بسم الله الرحمن الرحیم

ساختمان‌های داده

جلسه ۲۶

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

Recall Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e) inserts an entry e
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better? Balancing the above

Algorithm PriorityQueueSort(L, P):

Input: An STL list L of n elements and a priority queue, P, that compares elements using a total order relation

Output: The sorted list L

```
while !L.empty() do
    e ← L.front
    L.pop_front()        {remove an element e from the list}
    P.insert(e)          {... and it to the priority queue}
while !P.empty() do
    e ← P.min()
    P.removeMin()        {remove the smallest element e from the queue}
    L.push_back(e)       {... and append it to the back of L}
```

List-based vs. Heap-based

IUT-ECE

List-based

Operation	*Unsorted List*	*Sorted List*
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Heap-based

Operation	*Time*
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, empty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
  - Construction: n insertions
  - Actual sorting: n removals
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

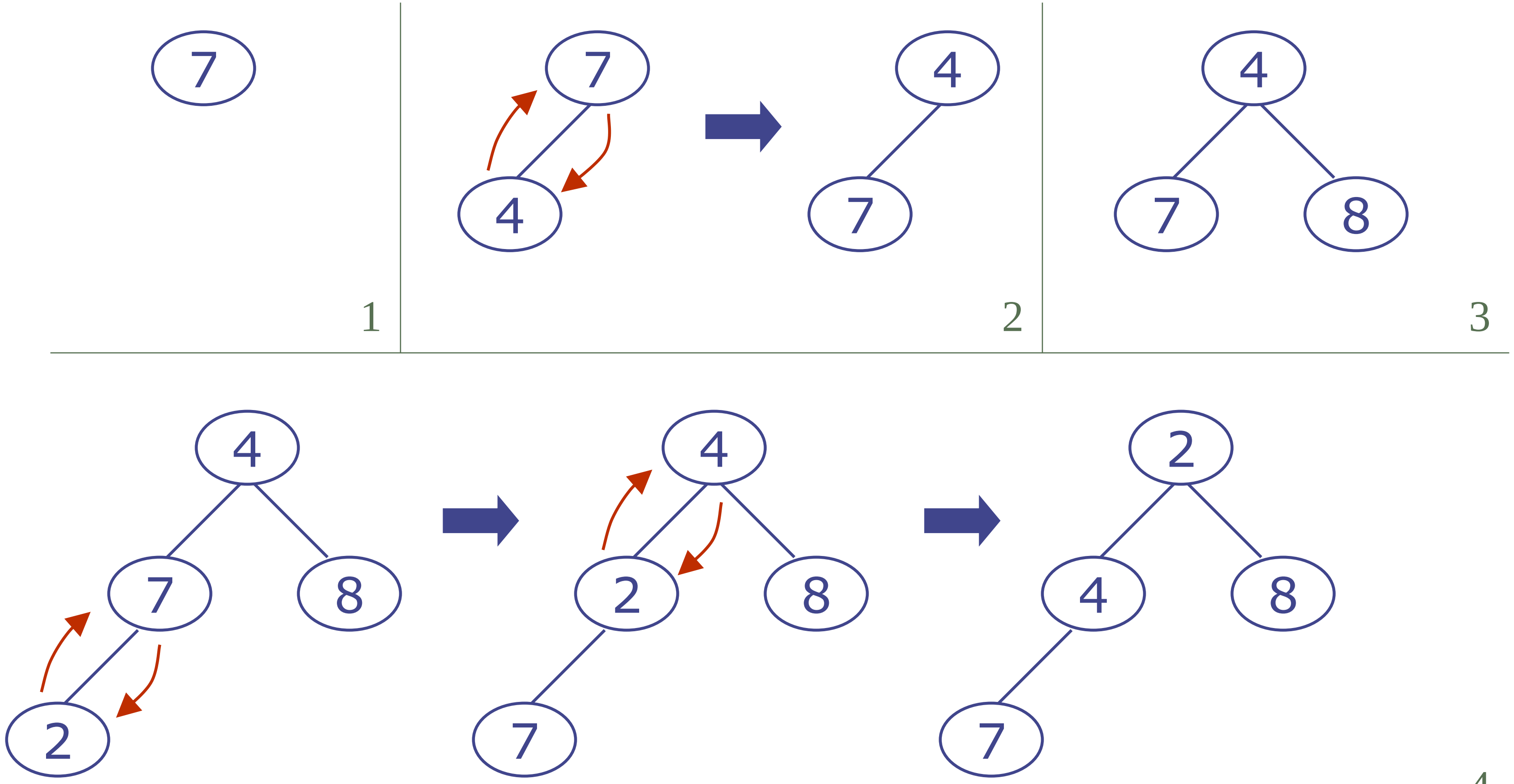
Heap-Sort

IUT-ECE

	Sequence/List S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	
(b)	(8,2,5,3,9)	
(c)	(2,5,3,9)	
(d)	(5,3,9)	
(e)	(3,9)	
(f)	(9)	
(g)	()	
Phase 2		
(a)	(2)	
(b)	(2,3)	
..	..	
(g)	(2,3,4,5,7,8,9)	

Heap-Sort

(7,4,8,2,5,3,9)



Heap-Sort

IUT-ECE

	Sequence/List S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	
(b)	(8,2,5,3,9)	
(c)	(2,5,3,9)	
(d)	(5,3,9)	
(e)	(3,9)	
(f)	(9)	
(g)	()	
Phase 2		
(a)	(2)	
(b)	(2,3)	
..	..	
(g)	(2,3,4,5,7,8,9)	

i-th insert operation $(1 \leq i \leq n)$ takes $O(1+\log i)$

n elements insertion: O(n logn)

j-th removeMin operation $(1 \leq j \leq n)$ runs in time $O(1+\log(n-j+1))$

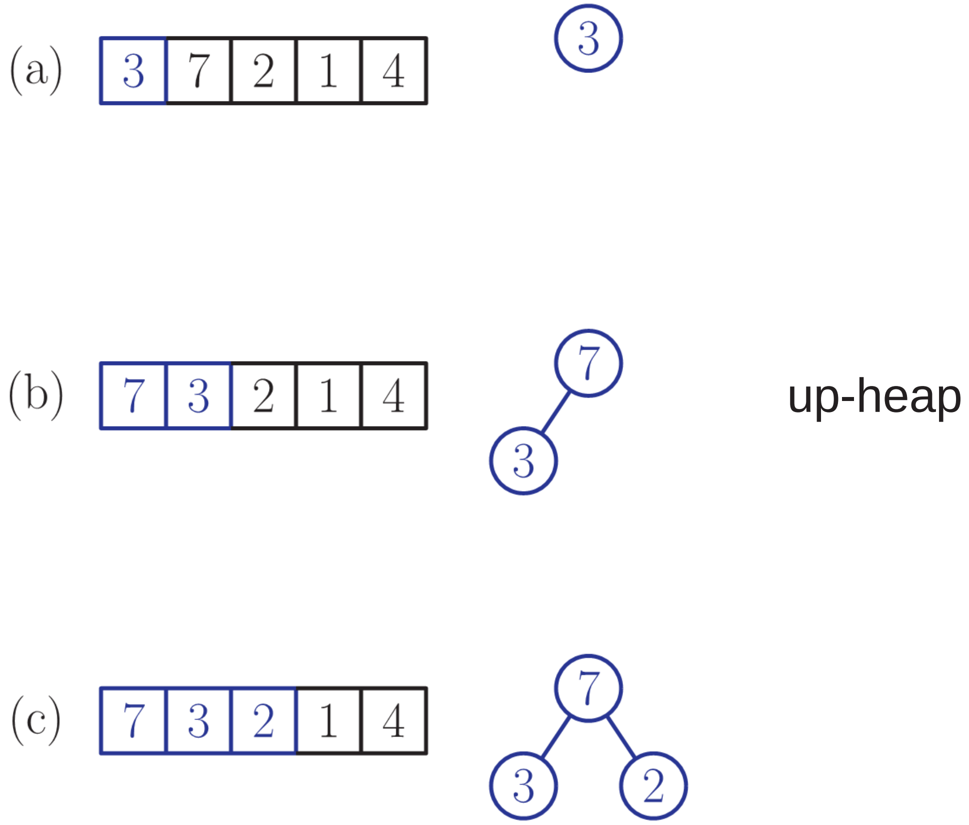
Heap-Sort In Place

IUT-ECE

- By max-heap (array implement)

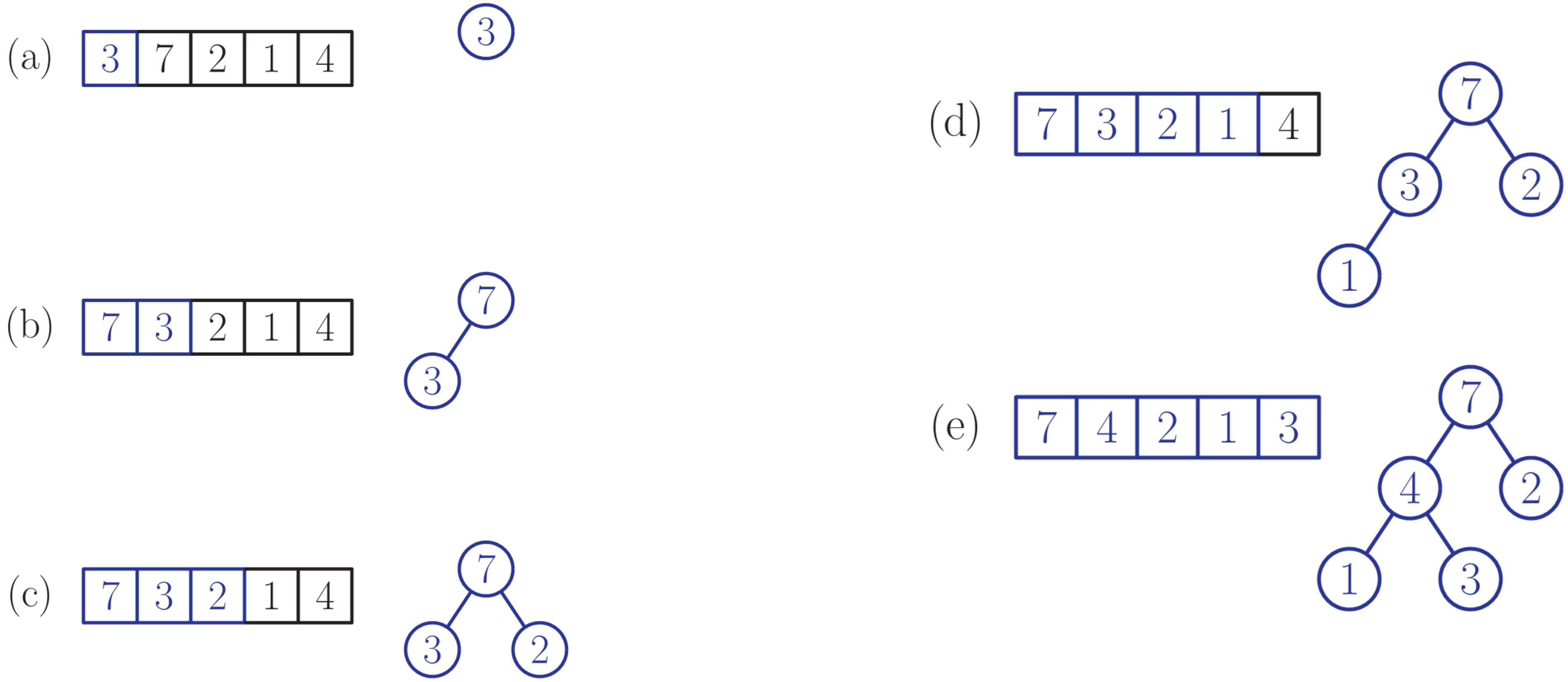
Heap-Sort In Place

- By max-heap (array implement)



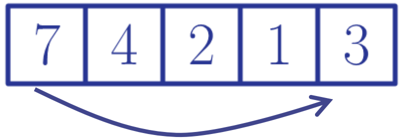
Heap-Sort In Place

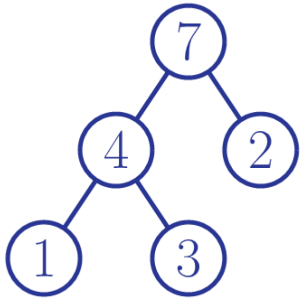
- By max-heap (array implement)



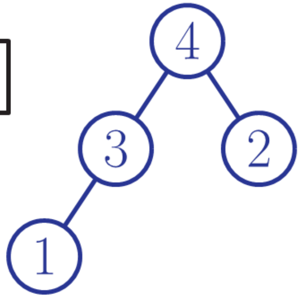
Heap-Sort In Place

- By max-heap (array implement)





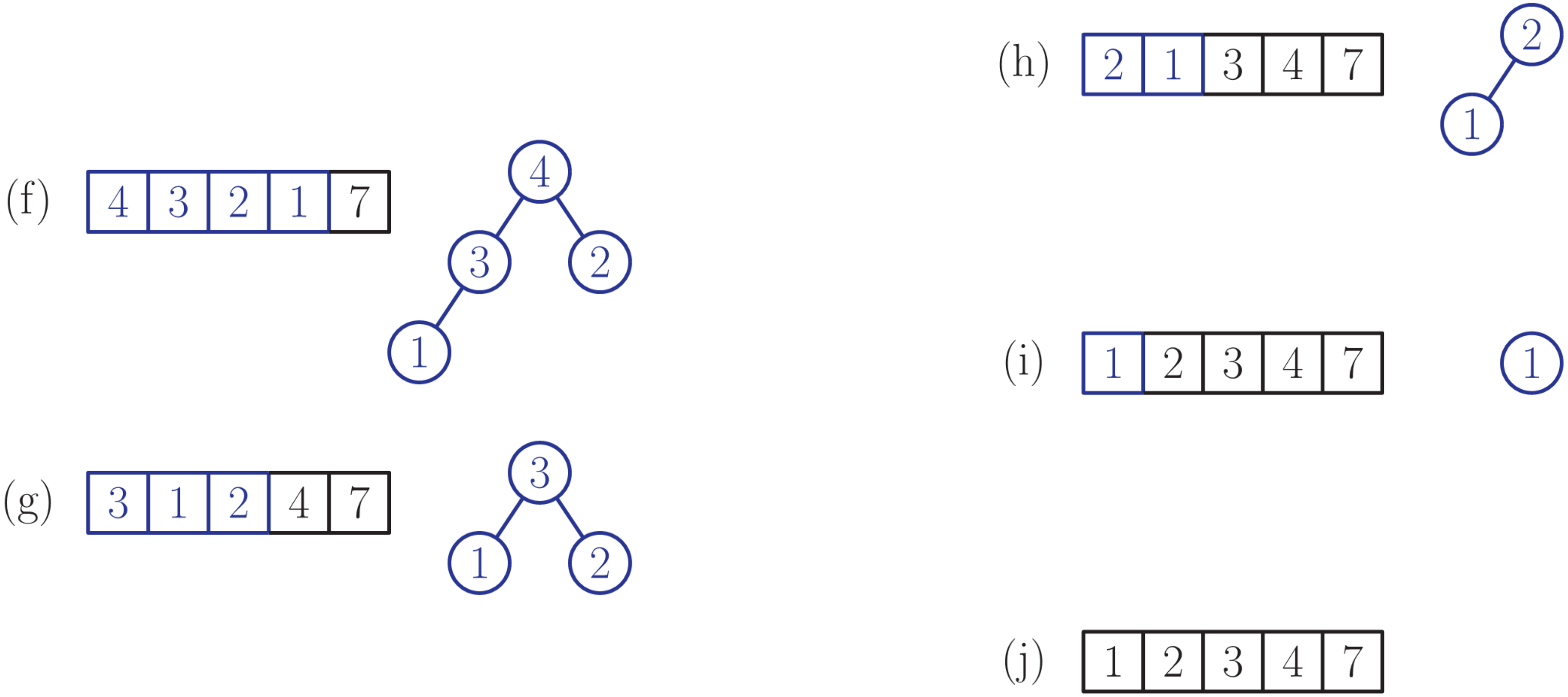
(f)



Heap down

Heap-Sort In Place

- By max-heap (array implement)

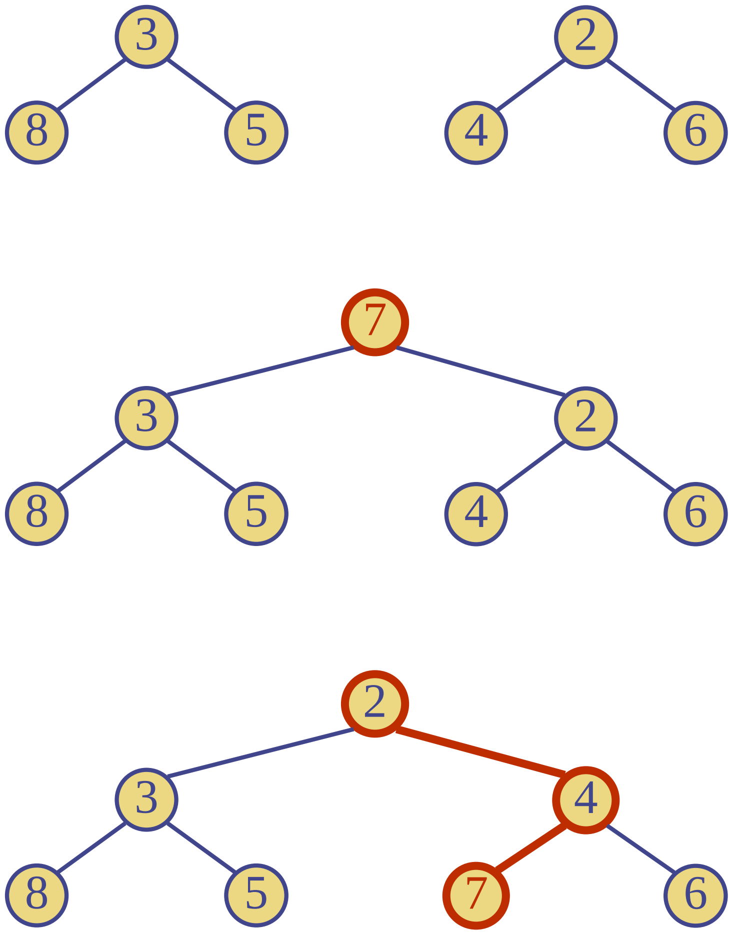


Sorting Comparison

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

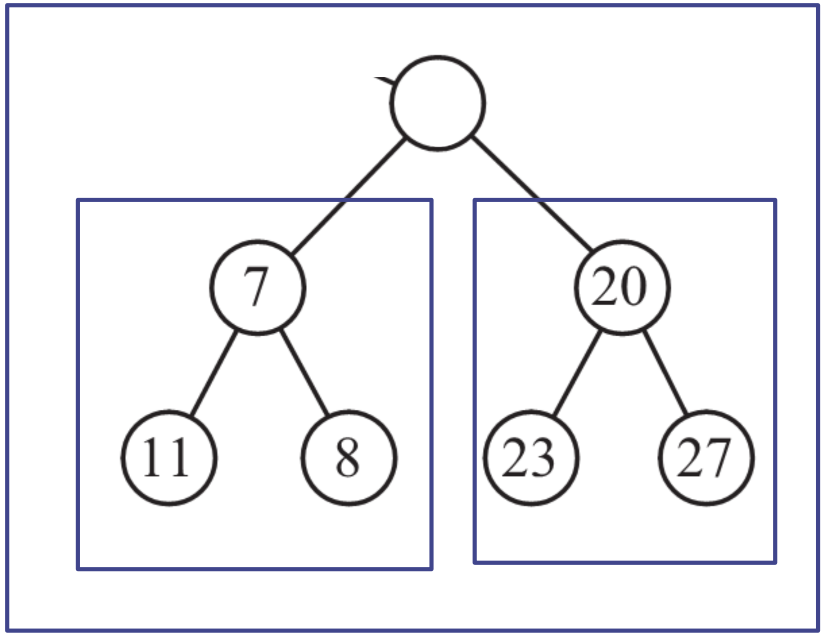
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Bottom-Up Heap Construction

- if all the elements to be stored in the heap are given in advance?



Bottom-Up Heap Construction

- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}

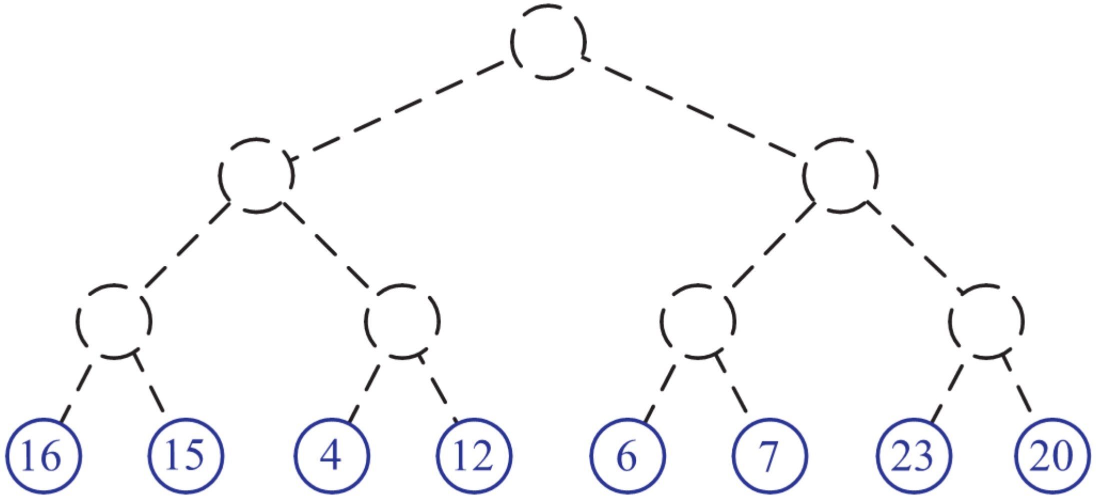
For simplicity, we describe this bottom-up heap construction assuming the number n of keys is an integer of the type $n = 2^h - 1$. That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1)$.

Bottom-Up Heap Construction

IUT-ECE

- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}

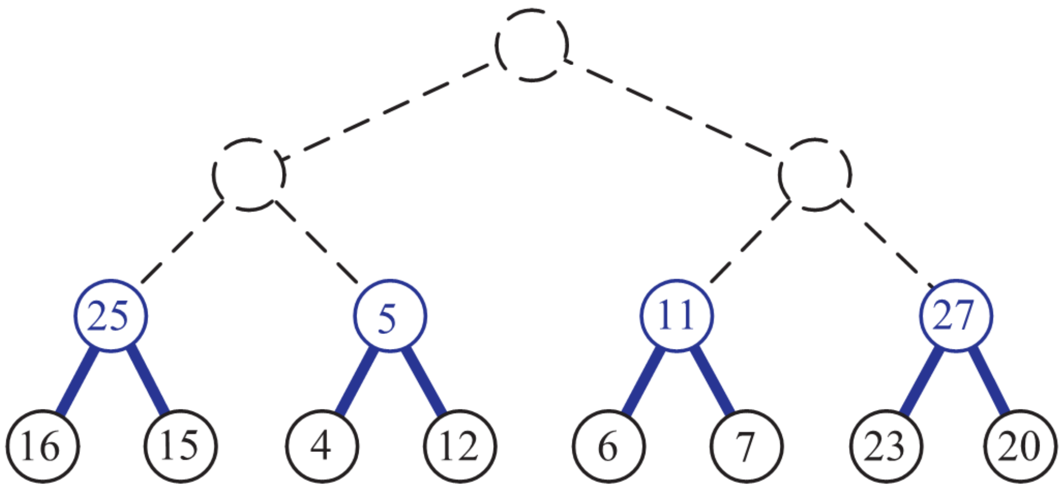
$(n+1)/2$ elementary heaps



Bottom-Up Heap Construction

- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}

$(n+1)/4$ heaps

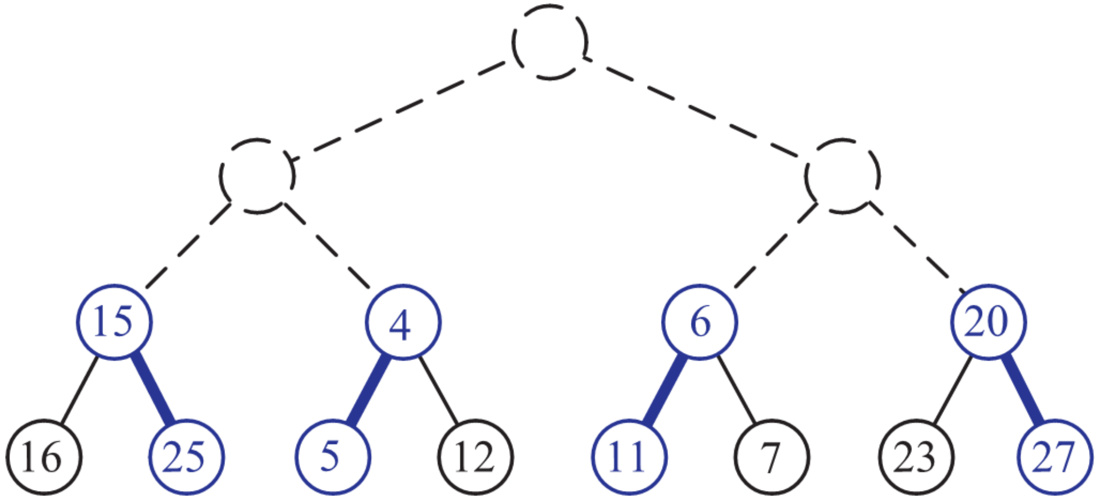


Bottom-Up Heap Construction

IUT-ECE

- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}

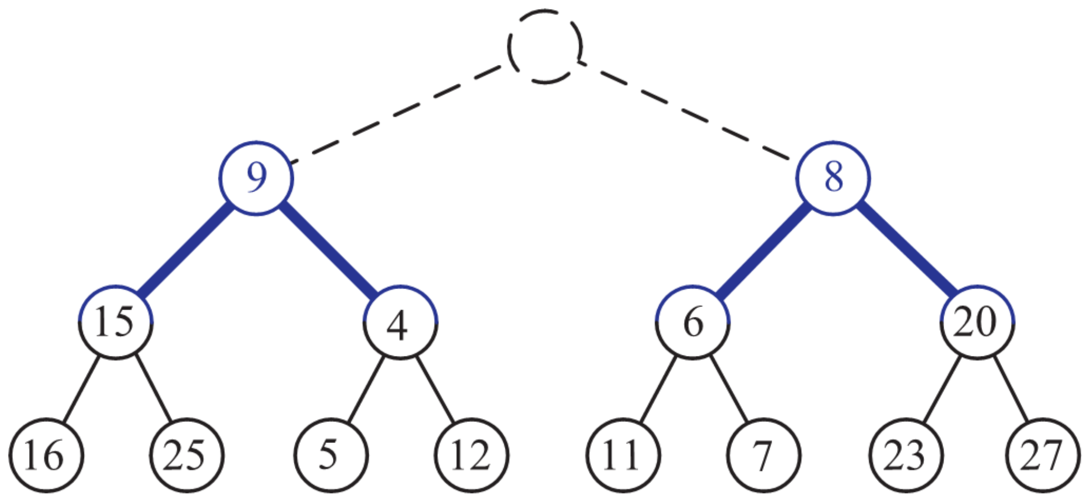
$(n+1)/4$ heaps



Bottom-Up Heap Construction

- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}

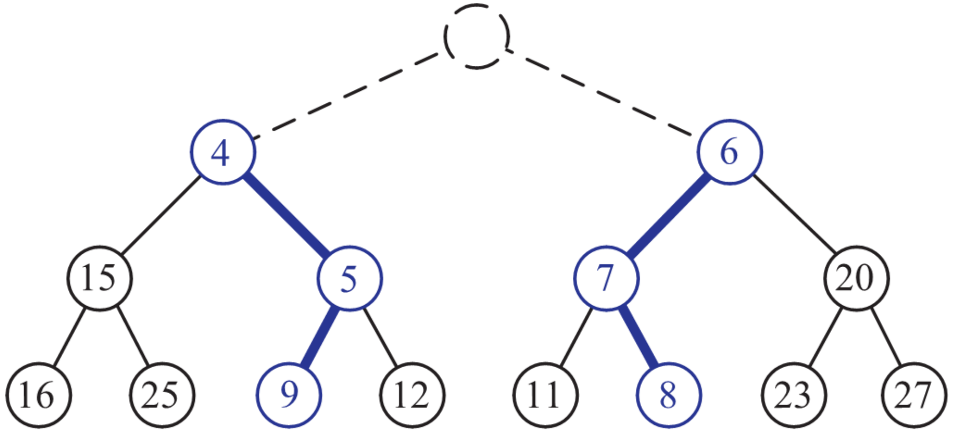
$2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing $2^i - 1$ entries,



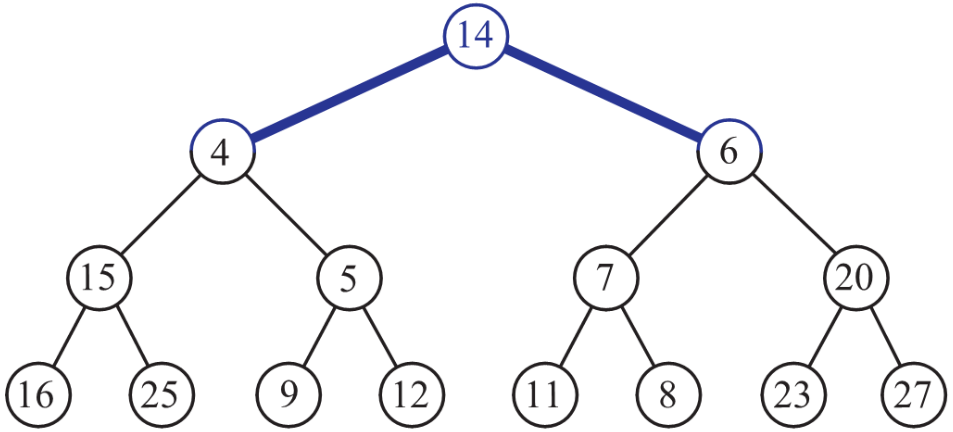
Bottom-Up Heap Construction

IUT-ECE

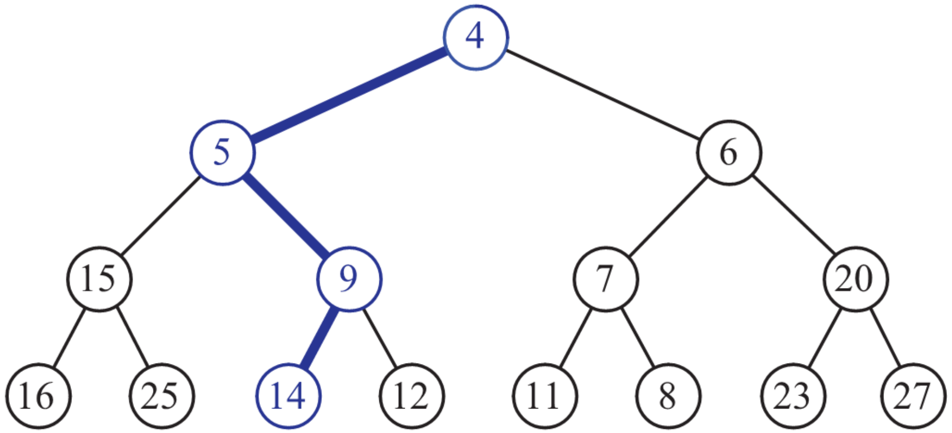
- Example : S={16,15,4,12,6,7,23,20,25,5,11,27,9,8,14}



(e)

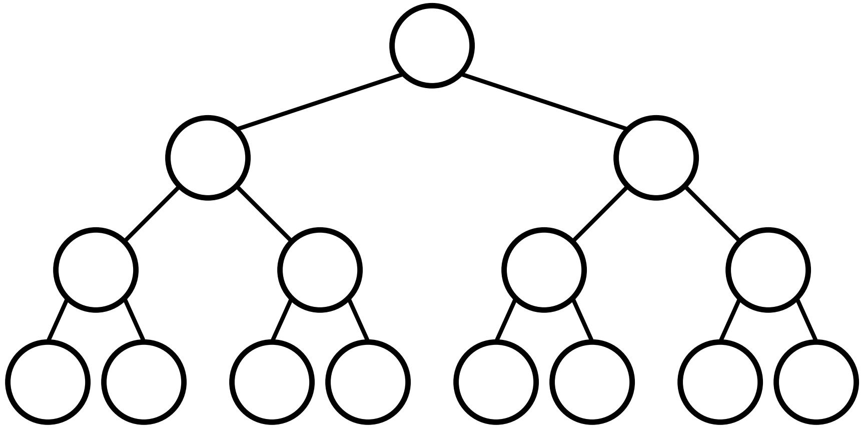


(f)



Bottom-Up Heap Construction

- Analysis?



Bottom-Up Heap Construction

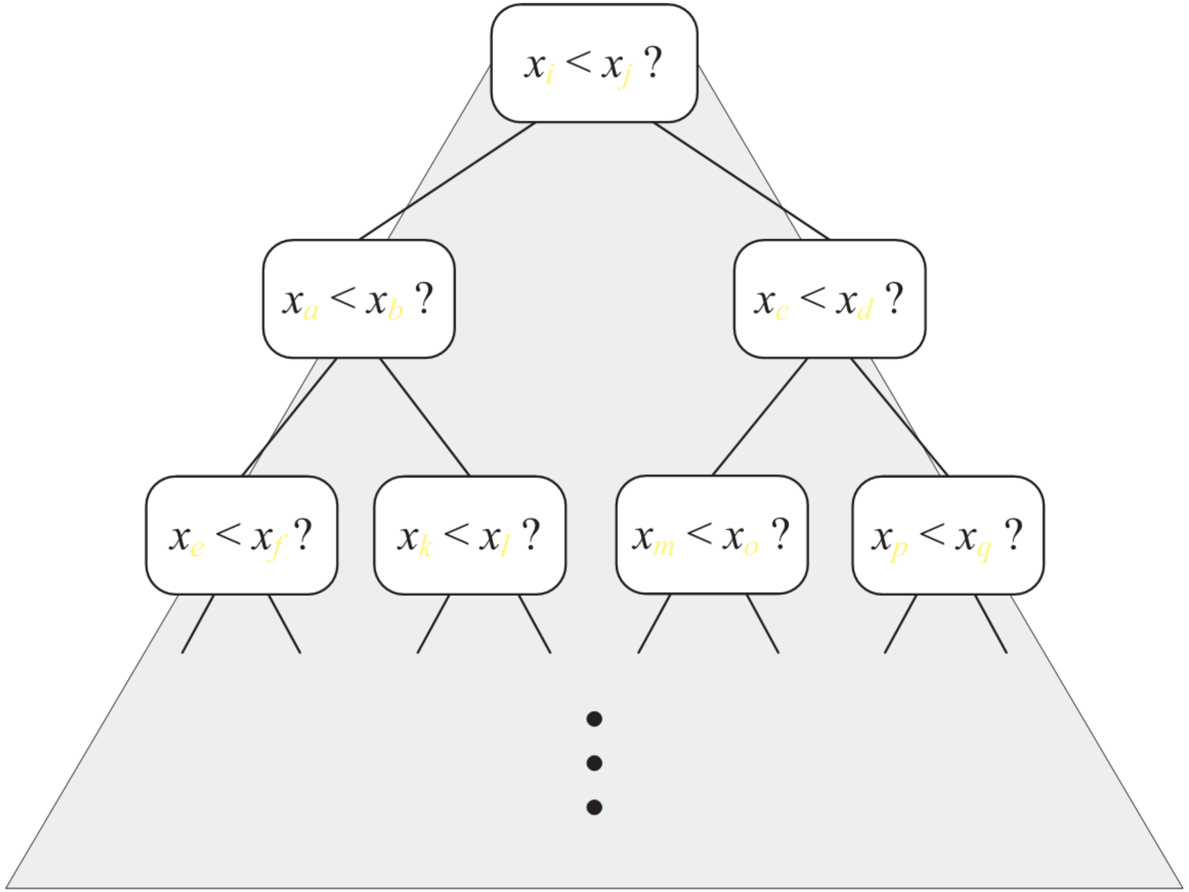
- if all the elements to be stored in the heap are given in advance, there is an alternative ***bottom-up*** **construction function that runs in** $O(n)$ **time.**
- **Compare?**
- **Impact on heap sort**
- **Can we improve second phase too?**

A Lower Bound for Sorting (comparison-based)

- Suppose we are given a sequence $S=(x0,x1,\ldots,xn-1)$ that we wish to sort, and assume that all the elements of *S* are distinct.
- we can represent a comparison-based sorting algorithm with a decision tree *T*.

A Lower Bound for Sorting (comparison-based)

- Worst-case #comparison?
- height

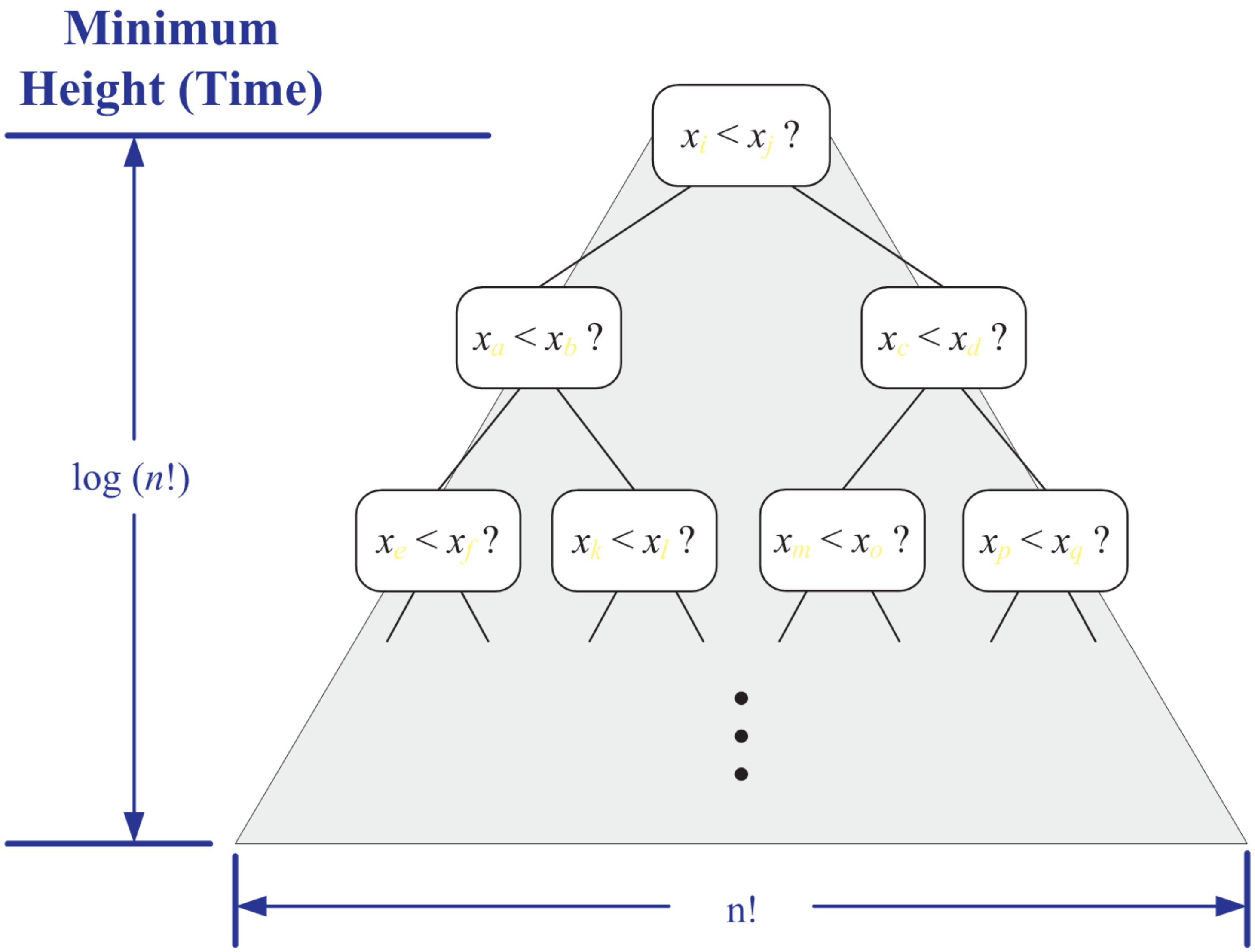


A Lower Bound for Sorting (comparison-based)

- Worst-case #comparison?
- Height
- Let us associate with each external node v in T , then, the set of permutations of S that cause our sorting algorithm to end up in v.
- The number of permutations of n objects is $n! = n(n-1)(n-2)\cdots 2\cdot 1$.

A Lower Bound for Sorting (comparison-based)

IUT-ECE



A Lower Bound for Sorting (comparison-based)

IUT-ECE

- Worst-case #comparison?
- Height
- Let us associate with each external node *v* in *T* , then, the set of permutations of *S* that cause our sorting algorithm to end up in *v*.
- The number of permutations of *n* objects is $n! = n(n-1)(n-2)\cdots 2\cdot 1$.

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2},$$

which is $\Omega(n \log n)$.

C++ STL

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

Binary heap

سوال

- بردار نامرتب A شامل n عنصر را در نظر بگیرید. kامین عنصر بزرگ این لیست را بیابید.