Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент:	С. М. Власова
Преподаватель:	А. А. Кухтичев
Группа:	М8О-206Б
Дата:	
Оценка:	
Подпись:	

Москва, 2019

Лабораторная работа №2

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.
Вариант задания: Красно-чёрное дерево.

1 Описание

Как сказано в [1]: «Красно-чёрное дерево представляет собой бинарное дерево поиска с одним дополнительным битом цвета в каждом узле. Цвет узла может быть либо красным, либо чёрным. В соответствии с накладываемыми на узлы дерева ограничениями, ни один путь в дереве не отличается от другого по длине более чем в два раза, так что красно-чёрные деревья являются приближенно сбалансированными».
Бинарное дерево поиска является красно-чёрным, если удовлетворяет следующим красно-чёрным свойствам:

1. Каждый узел является красным или чёрным.
2. Корень дерева является чёрным.
3. Каждый лист дерева (NIL) является чёрным.
4. Если узел – красный, то оба его дочерних узла – чёрные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов (чёрная высота).

Лемма Красно-чёрное дерево с n внутренними узлами имеет высоту не более чем $2lg(n+1)$.

Доказательство делится на два этапа. Сначала по индукции по высоте x доказывается, что поддерево любого узла x содержит как минимум $2^{bh(x)} - 1$ внутренних узлов, где *bh(x)* – чёрная высота.
Далее, пусть h – высота дерева. Тогда, согласно свойству 4, по крайней мере половина узлов на пути от корня к листу, не считая сам корень, должны быть чёрными. Следовательно, чёрная высота корня должна составлять как минимум $h/2$. Получаем, что

$$n \geq 2^{h/2} - 1$$

Перенося 1 в левую часть и логарифмируя, получим, что $\lg(n+1) \geq h/2$, или $h \leq 2\lg(n+1)$.

Согласно [1]: «непосредственным следствием Леммы является то, что такие операции над динамическими множествами, как поиск минимума, максимума, предыдущего и последующего элементов при использовании красно-чёрных деревьев выполняются за время $O(\lg n)$». Такие операции, как вставка и удаление так же могут быть выполнены за время $O(\lg n)$.

2 Исходный код

В каждом узле красно-чёрного дерева *RBNode* будет хранится пара «ключ-номер» (*char * key, unsigned long long value*), цвет узла (*int BLACK = 1, int RED = 0*), ссылки на родителя (*RBNode * parent*), правого потомка (*RBNode * right*) и левого потомка (*RBNode * left*).
Определим узел *RBNode EmptyNode = { NIL, NIL, NIL, BLACK, 0, 0}*, который будет соответствовать пустому узлу. Ссылку на такой узел обозначим, как *NIL.*
Корень дерева определим, как глобальную переменную *RBNode * root = NIL.*

Дерево создано, и теперь нам доступны такие операции, как добавление узла *RBInsert* и вспомогательная к ней процедура *RBInsertFixUp*; удаление узла по значению ключа *RBDelete* и вспомогательная к ней процедура *RBDeleteFixUp*; поиск узла по значению ключа *RBFind*; сохранение дерева в бинарном виде *UploadFile* и загрузка уже сохраненного ранее дерева *LoadFile.* Операция сохранения дерева реализуется с помощью таких операций над вектором, как *Push* и *Clear.*
Так же, такие процедуры, как *LeftRotate* и *RightRotate* используются для балансировки дерева при добавлении узла или же его удалении.

Чтобы идентифицировать операцию, при считывании данных идентифицируем первый входной символ.
Если это «+» – считываем до конца ключ, приводим его к нижнему регистру. Далее, проверяем, есть ли элемент с таким ключом в дереве с помощью процедуры *RBInFind.* Если дерево не пустое, и узел с таким ключом уже есть, то эта процедура возвращает значение *NIL.* Если же узел с таким ключом не найден, *RBInFind* она возвращает ссылку на потенциального родителя элемента, который мы собираемся добавить.
Так, если пара «ключ-значение» может быть добавлена, вызывается процедура *RBInsert*, параметрами которой являются ключ, номер и ссылка на родителя. Эта операция не представляет никакой сложности в отличие от процедуры восстановления красно-чёрных свойств *RBInsertFixUp*, которая вызывается сразу после добавления узла. Так как, добавленный узел всегда *красный*, может нарушиться второй свойство, согласно которому корень всегда *чёрный* и четвертое свойство: «если узел – красный, то оба его дочерних узла – чёрные».
Рассмотрим *RBInsertFixUp* подробнее.
В каждой итерации цикла *while* :

1. узел x – красный;

2. если x-> *parent* – корень дерева, то x-> *parent* – чёрный;

3. если имеется нарушение красно-чёрных свойств, то это нарушение либо свой-

ства 2, либо свойства 4. Если нарушено второе свойство, то это вызвано тем, что красный узел x – корень дерева; если же четвертое, то узлы x, x-> *parent* – оба красные.

При работе цикла ***while*** следует рассмотреть шесть случаев, однако три из них симметричны другим трём; разница лишь в том, является ли родитель x-> *parent* левым или правым дочерним узлом по отношению к своему родителю x-> parent-> parent. Узел x-> *parent*-> *parent* существует, поскольку цикл начинает свою работу только в том случае, если x-> *parent* – красный, а значит, не может быть корнем дерева. Случай 1 отличается от случаев 2 и 3 только цветом дяди y узла x.

Случай 1: узел y красный:

Поскольку x-> *parent*-> *parent* – чёрный, мы можем исправить ситуацию, покрасив и x-> *parent*, и y в чёрный цвет (после чего цвет красного родителя узла x становится чёрным, и нарушение между x и его родителем исчезает), а x-> *parent*-> *parent* – в красный цвет, для того, чтобы выполнялось свойство 5. После этого мы выполняем цикл ***while*** с узлом x-> *parent*-> *parent* в качестве нового узла x. Указатель x перемещается таким образом на два уровня вверх.

Случай 2: узел y чёрный и x – правый потомок.
Случай 3: узел y чёрный и x – левый потомок.

В случаях 2 и 3 цвет узла y, являющегося дядей узла x, чёрный. Эти два случая отличаются друг от друга тем, что x является правым или левым дочерним узлом по отношению к родительскому. В случае 2 узел x является правым потомком своего родительского узла. Мы используем левый поворот для преобразования случившейся ситуации в случай 3, когда x является левым потомком. Поскольку и x, и x-> *parent* красные узлы, поворот не влияет ни на чёрную высоту, ни на выполнения свойства 5. Когда мы приходим к случаю 3 (с помощью поворота или изначально), узел y имеет чёрные цвет. Кроме того, обязательно существует узел x-> *parent*-> *parent*, что было доказано в случае 1.
В случае 3 выполняется ряд изменений цвета и правых поворотов, которые сохраняют свойство 5. После этого, т.к. у нас больше нет подряд идущих красных узлов, процедура завершается. Теперь давайте покажем, что случаи 2 и 3 сохраняют инвариант цикла.

1. В случае 2 выполняется присвоение, после которого узел x указывает на красный узел x-> *parent*

2. В случае 3 узел x-> *parent* делается чёрным, так что если x-> *parent* в начале следующей итерации является корнем, то он становится чёрным.

3. Как и в случае 1, в случаях 2 и 3 свойства 1, 3 и 5 сохраняются.

Итак, в процедуре *RBInsertFixUp* цикл **while** повторно повторяется только в случае 1, и в этом случае указатель x перемещается вверх по дереву на два уровня.
Таким образом, общее количество возможных выполнений тела цикла **while** равно $O(\lg n)$. Таким образом, общее время работы процедуры равно $O(\lg n)$.

Далее, если необходимо удалить элемент, прежде всего выполняется его поиск с помощью процедуры *RBFind*, которая выполняется за время $O(\lg n)$. Если элемент найден, то *RBFind* возвращает указатель на этот элемент, который передается в качестве параметра процедуре .

Рассмотрим процедуру удаления подробнее.
Пусть z – удаляемый элемент. Так же, как и для удаления в обычном бинарном дереве поиска, выбираются два элемента x и y, где y – минимальный элемент, больший z или, если z – лист, равен z, а x – это левый или правый потомок y. Так, элемент x встает на место y, а на все данные узла y становятся данными удаленного узла z.
В случае, если y – чёрный, вызывается вспомогательная процедура *RBDeleteFixUp*.
Узел x, который передается в качестве параметра во вспомогательную процедуру является либо единственным потомком y перед его извлечением, либо ограничителем *NIL*.
Рассмотрим процедуру восстановления красно-чёрных свойств *RBDeleteFixUp*.
Если извлекаемый из дерева в процедуре узел y чёрный, то могут возникнуть три проблемы. Во-первых, если y был корнем, а теперь корнем стал красный потомок y, нарушается свойство 2. Во-вторых, если и x, и *y-> parent* (который теперь является *x-> parent*) были красными, то нарушается свойство 4. И, в-третьих, удаление y приводит к тому, что все пути, проходившие через y, теперь имеют на один чёрный узел меньше. Таким образом, для всех предков y становится нарушенным свойство 5.

Случай 1: узел x красный:

В первом случае нарушение можно компенсировать, перекрасив узел x в черный цвет. Это позволит ликвидировать и нарушение третьего свойства, если родитель вырезанного узла y -– красный.

Случай 2: узел x чёрный:

Во втором случае применяется перестройка в корне поддерева, где возникло нарушение четвертого свойства. Причем узел x считается дважды чёрным, способным поделиться своей чернотой с вышележащими на пути к нему узлами. Нужно перестроить вышележащие узлы таким образом, чтобы избавиться от дважды чёрного узла, не нарушив красно-чёрных свойств дерева. Если x — корень, тогда лишняя чернота просто удаляется из дерева. Иначе выполняется циклическая перестройка корней поддеревьев, лежащих на пути к x, путем перекрашивания и поворотов. Перестройка учитывает 4 возможных случая, которые могут встретиться на пути. Пусть w – брат x.

Случай 2.1: узел w чёрный:

Это наиболее благоприятный случай. Можно перекрасить родителя x в чёрный цвет, x сделать обычным чёрным узлом, а w -– красным. На этом перестройка заканчивается. Если родитель x -– чёрный, то он становится дважды чёрным, и перестройка поднимается в верхнее поддерево.

Случай 2.2: узел w красный:

Этот случай можно свести к первому, чтобы братом x стал чёрный узел. Для этого сделаем левый поворот родителя узла x.

Случай 2.3: сыновья w разного цвета:

Перекрасив родителя x в чёрный цвет, а w -– в красный, мы нарушим свойство 3 для w. Поэтому сделаем сначала правый поворот узла w.

Случай 2.4: правый сын узла w красный:

Перекрашивать нельзя. Левый сын может быть и красным, и черным. Опять же действие по схеме случая 1 не годится, т.к. это приведет к нарушению свойства 3. Тогда произведем левый поворот родителя x. Узел x отдает свою черноту родителю, а правый сын узла w перекрашивается в чёрный цвет. Узел w берет цвет корня поддерева до поворота.

Давайте разберем операцию сохранения красно-чёрного дерева *UploadFile*, т.к. она представляет наибольший интерес. Красно-чёрное дерево будет храниться в виде вектора целых чисел, где для каждого узла, начиная с наименьшего, будет записана длина ключа, каждый символ ключа и номер, приведенный к целому типу. Такая запись красно-чёрного дерева будет наиболее оптимальной. Чтобы загрузить дерево в таком виде, в процедуре *LoadFile* элементы вектора будут считываться по очереди, а дерево будет заново строиться.
В табличке представлены вспомогательные процедуры, не упомянутые в описании выше.

lab02.c	
void RBDestroy(RBNode * tree)	Функция, которая уничтожает дерево
void LeftRotate(RBNode * x)	Функция, выполняющая левый поворот
void RightRotate(RBNode * x)	Функция, выполняющая правый поворот

Листинг 1: Структуры

```
typedef struct RBTree{
  struct RBTree * left;
  struct RBTree * right;
  struct RBTree * parent;
  int color;
  unsigned long long value;
  char * key;
} RBNode;

typedef struct{
  int * value;
  int size;
  int capacity;
} Vector;
```

3 Консоль

svetlana@svetlana-VirtualBox /DA/lab2/2/d2 $ ./dys
+ abcdef 123
OK
+ ABCdef 456
Exist
+ qwerty 90
OK
+ qwer 14
OK
- qweR
OK
! Save tree
OK
- qwerty
OK
! Load tree
OK
qwerty
OK: 90
! Load TT
ERROR: No such file or directory

4 Тест производительности

Тест производительности программы будет заключаться в сравнении результатов работы красно-чёрного дерева, реализованного мной, и STL-контейнера *map* на сгенерированных тестах в 10.000 записей, 100.000 записей и 1.000.000 записей. Время выполнения будет записано в миллисекундах.

5 Консоль

```
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ wc -l test∗
10000 test1.txt
100000 test2.txt
1000000 test3.txt
1110000 итого
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./dys < test1.txt | tail -n 1
RBTreeTime: 0.061234
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./map < test1.txt | tail -n 1
MapTime: 0.056886
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./dys < test2.txt | tail -n 1
RBTreeTime: 0.750981
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./map < test2.txt | tail -n 1
MapTime: 0.639819
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./dys < test3.txt | tail -n 1
RBTreeTime: 7.914515
svetlana@svetlana-VirtualBox /DA/lab2/2/d2/dys $ ./map < test3.txt | tail -n 1
MapTime: 5.95697
```

Результаты теста производительности:

Data	RBTree	STL map
10.000	61.234	56.886
100.000	750.981	639.819
1.000.000	7914.515	5956.97

Исходя из результатов, можно сделать вывод о том, что функция добавления в красно-чёрное дерево, реализованная мной, несколько проигрывает по скорости STL-функции insert, реализованной на *std::map*. К слову, *std::map* так же реализован на красно-чёрном дереве.

6 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я познакомилась с такой структурой данных, как красно-чёрное дерево. Операции добавления и удаления в красно-чёрном дереве выигрывают по скорости у того же бинарного дерева поиска. Это объясняется тем, что каждый узел красно-чёрного дерева имеет дополнительный бит цвета, а дерево удовлетворяет красно-чёрным дополнительным свойствам, благодаря которым ни один путь в нем не отличается от другого более чем в два раза. Так, красно-чёрное дерево является приближенно сбалансированным.
Балансировка дерева реализуется с помощью поворотов и манипуляций с битами цвета.

Список литературы

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. *Алгоритмы: построение и анализ, 2-е издание* — М.: «Вильямс», 2005. — с. 230 - 234. — ISBN 5-8459-0857-4.