MALLA REDDY UNIVERSITY

MR22-1CS0104

ADVANCED DATA STRUCTURES

II YEAR B.TECH. (CSE) / II – SEM

Unit-2

Priority Queue: Priority Queue Abstract Data Type – Priorities, Priority Queue ADT

Heap Data Structure and its operations: The Heap Data Structure - Implementing a priority Queue with a Heap -Analysis of a Heap-based Priority Queue - Bottom-up Heap Construction - Using java.util.PriorityQueue class.

Priority Queue ADT

Priorities:

- The **queue ADT -** collection of objects added and removed according to the **FIFO** principle.
- Many applications in which a queue-like structure is used for which the first-in, first-out policy does not suffice.
- A new abstract data type known as a **priority queue**.
- Collection of prioritized elements
  - arbitrary element insertion
  - removal of the element that has first priority (minimal key)

Priority Queue ADT

- The queue ADT methods,
    - insert(k, v)
    - min( )
    - removeMin( )
    - size( )
    - isEmpty( )
- multiple entries with equivalent keys,
- **min** and **removeMin** makes an arbitrary choice.

Priority Queue ADT

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Heaps

- The two strategies for implementing a priority queue ADT,
  - **unsorted list:**
    - insertions in O(1) time
    - finding or removing an element with O(n)-time loop
  - **sorted list:**
    - find or remove the minimal element in O(1) time
    - adding a new element to the queue may require O(n) time
- More efficient realization of a priority queue using a data structure called a **binary heap.**
- perform both insertions and removals in logarithmic time.

The Heap Data Structure

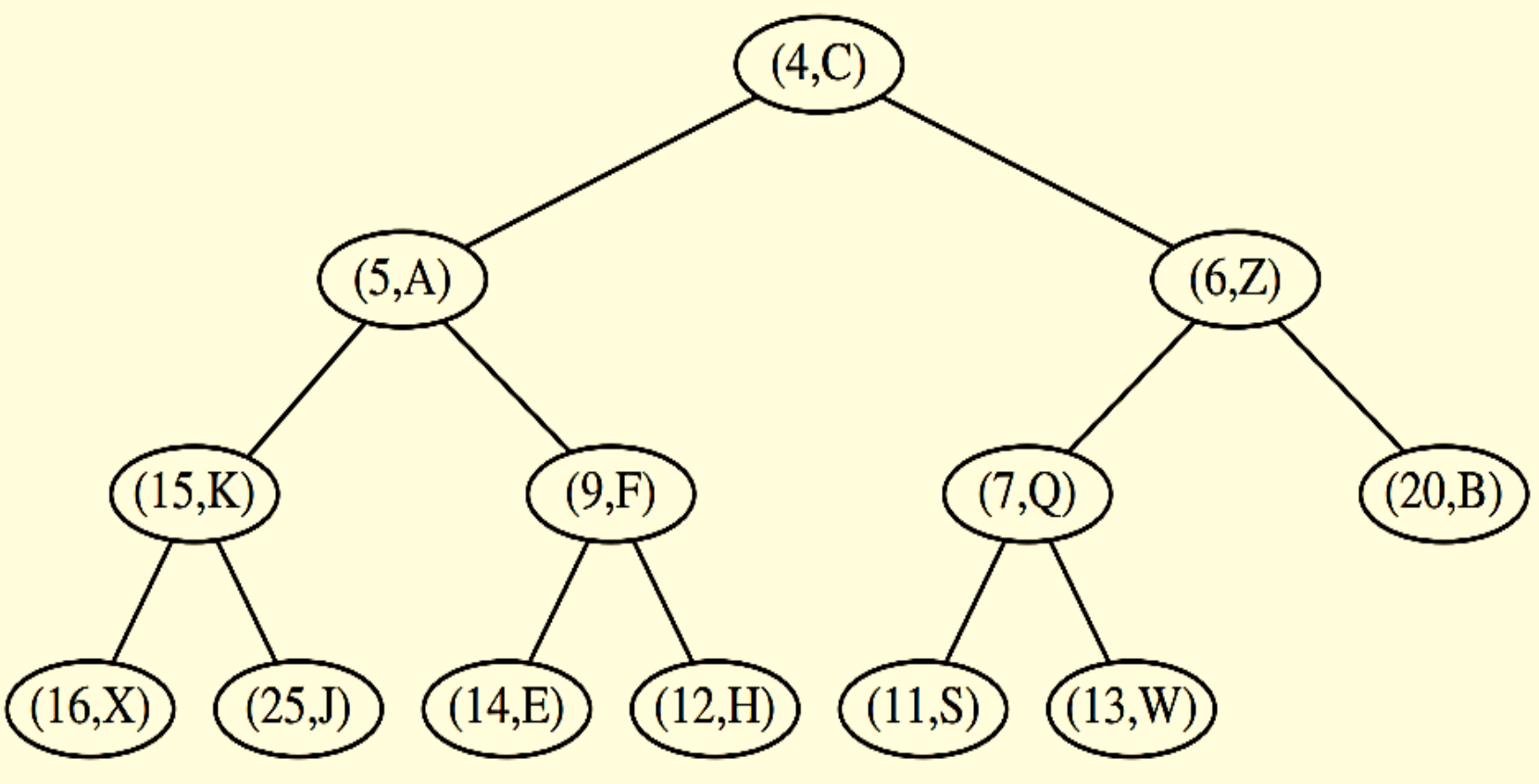
- a binary tree T that stores entries at its positions.
- Two additional properties:

  - **Heap-Order Property:** In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.

  - **Complete Binary Tree Property:** A heap T with height h is a complete binary tree if levels 0,1,2,... ,h − 1 of T have the maximal number of nodes possible and the remaining nodes at level h reside in the leftmost possible positions at that level.

The Heap Data Structure

Example of a heap storing 13 entries with integer keys



The Heap Data Structure

The Height of a Heap:

- Let h denote the height of T.
- **Proposition:** A heap T storing n entries has height

  $h = \lfloor \log n \rfloor$.
- **Justification:**

  number of nodes in levels 0 through h−1 of T is precisely

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

  - number of nodes in level h is at least 1 and at most 2^h.

  $\mathbf{n \geq 2^h - 1 + 1 = 2^h}$ **and** $\mathbf{n \leq 2^h - 1 + 2^h = 2^{h+1} - 1}$

  By taking the logarithm of both sides of inequality $n \geq 2^h$, we see that height $\mathbf{h \leq \log n}$.

  By rearranging terms and taking the logarithm of both sides of inequality $\mathbf{n \leq 2^{h+1} - 1}$, we see that $\mathbf{h \geq \log(n + 1) - 1}$.

Since h is an integer, these two inequalities imply that $\mathbf{h = \lfloor \log n \rfloor}$.

Implementing a Priority Queue with a Heap

- The size() and isEmpty() methods can be implemented based on examination of the tree.
- The min() operation is equally trivial because the heap property assures that the element at the root of the tree has a minimal key.

Adding an Entry to the Heap:

- The pair (k,v) as an entry at a new node of the tree.
- New node should be placed at a position p to maintain the complete binary tree property.

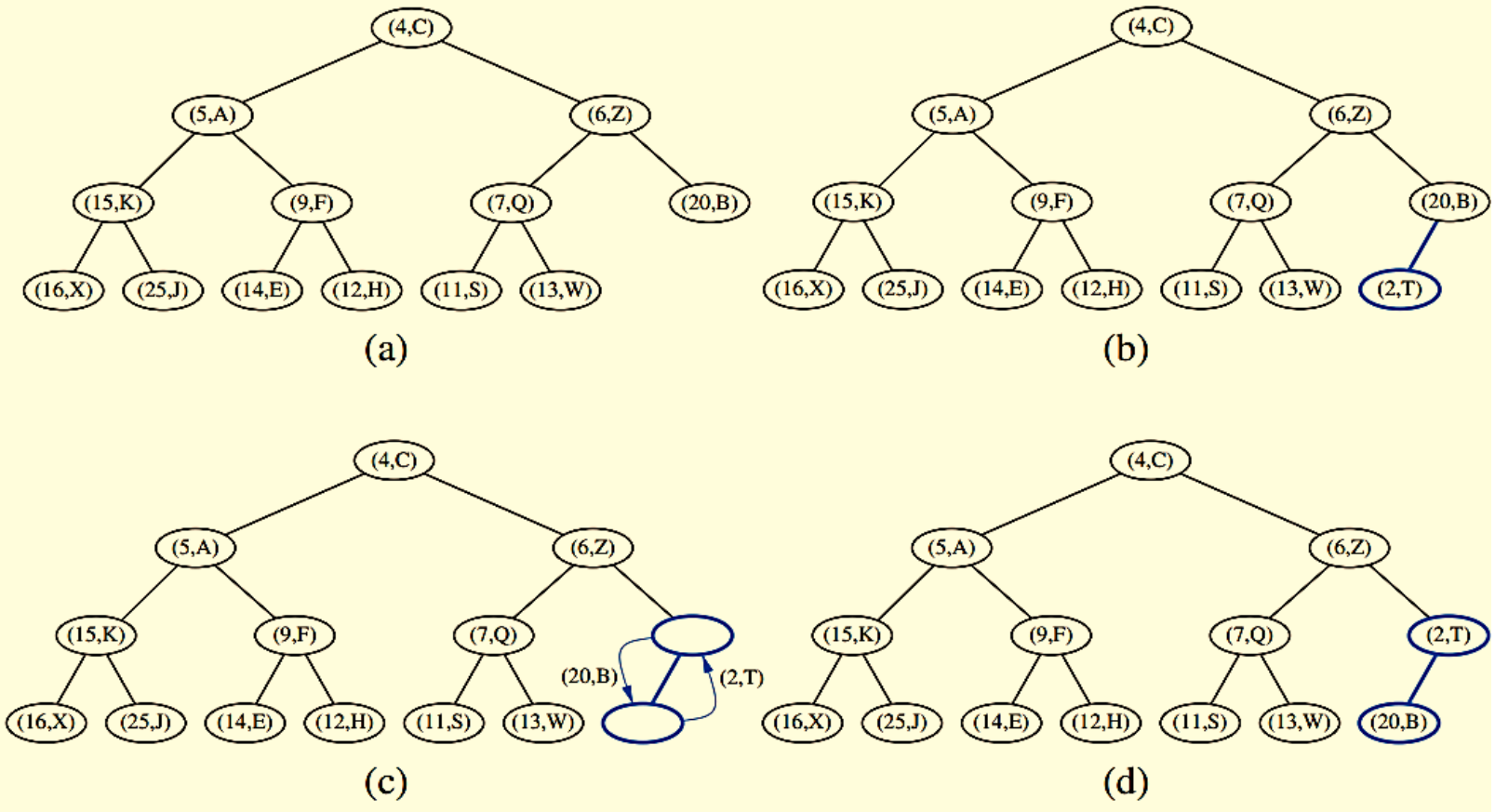
Implementing a Priority Queue with a Heap

Up-Heap Bubbling After an Insertion:

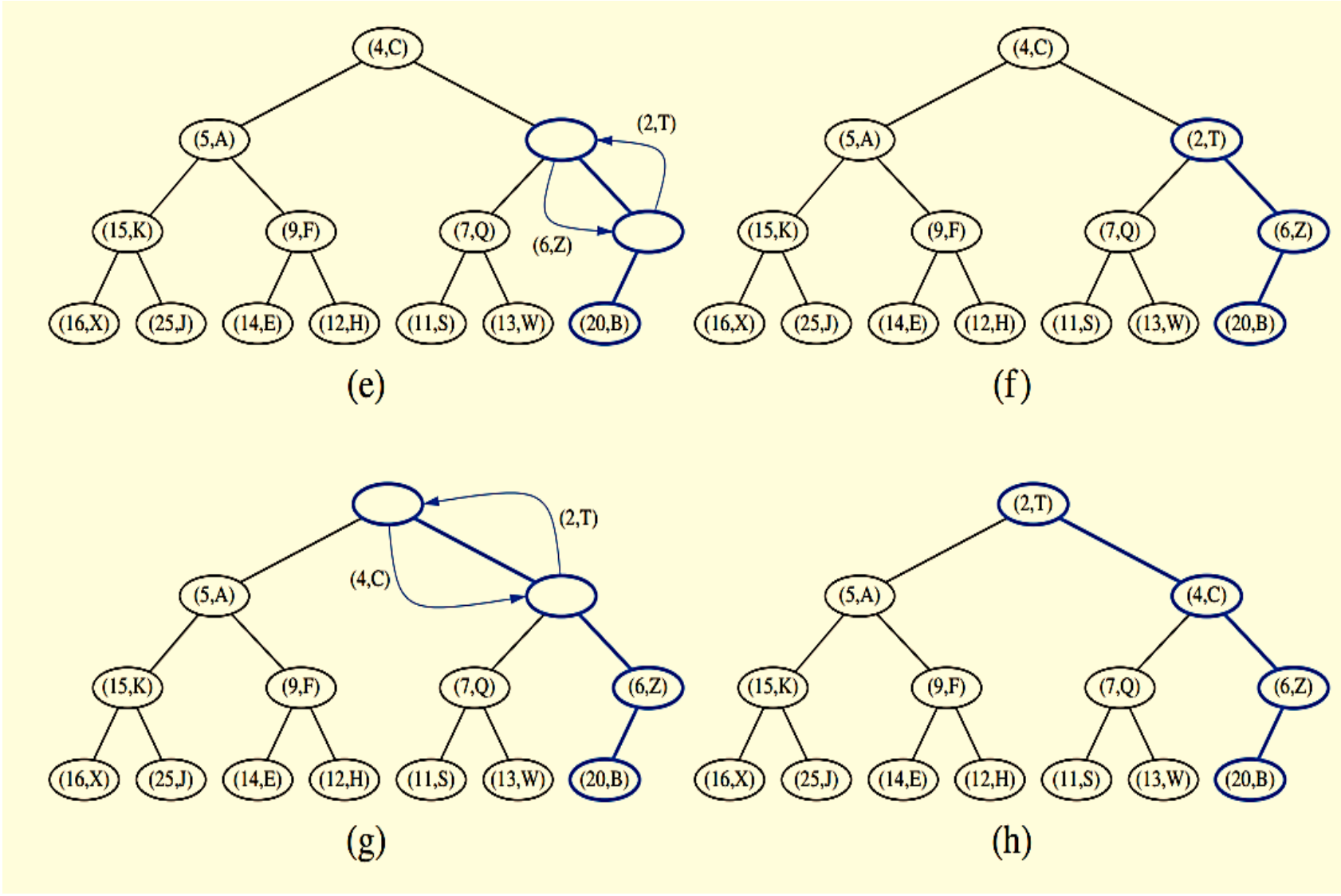
- The tree T is complete, but it may violate the heap-order property
- Compare the key at position p to that of p's parent, q.
- If key $k_p \geq k_q$, the heap-order property is satisfied.
- If $k_p < k_q$, then we need to restore the heap-order property by swapping the entries stored at positions p and q.
- This swap causes the new entry to move up one level.
- Again, the heap-order property may be violated, so we repeat the process, going up in T until no violation.
- The upward movement of the newly inserted entry by means of swaps is conventionally called **up-heap bubbling**.

Implementing a Priority Queue with a Heap

- In the worst case, the number of swaps performed is equal to the height of T.



Implementing a Priority Queue with a Heap



Implementing a Priority Queue with a Heap

Removing the Entry with Minimal Key:

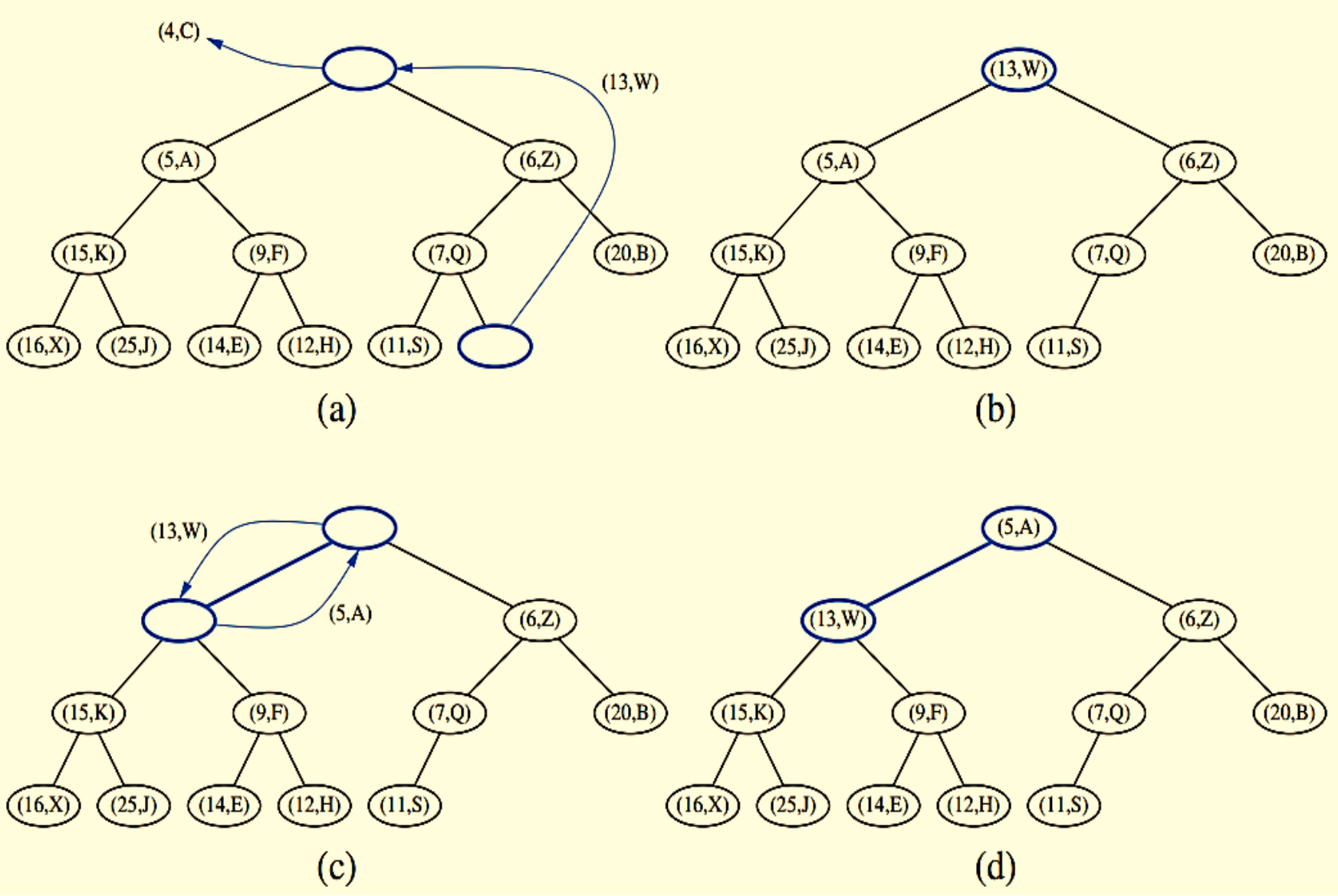
- method removeMin() of the priority queue ADT.
- entry with the smallest key is stored at the root r of T.
- delete node r, would leave two disconnected subtrees.
- ensure that the shape of the heap respects the complete binary tree property.
- delete the leaf at the last position p of T and copy it to the root r.

Implementing a Priority Queue with a Heap

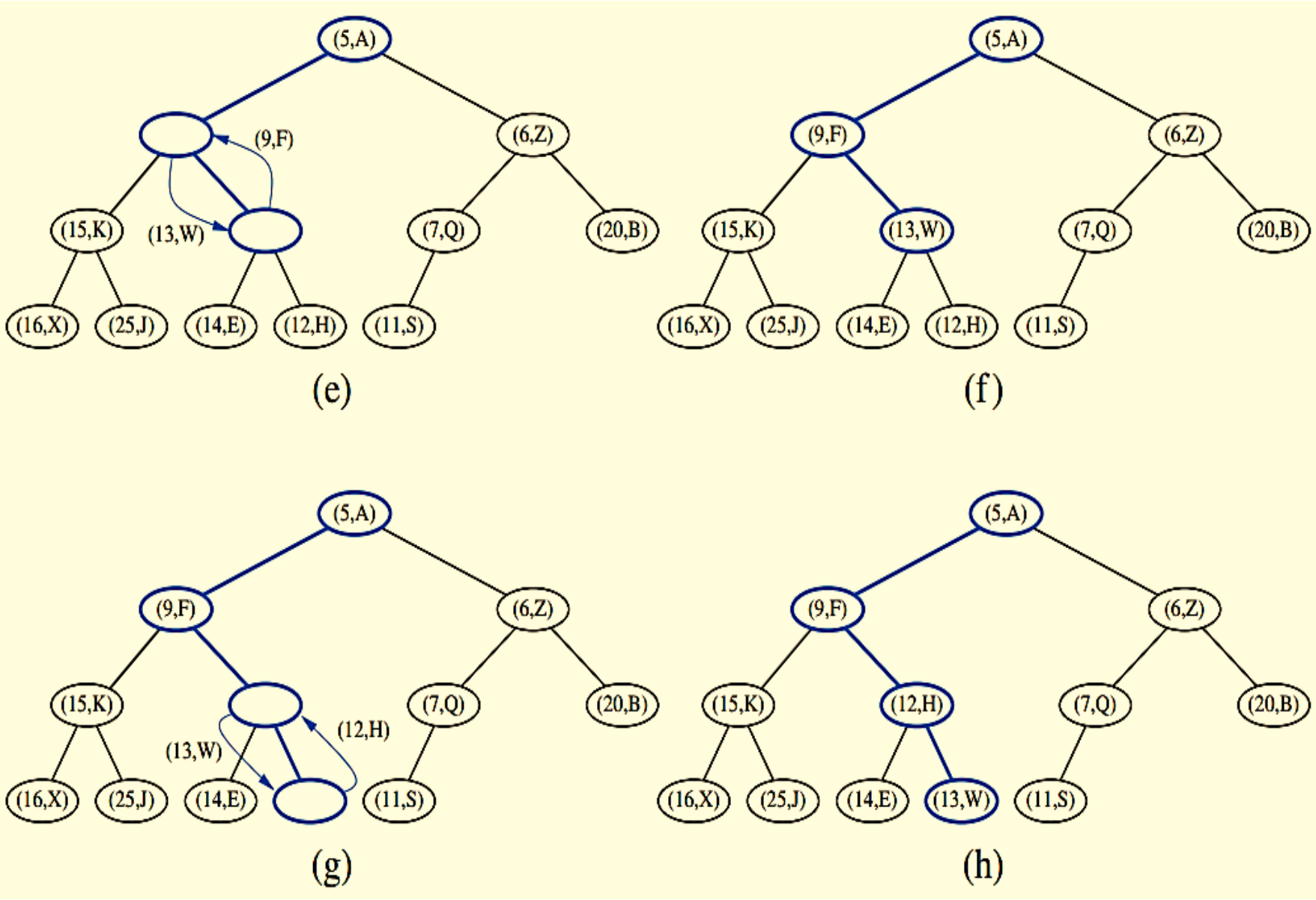
Down-Heap Bubbling After a Removal:

- Though T is now complete, it likely violates the heap-order property.
- we distinguish two cases, where p initially denotes the root of T:
    - If p has no right child, let c be the left child of p.
    - Otherwise , let c be a child of p with minimal key.
- Having restored the heap-order property for node p relative to its children, there may be a violation of this property at c
- Hence, continue swapping down T until no violation.
- This downward swapping process is called **down-heap bubbling**.

Implementing a Priority Queue with a Heap

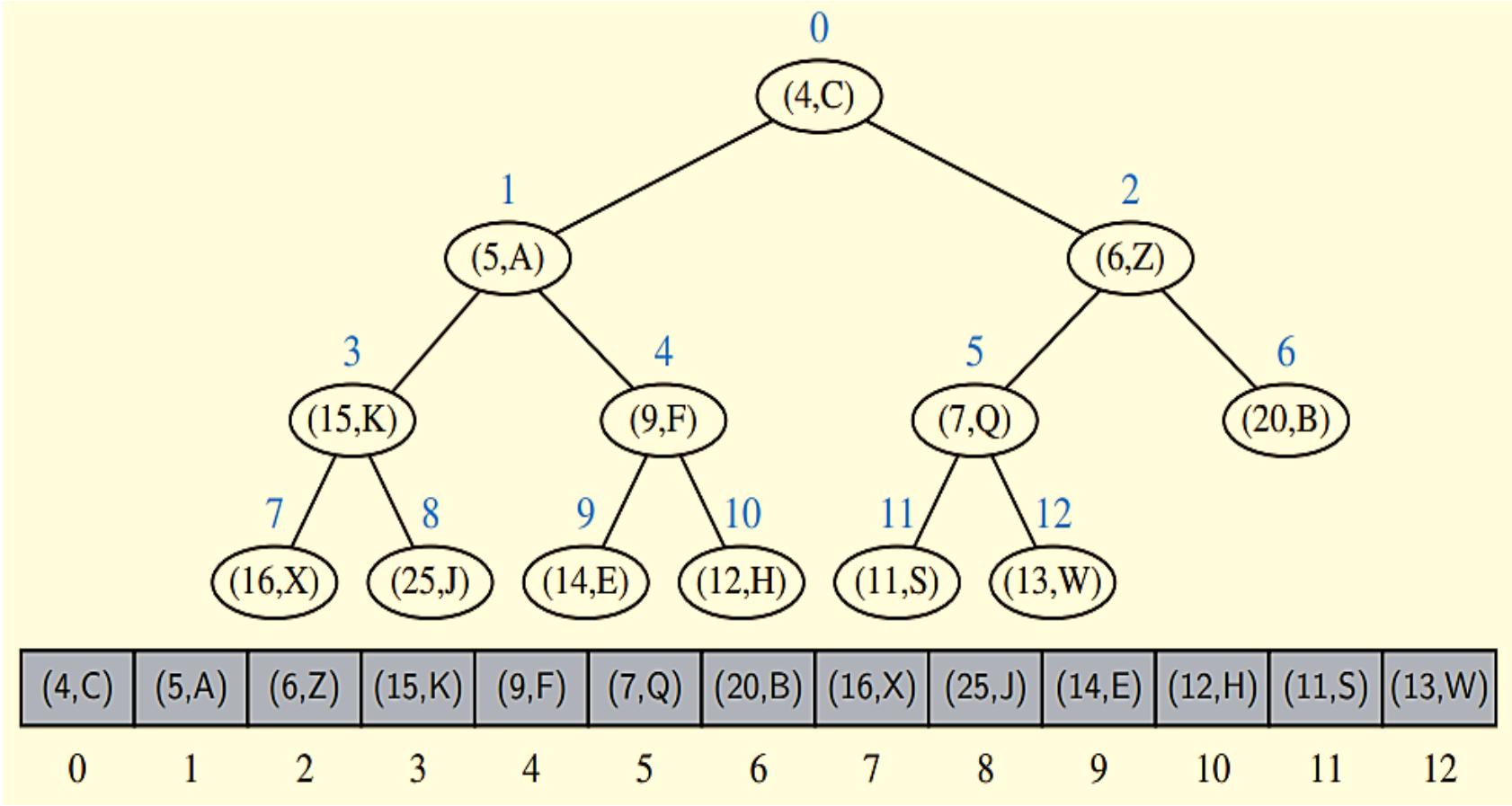


Implementing a Priority Queue with a Heap



Array-Based Representation of a Heap

- If p is the root, then **f(p) = 0**.
- If p is the left child of position q, then f(p) = **2 f(q) +1**.
- If p is the right child of position q, then f(p) = **2 f(q) +2**.



Analysis of a Heap-Based Priority Queue

- Each of the priority queue ADT methods can be performed in O(1) or in O(logn) time, where n is the number of entries at the time the method is executed.
- The analysis of the running time of the methods is based on the following:
  - The heap T has n nodes, each storing a reference to a key-value entry.
  - The height of heap T is O(log n), since T is complete.
  - The min operation runs in O(1) because the root of the tree contains such an element.
  - Locating the last position of a heap, as required for insert and removeMin, can be performed in O(1) time for an array-based representation, or O(logn) time for a linked-tree representation.
  - In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T

Analysis of a Heap-Based Priority Queue

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

*amortized, if using dynamic array

Bottom-Up Heap Construction

- An initially empty heap with n successive calls to the insert operation will run in O(nlog n) time in the worst case.
- There is an alternative **bottom-up** construction method that runs in **O(n)** time.
- The heap is a complete binary tree with every level being full, n = $\mathbf{2^{h+1} - 1}$
- bottom-up heap construction consists of the following steps,

1. construct (n+ 1)/2 elementary heaps

2. form (n+1)/4 heaps, each storing 3 entries. The new entry is placed at the root and may have to be swapped with the entry stored at a child.

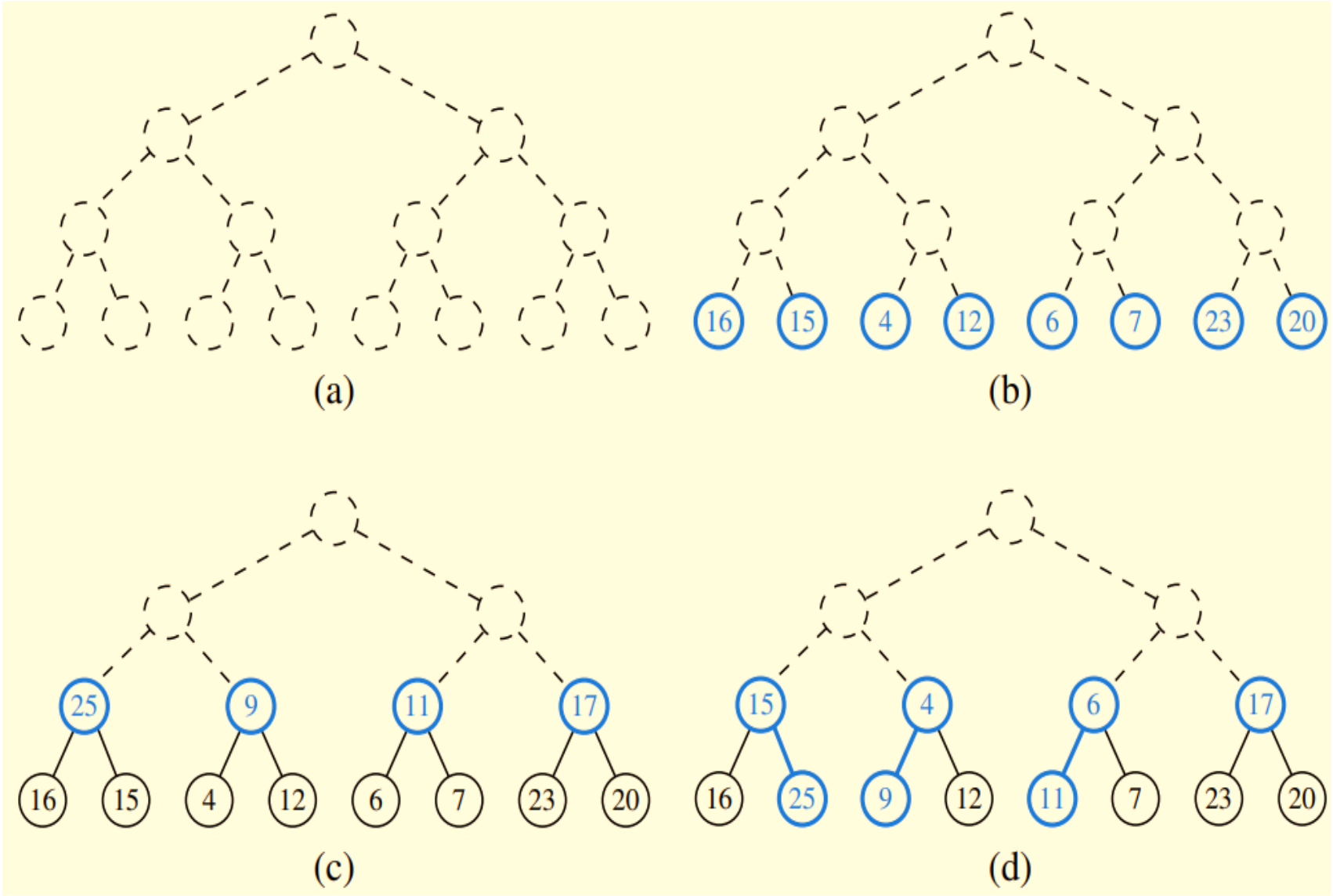
3. form (n+ 1)/8 heaps, each storing 7 entries. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling.

…….

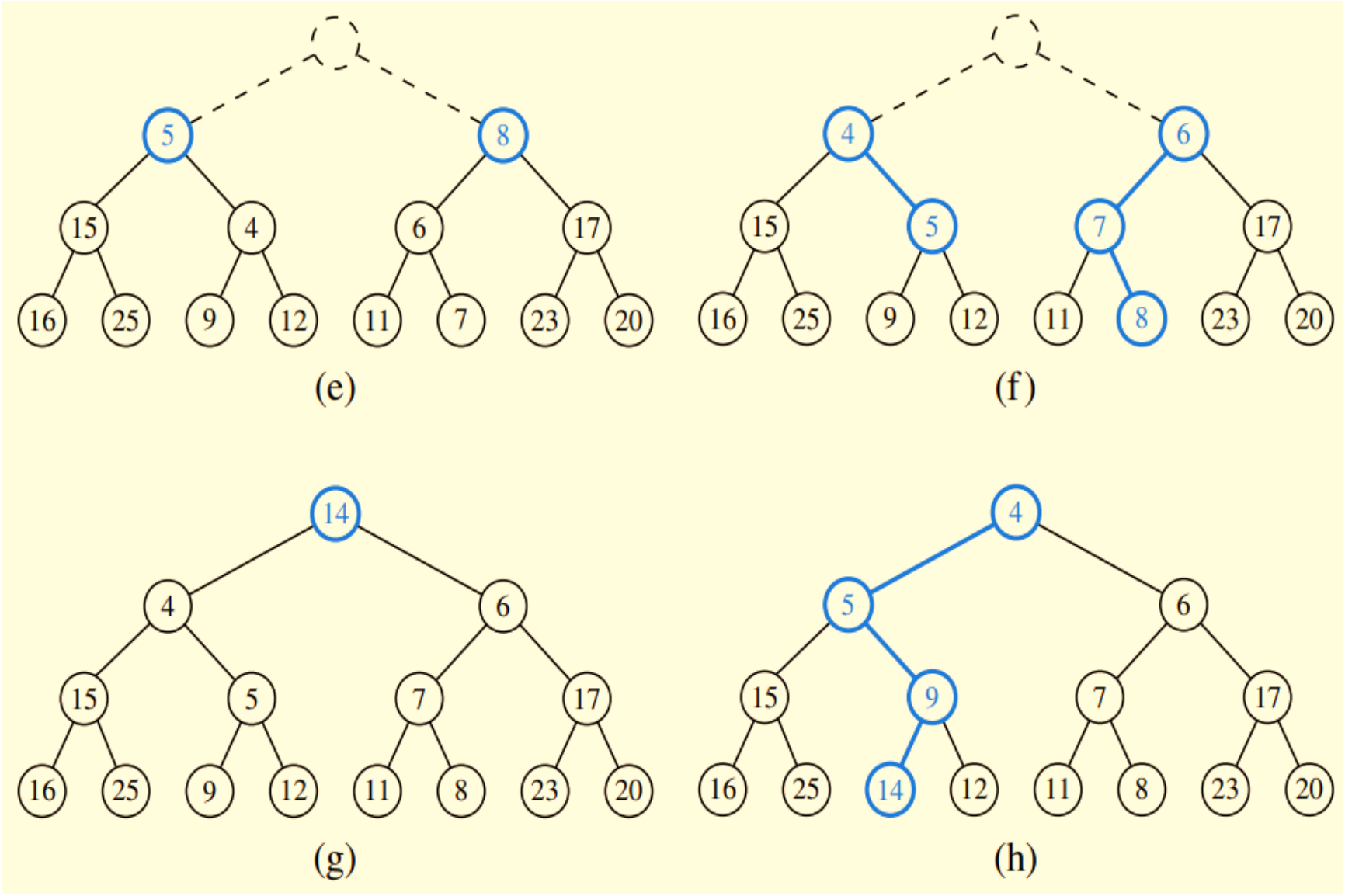
i. form (n+1)/ $\mathbf{2^i}$ heaps, each storing $\mathbf{2^{i-1}}$ entries

h+1. form the final heap, storing all the n entries

Bottom-Up Heap Construction



Bottom-Up Heap Construction



Java Heap Implementation

```
public class BottomUpHeapConstruction {
    public static void buildHeap(int[] array) {
        int n = array.length;
       for (int i = (n / 2) - 1; i >= 0; i--) {
            heapify(array, n, i);        }    }
private static void heapify(int[] array, int n, int i) {
        int smallest = i;    // Initialize largest as root
        int leftChild = 2 * i + 1;        int rightChild = 2 * i + 2;
             if (leftChild < n && array[leftChild] < array[smallest]) {
            smallest = leftChild;        }
             if (rightChild < n && array[rightChild] < array[smallest]) {
            smallest = rightChild;        }
        if (smallest != i) {
            swap(array, i, smallest);
            heapify(array, n, smallest);        }    }
    private static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;    }
```

Asymptotic Analysis of Bottom-Up Heap Construction

- Bottom-up heap construction is asymptotically faster than incrementally inserting n entries into an initially empty heap.
- we are performing a single down-heap operation at each position in the tree.
- more nodes are closer to the bottom of a tree than the top, the sum of the downward paths is linear

Proposition: Bottom-up construction of a heap with n entries takes O(n) time, assuming two keys can be compared in O(1) time.

Using java.util.PriorityQueue Class

- Java does include a class, **java.util.PriorityQueue**, which implements the **java.util.Queue interface**.
- java.util.PriorityQueue class processes its entries according to a priority.
- The “front” of the queue will always be a minimal element.
- The most notable difference between the java.util.PriorityQueue class and our own priority queue ADT is the model for managing keys and values.
- Whereas the public interface distinguishes between keys and values, the java.util.PriorityQueue class relies on a single element type. That element is effectively treated as a key.

Using java.util.PriorityQueue Class

- If a user wishes to insert distinct keys and values, the burden is on the user to define and insert appropriate composite objects, and to ensure that those objects can be compared based on their keys.
- The java.util.PriorityQueue class is implemented with a heap.
- It guarantees O(log n)-time performance for methods add and remove.
- Constant-time performance for accessors peek, size, and isEmpty.
- In addition, it provides a parameterized method, remove(e), that removes a specific element e from the priority queue.
- However, that method runs in O(n) time, performing a sequential search to locate the element within the heap.

Using java.util.PriorityQueue Class

Our Priority Queue ADT	java.util.PriorityQueue Class
insert(k, v)	add(**new** SimpleEntry(k, v))
min( )	peek( )
removeMin( )	remove( )
size( )	size( )
isEmpty( )	isEmpty( )