Лекция по алгоритмам #4
Тема: Кучи
07 октября 2014

Собрано 21 октября 2014 г. в 00:27

Содержание

1 Inplace Merge

Input: два отсортированных массива a и b, идущих подряд в памяти (сначала a).
Output: отсортированный массив из элементов массивов a и b, лежащий на их месте.
$n = |a| + |b|$

1.1 Merge0. $T = \mathcal{O}(n \cdot \log(n))$ $M = \mathcal{O}(\log(n))$

Пусть x – медиана большего массива.
Разделим каждый из массивов a и b на 3 части:
1. a_1, b_1 – части, в которых все элементы меньше x.
2. a_2, b_2 – равны x.
3. a_3, b_3 – больше x.
Далее, вызовем `merge()` от пар (a_1, b_1) и (a_3, b_3).
Новый массив получится таким: `merge(`a_1`,` b_1`)` a_2 b_2 `merge(`a_3`,` b_3`)`
При этом важно, чтобы a_2 было раньше b_2, для стабильности сортировки. Кстати, заметим, что такая сортировка лучше, чем `qsort`, так как она является стабильной и детерминированной, в отличии от быстрой.
Заметим, что дополнительной памяти требуется $\mathcal{O}(\log(n))$ из-за стека рекурсии и работает эта штука за $\mathcal{O}(n \cdot \log(n))$, т.к. на каждом уровне рекурсии надо упорядочить порядка n элементов.

1.2 Merge1. $T = \Theta(n)$ $M = \Theta(|a| + |b|)$

Создаем массив c размером $|a| + |b|$. Идем двумя указателями i и j по массивам a и b соответственно и кладем меньший элемент из $a[i]$ и $b[j]$ в массив c, увеличивая соответствующий указатель.

1.3 Merge2. $T = \mathcal{O}(n)$ $M = \mathcal{O}(|a|)$

Создаем массив c длины $|a|$ и копируем в него содержимое массива a. Так же, как в Merge0, установим два указателя – на начало c и b. Ещё один установим в начало массива a, но этот указатель, в отличии от двух предыдущих, будет не для считывания, а для записи. На очередной итерации сравниваем элемент массива c, в котором находится указатель, с аналогичным элементом массива b. Меньший из них записываем в массив a, соответствующий указатель считывания увеличиваем, увеличиваем указатель записи.
Заметим, что мы никогда не обгоним указателем записи в a указатель считывания в b, а значит не испортим значений массива b.

1.4 Merge3. $T = \mathcal{O}(n)$ $M = \mathcal{O}(min(|a|, |b|))$

Если массив a короче, чем b, то делаем все так же. Иначе, нам придется поменять их местами. Это можно сделать так:
`1. reverse(a)`
`2. reverse(b)`
`3. reverse(a+b)`
Несложно заметить, что после таких действий массивы a и b действительно поменяются местами. Дальше решаем задачу так же, как в прошлом пункте.

1.5 Merge5. $T = \mathcal{O}(n)$ $M = \mathcal{O}(1)$

В общем, такой есть, но он очень сложный. В 1963м году была придумана нестабильная версия. В 96м стабильная.

1.6 Экзотика

1. $M = \mathcal{O}(1) \quad T = \mathcal{O}(\frac{n \cdot \log n}{\log \log n})$
Прочитать можно тут.
2. $M = \mathcal{O}(1) \quad T = \Theta(m \cdot \log(\frac{n}{m}))$, где n - длина массива a, а m - массива b.
Прочитать можно тут.
Эти и некоторые другие ссылки можно найти тут.

2 MinMax heap

Памяти $\mathcal{O}(n)$.
`GetMin, GetMax` за $\mathcal{O}(1)$.
`Add, DelMin, DelMax` за $\mathcal{O}(\log n)$.
Модификация двоичной кучи, где каждая вершина удовлетворяет следующему свойству: если она находится на четном уровне, то она меньше всех своих потомков, если на нечетном, то больше. Тогда понятно, что её минимум находится в корне, а максимум – один из детей корня. `Add, DelMin, GetMin` такие же, как и в двоичной куче. Опишем реализацию `SiftUp`, `SiftDown`, `DelMax`.

2.1 SiftUp

Пусть надо просеить вверх вершину со значением и она находится на четном слое (четность слоя можно узнать, просто поднявшись до корня или предподсчитав заранее). Тогда, если X уже не является корнем, над ней есть элемент Max. Далее, возможны два случая:

1. $X > Max$. Но тогда заметим, что цепь минимумов до корня точно не изменилась, т.к. все они должны быть меньше Max, а значит и меньше X. Меняем местами X и Max, а дальше, сравнивая X и следующий максимум (вершину, на 2 уровня выше), просеиваем вверх. После очередного свопа свойство этой кучи для поддерева нового положения X не испортятся – очередной свопнутый максимум (Max_i) продолжит быть максимумом на поддереве (т.к. он не перестал быть больше своих потомков), а минимум, лежащий между новыми положениями X и Max_i не перестанет быть минимумом в своем поддереве, т.к. Max_i больше его.
2. $X \leq Max$. Тогда не испортилась цепочка максимумов. Делаем все то же самое, но сравнивая и свопая с ближайшим минимумом.
Если X был на четном уровне, то действия аналогичны, но поменяются знаки сравнений и слова "минимум"и "максимум".

2.2 SiftDown

Сравниваем данную вершину с внуками (если их нет, то с сыновьями). Если текущая вершина на четном уровне и больше какого-то из внуков, то меняем ее местами с минимальным из внуков. Теперь, сравниваем её значение со значением её нового отца. Если она больше, то меняем местами с отцом и запускаем `SiftDown` от этого места.

2.3 DelMax

Выберем максимум из детей корня. Он - ответ. Поставим на его место последний элемент кучи и просеим его вниз (этот элемент не может быть меньше корня, т.к. он был в его поддереве).

3 HeapBuild: lower bounds

Рассмотрим нижние оценки на количество сравнений для построения двоичной кучи из неупорядоченного массива.

3.1 $T(HeapBuild) \geq n - 1$

Корнем двоичной кучи должен быть минимальный элемент, поэтому ее нельзя построить за меньшее число сравнений, чем необходимое для поиска минимума в массиве.
Докажем, что нельзя найти минимум в произвольном массиве меньше, чем за $n-1$ сравнение.

Рассмотрим граф сравнений (числа - вершины, сравнение двух чисел - ребро между соответствующими им вершинами).
Предположим, что мы нашли минимум меньше, чем за $n-1$ сравнение.
Тогда граф сравнений не связный, т.е. есть хотя бы 2 набора чисел, между элементами которых не было ни одного сравнения. Но, тогда алгоритм сработал бы неверно на одном из следующих массивов:
1. В первой компоненте связности числа от 1 до A, во второй - от $A+1$ до $A+B$, в остальных - что угодно.
2. В первой компоненте связности числа от $B+1$ до $B+A$, во второй - от 1 до B, в остальных - что угодно.
Значит, такого алгоритма не существует и минимальное число сравнений - хотя бы $n-1$.

3.2 $T(HeapBuild) \geq 1.3644n$

Для доказательства этой оценки применим такое же рассуждение, как и в доказательстве минимального числа сравнений для сортировки массива.
Пусть число различных двоичных куч из n различных элементов равно $H(n)$.
Тогда, если алгоритм строит корректную кучу, должно выполняться неравенство $2^k \geq \frac{n!}{H(n)}$, где k - число сравнений, сделанных алгоритмом,
иначе, из какой-нибудь из $n!$ перестановок была бы построена некорректная куча.
Значит $k \geq \log_2 \frac{n!}{H(n)}$.
Но $H(n) = \frac{n!}{\prod size_i}$, где $\prod size_i$ - произведение размеров всех поддеревьев кучи,
т.к. куч из n элементов с минимумом в корне $\frac{n!}{n}$, а в корне каждого поддерева кучи должен быть минимум.
Значит $k \geq \log_2 \frac{n!}{H(n)} = \log_2 \prod size_i \geq 1.3644n$. Последнюю часть неравенства доказывать не будем.

4 Pairing heap

4.1 Оценки времени

`Add` $\mathcal{O}(1)$
`Merge` $\mathcal{O}(1)$
`GetMin` $\mathcal{O}(1)$
`DelMin` $\mathcal{O}(\log n)$
`DecreaseKey` $o(\log n) \quad (\Omega(\log \log n))$
Все оценки амортизированные.

4.2 Описание

Вся куча – это список из нескольких куч (список корней). Каждая из этих куч будет выглядеть следующим образом: каждый уровень – это список вершин (зацикленный), каждая из которых хранит значение, ссылается на следующий в этом слое (для данной кучи) элемент и на своего левого ребенка (при этом для нее выполнены все свойства кучи). Назовем эти сслыки `r` и `l` соответственно.

4.3 Add

Добавляет в список новую кучу, содержащую единственный элемент – данное значение.

4.4 Merge

Просто прибавляем один список к другому.

4.5 Merge(a, b)

(Это `merge` двух кучек (тех, что лежат в списке) одной кучи) Если значение в `a` меньше, чем значение в `b`, то `b` теперь будет находится во втором слое кучи с корнем в `a`, так что теперь добавляем ее в список детей `a` следующем образом:
`b` будет иметь ссылку `b.r` на текущего самого левого ребенка `a` и сам станет самым левым. Т.е. нужно обновить две ссылки:
`b.r` = `a.l`
`a.l` = `b`
Если значение в `a` меньше, чем значение в `b`, то действия аналогичны, но уже `a` станет ребенком `b`

4.6 Pairing

Это собственно операция объединения всего этого списка корней, из которых состоит наша в целом, в одну большую кучу. Это делать мы будем так: объединяем две первые, а потом объединим получившуюся кучу с результатом работы `Pairing` от оставшегося списка кучек.

```
1.  Pairing(l)
2.      if (|l| == 1)
3.          return l[0]
4.      if (|l| == 2)
5.          return Merge(l[0], l[1])
6.      return Merge(Merge(l[0], l[1]), Pairing(l[2:]))
```

4.7 GetMin

Просто всегда помним минимум в списке.

4.8 DelMin

Зная минимум во всем списке, удаляем его, делаем `Pairing` от списка его детей, добавляем получившуюся кучку к списку кучек (списку корней).
`DelMin`
`1. Del` $\mathcal{O}(1)$
`2. Pairing` $\mathcal{O}(k)$ `(где` k `- количество детей у удаленной вершины)`

4.9 DecreaseKey(x)

Удаляем x из списка вершин уровня, в котором он содержался, перенаправив ссылку с родителя (если таковая указывала в x) на `x.r`. x становится новым корнем в списке корней.