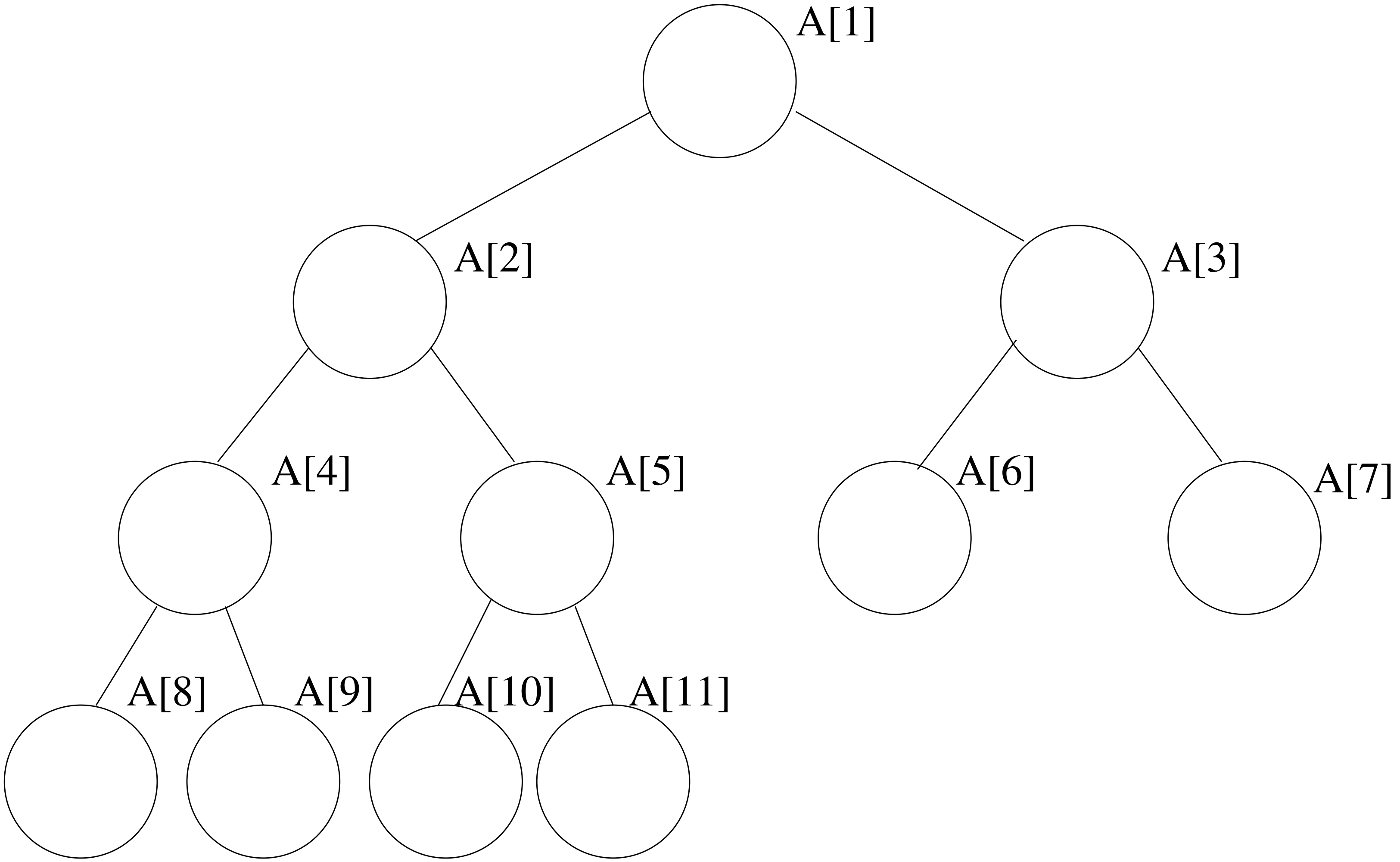
```
Heapify(node):
  if not(node.left) and not(node.right) done!!
  else
    if node.right and node.right.value<node.left.value
      k:=node.right
    else
      k:=node.left

    if node.value<=k.value done!!
    else
      swap(node.value,k.value);
      Heapify(k);
```


A[1]
A[2]
A[3]
A[4]
A[5]
A[6]
A[7]
A[8]
A[9]
A[10]
A[11]

```
Heapify(node):
  // pre: node.left (2*node), node.right (2*node+1) are roots
  //      of valid heaps or out of bounds
  // post: node is a root of a valid heap
  // for simplicity assume unused part of A filled with infty
  while (A[node]>A[2*node] or A[node]>A[2*node+1]) {
    if A[2*node]<A[2*node+1]
      k:=2*node
    else
      k:=2*node+1
    swap(A[node],A[k])
    node:=k
```

```
create an empty min priority queue PQ
for i:=1 to n
    PQ.insert(A[i])
for i:=1 to n
    A[i]=PQ.extractMin
```

Prioritné fronty

Implementácia	Insert	ExtractMin	Pozn.
Neutriedené pole	$\Theta(1)$	$\Theta(n)$	
Utriedené pole	$\Theta(n)$	$\Theta(1)$	
Heap	$\Theta(\log n)$	$\Theta(\log n)$	
Počítadlá	$\Theta(1)$	$\Theta(1)$	malá doména (napr. $[1, 1000]$)

```
Heapify(node):
  // pre: node.left (2*node), node.right (2*node+1) are roots
  //      of valid heaps or out of bounds
  // post: node is a root of a valid heap

BuildHeap(n):
  // build heap from values stored in A[1..n]
  for i:=n/2 downto 1
    // inv: elements i+1,...,n are roots of valid heaps
    Heapify(i);
    // inv: elements i,i+1,...,n are roots of valid heaps
```

```
MaxHeapify(node,size):
  while (2*node<=size and A[node]<A[2*node])
     or (2*node+1<=size and A[node]<A[2*node+1])
    if 2*node+1>size or A[2*node]>A[2*node+1]
      k:=2*node
    else
      k:=2*node+1
    swap(A[node],A[k]); node:=k
```

```
HeapSort:
  for i:=n/2 downto 1
    MaxHeapify(i,n)
  for i:=n downto 1
    Swap(A[1],A[i]);
    MaxHeapify(1,i-1);
```

```
QuickSort(from,to):
  if to>from
    i:=Partition(from,to);
    QuickSort(from,i-1);
    QuickSort(i+1,to);
```

```
Partition(from,to):
  // post: pivot is at its correct position A[ret]
  //         from<=j<=ret => A[j]<=pivot
  //         ret<j<=to => A[j]>pivot
  pivot:=A[to];
  i:=from-1;
  for j:=from to to
    // inv: from<=k<=i => A[k]<=pivot
    //       i<k<j => A[k]>pivot
    if A[j]<=pivot
      i:=i+1
      swap(A[j],A[i])
  return i;
```

```
RandomizedPartition(from,to):
  // post: pivot is at its correct position A[ret]
  //       from<=j<=ret => A[j]<=pivot
  //       ret<j<=to => A[j]>pivot
  *** change here ***
  swap(A[to],A[random(from,to)]); // pick a random index as a piv
  *******************
  pivot:=A[to];
  i:=from-1;
  for j:=from to to
    // inv: from<=k<=i => A[k]<=pivot
    //      i<k<j => A[k]>pivot
    if A[j]<=pivot
      i:=i+1
      swap(A[j],A[i])
  return i;
```