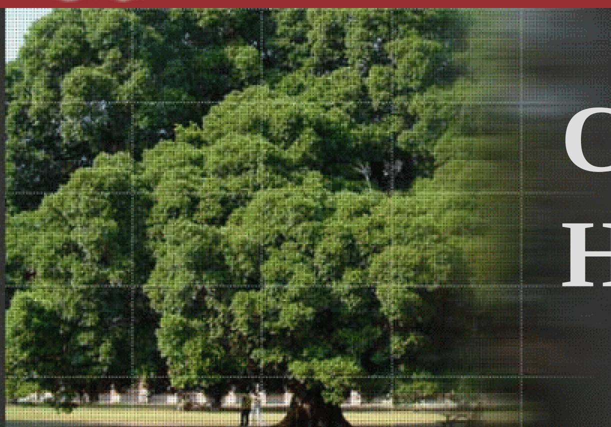
National Cheng Kung University



Chapter 6
Heapsort

Sun-Yuan Hsieh

謝孫源 教授

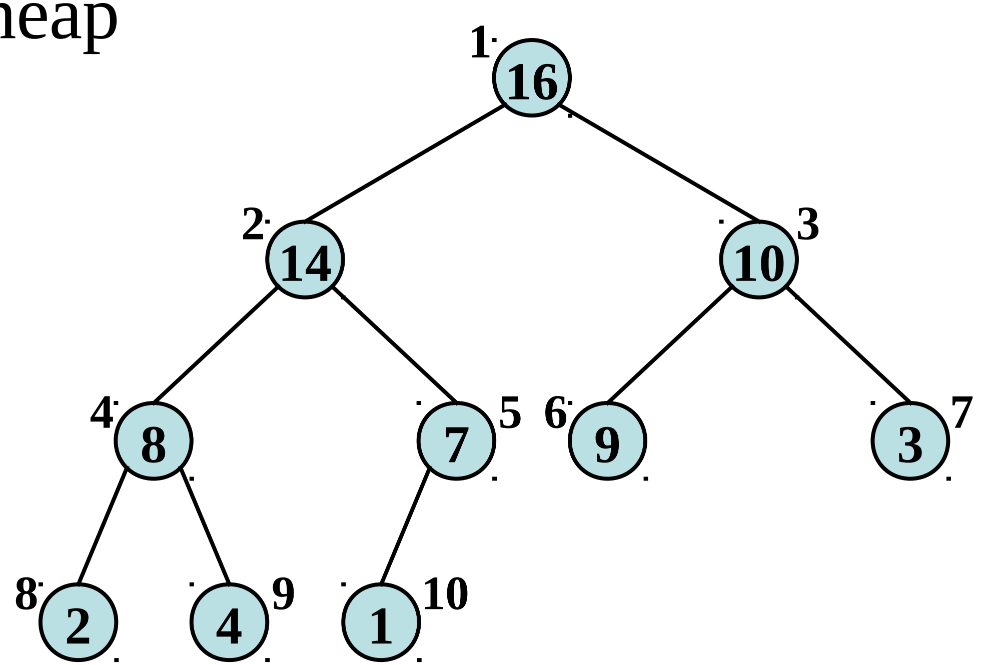
成功大學資訊工程學系

Heap

- Heap A is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\theta(\lg n)$
- A heap can be stores as an array A
  - **Root** of trees is $A[1]$
  - **Parent** of $A[i] = A[\lfloor i/2 \rfloor]$
  - **Left child** of $A[i] = A[2i]$
  - **Right child** of $A[i] = A[2i + 1]$
  - Computing is fast with binary representation implementation

Example

- A max-heap



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap property

- Heap property
  - For max-heap (largest element at root), ***max-heap property*: for** all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
  - For min-heap (smallest element at root), ***min-heap property*:** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

- Maximum element of a max-heap is at the root.

- The heapsort algorithm we'll use max-heaps.

Maintaining the heap property

- MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.
  - Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
  - Assume left and right subtrees of i are max-heaps.
  - After MAX-HEAPIFY, subtree rooted at i is a max-heap.

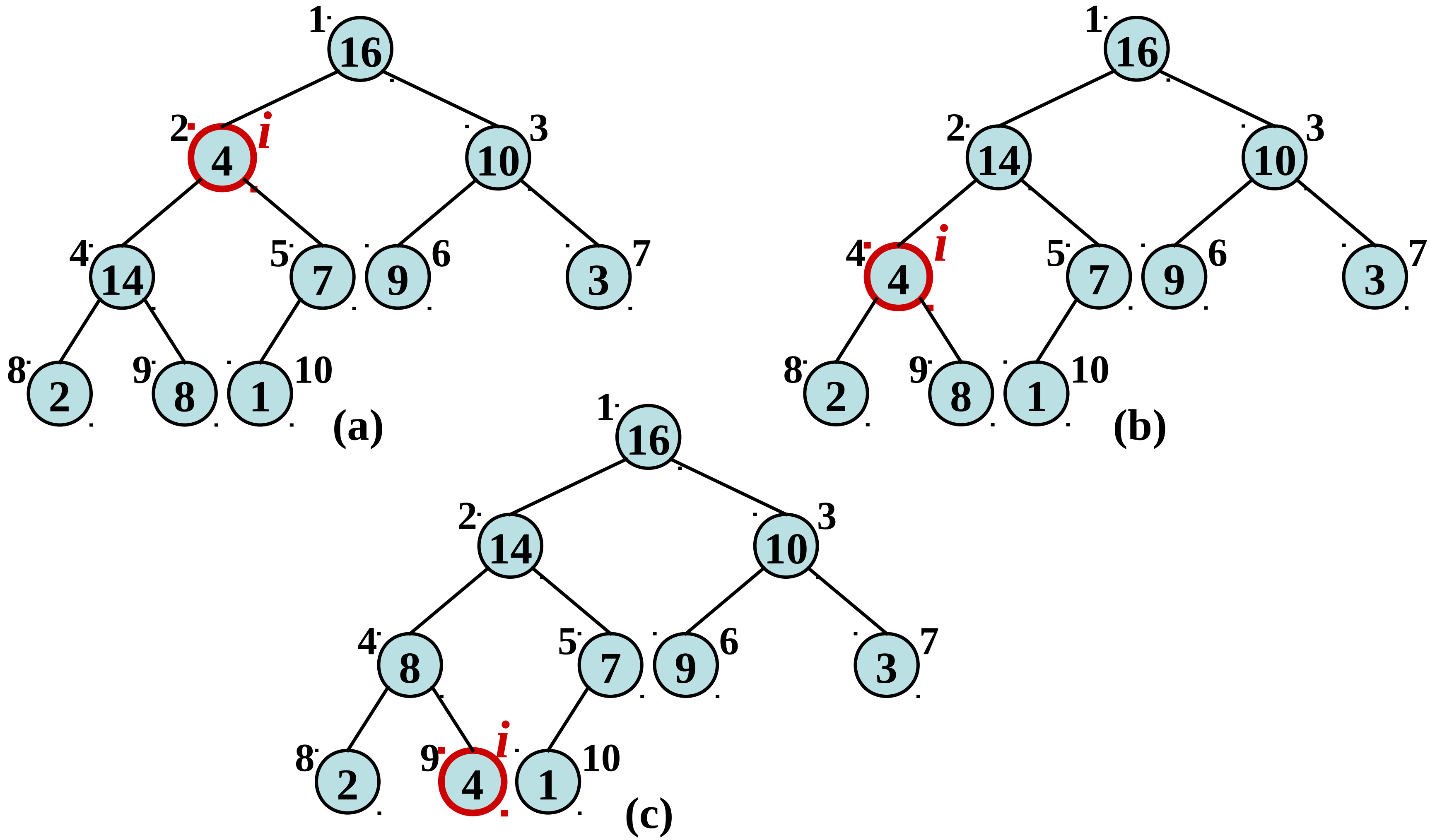
Pseudocode

MAX-HEAPIFY(A, i, n)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ n and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
            MAX-HEAPIFY(A, largest, n)
```

Example

- Run MAX-HEAPIFY on the following heap example.



Analysis of MAX-HEAPIFY

- ***Time:*** $O(\lg n)$

- ***Correctness:*** Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

- The following procedure, given an unordered array, will produce a max-heap.

BUILD-MAX-HEAP(A, n)

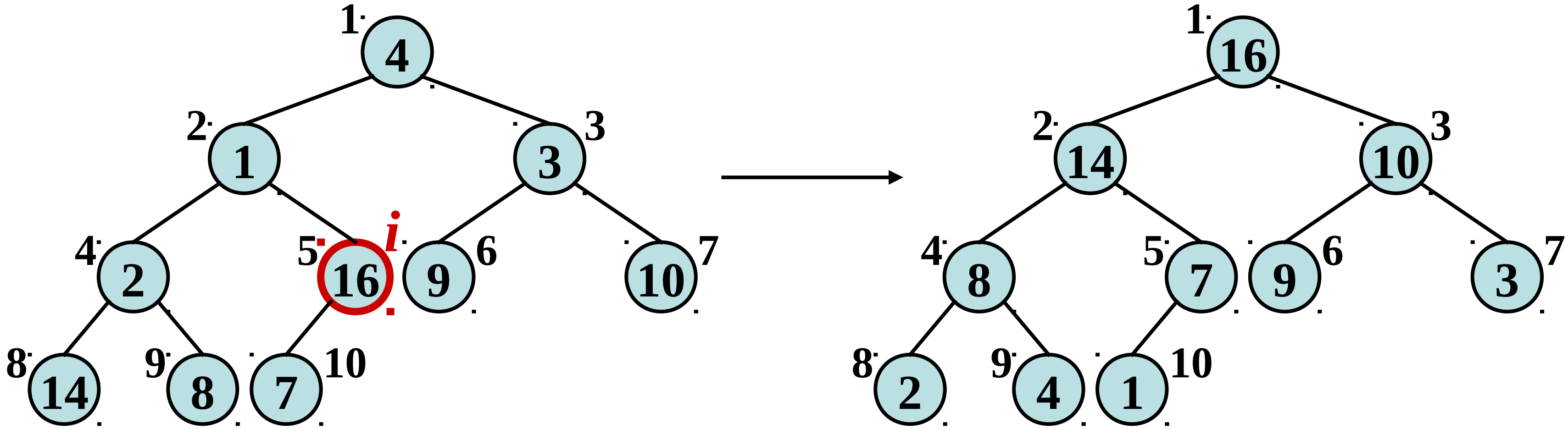
1. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

2. **do** MAX-HEAPIFY(A, i, n)

Example

- Building a max-heap from the following unsorted array results in the first heap example.
  - *i* starts off as 5.
  - MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

	1	2	3	4	5	6	7	8	9	10
A	**4**	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**



Correctness

- **Loop invariant:** At start of every iteration of for loop, each node $i + 1, i + 2, \ldots, n$ is root of a max-heap.
  - **Initialization:** By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the for loop, the invariant is initially true.
  - **Maintenance:** Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i + 1, i + 2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.
  - **Termination:** When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

- Analysis of BUILD-MAX-HEAP
  - ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time → $O(n\lg n)$.
  - ***Tighter analysis:*** Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2).
    - The Time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

    - Evaluate the last summation by substituting $x = ½$ in the formula (A.8)

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

    - Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap

- Building a min-heap from an unordered array can be done by calling **MIN-HEAPIFY** instead of MAX-HEAPIFY, also taking linear time.

The heapsort algorithm

HEAPSORT(A, n)

1. BUILD-MAX-HEAP(A, n)

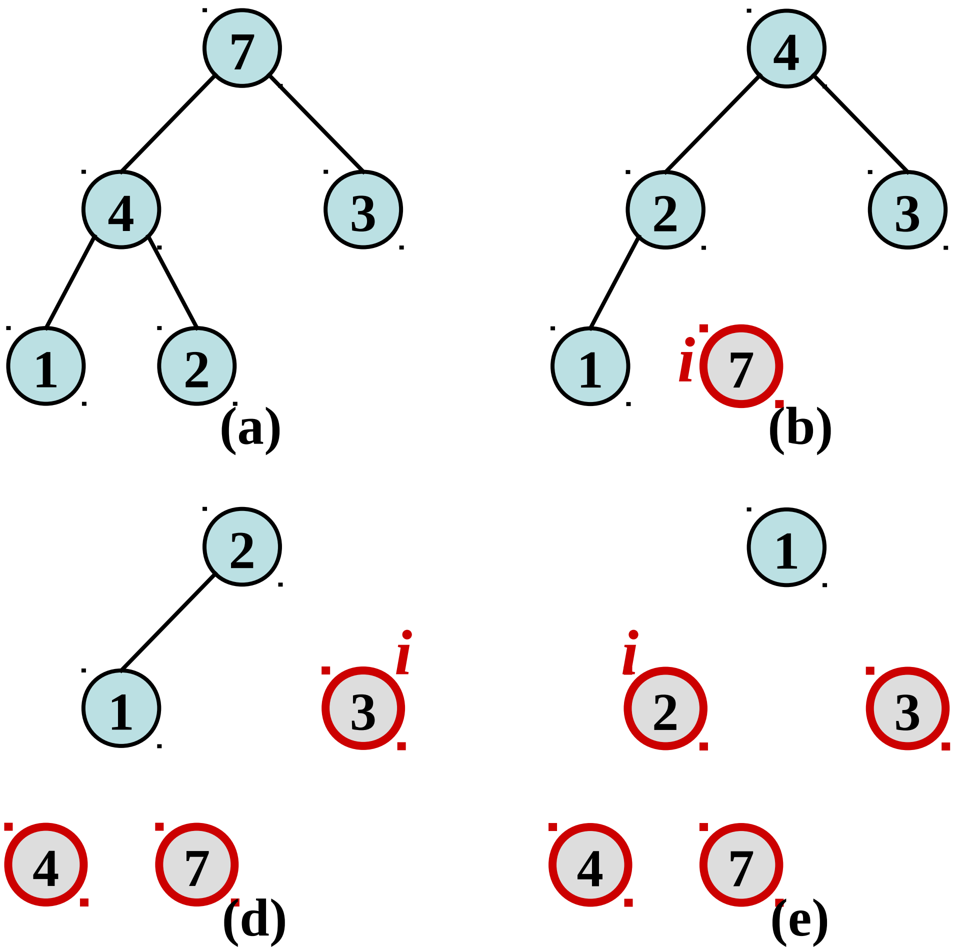
2. **for** $i \leftarrow n$ **downto** 2

3.   **do** exchange $A[1] \leftrightarrow A[i]$

4.   MAX-HEAPIFY(A, 1, $i - 1$)

Example

- The heapsort algorithm



A					
	1	2	3	4	7

Analysis

- Analysis of heapsort
  - BUILD-MAX-HEAP: $O(n)$
  - **for** loop: $n - 1$ times
  - Exchange elements: $O(1)$
  - MAX-HEAPIFY: $O(\lg n)$

  Total time: $O(n \lg n)$

- Though heapsort is a great algorithm, a well-implemented quicksort usually beats it in practice.

Priority queue

- **Heap implementation of priority queue**
  - Max-priority queues are implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.
- **Max Priority Queues**
  - Maintains a dynamic set of S of elements.
  - Each set element has a key: an associated value.
  - Max-priority queue supports dynamic-set operations:
    - INSERT(S, x): inserts element x into set S.
    - MAXIMUM(S): returns elements of S with largest key.
    - EXTRACT-MAX(S): removes and returns element of S with largest key.
    - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.

- **Min-priority queue supports similar operation:**
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.

- **Finding the maximum element**
  - Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)

1. **return** $A[1]$

- *Time:* $\Theta(1)$

- **Extracting max element**

Given the array A:

- Make sure heap is not empty
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)

```
if n < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[n]
MAX-HEAPIFY(A, 1, n – 1)        ► remakes heap
return max
```

- ***Analysis:*** constant time assignments plus time for MAX-HEAPIFY.
- ***Time:*** $O(\lg n)$

Increasing key value

- Given set *S*, element *x* and new key value *k*:
  - Make sure $k \geq x$'s current key.
  - Update *x*'s key value to *k*.
  - Traverse the tree upward comparing *x* to its parent and swapping keys if necessary, until *x*'s key in smaller than parent's key.

HEAP-INCREASE-KEY(*A*, *i*, *key*)

if *key* < *A*[*i*]
    then error “new key is smaller than current key”
A[*i*] ← *key*
while *i* > 1 and *A*[PARENT(*i*)] < *A*[*i*]
do exchange *A*[*i*] ↔ *A*[PARENT(*i*)]
        i ← PARENT(*i*)

- ***Analysis:*** Upward path from node *i* has length *O*(lg *n*) in an *n*-element heap.
- ***Time:*** *O*(lg *n*)

Inserting into the heap

- Given a key *k* to insert into the heap:
  - Insert a new node in the key last position in the tree with key $-\infty$
  - Increase the $-\infty$ key to *k* using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)

1. $A[n + 1] \leftarrow -\infty$
2. HEAP-INCREASE-KEY(A, $n + 1$, key)

- ***Analysis:*** constant time assignments plus time for HEAP-INCREASE-KEY
- ***Time:*** $O(\lg n)$

- **Min-priority queue** operations are implemented similarly with min-heaps.