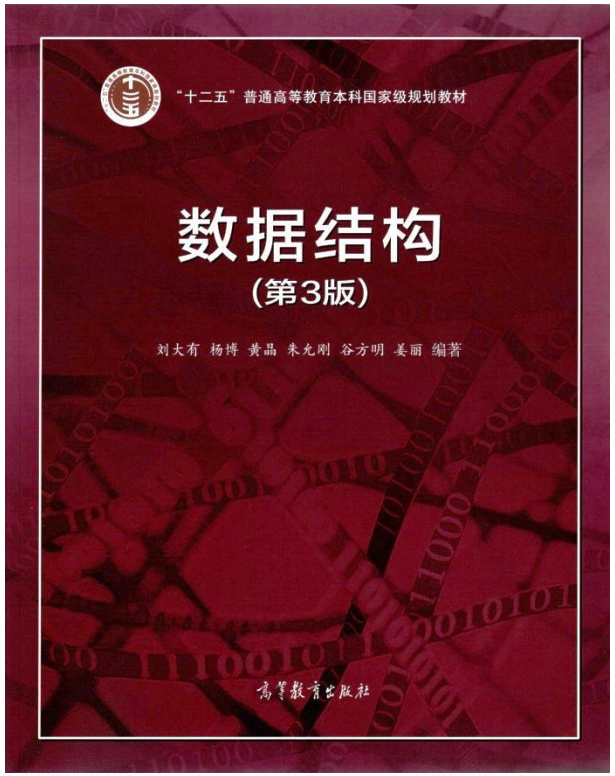


堆与优先级队列

- 堆的插入删除
- 优先级队列及应用
- Top K问题

数据之法 结构之美 算法之道

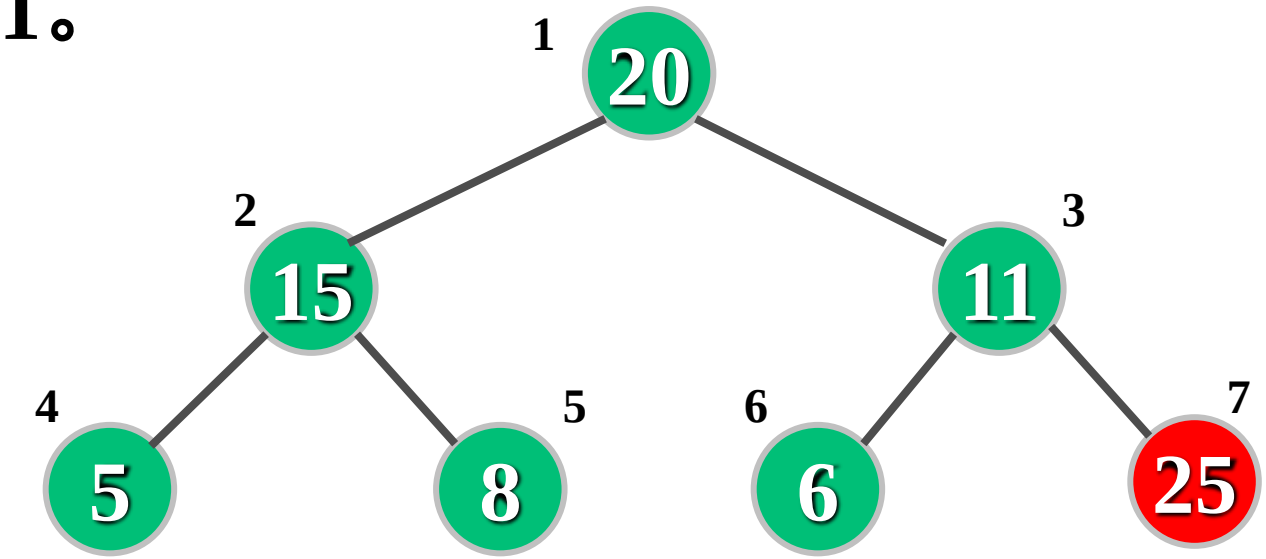
zhuyungang@jlu.edu.cn

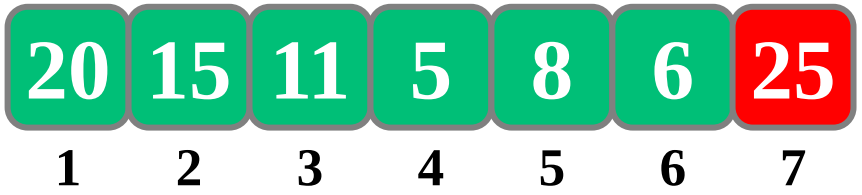
Last updated on 2024.11

堆的插入操作

B

- 目的：$(R_1, \cdots, R_n)$是一个堆，插入一个新元素R_{n+1}，使$(R_1, \ldots, R_{n+1})$成为一个堆。
- 做法：把新插入的元素放在堆尾，再对其做上浮操作，堆的元素个数加1。





插入元素
25

堆的插入操作

B

```
void Insert(int R[], int &n, int x){ //堆尾插入值x
    R[++n] = x;          //x放在R[n+1]处，堆元素个数加1
    ShiftUp(R, n, n); //元素R[n]上浮
}
```

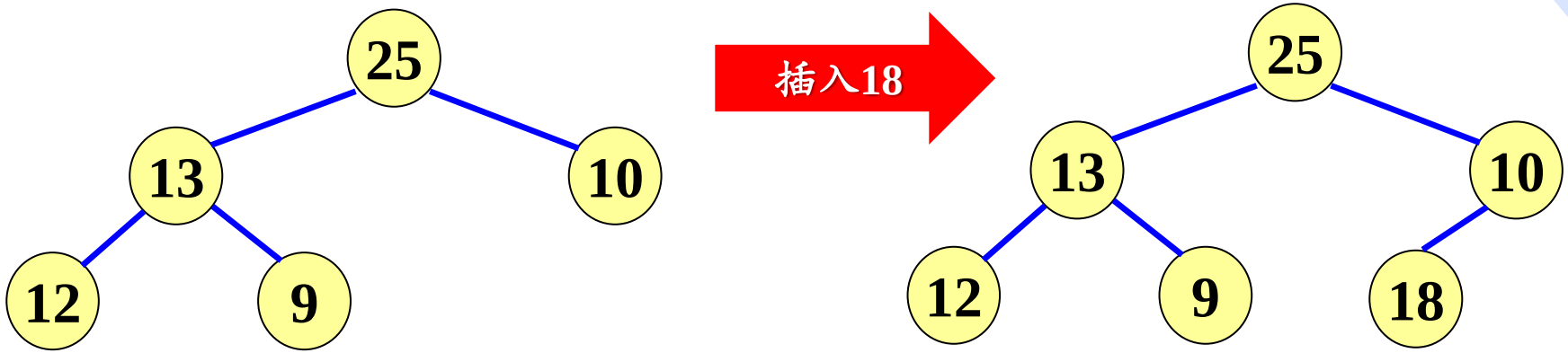
时间复杂度
O(log*n*)

课下思考

B

已知序列(25, 13, 10, 12, 9)是大根堆，在此序列尾部插入新元素18，将其再调整为大根堆，调整过程中元素比较次数为_____。【考研题全国卷】

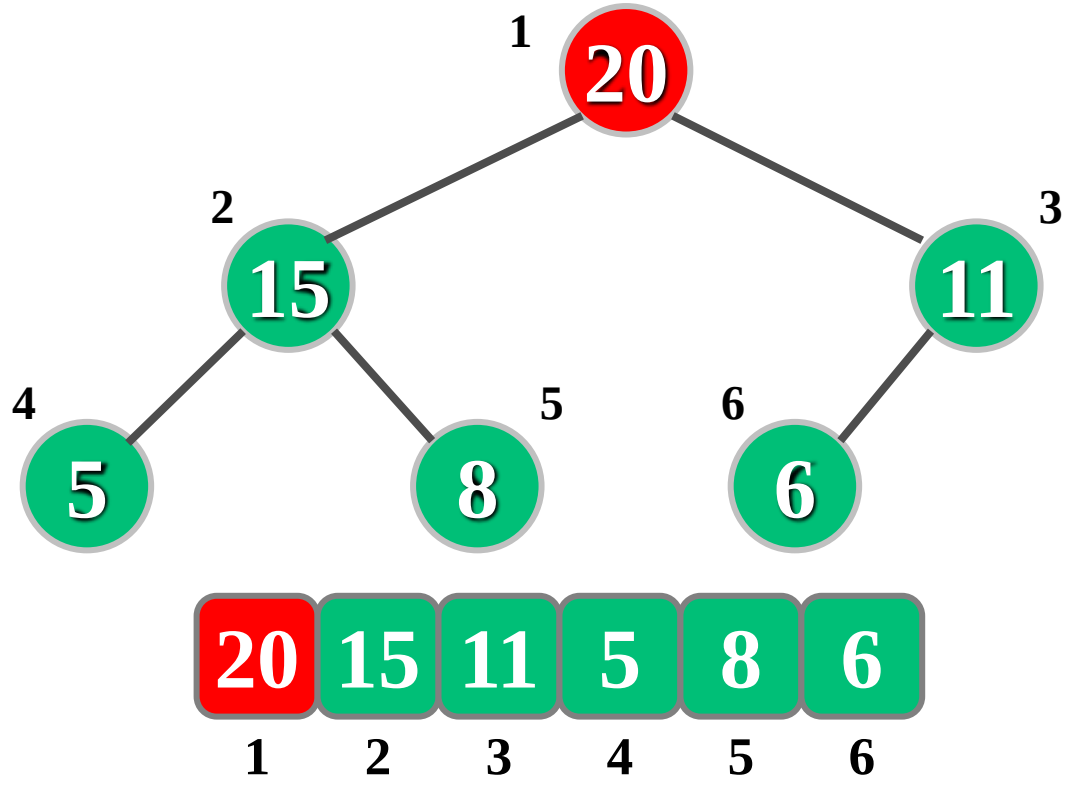
[A] 1　　[B] 2　　[C] 4　　[D] 5



堆的删除操作

B

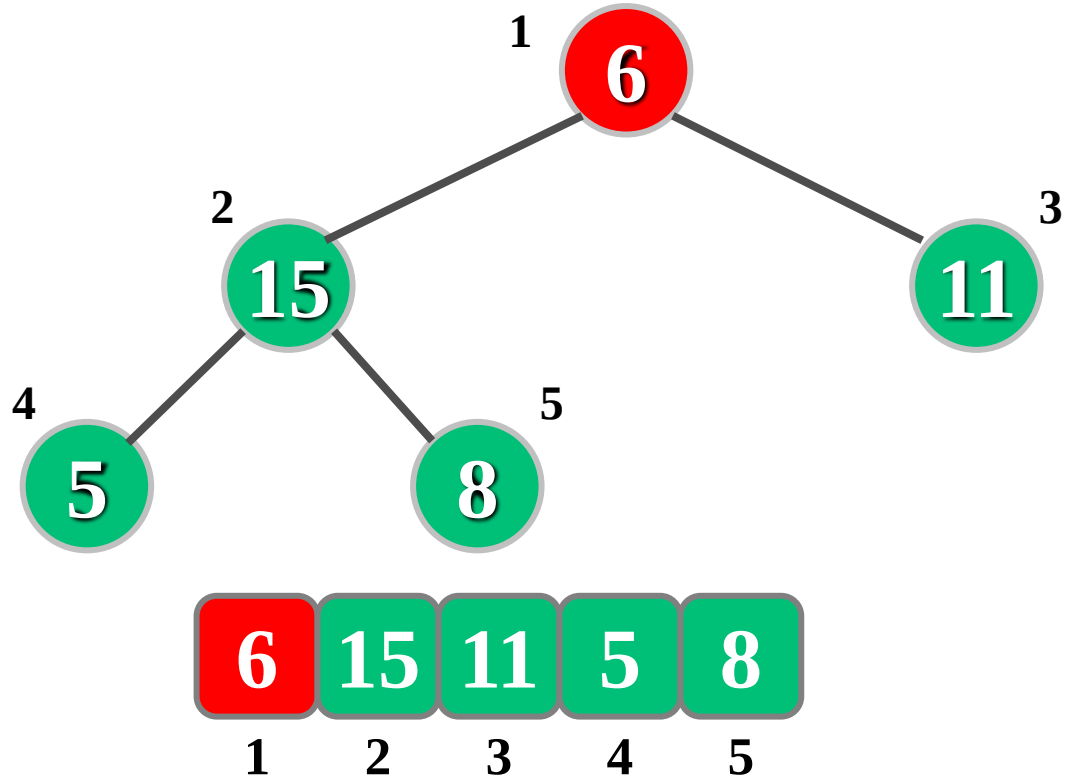
- 目的：$(R_1, R_2, \ldots, R_n)$是一个堆，从堆中删除并返回堆顶（即取出堆内最大的元素），且删除后的文件仍为堆。
- 做法：把R[n]移到根，堆元素个数减1，再对根R[1]做下沉操作。



堆的删除操作

B

- 目的：$(R_1, R_2, \ldots, R_n)$是一个堆，从堆中删除并返回堆顶（即取出堆内最大的元素），且删除后的文件仍为堆。
- 做法：把R[n]移到根，堆元素个数减1，再对根R[1]做下沉操作。



堆的删除操作

B

```
int DelMax(int R[],int &n){ //删除并返回堆顶,假设堆非空
    int MaxKey=R[1];   //暂存堆顶元素
    R[1] = R[n--];     //堆尾移至堆顶，堆元素个数减1
    ShiftDown(R,n,1); //新堆顶R[1]下沉
    return MaxKey;
}
```

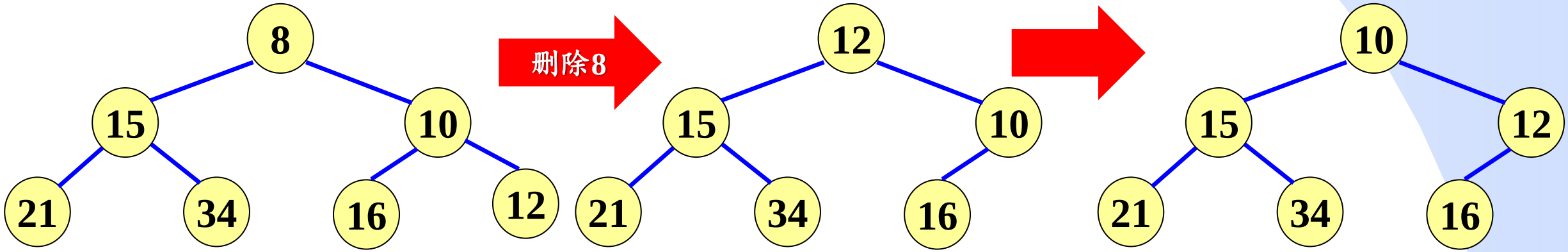
时间复杂度
O(log*n*)

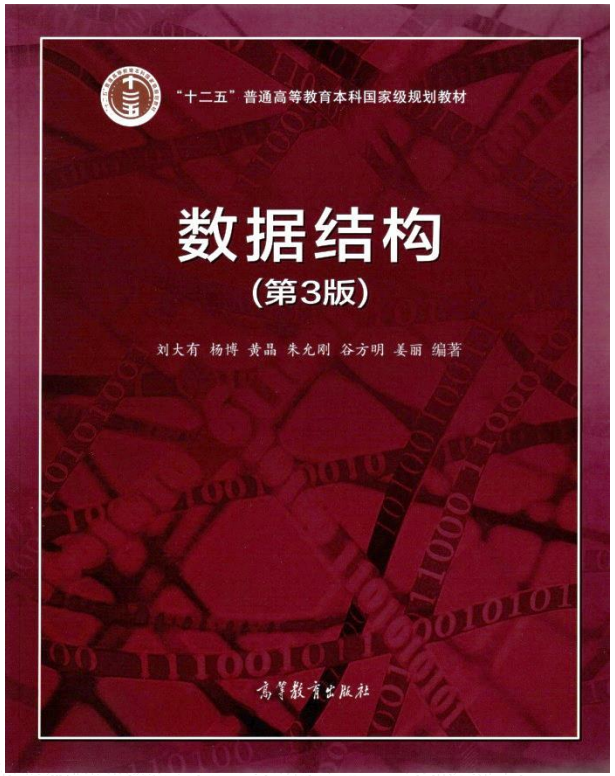
课下思考

B

已知小根堆(8, 15, 10, 21, 34, 16, 12)，删除关键字8后需要重建堆，在此过程中，关键词比较次数为_____。【考研题全国卷】

[A] 1　　[B] 2　　[C] 3　　[D] 4





堆与优先级队列

- 堆的插入删除
- 优先级队列及应用
- Top K问题

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2024.11

优先级队列（Priority Queue）

C

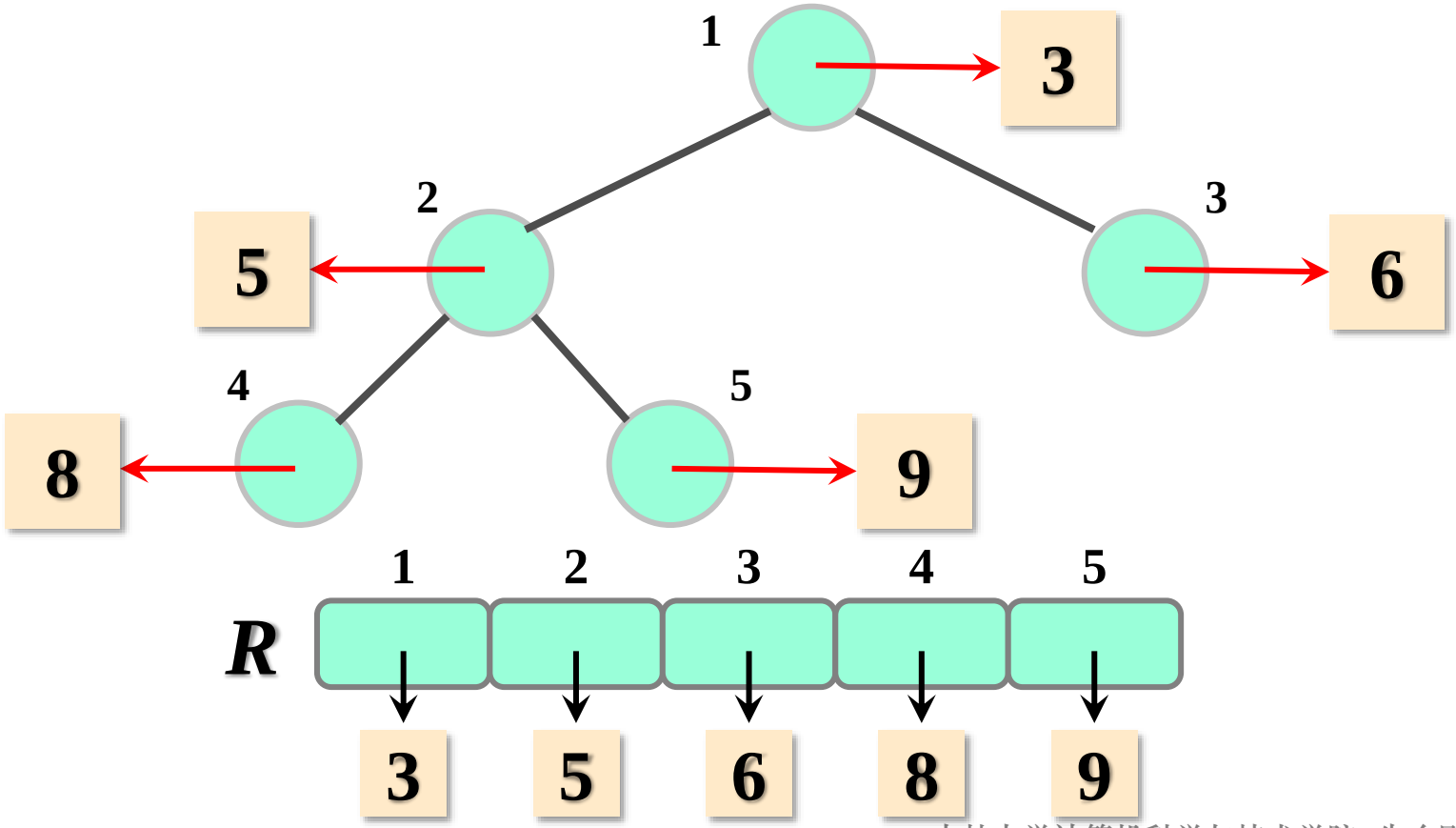
- 为每个元素设置一个优先级。
- 出队顺序：优先级高的元素先出队，优先级低的元素后出队。
- 可以用堆实现优先级队列，优先级对应关键词大小。
- 入队：堆的插入，时间O(log*n*)
- 出队：堆的删除，时间O(log*n*)
- 可以把优先级队列实现成类，插入、删除等操作作为成员函数，存储堆的数组*R*作为数据成员。

```
const int maxn=1e5+10;
Template<class T>
class PriorityQueue{
public:
    void Insert(T K);
    T DelMax();
    bool Empty();
    ……
private:
    void ShiftUp(int i);
    void ShiftDown(int i);
    T R[maxn]; //存堆元素
    int n; //堆的元素个数
    ……
};
```

优先级队列（最小堆）优化Huffman算法

C

- 堆的元素：哈夫曼树的结点指针（即HuffmanNode*类型）
- 堆元素R[i]的关键词：R[i]->weight
- 用堆选取weight最小的结点




```
const int maxn=1e5+10;
Template<class T>
class MinPQ{
public:
    void Insert(T K);
    T DelMin();
    bool Empty();
    ……
private:
    void ShiftUp(int i);
    void ShiftDown(int i);
    T R[maxn]; //存堆元素
    int n;  //堆的元素个数
    ……
};
```

优先级队列（最小堆）优化Huffman算法

C

```
MinPQ<HuffmanNode*> pq;        //基于最小堆创建优先级队列
   ... ...
for (int i=0; i<n-1; i++){
      HuffmanNode *t = new HuffmanNode;
      t->left = pq.DelMin(); //取出权值最小的结点
      t->right= pq.DelMin(); //取出权值最小的结点
      t->weight = t->left->weight + t->right->weight;
      pq.Insert(t);                  //新创建的结点插入堆
}
```

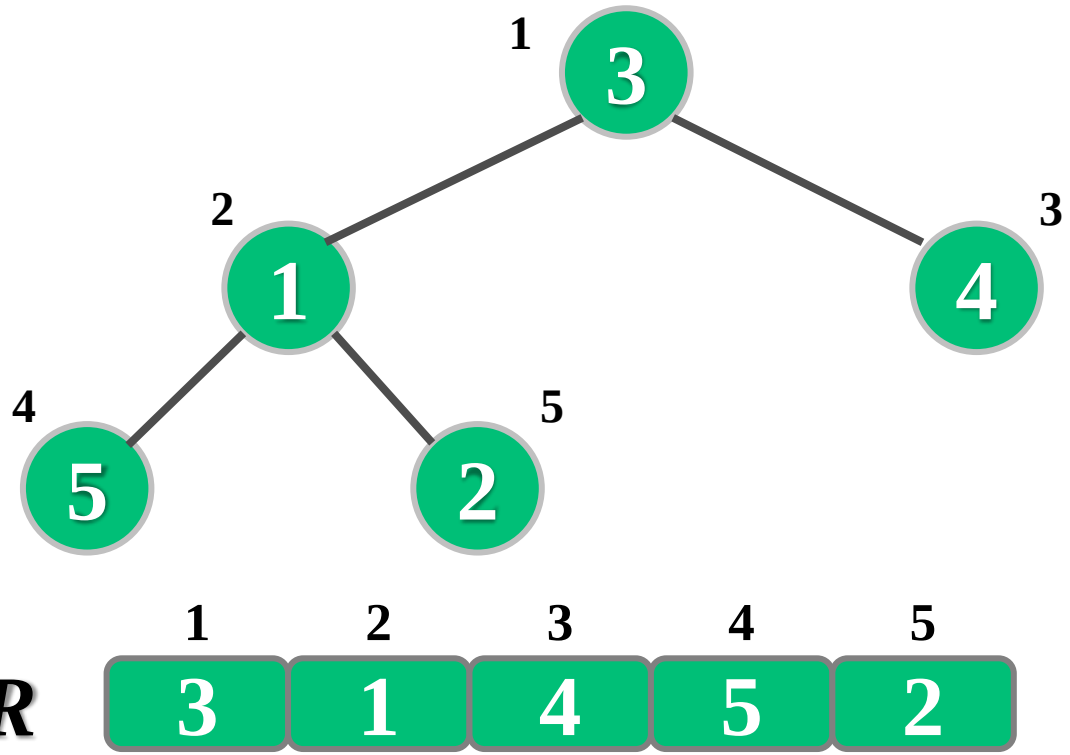
时间复杂度
$O(n\log n)$

用堆选取weight
最小的结点

优先级队列（最小堆）优化Dijkstra算法

C

- 堆的元素：图中顶点的编号
- 堆元素v的关键词：dist[v]
- 用堆选取dist值最小的顶点




```
const int maxn=1e5+10;
Template<class T>
class MinPQ{
public:
    void Insert(int v, T key);
    int DelMin();
    bool Empty();
    ……
private:
    void ShiftUp(int i);
    void ShiftDown(int i);
    int R[maxn]; //存堆元素
    T dist[maxn];//存关键词
    int n; //堆的元素个数
    ……
};
```

堆优化的Dijkstra算法（方案1）

C

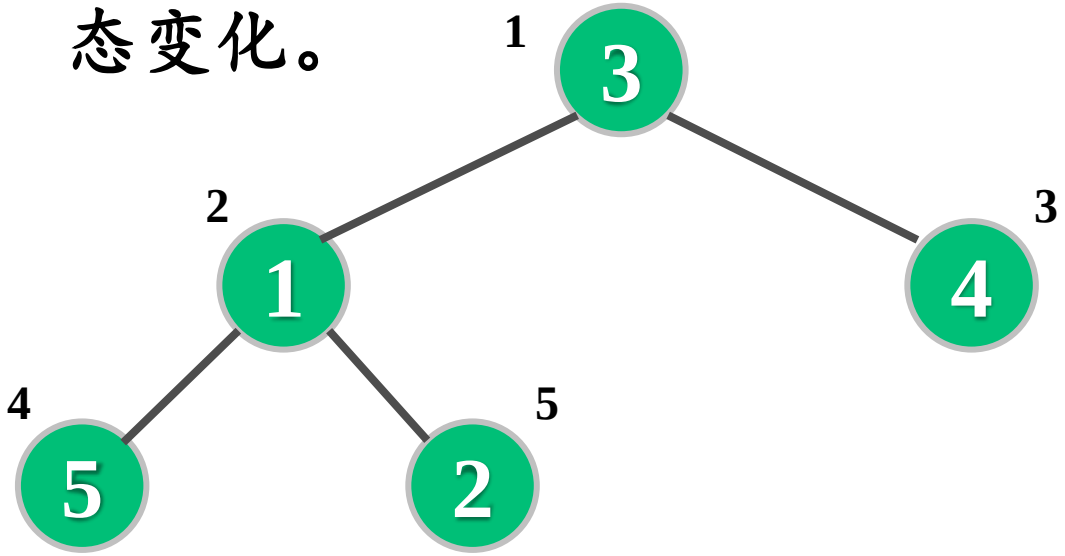
```
void Dijkstra(Vertex *Head, int n, int u, int dist[]){
   int S[N]={0}; indexMinPQ<int> pq;
   for(int i=1; i<=n; i++) dist[i]=(i==u)?0:INF;
   pq.Insert(u, 0);
   while(!pq.Empty()) {
      int v = pq.DelMin();          //选dist值最小的顶点v
      S[v]=1;                       //将顶点v放入S集合
      for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link){
         int w=p->VerAdj; //更新v的各邻接顶点w的dist值
         if(S[w]==0 && dist[v]+p->cost < dist[w]){
            dist[w] = dist[v]+p->cost;
            if(!pq.contain(w)) pq.Insert(w, dist[w]);
            else pq.DecreaseKey(w, dist[w]);
         }
      }
   }
}
```

DecreaseKey(w,d):将顶点w的关键词减小为d

如何实现DecreaseKey(*w*, *d*)?

C

- 将顶点*w*的*dist*值减小为*d*，并将点*w*上浮；时间复杂度O(log*n*)
- 如何在堆中高效查找顶点*w*?增加一个pos数组，pos[i]表示图中编号为i的顶点在堆数组*R*中的下标；
- 在顶点插入和删除时，其pos值应动态变化。



	1	2	3	4	5
R	3	1	4	5	2
pos	2	5	1	3	4

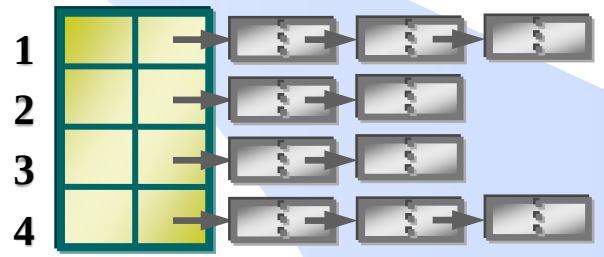
```
const int maxn=1e5+10;
Template<class T>
class indexMinPQ{
public:
    ……
    void DecreaseKey(int w,T d);
private:
    ……
    int R[maxn];    //存堆元素
    T dist[maxn];   //存关键词
    int pos[maxn]; //存元素在R中下标
    int n; //堆的元素个数
    ……
};
```

堆优化的Dijkstra算法（方案1）时间复杂度

C

```
void Dijkstra(Vertex *Head, int n, int u, int dist[]){
   int S[N]={0}; indexMinPQ<int> pq;
   for(int i=1; i<=n; i++) dist[i]=(i==u)?0:INF;
   pq.Insert(u, 0);
   while(!pq.Empty()) {
      int v = pq.DelMin();          //选dist值最小的顶点v
      S[v]=1;                       //将顶点v放入S集合
      for(Edge* p=Head[v].adjacent; p; p=p->link){
         int w=p->VerAdj; //更新v的各邻接顶点w的dist值
         if(S[w]==0 && dist[v]+p->cost < dist[w]){
            dist[w] = dist[v]+p->cost;
            if(!pq.contain(w)) pq.Insert(w, dist[w]);
            else pq.DecreaseKey(w, dist[w]);
         }
      }
   }
}
```

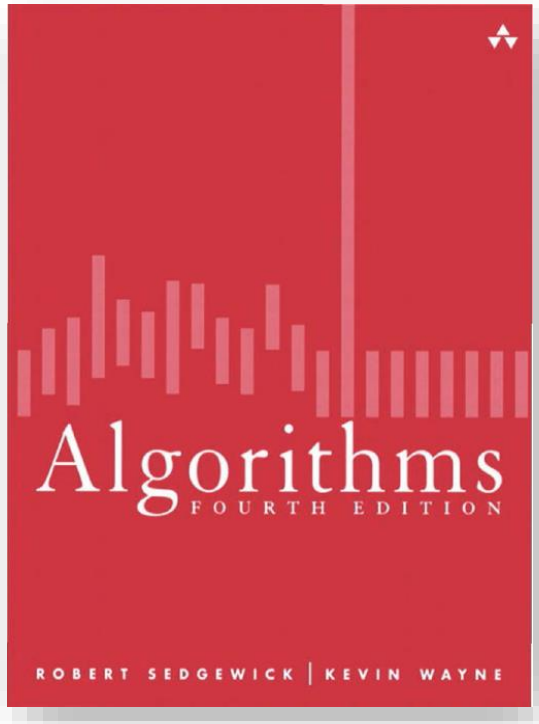
时间复杂度
$O((n+e)\log n)$
适合稀疏图

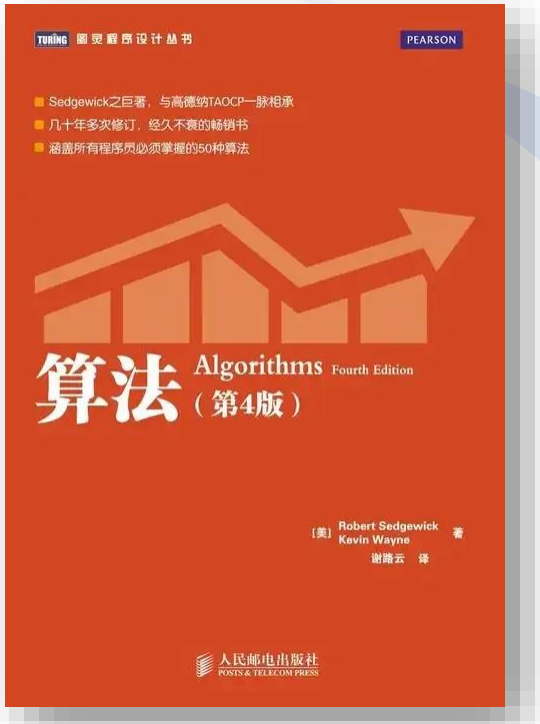


时间复杂度：$n \cdot T(\text{DelMin}) + e \cdot T(\text{Insert/DecreaseKey})$

索引优先级队列

- 关于indexMinPQ的更详细介绍，可参考：





R. Sedgewick, K. Wayne著，谢路云译. 算法（第4版）. 人民邮电出版社. 2012.

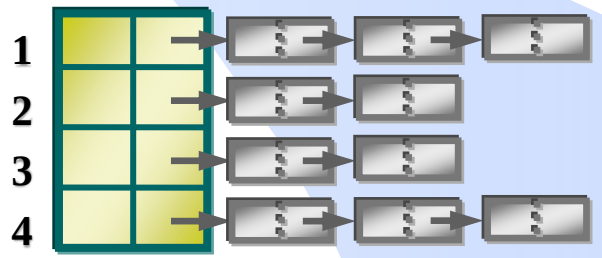
堆优化的Dijkstra算法（方案2, 更简单）

C

```
void Dijkstra(Vertex *Head, int n, int u, int dist[]){
   int S[N]={0}; MinPQ<int> pq;
   for(int i=1; i<=n; i++) dist[i]=(i==u)?0:INF;
   pq.Insert(u, 0);
   while(!pq.Empty()) {
      int v = pq.DelMin(); if(S[v]==1) continue;
      S[v]=1;                    //将顶点v放入S集合
      for(Edge* p=Head[v].adjacent; p; p=p->link){
         int w=p->VerAdj; //更新v的各邻接顶点w的dist值
         if(S[w]==0 && dist[v]+p->cost < dist[w]){
            dist[w] = dist[v]+p->cost;
            pq.Insert(w, dist[w]);
         }
      }
   }
}
```

时间：$e \cdot T(\text{DelMin}) + e \cdot T(\text{Insert})$

课下思考：时间复杂度O(*e*log*e*)。提示：堆内最多可能有*e*个顶点



(5, 13)
(5, 20)

Dijkstra
基于优先级队列的BFS

课下阅读：C++ STL实现堆优化的Dijkstra算法

C

- priority_queue：C++ STL内置的优先级队列，默认为最大堆。若需要最小堆，可对关键词取负存入堆中。常用方法如下：

方法	功能	时间复杂度
push()	把元素插入堆	O(log*n*)
pop()	删除堆顶元素	O(log*n*)
top()	返回堆顶元素	O(1)
size()	返回堆元素个数	O(1)

- pair<>:C++ STL内置的二元组，尖括号中分别指定二元组的第一元、第二元的类型，可以用make_pair函数创建二元组，用成员变量first访问第一元、second访问第二元。在比较大小时，以第一元为第一关键词、第二元为第二关键词。

课下阅读：C++ STL实现堆优化的Dijkstra算法

C

```
void Dijkstra(Vertex *Head, int n, int u, int dist[]){
    int S[N]={0};  priority_queue<pair<int,int>> pq;
    for(int i=1; i<=n; i++) dist[i]=(i==u)?0:INF;
    pq.push(make_pair(0, u));
    while(pq.size()) {
      int v = pq.top().second(); pq.pop();
      if(S[v]==1) continue;
      S[v]=1;   //将顶点v放入S集合
      for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link){
         int w=p->VerAdj; //更新v的各邻接顶点w的dist值
         if(S[w]==0 && dist[v]+p->cost < dist[w]){
            dist[w] = dist[v]+p->cost;
            pq.push(make_pair(-dist[w], w));
         }
      }
    }
}
```

堆优化的Dijkstra算法时间复杂度对比

C

	DelMin	*Insert*	*DecreaseKey*	*Dijkstra*算法
二叉堆	O(log*n*)	O(log*n*)	O(log*n*)	O((*n*+*e*)log*n*)
斐波那契堆	O(log*n*)	均摊O(1)	均摊O(1)	O(*n*log*n*+*e*)

*Dijkstra*算法时间复杂度：*n*·*T*(DelMin)+*e*·*T*(Insert/DecreaseKey)

Robert Tarjan
图灵奖获得者
普林斯顿大学教授
美国科学院/工程院院士
提出斐波那契堆

优先级队列（最小堆）优化Prim算法

C

- 若使用邻接表存图，堆优化的Prim算法时间复杂度为$O((n+e)\log n)$；
- 适合稀疏图，但不如Kruskal算法方便。

图的搜索策略

C

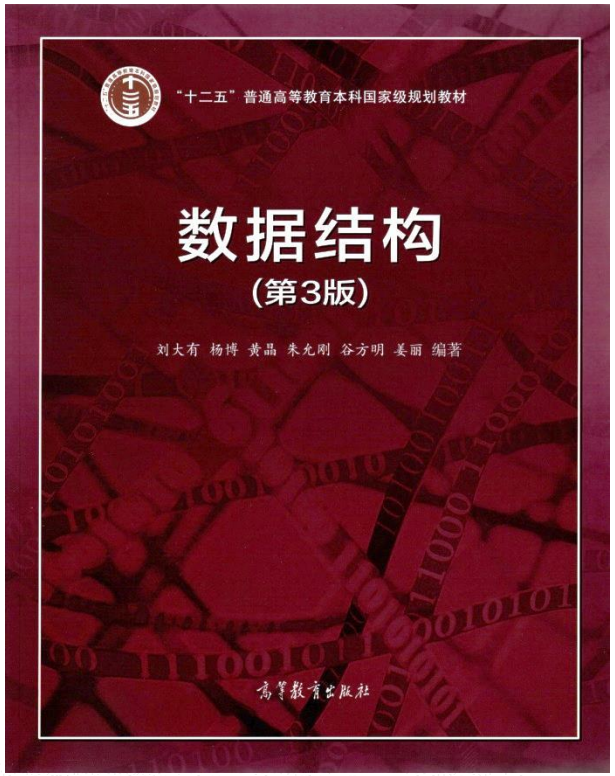
- ***DFS*：*Depth First Search*（深度优先搜索）**
- ***BFS*：*Breath First Search*（广度优先搜索）**
- ***PFS*：*Priority First Search*（优先级优先搜索）**
  - ✓**Dijkstra算法：优先级—*dist***
  - ✓**Prim算法：优先级—*Lowcost***

图搜索的一种统一框架

C

```
void GraphSearch(Graph G,int u){ //在图G中以u为起点进行某种搜索
    int visited[N]={0};
    CONTAINER container; //定义一个容器
    container⇐u;        //把起点放入容器
    while(!container.Empty()) {
       int v⇐container; //从容器中取出点v
       if(visited[v]==1) continue;
       visit(v); visited[v]=1;
       for(each w in neigbours(v)) //考察v的每个邻接顶点w
          if(visited[w]==0 && …)    //若w未被处理过且满足某些条件
             container⇐w;
    }
}
```

容器	搜索策略
栈	*DFS*
队列	*BFS*
优先级队列	*PFS*



堆与优先级队列

- 堆的插入删除
- 优先级队列及应用
- **Top K问题**

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2024.11

热搜榜

C

数组有n个元素，挑选其中最大的k $(k < n)$个元素。

TOP K问题

C

数组有n个元素，挑选其中最大的k ($k < n$)个元素。【华为、腾讯、字节跳动、阿里、京东、美团、苹果、微软、谷歌面试题】

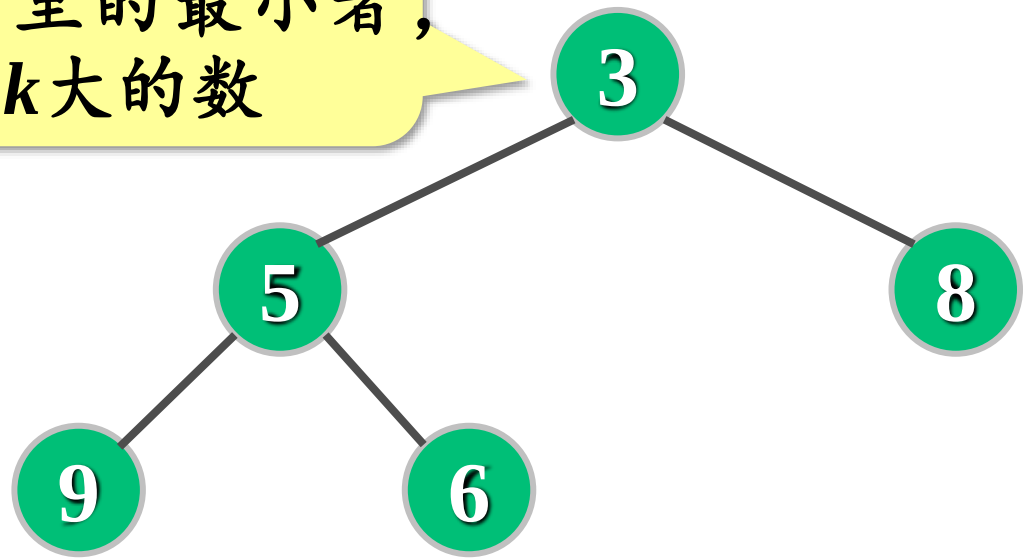
- 想法1：递减排序后，选前k个数，O($n\log n$)。
- 想法2：借助直接选择排序思想，选k次最大元素，O(nk)。
- 想法3：借助堆，取出k次堆顶元素，O($n+k\log n$)。

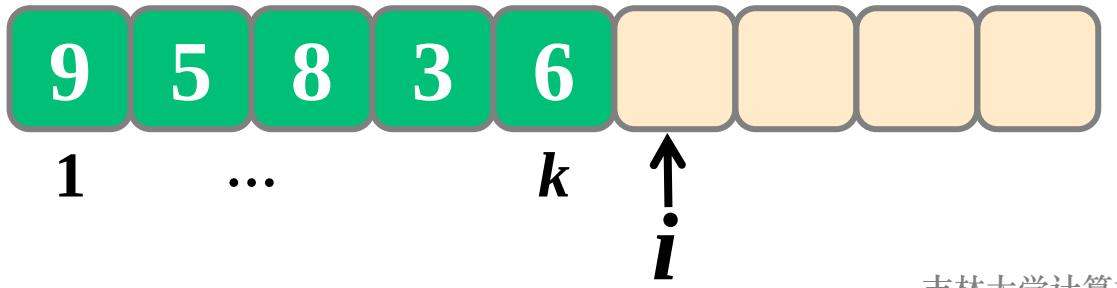
TOP K问题

C

使用包含k个元素的小根堆维护数组中最大的k个元素。

堆顶是目前Top k元素里的最小者，即第k大的数





① 用数组前k个元素建堆，此时堆里存的就是当前扫描过的元素里的最大的k个元素。

② 依次扫描数组剩余元素，若扫描到的元素x大于堆顶，则用x替换堆顶，下沉x以重建堆。使堆始终保存当前扫描过的元素里的Top k元素。

时间复杂度O(*n*log*k*)
适合增量环境