栈

一、栈 List In First Out

栈（Stack）是只允许在一端进行插入或删除操作的线性表。

LIFO 特点：后进先出

1.只能从n+1位置插入

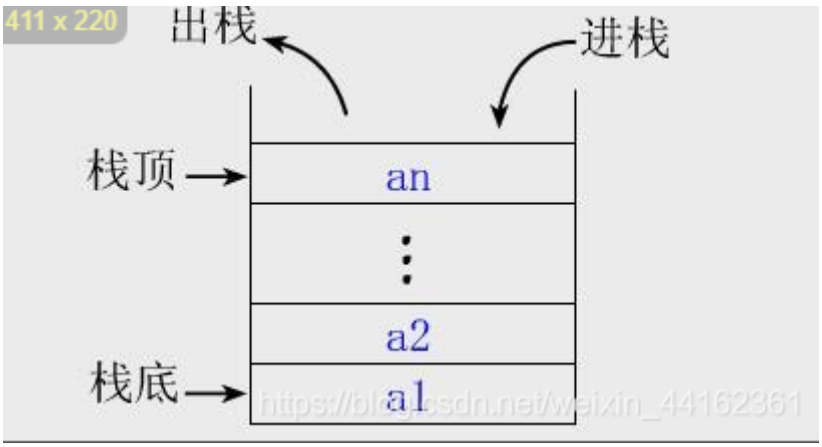
2.只能从n位置删除

- 栈顶Top：线性表允许插入和删除的那一端。
- 栈底Bottom：固定的，不允许进行插入和删除的另一端。

定义：特殊的线性表，限定仅在一端进行插入和删除操作的线性表

针对表尾进行插入和删除的线性表

表尾an为栈顶top，a1为栈底base



基本概括

- 插入元素到top为入栈 push 压入
- 从栈顶删除一个元素为出栈 pop 弹出
- 插入和删除都在栈顶执行 也就是表尾

性质：

限定只能在表的一端进行插入和删除运算的线性表

- 逻辑结构： 与线性表相同
- 存储结构： 用顺序栈或链栈存储均可，但以顺序栈更常见
- 运算规则：只能在栈顶运算，且访问结点时依照后进先出LIFO的原则
- 实现方式：关键就是入栈和出栈函数

  与线性表区别：仅在于运算规则不同

  线性表：随机存储

  栈：后进先出

  案例：进制 八皇后 迷宫问题 递归

基本操作

- 初始化InitStack(&S);
- 判空Empty(S);
- 进栈Push(&S, x);
- 出栈Pop(&S, &x);
- 读栈顶元素GetTop(S);
- 遍历栈PrintStack(&S);
- 销毁栈DestroyStack(&S);
- 栈置空操作clear（栈本身还在，但是里面没有元素）
  入栈 push 出栈 pop

存储结构 ：栈就是操作受限的线性表

1.按顺序结构存储的就是顺序栈

2.按链表存储的就是链栈

二、顺序栈的表示和实现

一组地址连续的存储的一次存放栈底到栈顶的数据元素 栈底一般在低地址端

描述：

> - top指针 指向真正的栈顶元素之上的下标地址
> - base指针 指向栈底元素的顺序栈的位置
> - stacksize 栈的大小 可使用的最大容量
>
> 标志：
>
> - 空栈：base == top 是栈空标志
> - 栈满：**top - base == stacksize**
> - 栈溢出 **top - base >stacksize**
> - 解决：1.报错 2.额外分配更大的空间，作为栈顶存储空间，将原栈的内容移入新栈
> - 栈下溢 top - base == stacksize 没有元素溢出
> - 数组作为顺序栈容易溢出
> - 1.overflow 栈已经满
> - 2.underflow 栈已经空
> - 上溢是一种错误，下溢是一种结束条件

代码实现：

1.结构体

```
typedef struct stack
{
    selemtype* base;//栈底指针
    selemtype* top;//栈顶指针
    int stacksize;//栈容量
}st;
```

1.初始化

```c
//初始化 内存中开辟一段空间来使用 top == base 都指向栈底
void initstack(st *s)
{
    s->top = (int*)malloc( MAXSIZE*sizeof(int));//分配空间 大小为100 指向这个空间的首元素
    if (s->top == NULL)
    {
        printf("开辟失败");
        exit(-1);
    }
    s->base = s->top;
    s->stacksize = MAXSIZE;
    printf("栈已经开辟成功!\n");
}
```

2.判断为空？

```c
//顺序栈是否为空
bool isempty(st *s)
{
    if (s->base == s->top)
    {
        return true;
        printf("栈空! ");
    }
    else
    {
        return false;
        printf("栈回收! ");
    }
}
```

3.len长

```c
int lenstack(st *s)
{
    return (s->top - s->base);
}
```

4.清空

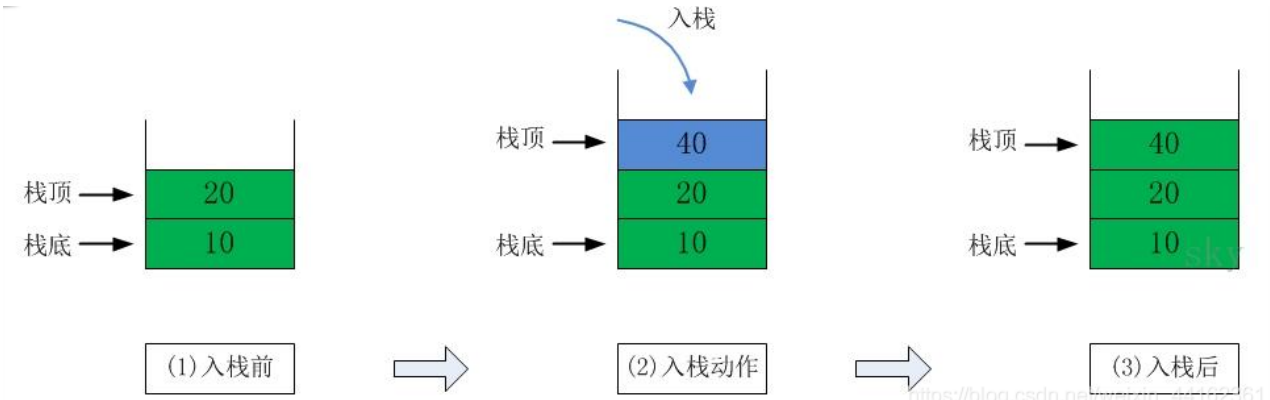
```c
//清空顺序栈
void qkstack(st *s)
{
    if (s->base)
    {
        s->top = s->base;
        printf("栈已经清空了！ ");
    }
}
```

4.销毁

```c
oid destoryst(st *s)
{
    if (s->base)
    {
        free(s->base);
        s->stacksize = 0;
        s->base = s->top = NULL;
        printf("栈已经释放了！收回内存");
    }
}
```

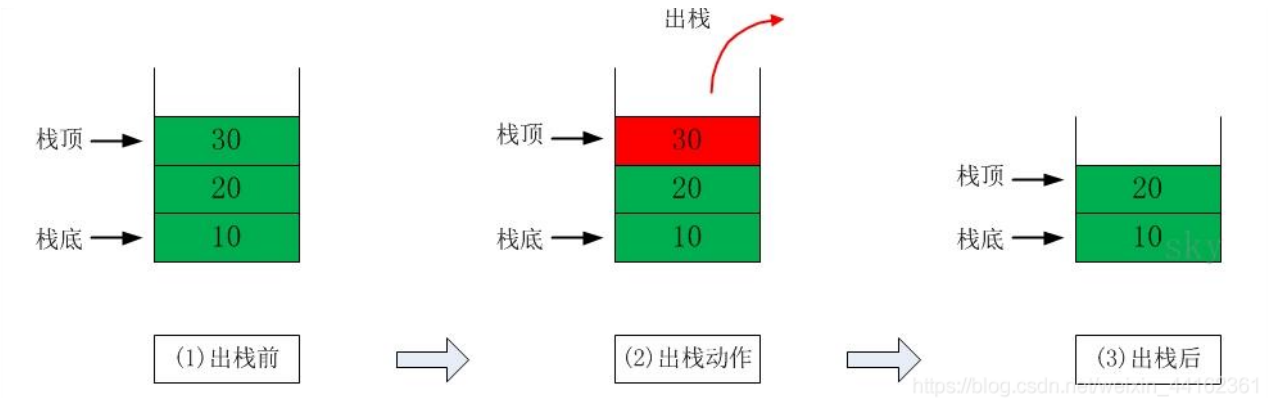
5.入栈

```c
void pushstack(st *s,selemtype x)
{
    if ((lenstack(s)) == s->stacksize)
    {
        exit(-1);
    }
    else
    {
        *(s->top) = x;//简引用
        s->top++;
        //*(s->top)++ = x;
    }
    printf("%d ", x);
}
```



6.出栈

```c
    void  popstack(st* s)
{
    int x;
    if (isempty(s))
    {
        printf("栈空! ");
        exit(-1);
    }
    else
    {
        --(s->top);
        x = *(s->top);
        printf("%d ", x);
    }
}
```



详细代码：

```c
#define _CRT_SECURE_NO_WARNINGS 1
/*加油! */


#include "test.h"

//两种存储结构 但是操作受限的线性表
/*
```

```
 * 1.顺序栈
 * 2.链栈
 */
//顺序栈的表示和实现 一组地址连续的存储的一次存放栈底到栈顶的数据元素 栈底一般在低地址端

#define MAXSIZE 100
typedef int selemtype;

typedef struct Node
{
    int data;
    struct Node* pNext;
} NODE, * PNODE;
typedef struct stack
{
    selemtype* base;//栈底指针
    selemtype* top;//栈顶指针
    int stacksize;//栈容量
}st;

//初始化 内存中开辟一段空间来使用 top == base 都指向栈底
void initstack(st *s)
{
    s->top = (int*)malloc( MAXSIZE*sizeof(int));//分配空间 大小为100 指向这个空间的首元素
    if (s->top == NULL)
    {
        printf("开辟失败");
        exit(-1);
    }
    s->base = s->top;
    s->stacksize = MAXSIZE;
    printf("栈已经开辟成功!\n");
}

//顺序栈是否为空
bool isempty(st *s)
{
    if (s->base == s->top)
    {
        return true;
        printf("栈空! ");
    }
    else
    {
        return false;
        printf("栈回收! ");
    }
}
//求栈len
int lenstack(st *s)
```

```
{
    return (s->top - s->base);
}
//清空顺序栈
void qkstack(st *s)
{
    if (s->base)
    {
        s->top = s->base;
        printf("栈已经清空了! ");
    }
}
//销毁栈
void destoryst(st *s)
{
    if (s->base)
    {
        free(s->base);
        s->stacksize = 0;
        s->base = s->top = NULL;
        printf("栈已经释放了! 收回内存");
    }
}
//入栈
void pushstack(st *s,selemtype x)
{
    if ((lenstack(s)) == s->stacksize)
    {
        exit(-1);
    }
    else
    {
        *(s->top) = x;//简引用
        s->top++;
        //*(s->top)++ = x;
    }
    printf("%d ", x);
}
//出栈
void  popstack(st* s)
{
    int x;
    if (isempty(s))
    {
        printf("栈空! ");
        exit(-1);
    }
    else
    {
        --(s->top);
        x = *(s->top);
```

```
        printf("%d ", x);
    }
}

int Find(st*s,int y)
{
    if (isempty(s))
    {
        printf("栈空! ");
        exit(-1);
    }
    for (int i = 0;i<lenstack(s);i++)
    {
        if (*(s->top) == y)
        {
            return i;
        }
        s->top++;
    }
    return -1;
}
int main()
{
    st s1;
    selemtype n;
    printf("初始化: \n");
    initstack(&s1);
    printf("请向栈中存入数据:\n");
    for (int i = 0;i<10;i++)
    {
        scanf("%d", &n);
        pushstack(&s1, n);
    }
    printf("入栈成功! \n");
    printf("栈len:\n");
    int len = lenstack(&s1);
    printf("len = %d\n", len);
    printf("出栈: \n");
    for (int i = 0; i < 10; i++)
    {
        popstack(&s1);
    }
    destoryst(&s1);
}
```

```
初始化：
栈已经开辟成功！
入栈个数：
8
请向栈中存入数据：
231 423 423 4241 12 32 12 1
231 423 423 4241 12 32 12 1 入栈成功！
栈len:
len = 8
出栈：
1 12 32 12 4241 423 423 231 栈已经释放了！ 收回内存
```

三、链栈的表示和实现

3.1链栈的表示

1.关于链栈的操作，它和链表有很大的相似之处，不同的是，栈只允许在栈顶进行入栈操作，和链表的头插法相同。

2.之所以要用链栈，是因为如果我们用顺序栈的话，需要提前申请一片内存空间，但是如果我们值存入少量的元素，那么这片内存空间难免会造成一定的浪费。如果使用链栈的话，我们只在入栈的时候进行内存的申请，然后再进行元素的存储，既可以进行动态的内存申请。根据实际入栈元素的多少申请所需的空间即可。

3.链栈和顺序栈的基本操作都是一样的，包括：初始化，求长度，判断栈是否为空，入栈，出栈，取栈顶元素。

3.2链栈的实现

1.先定义一个结构体，这个结构体中，包括栈中每个节点的节点值data，和指向下一个栈节点的指针next

```c
#include <stdio.h>
#include <stdlib.h>
typedef int ElemType;
//定义一个结构体栈
typedef struct StackNode
{
    ElemType data;
    struct StackNode *next;
}StackList;
typedef StackList *LStack;//声明一个指向这个结构体的指针类型Lstack
```

2.进行初始化，将链栈的顶处设置一个指针，用来指向这个栈，这个指针内不存储东西，那么刚开始就把这个栈顶指针设置为NULL即可。

```
//初始化链栈
void Init(LStack *s)
{
    (*s)=(LStack)malloc(sizeof(StackList));
    //p = (snode*)malloc(sizeof(snode)*MAXSIZE);
    (*s)->next=NULL;
}
```

3.判断栈是否为空，只需要判断栈顶指针的指针域是否为空即可。为空返回0，不为空返回1。

```
//判断栈是否为空
int Empty(LStack s)
{
    if(s->next==NULL)
    {
        return 0;//为空返回0,
    }
    return 1;    //不为空返回1,
}
```

4.求栈的长度，即从栈顶的下一个，也就是第一个元素开始，往下遍历，一直遍历到指针指向空，说明遍历完了，没遍历一个，就计数器加一，临时的指针变量往后移动一位p=p->next

```
//求栈中的元素数量，即长度
int Printf(LStack s)
{
    int count=0;
    LStack p;
    p=s->next;//指向第一个元素
    while(p){
        count++;
        p=p->next;
    }
    return count;
}
```

5.入栈，将值x，压入到栈中，我们首先声明一个指针变量，来存储这个元素值x，当然，因为我们要用这个指针变量存储东西，所以我们要先为这个指针变量申请内存空间malloc，然后存入。

```c
//入栈
void Push(LStack *s,ElemType x)
{
    LStack p;//声明一个指针变量
    p=(LStack)malloc(sizeof(StackList));//为这个指针变量分配空间
    p->data=x;          //将入站的元素值存入到这个指针的数据域
    p->next=(*s)->next;   //这两个步骤和顺序链表的操作相似。
    (*s)->next=p;
}
```

6.出栈，首先要判断栈是否为空，如果不为空，我们指针p指向栈中的第一个节点，然后用e记录下当前栈顶元素的值，接着将不存东西的栈顶指针的指针域指向指针p的下一个节点处，释放指针p即可

```c
//出栈
ElemType Pop(LStack *s)
{
    LStack p;//声明一个结构体类型的指针变量
    ElemType e;
    p=(*s)->next;//指向栈中的栈顶元素
    if(p==NULL)
    {
        return 0;
    }
    (*s)->next=p->next;//不为空的话就让栈顶元素变成当前栈顶p的下一个
    e=p->data;//存放以下删除的栈顶元素
    free(p);  //释放指针
    return e; //返回删除的栈顶指针的值
}
```

7.获取栈顶元素的值，首先要判断栈是否为空，如果不为空，直接返回栈中的第一个节点的值即可

```c
//取栈顶元素的值
ElemType GetTop(LStack s)
{
    if(Empty(s)){//如果栈顶元素不为空的话，则打印出栈顶元素
        return s->next->data;
    }
    return 0;
}
```

8.main

```c
int main()
{
    LStack s;
```

```
    int x;
    ElemType e;
    //初始化栈
    Init(&s);
    //判断栈是否为空
    x=Empty(s);
    if(x)
    {
        printf("栈空\n");
    }
    //扫描要入站的元素
    Push(&s,5);
    Push(&s,4);
    Push(&s,6);
    Push(&s,8);
    Push(&s,2);
    //获取栈的长度
    x=Printf(s);
    printf("栈的长度为%d\n",x);
    //获取当前栈的栈顶元素
    e=GetTop(s);
    if(e){//如果栈不为空的话输出
        printf("当前栈顶元素为: %d\n",e);
    }
    //删除当前栈的栈顶元素
    e=Pop(&s);
    if(x)//如果栈不为空的话进行出栈操作
    {
        printf("删除的元素为: %d\n",e);
    }
    //获取当前的栈顶元素
    e=GetTop(s);
    if(e){//如果栈不为空的话输出
        printf("当前栈顶元素为: %d\n",e);
    }
    x=Printf(s);
    printf("栈的长度为%d\n",x);
    return 0;
}
```

```
//#define _CRT_SECURE_NO_WARNINGS 1
///*加油! */
#include "test.h"
#define MAXSIZE 100
typedef int selemtype;

//通过链表表示栈 -- 链栈  栈的结点类型
/*
```

```
 * 链栈的指针方向是前驱元素
 *
 * 1.链表的头指针就是栈顶
 * 2.链栈不需要头节点
 * 3.基本不存在栈满的情况
 * 4.空栈相当于头指针指向空
 * 5.插入和删除仅在栈顶处执行
 *
 */
typedef struct snode
{
    selemtype data;
    struct snode* next;
}snode,*ls;

//初始化 就是开辟空间往里面存数据
void init(snode *s)
{
    s = NULL;
}

bool isempty(snode *s)
{
    if (s==NULL)
    {
        return false;
    }
    else
    {
        return true;
    }
}
void push(snode *s,int x)
{
    //栈里插入元素 栈顶 先找出一个空间 指针p指向给它 给数据域赋值 就成为了栈顶元素 在指向s的next域  然后在修改栈顶的指针p 然后s的链栈就创建完成
    snode* p;
    p = (snode*)malloc(sizeof(snode)*MAXSIZE);
    if (p==NULL)
    {
        printf("结点开辟失败! ");
        exit(-1);
    }
    //将数据给p指向的data域
    p->data = x;
    //将p的next指向s的栈顶
    p->next = s;
    //修改栈顶指针
    s = p;

}
```

```
59
60  void  pop(snode* s)
61  {
62      snode* p;
63      int x;
64      p = (snode*)malloc(sizeof(snode));
65      if (s == NULL)
66      {
67          printf("栈空! ");
68      }
69      x = p->data;
70      p = s;
71      s = s->next;
72      free(p);
73
74  }
75  //取栈顶元素
76  int gettop(snode*s)
77  {
78      if (s!= NULL)
79      {
80          return s->data;
81      }
82      return 1;
83  }
84  int main()
85  {
86
87      snode s;
88      printf("链栈初始化\n");
89      init(&s);
90      printf("链栈初始化成功\n");
91      printf("进行入栈! \n");
92      push(&s,1);
93      push(&s, 21);
94      push(&s, 31);
95      push(&s, 51);
96      printf("入栈成功! \n");
97      printf("进行出栈: \n");
98      pop(&s);
99      pop(&s);
100     pop(&s);
101     printf("出栈结束: \n");
102     //slen(&s);
103
104
105 }
```

四、链栈和顺序栈的总结

1.顺序栈：

> 结构体构造：
>
> 1. 栈顶指针 top
> 2. 栈底指针 base
> 3. 数据个数 size
>
> 特性：
>
> ~栈空：**top = base**
>
> ~栈满：**top - base = maxsize**
>
> 栈溢出：
>
> ~上溢报错：**top - base > maxsize**
>
> ~下溢结束：**top - base = maxsize**

2.链栈：

> 结构体构造：
>
> 1. 数据域
> 2. 结构体指针域
>
> 特性：
>
> 链栈的指针方向是前驱元素
> 1.链表的头指针就是栈顶
> 2.链栈不需要头节点
> 3.基本不存在栈满的情况
> 4.空栈相当于头指针指向空
> 5.插入和删除仅在栈顶处执行

> 首先从栈顶插入元素，开辟一个空间，造一个指针**p**指向改空间的的首地址
>
> 给数据域进行赋值，再指向链栈的**next**的域，然后修改栈顶指针**p**为**s,**链栈创建完成

都是通过动态分配函数开辟内存空间