Priority Queues (Heaps)

November 2, 2023

Priority Queues

- Many applications require that we process records with keys in order, but not necessarily in full sorted order.
- Often we collect a set of items and process the one with the current minimum value. e.g.
  - Jobs sent to a printer,
  - Operating system job scheduler in a multi-user environment.
  - Simulation environments (Discrete Event Simulator)
- An appropriate data structure for this purpose is called a priority queue.

Definition

- A priority queue is a data structure that supports two basic operations:
  - insert a new item and
  - remove the minimum item.

Implementations Options

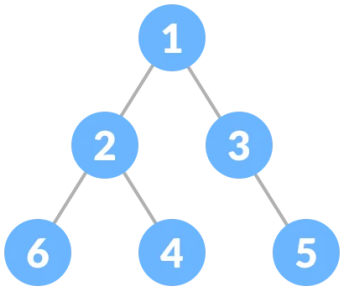
- A simple linked list:
  - Insertion at the front $O(1)$; `deleteMin` $O(N)$, or
  - Keep list sorted; insertion $O(N)$, `deleteMin` $O(1)$
- A binary search tree (BST):
  - This gives an $\Theta(logN)$ average for both operations.
  - But BST class supports many operations that are not required.
- Binary Heap:
  - Can be implemented as a simple array and does not require links.
  - Supports both operations in $O(logN)$ worst-case time.
- d-Heaps:
  - A more general case in which a parent node can have d children
- Leftist and Skew Heaps
- Binomial Queues

Binary Heap

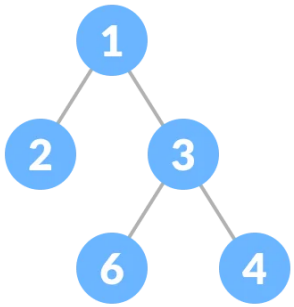
- The binary heap is the classic method used to implement priority queues.
- We use the term heap to refer to the binary heap.
- Heap is different from the term heap used in dynamic memory allocation.
- Heap has two properties:
  - Structure property
  - Ordering property

Structure Property

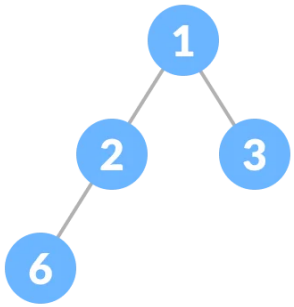
- A heap is a complete binary tree.
- A complete binary tree is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.



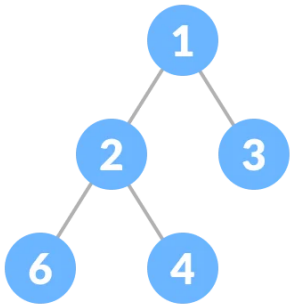
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✗ Complete Binary Tree



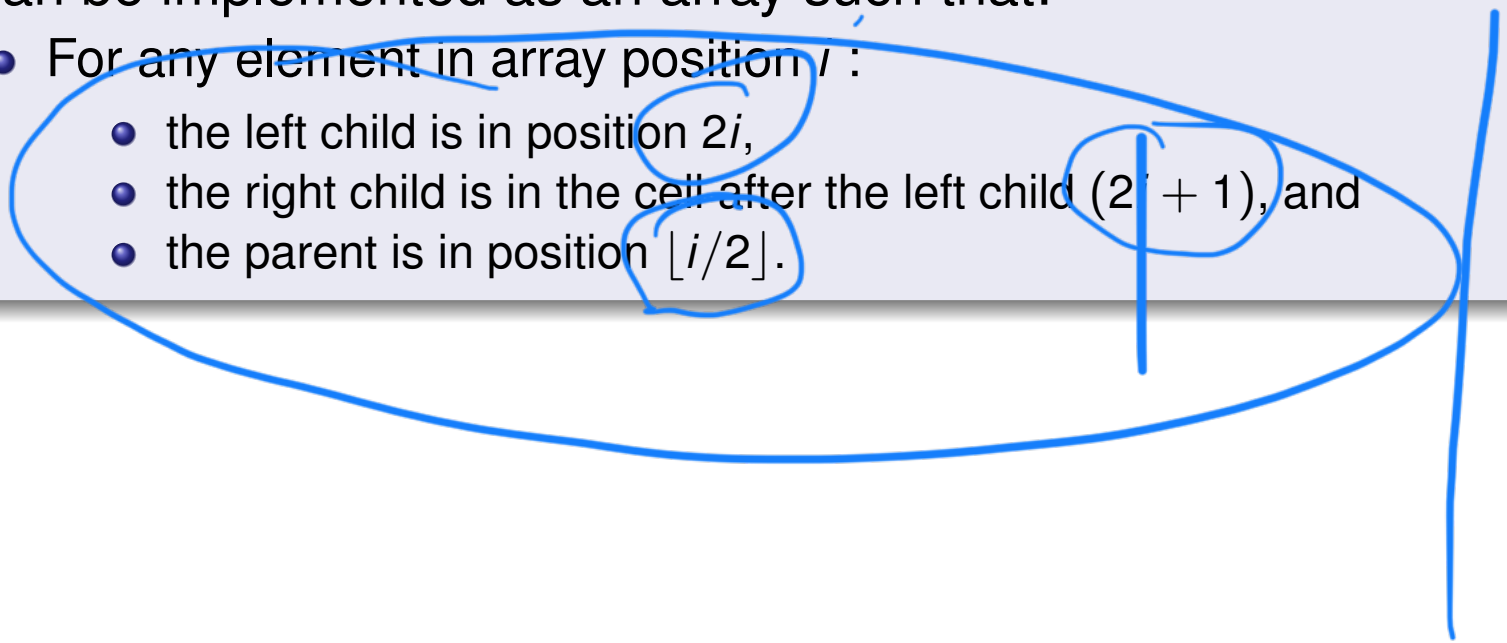
✗ Full Binary Tree
✓ Complete Binary Tree



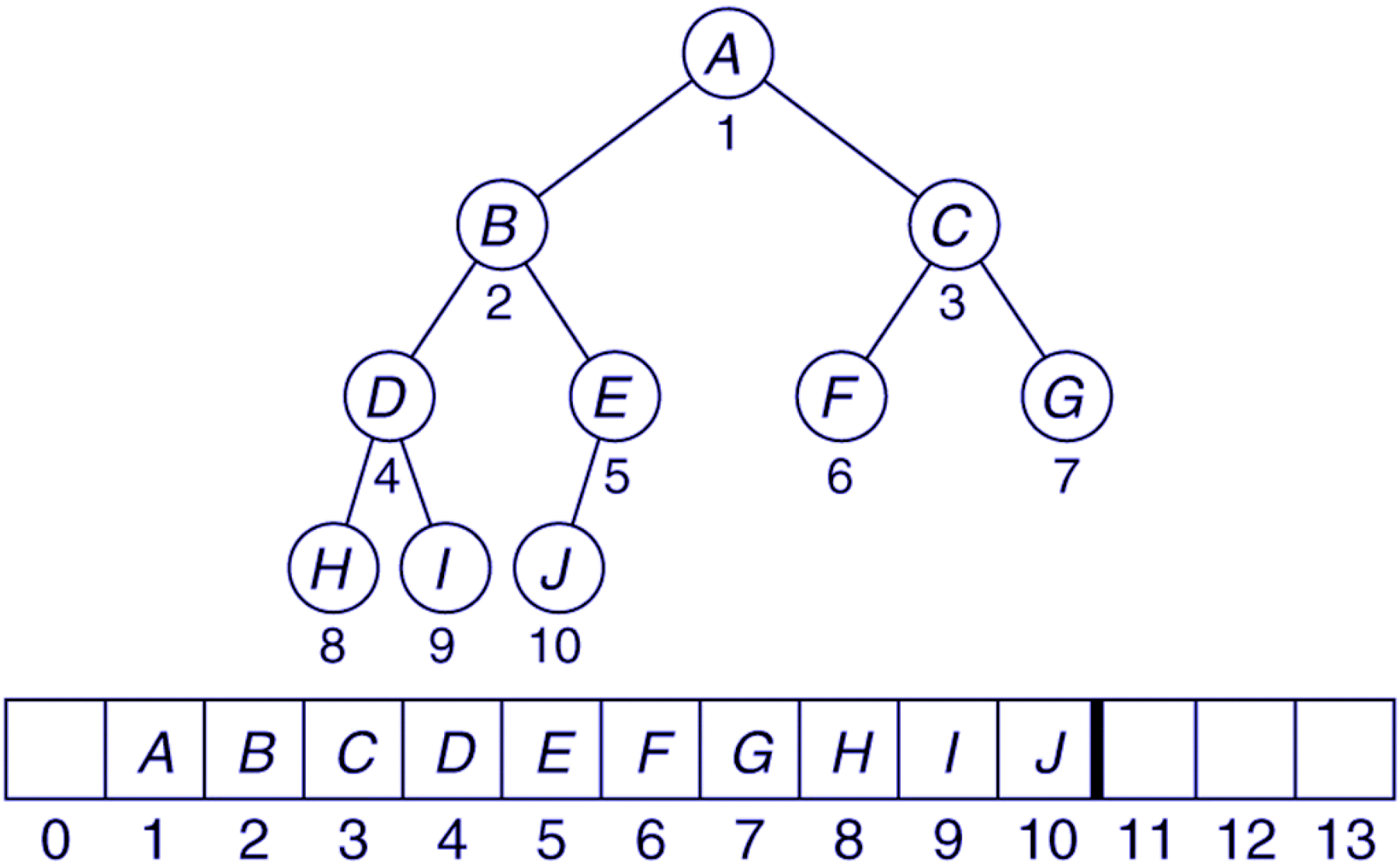
✓ Full Binary Tree
✓ Complete Binary Tree

Properties of a complete binary tree

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
- The height of a complete binary tree is $\lfloor logN \rfloor$.
- It can be implemented as an array such that:
  - For any element in array position i :
    - the left child is in position $2i$,
    - the right child is in the cell after the left child $(2i + 1)$, and
    - the parent is in position $\lfloor i/2 \rfloor$.



A complete binary tree and its array representation



Heap-Order Property

- In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.
- Thus the minimum element is always at the root.
  - Thus we get the operation `findMin` in constant time.
- A max heap supports access of the maximum element instead of the minimum, by changing the heap property slightly.

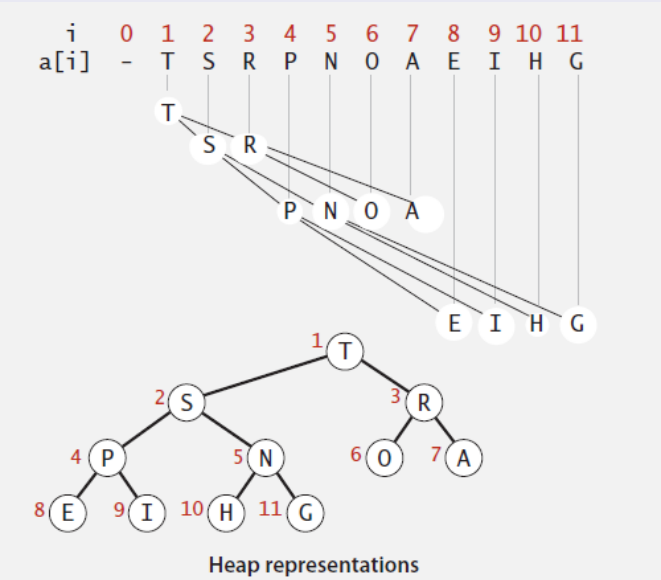
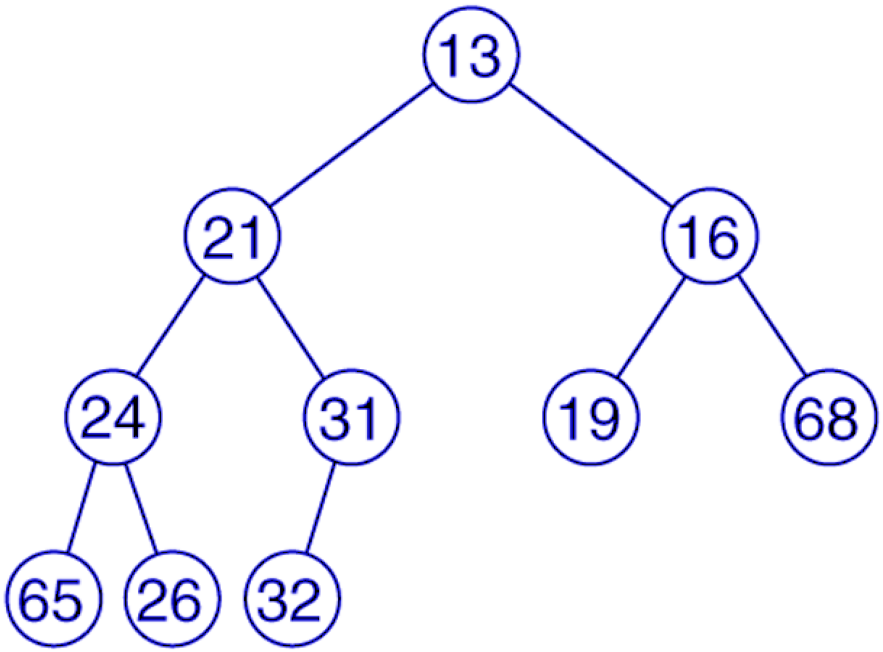
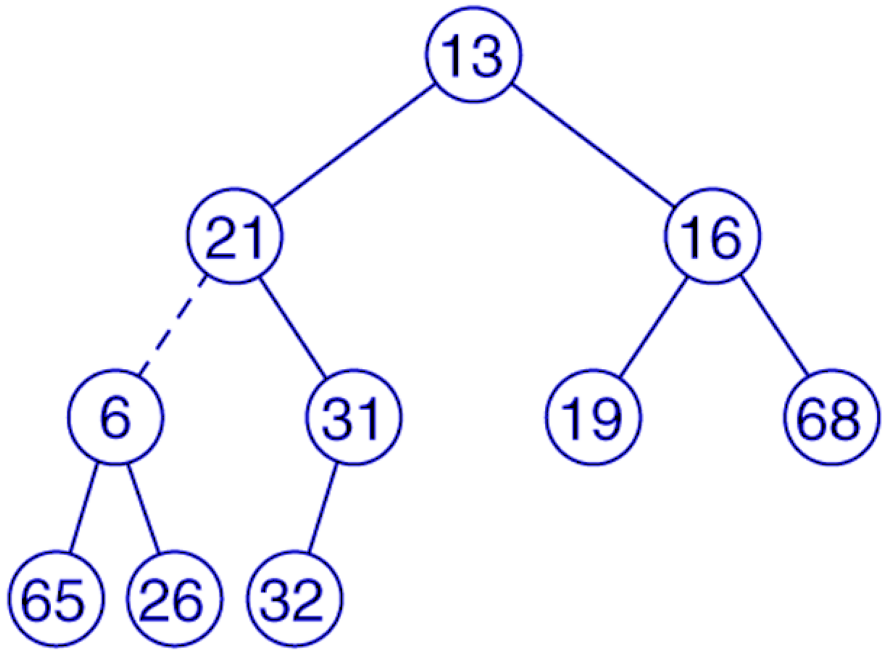


Figure: Max heap example

Two complete trees



(a)



(b)

(a) a heap; (b) not a heap

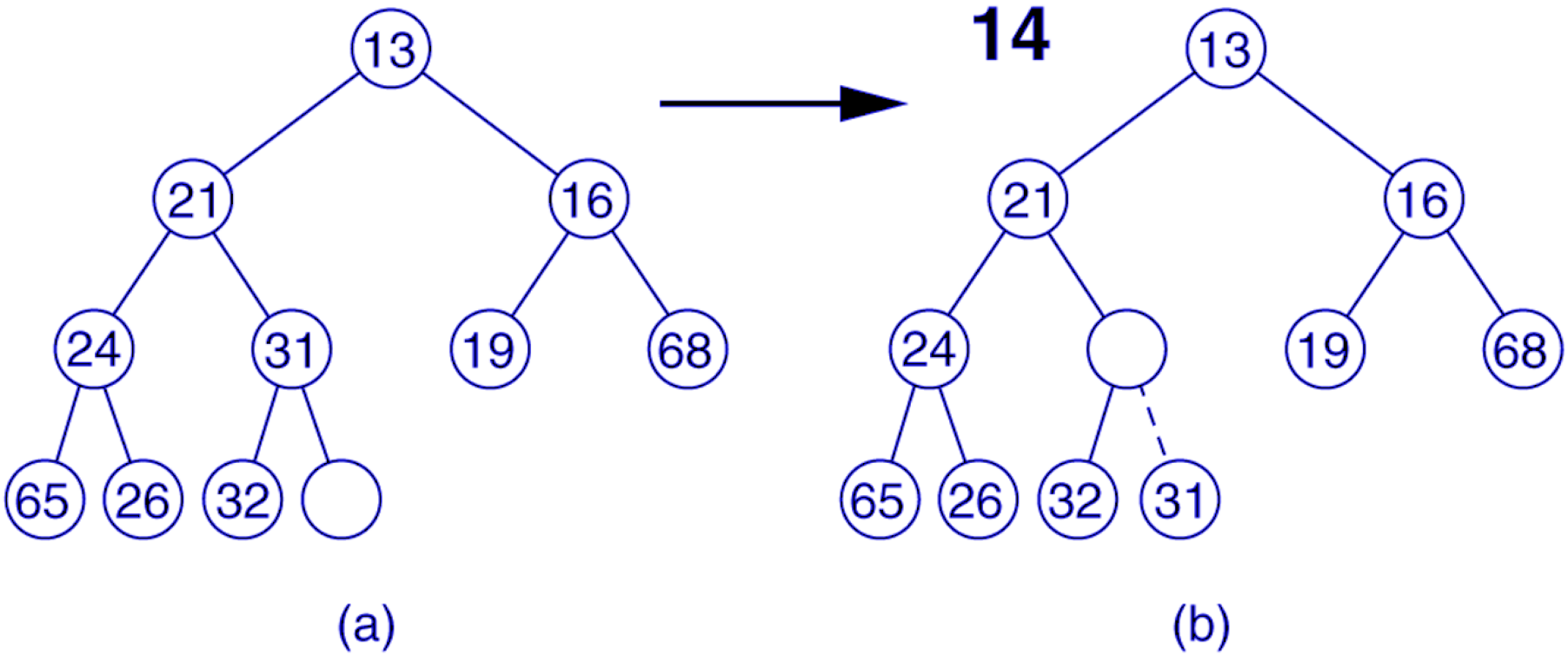
Binary Heap Class

```
public class BinaryHeap<AnyType extends Comparable<? super AnyType>>
{
        public BinaryHeap( ){...}
        public BinaryHeap( int capacity ){...}
        public BinaryHeap( AnyType [ ] items ){...}

        public void insert( AnyType x ){...}
        public AnyType findMin( ){...}
        public AnyType deleteMin( ){...}
        public boolean isEmpty( ){...}
        public void makeEmpty( ){...}

        private static final int DEFAULT_CAPACITY = 10;
        private int currentSize; // Number of elements in heap
        private AnyType [ ] array; // The heap array

        private void percolateDown( int hole ){...}
        private void buildHeap( ){...}
        private void enlargeArray( int newSize ){...}
}
```

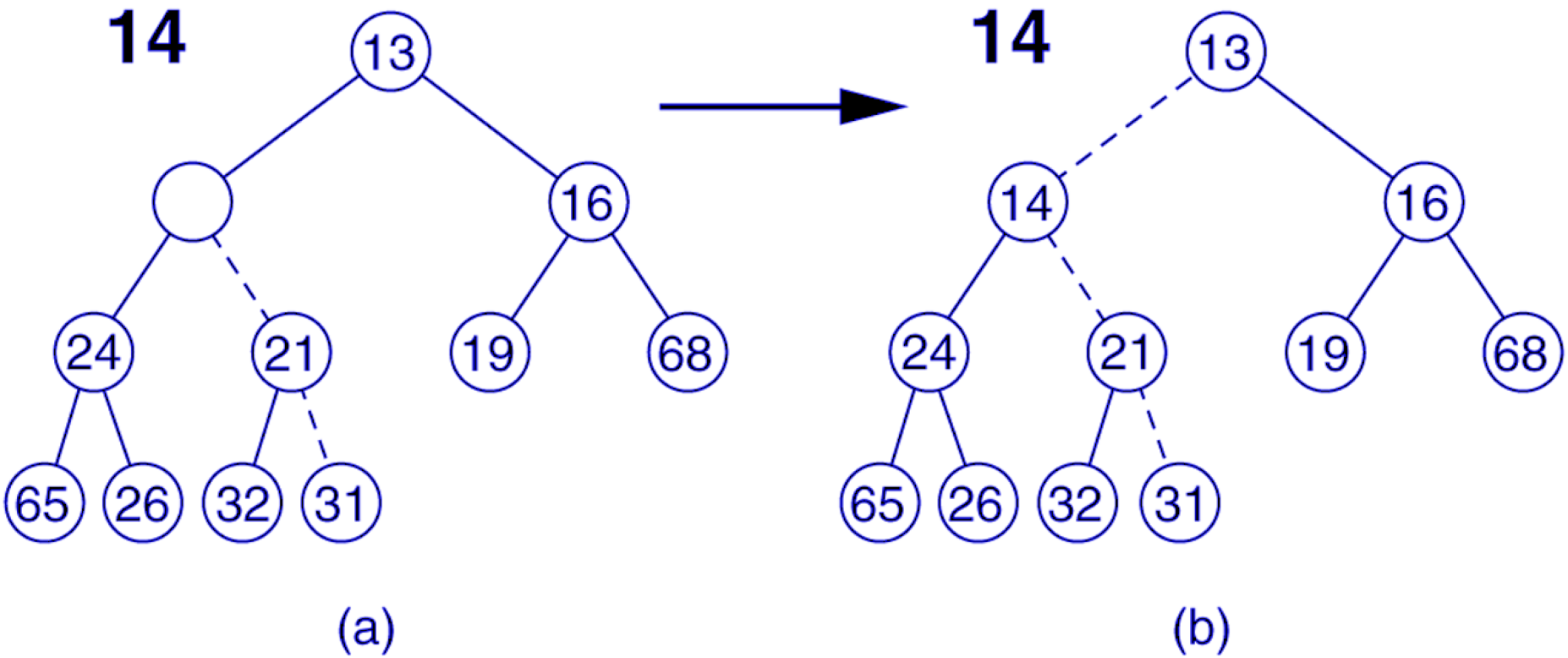
Basic Heap Operations: `Insert`

- To insert an element X into the heap:
  - We create a hole in the next available location.
  - If X can be placed there without violating the heap property, then we do so and are done.
  - Otherwise
    - We bubble up the hole toward the root by sliding the element in the hole's parent down.
    - We continue this until X can be placed in the hole.
- This general strategy is known as a percolate up.

Attempt to insert 14, creating the hole and bubbling the hole up



The remaining two steps required to insert 14



Insert procedure

```
/**
 * Insert item x, allowing duplicates.
 */
public void insert( AnyType x )
{
    if( currentSize == array.length - 1 )
        enlargeArray( array.length * 2 + 1 );

    // Percolate up
    int hole = ++currentSize;

    for(array[0] = x; x.compareTo(array[hole/2]) < 0; hole/=2)
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Why do we assign x to the position 0?

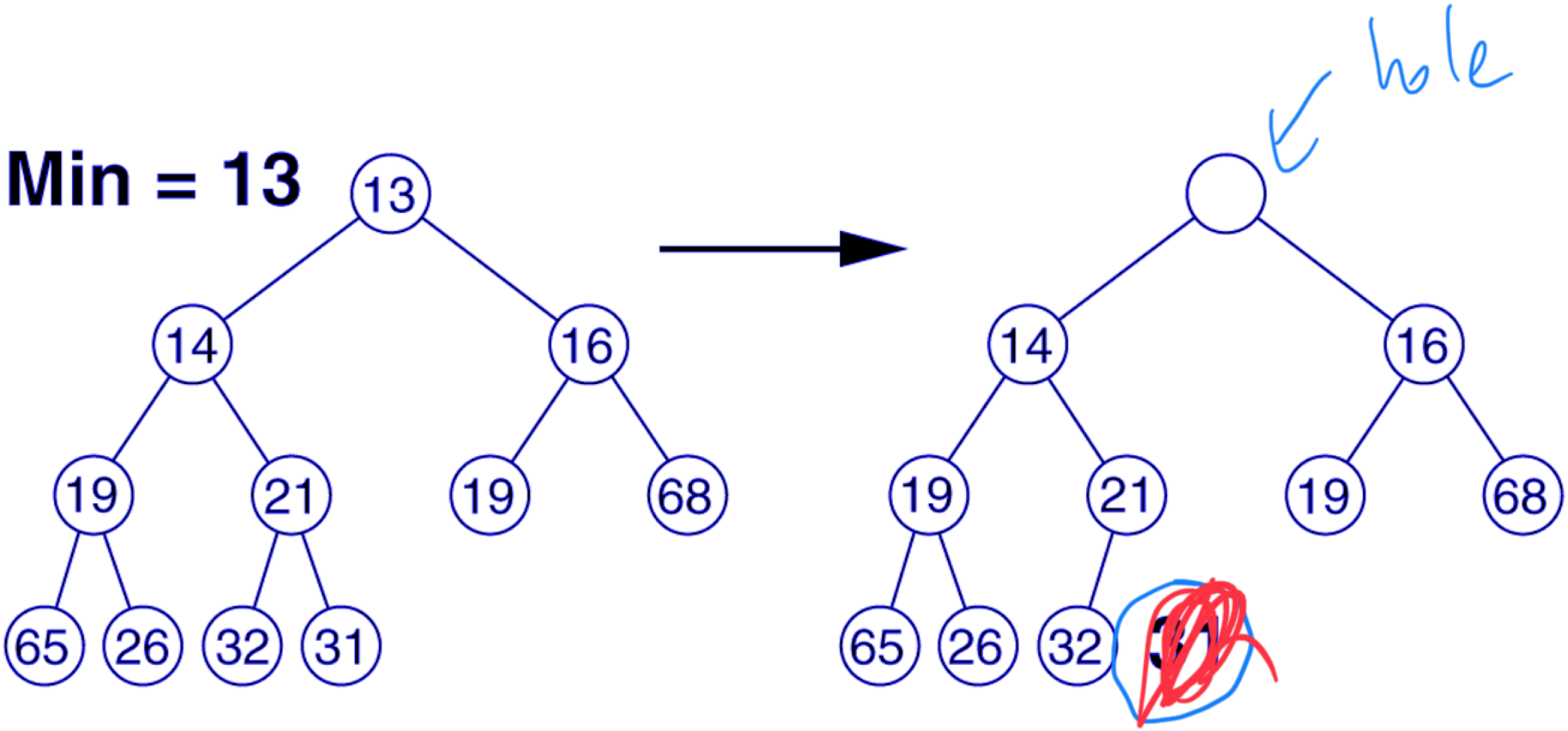
Analysis of `Insert`

- The time to do insertion could be as much as $O(logN)$ if the inserted element is the new minimum.
- On average, the percolation up terminates early and average runtime for insert is constant.

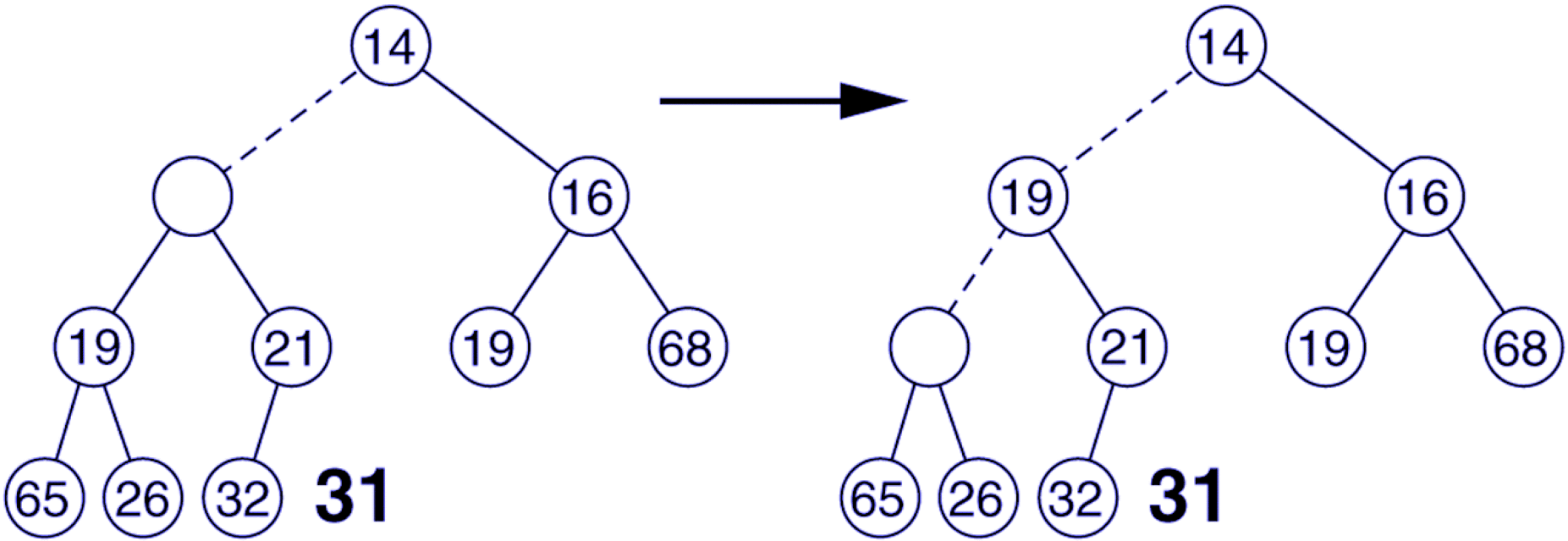
Delete Minimum

- `deleteMin` is handled in a similar manner as insertion:
- Remove the minimum; a hole is created at the root.
- The last element X must move to somewhere in the heap.
  - If X can be placed in the hole then we are done.
  - Otherwise,
    - We slide the smaller of the hole's children into the hole, thus pushing the hole one level down.
    - We repeat this until X can be placed in the hole.

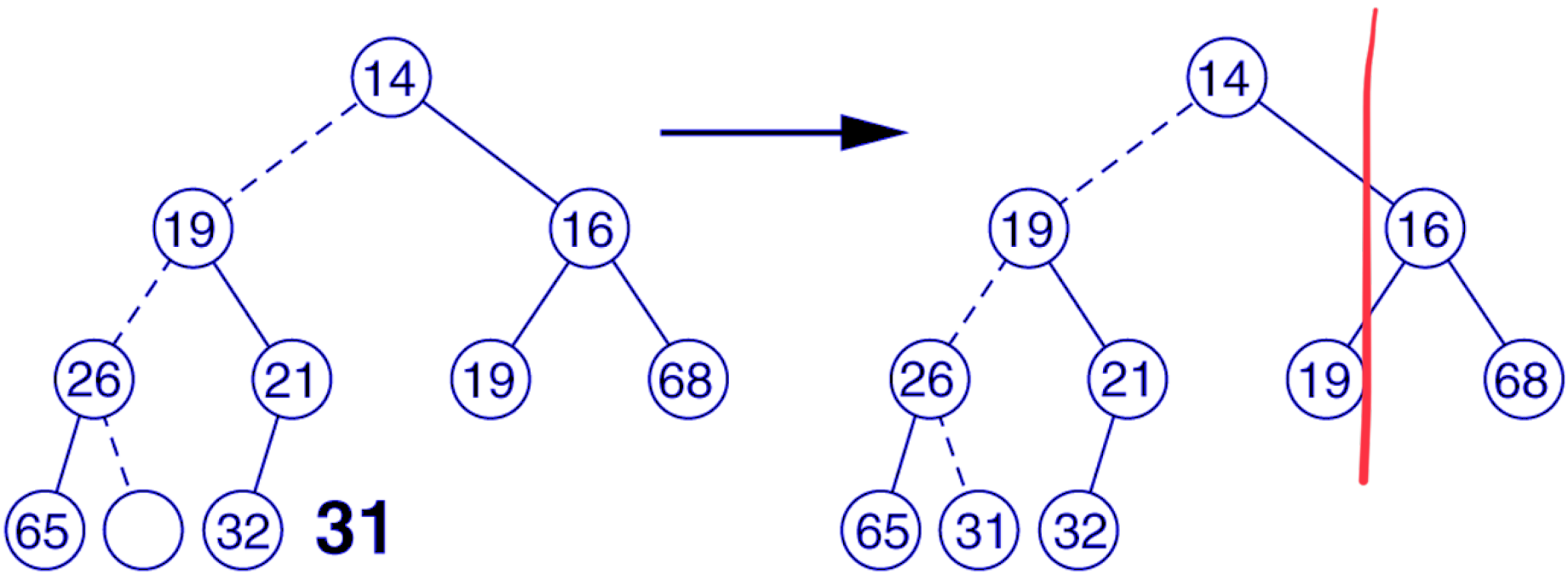
Creation of the hole at the root



The next two steps in the `deleteMin` operation



The last two steps in the `deleteMin` operation



deleteMin procedure

```
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
public AnyType deleteMin( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    AnyType minItem = findMin( ); // Array location 1

    array[ 1 ] = array[ currentSize-- ];

    percolateDown( 1 );

    return minItem;
}
```

percolateDown procedure

```
//**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */
private void percolateDown( int hole )
{
    int child;
    AnyType tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if(child != currentSize && array[child+1].compareTo(array[child]) < 0)
            child++;
        if( array[child].compareTo( tmp ) < 0 )
            array[hole] = array[child];
        else
            break;
    }
    array[ hole ] = tmp;
}
```

Analysis of Delete Minimum

- The worst case running time for `deleteMin` is $O(logN)$
- On average, the element placed at the root is percolated down almost to the bottom of the heap (its original level), so the average running time is $O(logN)$

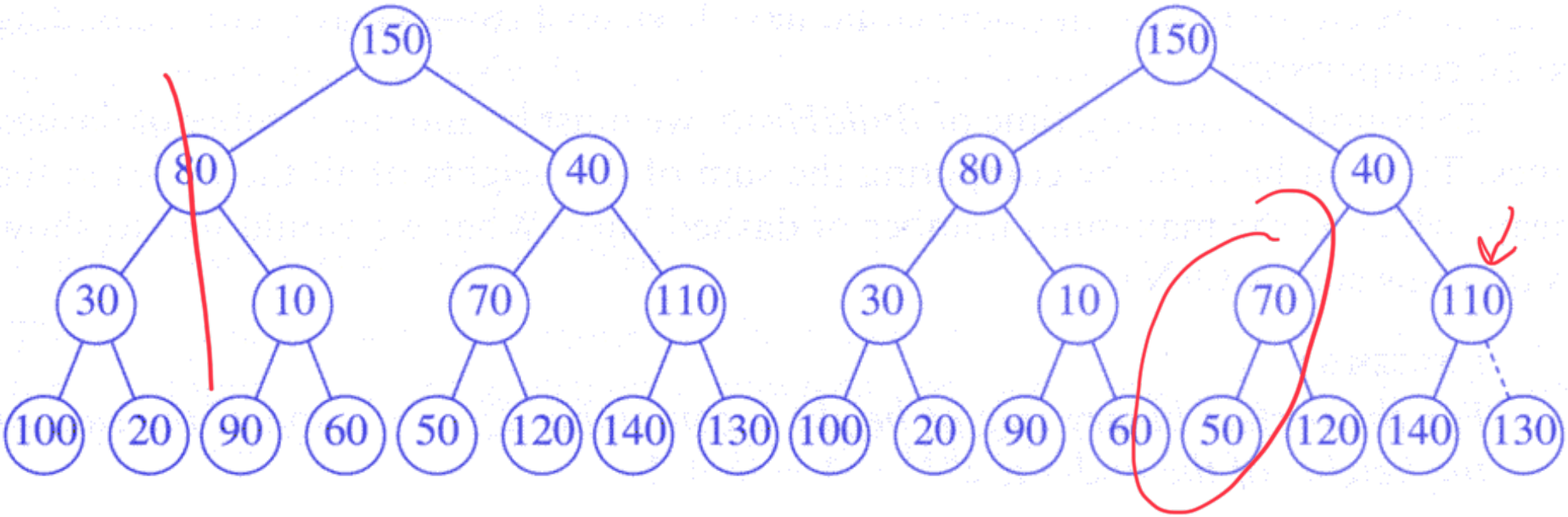
Building a Heap

- Take as input N items and place them into an empty heap.
- Obviously this can be done with N successive inserts: $O(NlogN)$ worst case, $O(N)$ on average.
- However `buildHeap` operation can be done in linear time ($O(N)$) by applying a percolate down routine to nodes in reverse level order.

buildHeap method

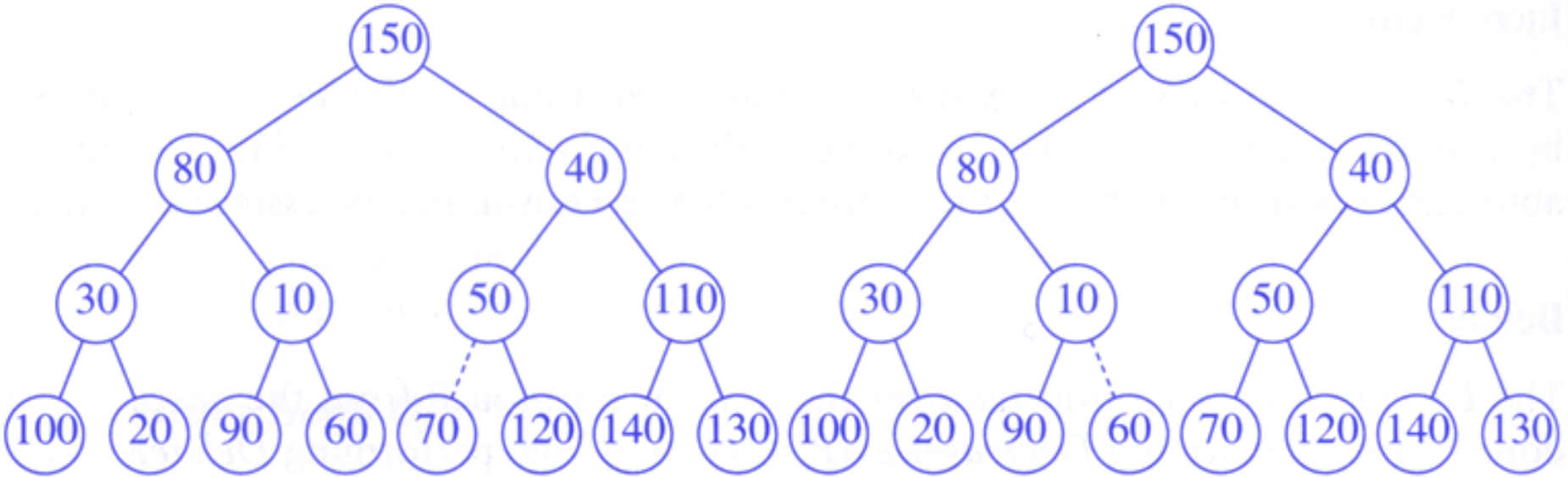
```
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

Implementation of the linear-time `BuildHeap` method



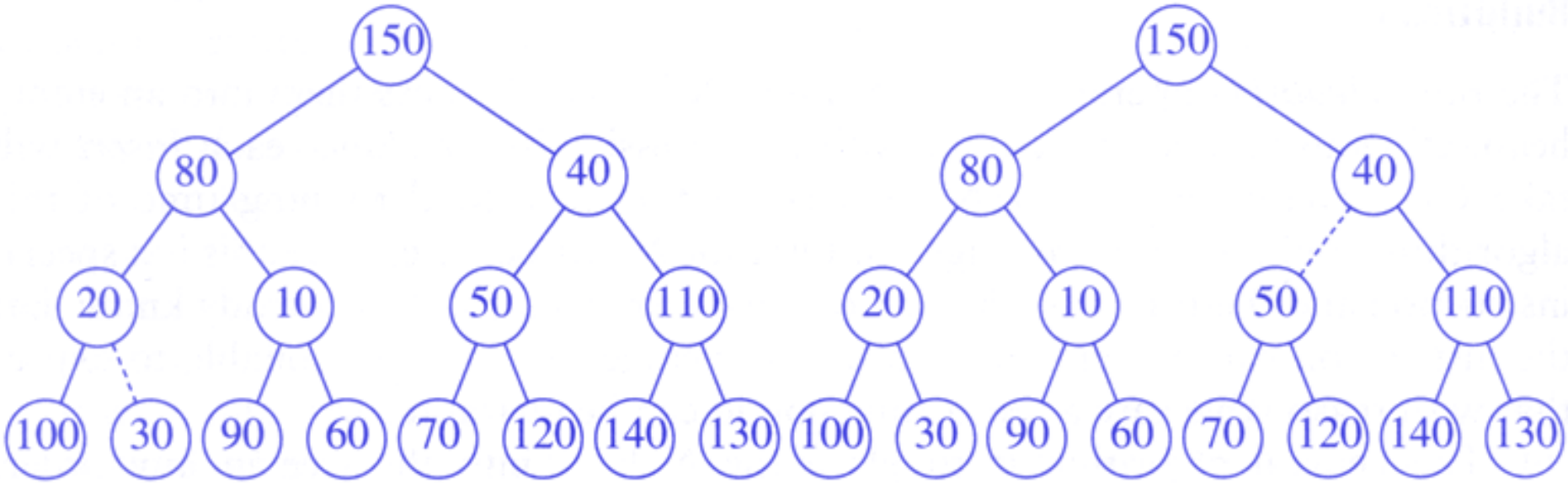
Initial heap After `percolateDown(7)`

`BuildHeap` (Cont.)



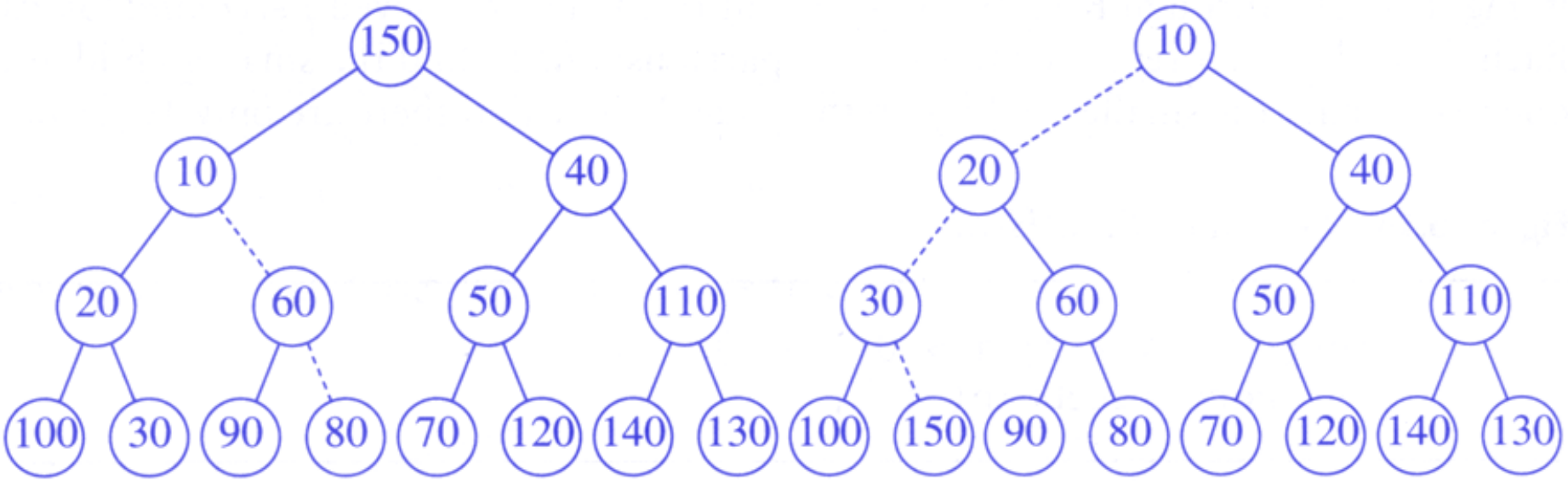
(a) After `percolateDown(6);` (b) after `percolateDown(5)`

`BuildHeap` (Cont.)



(a) After `percolateDown(4);` (b) after `percolateDown(3)`

`BuildHeap` (Cont.)



(a) After `percolateDown(2);` (b) after `percolateDown(1)` and `BuildHeap` terminates

Analysis of `BuildHeap`

- The linear time bound of `BuildHeap`, can be shown by computing the sum of the heights of all the nodes in the heap
- For the perfect binary tree of height H containing $N = 2^{H+1} - 1$ nodes, the sum of the heights of the nodes is $N - H - 1$.
- Thus it is $O(N)$.
- 1 node at height H
- 2 nodes at height $H - 1$
- 2^2 nodes at height $H - 2$ ...
- $S = \sum_{i=0}^{H} 2^i (H - i)$

Some Applications of Priority Queues

- Operating system design
- Implementation of several graph algorithms efficiently
- The selection problem (and sorting)
- Discrete Event Simulation

Selection Problem

- Given a list of N elements, and an integer k, the selection problem is to find the kth smallest element.
  - Read N elements into an array, apply *buildHeap* algorithm to this array. Then perform k *deleteMin* operations. The last element extracted from the heap is the answer.
  - The runtime is $O(N + klogN)$.
  - Note that if we run the algorithm for $k = N$ and record the values as they leave the heap, we will have sorted list that will run in $O(NlogN)$ time. This is known as *heapsort*.
  - We can find the kth largest element using max heap.

Discrete Event Simulation

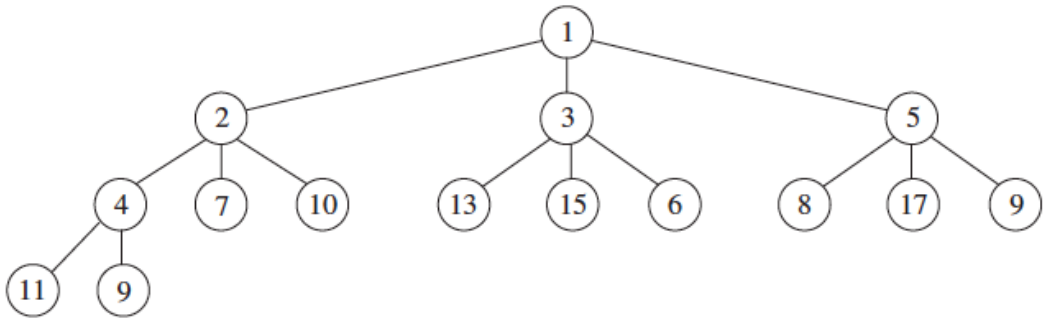
- For some queuing problems analytical analysis is complex hence simulation might be needed.
- In our simulations, we can use a quantum unit which we will refer as *tick*. One option is to start simulation clock at zero ticks and advance the clock one tick at a time to see if there is an event.
  - The problem is that runtime does not depend on customers or events but number of ticks, which is not part of the input.
- Instead, we can advance the clock to the next event time at each stage. To find the events that are nearest in the feature, it is appropriate to keep them in a priority queue.

DES Example:Emergency Room Simulation

- Patients arrive at time t with injury of criticality C
  - If no patients are waiting and there is a free doctor, assign them to doctor and create a future departure event; else put patient in the Criticality priority queue
- Patient departs at time t
  - If someone in Criticality queue, pull out most critical and assign to doctor; create a future departure event

d-Heaps

- A generalization which is exactly like a binary heap except that all nodes have *d* children, hence a binary heap is *2-heap*



- The tree will be shallower, hence inserts will be improved compared to binary heap.
- Due to $d - 1$ comparisons among the children, deleteMin will be more expensive.
- Unless *d* is a power of 2, multiplications and divisions to find children and parents will be much more expensive (can not be done by bit shift).

Leftist Heaps and Skew Heaps and Binomial Queues

- Any array based implementation of priority queue will have $\Theta(N)$ to **merge** (combine two heaps into one) two equal sized heaps.
- For efficient implementation of merge, we need linked data structure. Note that this will make all other operations slower.
- Leftist Heaps, Skew Heaps and Binomial Queues are linked based priority queue implementations that support efficient merge operation.

Leftist Heap

Null Path Length Definition

We define the null path length, npl(X), of any node X to be the length of the shortest path from X to a node without two children. Thus, the npl of a node with zero or one child is 0, while npl(null) = -1.

Leftist Heap Property

The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.

Observation

- Leftist heaps tend to have deep left paths
- In contrast to a binary heap (which is always a complete binary tree), a leftist tree may be very unbalanced.
- The general idea for the leftist heap operations is to perform all the work on the right path, which is guaranteed to be short

Leftist Heap Example

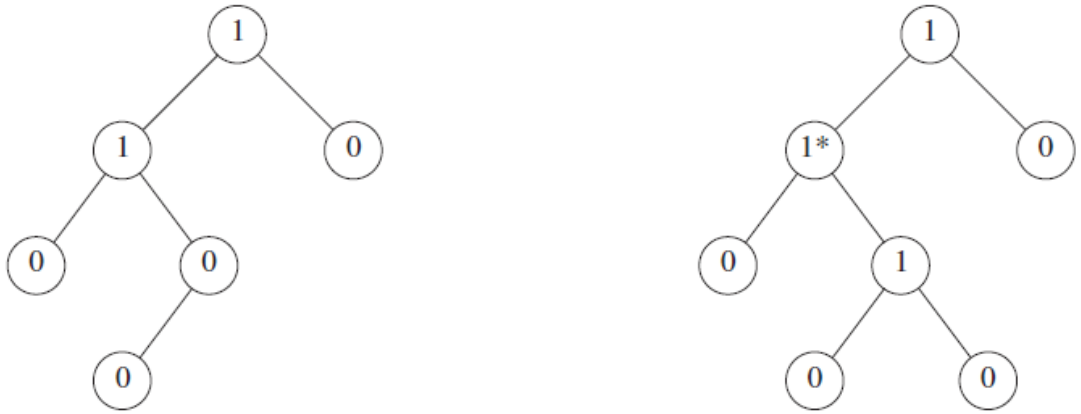


Figure 6.20 Null path lengths for two trees; only the left tree is leftist

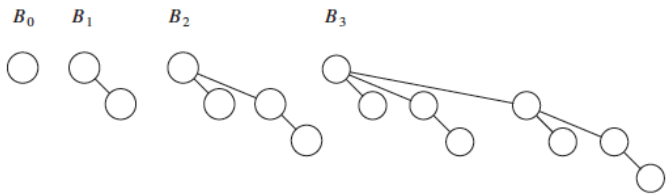
- Heavier on left side : npl(right(i)) <= npl(left(i)).
- The shortest path to a descendant node is through the right child. Every subtree is also a leftist tree and npl(i) = 1 + npl(right(i)).
- The path from root to rightmost leaf is the shortest path from root to a leaf.
- If the path to rightmost leaf has x nodes, then leftist heap has at least 2^x-1 nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

Leftist Heap Merge Algorithm

Visualization at: https://www.geeksforgeeks.org/leftist-tree-leftist-heap/
Merge can be performed in O(logN) time

Binomial Queues

- A binomial queue is not a heap-ordered tree but a collection of heap-ordered trees, known as **forest**.
- Each of the heap-ordered trees is of a constrained form known as a binomial tree
- There is at most one binomial tree of every height.
- A binomial tree of height 0 is one node tree. Binomial tree, B_k of height k is constructed by attaching a binomial tree B_{k-1} to the root of another binomial tree B_{k-1}.
- In other words, a Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one the rightmost child or the other (leftmost)



Binomial Queue Construction

Binomial trees of height k have exactly 2^k nodes.

If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can represent a priority queue of any size by a collection of binomial trees.

For instance, a priority queue of size 13 could be represented by the forest B3, B2, B0.

Advantage:

Merging easy with binomial queues in O(logN) time compared to O(N) binary min heap.

Binomial Queue Example