Лекция по алгоритмам #3
Тема: преобразование операций
16 сентября

Собрано 28 сентября 2014 г. в 20:01

Содержание

1 Доказательства нижней границы времени работы сортировок произвольных объектов.

Докажем, что любая сортировка произвольных объектов сделает не меньше $n \log n$ сравнений. Тогда асимптотика сортировки будет $\Omega(n \log n)$.
Пусть мы сделали $k \leq n \log n$ сравнений. Рассмотрим строчку 01010011101011100..., где 1 - больше, 0 - меньше, для каждого из k сравнений. Мы можем получить не более 2^k различных строк длины k, т.к. результат каждого сравнения увеличивает количество вариантов не более чем в 2 раза. Таким образом $2^k < n!$, а значит найдется два различных входа, для которых соответствующие им строчки совпадут. Выходит, что k должен быть не меньше $n \log n$, а сортировка имеет асимптотику $\Omega(n \log n)$.

2 Структуры данных

Мы знаем: стек, очередь, хеш-таблицу, сортировку, бинарный поиск, списки, вектора с удвоением.

Напишем небольшой язык программирования.
Нам нужно уметь:

1. Считать от переменной значение val[variable].
2. Разбирать выражения. $(1 + 3 * 4 - (3 + 4) * 2)$
3. if, for...
4. functions... Хотя кому это надо?

2.1 Переменные и значения

Воспользуемся хеш-таблицей. Единственное, чему нам надо научиться - преобразовывать название переменной(строчку) в число. Заведем хеш-функцию: String -> (hash). $Hash(string) = 0 \dots 2^{64} = \sum_i s_i * 179^i$

2.2 Разбор выражения стеком за O(n)

Пусть есть несколько операций $(+, -, *, /)$ числа и переменные (последовательности цифр, букв и подчеркиваний). Также должен быть задан приоритет на операции. В случае равного приоритета предполагается, что выражение левоассоциативно. Разобьем исходную строчку на лексемы:
$\cdots + \cdots * \cdots$
Начнем вычислять. Воспользуемся двумя стеками: значения и операции. Лексемы добавляем в значения, операции - в операции. При очередной операции смотрим на последнюю операцию в стеке. Если у нее приоритет больше, сразу выполним. Т.е. нужно достать

из стека операций операцию, из стека значений 2 переменные, посчитать, и добавить результат в стек значений. Текущую операцию в любом случае добавляем в стек. Когда выражение закончится, в стеке все еще могут быть какие-то операции и лексемы. Выполним их в обратном порядке.

Усовершенствуем алгоритм для выражения со скобками. Пусть мы дошли до открывающейся скобки. Добавим ее. Теперь, когда мы считаем операции, делаем это только до скобки. Когда встретим закрывающую скобку, выполняем все до первый открывающийся скобки, и удаляем ее. Для удобства можно добавить открывающую скобку в начало и закрывающую в конец всего выражения. Заметим, что если после завершения работы алгоритма что-то осталось в стеке, то исходное выражение не корректно. Таким образом, выражение невозможно посчитать тогда, когда:

1. Идут несколько операций подряд (++). Унарный минус обрабатывается отдельно.
2. Несколько лексем идут подряд.
3. Для какой-то закрывающейся скобки не нашлось открывающейся.
4. В конце стек операций не пуст.

Как считывать лексему?

1. пробел: пропускаем
2. цифра: пока цифра - лексема
3. операторы: пропускаем.
4. Унарный минус приходит только тогда, когда наш алгоритм ожидает оператор. После закрывающей скобки наш алгоритм ожидает оператор.

2.3 if, for...

Мы то знаем, что все конструкции подобного вида, это на самом деле нагромождение всевозможных выражений. Так что по большому случаю нам нужно:

1. Проверить скобочную последовательности на корректность. ПСП - это то, что можно получить удаляя значения из выражения. Если скобки одного типа, то достаточно считать баланс. Если несколько видов, то эта задача решается стеком, в который будем добавлять открывающиеся скобки и пытаться удалить последнюю, по приходу закрывающей.
2. Для каждой скобки найти позицию парной. Для этого храним в стеке и скобку и позицию. Теперь, когда мы удаляем очередную скобку из стека, мы знаем позицию и открывающей и закрывающей скобки.

3. Осталось обрабатывать работу с переменными. Старое решение с хеш-таблицей тоже подходит, но проще взять все запросы (на присвоение), отсортировать все имена переменных и хранить вместе с ними их значения. Тогда несколько подряд идущих имен изменяют значение у одной переменной. Единственный тонкий момент - это фиксированный порядок выполнения операций. Для этого, при присвоении текущей переменной - другую, придется найти ее бинпоиском и обновить до актуального состояния. Преимущество данного алгоритма в том, что он детерминирован, и мы не используем дополнительную память.

3 Теория. Преобразование операций.

.

Merge(A, B) = (A объединить B)
Add(A, x) = (A объединить с x)

3.1 Merge → Add

Дано: Мультимножество. Умеем сливать две структуры. Хотим научиться добавлять элемент.

Создаем множество, которое состоит из одного элемента, который хотим добавить. Сливаем два множества. Время работы Add = Merge

3.2 Add → Merge

Дано: Мультимножество. Умеем добавлять элемент. Хотим научиться объединять два мультимножества.

```
Merge(A, B) считаем, что в A элементов меньше, чем в B. Size(A) <= Size(B)
    for x in A
        Add(B, x)
    return B
```

Суммарное время работы: $\sum Merge \leq O(N \log N) Add$, где N общее количество элементов.

Рассмотрим элемент x. Каждый раз, когда к нему применяют Add, он оказывает в множестве хотя бы в два раза большем. Значит, каждый элемент будет рассмотрен не более $\log N$ раз.

3.3 Find → Delete

Дано: Множество. Умеем находить элемент. Хотим научиться удалять.

Для каждого элемента множества будем хранить флаг, удален элемент или нет. <val, isDeleted>

```
Delete(x):
    Find(x).isDeleted = 1
```

Время работы: Delete = Find

3.4 build, get → Merge → Add

Дано: Множество. Необходимо отвечать на запросы, которые не меняют элементы. Умеем из элементов создать структуру. Хотим научиться сливать два множества и добавлять элемент.

Пусть наше множество состоит из N элементов. Тогда представим наше множество в виде $\log N$ множеств. $N = \sum_{i=1}^{\log N} 2^{k_i}$

(N = 7) → (1) + (2) + (4)

Теперь, что бы ответить на запрос в таком представление, нужно ответить на запрос в каждом кусочке.

```
NewGet()
    for i = 1... log N
        get(A_i)
```

Время работы NewGet = $\log N$ get

```
Merge()
    while существует size(A_i) == size(A_j)
        Build(A_i, A_j)
```

Время работы Merge $\leq Build(N \log N)$, так как каждый элемент будет участвовать в построение максимум $\log N$ раз, так как он каждый раз переезжает в более большое множество.

3.5 Итог

Мы умеем делать:

1. Add → Merge

2. Merge $\rightarrow$ Add
3. Find $\rightarrow$ Delete
4. build, get $\rightarrow$ Add

Замечание: Find $\rightarrow$ Delete используется в хэш таблице с открытой адресацией.

4 Хеш-таблица с открытой адресацией

Каждому элементу сопоставляем ячейку в массиве. Если на этом месте уже что-то написано, то ищем ближайшую свободную справа. Элементы не удаляем явно, а ставим пометку, что он удален.

$2n \leq M$, где M - размер хэштабдицы, а n - количество элементов в структуре.

```
Index(x)
    i = x%M
    while(ht[i] != x and !empty[i])
        i = (i + 1)%M
    return i
```

Когда элементов в таблице, в том числе удаленных, стало больше, чем свободных ячеек, перестроим таблицу.