Санкт-Петербургский политехнический университет имени Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Отчёт

по курсовой работе

Дисциплина: Объектно-ориентированное программирование

Тема: splay tree

Студент гр. 3331506/80401 Мирошниченко Д. О.

Преподаватель Ананьевский М.С.

«___»_____2021 г.

Санкт-Петербург

2021

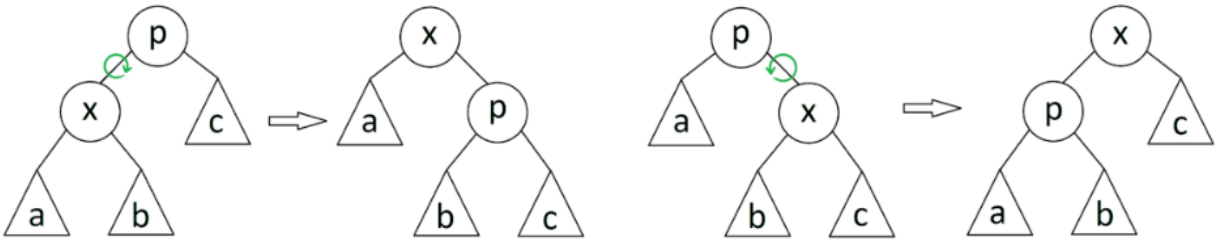
ВВЕДЕНИЕ

Splay-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году. Недавно использованные объекты перемещаются ближе к корню при обращениях к ним, что делает более быстрым доступ к ним. Данная особенность splay-деревьев позволяет использовать их там, где доступ к одним элементам необходим чаще, чем к другим.

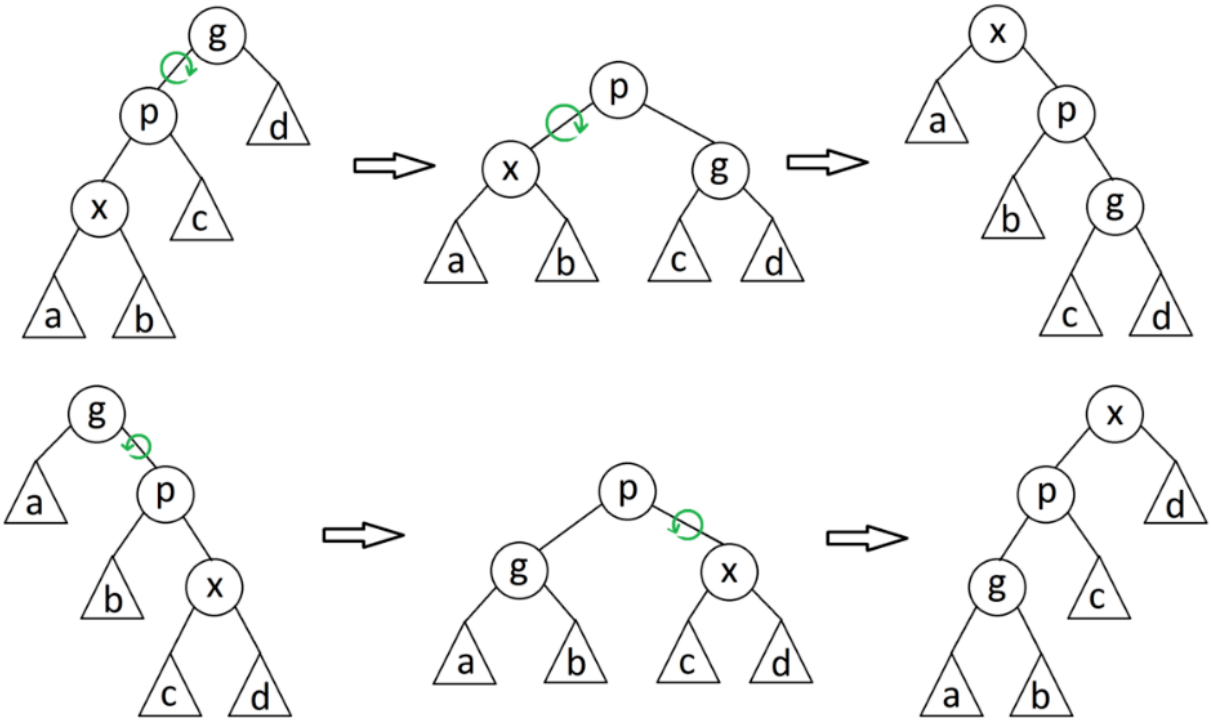
АЛГОРИТМ

Алгоритм splay-tree отличается тем, что помещает узел, к которому произошло обращение, в корень. Собственно, операции перемещения наиболее эффективным способом и составляют интерес.

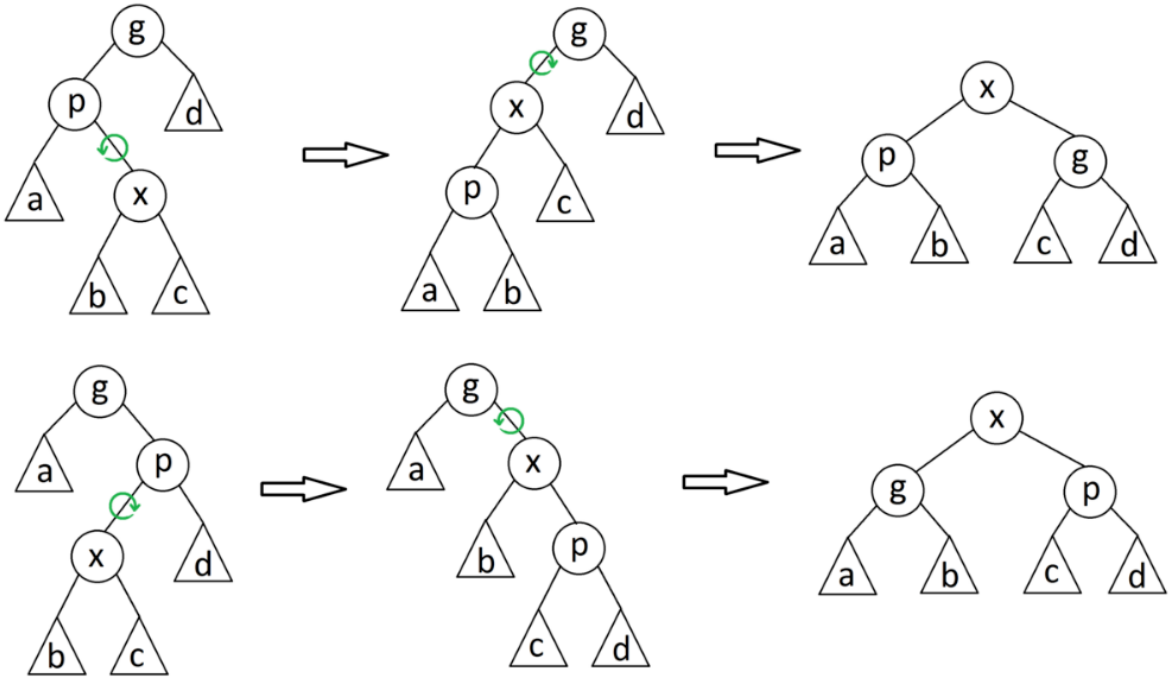
Если узел является ребёнком непосредственно корня, то происходит перестановка мест его с родителем при помощи поворота, называемая zig:



Если узел и его родитель оба правые или левые дети, то совершается операция zig-zig, совершаемая двумя поворотами:



А если узел правый, а его родитель левый ребёнок или наоборот, совершается операция zig-zag:



Ключевая операция – splay, перемещающая при помощи команд zig, zig-zig и zig-zag нужный узел в корень дерева.

Операция поиска «find» работает также как для обычного бинарного дерева поиска, однако после нахождения элемента он сразу при помощи команды splay отправляется в корень.

Операция удаления элемента «del» происходит следующим образом:

- При помощи splay достаём элемент в корень дерева и отделяем два дерева, начинающихся с его потомков – правое и левое.
- Находим наименьший элемент в левом дереве. Он получается таким, что он меньше любого элемента из левого дерева, но больше любого из правого.
- Применяем к этому наименьшему элементу операцию splay, получается так, что он является корнем левого дерева.
- Ставим этот элемент корнем всего дерева и подставляем неизменённое правое дерево справа.

Операция добавления элемента «add» происходит также при помощи отделения двух деревьев и подставления в корень нового элемента.

СКОРОСТЬ РАБОТЫ АЛГОРИТМА

Расход памяти алгоритма – O(n), поскольку он не требует дополнительного выделения памяти

Splay дерево является саморегулирующимся деревом (то есть доступ к любому элементу происходит за логарифмическое время), поддерживающим баланс ветвления (достаточно взглянуть на операции zig, zig-zig и zig-zag, чтобы понять, что при выполнении этих операций, структура дерева не изменяется, как и суммарный ранг узлов).

Поэтому доступ к любому элементу по ключу вычисляется как $O(\log n)$. Тогда необходимо узнать скорость операции splay.

Пусть поворот двух элементов (занимающий на самом деле константное количество времени) занимает время в условную единицу, проверка того, с какой стороны потомок/родитель пусть также занимает единицу (на самом же деле, эта операция занимает времени ещё меньше, чем поворот).

Для операции zig нужно 2 условные единицы, при этом элемент «поднимается» в дереве на 1 уровень. Для zig-zag и zig-zig нужно уже 4 условные единицы (две проверки и два поворота), при этом элемент поднимается на 2 уровня. Всего уровней $\log n$, поскольку дерево сбалансированное. То есть необходимо $2\log n$ единиц условного времени. Следовательно, операция splay выполняется за $O(\log n)$.

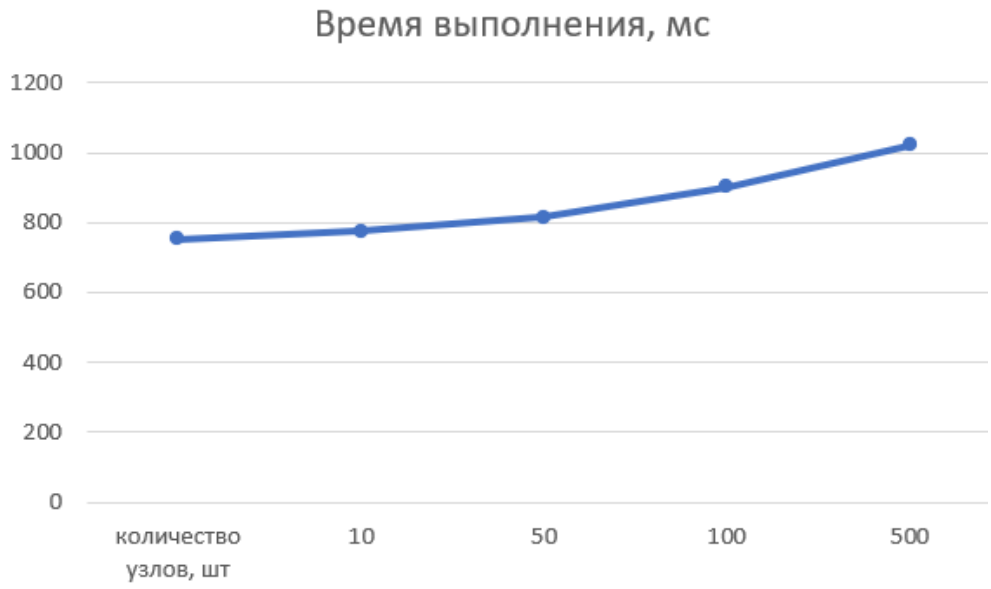
В операции find нужен доступ по ключу ($O(\log n)$) и splay ($O(\log n)$), следовательно, find выполняется за $O(\log n)$.

Однако преимущество данного дерева в том, что если снова обращаться к одним и тем же элементам, будет быстрее. Например, если во второй раз (и далее) обратиться к элементу, доступ к нему будет как $O(const)$

В операции add нужен доступ по ключу, splay и сливание деревьев, происходящее за константное время, следовательно, add выполняется за $O(\log n)$.

Операция del выполняется практически также (в плане операций), как и add и также имеет скорость $O(\log n)$.

АНАЛИЗ СКОРОСТИ ПРИ РАЗЛИЧНЫХ ВХОДНЫХ ДАННЫХ



Как можно видеть на графике, увеличение времени выполнения происходит по логарифмическому закону

СПИСОК ИСТОЧНИКОВ

1. Sleator, Daniel D.; Tarjan, Robert E."Self-Adjusting Binary Search Trees"
2. https://ru.wikipedia.org/wiki/Splay-дерево
3. https://habr.com/ru/company/JetBrains-education/blog/210296/
4. https://www.youtube.com/watch?v=zvZEFqxmgOY
5. https://www.geeksforgeeks.org/splay-tree-set-1-insert/

Код программы:

```
#include<iostream>
#include <string>

std::string FindError = "";
std::string NoRootError = "";

class Node {
public:
  int key;
  std::string data;
public:
  Node* left;
  Node* right;
  Node* parent;
public:
  Node();
  Node(const int key, const std::string data);
  ~Node();
};

class Tree {
private:
  Node* root;
  void rotate_left(Node* current);
  void rotate_right(Node* current);
  void zig();
  void splay(Node* current);
  Node* getmin(Node* p) { return p->left ? getmin(p->left) : p; };
public:
  bool add(const int key, const std::string data);  // false if key already exists
  bool del(const int key);  // false if no key
  std::string find(const int key);  // return '' if no key
public:
  Tree();
  ~Tree();
};

Node::Node() {
  left = nullptr;
  right = nullptr;
}

Node::Node(int key, std::string data) {
  this->key = key;
```

```
  this->data = data;
  left = nullptr;
  right = nullptr;
  parent = nullptr;
}

Tree::Tree() {
  root = nullptr;
}

void rotate_left(Node* current) {
  Node* p = current->parent;
  Node* r = current->right;
  if (p != nullptr) {
    if (p->left == current)
      p->left = r;
    else
      p->right = r;
  }
  Node* tmp = r->left;
  r->left = current;
  current->right = tmp;
  current->parent = r;
  r->parent = p;
  if (current->right != nullptr)
    current->right->parent = current;
}

void rotate_right(Node* current) {
  Node* p = current->parent;
  Node* l = current->right;
  if (p != nullptr) {
    if (p->left == current)
      p->left = l;
    else
      p->right = l;
  }
  Node* tmp = l->right;
  l->right = current;
  current->left = tmp;
  current->parent = l;
  l->parent = p;
  if (current->left != nullptr)
    current->left->parent = current;
}

bool zig(Node* current) {
```

```
  if (current->parent->left == current)
    rotate_right(current->parent);
  else
    rotate_left(current->parent);
}

void splay(Node* current) {
  while (current->parent != nullptr) {
    Node* grandp = current->parent->parent;
    if (grandp == nullptr)
      zig(current->parent);
    if (grandp->left == current->parent) {
      if (current->parent->left == current) {
        rotate_right(current->parent);
        rotate_right(grandp);
      }
      else {
        rotate_left(current->parent);
        rotate_right(grandp);
      }
    }
    else {
      if (current->parent->left == current) {
        rotate_right(current->parent);
        rotate_left(grandp);
      }
      else {
        rotate_left(current->parent);
        rotate_left(grandp);
      }
    }
  }
}

std::string Tree::find(const int key) {
  Node* current = root;
  if (current == nullptr)
    return NoRootError;
  while (key != current->key) {
    if (key > current->key) {
      if (current->left == nullptr) return FindError;
      current = current->left;
    }
    else {
      if (current->right == nullptr) return FindError;
      current = current->right;
    }
```

```
  }
  splay(current);
  root = current;
  return root->data;

}

bool Tree::add(const int key, const std::string data) {
  Node* current = root;
  while (true) {
    if (key > current->key) {
      if (current->left == nullptr) {
        current->left->key = key;
        current->left->data = data;
        current->left->parent = current;
        break;
      }
      current = current->left;
    }
    else {
      if (current->right == nullptr) {
        current->right->key = key;
        current->right->data = data;
        current->right->parent = current;
        break;
      }
      current = current->right;
    }
    splay(current);
  }
}

bool Tree::del(const int key) {
  if (root == nullptr)
    return false;
  if (root->key == key) {
    root = nullptr;
    return true;
  }
  Node* right_tree = nullptr;
  Node* left_tree = nullptr;
  find(key);
  right_tree = root->right;
  left_tree = root->left;
  splay(getmin(left_tree));
  root = left_tree;
  root->left = left_tree->left;
```

```
  root->right = right_tree;
}



Tree::~Tree() {
  delete root;
}

Node::~Node() {
  delete left;
  delete right;
  delete parent;
}

int main() {
  std::cout << "The end." << std::endl;
}
```