ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

Splay- Tree (Скошенные деревья)

Выполнил студент: Дураков Н.А

Группы: ИВ-721

Работу принял Преподаватель Кафедры ВС Д.М Берлизов

Защищена Оценка

Новосибирск – 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ………………………………………………………………………3

1.СТРУКТУРА……………………………………………………………..…......4

2. ОСНОВНЫЕ ОПЕРАЦИИ С ДЕРЕВОМ……………………………………5

3. СЛОЖНОСТЬОПЕРАЦИЙ ………………………………………………… 8

4. ЗАКЛЮЧЕНИЕ………………………………………………………………..9

5. ИСТОЧНИКИ ИНФОРМАЦИИ…………………………………………….10

6. ПРИЛОЖЕНИЕ.ЛИСТИНГ………………………..…………………….……………11

ВВЕДЕНИЕ

Расширяющееся (англ. splay tree) или косое дерево является двоичным деревом поиска, в котором поддерживается свойство сбалансированности. Это дерево принадлежит классу «саморегулирующихся деревьев», которые поддерживают необходимый баланс ветвления дерева, чтобы обеспечить выполнение операций поиска, добавления и удаления за логарифмическое время от числа хранимых элементов.

Это реализуется без использования каких-либо дополнительных полей в узлах дерева (как, например, в красно-чёрных деревьях или АВЛ-деревьях, где в вершинах хранится, соответственно, цвет вершины и глубина поддерева). Вместо этого «расширяющие операции» (splay operation), частью которых являются вращения, выполняются при каждом обращении к дереву.

Расширяющееся дерево придумали Роберт Тарьян и Даниель Слейтор в 1983 году.

Цель работы – реализовать и исследовать косое дерево.

СТРУКТУРА

Так выглядит каждый узел расширяющегося дерева.

|  |
| --- |
| typedef struct Node {  struct Node \*left;  struct Node \*right;  struct Node \*parent;  int value;  } Node; |

struct Node \*left - указатель на левый дочерний узел.

struct Node \*right - указатель на правый дочерний узел.

struct Node \*parent - указатель на родительский узел.

int value - значение узла.

Структура узла расширяющегося дерева выглядит как у обычного двоичного дерева поиска.

Так это выглядит на рисунке. (Рис.1)

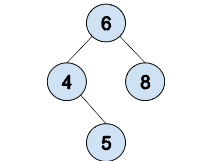


Рисунок 1. Структура узла.

ОСНОВНЫЕ ОПЕРАЦИИ С ДЕРЕВОМ

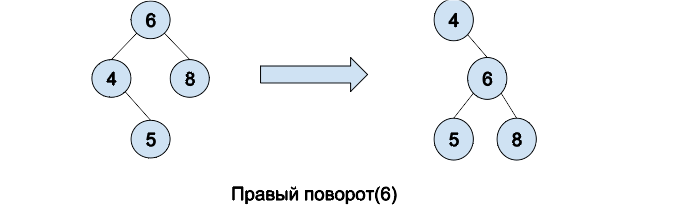
**ZIG(ZAG)** - обычный поворот влево (вправо). Он выполняется перестановкой указателей. Выполняется, когда у узла нет «дедушки». Если узел является левым потомком, то выполняется правый поворот, если узел является правым потомком, то выполняем левый поворот. (Рис. 2)

Рисунок 2. Правый поворот.

**ZIG-ZIG (ZAG-ZAG) -** комбинация из двух правых или левых поворотов. Выполняется если у узла есть «дедушка». Тут могут возникнуть два случая:

1. Узел является левым потомком, и родитель этого узла является левым потомком (Left-Left Case на рисунке 3). В этом случае выполняется два правых поворота.
2. Узел является правым потомком, и родитель этого узла является правым потомком (Right-Right Casе). В этом случае выполняется два левых поворота.

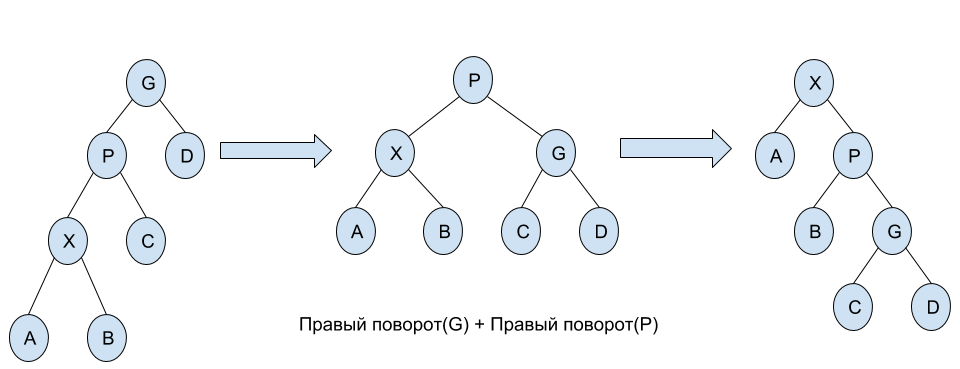


Рисунок 3. Левый-Левый случай(Х).

**ZIG-ZAG(ZAG-ZIG)** - комбинация из одного левого и правого поворота или наоборот. Выполняется если у узла есть «дедушка». Тут опять же есть два случая:

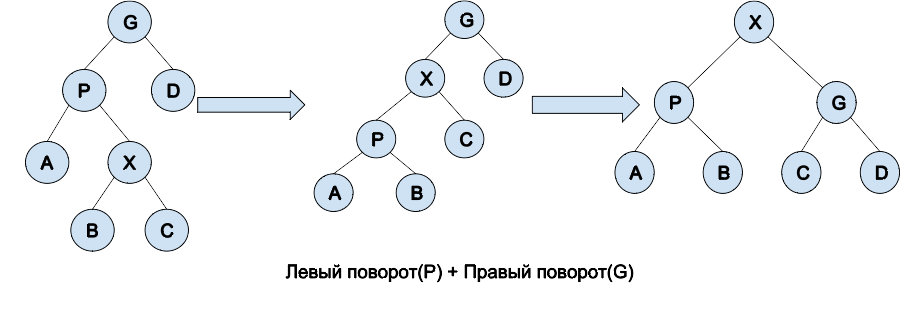
1. Узел является правым потомком, а родитель этого узла является левым потомком (Left-Right Case на рисунке). В этом случае выполняется один левый поворот, а затем один правый поворот. (Рис.4)
2. Узел является левым потомком, а родитель этого узла является правым потомком (Right - Left Case). В этом случае выполняется один правый поворот, а затем один левый поворот.

Рисунок 4. Левый-правый случай(Х).

**INSERT -** вставка происходит как в обычном двоичном дереве поиска + Splay(Рис.5)

1. Если элемент, который вставляют, меньше узла, то он становится левым дочерним узлом.
2. Если элемент, который вставляют, больше узла, то он становится правым дочерним узлом.

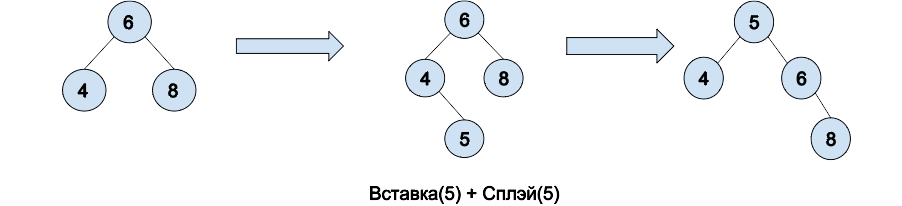


Рисунок 5. Вставка элемента 5.

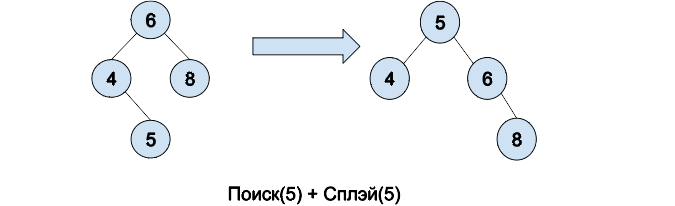
**SEARCH -** поиск происходит, как в обычном двоичном дереве поиска, только после того, как мы находим элемент мы делаем для него Splay. (Рис.6)

Рисунок 6. Поиск узла 5.

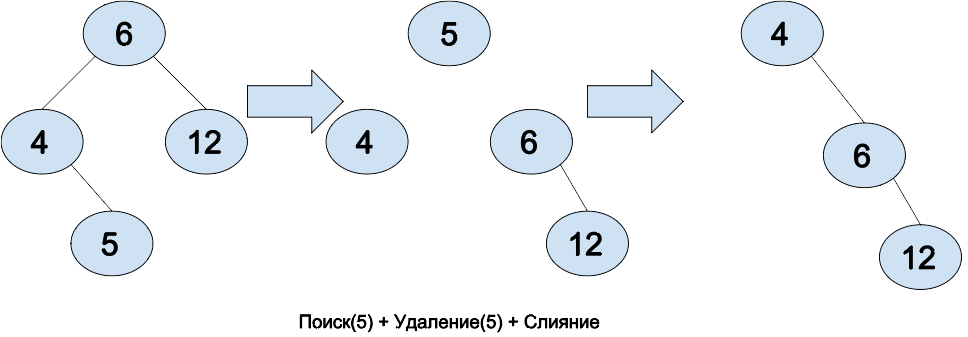
**DELETE** - Находим элемент в дереве, делаем Splay для него, делаем текущим деревом merge его детей. (Рис. 7)

Рисунок 7. Удаление узла 5.

СЛОЖНОСТЬ ОПЕРАЦИЙ

Сложность операций.

|  |  |  |
| --- | --- | --- |
| **Операция** | **Сложность(лучш)** | **Сложность(худш)** |
| Поиск | *О*(log*n*) | *О*(log*n*) |
| Удаление | *О*(log*n*) | *О*(log*n*) |
| Вставка | *О*(log*n*) | *О*(log*n*) |
| Zig,Zag, и т.п. | *О*(1) | *О*(1) |

Таблица скорости выполнения поиска.

|  |  |
| --- | --- |
| **Количество элементов** | **Скорость выполнения Lookup** |
| 10000 | 0,0012 c |
| 20000 | 0,0025 c |
| 30000 | 0,0032 c |
| 40000 | 0,0035 c |
| 50000 | 0,0042 c |

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработан и исследован алгоритм расширяющегося дерева.

В ходе исследования выявлено, что главное преимущество Splay tree заключается в том, что он хранит наиболее запрошенные узлы в верхней части дерева, уменьшая время для последующих запросов. Эта локальность ссылок делает splay tree очень полезным для таких систем, как сбор мусора для языков программирования или систем кэширования.

ИСТОЧНИКИ ИНФОРМАЦИИ

1) Сайт Университета ИТМО

URL: <https://neerc.ifmo.ru/wiki/index.php?title=Splay-tree>

2) Cайт GeeksforGeeks.com

URL: https://www.geeksforgeeks.org/splay-tree

3) Sleator D. D., Tarjan R. E. A data structure for dynamic trees //Journal of computer and system sciences. – 1983. – Т. 26. – №. 3. – С. 362-391.

doi: 10.1016/0022-0000(83)90006-5

ПРИЛОЖЕНИЕ. ЛИСТИНГ

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

struct Node \*left;

struct Node \*right;

struct Node \*parent;

int value;

} Node;

Node \*root;

void rightRotate(Node \*P) {

Node \*T = P->left;

Node \*B = T->right;

Node \*D = P->parent;

if (D) {

if (D->right == P)

D->right = T;

else

D->left = T;

}

if (B) B->parent = P;

T->parent = D;

T->right = P;

P->parent = T;

P->left = B;

}

void leftRotate(Node \*P) {

Node \*T = P->right;

Node \*B = T->left;

Node \*D = P->parent;

if (D) {

if (D->right == P)

D->right = T;

else

D->left = T;

}

if (B) B->parent = P;

T->parent = D;

T->left = P;

P->parent = T;

P->right = B;

}

void Splay(Node \*T) {

while (1) {

Node \*p = T->parent;

if (!p) break;

Node \*pp = p->parent;

if (!pp) // Zig

{

if (p->left == T)

rightRotate(p);

else

leftRotate(p);

break;

}

if (pp->left == p) {

if (p->left == T) { // ZigZig

rightRotate(pp);

rightRotate(p);

} else { // ZigZag

leftRotate(p);

rightRotate(pp);

}

} else {

if (p->left == T) { // ZigZag

rightRotate(p);

leftRotate(pp);

} else { // ZigZig

leftRotate(pp);

leftRotate(p);

}

}

}

root = T;

}

void Insert(int value) {

if (!root) {

root = (Node \*)malloc(sizeof(Node));

root->left = NULL;

root->right = NULL;

root->parent = NULL;

root->value = value;

return;

}

Node \*P = root;

while (1) {

if (P->value == value) break;

if (value < (P->value)) {

if (P->left)

P = P->left;

else {

P->left = (Node \*)malloc(sizeof(Node));

P->left->parent = P;

P->left->right = NULL;

P->left->left = NULL;

P->left->value = value;

P = P->left;

break;

}

} else {

if (P->right)

P = P->right;

else {

P->right = (Node \*)malloc(sizeof(Node));

P->right->parent = P;

P->right->right = NULL;

P->right->left = NULL;

P->right->value = value;

P = P->right;

break;

}

}

}

Splay(P);

}

void show\_it(Node \*R) {

if (!R) return;

show\_it(R->left);

printf("Value: %d |", R->value);

if (R->left) printf("Left is: %d |", R->left->value);

if (R->right) printf("Right is: %d |", R->right->value);

puts("");

show\_it(R->right);

}

Node \*lookup(int v) {

if (!root) return NULL;

Node \*P = root;

while (P) {

if (P->value == v) break;

if (v < (P->value)) {

if (P->left)

P = P->left;

else

break;

} else {

if (P->right)

P = P->right;

else

break;

}

}

Splay(P);

if (P->value == v)

return P;

else

return NULL;

}

int delete\_it(int v) {

Node \*N = lookup(v);

if (!N) return 0;

Splay(N); // check once more;

Node \*P = N->left;

if (!P) {

root = N->right;

root->parent = NULL;

free(N);

return 1;

}

while (P->right) P = P->right;

if (N->right) {

P->right = N->right;

N->right->parent = P;

}

root = N->left;

root->parent = NULL;

free(N);

return 1;

}