V.D. Graph ADT : Heaps

- Heaps are used to store values on graphs / networks (e.g., graph's vertices / edges) but not to store graphs themselves
- Heaps are designed to optimize fast searches for entries in the heap w/ min / max value (e.g., max edge weight)
- Example Usages:
  1. Shortest path on weighted graph (edge weight is a distance)
  2. Disease spreading:
     - Node weight - current estimate of the time at which the vertex will be infected by the disease
     - Node - person
     - Initially $w_i = \infty$ for $\forall\, v \in V$ except for $w_0 = 0$, with v_0 being the carrier of the disease
     - Disease spread / propagation is a simulation that (a) selects the vertex v_{min} with the smallest time of infection w_{min};
       (b) infects v_{min}
       (c) updates w_i's, especially the neighbors of v_{min}, $N(v_{min})$

Heaps (cont.)

Key requirements in disease propagation example:
- to quickly find the smallest value of the infection time on the graph
- to quickly update the values of the nearest neighbors (NN) of the infected v_{min}

Binary Heap: Graph ADT that meets these requirements

Binary Heap = POT + Index

① Binary tree (labeled binary tree)

② Node: a) Node ID (vertex of the graph): v_i
b) Value/label (vertex weight): $w(v_i)$ (numerical)

v_i
$w(v_i)$

In C:

```
struct NODE {
    int   nodeID;
    float nodeWeight;   <- /* could be int */
};
```

E.g.

17
3.5

③ Partially Ordered Tree (POT):

$\forall v \in V$: label(v) ≤ label(u) for $\forall u \in$ Children(v)

* Labels along any branch of the tree are ordered in non-decreasing manner.
* The root of any subtree is the smallest element of that subtree.
* The same labels can appear at different levels of the tree!
* The labels at the same level (across branches) are not ordered.

Heaps (cont.)

④ All levels are full, except for possibly last.

⑤ Nodes in last level are placed as far left as possible.

⑥ Index

tells the location of each item in the tree, using the 2D-array: (Row, Position)

- Row: in the binary tree (top to bottom)
- Position: within the level (left to right)

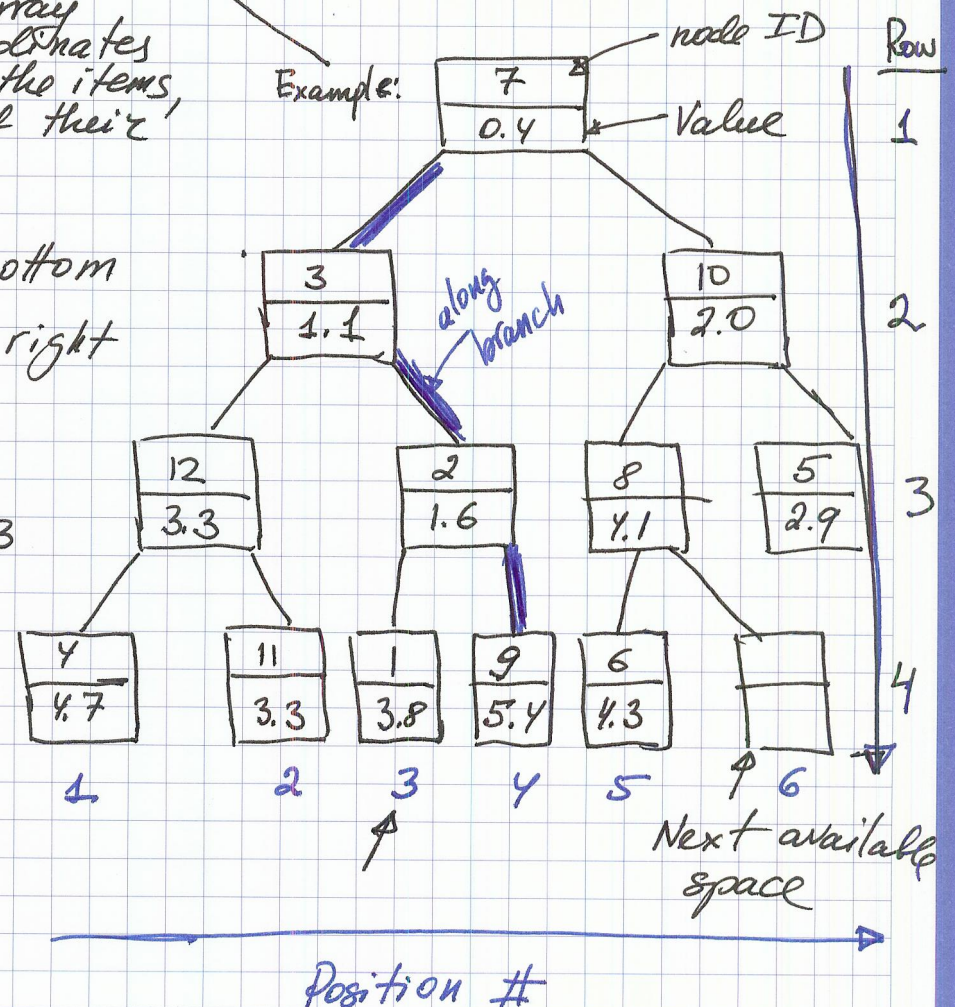
node ID	Row	Position
1	4	3
2	3	2
3	2	1
4	4	1
⋮	⋮	⋮
7	4	1

The index is an array containing the coordinates in the tree of all the items, listed in order of their node ID's:

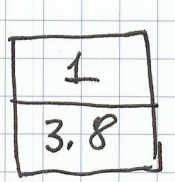
Row #'s: top to bottom

Position #'s: left to right

Example:



Node:



Row = 4

Position = 3

Values along the branch:

0.4; 1.1; 1.6; 5.4

→ non-decreasing

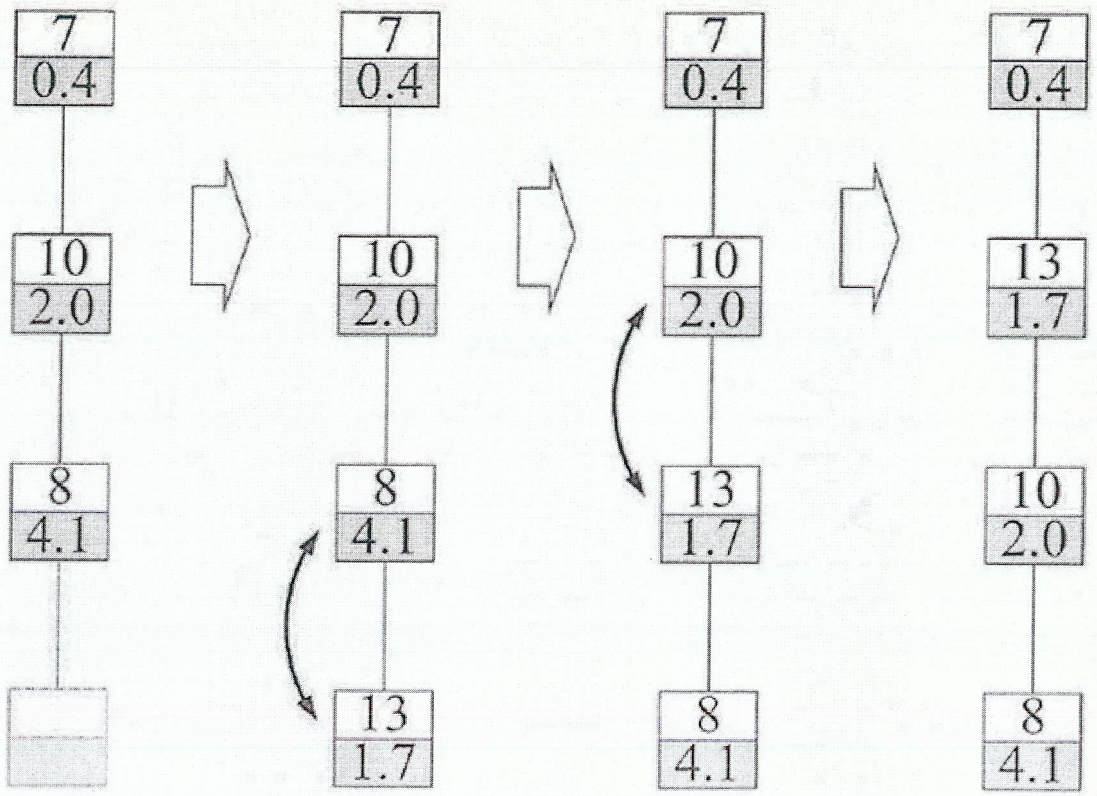
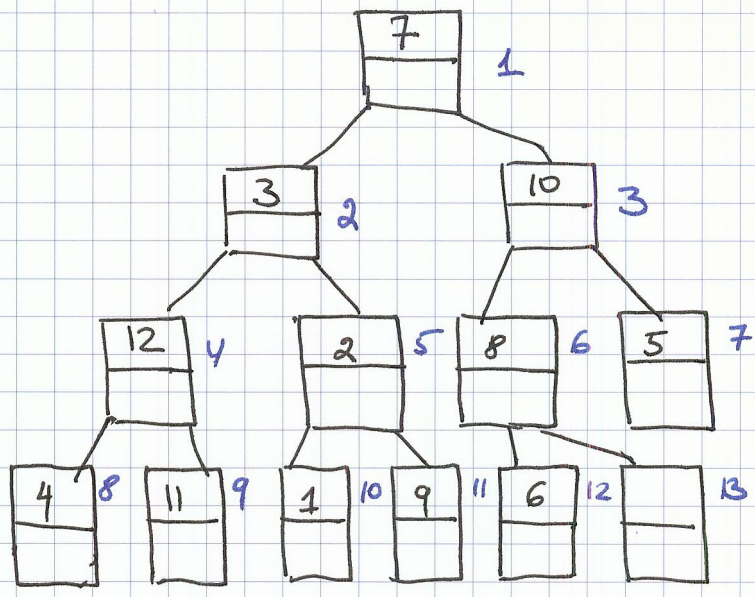


Figure 9.4: Sifting a value up the heap. A branch in the tree initially contains three items as shown. A new item with value 1.7 is added at the bottom. The upward sift repeatedly compares and swaps this value with the value above it until it reaches its correct place in the partial ordering. In this case the added value 1.7 gets swapped twice, ending up (correctly) between the values 0.4 and 2.0.

5.e3+

Heaps (cont.)

Index Implementation as a 1D-array



in-order traversal

parent(A[9]) = A[⌊9/2⌋] = A[4] = 12

A

7	3	10	12	2	8	5	4	11	1	9	6		
1	2	3	4	5	6	7	8	9	10	11	12	13	

Row=1 Pos=1 Row=2 Pos=1 Row=2 Pos=2 Row=3 Pos=1

A[1] - root = 7 - node ID

left child of A[1] is A[2] = 3 lchild(A[1]) = A[2]

right child of A[1] is A[3] = 10 rchild(A[1]) = A[3]

Properties:

$$\text{lchild}(A[i]) = A[2i]$$

$$\text{rchild}(A[i]) = A[2i+1]$$

$$\text{parent}(A[i]) = A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$$

parent(A[11]) = $A[\lfloor 11/2 \rfloor] = A[5]$

Heaps (cont.)

Operations on Heaps:

I. Build a heap
II. Add an item
III. Update a value in the heap item
IV. Find and remove the smallest value item

II. Add / insert an item

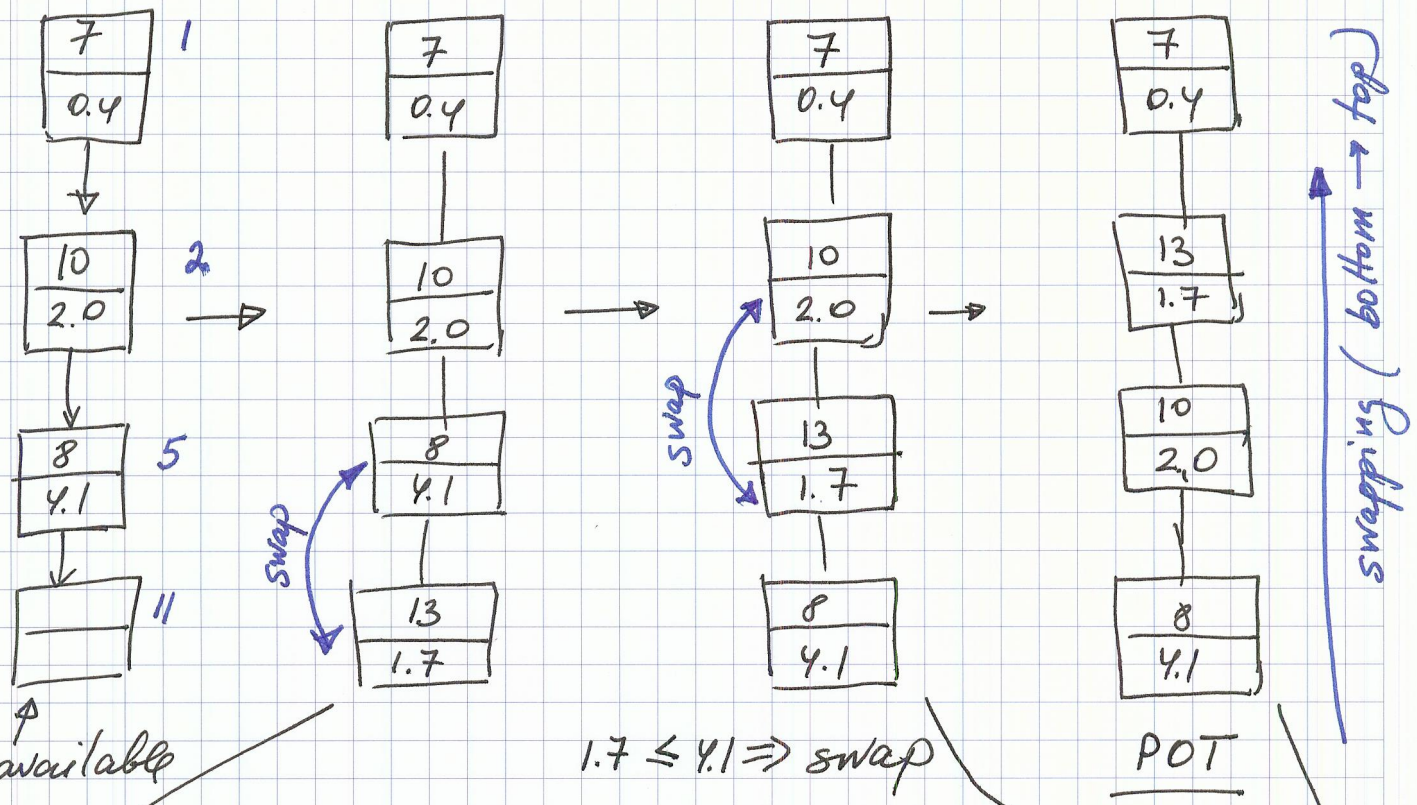
Let n be the current # of elements in the array implementation of the heap; A[1...n] satisfy POT property

void insert (int A[], int x)

1. Increment n
2. Assign A[n] = x ← the new node ID to be added
3. Heapify / Buble up / Sift up A
   - to make sure that the new A satisfies POT after adding x
   - by comparing value(x) w/ the value of parent(x)
   - and swaping x & parent(x) if out-of-order
   - continue till not in violation or untill reach the root of the tree

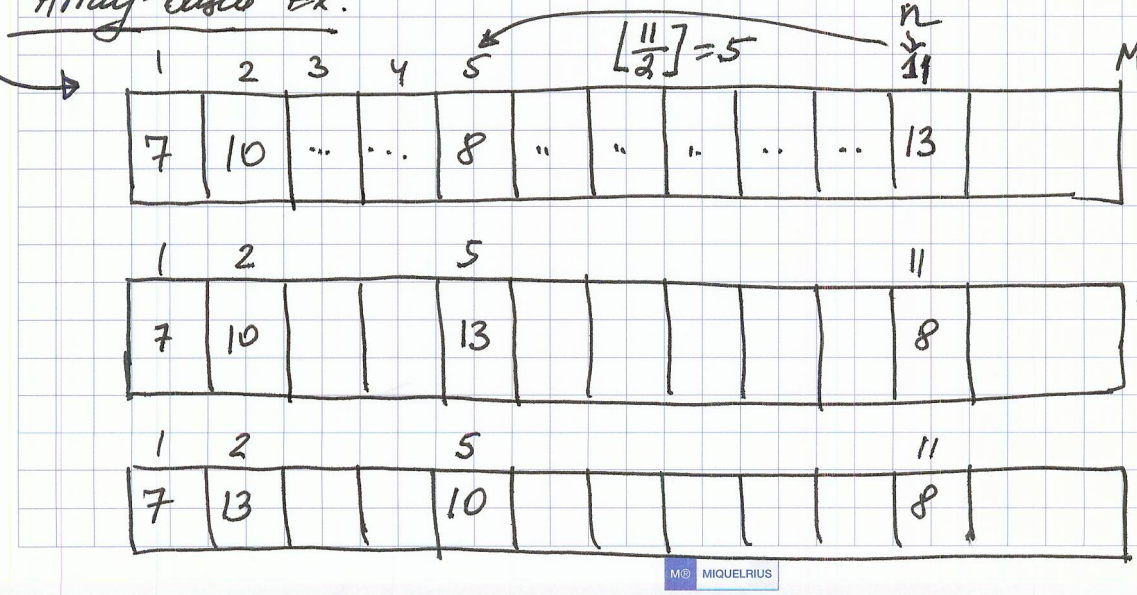
Heaps (cont.)

Consider the branch from root to the leftmost available item in the tree for the item to insert: Node ID = 13, value(13) = 1.7



Heapify / Bubble Up / Sift Up

Array-based Ex:



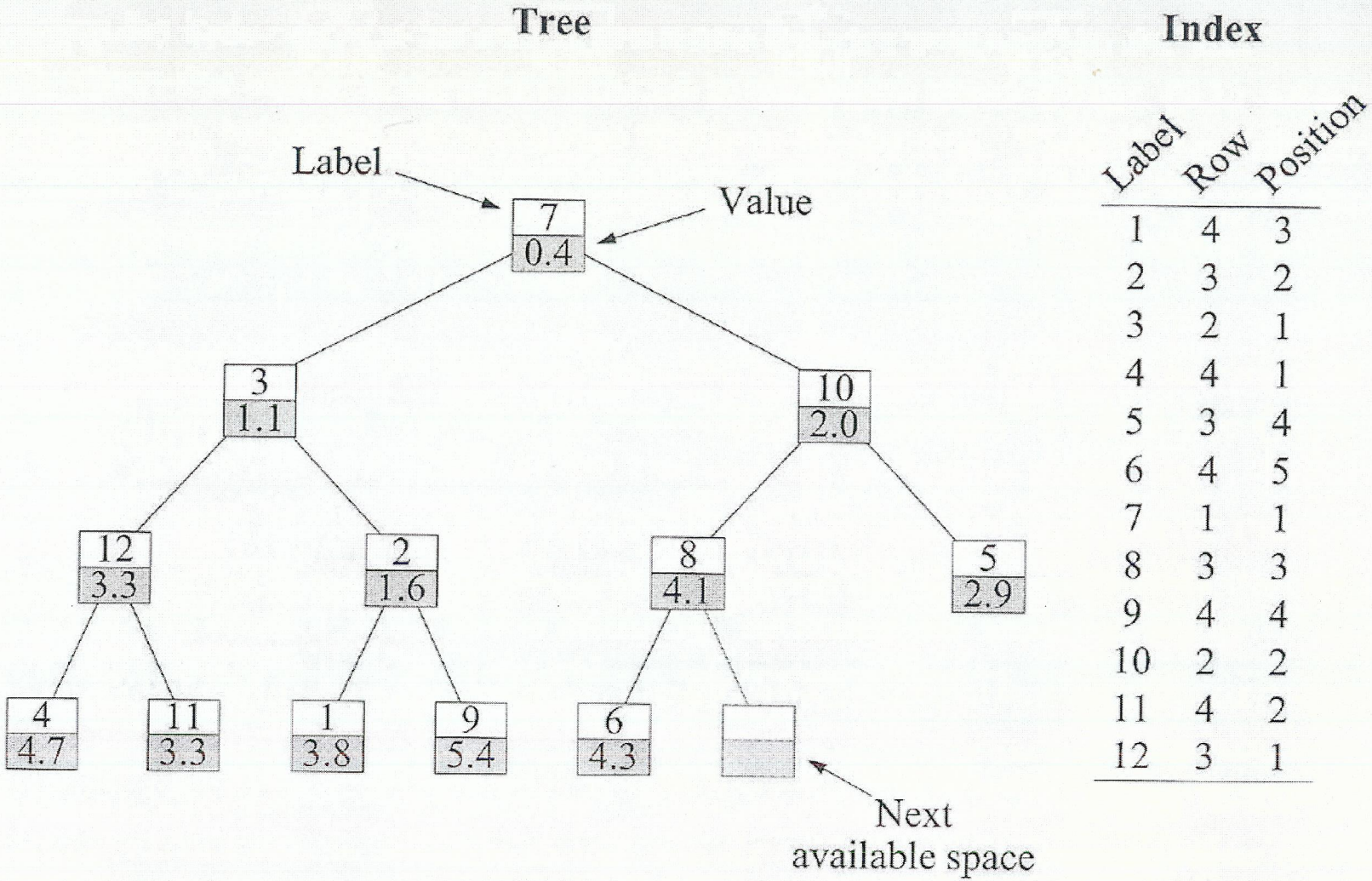


Figure 9.3: The structure of a binary heap. A binary heap consists of two parts, a tree and an index. The nodes of the tree each contain a label and a numerical value, and the tree is partially ordered so that the numerical value at each node is greater than or equal to the value above it in the tree and less than or equal to both of the values below it. The index is a separate array that lists by label the positions of each of the items in the tree, so that we can find a given item quickly. New items are added to the tree in the next available space at the bottom, starting a new row if necessary.

5.e.6+

Heap (cont.)

Full example in C:

```
void swap (int A[], int i, int j) {
    int tmp;
    tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
}

void bubbleUp (int A[], int i)
{
    if (i>1 && A[i] < A[i/2]) {
        swap (A, i, i/2);
        bubbleUp (A, i/2);
    }
}

void insert (int A[], int x, int *pn)
{
    (*pn)++;    // increment (assume n=*pn < MAX)
    A[*pn] = x;
    bubbleUp (A, *pn);
}
```

pointer to the current element in A[]

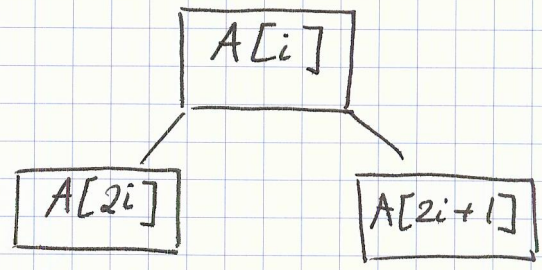
Heaps (cont.): Find & remove the smallest value item (at the root)

IV. void bubbleDown (int A[], int i, int n)

```
{
    int child;

    child = 2*i;
    // select a child with
    // which to swap A[i]
    // if POT is in violation

    if (child < n && A[child+1] < A[child])
        ++child;
    if ( child <= n && A[i] > A[child] ) {
        swap (A, i, child);
        bubbleDown (A, child, n);
    }
}
```




```
void deletemin (int A[], int *pn)
{
    swap (A, 1, *pn);   // swap root with A[n]
                        // root is the min
    --(*pn);            // decrement n by 1
    bubbleDown (A, 1, *pn); // new root after
}                           // swapping may violate
                            // POT property =>
```

push the offending element down till POT is OK or till reaches the leaf

VII Storing Graphs on Disks: File Formats

Notes: 1. Do NOT store as plain ASCI text; store as binary files to save space by a lot.

The most common file format is DIMACS

n ← # of vertices
m ← # of edges ← This way, no need for second pass to count the # of vertices & edges

v_1
v_2
⋮
v_n
} vertex/node ID ← Sometimes omitted, if $v_i = i$

v_{i_1} v_{j_1} w_1
v_{i_2} v_{j_2} w_2
⋮
v_{i_m} v_{j_m} w_m

← If w_i's are missing, then $G(V, E)$ is unweighted graph

v_{i_m} v_{j_m}: edges as vertex pairs
w_m: edge weights (optional)