Структуры данных, алгоритмы и сложность

(*беглое введение-недообзор*)

Практикум МК, 3 курс

Рассказывает:

Подымов Владислав Васильевич

Осень 2017

Вступление

C++, класс std::map, метод find(key), параграф документации:

Сложность этого метода – логарифм относительно размера (size())

Чтобы наиболее полно осознать смысл этого предложения, необходимо понять,

- Для чего формулируется *сложность*
- Что такое *сложность*
- Какая *сложность* считается *большой*, а какая – *маленькой*
- Как можно достичь такой *сложности*

Основная цель лекции – пролить хоть немного света на эти вопросы

Задачи, проблемы и решающие алгоритмы

(***з*1**) У Васи есть 5 яблок. 3 яблока он отдал Коле. Сколько яблок осталось у Васи?

Это **задача**, и у неё есть
- текст условия
- правильный ответ (2 яблока)

(***з*2**) У Васи есть N яблок. K яблок он отдал Коле. Сколько яблок осталось у Васи?

Это тоже задача, но она имеет совсем другое устройство:
- (N – K) – это *вроде бы* правильный ответ к задаче
- настолько же правильным будет и ответ ((N – 2) + K – 2*(K-1)), и много других (*что же такое “правильный ответ”?*)

В ***з*2** “скрыто” **бесконечно много** однотипных задач, аналогичных ***з*1**, по одной для каждой пары чисел(*, подставляемых на места символов*) N и K

Числа N и K – это **параметры** задачи, или **входные данные**

Для каждого набора значений параметров ***з*2** имеет единственный правильный ответ

Задачи, проблемы и решающие алгоритмы

(***з2***) У Васи есть N яблок. K яблок он отдал Коле. Сколько яблок осталось у Васи?

Задача, содержащая параметры, от **бесконечного** разнообразия значений которых зависит правильный ответ, имеет более точные названия: **массовая задача** и **проблема**

Понятие **алгоритма** вам *вроде бы* должно быть хорошо знакомо

Это понятие формулируется в контексте **массовых** задач

В частности, вы можете легко сформулировать “алгоритм”, решающий ***з2***:
1)Вычесть число K из числа N
2)Выдать результат в ответ

Действительно ли это алгоритм?

Действительно ли этот “алгоритм” решает *з*2?

Какова сложность этого “алгоритма”?

Задачи, проблемы и решающие алгоритмы

(***з2***) У Васи есть N яблок. K яблок он отдал Коле. Сколько яблок осталось у Васи?

(**А**):
1) Вычесть число K из числа N
2) Выдать результат в ответ

Смелое утверждение: если разрешить Васе и Коле иметь и передавать любое действительное число яблок, то ***з2*** в таком осмыслении **не имеет** решающего алгоритма

Пояснение:
1) **Тезис Чёрча-Тьюринга**: любой алгоритм решения массовой задачи может быть переформулирован в терминах машины Тьюринга
2) Не существует машины Тьюринга, способной оперировать **произвольными** действительными числами

Задачи, проблемы и решающие алгоритмы

(***з*****2**) У Васи есть N яблок. K яблок он отдал Коле. Сколько яблок осталось у Васи?

(**А**):
1) Вычесть число K из числа N
2) Выдать результат в ответ

Чтобы утверждать, что (**А**) – алгоритм, достаточно

- придумать конечное представление чисел, то есть отобразить каждое число в конечное слово в конечном алфавите
  - **например:** целое неотрицательное число представляется конечной последовательностью символов 0, 1 – двоичной записью этого числа
- придумать, в каком виде разрешено хранить вход и промежуточные состояния вычисления алгоритма
  - **например:** вход особенным образом располагается на одномерной ленте, счётно-бесконечной в обе стороны, и на каждом шаге работы алгоритма на этой ленте записана конечная последовательность символов 0, 1 и обозревается некоторая ячейка
- придумать, какие операции над промежуточными состояниями разрешены алгоритмом, и как именно они работают
  - **например,** (*вспоминаем полное определение машины Тьюринга*)

Задачи, проблемы и решающие алгоритмы

(***з2***) У Васи есть N яблок. K яблок он отдал Коле. Сколько яблок осталось у Васи?

(**А**):
1) Вычесть число K из числа N
2) Выдать результат в ответ

Чтобы утверждать, что алгоритм **решает массовую задачу**, достаточно
- понять, как каждые входные данные задачи отображаются в правильный ответ
- убедиться, что на каждых входных данных задачи алгоритм завершает свою работу и в качестве результата выдаёт правильный ответ

А чтобы понять, какова **сложность** алгоритма, требуется ответить ещё на массу вопросов

Например, используя только “классические” определения сложности, можно утверждать, что (**А**) имеет
- константную сложность (**O(1)**)
- логарифмическую сложность (**O(log n)**)
- линейную сложность (**O(n)**)
- ...

Виды сложности

(**А**):
1) Вычесть число K из числа N
2) Выдать результат в ответ

Сложность алгоритма – это то, сколько … требуется алгоритму (*в худшем случае)* для решения задачи в зависимости от размера входных данных, если мы считаем … алгоритмом и представляем данные так: ...

Машины Тьюринга:
- **сложность по времени**: это то, сколько шагов работы машины Тьюринга требуется для решения задачи, данные которой закодированы двоичной последовательностью (некоторого размера)
- **сложность по памяти**: это то, сколько дополнительных ячеек ленты обозревается машиной Тьюринга при решении задачи, данные которой ...

Схемная сложность:
- **по глубине** (***времени***): это то, какую глубину имеет схема в заданном базисе, выдающая на выходы требуемый ответ, если данные закодированы двоичной последовательностью (некоторого размера), подаваемой на вход схемы
- **по размеру** (***памяти***): это то, сколько функциональных элементов содержит схема в заданном базисе …
- ...

Виды сложности

(**А**):
1) Вычесть число K из числа N
2) Выдать результат в ответ

(**А**) имеет сложность
- O(n) по времени в модели машин Тьюринга
- O(log n) по глубине (времени) в модели схем из функциональных элементов в стандартном базисе (СФЭ)
- **O(1) по времени с точки зрения программиста**
- O(1) по памяти в модели машин Тьюринга
- O(n) по размеру (памяти) в модели СФЭ
- **O(1) по памяти с точки зрения программиста**

Почему понимание сложности программистом так сильно отличается от "классических математических" пониманий, и как определить эту сложность?

Виды сложности

(A):
1)Вычесть число K из числа N
2)Выдать результат в ответ

“**Программистская**” **сложность алгоритма**

- **по времени**: это то, насколько долго потребуется ждать завершения работы программы, **запущенной на процессоре**
- **по памяти**: это то, сколько примитивов языка программирования потребуется использовать помимо заданных входом программы

Эта сложность учитывает особенности устройства современных компьютеров и языков программирования

Например:

- В качестве примитивов часто целые числа **ограниченного диапазона**, трактующиеся как **произвольные** целые числа с явной или неявной поправкой “если не произойдёт переполнение при работе алгоритма”
- процессор содержит некоторый набор допустимых команд, каждая из которых выполняется **за фиксированное время** (число тактов)

Все современные процессоры способны выполнять основные арифметические действия над целыми числами (с заданной шириной двоичной записи; с переполнением) за фиксированое время

Виды сложности

(A):
1) Вычесть число K из числа N
2) Выдать результат в ответ

“**Программистская**” **сложность алгоритма**
- **по времени**: это то, насколько долго потребуется ждать завершения работы программы, **запущенной на процессоре**
- **по памяти**: это то, сколько примитивов языка программирования потребуется использовать помимо заданных входом программы

В дальнейших рассуждениях про сложность алгоритмов будет полагаться, что
- арифметический тип данных содержит все целые числа (переполнение всегда подразумевается неявно)
- **сложность алгоритма** (по времени) – это суммарное число операций над простыми типами данных (в т.ч. арифметическим), производимых алгоритмом
  - т.е. каждая операция имеет константную ненулевую сложность, а остальные составные части работы алгоритма имеют сложность 0

Алгоритмы будут записываться в C-подобном псевдокоде

Список

Есть много вариантов определения понятия “список”, и много разновидностей списков

С этого момента и до конца будем стараться придерживаться определений, приближенных к stl C++ (хотя и отличающихся некоторыми несущественными деталями), и давать эти определения местами не очень строго, но как можно более коротко

Определения “в духе stl” имеют декларативный характер:

- описывается набор концепций, при помощи которых можно коротко, ясно и наглядно ввести интерфейс работы с определяемым понятием и пояснить, как именно этот интерфейс должен работать
- предъявляется набор требований к конкретному устройству реализации понятия, которых должна придерживаться любая реализация: структуры данных этой реализации и алгоритмы работы с этими структурами должны быть выбраны так, чтобы все требования соблюдались
- конкретное устройство реализации опускается: оно может быть любым, подходящим под требования

Список

Концептуальное устройство (двусвязного) **списка**:

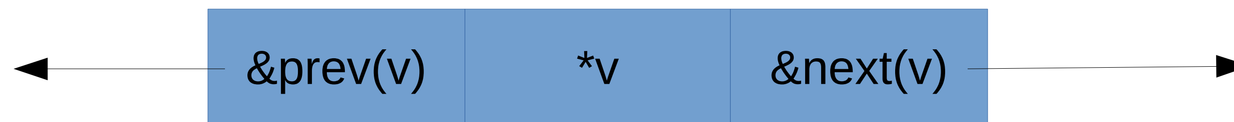
- он содержит конечную последовательность элементов заданного типа, каждый элемент хранится в уникальной вершине списка, порядок вершин – это порядок хранимых элементов
- длина списка – это количество хранимых элементов
- вершины можно проверять на совпадение (“**V1 == V2**” = “V1 и V2 – одна и та же вершина”)
- по текущей вершине V можно получить хранящийся в ней элемент ***V**
- по текущей вершине V можно получить следующую вершину **next(V)** и предыдущую вершину **prev(V)**, если они существуют
- в списке выделены специальные вершины: первая вершина **L.head()**, а также последняя вершина **L.end()**, которая не хранит никакой элемент и располагается после всех остальных вершин (хранящих элементы)
- в список можно вставлять элементы, и из него можно удалять элементы:
  - **L.insert(V, e)** – вставить элемент e перед вершиной V
  - **L.erase(V)** – удалить вершину V

Требования:

- Все процедуры интерфейса (*V, next(V), prev(V), L.head(), L.end(), L.insert(V, e), L.erase(V)) имеют константную сложность **относительно длины списка**
- Добавление/удаление вершины не изменяет другие вершины списка

Список

Типичная реализация вершины v списка (кроме пустой вершины):



Выбор структуры данных, как правило, непосредственно влияет на устройство некоторых (базовых) процедур интерфейса

Остальные процедуры можно описать “высокоуровнево”: на основе базовых

В конечном итоге важна не реализация как таковая, а **гарантии**, предоставляемые реализацией по отношению к процедурам

Например, гарантии типичной реализации по отношению к базовым процедурам *V, next(V), prev(V): **эти процедуры имеют константную сложность**

Следует иметь в виду, что даже самые маленькие детали выбора структур данных важны в реализации, и не все такие детали предельно очевидны

Например: **что такое список?**

Если отождествить список L с его головой L.head(),
то **не получится его реализовать согласно всем требованиям**

Для соблюдения упомянутых ранее требований достаточно сказать, что писок L – это пара **(L.head(), L.end())**

Список

Типичная реализация вершины v списка (кроме пустой вершины):

L == (H, E)

Алгоритм вычисления головы списка: L.head() – с константной **сложностью**:
- return H;

Алгоритм вычисления элемента L.end() с константной **сложностью**:
- return E;

Алгоритм вставки: L.insert(V, e) – с константной **сложностью**:
- new node V’; *V’ = e; next(V’) = V; prev(V’) = prev(V);
- if(V == H) L = (V’, E);
- if(V != H) next(prev(V)) = V’;

Алгоритм удаления: L.erase(V) – тоже можно реализовать с константной **сложностью** (*но теперь это совсем очевидно, и все вы знаете, как это сделать*)

А можно ли соблюсти все требования, если заменить “программистскую”сложность на сложность по времени в модели машин Тьюринга?

Список

Типичная реализация вершины v списка (кроме пустой вершины):

L == (H, E)

Ещё немного “подводных камней”

Останется ли сложность процедур вставки и удаления элемента константной, если размещение выбранного элемента в вершине имеет высокую сложность?

В простых случаях элемент, вставляемый в список, никак не связан с этим списком

В этих случаях достаточно **внимательно** посмотреть в определение сложности и убедиться, что проблем не возникает, т.к. рассматривается только зависимость относительно длины списка

Пример “непростого” случая: в C++ несложно создать элемент, при размещении которого в вершине списка будет совершаться обход всех вершин списка

При рассуждении об оценках сложности обычно неявно присутствует требование, согласно которому подобные “непростые” случаи не рассматриваются:
любая операция над элементами хранимого типа
имеет константную сложность

Вектор

Концептуальное устройство вектора V:
- он содержит конечную последовательность элементов заданного типа
- длина вектора – это количество элементов последовательности
- каждый элемент можно получить по его индексу – целому неотрицательному числу i: **V[i]** (*полагаем, что элементы индексируются с ноля*)
- В конец вектора можно вставлять элементы, и с конца вектора можно удалять элементы:
  - **L.push_back(e)** – добавить элемент e в конец вектора
  - **L.pop_back()** – удалить последний элемент из вектора

Требования:
- Процедура индексации (V[i]) имеет константную сложность относительно длины V
- **А как быть с добавлением и удалением элементов?**

Если вспомнить типичную реализацию вектора (непрерывный массив данных заданного размера в памяти), то немедленно возникает такое требование:
процедуры добавления и удаления имеют сложность,
линейную относительно длины вектора
(*т.к. в худшем случае может потребоваться*
размещение всех элементов вектора в новом массиве данных)

Это *не очень хорошее* требование: на практике элемент
- добавляется в вектор за константное время *в большинстве случаев*
- удаляется из вектора за константное время **всегда**

Вектор

Снижение сложности удаления элемента из вектора получается тем же “жонглированием деталями реализации”, как и для списков:

Вектор V – это система **(P, L, N)**, где
- P – указатель на начало массива данных, выделенного для элементов вектора
- L – то, сколько элементов вмещается в выделенный массив данных
- N – длина вектора

Алгоритм индексации: V[i] – с константной **сложностью**:
- return *(P + i); - где прибавление числа к адресу происходит так же, как в C

Алгоритм удаления: V.pop_back() – с константной **сложностью**:
- V = (P, L, N-1); (*и стереть последний элемент*)

Процедура V.push_back(e) всё равно будет иметь линейную сложность

Чтобы справиться с этим “недоразумением”, следует точнее сформулировать само понятие “сложности”

Вектор

Сложность в худшем случае – это сложность, о которой шла речь до этой фразы

Этот вид сложности далеко не всегда подходит для рассуждений о том, за какое время будет завершаться работа алгоритма *как правило*, или *в большинстве случаев*, или *в тех случаях, для которых прежде всего предназначен алгоритм*, или ...

Есть и другие виды сложности, более подходящие для таких рассуждений, например:

- **типичная сложность**: все входные данные разделяются на типичные и нетипичные, и приводится оценка сложности для типичных входных данных
- **средняя сложность**: предполагается семейство вероятностных распределений, описывающее частоту появления каждых конкретных входных данных заданного размера; средняя сложность – это мат.ожидание сложности с учётом предполагаемого распределения
- **усреднённая сложность**: средняя сложность в случае, когда все данные одного размера равновероятны (*семейство равномерных распределений*)

Все эти виды сложности по-своему полезны, но не подходят для нашего случая: анализа сложности процедуры V.push_back(e)

Вектор

Амортизированная сложность алгоритма определяется так:

- определяется начальное состояние рассматриваемого объекта (*например, (P, 1, 0)*)
- сложность формулируется для набора процедур (*например, для V.push_back(e)*)
- сложность определяется относительно **числа процедур** n, последовательно вызываемых из начального состояния
- пусть T(n) – **худшее время** последовательного выполнения n процедур
- амортизированная сложность алгоритма – это функция f(n) = T(n)/n

Амортизированная сложность тем ниже, чем реже при фактической последовательной работе с объектом встречаются “худшие случаи”

Требования к процедуре V.push_back(e) для вектора V:

- сложность этой процедуры в худшем случае линейна относительно длины вектора
- амортизированная сложность этой процедуры константна

Алгоритм вставки: (P, L, N).push_back(e) – с требуемой **сложностью**:

- if(L > N) {*(P + N) = e; N++;}
- if(L <= N):
  - P’ = new T[2*L];
  - P’[0:N] = P[0:N]; delete P;
  - V = (P’, 2*L, N); V.push_back(e);

А почему амортизированная сложность этого алгоритма константна?

Задача поиска

Словарь и **множество** – это (в числе других) два общеизвестных и часто используемых в программировании понятия, для которых требуются структуры данных и алгоритмы над ними, позволяющие эффективно рассуждать о таких наборе концепций и интерфейсе:

- на каждом шаге работы имеется совокупность вершин, в которых хранятся объекты заданного типа (в множестве – элементов, в словаре – ключей)
- в эту совокупность нужно уметь добавлять вершины с новыми объектами
- из этой совокупности нужно уметь удалять вершины
- в этой совокупности требуется уметь **искать** вершину, хранящую заданный объект

Чуть более строго, основные процедуры, требуемые для решения **задачи поиска** объекта в обновляемой совокупности, выглядят так:

- **S.add(e)** – добавить e в совокупность, хранимую в S
- **S.erase(v)** – удалить вершину v из S
  - *вроде бы это всё у вас было, так что S.erase(v) обсуждать не будем*
- **S.find(e)** – вернуть вершину, хранящую объект e, если она есть, или сказать, что такой вершины нет, если её нет

Структуры данных, наиболее подходящие для решения этой задачи,
сложность процедур добавления, удаления и поиска,
и дополнительные процедуры интерфейса зависят от особенностей типа объектов, хранящихся в совокупности

Бинарное дерево поиска

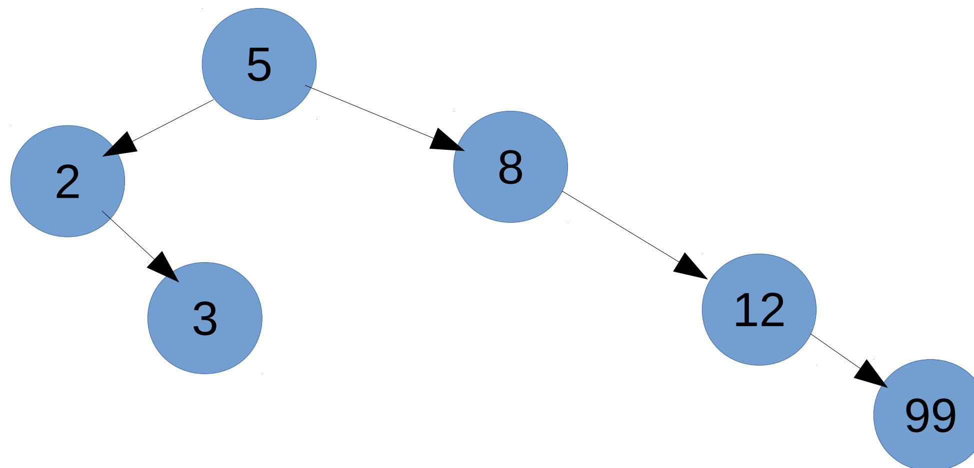
Бинарное дерево поиска T – это ориентированное корневое дерево со степенью ветвления не более 2, в вершинах которого хранятся сравнимые (<) элементы, такое что для каждой вершины v,

- кроме корня, существует родитель **par(v)**: вершина, из которой ведёт дуга в v
- не являющейся листом, существует левый ребёнок **left(v)**: вершина, в которую ведёт первая (левая) дуга из v, причём *left(v) < *v
  - ***v** – это элемент, хранящийся в вершине v
- кроме тех, для которых не существует левый ребёнок, может существовать правый ребёнок **right(v)**: вершина, в которую ведёт вторая (правая) дуга из v, причём *right(v) > *v

Для краткости “вершина не существует” отождествляем с “вершина **== 0**”

Гарантии: вычисление родителя и каждого из детей имеет константную сложность

Бинарное дерево поиска T – это (*указатель на*) его корень **T.root()**



Бинарное дерево поиска

Алгоритм поиска: T.find(e)

- return T.find(e, T.root());

Алгоритм поиска в поддереве: T.find(e, v)

- if(v == 0) return 0;
- else if(e < *v) return T.find(e, left(v));
- else if(*v < e) return T.find(e, right(v));
- else return v;

Сложность поиска **относительно числа n вершин в дереве**:

- в худшем случае: O(n)
- амортизированная: тоже O(n)

И чем это лучше использования обычного списка?

Сложность поиска **относительно глубины d дерева**:

- в худшем случае: O(d)

В лучшем случае d = O(log n), но для этого нужно, чтобы при добавлении элементов глубина сохранялась достаточно малой

Бинарное дерево поиска

Алгоритм поиска следующего элемента: next(v) – возвращает вершину, содержащую следующий по порядку элемент относительно *v

- if(right(v) != 0) return minimum(v);
- else return next_parent(v);

Алгоритм поиска вершины с минимальным элементом в поддереве: minimum(v)

- if(left(v) != 0) return minimum(left(v));
- else return v;

Алгоритм поиска следующего элемента среди родителей: next_parent(v)

- if(par(v) == 0) return 0;
- else if(v == left(par(v)) return par(v);
- else return next_parent(par(v));

Сложность поиска следующего элемента относительно глубины d дерева:

- в худшем случае: O(d)

Бинарное дерево поиска

Алгоритм вставки: T.add(e)

- if(T.root() == 0) T = Node(e);
  - Node(e) – новая вершина, хранящая e
- if(T.root() != 0) add(e, T.root());

Алгоритм обхода со вставкой: add(e, v)

- if(e < *v):
  - if(left(v) == 0) left(v) = Node(e);
  - if(left(v) != 0) add(e, left(v));
- else if(*v < e):
  - if(right(v) == 0) right(v) = Node(e)
  - if(right(v) != 0) add(e, right(v));

Сложность вставки относительно глубины d дерева:
- в худшем случае: O(d)

Но что-то не сходится: с таким алгоритмом вставки в худшем случае глубина d равна числу вершин n, а нам хочется иметь равенство d = O(log n)

Общий подход для достижения такого равенства:
- хранить в вершине дополнительную информацию (высота поддерева, цвет, …)
- в конец процедуры add(e, v) добавить **балансировку дерева**:
  - **if(bad(v)) rebalance(v);**
  - определив нужным образом bad и rebalance, можно получить **АВЛ-деревья**, **красно-чёрные деревья**, ...

Красно-чёрное дерево

Красно-чёрное дерево (**КЧД**) – это бинарное дерево поиска со следующими свойствами:

- каждая вершина окрашена либо в красный, либо в чёрный цвет
- корень дерева окрашен в чёрный цвет
- если left(v) == 0 или right(v) == 0, то соответствующий (*отсутствующий*) ребёнок полагается раскрашенным в чёрный цвет
- оба ребёнка красной вершины раскрашены в чёрный цвет
- для каждой вершины v верно следующее: все пути от v до достижимых листьев содержат одинаковое число чёрных вершин

Утверждение: глубина d и число вершин n любого КЧД соотносятся так:

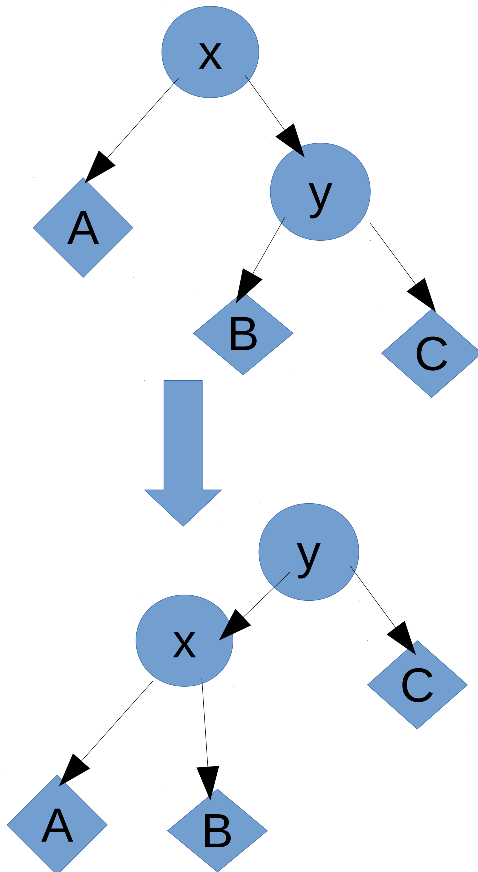
$$d = O(\log n)$$

Единственная нетривиальная (и отсутствующая в рассказе про бинарное дерево поиска) часть работы с КЧД – это **балансировка**

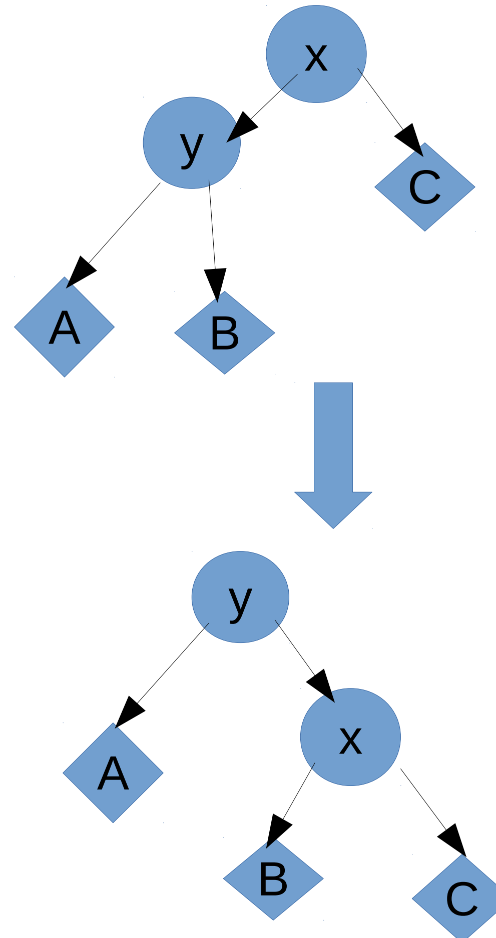
Красно-чёрное дерево

Основа балансировки КЧД – это два вида **вращений**:

Поворот x влево



Поворот x вправо



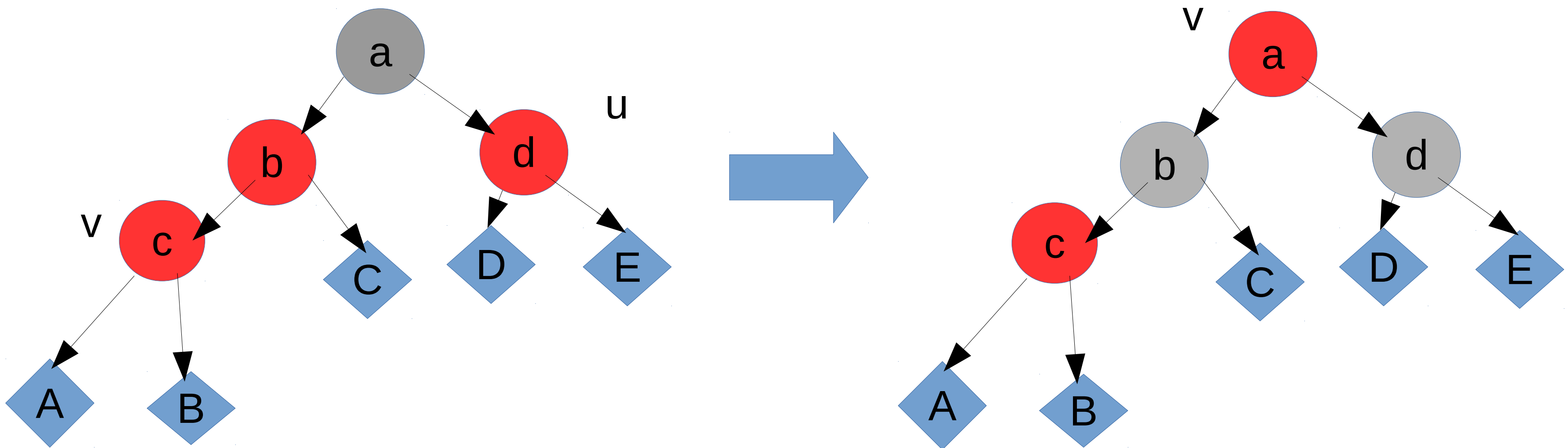
Красно-чёрное дерево

Балансировка КЧД

Новодобавленная вершина раскрашивается в красный цвет и объявляется текущей

Пока родитель текущей вершины v является красным, делается следующее:

- Если правый дядя u вершины v – красный, то
  - u и родитель v перекрашиваются в чёрный цвет
  - дед v перекрашивается в красный цвет
  - дед v объявляется текущей вершиной



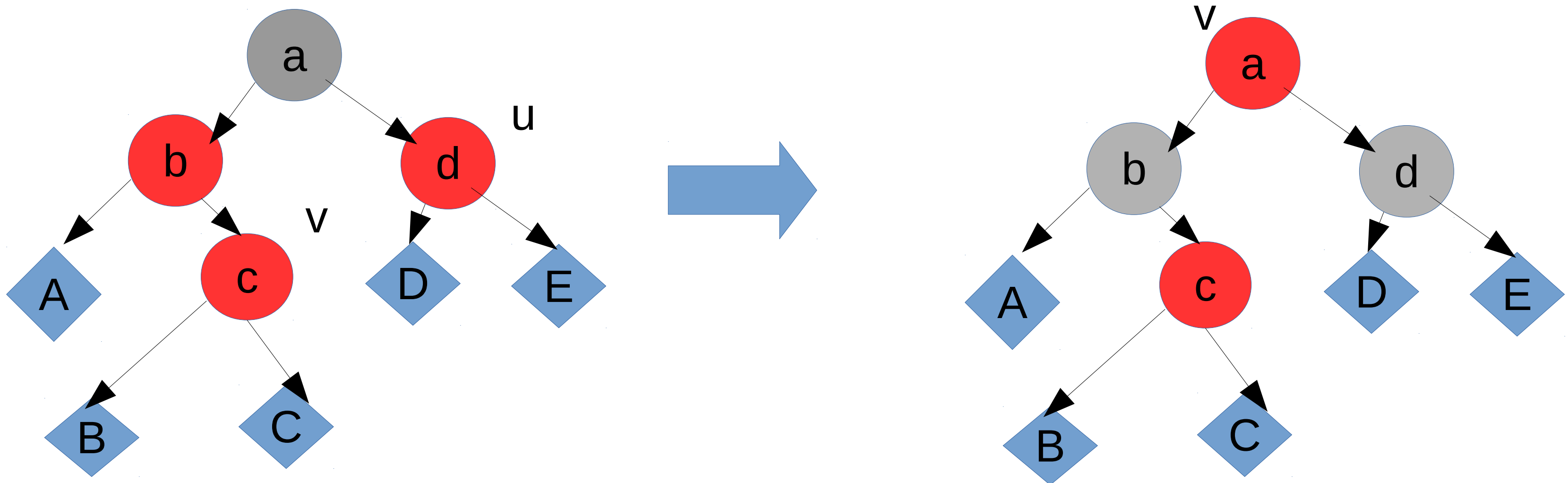
Красно-чёрное дерево

Балансировка КЧД

Новодобавленная вершина раскрашивается в красный цвет и объявляется текущей

Пока родитель текущей вершины v является красным, делается следующее:

- Если правый дядя u вершины v – красный, то
  - u и родитель v перекрашиваются в чёрный цвет
  - дед v перекрашивается в красный цвет
  - дед v объявляется текущей вершиной



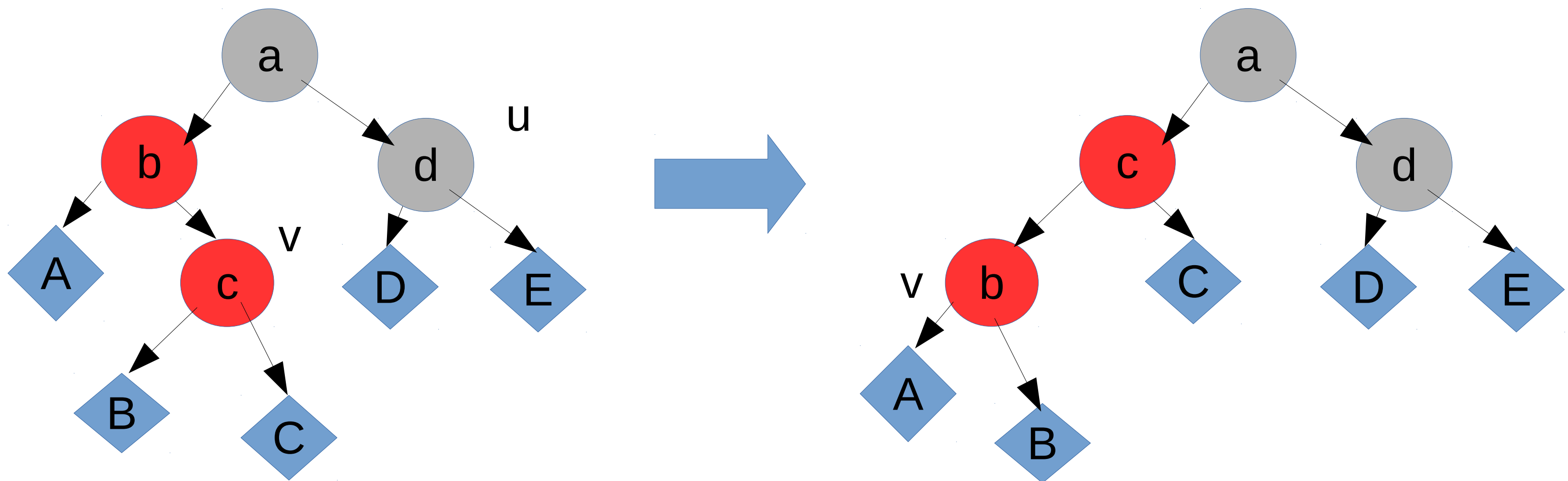
Красно-чёрное дерево

Балансировка КЧД

Новодобавленная вершина раскрашивается в красный цвет и объявляется текущей

Пока родитель текущей вершины v является красным, делается следующее:

- Если правый дядя u вершины v – серый, и вершина v – правый ребёнок, то
  - родитель v вращается влево
  - бывший родитель, а ныне ребёнок v объявляется текущей вершиной



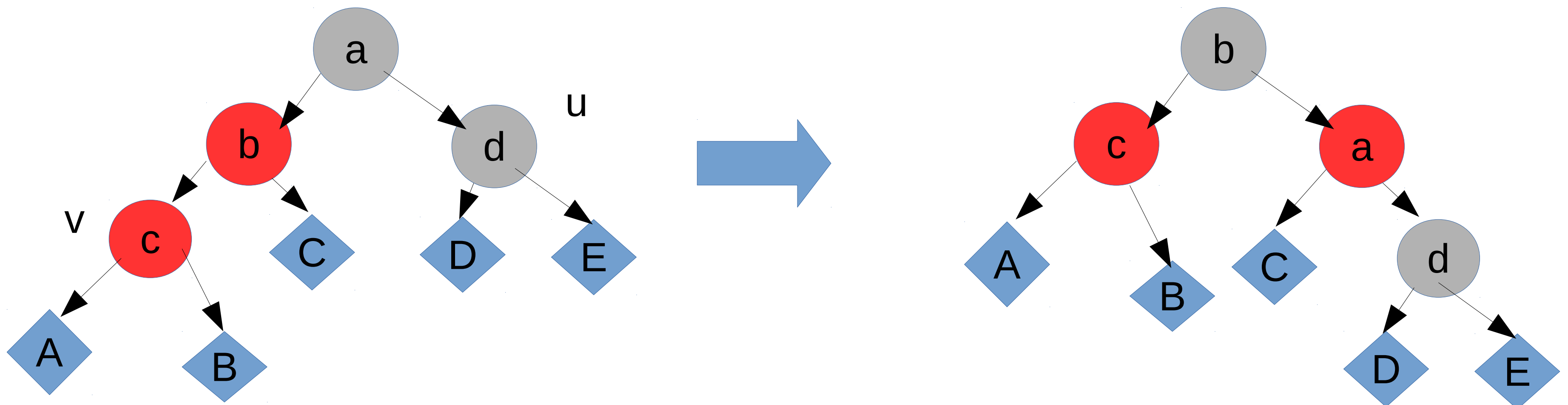
Красно-чёрное дерево

Балансировка КЧД

Новодобавленная вершина раскрашивается в красный цвет и объявляется текущей

Пока родитель текущей вершины v является красным, делается следующее:

- Если правый дядя u вершины v – серый, и вершина v – левый ребёнок, то
  - родитель v перекрашивается в чёрный цвет
  - дед v перекрашивается в красный цвет
  - дед v вращается вправо
  - череда вращений завершается



Красно-чёрное дерево

Балансировка КЧД

Новодобавленная вершина раскрашивается в красный цвет и объявляется текущей

Пока родитель текущей вершины v является красным, делается следующее:

- Если у вершины v есть левый дядя, то вращения происходят симметрично

Если по завершении всех вращений корень оказался красным,
то он перекрашивается в чёрный цвет

Балансировка завершается

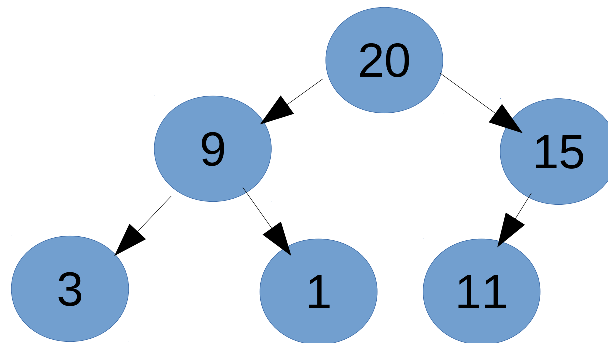
А правда ли это работает?

А если работает, то сколько раз вращаются вершины?
(*то есть какова сложность балансировки?*)

Пирамида

(Max-)**пирамида** отличается от бинарного дерева поиска следующим:
- свойства *left(v) < *v и *v < *right(v) заменяются на *left(v) <= *v и *right(v) <= *v
- длины путей от корня до любых двух листов различаются не более чем на 1
- длина пути от корня до более левого листа не меньше длины пути до более правого листа

Иными словами, пирамида – это почти совершенно сбалансированное бинарное дерево убывающих путей, последний ярус которого заполняется слева направо



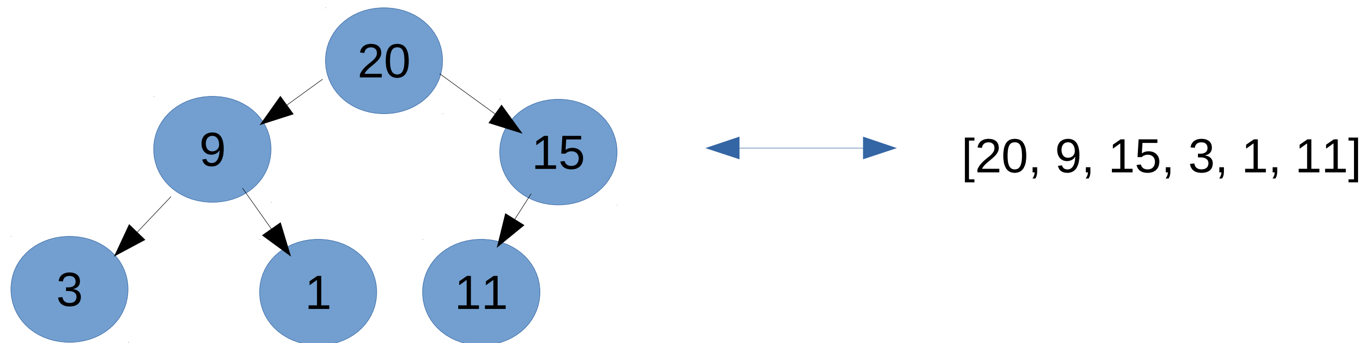
Пирамида применяется в тех задачах, в которых требуются только
- добавление элемента в совокупность
- поиск и удаление максимального элемента совокупности

Примеры таких задач:
- **управление приоритетами** (хранимая совокупность = мультимножество приоритетов)
- **сортировка**

Пирамида

Пирамида H – это **линеаризуемый объект**: её можно представить вектором V размера, равного числу вершин H, такого что

- Вершины v в H взаимно-однозначно соответствуют индексам i(v) в V
- i(H.root()) == 0
- i(left(v)) == v[2 * i(v) + 1]
- i(right(v)) == v[2 * i(v) + 2]
- i(par(v)) == (i(v) – 1)/2



Такое представление пирамиды позволяет эффективно реализовать содержательно описанный ранее интерфейс и даже нечто большее:

- **V.add(e)** – добавление элемента в мультимножество пирамиды
- **V.erase()** – удаление одного из максимальных элементов из пирамиды
- **V.max()** – вычисление максимального элемента пирамиды
- **V.heapify()** – переупорядочивание элементов вектора V в пирамиду

Пирамида

Алгоритм вычисления максимального элемента: V.max()

- return V[0];

Сложность: O(1) (*относительно числа N элементов кучи*)

Алгоритм добавления элемента: V.add(e), где V: (P, L, N)

- V.push_back(e); – добавить e **в следующий лист**
- V.lift(N-1);

Алгоритм восстановления пирамиды после добавления листа: V.lift(i)

- while(i > 0 && V[i] > V[(i-1)/2]):
  - swap(V[i], V[(i-1)/2]);
  - i = (i-1)/2;

Сложность восстановления пирамиды: O(log N)

Сложность добавления элемента:
- в худшем случае: O(N)
- амортизированная: O(log N)

Пирамида

Алгоритм удаления максимального элемента: V.erase(), где V: (P, L, N)

- V[0] = V[N-1];
- V.pop_back();
- V.fix(0);

Алгоритм восстановления подпирамиды с “плохим” корнем: V.fix(i)

- largest = i;
- if(V[2*i+1] > V[i]) largest = 2*i+1;
- if(V[2*i+2] > V[largest]) largest = 2*i+2;
- if(largest != I):
  - swap(V[i], V[largest]);
  - V.fix(largest);

Сложность восстановления подпирамиды: O(log N)

Сложность удаления элемента: O(log N)

Пирамида

Алгоритм построения пирамиды по произвольному вектору: V.heapify(), где V: (P, L, N)

- for(i : {N/2 – 1, …, 0}):
  - V.fix(i);

Эта процедура

- обходит ярусы дерева от предпоследнего (вершины перед листьями) до первого (корня)
- внутри яруса обходит вершины справа налево, игнорируя листья
- восстанавливает поддерево, “вырастающее” из обозреваемой вершины, до пирамиды, индуктивно используя тот факт, все поддеревья являются пирамидами

Сложность построения пирамиды: O(N)

А почему не O(N * log N)?

Пирамидальная сортировка

Вход: произвольный вектор V сравнимых (<) элементов, V: (P, L, N)

Выход: вектор, содержащий исходное мультимножество элементов, совпадающее с мультимножеством V, упорядоченное по неубыванию и располагающееся в той же области памяти, что и данные исходного вектора

Алгоритм пирамидальной сортировки: **V.heapsort()**

- V.heapify(); – построить пирамиду, сложность: O(N)
- for(i : {N-1, …, 1}): – O(N) итераций сортировки
  - swap(V[0], V[i]);
  - (P, L, i).fix(0); – сложность каждой итерации сортировки – O(log n)

Сложность алгоритма пирамидальной сортировки: O(N * log N)

А это много или мало?

Другие достоинства алгоритма: размер дополнительной памяти – O(1)

Недостатки алгоритма: … … ...

Пирамидальная сортировка

Пример: [9 1 11 3 20 15].heapsort()

Создание пирамиды:

[9 1 11 3 20 15] fix: 11

[9 1 15 3 20 11] fix: 1

[9 20 15 3 1 11] fix: 9

[20 9 15 3 1 11]

Это пирамида

Сортировка пирамиды:

[20 9 15 3 1 11] swap: 20<->11

[11 9 15 3 1 20] fix: 11

[15 9 11 3 1 20] swap: 15<->1

[1 9 11 3 15 20] fix: 1

[11 9 1 3 15 20] swap: 11<->3

[3 9 1 11 15 20] fix: 3

[9 3 1 11 15 20] swap: 9<->1

[1 3 9 11 15 20] fix: 1

[3 1 9 11 15 20] swap: 3<->1

[1 3 9 11 15 20] Конец