CS 521 Lecture III

DREXEL UNIVERSITY
DEPT. OF COMPUTER SCIENCE

FALL 2011

Heaps, Priority Queues and Heap Sort

Priority Queue

- Handles a collection of items, called keys.
- There exists a way to compare keys to each other. This is called an order relation.
- The result of these comparisons determines the priority of the keys.
- Operations supported:
  - *insert a* key
  - *Remove* the largest key

Applications

- Scheduling
- Operating systems
- Keeping track of largest n elements in a sequence
- *Sorting*

Methods of a Priority Queue

- *Initialize*: initialize the structure
- *Insert (key)*: insert a new key
- *Remove Max*: return and remove largest key

PQ-Sort in procedural pseudocode

- (sorting an array with using a priority queue)
  - *Initialize*
  - *for i = 1 to n*

    Insert (a[i])
  - *for i := n downto 1*

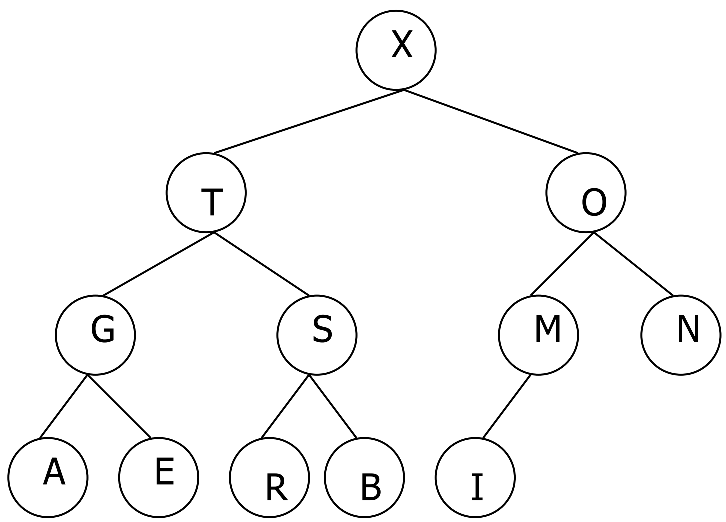
    a[i] :=RemoveMax

How to Implement a Priority Queue

Implementation	Insert	Remove Max	Delete	Average
Unsorted Array or Linked List	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Sorted Array or Linked List	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Heap

- A *heap* is a *binary tree* storing keys, with the following properties:
  - partial order:
    - **key (child) < key(parent)**
  - left-filled levels:
    - the last level is left-filled
    - the other levels are full



Logarithmic Height

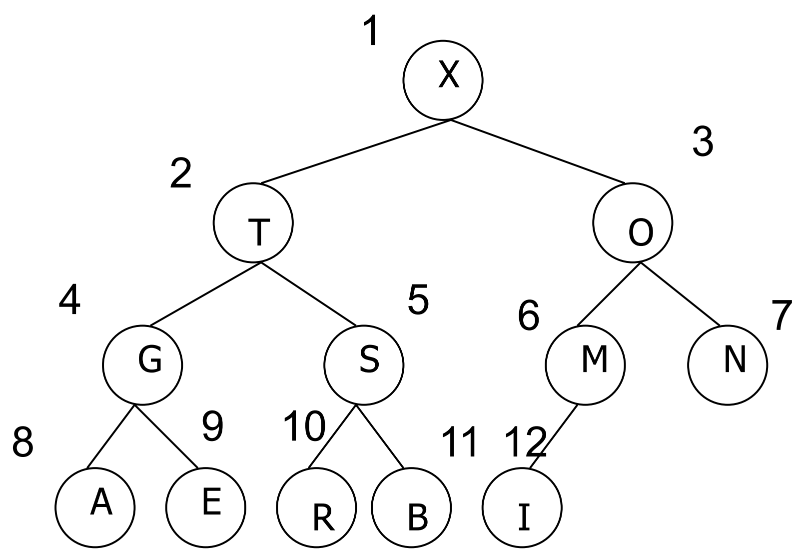
- A heap with n keys has height: $H(n) = \log_2 n$
- *Proof:*

Let n be the number of keys, and $H(n)$ be the height.
We have:

$$2^{H(n)-1} \le n \le 2^{H(n)}$$

Taking logarithm of both sides; the result will follow.

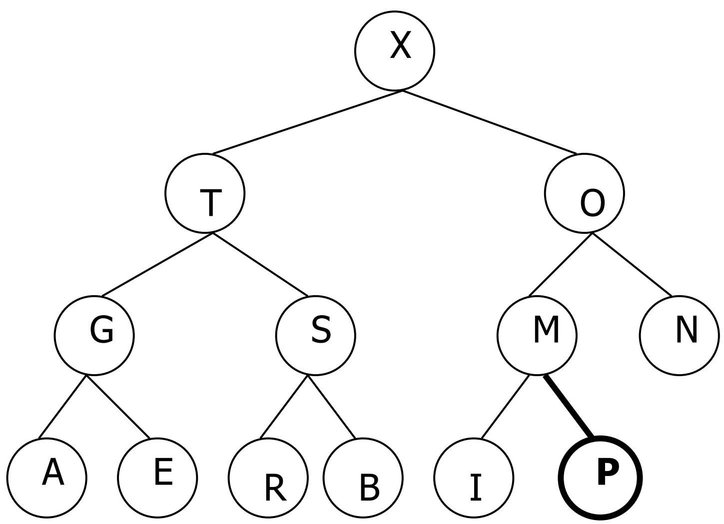
Heap Representations

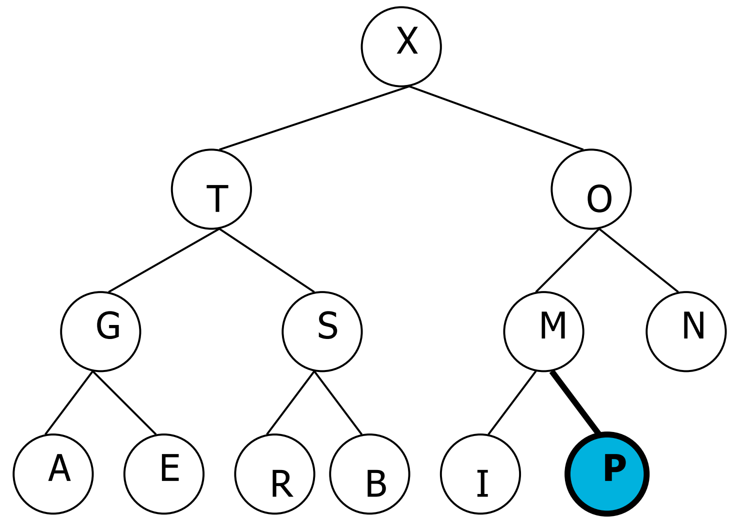


X	T	O	G	S	M	N	A	E	R	B	I
1	2	3	4	5	6	7	8	9	10	11	12

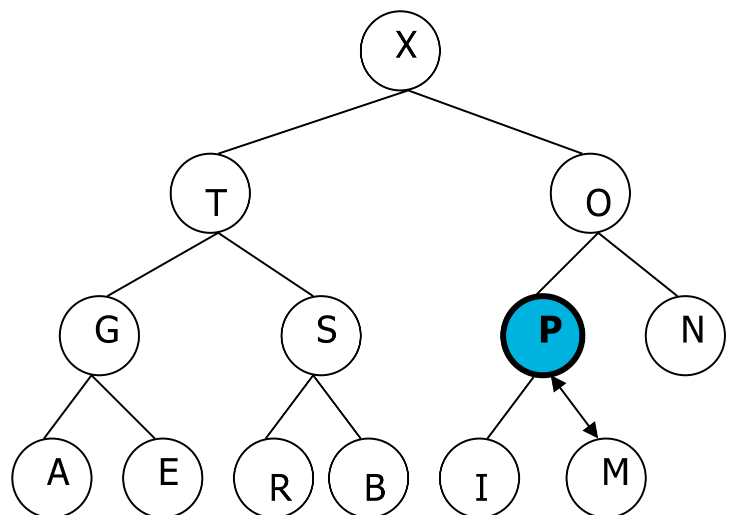
- left_child(i)= 2i
- right_child(i) = 2i+1
- parent(j) = j div 2

Heap Insertion





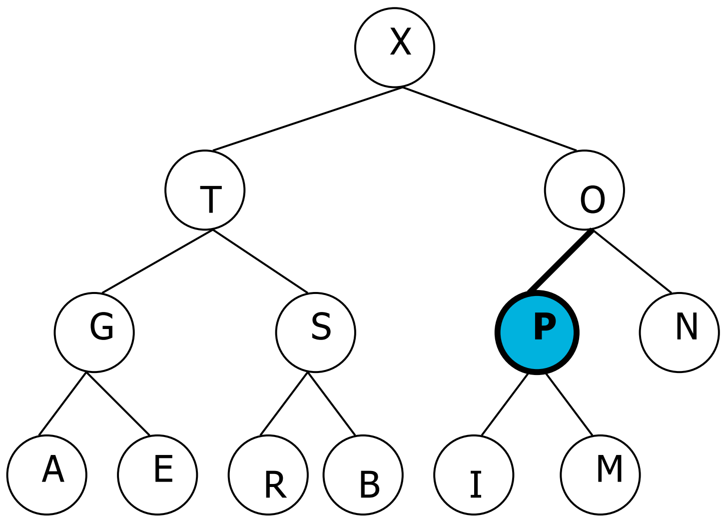
•Add the key in the next available spot in the heap.

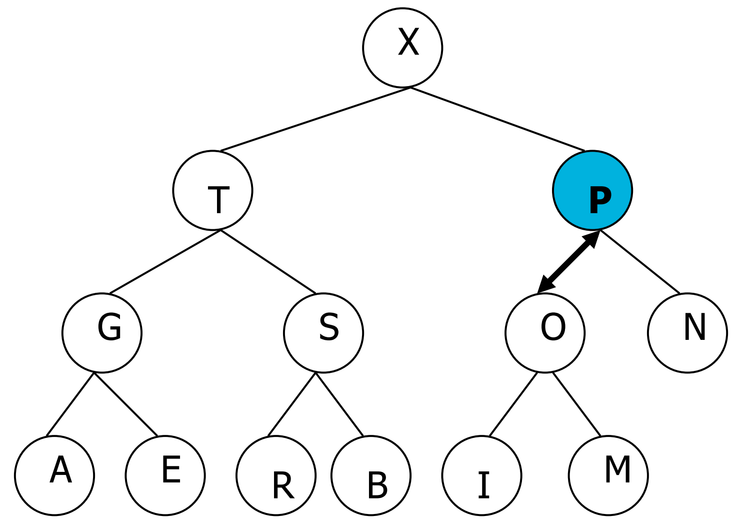


•***Upheap*** checks if the new node is greater than its parent. If so, it switches the two.

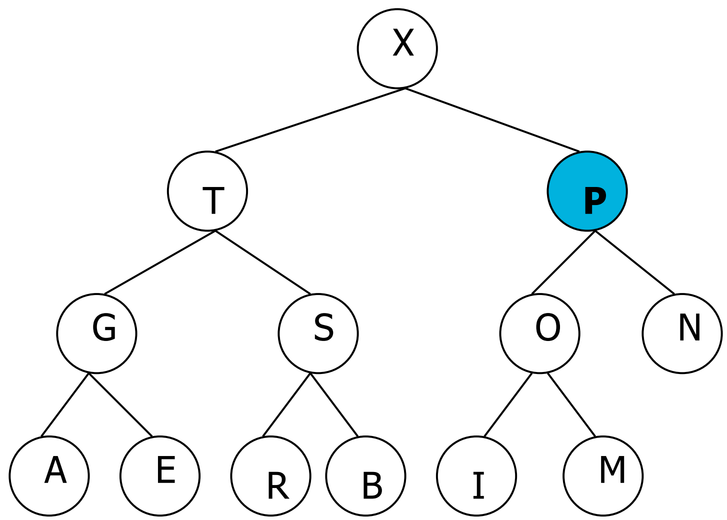
•***Upheap*** continues up the tree

Heap Insertion





Heap Insertion



- ***Upheap*** terminates when new key is less than the key of its ***parent*** or the ***top of the heap*** is reached.
- (total #switches) <= (height of tree- 1) =log n

Heapify Algorithm

- Assumes L and R sub-trees of *i* are already Heaps and makes tree rooted at *i* a Heap:

```
Heapify(A,i,n)
    If (2i<=n) & (A[2i]>A[i]) Then
        largest=2i
    Else largest=i
    If (2i+1<=n) & (A[2i+1]>A[largest]) Then
        largest=2i+1
    If (largest != i) Then
        Exchange (A[i],A[largest])
        Heapify(A,largest,n)
    Endif
End Heapify
```

Extracting the Maximum from a Heap:

- Here is the algorithm:

Heap-Extract-Max(A)

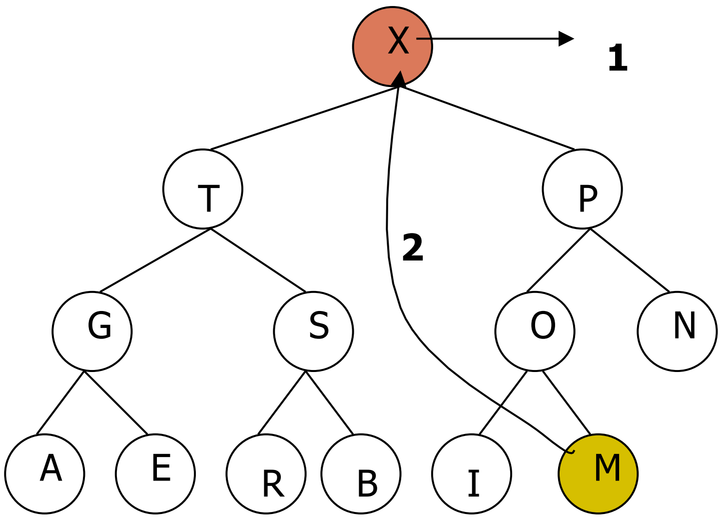
Remove A[1]

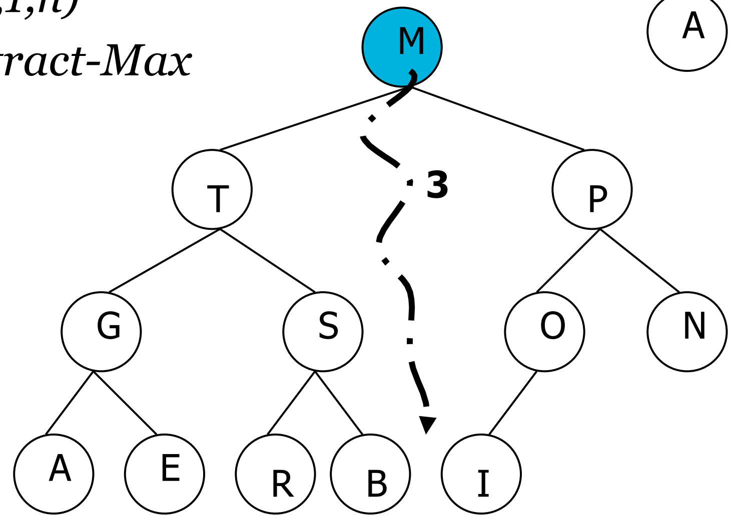
A[1]=A[n]

n=n-1

Heapify(a,1,n)

End Heap-Extract-Max





Building a Heap

- **Builds a heap from an unsorted array:**

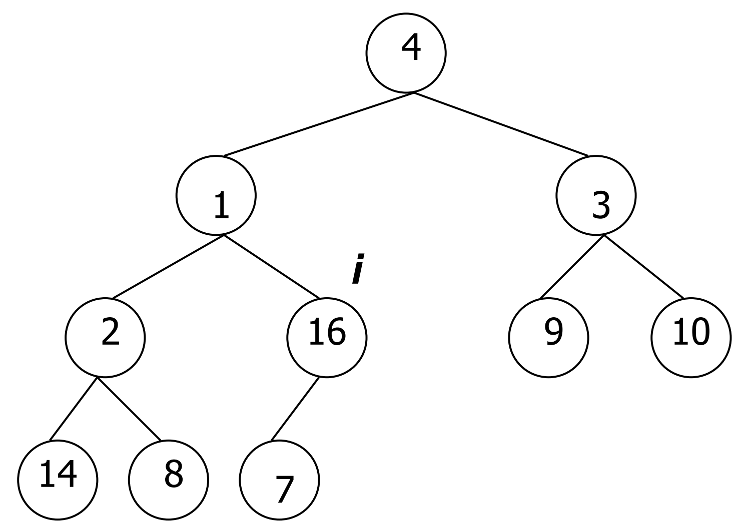
Build_Heap(A,n)

For i=floor(n/2) down to 1 do

Heapify(A,i,n)

End Build_Heap

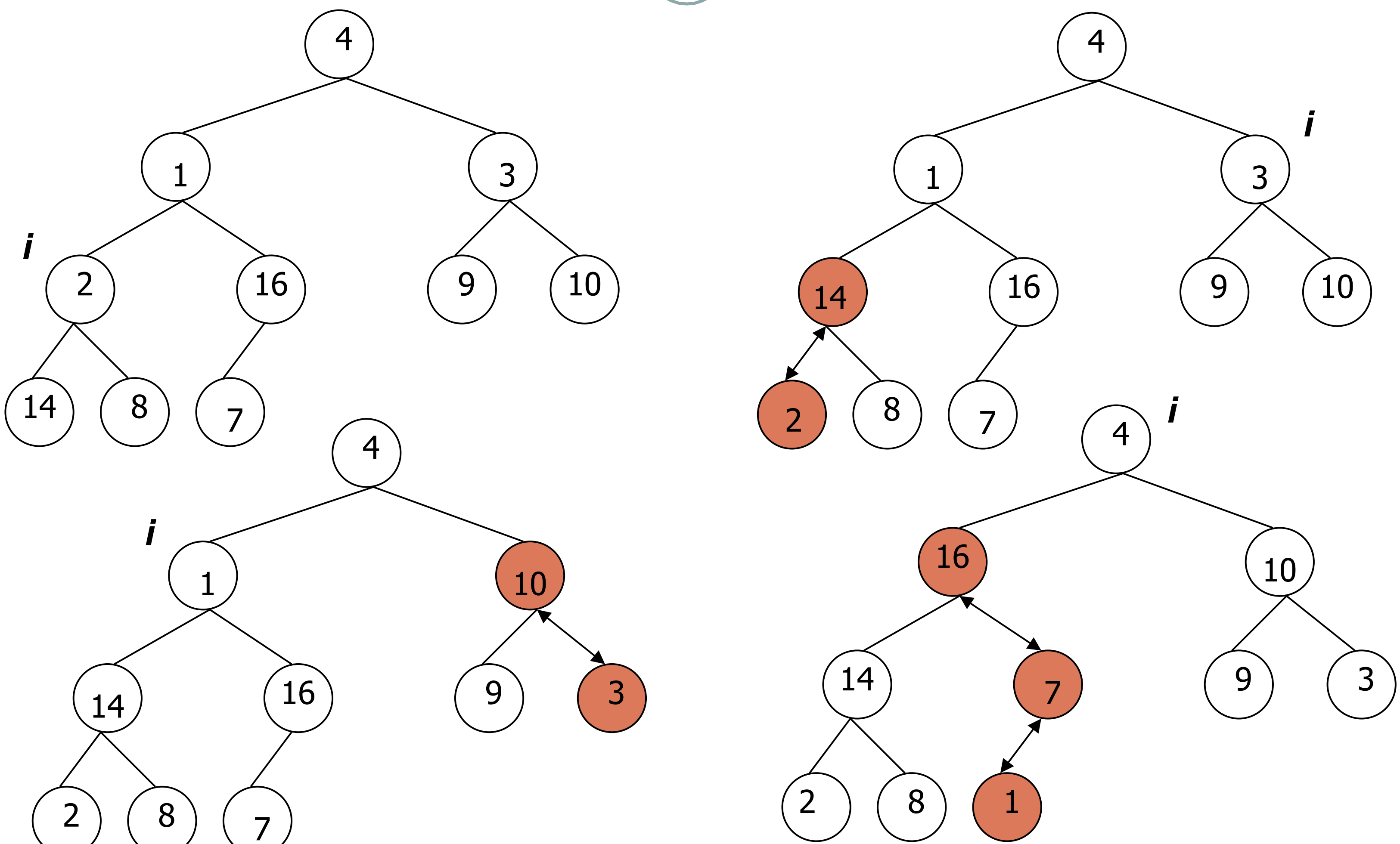
- **Example:**



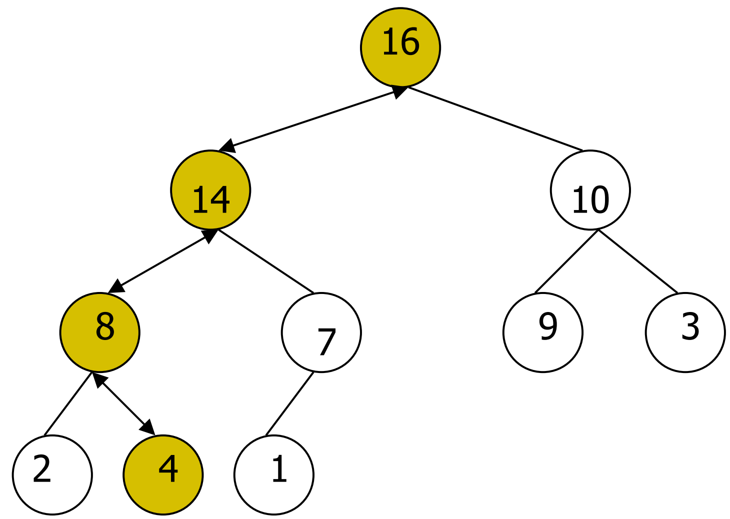
A

4	1	3	2	16	9	10	14	8	7

Building a Heap (cont' d.)



Building a Heap (cont' d.)



Running time of Building a Heap

- ***O(n* log *n)* is trivial*: n* calls of Heapify, each of cost *O(*log n*)***
- **Tighter Bound: *O(n)***
  - **The cost of "Heapify" is proportional to the number of levels visited (height of node *i)***
  - **Assume *n=2^k-1* (complete binary tree):**
    - **For each leaf node, the number of levels visited is 1,**
    - **For each node at next level is 2,**
    - **3 for next level, etc.**

$$\text{Total \# of levels visited} = \frac{n+1}{2}\times 1 + \frac{n+1}{4}\times 2 + \frac{n+1}{8}\times 3 + \cdots + \frac{n+1}{2^{\log(n+1)}}\times \log(n+1)$$

$$= (n+1)\sum_{i=0}^{\log(n+1)} \frac{i}{2^i}$$

Running time of Building a Heap (cont' d.)

- Using Induction, it is easy to see that

$$\sum_{i=0}^{\log(n+1)} \frac{i}{2^i} = O(1)$$

Implying:

$$T(n) = \text{Total \# of levels visited} = O(n)$$

Heapsort

Heapsort(A,n)

Build-Heap(A,n)

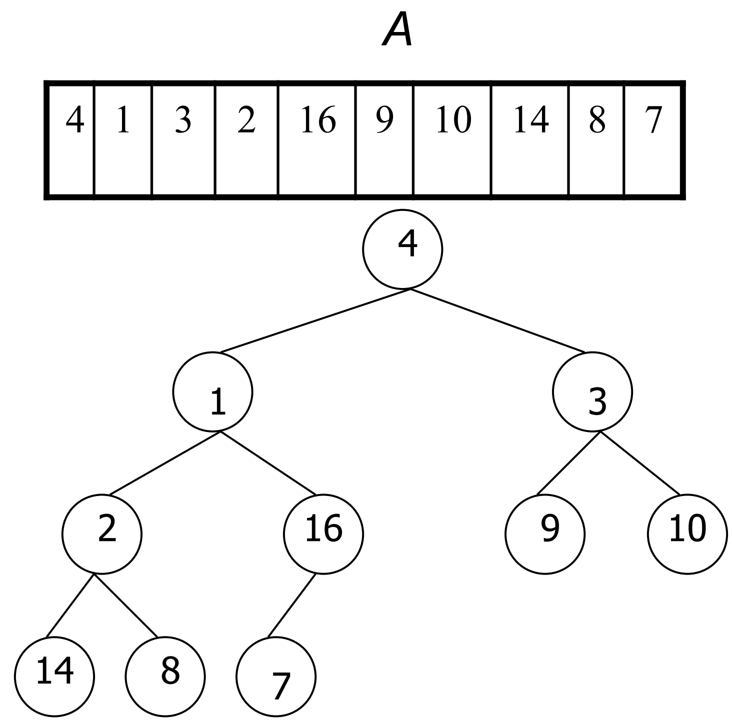
For i=n downto 2 do

Exchange A[1] & A[i]

Heapify(A,1,i)

End For

End Heapsort

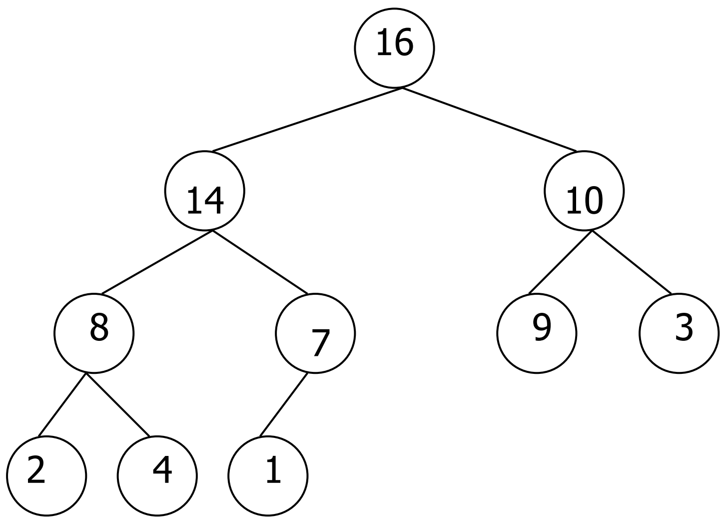


Heapsort (cont' d.)

- First build the corresponding Heap:

A

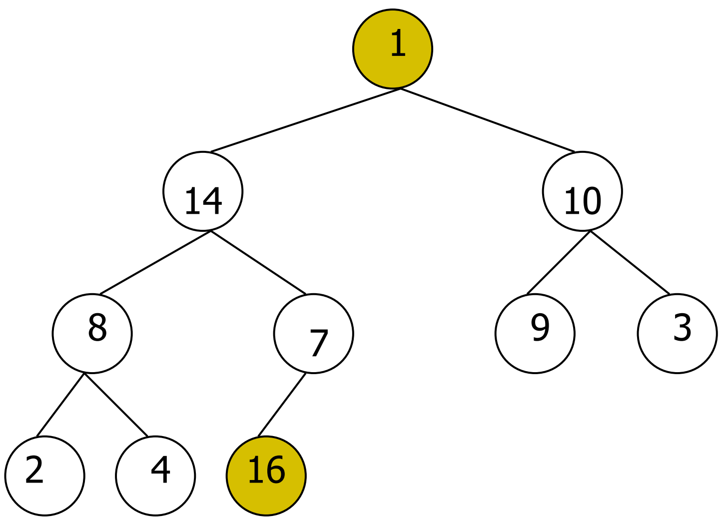
16	14	10	8	7	9	3	2	4	1



Heapsort (cont' d.)

A

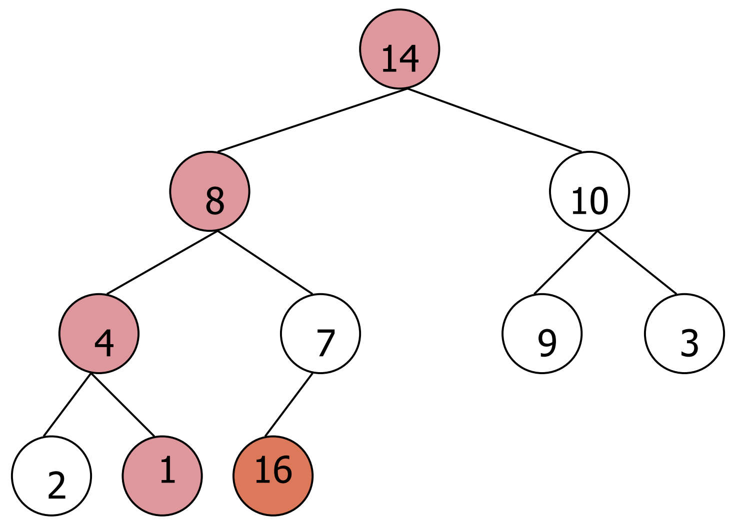
1	14	10	8	7	9	3	2	4	16



Exchange

A

14	8	10	4	7	9	3	2	1	16

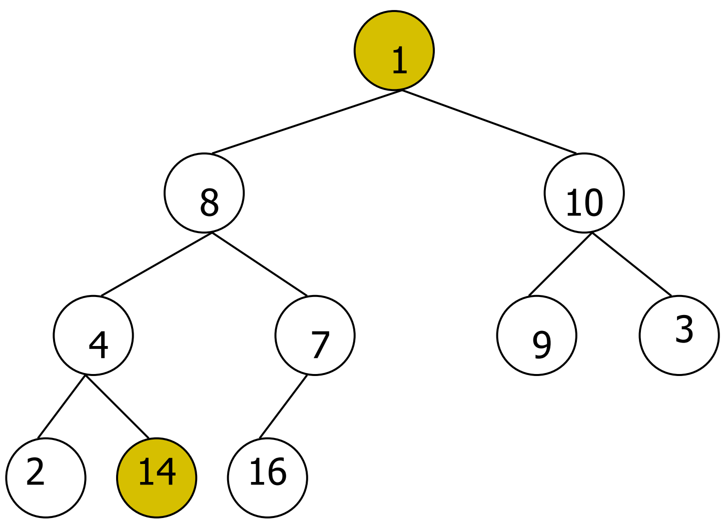


Heapify(A,1,n-1)

Heapsort (cont’d.)

A

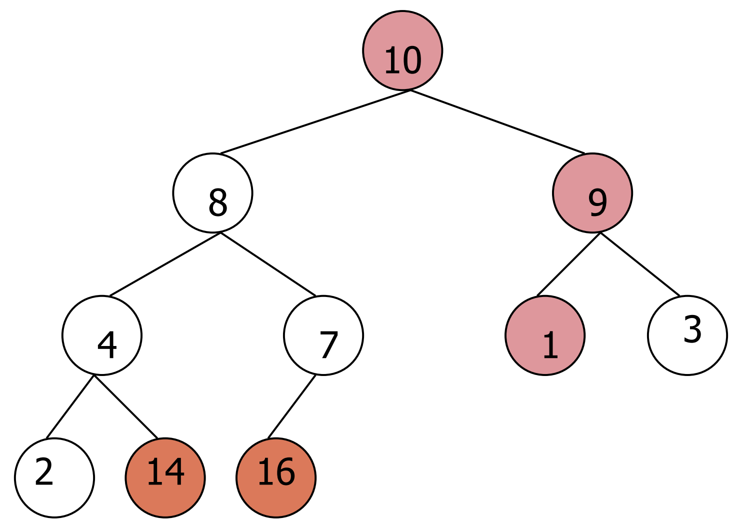
1	8	10	4	7	9	3	2	14	16



Exchange

A

10	8	9	4	7	1	3	2	14	16

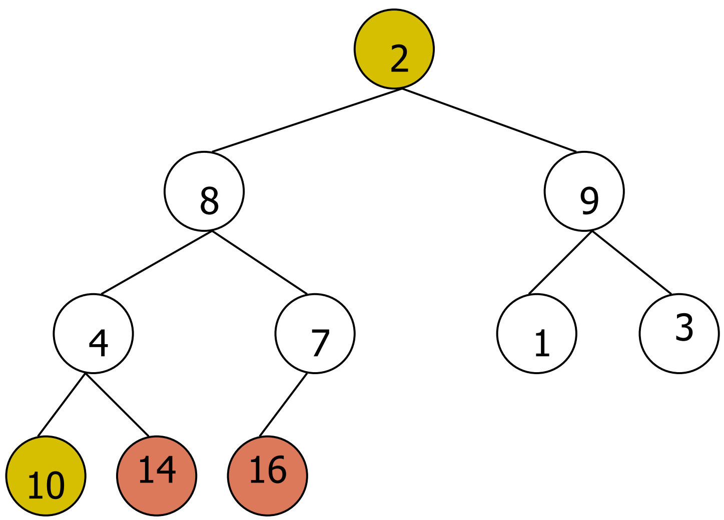


Heapify(A,1,n-2)

Heapsort (cont’d.)

A

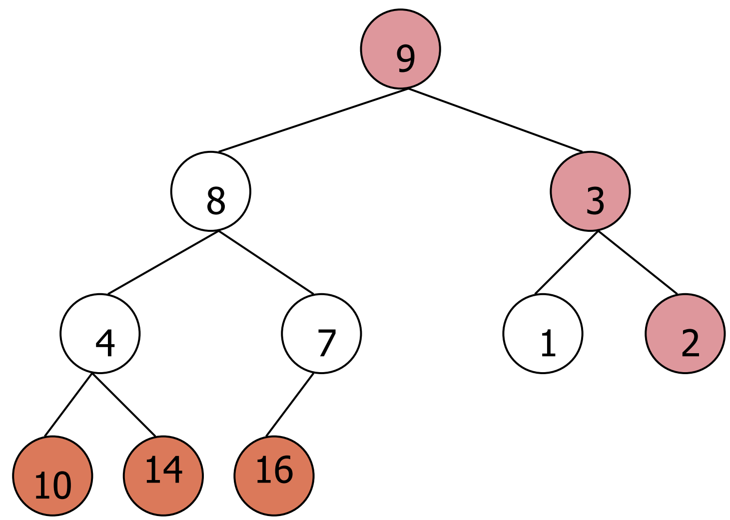
2	8	9	4	7	1	3	10	14	16



Exchange

A

9	8	3	4	7	1	2	10	14	16

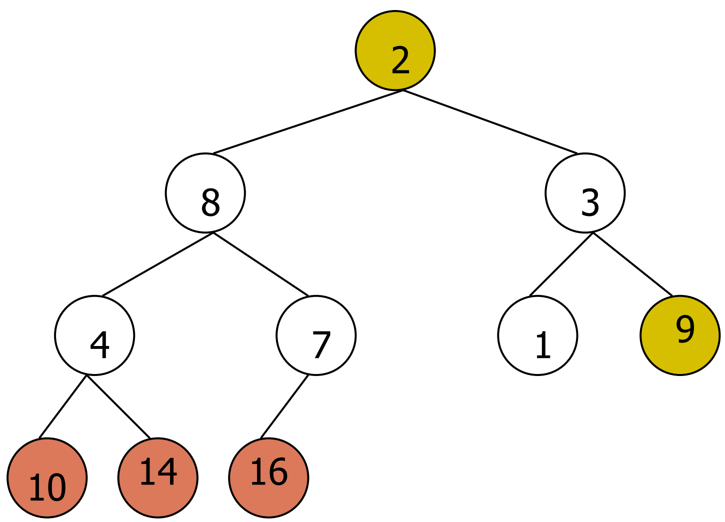


Heapify(A,1,n-3)

Heapsort (cont' d.)

A

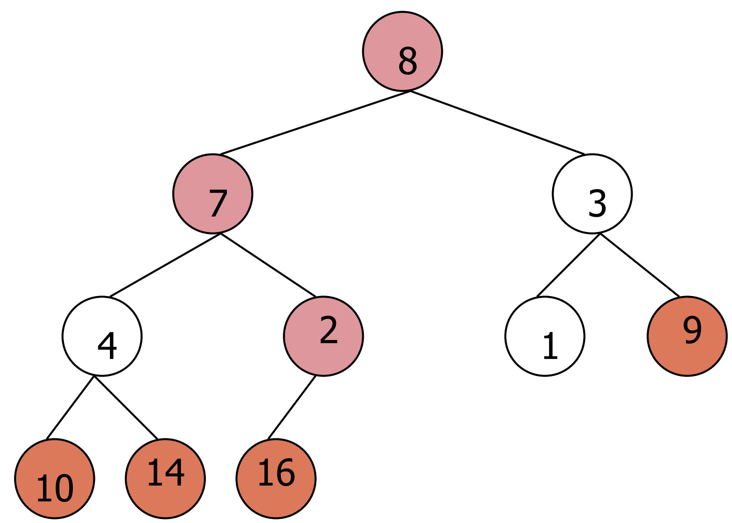
2	8	3	4	7	1	9	10	14	16



Exchange

A

8	7	3	4	2	1	9	10	14	16

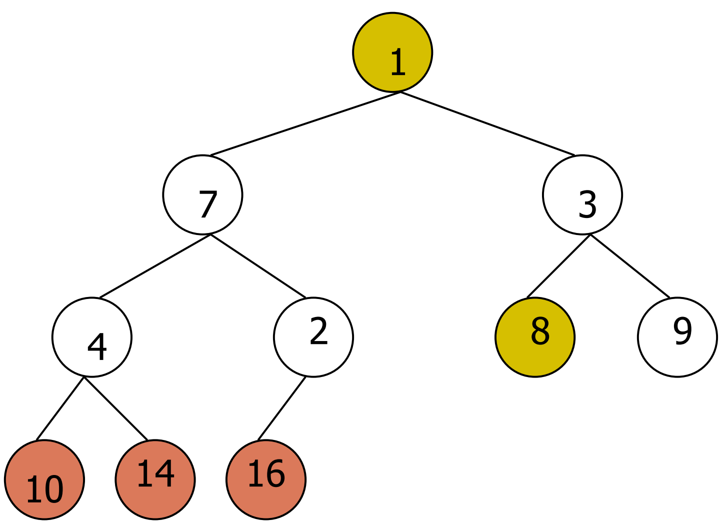


Heapify(A,1,n-4)

Heapsort (cont' d.)

A

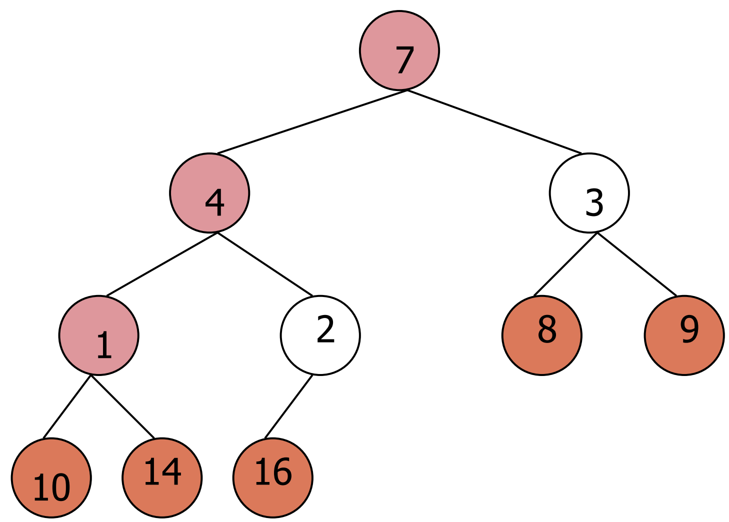
1	7	3	4	2	8	9	10	14	16



Exchange

A

7	4	3	1	2	8	9	10	14	16



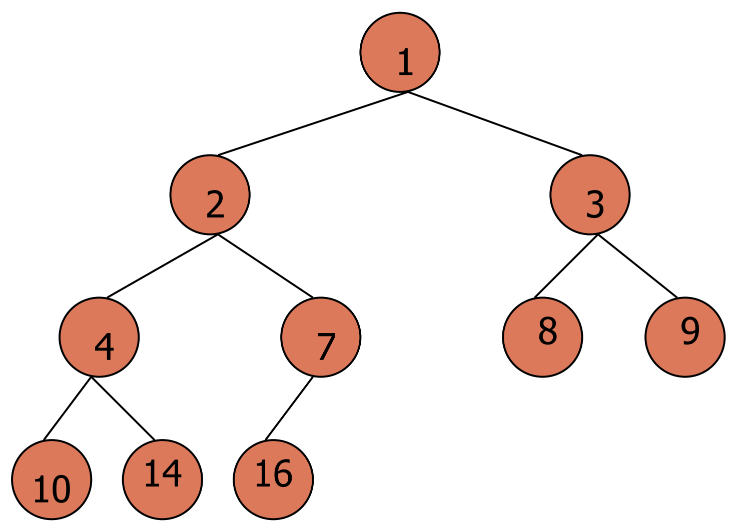
Heapify(A,1,n-5)

Heapsort (cont'd.)

Finally:

A

1	2	3	4	7	8	9	10	14	16



Running Time

Heapsort(A,n)

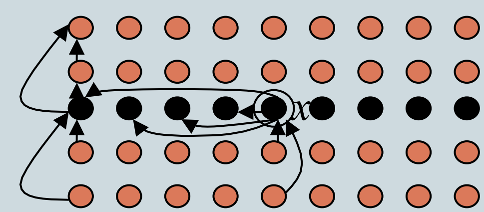
Build-Heap(A,n)	$O(n)$
For $i=n$ downto 2 do	$n-1$ Times
Exchange $A[1]$ & $A[i]$	$O(1)$
Heapify$(A,1,i)$	$O(\log n)$

End Heapsort

- Total Running time: $O(n \log n)$

Selection Problems

MEDIANS AND ORDER STATISTICS



Order Statistics

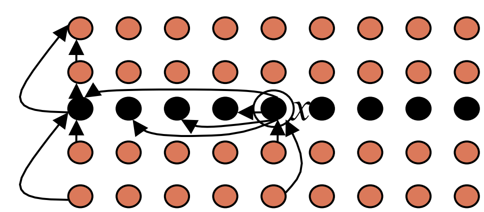
- The i^{th} order statistic of a set of n numbers is the i^{th} smallest element in sorted sequence:

A

4	1	3	2	16	9	10	14	8	7

- ***Minimum*** or first order statistic: 1
- ***Maximum*** or n^{th} order statistic: 16
- ***Median*** or $(n/2)^{th}$ order statistic: 7 or 8

  (both are medians, happens when n is even!)

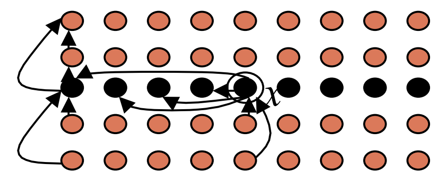


The Selection problem:

- **Input:** An array ***A*** of distinct numbers of size *n*, and a number ***i***.
- **Output:** The element ***x*** in ***A*** that is larger than exactly ***i-1*** other elements in ***A***.

- Finding ***maximum*** and ***minimum*** can be easily solved in linear time ($O(n)$).

  (it's actually $\Theta(n)$).



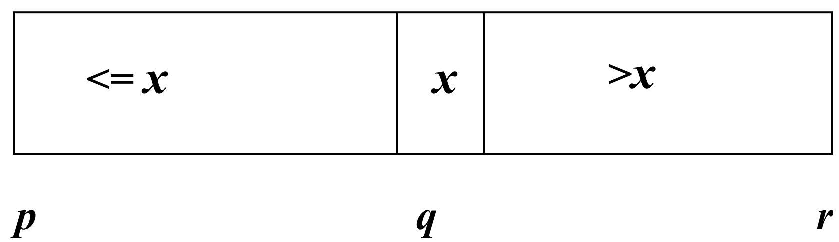
Trivial Solution:

- Sort the array ***A***, and return the entry in $\boldsymbol{i}^{\text{th}}$ position:
  - Sorting ***A*** takes ***O*(*n*** log ***n*).**
  - The $\boldsymbol{i}^{th}$ entry can be returned in constant time.
- Worst case running time: ***O*(*n*** log ***n*)**
- Can we do better?

  Comparing to ***maximum*** and ***minimum***, the general ***i*** is taking a long time.

A Randomized Selection Algorithm (idea):

- Think about the properties of **Partition( )** algorithm:

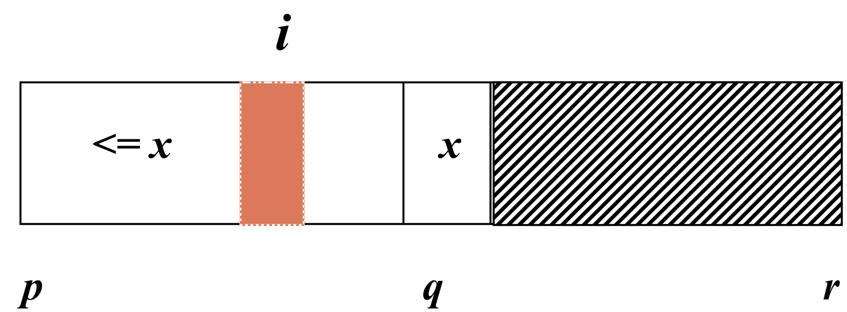


- If $\boldsymbol{i=q}$, then we have $\boldsymbol{x}$ as the $\boldsymbol{i}^{th}$ order statistic.

  (what if this is not the case?)

A Randomized Selection Algorithm (idea):

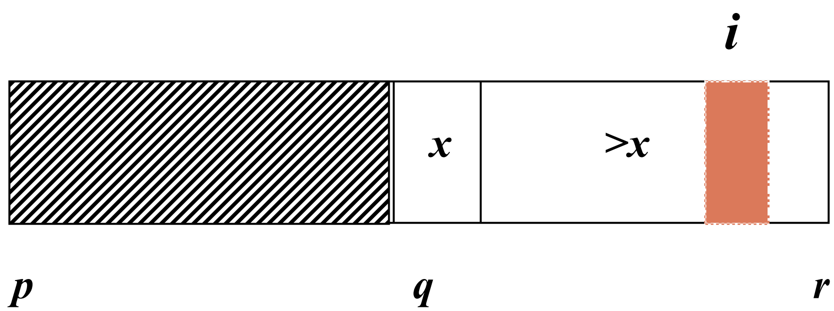
- If ***i*< *q*,** then we have to look for the ***i***th order statistic among first ***p-q+1*** elements:



- We can call **Partition( )**, with parameters ***(A,p,q)***

A Randomized Selection Algorithm (idea):

- If ***i*> *q*,** then we have look to for ***i***th order statistic among elements between ***q*** and ***r***:



- We can call **Partition( )**, with parameters ***(A,q,r)***

The Algorithm:

Randomized-Select(A,p,r,i)

  if $p=r$ then

    Return $A[p]$

  q=Randomized-Partition(A,p,r)

  $k=q-p+1$

  if $i<= k$ then

    Randomized-Select(A,p,q,i)

  else

    Randomized-Select($A,q,r,i-k$)

Running time:

- The recurrence:
  - Lucky: $T(n) = T(9n/10) + \Theta(n) = \Theta(n)$

    Using master theorem:

    $$n^{\log_{\frac{10}{9}} 1} = n^0 = 1$$

  - Unlucky: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

    Worst than sorting!

Average Case:

- Assume **Partition( )** Algorithm breaks ***A*** to two pieces with sizes ***k*** and ***n-k-1***,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

- Assume ***T(n)<= cn*** for some ***c***.

Average Case (cont' d.)

$$T(n) = \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2} k\right) + \Theta(n)$$

$$= \frac{2c}{n}\left(\frac{n}{2}(n-1) - \frac{1}{2}\frac{n}{2}\left(\frac{n}{2}-1\right)\right) + \Theta(n)$$

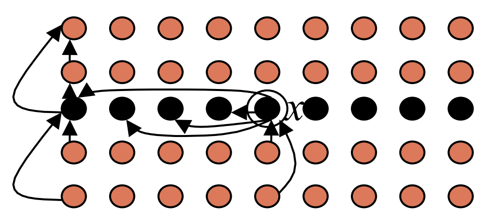
$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$

$$= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right)$$

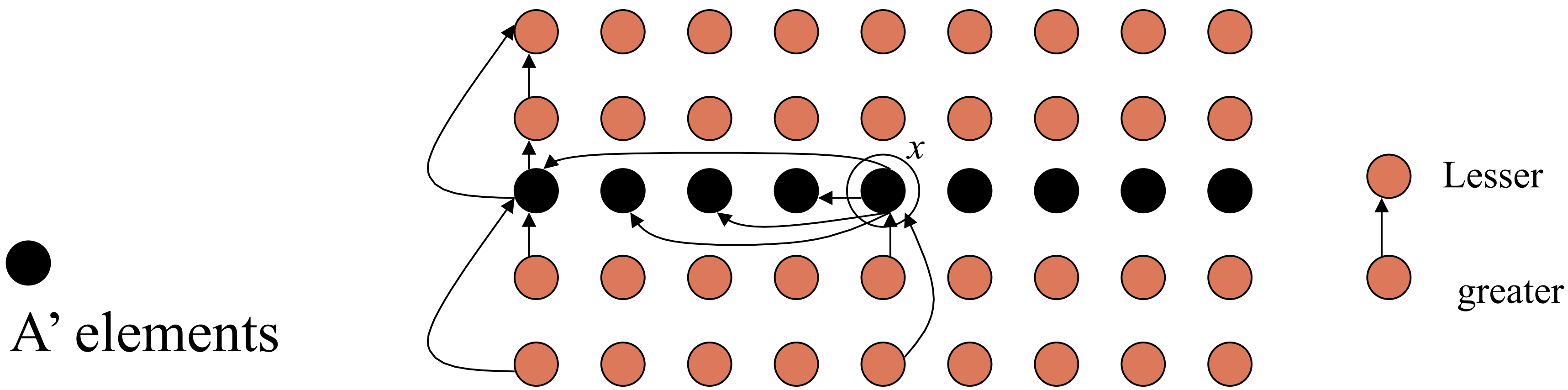
$$\leq cn$$

Worst-case Linear-Time O.S.

- Select(A,p,q,i) Algorithm:
  - Divide A to $n/5$ groups of size 5.
  - Find the median of each group of 5 by brute force, and store them in a set A' of size $n/5$.
  - Use **Select**($A',1,n/5,n/10$) to find the median x of $n/5$ medians.
  - Partition the n elements around x. Let $k = q-p+1$ (rank of x).
  - if $i=k$ then return x
  - if $i< k$ then **Select**(A,p,k-1,i)
  - else **Select**($A,k,q,i-k$)



Analysis



- At least half of ***A'*** is less than ***x***, which is at least ***n/10*** elements of ***A'***.
- Thus ***3n/10*** elements are smaller than ***x***.
- If ***n>=50*** then ***3n/10>=n/4***, so ***n/4*** elements are smaller than ***x***, and we know where they are!

- The components of recurrence for ***T(n):***
  - ***T(n/5)*:** to find median of n/5 medians,
  - ***T(3n/4)*:** the complexity of step ***5***.
  - $\Theta(n)$ **:** The time for **Partition( ).**
  - ***T(n)=T(n/5)+T(3n/4)+*** $\Theta(n)$

Analysis (cont' d.)

- **Claim: *T(n)=cn*.**

$$
\begin{aligned}
T(n) &= cn/5 + 3cn/4 + \Theta(n) \\
&\le 19cn/20 + O(n) \\
&= cn - (cn/20 - O(n)) \\
&\le cn, \text{ for large enough } c.
\end{aligned}
$$

Simplified Master Theorem:

- Assume that $T(1) = d$, and for $n > 1$:

$$T(n) = aT(n/b) + cn.$$

- If $a < b$, Then $T(n) = O(n)$;
- If $a = b$, Then $T(n) = O(n \log n)$;
- If $a > b$, Then $T(n) = O(n^{\log_b a})$;

e.g. $T(n) = 4T(n/2) + cn$ gives $T(n) = O(n^{\log_b a}) = O(n^2)$