Алгоритм решения задачи "10"

Антон Дроздовский

1 Условие задачи

Найти вершины, через которые проходит наибольшее число наибольших (по длине) полупутей, и удалить их (правым удалением) обратным левым обходом.

Input

Входной файл содержит последовательность чисел — ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Output

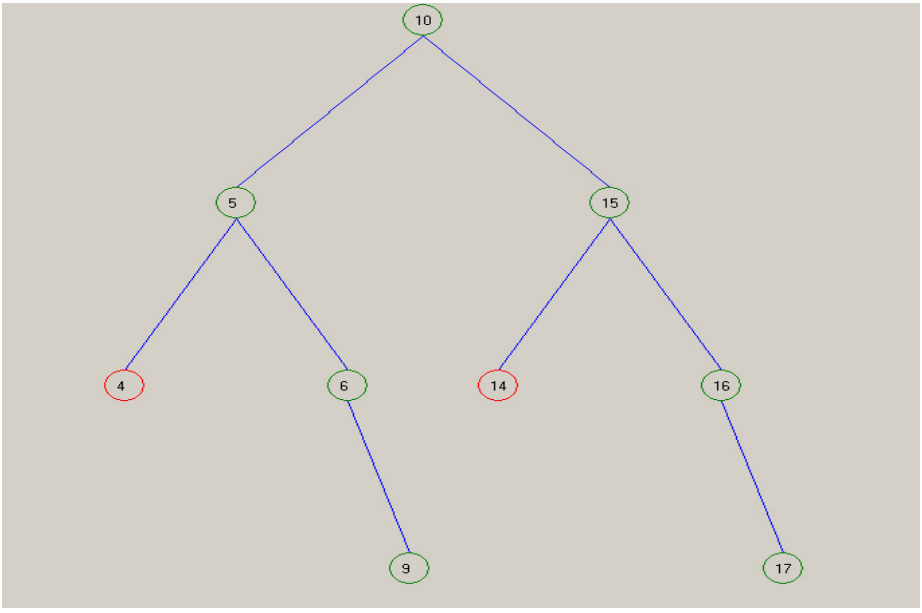
Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева. Гарантируется, что хотя бы одна вершина не подлежит удалению.

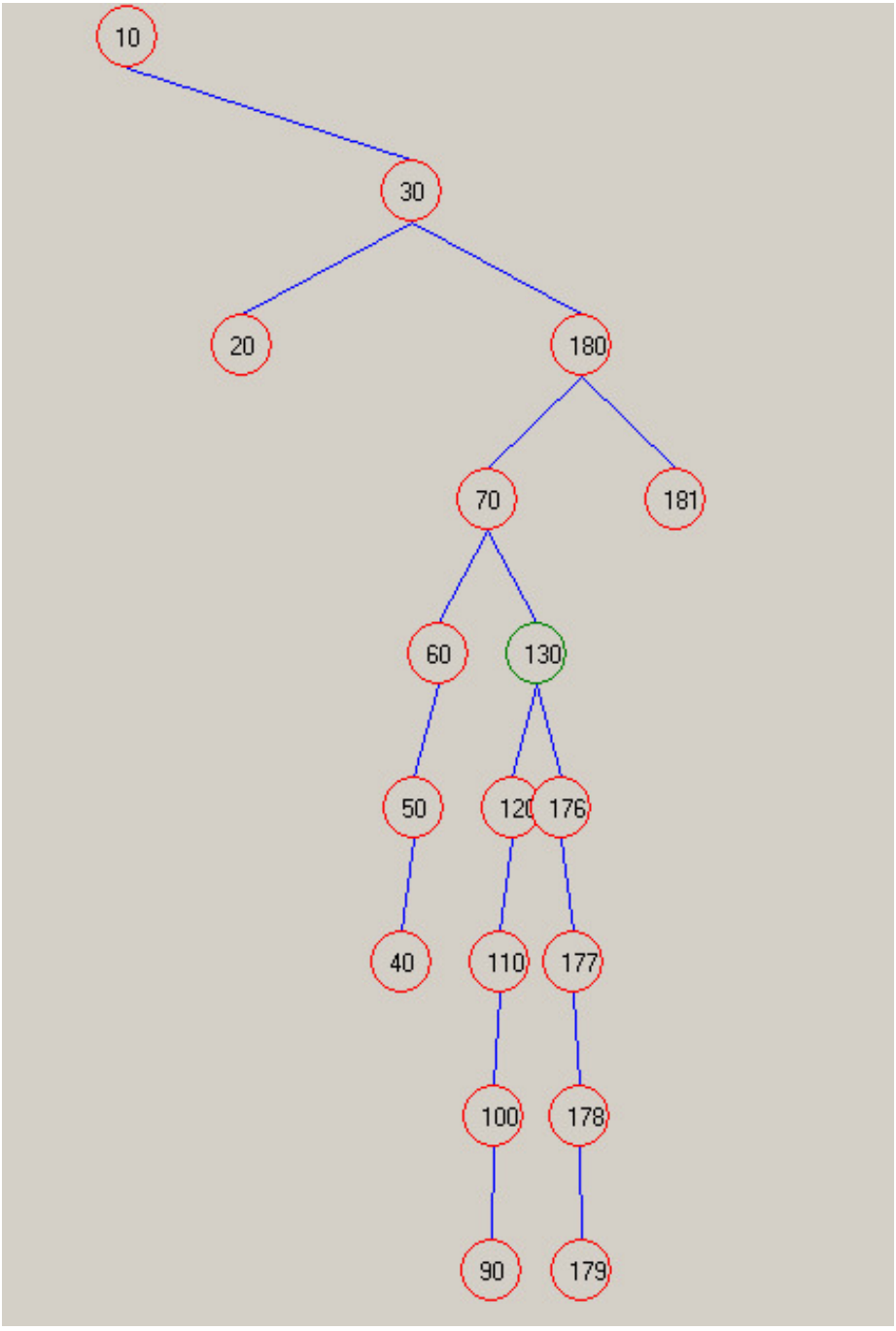
Note

Для удаления всех вершин, удовлетворяющих заданным требованиям, необходимо разработать алгоритм, работающий за время O(n).

Examples

in.txt	**out.txt**
10 5 15 4 6 14 16 17 9	14 4
10 30 180 181 20 70 60 50 40 130 176 177 178 179 120 110 100 90	10 30 20 180 70 60 50 40 176 120 110 100 90 177 178 179 181


10
5
15
4
6
14
16
9
17


10
30
20
180
70
181
60
130
50
120
176
40
110
177
100
178
90
179

2 Описание алгоритма

Первоначально строим дерево. Ищем корни наибольших полупутей. Для этого нужно найти те вершины, для которых сумма меток высот её поддеревьев, увеличенная на количество поддеревьев, является наибольшей. Это можно сделать за один обратный обход дерева. Среди всех корней наибольших полупутей наибольшее количество проходит через корень наименьшей высоты. Теперь нужно определить вершины для удаления. Рассмотрим два случая:

- Если корень один, то помечаем вершины для удаления у правого и левого поддерева. Если у некоторой вершины поддерева есть поддеревья, то продолжаем проход по большему по длине (достаточно сравнить высоты). Если длины у поддеревьев одинаковы, то останавливаем проход: через вершины поддеревьев проходит меньшее количество полупутей, чем через их корень.
- Если корней несколько, то берём корень наименьшой высоты и начинаем проход по наибольшему по длине поддереву и далее аналогично первому случаю. Если поддеревья корня наибольшего полупути по длине равны, то проход не нужен.

Проходим дерево обратным левым обходом и удаляем нужные вершины правым удалением. Для реализации нерекурсивного удаления каждая вершина должна знать своего предка. Выводим последовательность ключей, полученную прямым левым обходом итогового дерева.

3 Решение задачи на C++

```
#define _CRT_SECURE_NO_WARNINGS

#include <algorithm>
#include <cstdio>
#include <ios>
#include <iostream>

struct Node {
        int key;
        int height;
        Node* left;
        Node* right;
        Node* parent;
        bool lowestOne;
        bool toDelete;

        Node(int val, Node* father) : key(val), height(0),
            left(nullptr), right(nullptr), parent(father),
            lowestOne(false), toDelete(false) {}
```

```
};

int longestWay = -1;
Node* maxRoot = nullptr;
Node* top = nullptr;

int heights(Node* root) {
        if (root == nullptr) {
                return -1;
        }

        int leftHeight = heights(root->left);
        int rightHeight = heights(root->right);

        int isMaxRoot;
        if (root->left == nullptr && root->right == nullptr) {
                isMaxRoot = 0;
        }
        else if (root->left != nullptr && root->right != nullptr) {
                isMaxRoot = root->left->height + root->right->height + 2;
        }
        else {
                isMaxRoot = (root->left != nullptr ? root->left->height : root->right->height) + 1;
        }

        if (isMaxRoot > longestWay) {
                longestWay = isMaxRoot;
                maxRoot = root;
        }
        else if (isMaxRoot == longestWay) {
                maxRoot->lowestOne = true;
        }

        return root->height = std::max(leftHeight, rightHeight) + 1;
}

void defineNodesToDelete(Node* root) {
        while (true) {
                root->toDelete = true;
                if (root->left == nullptr && root->right == nullptr) {
                        break;
                }
                else if (root->left != nullptr && root->right != nullptr) {
```

```
61                          if (root->left->height == root->
                                right->height) {
62                                  break;
63                          }
64                          else {
65                                  root = root->left->height >
                                        root->right->height ?
                                        root->left : root->right;
66                          }
67                  }
68                  else {
69                          root = root->left == nullptr ? root
                                ->right : root->left;
70                  }
71          }
72  }
73
74  void remove(Node* root) {
75          if (root->left == nullptr && root->right == nullptr)
                {
76                  if (root->parent != nullptr) {
77                          if (root->parent->left == root) {
78                                  root->parent->left = nullptr
                                        ;
79                          }
80                          else {
81                                  root->parent->right =
                                        nullptr;
82                          }
83                  }
84
85                  return;
86          }
87
88          if (root->left == nullptr) {
89                  if (root->parent != nullptr) {
90                          if (root->parent->left == root) {
91                                  root->parent->left = root->
                                        right;
92                          }
93                          else {
94                                  root->parent->right = root->
                                        right;
95                          }
96                  }
97
98                  root->right->parent = root->parent;
99
100                 if (root->parent == nullptr) {
101                         top = root->right;
```

```
                }

                return;
        }

        if (root->right == nullptr) {
                if (root->parent != nullptr) {
                        if (root->parent->left == root) {
                                root->parent->left = root->
                                    left;
                        }
                        else {
                                root->parent->right = root->
                                    left;
                        }
                }

                root->left->parent = root->parent;

                if (root->parent == nullptr) {
                        top = root->left;
                }

                return;
        }

        Node* minRight = root->right;
        while (minRight->left != nullptr) {
                minRight = minRight->left;
        }

        root->key = minRight->key;
        remove(minRight);
}

void preorderTraversal(Node* root) {
        if (root == nullptr) {
                return;
        }

        std::cout << root->key << '\n';
        preorderTraversal(root->left);
        preorderTraversal(root->right);
}

void postorderTraversal(Node* root) {
        if (root == nullptr) {
                return;
        }

```

```
150         postorderTraversal(root->left);
151         postorderTraversal(root->right);
152         if (root->toDelete == true) {
153                 remove(root);
154         }
155 }
156
157 int main() {
158         std::ios_base::sync_with_stdio(false);
159         std::cin.tie(nullptr);
160
161         std::freopen("in.txt", "r", stdin);
162         std::freopen("out.txt", "w", stdout);
163
164         int key;
165         Node* root = nullptr;
166         std::cin >> key;
167         root = new Node(key, nullptr);
168         top = root;
169
170         Node* currNode = nullptr;
171         while (std::cin >> key) {
172                 currNode = root;
173                 while (true) {
174                         if (key < currNode->key) {
175                                 if (currNode->left ==
                                    nullptr) {
176                                         currNode->left = new
                                             Node(key,
                                            currNode);
177
178                                         break;
179                                 }
180                                 else {
181                                         currNode = currNode
                                             ->left;
182                                 }
183                         }
184                         else if (key > currNode->key) {
185                                 if (currNode->right ==
                                    nullptr) {
186                                         currNode->right =
                                             new Node(key,
                                            currNode);
187
188                                         break;
189                                 }
190                                 else {
191                                         currNode = currNode
                                             ->right;
```

```
192                                     }
193                             }
194                             else {
195                                     break;
196                             }
197                     }
198             }
199
200             heights(root);
201
202             maxRoot->toDelete = true;
203             if (maxRoot->lowestOne == true) {
204                     if (maxRoot->left->height > maxRoot->right->
                            height) {
205                             defineNodesToDelete(maxRoot->left);
206                     }
207                     else if (maxRoot->left->height < maxRoot->
                            right->height) {
208                             defineNodesToDelete(maxRoot->right);
209                     }
210             }
211             else {
212                     if (maxRoot->left != nullptr) {
213                             defineNodesToDelete(maxRoot->left);
214                     }
215
216                     if (maxRoot->right != nullptr) {
217                             defineNodesToDelete(maxRoot->right);
218                     }
219             }
220
221             postorderTraversal(root);
222
223             preorderTraversal(top);
224
225             return 0;
226 }
```

4 Оценка временной сложности

1. Построение дерева

$O(n^2)$

2. Нахождение корня наименьшей высоты наибольших полупутей

Выполняется за один обратный обход — $O(n)$.

3. Нахождение вершин, удовлетворяющих условию

По условию задачи гарантируется, что хотя бы одна вершина не подлежит удалению. Максимальное возможное количество вершин, которые нужно удалить, $n - 1$. $O(n)$

4. Удаление

Обратный левый обход. Если у корня два поддерева, то при поиске минимального ключа в правом поддереве проходится некоторая цепь. По такой цепи для нахождения нужного ключа проходим лишь один раз (через его вершины впоследующем проходить не придётся), так как попасть к этой цепи можно только с одной вершины, ключ ей заменили — она считается удалённой. $O(n)$.

5. Вывод последовательностей ключей

Прямой правый обход — $O(n)$.

5 Оценка сложости по памяти

При выполнении программы используется только построенное дерево поиска — $O(n)$. n — количество вершин в дереве