Heaps

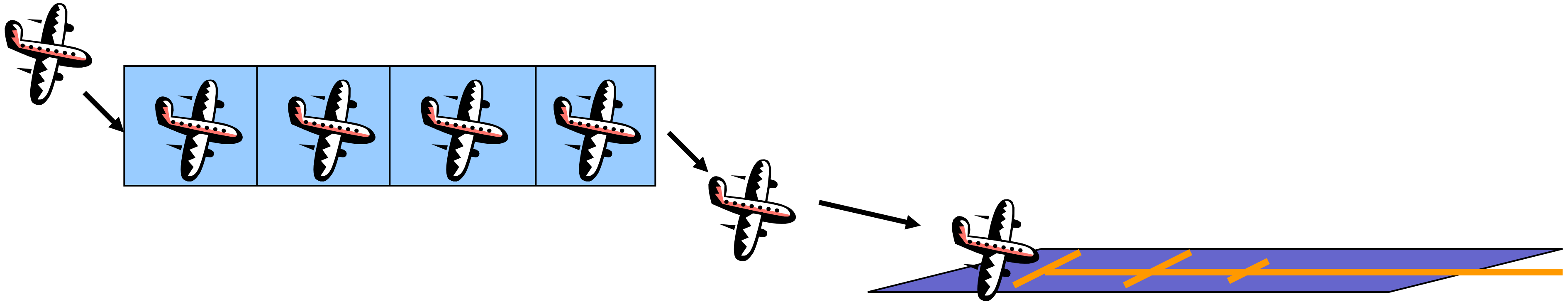
School of Computer Science and Informatics
University College Dublin, Ireland

Priority Queues: Concept

- A priority queue is an abstract data type for storing a collection of prioritized elements.
  - Like a queue but, removal is based on a **priority**

- Terminology:
  - Priority Queues hold **entries** (key + value) – the key is the priority
  - An object can be inserted at any time with any priority
  - Only the element with the highest priority can be removed

- Example: Airport landing schedule



Priority Queues: Funct. Spec.

- Priority Queues work with two objects:
  - a data object
  - A key (priority) object

- Core Operations:

insert(k,e)	Insert a new element e with key k into P
min()	Return (but don't remove) an element of P with highest priority; error occurs if P empty.
remove()	Remove from P and return an element with the highest priority; an error occurs if P is empty.

- Support Operations:

isEmpty()	Returns true if the priority queue is empty, or false otherwise
size()	Returns the number of elements in the priority queue

Priority Queue: Java Interfaces

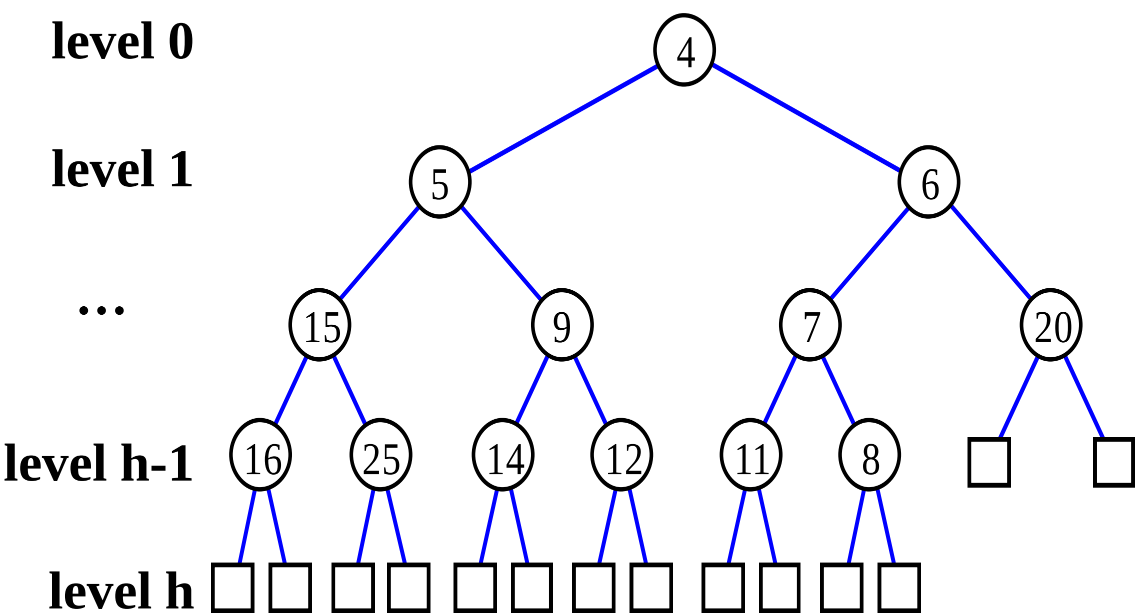
```
public interface PriorityQueue<K,V> {
    public int size();
    public boolean isEmpty();
    public void insert(K key, V value);
    public V min();
    public V remove();
}
```

Priority Queue Impl. Strategies

- List-Based Strategies:
  - Unsorted List:
    - Insert at end of the list
    - Use linear search to find which item to remove
  - Sorted List:
    - Insert based on priority (like insertion sort)
- Both strategies have O(n) performance on some operations
  - Unsorted List: min() and removeMin()
  - Sorted List: insert()
- Sorted Lists are “better” because they only affect one operation

Heaps

- A **heap** is a binary tree that stores keys (key-value pairs) at its internal nodes and that satisfies two additional properties:
  - **Order Property**: key(parent) ≤ key(child)
  - **Structural "Completeness" Property**: all levels are full, except the bottom, which is "left-filled"

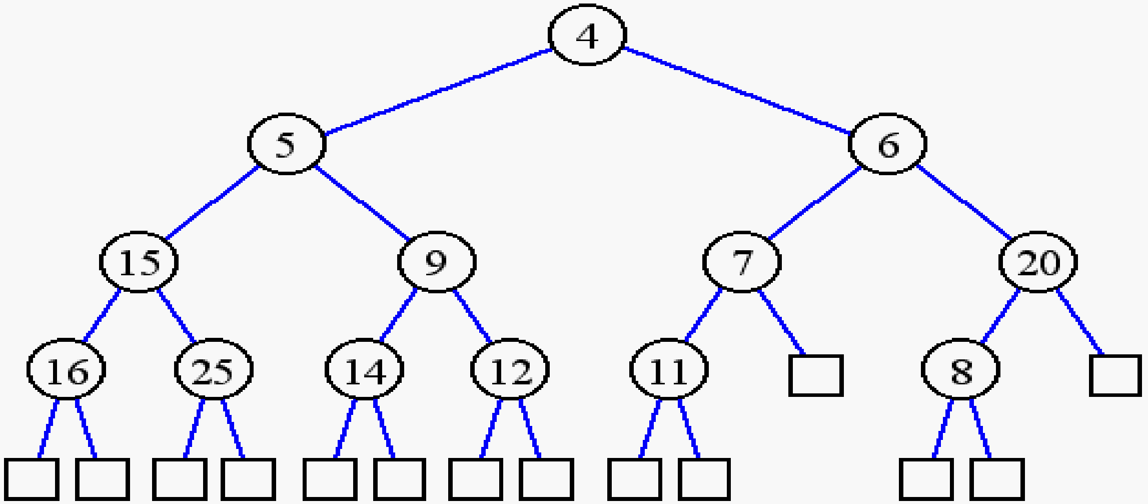


Rule 1. Level i has 2^i nodes, for $0 \le i < h$

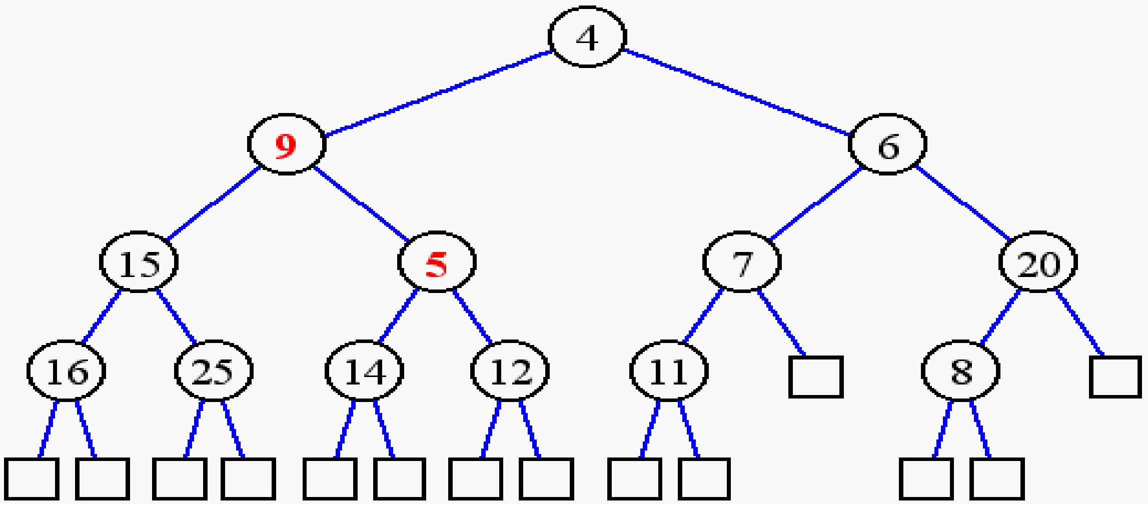
Rule 2. At level h-1, all leafs are to right of all internal nodes

Examples that are *NOT* heaps

- bottom level is not left-filled



- key(parent)>key(child)

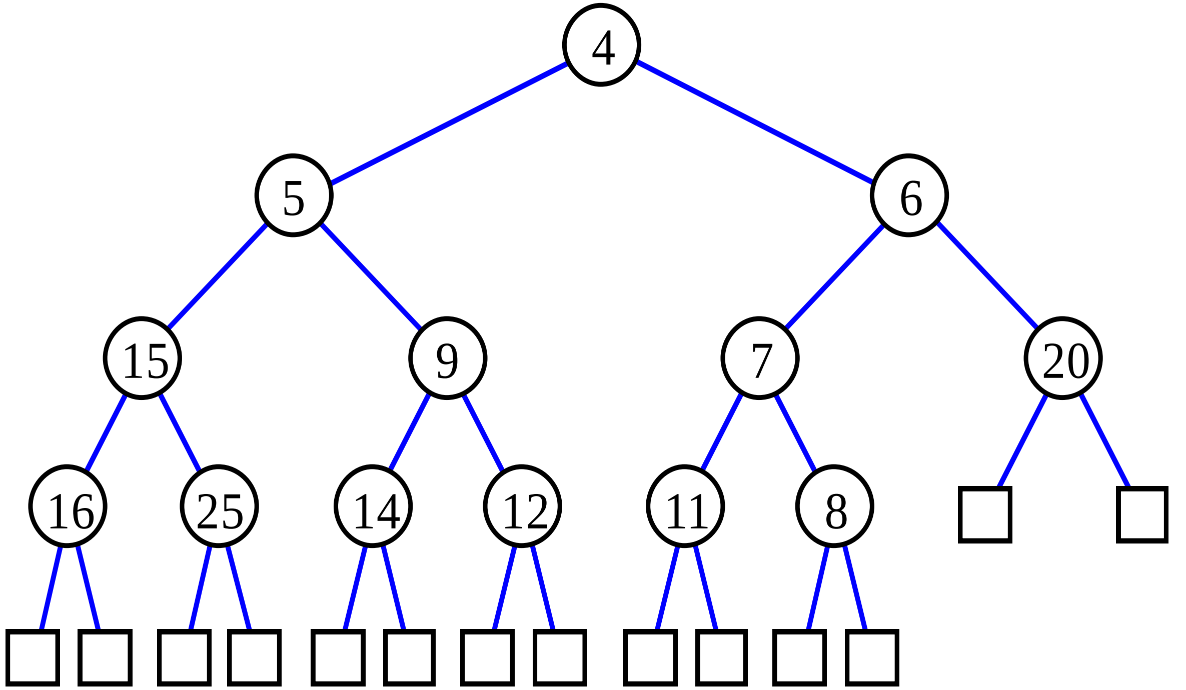


A useful fact

- A heap storing N keys has height

$h = \lceil \log(N+1) \rceil$ $\lceil x \rceil$ = smallest integer greater than x

$h = \lceil \log(13+1) \rceil = \lceil 3.8074... \rceil = 4$



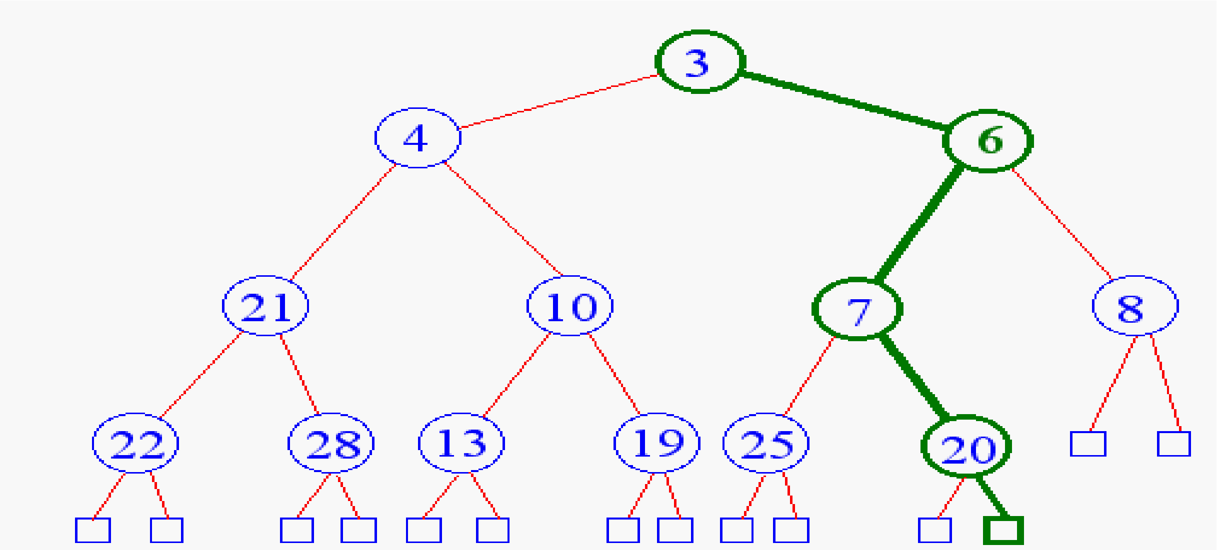
Inserting into Heaps

- To insert an entry into a heap, we must perform three steps:
  1. **Insert the new entry at the last position in the heap.**
  2. **Upheap to restore the “order property”.** This involves repeatedly swapping the newly added entry with parent entries until the property is restored.
  3. **Identify the “new” last position in the heap.** This involves finding the next position in an in-order traversal of the tree. If no next position exists, then the new “last” position is the leftmost child on the next level down.

- **Example**: Add the following numbers into a heap:

  5, 30, 12, 23, 26, 3, 6, 15, 18, 21, 9, 29, 33

Inserting into Heaps



- Upheap terminates when new value is greater than the value of its parent, or the top of the heap is reached.

- **Note**: after Upheap the heap is again valid!

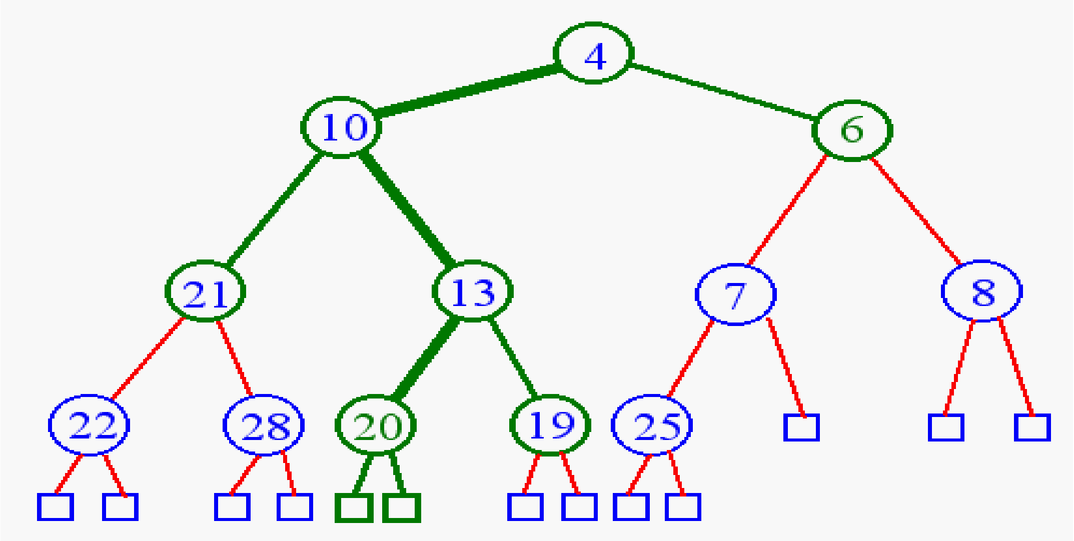
- Complexity Analysis:

  cost(insert) = (total#swaps) $\leq$ (h-1) = $\lceil \log_2(N+1) \rceil$ = **O(log n)**

Removing from Heaps

- To remove an entry from the heap, we must:
  1. **Remove the root node (this is the current min entry).**
  2. **Replace this node with the last item in the heap, node L.**
  3. **Perform a Downheap to re-establish the “ordering property”.** This involves checking whether L is less that its children. If it is not, then L is swapped with the smaller of the two children, and downheap is performed again on L in its new position.
  4. **Update the “last” reference.** Need to do an inverse pre-order traversal (I.e. goto the node that is before the last node in a pre-order traversal). If there is no previous node, then find the last node on the next level up.

Removing from Heaps



- Downheap terminates when the key is greater than the keys of both its children, or the bottom of the heap is reached.

- Note that after Downheap the heap is again valid!

- Complexity Analysis:

  cost(rMin) = (total#swaps) $\leq$ (h-1) = $\lceil \log_2(N+1) \rceil$ = O(log n)

Complete Binary Trees

- Heaps are a type of ***Binary Tree*** that is known as a ***Complete Binary Tree***.

- A Binary Tree T, with height h, is **complete** if:
  - Levels 0 to h-1 have the maximum number of nodes possible
  - In level h-1, all the internal nodes are to the left of the external nodes, and there is at most one node that has only a left child.

- When dealing with Complete Binary Trees, there are two important nodes:
  - The ***root*** node (the top of the tree)
  - The ***last*** node (the next insertion point for the tree).

- To support this, we introduce the ***CompleteBinaryTree ADT***.

The Complete Binary Tree ADT

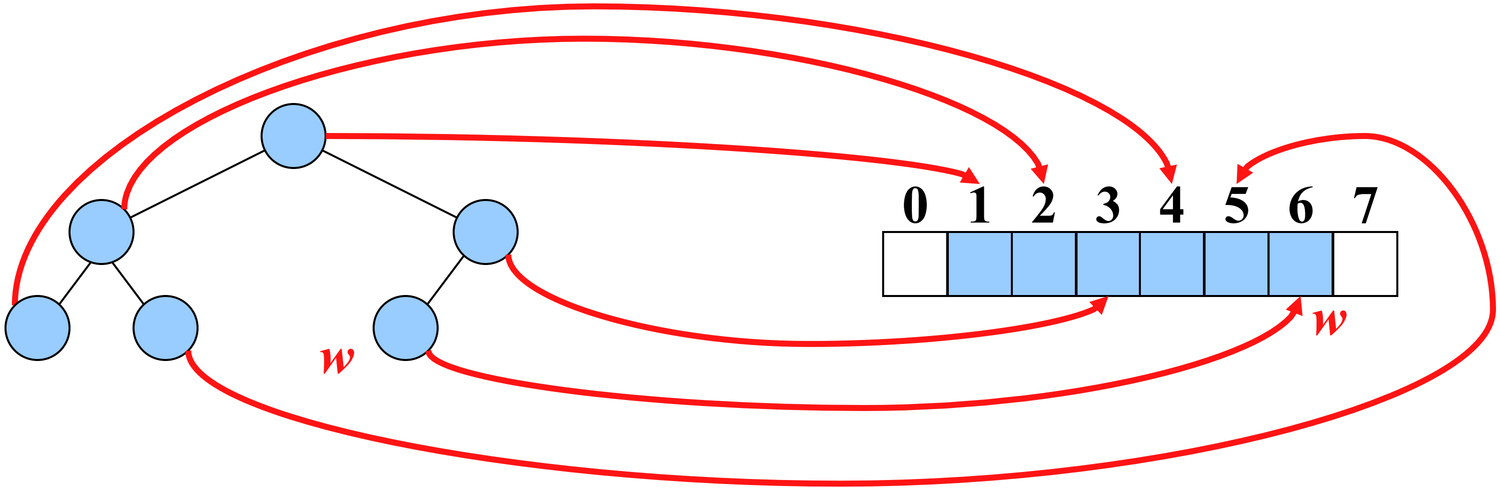
- A Complete Binary Tree (CBT) is a Binary Tree that also supports the following operations:
  - ***add(e)*** adds e to the tree and returns a new external node, v, such that the resulting tree is a complete binary tree with last node v.
  - ***remove()*** removes the last node of T and returns its element

- Java Interface:

```
interface CompleteBinaryTree<T> extends BinaryTree<T> {
        public Position<T> add(T elem);
        public T remove();
}
```

Implementing a CBT

- Vectors are a particularly suitable way of implementing Complete Binary Trees (CBT):
  - This is due to the structured way in which items are inserted into and removed from CBTs.
  - This is illustrated in the figure below…



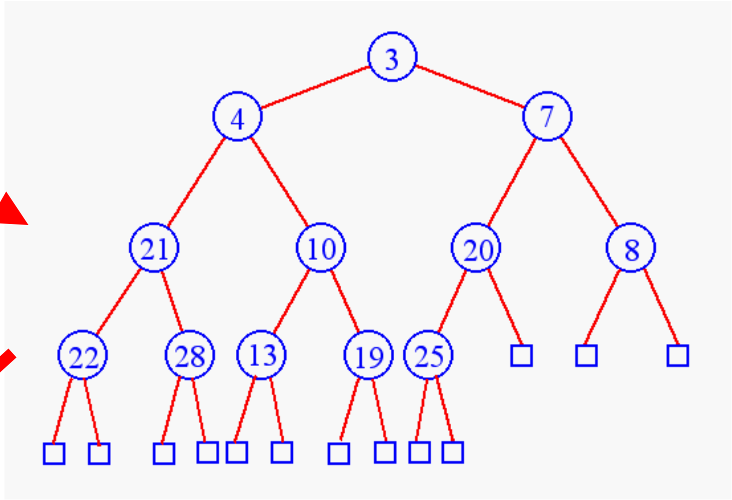
  - With a vector, the last node is the last item!

Heap Sort

- A simple PQ-based sorting algorithm called “heap sort”:
  - Insert all items to be sorted into a Priority Queue
  - Remove them; they’ll be in sorted order

[4,19,10,3,22,13,8,21,20,28,7,25]

cost = n×cost(insert) = n×O(log(n))



cost = n×cost(remove) = n×O(log(n))

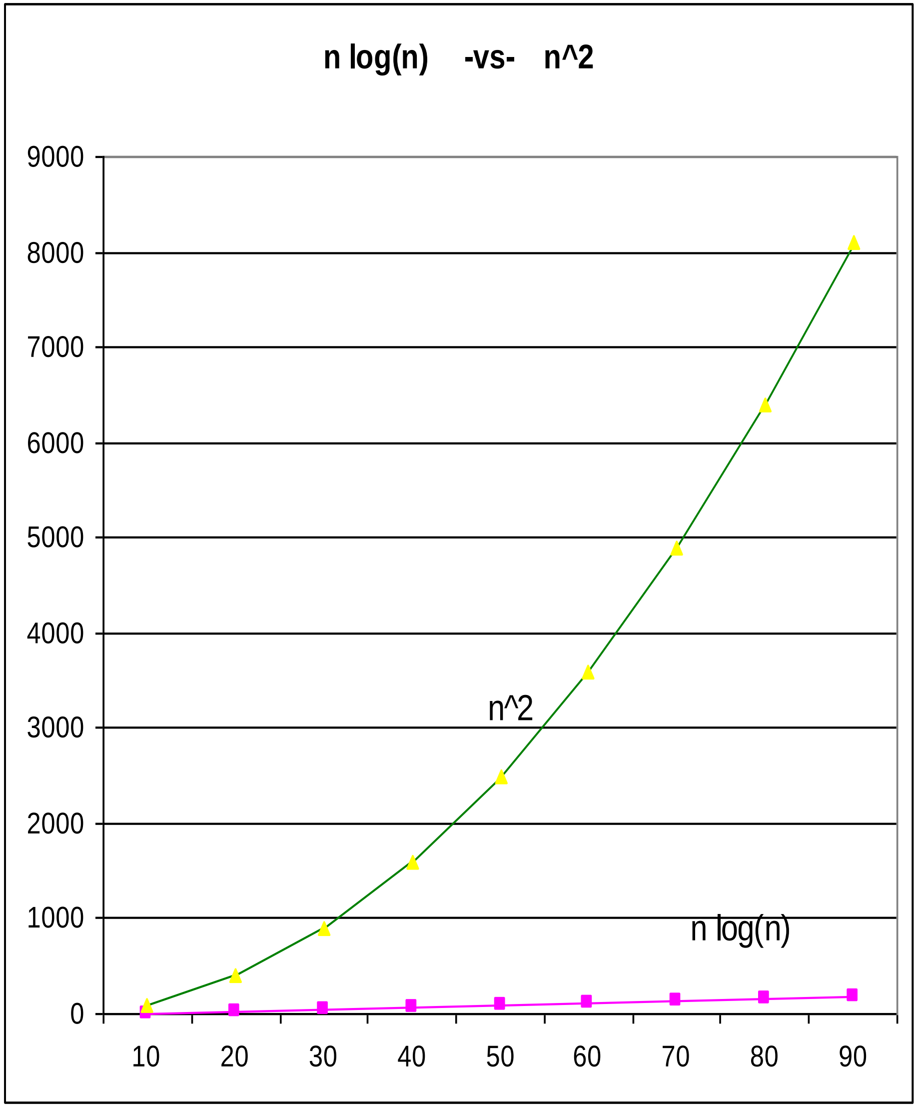
[3,4,7,8,10,13,19,20,21,22,25,28]

total cost = 2×n×O(log(n)) = O(n × log(n))

Heap Sort - analysis

- All heap methods run in O(log(n)) time, and need n insertions & n removals.
  - Therefore total time is O(2n log(n)) = O(n log(n))

- Compare to the selection and insertion sort's which had O(n2) complexity

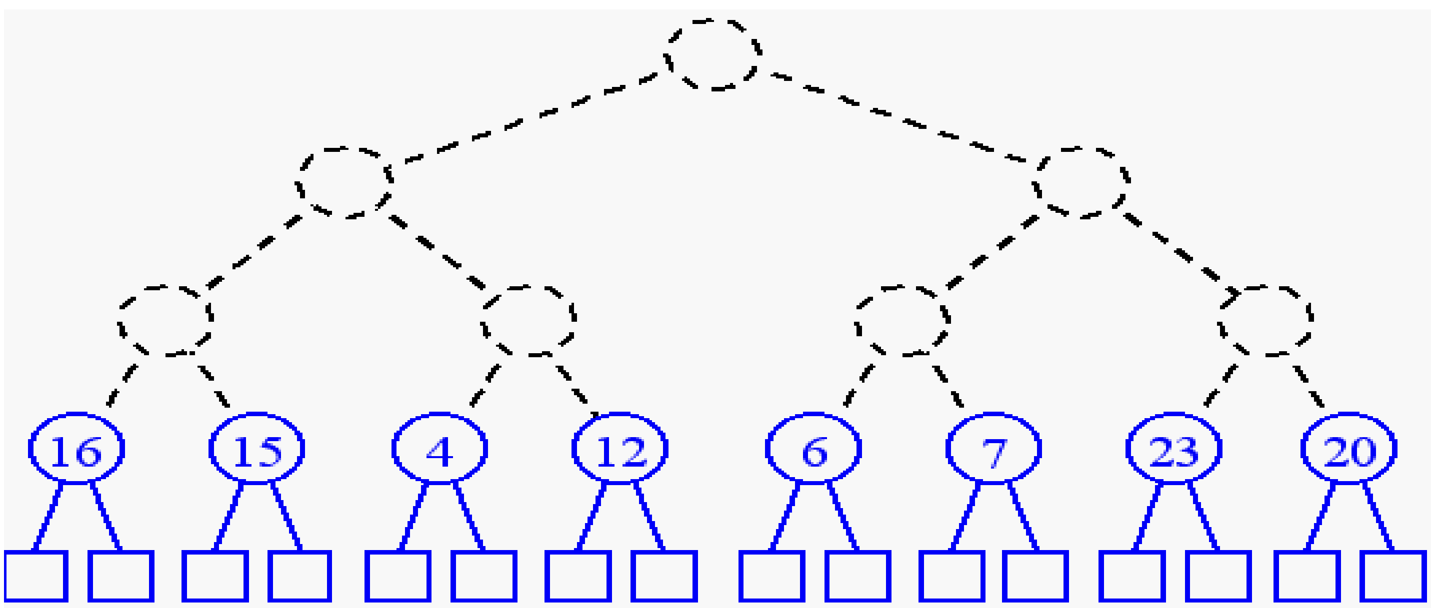


Bottom-Up Heap Construction

- Illustrate Bottom-Up heap construction by adding the following elements to a heap:

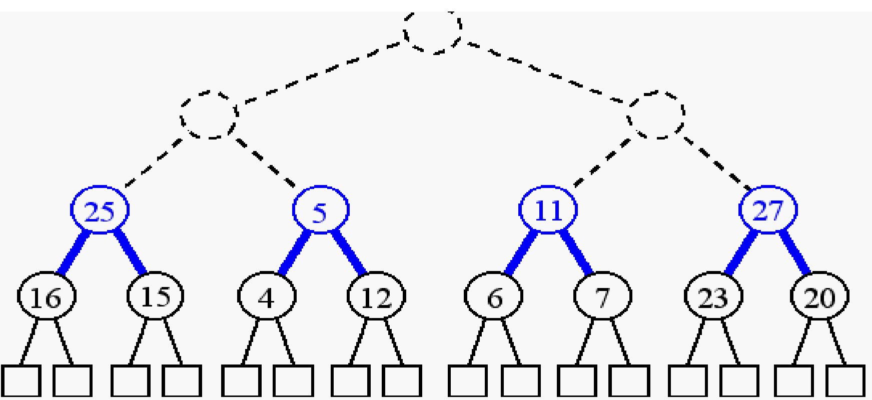
  [16,15,4,12,6,7,23,20,25,5,11,27,9,8,14]

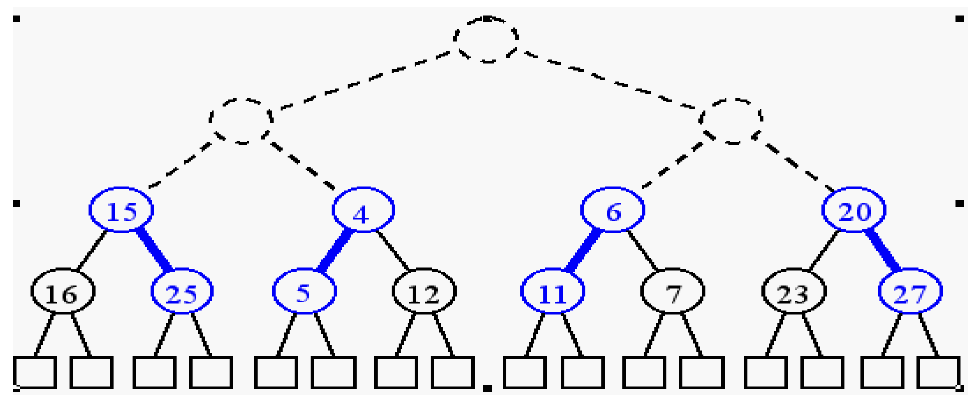
- Step 1: Construct (n + 1)/2 elementary heaps storing one entry each.



Bottom-Up Heap Construction

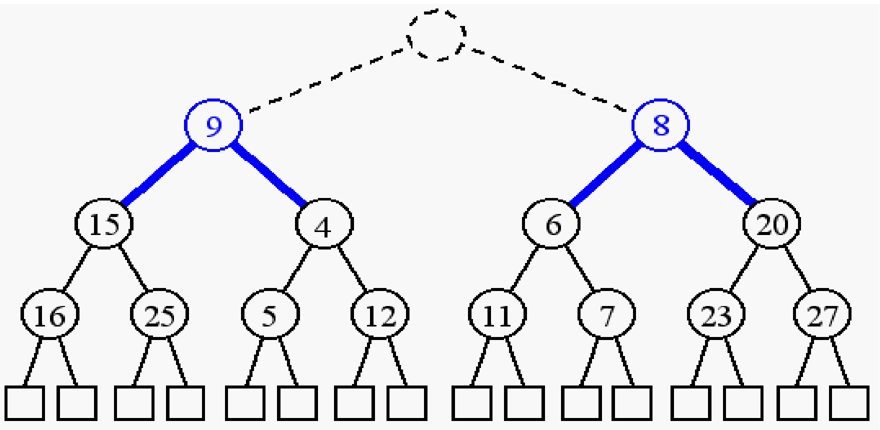
- Step 2: Construct (n + 1)/4 heaps, each storing 3 entries.
  - This is done by joining pairs of elementary heaps and adding a new entry placed at the root.
  - A downheap operation may be performed on the new root if necessary.

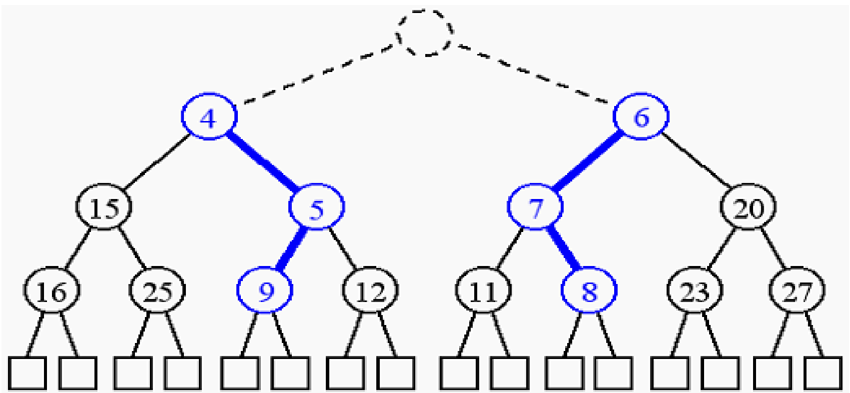




Bottom-Up Heap Construction

- Step 3: Construct (n + 1)/8 heaps, each storing 7 entries.
  - This is done by joining pairs of 3-entry heaps and adding a new entry placed at the root.
  - A downheap operation may be performed on the new root if necessary.

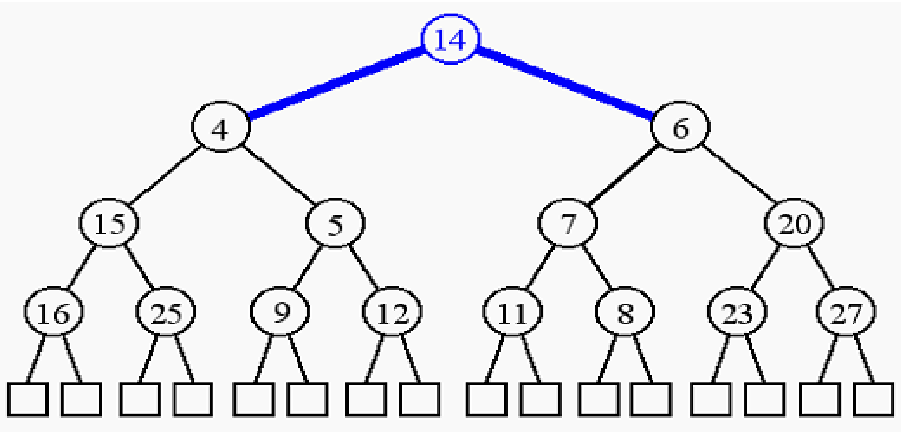


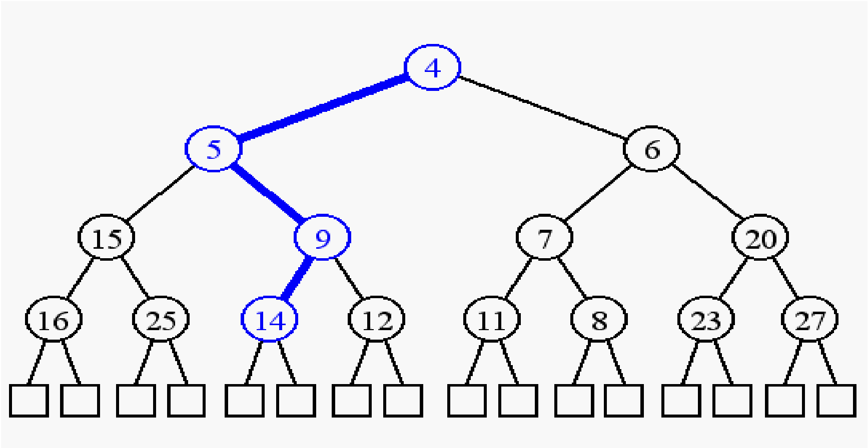


And so on…

Bottom-Up Heap Construction

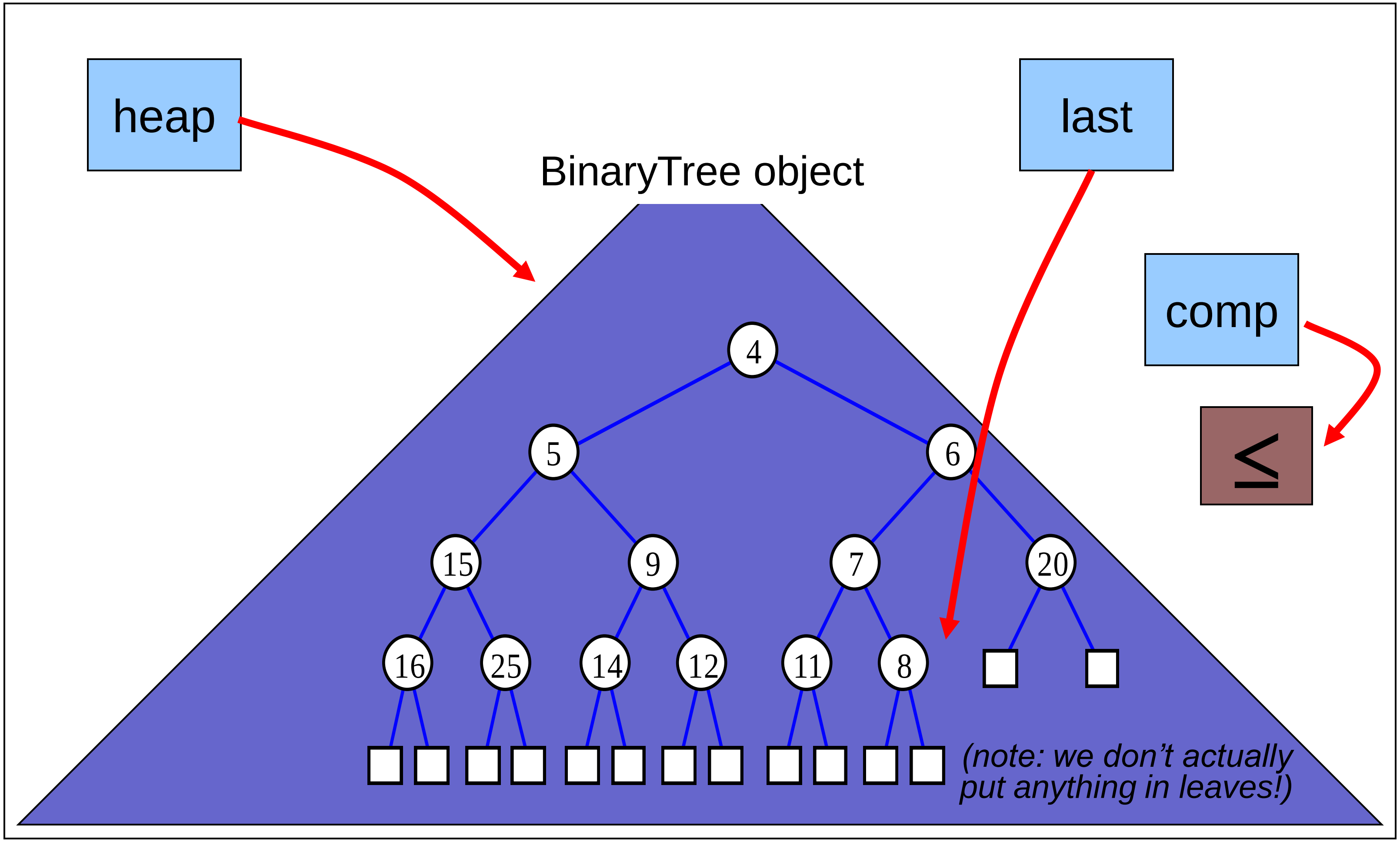
- Final Step: Construct (n + 1)/16 heaps, each storing 15 entries.
  - This is done by joining pairs of 7-entry heaps and adding a new entry placed at the root.
  - A downheap operation may be performed on the new root if necessary.





Using a heap to implement a PQ

an instance of the HeapPriorityQueue class



For simplicity… only keys are shown (elements not displayed)

keys are integers ordered by the ordinary "≤" relation

PQ Comparison

Operation	Heap	SortedSeq	UnsortedSeq
Size, isEmpty	O(1)	O(1)	O(1)
min	O(1)	O(1)	O(n) *ouch*
insert	O(log n)	O(n) *ouch*	O(1)
removeMin	O(log n)	O(1)	O(n) *ouch*
mixture of all operations	O(log n)	O(n)	O(n)