2. Divide-and-Conquer Approach

Pukar Karki
Assistant Professor

Recursion

- A programming technique in which a function calls itself.
- One of the most effective techniques in programming that makes problem solving conceptually simple.

Recursive Algorithms

Algorithm 1 : factorial

```
fact(n)
{
if(n == 0)
    return 1;
else
    return n*fact(n-1);
}
```

Recursive Algorithms

Algorithm 2 : fibonacci term

```
fibo(n)
{
if (n == 1 || n==2)
    return 1;
else
    return fibo(n-1) + fibo(n-2);
}
```

Recursive Algorithms

Algorithm 3: GCD

```
GCD(a,b)
{
if (b==0)
    return a;
else
    return GCD(b,a%b);
}
```

Recurrence Relations

- Consider the recursive algorithm for computing factorial of a number.

```
fact(n)
{
if (n == 0) then
    return 1; // base case
else
    return n*fact(n-1); // recursive call
}
```

T(0) = 1

T(n) = T(n – 1) + O(1) for n > 0

Recurrence Relations

- Consider the recursive algorithm for computing fibonacci term

```
fibo(n)
{
if (n == 1 || n == 2) then
    return 1; // base case
else
    return fibo(n-1)+fibo(n-2); // recursive call
}
```

1 1 2 3 5 8 . . . . .

T(1) = 1

T(2) = 1

T(n) = T(n – 1) + T(n-2) + O(1) for n > 2

Recurrence Relations

T(0) = 1

T(n) = T(n – 1) + O(1) for n > 0

- A recurrence is an equation or inequality that describes a function in terms of it's values on smaller inputs.

Recurrence Relations

T(0) = 1

T(n) = T(n – 1) + 2 for n > 0

- Let us expand the above equation

T(0) = 1

✔ T(n) = T(n – 1) + 2 for n > 0

✔ T(n) = T((n – 1) - 1) + 2 + 2 = T(n – 2) + 2.2

✔ T(n) = T((n-2) – 1) + 2 + 4 = T(n – 3) + 2.3

. . . . .

T(n) = T(n – k) + 2k

We want it to express it in terms of T(0) so, n - k = 0 i.e. n = k.

✔ **T(n) = T(0) + 2n**

✔ **T(n) = 1 + 2n = c*g(n) where c = 3 and g(n) = n.**

✔ **Therefor, T(n) = O(n)**

Recurrence Relations

- Solving a recurrence means that we have to obtain a function defined on the natural numbers that satisfy the recurrence.
- To analyze the complexity of recursive algorithms, we represent them in terms of recurrence relation and use any of the recurrence relation solving method.

Solving Recurrences

We will study the following methods to solve recurrences in this course.

1. Iteration method.
2. Recursion Tree method.
3. Substitution method.
4. Master method.

Iteration Method

- In this method, we expand the given recurrence relation until the boundary condition is met.
- Look at the following example.

Iteration Method

Example 1

Solve the following recurrence relation by using iterative method.

T(n) = 2T(n/2) + 1 when n > 1

T(n) = 1 when n = 1

- ✔ $T(n) = 2T(n/2) + 1$
- ✔ $T(n) = 2\{2T(n/4) + 1\} + 1 = 2^2T(n/2^2) + 2 + 1$
- ✔ $T(n) = 2^2\{2T(n/2^3) + 1\} + 2 + 1 = 2^3T(n/2^3) + 2^2 + 2 + 1$
- ✔ ……
- ✔ $T(n) = 2^kT(n/2^k) + 2^{k-1} + \ldots\ldots\ldots + 4 + 2 + 1$

Iteration Method

- ✔ $T(n) = 2^k T(n/2^k) + 2^{k-1} + \ldots\ldots + 4 + 2 + 1$ **---------- Eqn i)**
- ✔ Assume, $n/2^k = 1$ or, $n = 2^k$
- ✔ Taking $\log_2$ on both sides, $\log_2 n = \log_2 2^k$
- ✔ Or, $\log_2 n = k \log_2 2$
- ✔ $k = \log_2 n$

Iteration Method

- ✔ Putting the value of $n/2^k$ in in **Eqn i)**
- ✔ $T(n) = 2^kT(1) + 2^{k-1} + \ldots\ldots\ldots + 4 + 2 + 1$
- ✔ $T(n) = 2^k + 2^{k-1} + \ldots\ldots\ldots + 4 + 2 + 2^0$
- ✔ $T(n) = 1(2^{k+1} - 1)/(2\text{-}1)$ **{use $S_n = a(r^n - 1)/(r - 1)$}**
- ✔ $T(n) = 2.2^k - 1$
- ✔ $T(n) = 2.n - 1$
- ✔ $T(n) = O(n)$

Iteration Method

Example 2

Solve the following recurrence relation by using iterative method.

T(n) = T(n/3) + O(n) when n > 1

T(n) = 1 when n = 1

- ✔ $T(n) = T(n/3) + O(n)$
- ✔ $T(n) = T(n/3) + cn$
- ✔ $T(n) = T(n/3^2) + cn/3 + cn$
- ✔ $T(n) = T(n/3^3) + cn/3^2 + cn/3 + cn$
- ✔ …..
- ✔ $T(n) = T(n/3^k) + cn/3^{k-1} + \ldots\ldots + cn/3^2 + cn/3 + cn$

Iteration Method

- ✔ $T(n) = T(n/3^k) + cn/3^{k-1} + \ldots\ldots + cn/3^2 + cn/3 + cn$ **---- Eqn i**)
- ✔ Assume $n/3^k = 1$ or, $n = 3^k$
- ✔ Taking $\log_3$ on both sides, $\log_3 n = \log_3 3^k$
- ✔ Or, $\log_3 n = k \log_3 3$
- ✔ $k = \log_3 n$

Iteration Method

- ✔ Putting the value of $n/3^k$ in in **Eqn I)**
- ✔ $T(n) = T(n/3^k) + cn/3^{k-1} + \ldots\ldots + cn/3^2 + cn/3 + cn$
- ✔ $T(n) = T(1) + cn/3^{k-1} + \ldots\ldots + cn/3^2 + cn/3 + cn$
- ✔ $T(n) = 1 + \{ cn/3^{k-1} + \ldots\ldots + cn/3^2 + cn/3 + cn\}$
- ✔ $T(n) = 1 + cn(1/3^{k-1} + \ldots\ldots + 1/3^2 + 1/3 + 1\}$
- ✔ $T(n) = 1 + cn\{1.(1 – 1/3^k)/(1-1/3)\}$
- ✔ $T(n) = 1 + cn\{(1 – 1/n)/(2/3)\}$
- ✔ $T(n) = 1 + 3c(n-1)/2$
- ✔ $T(n) = O(n)$

Iteration Method

Example 3

Solve the following recurrence relation by using iterative method.

T(n) = T(n-1) + O(1) when n > 1

T(n) = 1 when n = 1

- ✔ T(n) = T(n-1) + O(1)
- ✔ T(n) = T(n-1) + 1
- ✔ T(n) = T(n-2) + 1 + 1
- ✔ T(n) = T(n-3) + 1 + 1 + 1
- ✔ …..
- ✔ T(n) = T(n-k) + 1 + …… + 1 + 1(k times)

Iteration Method

- ✔ T(n) = T(n-k) + k **---- Eqn i**)
- ✔ Assume n-k = 1 or, n = 1 + k
- ✔ k = **n - 1**

Iteration Method

- ✔ Putting the value of n-k in in **Eqn I)**
- ✔ $T(n) = T(n-k) + k$
- ✔ $T(n) = T(1) + k$
- ✔ $T(n) = 1 + n - 1$
- ✔ $T(n) = n$
- ✔ $T(n) = O(n)$

Iteration Method

Example 4

Solve the following recurrence relation by using iterative method.

T(n) = 2T(n/2) + n when n > 1

T(n) = 1 when n = 1

- ✔ $T(n) = 2T(n/2) + n$
- ✔ $T(n) = 2\{2T(n/4) + n/2\} + n = 2^2T(n/2^2)\} + n + n$
- ✔ $T(n) = 2^3 T(n/2^3)\} + n+ n + n$
- ✔ …..
- ✔ $T(n) = 2^kT(n/2^k) + n + ….. + n + n$ (k times)

Iteration Method

- ✔ $T(n) = 2^kT(n/2^k) + n + \ldots. + n + n$ (k times) **---- Eqn i**)
- ✔ Assume $n/2^k = 1$ or, $n = 2^k$
- ✔ Taking $\mathbf{log_2}$ on both sides, $\mathbf{log_2}\ n = \mathbf{log_2}\ 2^k$
- ✔ Or, $\mathbf{log_2}\ n = k\ \mathbf{log_2}\ 2$
- ✔ $k = \mathbf{log_2}\ n$

Iteration Method

- ✔ Putting the value of $n/2^k$ in in **Eqn I)**
- ✔ $T(n) = 2^kT(n/2^k) + n + ….. + n + n$ (k times)
- ✔ $T(n) = n.T(1) + n.k$
- ✔ $T(n) = n.1 + n.k$
- ✔ $T(n) = k.n + n$
- ✔ $T(n) = \mathbf{log_2}\ n*n + n$
- ✔ $T(n) = n * \mathbf{log_2}\ n + n$
- ✔ $T(n) = O(n\ \mathbf{log_2}\ n)$

Recursion Tree

- This method is a pictorial representation of the iteration method.
- It takes the form of the tree where at each level nodes are expanded.
- It diagrams the tree of recursive calls and the amount of work done at each call.

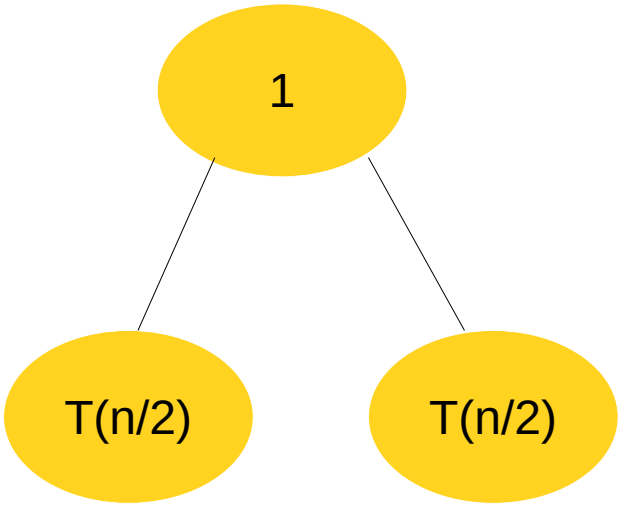
Recursion Tree

Example 1

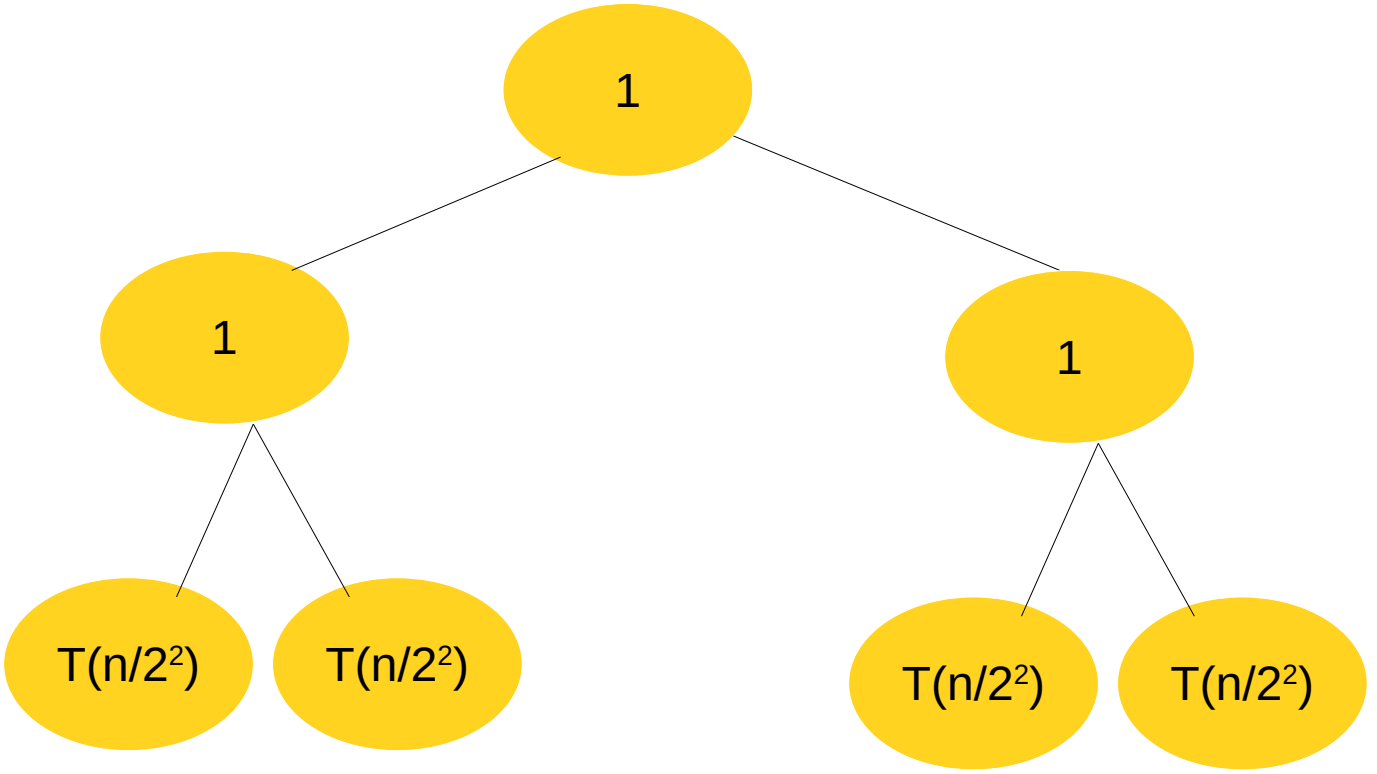
Solve the following recurrence relation by using recursion tree method.

T(n) = 2T(n/2) + 1 when n > 1

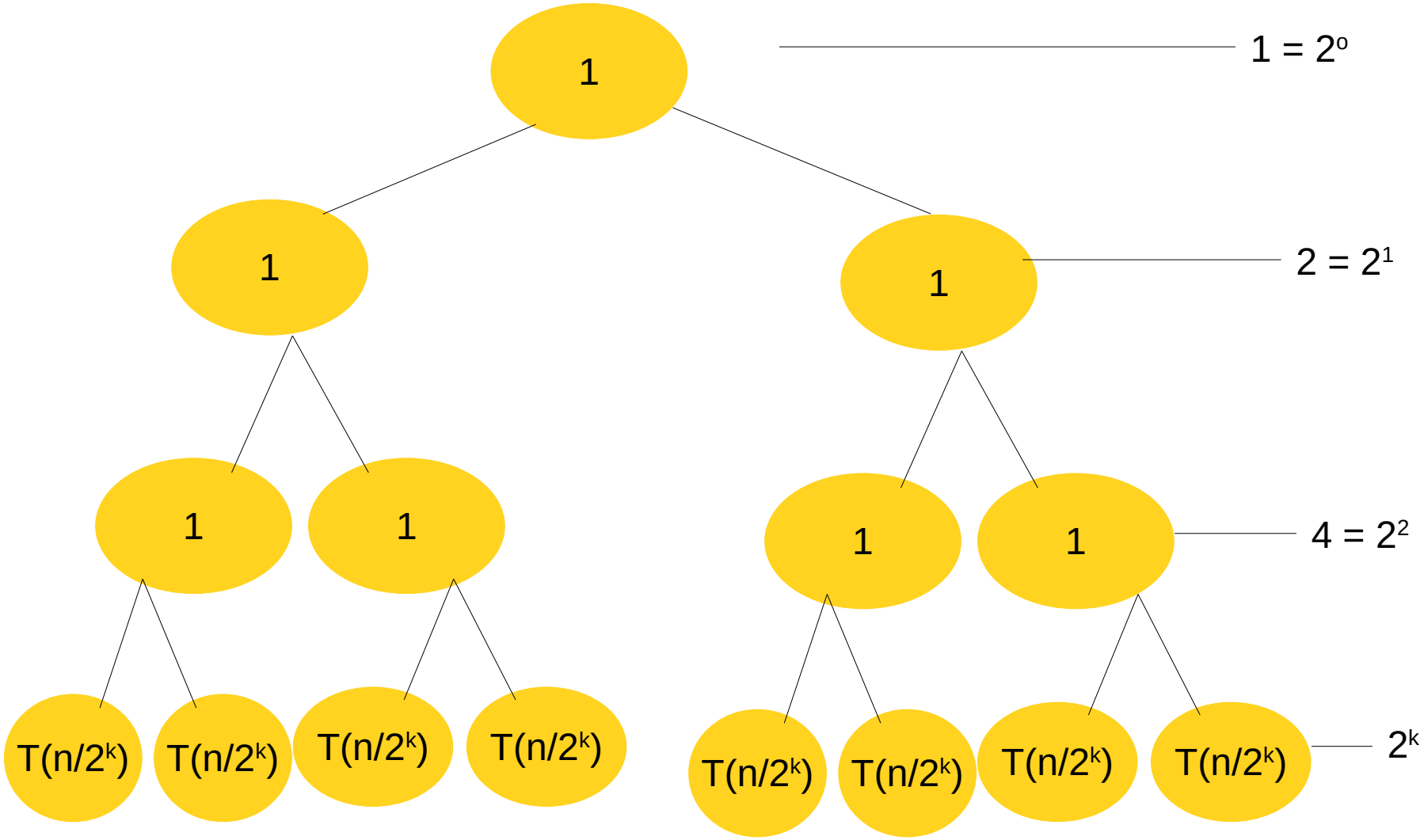
T(n) = 1 when n = 1



Recursion Tree



Recursion Tree



Recursion Tree

- $T(n) = 2^0 + 2^1 + \ldots\ldots + 2^k$
- $T(n) = 1 + 2(2^k - 1)/(2 - 1)$ **[Use $S_n = a(r^n - 1)/(r - 1)$]**
- $T(n) = 1 + 2(2^k - 1)$
- $T(n) = 2.2^k - 1$ **---------- Eqn i)**

Recursion Tree

- Assume, $\mathbf{n/2^k = 1}$ or $\mathbf{n = 2^k}$
- Taking $\mathbf{log_2}$ on both sides, $\mathbf{log_2}\ n = \mathbf{log_2}\ 2^k$
- Or, $\mathbf{log_2}\ n = k\ \mathbf{log_2}\ 2$
- $k = \mathbf{log_2}\ n$
- Putting the value of 2^k in in **Eqn i)**
- $T(n) = 2.2^k - 1 = 2.n - 1$
- $T(n) = O(n)$

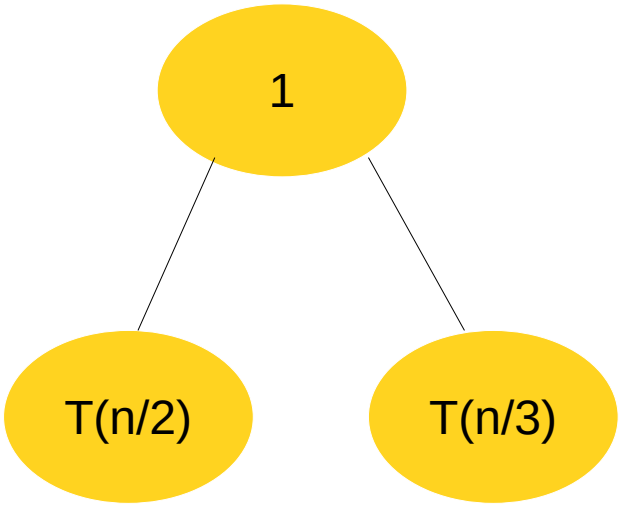
Recursion Tree

Example 2

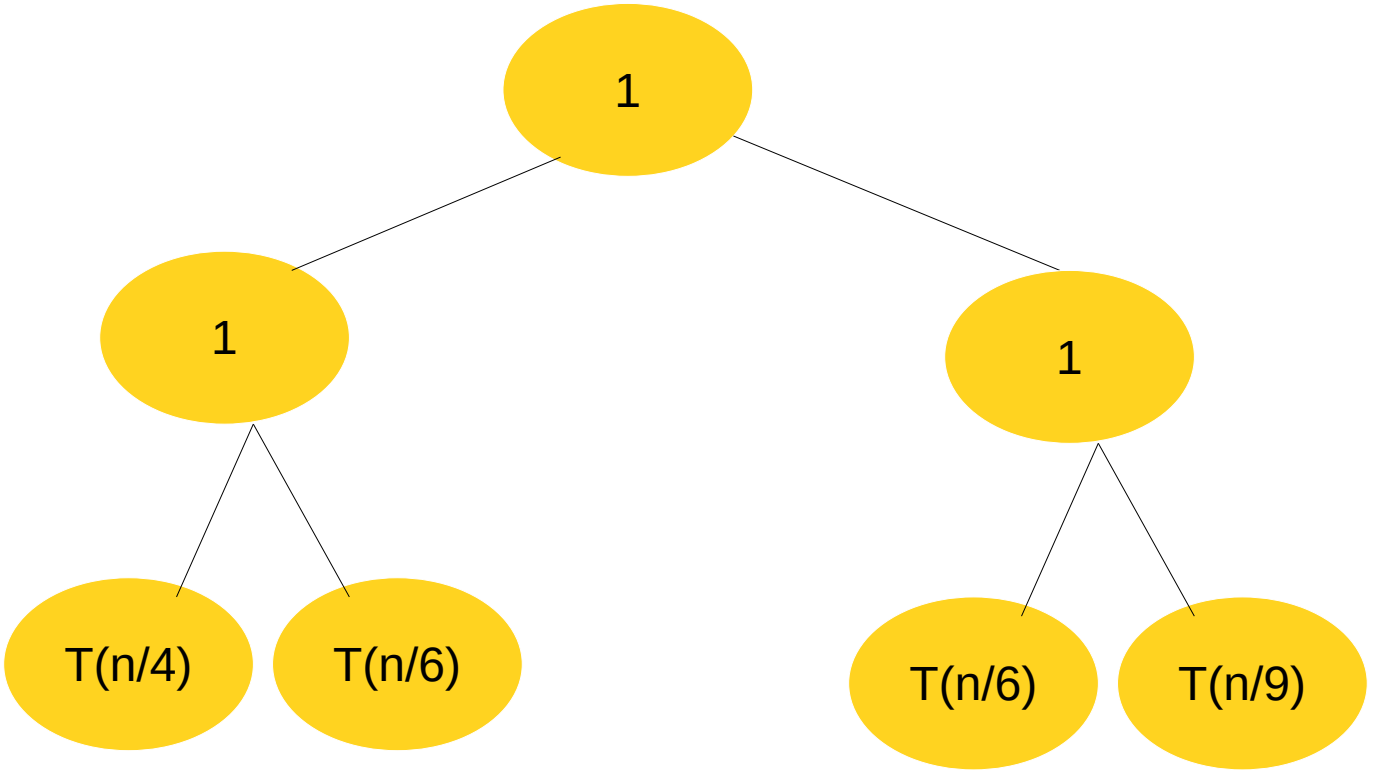
Solve the following recurrence relation by using recursion tree method.

T(n) = T(n/2) + T(n/3) + O(1) when n > 1

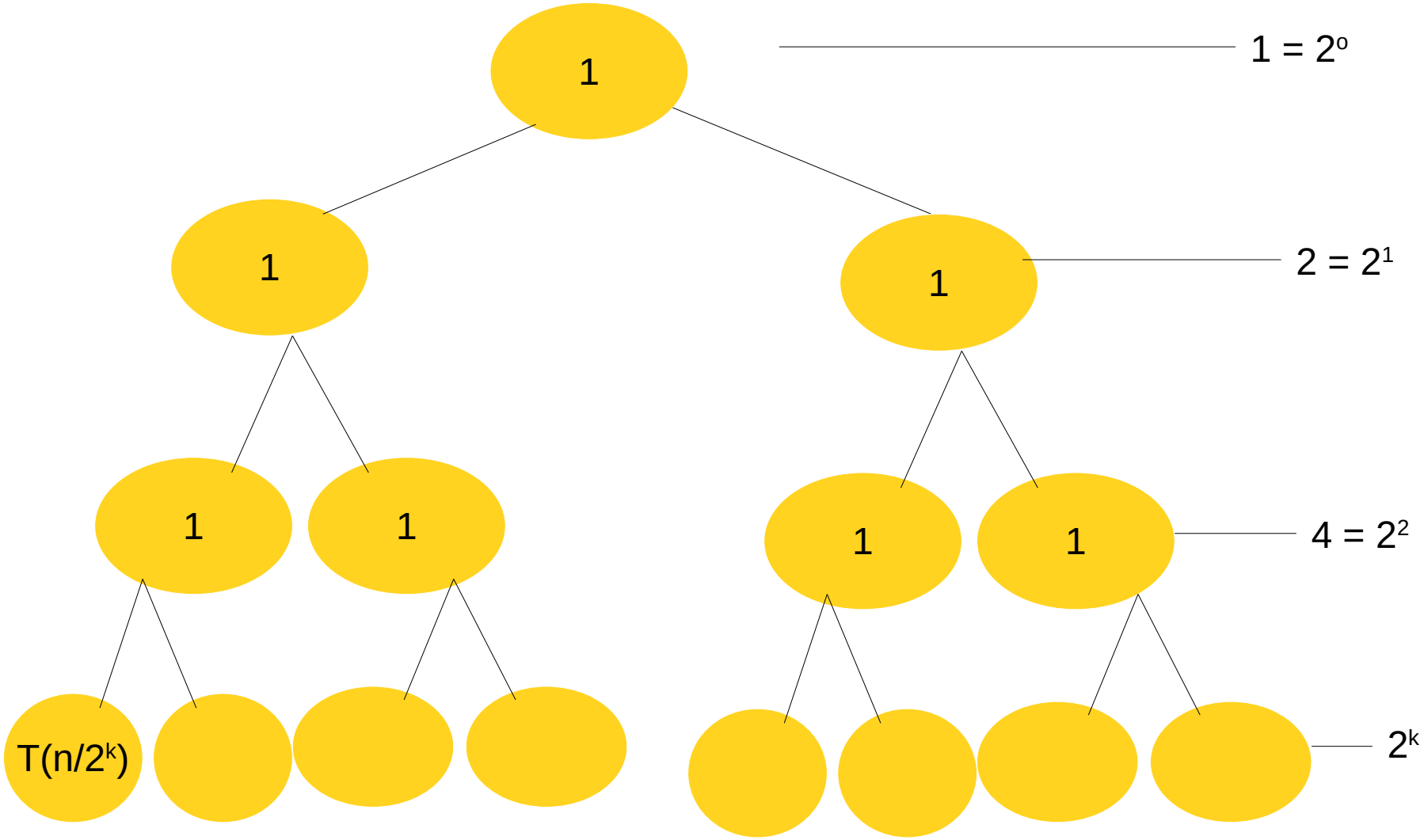
T(n) = 1 when n = 1



Recursion Tree



Recursion Tree



Recursion Tree

- $T(n) \leq 2^0 + 2^1 + \ldots\ldots + 2^k$
- $T(n) = 1 + 2(2^k - 1)/(2 - 1)$
- $T(n) = 1 + 2(2^k - 1)$
- $T(n) = 2.2^k - 1$ **---------- Eqn i)**

Recursion Tree

- Assume, $n/2^k = 1$ or $n = 2^k$
- Taking $\log_2$ on both sides, $\log_2 n = \log_2 2^k$
- Or, $\log_2 n = k \log_2 2$
- $k = \log_2 n$
- Putting the value of 2^k in in **Eqn i)**
- $T(n) \leq 2.2^k - 1$
- $T(n) \leq 2.n - 1$
- $T(n) = O(n)$

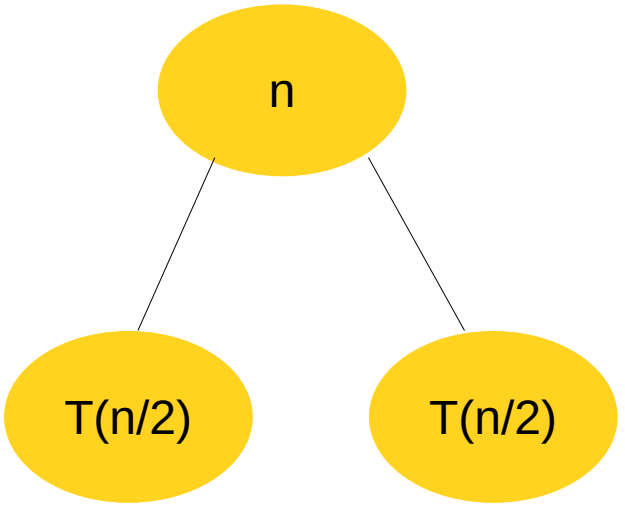
Recursion Tree

Example 3

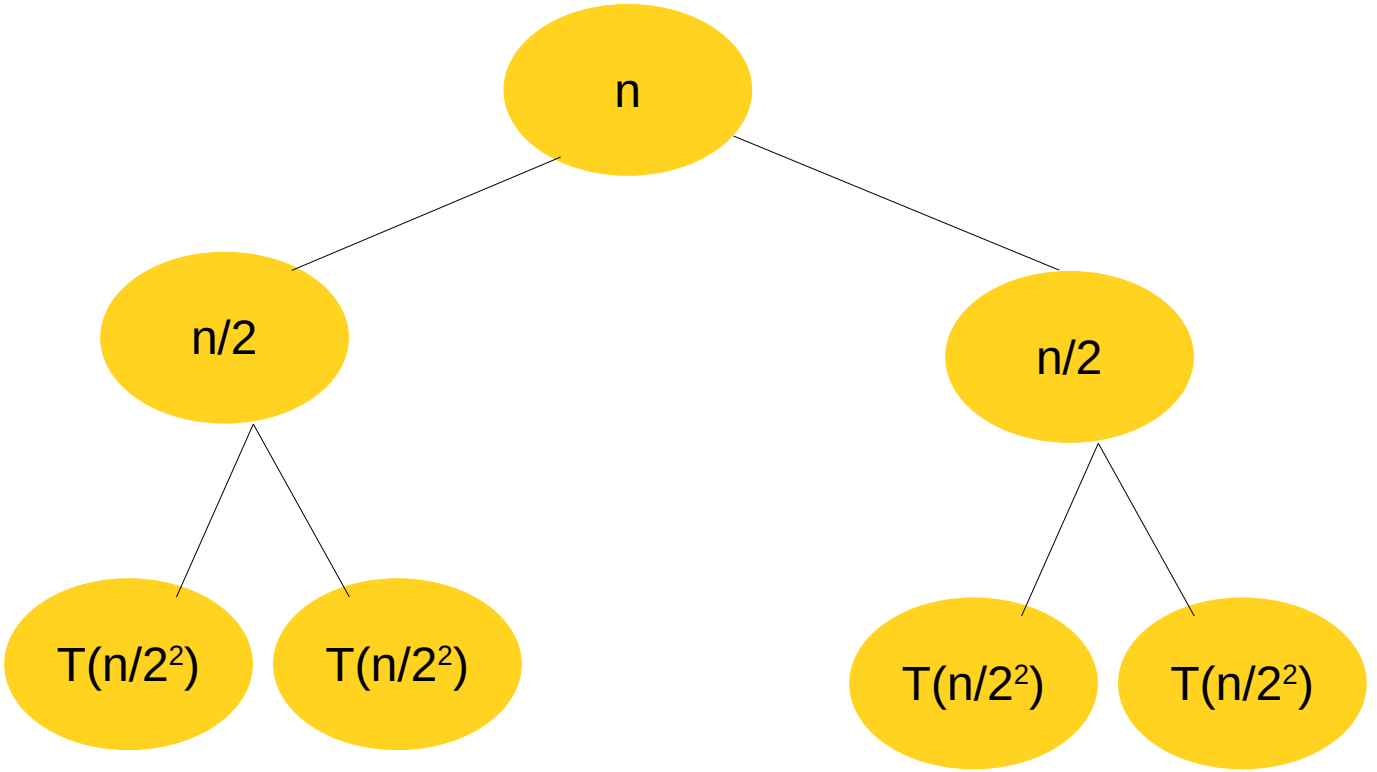
Solve the following recurrence relation by using recursion tree method.

T(n) = 2T(n/2) + n when n > 1

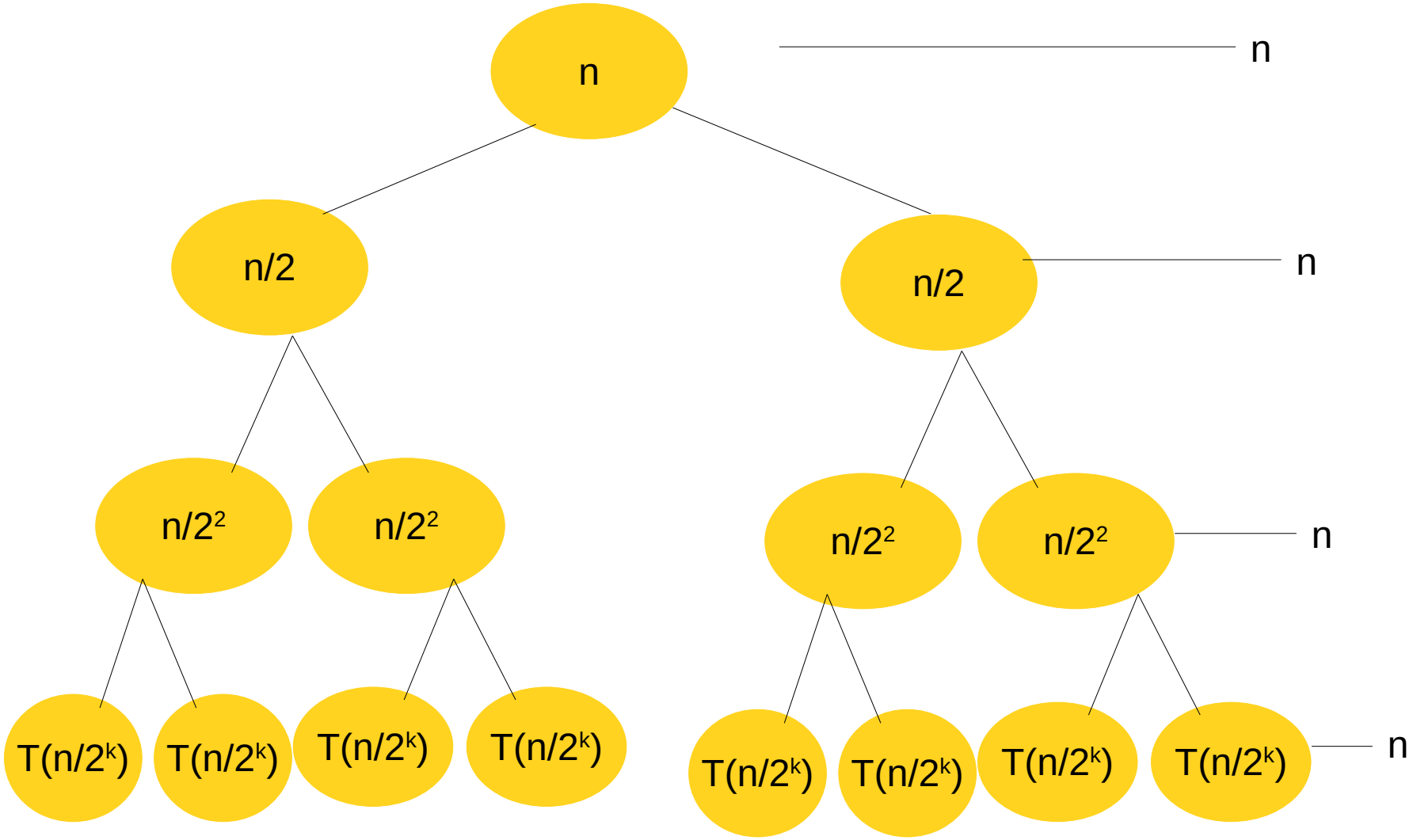
T(n) = 1 when n = 1



Recursion Tree



Recursion Tree



Recursion Tree

- T(n) = n + n + … + n (k times)
- T(n) = n.k **---------- Eqn i)**

Recursion Tree

- Assume, $n/2^k = 1$ or $n = 2^k$
- Taking $\log_2$ on both sides, $\log_2 n = \log_2 2^k$
- Or, $\log_2 n = k \log_2 2$
- $k = \log_2 n$
- Putting the value of k in in **Eqn i)**
- $T(n) = n.k$
- $T(n) = n \log_2 n$
- $T(n) = O(n \log_2 n)$

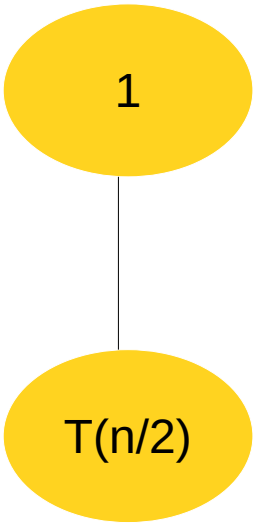
Recursion Tree

Example 4

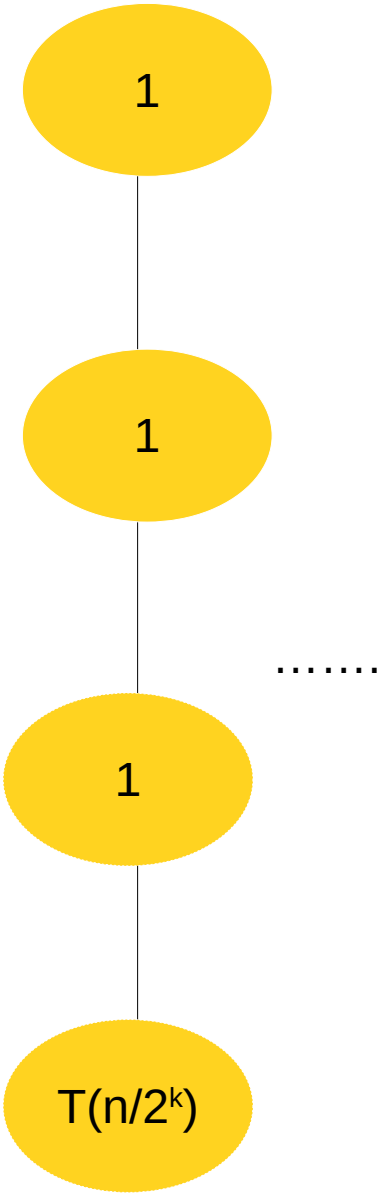
Solve the following recurrence relation by using recursion tree method.

T(n) = T(n/2) + 1 when n > 1

T(n) = 1 when n = 1



Recursion Tree



Recursion Tree

- $T(n) = 1 + 1 + 1 + \ldots + T(n/2^k)$ **---------- Eqn I)**
- Assume, $n/2^k = 1$ or $n = 2^k$
- Taking $\log_2$ on both sides, $\log_2 n = \log_2 2^k$
- Or, $\log_2 n = k \log_2 2$
- $k = \log_2 n$
- Putting the value of $n/2^k$ in in **Eqn i)**
- $T(n) = 1 + 1 + 1 + \ldots + T(n/2^k)$
- $T(n) = 1 + 1 + 1 + \ldots\ldots + 1(k \text{ times}) + T(1)$
- $T(n) = k*1 + 1$
- $T(n) = \log_2 n + 1 = O(\log_2 n)$

Recursion Tree

Homework

Solve the following recurrence relation by using recursion tree method.

T(n) = T(n-1) + 1 when n > 1

T(n) = 1 when n = 1

Recursion Tree

Homework

Solve the following recurrence relation by using recursion tree method.

$T(n) = T(n/4) + T(n/2) + n^2$ when $n > 1$

$T(n) = 1$ when $n = 1$

Substitution Method

- At first, we guess the form of solution.
- Then, we use induction to show that the guess is valid.

Substitution Method

Example 1

Show that the complexity if the following RR is $O(n^3)$ using substitution method.

$T(n) = 4T(n/2) + n$ for $n > 1$

$T(n) = 1$ for $n = 1$

- ✔ Our guess is $T(n) = O(n^3)$
- ✔ From the definition of Big O, $T(n) \leq c*n^3$ where $c>0$ and for all $n>n_0$ **……. Eqn i)**
- ✔ Now we show that Eqn i) is true using mathematical induction.

Substitution Method

Basic Step:

For n = 1

- ✔ $T(1) \leq c.\ 1^3$
- ✔ $1 \leq c$ which is true for all positive values of c.

Inductive Step:

- ✔ Let us assume that it is true for all k < n.
- ✔ Then, $T(k) \leq c.\ k^3$ **……. Eqn ii)**
- ✔ Since it is true for all k < n, it will be true for k = n/2
- ✔ Hence, **Eqn ii) becomes.**
- ✔ $T(n/2) \leq c.\ (n/2)^3$
- ✔ Or, $T(n/2) \leq c.\ n^3/8$

Substitution Method

Now,

- **$T(n) = 4T(n/2) + n$**
- $T(n) \leq 4 * c * n^3/8 + n$
- $T(n) \leq c * n^3/2 + n$
- $T(n) \leq c * n^3 - c * n^3/2 + n$
- $T(n) \leq c * n^3 - n(c * n^2/2 – 1)$
- Thus, $T(n) \leq cn^3$ for all $n > 0$
- Hence, **$T(n) = O(n^3)$**

Substitution Method

Example 2

Show that the complexity if the following RR is $O(n^2)$ using substitution method.

$T(n) = 4T(n/2) + n$ for $n > 1$

$T(n) = 1$ for $n = 1$

- ✔ Our guess is $T(n) = O(n^2)$
- ✔ From the definition of Big O, $T(n) \leq c*n^2$ for all $n>n_0$ **……. Eqn i)**
- ✔ Now we show that Eqn i) is true using mathematical induction.

Substitution Method

Basic Step:

For n = 1

- ✔ $T(1) \leq c.\ 1^2$
- ✔ $1 \leq c$ which is true for all positive values of c.

Inductive Step:

- ✔ Let us assume that it is true for all k < n.
- ✔ Then, $T(k) \leq c.\ k^2$ **……. Eqn ii)**
- ✔ Since it is true for all k < n, it will be true for k = n/2
- ✔ Hence, **Eqn ii) becomes.**
- ✔ $T(n/2) \leq c.\ (n/2)^2$
- ✔ Or, $T(n/2) \leq c.\ n^2/4$

Substitution Method

Now,

- **$T(n) = 4T(n/2) + n$**
- $T(n) \leq 4 * c * n^2/4 + n$
- $T(n) \leq c * n^2 + n$
- It is not possible to show $c * n^2 + n \leq cn^2$ for all $n > 0$
- Now, we try to subtract lower order terms

Substitution Method

Since, $T(n) = O(n^2)$ we can write

- ✔ $T(n) \leq cn^2 - dn$ for all $n > n_0$[Because $cn^2 - dn \leq cn^2$] **……. Eqn iii)**
- ✔ Now we show that **Eqn iii)** is true using mathematical induction.

Substitution Method

Basic Step:

For n = 1

- ✔ $T(1) \leq c.\ 1^2 - d.1$
- ✔ $1 \leq c - d$ which is true for all positive values of c and d<c.

Inductive Step:

- ✔ Let us assume that it is true for all k < n.
- ✔ Then, $T(k) \leq c.\ k^2 - d.k$ **……. Eqn iv)**
- ✔ Since it is true for all k < n, it will be true for k = n/2
- ✔ Hence, **Eqn iv) becomes.**
- ✔ $T(n/2) \leq c.\ (n/2)^2 - d.n/2$
- ✔ Or, $T(n/2) \leq c.\ n^2/4 - d.n/2$

Substitution Method

Now,

- **$T(n) = 4T(n/2) + n$**
- $T(n) \leq 4 [c * n^2/4 – d*n/2] + n$
- $T(n) \leq c * n^2 - 2*d*n + n$
- $T(n) \leq c * n^2 - d*n - d*n + n$
- $T(n) \leq (c * n^2 - d*n) – n(d - 1)$
- Thus, $T(n) \leq c.n^2 - d.n$ for all $n > 0$
- Hence, **$T(n) = O(n^2)$**

Substitution Method

Example 3

Show that the complexity if the following RR is $O(n^3)$ using substitution method.

$T(n) = 8T(n/2) + n^2$ for $n > 1$

$T(n) = 1$ for $n = 1$

- ✔ Our guess is $T(n) = O(n^3)$
- ✔ From the definition of Big O, $T(n) \leq cn^3$ for all $n>n_0$ **……. Eqn i)**
- ✔ Now we show that **Eqn i)** is true using mathematical induction.

Substitution Method

Basic Step:

For n = 1

- ✔ $T(1) \leq c.\ 1^3$
- ✔ $1 \leq c$ which is true for all positive values of c.

Inductive Step:

- ✔ Let us assume that it is true for all k < n.
- ✔ Then, $T(k) \leq c.\ k^3$ **……. Eqn ii)**
- ✔ Since it is true for all k < n, it will be true for k = n/2
- ✔ Hence, **Eqn ii) becomes.**
- ✔ $T(n/2) \leq c.\ (n/2)^3$
- ✔ Or, $T(n/2) \leq c.\ n^3/8$

Substitution Method

Now,

- **$T(n) = 8T(n/2) + n^2$**
- $T(n) \leq 8 * c * n^3/8 + n^2$
- $T(n) \leq c * n^3 + n^2$
- It is not possible to show $c * n^2 + n^2 \leq cn^3$ for all $n > 0$
- Now, we try to subtract lower order terms

Substitution Method

Since, $T(n) = O(n^3)$ we can write

- ✔ $T(n) \leq cn^3 - dn^2$ for all $n>n_0$[Because $cn^3 - dn^2 \leq cn^3$] **……. Eqn iii)**
- ✔ Now we show that **Eqn iii)** is true using mathematical induction.

Substitution Method

Basic Step:

For n = 1

- ✔ $T(1) \leq c.\ 1^3 - d.1^2$
- ✔ $1 \leq c - d$ which is true for all positive values of c and d<c.

Inductive Step:

- ✔ Let us assume that it is true for all k < n.
- ✔ Then, $T(k) \leq c.\ k^3 - d.k^2$ **……. Eqn iv)**
- ✔ Since it is true for all k < n, it will be true for k = n/2
- ✔ Hence, **Eqn iv) becomes.**
- ✔ $T(n/2) \leq c.\ (n/2)^3 - d.(n/2)^2$
- ✔ Or, $T(n/2) \leq c.\ n^3/8 - d.n^2/4$

Substitution Method

Now,

- **$T(n) = 8T(n/2) + n^2$**
- $T(n) \leq 8 [c * n^3/8 – d*n^2/4] + n^2$
- $T(n) \leq c * n^3 - 2*d*n^2 + n^2$
- $T(n) \leq c * n^3 - d*n^2 - d*n^2 + n$
- $T(n) \leq (c * n^3 - d*n^2) – n(d*n- 1)$
- Thus, $T(n) \leq c.n^3 - d.n^2 \leq cn^3$ for all $n > 0$
- Hence, **$T(n) = O(n^3)$**

Master's Method

- The master method is a formula for solving recurrence relations of the form

$$T(n) = aT(n/b) + f(n),$$

where,

– n = size of input

– a = number of sub-problems in the recursion

– n/b = size of each sub-problem All sub-problems are assumed to have the same size.

– f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

- Here, $a \geq 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function.

Master’s Method

- An asymptotically positive function means that for a sufficiently large value of n, we have f(n) > 0.
- The master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.

Master's Method

- If $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function, then the time complexity of a recursive relation is given by $T(n) = aT(n/b) + f(n)$ where, T(n) has the following asymptotic bounds

  1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

  2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(f(n) * \log n)$

  3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.

- $\epsilon > 0$ is a constant.

Master's Method

Example 1

Solve the following RR using Master's method

$$T(n) = 3\,T(n/2) + n$$

- Comparing with **$T(n) = aT(n/b) + f(n)$,** $a = 3$, $b = 2$ and $f(n) = n$
- Now $n^{\log_b a} = n^{\log_2 3} = n^{\log_{10} 3 / \log_{10} 2} = \mathbf{n^{1.584}}$
- Since, $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.[Choose $\epsilon = 0.1$]
- $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) = \Theta(\mathbf{n^{1.584}})$

Master's Method

Example 2

Solve the following RR using Master's method

$$\mathbf{T(n) = 4\,T(n/2) + n^2}$$

- Comparing with **T(n) = aT(n/b) + f(n),** a = 4, b = 2 and f(n) = n^2
- Now $n^{\log_b a} = n^{\log_2 4} = \mathbf{n^2}$
- Since, f(n) = $\Theta(n^{\log_b a})$, then T(n) = $\Theta(n^2 * \mathbf{log}\ n)$.[Choose $\epsilon = 0.1$]
- T(n) = $\Theta(n^2 * \mathbf{log}\ n)$

Master's Method

Example 3

Solve the following RR using Master's method

$$\mathbf{T(n) = 9\,T(n/3) + n}$$

- Comparing with **T(n) = aT(n/b) + f(n),** a = 9, b = 3 and f(n) = n^2
- Now $n^{\log_b a} = n^{\log_3 9} = \mathbf{n^2}$
- Since, f(n) = $O(n^{\log_b a - \epsilon})$, then T(n) = $\Theta(n^{\log_b a})$.[Choose $\epsilon = 0.1$]
- T(n) = $\Theta(n^{\log_b a}) = \Theta(n^2)$

Master's Method

Example 4

Solve the following RR using Master's method

$$\mathbf{T(n) = 3\,T(n/4) + n \log n}$$

- Comparing with **$T(n) = aT(n/b) + f(n)$,** $a = 3$, $b = 4$ and $f(n) = n \log n$
- Now $n^{\log_b a} = n^{\log_4 3} = \mathbf{n^{0.658}}$
- Since, $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.[Choose $\epsilon = 0.1$]
- $T(n) = \Theta(f(n)) = \Theta(n \log n)$

Master's Method

Example 5

Solve the following RR using Master's method

$$\mathbf{T(n) = 2\,T(n/4) + n^{0.5}}$$

- Comparing with **T(n) = aT(n/b) + f(n),** a = 2, b = 4 and f(n) = $n^{0.5}$
- Now $n^{\log_b a} = n^{\log_4 2} = \mathbf{n^{0.5}}$
- Since, $f(n) = \Theta(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n)*\mathbf{log}\ n)$.[Choose ϵ = 0.1]
- $T(n) = \Theta(f(n)*\mathbf{log}\ n) = \Theta(n^{0.5}\ \mathbf{log}\ n)$

Master's Method

Example 6

Solve the following RR using Master's method

$$\mathbf{T(n) = 2\,T(2n/3) + 1}$$

- Comparing with **$T(n) = aT(n/b) + f(n)$,** $a = 2$, $b = 1.5$ and $f(n) = 1$
- Now $n^{\log_b a} = n^{\log_{1.5} 2} = \mathbf{n^{1.709}}$
- Since, $f(n) = O(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.[Choose $\epsilon = 0.1$]
- $T(n) = \Theta(n^{\log_b a}) = \Theta(\mathbf{n^{1.709}})$

Substitution Method

Example 7

Solve the following RR using Master's method

$$\mathbf{T(n) = 8T(n/2) + n^2 \text{ for } n > 1}$$

- Comparing with **T(n) = aT(n/b) + f(n),** a = 8, b = 2 and f(n) = n^2
- Now $n^{\log_b a} = n^{\log_2 8} = n^3$
- Since $f(n) = O(n \log_b{}^{a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
- $T(n) = \Theta(n^3)$

Master's Method

Inadmissible Equations

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

  a is not a constant; the number of subproblems should be fixed

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

  non-polynomial difference between $f(n)$ and $n^{\log_b a}$ (see below; extended version applies)

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

  $a < 1$ cannot have less than one sub problem

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

  $f(n)$, which is the combination time, is not positive

- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

  case 3 but regularity violation.

Divide-and-Conquer Approach

- A **divide-and-conquer algorithm** recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.

Divide-and-Conquer Approach

Example

```
fibo(n)
{
    if (n == 1 || n==2)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
```

Divide-and-Conquer Approach

- Designing efficient divide-and-conquer algorithms can be difficult.
- The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Divide-and-Conquer Approach

Advantages

- **Solving difficult problem**

  – Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.

- **Algorithm efficiency**

  – The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to the quick-sort and merge-sort algorithms.

- **Parallelism**

  – Divide-and-conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

- **Memory access**

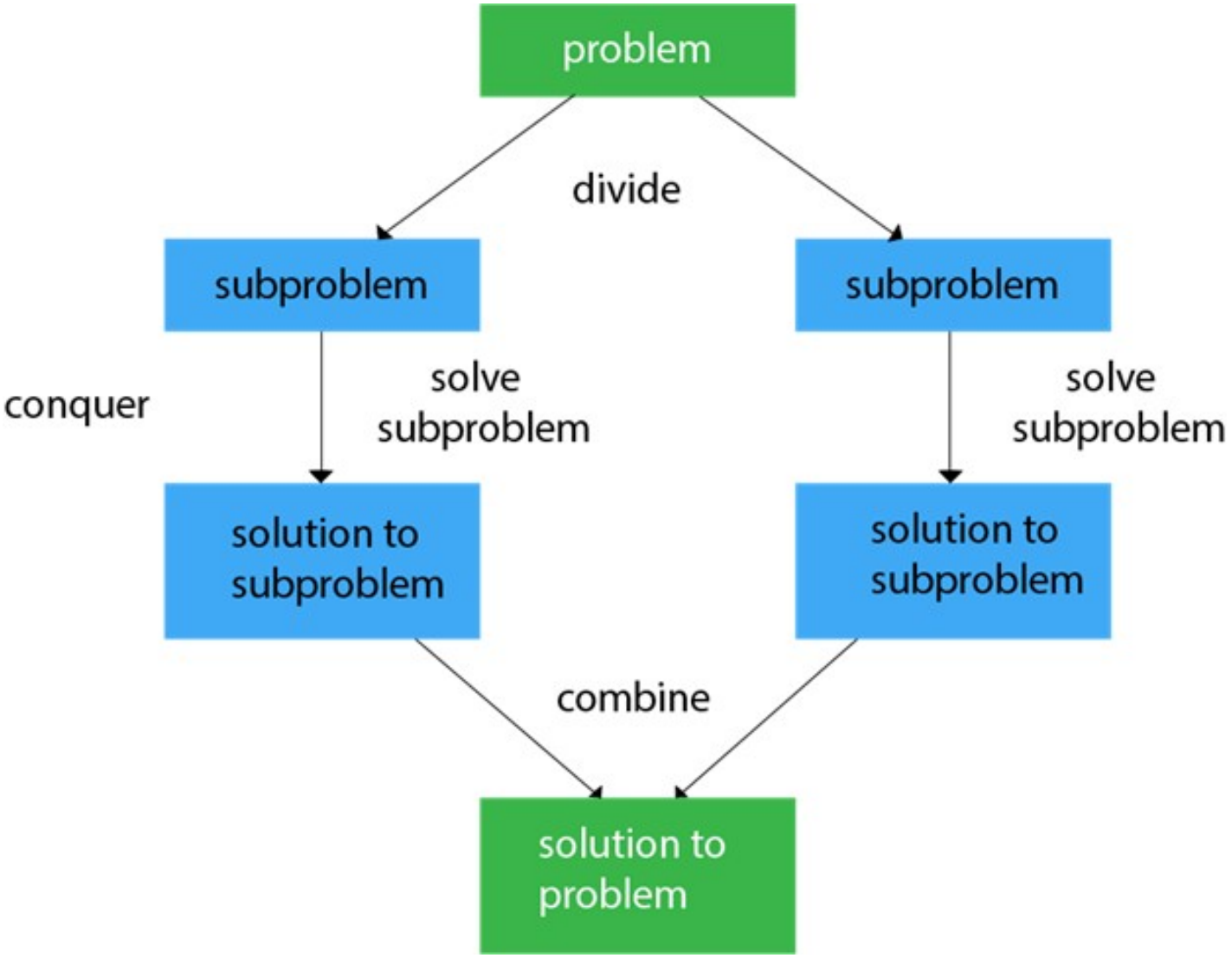
  – Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory.

Divide-and-Conquer Approach

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

Divide-and-Conquer Approach



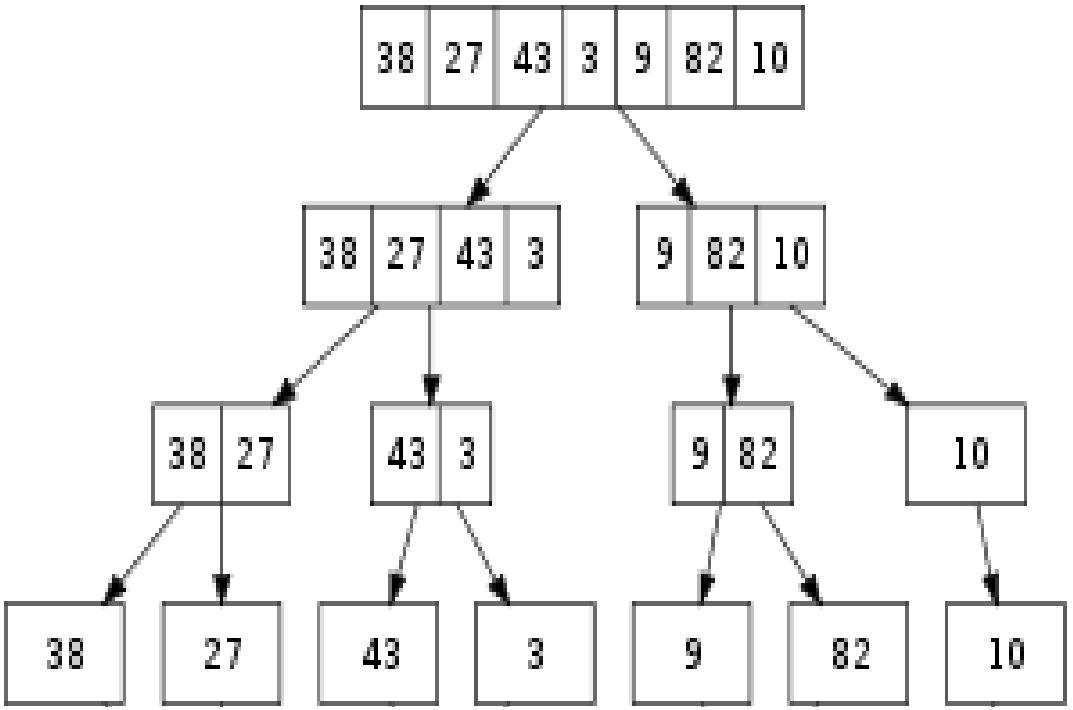
Sorting Algorithms

- Arrangement of data in some systematic order is called sorting
- We will discuss some recursive sorting algorithms.
  - Merge Sort
  - Quick Sort
  - Heap Sort

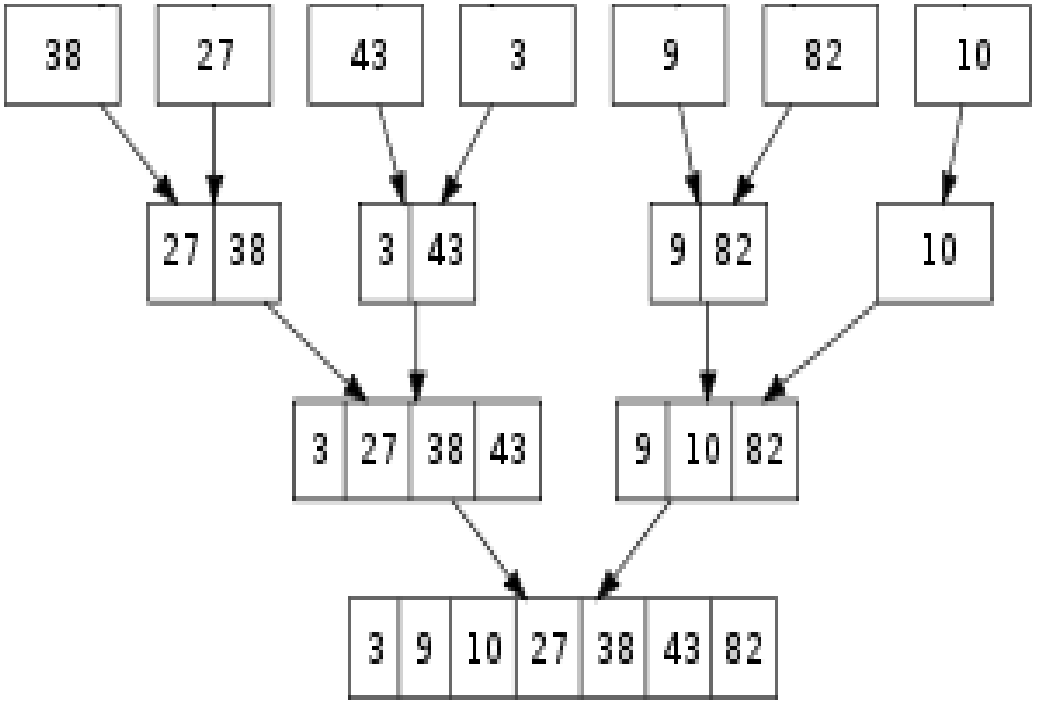
Merge Sort

- It is a divide and conquer algorithm.
- At first we divide the given list of item.
  - List is divided into two parts from middle.
  - The process is repeated until each sub-list contain exactly 1 item.
- Now is the turn for sort and combine (conquer)
  - A list with a single element is considered sorted automatically.
  - Pair of list is sorted and merged into one (i.e. approx. n/2 sublists of size 2).
  - The sort and merge is keep on repeated until a single list of size n is found.
- The overall dividing and conquering is done recursively.

Merge Sort



Divide



Conquer

Merge Sort

To sort A[l.. r]

1. **Divide Step**
   - If a given array A has zero or one element, simply return; it is already sorted.
   - Otherwise, split **A[l..r]** into two sub-arrays **A[l..m]** and **A[m+1..r]**, each containing about half of the elements of **A[l .. r]**. That is, m is the halfway point of **A[l..r]**
2. **Conquer Step**
   - Conquer by recursively sorting the two sub-arrays **A[l..m]** and **A[m+1..r]**
3. **Combine Step**
   - Combine the elements back in A[l..m] by merging the two sorted sub-arrays **A[l..m]** and **A[m+1..r]** into a sorted sequence.
   - To accomplish this step, we will define a procedure MERGE (***A***, ***l***, ***m***, ***r***).

Merge Sort

Declare and initialize necessary variables

n-total number of elements in an array

a[n]-array containing data

p=0, r=n-1; first and last index of the array

MERGE-SORT (*A*, *l*, *r*)

1. IF *l* < *r* // Check for base case
2. THEN *m* = (*p* + *r*)/2 // Divide step
3. MERGE-SORT (A, *l*, *m*) // Conquer step.
4. MERGE-SORT (A, m+ 1, *r*) // Conquer step.
5. MERGE (A, *l*, *m*,*r*) // Conquer step.

Merge Sort

Pseudocode

```
merge_sort(A, l, r)
{
    if(l<r)
    {
        m=(l+r)/2;
        merge_sort(a,l,m);
        merge_sort(a,m+1,r);
        merge(a,l,m+1,r);
    }
}
```

```
merge(A, l, m, r)
{
    x=l;
    k=l;
    y=m;
    while(x<m && y<=r)
    {
        if (a[x]<a[y])
            b[k++]=a[x++];
        else
            b[k++]=a[y++];
    }
    for(;x<m;x++,k++)
        b[k]=a[x];
    for(;y<=r;y++,k++)
        b[k]=a[y];
    for(i=l;i<=r;i++)
        a[i]=b[i];
```

Merge Sort

Analysis

- Since, there are two recursive sub-problems of size n/2, the problem is divided into two equal halves and there is also a need to merge the solutions.
- Size of sub-problems = n/2
- Dividing and merging takes O(n)
- The recurrence relation can be written as

  $T(n) = 2T(n/2) + O(n)$, if $n>1$

  $T(n) = 1$, if $n = 2$
- Solving the recurrence relation, we can get $T(n) = O(n \log_2 n)$

Quick Sort

- Also called partition-exchange sort.
- Uses divide and conquer concept.
- One pivot element is chosen from within the list.
- The list is divided into two partition.
  - All values less than the pivot are placed on left side of pivot.
  - All greater values are placed on right side of the pivot.
- After a single pass, the pivot is in its proper position.
- The left and right partitions are sorted recursively using the same method.
- Joining the left sorted, pivot and right sorted results with the list in sorted order.

Quick Sort

- Here is the three-step divide-and-conquer process for sorting a typical array **A[l..r]**

1) **Divide:**

   – Partition (rearrange) the array **A[l..r]** into two (possibly empty) sub-arrays **A[l..pivot-1]** and **A[pivot+1..r]** such that each element of **A[l..pivot-1]** is less than or equal to **A[pivot]**, which is, in turn, less than or equal to each element of **A[pivot+1..r]**.

   – Compute the index pivot as part of this partitioning procedure.

2) **Conquer:** Sort the two sub-arrays **A[l..pivot-1]** and **A[pivot+1..r]** by recursive calls to Quick sort.

3) **Combine:** Because the sub-arrays are already sorted, no work is needed to combine them: the entire array A[l..r] is now sorted.

Quick Sort

The following procedure implements quick sort:

```
QUICKSORT(A, l, r)
if (l < r)
{
    pivot =  PARTITION(A, l, r)
    QUICKSORT(A, l, pivot-1)
    QUICKSORT(A, pivot+1, r)
}
```

To sort an entire array A, the initial call is QUICKSORT(A, 0, r).

Quick Sort

Partitioning the array

```
partition(A,l,r)
{
   x=l;
   y=r;
   p=A[l];
while(x<y)
   {
      while(A[x]<=p)
         x++;
      while(A[y]>p)
         y--;
      if(x<y)
      {
         swap(A[x],A[y]);
      }
      A[l]=A[y];
      A[y]=p;
      return y;
   }
}
```

Quick Sort

Best Case Analysis

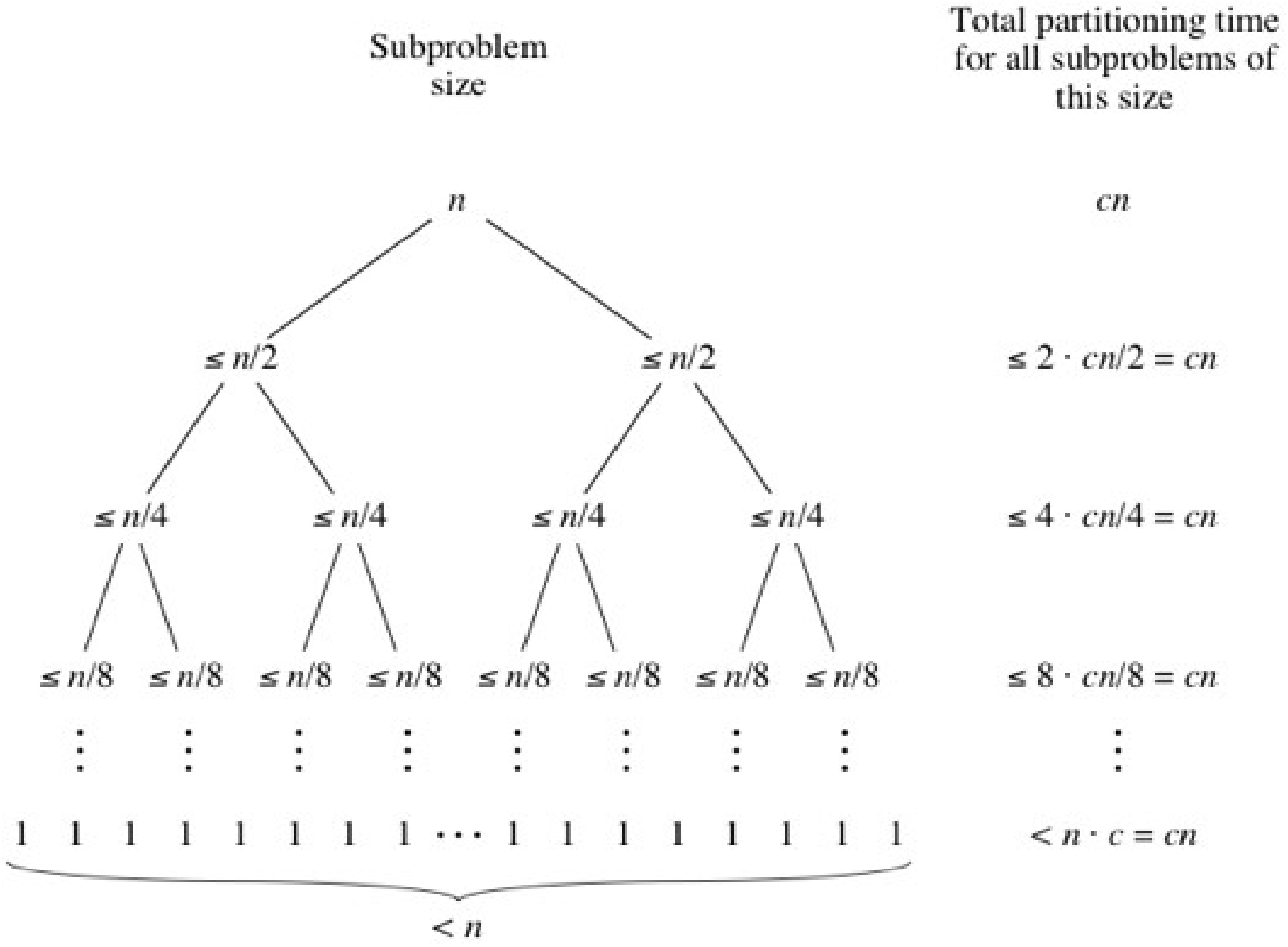
- This algorithms works the best when the elements are divided into two equal partitions.
- Thus the recurrence relation is

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

- On solving it, we get

  $T(n) = O(n \log_2 n)$

Quick Sort


Subproblem size
Total partitioning time for all subproblems of this size
n
cn
≤ n/2
≤ n/2
≤ 2 · cn/2 = cn
≤ n/4
≤ n/4
≤ n/4
≤ n/4
≤ 4 · cn/4 = cn
≤ n/8
≤ n/8
≤ n/8
≤ n/8
≤ n/8
≤ n/8
≤ n/8
≤ n/8
≤ 8 · cn/8 = cn
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
< n · c = cn
< n

Quick Sort

Worst Case Analysis

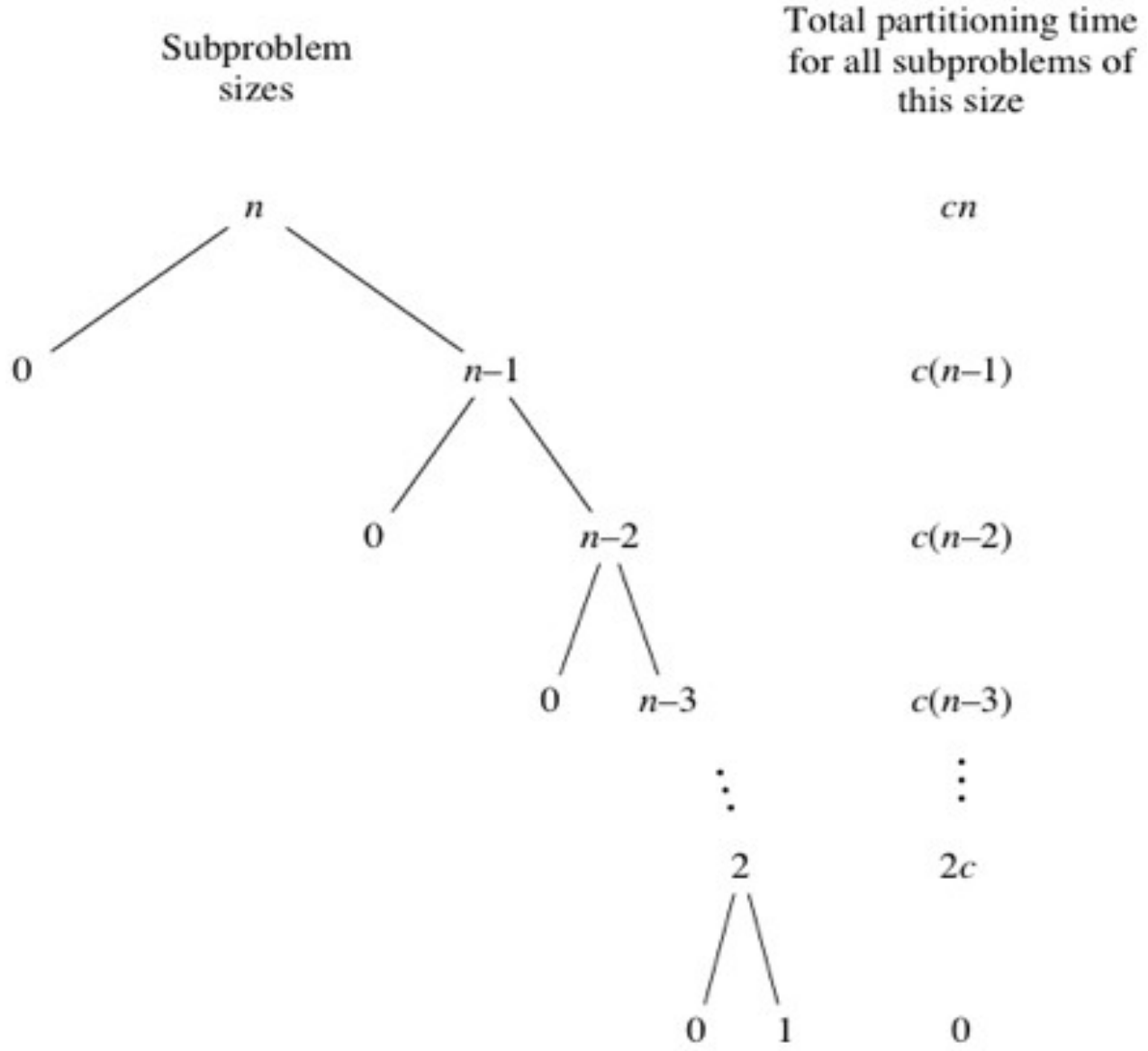
- This algorithms works in the worst way when the elements are already sorted.
- The worst-case behavior for Quick Sort occurs when the partitioning routine produces one sub-problem with n-1 elements and one with 0 elements.
- We assume that this unbalanced partitioning arises in each recursive call. The partitioning costs O(n) time.
- Since the recursive call on an array of size 0 just returns, T (0) = 1 and the recurrence for the running time is
- Thus the recurrence relation is

  $T(n) = T(n-1) + T(0) + O(n)$

  $T(n) = T(n-1) + O(n)$
- On solving it, we get

  $T(n) = O(n^2)$

Quick Sort


Subproblem sizes
Total partitioning time for all subproblems of this size
n
cn
0
n–1
c(n–1)
0
n–2
c(n–2)
0
n–3
c(n–3)
2
2c
0
1
0

Quick Sort

Average Case Analysis

Self Study

Heaps

- It is an **almost complete binary tree** whose elements have keys that satisfy the following heap property:

  – the value of each node is less than or equal to the value in the parent node.(**MAX Heaps**)

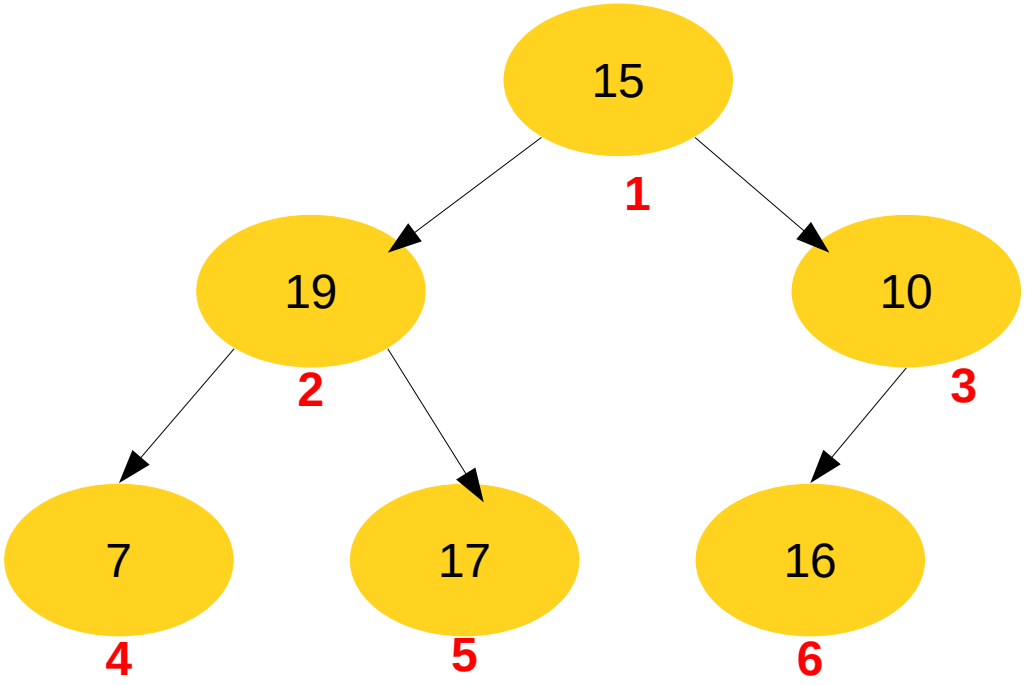
  – the value of each node is greater than or equal to the value in the parent node.(**MIN Heaps**)

- Heaps can be used to implement priority queue and heap sort algorithm.

Heaps

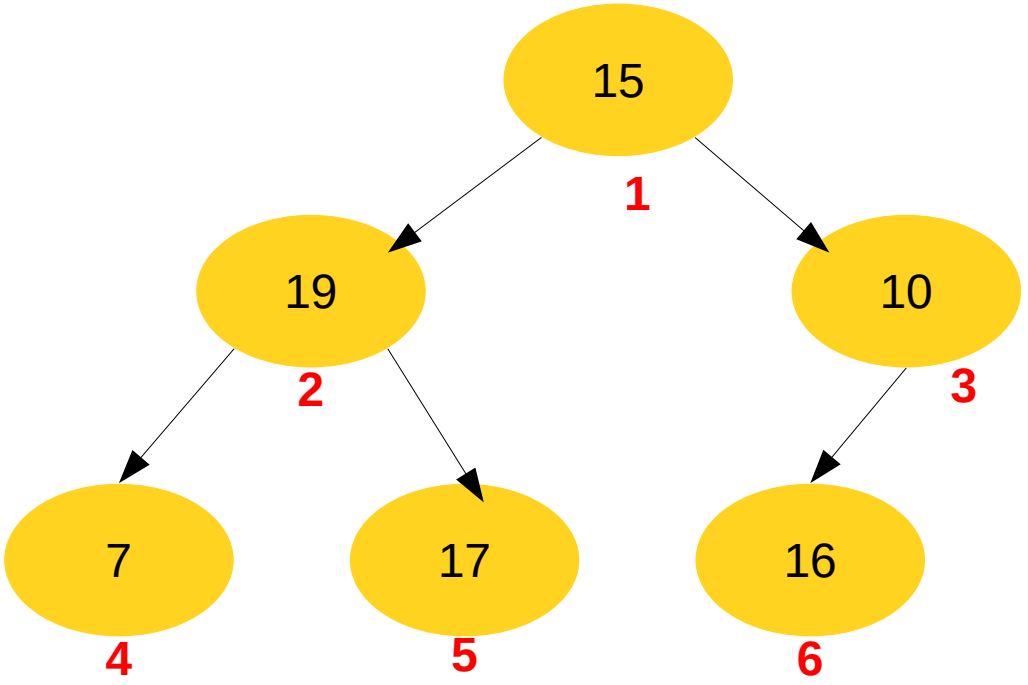
Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

Look at node number 1 to floor(6/2)



Heaps

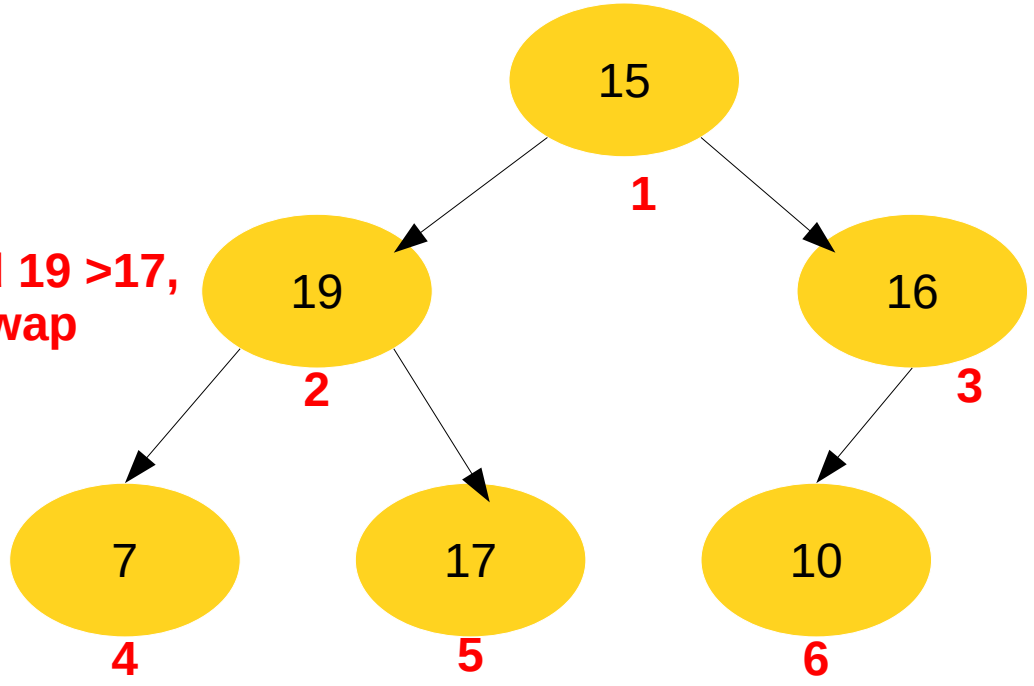
Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}



10>16, so we swap

Heaps

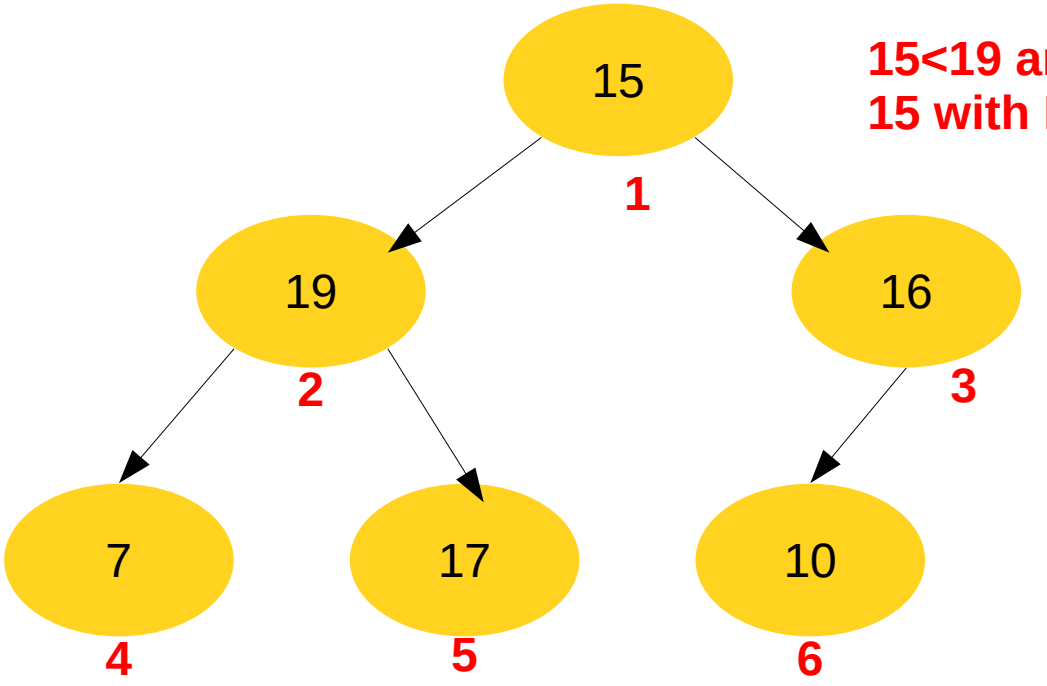
Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}



19>7 and 19 >17,
So NO swap

Heaps

Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

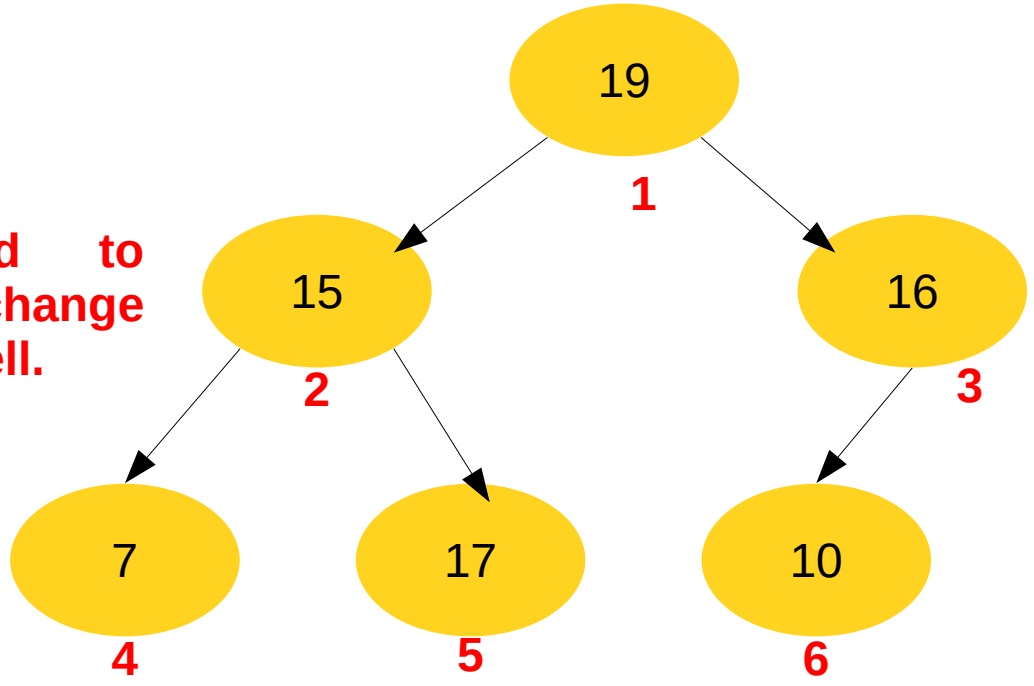


15<19 and 15 <16, we swap
15 with MAX(19,16)

Heaps

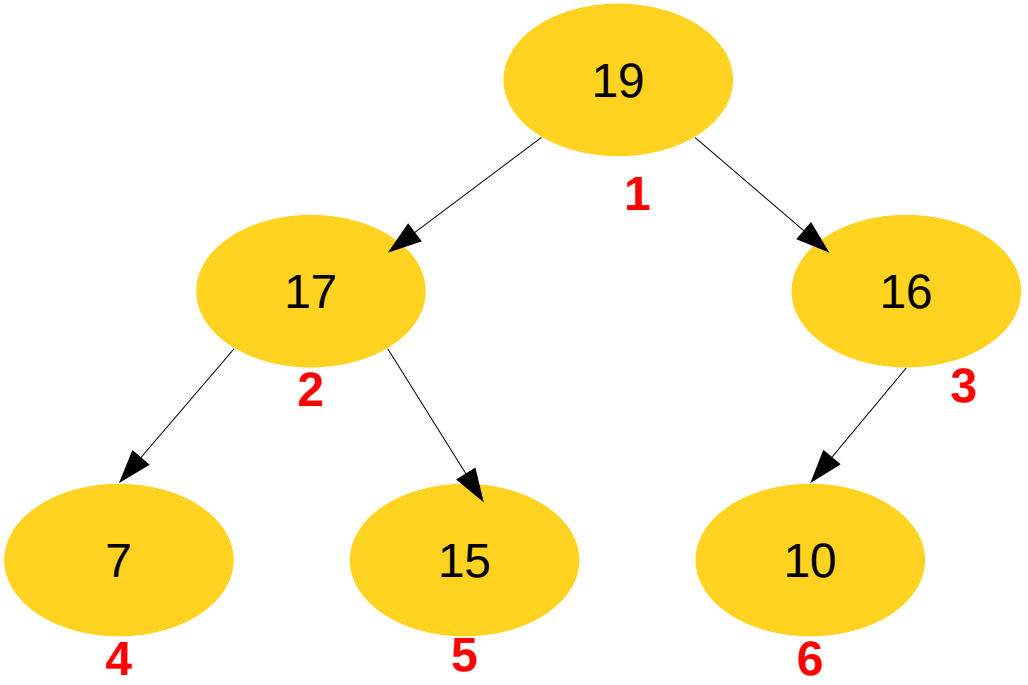
Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

We need to make a change here as well.



Heaps

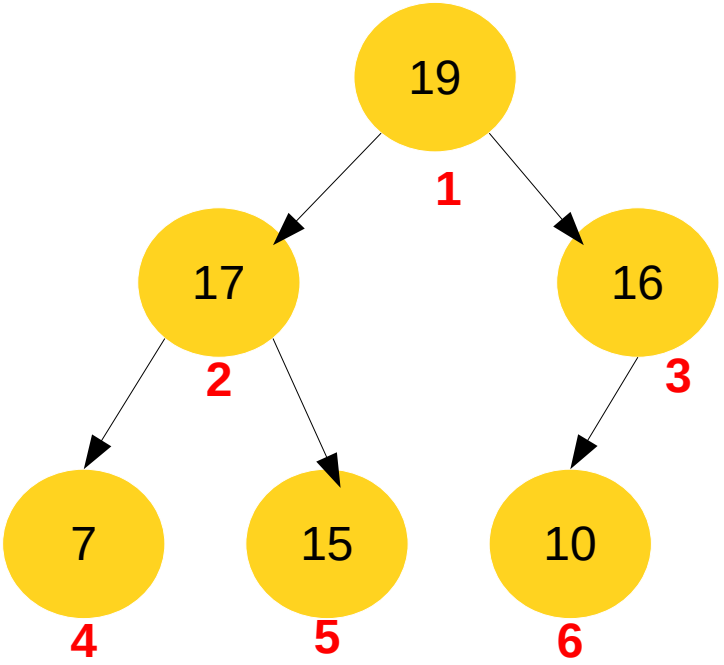
Example : Construct a MAX heap from a set of 6 elements {15, 19, 10, 7, 17, 16}



Heaps - Representation

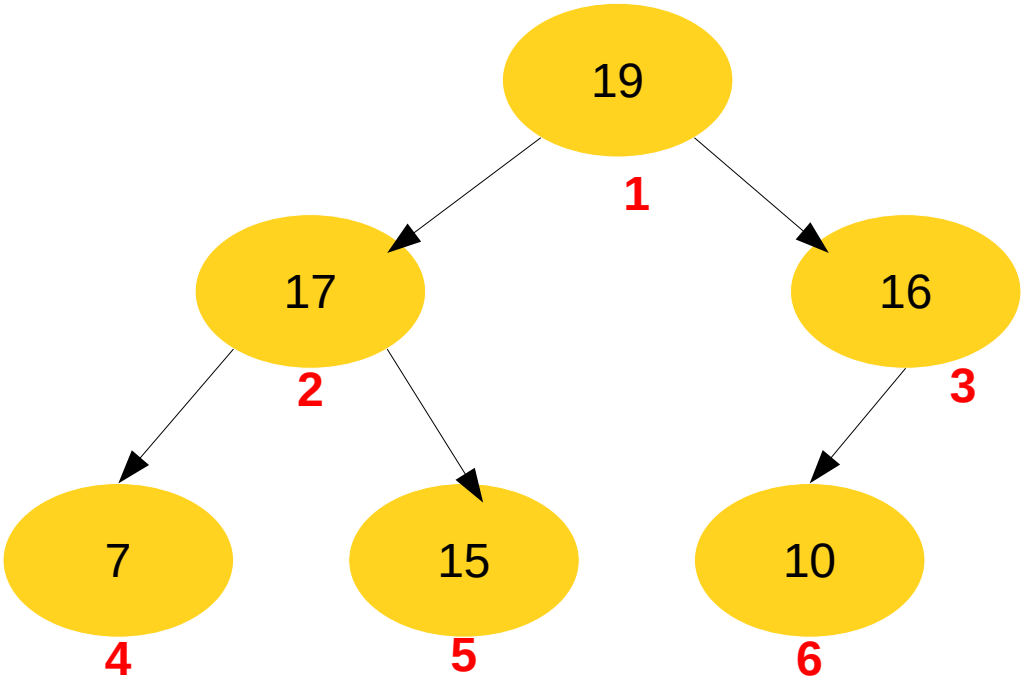
Array Representation of Heaps

- A heap can be stored as an array.
- Root of tree is at **A[1]**.
- Left child of A[i] will be at A[2i]
- Right child of A[i] will be at A[2i+1]
- Parent of A[i] = A[floor(i/2)]
- The elements in the sub-array A[floor(n/2)+1…….n] are leaves.



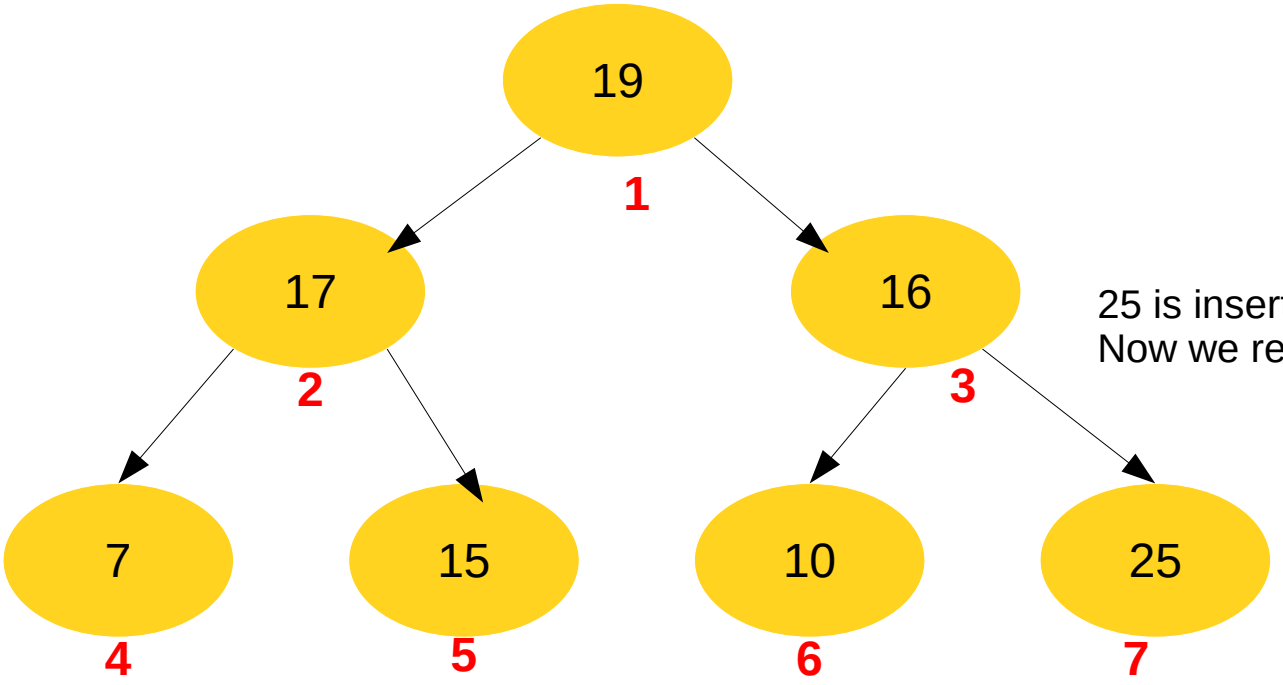
Heaps – Inserting an Element

- Elements are always inserted next to right-most leaf at the bottom level.
- Then, we restore the heap property.
- Suppose we are inserting 25 to the following heap



Heaps – Inserting an Element

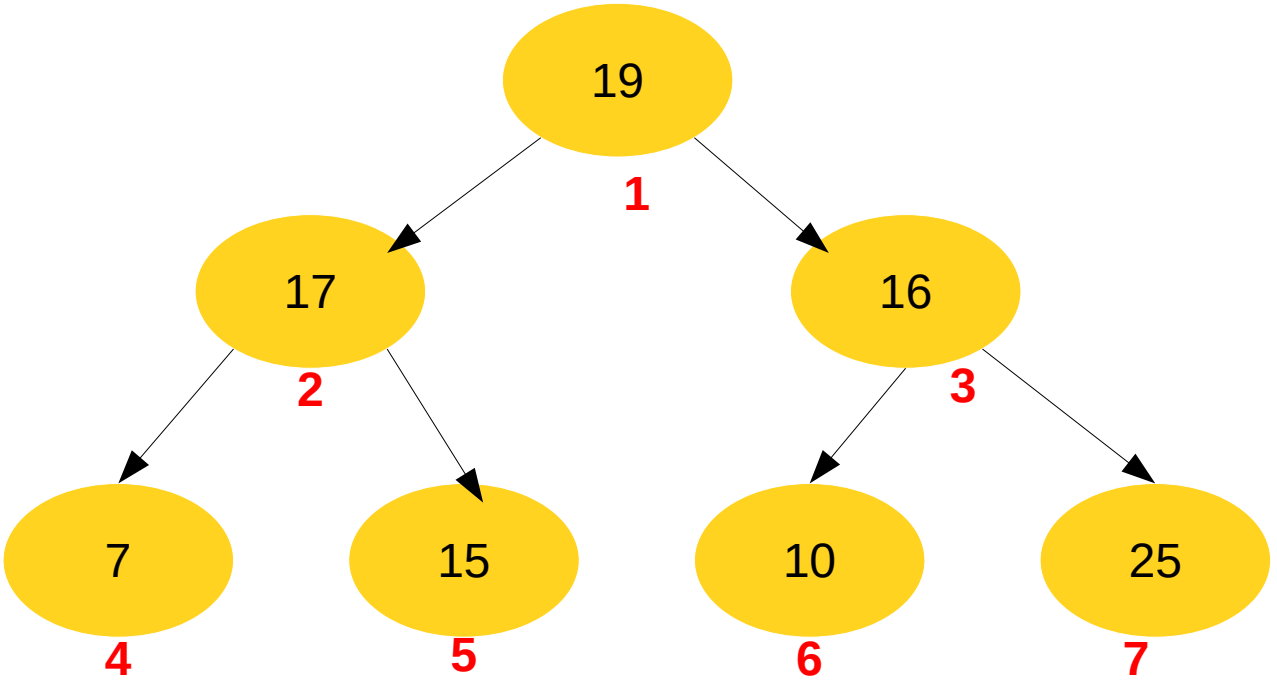
- Elements are always inserted next to right-most leaf at the bottom level.
- Then, we restore the heap property.
- Suppose we are inserting 25 to the following heap



25 is inserted next to right-most leaf,
Now we restore heap property.

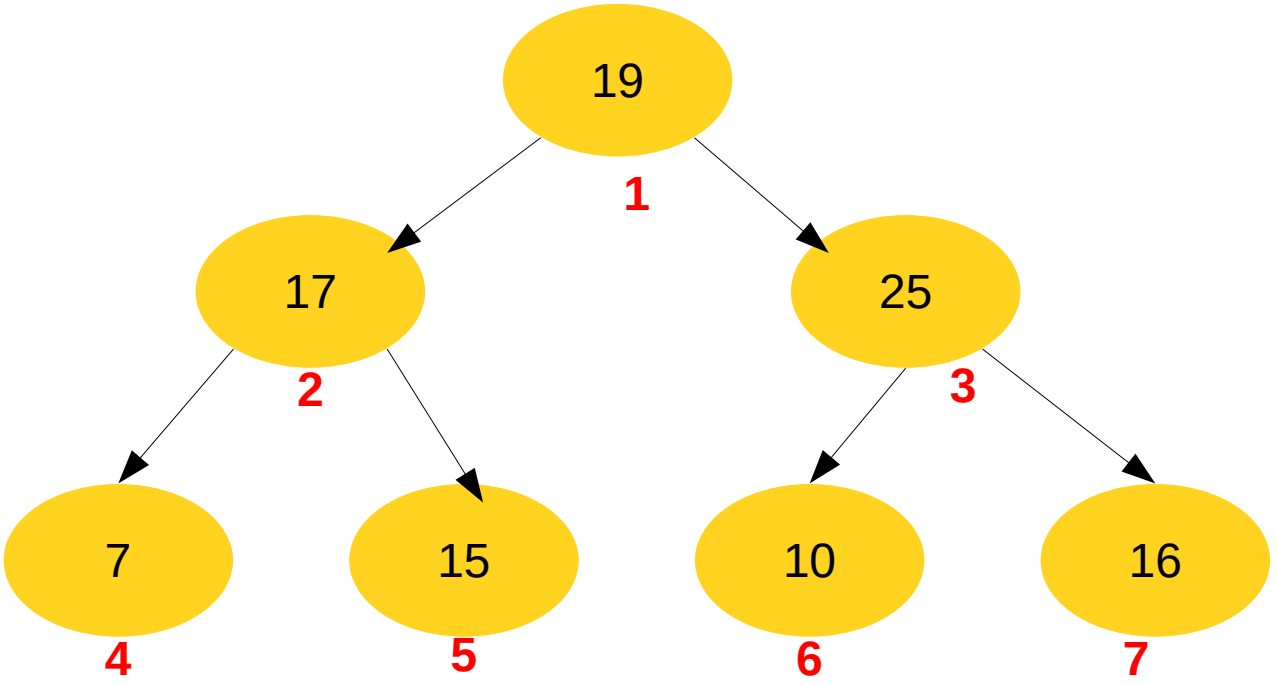
Heaps – Inserting an Element

- Suppose we are inserting 25 to the following heap.
- Now 25 > 16 so we swap.



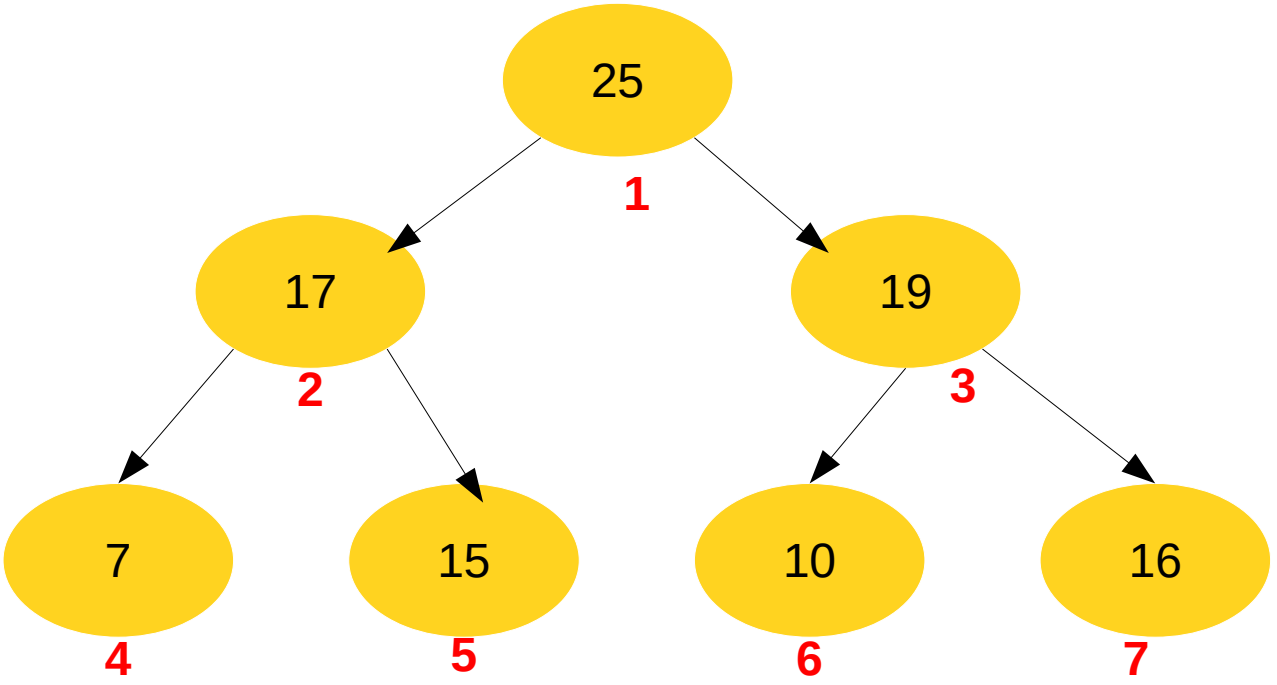
Heaps – Inserting an Element

- Suppose we are inserting 25 to the following heap.
- Now 25 > 19 so we swap.



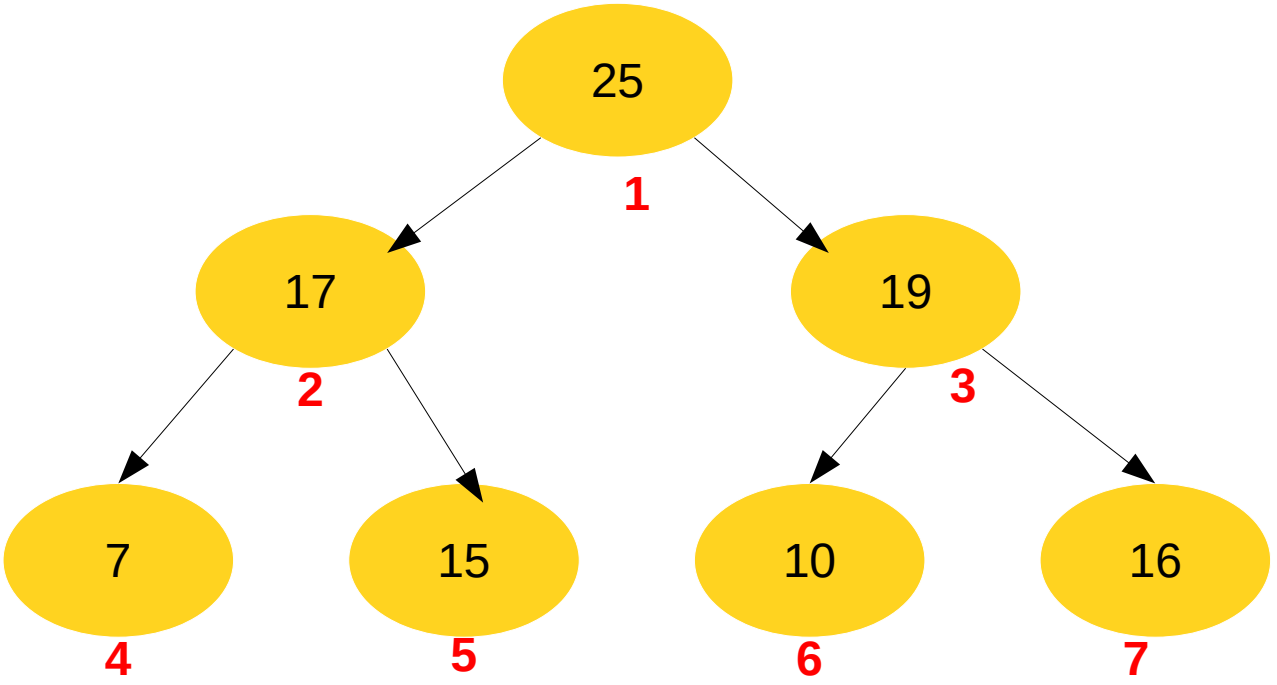
Heaps – Inserting an Element

- Suppose we are inserting 25 to the following heap.
- We finally have a heap.



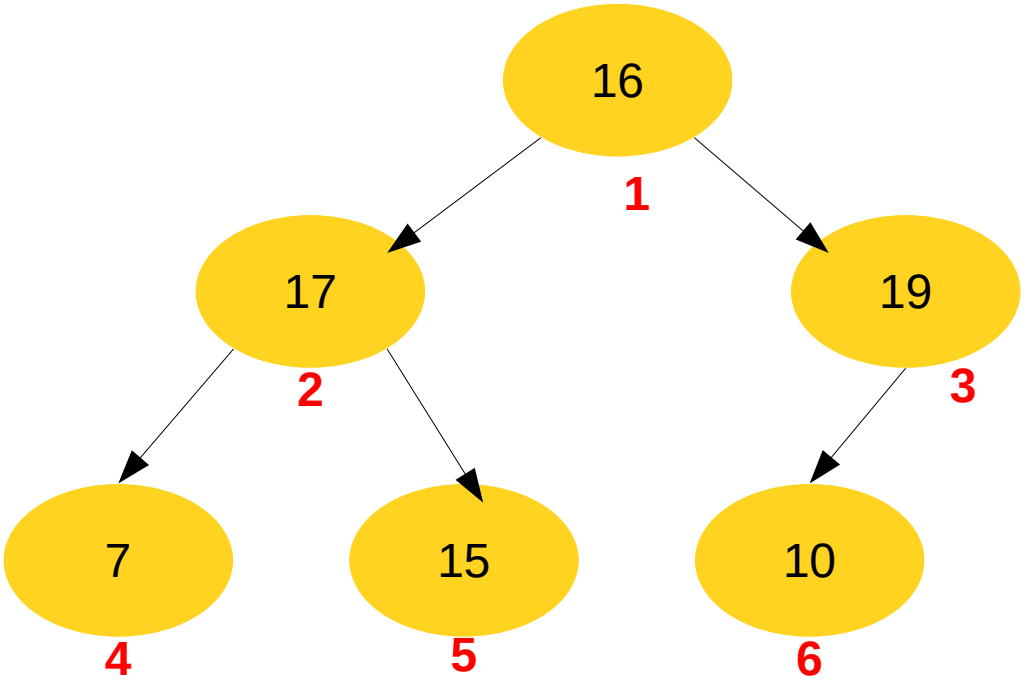
Heaps – Deleting an Element

- We always remove the root element from the heap.
- We move the last element in place of the root element and restore the heap property.
- Suppose we perform a deletion.



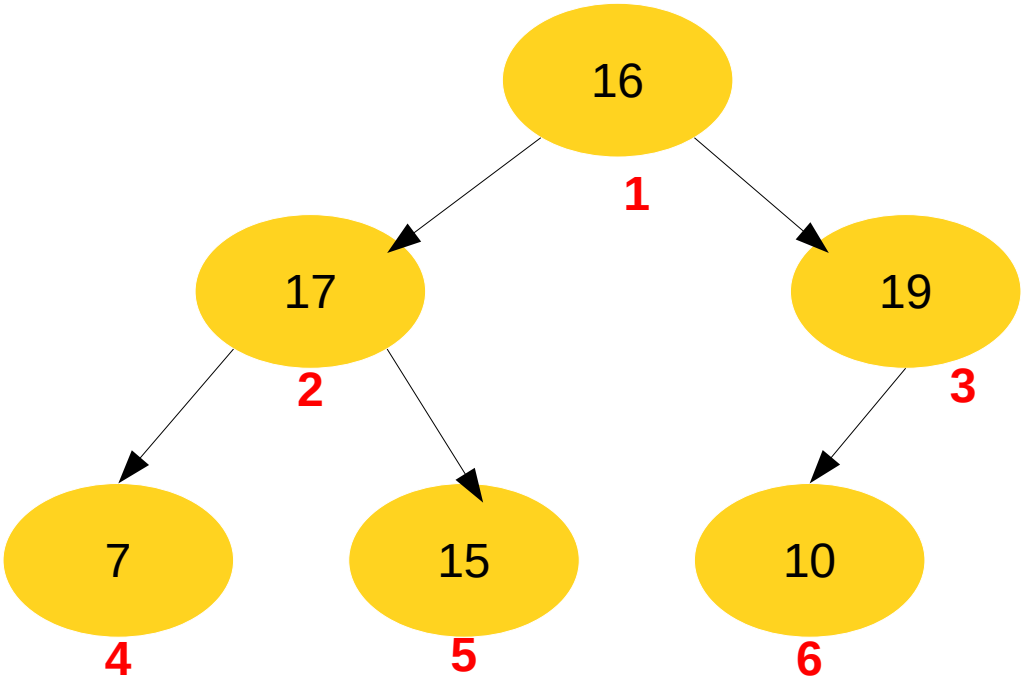
Heaps – Deleting an Element

- We can see that the last element 16 is moved to root and heap size decreases by 1.
- Now we restore heap property.



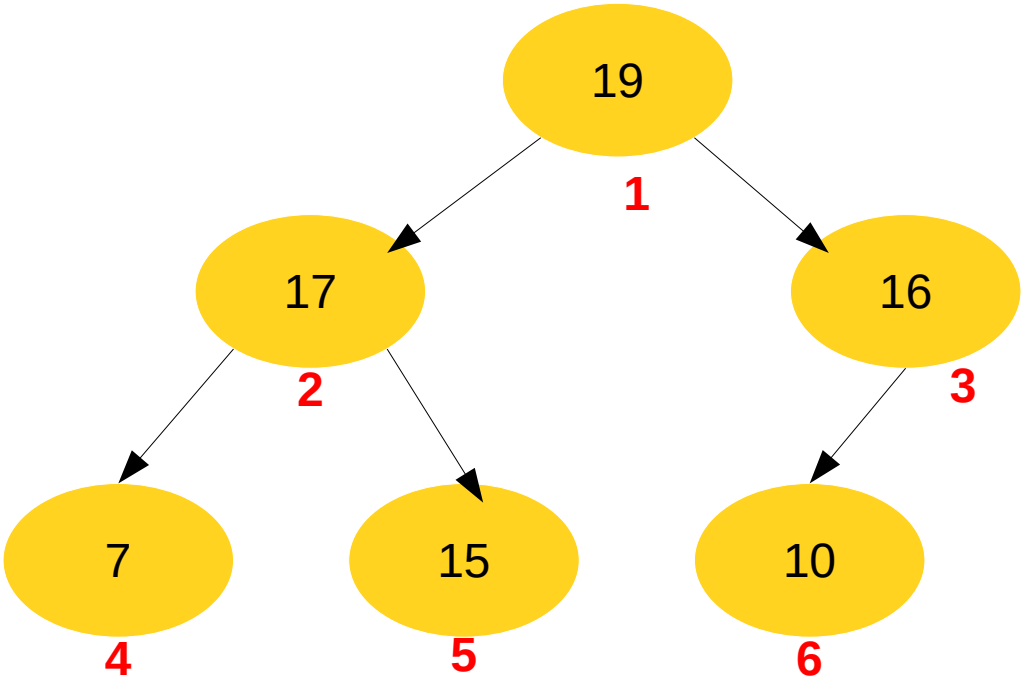
Heaps – Deleting an Element

- 16 < 19 so we swap.



Heaps – Deleting an Element

- Now all nodes are following the heap property.
- So, we successfully deleted an element.



Operations on Heap

1. Maintain/Restore the MAX-HEAP property.
   - **– MAX-HEAPIFY**
2. Create a MAX-HEAP from an array.
   - **– BUILD-MAX-HEAP**
3. Sort an array in place.
   - **– HEAPSORT**

MAX-HEAPIFY

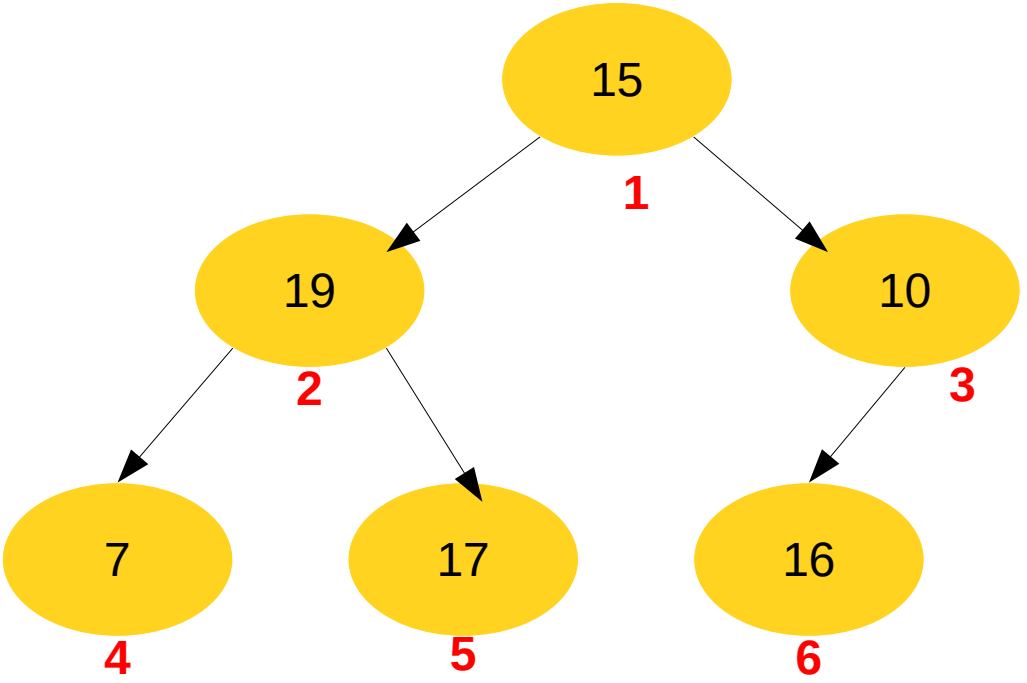
Algorithm

1. Find location of largest value among A[i], A[Left(i)] and A[Right(i)]
2. If the largest value is not A[i], MAX-HEAP property doesn't holds so exchange A[i] with the larger of two children to preserve MAX-HEAP property.
3. Continue this process of compare/exchange down the heap until sub-tree rooted at i is MAX-HEAP.
4. A sub-tree rooted at a leaf node is automatically MAX-HEAP

MAX-HEAPIFY

Construct a binary tree of the following data and then perform MAX-HEAPIFY operations on all the node that violets the heap property.

A[ ] = **{15, 19, 10, 7, 17, 16}**

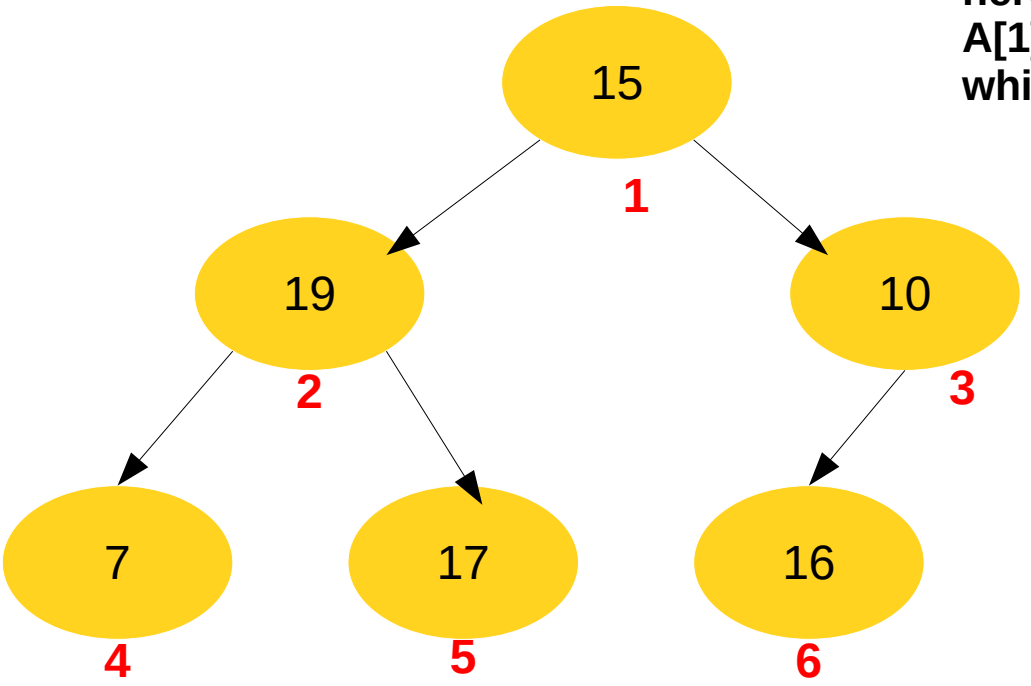


MAX-HEAPIFY

Construct a binary tree of the following data and then perform MAX-HEAPIFY operations on all the node that violets the heap property.

A[ ] = **{15, 19, 10, 7, 17, 16}**

Heap Property is Violated here at A[1], so we swap A[1] with it's larger children which is 19

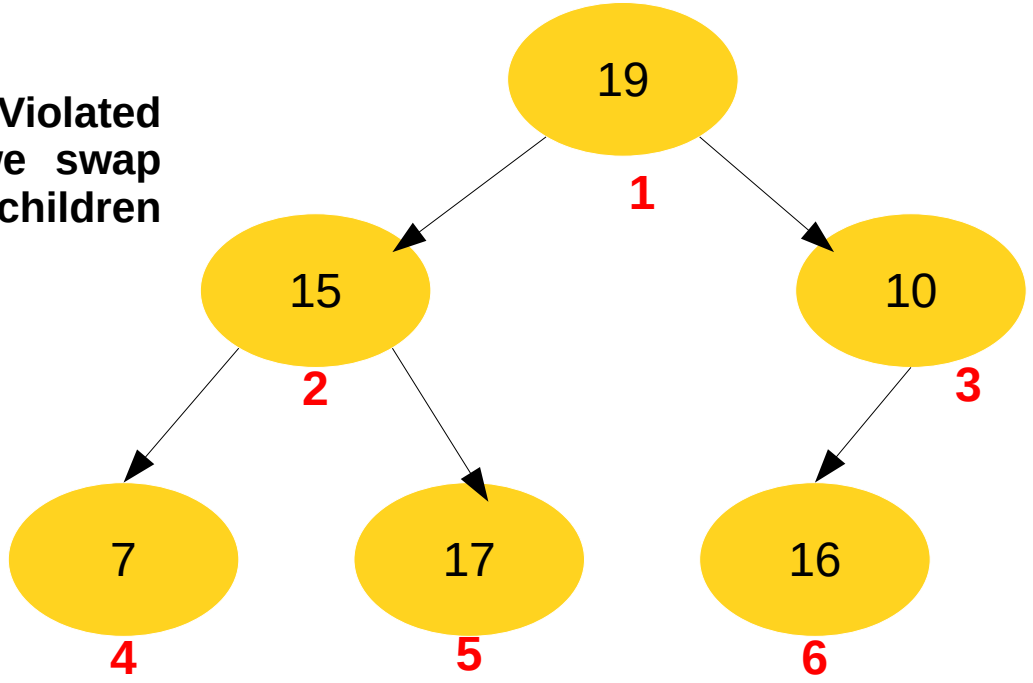


MAX-HEAPIFY

Construct a binary tree of the following data and then perform MAX-HEAPIFY operations on all the node that violets the heap property.

A[ ] = **{15, 19, 10, 7, 17, 16}**

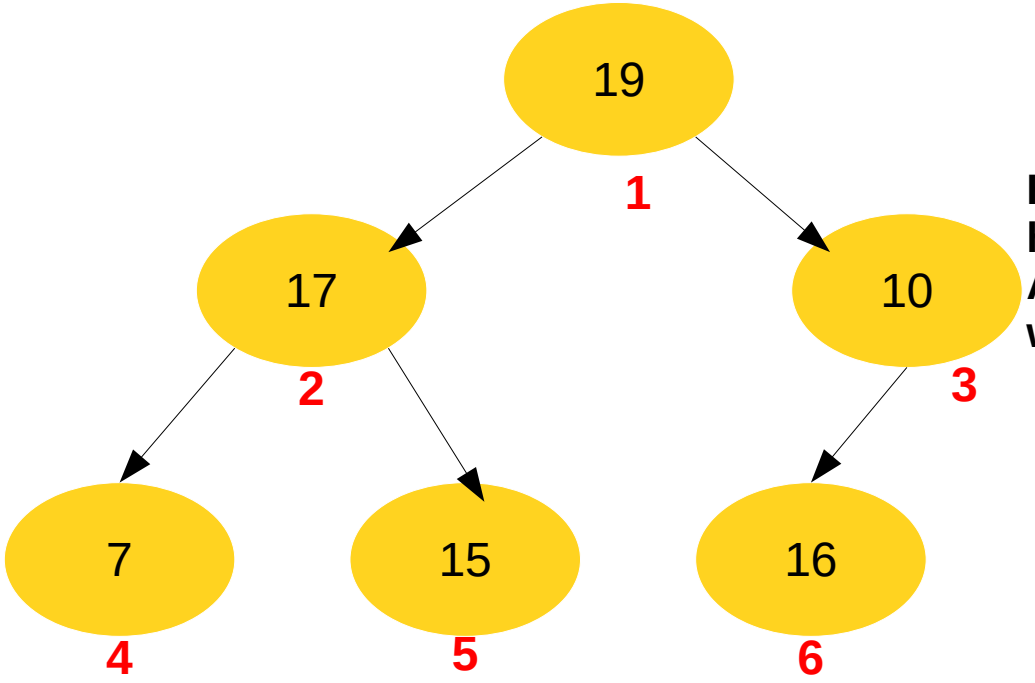
Heap Property is Violated here at A[2], so we swap A[2] with it's larger children which is 17



MAX-HEAPIFY

Construct a binary tree of the following data and then perform MAX-HEAPIFY operations on all the node that violets the heap property.

A[ ] = **{15, 19, 10, 7, 17, 16}**

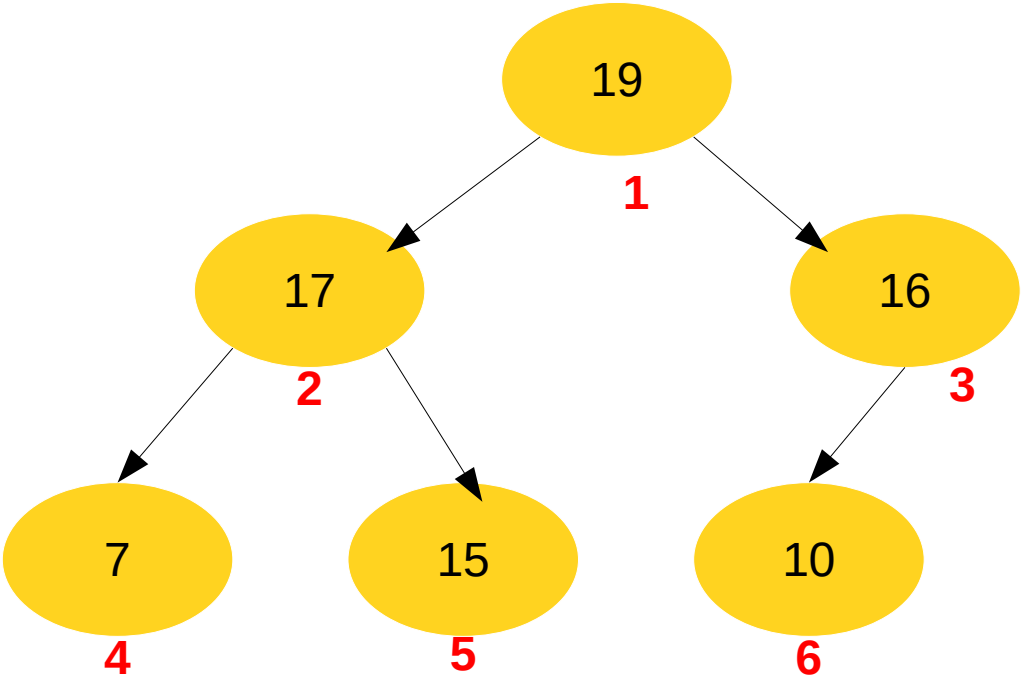


Heap Property is Violated here at A[3], so we swap A[3] with it's larger children which is 16

MAX-HEAPIFY

Construct a binary tree of the following data and then perform MAX-HEAPIFY operations on all the node that violets the heap property.

A[ ] = **{15, 19, 10, 7, 17, 16}**



MAX-HEAPIFY

Pseudocode

```
MAX-HEAPIFY(A, i, n)
{
    l = left(i)
    r = right(i)
    largest = l
    If l ≤ n and A[l] > A[largest]
        largest = l
    If r ≤ n and A[r] > A[largest]
        largest = r
    If largest ≠ i
        swap(A[i], A[largest]
MAX-HEAPIFY(A, largest, n)
}
```

MAX-HEAPIFY

Analysis

```
MAX-HEAPIFY(A, i, n)
{
    l = left(i)
    r = right(i)
    largest = l
    If l ≤ n and A[l] > A[largest]
        largest = l
    If r ≤ n and A[r] > A[largest]
        largest = r
    If largest ≠ i
        swap(A[i], A[largest]
MAX-HEAPIFY(A, largest, n)
}
```

- In the worst case, the MAX-HEAPIFY will be called recursively h times where h is the height of heap(binary tree).
- Since, each call to MAX-HEAPIFY takes constant time, $T(n) = O(h) = \mathbf{O(\log_2 n)}$

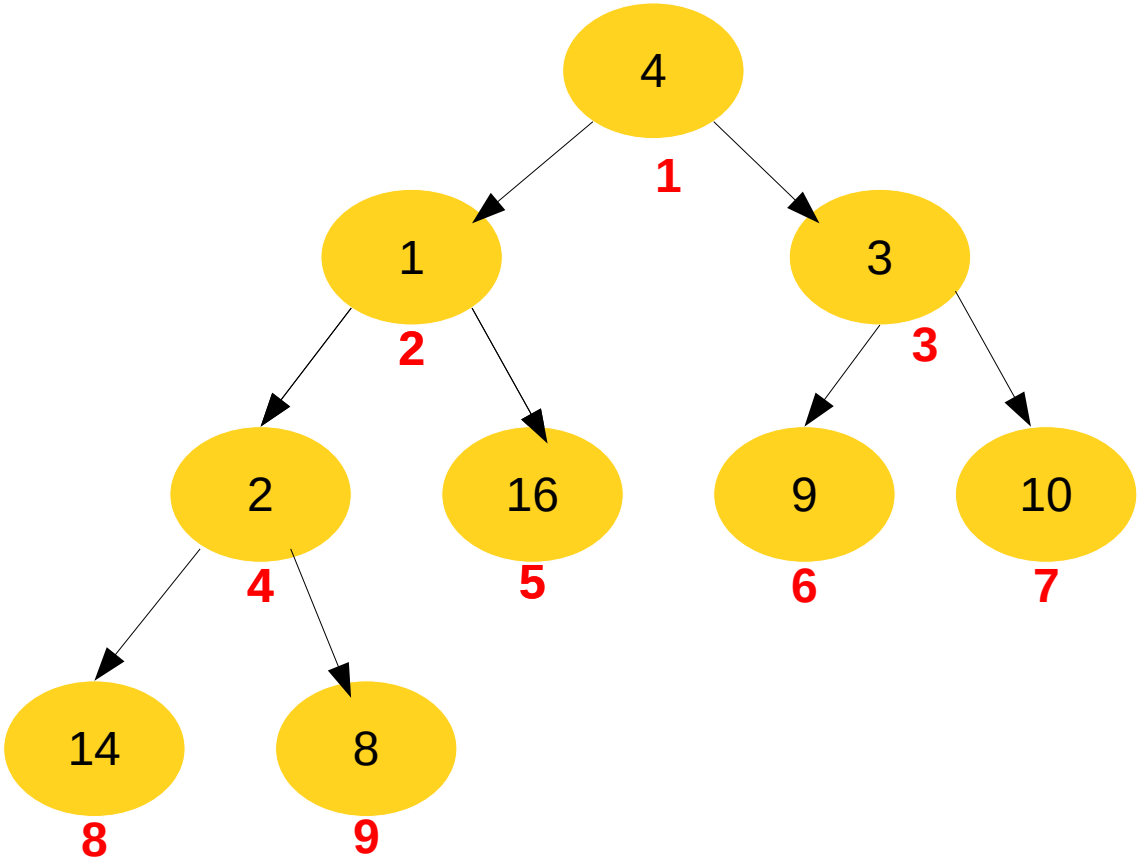
BUILD-MAX-HEAP

- To build a MAX-HEAP from any tree, we can start heapifying each sub-tree from the bottom up and end up with a MAX-HEAP after the entire tree satisfies MAX-HEAP property.
- We perform the heapify operation on all non-leaf nodes from floor(n/2) to 1.

BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

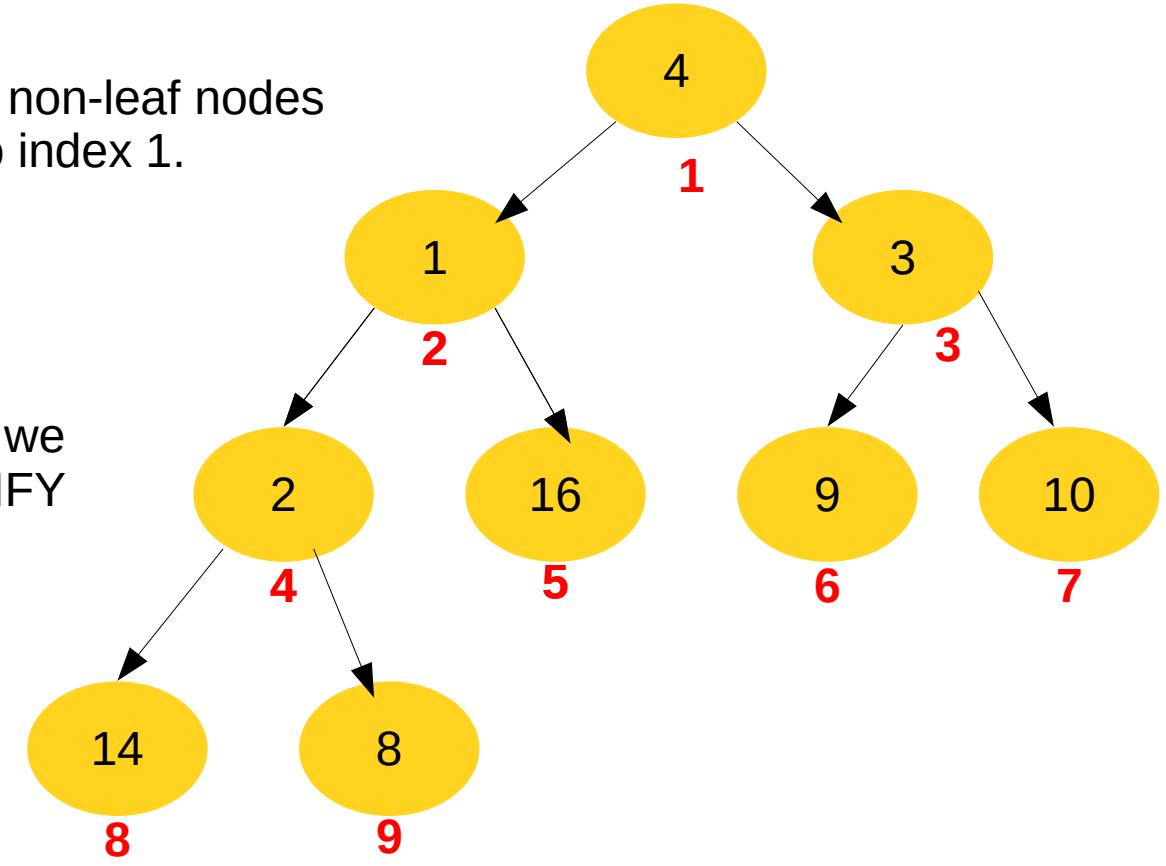


BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

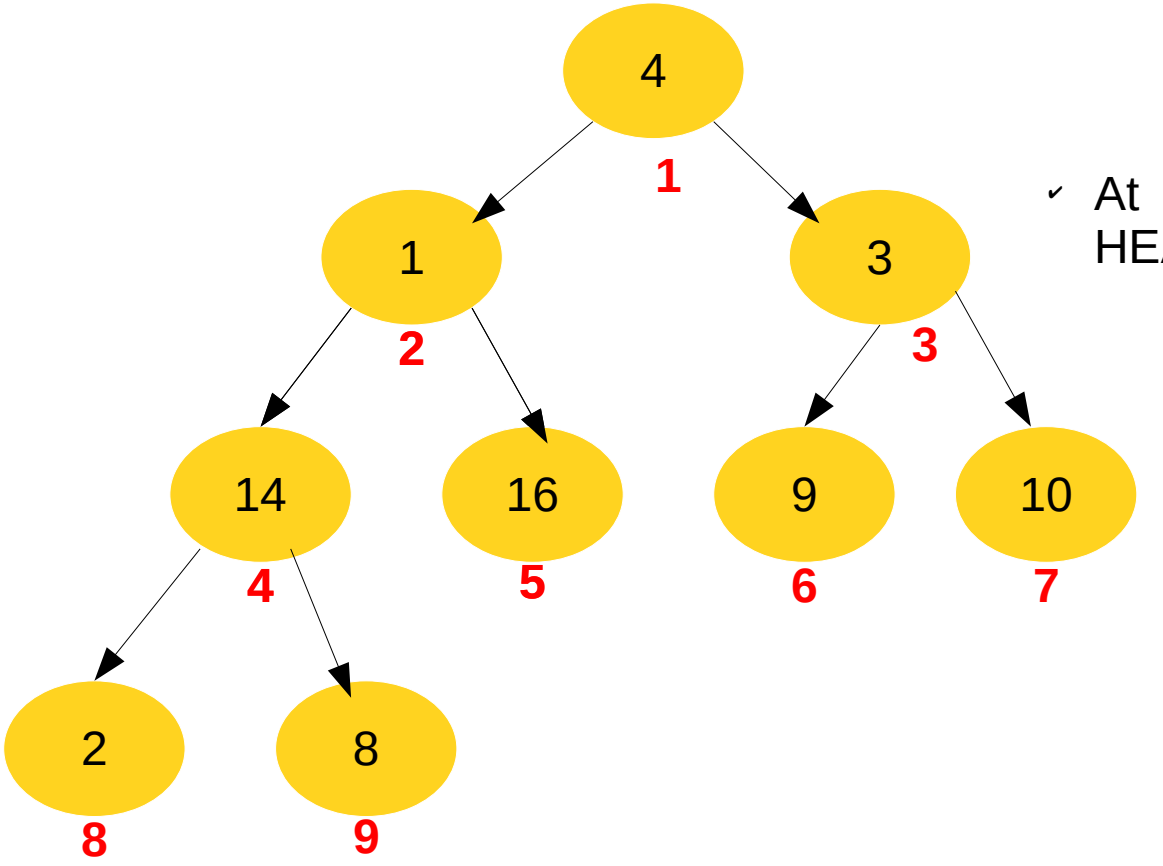
- We will look at non-leaf nodes from index 4 to index 1.
- At index 4, we perform HEAPIFY operation



BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}



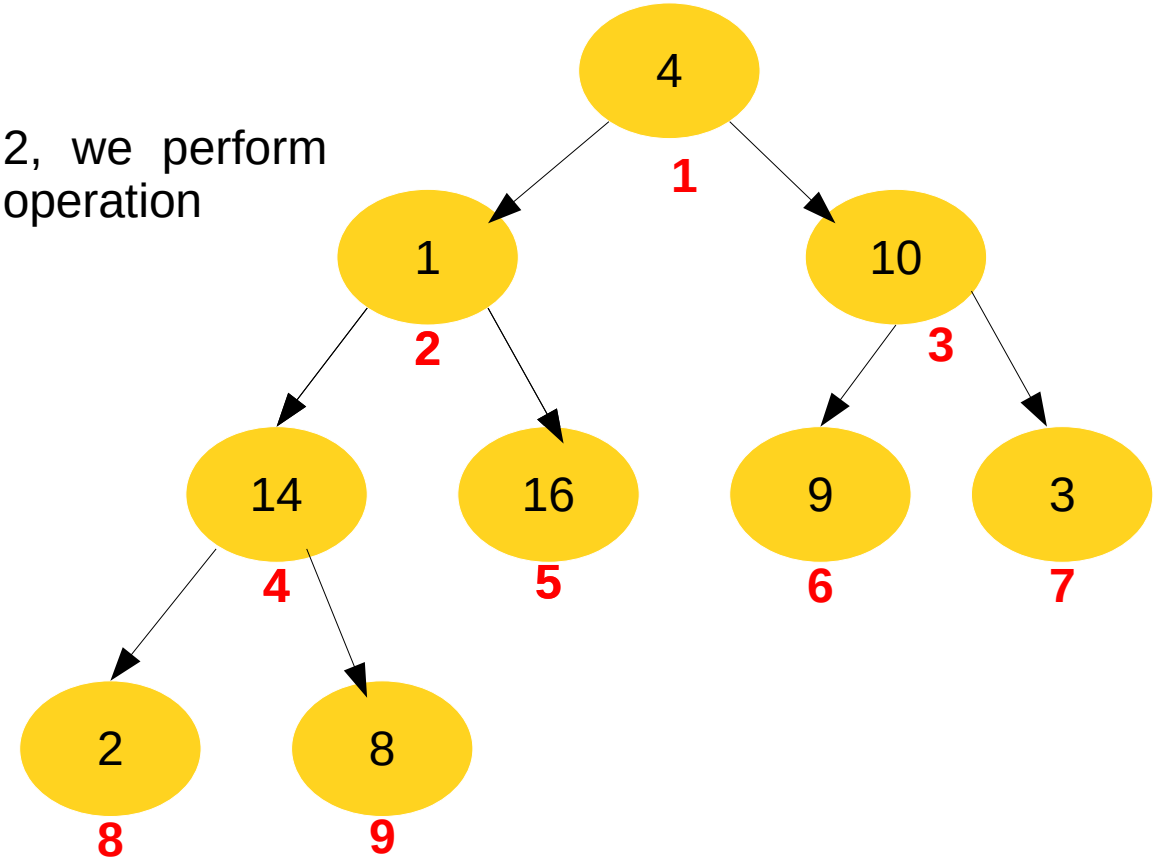
- At index 3, we perform HEAPIFY operation

BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- At index 2, we perform HEAPIFY operation

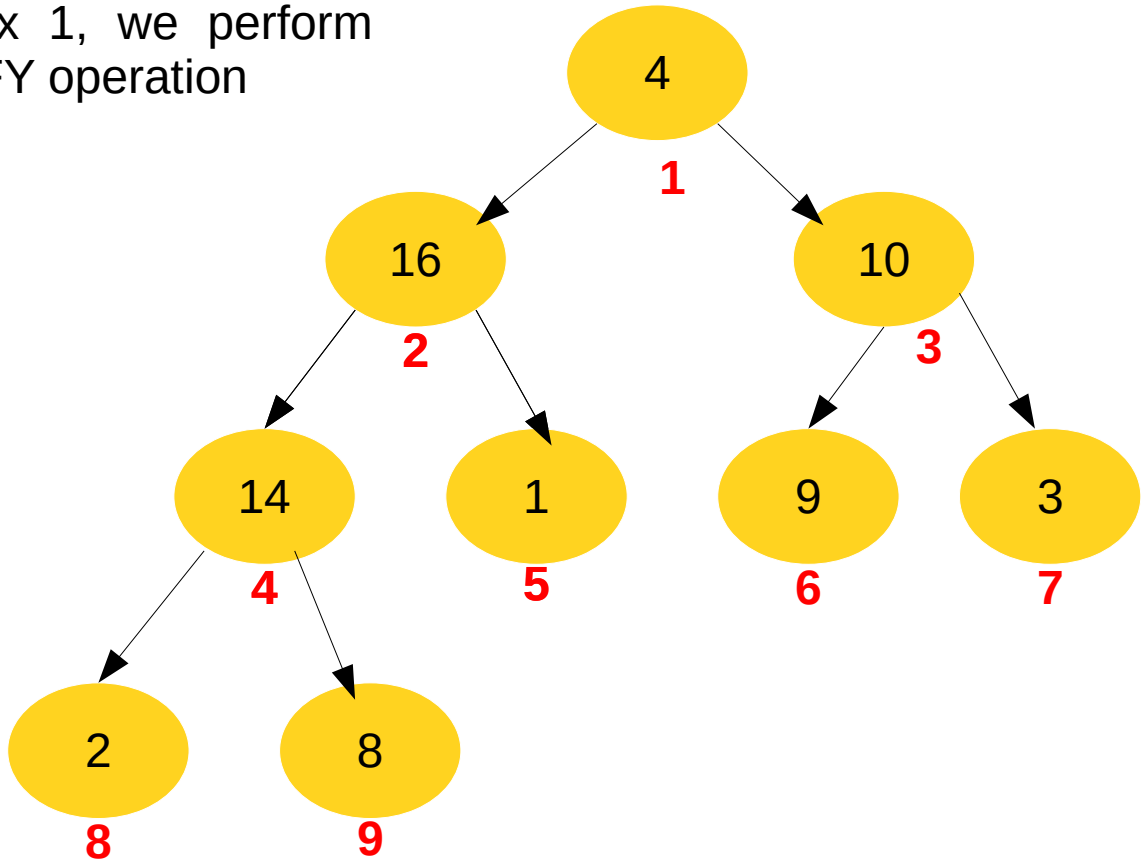


BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

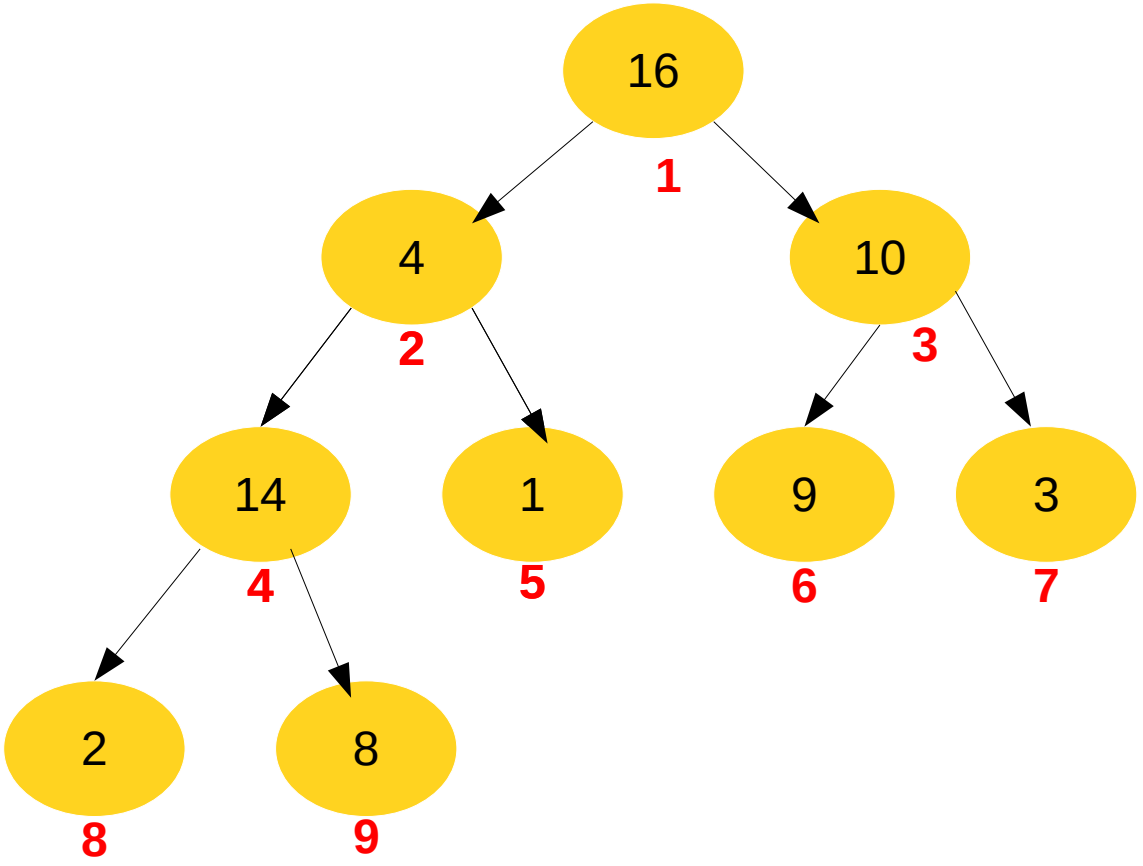
- At index 1, we perform HEAPIFY operation



BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

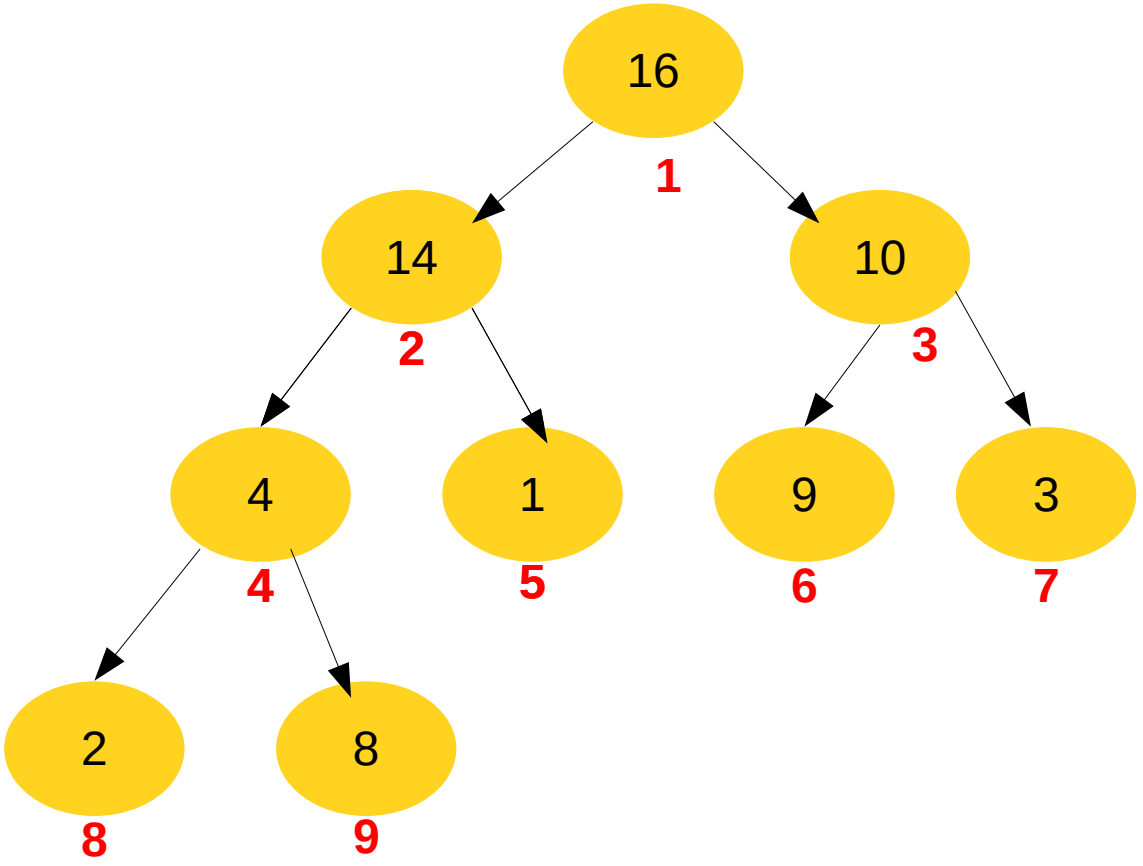
A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}



BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

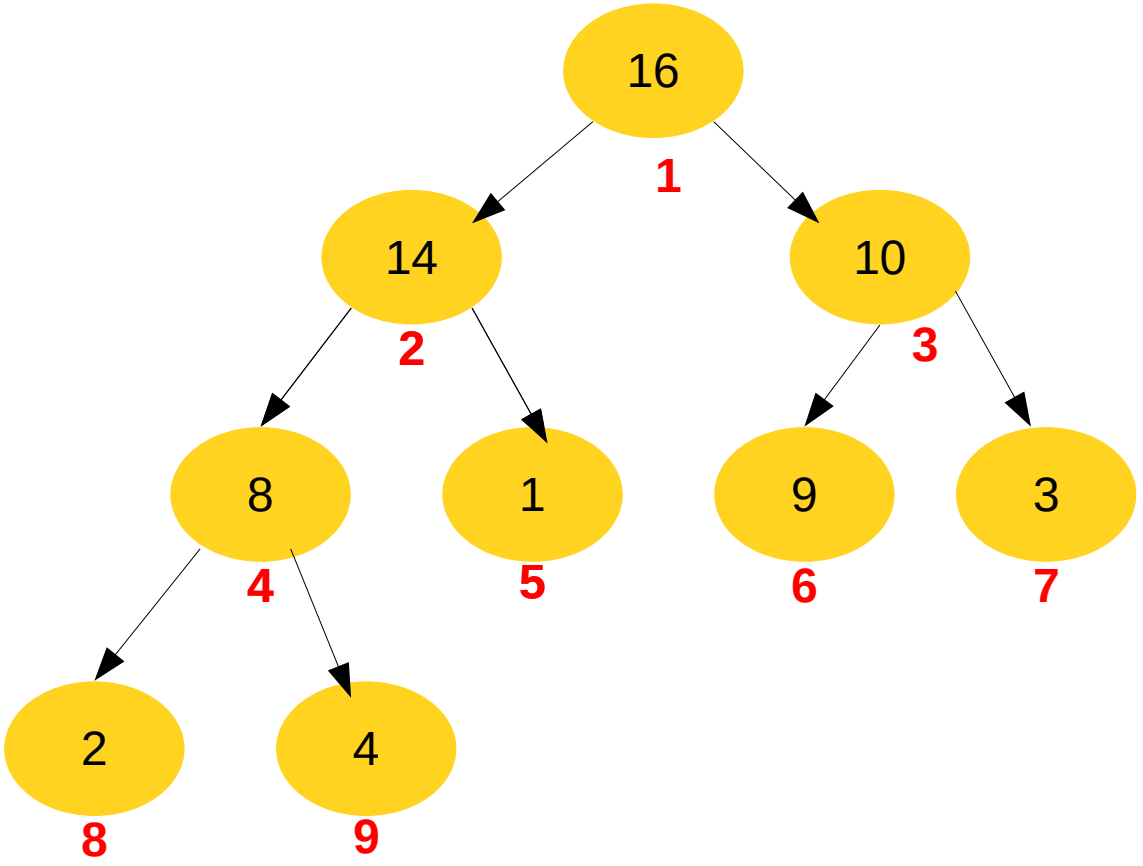
A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}



BUILD-MAX-HEAP

Example : Construct a MAX heap from a set of following elements

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}



BUILD-MAX-HEAP

Algorithm

```
BUILD-MAX-HEAP(A)
{
    n = length[A]
    for(i = floor(n/2); i>=1; i--)
    {
        MAX-HEAPIFY(A, i, n);
    }
}
```

BUILD-MAX-HEAP

Analysis

```
BUILD-MAX-HEAP(A)
{
    n = length[A]
    for(i = floor(n/2); i>=1; i--)
    {
        MAX-HEAPIFY(A, i, n);
    }
}
```

- In the worst case, the for loop runs for O(n) times.
- Inside the for loop, the MAX-HEAPIFY runs for $O(\log_2 n)$ times.
- $T(n) = O(n)*O*(\log_2 n)$
- $T(n) = O(n \log_2 n)$

Heap Sort

- We can use the concept of heaps to sort numbers efficiently.
- The procedure is quite similar to selection sort, where we first find the maximum element and place the maximum element at the end.
- We repeat the same process for other elements.

Heap Sort

Algorithm

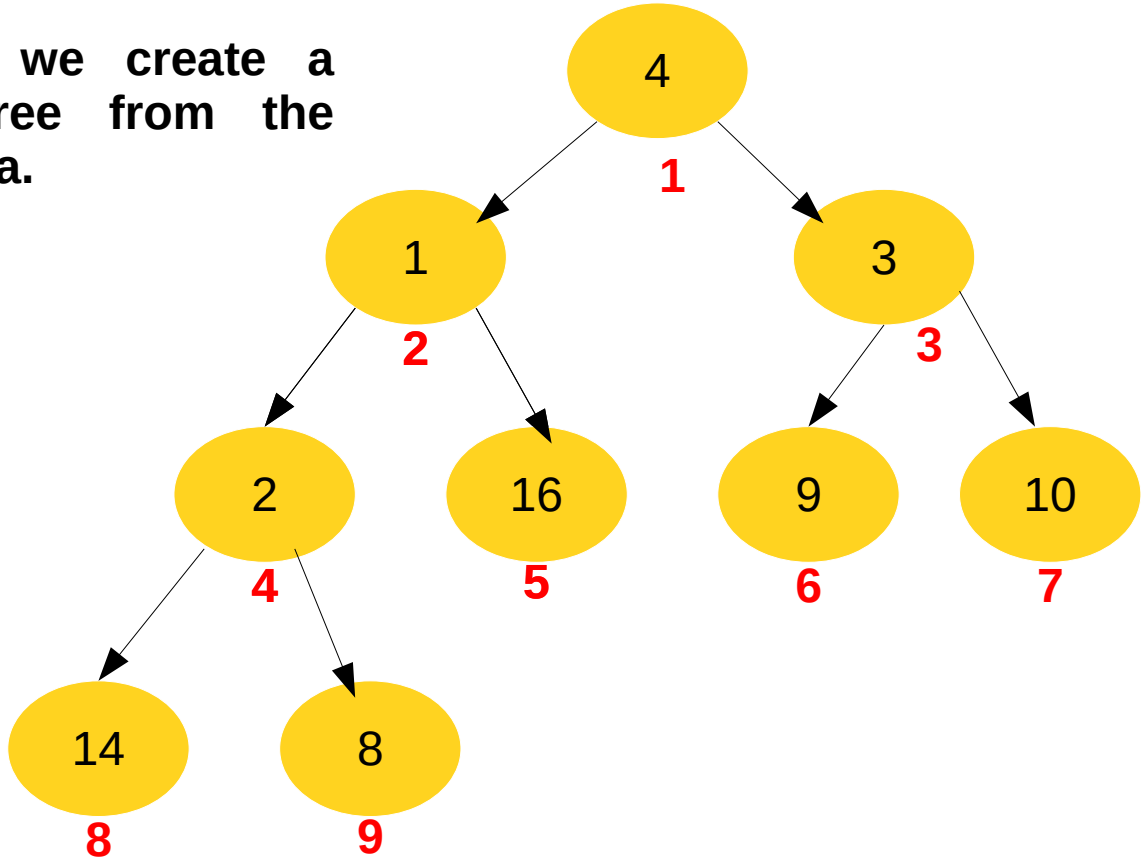
1. Build a MAX-HEAP from the array
2. Swap the root with the last element in the array.
3. Discard this last node by decreasing the heap size.
4. Perform MAX-HEAPIFY operation on the new root node.
5. Repeat until only one node remains.

Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **At first, we create a binary tree from the given data.**

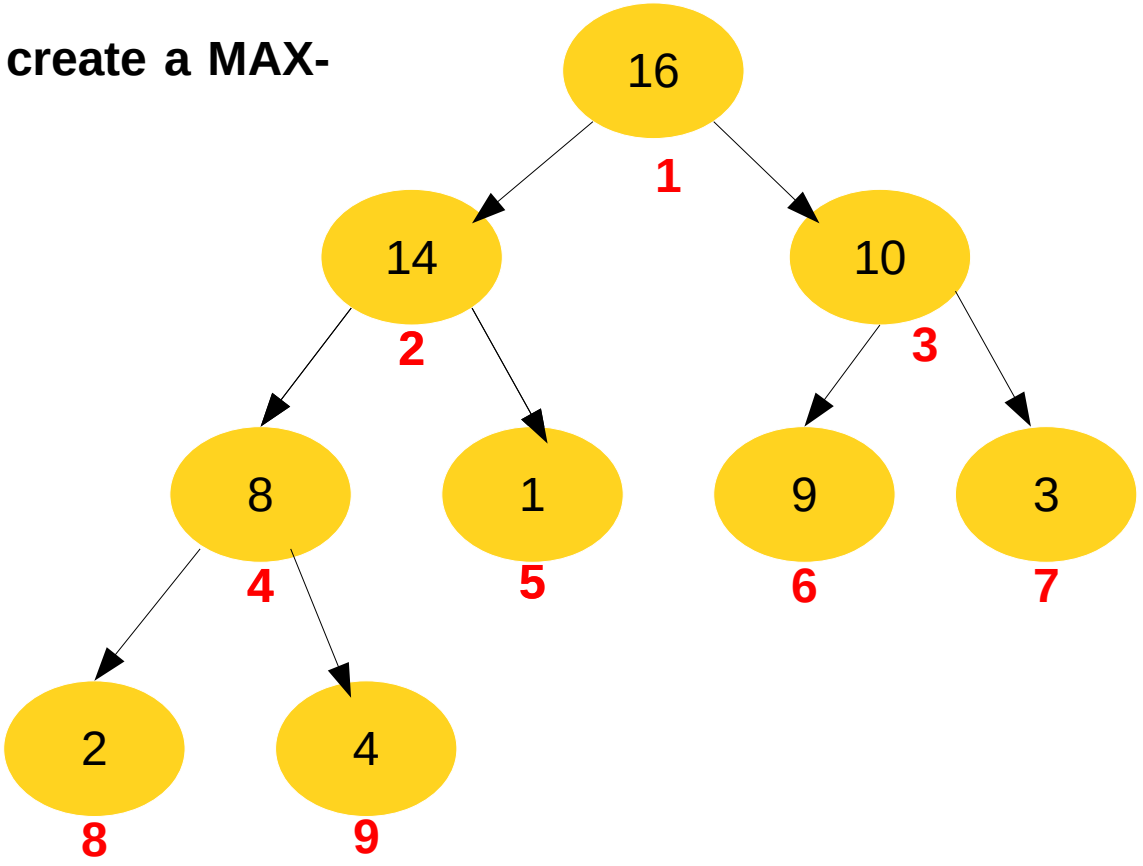


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **Then, we create a MAX-HEAP**

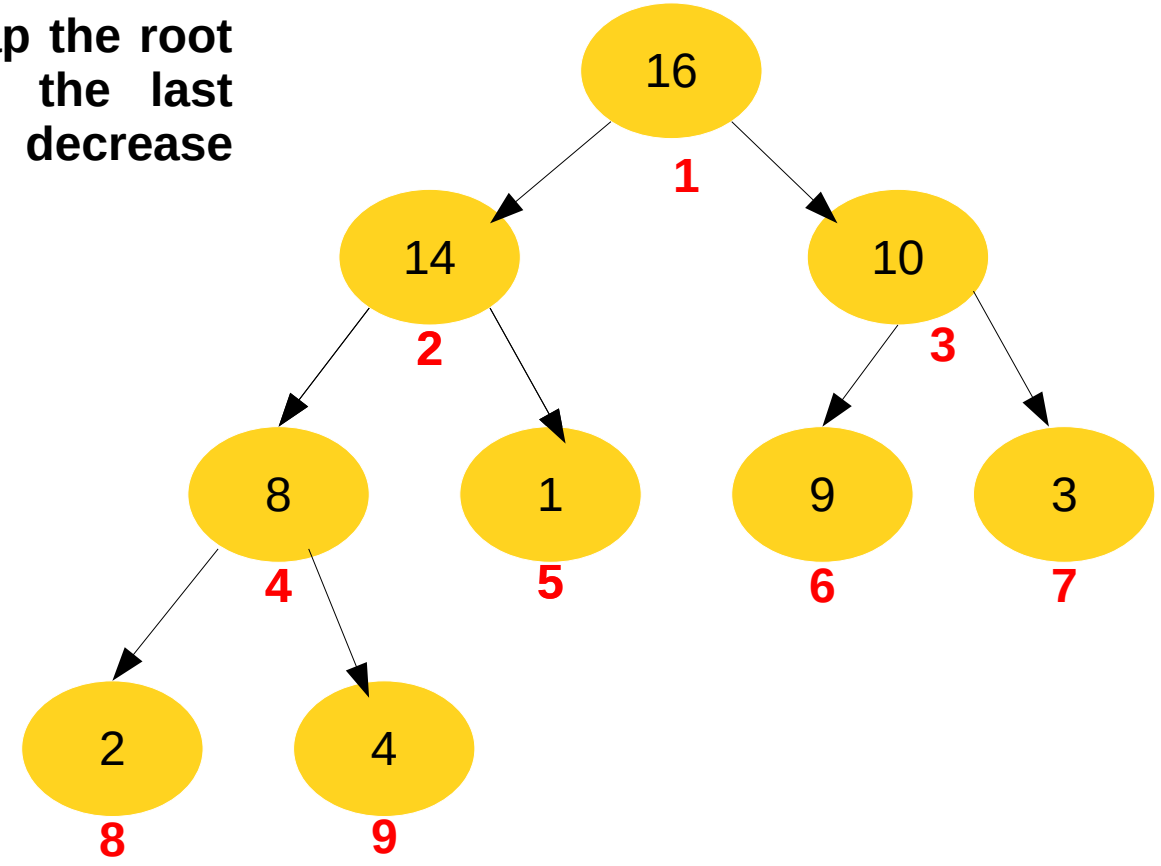


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

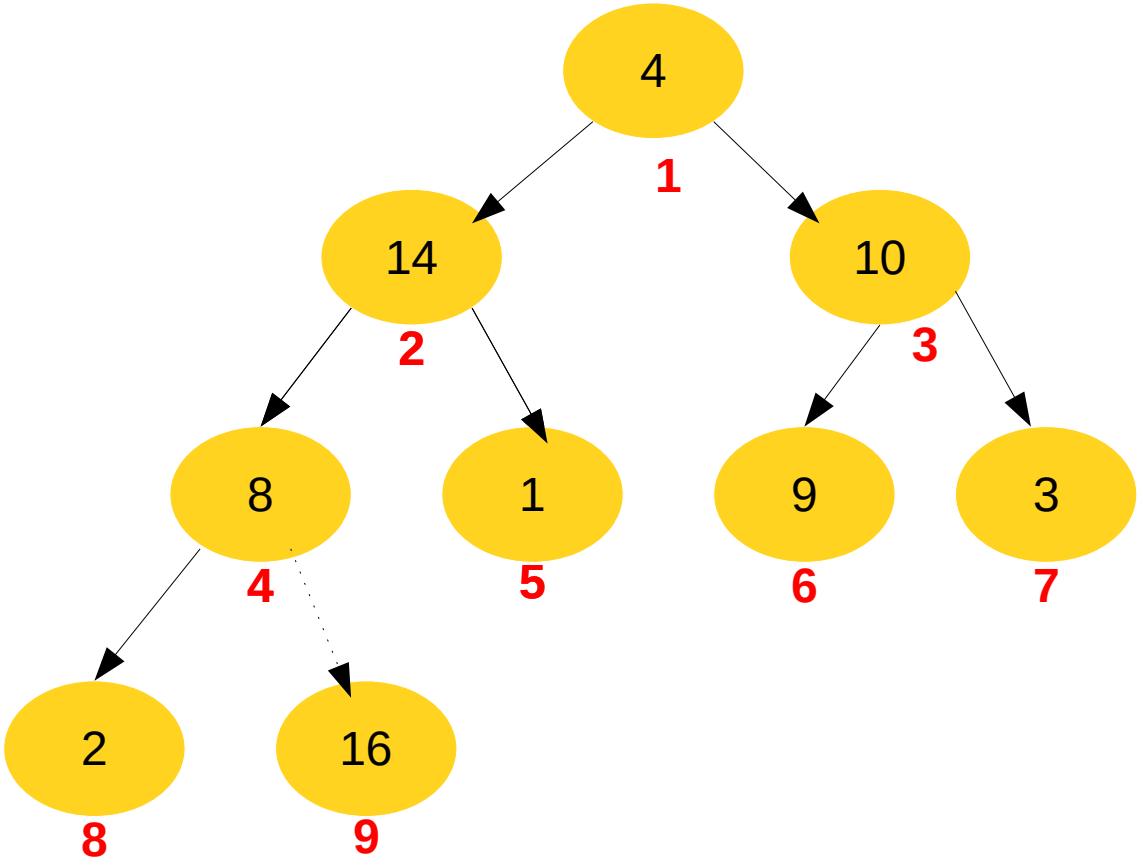
- **Then, we swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

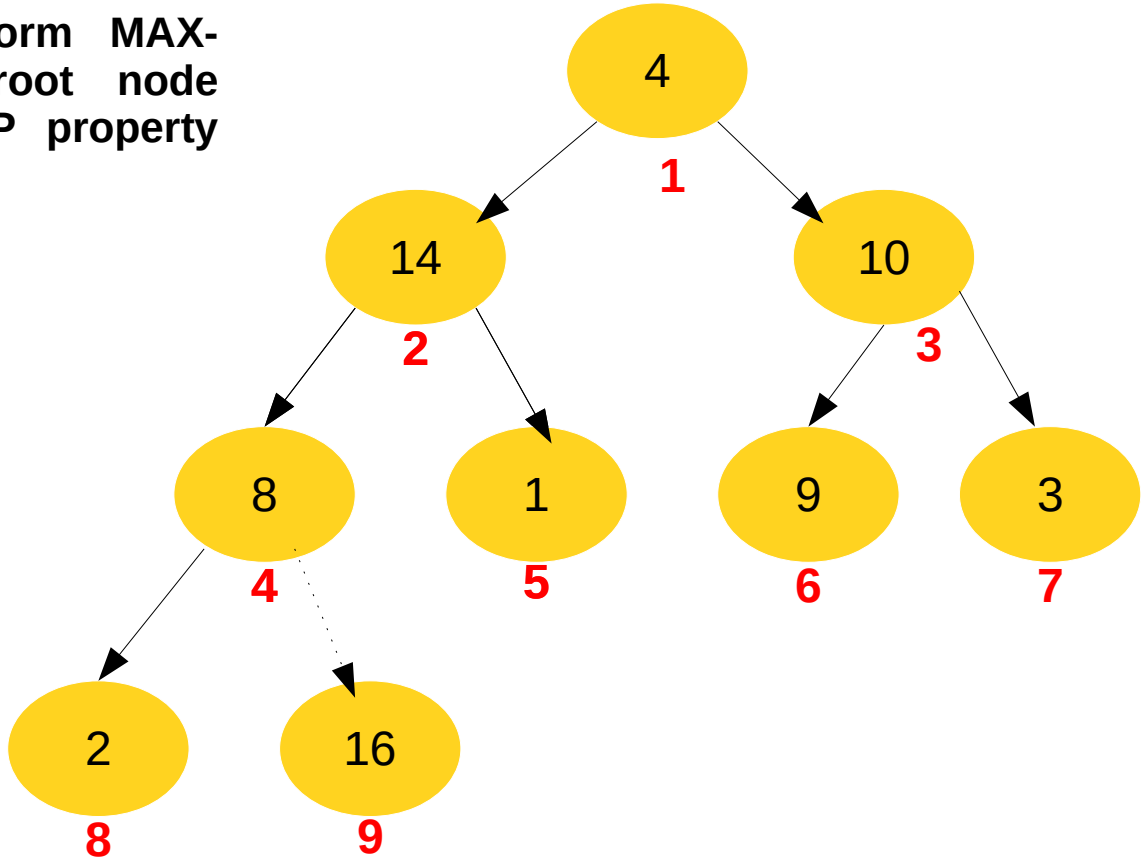


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

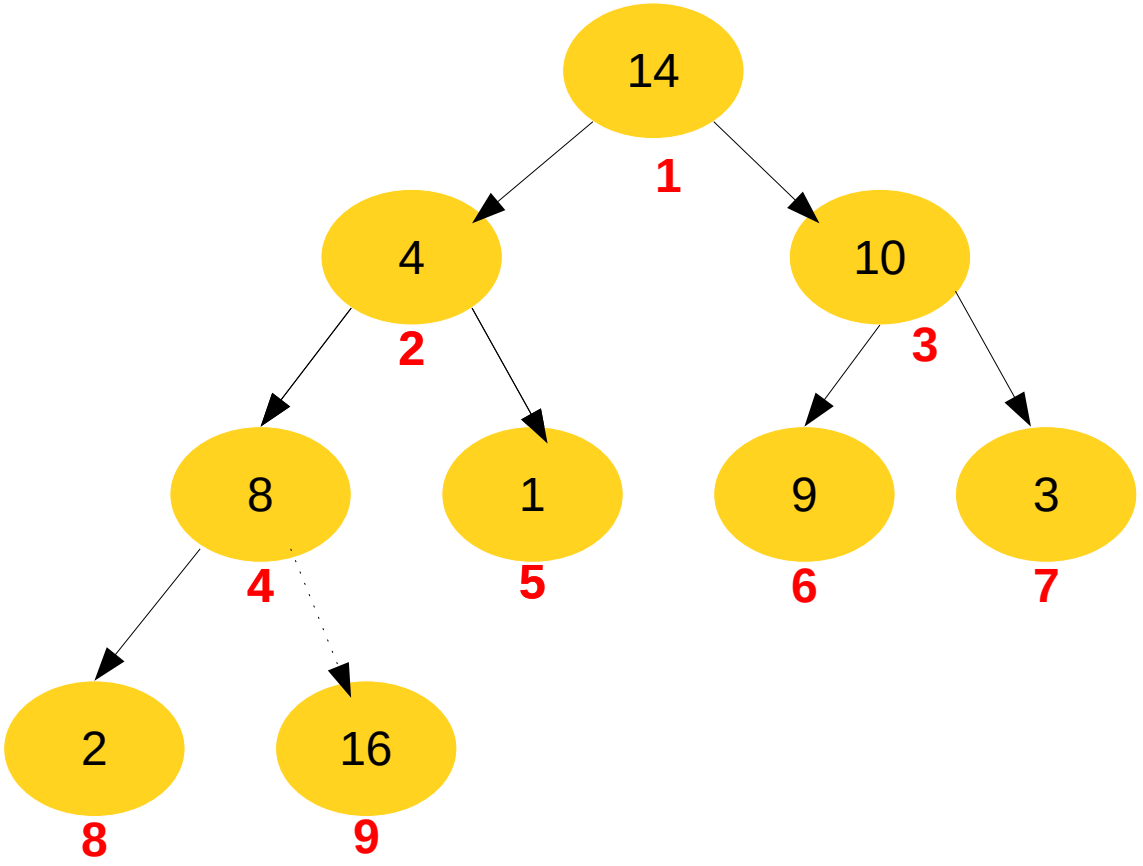
- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

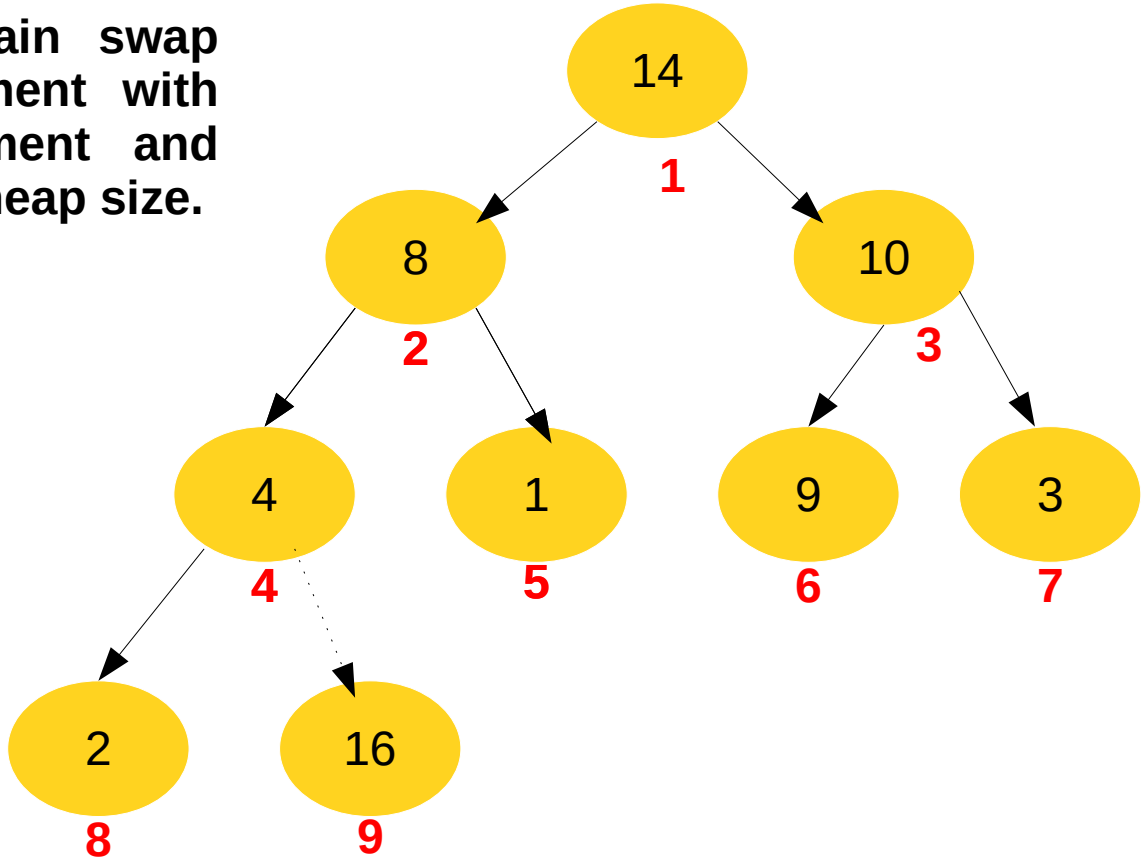


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

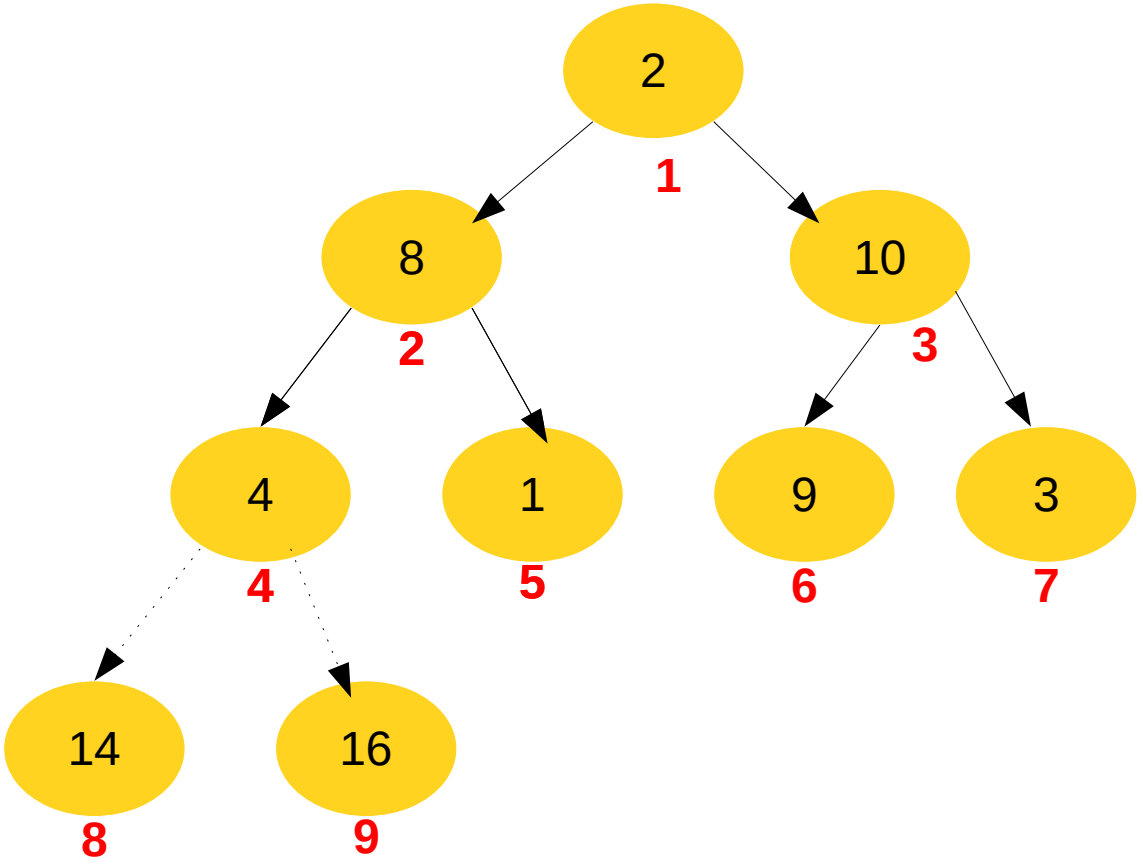
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

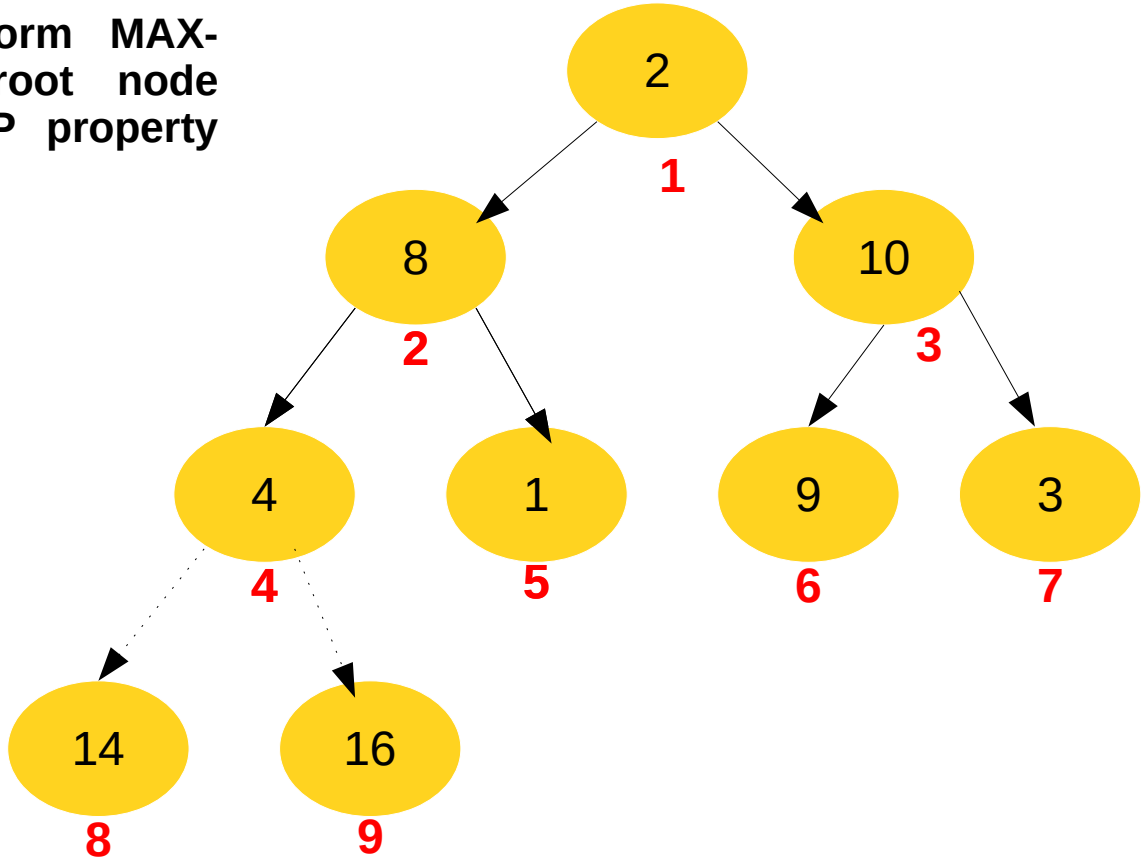


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

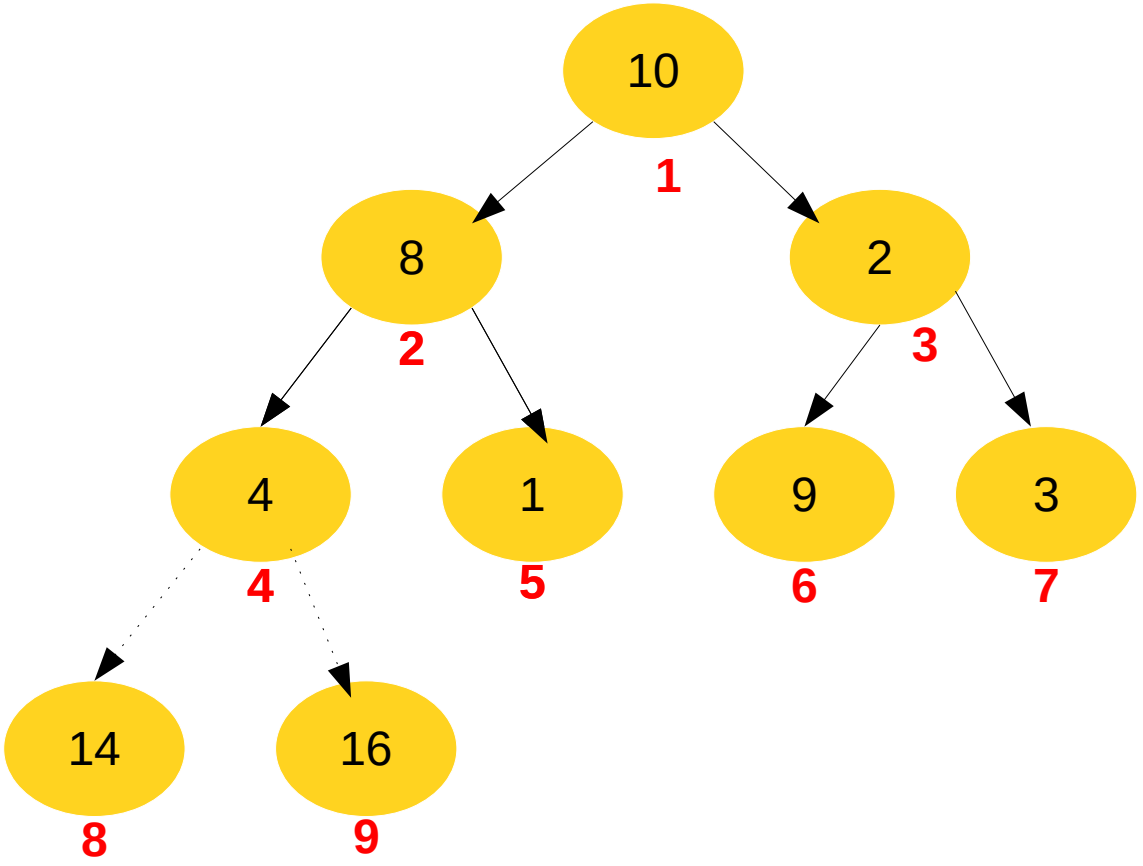
- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

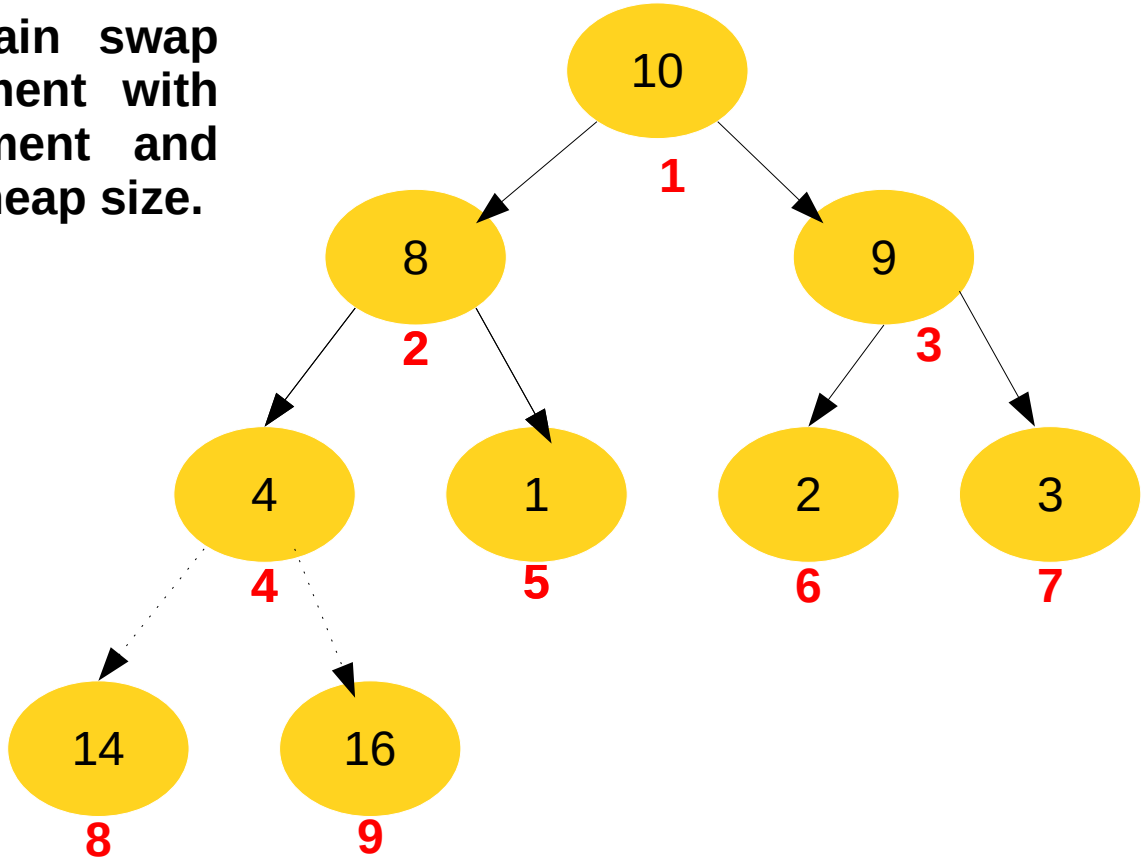


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

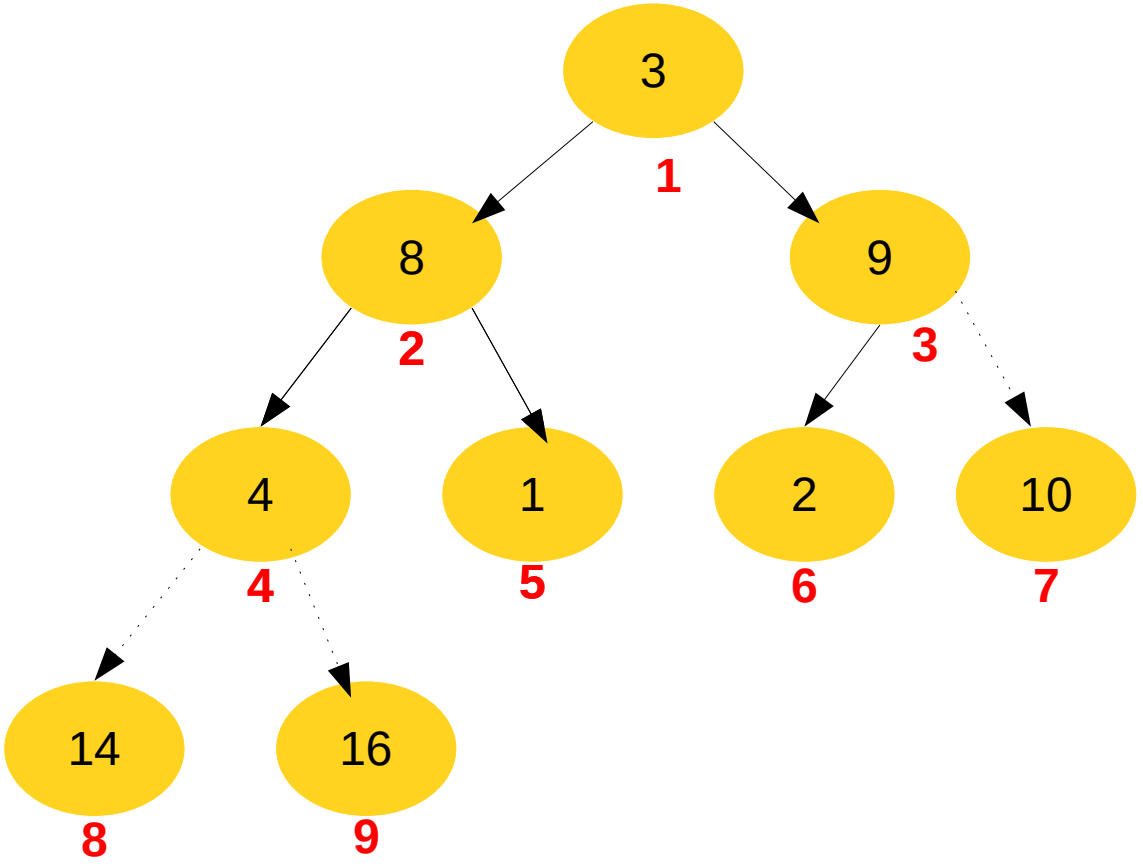
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

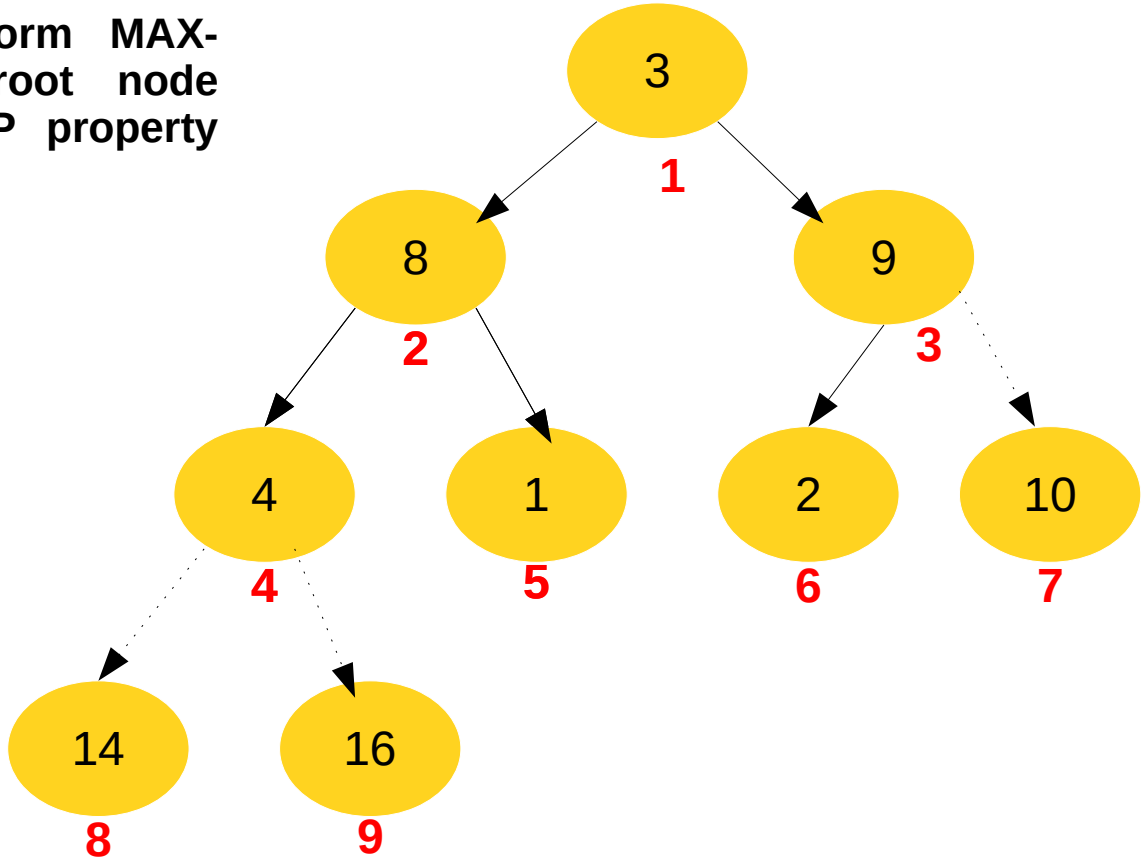


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**

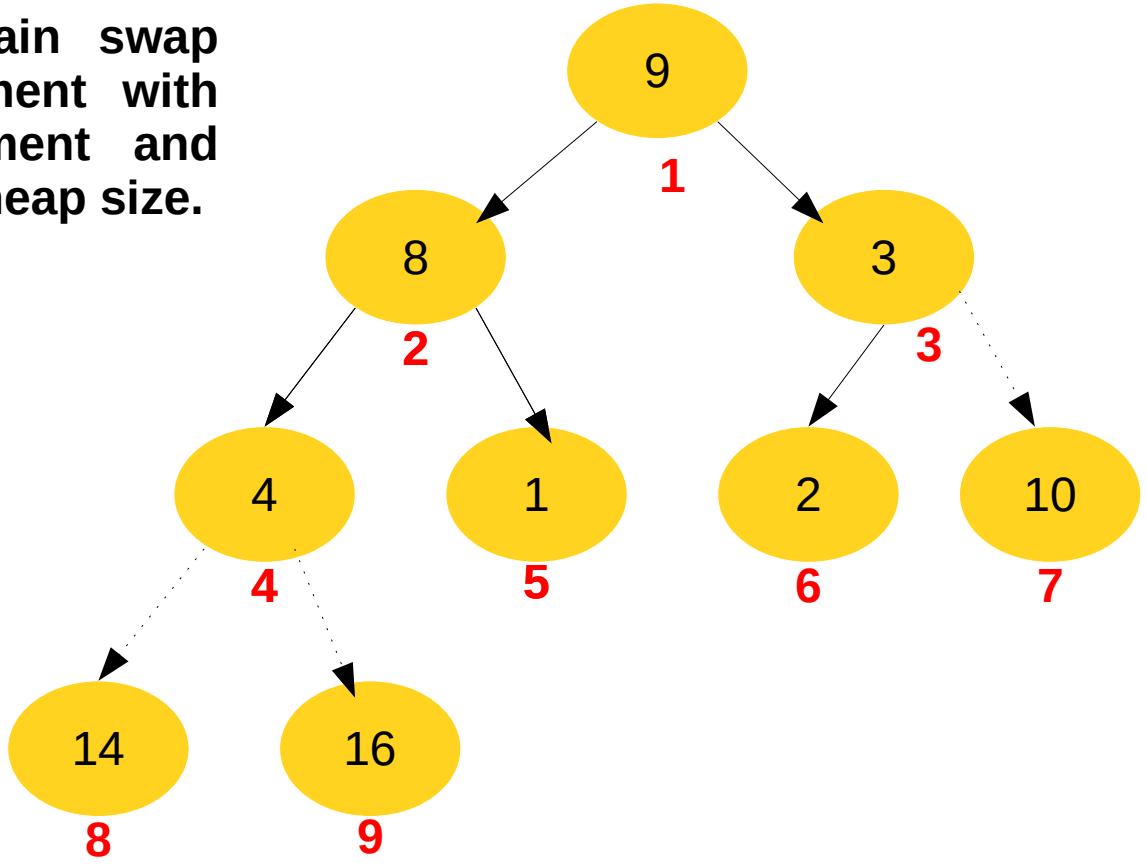


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

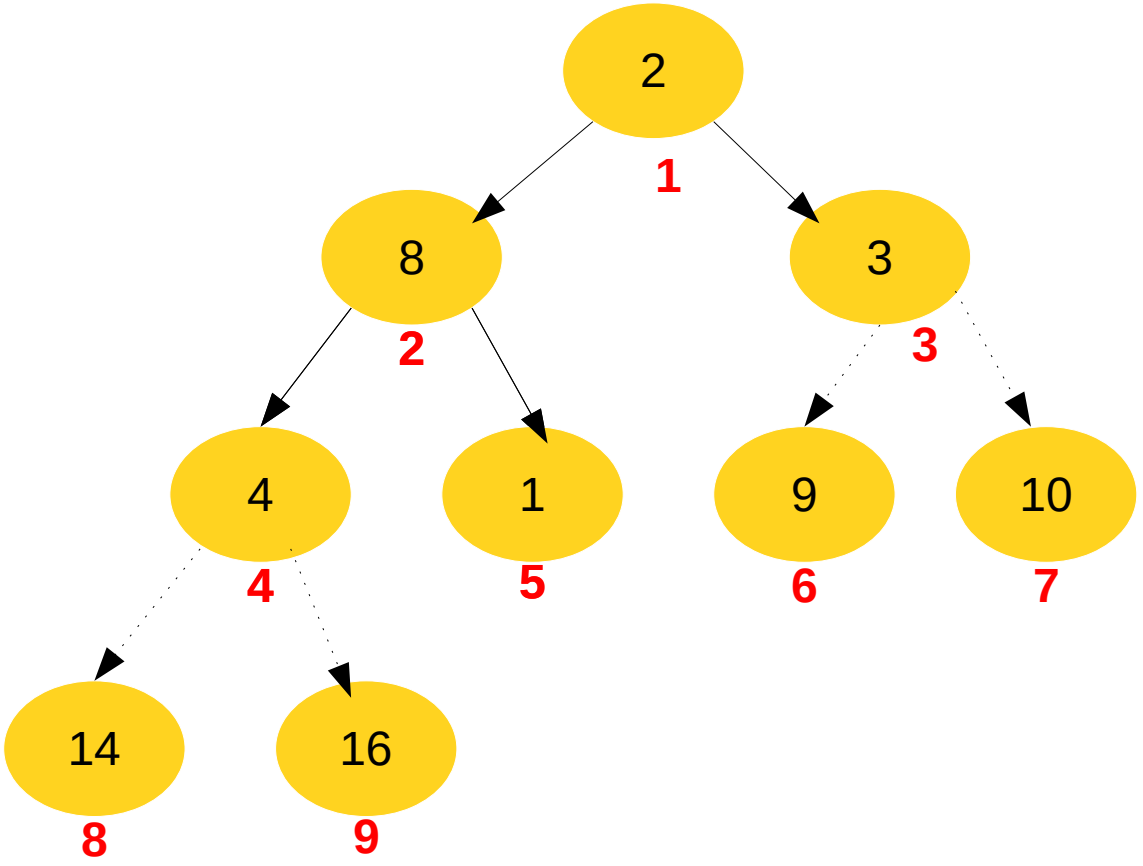
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

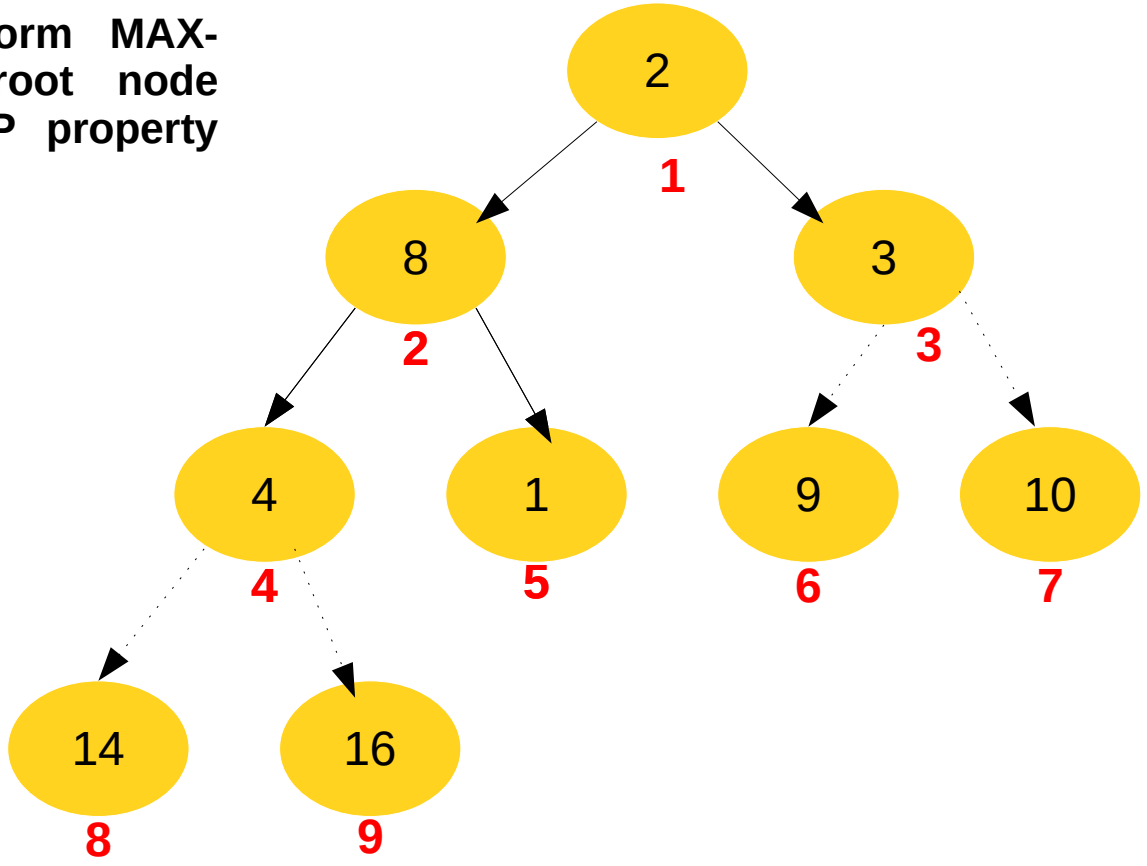


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

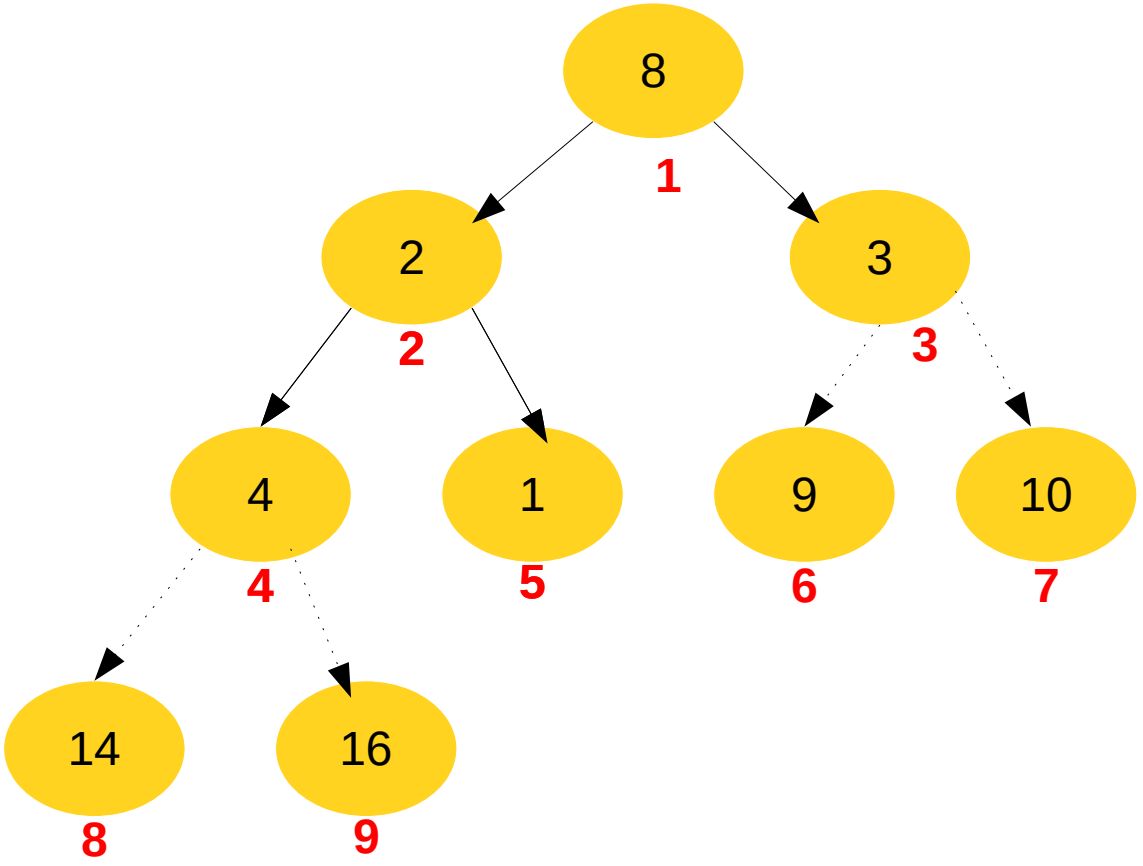
- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

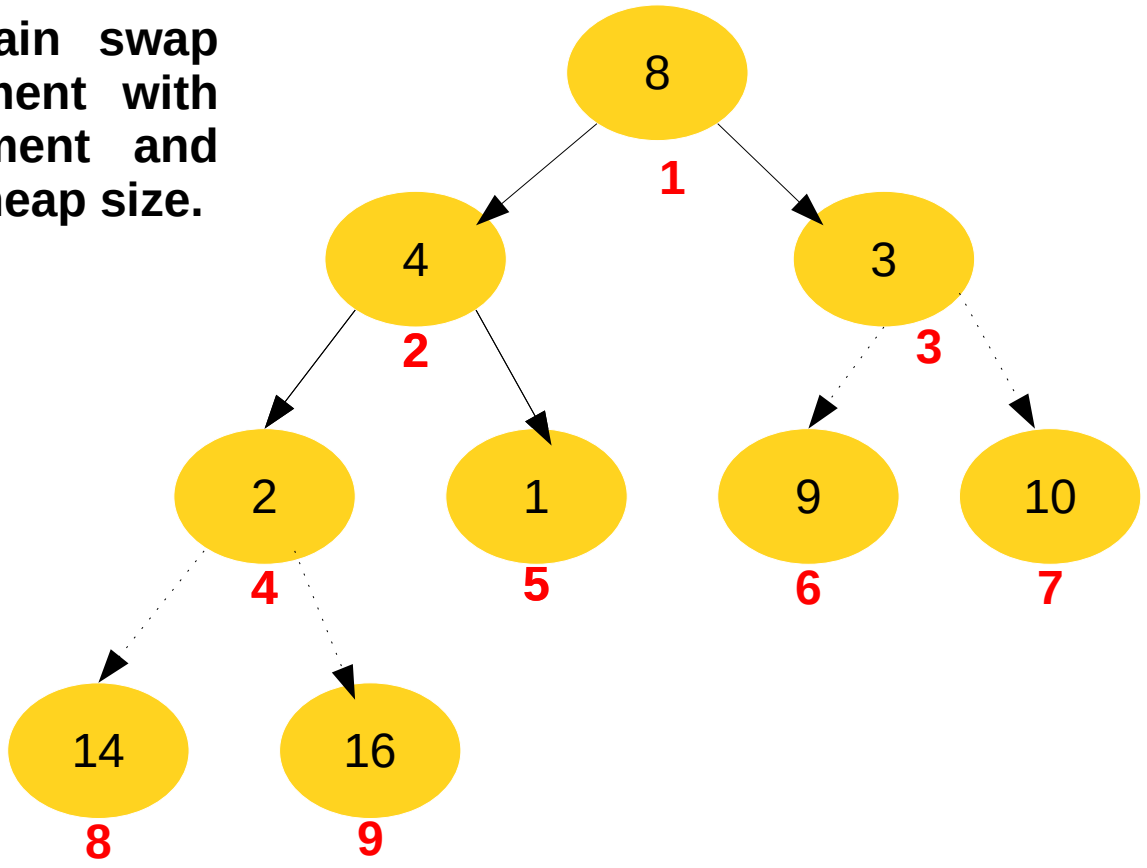


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

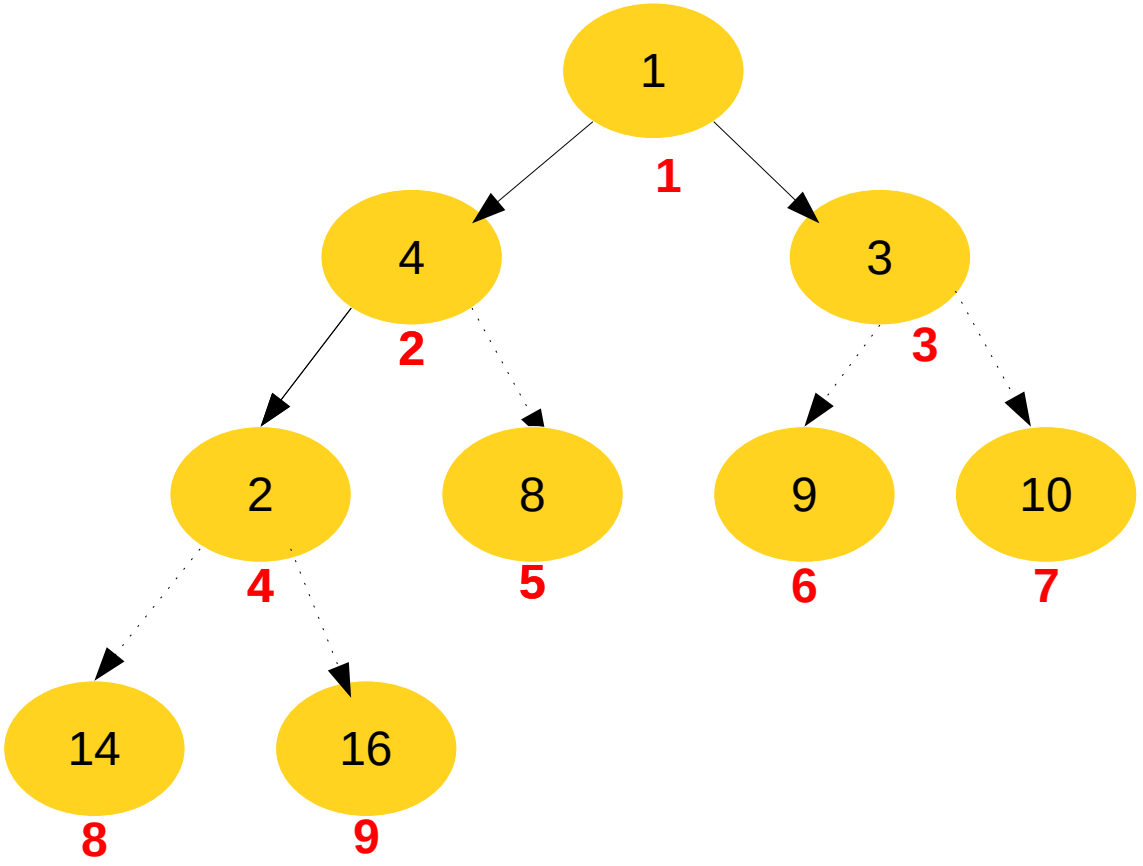
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

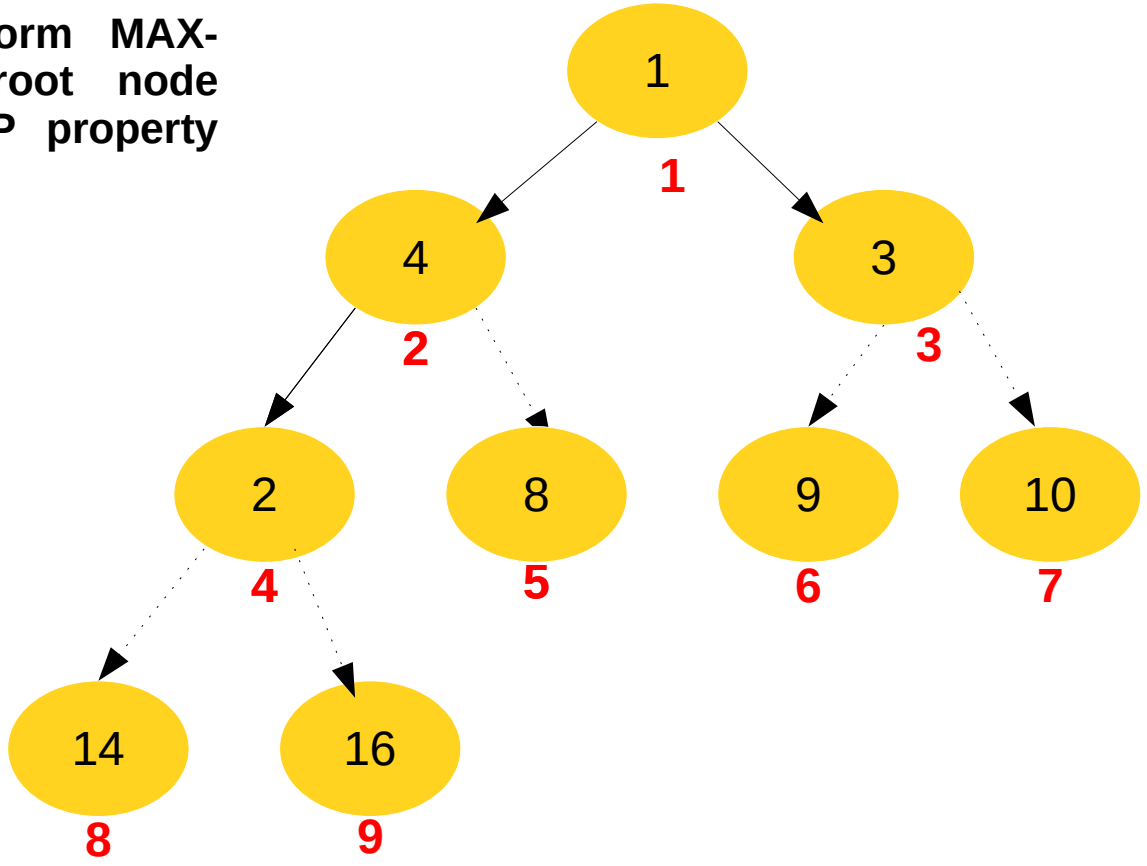


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

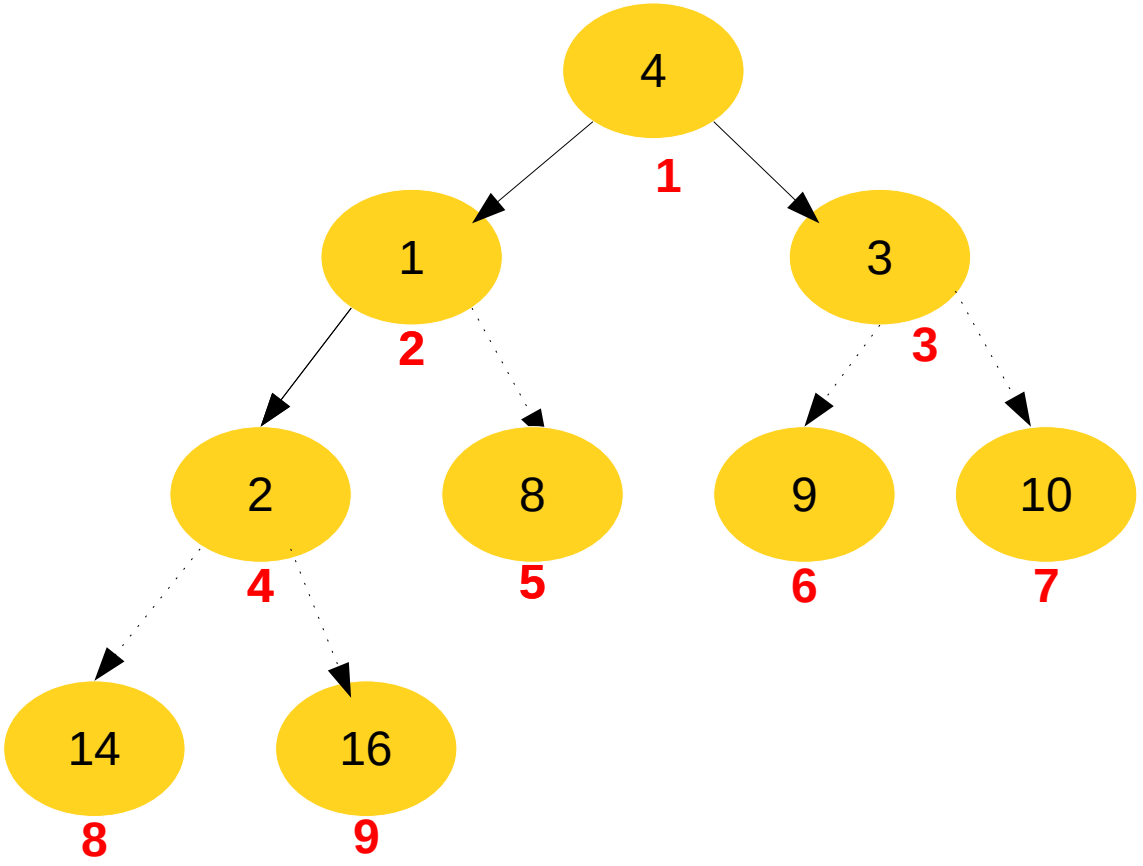
- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

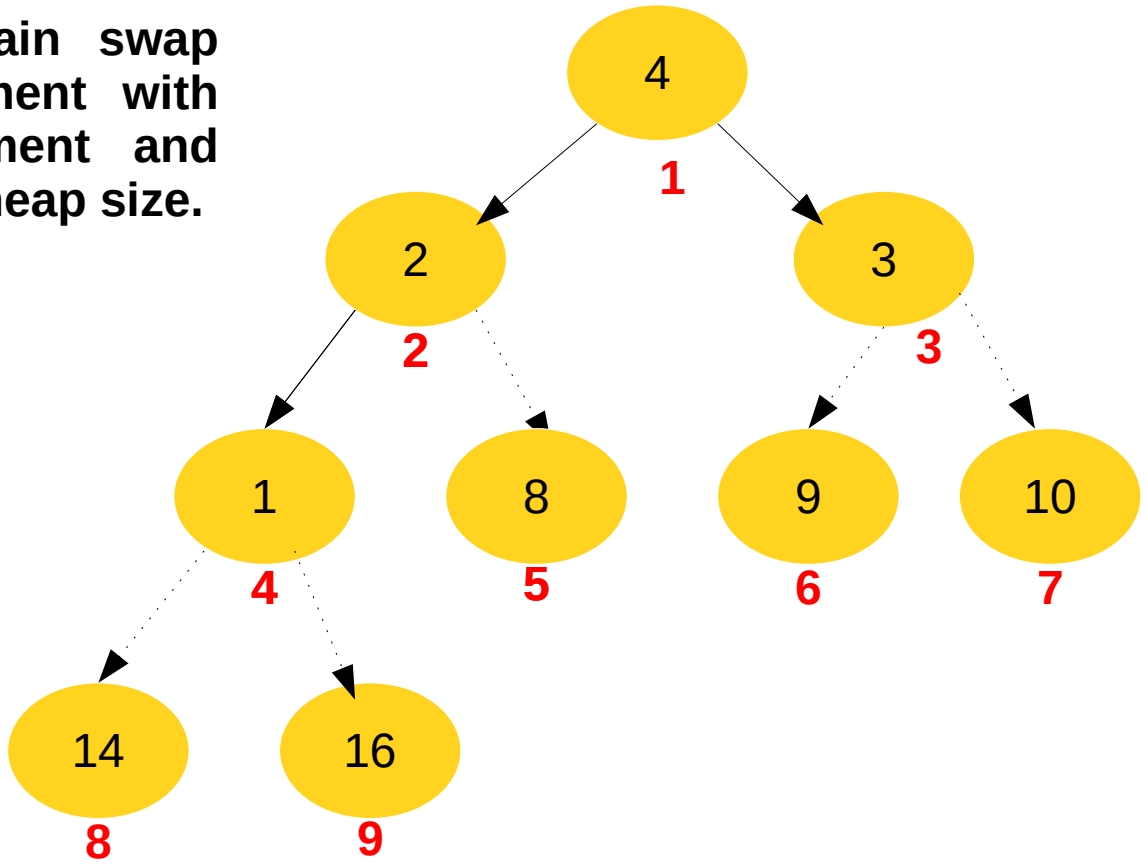


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

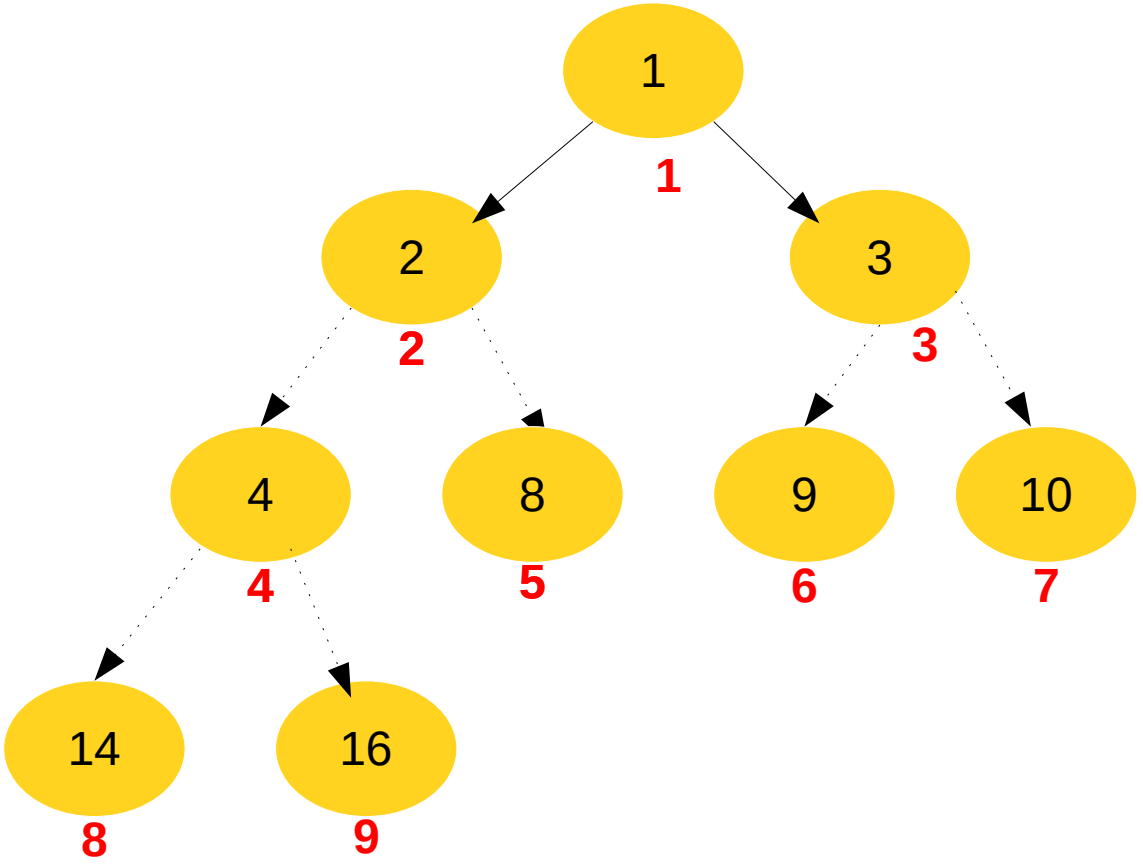
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

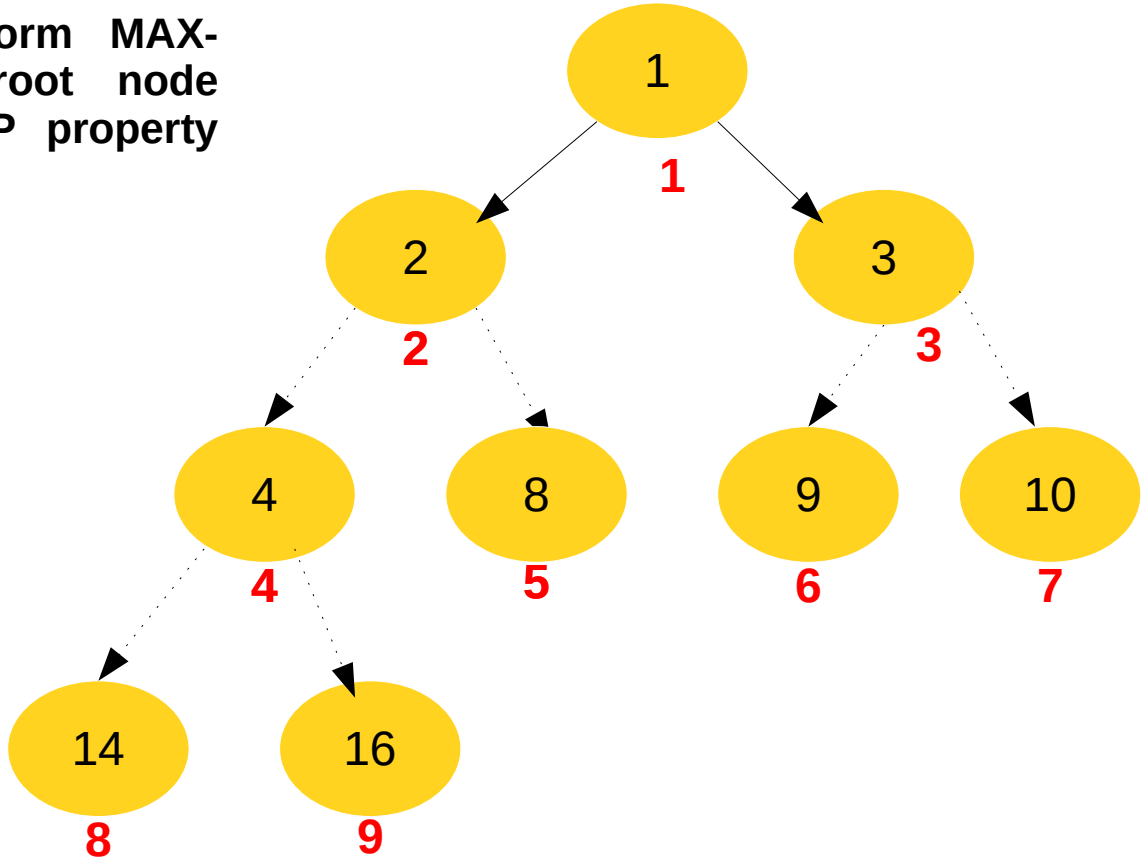


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**

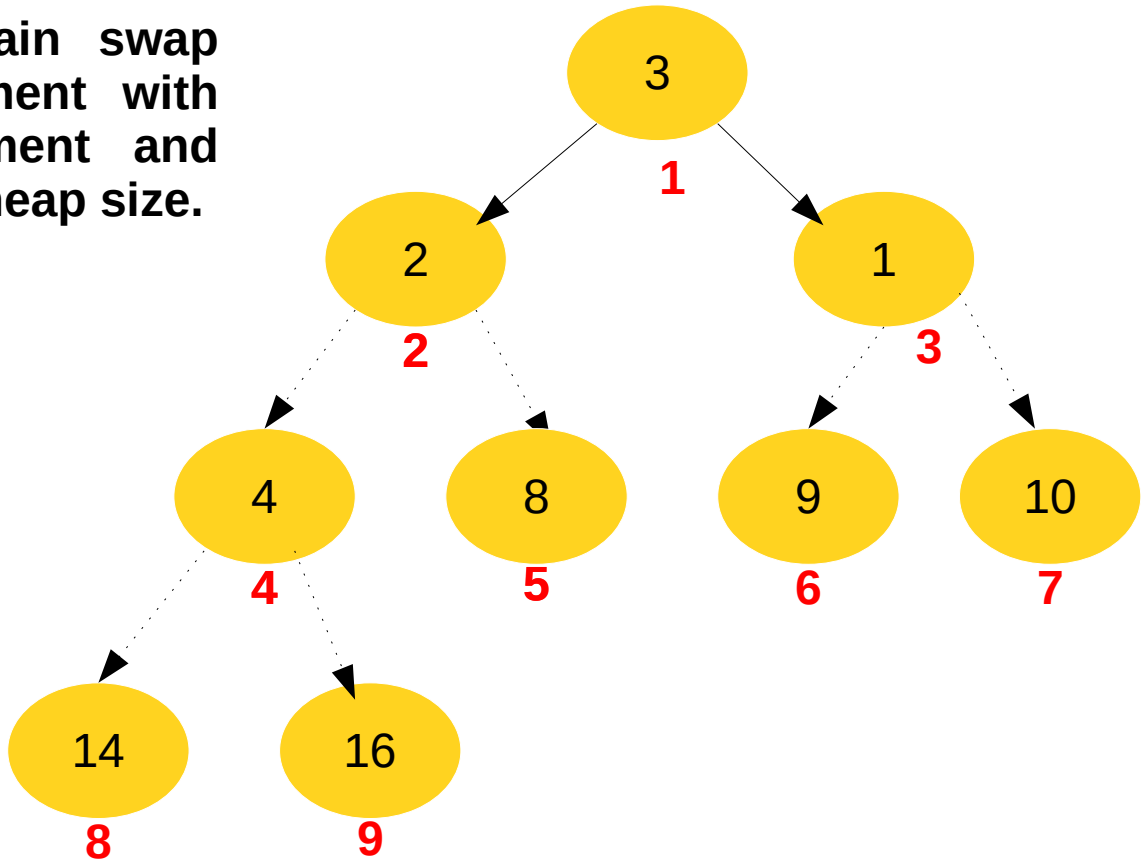


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

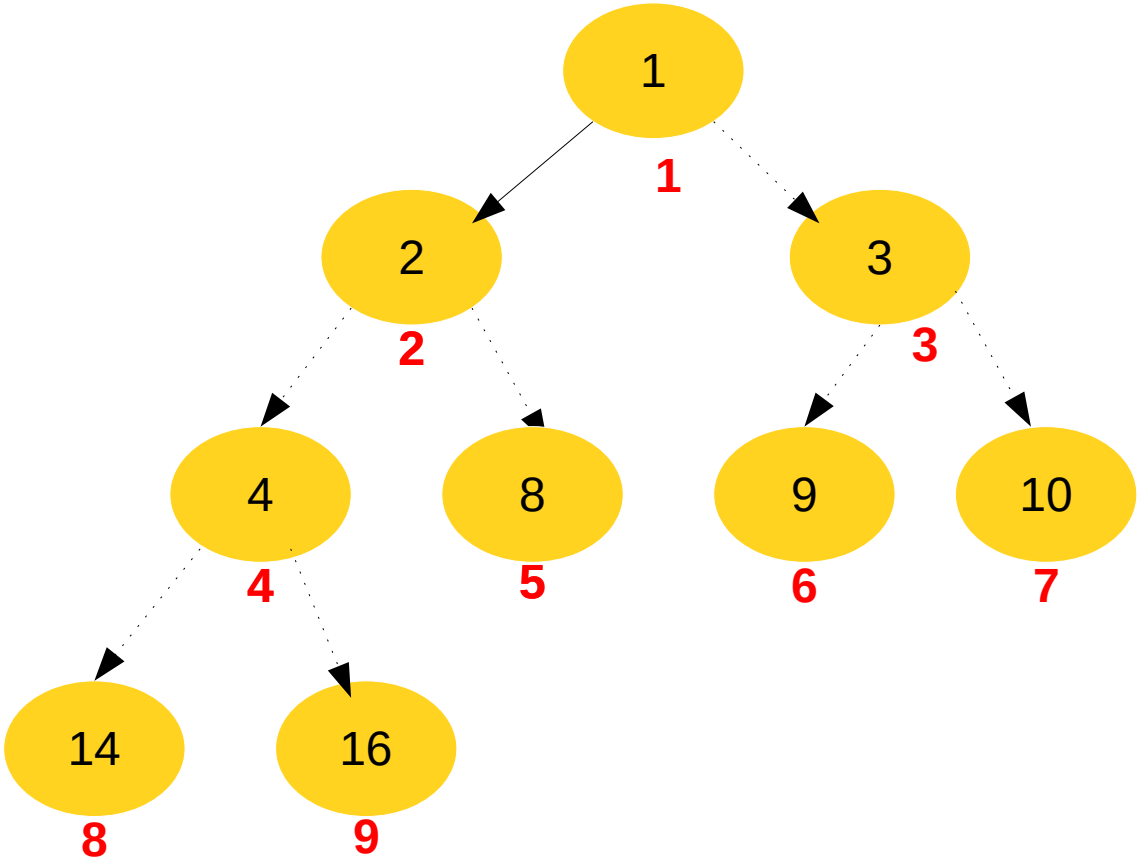
- **Then, we again swap the root element with the last element and decrease the heap size.**



Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

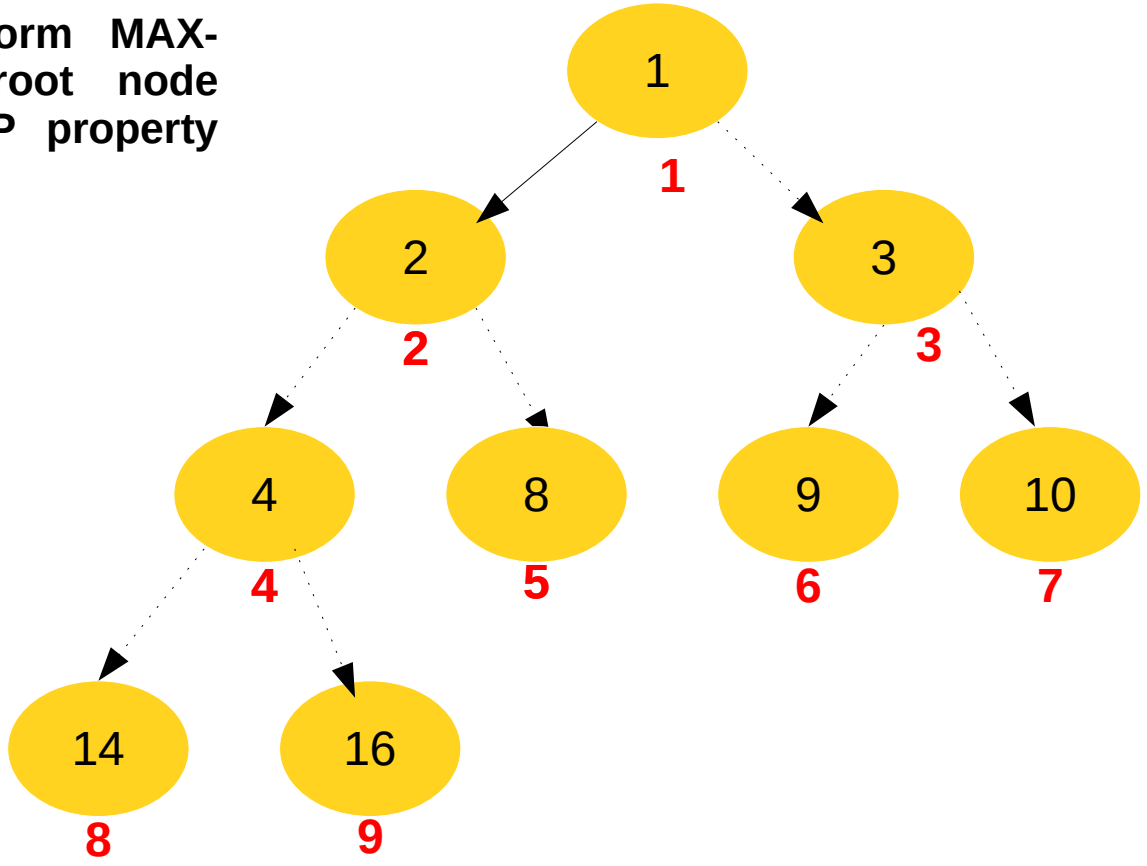


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **Then, we perform MAX-HEAPIFY on root node until MAX-HEAP property is restored.**

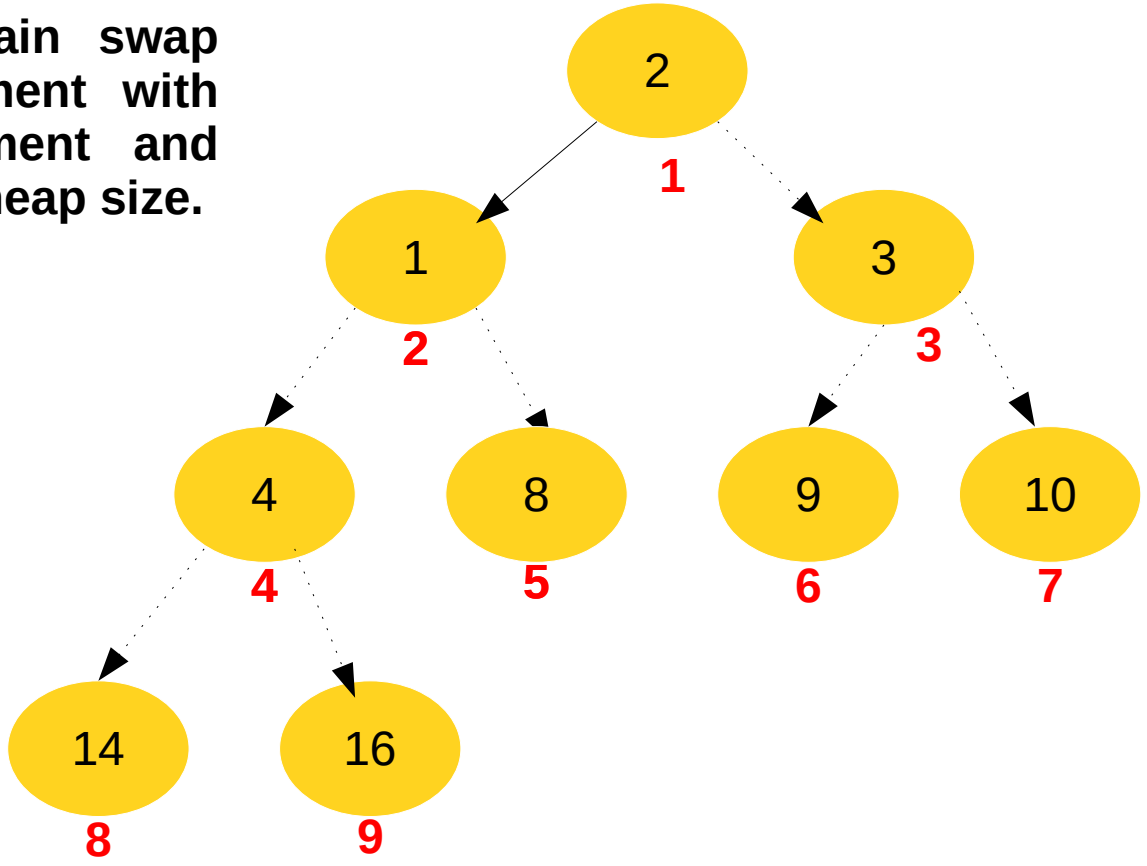


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **Then, we again swap the root element with the last element and decrease the heap size.**

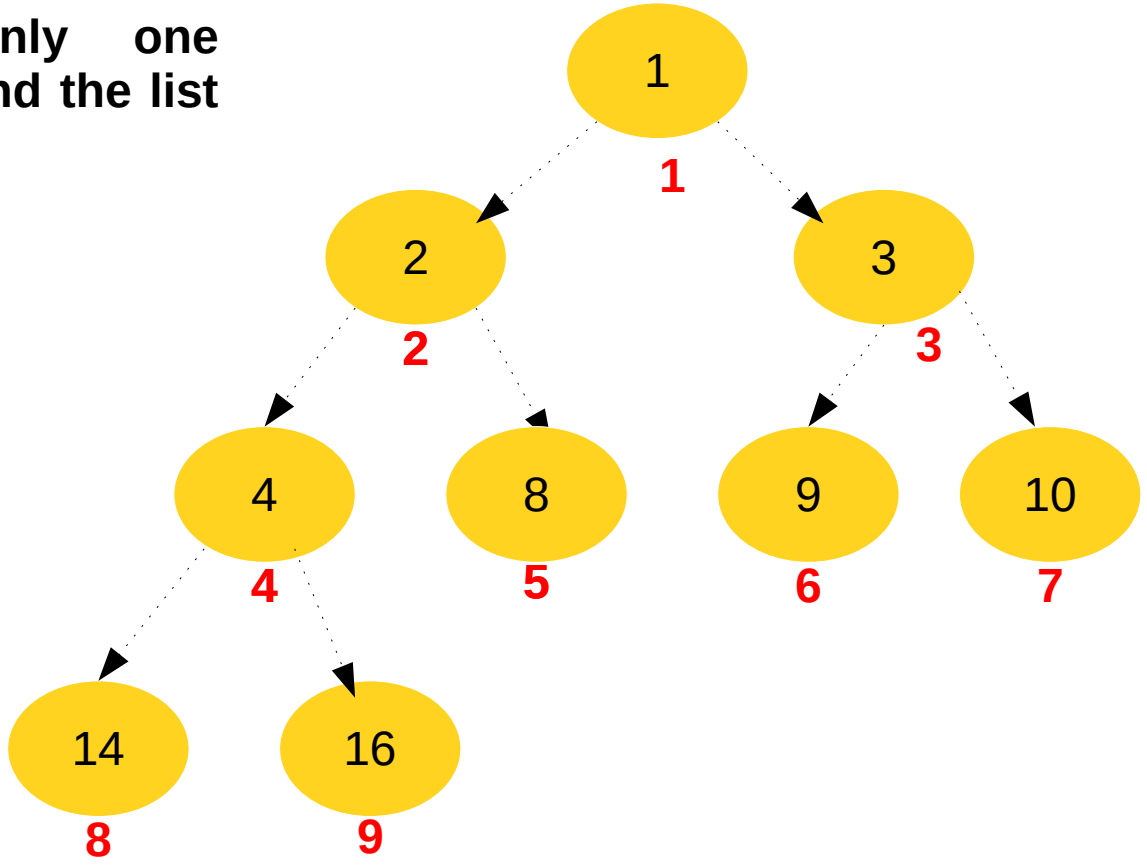


Heap Sort

Example : Sort the set of following elements using Heap Sort

A[ ] = {4, 1, 3, 2, 16, 9, 10, 14, 8}

- **There is only one element left and the list is sorted.**



Heap Sort

Pseudocode

```
HeapSort(A)
{
    BuildHeap(A);
    n = length[A];
    for(i = n; i>=2; i--)
    {
        swap(A[1], A[n]);
        n = n – 1;
        MAX-HEAPIFY(A,1);
    }
}
```

Heap Sort

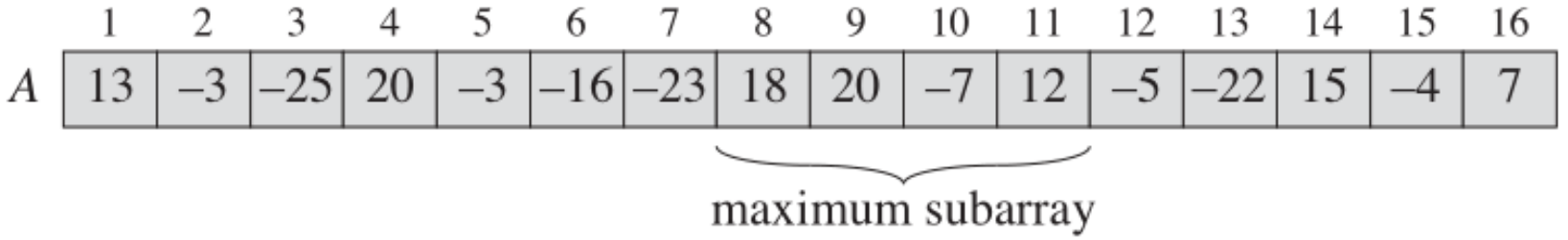
Analysis

```
HeapSort(A)
{
    BUILD-MAX-HEAP(A);
    n = length[A];
    for(i = n; i>=2; i--)
    {
        swap(A[1], A[n]);
        n = n – 1;
        MAX-HEAPIFY(A,1);
    }
}
```

- In the worst case, BUILD-MAX-HEAP takes $O(n \log_2 n)$ time.
- For loop runs for $O(n)$ time.
- Inside for loop MAX-HEAPIFY runs for $O(\log_2 n)$ time.
- Thus, $T(n) = O(n \log_2 n) + O(n)*O(\log_2 n)$
- $T(n) = O(n \log_2 n)$

The Maximum-Subarray Problem

- The aim is to find the nonempty, contiguous subarray of A whose values have the largest sum.
- We call this contiguous subarray the maximum subarray.

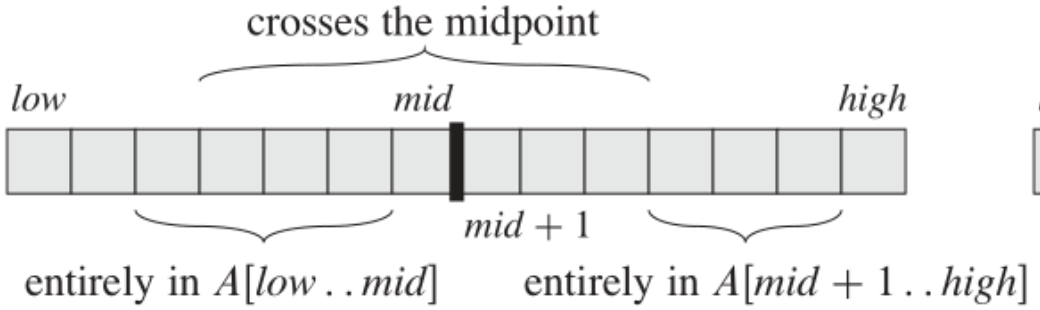


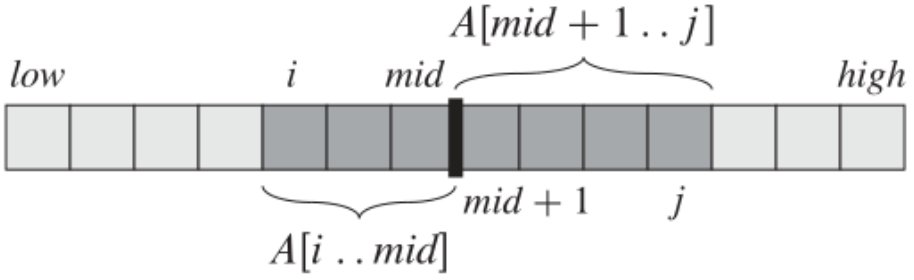
The Maximum-Subarray Problem

- Suppose we want to find a maximum subarray of the subarray A[low...high].
- Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible.
- That is, we find the midpoint, say mid, of the subarray, and consider the subarrays A[low...mid] and A[mid+1... high].

The Maximum-Subarray Problem

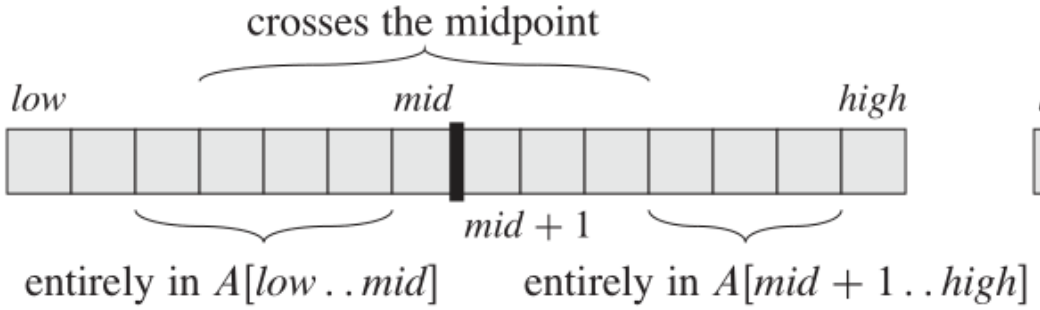
- Any contiguous subarray A[i...j] of A[low...high] must lie in exactly one of the following places:
- entirely in the subarray A[low...mid], so that low $\leq$ i $\leq$ j $\leq$ mid,
- entirely in the subarray A[mid+1...high], so that mid < i $\leq$ j $\leq$ high, or
- crossing the midpoint, so that low $\leq$ i $\leq$ mid < j $\leq$ high.

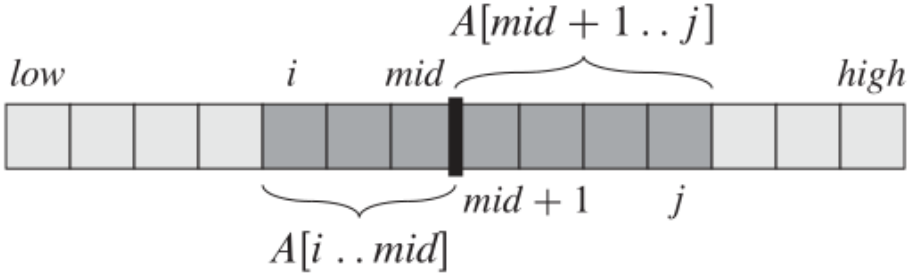




The Maximum-Subarray Problem

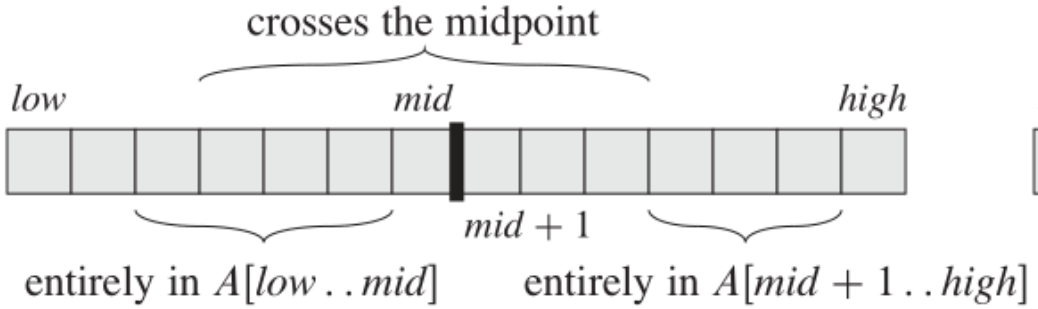
- Therefore, a maximum subarray of A[low...high] must lie in exactly one of these places.
- In fact, a maximum subarray of A[low...high] must have the greatest sum over all subarrays entirely in A[low...mid], entirely in A[mid+1...high] or crossing the midpoint.

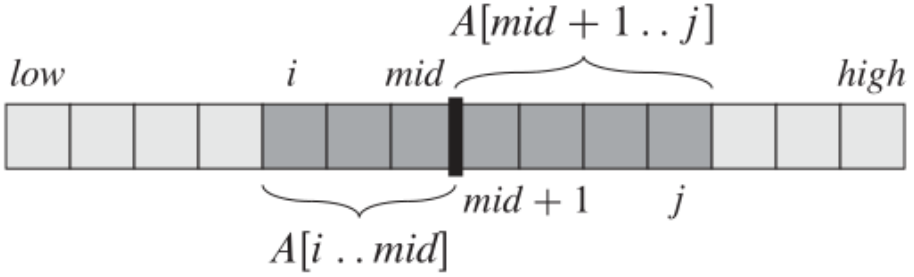




The Maximum-Subarray Problem

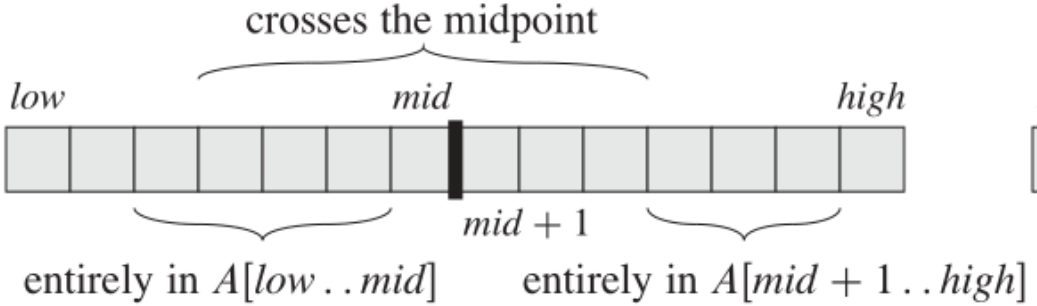
- We can find maximum subarrays of A[low...mid] and A[mid+1...high] recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray.
- Thus, all that is left to do is find a maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

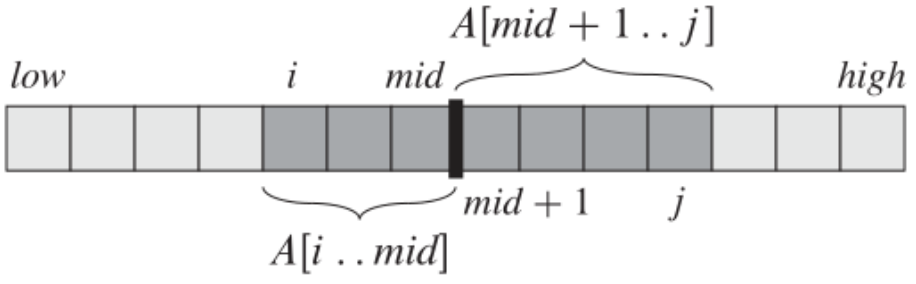




The Maximum-Subarray Problem

- We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray A[low...high].
- As we can see on the figure below any subarray crossing the midpoint is itself made of two subarrays A[i...mid] and A[mid+1...j], where low $\leq$ i $\leq$ mid and mid < j $\leq$ high.
- Therefore, we just need to find maximum subarrays of the form A[i... mid] and A[mid+1...j] and then combine them.





The Maximum-Subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

The Maximum-Subarray Problem

- With a linear-time FIND-MAX-CROSSING-SUBARRAY procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:

The Maximum-Subarray Problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

The Maximum-Subarray Problem

$$T(1) = \Theta(1) .$$ Equation 1

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned}$$ Equation 2

On combining 1 and 2, we get

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

This recurrence is the same as recurrence for merge sort.
As we can see from the master method in , this recurrence has the solution
$T(n) = O(n \log_2 n)$

Matrix Multiplication

- If A = (a_{ij}) and B = (b_{ij}) are square n x n matrices, then in the product C = A x B, we define the entry c_{ij}, for I,j = 1, 2,…. n, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ .$$

Matrix Multiplication

- The following procedure takes n x n matrices A and B and multiplies them, returning their n x n product C. We assume that each matrix has an attribute rows, giving the number of rows in the matrix.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

- **Because each of the triply-nested for loops run exactly n iterations, and each execution of line takes constant time, the SQUARE-MATRIX MULTIPLY procedure takes $\Theta(n^3)$ time.**

Strassen's Algorithm for Matrix Multiplication

- Strassen's remarkable recursive algorithm for multiplying n x n matrices runs in $\Theta(n^{\log_2 7})$ time.
- Since $\log_2 7$ lies between 2.80 and 2.81, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the simple SQUARE-MATRIX-MULTIPLY procedure.

Strassen's Algorithm for Matrix Multiplication

- To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product C = A x B, we assume that n is an exact power of 2 in each of the n x n matrices.
- We make this assumption because in each divide step, we will divide n x n matrices into four n/2 x n/2 matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geq 2$, the dimension n/2 is an integer.

Strassen's Algorithm for Matrix Multiplication

- Suppose that we partition each of A, B, and C into four n/2 x n/2 matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \textbf{Equation 1}$$

so that we rewrite the equation C = A x B as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} . \quad \textbf{Equation 2}$$

Equation 2 can be expanded as

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \quad \textbf{Equation 3}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \quad \textbf{Equation 4}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \quad \textbf{Equation 5}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \quad \textbf{Equation 6}$$

Strassen's Algorithm for Matrix Multiplication

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \quad \textbf{Equation 3}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \quad \textbf{Equation 4}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \quad \textbf{Equation 5}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \quad \textbf{Equation 6}$$

- Each of these four equations specifies two multiplications of n/2 x n/2 matrices and the addition of their n/2 x n/2 products.
- We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm

Strassen's Algorithm for Matrix Multiplication

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

1 $n = A.rows$
2 let C be a new $n \times n$ matrix
3 **if** $n == 1$
4 $\quad c_{11} = a_{11} \cdot b_{11}$
5 **else** partition A, B, and C as in equations (2)
6 $\quad C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
$\quad\quad +$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
7 $\quad C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
$\quad\quad +$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
8 $\quad C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
$\quad\quad +$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
9 $\quad C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
$\quad\quad +$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
10 **return** C

Strassen's Algorithm for Matrix Multiplication

- Let T(n) be the time to multiply two n x n matrices using this procedure. In the base case, when n = 1, we perform just the one scalar multiplication in line 4, and so

$$\mathbf{T(1) = \Theta(1)}$$

Strassen's Algorithm for Matrix Multiplication

- The recursive case occurs when n > 1.
- Partitioning the matrices in line 5 takes Θ(1) time, using index calculations.
- In lines 6–9, we recursively call SQUARE-MATRIX-MULTIPLY-RECURSIVE a total of eight times.
- Because each recursive call multiplies two n/2 x n/2 matrices, thereby contributing T(n/2) to the overall running time, the time taken by all eight recursive calls is 8T(n/2).

Strassen’s Algorithm for Matrix Multiplication

- We also must account for the four matrix additions in lines 6–9.
- Each of these matrices contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time.
- Since the number of matrix additions is a constant, the total time spent adding matrices in lines 6–9 is $\Theta(n^2)$.

Strassen's Algorithm for Matrix Multiplication

- The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls:

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) \ . \end{aligned}$$

- Combining equations for base case and recursive cases

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \ , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \ . \end{cases}$$

Using the master method, we obtain $T(n) = \Theta(n^3)$.

Strassen’s Algorithm for Matrix Multiplication

- The key to Strassen’s method is to make the recursion tree slightly less bushy.
- That is, instead of performing eight recursive multiplications of n/2 x n/2 matrices, it performs only seven.
- The cost of eliminating one matrix multiplication will be several new additions of n/2 x n/2 matrices, but still only a constant number of additions.
- Strassen’s method has four steps:

Strassen's Algorithm for Matrix Multiplication

1. Divide the input matrices A and B and output matrix C into n/2 x n/2 submatrices. This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.
2. Create 10 matrices S_1, S_2, ….. S_{10}, each of which is n/2 x n/2 and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1,P_2, ….. P_7. Each matrix P_i is n/2 x n/2.
4. Compute the desired submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

Strassen's Algorithm for Matrix Multiplication

- In step 2, we create the following 10 matrices

$$
\begin{aligned}
S_1 &= B_{12} - B_{22}\,, \\
S_2 &= A_{11} + A_{12}\,, \\
S_3 &= A_{21} + A_{22}\,, \\
S_4 &= B_{21} - B_{11}\,, \\
S_5 &= A_{11} + A_{22}\,, \\
S_6 &= B_{11} + B_{22}\,, \\
S_7 &= A_{12} - A_{22}\,, \\
S_8 &= B_{21} + B_{22}\,, \\
S_9 &= A_{11} - A_{21}\,, \\
S_{10} &= B_{11} + B_{12}\,.
\end{aligned}
$$

Strassen's Algorithm for Matrix Multiplication

- In step 3, we recursively multiply n/2 x n/2 matrices seven times to compute the following n/2 x n/2 matrices, each of which is the sum or difference of products of A and B submatrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \ , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \ , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \ , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \ , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \ , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \ , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} \ .
\end{aligned}
$$

Note that the only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1

Strassen's Algorithm for Matrix Multiplication

- Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four n/2 x n/2 submatrices of the product C. We start with

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

- Expanding out the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that C_{11} equals

$$\begin{array}{rrrr}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} & & & \\
- A_{22} \cdot B_{11} & + A_{22} \cdot B_{21} & & \\
- A_{11} \cdot B_{22} & & - A_{12} \cdot B_{22} & \\
- A_{22} \cdot B_{22} & - A_{22} \cdot B_{21} & + A_{12} \cdot B_{22} & + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} & & & + A_{12} \cdot B_{21} \ ,
\end{array}$$

Strassen's Algorithm for Matrix Multiplication

- Similarly, we set

$$C_{12} = P_1 + P_2 \ ,$$

- and so C_{12} equals

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad + A_{12} \cdot B_{22} \end{array}$$

Strassen's Algorithm for Matrix Multiplication

- Setting

$$C_{21} = P_3 + P_4$$

makes C21 equal

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \ , \end{array}$$

Strassen's Algorithm for Matrix Multiplication

- Finally, we set

$$C_{22} = P_5 + P_1 - P_3 - P_7 \ ,$$

so that C_{22} equals

$$\begin{array}{llll}
A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} & & & \\
\quad - A_{11}\cdot B_{22} & & + A_{11}\cdot B_{12} & \\
& - A_{22}\cdot B_{11} & & - A_{21}\cdot B_{11} \\
- A_{11}\cdot B_{11} & & - A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} & \\
\hline
& A_{22}\cdot B_{22} & & + A_{21}\cdot B_{12} \ ,
\end{array}$$

Example

Use Strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$.

Example

Use Strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$.

Let A be $\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}$

So, $A_{11} = 1$

$A_{12} = 3$

$A_{21} = 7$

$A_{22} = 5$

Let B be $\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$

So, $B_{11} = 6$

$B_{12} = 8$

$B_{21} = 4$

$B_{22} = 2$

Example

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

$A_{11} = 1$
$A_{12} = 3$
$A_{21} = 7$
$A_{22} = 5$

$B_{11} = 6$
$B_{12} = 8$
$B_{21} = 4$
$B_{22} = 2$

$S_1 = B_{12} - B_{22} = 6$

$S_2 = A_{11} + A_{12} = 4$

$S_3 = A_{21} + A_{22} = 12$

$S_4 = B_{21} - B_{11} = -2$

$S_5 = A_{11} + A_{22} = 6$

$S_6 = B_{11} + B_{22} = 8$

$S_7 = A_{12} - A_{22} = -2$

$S_8 = B_{21} + B_{22} = 6$

$S_9 = A_{11} - A_{21} = -6$

$S_{10} = B_{11} + B_{12} = 14$

Example

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

$P_1 = A_{11}.S_1 = 1*6 = 6$

$P_2 = S_2.B_{22} = 4*2 = 8$

$P_3 = S_3.B_{11} = 12*6 = 72$

$P_4 = A_{22}.S_4 = 5*-2 = -10$

$P_5 = S_5.S_6 = 6*8 = 48$

$P_6 = S_7.S_8 = -2*6 = -12$

$P_7 = S_9.S_{10} = -6*14 = -84$

$A_{11} = 1$
$A_{12} = 3$
$A_{21} = 7$
$A_{22} = 5$

$B_{11} = 6$
$B_{12} = 8$
$B_{21} = 4$
$B_{22} = 2$

$S_1 = 6$
$S_2 = 4$
$S_3 = 12$
$S_4 = -2$
$S_5 = 6$
$S_6 = 8$
$S_7 = -2$
$S_8 = 6$
$S_9 = -6$
$S_{10} = 14$

Example

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

$C_{11} = P_5 + P_4 - P_2 + P_6 = 48 - 10 - 8 - 12 = 18$

$C_{12} = P_1 + P_2 = 14$

$C_{21} = P_3 + P_4 = 72 - 10 = 62$

$C_{22} = P_5 + P_1 - P_3 - P_7 = 48 + 6 - 72 + 84 = 66$

$P_1 = 6$
$P_2 = 8$
$P_3 = 72$
$P_4 = -10$
$P_5 = 48$
$P_6 = -12$
$P_7 = -84$

Strassen's Algorithm for Matrix Multiplication

- ✔ Let us assume that once the matrix size n gets down to 1, we perform a simple scalar multiplication. So,

$$T(n) = \Theta(1), \text{ if } n = 1$$

- ✔ When n > 1, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of n/2 x n/2 matrices.
- ✔ Hence, we obtain the following recurrence for the running time T(n) of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases}$$

Using master's method T (n) = $\Theta(n^{\log_2 7})$

Review Questions

1. Why is solving recurrence relation essential while analyzing algorithms?
2. Write the algorithm for quick sort and compute it's worst case time complexity.
3. Explain and analyze the algorithm for merge sort.
4. What do you mean by a heap? Explain the algorithm for heap sort with an example. Also discuss about it's time and space complexity.
5. Is the array with values {23, 17, 14, 6, 13, 10, 1, 5, 7, 12} a max-heap?
6. Where in a max-heap might the smallest element reside, assuming that all elements are distinct?
7. Illustrate the operation of PARTITION on the array A = {13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11}.
8. Use Strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}\begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$.