Algorithms

Виталий Сергеевич Ерошин

September 2021

Содержание

1 Heap (Куча)

1.1 Методы

S - множество целых чисел. Нужно отвечать на запросы:

1. $insert(x)$ - добавить x в S
2. $getMin()$ - найти $min_{y \in S} y$
3. $extractMin()$ - извлечь, удалить $min_{y \in S} y$ из S
4. $decreaseKey(ptr,\ \Delta >= 0)$ - уменьшить число, лежаещее по указателю ptr, на Δ

1.2 Примеры использования

1. Обработка запросов
2. Алгоритмы Дейкстры, Прима, декартово дерево, Heap Sort (сортировка массива с помощью кучи)

1.3 Бинарная (двоичная) куча

$$S = \{a_1, a_2, \ldots, a_n\}$$

Представим кучу как дерево. Пусть для удобства a_v имеет сыновей a_{2v} и a_{2v+1}. Это позволяет нам хранить дерево неявно - в виде массива. Так же можно легко находить родительскую вершину $v = [\frac{u}{2}]$, где u - текущая вершина.

1.3.1 Требование кучи

$\forall v$ число, записанное в вершине v, должна не превосходить ($\leq$) все числа в поддереве v.
Утв. Требование кучи выполняется, если

$$\forall v \begin{cases} a[v] \leq a[2v],\ 2v \leq n \\ a[v] \leq a[2v+1],\ 2v+1 \leq n \end{cases}$$

Будем говорить, что массив $a_1, a_2 \ldots a_n$ задает корректную кучу, если для него выполняется требование кучи.

$$getMin() \quad a[1] \quad O(1)$$

1.3.2 Вспомогательные процедуры

```
siftUp(v) {
  while(v != 1) {
    if (a[v] < a[v/2]) {
       std::swap(a[v], a[v/2]);
        v /= 2;
    }
    else break;
  }
}


siftDown(v) {
  while (2v <= n) {
    int u = 2v;
    if (2v + 1 <= n && a[2v + 1] < a[2v]) {
      u = 2v + 1;
    }
    if (a[u] < a[v]) {
      std::swap(a[u], a[v]);
      v = u;
    }
    else break;
  }
}
```

Время работы $siftUp$ и $siftDown$ - $O(log\ n)$ (Глубина дерева не больше, чем $log_2\ n$)

Лемма Пусть $a_1, a_2, \ldots a_n$ - корректная куча. Пусть пришел запрос $a_v := x$. Тогда после $siftUp(v)$, если a_v уменьшилось, или $siftDown(v)$, если a_v увеличилось, куча станет корректной.

Доказательство
$siftUp()$, то есть уменьшение a_v. Индукция по v.

1. База: $v = 1$ - очевидно.
2. Переход: $v \neq 1$:

   Если $a[v/2] \leq x$, то все хорошо.

   Иначе $a[v/2] > x$. Заменим a_v на a_p. Тогда получим корректную кучу, в которой нет x, но есть две копии a_p. Затем верхнее a_p заменим на $x < a_p$. По предположению индукции в конце будет корректная куча

$siftDown()$: индукция от листьев к корню (индукция по $n - v$)

1. База: v - лист.
2. Переход: v - не лист.

   Если $a_u \leq x$, то куча уже корректна, а $shiftDown(v)$ ничего не делает.

   Иначе $a_n < x$. Заменим x на a_u. Получим корректную кучу. Увеличим нижнее вхождение a_u на x по предположению индукции куча корректна.

1.3.3 Процедуры

```
int getMin() {
  return a[1];
}

void decreaseKey(int v, int delta >= 0) {
  a[v] -= delta;
  siftUp(v);
}

void insert(int x) {
  a[++n] = x;
  siftUp(n);
}

void extractMin() {
  a[1] = a[n--];
  siftDown(1);
}
```

getMin()	$O(1)$
insert()	$O(log\ n)$
decreaseKey()	$O(log\ n)$
extractMin()	$O(log\ n)$

1.4 Heap Sort - сортировка кучей

$$a_1, a_2, a_3, \ldots a_n$$

1.4.1 Алгоритм (простая версия)

```
for i = 1...n insert(ai)
for i = 1...n print(getMin()); extractMin();

// Time: O(nlogn)
```

На i-м шаге напечатается i-я порядковая статистика.

1.4.2 Алгоритм (сложная версия) - In-place

```
heapify(); // процедура, строящая кучу по a_1...a_n
           // max в корне

for i = 1...n
  extractMax();

 // a_(n) встанет на место n
 // Time: O(nlogn)
```

На i-м шаге i-я порядковая статистика положится на i-е с конца место.

Процедура $heapify()$: куча с min в корне.

```
a_1, a-2...a_n // наш массив

for (int i = n; i >= 1; --i) {
  siftDown(i);
}
```

Утверждение Время работы $heapify()$ есть $O(n)$, при этом $heapify$ строит корректную кучу.

Доказательство на k-м уровне $shiftDown()$ работает $\leq H-k+1,\ H \leq log_2 n$. Суммарное время работы меньше, чем

$$1(H+1)+2H+4(H-1)+8(H-2)+\ldots+2^{H-1}+1=\sum_{k=0}^{H} 2^k(H-k+1)$$

$$H-k+1=m \quad k=H-m+1$$

$$\sum_{m=1}^{H+1} m2^{H-m+1}=\Theta(n)\sum_{m=1}^{H+1}\frac{m}{2^m}$$

Достаточно доказать, что $\sum_{m=1}^{\infty}\frac{m}{2^m}$ сходится. $\frac{m}{2^m}\leq\frac{5^m}{2^m}$ для всех m, начиная с некоторого (с m_0).

$$\sum_{m=1}^{\infty}\frac{m}{2^m}\leq\sum_{m=1}^{m_0}\frac{m}{2^m}+\sum_{m=m_0}^{\infty}\frac{m}{2^m}$$

Корректность? Докажем индукцией по i в порядке убывания, что после выполнения $siftDown(i)$ в поддереве вершины i будет корректная куча. В конце (после "замены"$-\infty$ на a_i и вызова $siftDown(i)$) получим корректную кучу.

Следствие Не существует такой реализации кучи, основанной на сравнениях, которая могла бы делать *extractMin* за $O(1)$ и при этом могла бы делать *insert* за $O(1)$. Иначе был бы алгоритм сортировки за $O(N)$.

1.4.3 decreaseKey по идентификатору

(номер запроса на котором соответствующее число было добавлено)

$pointer[t]$ - указатель на вершину в куче, которая соответствует t-му добавленному элементу.

$num[v]$ - идентификатор, соответствующий вершине v.

```
void exchange(int u, int v) {
  int k = num[u];
  int m = num[v];
  std::swap(num[u], num[v]);
  std::swap(pointer[k], pointer[n]);
  std::swap(a[u], a[v]);
}

void siftUp(int v) {
  while (v != 1) {
    if (a[v] < a[v / 2]) {
```

```
        exchange(v, v / 2);
        v /= 2;
      } else {
        break;
      }
    }
  }

  void decreaseKey(int t, int delta) {
    a[pointer[t]] -= delta;
    siftUp(t);
  }
```

1.4.4 erase

1. Удаление по указателю (или по идентификатору)

   ```
   a_v = -inf;
   siftUp(v);
   extractMin();
   ```

2. Удаление по значению

   ```
   getMin() {
     while (A.getMin() == D.getMin()) {
       A.extractMin();
       D.extractMin();
     }
     return A.getMin();
   }
   ```

Это все работает при корректности запросов (несуществующие элементы не должны удаляться)

1.5 Другие кучи

1.5.1 Куча Фибоначчи

$decreaseKey$ за $O(1)$ амортизированно. Все остальное за $O(logn)$ амортизированно.

1.5.2 Биномиальная куча

1. $insert$
2. $getMin$
3. $extractMin$
4. $decreaseKey$
5. $merge$

Дает построить кучу Фибоначчи.

Биномиальное дерево порядка k (T_k). Если есть дерево T_k, можно построить еще одно дерево T_k и подвесить его к корню первого. Таким образом получим T_{k+1}. На m-м уровне дерева T_k находятся C_m^k вершин. В вершинах дерева храним элементы мультимножества. Требование кучи: число, записанное в вершине v не превосходит чисел, записанных в поддереве v.

Биномиальная куча - набор биномиальных деревьев попарно различных порядков. В дереве T_k содержится 2^k вершин.

```
getMin: найти минимальный корень среди всех деревьвев // O(logn)
```

Если всего в куче n элементов, то все деревья в куче имеют порядок $\leq \lfloor log_2 n \rfloor$
Всего деревьев в куче $\leq log_2 n$

```
decreaseKey(ptr, delta) // O(logn)
```

$H_1 :\ t_1[0] \ldots t_1[logn]$
$H_2 :\ t_2[0] \ldots t_2[logn]$

```
merge {
  carry = -1; // t[0]...t[logn + 1] - результат
  for (int i = 0...logn + 1) {
    t_1[i], t_2[i], carry
    if (все 3 дерева есть) {
      t[i] = carry;
      carry = unite(t_1[i], t_2[i]);
    }
    if (есть 2 дерева из 3) {
      t[i] = -1;
      carry = unite(2 дерева);
    }
    if (есть 1 дерево из 3) {
      t[i] = то самое дерево;
      carry = -1;
    }
    if (ноль деревьев) {
      t[i] = -1;
    }
  }
}
```