Homework2

ChangeTwoInteger

问题分析

实现两个整数的交换，可以采用temp作为中间变量，也可以采用异或运算，还可以采用四则运算的方法

算法思路

- temp法，可用temp暂存一个值，进行交换
- 异或法，可利用异或的同0异1的性质，进行交换
- 四则运算法，可使用加法与减法的结合，或乘法与除法的结合

算法实现与分析

算法实现

- temp法代码如下:

```
    // We assume num1, num2 and temp have been initialized, and all of their type are ElementType
    temp = num1;
    num1 = num2;
    num2 = num1;
```

- 异或法代码如下:

```
// We assume num1, num2 and temp have been initialized, and all of their type are ElementType
    num1 = num1 ^ num2;
    num2 = num1 ^ num2;
    num1 = num1 ^ num2;
```

- 四则运算法代码如下:
    - 加减法代码如下

```
// We assume num1, num2 and temp have been initialized, and all of their type are ElementType
    num1 = num1 + num2;
    num2 = num1 - num2;
    num1 = num1 - num2;
```

    - 乘除法代码如下

```
// We assume num1, num2 and temp have been initialized, and all of their type are ElementType
    num1 = num1 * num2;
    num2 = num1 / num2;
    num1 = num1 / num2;
```

算法分析

- 四则运算法与异或法相比，存在溢出风险
- temp法与异或法相比，需额外设置一个变量
- 异或法因采用位运算，故速度较快

GetTwoMin

问题分析

得到序列中的两个最小值，根据不同的数据存储方式有不同做法，如最小堆的两次GetMin操作，如设置辅助栈进行两次Pop操作

算法思路

- 最小堆
    - 最小值，一直为Data[1]处
    - 管理最小堆，首先需要判断堆满或空，故设置了IsEmpty,IsFull函数
    - 插入之前，需创造一个空堆，故设计了CreateMinHeap函数
    - 当在MinHeap中插入一个key时，根据树的性质，进行下滤，找到对应位置，故对于下滤操作设置了PercDown函数
    - 插入key时，设置了Insert函数，并在函数内部调用PercDown函数
    - 将序列建造为最小堆，即进行多次插入，故设置了BuildMinHeap函数，进行最小堆的创建
    - 删除最小元素时，我们用堆中最后一个数据进行下滤，调整其位置，并处理堆的规模，故设置了DeleteMin函数
- 辅助栈
    - 我们可以设计一个包含两个栈的栈，DataStack存所有数据，minStack存最小数据，最小数据对应index放在栈顶
    - 入栈：
        - First Element: 第一个元素进入栈DataStack时候，其对应index 0 进入minStack
        - Other Elements X: 若X<=DataStack[minStack[0]],则X的index入minStack栈
    - 出栈
        - 当index of DataStack[0]==minStack[0],minStack[0]出栈，此时新栈顶为DataStack最小元素index
    - FindMin
        - 取DataStack[minStack[0]]，通过调用peek函数获得

算法实现与分析

算法实现

最小堆法实现代码如下，实现题意仅需调用两次DeleteMin函数即可

```
typedef struct HNode* Heap;
struct HNode{
    ElementType* Data;
    int Size;
    int Capacity;
};
typedef Heap MinHeap;
#define MinData 1001
#define MinData -1
#define Error -6

int IsEmpty(Heap H){
    return (H->Size == 0);
}
int IsFull(Heap H){
    return (H->Capacity == H->Size);
}

MinHeap CreateMinHeap(int MaxSize){
    MinHeap H = (MinHeap)malloc(sizeof(struct HNode));
    H->Data = (ElementType*)malloc((MaxSize + 1) * sizeof(ElementType));
    H->Size = 0;
    H->Data[0] = MinData;
    H->Capacity = MaxSize;
    return H;
}

void InsertMinHeap(MinHeap H, ElementType X){
    //将元素X插入最小堆H，其中H->Data[0]已经定义为哨兵
    int i;
    if (!IsFull(H))
    {
        i = ++H->Size;
        for (; H->Data[i / 2] < X; i /= 2)
            H->Data[i] = H->Data[i / 2];
        H->Data[i] = X;
    }
}

void PercDownMinHeap(MinHeap H, int p){
    //下滤：将H中以H->Data[p]为根的子堆调整为最小堆
    int Parent, Child;
    ElementType X;
    X = H->Data[p];
    for (Parent = p; Parent * 2 <= H->Size; Parent = Child)
    {
        Child = Parent * 2;
```

```
        if ((Child != H->Size) && (H->Data[Child] > H->Data[Child + 1]))
            Child++;//Child指向左右子结点的较小者
        if (X <= H->Data[Child])
            break;
        else
            H->Data[Parent] = H->Data[Child];
    }
    H->Data[Parent] = X;
}

void BuildMinHeap(MinHeap H)//建造最小堆{
    调整H->Data[]中的元素，使满足最小堆的有序性
    假设所有H->Size个元素已经存在H->Data[]中
    int i;
    for (i = H->Size / 2; i > 0; i--)
        PercDownMinHeap(H, i);
}
ElementType DeleteMin(MinHeap H){
    //从最小堆H中取出键值为最小的元素，并删除一个结点
    int Parent, Child;
    ElementType MinItem, X;
    if (!IsEmpty(H)){
        MinItem = H->Data[1];//取出根结点存放的最小值
        X = H->Data[H->Size--];
        //用最小堆中最后一个元素从根结点开始向上过滤下层结点,当前堆的规模要减小
        for (Parent = 1; Parent * 2 <= H->Size; Parent = Child)
        {
            Child = Parent * 2;
            if ((Child != H->Size) && (H->Data[Child] > H->Data[Child + 1]))
                Child++;//Child指向左右子结点的较小者
            if (X <= H->Data[Child])
                break;//找到了合适位置
            else
                H->Data[Parent] = H->Data[Child];//下滤X
        }
        H->Data[Parent] = X;
        return MinItem;
    }
    else
        return Error;
}
```

辅助栈实现代码如下,实现题意仅需调用FindMin1和FindMin2即可

```
    class MinStack {
    private Stack<Integer> DataStack;
    private Stack<Integer> minStack;
    public MinStack() {
        minStack = new Stack<>();
        DataStack = new Stack<>(); //存储原始数据
    }

    public void push(int x) {
```

```
        DataStack.push(x);
        if (minStack.isEmpty() || x < minStack.peek()) {
            minStack.push(x);
            //栈空或元素更小时X入栈，若X==minStack.peek（），不更新
        } else {
            minStack.push(minStack.peek());
        }
    }

    public void pop() {
        DataStack.pop();
        minStack.pop();
    }

    public int FindMin1() {
        return DataStack[minStack.peek()];
    }
    public int FindMin2() {
        return DataStack[minStack.peek()-1];
    }
}
```

算法分析

- 从时间复杂度上分析，最小堆和最小栈的时间复杂度都为O(log n)
- 从空间复杂度上分析，最小堆需要维护一个树形结构，空间复杂度较高，最小栈只需要维护两个堆栈，因此空间消耗较小
- 需要在一个数据集中快速找到最小元素，最小堆更优；若同时需要优化空间，则最小栈更优