面试算法题记录

- O(n)时间内找出一个无序数组中的元素，该元素比前面的都大，比后面的都小，输出所有该特性的元素。

```
vector<int> g_fPrintThePivotElements(vector<int>& data){
    int n = data.size();
    vector<int> rightMin(n);
    vector<int> result;
    rightMin[n-1] = data[n-1];
    for(int i = n-2;i>=0;i--){
        if(data[i]<rightMin[i+1]){
            rightMin[i] = data[i];
        }else{
            rightMin[i] = rightMin[i+1];
        }
    }
    int lMax = data[0];
    for(int i = 1;i<n-1;i++){
        if(data[i]>lMax && data[i]<rightMin[i+1]){
            result.push_back(data[i]);
        }
        if(lMax<data[i]){
            lMax = data[i];
        }

    }
    return result;
}
```

- 两数相加，两个正序存在链表中的数字相加。

```
struct ListNode* add(struct ListNode *a, struct ListNode *b){
    stack<int> stack1,stack2;
    while(a!=NULL){
        stack1.push(a->val);
        a = a->next;
    }
    while(b!=NULL){
        stack2.push(b->val);
        b = b->next;
    }
    int C = 0;
    struct ListNode * root  = NULL;//= (struct ListNode
*)malloc(sizeof(struct  ListNode));
    while(!stack1.empty() || !stack2.empty()){
        struct ListNode * temp = (struct ListNode *)malloc(sizeof(struct
 ListNode));
        temp->val = 0;
        if(!stack1.empty()){
            temp->val += stack1.top();
            stack1.pop();
        }
```

```
        if(!stack2.empty()){
            temp->val += stack2.top();
            stack2.pop();
        }
        temp->val = temp->val+C;
        C = 0;

        if(temp->val>9){
            C = 1;
            temp->val = temp->val%10;
        }
        if(temp->val == 1){
            printf("%d,%d,,",stack1.size(),stack2.size());
        }
        temp->next = root;
        root = temp;
    }
    return root;
}
```

1，并查集

并查集主要涉及到两种操作，查找和归并

```
int pre[1000];

int unionsearch(int root){
    while(root! = pre[root]){
        root = pre[root];
    }
    return root;
}

void join(int root1,int root2){
    int x = unionsearch(root1);
    int y = unionsearch(root2);
    pre[y] = x;
}
```

2，非递归快排

```
int singleSort(vector<int> &arr,int l,int r){
    int i = l,j = r;
    int temp = arr[i];
    while(i<j){
        while(i<j && arr[j]>=temp){
            j--;
        }
        if(i<j){
            arr[i] = arr[j];
            printf("arr[%d] = %d\n",i,arr[i]);
        }
```

```
        while(i<j && arr[i]<temp){
            i++;
        }
        if(i<j){
            arr[j] = arr[i];
            printf("arr[%d] = %d\n",j,arr[j]);
        }
        arr[i] = temp;
    }
    return i;
}

int quickSort2(vector<int> &arr){
    int n = arr.size();
    stack<int> s;
    s.push(0);
    s.push(n-1);

    while(!s.empty()){
        int r = s.top();
        s.pop();
        int l = s.top();
        s.pop();
        int temp = arr[l];
        printf("l = %d,r = %d\n",l,r);

        int i = singleSort(arr,l,r);
        if(l<i-1){
            s.push(l);
            s.push(i-1);
        }

        if(i+1<r){
            s.push(i+1);
            s.push(r);
        }
    }
}
int main(){
    int b[] = {2,3,4,5,6,7,8,23,42,3,1,5,6,7,12};
    vector<int> arr(b,b+15);
    quickSort2(arr);
    for(int i = 0;i<15;i++)
        printf("%d ",arr[i]);

}
```

3，约瑟夫环问题

对于n个人报数的约瑟夫环问题，报到m退出。第一个人（数组编号m-1）退出以后，以编号$ $

$$F(N,M) = (F(N-1,M)+M)\%N$$

4，哈希与哈希冲突

- 开放定址法
    - 当冲突发生时，使用某种探测技术在散列表中形成一个探测序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址（即该地址单元为空）为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探测到开放的地址则表明表 中无待查的关键字，即查找失败
- 再哈希法
    - 增加多个哈希函数
- 链地址法
    - 每个哈希节点增加一个next指针
- 建立公共溢出区
    - 将哈希表分为基本表和溢出表，凡是发生冲突的元素都放入溢出表

5，BFS与DFS算法

- BFS（广度优先搜索）,从算法的观点，所有因为展开节点而得到的子节点都会被加进一个先进先出的队列中。一般的实验里，其邻居节点尚未被检验过的节点会被放置在一个被称为 open 的容器中（例如队列或是链表），而被检验过的节点则被放置在被称为 closed 的容器中。使用队列不使用递归计算二叉树深度

```
int TreeDepth(TreeNode* pRoot)
{
    queue<TreeNode*> q;
    if(!pRoot) return 0;
    q.push(pRoot);
    int level=0;
    while(!q.empty()){
        int len=q.size();
        level++;
        while(len--){
            TreeNode* tem=q.front();
            q.pop();
            if(tem->left) q.push(tem->left);
            if(tem->right) q.push(tem->right);
        }
    }
    return level;
}
```

6，B树，B+树，B*树

- B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径不只两个）

  （1）排序方式：所有节点关键字是按递增次序排列，并遵循左小右大原则；

  （2）子节点数：非叶节点的子节点数>1，且<=M ，且M>=2，空树除外（注：M阶代表一个树节点最多有多少个查找路径，M=M路,当M=2则是2叉树,M=3则是3叉）；

  （3）关键字数：枝节点的关键字数量大于等于ceil(m/2)-1个且小于等于M-1个（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2);

  （4）所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应下图最后一层节点的空格子;

- B+树是B树的一个升级版，相对于B树来说B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度完全接近于二分法查找。为什么说B+树查找的效率要比B树更高、更稳定；我们先看看两者的区别

  （1）B+跟B树不同B+树的**非叶子**节点不保存关键字记录的指针，只进行数据索引，这样使得B+树每个**非叶子**节点所能保存的关键字大大增加；

  （2）B+树**叶子**节点保存了父节点的所有关键字记录的指针，所有数据地址必须要到叶子节点才能获取到。所以每次数据查询的次数都一样；

  （3）B+树叶子节点的关键字从小到大有序排列，左边结尾数据都会保存右边节点开始数据的指针。

  （4）非叶子节点的子节点数=关键字数（来源百度百科）（根据各种资料 这里有两种算法的实现方式，另一种为非叶节点的关键字数=子节点数-1（来源维基百科），虽然他们数据排列结构不一样，但其原理还是一样的Mysql 的B+树是用第一种方式实现）；

- B*树是B+树的变种，相对于B+树他们的不同之处如下：

  （1）首先是关键字个数限制问题，B+树初始化的关键字初始化个数是cei(m/2)，b*树的初始化个数为（cei(2/3*m))

  （2）B+树节点满时就会分裂，而B*树节点满时会检查兄弟节点是否满（因为每个节点都有指向兄弟的指针），如果兄弟节点未满则向兄弟节点转移关键字，如果兄弟节点已满，则从当前节点和兄弟节点各拿出1/3的数据创建一个新的节点出来；

7，KMP算法

- 问题：有一个文本串S，和一个模式串P，现在要查找P在S中的位置，怎么查找呢?
- 步骤1：根据P构造next数组，代码如下：

```
//优化过后的next 数组求法
void GetNextval(char* p, int next[])
{
    int pLen = strlen(p);
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < pLen - 1)
    {
        //p[k]表示前缀，p[j]表示后缀
        if (k == -1 || p[j] == p[k])
        {
            ++j;
            ++k;
            //较之前next数组求法，改动在下面4行
            if (p[j] != p[k])
                next[j] = k;   //之前只有这一行
            else
                //因为不能出现p[j] = p[ next[j ]]，所以当出现时需要继续递归，k =
next[k] = next[next[k]]
                next[j] = next[k];
        }
        else
        {
            k = next[k];
        }
    }
}
```

- KMP主模块代码：

```
int KmpSearch(char* s, char* p)
{
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
    {
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            //②如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j =
next[j]
            //next[j]即为j所对应的next值
            j = next[j];
        }
    }
    if (j == pLen)
        return i - j;
    else
        return -1;
}
```

8，辗转相除法

```
unsigned int Gcd(unsigned int M,unsigned int N)
{
    unsigned int Rem;
    while(N > 0)
    {
        Rem = M % N;
        M = N;
        N = Rem;
    }
    return M;
}
```

```
int gcd(int a, int b)
{
    if (a < b)
        std::swap(a, b);
    return b == 0 ? a : gcd(b, a % b);
}
```

9，二叉树的遍历

- 前序遍历：先访问根节点，在访问左子节点，最后访问右子节点。
- 中序遍历：先访问左子节点，在访问根节点，最后访问右子节点。
- 后序遍历：先访问左子节点，在访问右子节点，最后访问根节点。
- 重建二叉树：前序遍历的第一个是根节点，找到中序遍历数组中的该数字得到跟节点左侧和右侧数组的长度，然后将前序遍历数组分开，递归调用。

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <vector>
#include <iomanip>
#include <string>
#include <cstring>
#include <stack>
using namespace std;


struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    //TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

TreeNode* ConstructCore(int* startPreorder, int* endPreorder,int*
startInorder,int* endInorder){
    int rootValue = startPreorder[0];
    TreeNode* root = new TreeNode();
    root->left = nullptr;
    root->right = nullptr;
    root->val = rootValue;

    if(startPreorder == endPreorder){
        if(startInorder == endInorder && *startPreorder == *endPreorder){
            return root;
        }else{
            return nullptr;
        }
    }

    int* rootInorder = startInorder;
    while(rootInorder<=endInorder && *rootInorder!=rootValue)
        ++rootInorder;
    int leftLength = rootInorder-startInorder;
    int* leftPreOrderEnd = startPreorder+leftLength;
    if(leftLength>0){
        root->left =
ConstructCore(startPreorder+1,leftPreOrderEnd,startInorder,startInorder+left
Length);
    }
    if(leftLength<endPreorder-startPreorder){
        root->right =
ConstructCore(leftPreOrderEnd+1,endPreorder,rootInorder+1,endInorder);
```

```
    }
    return root;
}

TreeNode* Construct(int* preorder,int* inorder,int length){
    if(preorder == nullptr || inorder == nullptr || length == 0){
        return nullptr;
    }

    return ConstructCore(preorder,preorder+length-1,inorder,inorder+length-
1);
}
```

10，回溯法

- 回溯法（探索与回溯法）是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

11，最小生成树算法

12，红黑树

- 可以在O(log n)的时间内查找，插入和删除
- 特征：
    - 节点是红色或者黑色
    - 根节点是黑色
    - 所有叶子都是黑色
    - 每个红色节点的两个子节点都是黑色
    - 从任意节点到每个叶子节点的所有路径都包含相同数目的黑色节点
- 节点插入算法：
- 节点删除算法：