10.23

栈、队列和哈希表

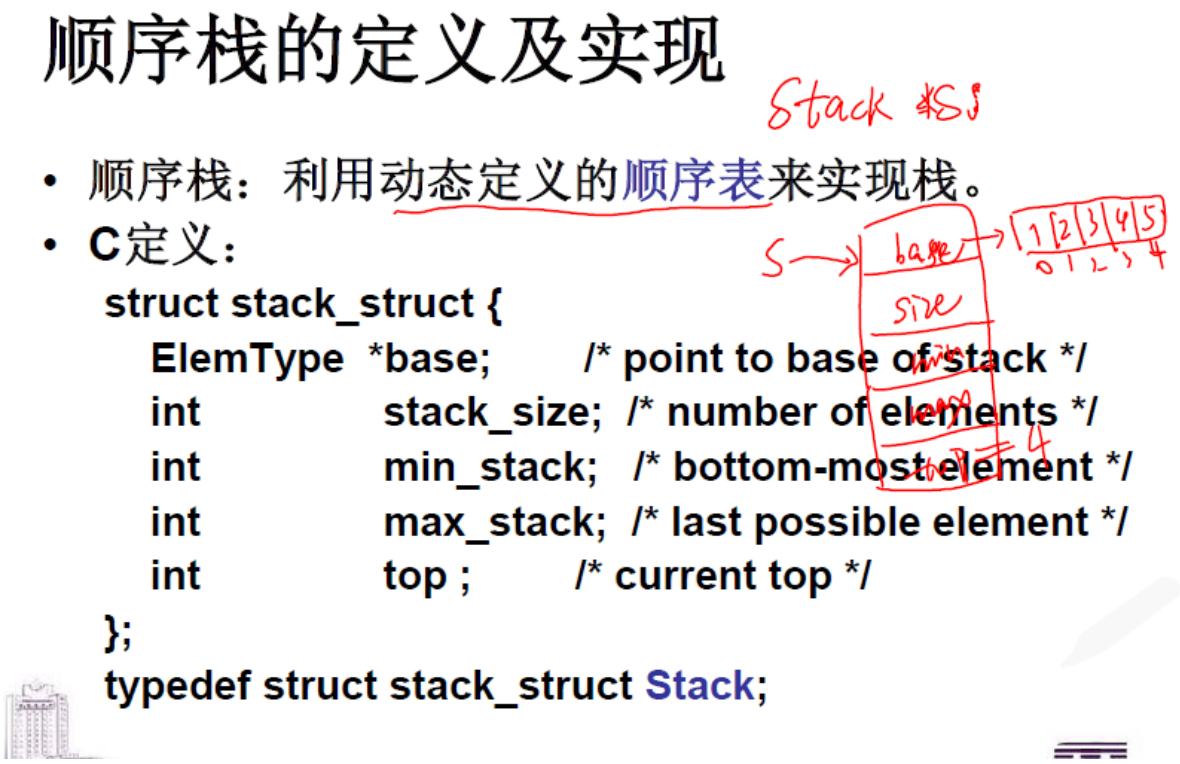
栈

栈顶top；栈底base/bottom

特性：先进后出

顺序栈：**top栈顶在尾部**

链栈：top栈顶在头部，在表尾时删除等操作不方便



判断栈空：s->top == -1　栈满：stack->top == this_stack->max_stack

初始化栈

```
Stack *CreateStack ( int how_many )
{
    Stack *pstk;
    assert ( how_many > 0 );  /* make sure the size is legal */
    pstk = (Stack *) malloc ( sizeof ( Stack ));
    if ( pstk == NULL )
        return ( NULL );
    pstk-->stack_size = how_many;
    pstk-->base = ( struct StkElement*) malloc ( how_many * sizeof ( struct
StkElement ));
    if ( pstk-->base == NULL )       /* error in allocating stack*/
         return ( NULL );
    pstk-->min_stack =0;
    pstk-->max_stack = how_many - 1;
    ClearStack(pstk);
    return (pstk);
```

```
}
```

清除栈

```
void ClearStack ( Stack *this_stack )
{
    this_stack-->top = 1;
}
```

销毁栈

```
void DestroyStack ( Stack *this_stack )
{
    ClearStack (this_stack);
    free(this_stack-->base);
    this_stack-->base=NULL;
}
```

随机访问栈元素

```
struct StkElement * ViewElement ( Stack
*this_stack, int which_element
{
    if ( this_stack-->top == 1 )
        return ( NULL );
    if ( this_stack-->top - which_element < 0 )
        return ( NULL );
    return ( &(( this_stack-->base)[this_stack-->top - which_element] ));
}
```

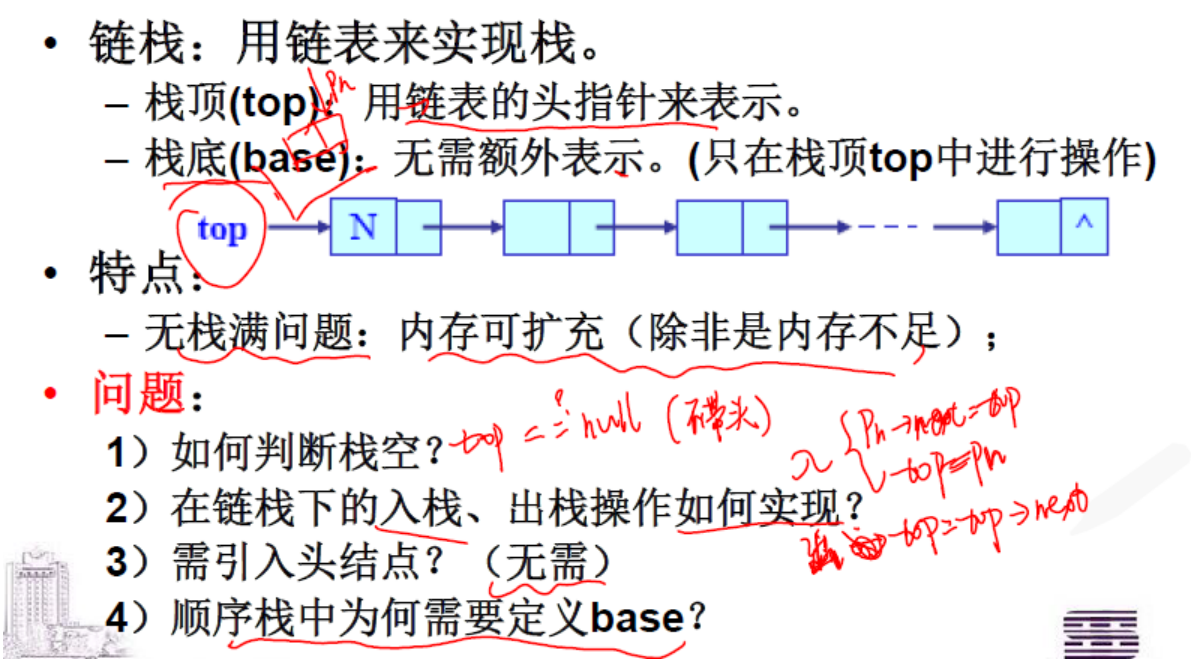
入栈

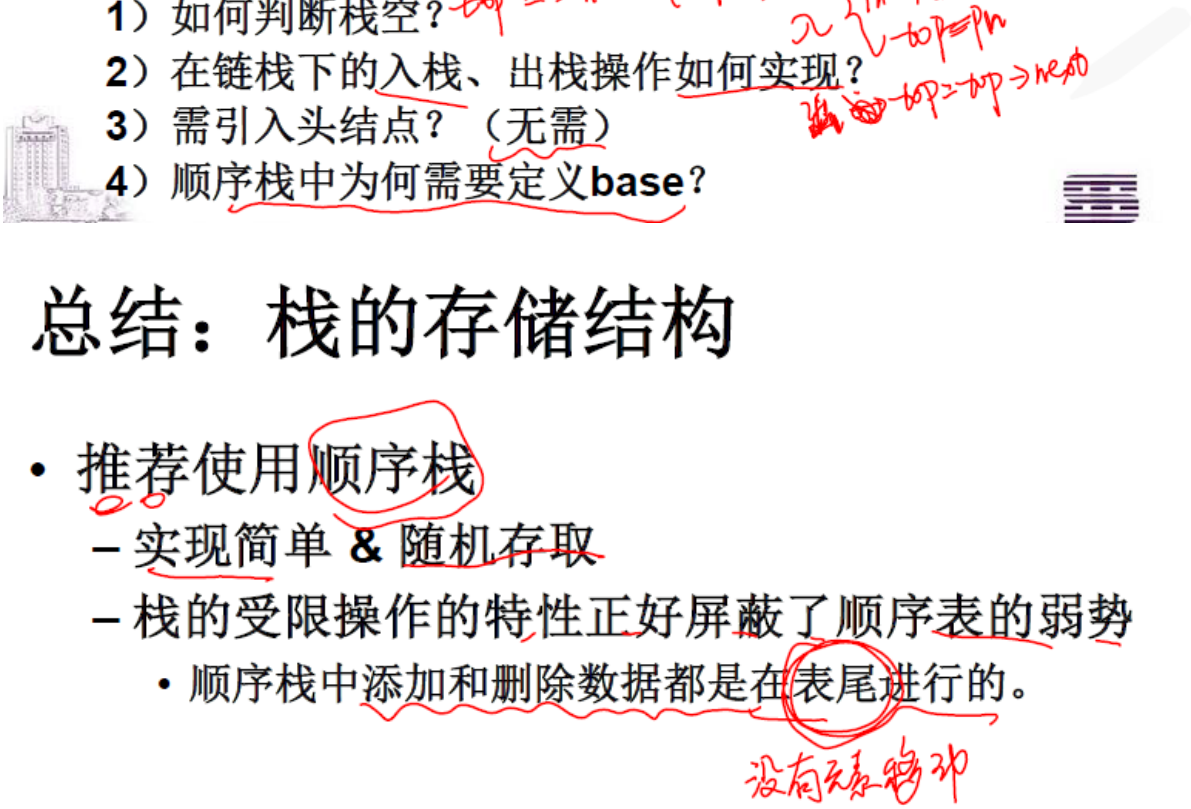
```
int PushElement ( Stack *this_stack, struct StkElement
* to_push
{
    /* is stack full? */
    if ( this_stack-->top == this_stack -->max_stack
        return ( 0 );
    this_stack-->top += 1;
    memmove ( &(( this_stack-->base )[this_stack -->top] ),
            to_push, sizeof ( struct StkElement ));
    return ( 1 );
}
```

出栈

```
int PopElement ( Stack *this_stack, struct StkElement *
destination )
{
    if ( this_stack-->top == 1 ) /* stack empty, return error*/
        return ( 0 );
    memmove ( destination,&(( this_stack-->base)
            [this_stack -->top] ), sizeof(struct StkElement
    this_stack-->top -=1;
    return ( 1 );
}
```

链栈





栈的应用：函数调用

```
long fact( long n)
{
    if (n==0) return 1; //递归结束条件
    else return n*fact(n-1);// 递归的规则
}
```

调用的顺序和返回的顺序是相反的，正好符合了“栈”的特点

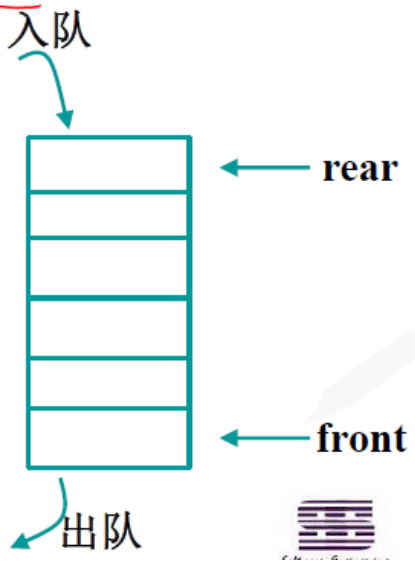
2020/10/23

队列

队头 front：是允许**删除**的一端

队尾rear：允许**插入**的一端

- 队列：
  - 是操作受限的特殊线性表。
  - 是限定在一端进行插入，而在另一端进行删除的线性表；
  - 队头(front)：允许删除的一端；
  - 队尾(rear)：允许插入的一端
- 特性：先进先出
- 存储结构：
  - 循环队列：顺序存储
  - 链队列：链式存储
- 队列的应用举例：操作系统的作业排队；
- 原子操作：
  - 创建空队列
  - 销毁已有队列
  - 查找直接后继和直接前驱
  - 入队
  - 出队



顺序队列

队空：Q.front == Q.rear

队满：Q.rear == MAXQSIZE

顺序队列存在假上溢问题，因此提出循环队列

循环队列

循环队列的定义及实现

- 利用顺序表来实现队列。约定front指向队列头元素，rear 指向队尾元素的下一位置。
- C定义：

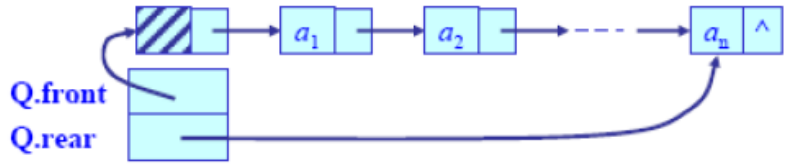
```
#define MAXQSIZE    100    /* 最大队列长度        */
typedef struct{
ElemType        *base;                  /* 存储空间    */
int             front;          /* 头指针，指向队列的头元素 */
int             rear;     /* 尾指针，指向队尾元素的下一个位
  置        */
} SqQueue;       /* 非增量式的空间分配 */
```

- 假上溢的解决
  - 将队列假想为首尾相接的环，即循环队列。
    - 入队：……，Q.rear = (Q.rear+1)%MAXQSIZE
    - 出队：……，Q.front = (Q.front+1)%MAXQSIZE
    - 队空条件：Q.front == Q.rear，由于出队Q.front追上了Q.rear
    - 队满条件：Q.front == Q.rear，由于入队Q.rear追上了Q.front
- 问题：队空和队满的判断条件一样

- 如何区分队空和队满？
  - 方案1：设标志位：不足在于需要额外对标志位的判断及维护
  - 方案2：在队列的结构中引入长度成员，在初始化队列、入队、出队操作中维护这个成员。
  - 方案3：少用一个元素空间，即队满的条件如下：

(Q.rear+1)% MAXQSIZE == Q.front

- 链队列：用双向链表来实现队列。
  - 队头(front)：用链表的头指针来表示；
  - 队尾(rear)：用链表的尾指针来表示
- 特点：
  - 无队列满问题：内存可扩充（除非是内存不足）；



- 问题：
  1）是否需引入头结点？（需要。特殊：对于空队列的入队）
  2）在链队列下的队列初始化、入队、出队算法如何实现？
  3）如何判断队空？

链队列的C定义：

```
struct Node {
   struct Node    *prev;  /* link to previous node */
   struct Node    *next;  /* link to next node */
   void           *pdata; /* generic pointer to data */
};
typedef struct Node *Link;
/* a linked list data structure */
struct List {
   Link           LHead; /* 队头指针，指向头元素          */
   Link           LTail; /* 队尾指针，指向队尾元素        */
   unsigned int   LCount;
   void * ( * LCreateData )    ( void * );
   int    ( * LDeleteData )    ( void * );
   int    ( * LDuplicatedNode ) ( Link, Link );
   int    ( * LNodeDataCmp )   ( void *, void * );
};
```

循环队列需要额外区分队空和队满，因此推荐使用链队列（直接利用带头节点的双向链表实现）

哈希表

哈希表属于**线性结构**

作用：1）按值查找 2）物以类聚

5.1 什么是哈希表

5.2 哈希表的构造

5.3 冲突解决方法

5.4 哈希表的查找及性能分析

Hash 表（哈希表）是一种线性结构。是有限个数据项组成的序列，记作 (a1,a2, ... ,an)
Hash 表（哈希表）可以建立数据项的关键字和其逻辑存储位置之间的对应关系。即： HashKey = H (key)Key （关键字 ）：是数据项（或记录）中某个分量的值，它可以用来标识一个数据元素（或记录）。
主关键字 ：唯一标识一个记录的关键字
HashKey （hash 键 ）：也称为槽。它是关键字的"像"，是关键字在该表中的逻辑存储位置。 （必须合法 ，∈∈[0,hash 表长 1]
h （Hash 函数）：是一个映像，它将一组关键字映像到一个有限的、地址连续的地址区间上。（ h : Key --> HashKey ）。
Hash 表的**构造过程** ，是将关键字映像到其逻辑存储位置（或 hash 键）的过程。
冲突 ：两个不同的数据项映像到同一个 HashKey 上。即：Key1≠Key2 ，但 H(Key1)= H(Key2) 。

- 利用动态定义的顺序表来实现基本hash表。

```
#define Table_Size 100/*分配空间的大小*/
Typedef HashTable_Struct{
   ElemType *elem;      /*顺序表的存储空间*/
   int len;             /*实际长度*/
   int TableSize ; /*当前分配的空间大小*/
};
typedef struct HashTable_Struct HashTable;
```

哈希表适用场景

- 要求在内存中存储具有线性结构的数据集合
- 集合中的数据项的数量预先无法确定
- 要求能快速、近似随机的访问数据项（按值查找）

- 散列函数的通用形式：
  - Hashkey = calculated-key(key) % Table_Size
- 完美散列函数：不同数据项，对应的HashKey也不同。（永远不会出现冲突）。
- 实际：几乎不可能构造出完全散列函数，因此应选择良好的通用算法。
- 良好的散列函数：HashPJW, ElfHash
  1）计算快速，Hashkey分布均匀；
  2）必须弥补可能出现在输入数据中的聚集。

“冲突”≠“聚集”

处理冲突的三种方法

- 线性再散列法
  - di 为线性的
- 非线性再散列法
  - di 为非线性的
- 外部拉链法
  - 将散列表看做一个链表数组

- 负载因子α：hash表中的数据项个数(n)除以可用槽的总数(HashTable_Size)。
  - 负载因子越大，冲突概率越大。

总结：再散列法

- 优点：
  - 容易进行动态编码；
  - 负载因子较低并且不太可能执行删除操作的情况下，它的速度足够快；
  - 通常认为，负载因子α>0.5时，再散列将不是一种切实可行的解决方案。
- 适用场景：
  - 只应该在快速而又随性的情况下，或者在快速原型化的环境中使用再散列法解决冲突。
  - 若不满足上述需求，则应该使用外部拉链法。

外部拉链法

- 将散列表看做一个链表数组。
  - Hash表中的每个槽要么为空，要么指向一个链表。
- 可以通过将数据项添加到链表中的方法来解决冲突：将所有hashkey相同的数据项存储在同一链表中。（“聚集”的效果）
- 解决冲突的代价：
  - 不会超过向链表中添加一个结点(采用“头插法”)
  - 无需执行再散列。
- 与前面的2种再散列法不同之处：
  - 外部拉链法可以容纳的元素只取决于可用的内存大小；
  - 而再散列法中hash表的最大表项取决于表的大小。

用外部拉链法解决冲突时哈希表的实现

- 利用链表数组来描述。

```
#define Table_Size 100/*分配空间的大小*/
Typedef HashTable_Struct{
   Link *elem;      /*顺序表的存储空间*/
   int len;          /*实际长度*/
   int TableSize ; /*当前分配的空间大小*/
};
typedef struct HashTable_Struct HashTable;
```

总结：

- 优点：
  - 平均查找时间=链表长度/2+1（链表非空时）；
- 缺点：
  - 需要多一些的存储空间，因为每次探查时都需要添加结点，而不仅仅是数据项。但是，在硬件便宜的现在，可以忽略不计，故该方法现在用得最多。

平均查找长度ASL

关注：探查次数

- 采用ASL (Avarage Search Length)来衡量Hash表的性能。

Case 1：采用线性再散列法解决冲突:di=1,2,3, …

	0	1	2	3	4	5	6	7	8	9	10
	55	01	23	14	68	11	82	36	19		
探查次数：	1	1	2	1	3	6	2	5	1		

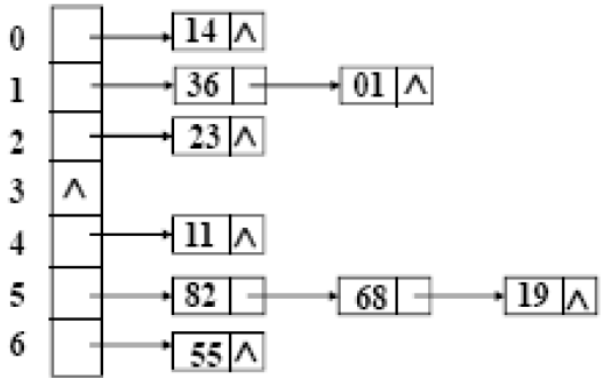
- 查找成功时的平均查找长度(查找概率相等):

  ASL=(1·4+2·2+3·1+5·1+6·1)/9=22/9

- 查找不成功时的平均查找长度(查找概率相等):

  ASL=(10+9+8+7+6+5+4+3+2+1+2)/11=56/11

Case 2:



- 查找成功(查找概率相等)时：

  ASL=(1·6+2·2+3·1)/9=13/9

- 查找不成功时的平均查找长度(查找概率相等):

  ASL=(2+3+2+1+2+4+2)/7=16/7