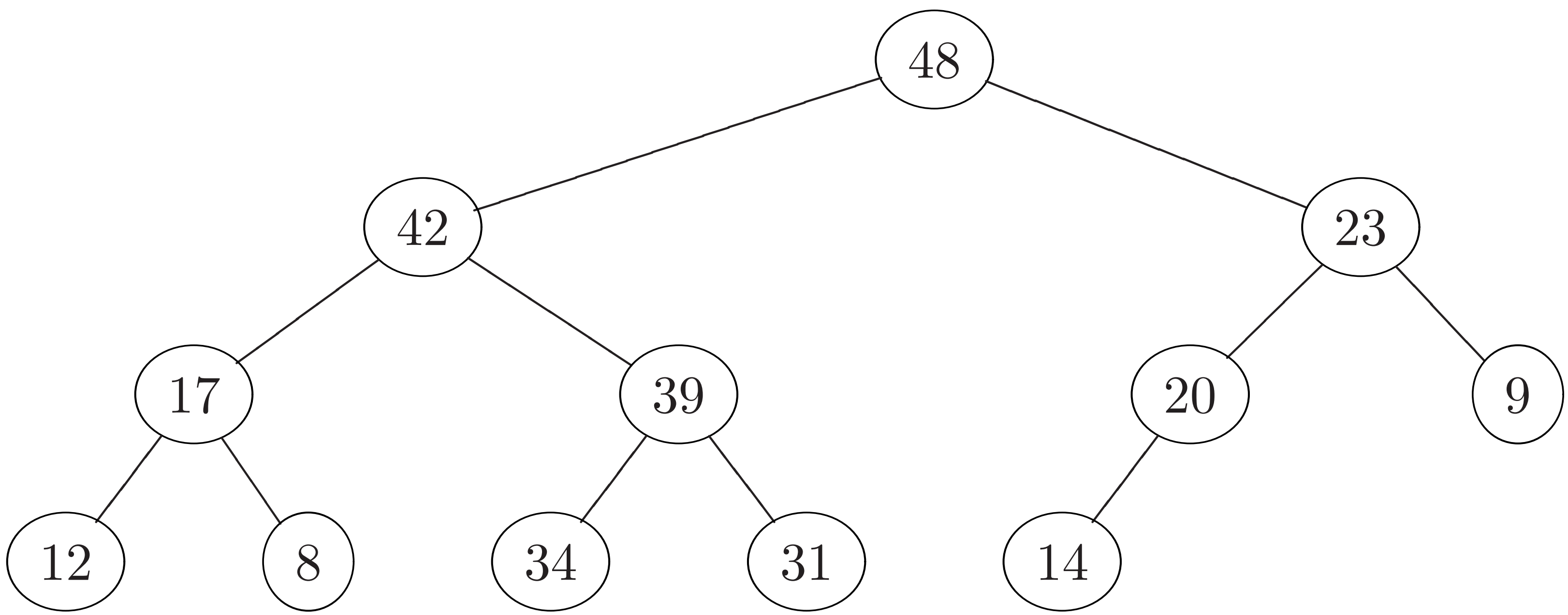
Heaps

Priority Queue

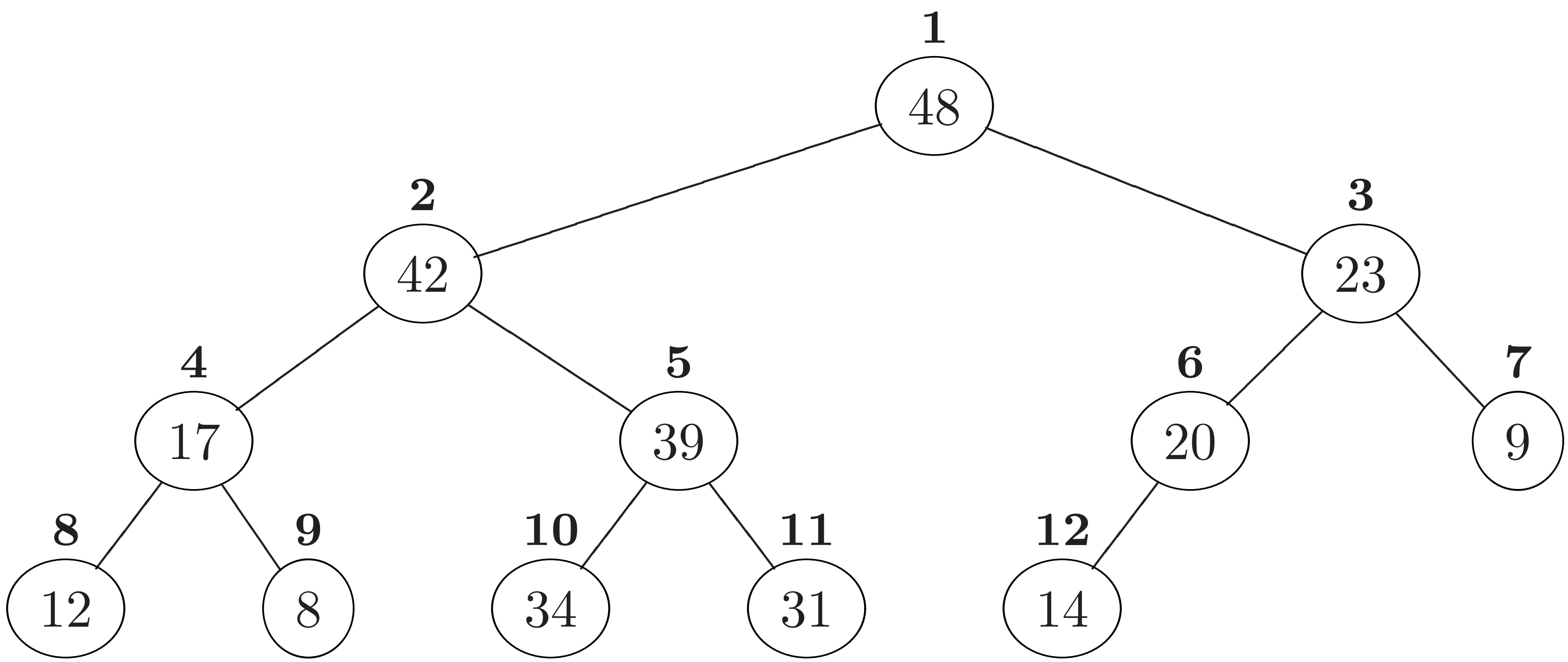
Operations:

- D.`Init` () - Initialize data structure D;
- D.Insert (x) - Insert element x into data structure D;
- D.`ExtractMax` (x) - Extract maximum element from D.

(Max) Heap



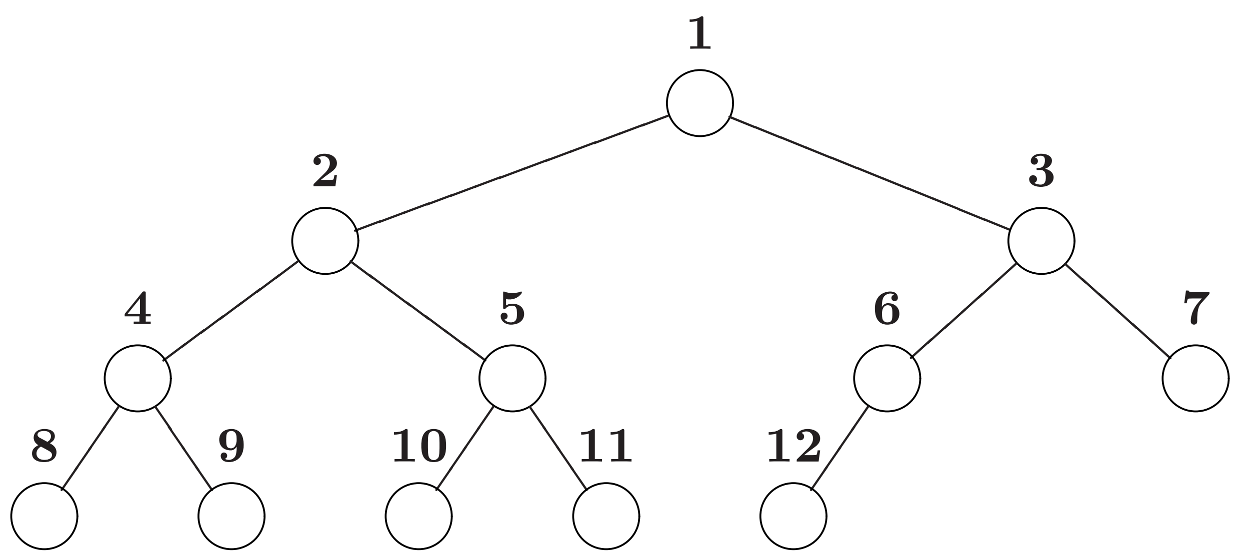
(Max) Heap



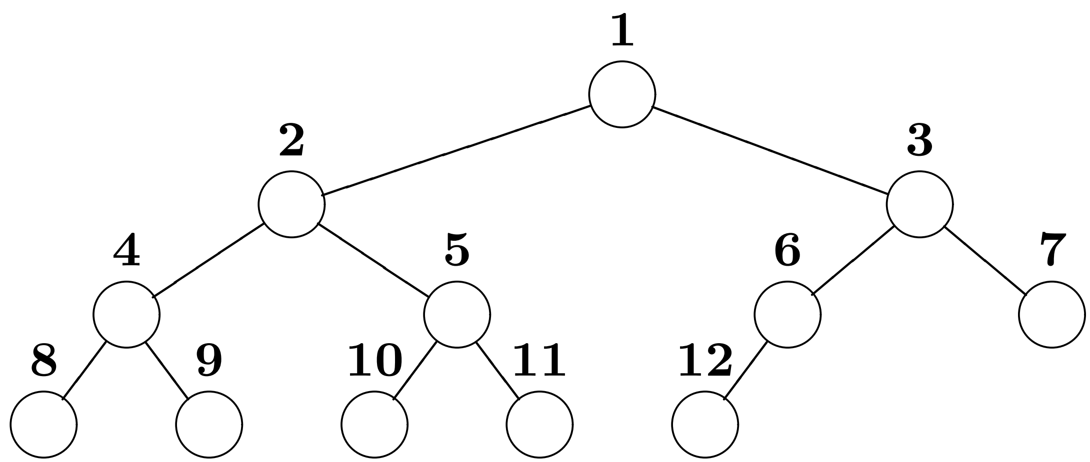
Array representation:

$$[48, 42, 23, 17, 39, 20, 9, 12, 8, 34, 31, 14]$$

Nearly Complete Binary Tree



Only nodes on bottom right of tree are missing.



function `Parent`(i)

return $(\lfloor i/2 \rfloor)$;

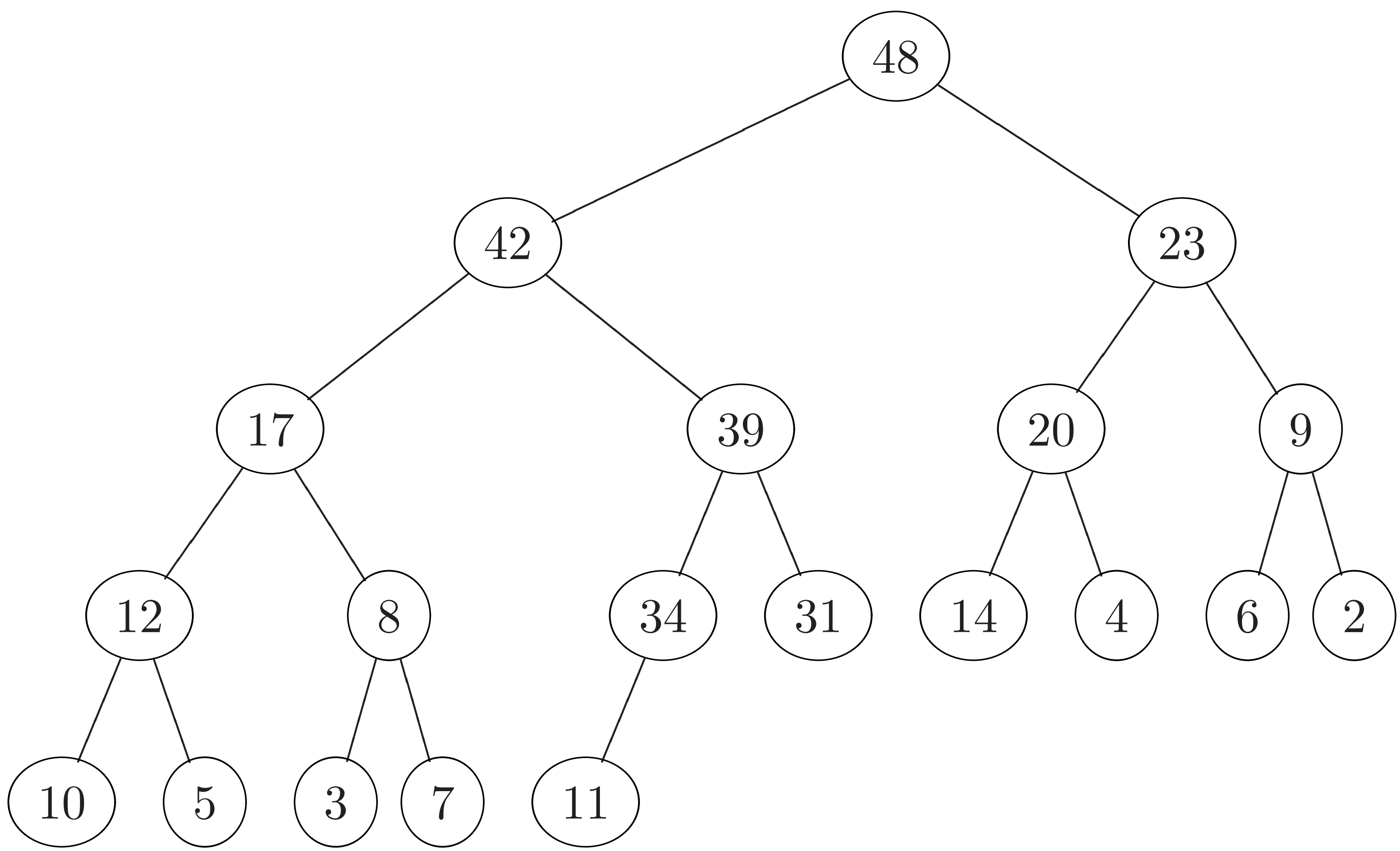
function `Left`(i)

return $(2i)$;

function `Right`(i)

return $(2i + 1)$;

Max Heap Insertion

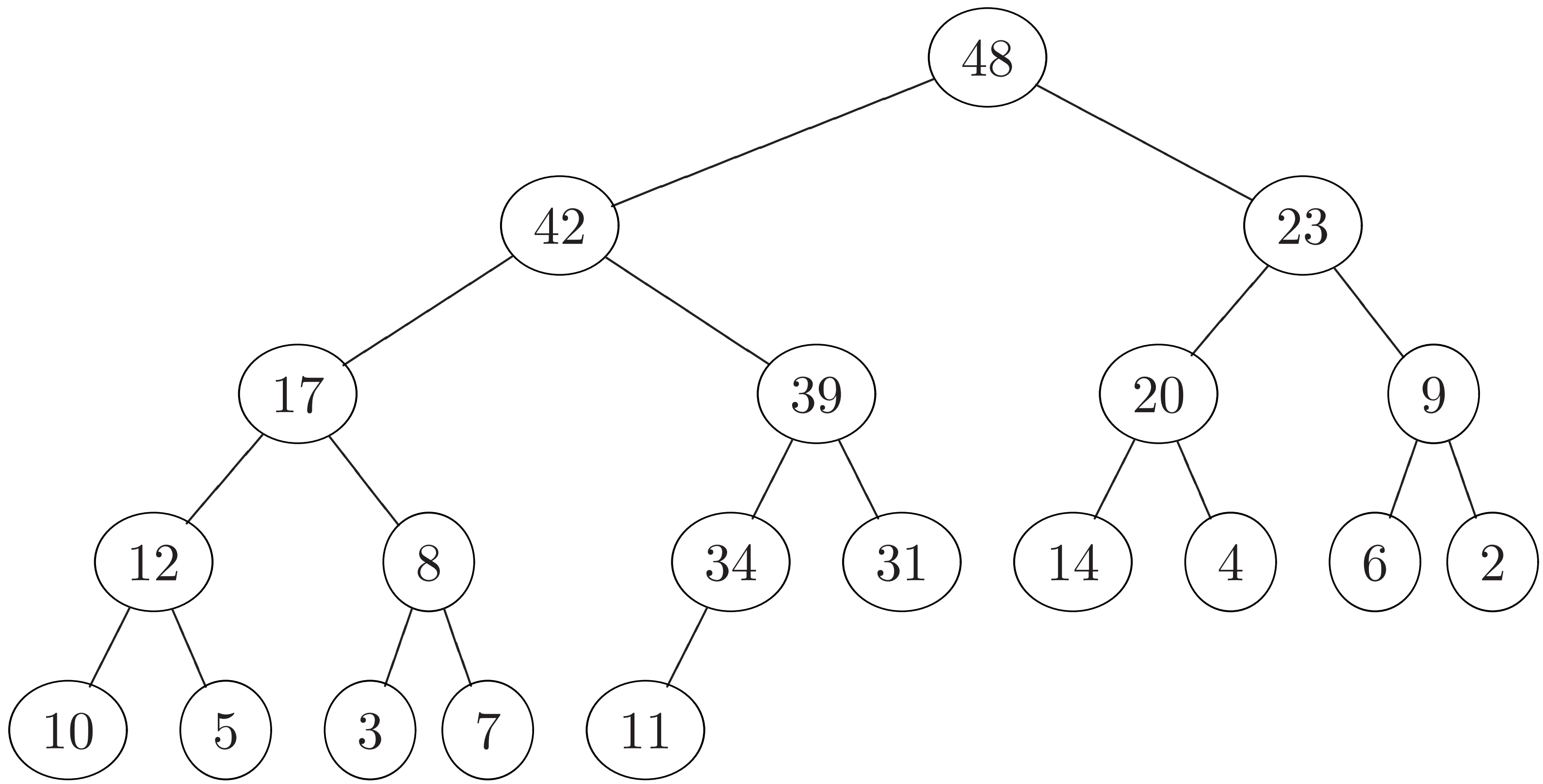


Max Heap Insertion

Input : Array A of size elements.
Key K.

```
function MaxHeapInsert(A[ ],size,K)
size ← size + 1;
i ← size;
A[i] ← K;
while (i > 1) and (A[Parent(i)] < A[i]) do
    Swap (A[i],A[Parent(i)]);
    i ← Parent(i);
end
```

Extract Heap Max



Extract Heap Max

Input : Array A of size elements.

Output : Maximum element in the heap.

```
function HeapExtractMax(A[ ],size)
if (size < 1) then error("heap underflow");
maxKey ← A[1];
A[1] ← A[size];
size ← size − 1;
MaxHeapify (A,size);
return (maxKey);
```

procedure MaxHeapify

```
procedure MaxHeapify(A[ ],size)
flagDone ← false;
j ← 1;
repeat
    L ← Left(j);
    R ← Right(j);
    /* Find largest ∈ {j, L, R} such that
        A[largest] = max(A[j], A[L], A[R])                */
    largest ← j;
    if (L ≤ size) and (A[L] > A[largest]) then largest ← L;
    if (R ≤ size) and (A[R] > A[largest]) then largest ← R;
    if (j ≠ largest) then Swap (A[j],A[largest]);
    else flagDone ← true;
    j ← largest;
until (flagDone = true);
```

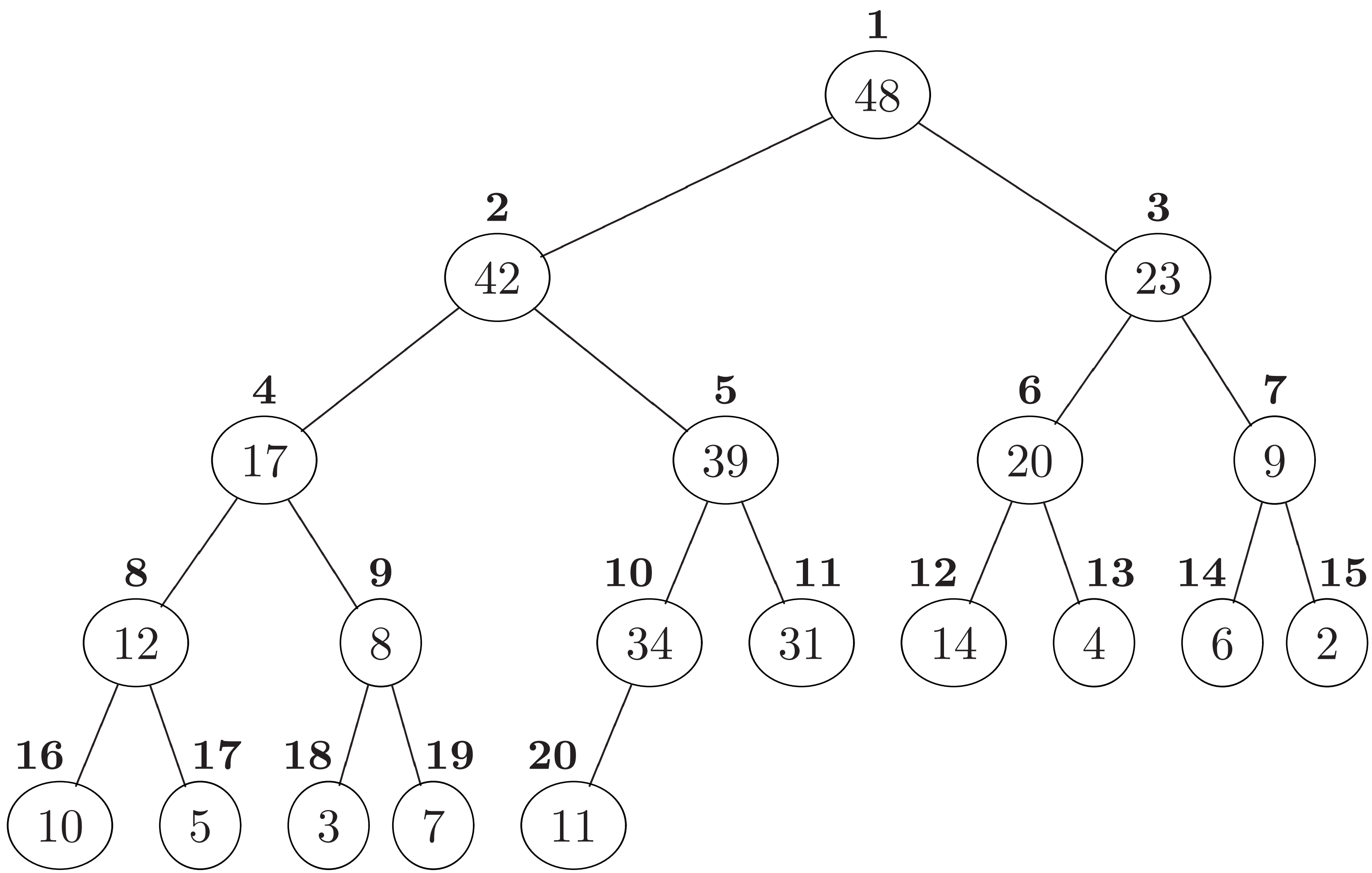
Extract Heap Max

Input : Array A of size elements.

Output : Maximum element in the heap.

```
function HeapExtractMax(A[ ],size)
if (size < 1) then error("heap underflow");
maxKey ← A[1];
A[1] ← A[size];
size ← size − 1;
MaxHeapify (A,size,1);
return (maxKey);
```

Heap Increase Key



Heap Increase Key

Input : Array A of size elements.
Index i.
Key K.

```
function MaxHeapIncreaseKey(A[ ],size,i,K)
if (K < A[i]) then error "new key is less than current key";
A[i] ← K;
while (i > 1) and (A[Parent(i)] < A[i]) do
    Swap (A[i],A[Parent(i)]);
    i ← Parent(i);
end
```

HeapSort

Input : Array B of n elements.

Result : Permutation of B such that $B[1] \leq B[2] \leq B[3] \leq \ldots \leq B[n]$.

```
procedure HeapSort(B[ ], n)
size ← 0;
for i ← 1 to n do
    MaxHeapInsert (A, size, B[i]);
end
for i ← n downto 1 do
    B[i] ← HeapExtractMax(A, size);
end
```

Analyzing Data Structures

Priority Queue

Operations:

- D.`Init` () - Initialize data structure D;
- D.Insert (x) - Insert element x into data structure D;
- D.`ExtractMax` (x) - Extract maximum element from D.

Heap:

- D.`Init` () - $\Theta(1)$ time;
- D.Insert (x) - $\Theta(\log(s))$ time;
- D.`ExtractMax` (x) - $\Theta(\log(s))$ time.

$s = \#$ elements in the data structure.

Priority Queue: Array Implementation

Elements of data structure D: Array A[ ], length.

```
procedure D.Init()
length ← 0;
```

Priority Queue: Array Implementation

Elements of data structure D: Array A[ ], length

```
procedure D.Insert(x)
length ← length + 1;
A[length] ← x;

procedure D.ExtractMax()
if (length < 1) then error "array underflow";
for i ← 1 to (length − 1) do
|   if (A[i] > A[length]) then Swap (A[i], A[length]);
end
x ← A[length];
length ← length − 1;
return (x);
```

Sort Using a Priority Queue: Array Implementation

Input : Array B of n elements.

Result : Permutation of B such that
$B[1] \leq B[2] \leq B[3] \leq \ldots \leq B[n]$.

procedure `Sort`(B[ ], n)

```
D.Init();
for i ← 1 to n do
    D.Insert(B[i]);
end
for i ← n downto 1 do
    B[i] ← D.ExtractMax();
end
```

Priority Queue: Sorted Array Implementation

Elements of data structure D: Array A[ ], length.

procedure D.`Init()`

```
length ← 0;
```

Priority Queue: Sorted Array Implementation

Elements of data structure D: Array A[ ], length

```
procedure D.Insert(x)
length ← length + 1;
A[length] ← x;
k ← length;
while (k > 1) and (A[k − 1] > A[k]) do
    Swap(A[k − 1],A[k]);
    k ← k − 1;
end
```

```
procedure D.ExtractMax()
if (length < 1) then error "array underflow";
x ← A[length];
length ← length − 1;
return (x);
```

Sort Using a Priority Queue: Sorted Array Implementation

Input : Array B of n elements.

Result : Permutation of B such that
$B[1] \leq B[2] \leq B[3] \leq \ldots \leq B[n]$.

```
procedure Sort(B[ ], n)
D.Init();
for i ← 1 to n do
    D.Insert(B[i]);
end
for i ← n downto 1 do
    B[i] ← D.ExtractMax();
end
```

Priority Queue Comparisons

s = # elements in the priority queue.
n = # elements in input to `Sort()`.
c is a constant.

	Heap	Array	Sorted Array
Insert	$\log(s)$	c	s
`ExtractMax`	$\log(s)$	s	c
`Sort`	$n \log(n)$	n^2	n^2

Example 1

```
procedure func(B[ ], n)
/* B is an array of n elements                                  */
D.Init();
for i ← 1 to n do
    for j ← 1 to n do
        D.Insert(B[i] ∗ B[j]);
    end
end
x ← 0;
for i ← 1 to n do
    x ← x + D.ExtractMax();
end
return (x);
```

Insert time: $\Theta(s)$. (s = Number of elements in D.)
ExtractMax time: $\Theta(1)$.

Example 2

```
procedure func(B[ ], n)
/* B is an array of n elements                                  */
D.Init();
for i ← 1 to n do
    for j ← 1 to n do
        D.Insert(B[i] * B[j]);
    end
end
x ← 0;
for i ← 1 to n do
    x ← x + D.ExtractMax();
end
return (x);
```

Insert time: $\Theta(1)$.
ExtractMax time: $\Theta(s)$. (s = Number of elements in D.)