0.前言

堆（Heap）与栈（Stack）是开发人员必须面对的两个概念，在理解这两个概念时，需要放到具体的场景下，因为不同场景下，堆与栈代表不同的含义。一般情况下，有两层含义：

（1）程序 内存　布局场景下，堆与栈表示两种内存管理方式；

（2） 数据结构　场景下，堆与栈表示两种常用的数据结构。

1.程序内存分区中的堆与栈

1.1 栈简介

栈由操作系统自动分配释放 ，用于存放函数的参数值、局部变量等，其操作方式类似于数据结构中的栈。参考如下代码：

```
int main() {
        int b;                          //栈
        char s[] = "abc";       //栈
        char *p2;                       //栈
}
```

其中函数中定义的局部变量按照先后定义的顺序依次压入栈中，也就是说相邻变量的地址之间不会存在其它变量。栈的内存地址生长方向与堆相反，由高到底，所以后定义的变量地址低于先定义的变量，比如上面代码中变量 s 的地址小于变量 b 的地址，p2 地址小于 s 的地址。栈中存储的数据的生命周期随着函数的执行完成而结束。

1.2 堆简介

堆由开发人员分配和释放， 若开发人员不释放，程序结束时由 OS 回收，分配方式类似于链表。参考如下代码：

```

int main() {
        // C 中用 malloc() 函数申请
        char* p1 = (char *)malloc(10);
        cout<<(int*)p1<<endl;           //输出：00000000003BA0C0

        // 用 free() 函数释放
        free(p1);

        // C++ 中用 new 运算符申请
        char* p2 = new char[10];
        cout << (int*)p2 << endl;               //输出：00000000003BA0C0

```

```
14          // 用 delete 运算符释放
15          delete[] p2;
    }
```

其中 p1 所指的 10 字节的内存空间与 p2 所指的 10 字节内存空间都是存在于堆。堆的内存地址生长方向与栈相反，由低到高，但需要注意的是，后申请的内存空间并不一定在先申请的内存空间的后面，即 p2 指向的地址并不一定大于 p1 所指向的内存地址，原因是先申请的内存空间一旦被释放，后申请的内存空间则会利用先前被释放的内存，从而导致先后分配的内存空间在地址上不存在先后关系。堆中存储的数据若未释放，则其生命周期等同于程序的生命周期。

关于堆上内存空间的分配过程，首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序。另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确地释放本内存空间。由于找到的堆节点的大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表。

1.3 堆与栈区别

堆与栈实际上是操作系统对进程占用的内存空间的两种管理方式，主要有如下几种区别：
（1）管理方式不同。栈由操作系统自动分配释放，无需我们手动控制；堆的申请和释放工作由程序员控制，容易产生内存泄漏；

（2）空间大小不同。每个进程拥有的栈大小要远远小于堆大小。理论上，进程可申请的堆大小为虚拟内存大小，进程栈的大小 64bits 的 Windows 默认 1MB，64bits 的 Linux 默认 10MB；

（3）生长方向不同。堆的生长方向向上，内存地址由低到高；栈的生长方向向下，内存地址由高到低。

（4）分配方式不同。堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。动态分配由 `alloca()` 函数分配，但是栈的动态分配和堆是不同的，它的动态分配是由操作系统进行释放，无需我们手工实现。

（5）分配效率不同。栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是由C/C++提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。

（6）存放内容不同。栈存放的内容，函数返回地址、相关参数、局部变量和寄存器内容等。当主函数调用另外一个函数的时候，要对当前函数执行断点进行保存，需要使用栈来实现，首先入栈的是主函数下一条语句的地址，即扩展指针寄存器的内容（EIP），然后是当前栈帧的底部地址，即扩展基址指针寄存器内容（EBP），再然后是被调函数的实参等，一般情况下是按照从右向左的顺序入栈，之后是被调函数的局部变量，注意静态变量是存放在数据段或者BSS段，是不入栈的。出栈的

顺序正好相反，最终栈顶指向主函数下一条语句的地址，主程序又从该地址开始执行。堆，一般情况堆顶使用一个字节的空间来存放堆的大小，而堆中具体存放内容是由程序员来填充的。

从以上可以看到，堆和栈相比，由于大量malloc()/free()或new/delete的使用，容易造成大量的内存碎片，并且可能引发用户态和核心态的切换，效率较低。栈相比于堆，在程序中应用较为广泛，最常见的是函数的调用过程由栈来实现，函数返回地址、EBP、实参和局部变量都采用栈的方式存放。虽然栈有众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，主要还是用堆。

无论是堆还是栈，在内存使用时都要防止非法越界，越界导致的非法内存访问可能会摧毁程序的堆、栈数据，轻则导致程序运行处于不确定状态，获取不到预期结果，重则导致程序异常崩溃，这些都是我们编程时与内存打交道时应该注意的问题。

2.数据结构中的堆与栈

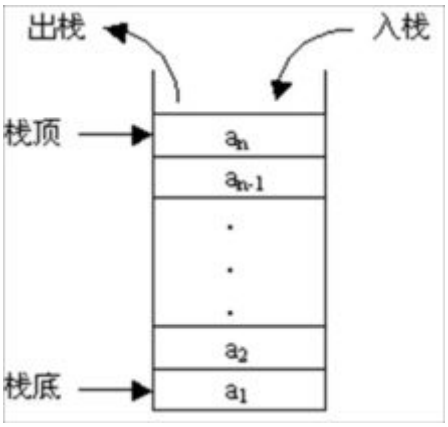
数据结构中，堆与栈是两个常见的数据结构，理解二者的定义、用法与区别，能够利用堆与栈解决很多实际问题。

2.1 栈简介

栈是一种运算受限的线性表，其限制是指只仅允许在表的一端进行插入和删除操作，这一端被称为栈顶（Top），相对地，把另一端称为栈底（Bottom）。把新元素放到栈顶元素的上面，使之成为新的栈顶元素称作进栈、入栈或压栈（Push）；把栈顶元素删除，使其相邻的元素成为新的栈顶元素称作出栈或退栈（Pop）。这种受限的运算使栈拥有“先进后出”的特性（First In Last Out），简称FILO。

栈分顺序栈和链式栈两种。栈是一种线性结构，所以可以使用数组或链表（单向链表、双向链表或循环链表）作为底层数据结构。使用数组实现的栈叫做顺序栈，使用链表实现的栈叫做链式栈，二者的区别是顺序栈中的元素地址连续，链式栈中的元素地址不连续。

栈的结构如下图所示：



栈的基本操作包括初始化、判断栈是否为空、入栈、出栈以及获取栈顶元素等。下面以顺序栈为例，使用 C++ 给出一个简单的实现。

```
1
```

```c

#include<stdio.h>
#include<malloc.h>

#define DataType int
#define MAXSIZE 1024
struct SeqStack {
        DataType data[MAXSIZE];
        int top;
};

//栈初始化,成功返回栈对象指针，失败返回空指针NULL
SeqStack* initSeqStack() {
        SeqStack* s=(SeqStack*)malloc(sizeof(SeqStack));
        if(!s) {
                printf("空间不足\n");
                return NULL;
        } else {
                s->top = -1;
                return s;
        }
}

//判断栈是否为空
bool isEmptySeqStack(SeqStack* s) {
        if (s->top == -1)
                return true;
        else
                return false;
}

//入栈，返回-1失败，0成功
int pushSeqStack(SeqStack* s, DataType x) {
        if(s->top == MAXSIZE-1)
        {
                return -1;//栈满不能入栈
        } else {
                s->top++;
                s->data[s->top] = x;
                return 0;
        }
}

//出栈，返回-1失败，0成功
int popSeqStack(SeqStack* s, DataType* x) {
        if(isEmptySeqStack(s)) {
                return -1;//栈空不能出栈
        } else {
                *x = s->data[s->top];
                s->top--;
```

```
                return 0;
        }
}

//取栈顶元素，返回-1失败，0成功
int topSeqStack(SeqStack* s,DataType* x) {
        if (isEmptySeqStack(s))
                return -1;      //栈空
        else {
                *x=s->data[s->top];
                return 0;
        }
}

//打印栈中元素
int printSeqStack(SeqStack* s) {
        int i;
        printf("当前栈中的元素:\n");
        for (i = s->top; i >= 0; i--)
                printf("%4d",s->data[i]);
        printf("\n");
        return 0;
}

//test
int main() {
        SeqStack* seqStack=initSeqStack();
        if(seqStack) {
                //将4、5、7分别入栈
                pushSeqStack(seqStack,4);
                pushSeqStack(seqStack,5);
                pushSeqStack(seqStack,7);

                //打印栈内所有元素
                printSeqStack(seqStack);

                //获取栈顶元素
                DataType x=0;
                int ret=topSeqStack(seqStack,&x);
                if(0==ret) {
                        printf("top element is %d\n",x);
                }

                //将栈顶元素出栈
                ret=popSeqStack(seqStack,&x);
                if(0==ret) {
                        printf("pop top element is %d\n",x);
                }
```

```
        }
        return 0;
}

```

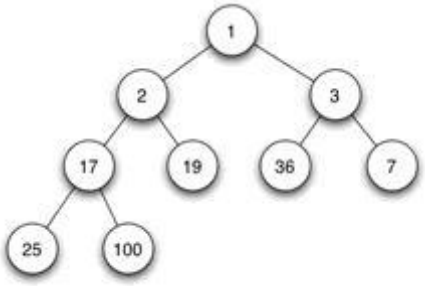
运行上面的程序，输出结果：

```
当前栈中的元素：
   7   5   4
top element is 7
pop top element is 7
```

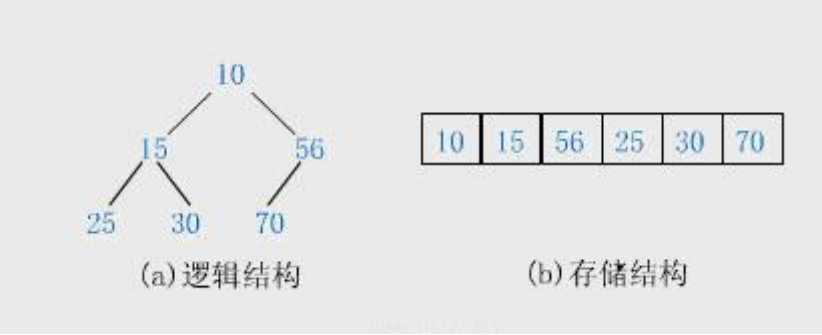
2.2 堆简介

2.2.1 堆的性质

堆是一种常用的树形结构，是一种特殊的完全二叉树，当且仅当满足所有节点的值总是不大于或不小于其父节点的值的完全二叉树被称之为堆。堆的这一特性称之为堆序性。因此，在一个堆中，根节点是最大（或最小）节点。如果根节点最小，称之为小顶堆（或小根堆），如果根节点最大，称之为大顶堆（或大根堆）。堆的左右孩子没有大小的顺序。下面是一个小顶堆示例：



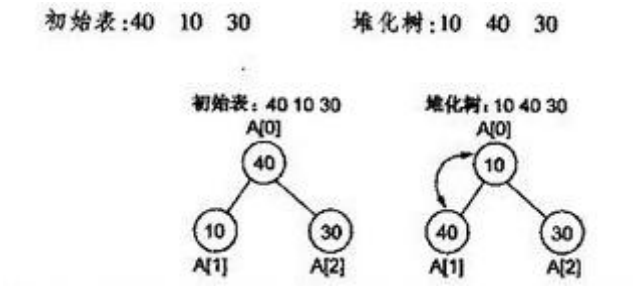
堆的存储一般都用数组来存储堆，i节点的父节点下标就为$(i-1)/2$。它的左右子节点下标分别为 $2*i+1$ 和 $2*i+2$。如第0个节点左右子节点下标分别为1和2。



2.2.2 堆的基本操作

- 建立

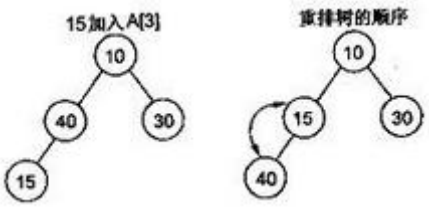
以最小堆为例，如果以数组存储元素时，一个数组具有对应的树表示形式，但树并不满足堆的条件，需要重新排列元素，可以建立“堆化”的树。



- 插入

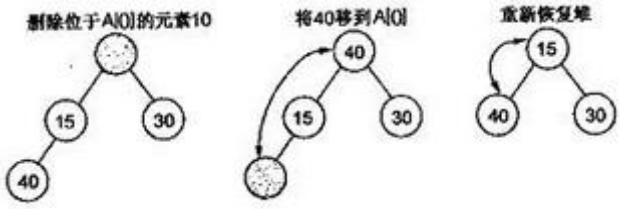
将一个新元素插入到数组末尾，如果新构成的二叉树不满足堆的性质，需要将新元素在其到堆顶的路径上，找到属于自己的位置，即进行上浮操作。

下图演示了插入 15 时，堆的调整。



- 删除

删除一个元素总是发生在堆顶，因为堆顶的元素是最小的（小顶堆中）。数组中最后一个元素用来填补空缺位置，然后对顶部元素进行下沉，如果左右孩子有比自己小的，则选择选择最小的那个进行交换。重复进行下沉操作，以满足堆的条件。



2.2.3 堆操作实现

（1）插入实现

每次插入都是将新数据放在数组最后。可以发现从这个新数据的父节点到根节点必然为一个有序的数列，现在的任务是将这个新数据插入到这个有序数据中，这就类似于直接插入排序中将一个数据并入到有序区间中，这是节点“上浮”调整。不难写出插入一个新数据时堆的调整代码：

```
1
2  // 新加入i节点,其父节点为(i-1)/2
3  // 参数：a：数组，i：新插入元素在数组中的下标
4  void minHeapFixUp(int a[], int i) {
5      int j, temp;
6      temp = a[i];
7      j = (i-1)/2;       //父节点
       while (j >= 0 && i != 0) {
```

```
        if (a[j] <= temp)//如果父节点不大于新插入的元素，停止寻找
            break;
        a[i]=a[j];                //把较大的子节点往下移动,替换它的子节点
        i = j;
        j = (i-1)/2;
    }
    a[i] = temp;
}
```

因此，插入数据到最小堆时：

```
// 在最小堆中加入新的数据data
// a：数组，index：插入的下标，
void minHeapAddNumber(int a[], int index, int data) {
    a[index] = data;
    minHeapFixUp(a, index);
}
```

（2）删除实现

按照堆删除的说明，堆中每次都只能删除第0个数据。为了便于重建堆，实际的操作是将数组最后一个数据与根节点交换，然后再从根节点开始进行一次从上向下的调整。

调整时先在左右儿子节点中找最小的，如果父节点不大于这个最小的子节点说明不需要调整了，反之将最小的子节点换到父节点的位置。此时父节点实际上并不需要换到最小子节点的位置，因为这不是父节点的最终位置。但逻辑上父节点替换了最小的子节点，然后再考虑父节点对后面的节点的影响。堆元素的删除导致的堆调整，其整个过程就是将根节点进行“下沉”处理。下面给出代码：

```

// minHeapFixDown 小顶堆结点下沉操作。
// a 为数组，len 为结点总数；从 index 结点开始调整，index 从 0 开始计算 index 其子
void minHeapFixDown(int a[],int len,int index) {
        // index 为叶子节点不用调整。
        if(index>(len/2-1)) return;
        int tmp=a[index];
        lastIndex=index;

        // 当下沉到叶子节点时，就不用调整了。
        while(index<=len/2-1) {
                // 如果左子节点小于待调整节点
                if(a[2*index+1]<tmp) {
                        lastIndex = 2*index+1;
                }
                //如果存在右子节点且小于左子节点和待调整节点
                if(2*index+2<len && a[2*index+2]<a[2*index+1]&& a[2*index+
```

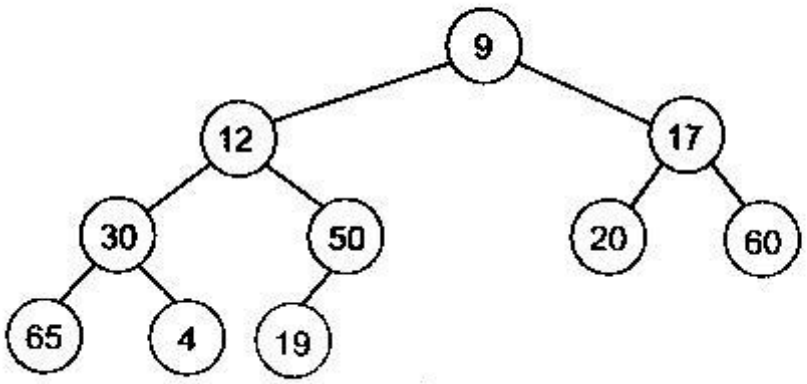
```
                        lastIndex=2*index+2;
                }
                //如果左右子节点有一个小于待调整节点，选择最小子节点进行上浮
                if(lastIndex!=index) {
                        a[index]=a[lastIndex];
                        index=lastIndex;
                } else break;                //  否则待调整节点不用下沉调整
        }
        //  将待调整节点放到最后的位置。
        a[lastIndex]=tmp;
}
```

根据堆删除的下沉思想，可以有不同版本的代码实现，以上是和孙凛同学一起讨论出的一个版本，在这里感谢他的参与，读者可另行给出。个人体会，这里建议大家根据对堆调整过程的理解，写出自己的代码，切勿看示例代码去理解算法，而是理解算法思想写出代码，否则很快就会忘记。

（3）建堆

有了堆的插入和删除后，再考虑下如何对一个数据进行堆化操作。要一个一个的从数组中取出数据来建立堆吧，不用！先看一个数组，如下图：

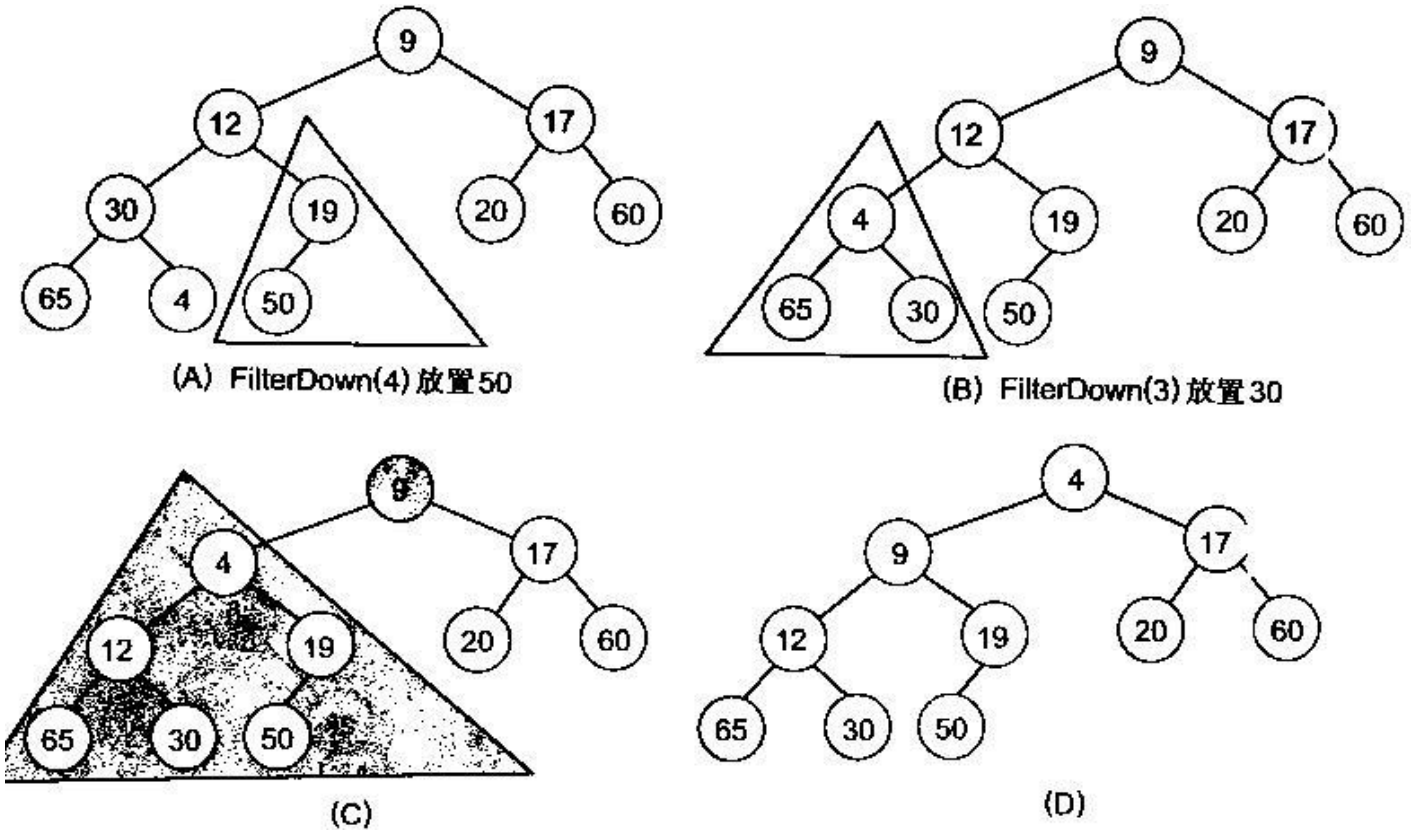
int A[0] = {9,12,17,30,50,20,60,65,4,49};



初始表

很明显，对叶子结点来说，可以认为它已经是一个合法的堆了，即 20，60，65，4，49 都分别是一个合法的堆。只要从 A[4]=50 开始向下调整就可以了。然后再取 A[3] = 30，A[2] = 17，A[1] = 12，A[0] = 9 分别作一次向下调整操作就可以了。

下图展示了这些步骤：



写出堆化数组的代码：

```
// makeMinHeap 建立最小堆。
// a:数组，n：数组长度。
void makeMinHeap(int a[], int n) {
    for (int i = n/2-1; i >= 0; i--) {
        minHeapFixDown(a, i, n);
    }
}
```

2.2.4 堆的具体应用——堆排序

堆排序（Heapsort）是堆的一个经典应用，有了上面对堆的了解，不难实现堆排序。由于堆也是用数组来存储的，故对数组进行堆化后，第一次将A[0]与A[n - 1]交换，再对A[0…n-2]重新恢复堆。第二次将A[0]与A[n – 2]交换，再对A[0…n - 3]重新恢复堆，重复这样的操作直到A[0]与A[1]交换。由于每次都是将最小的数据并入到后面的有序区间，故操作完成后整个数组就有序了。有点类似于直接选择排序。

因此，完成堆排序并没有用到前面说明的插入操作，只用到了建堆和节点向下调整的操作，堆排序的操作如下：

```

// array:待排序数组，len：数组长度
void heapSort(int array[],int len) {
        // 建堆
        makeMinHeap(array,len);
```

```
6
7           // 最后一个叶子节点和根节点交换，并进行堆调整，交换次数为len-1次
8           for(int i=len-1;i>0;--i) {
9                   //最后一个叶子节点交换
10                  array[i]=array[i]+array[0];
11                  array[0]=array[i]-array[0];
12                  array[i]=array[i]-array[0];
13
14          // 堆调整
15                  minHeapFixDown(array, 0, len-i-1);
16          }
    }
```

（1）稳定性。堆排序是不稳定排序。

（2）堆排序性能分析。由于每次重新恢复堆的时间复杂度为O(logN)，共N-1次堆调整操作，再加上前面建立堆时N/2次向下调整，每次调整时间复杂度也为O(logN)。两次操作时间复杂度相加还是O(NlogN)，故堆排序的时间复杂度为O(NlogN)。

最坏情况：如果待排序数组是有序的，仍然需要O(NlogN)复杂度的比较操作，只是少了移动的操作；

最好情况：如果待排序数组是逆序的，不仅需要O(NlogN)复杂度的比较操作，而且需要O(NlogN)复杂度的交换操作，总的时间复杂度还是O(NlogN)。

因此，堆排序和快速排序在效率上是差不多的，但是堆排序一般优于快速排序的重要一点是数据的初始分布情况对堆排序的效率没有大的影响。

参考文献

[1] 浅谈堆和栈的区别
[2] 栈内存和堆内存的区别
[3] 浅谈内存分配方式以及堆和栈的区别（很清楚）
[4] C++函数调用过程深入分析
[5] 十种排序算法

杂注

我的博客即将搬运同步至腾讯云+社区，邀请大家一同入驻：
https://cloud.tencent.com/developer/support-plan?invite_code=2z2o0f9ecoo44