数据结构

时间复杂度

推导大O阶:

1. 用常数1 取代运行时间中的所有加法常数.
2. 在修改后的运行次数函数中, 只保留最高阶项.
3. 如果最高阶项存在且不是 1 , 则去除与这个项相乘的常数

线性表

```
ADT 线性表(List)
Data
    线性表的数据对象集合为{a1,a2,a3,...an},所有元素类型均为DataType.其中除a1外,每个元素有且只有一个直接前驱元素;其中除an外,每个元素有且只有一个直接后继元素.元素之间是一对一的关系.
Operation
    InitList(List* L);
    ListEmpty(List L);
    ListLength(List L);
    ClearList(List* L);

    GetElem(List L,int i,DataType* e);
    LocalElem(List L,DataType e);

    ListInsert(List L,int position,DataType x);
    ListDelete(List L,int position,DataType* x);
endADT
```

· 顺序表

起始位置 + 最大容量 + 当前(已占用)长度

```
# define MAXSIZE 20

typedef int ElemType;
typedef struct {
    ElemtType data[MAXSIZE];
    int length;
}SqList
```

· 链表

节点 = 数据域(存储数据元素信息) + 指针域(存储指针)

对比 头节点(为方便操作在第一个节点前设计头节点, 可不存信息或存表长) 与 头指针(指向第一个节点)

```
typedef int ElemType;
typedef struct {
    ElemType data;
    struct Node *next;
}Node;
typedef struct Node* LinkList;
```

Summary

线性表

顺序存储结构

链式存储结构

单链表

静态链表 (用数组实现)

循环链表

双向链表

栈

· 子弹弹夹 “压入” “弹出”

```
ADT 栈(stack)
Data
    同线性表,元素具有相同类型,相邻元素具有前驱和后继的关系.FILO(先进后出)
Operation
    InitStack(stack *S);
    EmptyStack(stack S);
    ClearStack(stack *S);
    StackLength(stack S);
    DestoryStack(Stack * S)

    StackPush(stack *S,ElemType e);
    StackPop(stack *S,ElemType *e);
    StackTop(stack S,ElemType *e);
```

```
endADT
```

栈的结构定义 (顺序表)

```
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int top;
}SqStack;
```

栈的结构定义 (链表)

```
typedef int ElemType;
typedef struct {
    ElemType data;
    ElemType *next;
}StackNode, *LinkStackPtr;

typedef struct {
    StackNode s;
   *LinkStackPtr top;
    int count;
}LinkStack;
```

栈的作用: 实现递归, 逆波兰式(后缀表达式)

队列

电脑死机, 一连串的指令会未响应, 末了突然全部执行(eg. 无数弹窗)

```
ADT 队列(Queue)
Data
    同线性表，元素具有相同的类型，相邻元素具有前驱动和后驱关系
Operation
    InitQueue(*Q);
    QueueEmpty(Q);
    ClearQueue(*Q);
    QueueLength(Q);
    DestoryQueue(*Q);
```

```
    InQueue(*Q, e);
    OutQueue(*Q, *e);
    GetHead(Q, *e); //front ptr
endADT
```

```
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int front;
    int rear;
    // int length = (rear - front + MAXSIZE)% MAXSIZE;
}SqQueue
```

```
typedef int ElemType;
typedef struct {
    ElemType data;
    struct QNote *next;
}QNode, *QNodePtr;
typedef struct {
    QNodePtr front;
    QNodePtr rear;
}LinkQueue;
```

栈

顺序栈

两栈共享空间

链栈

队列

顺序队列

循环队列

链队列

串

```
ADT 串(String)

Data
    串中元素仅有一个字符组成，相邻元素具有前驱和后继有人的关系.
Operation
 InitString(*S);
 StringEmpty(S);
 ClearString(S);
 DestoryString(S);
 StringLength(S);

 StringInsert(S, pos, T);
 StringDelete(*S, pos, len);
 StringCmp(*S, *T);
 Concat(*S, *T);
 StringSubStr(Sub,S,pos,len);
 StringIndex(*S, *T);
 StringReplace(*S, e, *T);
```

```
int Index(String S, String T,int pos) {
    int n,m,i;
    String sub;
    if(pos > 0) {
        n = StrLength(S);
        m = StrLength(T);
        i = pos;
        while(i<=n-m+1) {
            Substring(sub,S,i,m);
            if(StringCompare(sub,T) != 0)
                ++i;
            else
                return i;
        }
    }
}
```

模式匹配: KMP算法

```
void get_next(String T, int *next) {
    int i, j;
    i = 1;
    j = 0;
    next[1] = 0;
    while(i<T[0]) {
        if( j == 0 || T[i] == T[j]) {
            ++i;
            ++j;
            next[i] = j;
        }
        else {
            j = next[j];
        }
    }
}
```

改进KMP算法,

它是计算出NEXT值的同时, 如果 a 位字符与他的next指向的 b 位字符相等,则 a 位的nextval就指向 b 位的 nextval.

树

```
ADT 树 (tree)
Data
    树是由一个根结点和若干子树构成.树中节点具有相同的数据类型及层次关系
Operation
    InitTree(*T);
    DestoryTree(*T);
    CreateTree(*T, definition);
    ClearTree(*T);
    TreeEmpty(T);

    TreeDepth(T);
    Root(T);
```

```
    Value(T,cur_e);
    Assign(T,cur_e,value);

    Parent(T,cur_e);
    LeftChild(T,cur_e);
    RightSibling(T,cur_e);
    InsertChild(*T,*p,i,c);
    DeleteChild(*T,*p,i);
endADT
```

双亲表示法

```
#define MAX_TREE_SIZE 100
typedef int TElemType;
typedef struct PTNode {
    TElemType data;
    int parent;
}PTNode;

typedef struct {
    PTNode nodes[MAX_TREE_SIZE];
    int r,n;
}PTree;
```

孩子表示法

```
#define MAX_TREE_SIZE 100
typedef struct CTNode {
    int child;
    struct CTNode *next;
}*ChildPtr;
typedef struct {
    TElemType data;
    ChildPtr firstchild;
}CTBox;
typedef struct {
    CTBox nodes[MAX_TREE_SIZE];
    int r,n;
}CTree;
```

孩子兄弟表示法

```
typedef struct CSNode
{
    TElemType data;
    struct CSNode *firstchild, *rightsib;
} CSNode, *CSTree;
```

二叉树

性质 1 第 i 层最多 2^(i-1) 个节点

性质 2 深度为k的二叉树至多有2^k - 1 个节点

性质 3 终端节点 n0 度为二的节点 n2 则 n0 = n2 + 1

推导: 设 n 为总节点数 n1 为度为一的节点数

n = n0 + n1 + n2 = n1 + n2 *2 + 1 => n0 = n2 + 1

性质 4 具有 N 个节点的完全二叉树的深度为[log2N] + 1

```
typedef struct BitNode {
    TElemType data;
    struct BitNode *lchild, *rchild;
}BitNode, *BiTree;
```

前序遍历算法

```
void PreOrderTraverse(BiTree T) {
    if(T == NULL)
        return;
    printf("%c", T->data);
    PreOrderTraversw(T->lchild);
    PreOrderTraversw(T->rchild);
}
```

中序遍历算法

```
void MidOrderTraverse(BiTree T) {
    if(T == NULL) {
         return;
    }
    MidOrderTraverse(lchild);
    printf("%c", T->data);
    MidOrderTraverse(rchild);
}
```

后序遍历算法

```
void LastOrderTraverse(BiTree T) {
    if(T == NULL) {
        return;
    }
    LastOrderTraverse(BiTree T);
    LastOrderTraverse(BiTree T);
    printf("%c", T->data);
}
```

二叉树建立

```
/* 安前序输入二叉树中节点的值(一个字符) */
/* #表示空树 */

void CreateBiTree(BiTree *T) {
    TElemType ch;
    scanf("%c", &ch);
    if(ch == "#")
        *T=NULL;
    else {
        *T = (BiTree)malloc(sizeof(BiTNode));
        if(!*T)
            exit(OVERFLOW);
        (*T)->data = ch;
        CreateBiTree(&(*T)->lchild);
        CreateBiTree(&(*T)->rchild);
    }
}
```

线索二叉树

```
typedef enum {Link,Thread} PointerTag; // Link == 0 表示左右孩子指针 //
Thread == 1 表示前驱/后继

typedef struct BiThrNode
{
    TElemType data;
    struct BiThrNode *lchild, *rchild;
    PointerTag LTag;
    PointerTag RTag;
} BiThrNode, *BiThrTree;
```

```
BiThrTree pre; // 全局变量，始终指向刚刚访问过的节点

// 中序遍历进行中序线索化

void InThreading(BiThrTree p) {
    if(p) {
        InThreading(p->lchild);
        if(!p->lchild) {
            p->LTag = Thread;
            p->lchild = pre;
        }
    }
}
```