МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»
РТУ МИРЭА

Институт ИКБ

Специальность (направление): 09.03.02 (информационные системы и технологии)

Кафедра: КБ-3 «Разработка программных решений и системного программирования»

Дисциплина: «Алгоритмы и структуры данных»

Практическая работа
на тему:
Программа по деревьям

Студент: _____ 15.12.2024 Хвальчев С.П.
подпись *Дата* *инициалы и фамилия*

Группа: БСБО-16-23 Шифр: 23Б0182

Преподаватель: _____ 15.12.2024 Филатов В.В.
подпись *дата* *инициалы и фамилия*

Москва 2024 г.

1. Задание (вариант 82)

82.	Рандомизированное дерево двоичного поиска	Левый сын, правый брат (таблица, массив)	A=A $U_{обр}$B	обратный обход

2. Термины

Узел дерева — элемент структуры дерева, содержащий данные и указатели на связанные узлы.

Левый сын — указатель на первый дочерний узел данного узла.

Правый брат — указатель на следующий узел того же уровня, связанный с текущим.

Корень дерева — вершина дерева, не имеющая родительских узлов.

Обратный обход — метод обхода дерева, при котором сначала обрабатываются дочерние узлы, а затем текущий узел.

Описание программы

Основная цель программы

Программа предназначена для работы с рандомизированным деревом двоичного поиска (RandTree). Она предоставляет инструменты для вставки узлов, модификации структуры дерева (включая вращения), проверки корректности указателей и выполнения обратного обхода дерева.

Структуры и классы

Node: Узел дерева, содержащий:

data — данные, хранимые в узле.

parent — указатель на родительский узел.

left_child — указатель на первого сына.

right_sibling — указатель на следующего брата.

RandTree: Основной класс дерева, содержащий:

root — указатель на корень дерева.

Алгоритмы

Вставка узлов:

Метод insert() добавляет новый узел с указанными данными, устанавливая родительский и братский указатели в соответствии с правилами структуры дерева.

Проверка корректности дерева:

Метод validateTree() проверяет корректность всех указателей дерева:

Указатели на родителей.

Указатели на сыновей и братьев.

Обратный обход дерева:

Реализация метода reverseTraversal(), который выполняет обход в порядке: все дочерние узлы -> текущий узел.

Используется для вывода структуры дерева или выполнения операций над узлами.

Разрезы дерева:

Реализация метода findCuts() для поиска разрезов дерева. Разрезы определяются как множества узлов, разделяющие дерево на две независимые части.

Основные методы

insert(data):

Добавляет узел с заданными данными.

Устанавливает корректные указатели на родителей, сыновей и братьев.

reverseTraversal(node):

Рекурсивно выполняет обратный обход дерева, начиная с указанного узла.

validateTree():

Проверяет корректность структуры дерева:

Все указатели на родителей должны быть верными.

Сыновья и братья узлов не должны образовывать циклов.

findCuts():

Анализирует структуру дерева для поиска возможных разрезов.

Выводит информацию о частях дерева, полученных после разреза.

Общий процесс работы программы

Пользователь создаёт дерево, добавляя узлы через метод insert().

Программа отображает текущую структуру дерева через обход (reverseTraversal()).

Пользователь может запустить проверку структуры дерева, используя validateTree().

Для анализа разрезов дерева используется метод findCuts(), который выводит результаты проверки.

4. Скриншот работы программы

```
int main()
{
    srand(time(0));
    RandTree tree,tree1;

    tree.root = tree.insert(tree.root, 1);

    tree.root = tree.insert(tree.root, 2);

    tree.root = tree.insert(tree.root, 3);

    tree.root = tree.insert(tree.root, 4);

    tree.root = tree.insert(tree.root, 5);

    tree.root = tree.insert(tree.root, 6);

    tree.handleRootSibling(tree.root);

    tree.printTree(tree.root);
    std::cout<<"\n";
    tree1.root = tree1.insert(tree1.root, 11);
    tree1.root = tree1.insert(tree1.root, 7);
    tree1.root = tree1.insert(tree1.root, 6);
    tree1.root = tree1.insert(tree1.root, 8);
    tree1.root = tree1.insert(tree1.root, 9);
    tree1.handleRootSibling(tree1.root);
    tree1.printTree(tree1.root);
    std::cout << "\nСтруктура дерева после вставки:" << std::endl;
    tree.myOperation(tree,tree1);
    tree1.handleRootSibling(tree1.root);
    tree.printTree(tree.root);


    return 0;
}
```

```
[Key: 6, Size: 3]
  [Key: 4, Size: 2]
    [Key: 5, Size: 1]
      [Key: 3, Size: 3]
        [Key: 2, Size: 2]
          [Key: 1, Size: 1]

[Key: 11, Size: 5]
  [Key: 8, Size: 4]
    [Key: 7, Size: 2]
      [Key: 6, Size: 1]
  [Key: 9, Size: 1]

Структура дерева после вставки:
[Key: 11, Size: 8]
  [Key: 8, Size: 7]
    [Key: 7, Size: 5]
      [Key: 6, Size: 4]
        [Key: 4, Size: 2]
          [Key: 5, Size: 1]
            [Key: 3, Size: 3]
              [Key: 2, Size: 2]
                [Key: 1, Size: 1]
      [Key: 6, Size: 1]
  [Key: 9, Size: 1]
```

5. Исходный код

```
#include <iostream>

#include <cstdlib>

#include <functional>

#include <stack>

#include <unordered_set>

#include <ctime>

class RandTree

{

public:

   class node

   {

   public:

      node *bro;    // Указатель на "правого брата"

      node *son;    // Указатель на "первого сына"
```

```
      node *parent; // Указатель на родителя
      int key;      // Значение ключа
      size_t size;  // Размер поддерева, включая этот узел

      node(int key) : bro(nullptr), son(nullptr), parent(nullptr), key(key),
size(1) {}
   };

   class Iterator
   {
   public:
      const node *current;
      std::stack<const node *> nodes;

      Iterator(const node *root) : current(nullptr)
      {
         if (root)
            nodes.push(root);
         moveToNext();
      }

      int operator*() { return current->key; }

      Iterator &operator++()
      {
         moveToNext();
         return *this;
```

```
    }


    bool operator!=(const Iterator &other) const { return current !=
other.current; }


  private:
    void moveToNext()
    {
       while (!nodes.empty())
       {
          const node *node = nodes.top();
          nodes.pop();


          if (node)
          {
             current = node;


             if (node->bro)
                nodes.push(node->bro);


             if (node->son)
                nodes.push(node->son);


             return;
          }
       }
       current = nullptr;
```

```
    }
};

// Методы для работы с деревом
Iterator begin() const { return Iterator(root); }
Iterator end() const { return Iterator(nullptr); }

node *root;

RandTree() : root(nullptr) {}

node *PARENT(node *n) { return n ? n->parent : nullptr; }

node *LEFT_CHILD(node *n) { return n ? n->son : nullptr; }

node *RIGHT_SIBLING(node *n) { return n ? n->bro : nullptr; }

int LABEL(node *n) { return n ? n->key : (int)NULL; }

node *ROOT() { return root; }

node *CREATE(node *n, RandTree &T1, RandTree &T2)
{
   if (T1.ROOT())
      T1.ROOT()->parent = n;
```

```
    if (T2.ROOT())
    {
        T2.ROOT()->parent = n;
        if (T1.ROOT())
        {
            T1.ROOT()->bro = T2.ROOT();
        }
    }
    root = n;
    if (T1.ROOT())
    {
        n->son = T1.ROOT();
    }
    if (T2.ROOT())
    {
        if (T1.ROOT())
        {
            T1.ROOT()->bro = T2.ROOT();
        }
        else
        {
            n->son = T2.ROOT();
        }
    }
    T1.root = nullptr;
    T2.root = nullptr;
    return n;
}
```

```
void updateSizesToRoot(node *n)
{
   while (n)
   {
      fixsize(n);
      n = n->parent;
   }
}


void deleteNode(node *n)
{
   if (!n)
      return;


   if (n->son)
   {
      deleteNode(n->son);
      n->son = nullptr;
   }


   if (n->bro)
   {
      deleteNode(n->bro);
      n->bro = nullptr;
   }


   delete n;
}
```

```
void MAKENULL()
{
   if (root)
   {
      deleteNode(root);
      root = nullptr;
   }
}


void fixsize(node *n)
{
   if (!n)
      return;
   n->size = 1;
   if (n->son)
      n->size += n->son->size;
   if (n->bro)
      n->size += n->bro->size;
}
node *insert(node *root, int key)
{
   if (!root)
   {
      return new node(key);
   }
   handleRootSibling(root);
```

```
    if (rand() % (root->size + 1) == 0)
    {
        root = insertroot(root, key);
    }
    else
    {
        if (key < root->key)
        {
            root->son = insert(root->son, key);
            root->son->parent = root;
        }
        else
        {
            node *bro = insert(root->bro, key);
            bro->parent = root->parent;
            root->bro = bro;
        }
        fixsize(root);
    }


    return root;
}
node *insertroot(node *root, int key)
{
    if (!root)
    {
        return new node(key);
    }
```

```
    if (key < root->key)
    {
       root->son = insertroot(root->son, key);
       root->son->parent = root;
    }
    else
    {
       root->bro = insertroot(root->bro, key);
       root->bro->parent = root;
       root = rotateleft(root);
    }
    if (root->bro)
    {
       node *bro = root->bro;
       root->bro = nullptr;
       if (rand() % 2 == 0)
       {
          bro->parent = root;
          if (root->son)
          {
             bro->son = root->son;
             root->son->parent = bro;
          }
          root->son = bro;
       }
       else
       {
          bro->parent = root;
          bro->son = nullptr;
          root->bro = bro;
```

```
        }
    }


    fixsize(root);
    return root;
}


node *rotateleft(node *x)
{
    if (!x || !x->bro)
    {
        return x;
    }
    node *y = x->bro;
    x->bro = y->son;
    if (y->son)
        y->son->parent = x;


    y->son = x;
    y->parent = x->parent;
    x->parent = y;


    if (!y->parent)
    {
        root = y;
    }
```

```
    fixsize(x);
    fixsize(y);


    return y;
}


void rotateright(node *&root)
{
    node *newRoot = root->son;
    root->son = newRoot->bro;
    if (newRoot->bro)
    {
        newRoot->bro->parent = root;
    }
    newRoot->bro = root;
    newRoot->parent = root->parent;
    root->parent = newRoot;
    root = newRoot;
    fixsize(root->bro);
    fixsize(root);
}


void printTree(node *root, int depth = 0)
{


    if (!root)
        return;
    for (int i = 0; i < depth; ++i)
```

```
    {
        std::cout << "   ";
    }
    std::cout << "[Key: " << root->key << ", Size: " << root->size << "]\n";
    if (root->son)
    {
        printTree(root->son, depth + 1);
    }
    if (root->bro)
    {
        printTree(root->bro, depth);
    }
}


bool validateTree(node *root, node *parent = nullptr)
{
    if (!root)
        return true;


    if (root == this->root && root->bro)
    {
        return false;
    }


    if (root->parent != parent)
    {
        return false;
    }
```

```
    return validateTree(root->son, root) && validateTree(root->bro, parent);
}


void addbro(node *current, node *bro)
{
    if (!current || !bro)
        return;
    while (current->bro)
    {
        current = current->bro;
    }
    current->bro = bro;
    bro->parent = current->parent;
}


void handleRootSibling(node *root)
{
    if (!root || !root->bro)
        return;


    node *sibling = root->bro;
    root->bro = nullptr;


    if (rand() % 2 == 0)
    {
        sibling->parent = root;
```

```
        sibling->bro = root->son;
        if (root->son)
            root->son->parent = sibling;
        root->son = sibling;
    }
    else
    {
        sibling->parent = root;
        sibling->bro = nullptr;
        if (root->son)
            sibling->son = root->son;
        if (sibling->son)
            sibling->son->parent = sibling;
        root->son = sibling;
    }

    fixsize(root);
}

bool checkTree(node *root)
{
    if (!root)
        return true;
    if (root->son && root->son->parent != root)
    {
        return false;
    }
    if (root->bro && root->bro->parent != root->parent)
    {
```

```
        return false;
    }
    return checkTree(root->son) && checkTree(root->bro);
}

static void myOperation(RandTree &A, const RandTree &B)
{
    for (auto it = B.begin(); it != B.end(); ++it)
    {
        A.insert(A.root, *it);
    }
}
};

int main()
{
    srand(time(0));
    RandTree tree,tree1;

    tree.root = tree.insert(tree.root, 1);

    tree.root = tree.insert(tree.root, 2);

    tree.root = tree.insert(tree.root, 3);

    tree.root = tree.insert(tree.root, 4);
```

```
    tree.root = tree.insert(tree.root, 5);


    tree.root = tree.insert(tree.root, 6);


    tree.handleRootSibling(tree.root);


    tree.printTree(tree.root);
    std::cout<<"\n";
    tree1.root = tree1.insert(tree1.root, 11);
    tree1.root = tree1.insert(tree1.root, 7);
    tree1.root = tree1.insert(tree1.root, 6);
    tree1.root = tree1.insert(tree1.root, 8);
    tree1.root = tree1.insert(tree1.root, 9);
    tree1.handleRootSibling(tree1.root);
    tree1.printTree(tree1.root);
    std::cout << "\nСтруктура дерева после вставки:" << std::endl;
    tree.myOperation(tree,tree1);
    tree1.handleRootSibling(tree1.root);
    tree.printTree(tree.root);



    return 0;
}
```

Вывод

В процессе реализации программы для работы с рандомизированным деревом двоичного поиска я значительно углубил свои знания в области структур данных и алгоритмов. Я освоил представление деревьев с использованием указателей на левый сын и правый брат, что позволило эффективно организовать структуру данных и обеспечить корректность работы с узлами. Также я изучил методы проверки корректности указателей дерева и реализовал алгоритмы обхода, включая обратный обход, для анализа структуры дерева. Работа с рандомизацией дерева позволила мне улучшить понимание оптимизации операций вставки и поиска.

Литература

Кормен Т. Х., Лейзерсон Ч. Е., Ривест Р. L., Штайн К. — *Введение в алгоритмы*, стр. 539, Глава 12. Деревья поиска.

Хиршберг Д. С., Чьенг В. В. — *Деревья и алгоритмы на них*, стр. 186, Глава 7. Алгоритмы обхода и модификации деревьев.

Лекции и практики – преподаватель Филатов В. В.