем случае
Расход памяти	**O(n)**	**O(n)**
Поиск	**O(log n)**	**O(log n)**
Вставка	**O(log n)**	**O(log n)**
Удаление	**O(log n)**	**O(log n)**

Таблица 1

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработан и исследован алгоритм splay-tree. Осуществлено моделирование разработанного алгоритма. Показано, что splay-tree о предоставляет самоизменяющуюся структуру — структуру, характеризующуюся тенденцией хранить узлы, к которым часто происходит обращение, вблизи верхушки дерева, в то время как узлы, к которым обращение происходит редко, перемещаются ближе к листьям. Таким образом время обращения к часто посещаемым узлам будет меньше, а время обращения к редко посещаемым узлам — больше среднего. Расширяющееся дерево не обладает никакими явными функциями балансировки, но процесс скоса узлов к корню способствует поддержанию дерева в сбалансированном виде.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Tarjan* «Data Structures and Networks Algorithms» *1983г.*
2. *Sleator, Daniel D.; Tarjan, Robert E.* (1985), «Self-Adjusting Binary Search Trees»
3. *Томас Х. Кормен и др.* Алгоритмы: построение и анализ. — 2-е изд. — М.: Издательский дом «Вильямс», 2007. — С. 1296.
4. *Daniel Sleator, Robert Tarjan.* «A data structure for dynamic trees» — Journal of Computer and System Sciences, 1983. — С. 262—391.
5. *М. Г. Курносов, Д. М. Берлизов.* «Алгоритмы и структуры обработки информации» – Новосибирск: Параллель, 2019. – 227 с.

ПРИЛОЖЕНИЯ

В программе идет заполнения скошенного дерева из массива чисел, одновременно со вставкой в дерево идет вывод максимального и минимального элемента дерева.

```c
int main() {
    splay_tree *tree = new_tree(&compare_int);
    int values[] = {3, 4, 1, 2, 8, 5, 7};
    output(tree);
    for(int i = 0; i < 7; i++) {
        insert(tree, &values[i]);
        printf("+%d: min %d max %d\n", values[i],
               *(int*)first(tree)->value, *(int*)last(tree)->value);
        output(tree);
    }
    for(int i = 0; i < 7; i++) {
        delete(tree, &values[i]);
        printf("-%d: ", values[i]);
        if(first(tree) == NULL) {
            printf("EMPTY\n");
        } else {
            printf("min %d max %d\n",
                   *(int*)first(tree)->value, *(int*)last(tree)->value);
        }
        output(tree);
    }
    return 0;
}
```

Результат работы программы:

```
+3: min 3 max 3
3
3
+4: min 3 max 4
3 4
3 4
+1: min 1 max 4
1 3 4
1 3 4
+2: min 1 max 4
1 2 3 4
1 2 3 4
+8: min 1 max 8
1 2 3 4 8
1 2 3 4 8
+5: min 1 max 8
1 2 3 4 5 8
1 2 3 4 5 8
+7: min 1 max 8
1 2 3 4 5 7 8
1 2 3 4 5 7 8
-3: min 1 max 8
1 2 4 5 7 8
1 2 4 5 7 8
-4: min 1 max 8
1 2 5 7 8
1 2 5 7 8
-1: min 2 max 8
2 5 7 8
2 5 7 8
-2: min 5 max 8
5 7 8
5 7 8
-8: min 5 max 7
5 7
5 7
-5: min 7 max 7
7
7
-7: EMPTY
```

Листинг программы:

Main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "splay.h"

int compare_int(void *l, void *r) {
    return *((int*)l) - *((int*)r);
}

void output(splay_tree *tree) {
    splay_node *node = first(tree);
    while(node != NULL) {
        printf("%d ", *(int*)node->value);
        node = next(node);
    }
    printf("\n");
    void **t = contents(tree);
    for(int i = 0; i < tree->size; i++) {
        printf("%d ", *(int*)t[i]);
    }
    printf("\n");
}

int main() {
    splay_tree *tree = new_tree(&compare_int);
    int values[] = {3, 4, 1, 2, 8, 5, 7};
    output(tree);
    for(int i = 0; i < 7; i++) {
        insert(tree, &values[i]);
        printf("+%d: min %d max %d\n", values[i],
                *(int*)first(tree)->value, *(int*)last(tree)->value);
        output(tree);
    }
    for(int i = 0; i < 7; i++) {
        delete(tree, &values[i]);
        printf("-%d: ", values[i]);
        if(first(tree) == NULL) {
            printf("EMPTY\n");
        } else {
            printf("min %d max %d\n",
                    *(int*)first(tree)->value, *(int*)last(tree)->value);
        }
        output(tree);
    }
    return 0;
}
```

Splay.c

```c
#include <stdlib.h>
#include <assert.h>
#include "splay.h"

void check_sanity(splay_tree *tree);

/*наиболее важная функция */
void rotate(splay_node *child);
/* и еще несколько */
```

```
splay_node* leftmost(splay_node *node);
splay_node* rightmost(splay_node *node);


/* splay узла x */
void zig(splay_node *x, splay_node *p);
void zigzig(splay_node *x, splay_node *p);
void zigzag(splay_node *x, splay_node *p);
void splay(splay_tree *tree, splay_node *x) {
    while(1) {
        splay_node *p = x->parent;
        if(p == NULL) {
            tree->root = x;
            return;
        }
        splay_node *g = p->parent;
        if(p->parent == NULL)
            zig(x, p);
        else
            if((x == p->left && p == g->left) ||
                    (x == p->right && p == g->right))
                zigzig(x, p);
            else
                zigzag(x, p);
    }
}

/* Когда p является корнем, поверните по краю между x и p.*/
void zig(splay_node *x, splay_node *p) {
    rotate(x);
}

/* Когда оба x и p являются левыми (или оба правыми) дочерними элементами,
* поверните по краю между p и g, затем по краю между x и p.
 */
void zigzig(splay_node *x, splay_node *p) {
    rotate(p);
    rotate(x);
}

/* Когда один из x и p является левым дочерним элементом, а другой - правым
дочерним элементом,
 * поверните по краю между x и p, затем по новому краю между x и g.
 */
void zigzag(splay_node *x, splay_node *p) {
    rotate(x);
    rotate(x);
}

/* Возвращает пустое дерево с сохранением компаратора.*/
 splay_tree* new_tree(comparator comp) {
    splay_tree *new = malloc(sizeof(splay_tree));
    new->comp = comp;
    new->root = NULL;
    new->size = 0;
    return new;
}

/* Вставьте и верните новый узел с заданным значением, расширяя дерево.
```

```
 * Вставка, по сути, представляет собой обычную вставку в двоичном дереве поиска.
 */
 splay_node* insert(splay_tree *tree, void *value) {
    splay_node *new = malloc(sizeof(splay_node));
    new->value = value;
    new->left = NULL;
    new->right = NULL;
    if(tree->root == NULL) {
        tree->root = new;
        new->parent = NULL;
    } else {
        splay_node *curr = tree->root;
        splay_node *parent;
        int left;
        while(curr != NULL) {
            parent = curr;
            if(tree->comp(new->value, curr->value) < 0) {
                left = 1;
                curr = curr->left;
            } else {
                left = 0;
                curr = curr->right;
            }
        }
        new->parent = parent;
        if(left)
            parent->left = new;
        else
            parent->right = new;
    }
    splay(tree, new);
    tree->size++;
    return new;
}

/* Найдите узел с заданным значением, развернув дерево. */
 splay_node* find(splay_tree *tree, void *value) {
    splay_node *curr = tree->root;
    int found = 0;
    while(curr != NULL && !found) {
        int relation = tree->comp(value, curr->value);
        if(relation == 0) {
            found = 1;
        } else if(relation < 0) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }
    if(curr != NULL)
        splay(tree, curr);
    return curr;
}

/* Удалите узел с заданным значением, развернув дерево.. */
 void delete(splay_tree *tree, void *value) {
    splay_node *node = find(tree, value);
    delete_hint(tree, node);
}
```

```
/*Удалите узел, указанный указателем, развернув дерево */
 void delete_hint(splay_tree *tree, splay_node *node) {
    if(node == NULL)
        return;
    splay(tree, node); /* Теперь узел - это корень дерева. */
    if(node->left == NULL) {
        tree->root = node->right;
        if(tree->root != NULL)
            tree->root->parent = NULL;
    } else if(node->right == NULL) {
        tree->root = node->left;
        tree->root->parent = NULL;
    } else {
        splay_node *x = leftmost(node->right);
        if(x->parent != node) {
            x->parent->left = x->right;
            if(x->right != NULL)
                x->right->parent = x->parent;
            x->right = node->right;
            x->right->parent = x;
        }
        tree->root = x;
        x->parent = NULL;
        x->left = node->left;
        x->left->parent = x;
    }
    free(node);
    tree->size--;
    check_sanity(tree);
}

 splay_node* first(splay_tree *tree) {
    return leftmost(tree->root);
}

/* Возвращает минимальный узел, который больше заданного.
 * Это либо:
 * - крайний левый дочерний элемент в правом поддереве
 * - ближайший асцендент, для которого данный узел находится в левом поддереве
 */
 splay_node* next(splay_node *node) {
    if(node->right != NULL)
        return leftmost(node->right);
    while(node->parent != NULL && node == node->parent->right)
        node = node->parent;
    return node->parent;
}

 splay_node* last(splay_tree *tree) {
    return rightmost(tree->root);
}

/* Обход дерева по порядку. */
void store(splay_node *node, void ***out);
 void* contents(splay_tree *tree) {
    if(tree->size == 0)
        return NULL;
    void **out = malloc(tree->size * sizeof(void*));
```

```
    void ***tmp = &out;
    store(tree->root, tmp);
    return out - tree->size;
}

void store(splay_node *node, void ***out) {
    if(node->left != NULL)
        store(node->left, out);
    **out = node->value;
    (*out)++;
    if(node->right != NULL)
        store(node->right, out);
}
/* Это изменяет родительские отношения, копирует указатель на старого родителя. */
void mark_gp(splay_node *child);

/* Поверните, чтобы данный дочерний элемент занял место своего родителя в дереве.
*/
void rotate(splay_node *child) {
    splay_node *parent = child->parent;
    assert(parent != NULL);
    if(parent->left == child) { /* левый ребенок */
        mark_gp(child);
        parent->left = child->right;
        if(child->right != NULL)
            child->right->parent = parent;
        child->right = parent;
    } else { /* правый ребенок */
        mark_gp(child);
        parent->right = child->left;
        if(child->left != NULL)
            child->left->parent = parent;
        child->left = parent;
    }
}

void mark_gp(splay_node *child) {
    splay_node *parent = child->parent;
    splay_node *grand = parent->parent;
    child->parent = grand;
    parent->parent = child;
    if(grand == NULL)
        return;
    if(grand->left == parent)
        grand->left = child;
    else
        grand->right = child;
}

splay_node* leftmost(splay_node *node) {
    splay_node *parent = NULL;
    while(node != NULL) {
        parent = node;
        node = node->left;
    }
    return parent;
}

splay_node* rightmost(splay_node *node) {
```

```
    splay_node *parent = NULL;
    while(node != NULL) {
        parent = node;
        node = node->right;
    }
    return parent;
}

#ifdef DEBUG
int check_node_sanity(splay_node *x, void *floor, void *ceil, comparator comp) {
    int count = 1;
    if(x->left != NULL) {
        assert(x->left->parent == x);
        void *new_floor;
        if(floor == NULL || comp(x->value, floor) < 0)
            new_floor = x->value;
        else
            new_floor = floor;
        count += check_node_sanity(x->left, new_floor, ceil, comp);
    }
    if(x->right != NULL) {
        assert(x->right->parent == x);
        void *new_ceil;
        if(ceil == NULL || comp(x->value, ceil) > 0)
            new_ceil = x->value;
        else
            new_ceil = ceil;
        count += check_node_sanity(x->right, floor, new_ceil, comp);
    }
    return count;
}
#endif

void check_sanity(splay_tree *tree) {
#ifdef DEBUG
    if(tree->root == NULL) {
        assert(tree->size == 0);
    } else {
        int reachable = check_node_sanity(tree->root, NULL, NULL, tree->comp);
        assert(reachable == tree->size);
    }
#endif
}
```

Splay.h

```
#ifndef SPLAY_H
#define SPLAY_H

typedef int (*comparator)(void *left, void *right);

typedef struct splay_node {
    struct splay_node *parent, *left, *right;
    void *value;
} splay_node;

typedef struct splay_tree {
    splay_node *root;
```

```
    comparator comp;
    int size;
} splay_tree;

splay_tree* new_tree(comparator comp);
splay_node* insert(splay_tree *tree, void *value);
splay_node* find(splay_tree *tree, void *value);
splay_node* first(splay_tree *tree);
splay_node* next(splay_node *node);
splay_node* last(splay_tree *tree);
void* contents(splay_tree *tree);
void delete(splay_tree *tree, void *value);
void delete_hint(splay_tree *tree, splay_node *node);

#endif
```

Makefile

```
all: main
.PHONY: main
main: main.c splay.c
        gcc -Wall -Wextra -o $@ $^

.PHONY: clean
clean:
        rm -rf main
```

Отзыв о работе студента

Наумов Алексей Александрович

(ФИО студента)

Уровень освоения компетенций

Наумов Алексей Александрович

(ФИО студента)

Компетенции	Уровень сформированности компетенций
ОПК-1 - Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности	

отметка о зачете _____

Руководитель практики от СибГУТИ:

_____	_____	_____
Должность руководителя	подпись	ФИО руководителя

"___" _____ 20___ г.