Санкт-Петербургский политехнический университет Петра Великого

Институт металлургии, машиностроения и транспорта

Кафедра мехатроники и роботостроения при ЦНИИ РТК

**Курсовая работа**

на тему «Красно-черное дерево»

по дисциплине «Программирование на языках высокого уровня»

Выполнил

Студент гр.33335/2 Мальцева Н.А.

Преподаватель Ананьевский М.С.

<<\_\_\_>>\_\_\_\_\_\_\_\_\_\_2018 г.

Санкт-Петербург

2018

1. **Введение**

Для решения различных задач возникает проблема выбора наиболее подходящей структуры данных. Одной из таких структур являются бинарные деревья. Дерево представляет собой связный граф без петель и циклов. Бинарное дерево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков. Каждый элемент дерева – узел, связи между узлами – ветви дерева. Графическая интерпретация бинарного дерева представлена на рисунке 1.

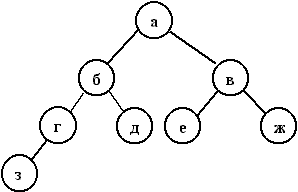


Рисунок 1 – Бинарное дерево

Здесь А – корень дерева, Б – корень левого поддерева, В – корень правого поддерева. Корень дерева расположен на уровне с минимальным значением. Узел Г, который находится непосредственно под узлом Б, называется потомком Б. Узел Б называется родителем или предком узла Г. Каждый узел имеет значение (оно же является и ключом) и ссылки на левого и правого потомка. Если Г находится на уровне i, то Б находится на уровне i-1. Максимальный уровень элемента дерева называется его высотой. Если элемент не имеет потомков, он называется листом дерева. Остальные элементы – внутренние узлы дерева. Число ветвей, которое нужно пройти от корня к узлу x, называется длиной пути к x. Корень имеет длину пути равную 0; узел на уровне i имеет длину пути равную i. Бинарное дерево применяется в тех случаях, когда в каждой точке вычислительного процесса должно быть принято одно из двух возможных решений. Одной из распространенных задач, для которых применяется бинарное дерево, является выполнение заданной операции с каждым элементом дерева или задача обхода дерева.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Сбалансированное бинарное дерево поиска — это бинарное дерево, для каждой вершины которого количество вершин в левом и правом поддереве (высота) различаются не более чем на 1. Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалениями существующих. В случае, если набор элементов, хранящийся в структуре данных фиксирован и нет новых вставок и удалений, то массив предпочтительнее. Потому что поиск можно осуществлять алгоритмом бинарного поиска за то же время, но отсутствуют дополнительные издержки по хранению и использованию указателей. Например, в С++ ассоциативные контейнеры set и map представляют собой сбалансированное бинарное дерево поиска.

В данной работе рассматривается красно-чёрное дерево — это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный». При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными. Графическая интерпретация красно-черного дерева представлена на рисунке 2.

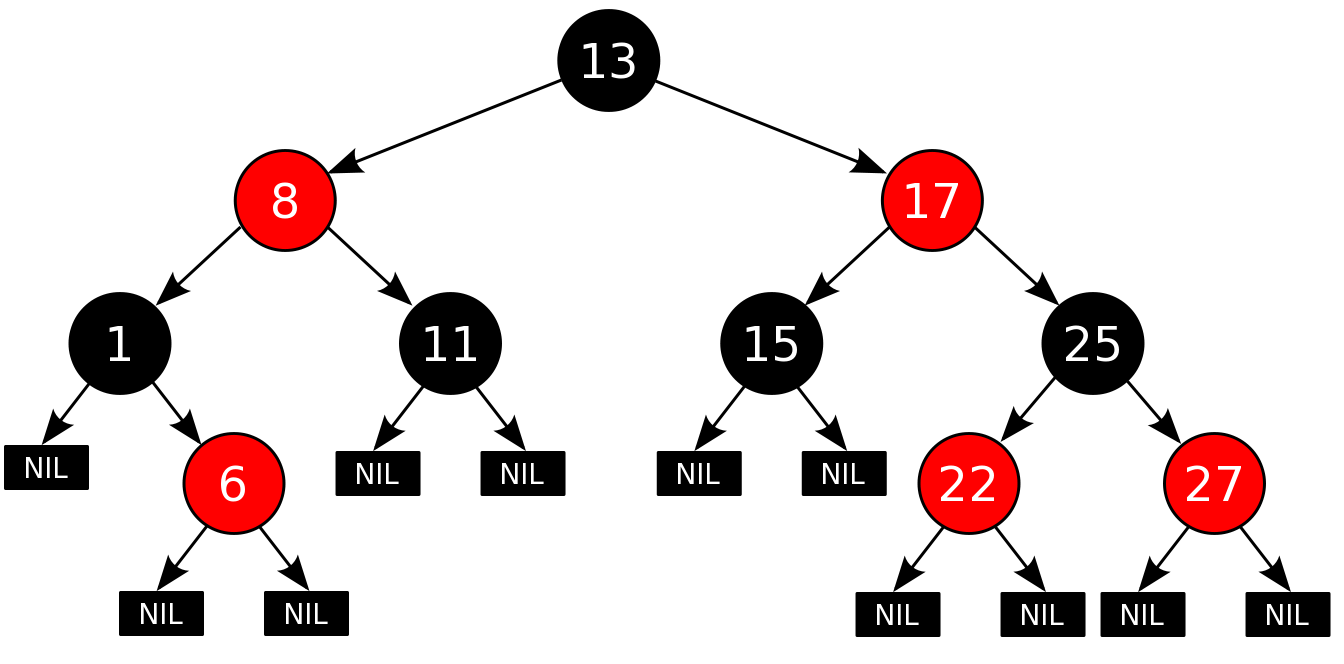


Рисунок 2 – Красно-черное дерево

Красно-черные деревья могут быть использованы для баз данных, в которых хранятся некоторые связанные данные, в которых можно выделить одно из значений (которое более-менее уникально характеризует запись) в качестве ключа. Во многих библиотеках через красно-черные деревья реализуются ассоциативные массивы (абстрактные типы данных, позволяющие хранить пары вида «ключ, значение»).

1. **Постановка задачи**

Изучить и описать словесно алгоритм красно-черного дерева поиска с возможностью добавления, удаления и поиска элементов и его применение. Исследовать эффективность алгоритма по времени работы. Реализовать данный алгоритм на псевдокоде и на языке С или С++.

1. **Свойства красно-черного дерева и их доказательство**

Все листья дерева являются фиктивными и не содержат данных. Число чёрных узлов между корнем и узлом называется чёрной высотой дерева.

**Свойство 1:** Число листьев в дереве с чёрной высотой h не менее чем 2h−1.

Доказательство: При h=1 получаем дерево, в котором чёрными являются только листья, а их больше одного. Иначе пусть корень - чёрный, тогда оба его поддерева имеют чёрную высоту h-1 и, следовательно, не менее чем 2h−2 элементов. Тогда всё дерево имеет более 2h−1 элементов. В случае, если корень красный, то оба его поддерева имеют чёрные корни и чёрную высоту h, то есть, как только что было показано, не менее 2h−1 элементов. Таким образом, дерево будет иметь более 2h элементов.

**Свойство 2:** Число листьев в красно-чёрном дереве высоты h не менее 2(h−1)/2.

Доказательство: Возьмём самый длинный путь в дереве. Всего в нём h+1 узлов. По крайней мере половина узлов чёрные, поскольку двух подряд идущих красных узлов быть не может, а лист чёрный. Поэтому чёрная высота дерева не меньше (h−1)/2, и, по первому утверждению, 2(h−1)/2 элементов.

Отсюда также получаем, что высота дерева не более 2log2N+1, где N - число элементов дерева, т.е. O(logN)

1. **Время работы алгоритма**

Соблюдение свойств красно-черного дерева позволяет обеспечить выполнение операций вставки, удаления и выборки за время O(logN). Доказательство:

Красно-черное дерево имеет черную высоту h. Если путь от корневого узла до листового содержит минимальное количество красных узлов (т.е. ноль), значит этот путь равен h

Если же путь содержит максимальное количество красных узлов (h), то этот путь будет равен 2h.

То есть, пути из корня к листьям могут различаться не более, чем вдвое h’<=2log(N + 1), где h’ — высота поддерева. Этого достаточно, чтобы время выполнения операций в таком дереве было O(logN).

1. **Описание алгоритма**

Для балансировки дерево должно удовлетворять следующим свойствам:

1. Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла дополнительное поле – бит цвета).
2. Корень окрашен в черный цвет.
3. Листья (NULL-узлы) окрашены в черный цвет.
4. Каждый красный узел должен иметь два черных дочерних узла. У черного узла могут быть черные и красные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.
5. Пути от узла к его листьям должны содержать одинаковое количество черных узлов

Каждый узел содержит следующие атрибуты:

1. цвет узла
2. данные
3. левый потомок
4. правый потомок
5. родитель
   1. **Вставка узла в дерево**

Изначально каждый элемент вставляется на позицию NULL – листа и окрашивается в красный цвет. Затем проводится балансировка - проверяется нарушение каждого из свойств дерева.

Проведем балансировку: в общем случае при вставке нового узла возможны три варианта.

* происходит изменение цвета;
* требуется сделать один поворот;
* требуется сделать двойной поворот.

Прежде всего, нужно проверить, есть ли родитель у вставляемого узла – свойство 2. Если его нет, то узел становится корнем и перекрашивается в черный.

Проверка свойства 4: если предок узла – черный, то свойство не нарушается. При этом не нарушается и свойство 5, так как количество черных узлов не изменилось. Поэтому остается проверить следующие 2 случая:

1. Если предок узла – красный, то нужно проверить цвет «дяди». Если он красный, нужно произвести перекрашивание дерева, вплоть до корня, чтобы красно-черные свойства выполнялись. Иллюстрация к этому – рисунок 3.

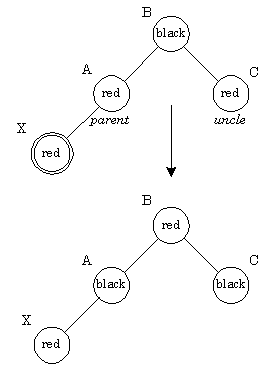


Рисунок 3 – Перекрашивание дерева

1. Если предок узла – красный и "дядя" нового узла оказался черным, то узлы может понадобиться вращать, чтобы скорректировать поддеревья. Иллюстрация этого – рисунок 4.

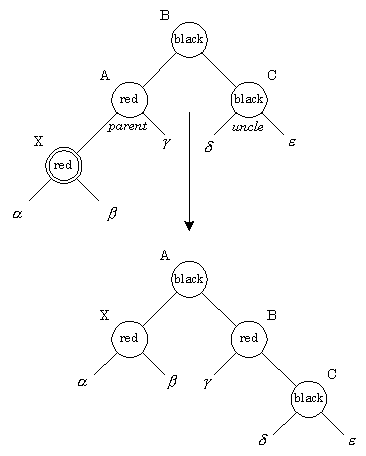


Рисунок 4 – Перекрашивание и вращение дерева

* 1. **Удаление узла**

Здесь может возникнуть 3 случая в зависимости от количества потомков узла:

1. Если у вершины нет потомков, то изменяем указатель на неё у родителя на NULL
2. Если у неё только один потомок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба потомка, то находим вершину со следующим значением ключа. У такой вершины нет левого потомка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого потомка). Иными словами, сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый потомок. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Далее нужно произвести балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим потомка удалённой вершины.

1. Если брат (b) этого потомка красный, то делаем вращение вокруг ребра между отцом (a) и братом, тогда брат становится родителем отца. Перекрашиваем его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас потомок имеет чёрного брата и красного отца.

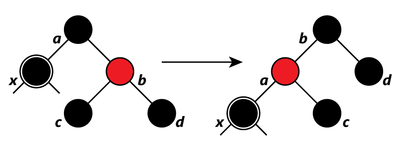


Рисунок 5 – Перекрашивание дерева

2. Если брат текущей вершины был чёрным, то получаем три случая:

а) Оба ребёнка у брата чёрные. Перекрашиваем брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

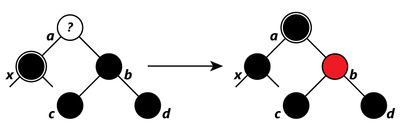


Рисунок 6 – Иллюстрация к пункту а)

б) Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у Х есть чёрный брат с красным правым потомком (рисунок 7).

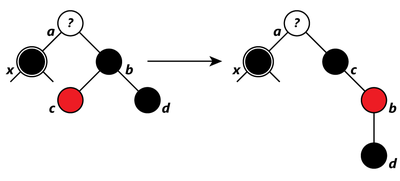


Рисунок 7 – Иллюстрация к пункту б)

в) Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство не нарушается.

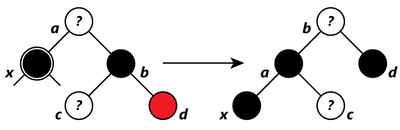


Рисунок 8 – Иллюстрация к пункту в)

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева.

1. **Псевдокод**

Вставка узла:

***func*** *add\_to\_tree (root, key)*

*new\_node = Node(key, red, null, null) // конструктор, в который передаем ключ, цвет, левого и правого ребенка*

***if*** *дерево пустое*

*root= new\_node //узел становится корнем дерева*

*new\_node->предок = null*

***else***

***root = find\_successor(root, new\_node ) //****находим самого «глубокого» потомка, после которого будет вставлен узел*

*balance\_insert\_1(new\_node)*

***func*** *balance\_insert\_1*

***if*** *узел не имеет предка*

*new\_node->черный*

***else***

*balance\_insert\_2*

***func*** *balance\_insert\_2*

***if*** *предок – черный*

***//завершение вставки***

***else***

*balance\_insert\_3*

***func*** *balance\_insert\_3*

***if*** *дядя - красный*

*предок-> черный*

*дядя-> черный*

*дедушка->красный*

*balance\_insert\_1*

***else***

*balance\_insert\_4*

***func*** *balance\_insert\_4*

***if*** *new\_node – правый ребенок и предок – левый ребенок*

*поворот влево относительно предка*

*new node – левый ребенок*

***else***

***if*** *new\_node – левый ребенок и предок – правый ребенок*

*поворот вправо относительно предка*

*new\_node – правый ребенок*

*balance\_insert\_5*

***func*** *balance\_insert\_5*

*предок -> черный*

*дедушка -> красный*

***if*** *new\_node – левый ребенок и предок – левый ребенок*

*поворот вправо относительно дедушки*

***else***

*поворот влево относительно дедушки*

Удаление узла:

***func*** *delete\_node(node)*

***if*** *node – корень дерева*

*node = null*

***else***

***if*** *у node нет потомков*

*предок node->node = null*

***if*** *node->красный*

*node = предок*

***else*** *balance\_delete\_1(node);*

***else***

***if*** *есть хотя бы один потомок*

*предок node->node=потомок*

***if*** *есть 2 потомка*

*node =find\_left\_node->значение //левая нижняя вершина правого поддерева*

*delete\_node (find\_left\_node)*

***func*** *balance\_delete\_1(node)*

***if*** *брат – красный*

*поворот относительно предка*

*брат-> черный*

*предок->красный*

***else*** *balance\_delete\_2(node)*

***func*** *balance\_delete\_2(node)*

***if*** *оба ребенка брата – черные*

*брат->красный*

*предок->черный*

*balance\_delete\_1(предок node) //проверяем следующую вершину, пока не дойдем до корня дерева*

***else*** *balance\_delete\_3(node)*

***func*** *balance\_delete\_3(node)*

***if*** *правый ребенок брата – черный, левый – красный*

*брат->красный*

*левый сын брата ->черный*

*поворот влево относительно брата*

*balance\_delete\_1(предок node) //проверка следующей вершины*

***else*** *balance\_delete\_4(node)*

***func*** *balance\_delete\_4(node) //если правый ребенок брата – красный*

*брат->цвет предка*

*правый ребенок брата->черный*

*предок->черный*

*правый поворот относительно предка*

*balance\_delete\_1(предок node) //проверка следующей вершины*