ALGORITHMS AND DATA STRUCTURES

PRIORITY QUEUES

PROBLEMS THAT REQUIRE A PRIORITY CRITERIUM

- Printing server: Takes several print jobs and executes them in certain order (e.g. according to the number of pages)
- Resources distribution: Managing limited resources such as bandwidth. If outgoing data queues due to insufficient bandwidth, all other requests can be halted and sent the important ones according to priority
- Emergency Room: Tend to patients according to the urgency of their health situation

PRIORITY QUEUES

- is an abstract data type that allows (at least)
  - `insert`: does the obvious thing
  - `deleteMin`: returns and removes minimum element
- can be viewed as queues that bookkeep some sort of order

PRIORITY QUEUES

- is an abstract data type that allows (at least)
  - `insert`: does the obvious thing
  - `deleteMin`: returns and removes minimum element
- can be viewed as queues that bookkeep some sort of order
- Queue analogues
  - `insert` would the equivalent of enqueue
  - `deleteMin` would equivalent of dequeue

PRIORITY QUEUES

- Applications
  - Bandwidth management of info transmission
  - SO process scheduler
  - Implementation of greedy algorithms
  - `heapsort`: sorting algorithm
  - Discrete-event simulations

PRIORITY QUEUES

- Implementation
  - Linked list: insertion $O(1)$; delete $O(N)$
  - If sorted: insertion $O(N)$; delete $O(1)$
  - Binary search tree: both operations $O(\lg N)$
  - If balanced: worst-case bound $O(\lg N)$

BINARY HEAP

- is an implementation of a priority queue
- is a binary tree that is complete (exception: deepest layer)

BINARY HEAP

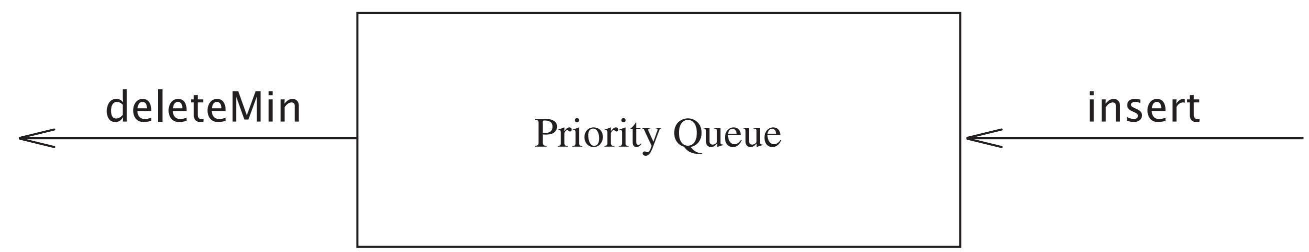
- is an implementation of a priority queue
- is a binary tree that is complete (exception: deepest layer)
- has two properties:
  - **structure property**, and
  - **heap-order property**

BINARY HEAP

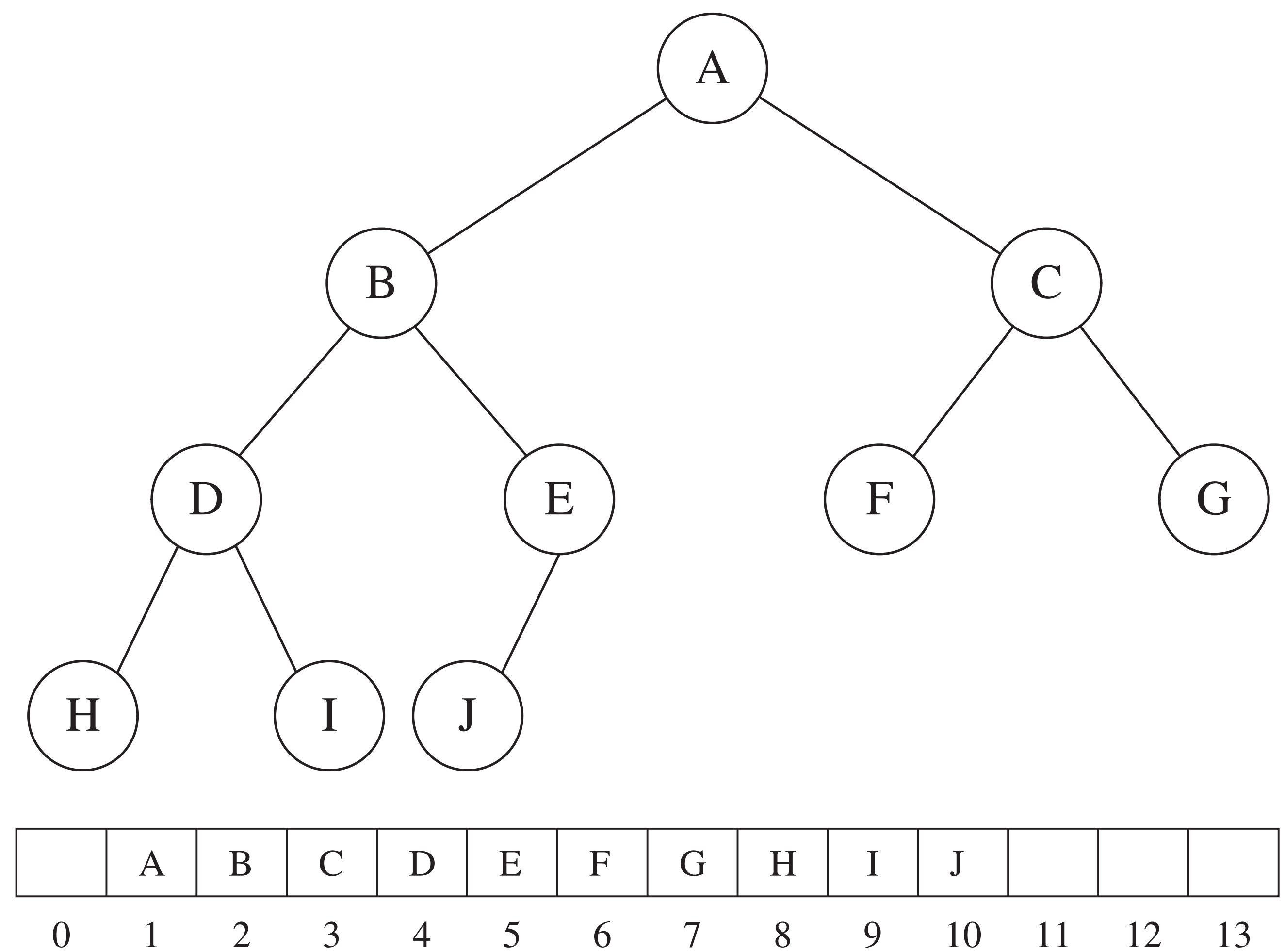
- is an implementation of a priority queue
- is a binary tree that is complete (exception: deepest layer)
- has two properties:
    - **structure property**, and
    - **heap-order property**
- a heap operation might destroy one of such properties
- an operation can never end without restoring properties

BINARY HEAPS & PRIORITY QUEUES

priority queue



binary heap



BINARY HEAP

- **Structure property**
  - a heap is a complete binary tree
  - for depth h it has at most from 2^h to 2^{h+1}-1 nodes
  - heap's regularity => can be casted to an array w/o links
  - leads to extremely fast traversing operations
  - for an element at position i: parent $\lfloor i/2 \rfloor$, children $2i$ and $2i+1$

BINARY HEAP

- **Heap-order property**
  - makes sense to have minimum element at root because we want to find it
  - then in a tree all nodes should be smaller than all of its descendants
  - in a heap, for every node X, the key of its parent is smaller than (or equal to) X's key
  - exception is the root node which has no parent

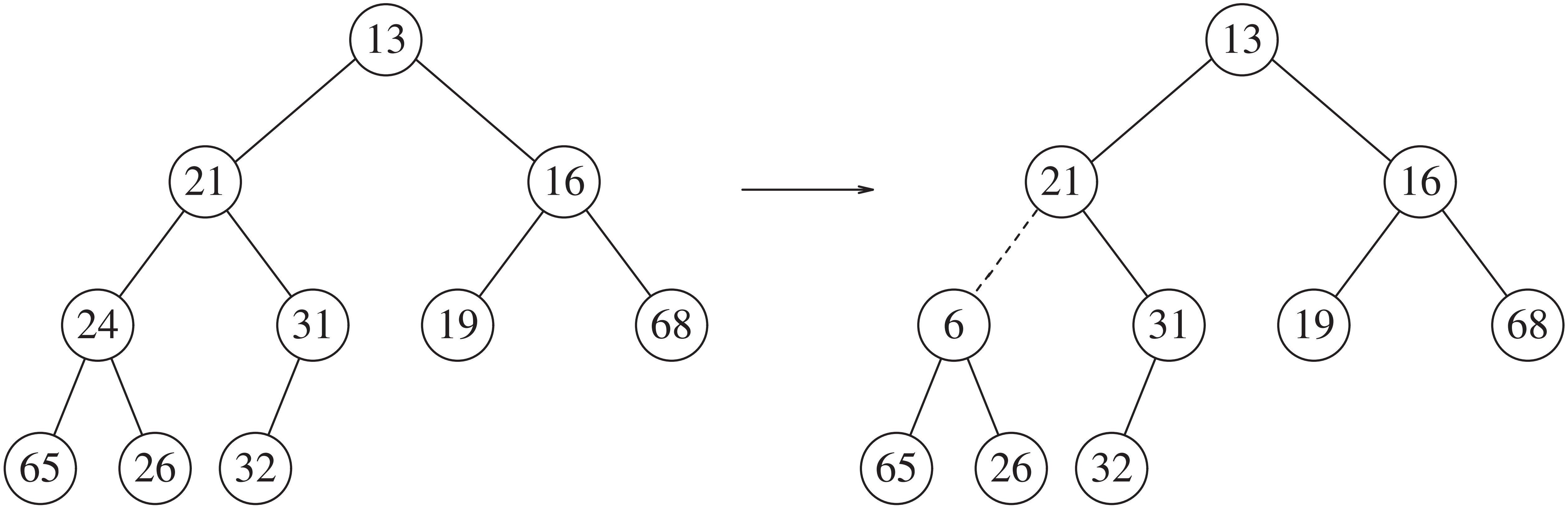
```cpp
template <typename keyType>
class BinaryHeap {
public:
    BinaryHeap(int capacity = 100);
    BinaryHeap(vector<keyType> & items);

    bool isEmpty();
    const keyType & findMin();
    void insert(keyType & x);
    void deleteMin();
    void empty();

private:
    int size;
    vector<keyType> array;

    void buildHeap();
    void percolateDown(int hole);
}
```

BINARY HEAP



Which is a min binary heap?

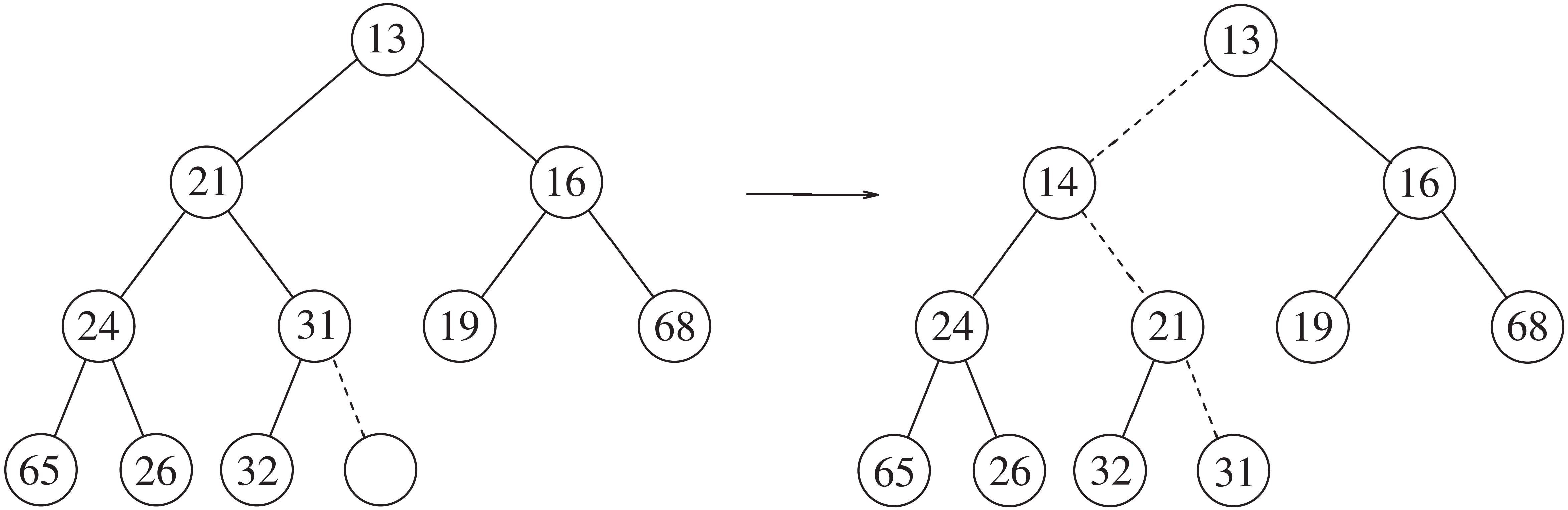
BINARY HEAP OPERATIONS

- **insert**
  - insert key X at bottom (structure property)
  - work its way up to right spot (heap-order property)
    - this is known as ***percolate up***

BINARY HEAP OPERATIONS

- **insert**
  - insert key X at bottom (structure property)
  - work its way up to right spot (heap-order property)
    - this is known as ***percolate up***
  - **2.607** comparisons on average to perform an insert
  - so the average insert moves an element up **1.607** levels
  - time to do the insertion could be as much as $\mathbf{O(\lg} \boldsymbol{N}\mathbf{)}$

BINARY HEAP OPERATIONS



inserting item 14

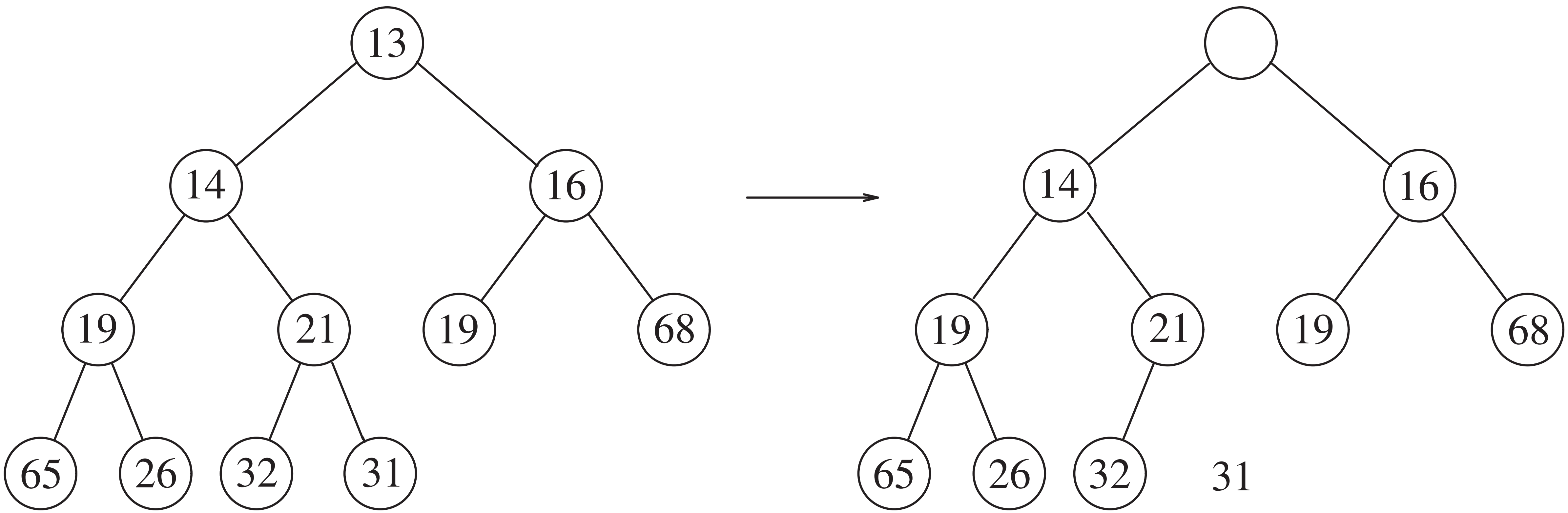
BINARY HEAP OPERATIONS

- **deleteMin**
  - easy: finding the minimum; hard: actually removing it
  - we remove the minimum
  - move the hole created down choosing always to move up the smallest children in current node (the path of minimum children)

BINARY HEAP OPERATIONS

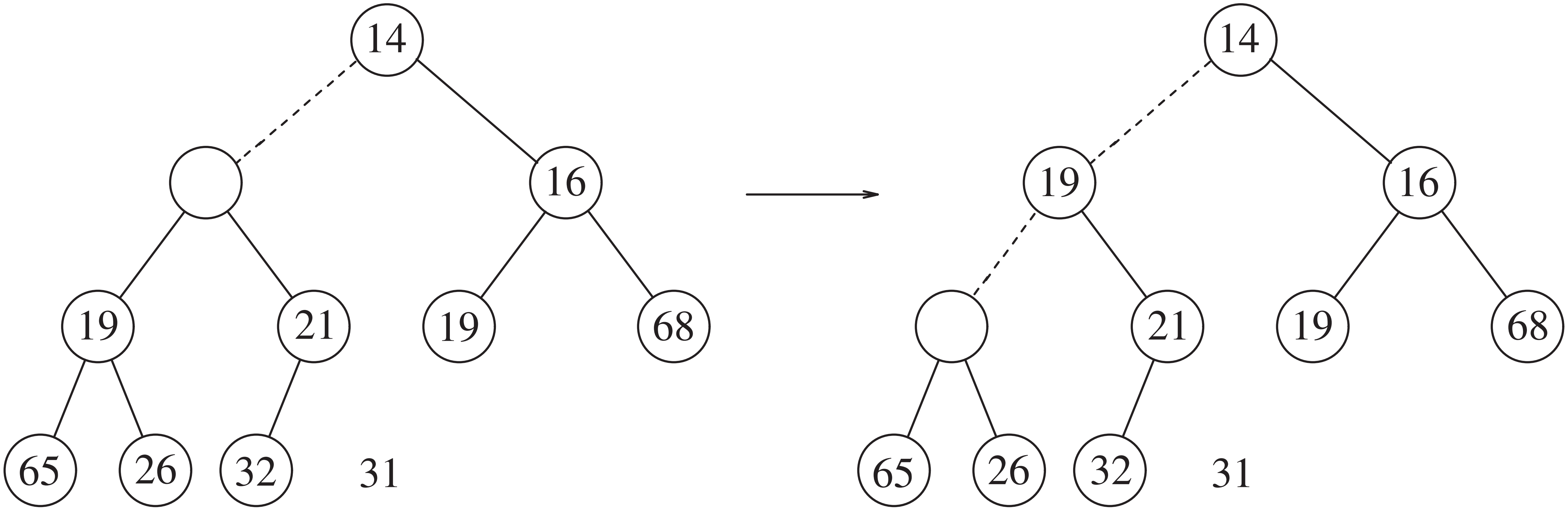
- **deleteMin**
  - easy: finding the minimum; hard: actually removing it
  - we remove the minimum
  - move the hole created down choosing always to move up the smallest children in current node (the path of minimum children)
    - this is known as ***percolate down***
  - worst-case running time for this operation is ***O*(lg *N*)**

BINARY HEAP OPERATIONS



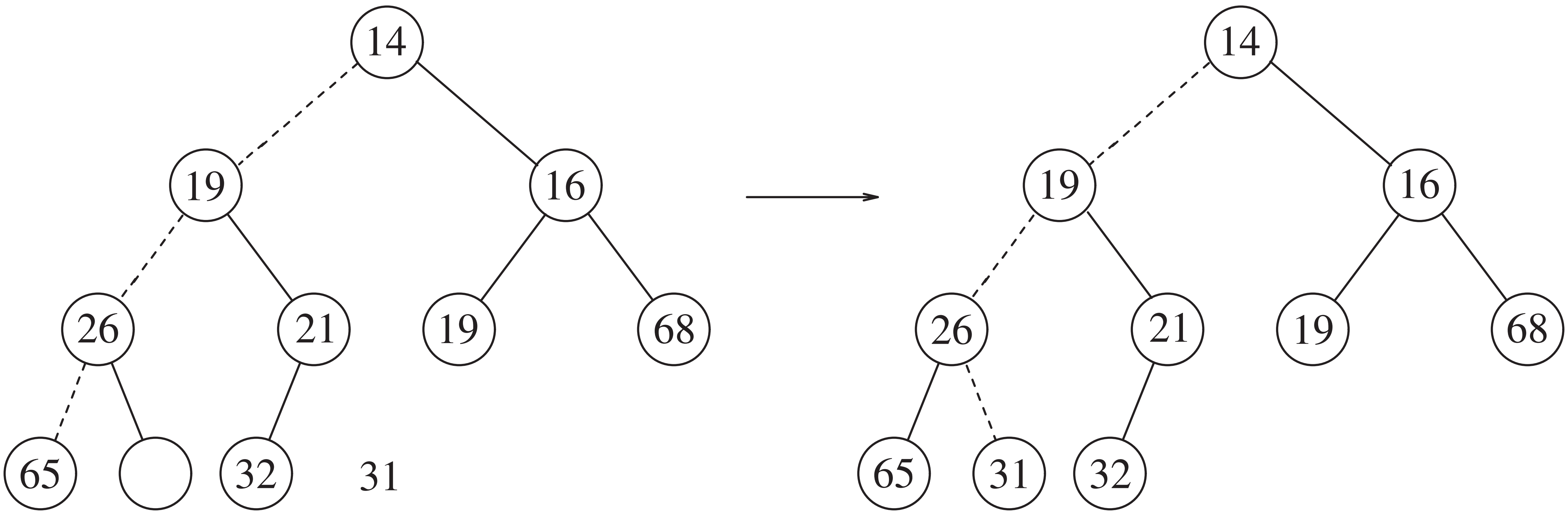
deleting minimum entry

BINARY HEAP OPERATIONS



deleting minimum entry

BINARY HEAP OPERATIONS



deleting minimum entry

BINARY HEAP OPERATIONS

- **Other operations**
  - How to find the maximum? (another ADT)
  - How to change the key (priority) of a given node?
  - How to remove a node at a given position?
  - How to build a heap from an array of elements?

```
FUNCTION max_heapify:
 INPUT: heap A, integer i
 OUTPUT: NONE
 USAGE: max_heapify(A, i)
BEGIN
 l, r = left(i), right(i)
 IF l <= A.size AND A[l] > A[i]
  largest = l
 ELSE, largest = i
 IF r <= A.size AND A[r] > A[largest]
  largest = r
 IF largest != i
  swap(A[i], A[largest])
  max_heapify(A, largest)
END // max_heapify
```

```
FUNCTION build_max_heap:
 INPUT: array A
 OUTPUT: heap B
 USAGE: B = build_max_heap(A)
BEGIN
 A.size = A.length

 FOR i = floor(A.size / 2), 1
 BEGIN
  max_heapify(A, i)
 END

 RETURN A
END // build_max_heap
```

RETURNING MAXIMUM ELEMENT IN MAX-HEAP

```
FUNCTION heap_maximum:
 INPUT: heap A
 OUTPUT: valueType A[1]
 USAGE: max = heap_maximum(A)
BEGIN
 RETURN A[1]
END // heap_maximum
```

```
FUNCTION heap_remove_max:
 INPUT: heap A
 OUTPUT: valueType max
 USAGE: max = heap_remove_max(A)
BEGIN
 max = A[1]
 A[1] = A[A.size]
 A.size = A.size - 1
 max_heapify(A, 1)

 RETURN max
END // heap_remove_max
```

APPLICATIONS OF PRIORITY QUEUES

- **Selection problem**
  - Given a list of N elements (perhaps ordered) and an integer k
  - Problem: Find the smallest/largest k-th element?
  - Previous solutions?

APPLICATIONS OF PRIORITY QUEUES

- **Selection problem**
  - Given a list of N elements (perhaps ordered) and an integer k
  - Problem: Find the smallest/largest k-th element?
  - Previous solutions
    - Sort and get k-th element
    - Partially sort and compare $N-k$ times

APPLICATIONS OF PRIORITY QUEUES

- **Selection problem**
  - Using binary heaps:
    - `buildHeap` with array of N elements, $O(N)$
    - Execute `deleteMin` k times (k-th smallest element)
    - Computational complexity? $O(N+k \lg N)$
    - Keeping record of removes gives a sorted array (*heapsort*!)

APPLICATIONS OF PRIORITY QUEUES

- **Event simulation**
  - Imagine a queue of customers in a bank being served by *k* tellers (input: times of arrival and departure)
  - Problem: what's the average waiting time and line length?
  - Solution: divide problem in events
    - Event 1: customer arrives (occupying a teller)
    - Event 2: customer leaves (freeing a teller)

APPLICATIONS OF PRIORITY QUEUES

- **Event simulation**
  - One way to do it: start a simulation clock at zero ticks
  - Advance the clock one tick at a time, checking if there is an event
    - If there is, then we process the event(s) and compile statistics
  - When there are no customers left and all the tellers are free, the simulation is over

APPLICATIONS OF PRIORITY QUEUES

- **Event simulation**
  - There is a problem with this strategy: simulation depends on time not customers (events)
    - Simulation time could vary greatly
  - *Insight*: advance clock to next time event not to next time tick
  - Since the input is time, we can find out when next event happens and advance the clock to that stage

APPLICATIONS OF PRIORITY QUEUES

- **Event simulation**
- Better strategy
  - The waiting line can be implemented as a queue
  - Set of events happening in the near future can be organized in a priority queue
  - The next event is thus the next arrival or next departure (whichever is sooner)
  - Computational complexity? $O(C \lg(k + 1))$

OTHER TYPES OF HEAPS

- **d-heaps** (binary heap is a 2-heap)
- **Leftist heap** ([very] unbalanced binary heap, merge well)
- **Skew heap** (self-adjusting version of a leftist heap)
- **Binomial heap** (better complexity than LH and SH)

- In the STL: `priority_queue`