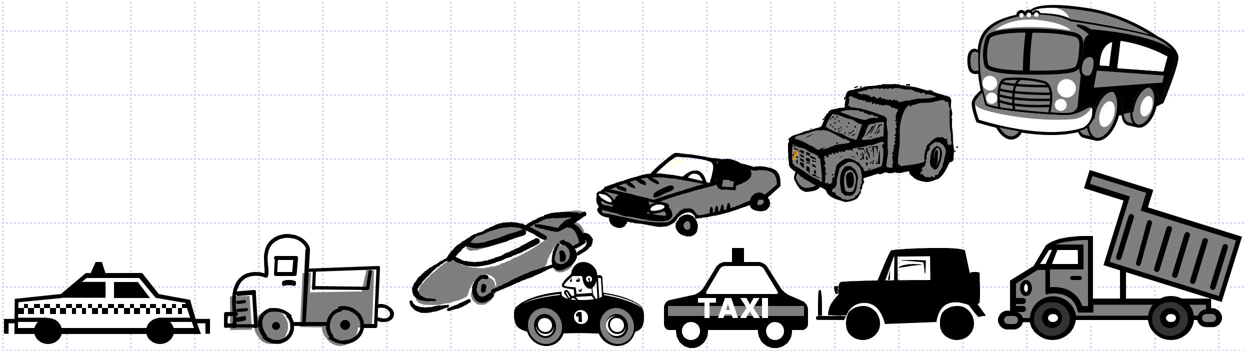
# 합병 정렬

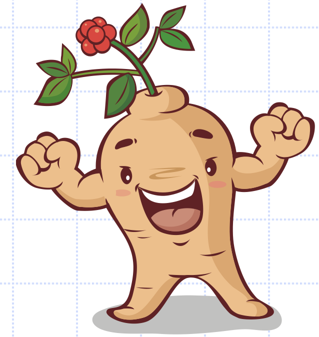


# Outline

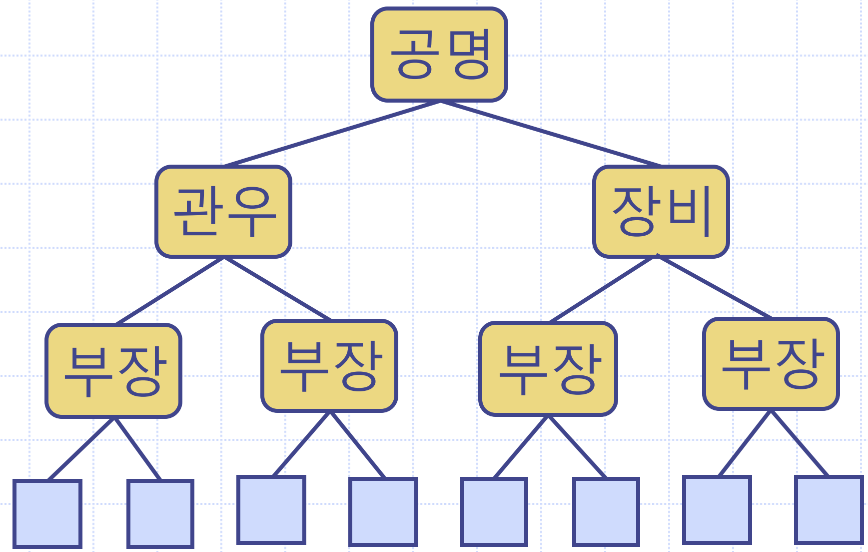
- 7.1 분할통치법
- 7.2 합병 정렬
- 7.3 응용문제

# 분할통치법

- **분할통치법**(divide-and-conquer): 일반적인 알고리즘 설계 기법(algorithm design paradigm)의 일종
  1. **분할**(divide): 입력 데이터 $L$을 둘 이상의 분리된 부분집합 $L_1, L_2, \ldots$ 으로 나눈다
  2. **재귀**(recur): $L_1, L_2, \ldots$ 각각에 대한 부문제를 재귀적으로 해결
  3. **통치**(conquer): 부문제들에 대한 해결을 합쳐 $L$을 해결
- 재귀의 **베이스 케이스:** 상수 크기의 부문제들
- **점화식**(recurrence equations)을 사용하여 분석

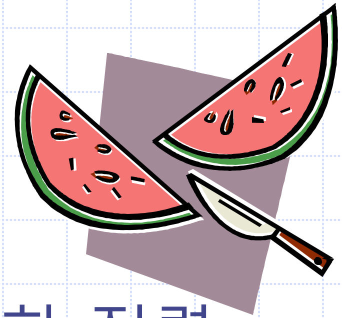


- **예**
  - 합병 정렬(merge-sort)
  - 퀵 정렬(quick-sort)




**산삼채집 작전**

# 합병 정렬



- **합병 정렬**(merge-sort): **분할통치법**에 기초한 정렬 알고리즘
- 힙 정렬(heap-sort)처럼,
  - 비교에 기초한 정렬
  - $\mathbf{O}(n \log n)$ 시간에 수행
- 힙 정렬(heap-sort)과는 달리,
  - 외부의 우선순위 큐를 사용하지 않는다
  - 데이터를 **순차적** 방식으로 접근 (따라서 디스크의 데이터를 정렬하기에 적당)

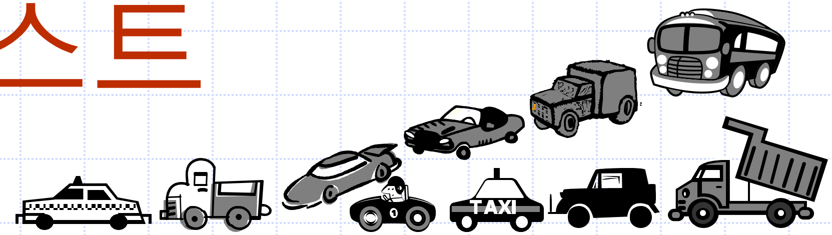
# 합병 정렬 (conti.)

- $n$개의 원소로 이루어진 입력 리스트 $L$에 대한 합병 정렬(merge-sort)의 세 단계
  1. **분할**(divide): 무순리스트 $L$을 각각 $n/2$개의 원소를 가진 두 개의 부리스트 $L_1$과 $L_2$로 분할
  2. **재귀**(recur): $L_1$과 $L_2$를 각각 재귀적으로 정렬
  3. **통치**(conquer): $L_1$과 $L_2$를 단일 순서리스트로 합병

```
Alg mergeSort(L)
    input list L with n elements
    output sorted list L

1. if (L.size() > 1)
       L1, L2 ← partition(L, n/2)
       mergeSort(L1)
       mergeSort(L2)
       L ← merge(L1, L2)
2. return
```

# 두 개의 정렬 리스트 합병하기

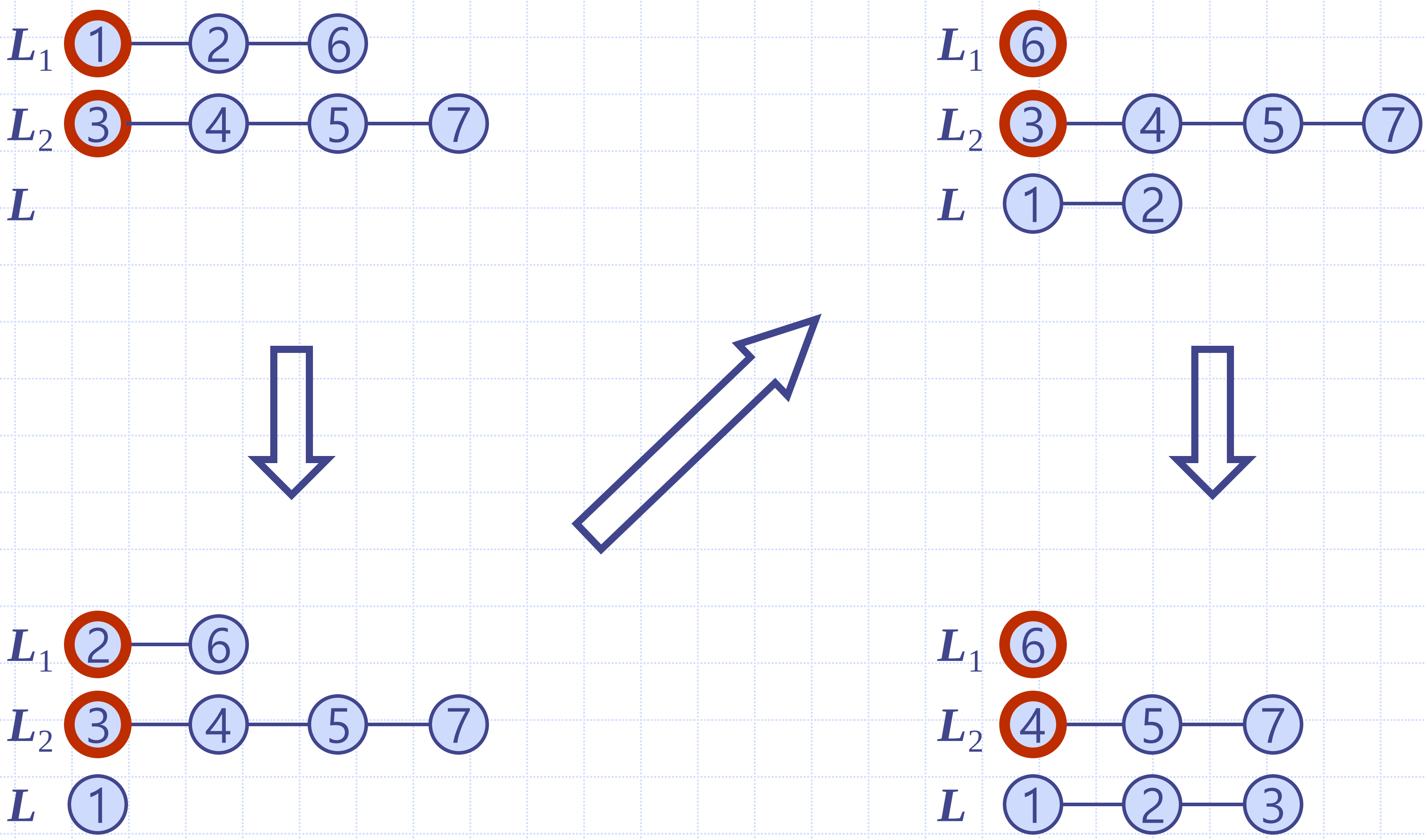


- merge-sort의 **통치** 단계
  - 두 개의 정렬된 리스트 $L_1$과 $L_2$를 $L_1$과 $L_2$의 원소들의 합을 포함하는 정렬 리스트 $L$로 합병하는 과정
- 각각 $n/2$개의 원소를 가지며, 이중연결리스트로 구현된 두 개의 정렬 리스트를 합병하는데 **O($n$)** 시간 소요

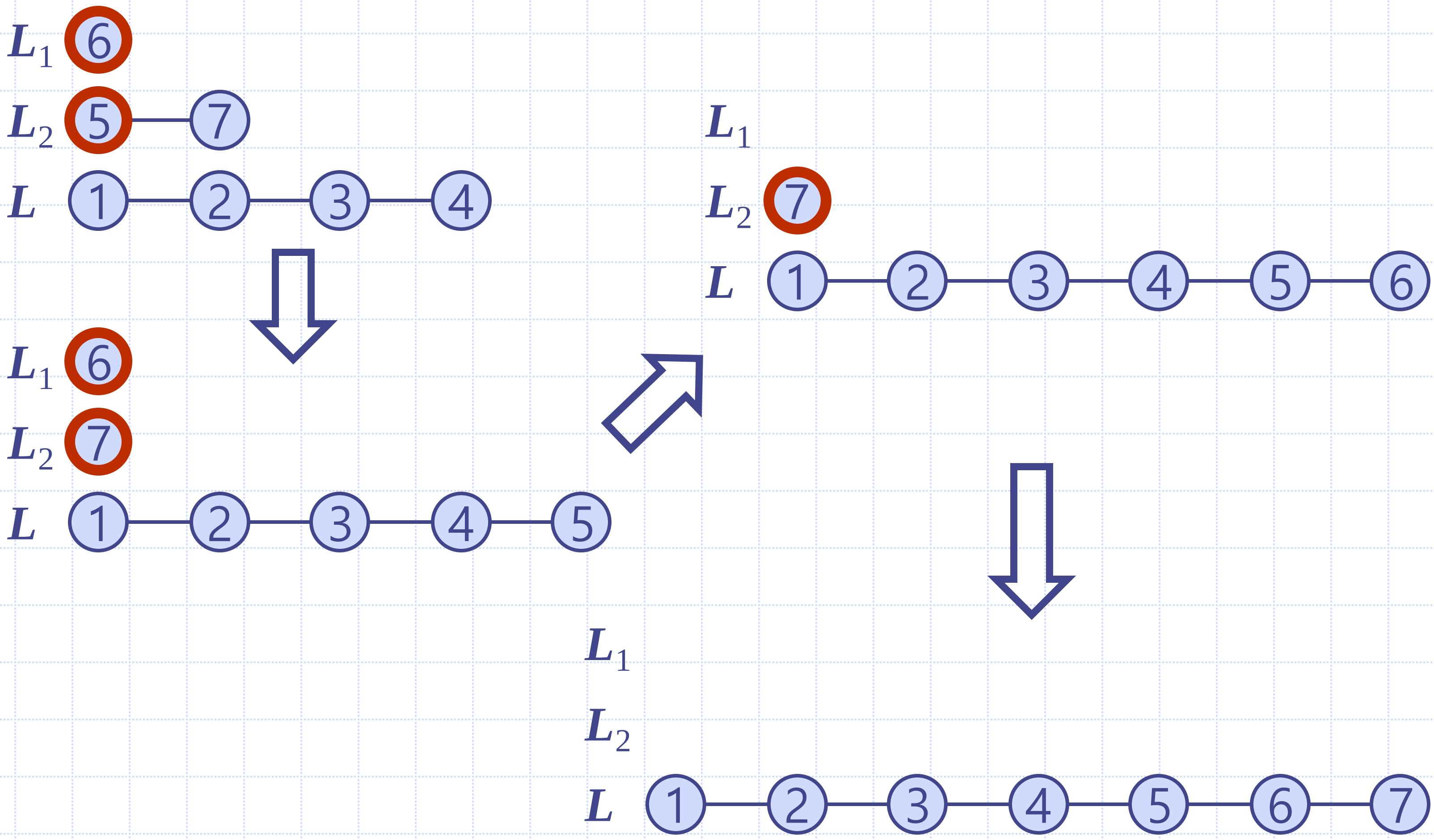
**Alg** ***merge*($L_1$, $L_2$)**
  **input** sorted list $L_1$ and $L_2$ with $n/2$ elements each
  **output** sorted list of $L_1 \cup L_2$

1\. $L \leftarrow$ *empty list*
2\. **while** (!$L_1$.*isEmpty*() & !$L_2$.*isEmpty*())
    **if** ($L_1$.*get*(1) $\leq$ $L_2$.*get*(1))
      $L$.*addLast*($L_1$.*removeFirst*())
    **else**
      $L$.*addLast*($L_2$.*removeFirst*())
3\. **while** (!$L_1$.*isEmpty*())
    $L$.*addLast*($L_1$.*removeFirst*())
4\. **while** (!$L_2$.*isEmpty*())
    $L$.*addLast*($L_2$.*removeFirst*())
5\. **return** $L$

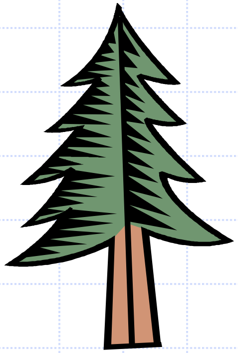
# 합병 예



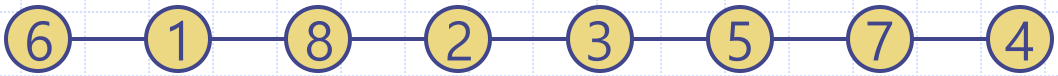
# 합병 예 (conti.)



# 합병정렬 트리

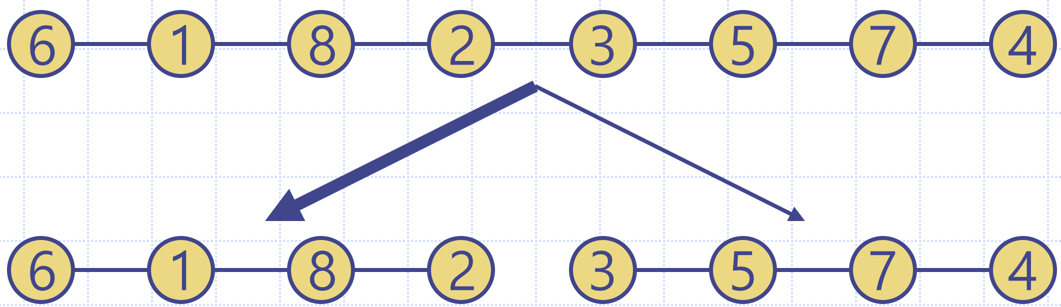


- merge-sort의 실행을 이진트리로 보이기
  - 이진트리의 각 노드는 merge-sort의 **재귀호출**을 표현하며 다음을 저장
    - 실행 이전의 무순리스트 및 분할
    - 실행 이후의 정렬리스트
  - 루트는 **초기 호출**을 의미
  - 잎들은 크기 1의 부리스트에 대한 호출을 의미
- 실행예를 위한 입력 리스트



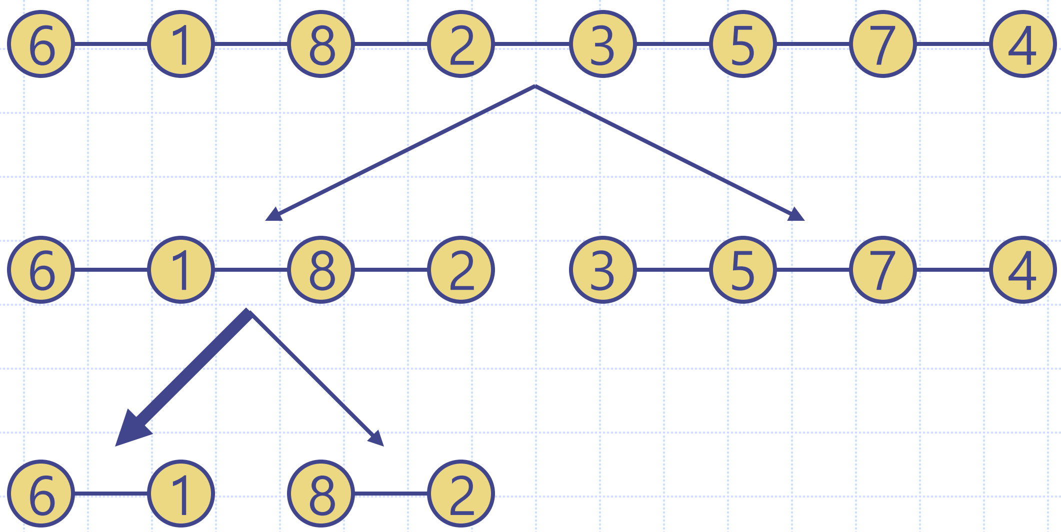
# 합병 정렬 수행 예

- 초기 호출, 분할, 재귀 호출



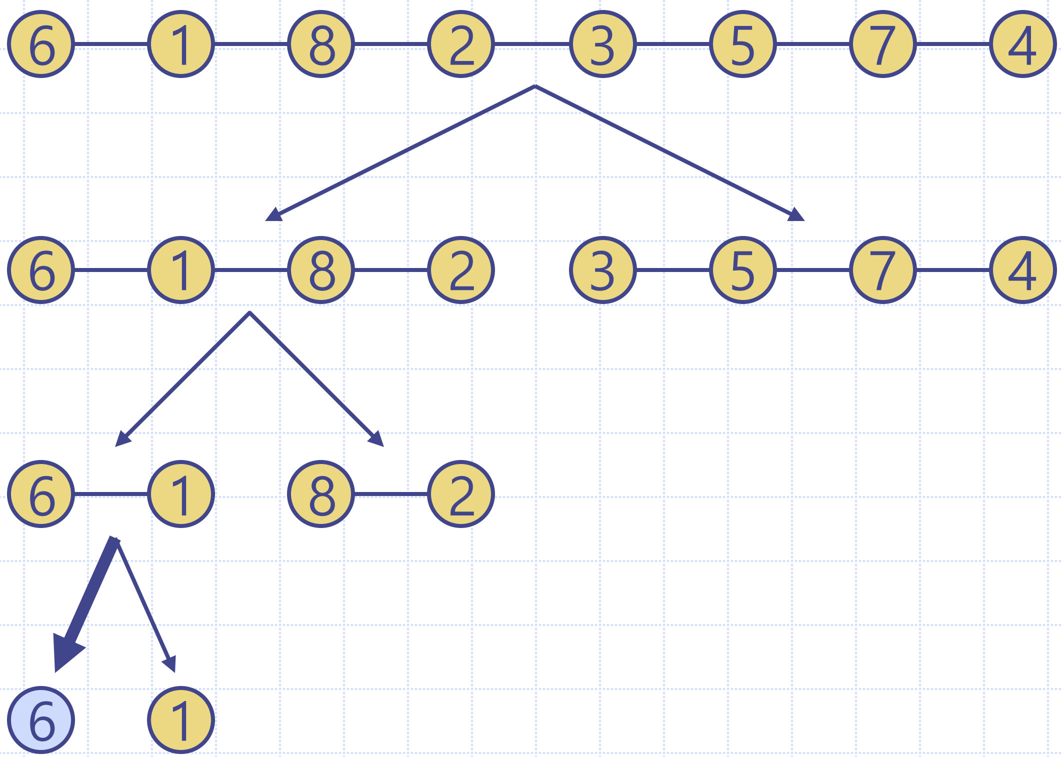
# 합병 정렬 수행 예 (conti.)

- 분할, 재귀호출



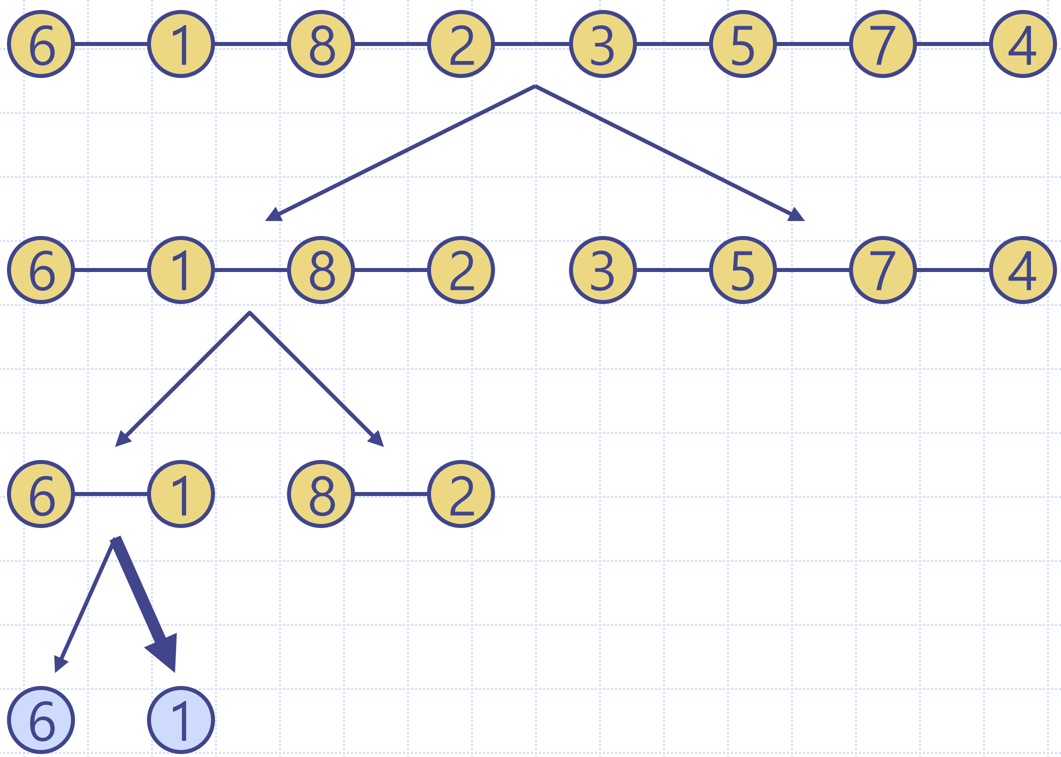
# 합병 정렬 수행 예 (conti.)

- 분할, 재귀 호출, 베이스 케이스



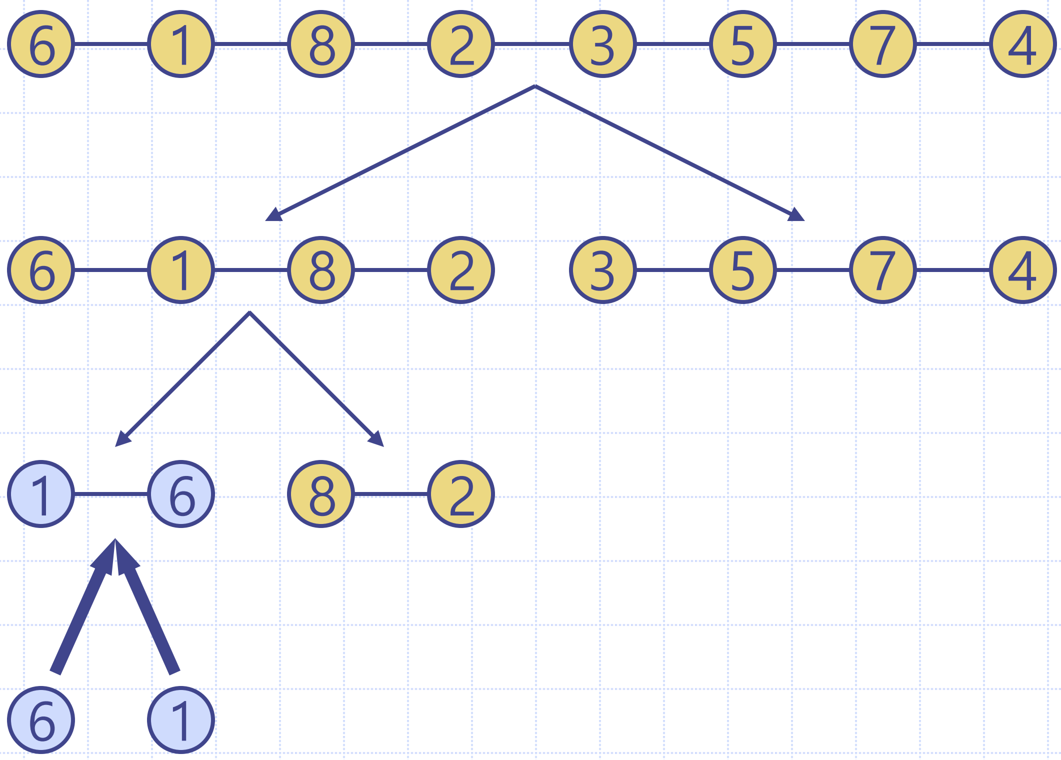
# 합병 정렬 수행 예 (conti.)

- 재귀 호출, 베이스 케이스



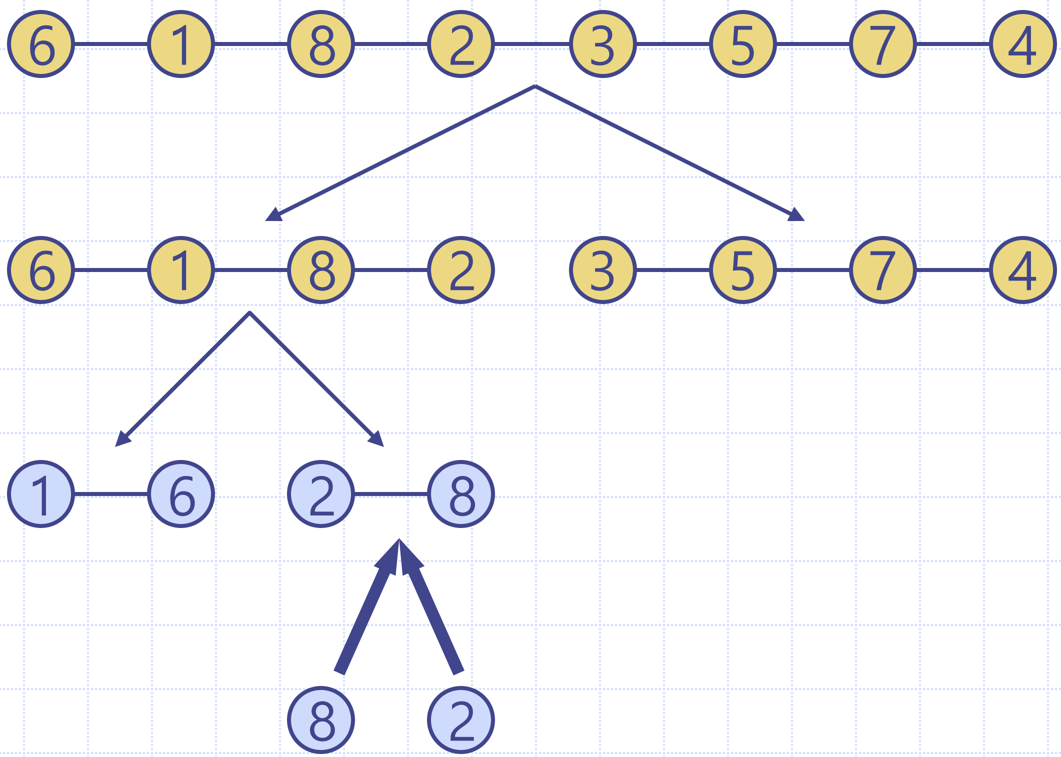
# 합병 정렬 수행 예 (conti.)

- 합병



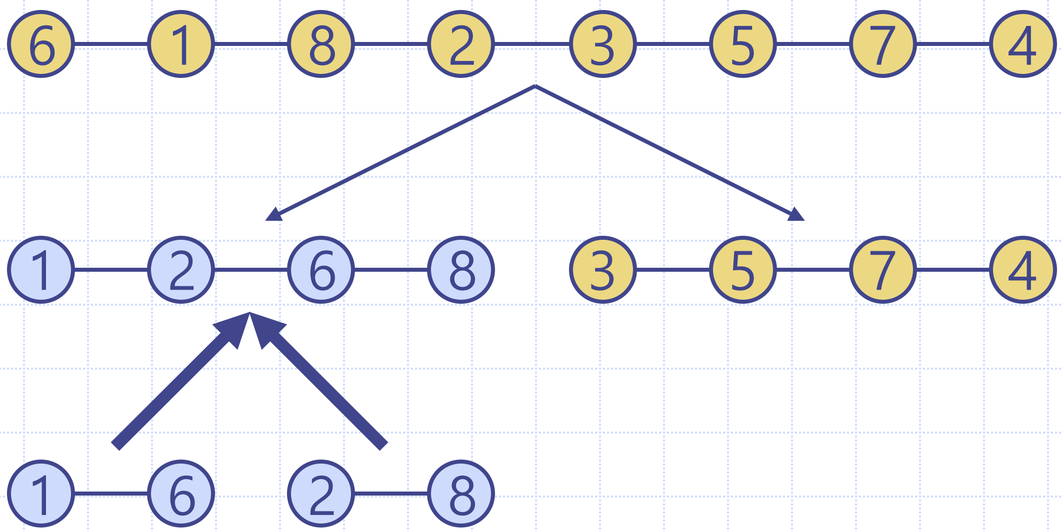
# 합병 정렬 수행 예 (conti.)

- 재귀 호출, … , 합병



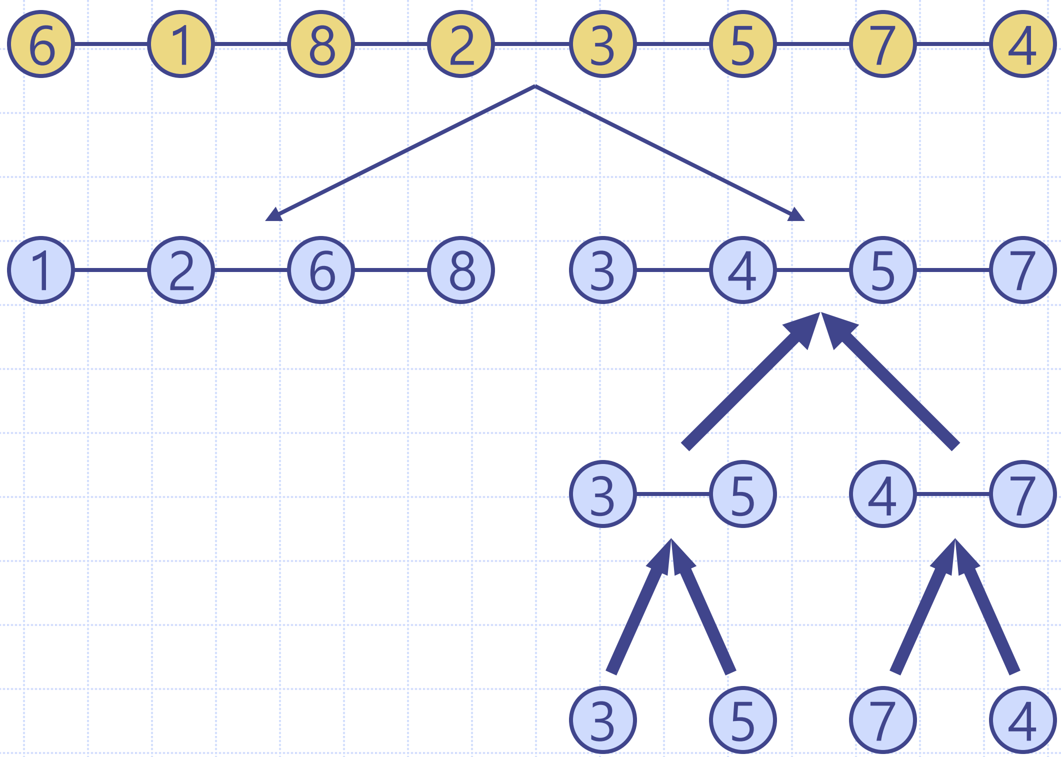
# 합병 정렬 수행 예 (conti.)

- 합병



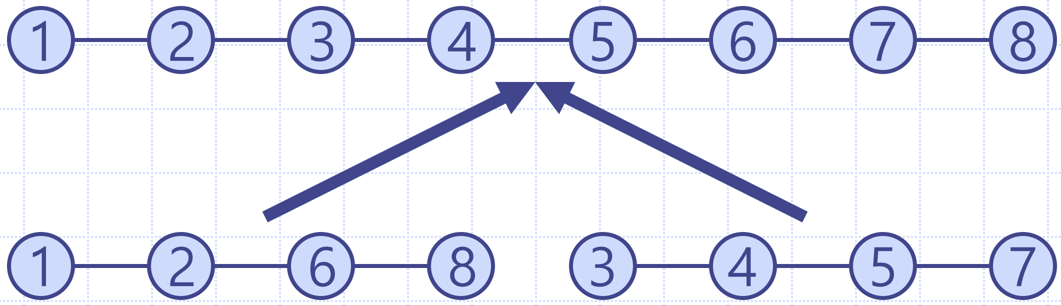
# 합병 정렬 수행 예 (conti.)

- 재귀 호출, … , 합병



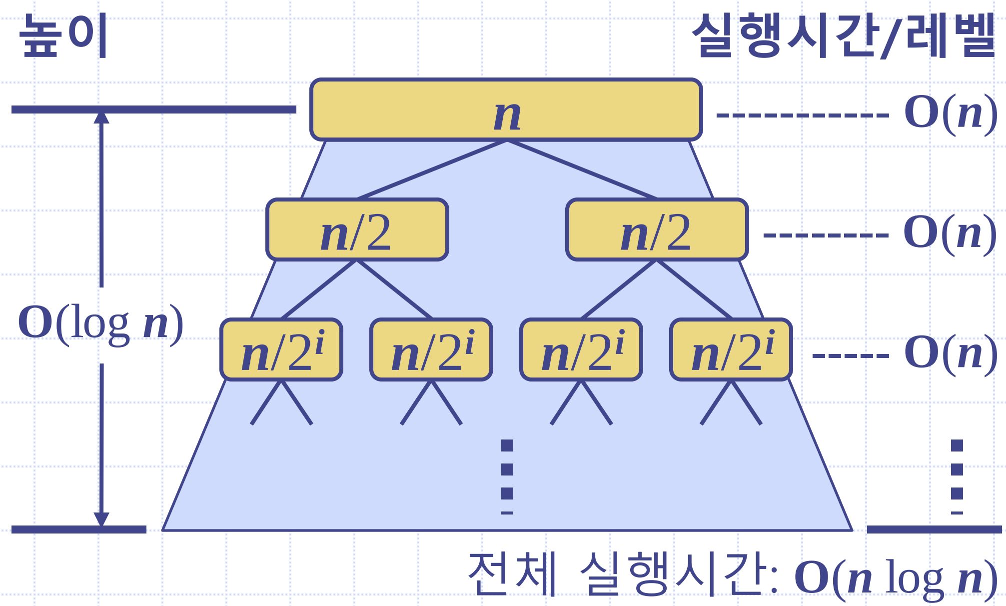
# 합병 정렬 수행 예 (conti.)

◆ 합병



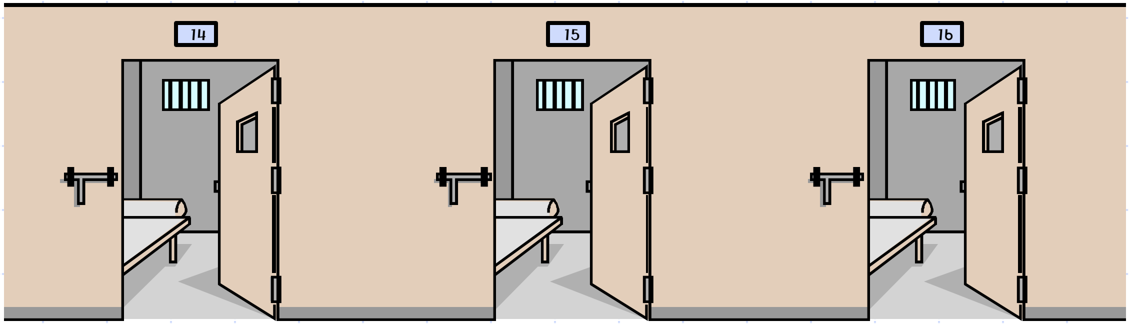
# 합병 정렬 분석

- 합병 정렬 트리의 높이 $h$: $\mathbf{O}(\log n)$
  - 각 재귀호출에서 리스트를 절반으로 나누기 때문
- 깊이 $i$의 노드들에서 이루어지는 총 작업량: $\mathbf{O}(n)$
  - $2^i$개의 크기 $n/2^i$의 리스트들을 분할하고 합병하기 때문
  - $2^{i+1}$번의 재귀호출
- 따라서, merge-sort의 전체 **실행시간: $\mathbf{O}(n \log n)$**
- 점화식으로도 해결 가능
  - $T(n) = c \qquad (n < 2)$
  - $T(n) = 2T(n/2) + \mathbf{O}(n)$
    $(n \geq 2)$
  - $\therefore\ T(n) = \mathbf{O}(n \log n)$



# 응용문제: 배열에 대한 합병 정렬

- 일반 리스트가 아닌, **배열**에 대해 작동하는 merge-sort 알고리즘의 버전을 작성하라
  - mergeSort(A): $\boldsymbol{n}$개의 원소로 구성된 배열 $\boldsymbol{A}$를 합병 정렬
- **힌트:** 외부 배열을 "버퍼", 즉 임시 저장 공간으로 사용하라



# 해결

**Alg** *mergeSort(A)*
    **input** array $A$ of $n$ keys
    **output** sorted array $A$

1. $rMergeSort(A,\ 0,\ n-1)$
2. **return**

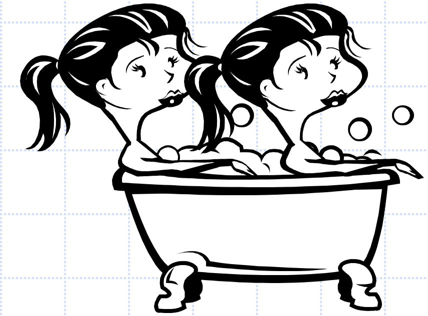
**Alg** *rMergeSort(A, l, r)*
    **input** array $A[l..r]$
    **output** sorted array $A[l..r]$

1. **if** $(l < r)$
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        $rMergeSort(A,\ l,\ m)$
        $rMergeSort(A,\ m+1,\ r)$
        $merge(A,\ l,\ m,\ r)$
2. **return**

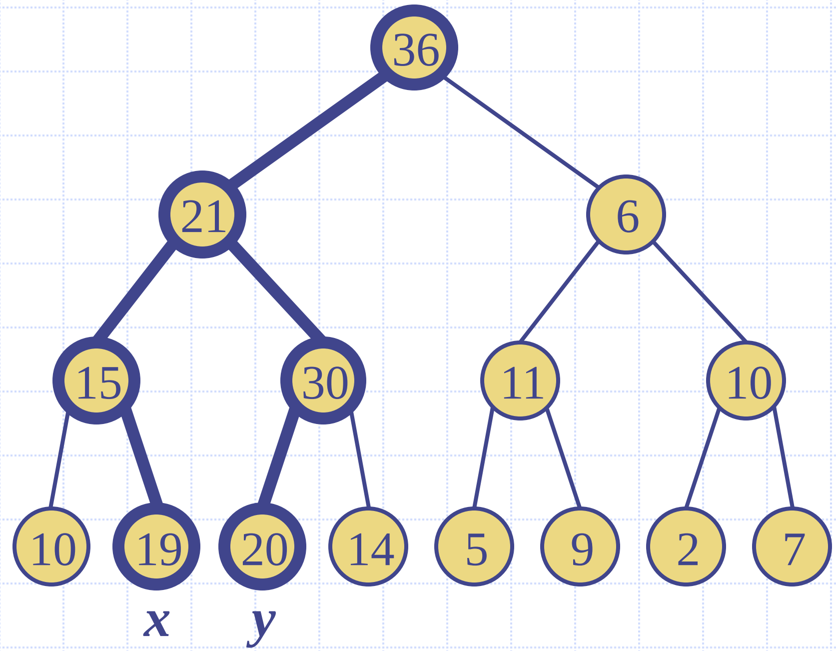
**Alg** *merge(A, l, m, r)*
    **input** sorted array $A[l..m]$, $A[m+1..r]$
    **output** sorted array $A[l..r]$ merged from
        $A[l..m]$ and $A[m+1..r]$

1. $i, k \leftarrow l$
2. $j \leftarrow m+1$
3. **while** $(i \le m \ \&\ j \le r)$
        **if** $(A[i] \le A[j])$
            $B[k{+}{+}] \leftarrow A[i{+}{+}]$
        **else**
            $B[k{+}{+}] \leftarrow A[j{+}{+}]$
4. **while** $(i \le m)$
        $B[k{+}{+}] \leftarrow A[i{+}{+}]$
5. **while** $(j \le r)$
        $B[k{+}{+}] \leftarrow A[j{+}{+}]$
6. **for** $k \leftarrow l$ to $r$
        $A[k] \leftarrow B[k]$
7. **return**

# 응용문제: 포화이진트리



- 높이 $h$의, $n = 2^h$개의 외부노드로 이루어진 **포화이진트리**(full binary tree)가 있다
- 트리의 각 노드 $v$는 임의의 실수값 $value(v) = k$를 저장
- $v$가 외부노드라면, $A(v)$는 $v$의 **조상**들의 집합을 나타낸다 ($v$의 조상에는 $v$ 자신도 포함)
- $a$와 $b$가 서로 다른 외부노드라면, $A(a, b)$는 $a$ 또는 $b$의 조상들의 집합을 나타낸다
  - 즉, $A(a, b) = A(a) \cup A(b)$
- $f(a, b)$를 $A(a, b)$에 포함된 노드 $v$의 $value(v)$ 값의 합이라 정의
- $f(x, y)$를 **최대**로 하는 두 개의 외부노드 $x$와 $y$를 찾는 **효율적인** 알고리즘을 작성하고 **분석**하라




- **예:** $f(x, y)$
  $= 19 + 15 + 21 + 36 + 20 + 30$
  $= 141$
- **힌트: 분할통치법**으로 해결

# 해결: 포화이진트리

- **분할통치법**으로 해결 가능한 문제
- 단순성을 위해, 여기서 제시하는 알고리즘은 최대값을 계산하기만 할 뿐 두 개의 외부노드를 반환하지는 않는다 – 이 값을 주는 두 개의 노드를 반환하도록 수정하는 것은 어렵지 않다
- max1(v)는 노드 $v$를 루트로 하는 부트리 내의 모든 외부노드 $x$의 조상 노드들의 $value(x)$의 합 $f(x)$ 중 최대값을 $O(n)$ 시간에 계산
  - $T_{max1}(n) = c \qquad (n < 2)$
  - $T_{max1}(n) = 2T_{max1}(n/2) + c \qquad (n \geq 2)$
  - $\therefore\ T_{max1}(n) = O(n)$
- max2(v)는 노드 $v$를 루트로 하는 부트리 내의 모든 외부노드 $x, y$ 쌍에 대한 $f(x, y)$ 중 최대값을 $O(n \log n)$ 시간에 계산
  - $T_{max2}(n) = c \qquad (n < 4)$
  - $T_{max2}(n) = 2T_{max2}(n/2) + 2T_{max1}(n/2) + c = 2T_{max2}(n/2) + O(n) \qquad (n \geq 4)$
  - $\therefore\ T_{max2}(n) = O(n \log n)$

# 해결: 포화이진트리 (conti.)

**Alg** *max1*(*v*)
    **input** node ***v***
    **output** maximum value of ***f***(***x***), i.e. the sum of the ancestors of all leaves ***x*** in ***v***'s subree

1. **if** (***isExternal***(*v*))
        **return** ***value***(*v*)
2. **return** ***value***(*v*) + ***max***(***max1***(***leftChild***(*v*)), ***max1***(***rightChild***(*v*)))
    {Total **O**(***n***)}

**Alg** *max2*(*v*)
    **input** node ***v***
    **output** maximum ***f***(***x***, ***y***) over all pairs of leaves ***x***, ***y*** in ***v***'s subree

1. **if** (***isInternal***(*v*) & ***isExternal***(***leftChild***(*v*)) & ***isExternal***(***rightChild***(*v*)))
        **return** ***value***(*v*) + ***value***(***leftChild***(*v*)) + ***value***(***rightChild***(*v*))
2. **return** ***value***(*v*) + ***max***(***max2***(***leftChild***(*v*)),
        ***max2***(***rightChild***(*v*)),
        ***max1***(***leftChild***(*v*)) + ***max1***(***rightChild***(*v*)))
    {Total **O**(***n*** log ***n***)}