Lecture 11

Binary Search Trees II & Heaps

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Huffman Code

- ASCII (Fixed-Length Code)
  - Fixed 7-bit length for all characters.
  - No consideration of character frequency (e.g., 'E' and 'Z' use the same bits).
  - Every character uses maximum bit length.
- Huffman Code
  - Shorter codes for frequent characters, longer codes for rare ones.
  - Frequent ('E', 'T') → 1 bit.
  - Less frequent ('A', 'O', 'R', 'N') → 2 bits.
  - Rare characters → more bits.
- Reduces overall transmission size by encoding efficiently.
- Widely used for data compression.

Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

https://web.alfredstate.edu/faculty/weimandn/miscellaneous/ascii/ascii_index.html

Huffman Code

- Use character weights to construct a binary tree.
- Frequent characters are placed closer to the root (shorter codes).
- Rare characters are deeper in the tree (longer codes).

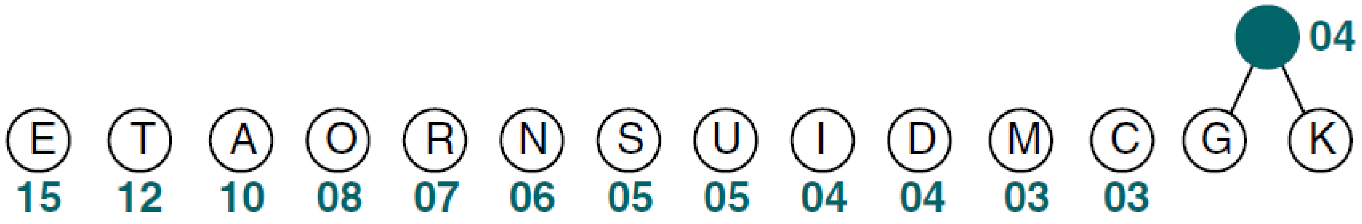
- **Step 1:** Sort characters by frequency
- **Step 2:** Merge lowest-frequency nodes
- **Step 3:** Repeat Step 2 until a single tree remains

Huffman Code

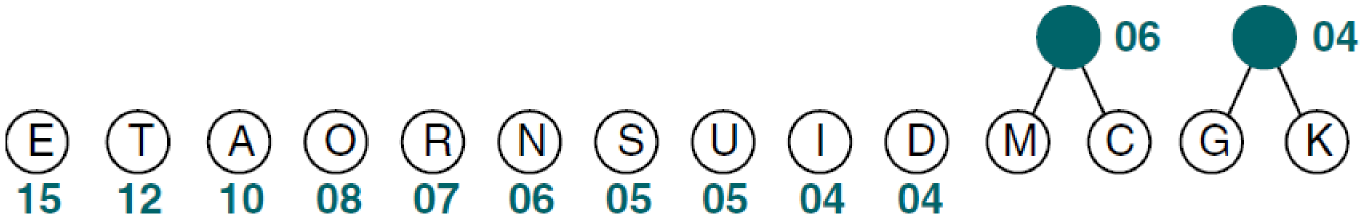
Character	Weight	Character	Weight	Character	Weight
A	10	I	4	R	7
C	3	K	2	S	5
D	4	M	3	T	12
E	15	N	6	U	5
G	2	O	8		

E	T	A	O	R	N	S	U	I	D	M	C	G	K
15	12	10	08	07	06	05	05	04	04	03	03	02	02

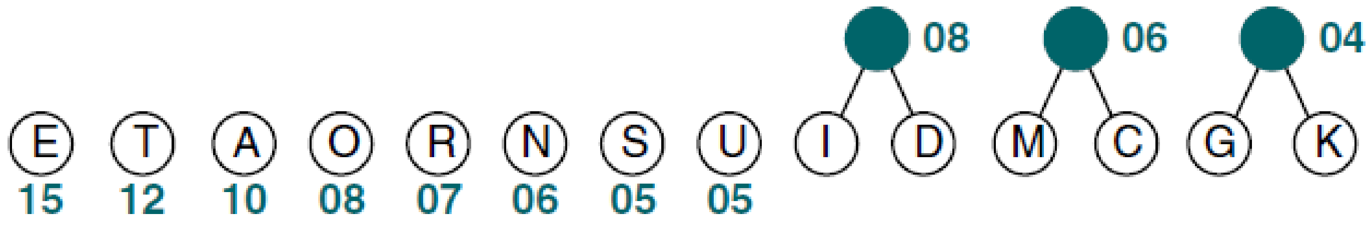
Huffman Code



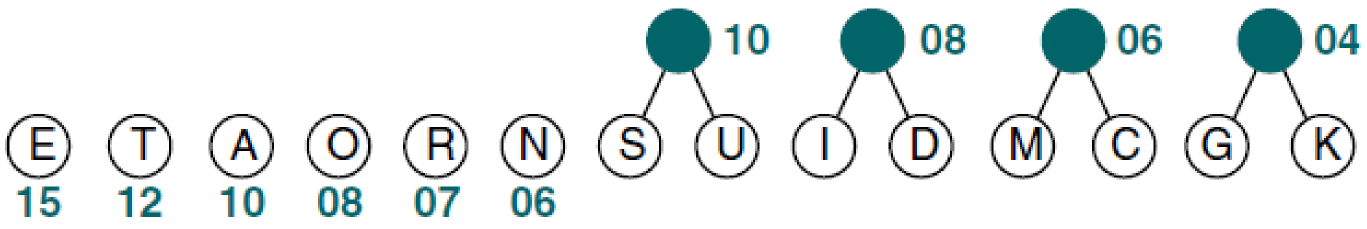
Huffman Code



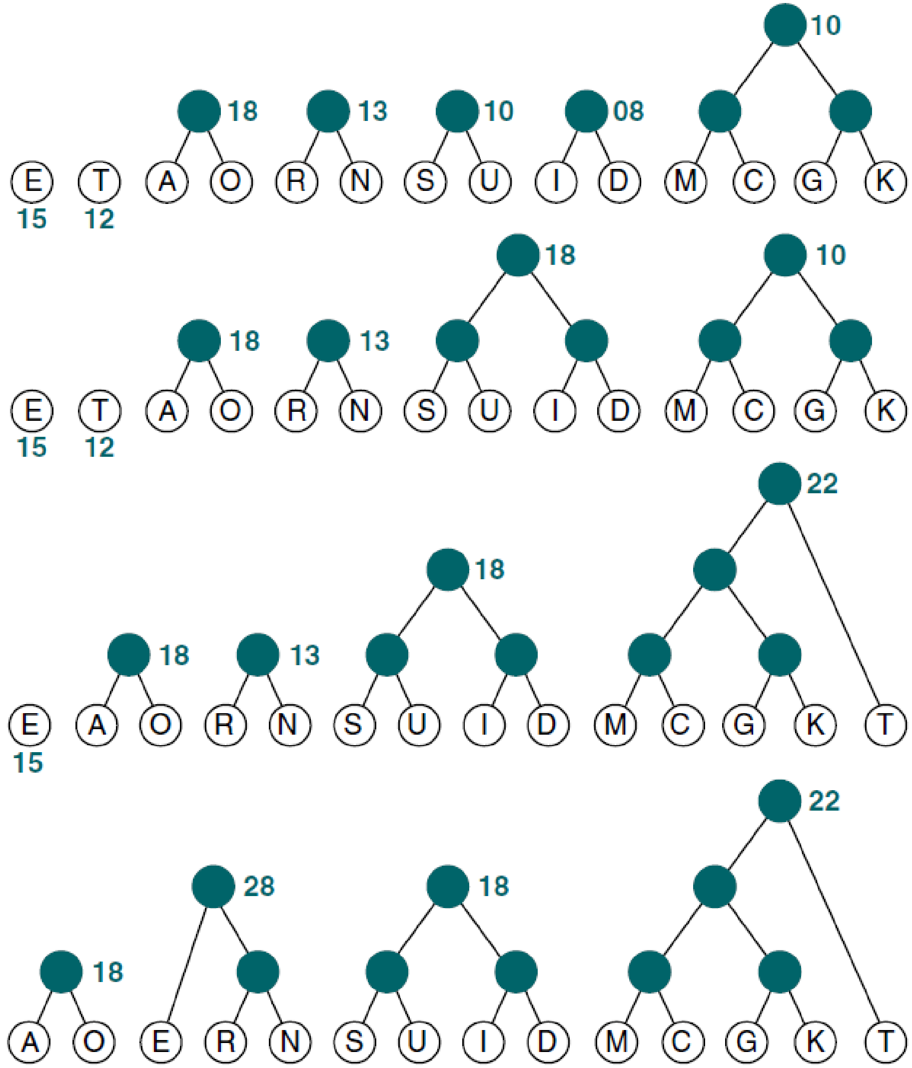
Huffman Code



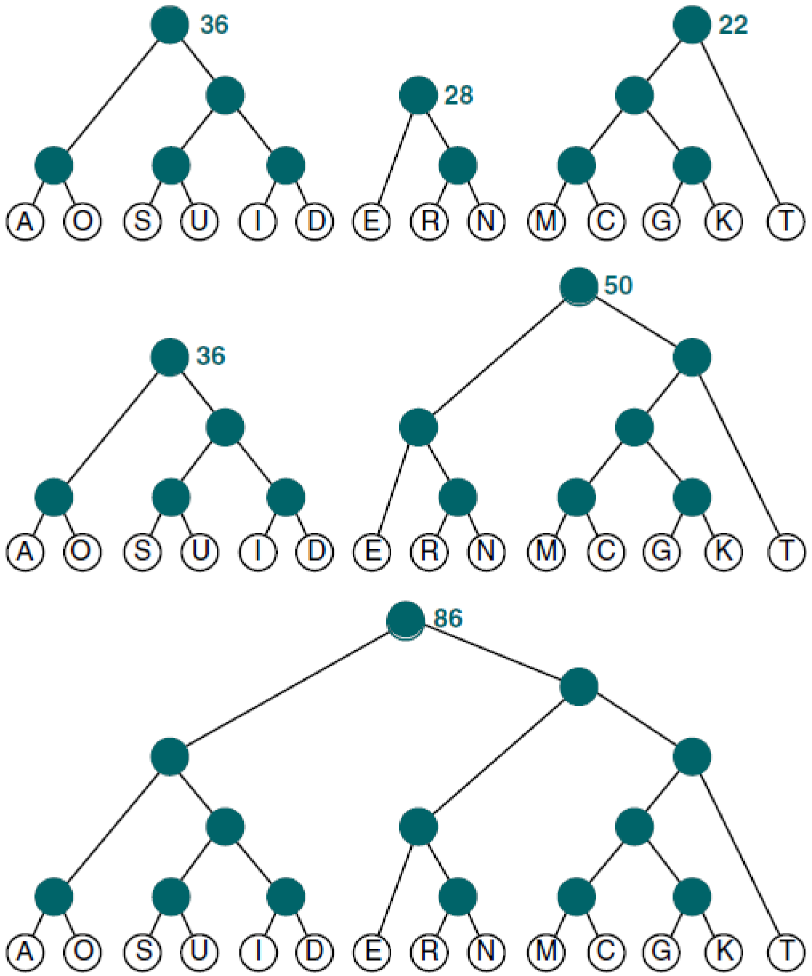
Huffman Code



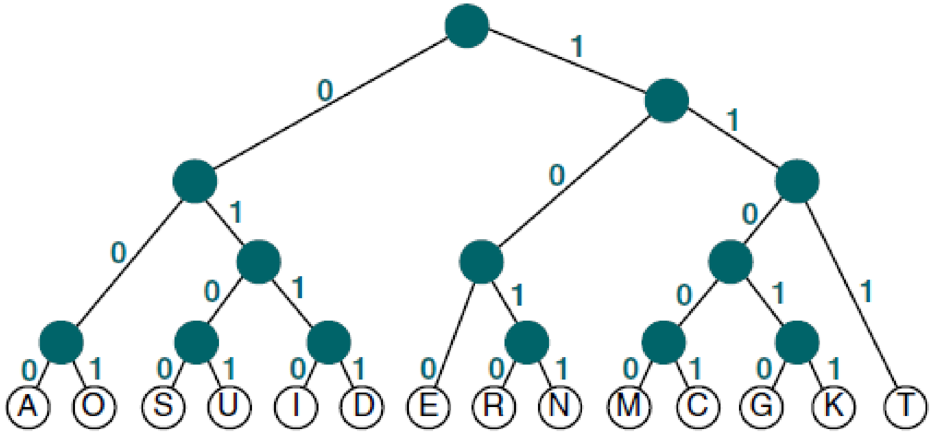
Huffman Code



Huffman Code



Huffman Code



A = 000	U = 0101	E = 100	M = 11000	K = 11011
O = 001	I = 0110	R = 1010	C = 11001	T = 111
S = 0100	D = 0111	N = 1011	G = 11010	

Huffman Code

```c
typedef struct Node {
    char character;
    int frequency;
    struct Node *left, *right;
} Node;

Node* createNode(char character, int frequency) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->character = character;
    node->frequency = frequency;
    node->left = node->right = NULL;
    return node;
}
```

```c
void findTwoSmallest(Node* nodes[], int size, int* smallest, int* secondSmallest) {
    *smallest = -1;
    *secondSmallest = -1;

    for (int i = 0; i < size; i++) {
        if (nodes[i] != NULL) {
            if (*smallest == -1 || nodes[i]->frequency < nodes[*smallest]->frequency) {
                *secondSmallest = *smallest;
                *smallest = i;
            } else if (*secondSmallest == -1 || nodes[i]->frequency < nodes[*secondSmallest]->frequency) {
                *secondSmallest = i;
            }
        }
    }
}
```

Huffman Code

```c
Node* buildHuffmanTree(char characters[], int frequencies[], int size) {
    Node** nodes = (Node**)malloc(size * sizeof(Node*));

    for (int i = 0; i < size; i++) {
        nodes[i] = createNode(characters[i], frequencies[i]);
    }

    for (int count = 1; count < size; count++) {
        int smallest, secondSmallest;
        findTwoSmallest(nodes, size, &smallest, &secondSmallest);

        Node* newNode = createNode('\0', nodes[smallest]->frequency + nodes[secondSmallest]->frequency);
        newNode->left = nodes[smallest];
        newNode->right = nodes[secondSmallest];

        nodes[smallest] = newNode;
        nodes[secondSmallest] = NULL;
    }

    for (int i = 0; i < size; i++) {
        if (nodes[i] != NULL) {
            return nodes[i];
        }
    }
    return NULL;
}
```

```c
void printHuffmanTree(Node* root, char* code, int top) {
    if (root->left) {
        code[top] = '0';
        printHuffmanTree(root->left, code, top + 1);
    }
    if (root->right) {
        code[top] = '1';
        printHuffmanTree(root->right, code, top + 1);
    }
    if (!root->left && !root->right) {
        code[top] = '\0';
        printf("Character: %c, Code: %s\n", root->character, code);
    }
}
```

```c
char characters[] = {'a', 'b', 'c', 'd', 'e', 'f'};
int frequencies[] = {5, 9, 12, 13, 16, 45};
int size = sizeof(characters) / sizeof(characters[0]);

Node* root = buildHuffmanTree(characters, frequencies, size);

char code[100];
printHuffmanTree(root, code, 0);
```

```
Character: f, Code:
Character: c, Code:
Character: d, Code:
Character: a, Code:
Character: b, Code:
Character: e, Code:
```

Huffman Code

```c
Node* buildHuffmanTree(char characters[], int frequencies[], int size) {
    Node** nodes = (Node**)malloc(size * sizeof(Node*));

    for (int i = 0; i < size; i++) {
        nodes[i] = createNode(characters[i], frequencies[i]);
    }

    for (int count = 1; count < size; count++) {
        int smallest, secondSmallest;
        findTwoSmallest(nodes, size, &smallest, &secondSmallest);

        Node* newNode = createNode('\0', nodes[smallest]->frequency + nodes[secondSmallest]->frequency);
        newNode->left = nodes[smallest];
        newNode->right = nodes[secondSmallest];

        nodes[smallest] = newNode;
        nodes[secondSmallest] = NULL;
    }

    for (int i = 0; i < size; i++) {
        if (nodes[i] != NULL) {
            return nodes[i];
        }
    }
    return NULL;
}
```

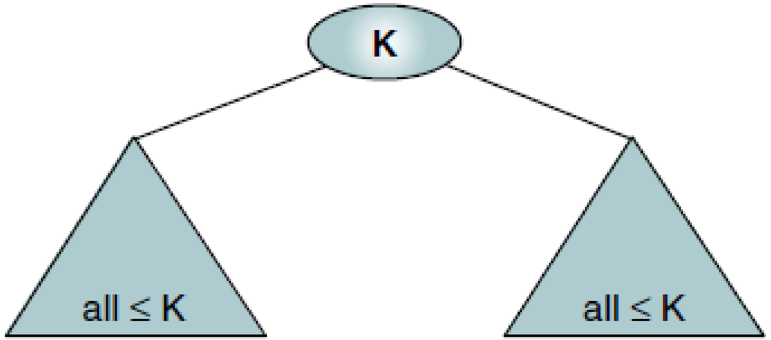
```c
void printHuffmanTree(Node* root, char* code, int top) {
    if (root->left) {
        code[top] = '0';
        printHuffmanTree(root->left, code, top + 1);
    }
    if (root->right) {
        code[top] = '1';
        printHuffmanTree(root->right, code, top + 1);
    }
    if (!root->left && !root->right) {
        code[top] = '\0';
        printf("Character: %c, Code: %s\n", root->character, code);
    }
}
```

```c
char characters[] = {'a', 'b', 'c', 'd', 'e', 'f'};
int frequencies[] = {5, 9, 12, 13, 16, 45};
int size = sizeof(characters) / sizeof(characters[0]);

Node* root = buildHuffmanTree(characters, frequencies, size);

char code[100];
printHuffmanTree(root, code, 0);
```

```
Character: f, Code: 0
Character: c, Code: 100
Character: d, Code: 101
Character: a, Code: 1100
Character: b, Code: 1101
Character: e, Code: 111
```

Heap

- **Parent Nodes:** Always have greater values than their child nodes.
- **Root Node:** Holds the largest value in the entire tree.
- **Subtrees:** Contain values that are less than the root.



- Lesser-valued nodes can be placed in either the left or right subtree.
- Both branches have the same structural properties, unlike a binary search tree.

- **Not a BST!** In a heap, node placement is not restricted by strict ordering rules.

Heapsort

Content from
Susan Bridges are
borrowed for further slides.

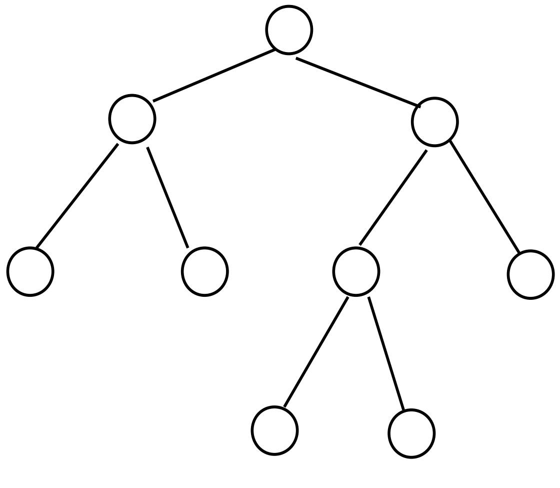
- Running time of heapsort is $O(n \log_2 n)$
- It sorts in place
- It uses a data structure called a *heap*
- The heap data structure is also used to implement a priority queue efficiently

Full and Complete Binary Trees

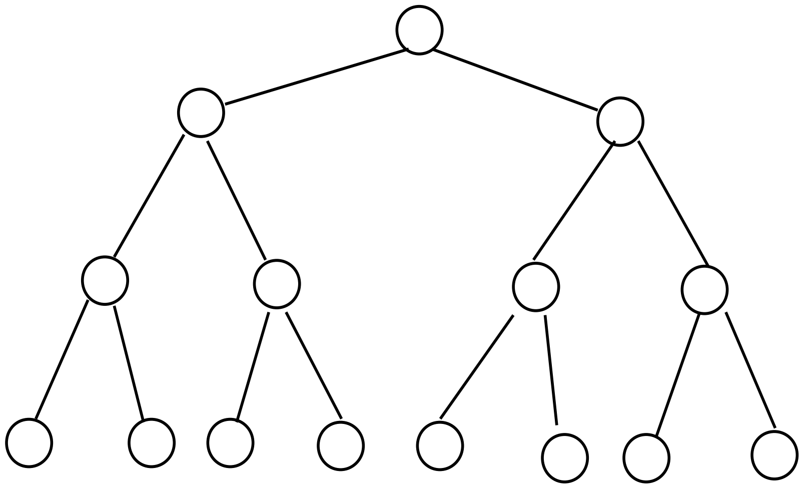
- **Full binary tree:** binary tree in which each node is either a leaf node or has degree 2 (i.e., has exactly 2 children)
- **Complete binary tree:** full binary tree in which all leaves have the same depth
- **Nearly complete binary tree:** completely filled on all levels except possibly the lowest, which is filled *from the left* up to a point

Examples

Full binary tree:



Complete binary tree:



Representation of Nearly Complete Binary Tree

A nearly complete binary tree may be represented as an array (i.e., no pointers):

Number the nodes, beginning with the root node and moving from level to level, left to right within a level.

The number assigned to a node is its index in the array.

Additional Properties of Nearly Complete Binary Trees

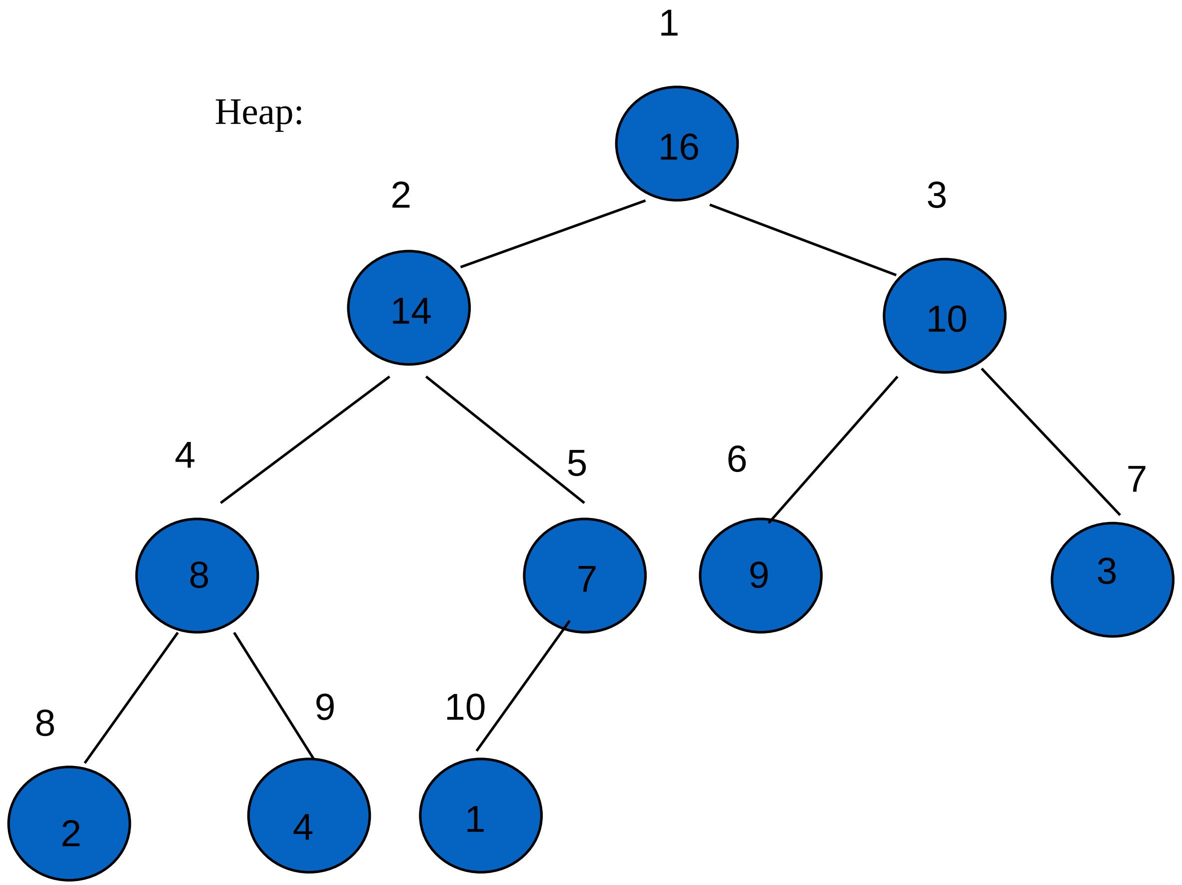
- The root of the tree is A[1].
- If a node has index i, we can easily compute the indices of its:
  - parent $\lfloor i/2 \rfloor$
  - left child $2i$
  - right child $2i + 1$

Numbering

Array:

16	14	10	8	7	9	3	2	4	1

Heap:



Heap

- Implemented as an array object, A[ ]
- Array A that implements the heap has two attributes
    - length(A)
    - heap-size(A)

Heap

A binary tree with n nodes and of height *h* is **almost complete** iff its nodes correspond to the nodes which are numbered 1 to *n* in the complete binary tree of height *h*.

A **heap** is an *almost complete binary tree* that satisfies the **heap property**:

max-heap: For every node *i* other than the root:

$$A[\text{Parent}(i)] \geq A[i]$$

min-heap: For every node i other than the root:

$$A[\text{Parent}(i)] \leq A[i]$$

Max-Heap

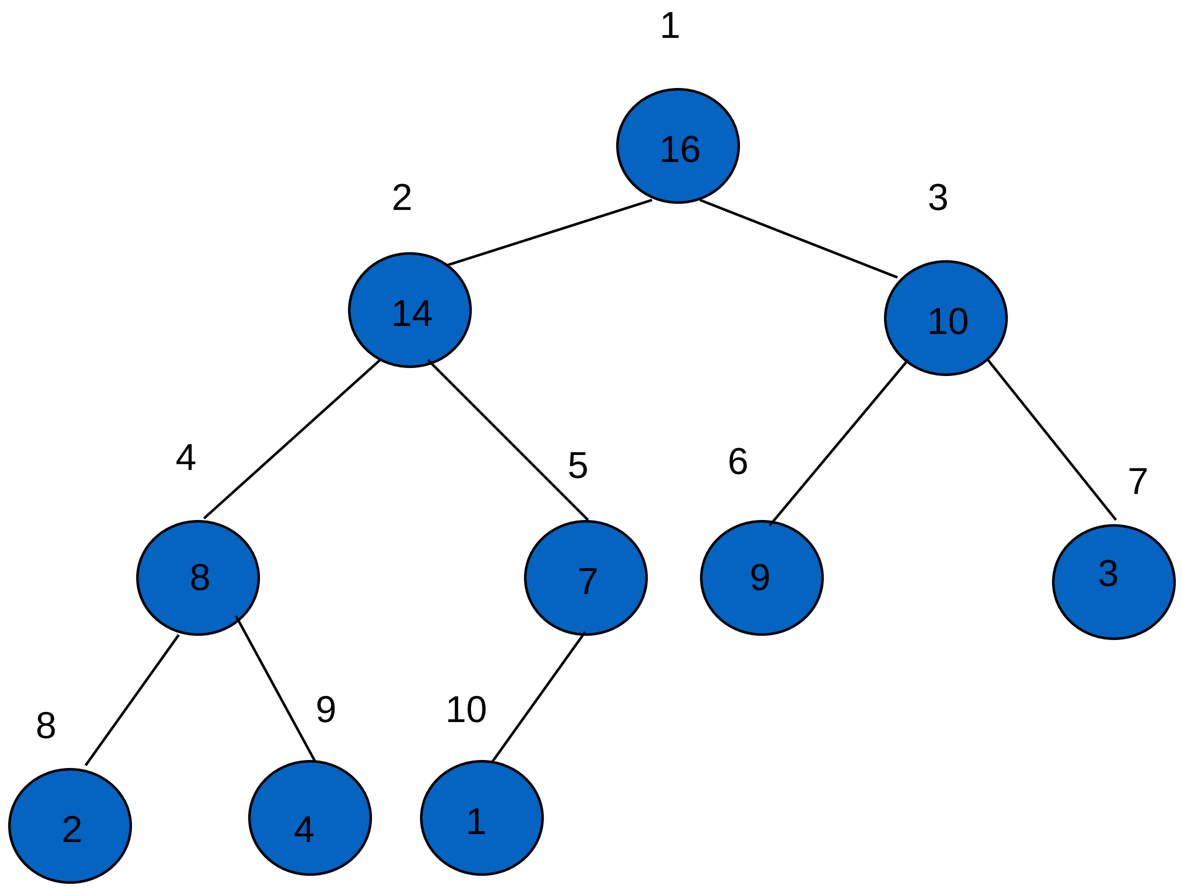
A **max-heap** is an *almost complete binary tree* that satisfies the **heap property**:

For every node i other than the root,

$$A[\mathrm{PARENT}(i)] \geq A[i]$$

What does this mean?

- the value of a node is at most the value of its parent
- the largest element in the heap is stored in the root
- subtrees rooted at a node contain smaller values than the node itself



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

Height of a node in a heap

The ***height*** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf.

The height of a heap is the height of its root.

Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$.

Heaps have 5 basic procedures

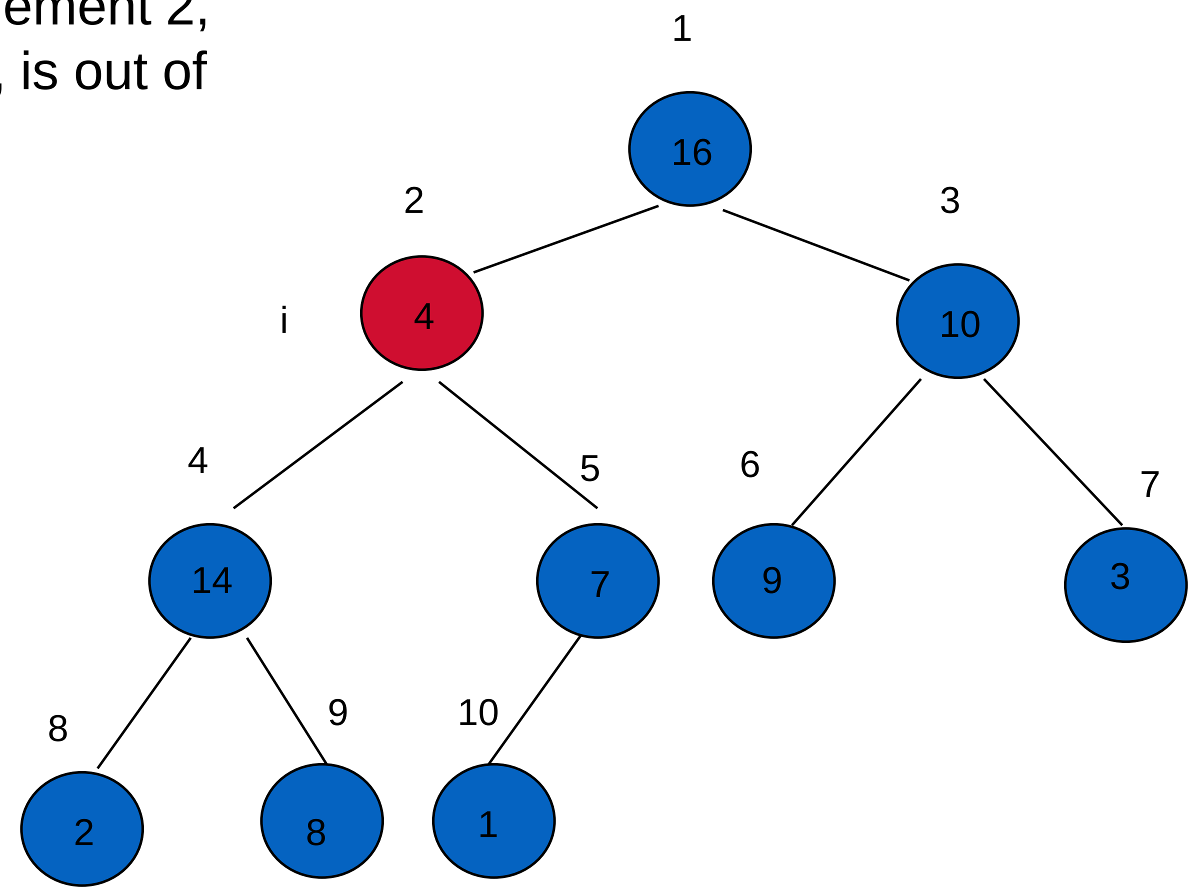
- HEAPIFY: maintains the heap property
- BUILD-HEAP: builds a heap from an unordered array
- HEAPSORT: sorts an array in place
- EXTRACT-MAX: selects max element
- INSERT: inserts a new element

We will work with MAX heaps

MAX-HEAPIFY(A,i)

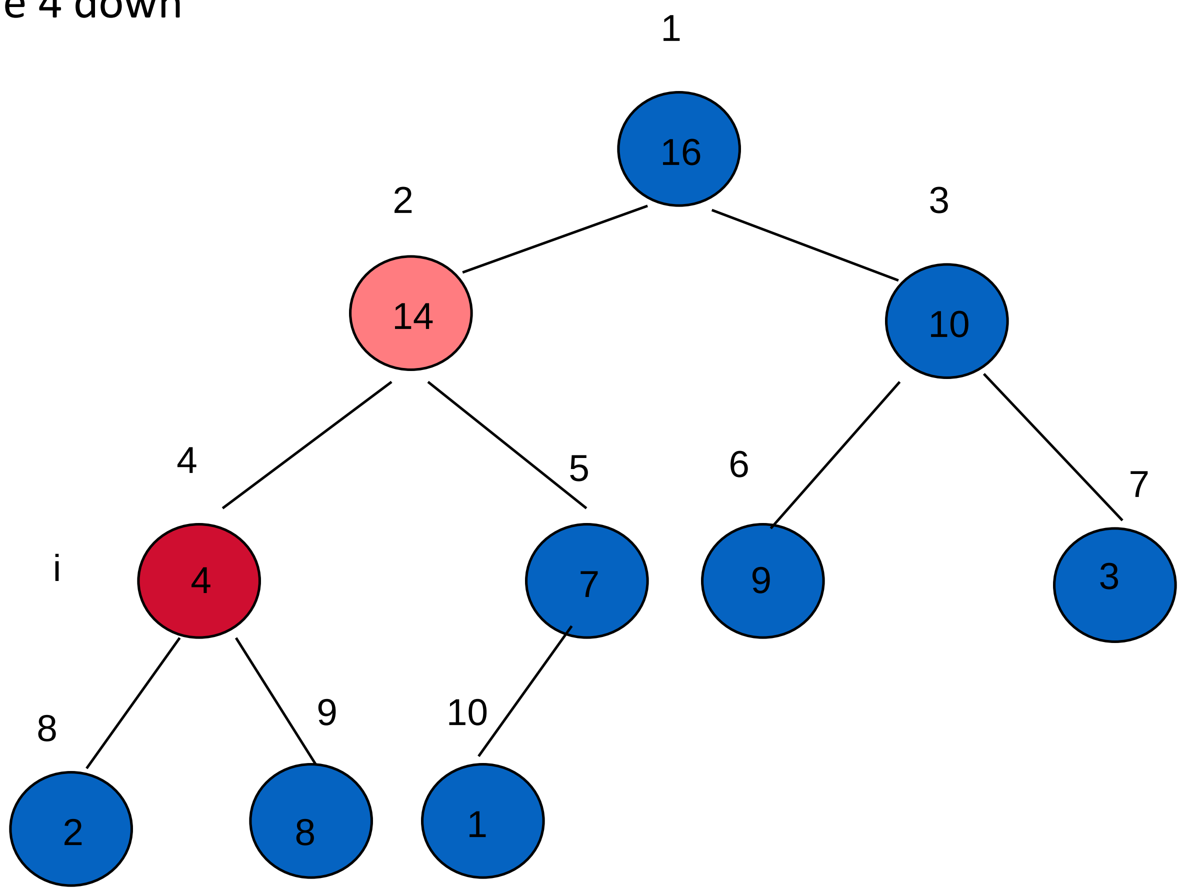
- Goal is to put the i^{th} element in the correct place in a portion of the array that “almost” has the heap property.
- The only element with index of *i* or greater that is out of place is A[*i*].
- Assume that left and right subtrees of A[*i*] have the heap property.
- “Sift” A[*i*] down to the right position.

Array element 2, the “4”, is out of place



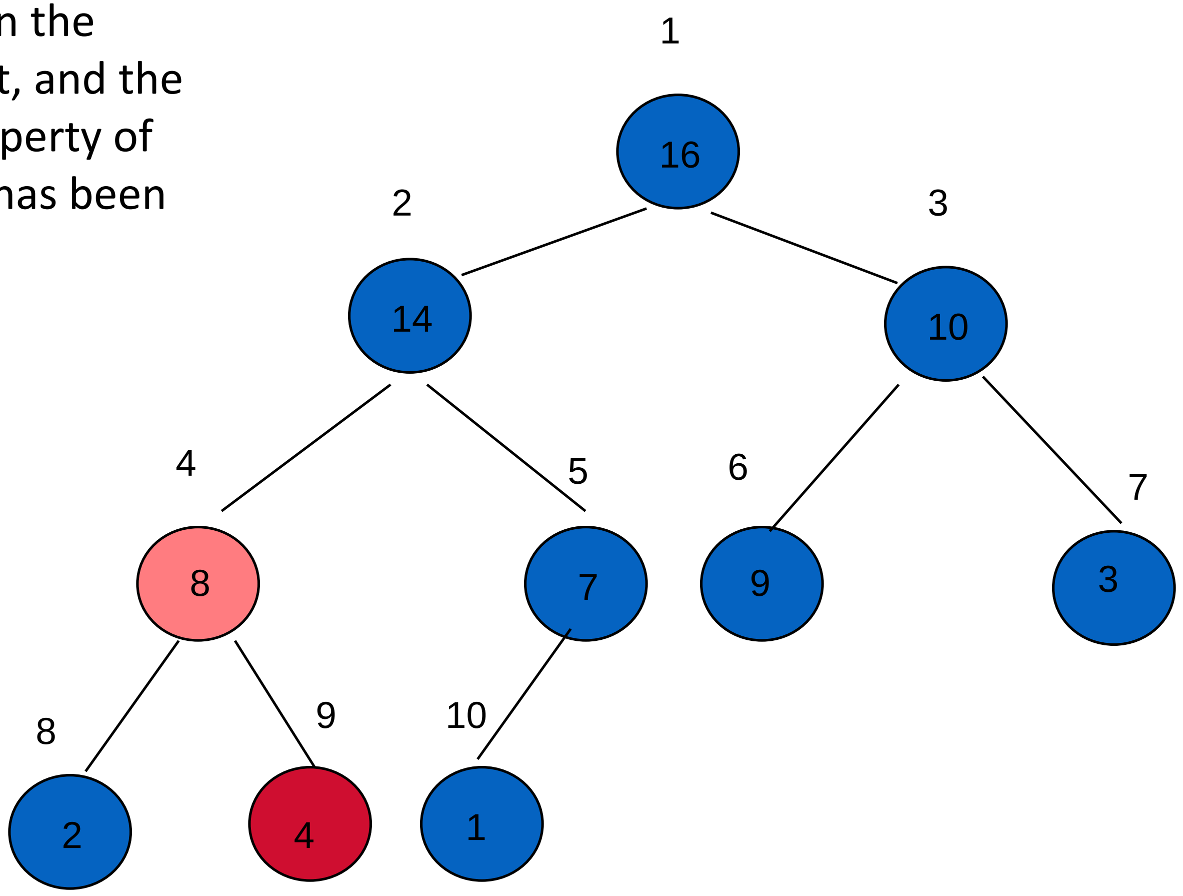
MAX-HEAPIFY(A,2)   heap-size[A] = 10

Moving the 4 down



MAX-HEAPIFY(A,4) heap-size[A] = 10

The 4 is in the right spot, and the heap property of the tree has been restored.



MAX-HEAPIFY(A,9) heap-size[A] = 10

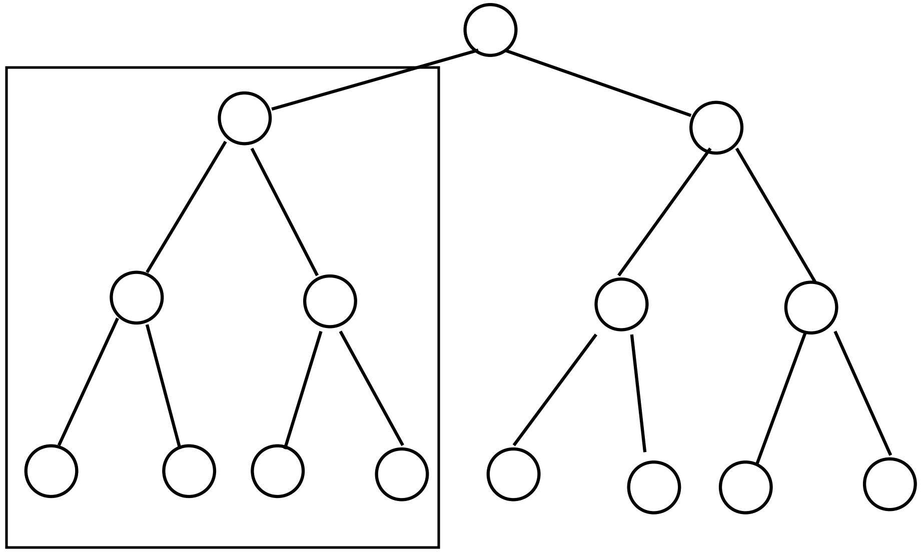
MAX-HEAPIFY

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

Running time of MAX-HEAPIFY

How many nodes might be involved?

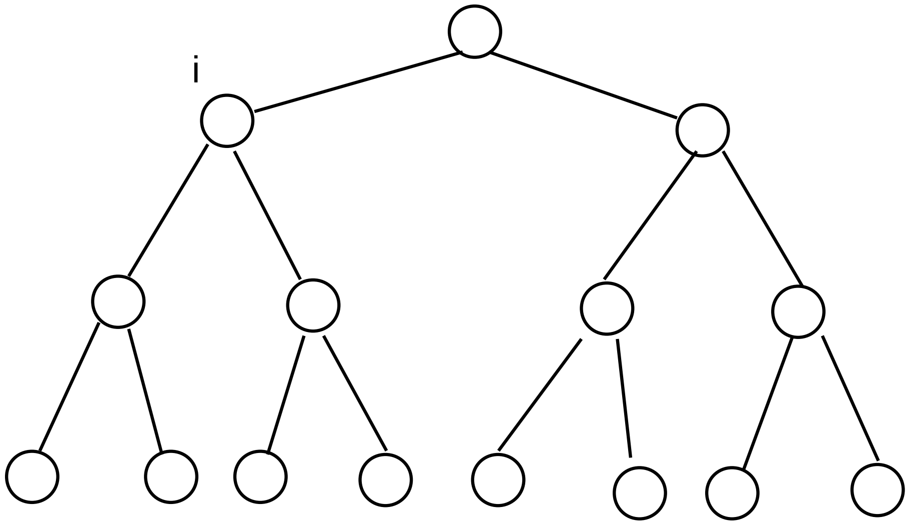
In the case of a full binary tree, about half of the tree might be involved.



Running time of MAX-HEAPIFY

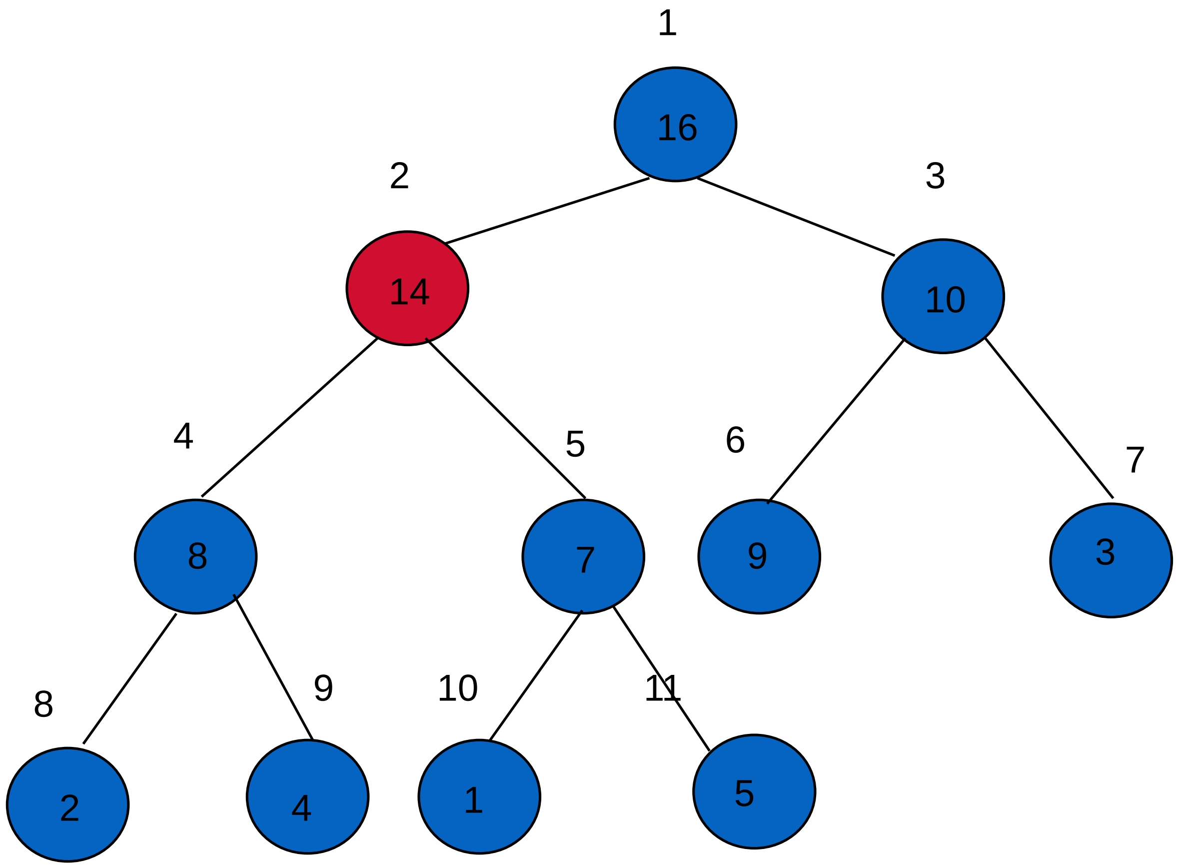
In a complete binary tree with 15 nodes, 8 of those nodes are leaves at the bottom level.

If we perform MAX-HEAPIFY on node i, 7 of the 15 nodes will be involved – about ½ of the nodes.



Running time of MAX-HEAPIFY

What is the worst case?
When the last row of the tree is half full.



Here 7 out of 11 nodes are involved.

In general, $\leq 2/3^{rds}$ of the tree might be involved in the worst case.

Running time of MAX-HEAPIFY

Remember that, in a complete binary tree, *more than half* of the nodes in the entire tree are the leaf nodes on the bottom level of the tree.

But the only nodes involved in MAX-HEAPIFY are the descendants of A[i], which must be in A[i]'s half of the tree.

So worst case is when the last row of the tree is half full on the left side and A[i] is their ancestor.

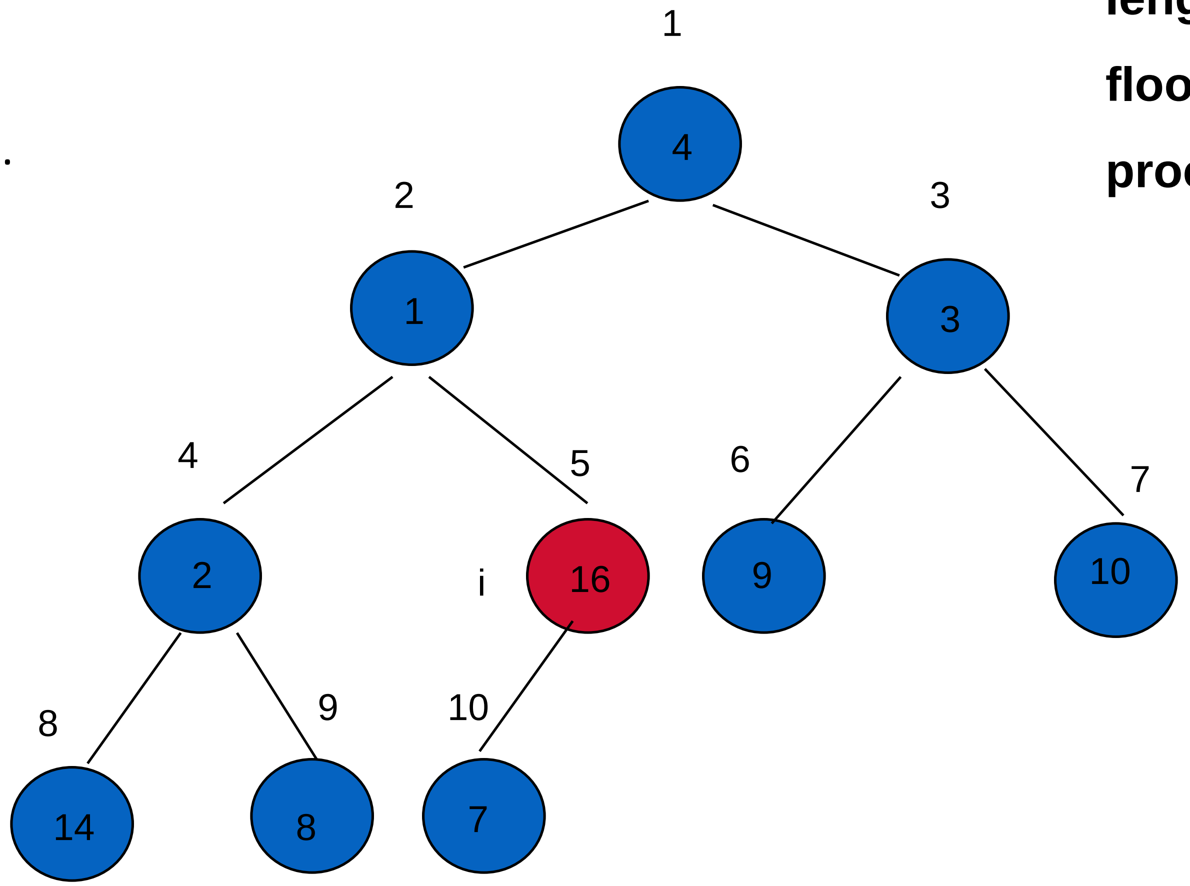
BUILD-MAX-HEAP

- Use MAX-HEAPIFY in a bottom-up manner to convert an array A[1..n] into a heap.
- Each leaf is initially a one-element heap. Elements A[$\lfloor n/2 \rfloor + 1..n$] are leaves.
- MAX-HEAPIFY is called on all interior nodes.

BUILD-MAX-HEAP

```
BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← floor(length[A]/2) downto 1 do
    MAX-HEAPIFY(A, i)
```

a.



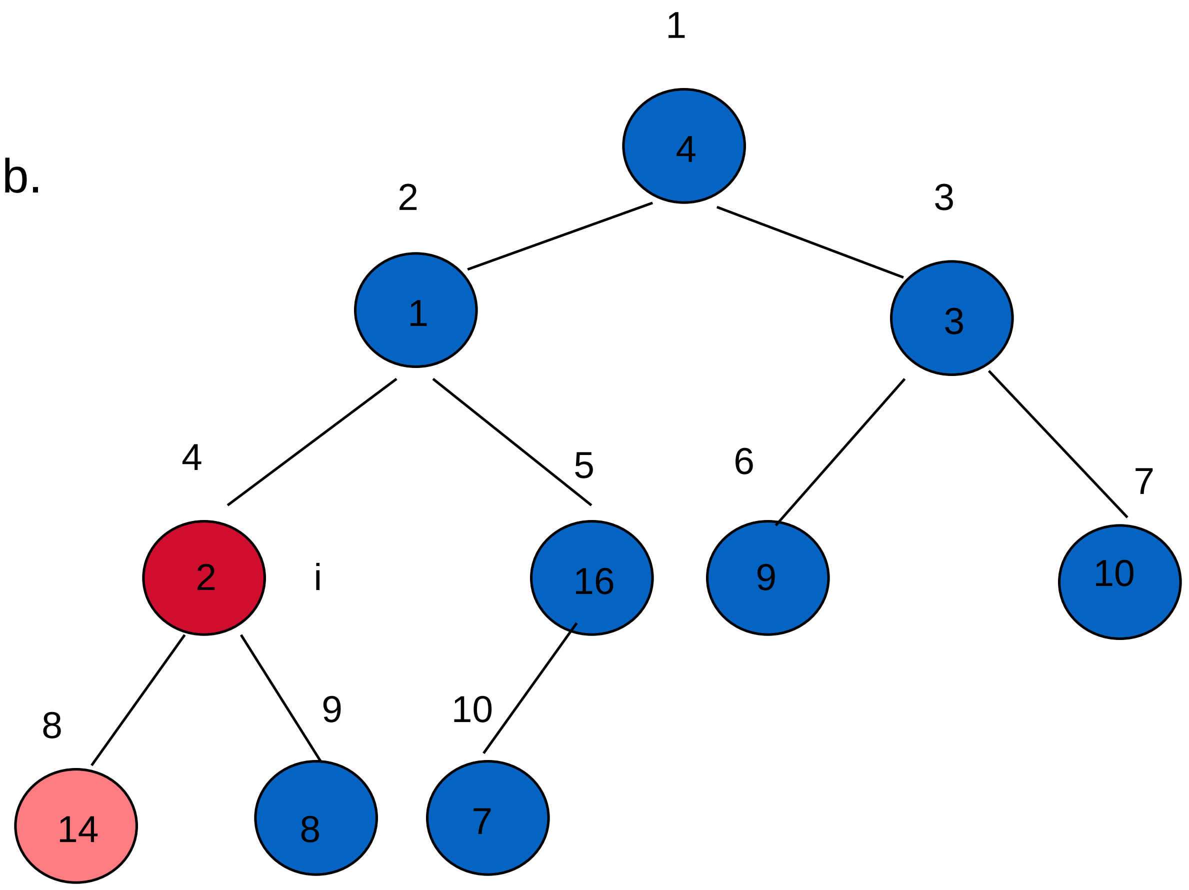
length(A) = 10

floor(length(A)/2) = 5

process from 5 to 1

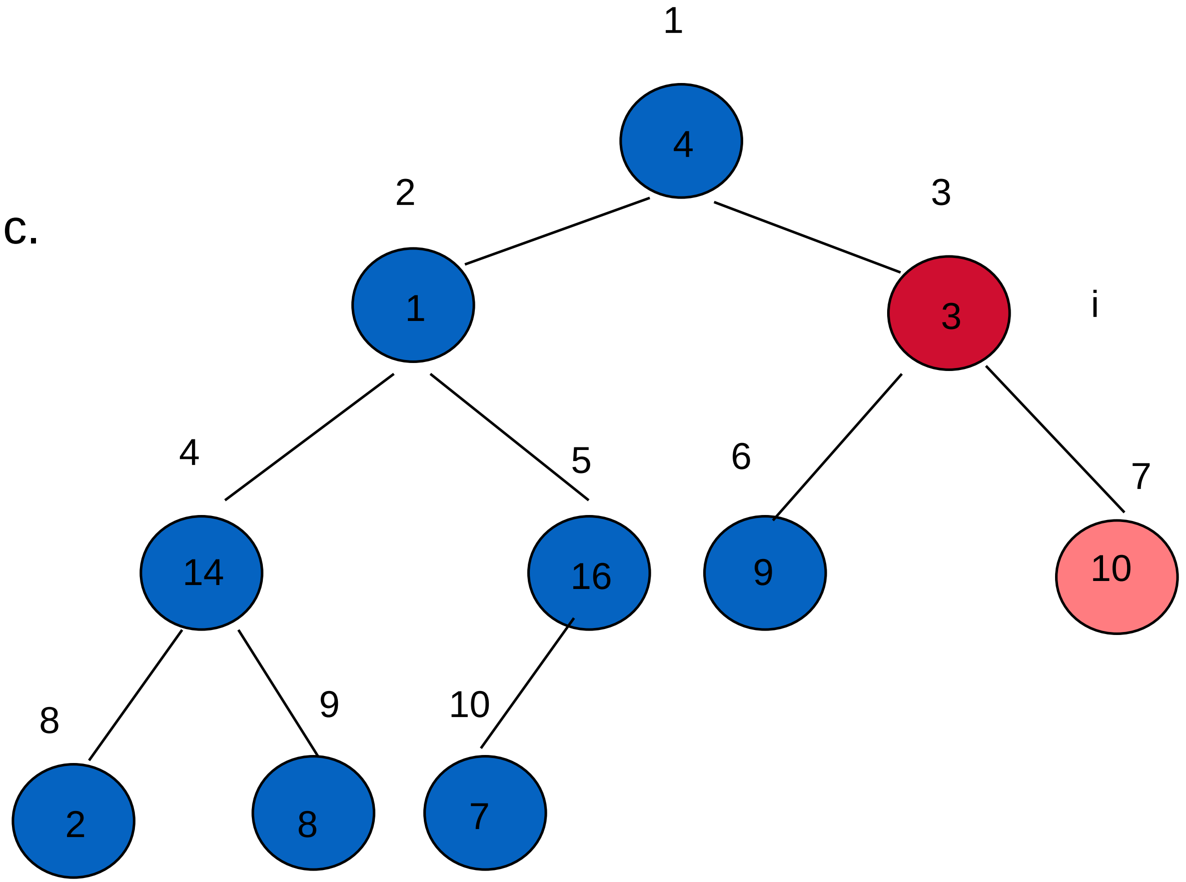
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

b.



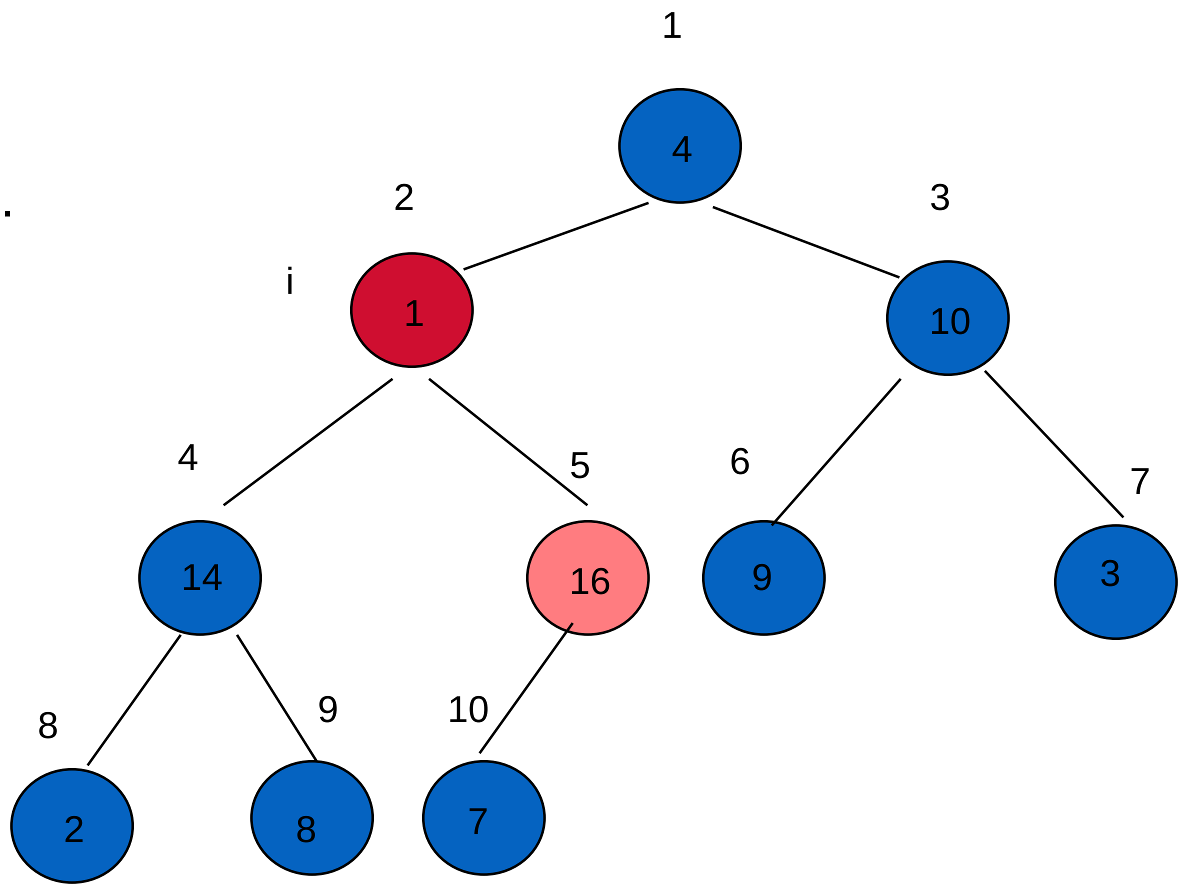
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

c.



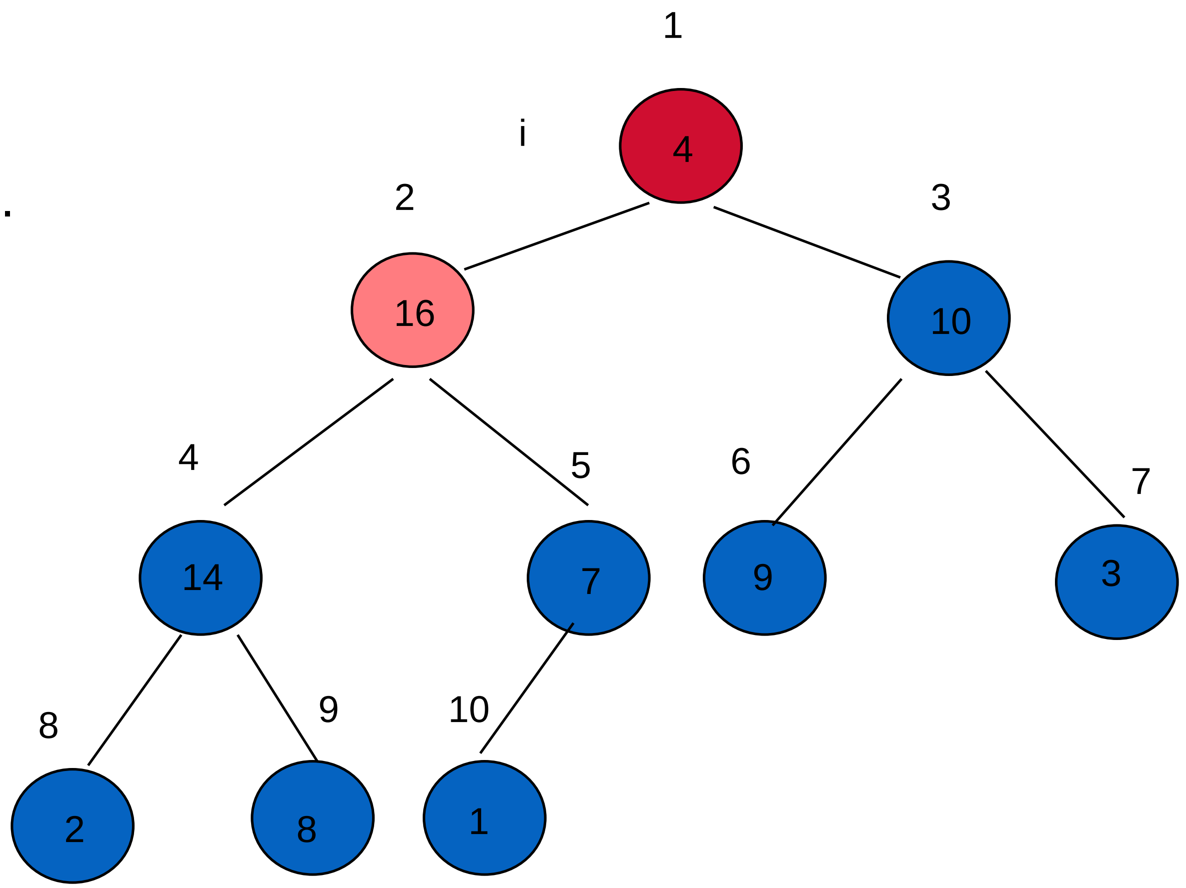
4	1	3	14	16	9	10	2	8	7
1	2	3	4	5	6	7	8	9	10

d.



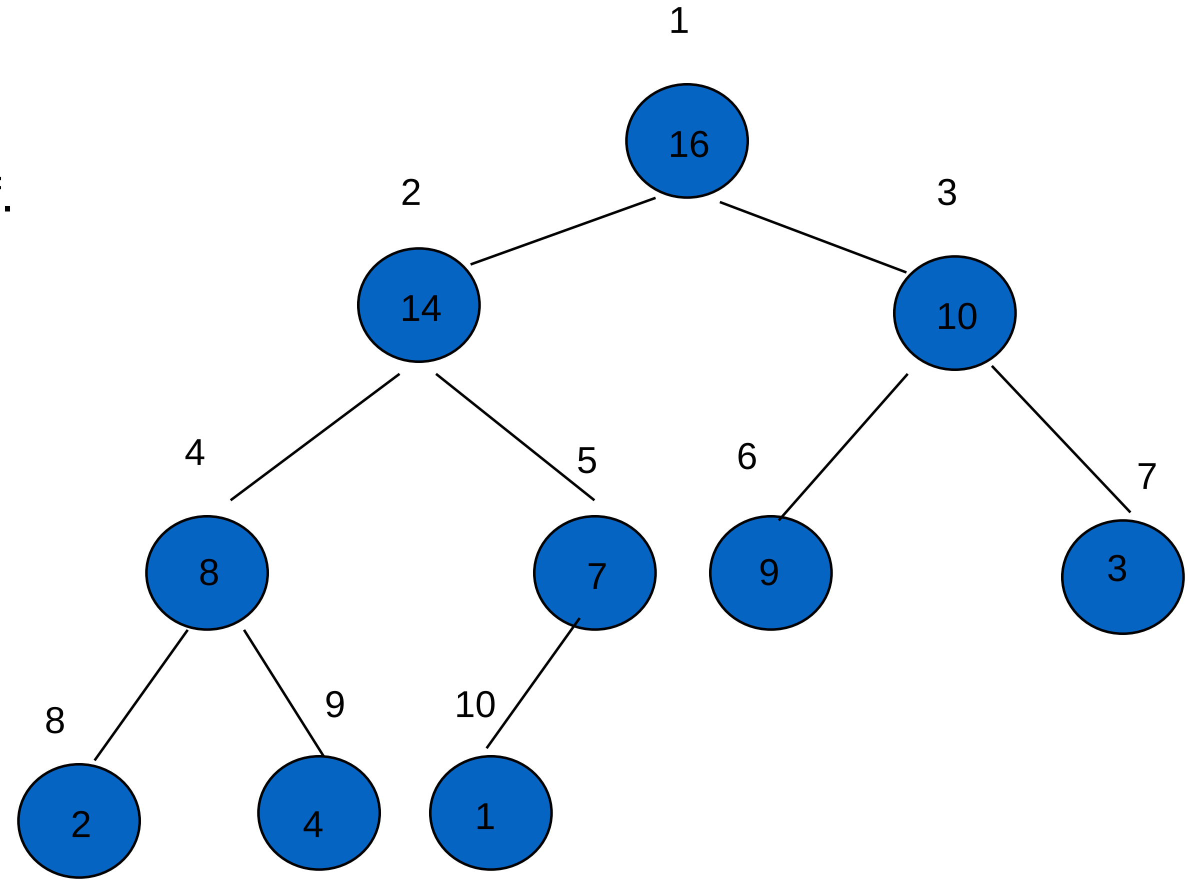
4	1	10	14	16	9	3	2	8	7
1	2	3	4	5	6	7	8	9	10

e.



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

f.



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

Running Time of BUILD-MAX-HEAP

- Simple upper bound:
  - each call to MAX-HEAPIFY costs O(lg n)
  - O(n) such calls
  - running time at most O(n lg n)
- Previous bound is not tight:
  - lots of the elements are leaves
  - most elements are near leaves (small height)

Tighter Bound for BUILD-MAX-HEAP

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

By substituting x = ½ in the formula for differentiating infinite geometric series, we have:

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

Tighter Bound for BUILD-MAX-HEAP (continued)

Thus the running time is bounded by:

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

Therefore, we can build a heap from an unordered array in linear time.

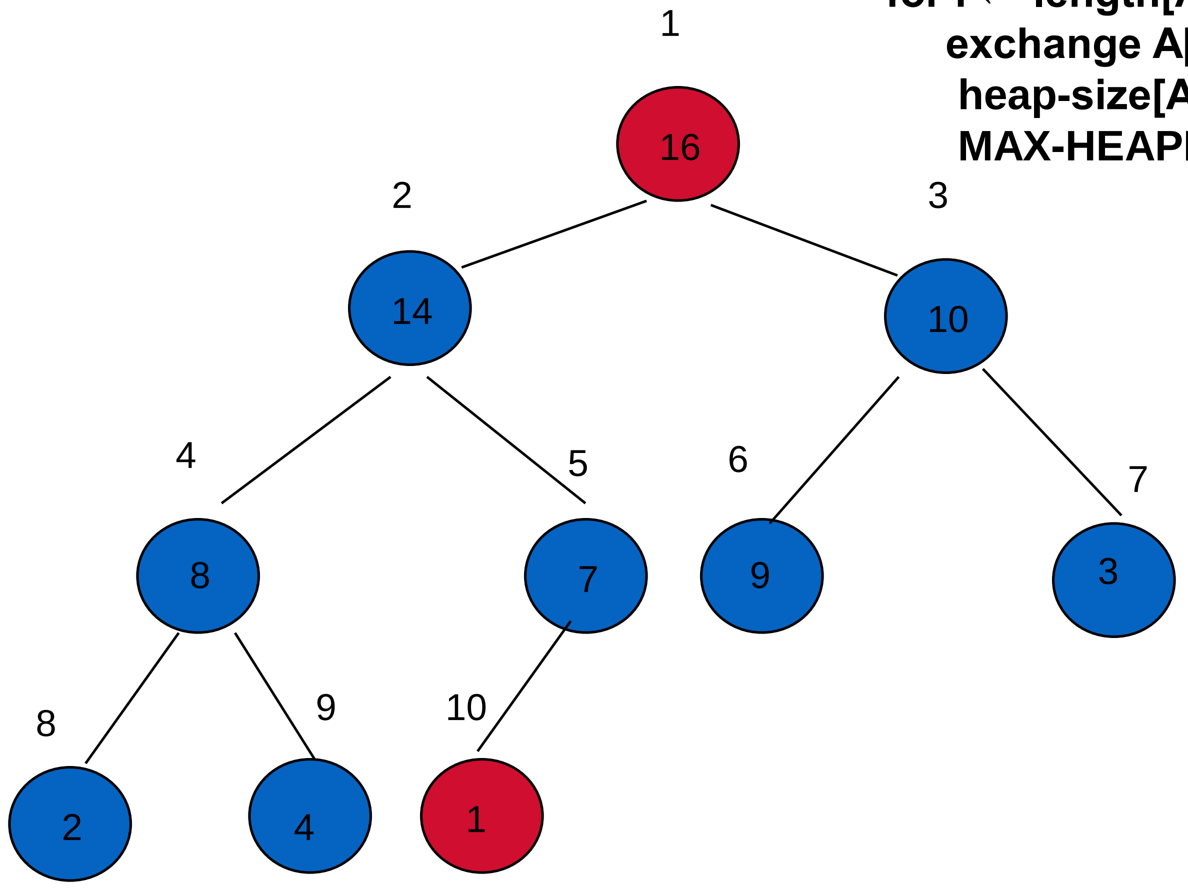
Heapsort

- First build a heap.
- Then successively remove the biggest element from the heap and move it to the first position in the sorted array.
- The element currently in that position is then placed at the top of the heap and sifted to the proper position.

HEAPSORT

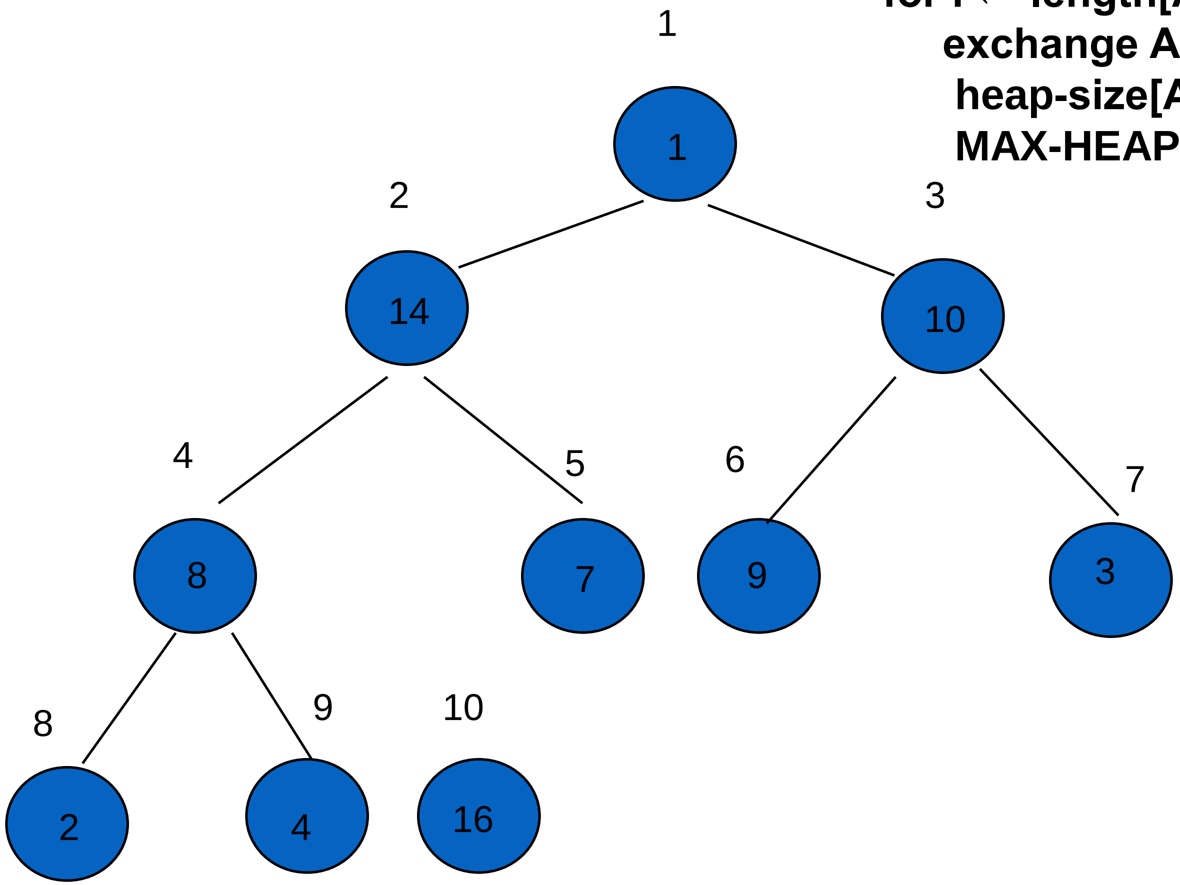
```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
  exchange A[1] ↔ A[i]
  heap-size[A] ← heap-size[A] - 1
  MAX-HEAPIFY(A, 1)
```

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



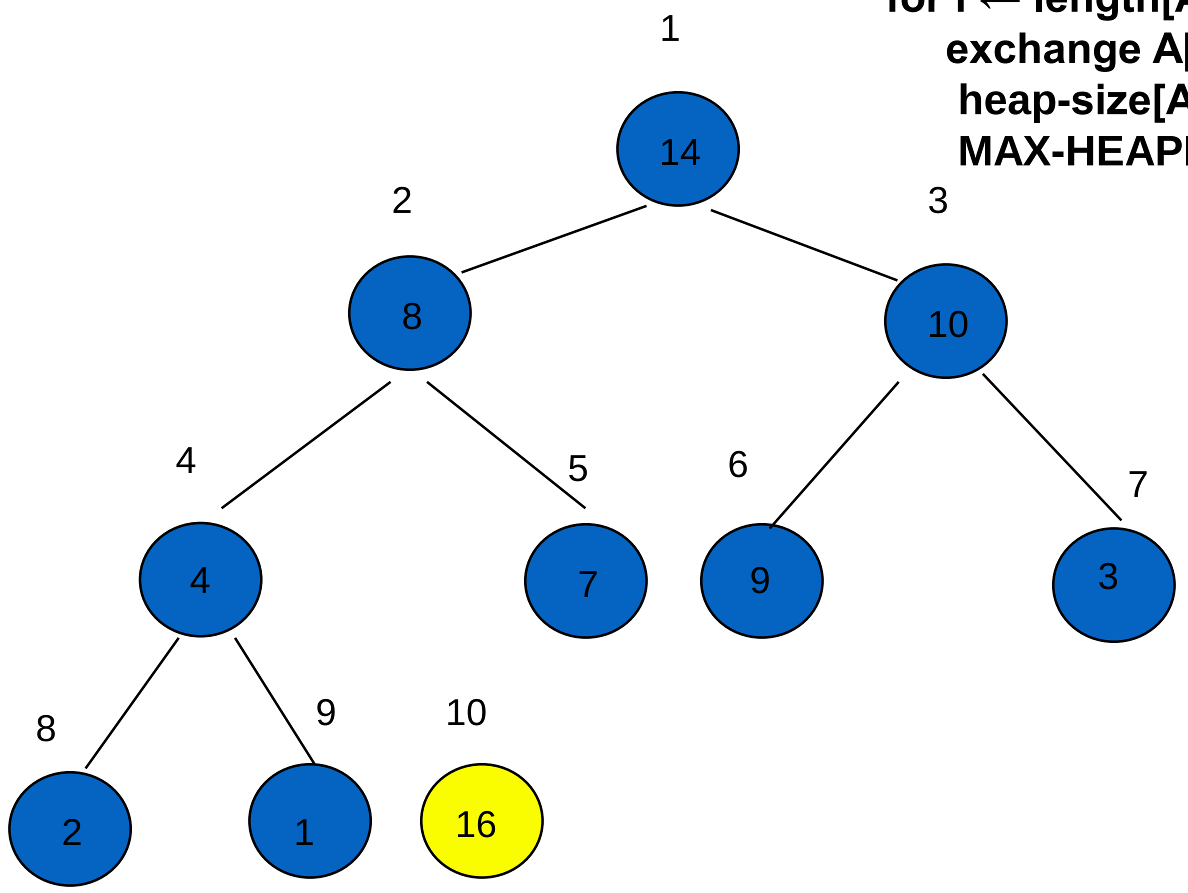
16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



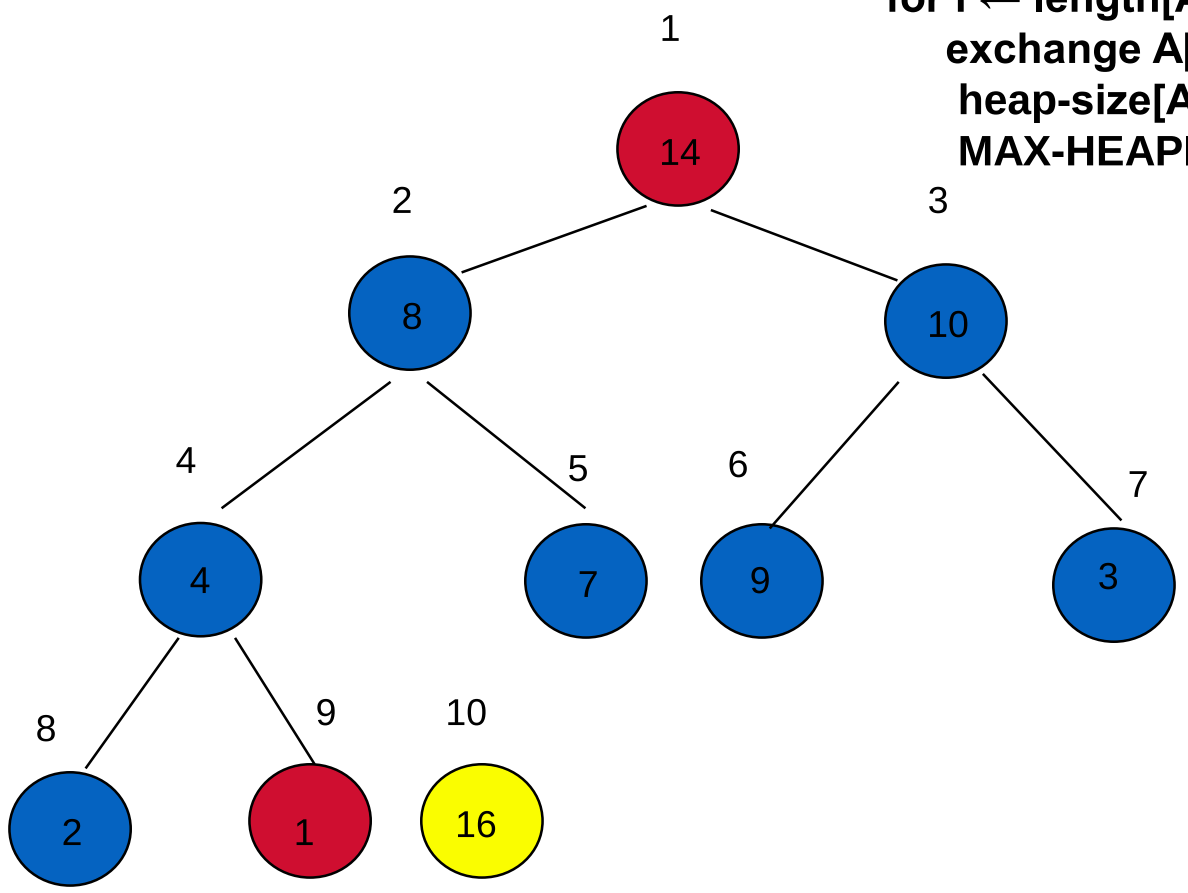
1	14	10	8	7	9	3	2	4	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
  exchange A[1] ↔ A[i]
   heap-size[A] ← heap-size[A] – 1
   MAX-HEAPIFY(A, 1)



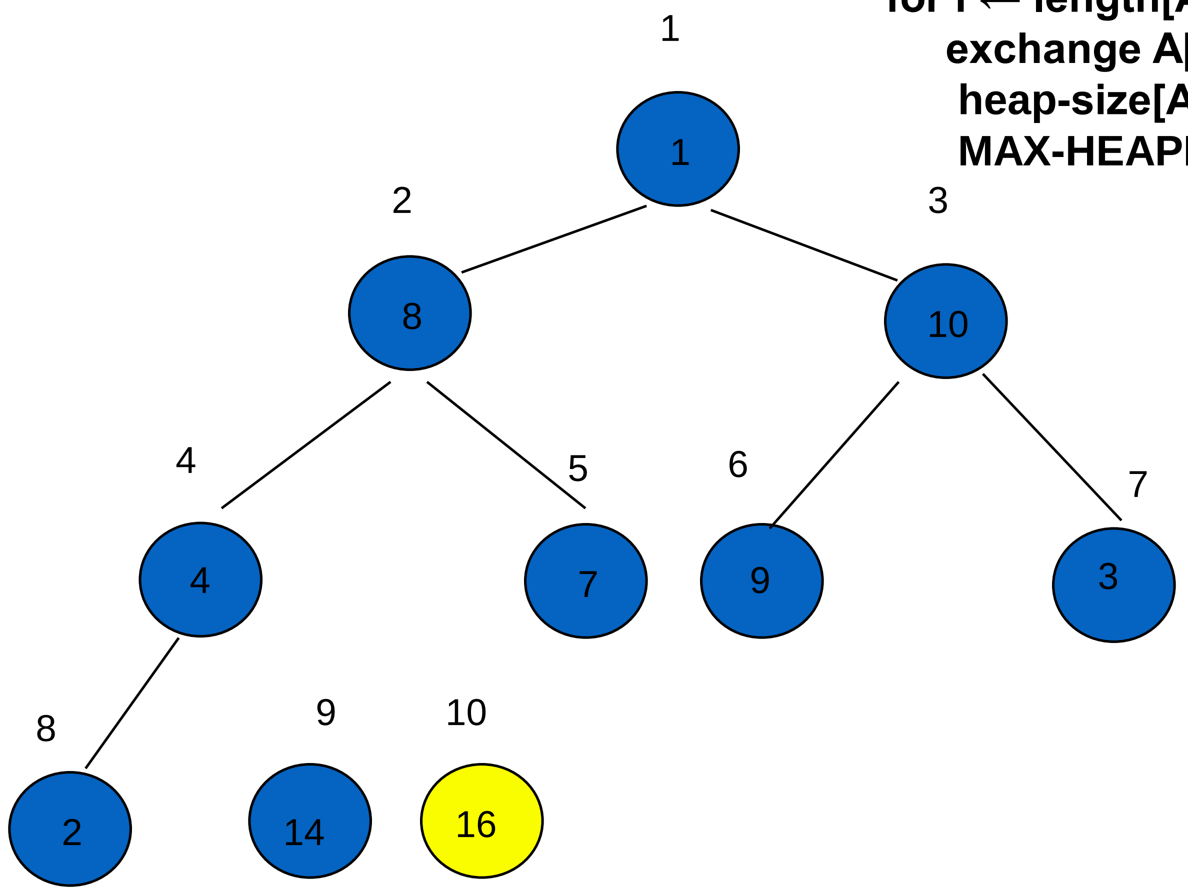
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



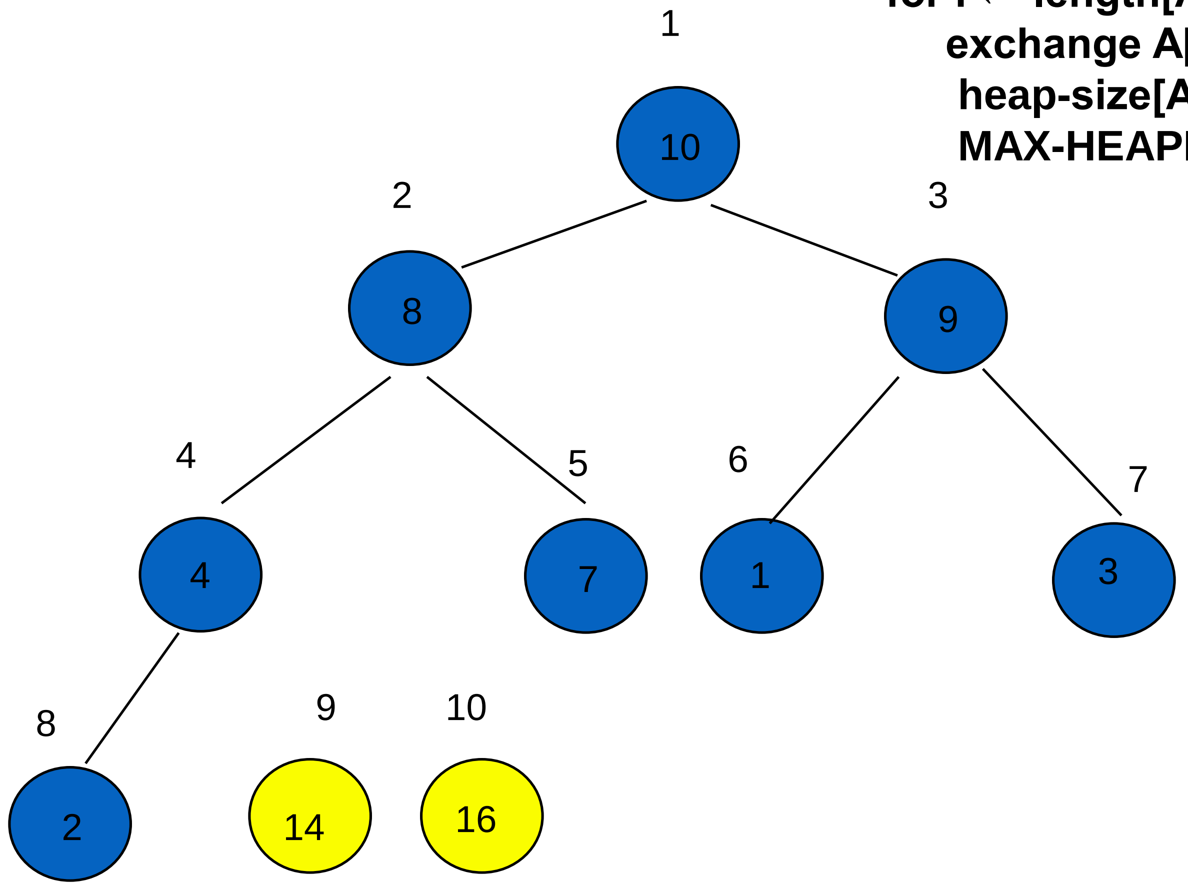
14	8	10	4	7	9	3	2	1	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



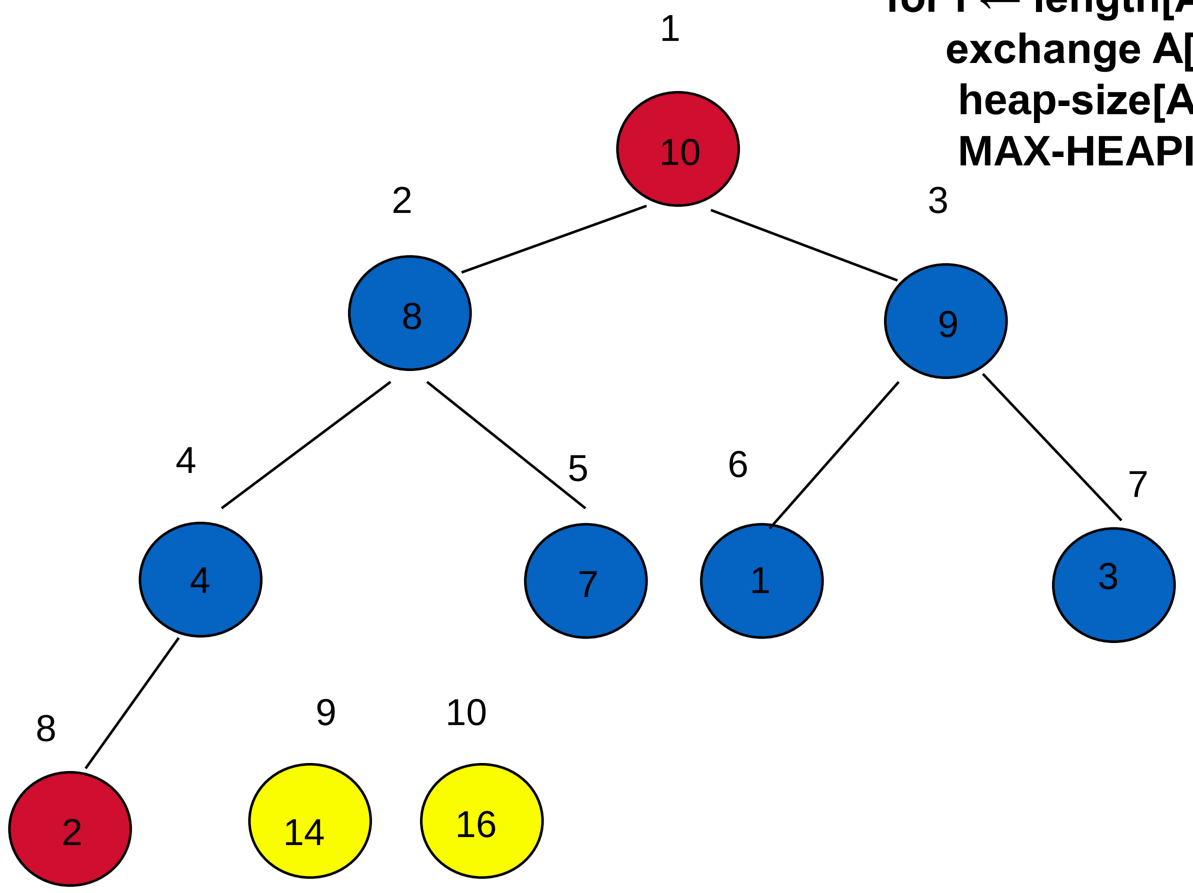
1	8	10	4	7	9	3	2	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



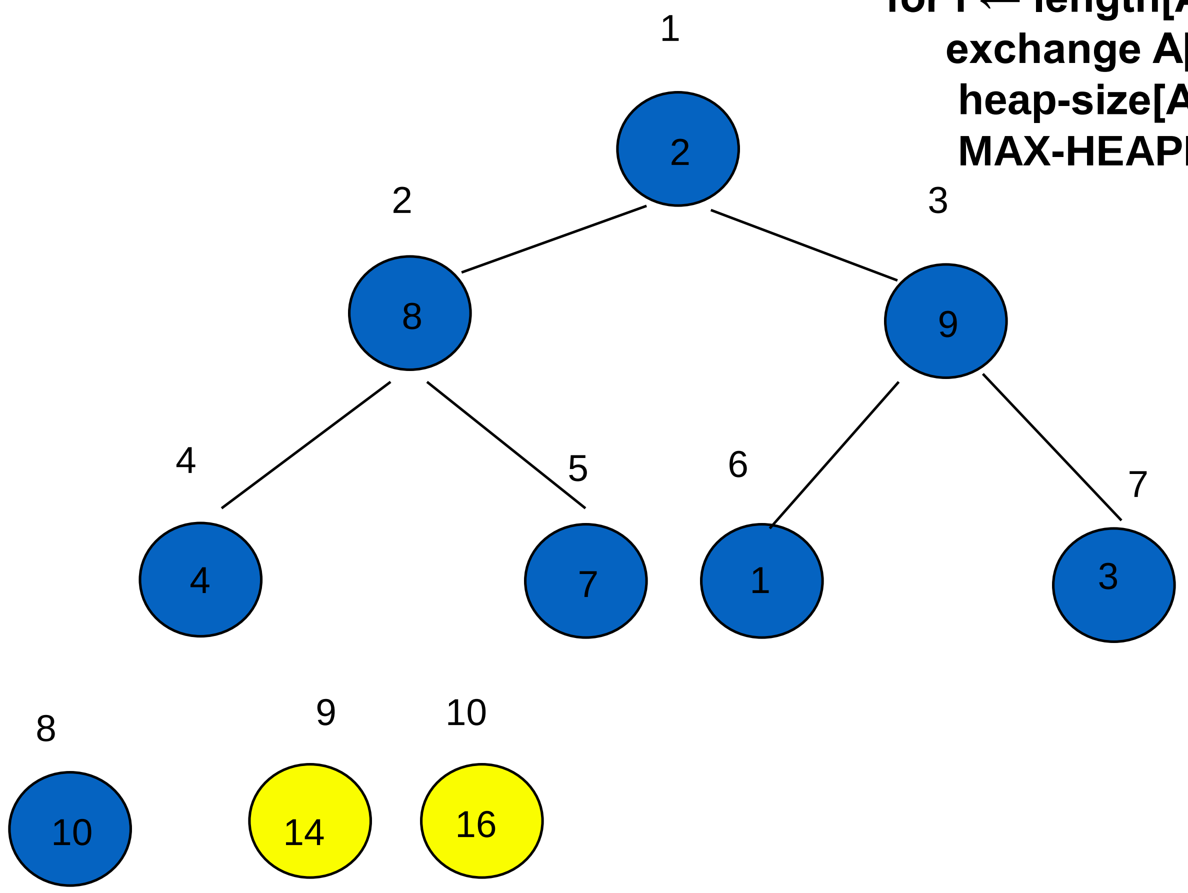
10	8	9	4	7	1	3	2	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



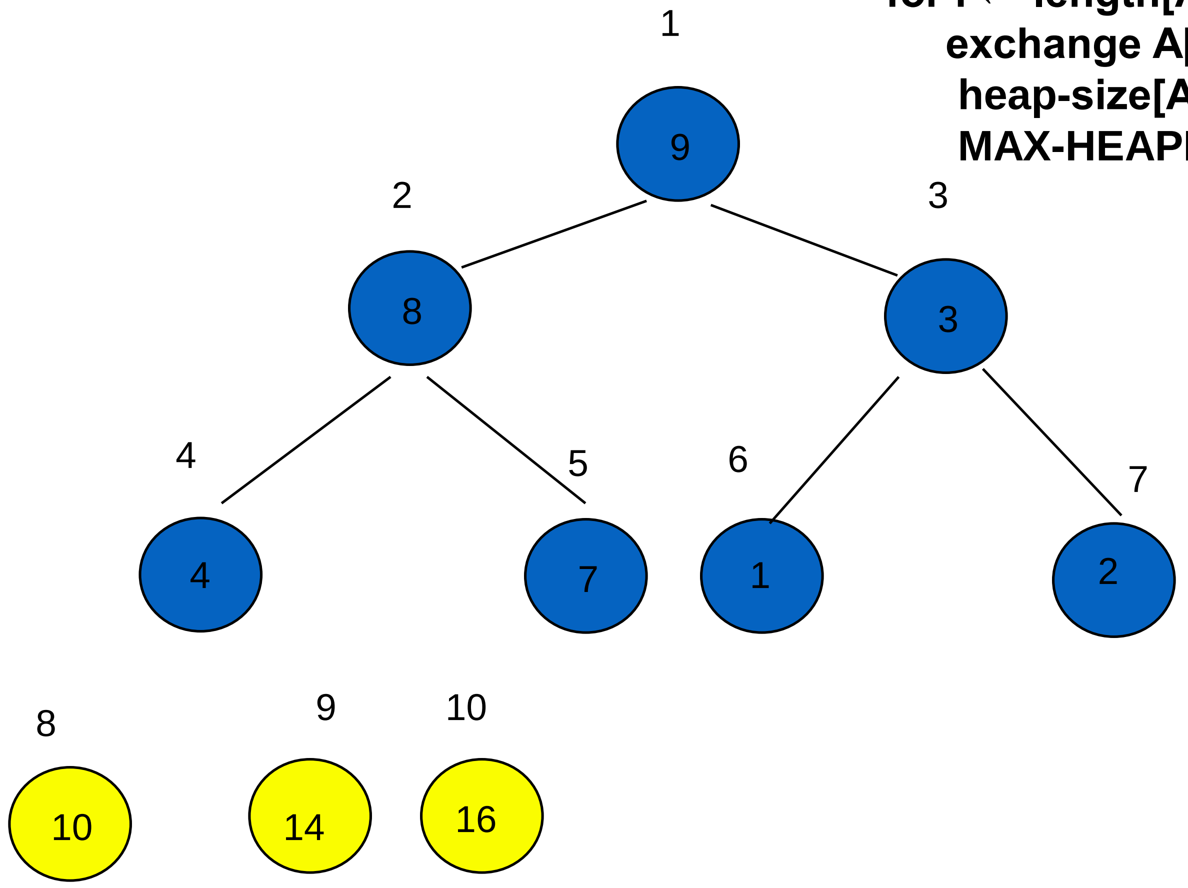
10	8	9	4	7	1	3	2	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



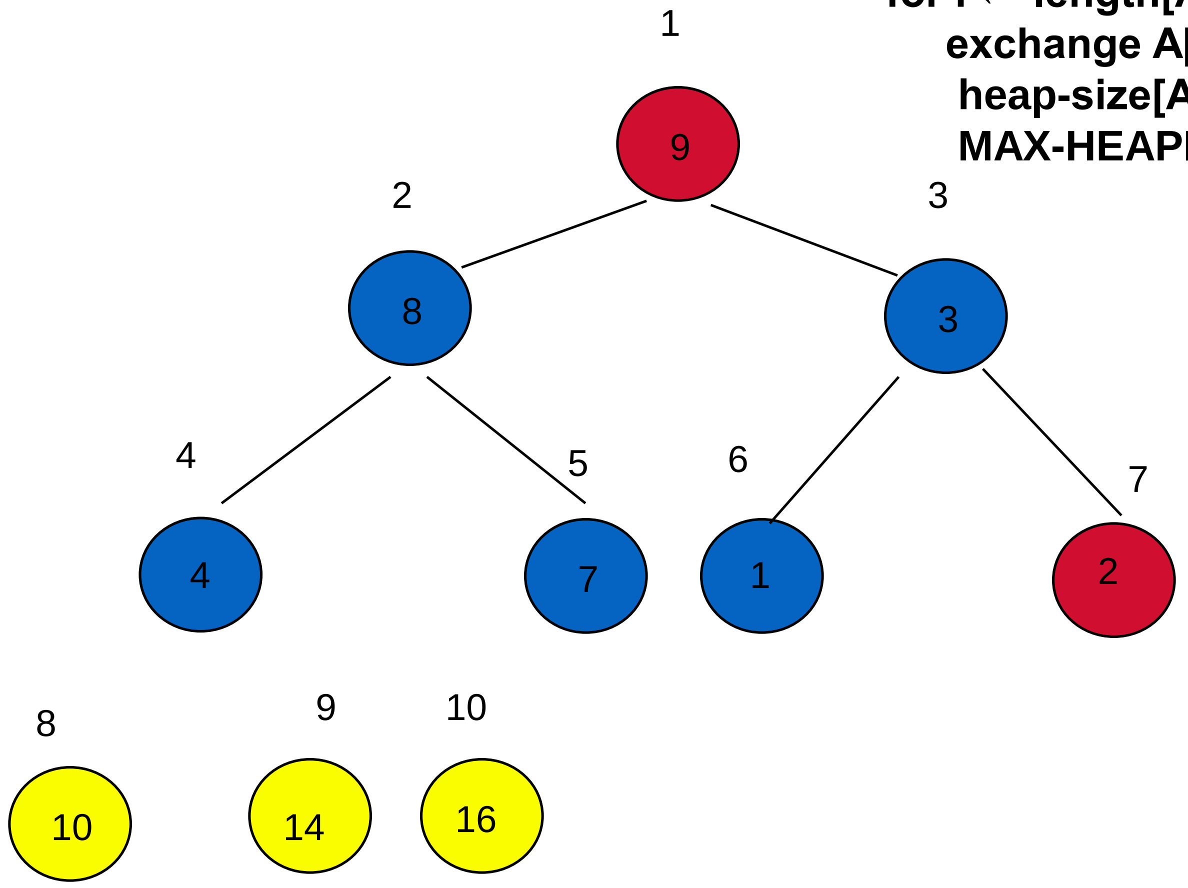
2	8	9	4	7	1	3	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



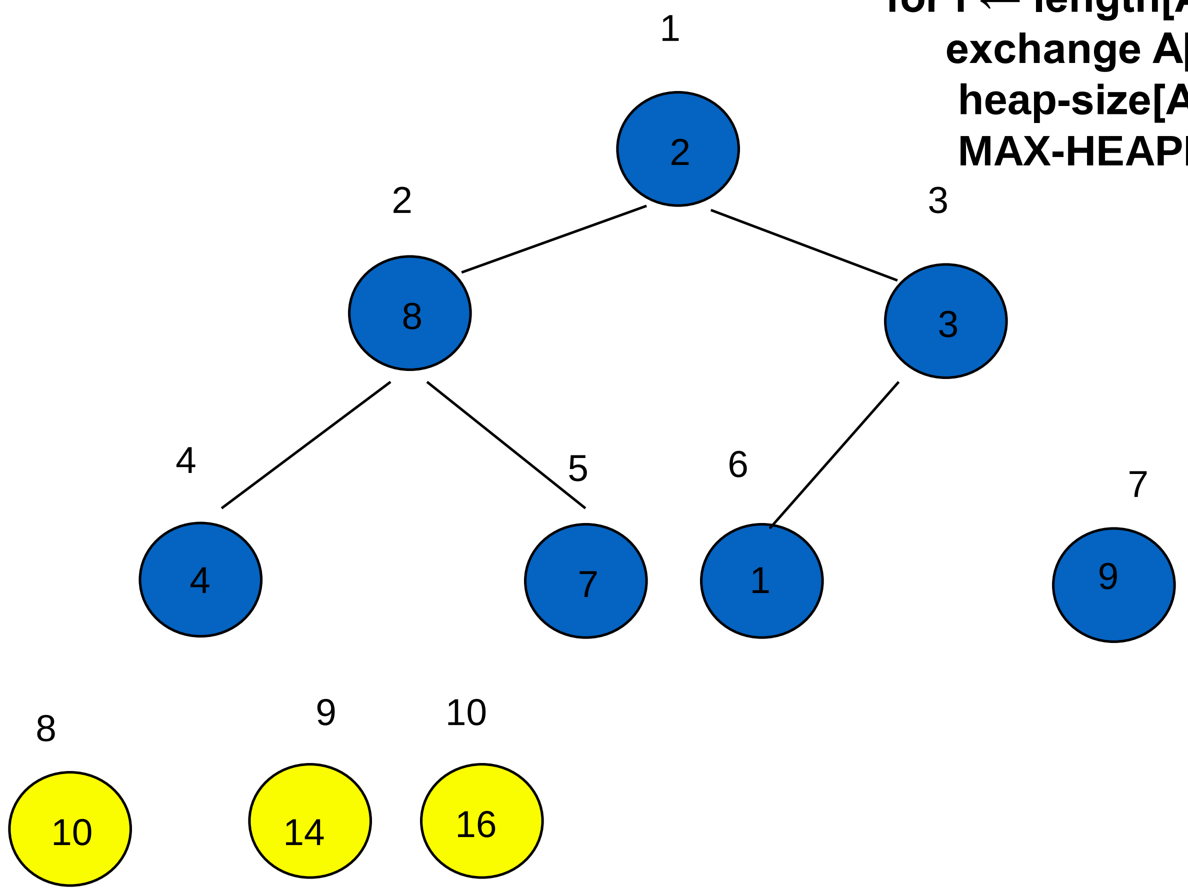
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



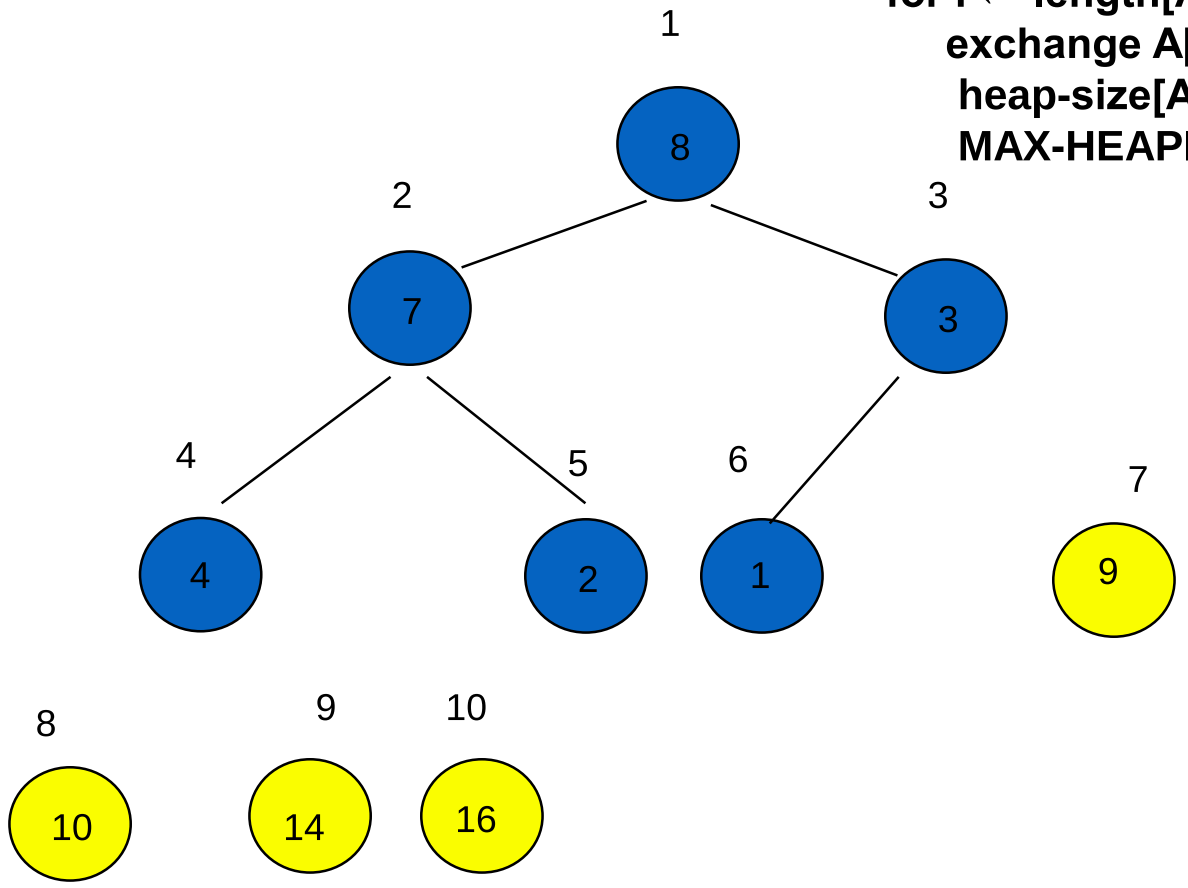
9	8	3	4	7	1	2	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



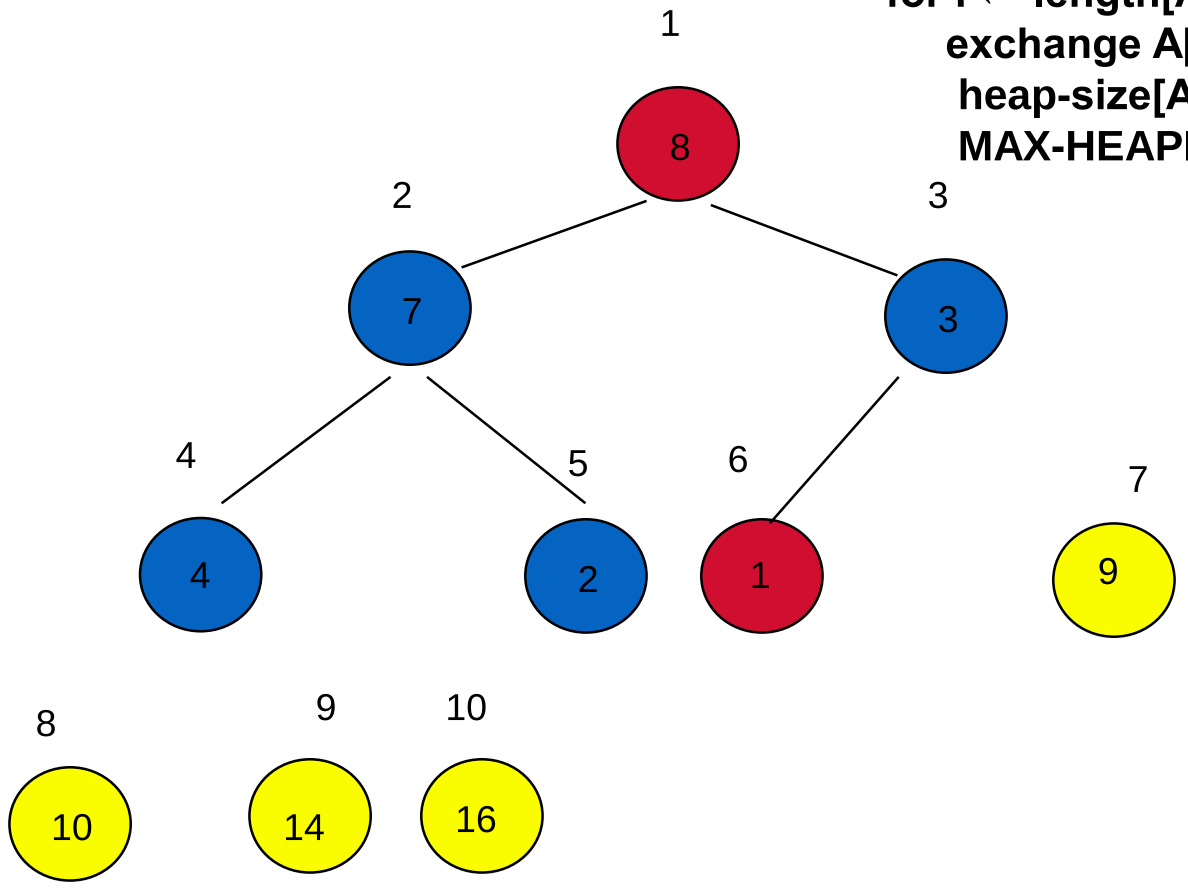
2	8	3	4	7	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



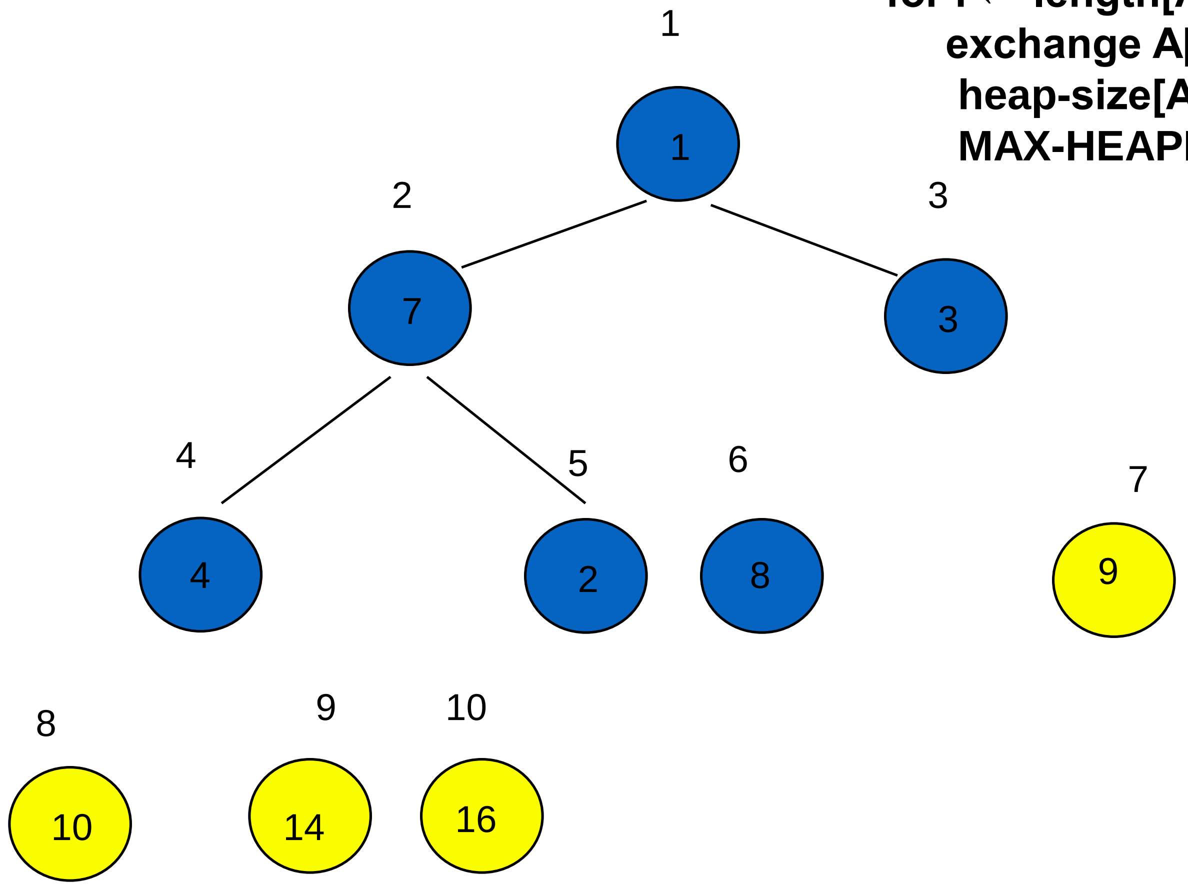
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



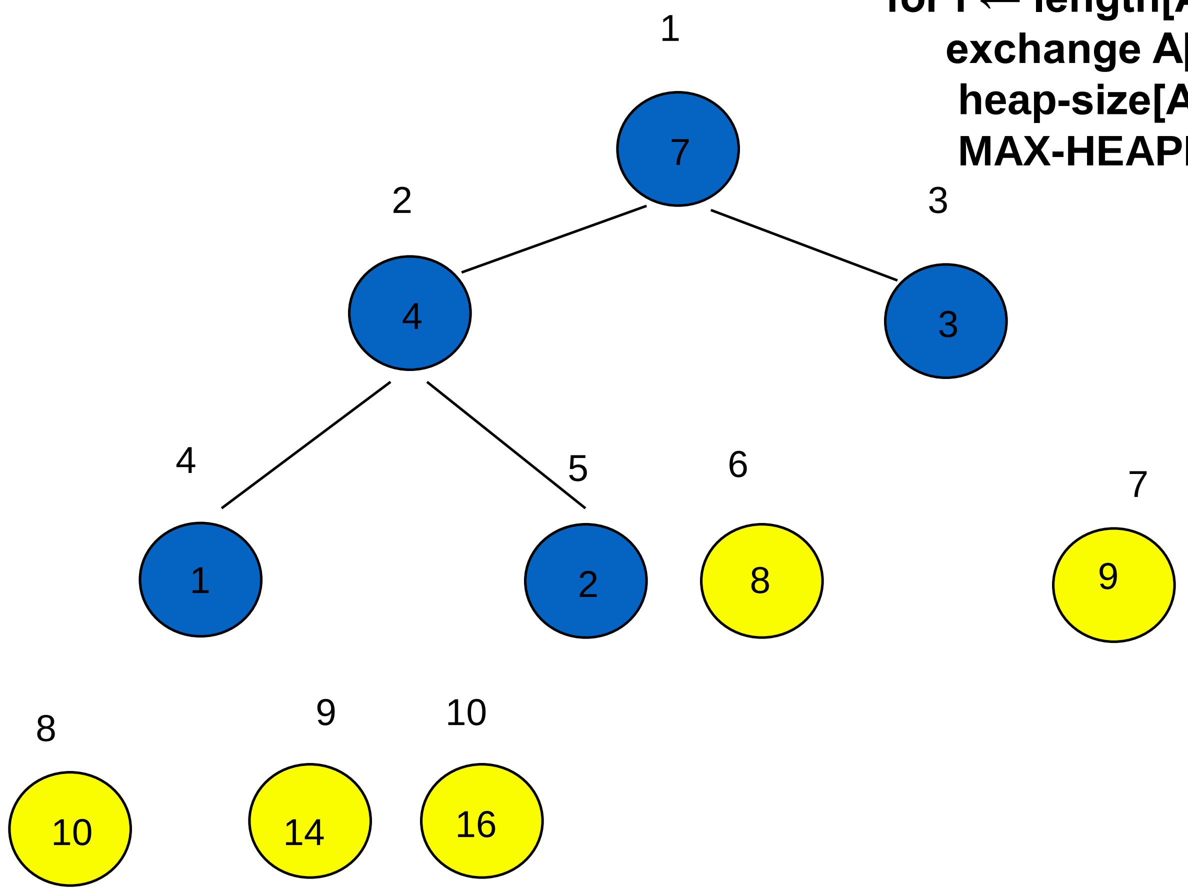
8	7	3	4	2	1	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



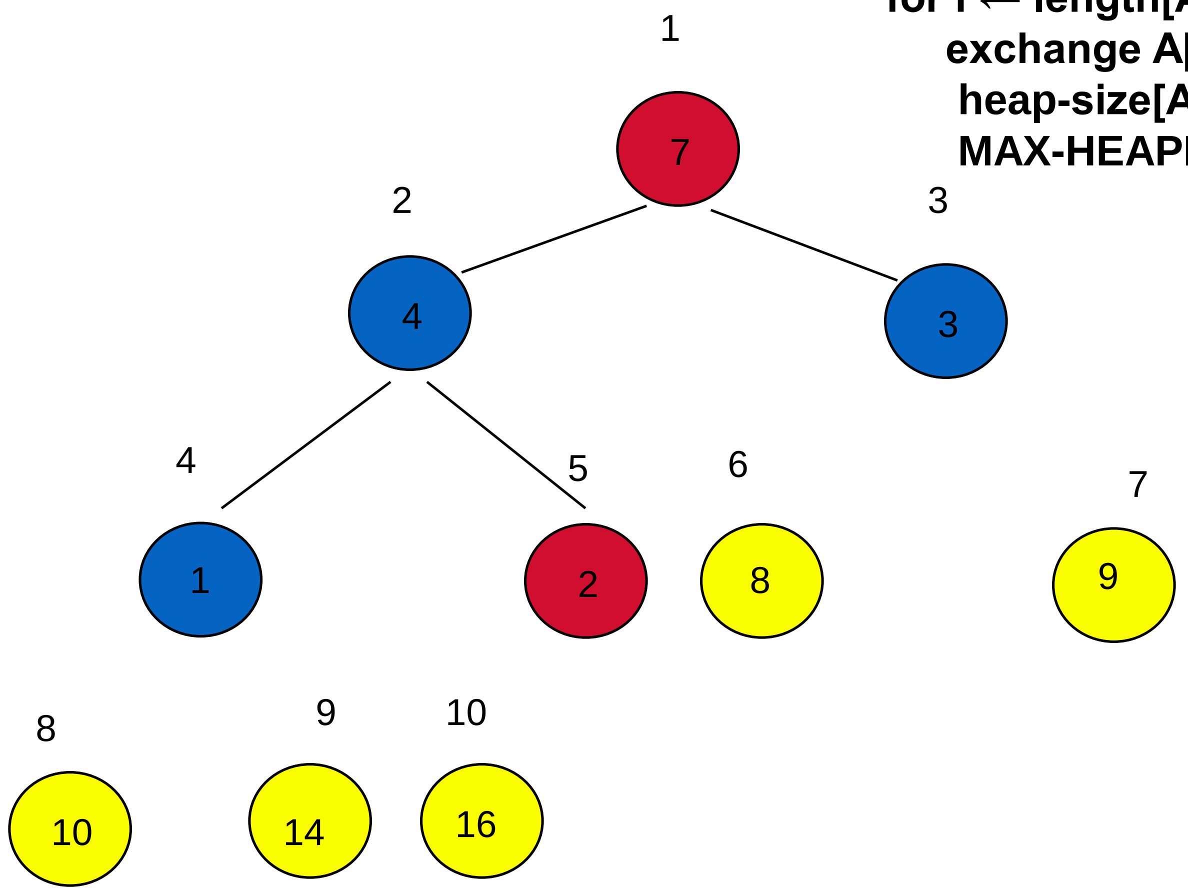
1	7	3	4	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



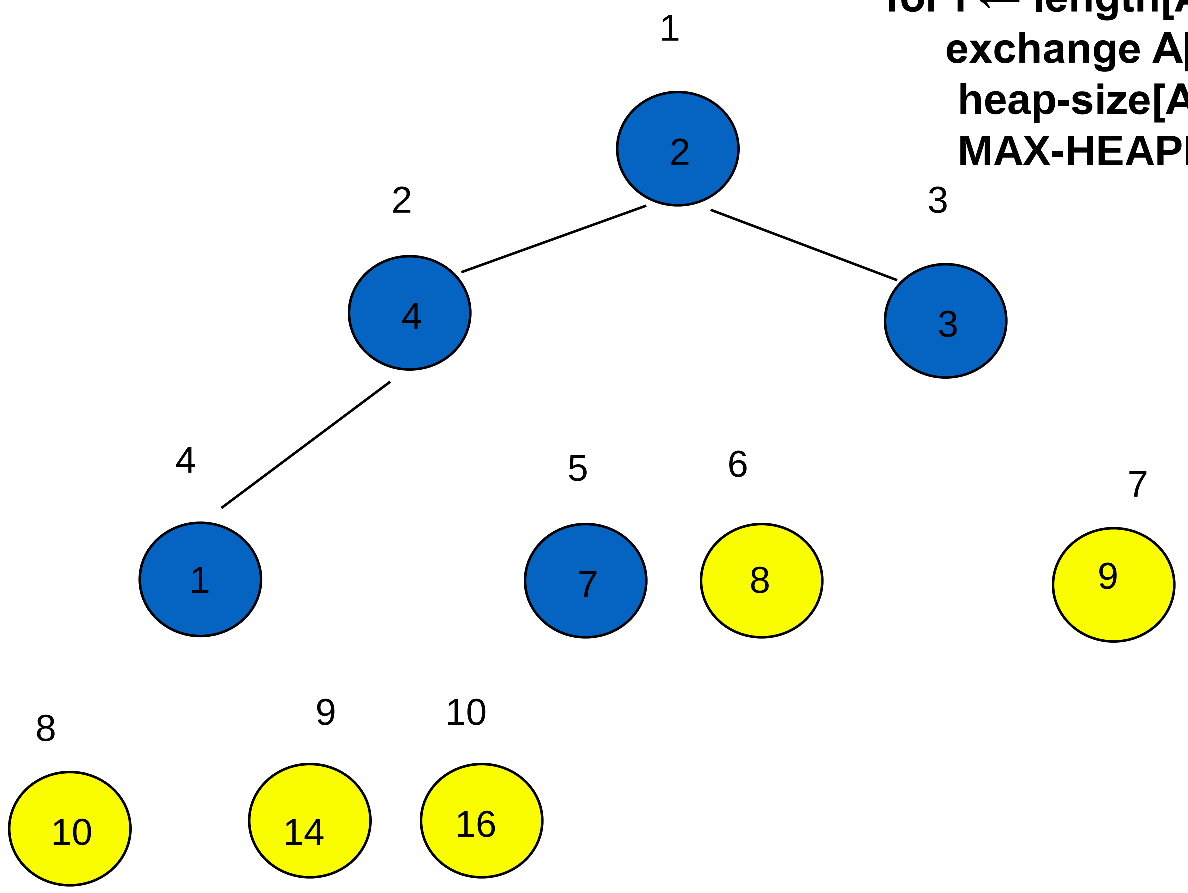
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



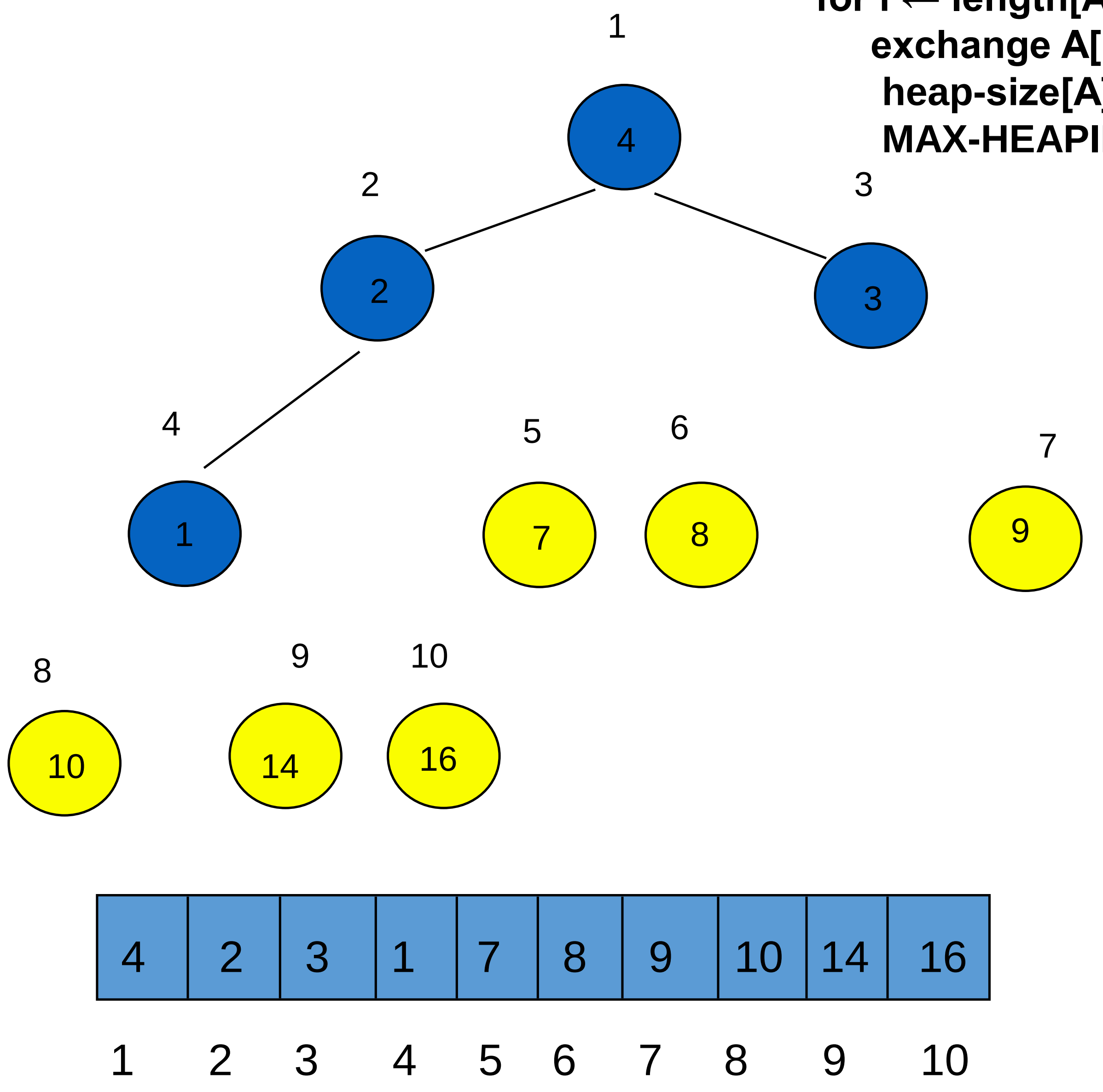
7	4	3	1	2	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```

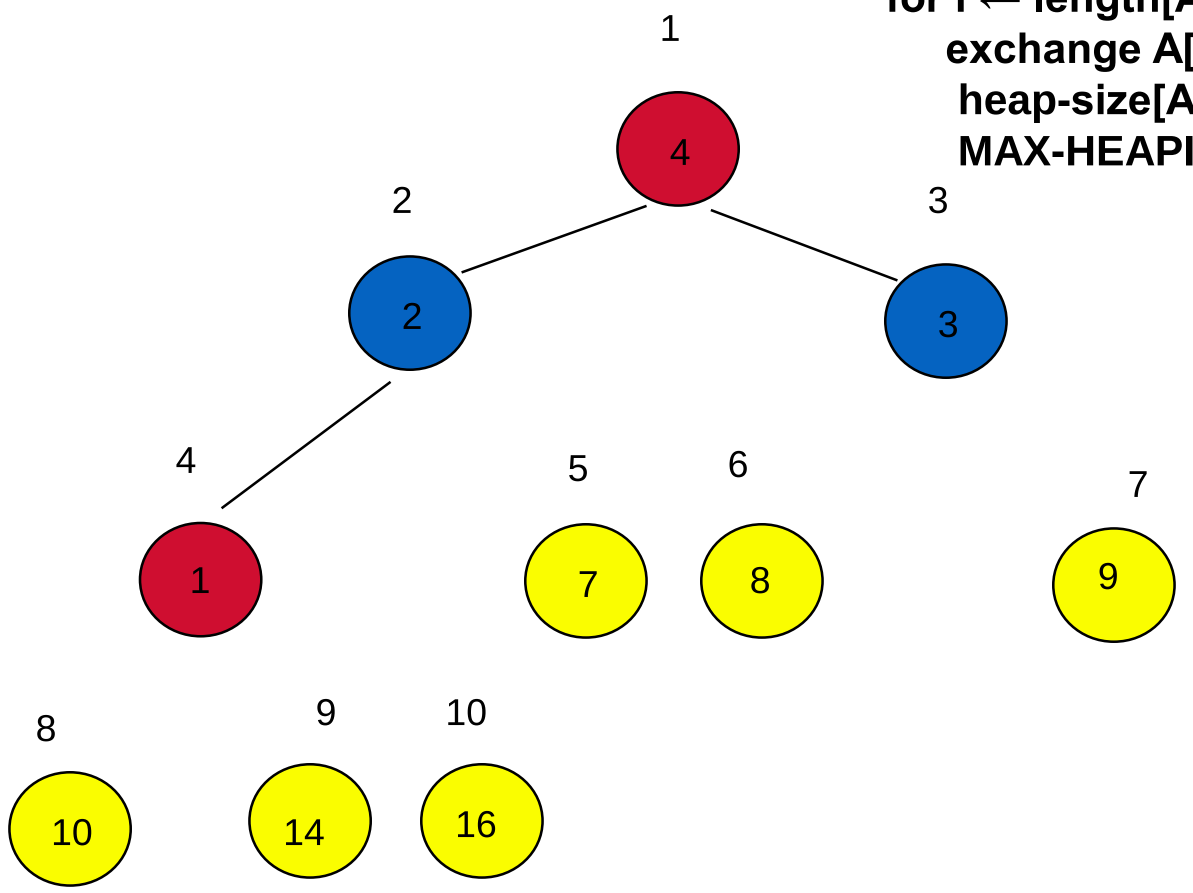


2	4	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)

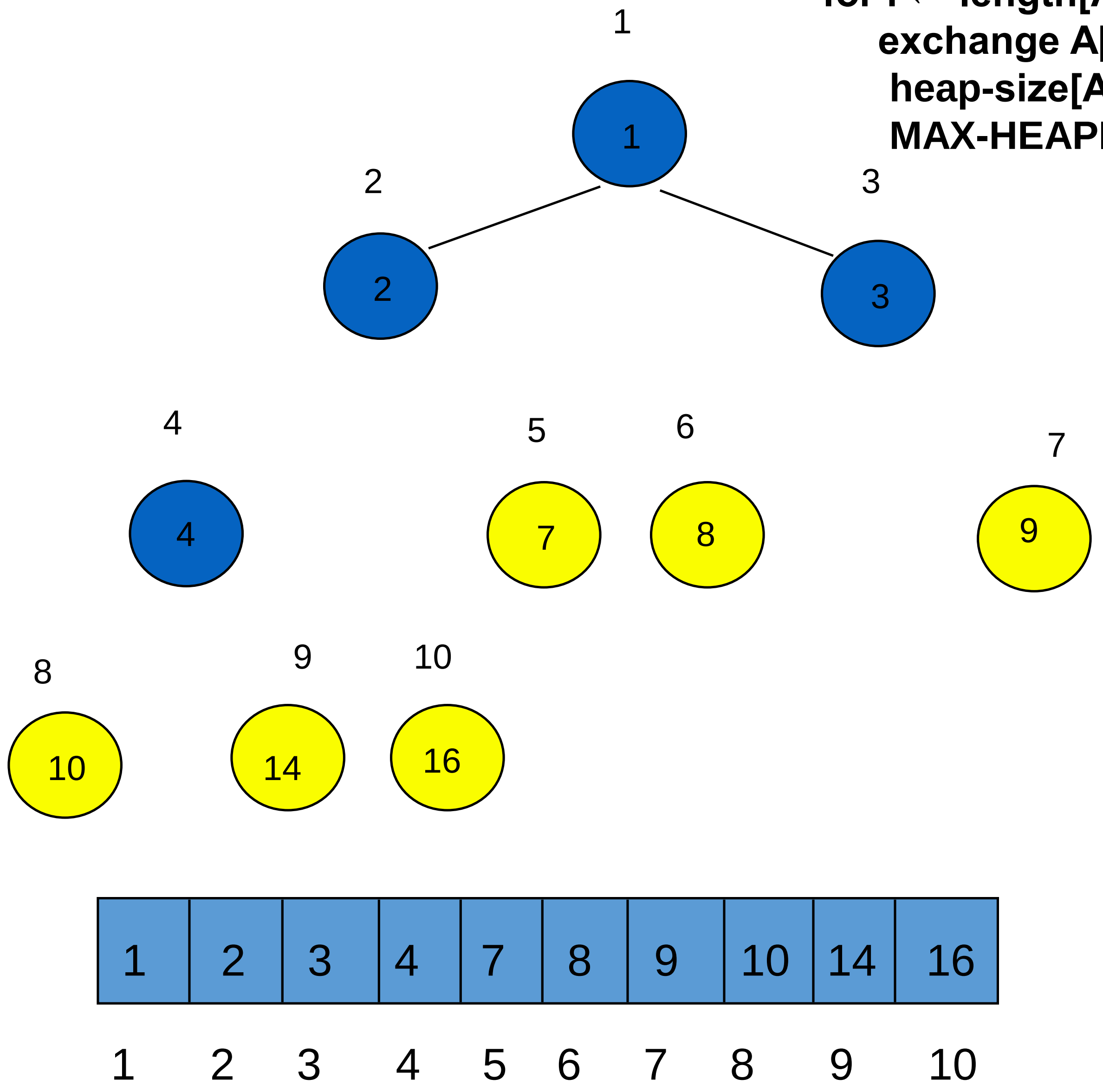


```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```

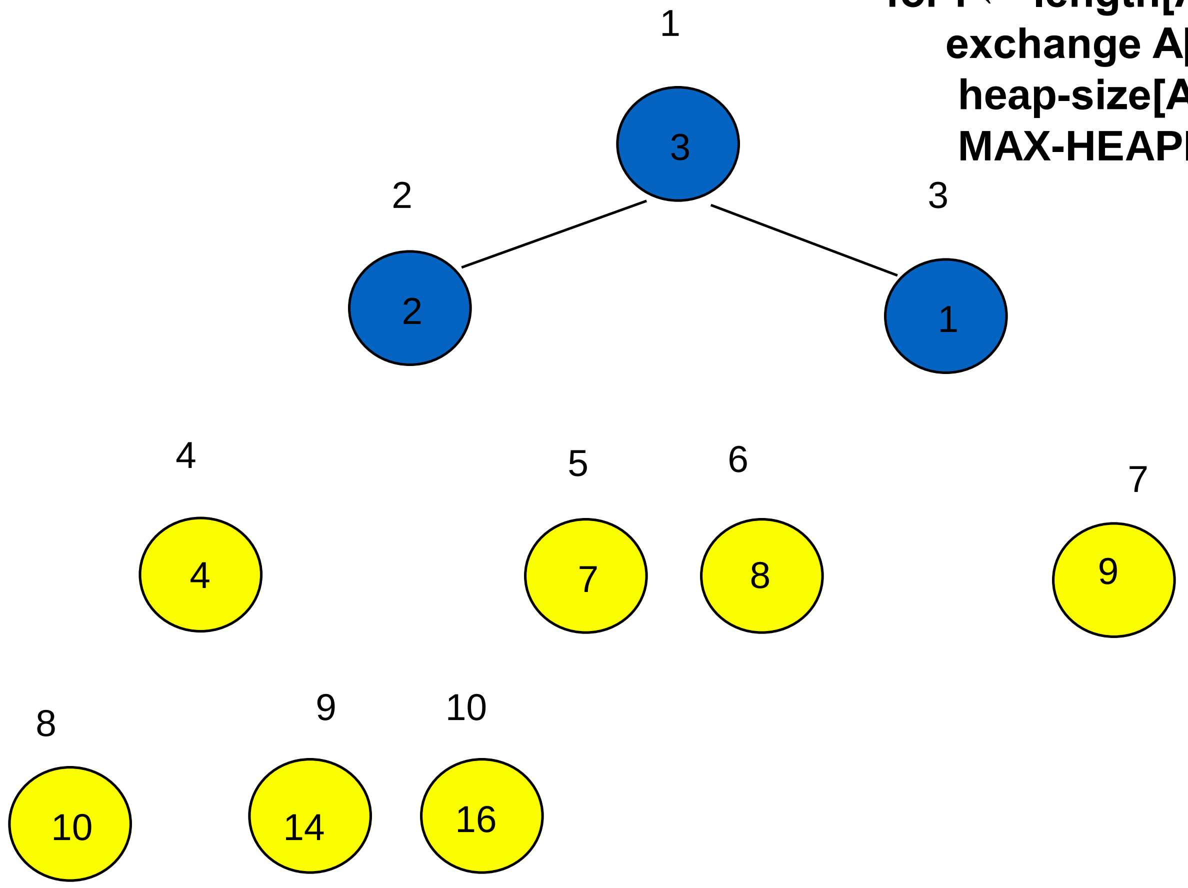


4	2	3	1	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)

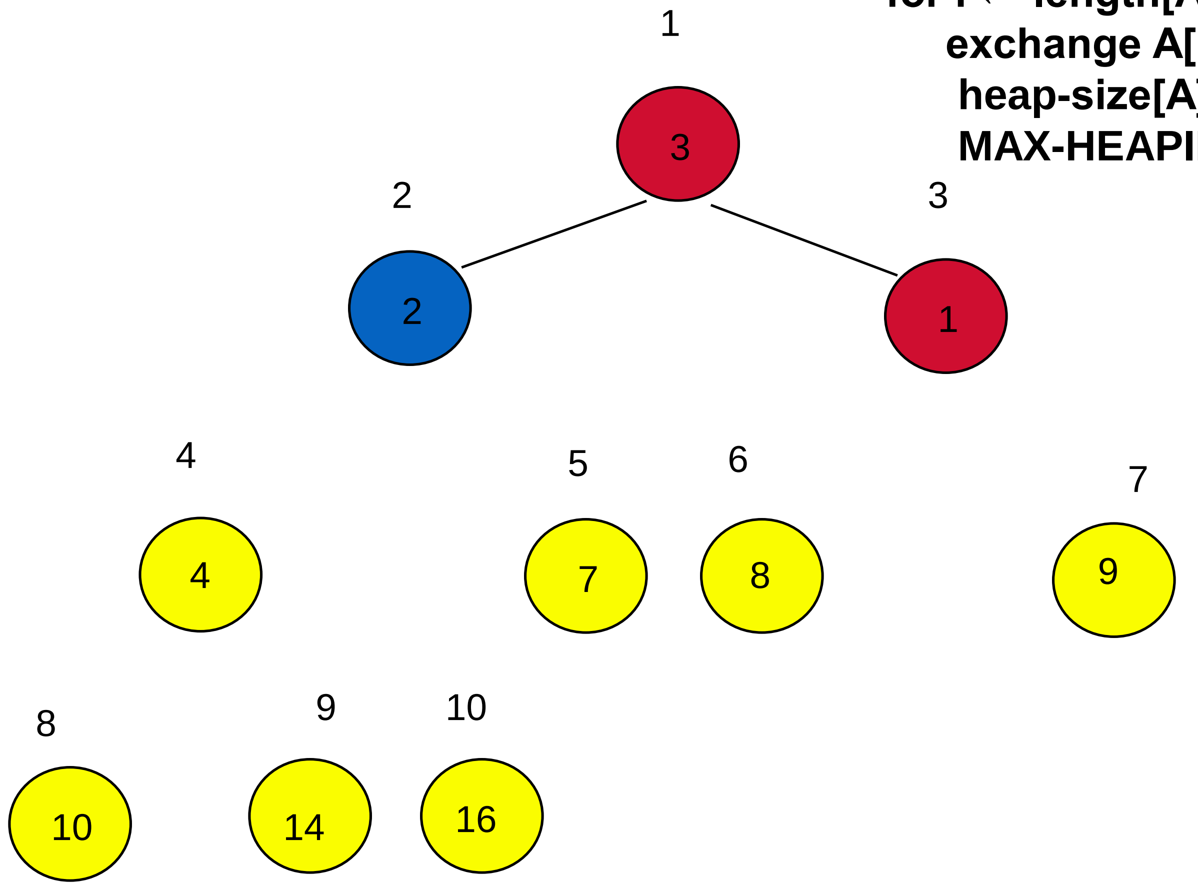


```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



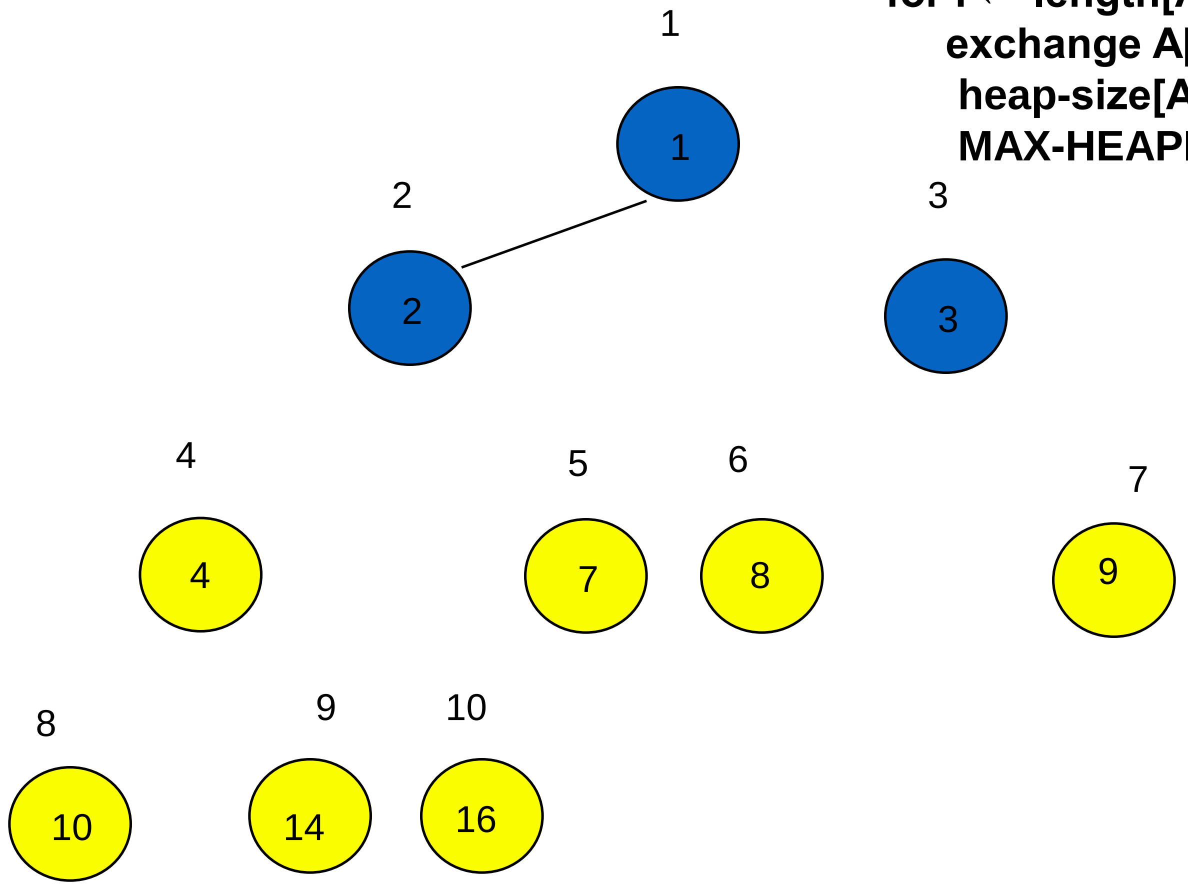
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```



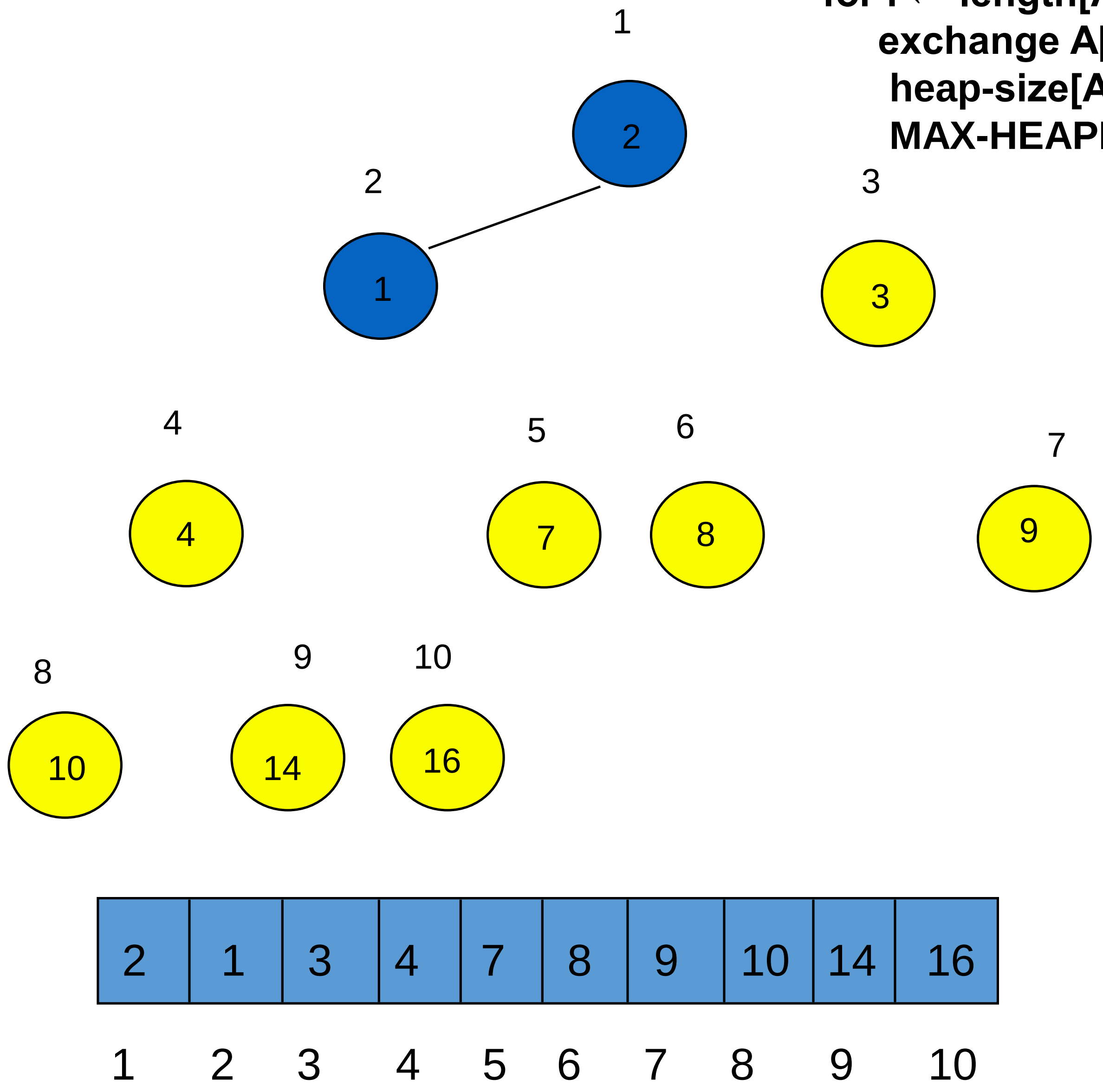
3	2	1	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
     exchange A[1] ↔ A[i]
      heap-size[A] ← heap-size[A] – 1
      MAX-HEAPIFY(A, 1)
```

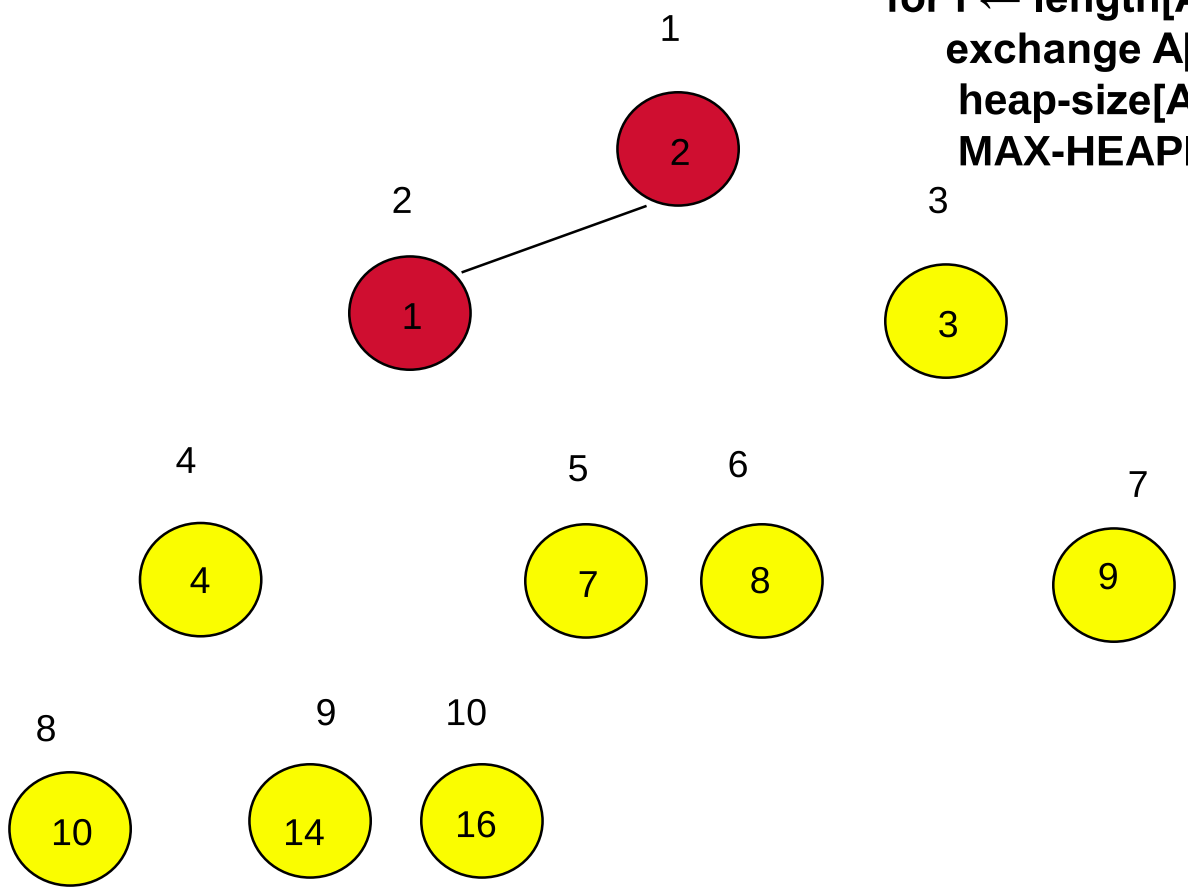


1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)

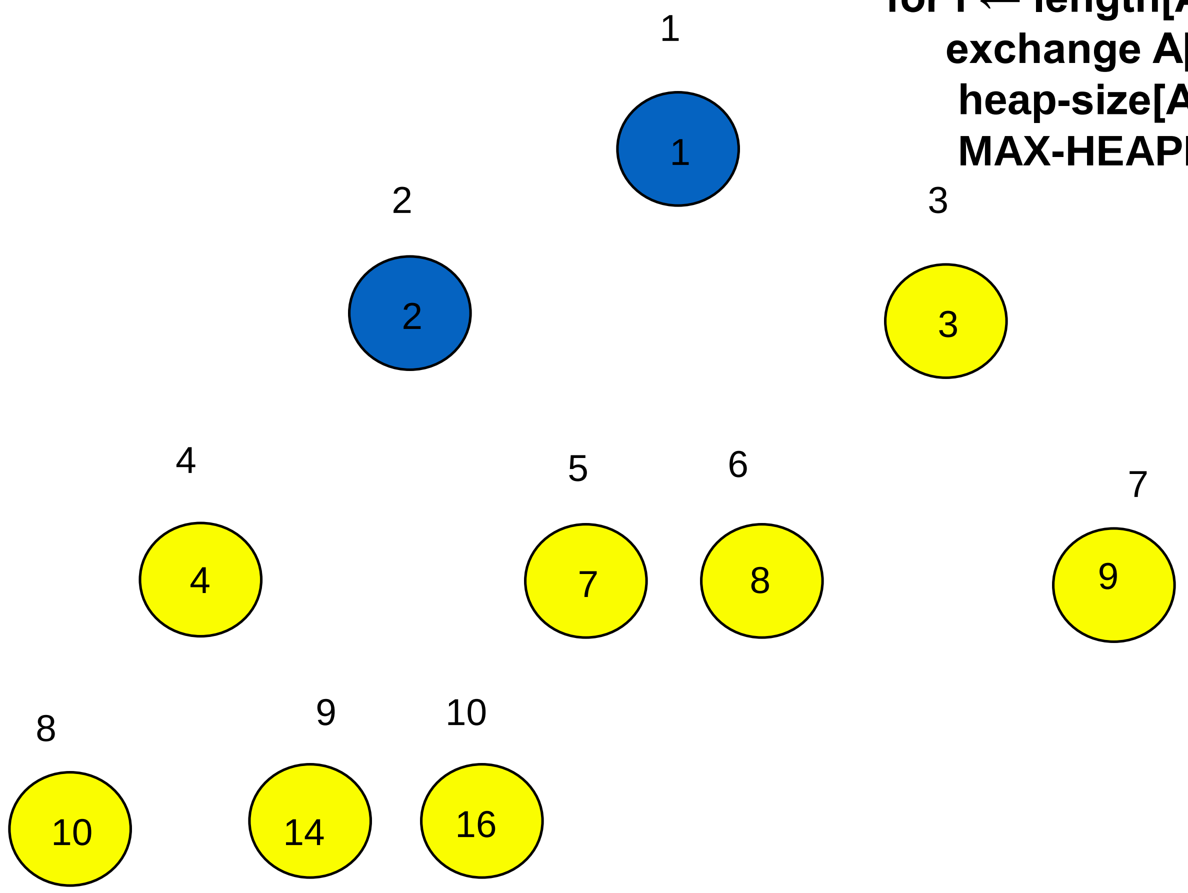


```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



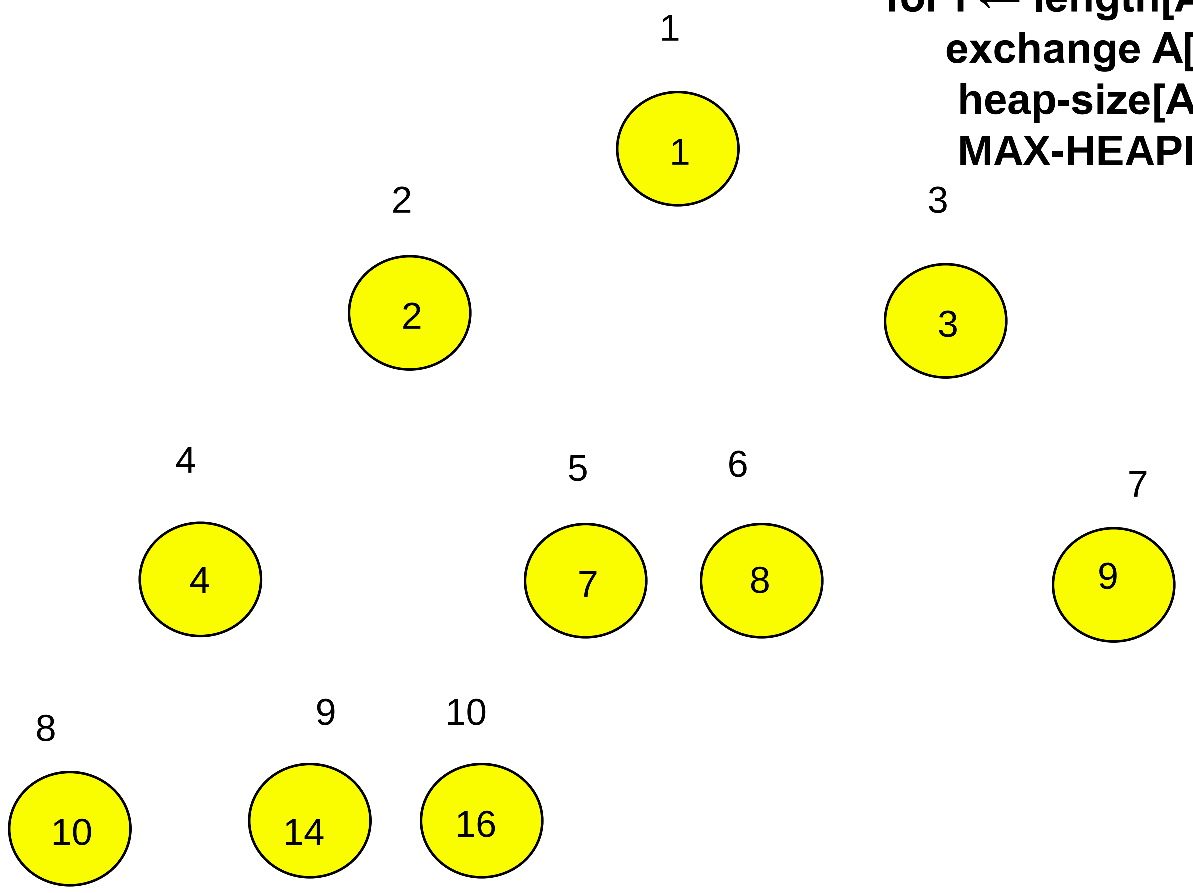
2	1	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
exchange A[1] ↔ A[i]
heap-size[A] ← heap-size[A] – 1
MAX-HEAPIFY(A, 1)



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

```
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
     heap-size[A] ← heap-size[A] – 1
     MAX-HEAPIFY(A, 1)
```



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

Running time of Heapsort

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i ← length[A] downto 2 do
    exchange A[1] ↔ A[i]
    heap-size[A] ← heap-size[A] – 1
    MAX-HEAPIFY(A, 1)
```

Is there a loop? If so, how many times will it execute? What is the cost of one iteration of the loop?

Running time of Heapsort

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)                          O(n)
  for i ← length[A] downto 2 do              O(n-1)
   exchange A[1] ↔ A[i]                      O(1)
   heap-size[A] ← heap-size[A] – 1           O(1)
   MAX-HEAPIFY(A, 1)                         O(lg n)
```

Total time is:

$$O(n) + O(n\text{-}1) * [\ O(1) + O(1) + O(\lg n)\ ]$$

which is approximately

$$O(n) + O(n \lg n)$$

or just $O(n \lg n)$