

下流算法, 上流算法?

Chapter 6

Priority Queues

1. 堆
2. 堆排序算法
上流, 下流

6.1 Introduction



- A priority queue is a collection of zero or more elements. Each element has a priority or value.

6.1 Introduction

- In a min priority queue the find operation finds the element with minimum priority, while the delete operation delete this element.
- In a max priority queue, the find operation finds the element with maximum priority, while the delete operation delete this element.

6.1 Introduction

ADT of a max priority queue

AbstractDataType MaxPriorityQueue

{

instances

finite collection of elements, each has a priority

operations

Create(): create an empty priority queue

Size(): return number of element in the queue

Max(): return element with maximum priority

Insert(x): insert x into queue

DeleteMax(x): delete the element with largest priority

from the queue; return it in x;

}

6.2 Linear List Representation



Use an unordered linear list

Insertions are performed at the right end of the list, $\theta(1)$

A deletion requires a search for the element with largest priority, $\theta(n)$

6.3 Heaps

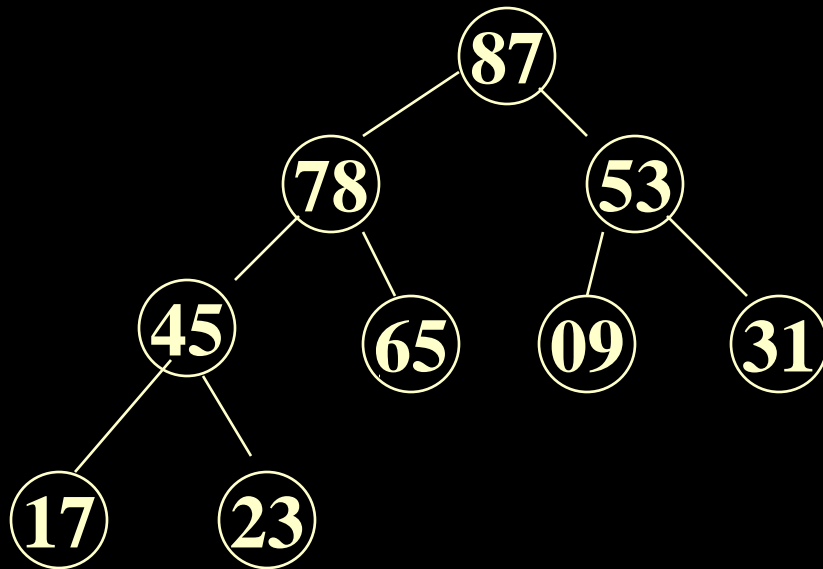
1.definition: A max heap(min Heap)

- **is A complete binary tree**
- **The value in each node is greater(less) than or equal to those in its children(if any).**

6.3 Heaps

Example of a max heap

$k = \{87, 78, 53, 45, 65, 09, 31, 17, 23\}$

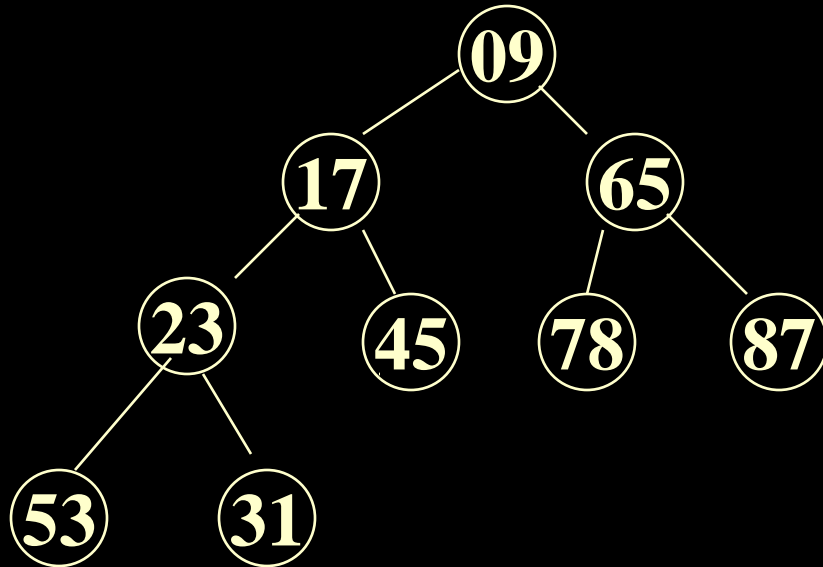


手 = 又手、
有点像

6.3 Heaps

Example of a min heap

$k=\{09,17,65,23,45,78,87,53,31\}$




6.3 Heaps

2. class MaxHeap

Data member of heap: T * heap, int MaxSize, CurrentSize

```
template<class T>class MaxHeap
{ public:
    MaxHeap(int MaxHeapSize=10);
    ~MaxHeap(){delete[] heap;}
    int size()const{return CurrentSize;}
    T Max(){ if (CurrentSize==0)throw OutOfBounds();
             return heap[1];}
    MaxHeap<T>&insert(const T&x);
    MaxHeap<T>& DeleteMax(T& x);
    void initialize(T a[], int size, int ArraySize);
private:
    int CurrentSize, MaxSize;
    T * heap;
}
```



6.3 Heaps

3.member function of MaxHeap

1)Constructor for MaxHeap

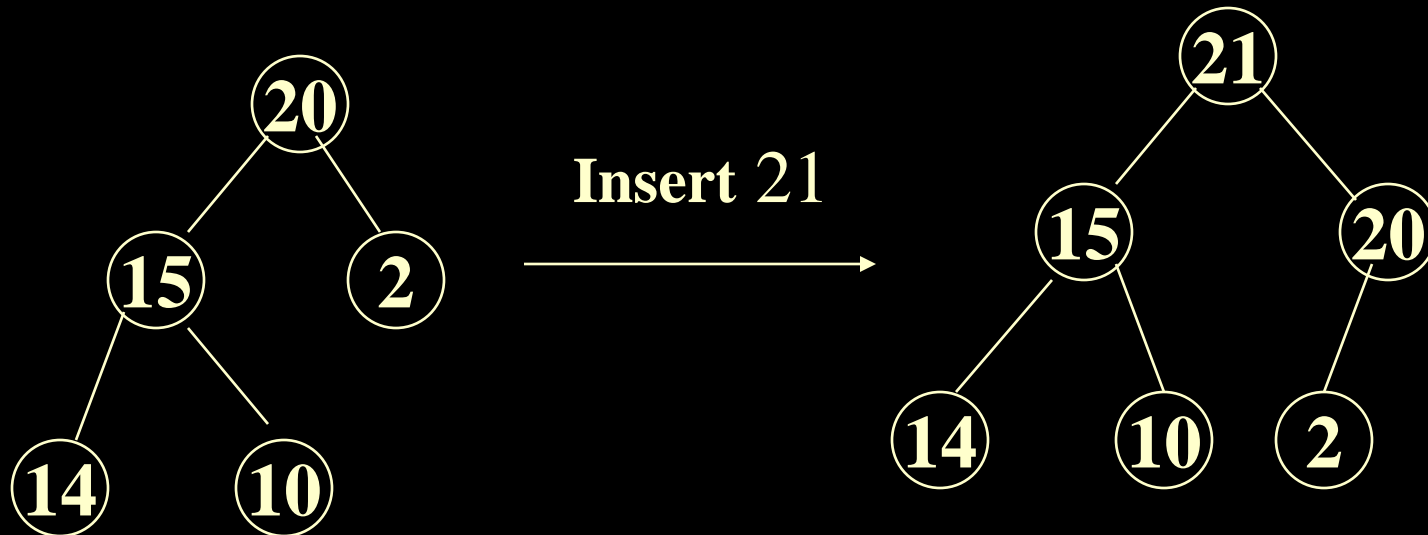
```
template<class T>
MaxHeap<T>::MaxHeap(int MaxHeapSize)
{
    MaxSize=MaxHeapSize;
    Heap=new T[MaxSize+1];
    CurrentSize=0;
}
```

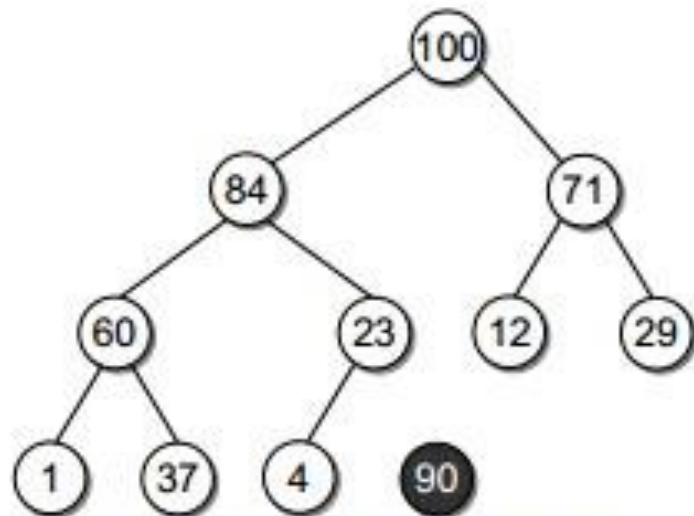


6.3 Heaps

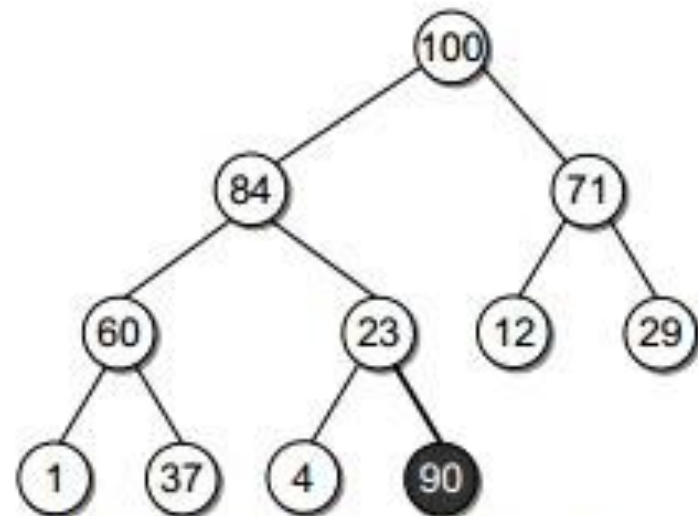
2) Insertion

Example:

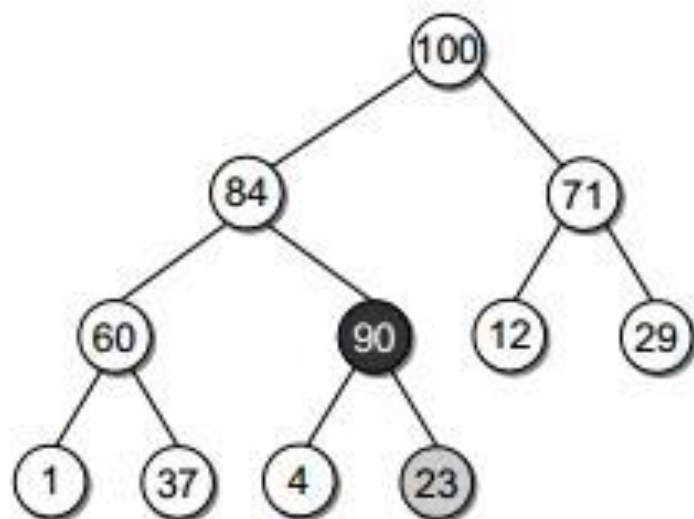




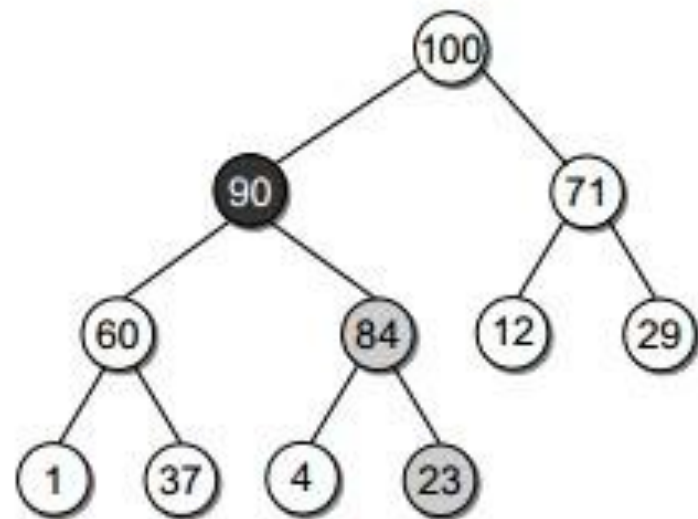
(a) create a new node for 90.



(b) link the node as the last child.



(c) sift-up: swap 23 and 90.



(d) sift-up: swap 84 and 90.

6.3 Heaps

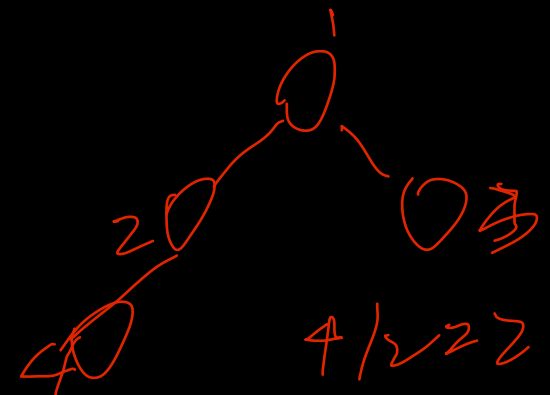
Insertion

```
template<class T>MaxHeap<T>& MaxHeap<T>::  
    Insert(const T& x)  
{ if(CurrentSize==MaxSize)throw NoMem();  
  int i=++CurrentSize;  
  while(i!=1&& x>heap[i/2])  
    { heap[i]=heap[i/2]; i/=2; }  
  heap[i]=x;  
  return *this;  
}
```

time complexity is $O(\log_2 n)$

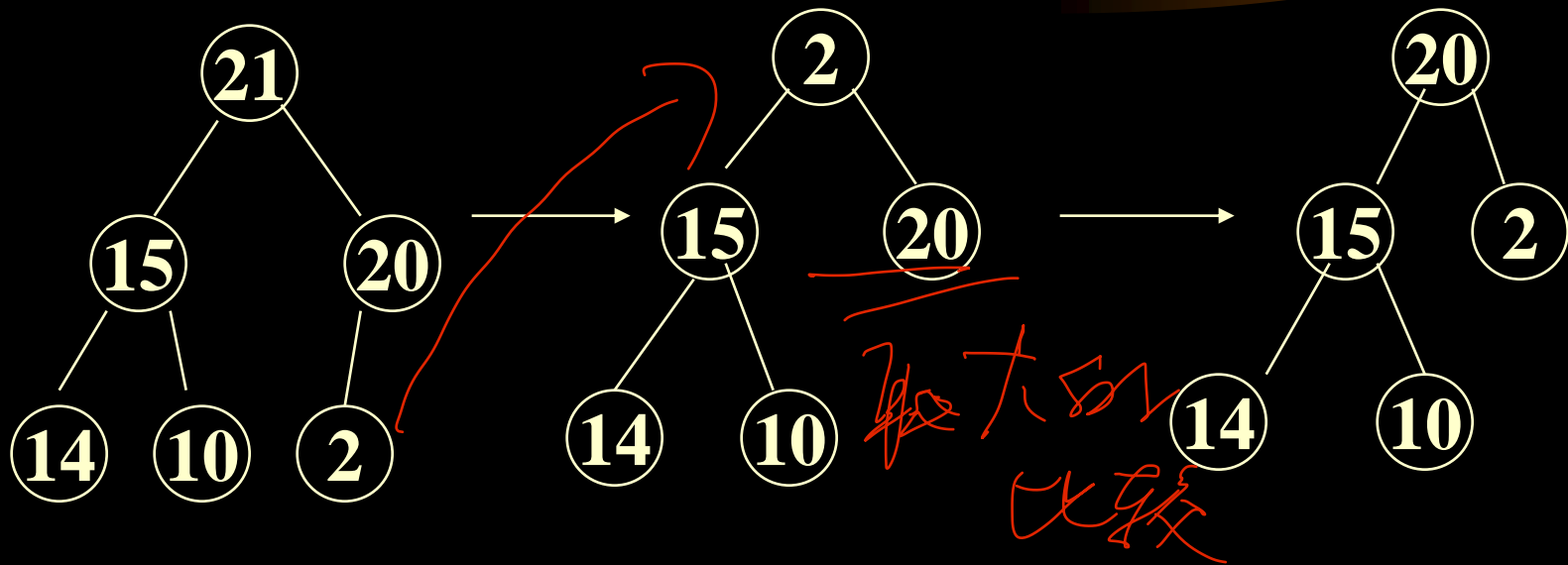
父节点下移

这里下移从开始



6.3 Heaps

3) deletion



下堆调整

6.3 Heaps

deletion from a max heap

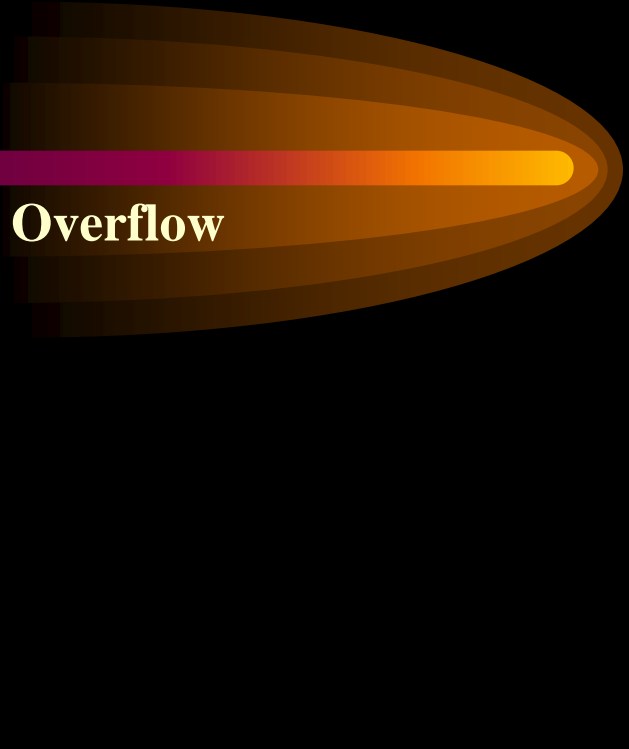
```
template<class T>MaxHeap<T>& MaxHeap<T>:: DeleteMax(T& x)
{ if (CurrentSize==0)throw OutOfBounds();
  x=heap[1];
  T y=heap[CurrentSize--];
  int i=1; ci=2;
  while(ci<=CurrentSize)
  { if(ci<CurrentSize&&heap[ci]<heap[ci+1]) ci++;
    if(y>=heap[ci]) break;
    heap[i]=heap[ci];
    i=ci; ci*=2;
  }
  heap[i]=y;  return *this;
}    Time complexity is  $O(\log_2 n)$ 
```

6.3 Heaps

```
java program(MinHeap)
public class BinaryHeap
{
    public BinaryHeap( )
    public BinaryHeap( int capacity )
    public void insert( Comparable x ) throws Overflow
    public Comparable findMin( )
    public Comparable deleteMin( )

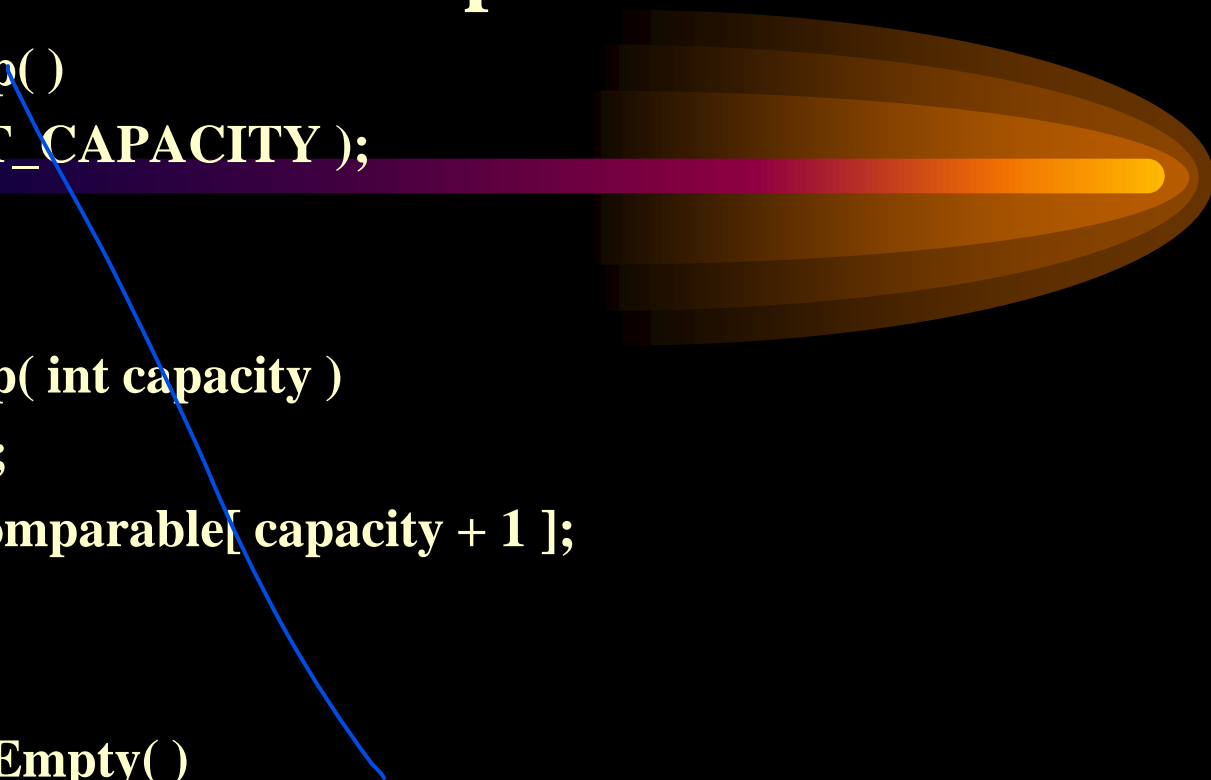
    public boolean isEmpty( )
    public boolean isFull( )
    public void makeEmpty( )

    private static final int DEFAULT_CAPACITY = 100;
    private int currentSize;
    private Comparable [ ] array;
    private void percolateDown( int hole )
    private void buildHeap( )
}
```



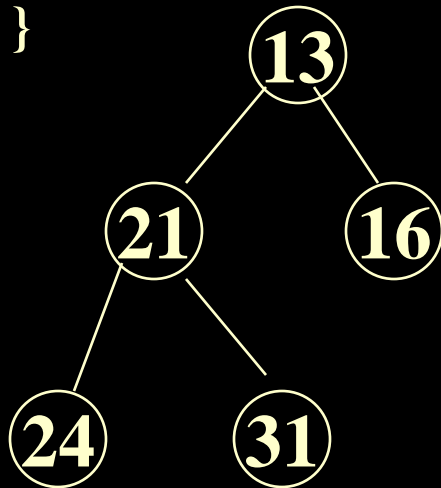
6.3 Heaps

```
public BinaryHeap( )  
{  this( DEFAULT_CAPACITY );  
}  
  
public BinaryHeap( int capacity )  
{  currentSize = 0;  
    array = new Comparable[ capacity + 1 ];  
}  
  
public void makeEmpty()  
{  currentSize = 0;  
}
```

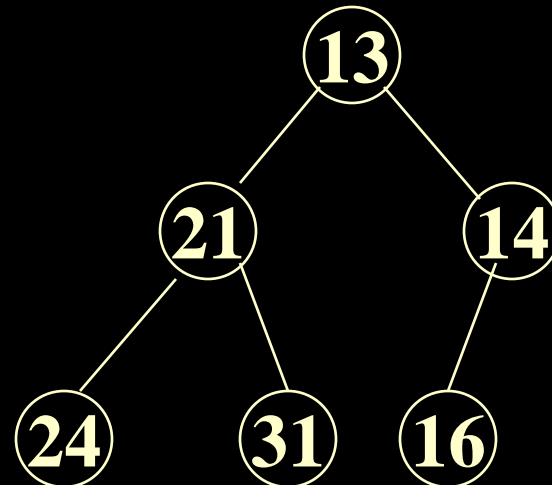


6.3 Heaps

```
public void insert( Comparable x ) throws Overflow
{ if( isFull( ) )
    throw new Overflow( );
  int hole = ++currentSize;
  for( ; hole > 1 && x.comparebleTo( array[ hole / 2 ] ) < 0;
      hole /= 2 )
    array[ hole ] = array[ hole / 2 ];
  array[ hole ] = x;
}
```



Insert 14



6.3 Heaps

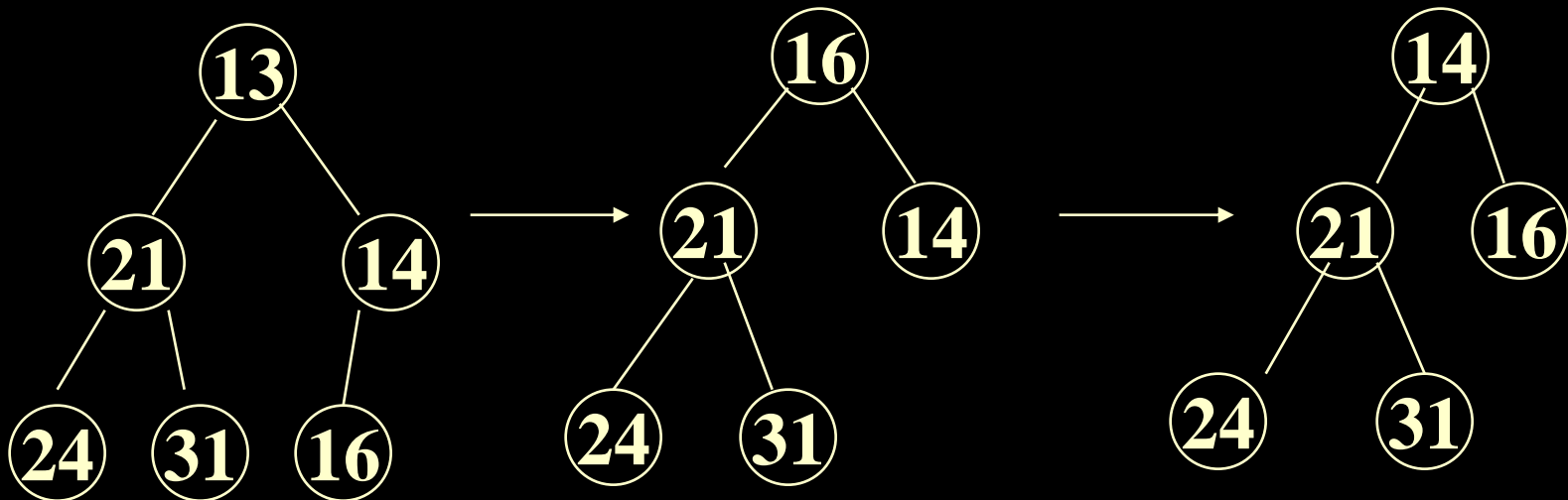
```
public Comparable deleteMin()  
{ if( isEmpty( ) )  
    return null;
```

```
    Comparable minItem = findMin( );  
    array[ 1 ] = array[ currentSize-- ];  
    percolateDown( 1 );
```

```
    return minItem;  
}
```

6.3 Heaps

```
private void percolateDown( int hole )
{   int child;
    Comparable tmp = array[ hole ];
    for( ; hole *2 <= currentSize; hole = child )
    {   child = hole * 2;
        if ( child != currentSize && array[ child + 1 ].compareTo( array[ child ] ) < 0 )
            child++;
        if( array[child ].compareTo( tmp ) < 0 )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```



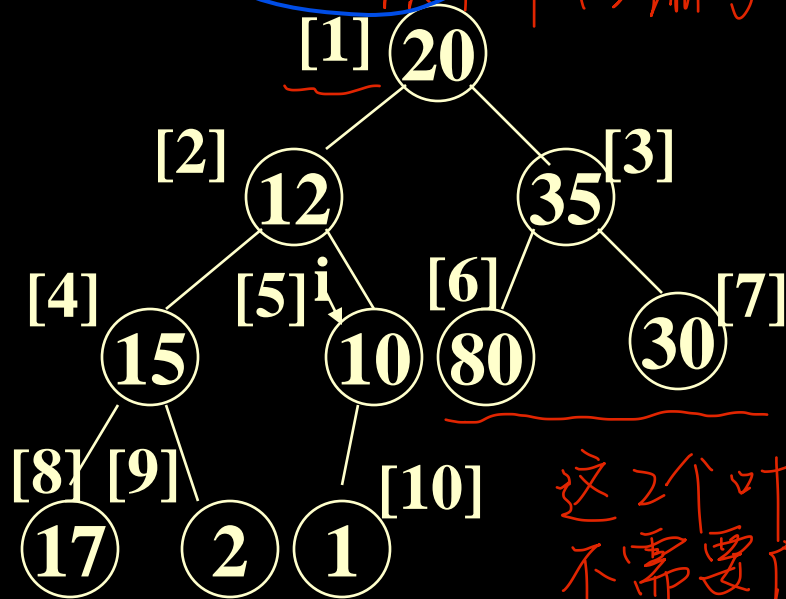
6.3 Heaps

4) Initialize a nonempty max heap

Example: {20,12,35,15,10,80,30,17,2,1}

书中称为由底向上:

从1开始编号



最后几个元素的父结点, 然后依次
 $i = [n/2], [n/2]-1, \dots, 1$ 做下滤

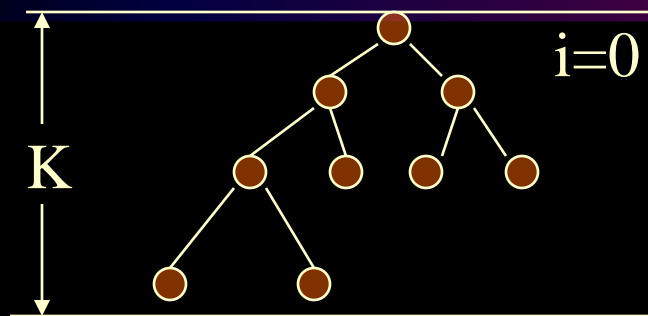
$O(n)$

Turn into max heap from
these subtree roots

这两个叶结点
不需要做下滤

算法分析

初始建堆: n 个结点, $K = \lfloor \log_2 n \rfloor$, 从0层开始



树高

第 i 层交换的最大次数为 $k-i$
第 i 层有 2^i 个结点

即第 i 层的数
最多 $(k-i)$ 次
下滤

$$\text{总交换次数: } \sum_{i=0}^{k-1} 2^i \cdot (k-i) = \sum_{j=1}^k j \cdot 2^{k-j} = \sum_{j=1}^k j(2^k \cdot 2^{-j})$$

令 $k-i=j$

$$= 2^k \cdot \sum_{j=1}^k j \cdot 2^{-j} \leq 2^k \cdot 2 \leq 2^{\log n} \cdot 2 = 2n = O(n)$$

走1个树高

最后一层要上滤

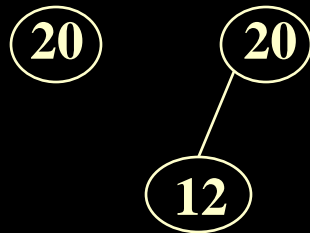
为什么降低这么多, 最后一层下滤时都可以不走了, 而原来

6.3 Heaps

4) Initialize a nonempty max heap

Example: {20,12,35,15,10,80,30,17,2,1}

还可以这样做: 依次插入一个元素到堆中. 书中称为由顶向下(也可见书中例子).



每次加入一个数据
然后做上堆 ($\log n$)
复杂度: $n \log n$

Complexity of Initialize:

$$\sum_{K=1}^n \lfloor \lg k \rfloor \leq \sum_{K=1}^n \lg k = \lg 1 + \dots + \lg n = \lg(1 \cdot 2 \cdot \dots \cdot n) =$$

$$\lg(n!) = O(n \lg n)$$

(提示: 因 $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$

显然该 n 项都小于 n , 所以 $\lg(n!) \leq \lg(n \cdot n \cdot n \cdot \dots \cdot n)$)

6.3 Heaps

(从最后1个结点的父结点开始, 往前)

initialize (C++ program)

```
Template<class T> void MaxHeap<T>::
```

```
    Initialize (T a[],int size,int ArraySize)
```

```
{ delete[] heap;
```

对于每个结点做下滤

```
    heap=a; CurrentSize=Size; MaxSize=ArraySize;
```

```
    for( int i=CurrentSize/2; i>=1; i--)
```

```
    { T y=heap[i]; int c=2*i;
```

```
        while(c<=CurrentSize)
```

```
        { if(c<CurrentSize && heap[c]<heap[c+1]) c++;
```

```
            if(y>=heap[c]) break;
```

```
            heap[c/2] = heap[c];
```

```
            c*=2;
```

```
        }
```

```
        heap[c/2]=y;
```

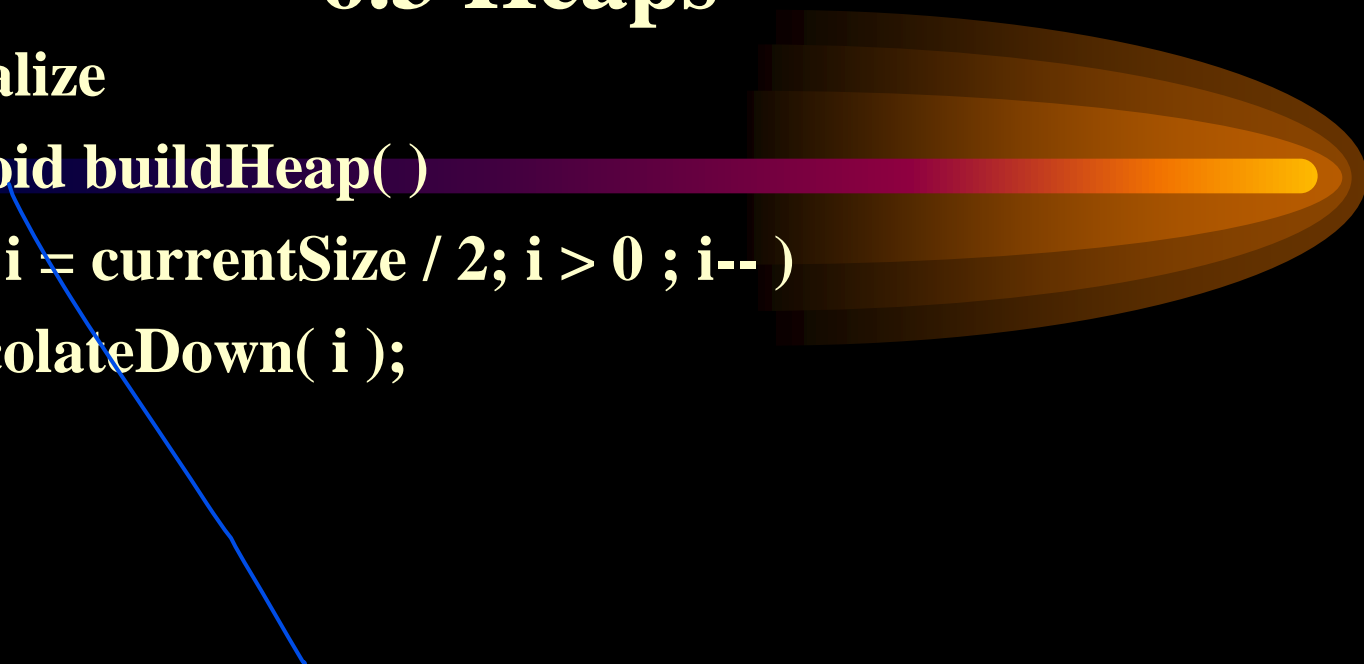
```
    }
```

```
}
```


6.3 Heaps

java initialize

```
private void buildHeap()  
{ for( int i = currentSize / 2; i > 0 ; i-- )  
    percolateDown( i );  
}
```



6.4.Applications of Priority Queues

1.heap sort

Method:

1)initialize a max heap with the n elements to be sorted $O(n)$

2)each time we delete one element, then adjust the heap $O(\log_2 n)$

出堆过程

下滤复杂度

n 个结点则 $n \log n$

Time complexity is $O(n) + O(n * \log_2 n) = O(n * \log_2 n)$

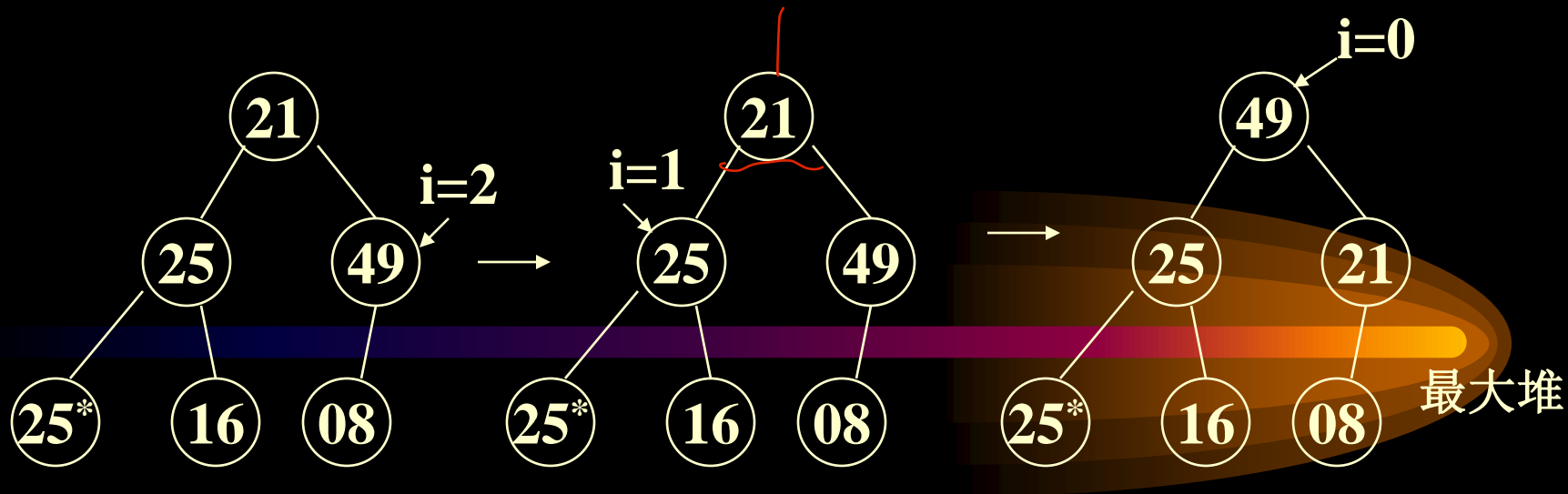
复杂度低的加高的 = 高的.

6.4.Applications of Priority Queues

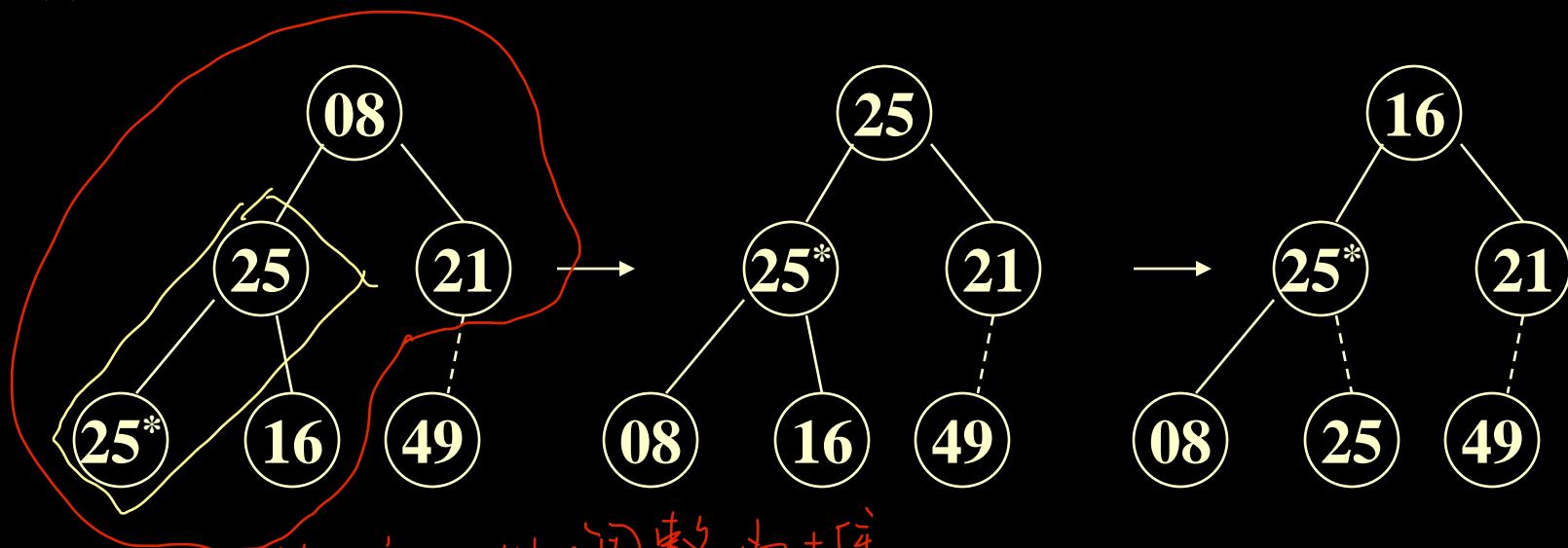
heap sort

Example : {21,25,49,25*,16,08}

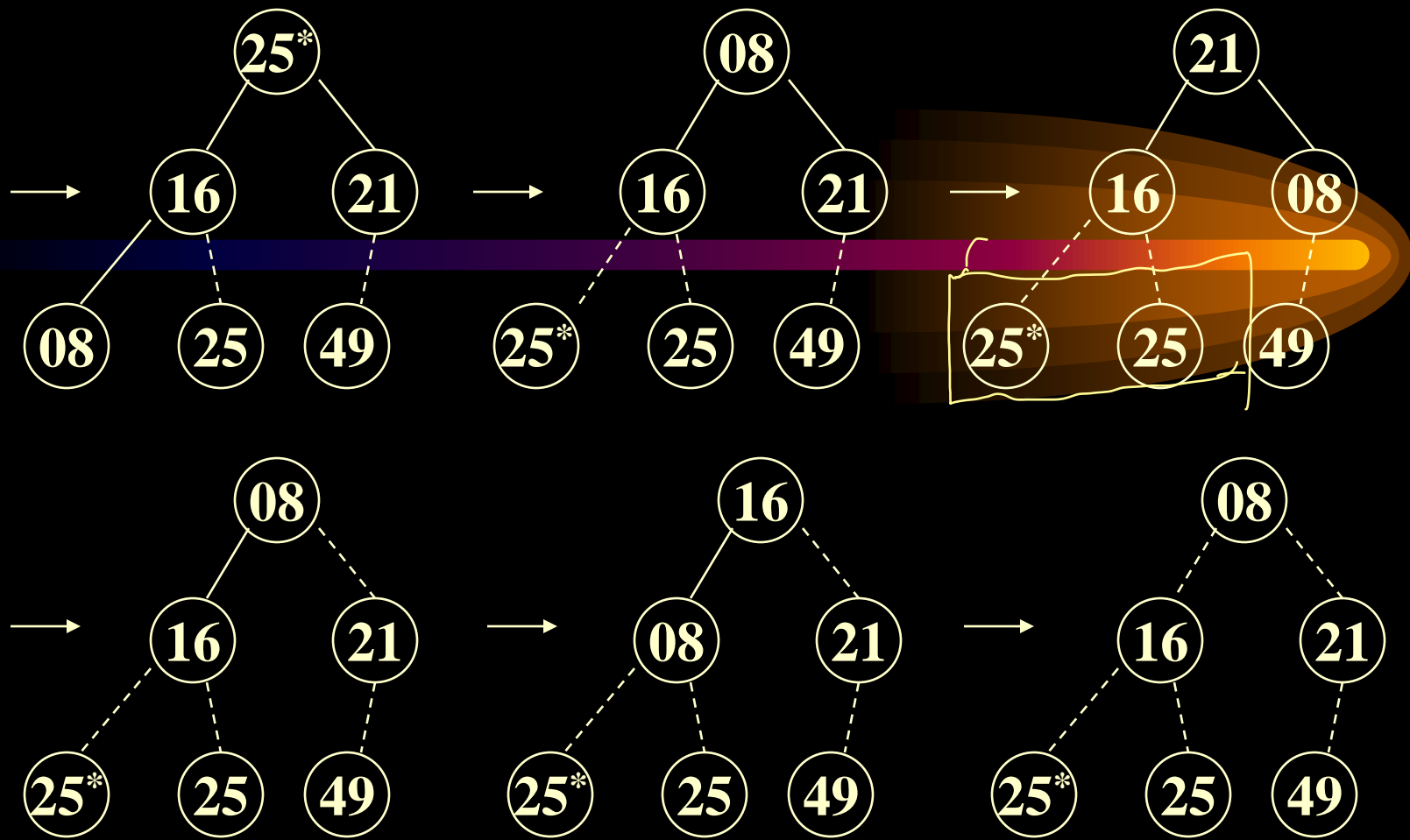
21 下流
25 < 49
21 < 49
21 和 49 交换



调整



将这一块调整为堆
 \Rightarrow 根结点成为最大数结点，出队，以此类推。



原来 25 在 25* 之前, 排完后 25* 在 25 后面.

从以上例子可以看出堆排序是不稳定的 顺序改变

没有用额外数组存放出堆的数,
节省空间.

heap sort

算法: c++

```
Template<class Type>void HeapSort(datalist<Type>&list)
{for(int i=(list.currentsize)/2;i>=1;i--)
```

```
    FilterDown(i,list.currentsize);
```

建堆

```
    for(i=list.currentsize;i>1;i--)
```

```
        {Swap(list.Vector[1],list.vector[i]);
```

根和最后一个交换

```
        FilterDown(1,i-1);
```

```
    }
```

```
}
```

去掉虚浅连接的
已出堆结点, 根结点
做下滤

heap sort

java program

```
public static void heapsort( Comparable [ ] a )
{ for( int i = a.length / 2; i >= 1; i-- )
    percDown( a, i, a.length );
  for( int i = a.length ; i > 1; i-- )
  { swapReferences( a, 1, i );
    percDown( a, 1, i-1);
  }
}
```

heap sort

```
private static void percDown( Comparable [ ] a, int i, int n )
{   int child;
    Comparable tmp;

    for( tmp = a[ i ]; leftChild( i ) < n; i = child )
    {   child = leftChild( i );
        if( child != n - 1 && a[child ].compareTo( a[ child + 1 ] ) < 0 )
            child++;
        if( tmp.compareTo( a[ child ] ) < 0 )
            a[ i ] = a[ child ];
        else break;
    }
    a[ i ] = tmp;
}

private static int leftChild( int i )
{   return 2 * i + 1;
}
```


6.4.Applications of Priority Queues

2. The Selection Problem

A decorative graphic consisting of several overlapping, horizontal, teardrop-shaped elements. The central element is a bright orange-yellow gradient, while the surrounding elements are in shades of brown and gold, creating a sense of depth and motion.

6.4.Applications of Priority Queues

2. The Selection Problem

在N个元素中找出第K个最大元素。

1A算法：读入N个元素放入数组，并将其选择排序，返回适当的元素。

运行时间： $O(N^2)$ (指最大不超过 N^2)

1B算法：

1) 将K个元素读入数组，并对其排序（按递减次序）。

最小者在第K个位置上。

2) 一个一个地处理其余元素：

每读入一个元素与数组中第K个元素(在K个元素中为最小)比较，
如果 $>$ ，则删除第K个元素，再将该元素放在合适的位置上。

如果 $<$ ，则舍弃。

最后在数组K位置上的就是第K个最大元素。

例如：3, 5, 8, 9, 1, 10 找第3个最大元素。

6.4.Applications of Priority Queues

运行时间(1B 算法):

$$O(K^2 + (N - K) * K)$$

$$= O(N * K)$$

$$\text{当 } K = \lceil N/2 \rceil, O(N^2)$$

试验：在 $N = 100$ 万个元素中，找第 500,000 个最大元素。

以上两个算法在合理时间内均不能结束，都要处理若干天才算完。

用堆来实现：

6A 算法：假设求第 K 个最小元素

1) 将 N 个元素建堆（最小） $O(N)$

2) 执行 K 次 delete

$$O(K * \log N)$$

$$O(N + K * \log N)$$

如果 $K = \lceil N/2 \rceil$,

$$\theta(N * \log N)$$

如果 $K = N$,

$$O(N * \log N)$$

—— 堆排序

6.4.Applications of Priority Queues

6B算法：假设求第K个最大元素

1) 读入前K个元素，建立最小堆 $O(K)$

2) 其余元素一一读入：

每读入一个元素与堆中第K个最大元素比（实际上是堆中最小元素）

$O(1)$

大于，则将小元素去掉（堆顶），该元素进入，进行一次调整。

$O(\log K)$

小于，则舍弃。

$$O(K + (N-K) * \log K) = O(N * \log K)$$

当 $K = \lceil N/2 \rceil$, $\theta(N * \log N)$

下堆

对6A, 6B,用同样的数据进行测试， 只需几秒钟左右给出问题解。

Chapter 6

2009年统考题:

8. 已知关键字序列 5, 8, 12, 19, 28, 20, 15, 22 是最小根堆(最小堆), 插入关键字 3, 调整后得到的小根堆是

A. 3, 5, 12, 8, 28, 20, 15, 22, 19

B. 3, 5, 12, 19, 20, 15, 22, 8, 28

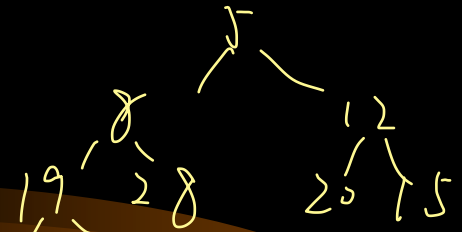
C. 3, 8, 12, 5, 20, 15, 22, 28, 19

D. 3, 12, 5, 8, 28, 20, 15, 22, 19

exercises:

- a. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.
b. Show the result of using the linear-time algorithm to build a binary heap using the same input.
- Show the result of performing three deleteMin operations in the heap of the previous exercise.

把3放到最后做上浮



Chapter 6

3. 判别以下序列是否是堆？如果不是，将它调整为堆。

1) { 100, 86, 48, 73, 35, 39, 42, 57, 66, 21 }

2) { 12, 70, 33, 65, 24, 56, 48, 92, 86, 33 }

3) { 103, 97, 56, 38, 66, 23, 42, 12, 30, 52, 06, 20 }

4) { 05, 56, 20, 23, 40, 38, 29, 61, 35, 76, 28, 100 }

4. 设待排序的关键码序列为{ 12, 2, 16, 30, 28, 10, 16*, 20, 6, 18 }, 使用堆排序方法进行排序。写出建立的初始堆, 以及调整的每一步。