Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Recap

- 2-3-4 Trees
- B-Trees
- RB Trees

Today’s Objectives

- Priority Queues
- Binary Heap
- Heap-Sort
- Merge-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

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

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)

Binary Heap

- A **complete binary tree**

Binary Heap

- A **complete binary tree**

➢ Filled out on every level, expect perhaps on the last one

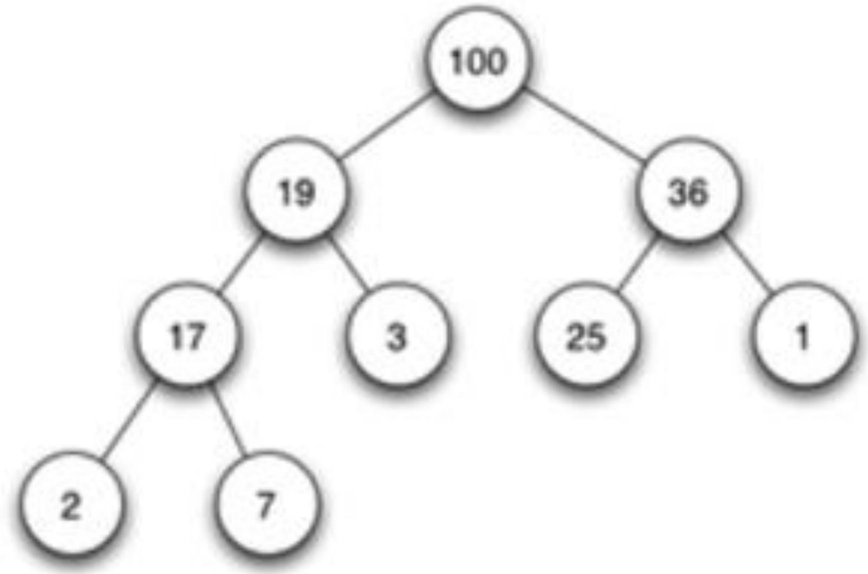
➢ All nodes on the last level, should be as far to left as possible

Binary Heap

- A **complete binary tree**

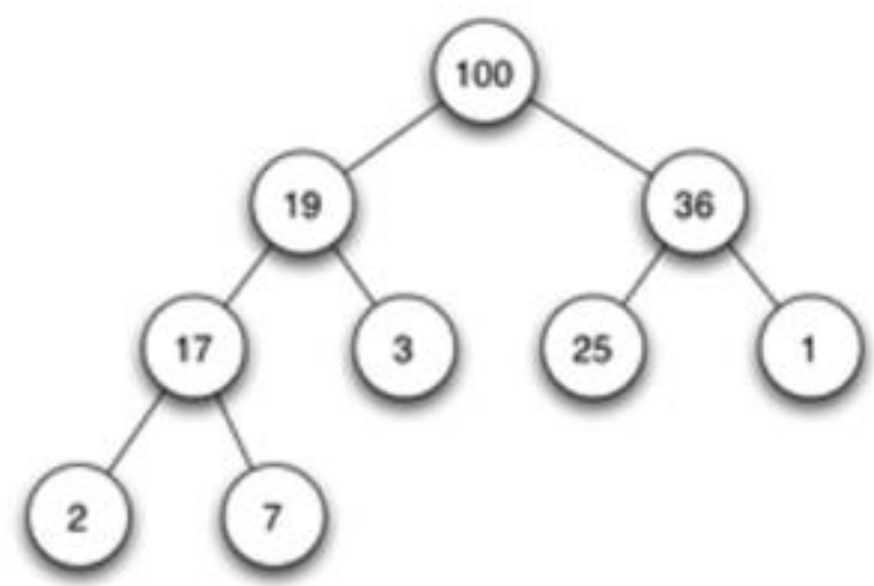
➢ Filled out on every level, expect perhaps on the last one

➢ All nodes on the last level, should be as far to left as possible



Binary Heap

- Maintains 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)



Binary Heap

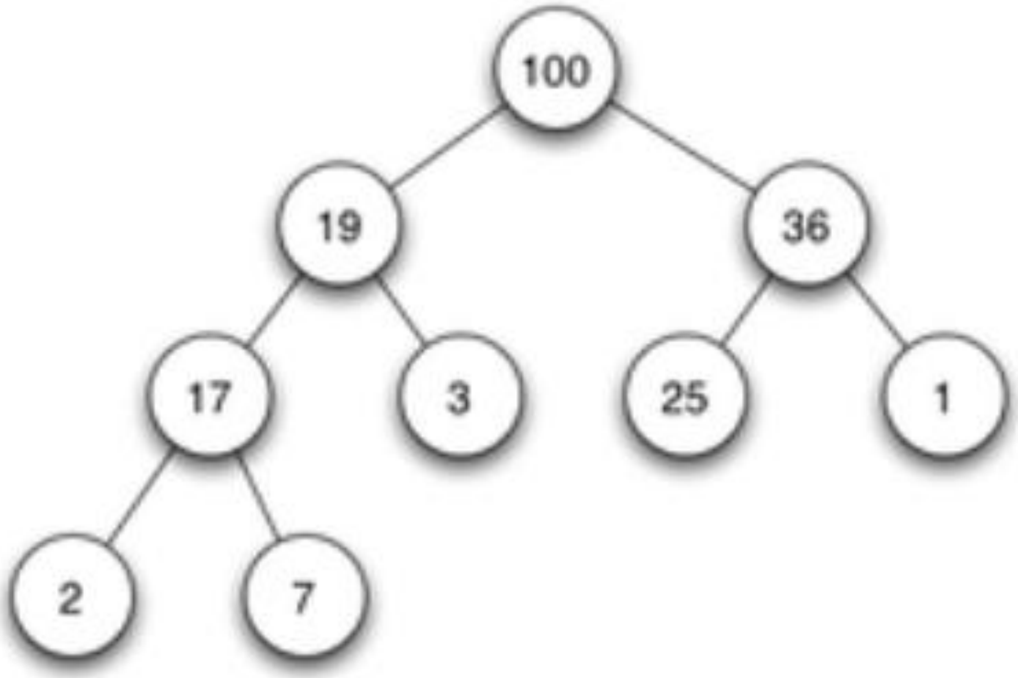
- "**Heap**" refers to being "**top of the heap**", i.e. what's on the top dominates what is underneath
    - greater than or less than (or equal to) everything under it

Binary Heap

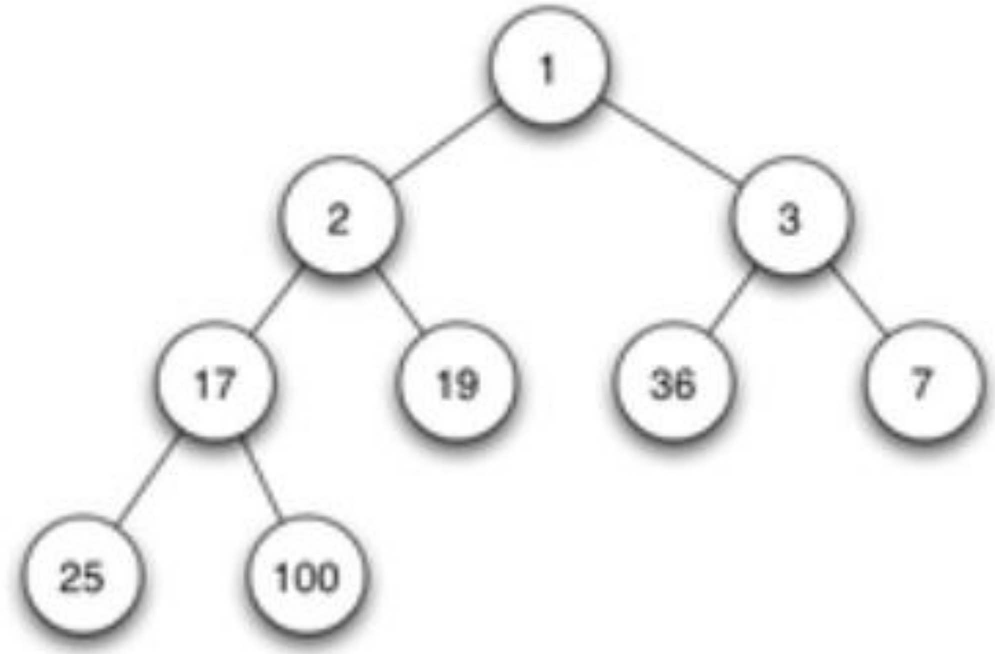
Keys in each node dominate the keys of its children

- **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 ($\mathbf{2^i}$ nodes at the **ith-level**)

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

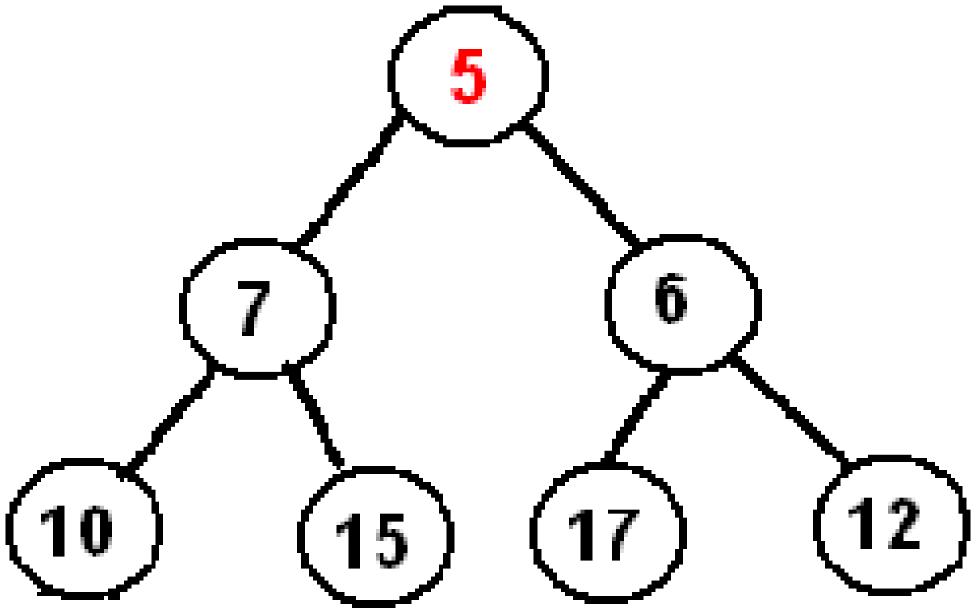
❖ Each node is "$>=$" or "$<=$" each of its children according to some comparison predicate which s fixed for the entire data structure

Binary Heap

- 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 $h = \lfloor \log n \rfloor$

Binary Heap

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

1. Add element to the bottom level

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

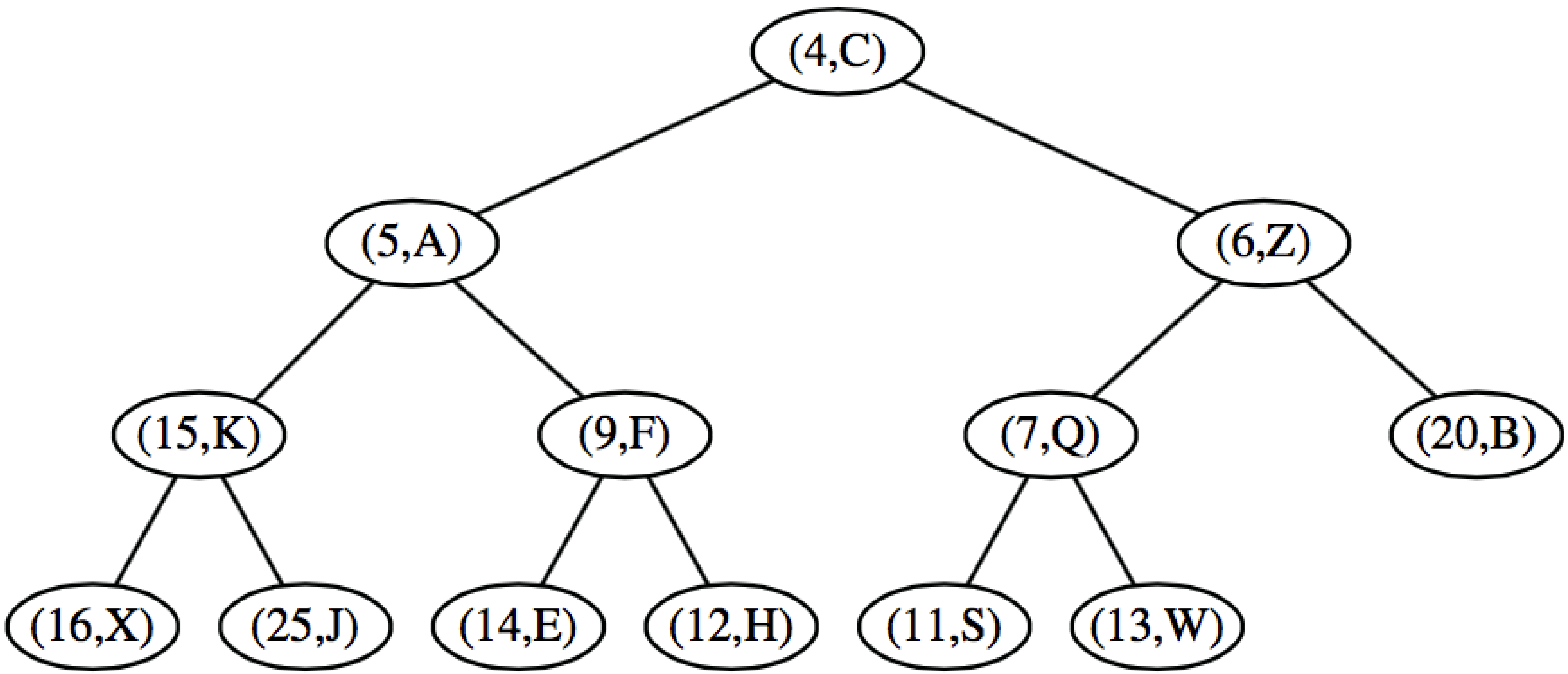
Binary Heap

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

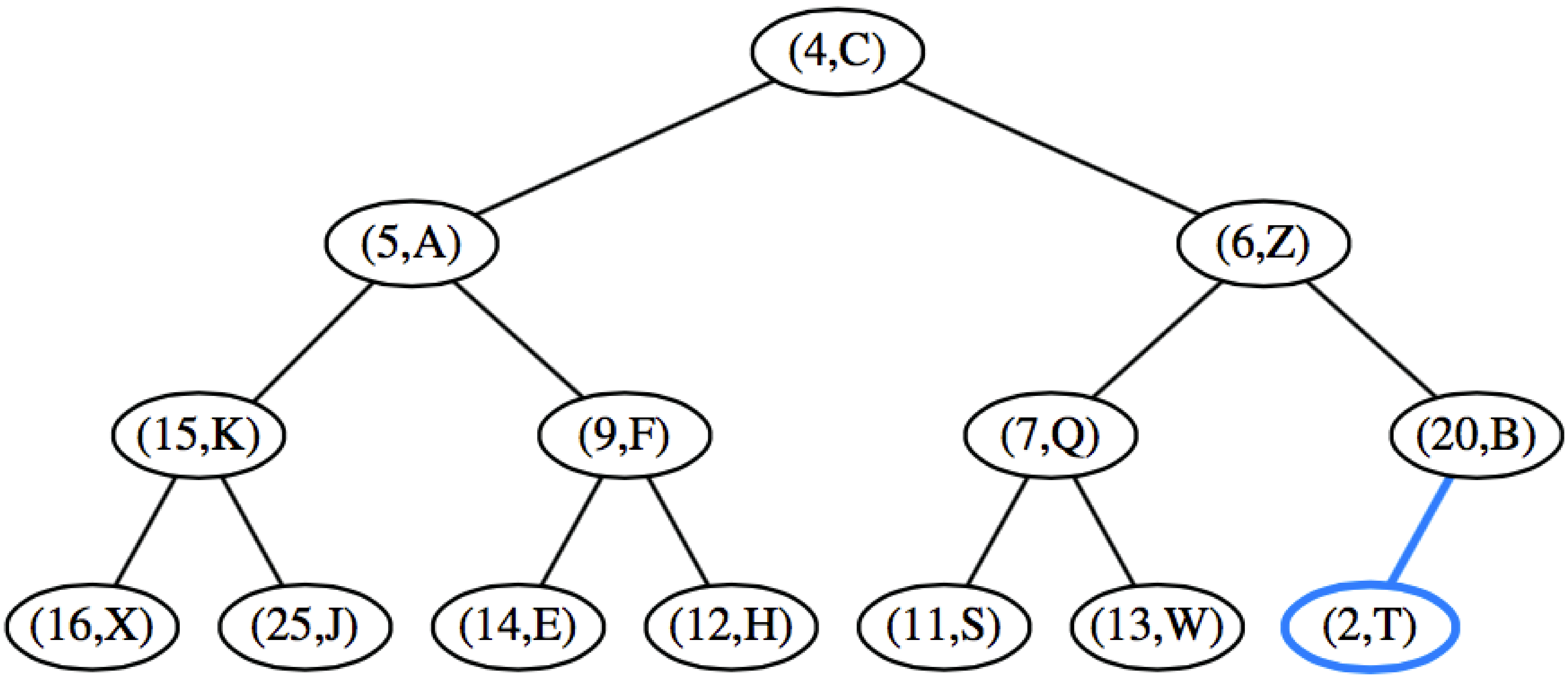
1. Add element to the bottom level
2. Compare the added element with its parent; if they are in correct order, stop
3. If not, swap the element with its parent and return to previous step

Binary Heap

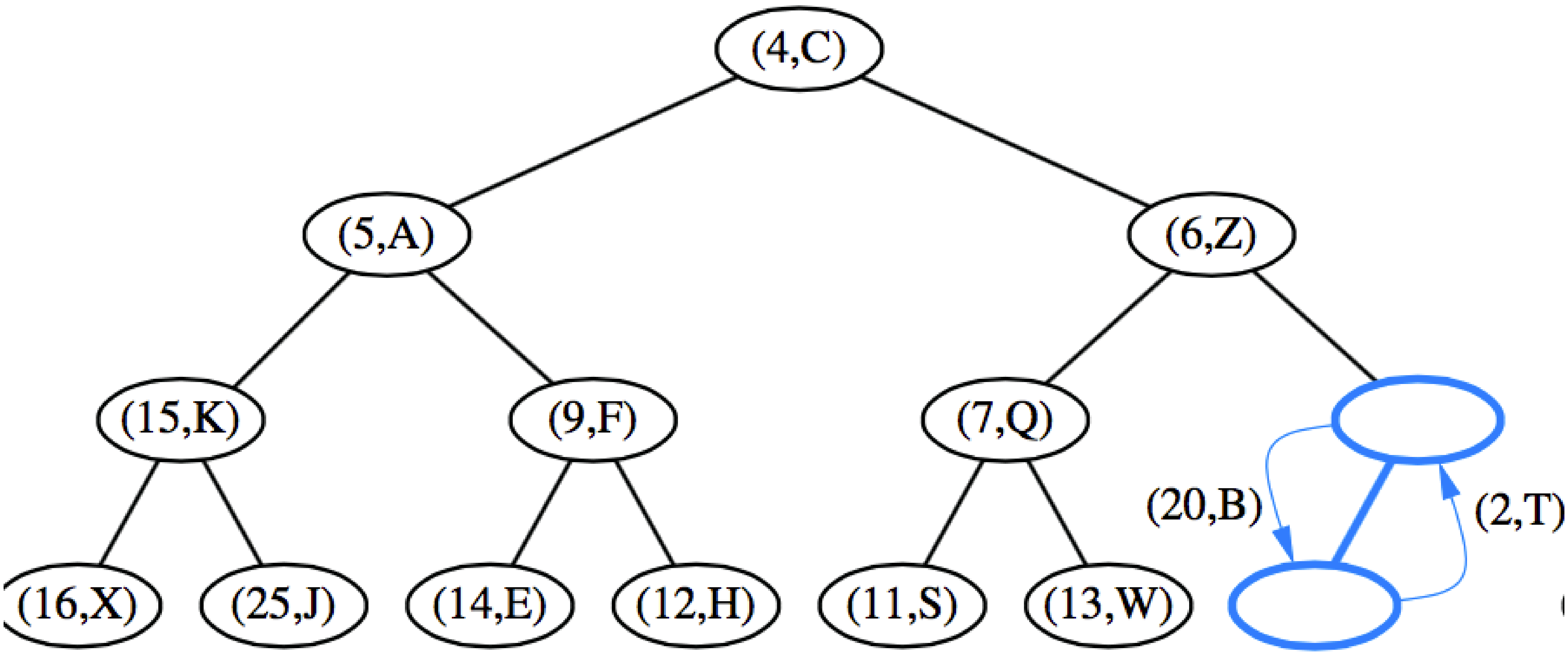
- Insert an item T with key 2 into the following heap



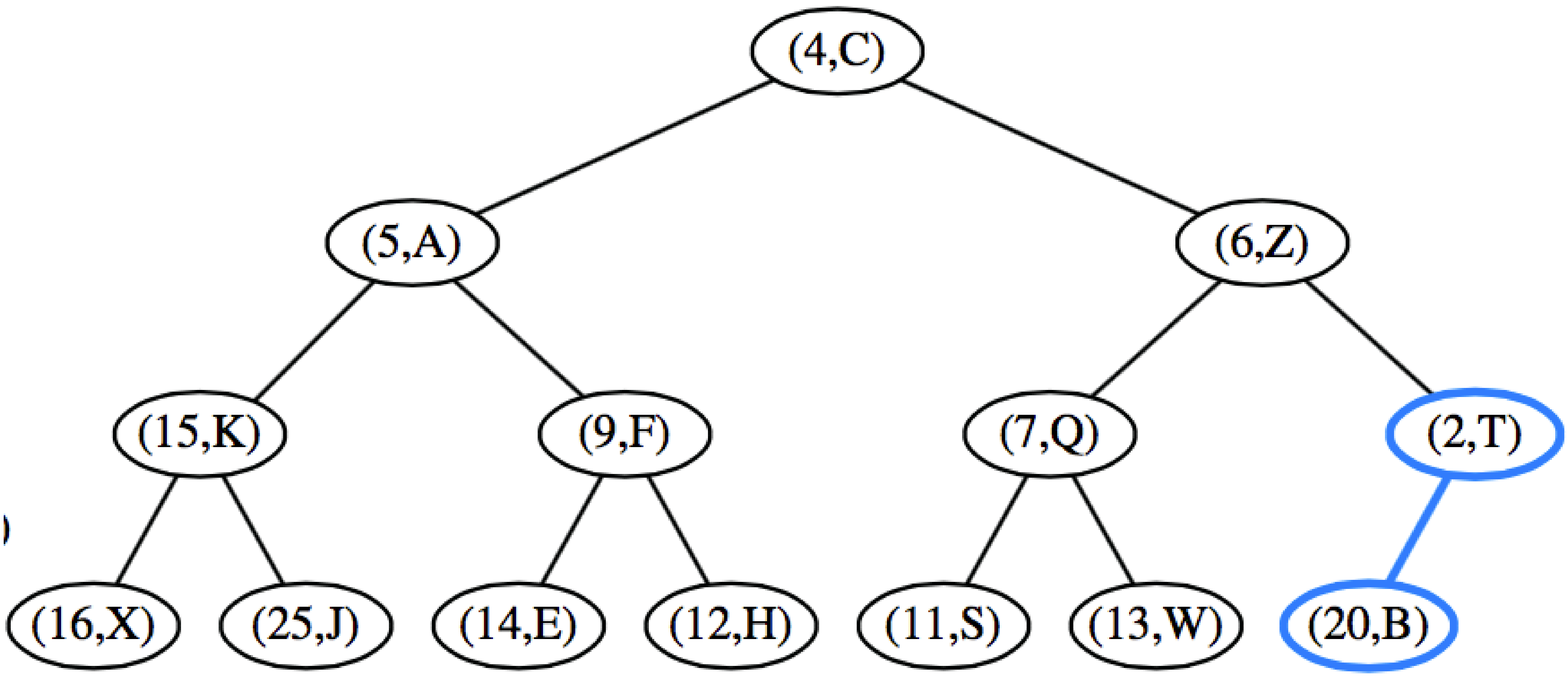
Binary Heap



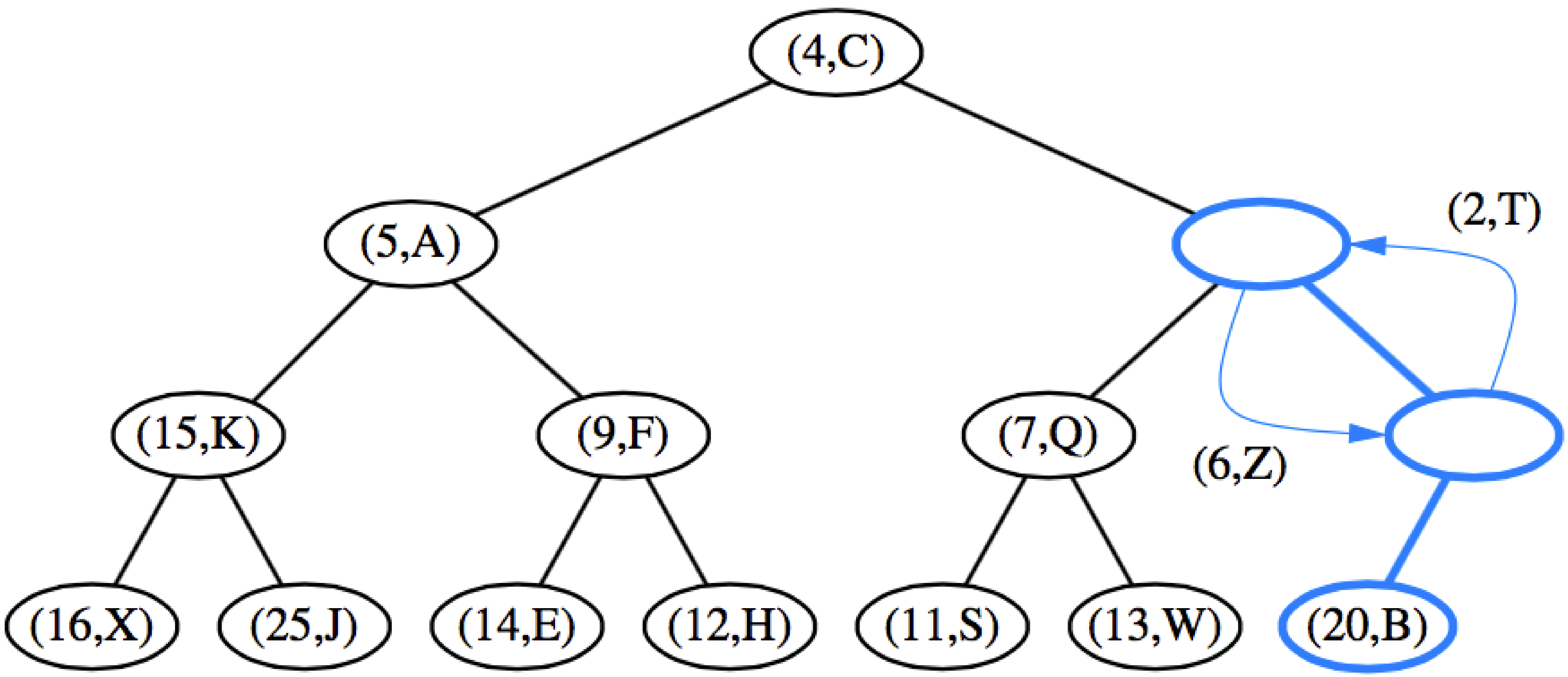
Binary Heap



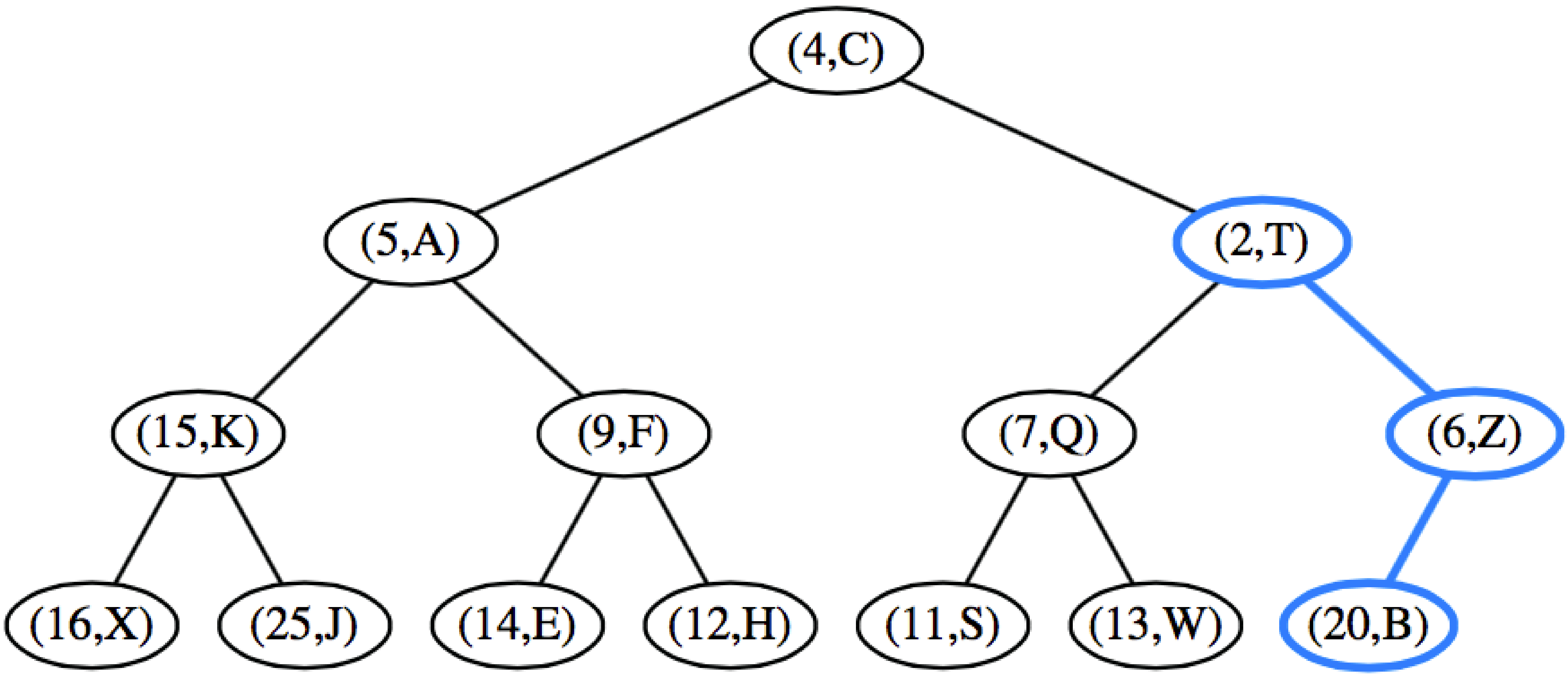
Binary Heap



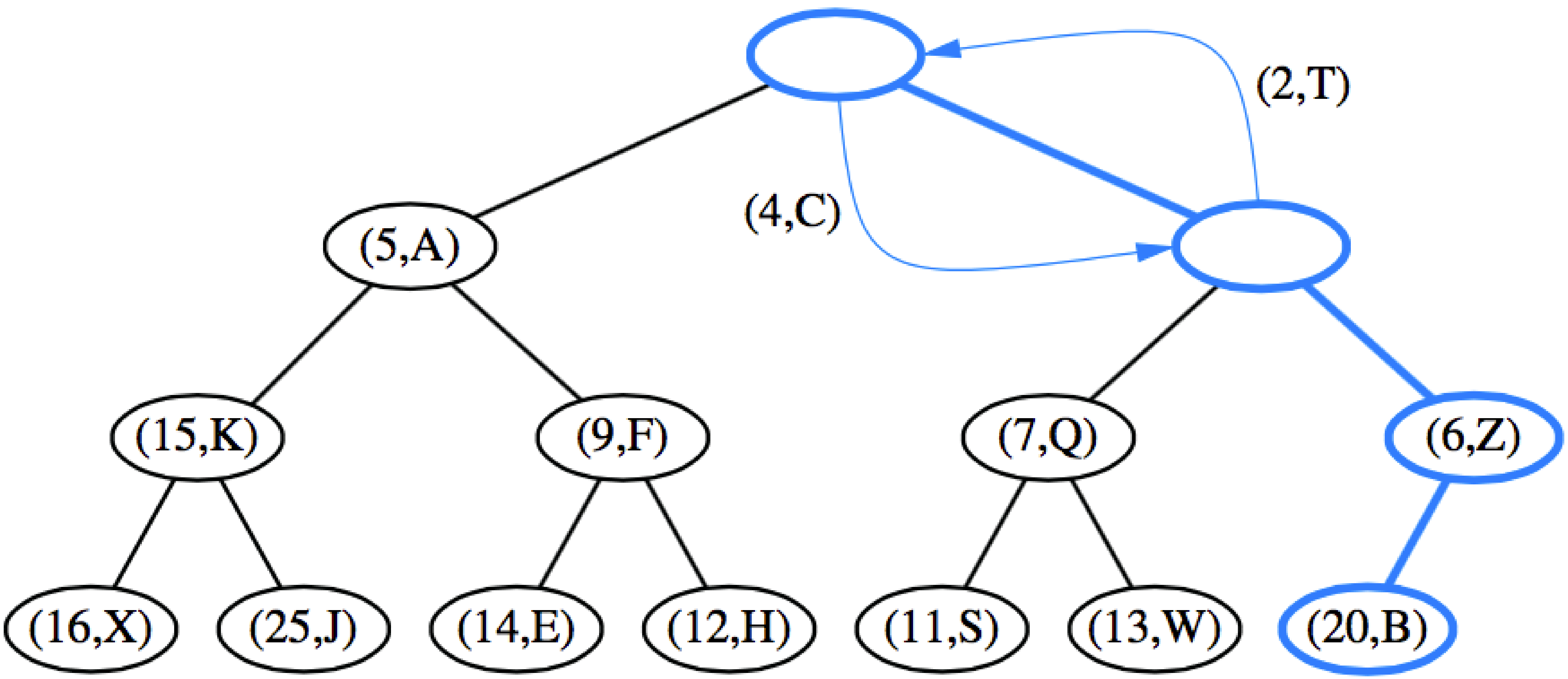
Binary Heap



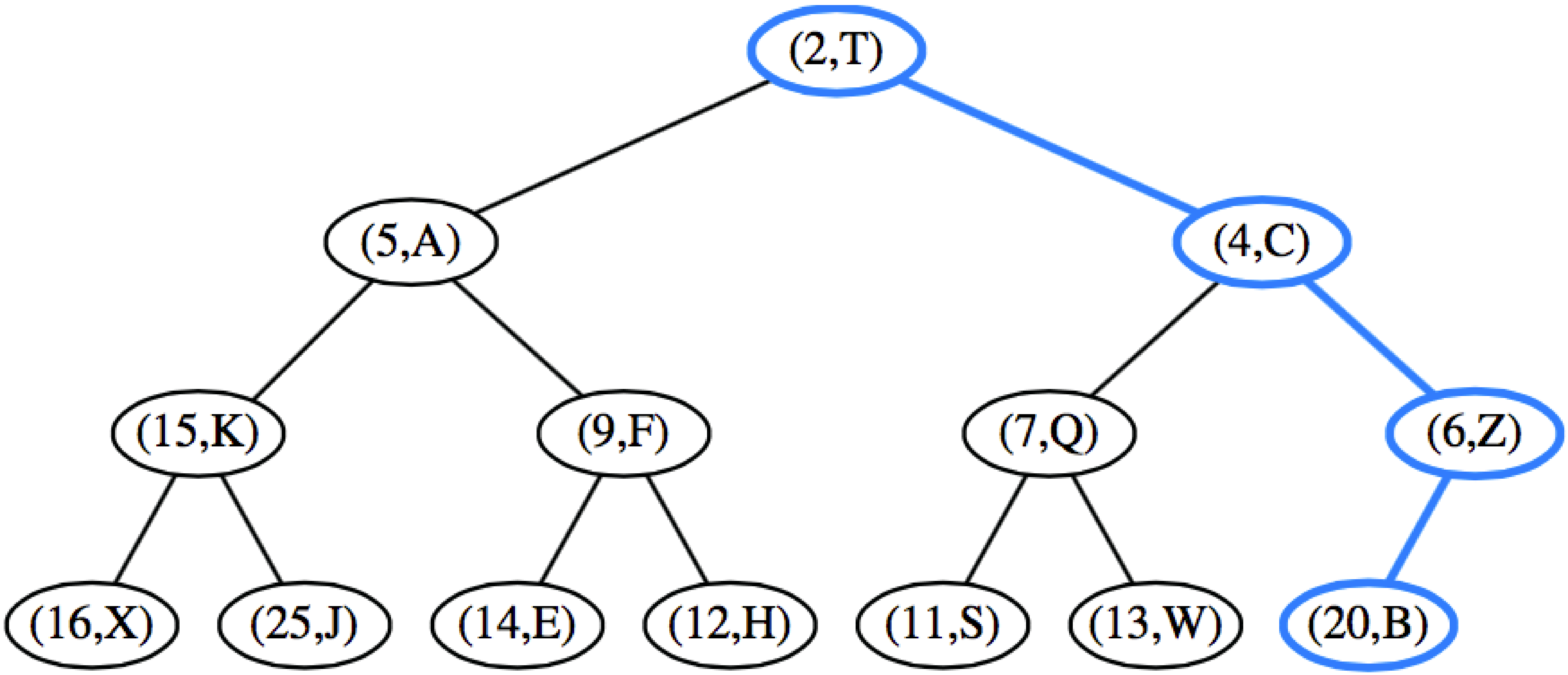
Binary Heap



Binary Heap



Binary Heap



Binary Heap

- **Removal**

- 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)**

  1. Replace root with the last element on the bottom level

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)

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

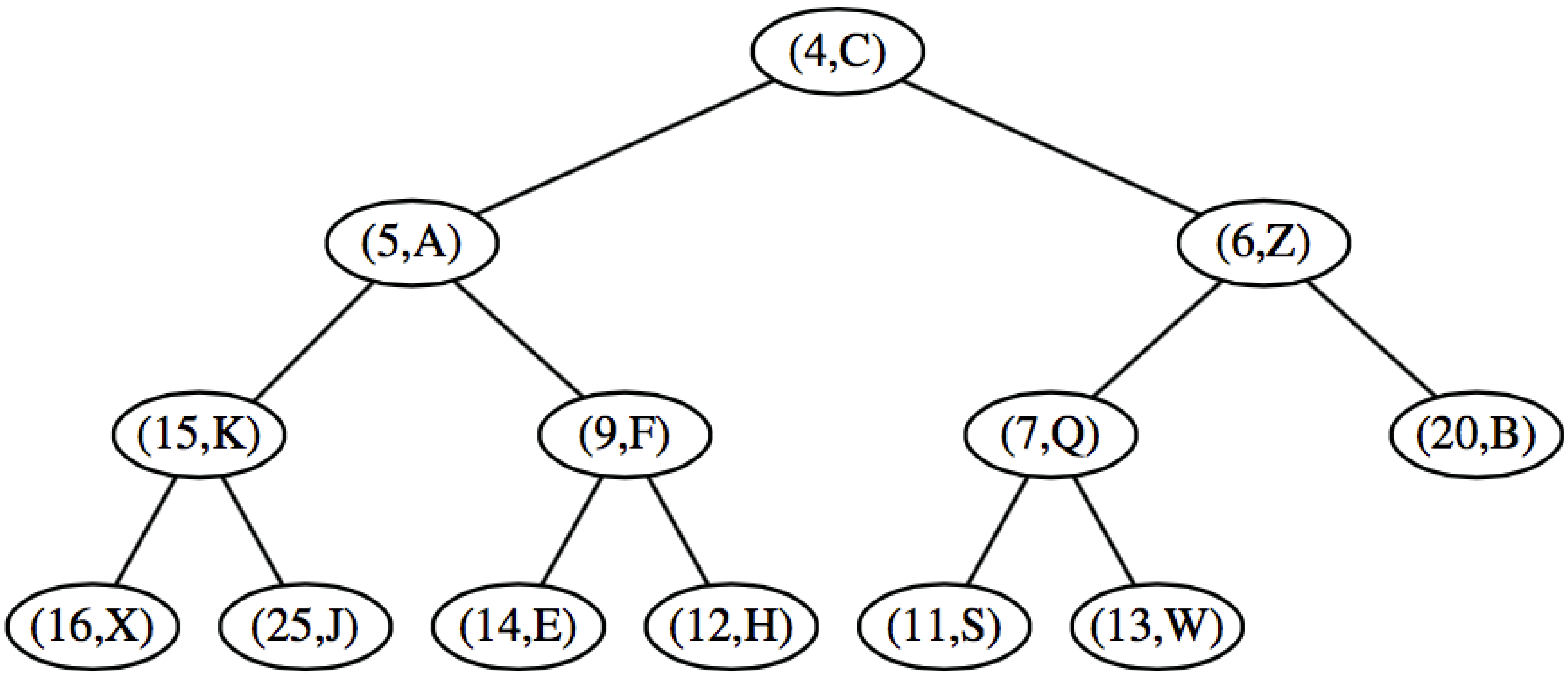
Binary Heap

- 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

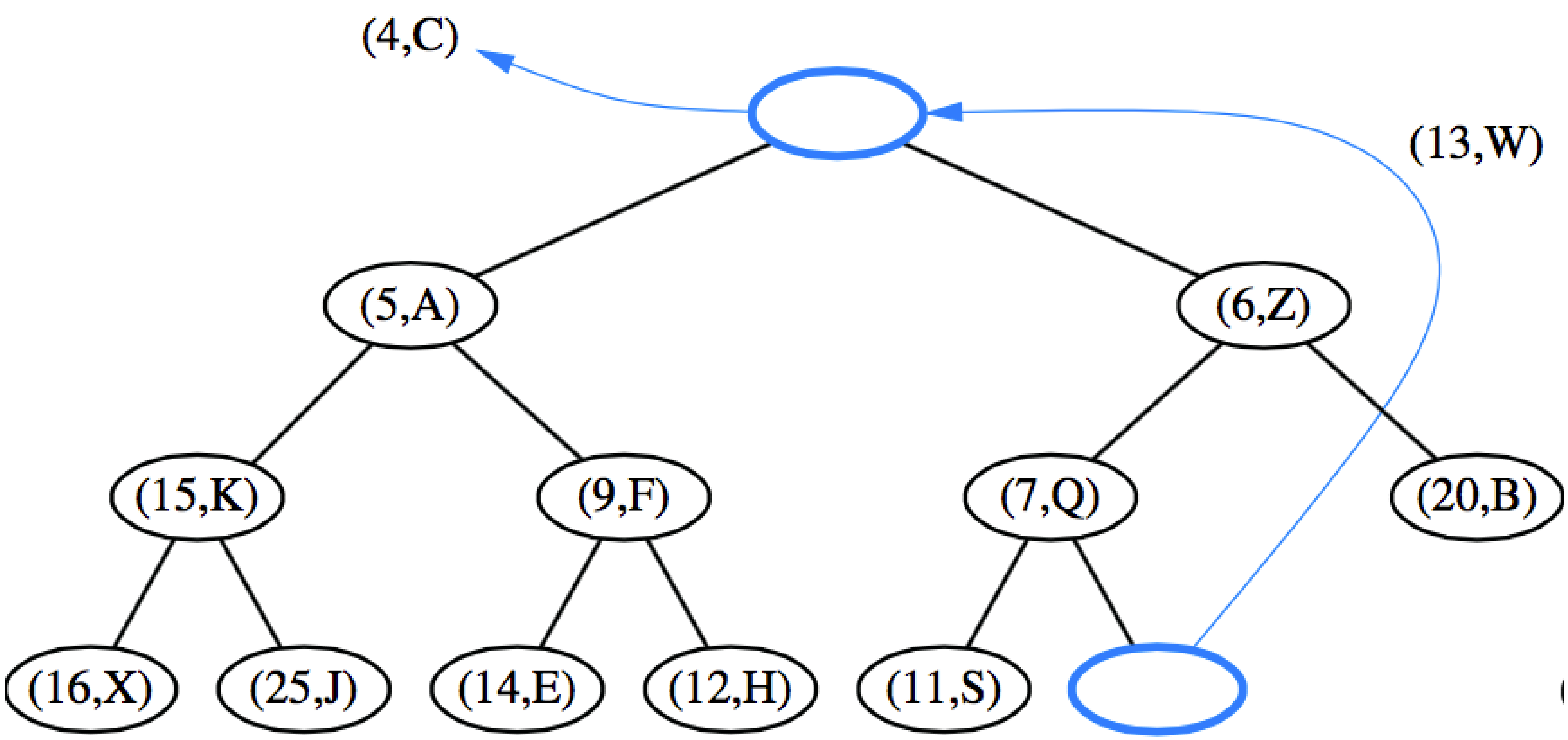
Binary Heap

- Deletion in a binary heap

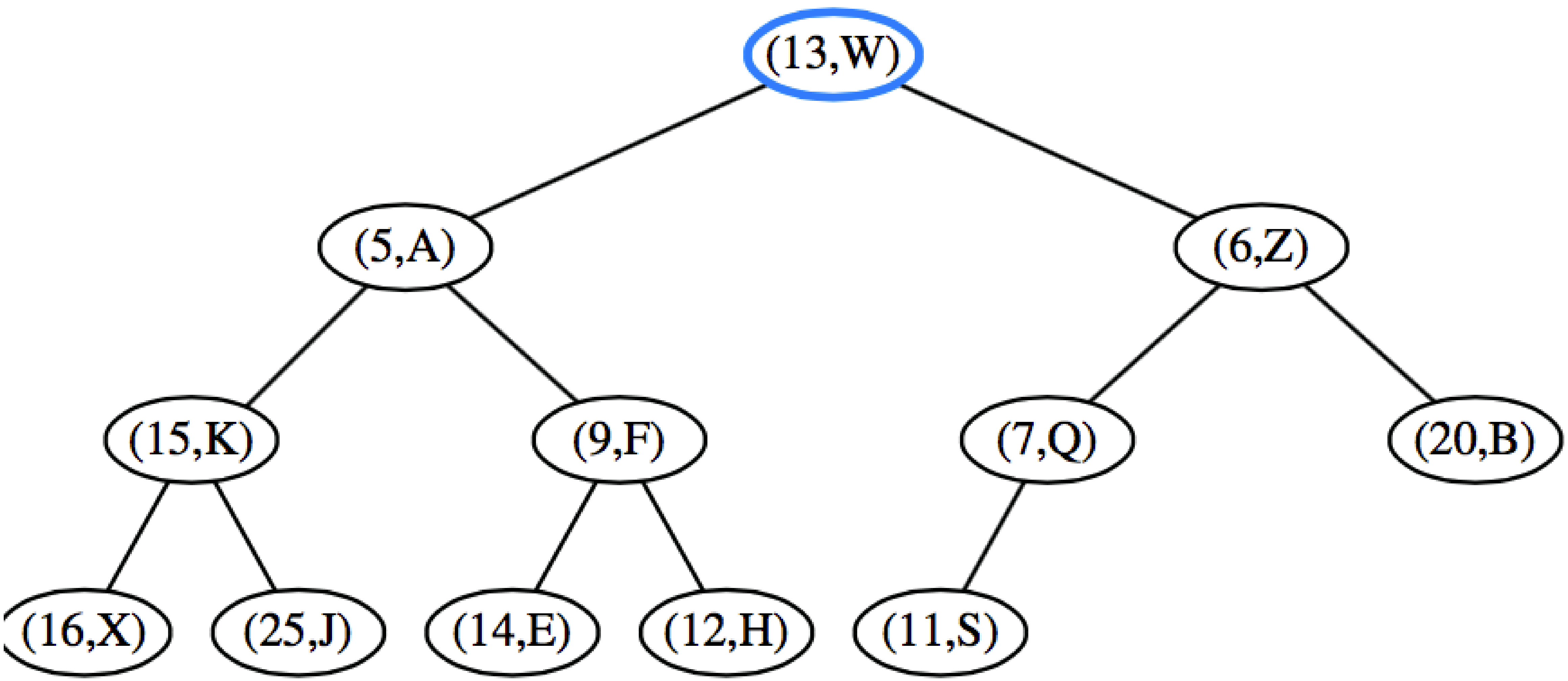


Binary Heap

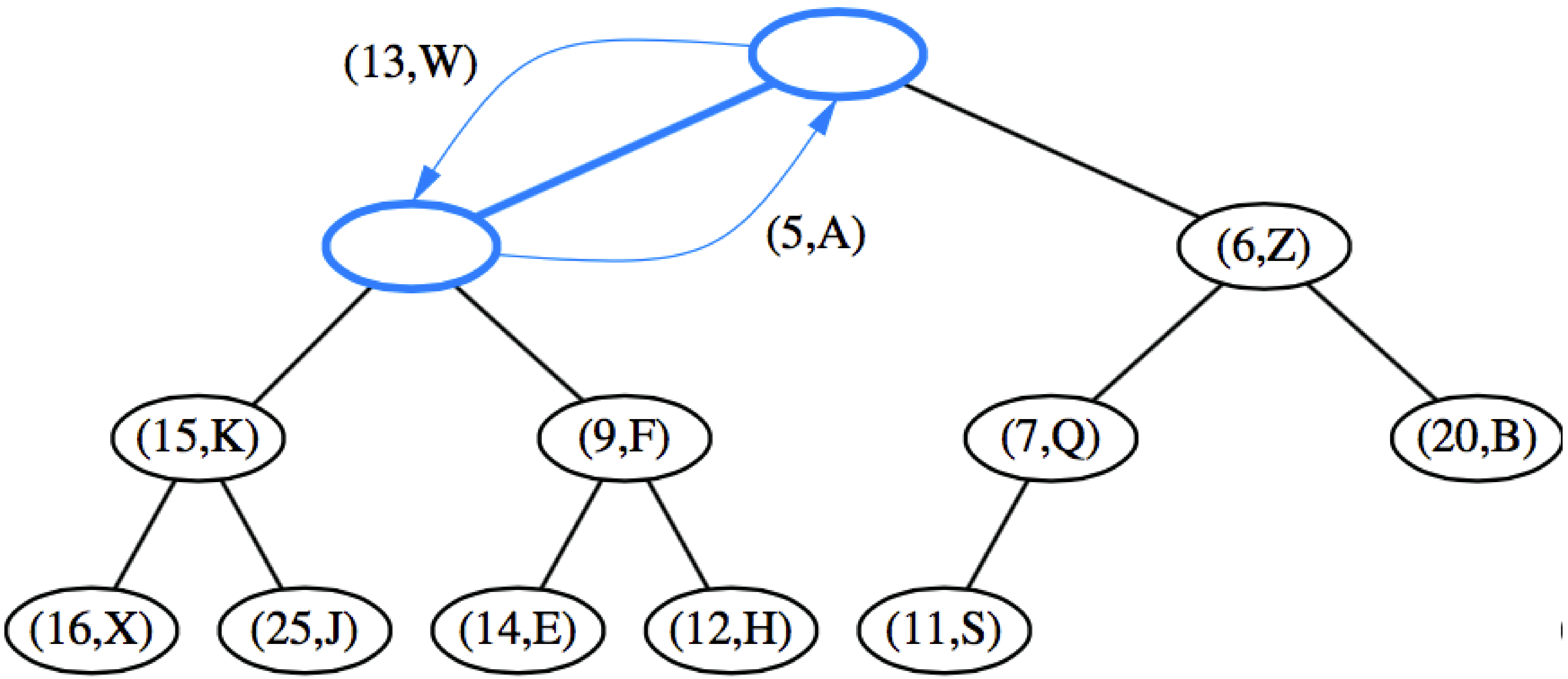
- Deletion in a binary heap



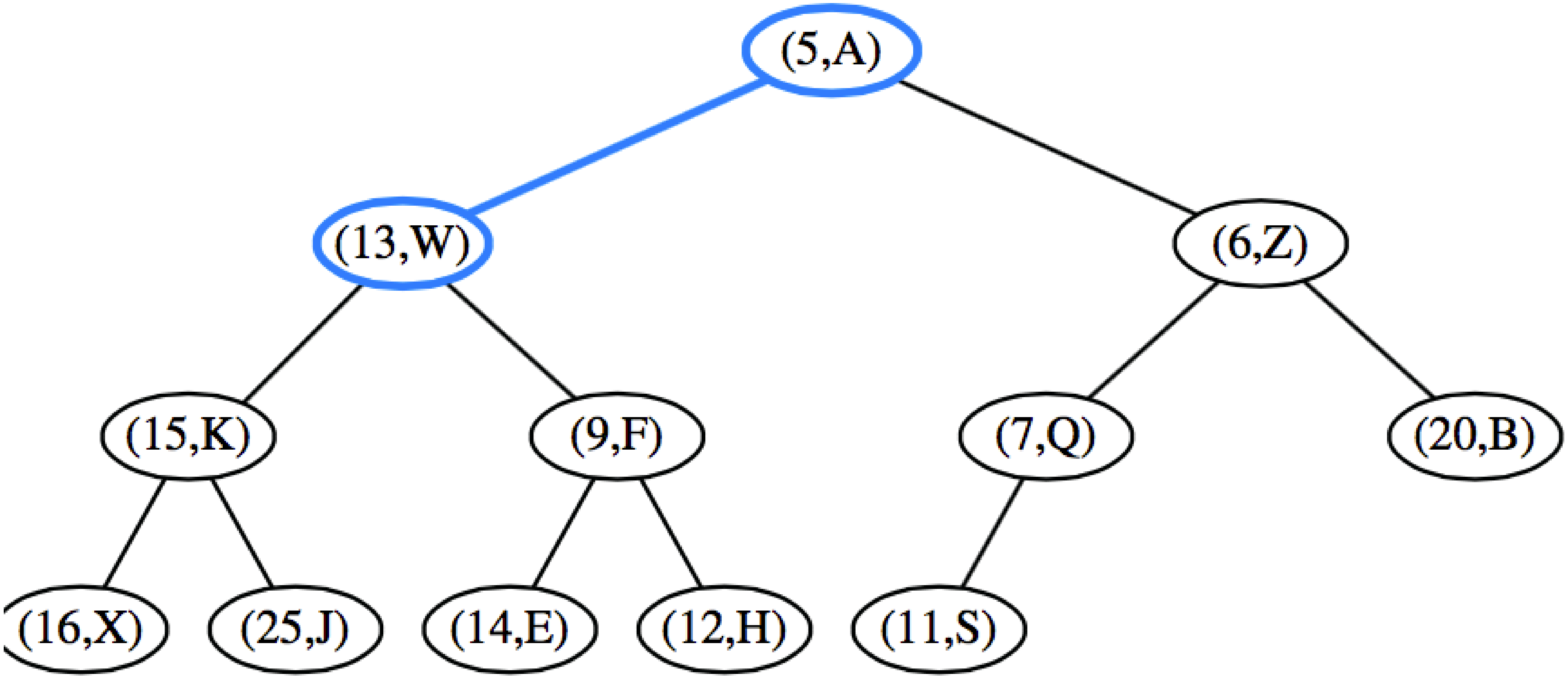
Binary Heap



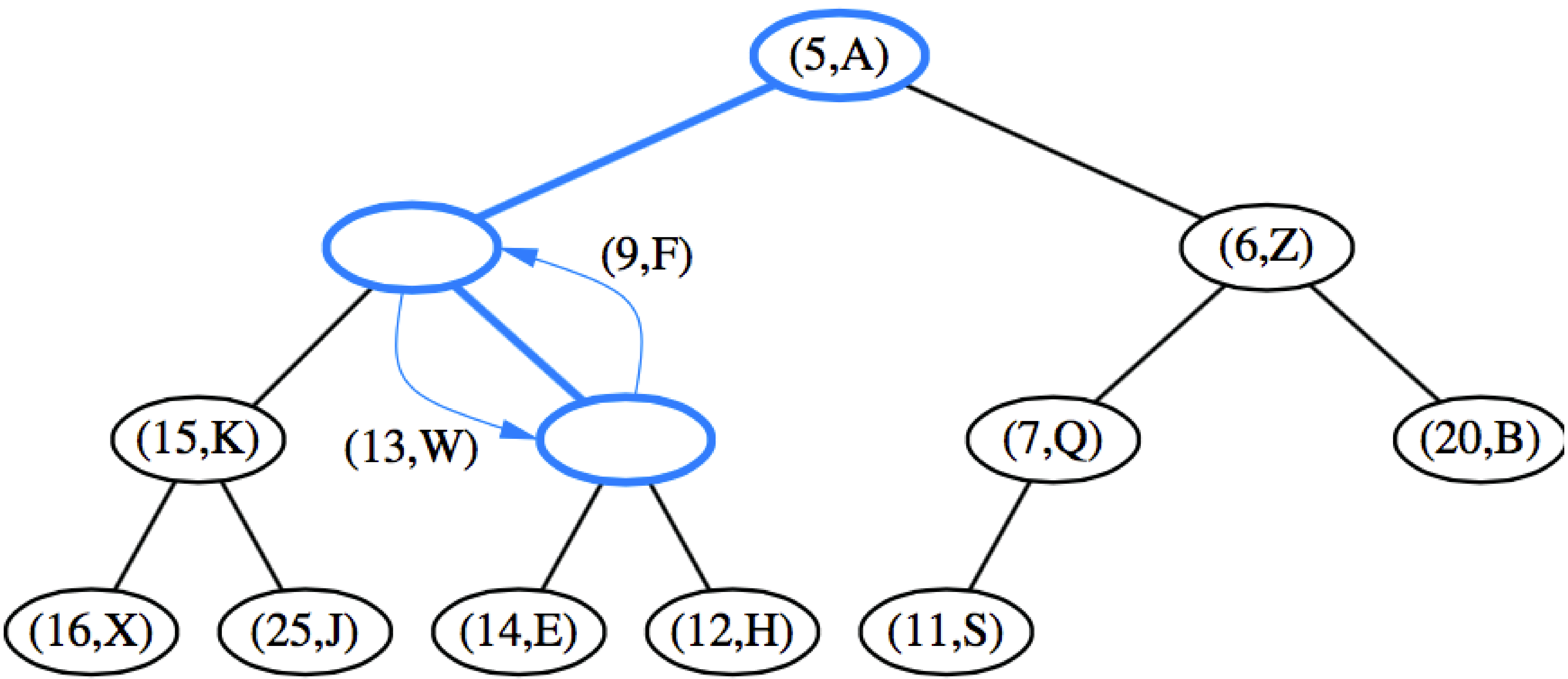
Binary Heap



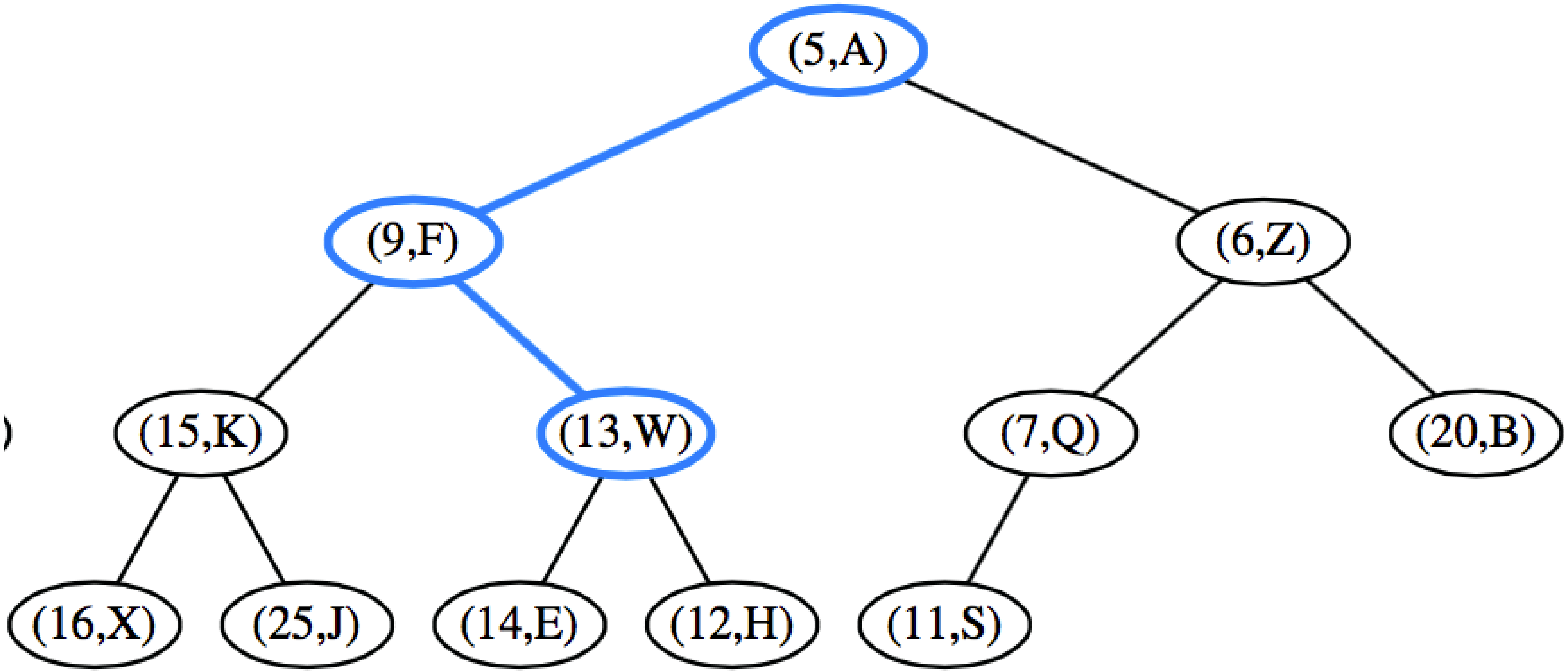
Binary Heap



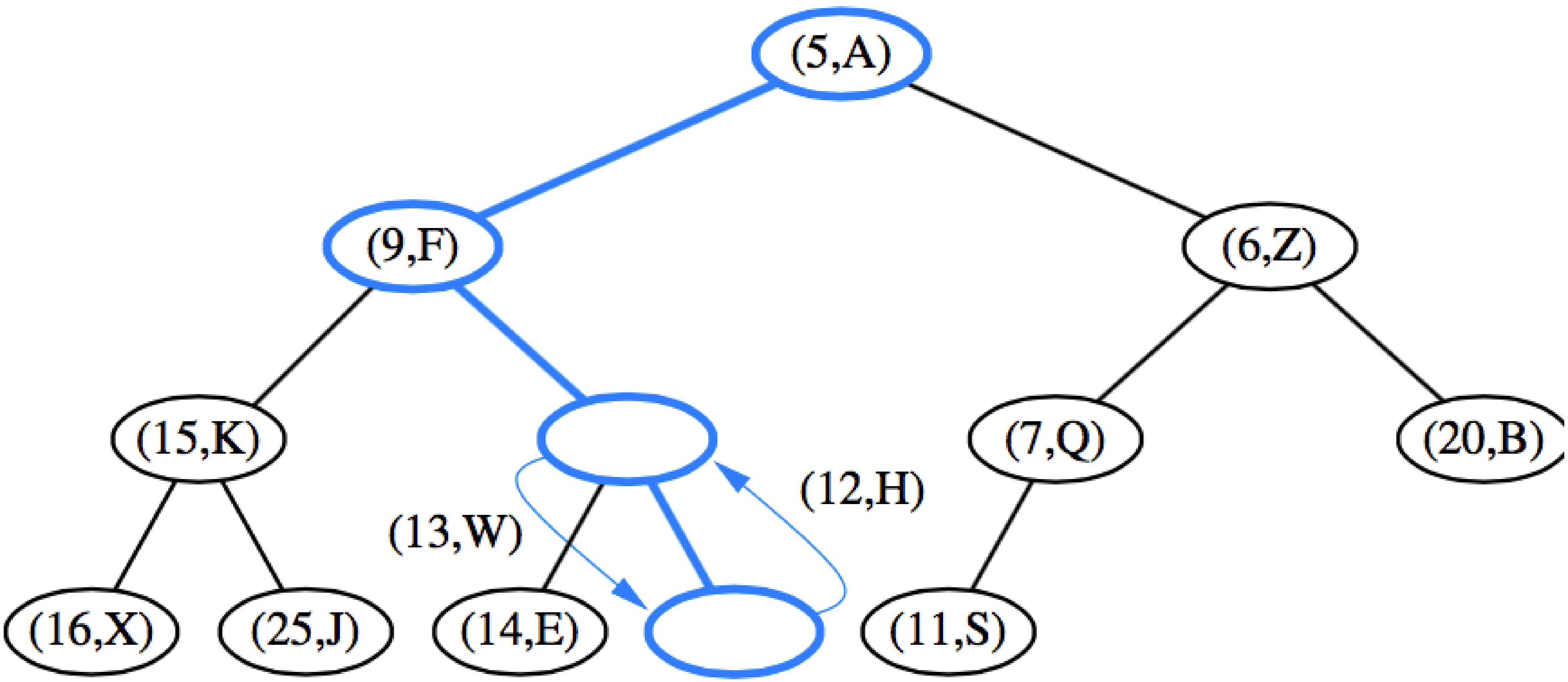
Binary Heap



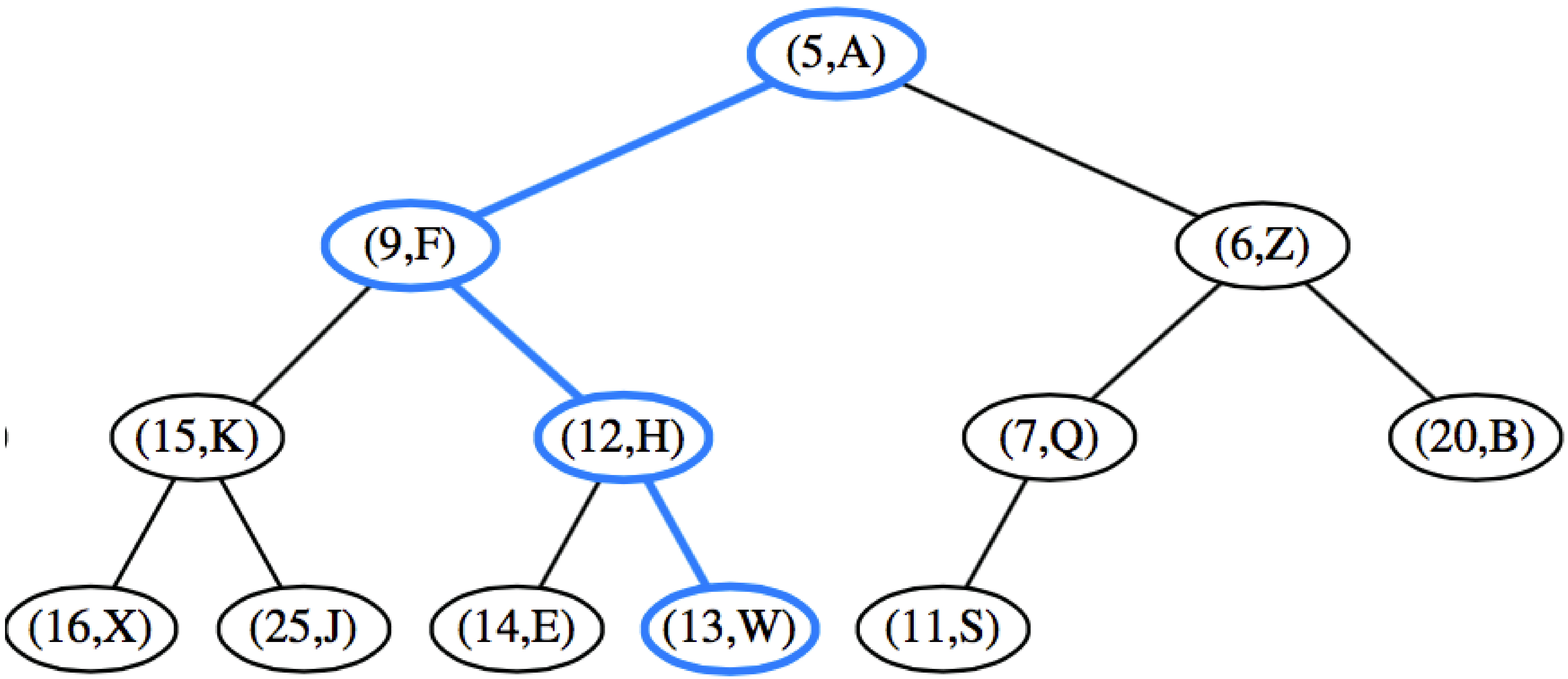
Binary Heap



Binary Heap



Binary Heap



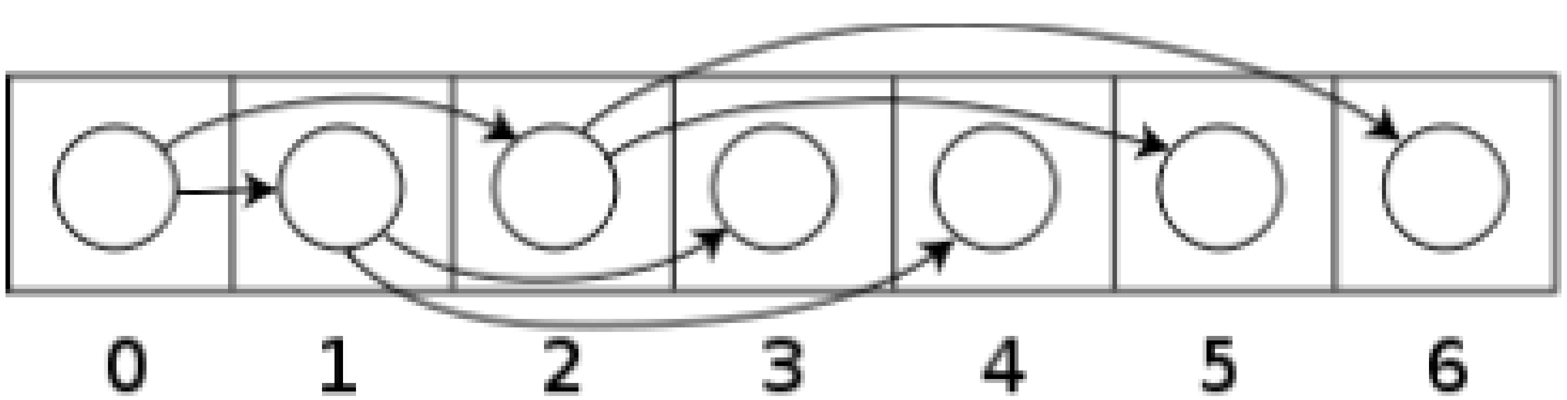
Binary Heap

- **Implementation as an array**

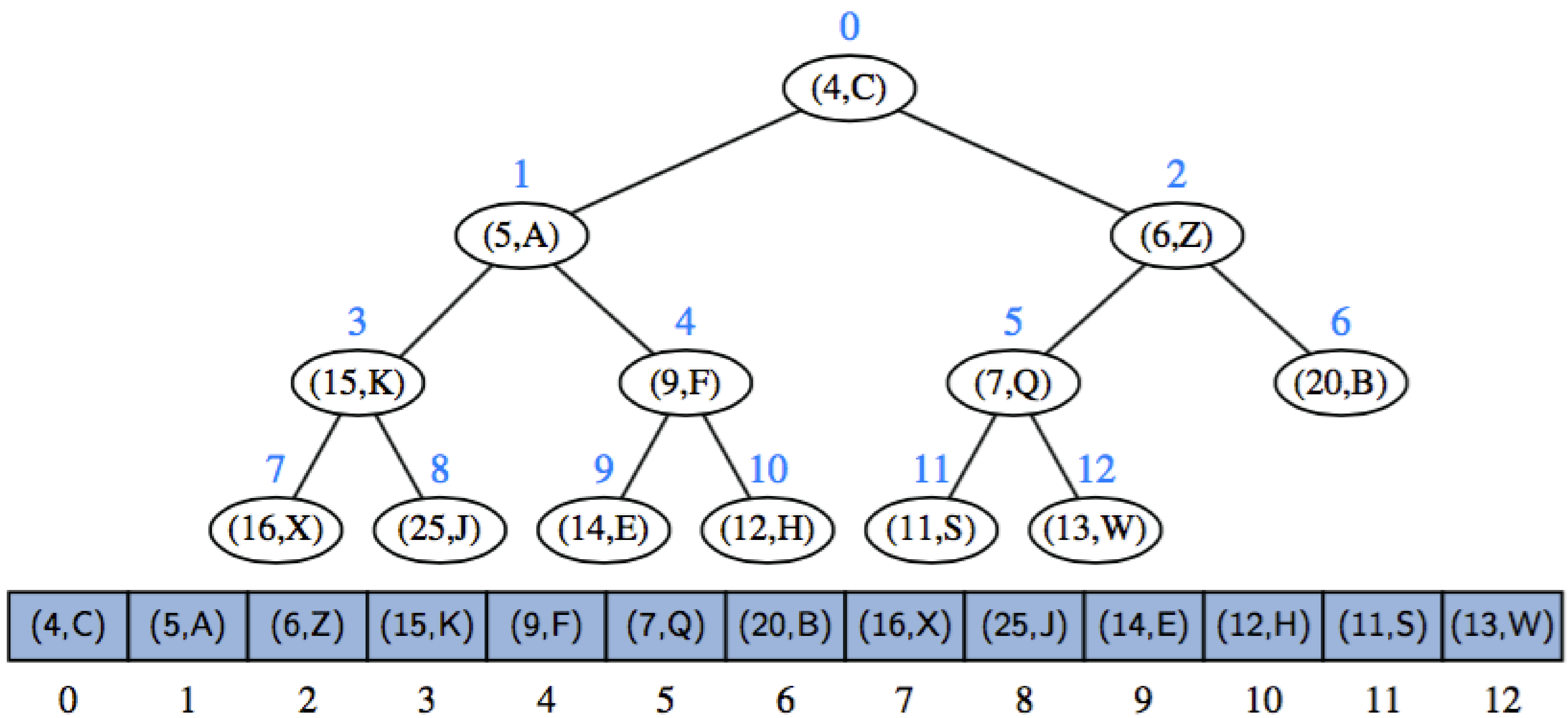
Represent a binary tree without any pointers by using an array of keys and a mapping function

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



Goodrich's Book!

Different Books, Different Representation

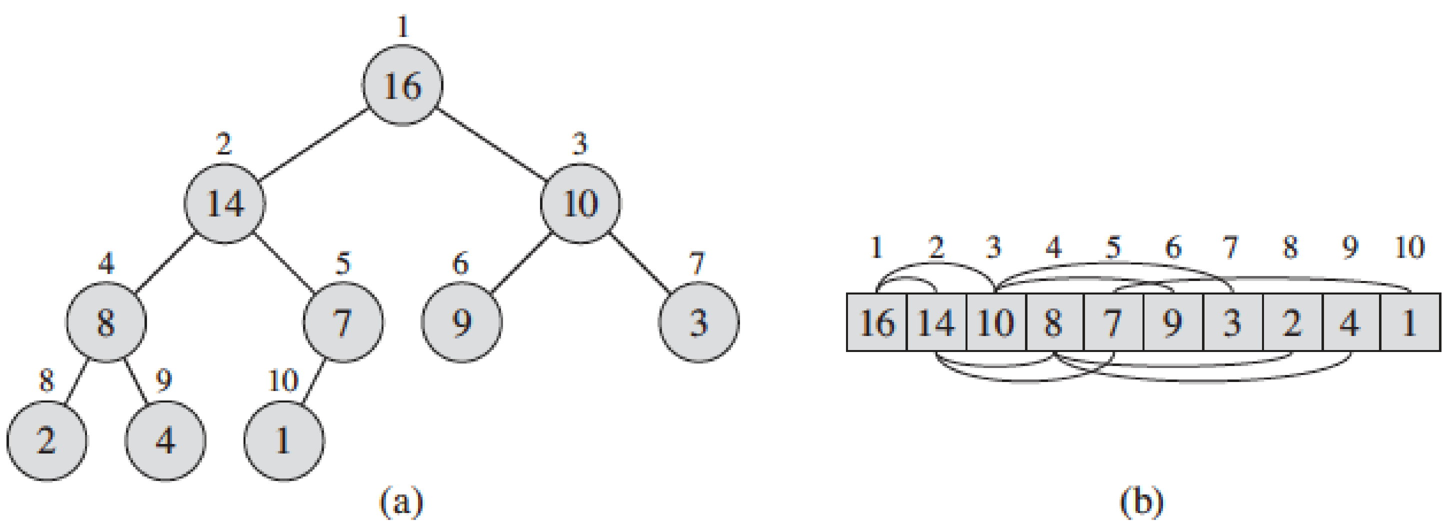


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Cormen's Book!

Binary Heap

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

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

```
// Now perform downheap to restc... heap order
i = 0
childIndex = findSmallerChild(i)
while (childIndex != 0 && H[childI...
    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

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
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

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*** + ***B***

```
S ← 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
```

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

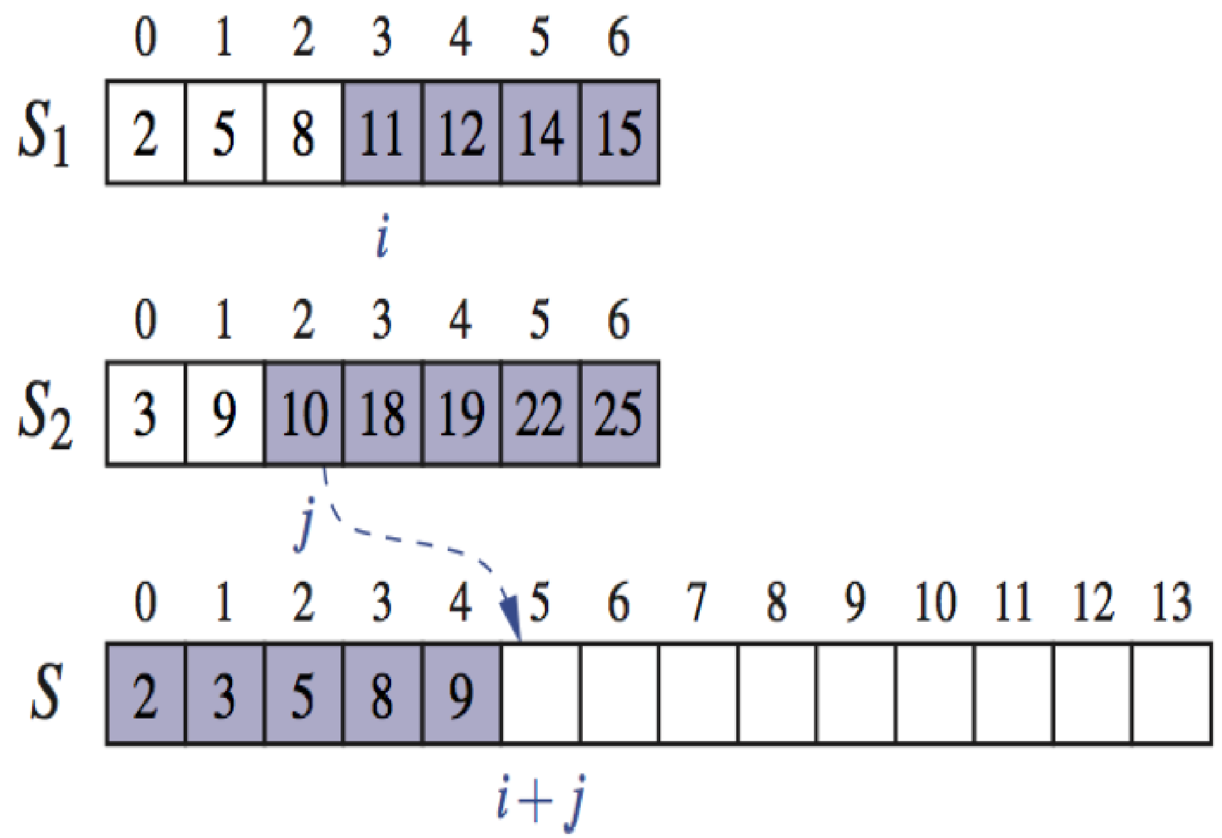
```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A + B

      S ← 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
```

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

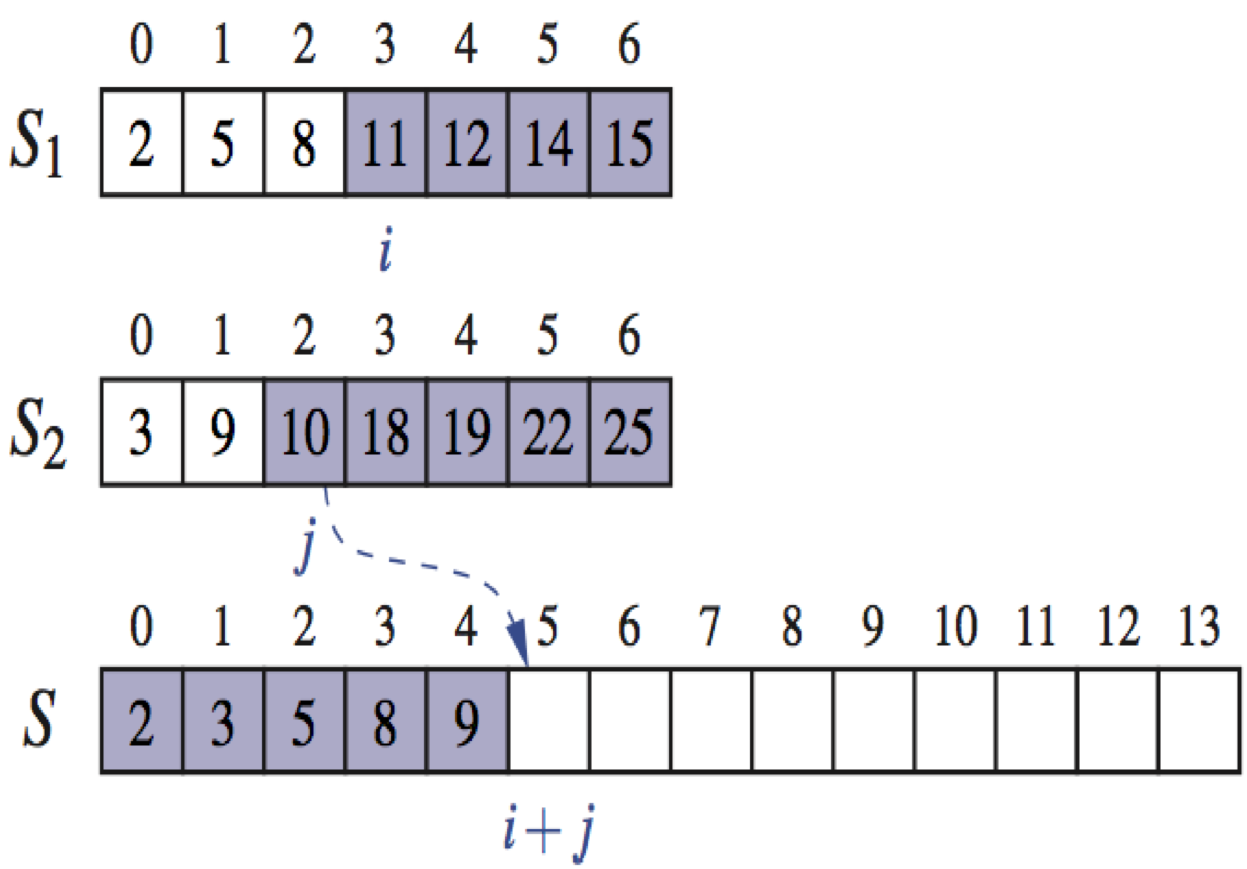
```
Algorithm merge(A, B)
   Input sequences A and B with
      n/2 elements each
   Output sorted sequence of A + B

    S ← 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

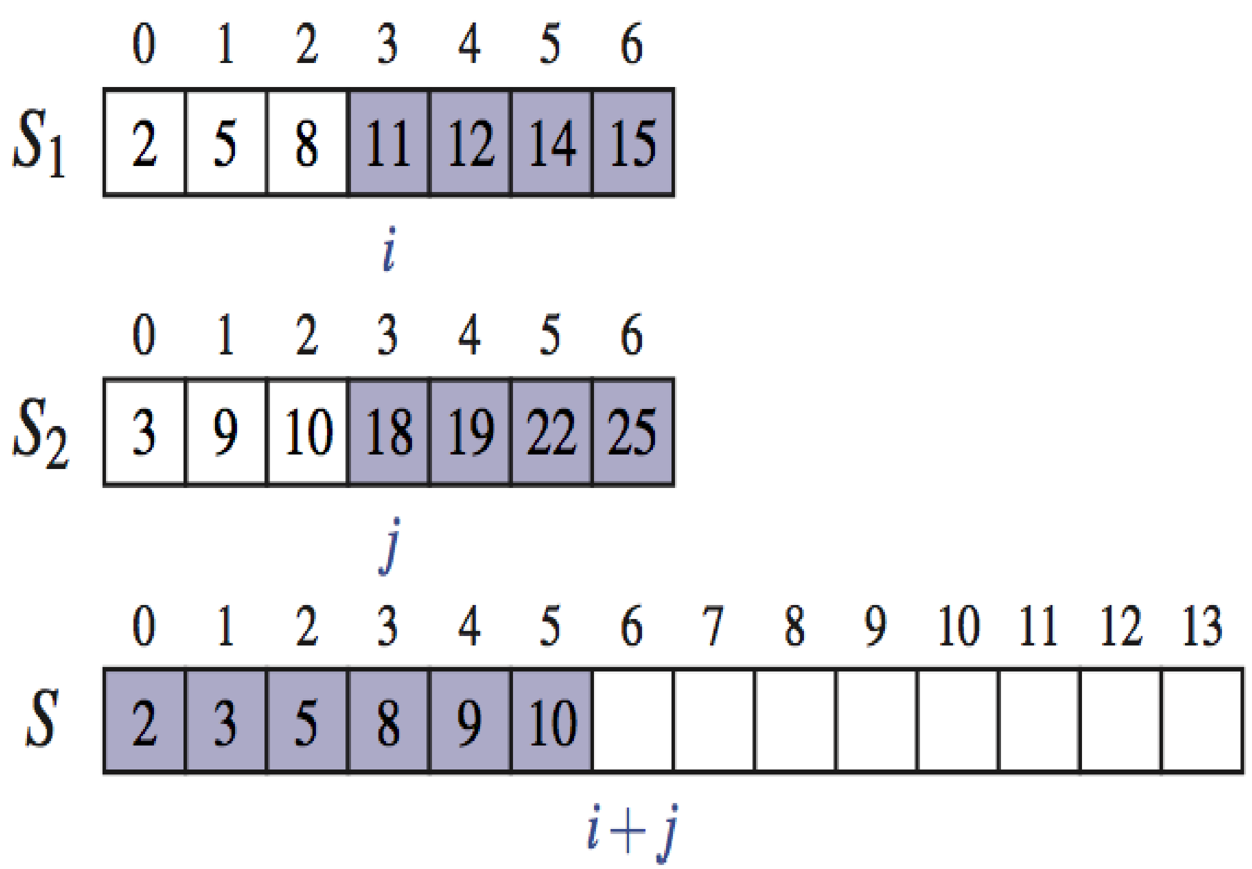


0 1 2 3 4 5 6
S1
2 5 8 11 12 14 15
i
0 1 2 3 4 5 6
S2
3 9 10 18 19 22 25
j
0 1 2 3 4 5 6 7 8 9 10 11 12 13
S
2 3 5 8 9
i+j

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
$i+j$

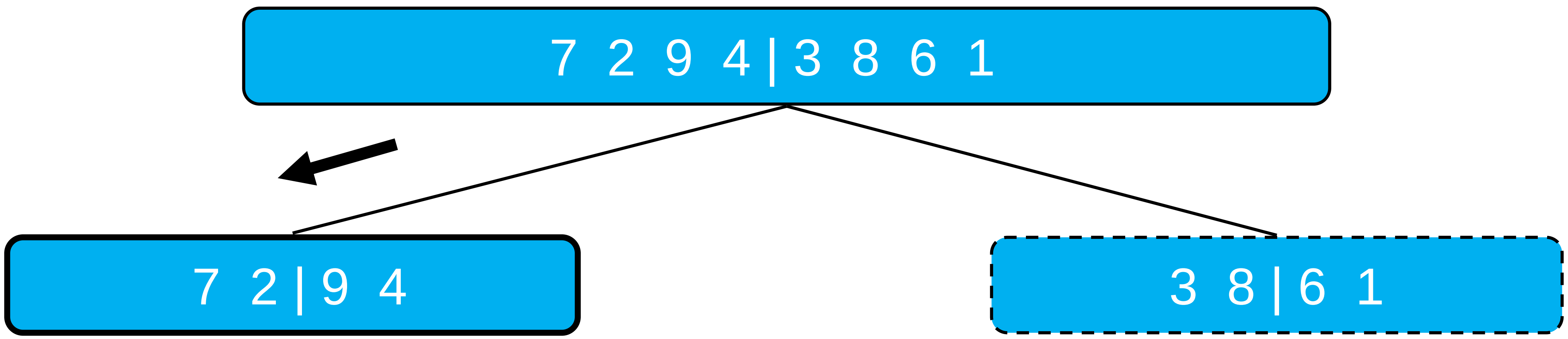
Execution Example

- Partition

7 2 9 4 | 3 8 6 1

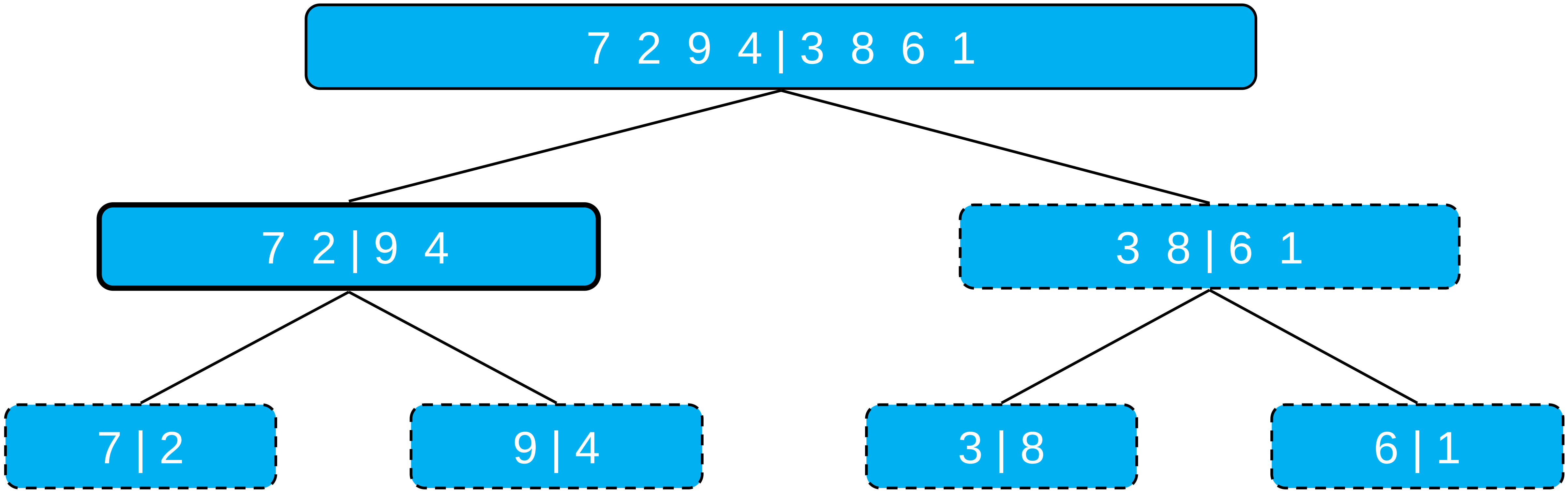
Execution Example (cont.)

- Recursive call, partition



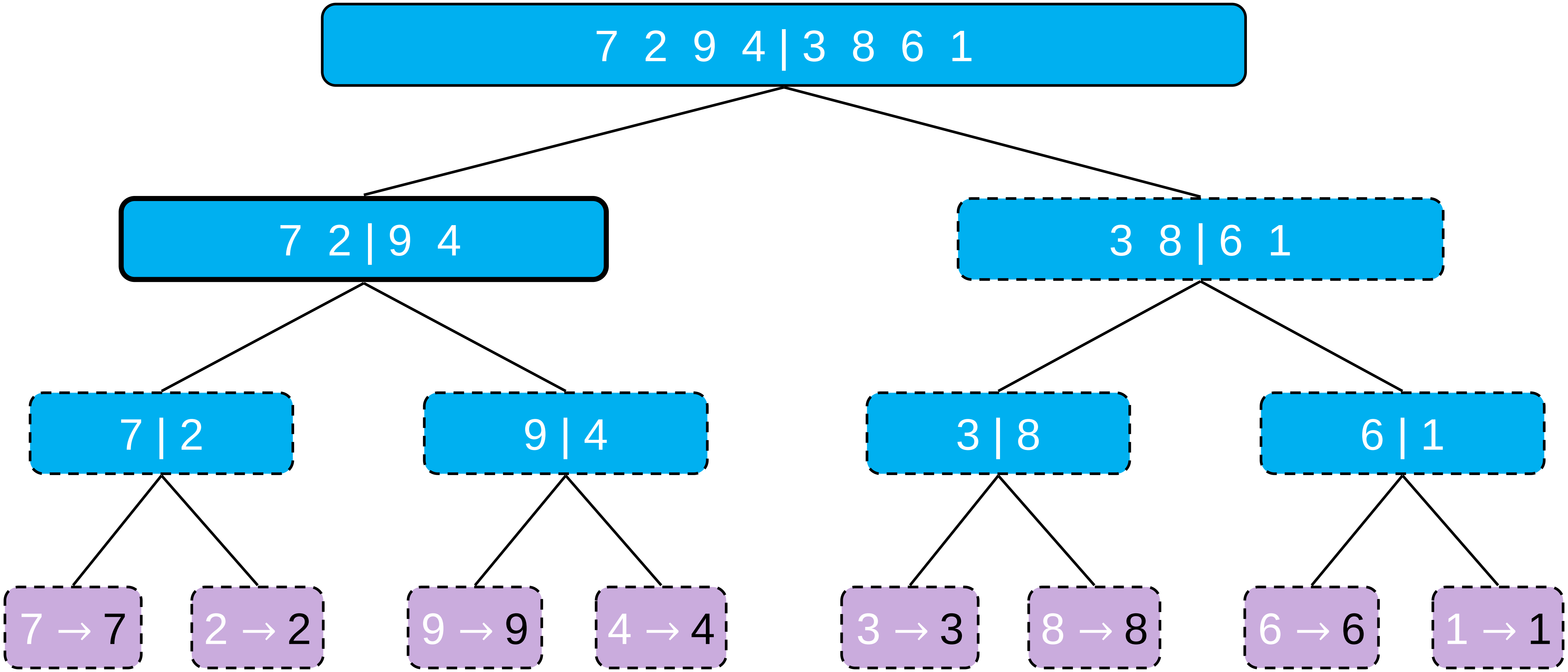
Execution Example (cont.)

- Recursive call, partition



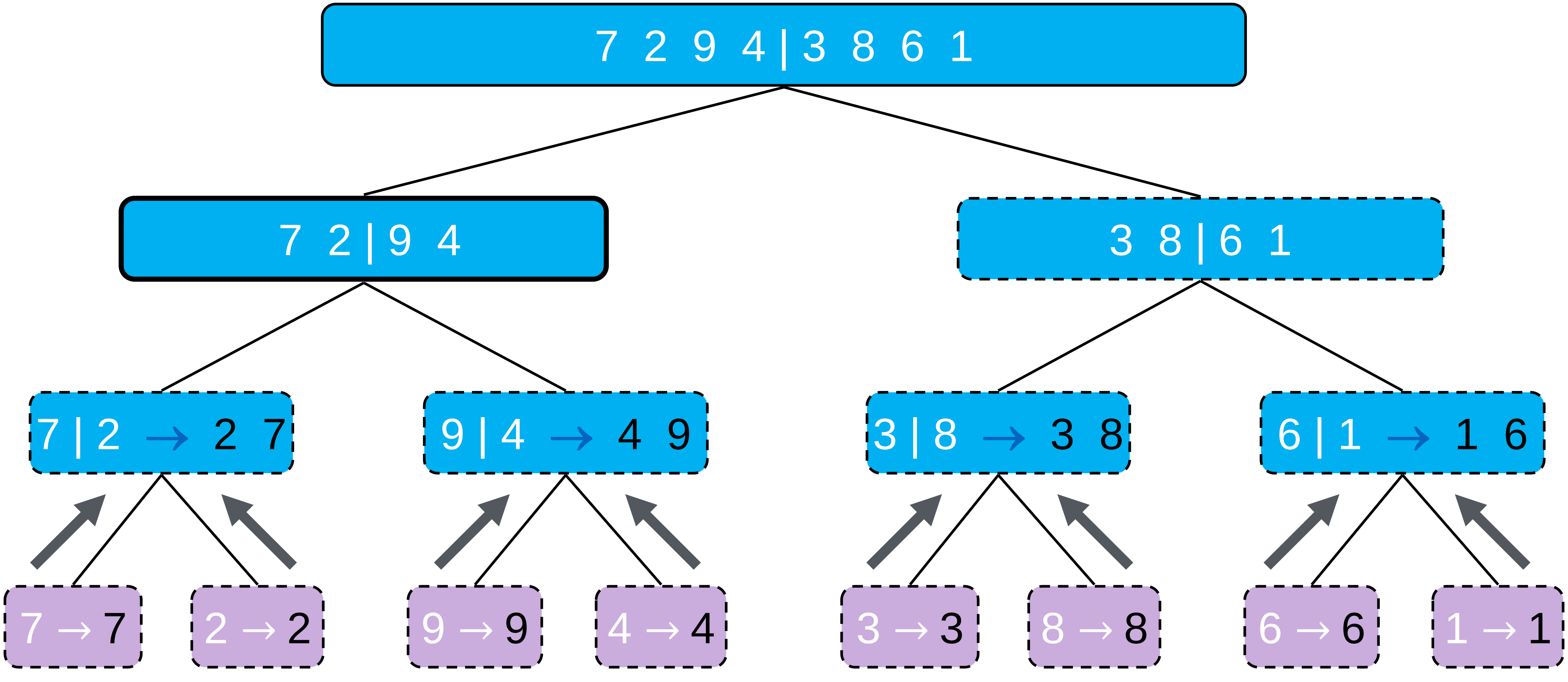
Execution Example (cont.)

- Recursive call, Base Case



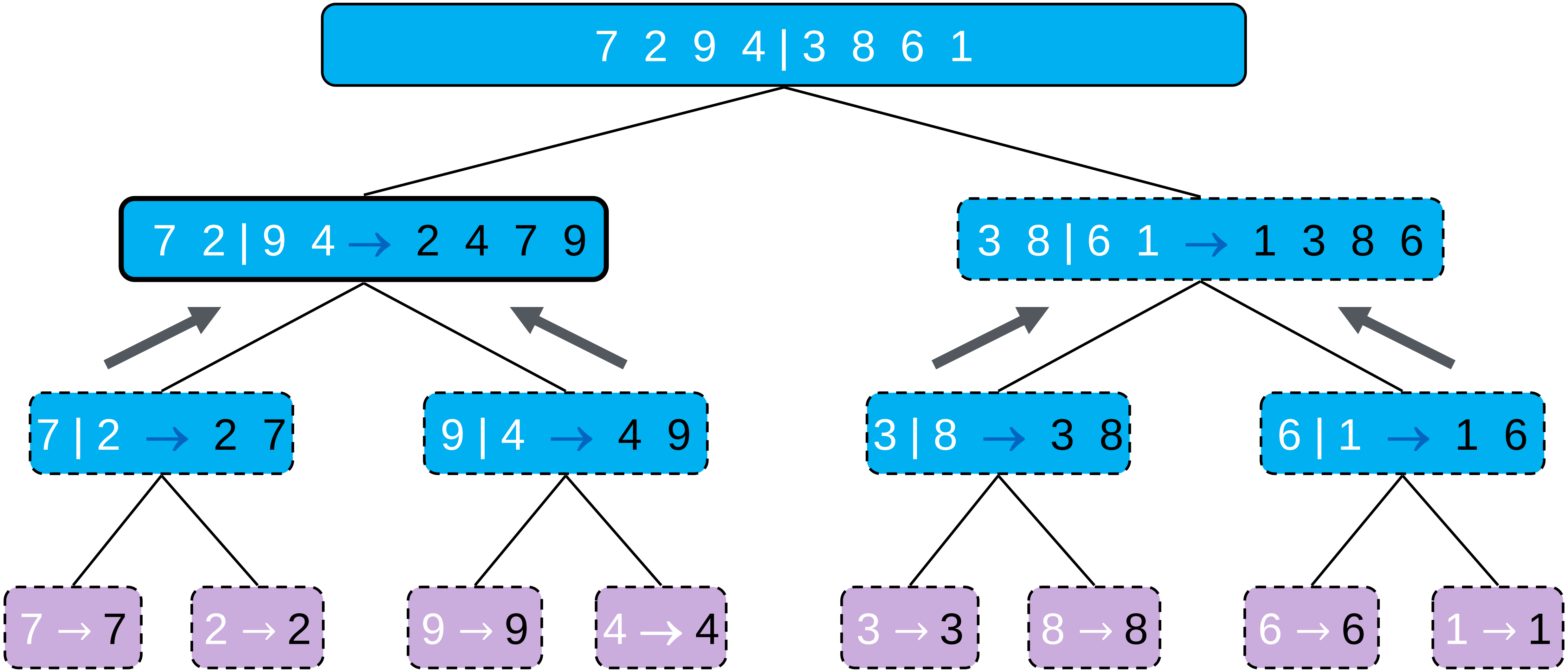
Execution Example (cont.)

- Merge



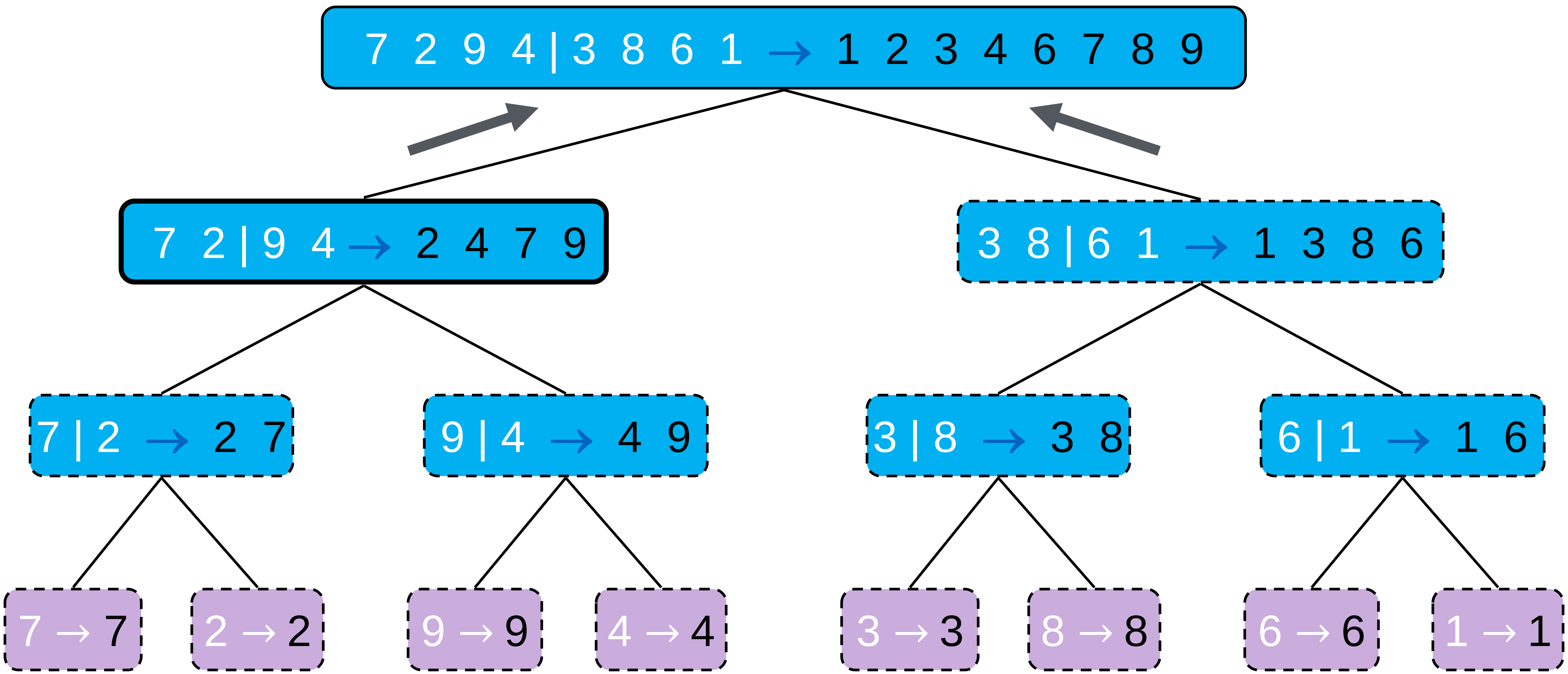
Execution Example (cont.)

- Merge



Execution Example (cont.)

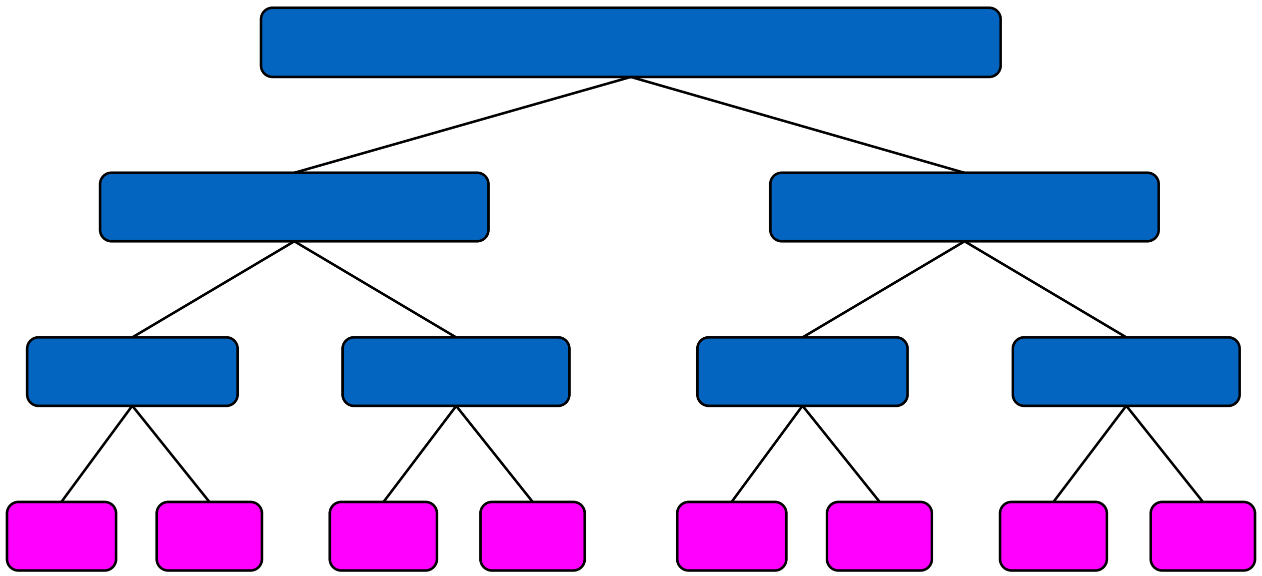
- Merge



Analysis of Merge-Sort

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

depth	#seqs	size
0	1	$\boldsymbol{n}$
1	2	$\boldsymbol{n}/2$
$\boldsymbol{i}$	2^i	$\boldsymbol{n}/2^i$
…	…	…



Did we achieve todays objectives?

- Priority Queues
- Binary Heap
- Heap-Sort
- Merge-Sort