Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

Recap

- Binary Search Trees
- AVL Trees
- Red-black Trees

Today’s Objectives

- Priority Queues
- Binary Heap
- Heap-Sort

Priority Queues

Priority Queues

- Many applications require algorithms to process items in a specific order (e.g. relative importance)
    - Standby fliers
    - Patients waiting at a clinic
    - Operating system scheduling
- Priority can be based on anything relevant to the scenario (treated as the key)

Priority Queues

- Main operations

❖ **add(priority, value)**

❖ **peek()**

❖ **remove ()**

Priority Queues

- Possible implementations

❖ Unsorted List

❖ Sorted List

Unsorted List

- Insertion – O(1)
- Removal – O(n)

Sorted List

- Insertion – O(n)
- Removal – O(1)

Priority Queues

- There is one more way to implement priority queues

Heap or sometimes min/max heap

Heap Based Priority Queues

- Main operations

  insert(k, v) - inserts an item with key k (priority) and value v to the priority queue – the same as add

Heap Based Priority Queues

- Main operations

  insert(k, v) - inserts an item with key k (priority) and value v to the priority queue – the same as add

  min() or max() - returns the items with smallest or the largest key (highest priority) than any other key in the priority queue – the same as peek

Heap Based Priority Queues

- Main operations

  insert(k, v) - inserts an item with key k (priority) and value v to the priority queue – the same as add

  min() or max() - returns the items with smallest or the largest key (highest priority) than any other key in the priority queue – the same as peek

  removeMin() or removeMax() - removes the item from the priority queue whose key is the minimum or maximum (highest priority) – the same as remove

Heap Based PQs

- fast insertions - O(log n)
- fast removals - O(log n)

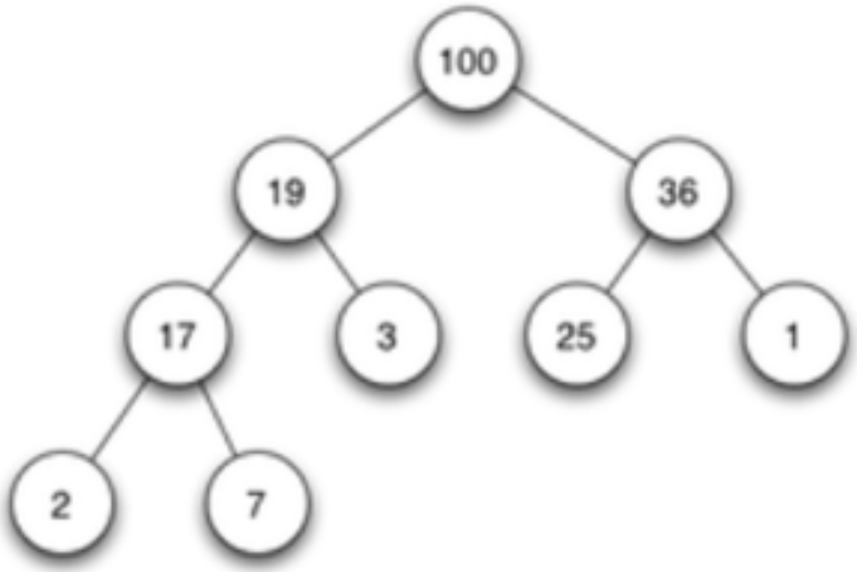
But first we must understand what is a Complete Binary Tree!

Complete Binary Tree

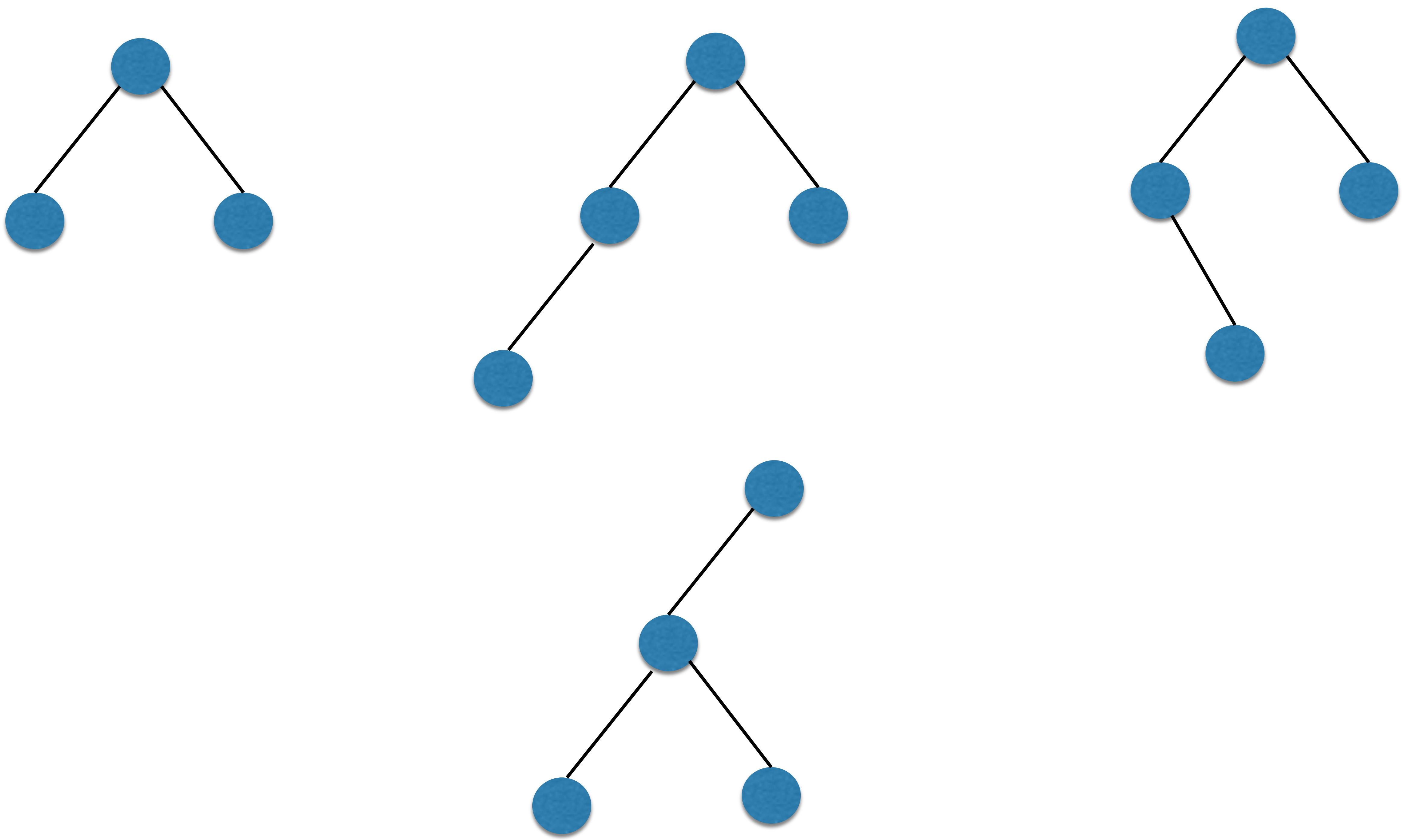
- A complete binary tree is
  - Filled out on every level, expect perhaps on the last one
  - All nodes on the last level, should be as far to left as possible

Complete Binary Tree

- A complete binary tree is
  - ➢ Filled out on every level, expect perhaps on the last one
  - ➢ All nodes on the last level, should be as far to left as possible



Complete Binary Tree



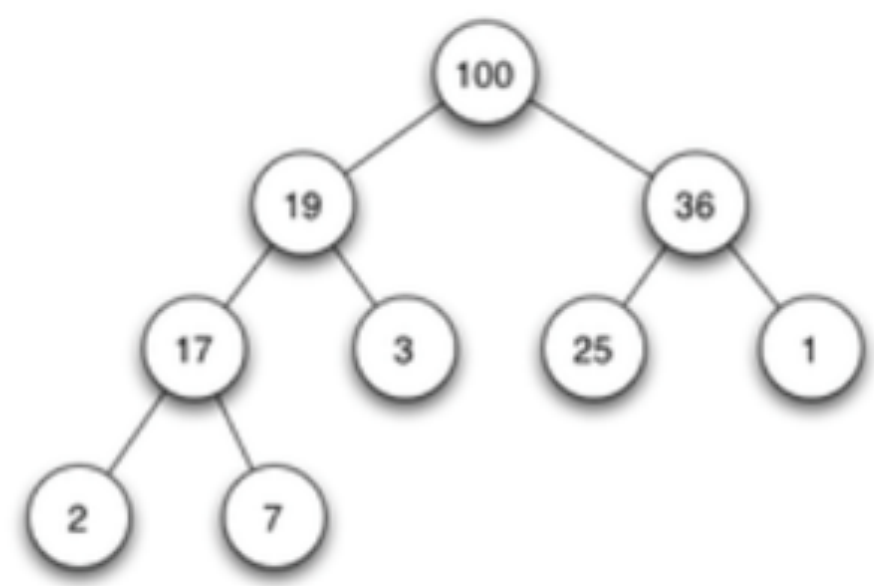
Binary Heap

1. Is a Complete Binary Tree

2. Maintains flexible order on the set of elements

   - Weaker than sorted order (& so it is efficient)

   - Stronger than random order (& so highest priority element can be quickly identified)



Binary Heap

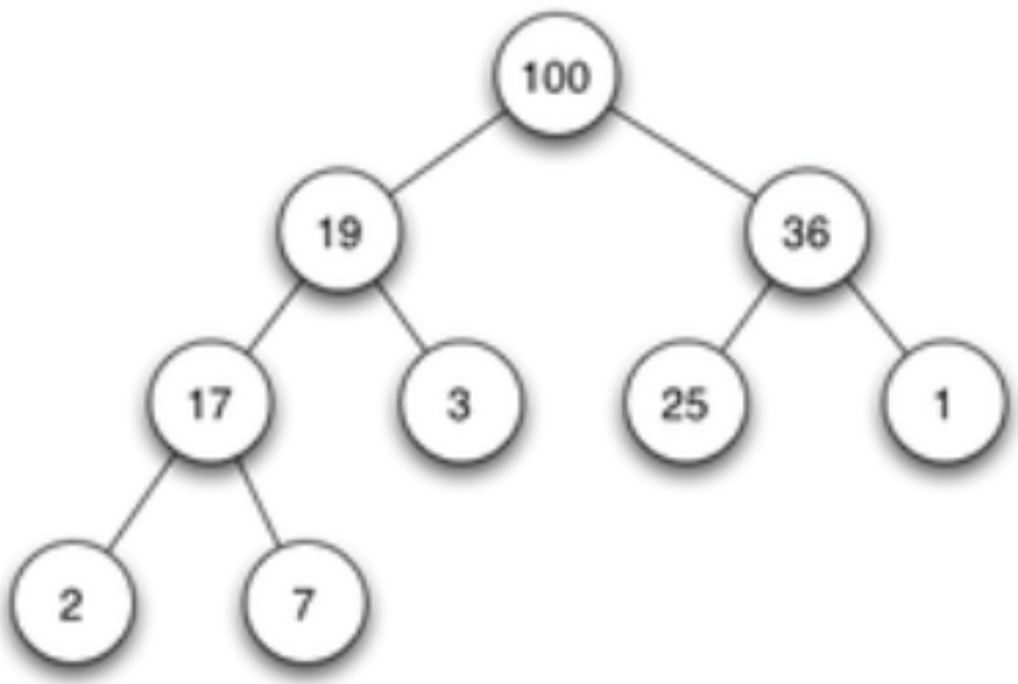
- “Binary” as in binary tree
- “Heap” refers to being “top of the heap”, i.e. what’s on the top dominates what is underneath
    - greater than or less than (or equal to) everything under it

Binary Heap

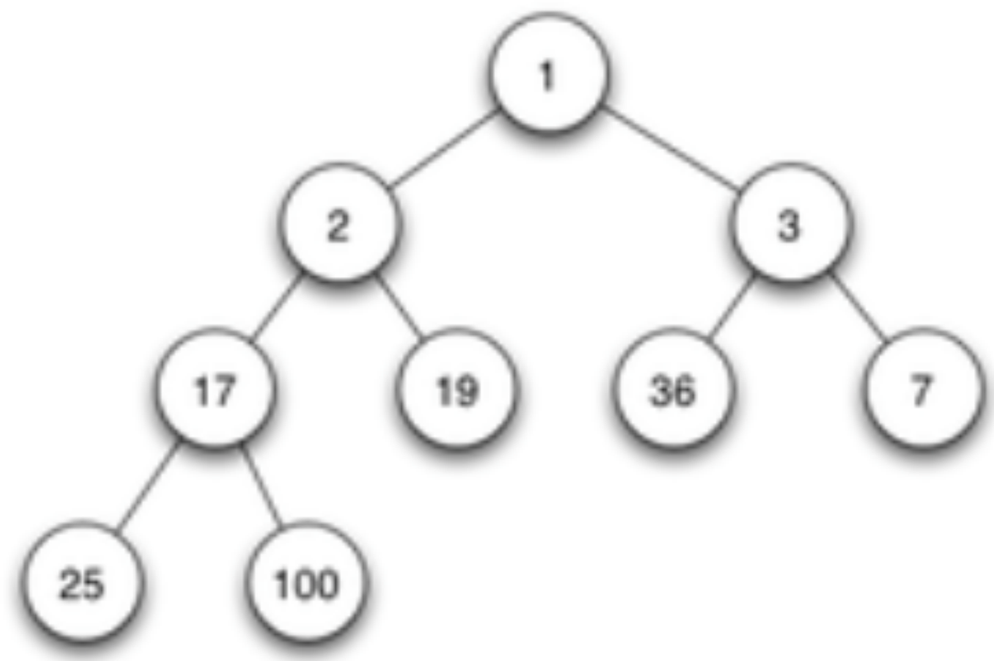
- **Min-heap** — less than (or equal to) its children

- **Max-heap** — greater than (or equal to) its children

Binary Heap



Max-heap



Min-heap

Binary Heap

- Thus four properties of Binary heap are

1. All levels of the tree, except possibly the last one are completely filled ($\boldsymbol{2^i}$ nodes at the **ith-level**)

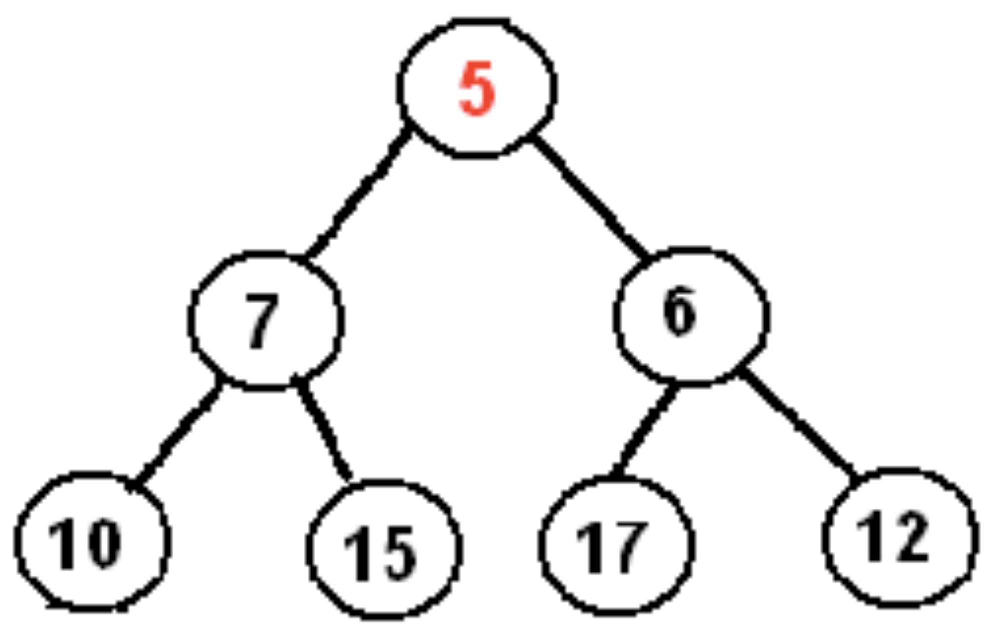
2. If the last level is not complete, the nodes of that level are filled from left to right

3. Each node is “>=” or “<=” each of its children according to some comparison predicate which is fixed for the entire data structure

Binary Heap

4. Two children can be freely interchanged

As long as it doesn't violate the shape and heap properties



Binary Heap

A binary heap *T* storing *n* entries has height $h = \lfloor \log n \rfloor$

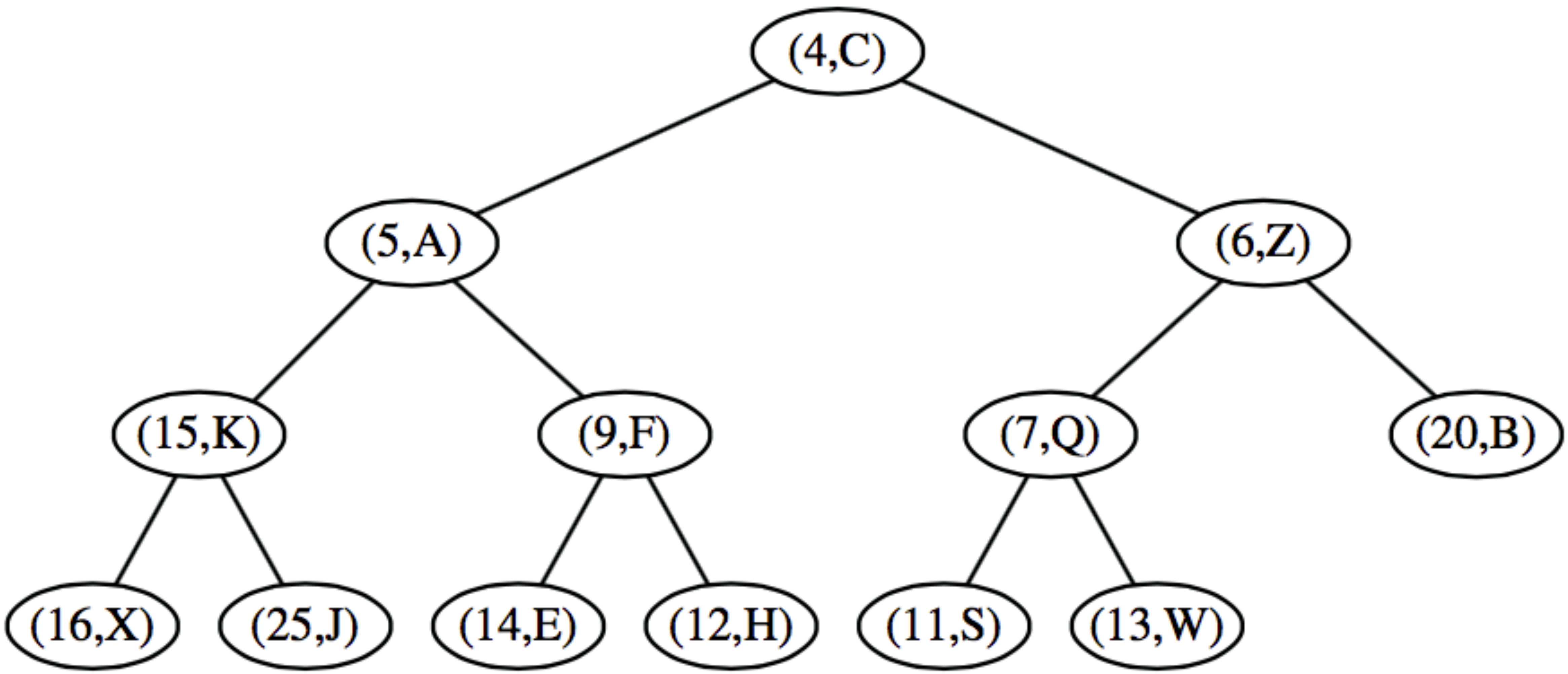
Binary Heap

- **Insertion**

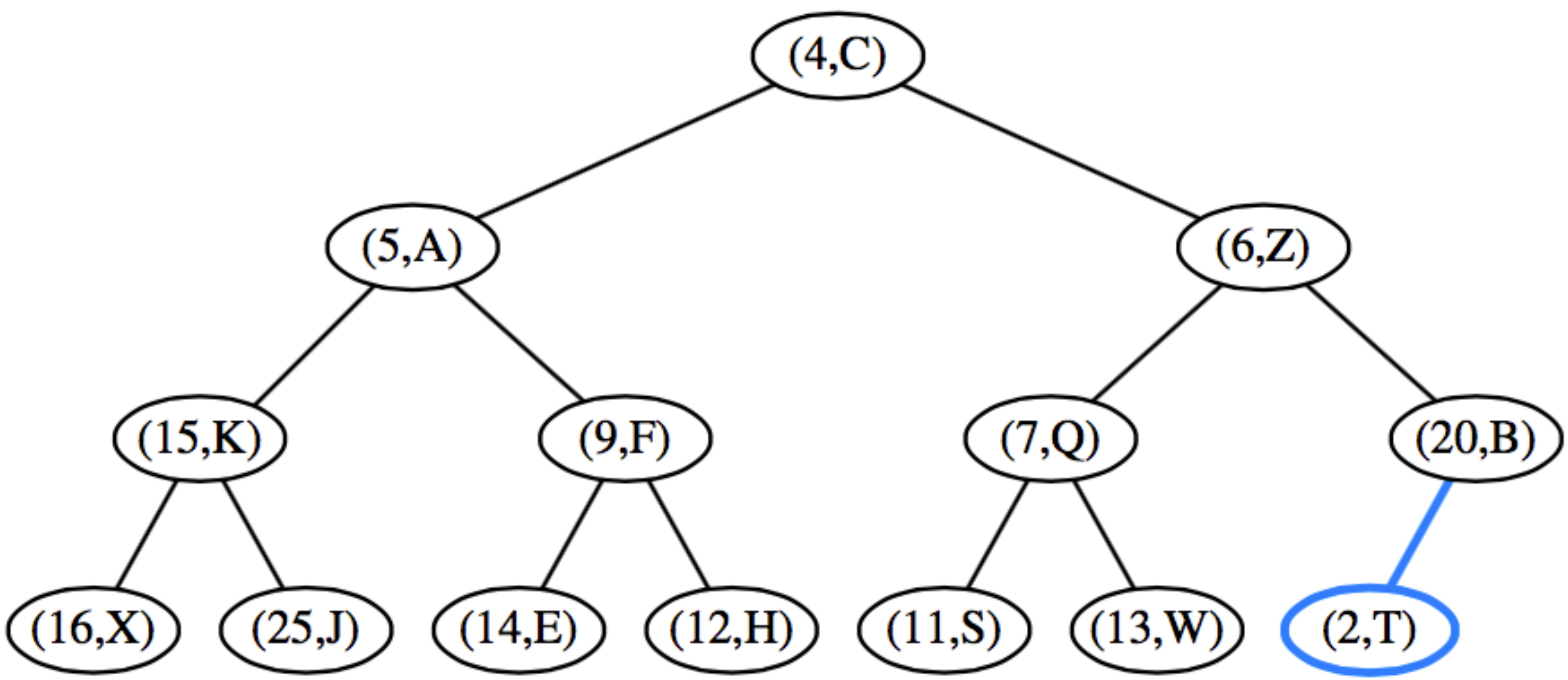
- Algorithm: **upheap / heapify-up / shift-up** — **O(log n)**

Binary Heap

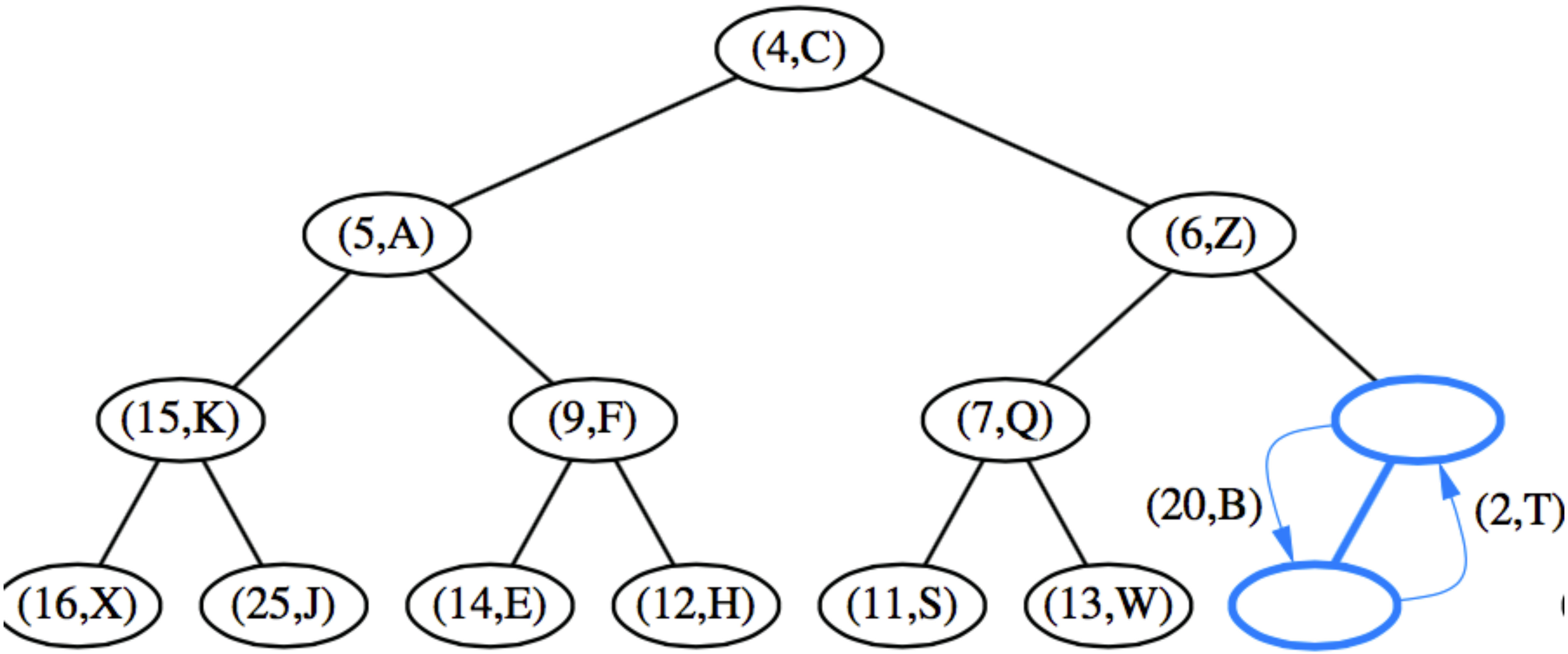
- Insert an item T with key 2 into the following heap



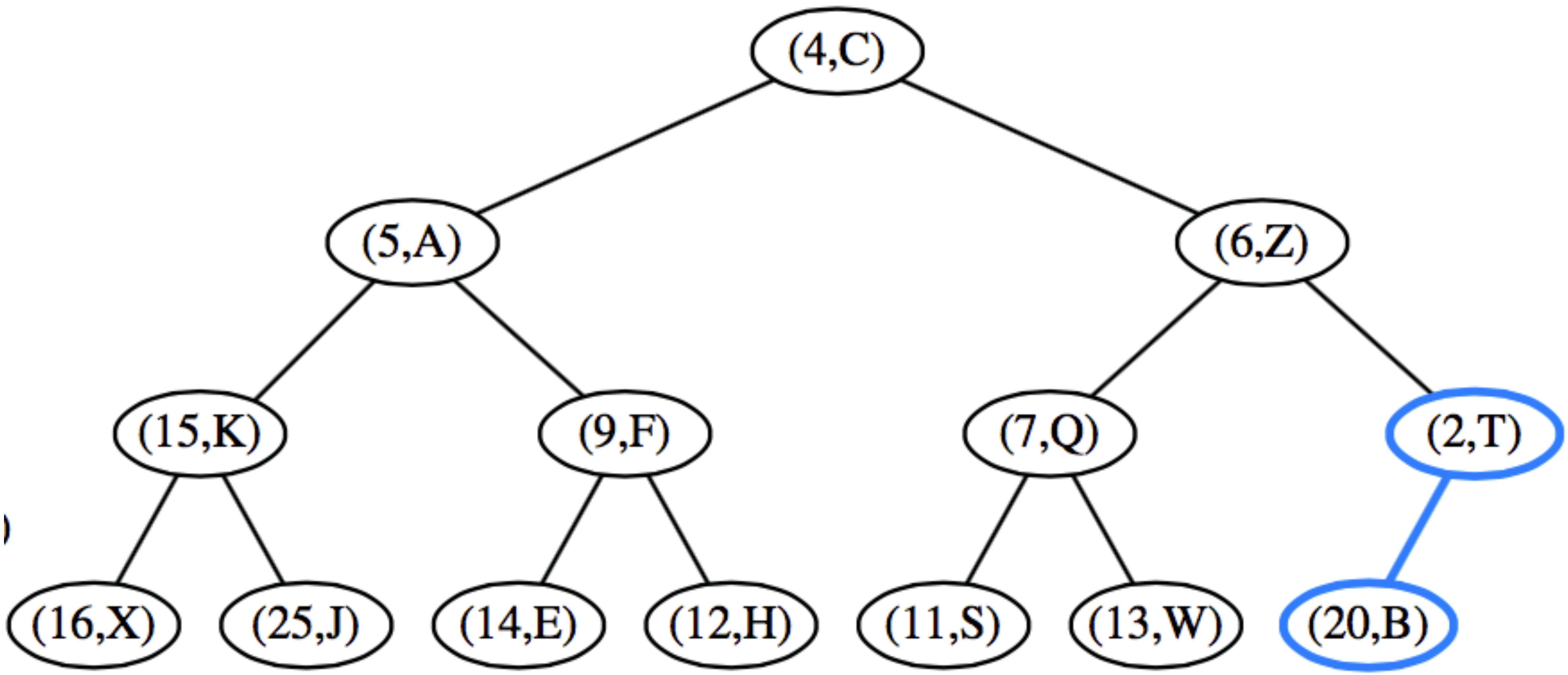
Binary Heap



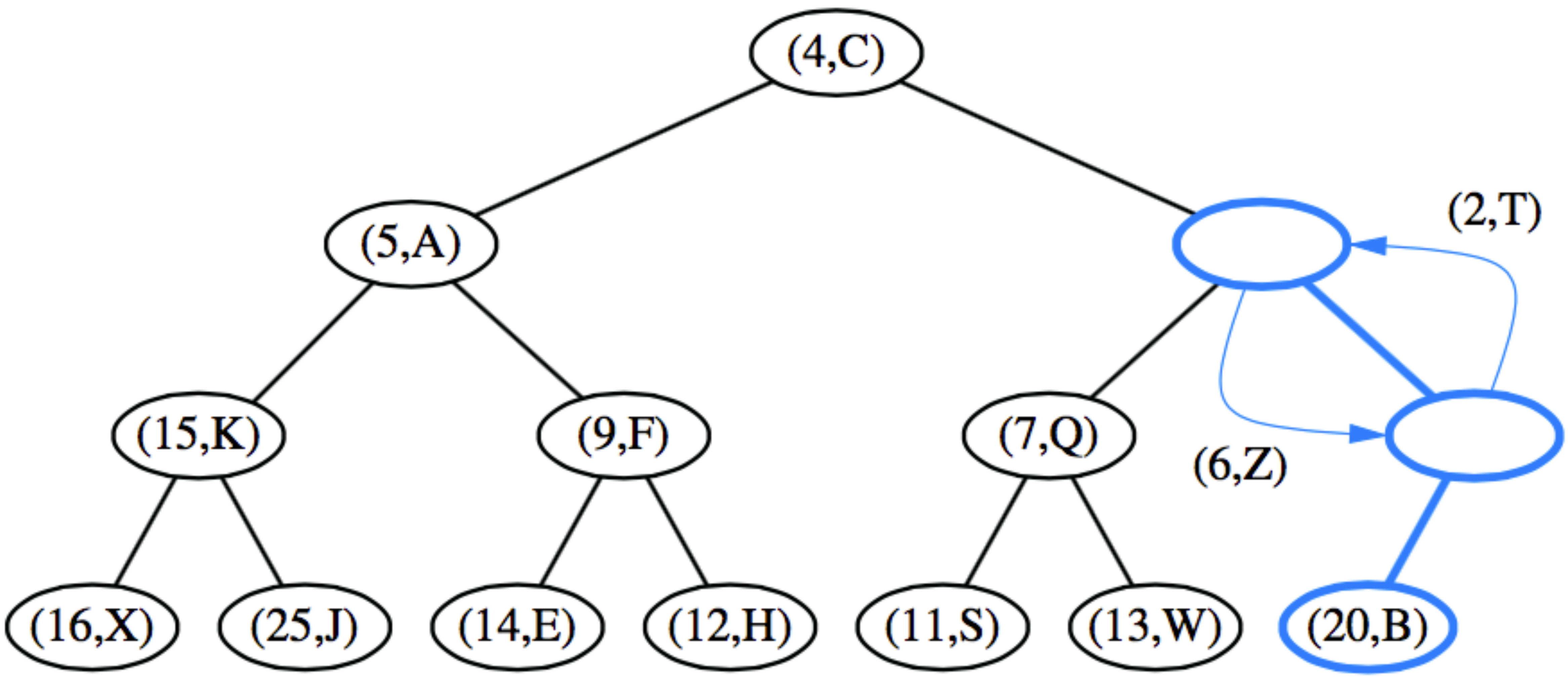
Binary Heap



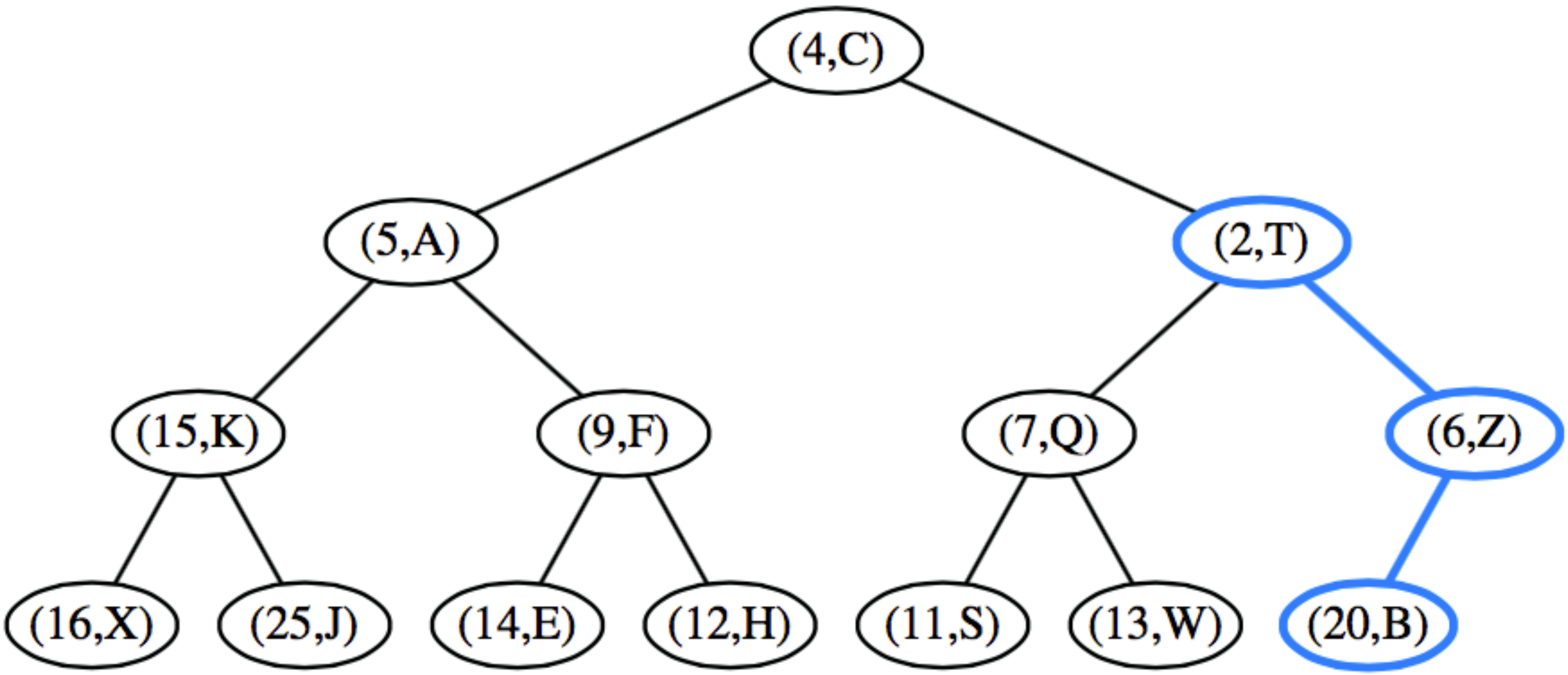
Binary Heap



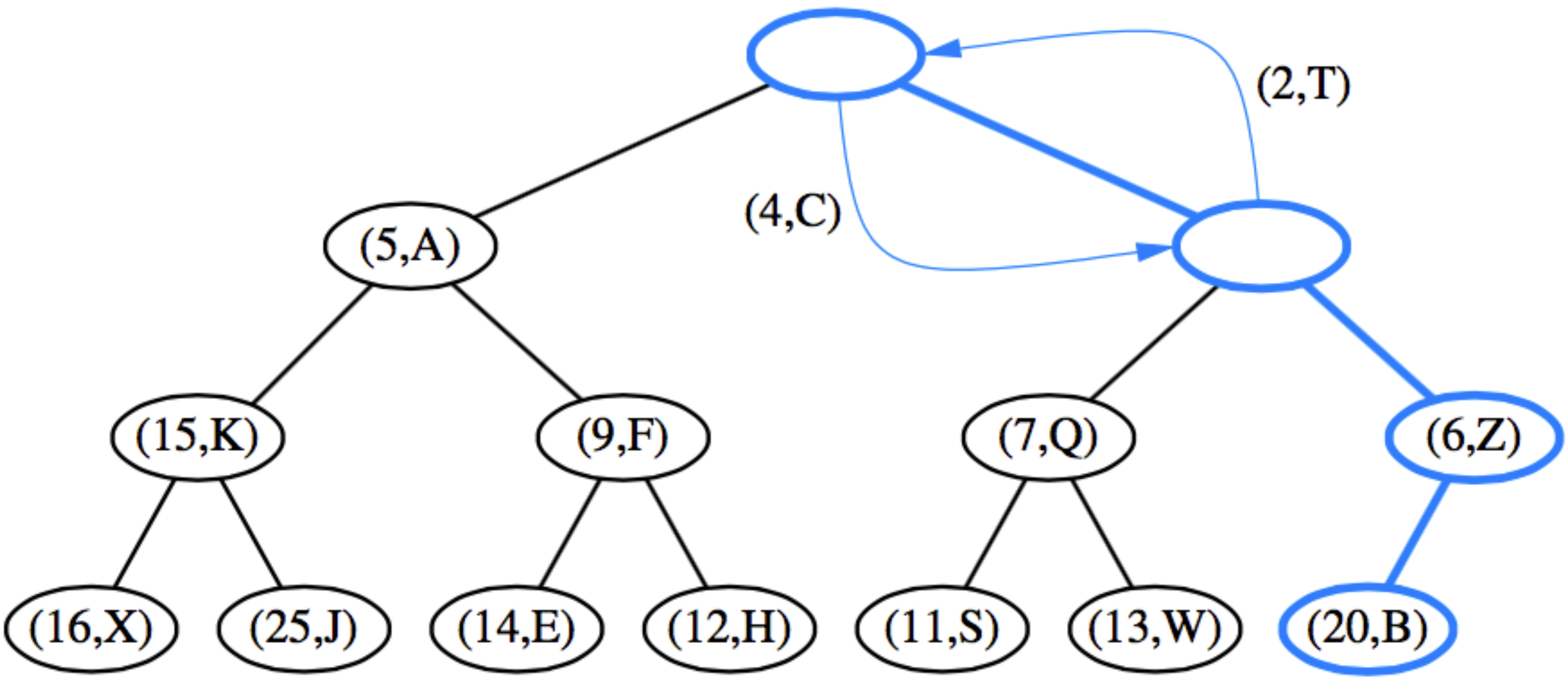
Binary Heap



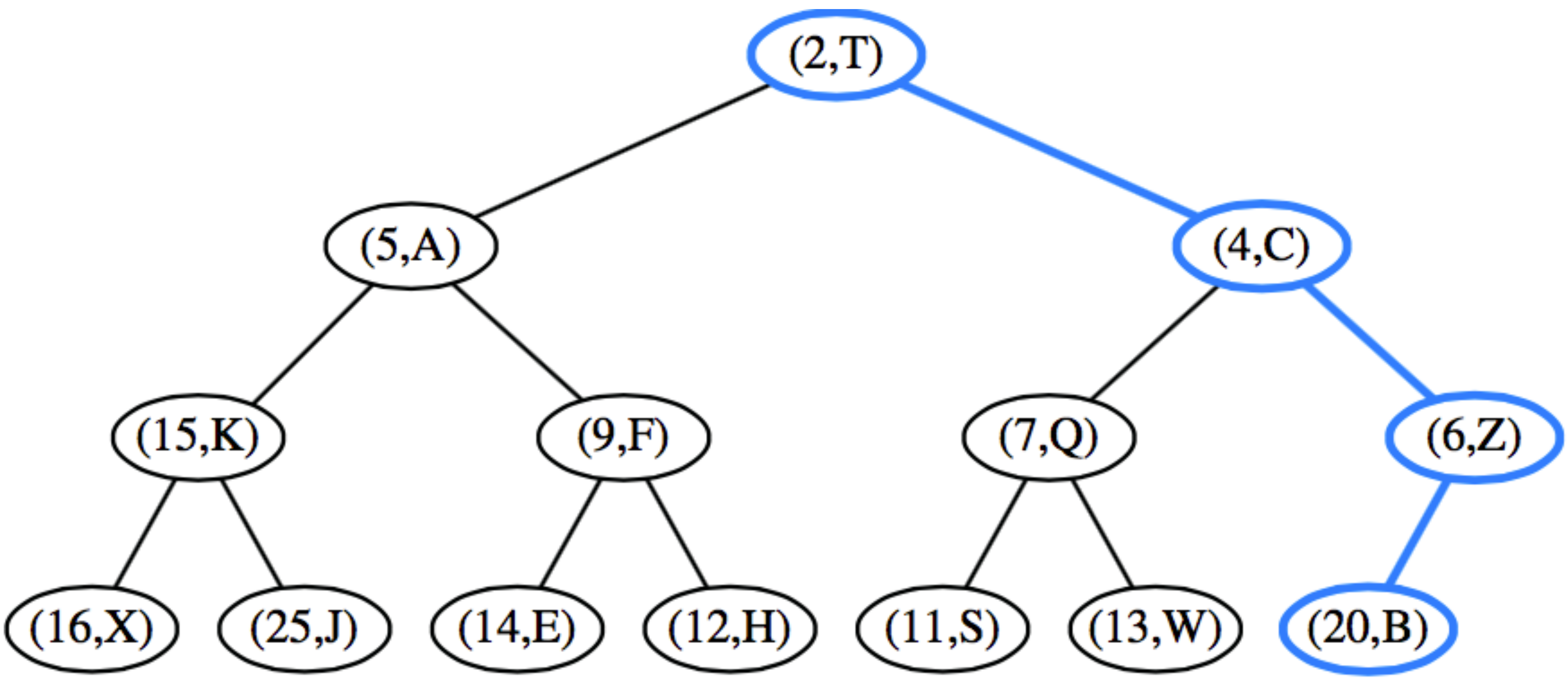
Binary Heap



Binary Heap



Binary Heap



Binary Heap

❖ **Insertion**

- Algorithm: **upheap / heapify-up / shift-up** — **O(log n)**

1. Add element to the bottom level

2. Compare the added element with its parent; if they are in correct order, stop

3. If not, swap the element with its parent and return to previous step

Binary Heap

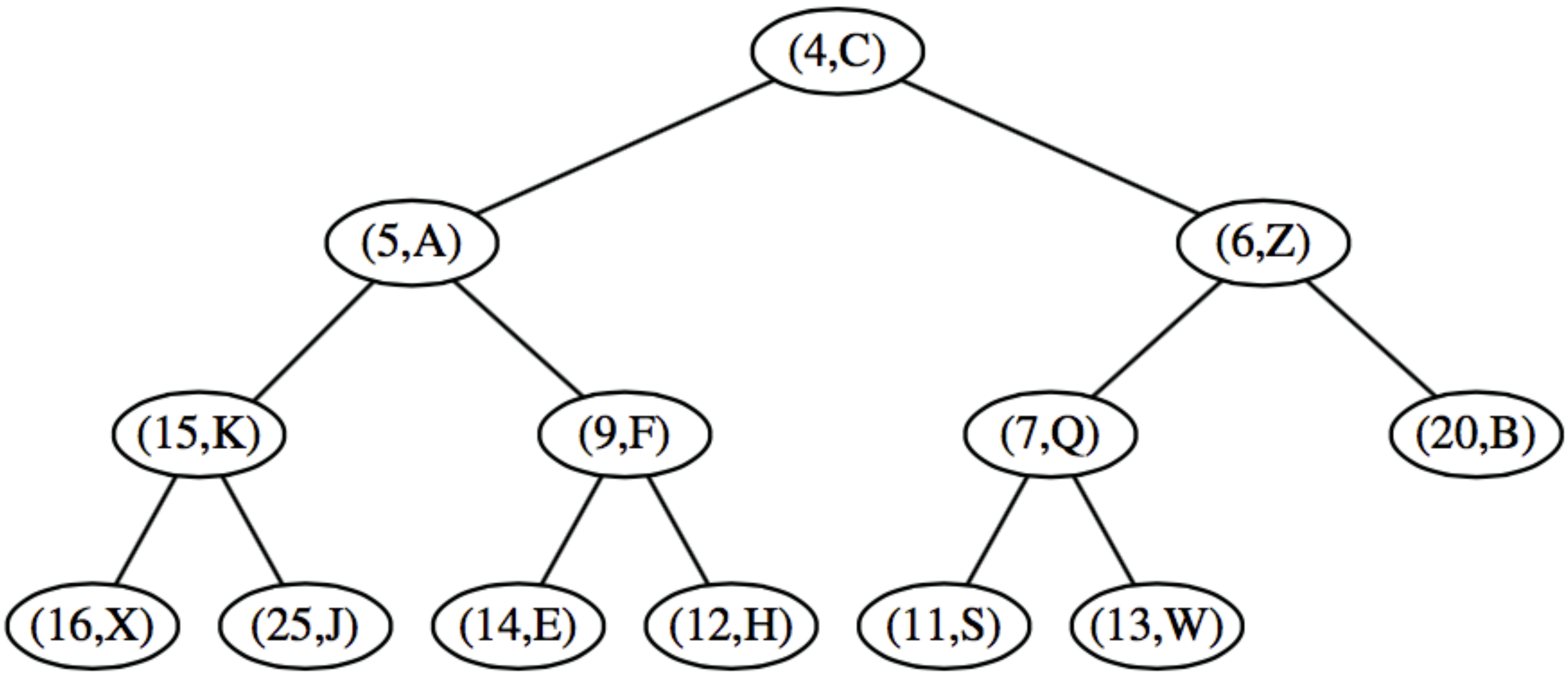
- **Removal**

- Always delete the root node (removing either the min or max)

- Algorithm: **downheap / heapify-down / sift-down — O(log n)**

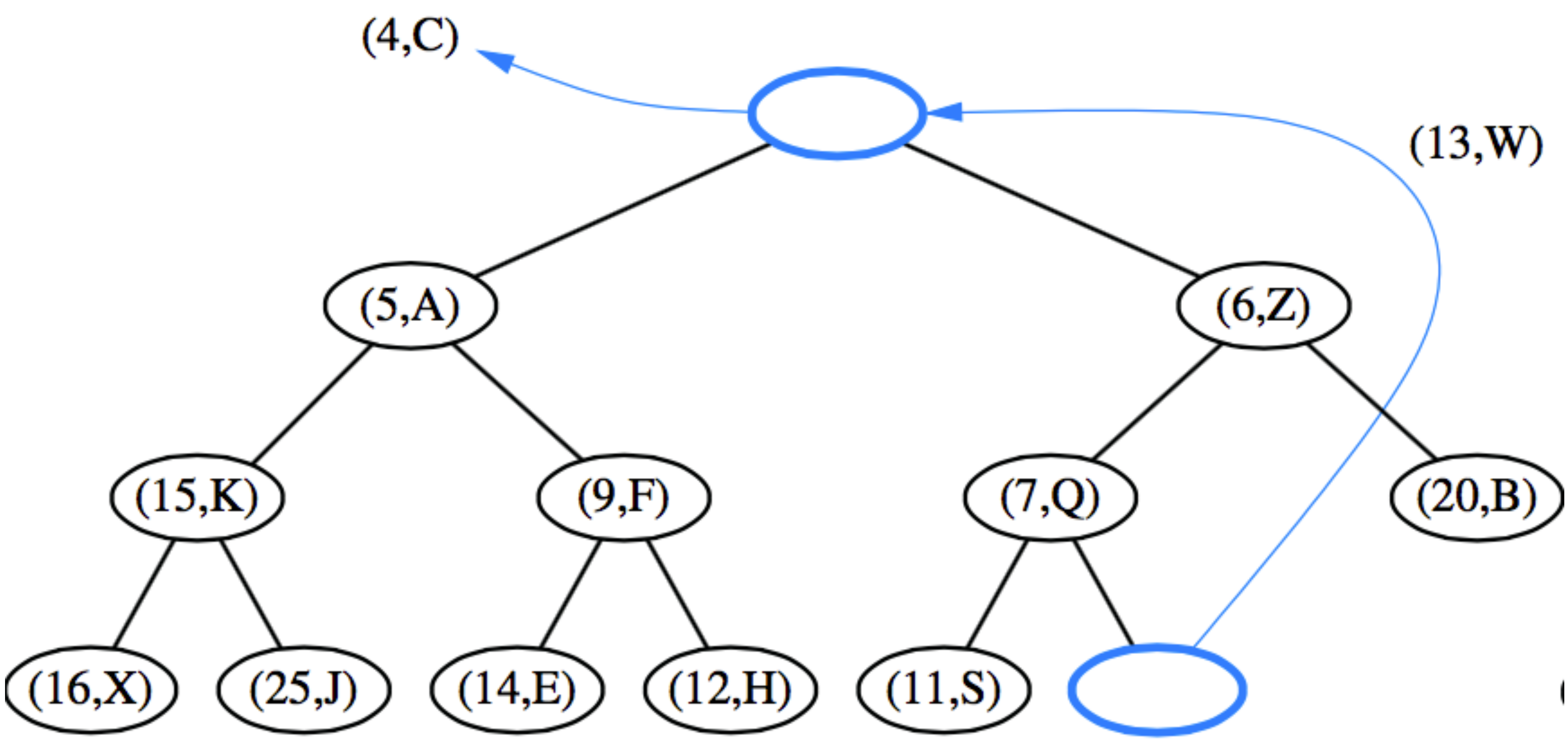
Binary Heap

- Deletion in a binary heap

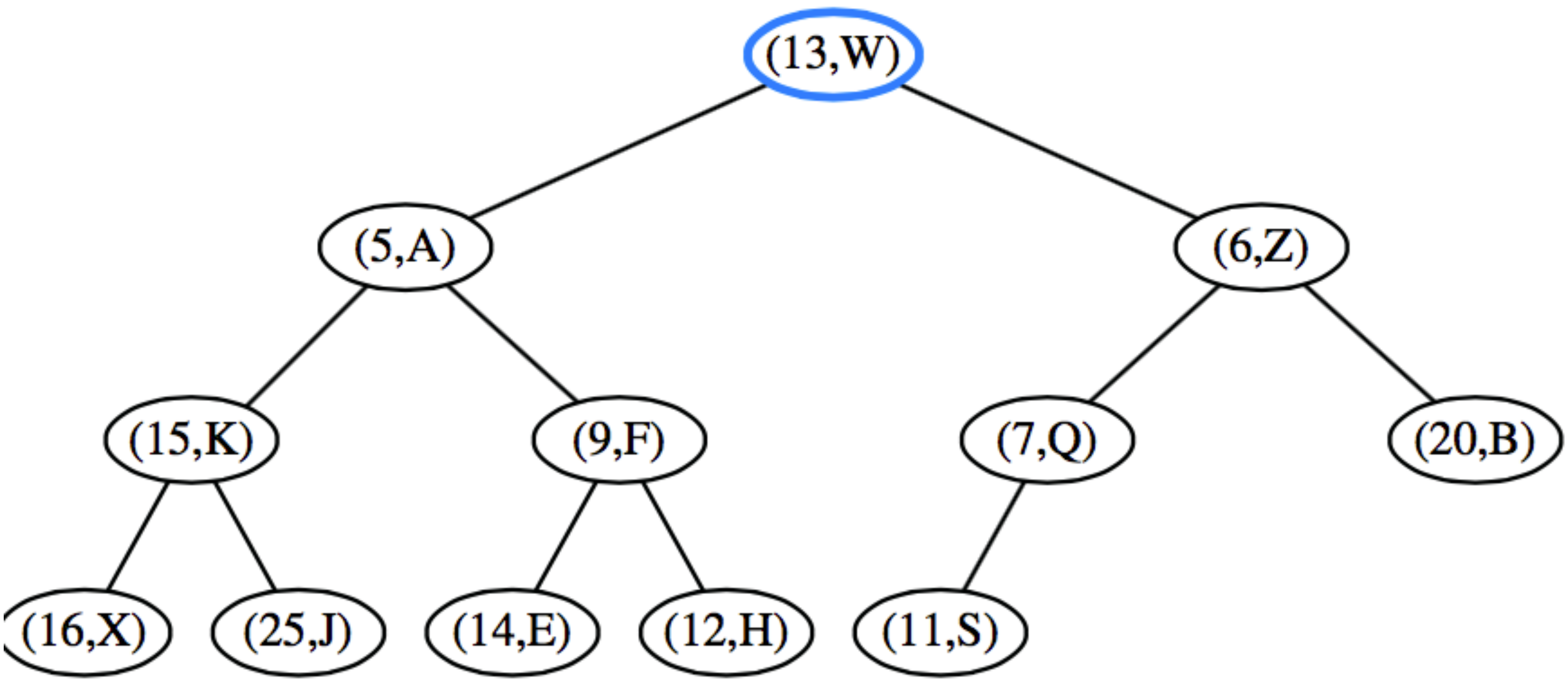


Binary Heap

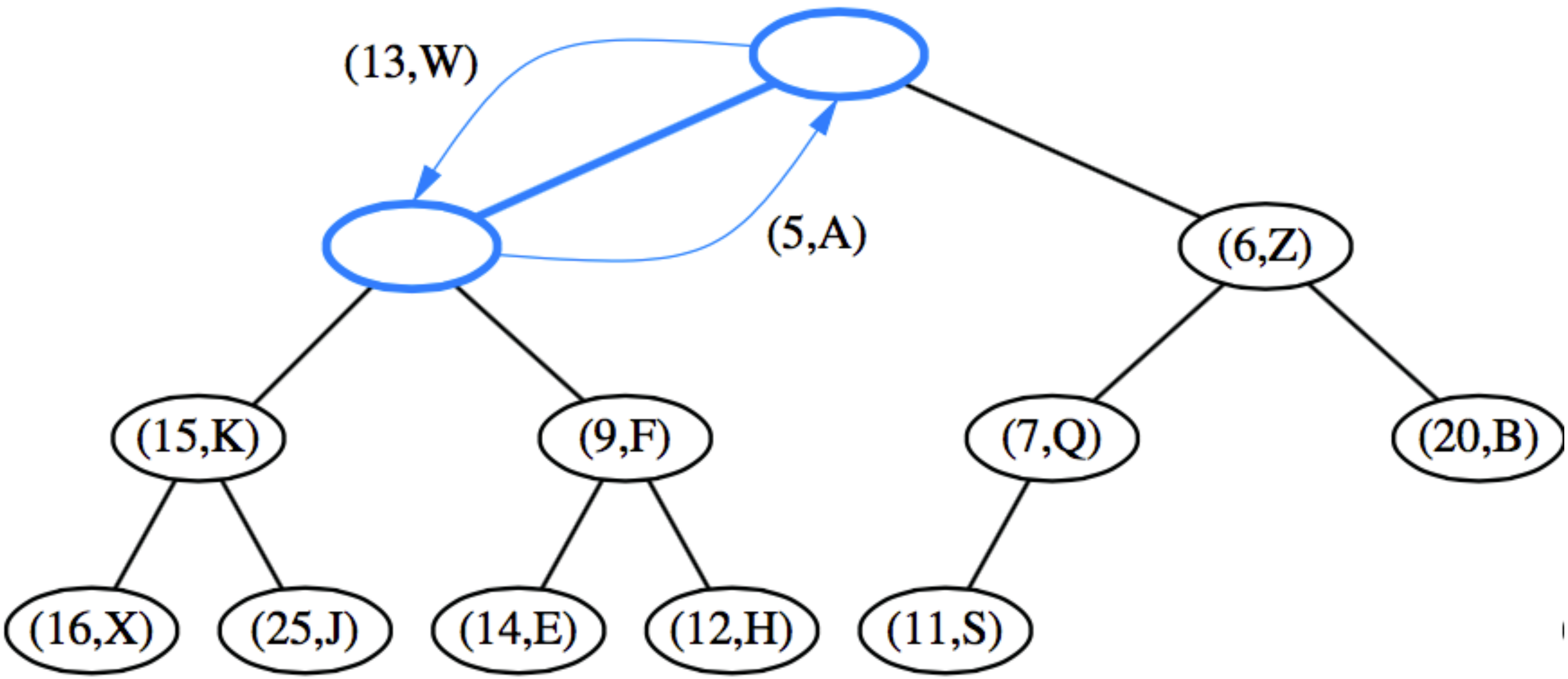
- Deletion in a binary heap



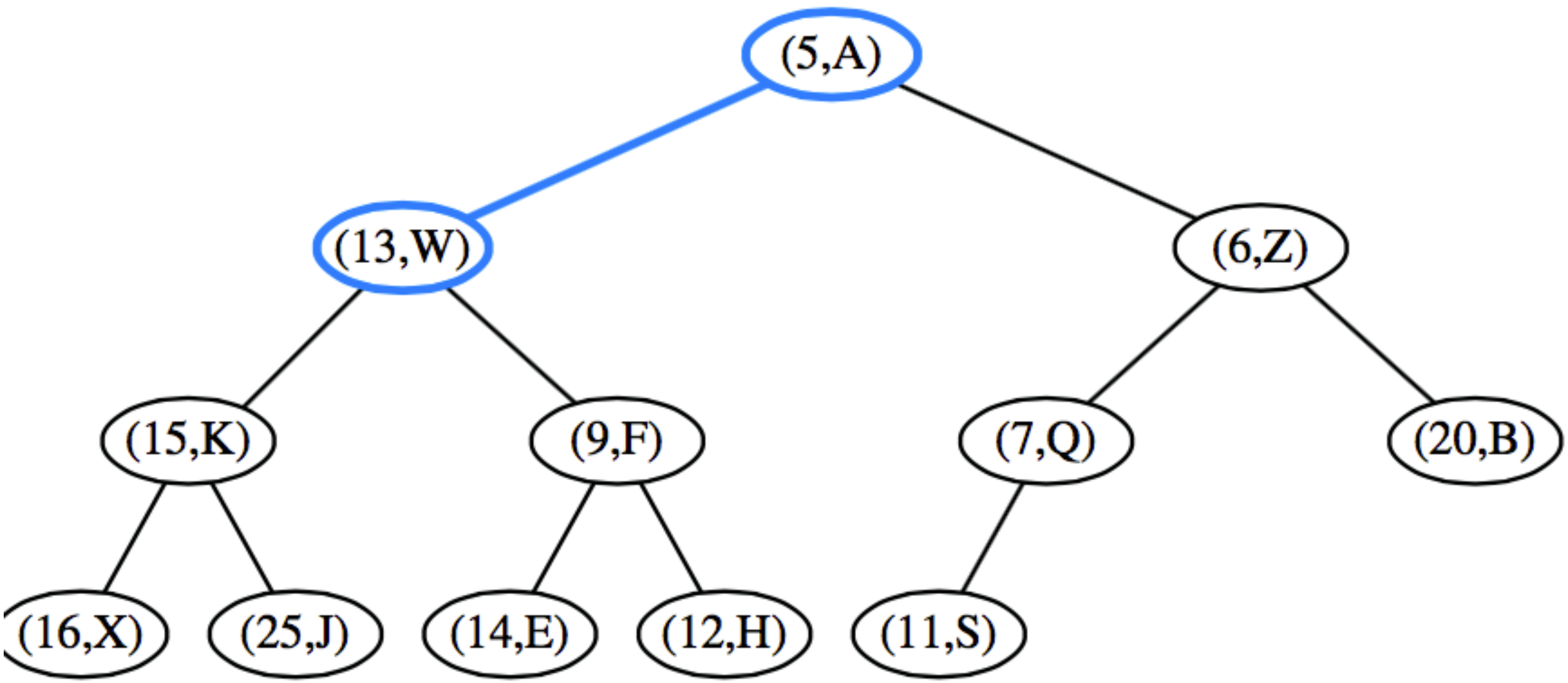
Binary Heap



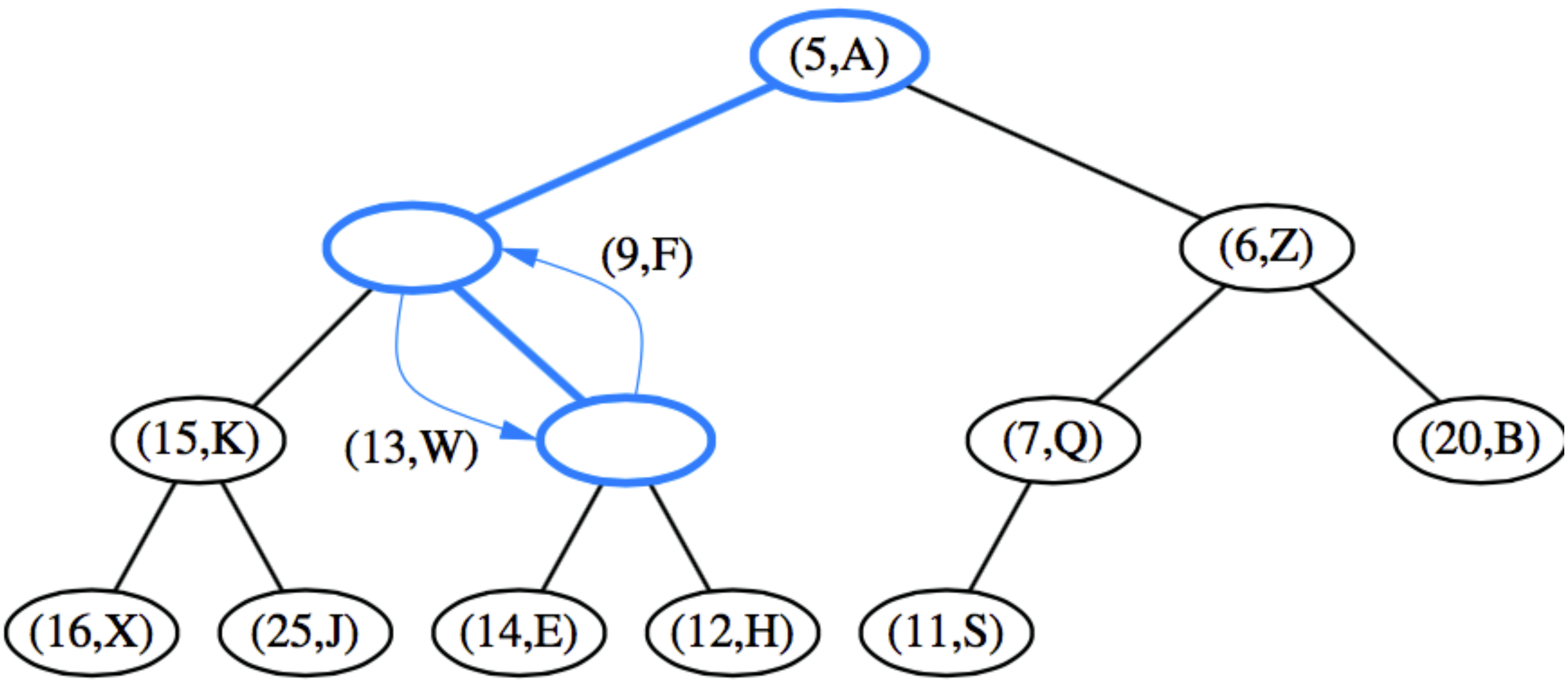
Binary Heap



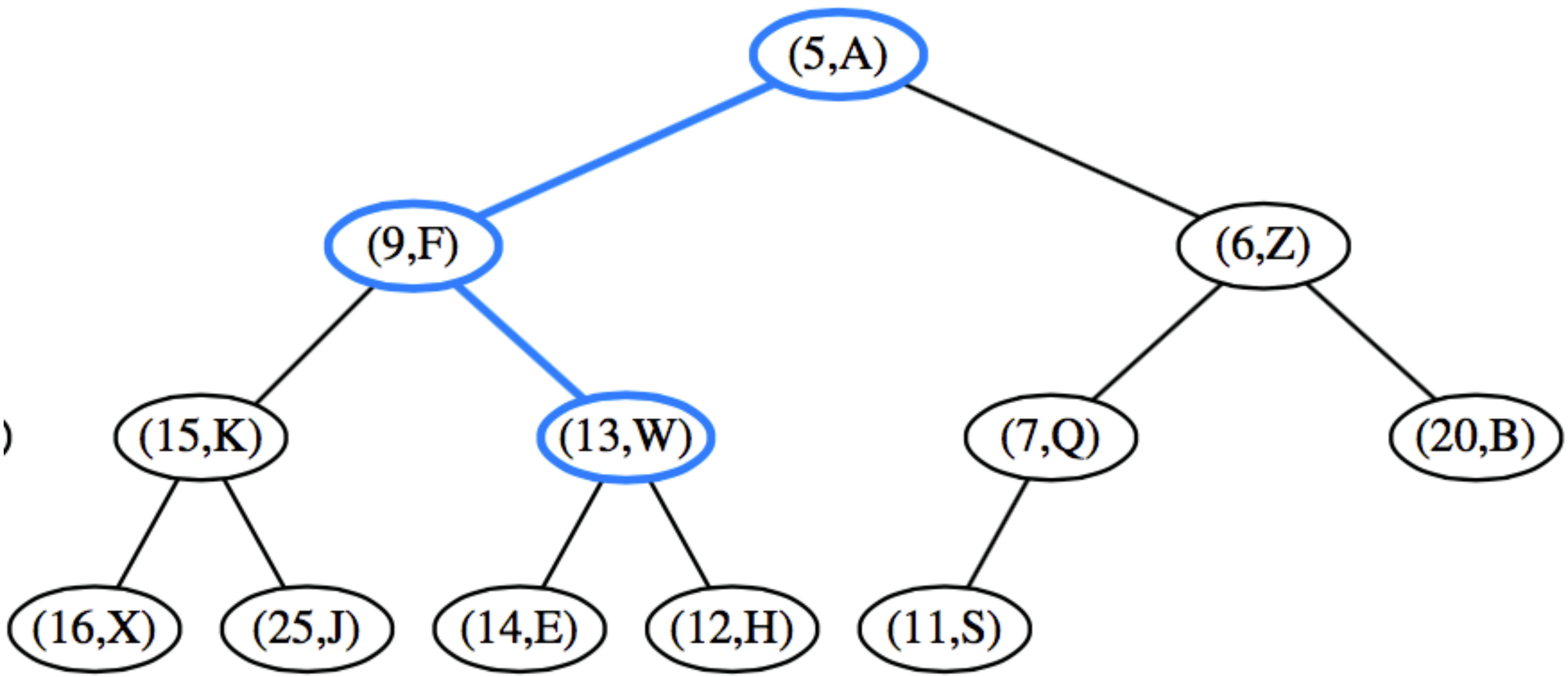
Binary Heap



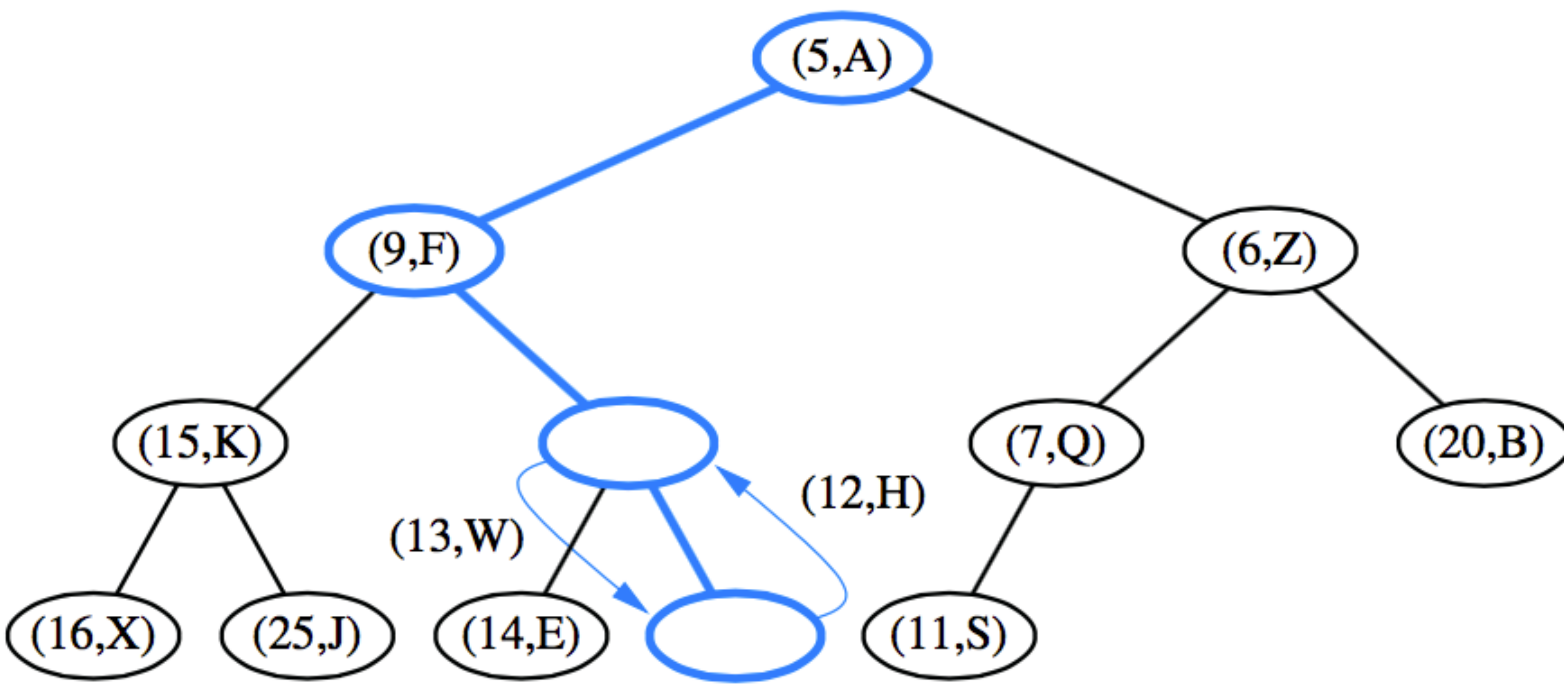
Binary Heap



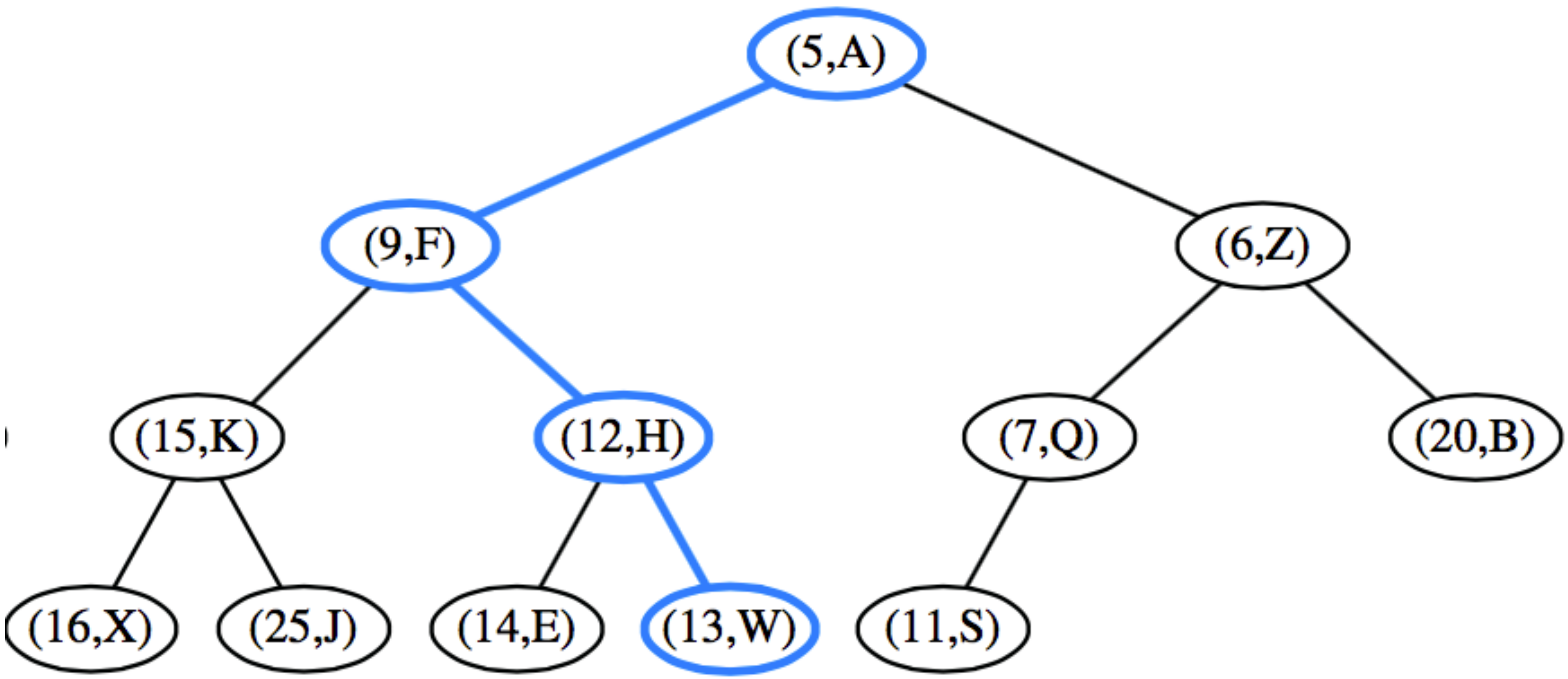
Binary Heap



Binary Heap



Binary Heap



Binary Heap

- Algorithm: **downheap / heapify-down / shift-down** — **O(log n)**

1. Replace root with the last element on the bottom level
2. Compare the swapped element with
   - The larger child (max-heap)
   - The smaller child (min-heap)
3. If they are in correct order, stop
4. If not, swap the element with the child and return to previous step

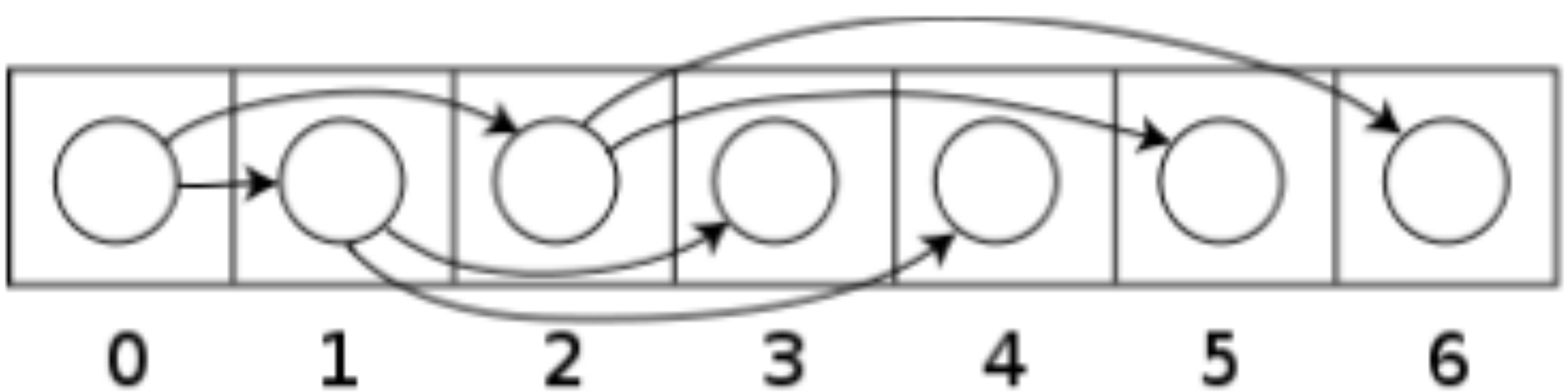
Binary Heap

- **Implementation as an array**

Represent a binary tree without any pointers by using an array of keys and a mapping function

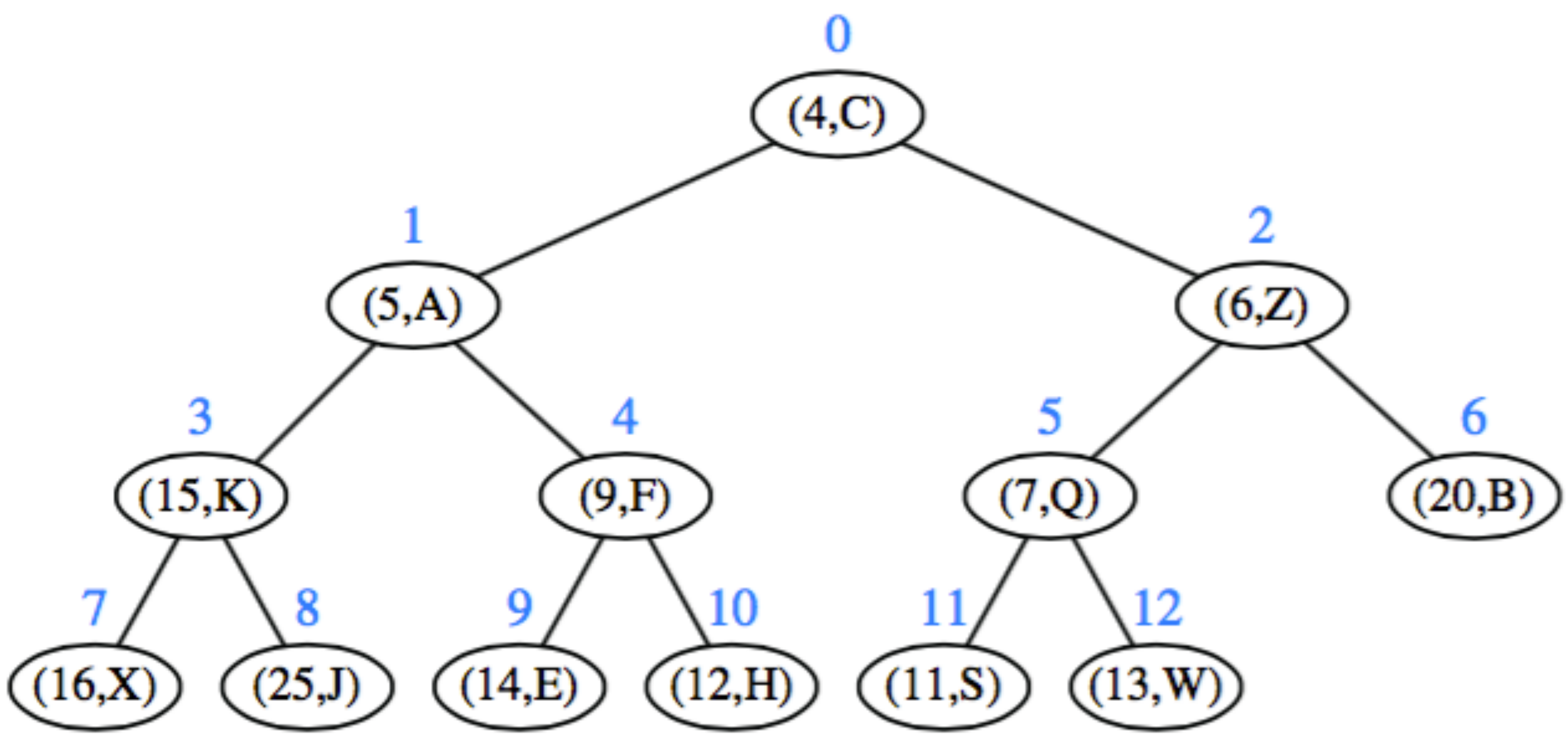
Mapping functions helps find parents and children of a node

- Node at index $\boldsymbol{i}$ has **children** at indices $\mathbf{2}\boldsymbol{i}+\mathbf{1}$ and $\mathbf{2}\boldsymbol{i}+\mathbf{2}$
- Node at index $\boldsymbol{i}$ has **parent** at index $(\boldsymbol{i}-\mathbf{1})/\mathbf{2}$



Binary Heap

- Node at index i has **children** at indices $2i + 1$ and $2i + 2$
- Node at index i has **parent** at index $(i - 1)/2$



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

Data Structures and Algorithms in Java

Different Books, Different Representation

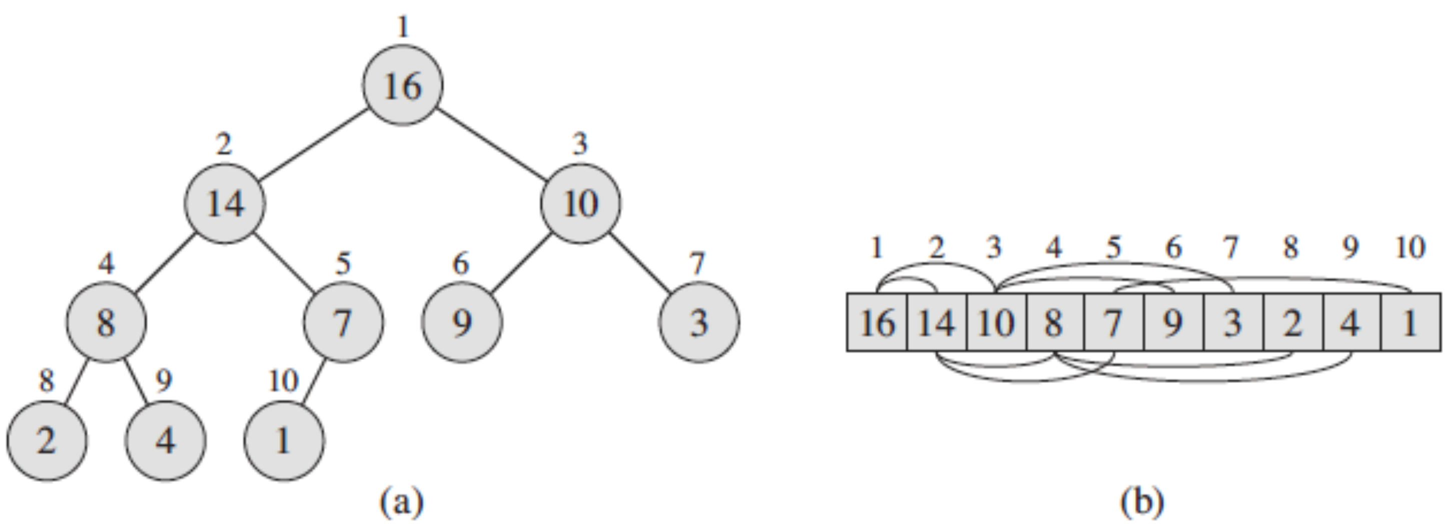


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

PARENT(i)
return $\lfloor i/2 \rfloor$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

Introduction to Algorithms

Binary Heap

- **Inserting in an array based heap, represented as H**

Algorithm InsertInHeap(k, v)

Input: priority k, **value** v; **Output**: none

```
1. H[size] = new entry (k, v)
   // insert entry (k, v) at rank =  size of array

2. size = size + 1  // increase heap size
```

Binary Heap

```
// Now perform upheap, starting at the last node

3. i = size - 1

4. while i > 0 and H[(i-1)/2].key() > k

   5. swap(H[i], H[i/2])    // swap entry (k, v) with the entry at parent node

   6. i = (i-1)/2           // after this statement, index i holds entry (k, v)
```

Binary Heap

- **Deleting in an array based heap H**

  Algorithm RemoveMin()

  Input: none; **Output**: entry with the smallest key

```
  1. if size == 0 then ReportError("Empty Heap")

  2. itemToReturn = H[0] // minimum is at rank 0

  3. H[0] = H[size-1] // put the entry at last rank at root
location

  4. size = size - 1  // decrease heap size
```

Binary Heap

```
// Now perform downheap to restore heap order

5. i = 0

6. childIndex = findSmallerChild(i)

7. while (childIndex != 0 && H[childIndex].key < H[i].key)

   8. swap(H[childIndex], H[i])

   9. i = childIndex

   10. childIndex = findSmallerChild(i)

11. return itemToReturn
```

Binary Heap

```
// Now perform downheap to rest
i = 0
childIndex = findSmallerChild(i)
while (childIndex != 0 && H[child
    swap(H[childIndex], H[i])
    i = childIndex
    childIndex = findSmallerChild
return itemToReturn
```

```
Algorithm findSmallerChild(i)
Input: index i of a node
Output: index of the child of node i with smaller key, 0 if node is a leaf
if (2*i + 1) < size // Node has two children
   if  (H[2*i + 1].key < H[2*i + 2].key) // Left child is smaller
      return  (2*i + 1)
   else return (2*i + 2) // Right child is smaller
else if (2*i + 1) == size // Node has one child
    return (2*i + 1)
else
    return (0) // Node is a leaf
```

Heap-Sort

Heap-Sort

- Heap based priority queue can be used to create a very efficient sorting algorithm: **heap-sort**

Heap-Sort

- Heap based priority queue can be used to create a very efficient sorting algorithm: **heap-sort**

1. Construct the priority queue: **O(n log n)**

Heap-Sort

- Heap based priority queue can be used to create a very efficient sorting algorithm: **heap-sort**

1. Construct the priority queue: **O(n log n)**

2. Repeatedly extract the minimum: **O(n log n)**

Heap-Sort

- Heap based priority queue can be used to create a very efficient sorting algorithm: **heap-sort**

  1. Construct the priority queue: **O(n log n)**

  2. Repeatedly extract the minimum: **O(n log n)**

  Overall complexity is **O(n log n)**

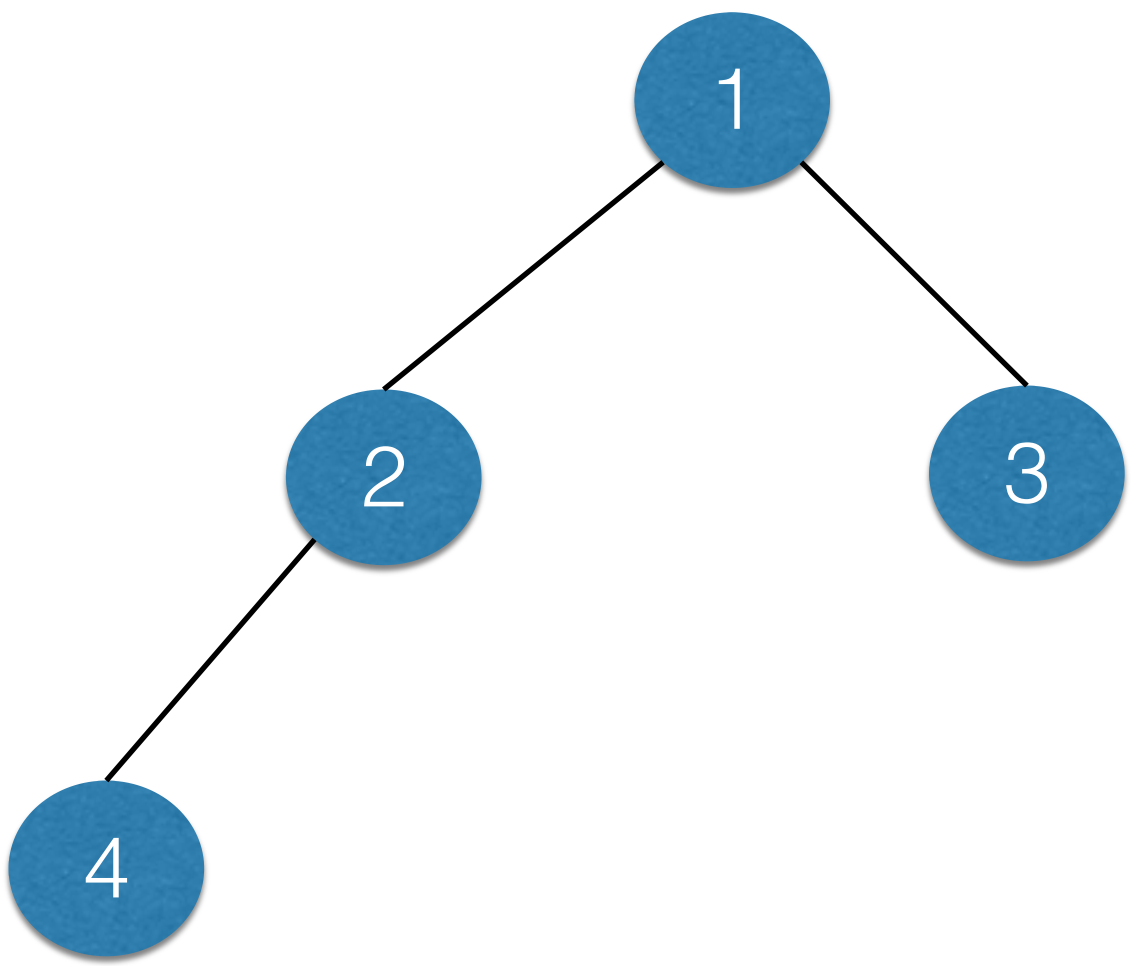
  This is the best that can be expected from any comparison based sorting algorithm

Binary Heap Height

A heap T storing n entries has height $h = \lfloor \log n \rfloor$

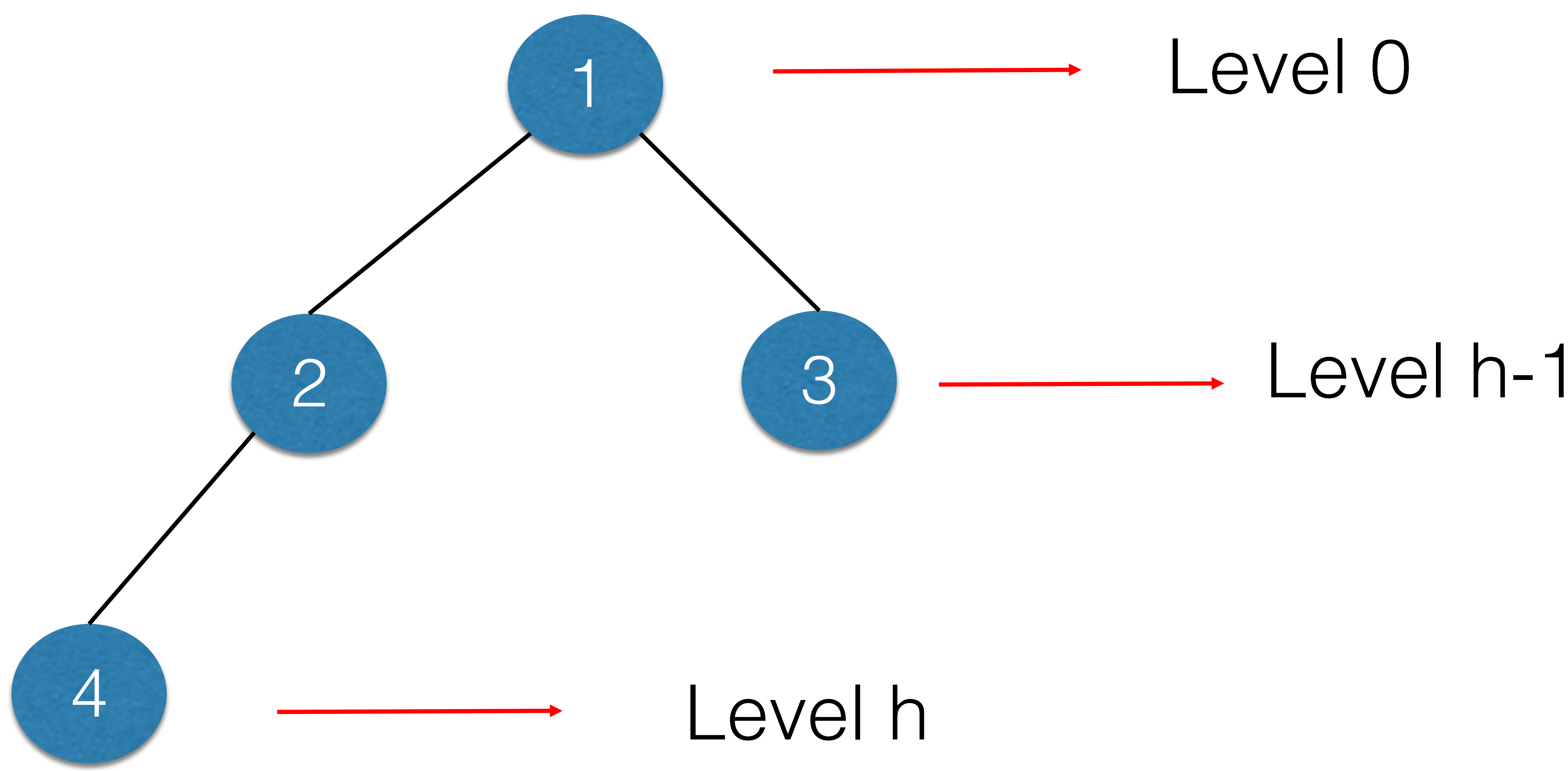
Binary Heap Height

- We know that a binary heap is a complete binary tree



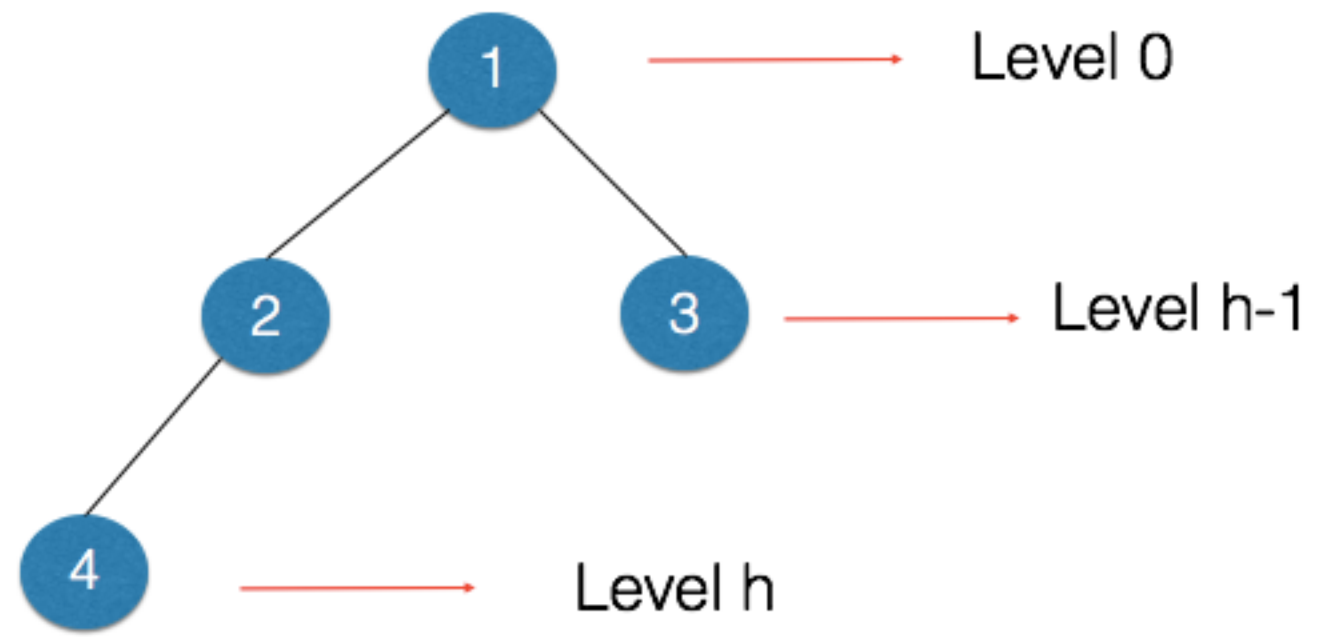
Binary Heap Height

- We know that a binary heap is a complete binary tree



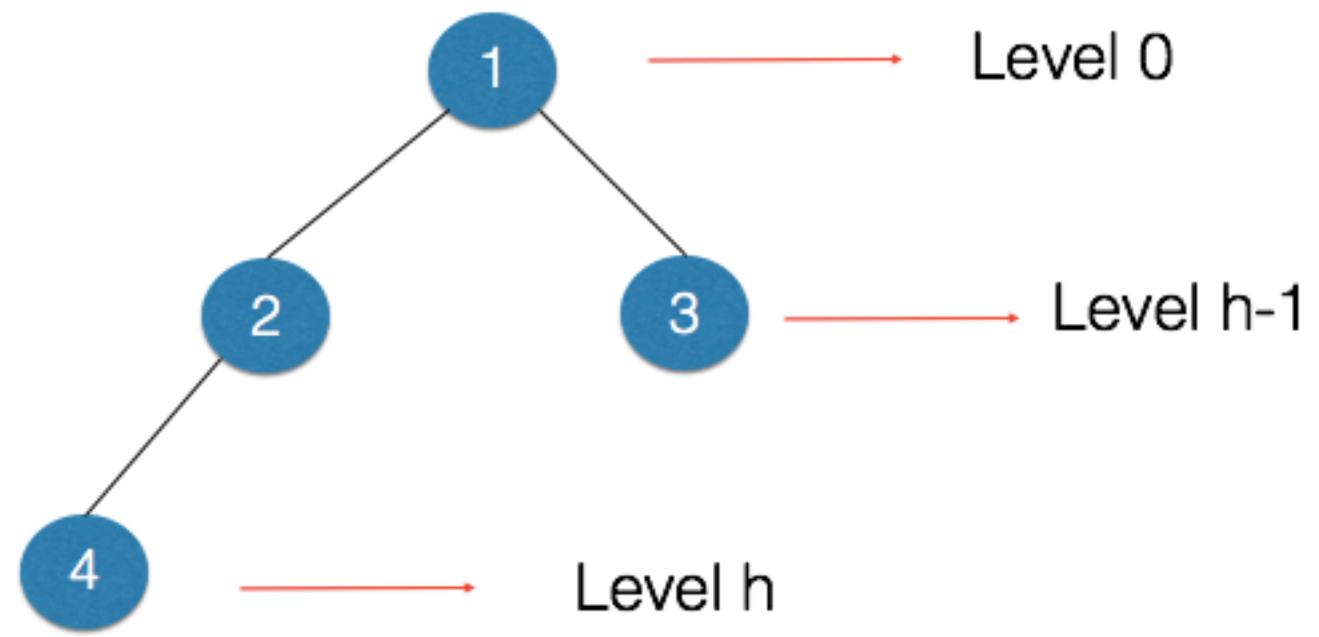
Binary Heap Height

- What is the number (sum) of nodes from level 0 through level h-1?



Binary Heap Height

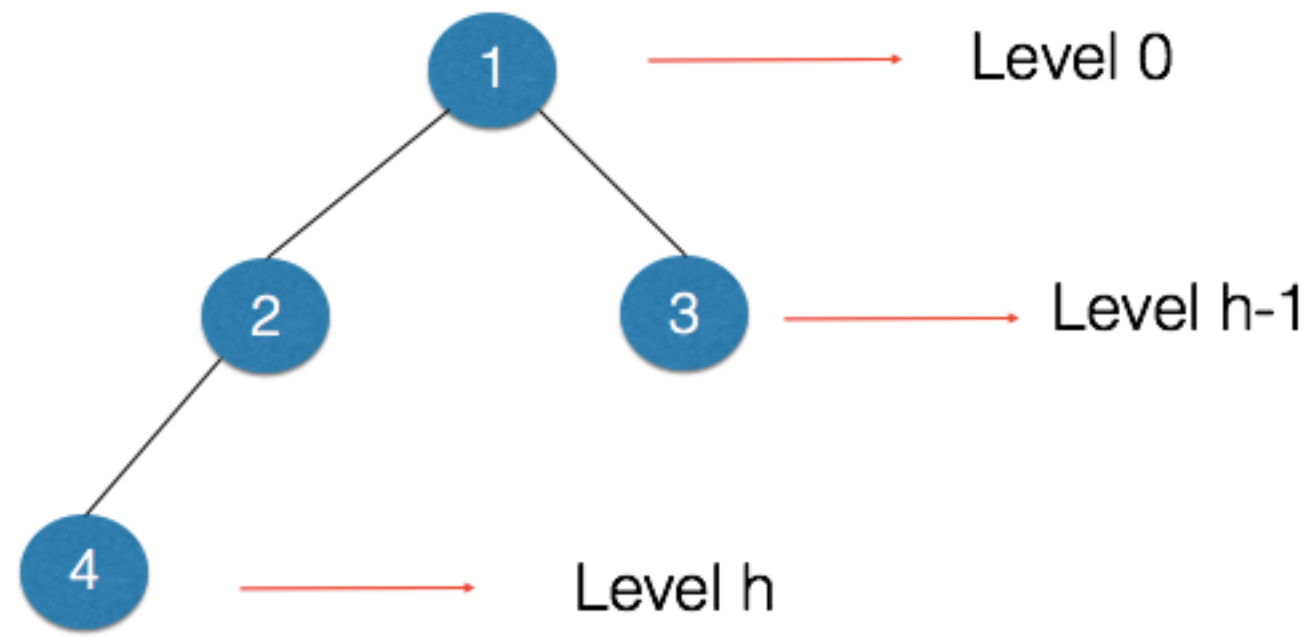
- What is the number (sum) of nodes from level 0 through level h-1?



$$1 + 2 + 4 + \cdots + 2^{h-1}$$

Binary Heap Height

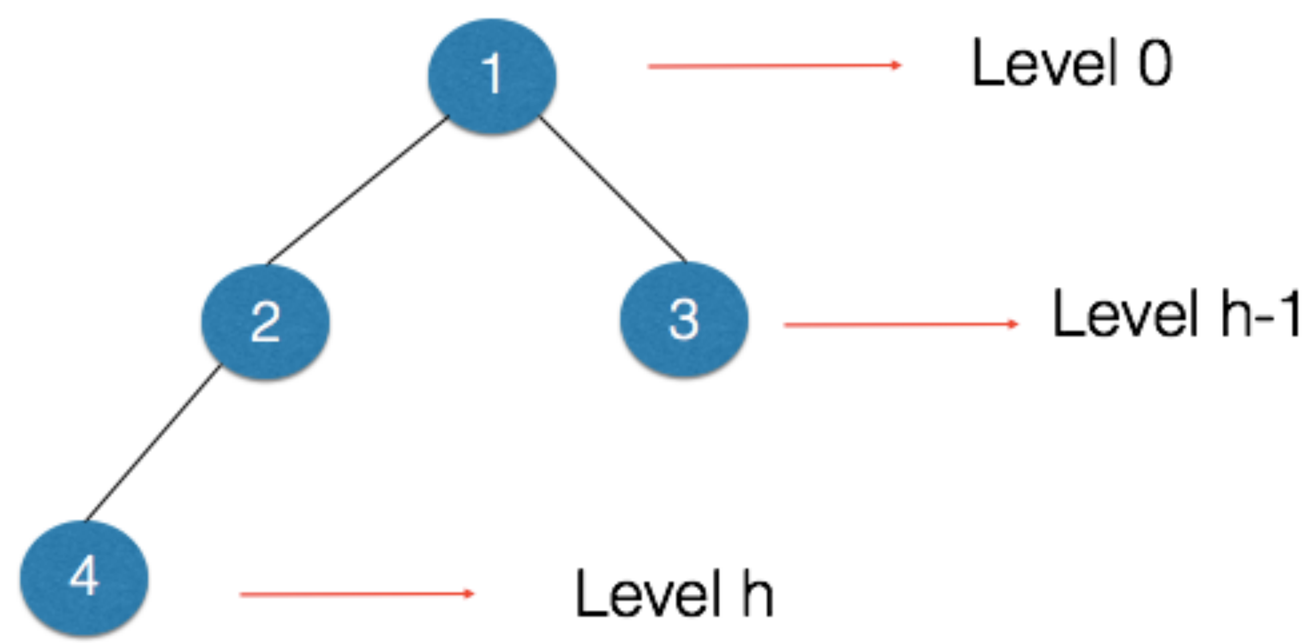
- What is the number (sum) of nodes from level 0 through level h-1?



$$1+2+4+\cdots+2^{h-1}=2^{h}-1$$

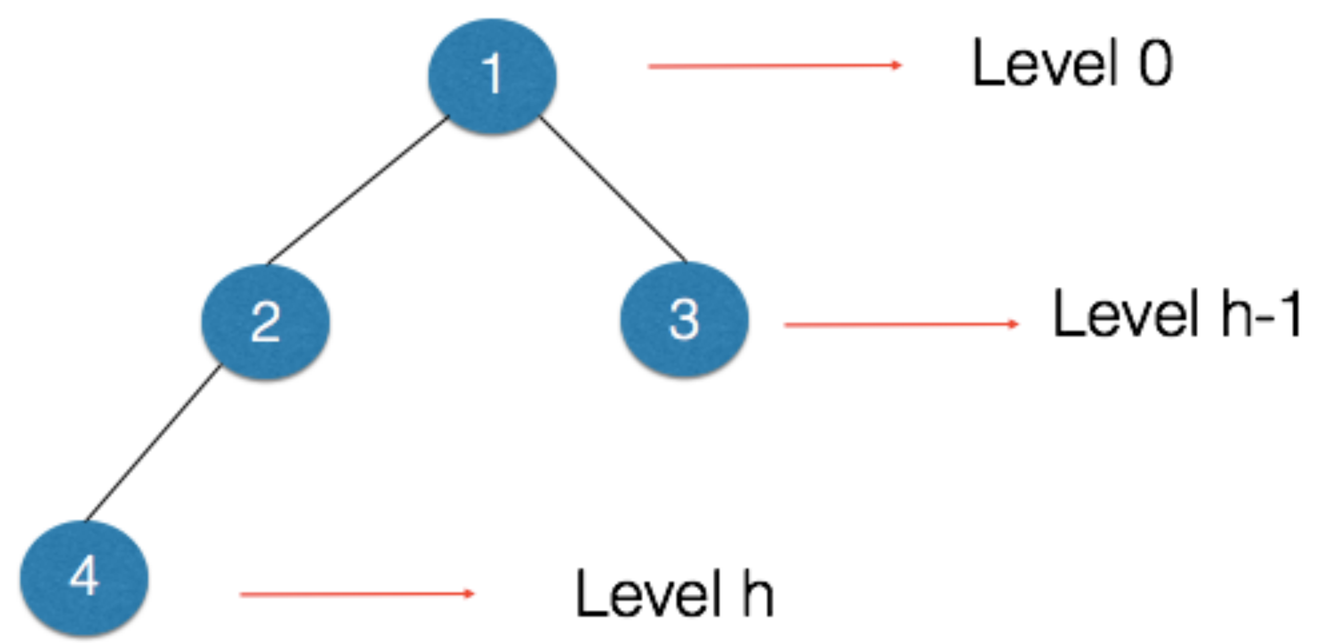
Binary Heap Height

- How many nodes do we have at level h?



Binary Heap Height

- How many nodes do we have at level h?



❖ Minimum 1
❖ Maximum 2^h

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \qquad \text{and} \qquad n \geq 2^h - 1 + 1$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \qquad \text{and} \qquad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1}$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

- Take log on both sides

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

$$\log(n+1) - 1 \leq h$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

$$\log(n+1) - 1 \leq h \quad \text{and} \quad \log(n) \geq h$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

$$\log(n + 1) - 1 \leq h \quad \text{and} \quad \log(n) \geq h$$

$$\log(n + 1) - 1 \leq h \leq \log(n)$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$n \leq 2^h - 1 + 2^h \quad \text{and} \quad n \geq 2^h - 1 + 1$$

$$n + 1 \leq 2^{h+1} \quad \text{and} \quad n \geq 2^h$$

$$\log(n + 1) - 1 \leq h \quad \text{and} \quad \log(n) \geq h$$

$$\log(n + 1) - 1 \leq h \leq \log(n)$$

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$\log(n+1) - 1 \leq h \leq \log(n)$$

- Also, we know that h is an integer

Binary Heap Height

- Thus, if n is the total number of nodes in a complete binary tree having height h, then

$$\log(n+1) - 1 \leq h \leq \log(n)$$

- Also, we know that h is an integer

- Thus the two inequalities imply that $h = \lfloor \log n \rfloor$

Did we achieve todays objectives?

- Priority Queues
- Binary Heap
- Heap-Sort