ИКН НИТУ МИСИС

Комбинаторика и теория графов

Деревья поиска, их представление в компьютере

Исполнитель:

Лосев В.Л. БИВТ-23-18

(https://github.com/SomethingWF/Bin_Search_tree)

Москва 2024 год

Определения

Дерево – связный, ациклический граф, обладающий следующими свойствами:

1. Конечный связный граф является деревом т. и т.т., когда число ребер (m) и число вершин в графе (n) связаны соотношением m = n - 1.
2. Граф является деревом т. и т.т. две его различные вершины можно соединить единственной простой цепью.
3. Не содержит кратных ребер и петель.
4. Любое дерево однозначно определяется расстояниями (длиной наименьшей цепи) между его концевыми (степени 1) вершинами.
5. Любое дерево является двудольным графом

Бинарное дерево – дерево, степени вершин которого не превосходят трех для неориентированных графов или двух для ориентированных.

В рамках данной работы было реализовано два вида деревьев поиска:

Двоичное дерево поиска – двоичное дерево, для которого выполняются следующие дополнительные условия:

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

АВЛ-дерево – сбалансированное по высоте двоичное дерево поиска, для каждой вершины которого высота двух ее поддеревьев различается не более чем на 1.

Вышеописанные структуры данных поддерживают следующие операции:

1. search(k) – поиск узла, в котором хранится пара (key, value) с key = k.
2. insert(k, v) – добавление в дерево пары (key, value) = (k, v).
3. remove(k) – удаление узла, в котором хранится пара (key, value) с key = k.

Перечень инструментов

Языком программирования, использовавшимся для реализации структур данных, хранящих графы, является C++. В частности, стандартная библиотека шаблонов(STL).

Описание реализации

Для реализации узлов бинарного дерева поиска была реализована структура Node, с полями ключа, значения, указателей на левое и правое поддеревья. Также в коде используются шаблоны, для обеспечения возможности создания деревьев с разными типами данных.

```
template <typename T>
struct Node {
  int key_;
  T value_;
  Node* left_ = nullptr;
  Node* right_ = nullptr;

  Node(int key, T value) : key_(key), value_(value), left_(nullptr), right_(nullptr) {}
};
```

Рис 1 – Поля узла бинарного дерева поиска

У самого же дерева поиска единственным полем является указатель на корневой узел дерева.

```
Node<T>* root_ = nullptr;
```

Рис 2 – Поле бинарного дерева поиска

Все операции в двоичных деревьях поиска были реализованы с помощью рекурсии. Поэтому непосредственная реализация рекурсивных функций находится в private-секции класса, а в методах, доступных пользователю, происходят вызовы этих самых функций с указанием корневого узла в качестве точки старта рекурсии.

```
int search(int key) {
  return search_pr(root_, key);
}
void insert(int key, T value) {
  ins(root_, key, value);
}
void remove(int key) {
  root_ = delete_pr(root_, key);
}
```

Рис 3 – Основные методы, доступные пользователям

Операция поиска элемента осуществляется путем сравнения ключа текущего узла с искомым. Если искомый ключ меньше – происходит вызов функции для левого потомка текущего узла, если больше – для правого. Также предусмотрен механизм выброса исключения в случае поиска несуществующего ключа.

```
T search_pr(Node<T>* node, int key) {
  if (node == nullptr) {
    throw std::runtime_error("Trying to search data in empty tree!");
  }
  if (node->key_ == key) return node->value_;
  return ((key < node->key_) ? search_pr(node->left_, key) : search_pr(node->right_, key))
}
```

Рис 4 – Реализация операции поиска элемента

Операция вставки элемента также основана на сравнении ключа текущего узла с искомым. Особенными случаями являются вставка элемента в пустое дерево и попытка вставки повторяющегося ключа, что вызовет исключение.

```
void ins(Node<T>* node, int key, T value) {
  if (node == nullptr) {
    root_ = new Node<T>(key, value);
  }
  else if (key < node->key_) {
    if (node->left_ == nullptr) node->left_ = new Node<T>(key, value);
    else ins(node->left_, key, value);
  }
  else if (key > node->key_) {
    if (node->right_ == nullptr) node->right_ = new Node<T>(key, value);
    else ins(node->right_, key, value);
  }
  else if (key = node->key_) {
    throw std::runtime_error("Trying to insert duplicating key!");
  }
}
```

Рис 5 – Реализация операции вставки элемента

Операция удаления помимо сравнения ключей и соответствующего ему рекурсивного вызова, рассматривает два сценария действий.

Первый – у удаляемого узла ноль или один потомок. Предварительно сохранив адрес памяти удаляемого узла, приоритетно выбирается потомок, содержащий какие-либо данные, после чего его адрес сохраняется в текущий узел, который будет передан функции, вызвавшей текущей, поддерживая тем самым связность дерева. После, ввиду особенностей программирования на C++, вручную удаляем искомый узел из динамической памяти с помощью предварительно сохраненного адреса.

Во втором случае у удаляемого узла в потомках имеется два поддерева. В рассмотренной реализации производится поиск узла с наибольшим ключом в левом поддереве. Его данные, в том числе ключ, передаются удаляемому узлу, а ранее найденный наибольший узел левого поддерева удаляется согласно первому сценарию.

```
Node<T>* delete_pr(Node<T>* node, int key) {
    if (node == nullptr) return nullptr;
    else if (key < node->key_) node->left_ = delete_pr(node->left_, key);
    else if (key > node->key_) node->right_ = delete_pr(node->right_, key);
    else {
        if (node->left_ == nullptr || node->right_ == nullptr) {
            Node<T>* temp = node;
            if (node->left_ == nullptr) node = node->right_;
            else node = node->left_;
            delete temp;
        }
        else {
            Node<T>* maxInLeft = getMax(node->left_);
            node->key_ = maxInLeft->key_;
            node->value_ = maxInLeft->value_;
            node->left_ = delete_pr(node->left_, maxInLeft->key_);
        }
    }
    return node;
}
```

Рис 6 – Реализация функции удаления элемента

Для реализации узлов в АВЛ-дереве используется структура Node с полями ключа, значения, указателей на левое и правое поддеревья, а также значения высоты узла. Высота узла представляет собой длину пути от текущего узла до самого низкого из поддеревьев. Таким образом высота несуществующего узла равна нулю, а узла без потомков единице.

```
template <typename T>  <T> Укажите аргументы примера шаблона для IntelliSense
struct Node
{
  int key_;
  T value_;
  Node<T>* left_;
  Node<T>* right_;
  int height_;

  Node (int key, T value) : key_(key), value_(value), left_(nullptr), right_(nullptr), height_(1) {}
};
```

Рис 7 – Поля узла АВЛ-дерева

У АВЛ-дерева единственное поле – указатель на корневой узел.

```
private:
  Node<T>* root_;
```

Рис 8 – Поле АВЛ-дерева

Принцип действия и код функции поиска элемента в АВЛ-дереве полностью идентичны таковой из бинарного дерева поиска.

Функция вставки элемента аналогична таковой из бинарного дерева поиска, за тем исключением, что в АВЛ-дереве после вставки, при скрутке стека вызовов, производится пересчет высот узлов и ребалансировка дерева.

```
Node<T>* insert_pr(Node<T>* node, int key, T value) {
  if (node == nullptr) {
    return new Node(key, value);
  }
  if (key < node->key_) node->left_ = insert_pr(node->left_, key, value);
  else if (key > node->key_) node->right_ = insert_pr(node->right_, key, value);
  else throw std::runtime_error("Trying to insert duplicating key!");

  node->height_ = std::max(height(node->left_), height(node->right_)) + 1;

  return balance(node);
}
```

Рис 9 – Реализация функции вставки нового элемента в АВЛ-дерево

Принцип работы операции удаления элемента также не отличается от таковой из бинарного дерево поиска, кроме вышеупомянутых пересчета высот узлов и балансировки дерева.

```
Node<T>* delete_pr(Node<T>* node, int key) {
  if (node == nullptr) return nullptr;
  else if (key < node->key_) node->left_ = delete_pr(node->left_, key);
  else if (key > node->key_) node->right_ = delete_pr(node->right_, key);
  else {
    if (node->left_ == nullptr || node->right_ == nullptr) {
      Node<T>* temp = node;
      if (node->left_ == nullptr) node = node->right_;
      else node = node->left_;
      delete temp;
    }
    else {
      Node<T>* minInRight = getMin(node->right_);
      node->key_ = minInRight->key_;
      node->value_ = minInRight->value_;
      node->right_ = delete_pr(node->left_, minInRight->key_);
    }
  }
  if (node == nullptr) return node;
  node->height_ = std::max(height(node->left_), height(node->right_)) + 1;

  return balance(node);
}
```

Рис 10 – Реализация функции удаления элемента в АВЛ-дерево

Указанные выше операции используют вспомогательные методы, недоступные пользователю реализованных структур данных, рассмотрим их.

Операции поиска максимального и минимального элемента в дереве основываются на ключевом свойстве деревьев поиска, которое гарантирует, что наибольший и наименьший элементы находятся в самом правом и левом узлах соответственно.

```
Node<T>* getMin(Node<T>* node) {
  if (node == nullptr) throw std::runtime_error("Trying to get minimum key in empty tree!");
  if (node->left_ == nullptr) return node;
  return getMin(node->left_);
}

Node<T>* getMax(Node<T>* node) {
  if (node == nullptr) throw std::runtime_error("Trying to get maximum key in empty tree!");
  if (node->right_ == nullptr) return node;
  return getMax(node->right_);
}
```

Рис 11 – Реализация функций поиска минимального и максимального элементов поддерева

Операция вычисления высоты текущего узла возвращает разницу между высотой левого и правого дочерних узлов.

```
int balanceFactor(Node<T>* node) {
  if (node == nullptr) {
    return 0;
  }
  return height(node->left_) - height(node->right_);
}
```

Рис 12 – Реализация функции вычисления высоты текущего узла

Операции левого и правого вращения необходимы для балансировки дерева и совершают преобразование поддеревьев, как продемонстрировано на рисунке.

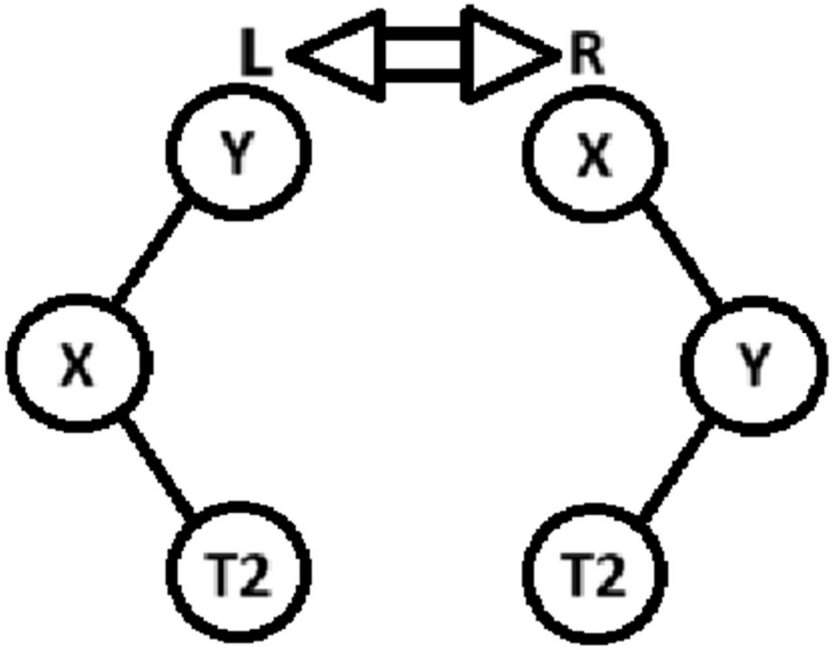

Рис 13 – Схема работы левого и правого вращений

```
Node<T>* rRotate(Node<T>* y) {
  Node<T>* x = y->left_;
  Node<T>* t2 = x->right_;

  x->right_ = y;
  y->left_ = t2;

  y->height_ = std::max(height(y->left_), height(y->right_)) + 1;
  x->height_ = std::max(height(x->left_), height(x->right_)) + 1;

  return x;
}
```

Рис 14 – Реализация функции правого вращения

```
Node<T>* lRotate(Node<T>* x) {
    Node<T>* y = x->right_;
    Node<T>* t2 = y->left_;

    y->left_ = x;
    x->right_ = t2;

    x->height_ = std::max(height(x->left_), height(x->right_)) + 1;
    y->height_ = std::max(height(y->left_), height(y->right_)) + 1;

    return y;
}
```

Рис 15 – Реализация функции левого вращения

Непосредственная операция балансировки рассматривает 4 сценария, в соответствии с которыми производит следующие комбинации вращений:

1. Влево влево. Если у текущего узла и его левого потомка наблюдается перевес в левую сторону производится правое вращение текущего потомка.
2. Влево вправо. Если у текущего узла наблюдается перевес влево, а у его левого потомка – вправо, то сначала производится левое вращение левого потомка, а затем правое вращение текущего узла.
3. Вправо вправо. Если у текущего узла и его правого потомка наблюдается перевес в правую сторону производится левое вращение текущего потомка.
4. Вправо влево. Если у текущего узла наблюдается перевес вправо, а у его левого потомка – влево, то сначала производится правое вращение левого потомка, а затем левое вращение текущего узла.

```
Node<T>* balance(Node<T>* node) {
    int cur_balance = balanceFactor(node);

    if (cur_balance > 1 && balanceFactor(node->left_) >= 0) return rRotate(node);
    if (cur_balance > 1 && balanceFactor(node->left_) < 0) {
        node->left_ = lRotate(node->left_);
        return rRotate(node);
    }
    if (cur_balance < -1 && balanceFactor(node->right_) <= 0) return lRotate(node);
    if (cur_balance < -1 && balanceFactor(node->right_) > 0) {
        node->right_ = rRotate(node->right_);
        return lRotate(node);
    }
    return node;
}
```

Рис 16 – Реализация функции балансировки дерева

Анализ временной сложности реализованных операций

Рассмотрим асимптотическую сложность операций, реализованных в бинарном дереве поиска:

1. Функция поиска элемента. Так как бинарное дерево поиска несбалансированно, в худшем случае оно выродится в односвязный список, и высота дерева будет O(n). В этом случае, поиск элемента будет выполняться за O(n) времени, так как нам придется пройти все узлы дерева.
2. Функция вставки элемента. Так как дерево не сбалансировано и в худшем случае представляет собой линейную структуру (например, все узлы добавлены в порядке возрастания или убывания), высота дерева будет O(n). В этом случае, вставка нового узла будет выполняться за O(n) времени.
3. Функция удаления элемента. Так как дерево не сбалансировано и в худшем случае представляет собой линейную структуру (например, все узлы добавлены в порядке возрастания или убывания), высота дерева будет O(n). В этом случае, удаление узла будет выполняться за O(n) времени, так как нам придется пройти все узлы дерева, чтобы найти узел для удаления. Также при удалении узла в середине дерева, нам в любом случае придется спускаться к максимальному узлу левого поддерева этого узла, оттого асимптотическая сложность все также составляет O(n).

Сложность данной структуры по занимаемой памяти O(n).

Перед рассмотрением асимптотической сложности основных функций АВЛ-дерева, следует рассмотреть сложность вспомогательных функций:

1. Вычисление высоты узла. Все производимые в этой функции операции производятся за константное время, а при сложении асимптотического времени одинакового порядка константы не учитываются, то итоговое время выполнения данной функции O(1).
2. Операции левого и правого поворота. Данные операции являются зеркальными, оттого их асимптотическое время одинаково и составляет O(1).
3. Функция балансировки дерева. Является функцией с несколькими проверками условий, каждая из которых выполняется за константное время и вызывает функции вращения, оттого и сама исполняется за константное время.

Становится очевидным, что основные функции АВЛ-дерева, по реализации совпадающие с функциями бинарного дерева поиска, за тем лишь исключением, что при

исполнении функций вставки и удаления элемента производится балансировка дерева. При этом функция балансировки производится за константное время и не оказывает значительного влияния при теоретическом расчете асимптотической сложности. Куда более важное влияние имеет сбалансированность АВЛ-дерева, которое гарантирует максимальную высоту дерева $\log_2(n)$, где n – количество узлов дерева. Таким образом время исполнения всех основных функций АВЛ-дерева составляет $O(\log_2(n))$. Сложность же по памяти все также составляет $O(n)$.

Ссылка на реализацию

https://github.com/SomethingWF/Bin_Search_tree

Список источников

https://en.wikipedia.org/wiki/Binary_search_tree

https://ru.wikipedia.org/wiki/АВЛ-дерево

https://habr.com/ru/articles/267855/

https://habr.com/ru/articles/150732/

https://www.geeksforgeeks.org/cpp-binary-search-tree/

https://www.geeksforgeeks.org/cpp-program-to-implement-avl-tree/