by harveechen@gmail.com

last modified 20-05-23

2018

* 符号标注的题目存在争议

选择题

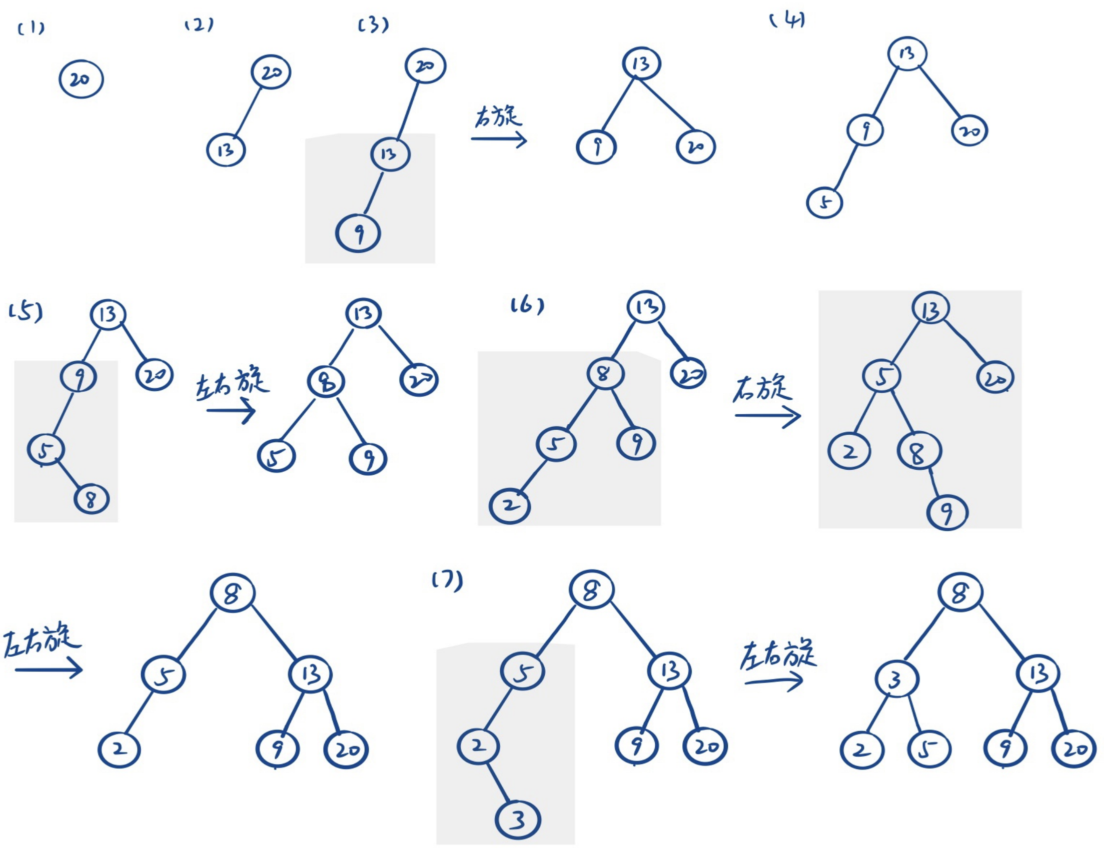
1	2	3	4	5	6	7	8	9	10
C	B	D	A	A	B	A	A	C	B

11	12	13	14	15	16	17	18	19	*20
C	B	D	B	C	D	A	A	B	B

应用题

21

1)



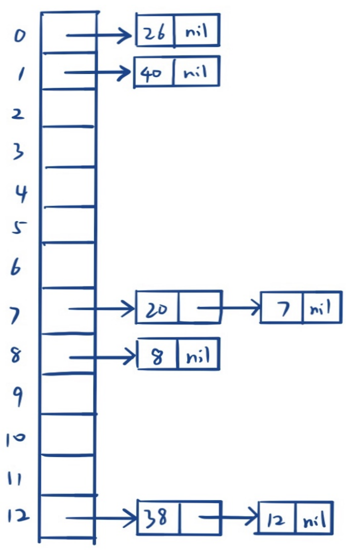
2)

```
/*
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
    TreeNode(int x) :
        val(x), left(NULL), right(NULL) {
    }
};
*/
bool AvlSearch(TreeNode *t, int key) {
    if (t == NULL)
        return false;
    if (t->val == key)
        return true;
    if (t->val > key)
        return AvlSearch(t->left, key);
    else
        return AvlSearch(t->right, key);
}
```

3)
因为 AVL 树是平衡的二叉树，任何节点的左右子树的高度差不超过1，树的深度为 $\lfloor logn \rfloor + 1$，所以搜索路径长度最大也是 $\lfloor logn \rfloor + 1$，因为沿着搜索路径的每一步递归花费常数时间，所以总的算法复杂度为 $O(logn)$

22

1)



2)
$ASL_{succ} = (1 + 1 + 2 + 1 + 1 + 2 + 1)/7 = 9/7$

3)
$ASL_{unsucc} = (2 + 2 + 8 * 1 + 3 + 2 + 3)/13 = 20/13$

23

1)

```
vector<int> A;
bool AscendingBinarySearch(int low, int high, int key) {
    // recurse end condition
    if (low > high)    // empty array
        return false;
    if (low == high)    // single element
        return A[mid] == key;

    int mid = (low+high)/2;    // low <= mid < high
    if (A[mid] == key) return true;
    // then key may exist in two side
    if (A[mid] < key) {    // A[mid] < key <= A[high]
        return AscendingBinarySearch(mid+1, high, key);
    } else {
        return AscendingBinarySearch(low, mid-1, key);
    }

}

bool DescendingBinarySearch(int low, int high, int key) {
    // recurse end condition
    if (low > high)    // empty array
        return false;
    if (low == high)    // single element
        return A[mid] == key;

    int mid = (low+high)/2;    // low <= mid < high
    if (A[mid] == key) return true;
    // then key may exist in two side
    if (A[mid] > key) {    // A[mid] > key >= A[high]
        return DescendingBinarySearch(mid+1, high, key);
    } else {
        return DecendingBinarySearch(low, mid-1, key);
    }

}

bool BitnoicSearch(int low, int high, int key) {
    // recurse end condition
    if (low > high)    // empty array
        return false;
    else if (low == high)    // single element
        return A[low] == key;

    int mid = (low+high)/2;    // low <= mid < high
    if (A[mid] == key) return true;
    // then key may exist in two side
    if (A[mid+1] > A[mid]) {    // mid is in ascending side
        if (key > A[mid])   // key val is not in monotonous sequence,
                            // what is to say, bitnoic sequence
            return BitnoicSearch(mid+1, high, int key);
        else {    // see explanation following code
            return AscendingBinarySearch(low, mid-1, key) ||
                DescendingBinarySearch(mid+1, high, key);
        }
    } else {    // mirror case
        if (key > A[mid])
```

```
                return BitnoicSearch(low, mid-1, int key);
            else {
                return AscendingBinarySearch(low, mid-1, key) ||
                    DescendingBinarySearch(mid+1, high, key);
            }
        }
    }
    int main() {
        init(A);
        cout << (BitnoicSearch(0, A.size()-1, key)? "Yes" : "No") << endl;
    }

```

算法思想:

二分法。将原数组分成两半，当待查值 key 比中间值 A[mid] 大时，mid 所在的单调序列中小于 A[mid] 的一侧将不再需要考虑，另一侧是问题规模缩小一半的子问题；另一方面，当待查值 key 比中间值 A[mid] 小时，key 可能落在 mid 值划分的两边，可以直接当作单调序列对两边分别进行二分查找（哪怕一边是 bitnoic 数组），原因是 bitnoic 中比 A[mid] 大的部分除了第一次二分，之后都不会再被访问到，这种情况 BitnoicSearch 函数直接变成做两次二分查找。

> - 一半做简单二分，一半递归的思路不能达到 $logn$ 的复杂度
>
> $T(n) = T(n/2) + log(n/2) + c$
>
>   上式的解是 $O(log^2 n)n$
> - 此题还可以用二分法找到 peek 点，再以 peek 点为界两边做普通二分查找，
>
>   复杂度是 $O(logn)$

2)

$T(n) = T(n/2) + c_1$ 或者 $T(n) = 2log(n/2) + c_2$

由于算法的递推树的展开是不确定的，很难直接推导出这个树的深度，但是可以发现 $O(logn)$ 是方程的解，故算法的总的比较次数是 $O(logn)$

24

1)

略

2)
CA AD DF FB FE
cost = 9

3)

```
/*
queue  --> the prioir queue maintain the minimum edge
MST    --> set of edges that have already been chosen
unseen --> vector of tags that indicate whether the node has ever been visited
*/
// initial queue, mst, unseen, backEdge and cutWeight of every node

function MinimalCostPlan begin
    s.backEdge = NULL;    // init start node
    queue.insert(s, MinValue);
```

```
    while !queue.IsEmpty() do
        u = queue.extractMin();
        MST = MST + {u.backEdge};
        Update_Fringe(queue, u);
end

subroutine Update_Fringe(queue, u) begin
    foreach v neighbor of u do
        w = <u, v>.weight;
        if unseen[v] then
            v.backEdge = <u, v>;    // trace back how v is found
            queue.insert(v, w);
        else if  w < v.cutWeight then   // now find a lighter edge from MST to v
            v.backEdge = <u, v>;   // update trace info
            v.cutWeight = w;    // update edge info
end
```

4)

参考：codeforce 160D

关键边（Critical Edge，CE）是删除后（权值设为MaxValve）会使得最小生成树（MST）的权值增加的边。更进一步，CE 出现在所有的 MST 中，即 CE 是不能被替代的边；非平凡的， CE 不能被权值比自己小或相等的边替代。

下面给出 CE 的局部性质。对于无向图中的一条边，在图中删除所有比它权值大的边后，如果这条边是残图的桥，则它是关键的。证明如下。

> **引理**
> 设 $e' \in E$，$G'(V,E')$ 是图 $G(V,E)$ 的子图，$E' = \{e|e \in E, e.weight \le e'\}$。
> e' 是图 $G(V,E)$ 的 CE，当且仅当 e' 是子图 $G'(V,E')$ 的桥

证明：

充分性：当 e' 是**子图** G'的桥，假设 e' 不是 **原图** G 的 CE，则**原图** G 存在一个 MST 且 $e' \notin MST$。将 e' 加入进 MST 中，则形成一个环 L （这个环恰好也在子图 G' 中，原因见后文），且环上各条边的权值不大于 e' 的权值（否则从环上删去它，形成一个权值更小的 MST，则 MST 不是最小生成树），因此 $L \subset G'$，与 e' 是 G' 的桥矛盾。

必要性：当 e' 是**原图** G 的 CE，则存在 MST 包含 e' 。假设 e' 不是**子图** G' 的桥，又因为 e' 是**子图** G' 权值最大的边之一，令 u， v 是 e' 的端点，则子图 G' 中必定存在权值不大于 e' 的边连通 u， v 。在 MST 中用这条边替换 e'，产生的生成树还是最小生成树（事实上这条边的权值和 e' 相等，否则原 MST 不是最小生成树）。则 e' 是可替代的，与 e' 是**原图** G 的 CE 矛盾。

算法主要思想是，先将所有边用最小堆维护，每次处理所有权值相等边，初始时是一个个孤立的顶点。将这些权值相等的边加入进来，用深度优先找到所有的桥，这些桥就是关键边。
为了避免重复遍历边，每一轮找到所有的桥后，需要收缩连通片（用并查集维护），以后加入下一组边时，它的顶点变成这些连通片的编号，深度遍历时只需要在此收缩图上进行。

复杂度：建堆 O(m)，出堆 O(mlogm)，维护并查集 O(m)，DFS 搜索桥 O(n+m)

25

虚存管理基本原理：在物理地址之上建立虚拟地址及其映射，进程有统一的方法使用操作系统提供的存储服务。另一方面逻辑上连续的数据不必再在物理上连续，从而为高速存储介质的充分利用提供可能。系统可以根据活跃程度，将各个进程的一部分数据而不是全部移入靠近cpu的介质或者相反。

例子：多个进程使用相互独立的虚拟地址空间，从而支持并发。操作系统负责在内存和外存之间换入换出数据，从而进程只需要工作在内存就能得到空间巨大的外存容量，从而扩大了运行空间。

26

系统初始时 Resource[*] = Available[*]，Allocation[i, *] = (0, 0 , 0, 0)，Need[i, *] = Claim[i, *]；因为 Need[i, *] <= Available[*]，所以存在进程序列 P0，P1，P2，P3，P4 来满足资源要求，系统处于安全状态。
因为 R0 <= Need[0, *]，所以 P0 资源请求合法，又由上面结论可知 P0 的请求可以直接满足（安全序列中P0在开头），所以接受 R0；
假设继续接受 R1，此时

Thead	Allocation	Need
P0	3 2 0 0	2 0 1 3
P1	0 0 0 0	4 1 1 3
P2	2 1 0 0	1 0 0 0
P3	0 0 0 0	2 1 2 0
P4	0 0 0 0	1 0 2 1

而 Available = (0, 3, 5, 4) <= Need[i, *] 不能满足任意进程的后续资源请求，所以拒绝 R1；
假设接受R0后紧接着接受R2，此时

Thead	Allocation	Need
P0	3 2 0 0	2 0 1 3
P1	0 0 0 0	4 1 1 3
P2	0 0 0 0	3 0 1 0
P3	0 0 0 0	2 1 2 0
P4	1 0 1 1	0 0 1 0

Available = (1, 4, 4, 3) 只能满足 P4的后续资源请求，所以存在进程序列P4，P0，P1，P2，P3满足资源要求，所以系统安全，故接受 R2；

27

思路类似于多读写者问题，只不过这里读写的地位相等；
最外层的框架是南北两边竞争使用桥，第一个车的阻塞导致同方向后续车辆也阻塞，最后一个离开的车释放桥的使用。所以需要计数器，南北相互独立，各需要一个。内层在同方向上还有容量限制，所以需要一个同步信号量。

```
semaphore mutex1 = 0, mutex2 = 0; // 计数器保护信号量
semaphore bridge = 0; // 互斥使用桥
semaphore capacity = 12; // 桥的容量同步信号量
int count1= 0, count2 = 0; // 计数器
codebegin
EastCar_i() {
    来到东桥头;
    P(mutex1);
    count1++;
    if (count1 == 1)
        P(bridge); // 只有首辆车竞争使用桥  阻塞后同时阻塞等待在mutex1上的后续车辆
    V(mutex1);
```

```
    P(capacity);
    过桥
    V(capacity);
    P(mutex1)
    count1--;
    if (count1 == 0)
        V(bridge); //  最后一辆车释放桥的使用
    V(mutex1);
    从西桥头离开;
}
WestCar_j() {
    来到西桥头;
    P(mutex2);
    count2++;
    if (count2 == 1)
        P(bridge);
    V(mutex2);
    P(capacity);
    过桥
    V(capacity);
    P(mutex2)
    count2--;
    if (count2 == 0)
        V(bridge);
    V(mutex2);
    从东桥头离开;
}
codeend
```

28

1)

见书

2)

-259 = 0xfffffefd

所以从0x80496dc开始的4个存储单元依次为 0xfd，0xfe，0xff，0xff；

则0x80496dc存储单元内容为0xfd，0x80496de存储单元内容为0xff。

3)

0x804847d - 0x8048448 + 1 = 0x36 = 54

所以 sum 函数机器码占54字节。

非顺序执行的跳转类指令有0x8048455、0x8048475、0x804847d处的指令。

4)

4条指令实现 `s+=buf[i]`；

EDX寄存器存放内容是 `buf[i]`，0x80496f0存放内容是 `s`。

5)

buf、s 在可读写数据段；

sum 在只读代码段。

6)

因为main和test的代码在最后的可执行文件中一起放在了只读代码段，进入sum之前，已经掉入了内存，所以sum执行时访问指令发生0次缺页。

进入sum函数前，main需要在用户栈保存返回地址，此时已经将用户栈段调入内存，在sum运行期间用户栈段的访问不会发生缺页，只有在第一次访问buf 、s 所在可读写段时会将其一次性调入内存，所以访问数据发生1次缺页。

7)

在预处理阶段会把 stdio.h 中与 printf 相关的函数和数据申明复制进原代码文件中，以供编译器进行语法分析以及连接器进行符号解析；
printf() 函数执行中会使用trap指令，向 cpu 发起中断信号，cpu 从用户态陷入内核态，并转入相关中断处理程序。

29

1)
R_{XY} 链路AB分配到 10G/100 = 0.1G 的带宽，则AB之间可用带宽为 min{0.1G, 10G} = 0.1G
2)
RTT = 2*(50km + 1500km + 50km)/(2*10^8) = 16ms
3)
每当收到期望的 packet 或 ACK packet，滑动窗口向前推进一步，只有序号落在滑动窗口内的报文才能被发送和接收。
滑动窗口可以用来协调通行双方的传输速率。
4)
吞吐量 W/RTT = 64K*8bit/16ms = 32Mbps
5)
W = (1/8 B/bit)*0.1Gbps*(1-10%)*16ms = 180KB
6)
W = (1/8 B/bit)* 10*0.1*Gbps*(1-10%)*16ms = 1800KB
7)
初始时**慢启动**，cwnd = 1MSS 每次收到一个 ACK，cwnd 增加一个 MSS
当 cwnd >= ssthresh 时，进入**拥塞避免**阶段，每个 RTT 增加一个 MSS
在任何时候检测到分组丢失时，ssthresh = cwnd/2
如果是超时引起的，cwnd = 1MSS，进入慢启动阶段
如果是冗余 ACK 引起的，cwnd = ssthresh + 3MSS，进入快速恢复阶段
快速恢复阶段，每再收到一个冗余 ACK，cwnd 增加一个 MSS，若收到新的 ACK，则 cwnd 立刻从 ssthresh 开始，并进入拥塞避免状态
8)
超时发生后的下一个RTT
ssthresh = W/2 = 1M = 1000MSS
cwnd = 1MSS
进入慢启动，cwnd 指数增加
从 1MSS 到 1000MSS 需要10个RTT
然后进入拥塞避免，cwnd 线性增加
从 1000MSS 到 2000MSS 需要1000个RTT
所以在此达到原来窗口需要1010RTT = 16160ms
9)
对于高速链路，Jacobson 算法恢复到原来的传输速率需要很长的时间。
改进方法，使用 CUBIC 算法，使得 cwnd 幂次增长。
