Красно-черное дерево

С. А. Шершаков

15 мая 2017 г.

Ред. 1.0 от 15.05.2017 г.

В документе представлено задание для самостоятельного выполнения студентами курса «Алгоритмы и структуры данных» и методические рекомендации по их выполнению, посвященное красно-черным бинарным деревьям поиска. Красно-черное дерево является самобалансирующимся бинарным деревом поиска, основные операции над которым выполняется за логарифмическое время. Это обуславливает широкое применение этой структуры данных на практике для представления таких ADT, как множество, отображение и др Реализация задачи включает взаимодействие с инструментом автодокументирования кода Doxygen и unit-тестированием на основе библиотеки gtest.

1 Требования, цели и ожидаемые результаты

1.1 Требования

Студенты должны владеть следующими знаниями, умениями и навыками для выполнения задания.

- Разработка модульных приложений на языке C++.
- Концепция *шаблонов*, *обобщенного программирования*, параметризованные функции и классы C++.
- Основы стандартной библиотеки C/C++.
- Бинарные деревья поиска (BST) и их основные операции.

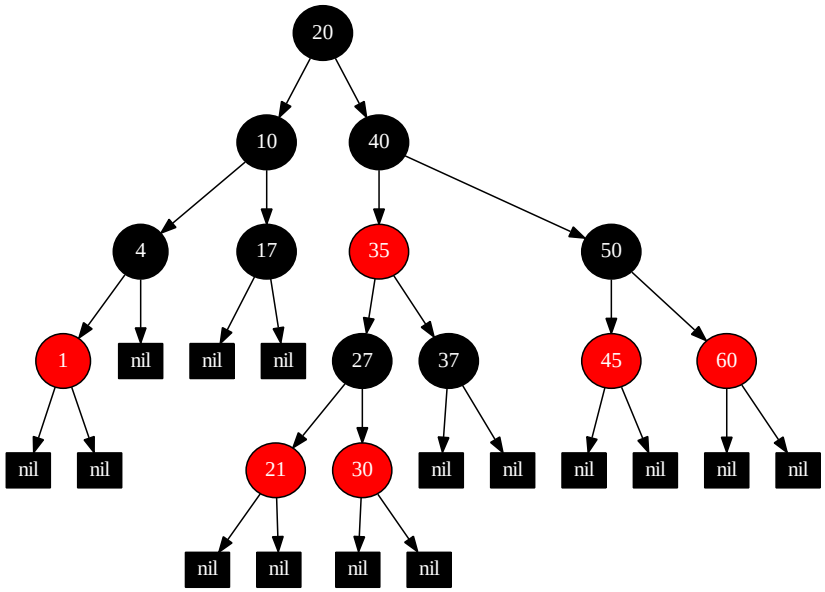

Рис. 1: Черно-красное дерево, построенное для последовательности вставок: 4, 50, 10, 40, 17, 35, 20, 27, 37, 45, 60, 21, 1, 30.

Основная цель работы — практическое изучение реализации черно-красного дерева (КЧД, RBT) и основных операций над ним.

Связанные задачи: автодокументирование кода с использование инструмента Doxygen[1], unit-тестирование с использованием библиотеки gtest[2], построение графического представления дерева на языке DOT для инструмента Graphviz [3].

[1] http://www.doxygen.org

[2] https://github.com/google/googletest

[3] http://www.graphviz.org/

Задачи для выполнения в рамках самостоятельной работы:

- Изучить способы реализации (обыкновенного) бинарного дерева поиска (BST) и основных операций над ним (рис. 1).
- Изучить интерфейсы шаблонных классов `RBTree<Element, Compar>` (КЧ дерево) и `RBTree<Element, Compar>::Node` (узел КЧ дерева), заголовочный файл `rbtree.h`, по предложенному коду и Doxygen-документации.
- Изучить реализацию методов этих классов, представленных в файле `rbtree.hpp`.
- Разобраться с механизмом подписки на внутренние события классов дерева и узла для формирования дампа промежуточного состояния структуры дерева с целью визуализации и отладки промежуточных шагов, выполняемых при балансировке дерева. Интерфейс `IRBTreeDumper<Element, Compar>` (файл `rbtree.h`) и его реализация по умолчанию (файл `def_dumper.h` из проекта unit-тестов).
- Разобраться с реализацией по умолчанию компонент логгирования событий дерева (класс `RBTreeDefDumper<Element, Compar>`) и вывода дампа структуры дерева в формате DOT (класс `RBTreeGvDumper<Element, Compar>`).
- Изучить концепцию методов левого и правого поворота поддерева по базовой книге [1].
- Изучить алгоритм вставки элемента в красно-черное дерево по [1].
- Закончить реализацию некоторых методов классов КЧ дерева и узла КЧ дерева, ориентируясь на спецификацию метода (см. Doxygen-документацию) и настоящее руководство;
- [*Опционально*] протестировать полученную реализацию на множестве предложенных unit-тестов;
- [* *Опционально*] реализовать и протестировать метод `remove` удаления элемента из КЧ дерева[4].

[4] Задача «со звездочкой» на 10 баллов.

2 *Предпосылки*

Красно-черное дерево (КЧД) является специальной разновидностью бинарного дерева поиска (BST), обладающей следующими *основными свойствами*:

1. Любой узел дерева помечен дополнительным битом, определяющий, является ли узел «красным» или «черным».
2. Корень дерева черный.

3. Если некоторый узел красный, то оба его потомков обязательно черные.
4. Любой простой пусть от некоторого узла до всех листьев дерева содержит одинаковое число черных узлов.

Среди свойств иногда выделяют дополнительно, что все листья содержат квази-подлистья, помеченные `NIL`, и они в таком случае являются черными. На рисунке 1 изображено дерево, где все листья являются как раз такими нулевыми элементами. В программной реализации в данном проекте в качестве нулевого листа будет использовать строко нулевой указатель `nullptr`[5]

Иногда требования, что корень дерева является черным специально не выделяют, т.к. его всегда можно изменить с красного на черный без потери строгости остальных свойств дерева[6].

Основным утверждением по отношению к «правильному» КЧД является следующая

***Теорема* 1.** *Красно-черное дерево с n-внутренними узлами имеет высоту* $h \leqslant 2\ \log(n+1)$.

Теорема доказывается по индукции. Интуитивно можно сделать следующее обобщение: если все красные узлы объединить в один «мегаузел» со своим черным предком[7], тем самым исключив красные узлы из рассмотрения, то дерево будет представлено только своими черными узлами. Далее, пользуясь утверждением, что любой путь от любого узла до черных листье одинаковый, мы можем рассмотреть все возможные пути от корня до листьев. Так как между двумя черными узлами в пути не может быть более одного красного узла, то максимальное число узлов не может увеличиться более, чем в два раза, что и дает соответствующий коэффициент при логарифме.

Прямым и важным следствием данного утверждения является то, что сложность операции поиска элемента в таком дереве относится к классу $O(\log n)$ и в среднем, и в худшем случаях.

При вставке (удаления) очередного элемента в КЧД как в обычное BST, может нарушаться одно или несколько свойств, что в свою очередь привело бы к нарушению утверждения (теоремы 1). Чтобы этого не произошло, после вставки (удаления) элемента выполняется перебалансировка дерева таким образом, чтобы восстановиться справедливость указанных свойств. Данный процесс выполняется в некоторой локальной окрестности добавляемого (удаляемого) узла, что приводит к восстановлению свойств в этом окрестности, но может в свою очередь приводить к нарушению свойств в окрестности родителя этого узла. Однако, применяя процедуру восстановления рекурсивно ко всем родителям (вплоть до корня), удается восстановить «баланс» дерева целиком. Интуитивно понятно, что для этого необходимо выполнить операций локальной перебалансировки не больше, чем высота дерева, а

[5] Следует отметить, что часто предлагаемые алгоритмы ориентируются на то, что листья действительно относятся к специальной разновидности узлов `NIL`. Иногда для этих целей выделяется специальны единственный узел, и все действительные листья дерева «замыкаются» на него.

[6] С точки зрения реализации, корневой узел иногда делают подчиненным специальному искусственному «предкорневому» узлу. Тогда «настоящий» корень становится правым потомком такого «сторожа», а его левый брат получает условный вес $-\infty$. В данном случае такой подход не используется.

[7] В таком виде наглядным становится известный факт, что КЧД является *изометрической проекцией* другого виде деревьев — *B-дерева порядка 4*, или 2-3-4-дерева.

значит сложность вставки (удаления) элементов в КЧД также является $O(\log n)$.

Итеративный алгоритм перебалансировки дерева включает две основные процедуры: 1) перекраска узлов, 2) локальный поворот поддерева[8]. Сложность алгоритмов вставки (и в еще большей мере — удаления) заключается в большом числе специальных случаев[9], определяющих что именно — перекрашивание узла или поворот поддерева — следует осуществлять на очередном шаге алгоритма. Количество вариантов удваивается за счет симметрии «левых-правых» случаев, а также за счет специального случая с корнем дерева[10]

Алгоритмы со всеми возможными случаями подробно рассматриваются в литературе [1].

[8] Процедуры, реализуемые студентами в рамках настоящего задания.

[9] В следующих заданиях будет показано, что вместо изучения множества специальных случаев, можно рассматривать КЧД как изометрию 2-3-4-дерева, для которых каждый из таких специальных случаев имеет более простое и наглядное трактование.

[10] Как раз по этой причине в некоторых реализациях алгоритма вводится дополнительный «предкорневой» элемент.

3 *Описание задания*

В рамках самостоятельной работы студентам предлагается закончить реализацию указанных ниже методов классов КЧ дерева и узла КЧ дерева.

Далее представлена краткая информация по основным модулям/типам программного кода. Дополнительная информация может быть получена из автосоздаваемой документации Doxygen[11] или из комментариев к файлам/типам.

[11] Предоставляется в виде архива с каталогом `/html`, из которого необходимо открыть файл `/html/index.html` в любом браузере.

3.1 *Модуль `rbtree.h/.hpp`*

Содержит описание шаблонного класса КЧ дерева и вспомогательные типы. Основной рабочий модуль проекта. Используется описанный в ранних заданиях подход по разделению на декларационную часть (`.h`) и квазиимплементационную (`.hpp`).

3.2 *Модуль `def_dumper.h`*

Модуль для тестирования КЧ дерева, содержащий модуль вывода структуры дерева в виде DOT-разметки графа и модуль-обработчик событий дерева для отслеживания и отладки изменения его структуры.

3.3 *Модуль `rbtree_prv1_test.cpp`*

Содержит unit-тесты (gtest) для тестирование некоторых *закрытых* методов КЧ дерева. Используется специальным образом, подробнее — см. комментарии в коде или Doxygen-документацию.

3.4 *Модуль `rbtree_pub1_test.cpp`*

Содержит unit-тесты (gtest) для тестирование открытого интерфейса КЧ дерева (в режиме «черного ящика»).

3.5 *Модуль* `main.cpp`

Содержит простой smoke-тест, создающий реализацию КЧ дерева с конкретизацией целочисленным типом.

3.6 *Класс* `RBTree<Element, Compar>`

Описывает КЧ дерево. Класс является верхним в иерархии наследования (т.е. не наследует у других классов). Содержит следующие (основные) открытые методы: `insert()`, `remove()`[12], `find()`, `isEmpty()`, `getRoot()`, `setDumper()`, `resetDumper()`. Подробное описание методов представлено в комментариях и Doxygen-документации.

Основными методами этого класса, реализуемыми студентами самостоятельно, являются методы `insert()`, `remove()`[12], `rotLeft()` и `rotRight()`.

[12] Опционально, для «задачи со звездочкой.»

3.6.1 *Метод* `insert()`

Осуществляет вставку элемента в корректную позицию бинарного дерева поиска и выполняет перебалансировку дерева так, что оно продолжает удовлетворять основным свойствам КЧД.

Заготовка метода включает два основных шага:

1. Вставка элемента в дерево, как если бы это было обычное BST. При этом сохраняется свойства BST, но возможно нарушается одно или несколько свойств RBT. Предлагается реализовать в виде метода `insertNewBstEl()`.
2. Перебалансировка RBT так, чтобы восстановить возможно нарушенные свойства RBT после вставки в него элемента, как в обычное BST. Предлагается реализовать в виде метода `rebalance()`.

После вызова каждого из этих методов эмитируется соответствующее отладочное событие, на которое можно подписаться с целью изучения промежуточного состояния дерева. Пример эмиссии события после вставки элемента в BST:

```
1  if (_dumper)
2      _dumper->rbTreeEvent(IRBTreeDumper<Element, Compar>::▼
       DE_AFTER_BST_INS, this, newNode);
```

где `_dumper` — это объект-отладчик (дампер), `rbTreeEvent` — единый метод-событие, `DE_AFTER_BST_INS` — идентификатор типа события.

В целом, метод `insert()` сам по себе не требует дополнительной доработки[13], а основной вклад состоит в дополнении/разработке методов `insertNewBstEl()` и `rebalance()`[14].

[13] При необходимости он может быть дополнен или даже полностью переписан.

[14] Чтобы корректно реализовать эти методы, необходимо очень аккуратно декомпозировать задачи и выделить вспомогательные методы для решения каждой индивидуальной задачи.

3.6.2 *Метод `remove()`*

Осуществляет удаление из дерева элемента и выполняет перебалансировку дерева так, что оно продолжает удовлетворять основным свойствам КЧД.

Реализация данного метода является опциональным[12] по желанию студента. Метод становится видимым для компилятора при условии определения символа препроцессора `RBTREE_WITH_DELETION`. Таким образом, если студент претендует на решение «задачи со звездочкой», он(а) должен явным образом задекларировать этот символ.[15]

Декомпозиция[16] и инструментирование[17] метода `remove()` осуществляется студентом самостоятельно.

3.6.3 *Методы `rotLeft()` и `rotRight()`*

Выполняют поворот поддерева относительно узла соответственно влево или вправо (рис. 2). Наравне с перекраской узлов является одним из основных шагов в процессе балансировки дерева.

Методы являются закрытыми, так как произвольные повороты в дереве могут нарушить его свойства. Данные методы тестируются специальным образом.

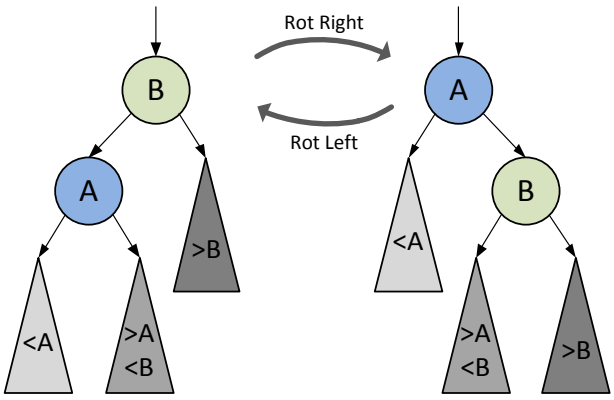

Рис. 2: Поворот дерева вправо/влево.

3.7 *Класс `RBTree<Element, Compar>::Node`*

Описывает узел КЧ дерева. Большинство методов узла являются закрытыми, так как не имеет большого смысла предоставлять открытый интерфейс, гарантируя при этом внутреннюю целостность дерева. Класс дерева имеет доступ ко всем закрытым членам класса узла за счет того, что последний объявляет первого своим другом:

```
1 friend class RBTree<Element, Compar>;
```

[15] Декларация этого символа помимо самого метода открывает и другие участки кода, связанные с методом. Например, соответствующие unit-тесты.

[16] Методы, выделенные в результате декмопозиции `remove()`, предлагается также включить в препроцессорное условие `RBTREE_WITH_DELETION`.

[17] Добавление не влияющих на логику алгоритма конструкций (инъекций) с целью получения статистики по исполнению. В данном случае код инструментируется вызовами методов отладчика `_dumper`.

Открытые методы класса узла носят вспомогательный характер и объявлены константными, что предотвращает возможность изменения состояния узла. Методы задокументированы в коде (и в документации Doxygen), их состав и назначение определяется реализацией методов `insert()`, `remove()` и `find()` дерева.

Семантика класса узла может при необходимости расширяться (в части открытых и закрытых методов).

3.8 Отладочный интерфейс `IRBTreeDumper<Element, Compar>` и его реализация по умолчанию `RBTreeDefDumper<Element, Compar>`

Интерфейс описывает единственный метод `rbTreeEvent()`, соответствующий одному из событий, описанных в перечислении `IRBTreeDumper<E,C>::RBTreeDumperEvent`.

Класс `RBTree<...>` включает в себя указатель `_dumper` на объект типа `IRBTreeDumper<...>`, установленный по умолчанию в `nullptr`. Если данному указателю присваивается[18] адрес объекта дампера/отладчика, то определенные события начинают генерироваться в процессе изменения структуры дерева.

[18] Установка и сброс объекта-отладчика осуществляется методами `setDumper()` и `resetDumper()` соответственно.

Одна из возможных реализаций по умолчанию дампера предложена в виде класса `RBTreeDefDumper<Element, Compar>`. Данный класс открывает текстовый файл журнала, в который записывает отладочные события, происходящие с деревом. Большинству событий также соответствует вывод промежуточной структуры дерева в виде DOT-разметки[19] с помощью объекта класса `RBTreeGvDumper<E,C>`.

[19] Для последующей визуализации с помощью инструмента Graphviz.

Пример использования отладчика представлен в unit-тесте `TEST_F(RBTreePubTest, insert1)`.

4 Структура проекта

Структура проекта является стандартной для аналогичных заданий курса. Папка `/scripts` содержит примеры сценариев для командного интерпретатора Windows, выполняющих конвертацию в текущей папке всех файлов с расширением `.gv` в графические файлы в форматах `.ps` и `.gif`. Данные сценарии легко могут быть адаптированы под другие интерпретаторы.

5 Указания по выполнению задания

Выполнение задачи сводится к следующей рекомендуемой последовательности действий.

- Ознакомиться с настоящим руководством-заданием.
- Получить исходные файлы для работы[20].

[20] С использованием системы LMS: `http://lms.hse.ru`

- Ознакомиться с предлагаемыми для работы исходными файлами и автодоксидокументацией к ним.
- Определиться с объемом выполняемой работы. Задекларировать макрос `RBTREE_WITH_DELETION`, если планируется выполнять задачу «со звездочкой».
- Реализовать недостающие методы, модифицировав файлы `rbtree.h/.hpp`. **Важный момент**: необходимо внимательно просмотреть код модуля на предмет комментариев, указывающих, какие методы в каком объеме надо реализовывать.
- Протестировать (по желанию) реализацию на предложенном наборе тестов (файлы `rbtree_prv1_test.cpp` и `rbtree_pub1_test.cpp`).
- При необходимости расширения множества тестов, необходимо поместить их в новый файл `rbtree_student_test.cpp`.
- Загрузить результаты работы в виде единого архива в ассоциированный проект системы LMS до крайнего срока, указанного в сводке проекта.

5.1 Сдаваемые файлы

- `rbtree.h/.hpp` — реализация основной задачи;
- `rbtree_student_test.cpp` — дополнительные тесты, если разрабатывались;
- *при необходимости*, другие файлы с учетом стандартных замечаний[21].

[21] При выполнении задачи нет необходимости в изменении каких-либо файлов, не перечисленных выше. В случае, если такая необходимость возникает, при сдаче такого файла необходимо сопроводить его комментариями в виде файла `student_notes.md`, который следует поместить в каталог со сдаваемым файлом. В `student_notes.md` комментарии, почему соответствующий файл сдается.

Список литературы

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3 ed. The MIT Press, 2009.