COMP225: Algorithms and Data Structures

Priority Queues, Heaps and Heapsort

Mark Dras
Mark.Dras@mq.edu.au
E6A380

Priority Queues, Heaps and Heapsort

- Priority queues
  - Java class
  - Possible implementations
- Heaps
  - Definition
  - Implementation
- Heapsort

Some Examples

- Hospital emergency room queue
  - Urgent cases handled first
- Standby queue at airport
  - Frequent flyers handled first?
- Operating system queue
  - Most / least CPU-intensive handled first?

Priority Queues

- Highest priority element comes off queue first
- Need functions to:
  - create an empty PQ
  - check if a PQ is empty
  - insert an element into a PQ
  - **retrieve the element with the highest priority and remove it from a PQ**

Java PQ Class

- In java.util.PriorityQueue
- Parametrisable by type: Integer, String, ...

Constructor Summary
`PriorityQueue()` Creates a `PriorityQueue` with the default initial capacity (11) that orders its elements according to their natural ordering.
`PriorityQueue(Collection<? extends E> c)` Creates a `PriorityQueue` containing the elements in the specified collection.
`PriorityQueue(int initialCapacity)` Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to their natural ordering.
`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)` Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.
`PriorityQueue(PriorityQueue<? extends E> c)` Creates a `PriorityQueue` containing the elements in the specified priority queue.
`PriorityQueue(SortedSet<? extends E> c)` Creates a `PriorityQueue` containing the elements in the specified sorted set.

Java PQ Class

Method Summary	
boolean	add(E e) Inserts the specified element into this priority queue.
void	clear() Removes all of the elements from this priority queue.
Comparator<? super E>	comparator() Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
boolean	contains(Object o) Returns true if this queue contains the specified element.
Iterator<E>	iterator() Returns an iterator over the elements in this queue.
boolean	offer(E e) Inserts the specified element into this priority queue.
E	peek() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	poll() Retrieves and removes the head of this queue, or returns null if this queue is empty.
boolean	remove(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	size() Returns the number of elements in this collection.
Object[]	toArray() Returns an array containing all of the elements in this queue.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Basic Java PQ Example

Default ordering in Java: smallest value is highest priority

```
package pqheap;

import java.util.PriorityQueue;

public class PQlec1 {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
        pq.add(5);
        pq.add(9);
        pq.add(3);
        System.out.println(pq.peek());
        System.out.println(pq.size());
        pq.poll();
        System.out.println(pq.peek());
    }
}
```

Output:
3
3
5

Basic Java PQ Example

Operation	Output	PQ
`pq.add(5)`	-	{5}
`pq.add(9)`	-	{5,9}
`pq.add(3)`	-	{3,5,9}
`println(pq.peek())`	3	{3,5,9}
`println(pq.size())`	3	{3,5,9}
`pq.poll()`	-	{5,9}
`println(pq.peek())`	5	{5,9}

Reversed Java PQ Example

- Priority queues (and other classes too) are parametrisable by a comparator
- The comparator defines how objects are ordered
  - Otherwise, the Java PQ uses a *natural order* where one exists

Reversed Java PQ Example

```
package pqheap;

import java.util.PriorityQueue;
import java.util.Comparator;

class RevComparator implements Comparator<Integer>
{
    @Override
    public int compare(Integer x, Integer y)
    {
     return y - x;
    }

}

public class PQlec1a {
     public static void main(String[] args) {
          Comparator<Integer> comp = new RevComparator();
          PriorityQueue<Integer> pq = new PriorityQueue<Integer>(10, comp);
          pq.add(5);
          pq.add(9);
          pq.add(3);
          System.out.println(pq.peek());
          System.out.println(pq.size());
          pq.poll();
          System.out.println(pq.peek());
     }
}
```

comparator declared

two parameters

Output:
9
3
5

Reversed Java PQ Example

Operation	Output	PQ
`pq.add(5)`	-	{5}
`pq.add(9)`	-	{9,5}
`pq.add(3)`	-	{9,5,3}
`println(pq.peek())`	9	{9,5,3}
`println(pq.size())`	3	{9,5,3}
`pq.poll()`	-	{5,3}
`println(pq.peek())`	5	{5,3}

Java PQ: Other Types

```java
package pqheap;

import java.util.PriorityQueue;
import java.util.Comparator;

class Item {
    private int key;
    private String name;
    public Item(int k, String s) {
        key = k;
        name = s;
    }
    int getKey() {
        return key;
    }
    String getName() {
        return name;
    }
}
```

Java PQ: Other Types

```
class ItemComparator implements Comparator<Item>
{
    @Override
    public int compare(Item x, Item y)
    {
     return x.getKey() - y.getKey();
    }

}

public class PQlec2 {
     public static void main(String[] args) {
          Comparator<Item> comp = new ItemComparator();
          PriorityQueue<Item> pq = new PriorityQueue<Item>(10, comp);
          Item e = new Item(3, "Mark");
          pq.add(e);
          e = new Item(4, "Lulu");
          pq.add(e);
          e = new Item(1, "Tim");
          pq.add(e);
          while (pq.size() > 0)
           {
            Item x = pq.poll();
             System.out.println(x.getKey() + " " + x.getName());
           }
     }
}
```

Output:
1 Tim
3 Mark
4 Lulu

Java PQ: Other Types

Operation	Output	PQ
`pq.add(Item(3, "Mark"))`	-	`{(3, "Mark")}`
`pq.add(Item(4, "Lulu"))`	-	`{(3, "Mark"),(4, "Lulu")}`
`pq.add(Item(1, "Tim"))`	-	`{(1, "Tim"),(3, "Mark"),(4, "Lulu")}`

Java PQ: Other Types Exercise

```java
public static void exercise() {
    Comparator<Item> comp = new ItemComparator();
    PriorityQueue<Item> pq = new PriorityQueue<Item>(10, comp);
    Item e = new Item(4, "A");
    pq.add(e);
    e = new Item(3, "C");
    pq.add(e);
    Item x = pq.poll();
    e = new Item(7, "B");
    pq.add(e);
    e = new Item(6, "F");
    pq.add(e);
    System.out.println(x.getKey() + " " + x.getName());
    System.out.println(pq.size());
    x = pq.poll();
    if (pq.isEmpty())
        System.out.println("empty");
    else {
        x = pq.poll();
        System.out.println(x.getKey() + " " + x.getName());
    }
}
```

Java PQ: Other Types

- An alternative comparator: lexicographic ordering on string component

```
class ItemComparatorStr implements Comparator<Item>
{
    @Override
    public int compare(Item x, Item y)
    {
     return x.getName().compareTo(y.getName());
    }

}

public class PQlec2a {
      public static void main(String[] args) {
            Comparator<Item> comp = new ItemComparatorStr();
            PriorityQueue<Item> pq = new PriorityQueue<Item>(10, comp);
            Item e = new Item(3, "Mark");
            pq.add(e);
            e = new Item(4, "Lulu");
            pq.add(e);
            e = new Item(1, "Tim");
            pq.add(e);
            while (pq.size() > 0)
            {
                  Item x = pq.poll();
                        System.out.println(x.getKey() + " " + x.getName());
            }
      }
}
```

only difference

Output:
4 Lulu
3 Mark
1 Tim

Java PQ: Other Types

```java
class Dict3 {
     private ArrayList<PriorityQueue<String>> dict;
               // an array of priority queues, one per letter of the alphabet
     public Dict3(int maxSize) {
          dict = new ArrayList<PriorityQueue<String>>(maxSize);
               // initialise overall ArrayList
          for (int i = 0; i < maxSize; i++) {
                    // initialise each priority queue
               PriorityQueue<String> pq = new PriorityQueue<String>();
               dict.add(pq);
          }
     }
     public void add(String s) {
          int indx = (int) s.charAt(0) - (int) 'a';
               // converts the first letter of s into its position in the alphabet
          dict.get(indx).add(s);
     }

     // more functions ...
}

public class PQexDict3 {
     public static void main(String[] args) {
          String words[] = {"in", "the", "second", "century", "of", "the",
                 "christian",  "era", "the", "empire", "of", "rome",
                 "comprehended", "the", "fairest", "part", "of",
                 "the", "earth"};

          Dict3 myDict = new Dict3(26);

          for (int i = 0; i < words.length; i++)
               myDict.add(words[i]);

          // ...
     }
}
```

Priority Queues, Heaps and Heapsort

- Priority queues
  - Java class
  - Possible implementations
- Heaps
  - Definition
  - Implementation
- Heapsort

Arrays

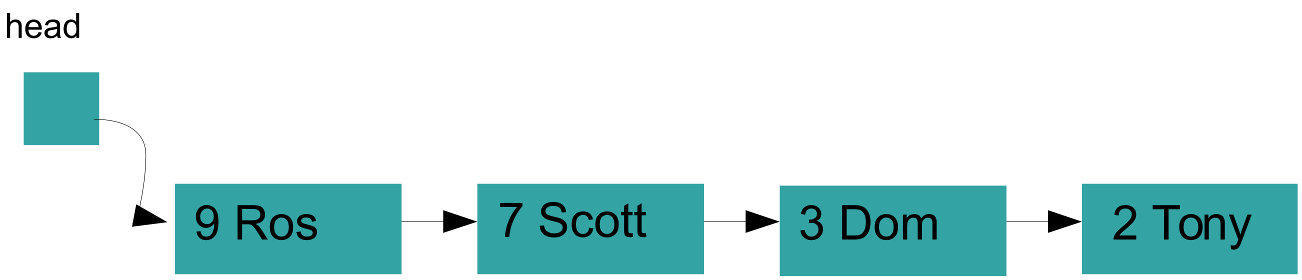
- Ordered: highest priority at end
  - Note from now on we're using larger numbers as highest priority

Index	0	1	2	3	4
(Key, Val)	2 Tony	3 Dom	7 Scott	9 Ros	-

- Deletion: O(1)
- Insertion: O(n)

Linked List

- Ordered: highest priority at head



- Deletion: O(1)
- Insertion: O(n)

Binary Search Tree

- Insertion and deletion:
  - O(log n) if tree isn't too unbalanced
  - O(n) in worst case
- Has more structure than we need
  - BSTs are totally sorted
  - We only need to access element with highest priority

Priority Queues, Heaps and Heapsort

- Priority queues
  - Java class
  - Possible implementations
- Heaps
  - Definition
  - Implementation
- Heapsort

Heaps: Definition

- A particular kind of binary tree, called a **heap**, has two properties:
  - The value of each node is greater than or equal to the values stored in each of its children
  - The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions: this is called a **complete binary tree**
- These two properties define a **max heap**
- If “greater” in the first property is replaced with “less,” then the definition specifies a **min heap**

Heaps: Definition

A more formal, recursive definition:

A (max)heap is a complete binary tree
1. that is empty; or
2a. whose root contains a key that is >= to the key in each of its children, and
2b. where the subtrees are heaps.

Heaps: Examples

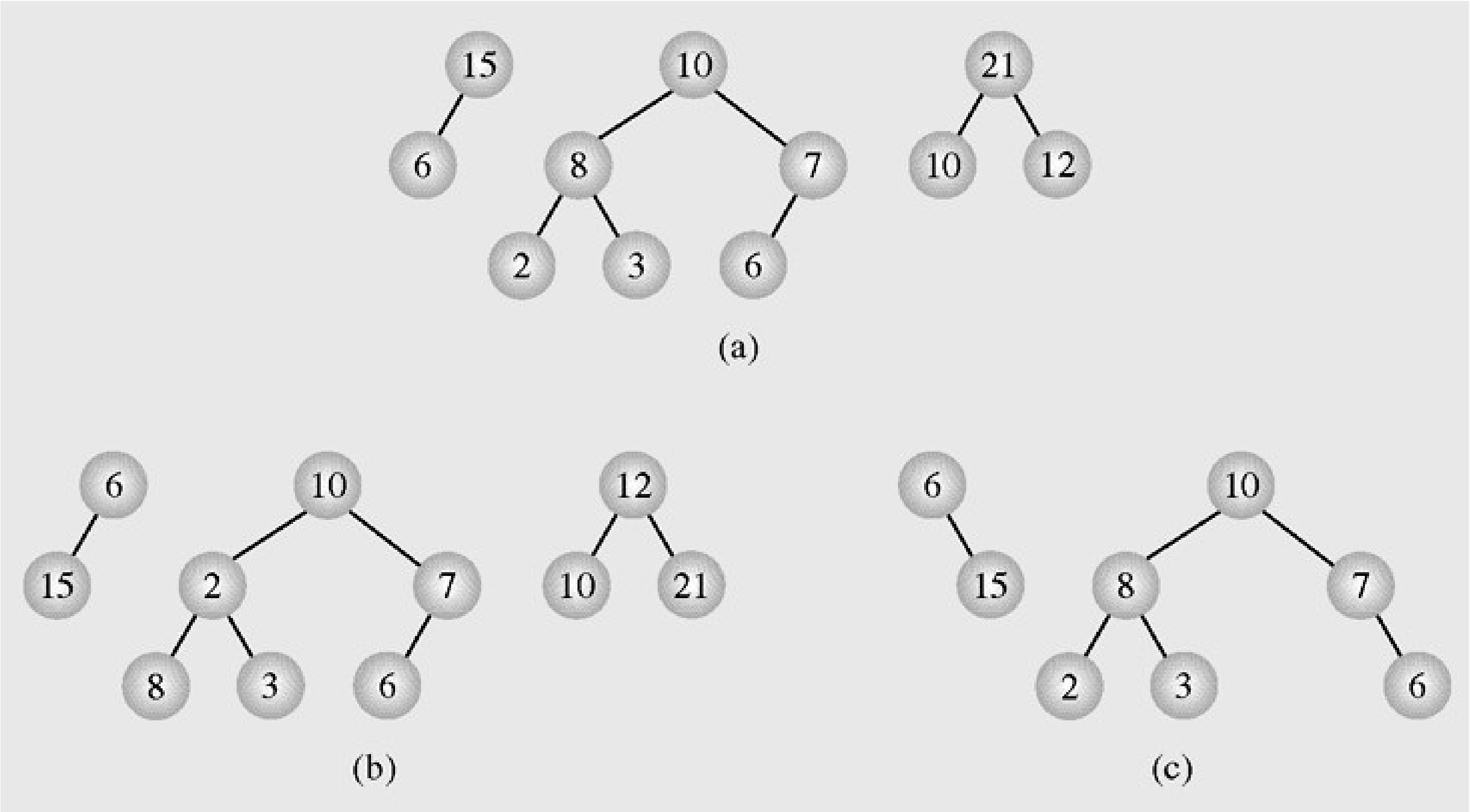


Figure 6-51 Examples of (a) heaps and (b–c) nonheaps

Heaps: Examples

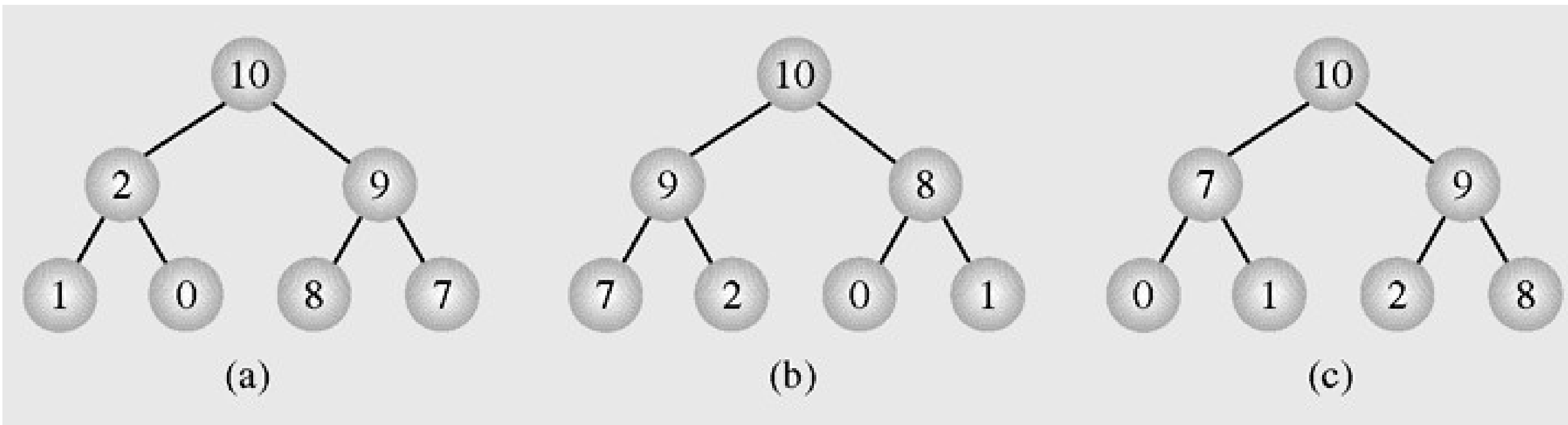


Figure 6-53 Different heaps constructed with the same elements

Heaps: Exercise

- Make a heap out of the integers 12, 5, 9, 18, 11, 10, 16, 13

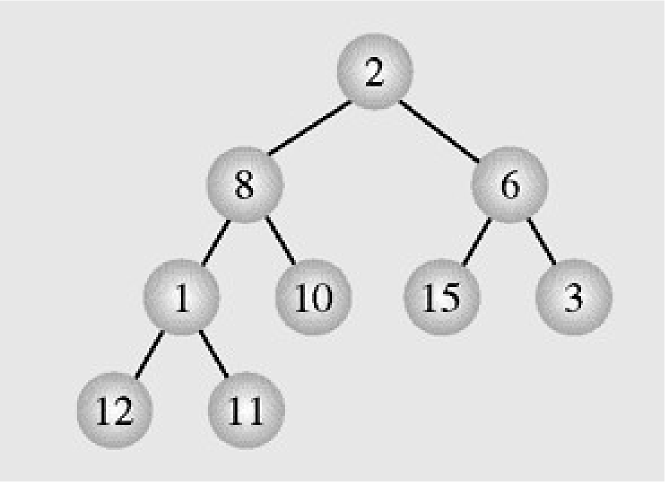
Priority Queues, Heaps and Heapsort

- Priority queues
  - Java class
  - Possible implementations
- Heaps
  - Definition
  - Implementation
- Heapsort

Heaps as Arrays

- Because the binary tree is complete, it can be represented compactly as an array
- In addition:
  - the left child of A[i] is stored at A[2*i+1] (if it exists)
  - the right child of A[i] is stored at A[2*i+2] (if it exists)
  - the parent of A[i] is stored at A[(i-1)/2] (if not the root)
- Note: heaps are **stored** as arrays, and **visualised** as binary trees

Trees as Arrays



Index	Value
0	2
1	8
2	6
3	1
4	10
5	15
6	3
7	12
8	11

Figure 6-52 The array [2 8 6 1 10 15 3 12 11] seen as a tree

Heap Operations

- Need functions to:
  - create a new heap;
  - check if a heap is empty;
  - insert an element into a heap;
  - delete the largest element in the heap.

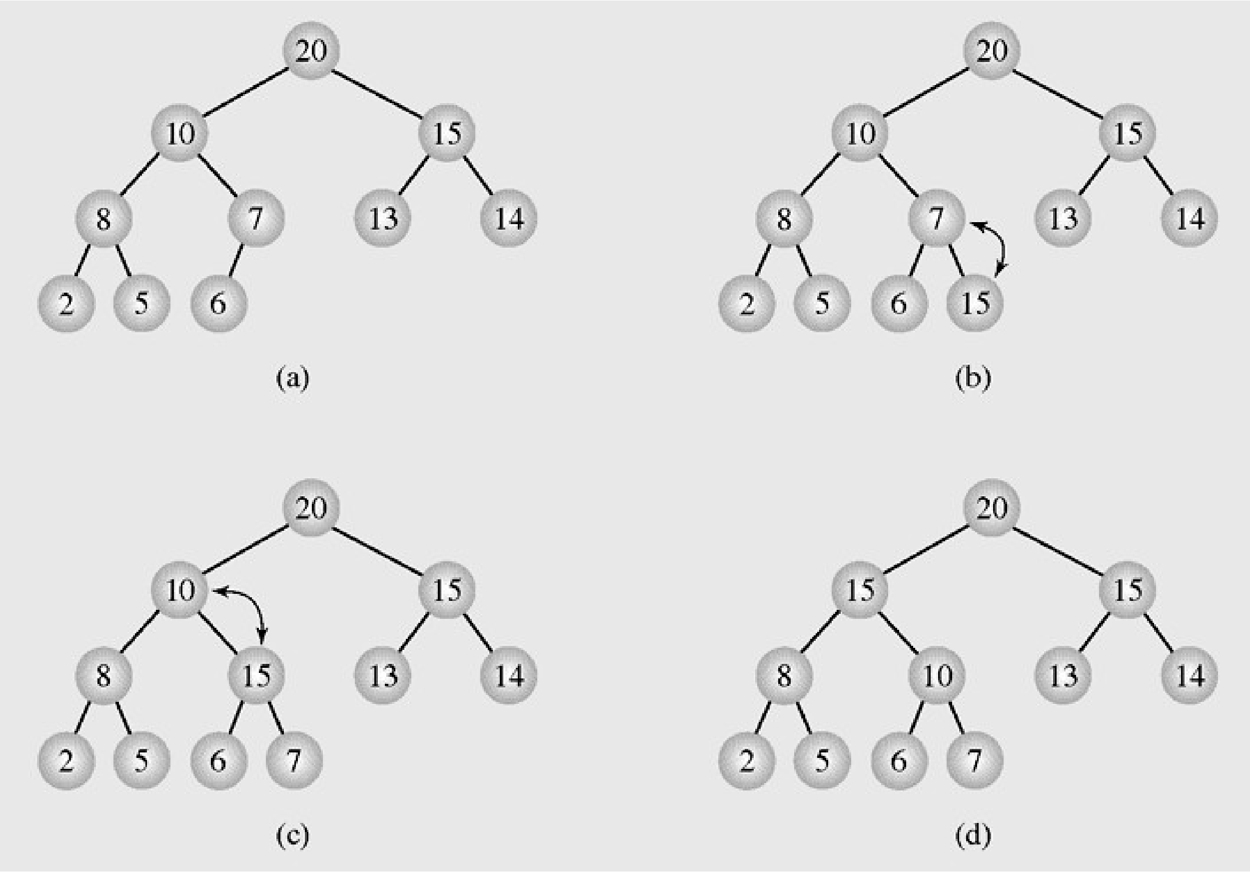
Heaps: Java Prelims

```
public class Heap {
    private static int maxHeap = 20;
    private static int[] items;
    private static int size;

    public Heap() {
        items = new int[maxHeap];
        size = 0;
    }

    private static void swap (int i, int j) {
    // swaps elements items[i] and items[j]
    // note that Java can't do a regular swap()
        int temp = items[i];
        items[i] = items[j];
        items[j] = temp;
    }

    public int heapTop () {
        return items[0];
    }

    // other operations ...
}
```

Heap Operations: Insert

- General strategy:
  - insert new item into the bottom of the tree
  - new item trickles up to an appropriate spot in the tree

Heap Operations: Insert



0	1	2	3	4	5	6	7	8	9	10
20	10	15	8	7	13	14	2	5	6	-

0	1	2	3	4	5	6	7	8	9	**10**
20	10	15	8	7	13	14	2	5	6	**15**

0	1	2	3	**4**	5	6	7	8	9	**10**
20	10	15	8	**15**	13	14	2	5	6	**7**

0	**1**	2	3	**4**	5	6	7	8	9	10
20	**15**	15	8	**10**	13	14	2	5	6	7

Figure 6-54 Enqueuing an element to a heap

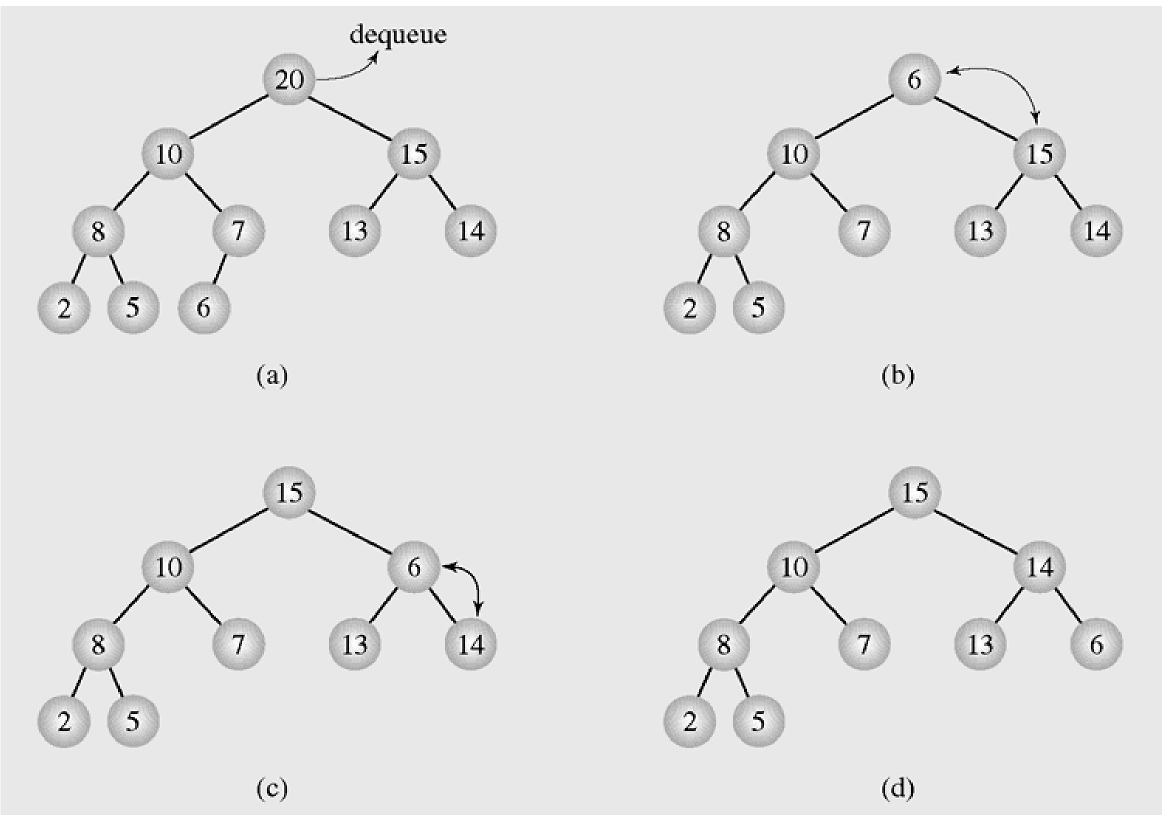
Heap Operations: Insert

```
public void heapInsert (int newItem) {
    if (size < maxHeap) {
    // place the new item at the end of the heap
        items[size] = newItem;

        // trickle new item up to its proper position
        int place = size;
        int parent = (place - 1)/2;
        while ( (parent >= 0) &&  (items[place] > items[parent] ))
        {  // swap items[place] and items[parent]
              swap(place, parent);
               // move up to parent
              place = parent;
          parent = (place - 1)/2;
        } // end while

        ++size;
    }
} // end heapInsert
```

Heap Operations: Delete

- General strategy
  - delete the root
  - do this by first swapping root with last element (why?)
  - the new structure will have children which are heaps, but which is not itself a heap: it's a **semi-heap**
  - use heapRebuild to convert semi-heap back to heap

Heap Operations: Delete



0	1	2	3	4	5	6	7	8	9
20	10	15	8	7	13	14	2	5	6

0	1	2	3	4	5	6	7	8	**9**
6	10	15	8	7	13	14	2	5	**-**

0	1	**2**	3	4	5	6	7	8	9
15	10	**6**	8	7	13	14	2	5	-

0	1	**2**	3	4	5	**6**	7	8	9
15	10	**14**	8	7	13	**6**	2	5	-

Figure 6-55 Dequeuing an element from a heap

Heap Operations: Delete

```
public void heapDelete()
// Method: Swaps the last item in the heap with the
root
// and trickles it down to its proper position.
{
    if (size > 0 )
    {
        items[0] = items[--size];
        heapRebuild(0);
    } // end if
} // end heapDelete
```

Heap Operations: Delete

```
private static void heapRebuild(int root)
{
   // if the root is not a leaf and the root's search key
   // is less than the larger of the search keys in the
   // root's children
   int child = 2 * root + 1; // index of root's left
                             // child, if any

   if ( child < size )
   {  // root is not a leaf, so it has a left child at child
      int rightChild = child + 1; // index of right child,
                                  // if any

      // if root has a right child, find smaller child
      if ( (rightChild < size) &&
           (items[rightChild] > items[child]) )
         child = rightChild; // index of smaller child

      // if the root's value is larger than the
      // value in the larger child, swap values
      if ( items[root] < items[child] )
      {
         swap(root, child);
         // transform the new subtree into a heap
         heapRebuild(child);
      } // end if
   } // end if

   // if root is a leaf, do nothing
}
```

Heap Operations: Exercise

- What's the output of the function below?

```
public static void main (String[] args) {
    Heap h = new Heap();
    h.heapInsert(10);
    h.heapInsert(12);
    h.heapInsert(5);
    h.heapInsert(14);
    h.heapInsert(15);
    h.heapInsert(9);

    h.heapPrint();  // prints heap as array
    h.heapDelete();
    h.heapDelete();
    System.out.println(h.heapTop());
    h.heapDelete();
    h.heapDelete();
    h.heapPrint();
}
```

Heap Operations: Properties

- Because a heap is always guaranteed to be a complete binary tree:
    - insert is O(log n)
    - delete is O(log n)

Heaps and Priority Queues

- Heaps provide all the functionality necessary for a PQ; e.g.
  - peek() returns heapTop()
  - poll() returns heapTop() and calls heapDelete()
  - add() calls heapInsert()
- A PQ isn't a *type* of heap (i.e. not inheritance); a PQ would be represented as a heap.

Priority Queues, Heaps and Heapsort

- Priority queues
  - Java class
  - Possible implementations
- Heaps
  - Definition
  - Implementation
- Heapsort

Heapsort

- Two steps:
  1. transform array into heap
  2. repeatedly move root (i.e. biggest element) to sorted region and rebuild remaining heap
- Can transform array into heap in two ways:
  1. repeated insertion into initially empty array, top down
  2. in place, bottom up

Organizing Arrays as Heaps

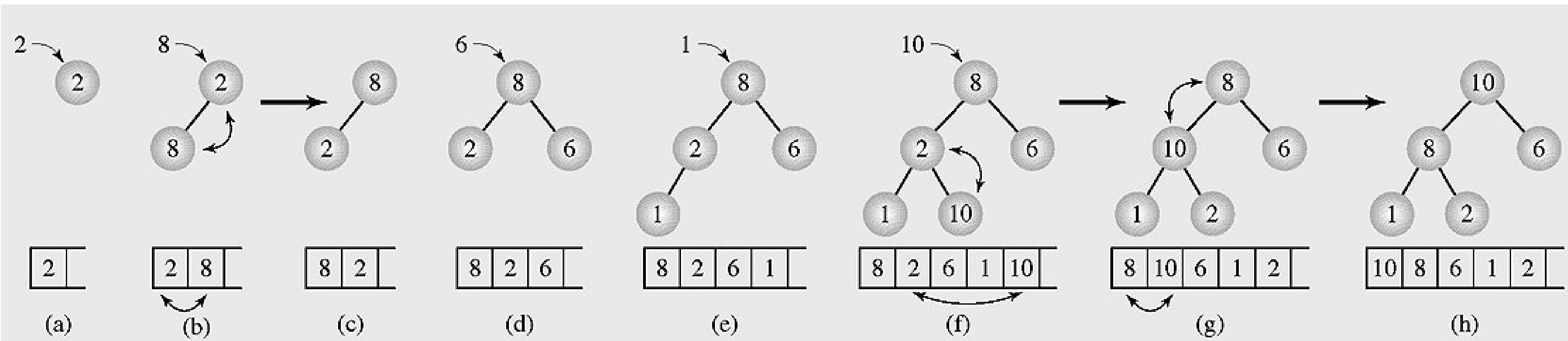


Figure 6-57 Organizing an array as a heap with a top-down method

Organizing Arrays as Heaps

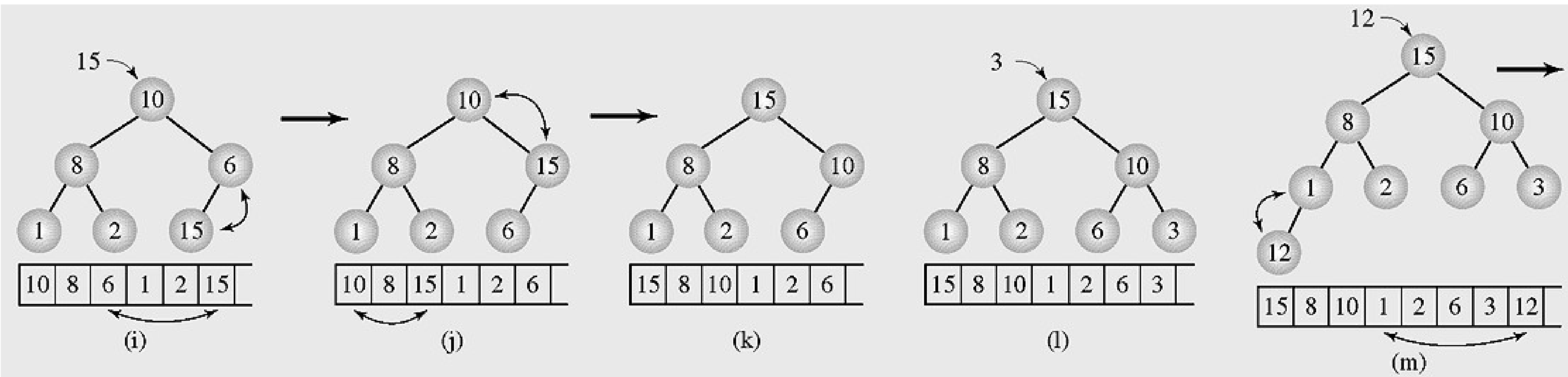


Figure 6-57 Organizing an array as a heap with a top-down method (continued)

Organizing Arrays as Heaps

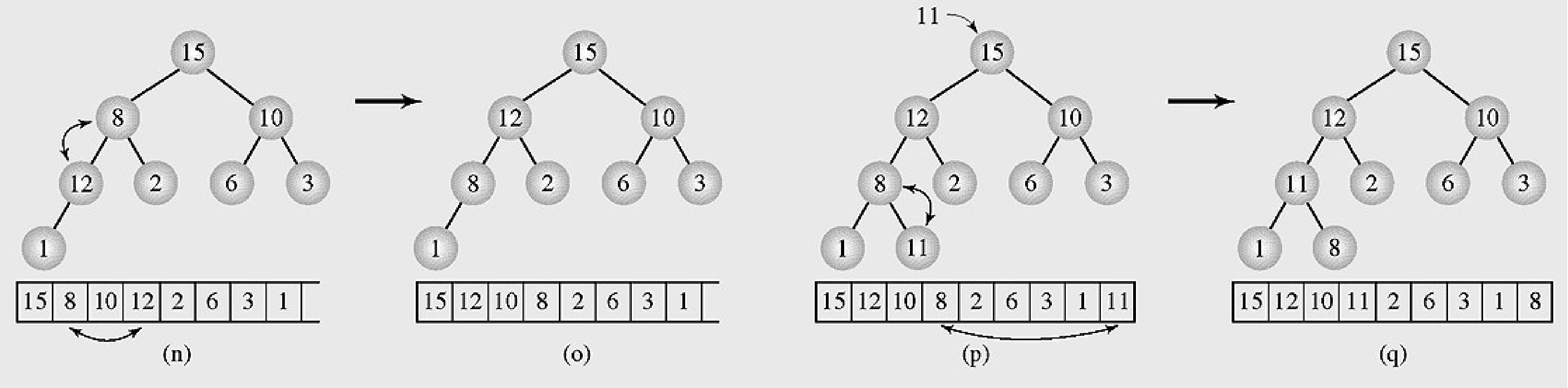


Figure 6-57 Organizing an array as a heap with a top-down method (continued)

Organising Arrays as Heaps: Bottom-up

- Imagine array as complete binary tree
- Transform tree into heap by repeatedly calling heapRebuild()
  - Note that leaves are 1-node heaps
  - Can start heapRebuild() at level h-1, which are semi-heaps
  - Can move up tree, knowing that each higher level will be a semi-heap after its children have been rebuilt as heaps

Organizing Arrays as Heaps

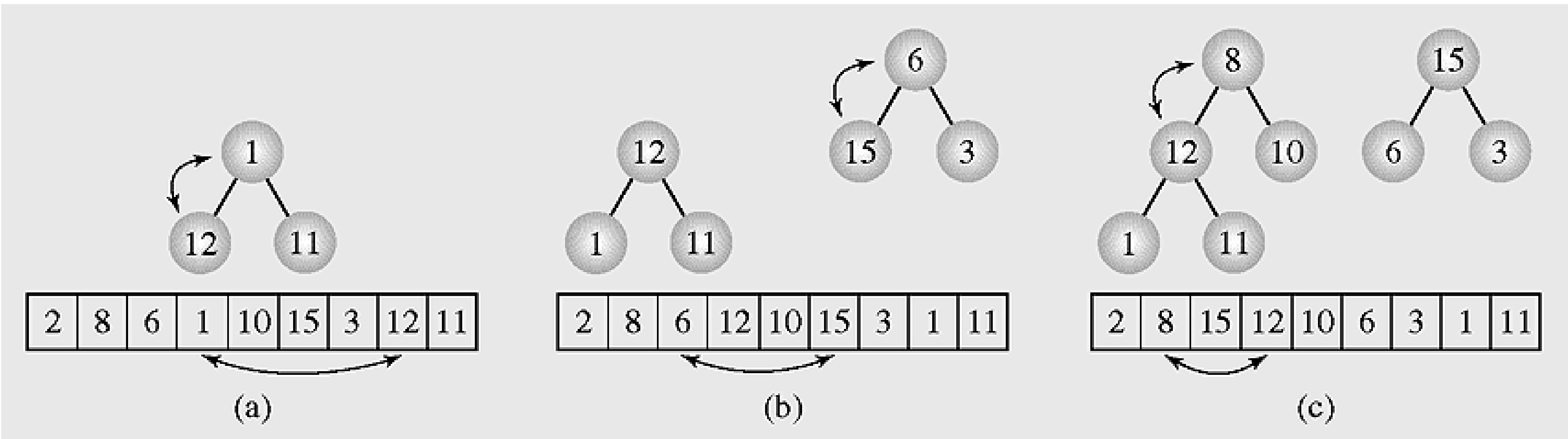


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method

Organizing Arrays as Heaps

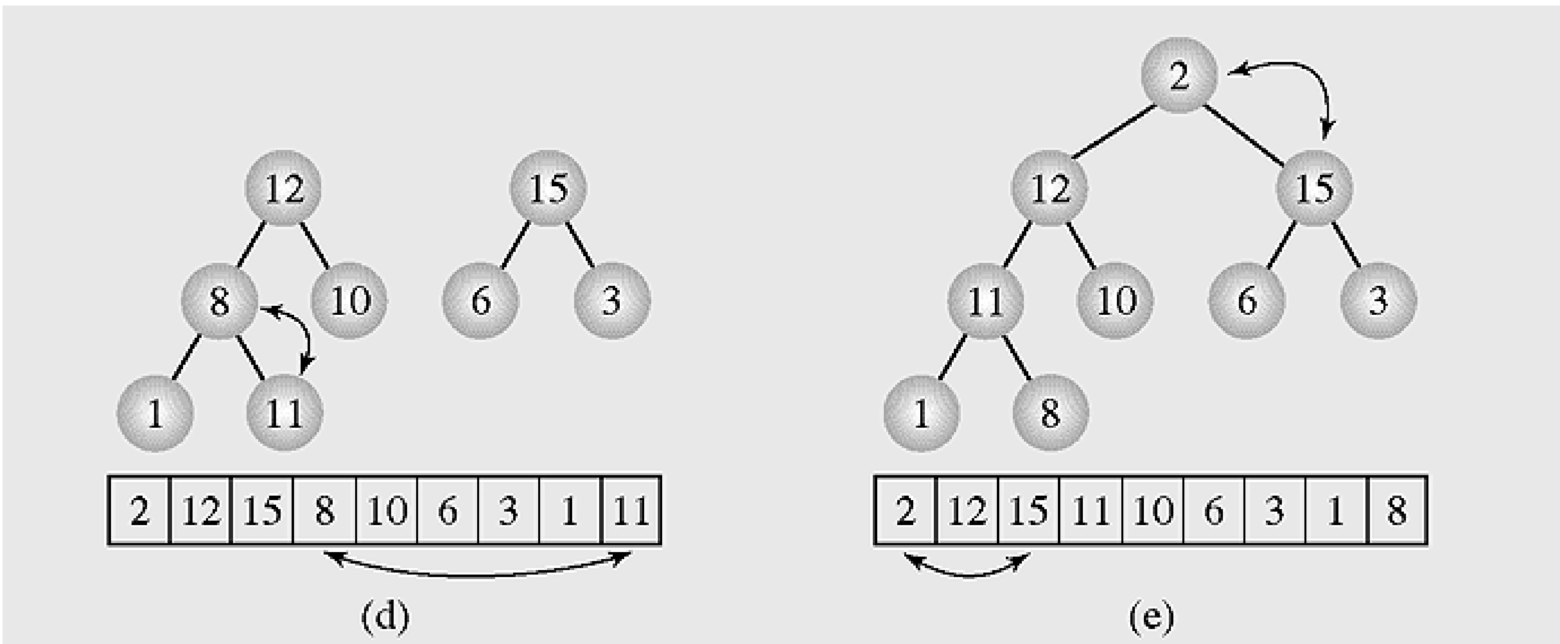


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method (continued)

Organizing Arrays as Heaps

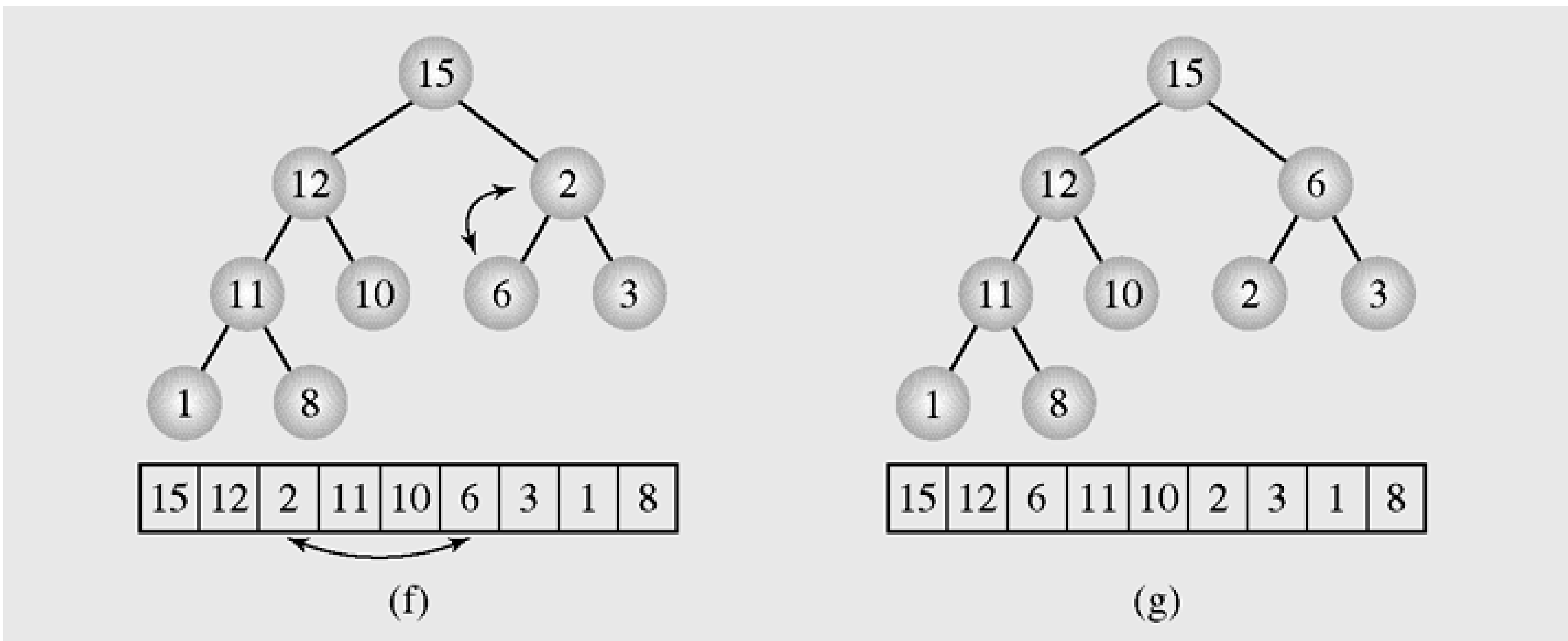


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method (continued)

Heapsort Step 2

- Array is partitioned into
  - A[0..last] (unsorted heap)
  - A[last+1..n-1] (sorted)
- Last is initially n-1
- At each step, move heap root to sorted region
  - Swap root with last element of heap (in the same way as when deleting the root)
    - this produces a semi-heap in the non-sorted region
  - Extend sorted region (by assigning last = last - 1)
    - the former heap root is now in the sorted region
  - Carry out a heapRebuild on the semi-heap

Heapsort Step 2

- Invariant
  - After step k, the sorted region contains the k largest values in A, and they are in ascending order
  - The items in the heap region form a heap

Heapsort

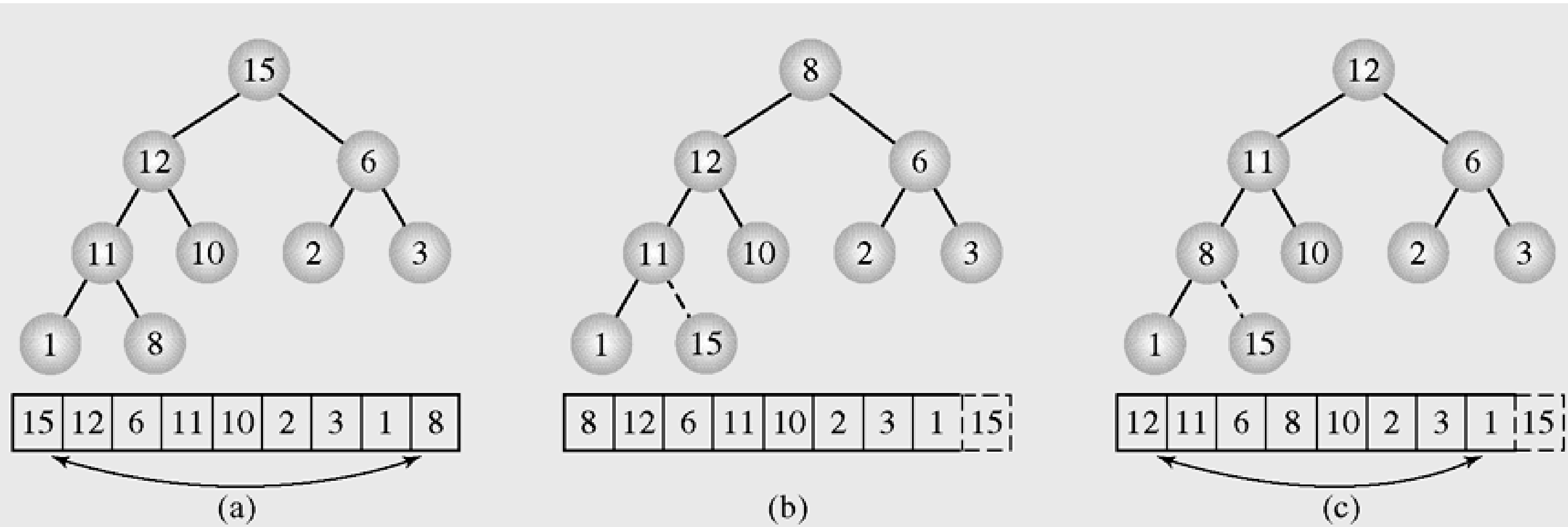


Figure 9-10 Execution of heap sort on the array [15 12 6 11 10 2 3 1 8], which is the heap constructed in Figure 9.9 (also Figure 6.58)

Heapsort

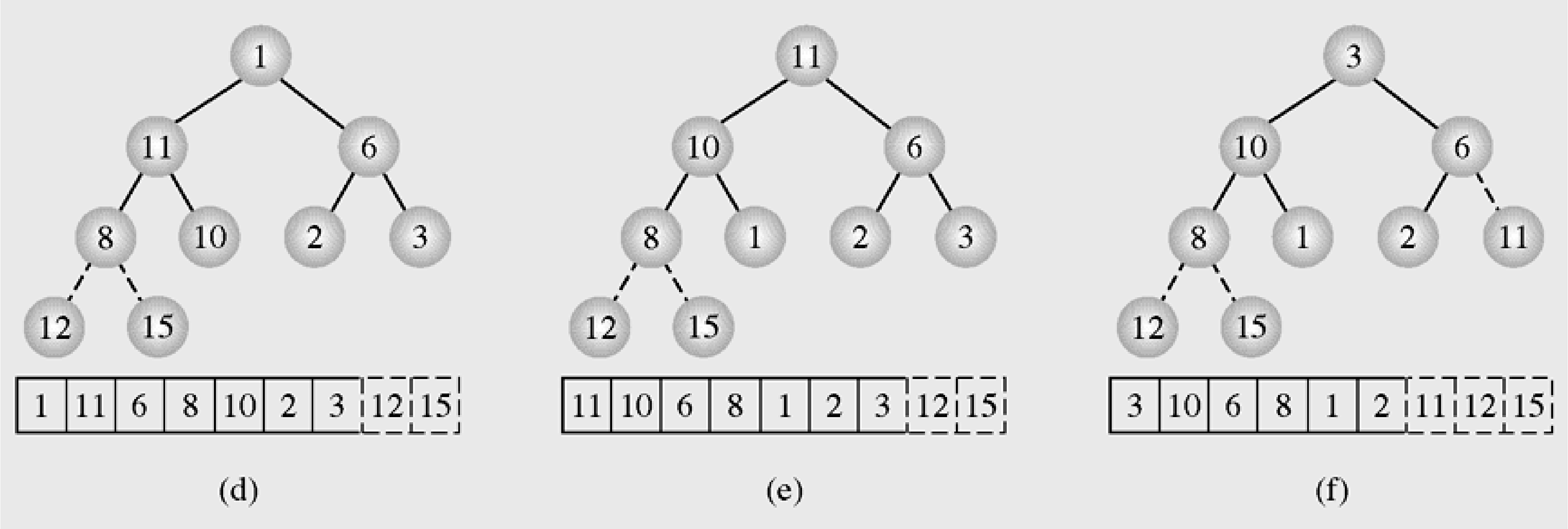


Figure 9-10 Execution of heap sort on the array [15 12 6 11 10 2 3 1 8], which is the heap constructed in Figure 9.9 (continued)

Heapsort

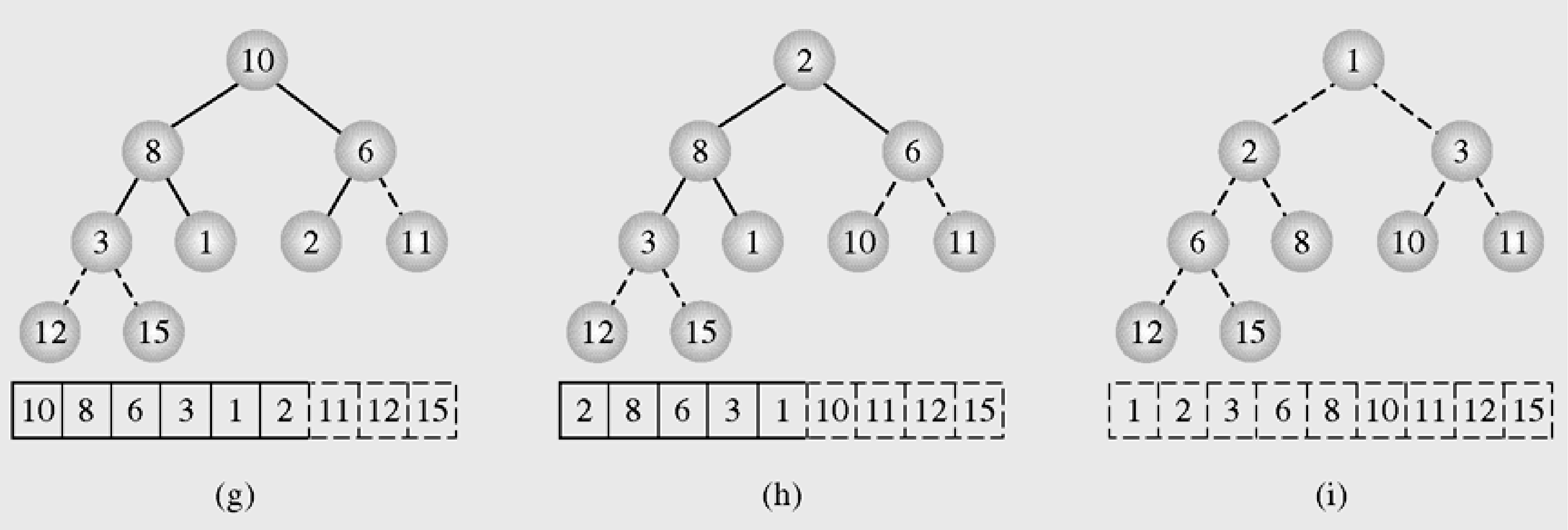


Figure 9-10 Execution of heap sort on the array [15 12 6 11 10 2 3 1 8], which is the heap constructed in Figure 9.9 (continued)

Heapsort Exercise

- Carry out the first two steps of heapsort on the following array
  - Note that it's already a heap

0	1	2	3	4	5
15	14	9	10	12	5

Comparison of Sorts

- Heapsort
  - first step: O(n log n)
    - either inserting n elements, with each insert O(log n), or
    - carrying out heapRebuild for n/2 elements, with each heapRebuild O(n log n)
  - second step: O(n log n)
    - for each of the n elements in the array, carry out a delete, with delete O(log n)
  - overall: O(n log n)

Comparison of Sorts

- Mergesort
  - Like Heapsort is O(n log n)
  - Heapsort does not require additional array space (if using second approach to constructing original heap)
- Quicksort
  - Average case of O(n log n) is comparable to Heapsort
  - However, Quicksort has worst case of $O(n^2)$
  - Regardless of this, Quicksort often preferred in practice