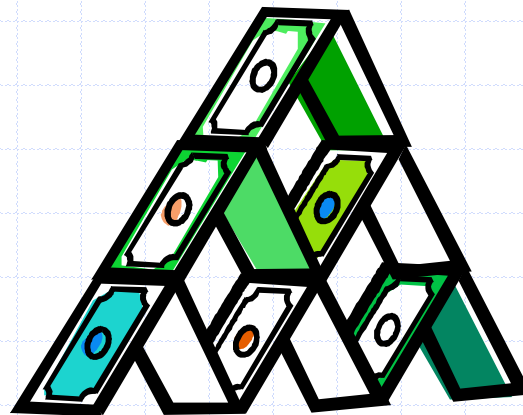


ힵ과 ힵ 정렬



Outline

- ◆ 6.1 힙
- ◆ 6.2 힙을 이용한 우선순위 큐 구현
- ◆ 6.3 힙 구현과 성능
- ◆ 6.4 힙 정렬
- ◆ 6.5 제자리 힙 정렬
- ◆ 6.6 상향식 힙생성
- ◆ 6.7 응용문제

힉

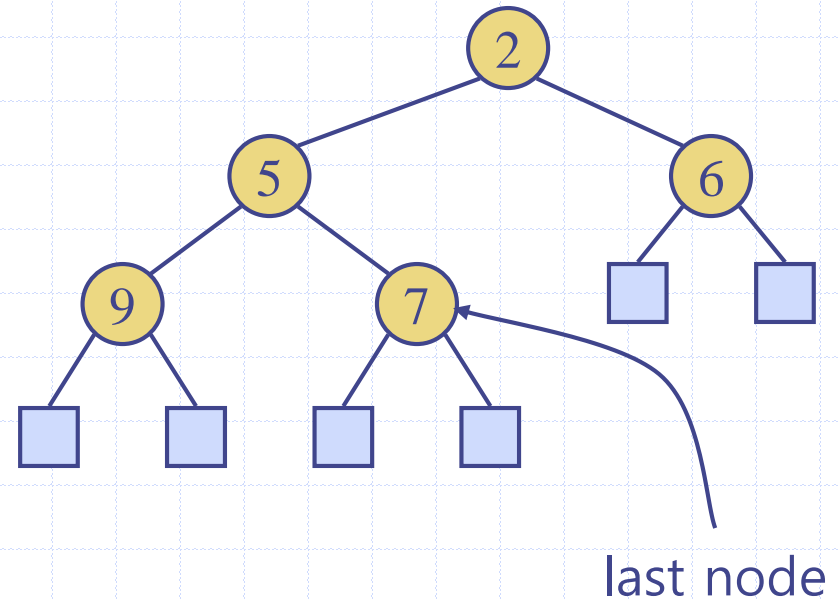


◆ **힉(heap):** 내부노드에 키를 저장하며 다음 두 가지 속성을 만족하는 이진트리

- **힉순서(heap-order):** 루트를 제외한 모든 내부노드 v 에 대해, $key(v) \geq key(parent(v))$
- **완전이진트리(complete binary tree):** 힉의 높이를 h 라 하면

- ◆ $i = 0, \dots, h - 1$ 에 대해, 깊이 i 인 노드가 2^i 개 존재
- ◆ 깊이 $h - 1$ 에서, 내부노드들은 외부노드들의 왼쪽에 존재

◆ **힉의 마지막 노드(last node):** 깊이 $h - 1$ 의 가장 오른쪽 내부노드



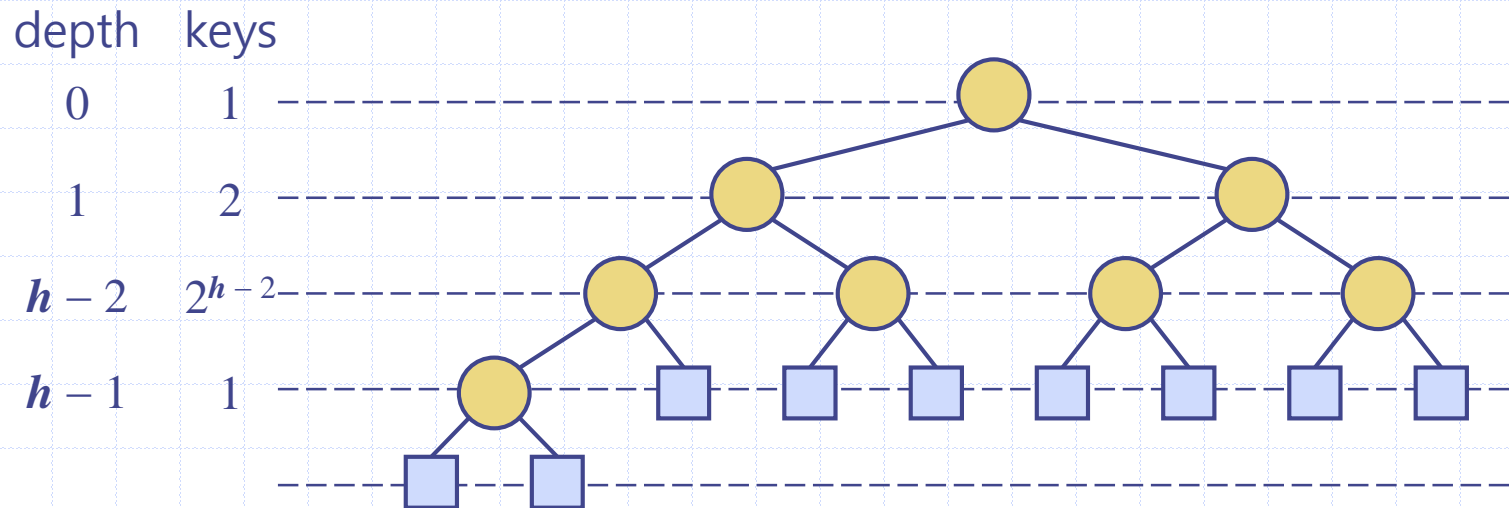
힅의 높이



◆ 정리: n 개의 키를 저장한 힅의 높이는 $O(\log n)$ 이다

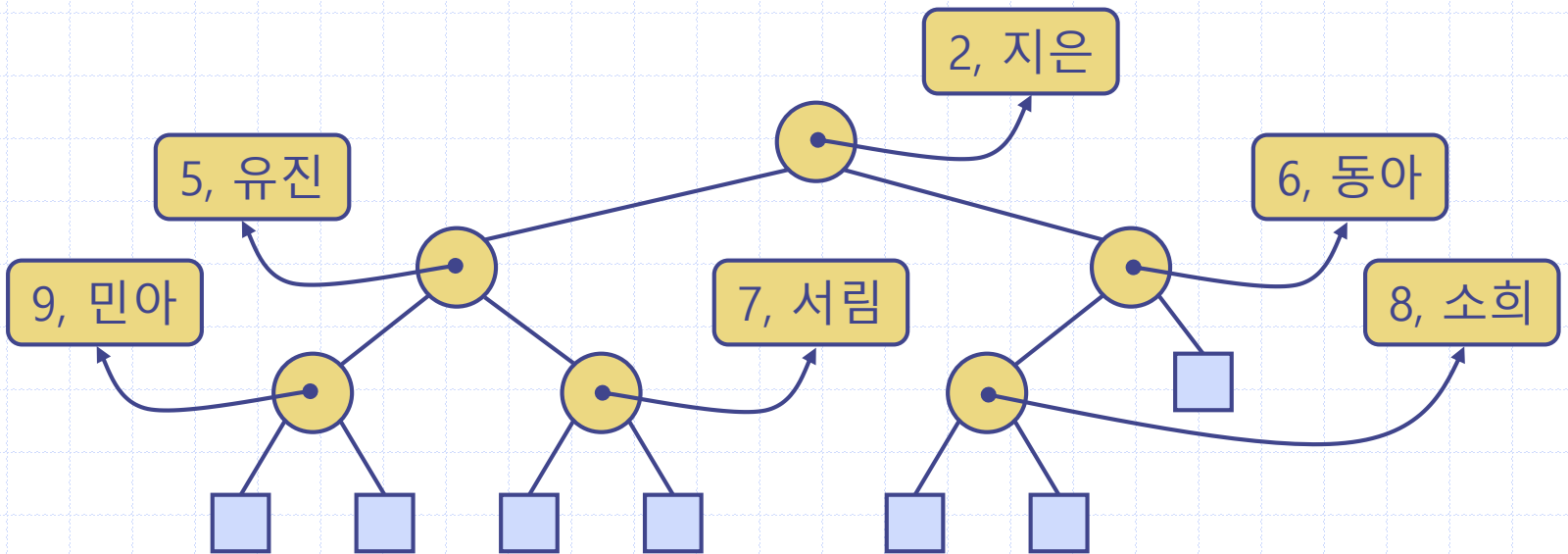
증명: (완전이진트리의 성질을 이용)

- n 개의 키를 저장한 힅의 높이를 h 라 하자
- 깊이 $i = 0, \dots, h-2$ 에 2^i 개의 키, 그리고 깊이 $h-1$ 에 적어도 한 개의 키가 존재하므로, $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$
- 따라서, $n \geq 2^{h-1}$, 즉 $h \leq \log n + 1$



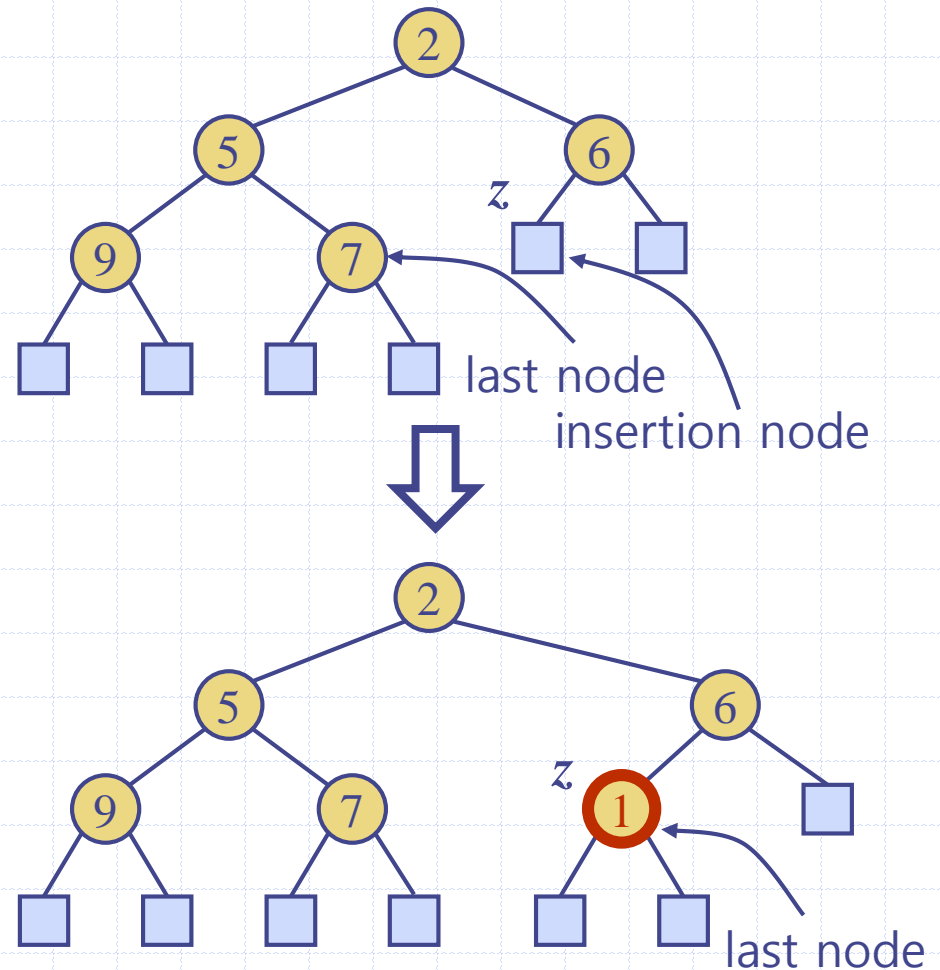
힙과 우선순위 큐

- ◆ 힙을 사용하여 우선순위 큐(priority queue) 구현 가능
- ◆ 전제: 적정이진트리로 구현
- ◆ 마지막 노드의 위치를 관리
- ◆ 그림 표기: 내부노드 내에 간단히 키만 표시



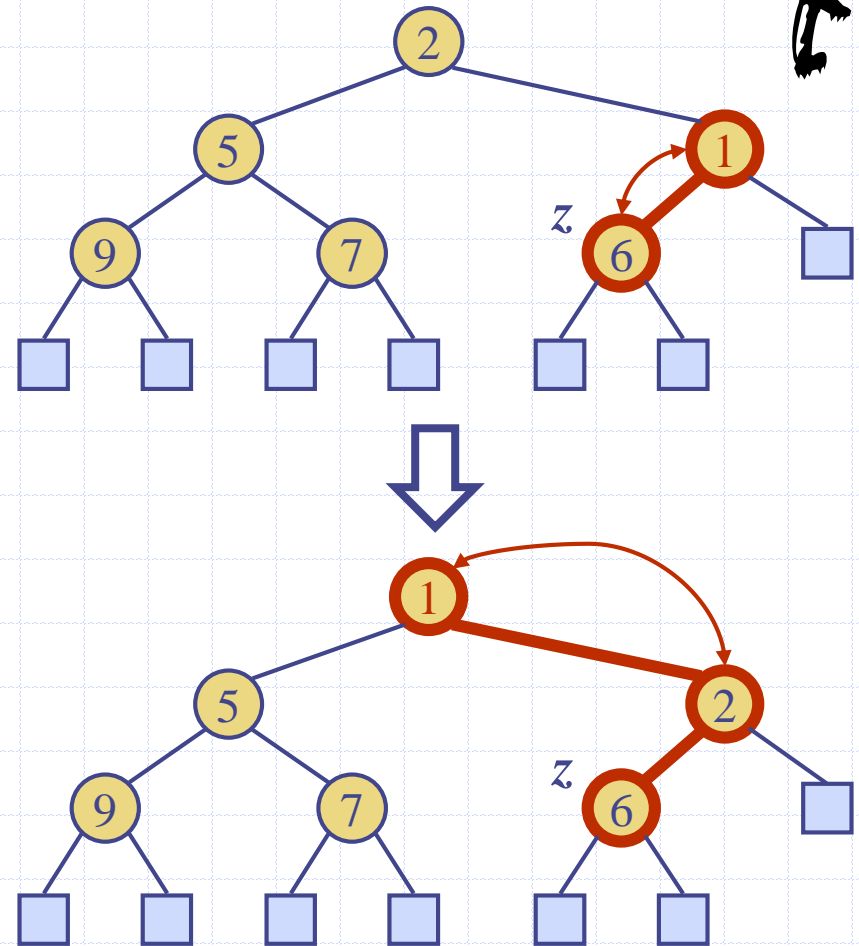
ힵ에 삽입

- ◆ 우선순위 큐 ADT의 메소드 **insertItem**은 ힵ에 키 k 를 삽입하는 것에 해당
- ◆ 삽입 알고리즘의 세 단계
 1. 삽입 노드 z , 즉 새로운 마지막 노드를 찾는다
 2. k 를 z 에 저장한 후 **expandExternal(w)** 작업을 사용하여 z 을 내부노드로 확장
 3. ힵ순서 속성을 복구



Upheap

- ◆ 새로운 키 k 가 삽입된 후, **힅순서 속성**이 위배될 수 있다
- ◆ 알고리즘 **upheap**은 삽입노드로부터 상향 경로를 따라가며 키 k 를 교환함으로써 힅순서 속성을 복구
- ◆ **upheap**은 키 k 가 루트에, 또는 부모의 키가 k 보다 작거나 같은 노드에 도달하면 정지
- ◆ 힅의 높이는 $O(\log n)$ 이므로 **upheap**은 $O(\log n)$ 시간에 수행



힉 삽입 알고리즘

Alg *insertItem*(*k*)

input key *k*, node *last*

output none

1. *advanceLast*()
2. $z \leftarrow last$
3. Set node *z* to *k*
4. *expandExternal*(*z*)
5. *upHeap*(*z*)
6. **return**

Alg *upHeap*(*v*)

input node *v*

output none

1. **if** (*isRoot*(*v*))
 return
2. **if** ($key(v) \geq key(parent(v))$)
 return
3. *swapElements*(*v*, *parent*(*v*))
4. *upHeap*(*parent*(*v*))

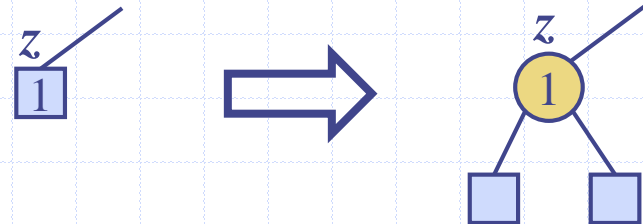
◆ 알고리즘 *insertItem*과 *upHeap*의 작업 대상 힉은 일반(generic) 힉

삽입 (conti.)

Alg *expandExternal*(*z*) {linked}
 input external node *z*
 output none

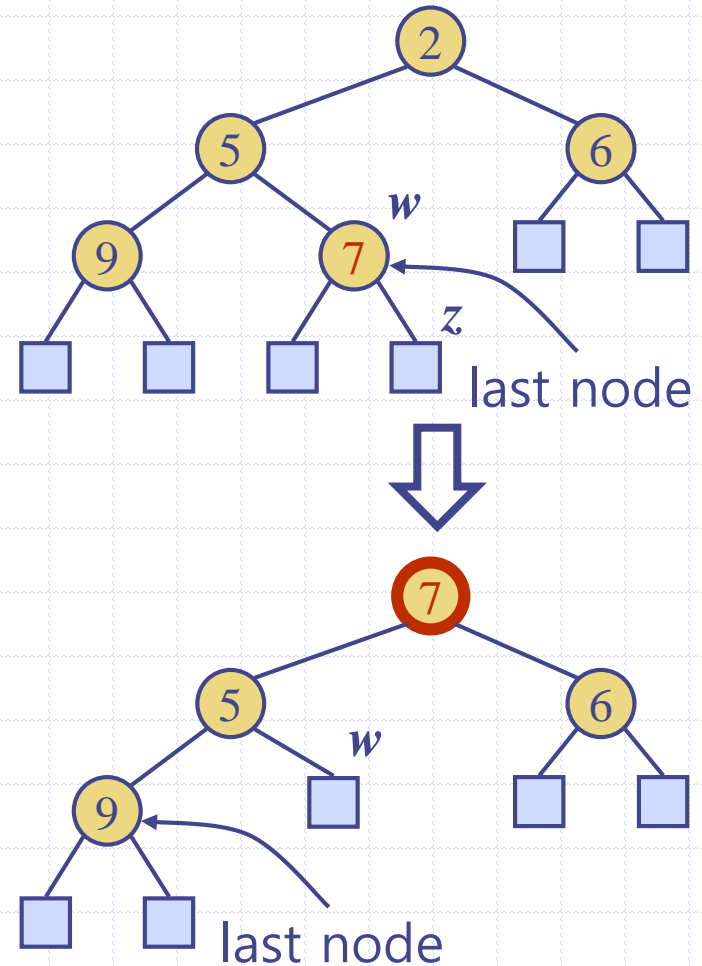
```
1. l ← getnode()  
2. r ← getnode()  
3. l.left ← ∅  
4. l.right ← ∅  
5. l.parent ← z  
6. r.left ← ∅  
7. r.right ← ∅  
8. r.parent ← z  
9. z.left ← l  
10. z.right ← r  
11. return
```

◆ 예: *expandExternal*(*z*)



힉으로부터 삭제

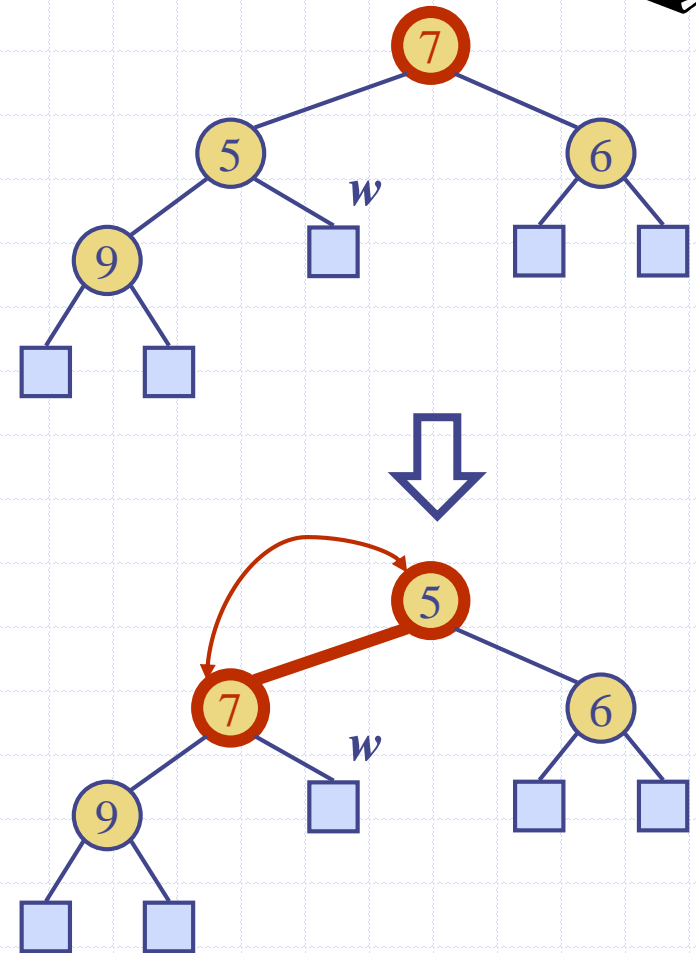
- ◆ 우선순위 큐 ADT의 메소드 **removeMin**은 힉으로부터 루트 키를 삭제하는 것에 해당
- ◆ 삭제 알고리즘의 세 단계
 1. 루트 키를 마지막 노드 w 의 키로 대체
 2. **reduceExternal**(z) 작업을 사용하여 w 와 그의 자식들을 외부노드로 축소
 3. 힉순서 속성을 복구





Downheap

- ◆ 루트 키를 마지막 노드의 키로 대체한 후, **힉순서 속성**이 위배될 수 있다
- ◆ 알고리즘 **downheap**은 루트로부터 하향 경로를 따라가며 키 k 를 교환함으로써 힉순서 속성을 복구
- ◆ **downheap**은 키 k 가 앞에, 또는 자식의 키가 k 보다 크거나 같은 노드에 도달하면 정지
- ◆ 힉의 높이는 $O(\log n)$ 이므로 **downheap**은 $O(\log n)$ 시간에 수행



힙 삭제 알고리즘

Alg *removeMin()*

input node *last*

output key

1. $k \leftarrow \text{key}(\text{root}())$
2. $w \leftarrow \text{last}$
3. Set root to $\text{key}(w)$
4. *retreatLast()*
5. $z \leftarrow \text{rightChild}(w)$
6. *reduceExternal*(z)
7. *downHeap*($\text{root}()$)
8. return k

Alg *downHeap*(v)

input node v whose left and right subtrees
are heaps

output a heap with root v

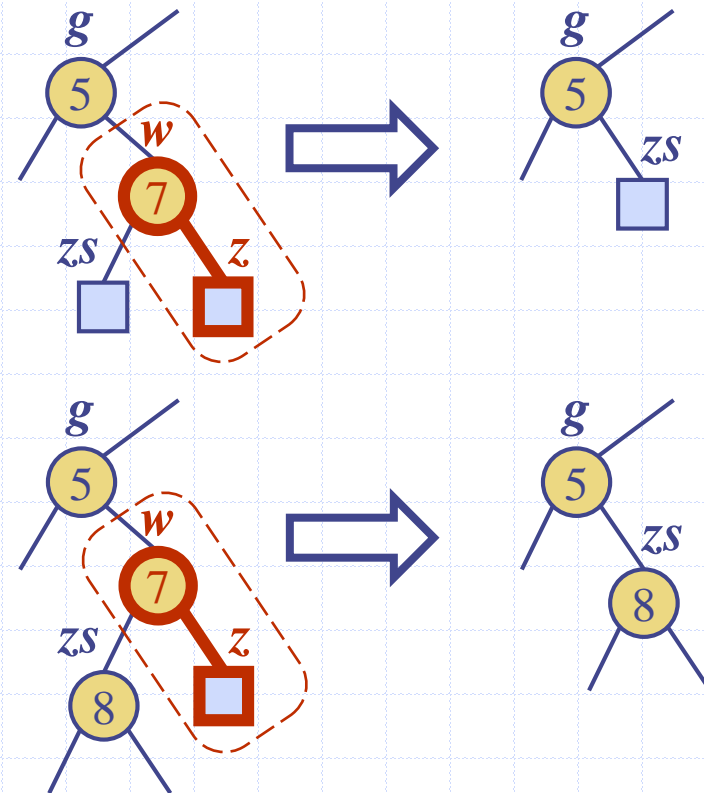
1. **if** (*isExternal*(*leftChild*(v)) &
isExternal(*rightChild*(v)))
return
2. $\text{smaller} \leftarrow \text{leftChild}(v)$ {internal node}
3. **if** (*isInternal*(*rightChild*(v)))
if ($\text{key}(\text{rightChild}(v)) < \text{key}(\text{smaller})$)
 $\text{smaller} \leftarrow \text{rightChild}(v)$
4. **if** ($\text{key}(v) \leq \text{key}(\text{smaller})$)
return
5. *swapElements*($v, \text{smaller}$)
6. *downHeap*(smaller)

삭제 (conti.)

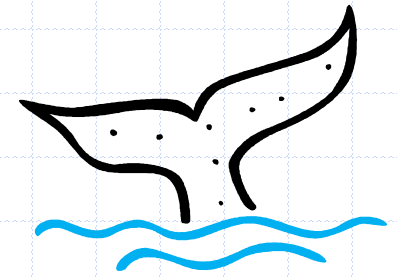
Alg *reduceExternal*(*z*) {linked}
input external node *z*
output the node replacing the parent
 node of the removed node *z*

1. $w \leftarrow z.\text{parent}$
2. $zs \leftarrow \text{sibling}(z)$
3. **if** (*isRoot*(*w*))
 $\text{root} \leftarrow zs$ {renew root}
 $zs.\text{parent} \leftarrow \emptyset$
 else
 $g \leftarrow w.\text{parent}$
 $zs.\text{parent} \leftarrow g$
 if ($w = g.\text{left}$)
 $g.\text{left} \leftarrow zs$
 else { $w = g.\text{right}$ }
 $g.\text{right} \leftarrow zs$
4. *putnode*(*z*) {deallocate node *z*}
5. *putnode*(*w*) {deallocate node *w*}
6. **return** *zs*

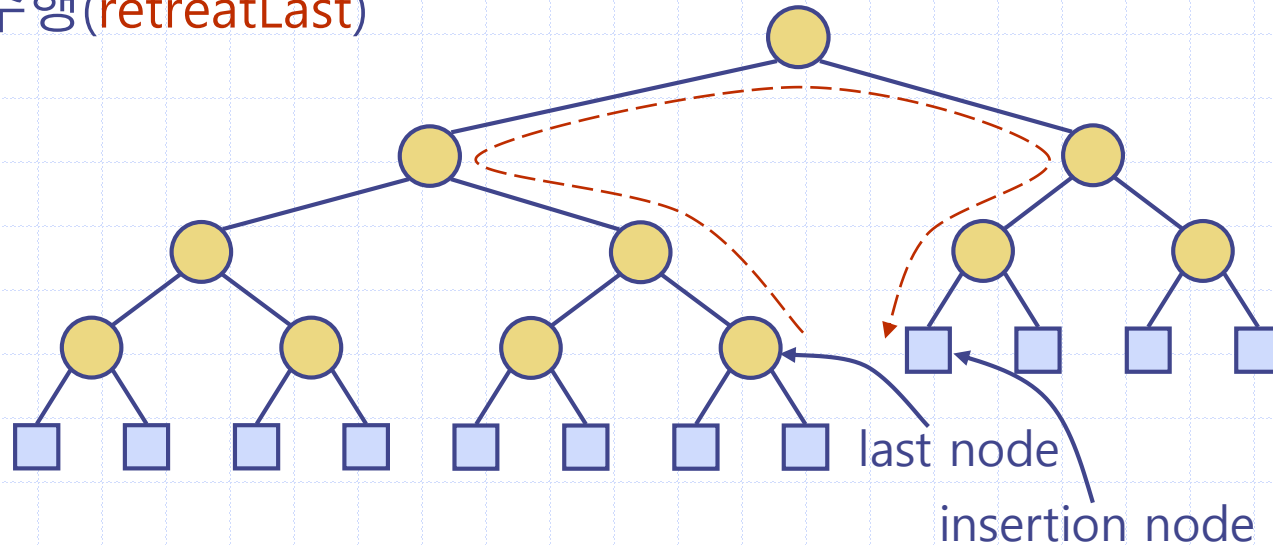
◆ 예: *reduceExternal*(*z*)



마지막 노드 갱신

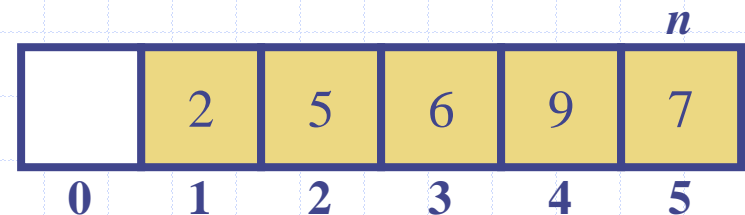
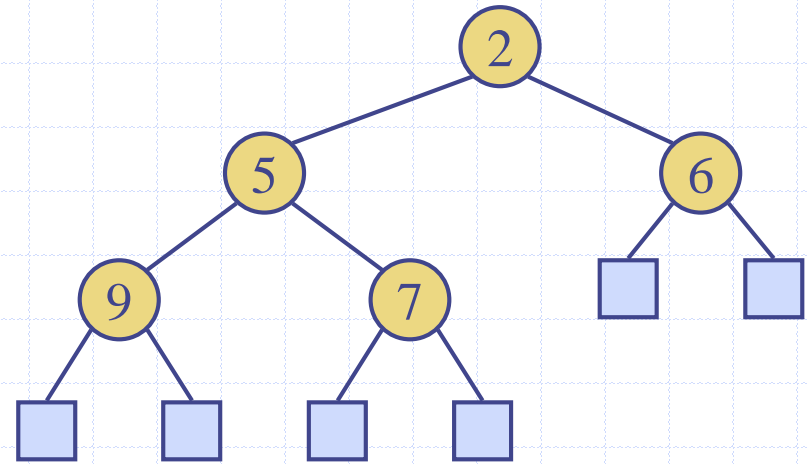


- ◆ $O(\log n)$ 개의 노드를 순회함으로써 **삽입** 노드를 찾을 수 있다(**advanceLast**)
 - 현재 노드가 오른쪽 자식인 동안, 부모 노드로 이동
 - 현재 노드가 왼쪽 자식이면, 형제 노드로 이동
 - 현재 노드가 내부노드인 동안, 왼쪽 자식으로 이동
- ◆ **삭제** 후 마지막 노드를 갱신하는 작업은 위와 반대 방향으로 수행(**retreatLast**)



배열에 기초한 힙 구현

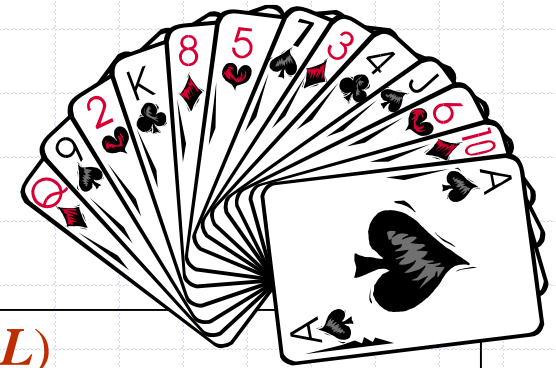
- ◆ n 개의 키를 가진 힙을 크기 n 의 배열을 사용하여 표현 가능
- ◆ 첨자 i 에 존재하는 노드에 대해
 - 왼쪽 자식은 첨자 $2i$ 에 존재
 - 오른쪽 자식은 첨자 $2i + 1$ 에 존재
 - 부모는 첨자 $i/2$ 에 존재
- ◆ 노드 사이의 링크는 명시적으로 저장할 필요가 없다
- ◆ 외부노드들은 표현할 필요 없다
- ◆ 첨자 0 셀은 사용하지 않는다
- ◆ 마지막 노드의 첨자: 항상 n
 - **insertItem** 작업은 첨자 $n + 1$ 위치에 삽입하는 것에 해당
 - **removeMin** 작업은 첨자 n 위치에서 삭제하는 것에 해당



성능 요약

힙 구현	작업				공간 소요
	size, isEmpty	insertItem, removeMin	minKey, minElement	advanceLast, retreatLast	
연결	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(n)$
순차	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(n)$

힙 정렬



- ◆ 힙을 사용하여 구현된 n -항목 우선순위 큐를 고려하면,
 - 공간 사용량은 $O(n)$
 - `insertItem` 메소드와 `removeMin` 메소드는 $O(\log n)$ 시간에 수행
 - `size`, `isEmpty`, `minKey`, `minElement` 메소드는 $O(1)$ 시간에 수행
- ◆ 힙에 기초한 우선순위 큐를 사용함으로써, n 개의 원소로 이루어진 리스트를 $O(n \log n)$ 시간에 정렬할 수 있다
 - 선택 정렬이나 삽입 정렬과 같은 2차 정렬 알고리즘보다 훨씬 빠르다

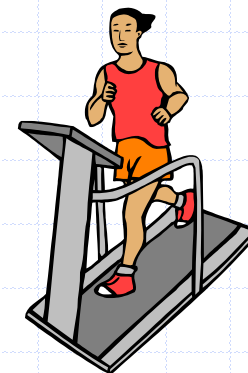
Alg **heapSort**(L)
input list L
output sorted list L

1. $H \leftarrow \text{empty heap}$
2. while ($!L.isEmpty()$) {phase 1}
 $k \leftarrow L.removeFirst()$
 $H.insertItem(k)$
3. while ($!H.isEmpty()$) {phase 2}
 $k \leftarrow H.removeMin()$
 $L.addLast(k)$
4. return

- ◆ 이 알고리즘을 **힙 정렬**(**heap sort**) 알고리즘이라고 부른다

힙 정렬 개선

- ◆ Heap sort의 성능 향상을 위한 두 가지 개선점
 - 제자리 힙 정렬은 heap sort의 공간 사용을 줄인다
 - 상향식 힙 생성은 heap sort의 속도를 높인다



제자리 힙 정렬

- ◆ 이 방식은 정렬되어야 할 리스트가 **배열**로 주어진 경우에만 적용
- ◆ 힙을 저장하는데 리스트 L 의 일부를 사용함으로써 외부 힙 사용을 피한다
- ◆ 지금까지 사용했던 **최소힙**(min-heap) 대신, 최대 원소가 맨위에 오게 되는 **최대힙**(max-heap)을 사용

제자리 힙 정렬 알고리즘

Alg *inPlaceHeapSort*(A)

input array *A* of *n* keys

output sorted array *A*

1. *buildHeap*(A) {phase 1}
2. **for** *i* \leftarrow *n* **downto** 2 {phase 2}
 $A[1] \leftrightarrow A[i]$
 downHeap(1, *i* - 1)
3. **return**

Alg *buildHeap*(A)

input array *A* of *n* keys

output heap *A* of size *n*

1. **for** *i* \leftarrow 1 **to** *n*
 insertItem(*A*[*i*])
2. **return**

Alg *downHeap*(*i*, *last*)

input index *i* of array *A* representing a
 maxheap of size *last*

output none

1. *left* \leftarrow 2*i*
2. *right* \leftarrow 2*i* + 1
3. **if** (*left* > *last*) {external node}
 return
4. *greater* \leftarrow *left*
5. **if** (*right* \leq *last*)
 if (*key*(*A*[*right*]) > *key*(*A*[*greater*]))
 greater \leftarrow *right*
6. **if** (*key*(*A*[*i*]) \geq *key*(*A*[*greater*]))
 return
7. $A[i] \leftrightarrow A[\text{greater}]$
8. *downHeap*(*greater*, *last*)

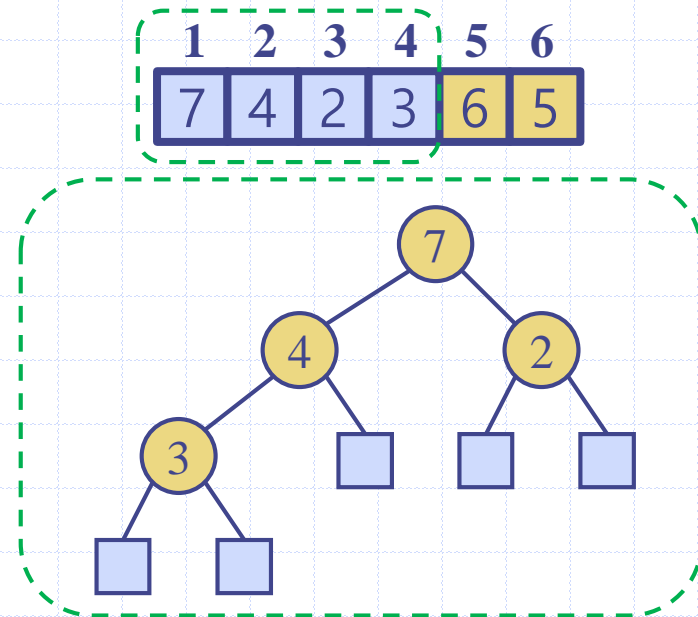
제자리 힙 정렬 (conti.)

◆ 알고리즘 수행의 어떤 시점에서든,

- L 의 첨자 1부터 i 까지의 왼쪽 부분은 힙의 원소들을 저장하는데 사용
- 그리고 첨자 $i+1$ 부터 n 까지의 오른쪽 부분은 리스트의 원소들을 저장하는데 사용

◆ 예: $i = 4$

◆ 그러므로, L 의 (첨자 1, ..., i 에 있는) 첫 i 개의 원소들은 힙의 **배열** 표현 - 즉, 첨자 k 의 원소는 첨자 $2k$ 및 $2k+1$ 의 자식들보다 크거나 같다



제자리 힙 정렬 (conti.)

◆ 1기

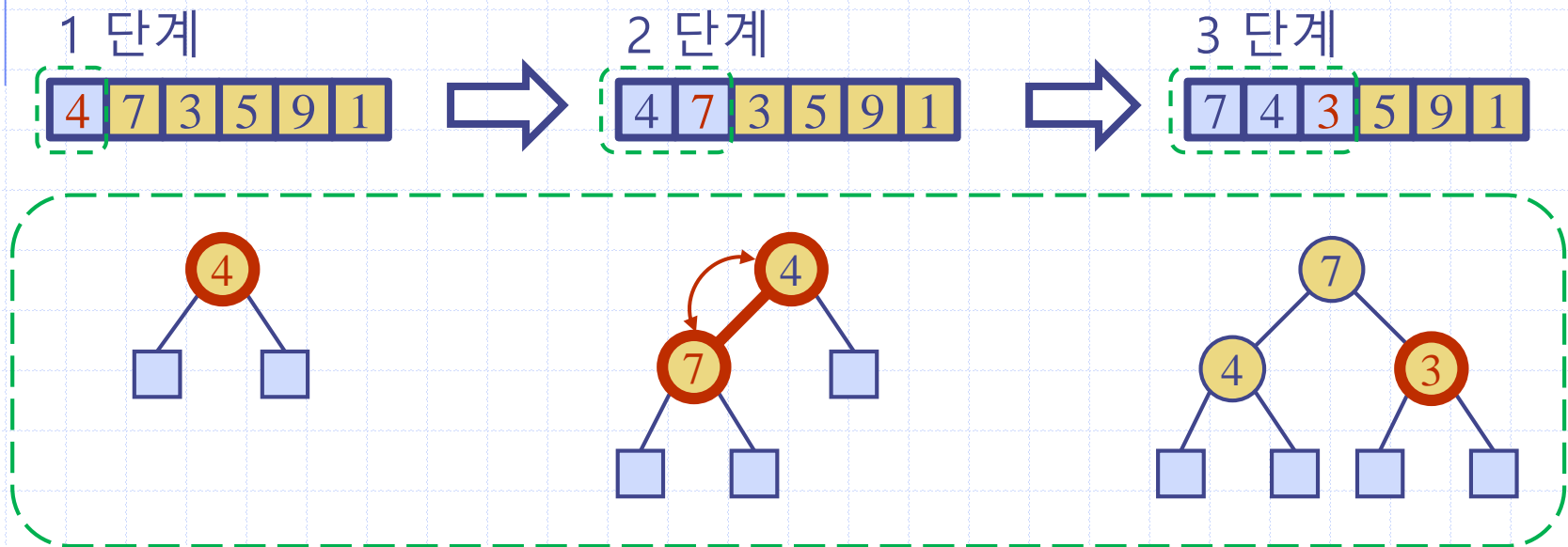
- 비어 있는 힙으로부터 출발하여 힙과 리스트의 경계를 **왼쪽에서 오른쪽으로** 한 번에 한 칸씩 이동
- 단계 $i(i = 1, \dots, n)$ 에서, 첨자 i 에 있는 원소를 힙에 추가함으로써 힙을 확장

◆ 2기

- 비어 있는 리스트로부터 출발하여 힙과 리스트의 경계를 **오른쪽에서 왼쪽으로** 한 번에 한 칸씩 이동
- 단계 $i(i = n, \dots, 2)$ 에서, 힙의 최대 원소를 삭제하여 리스트의 첨자 i 에 저장함으로써 리스트를 확장

제자리 힙 정렬 예

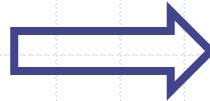
- ◆ 1기와 2기의 각 단계에서, 배열 가운데 힙에 쓰인 부분을 파란색 원소로 표시
- ◆ 아래 점선 내에 보인 이진트리 관점의 힙은 가상적일 뿐, 제자리 알고리즘에 의해 실제 생성되지는 않음에 유의
- ◆ 1기 작업 시작



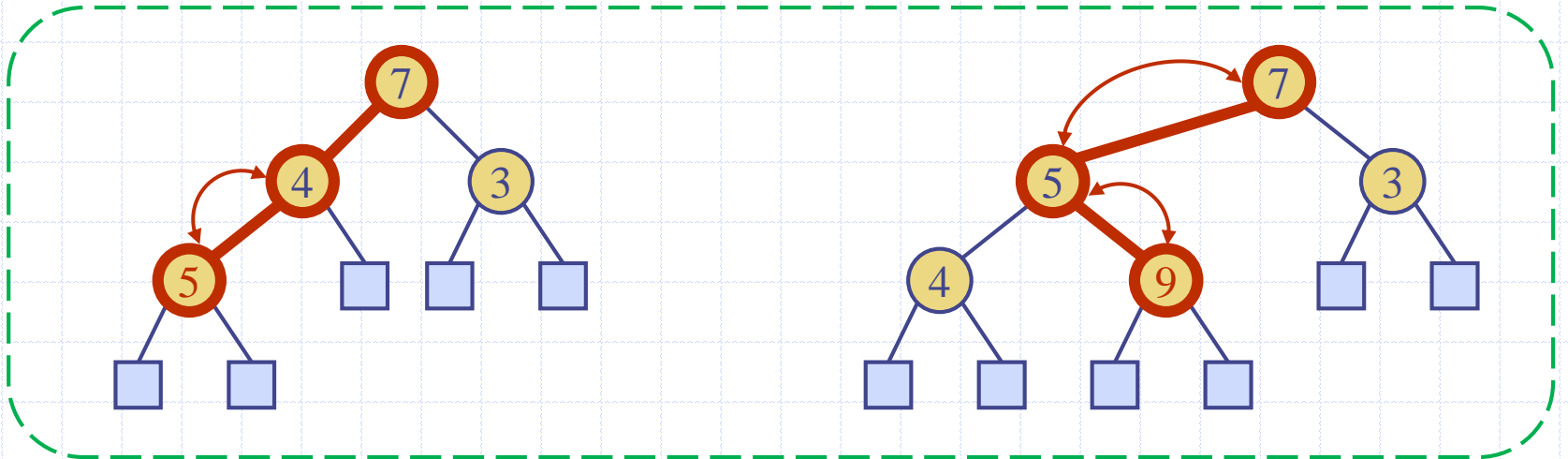
제자리 힙 정렬 예 (conti.)

◆ 1기 작업 계속

4 단계



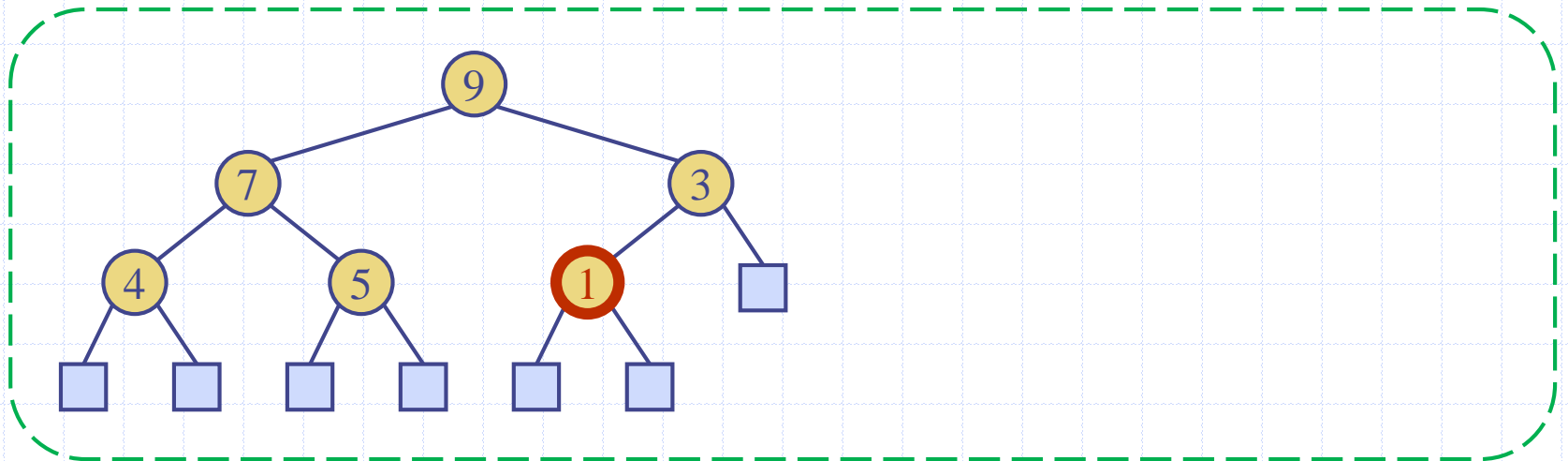
5 단계



제자리 힙 정렬 예 (conti.)

- ◆ 1기 작업 완료
- ◆ 리스트가 **최대힙**으로 변환됨

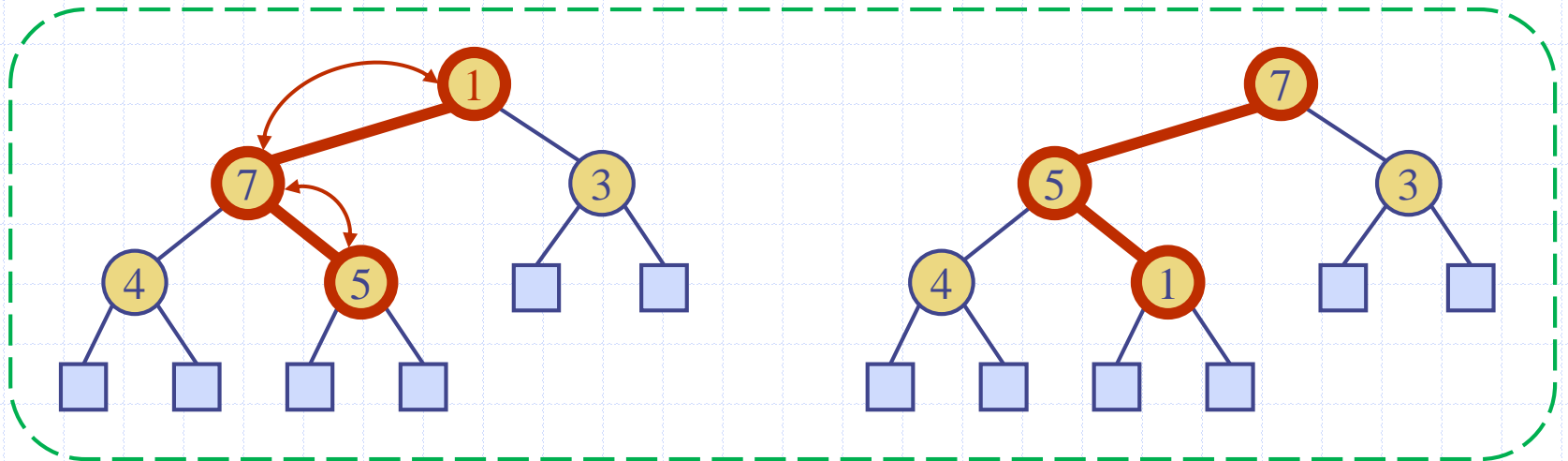
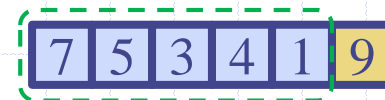
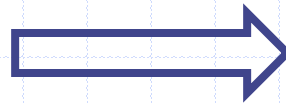
6 단계



제자리 힙 정렬 예 (conti.)

◆ 2기 작업 시작

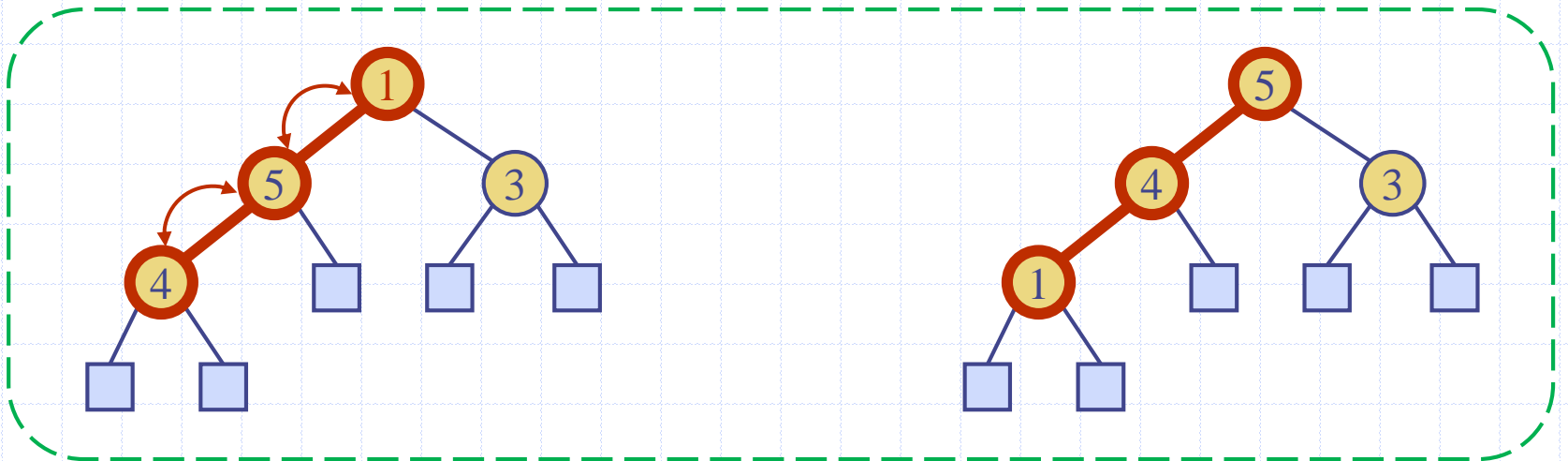
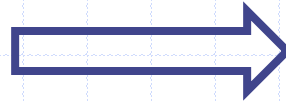
n 단계



제자리 힙 정렬 예 (conti.)

◆ 2기 작업 계속

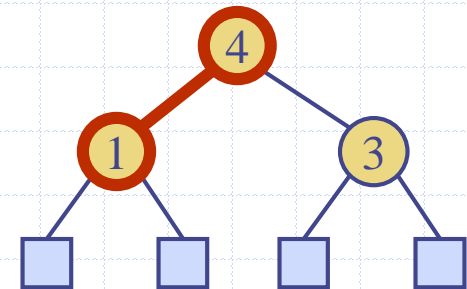
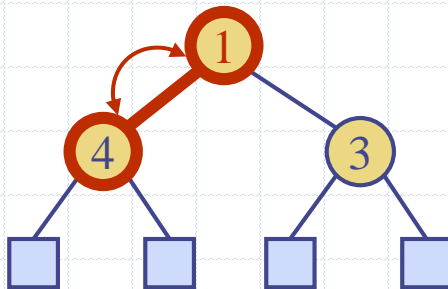
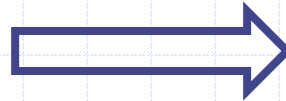
n - 1 단계



제자리 힙 정렬 예 (conti.)

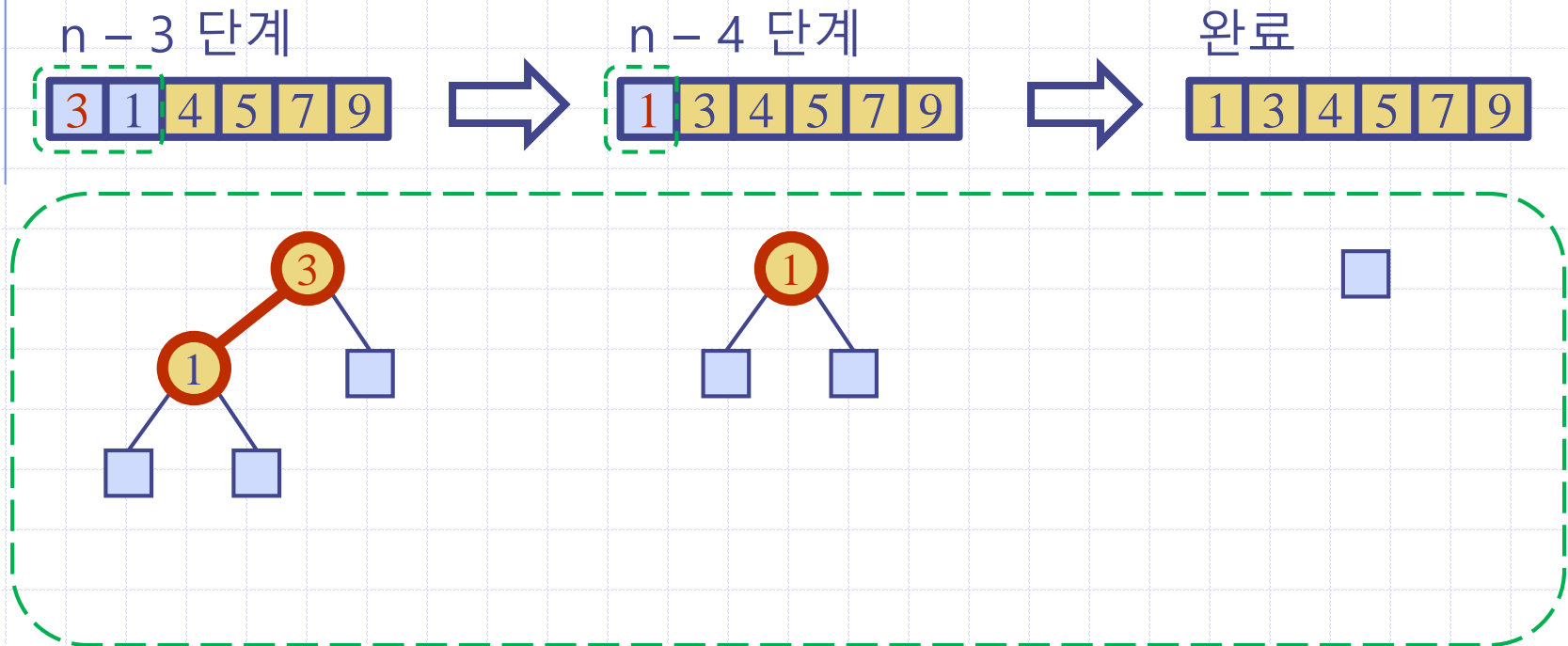
◆ 2기 작업 계속

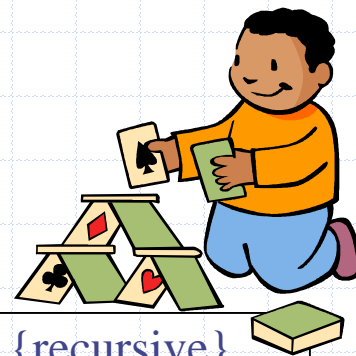
n - 2 단계



제자리 힙 정렬 예 (conti.)

- ◆ 2기 작업 계속
- ◆ 리스트 정렬 완료





상항식 힙 생성

- ◆ heap-sort의 1기에서, n 회의 연속적인 **insertItem** 작업을 사용하여 $O(n \log n)$ 시간에 힙을 생성했다
- ◆ 대안으로, 만약 힙에 저장되어야 할 모든 키들이 **미리** 주어진다면, $O(n)$ 시간에 수행하는 **상항식 생성 방식**이 있다

Alg *buildHeap*(L) {recursive}
input list L storing n keys
output heap T storing the keys in L

1. **if** ($L.isEmpty()$)
 return empty heap
2. Remove the first key, k , from L
3. Split L into two sublists, L_1 and L_2 ,
 each of size $(n - 1)/2$ approximately
4. $T_1 \leftarrow \text{buildHeap}(L_1)$
5. $T_2 \leftarrow \text{buildHeap}(L_2)$
6. Create a binary tree T with root r
 storing k , left subtree T_1 , and right
 subtree T_2
7. **downHeap**(r)
8. **return** T

buildHeap의 split 수행 내용

◆ 순차힙 생성 시

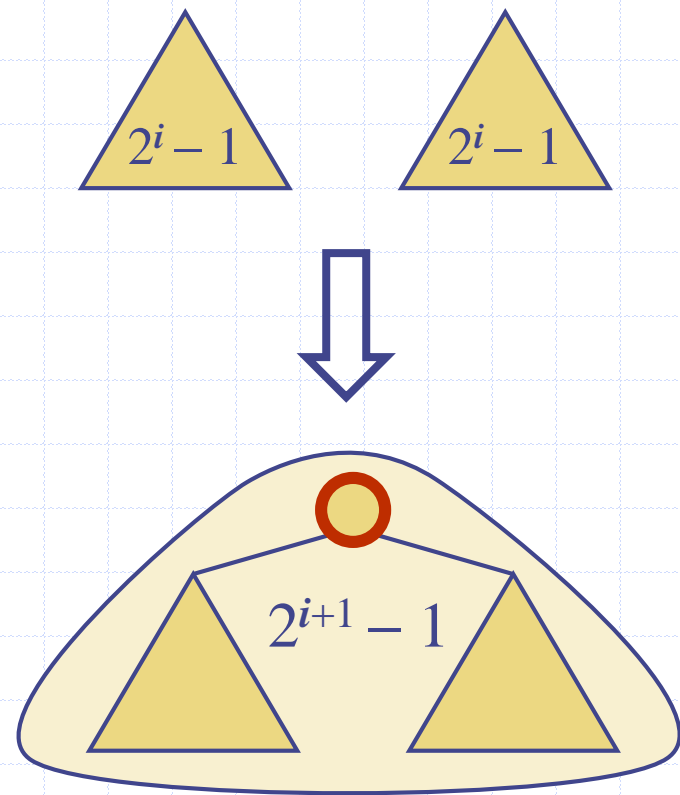
- 입력 배열 L 을 첫번째 원소 k 를 루트로 하는 **완전이진트리**로 본다 – 배열 특성 상 k 삭제는 불필요
- L_1 을 왼쪽 부트리로, L_2 를 오른쪽 부트리로 설정

◆ 연결힙 생성 시

- $|L| = 2^i - 1$ 인 경우(즉, **포화이진트리**):
 $|L_1| = |L_2| = 2^{i-1} - 1$ 로 입력 리스트를 이등분
- 그렇지 않은 경우:
별도의 **정교한** 분할 수식 필요

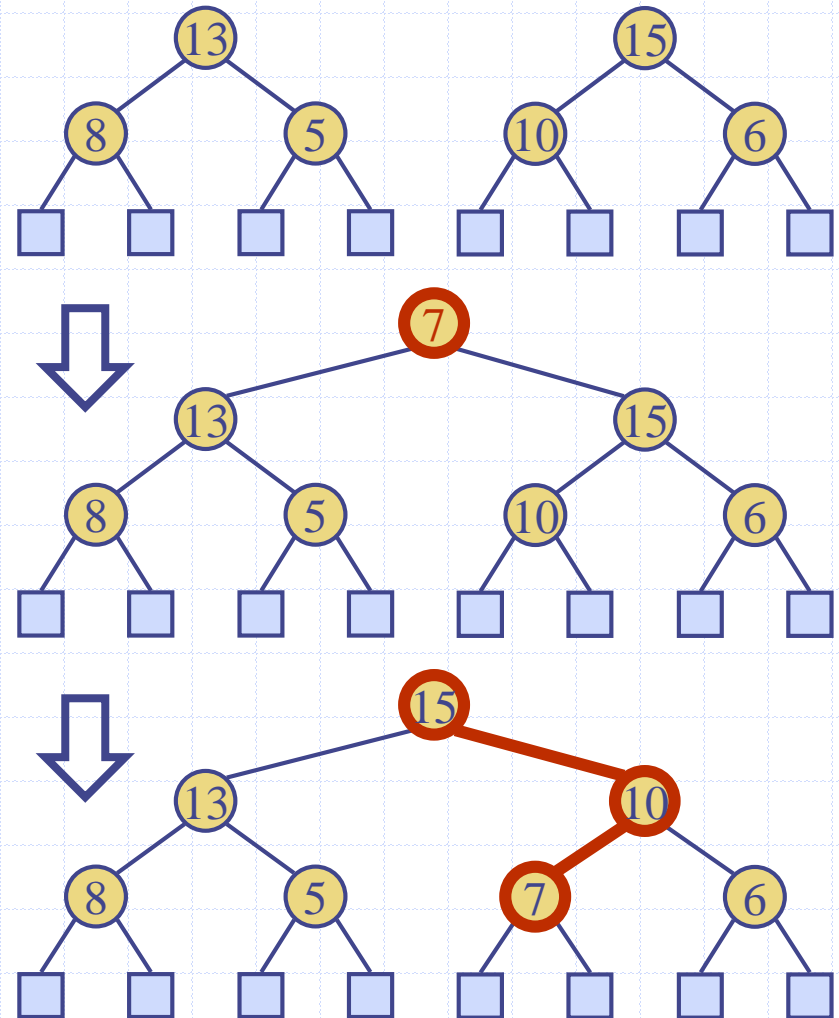
상항식 힙생성 (conti.)

- ◆ $\log n$ 단계만을 사용하여 주어진 n 개의 키를 저장하는 힙 생성 가능
- ◆ 단계 i 에서, 각각 $2^i - 1$ 개의 키를 가진 두 개의 힙을 $2^{i+1} - 1$ 개의 키를 가진 힙으로 합병



두 개의 힙을 합병

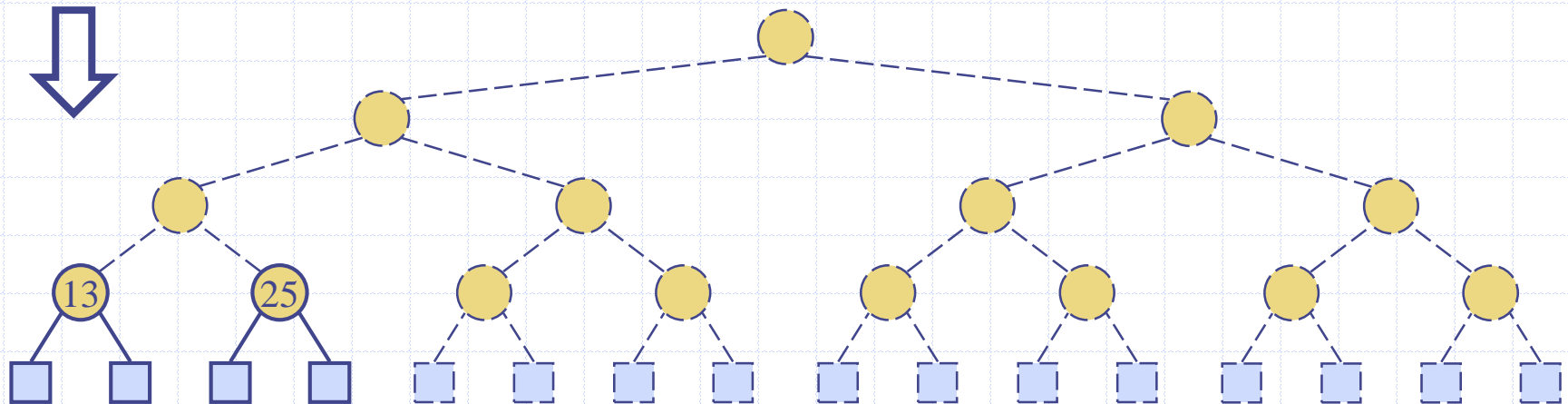
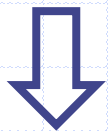
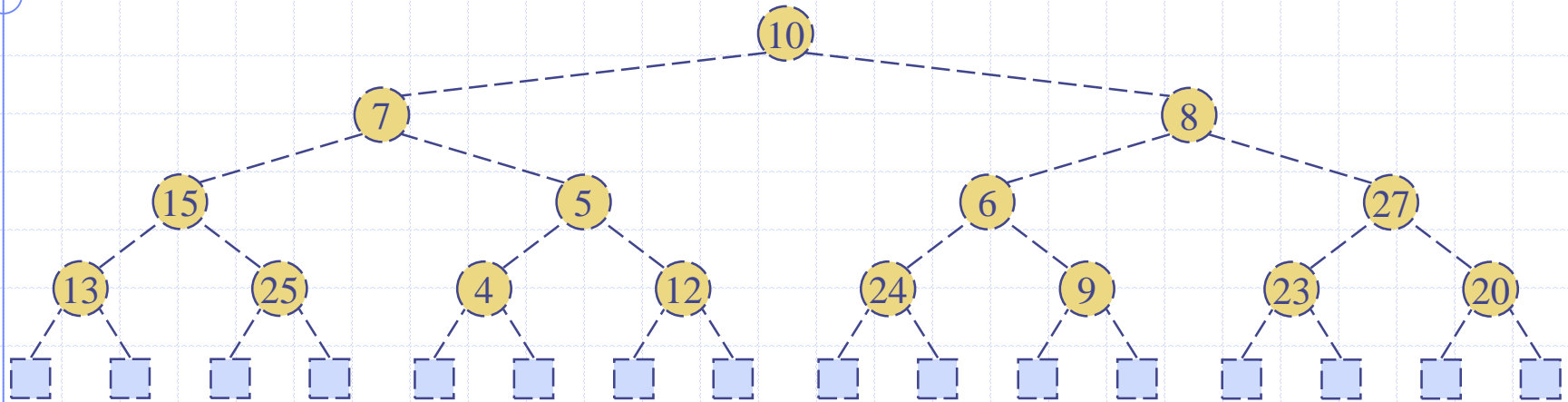
- ◆ 두 개의 힙과 키 k 가 주어졌을 때,
- ◆ k 를 저장한 노드를 루트로, 두 개의 힙을 부트리로 하는 새 힙을 생성
- ◆ 힙순서 속성을 복구하기 위해 **downheap**을 수행

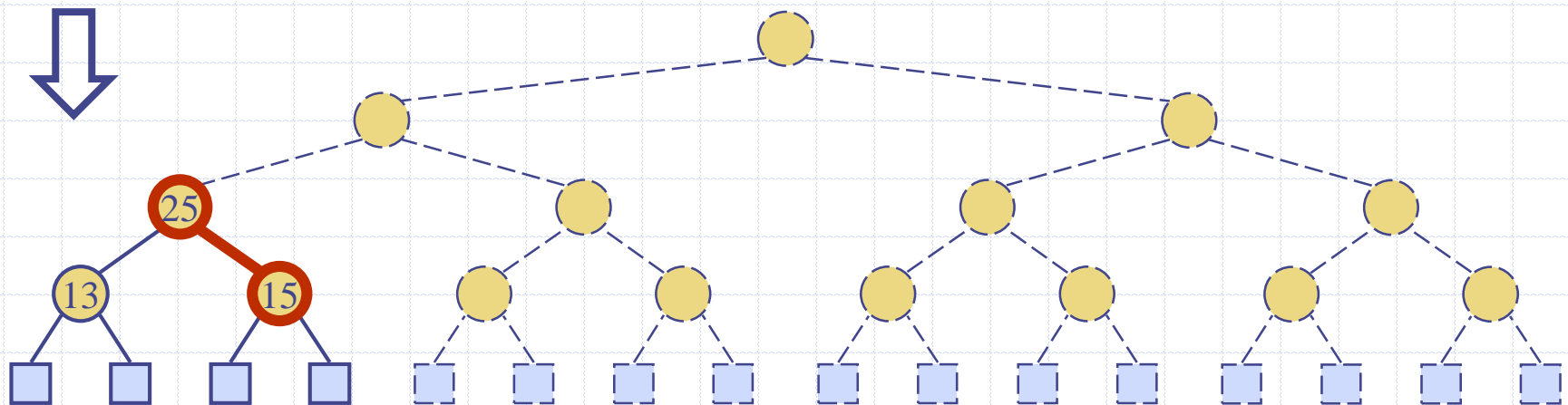


“상향식”이라 불리는 이유?

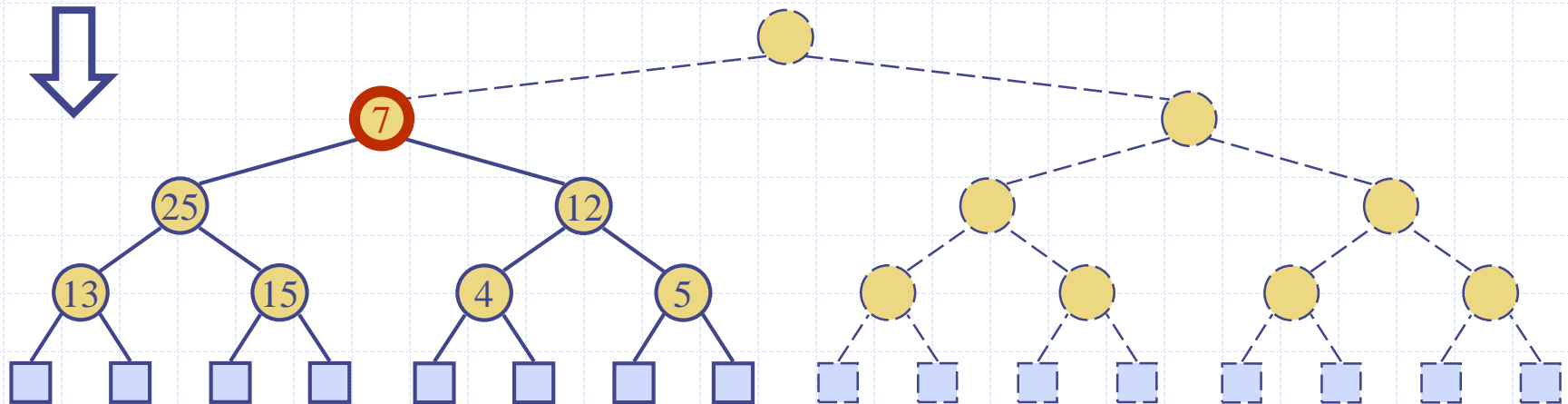
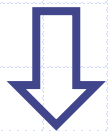
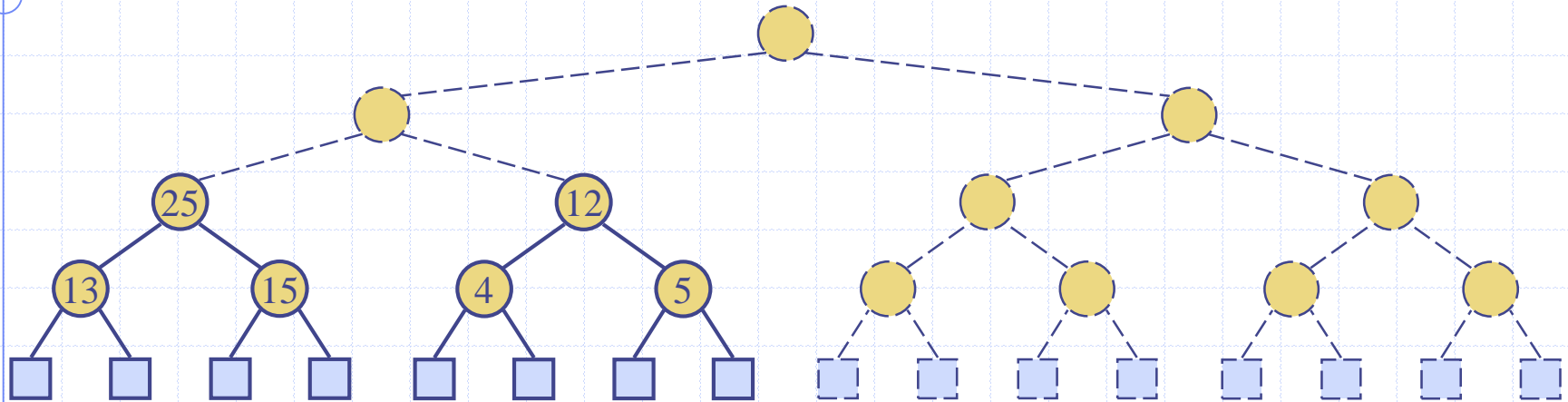
- ◆ 이 버전은 각 재귀호출이 힙인 부트리를 반환하는 방식 때문에 **상향식**이라 불린다
- ◆ 다시 말해, T 의 **힙화**(heapification)는 외부노드에서 시작하여, 각 재귀호출이 반환함에 따라 트리 위쪽으로 진행
- ◆ 이 때문에 종종 **힙화한다**(heapify)고 말하기도 한다

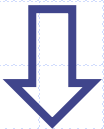
상항식 힙생성 예



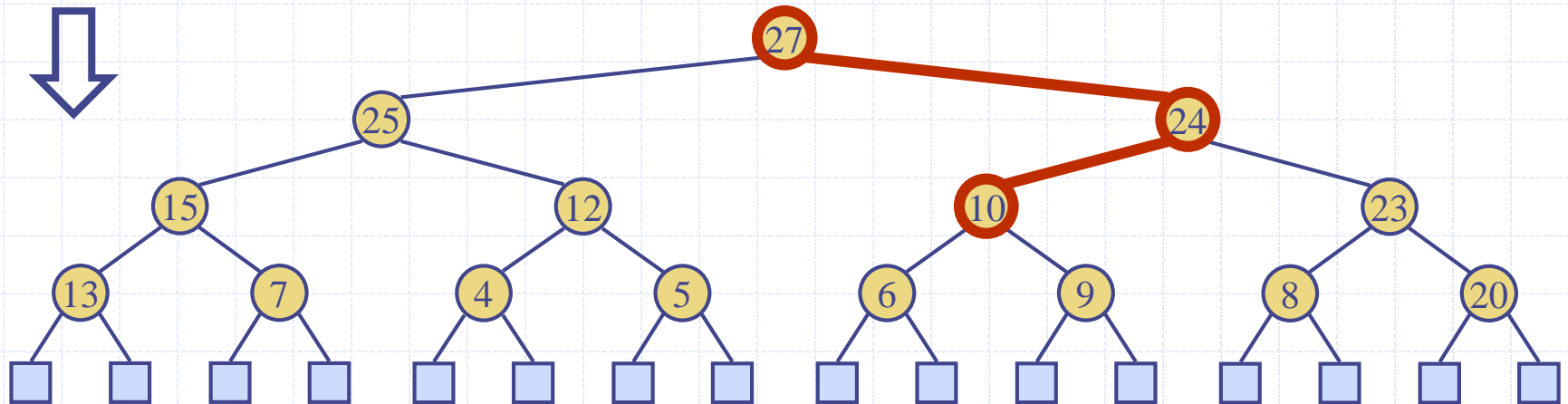
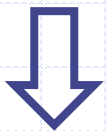
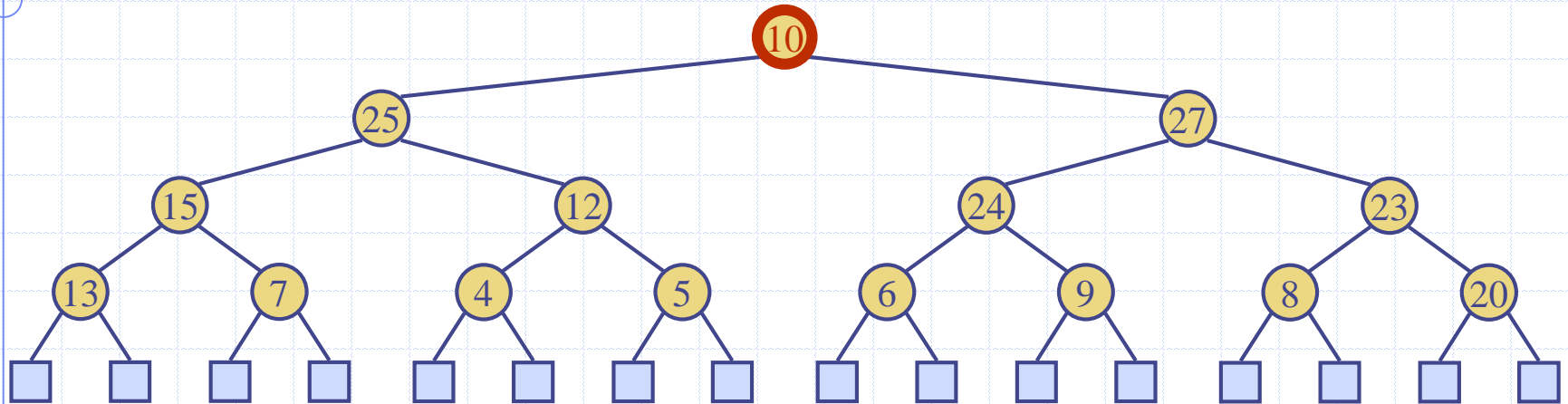


상항식 힙생성 예 (conti.)





상항식 힙생성 예 (conti.)

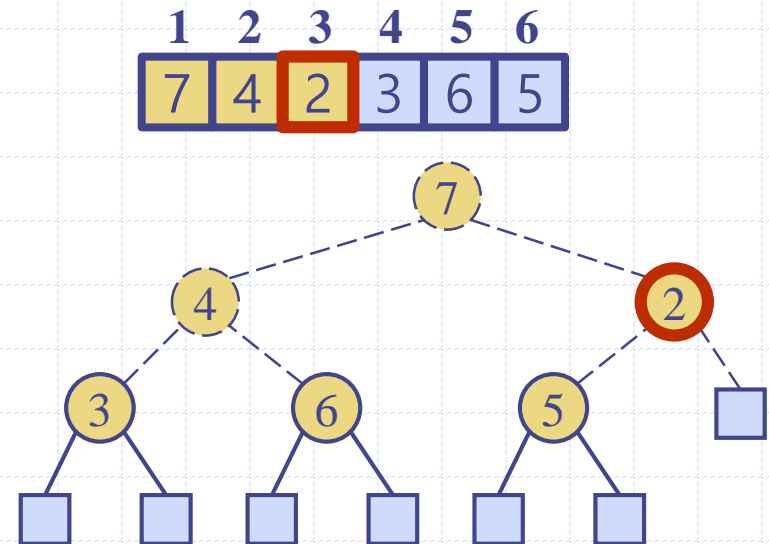


비재귀적 상향식 힙생성

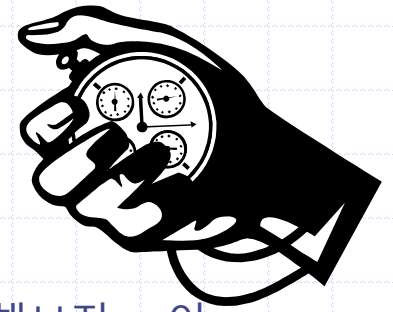
- ◆ 상향식 힙생성의 비재귀 버전
- ◆ 정렬되어야 할 리스트가 **배열**로 주어진 경우에만 적용
- ◆ 이 방식은 힙생성 절차가 "내부노드를 왼쪽 자식으로 가지는 가장 깊은 내부노드 가운데 가장 오른쪽 노드"에서 시작하여 루트로 향하는 후진방향으로 반복 수행
- ◆ 시작 노드: 첨자 $\lfloor n/2 \rfloor$ 인 노드
- ◆ 예: $n = 6$

Alg *buildHeap*(*A*)
input array *A* of *n* keys
output heap *A* of size *n*

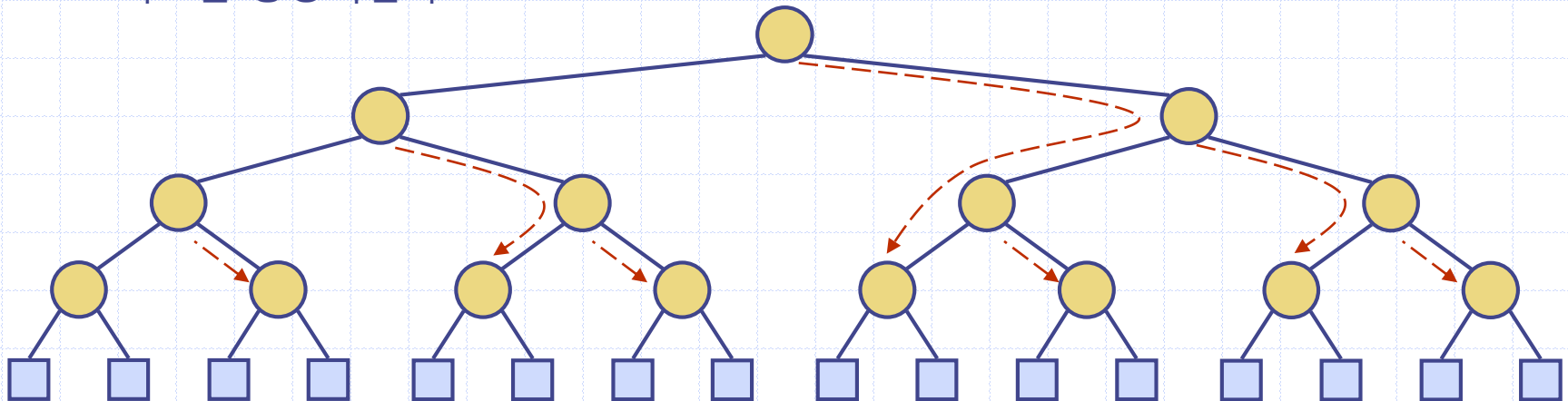
1. for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
 downHeap(*i*, *n*)
2. return

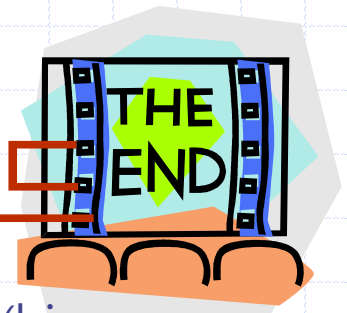


분석



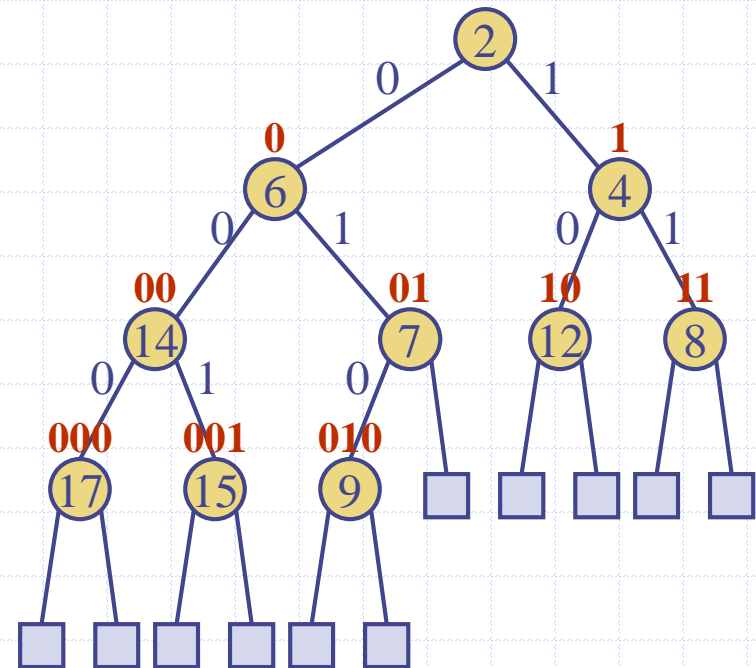
- ◆ **downheap**의 최악의 경우 시간을 **대리경로**를 사용하여 시각화해보자 – 이 대리경로는 먼저 오른쪽 자식노드로 이동한 후 힙의 바닥까지 반복적으로 왼쪽 자식노드를 따라 이동 – 이 경로는 대리경로일 뿐 실제의 **downheap** 경로와는 다를 수 있다
- ◆ 각 노드는 최대 두 개의 대리경로에 의해 순회되므로, 대리경로들의 전체 노드 수는 $O(n)$
- ◆ 따라서, 상향식 힙생성은 $O(n)$ 시간에 수행
- ◆ 하지만, **heap-sort** 2기의 최악실행시간은 $\Omega(n \log n)$
- ◆ 그래도, 상향식 힙생성은 n 회의 연속적인 삽입보다 빠르므로 **heap-sort** 1기의 속도를 향상시킨다





응용문제: 힙의 마지막 노드

- ◆ 이진트리의 루트로부터 어떤 노드까지의 경로를 **이진수열**(binary string)로 표현할 수 있다 – 여기서 0은 “왼쪽으로 이동”을, 1은 “오른쪽으로 이동”을 의미
- ◆ 예를 들어, 이진수열 “010”은 루트로부터 시작하는 다음의 경로를 의미
 - 왼쪽자식으로 이동,
 - 오른쪽 자식으로 이동,
 - 마지막으로, 왼쪽 자식으로 이동
- ◆ 이 표현법에 기초하여, n 개의 원소를 가지는 힙의 **마지막** 노드를 찾기 위한 **로그시간**(logarithmic-time) 알고리즘을 **의사코드**로 작성하라
 - **findLastNode**(v, n): 루트가 v 며 n 개의 원소로 이루어진 힙의 마지막 노드를 반환



해결

- ◆ 힙의 마지막 노드에 이르는 경로는 n 의 이진수 표기에서 최상위 비트를 제거하여 얻은 경로로 주어진다
 - 예: $n = 10$ 인 경우, 경로는 "010"

해결 (conti.)

Alg *findLastNode*(v, n)

input root v of a heap, integer n

output last node of the heap with root v

1. $S \leftarrow \text{empty stack}$
2. *binaryExpansion*(n, S)
3. $S.\text{pop}()$ {remove highest-order bit}
4. **while** ($\neg S.\text{isEmpty}()$)
 $\text{bit} \leftarrow S.\text{pop}()$
 if ($\text{bit} = 0$)
 $v \leftarrow \text{leftChild}(v)$
 else { $\text{bit} = 1$ }
 $v \leftarrow \text{rightChild}(v)$
5. **return** v

Alg *binaryExpansion*(i, S)

input, integer i , empty stack S

output binary representation of i in stack S

1. **while** ($i \geq 2$)
 $S.\text{push}(i \% 2)$
 $i \leftarrow i/2$
2. $S.\text{push}(i)$
3. **return**