МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарные деревья поиска

Студент гр. 9382 _____ Павлов Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель

Построение БДП и выполнение над ним операций, указанных в задании.

Основные теоретические сведения

Двоичное дерево поиска — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше либо равны, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Декартово дерево - это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) и дерамида (дерево+пирамида).

Более строго, это структура данных, которая хранит пары (X,Y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y. Предполагая, что все X и все Y являются различными, получаем, что если некоторый элемент дерева содержит (X_0,Y_0), то у всех элементов в левом поддереве $X < X_0$, у всех элементов в правом поддереве $X > X_0$, а также и в левом, и в правом поддереве имеем: $Y < Y_0$.

Задание

14) БДП: Рандомизированная дерамида поиска (treap); действие: 1+2в

Алгоритм

Создание и работа со списком базируются на двух основных операциях, применимых к дерамиде: **merge** и **split**

1. **merge** – функция слияния двух дерамид. Деревья корректно сливаются только при условии, что две поданных на вход дерамиды расположены таким образом, что все ключи второй дерамиды больше, чем все ключи первой. Сравниваются значения (приоритеты) листьев, а не их ключи. В зависимости от этого в корень устанавливается текущий корневой элемент либо правой, либо левой ветви, а другая ветвь «скользит» по правой/левой ветви нового корня до тех пор, пока её корневой элемент не станет больше по приоритету, чем элемент ветви, по которой он «скользит». Возвращает указатель на полученное в результате обработки дерево.
2. **split** – функция разбиения дерамиды на два поддерева по ключу. В данной функции осуществляется поиск элемента по ключу, возвращается пара указателей — на дерево, в котором все элементы по ключу меньше, чем искомое значение, и дерево, в котором содержится (если существует в дереве вообще) элемент с таким ключом и все элементы с ключом большим, чем искомый. На самом деле дерево не разбивается (указатели не удаляются), лишь осуществляется поиск подходящих указателей.

Добавление элемента осуществляется разбиением дерева по указанному ключу и слиянием левого поддерева с добавляемым элементом, а затем — слиянием результата с правым поддеревом.

Удаление элемента осуществляется слиянием поддеревьев элемента с указанным ключом (если он есть, если нет — поддеревьев с элементами меньше и больше него соответственно) и креплением результата к меньшему (по ключу) элементу, идущему перед ним. При этом освобождается память, выделенная под предыдущий элемент с ткаим ключом, если он до этого существовал.

Функции и СД

- **template <typename T> struct Pair {**

  T x;
  T y;

  Pair(T x, T y) : x(x), y(y) {}

  }; –

  структура из пары значений (здесь используется либо пара значений **int**, либо пара указателей на **Node**)

- **struct Node {**

  Pair<int> p;
  Node* l, * r;

  Node(int x, int y) : p(Pair<int>(x, y)), l(nullptr), r(nullptr) {}
  Node(int x, int y, Node* l, Node* r) : p(Pair<int>(x, y)), l(l), r(r) {}

  }; –

  Структура узла дерамиды. Содержит пару целочисленных значений и указатели на левое и правое поддеревья, а также два конструктора.

- **void indent(int n)** – вывод отступа, соответствующего глубине рекурсии; n – глубина рекурсии
- **template <typename T> void writeToFile(const string filename, const T arg)** — запись в файл данных типа T; filename – имя файла для записи, arg – данные, которые будут записаны
- **void dispose(Node* t) —** освобождение памяти от данных дерамиды; t – указатель на корень дерева
- **bool isIn(Node* t, int k)** — проверка на вхождение элемента с указанным ключом в дерамиду; t - указатель на корень дерева, k – ключ искомого элемента
- **Node* merge(Node* l, Node* r, int& depth)** — слияние двух поддеревьев; l – указатель на «левое» поддерево, r - указатель на «правое» поддерево, depth – ссылка на счётчик, отслеживающий глубину

- **Pair<Node*> split(Node* t, int k, int& depth) —** разбиение дерева по ключу; t – указатель на корень дерева, k – ключ, по которому разбивается дерамида, depth – сслыка на счётчик, отслеживающий глубину
- **Node* add(Node* t, int k, int v)** — добавление элемента в дерамиду; t – указатель на корень дерева, k – ключ добавляемого элемента, v – значение добавляемого элемента
- **Node* remove(Node* t, int k, int& depth)** – удаление элемента из дерамиды; t – указатель на корень дерева, k – ключ удаляемого элемента, depth – сслыка на счётчик, отслеживающий глубину
- **void printTreap(Node* t, int& depth)** – вывод дерамиды на экран в наглядной форме; t – указатель на корень дерева, depth – сслыка на счётчик, отслеживающий глубину
- **void printAscendingToFile(Node* t, const string& filename) —** вывод элементов дерамиды в возрастающем порядке в файл; t – указатель на корень дерева, filename – имя файла, в который выводятся элементы
- **int main()** :
  - Файловые ввод/вывод
  - Консольные ввод/вывод
  - Вывод промежуточных/итоговых результатов

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1

Входные данные	**Выходные данные**
1 d	
7 9 -5 11 10 12 -1 9 0 1 2 13 11 2	-5 11 -1 9 0 1 2 13 7 9 10 12 11 2
7 9 -1 9 0 1 6 r -5 11	-1 9 0 1 7 9

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

```
#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

template <typename T> struct Pair {
	T x;
	T y;

	Pair(T x, T y) : x(x), y(y) {}
};

struct Node {
	Pair<int> p;
	Node* l, * r;

	Node(int x, int y) : p(Pair<int>(x, y)), l(nullptr), r(nullptr) {}
	Node(int x, int y, Node* l, Node* r) : p(Pair<int>(x, y)), l(l), r(r) {}
};

template <typename T>
void writeToFile(const string filename, const T arg) {
	ofstream output;
	output.open(filename, ios::app);
	output << arg;
	output.close();
}

void indent(int n) {
	for (int i = 0; i < n; i++) {
		cout << "\t";
	}
}

void dispose(Node* t) {
	if (t == nullptr) {
		return;
	}
	dispose(t->l);
	dispose(t->r);
	delete t;
}

bool isIn(Node* t, int k) {
	if (t != nullptr) {
		if (k > t->p.x) {
			return isIn(t->r, k);
		}
		else if (k < t->p.x) {
			return isIn(t->l, k);
		}
		return true;
	}
	return false;
}

Node* merge(Node* l, Node* r, int& depth) {
	if (!depth) {
```

```
            cout << "\nAttempting to merge trees\n";
      }
      if (l == nullptr) {
            return r;
      }
      if (r == nullptr) {
            return l;
      }

      if (l->p.y > r->p.y) {
            indent(depth);
            cout << "Merging with the right branch of  " << l->p.x << "  " << l-
>p.y << "\n";
            depth++;
            l->r = merge(l->r, r, depth);
            depth--;
            return l;
      }
      indent(depth);
      cout << "Merging with the left branch of  " << r->p.x << "  " << r->p.y <<
"\n";
      depth++;
      r->l = merge(l, r->l, depth);
      depth--;
      return r;
}

Pair<Node*> split(Node* t, int k, int& depth) {
      if (!depth) {
            cout << "\nAttempting to split the tree.\n";
      }
      if (t == nullptr) {
            return Pair<Node*>(nullptr, nullptr);
      }
      else if (k > t->p.x) {
            indent(depth);
            cout << "Descending to  the right branch of  " << t->p.x << "  " <<
t->p.y << "\n";
            depth++;
            Pair<Node*> nPair = split(t->r, k, depth);
            depth++;
            t->r = nPair.x;
            return Pair<Node*>(t, nPair.y);
      }
      else{
            indent(depth);
            cout << "Descending to  the left branch of  " << t->p.x << "  " <<
t->p.y << "\n";
            depth++;
            Pair<Node*> nPair = split(t->l, k, depth);
            depth--;
            t->l = nPair.y;
            return Pair<Node*>(nPair.x, t);
      }
      return Pair<Node*>(t, nullptr);
}

Node* add(Node* t, int k, int v) {
      int depth = 0;
      cout << "\n\nSplitting the treap by key " << k << "\n";
      Pair<Node*> p = split(t, k, depth);
      Node* n = new Node(k, v);
      return merge(merge(p.x, n, depth), p.y, depth);
```

```
}

Node* remove(Node* t, int k, int& depth) {
	if (!depth) {
		cout << "\nAttempting to remove existing element.\n";
	}
	if (t != nullptr) {
		indent(depth);
		cout << "Current element: " << t->p.x << "  " << t->p.y << "\n";
		if (t->p.x < k) {
			depth++;
			t->r = remove(t->r, k, depth);
			depth--;
			return t;
		}
		else if (t->p.x > k) {
			depth++;
			t->l = remove(t->l, k, depth);
			depth--;
			return t;
		}
		else {
			indent(depth);
			cout << "\nRequired element found: " << t->p.x << "  " << t-
>p.y << " , removing...\n";
			Node* l = t->l, * r = t->r;
			t->l = nullptr;
			t->r = nullptr;
			dispose(t);
			return merge(l, r, depth);
		}
	}
	return t;
}

void printTreap(Node* t, int& depth) {
	if (t != nullptr) {
		if (t->r != nullptr) {
			depth++;
			printTreap(t->r, depth);
			depth--;
		}
		indent(depth);
		cout << t->p.x << "  " << t->p.y << "\n";
		if (t->l != nullptr) {
			depth++;
			printTreap(t->l, depth);
			depth--;
		}
	}
}

void printAscendingToFile(Node* t, const string& filename) {
	if (t != nullptr) {
		if (t->l != nullptr) {
			printAscendingToFile(t->l, filename);
		}
		writeToFile(filename, t->p.x);
		writeToFile(filename, "  ");
		writeToFile(filename, t->p.y);
		writeToFile(filename,  '\n');
		if (t->r != nullptr) {
			printAscendingToFile(t->r, filename);
```

```
		}
	}
}

int main() {
	srand(time(nullptr));
	int command = 0;
	int key = 0, value = 0;
	int depth = 0;
	Node* treap = nullptr;
	cout << "Enter : 1 - Console input , 2 - File Input:\n\n";
	cin >> command;

	if (command == 2) {
		string input_filename;
		const string output_filename = "output.txt";
		ifstream in;
		ofstream out;

		out.open(output_filename);
		out << "";
		out.close();

		cout << "Enter the input file name: \n\n";
		cin >> input_filename;
		in.open(input_filename);

		if (in.is_open()) {
			while (!in.eof()) {
				if (!(in >> key) || !(in >> value)) {
					break;
				}
				if (isIn(treap, key)) {
					cout << "\n\nElement with key = " << key << "
already exists. Replacing...";
					treap = remove(treap, key, depth);
				}
				cout << "\n\nAdding new element with key = " << key << "
to the treap\n";
				treap = add(treap, key, value);
			}
			if (treap != nullptr) {
				printTreap(treap, depth);
				printAscendingToFile(treap, output_filename);
				cout << "\nCheck out the results in \"output.txt\"\n";
			}
			else {
				cout << "File is empty or contains inappropriate data.\
n";
			}
		}
		else {
			cout << input_filename << " doesn't exist!\n";
		}
	}
	else if (command == 1) {
		cout << "Enter the pairs (key then value, both integer) or anything
except that (to finish): \n\n";
		while ((cin >> key)) {
			if (!(cin >> value)) {
				break;
			}
			if (isIn(treap, key)) {
```

```
                        cout << "\n\nElement with key = " << key << " already
exists. Replacing...";
                        treap = remove(treap, key, depth);
                  }
                  cout << "\n\nAdding new element with key = " << key << " to
the treap\n";
                  treap = add(treap, key, value);
            }
            if (treap != nullptr) {
                  printTreap(treap, depth);
            }
            else {
                  cout << "The treap is empty.\n";
            }
      }
      else {
            cout << "\nUnknown command, program finished\n";
      }

      return 0;
}
```