За хийповете и стоенето им

Хийпът(Heap) е една от най-простите и същевременно широко приложими нетривиални структури. Използва се за организиране на приоритетна опашка. Представлява пълно двоично дърво от ключове, със свойството ключът във всеки възел да е по-малък(или по-голям) от ключовете на децата му. Структурата е изключително ефективна, и поради пълнотата на двоичното дърво, позволява реализация в паметта под формата на масив.

За работа със хийпа (поддръжка на вмъкване, изтриване, промяна на приоритет и други), се въвеждат двете главни операции за „балансиране" на структурата (както двоичните дървета за търсене си имат ротации) – избутване нагоре и избутване надолу.

Примерен код – C++

```
#include <iostream>
using namespace std;

const int max_h = 1024;
// min-heap

// this is the heap as a static array
// the size of the heap will be in heap[0]
// the minimal element will be in heap[1]
int heap[max_h];

// Move an element to the top of the heap
void fix_up(int i){
	while(i > 0 && heap[i] < heap[i/2])
		swap(heap[i], heap[i/2]), i/=2;
}

// Move an element to the bottom of the heap
void fix_dn(int i){
	int p = 2*i;
	while(p <= heap[0]){
		if(p < heap[0] && heap[p+1] < heap[p]) p++;
		if(heap[p] >= heap[i]) break;
		swap(heap[i], heap[p]);
		i = p;
		p = 2*i;
	}
}
```

Двете основни операции добавяне на нов елемент и изтриване на минималния се извършват съответно като се добави новият елемент след последният във масива, и се избута нагоре, и като

първият елемент (минималният) се размени с последният в масива, и след това последният се избута надолу:

Примерен код – C++

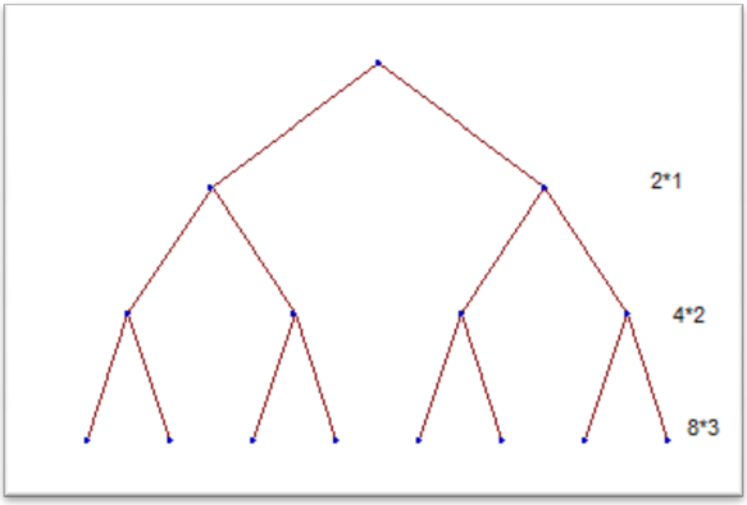
```
void add(int k){
	heap[++heap[0]] = k;
	fix_up(heap[0]);
}

void del(){
	if(heap[0] <= 1) heap[0] = 0;
	else{
		swap(heap[1], heap[heap[0]]);
		heap[0]--;
		fix_dn(1);
	}
}
```

Най-естественият подход за начално стоене на хийп от първоначално зададен масив ключове, е да се започне със празен хийп и да се добавя всеки следващ елемент. Елементарните операции имат логаритмична сложност, затова общата сложност на това стоене е $O(n \log n)$.

Примерен код – C++

```
// We assume the elements to be heapified are in
heap[1..heap[0]]
void heapify1(){
	for(int i = 2; i <= heap[0]; i++) fix_up(i);
}
```

Интересно е, че малко след откриването на хийпа, е изнамерен прост начин да се извърши началното строене със линейна сложност. Идеята е, вместо да се стори хийпът от корена към листата, да се стои на обратно, като по този начин половината от елементите (листата) въобще не се разглеждат. Ключовото наблюдение е, че в едно пълно двоично дърво повечето от възлите са по-близо до листата отколкото до корена, и само малка част са далече. За да бъдем по-конкретни, да разгледаме пълно двоично дърво с 2^k възела(фиг. 1).



Фиг. 1

Вижда се, че сумата от разстоянията от всеки възел до корена е:

$$S_1 = (k-1)2^{k-1} + (k-2)2^{k-2} + \cdots + 1.2^1 = \sum_{i=0}^{k-1} i2^i = 2 - 2^{k+1} + 2^k k$$

От друга страна, сумата от разстоянията от всеки възел до най-близкото листо е:

$$S_2 = \sum_{i=0}^{k-1} (k-1-i)2^i = 2^k - k - 1$$

Сега, ако положим $n = 2^k, k = \log n$:

$$S_1 \cong 2^k k = O(n \log n)$$

$$S_2 \cong 2^k = O(n)$$

Следователно, ако строим хийпа от листата към корена, ще постигнем линейна сложност!

Примерен код – C++

```
void heapify2(){
      for(int i = heap[0]/2; i > 0; i--) fix_dn(i);
}
```

Ценното тук е не просто новият, асимптотично по-добър алгоритъм, а и методът, идеята, за намаляване на сложността, вървейки от листата към корена.