DTS203TC
Design and Analysis of Algorithms

Lecture 6: Heap, Heapsort, Hash Tables

Dr. Qi Chen

School of AI and Advanced Computing

Learning outcomes

- Heaps
- Heapsort
- Priority queue

- Hash tables
- Hash function
- Collisions
- Table size
- Applications

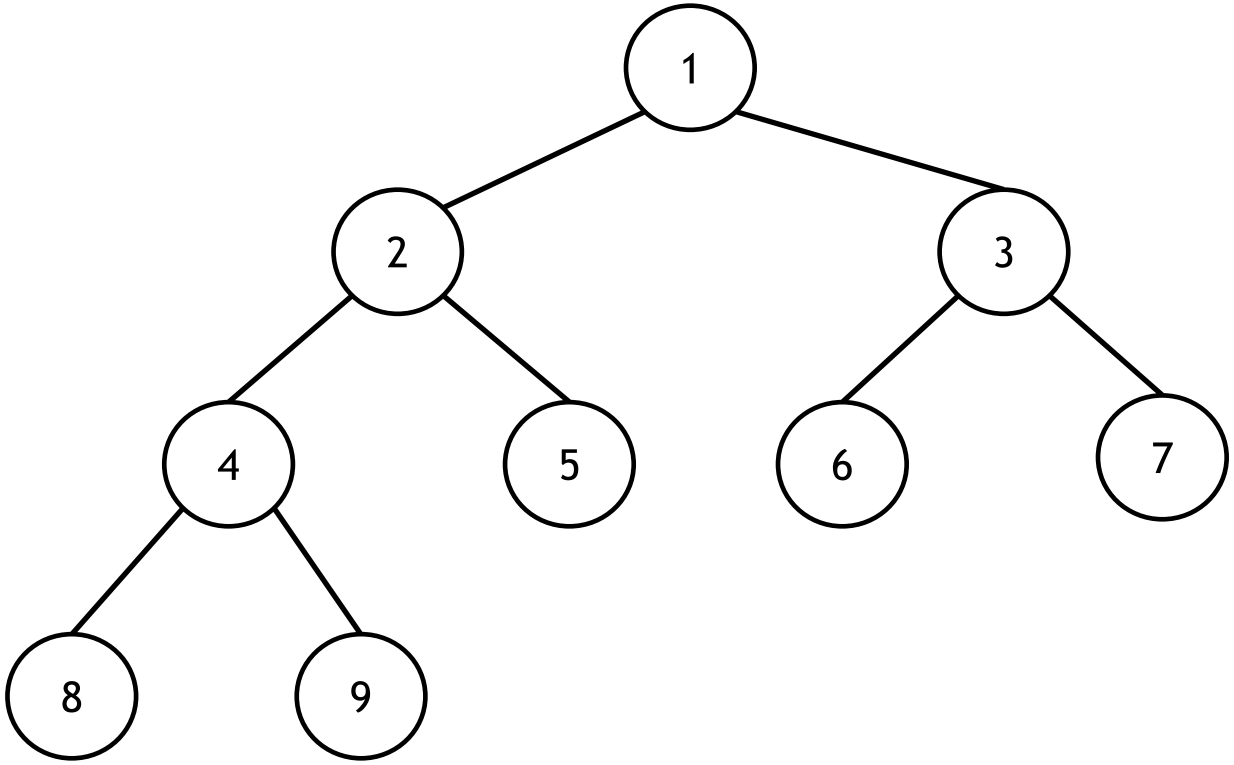
Binary Tree

- Binary tree is a tree data structure, each node has at most two children (*left child* and *right child*).

- Complete (perfect) binary tree
  - all interior nodes have two children
  - all leaves have the same depth or same level.

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

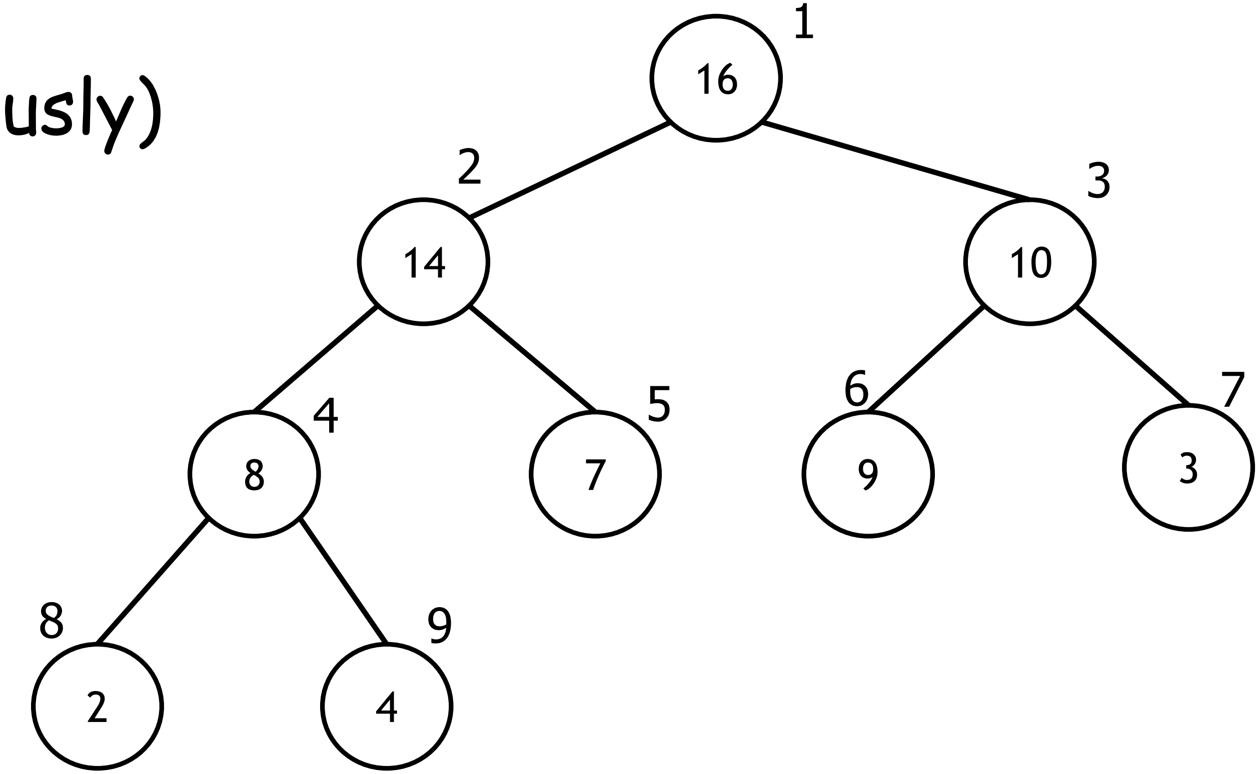
Binary Tree



Heap

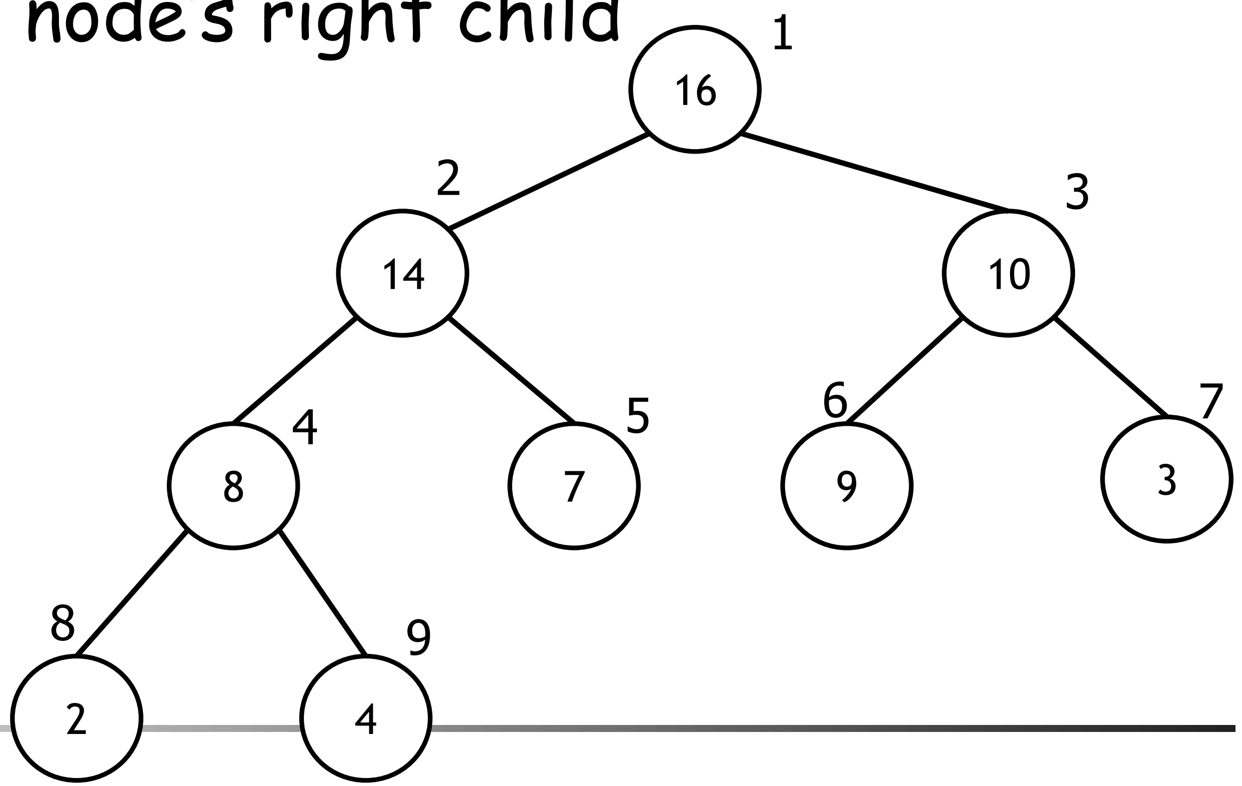
- An array, visualized as a nearly complete binary tree.
- Max Heap Property: The key of a node is ≥ than the keys of its children
- (Min Heap defined analogously)

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



Heap as a tree

- Root of tree: first element in the array, corresponding to i = 1
- Parent(i)=i/2: returns index of node's parent
- left(i)=2i: returns index of node's left child
- right(i)=2i+1: returns index of node's right child



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

No pointers required!
Height of a binary heap is O(logn)

Heap Operations

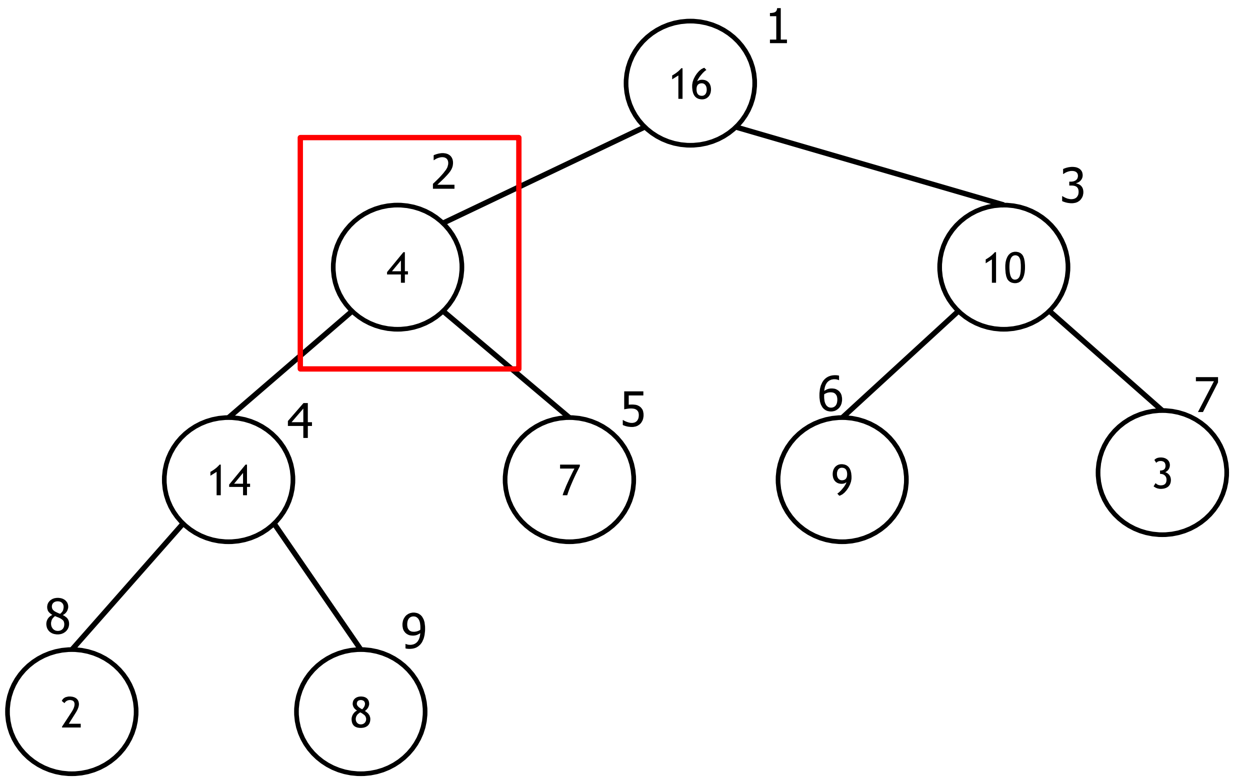
- Max_heapify: correct a single violation of the heap property in a subtree at its root
- Build_max_heap: produce a max-heap from an unordered array
- Insertion: add a new item in the heap
- Deletion: delete an item from the heap

Heapify

- If element A[i] violates the max-heap property, MAX-HEAPIFY lets the value at A[i] “float down” in the max-heap so that the subtree rooted at index i obeys the max-heap property.

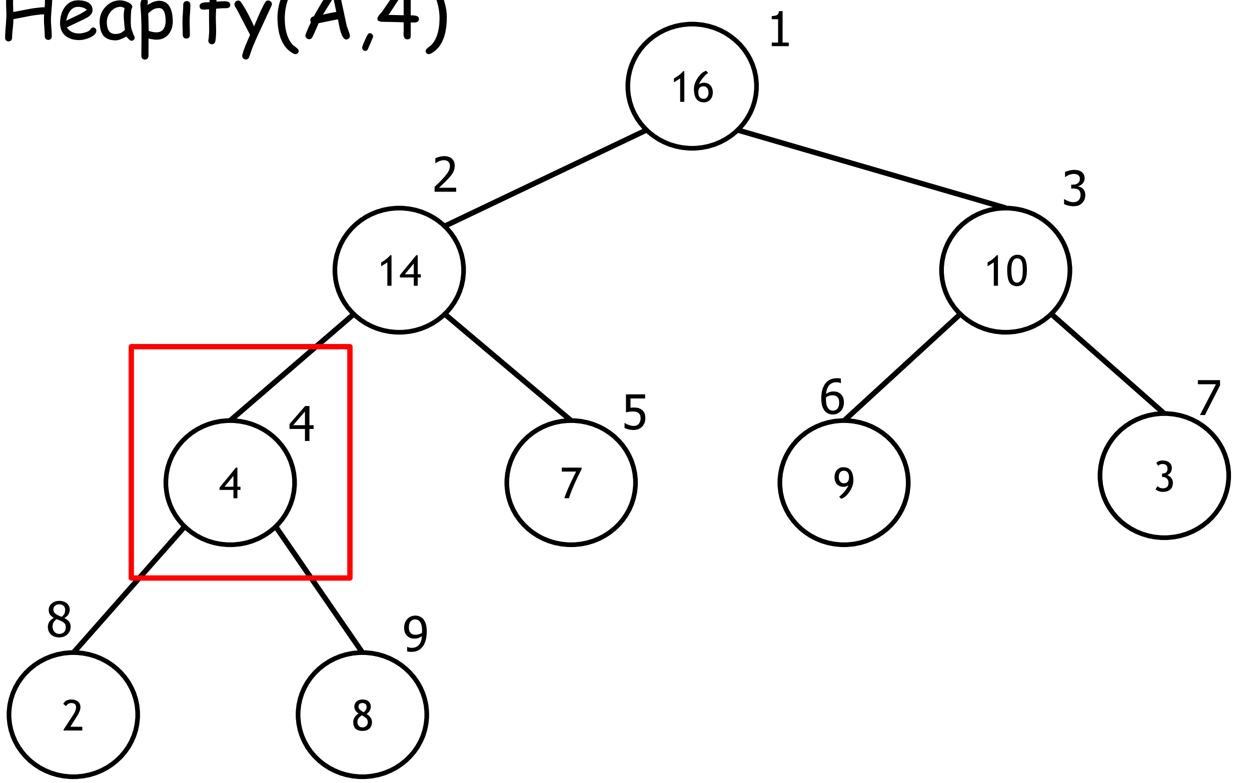
Heapify

- MAX-Heapify(A,2)



Heapify

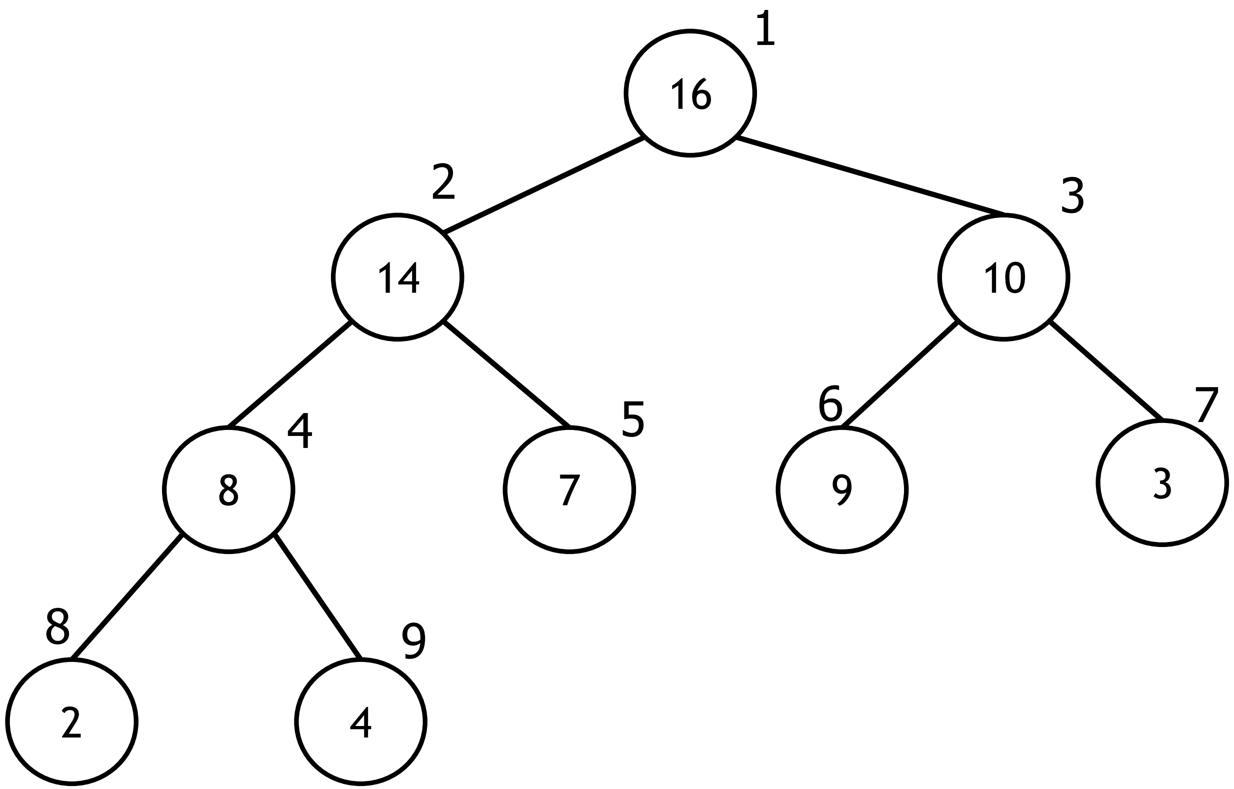
- Exchange A[2] with A[4]
- Then, call MAX-Heapify(A,4)



Heapify

- Time Complexity ?

O(logn)



MAX-HEAPIFY Pseudocode

```
Max_Heapify(A,i):
l = left(i)
r = right(i)
if (l <= heap-size(A) and A[l] > A[i])
        largest = l
else largest = i
if (r <= heap-size(A) and A[r] > A[largest])
        largest = r
if largest ≠ i
        exchange A[i] and A[largest]
        Max_Heapify(A, largest)
```

Build_Max_Heap(A)

- Converts A[1…n] to a max heap

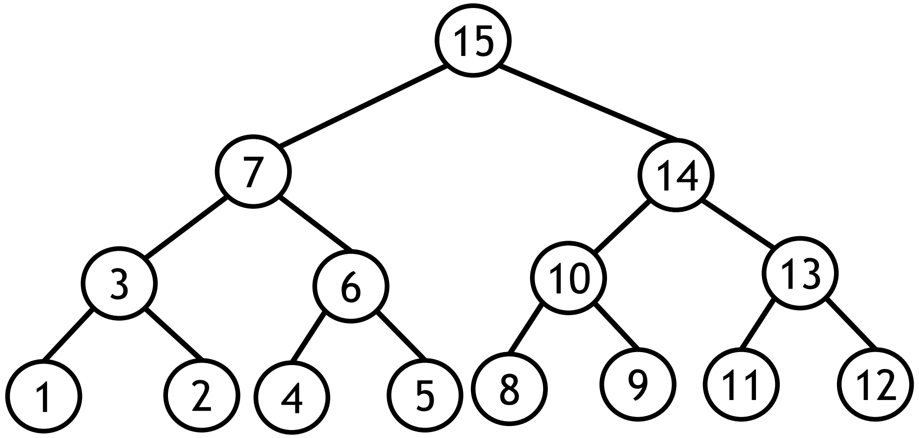
```
Build_Max_heap(A):
for i=n/2 down to 1
        do Max_heapify(A,i)
```

- Why start at n/2?
  - A[n/2+1,….n] are all leaves of the tree

- Time Complexity?
  - O(nlogn) ???

Build_Max_Heap(A) Analysis

- Converts A[1…n] to a max heap

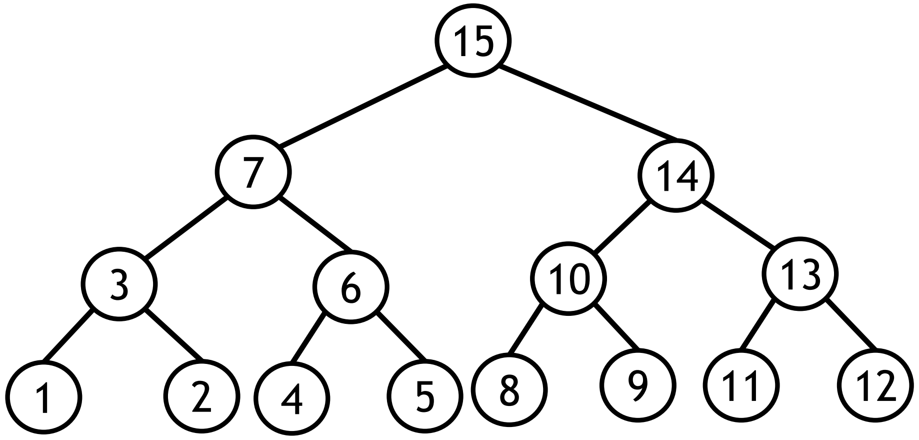



```
Build_Max_heap(A):
for i=n/2 down to 1
        do Max_heapify(A,i)
```

- Max_heapify takes
  - O(1) time for nodes that are one level above the leaves
  - O(k) time for the nodes that are k levels above the leaves
- How many nodes we have in each level?
  - 1 node, 2 nodes, 4 nodes,…, n/4, n/2 nodes

Build_Max_Heap(A) Analysis

- Converts A[1…n] to a max heap



Build_Max_heap(A):
for i=n/2 down to 1
do Max_heapify(A,i)

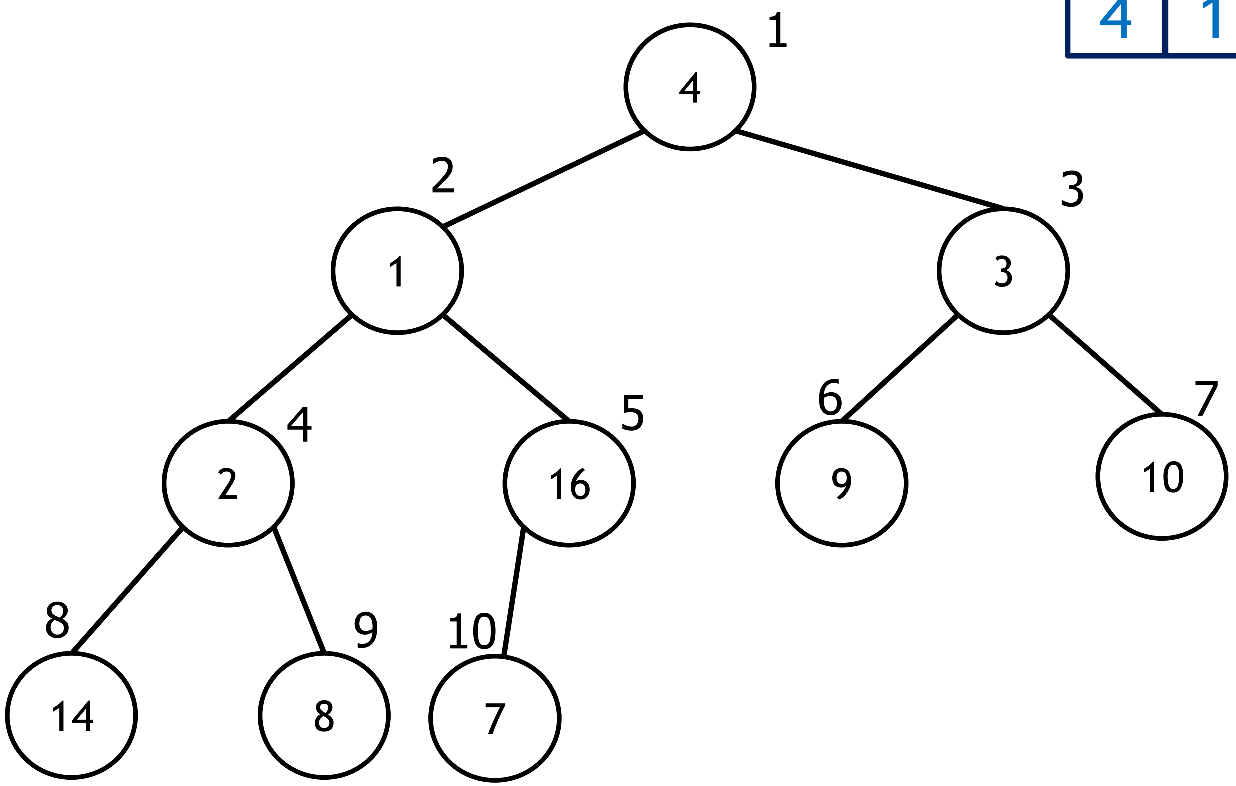
- Total amount of work in the for loop:
- $\mathrm{T(n)} = \sum_{h=0}^{\lfloor logn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \boxed{\sum_{h=0}^{\lfloor logn \rfloor} \frac{h}{2^h}}\right)$

Bounded by a Constant!

- Build Max Heap is O(n).

Build_Max_Heap

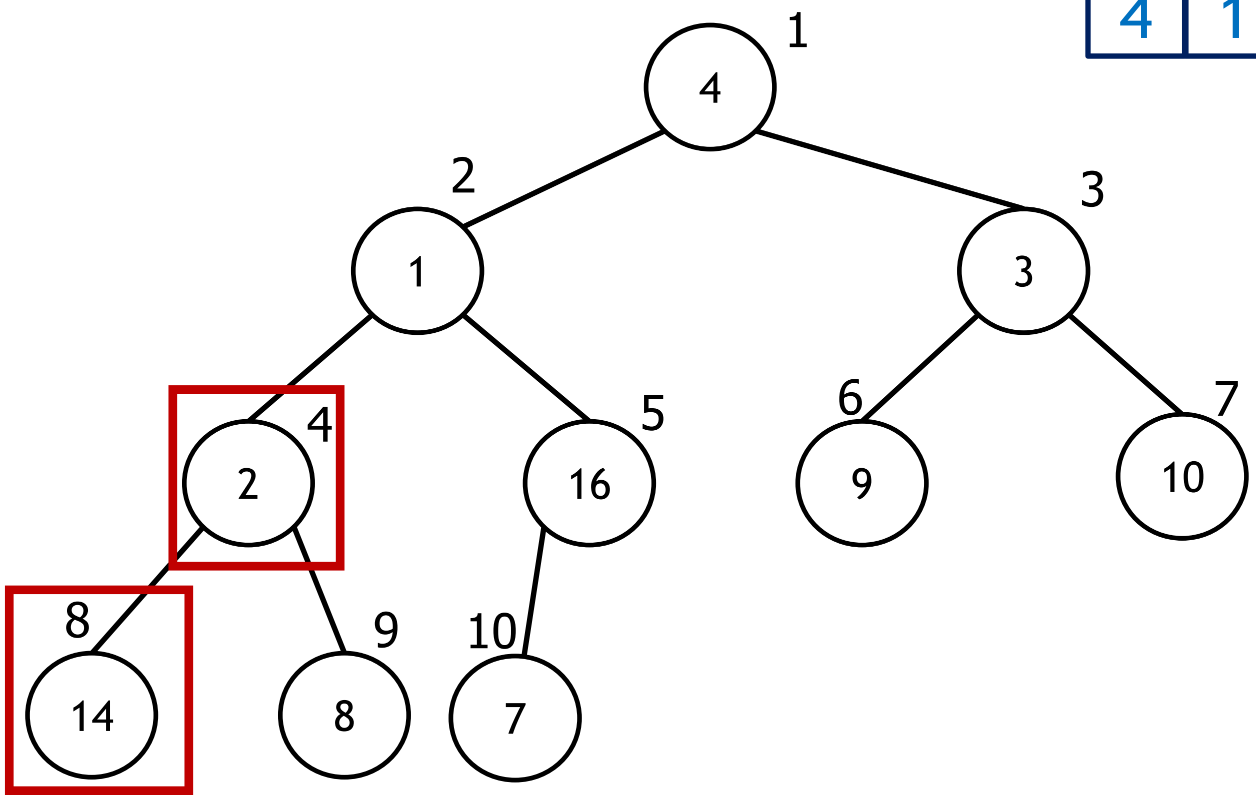
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Max-Heapify(A,5), no change

Build_Max_Heap

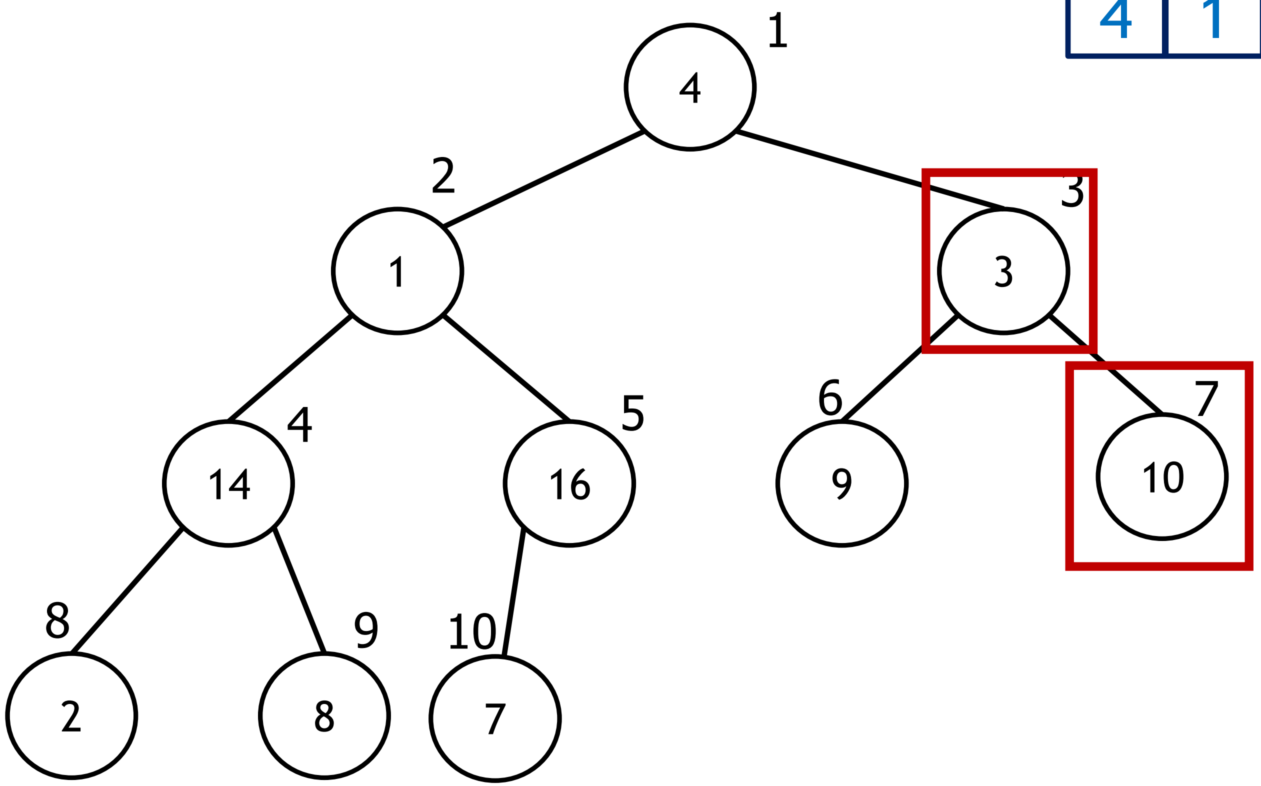
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Max-Heapify(A,4),
Swap A[4] and A[8]

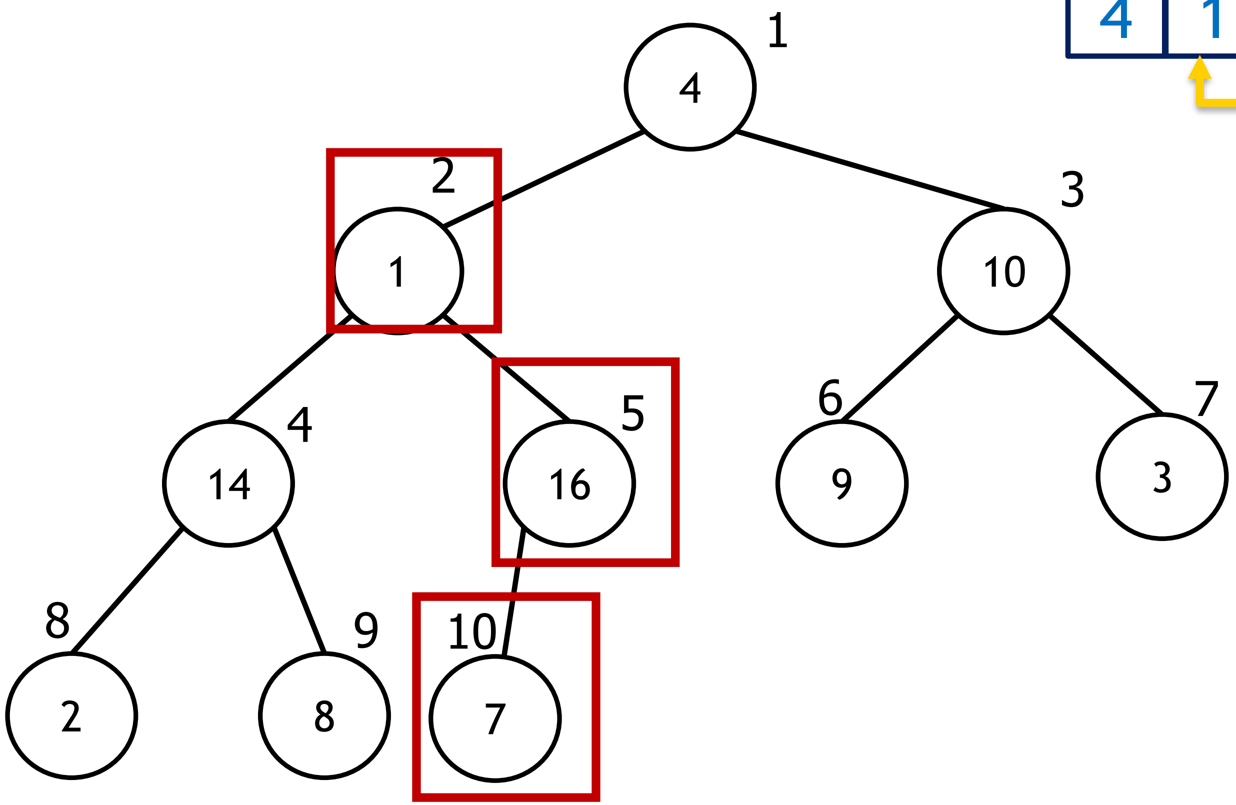
Build_Max_Heap

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



Max-Heapify(A,3),
Swap A[3] and A[7]

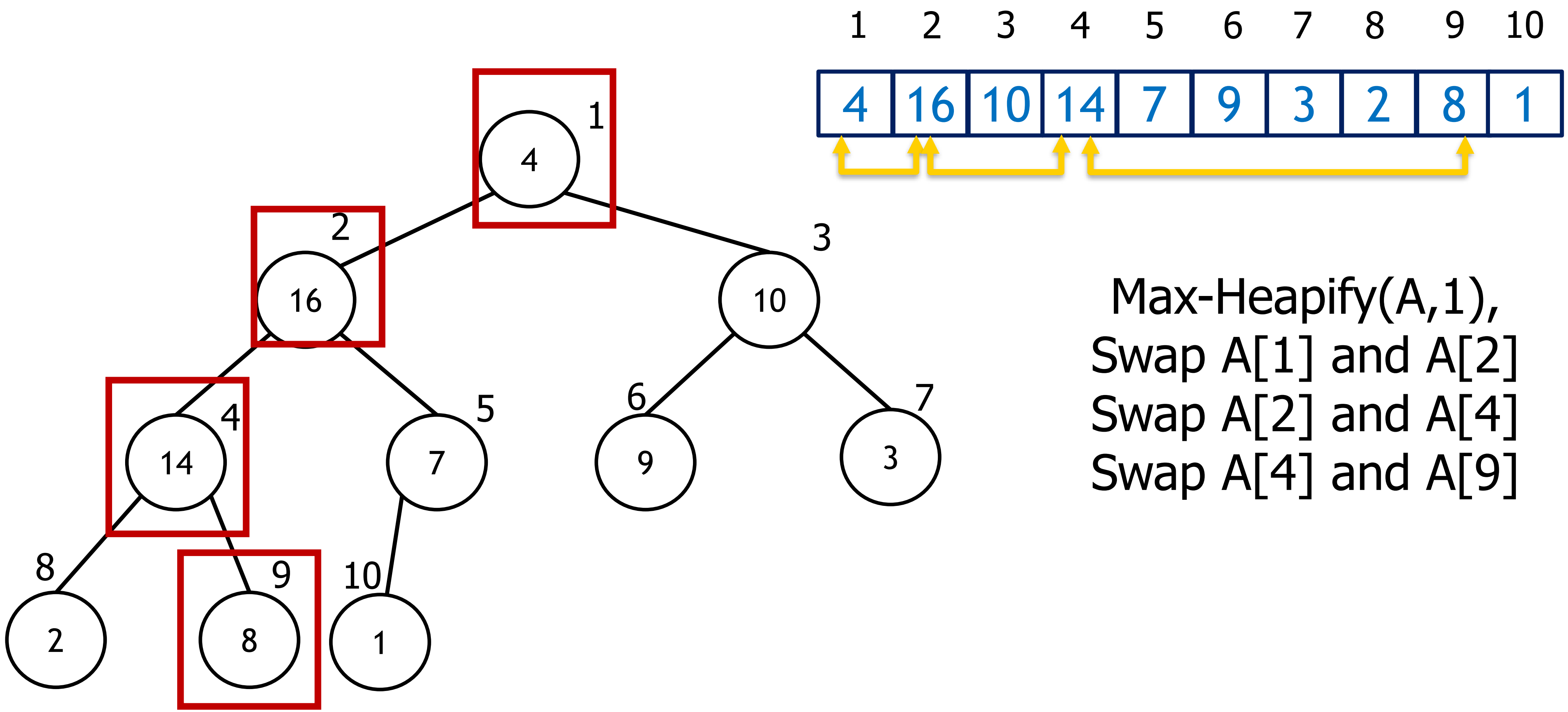
Build_Max_Heap



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

Max-Heapify(A,2),
Swap A[2] and A[5]
Swap A[5] and A[10]

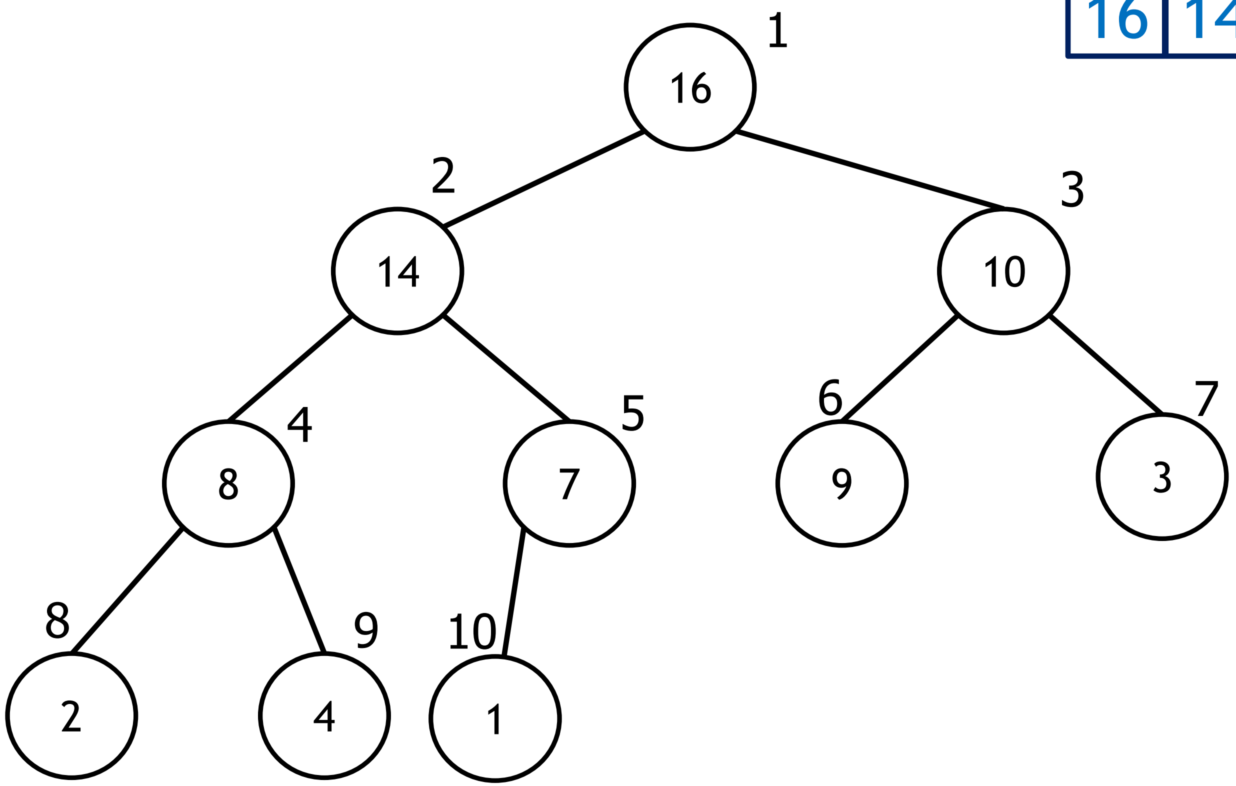
Build_Max_Heap



Max-Heapify(A,1),
Swap A[1] and A[2]
Swap A[2] and A[4]
Swap A[4] and A[9]

Build_Max_Heap

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



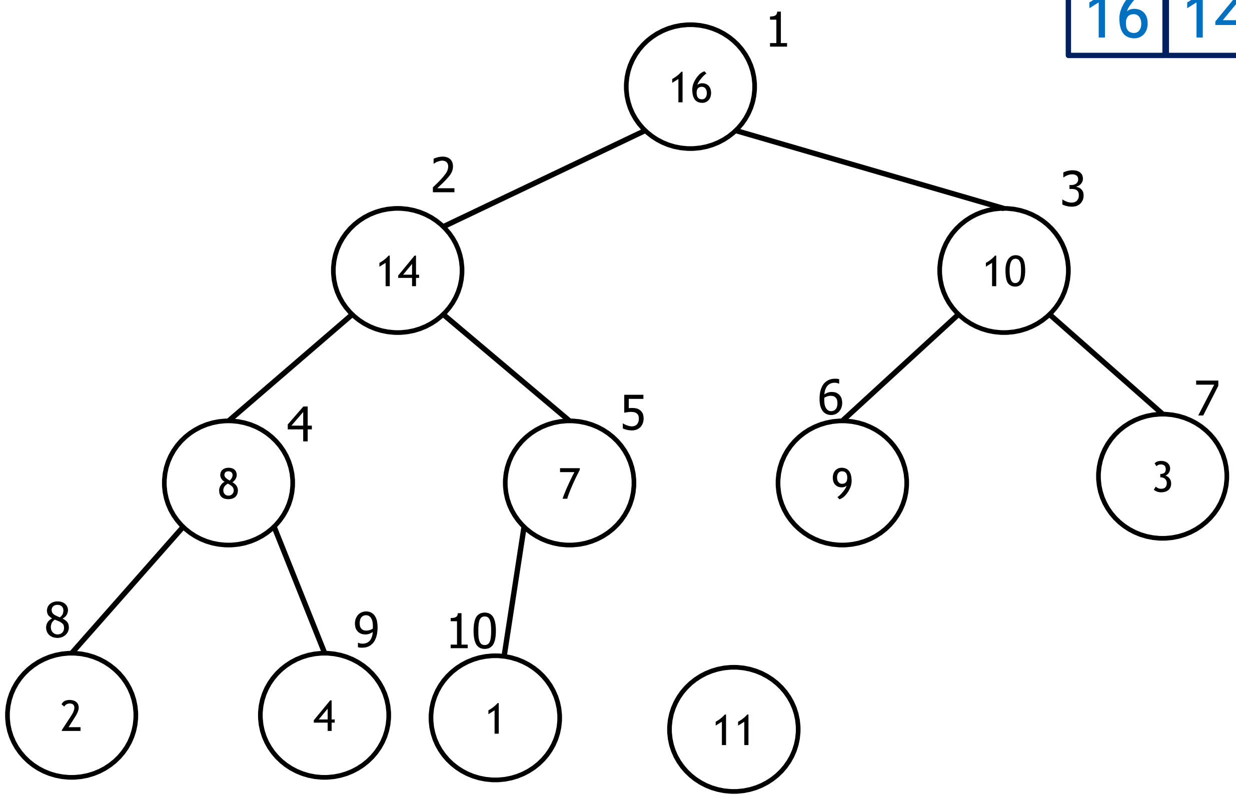
Heap Insertion

- To add an element to a heap:
  - 1) Add the element to the bottom level of the heap at the leftmost open space.
  - 2) Compare the added element with its parent; if they are in the correct order, stop.
  - 3) If not, swap the element with its parent and return to the previous step.

Heap Insertion

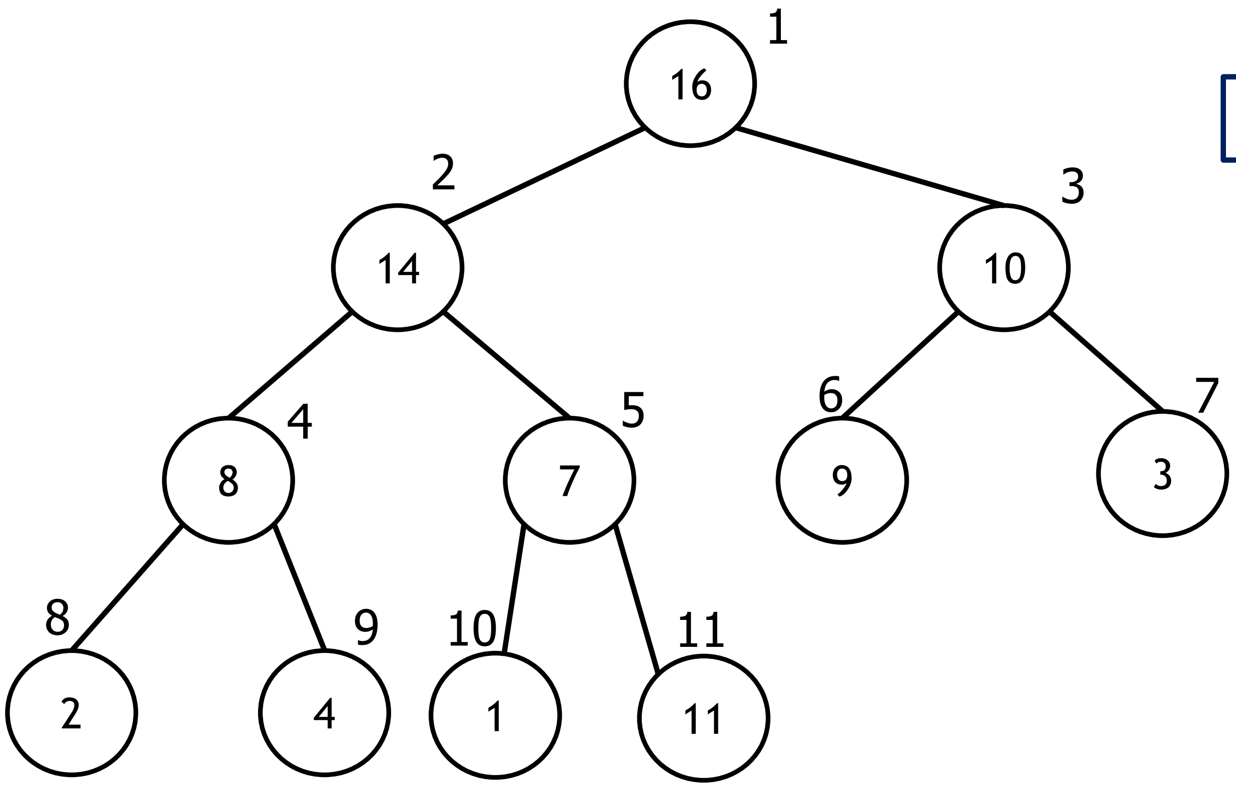
- Insert 11

1	2	3	4	5	6	7	8	9	10	
16	14	10	8	7	9	3	2	4	1	11



Heap Insertion

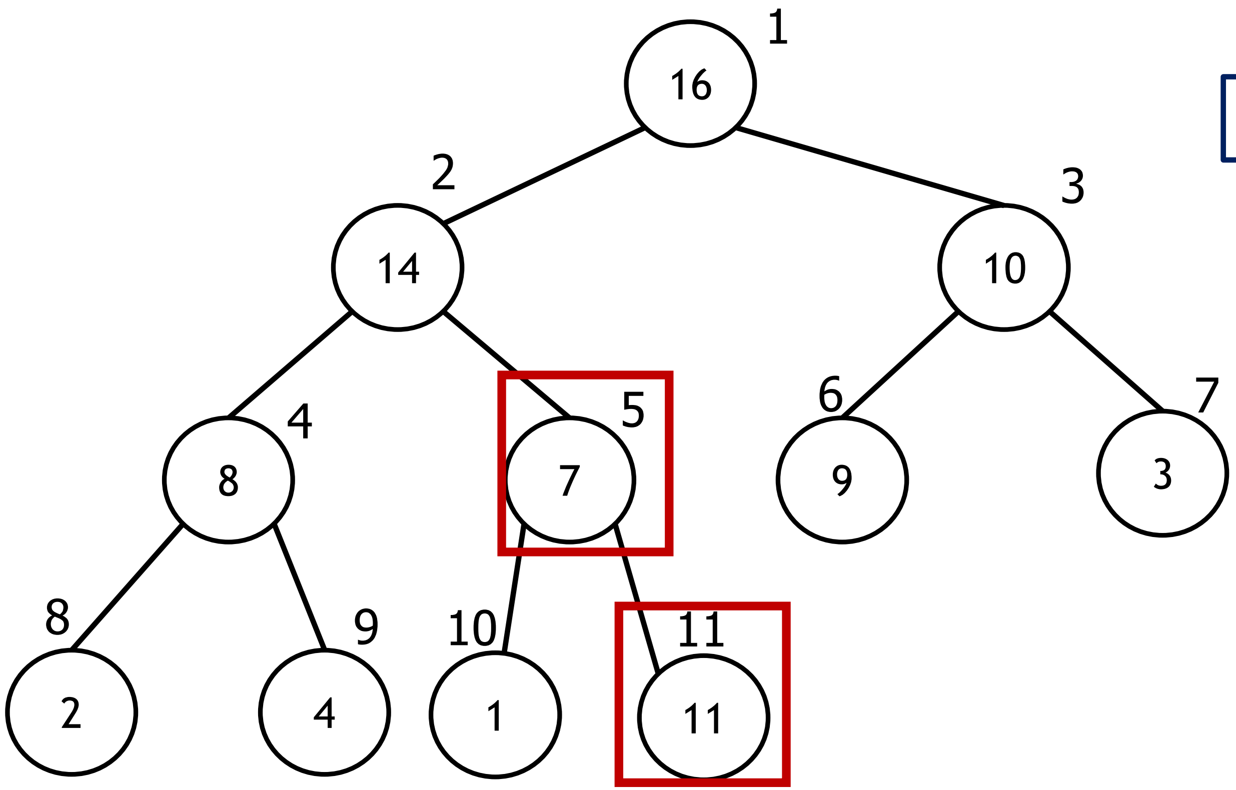
Add the element to the bottom level of the heap at the leftmost open space.



1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	11

Heap Insertion

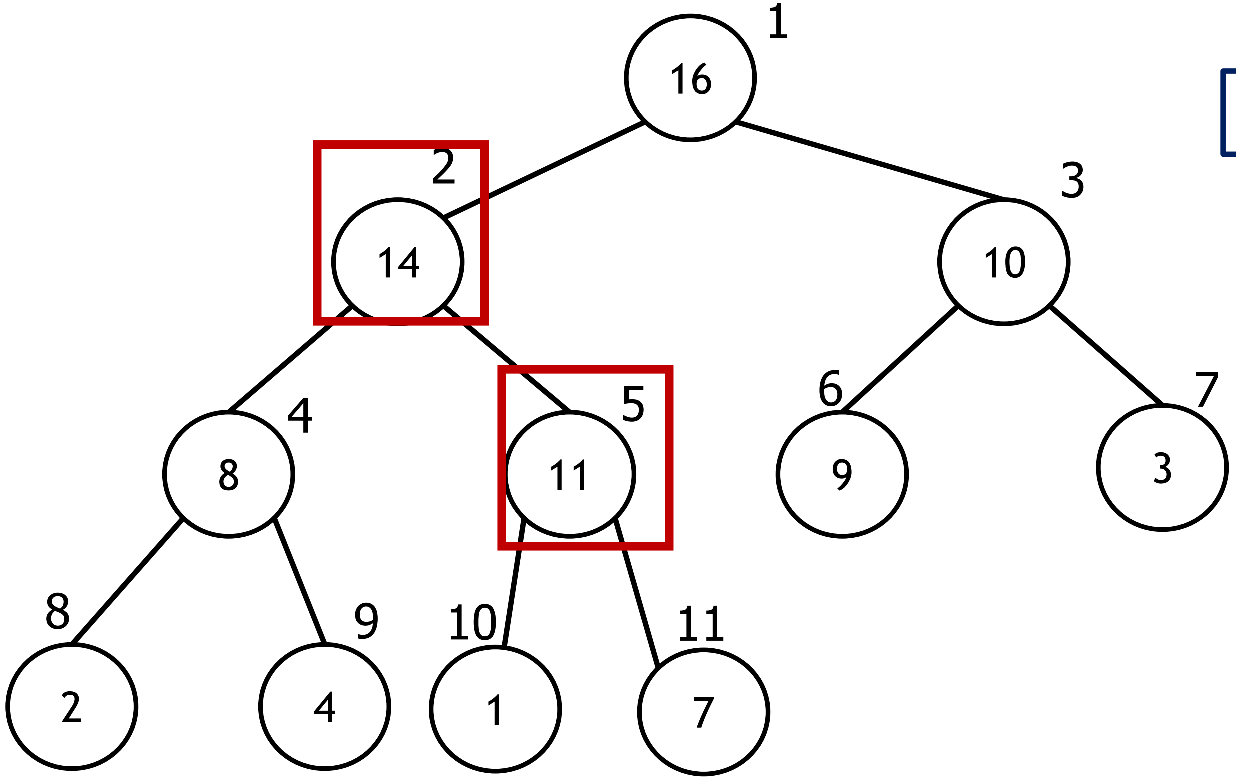
Compare the added element with its parent



1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	11

Heap Insertion

Compare the added element with its parent



1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	11	9	3	2	4	1	7

Correct order, Stop

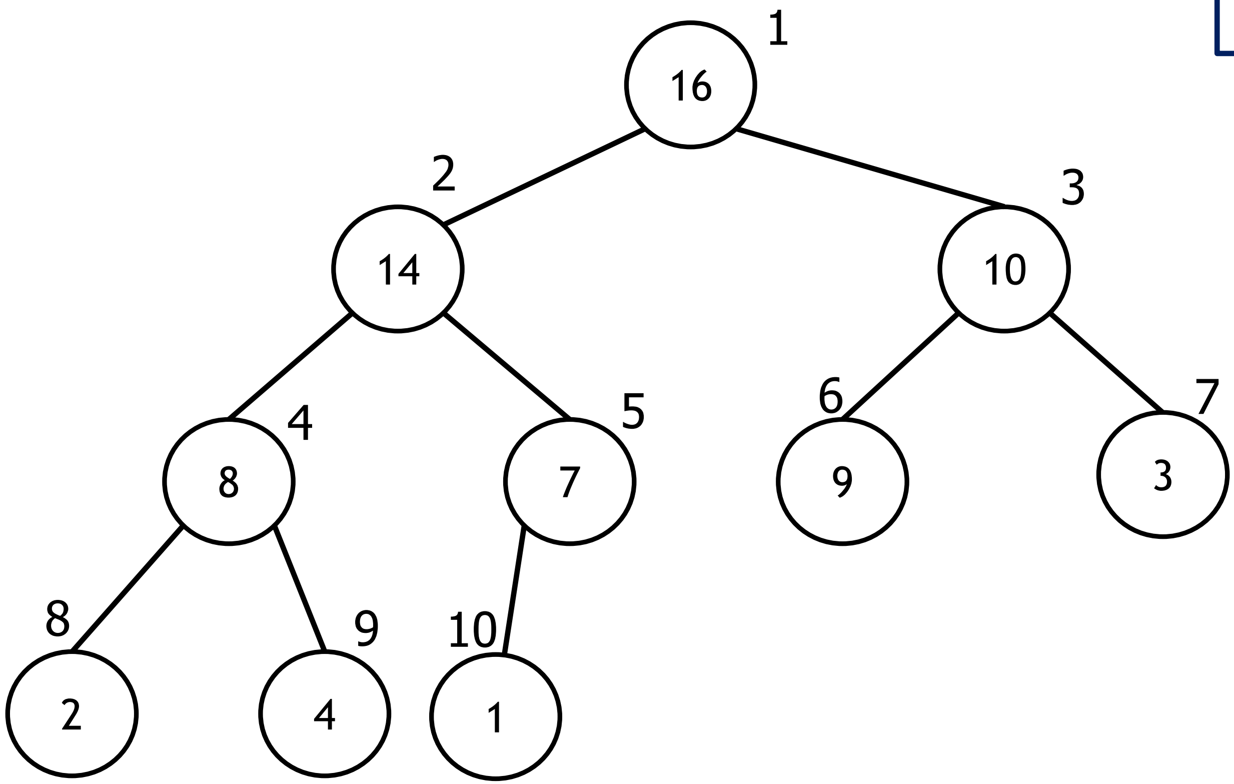
Heap Deletion

- To delete an element from a heap:
  - 1) Find the index i of the element we want to delete
  - 2) Swap this element with the last element
  - 3) Max-Heapify(A,i)

Heap Deletion

- Delete 14

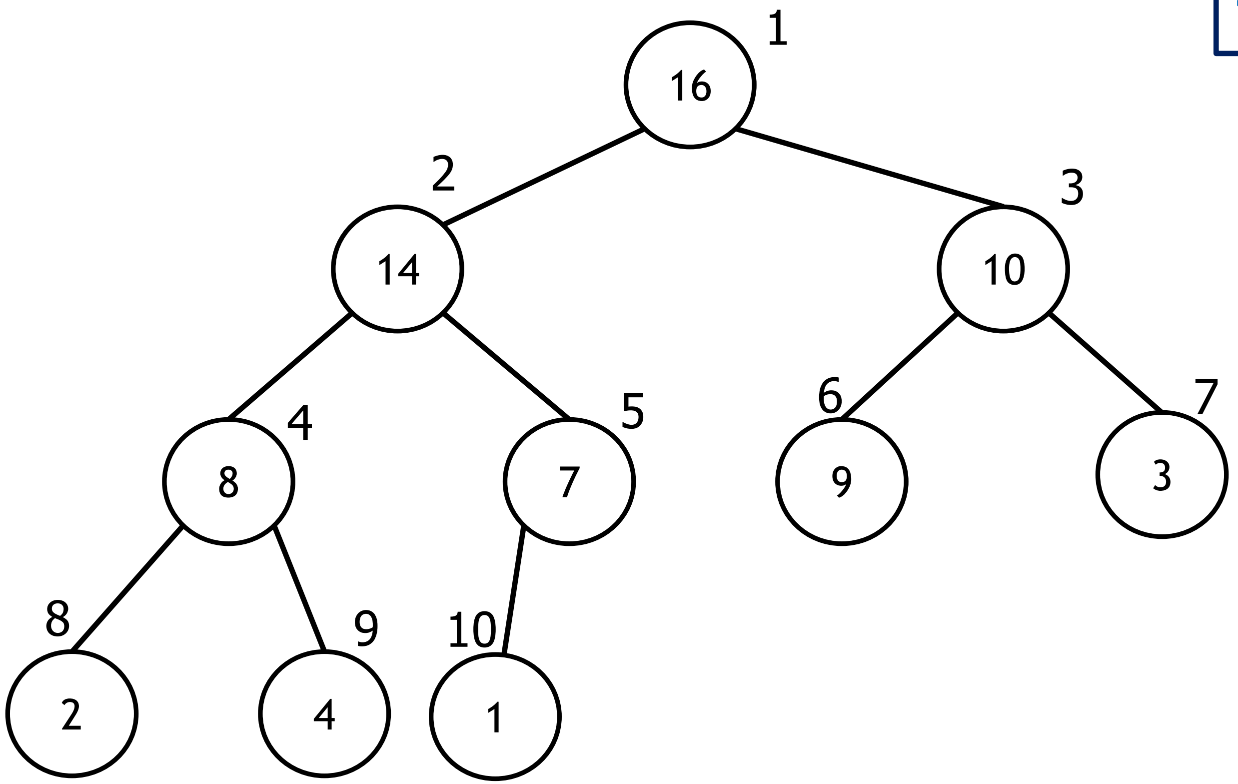
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Heap Deletion

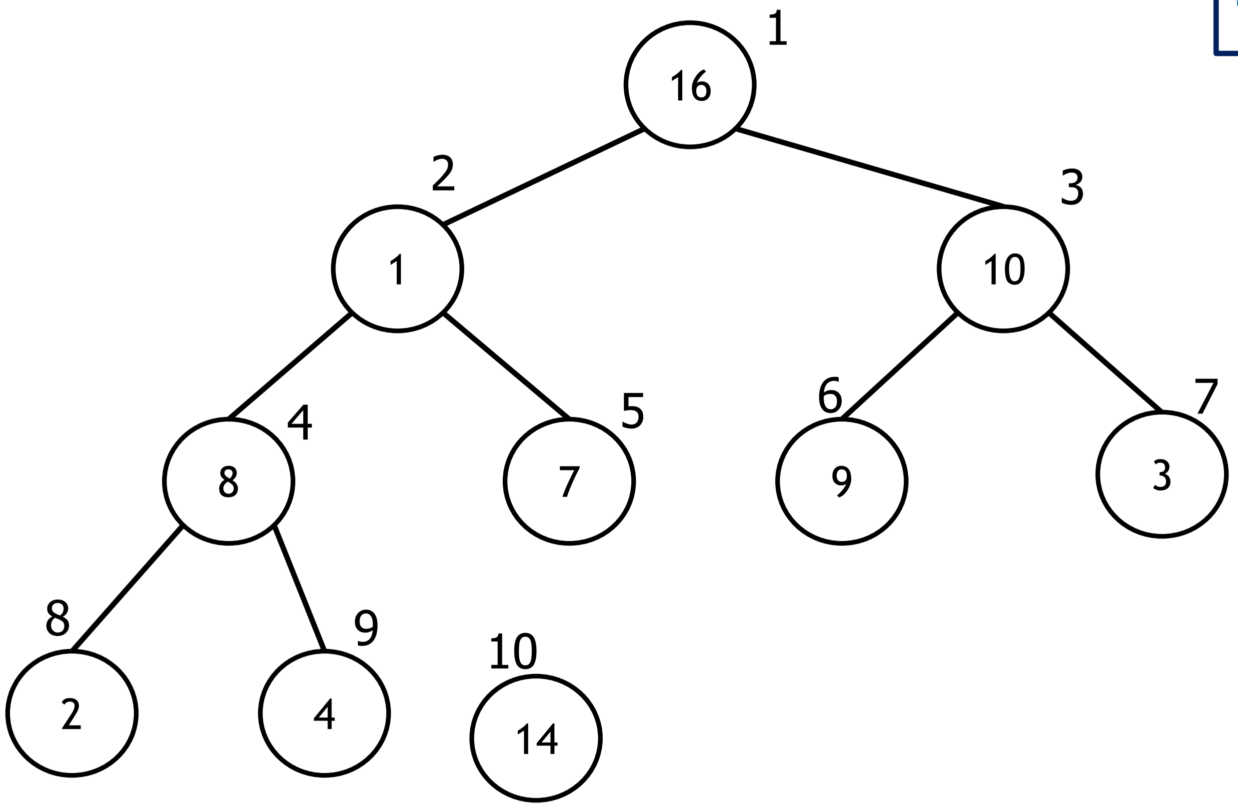
- Swap 14 with the last element

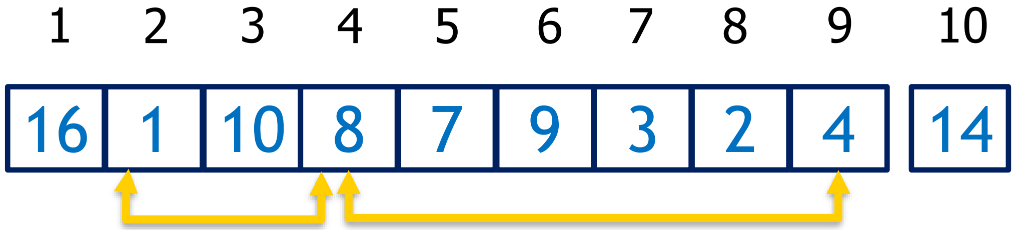
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Heap Deletion

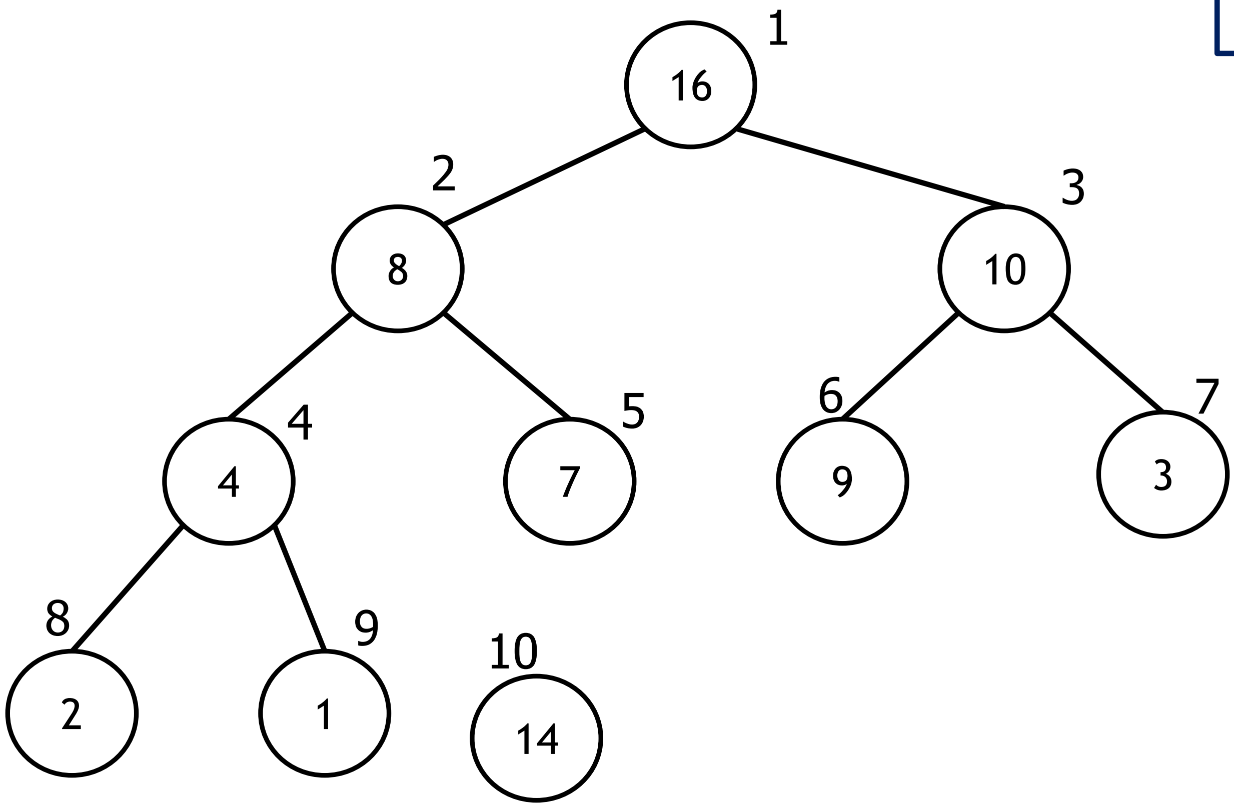
- MAX-heapify(A,2)





Swap A[2] and A[4]
Swap A[4] and A[9]

Heap Deletion



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

Heapsort

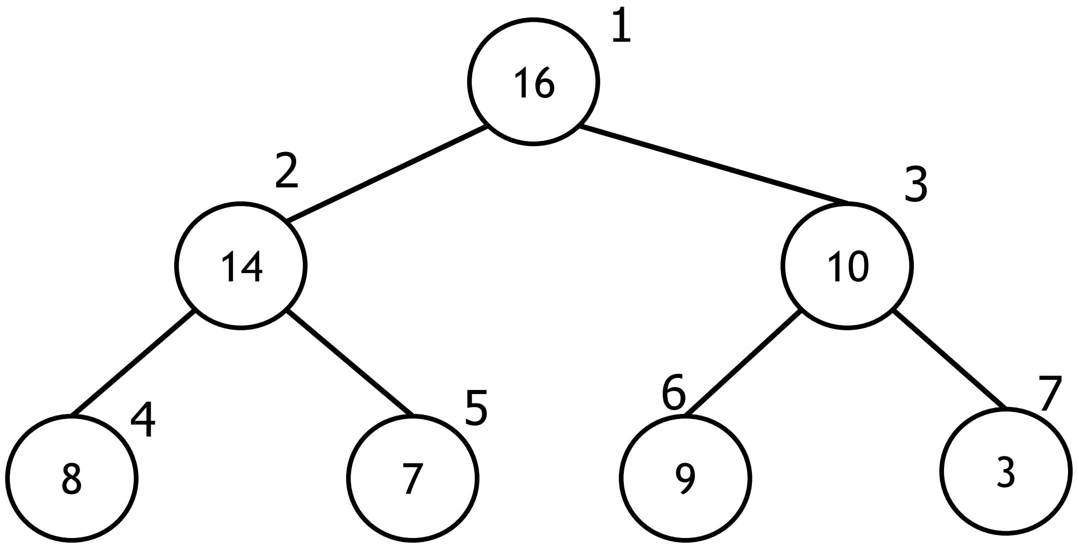
- To sort an array using Heapsort:
  - 1) Build Max Heap
  - 2) Delete the root iteratively

```
Heapsort(A)
Build_Max_Heap(A)
for i =A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size – 1
        Max_Heapify(A,1)
```

Heapsort

- Delete 16

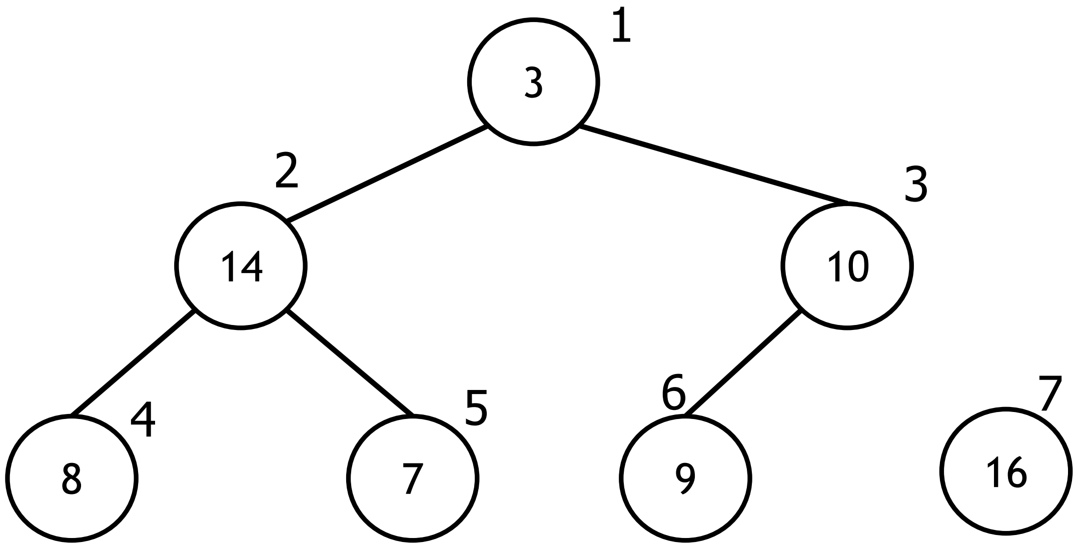
1	2	3	4	5	6	7
16	14	10	8	7	9	3



Heapsort

- Delete 16

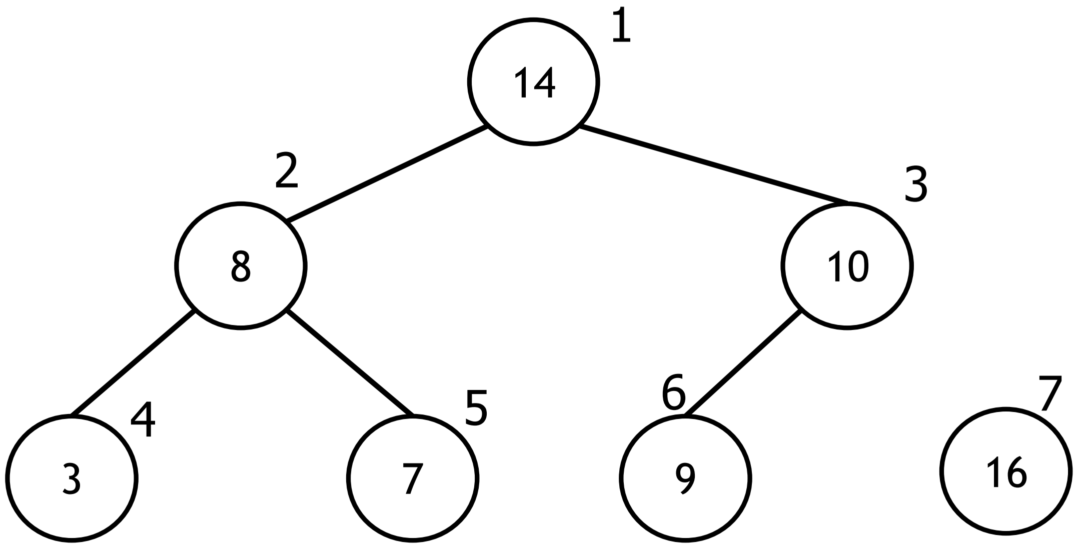
1	2	3	4	5	6	7
3	14	10	8	7	9	16



Heapsort

- Heapify(A,1)

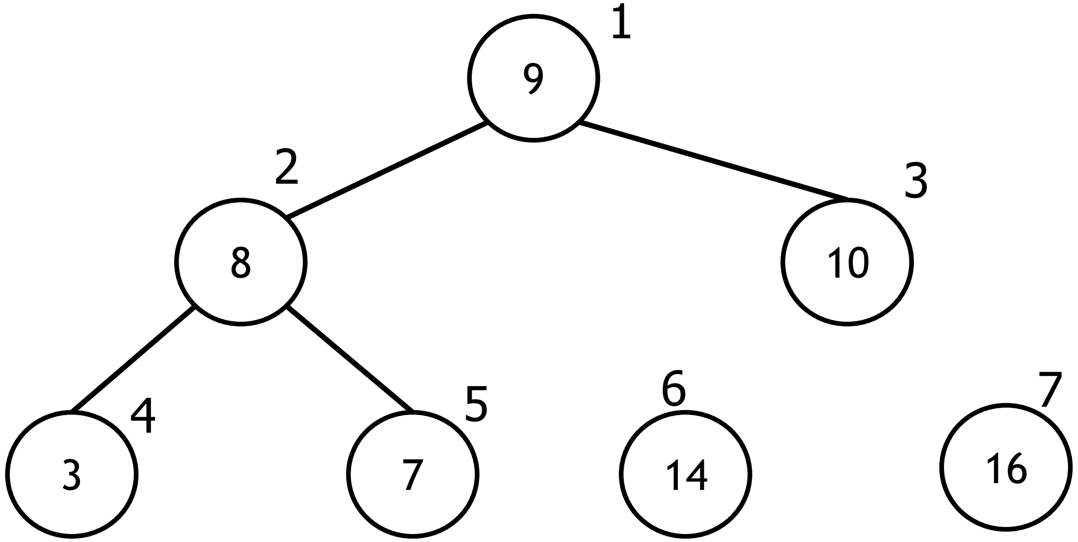
1	2	3	4	5	6	7
14	8	10	3	7	9	16



Heapsort

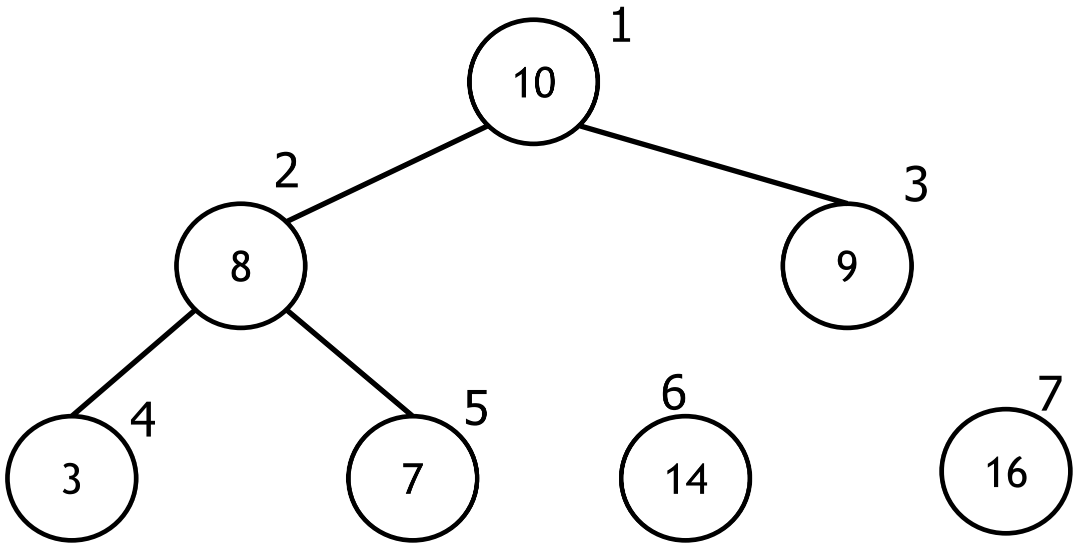
- Delete 14

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



Heapsort

- Heapify(A,1)

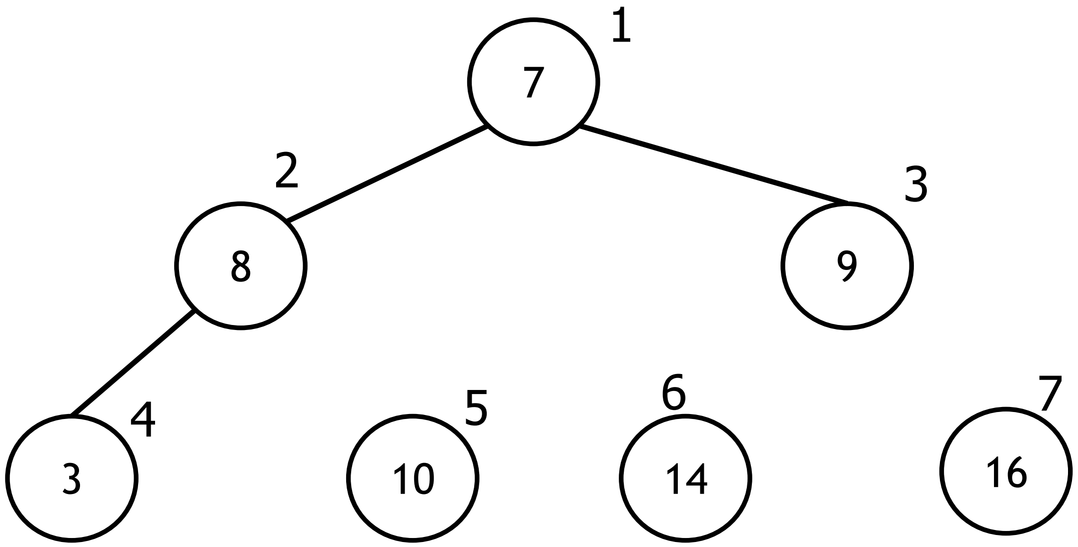


1	2	3	4	5		6	7
10	8	9	3	7		14	16

Heapsort

- Delete 10

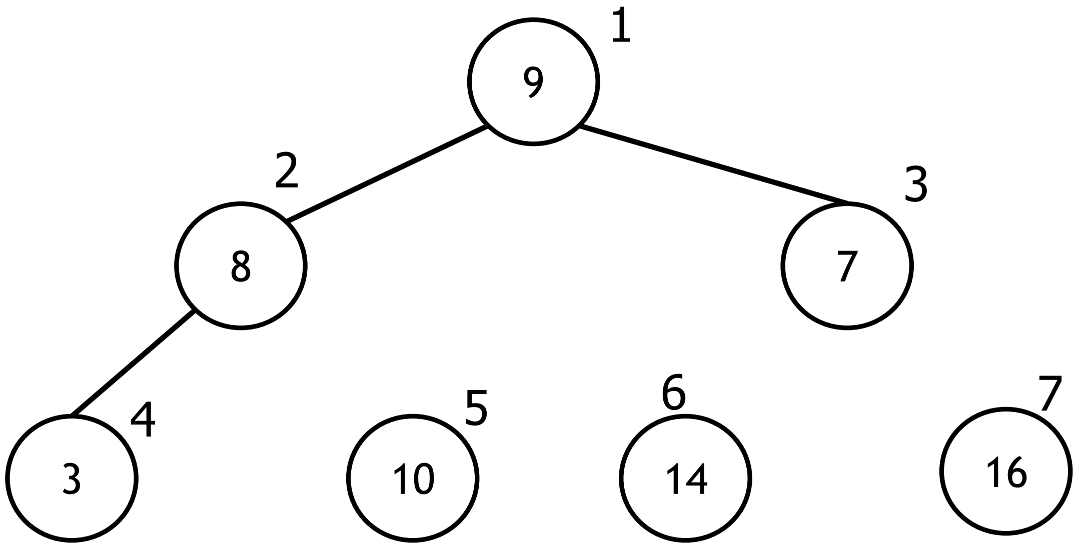
1	2	3	4		5	6	7
7	8	9	3		10	14	16



Heapsort

- Heapify(A,1)

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

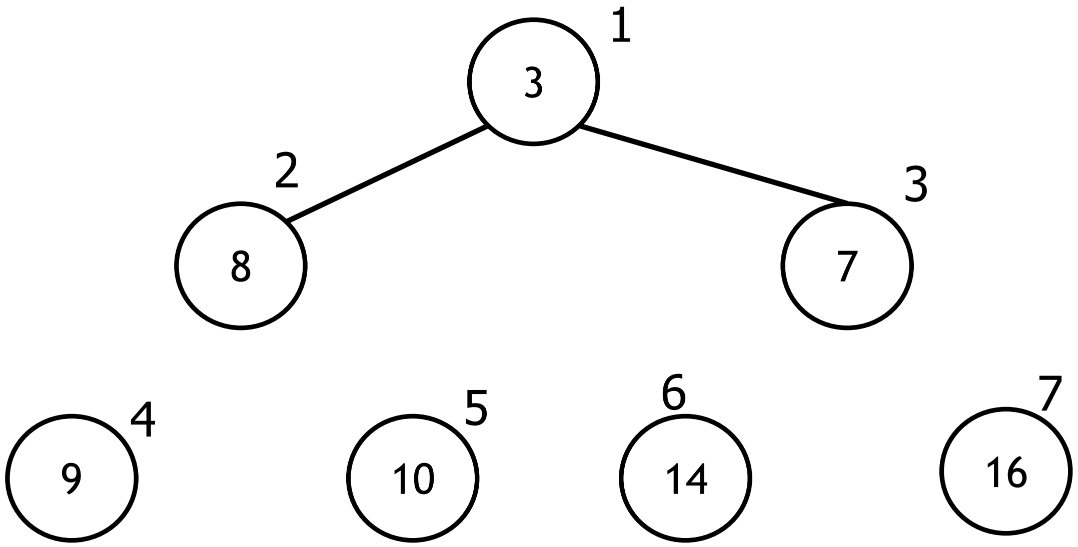


Heapsort

- Delete 9

1	2	3
3	8	7

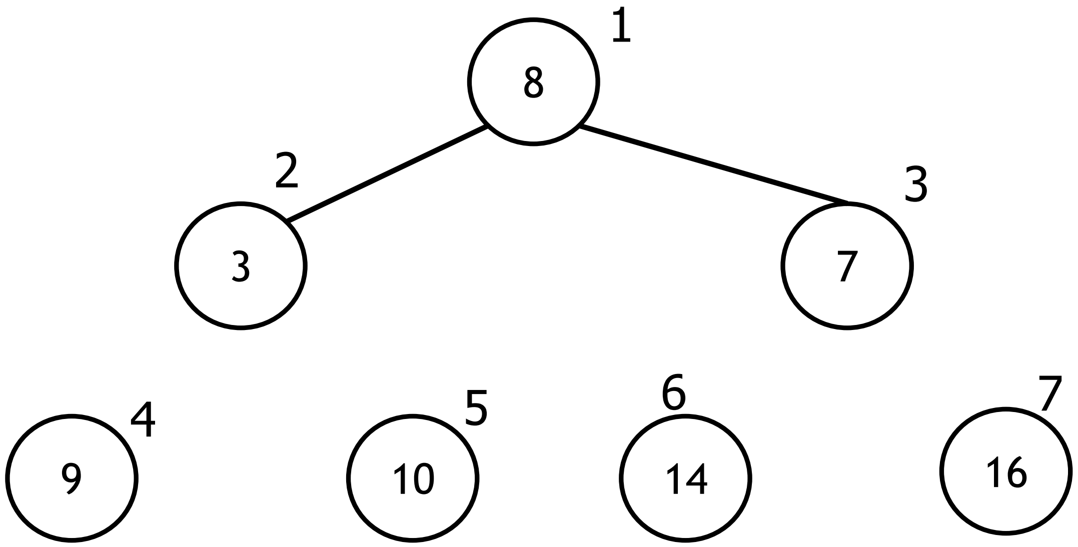
4	5	6	7
9	10	14	16



Heapsort

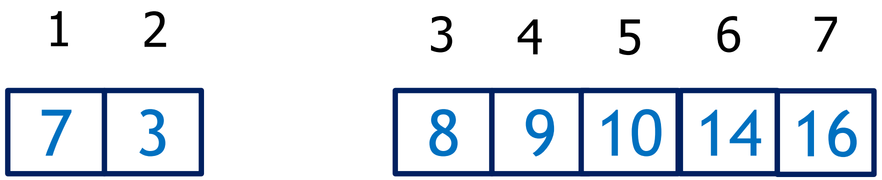
- Heapify(A,1)

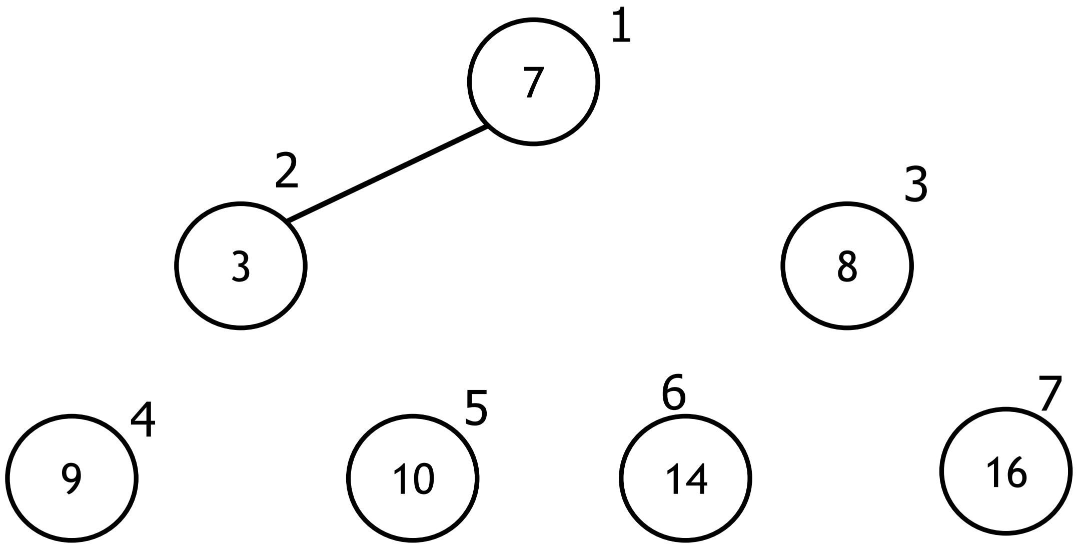
1	2	3	4	5	6	7
8	3	7	9	10	14	16



Heapsort

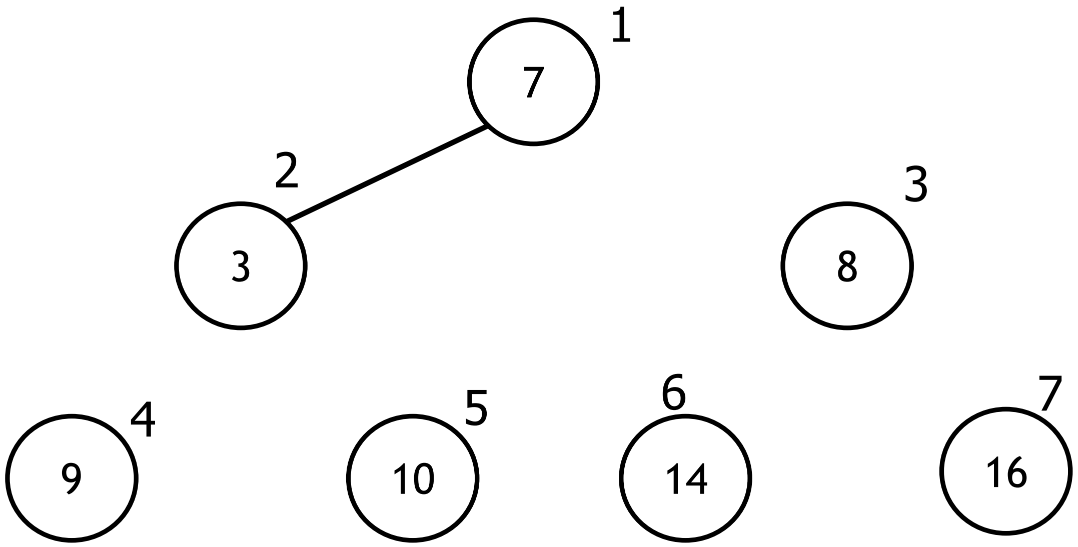
- Delete 8

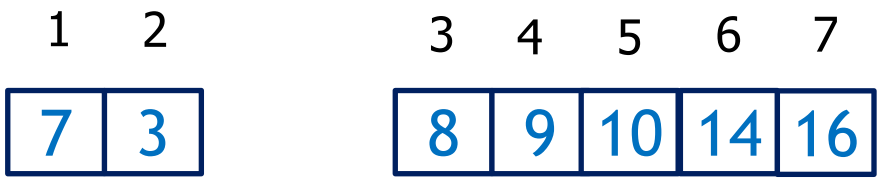




Heapsort

- Heapify(A,1)

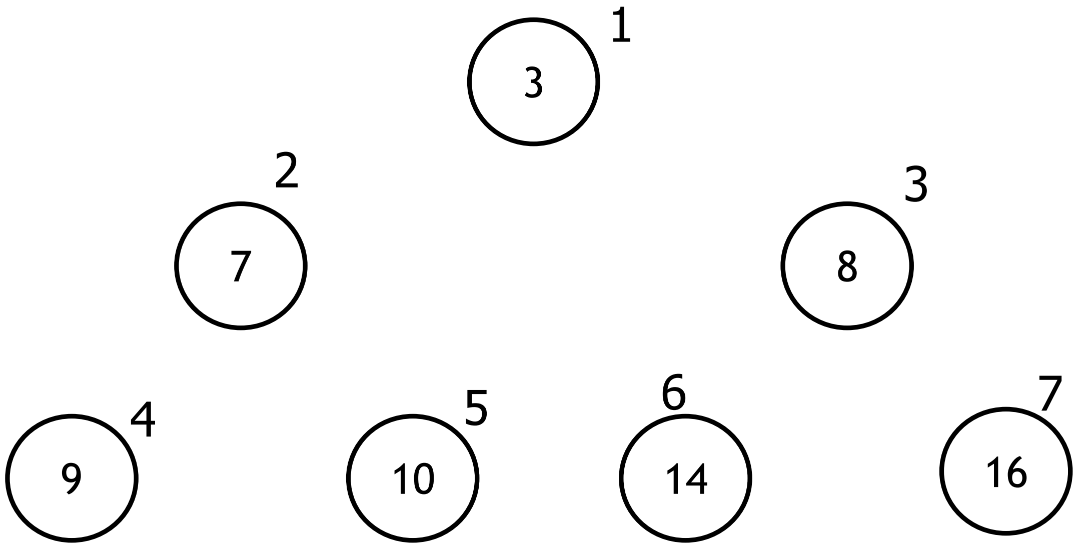




Heapsort

- Delete 7

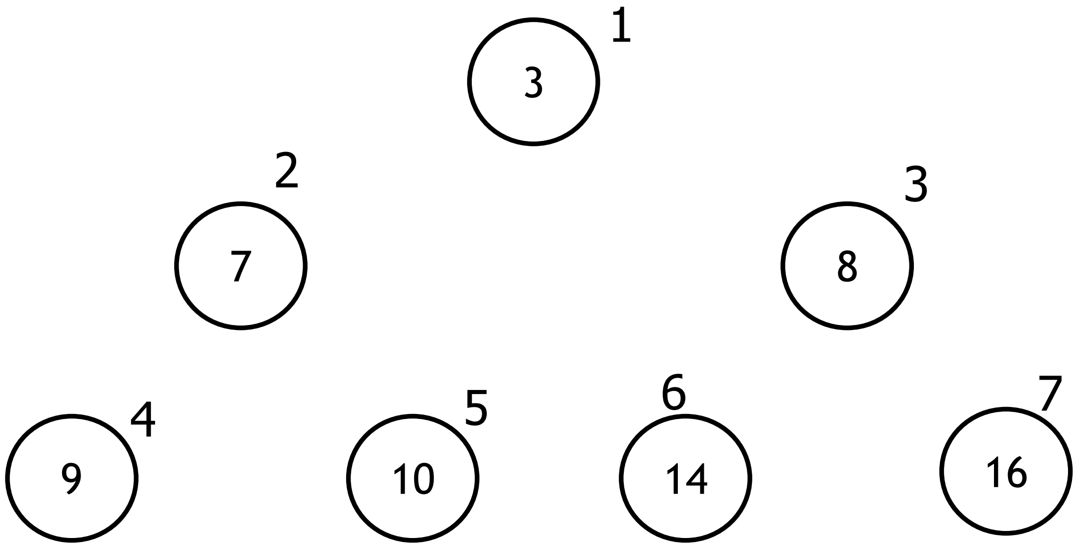
1	2	3	4	5	6	7
3	7	8	9	10	14	16



Heapsort

- Finished

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



Heapsort

- Time Complexity?
  - n iterations
  - Heapify takes O(logn)

O(nlogn)

Priority Queue

- A data structure implementing a set S of elements, each associated with a key, supporting the following operations:
  - insert(S, x) : insert element x into set S
  - max(S) : return element of S with largest key
  - extract_max(S) : return element of S with largest key and remove it from S
  - increase_key(S, x, k) : increase the value of element x' s key to new value k

- One of the most popular applications of a heap: an efficient priority queue.
  - Insertion time complexity? O(logn)
  - Deletion time complexity? O(logn)

Exercise

- Given an array A = [10,20,15,12,40,25,18], show the resulting max heap after the operation of heapify, and draw the corresponding binary tree.

- Given a max heap H = [40,30,15,10,20], draw the intermediate heaps while conducting the heap sort algorithm.

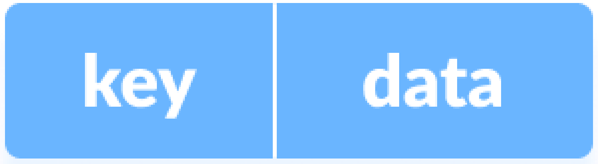
Hash Tables

Hash Tables

- Another kind of Table
- Use an array named T of capacity m
- Define a hash function that returns an integer h(k)
  - Must return an integer between 0 and m-1
  - h() must always return the same integer for a given key
- Store the key and info at T[h(k)]
- O(1) on average for insert, lookup, and remove

Hash Tables

- The Hash table data structure stores elements in key-value pairs:
  - Key – is used for indexing the values
  - Value – data that are associated with keys



What is a Hash function?

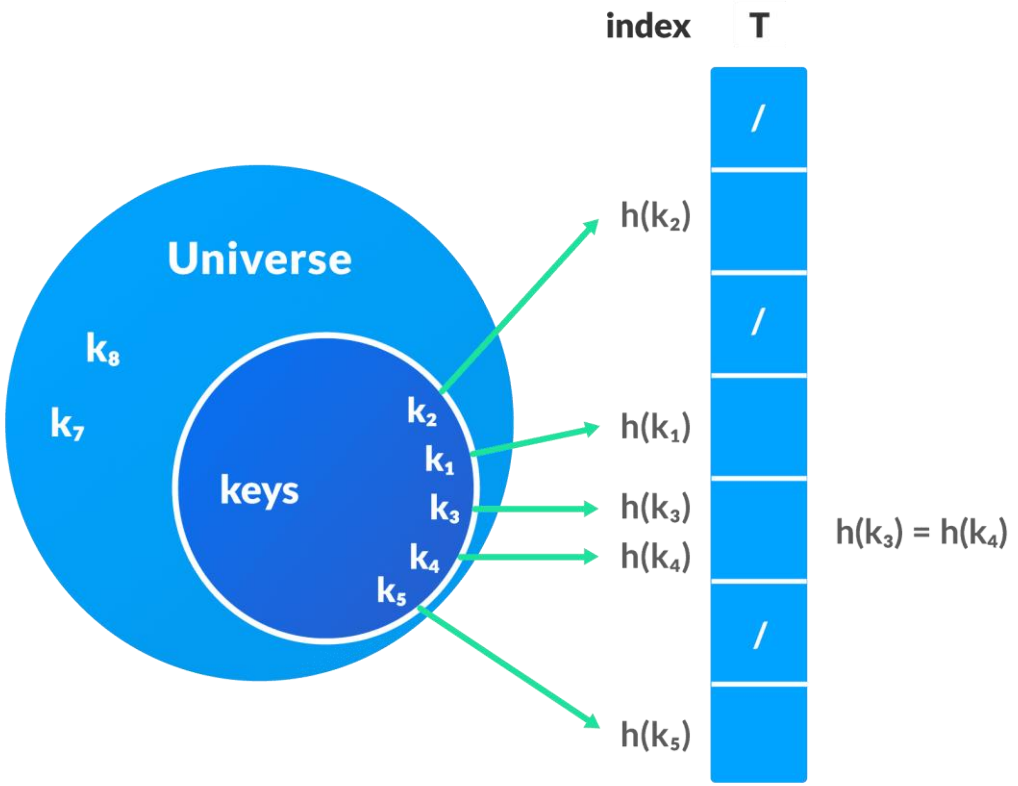
- A hash function is any well-defined procedure or mathematical function for mapping data into an index in the table.

- A hash function h is a transformation that
  - Takes a key k and
  - Returns an index h(k), which is called the hash value or simply hashes

- Must return an integer between 0 and m-1
- h(x) must always return the same integer for a given key

Hash Function

- Let k be a key and h(x) be a hash function.
- Here, h(k) will give us a new index to store the element linked with k.



Example of a Modular Hash Function

- The division method:
  - h(k) = k mod m (or k % m)
  - key1 = 1234
  - Hash function = modulo 11
  - Hash value = 1234 mod 11 = 2

  - key2 = "test" = ASCII '74', '65', '73', '74'
  - Hash value = (74+65+73+74) mod 11 = 0
- The multiplication method:
  - floor(m(kA mod 1))

Why Hashing?

- After storing a large amount of data, we need to perform various operations on these data.
  - Linear search and binary search perform lookups/search with time complexity of $O(n)$ and $O(\log n)$ respectively.
  - Hashing allows insert, lookup, and remove to occur in constant time i.e. $O(1)$

Hash table give O(1) performance?

- While you insert, lookup, or delete a key, you first calculate its hash and then you know which exact position the key is in the array. The operation is O(1) as you go directly there and delete the key.

- What if we insert data and there is already something in that position of the array? Hash function doesn't guarantee that it won't produce the same output for two different inputs.

Hash Functions

- A good hash function has the following characteristics:
  - Avoid collisions
  - Spreads keys evenly in the array
  - Inexpensive to compute – must be $O(1)$

Hash Collision

- When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.**

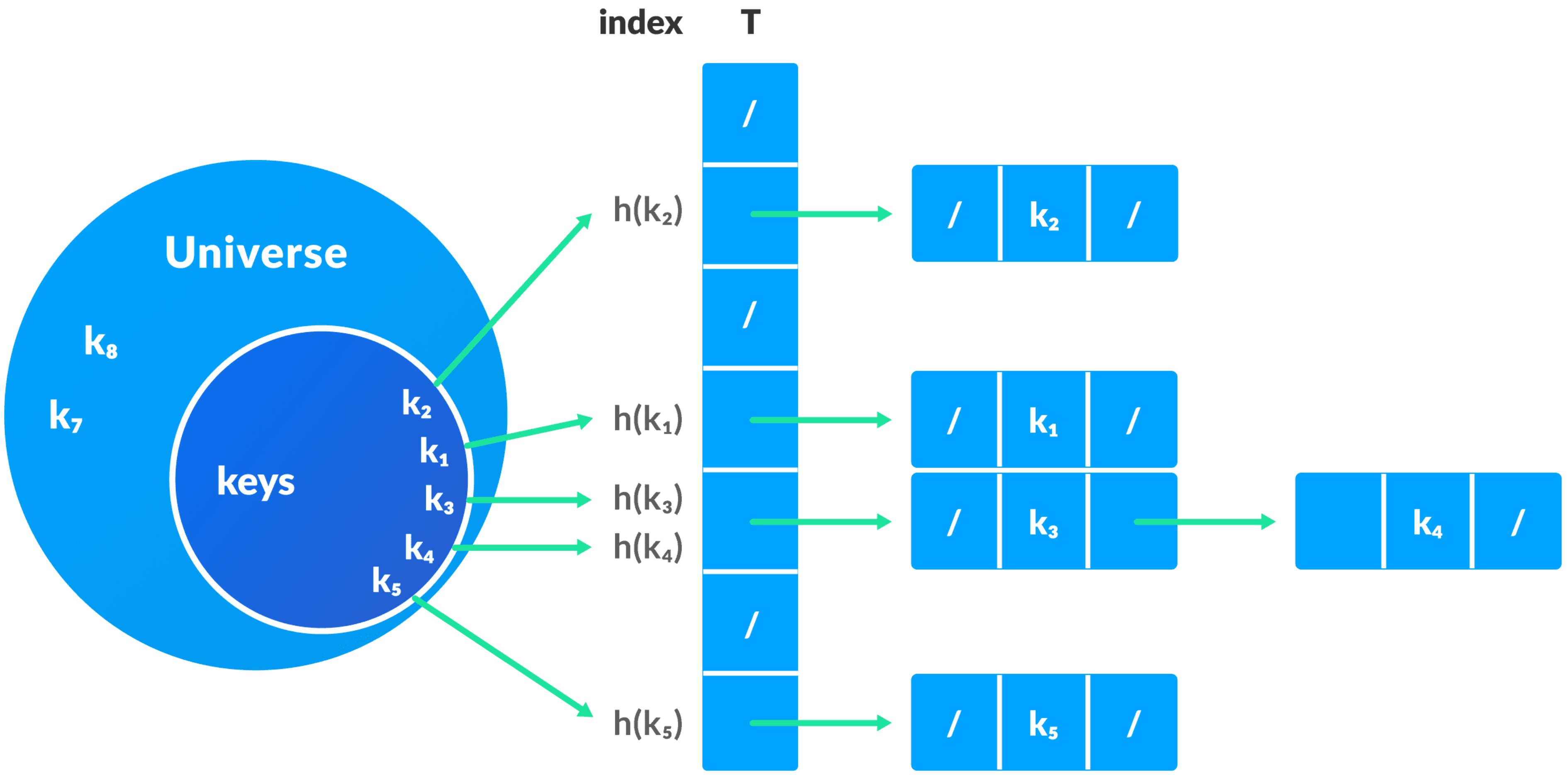
- We can resolve the hash collision using one of the following techniques.
  - Collision resolution by chaining
  - Open Addressing: Linear/Quadratic Probing and Double Hashing

Collision resolution by chaining

- In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.

- If j is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, j is NULL.

Collision resolution by chaining



Open Addressing

- Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left NULL.

- Different techniques used in open addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- In linear probing, collision is resolved by checking the next slot.

$$h(k, i) = (h_1(k) + i) \bmod m$$

where

  - $i = \{0,1,2,\ldots\}$
  - $h_1(k) = k \bmod m$

- If a collision occurs at $h(k,0)$, then $h(k,1)$ is checked. Increment hash value by a constant 1, until free slot is found.

- simplest to implement, but leads to primary clustering.

Linear Probing

- Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Quadratic Probing

- It works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

$$h(k, i) = (h_1(k) + c_1*i + c_2*i*i) \bmod m$$

where

- c_1 and c_2 are postive constants
- $i = \{0, 1, 2, ...\}$

- leads to secondary clustering.

Double Hashing

- If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

$$h(k,i) = (h_1(k)+i*h_2(k)) \bmod m$$

- Avoids clustering

Table Size

- Table size is usually prime to avoid bias
- Overly large table size means wasted space
- Overly small table size means more collisions
- What happens as table size approaches 1?

Dealing with A Full Table

- Allocate a larger hash table
- Rehash each from the smaller into the larger
- Delete the smaller

Applications

- Some of Hash Table applications:
  - Message Digest (MD5, SHA256, …)
  - Password verification
  - Rabin-Karp Algorithm
  - Application requires constant time lookup and insertion
  - Etc.

Learning outcomes

- Heaps
- Heapsort
- Priority queue

- Hash tables
- Hash function
- Collisions
- Table size
- Applications