Министерство образования Российской Федерации
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. Н.Э. БАУМАНА

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Информационная безопасность (ИУ-8)

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

ДОМАШНЯЯ РАБОТА.
ЗАДАНИЕ № 6:

Сравние В-дерева и косого дерева

Преподаватель:
Чесноков В. О.
Студент:
Дацук Н.А.
ИУ8-53

Москва 2017

Условие Задачи

Сравнить B-дерево и косое дерево

Теоретическая часть

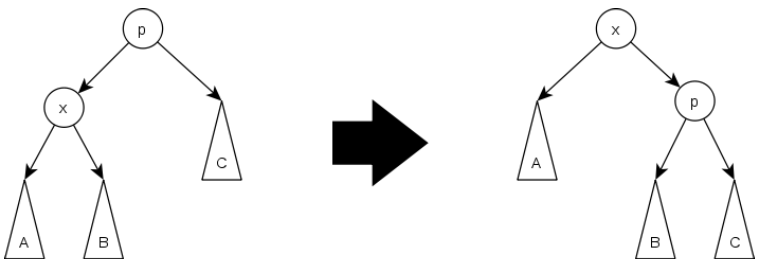
Расширяющееся (англ. splay tree) или **косое дерево** является двоичным деревом поиска, в котором поддерживается свойство сбалансированности. Это дерево принадлежит классу «саморегулирующихся деревьев», которые поддерживают необходимый баланс ветвления дерева, чтобы обеспечить выполнение операций поиска, добавления и удаления за логарифмическое время от числа хранимых элементов. Это реализуется без использования каких-либо дополнительных полей в узлах дерева (как, например, в Красно-чёрных деревьях или АВЛ-деревьях, где в вершинах хранится, соответственно, цвет вершины и глубина поддерева). Вместо этого «расширяющие операции» (splay operation), частью которых являются вращения, выполняются при каждом обращении к дереву.

Основные операции

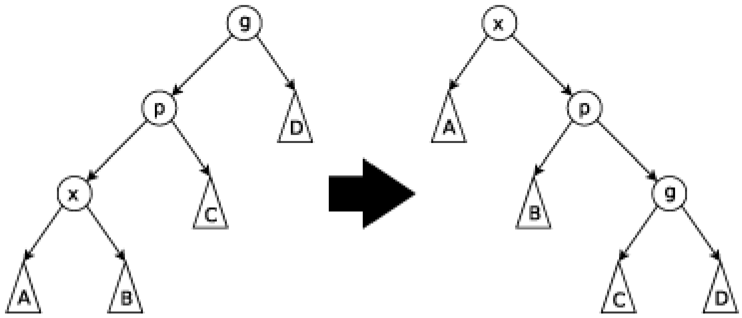
Splay

Основная операция дерева. Заключается в перемещении вершины в корень при помощи последовательного выполнения трёх операций: Zig, Zig-Zig и Zig-Zag. Обозначим вершину, которую хотим переместить в корень за x, её родителя — p, а родителя p (если существует) — g.

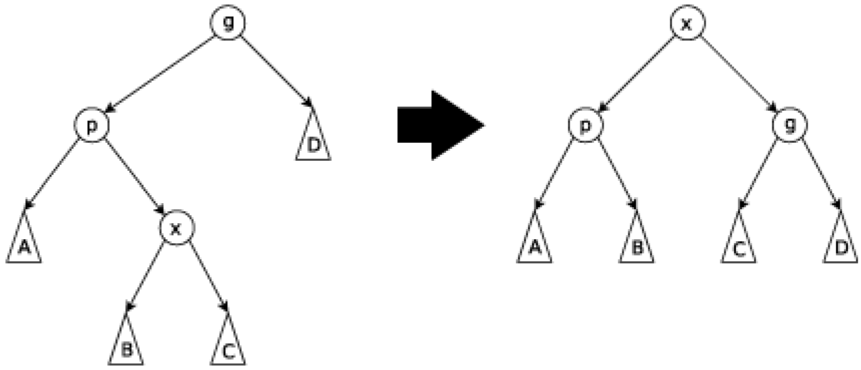
Zig: выполняется, когда p является корнем. Дерево поворачивается по ребру между x и p. Существует лишь для разбора крайнего случая и выполняется только один раз в конце, когда изначальная глубина x была нечётна.



Zig-Zig: выполняется, когда и x, и p являются левыми (или правыми) сыновьями. Дерево поворачивается по ребру между g и p, а потом — по ребру между p и x.



Zig-Zag: выполняется, когда x является правым сыном, а p — левым (или наоборот). Дерево поворачивается по ребру между p и x, а затем — по ребру между x и g.



Search (поиск элемента)

Поиск выполняется как в обычном двоичном дереве поиска. При нахождении элемента запускаем Splay для него.

Insert (добавление элемента)

Вставка происходит как в обычном бинарном дереве поиска, после, запускаем Split() от добавляемого элемента и подвешиваем получившиеся деревья за него.

Delete (удаление элемента)

Находим элемент в дереве, делаем Splay для него, удаляем, делаем текущим деревом Merge его детей.

Merge (объединение двух деревьев)

Для слияния деревьев T1 и T2, в которых все ключи T1 меньше ключей в T2, делаем Splay для максимального элемента T1, тогда у корня T1 не будет правого ребенка. После этого делаем T2 правым ребенком T1.

B-дерево (по-русски произносится как **Би-дерево**) — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

Использование B-деревьев впервые было предложено Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году.

Структура

При построении B-дерева применяется фактор t, который называется минимальной степенью. Каждый узел, кроме корневого, должен иметь, как минимум t – 1, и не более 2t – 1 ключей. Обозначается n[x] – количество ключей в узле x.

Ключи в узле хранятся в неубывающем порядке. Если x не является листом, то он имеет n[x] + 1 детей. Если занумеровать ключи в узле x, как k[i], а детей c[i], то для любого ключа в поддереве с корнем c[i] (пусть k1), выполняется следующее неравенство – k[i-1] ≤k1≤k[i] (для c[0]: k[i-1] = -∞, а для c[n[x]]: k[i] = +∞). Таким образом, ключи узла задают диапазон для ключей их детей.

Все листья B-дерева должны быть расположены на одной высоте, которая и является высотой дерева. Высота B-дерева с n ≥ 1 узлами и минимальной степенью t ≥ 2 не превышает $\log_t(n+1)$. $h \le \log_t((n+1)/2)$ — логарифм по основанию t.

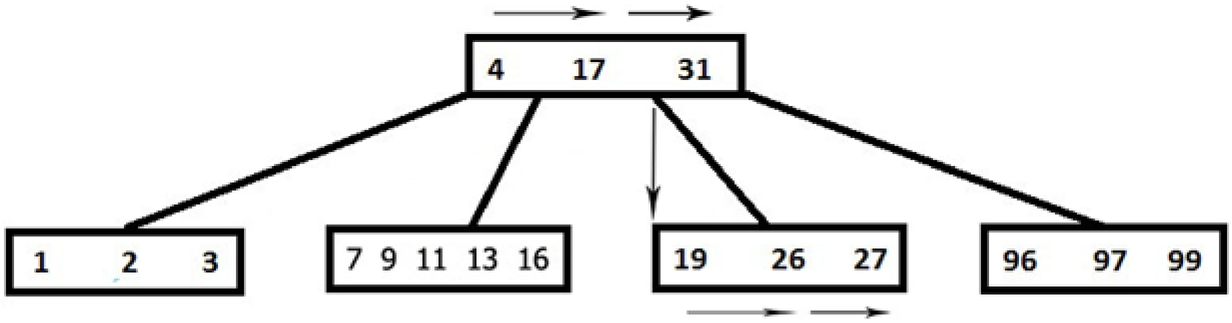
Операции, выполнимые с B-деревом

Как упоминалось выше, в B-дереве выполняются все стандартные операции по поиску, вставке, удалению и т.д.

Поиск

Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска):

- Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31.
- Спускаемся к ребенку, который находится левее этого ключа.
- Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.

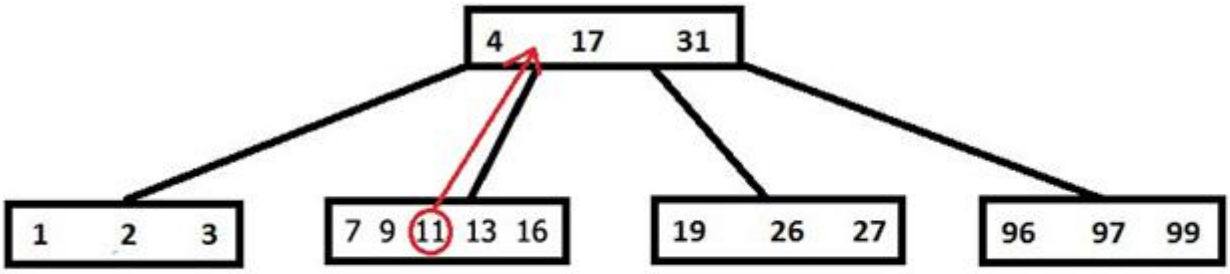


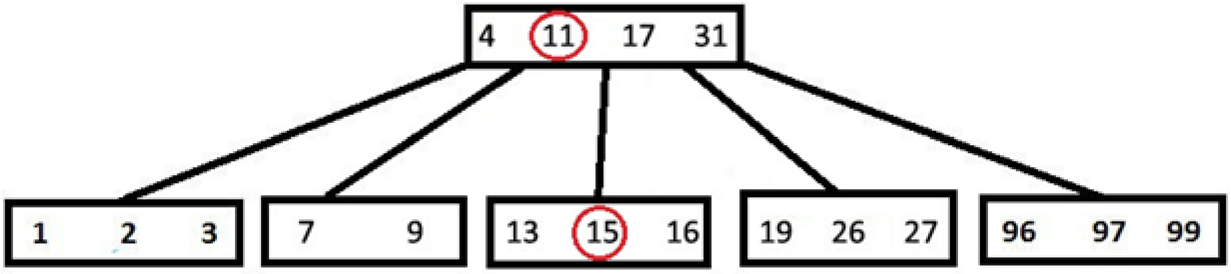
Операция поиска выполняется за время O(t logt n), где t – минимальная степень. Важно здесь, что дисковых операций мы совершаем всего лишь O(logt n)!

Добавление

В отличие от поиска, операция добавления существенно сложнее, чем в бинарном дереве, так как просто создать новый лист и вставить туда ключ нельзя, поскольку будут нарушаться свойства B-дерева. Также вставить ключ в уже заполненный лист невозможно => необходима операция разбиения узла на 2. Если лист был заполнен, то в нем находилось 2t-1 ключей => разбиваем на 2 по t-1, а средний элемент (для которого t-1 первых ключей меньше его, а t-1 последних больше) перемещается в родительский узел. Соответственно, если родительский узел также был заполнен – то нам опять приходится разбивать. И так далее до корня (если разбивается корень – то появляется новый корень и глубина дерева увеличивается). Как и в случае обычных бинарных деревьев, вставка осуществляется за один проход от корня к листу. На каждой итерации (в поисках позиции для нового ключа – от корня к листу) мы разбиваем все заполненные узлы, через которые проходим (в том числе лист). Таким образом, если в результате для вставки потребуется разбить какой-то узел – мы уверены в том, что его родитель не заполнен!

На рисунке ниже проиллюстрировано то же дерево, что и в поиске (t=3). Только теперь добавляем ключ «15». В поисках позиции для нового ключа мы натыкаемся на заполненный узел (7, 9, 11, 13, 16). Следуя алгоритму, разбиваем его – при этом «11» переходит в родительский узел, а исходный разбивается на 2. Далее ключ «15» вставляется во второй «отколовшийся» узел. Все свойства B-дерева сохраняются!



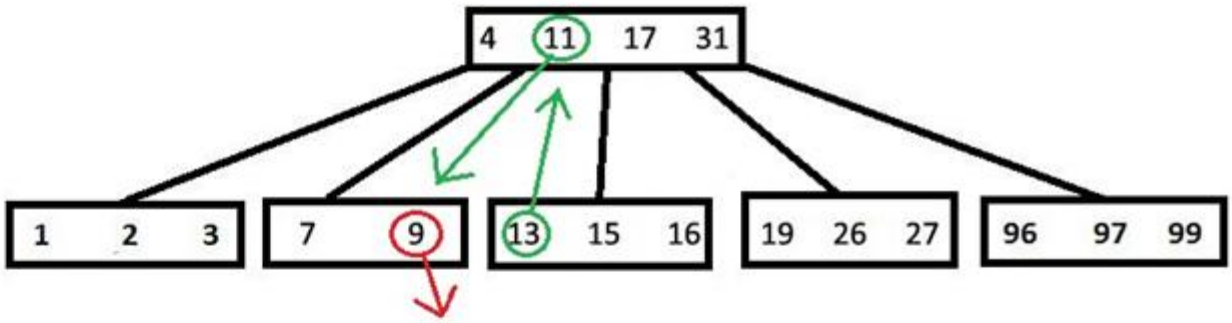


Операция добавления происходит также за время O(t logt n). Важно опять же, что дисковых операций мы выполняем всего лишь O(h), где h – высота дерева.

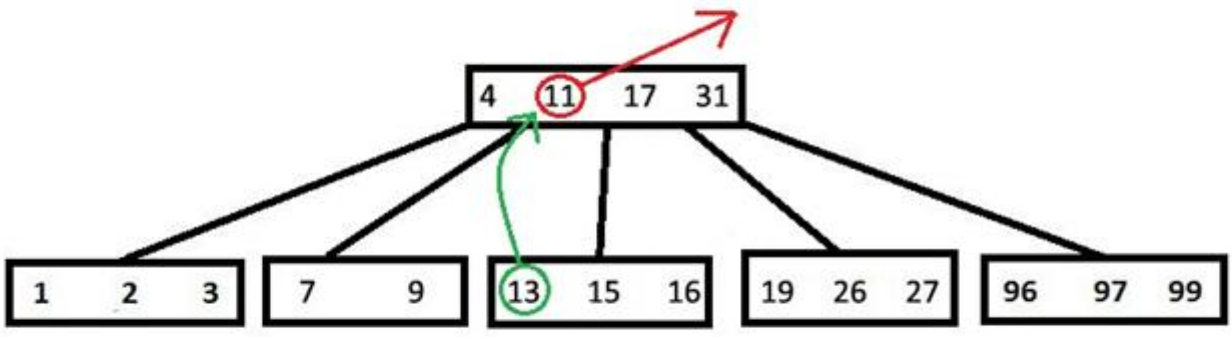
Удаление

Удаление ключа из B-дерева еще более громоздкий и сложный процесс, чем вставка. Это связано с тем, что удаление из внутреннего узла требует перестройки дерева в целом. Аналогично вставке необходимо проверять, что мы сохраняем свойства B-дерева, только в данном случае нужно отслеживать, когда ключей t-1 (то есть, если из этого узла удалить ключ – то узел не сможет существовать). Рассмотрим алгоритм удаления:

1)Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем. Если больше t-1, то просто удаляем и больше ничего делать не нужно. Иначе, если существует соседний лист (находящийся рядом с ним и имеющий такого же родителя), который содержит больше t-1 ключа, то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла-соседа и исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Пусть это ключ k1. Выберем ключ k2 из узла-родителя, который является разделителем исходного узла и его соседа, который мы выбрали ранее. Удалим из исходного узла нужный ключ (который необходимо было удалить), спустим k2 в этот узел, а вместо k2 в узле-родителе поставим k1. Чтобы было понятнее ниже представлен рисунок (рис.1), где удаляется ключ «9». Если же все соседи нашего узла имеют по t-1 ключу. То мы объединяем его с каким-либо соседом, удаляем нужный ключ. И тот ключ из узла-родителя, который был разделителем для этих двух «бывших» соседей, переместим в наш новообразовавшийся узел (очевидно, он будет в нем медианой).



2)Теперь рассмотрим удаление из внутреннего узла x ключа k. Если дочерний узел, предшествующий ключу k содержит больше t-1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его (рекурсивно запускаем наш алгоритм). Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t-1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t-1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла (рекурсивно запускаем наш алгоритм). Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается. Ниже представлен рисунок (рис.2), где из корня удаляется «11» (случай, когда у следующего узла больше t-1 ребенка).



Операция удаления происходит за такое же время, что и вставка O(t logt n).

План решения задания

1. Реализовать оба дерева на языке C++
2. С помощью различных тестов получить информацию о времени работы обоих деревьев
3. Сравнивать полученные результаты
4. Сделать выводы

Практическая часть

Ссылка на программу, размещенную в публично доступном репозитории сервиса GitHub: https://github.com/ILLLIGION/SplayTree-vs-BTree. Описании программы, формат входных и выходных данных, а так же инструкция по использованию и компилированию программы находятся в файле README.md в репозитории.

Итоговое теоретическое сравнение времени работы всех основных операций над обоими деревьями представлено в таблице 1.

Таблица 1

Операция	Splay Tree	B Tree
Поиск	O(logn)	$O(t*\log_t n)$
Вставка	O(logn)	$O(t*\log_t n)$
Удаление	O(logn)	$O(t*\log_t n)$
Поиск минимального ключа	O(logn)	$O(t*\log_t n)$
Поиск максимального ключа	O(logn)	$O(t*\log_t n)$

Также стоит сравнить эти два дерева в сценариях, где часто используются все основные операции, чтобы на практике установить разницу в работе этих структур данных. Сравнение проводилось с помощью файлов-тестов, содержащих набор соответствующих команд. Эти файлы подавались на вход программы в качестве аргументов, затем программа создавала файл, в который записывала вывод всех поданых команд (если таковой имеется) и в конце время работы в микросекундах. Результаты сравнения приведены в таблице 2.

Тесты	Splay Tree	B Tree
test5.dat и test6.dat (множество вставок)	529	170
test7.dat и test8.dat (множество поисков)	1541	659
test9.dat и test10.dat (множество удалений)	1579	745
test11.dat и test12.dat (множество поисков минимального ключа)	1060	564
test13.dat и test14.dat (множество поисков максимального ключа)	939	566

Вывод

В результате проделанной работы я пришел к выводу, что хоть теоретическая сложность по времени для обеих структур данных во всех случаях одинакова: ($log_2 n$), на практике же B-дерево является гораздо более быстродействующей структурой.