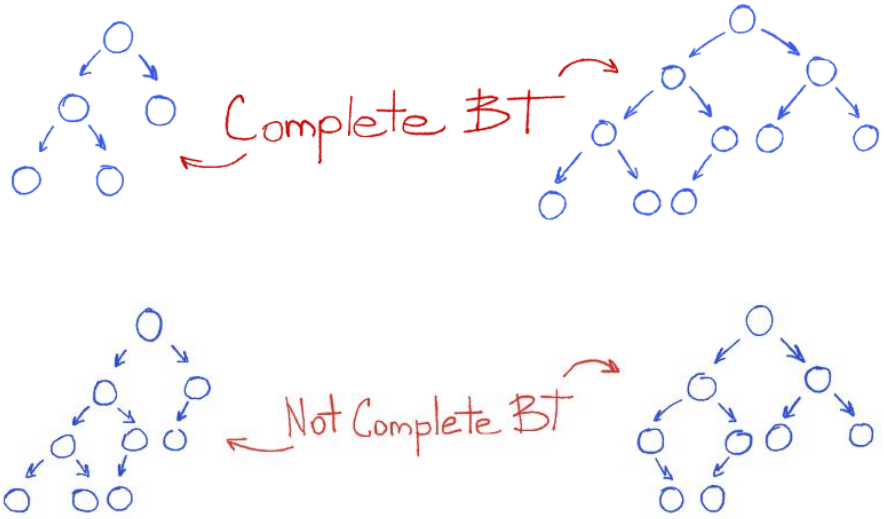
Двоичные кучи. Очередь с приоритетами

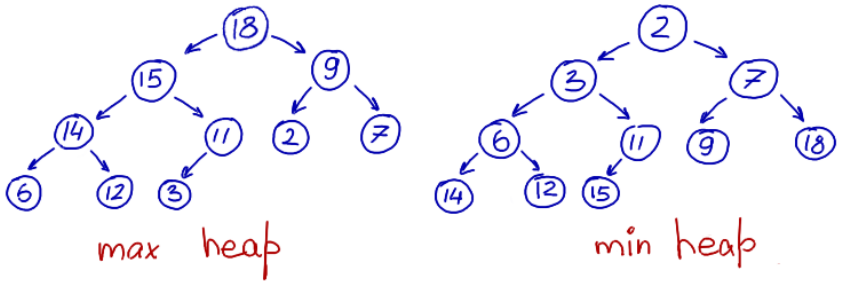
Двоичная куча (пирамида, сортирующее дерево, binary heap) это полное двоичное дерево, где поддерживается свойство порядка размещения вершин (heap order property).

Напомним, что **полное двоичное дерево** (complete binary tree) это такое дерево, все уровни которого заполнены, возможно, за исключением последнего:

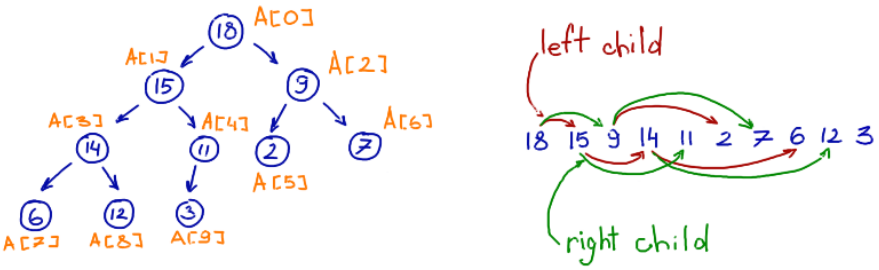


Свойство порядка размещения вершин (heap order property) заключается в том, что двоичная куча может быть **максимальной** (maximum heap) или **минимальной** (minimum heap):

- *maximum heap* - если значение (приоритет) в каждом узле больше или равно значениям в узлах потомков;
- *minimum heap* - если значение (приоритет) в каждом узле меньше или равно значениям в узлах потомков.



Двоичную кучу легко представить в виде массива[1]:



Вершиной такого дерева является элемент с индексом 0. Для i-го узла достаточно просто вычислить его родителя и его потомков:

```
int parent(int i)
{
    return (i-1) / 2;
}

int left(int i)
{
    return 2*i + 1;
}

int right(int i)
{
    return 2*i + 2;
}
```

Предположим, что у нас есть куча со свойством *maximum heap*, тогда операции добавления или удаления элементов из кучи могут нарушить это свойство (аналогично и для кучи со свойством *minimum heap*). Восстановить это свойство позволит функция heapify:

```
bool max_heap(EType a, Etype b) { return a > b ? true : false; }
bool min_heap(EType a, Etype b) { return a < b ? true : false; }

void heapify(Array A, int heap_size, int i, bool (*cmp)(EType,
EType))
{
    int l = left(i);
    int r = right(i);
    int current = i;

    if (l < heap_size && cmp(get(A, l), get(A, current)))
        current = l;
    if (r < heap_size && cmp(get(A, r), get(A, current)))
        current = r;
```

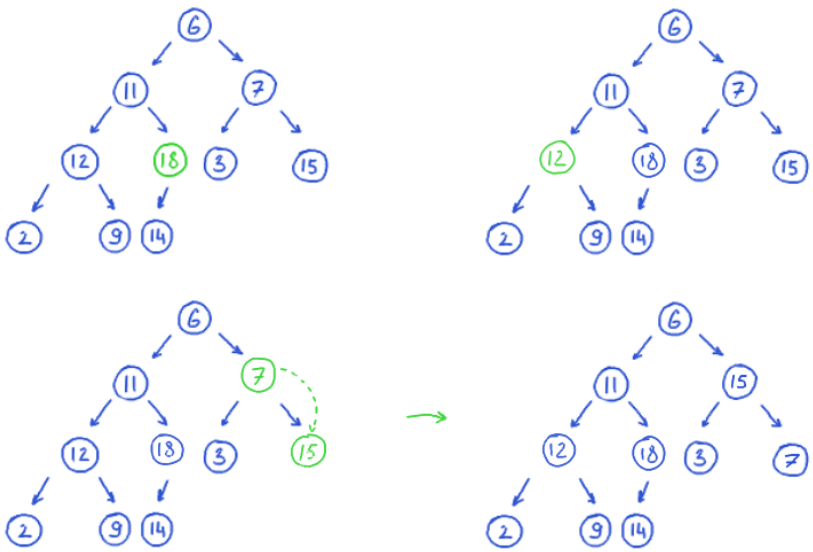
```
    if (current != i)
    {
        swap(A, i, current);
        heapify(A, heap_size, current, cmp);
    }
}
```

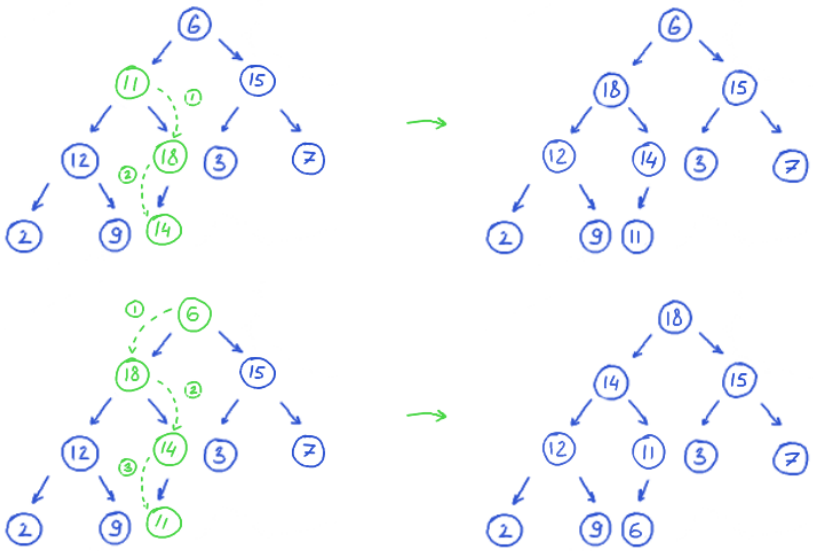
Если `i`-й элемент больше (меньше), чем его сыновья, то всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами `i`-й элемент с наибольшим (наименьшим) из его сыновей, после чего выполняем `heapify` для этого сына.

Для построения кучи из неупорядоченного массива элементов достаточно последовательно применить `heapify` ко всем элементам кучи, у которых есть сыновья:

```
void build_heap(Array A, bool(*cmp)(EType, EType))
{
    for (int i = size(A)/2 - 1; i >= 0; i--)
       heapify(A, size(A), i, cmp);
}
```

Иллюстрация работы функции `build_heap`:





Над кучами также можно выполнять следующие операции:

- `heap_insert(H, e, cmp)` - вставка элемента e в кучу H;
- `heap_get(H)` - получить значение максимального/минимального элемента (без его удаления) в куче H;
- `heap_extract(H)` - удалить максимальный/минимальный элемент из кучи H;
- `heap_union(H1, H2, cmp)` - объединить кучи H1 и H2.

Важно помнить, что куча может быть или максимальной или минимальной.

Алгоритм вставки элемента достаточно простой, мы добавляем новый элемент в конец массива и затем двигаем его вверх по дереву до тех пор, пока значение в родителе не станет больше/меньше чем значение, которое мы добавляем:

```
void heap_insert(Array A, EType k, bool (*cmp)(EType, EType))
{
    add(A, k);
    heap_shift_up(A, size(A)-1, cmp);
}

void heap_shift_up(Array A, int i, bool (*cmp)(EType, EType))
{
    while (i > 0 && cmp(get(A, i), get(A, parent(i))))
    {
        swap(A, i, parent(i));
        i = parent(i);
    }
}
```

Для того, чтобы получить значение максимального или минимального элемента (в зависимости от поддерживаемого свойства) достаточно обратиться к нулевому элементу массива:

```
EType heap_get(Array A)
{
    return get(A, 0);
}
```

Удаление элемента производится путем замены значения, которое находилось в вершине, на значение последнего узла в дереве с последующим удалением этого узла. После замены выполняется функция `heapify` над новой вершиной:

```
EType heap_extract(Array A, bool (*cmp)(EType, EType))
{
    if (size (A) <= 0)
    {
        fprintf(stderr, "Heap underflow");
        exit(1);
    }
    EType e = heap_get(A);
    set(A, 0, get(A, size(A) - 1));
    remove(A);
    heapify(A, size(A), 0, cmp);
    return e;
}
```