# Data Structures – LECTURE 7

## Heapsort and priority queues

- Motivation
- Heaps
- Building and maintaining heaps
- Heap-Sort
- Priority queues
- Implementation using heaps



## Priority queues and heaps

- We need an efficient ADT to keep a dynamic set *S* of elements *x* to support the following operations:
  - Insert($x$,$S$) – insert element *x* into S
  - Max($S$) – returns the maximum element
  - Extract-Max($S$) – remove and return the max. element
  - Increase-Key($x$,$k$,$S$) – increase *x*'s value to *k*
- This is called a *priority queue* (max-priority or min-priority queue)
- Priority queues are implemented using a *heap*, which is a tree structure with special properties.

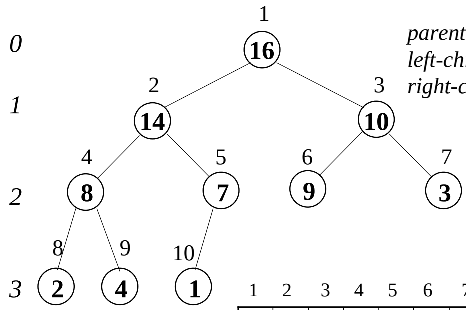


## Heaps

- A heap is a nearly complete binary tree.
- The binary tree is filled on all levels except possibly the last one, which is filled from the left to the right up to the last element.
- The tree is implemented as an array $A[i]$ of length *length*[*A*]. The number of elements is *heapsize*[*A*]
- Nodes in the tree have the property that parent node elements are greater or equal to children's node elements: ***A*[*parent*(*i*)] ≥ *A*[*i*]**
- Therefore, the maximum is at the root of the tree



## Example of a heap




*parent*($i$) = floor($i$/2)
*left-child*($i$) = 2*i*
*right-child*($i$) 2*i* +1

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



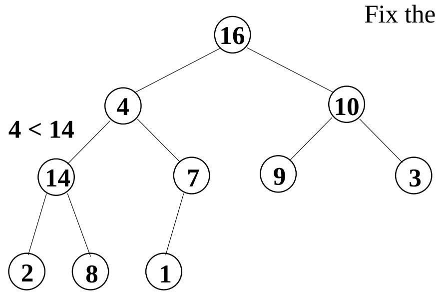
## Maintaining the heap property

- With a max-heap, finding the maximum element takes $O(1)$. Removing and inserting an element will take $O(\lg n)$, where $n$ = *heapsize*($A$)
- We need a procedure to maintain the heap property → Max-Heapify
- <u>The idea</u>: when inserting a new element *x* in the heap, find its place by "floating it down" when its value is smaller than the current node to the child with the *largest value*. Apply this method recursively until the right place is found.
- Since the tree has height $d = \lg n$, it will take $O(\lg n)$.

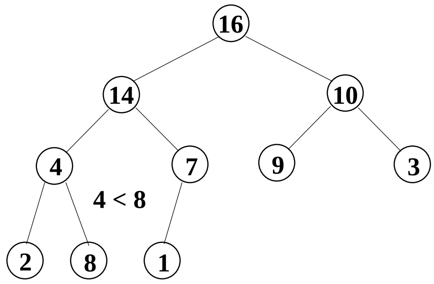


## Example of max-heapify

Fix the heap

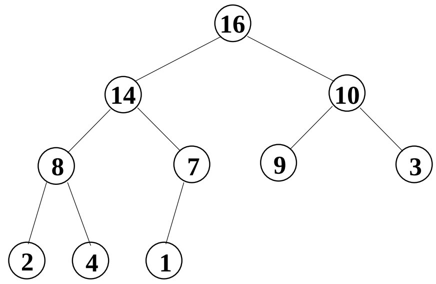


## Example of max-heapify






## Example of max-heapify






## Max-Heapify routine

```
Max-Heapify(A, i)
l ← left(i)
r ← right(i)
if  l ≤ heapsize[A] and A[l] > A[i]
   then largest ← l
   else  largest ← r
if  r ≤  heapsize[A] and A[r] > A[largest]
   then largest ← r
if  largest != i
   then Exchange(A[i],A[largest])
Max-Heapify(A,largest)
```



## Max-Heapify complexity

- The running time on a sub-tree of size $n$ rooted at node $i$ is:
  - $\Theta(1)$ to fix relations among elements $A[i]$
  - Time to recursively call Max-Heapify on a sub-tree rooted at one of the children of *I*. In the worst case, the size of such sub-tree is $2n/3$, which occurs when the last row of the tree is exactly half full.
- Thus, the recurrence is

$$T(n/2) + \Theta(1) \leq T(n) \leq T(2n/3) + \Theta(1)$$

- By the master theorem, $a = 1$, $b = 3/2$, $f(n) = \Theta(1)$ so $\log_b a = 0$, so case 2 applies: $T(n) = O(\lg n)$.



## Building a heap

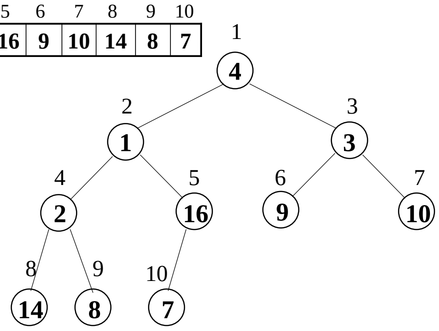
- Use Max-Heapify to recursively convert the array $A[i]$ into a max-heap from bottom to top
- The elements in the sub-array $A[(\lfloor n/2 \rfloor + 1)...n]$ are all leaves of the tree, so each is a 1-element heap to begin with. The Build-Max-Heap procedure has to go through the remaining nodes of the tree and run Max-Heapify on each one

```
Build-Max-Heap(A)
heapsize[A] ← length(A)
for  i ← ⌊length[A]/2⌋  downto 1
     do Max-Heapify(A,i)
```

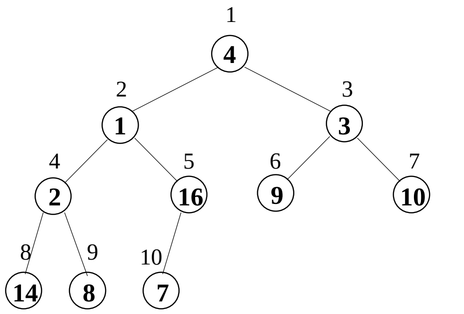


## Example: building a heap (1)

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

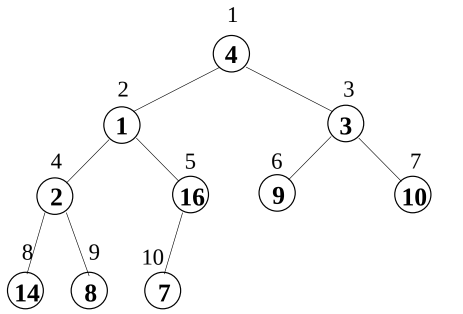


## Example: building a heap (2)



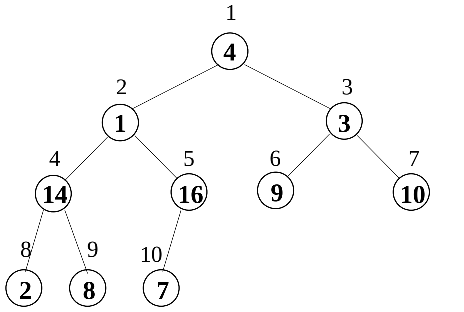



## Example: building a heap (3)



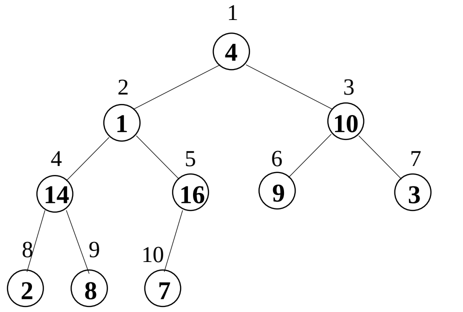



## Example: building a heap (4)



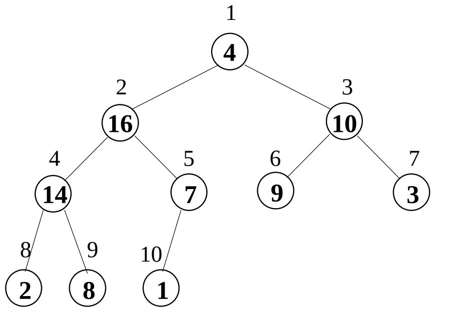



## Example: building a heap (5)



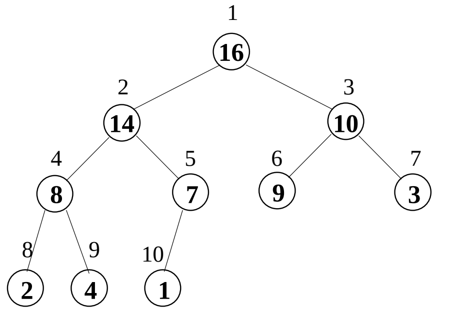



## Example: building a heap (6)






## Example: building a heap (7)



## Correctness of Build-Heap

- A useful technique for proving the correctness of an algorithm is to use *loop invariants*, which are properties that hold throughout the loop.
- It is very similar to induction, but it is stated in terms of the loop. We show that the loop invariant holds before the loop is executed, during the loop, and after the loop terminates.



## Invariant of Build-Heap

**Build-Max-Heap**($A$)
1. *heapsize*[$A$] ← *length*($A$)
2. **for** $i$ ← $\lfloor length[A]/2 \rfloor$ **downto** 1
3. **do** Max-Heapify($A$,$i$)

The loop invariant is:

Before the execution of each **for** step each node $i + 1, i + 2, \ldots , n$ is the root of a max-heap



## Proof of loop invariant

- Initialization: before the first iteration, $i$ = *floor*($n$/2) and each node is a leaf and is thus trivially a max-heap of size 1.
- Maintenance: show that if the invariant holds before the iteration, it will also hold after the iteration. Note that all the the nodes larger than $i$ are roots of a max-heap, from previous iterations. Therefore, the sub-tree rooted at $i$ is also a heap, but not a max-heap. After the execution of the Max-Heapify routine, it becomes a max-heap.
- Termination: $i$ = 0, so $A$[1] is the root of a max-heap



## Complexity analysis of Build-Heap (1)

- For each height $0 < h \leq \lg n$ , the number of nodes in the tree is at most $n/2^{h+1}$
- For each node, the amount of work is proportional to its height $h$, $O(h)$ → $n/2^{h+1} .O(h)$
- Summing over all heights, we obtain:

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



## Complexity analysis of Build-Heap (2)

- We use the fact that $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad for\ |x| < 1$

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

- Therefore:

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

- Building a heap takes only linear time and space!



## Sorting with heaps: Heap-Sort

- We can use the heap structure to sort an array $A[i]$ of $n$ elements in place:
- Since the maximum element of the array is its first element, $A$[1], it can be put in its correct final position at the end of the array, in $A[n]$.
- We can now recursively fix the heap of the sub-tree rooted at node 1 and containing $n - 1$ elements with Max-Heapify until two elements are left.
- Each call to Max-Heapify takes $O(\lg n)$, and it is called once for each element in the array, so the running time is $O(n \lg n)$ always (best, average, and worst case) with $O(n)$ space.

## Heap-Sort

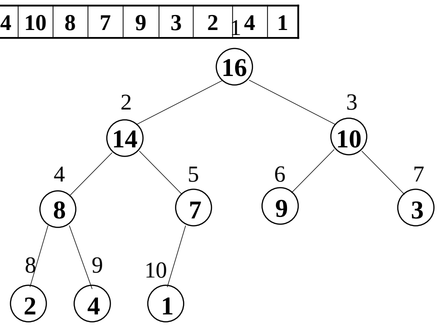
**Heap-Sort**(*A*)

1. Build-Max-Heap(*A*)
2. *heapsize*[*A*] ← *length*(*A*)
3. **for** *i* ← *length*[*A*] **downto** 2 *put maximum at the root*
4. **do** Exchange(*A*[*1*],*A*[*i*])
5. *heapsize*[*A*] ← *length*(*A*) –1
6. Max-Heapify(*A*,1)



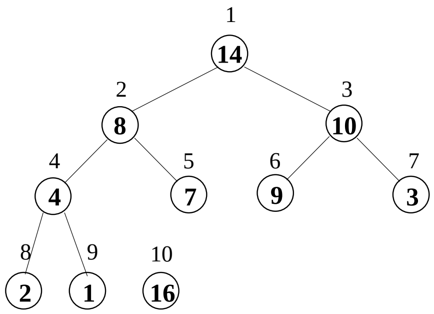
## Example: Heap-Sort

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **16** | **14** | **10** | **8** | **7** | **9** | **3** | **2** | **4** | **1** |



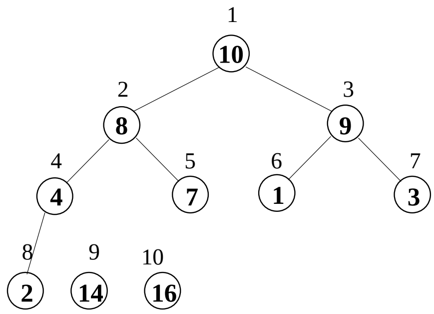



## Example: Heap-Sort (2)



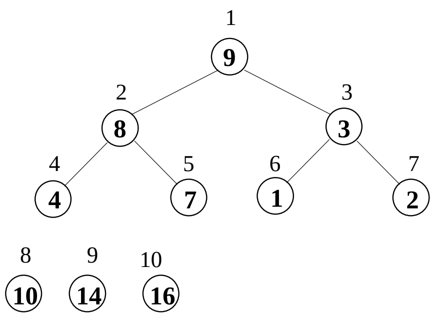



## Example: Heap-Sort (3)



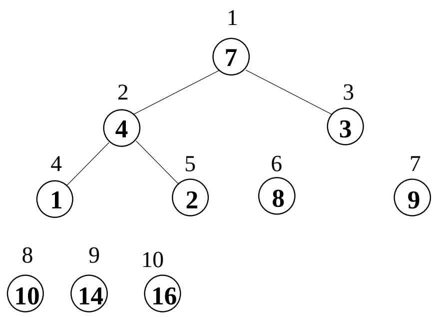



## Example: Heap-Sort (4)

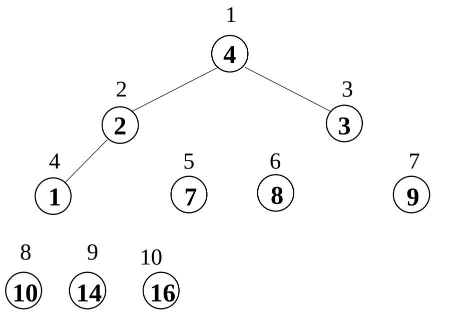





## Example: Heap-Sort (5)

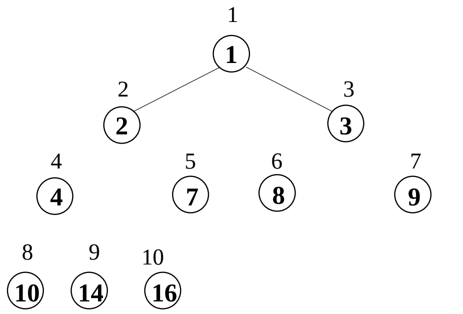


## Example: Heap-Sort (6)



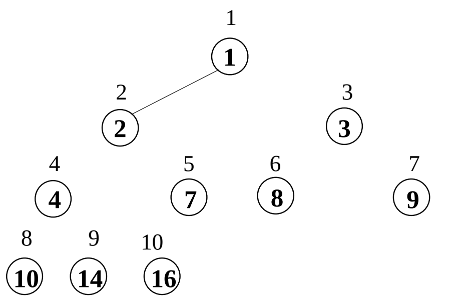



## Example: Heap-Sort (7)






## Example: Heap-Sort (8)




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **2** | **3** | **4** | **7** | **8** | **9** | **10** | **14** | **16** |



## Priority queues

We can implement the priority queue ADT with a heap. The operations are:

- Max(*S*) – returns the maximum element
- Extract-Max(*S*) – remove and return the maximum element
- Insert(*x*,*S*) – insert element *x* into S
- Increase-Key(*S*,*x*,*k*) – increase *x*'s value to *k*



## Priority queues: Extract-Max

**Heap-Maximum**(*A*) **return** *A*[1]

**Heap-Extract-Max**(*A*)

**1.** **if** *heapsize*[*A*] < 1
**2.** **then error** "heap underflow"
*3.* *max* ← *A*[*1*]
*4.* *A*[*1*] ← *A*[*heapsize*[*A*]]
*5.* *heapsize*[*A*] ← *heapsize*[*A*] –1
6. Max-Heapify(*A*,1)
**7.** **return** *max*

*Make last element the root*



## Priority queues: Increase-Key

**Heap-Increase-key**(*A*, *i*, *key*)

**1.** **if** *key* < *A*[*i*]
**2.** **then error** "new key smaller than existing one"
3. *A*[*i*] ← *key*
**4.** **while** *i* > 1 and *A*[*parent*(*i*)] < *A*[*i*]
**5.** **do** Exchange(*A*[*i*], *parent*(*A*[*i*]))
*6.* *i* ← *parent*(*i*)

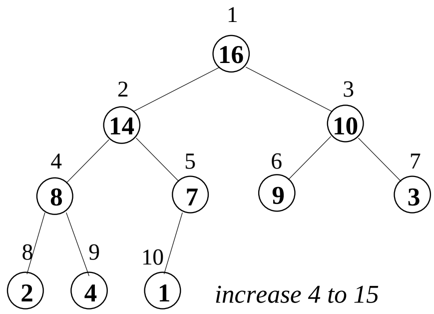
## Priority queues: Insert-Max

**<u>Heap-Insert-Max</u>**($A$, $key$)

1. $heapsize[A] \leftarrow heapsize[A] + 1$
2. $A[heapsize[A]] \leftarrow -\infty$
3. Heap-Increase-Key($A$, $heapsize[A]$, $key$)



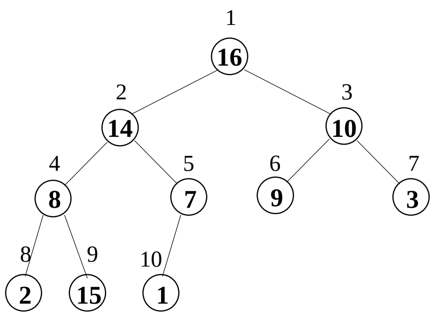
## Example: increase key (1)




*increase 4 to 15*

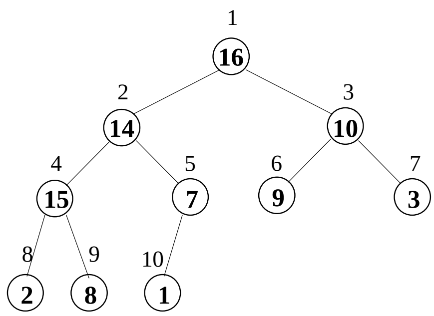


## Example: increase key (2)



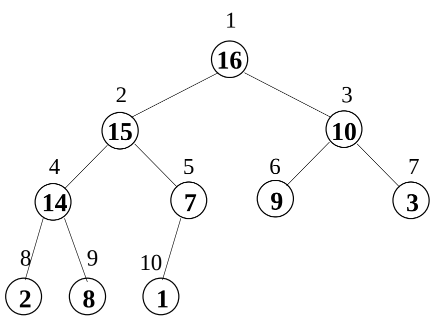



## Example: increase key (3)






## Example: increase key (4)






## Summary

- All dynamic operations on a heap take $O(\lg n)$
- All operations preserve the structure of the heap as an almost complete binary tree
- Tree rearrangement is in place
- Heapsort takes time $\Omega(n \lg n)$ and space $\Omega(n)$