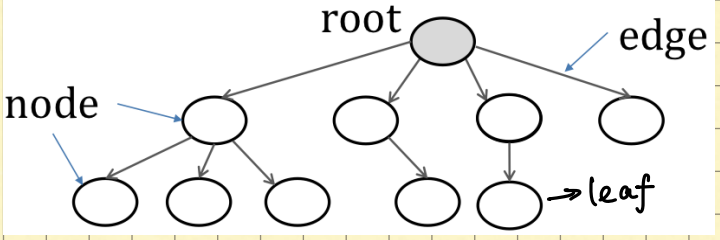


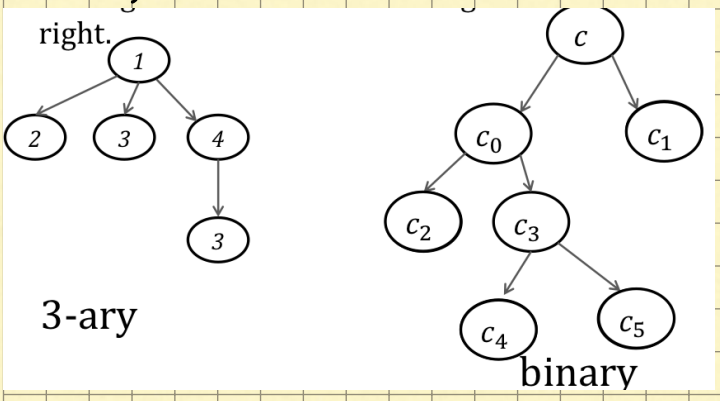
(parent)
↕
sibling ↔ sibling (child)

① n nodes → n-1 edges

② 若tree的所有internal node都至少有2个子结点，且leaf nodes = m，则 internal nodes 至多有m-1个.

tree中的每个node都是更小tree的root.
path：从root到每个node都有唯一的path.
depth：到root的edges数
level：有相同depth的nodes组成level.
height：最长的depth

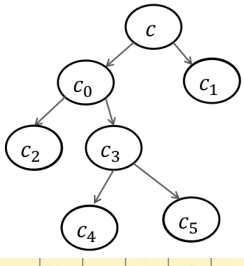
k-ary tree：internal tree 有着最多k个child nodes的rooted tree.



Binary tree:
(1) 空树，或
(2) 一个node有一些数据，其下有左、右子树.

Full Level：当前level之上的level中的每个node都有2个child node（0 level除外）

- Consider a binary tree with height h, its level l $(0 \le l \le h)$ is full if it contains 2^l nodes.

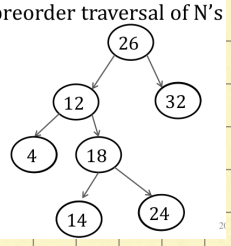


Complete Binary Tree：h height中有0~h-1 level是full的

③ 有n≥2节点的完全二叉树有高度O(logn).

Preorder Traversal

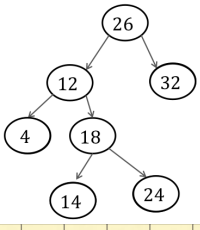
- Preorder traversal of the tree whose root is N
  - visit the root N
  - Recursively perform a preorder traversal of N's left subtree
  - Recursively perform a preorder traversal of N's right subtree
- Preorder traversal
  - 26, 12, 4, 18, 14, 24, 32




```
preorderiteratively(treeNode root):
1. treeNode stack s
2. s.push(root)
3. while(s!=empty)
4.     treeNode node= s.top()
5.     print(node)
6.     s.pop()
7.     if(node->right!=null)
8.         s.push(node->right)
9.     if(node->left!=null)
10.        s.push(node->left)
```

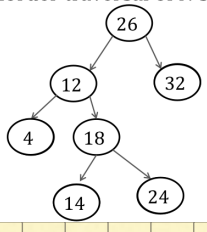
Postorder Traversal

- Postorder traversal of the tree whose root is N
  - Recursively perform a postorder traversal of N's left subtree
  - Recursively perform a postorder traversal of N's right subtree
  - visit the root N
- Postorder traversal
  - 4, 14, 24, 18, 12, 32, 26



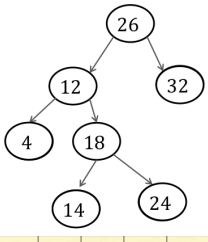
Inorder Traversal

- Inorder traversal of the tree whose root is N
  - Recursively perform a inorder traversal of N's left subtree
  - Visit the root N
  - Recursively perform a inorder traversal of N's right subtree
- Inorder traversal
  - 4, 12, 14, 18, 24, 26, 32



Level Traversal

- Visit the nodes one level at a time, from top to bottom, and left to right.
- Level-order of the tree:
  - 26, 12, 32, 4, 18, 14, 24
- How to implement?

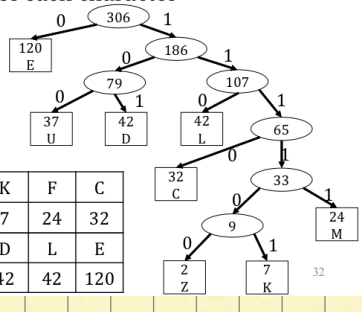


- Preorder: root, left subtree, right subtree
- Postoder: left subtree, right subtree, root
- Inorder: left subtree, root, right subtree
- Level-order: top to bottom, left to right

Huffman Encoding

Prefix (free) code: 没有任何的编码是另一个编码的前缀

- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file
- Example:
  - Leaf nodes are characters
  - 101 = 'D'
  - "000110" = "EEEL"



Z	K	F	C
2	7	24	32
U	D	L	E
37	42	42	120

- 1) Begin by reading through the text to determine the frequencies
- 2) Create a list of nodes that contain (character, frequency) pairs for each character that appears in text
- 3) Remove and "merge" the nodes with the two lowest frequencies, forming a new node that is their parent
- 4) Add the parent to the list of nodes
- 5) Repeat steps 3) and 4) until there is only a single node in the list, which will be the root of the Huffman tree.
- Example: build the Huffman tree for the following (character, frequency) pairs:

Z	K	F	C	U	D	L	E
2	7	10	12	27	30	43	65

Huffmann tree 的正确性

- Given a Huffman tree, it includes at least 2 nodes, assume node u and node v have the top-2 lowest frequencies, then
  - 1) node u and v have the same parent node
  - 2) depth(u) and depth(v) >= depth(x), where node x is any leaf node in the Huffman tree.
- Huffman encoding is the optimum prefix code, i.e., the space cost is minimized.

Priority Queue：

- A **priority queue** stores a set S of n integers and supports the following operations:
  - *Insert(e)*: adds a new integer to S
  - *Delete-min:* removes the smallest integer in S, and returns it.

By using binary heap to implement

- O(n) space consumption
- O(log n) insertion time
- O(log n) delete-min time

最小堆

- Let S be a set of n integers. A binary heap on S is a binary tree T satisfying:
  - (1) T is complete
  - (2) Every node u in T corresponds to a distinct integer in S, the integer is called the key of u (and is stored at u)
  - (3) If u is an internal node, the key of u is smaller than those of its child nodes

插入

- We perform insert(e) on a binary heap T as follows:
  - Step 1: Create a leaf node z with key e, while ensuring that T is a complete binary tree, it means there is only one place where z could be added.
  - Step 2: Set u ← z
  - Step 3: If u is the root, return.
  - Step 4: If the key of u > the key of its parent p, return
  - Step 5: Otherwise, swap the keys of u and p. Set u ← p, and repeat from Step 3.

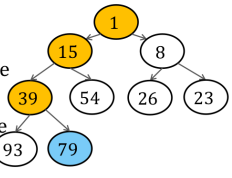
删除

- We perform delete-min on a binary heap T as follows:
  - Step 1: Report the key of the root
  - Step 2: Identify the rightmost leaf z at the bottom level of T
  - Step 3: Delete z, and store the key of z at the root
  - Step 4: Set u ← the root
  - Step 5: If u is leaf, return
  - Step 6: If the key of u < the keys of the children of u, return
  - Step 7: Otherwise, let v be the child of u with a smaller key Swap the keys of u and v. Set u ← v, and repeat from Step 5

如何找到 rightmost leaf. (O(logn))

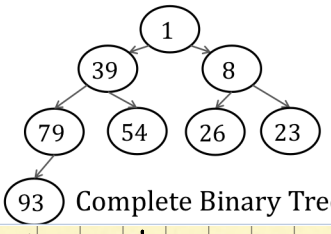
- Write the value n in binary form. We can do that in O(log n) time.
- Skip the most significant bit. We will scan the remaining bits from left to right, start from root,
  - If the bit is 0, we go to the left child of the current node
  - Otherwise, go to right child

- Here n = 9, binary form: 1001
- Skip the first bit '1'
- We scan the remaining bits
- Start from root node 1.
- The 2nd leftmost bit is 0, so we visit left, and go to node 15
- The 3rd leftmost bit is 0, so we visit left, and go to node 39
- The 4th leftmost bit is 1, so we turn right, and go to node 79 (done).



用二叉堆排序 O(nlogn) = build heap O(nlogn) + n×delete O(logn)

用数组实现完全二叉树



Complete Binary Tree

1	39	8	79	54	26	23	93

Storing in an array

- Put nodes at a higher level before those at a lower level
- Within the same level, order the nodes from left to right

假设数组下标从1开始

- Lemma: Suppose that node u of T is stored at A[i]. Then, the left child of u is stored at A[2i], and the right child at A[2i+1].

父
$A[\frac{i}{2}] \rightarrow A[i] \rightarrow \begin{cases} A[2i] & \text{左} \\ A[2i+1] & \text{右} \end{cases}$

- Corollary: Suppose that node u of T is stored at A[i]. Then, the parent of u is stored at A[⌊i/2⌋].
- Lemma: the rightmost leaf node at the bottom level is stored at A[n].

当整棵树都建起来之前，没法要实现 delete_min，因此，在建树时能更快.

Root-fix operation

- We are given a complete binary tree T with root r. It guaranteed that:
  - The left subtree of r is binary heap
  - The right subtree of r is a binary heap
  - However, the key of r may not be smaller than the keys of its children.
- We define the root-fix operation, it fixes the issue and makes T a binary heap.
- Root-fix can be done in O(log n) time – in the same manner as the delete-min algorithm (step 4 - 7)

先提供一个任意数组，
可用root-fix从底到顶将其转为二叉堆

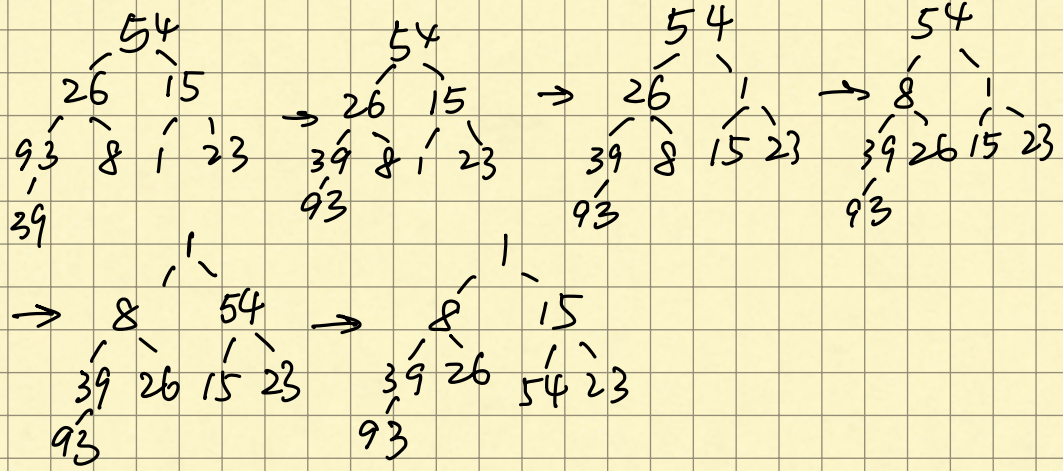
- For each i=n downto 1:
  - Perform root-fix on the subtree of T rooted at A[i]

每一个A都是完全二叉树
从最后一个节点开始做下沉操作.

54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
54	8	1	39	26	15	23	93

Root-fix

54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
54	8	1	39	26	15	23	93
1	8	15	39	26	54	23	93



root-fix 和 insertion 建堆是不一样的

Binary Search Tree (BST)

Balanced BST (平衡二叉查找树)

- A predecessor query: give an integer q, find its predecessor in S, which is the largest integer in S that does not exceed q.
- Insertion: adds a new integer to S
- Deletion: removes a integer from S

BST中的复杂度

- O(n) space consumption
- O(h) time per predecessor query (hence, also per dictionary lookup)
- O(h) time per insertion
- O(h) time per deletion
- where n = |S|, h is the height of BST, Note that all the above complexities hold in the worst case.

BST

- A BST on a set S of n integers in a binary tree T satisfying all the following requirements:
  - T has n nodes
  - Each node u in T stores a distinct integer in S, which is called the key of u
  - For every internal u, it holds that:
    - The key of u is larger than all the keys in the left subtree of u.
    - The key of u is smaller than all the keys in the right subtree of u.

BST不一定是完全二叉树

Predecessor Query

- Suppose that we have created a BST T on a set S of n integers. A predecessor query with search value q can be answered by descending a single root-to-leaf path:
  - (1) Set p ← −∞ (p will contain the final answer at the end)
  - (2) Set u ← the root of T
  - (3) If u = nil, then return p
  - (4) If key of u = q, the set p to q, and return p
  - (5) If key of u > q, then set u to the left child (now u = nil if there is no left child), and repeat from Step (3)
  - (6) Otherwise, set p to the key of u and u to the right child, and repeat from Step (3)

每一次访问节点都是O(1)
故若BST深h时，最坏O(h).

找小于q的最大值

Successor Query

- The successors of n integer q in S is the smallest integer in S that is no smaller than q.

找大于q的最小值

与Predecessor Query类似.

BST Insertion

- Suppose that we need to insert a new integer e. First create a new leaf z storing the key e. This can be done by descending a root-to-leaf path:
  - 1. Set u ← the root of T
  - 2. If e < the key of u
    - 2.1 If u has a left child, then set u to the left child
    - 2.2 Otherwise, make z the left child of u, and done
  - 3. Otherwise:
    - 3.1 If u has a right child, then set u to the right child
    - 3.2 Otherwise, make z the right child of u, and done.
  - Repeat from Step 2.
- The total cost is proportional to the height of T, i.e., O(h)

Red-black tree

Splay tree

AVL tree

BST Deletion

- Suppose that we want to delete an integer e. Frist, find the node u whose key equals to e in O(h) time (through a predecessor query).
- Case 1: if u is a leaf node, simply remove it from T.
- Case 2: if u has a right subtree:
  - Find the node v storing the successor s of e.
  - Set the key of u to s
  - Case 2.1: if v is a leaf node, them remove it from T
  - Case 2.2: otherwise, it must hold that v has a right child w, but not left child. Replace node v by subtree which rooted at w.
- Case 3: if u has no right subtree:
  - It must hold that u has a left child v, Replace node u by the subtree rooted at v.

先通过 predecessor / successor query 找到要删的结点.
若该结点为叶子，就直接删.
若该结点为内结点，由 predecessor / successor 的性质知，若有右子树/左子树，则(必无)左子树/右子树

- In all above cases, we have essentially descended a root-to-leaf path (call it deletion path), and removed a leaf node.
- The cost so far is O(h), recall that the successor of an integer can be found in O(h) time.

BBST

- A binary tree T is balanced if the following holds on every internal node u of T:
  - The height of the left subtree of u differs from that the right subtree of u by at most 1.

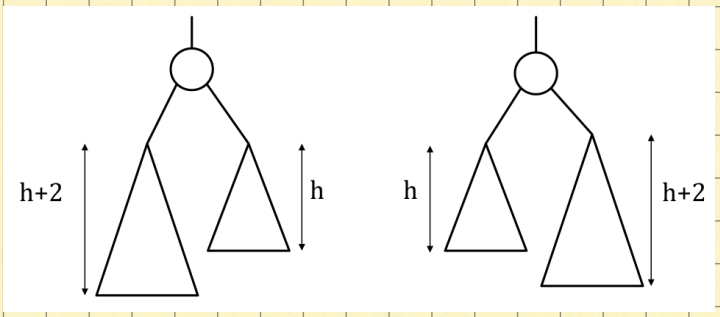
$$f(root, n) = \max(f(left, n-1), f(right, n-1)) + 1$$

- Theorem: a balanced binary tree with n nodes has height O(log n).

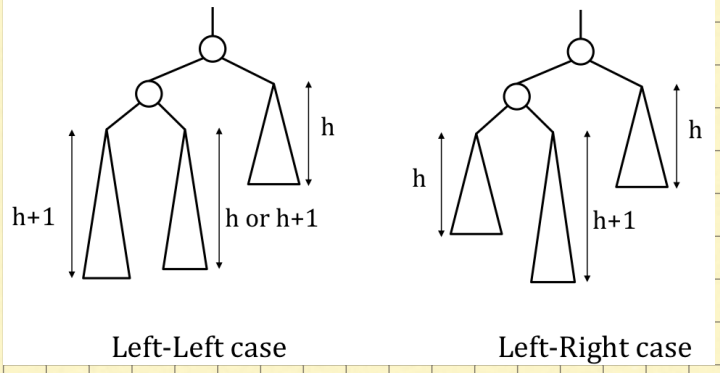
AVL-tree

- An AVL-tree on a set S of n integers is a balanced binary search tree T, where the following hold on every internal node u
  - u stores the heights of its left and right subtrees.

2-level imbalance



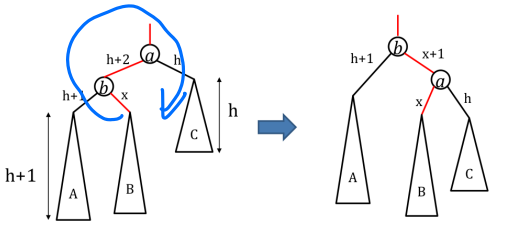
仅讨论左子树高2的情况，右子树高2对称.



Left-Left case　　Left-Right case

Rebalance Left-Left

- By a rotation:

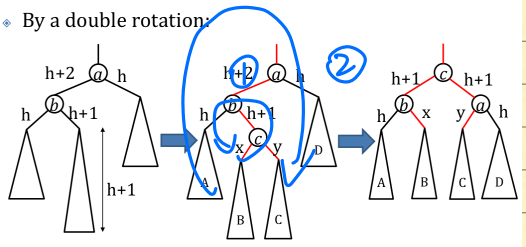


- Only 3 pointers to change (the red ones). The cost is O(1).
- Recall that x = h or h+1

65

Rebalance Left-Right

- By a double rotation:



- Only 5 pointers to change (see above). Hence, the cost is O(1).
- Note that x and y must be h or h-1. Furthermore, at least one of them must be h (why?)

67

- Insertion time analysis
  - It will be left as an exercise for you to prove
    - Only 2-level imbalance can occur in an insertion
    - Once we have remedied the lowest imbalance node, all the nodes in the tree will become balanced again
  - Thus, we can conclude the insertion cost in a balanced BST is O(log n), why?
- Deletion time analysis
  - It will be left as an exercise for you to prove
    - Only 2-level imbalance can occur after a deletion
  - Thus, we can conclude the deletion cost in a balanced BST is O(log n)

- O(n) space consumption
- O(log n) time per predecessor query (hence, also per dictionary lookup)
- O(log n) time per insertion
- O(log n) time per deletion