Московский государственный технический университет имени Н. Э. Баумана

Домашнее задание №2 по курсу «Архитектура Компьютеров»

Разработка эффективного представления бинарных деревьев

Студент:	Борнев Антон Юрьевич
Преподаватель:	Макаров Андрей Владимиро
Группа:	ИУ9-21

Москва

2014

Аннотация

Ставится задача разработать эффективные а) представление и б) реализации операций вставки, удаления и поиска для двоичных деревьев; параметры двухуровневой кэш-памяти должны задаваться опциями командной строки компилятора. Будем считать, что ключи и значения имеют тип *uint64_t*

1 Обозначения

C_0 — размер кэш строки
C_i — размер кэша *i-ого* уровня
B_0 — размер одной вершины бинарного дерева
$h(T)$ — высота бинарного дерева T
$B = C_0/B_0$ — сколько, в лучшем случае, ”можно получить” вершин дерева T за одно обращение к памяти.

2 Анализ бинарного дерева «на указателях»

Рассмотрим обычное бинарное дерево «на указателях». Пусть вершина реализована следующим образом:

Листинг 1: Пример кода

```
struct Node {
        key_t Key;
        item_t Item;
        Node  *left, *right, *parent;
}
```

На каждом шаге алгоритма поиска происходит следующие:

1. разыменование указателя на текущую вершину
2. сравнение искомого ключа с ключом в вершине
3. если мы нашли ключ, останавливаемся, иначе: если искомый ключ меньше ключа в вершине идём влево, иначе в право

Такое представление имеет большой минус: при каждом разыменовании указателя, мы получаем порцию данных размера B_0, в то время как в большинстве случаев необходим только ключ (Key) в текущей вершине. Обращение же к хранимому значению ($Item$) происходит крайне редко. В качестве решения, можно хранить ключи отдельно от значений. Так же лучше разбивать ключи на массивы по размеру строки кэша. Тогда за одно обращения мы будем получать B элементов.

***Лемма* 1.** *Будем считать, что за одно обращение, вся вершина полностью попадает в память. Рассмотрим T — бинарное дерево, его высота $h(T) \in [1; N]$, где N - количество вершин в дереве. По свойствам бинарного дерева сложность поиска, вставки и удаления $O(h)$, т.е. при данных операциях мы посещаем $O(h)$ вершин, следовательно в худшем случае будет происходить $O(h)$ транзакций памяти (кэш-промахов)*

3 Статические деревья

Пусть все вершины дерева располагаются в одном массиве, $h(u)$ — высота дерева с корнем в вершине *u*, тогда имеют место следующие определения:

***Определение* 3.1.** *BFS порядок (рис. 1) — это порядок, в котором вершины располагаются в той последовательности, в которой они были бы посещены, если бы использовался алгоритм поиска в ширину.*

***Определение* 3.2.** *DFS порядок (рис. 2) — это порядок, в котором вершины располагаются в той последовательности, в которой они были бы посещены, если бы использовался алгоритм поиска в глубину.*

***Определение* 3.3.** *Порядок van Emde Boas (рис. 3) определяется рекурсивно:*

1. *Если дерево T состоит из одной вершины — то нумеруем эту вершину.*
2. *Если дерево T состоит из двух и более: пусть $h_0 = \lceil h(T)/2 \rceil$ — высота дерева T_0, состоящего из всех вершин дерева T, глубина которых меньше, либо равна h_0, и пусть $T_1, ..., T_k$ — поддеревья T с корнями, расположенными на уровне $\lceil h(T)/2 \rceil + 1$ (Нумерация слева направо). Применим эти же рассуждения к $T_0, \ldots, T_k$.*

Порядки из определений 3.1, 3.2, 3.3 дают возможность избавиться от указателей и вычислять детей текущей вершины с помощью арифметических преобразований. Так, в *BFS* порядке для *i-ого* элемента детьми будут $2i$ и $2i + 1$. Аналогично, в *DFS* порядке для *i-ого* элемента: $i - 2^{h(u)-2}$ и $i + 2^{h(u)-2}$, где *u* — вершина с номером i. Для *vEB* алгоритм вычисления индексов более сложный. Двигаясь от корня вниз по дереву, мы отслеживаем *BFS* индекс текущей вершины и используем преобразование, описанные в [1], для получения индекса в *vEB*.

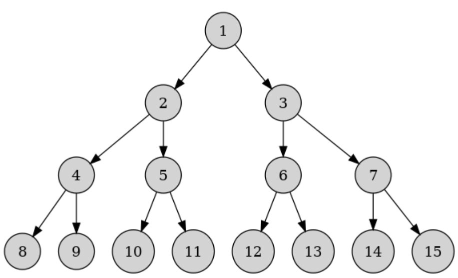

Рис. 1 — *BFS* дерево

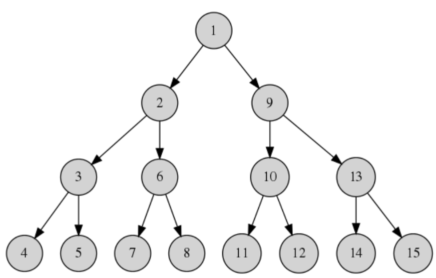

Рис. 2 — *DFS* дерево

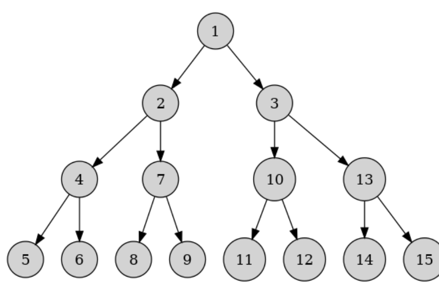

Рис. 3 — *vEB* дерево

Проанализируем количество транзакций памяти. При поиске по дереву, расположенному в *BFS* порядке, верхние $\lfloor log_2(B+1) \rfloor$ уровней будут содержать по две вершины, расположенные рядом в одном блоке размером C_0, но так как использоваться будет только один из них, получаем $O(log_2(N) - log_2(B))$ транзакций памяти. Для *DFS*, худший случай такой же как у *BFS*, лучший же $O(log_2(N)/B)$ (весь путь полностью лежит в блоках размером C_0). Для *vEB*, как показал Prokop [2], совершается $O(log_B(N))$ транзакций памяти.

Понятно, что любое динамическое дерево можно вписать в статическое. Более того, появляется возможность избавиться от указателей, а следовательно сократить размер одной вершины. Напомним, что размер константы *B* обратно пропорционален размеру одной вершины, следовательно уменьшив одну вершину, мы повышаем кэшируемость всего дерева. Используем данное наблюдение при построении кэш-эффективного алгоритма.

4 Кэш-эффективный алгоритм

Как уже было предложено в секции 2, будем хранить ключи отдельно от значений. Также разобьём дерево на кластеры по C вершин. Каждый такой кластер является статическим деревом. Пусть $h(C) = log_2(C+1)$ — высота такого кластера, и пусть $2^{h(C)+1} - 1$ вершин будет содержаться в каждом кластере, причём $2^{h(C)} - 1$ вершин — ключи, а $2^{h(C)}$ — вершины-указатели на другие кластеры. Так как размер ключей и указателей, в нашем случае, совпадает,то ключи и указатели будут храниться в одном массиве. Укладывать в массив будем в одном из трёх порядков, предложенных в секции 3. При навигации будут высчитываться индексы с помощью арифметических преобразований. При переходе на нижний уровень кластера, будет использоваться вершину-указатель для перехода в другой кластер. Рисунок 4 представляет собой иллюстрацию такого дерева для $C = 15$ и порядка *vEB*.

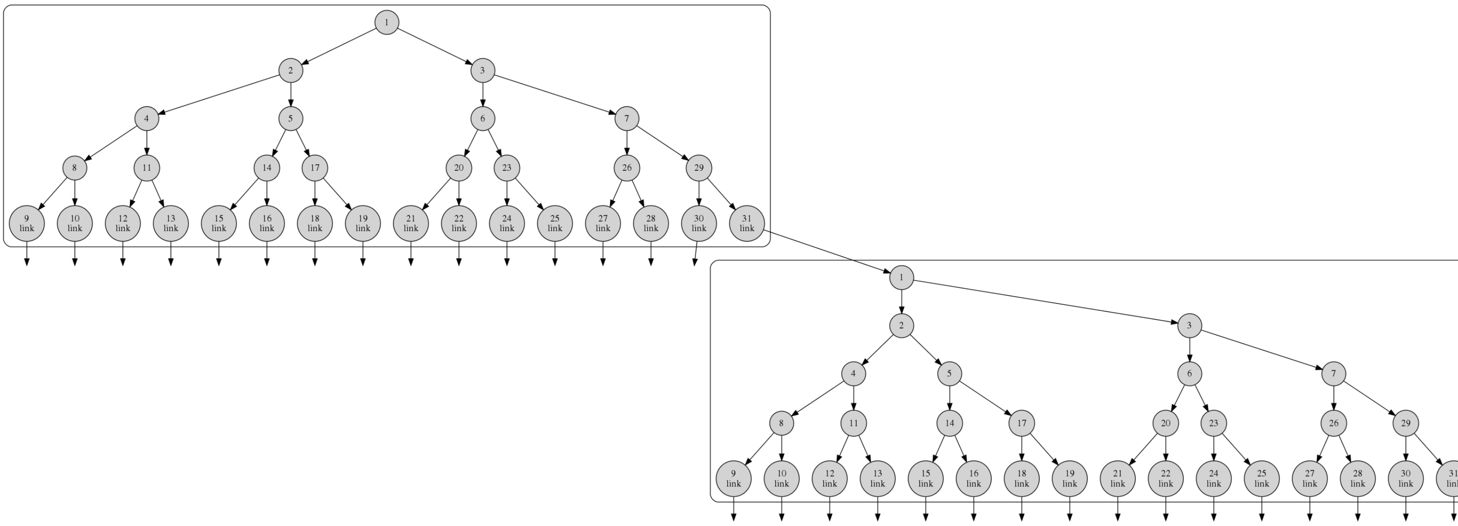

Рис. 4 — Кэш-эффективное расположение в памяти

***Определение* 4.1.** *Пусть R – дерево, разбитое на кластеры, тогда z(R) = h(R)/h(C) — высота в кластерах.*

***Определение* 4.2.** *Полностью плотный кластер — это кластер содержащий C ключей.*

***Лемма* 2.** *Поддержание кластера плотным даёт сложность на операцию вставки*
$O(h + C)$ *в худшем случае.*

Доказательство. Пусть R — кластер, в который выполняется операция вставки. V — поддерево дерева T, вписанное в данный кластер. Для поддержания R плотным, при вставке, если достигается высота $log_2(C+1)+1$, а в кластере имеются пустые места, то вместо того, чтобы создавать новый кластер, мы перестраиваем дерево V так, чтобы добиться идеальной сбалансированности. Для этого выполняется обход данного дерева V в предпорядке, формируя массив ключей и значений. Затем выполняется второй обход, во время которого строится сбалансированное дерево V. Сложность этих операций $O(n)$. В худшем случае, при вставках в V строится линейное дерево (вставки будут каждый раз увеличивать высоту дерева V). В таком случае, средняя сложность вставки в T:

$$O\left(\frac{n*L+B}{n}\right) \tag{1}$$

где n – количество вставок в V, L – сложность одной вставки в T, B – суммарная сложность балансировки. Дадим верхнюю оценку на B: пусть на каждую операцию вставки в R происходит перебалансировка дерева V. Будем считать, что каждый раз алгоритм применяется к C+1 вершине. Тогда: $B = nO(C+1)$. Упростим формулу 1:

$$O\left(\frac{n*L+B}{n}\right) \leq O\left(\frac{nh+n(C+1)}{n}\right) = O(h+C) \tag{2}$$

□

Лемма 3. *Данный алгоритм совершает $O(z(T)log_B(2C+1))$ транзакций памяти при поиске. (Если используется порядок van Emde Boas).*

Доказательство. Пусть T - бинарное дерево, разбитое на кластеры. Внутри одного кластера содержится *C* ключей и $C+1$ ссылка, расположенные в *vEB* порядке, следовательно при проходе по одному кластеру совершается $O(log_B(2C+1))$ транзакций памяти, отсюда при проходе по всему дереву потребуется $O(\frac{h(T)}{h(C)}log_B(2C+1)) = O(z(T)log_B(2C+1))$ □

Лемма 4. *Аналогично доказательству леммы 3, если используется порядок BFS/DFS, то количество транзакций памяти при поиске $O(z(T)(log_2(2C+1) - log_2(B))$.*

Лемма 5. *Пусть $2 \leq B < C$. Ожидаемое ускорение относительно обычного бинарного дерева (на указателях) составляет:*

$$S_{vEB} = O\left(\frac{log_2(C+1)}{log_B(2C+1)}\right) \tag{3}$$

$$S_{BFS/DFS} = O\left(\frac{log_2(C+1)}{log_2(2C+1) - log_2(B)}\right) \tag{4}$$

Доказательство. Для (3) имеем:

$$S_{vEB} = O\left(\frac{h(T)}{\frac{h(T)}{h(C)} log_B(2C+1)}\right) = O\left(\frac{log_2(C+1)}{log_B(2C+1)}\right)$$

Для (4):

$$S_{BFS/DFS} = O\left(\frac{h(T)}{\frac{h(T)}{h(C)}(log_2(2C+1) - log_2(B))}\right) =$$

$$= O\left(\frac{log_2(C+1)}{log_2(2C+1) - log_2(B)}\right)$$

□

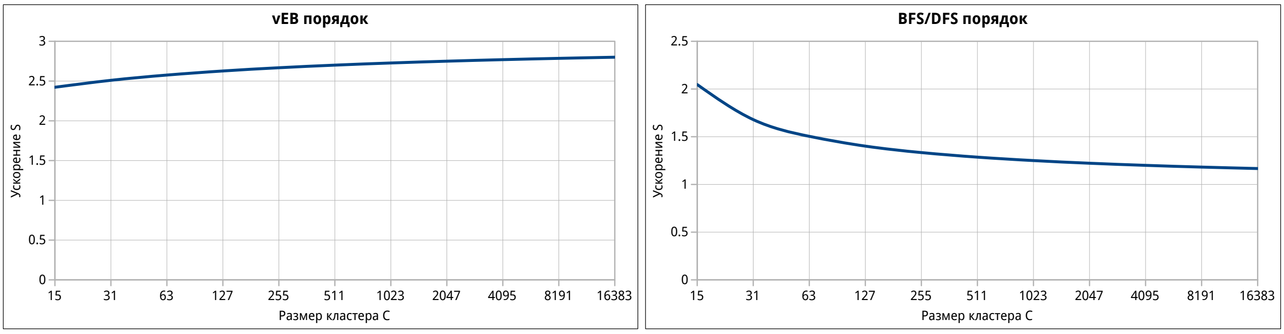

Рис. 5 — Графики зависимости ускорения от размера кластера, при $B = 8$

4.1 Эксперимент

Конфигурация системы:

Cpu: Intel Core i7-4700MQ CPU @ 2.400GHz

Ram: 12 GB, 1600 MHz

C_3: 6MB

C_2: 256KB

C_1: 64KB

C_0: 64B

Все тесты проводились на деревьях с количеством вершин 2^{24}, количество тестов для поиска 2^{27}. Данные, подаваемые на вход, были одинаковы для кэш-эффективного и обычного алгоритмов.

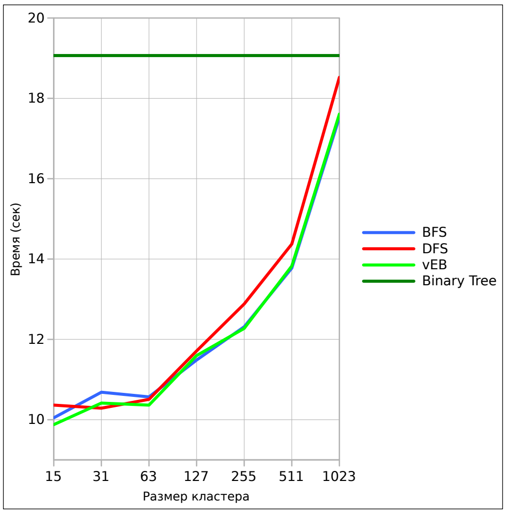

Рис. 6 — Зависимость скорости вставки от размера кластера

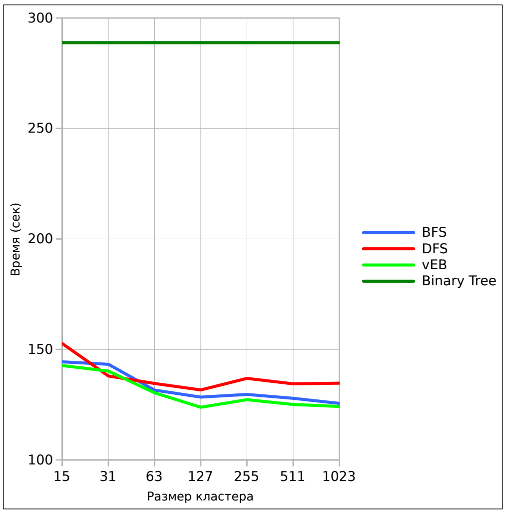

Рис. 7 — Зависимость скорости поиска от размера кластера

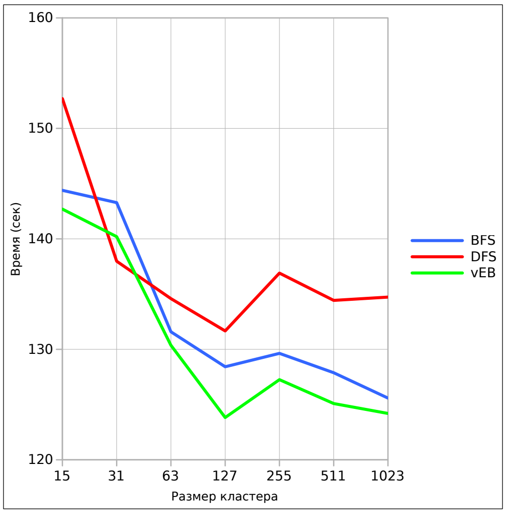

Рис. 8 — Зависимость скорости поиска от размера кластера (без сравнения с обычным деревом)

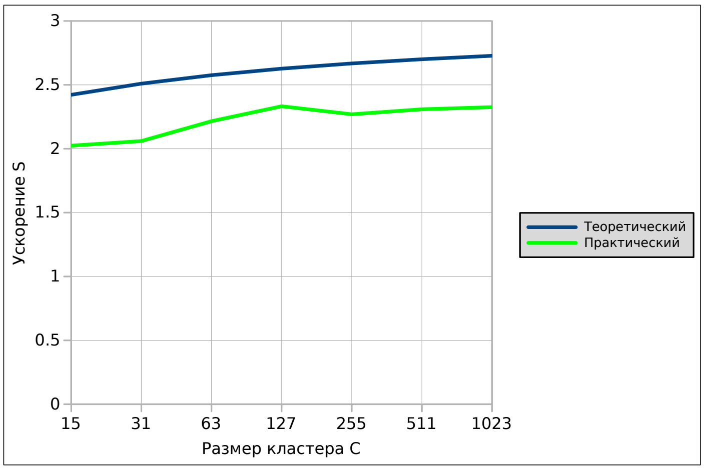

Рис. 9 — Выигрыш в скорости. Теоретические и практические результаты (vEB)

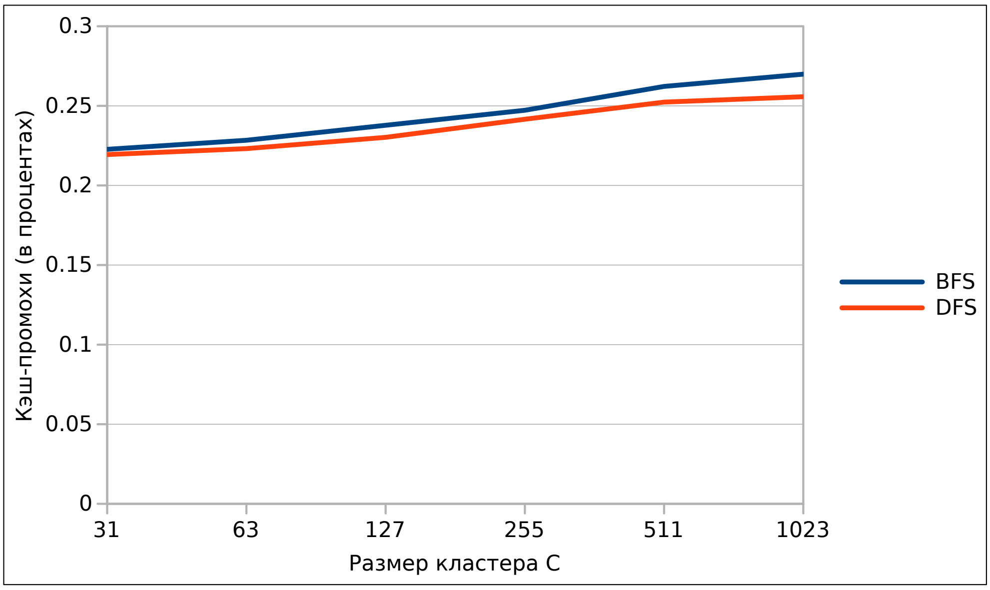

Рис. 10 — Замеры кеш-гриндом

Как видно из рисунка 8, порядок *vEB* является наилучшим. Рисунки 8 и 6 показывают, что лучший результат достигается разбиением на $C = 63$, или 127. Выбирать $C \geq 1023$ не рекомендуется, из-за большого расхода памяти, и ухудшения скорости вставок. Заметим, что достигнут выигрыш в скорости поиска более чем в 2 раза для всех тестируемых разбиений (рис. 7). Из рисунка 9, 8 видно, что выигрыш относительно бинарного дерева, соответствует формуле 3, с точностью до константы $c \approx 1.18$. Несоответствие результата *BFS/DFS* практического (рис. 8) и теоретического (рис. **??**) вызвано тем, что с увеличением параметра разбиение C, дерево становится более сбалансированным, а следовательно поиск выполняется быстрее. Поэтому дополнительно был поставлен ещё один эксперимент, с использованием cachegrind, с параметрами C_1 = 512B, C_0=64B, размером дерева 2^{16} вершин и 2^{17} тестов на поиск. Из рисунка 10 видно, что с увеличением C происходит деградация *BFS* и *DFS* по кэш-промахам, как и предполагалось в теории.

5 Заключение

В рамках данной работы был предложен простой алгоритм. Он имеет множество недостатков и был разработан с целью улучшения понимания работы кэша.

Список литературы

[1] G. S. Brodal, R. Fagerberg and R. Jacob. Cache oblivious search trees via binary trees of small height. In 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), pages 39–48. Society for Industrial and Applied Mathematics, 2002.

[2] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.