

# 1 ADT Model

```

1 PriorityQueue Initialize( int MaxElements );
2 void Insert( ElementType X, PriorityQueue H );
3 ElementType DeleteMin( PriorityQueue H );
4 ElementType FindMin( PriorityQueue H );

```

## 2 Simple Implementations

### ✍ Array :

**Insertion** — add one item at the end  $\sim \Theta(1)$

**Deletion** — find the largest \ smallest key  $\sim \Theta(n)$   
 remove the item and shift array  $\sim O(n)$

### ✍ Linked List :

**Insertion** — add to the front of the chain  $\sim \Theta(1)$

**Deletion** — find the largest \ smallest key  $\sim \Theta(n)$   
 remove the item  $\sim \Theta(1)$

### ✍ Ordered Array :

**Insertion** — find the proper position  $\sim O(n)$   
 shift array and add the item  $\sim O(n)$

**Deletion** — remove the first \ last item  $\sim \Theta(1)$

### ✍ Ordered Linked List :

**Insertion** — find the proper position  $\sim O(n)$   
 add the item  $\sim \Theta(1)$

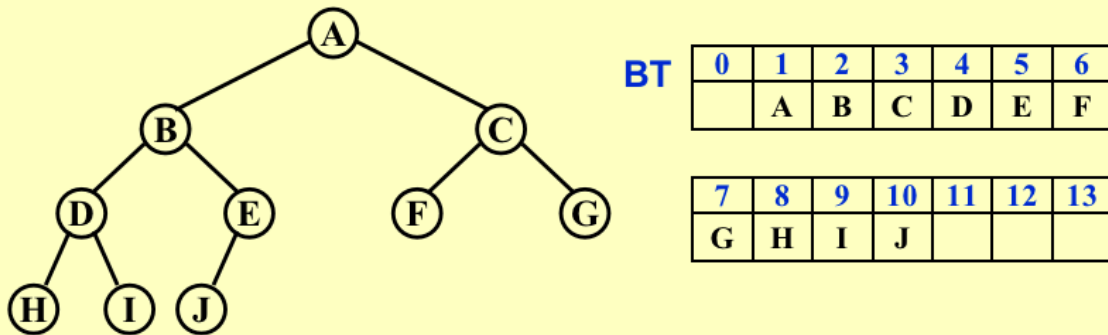
**Deletion** — remove the first \ last item  $\sim \Theta(1)$

# 3 Binary Heap

## 3.1 Structure Property

- **【Definition】** A binary tree with  $n$  nodes and height  $h$  is complete iff its nodes correspond to the nodes numbered from 1 to  $n$  in the perfect binary tree of height  $h$ .

### ❖ Array Representation : $BT[n+1]$ ( $BT[0]$ is not used)



- 先保证层次遍历连续，那么就可以用数组来存他，可以用公式找parent和child

**【Lemma】** If a complete binary tree with  $n$  nodes is represented sequentially, then for any node with index  $i$ ,  $1 \leq i \leq n$ , we have:

$$(1) \text{ index of } \textit{parent}(i) = \begin{cases} \lfloor i/2 \rfloor & \text{if } i \neq 1 \\ \text{None} & \text{if } i = 1 \end{cases}$$

$$(2) \text{ index of } \textit{left\_child}(i) = \begin{cases} 2i & \text{if } 2i \leq n \\ \text{None} & \text{if } 2i > n \end{cases}$$

$$(3) \text{ index of } \textit{right\_child}(i) = \begin{cases} 2i + 1 & \text{if } 2i + 1 \leq n \\ \text{None} & \text{if } 2i + 1 > n \end{cases}$$

```

1 PriorityQueue Initialize( int MaxElements )
2 {
3     PriorityQueue H;
4     if ( MaxElements < MinPQSize )

```

```

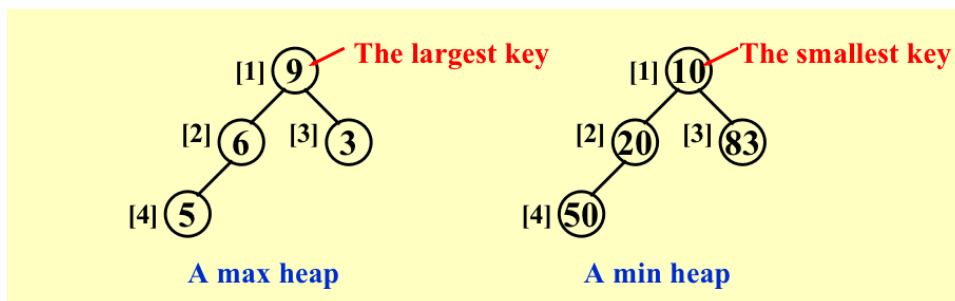
5     return Error( "Priority queue size is too small" );
6     H = malloc( sizeof ( struct HeapStruct ) );
7     if ( H == NULL )
8         return FatalError( "Out of space!!!" );
9     /* Allocate the array plus one extra for sentinel */
10    H->Elements = malloc(( MaxElements + 1 ) * sizeof( ElementType
    ));
11    if ( H->Elements == NULL )
12        return FatalError( "Out of space!!!" );
13    H->Capacity = MaxElements;
14    H->Size = 0;
15    H->Elements[ 0 ] = MinData; /* set the sentinel */
16    return H;
17 }

```

- 这里的0单元存了一个很小的数

## 3.2 Heap Order Property

- **【Definition】** A min tree is a tree in which the key value in each node is no larger than the key values in its children (if any). A min heap is a complete binary tree that is also a min tree.

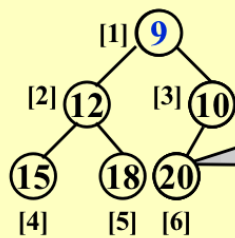


## 3.3 Basic Heap Operations

### 3.3.1 insertion

## 👉 insertion

### ➤ Sketch of the idea:



The only possible position for a new node since a heap must be a complete binary tree.

Case 1 : new\_item = 21  $20 < 21$  ✓

Case 2 : new\_item = 17  $20 > 17$   $10 < 17$  ✓

Case 3 : new\_item = 9  $20 > 9$   $10 > 9$  ✓

- 在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。根据这一属性，那么最大堆总是将其中的最大值存放在树的根节点。而对于最小堆，根节点中的元素总是树中的最小值。

```

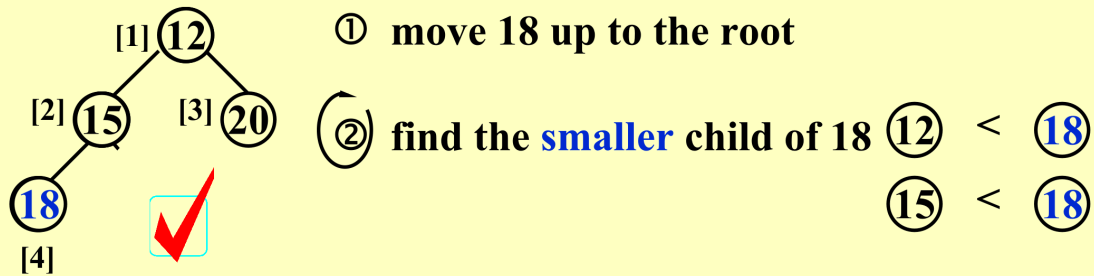
1  /* H->Element[ 0 ] is a sentinel */
2  // 实际上分为两步：找到合适位置（小于父节点 && 大于子节点），插入
3  void Insert( ElementType X, PriorityQueue H )
4  {
5      int i;
6
7      if ( IsFull( H ) ) {
8          Error( "Priority queue is full" );
9          return;
10     }
11
12     for ( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
13         H->Elements[ i ] = H->Elements[ i / 2 ];
14     // 循环
15     // 父节点的值换一下，要继续找值，最后会找到一个大于自己的父节点
16     // 退出，直接执行下面这条
17     H->Elements[ i ] = X; //对的，直接插进去就好了
18 }
19

```

- 0这个位置存了一个很小的数，是为了防止跳到根节点，一直在0这个节点死循环，因此置一个很小的数来使循环跳出

### 3.3.2 Delete Min

#### ➤ Sketch of the idea:



$$T(N) = O(\log N)$$

- 把最小的数删掉，直接上就是把根节点摘掉，但是要保持一个完整的树
- 先把最后一个数18移过来，然后根据自己和两个子节点来判断往哪移

```

1  ElementType DeleteMin( PriorityQueue H )
2  {
3      int i, Child;
4      ElementType MinElement, LastElement;
5      if ( IsEmpty( H ) ) {
6          Error( "Priority queue is empty" );
7          return H->Elements[ 0 ];
8      }
9      MinElement = H->Elements[ 1 ]; /* save the min element */
10
11     LastElement = H->Elements[ H->Size-- ]; /* take last and reset
size */
12
13     for ( i = 1; i * 2 <= H->Size; i = Child ) {
14         /* Find smaller child */
15         Child = i * 2;
16
17         if ( Child != H->Size && H->Elements[Child+1] < H-
>Elements[Child] )
18             Child++; // 右节点更小，那就选择右子节点作为下个可能要更换的节点
19
20         if ( LastElement > H->Elements[ Child ] )
21             /* Percolate one level */
22             // 说明此时的节点数值仍大于子节点数值，要更换，每次都是选三个里面最小的
那个换

```

```

23      // 要么就是和子节点换，那么就是进入if这里，要么就是自己是最小的
24      // 进入else，此时可以break跳出循环
25      H->Elements[ i ] = H->Elements[ Child ];
26      else break;
27      /* find the proper position */
28
29  }
30  H->Elements[ i ] = LastElement;
31  return MinElement;
32 }
33

```

### 3.4 Other Heap Operation

#### DecreaseKey ( $P, \Delta, H$ )

*Percolate up*



Lower the value of the key in the heap  $H$  at position  $P$  by a positive amount of  $\Delta$ .....so my programs can run with highest priority ☺.


#### IncreaseKey ( $P, \Delta, H$ )

*Percolate down*




Increases the value of the key in the heap  $H$  at position  $P$  by a positive amount of  $\Delta$ .....drop the priority of a process that is consuming excessive CPU time.

**Delete ( P, H )** DecreaseKey(P, ∞, H); DeleteMin(H)

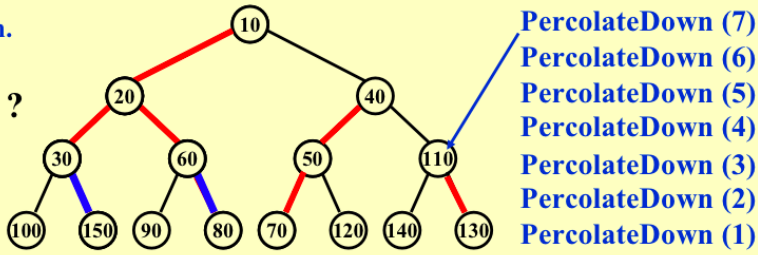
 **Remove the node at position P from the heap H ..... delete the process that is terminated (abnormally) by a user.**

**BuildHeap ( H )** Nehhhhhh that would be tooooo slow !

 **Place N input keys into an empty heap H.**

150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130

$T(N) = ?$



PercolateDown (7)  
PercolateDown (6)  
PercolateDown (5)  
PercolateDown (4)  
PercolateDown (3)  
PercolateDown (2)  
PercolateDown (1)

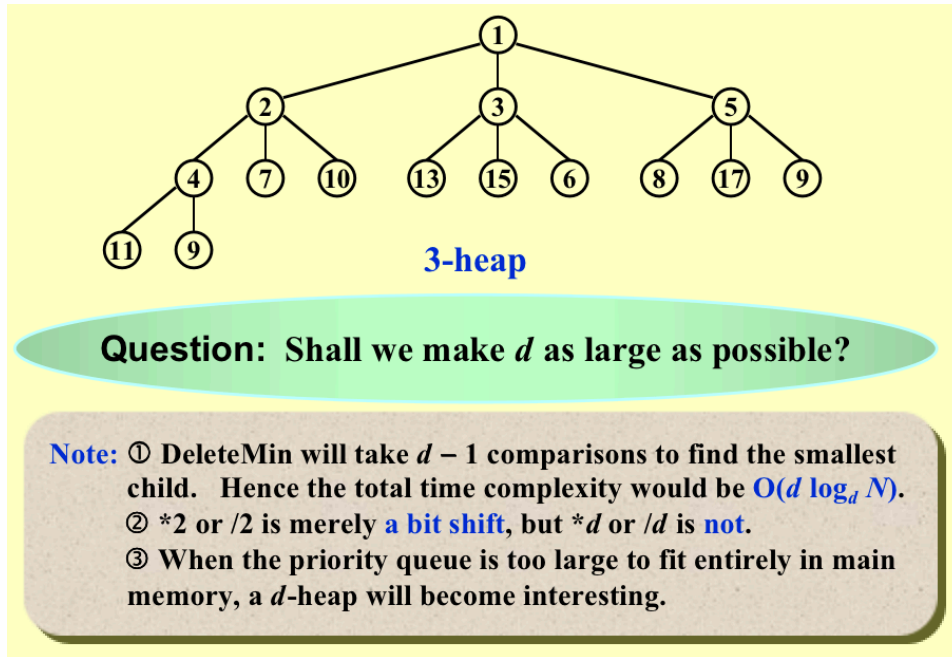
### 3.建堆

- (1)首先把数组按照层序 (level order) 放在一个空堆中
- (2)从最后一个父节点开始, 让父节点, 右孩子, 左孩子中**最小的放在父节点的位置**。
- (3)如果父节点被换下去了, 那么必须执行shiftdown操作, 即被换下去的结点与当前的子节点比较, 并交换, 直到符合比任何一个子节点大的条件。

- $T(N) = O(N)$ , 最多需要 $2N-2$ 次

## 4 Application

# 5 d-Heaps ---- All nodes have d children



- Delete Min : 每层进行 $d$ 次比较，找最小；一共走 $\log_d N$ 层。故  $T(N) = O(d \log_d N)$ .

## 6 Problems

- If a binary search tree of  $N$  nodes is complete, which one of the following statements is FALSE?
  - the maximum key must be at a leaf node (F) 可以没有右节点
  - the median node must either be the root or in the left subtree (T) 完整二叉搜索树是左节点多于右边的，中位数偏向左边
- 完整二叉搜索树：一棵深度为 $k$ 的有 $n$ 个结点的**二叉树**，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 $i$  ( $1 \leq i \leq n$ ) 的结点与**满二叉树**中编号为 $i$ 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。
- 也就是符合堆的排列的那种上面、左边先填满
- If a complete binary tree with 137 nodes is stored in an array (root at position 1), then the nodes at positions 128 and 137 are at the same level.(T)



$$N = \frac{a_1 \cdot (1 - q^n)}{1 - q} + M = q^n - 1 + M = 2^n - 1 + M \quad (1)$$

- 前127个，符合 $2^7 - 1 = 127$ ，他们位于一个完整无多的二叉树，后面的位于同一层