Всестороннее изучение структур данных с использованием C# 2.0

Часть IV: Строим более лучшее бинарное дерево поиска

Эта статья, четвертая из цикла статей, начинается с быстрого рассмотрения АВЛ деревьев и красно-черных деревьев, которые являются двумя различными структурами данных самобалансирующихся бинарных деревьев поиска. Оставшаяся часть статьи изучает структуру данных списков пропуска. Списки пропуска это остроумная структура данных, которая превращает связанный список в структуру данных, которая предлагает такое же время выполнения, как и более сложные структуры данных самобалансирующихся деревьев. Как мы увидим, списки пропуска делают свою магическую работу давая каждому элементы в связанном списке случайную «высоту».

1 Введение

В третьей части нашей серии статей мы рассмотрели обобщенную структуру данных дерева. Дерево – это структура данных, которая состоит из узлов, в которой каждый узел имеет какое-нибудь значение и произвольное число узлов потомков. Деревья это широко распространенная структура данных, поскольку многие проблемы из реального мира проявляю поведение подобное деревьям. Например, любой вид иерархического взаимоотношения между людьми, предметами или объектами может быть смоделировано с помощью деревьев.

Бинарное дерево это особый вид дерева, такой в котором каждый узел имеет не более двух потомков. Бинарное дерево поиска или BST, это бинарное дерево, чьи узлы организованы таким образом, что для каждого узла n, все узлы левого поддерева n имеют значения меньше, чем n, а все узлы правого поддерева n имеют значения больше, чем n. Как мы это уже обсуждали, в общем случае BST предлагает асимптотическое время выполнения $\log_2 n$ для операция вставки, удаления и поиска.

Недостаток BST состоит в том, что в наихудшем случае их асимптотическое время выполнения снижается до линейного времени. Это происходит тогда, когда новые элементы добавляются в BST в отсортированном или почти отсортированном порядке. В таком случае BST работает не лучше, чем обычный массив. Как мы обсуждали в конце Части III, существуют самобалансирующиеся бинарные деревья поиска, такие, которые обеспечивают, не зависимо от порядка в котором в него добавляются данные, время выполнения порядка $\log_2 n$. В этой статье мы вкратце обсудим два самобалансирующихся вида бинарных деревьев поиска: АВЛ деревья и красно-черные деревья. Следом за этим мы, проведем детальное изучение списков пропуска (skip lists). Списки пропуска это действительно приятные структуры данных, поскольку их гораздо легче реализовать, чем АВЛ деревья или красно-черные деревья, гарантирующие при этом время выполнения $\log_2 n$.

Comment [T1]: Эта статья подразумевает, что вы читали Часть III. Если же вы не читали третью часть этой серии статей, то мы самым настоятельным образом рекомендуем сделать это перед прочтением Части IV.

2 Самобалансирующееся бинарное дерево поиска

Давайте вспомним, что новые узлы добавляются в бинарное дерево поиска в виде листьев. То есть, добавление узла в бинарное дерево поиска требует прохождения пути вниз дерева, двигаясь вправо или влево в зависимости от результата сравнения значения в

текущем узле и значения вставляемого узла, до тех пор пока не будет достигнут тупик. В этом месте новый вставляемый узел вставляется в дерево в месте этого тупика. Рисунок 1 иллюстрирует процесс вставки нового узла в BST.

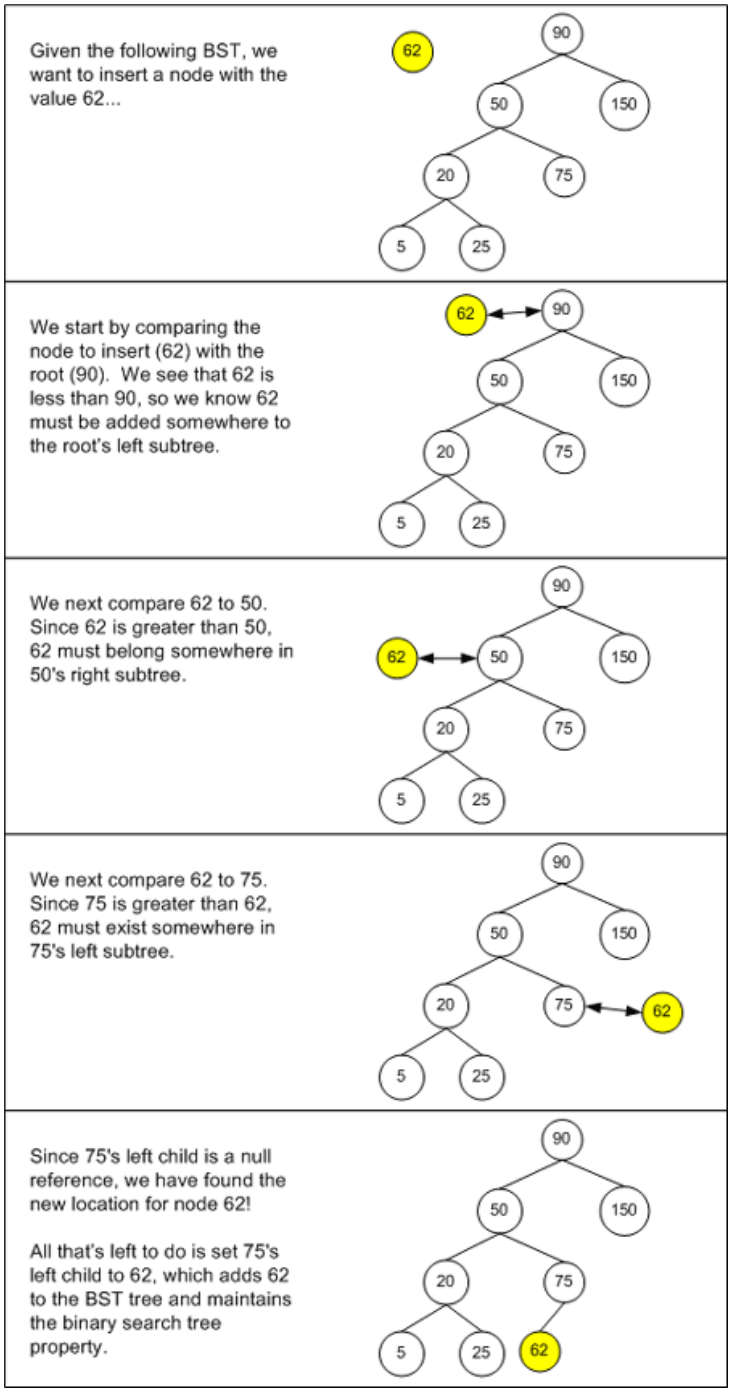

Рисунок 1. Вставка нового узла в BST

Пусть дано следующее BST и мы хотим вставить в него узел со значением 62 ...

Мы начинаем со сравнения вставляемого узла (62) со значением корня (90). Мы видим, что 62 меньше, чем 90, поэтому мы знаем, что 62 должно быть добавлено в левое поддерево корня.

Далее мы сравниваем 62 с 50. Так как 62 больше, чем 50, то 62 должно находиться где-то в правом поддереве 50.

Затем мы сравниваем 62 с 75. Так как 75 больше, чем 62, то 62 должно находиться где-то в левом поддереве 75.

Далее мы сравниваем 62 с 75. Так как 75 больше, чем 62, то 62 должно существовать где-то в левом поддереве узла 75.

Так как левый потомок узла 75 является пустой ссылкой, то мы нашли новое место для узла 62!
Все, что осталось сделать это назначить левого потомка узла 75 узлу 62, что добавляет узел 62 в дерево и сохраняет свойство бинарного дерева поиска.

Как показывает Рисунок 1, когда выполняется сравнение в текущем узле, то вставляемый узел двигается вниз по левому пути, если его значение меньше значения текущего узла, и вниз по правому пути, если его значение больше, чем значение текущего. Поэтому структура BST зависит от порядка в котором вставляются узлы. Рисунок 2 показывает BST после того, как в него были добавлены значения 20, 50, 90, 150, 175, и 200. В частности, эти узлы были добавлены в порядке увеличения. Результатом является BST не имеющее ширины. То есть, его топология состоит из узлов выстроенных в одну линию, а не развернутых веером.

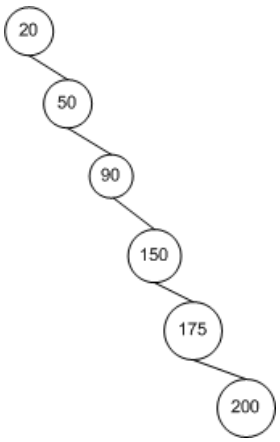

Рисунок 2. BST после того, как в него были добавлены величины 20, 50, 90, 150, 175, и 200

BST — которые показывают сублинейное время выполнения для вставки, удаления и поиска — работают оптимальным образом тогда, когда их узлы организованы веерообразным способом. Это происходит потому, что при поиске узла в BST, каждый шаг вниз по дереву уменьшает число узлов, которые потенциально нужно проверить, вдвое. Однако, когда топология BST аналогична представленной на Рисунке 2, то время выполнения операций над BST очень близко к линейному времени потому, что каждый шаг вниз по дереву уменьшает число узлов для поиска только на один. Чтобы увидеть почему это так, давайте рассмотрим, что произойдет при поиске определенного значения, скажем 175. Начиная с корня, 20, мы должны передвигаться вниз через каждого правого потомка до тех пор пока мы не достигнем 175. То есть в данном случае нет никаких узлов, которые не нужно проверять на каждом шаге. Поиск в BST подобном представленному на Рисунке 2 идентично поиску в массиве — каждый элемент должен быть в точности проверен один раз. Поэтому такая структура BST покажет линейное время поиска.

Очень важно понимать, что время выполнения операций над BST связано с высотой BST. Высота дерева определяется как длина самого длинного пути, начиная с корня. Высота дерева может быть определена рекурсивно следующим образом:

- Высота узла без потомков равна 0.
- Высота узла с одним потомком равна высоте этого потомка плюс один.
- Высота узла с двумя потомками равна единице плюс высота большего из потомков.

Чтобы вычислить высоту дерева, начните с листьев и назначьте им вес 0. Затем продвигайтесь вверх по дереву используя три правила для вычисления высоты в каждом узле-родителе каждого листа. Продолжить таким образом до тех пор пока не будет помечен каждый узел. Тогда высота дерева это высота корневого узла. Рисунок 3 показывает несколько бинарных деревьев, для которых вычислены высоты в каждом узле. Для того чтобы попрактиковаться, потратьте минуту времени, чтобы вычислить высоту самому и проверить, что ваши вычисления совпадают с числами на рисунке.

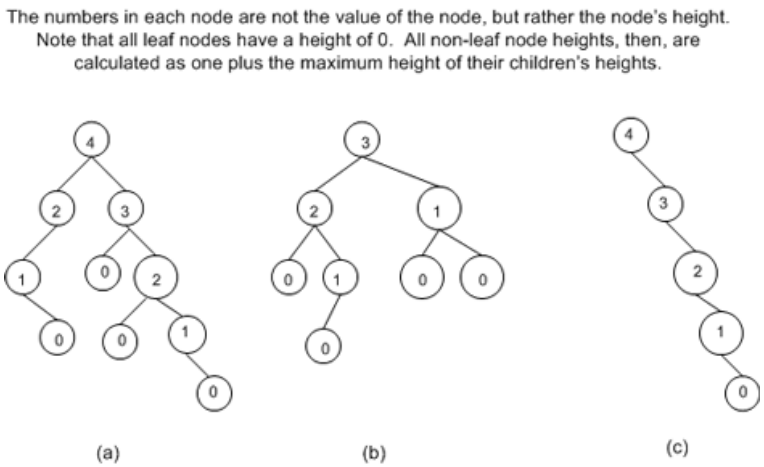

Рисунок 3. Пример бинарных деревьев с высотами вычисленными в каждом узле

Числа в узлах не являются значениями узлов, а представляют высоту узла. Отметьте, что все листовые узлы имеют вес 0. Высоты всех не листовых узлов вычисляются как единица плюс максимальная высота их потомков.

BST показывает время выполнения $\log_2 n$ тогда, когда его высота, определенная в терминах числа узлов n в дереве, лежит в пределах «пола» $\log_2 n$. (Функция «пол» числа x это наибольшее целое число меньше x. Так, «пол» 5.38 будет 5 и «пол» 3.14159 будет 3. Для положительных чисел x, «пол» x может быть найден простым отсечением десятичной части числа x.) Из трех деревьев на Рисунке 3, дерево (b) имеет наилучшее отношение высоты к числу узлов, так как высота дерева равна 3 и число узлов в дереве 8. Как мы уже обсуждали в первой части цикла статей, $\log_a b = y$ это другой вид записи $a^y = b$. $\log_2 8$, тогда, равно 3, потому что $2^3 = 8$. Дерево (a) имеет 10 узлов и высоту 4. $\log_2 10$ равно 3.3219 и пол этого числа равен 3. Поэтому 4 не является идеальной высотой. Заметьте, что если переорганизовать топологию дерева (a)—переместить самый нижний правый узел—мы сможем уменьшить высоту дерева на единицу, формируя тем самым оптимальное отношение высоты к числу узлов. И наконец, дерево (c) имеет наихудшее отношение высоты к числу узлов. Имея 5 узлов это дерево могло бы иметь оптимальную высоту 2, но из-за линейной топологии дерево имеет высоту 4.

Проблема, перед которой мы стоим, состоит в том, чтобы удостовериться, что топология результирующего BST представляет оптимальное отношение высоты к числу узлов. Так как топология BST основана на порядке в котором добавляются узлы вы можете интуитивно решить эту проблему удостоверяясь, что данные, которые добавляются в BST не добавляются в почти отсортированном порядке. Если вы не уверены в порядке данных, которые добавляются — как, например, данные основанные на вводе от пользователя, или данные считываются с сенсора — то в этом случае нельзя быть уверенным в том, что данные не поступают в отсортированном порядке. В этом случае решение состоит не в попытке предсказать порядок, в котором данные вставляются в дерево, а в том, чтобы обеспечить сбалансированность BST после каждой операции добавления данных. Структуры данных, которые спроектированы для поддержки сбалансированности, называются самобалансирующимися бинарными деревьями поиска (*self-balancing binary search trees*).

Сбалансированное дерево (*balanced tree*) это дерево, которое поддерживает некоторое предопределенное отношение между его высотой и шириной. Различные структуры данных определяют свое собственное отношение для сбалансированности, но все они близки к $\log_2 n$. В этом случае самобалансирующееся BST показывает асимптотическое время выполнения $\log_2 n$. Существует большое количество самобалансирующихся структур данных BST, такие как АВЛ деревья, красно-черные деревья, 2-3 деревья, 2-3-4 деревья, B-деревья, и прочих. В следующих двух разделах мы вкратце рассмотрим два самобалансирующихся дерева—АВЛ дерево и красно-черные деревья.

2.1 Изучение АВЛ (AVL) деревьев

В 1962 русские математики Г. М. Андельсон-Вельский и Е. М. Ландис изобрели первое самобалансирующееся BST, названное АВЛ (AVL) деревом. АВЛ (AVL) деревья должны поддерживать следующее свойство баланса — для каждого узла *n*, высота левого и правого поддеревьев *n* могут отличаться по крайней мере на 1. Высота левого и правого поддеревьев узла это высота, вычисленная с использованием техники, описанной в предыдущем разделе. Если узел имеет только одного потомка, то высота пустого поддерева определяется как -1.

Рисунок 4 концептуально показывает как АВЛ дерево должно поддерживать свойство отношения высот. Рисунок 5 предоставляет три примера BST. Числа в узлах этих деревьев представляют значения узлов; числа справа и слева от каждого узла представляют высоту левого и правого поддеревьев соответственно. На Рисунке 5, деревья (a) и (b) являются валидными АВЛ деревьями, а деревья (c) и (d) таковыми не являются, так как не все узлы придерживаются свойства сбалансированности АВЛ.

Comment [T2]: Помните, что АВЛ деревья являются бинарными деревьями поиска, поэтому в дополнение к поддержке свойства сбалансированности АВЛ дерево также должно поддерживать свойство бинарного дерева поиска.

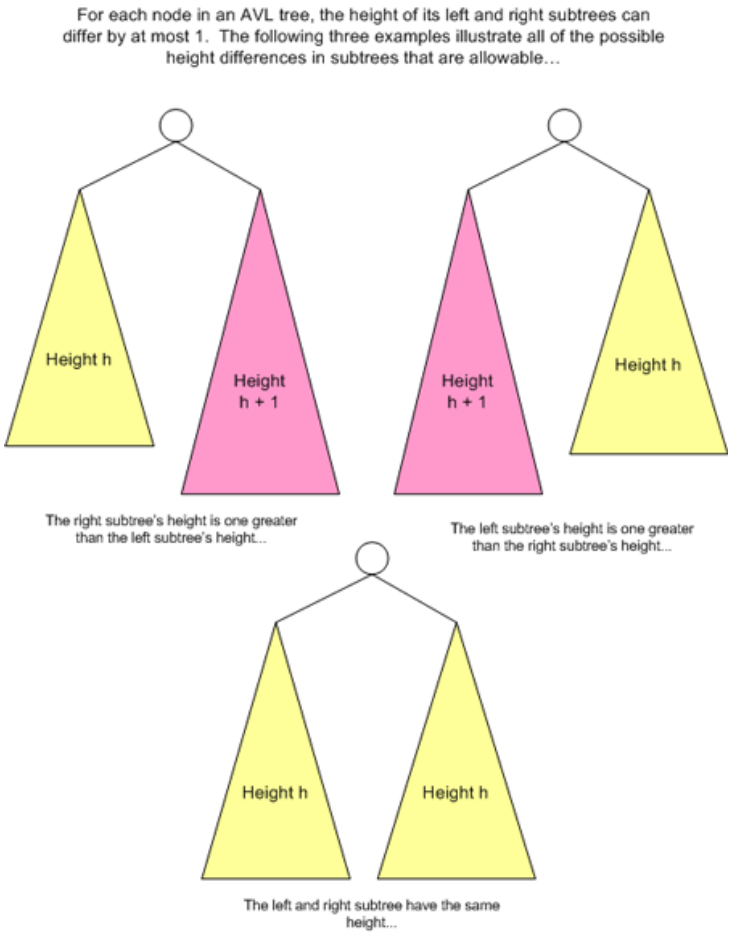

Рисунок 4. Высота левого и правого поддеревьев в АВЛ дереве не могут отличаться более чем на единицу.

Для каждого узла в АВЛ дереве высота его левого и правого поддеревьев могут отличаться не более чем на 1. Следующие три примера иллюстрируют все возможные различия в поддеревьях, которые допустимы...

Высота

Высота правого поддерева на единицу больше высоты левого поддерева...

Высота левого поддерева на единицу больше высоты правого поддерева...

Левое и правое поддеревья имеют одинаковую высоту...

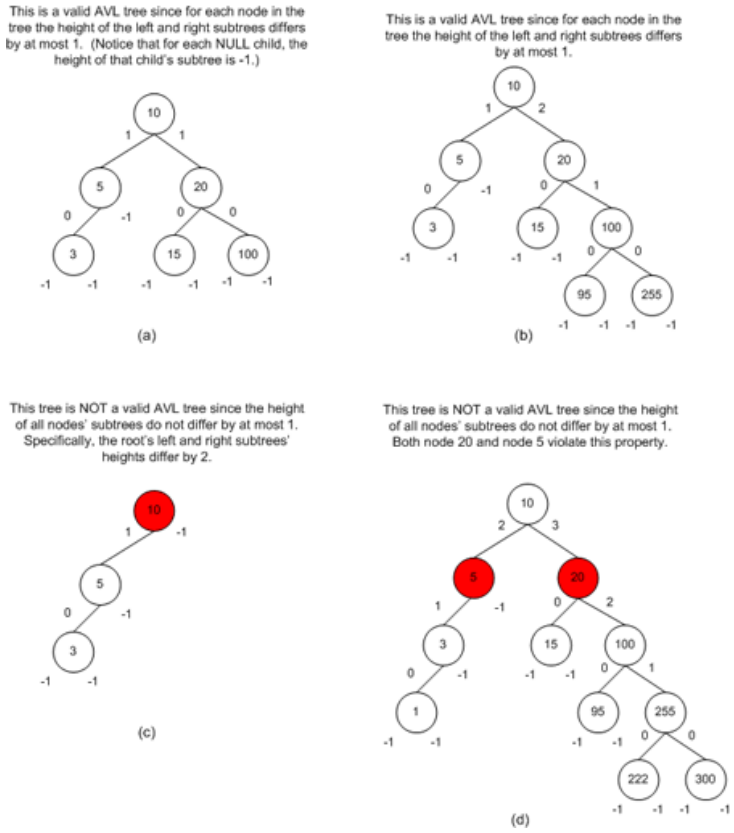

Рисунок 5. Пример деревьев, где (a) и (b) являются правильными АВЛ деревьями, а (c) и (d) нет.

(a) Это правильное АВЛ дерево так как для каждого узла дерева высота правого и левого поддеревьев отличаются не более чем на 1. (Отметьте, что для каждого NULL потомка высота этого поддерева равна -1.)

(b) Это правильное АВЛ дерево так как для каждого узла в дереве высота левого и правого поддеревьев отличаются не более чем на 1.

(c) Это дерево НЕ ЯВЛЯЕТСЯ правильным АВЛ деревом так как высота поддеревьев всех узлов отличаются более, чем на 1. В частности, правое и левое поддеревья корневого узла различаются по высоте на 2.

(d) Это дерево НЕ ЯВЛЯЕТСЯ правильным АВЛ деревом так как высота поддеревьев всех узлов отличаются более, чем на 1. Узлы 20 и 5 нарушают это свойство.

При создании структуры АВЛ дерева основная цель состоит в том, чтобы обеспечить сбалансированность дерева не зависимо от операций, выполняемых над деревом. То есть, при добавлении или удалении узлов жизненно важно, чтобы свойство сбалансированности сохранялось. АВЛ дерево обеспечивает сбалансированность посредством *вращений*. Вращение слегка изменяет топологию дерева таким образом, что восстанавливается свойство сбалансированности АВЛ и, что также важно, сохраняется свойство бинарного дерева поиска.

Добавление нового узла в АВЛ дерево представляет собой двухфазный процесс. Сперва узел вставляется в дерево с использованием того же алгоритма, что и добавление нового

узла в BST. То есть, новый узел добавляется как лист в подходящее место, чтобы сохранить свойство BST. После добавления нового узла может возникнуть ситуация, при которой этот новый узел нарушает свойство сбалансированности АВЛ в некотором узле на пути от корня к этому вновь добавленному узлу. Чтобы исправить любые нарушения вторая фаза включает обратный проход вверх, проверяя высоту левого и правого поддеревьев в каждом узле на протяжении обратного пути. Если высоты поддеревьев отличаются более чем на 1, то выполняется вращение, чтобы исправить аномалию.

Рисунок 6 иллюстрирует шаги для вращения узла 3. Отметьте, что после первой фазы операции вставки свойство АВЛ дерева было нарушено в узле 5, потому что высота левого поддерева узла 5 была на 2 больше, чем высота правого поддерева. Чтобы исправить это применяется вращение над узлом 3 – корнем левого поддерева узла 5. Это вращение исправляет разбалансированность и также восстанавливает свойство BST.

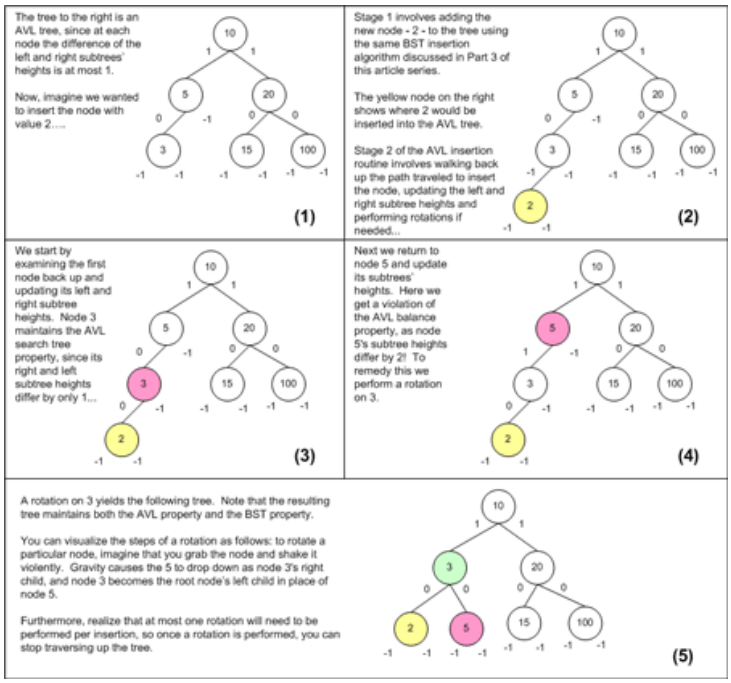

Рисунок 6. АВЛ деревья остаются сбалансированными посредством операций вращения

В дополнение к простому единственному вращению, показанному на Рисунке 6, возможен случай, когда требуется большее количество вращений. Исчерпывающее обсуждение множества вращений, потенциально необходимых для балансирования АВЛ дерева находится за объемом этой статьи. Что важно понять, так это то, что и вставки и удаления могут нарушить свойство сбалансированности, которому должно соответствовать АВЛ дерево. Чтобы исправить любые возмущения используются операции вращения.

Comment [T3]: Чтобы познакомиться с операциями вставки, удаления и вращениями над АВЛ деревьями, рекомендуем воспользоваться апплетом http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm. Этот апплет Java иллюстрирует как изменяется топология АВЛ дерева при добавлении и удалении узлов.

Гарантия того, что все высоты поддеревьев всех узлов отличаются не более чем на 1, дает гарантию того, что вставки, удаления и поиск в АВЛ деревьях всегда будет иметь ассимптотическое время выполнения $\log_2 n$, независимо от порядка в котором узлы добавляются в дерево.

2.2 Взгляд на красно-черные деревья

Структура данных красно-черного дерева была изобретена в 1972 Рудольфом Байером (Rudolf Bayer), профессором Мюнхенского Технического Университета. В дополнение к данным и правому и левому потомкам узлы красно-черного дерева содержат дополнительную информацию — цвет, который может быть одним из двух возможных, красным или черным. Красно-черные деревья также дополнены концепцией особого класса узлов, называемых NIL узлами. NIL узлы – это псевдо узлы, которые существуют как листья красно-черного дерева. То есть, все обычные узлы — это те, с которыми ассоциированы какие-либо данные — являются внутренними узлами. Вместо того, чтобы иметь NULL указатель для обычного узла без потомка, считается, что узел имеет вместо этой пустой ссылки NIL узел. Эта концепция может быть несколько сбивающей с толку. К счастью диаграмма на Рисунке 7 прояснит эту ситуацию.

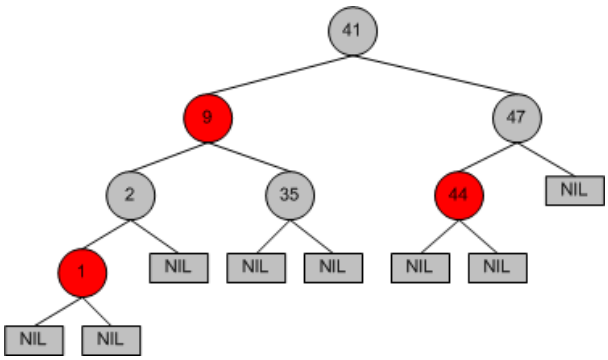

Рисунок 7. Красно-черные деревья добавляют концепцию NIL узла.

Красно-черные деревья это деревья, которые имеют следующие четыре свойства:

1. Каждый узел покрашен либо в красный, либо в черный цвет.
2. Каждый NIL узел покрашен в черный цвет.
3. Если узел покрашен в красный цвет, то оба его потомка покрашены в черный цвет.
4. Любой путь от узла к листу-потомку содержит одно и то же число черных узлов.

Первые три свойства являются достаточно самопоясняющими. Четвертое свойство, которое наиболее важное из четырех, просто утверждает, что начиная с любого узла в дереве, число черных узлов из этого узла к любому листу (NIL), должно быть одним и тем же. Возьмите в качестве примера корневой узел на Рисунке 7. Начиная от 41 и двигаясь к любому NIL узлу, вы встретите одно и то же число черных узлов—3. Например, рассматривая путь от 41 к самому левому узлу NIL, мы начинаем с 41 – черного узла. Затем мы переходим вниз к узлу 9, затем к узлу 2, который также черный, затем к узлу 1, и наконец к самому левому узлу NIL. В этом путешествии мы встретили три черных узла—41, 2, и коечный NIL узел. В действительности, если мы движемся от узла 41 к любому NIL узлу, мы всегда встретим точно три черных узла.

Так же как и АВЛ дерево, красно-черные деревья это другой вид самобалансирующегося бинарного дерева поиска. В то время как свойство сбалансированности АВЛ дерева было явно сформулировано как отношение между высотами правого и левого поддеревьев каждого узла, красно-черные деревья гарантируют их сбалансированность более заметным образом. Можно показать, что дерево, которое реализует четыре свойства

красно-черного дерева имеет высоту, которая всегда меньше, чем 2 * $\log_2(n+1)$, где n это общее число узлов в дереве. Поэтому красно-черные деревья гарантируют, что все операции могут быть выполнены за асимптотическое время of $\log_2 n$.

Так же как и в случае АВЛ деревьев, каждый раз когда добавляется или удаляется узел из красно-черного дерева, очень важно проверить, что свойства красно-черного дерева остались не нарушенными. В случае АВЛ деревьев свойство сбалансированности восстанавливалось с помощью вращений. В случае же красно-черных деревьев свойство сбалансированности восстанавливается посредством перекрашивания и вращений. Красно-черные деревья печально известны сложными правилами перекрашивания и вращений. Детальное обсуждение правил перекрашивания и вращений находится вне рамок этой статьи.

Чтобы посмотреть на перекраску и вращения красно черного дерева при добавлении и удалении узлов рекомендуем вам посмотреть апплет красно-черного дерева, который может быть найден по адресу http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm.

3 Краткий учебник по связанным спискам

Еще одна структура данных, которую нам нужно обсудить называется связанный список (*linked list*). Так как структура данных списка пропусков, которую мы рассмотрим далее, это видоизменение связанного списка в структуру данных с временем выполнения самобалансирующегося бинарного дерева, то очень важно сперва обсудить связанные списки перед погружением в специфику списков пропуска.

Вспомните, что в бинарном дереве каждый узел хранит некоторые данные и имеет ссылки на правого и левого потомков. Связанный список можно рассматривать как унарное дерево. То есть, каждый элемент связанного списка хранит некоторые связанные с ним данные и имеет единственную ссылку на своего соседа. Как это проиллюстрировано на Рисунке 8, каждый элемент в связанном списке формирует связь в цепочке. Каждая связь привязана к соседнему узлу справа.

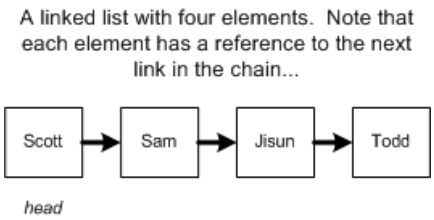

Рисунок 8. Связанный список из четырех элементов

Связанный список с четырьмя элементами. Отметьте, что каждый элемент имеет ссылку на следующее звено в цепи ...

Голова

Когда мы создавали структуру данных бинарного дерева в Части III, то эта структура нуждалась только в одной ссылке к корню дерева. Сам корень имел ссылки к своим потомкам, а эти потомки содержали ссылки к своим потомкам и так далее. Аналогично, в случае структуры данных связанного списка, при реализации структуры нам необходима

ссылка только на голову списка, поскольку каждый элемент в списке хранит ссылку на следующий элемент списка.

Связанные списки имеют линейное время выполнения для поиска, так же как и массивы. То есть, чтобы найти элемент Sam в связанном списке на Рисунке 8, мы должны начать с головы списка и проверить каждый элемент один за другим. Здесь нет никаких возможностей сокращения поиска, как это было в бинарных деревьях или хэш-таблицах. Аналогично, удаление из связанного списка также потребует линейного времени, потому что сначала нужно выполнить поиск данного элемента в списке. Как только элемент найден, удаление его из связанного списка потребует переназначения ссылок соседних элементов. Рисунок 9 иллюстрирует назначение указателей, которое должно произойти при удалении элемента из связанного списка.

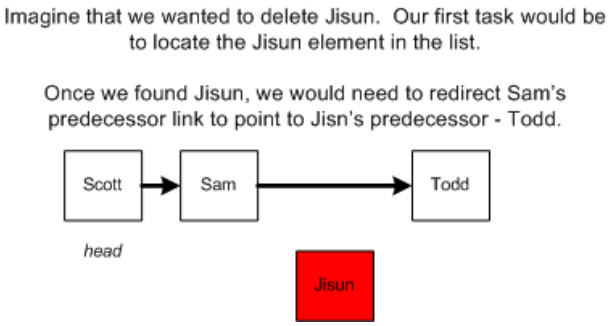

Рисунок 9. Удаление элемента из связанного списка

Представьте, что мы хотим удалить Jisun. Наша задача будет состоять в том, чтобы найти элемент Jisun в списке.

Когда мы нашли элемент Jisun, то нам потребуется перенаправить ссылку элемента Sam на непосредственно следующий за Jisun элемент -- Todd.

Голова

Асимптотическое время, требуемое для вставки нового элемента в связанный список зависит от того является ли список отсортированным или нет. Если не требуется, чтобы элементы списка были отсортированы, то операция вставки может быть выполнена за константное время потому что мы можем добавить элемент в начало списка. Это требует создания нового элемента, установки ссылки соседа на текущую голову связанного списка и, наконец, переназначение указателя на голову связанного списка на новый вставленный элемент.

Если же элементы связанного списка должны быть организованы в отсортированном порядке, то первым шагом при добавлении нового элемента будет поиск места, куда этот элемент необходимо вставить. Это может быть осуществлено полным перебором от начала списка до элемента, где должен быть вставлен новый элемент. Пусть *e* будет элементом стоящим непосредственно перед местом, где будет добавлен новый элемент. Для вставки нового элемента ссылка от элемента *e* должна теперь указывать на вновь вставленный элемент, а ссылка от нового элемента должна указывать на старого соседа элемента *e*. Рисунок 10 иллюстрирует эту концепцию графически.

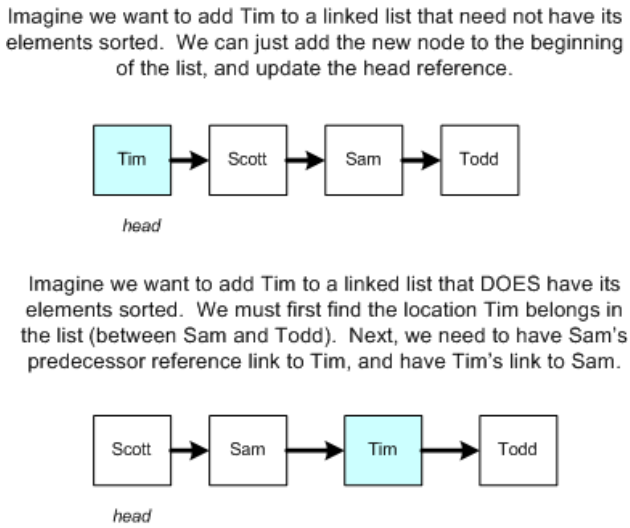

Рисунок 10. Вставка элемента в отсортированный связанный список

Представьте, что мы хотим добавить элемент Tim в связанный список, которые не требует, чтобы элементы располагались в отсортированном порядке. Мы просто можем добавить новый узел в начало списка и обновить ссылку головы списка.

Голова

Представьте, что мы хотим добавить элемент Tim в связанный список и что ЭТОТ ЭЛЕМНЕТ ДОЛЖЕН РАСПОЛАГАТЬСЯ в отсортированном порядке. Сперва мы должны найти место, где элемент Tim должен находиться (между Sam и Todd). Далее нам необходимо, чтобы Sam ссылался на новый элемент Tim, а Tim ссылался на старого соседа Sam'а Todd.

Голова

Заметьте, что связанные списки не обеспечивают прямого доступа к элементам, так как это делают массивы. То есть, если вы хотите получить доступ к *i*-му элементу списка, то вы должны начать с головы списка и пройти по нему до *i* связи. Однако, в случае массива вы можете непосредственно обратиться к *i*-му элементу. Зная все это, наряду с фактом того, что связанные списки не предоставляют лучшего времени поиска, чем массивы, вы можете задаться вопросом, почему вообще кого-нибудь может интересовать использование связанного списка.

Основное преимущество связанных списков состоит в том, что добавление и удаление элементов не требует сложного и длительного процесса перераспределения памяти. Вспомните, что массив имеет фиксированный размер и поэтому если в массив нужно добавить больше элементов, чем его емкость, то это потребует перераспределения памяти под массив. Более того, связанные списки идеально подходят для интерактивного добавления элементов в отсортированном порядке. В случае массива, вставка элемента в середину массива требует перемещения всех оставшихся элементов для создания места для вставляемого элемента. Короче говоря, массив обычно является подходящим выбором если вы представляете верхнюю границу количества данных, которые нужно хранить. Если вы не имеете возможной оценки того, сколько элементов нужно хранить, то связанный список будет более предпочтительным выбором.

И в заключение, связанные списки гораздо более просты в реализации. Основная проблема с которой придется столкнуться состоит в «прошивке» или «пере-прошивке» соседних ссылок при добавлении или удалении элементов, но сложность добавления или удаления элемента из связанного списка теряет актуальность в сравнении со сложностью балансирования АВЛ или красно-черных деревьев.

В .NET Framework версии 2.0, класс связанных списков был добавлен к Base Class Library—`System.Collections.Generic.LinkedList`. Этот класс реализует дважды связанный список (*doubly-linked list*), который представляет собой список у которого узлы имеют ссылки как на следующего соседа, так и на предыдущего соседа. `LinkedList` состоит из произвольного числа экземпляров `LinkedListNode`, которые, в дополнение к ссылкам на предыдущего и следующего соседей, хранят также величину `Value` тип которой может быть задан с использованием параметризованных классов (Generics).

4 Список пропусков (Skip Lists): связанный список со свойствами аналогичными самобалансирующемуся BST

В 1989 году Вильям Пуф (William Pugh), профессор из Университета Мэриленда (University of Maryland), рассматривал связанные списки с точки зрения их времени выполнения. Отсортированный связанный список требует линейного времени для поиска так как потенциально необходимо проверить каждый элемент списка, один за другим. Пуф подумал, что если бы половина элементов в отсортированном списке имела две ссылки на соседние элементы—одна, указывающая на непосредственного соседа, и вторая, указывающая на соседа на два элемента вперед—в то время, как оставшаяся половина имела бы только одну ссылку, то поиск в отсортированном списке можно было бы выполнить за половину времени. Рисунок 11 иллюстрирует двух-ссылочный отсортированный связанный список.

Рисунок 11. Список пропуска (skip list)

Способ, которым такой связанный список позволяет сократить время поиска, состоит в том, что, во-первых, элементы списка отсортированы, и, во-вторых, элементы списка имеют разную высоту. Чтобы найти, скажем Dave, мы начинаем с головы списка, который является пустым элементом, чья высота такая же как и максимальная высота элемента в списке. Голова списка не содержит никаких данных, она просто служит местом для начала поиска.

Мы начинаем с самой высокой связи потому что это позволит нам пропускать более низкие элементы. Мы начинаем со следования самой верхней связи головного элемента к элементу Bob. В этом месте мы можем спросить себя следует ли Bob до или после Dave? Если он следует перед Dave, то мы тогда знаем, что Dave, если он существует в списке, должен существовать где-либо справа от элемента Bob. В этом случае, Dave следует после элемента Bob в алфавитном порядке, поэтому мы можем повторить наш поиск снова с элемента Bob. Отметьте, что перемещаясь к элементу Bob мы пропускаем элемент Alice. У элемента Bob мы повторяем поиск на том же самом уровне. Следуя самому верхнему указателю мы достигаем элемента Dave, пропуская элемент Cal. Так как мы нашли то, что искали, мы можем остановить поиск.

Теперь давайте представим, что мы хотим найти имя Cal. Мы снова начинаем с головы списка и затем перемещаемся к элементу Bob. У элемента Bob мы продолжим движение и переместимся к элементу Dave. Так как Dave следует *за* элементом Cal, то мы знаем, что Cal должен существовать где-то между элементами Bob и Dave. Поэтому мы переходим к следующей более низкой по высоте ссылке и продолжаем наше сравнение.

Эффективность такого связанного списка может возрасти еще больше, так как мы сможем проходить поверх двух элементов за один раз вместо одного. Это приведет к времени выполнения порядка $n/2$, которое, хотя и лучше, чем для случая отсортированного связанного списка, однако все еще ассимптотически являющееся линейным временем выполнения. Понимая это Пуф пожелал узнать, что произойдет если вместо ограничения высоты элементов до величины 2, позволить ей расти до величины $\log_2 n$ для случая n элементов. То есть, если в связанном списке 8 элементов, то высота элементов списка может доходить до 3; если в списке 16 элементов, то высота элементов может достигать 4. Как показано на Рисунке 12, разумно выбирая высоту каждого элемента время поиска может быть снижено до $\log_2 n$.

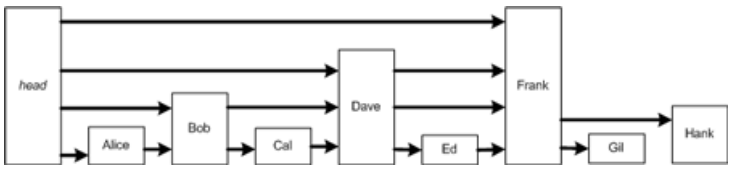

Рисунок 12. Увеличивая высоту каждого узла в списке пропусков можно достичь лучшей производительности операции поиска.

Отметьте, что узлы на Рисунке 12 каждый 2^i–й узел имеет ссылки на 2^i–й элемент вперед. То есть, элемент 2^0, Alice, имеет ссылки на 2^0 элементов вперед—Bob. Элемент 2^1 , Bob, имеет ссылку на узел 2^1 элементов вперед—Dave. Dave, элемент 2^2, имеет ссылку 2^2 элементов вперед—Frank.

Недостаток подхода проиллюстрированного на Рисунке 12 состоит в том, что добавление новых элементов или удаление существующих может привести к нарушению этой точной структуры. То есть, если удаляется элемент Dave, то элемент Ed становится элементом 2^2, а Gil элементом 2^3, и так далее. Это означает, что *все* элементы справа от удаляемого элемента требуют уточнения их высот и ссылок. Та же самая проблема возникает и в случае вставки. Это перераспределение высот и ссылок не только усложнит код реализации этой структуры данных, но также снизит время выполнения операций вставки и удаления до линейного.

Пуф также отметил, что эта структура создала 50% элементов высотой 1, 25% высотой 2, 12.5% высотой 3, и так далее. То есть, $1/2i$ процентов элементов имеют высоту i. Вместо того, чтобы пытаться обеспечить правильную высоту для каждого элемента в отношении его порядкового номера в списке, Пуф решил выбирать высоту случайным образом используя идеальное распределение—50% высотой 1, 25% высотой 2, и так далее. Что обнаружил Пуф так это то, что такой связанный список, построенный случайным образом, было не так легко реализовать в коде, но также и то, что он показал асимптотическое время выполнения $\log_2 n$ для операций вставки, удаления и поиска. Пуф назвал его изобретение *списками пропусков* так как проход по списку осуществляется с пропуском элементов с меньшей высотой.

В оставшихся разделах мы исследуем операции вставки, удаления и поиска для списка пропусков и реализуем его в виде класса на языке C#. И закончим мы обзором производительности списков пропуска и обсуждением различий между списками пропусков и самобалансирующихся BST.

4.1 Создание классов SkipListNode и SkipListNodeList

Список пропусков, так же как и бинарное дерево, состоит из набора элементов. Каждый элемент в списке пропусков содержит некоторые данные, связанные с ним, высоту и набор ссылок на элементы. Например, на Рисунке 12 элемент Bob содержит данные «Bob», высоту 2, и две ссылки на другие элементы: одну на Dave и одну на Cal. Перед созданием класса списков пропуска нам сперва необходимо создать класс, представляющий элемент в списке пропусков. Я создал класс, который расширяет функциональность базового класса `Node`, который мы изучили в Части III нашей серии статей, и назовем этот расширенный класс `SkipListNode`. Код класса `SkipListNode` приведен далее. Отметьте, что класс `SkipListNode` реализован с использованием параметризованных типов (Generics), позволяя тем самым разработчику создавать списки пропуска для хранения данных типа, определяемого во время разработки.

```
public class SkipListNode<T> : Node<T>
{
    private SkipListNode() {}   // нет конструктора по умолчанию
    public SkipListNode(int height)
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public SkipListNode(T value, int height) : base(value)
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public int Height
    {
        get { return base.Neighbors.Count; }
    }

    public SkipListNode<T> this[int index]
    {
        get { return (SkipListNode<T>) base.Neighbors[index]; }
        set { base.Neighbors[index] = value; }
    }
}
```

Класс `SkipListNode` использует класс `SkipListNodeList` для хранения набора ссылок на экземпляры `SkipListNode`; эти ссылки на экземпляры `SkipListNode` являются соседями `SkipListNode`. (Вы можете видеть, что свойству `Neighbors` в конструкторе базового класса присваивается экземпляр `SkipListNodeList<T>` заданной высоты.) Класс `SkipListNodeList` расширяет базовый класс NodeList добавлением двух методов— `IncrementHeight()` и `DecrementHeight()`. Как мы увидим в разделе "Добавление в список пропусков", высота SkipList может нуждаться в увеличении или уменьшении в зависимости от высоты добавляемого или удаляемого элемента.

```
public class SkipListNodeList<T> : NodeList<T>
{
    public SkipListNodeList(int height) : base(height) { }
```

```
    internal void IncrementHeight()
    {
        // добавить пустой элемент
        base.Items.Add(default(Node<T>));
    }

    internal void DecrementHeight()
    {
        // удалить последний элемент
        base.Items.RemoveAt(base.Items.Count - 1);
    }
}
```

Конструктор `SkipListNodeList` имеет входной параметр высоты, который определяет число ссылок на соседние элементы, которые нужны узлу. Он распределяет заданное число элементов в свойстве `Neighbors` базового класса вызывая конструктор базового класса, и передавая ему высоту. Методы `IncrementHeight()` и `DecrementHeight()` добавляют новый узел к набору и удаляют самый верхний узел, соответственно. Вскоре мы увидим для чего нужны эти два вспомогательных метода.

Имея классы `SkipListNode` и `SkipListNodeList` мы готовы перейти к созданию класса `SkipList`. Класс `SkipList` как мы увидим, содержит одну ссылку на головной элемент. Он также предоставляет методы для поиска в списке, проходу по элементам списка, добавления элементов в список и удаления элементов из списка.

Comment [T4]: Графическое представление списка пропусков в действии можно найти в виде апплета по адресу http://iamwww.unibe.ch/~wenger/DA/SkipList/. Вы можете добавлять или удалять элементы из списка пропусков наглядно видя как изменяется структура и высота списка пропусков после каждой операции.

4.2 Создание класса SkipList

Класс **SkipList** предоставляет абстракцию списков пропуска. Он содержит такие методы общего доступа:

- **Add(value):** добавляет новый элемент в список пропусков.
- **Remove(value):** удаляет существующий элемент из списка пропусков.
- **Contains(value):** возвращает ИСТИНУ, если элемент существует в списке пропусков, и ЛОЖЬ в противном случае.

А также такие свойства общего доступа как:

- **Height:** высота самого высокого элемента в списке.
- **Count:** общее число элементов в списке пропусков.

Скелет структуры класса показан ниже. На протяжении следующих разделов мы изучим операции над списком пропусков и создадим код для этих методов.

```
public class SkipList<T> : IEnumerable<T>, ICollection<T>
{
   SkipListNode<T> _head;
   int _count;
   Random _rndNum;
   private IComparer<T> comparer = Comparer<T>.Default;

   protected readonly double _prob = 0.5;

   public int Height
   {
      get { return _head.Height; }
   }
```

```
   public int Count
   {
      get { return _count; }
   }

   public SkipList() : this(-1, null) {}
   public SkipList(int randomSeed) : this(randomSeed, null) {}
   public SkipList(IComparer<T> comparer) : this(-1, comparer) {}
   public SkipList(int randomSeed, IComparer<T> comparer)
   {
      _head = new SkipListNode<T>(1);
      _count = 0;
      if (randomSeed < 0)
         _rndNum = new Random();
      else
         _rndNum = new Random(randomSeed);

      if (comparer != null) this.comparer = comparer;
   }

   protected virtual int ChooseRandomHeight(int maxLevel)
   {
      ...
   }

   public bool Contains(T value)
   {
      ...
   }

   public void Add(T value)
   {
      ...
   }

   public bool Remove(T value)
   {
      ...
   }
}
```

Мы вскоре напишем код для методов класса, а сейчас мы уделим пристальное внимание закрытым (private) членам класса, открытым (public) свойствам и конструкторам. Класс содержит три закрытых (private) члена класса:

- `_head`, который является головным элементом списка. Вспомните, что список пропусков имеет пустой головной элемент (вернитесь к Рисункам 11 и 12, чтобы увидеть графическое представление головного элемента).
- `_count`, целочисленная величина отслеживающая количество элементов присутствующих в списке пропусков.
- `_rndNum`, экземпляр класса `Random`. Так как нам необходимо случайным образом формировать высоту элемента, добавляемого в список, то мы будем использовать этот экземпляр класса `Random` для генерирования случайных чисел.

Класс **SkipList** имеет два открытых (public) свойства доступных только для чтения `Height` и `Count`. `Height` возвращает высоту самого высокого элемента списка пропусков. Так как высота головного элемента всегда равна высоте самого высокого элемента, то мы просто возвращаем значение свойства `Height` головного элемента. Свойство `Count` просто

возвращает текущее значение закрытой (private) переменной класса `count`. (`count`, как мы увидим, наращивается в методе `Add()` и уменьшается в методе `Remove()`.)

Отметьте, что существует четыре формы конструктора `SkipList`, которые предоставляют все перестановки для задания начального значения генератора случайных чисел и функции сравнения. Конструктор, вызываемый по умолчанию создает список пропусков с использованием принимаемого по умолчанию метода для сравнения элементов типа `T`, и позволяя экземпляру класса `Random` самому выбирать начальное значение для генератора случайных чисел. Не зависимо от того какой конструктор используется создается головной элемент списка пропусков `SkipListNode<T>` высотой 1 и значение `count` устанавливается равным 0.

Comment [T5]: Компьютерный генератор случайных чисел, такой как класс `Random` из .NET Framework, называется *псевдо-случайным генератором чисел* потому что он в действительности не генерирует случайных чисел, а вместо этого использует функцию для вычисления следующего случайного числа. Функция генерации случайного числа работает начиная с некоторого значения, называемого *зерном (seed)*. Основываясь на этом зерне – начальном значении – вычисляется последовательность случайных чисел. Небольшие изменения зерна ведут к кажущимся случайным изменениям в последовательности чисел.

Если вы используете вызываемый по умолчанию конструктор класса `Random`, то используется значение системных часов для генерации зерна. Вы можете опционально задать специфическое значение зерна. Преимущество в задании зерна состоит в том, что если вы используете одно и то же зерно, то вы получите одну и ту же последовательность случайных чисел. Имея возможность получить один и тот же результат важно при проверке правильности и эффективности работы таких алгоритмов, как алгоритмы работы со списками пропусков.

4.2.1 Поиск в списке пропусков

Алгоритм для поиска определенного значения в списке пропусков достаточно прямой. Неформально процесс поиска может быть описан следующим образом: мы начинаем с самой верхней ссылки головного элемента. Сперва мы выполняем проверку является ли значение *e* меньше чем, больше чем или равное значению, которое мы ищем. Если оно равно искомому значению, то мы нашли то, что искали. Если оно больше искомого значения, то если это значение существует в списке, оно должно находиться слева от *e*, означая тем самым, что его высота меньше высоты *e*. Поэтому мы перемещаемся вниз ко второму уровню ссылок и повторяем процесс.

Если, с другой стороны, величина *e* меньше, чем искомая величина, то значение, если оно существует в списке, должно быть с правой стороны от *e*. Поэтому, мы повторяем эти шаги для самой верхней ссылки *e*. Этот процесс продолжается до тех пор пока мы либо найдем значение, которое мы ищем, либо переберем все «уровни» и не найдем требуемое значение.

Более формально алгоритм может быть выражен следующим псевдокодом:

```
SkipListNode current = head
for i = skipList.Height downto 1
  while current[i].Value < valueSearchingFor
    current = current[i]  // перейти к следующему узлу

if current[i].Value == valueSearchingFor then
   return true
else
return false
```

Потратим немного времени, чтобы пройтись алгоритмом по списку пропусков показанному на Рисунке 13. Красные стрелки показывают путь проверок при поиске в списке пропусков. Список пропусков (a) показывает результаты при поиске значения Ed; список пропусков (b) показывает результаты при поиске значения Cal; список пропусков (c) показывает результаты при поиске значения Gus, которое не существует в списке

пропусков. Отметьте, что на протяжение всего алгоритма мы производим движение в направлении вправо и вниз. Алгоритм никогда не перемещается к узлу, находящемуся слева от текущего узла и никогда не переходит к ссылке на более высоком уровне.

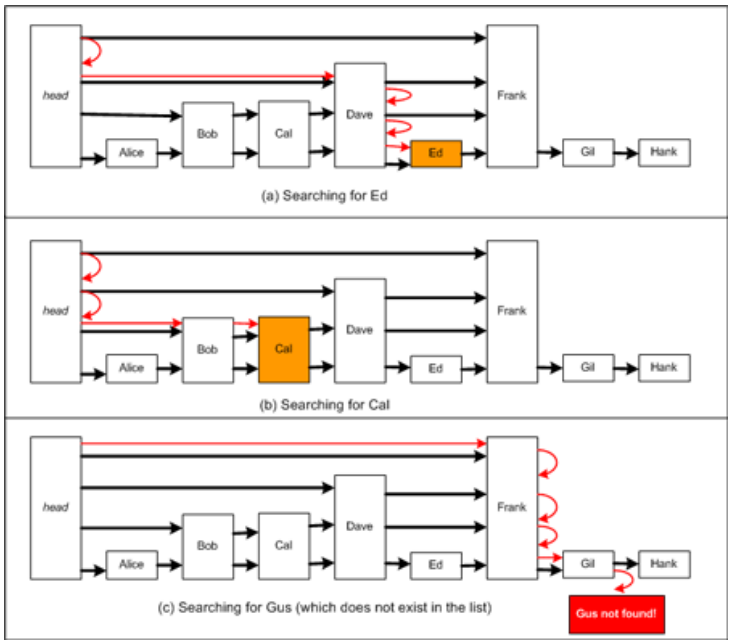

Рисунок 13. поиск в списке пропусков.

(a) Поиск элемента Ed

(b) Поиск элемента Cal

(c) Поиск элемента Gus (который не существует в списке)

Элемент Gus не найден

Код метода `Contains(value)` достаточно прост и состоит только из двух циклов `while` и `for`. Цикл `for` проходит сверху вниз по слоям ссылок; а цикл `while` проходит по элементам списка пропусков.

```
public bool Contains(T value)
{
   SkipListNode<T> current = _head;

for (int i = _head.Height - 1; i >= 0; i--)
   {
      while (current[i] != null)
      {
         int results = comparer.Compare(current[i].Value, value);
         if (results == 0)
            return true;   // мы нашли элемент
         else if (results < 0)
            current = current[i];   // элемент справа, перейти на уровень
ниже
         else // results > 0
```

```
            break;  // выйти из цикла while, так как элемент находится справа
от этого узла, на (или ниже чем) текущем уровне
        }
    }

    // если мы дошли до этого места, то мы произвели поиск до конца списка и
не нашли требуемый элемент
    return false;
}
```

4.2.2 Вставка в список пропусков

Вставка нового элемента в список пропусков сродни добавлению нового элемента в отсортированный связанный список, и состоит из двух шагов. Первый, мы должны найти место, где должен быть расположен новый элемент в списке пропусков. Это место находится с использованием алгоритма поиска для того, чтобы найти место, которое следует непосредственно перед той позицией, куда будет вставлен новый элемент. Второй, мы должны «вшить» новый элемент в список обновляя необходимые ссылки.

Так как элементы списка пропусков могут иметь много уровней и поэтому много ссылок, то «вшивание» нового элемента в список пропусков не является такой простой задачей как вставка нового элемента в простой связанный список. Рисунок 14 показывает диаграмму списка пропусков и процесс «вшивания», который необходимо проделать для добавления элемента Gus. В этом примере представим, что случайно определенная высота элемента Gus равна 3. Чтобы успешно «вшить» элемент Gus, нам необходимо обновить ссылки уровней 3 и 2 элемента Frank, а также ссылку уровня 1 элемента Gil. Ссылка уровня 1 элемента Gus будет указывать на Hank. Если бы справа от элемента Hank были другие узлы, то ссылка уровня 2 элемента Gus указывала бы на первый элемент справа от элемента Hank с высотой 2 или более, в то время как ссылка уровня 3 элемента Gus будет указывать на первый элемент справа от Hank высотой 3 и более.

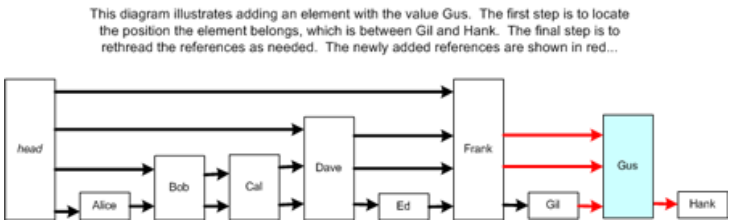

Рисунок 14. Вставка элемента в список пропусков

Диаграмма иллюстрирует добавление элемента со значением Gus. Первый шаг состоит в нахождении позиции, куда должен быть добавлен элемент, и которая находится между Gil и Hank. Последний шаг состоит в «перепрошивке» необходимых ссылок. Вновь добавленный элемент показан красным цветом ...

В случае, чтобы удачно перепрошить список пропусков после вставки нового элемента нам необходимо отслеживать последний элемент каждой высоты. На Рисунке 14, Frank был последним элементом который использовался для ссылок уровней 4, 3 и 2, в то время как Gil был последним элементом для ссылок уровня 1. В алгоритме вставки приведенном ниже эти записи о последнем элементе для каждого уровня поддерживается с помощью массива `updates`. Этот массив, который заполняется во время выполнения операции поиска места для нового элемента, создается в методе `BuildUpdateTable()` (также приведенном ниже).

```
public void Add(T value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0];

   // see if a duplicate is being inserted
   if (current[0] != null && current[0].Value.CompareTo(value) == 0)
      // cannot enter a duplicate, handle this case by either just returning
or by throwing an exception
      return;

   // create a new node
   SkipListNode<T> n = new SkipListNode<T>(value,
ChooseRandomHeight(head.Height + 1));
   _count++;   // increment the count of elements in the skip list

   // if the node's level is greater than the head's level, increase the
head's level
   if (n.Height > _head.Height)
   {
      _head.IncrementHeight();
      _head[_head.Height - 1] = n;
   }

   // splice the new node into the list
   for (int i = 0; i < n.Height; i++)
   {
      if (i < updates.Length)
      {
         n[i] = updates[i][i];
         updates[i][i] = n;
      }
   }
}

protected SkipListNode<T>[] BuildUpdateTable(T value)
{
    SkipListNode<T>[] updates = new SkipListNode<T>[_head.Height];
    SkipListNode<T> current = _head;

    // determine the nodes that need to be updated at each level
    for (int i = _head.Height - 1; i >= 0; i--)
    {
        while (current[i] != null && comparer.Compare(current[i].Value,
value) < 0)
            current = current[i];

        updates[i] = current;
    }

    return updates;
}
```

В методах `Add(value)` и `BuildUpdateTable()` есть пару ключевых мест на которые требуется обратить пристальное внимание. Первое, в методе `BuildUpdateTable()`, проверьте цикл `for`. В этом цикле полностью заполняется массив `updates`. Экземпляр `SkipListNode` непосредственно предшествующий месту, где будет вставлен новый узел, представлен в `updates[0]`.

После `BuildUpdateTable()`, выполняется проверка для удостоверения того, что добавляемые данные не дублируются. Я предпочитаю реализовать список пропусков

таким образом, чтобы дубликаты данных не повторялись; однако, списки пропусков могут с успехом работать и с дублирующимися данными. Если вы желаете разрешить появление дубликатов просто уберите эту проверку.

Далее создается новый экземпляр класса `SkipListNode`. Он представляет элемент, который будет добавлен в список. Отметьте, что высота вновь созданного экземпляра класса `SkipListNode` определяется в результате вызова метода `ChooseRandomHeight()`, передавая ему текущую высоту списка пропусков плюс единица. Мы изучим этот метод несколько ниже. Другое действие на которое нужно обратить внимание это то, что после добавления `SkipListNode`, выполняется проверка, чтобы увидеть не является ли высота нового `SkipListNode` больше, чем высота заглавного элемента списка пропусков. Если это так, то необходимо увеличить высоту заглавного элемента, так как высота заглавного элемента должна быть такой же как и высотат самого высокого элемента в списке пропусков.

4.2.3 Случайное определение высоты вновь вставляемого экземпляра SkipListNode

При добавлении нового элемента в список пропусков нам необходимо случайным образом выбрать высоту для этого нового экземпляра класса `SkipListNode`. Вспомните с наших предыдущих рассуждений списков пропуска, что когда Пуф впервые рассматривал многоуровневый связанный список элементов, он представлял себе связанный список в котором каждый 2^i-й элемент имеет ссылку на элемент, который отстоит на 2^i элементов вперед. В таком списке в точности 50% будут иметь высоту 1, 25% высоту 2, и так далее.

Метод `ChooseRandomHeight()` использует простую технику для вычисления высот таким образом, чтобы их распределение соответствовало начальному представлению Пуфа. Это распределение можно получить бросая монетку и устанавливая высоту на единицу больше, количества «орлов» выпавших подряд. То есть, если после первого броска у вас выпала «решка», то высота нового элемента будет равна единице. Если у вас выпадет «орел», а затем «решка», то высота будет 2. Два «орла» за которыми следует «решка» задает высоту три и так далее. Так как вероятность того, что у вас выпадет «решка» равна 50%, «орел» и затем «решка» 25%, два «орла» и затем «решка» 12.5%, и так далее, то это распределение работает так же как и необходимое нам распределение.

Код для вычисления случайных высот приведен ниже:

```
const double _prob = 0.5;
protected virtual int ChooseRandomHeight()
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob)
      level++;

   return level;
}
```

Единственное беспокойство касательно метода, приведенного выше, состоит в том, что возвращаемое значение может быть невероятно большим. То есть представьте, что мы имеем список пропусков, состоящий из, скажем, двух элементов, каждый из которых имеет высоту 1. Когда мы добавляем третий элемент, то мы случайным образом выбираем высоту 10. Это не очень желательное событие, поскольку существует приблизительно 0.1% шанса выбрать такую высоту, но это все-таки может случиться. Другая сторона этого теперь в том, что наш список пропусков имеет элемент с высотой 10, означая тем самым,

что в нашем списке пропусков будет достаточное количество излишних уровней. Проще говоря, ссылки на уровнях от второго до десятого использоваться не будут.

Пуф предложил пару решений этой проблемы. Первая, просто игнорировать ее. Наличие лишних уровней не требует никакого изменения кода структуры данных и никак не влияет на асимптотическое время выполнения. Другое решение, предложенное Пуфом, требует использования так называемых «фиксированных игральных костей» ("fixed dice") при выборе случайного уровня, и является именно тем подходом, который я использую в реализации класса `SkipList`. Используя «фиксированные игральные кости» вы ограничиваете высоту нового элемента до высоты самого высокого элемента в списке плюс единица. Реализация метода `ChooseRandomHeight()` показана ниже, и она использует подход «фиксированных игральных костей».

```
protected virtual int ChooseRandomHeight(int maxLevel)
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob && level < maxLevel)
      level++;

   return level;
}
```

Так как головной элемент должен быть такой же высоты как и самый высокий элемент в списке пропусков, то в методе `Add(`*`value`*`)`, если высота вновь добавленного `SkipListNode` больше, чем высота головного элемента, то вызывается метод `IncrementHeight()`:

```
/* - snippet from the Add() method... */
if (n.Height > _head.Height)
{
   _head.IncrementHeight();
   _head[_head.Height - 1] = n;
}
/*********************************/
```

Метод `IncrementHeight()` просто добавляет новый экземпляр `SkipListNode` к головному элементу. Так как высота головного элемента должна быть равна высоте самого высокого элемента в списке пропусков, то если мы добавляем элемент, чья высота больше, чем высота головного элемента, то мы должны увеличить высоту головного элемента, именно то, что и выполняет код в предложении `if`.

4.2.4 Удаление элемента из списка пропусков

Так же как и добавление элемента в список пропусков, удаление элемента представляет собой процесс из двух шагов. Первый, необходимо найти удаляемый элемент. После этого элемент должен быть вырезан из списка и затем все ссылки «перепрошиты». Рисунок 15 показывает «перепрошивку», которая должна произойти, если элемент Dave удаляется из списка пропусков.

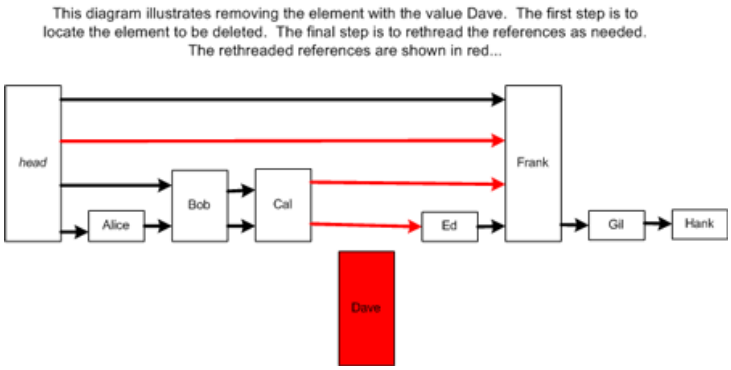

Рисунок 15. Удаление элемента из списка пропусков

Эта диаграмма иллюстрирует удаление элемента со значением Dave. Первый шаг состоит в нахождении элемента, который подлежит удалению. Заключительный шаг производит «перепрошивку» требуемых связей. «Перепрошитые» ссылки показаны красным цветом ...

Также как и в методе `Add(value)`, метод `Remove(value)` поддерживает массив `updates`, который отслеживает элементы на каждом уровне, которые следуют непосредственно перед удаляемым элементом (так же как и в `Add()` этот массив `updates` заполняется с помощью метода `BuildUpdateTable()`.). Как только этот массив заполнен он анализируется снизу вверх и все элементы в массиве «перепрошиваются» таким образом, чтобы указывать на ссылки удаляемого элемента на соответствующих уровнях. Код метода `Remove(value)` приведен ниже.

```
public virtual void Remove(IComparable value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0][0];

   if (current != null && comparer.Compare(current.Value, value) == 0)
   {
      _count--;

      // Мы нашли данные для удаления
      for (int i = 0; i < _head.Height; i++)
      {
         if (updates[i][i] != current)
            break;
         else
            updates[i][i] = current[i];
      }

      // наконец, посмотрим нужно ли нам усечь высоту списка
      if (_head[_head.Height - 1] == null)
         _head.DecrementHeight();

      return true;   }
   else
      // данные для удаления найдены не были - вернтуь ЛОЖЬ
            return false;
}
```

Метод `Remove(value)` начинается также как и метод `Add(value)`, а именно заполнением массива `updates` с помощью метода `BuildUpdateTable()`. Имея заполненный массив

`updates` мы далее должны удостовериться, что найденный элемент действительно содержит удаляемую величину. Если это не так, то это означает, что элемент подлежащий удалению найден не был, поэтому метод `Remove()` возвращает логическое значение ЛОЖЬ. Продполагая, что найденный элемент это элемент подлежащий удалению производиться уменьшение переменной класса `_count` и ссылки «перепрошиваются». И в конце, если мы удалили элемент с самой большой высотой, то нам нужно уменьшить высоту головного элемента. Это осуществляется вызовом метода `DecrementHeight()` класса `SkipListNode`.

4.2.5 Анализ времени выполнения списка пропусков

В статье «Списки пропусков: вероятностная альтернатива сбалансированным деревьям» ("Skip Lists: A Probabilistic Alternative to Balanced Trees") Пуф предложил краткое доказательство, показывающее, что время выполнения операций поиска, вставки и удаления над списком пропусков асимптотически ограничено величиной $\log_2 n$ в общем случае. Однако, список пропусков может показать линейное время в худшем случае, но вероятность возникновения такого худшего случая очень очень очень очень невелика.

Так как высоты элементов списка пропусков выбираются случайным образом, то существует шанс, что все, или виртуально все, элементы в списке пропусков будут иметь одинаковую высоту. Например, представим, что мы имеем список пропусков со 100 элементами, и что все их высоты были случайным образом выбраны равными 1. Такой список пропусков по своей сути будет обычным связанным списком, например таким, как показан на Рисунке 8. Как мы это уже обсуждали ранее время выполнения операций для обычного связанного списка является линейным.

В то время как такие сценарии наихудшего случая возможны, следует помнить, что они маловероятны. В качестве примера отметим, что вероятность иметь список пропусков со 100 элементами высотой 1 такая же как и вероятность бросания монеты 100 раз и все 100 раз выпадал бы «орел». Шансы, что это произойдет в точности равны 1 из 1,267,650,600,228,229,401,496,703,205,376. Конечно же, с увеличением количества элементов, вероятность снижается еще больше. Более детальную информацию о вероятностном анализе списков пропуска (skip lists) вы можете найти в статье Пуфа (Pugh).

4.2.6 Исследование некоторых эмпирических результатов

График на Рисунке 16 показывает среднее число сравнений на одну операцию для увеличивающегося размера списка пропусков. Отметьте, что если список пропусков удваивается в размере, то среднее число сравнений необходимое на одну операцию увеличивается незначительно (на одно или два сравнения больше). Чтобы полностью понимать смысл логарифмического роста давайте рассмотрим какова будет плата временем за операцию поиска. Для массива из 256 элементов в среднем потребуется 128 сравнений, чтобы найти элемент. Для массива из 512 элементов в среднем потребуется 256 сравнений. Давайте теперь посмотрим на списки пропусков с 256 и 512 элементами – они потребуют только 9 и 10 сравнений в среднем, соответственно.

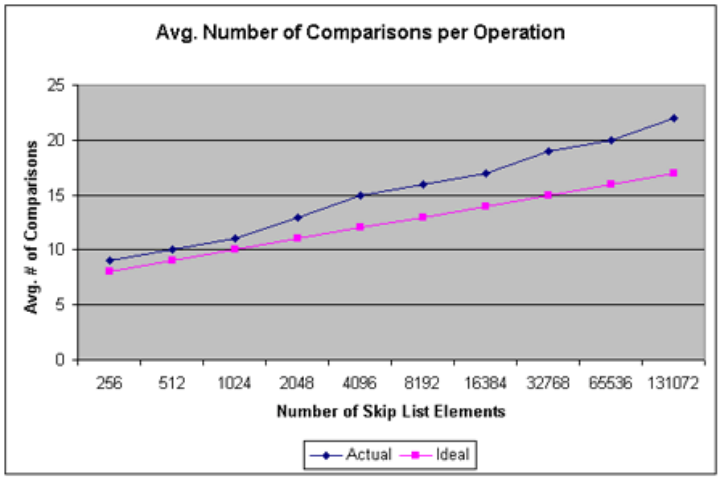

Рисунок 16. График логарифмического роста числа сравнений требуемых для списков пропуска с увеличивающимся числом элементов.

Среднее число сравнений на операцию

Среднее число сравнений

Число элементов в списке пропусков

Реальное

Идеальное

5 Заключение

В Части III этой серии статей мы рассмотрели бинарные деревья и бинарные деревья поиска. BST предоставляют эффективное время выполнения $\log_2 n$. Однако, время выполнения чувствительно к топологии дерева и дерево с субоптимальным отношением ширины к высоте может снизить время выполнения операций над BST до линейного времени.

Чтобы улучшить это время худшего случая для BST, которые могут появляться достаточно легко так как топология BST напрямую зависит от порядка в котором добавляются элементы, было изобретено несметное число самобалансирующихся BST, начиная с АВЛ дерева созданного в 1960-х годах. В то время как такие структуры данных как АВЛ дерево, красно-черное дерево, и множество других специализированных BST предлагают время выполнения $\log_2 n$ как в среднем таки в худшем случаях, они требуют особо сложного кода, который сложно как создавать так и поддерживать.

Альтернативная структура данных, которая предлагает такое же асимптотическое время выполнения как и самобалансирующееся BST, это список пропусков Вильяма Пуфа (William Pugh). Список пропусков это специализированный отсортированный связанный список элементы которого характеризуются ассоциированной с ними высотой. В этой статье мы сконструировали класс `SkipList` и увидели, как легко реализовать операции для работы со списком пропусков.

Эта четвертая часть в серии статей завершает наше обсуждение деревьев. В пятой части мы рассмотрим графы. Граф это набор вершин с произвольным числом ребер, соединяющих узлы друг с другом. Как мы увидим в Части V, деревья это частный случай графов. Графы имеют необычное количество применений в реальном мире.

Счастливого программирования!

Игорь Изварин