COMP26120: Algorithms and Imperative Programming

Lecture 2

Data structures for binary trees

Priority queues

Heaps

Lecture outline

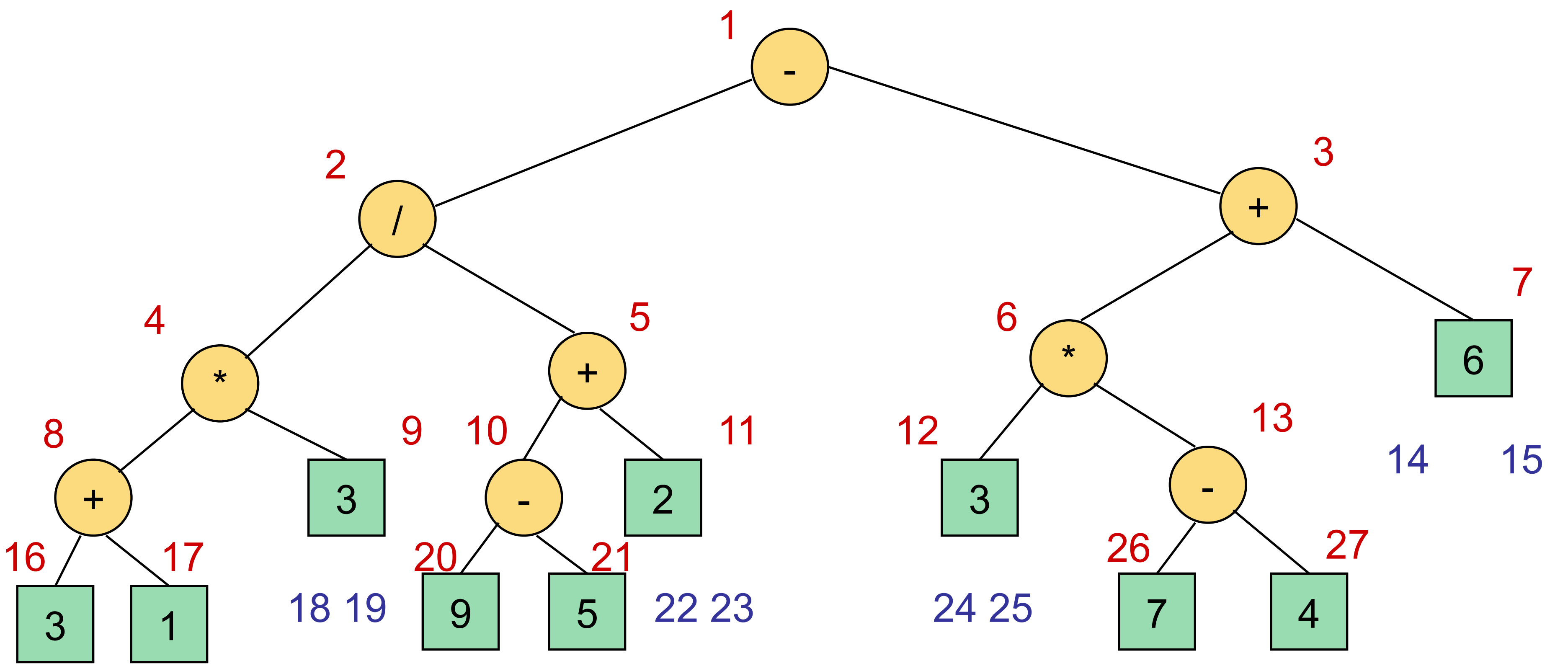
- Different data structures for representing binary trees (vector-based, linked), linked structure for general trees;
- Priority queues (PQs);
- The heap data structure;
- Implementing priority queues as heaps;
- The vector representation of a heap and basic operations (insertion, removal);

Data structures for representing trees
A vector-based data structure

- A vector-based structure for binary trees is based on a simple way of numbering the nodes of *T*.
- For every node *v* of *T* define an integer *p(v)*:
  - If *v* is the root, then *p(v)*=1;
  - If *v* is the left child of the node *u*, then *p(v)*=2*p(u)*;
  - If *v* is the right child of the node *u*, then *p(v)*=2*p(u)+1*;
- The numbering function *p(.)* is known as a level numbering of the nodes in a binary tree *T*.

Data structures for representing trees
A vector-based data structure

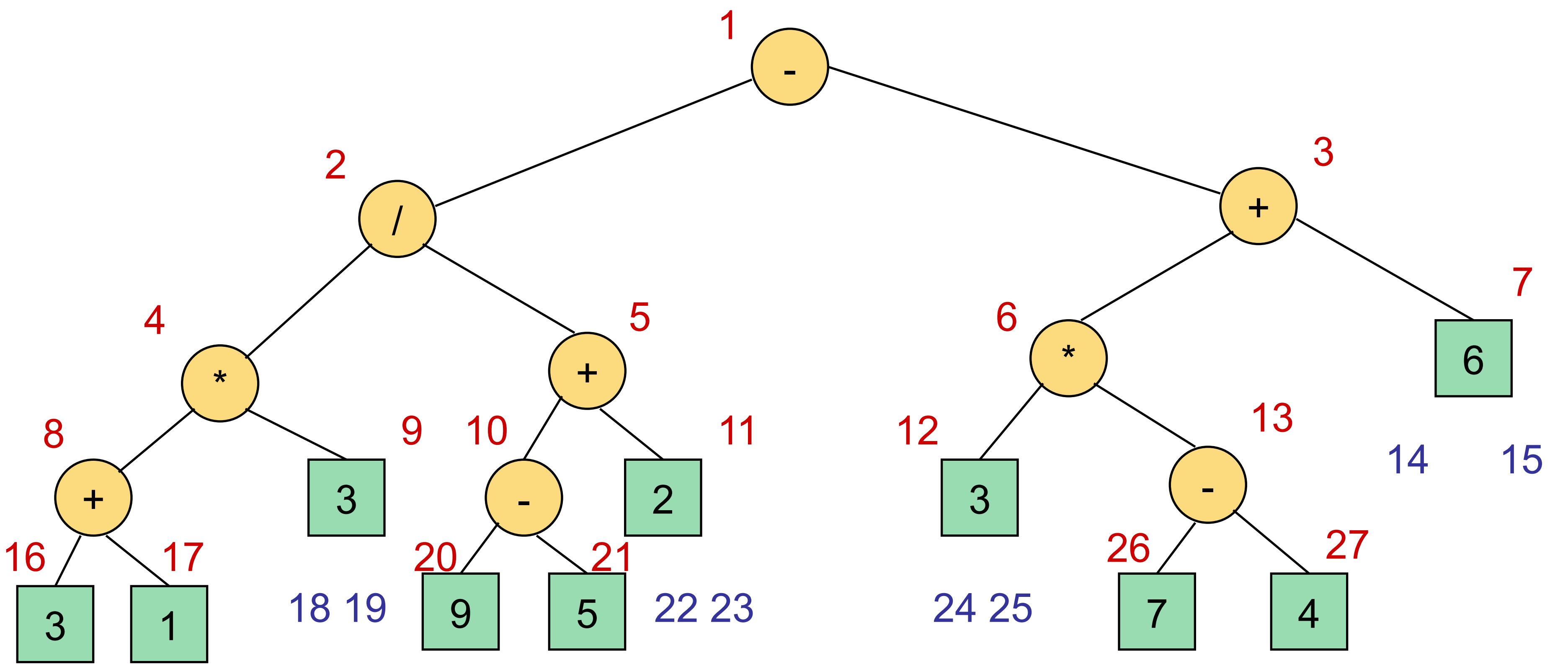
- ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))



Binary tree level numbering

Data structures for representing trees
A vector-based data structure

- The level numbering suggests a representation of a binary tree *T* by a vector *S*, such that the node *v* from *T* is associated with an element *S*[*p(v)*];



S

-	/	+	*	+	*	6	+	3	-	2	3	-			3	1			9	5					7	4

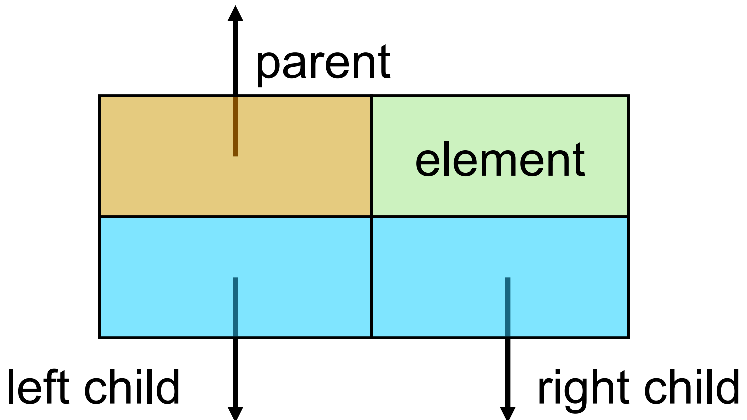
Data structures for representing trees
A vector-based data structure

Operation	Time
positions(), elements()	$O(n)$
swapElements(), replaceElement()	$O(1)$
root(), parent(), children()	$O(1)$
leftChild(), rightChild(), sibling()	$O(1)$
isInternal(), isExternal(), isRoot()	$O(1)$

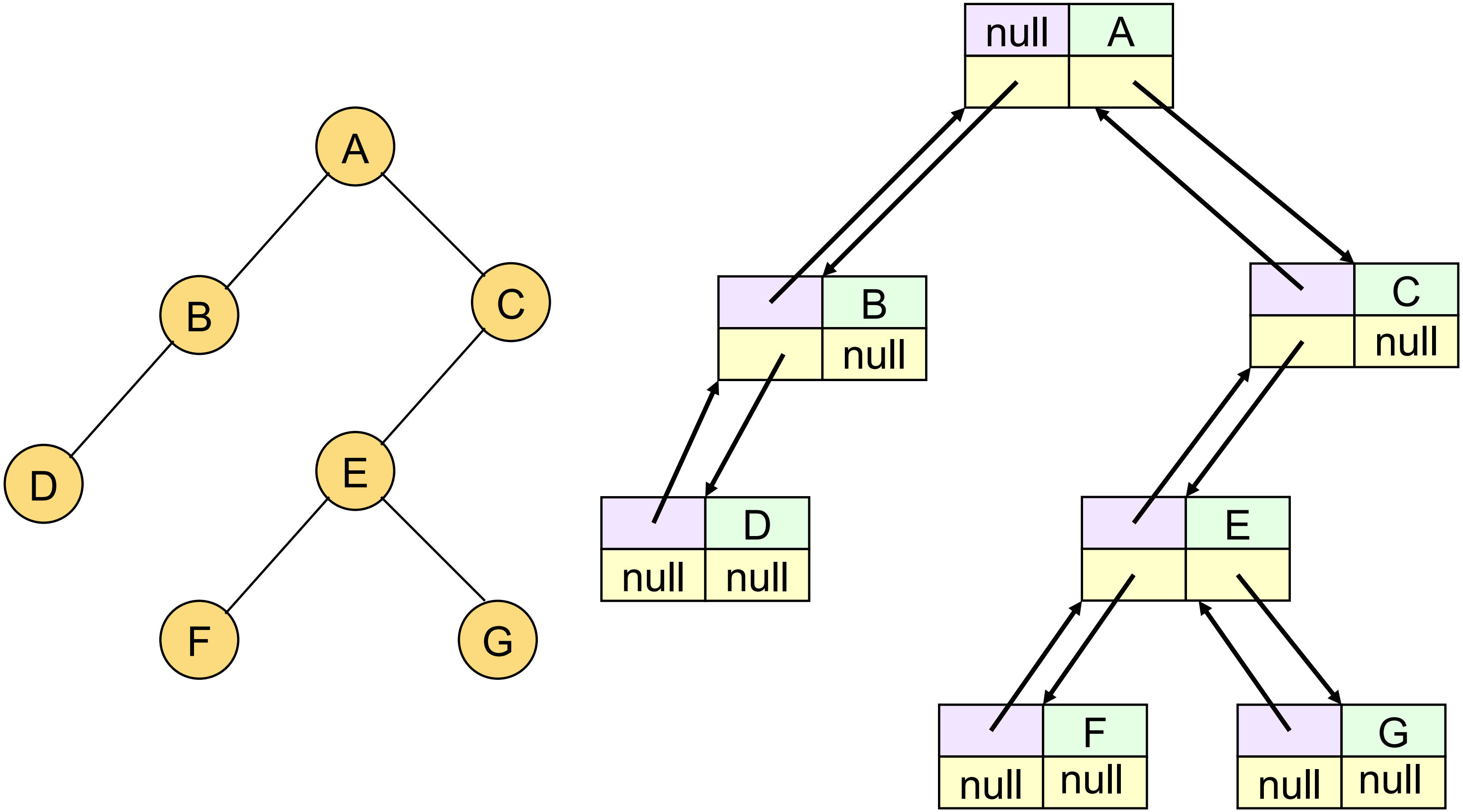
Running times of the methods when a binary tree *T* is implemented as a vector

Data structures for representing trees
A linked data structure

- The vector implementation of a binary tree is fast and simple, but it may be space inefficient when the tree height is large (why?);
- A natural way of representing a binary tree is to use a linked structure.
- Each node of *T* is represented by an object that references to the element *v* and the positions associated with with its parent and children.

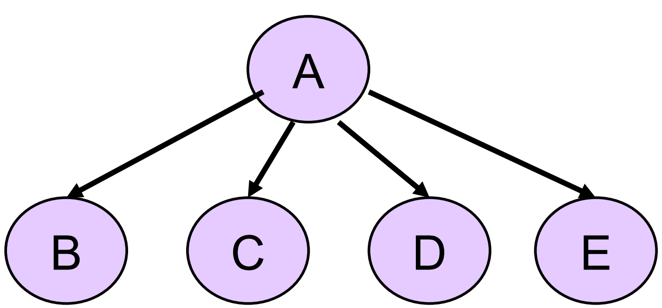


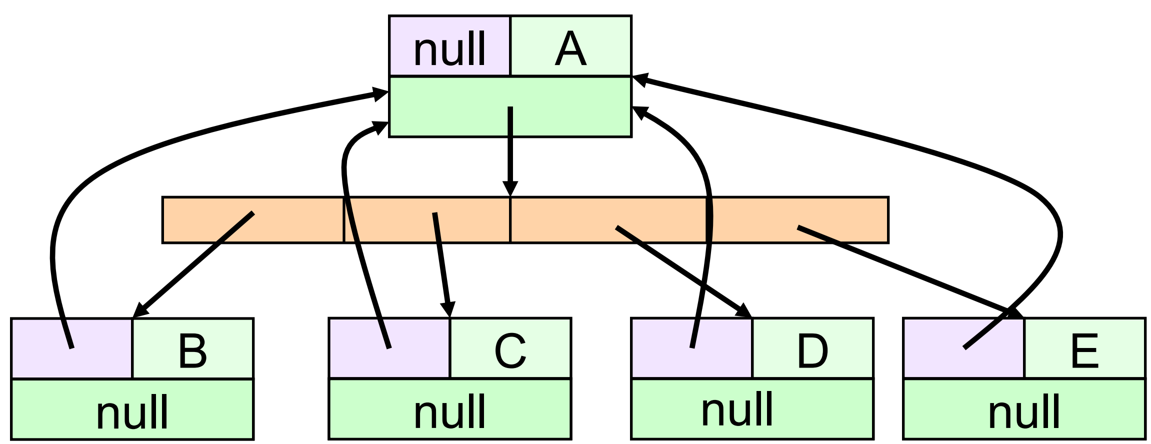
Data structures for representing trees
A linked data structure for binary trees



Data structures for representing trees
A linked data structure for general trees

- Extends the previous data structure to the case of general trees;
- In order to register a potentially large number of children of a node, we need to use a container (a list or a vector) to store the children, instead of using instance variables;





Keys and the total order relation

- In various applications it is frequently required to compare and rank objects according to some parameters or properties, called keys that are assigned to each object in a collection.
- A key is an object assigned to an element as a specific attribute that can be used to identify, rank or weight that element.
- A rule for comparing keys needs to be robustly defined (not contradicting).
- We need to define a total order relation, denoted by $\leq$ with the following properties:
  - Reflexive property: $k \leq k$;
  - Antisymmetric property: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$;
  - Transitive property: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$;
- The comparison rule that satisfies the above properties defines a linear ordering relationship among a set of keys.
- In a finite collection of elements with a defined total order relation we can define the smallest key $k_{\min}$ as the key for which $k_{\min} \leq k$ for any other key *k* in the collection.

Priority queues

- A priority queue *P* is a container of elements with keys associated to them at the time of insertion.
- Two fundamental methods of a priority queue *P* are:
  - insertItem(*k*,*e*) – inserts an element *e* with a key *k* into *P*;
  - removeMin() – returns and removes from *P* an element with a smallest key;
- The priority queue ADT is simpler than that of the sequence ADT. This simplicity originates from the fact that the elements in a PQ are inserted and removed based on their keys, while the elements are inserted and removed from a sequence based on their positions and ranks.

Priority queues

- A comparator is an object that compares two keys. It is associated with a priority queue at the time of construction.
- A comparator method provides the following objects, each taking two keys and comparing them:
  - isLess(k_1, k_2) – true if $k_1 < k_2$;
  - isLessOrEqualTo(k_1, k_2) – true if $k_1 \leq k_2$;
  - isEqualTo(k_1, k_2) – true if $k_1 = k_2$;
  - isGreater(k_1, k_2) – true if $k_1 > k_2$;
  - isGreaterOrEqualTo(k_1, k_2) – true if $k_1 \geq k_2$;
  - isComparable(*k*) – true if *k* can be compared;

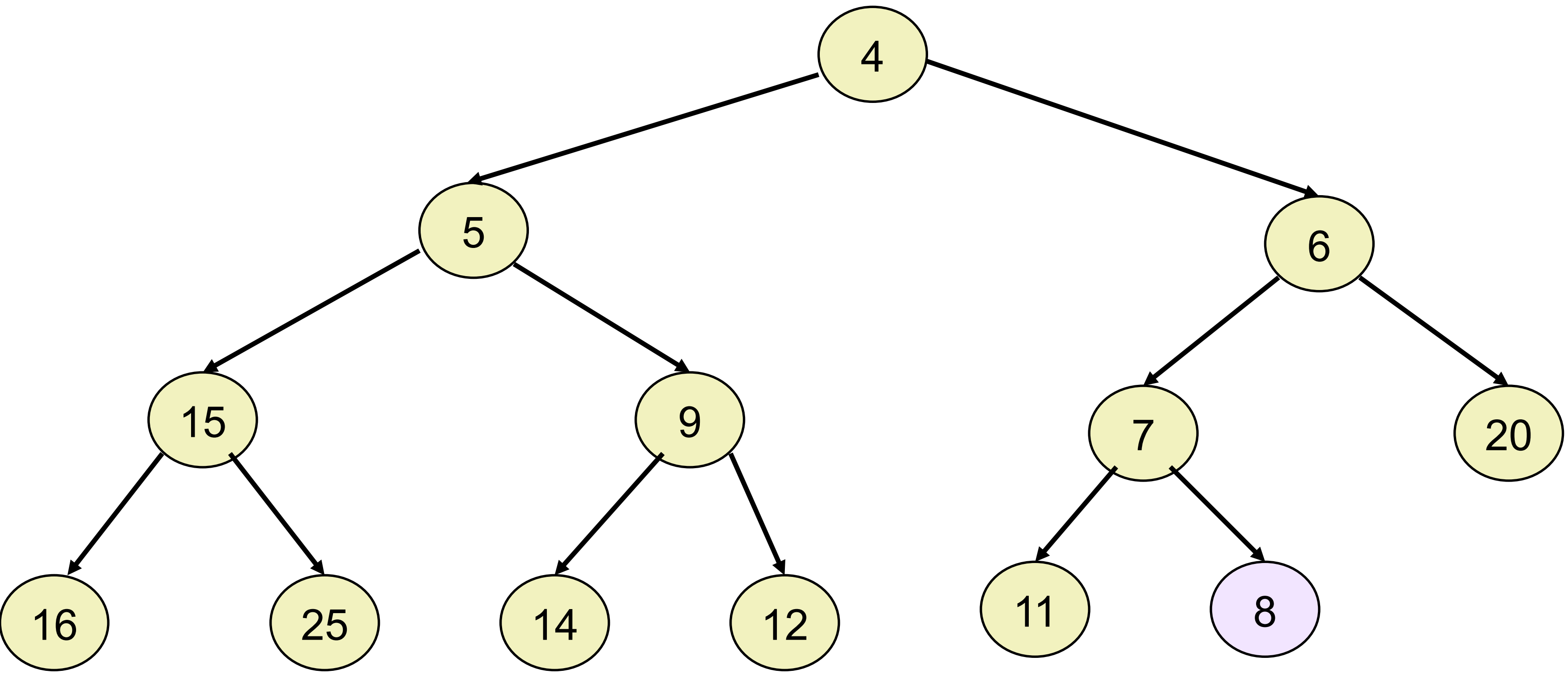
The heap data structure

- The aim is to provide a realisation of a priority queue that is efficient for both insertions and removals.
- This can be accomplished with a data structure called a heap, which enables to perform both insertions and removals in logarithmic time.
- The idea is to store the elements in a binary tree instead of a sequence.

The heap data structure

- A heap is a binary tree that stores a collection of keys at its internal nodes that satisfies two additional properties:
  - A relational property (that affects how the keys are stored);
  - A structural property;
- We assume a total order relationship on the keys.
- Heap-Order property: In a heap *T* for every node *v* other than a root, the key stored in *v* is greater or equal than the key stored at its parent.
- The consequence is that the keys encountered on a path from the root to an external node are in non-decreasing order and that a minimum key is always stored at the root.
- Complete binary tree property: A binary tree *T* with height *h* is complete if the levels 0 to *h*-1 have the maximum number of nodes (level *i* has 2^i nodes for *i*=0,…,h-1) and in the level *h*-1 all internal nodes are to the left of the external nodes.

The heap data structure

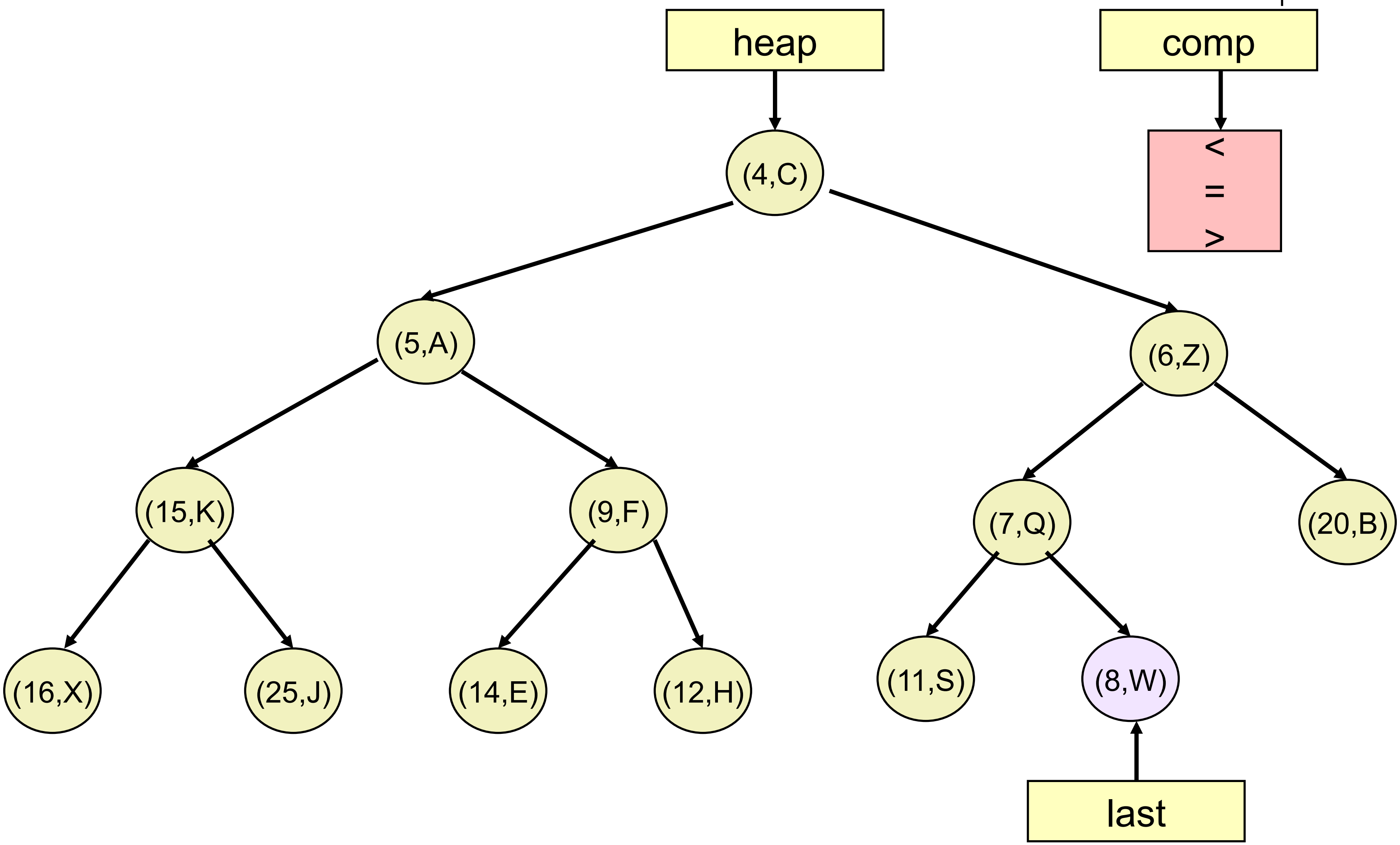


An example of a heap *T* storing 13 integer keys. The last node (the right-most, deepest internal node of *T*) is 8 in this case

Implementing a Priority Queue using a Heap

- A heap-based priority queue consists of:
  - **heap:** a complete binary tree with keys that satisfy the heap-order property. The binary tree is implemented as a vector.
  - **last:** A reference to the last node in *T.* For a vector implementation, last is an integer index to the vector element storing the last node of *T*.
  - **comp:** A comparator that defines the total order relation among the keys. The comparator should maintain the minimal element at the root.
- A heap *T* with *n* keys has height $h = \lceil \log(n+1) \rceil$
- If the update operations on a heap can be performed in time proportional to its height, rather than to the number of its elements, then these operations will have complexity $O(\log n)$.

Implementing a Priority Queue using a Heap



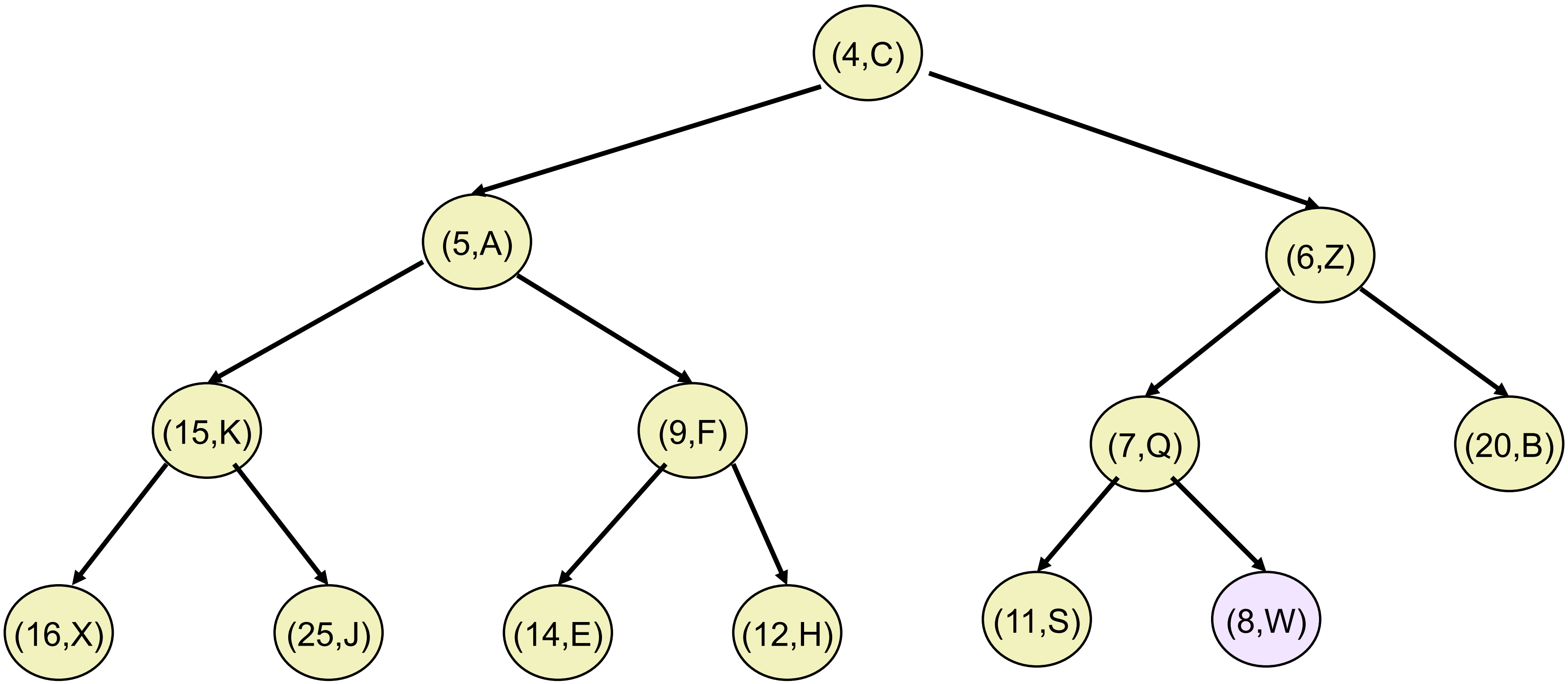
Insertion into the PQ implemented using a heap

- In order to store a new key-element pair (*k,e*) into *T*, we need to add a new node to *T*. To keep the complete tree property, the new node must become the last node of *T*.
- If a heap is implemented as a vector, the insertion node is added at index *n*+1, where *n* is the current size of the heap.

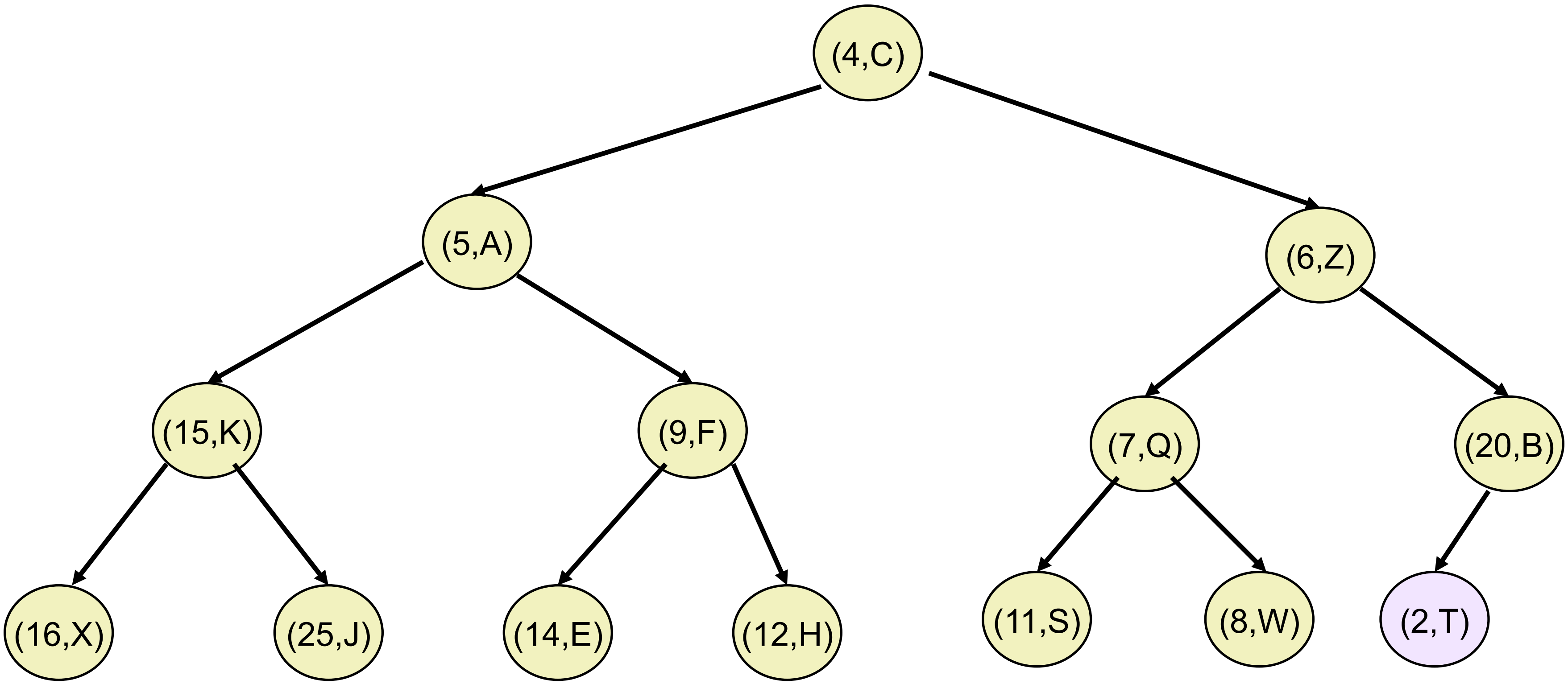
Up-heap bubbling after an insertion

- After the insertion of the element z into the tree T, it remains complete, but the heap-order property may be violated.
- Unless the new node is the root (the PQ was empty prior to the insertion), we compare keys $k(z)$ and $k(u)$ where u is the parent of z. If $k(u)>k(z)$, the heap order property needs to be restored, which can locally be achieved by swapping the pairs $(u,k(u))$ and $(z,k(z))$, making the element pair $(z,k(z))$ to go up one level. This upward movement caused by swaps is referred to as up-heap-bubbling.
- In the worst case the up-heap-bubbling may cause the new element to move all the way to the root.
- Thus, the worst case running time of the method insertItem() is proportional to the height of T, i.e $O(\log n)$, as T is complete.

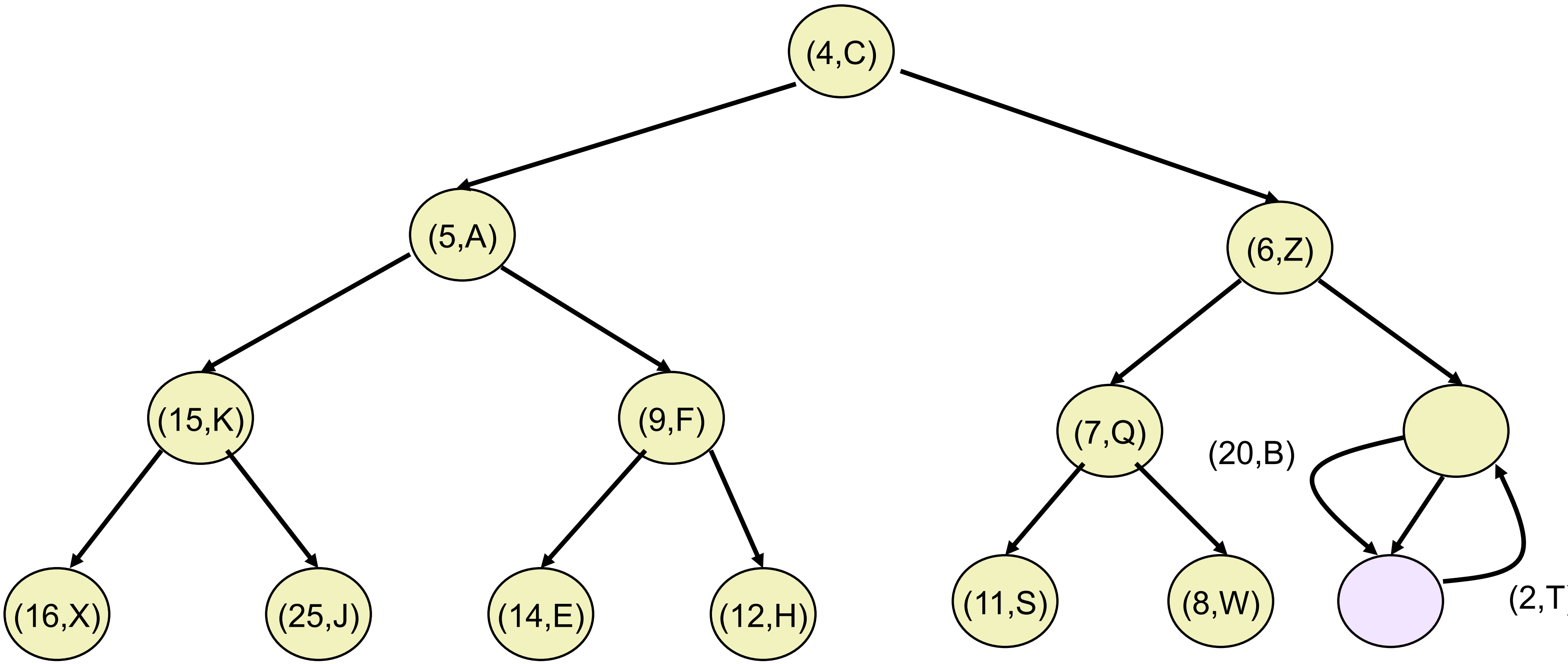
Up-heap bubbling after insertion (an example)



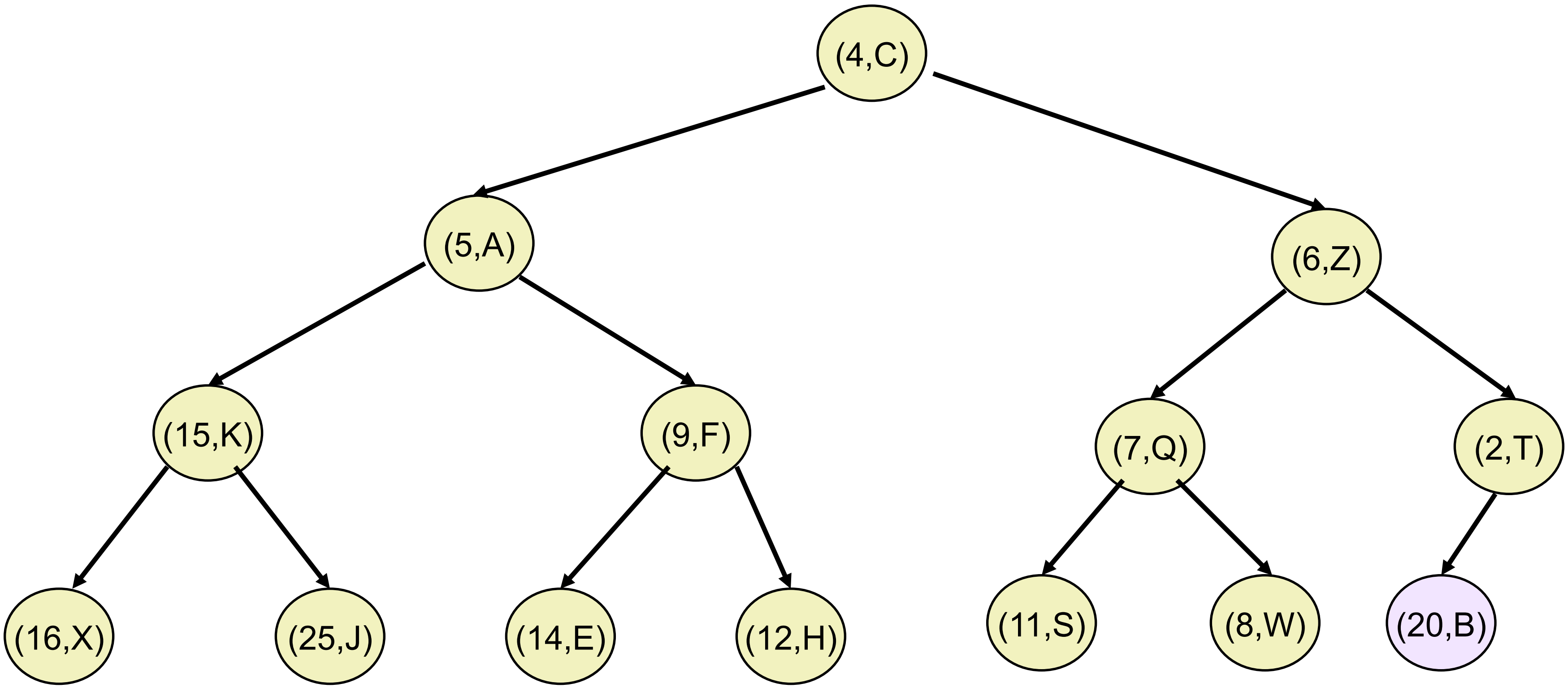
Up-heap bubbling after insertion (an example)



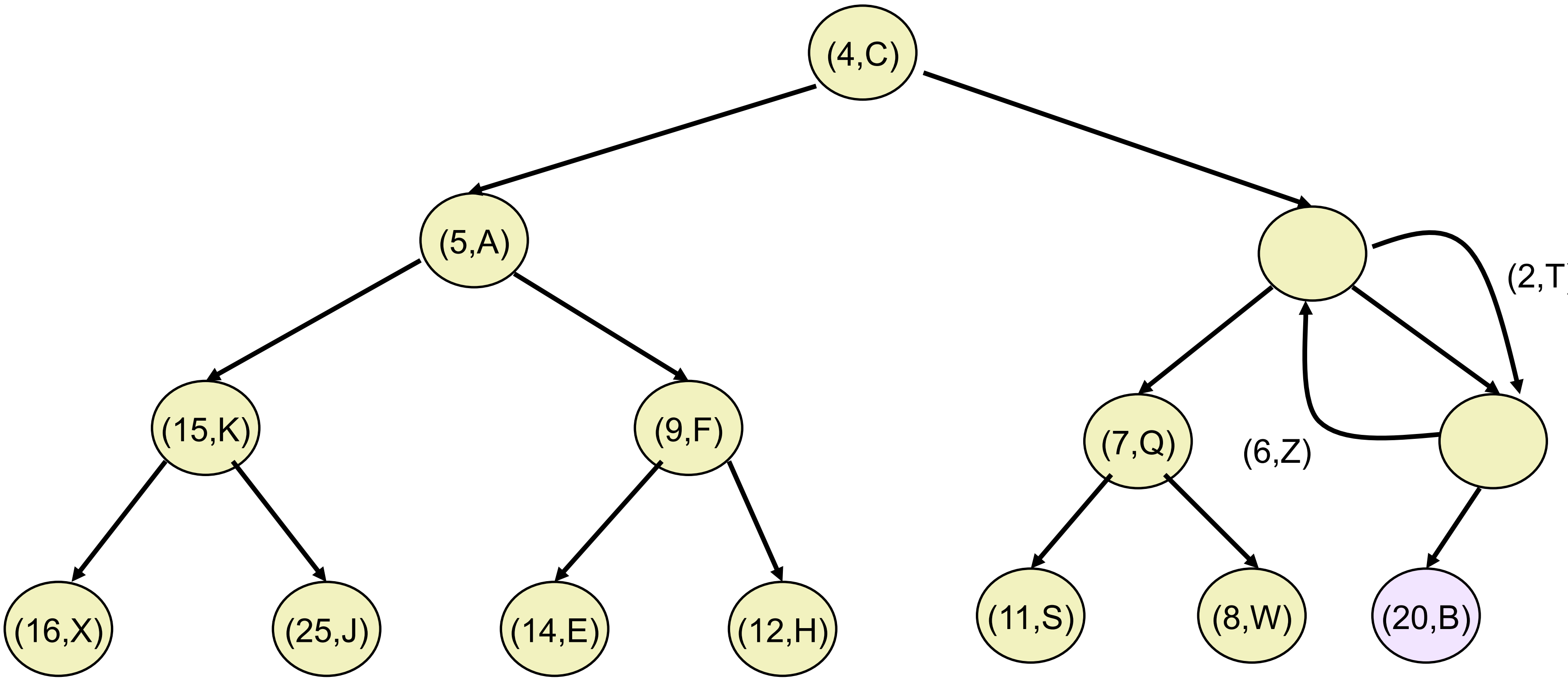
Up-heap bubbling after insertion (an example)



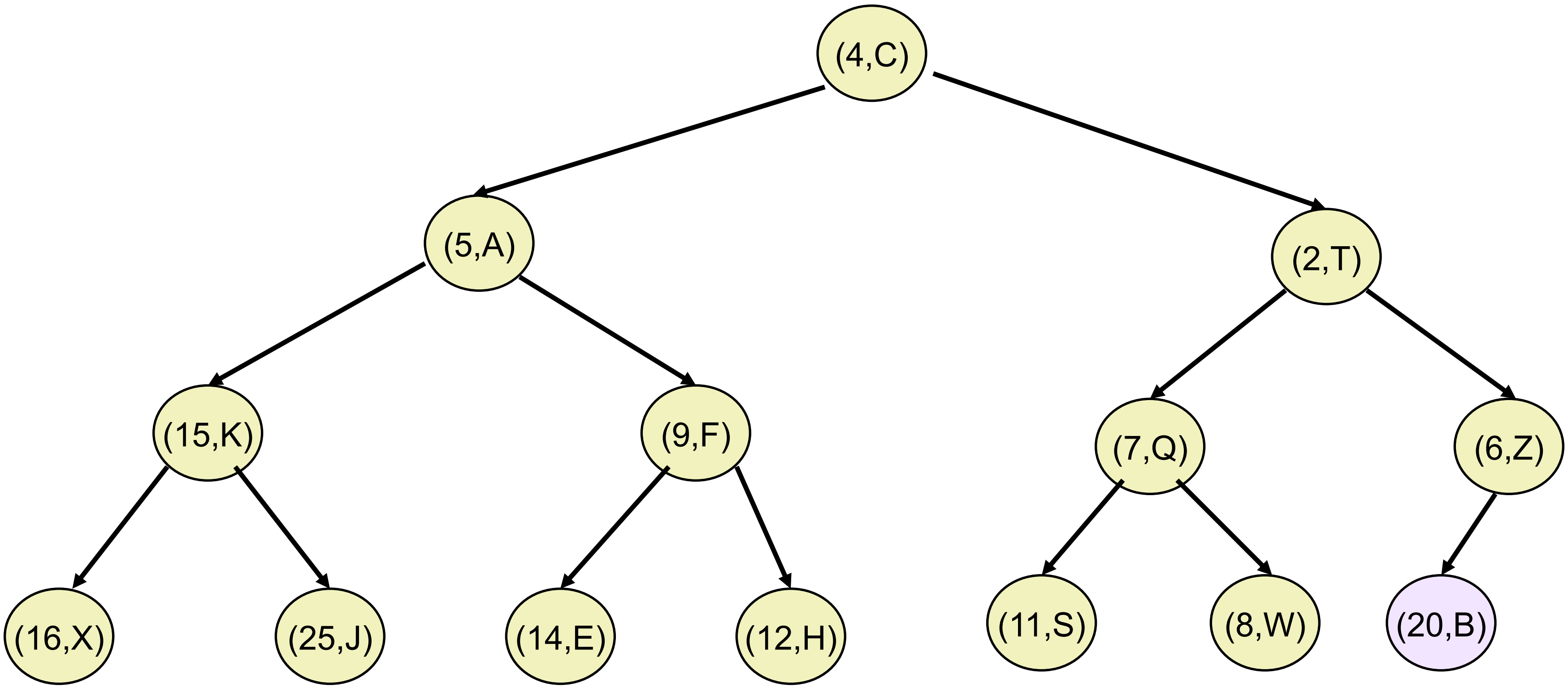
Up-heap bubbling after insertion (an example)



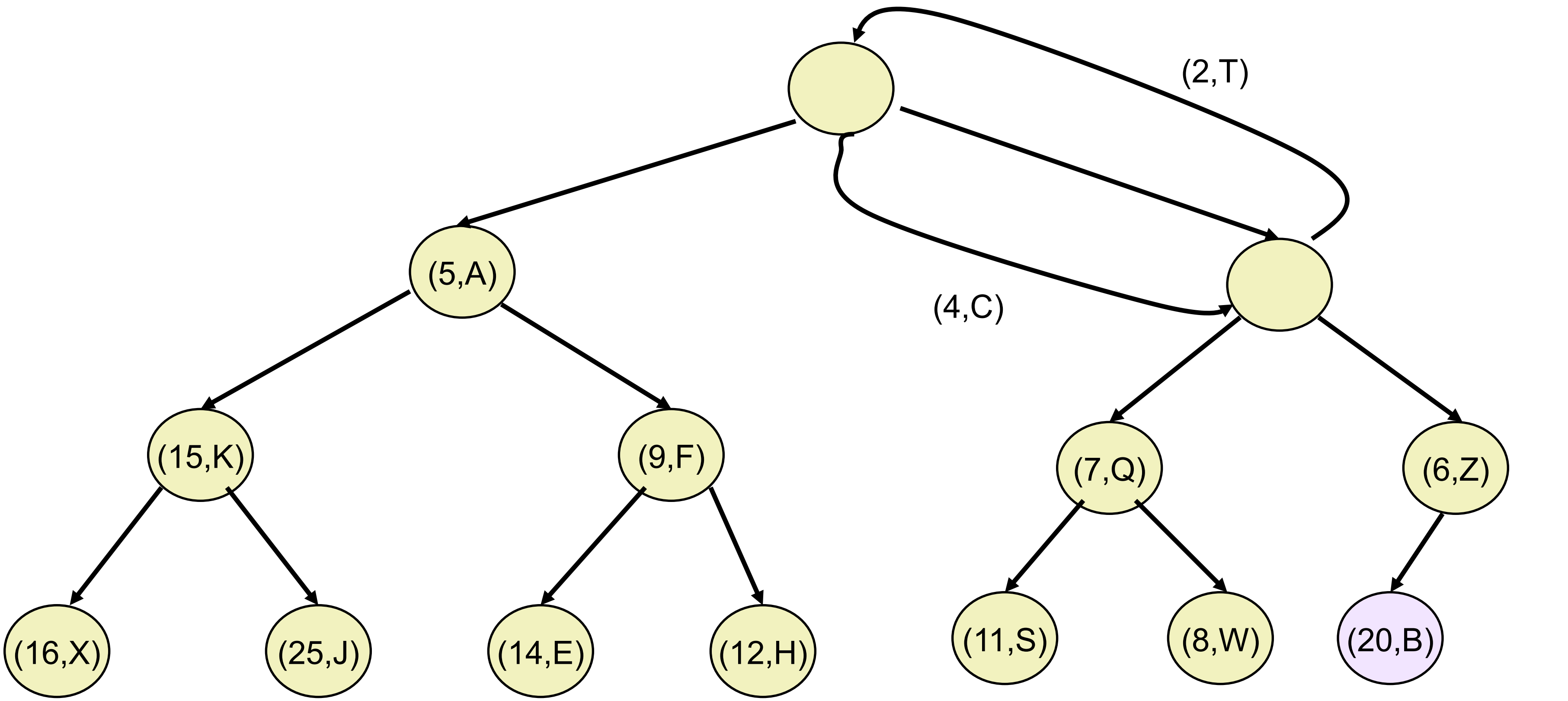
Up-heap bubbling after insertion (an example)



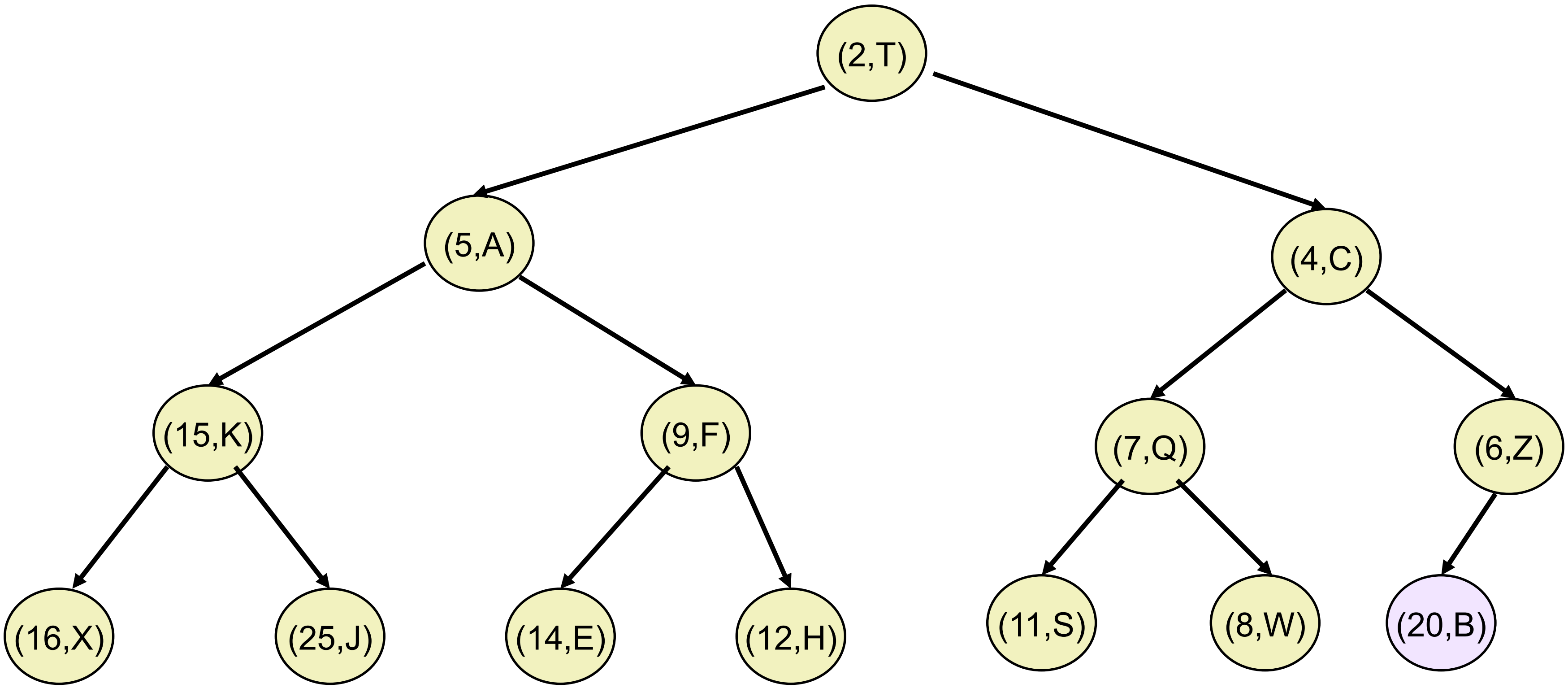
Up-heap bubbling after insertion (an example)



Up-heap bubbling after insertion (an example)



Up-heap bubbling after insertion (an example)



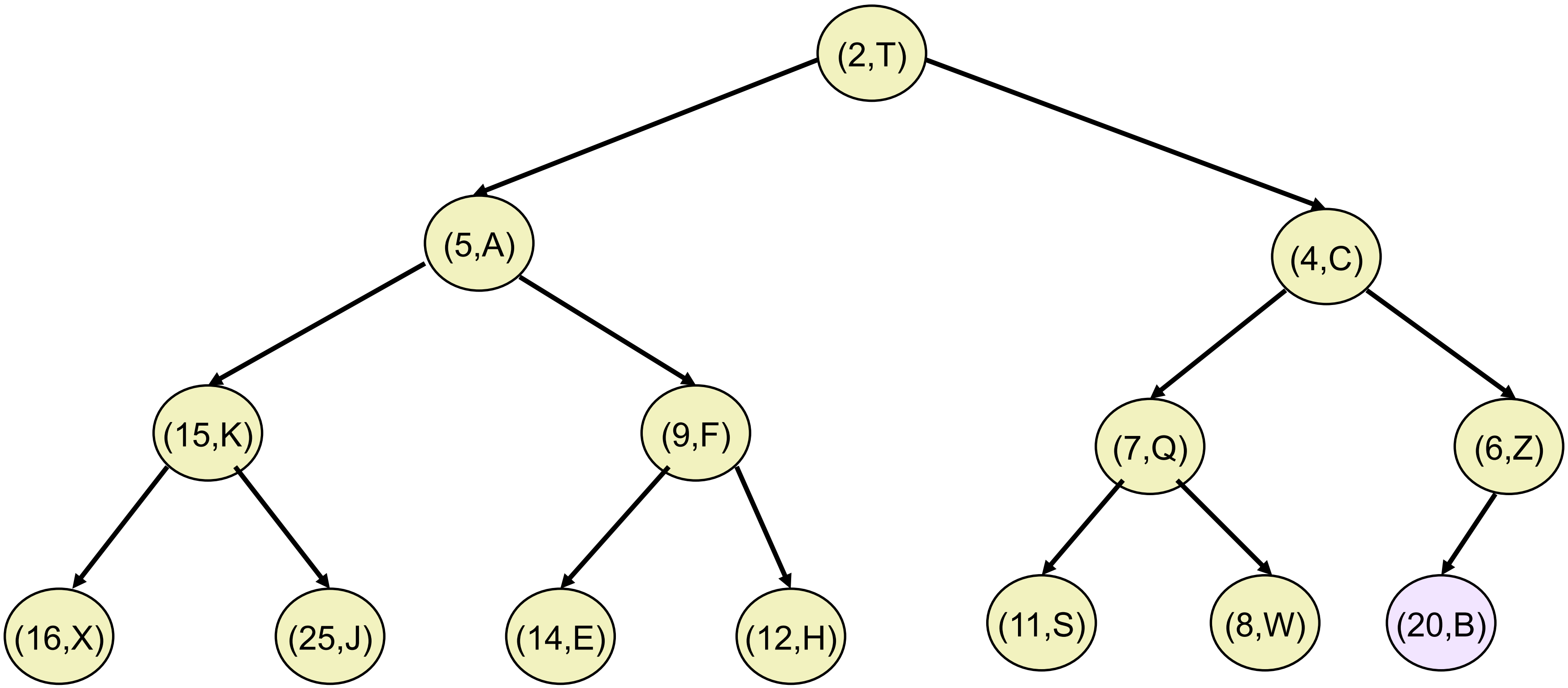
Removal from the PQ implemented as a heap

- We need to perform the method removeMin() from the PQ.
- The element *r* with a smallest key is stored at the root of the heap. A simple deletion of this element would disrupt the binary tree structure.
- We access the last node in the tree, copy it to the root, and delete it. This makes *T* complete.
- However, these operations may violate the heap-order property.

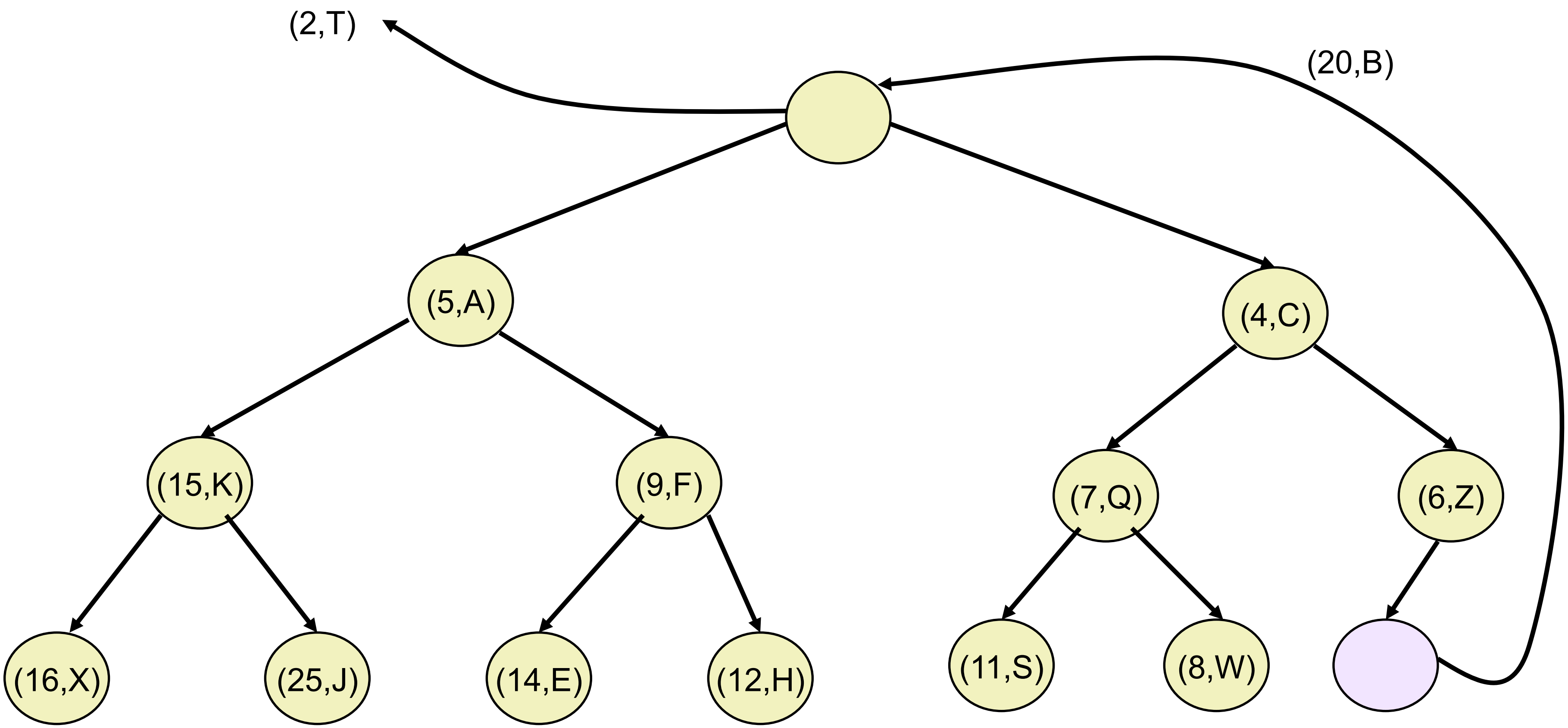
Removal from the PQ implemented as a heap

- To restore the heap-order property, we examine the root *r* of *T*. If this is the only node, the heap-order property is trivially satisfied. Otherwise, we distinguish two cases:
  - If the root has only the left child, let *s* be the left child;
  - Otherwise, let *s* be the child of *r* with the smallest key;
- If *k*(*r*)>*k*(*s*), the heap-order property is restored by swapping locally the pairs stored at *r* and *s*.
- We should continue swapping down *T* until no violation of the heap-order property occurs. This downward swapping process is referred to as down-heap bubbling. A single swap either resolves the violation of the heap-order property or propagates it one level down the heap.
- The running time of the method removeMin is thus $O(\log n)$.

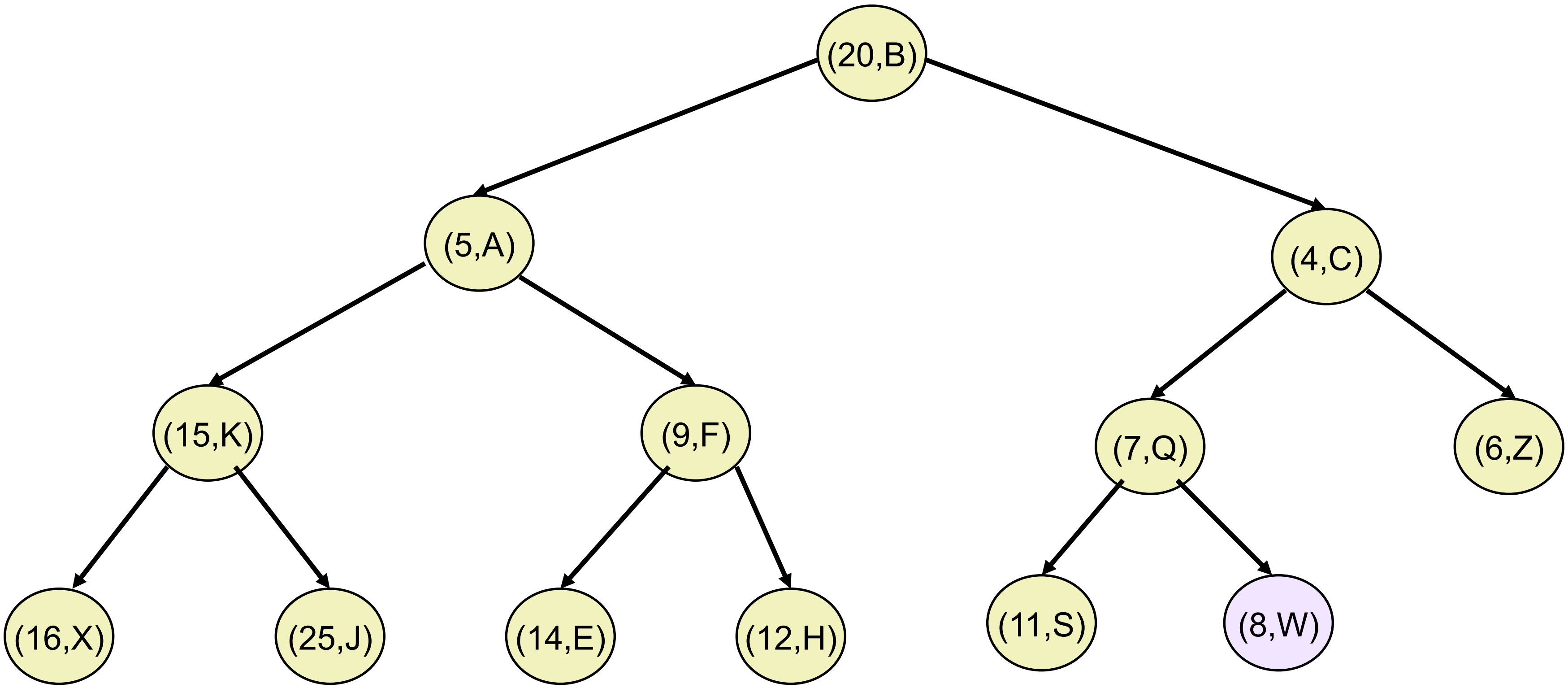
Down-heap bubbling after removal (an example)



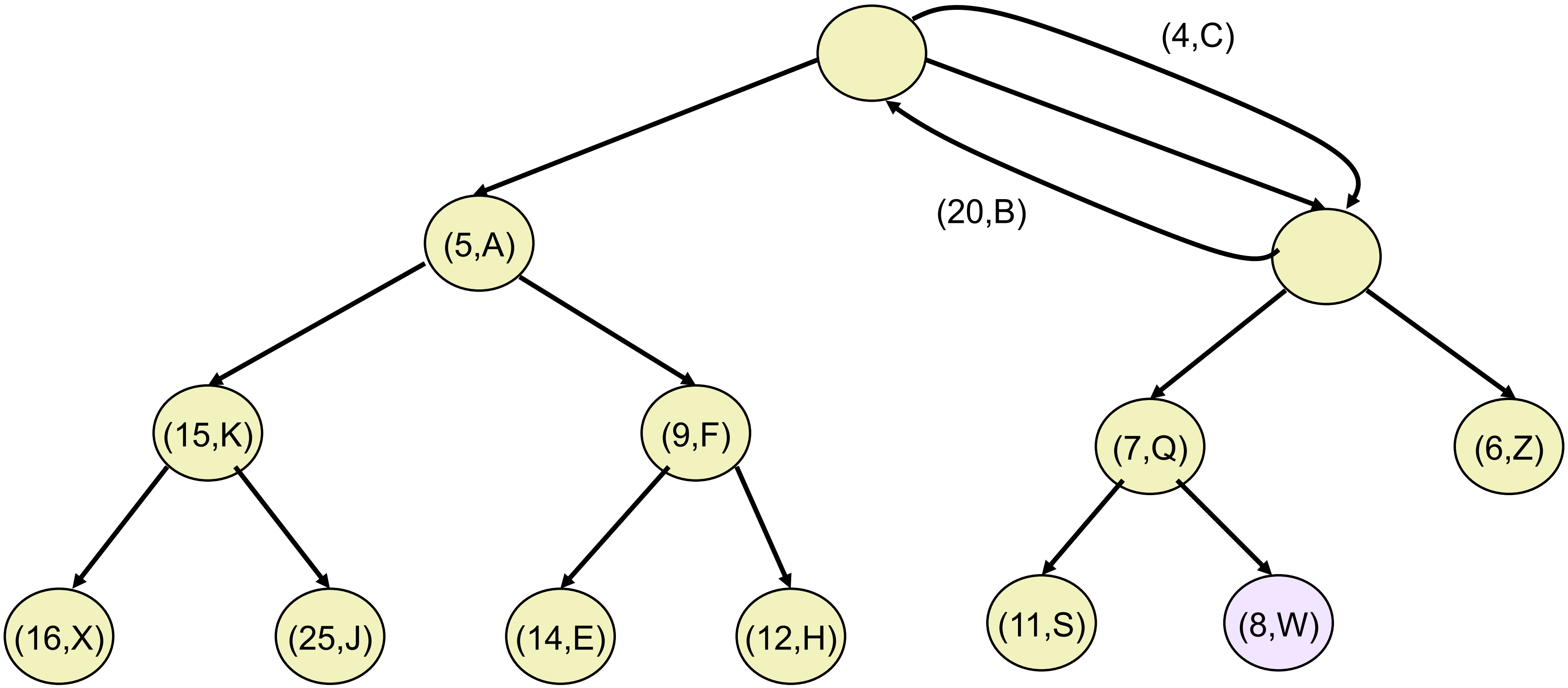
Down-heap bubbling after removal (an example)



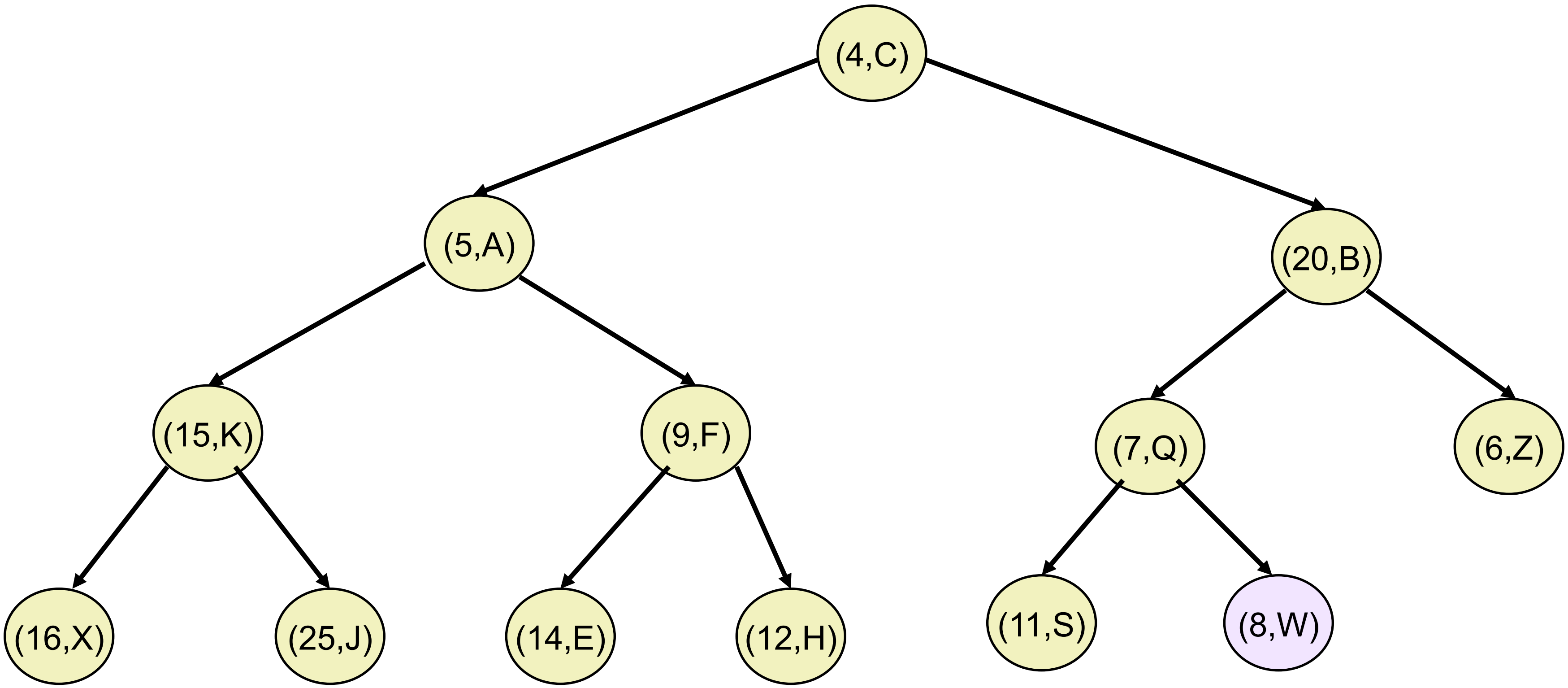
Down-heap bubbling after removal (an example)



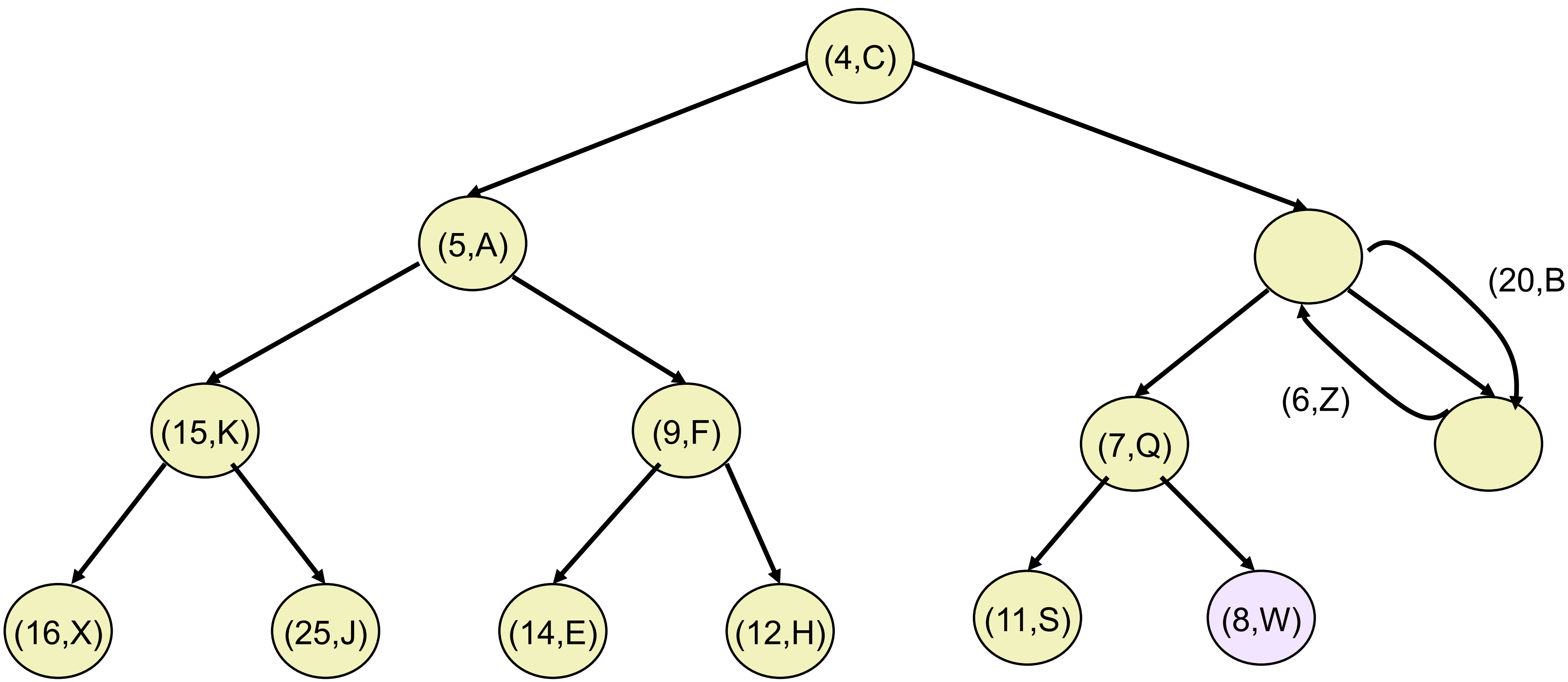
Down-heap bubbling after removal (an example)



Down-heap bubbling after removal (an example)



Down-heap bubbling after removal (an example)



Down-heap bubbling after removal (an example)