Data Structures and Algorithms (DSA) Course 12

Heap tree and sorting techniques

Iulian Năstac

Special binary trees

(Recapitulation)

P: A binary tree with the height **i** could have a maximum number of $2^{i+1}-1$ nodes.

Notes:

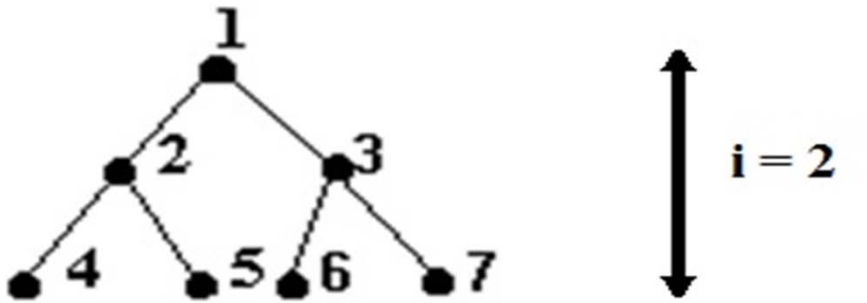
A level with the depth **k** could have maximum 2^k nodes

The full binary tree

(Recapitulation)

A full binary tree is the one that has the maximum number of vertices ($2^{i+1}-1$) for a specified height of **i**.

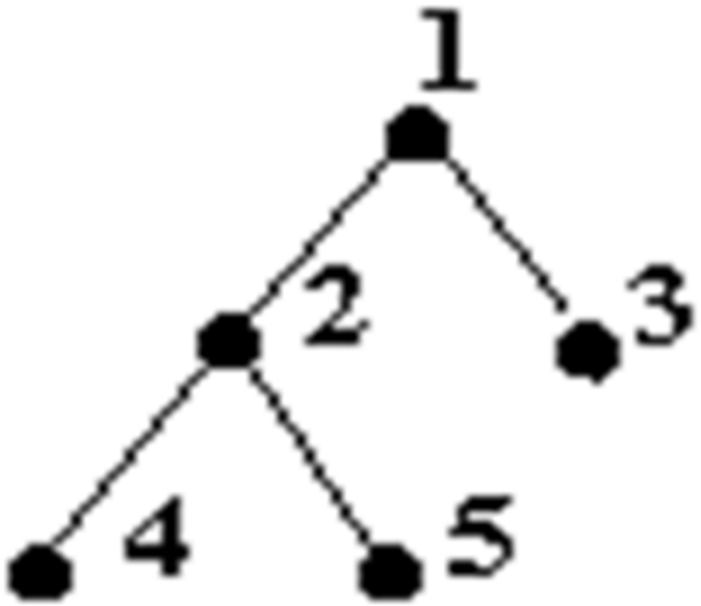
For example, a full binary tree of height 2 is as follows :



A full binary tree is that one in which every node has two children (excepting the last level with the leaves).

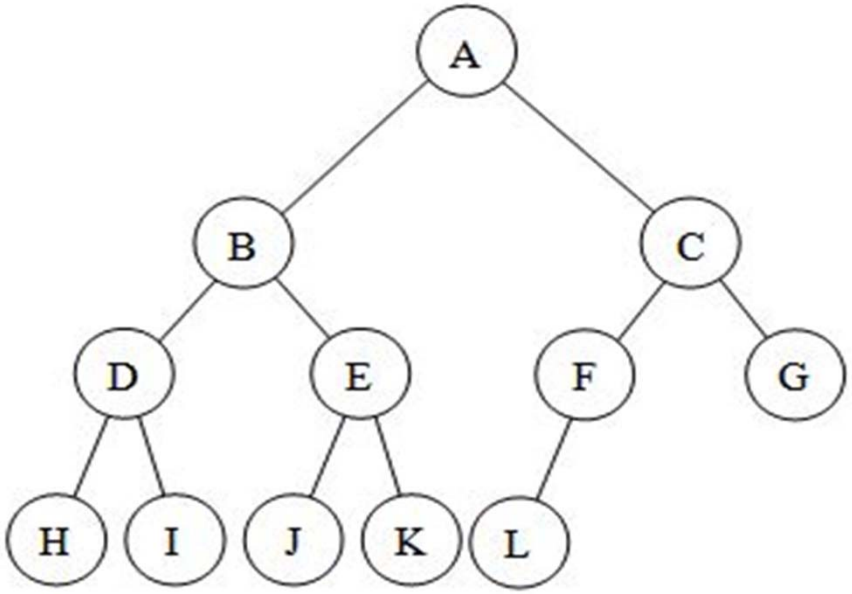
Complete binary tree
(Recapitulation)

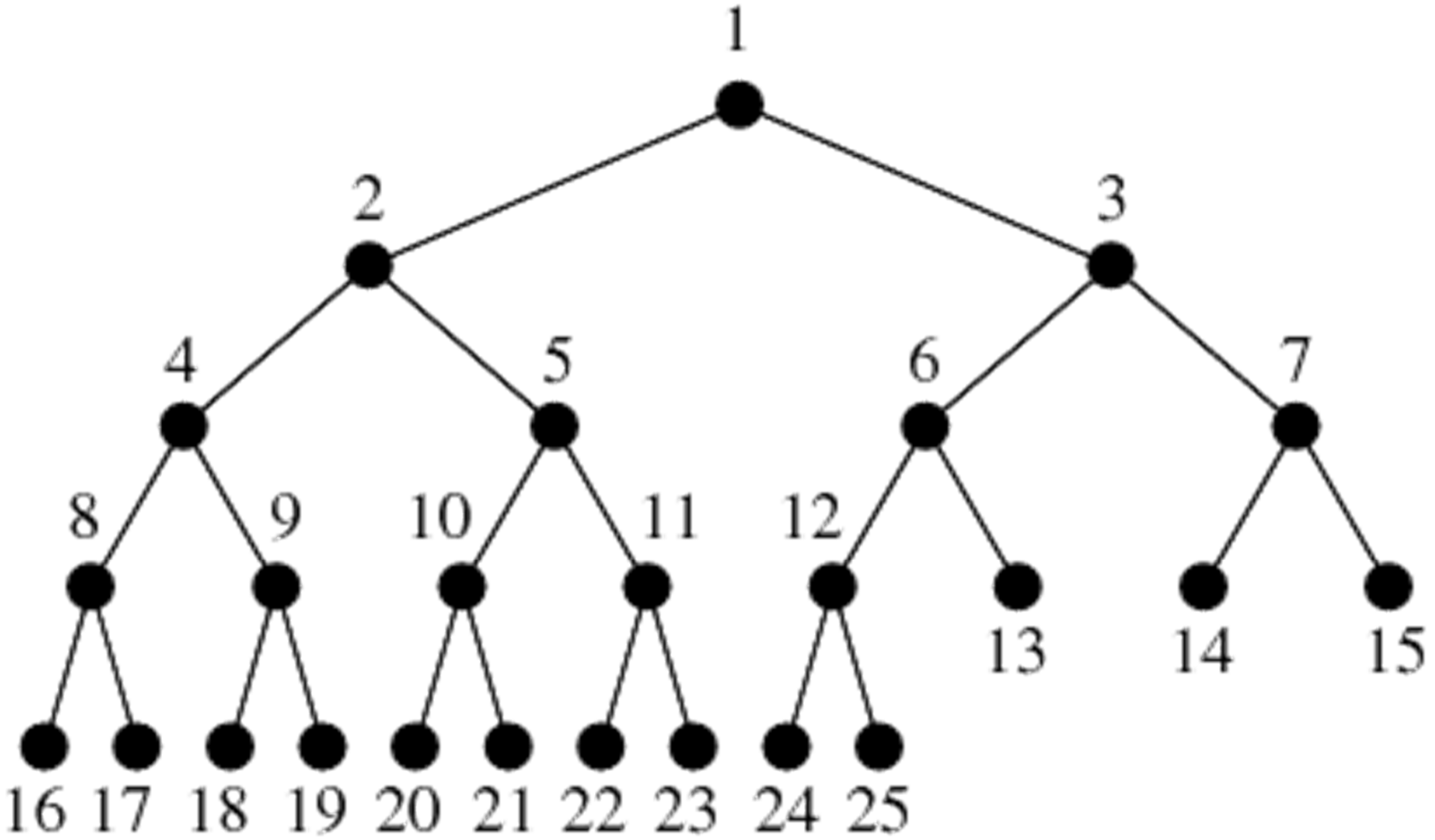
A binary tree with ***n*** nodes that has a height of ***i*** is called as being a **complete binary tree** if it is obtained from a full binary tree with a height of ***i***, in which there are eliminated the last consecutive nodes, numbered with ***n+1***, ***n+2***, ... up to $2^{i+1}-1$.



Notes:

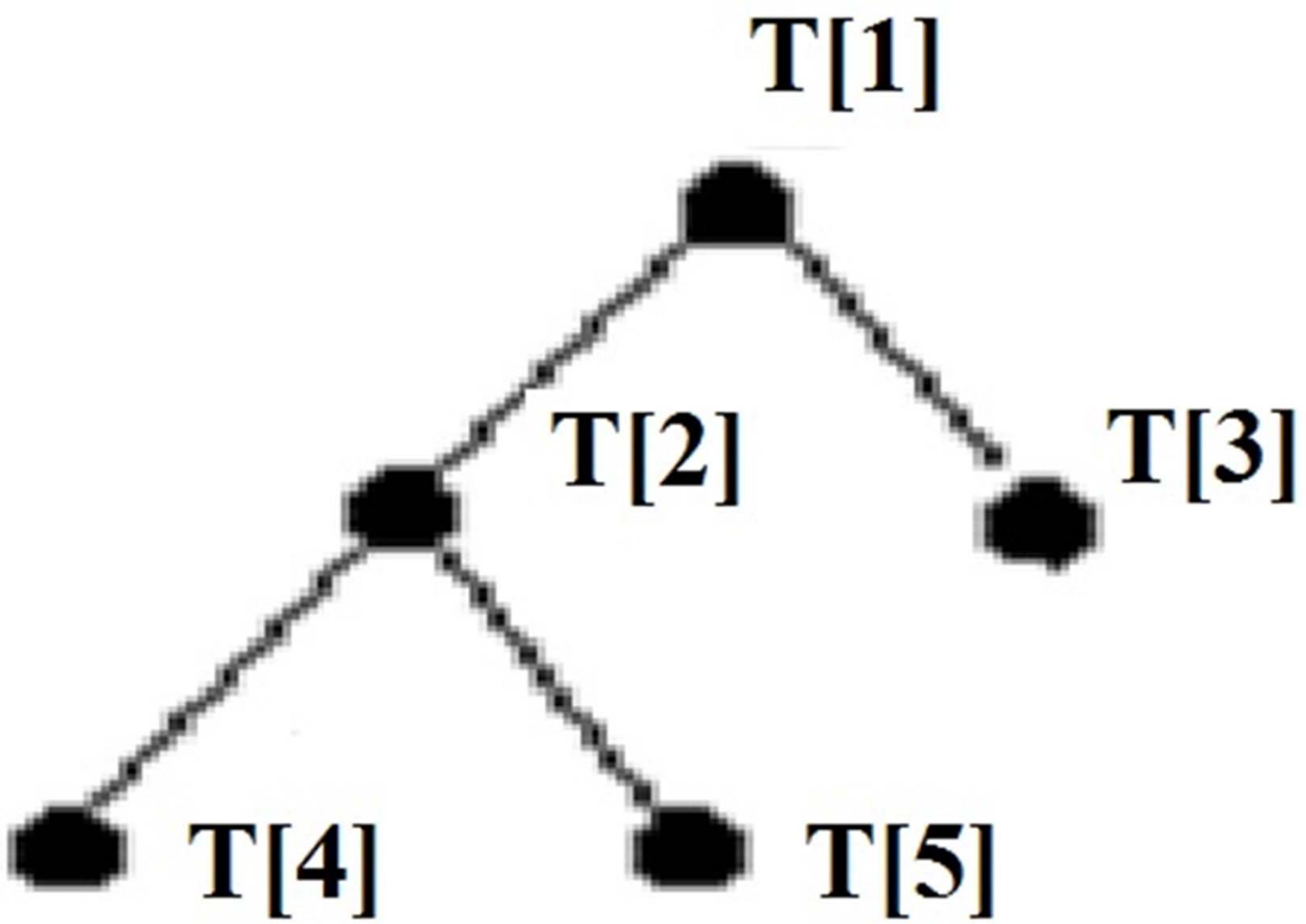
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

Notes:

- A complete binary tree can be sequentially represented using a vector (noted **T**), in which the nodes of depth **k**, from left to right, are inserted in the following positions: **$T[2^k]$**, **$T[2^k+1]$**, …, **$T[2^{k+1}-1]$**, excepting the final level, which may be incomplete.



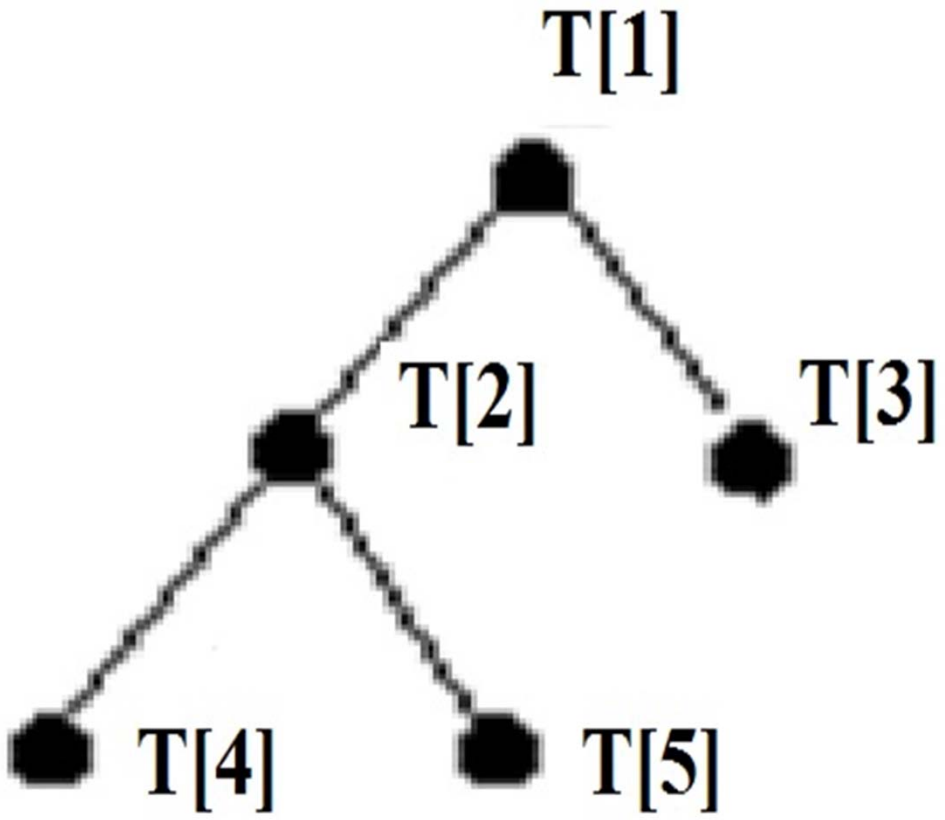
• **Warning**: This is a generic vector, which it begins with T [1] (not with T [0], as usual in the C programing).

• We can make the necessary changes when we will write the code in C.

Notes:

- The parent of a node from **T[i]**, i>1, can be found in **T[i div 2]** .

- The sons of a node from **T[i]**, can be found (if exist) in **T[2·i]** and **T[2·i + 1]** .



The height of a complete binary tree (Recapitulation)

- We demonstrated in previous course that the height of a complete binary tree with n vertices is:

$$i = \lfloor \log_2 n \rfloor$$

The heap tree

(Recapitulation)

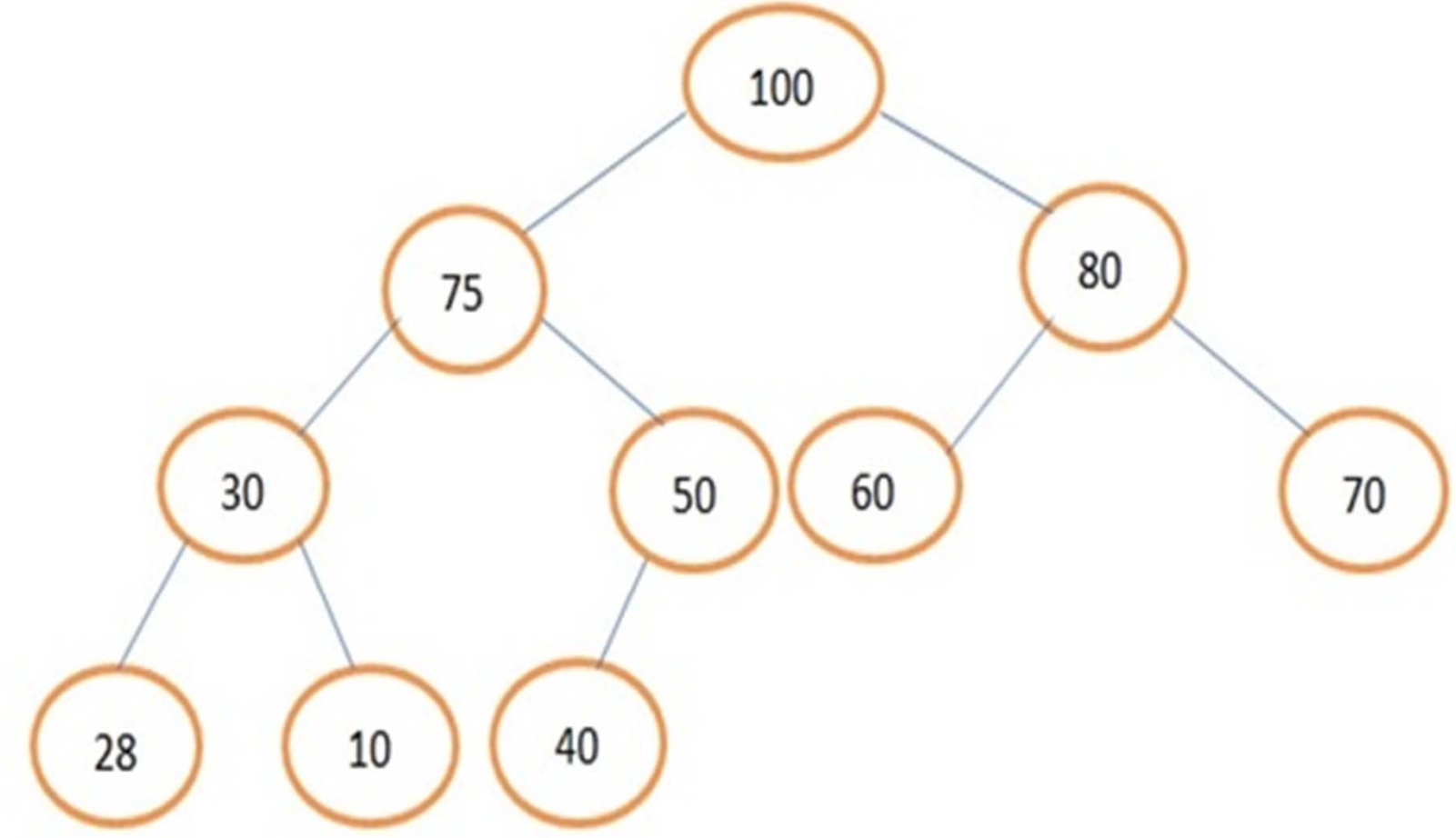
A **heap** is a specialized tree-based data structure that satisfies the **heap property**:

If *A* is a parent node of *B* then the key of node *A* is ordered with respect to the key of node *B*. The same ordering is applied across the entire heap.

The heap is not a classic binary tree!

There is no order between left and right son of a father inside of a heap…

Example of a heap tree



100	75	80	30	50	60	70	28	10	40
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Heap tree types

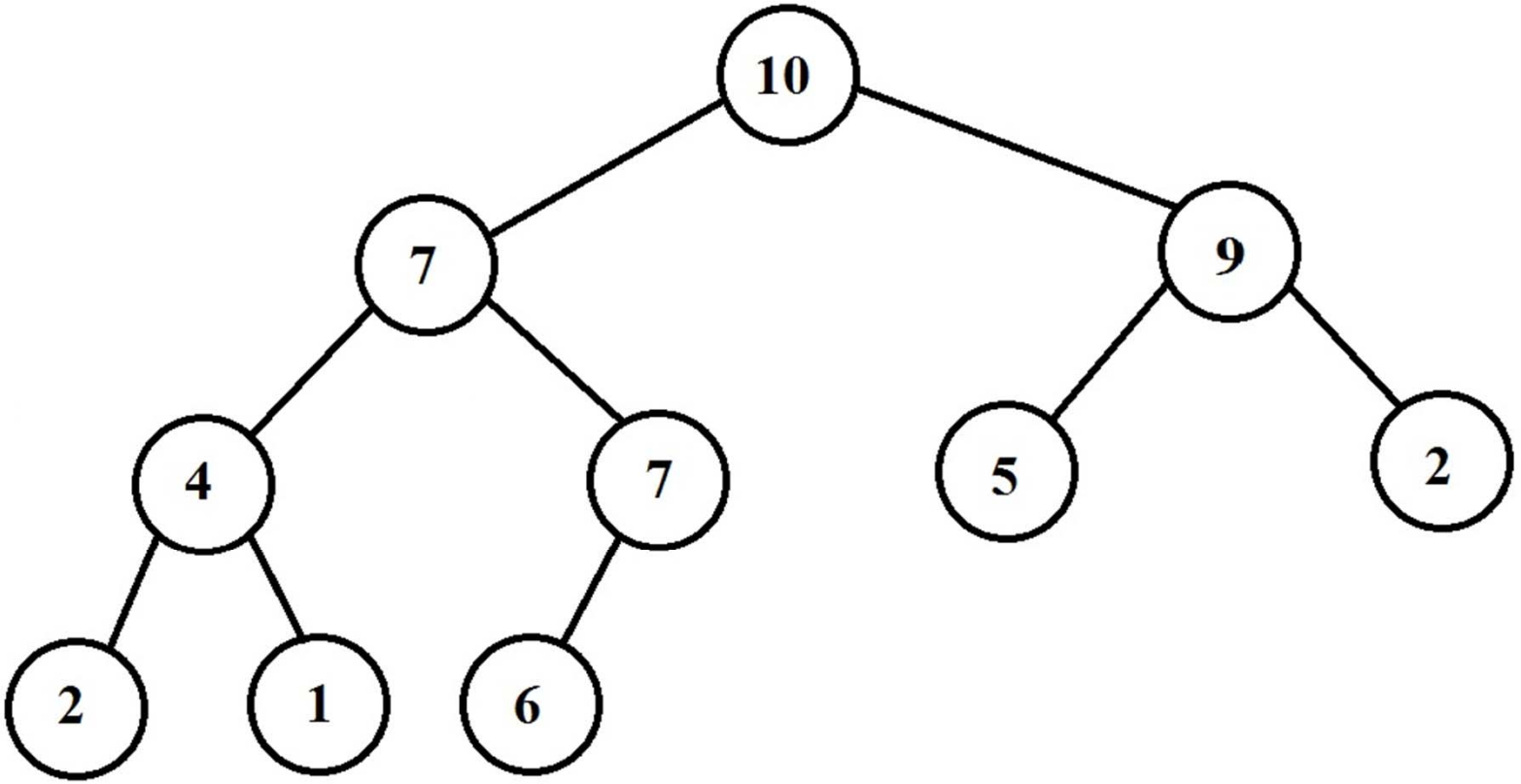
- Heaps can be classified further as either a "**max heap**" or a "**min heap**".
- In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node.
- In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

Notes:

- Usually, in many applications, a **max heap** is simply called **heap tree**

- Any **heap tree** can be represented by a vector (one-dimensional array)

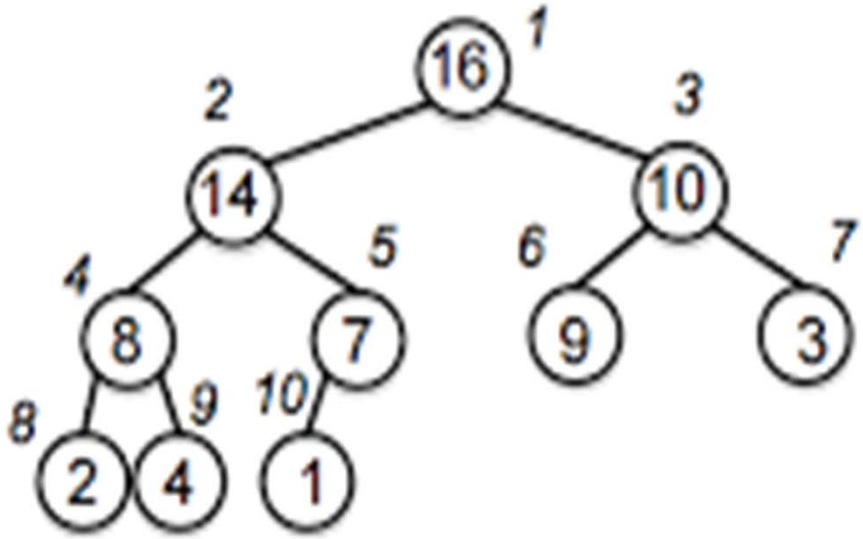
Example:



This heap can be represented by the following vector:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Other example:



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

This heap can be represented by the following vector:

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

Notes:

- In a heap tree we can make modifications at the node level (changing the value of the current node).
- Thus the value of a node can be increased or decreased, resulting a canceling of the specific order inside the heap tree.
- The order of the heap can be simply restored through two operations called ***sift-down*** and ***sift-up***.

sift-up (percolate) in a heap

sift-up = means to move a node up in the tree, as long as needed; used to restore heap condition after insertion.

- Called "sift" because node moves up the tree until it reaches the correct level, as in a sieve. Often incorrectly called "shift-up".

- It is also said that the changed value was filtered (percolated) to his new position.

sift-down in a heap

***sift-down* = moves a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement**.

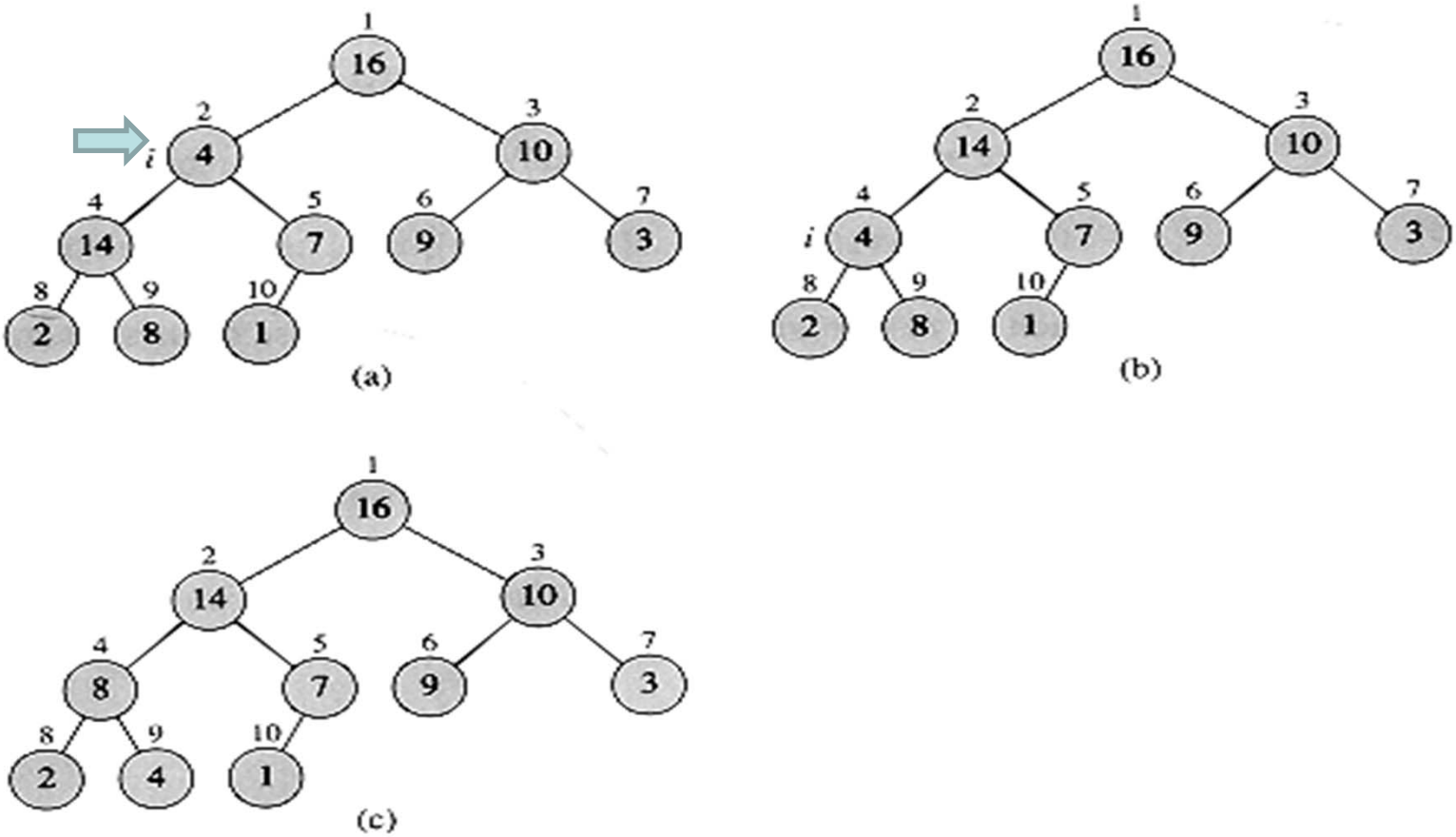
- If a node value decreases so that it becomes lower than the elder son, it is enough to change between them these two values, and continue the process (downward) until the heap property is restored.
- It is said that the changed value was sieved (sift down) to his new position.

Note:

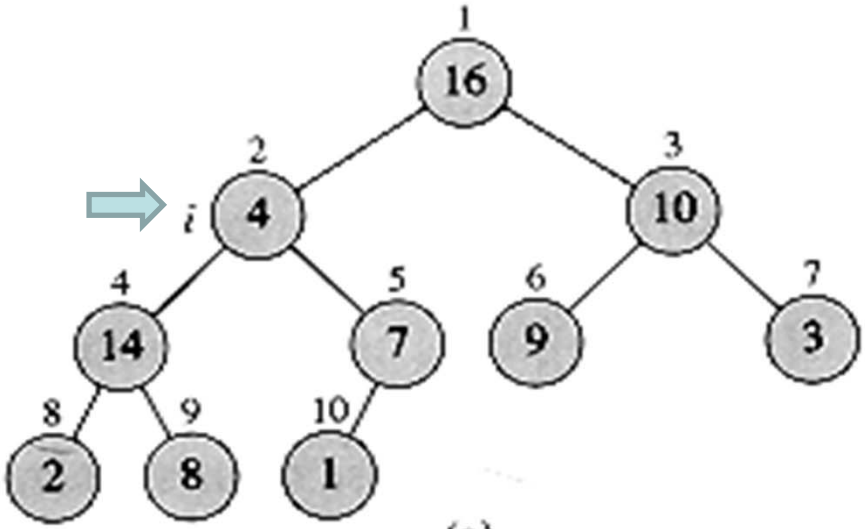
- Next, the vast majority of functions will be written in a **pseudocode** version

Pseudocode of the sift-down function

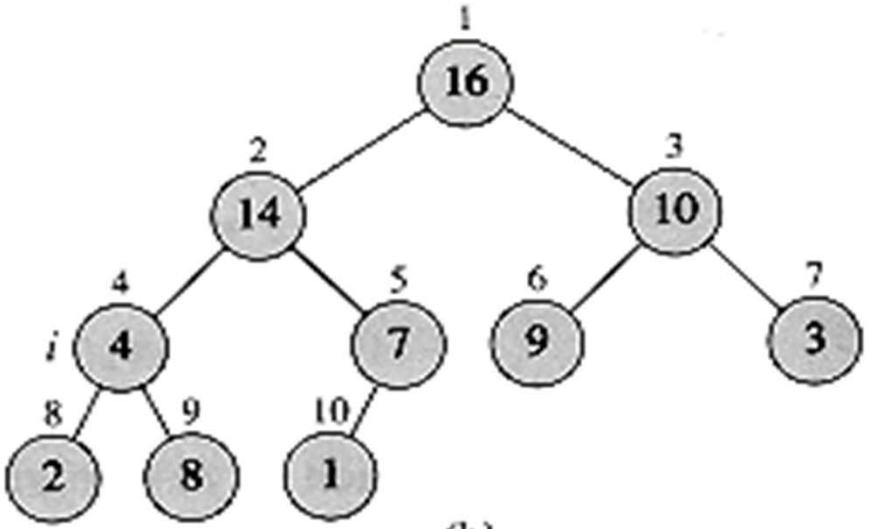
```
void sift_down (T[1…n], i)
{ int k, x, j;
  k ← i ;
  do      {
            j ← k ;
            if ((2j ≤ n) ∧ (T[2j] > T[k]))  then k ← 2j;
            if ((2j+1 ≤ n) ∧ (T[2j+1] > T[k]))  then k ← 2j+1;
            x ← T[j];
            T[j] ← T[k];
            T[k] ← x
          } while (j ≠ k)
}
```



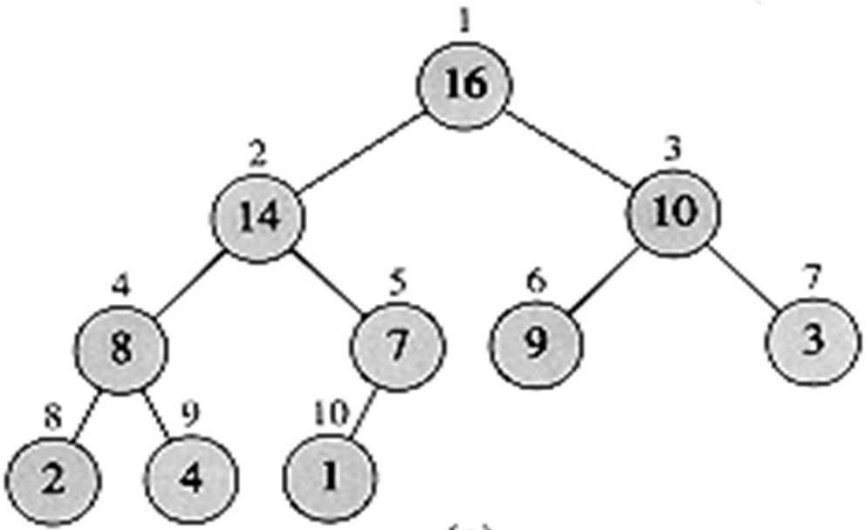
(a) The initial configuration of the heap, with **A[2]** at node **i** = 2 violating the heap property since it is not larger than both children. The heap property is restored for node 2 in **(b)** by exchanging A[2] with A[4] (by using other two variable k, j and a buffer x, where initially k=i and j=k), which destroys the heap property for node 4. The sift-down function continues with a loop till there are no further change to the data structure. Here, this is visible by swapping A[4] with A[9], as shown in **(c).**


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

(a)


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

(b)


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

(c)

Pseudocode of the sift-up function

```
void sift_up (T[1…n], i)
{ int k, j, x;
  k ← i ;
  do      {
           j ← k ;
           if ((j > 1) ∧ (T[ j div 2] < T[k]))  then k ← j div 2;
           x ← T[ j];
           T[ j] ← T[k];
           T[k] ← x
          } while (j ≠ k)
}
```

Restore the heap property

We consider T[1..n] as being a heap.

Having **i**, 1≤i≤n, we can assign to T[i] the value ν, and then we can restore the heap property.

```
void restore_heap (T[1..n], i, ν)
{local variable x;
  x ← T[i];
  T[i] ← ν;
  if  ν < x       then sift_down(T, i);
                  else sift_up(T,i);
}
```

The heap is a useful model for:

- Find the maximum item of a max-heap or a minimum item of a min-heap.
- Adding a new node to the heap.
- Change the value of a node (with **restore_heap**).

Previous operations can be used to implement a dynamic list of priorities:

- The node value of a corresponding element will indicate its priority.
- The event with the highest probability will always be at the root of the heap.
- The priority of a node can be changed dynamically.

These are some principles underlying the database programs.

Examples of useful functions:

1) **The function for finding the maximum value**:

```
find_maxim (T[1..n])
{  return T[1];
}
```

2) **The function to extract a maximum (and remove it):**

```
extr_max (T[1..n])
{ var loc x;
  x ← T[1];
  T[1] ← T[n];
  sift_down (T[1..n-1], 1);
  return x;
}
```

3) **Insertion of a new element in the heap**:

```
insert (T[1..n], ν)
{
    T[n+1] ← ν;
    sift_up (T[1..n+1], n+1);
}
```

Notes: **in C language**, it is not taken into consideration the maximum number of elements of an array used as a parameter function, so that, for example, for the position of sift-up, sift-down, etc., will have to take into account an additional parameter.

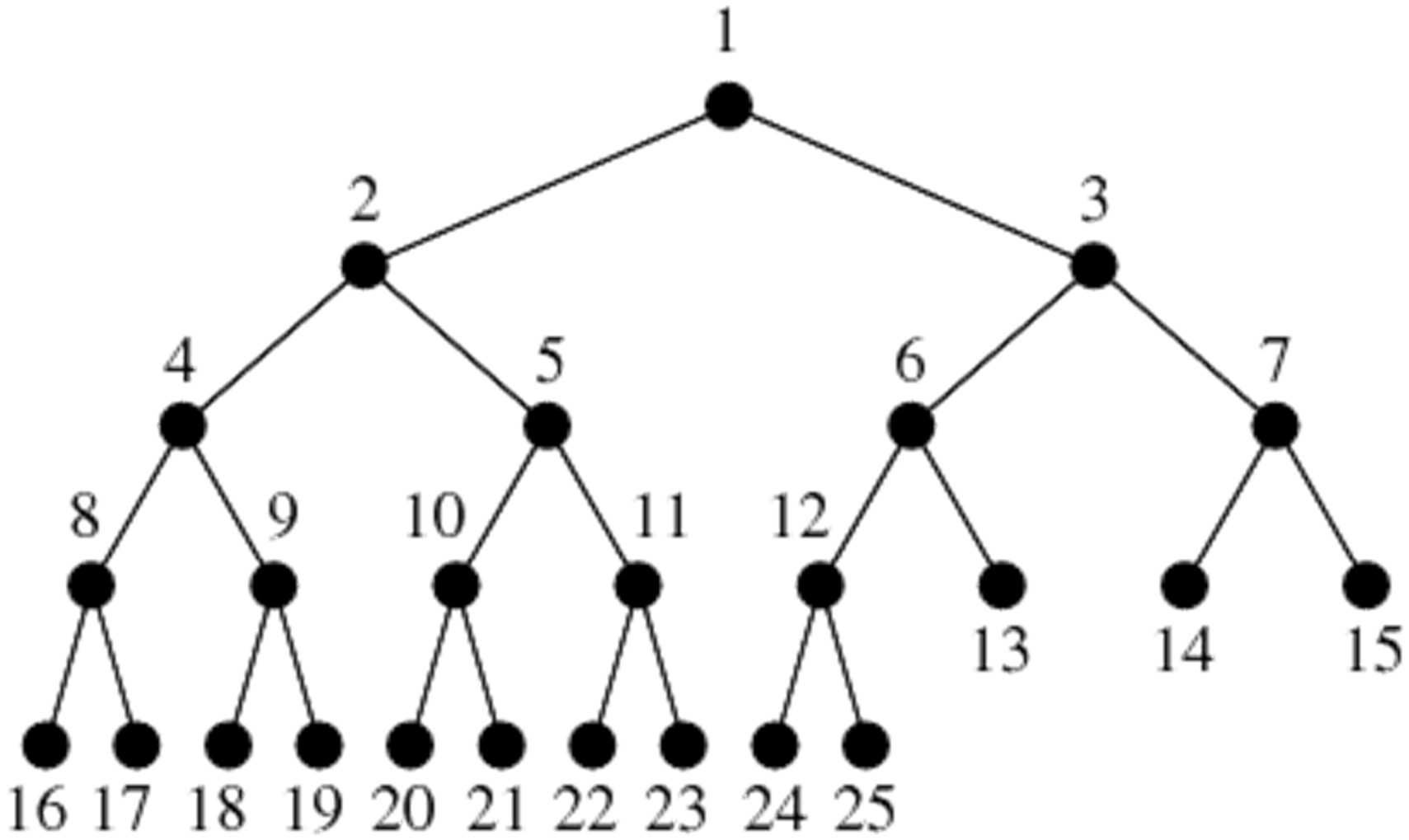
Example: **sift_down (T[ ], *n*, i)**

where **n** indicates the index of the last element of the vector (the numbering should start from **0** in C language!).

How can we create a heap from an unordered vector T[1..n] ?

- A less effective solution is to start from a heap of one element and add items one by one.

```
slow_make_heap(T[1..n])
{
   for (i=2; i ≤ n; i++)
        sift_up (T[1..i], i);
}
```


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

But there is another linear algorithm that works better (in terms of order / efficiency):

```
make_heap(T[1..n])
{
        for (i = n div 2; i ≥ 1; i - -)
                sift_down (T[ ], i);
}
```

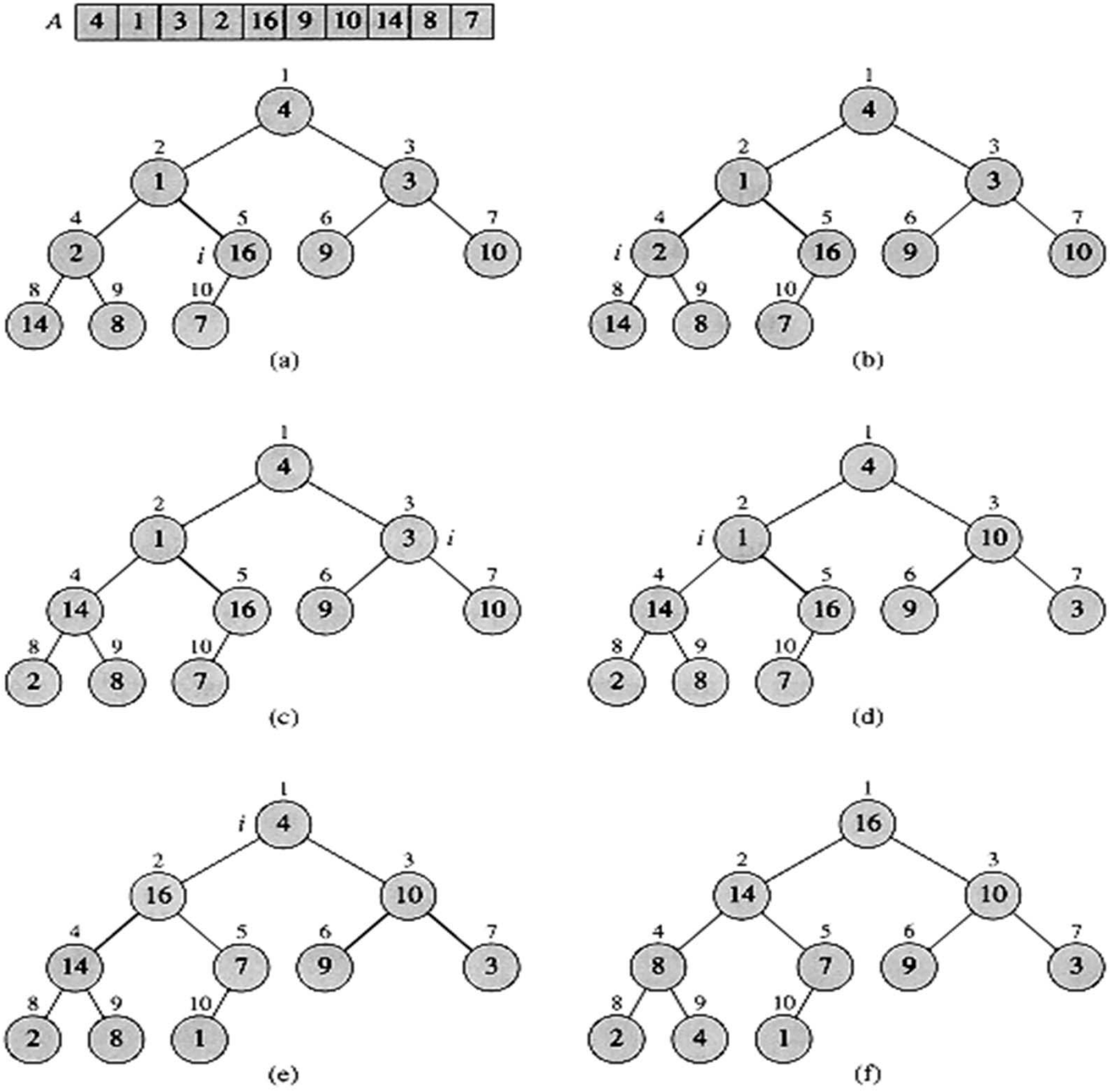
How to build a heap tree from an arbitrary array

- Starting from the following vector (which is not a heap):

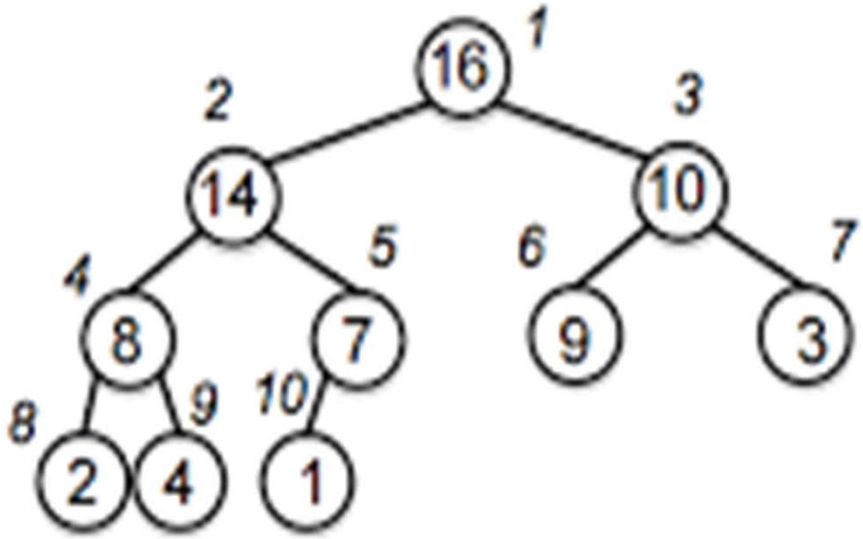
4	1	3	2	16	9	10	14	8	7
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Through a sequence of few steps we can obtain:

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


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

The obtained heap tree:



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

This heap can be represented by the following vector:

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

Other example (homework):

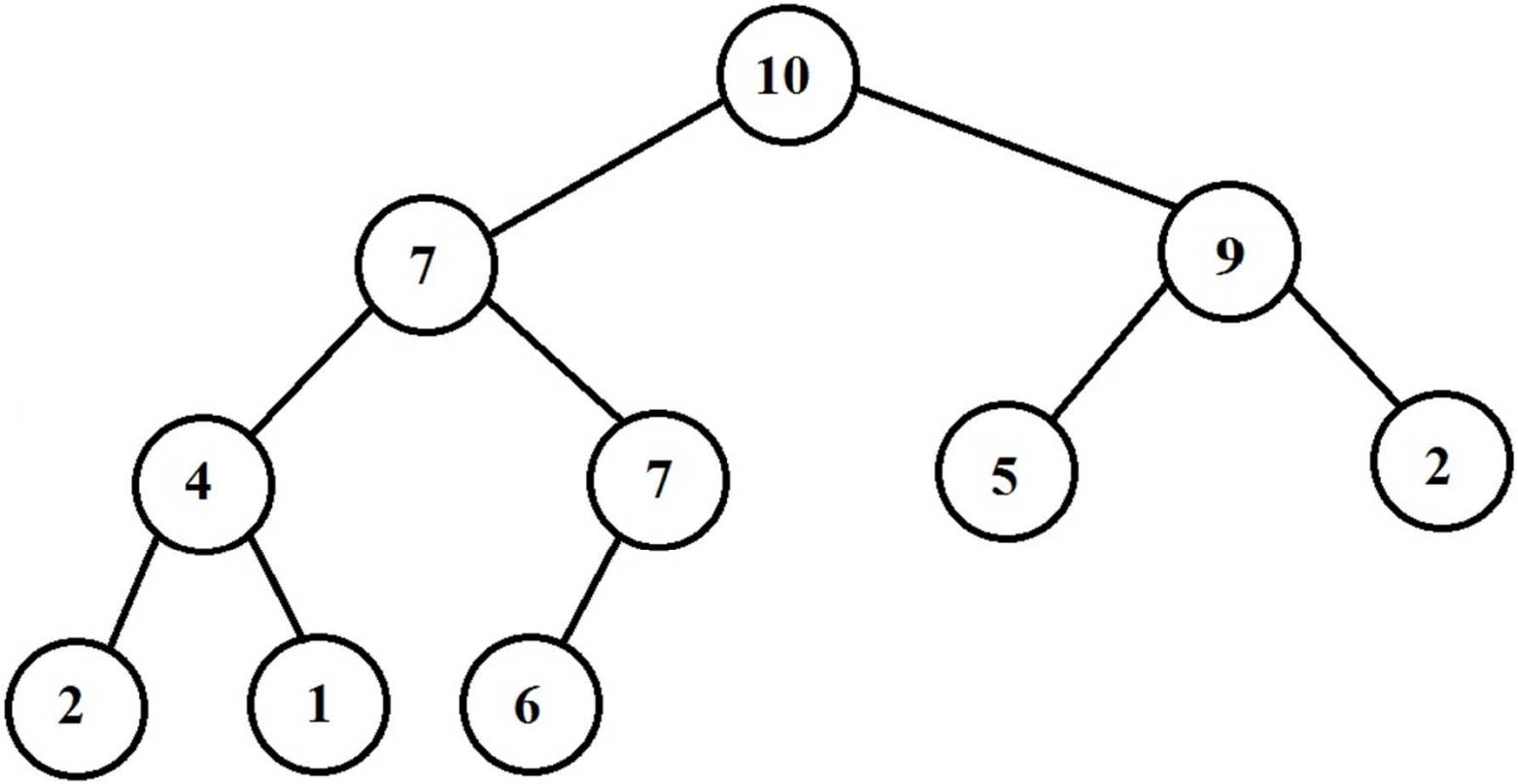
- Starting from the following vector (which is not a heap):

1	6	9	2	7	5	2	7	4	10
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Through a sequence of few steps we can obtain:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

The result should be like this:



This heap can be represented by the following vector:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

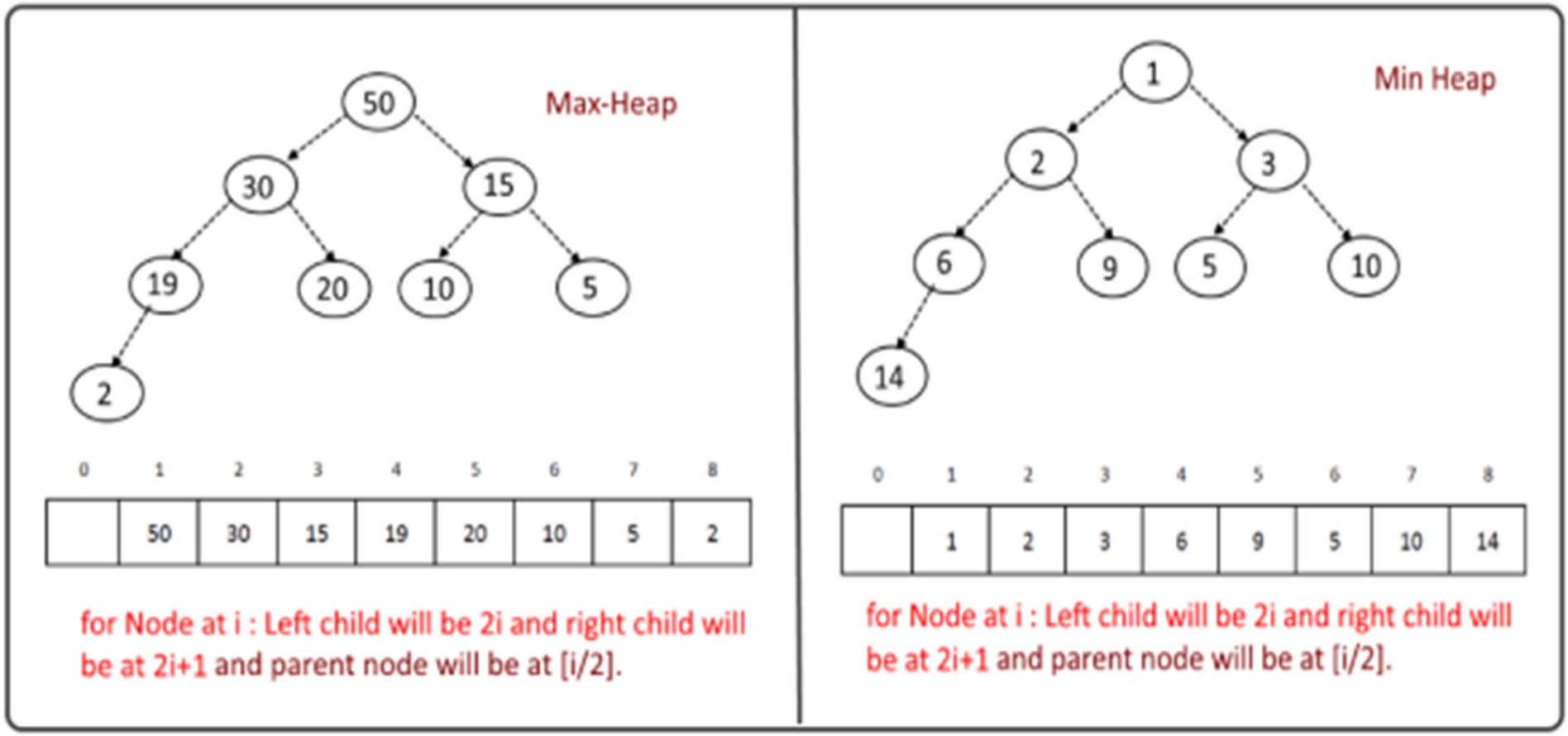
Remember

- In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

Note:

- All procedures, used to handle a heap, are changed accordingly to this new situation.

Classic heap (Max Heap) vs Min Heap



Warning!

- The heap data structure is very attractive, but it does have limitations.

- There are operations that can not be performed effectively in a heap.

The disadvantages of a heap:

- The tree has to be a complete one.

- Finding a peak with a certain value is inefficient (because there may be several nodes with the same value placed on different branches/ or levels in the heap).

• An extension of the concept of heap is possible for the complete trees with more than two children.

• Such an approach will accelerate the sift-down procedure.

The main application of the heap concept: A new sorting technique (**heapsort**)

- The **heapsort** algorithm was invented by J. W. J. Williams in 1964. In the same year, R. W. Floyd published an improved version that could sort an array in-place, continuing his earlier research into the treesort algorithm.

```
heapsort (T[1..n])
{
        make_heap(T);
        for( i = n; i ≥ 2; i - -)
                {
                 T[1] ↔ T[i];
                 sift_down (T[1…i-1], 1)
                }
}
```

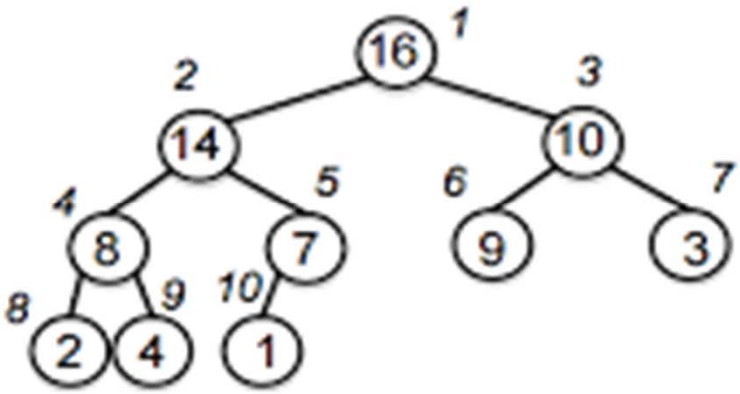
Example

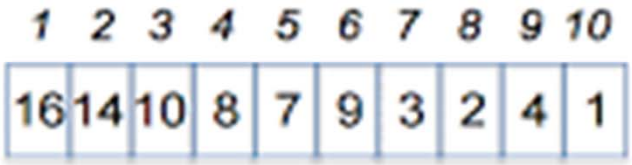
First task is to obtain a heap tree from an unsorted vector

Start from the following vector (which is not a heap):

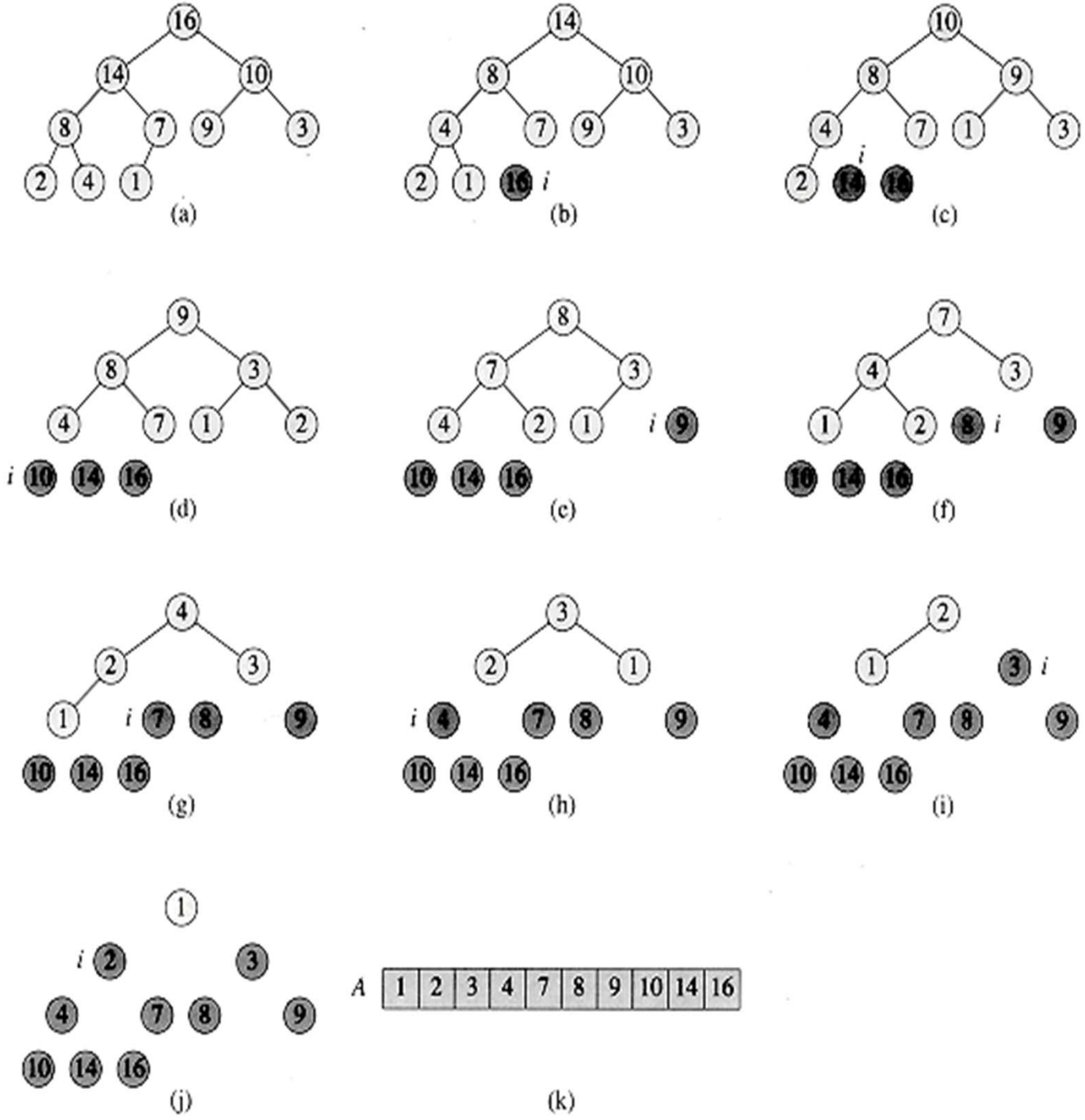
4	1	3	2	16	9	10	14	8	7
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

After **make_heap** we obtain:





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

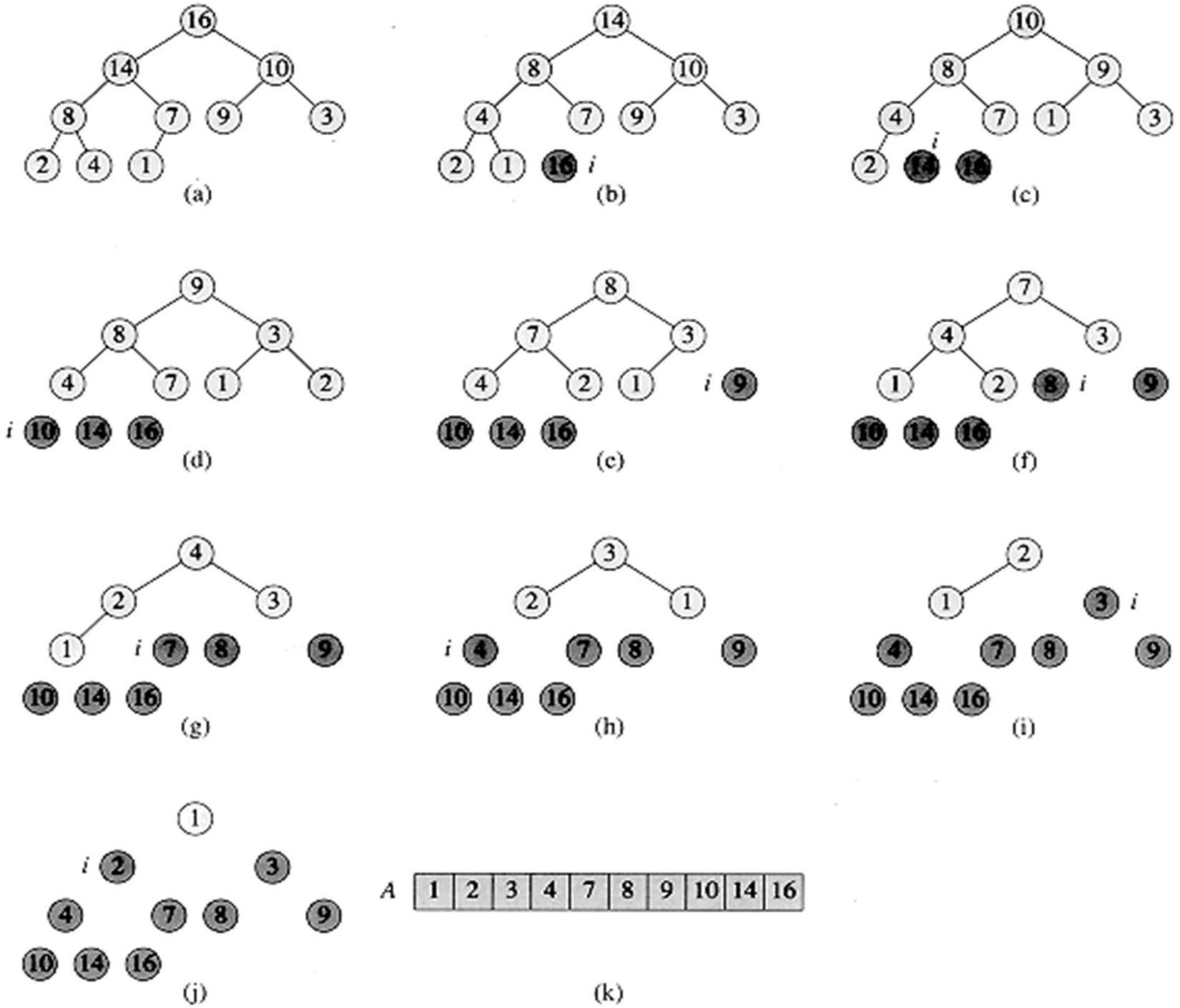


The operation of HEAPSORT:

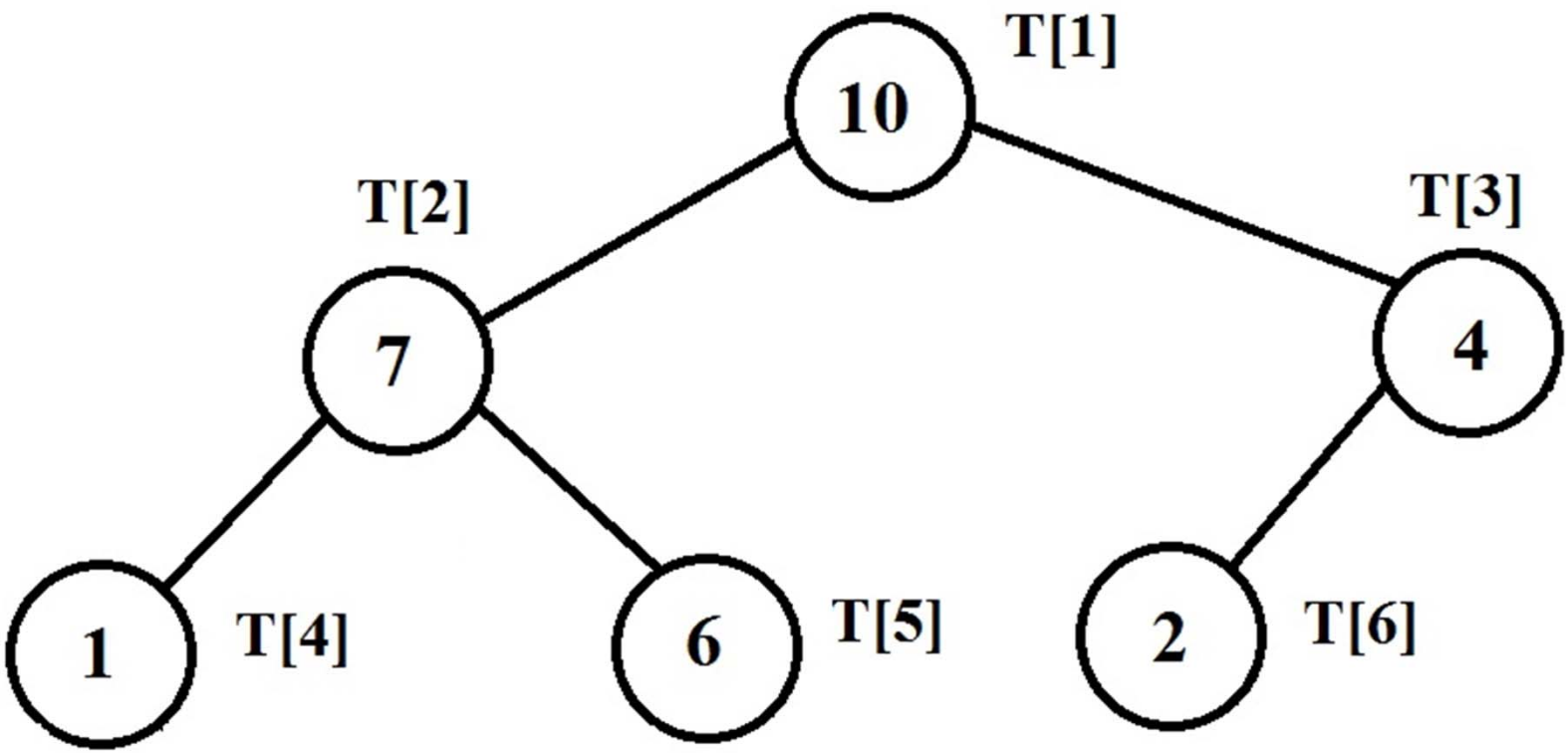
(a) The heap data structure just after it has been built by **make_heap**.

(b)-(j) The heap just after each call of **sift_down(T[1…i-1], 1)**. The value of i at that time is shown. Only lightly shaded nodes remain in the heap.

(k) The resulting sorted array A


16 14 10 8 7 9 3 2 4 1
(a)
14 8 10 4 7 9 3 2 1 16 i
(b)
10 8 9 4 7 1 3 i 2 14 16
(c)
9 8 3 4 7 1 2 i 10 14 16
(d)
8 7 3 4 2 1 i 9 10 14 16
(e)
7 4 3 1 2 8 i 9 10 14 16
(f)
4 2 3 1 i 7 8 9 10 14 16
(g)
3 2 1 i 4 7 8 9 10 14 16
(h)
2 1 3 i 4 7 8 9 10 14 16
(i)
1 i 2 3 4 7 8 9 10 14 16
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Other example (homework):



Other ordering (or sorting) algorithms:

- **Insertion**
- **Selection**
- **Merge sort**
- **Quicksort**
- **etc.**

Ordering methods

- Ordering = arranging items of the same kind, class or nature, in an ordered sequence.

- For this purpose we consider that the data are a collection of items of a certain type and each item comprises one or even more values (variables), which are decisive in the ordering that is performed. Such a value is called **key**.

For the C programming language, a sorting algorithm can be achieved by one of the following methods:

1. Arranging data (which are sorted) so that their keys will finally correspond to the desired order.

2. By ordering an array of pointers (to the data that must be sorted) in order to form an ordered set.

Note:

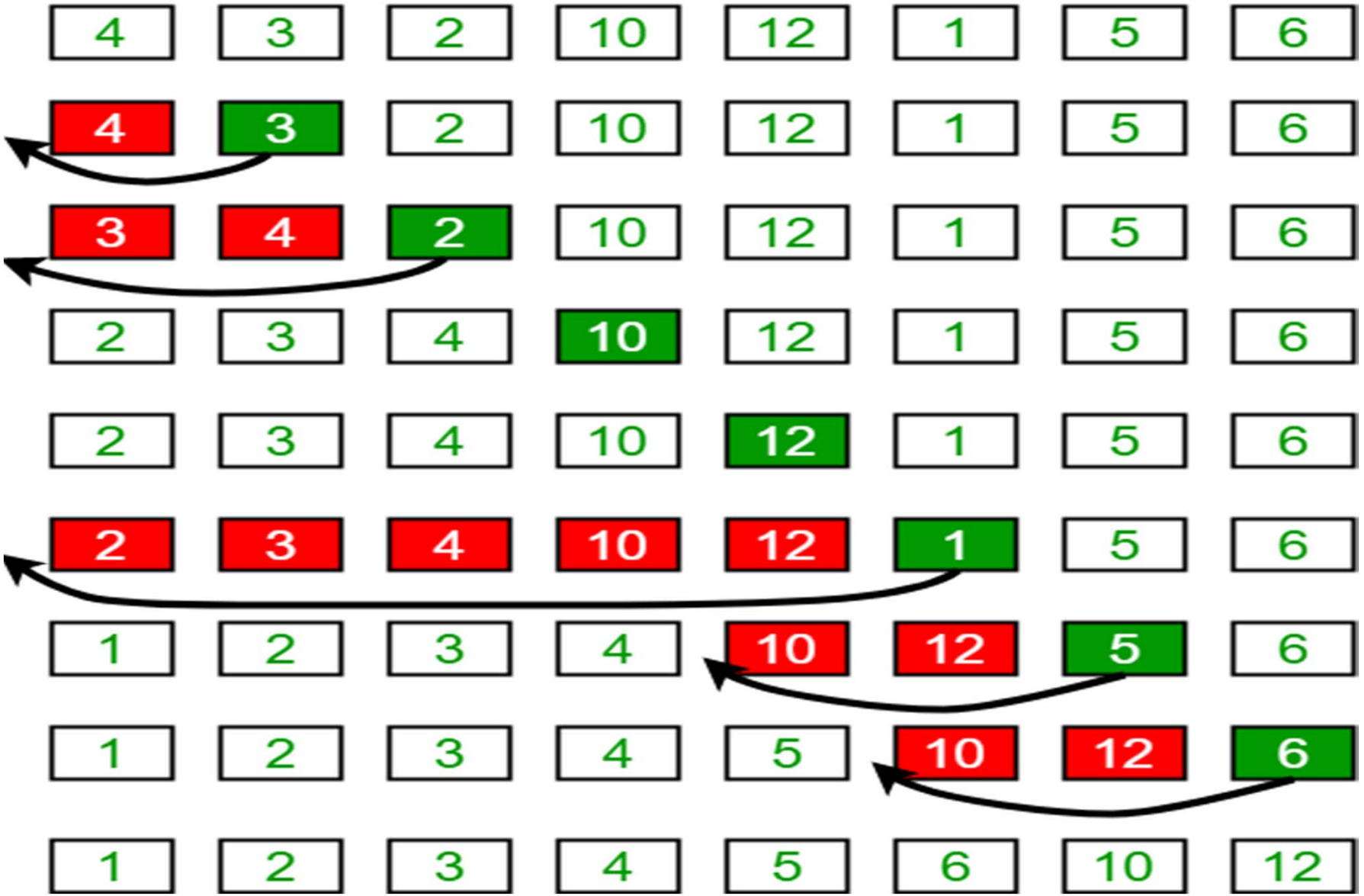
- In the following we will discuss only about sorting unidimensional arrays (vectors) with numerical data.

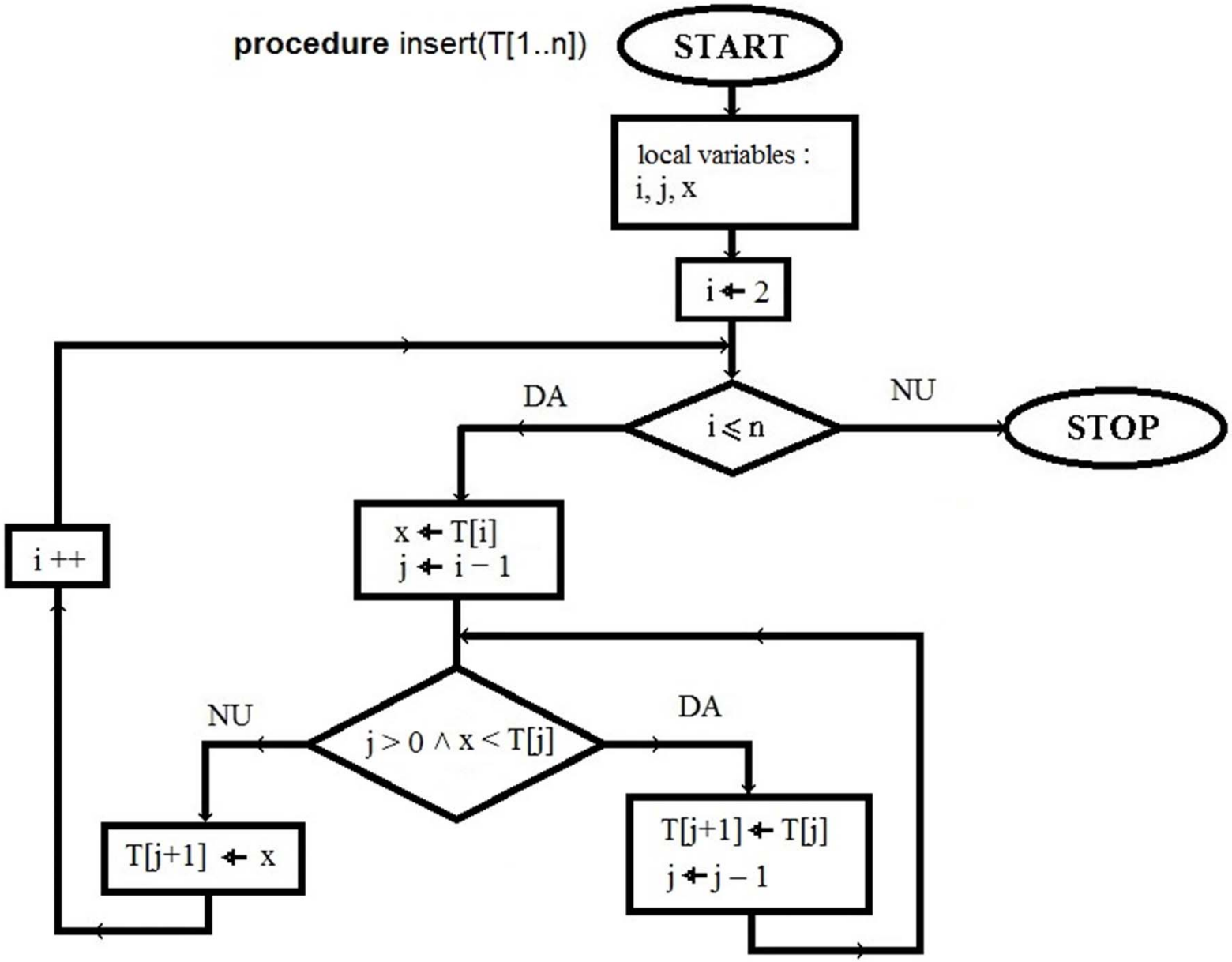
Insertion sorting algorithm

- Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.

- The general idea of sorting by insertion is to consider (at a time) each element of the array and insert it into the substring previously ordered.
- The operation involves a growing sequence, which is moved to the right.

Insertion Sort Execution Example




procedure insert(T[1..n])
START
local variables :
i, j, x
i ← 2
DA
i ⩽ n
NU
STOP
x ← T[i]
j ← i − 1
i ++
NU
j > 0 ∧ x < T[j]
DA
T[j+1] ← x
T[j+1] ← T[j]
j ← j − 1

A brief description of the algorithm in pseudocode is as follows:

```
procedure insert(T[1..n])
{
        local variables  i, j, x
        for i ← 2 to n do
        {
                x ← T[i]
                j ← i -1
                while (j>0 and x<T[ j]) do
                        {
                         T[ j+1] ← T[ j]
                          j ← j – 1
                        }
                T[ j+1] ← x
        }
}
```

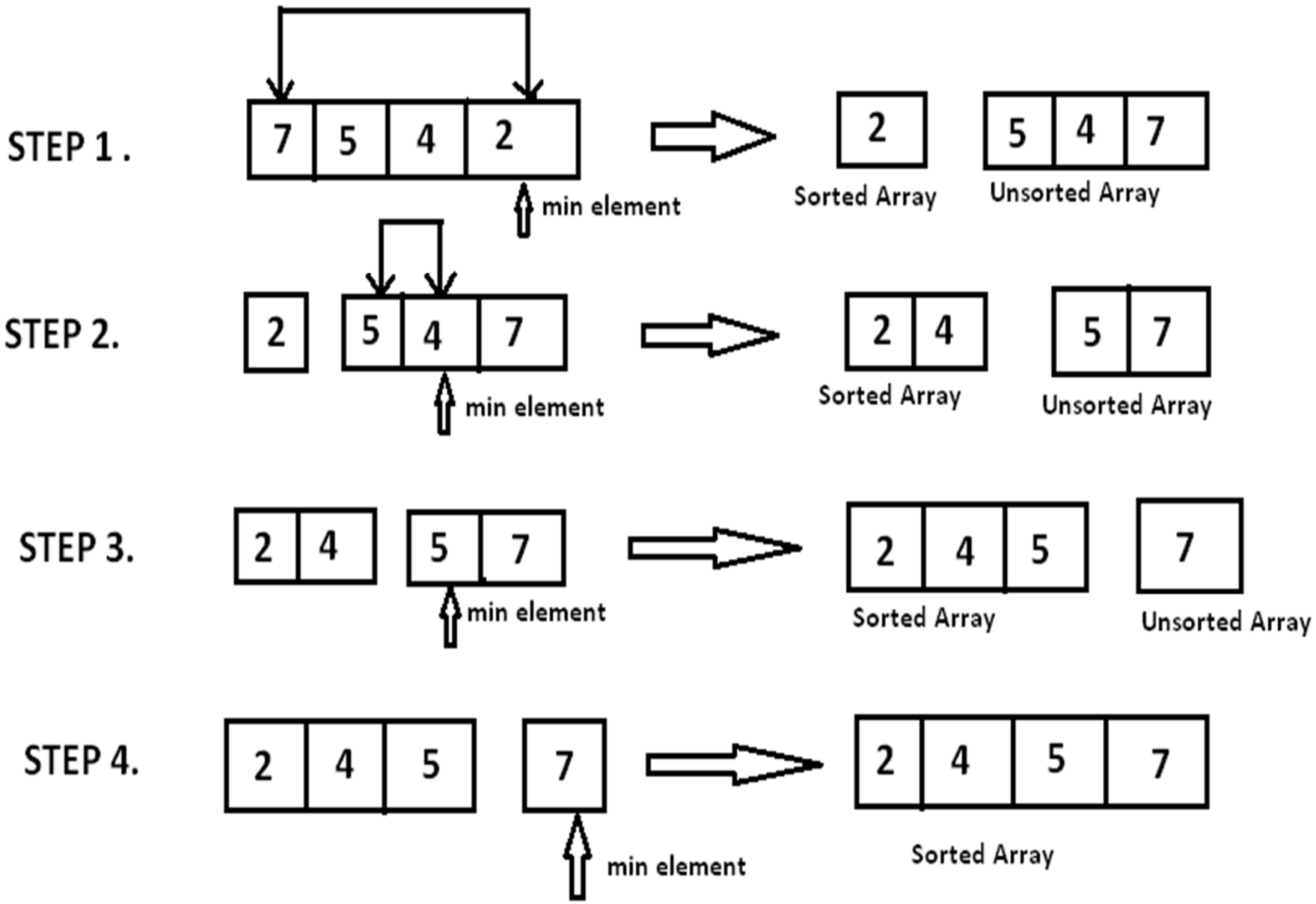
Note: In implementing the above algorithm in C code, we have to keep in mind that a vector is starting with an zero index.

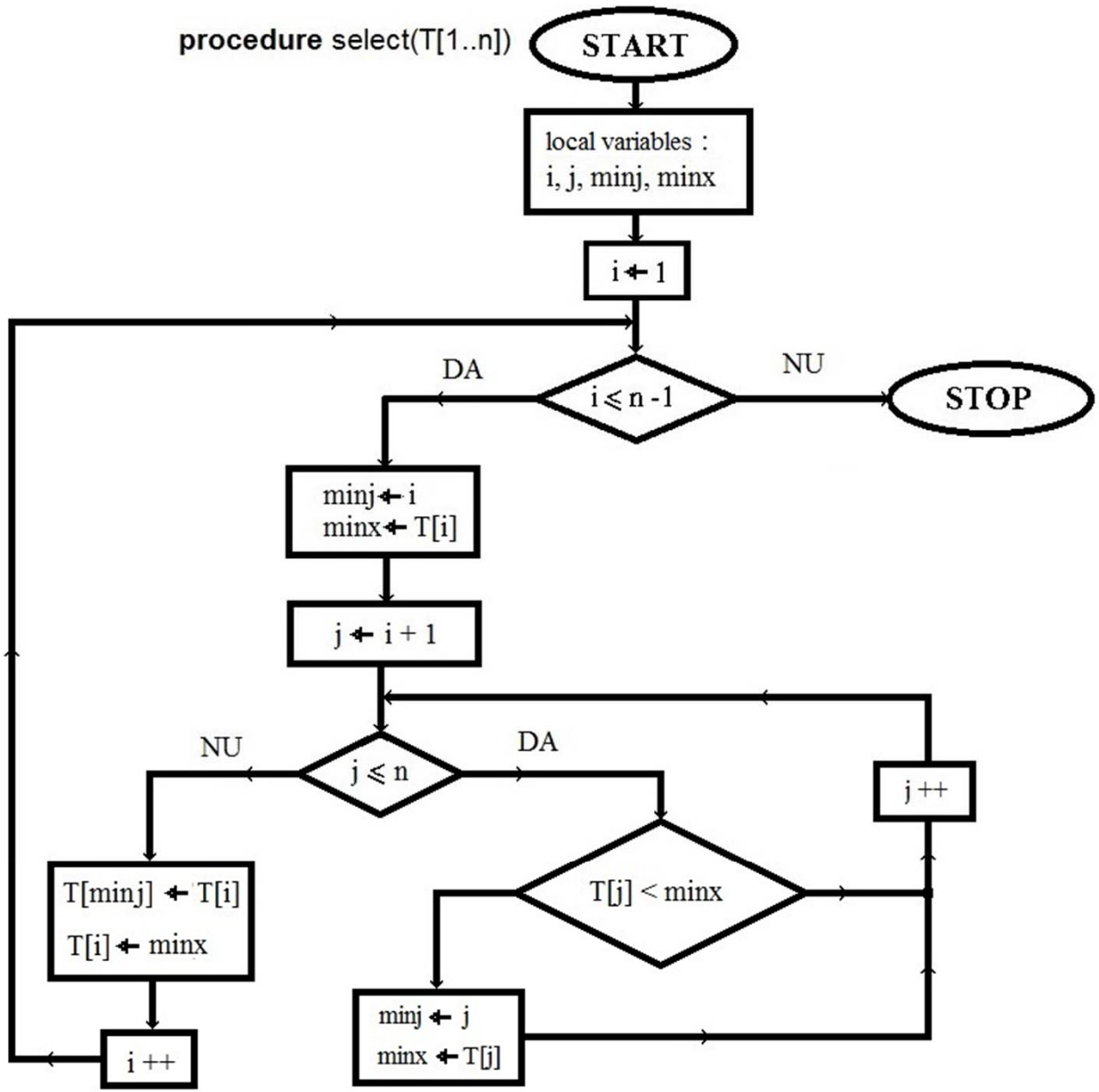
The selection sorting algorithm

- Selection sort is a sorting algorithm, specifically an in-place comparison sort.
- Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

How it works

- The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.
- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.
- The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array


procedure select(T[1..n])
START
local variables :
i, j, minj, minx
i ← 1
DA
i ⩽ n -1
NU
STOP
minj ← i
minx ← T[i]
j ← i + 1
NU
j ⩽ n
DA
j ++
T[j] < minx
T[minj] ← T[i]
T[i] ← minx
minj ← j
minx ← T[j]
i ++

A brief description of the algorithm in pseudocode is as follows:

```
procedure select(T[1..n])
{
   local variables  i, j, min_j, min_x,
   for i ← 1 to n-1 do
            { min_j ← i
              min_x ← T[i]
              for j ← i+1 to n do
                        if T[j] < min_x then
                                  { min_j ← j
                                    min_x ← T[j]
                                  }
              T[min_j] ← T[i]
              T[i] ← min_x
            }
}
```

Note: In implementing the above algorithm in C code, we have to keep in mind that a vector is starting with an zero index.

Bubblesort algorithm

- Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.

Shellsort algorithm

- Shellsort is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).
- The method starts by sorting pairs of elements far apart from each other, then progressively reducing the **gap** between elements to be compared. Starting with far apart elements can move some out-of-place elements into position faster than a simple nearest neighbor exchange.
- Donald Shell published the first version of this sort in 1959.

- The running time of Shellsort is heavily dependent on the **gap** sequence it uses.

- For many practical variants, determining their time complexity remains an open problem.

***Shell sort method can be defined as in the following* :**

1) Start with a **gap = n/2**, where n is the number of the elements that will be sorted.
2) Make a crossing of the vector of items that are sorted.
3) The gap it halves **gap** = **gap/2** .
4) If **gap > 0**, then jump to step 2, otherwise the algorithm stops.

Note *Each crossing through the elements involves the following substeps*:

1) **i = gap**.
2**) j = i – gap + 1** .
3) If **j > 0** and the elements from the positions: **j** and **j+gap** are not ordered, then we will interchange their values. Otherwise jump to substep 6.
4) **j = j – gap**.
5) Jump to substep 3.
6) **i = i + 1**.
7) If **i > n**, the crossing is stopped. Otherwise jump to substep 2.

Notes:

- In implementing the above algorithm in C code, we have to remember that a vector is starting with the zero index.

- Therefore, the initialization from the substep 2) becomes: **j = i – gap** .

Backtracking

- **Backtracking is a general algorithm for finding all solutions of a problem of calculation algorithm that is based on building incremental candidate solutions, each candidate partially abandoned as soon as it becomes clear that he has no chance to be a valid solution.**

- See the last laboratory…

http://www.euroqual.pub.ro/wp-content/uploads/sda_lab_06_backtracking.pdf

Backtracking

- Backtracking depends on:

  - user-given "black box procedures" that define the problem to be solved,

  - the nature of the partial candidates,

  - how they are extended into complete candidates.

It is therefore a metaheuristic rather than a specific algorithm – although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.

Analysis of algorithms efficiency

- The analysis of algorithms is the determination of the amount of resources, which are necessary to execute them.

Resources mean:

- ***The memory space*** required for storing the data, which are being processed by the algorithm.

- ***The time*** required for execution of all specified processes of the algorithm.