Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Priority Queues

Binary Heap

Heap-Sort

Merge-Sort

Something from the Last Lecture

Height of Red-Black Trees

2. In the worst case, that is the case with the tallest tree, there must be some long path from the root to a leaf.
3. Since the number of black nodes on that long path is limited to $O(\log n_b)$, the only way to make it longer is to have lots of red nodes.
4. Since red nodes cannot have red children, in the worst case, the number of nodes on that path must alternate red/black.
5. thus, that path can be only twice as long as the black depth of the tree.
6. **Therefore, the worst case height of the tree is $O(2 \log n_b)$.**
7. Therefore, the height of a red-black tree is $O(\log n)$.

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

Priority Queues

- Main operations

  add(priority, value)

  peek()

  remove ()

Priority Queues

- Possible implementations

  Unsorted Array

  Unsorted Linked List

  Sorted Array

  Sorted Linked List

Unsorted Array

- Insertion – O(1)
- Removal – O(n)

Unsorted Linked List

- Insertion – O(1)
- Removal – O(n)

Sorted Array

- Insertion – O(n)
- Removal – O(1)

Sorted Linked 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

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)

Binary Heap

- Favoured implementation of priority queue
- Heaps maintained partial 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)
    - “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

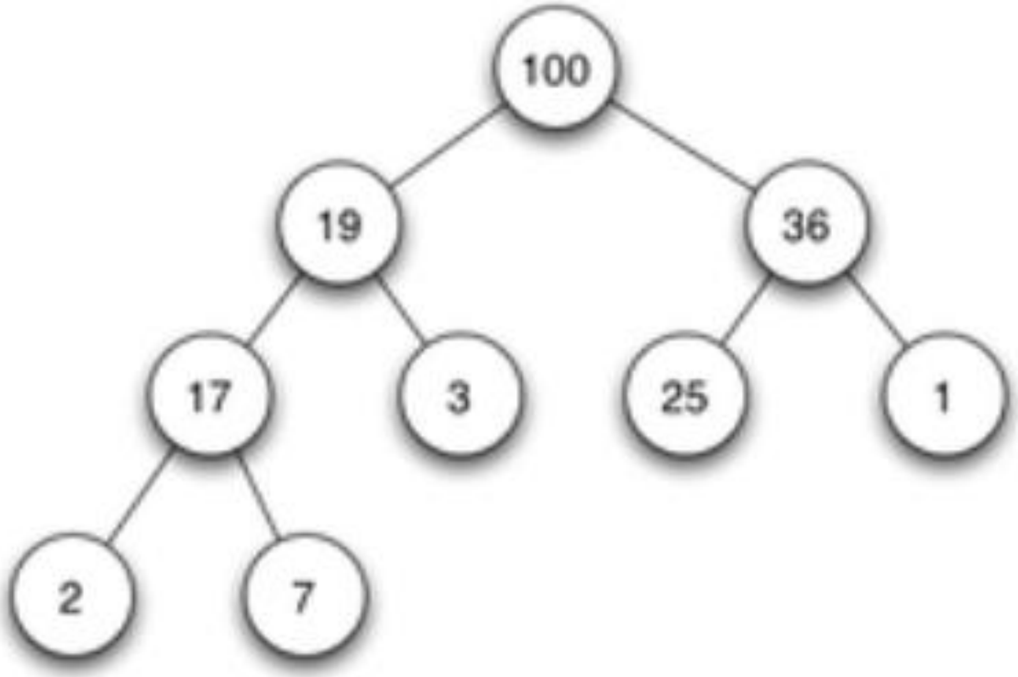
- Binary heap is a binary tree

  Keys in each node dominate the keys of its children

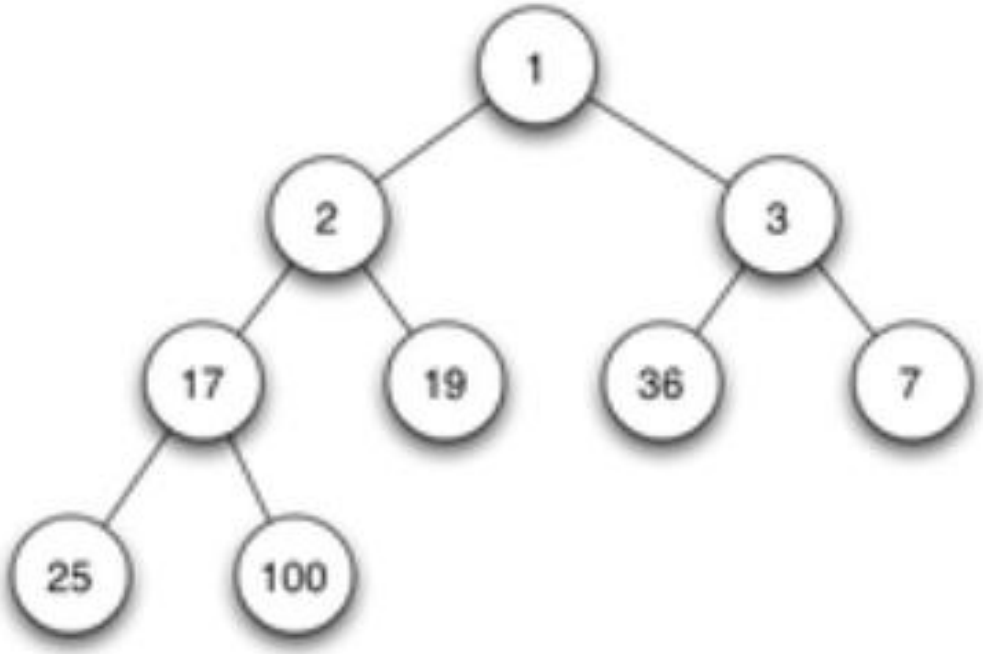
  Min-heap — less than (or equal to) its children

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

Binary Heap



Max-heap



Min-heap

Binary Heap

- Binary heap properties

All levels of the tree, except possibly the last one are completely filled (**2^i** nodes at the **ith-level**)

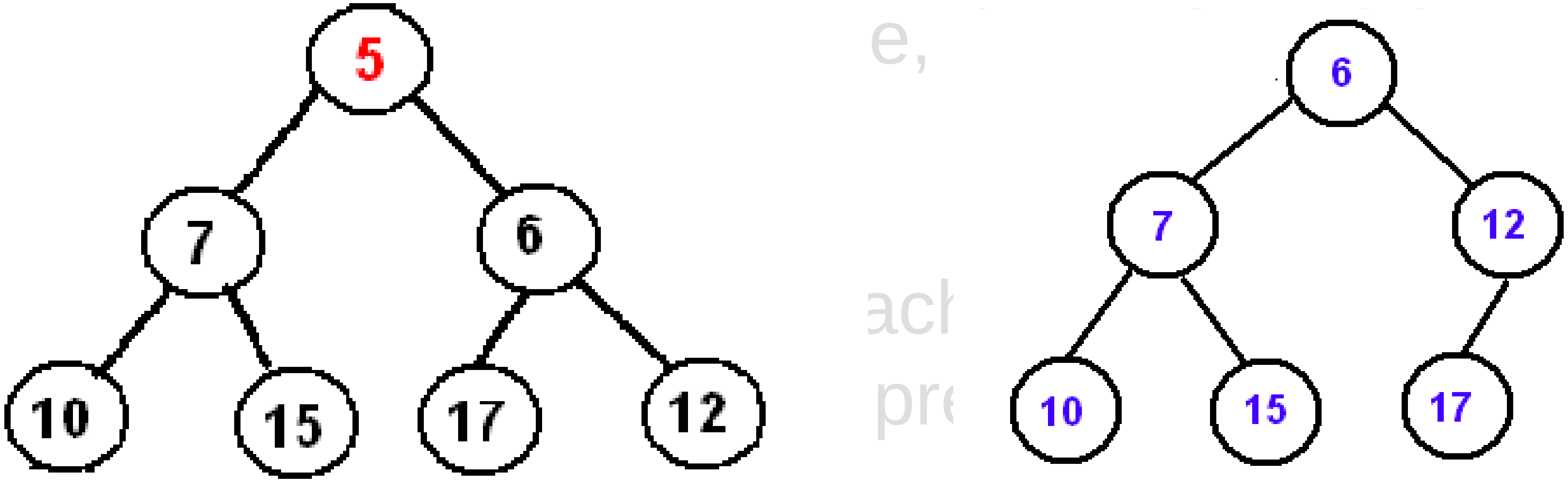
If the last level is not complete, the nodes of that level are filled from left to right

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

Binary Heap

- Binary heap properties

All levels of the tree, except possibly the last one are completely filled (**2^i** nodes at the **ith-level**)



If e,
leve

Ea ach
acc pre d
for the entire data structure

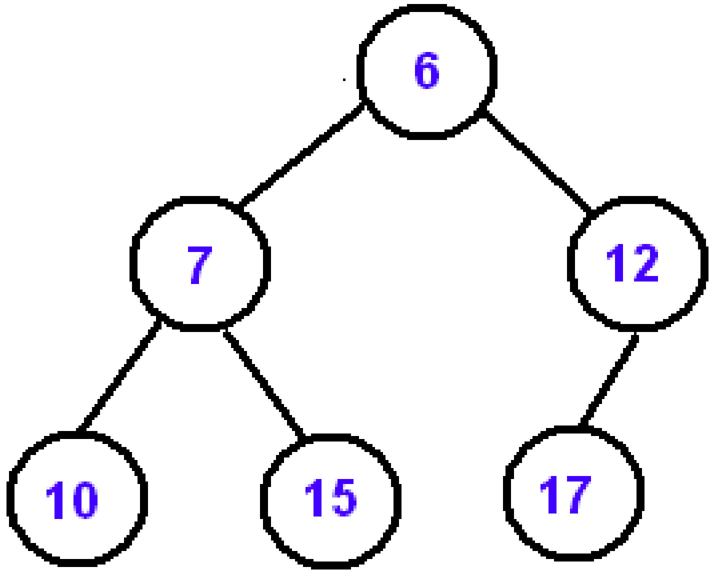
Binary Heap

- Binary heap properties

All levels of the tree, except possibly the last one are completely filled (**2^i** nodes at the **ith-level**)

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

Each node is **“>=”** or **“<=”** each according to some comparison pre for the entire data structure

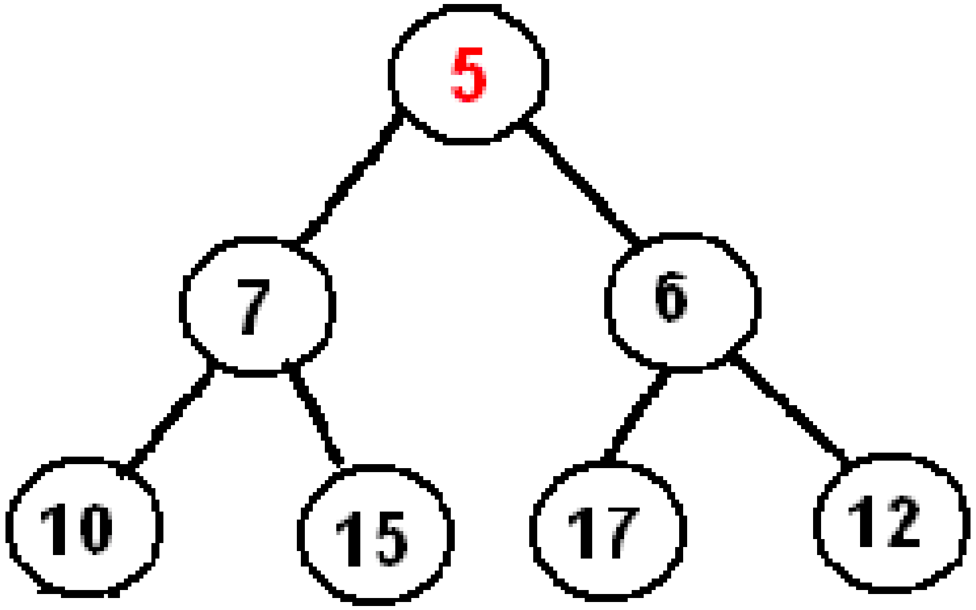


Binary Heap

- Binary heap properties

All levels of the tre ne are
completely filled (2^i

If the last level is n at
level are filled from le



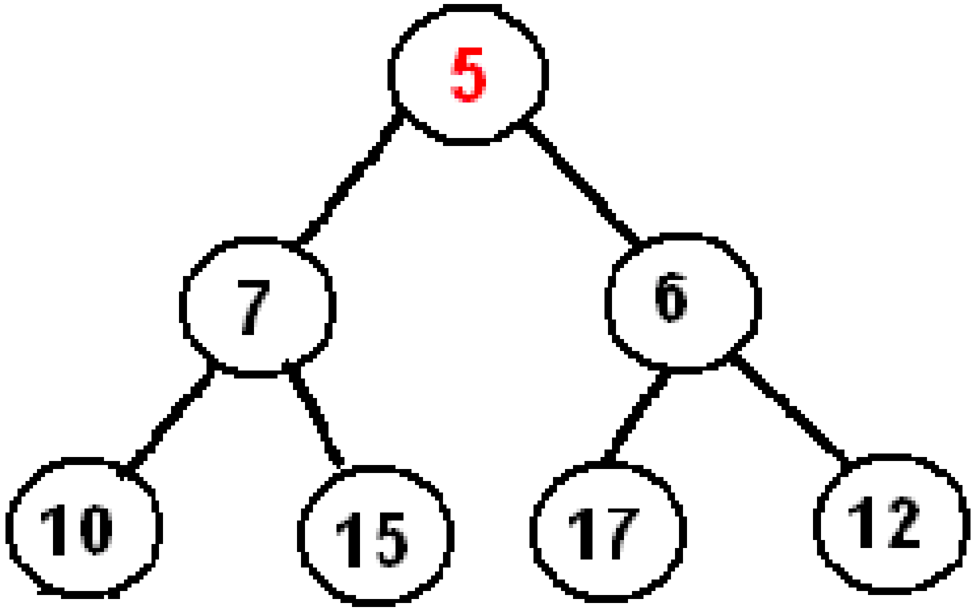
Each node is **“>=”** or **“<=”** each of its children according to some comparison predicate which s fixed for the entire data structure

Binary Heap

- The order of the children is not specified

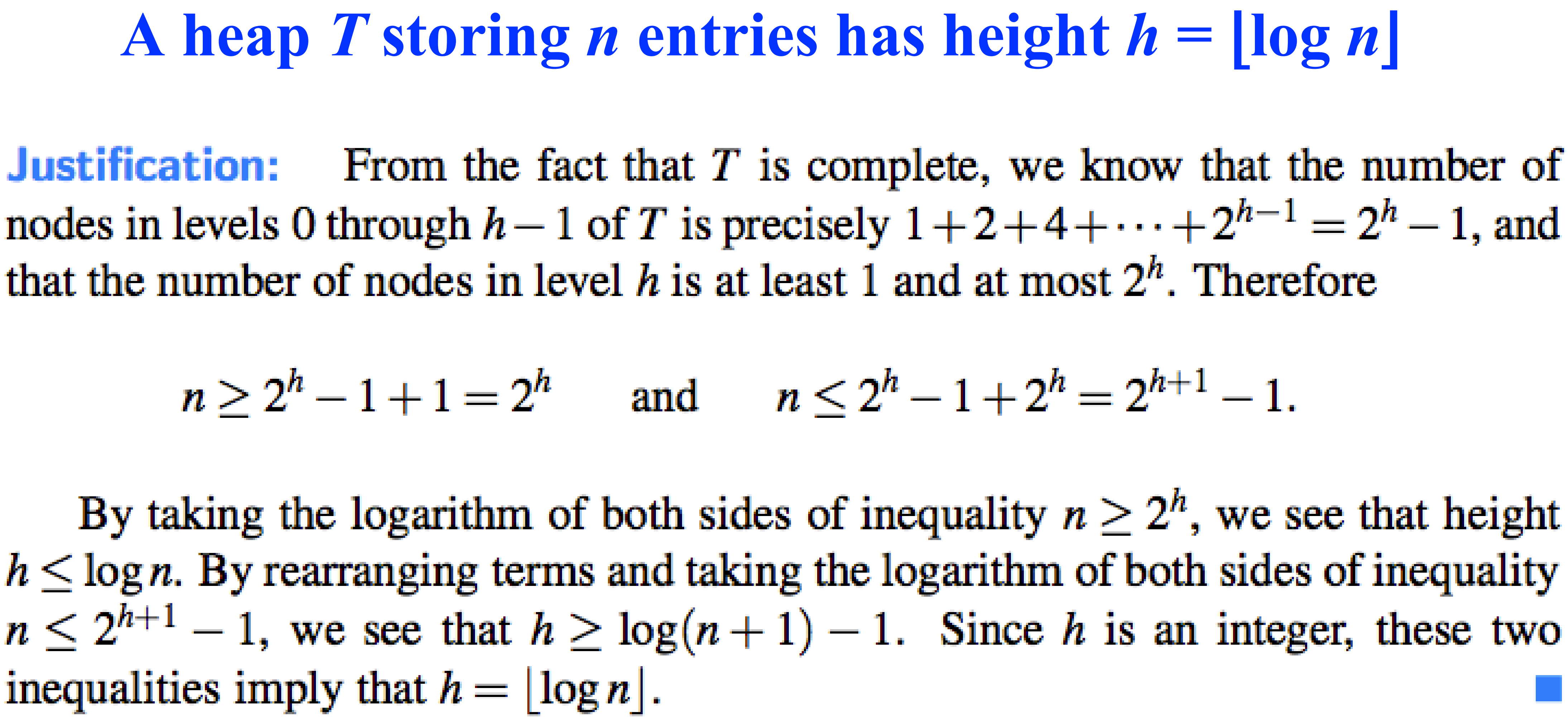
  Two children can be freely interchanged

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



Binary Heap

- Proposition:



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

Justification: From the fact that T is complete, we know that the number of nodes in levels 0 through $h-1$ of T is precisely $1+2+4+\cdots+2^{h-1} = 2^h - 1$, and that the number of nodes in level h is at least 1 and at most 2^h. Therefore

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

By taking the logarithm of both sides of inequality $n \geq 2^h$, we see that height $h \leq \log n$. By rearranging terms and taking the logarithm of both sides of inequality $n \leq 2^{h+1} - 1$, we see that $h \geq \log(n+1) - 1$. Since h is an integer, these two inequalities imply that $h = \lfloor \log n \rfloor$. ■

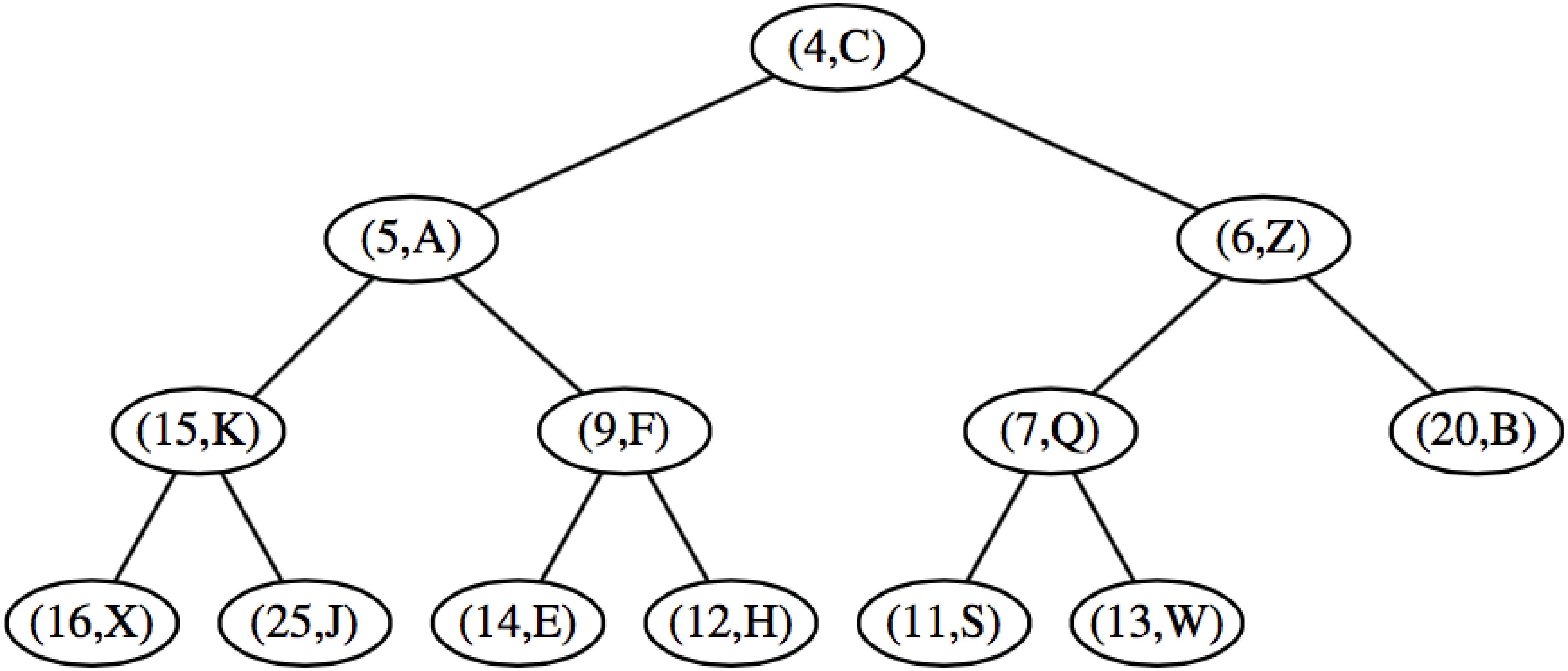
Binary Heap

- **Adding to a heap**
- Algorithm: **upheap / heapify-up / shift-up — O(log n)**

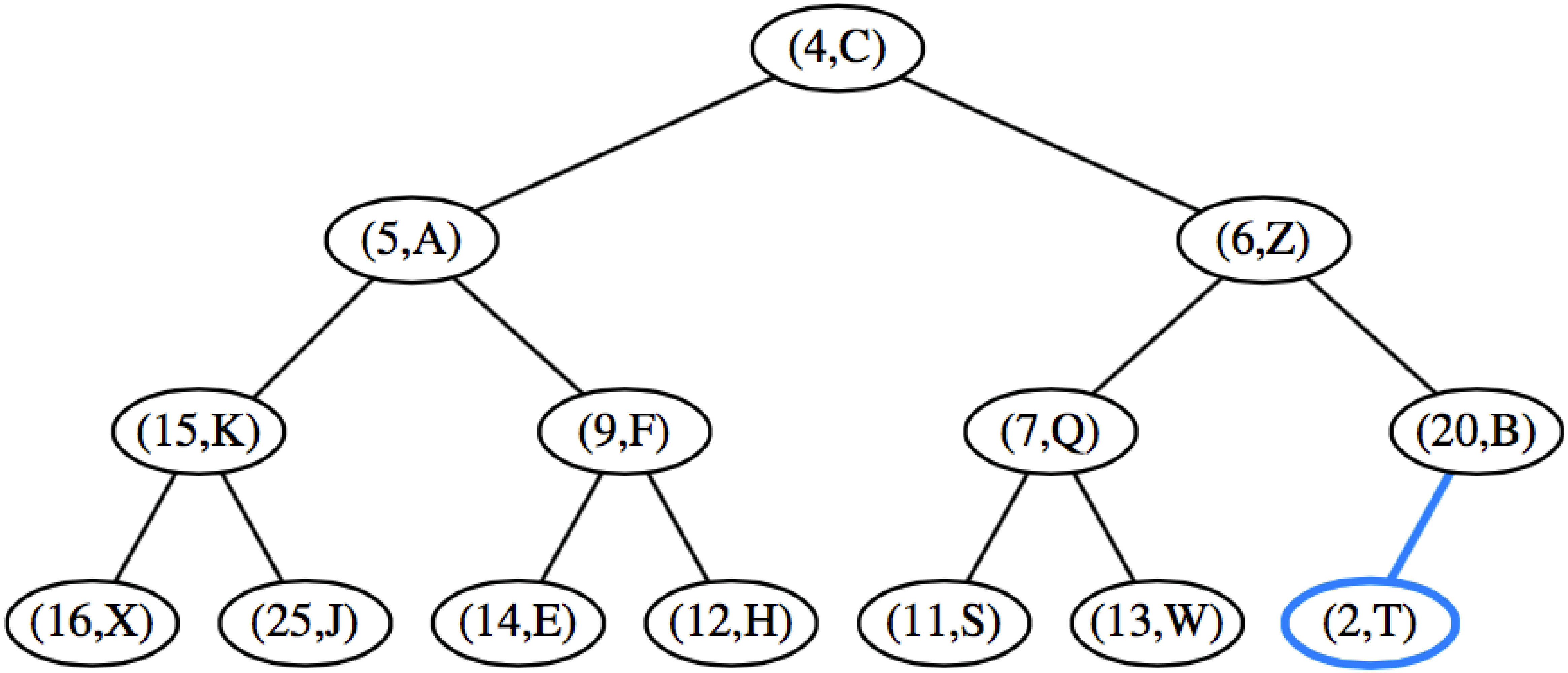
1. Add element to the bottom level
2. Compare the added element with its parent; if they are in correct order, stop
3. If not, swap the element with its parent and return to previous step

Binary Heap

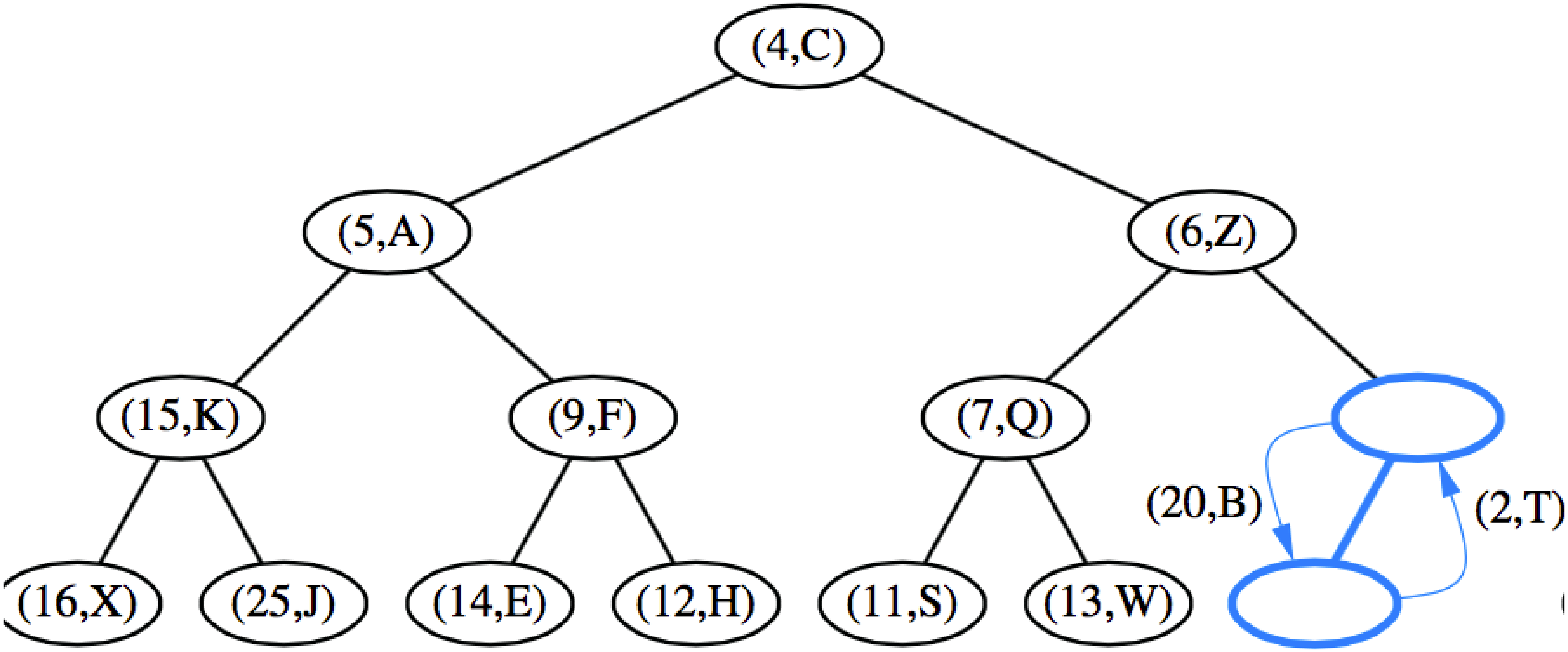
- Insert an item with key 2 into the following heap



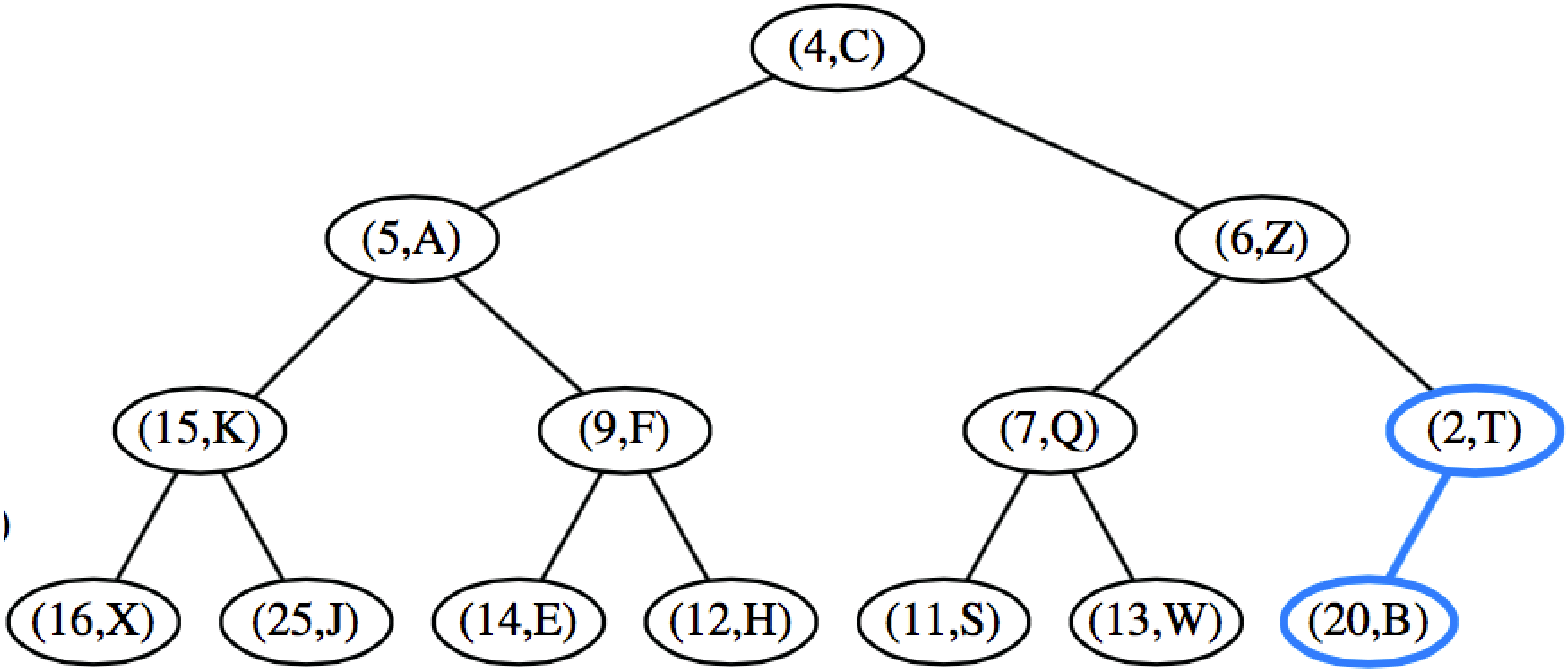
Binary Heap



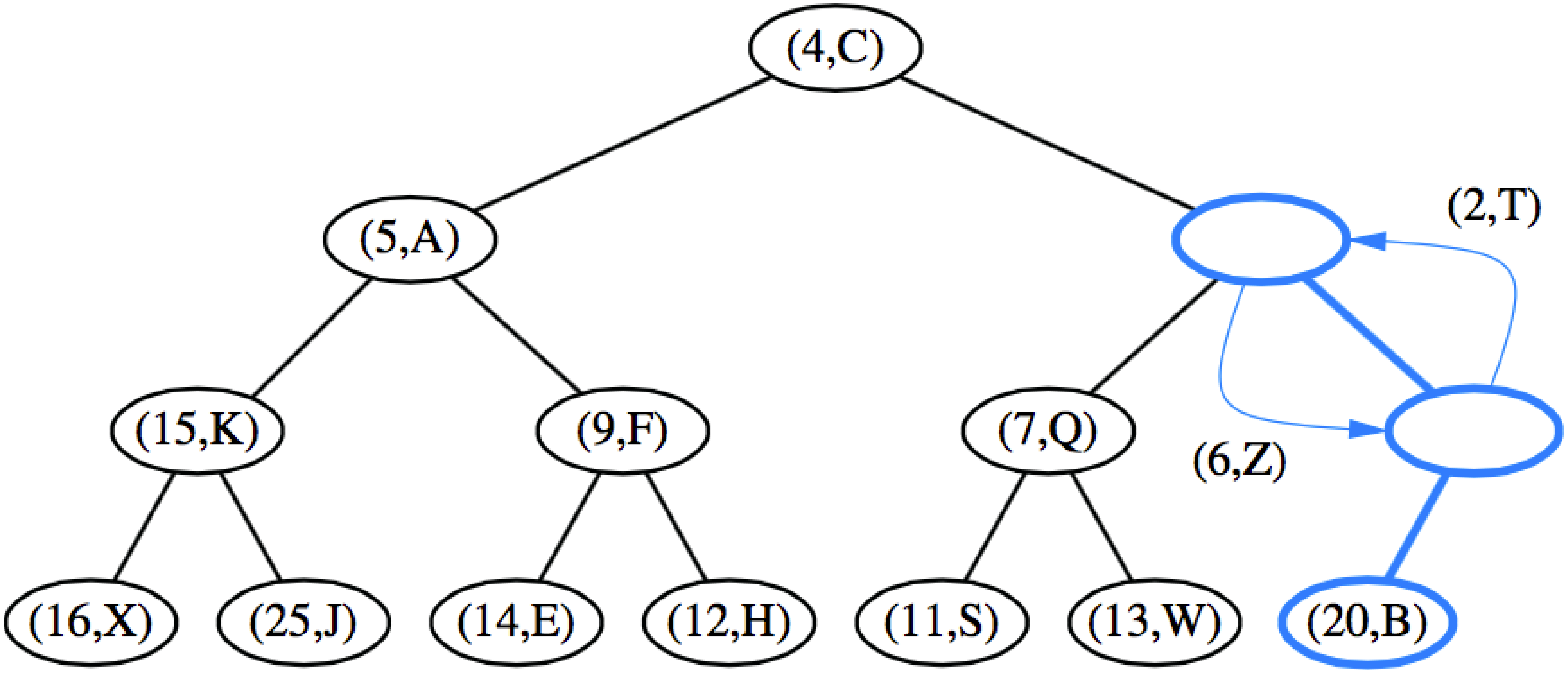
Binary Heap



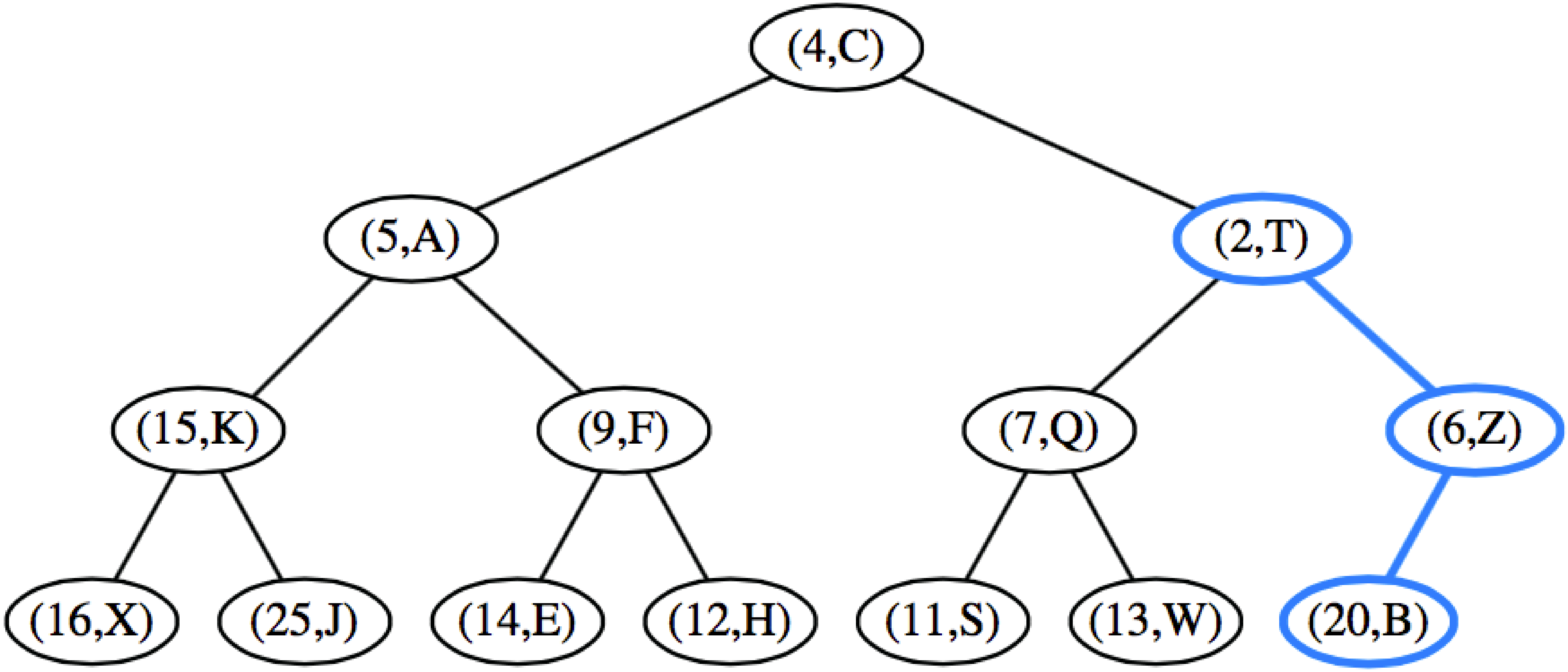
Binary Heap



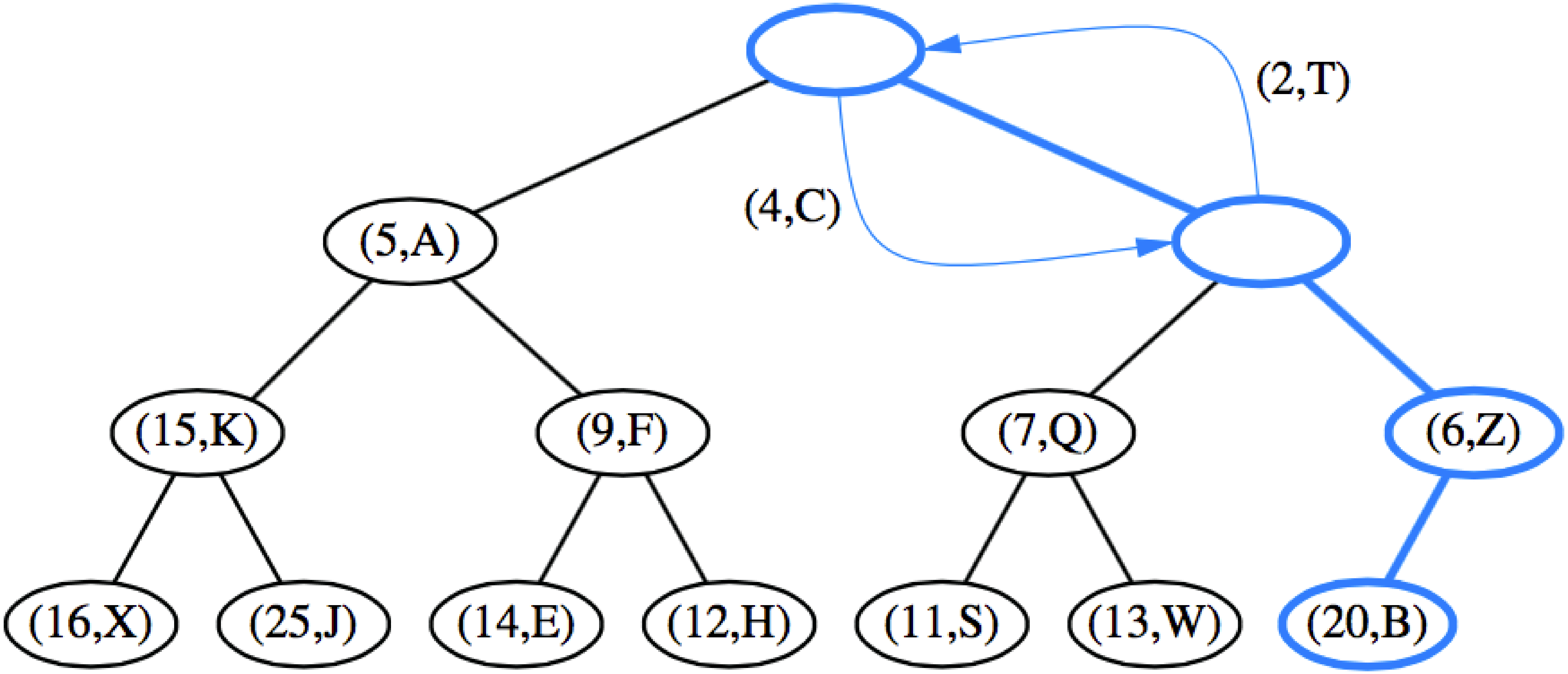
Binary Heap



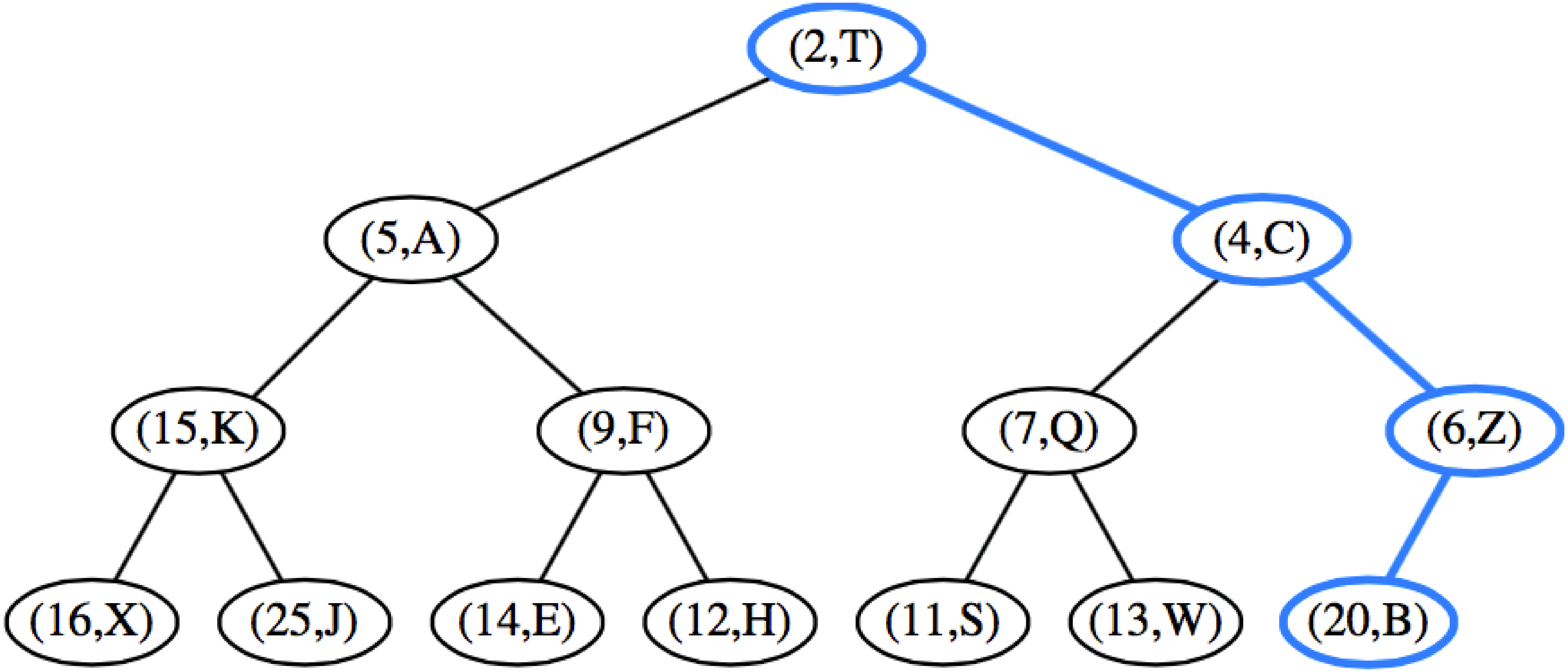
Binary Heap



Binary Heap



Binary Heap



Binary Heap

- **Removing from a heap**

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

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

Binary Heap

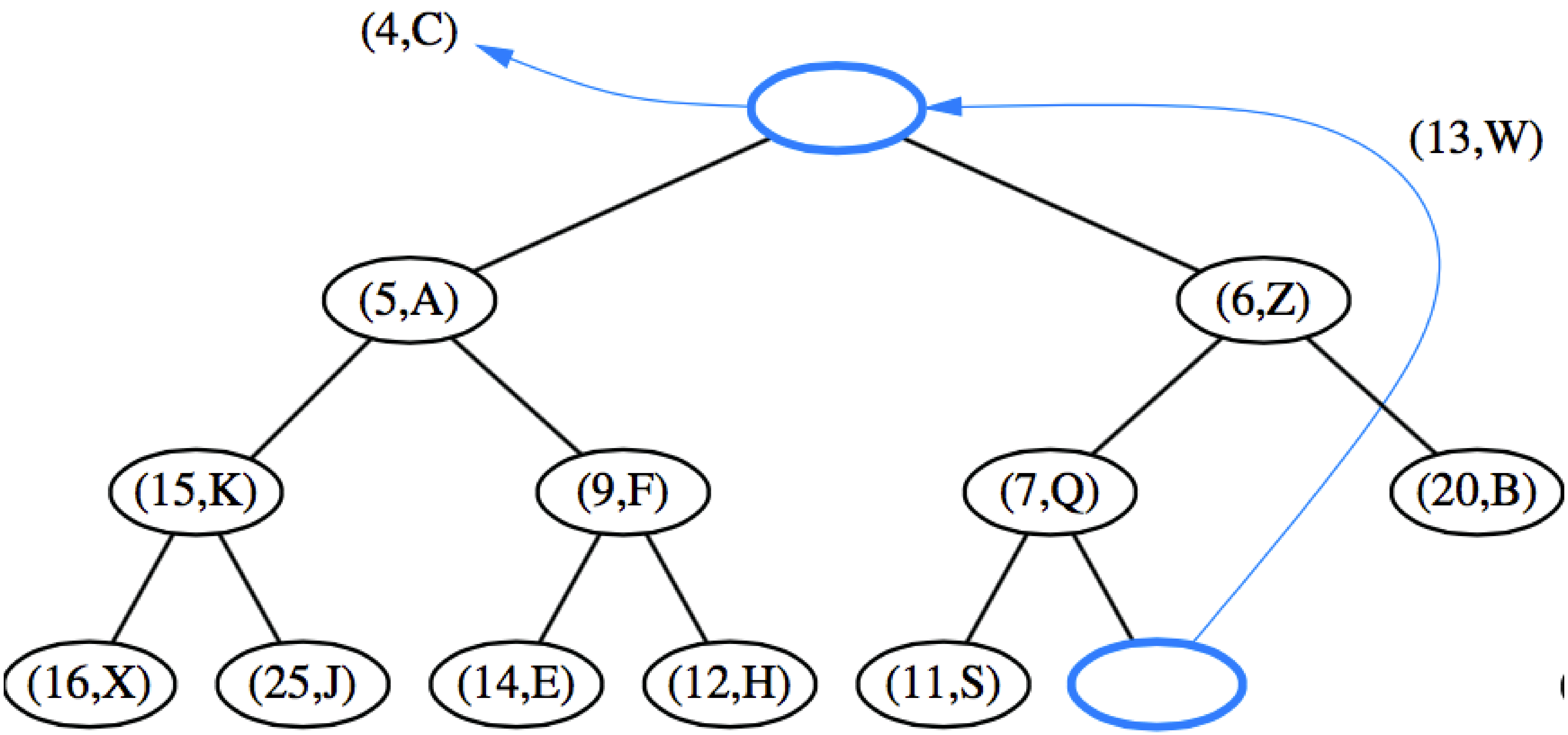
- Algorithm: **downheap / heapify-down / shift-down — O(log n)**
  - Replace root with the last element on the bottom level
  - Compare the swapped element with

    The larger child (max-heap)

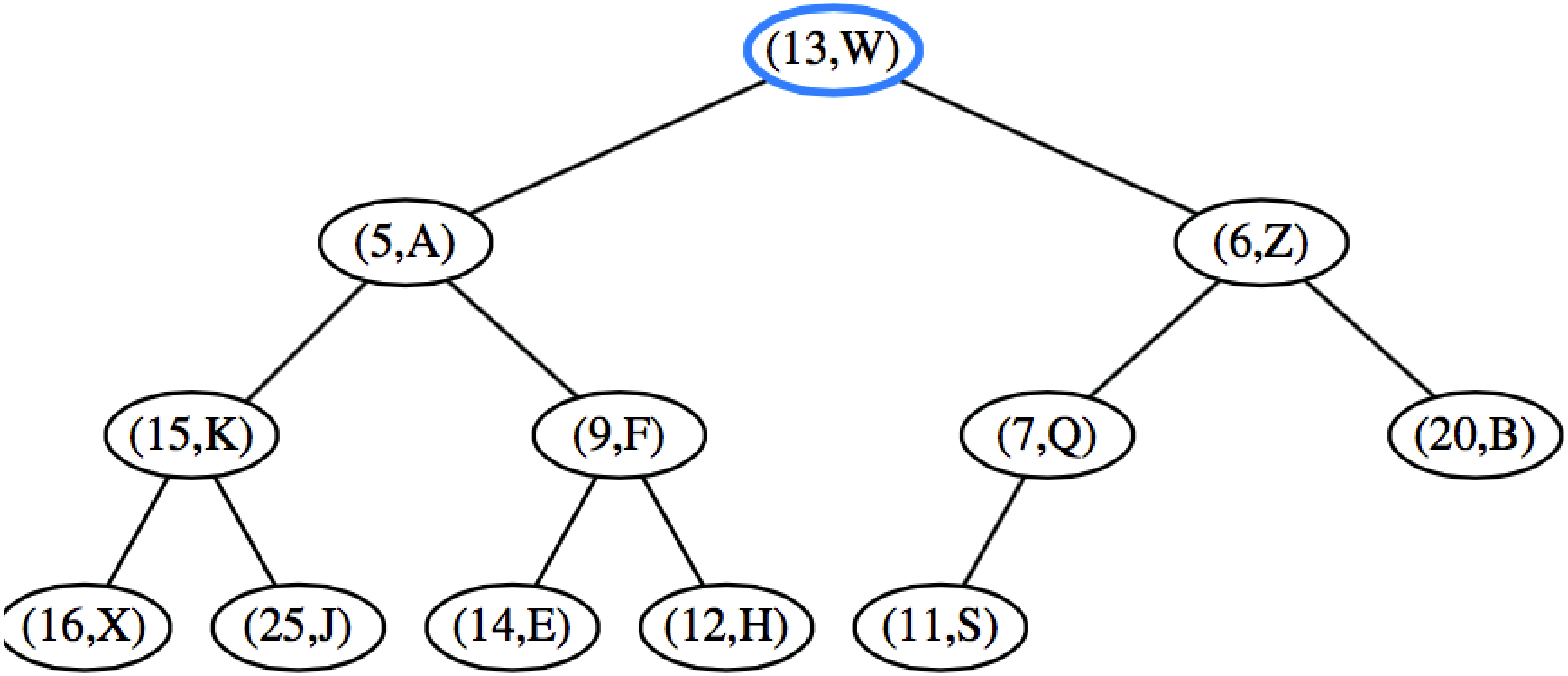
    The smaller child (min-heap)
  - If they are in correct order, stop
  - If not, swap the element with the child and return to previous step

Binary Heap

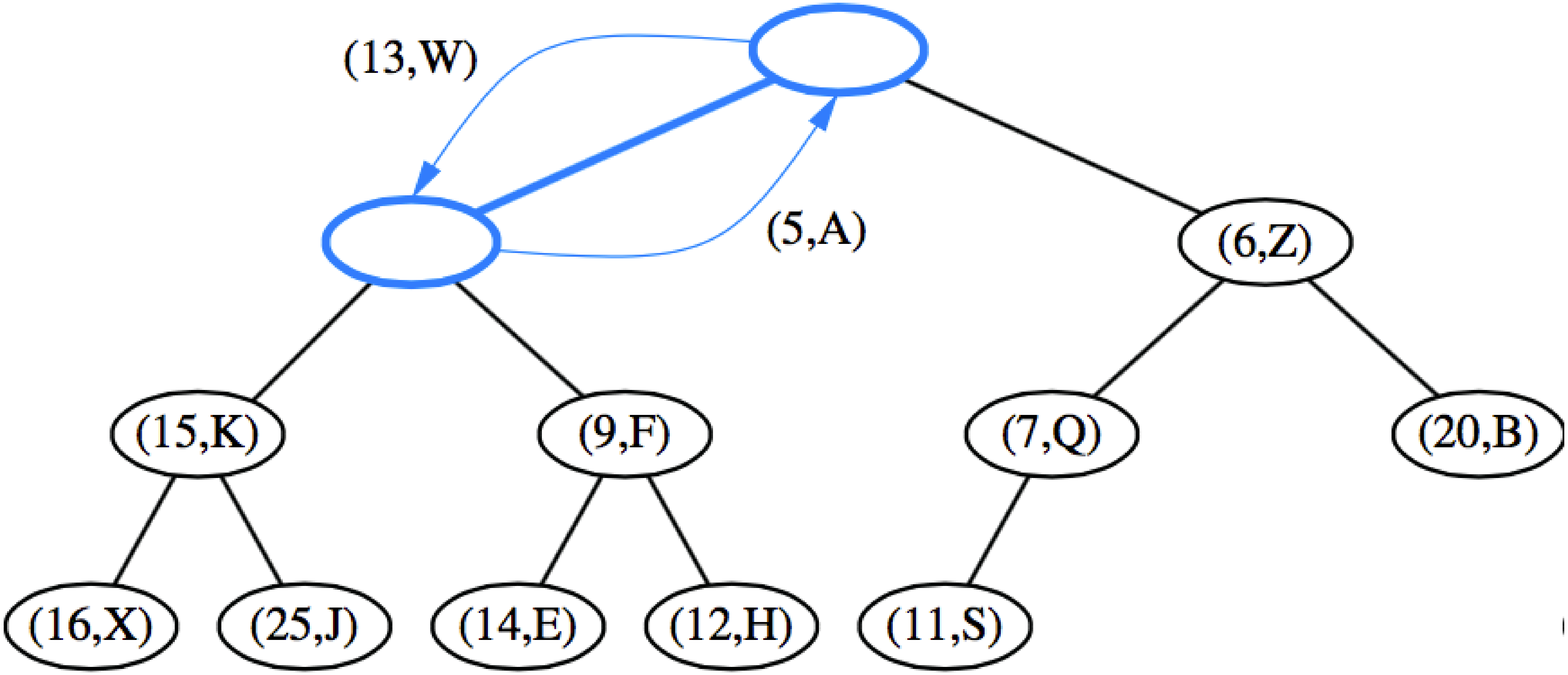
- Deletion in a binary heap



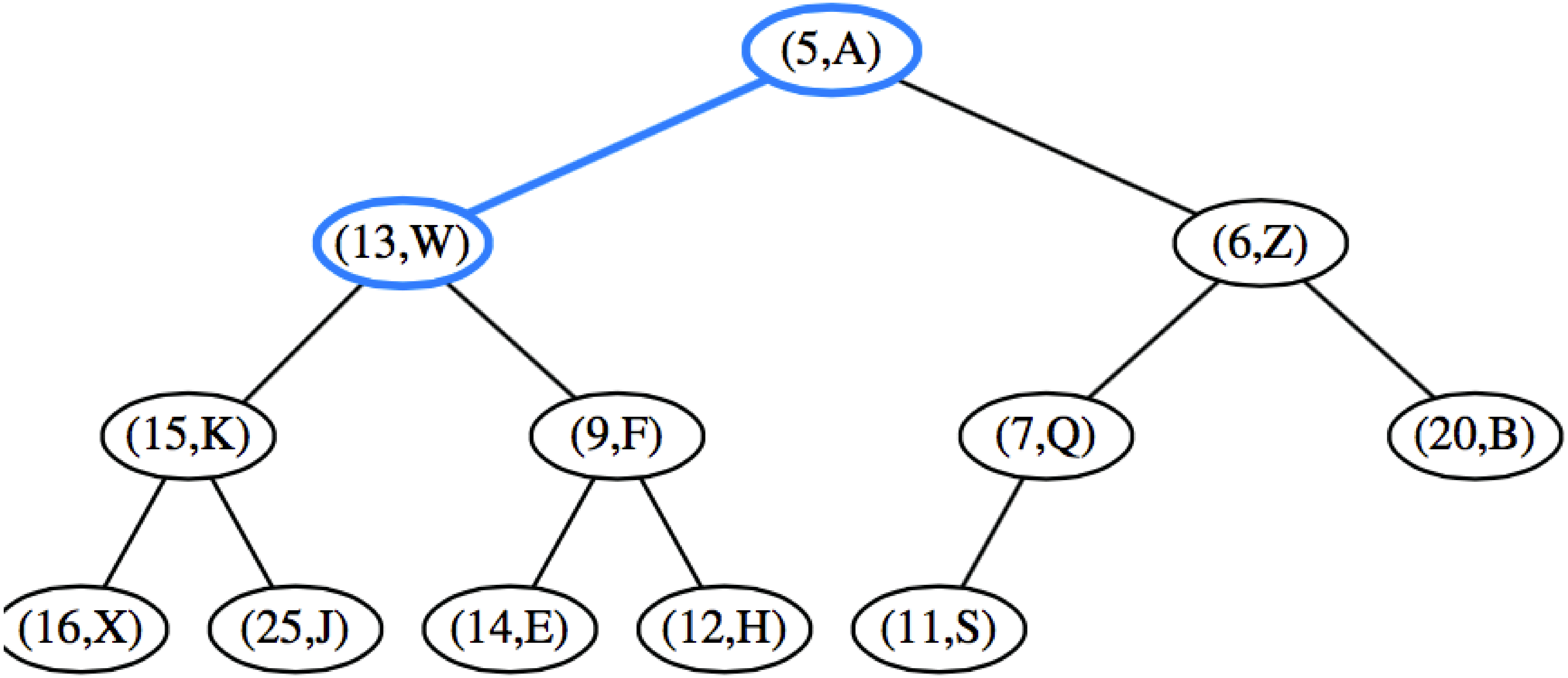
Binary Heap



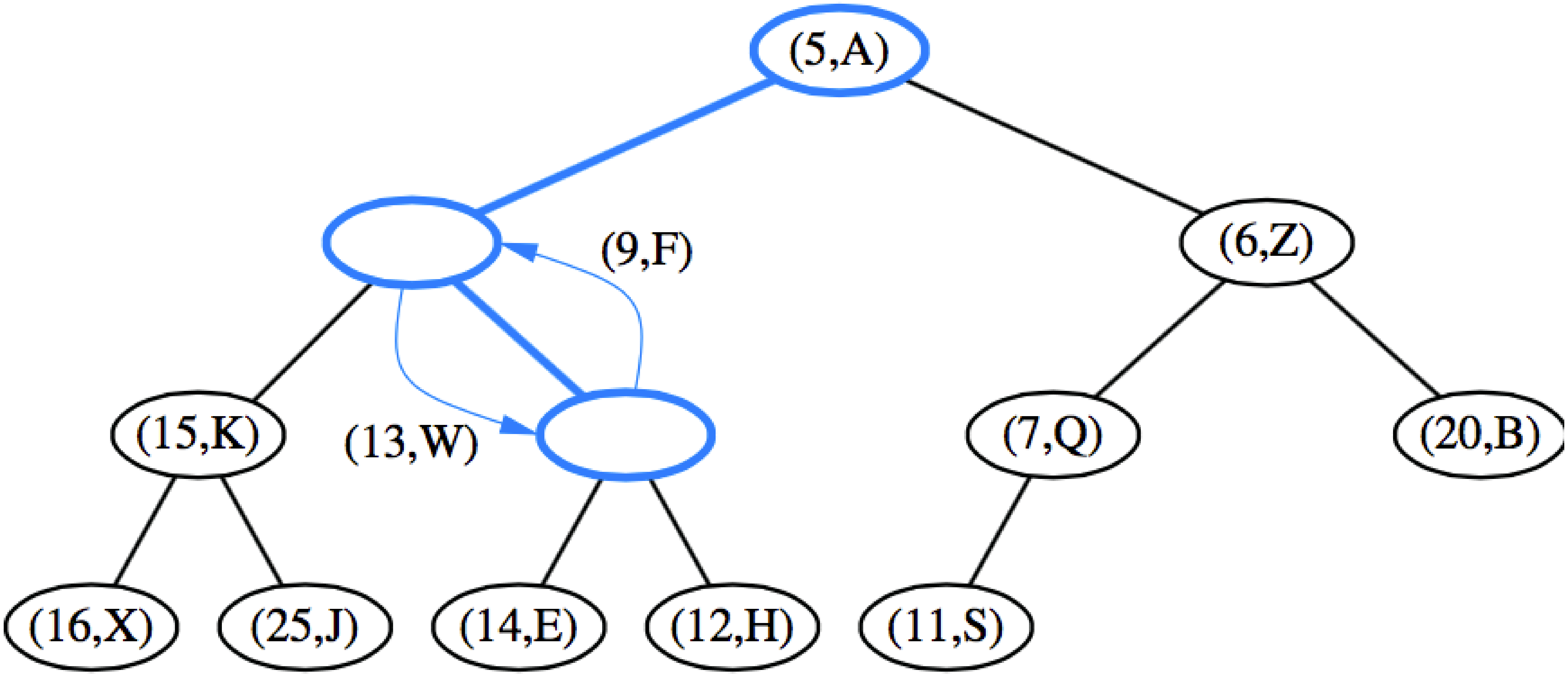
Binary Heap



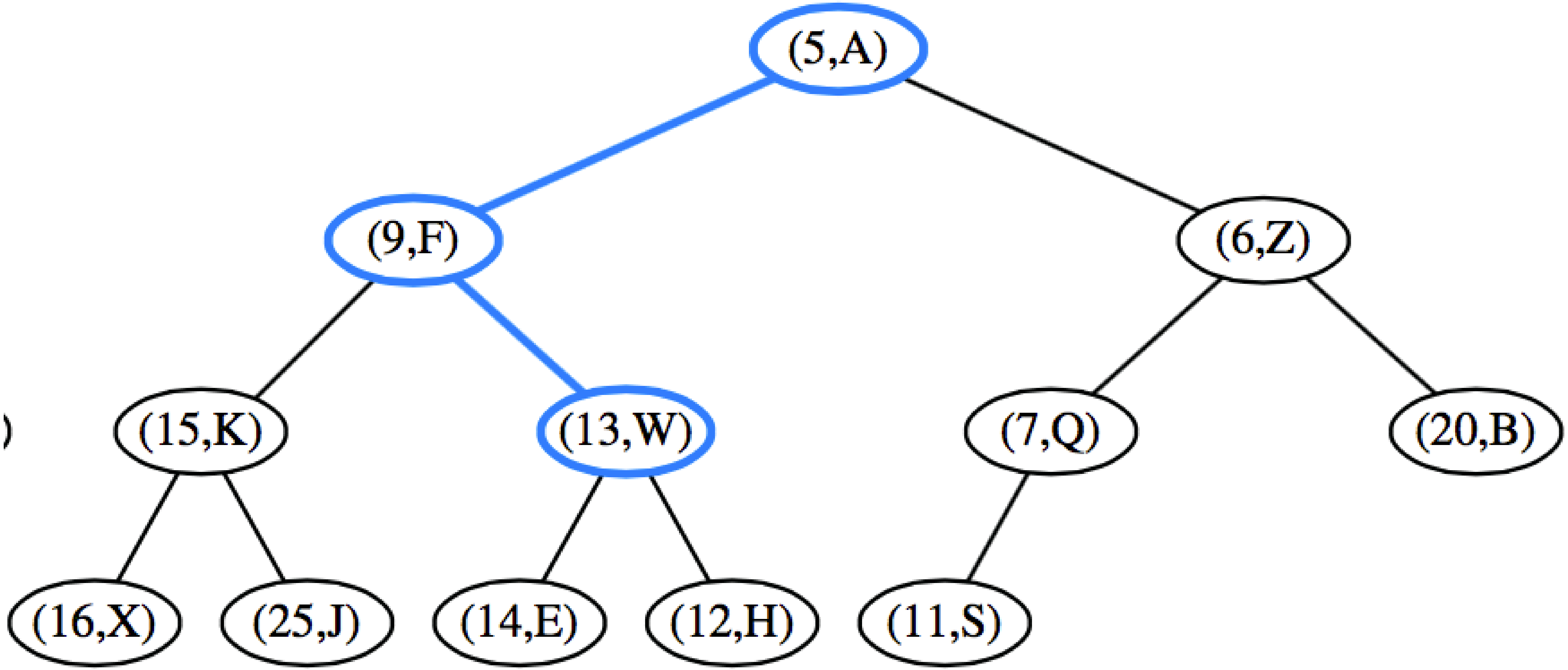
Binary Heap



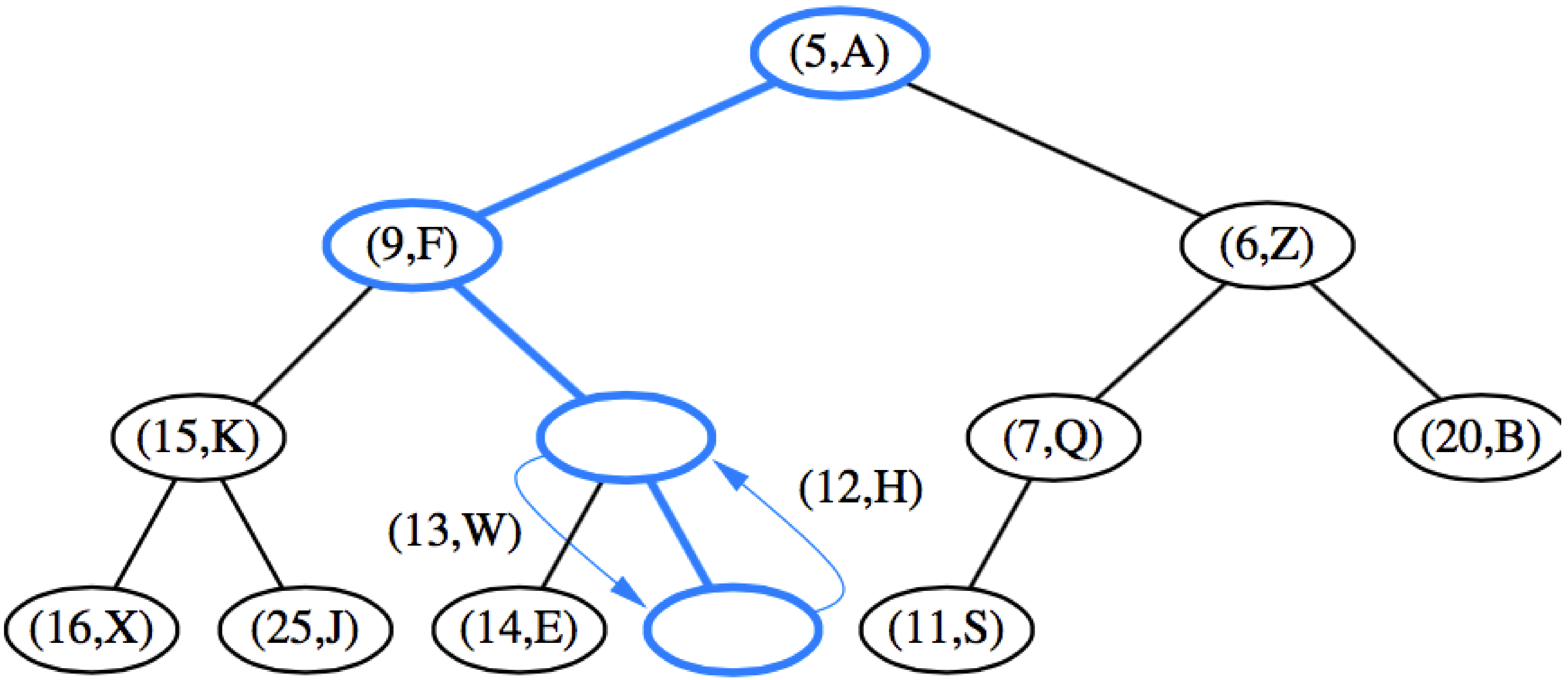
Binary Heap



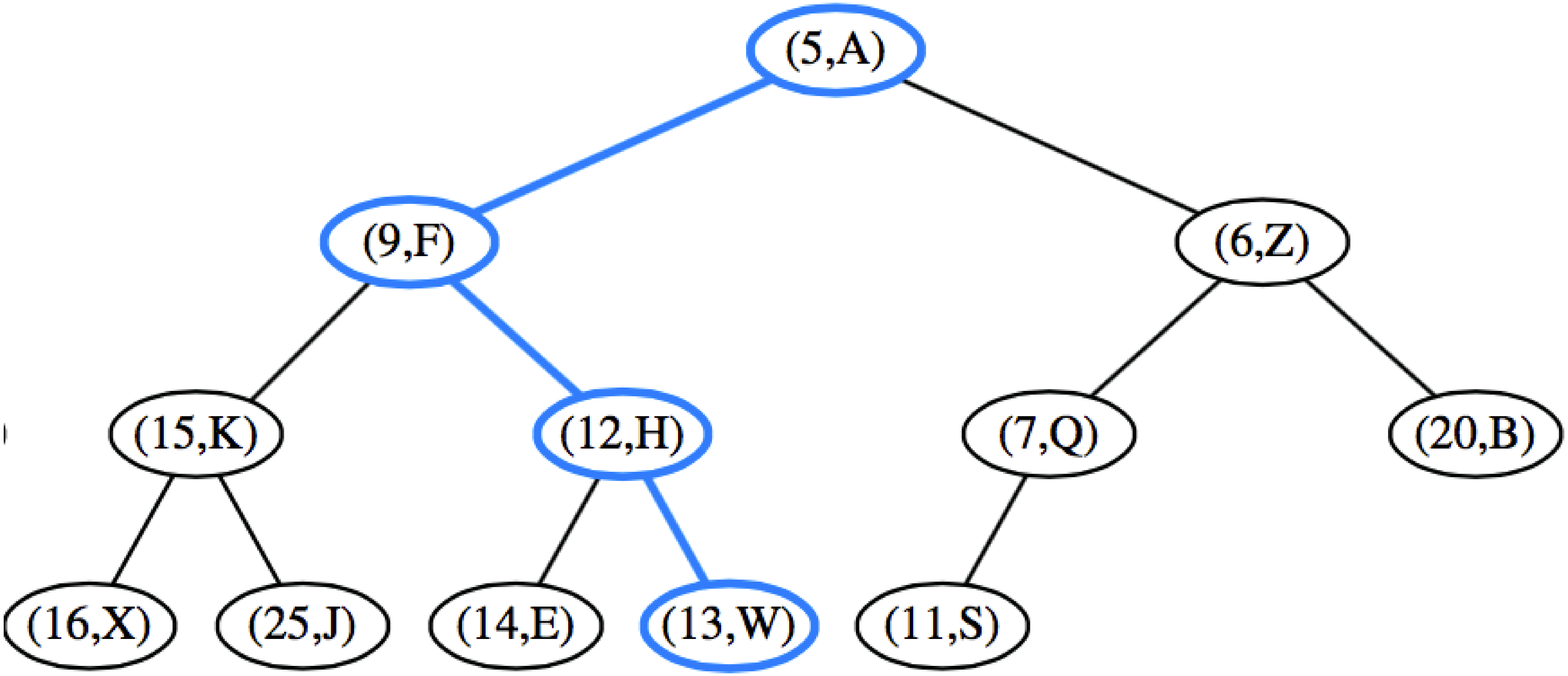
Binary Heap



Binary Heap



Binary Heap



Binary Heap

- **Implementation as a binary tree**

Problem finding adjacent element on the last level

Finding it algorithmically takes time

Using additional links between siblings: threading the tree takes space

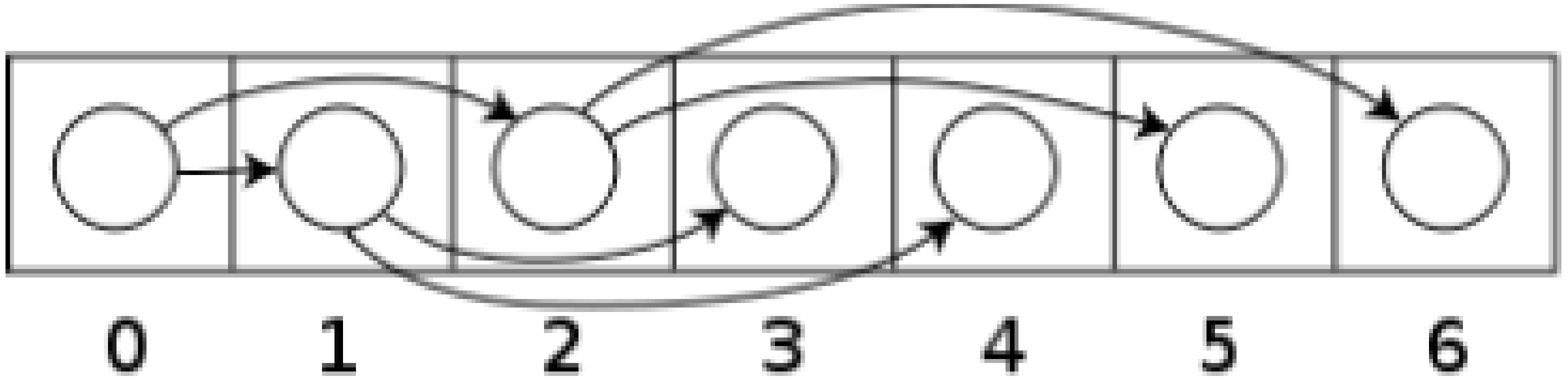
Binary Heap

- **Implementation as an array**

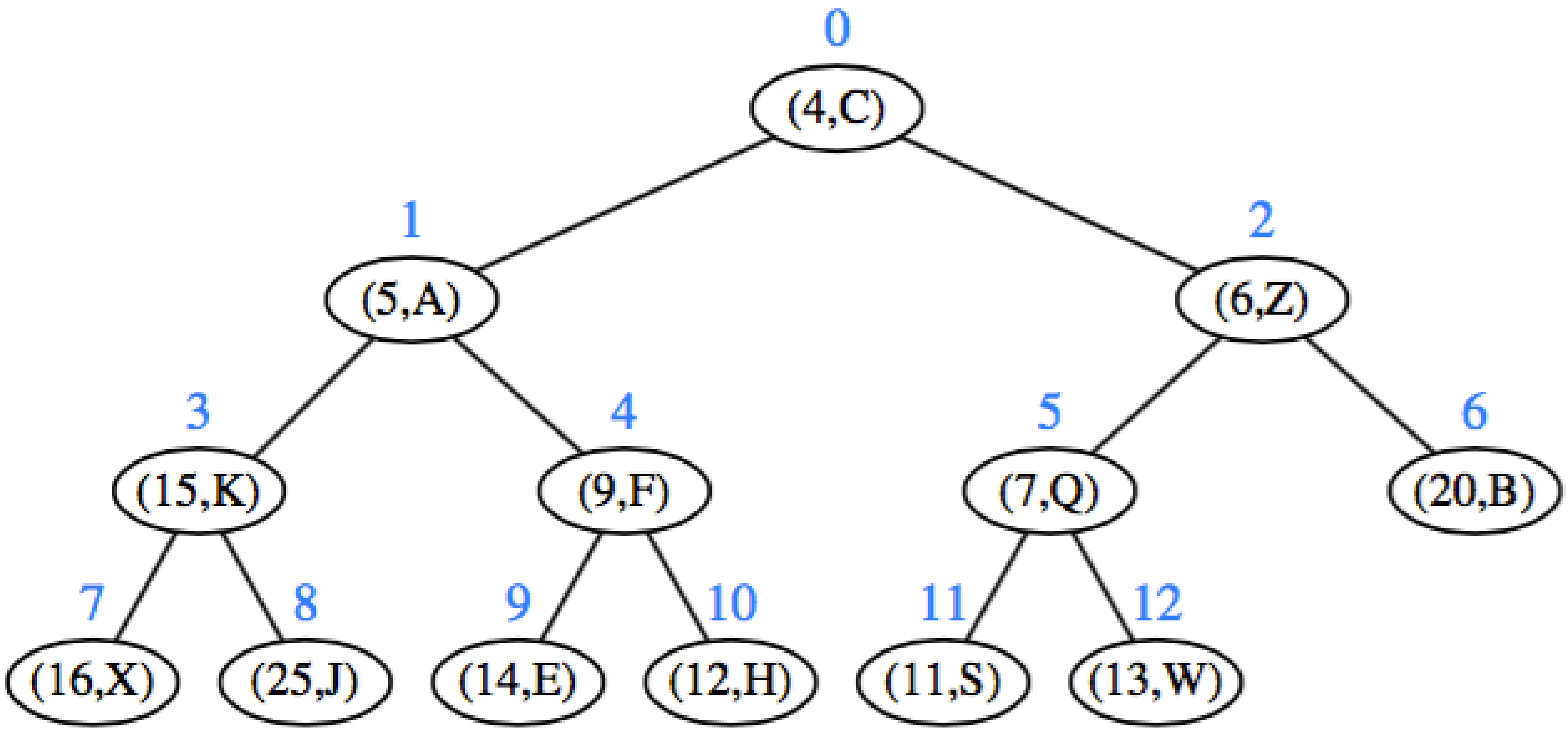
Represent a binary tree without any pointers by using an array of keys and a mapping function

Mapping functions helps find parents and children of a node

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



Binary Heap



(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

Binary Heap

- **Inserting in an array based heap**

```
Algorithm InsertInHeap(k, v)
Input: priority k, value v; Output: none
H[size] = new entry (k, v) // insert entry (k, v) at rank = size of array
size = size + 1  // increase heap size
```

Binary Heap

```
// Now perform upheap, starting at the last node
i = size - 1
while i > 0 and H[(i-1)/2].key() > k
    swap(H[i], H[i/2])   // swap entry (k, v) with the entry at parent
                          node
     i = (i-1)/2                  // after this statement, index i holds entry (k, v)
```

Binary Heap

- **Deleting in an array based heap**

```
Algorithm RemoveMin()
Input: none; Output: entry with the smallest key
if size == 0 then ReportError("Empty Heap")
itemToReturn = H[0] // minimum is at rank 0
H[0] = H[size-1] // put the entry at last rank at root location
size = size - 1  // decrease heap size
```

Binary Heap

```
// Now perform downheap to restore heap order
i = 0
childIndex = findSmallerChild(i)
while (childIndex != 0 && H[childIndex].key < H[i].key)
    swap(H[childIndex], H[i])
    i = childIndex
    childIndex = findSmallerChild(i)
return itemToReturn
```

Binary Heap

```
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**

  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

Merge-Sort

Divide-and-Conquer

- Divide-and-conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two (or more) disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue

Merge-Sort

- Merge-sort on an input sequence ***S*** with ***n*** elements consists of three steps:
  - Divide: partition ***S*** into two sequences S_1 and S_2 of about ***n***/2 elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S)
    Input sequence S with n
    elements
    Output sequence S sorted
        according to C
      if S.size() > 1
        (S_1, S_2) ← partition(S, n/2)
        mergeSort(S_1)
        mergeSort(S_2)
        S ← merge(S_1, S_2)
```

Merging Two Sorted Sequences

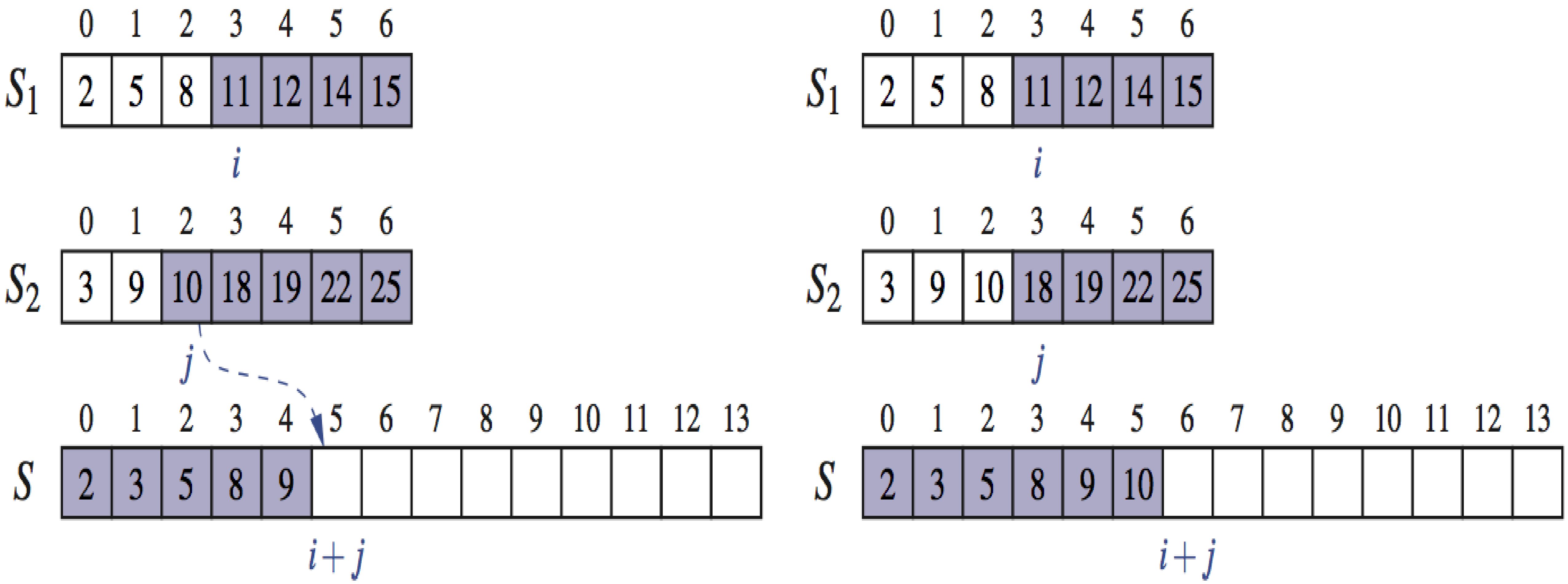
- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***

- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***O***(***n***) time

```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of $A \cup B$

      $S \leftarrow$ empty sequence
      while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() < B.first().element()
            S.addLast(A.remove(A.first()))
        else
            S.addLast(B.remove(B.first()))
      while ¬A.isEmpty()
        S.addLast(A.remove(A.first()))
      while ¬B.isEmpty()
        S.addLast(B.remove(B.first()))
      return S
```

Merge-Sort



0 1 2 3 4 5 6
S_1 2 5 8 11 12 14 15
i
0 1 2 3 4 5 6
S_2 3 9 10 18 19 22 25
j
0 1 2 3 4 5 6 7 8 9 10 11 12 13
S 2 3 5 8 9
$i+j$
0 1 2 3 4 5 6
S_1 2 5 8 11 12 14 15
i
0 1 2 3 4 5 6
S_2 3 9 10 18 19 22 25
j
0 1 2 3 4 5 6 7 8 9 10 11 12 13
S 2 3 5 8 9 10
$i+j$

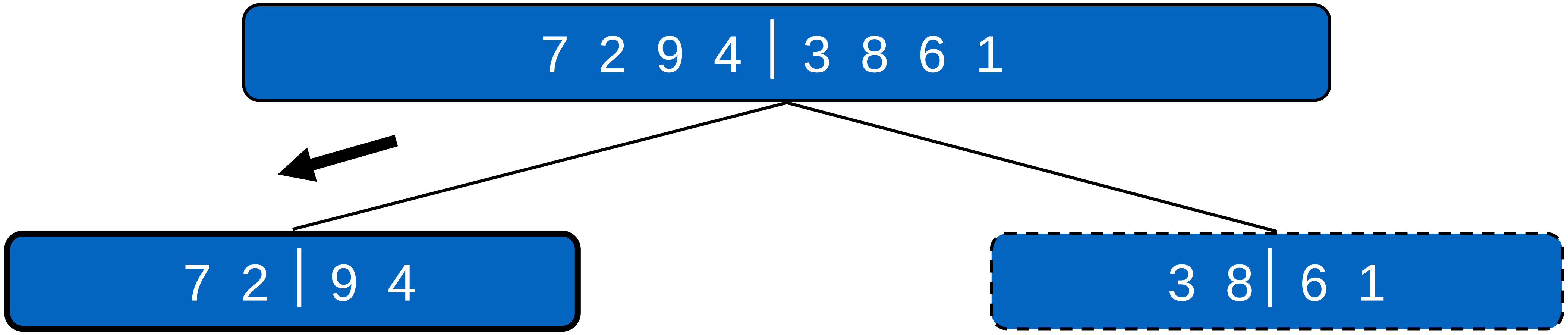
Execution Example

- Partition

7 2 9 4 | 3 8 6 1

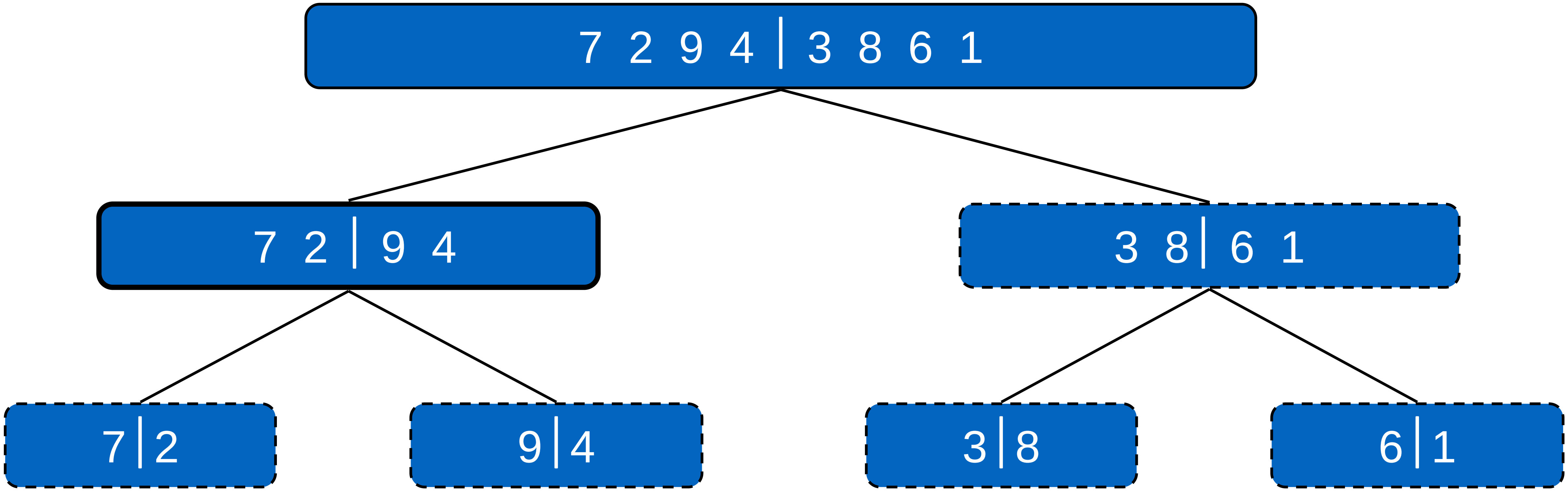
Execution Example (cont.)

- Recursive call, partition



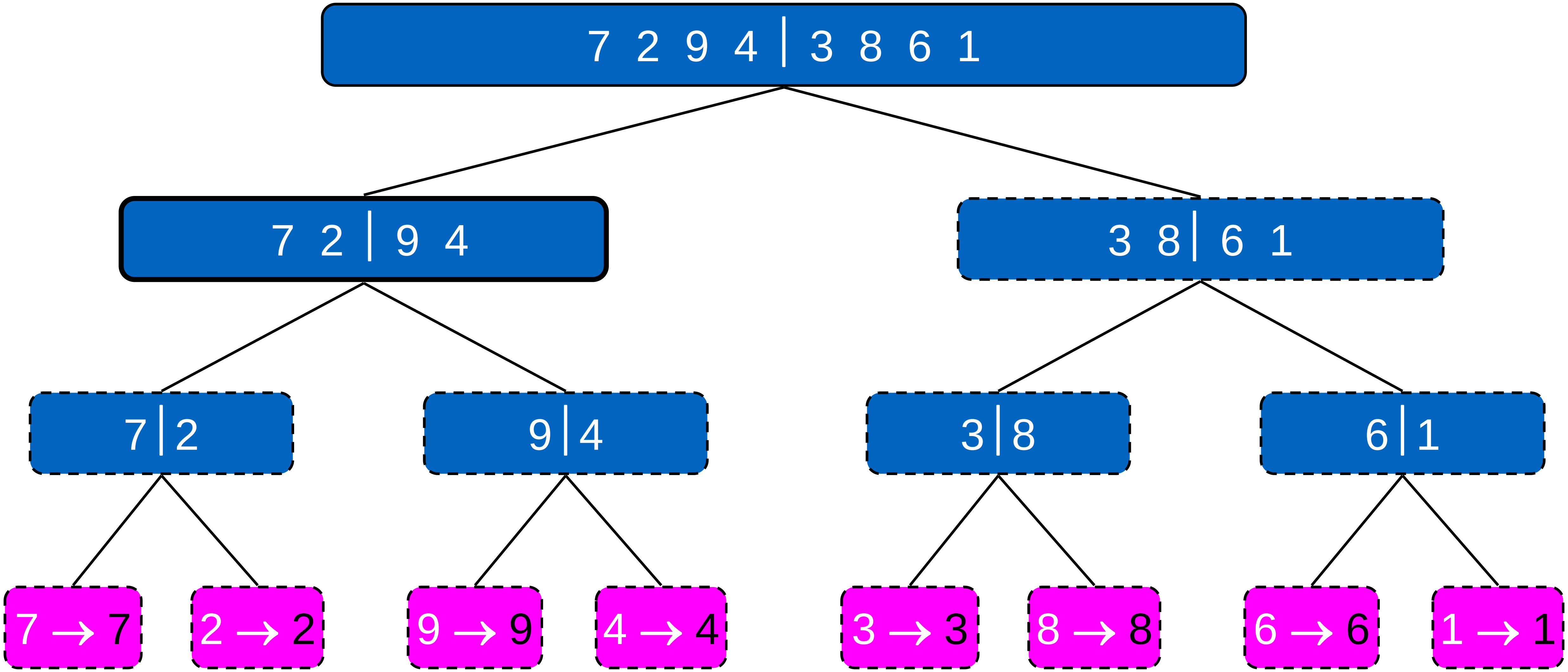
Execution Example (cont.)

- Recursive call, partition



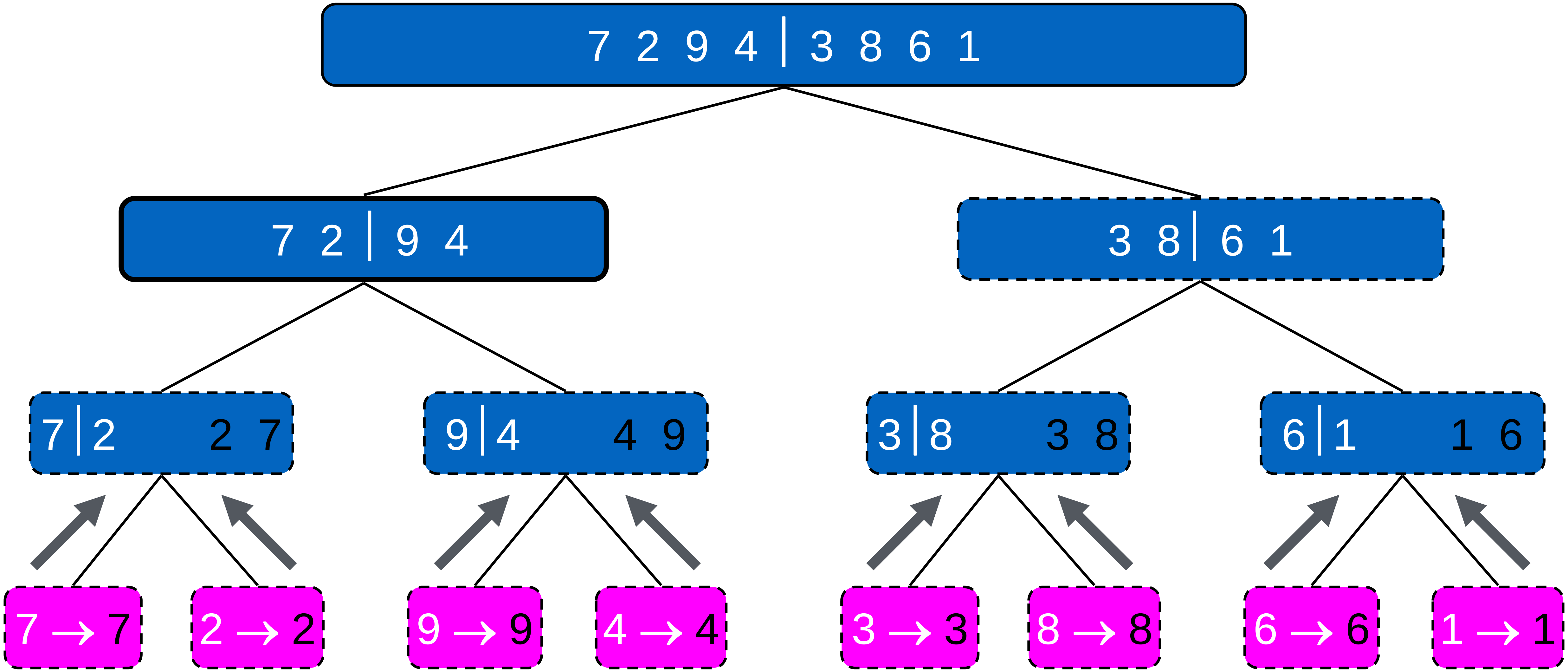
Execution Example (cont.)

- Recursive call, Base Case



Execution Example (cont.)

- Merge



7 2 9 4 | 3 8 6 1

7 2 | 9 4

3 8 | 6 1

7 | 2 2 7

9 | 4 4 9

3 | 8 3 8

6 | 1 1 6

7 → 7

2 → 2

9 → 9

4 → 4

3 → 3

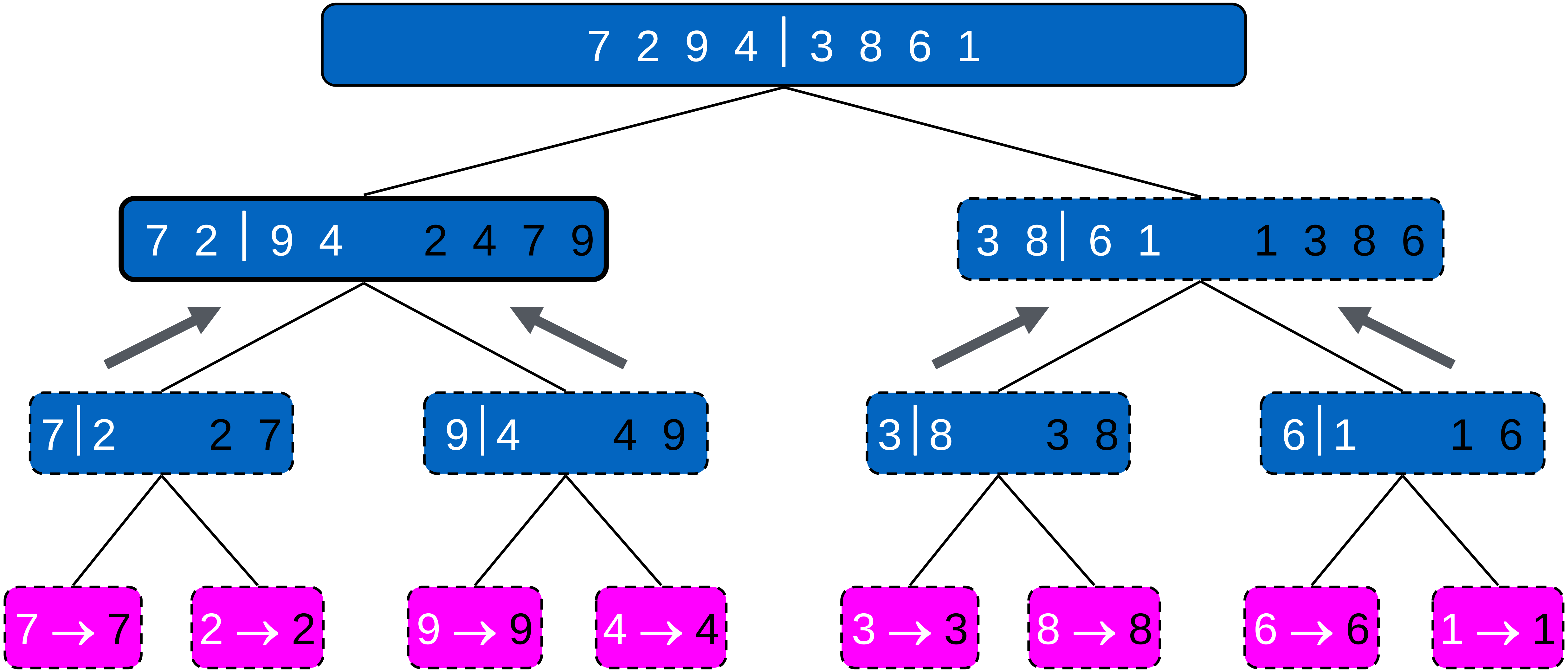
8 → 8

6 → 6

1 → 1

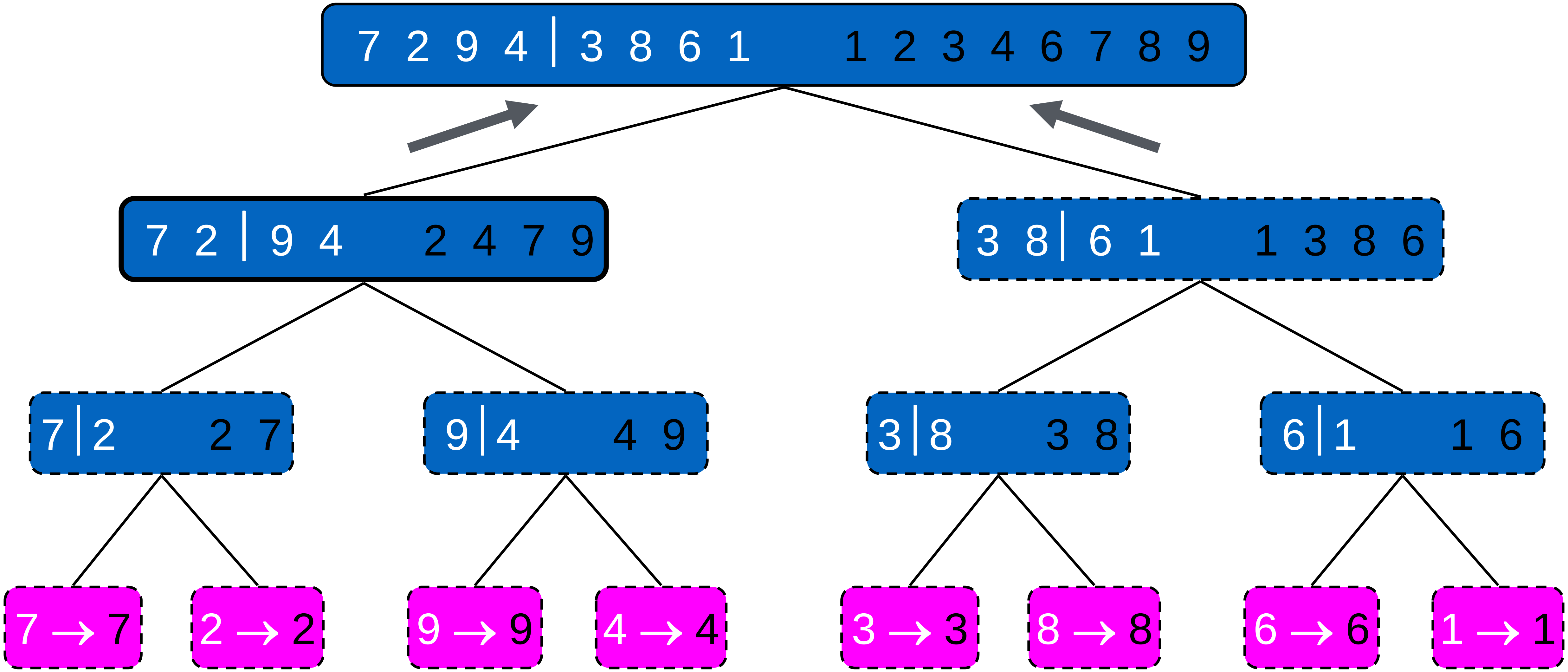
Execution Example (cont.)

- Merge



Execution Example (cont.)

- Merge



7 2 9 4 | 3 8 6 1 1 2 3 4 6 7 8 9

7 2 | 9 4 2 4 7 9

3 8 | 6 1 1 3 8 6

7 | 2 2 7

9 | 4 4 9

3 | 8 3 8

6 | 1 1 6

7 → 7

2 → 2

9 → 9

4 → 4

3 → 3

8 → 8

6 → 6

1 → 1

Analysis of Merge-Sort

- The height ***h*** of the merge-sort tree is ***O***(log ***n***)
- The overall amount or work done at the nodes of depth ***i*** is ***O***(***n***)
- Thus, the total running time of merge-sort is ***O***(***n*** log ***n***)

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…