Unit 3
Priority Queues and Heaps

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 6.

R. Sedgewick, "Algorithms in C++", Chapter 11.

Priority Queues

- 由 priority 大小來決定處理次序的 queue, 可視為一般 queue 及 stack 的推廣
- 應用相當廣：Huffman Code, Shortest Path, Minimum Spanning Tree, Scheduling, O.S., … etc.

Some Operations

- ***Construct*** a priority queue from ***n*** given items.
- ***Insert*** a new item.
- ***Remove*** the largest item. (sometimes we may use the term Extract-Max)
- ***Replace*** the largest item with a new item (unless the new item is larger).
- ***Change*** the priority of an item.
- ***Delete*** an arbitrary specified item.
- ***Join*** two priority queues into one large one.

Some Equivalent Operations

☛有些 operation 可以被其他數個 operations 取代，但 performance 可能降低，例：

Construct(n) ≅ Construct(0) + n insertions

Replace( ) ≅ insert( ) + remove( )
note: not remove( ) + insert( )

change( ) ≅ delete( ) + insert( )

用 Priority Queue 做 sorting:

- ☛ construct(n) + n removes

  or

- ☛ construct(0) + n insertions + n removes

基本 implementations:

- ☛ 用 unsorted list 做 priority queue

  用在 sort ≅ ________ sort
- ☛ 用 sorted list 做 priority queue

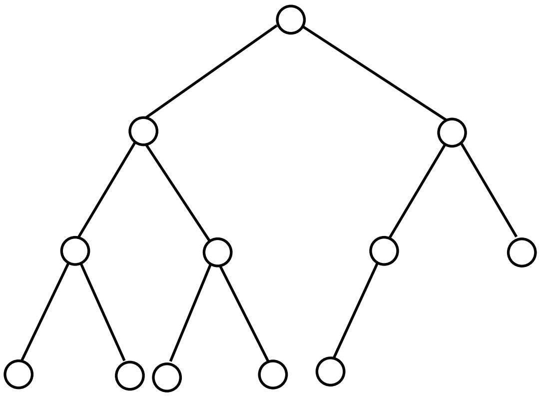
  用在 sort ≅ ________ sort
- ☛ 用 heap 做 priority queue

  用在 sort ≅ ________ sort
- ☛ 用 binary search tree 做 priority queue

  用在 sort ≅ ________ sort

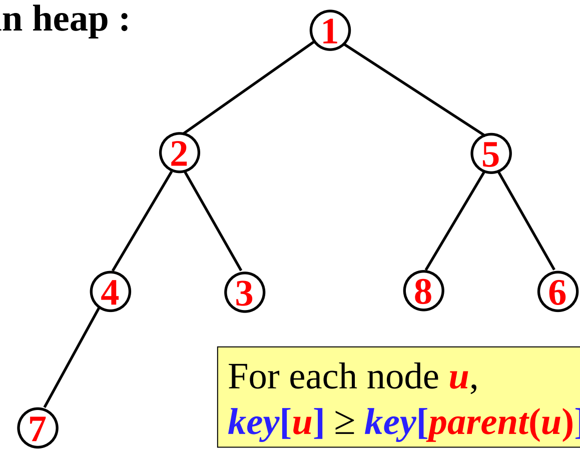
The Data Structure of Heap (1)

- An almost complete binary tree:



The Data Structure of Heap (2)

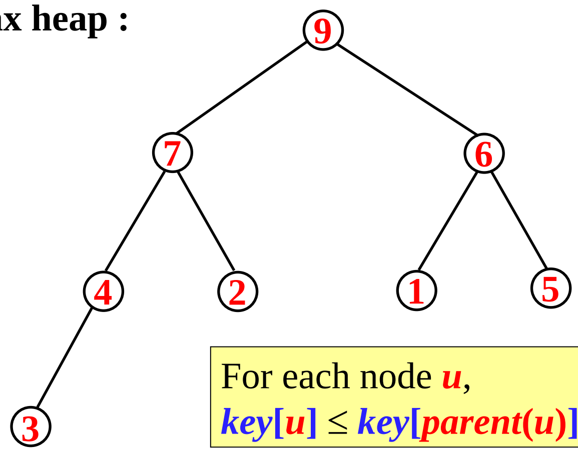
- A min heap :



For each node ***u***,
***key*[*u*]** $\geq$ ***key*[*parent*(*u*)]**.

The Data Structure of Heap (3)

- **A max heap :**

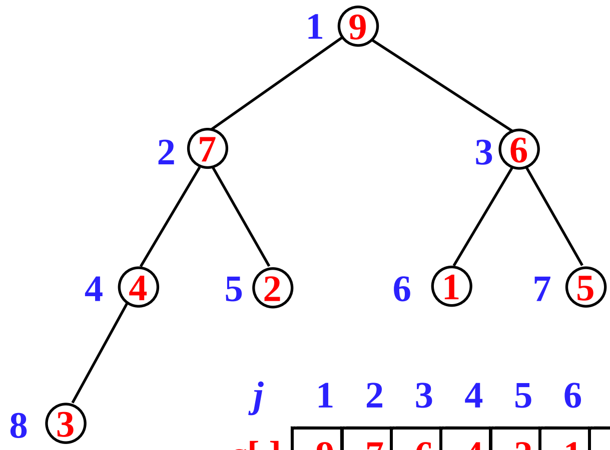


For each node ***u***,
***key*[u]** ≤ ***key*[*parent*(u)]**.

為何用 almost complete binary tree (1)

- 由 heap 的定義，理論上任何一種 rooted tree 都可拿來當 heap，那為何要用 almost complete binary tree？

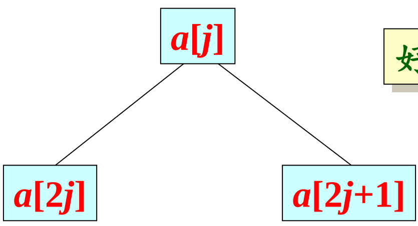
為何用 almost complete binary tree (2)



j	1	2	3	4	5	6	7	8
a[ ]	9	7	6	4	2	1	5	3

為何用 almost complete binary tree (3)

☛ 可用一 array ***a*[1..*n*]** 表示 (root 放在 ***a*[1]**)



好處1：不需 pointers

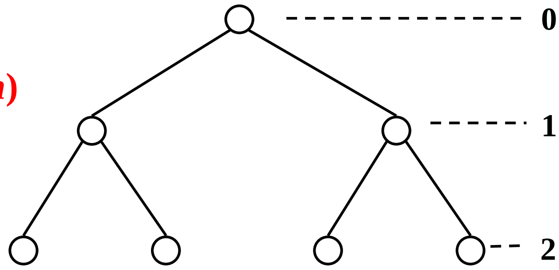
For any ***a*[*j*]**, its parent = ***a*[*j*/2]**

left child = ***a*[2*j*]**, right child = ***a*[2*j*+1]**

sibling = ***a*[*j*–1]** or ***a*[*j*+1]**

為何用 almost complete binary tree (4)

- 好處2：樹高 = $O(\lg n)$



$$\sum_{k=0}^{h-1} 2^k < n \le \sum_{k=0}^{h} 2^k \Rightarrow 2^h - 1 < n \le 2^{h+1} - 1$$

$$\Rightarrow \lg n - 1 < h \le \lg n \Rightarrow h = \lfloor \lg n \rfloor$$

基本模組 upheap( ) & insert( )

```
upheap(j) {
    itemtype v;
    v = a[j];  a[0] = ∞;
    while( a[j/2] < v) {
        a[j] = a[j/2];
        j = j/2;  }
    a[j] = v;
}
```

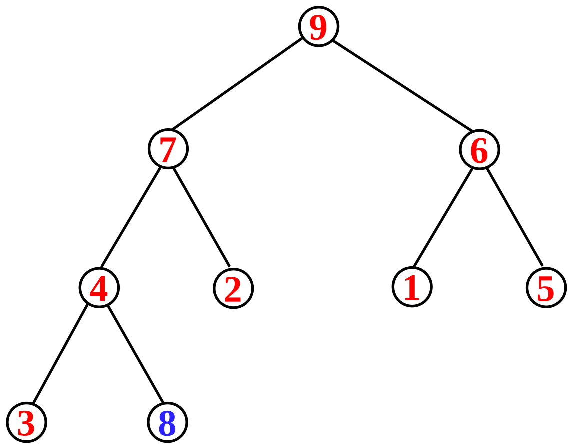
- Precondition : $a[1\ldots j-1]$ is a max heap
- Postcondition : $a[1\ldots j]$ is a max heap

\# A sentinel is put on $a[0]$

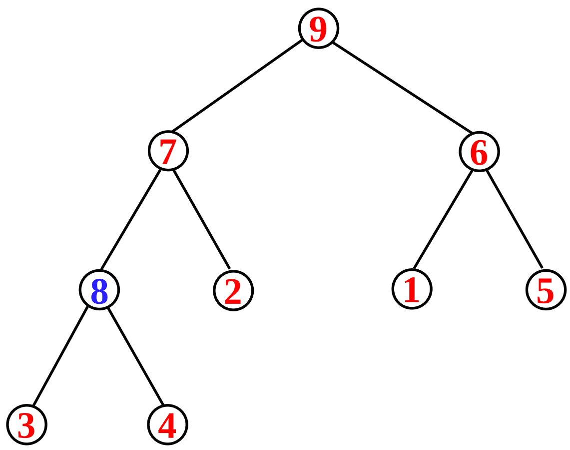
\# comparisons $\le \lg n + 1$

```
insert(v) {
    a[++n] = v;
    upheap(n);
}
```

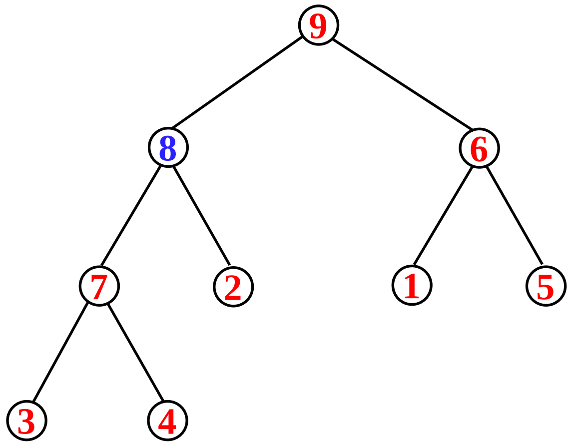
例: insert(8) (例 1/3)



例: insert(8) (例 2/3)

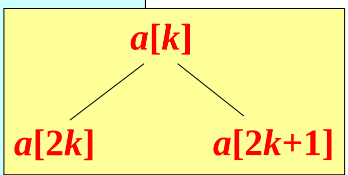


例: insert(8)　(例 3/3)



基本模組： downheap(*k*)

```
downheap(k) {
   int j;    itemtype v;
   v = a[k];
   while (k <= n/2) {
      j = k + k;
      if (j < n && a[j] < a[j+1] ) j++;
      if (v >= a[j]) break;
      a[k] = a[j];  k = j;  }
   a[k] = v;
}
```



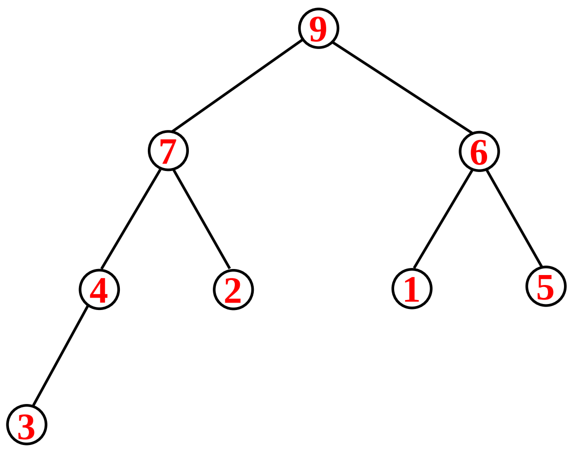
- Precondition : … heaps
- Postcondition : … *a*[*k*] is a max heap

 # comparisons ≤ 2 lg *n*.

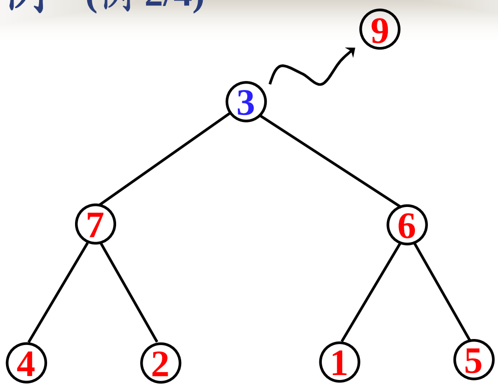
基本操作： remove( )

```
remove( ) {
   itemtype v = a[1];
   a[1] = a[n--];
   downheap(1);
   return v;
}
```

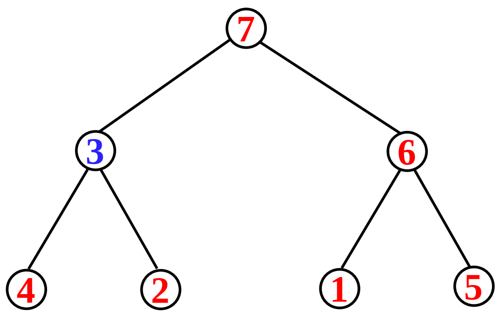

remove( )：例 (例 1/4)



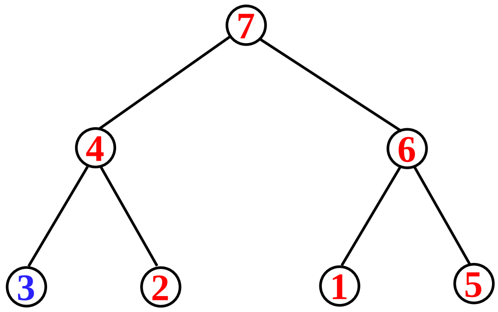
remove( ) : 例 (例 2/4)



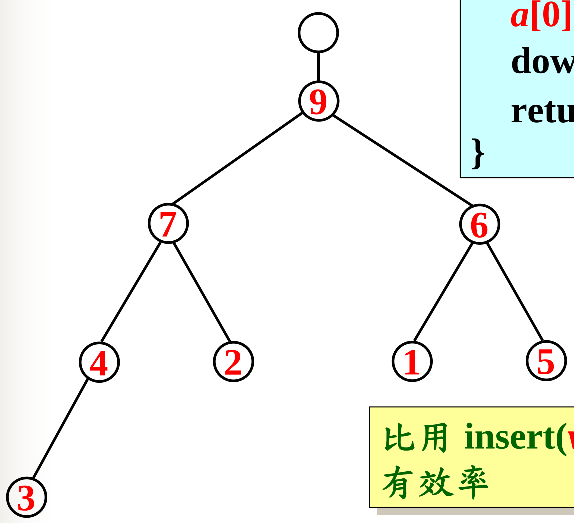
remove( ) : 例 (例 3/4)



remove( ) : 例 (例 4/4)



replace(*v*)




```
replace(v) {
   a[0] = v;
   downheap(0);
   return a[0];
}
```

比用 insert(*v*)+remove( ) 有效率

如何建 heap

- 用 upheap( ) 建 heap (an incremental approach)
  ≃ construct an empty heap + ***n*** insertions
  執行時間 = **Θ(*n*lg*n*)**
- 用 downheap( ) 建 heap (divide-and-conquer)
  執行時間 = **Θ(*n*)**

用 upheap( ) 建 heap 分析

執行時間：

$$\Theta(\lg 2 + \lg 3 + \ldots + \lg \boldsymbol{n})$$

$$= \Theta(\sum_{i=1}^{n} \lg \boldsymbol{i})$$

$$= \Theta(\int_1^n \lg x\, dx) \quad \boxed{\int_1^n \lg x\, dx = n \log n - n + C}$$

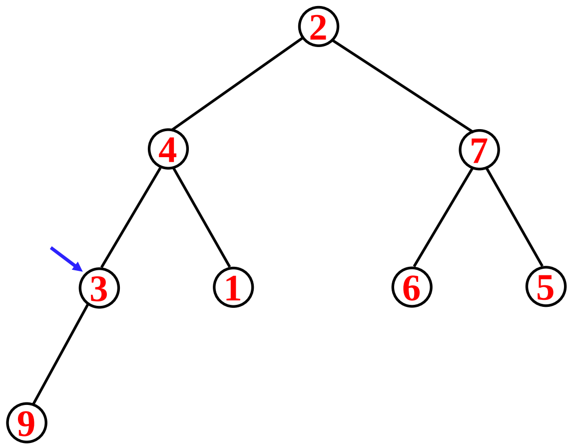
$$= \Theta(\boldsymbol{n} \lg \boldsymbol{n})$$

用 downheap( ) 建 heap

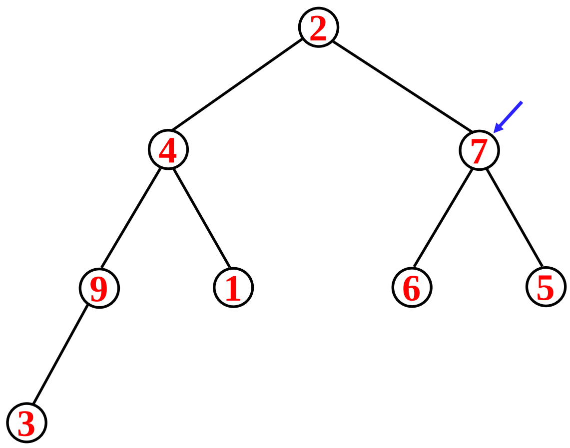
```
construct_heap( ) {
    int k;
    for (k = n/2; k >= 1; k−−)
        downheap(k);
}
```


用 downheap( ) 建 heap (例 1/7)

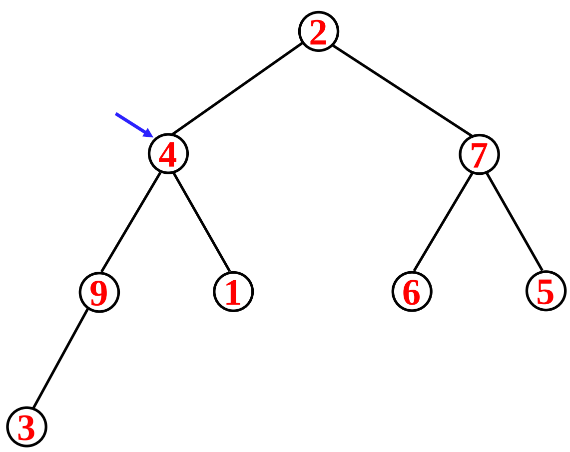
Input: 2 4 7 3 1 6 5 9



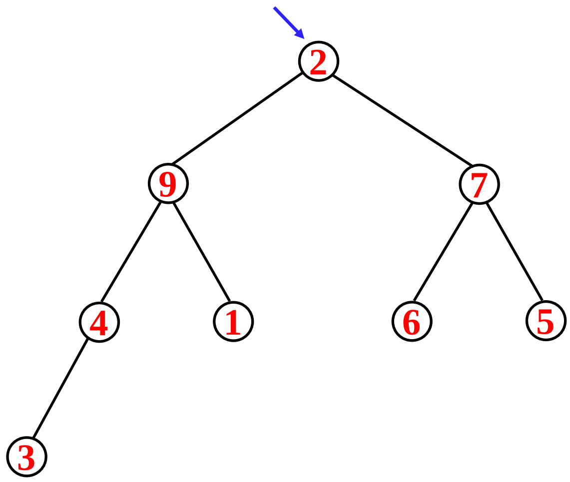
用 downheap( ) 建 heap (例 2/7)



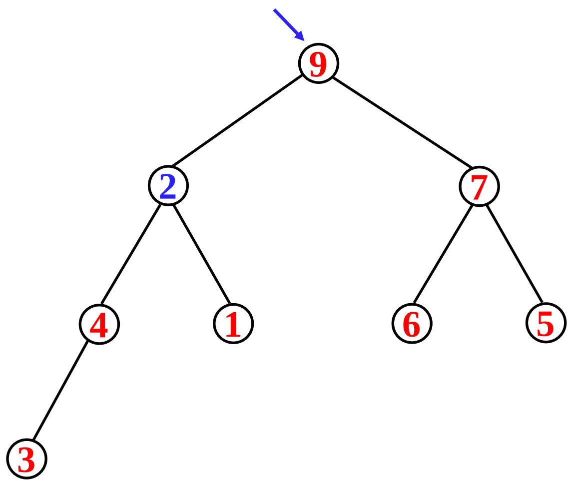
用 downheap( ) 建 heap (例 3/7)



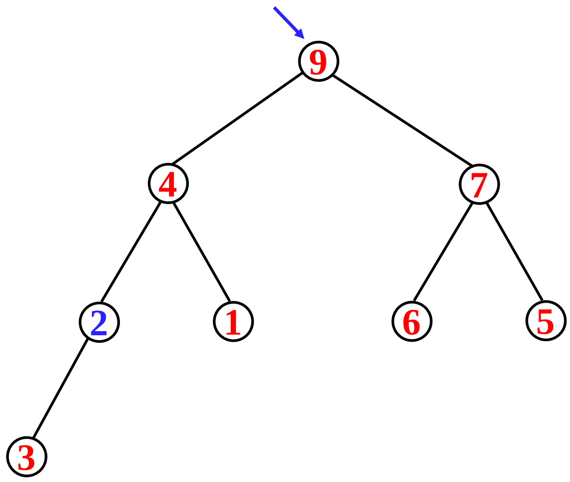
用 downheap( ) 建 heap (例 4/7)



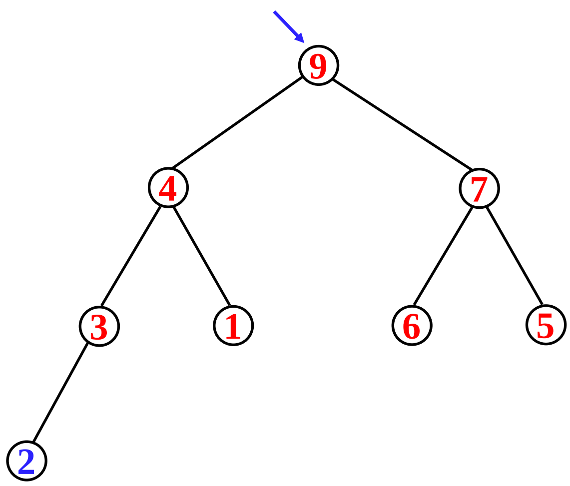
用 downheap( ) 建 heap (例 5/7)



用 downheap( ) 建 heap (例 6/7)



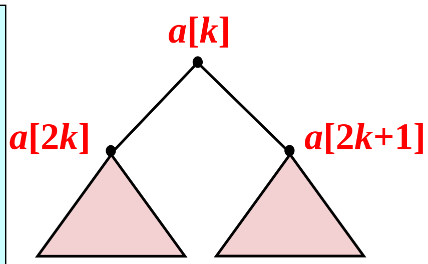
用 downheap( ) 建 heap (例 7/7)



用 downheap( ) 建 heap 分析 (1)

- An equivalent and recursive implementation to construct a heap of size n: Use the following recursive subroutine and then call **F(1).**

```
F(k) {
    if (k > n/2) return;
    F(2k);
    F(2k+1);
    downheap(k);
}
```



用 downheap( ) 建 heap 分析 (2)

- Without loss of generality, assume that the heap is a complete binary tree; i.e. $n = 2^m - 1$ for some integer m and let $T(n)$ denote the number of comparisons needed to construct a heap of size n. According to the recursive implementation, we have:

$$
\begin{aligned}
T(n) &= 2T(n/2) + 2\lg n \quad &\text{for } n > 1\\
&= 0 \quad &\text{for } n = 1
\end{aligned}
$$

- Hence, $T(n) = O(n)$.

Notes

- Heap sort ≅ construct(***n***) + ***n*** removes

  # comparisons = $O(n \lg n)$
- Implement **change( )** & **delete( )**.
- Indirect heaps and indirect priority queues.
- How to implement **join( )** efficiently?
  (See Ch.19 Fibonacci Heaps.)