heap sort

```
算法 Heapsort (X, n)
输入：X（下标从 1 至 n 的数组）
输出：X（排序后的数组）
begin
    Build_Heap (X) ; {见下文}
    for i := n downto 2 do
        swap (A [1], A [i]) ;
        Rearrange_Heap (i − 1)

        {基本等同于图 4.7 中的程序 Remove_Max_from_Heap}
end
```

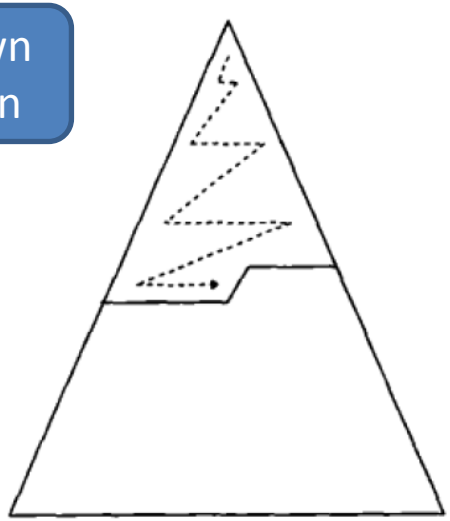
$O(n \lg n)$

图 6.13　算法 *Heapsort*

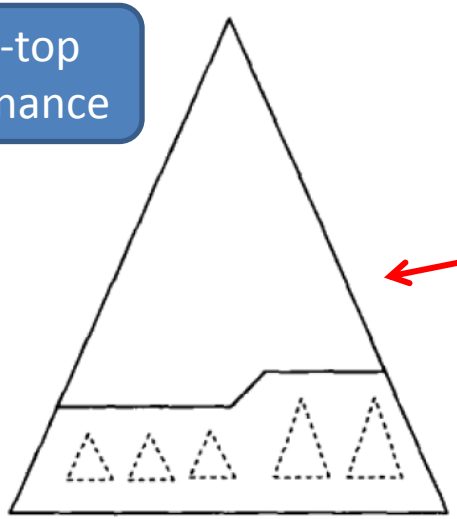
Two ways to build heap

substantial different performance !

Top-down insertion

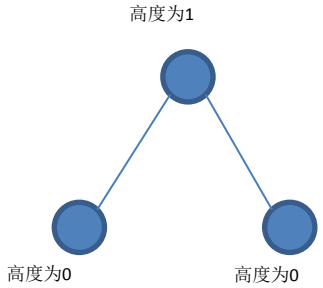


Down-top maintenance



Better performance!

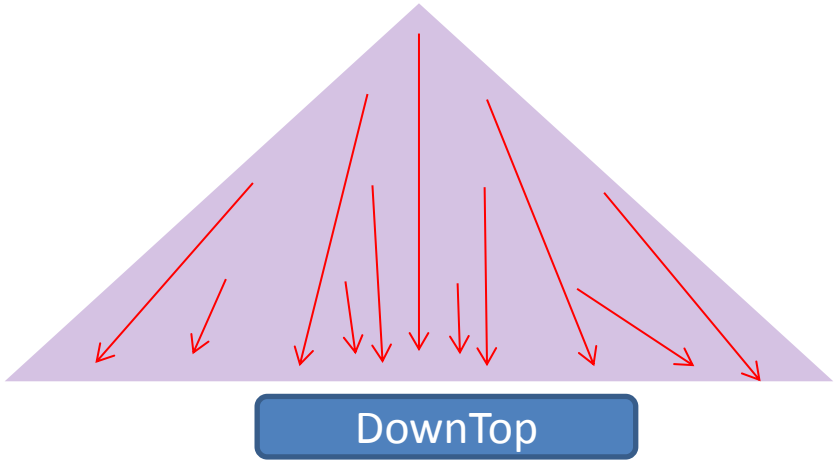
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
2	6	8	5	10	9	12	(14)	15	7	3	13	4	11	16	(1)
2	6	8	5	10	9	(16)	14	15	7	3	13	4	11	(12)	1
2	6	8	5	10	(13)	16	14	15	7	3	(9)	4	11	12	1
2	6	8	5	10	13	16	14	15	7	3	9	4	11	12	1
2	6	8	(15)	10	13	16	14	(5)	7	3	9	4	11	12	1
2	6	(16)	15	10	13	(12)	14	5	7	3	9	4	11	(8)	1
2	(15)	16	(14)	10	13	12	(6)	5	7	3	9	4	11	8	1
(16)	15	(13)	14	10	(9)	12	6	5	7	3	(2)	4	11	8	1

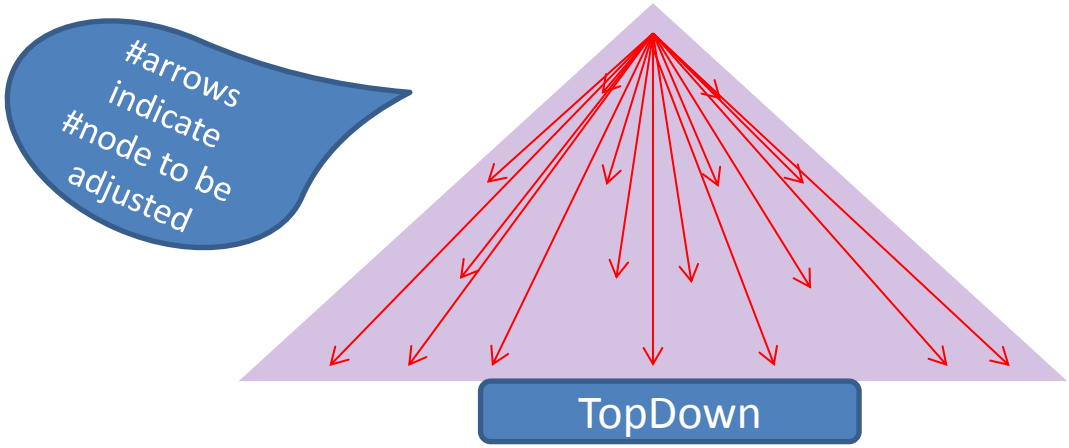


What if Top-down

- 6
- 6 2
- 8 2 6
- 8 5 6 2
- 10 8 6 2 5
- ∘ ∘ ∘ ∘

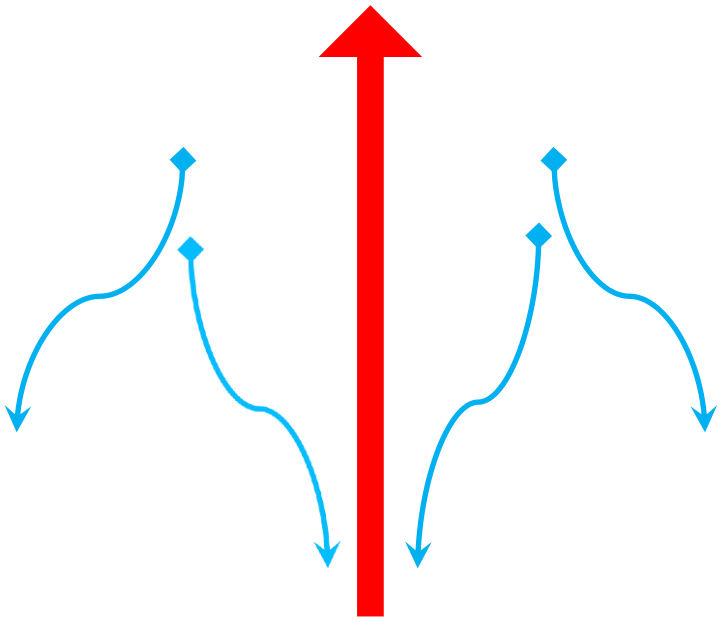
Swap次数超过自底而上法!
Performance speed up ~ approx 50%.





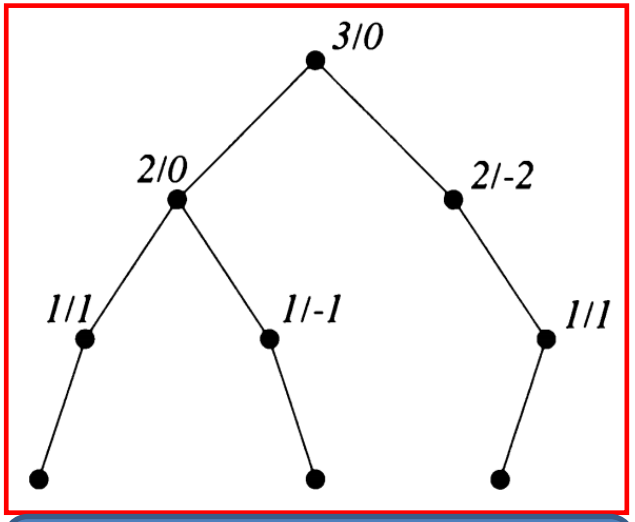
Down-top maintenance

- 隐式建立完全二叉树，将每个节点看做一个子堆的堆头.
- 倒序逐个处理元素，对其代表的子堆，做自上而下的维护.



Some Properties of Trees

- **Level** – The level of a node is defined by 1 + the number of connections between the node and the root.
- **Height of tree** –The height of a tree is the number of edges on the longest downward path between the root and a leaf.
- **Height of node** –The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Depth** –The depth of a node is the number of edges from the node to the tree's root node.



本textbook可以此图为范例！

GPU parallel sorting

- http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
- https://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf
- http://en.wikipedia.org/wiki/NVIDIA_GPU

Max & Min

- Non-divide-and-conquer method is worse.
- Best complexity : 3n/2 . (step=3: 5n/3, step=4: 6n/4)

[So each time taking a **pair** out is the best.]

K-th Min

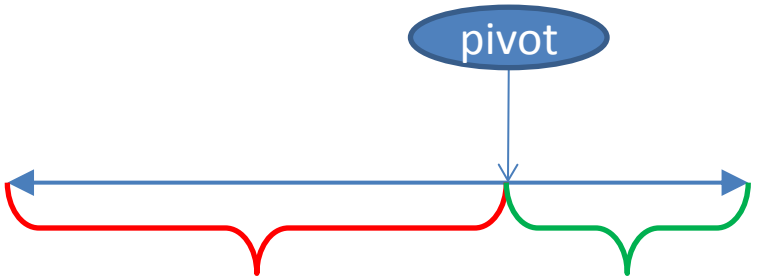
问题　已知序列 $S=x_1, x_2, \cdots, x_n$ 以及整数 k，$1 \leqslant k \leqslant n$，试查找 S 中第 k 小的数。

算法 *Selection* (*X*, *n*, *k*)
输入：*X*（下标从 1 至 *n* 的数组）和 *k*（某个整数）
输出：*S*（第 *k* 小的数；数组 *X* 也做过变动）

```
begin
    if (k < 1) or (k > n) then print "error"
    else
        S := Select(1, n, k)
end

procedure Select (Left, Right, k) ;
begin
    if Left = Right then
        Select := Left
    else
        Partition (X, Left, Right) ; {见图 6.9}
        Let Middle be the output of Partition;
        if Middle − Left + 1 ⩾ k then
            Select (Left, Middle, k )
        else
            Select (Middle + 1, Right, k − (Middle − Left + 1))
end
```

图 6.16　算法 *Selection*



错误！ Select= X[Left]

Selection
常用于查找中数，
why?Pivot is
approaching the middle

Majority

令 E 是整数序列 $x_1, x_2, \cdots, x_n$。E 中 x 的重数（multiplicity）是 x 在 E 中出现的次数。如果某个数 z 的重数大于 $n/2$，则它就是 E 中的众数（majority）。

- **众数必是中数**; 若不是，则众数不存在.
- 如果Selection(n/2)返回的中数不是众数，则众数不存在！

中数法:

```
Majority(X,n)
{
        int m = Selection(X, n, n/2);
        return CheckMajority(m);
}
```

[观察]众数保留:

若 $x_i \neq x_j$，且两个数都被删除，那么原来集合中的众数现在仍为新集合中的众数。

- voting:

```
算法 Majority (X, n)
输入：X（有 n 个正数的数组）
输出：Majority（X 中的众数；若不存在，则输出-1）
begin
    C := X[1];
    M := 1;
    {首次扫描：删去除 C 以外的所有选票}
    for i := 2 to n do
        if M = 0 then
            C := X[i];
            M := 1
        else
            if C = X[i] then M := M + 1
            else M := M - 1;
    {第二次扫描：检验 C 是否为众数}
    if M = 0 then Majority := -1
    else
        Count := 0;
        for i := 1 to n do
            if X[i] = C then Count := Count + 1;
        if Count > n/2 then Majority := C
        else Majority := -1
end
```

Works for dominant voting (e.g. win over 60% votes)

Compression!

- Synonyms : source coding, bit-rate reducing

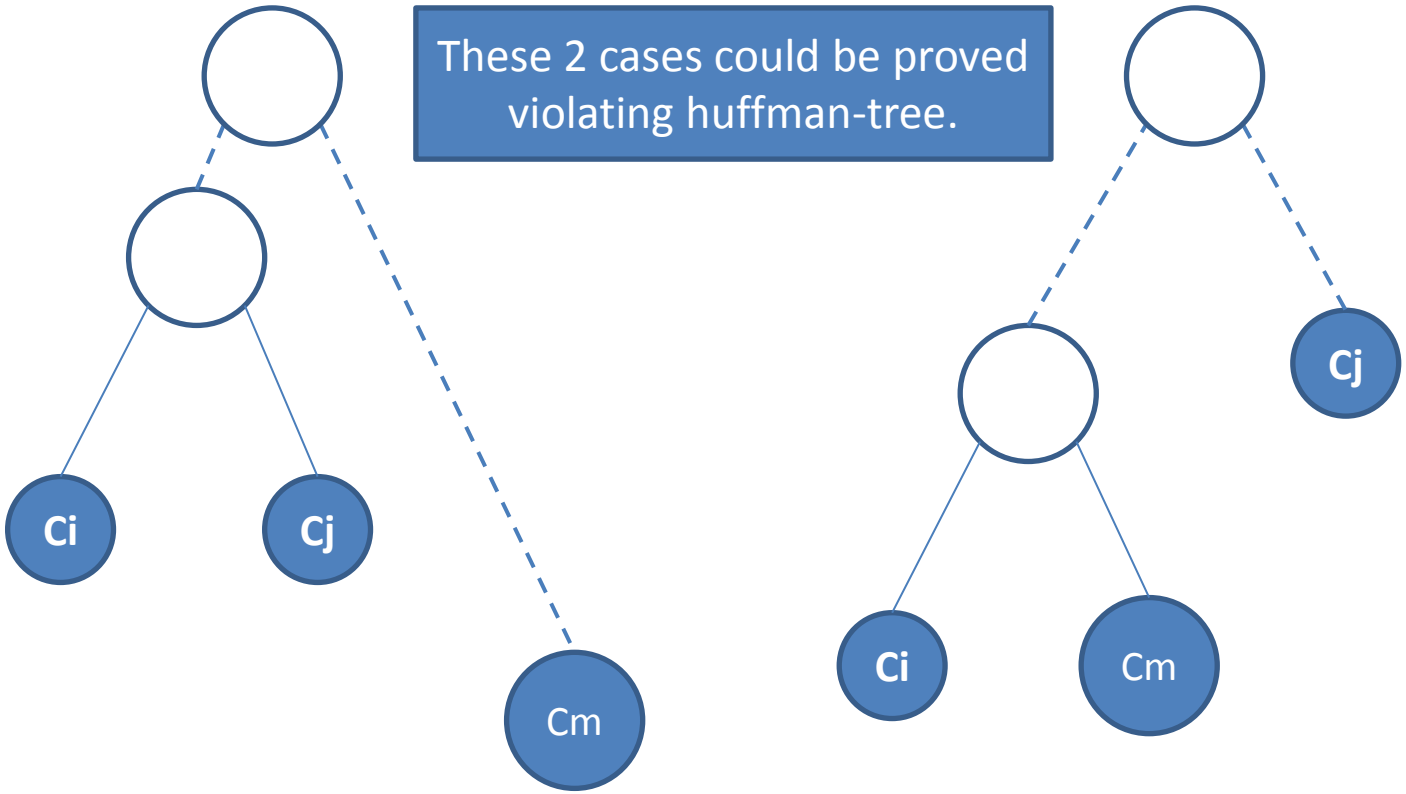
- Huffman-encoding for text bit-streaming
  -lossless data compression
  -variable length encoding.

Huffman Tree(optimal B-tree)

- Degree(vi) = 2 or 0 ;
- With lowest sum-weighted-path-length.

property:

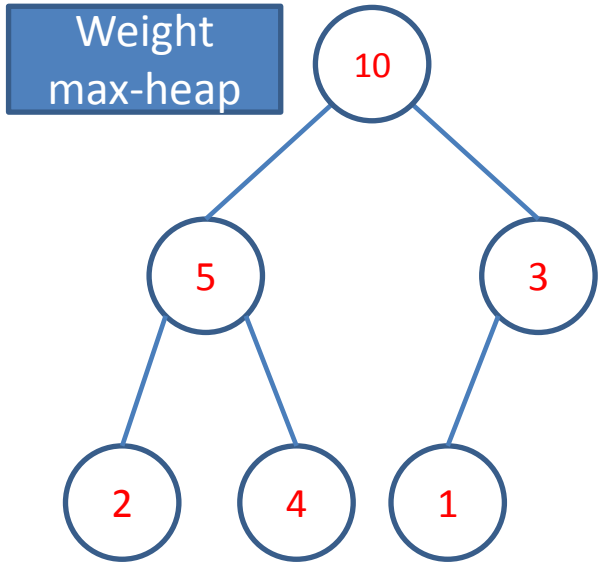
If ci, cj has the lowest weights, they are at the lowest level, sharing a parent node.



Huffman-encoding

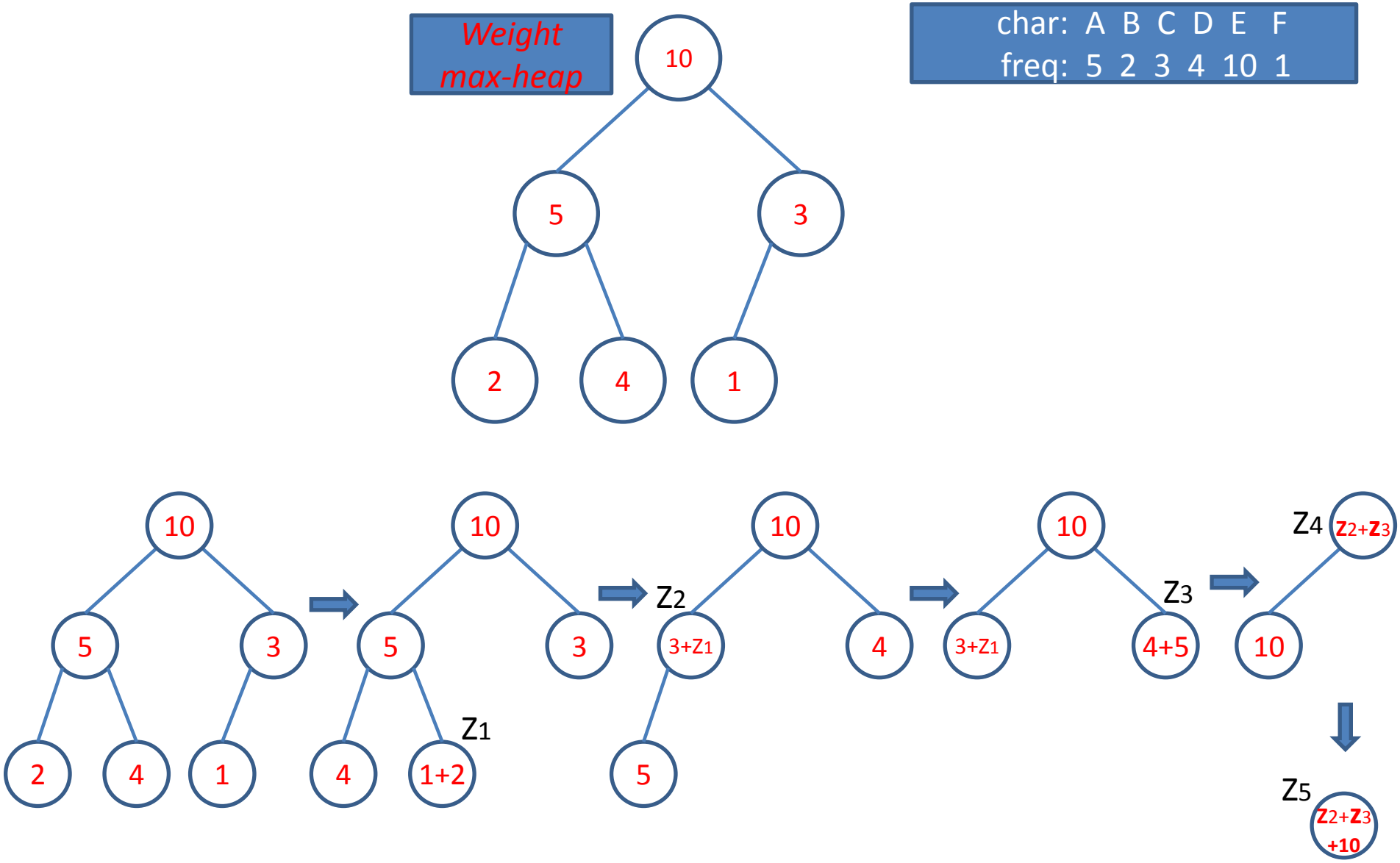
```
算法 Huffman_Encoding (S, f)
输入：S（字符串）和 f（字频数组）
输出：T（S 的霍夫曼树）
begin
    insert all characters into a heap H according to their frequencies ;
    while H is not empty do
        if H contains only one character X then
            make X the root of T
        else
            pick two characters X and Y with lowest frequencies
                and delete them from H ;
            replace X and Y with a new character Z whose frequency is
                the sum of the frequencies of X and Y ;
            insert Z to H ;
            make X and Y children of Z in T {Z 仍没有父节点}
end
```

图 6.18　算法 *Huffman_Encoding*



以正常方式minHeap建立编码树
的流程图在书上106页红色图片

下一页是maxHeap的，推出的元素顺序
没有错，但是这maxHeap压根没帮上忙！


Weight
max-heap
10
5
3
2
4
1
char: A B C D E F
freq: 5 2 3 4 10 1
10
5
3
2
4
1
10
5
3
4
1+2
Z1
10
3+Z1
4
5
Z2
10
3+Z1
4+5
Z3
Z4 Z2+Z3
10
Z5
Z2+Z3
+10