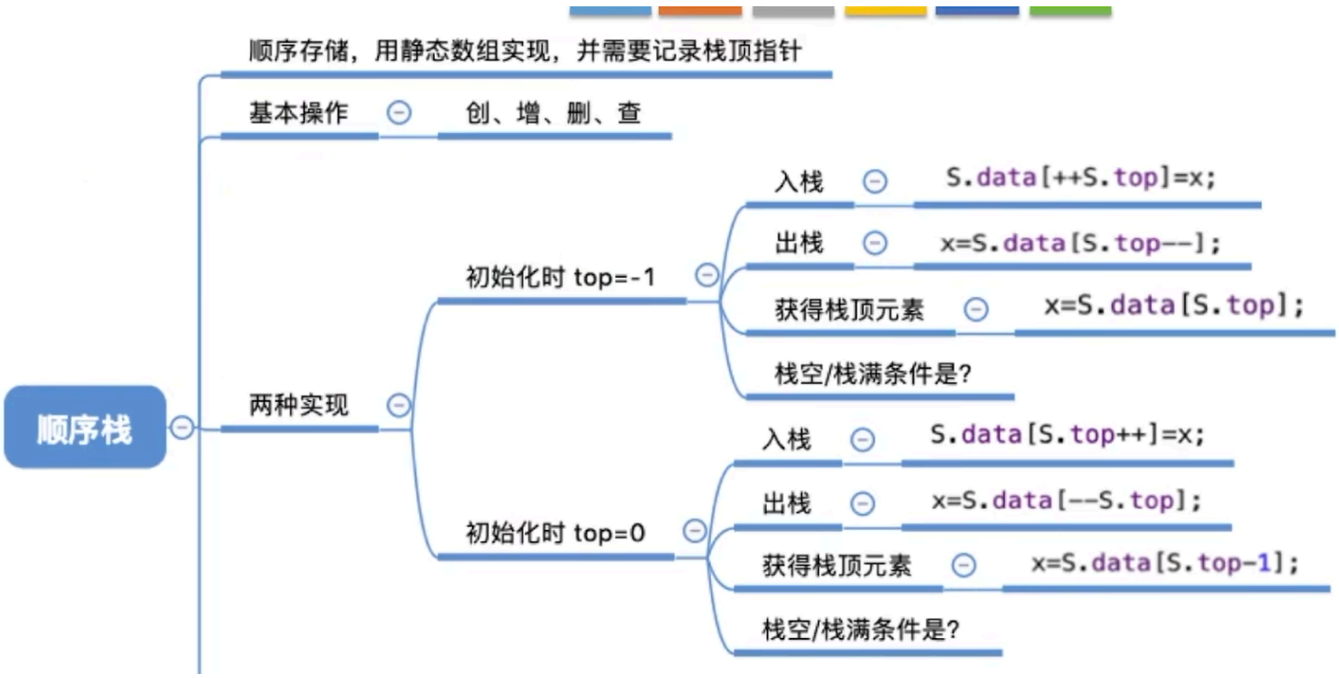
三.栈

- 栈是只允许在一端进行插入或删除操作的线性表
- 后进先出
- 题型：有哪些合法的出栈顺序

栈的基本操作

- 创、销
  InitStack(&S)：初始化栈。构造一个空栈 S，分配内存空间。
  DestroyStack(&S)：销毁栈。销毁并释放栈 S 所占用的内存空间。
- 增、删
  Push(&S, x)：进栈。若栈 S 未满，则将 x 加入使其成为新的栈顶元素。
  Pop(&S, &x)：出栈。若栈 S 非空，则弹出（删除）栈顶元素，并用 x 返回。
- 查
  GetTop(S, &x)：读取栈顶元素。若栈 S 非空，则用 x 返回栈顶元素。
- 其它
  StackEmpty(S)：判空。断一个栈 S 是否为空，若 S 为空，则返回 true，否则返回 false。

顺序栈的定义和基本操作



```
//定义
#define MaxSize 10              //定义栈中元素的最大个数

typedef struct{
    ElemType data[MaxSize];     //静态数组存放栈中元素
    int top;                    //栈顶指针
}SqStack;

void testStack(){
    SqStack S;                 //声明一个顺序栈(分配空间)

}


// 初始化栈
void InitStack(SqStack &S){
    S.top = -1;                   //初始化栈顶指针为-1;也可设为0，这样top指向下一个可以插入的位置
}

// 判断栈是否为空
bool StackEmpty(SqStack S){
    if(S.top == -1)
        return true;
    else
        return false;
}
```

```
// 新元素进栈
bool Push(SqStack &S, ElemType x){
    if(S.top == MaxSize - 1)// 栈满，报错；如果S.top初始化为0，则判断栈满：top == MaxSize
        return false;
    S.data[++S.top] = x; //指针先加1，新元素入栈
    //S.data[S.top++] = x; top初始为0
    return true;
}

// 出栈
bool Pop(SqStack &x, ElemType &x){
    if(S.top == -1) // 栈空，报错
        return false;
    x = S.data[S.top--];
    // x = S.data[--S.top] S.top初始化为0
    return true;
}
//数据还残留在内存中，只是逻辑上被删除了


// 读栈顶元素
bool GetTop(SqStack S, ElemType &x){
    if(S.top == -1)
        return false;
    x = S.data[S.top];
    return true;
}
```

链栈的定义和基本操作

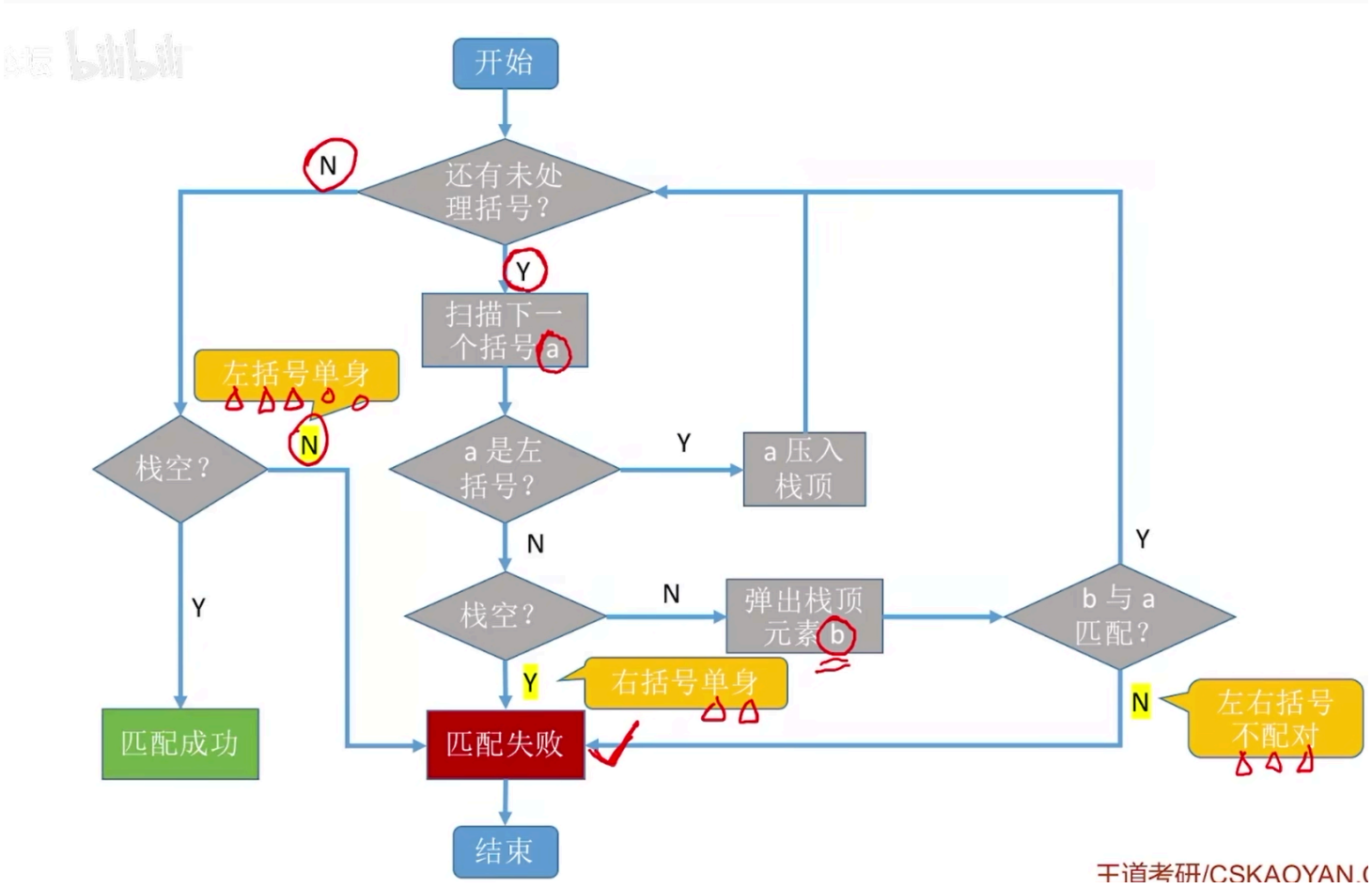
- 链栈实际上就是一个只能采用头插法插入或删除数据的单链表;

```
typedef struct Linknode{
    ElemType data;        //数据域
    Linknode *next;       //指针域
}Linknode,*LiStack;

void testStack(){
    LiStack L;            //声明一个链栈
}
```

```
// 初始化栈
bool InitStack(LiStack &L){
    L = new Linknode;
    if(L == NULL)
        return false;
    L->next = NULL;
    return true;
}

// 判断栈是否为空
bool isEmpty(LiStack &L){
    if(L->next == NULL)
        return true;
    else
        return false;
}


// 新元素入栈
bool pushStack(LiStack &L, ElemType x) {
    Linknode *s = new Linknode;
    if (s == NULL)
        return false;
    s->data = x;
    // 头插法
    s->next = L->next;
    L->next = s;
    return true;
}

// 出栈
bool popStack(LiStack &L, int &x) {
    // 栈空不能出栈
    if (L->next == NULL)
        return false;
    Linknode *s = L->next;
    x = s->data;
    L->next = s->next;
    delete s;
    return true;
}
```

栈的应用

——1.括号匹配



- 实现
  最后出现的左括号最先被匹配 （栈的特性——LIFO）。
  遇到左括号就入栈。
  遇到右括号，就“消耗”一个左括号（出栈）。
- 匹配失败情况：
  扫描到右括号且栈空。
  扫描完所有括号后，栈非空。
  左右括号不匹配。

```
#define MaxSize 10
typedef struct{
    char data[MaxSize];
    int top;
}SqStack;
//初始化栈
void InitStack(SqStack &S);
//判断栈是否为空
bool StackEmpty(SqStack &S);
//新元素入栈
bool Push(SqStack &S, char x);
//栈顶元素出栈，用x返回
bool Pop(SqStack &S, char &x);

// 判断长度为length的字符串str中的括号是否匹配
bool bracketCheck(char str[], int length){
    SqStack S;
    InitStack(S);
    // 遍历str
    for(int i=0; i<length; i++){
        // 扫描到左括号，入栈
        if(str[i] == '(' || str[i] == '[' || str[i] == '{'){
            Push(S, str[i]);
        }else{
            // 扫描到右括号且栈空直接返回
            if(StackEmpty(S))
                return false;
            char topElem;
            // 用topElem接收栈顶元素
            Pop(S, topElem);
            // 括号不匹配
            if(str[i] == ')' && topElem != '(' )
                return false;
            if(str[i] == ']' && topElem != '[' )
                return false;
            if(str[i] == '}' && topElem != '{' )
                return false;              }
    }
    // 扫描完毕若栈空则说明字符串str中括号匹配
    return StackEmpty(S);
}
```

——2.表达式求值问题

算术表达式由操作数、运算符、界限符“（”和“）”三个部分组成。

- **中缀表达式**：中缀表达式是一种通用的算术或逻辑公式表示方法，运算符以中缀形式处于操作数的中间。
  对于计算机来说中缀表达式是很复杂的，因此计算表达式的值时，通常需要先将中缀表达式转换为前缀或后缀表达式，然后再进行求值。
- **后缀表达式**（逆波兰表达式）：后缀表达式的运算符位于两个操作数之后。
- **前缀表达式**（波兰表达式）：前缀表达式的运算符位于两个操作数之前。

中缀表达式转后缀表达式-手算

步骤1： 确定中缀表达式中各个运算符的运算顺序
步骤2： 选择下一个运算符，按照[左操作数 右操作数 运算符]的方式组合成一个新的操作数
步骤3： 如果还有运算符没被处理，继续步骤2
注意：只要左边的运算符能先计算，就优先算左边的 (这样可以保证运算顺序唯一);

```
中缀：A + B - C * D / E + F
        ①   ④   ②   ③   ⑤
后缀：A B + C D * E / - F +
```

后缀表达式的计算—手算:

从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应的运算，合体为一个操作数。

中缀表达式转后缀表达式-机算

初始化一个栈，用于保存暂时还不能确定运算顺序的运算符。
从左到右处理各个元素，直到末尾。可能遇到三种情况:
1.**遇到操作数**：直接加入后缀表达式。
2.**遇到界限符**：遇到“(”直接入栈；遇到“)”则依次弹出栈内运算符并加入后缀表达式，直到 弹出“(”为止。
注意：“(”不加入后缀表达式。
3.**遇到运算符**：依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式， 若碰到“(” 或栈空则停止。之后再把当前运算符入栈。

后缀表达式的计算—机算:

步骤1: 从左往后扫描下一个元素，直到处理完所有元素;
步骤2: 若扫描到操作数，则压入栈，并回到步骤1;否则执行步骤3;
步骤3: 若扫描到运算符，则弹出两个栈顶元素，执行相应的运算，运算结果压回栈顶，回到步骤1;

注意：先出栈的是“右操作数”;
若表达式合法，则最后栈中只会留下一个元素，就是最终结果。

综合实现（中缀表达式的计算）

--中缀转后缀机算+后缀表达式机算

- 初始化两个栈，操作数栈和运算符栈;
- 若扫描到操作数，压入操作数栈;
- 若扫描到运算符或界限符，则按照“中缀转后缀”相同的逻辑压入运算符栈
  (期间也会弹出运算符，每当弹出一个运算符时，就需要再弹出两个操作数栈的栈顶元素并执行相应运算，运算结果再压回操作数栈)

——3.在递归中的应用

- 函数调用的特点：最后被调用的函数最先执行结束
- 函数调用时，需要用一个栈存储：
  - 调用返回地址
  - 实参
  - 局部变量
- 递归调用时，
  - 每进入一层递归，就将递归调用所需信息压入栈顶
  - 每退出一层递归，就从栈顶弹出相应信息
- 缺点：太多层递归可能回导致栈溢出；有时可能会包含很多重复计算。
  适合用“递归”算法解决：可以把原始问题转换为属性相同，但规模较小的问题。
  可以自定义栈将递归算法改造成非递归算法。

```
170  int func2 (int x) {
171      int m, n;
172      m = x + 1;
173      n = x + 2;
174  }
175
176  int func1 (int a, int b) {  a: 1  b: 2
177      int x= a+b;  x: 3
178      func2 (x);
179      x = x+10086;
180  }
181
182  int main() {
183      int a = 1,b = 2,c = 3;
184      func1(a, b);
185      c = a+b;
186  }
```