

Advanced Data Structure

Advanced Data Structure

- 1. 1 Trees
 - 1.1 AVL trees 平衡树
 - 1.1.1 Insert
 - 1.1.2 单旋转
 - 1.1.3 双旋转
 - 1.1.4 Questions
 - 1.2 Splay trees 伸展树
 - 1.3 Amortized Analysis 平摊分析
 - 1.3.1 Aggregate for analysis 合计法
 - 1.3.2 Accounting method 会计法
 - 1.3.3 Potential method 势能法
 - 1.4 Red-Black Trees 红黑树
 - 1.4.1 Insert 插入
 - 1.4.2 Delete 删除
 - 1.5 B+ trees
 - 1.5.1 Find
 - 1.5.2 Insert
 - 1.5.3 Delete
 - 1.6 Questions
- 2. 2 Inverted file index 倒排文件索引
 - 2.1 structure
 - 2.2 modules
 - 2.3 topic
 - 2.4 Measure
 - 2.5 Questions
- 3. Heap
 - 3.1 Lefttest Heap 左式堆
 - 3.1.1 Merge
 - 3.1.1.1 recursive
 - 3.1.1.2 iterative
 - 3.1.2 DeleteMin
 - 3.2 Skew Heaps 斜堆
 - 3.2.1 Merge
 - 3.2.2 Insert
 - 3.2.3 平摊分析
 - 3.3 Questions
- 4. Binomial Queue
 - 4.1 FindMin
 - 4.2 Merge
 - 4.3 Insert
 - 4.4 DeleteMin
 - 4.5 Analysis
 - 4.5.1 Aggregate
 - 4.5.2 Potential
 - 4.6 Questions
- 5. Backtracking
 - 5.1 8-Queen problem
 - 5.2 Turnpike problem
 - 5.3 Game
 - 5.4 Questions
- 6. Divide and Conquer

- 6.1 closest points
- 6.2 三种方法解决递归
- 6.3 Questions
- 7. Dynamic programming
 - 7.1 Fibonacci number
 - 7.2 Matrix Multiplication
 - 7.3 Optimal binary search tree
 - 7.4 Floyd-warshall算法
- 8. Greedy Algorithms
 - 8.1 Introduction
 - 8.2 Activity selection
 - 8.2.1 A DP problem
 - 8.2.2 DP solution
 - 8.3 Huffman code
 - 8.4 Questions
- 9. NP-completeness
 - 9.1 NP definition
 - 9.2 NPC problems
 - 9.2.1 Proof
 - 9.3 Formal language framework
 - 9.4 Questions
- 10. Approximation algorithm
 - 10.1 Introduction
 - 10.2 Approximate bin packing
 - 10.3 Knapsack problem
 - 10.4 10.4 K-center
 - 10.5 Questions
- 11. Local search
 - 11.1 basic
 - 11.2 Vertex cover
 - 11.3 Hopfield neural
 - 11.4 Maximum Cut problem
 - 11.5 Questions
- 12. Randomize
 - 12.1 Introduction
 - 12.2 Hiring problems
 - 12.3 Randomized quick sort
- 13. Parallel Algorithm
 - 13.1 Introduction
 - 13.1.1 Parallel Random Access Machine(PRAM)
 - 13.1.2 Work-Depth(WD)
 - 13.2 Prefix-sums
 - 13.3 Merging
 - 13.3.1 Parallel ranking
 - 13.4 Maximum finding
 - 13.5 Questions
- 14. External Sorting
 - 14.1 Introduction
 - 14.1.1 Example
 - 14.2 Pass reduction
 - 14.2.1 Polyphase merge
 - 14.3 Buffer handling
 - 14.4 Run generation and merge

1. 1 Trees

1.1 AVL trees 平衡树

平衡的二叉搜索树，目的是使动态搜索更快

所有操作的时间复杂度为： $O(\log N)$ ，即高度 h

在高度为 h 的 AVL 树中，最少节点数 $n_h \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1$ ，其中 $h = -1, n_h = 0$ (空树的高度定义为-1)

$$n_h = n_{h-1} + n_{h-2} + 1$$

h/depth	n_h 最少节点数
0	1
1	2
2	4
3	7
4	12
5	20
6	33

Table 1

【Definition】The balance factor $BF(node) = h_L - h_R$. In an AVL tree, $BF(node) = -1, 0$, or 1 .

在 AVL 树中，不可能出现一个节点及其两个子节点的平衡因子都为 -1 的情况

- In an AVL tree, it is possible to have this situation that the balance factors of a node and both of its children are all +1.

1.1.1 Insert

先插入到正确的位置上 bst，从下到上更新 BF，有问题做旋转

1.1.2 单旋转

1.1.3 双旋转

left-right 是 zjg-zag 双旋转

1.1.4 Questions

Delete a node v from an AVL tree T_1 , we can obtain another AVL tree T_2 . Then insert v into T_2 , we can obtain another AVL tree T_3 . Which one(s) of the following statements about T_1 and T_3 is(are) true?

从一个 AVL 树 T_1 中删除一个节点 v ，就可以得到另一个 AVL 树 T_2 。然后将 v 插入 T_2 ，我们可以得到另一个 AVL 树 T_3 。下列关于 T_1 和 T_3 的陈述哪一个是正确的？

- I、 If v is a leaf node in T_1 , then T_1 and T_3 might be different.

如果 v 是 T_1 的叶节点，那么 T_1 和 T_3 可能是不同的。

- II、 If v is not a leaf node in T_1 , then T_1 and T_3 must be different.
- III、 If v is not a leaf node in T_1 , then T_1 and T_3 must be the same.

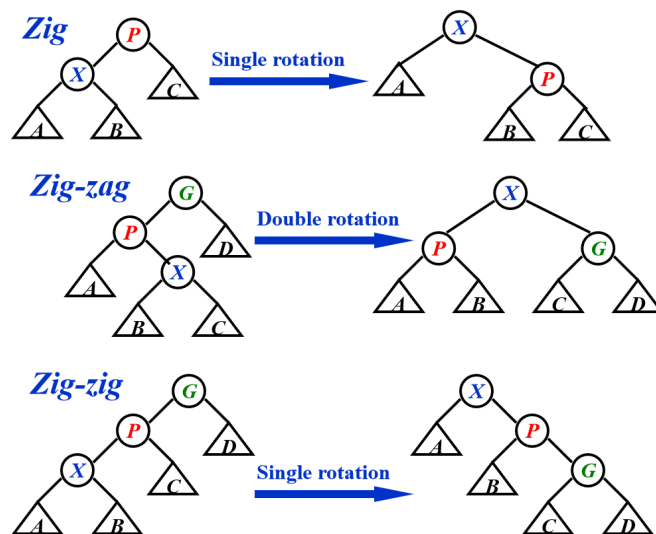
- ☒ A. I only
- ☐ B. II only
- ☐ C. I and II only
- ☐ D. I and III only

1.2 Splay trees 伸展树

性能类 AVL 树，但不关心是否平衡

每访问一个节点 X ，从这个节点向上对树进行一系列旋转，使这个节点变成根

如果 X 的父节点是根，单旋转 X 和树根；如果 X 有父亲 P 和祖父 G ，双旋转 (zigzig 或 zigzag)



All of the Zig, Zig-zig, and Zig-zag rotations not only move the accessed node to the root, but also roughly half the depth of most nodes **on the path**.

均摊时间复杂度 $T = O(\log N)$

1.3 Amortized Analysis 平摊分析

确保 amortized bound 是实际 cost 的上界，即 $\text{worst case bound} \geq \text{amortized bound} \geq \text{actual cost}$

1.3.1 Aggregate for analysis 合计法

假设每个操作的平摊时间相同

- 只有简单的算法可以这样计算

1.3.2 Accounting method 会计法

类似银行存钱

平摊 amortized cost \geq 实际 actual cost

平摊=实际+信用 credit

定义: credit=摊还开销-实际开销

- 每个操作的平摊时间可能不同
- 困难是要猜对每个操作的 credit, 所以引入第三种方法

1.3.3 Potential method 势能法

定义: $\hat{c}_i - c_i = \text{Credit}_i = \Phi(D_i) - \Phi(D_{i-1})$

D_{i-1} 是在本次操作前的结构; D_i 是本次操作后的结构

potential function 将当前问题的结构映射成一个数字——结构的势能

credit = 两个势能函数 potential function 的值的差——这个操作把问题的结构改变了多少

- 困难是要找到一个好的势能函数
- In amortized analysis, a good potential function should always assume **its minimum at the start of the sequence.**

对正整数 a, b, c, 如果 $a + b \leq c$, 那么 $\lg a + \lg b \leq \lg c - 2$

1.4 Red-Black Trees 红黑树

平衡的二叉搜索树, 目的是使动态搜索更快

定义:

1. Root property: The root is black.
2. External property: Every leaf (Leaf is a NULL child of a node) is black in the Red-Black tree.
3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.
4. Depth property: All the leaves have the same black depth.
5. Path property: Every simple path from root to descendant leaf node contains the same number of black nodes.

The result of all these above-mentioned properties is that the Red-Black tree is *roughly balanced*.

一棵有 N 个内部节点的红黑树, 其高度最多为 $2\lg(N + 1)$ 可由以下两个命题得证

- 对于任何一个节点 x, $\text{sizeof}(x) \geq 2^{\text{blackheight}(x)} - 1$, 即 the number of internal nodes in the subtree rooted at x is more than $\text{bh}(x)$ 子树内点数量大于等于 $2^{\text{bh}(x)} - 1$
- $\text{blackheight}(T) \geq h(T)/2$

Number of <i>rotations</i>		
	AVL	Red-Black Tree
Insertion	≤ 2	≤ 2
Deletion	$O(\lg N)$	≤ 3

2 和 3 是旋转次数

- In a red-black tree, an internal red node cannot be a node of degree 1

1.4.1 Insert 插入

插入，检查颜色，旋转

可以循环实现

1.4.2 Delete 删除

- 删除叶子
- 删除 1 度的节点：用孩子代替
- 删除 2 度的节点：用左子树的最大或右子树的最小代替（保持颜色不变）；删除子树的代替节点

总之，必须在被替代（删除）的路径上加一个黑节点 或者 整棵树的黑高-1 rebalance，不需要旋转

the number of rotations in the DELETE operation is $O(1)$

运行时间： $O(\log N)$ 对比 AVL，红黑树删除有优势

被删除的节点 x 为黑色（x 为红色可以直接删掉）

case 1: x 有个红色的兄弟 w

交换兄弟和它父亲的颜色；对兄弟 w 和父亲 p 进行一次旋转，转化为以下三种情况

case 2: x 的兄弟 w 是黑色，且 w 的两个子节点都是黑色

将兄弟变红色；将 x 上移（到它原本的父亲处）；检查如果 x 变成根，则表明虽然没有在这条路径上加一个黑节点，但是成功从其他所有路径上移除了一个黑节点，可以删除原来 x；循环

case 3: x 的兄弟 w 是黑色，且左孩子是红色，右孩子是黑色

交换兄弟和它左孩子的颜色，两者进行一次旋转；转化为情况 4

case 4: x 的兄弟 w 是黑色，且右孩子是红色

将兄弟变红，它的孩子和父亲变黑（或其他颜色更改）；旋转父亲 p 和兄弟 w

1.5 B+ trees

一种**查找树**，类似红黑树，不是二叉树，平衡，自下而上构建而成

一棵 m 阶的 B+ 树定义如下：

- (1) 每个结点至多有 m 个子女；
- (2) 除根结点外，每个结点至少有 $\lceil m/2 \rceil$ 个子女，根结点至少有两个子女；
- (3) 有 k 个子女的结点必有 k 个关键字。

- B 树的高度仅随它所包含的节点数按对数增长

M 阶 B+ 树，总节点数 N, $Depth(M, N) = O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$, 通常 M=3 or 4

2-3 tree 各节点度数为 2 或 3

- 叶子结点存储真实值；中间结点有 M 个指向叶子结点的指针，以及 M-1 个叶节点中最小值在中间

1.5.1 Find

B+ 树的查找与 B 树不同，当索引部分某个结点的关键字与所查的关键字相等时，并不停止查找，应继续沿着这个关键字左边的指针向下，一直查到该关键字所在的叶子结点为止。

$$T_{find}(M, N) = O(\log N)$$

和是M阶B+树的阶数无关

1.5.2 Insert

分裂

1.5.3 Delete

insert 的反操作，结点过少时和兄弟合并，递归向上检查

1.6 Questions

Consider the following buffer management problem. Initially the buffer size (the number of blocks) is one. Each block can accommodate exactly one item. As soon as a new item arrives, check if there is an available block. If yes, put the item into the block, induced a cost of one. Otherwise, the buffer size is doubled, and then the item is able to put into. Moreover, the old items have to be moved into the new buffer so it costs $k+1$ to make this insertion, where k is the number of old items. Clearly, if there are N items, the worst-case cost for one insertion can be $\Omega(N)$. To show that the average cost is $O(1)$, let us turn to the amortized analysis. To simplify the problem, assume that the buffer is full after all the N items are placed. Which of the following potential functions works?

- A. The number of items currently in the buffer
- B. The opposite number of items currently in the buffer
- C. The number of available blocks currently in the buffer
- D. The opposite number of available blocks in the buffer

题目解析：设 $size_i$ 为第 i 次插入前 buffer 的大小。 $c^i = c_i + \phi_i - \phi_{i-1}$ 。如果插入前 buffer 没满， $c_i=1$ ，否则 $c_i=size_i+1$ 。A: 如果插入前 buffer 没满，
 $c^i = c_i + \phi_i - \phi_{i-1} = 1 + (size_i + 1) - size_i = 2$ 。如果插入前 buffer 满，
 $c^i = c_i + \phi_i - \phi_{i-1} = size_i + 1 + (size_i + 1) - size_i = size_i + 2$ 。B: 同理，两种情况 $\phi_i - \phi_{i-1}$ 要么是 1 要么是 -1，而 c_i 却和当前 buffer 大小有关， c^i 肯定不是常数。C: 如果插入前 buffer 没满， $\phi_i - \phi_{i-1} = -1$ ， $c^i = c_i + \phi_i - \phi_{i-1} = 1 + (-1) = 0$ 。
如果插入前 buffer 满， $\phi_i - \phi_{i-1} = (size_i - 1) - 0 = size_i - 1$ ，
 $c^i = size_i + 1 + \phi_i - \phi_{i-1} = 2size_i$ 。D: 如果插入前 buffer 没满，
 $\phi_i - \phi_{i-1} = 1$ ， $c^i = c_i + \phi_i - \phi_{i-1} = 1 + 1 = 2$ 。如果插入前 buffer 满，
 $\phi_i - \phi_{i-1} = 1 - size_i$ ， $c^i = size_i + 1 + \phi_i - \phi_{i-1} = 2$ 。

[题集][Lecture2.Red-Black Trees and B+ Trees](#) [a 2-3 tree with 3 nonleaf nodes must have 18 keys - CSDN博客](#)

[【PTA】](#) [【数据结构与算法】B-树和B+树 - 代码天地\(codetd.com\)](#)

2. 2 Inverted file index 倒排文件索引

反向索引 哪些书里有这个词；正向，这本书里有哪些词

2.1 structure

链表指针连接

搜索引擎

posting list: 所有包含特定term文档的id的集合，需要从词典映射到这个集合，即 Inverted file contains a list of pointers to all occurrences of a term in the text.

2.2 modules

index generator 包括 token analyzer stop filter, vocabulary scanner, vocabulary insertor, memory management

1. **word stemming** 提取词干 and stop filter 将单词的不同形式都记为原型
2. 常见的词为 **stop words**, 将它们剔除出原来的文件, 但还是有它们的 posting list
3. What are the pros and cons of using *hashing*, compared to *search trees*? Faster for a single word but expansive for a range of search.

2.3 topic

Distributed indexing: each node contains a **subset** of a collection 集合的部分

Dynamic indexing: 新的文件先存到附加序号集中, 等到用户搜索时查询 main index 以及 auxiliary index

Compression 压缩:

Thresholding 阈值: for documents, 返回前 k 个权重最大的文件; for query, 查询单词频率最低的一些文件

2.4 Measure

$$precision = R_R / (R_R + I_R)$$

$$recall = R_R / (R_R + R_N)$$

2.5 Questions

1. While accessing a term stored in a B+ tree in an inverted file index, range searches are expensive. **F**

[题集][Lecture3. Inverted File Index in distributed indexing, document-partitioned str](#)-CSDN博客

3. Heap

3.1 Lefttest Heap 左式堆

用建堆的方法**合并**两个堆的时间复杂度是 $\Theta(N)$; 用指针的方式会使其他操作 (find) 变慢

需要频繁的 Merge 时使用左式堆

性质: 有序性 (和普通堆一样); 结构不平衡的二叉树

定义: $Npl(x)$ **null path length** 任意一个节点 x 的 Npl 是从 x 到一个没有两个孩子的节点的最短路径长; 令 $Npl(NULL) = -1$

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$
 从下往上算 Npl

The leftist heap property is that for every node X in the heap, the **null path length** of the left child is **at least** as large as that of the right child.

定理:

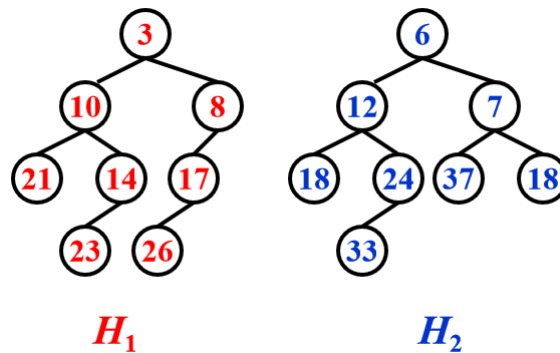
- 右路径上有 k 个节点的左式树至少有总共 $2^k - 1$ 个节点; 即有 N 个节点的左式树, 它的右路径上最多有 $\lfloor \log(N + 1) \rfloor$ 个节点

我们可以把所有工作都放在右路径上, 因为它更短

- A leftist heap with the null path length of the root being r must have at least $2^{r+1} - 1$ nodes

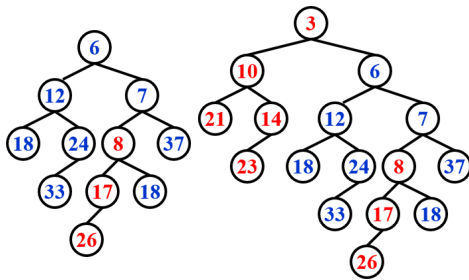
3.1.1 Merge

3.1.1.1 recursive



step1 Merge($H_1 \rightarrow \text{Right}, H_2$); step2 Attach($H_2, H_1 \rightarrow \text{Right}$)

大的根值的堆 和 小的根值的堆的右子堆 合并



step3 Swap($H_1 \rightarrow \text{Right}, H_1 \rightarrow \text{Left}$) if necessary

要求 h_1 的 root < h_2 的 root

$$T = O(\log N)$$

3.1.1.2 iterative

step1 对两棵树的右路径进行排序，不改变他们的左孩子

step2 交换孩子（如果有必要）

3.1.2 DeleteMin

step1: delete min

step2: merge 2 subtrees

3.2 Skew Heaps 斜堆

左式堆的自调节形式，目的是使任意 M 种操作花费最多 $O(M \log N)$ 的时间

结构不平衡的二叉树，斜堆的右路径可以是任意长度（左右旋转）

3.2.1 Merge

总是交换左右孩子 直到包含右路径的最大节点的根没有孩子可以交换 No Npl

merge(h_1, h_2) 摊还运行时间: $O(\log N)$

1. Let h_1 and h_2 be the two min skew heaps to be merged. Let h_1 's root be smaller than h_2 's root (If not smaller, we can swap to get the same).

2. We swap $h_1 \rightarrow \text{left}$ and $h_1 \rightarrow \text{right}$. 交换小根左右堆

3. `h1->left = merge(h2, h1->left)` 将大根和小根的左堆合并

3.2.2 Insert

保持最小的节点，交换（左树变到右树），再连另外一棵树

优势：不需要额外空间去保存路径长度，不需要测试来确保什么时候要交换孩子

- A perfectly balanced tree forms if keys 1 to 2^k-1 are inserted in order into an initially empty skew heap.

3.2.3 平摊分析

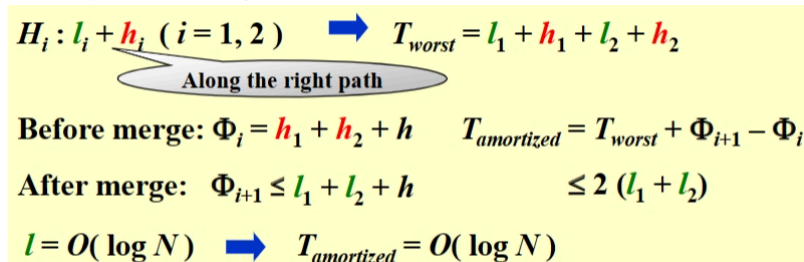
$$T_{amortized} = O(\log N)$$

$\phi(D_i)$ = number of heavy nodes

【Definition】A node p is *heavy* if the number of descendants of p 's *right* subtree is **at least half** of the number of descendants of p , and light otherwise. Note that the number of descendants of a node includes the node itself.

只有在原来右路径上的节点 heavy/light 状态会改变

h -heavy nodes; l -light nodes



$H_i : l_i + h_i \ (i = 1, 2) \rightarrow T_{worst} = l_1 + h_1 + l_2 + h_2$

Along the right path

Before merge: $\Phi_i = h_1 + h_2 + h$ $T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i$

After merge: $\Phi_{i+1} \leq l_1 + l_2 + h \leq 2(l_1 + l_2)$

$l = O(\log N) \rightarrow T_{amortized} = O(\log N)$

为什么 light node 的复杂度是 $\log(N)$ ：每一个 light node 的右子树 node 数量都要减半 ($\leq (n-1)/2$)

3.3 Questions

[题集][Lecture 4. Leftist Heaps and Skew Heaps when doing amortized analysis, which one of the fo-CSDN博客](#)

[Assignment 4 2-2 we can perform buildheap for leftist heaps by cons-CSDN博客](#)

4. Binomial Queue

一种优先队列，heap-ordered 树的集合=array of binomial trees

一棵 B_k 二项式树有 k 个孩子，即 B_0, \dots, B_{k-1} ， B_k 有共 2^k 个节点，每个节点的深度是 C_k^d

一个不管多少大小的优先队列，可以被一组二项树唯一地表示，eg: $13 = 1101_2$ ，相当于一个 B_1 ，一个 B_2 ，一个 B_3

4.1 FindMin

遍历所有根

对共有 N 个节点的二项队列，最多有 $\lfloor \log N \rfloor$ 个根，因此时间复杂度为 $O(\log N)$ ，平均时间是常数

4.2 Merge

二进制加法算队列，再合并；确保根是最小数

二项树队列必须高度有序，而不是根大小有序

摊还运行时间： $O(\log N)$

4.3 Insert

一种特殊的 merge

将 n 个节点插入一个空二项队列，最坏时间复杂度为 $O(N)$

若最小还没出现的二项式树是 B_i ，那么时间复杂度为 $Const(i + 1)$ ，即平均时间是常数

摊还运行时间： $O(1)$

4.4 DeleteMin

- 1. FindMin in B_k // $O(\log N)$
- 2. 将 B_k 从 H 中移走 // $O(1)$
- 3. 将根从 B_k 中移走 // $O(\log N)$
- 4. Merge(H' , H'') // $O(\log N)$

摊还运行时间： $O(\log N)$

删除要将原队列一分为二进行合并

4.5 Analysis

Operation	Property	Solution
DeleteMin	Find all the subtrees quickly	Left-child-next-sibling with linked lists
Merge	The children are ordered by their sizes	The new tree will be the largest. Hence maintain the subtrees in decreasing sizes

- 一个 N 个元素的二项队列可以由 N 次连续的插入在 $O(N)$ 时间内实现（最坏情况 $O(N)$ ）

证明：

4.5.1 Aggregate

4.5.2 Potential

$T_{worst} = O(\log N)$

$T_{amortized} = 2$

4.6 Questions

1. For a binomial queue, __ take(s) a constant time on average. **C**

A. merging and delete-min B. find-min and delete-min C. find-min and insertion D. merging and insertion

二项队列找到最小和插入的平均时间是常数

2. The potential function Q of a binomial queue is the number of the trees. After merging two binomial queues H_1 with 22 nodes and H_2 with 13 nodes, what is the potential change $Q_{H_1+H_2} - (Q_{H_1} + Q_{H_2})$?

A.-3 B.0 C.-2 D.2

A

$$H_1 = 16 + 4 + 2$$

$$H_2 = 8 + 4 + 1$$

$$H_1 + H_2 = 35 = 32 + 2 + 1$$

$$\text{势能变化} = 3 - 3 - 3 = -3$$

5. Backtracking

回溯：分步解决问题，如果此路不通，就回到之前的点走另一条路。节约穷举时间

5.1 8-Queen problem

1. 建一个游戏树（不是真的在program里），路径个数为叶子节点个数
2. 对所有路径做深度优先搜索（post-order traversal）

5.2 Turnpike problem

N points有 $N(N-1)/2$ 个距离，第一个点 $x_1 = 0$ ，要求给定距离，找到点集

1. 根据距离个数算 N
2. 得到最小点和最大点
3. 找到次大的距离并检查

5.3 Game

tik-tack-toe: Minimax Strategy

$f(P) = W_{\text{computer}} - W_{\text{human}}$ ，函数值越小，人类胜利可能性越大

α - β pruning: 将搜索的时间复杂度限制到 $O(\sqrt{N})$ 个节点

α pruning: 在Max层，如果 $\alpha \geq \beta$ 则剪枝 α

β pruning: 在Min层，如果 $\alpha \geq \beta$ 则剪枝 β

[剪枝](#)发生的情况：

其兄弟节点比它的节点更优。它的另外一个子节点被剪枝？

由于不会选它的父亲了，因此它被剪枝了

根据[深度优先搜索](#)，必须先遍历左孩子，才是根，才是右孩子，只可能是右孩子被剪枝

Min里面找最大，Max里面找最小？

5.4 Questions

What makes the time complexity analysis of a backtracking algorithm very difficult is that *the number of solutions* that do satisfy the restriction is hard to estimate.

TRUE

In backtracking, if different solution spaces have different sizes, start testing from the partial solution with the largest space size would have a better chance to reduce the time cost.

FALSE

[题集][Lecture 6. Backtracking in the tic-tac-toe game, a "goodness" function of -CSDN博客](#)

6. Divide and Conquer

6.1 closest points

分成多个子问题，用循环解决

在N个点中，找到距离最近的两个点（重合为0）

1. 穷举：check $N(N-1)/2$ 次, $T = O(N^2)$
2. 分治：分成两部分，最小只有3种情况, $T = O(N \log N)$

代码：

6.2 三种方法解决递归

1. substitution

先猜，然后用归纳法证明

eg, $T(N) = 2T(\lfloor N/2 \rfloor) + N$, $T = O(N \log N)$

2. recursion-tree

eg, $T(N) = 3T(N/4) + \Theta(N^2)$

高度： $\log_4 N$

3. master theorem

【Master Theorem】 The recurrence $T(N) = aT(N/b) + f(N)$ can be solved as follows:

1. If $af(N/b) = \kappa f(N)$ for some constant $\kappa < 1$, then $T(N) = \Theta(f(N))$
2. If $af(N/b) = K f(N)$ for some constant $K > 1$, then $T(N) = \Theta(N^{\log_b a})$
3. If $af(N/b) = f(N)$, then $T(N) = \Theta(f(N) \log_b N)$

【Theorem】 The solution to the equation

$T(N) = a T(N/b) + \Theta(N^k \log^p N)$,
where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

算法设计教材中给出的Master定理可以解决该类方程的绝大多数情况，根据Master定理：o-渐进上界、w-渐进下界、O-渐进确界。

设 $a \geq 1$, $b > 1$ 为常数, $f(n)$ 为函数, $T(n) = aT(n/b) + f(n)$ 为非负数, 令 $x = \log_b a$:

1. $f(n) = o(n^{x-\epsilon})$, $\epsilon > 0$, 那么 $T(n) = O(n^x)$ 。
2. $f(n) = O(n^x)$, 那么 $T(n) = O(n^x \log n)$ 。
3. $f(n) = \omega(n^{x+\epsilon})$, $\epsilon > 0$ 且对于某个常数 $c < 1$ 和所有充分大的 n 有 $af(n/b) \leq cf(n)$, 那么 $T(n) = O(f(n))$ 。

6.3 Questions

快排 quick sort 和归并 merge 用了 divide and conquer 算法

For the recurrence equation $T(N) = aT(N/b) + f(N)$, if $af(N/b) = f(N)$, then $T(N) = \Theta(f(N) \log_b N)$.

☒ T ☐ F

答案正确: 2 分  创建提问 

[数据结构错题整理 \(二\) for the recurrence equation \$t\(n\) = at\(n/b\) + f\(n\)\$, if - CSDN 博客](#)

Which one of the following is the lowest upper bound of $T(n)$. $T(n)$ for the following recursion $T(n) = 2T(\sqrt{n}) + \log n$? (4分)

- ☒ A. $O(\log n \log \log n)$
☐ B. $O(\log^2 n)$
☐ C. $O(n \log n)$
☐ D. $O(n^2)$

设 $m = \log n$, 则 $2^m = n$.

$T(2^m) = 2T(2^{m/2}) + m$

设 $G(m) = T(2^m)$, 则原式转化为 $G(m) = 2G(m/2) + m$

根据主定理, $a = 2$, $b = 2$, $k = 1$, $p = 0$. $a = b^k$, 满足条件2, 所以算法复杂度为 $O(m \log m)$

又因为 $m = \log n$, 所以算法复杂度为 $O(\log n \log \log n)$

7. Dynamic programming

动态规划可以解决 Longest common subsequence problem

polynomial time 多项式时间

- 动态规划不保证都能能在多项式时间解决问题

To solve a problem by dynamic programming instead of recursions, the key approach is to store the results of computations for the subproblems so that we only have to compute each different subproblem once. Those solutions can be stored in an array or a hash table.

7.1 Fibonacci number

只解决子问题一次，将答案存在表中

$$T = O(N)$$

7.2 Matrix Multiplication

矩阵链乘法

$$T = O(N^3)$$

7.3 Optimal binary search tree

最优二叉搜索树是静态搜索（没有插入和删除）最佳，使所有操作访问的节点总数最少

在weight一定时，选cost最小的

optimal cost = total weight + this cost

$$T = O(N^3)$$

7.4 Floyd-warshall算法

All-pairs shortest path

所有结点对的最短路径问题：

1. Dijkstra 单源最短路径 $T = O(V^3)$ 适合稀疏图
2. Floyd-warshall，不能处理负数cost，因为弗洛伊德算法将在有限步后终止，但如果存在负cost循环，则最短距离为负无穷大。

$$T = \Theta(V^3), \text{稠密图会更快}$$

8. Greedy Algorithms

8.1 Introduction

贪心算法在每个阶段都做当前限制下的最优解，结果不回退

note：贪心只在当前最佳=全局最佳时有用，它并不保证最优解（但接近）

8.2 Activity selection

找最多不冲突的时间段

8.2.1 A DP problem

Greedy Rule 1：选出时间段开始最早的（但不与已经选出的间隔重叠）×

Greedy Rule 2：选出时间段最短的×

Greedy Rule 3：选出时间段冲突最少的×

Greedy Rule 4：选出时间段结束最早的√

【Theorem】

令S为活动选择问题（Activity Selection Problem）中所有活动的集合。则**最早结束的活动** a_s 一定被包含在S的某个最大相容活动子集中。

$$T = O(N \log N)$$

8.2.2 DP solution

If each activity has a weight, DP solution is still correct but Greedy solution may be not.

0-1背包问题

```
f[i][j]=max(f[i-1][j-w[i]]+v[i], f[i-1][j])
```

$$T = O(N^2 W), W \text{是个数}$$

8.3 Huffman code

频率高的字符编码简短

$$cost = \sum d_i f_i$$

为了得到唯一的解码（消除二义性），对不同字母编码前缀要不同

所有结点不是叶子就必须有两个孩子

build the tree from bottom-up

```
[](#__code_lineno-5-1)
```

$$T = O(C \log C)$$

8.4 Questions

[实验12 Greedy Algorithm练习题 答案与解析 given 4 cases of frequencies of four characters. in-CSDN博客](#)

9. NP-completeness

9.1 NP definition

停机问题(halting problem)是不可判定问题(undecidable problem)

P: 能在多项式时间内解决的问题

NP (nondeterministic polynomial-time)：不能在多项式时间内解决或不确定能不能在多项式时间内解决，但如果我们可以在多项式时间内证明一个问题的任意“是”的实例是正确的，那么这个问题属于NP类

NP类包含所有多项式时间解的问题

NP-complete(NPC)：NPC问题是NP问题的一个子集，是最难的NP问题。任意NP问题在多项式时间内都能够规约到它的NP问题，即解决了此NPC问题，所有NP问题也都得到解决。

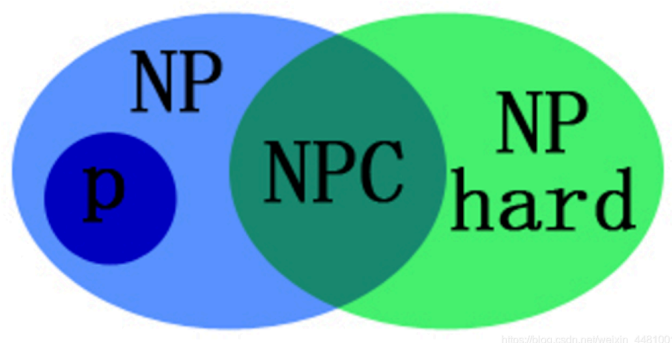
如果NPC问题有一个多项式的解，那么NP中的每一个问题必然都有一个多项式的解，这使得NPC问题是所有NP问题中最难的问题。

NP hard：NP难问题，所有NP问题在多项式时间内都能约化(Reducibility)到它的问题(不一定是NP问题)。

All NP problems can be solved in polynomial time in a non-deterministic machine. NP问题可以被非确定图灵机在多项式时间内解决

All Np problems are decidable.

如果A是NPC且A能在多项式时间内化成B，则A比B要难



9.2 NPC problems

NPC问题有：circuit-SAT, traveling salesman巡回售货员, Hamilton cycle哈密顿圈, 最长路径, bin packing装箱问题, knapsack背包问题, graph coloring图着色, vertex cover

- If a decision problem B is in P and A reduces to B, then decision problem A is in P.
- A decision problem B is **NP-complete** if B is in NP and for *every NP problem A*, A reduces to B.
- A decision problem C is **NP-complete** if C is in NP and for *some NP-complete problem B*, B reduces to C.
- X reduces to Y means $X \leq Y$, $X \Rightarrow Y$, X可以多项式规约到Y

9.2.1 Proof

为了证明某个新问题是NPC，必须先证明它属于NP，然后将一个适当的NPC问题变换到该问题。

第一个NPC问题是可满足性问题（satisfiability）：把一个bool表达式作为输入并提问是否该表达式对式中各变量的一次赋值取值1

9.3 Formal language framework

【Formal language theory】A language L belongs to **NP** iff there exist a two-input polynomial-time algorithm A that verifies language L in polynomial time.

co-NP: A problem has its complement in NP

9.4 Questions

1. Given that problem A is NP-complete. If problem B is in NP and can be polynomially reduced to problem A, then problem B is NP-complete. ×

F The correct requirement for B to be NP-complete is not just that it is in NP and reduces to A, but that every problem in NP can be reduced to B. 少了个条件：B还必须是NP-hard，即 every problem in NP can be reduced to B

[题解]ADS10 NP-Completeness if $1 \leq p \mid 2$ and $1 \mid 2 \in np$, then $1 \mid 1 \in np$.-CSDN博客

10. Approximation algorithm

近似算法

10.1 Introduction

An **approximation** algorithm guarantees to seek out high accuracy and top quality solution (say within 1% of optimum) Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time

approximation ratio: $\rho(n)$, $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$

PTAS: polynomial time approximation schema

PTAS 's complexity: $O(n^{2/\epsilon})$

FPTAS 's complexity: $O(n^3(1/\epsilon)^2)$

10.2 Approximate bin packing

近似装箱问题: np-hard问题

Next fit

眼前的bin放不下就开新的bin

approximation ratio: 2

Next fit never uses more than **2M - 1** bins (M: the optimal number)

```
[ ](#__codeline-6-1)void NextFit ( ) [ ](#__codeline-6-2){ read item1; [ ]
(#__codeline-6-3) while ( read item2 ) { [ ](#__codeline-6-4) if (
item2 can be packed in the same bin as item1 ) [ ](#__codeline-6-5) place item2
in the bin; [ ](#__codeline-6-6) else [ ](#__codeline-6-7) create a new
bin for item2; [ ](#__codeline-6-8) item1 = item2; [ ](#__codeline-6-9)
} /* end-while */ [ ](#__codeline-6-10)}
```

First fit

从左到右遍历找到可以放下当前package的bin

First fit never uses more than **17M / 10** bins (M: the optimal number)

```
[ ](#__codeline-7-1)void FirstFit ( ) [ ](#__codeline-7-2){ while ( read item )
{ [ ](#__codeline-7-3) scan for the first bin that is large enough for item;
[ ](#__codeline-7-4) if ( found ) [ ](#__codeline-7-5) place item in
that bin; [ ](#__codeline-7-6) else [ ](#__codeline-7-7) create a new
bin for item; [ ](#__codeline-7-8) } /* end-while */ [ ](#__codeline-7-9)}
```

Can be implemented in $O(N \log N)$

Best fit

找一个尽量塞得最满的bin放package, 可以回退以达到best fit

```
[ ](#__codeline-8-1)
```

online algorithm

不改决定，一个一个处理package，不能一直保证结果是最佳

【Theorem】 There are inputs that force any on-line bin-packing algorithm to use at least $5/3$ the optimal number of bins.

offline algorithm

全局判断，等到输入数据被读完后再做决定

trouble maker：大物品

解决办法：把物品大小从小到大排序，接着使用first fit/best fit decreasing算法

First fit decreasing never uses more than $11M/9 + 6/9$ bins (M: the optimal number)

10.3 Knapsack problem

第一种：simple 可取物品的部分，对 $profit_i/weight_i$ 贪心（取最大的放）

第二种：hard 0-1version 只能取整个物品或者不取

approximation ratio: 2

动态规划解决：

10.4 10.4 K-center

确定中心使到点的距离最小

```
[ ](#__codelineno-9-1)Centers Greedy-2r ( Sites S[ ], int n, int K, double r ) [ ]
(#__codelineno-9-2){ Sites S'[ ] = S[ ]; /* S' is the set of the remaining sites
*/ [ ](#__codelineno-9-3) Centers C[ ] = [ ]; [ ](#__codelineno-9-4) while ( S'[
] != [ ] ) { [ ](#__codelineno-9-5) Select any s from S' and add it to C; [ ]
(#__codelineno-9-6) Delete all s' from S' that are at dist(s', s) > 2r; [ ]
(#__codelineno-9-7) } /* end-while */ [ ](#__codelineno-9-8) if ( |C| ≤ K )
return C; [ ](#__codelineno-9-9) else ERROR(No set of K centers with covering
radius at most r); [ ](#__codelineno-9-10)}
```

算 $r(c^*)$

【Theorem】 Suppose the algorithm selects more than K centers. Then for any set C of size at most K, the covering radius is $r(C) > r$.

approximation ratio: 2

binary search:

a smarter solution:

```
[ ](#__codelineno-10-1)
```

10.5 Questions

[题解]ADS11 Approximation as we know there is a 2-approximation algorithm fo-CSDN博客

11. Local search

11.1 basic

solve problems approximately at a local optimum

neighbor relation $S \sim S'$: S' is a neighboring solution of S - S' can be obtained by a small modification of S

$N(S)$: neighborhood of S - the set $\{S' : S \sim S'\}$

```
[ ]( #__code_lineno-11-1 )//梯度下降
```

11.2 Vertex cover

problem:

Metropolis algorithm

```
[ ]( #__code_lineno-12-1 )
```

11.3 Hopfield neural

state-flipping algorithm

11.4 Maximum Cut problem

```
[ ]( #__code_lineno-13-1 )
```

最多 $O(n/\varepsilon * \log W)$

11.5 Questions

greedy&local search 最小生成树问题

2. Max-cut problem: Given an undirected graph $G = (V, E)$ with positive integer edge weights w_e , find a node partition (A, B) such that $w(A, B)$, the total weight of edges crossing the cut, is maximized. Let us define S' be the neighbor of S such that S' can be obtained from S by moving one node from A to B , or one from B to A . We only choose a node which, when flipped, increases the cut value by at least $w(A, B)/|V|$. Then which of the following is true?
- ☐ A. Upon the termination of the algorithm, the algorithm returns a cut (A, B) so that $2.5w(A, B) \geq w(A^*, B^*)$, where (A^*, B^*) is an optimal partition.
 - ☐ B. The algorithm terminates after at most $O(\log |V| \log W)$ flips, where W is the total weight of edges.
 - ☒ C. Upon the termination of the algorithm, the algorithm returns a cut (A, B) so that $2w(A, B) \geq w(A^*, B^*)$.
 - ☐ D. The algorithm terminates after at most $O(|V|^2)$ flips.

The answer is A: "Upon the termination of the algorithm, the algorithm returns a cut (A, B) so that $2.5w(A, B) \geq w(A^*, B^*)$, where (A^*, B^*) is an optimal partition".

The analysis is given as below.

Let $\epsilon = 1/2$.

Claim: the above algorithm returns a cut (A, B) with $(2 + \epsilon)w(A, B) \geq w(A^*, B^*)$.

We can show that the algorithm terminates after $O(\epsilon^{-1} n \log W)$ flips, and each flip improves a factor of $(1 + \epsilon/n)$, thus after n/ϵ iterations the cut value improves by a factor of 2, due to the fact that $(1 + 1/x)^x \geq 2$ for positive number x .

Proof of the claim.

Given a cut (A, B) , and for each $u \in A$, we have for each $2 \sum_{v \in A} w_{u,v} \leq (1 + 2\epsilon/n)w(A, B)$, where $v \in A$ and $t \in B$.

$\sum_{(u,v) \in A} w_{u,v} + \sum_{(u,v) \in B} w_{u,v} \leq (1 + \epsilon)w(A, B)$, which implies $2 + \epsilon$ approximation of the algorithm.

12. Randomize

12.1 Introduction

$\Pr[A]$: 事件A的概率

\overline{A} : 事件A的补集

算法随机, 不是输入随机

12.2 Hiring problems

Naive solution: 遇见比之前最好的质量更好的就替换

worst case: 候选人质量升序排序 $O(NC_h)$

Randomness assumption: 前i个候选者都平等的可能是最好的

Randomized permutation algorithm

Online hiring algorithm--hire only once

$$\Pr[S_i] = \Pr[A \cap B] = \Pr[A] * \Pr[B] = 1/N = \frac{1}{N} (i - 1)$$

$$\Pr[S] = \sum_{i=k+1}^N \Pr[S_i] = \sum_{i=k+1}^N \frac{1}{N} (i - 1) = \frac{1}{N} \sum_{i=k+1}^N (i - 1)$$

$$\frac{1}{N} \sum_{i=k+1}^N (i - 1) \leq \Pr[S] \leq \frac{1}{N} \sum_{i=k+1}^N (i - 1)$$

best value of $k = \frac{N}{e}$

succeed in hiring the best-qualified applicant with probability at least $1/e$

12.3 Randomized quick sort

随机选pivot, 得到central splitter的概率 $\Pr=0.5$

$$E[T_{typej}] = O(N)$$

number of different types = $O(\log N)$

总 $O(N \log N)$

13. Parallel Algorithm

13.1 Introduction

13.1.1 Parallel Random Access Machine(PRAM)

To solve access conflicts, Exclusive-Read-Exclusive-Write(EREW)

concurrent-read Exclusive-Write(CREW)

Concurrent-Read Concurrent-Write(CRCW)

$$T(n) = \log n + 2$$

13.1.2 Work-Depth(WD)

$W(n)$: total number of operations

$$W(n) = 2n$$

13.2 Prefix-sums

$$W(n) = O(n)$$

$$T(n) = O(\log n)$$

C是从上往下计算的，B是从下往上算的

13.3 Merging

merge \rightarrow rank

$$T = O(1)$$

$$W = O(n+m) = O(n \log n) \text{ -- binary search}$$

13.3.1 Parallel ranking

1. partition 分组AB: $p = n / \log n$

2. actual ranking: 最多 $2p$ 个 $O(\log n)$ 子问题

$$W(n) = O(n); T(n) = O(\log n)$$

13.4 Maximum finding

a doubly-logarithmic paradigm

$$h = \log \log n$$

$$\text{All pairs: } T(n) = O(1)$$

$$\text{Random samplin: } W(n) = O(n); T(n) = O(1)$$

$$O(1/n^c)$$

并行方法将问题分成子问题只能减少workload，不能降低最坏情况

13.5 Questions

[题解]ADS14 Parallel Algorithms while comparing a serial algorithm with its parall-CSDN博客

14. External Sorting

14.1 Introduction

mergesort

store data on tapes, can use at least 3 drives

N: number of records M: size of internal memory

14.1.1 Example

10M(10,000,000) records of 128 Bytes each and 4MB internal memory.

1. the number of runs: $128 \times 10\text{MB} / 4\text{MB} = 320$ runs
2. 1(the first run generation) + $\log_2 320 = 10$ passes

Then, **the number of passes is** $1 + \lceil \log_2 N / M \rceil$

seek time: O(number of passes)

14.2 Pass reduction

use k-way merge

每组第一个先加入heap，最小的弹出，然后其余两个上移左移，再加入新的元素（按第一组第一个、第二组第一个、第k组第一个、第一组第二个、第二组第二个……）

The number of passes is $1 + \lceil \log_k N / M \rceil$; we need **2k tapes**

14.2.1 Polyphase merge

require k+1 tapes only, split into $F_{N-1}, F_{N-2}, \dots, F_{N-k}$, where $F_N^k = 0, F_{k-1}^k = 1$, number of runs is the Fibonacci number F_N

按斐波那契数分割更好

14.3 Buffer handling

k-way merge: **2k input** buffers & **2 output** buffers

I/O time will increase despite the decrease of passes, the optimal k depends on the disk parameters and amount of internal memory available for buffers

14.4 Run generation and merge

Replacement selection

$L_{avg} = 2M$ (expected value) 对基本有序的输入很有用，输出有序段长短不同

Huffman哈夫曼 tree can get minimum merge times.

提高内存利用率，升级硬盘，提升I/O速度

To sort N numbers by external sorting using a k -way merge and a k -size heap, which statement is TRUE about the total comparison times $T(N, k)$ and k ?

- ☐ A. $T(N, k)$ is $O(k)$ for fixed N .
- ☐ B. $T(N, k)$ has nothing to do with k .
- ☐ C. $T(N, k)$ is $O(k^2)$ for fixed N .
- ☒ D. $T(N, k)$ is $O(k \log k)$ for fixed N .

答案错误: 0 分

🔒 创建提问

A. $T(N, k)$ is $O(N \log k)$ for a fixed N

[数据结构与算法 外部排序 to merge 55 runs using 3 tapes for a 2-way merge, -CSDN博客](#)

2024年6月26日 00:54:04 2024年6月25日 23:02:31