# CS 521 Lecture II

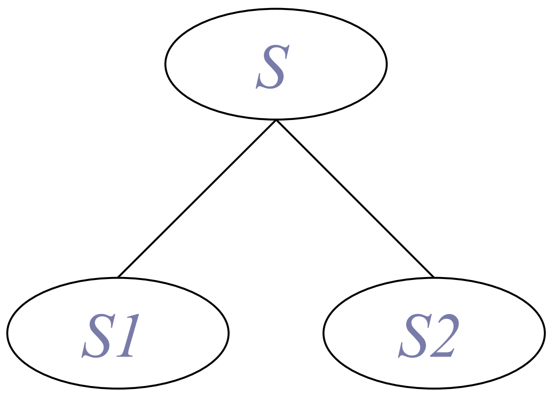
DREXEL UNIVERSITY
DEPT. OF COMPUTER SCIENCE

FALL 2011

# Today's Lecture

- Continue with Recurrences
  - Merge Sort
  - Master Theorem
  - Quicksort
- First Data-Structure:
  - Heap
  - Its Application in Sorting

# Merge-Sort

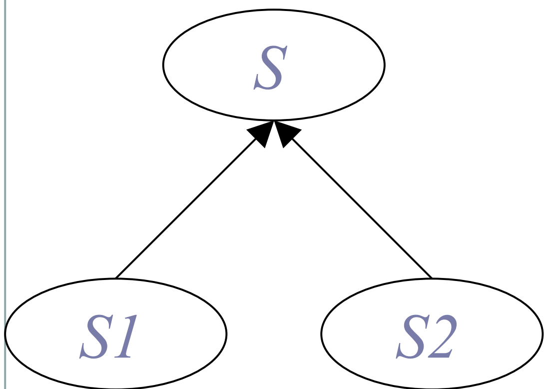



- Problem: Given a list *S* of *n* integers, create a sorted list of elements in *S*.
- Merge-sort Algorithm:

Divide: If ***S*** has at lest two elements (nothing needs to be done if ***S*** is empty or has only one element), remove all the elements from ***S*** and put them into two sequences, ***S1*** and ***S2***, each containing about half of the elements of ***S***.

  - **Recursion:** Sort sequences *S1* and *S2*.
  - **Conquer:** Put back the elements into *S* by merging the sorted sequences *S1* and *S2* into a unique sorted sequence.

# Merging Two Sorted Sequences




- Problem: Given two sequences $S_1$ and $S_2$ of sizes $n_1$ and $n_2$, create a (union) sorted list S (of size $n=n_1+n_2$).
- Algorithm Merge($S_1$,$S_2$,$S$):
  - ***top***$(S_i)$ = first element in $S_i$, for $i$ in **{1,2}.**
  - **While** $S_1$ is not empty and $S_2$ is not empty do

    **if** ***top***$(S_1)$ < ***top***$(S_2)$ **then**

    move ***top***$(S_1)$ at the end of $S$

    advance ***top***$(S_1)$

    **else**

    move ***top***$(S_2)$ at the end of $S$

    advance ***top***$(S_2)$

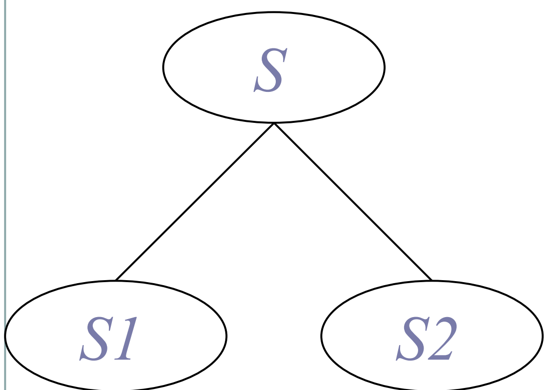
    **While** $S_1$ is not empty **do**

    move the remaining *of* $S_1$ *to* $S$

    **While** $S_2$ is not empty **do**

    move the remaining *of* $S_2$ *to* $S$

# Recurrence for Merge Sort:




- Recurrence Relation:

  $$T(n)=2T(n/2)+n$$
  $$T(1)=1$$

- Solution by unfolding:

  $$\begin{aligned} T(n) &= 2(2T(n/4)+(n/2))+n \\ &= 4T(n/2)+2n \\ &= 4(2T(n/8)+(n/4))+2n \\ &= 8T(n/8)+3n= \\ &\ldots \\ &= 2^iT(n/2^i)+i.n \\ &= \ldots \end{aligned}$$

The expansion stops for $i=\log n$

$$T(n) = 2^{\log n}+n \log n$$

Total Number of moves:

$$T(n)= n + n \log n = O(n \log n)$$

# Iterative recurrences

- Example:

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= n + 4(n/2 + 4T(n/4)) \\
&= n + 2n + 16T(n/4) \\
&= n + 2n + 16\left[n/4 + 4T(n/8)\right] \\
&= n + 2n + 4n + 4T(n/8) \\
&= n + 2n + 4n + \ldots \\
&= n \sum_{i=0}^{\log n - 1} 2^i + 4^{\log n} T(1) \\
&= \Theta(n^2) + \Theta(n^2)
\end{aligned}
$$

- Disadvantage:
  - Tedious
  - Error-Prone
- Use to generate initial guess, and then prove by induction.

# Master Theorem

- Let $a$ and $b$ be constants, and let $f(n)$ be a nonnegative function defined on integral powers of $b$. Let $T(n)$ be defined on the integral powers of $b$ as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{if } n = b^k \end{cases}$$

Then we have:

•If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) = O\left(n^{\log_b a}\right)$

•If $f(n) = \Theta\left(n^{\log_b a}\right)$ for some constant $\varepsilon > 0$, then $T(n) = O\left(n^{\log_b a} \log n\right)$

•If $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $n \geq b \Rightarrow af(n/b) \leq cf(n)$

for some positive constant $c \geq 0$,then $T(n) = \Theta\left(f(n)\right)$

# Example

- Consider the recurrence $T(n)=T(n/2)+1$ (binary search)

  Then $a=1, b=2$ and $f(n)=1=n^{\log_2 1}$, so by case 2 of Master Theorem

  $T(n)=\Theta\left(n^{\log_2 1}\log n\right)=\Theta(\log n)$.

- Consider the recurrence $T(n)=2T(n/2)+n$ (merge sort)

  Then $a=2, b=2$ and $f(n)=n=n^{\log_2 2}$, so by case 2 of Master Theorem

  $T(n)=\Theta(n\log n)$.

- Consider the recurrence $T(n)=T(n/4)+n^{1/2}$

  Then $a=1, b=4$ and $f(n)=n^{1/2}=\Omega(n^{\log_2 1})$,

  and $af(n/b)=(n/4)^{1/2}=n^{1/2}/2=0.5f(n)$.

  So by case 3 of Master Theorem $T(n)=\Theta\left(n^{1/2}\right)$.

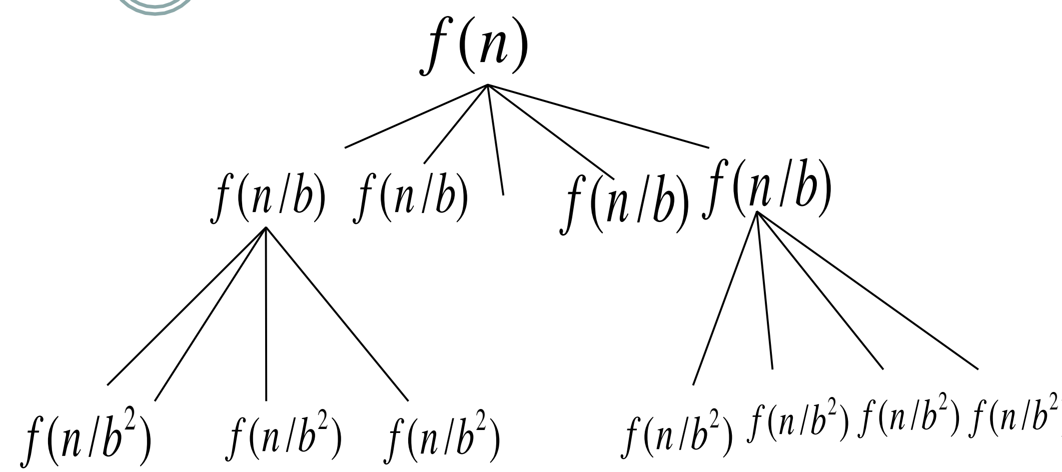
# Build recursive tree

- The tree:

$f(n)$

$af(n/b)$

$a^2 f(n/b^2)$




Last row : $\Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$elemenst, each one $\Theta(1)$.

Total : $\Theta(n^{\log_b a}) + \sum_{i=1}^{\log_b n-1} a^i f(n/b^i)$

Which term dominates?

# Back to Algorithms

- Quick Sort
  - Sort in place
  - Very practical
  - Divide-and-conquer
- Algorithm
  - Divide into two arrays around the first element
  - Recursively sort each array
  - Merge/combine-trivial

# Partition Routine

```
Partition(A,p,r)
    x=A(r)
    i=p-1
    for j=p to r-1
        if A(j)<=x then
            i++
            exchange(A(i),A(j))
    exchange(A(i+1),A(r))
    return(i+1)
```

| | | | |
|---|---|---|---|
| $<= x$ | $>x$ | ?? | |
| $p$ | $i$ | $j$ | $r$ |

# Quick Sort

Quicksort($A,p,r$)
    while ($p<r$)
        $q$=partition($A,p,r$)
        Quicksort($A,p,q$-1)
        Quicksort($A,q$+1,$r$)
    end

- To simplify, assume distinct elements: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
  - Lucky always an even element: $T(n) = 2T(0) + T(n-1) + \Theta(n) = \Theta(n^2)$
  - Unlucky:
- How to avoid bad case?
  - Partition around middle element (does not work!)
  - Idea: Partition around a random element!

# Quicksort (cont'd.)

- Partition around a ***Randomly*** chosen element and let ***T(n)*** be the *expected* time to sort.
- Consider the case where the partition is (***k,n-k-1***). In this case, the expected time to terminate is:

$$T(k)+T(n-k-1)+\Theta(n)$$

- Condition on ***k*** being a specific value, note that any value of ***k*** from ***0*** to ***n-1*** is equally likely:

$$\begin{aligned}
T(n) &= \sum_k \Pr[(k,n-k-1)\text{ split}]\,T(n \mid (k,n-k-1)\text{ split}) \\
&= \frac{1}{n}\sum_k [T(k)+T(n-k-1)+\Theta(n)] \\
&= \frac{2}{n}\sum_{k=1}^{n-1}[T(k)+\Theta(n)]
\end{aligned}$$

# Solving the recurrence

- Next:

We try to prove that $T(n) \le an\log n + b$

First, Choose $b$ large enough to satisfy $T(1) \le a\log 1 + b = b$

Inductive step:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \le \frac{2}{n}\sum_{k=1}^{n-1} ak\log k + b + \Theta(n)$$

$$= \frac{2a}{n} \underbrace{\sum_{k=1}^{n-1} k\log k}_{\text{Need to prove this is } \le \frac{1}{2}n^2\log n - \frac{1}{8}n^2} + \frac{2}{n}nb + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

$$= an\log n + b + \left(\Theta(n) + b - \frac{an}{4}\right)$$

# Technical Lemma

- We need to show $n^2 \log n$ bound is true.

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\left\lceil \frac{n}{2} - 1 \right\rceil} k \log k + \sum_{\left\lceil \frac{n}{2} \right\rceil}^{n-1} k \log k$$

$$\leq 2 \log n \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lceil \frac{n}{2} - 1 \right\rceil} k \right)$$

$$\leq \log n \frac{n(n-1)}{2} - \frac{\frac{n}{2}\left(\frac{n}{2} - 1\right)}{2}$$

$$\leq \frac{1}{2} n^2 \log n - \frac{n^2}{8}$$

# 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*

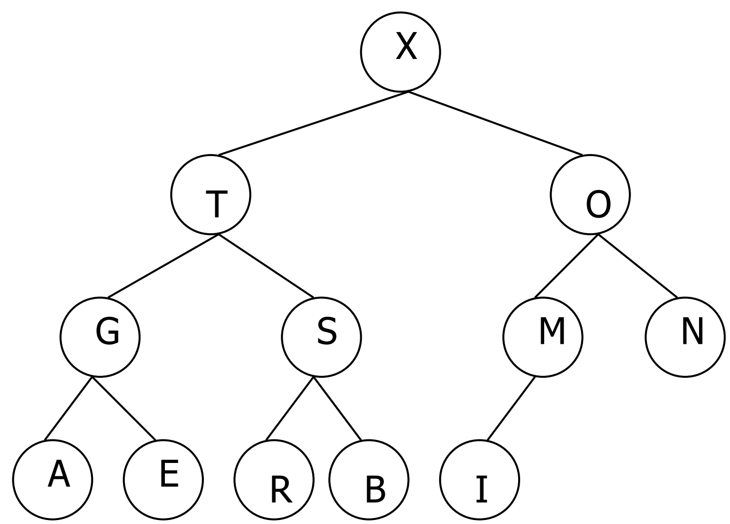
    *a[i] :=RemoveMax*

# 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

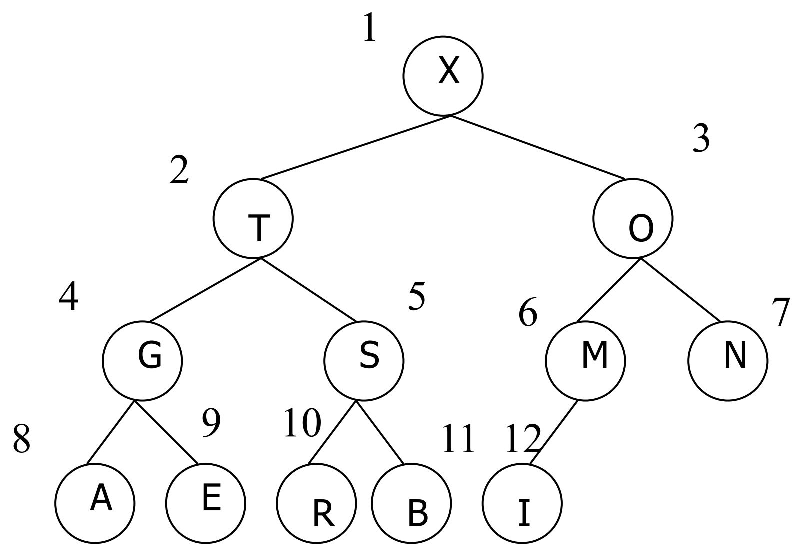
- A heap with $n$ keys has height: $H(n) = \log_2 n$
- *Proof:*

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

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

**Taking logarithm of both sides; the result will follow.**

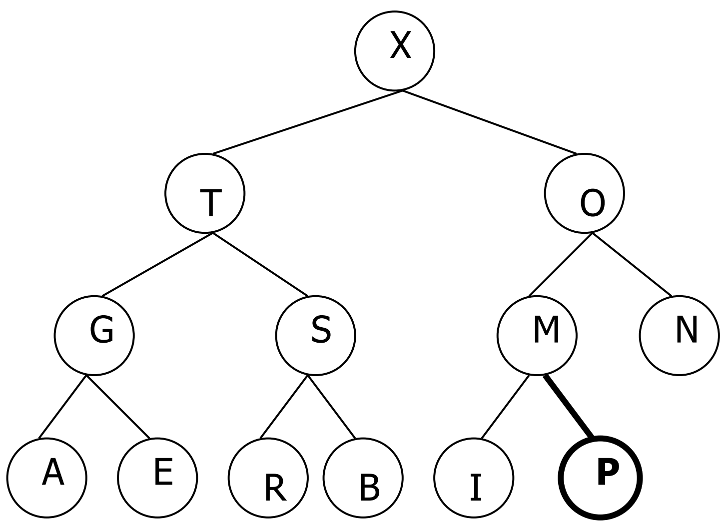
# Heap Representations

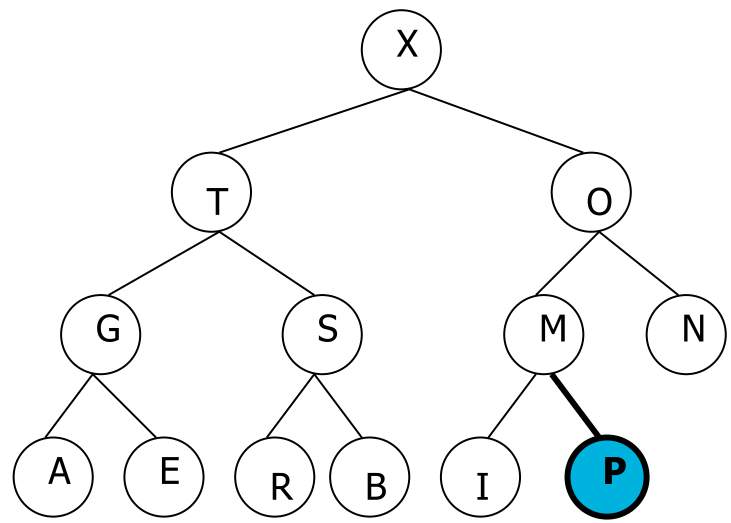



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| X | T | O | G | S | M | N | A | E | R | B | I |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

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

# Heap Insertion

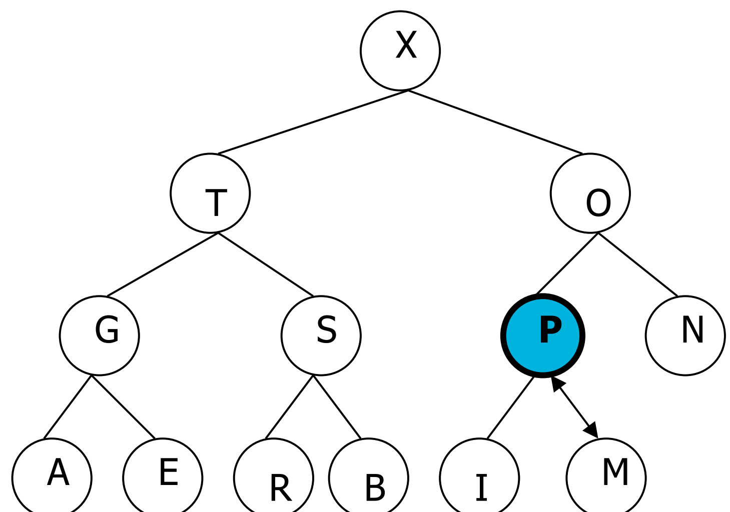


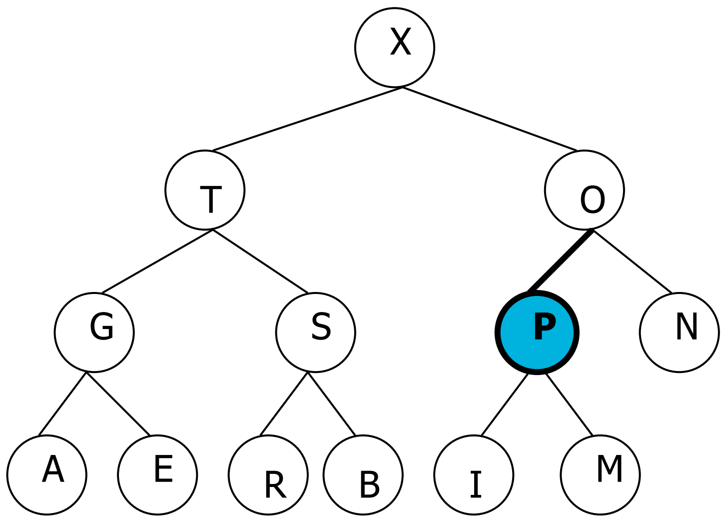

- Add the key in the next available spot in the heap.

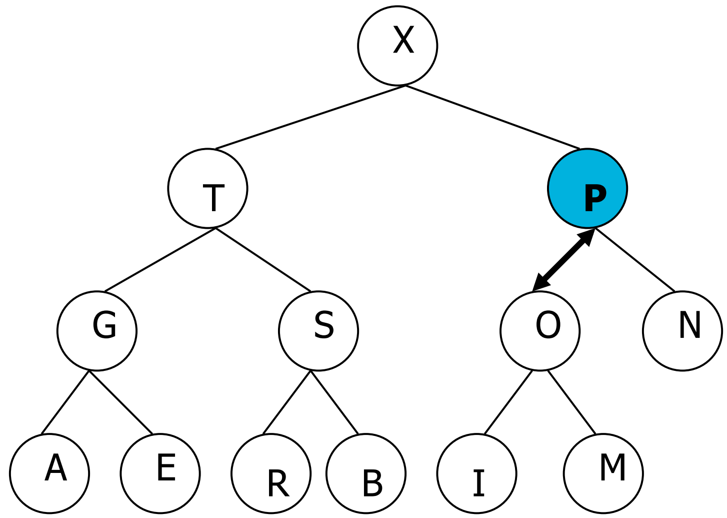
- ***Upheap*** checks if the new node is greater than its parent. If so, it switches the two.

- ***Upheap*** continues up the tree

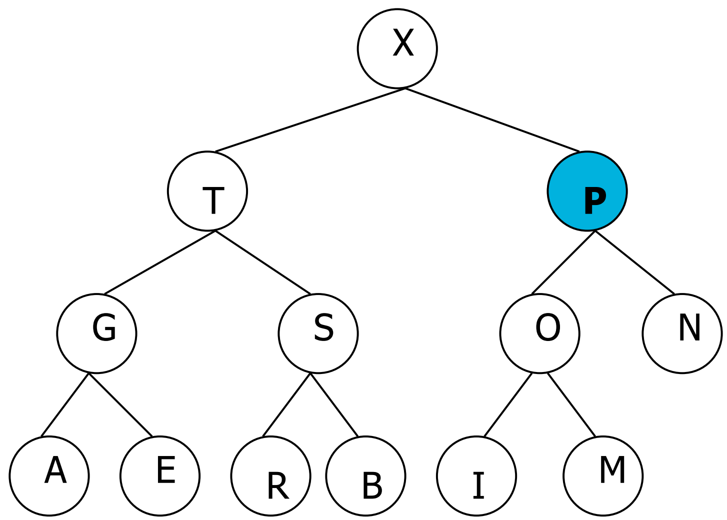


# Heap Insertion



# 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)*

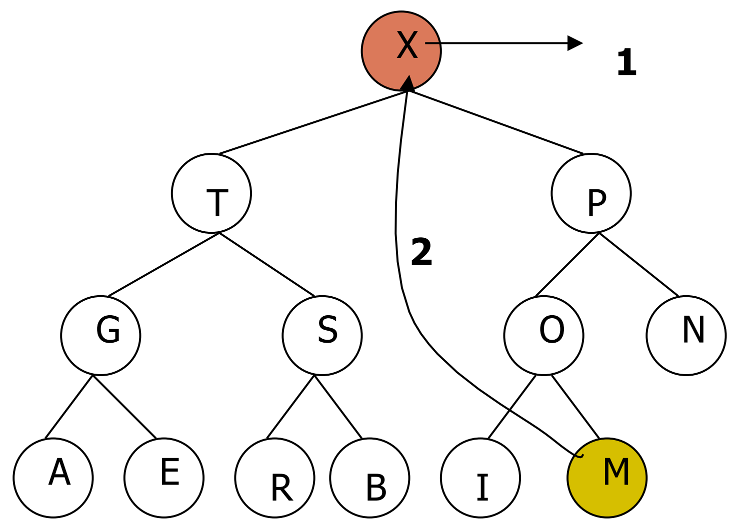
*Remove A[1]*

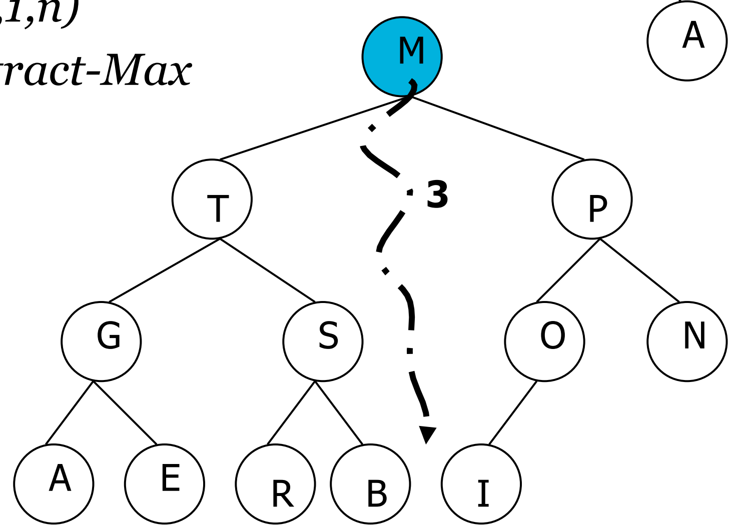
*A[1]=A[n]*

*n=n-1*

*Heapify(a,1,n)*

*End Heap-Extract-Max*



# Building a Heap

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

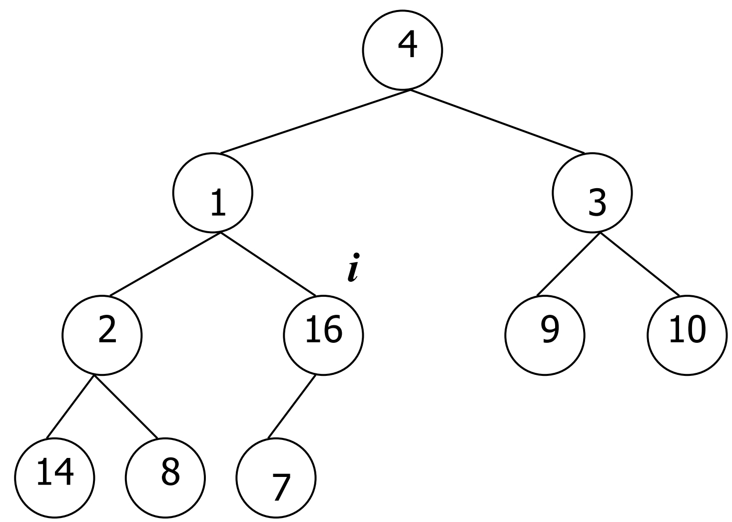
*Build_Heap(A,n)*

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

        *Heapify(A,i,n)*

*End Build_Heap*

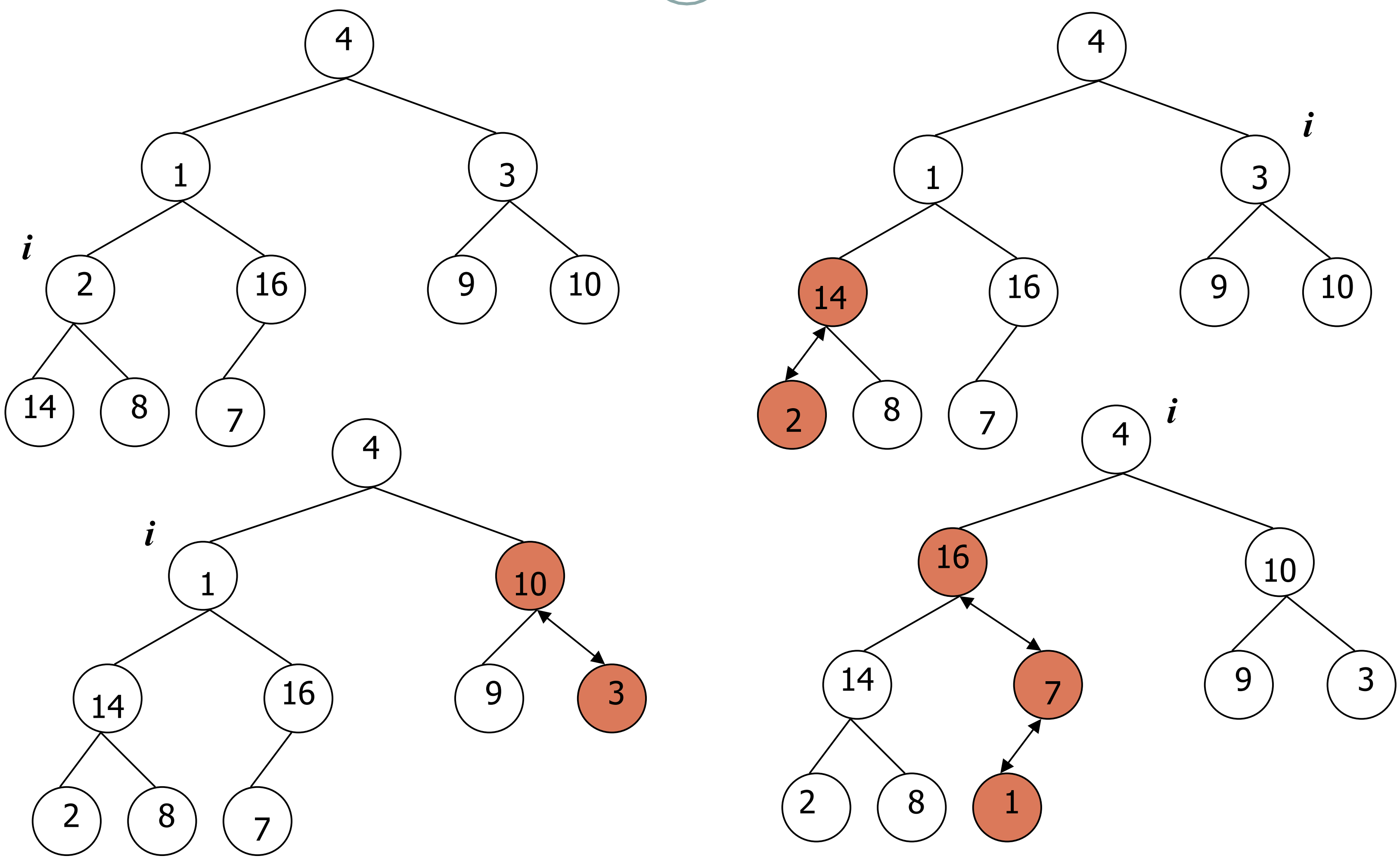
- **Example:**



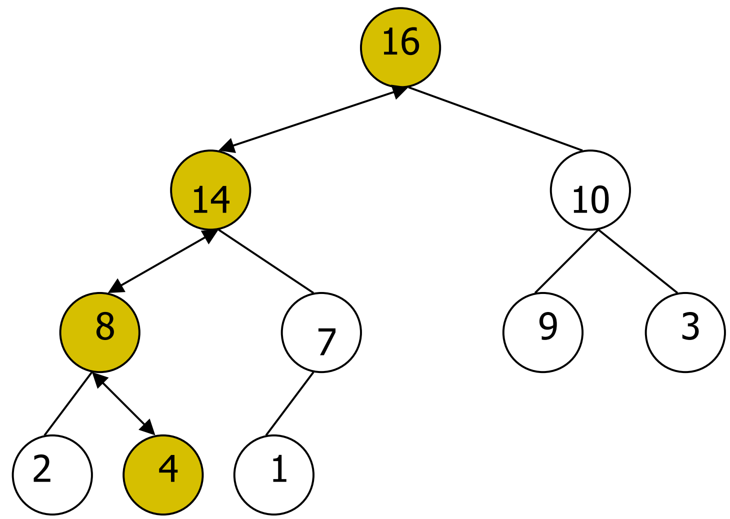

*A*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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**($A$,$n$)

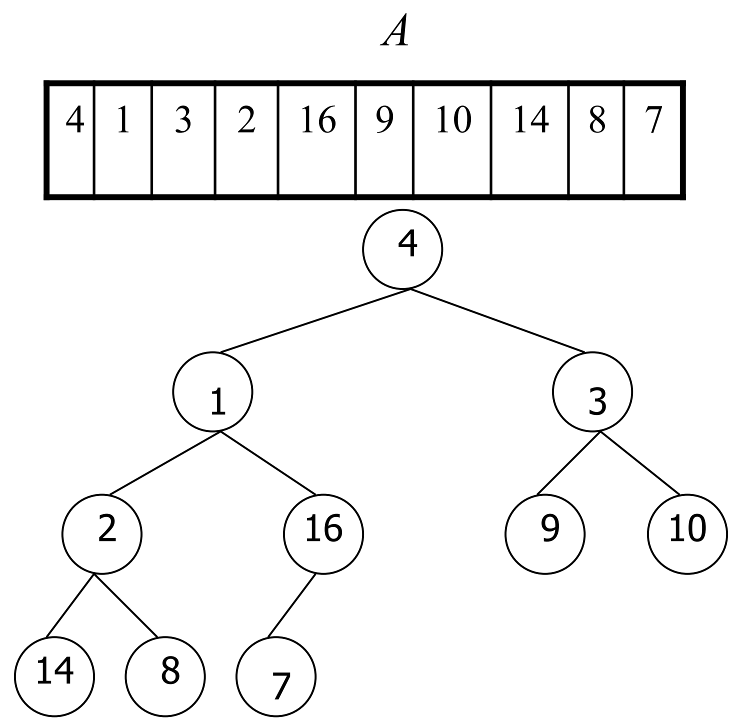
For $i$=$n$ downto 2 do

Exchange $A$[1] & $A$[$i$]

Heapify($A$,1,$i$ )

End For

End Heapsort

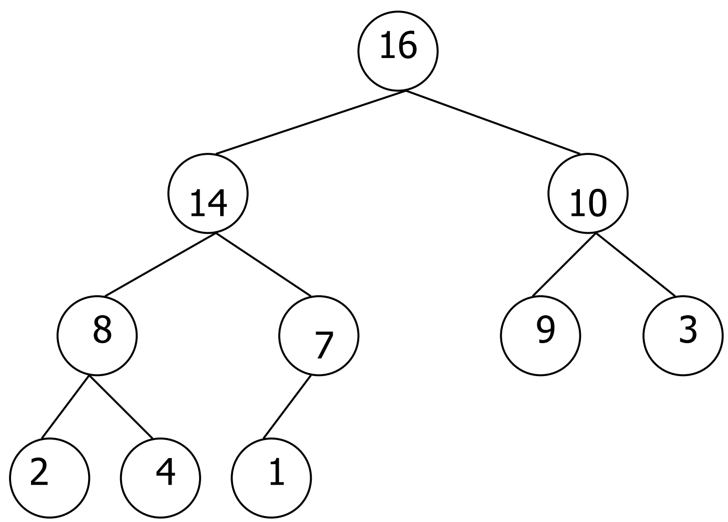


# Heapsort (cont'd.)

- First build the corresponding Heap:

*A*

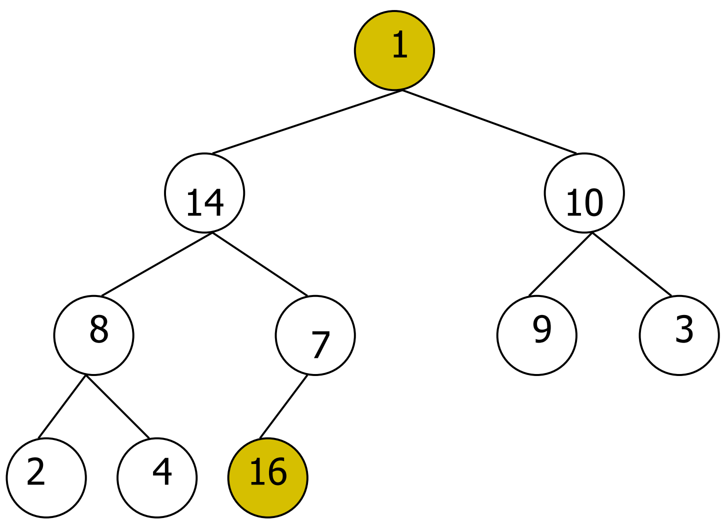
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |



# Heapsort (cont'd.)

*A*

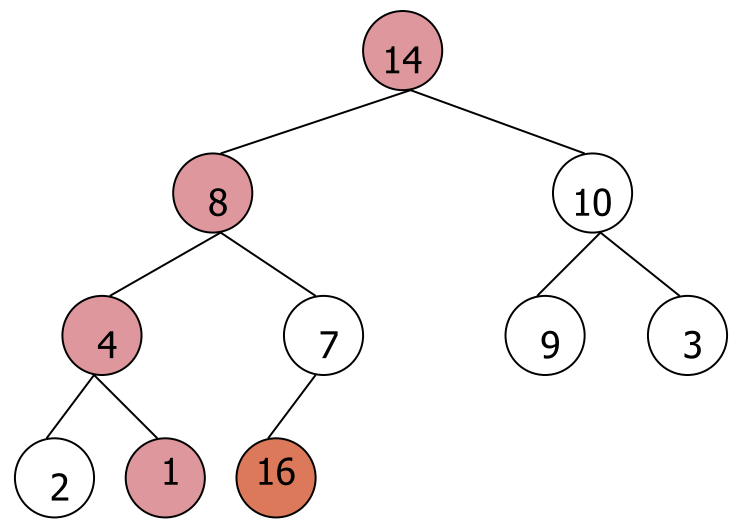
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 16 |




Exchange

*A*

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

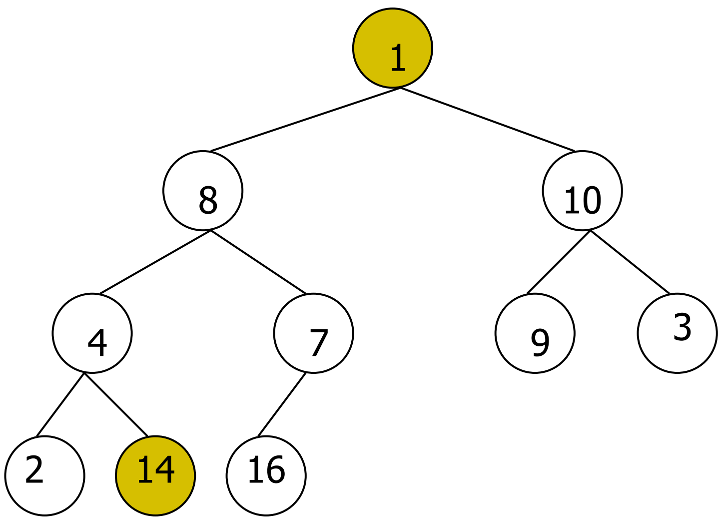



Heapify($A$,1,$n$-1)

# Heapsort (cont'd.)

*A*

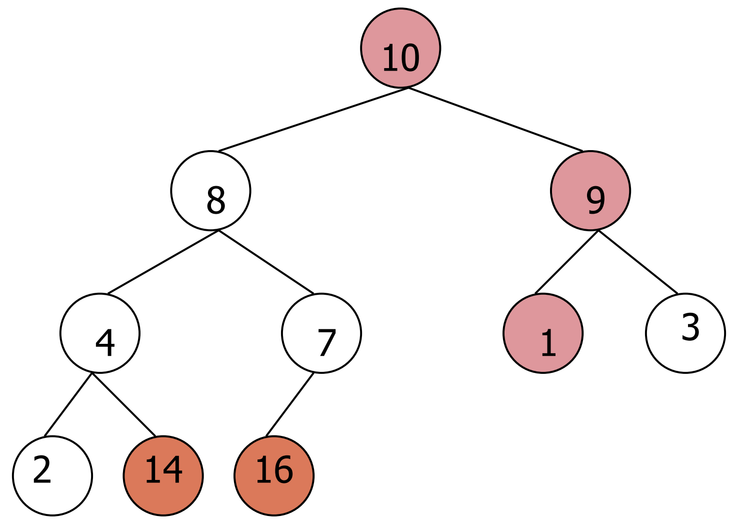
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 10 | 4 | 7 | 9 | 3 | 2 | 14 | 16 |




Exchange

*A*

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

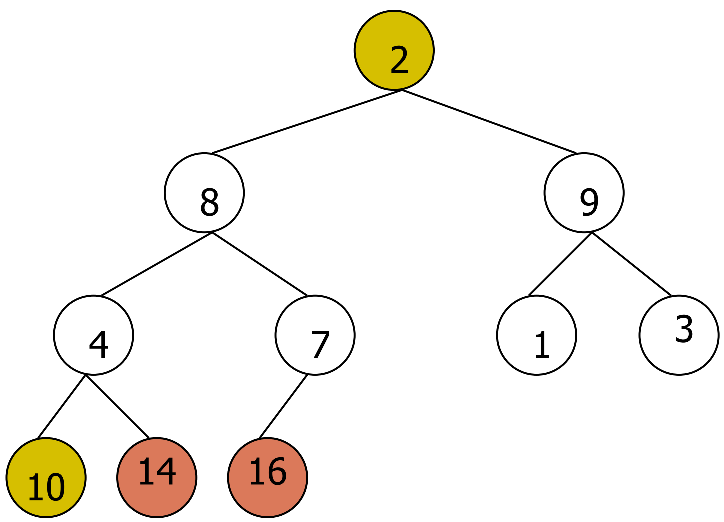



Heapify($A$,1,$n$-2)

# Heapsort (cont'd.)

*A*

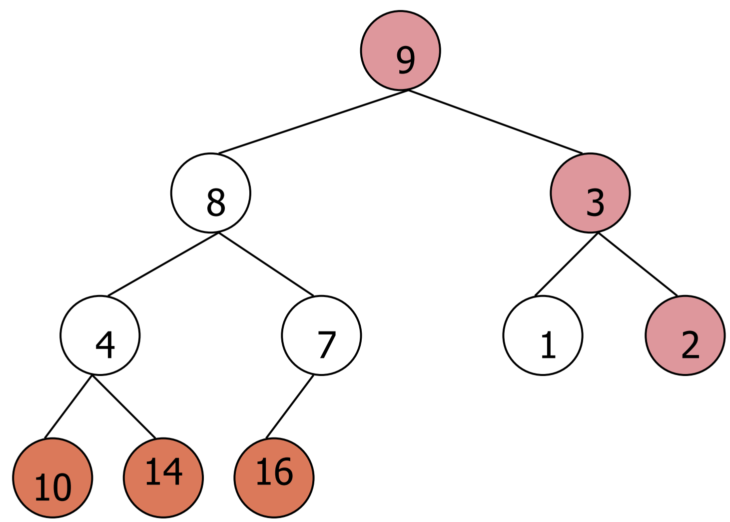
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 9 | 4 | 7 | 1 | 3 | 10 | 14 | 16 |




Exchange

*A*

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

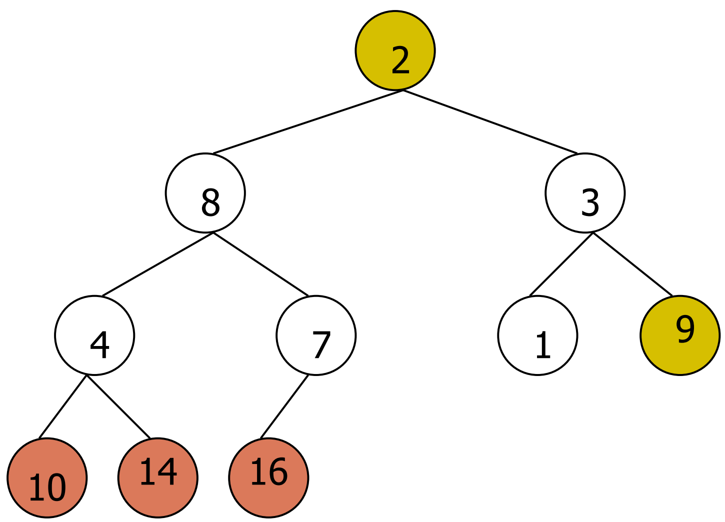



Heapify($A$,1,$n$-3)

# Heapsort (cont'd.)

*A*

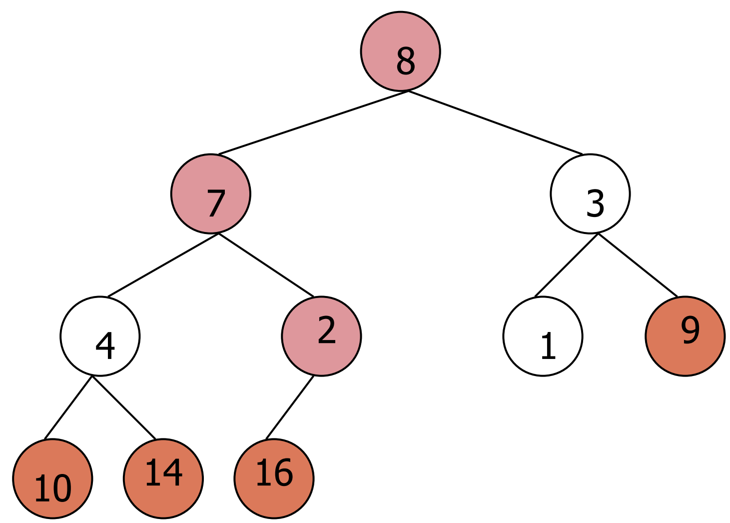
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 3 | 4 | 7 | 1 | 9 | 10 | 14 | 16 |




Exchange

*A*

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

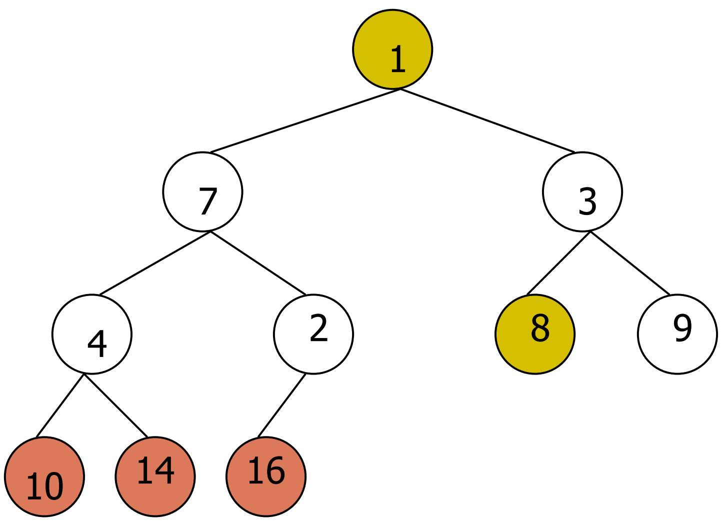



Heapify($A$,1,$n$-4)

# Heapsort (cont'd.)

*A*

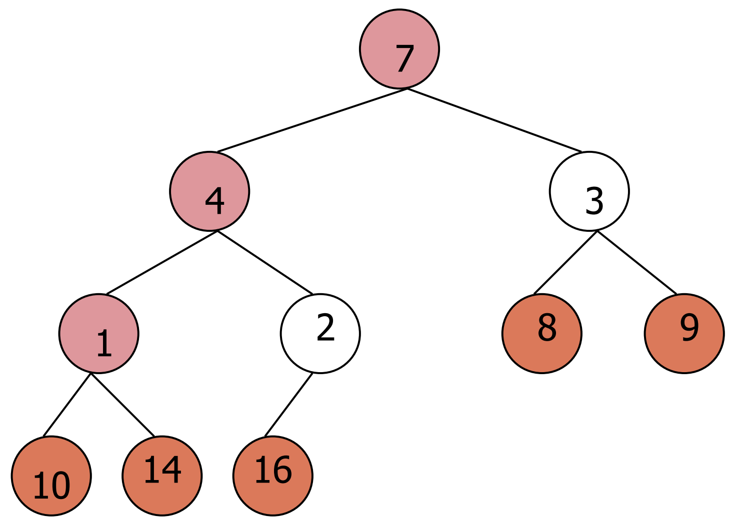
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 7 | 3 | 4 | 2 | 8 | 9 | 10 | 14 | 16 |




Exchange

*A*

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



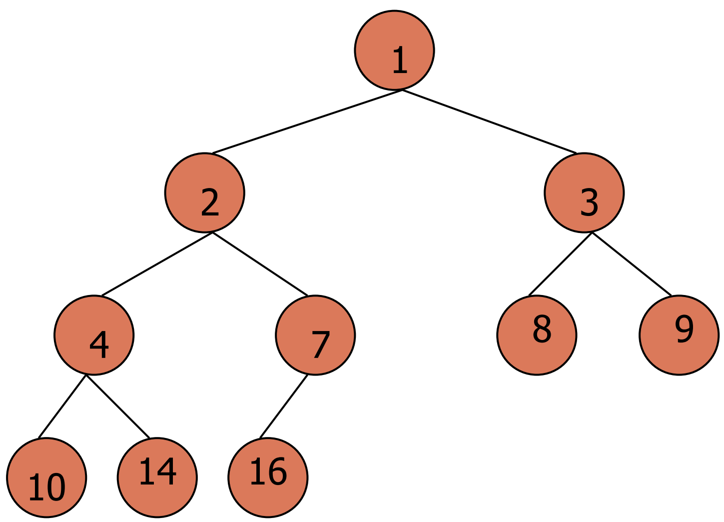

Heapify($A$,1,$n$-5)

# 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: $O(n \log n)$