C/C++从入门到精通-高级程序员之路

【 数据结构 】

堆及其企业级应用

第 1 节 堆的故事导入

故事情节一

比武招亲，为古代的招亲方式之一，由女方设下擂台，邀请公众参与，候选人以武功最好者获得婚约。通常有两种形式：一是擂主由招亲的女生担任，谁挑战成功就成为新擂主。没人再比试的话直接获得婚约；要是还有人比试，武功最好者获得婚约。二是自由擂主，武功最高者成为擂主！

这种擂台式的结婚有如下优点和缺点：

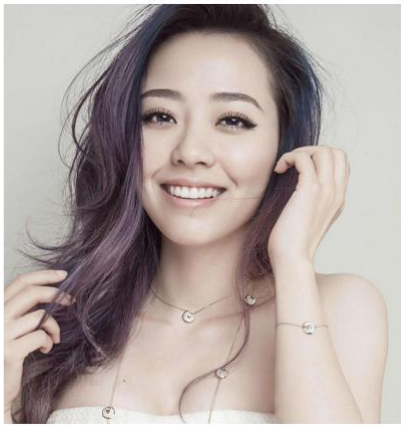
优点：可以找个猛男保镖保护，甚至一起闯荡江湖，四海为家
缺点：不怕找个东方不败，就怕找个像同治帝一样的得了“怪病”

问题来了？如果真的碰到怪病的？如何挑选第二任如意郎君？

故事情节二

李宇春	周笔畅	张靓颖



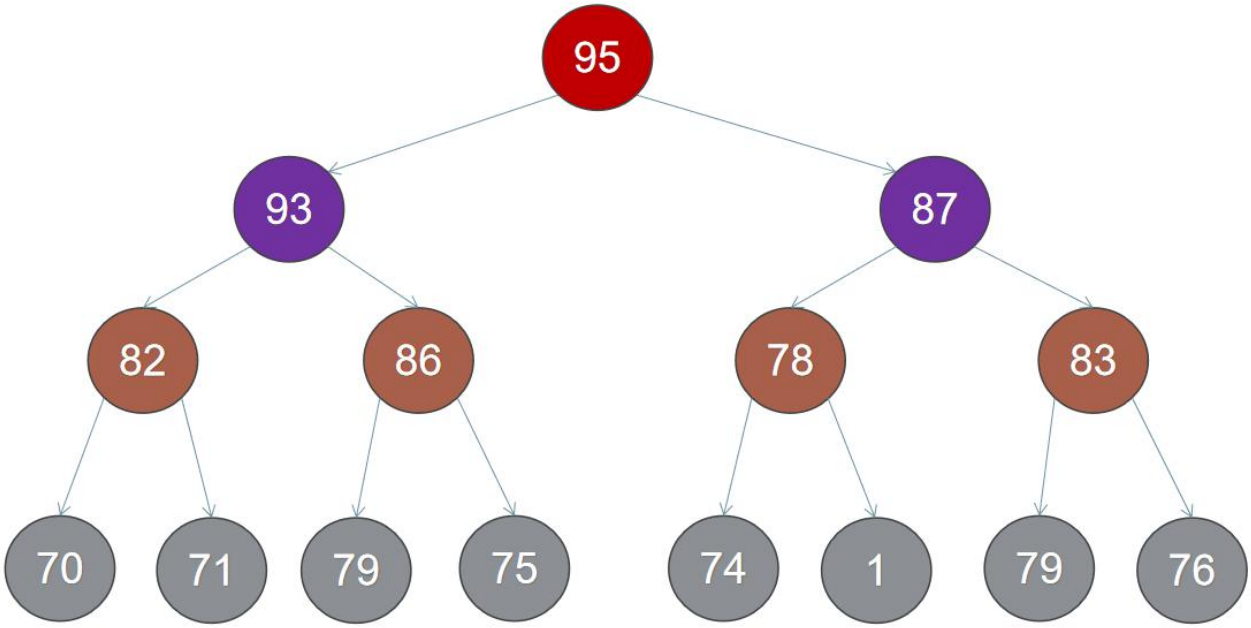


（2006 超级女声）

女神们的排行榜

95							
93				92			
82		86		87		83	
70	71	79	75	74	78	79	76
长沙	广州	天津	沈阳	南京	四川	云南	哈尔滨

第 2 节 堆的原理精讲



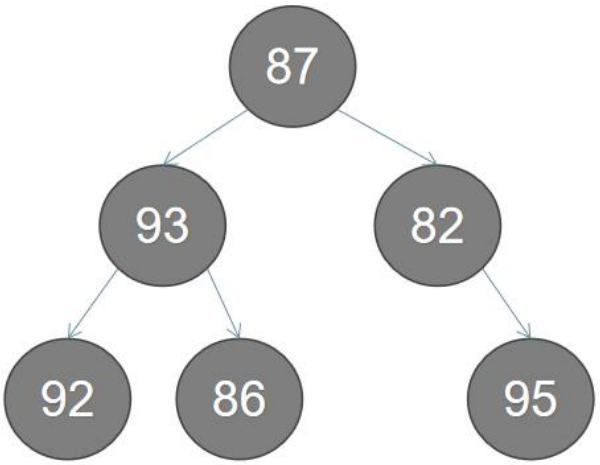
最大堆特点：

1.每个节点最多可以有两个节点

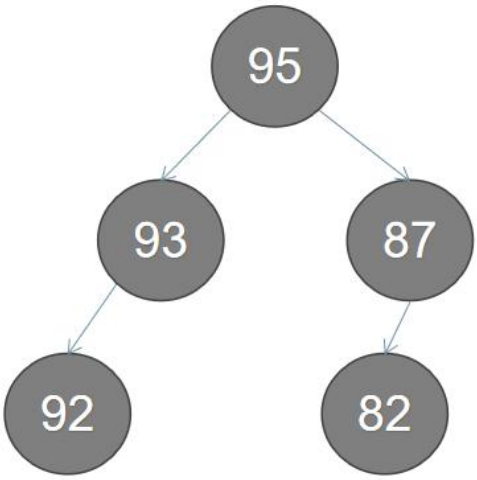
2.根结点的键值是所有堆结点键值中最大者，且每个结点的值都比其孩子的值大。

2.除了根节点没有兄弟节点，最后一个左子节点可以没有兄弟节点，其他节点必须有兄弟节点

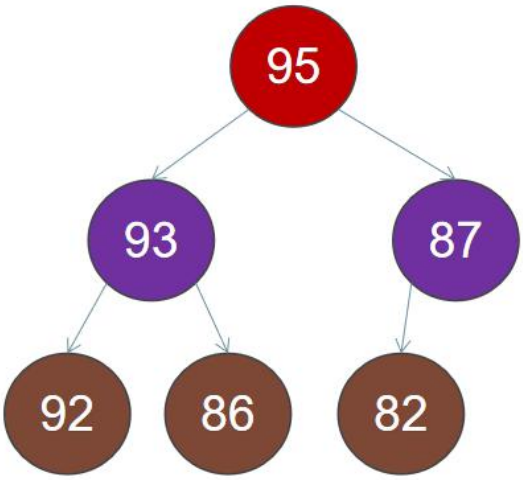
看图识最大堆： A B 不是堆，C 是最大堆



（A）



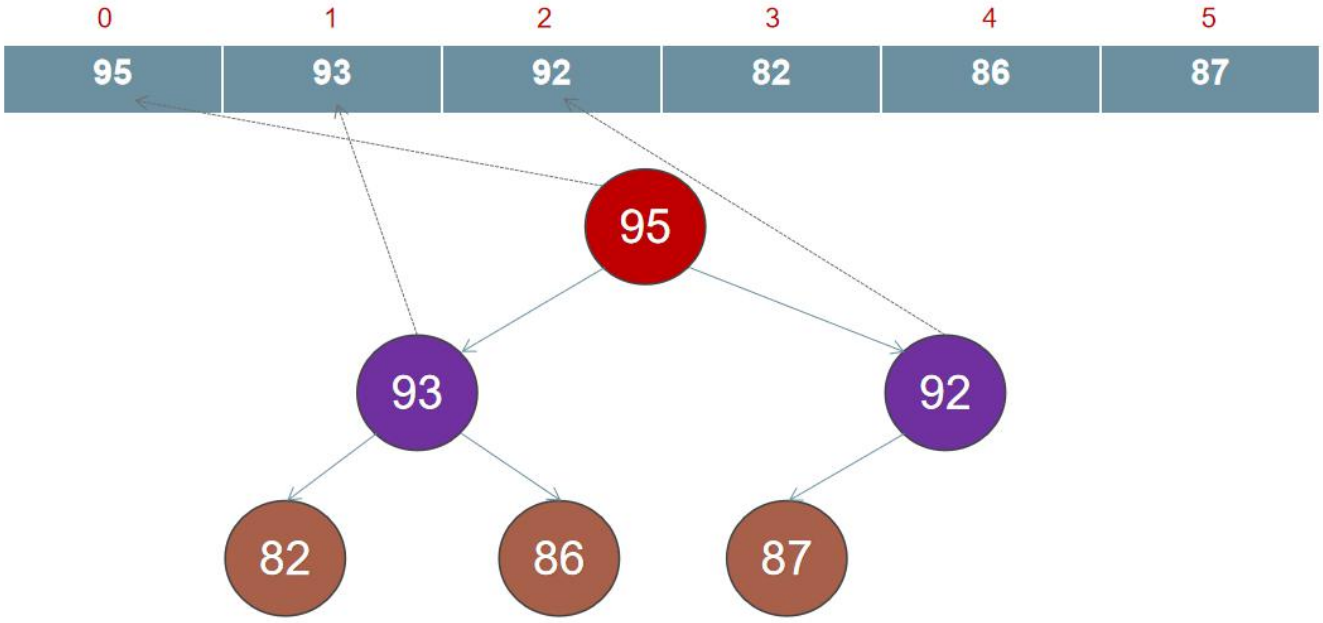
（B）



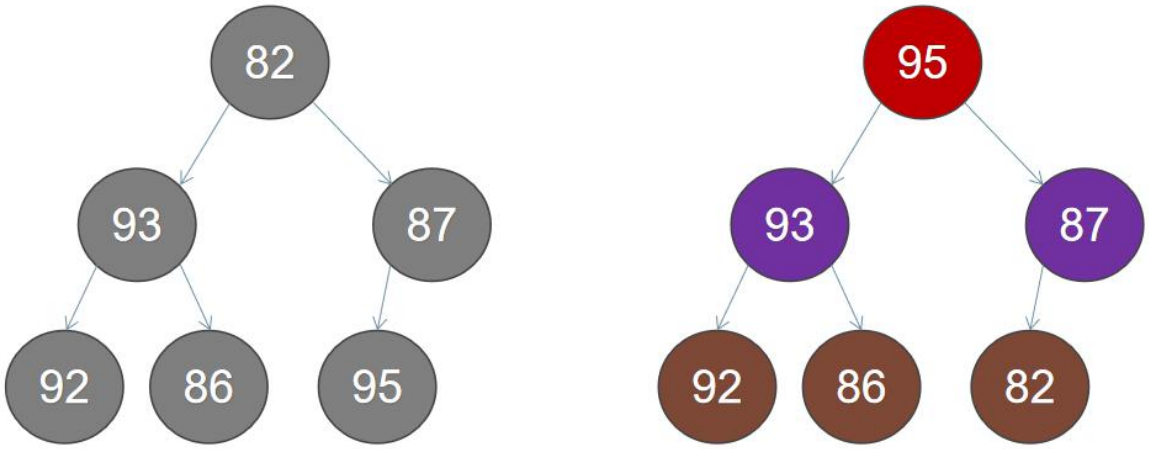
（C）

堆是你见过的最有个性的树！它是用数组表示的树

i的子节点左：2i+1 i的右子节点：2i+2
i的父节点：（i-1)/2

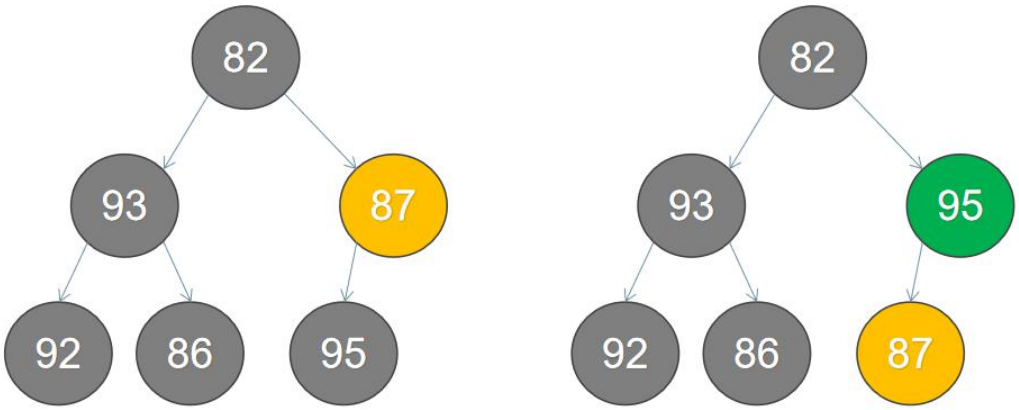


在数组中快速创建堆



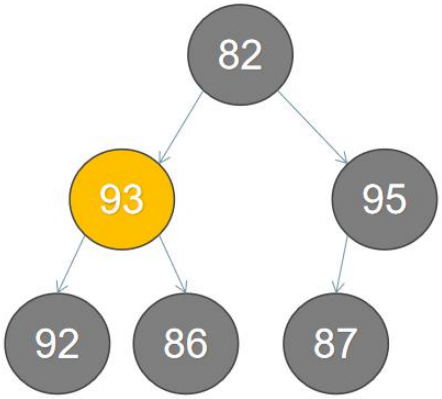
1. 首先我们需要找到最后一个结点的父结点如图(a),我们找到的结点是 87,然后找出该结点的最大子节点与自己比较,若该子节点比自身大,则将两个结点交换.

图(a)中,87 比左子节点 95 小,则交换之.如图(b)所示



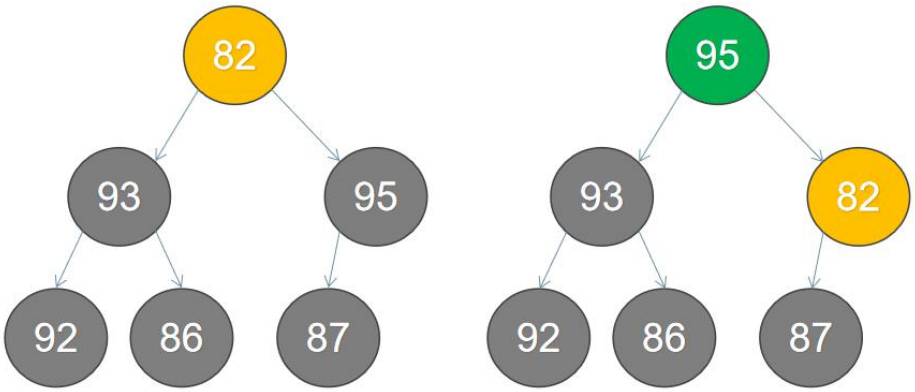
(a) (b)

2.我们移动到前一个父结点 93,如图(c)所示.同理做第一步的比较操作,结果不需要交换.

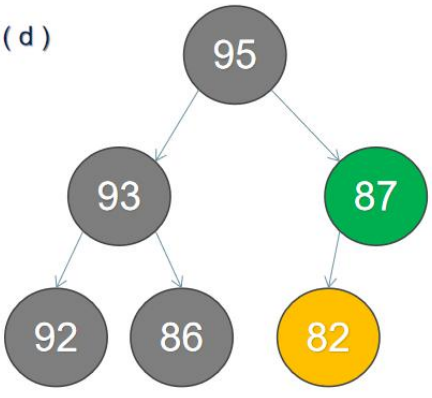


（c）

3.继续移动结点到前一个父结点 82,如图(d)所示,82 小于右子节点 95，则 82 与 95 交换,如图(e)所示，82 交换后，其值小于左子节点，不符合最大堆的特点，故需要继续向下调整，如图(f)所示



（d）　　（e）



（f）

4.所有节点交换完毕,最大堆构建完成

第 3 节　堆的算法实现

堆数据结构的定义

```
#define DEFAULT_CAPCITY  128

typedef struct _Heap{
    int  *arr;          //存储堆元素的数组
    int  size;          //当前已存储的元素个数
    int  capacity;      //当前存储的容量
}Heap;
```

建(最大)堆

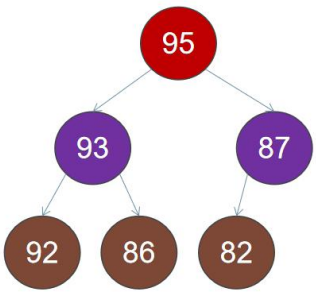
```
bool initHeap(Heap &heap, int *orginal, int size);
static void buildHeap(Heap &heap);
static void adjustDown(Heap &heap, int index);

/*初始化堆*/
bool initHeap(Heap &heap, int *orginal, int size){
    int capacity = DEFAULT_CAPCITY>size? DEFAULT_CAPCITY:size;

    heap.arr = new int[capacity];
    if(!heap.arr) return false;

    heap.capacity = capacity;
    heap.size = 0;

    //如果存在原始数据则构建堆
    if(size>0){
        memcpy(heap.arr, orginal, size*sizeof(int));
        heap.size = size;
        buildHeap(heap);
    }else {
        heap.size = 0;
    }
    return true;
}

/*将当前的节点和子节点调整成最大堆*/
void adjustDown(Heap &heap, int index)
{   int cur=heap.arr[index];//当前待调整的节点
    int parent,child;

    /*判断否存在大于当前节点子节点，如果不存在 ，则堆本身是平衡的，不需要调整；如果存在，则将最大的子节点与之交换，交换后，如果这个子节点还
```

```
有子节点，则要继续按照同样的步骤对这个子节点进行调整*/
    for(parent=index; (parent*2+1)<heap.size; parent=child){
        child=parent*2+1;

        //取两个子节点中的最大的节点

    if(((child+1)<heap.size)&&(heap.arr[child]<heap.arr[child+1])){
            child++;
        }

        //判断最大的节点是否大于当前的父节点
        if(cur>=heap.arr[child]){//不大于，则不需要调整，跳出循环
            break;
        }else {//大于当前的父节点，进行交换，然后从子节点位置继续向下调
整
            heap.arr[parent]=heap.arr[child];
            heap.arr[child]=cur;
        }
    }
}

/* 从最后一个父节点(size/2-1 的位置)逐个往前调整所有父节点（直到根节
点）,确保每一个父节点都是一个最大堆，最后整体上形成一个最大堆 */
void buildHeap(Heap &heap){
    int i;
    for(i=heap.size/2-1; i>=0; i--){
        adjustDown(heap, i);
    }
}
```

插入新元素

将数字 99 插入到上面大顶堆中的过程如下:

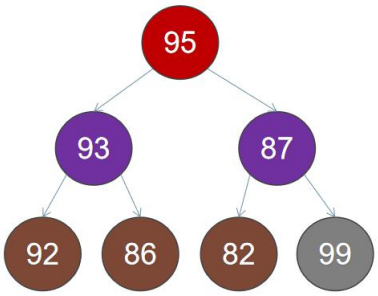
1） 原始的堆,如图 a



a. 原始大顶堆

对应的数组：{95, 93, 87, 92, 86, 82}

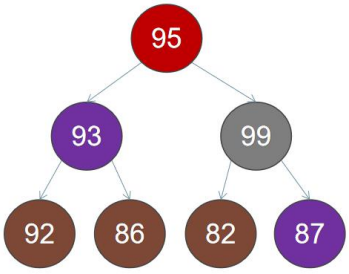
2） 将新进的元素插入到大顶堆的尾部,如下图 b 所示:



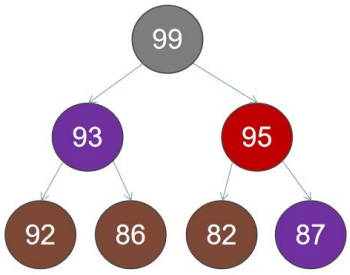
b. 加入新的元素

对应的数组：{95, 93, 87, 92, 86, 82, 99}

3） 此时最大堆已经被破坏，需要重新调整，因加入的节点比父节点大，则新节点跟父节点调换即可,如图 c 所示；调整后，新节点如果比新的父节点小，则已经调整到位，如果比新的父节点大，则需要和父节点重新进行交换，如图 d，至此，最大堆调整完成。



c. 新节点和父节点交换



d. 新节点和父节点交换

代码实现：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DEFAULT_CAPCITY  128

typedef struct _Heap {
    int  *arr;        //存储堆元素的数组
    int  size;        //当前已存储的元素个数
    int  capacity;    //当前存储的容量
}Heap;

bool initHeap(Heap &heap, int *orginal, int size);
bool insert(Heap &heap, int value);
static void buildHeap(Heap &heap);
static void adjustDown(Heap &heap, int index);
static void adjustUp(Heap &heap, int index);

/*初始化堆*/
bool initHeap(Heap &heap, int *orginal, int size) {
    int capacity = DEFAULT_CAPCITY>size ? DEFAULT_CAPCITY : size;

    heap.arr = new int[capacity];
    if (!heap.arr) return false;

    heap.capacity = capacity;
    heap.size = 0;

    //如果存在原始数据则构建堆
    if(size > 0){
        /*方式一: 直接调整所有元素
        memcpy(heap.arr, orginal, size*sizeof(int));
        heap.size = size;
        //建堆
        buildHeap(heap);
        */

        //方式二:  一次插入一个
        for(int i=0; i<size; i++){
            insert(heap, orginal[i]);
        }
    }
    return true;
}

/* 从最后一个父节点(size/2-1 的位置)逐个往前调整所有父节点（直到根节点），
```

```
确保每一个父节点都是一个最大堆，最后整体上形成一个最大堆 */
void buildHeap(Heap &heap) {
    int i;
    for (i = heap.size / 2 - 1; i >= 0; i--) {
        adjustDown(heap, i);
    }
}

/*将当前的节点和子节点调整成最大堆*/
void adjustDown(Heap &heap, int index)
{
    int cur = heap.arr[index];//当前待调整的节点
    int parent, child;

    /*判断否存在大于当前节点子节点，如果不存在 ，则堆本身是平衡的，不需要调整；
    如果存在，则将最大的子节点与之交换，交换后，如果这个子节点还有子节点，则要继续
    按照同样的步骤对这个子节点进行调整
    */
    for (parent = index; (parent * 2 + 1)<heap.size; parent = child) {
        child = parent * 2 + 1;

        //取两个子节点中的最大的节点
        if (((child + 1)<heap.size) && (heap.arr[child]<heap.arr[child + 1])) {
            child++;
        }

        //判断最大的节点是否大于当前的父节点
        if (cur >= heap.arr[child]) {//不大于，则不需要调整，跳出循环
            break;
        }
        else {//大于当前的父节点，进行交换，然后从子节点位置继续向下调整
            heap.arr[parent] = heap.arr[child];
            heap.arr[child] = cur;
        }
    }
}

/*将当前的节点和父节点调整成最大堆*/
void adjustUp(Heap &heap, int index) {
    if(index<0 ||  index >= heap.size){//大于堆的最大值直接 return
        return;
    }
```

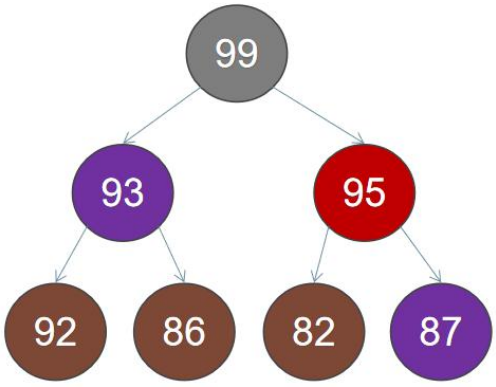
```
    while(index>0){
        int temp = heap.arr[index];
        int  parent = (index - 1) / 2;

        if(parent >= 0){//如果索引没有出界就执行想要的操作
            if(temp > heap.arr[parent]){
                heap.arr[index] = heap.arr[parent];
                heap.arr[parent] = temp;
                index = parent;
            }else {//如果已经比父亲小 直接结束循环
                break;
            }
        }else {//越界结束循环
            break;
        }
    }
}

/*最大堆尾部插入节点，同时保证最大堆的特性*/
bool insert(Heap &heap, int value) {
    if (heap.size == heap.capacity) {
        fprintf(stderr, "栈空间耗尽！\n");
        return false;
    }

    int index = heap.size;
    heap.arr[heap.size++] = value;
    adjustUp(heap, index);
    return true;
}




int main(void) {
    Heap  hp;
    int origVals[] = { 1, 2, 3, 87, 93, 82, 92, 86, 95 };
    int i = 0;

    if(!initHeap(hp, origVals, sizeof(origVals)/sizeof(origVals[0]))){
        fprintf(stderr, "初始化堆失败！\n");
        exit(-1);
    }

    for (i = 0; i<hp.size; i++) {
        printf("the %dth element:%d\n", i, hp.arr[i]);
    }
```

```
    insert(hp, 99);
    printf("在堆中插入新的元素 99，插入结果:\n");
    for (i = 0; i<hp.size; i++) {
        printf("the %dth element:%d\n", i, hp.arr[i]);
    }

    system("pause");
    return 0;
}
```

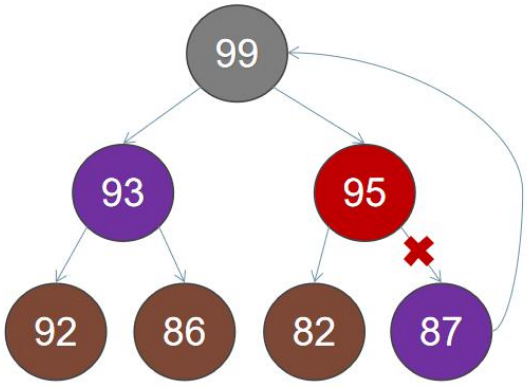
堆顶元素出列

如果我们将堆顶的元素删除，那么顶部有一个空的节点，怎么处理？

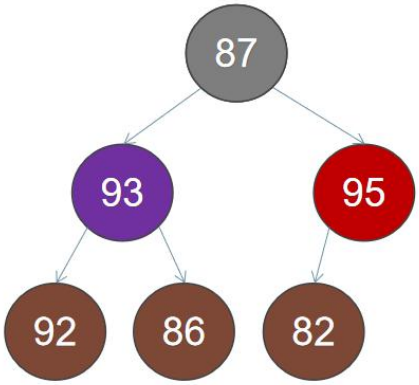
当插入节点的时候，我们将新的值插入数组的尾部。现在我们来做相反的事情：我们取出数组中的最后一个元素，将它放到堆的顶部，然后再修复堆属性。



a. 删除前



b. 删除时，用最后一个元素替换堆顶元素



c. 替换后

```
bool initHeap(Heap &heap, int *orginal, int size);
static void buildHeap(Heap &heap);
static void adjustDown(Heap &heap, int index);

/*初始化堆*/
bool initHeap(Heap &heap, int *orginal, int size){
    int capacity = DEFAULT_CAPCITY>size? DEFAULT_CAPCITY:size;

    heap.arr = new int[capacity];
    if(!heap.arr) return false;

    heap.capacity = capacity;
    heap.size = 0;

    //如果存在原始数据则构建堆
    if(size>0){
        memcpy(heap.arr, orginal, size*sizeof(int));
        heap.size = size;
        buildHeap(heap);
    }else {
        heap.size = 0;
    }
    return true;
}

/*将当前的节点和子节点调整成最大堆*/
void adjustDown(Heap &heap, int index)
{   int cur=heap.arr[index];//当前待调整的节点
    int parent,child;

    /*判断否存在大于当前节点子节点，如果不存在 ，则堆本身是平衡的，不需要调整；如果存在，则将最大的子节点与之交换，交换后，如果这个子节点还有子节点，则要继续按照同样的步骤对这个子节点进行调整*/
    for(parent=index; (parent*2+1)<heap.size; parent=child){
        child=parent*2+1;

        //取两个子节点中的最大的节点

    if(((child+1)<heap.size)&&(heap.arr[child]<heap.arr[child+1])){
            child++;
        }

        //判断最大的节点是否大于当前的父节点
        if(cur>=heap.arr[child]){//不大于，则不需要调整，跳出循环
            break;
        }else {//大于当前的父节点，进行交换，然后从子节点位置继续向下调整
```

```
            heap.arr[parent]=heap.arr[child];
            heap.arr[child]=cur;
        }
    }
}

/* 从最后一个父节点(size/2-1 的位置)逐个往前调整所有父节点（直到根节点）,确保每一个父节点都是一个最大堆，最后整体上形成一个最大堆 */
void buildHeap(Heap &heap){
    int i;
    for(i=heap.size/2-1; i>=0; i--){
        adjustDown(heap, i);
    }
}
```

第 4 节 堆的企业级应用案例

4.1 优先队列

操作系统内核作业调度是优先队列的一个应用实例，它根据优先级的高低而不是先到先服务的方式来进行调度；

如果最小键值元素拥有最高的优先级，那么这种优先队列叫作**升序优先队列**（即总是先删除最小的元素），类似的，如果最大键值元素拥有最高的优先级，那么这种优先队列叫作**降序优先队列**（即总是先删除最大的元素）；由于这两种类型是完全对称的，所以只需要关注其中一种，如升序优先队列.

优先队列算法实现：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DEFAULT_CAPCITY  128

typedef    int  DataType;
#define    isLess(a,b)   (a<b)

typedef struct _PriorityQueue {
    DataType  *arr;         //存储堆元素的数组
    int  size;         //当前已存储的元素个数
    int  capacity;     //当前存储的容量
}PriorityQueue;

bool init(PriorityQueue &pq, int *orginal, int size);
bool push(PriorityQueue &pq, DataType value);
bool pop(PriorityQueue &pq, DataType &value);
bool isEmpty(PriorityQueue &pq);
bool isFull(PriorityQueue &pq);
void destroy(PriorityQueue &pq);


static void build(PriorityQueue &pq);
static void adjustDown(PriorityQueue &pq, int index);
static void adjustUp(PriorityQueue &pq, int index);

/*初始化优先队列*/
bool init(PriorityQueue &pq, DataType *orginal, int size) {
    int capacity = DEFAULT_CAPCITY>size ? DEFAULT_CAPCITY : size;

    pq.arr = new DataType[capacity];
    if (!pq.arr) return false;

    pq.capacity = capacity;
    pq.size = 0;

    //如果存在原始数据则构建最大堆
    if(size > 0){
        //方式一: 直接调整所有元素
        memcpy(pq.arr, orginal, size*sizeof(int));
        pq.size = size;
        //建堆
        build(pq);
```

```
    }
    return true;
}

/*销毁优先级队列*/
void destroy(PriorityQueue &pq){
    if(pq.arr) delete[] pq.arr;
}


/*优先队列是否为空*/
bool isEmpty(PriorityQueue &pq){
    if(pq.size<1) return true;

    return false;
}

/*优先队列是否为满*/
bool isFull(PriorityQueue &pq){
    if(pq.size<pq.capacity) return false;

    return true;
}

int size(PriorityQueue &pq){
    return pq.size;
}

/* 从最后一个父节点(size/2-1 的位置)逐个往前调整所有父节点（直到根节点），
确保每一个父节点都是一个最大堆，最后整体上形成一个最大堆 */
void build(PriorityQueue &pq) {
    int i;
    for (i = pq.size / 2 - 1; i >= 0; i--) {
        adjustDown(pq, i);
    }
}

/*将当前的节点和子节点调整成最大堆*/
void adjustDown(PriorityQueue &pq, int index)
{
    DataType cur = pq.arr[index];//当前待调整的节点
    int parent, child;

    /*判断否存在大于当前节点子节点，如果不存在 ，则堆本身是平衡的，不需要调整；
    如果存在，则将最大的子节点与之交换，交换后，如果这个子节点还有子节
```

```
点，则要继续
    按照同样的步骤对这个子节点进行调整
    */
    for (parent = index; (parent * 2 + 1)<pq.size; parent = child) {
        child = parent * 2 + 1;

        //取两个子节点中的最大的节点
        if (((child + 1)<pq.size) && isLess(pq.arr[child], pq.arr[child
+ 1])) {
            child++;
        }

        //判断最大的节点是否大于当前的父节点
        if (isLess(pq.arr[child], cur)) {//不大于，则不需要调整，跳出循
环
            break;
        }
        else {//大于当前的父节点，进行交换，然后从子节点位置继续向下调
整
            pq.arr[parent] = pq.arr[child];
            pq.arr[child] = cur;
        }
    }
}

/*将当前的节点和父节点调整成最大堆*/
void adjustUp(PriorityQueue &pq, int index) {
    if(index<0 ||  index >= pq.size){//大于堆的最大值直接 return
        return;
    }

    while(index>0){
        DataType temp = pq.arr[index];
        int  parent = (index - 1) / 2;

        if(parent >= 0){//如果索引没有出界就执行想要的操作
            if( isLess(pq.arr[parent], temp) ){
                pq.arr[index] = pq.arr[parent];
                pq.arr[parent] = temp;
                index = parent;
            }else {//如果已经比父亲小 直接结束循环
                break;
            }
        }else {//越界结束循环
            break;
        }
    }
```

```
}

/* 删除优先队列中最大的节点，并获得节点的值*/
bool pop(PriorityQueue &pq, DataType &value) {

    if (isEmpty(pq)) return false;

    value = pq.arr[0];
    pq.arr[0] = pq.arr[--pq.size];
    //heap.arr[0] = heap.arr[heap.size-1];
    //heap.size--;
    adjustDown(pq, 0);// 向下执行堆调整
    return true;
}


/*优先队列中插入节点*/
bool push(PriorityQueue &pq, DataType value) {
    if (isFull(pq)) {
        fprintf(stderr, "优先队列空间耗尽！\n");
        return false;
    }

    int index = pq.size;
    pq.arr[pq.size++] = value;
    adjustUp(pq, index);
    return true;
}


int main(void) {
    PriorityQueue  pq;
    int task[] = { 1, 2, 3, 87, 93, 82, 92, 86, 95 };
    int i = 0;

    if(!init(pq, task, sizeof(task)/sizeof(task[0]))){
        fprintf(stderr, "初始化优先队列失败！\n");
        exit(-1);
    }

    for (i = 0; i<pq.size; i++) {
        printf("the %dth task:%d\n", i, pq.arr[i]);
    }

    //堆中插入优先级为 88 的任务
    push(pq, 88);
```

```
    //堆中元素出列
    printf("按照优先级出列：\n");
    DataType value;
    while( pop(pq, value)){
        printf(" %d\n", value);
    }

    destroy(pq);
    system("pause");
    return 0;
}
```

课后思考：

上面优先队列算法实现中的 DateType 如果换成以下类型，上面的代码该如何调整？

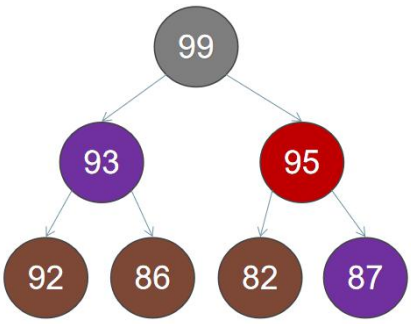
```
typedef struct _Task{
    int priority; //降序优先队列
    //其它的状态属性省略
}Task;
```

4.2 堆排序

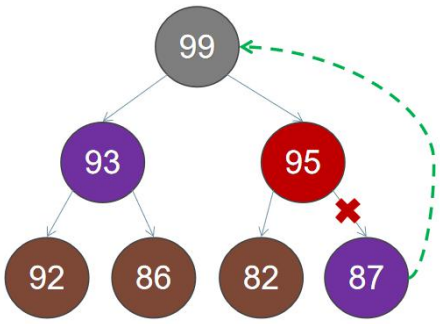
堆排序(Heapsort)是指利用堆这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素.

(选择排序工作原理 - 第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零)

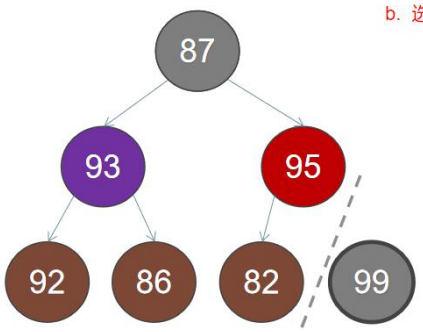
其排序核心实现如下：



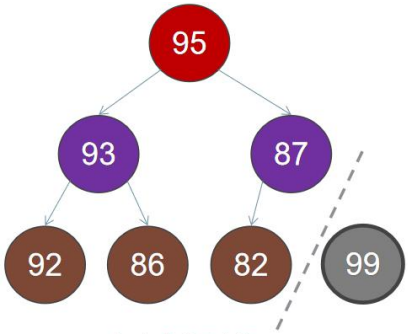
a. 选择最大元素前



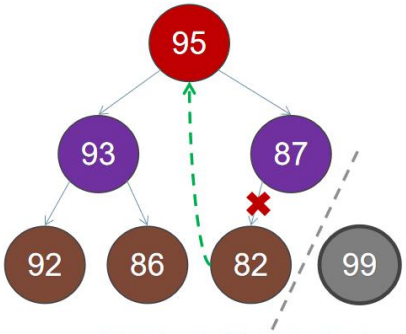
b. 选择最大元素，并用最后一个元素替换



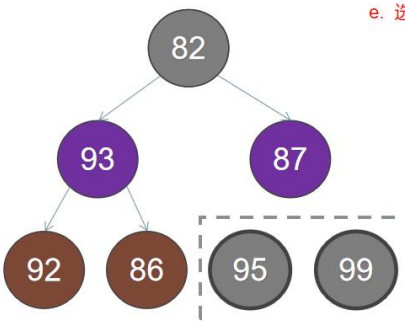
c. 最大元素移动到堆的最后一个元素



d. 向下调整为最大堆



e. 选择最大元素，并用最后一个元素替换



f. 最大元素移动到堆的最后一个元素

具体算法实现:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _Heap {
    int  *arr;        //存储堆元素的数组
    int  size;        //当前已存储的元素个数
    int  capacity;    //当前存储的容量
}Heap;

bool initHeap(Heap &heap, int *orginal, int size);
bool popMax(Heap &heap, int &value);
void heapSort(Heap &heap);

static void buildHeap(Heap &heap);
static void adjustDown(Heap &heap, int index);


/*初始化堆*/
bool initHeap(Heap &heap, int *orginal, int size) {
    //heap.arr = new int[capacity];
    heap.arr = orginal;
    if (!heap.arr) return false;

    heap.capacity = size;
    heap.size = size;

    //如果存在原始数据则构建堆
    if(size > 0){
        //方式一: 直接调整所有元素
        //建堆
        buildHeap(heap);
    }
    return true;
}

/* 从最后一个父节点(size/2-1 的位置)逐个往前调整所有父节点（直到根节点），
确保每一个父节点都是一个最大堆，最后整体上形成一个最大堆 */
void buildHeap(Heap &heap) {
    int i;
    for (i = heap.size / 2 - 1; i >= 0; i--) {
        adjustDown(heap, i);
    }
}
```

```
/*将当前的节点和子节点调整成最大堆*/
void adjustDown(Heap &heap, int index)
{
    int cur = heap.arr[index];//当前待调整的节点
    int parent, child;

    /*判断否存在大于当前节点子节点，如果不存在 ，则堆本身是平衡的，不需要调整；
    如果存在，则将最大的子节点与之交换，交换后，如果这个子节点还有子节点，则要继续
    按照同样的步骤对这个子节点进行调整
    */
    for (parent = index; (parent * 2 + 1)<heap.size; parent = child) {
        child = parent * 2 + 1;

        //取两个子节点中的最大的节点
        if (((child + 1)<heap.size) && (heap.arr[child]<heap.arr[child + 1])) {
            child++;
        }

        //判断最大的节点是否大于当前的父节点
        if (cur >= heap.arr[child]) {//不大于，则不需要调整，跳出循环
            break;
        }
        else {//大于当前的父节点，进行交换，然后从子节点位置继续向下调整
            heap.arr[parent] = heap.arr[child];
            heap.arr[child] = cur;
        }
    }
}


/*  实现堆排序 */
void heapSort(Heap &heap){

    if (heap.size<1) return ;

    while(heap.size>0){
        int tmp = heap.arr[0];
        heap.arr[0] = heap.arr[heap.size-1];
        heap.arr[heap.size-1] = tmp;
        heap.size--;
        adjustDown(heap, 0);// 向下执行堆调整
    }
```

```
}

/* 删除最大的节点，并获得节点的值*/
bool popMax(Heap &heap, int &value) {

    if (heap.size<1) return false;

    value = heap.arr[0];
    heap.arr[0] = heap.arr[--heap.size];
    //heap.arr[0] = heap.arr[heap.size-1];
    //heap.size--;
    adjustDown(heap, 0);// 向下执行堆调整
    return true;
}




int main(void) {
    Heap  hp;
    int origVals[] = { 1, 2, 3, 87, 93, 82, 92, 86, 95 };
    int i = 0;

    if(!initHeap(hp, origVals, sizeof(origVals)/sizeof(origVals[0]))){
        fprintf(stderr, "初始化堆失败！\n");
        exit(-1);
    }

    for (i = 0; i<hp.size; i++) {
        printf("the %dth element:%d\n", i, hp.arr[i]);
    }

    //执行堆排序
    heapSort(hp);

    printf("堆排序后的结果：\n");
    for(i=0; i<sizeof(origVals)/sizeof(origVals[0]); i++){
        printf("  %d", origVals[i]);
    }

    system("pause");
    return 0;
}
```

4.3 快速查找无序集合中前 N 大（小）的记录

- 请同学们自行完成并交给我检查