сишные трюки
(0xA выпуск)

крис касперски ака мыщъх, ака souriz, aka nezumi, aka elraton, no-email

сегодняшний выпуск трюков посвящен двоичным деревьям — этим простым, но в тоже время мощнейшим структурам данных, без не обходится практически ни одна серьезная программа. как раскрыть их потенциал, используя двоичные деревья с максимальной эффективностью, вот в чем вопрос!

организация хранения дв. деревьев на raw-уровне

Дерево состоит из узлов, каждый из которых в языках Си/Си++ традиционно определяется так:

```
struct my_tree
{
       struct my_tree *left_nest;     // ссылка на левый узел
       struct my_tree *right_nest;    // ссылка на правый узел
       int            leaf;           // элемент дерева
}
```

Листинг 1 каноническое определение двоичного дерева

Память под узлы выделяется либо функцией malloc (в языке Си), либо оператором new (в Си++). Это делают все или практически все, совершенно не задумываясь о тех накладных расходах, которыми их облагает менеджер кучи — выделение крошечных блоков памяти совершенно неэффективно! К тому же, зачастую соседние узлы оказываются в различных физических страницах памяти, в результате чего операции с деревом существенно замедляются, производительность падает в разы, а при хронической нехватке оперативной памяти винчестер трещит как бешенный!

К тому же, такое дерево существует _только_ в памяти и не может быть сохранено на диск в двоичном виде. То есть сохранить-то его можно, вот только при последующем считывании с диска ранее выделенные указатели будут указывать в "космос" со всеми вытекающими отсюда последствиями и перед сохранением дерева все указатели в обязательном порядке должны быть преобразованы в индексы, а при считывании дерева, его приходится реконструировать вновь, что совсем не добавляет производительности.

Выход — хранить дерево в массиве, используя индексы вместо указателей. В результате мы получим следующее:

```
struct my_good_tree
{
       unsigned int   left_nest;     // ссылка на левый узел
       unsigned int   right_nest;    // ссылка на правый узел
       int            leaf;          // элемент дерева
};
```

Листинг 2 структура двоичного дерева, подготовленная к хранению в массиве

Совокупность узлов, сложенных в массив (выделенный либо в статической памяти, либо в куче), обеспечивает: а) физическую близость соседних узлов, значительно ускоряющих операции с деревьям; б) делает операции копирования деревьев, передачу их по значению и сохранение/восстановления с диска тривиальной операцией.

Единственный минус данного решения заключается в том, что память под древесный массив необходимо выделять заранее и если этой памяти вдруг окажется недостаточно, придется реаллоцировать блок, что влечет за собой ощутимые накладные расходы. С другой стороны, операционные системы семейства Windows/UNIX позволяют не выделять, а лишь _резервировать_ память в адресном пространстве, поэтому размер массива можно (и нужно!) брать с большим запасом.

обход двоичного дерева

Операция обхода двоичного дерева (т. е. прохождения по каждому из его узлов) не самая тривиальная задача, решение которой выливается в десятки строк кода, которые еще отладить надо! Рекурсивные алгоритмы — тормозят и требуют очень много стековой памяти, не рекурсивные — намного более сложны в реализации.

Но стоп! Если дерево хранится в массиве, то его обход осуществляется простым перебором элементов массива один за другим и (за исключением операций объявления и инициализации) свободно укладывается всего в две (!) строки на Си и работает с ошеломляющей скоростью, особенно на процессорах использующих предвыборку, к тому же легко масштабируется, что на HT- и многоядерных проецссорах совсем немаловажно:

```
// объявление
int a; struct my_good_tree tree_array[MAX_TREE_SIZE];

// инициализация
memset(tree_array, 0xDEADBEEF, sizeof(tree_array));

for(a = 0;a < MAX_TREE_SIZE; a++)
        if (tree_array[a].leaf!=0xDEADBEEF)printf("x\n",tree_array[a].leaf);else break;
```

Листинг 3 алгоритм обхода двоичного дерева, хранящегося в массиве

удаление узлов двоичного дерева

В отличии от списков, в которых операции вставки/удаления новых элементов осуществляются элементарно, деревья легко добавляют лишь новые элементы, а вот удаление старых зачастую требует реконструкции оставшейся части дерева, что, во-первых, непроизводительно, а, во-вторых, это же сколько программировать и отлаживать надо! Самое обидное, что один и те же элементы в ходе "разрастания" дерева могут добавляться/удаляться многократно!

А что если… не удалять элементы, а только помечать их удаленными?! Для этого в структуру дерева будет достаточно добавить всего лишь одно поле: "deleted". Что это дает?! Во-первых, перестраивать структуру дерева при удалении более не придется. Во-вторых, при операциях поиска существует вероятность "натолкнуться" на удаленный элемент раньше, чем достичь конца дерева, следовательно, средняя скорость поиска несколько возрастает. В-третьих, повторное добавление ранее удаленного элемента решается простым сбросом флага "deleted".

Естественно, при накоплении большого количества удаленных элементов, эффективность использования двоичного дерева будет неуклонно уменьшаться, но эту проблему легко решить перестройкой дерева, т.е. реальным удалением элементов, помеченных как удаленные! Кстати, совсем неплохая идея — завести счетчик удалений/добавлений каждого из элементов, и при перестройке дерева удалять только "непопулярные" элементы.

балансировка или скремблирование?

"Простые" двоичные деревья, часто использующиеся для быстрого поиска данных, хорошо работают в том _и_ _только_ в _том_ случае, если "кушают" поток случайных данных. Если же им "скормить" возрастающую или убывающую последовательность чисел, то время поиска в двоичном дереве будет даже больше времени поиска в списке/линейном массиве, поскольку дерево требует для своей организации значительно больших "телодвижений".

Выход? Любой преподаватель скажет вам — использовать сбалансированные деревья, которые вы наверняка проходили в университете. Готовых библиотек куча… вот только, реализовать сбалансированное дерево намного сложнее нормального, да и всех проблем оно не решает. На самом же деле…

Эффективное использование обычных деревьев достигается в случае _гарантированного_ поступления на их вход случайных данных, чего легко добиться скремблированием, т.е. наложением на входной поток псевдослучайной последовательности данных, сгенерированной, например, функций rand(). Естественно, при извлечении элементов из дерева, операцию скремблирования необходимо развернуть на 180%. И хотя существует возможность, что даже после скремблирования поступающие на вход дерева данных сохранят некоторую упорядоченность, отказываться от этой идеи, не обкурив ее не стоит!!!

случайные пермутации дерева

Если дерево хранится в виде массива и если мы видим, что оно приобретает несбалансированную структуру, перекашиваясь либо в одну, либо в другую сторону, мы можем просто… да-да!!! просто переставить элементы дерева в случайном порядке, т.е. задача балансировки дерева сводится к алгоритму "тасовки колоды карт", которых придумано очень много. Конечно, операции переупорядочивания снижают производительность, проигрывая сбалансированным деревьям, но… сбалансированные деревья оправдывают себя только при обработке очень больших объемов информации, в противном случае, обычное двоичное дерево, хранимое в массиве и "тасуемое" время от времени, вырывается вперед!!!

гибрид дерева и социалистической очереди

Другим способов избежания дисбаланса двоичного дерева, является организация входной очереди по типу "цепочки задержки". Рассмотрим это на следующем примере. Допустим, к нашему дереву последовательно добавляются числа 1, 2, 3, 4, 5, 6… Любой, кто хоть однажды имел дело с деревьями, сразу же скажет, поскольку $1 < 2 < 3 < 5 < 6$, то все эти числа попадут на _одну_ ветвь, а другая ветвь окажется совсем пустой.

А теперь представим, что на подступах к дереву стоит "демон" и складывает поступающие числа в некоторый контейнер, а затем извлекает их оттуда и переупорядочивает в наиболее выгодном для дерева порядке. Т.е. в данном случае это — 3, 4, 2, 5, 1, 6, т.е. для всей последовательности должно выполняться условие $Xn < Xn+1 > Xn+2 < Xn+3$… Это легко обеспечить сортировкой элементов с их последующей выборкой. Поскольку, необязательно организовывать длинную очередь, то временем сортировки можно пренебречь. Естественно, при операции поиска элементов в дереве, нельзя забывать об очереди ;-)