Деревья поиска

Очень часто в программировании возникает задача построения ассоциативного массива, т. е. массива, индексами которого являются либо вовсе не целые числа (например, строки или вещественные числа), либо целые числа из очень широкого диапазона, при том, что используется реально лишь очень небольшая часть индексов, так что заводить отдельную ячейку под каждое возможное значение индекса нецелесообразно.

В дальнейшем индекс в ассоциативном массиве, по которому ищется элемент, будем называть ключом, а сам элемент — данными. Термин «ключ» взят из теории баз данных.

Самыми простыми решениями задачи построения ассоциативного массива являются списки и массивы пар (ключ, данные). Однако эти решения обладают рядом недостатков.

Списки обеспечивают быструю вставку и удаление элементов из произвольной позиции (при условии, что мы знаем указатель на элемент в требуемой позиции), но поддерживают только линейный поиск, что в большинстве случаев неприемлемо медленно.

Массивы могут обеспечить быстрый (двоичный или интерполяционный) поиск, но для этого массив должен быть упорядочен, и вставка в упорядоченный массив требует сдвига большого числа элементов, что также неприемлемо медленно.

Дальнейшее развитие технологии предлагает использовать для построения ассоциативных массивов деревья. В общем существует два основных способа использования деревьев для организации ассоциативных массивов.

Деревья поразрядного поиска

Первый способ — так называемые деревья поразрядного поиска. В основе данной технологии лежит следующая идея. Предположим, у нас имеется функция, которая по любому допустимому значению x ключа и неотрицательному целому номеру n выдает значение n-го бита x, причем если все биты у двух ключей совпадают, то такие ключи также совпадают. Нумерация битов начинается с 0, причем 0-й — это младший бит (для целых чисел). Если ключи представляют собой целые числа, эта функция устроена очевидным образом; если ключи представляют собой вещественные числа или строки, она устроена чуть сложнее; на самом деле, такая функция существует для любого типа ключа, который можно использовать в компьютере, поскольку если ключ хранится в памяти, то он (с точностью до некоторой процедуры, называемой сериализацией — нужно только для связных структур вроде списков и т. д.) представляет собой последовательность битов.

Дерево поразрядного поиска — двоичное дерево, каждый узел которого содержит пару из ключа и данных, а также указатели на сыновей, в рамках данной технологии называющихся не «правый» и «левый», а «нулевой» и «единичный». В корневом узле может храниться любой ключ. В «нулевом» поддереве корневого узла содержатся только ключи, нулевой по номеру бит которых (в смысле функции из предыдущего абзаца, т. е. младший бит) равен 0, в «единичном» — 1.

Далее, в соответствии с вышесказанным, в «нулевом» сыне корня может храниться любой ключ, младший бит которого равен 0. В его «нулевом» поддереве встречаются только ключи, у которых как младший, так и следующий за ним по старшинству биты имеют значение 0, а в «первом» поддереве — только те ключи, у которых последние два бита имеют значение 1 и 0.

Итак, на глубине i (корень имеет глубину 0) идет разделение ключей по значению i-го бита (в соответствии с функцией, выдающей значение битов по номеру). Никаких других ограничений на ключи нет; таким образом, у ключа, стоящего на глубине i, фиксированы только i младших битов, а все остальные могут иметь любые значения.

Это свойство позволяет перенести на такие деревья алгоритм двоичного поиска в несколько видоизмененном варианте (по значению битов, начиная с младших), и время, затрачиваемое на основные операции с таким деревом (вставка/поиск/удаление пары) определяется не количеством элементов в дереве, а его высотой, которая, в свою очередь, ограничена максимальной длиной ключа в битах.

Для ключей потенциально небольшой длины (например, целых или вещественных чисел, или не очень длинных строк) такое решение задачи о построении ассоциативного массива вполне приемлемо.

Задачи

0. Написать функцию digit с двумя целыми параметрами x и n, возвращающую значение n-го бита числа x.

1. Написать шаблонный класс DSNode узла дерева поразрядного поиска. В нем должны быть следующие поля: ключ (в данном разделе всегда будет целое число типа int), данные (типа шаблонного параметра) и указатели на сыновей.

2. Написать класс DSTree (единственное поле — указатель на корень дерева), содержащий конструктор по умолчанию (пустое дерево) и деструктор.

3. Добавить в класс DSTree метод print, печатающий дерево (как обычно, но для каждого узла печатаются ключ и данные).

4. Добавить в класс DSTree метод search, принимающий ключ в качестве параметра и возвращающий указатель на узел дерева, содержащий этот ключ. Если такого узла в дереве нет, метод должен вернуть nullptr.

5. Добавить в класс DSTree метод insert, вставляющий новую пару (ключ, данные). Подсказка: проще всего сделать это при помощи рекурсии, написав вспомогательную рекурсивную процедуру, получающую (по ссылке) указатель на узел, в поддерево которого надо вставить пару (ключ, данные), глубину этого узла в исходном дереве, значения ключа и данных.

6. Добавить в класс DSTree метод del, удаляющий узел с заданным ключом. Подсказка: если у удаляемого узла есть хотя бы один сын, мы находим любой лист x в поддереве удаляемого узла и ключ и данные листа x переписываем в тот узел, который хотели удалить, и вместо него удаляем лист x.

7. Написать программу, которая вводит с клавиатуры имя файла, в котором записана последовательность целых чисел через пробел, и выводит таблицу, в которой указано, сколько раз в файле встречается каждое число (не обязательно выводить встречающиеся в файле числа по порядку).

8. Добавить в класс DSTree метод searchMask1, принимающий маску в качестве параметра. Этот метод должен осуществлять поиск по ключу с маской. Известными и равными единице считаются только те биты ключа, для которых соответствующие биты маски имеют значение 1. Значения остальных битов ключа могут быть любыми. Этот метод должен печатать на экран все имеющиеся в дереве пары (ключ, данные) с подходящими ключами.

9. Добавить в класс DSTree метод searchMask2, принимающий ключ и маску в качестве параметров. Этот метод должен осуществлять поиск по ключу с маской. Считаются известными только те биты ключа, для которых соответствующие биты маски имеют значение 1. Значения таких битов берутся из ключа-параметра метода. Значения остальных битов ключей в узлах могут быть любыми. Этот метод должен печатать на экран все имеющиеся в дереве пары (ключ, данные) с подходящими ключами.

10. Добавить в класс DSTree метод replace, заменяющий ключи вида $3n + 5$, где $n = 1, 2, \ldots, 7$ на $10 - n$ (при этом данные сохраняются, т. е. если в дереве была пара из ключа 11 с данными 146, то она заменяется на пару (8, 146); в данном примере $n = 2$).

11. Написать функцию, принимающую два дерева поразрядного поиска (объекта класса DSTree) и возвращающую дерево, содержащее пары (ключ, данные) из обоих деревьев. Если один и тот же ключ присутствует в обоих деревьях с разными данными, берутся данные из первого дерева. При этом оба исходных дерева не должны меняться.

Двоичные деревья поиска

Наряду с деревьями поразрядного поиска, задача хранения и поиска информации имеет и другое решение, также построенное с использованием деревьев, но несколько на другом принципе. Речь идет о двоичных деревьях поиска.

Такие деревья точнее соответствуют идее двоичного поиска. Как и у деревьев поразрядного поиска, в каждом узле таких деревьев имеется пара из ключа и данных; однако принцип записи этих пар в узлы дерева несколько отличается.

Как для построения дерева поразрядного поиска была необходима функция, возвращающая значение бита ключа с определенным номером, так для построения двоичного дерева поиска необходимо, чтобы ключи можно было сравнивать. Как правило, для традиционных наборов ключей (целые, вещественные числа, строки) это так.

Однако, ключами в двоичном дереве поиска могут быть значения любого типа, лишь бы для них была определена функция, которая их сравнивает, т. е. принимает два параметра такого типа и выдает логическое значение (true, если первый параметр меньше второго, и false иначе). Обычно, в качестве такой функции используется перегруженная версия операции $<$. От этой функции требуется только, чтобы задаваемое ею отношение действительно являлось отношением линейного порядка. Если f(x,y) — такая функция, то от нее требуется выполнение следующих условий:

1) Для любого x значение f(x,x) ложно.

2) Для любых x и y f(x,y) и f(y,x) не могут одновременно быть истинными.

3) Транзитивность: если f(x,y) и f(y,z) оба истинны, то f(x,z) тоже истинно.

4) Линейность: для любых различных x и y, всегда f(x,y) или f(y,x) истинно.

Здесь имеет место удивительное явление: смысл задаваемого этой функцией отношения порядка не играет никакой роли. Важно только, чтобы были выполнены вышеприведенные 4 условия, и чтобы функцию f можно было относительно просто вычислить; имеет ли при этом задаваемое такой функцией отношение порядка вообще какой-либо смысл для тех объектов, которые мы беремся сравнивать, совершенно не важно! Например, если мы разрабатываем класс множества и хотим его объекты использовать в качестве индексов ассоциативного массива, первый вариант упорядочения, приходящий в голову — упорядочение множеств по включению. Однако, такой способ упорядочивания множеств, хотя и является естественным для множеств, обеспечивает лишь частичный (не линейный) порядок. А вот если определить порядок на множествах, например, как лексикографический порядок для упорядоченных по возрастанию наборов элементов множества, то он будет линейным, и следовательно, подходит для построения ассоциативного массива с индексами-множествами, несмотря на то, что такой порядок естественным для множеств не является, т. е. его смысл для двух сравниваемых множеств кажется очень странным.

Все дальнейшее будет сказано в предположении, что для интересующего нас типа ключей у нас уже имеется такая сравнивающая функция (удовлетворяющая всем 4 условиям). Мы будем обозначать ключи латинскими буквами и писать между ними неравенства, имея в виду, что либо они принадлежат традиционному набору (числа или строки), тогда имеется в виду обычное отношение порядка; либо для них определена некоторая сравнивающая функция, и отношение порядка определяется ею (хотя мы и пишем $x < y$ вместо «f(x,y) истинно»).

Двоичное дерево, в каждом узле которого стоит пара из ключа и данных, называется двоичным деревом поиска, если в каждом его узле выполняется следующее условие: все ключи в левом поддереве меньше, а в правом — больше ключа в данном узле (повторяющиеся ключи не допускаются по очевидным причинам).

Для поиска и вставки информации в двоичное дерево поиска удобно написать отдельную от метода рекурсивную функцию, в которую указатель на узел дерева передается явно первым параметром.

Свойство двоичного дерева поиска позволяет организовать поиск в таком дереве по тому же принципу, как и двоичный поиск в отсортированном массиве (почему собственно такое дерево и называется двоичным деревом поиска). Проще всего описать такой поиск рекурсивно. Будет два параметра: указатель на узел, в поддереве которого мы ищем

нужный ключ, и само значение искомого ключа. Если указатель на узел равен nullptr, надо вернуть nullptr — ничего не найдено; если же искомый ключ равен значению ключа в узле, на который указывает указатель — вернуть этот указатель; иначе, если искомый ключ меньше значения ключа в узле, на который указывает указатель — искать (при помощи той же функции поиска) в левом поддереве, больше — в правом.

Вставка информации, т. е. новой пары (ключ, данные), в двоичное дерево поиска выполняется похожим образом, только указатель на узел проще всего передавать по ссылке, и будет один дополнительный параметр — значение данных. Если указатель на узел равен nullptr — присваиваем ему указатель на новый узел, построенный с помощью new и заполняем поля надлежащими значениями — это и есть вставка. Если же нет — сравниваем ключ, который хотим вставить, со значением ключа в том узле, на который указывает указатель-параметр, и в зависимости от результатов этого сравнения вставляем новую пару в правое или левое поддерево при помощи рекурсивного вызова той же функции. В итоге, как и в дереве поразрядного поиска, новые пары (ключ, данные) вставляются только в листья нашего дерева. Правда, последующие вставки могут привести к тому, что эти конкретные узлы перестанут быть листьями.

Удаление узла по значению ключа — значительно более сложная функция. Здесь уместно выделить несколько случаев. Первый случай — когда у удаляемого узла имеется не более одного сына. В этом случае все просто — узел удаляется, а его единственный сын (если такой был) вместе со всем своим поддеревом подтягивается наверх, становясь сыном отца удаляемого узла с той стороны, с которой удаляемый узел был прикреплен к своему отцу.

Второй, более сложный случай — когда у удаляемого узла есть оба сына. В этом случае мы ищем самого левого потомка правого сына удаляемого узла. Этот потомок обладает двумя важными свойствами: во-первых, он содержит следующее по величине значение ключа за удаляемым, и, во-вторых, у него нет левого сына. Мы переписываем его ключ и данные в тот узел, который собирались удалять, и удаляем его (так как у него нет левого сына, см. первый случай).

Кроме того, важно также рассматривать случаи, удаляем ли мы корень или нет.

Есть и другие способы удаления узла с двумя сыновьями из двоичного дерева поиска, но по соображениям, которые мы изучим в дальнейшем, этот способ предпочтительнее.

Задачи

1. Написать шаблонный класс двоичного дерева поиска (шаблонные параметры, а их теперь будет два, — типы ключей и данных) с конструктором по умолчанию (пустое дерево) и деструктором.
2. Добавить в класс двоичного дерева поиска метод поиска по значению ключа.
3. Добавить в класс двоичного дерева поиска метод поиска максимального ключа.
4. Добавить в класс двоичного дерева поиска метод поиска второго по величине ключа (следующего за минимальным).
5. Добавить в класс двоичного дерева поиска метод вставки пары ключ, значение.
6. Добавить в класс двоичного дерева поиска метод, принимающий параметр того же типа, что и ключи, и два указателя на узлы по ссылке, и устанавливающий эти указатели на такие узлы с соседними по порядку ключами, что параметр типа ключа лежит между этими ключами. Если этот параметр равен одному из ключей, значения указателей должны быть равны; если же параметр больше наибольшего или меньше наименьшего ключа дерева, соответствующий указатель (с той стороны, с которой ключи дерева не ограничивают параметр) должен быть равен nullptr.
7. Добавить в класс двоичного дерева поиска метод, исправляющий дерево, если известно, что один из листовых узлов стоит не на своем месте.
8. Добавить в класс двоичного дерева поиска метод удаления по значению ключа.
9. Добавить в класс узла двоичного дерева поиска указатель на отца, и изменить соответствующим образом решения задач 5 и 8.
10. Написать функцию, принимающую указатель на узел и возвращающую указатель на узел, содержащий следующее по порядку значение ключа.
11. Написать функцию, принимающую указатель на корень двоичного дерева поиска и удаляющую узлы с ключами > 50.

АВЛ-деревья

Определение двоичного дерева поиска позволяет перенести на такие деревья алгоритм двоичного поиска (откуда и происходит название таких деревьев), и время, затрачиваемое на основные операции с таким деревом (вставка/поиск/удаление пары) определяется не количеством элементов в дереве, а его высотой. В лучшем случае высота примерно равна $\log_2 N$, где N — число узлов дерева, т. е. число хранящихся в ассоциативном массиве элементов.

Однако, двоичное дерево поиска также имеет один существенный недостаток. Дело в том, что при фиксированном наборе ключей основное свойство двоичного дерева поиска не определяет структуру дерева однозначно, и в худшем случае такое дерево может вырождаться в список (например, у каждого узла нет левого сына). Так происходит, если мы добавляем элементы в ассоциативный массив в порядке возрастания ключей. В этом случае двоичное дерево поиска ничем не отличается от обычного связного списка со всеми его недостатками.

Для исправления такой ситуации было предложено понятие сбалансированного дерева. В самом жестком варианте дерево сбалансировано, если число узлов в левом и правом поддереве каждого узла отличается максимум на 1. Очевидно, высота такого дерева всегда приблизительно равна $\log_2 N$.

Однако, такой подход обладает недостатками упорядоченного массива — в дерево с такими требованиями очень трудно добавить новый узел; точнее, добавить просто, но это может привести к нарушению условия сбалансированности, а вот восстановить такую сбалансированность — очень трудно и долго.

Поэтому в итоге решение проблемы состоит в том, чтобы ослабить требование сбалансированности. Известно два варианта такого ослабления.

Первый вариант был предложен в 1962 году Г. М. Адельсоном-Вельским и Е. М. Ландисом, по первым буквам фамилий которых и получили свое название АВЛ-деревья. Их предложение состояло в том, чтобы ослабить требование сбалансированности, заменив его АВЛ-условием: высоты поддеревьев любого узла должны отличаться не более, чем на 1.

Оказывается, во-первых, этого достаточно, чтобы высота дерева не превосходила $C \log_2 N$, где N — число узлов дерева и C — ни от чего не зависящая константа. Во-вторых, после добавления или удаления из дерева одного узла справедливость АВЛ-условия может быть восстановлена за время, ограниченное другой константой, умноженной на высоту дерева.

Задачи

1. Написать класс AVLNode узла АВЛ-дерева. В нем должны быть, кроме ключа, данных (для простоты и то, и другое — целые числа) и указателей на сыновей, следующие поля: указатель на родительский узел и высота (целое число). Кроме того, должны быть реализованы методы: конструктор, инициализирующий поля значениями своих параметров, correct_h, вычисляющий высоту узла на основании высот сыновей и записывающий результат в соответствующее поле, defect_h, возвращающий разность высот левого и правого сыновей, и print, печатающий данные узла (указатель this и значения полей).

2. Написать класс AVLTree (единственное поле root — указатель на корень дерева), содержащий конструктор по умолчанию (пустое дерево), конструктор копирования, операцию присваивания, деструктор.

3. Добавить в класс AVLTree метод print, печатающий дерево.

4. Добавить в класс AVLTree методы rotate_left (левый поворот) и rotate_right (правый поворот). Единственный параметр — указатель на узел, в котором производится поворот, по ссылке (чтобы менять значение этого указателя в соответствии с преобразованием поворота). Не забыть также поменять правильным образом указатели в дереве и root, если нужно. Рассмотреть случаи, когда указатель, переданный в качестве параметра, является частью дерева (т. е. это сам root или указатель, являющийся полем какого-либо узла дерева) или нет.

5. Добавить в класс AVLTree методы correctm2 и correct2, принимающие в качестве параметра указатель на узел дерева по ссылке и меняющие поддерево этого узла при помощи поворотов так, чтобы оно удовлетворяло АВЛ-условию. Можно предполагать, что все нижестоящие узлы удовлетворяют АВЛ-условию, а в самом узле, на который указывает параметр, разность высот левого и правого поддеревьев равна -2 и 2 соответственно. Подсказка: имеется всего два случая, и в одном из них нужен один дополнительный поворот, и затем поворот, нужный в обоих случаях.

6. Добавить в класс AVLTree метод rebalance, исправляющий АВЛ-свойство дерева, нарушенное вследствие изменения высоты одного из узлов на 1. Единственным параметром является указатель на самый глубокий узел, высота которого изменилась.

7. Добавить в класс AVLTree метод insert, вставляющий новую пару ключ-данные.

8. Добавить в класс AVLTree методы для удаления корня/правого/левого сына своего отца, не имеющего правого/левого сына (всего 6 вариантов).

9. Добавить в класс AVLTree метод del, удаляющий узел с заданным ключом. Подсказка: если у удаляемого узла имеются оба сына, мы находим узел x, содержащий следующее по величине (за удаляемым) значение ключа, и ключ и данные этого узла переписываем в тот узел, который хотели удалить, и вместо него удаляем узел x.

10*. Используя идеи, лежащие в основе АВЛ-деревьев, для хранения упорядоченной последовательности элементов разработать дерево, позволяющее быстро получать элемент по номеру, а также вставлять элемент в позицию с указанным номером или удалять его.

11*. Для хранения неупорядоченной последовательности элементов разработать дерево, позволяющее быстро получать элемент по номеру и номер по элементу, а также вставлять элемент в дерево и удалять его (номер в этом случае определяется автоматически).

12*. Разработать структуру данных для решения предыдущей задачи, но для упорядоченной последовательности элементов.

B-деревья

B-деревья — еще один вид сбалансированных деревьев. B-деревья изобрели Байер (Bayer) и МакКрейт (McCreight) в 1970 году.

Назначение B-деревьев.

Известно, что доступ к жесткому диску осуществляется гораздо медленнее, чем к оперативной памяти. Кроме того, за одну операцию чтения или записи на жесткий диск можно считать или записать только целое число блоков (блок — последовательность байтов определенной фиксированной длины, зависящей от операционной системы, объема жесткого диска и типа файловой системы, т. е. способа организации хранения файлов, на нем).

Поэтому при разработке способа хранения информации на жестком диске обычно минимизируют количество раз доступа к нему, исходя из того, что практически любая обработка полученного с диска блока выполняется гораздо быстрее, чем само считывание этого блока.

Поскольку малый размер узла дерева не дает никаких преимуществ в скорости работы (все равно меньше блока за один раз не считать), у B-деревьев размер одного узла примерно равен одному блоку. Это позволяет хранить в одном узле большое количество ключей и указателей на сыновей, что обеспечивает высокую степень ветвления; в свою очередь, это приводит к очень маленькой высоте дерева даже при огромном числе узлов. Но скорость работы деревьев поиска пропорциональна высоте дерева, и в итоге B-дерево для стандартных операций по его обработке (добавление новой пары ключ/данные, поиск, удаление) требует считывания или изменения очень небольшого числа блоков на диске, что существенно повышает производительность B-деревьев по сравнению с другими деревьями поиска, если дерево хранится на диске.

При всем том, B-дерево остается деревом поиска, т. е. в каждом его узле для каждого его сына хранится минимальное значение ключей соответствующего поддерева. Это позволяет при поиске информации в B-дереве во время анализа конкретного узла определить, в поддереве какого сына лежит (если он вообще имеется в дереве) конкретный ключ.

Однако, для того, чтобы B-дерево не вырождалось, на него накладывается дополнительное требование: глубина всех листьев должна быть одинакова. Чтобы выполнить это требование, и вместе с тем, иметь возможность быстро производить над деревом стандартные операции, вводится еще одно условие: число сыновей каждой вершины должно лежать в пределах от $N/2$ (включая) до N (не включая), где N — максимально возможное число записей, описывающих одного сына, которое может быть записано в один узел. Таким образом, в каждом узле имеется массив для описания его сыновей, заполненный не менее, чем наполовину, но не полностью. Единственным исключением из этого правила является корневой узел, в котором должно быть не менее двух записей, если в дереве кроме корневого узла имеются другие узлы. Удобно, чтобы N было четным.

В B-деревьях, которые используются в реальности, данные хранятся только в листьях; в промежуточных узлах хранятся только ключи и указатели на сыновей. Однако, для упрощения ситуации, мы рассмотрим вариант B-дерева с одинаковым типом узла, не зависящим от его положения в дереве, т. е. от того, является он листом или нет.

Пример B-дерева с $N = 4$ высоты 3 (в реальности N велико — порядка сотен или даже тысяч).

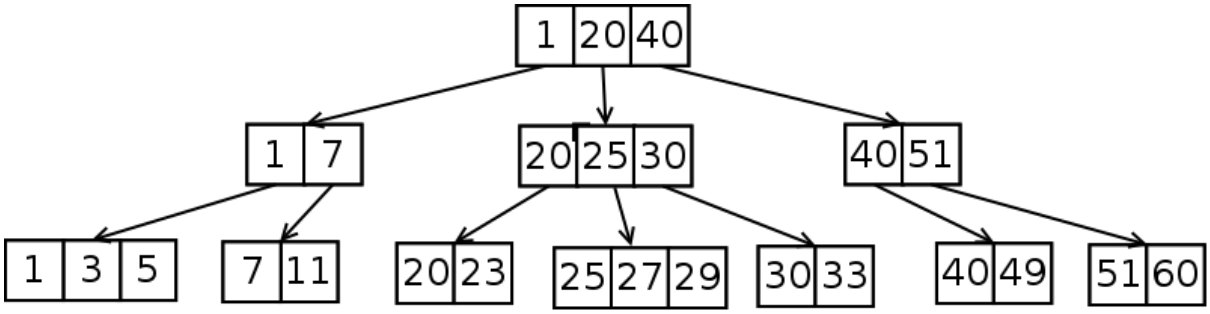

Рис. 1

Узел B-дерева описывается структурой BTNode (см. ниже).

```
const int N=4; /* размер массива
записей о сыновьях в каждом узле */

template<class K, class T>
struct BTNode; /* тип узла дерева */

template<class K, class T>
struct Item /* тип записи об одном сыне */
{
    K key;
    T data;
    BTNode<K, T> *down;
};

template<class K, class T>
struct BTNode
{
    int n; /* реальное число сыновей */
    Item ar[N];
};

template<class K, class T>
struct BTTree /* собственно тип дерева */
```

```
{
    BTNode<K, T> *root; /* корневой узел */
    int height; /* высота дерева */
};
```

Задачи

1. Добавить в класс BTTree конструктор без параметров, строящий пустое дерево. В отличие от двоичных деревьев поиска и деревьев поразрядного поиска, в пустом B-дереве имеется один (корневой) узел без сыновей (поле n в этом узле равно 0).

2. Добавить в класс BTTree деструктор, освобождающий память, занятую узлами дерева. Для этого проще всего завести отдельную рекурсивную функцию, принимающую указатель на корневой узел, и если он не nullptr, вызывающую себя же для всех сыновей этого узла, а затем освобождающую память, занятую этим узлом, при помощи delete. Деструктору остается только вызвать эту функцию с параметром root.

3. Добавить в класс BTTree метод print, печатающий дерево. Как и в предыдущей задаче, проще всего завести отдельную рекурсивную функцию, принимающую указатель на узел и глубину этого узла (считается от корня, его глубина 0) и для каждого сына этого узла печатающую сначала ключ и данные в соответствующей записи, со сдвигом $3n$ пробелов (n — глубина), а затем все поддерево этого сына при помощи рекурсивного вызова той же процедуры.

4. Добавить в класс BTTree метод search с параметром-искомым ключом и возвращающий указатель на элемент данных, соответствующий указанному ключу (если такого нет, возвращается нулевой указатель). Для поиска заданного ключа нужно начать с корня и определить, в каком из поддеревьев его сыновей лежит ключ (это можно сделать, найдя последний ключ в массиве данного узла, который не превосходит искомого; если таких нет, значит ключ меньше минимального ключа в дереве). Затем перейти в это поддерево, пользуясь указателем на сына в найденной записи, и т. д. до тех пор, пока не окажемся в листе, а там уже можно искать сам ключ — если его там нет, значит, его вообще нет в дереве.

5. Добавить в класс BTTree метод insertB, принимающий в качестве параметров указатель p на узел, номер (целое число) i и запись о сыне x (параметр типа Item), и вставляющий элемент x в массив ar записей узла, на который указывает p, в позицию i (все элементы с номерами i и более сдвигаются на одну позицию вправо, общее число сыновей увеличивается на 1). Можно предполагать, что до вставки число сыновей узла $*p$ было меньше N.

6. Добавить в класс BTTree метод split, который принимает в качестве параметра указатель p на узел, заводит новый узел и переписывает в него вторую половину элементов массива ar из узла, на который указывает p. Можно предполагать, что узел $*p$ был заполнен до конца, т. е. в нем было N записей о сыновьях. Этот метод должен устанавливать у обоих узлов (нового и старого) правильное число сыновей и возвращать указатель на новый узел.

7. Добавить в класс BTTree метод insertR, принимающий в качестве параметров указатель на узел (в дальнейшем этот узел называется текущим), ключ, данные и высоту текущего узла и вставляющий указанную пару (ключ, данные) в поддерево указанного узла. Этот метод должен возвращать nullptr, если вставка в текущий узел не привела к его заполнению до конца (N сыновей), и указатель на новый узел, если после вставки текущий узел был заполнен до конца; в этом случае insertR должен расщеплять текущий узел на два при помощи split из предыдущей задачи и вернуть указатель на новый узел, содержащий вторую половину записей из текущего узла. Существует два случая (распознаются по значению высоты текущего узла, переданной как параметр): вставка в лист и в промежуточный узел. Если мы вставляем новую пару из ключа и данных в лист, мы должны найти место для нее (а ключи в записях о сыновьях должны быть упорядочены), осуществить собственно вставку при помощи insertB (ключ и данные берутся из параметров, указатель на сына — nullptr), и проверить, до конца ли заполнен тот лист, куда мы осуществляли вставку. Если нет, вернуть nullptr, иначе вызвать split и вернуть его результат. Если же мы вставляем пару не в лист, нужно сначала определить того сына, в поддерево которого будет осуществляться вставка (а это первый сын, если вставляемый ключ меньше ключа первого сына, или последний сын из тех, ключи которых не больше вставляемого; в первом из этих двух случаев нужно записать вставляемый ключ на место ключа первого сына). Далее, нужно рекурсивно вызвать insertR для найденного поддерева для вставки туда новой пары. Если этот вызов вернул nullptr, на этом процесс вставки можно закончить; в противном случае нужно вставить новый указатель на сына в текущий узел сразу после того сына, в поддерево которого осуществлялась вставка. Ключ в этой новой записи о сыне должен браться из первой записи того узла, на который указывает результат insertR (это и будет минимальный ключ соответствующего поддерева). Наконец, должна быть та же проверка, что и при вставке в лист: до конца ли заполнен текущий узел. Действия, связанные с этой проверкой, тоже аналогичны: если нет, вернуть nullptr, иначе вызвать split и вернуть его результат. На самом деле, все это может быть записано достаточно компактно; как это делается, см. далее. Для этого нужно сделать следующее: сначала найти среди ключей текущего узла первый, который больше вставляемого; затем, подготовить переменную типа Item, которую мы будем вставлять (заполнить ее поля значениями вставляемых ключа и данных, указатель на сына установить в nullptr). Далее, нужно рассмотреть случай, когда текущий узел — не лист (параметр высоты > 1). В этом случае рекурсивно вызывается метод insertR для вставки новой пары в нужное поддерево. Затем, если мы вставляли ключ, который был меньше минимального ключа в текущем узле, нужно подправить значение минимального ключа в первом поддереве, а заодно и позицию, которую мы нашли в самом начале метода. Наконец, если рекурсивный вызов insertR вернул nullptr, нужно закончить, иначе нужно изменить ту переменную типа Item, которую мы подготовили для вставки, а именно записать в поле ключа первый ключ нового узла (указатель на который вернул рекурсивный вызов insertR), а в поле указателя на сына — сам результат insertR. На этом обработка случая, когда мы вставляем не в лист, закончена. Последнее, что надо сделать — надо вставить

подготовленную переменную типа Item в текущий узел на подготовленную позицию (найденную в начале метода и возможно подправленную впоследствии) при помощи insertB, и после этого, если текущий узел не заполнен до конца, вернуть nullptr; если же он заполнен, разделить его на два при помощи split, и вернуть результат этого вызова split.

8. Добавить в класс BTTree метод insert, принимающий в качестве параметров ключ и данные и вставляющий указанную пару (ключ, данные) в дерево при помощи вызова метода insertR. Если он вернул не nullptr, значит, ему пришлось разделить старый корень на два узла; это означает, что надо надстроить новый корень (с двумя записями — одна про старый корень, другая — про новый узел, т. е. результат insertR). Более подробное описание следует ниже. Нужно завести новый корень с двумя записями (одна соответствует старому корню: ключ берется из первой записи старого корня, данные — результат конструктора по умолчанию для класса данных, указатель на сына — старое значение root; другая — результату вызова insertR: ключ берется из первой записи узла, на который указывает результат insertR, данные — результат конструктора по умолчанию для класса данных, указатель на сына — результат insertR). При этом высота всего дерева (поле height) увеличивается на 1.

9. Добавить в класс BTTree метод removeB, принимающий в качестве параметров указатель на узел и номер и удаляющий указанную запись из массива записей узла, на который указывает первый параметр (все записи с большими номерами сдвигаются на одну позицию в сторону уменьшения номеров, общее число сыновей уменьшается на 1).

10. Добавить в класс BTTree метод join, принимающий в качестве параметров указатель p на узел и номер n и соединяющий два узла, на которые указывают записи под номерами n и $n+1$ в узле $*p$, в один. При этом все записи из сына узла $*p$ с номером $n+1$ переписываются в конец массива записей у сына узла $*p$ с номером n, у этого сына подправляется число сыновей, тот сын, из которого были взяты записи, удаляется при помощи delete, и, наконец, запись с номером $n+1$ удаляется из узла $*p$ с помощью метода removeB из предыдущей задачи. Можно предполагать, что суммарное количество записей в обоих объединяемых узлах меньше N, но не меньше $N/2$.

11. Добавить в класс BTTree метод pass_forward, принимающий в качестве параметров указатель p на узел и номер n и переписывающий последнюю запись из сына, на которого указывает n-я запись в $*p$ в сына, на которого указывает $n+1$-я (новая запись должна быть там первой в силу упорядоченности ключей). Вставить запись в новый узел можно при помощи insertB, а удалить из того узла, откуда она берется, можно при помощи removeB. Нужно также не забыть подправить значение ключа в записи того сына, куда попадает новая запись (чтобы в записи этого сына действительно оказался минимальный ключ соответствующего поддерева, который теперь будет равен ключу перемещенной записи).

12. Добавить в класс BTTree метод pass_backward, принимающий в качестве параметров указатель p на узел и номер n и переписывающий первую запись из узла, на который указывает $n+1$-я запись в $*p$ в узел, на который указывает n-я. Это делается аналогично тому, что происходило в предыдущей задаче.

13. Добавить в класс BTTree метод correct_filling, принимающий в качестве параметров указатель p на узел и номер n и исправляющий нагрузку (т. е. число сыновей) узла, на которого указывает n-я запись в $*p$, в предположении, что она равна $N/2-1$. Самый простой способ это сделать — позаимствовать эту запись (при помощи одного из методов двух предыдущих задач) у одного из ближайших братьев, если это не приведет к недостаче записей у него (т. е. после этого у него останется не меньше $N/2$ сыновей). Если же этот способ не сработает, то можно объединить (при помощи join) сына с недостатком записей с соответствующим его братом, в результате чего мы получим узел с $N-1=(N/2-1)+(N/2)$ записями — проблема опять будет решена.

14. Добавить в класс BTTree метод removeR, принимающий в качестве параметров указатель на узел, ключ и высоту узла, на который указывает первый параметр, и две логические переменные corr_key и corr_fill по ссылке и удаляющий пару с указанным ключом из поддерева указанного узла. Механизм удаления следующий: как и в случае вставки, существуют два случая. Если нам надо удалить пару из листа, мы ищем в нем удаляемый ключ; если его нет, делать нечего. Иначе, пользуемся removeB и удаляем интересующую нас пару. Наконец, при помощи логических параметров, переданных по ссылке, сигнализируем вызывавшей функции о двух обстоятельствах: corr_key устанавливается в true, если мы удаляли первую запись — тогда в вышестоящем узле надо подправить ключ в записи, описывающей того сына, с которым мы работали; corr_fill устанавливается в true, если в результате удаления в листе осталось меньше $N/2$ записей. Если мы удаляем пару из промежуточного узла, нужно завести две логические переменные, определить сына, из поддерева которого нужно удалить ключ (это будет последний сын, минимальный ключ поддерева которого не больше удаляемого ключа; если же таких сыновей нет вовсе, это означает, что удаляемый ключ меньше минимального ключа в дереве, значит, такого ключа в дереве нет, и делать нечего). Далее, нужно рекурсивно вызвать метод removeR для этого сына. Затем, нужно проанализировать те логические значения, которые метод вернул: если в поддереве изменился минимальный ключ, соответствующую запись (того сына, для которого был вызван метод), нужно подправить, записав туда новый ключ из первой записи этого сына. Если же после удаления у этого сына образовался недостаток записей, это должно быть исправлено при помощи метода correct_filling. Наконец, делается то же, что и в конце первого случая: проверяем, изменился ли первый ключ (при исправлении ключа в записи сына, в поддереве которого было удаление, после removeR, если этот сын был первым) и не осталось ли в текущем узле (после вызова correct_filling) недостатка в сыновьях; в соответствии с результатами этих проверок устанавливаются параметры corr_key и corr_fill.

15. Добавить в класс BTTree метод remove, принимающий в качестве параметров ключ и удаляющий пару с указанным ключом. Для этого достаточно вызвать метод предыдущей задачи с указателем root (и естественно ключом, а также двумя логическими переменными), при этом на значения логических переменных можно не обращать внимания. Но после вызова removeR, нужно проверить следующее: если в корне осталась одна запись, и высота дерева больше 1 (т. е. кроме корня в дереве есть еще узлы), то нужно удалить корневой узел, указателю на корень присвоить

указатель на единственного сына старого корня, и уменьшить высоту дерева (поле height) на 1. Разумеется, нужно делать это правильно (потому, что, если удалить корень сначала, то и указатель на его единственного сына будет потерян; здесь не обойтись без дополнительной переменной типа указатель на узел). Это действие похоже на удаление первого элемента из списка.

Красно-черные деревья

Красно-черные деревья представляют собой другое, впрочем, не менее популярное, решение задачи о построении балансирующегося двоичного дерева поиска. Они дают несколько более медленный поиск, но более быструю перебалансировку, чем АВЛ-деревья.

На самом деле, красно-черные деревья — это попытка превратить B-дерево с N=4 (см. предыдущий раздел) в двоичное дерево. В таком дереве имеются два типа узлов: узлы с двумя и с тремя сыновьями (в промежуточном варианте, во время работы функций вставки и удаления, возможны также узлы с четырьмя сыновьями).

Для указанного превращения можно воспользоваться представлением узла с тремя сыновьями как фрагмента двоичного дерева, состоящего из двух узлов (один из них является отцом другого), каждый из которых имеет, как положено узлу двоичного дерева, двух сыновей. Узел с четырьмя сыновьями представляется как конструкция из трех двоичных узлов: один из них является отцом оставшихся, которые друг другу являются братьями.

В таком виде функция поиска работает ровно также, как и для обычного двоичного дерева поиска. Однако, для поддержания сбалансированности необходимо знать, какие фрагменты полученного двоичного дерева произошли из каких узлов исходного B-дерева. Для этого у каждого узла вводится дополнительное поле, хранящее его «цвет», который может быть красным или черным. Правила раскраски узлов двоичного дерева следующие:

1) Узел исходного B-дерева с двумя сыновьями при преобразовании сохраняется как есть, и является черным.

2) Узел исходного дерева с тремя сыновьями представляется конструкцией из двух узлов, из которых один является отцом другого. В этом случае отец имеет черный цвет, а сын — красный.

3) Узел исходного дерева с четырьмя сыновьями представляется конструкцией из трех узлов, из которых один является отцом двух других. В этом случае отец имеет черный цвет, а оба сына — красный.

Задачи

1. Написать шаблонный тип узла красно-черного дерева.

2. Написать шаблонный тип красно-черного дерева, содержащий конструктор по умолчанию, конструктор копирования, операцию присваивания, деструктор.

3. Добавить в класс красно-черного дерева метод поиска узла по ключу.

4. Добавить в класс красно-черного дерева метод вставки пары (ключ, данные) (с необходимой перебалансировкой; можно переделать соответствующий метод из B-деревьев).

5. Добавить в класс красно-черного дерева метод удаления пары по значению ключа (тоже с необходимой перебалансировкой; можно опять же переделать соответствующий метод из B-деревьев).

Гибридные деревья

Кроме вышеизложенных основных вариантов использования деревьев для решения задачи об ассоциативном массиве, существуют и гибридные варианты, например, деревья троичного поиска. Если ключи — строки, то такое дерево будет устроено следующим образом: у каждой вершины дерева имеются данные (символьного типа) и не более трех сыновей.

Если нам нужно найти данные по ключу-строке, нужно начать перемещение по дереву с корня, находясь в котором, анализируется первый символ ключа. Если его код меньше, чем код символа в корне, мы переходим в первого (левого) сына корня и продолжаем анализировать первый символ ключа. Если имеет место равенство, переходим во второго (среднего) сына корня, и далее рассматриваем второй символ ключа. Наконец, в оставшемся случае переходим к третьему (правому) сыну корня, продолжая рассматривать первый символ ключа.

Таким образом, мы перемещаемся вниз по дереву, в каждой его вершине рассматривая определенный символ ключа и сравнивая его с тем символом, который хранится в данной вершине. Если имеет место равенство, мы перемещаемся в среднего сына текущей вершины, переходя к рассмотрению следующего символа в ключе. Если же равенства нет, мы выбираем левого или правого сына в зависимости от того, в какую сторону имеем неравенство, и перейдя в него, продолжаем рассматривать тот же символ ключа, что и в текущей вершине.

Так мы движемся вниз по дереву до тех пор, пока не будут рассмотрены до конца все символы ключа. Та вершина, в которую мы при этом попадем, и будет содержать искомую информацию.