CHAPTER 5

C++ Heaps

LESSON 5.1

Heap

The two implementations of the PriorityQueueSort scheme presented in the previous section suggest a possible way of improving the running time for priority-queue sorting. One algorithm (selection-sort) achieves a fast running time for the first phase, but has a slow second phase, whereas the other algorithm (insertion-sort) has a slow first phase, but achieves a fast running time for the second phase. If we could somehow balance the running times of the two phases, we might be able to significantly speed up the overall running time for sorting. This approach is, in fact, exactly what we can achieve using the priority-queue implementation discussed in this section.

An efficient realization of a priority queue uses a data structure called a heap. This data structure allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations discussed in previous lessons. The fundamental way the heap achieves this improvement is to abandon the idea of storing elements and keys in a list and take the approach of storing elements and keys in a binary tree instead.

Heap Data Structure

A heap (see Figure 5.1.1) is a binary tree T that stores a collection of elements with their associated keys at its nodes and that satisfies two additional properties: a relational property, defined in terms of the way keys are stored in T, and a structural property, defined in terms of the nodes of T itself. We assume that a total order relation on the keys is given, for example, by a comparator.

The relational property of T, defined in terms of the way keys are stored, is the following:

Heap-Order Property: In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to an external node of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. This is the most important key and is informally said to be "at the top of the heap," hence, the name "heap" for the data structure.

By the way, the heap data structure defined here has nothing to do with the freestore memory heap used in the run-time environment supporting

programming languages like C++.

You might wonder why heaps are defined with the smallest key at the top, rather than the largest. The distinction is arbitrary. (This is evidenced by the fact that the STL priority queue does exactly the opposite.) Recall that a comparator implements the less-than operator between two keys. Suppose that we had instead defined our comparator to indicate the opposite of the standard total order relation between keys (so that, for example, isLess(x,y) would return true if x were greater than y). Then the root of the resulting heap would store the largest key. This versatility comes essentially for free from our use of the comparator pattern. By Caution defining the minimum key in terms of the comparator, the "minimum" key with a "reverse" comparator is in fact the largest. Thus, without loss of generality, we assume that we are always interested in the minimum key, which is always at the root of the heap.

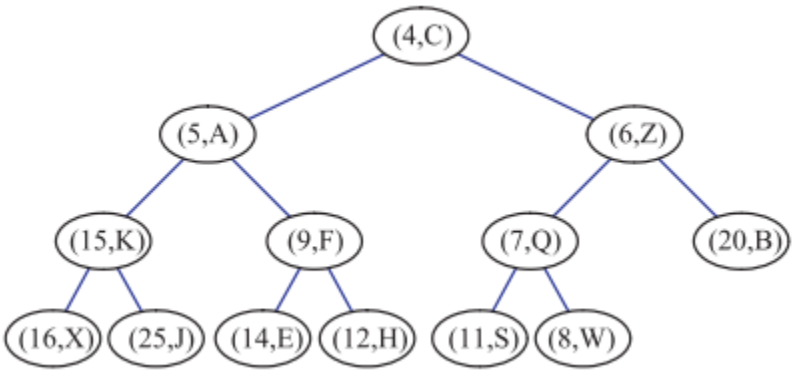

Figure 5.1.1 Example of a heap storing 13 elements. Each element is a key-value pair of the form (k,v). The heap is ordered based on the key value, k, of each element.

For the sake of efficiency, which becomes clear later, we want the heap T to have as small a height as possible. We enforce this desire by insisting that the heap T satisfy an additional structural property, it must be complete. Before we define this structural property, we need some definitions. We recall from Section of binary trees, that level i of a binary tree T is the set of nodes of T that have depth i. Given nodes v and w on the same level of T, we say that v is to the left of w if v is encountered before w in an inorder traversal of T. That is, there is a node u of T such that v is in the left subtree of u and w is in the right subtree of u. For example, in the binary tree of Figure 5.1.1, the node storing entry (15,K) is to the left of the node storing entry (7,Q). In a standard drawing of a binary tree, the "to the left of" relation is visualized by the relative horizontal placement of the nodes.

Complete Binary Tree Property: A heap T with height h is a complete binary tree, that is, levels 0,1,2,...,h−1 of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h - 1$) and the nodes at level h fill this level from left to right.

The Height of a Heap

Let h denote the height of T. Another way of defining the last node of T is that it is the node on level h such that all the other nodes of level h are to the left of it. Insisting that T be complete also has an important consequence as shown in Proposition 5.1.1.

Proposition 5.1.1 A heap T storing n entries has height

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

Justification: From the fact that T is complete, we know that there are 2^i nodes in level, i for $0 \le i \le h - 1$, and level h has at least 1 node. Thus, the number of nodes of T is at least

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

$$= 2^h$$

Level h has at most 2h nodes, and thus the number of nodes of T is at most

$$(1+2+4+\cdots+2^{h-1}) +2^h = 2^{h+1} -1.$$

Since the number of nodes is equal to the number n of entries, we obtain

$$2^h \le n$$

and

$$n \le 2^{h+1} -1.$$

Thus, by taking logarithms of both sides of these two inequalities, we see that

$$h \le \log n$$

and

$$\log(n+1)-1 \le h.$$

Since h is an integer, the two inequalities above imply that

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

Proposition 5.1.1 has an important consequence. It implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time. Therefore, let us turn to the problem of how to efficiently perform various priority queue functions using a heap.

Complete Binary Tree ADT

As an abstract data type, a complete binary tree T supports all the functions of the binary tree ADT, plus the following two functions:

- add(e): Add to T and return a new external node v storing element e, such that the resulting tree is a complete binary tree with last node v.
- remove(): Remove the last node of T and return its element.

By using only these update operations, the resulting tree is guaranteed to be a complete binary. As shown in Figure 5.1.2, there are essentially two cases for the effect of an add (and remove is similar).

- If the bottom level of T is not full, then add inserts a new node on the bottom level of T, immediately after the rightmost node of this level (that is, the last node); hence, T's height remains the same.
- If the bottom level is full, then add inserts a new node as the left child of the leftmost node of the bottom level of T; hence, T's height increases by one.

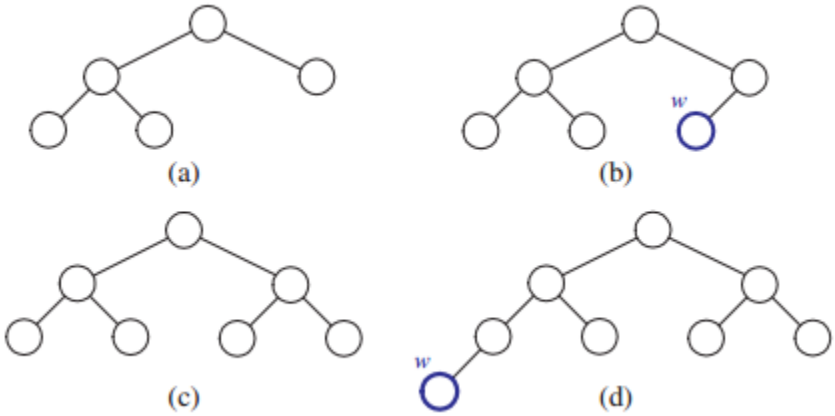

Figure 5.1.2 Examples of operations add and remove on a complete binary tree, where w denotes the node inserted by add or deleted by remove. The trees shown in (b) and (d) are the results of performing add operations on the trees in (a) and (c), respectively. Likewise, the trees shown in (a) and (c) are the results of performing remove operations on the trees in (b) and (d), respectively.

A Vector Representation of Complete Binary tree

The vector-based binary tree representation is especially suitable for a complete binary tree T. We recall that in this implementation, the nodes of T are stored in

a vector A such that node v in T is the element of A with index equal to the level number f(v) defined as follows:

- If v is the root of T, then f(v) = 1
- If v is the left child of node u, then f(v) = 2 f(u)
- If v is the right child of node u, then f(v) = 2 f(u) +1

With this implementation, the nodes of T have contiguous indices in the range [1,n] and the last node of T is always at index n, where n is the number of nodes of T. Figure 5.1.2 shows two examples illustrating this property of the last node.

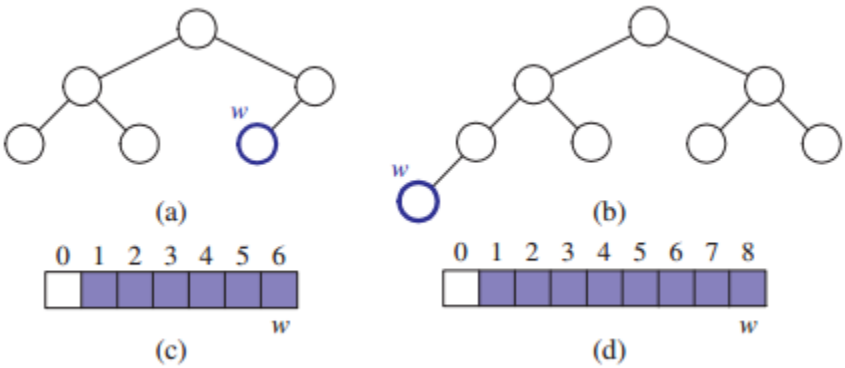

Figure 5.1.2 Two examples showing that the last node w of a heap with n nodes has level number n: (a) heap T1 with more than one node on the bottom level; (b) heap T2 with one node on the bottom level; (c) vector-based representation of T1; (d) vector-based representation of T2.

The simplifications that come from representing a complete binary tree T with a vector aid in the implementation of functions add and remove. Assuming that no array expansion is necessary, functions add and remove can be performed in O(1) time because they simply involve adding or removing the last element of the vector. Moreover, the vector associated with T has n+ 1 elements (the element at index 0 is a placeholder). If we use an extendable array that grows and shrinks for the implementation of the vector (for example, the STL vector class), the space used by the vector-based representation of a complete binary tree with n nodes is O(n) and operations add and remove take O(1) amortized time.

LESSON 5.2

Priority Queue with a Heap

We now discuss how to implement a priority queue using a heap. Our heap-based representation for a priority queue P consists of the following (see Figure 5.2.1):

- heap: A complete binary tree T whose nodes store the elements of the queue and whose keys satisfy the heap-order property. We assume the binary tree T is implemented using a vector, as described in Section 8.3.2. For each node v of T, we denote the associated key by k(v).
- comp: A comparator that defines the total order relation among the keys.

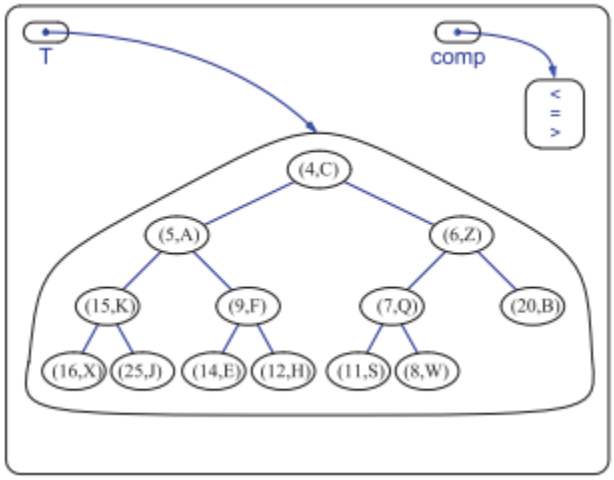

Figure 5.2.1: Illustration of the heap-based implementation of a priority queue.

With this data structure, functions size and empty take O(1) time, as usual. In addition, function min can also be easily performed in O(1) time by accessing the entry stored at the root of the heap (which is at index 1 in the vector).

Insertion

Let us consider how to perform insert on a priority queue implemented with a heap T. To store a new element e in T, we add a new node z to T with operation add, so that this new node becomes the last node of T, and then store e in this node.

After this action, the tree T is complete, but it may violate the heap-order property. Hence, unless node z is the root of T (that is, the priority queue was empty before the insertion), we compare key k(z) with the key k(u) stored at the

parent u of z. If k(z) ≥ k(u), the heap-order property is satisfied and the algorithm terminates. If instead k(z) < k(u), then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at z and u. (See Figures 5.2.2(c) and (d).) This swap causes the new entry (k,e) to move up one level. Again, the heap-order property may be violated, and we continue swapping, going up in T until no violation of the heap-order property occurs. (See Figures 5.2.2(e) and (h).)

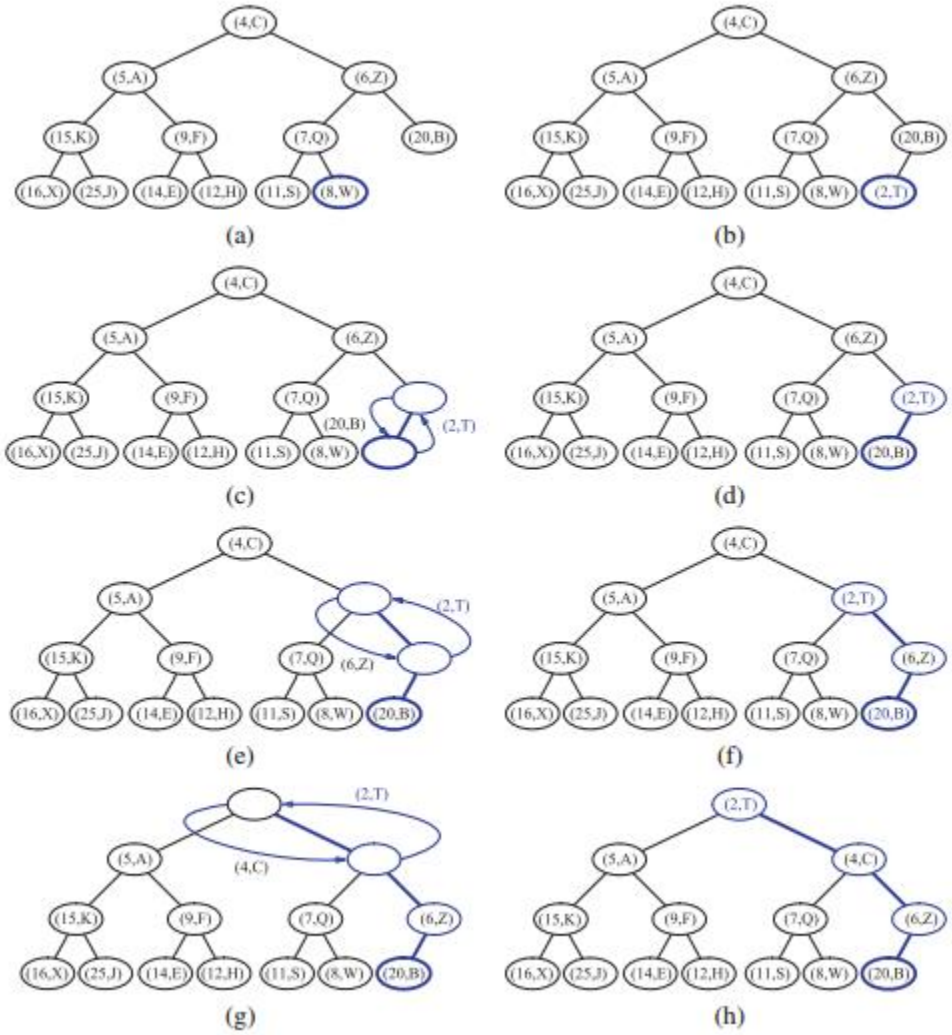

Figure 5.2.2: Insertion of a new entry with key 2 into the heap of Figure 5.2.1: (a) initial heap; (b) after performing operation add; (c) and (d) swap to locally restore the partial order property; (e) and (f) another swap; (g) and (h)final swap.

The upward movement of the newly inserted entry by means of swaps is conventionally called up-heap bubbling. A swap either resolves the violation of the heap-order property or propagates it one level up in the heap. In the worst case, up heap bubbling causes the new entry to move all the way up to the root

of heap T. (See Figure 5.2.2.) Thus, in the worst case, the number of swaps performed in the execution of function insert is equal to the height of T, that is, it is ⌊log n⌋.

Removal

Let us now turn to function removeMin of the priority queue ADT. The algorithm for performing function removeMin using heap T is illustrated in Figure 5.2.3.

We know that an element with the smallest key is stored at the root r of T (even if there is more than one entry with the smallest key). However, unless r is the only node of T, we cannot simply delete node r, because this action would disrupt the binary tree structure. Instead, we access the last node w of T, copy its entry to the root r, and then delete the last node by performing operation remove of the complete binary tree ADT. (See Figure 5.2.3(a) and (b).)

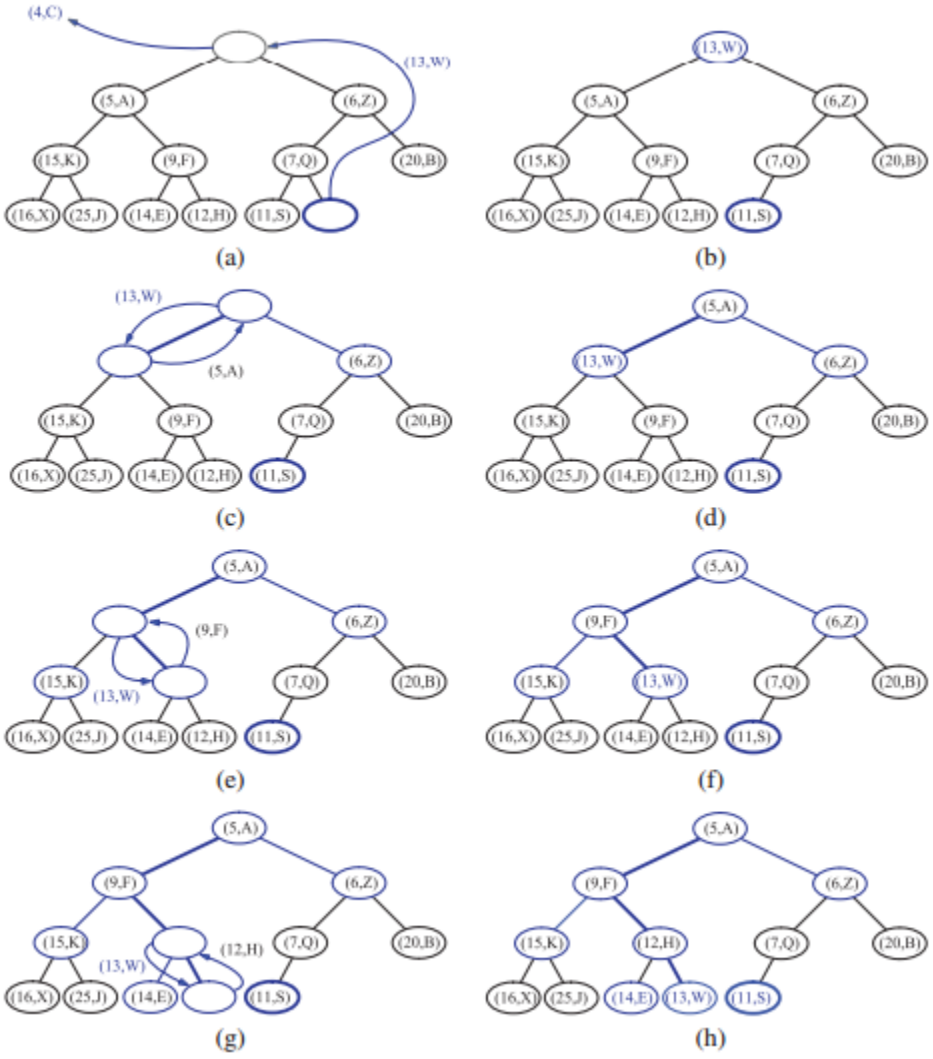

Figure 5.2.3 : Removing the element with the smallest key from a heap: (a) and (b) deletion of the last node, whose element is moved to the root; (c) and (d) swap to locally restore the heap-order property; (e) and (f) another swap; (g) and (h) final swap.

Down-Heap Bubbling after a Removal

We are not necessarily done, however, for, even though T is now complete, T may now violate the heap-order property. If T has only one node (the root), then the heap-order property is trivially satisfied and the algorithm terminates. Otherwise, we distinguish two cases, where r denotes the root of T:

- If r has no right child, let s be the left child of r
- Otherwise (r has both children), let s be a child of r with the smaller key

If $k(r) \leq k(s)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(r) > k(s)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at r and s. (See Figure 5.2.3(c) and (d).) (Note that we shouldn't swap r with s's sibling.) The swap we perform restores the heap-order property for node r and its children, but it may violate this property at s; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 5.2.3(e) and (h).)

This downward swapping process is called down-heap bubbling. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level. (See Figure 5.2.3.) Thus, the number of swaps performed in the execution of function removeMin is, in the worst case, equal to the height of heap T, that is, it is $\lfloor \log n \rfloor$.

LESSON 5.3

C++ Implementation

In this section, we present a heap-based priority queue implementation. The heap is implemented using the vector-based complete tree implementation.

In Code below, we present the class definition. The public part of the class is essentially the same as the interface, but, in order to keep the code simple, we have ignored error checking. The class's data members consists of the complete tree, named T, and an instance of the comparator object, named isLess. We have also provided a type definition for a node position in the tree, called Position.

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
int size() const;                       // number of elements
bool empty() const;                     // is the queue empty?
void insert(const E& e);                // insert element
const E& min();                         // minimum element
void removeMin();                       // remove minimum
private:
VectorCompleteTree<E> T;                // priority queue contents
C isLess;                               // less-than comparator
                                // shortcut for tree position
typedef typename VectorCompleteTree<E>::Position Position;
};
```

- A heap-based implementation of a priority queue

In Code Fragment below, we present implementations of the simple member functions size, empty, and min. The function min returns a reference to the root's element through the use of the "*" operator, which is provided by the Position class of VectorCompleteTree.

```
template <typename E, typename C>           // number of elements
int HeapPriorityQueue<E,C>::size() const
{ return T.size(); }
template <typename E, typename C>           // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
{ return size() == 0; }
template <typename E, typename C>           // minimum element
const E& HeapPriorityQueue<E,C>::min()
{ return *(T.root()); }             // return reference to root element
```

- The member functions size, empty, and min

Next, in Code Fragment below, we present an implementation of the insert operation. As outlined in the previous lessons, this works by adding the new element to the last position of the tree and then it performs up-heap bubbling by repeatedly swapping this element with its parent until its parent has a smaller key value.

```
template <typename E, typename C>              // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
T.addLast(e);                                  // add e to heap
Position v = T.last();                         // e's position
while (!T.isRoot(v)) {                         // up-heap bubbling
Position u = T.parent(v);
if (!isLess(*v, *u)) break;                    // if v in order, we're done
T.swap(v, u);                                  // . . .else swap with parent
v = u;
}
}
```

- An implementation of the function insert.

Finally, let us consider the removeMin operation. If the tree has only one node, then we simply remove it. Otherwise, we swap the root's element with the last element of the tree and remove the last element. We then apply down-heap bubbling to the root. Letting u denote the current node, this involves determining u's smaller child, which is stored in v. If the child's key is smaller than u's, we swap u's contents with this child's. The code is presented in Code Fragment below.

```
template <typename E, typename C>     // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
if (size() == 1)                      // only one node?
T.removeLast();                       // . . .remove it
else {
Position u = T.root();                // root position
T.swap(u, T.last());                  // swap last with root
T.removeLast();                       // . . .and remove last
while (T.hasLeft(u)) {                // down-heap bubbling
Position v = T.left(u);
if (T.hasRight(u) && isLess(*(T.right(u)), *v))
v = T.right(u);                       // v is u's smaller child
if (isLess(*v, *u)) {                 // is u out of order?
T.swap(u, v);                         // . . .then swap
u = v;
}
else break;                           // else we're done
}}}
```

- A heap-based implementation of a priority queue.

LESSON 5.4

Heap Sort

As we have previously observed, realizing a priority queue with a heap has the advantage that all the functions in the priority queue ADT run in logarithmic time or better. Hence, this realization is suitable for applications where fast running times are sought for all the priority queue functions. Therefore, let us again consider the PriorityQueueSort sorting, which uses a priority queue P to sort a list L.

During Phase 1, the i-th insert operation ($1 \leq i \leq n$) takes $O(1 + \log i)$ time, since the heap has i entries after the operation is performed. Likewise, during Phase 2, the j-th removeMin operation ($1 \leq j \leq n$) runs in time $O(1+\log(n- j+1)$, since the heap has $n- j + 1$ entries at the time the operation is performed. Thus, each phase takes $O(n \log n)$ time, so the entire priority-queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue. This sorting algorithm is better known as heap-sort, and its performance is summarized in the following proposition.

Proposition 5.4.1: The heap-sort algorithm sorts a list L of n elements in $O(n \log n)$ time, assuming two elements of L can be compared in O(1) time.

Let us stress that the $O(n \log n)$ running time of heap-sort is considerably better than the $O(n^2)$ running time of selection-sort and insertion-sort and is essentially the best possible for any sorting algorithm.

Implementing Heap-Sort In-Place

If the list L to be sorted is implemented by means of an array, we can speed up heapsort and reduce its space requirement by a constant factor using a portion of the list L itself to store the heap, thus avoiding the use of an external heap data structure.

This performance is accomplished by modifying the algorithm as follows:

1. We use a reverse comparator, which corresponds to a heap where the largest element is at the top. At any time during the execution of the algorithm, we use the left portion of L, up to a certain rank $i-1$, to store the elements in the heap, and the right portion of L, from rank i to $n-1$ to store the elements in the list. Thus, the first i elements of L (at ranks $0,\ldots,i-1$) provide the vector representation of the heap (with modified level numbers starting at 0 instead of 1), that is, the element at rank k is greater than or equal to its “children” at ranks $2k+1$ and $2k+2$.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the list from left to right, one step

at a time. In step i (i = 1,...,n), we expand the heap by adding the element at rank i−1 and perform up-heap bubbling.

3. In the second phase of the algorithm, we start with an empty list and move the boundary between the heap and the list from right to left, one step at a time. At step i (i = 1,...,n), we remove a maximum element from the heap and store it at rank n−i.

The above variation of heap-sort is said to be in-place, since we use only a constant amount of space in addition to the list itself. Instead of transferring elements out of the list and then back in, we simply rearrange them. We illustrate in-place heap-sort in Figure 5.4.1. In general, we say that a sorting algorithm is in-place if it uses only a constant amount of memory in addition to the memory needed for the objects being sorted themselves. A sorting algorithm is considered space-efficient if it can be implemented in-place.

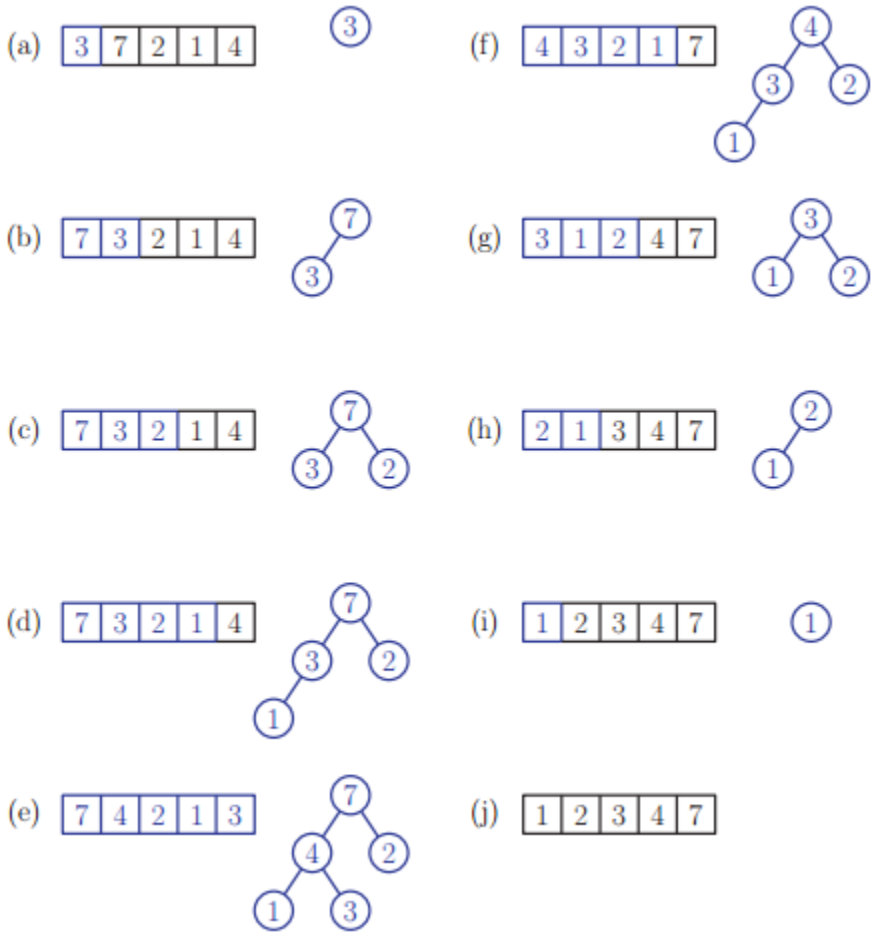

Figure 5.4.1 In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.

Bottom-Up Heap Construction

The analysis of the heap-sort algorithm shows that we can construct a heap storing n elements in O(n log n) time, by means of n successive insert operations, and then use that heap to extract the elements in order. However, if all the elements to be stored in the heap are given in advance, there is an alternative bottom-up construction function that runs in O(n) time. We describe this function in this section, observing that it can be included as one of the constructors in a Heap class instead of filling a heap using a series of n insert operations. For simplicity, we describe this bottom-up heap construction assuming the number n of keys is an integer of the type $n = 2^h - 1$. That is, the heap is a complete binary tree with every level being full, so the heap has height h = log(n+1). Viewed nonrecursively, bottom-up heap construction consists of the following h = log(n+1) steps:

1. In the first step (see Figure 5.4.2(a)), we construct (n+1)/2 elementary heaps storing one entry each.
2. In the second step (see Figure 5.4.2(b)–(c)), we form (n+ 1)/4 heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.
3. In the third step (see Figure 5.4.2(d)–(e)), we form (n + 1)/8 heaps, each storing 7 entries, by joining pairs of 3-entry heaps (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

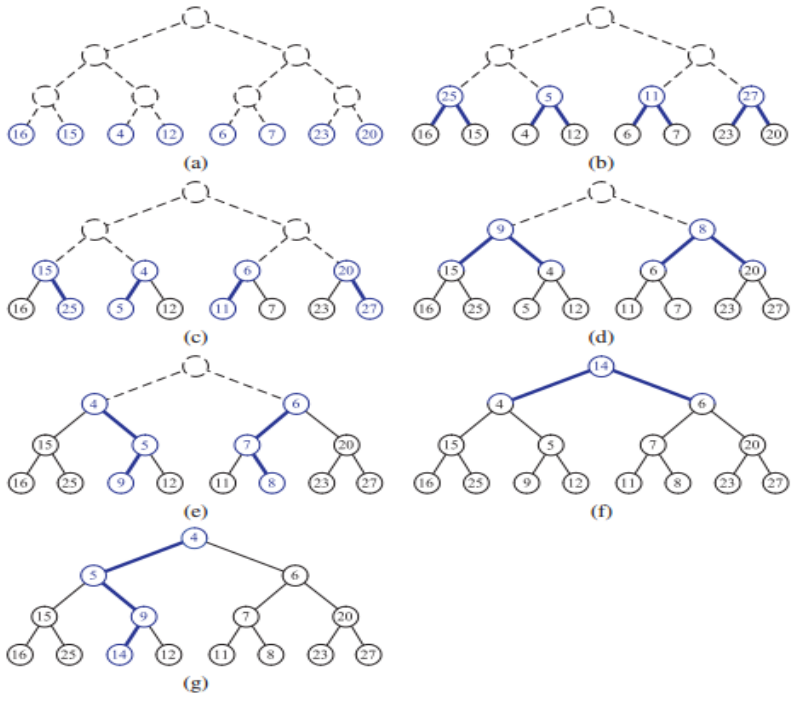

Figure 5.4.2: Bottom-up construction of a heap with 15 entries: (a) we begin by constructing one-entry heaps on the bottom level; (b) and (c) we combine these heaps into three-entry heaps; (d) and (e) seven-entry heaps; (f) and (g) we create the final heap. The paths of the down-heap bubblings are highlighted in blue. For simplicity, we only show the key within each node instead of the entire entry.

i. In the generic ith step, $2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing 2^{i-1} entries, by joining pairs of heaps storing $(2^{i-1} - 1)$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.
ii. $h+1$. In the last step (see Figure 5.4.2(f)–(g)), we form the final heap, storing all the n entries, by joining two heaps storing $(n - 1)/2$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

We illustrate bottom-up heap construction in Figure 5.4.2 for $h = 3$.