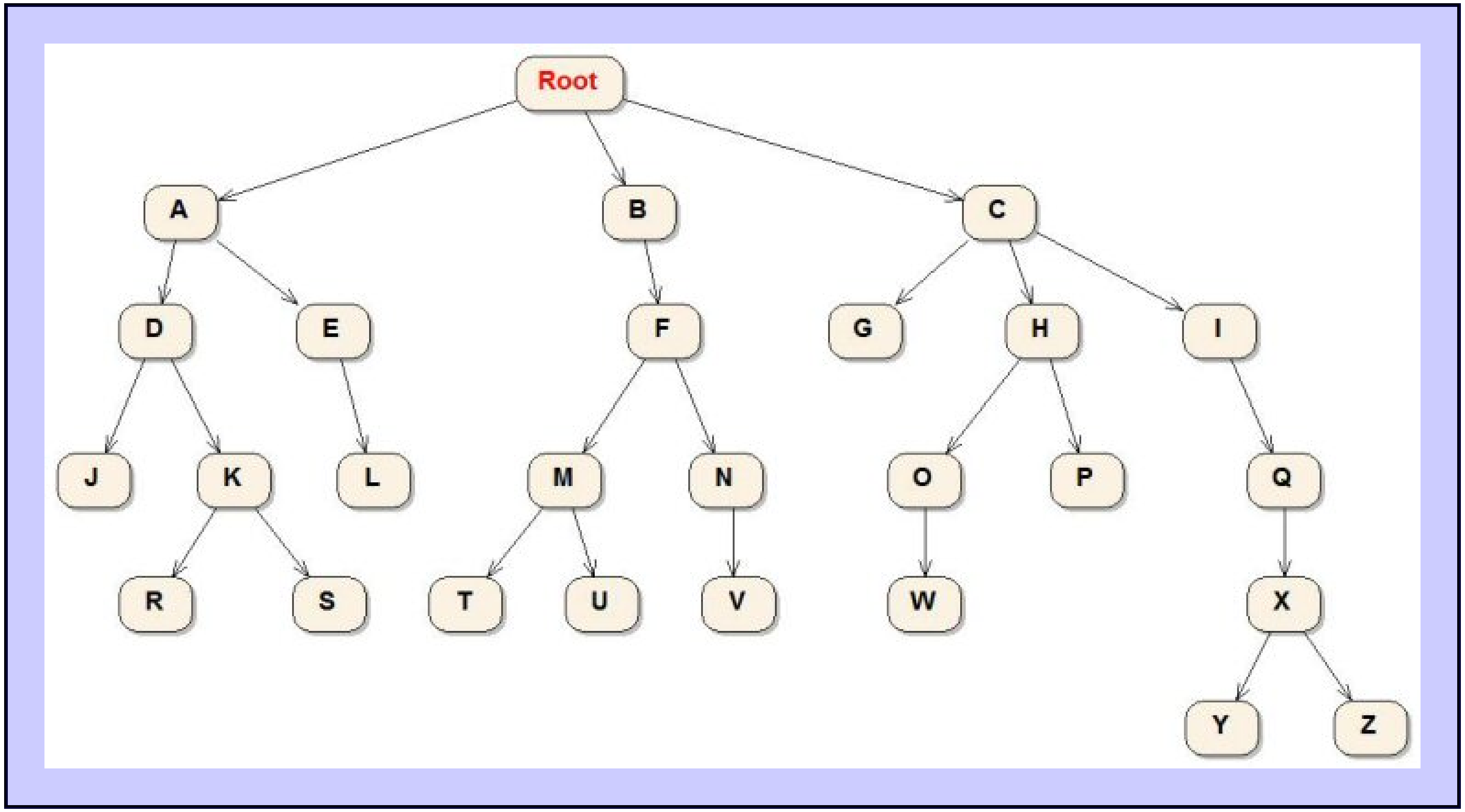
Keynotes on Data Structure

Instructor: Shizhe Zhou

Course Code:00125401

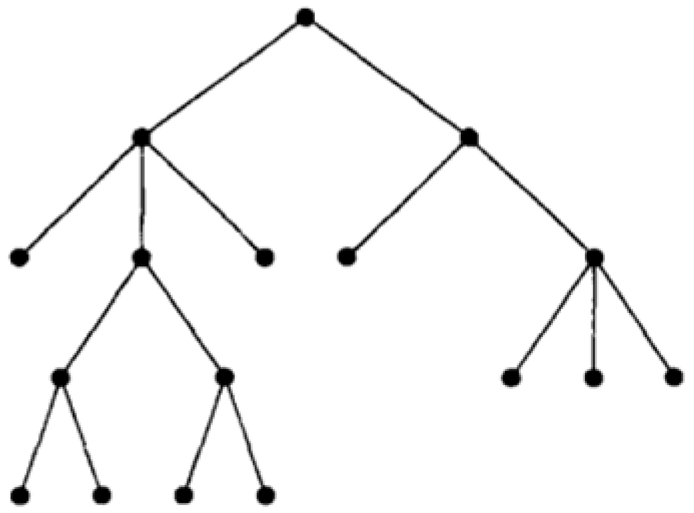
tree

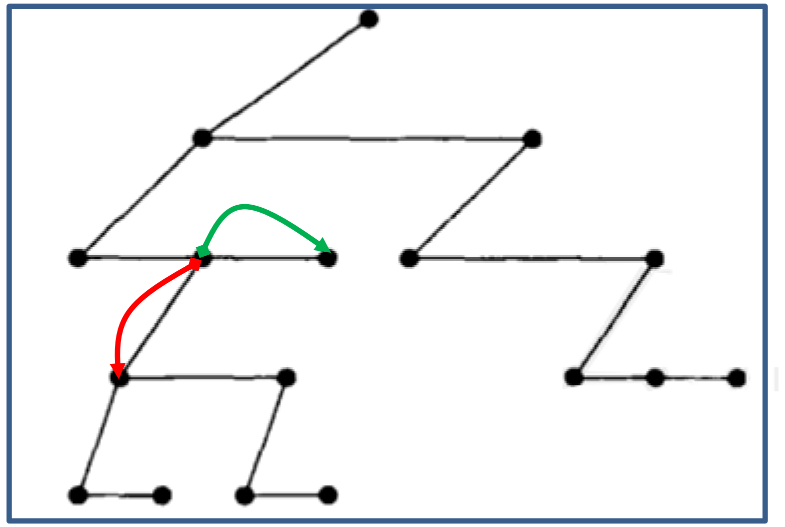
- Populate Alphabet



Binary representation

NEED POINTER!



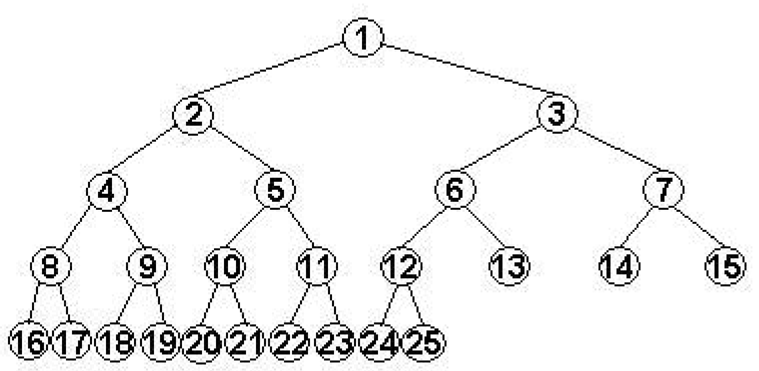


2pointers for each node:

1st points to its first child;

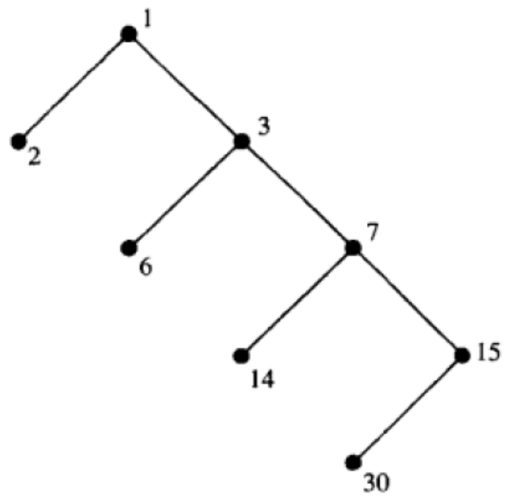
2nd points to its sibling(if any).

Implicit representation

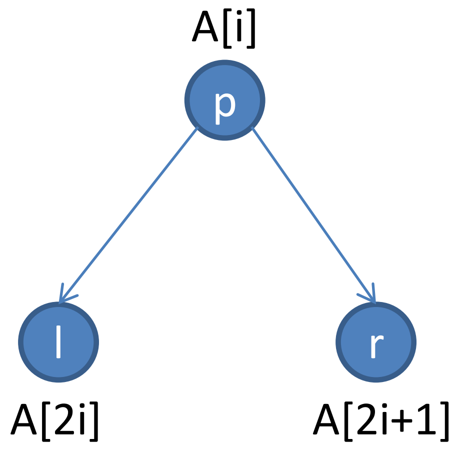


A Complete Binary Tree

12 internal nodes, 13 terminal nodes



NEED NO POINTER!



Linear storage in an array!

heap

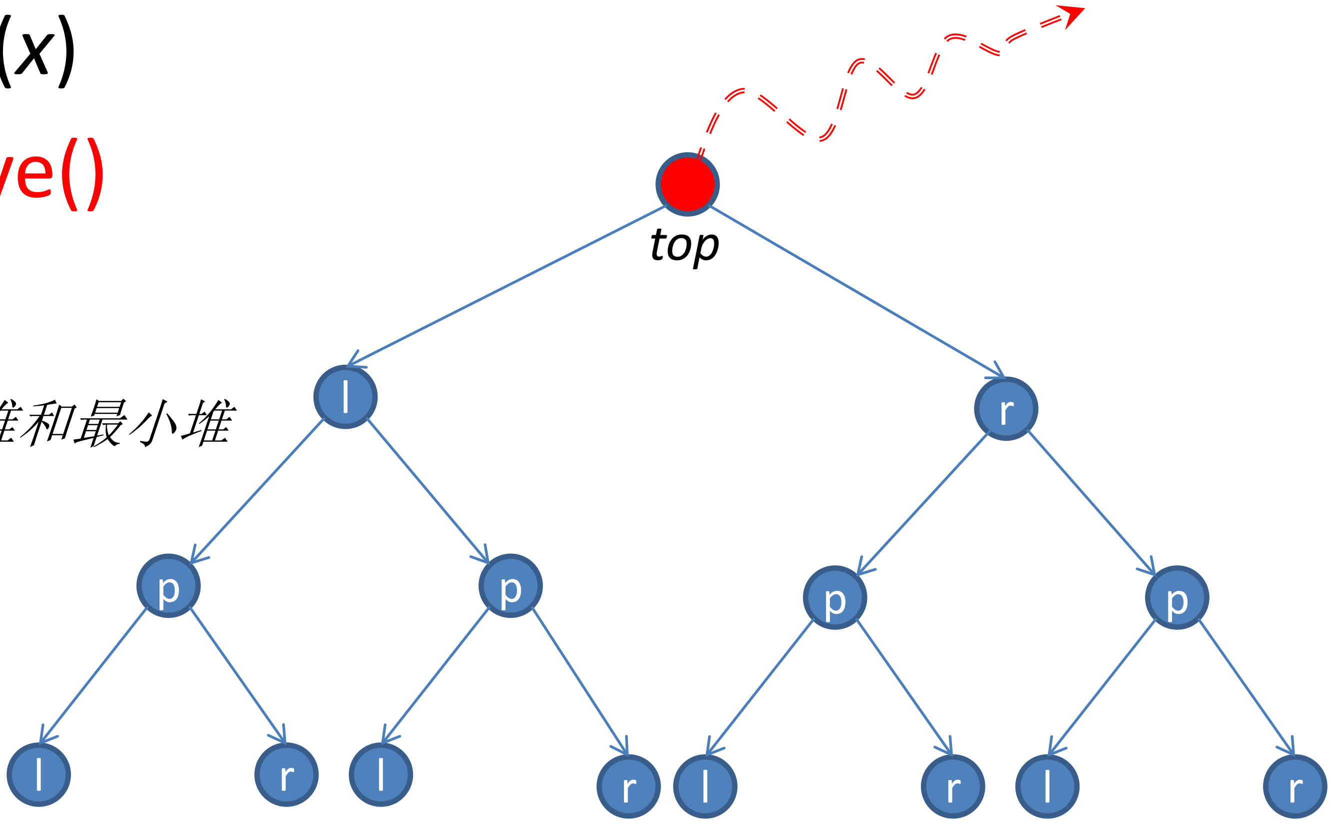
Every subtree of a heap is also a heap.

- Priority-queue

  -insert(*x*)

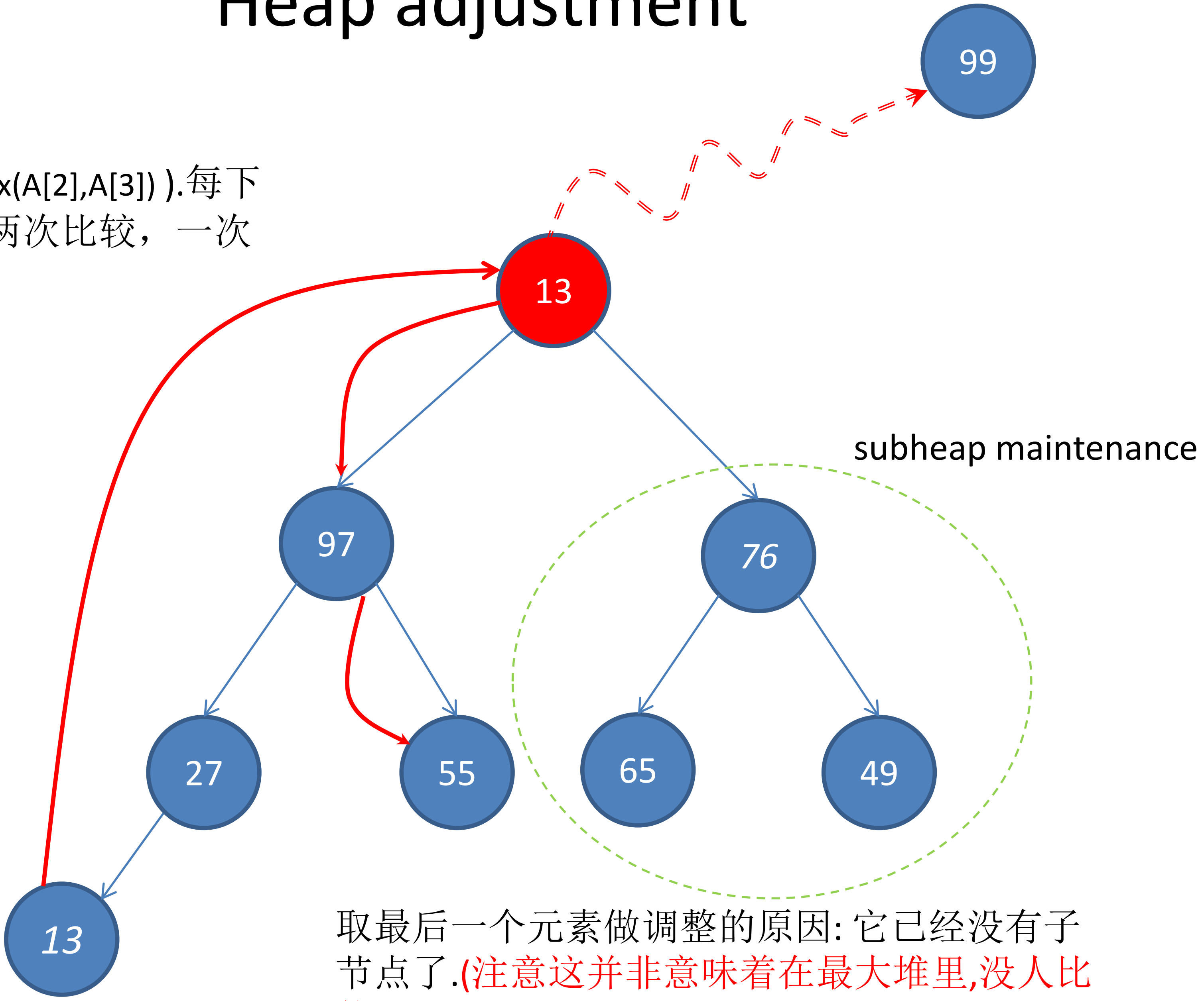
  -remove()

两种：最大堆和最小堆



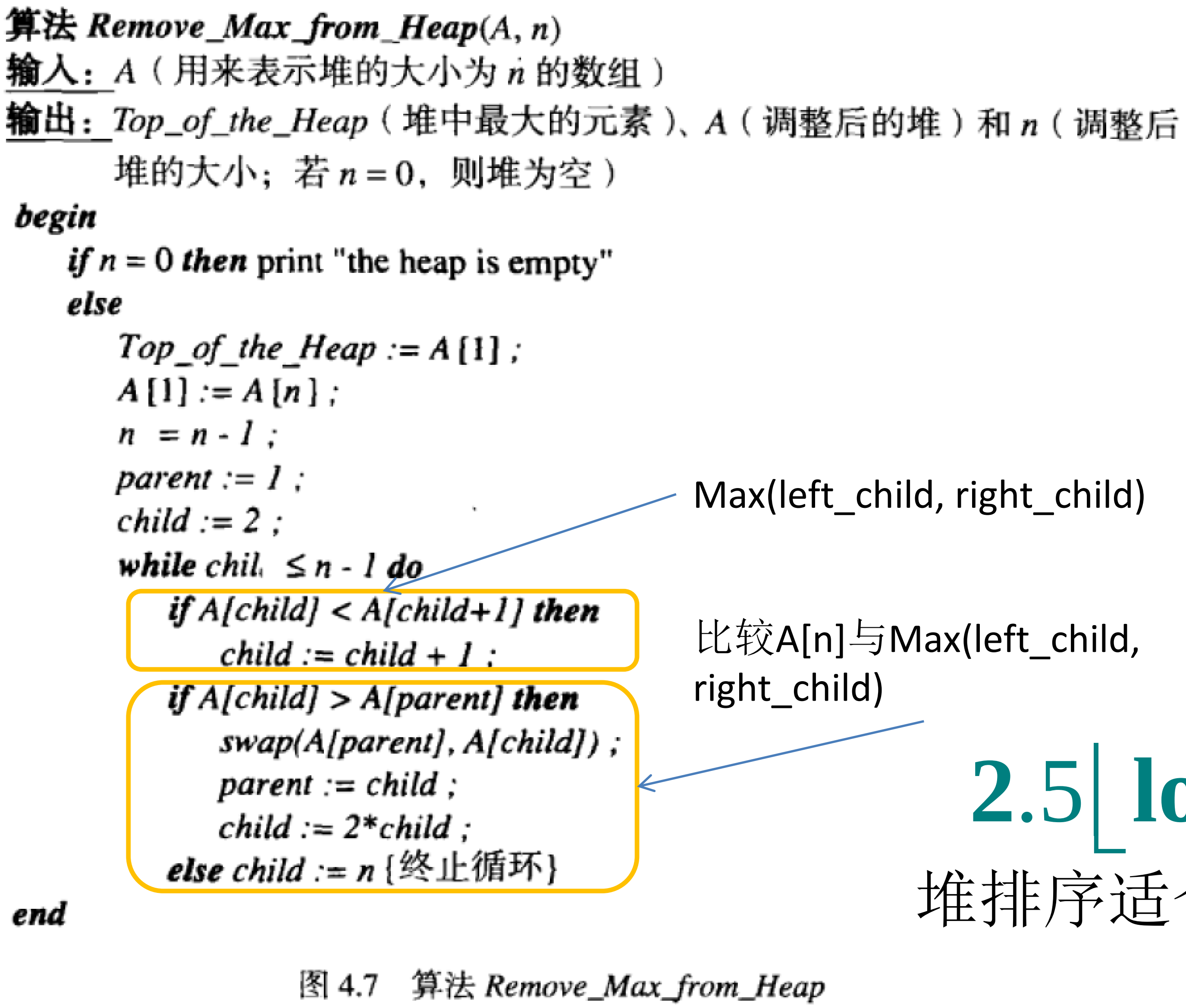
Heap adjustment

Max(A[1], max(A[2],A[3])).每下降一层，需两次比较，一次交换



取最后一个元素做调整的原因: 它已经没有子节点了.(注意这并非意味着在最大堆里,没人比他小了)

Heap adjustment *for 最大堆*



算法 ***Remove_Max_from_Heap***(A, n)
输入： A（用来表示堆的大小为 n 的数组）
输出： *Top_of_the_Heap*（堆中最大的元素）、A（调整后的堆）和 n（调整后堆的大小；若 $n = 0$，则堆为空）

```
begin
    if $n = 0$ then print "the heap is empty"
    else
        Top_of_the_Heap := A[1] ;
        A[1] := A[n] ;
        $n = n - 1$ ;
        parent := 1 ;
        child := 2 ;
        while chil $\le n - 1$ do
            if A[child] < A[child+1] then
                child := child + 1 ;
            if A[child] > A[parent] then
                swap(A[parent], A[child]) ;
                parent := child ;
                child := 2*child ;
            else child := n {终止循环}
end
```

图 4.7　算法 *Remove_Max_from_Heap*

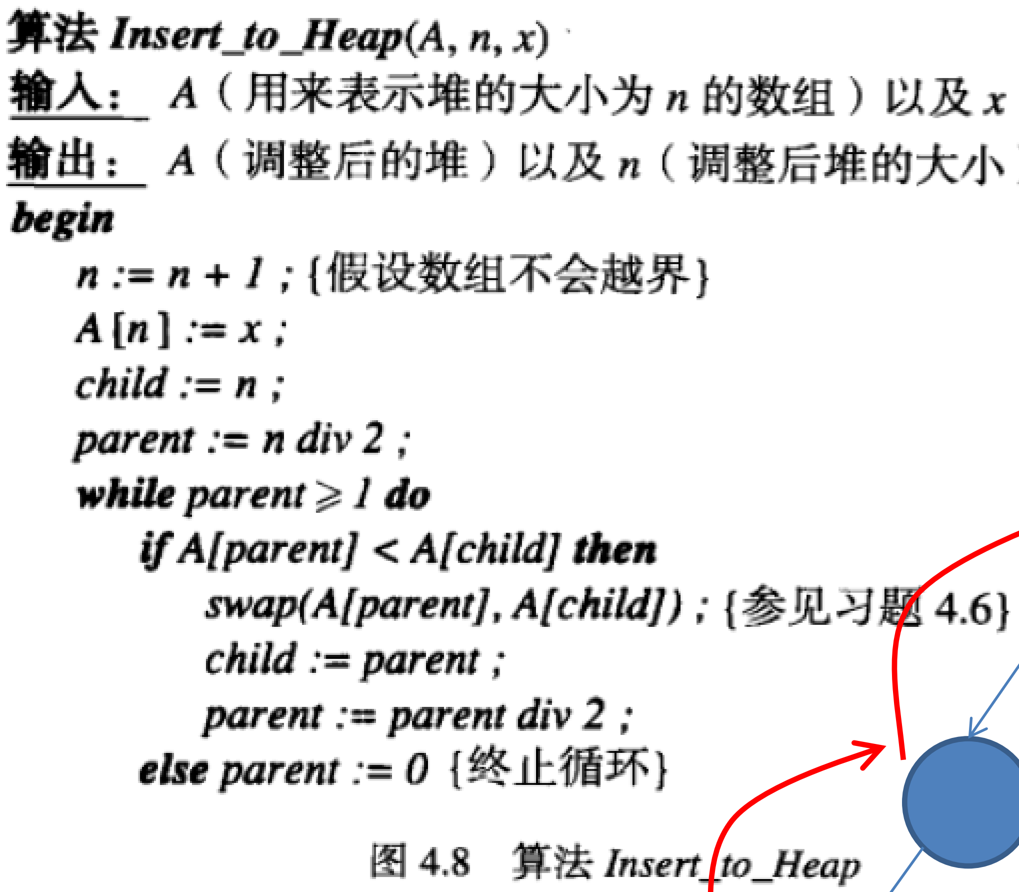
Max(left_child, right_child)

比较A[n]与Max(left_child, right_child)

$$2.5\lfloor \log_2 n \rfloor$$

堆排序适合较大的n

insert(*x*) for 最大堆



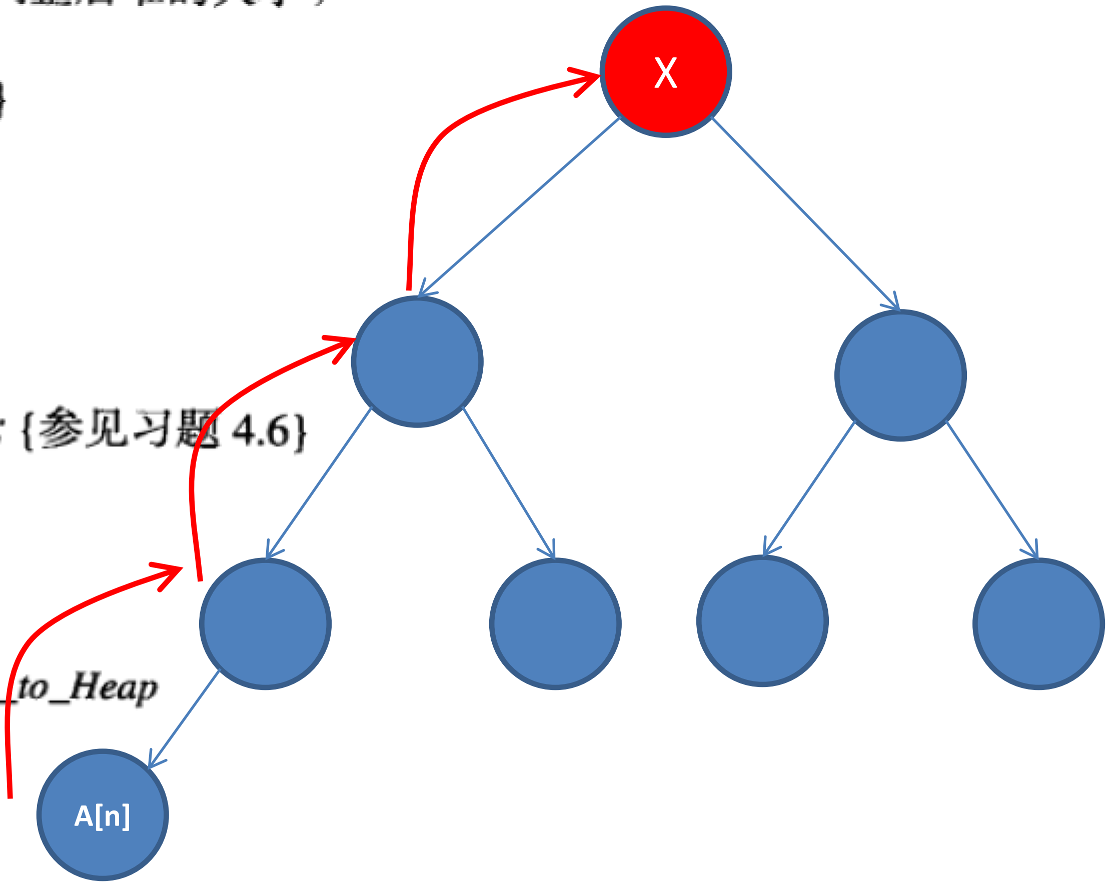
算法 ***Insert_to_Heap*(*A*, *n*, *x*)**

输入： *A*（用来表示堆的大小为 *n* 的数组）以及 *x*（某个数）

输出： *A*（调整后的堆）以及 *n*（调整后堆的大小）

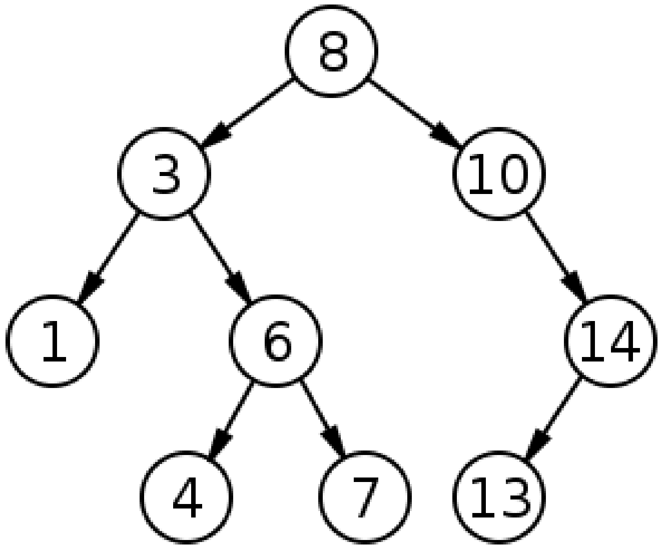
```
begin
    n := n + 1 ; {假设数组不会越界}
    A[n] := x ;
    child := n ;
    parent := n div 2 ;
    while parent ⩾ 1 do
        if A[parent] < A[child] then
            swap(A[parent], A[child]) ; {参见习题 4.6}
            child := parent ;
            parent := parent div 2 ;
        else parent := 0 {终止循环}
```

图 4.8 算法 *Insert_to_Heap*



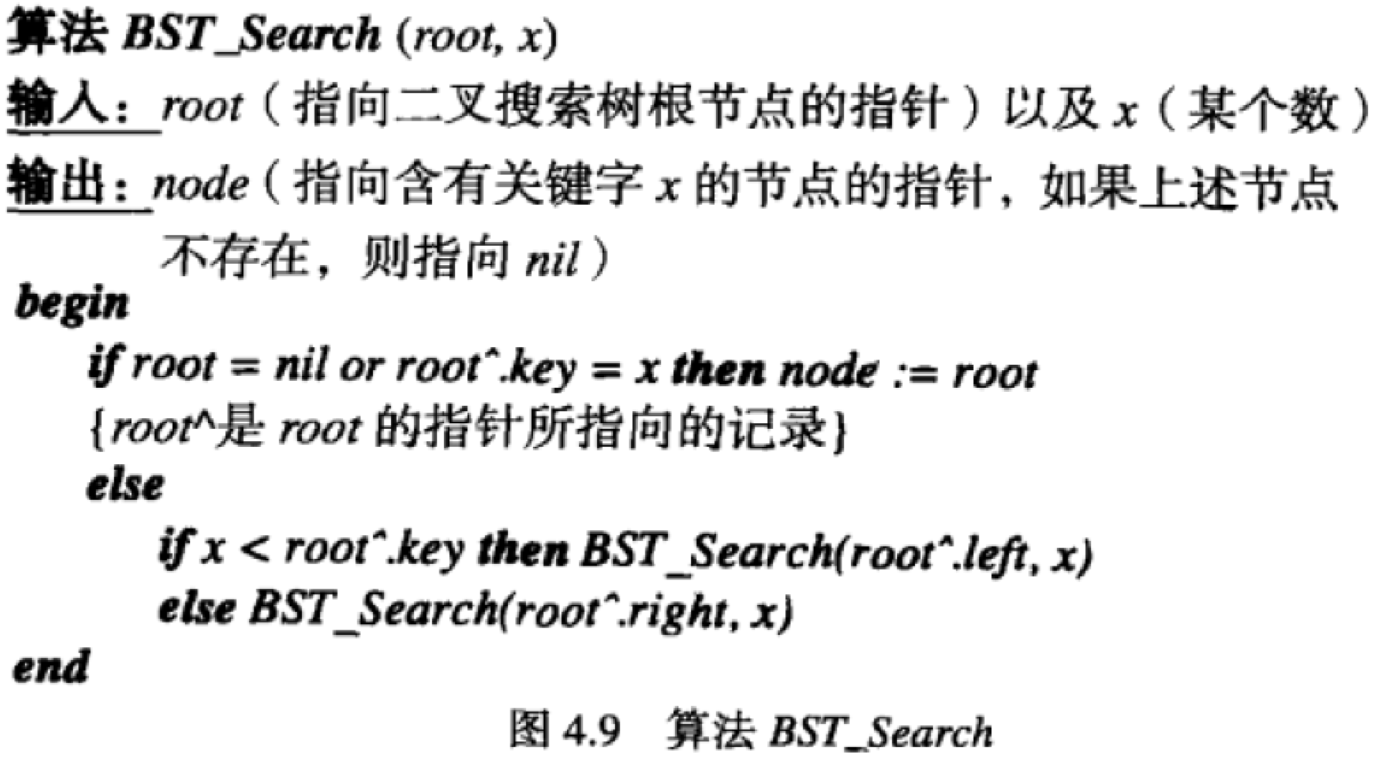
BST: Binary Search Tree

- 左小右大.
- 右左子树都是BST树.
- 无重节点.
- 表示的是record,存储的是key.
- 支持快速Sorting和Searching.
- Base DS for *set*, *multisets*, *associative arrays*



BST search

- Simple recursive compare



算法 ***BST_Search*** (*root, x*)

输入： *root*（指向二叉搜索树根节点的指针）以及 *x*（某个数）

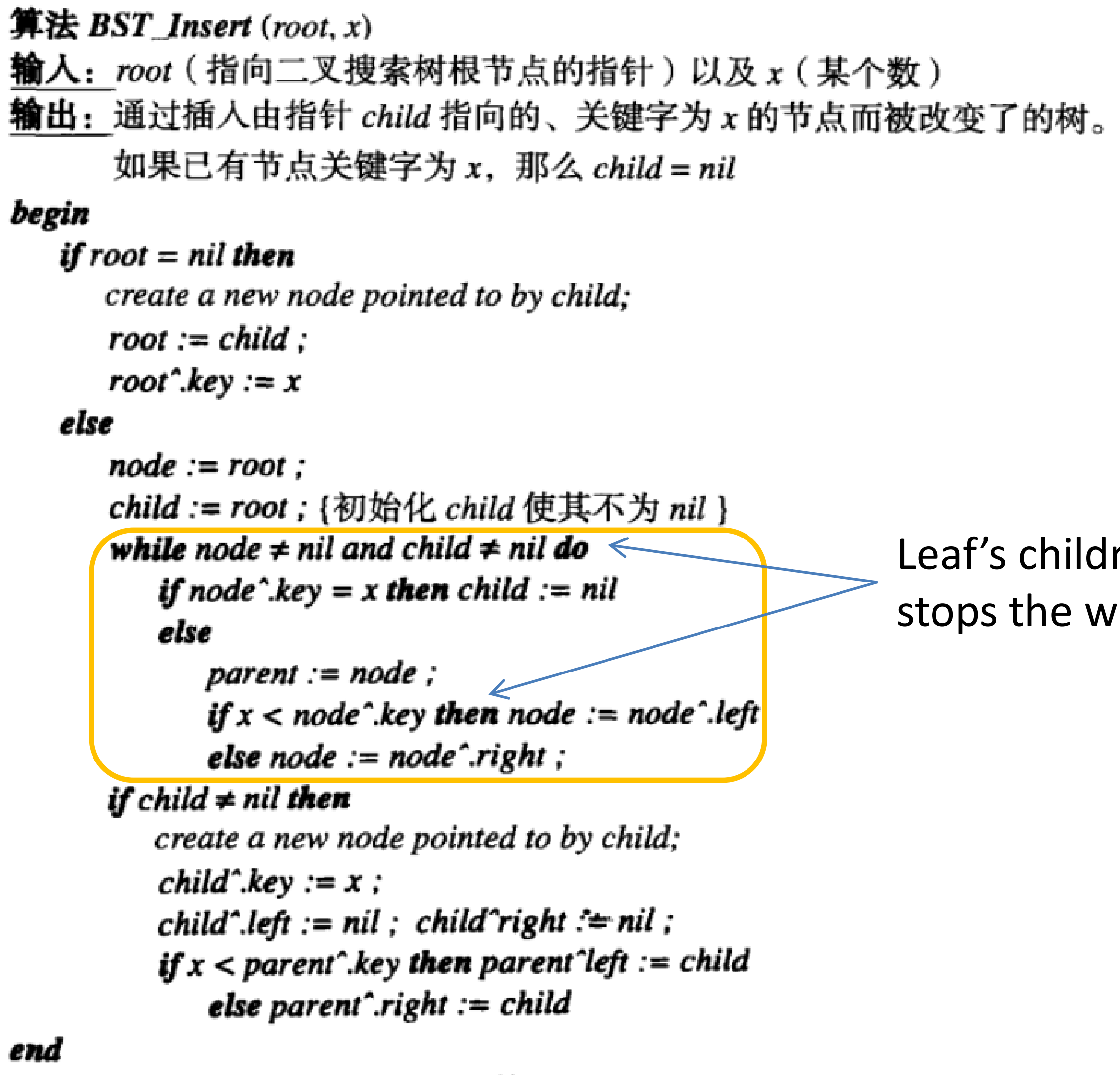
输出： *node*（指向含有关键字 *x* 的节点的指针，如果上述节点不存在，则指向 *nil*）

```
begin
    if root = nil or root^.key = x then node := root
    {root^是 root 的指针所指向的记录}
    else
        if x < root^.key then BST_Search(root^.left, x)
        else BST_Search(root^.right, x)
end
```

图 4.9 算法 *BST_Search*

BST insert

- Alwavs inserted on as leaf node.



算法 ***BST_Insert*** (*root*, *x*)

输入： *root*（指向二叉搜索树根节点的指针）以及 *x*（某个数）

输出： 通过插入由指针 *child* 指向的、关键字为 *x* 的节点而被改变了的树。
如果已有节点关键字为 *x*，那么 *child* = *nil*

```
begin
    if root = nil then
        create a new node pointed to by child;
        root := child ;
        root^.key := x
    else
        node := root ;
        child := root ; {初始化 child 使其不为 nil }
        while node ≠ nil and child ≠ nil do
            if node^.key = x then child := nil
            else
                parent := node ;
                if x < node^.key then node := node^.left
                else node := node^.right ;
        if child ≠ nil then
            create a new node pointed to by child;
            child^.key := x ;
            child^.left := nil ;  child^right := nil ;
            if x < parent^.key then parent^left := child
                else parent^.right := child
end
```

图 4.10 算法 *BST_Insert*

Leaf's children ==nil, stops the while

Sort

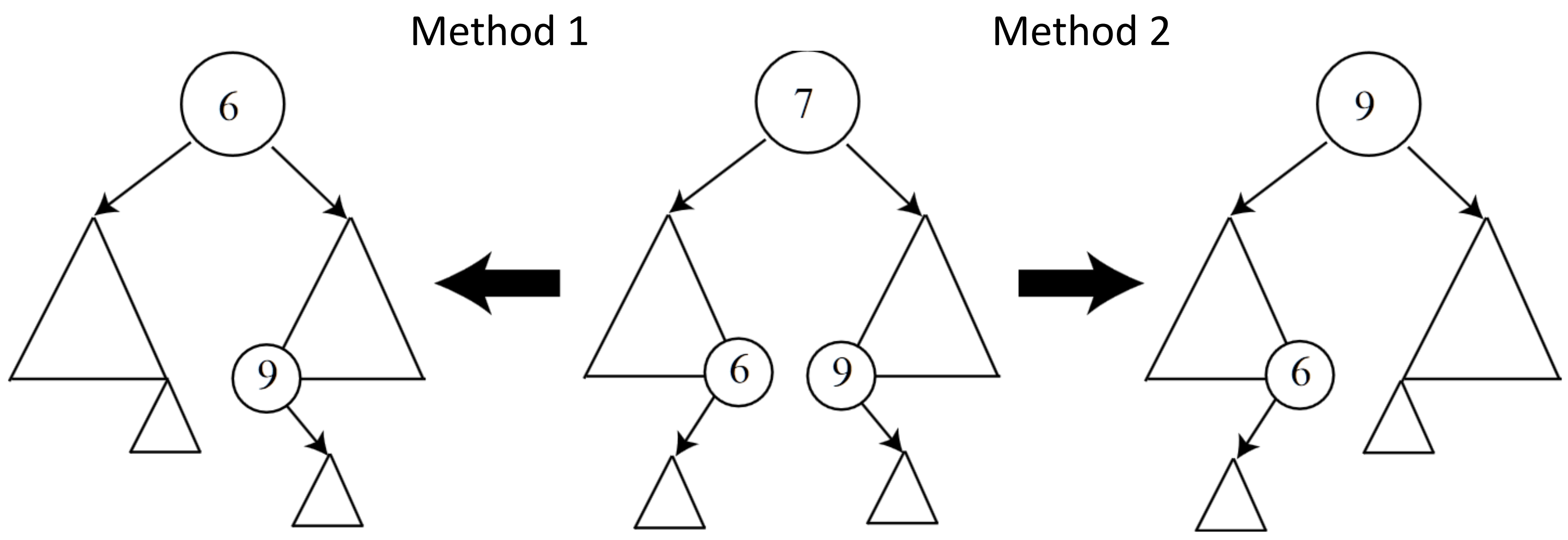
- Insertion, then traverse *in-order*

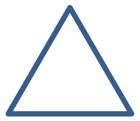
BST delete

Say delete node *p

- A. p has no children
  - simple deletion
- B. p has only one children
  - simple deletion and concatenantaion
- C. p has two children
  - two methods

C: p has 2 children




:Arbitrary size sub-tree, whose leftmost(9) and rightmost(6) are shown here.

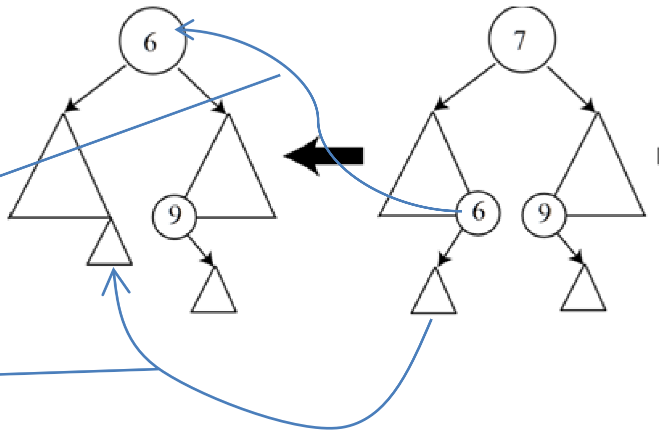
We need pop a node larger than whole left subtree,
or
Pop a node smaller than whole right subtree

算法 ***BST_Delete*** *(root, x)*
输入： *root*（指向二叉搜索树根节点的指针）以及 *x*（某个数）
输出： 如果存在关键字为 *x* 的节点，则将其删除从而改变这棵树

```
 {假设永远不会删除根节点，且任意两个节点都不相同。}
begin
    node := root ;
    while node ≠ nil and node^.key ≠ x do
        parent := node ;
        if x < node^.key then node := node^.left
        else node := node^.right ;
    if node = nil then print("x is not in the tree") ; halt ;
    if node ≠ root then
        if node^.left = nil then
            if x ≤ parent^.key then
                parent^.left := node^.right
            else parent^.right := node^.right
        else if node^.right = nil then
            if x ≤ parent^.key then
                parent^.left := node^.left
            else parent^.right := node^.left
        else {两个子节点的情况}
            node1 := node^.left ;
            parent1 := node ;
            while node1^.right ≠ nil do
                parent1 := node1 ;
                node1 := node1^.right ;
              {下面开始做真正的删除}
            parent1^.right := node1^.left ;
            node^.key := node1^.key
end
```

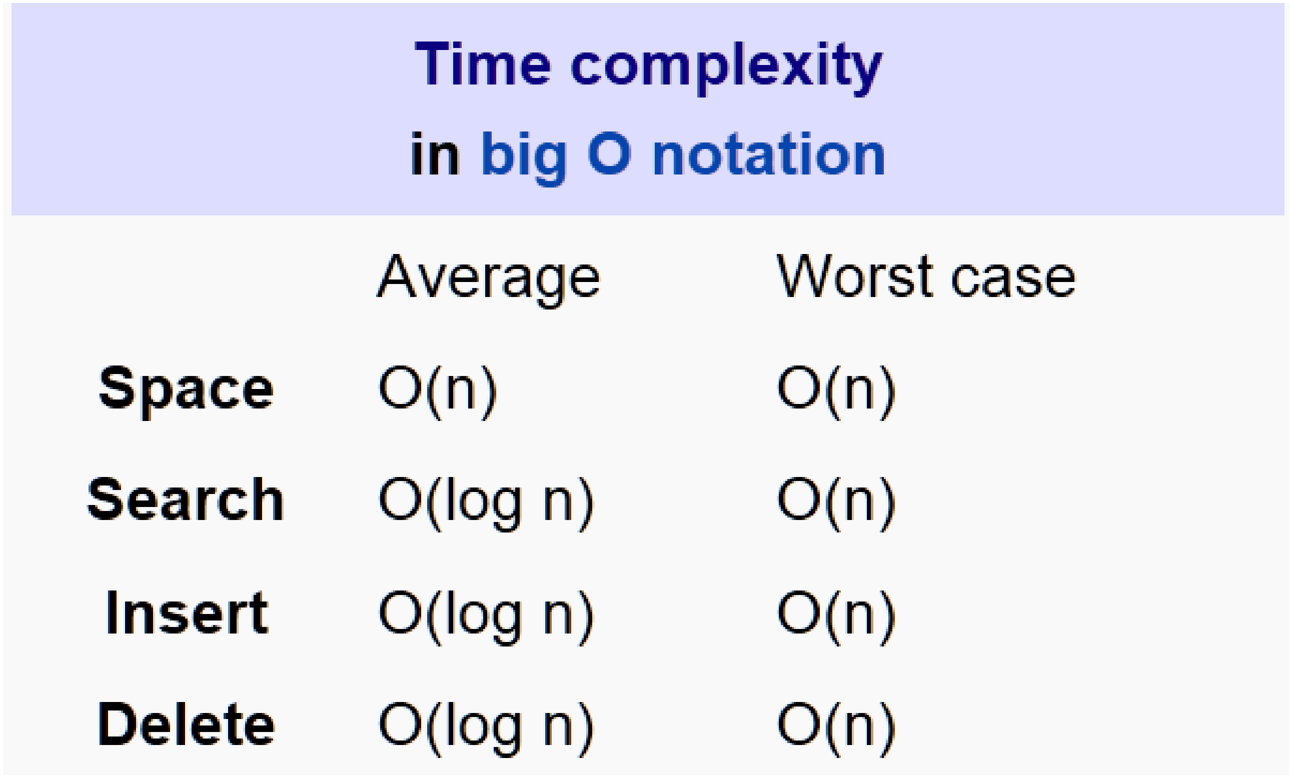
Method 1

找left subtree中最右下的节点(key最大者)



complexity

- Search, insertion and deletion



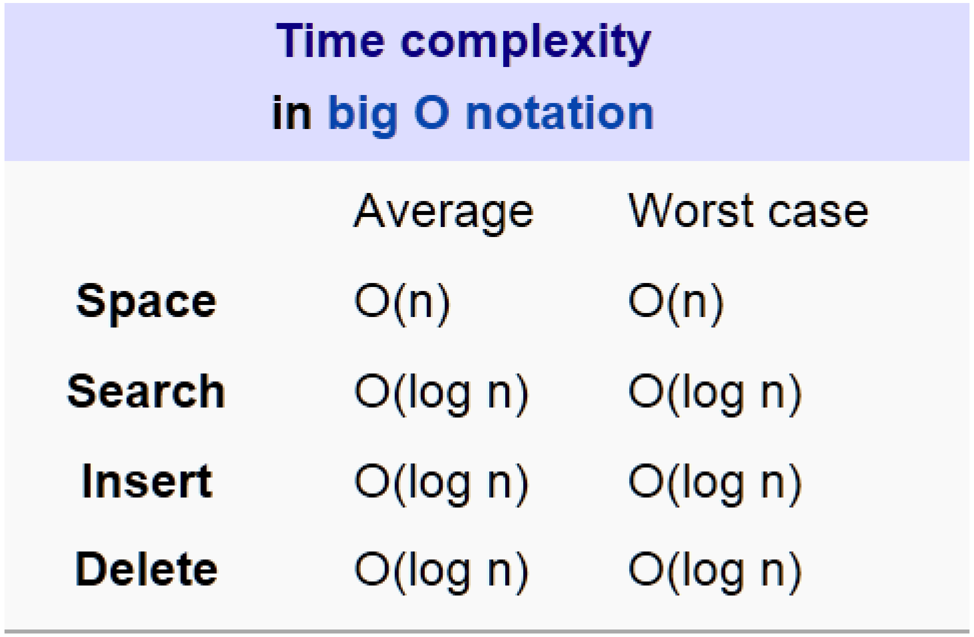
Time complexity in big O notation

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Depth of the tree depends on the balance of the tree.
Induction to **self-balancing tree**

AVL and Red-Black

- Self-balancing tree

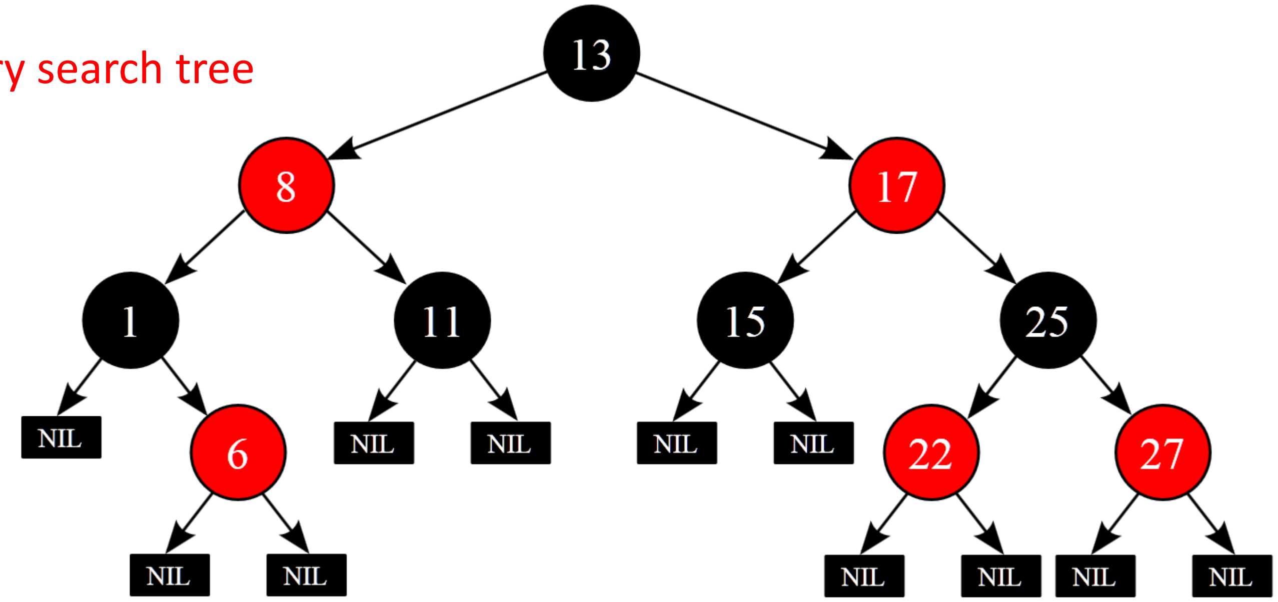


Time complexity in big O notation

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Red-Black tree for sorting(红黑树)

a type of binary search tree



Self-balancing tree algorithm: AVL and Red-Black tree; The key operation: rotating the node.