

Лабораторная работа № 2 по курсу Дискретный Анализ. Словарь

Выполнил студент группы 08-207 МАИ *Павлов Иван*.

Условие

Кратко описывается задача:

1. Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до 264 - 1. Разным словам может быть поставлен в соответствие один и тот же номер.
2. Вариант задания: 1. AVL-дерево.

Описание алгоритма

Я ознакомился с алгоритмами для работы с AVL-деревом, используя материалы лекций, статью на Википедии и материалы с Ютуба.

Словари в большинстве языков программирования позволяют хранить пары ключ-значение, с быстрым поиском элемента по ключу, возможностью вставки и удаления элемента за приемлемое время. Одним из способов быстрого поиска элемента в отсортированной структуре является бинарный поиск. Двоичное дерево поиска использует данный алгоритм для всех операций и является всегда отсортированным, однако при неудачной последовательности вставки элементов оно может вырождаться в линейный список, где сложность операций составит $O(N)$, где N - это количество элементов.

Поэтому для реализации словарей используются сбалансированные деревья поиска, например, AVL-дерево. У AVL-дерева есть ключевое свойство: модуль разности высот левого и правого поддеревьев любого узла дерева не превышает 1. Для сохранения этого свойства используется балансировка при вставке и удалении элемента. Она осуществляется при помощи поворотов: левого и правого. Эти повороты меняют местами узел и его левого (правого) ребенка, сохраняя при этом свойства бинарного дерева поиска.

Поиск элемента аналогичен алгоритму бинарного поиска.

Вставка элемента делается в 3 этапа:

1. Вставляем новый элемент, аналогично алгоритму *BST*.
2. Начинаем подниматься к корню дерева, каждый раз пересчитывая баланс узлов.
3. Если баланс узла равен 2 (левый дисбаланс), то смотрим на баланс его левого сына. Он не может быть равен 0, так как его высота равна 1. Если его баланс равен 1

(имеется левый ребенок), то делаем правый поворот относительно исходного узла. Если баланс равен -1 (правый ребенок), то делаем левый поворот относительно его (приводим к первому случаю), а затем правый поворот относительно исходного узла. Если баланс исходного узла равен -2 (правый дисбаланс), поступаем симметрично противоположным образом.

При удалении элемента в стандартном алгоритме *BST* мы не можем просто удалить узел, у которого есть левое и правое поддерево одновременно. В таком случае элементу ищется замена - минимальный узел в правом поддереве. Если удаление элементу пошло по этому кейсу, то мы обратный обход к корню делаем от родителя этой замены, иначе - от родителей этого узла. Балансировка при обратном обходе при удалении аналогична балансировке при вставке.

Описание программы

Для реализации АВЛ-дерева я использовал рекурсивный подход. В каждом узле *Tree* дерева хранятся ключ *key* (строка длиной 256 или меньше), значение *value* (число типа unsigned long long), значение высоты узла *height*, указатели на левого *Tree.left* и правого *Tree.right* потомка, по умолчанию равны *NULL*.

Узел можно создать, используя оператор *new* и конструктор с параметрами ключ и значение. Над узлом определены следующие операции:

- *getHeight* - получить высоту узла (для *NULL*-узлов возвращаем 0, иначе поле *height*).
- *balance* - получить баланс узла (разность высот левого и правого поддерева узла).
- *leftRotate* - левый поворот. Пусть *x* - исходный узел, *y* - его правый потомок. По свойству *BST* $y > x$. Мы хотим "повернуть" дерево влево - значит поставить *y* на место *x*. По свойствам *BST* левый потомок *y* больше *x*, поэтому он должен стать правым потомком *x*. Сам *x* теперь является левым потомком *y*, а *y* встал на место *x*. Далее необходимо с помощью *getHeight* пересчитать высоты *x* и *y*.
- *rightRotate* - правый поворот. Реализован аналогично левому повороту.
- *successor* - минимальный элемент, больший текущего узла. Для ЛР2 не обязательно писать полностью функцию *successor*, так как она используется лишь при кейсе в удалении, когда у узла есть и левое, и правое поддерево (правый узел не *NULL*).

Поиск элемента: функция *find*.

Вход: *Tree*, *key*.

Выход: *true*, если элемент есть в дереве, *false* если его нет.

Реализация: проверяем равенство *Tree* нулю. Если это так (элемент не найден), выводим *NoSuchWord*, возвращаем *false*. Если нет, то 3 случая: 1) если *Tree.key > key*,

то возвращаем функцию *find* от *Tree.left* и *key*. 2) если *Tree.key* < *key*, то возвращаем функцию *find* от *Tree.right* и *key*. 3) если *Tree.key* = *key*, то получен искомым элемент, выводим *OK*, возвращаем *true*.

Вставка элемента: функция *insert*.

Вход: *Tree*, *key*, *value*.

Выход: узел *x*.

Реализация:

1. Если *Tree* = *NULL*, то найдено место для вставки. Выводим *OK*, создаем с помощью конструктора *x*, возвращаем его.
2. Если *Tree.key* > *key*, то присваиваем *Tree.left* функцию *insert* от *Tree.left*, *key* и *value*.
3. Если *Tree.key* < *key*, то присваиваем *Tree.right* функцию *insert* от *Tree.right*, *key* и *value*.
4. Если *Tree.key* = *key*, то элемент *x* уже существует в дереве. Выводим *Exist*, возвращаем *x*.

Обратный проход к корню в рекурсивной реализации получается сам собой при возвращении из стека вызовов случаев 2 и 3. Все что нужно сделать - написать реализацию балансировки после этих вызовов. Она автоматически применится для случаев 2 и 3 при возвращении из случая 1; в случае 4 она просто не будет менять балансы узлов, так как мы никакую высоту не поменяли.

Реализация балансировки:

1. Пересчитываем высоты узлов: берем максимум *getHeight* левого и правого потомка, прибавляем к нему 1. Присваиваем данное значение полю *height* текущего узла.
2. Получаем *balance* текущего узла.
3. Если модуль значения больше 1, то применяем алгоритм балансировки, описанный выше. Возвращаем *x*, полученный в результате всех необходимых поворотов.

Удаление узла: функция *erase*.

Вход: *Tree*, *key*.

Выход: элемент *x*, вставший на место удаленного узла.

Реализация:

Для кейса, когда у искомого узла есть и левый, и правый потомки я реализовал дополнительную функцию *eraseMinimalNode*, которая занимается поиском замены (минимального правого узла с высотой 1) и удалением ее. Функция принимает на вход *Tree*; если *Tree.left = NULL*, то заменяет *Tree.left* на *Tree.right*, а *Tree.left* удаляет. В противном случае рекурсивно вызывается от *Tree.left*, затем балансируется аналогично вставке.

Сама функция *erase*:

1. Если *Tree = NULL*, то искомый элемент не найден. Выводим *NoSuchWord*, возвращаем *NULL*.
2. Если *Tree.key > key*, то присваиваем *Tree.left* функцию *erase* от *Tree.left*, *key* и *value*.
3. Если *Tree.key < key*, то присваиваем *Tree.right* функцию *erase* от *Tree.right*, *key* и *value*.
4. Если *Tree.key = key*, то выводим *OK* и проверяем, какой из кейсов удаления нам использовать. Если оба потомка *NULL*, то просто удаляем искомый элемент, возвращаем *NULL*. Если у элемента есть один левый (правый) потомок, то удаляем элемент, возвращаем левый (правый) потомок. Если есть оба потомка, то присваиваем элементу $\{successor.key, successor.value\}$ и вызываем *eraseMinimalNode*.

После удаления производится балансировка, аналогичная балансировке при вставке.

Уничтожение дерева: функция *destroy*.

Вход: *Tree*.

Выход: *NULL*.

Реализация: Если левый и правый потомок текущего элемента равны *NULL*, то удаляем элемент, возвращаем *NULL*, присваиваем *Tree.left destroy* от *Tree.left* и *Tree.right destroy* от *Tree.right*, затем удаляем *Tree*, возвращаем *NULL*.

Сохранение дерева в файл в компактном двоичном представлении. Реализовано в соответствии с требованиями чекера.

1. открываем бинарный файл на запись.
2. если файл существует, то вызываем функцию *save* и закрываем файл. Выводим *OK*. Иначе просто выводим *OK*.

Функция *save*.

Вход: *Tree*, *f*.

Реализация: есть две метки: *NODE*, означающая начало узла в файле и *END*, означающая *NULL*-узел в дереве. Если *Tree = NULL*, пишем в файл *END*, иначе пишем последовательно пишем *NODE*, длину ключа, ключ, значение, высота. затем вызываем *save* от *Tree.left* и *Tree.right*.

Загрузка дерева из файла. Реализовано в соответствии с требованиями чекера.

Если файла нет, то вызываем *destroy* от *Tree* и присваиваем ему *NULL* (так как из не существующего файла загружается пустое дерево). Иначе если *Tree! = NULL* то уничтожаем дерево и вызываем функцию *load*.

Функция *load*.

Вход: *Tree, f*.

Реализация: считываем метку из файла, если она равна *NODE*, считываем все параметры и создаем узел дерева. Иначе завершаем. Рекурсивно вызываем *load* от *Tree.left* и *Tree.right*.

Дневник отладки

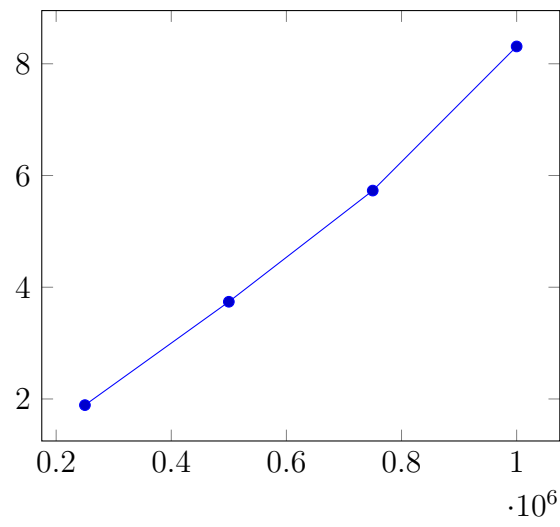
RE7 закрывал файл, равный *NULL*. Добавил *if*.

RE9 undefined behavior при вызове методов от объектов, равных *NULL*. Заменял все методы на аналогичные функции.

Тест производительности

Сравним производительность вставки, удаления и поиска АВЛ-дерева и *std::map*, который использует под капотом красно-черное дерево. Синим цветом выделена зависимость времени работы АВЛ-дерева от количества входных данных, красным цветом - зависимость времени работы *std::map* от количества входных данных.

1. Вставка



Выводы