Михаил Густокашин. Кучи.

Куча (по-английски heap) — структура данных, которая может выдавать минимальный (или максимальный) элемент за $O(1)$, добавление нового элемента и удаление минимального элемента происходит за $O(\log N)$ (где N — количество элементов в куче). Другие операции над кучей не определены (хотя при необходимости могут быть введены, однако эффективность их будет невысокой, и они обязаны поддерживать свойства кучи).

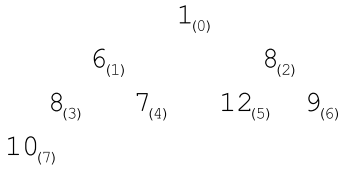

Рис. 1: Пример кучи

Другое название кучи — очередь с приоритетами, что и отражает ее сущность.

Сразу перейдем к рассмотрению реализации кучи на одномерном массиве. Назовем элементы с индексами $i \times 2 + 1$ и $i \times 2 + 2$ потомками элемента с индексом i. Элемент i будет называться предком этих элементов. Несложно заметить, что потомки двух разных элементов не пересекаются, и каждый элемент, кроме нулевого, является чьим-либо потомком. Основное свойство кучи: каждый элемент не больше своих потомков.

Например, массив 1, 6, 8, 7, 12, 9, 10 может являться кучей (напомним, что индексация в массиве начинается с нуля).

Для хранения кучи создадим структуру, аналогичную предыдущим:

```
typedef struct
{
  int hs;
  int val[MAXN];
} heap;
```

Опишем три функции. В первую очередь напишем функцию, возвращающую наименьший элемент. Она будет очень простая, т.к. из свойства кучи следует, что минимум находится в нулевом элементе:

```
int get_min(heap *h)
{
  return h->val[0];
}
```

Перед использованием этой функции необходимо обязательно проверить, что куча не пуста!

Следующей определим функцию добавления элемента в кучу. Новый элемент будем добавлять в конец кучи, а затем обменивать его с предком, пока он меньше, чем его предок или мы не достигли нулевого индекса. При этом свойство кучи не нарушится.

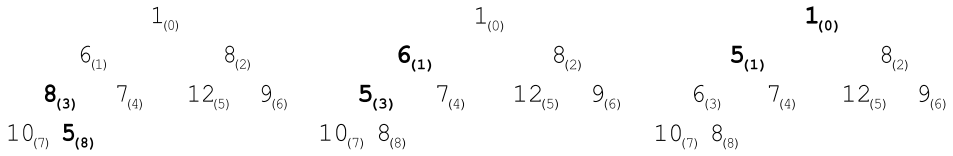

Рис. 2: Добавление в кучу (сравнивыемые элементы выделены)

```
void add_heap(heap *h, int x)
{
  int y, pos=h->hs, npos;
  h->val[h->hs++] = x;
  npos=(pos-1)/2;
  while (pos && h->val[pos] < h->val[npos]) {
    y=h->val[pos];
    h->val[pos]=h->val[npos];
    h->val[npos] = y;
    pos=npos;
    npos=(pos-1)/2;
  }
}
```

Как уже написано выше, сложность добавления элемента в кучу составляет $O(\log N)$ — каждый раз индекс текущего элемента уменьшается вдвое.

Кроме того, часто возникает задача удаления минимального элемента. Мы будем реализовывать это следующим образом: запишем на место нулевого элемента последний, уменьшим размер кучи на 1 и просеем нулевой элемент

по куче так, чтобы сохранилось свойство кучи. Будем реализовывать просеивание следующим образом: если элемент больше, чем меньший из своих потомков, то меняем их местами и продолжаем процесс, пока выполнено условие или мы не вышли за пределы кучи. В реализации этого алгоритма применена одна хитрость, которая будет пояснена ниже.

```
void del_heap(heap *h)
{
  int minp, pos=0, y;
  h->val[0]=h->val[--h->hs];
  while (pos*2+1 < h->hs) {
    y=pos*2+1;
    minp=h->val[y]<h->val[y+1]?y:y+1;
    if (h->val[pos]>h->val[minp]) {
      y=h->val[pos];
      h->val[pos]=h->val[minp];
      h->val[minp]=y;
      pos=minp;
    } else break;
  }
}
```

На первый взгляд, единственный случай, где теоретически возможна ошибка, когда у нас имеется всего один потомок (т.е. `pos*2+2 == h->hs`). Такой вариант возможен, если количество элементов в куче четно.

В этом случае мы выбираем минимум из первого потомка и первого элемента вне кучи, что, казалось бы, является грубой ошибкой. Но мы знаем, что просеиваемое число и первое число вне кучи, равны. Это гарантирует нам, что обмена с элементом вне кучи не произойдет, а если необходим обмен с единственным потомком, то он будет выполнен корректно.