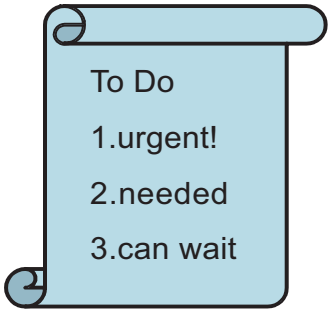
Chapter 11: Priority Queues and Heaps

In this chapter we examine yet another variation on the simple Bag data structure. A *priority queue* maintains values in order of importance. A metaphor for a priority queue is a to-do list of tasks waiting to be performed, or a list of patients waiting for an operating room in a hospital. The key feature is that you want to be able to quickly find the most important item, the value with highest priority.



Like a bag, you can add new elements into the priority queue. However, the only element that can be accessed or removed is the one value with highest priority. In this sense the container is like the stack or queue, where it was only the element at the top or the front of the collection that could be removed.

The Priority Queue ADT specification

The traditional definition of the Priority Queue abstraction includes the following operations:

add (newelement)	Add a new value to queue
first ()	Return first element in queue
removeFirst ()	Remove first element in queue
isEmpty()	Return true if collection is empty

Normally priority is defined by the user, who provides a comparison function that can be applied to any two elements. The element with the largest (or sometimes, the smallest) value will the deemed the element with highest priority.

A priority queue is not, in the technical sense, a true queue as described in Chapter 7. To be a queue, elements would need to satisfy the FIFO property. This is clearly not the case for the priority queue. However, the name is now firmly attached to this abstraction, so it is unlikely to change.

The following table shows priority queue operation names in the C++ STL and in the Java class PriorityQueue.

Operation	C++ STL class priorityQueue<T>	Java Container class priorityQueue
Add value	push(E)	add(E)
First Value	top()	peek()
Remove First Value	pop()	poll()
Size, or test size	empty()	size() = = 0

In C++ it is the element with the largest value, as defined by the user provided comparison function, that is deemed to be the first value. In the Java library, on the other hand, it is the element with the smallest value. However, since the user provides the comparison function, it is easy to invert the sense of any test. We will use the smallest value as our first element.

Applications of Priority Queues

The most common use of priority queues is in simulation. A simulation of a hospital waiting room, for example, might prioritize patients waiting based on the severity of their need.

A common form of simulation is termed a "discrete, event-driven simulation". In this application the simulation proceeds by a series of "events". An event is simply a representation of an action that occurs at a given time. The priority queue maintains a collection of events, and the event with highest priority will be the event with lowest time; that is, the event that will occur next.

For example, suppose that you are simulating patrons arriving at a small resturant. There are three main types of event of interest to the simulation, namely patrons arriving (the arrival event), patrons ordering (the order event) and patrons leaving (the leave event). An event might be represented by a structure, such as the following:

```
struct eventStruct {
  int time;
  int groupsize;
  enum {arrival, order, leave} eventType;
};
```

To initialize the simulation you randomly generate a number of arrival events, for groups of various sizes, and place them into the queue. The execution of the simulation is then described by the following loop:

```
while the event queue is not empty
  select and remove the next event
  do the event, which may generate new events
```

To "do" the event means to act as if the event has occurred. For example, to "do" an arrival event the patrons walk into the resturant. If there is a free table, they are seated and an subsequent "order" event is added to the queue. Otherwise, if there is not a free table, the patrons either leave, or remain in a queue of waiting patrons. An "order" event produces a subsequent "leave" event. When a "leave" event occurs, the newly emptied table is then occupied by any patrons waiting for a table, otherwise it remains empty until the next arrival event.

Many other types of simulations can be described in a similar fashion.

Priority Queue Implementation Techniques

We will explore three different queue implementation techniques, two of which are developed in the worksheets. The first is to examine how any variety of ordered bag (such as the SortedBag, SkipList, or AVLtree) can be used to implement the priority queue. The second approach introduces a new type of binary tree, termed the *heap*. The classic heap (known simply as the heap) provides a very memory efficient representation for a priority queue. Our third technique, the *skew heap*, uses an interesting variation on the heap technique.

A note regarding the name heap.

The term *heap* is used for two very different concepts in computer science. The heap data structure is an abstract data type used to implement priority queues. The terms *heap, heap memory*. *heap allocation*, and so on are used to describe memory that is allocated directly by the user, using the **malloc** function. You should not confuse the two uses of the same term.

Building a Priority Queue using an Ordered Bag

In earlier chapters you have encountered a number of different bag implementation techniques in which the underlying collection was maintained in sequence. Examples included the sorted dynamic array, the skip list, and the AVL tree. In each of these containers the smallest element is always the first value. While the bag does not support direct access to the first element (as does, say, the queue), we can nevertheless obtain access to this value by means of an iterator. This makes it very easy to implement a priority queue using an ordered bag as a storage mechanism for the underlying data values, as the following:

add(E). Simply add the value to the collection using the existing insertion functions.

first(). Construct an iterator, and return the first value produced by the iterator.

removeFirst(). Use the existing remove operation to delete the element returned by first().

isEmpty(). Use the existing size operation for the bag, and return true if the size is zero.

We have seen three ordered collections, the sorted array, the skip list, and the AVL tree. Based on your knowledge of the algorithmic execution speeds for operations in those data structures, fill in the following table with the execution times for the bag heap constructed in a fashion described above.

Operation	SortedArray	SkipList	AVLtree
Add(newElement)			
First()			

removeFirst()			

A note on Encapsulation

There are two choices in developing a new container such as the one described above. One choice is to simply add new functions, or extend the interface, for an existing data structure. Sometimes these can make use of functionality already needed for another purpose. The balanced binary tree, for example, already needed the ability to find the leftmost child in a tree, in order to implement the remove operation. It is easy to use this function to return the first element in the three. However, this opens the possibility that the container could be used with operations that are not part of the priority queue interface.

An alternative would have been to create a new data structure, and encapsulate the underlying container behind a structure barrier, using something like the following:

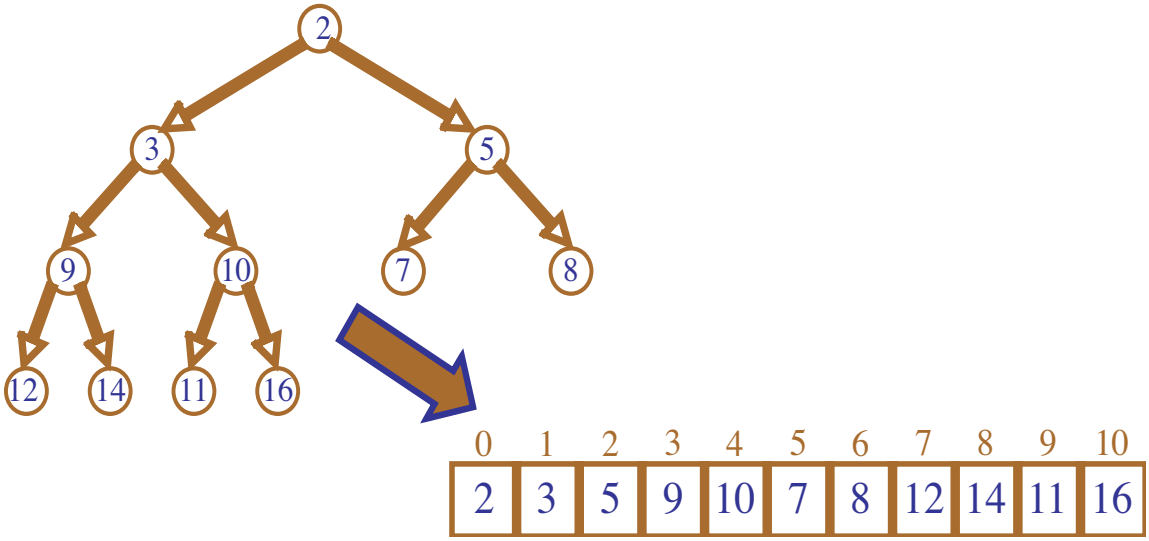
```
struct SortedHeap {
   struct AVLtree data;
};
```

We would then need to write routines to initialize this new structure, rather than relying on the existing routines to initialize an AVL tree. At the cost of an additional layer of indirection we can then more easily guarantee that the only operations performed will be those defined by the interface.

There are advantages and disadvantages to both. The bottom line is that you, as a programmer, should be aware of both approaches to a problem, and more importantly be aware of the implications of whatever design choice you make.

Building a Priority Queue using a Heap

In a worksheet you will explore two alternative implementation techniques for priority queues that are based around the idea of storing values in a type of representation termed a *heap*. A heap is a binary tree that also maintains the property that the value stored at every node is less than or equal to the values stored at either of its child nodes. This is termed the *heap order property*. The classic heap structure (known just as the heap) adds the additional requirement that the binary tree is complete. That is, the tree if full except for the bottom row, which is filled from left to right. The following is an example of such a heap:

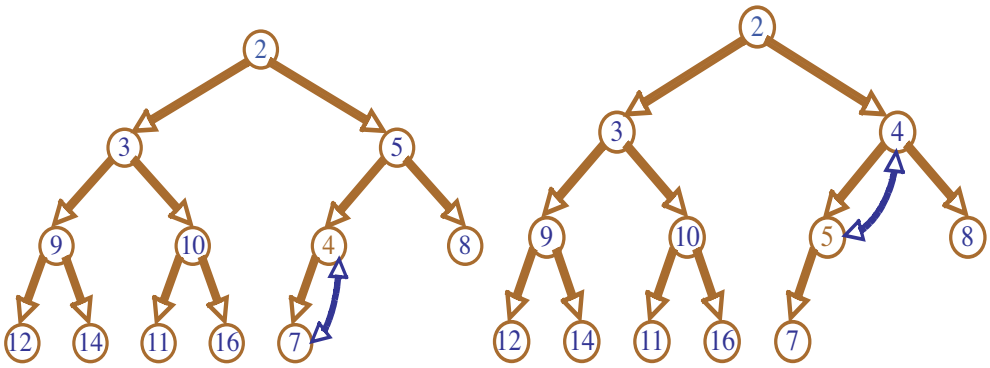


Notice that a heap is partially ordered, but not completely. In particular, the smallest element is always at the root. Although we will continue to think of the heap as a tree, we will make use of the fact that a complete binary tree can be very efficiently represented as an array. To root of the tree will be stored as the first element in the array. The children of node i are found at positions 2i+1 and 2i+2, the parent at (i-1)/2. You should examine the tree above, and verify that the transformation given will always lead you to the children of any node. To reverse the process, to move from a node back to the parent, simply subtract 1 and divide by 2. You should also verify that this process works as you would expect.

We will construct our heap by defining functions that will use an underlying dynamic array as the data container. This means that users will first need to create a new dynamic array before they can use our heap functions:

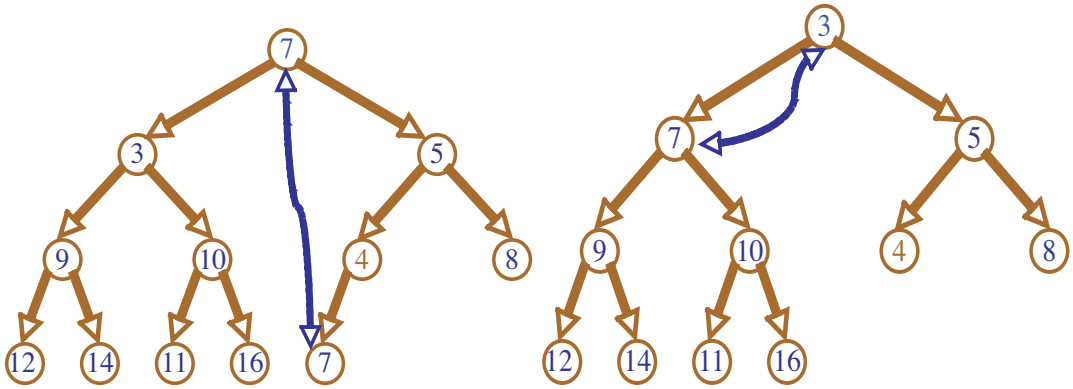
```
struct dyArray heap;  /* create a new dynamic array */
dyArrayInit (&heap, 10);  /* initialize the array to 10 elements */
…
```

To insert a new value into a heap the value is first added to the end. (This operation has actually already been written in the function dyArrayAdd). Adding an element to the end preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is *percolated up* into position. It is compared to its parent node. If smaller, the node and the parent are exchanged. This continues until the root is reached, or the new value finds its correct position. Because this process follows a path in a complete binary tree, it is O(log n). The following illustrates adding the value 4 into a heap, then percolating it up until it reaches its final position. When the value 4 is compared to the 2, the parent node containing the 2 is smaller, and the percolation process halts.



Because the process of percolating up traverses a complete binary tree from leaf to root, it is O(log n), where n represents the number of nodes in the tree.

Percolating up takes care of insertion into the heap. What about the other operations? The smallest value is always found at the root. This makes accessing the smallest element easy. But what about the removal operation? When the root node is removed it leaves a "hole." Filling this hole with the last element in the heap restores the complete binary tree property, but not the heap order property. To restore the heap order the new value must *percolate down* into position.

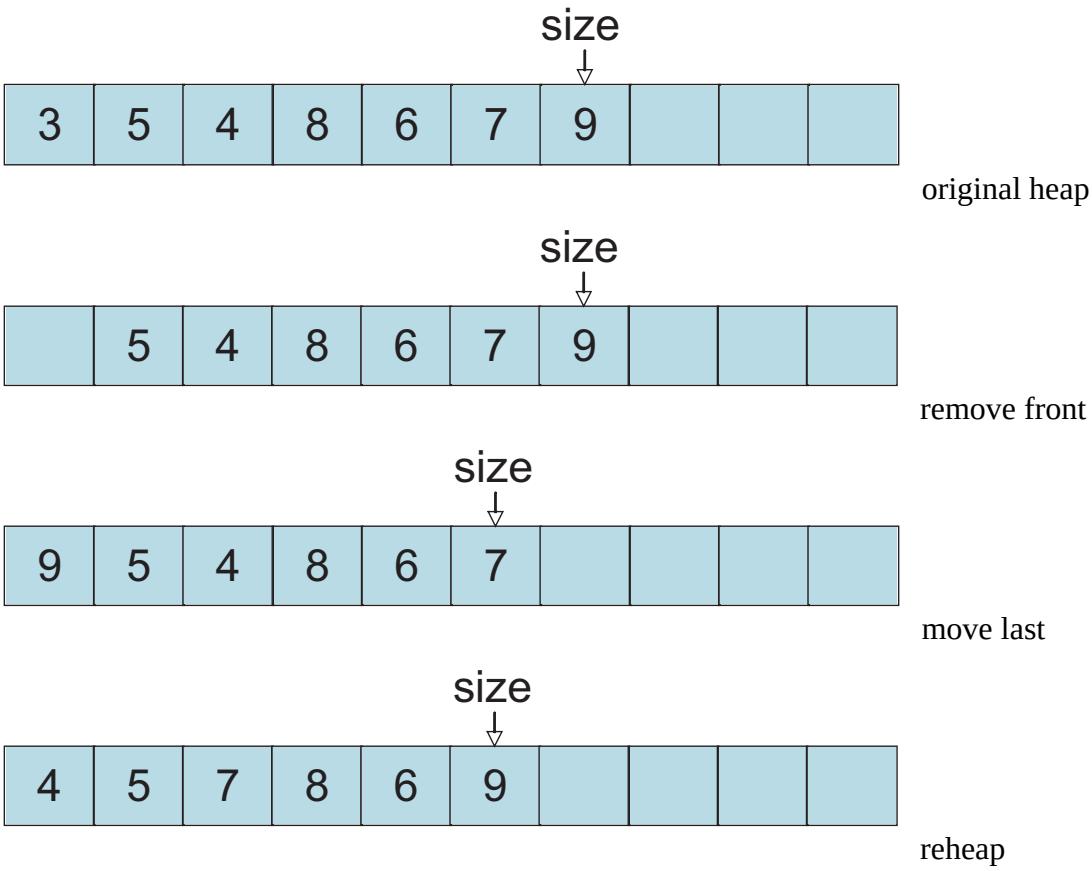


To percolate down a node is compared to its children. If there are no children, the process halts. Otherwise, the value of the node is compared to the value of the smallest child. If the node is larger, it is swapped with the smallest child, and the process continues with the child. Again, the process is traversing a path from root to leaf in a complete binary tree. It is therefore O(log n).

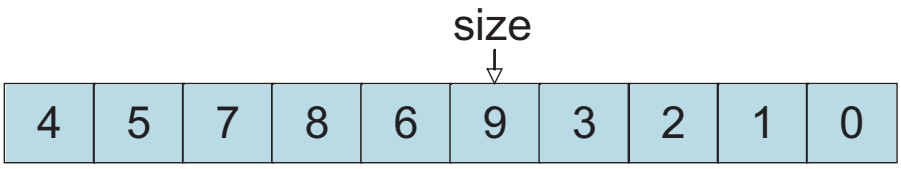
In worksheet 33 you will complete the implementation of the priority queue constructed using a heap by providing the functions to percolate values into position, but up and down.

Heap Sort

When a value is removed from a heap the size of the heap is reduced. If the heap is being represented in an array, this means that the bottom-most element of the array is no longer being used. (Using the terminology of the dynamic array we examined earlier, the size of the collection is smaller, but the capacity remains unchanged).

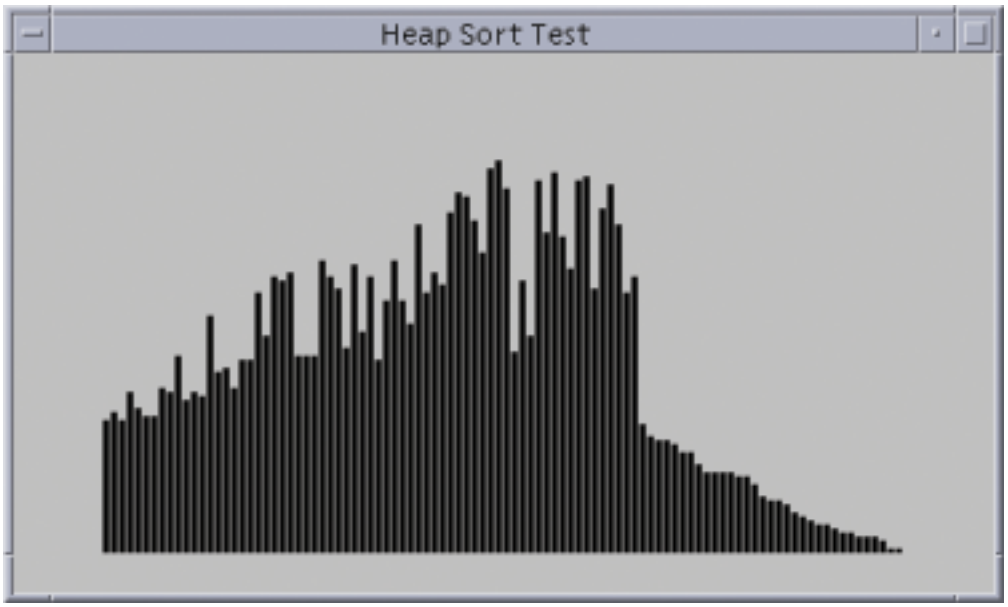


What if we were to store the removed values in the now unused section of the array? In fact, since the last element in the array is always moved into the hole made by the removal of the root, it is a trivial matter to simply swap this value with the root.



Suppose we repeat this process, always swapping the root with the currently last element, then reheaping by percolating the new root down into position. The result would be a sorting algorithm, termed *heap sort*. The following is a snapshot illustrating heap sort in the middle of execution. Notice that the smallest elements have been moved to the right. The current size of the dynamic array is indicated by the sharp drop in values. The

elements to the left of this point are organized in a heap. Notice that the heap is not completely ordered, but has a tendency towards being ordered.



To determine the algorithmic execution time for this algorithm, recall that adjustHeap requires O(log n) steps. There are n executions of adjustHeap to produce the initial heap. Afterwards, there are n further executions to reheap values during the process of sorting. Altogether the running time is O(n log n). This matches that of merge sort, quick sort, and tree sort. Better yet, heap sort requires no additional storage.

Question: Simulate execution of the Heap sort algorithm on the following values:

9 3 2 4 5 7 8 6 1 0

First make the values into a heap (the graphical representation is probably easier to work with than the vector form). Then repeatedly remove the smallest value, and rebuild the heap.

Skew Heaps

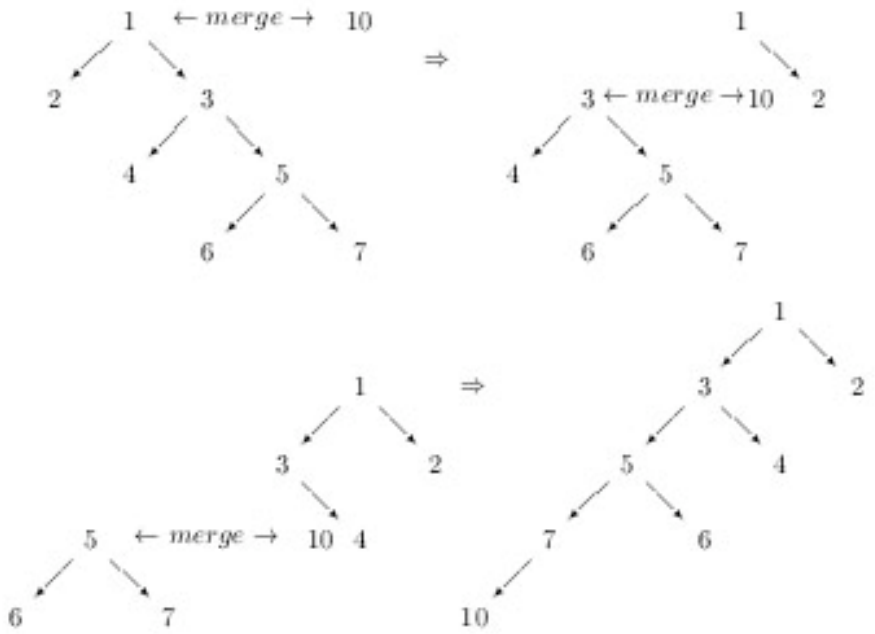
In a heap the relative order of the left and right children is unimportant. The only property that must be preserved is that each node is smaller than either of its children. However, the heap order property does not insist that, for example, a left child is smaller than a right child. The *skew heap* builds on this flexibility, and results in a very different organization from the traditional heap. The skew heap makes two observations. First, left and right children can always be exchanged with each other, since their order is unimportant. Second, both insertions and removals from a heap can be implemented as special cases of a more general task, which is to merge two heaps into one.

It is easy to see how the remove is similar to a merge. When the smallest (that is, root) element is removed, you are left with two trees, namely the left and right child trees. To build a new heap you can simply merge the two.

To view addition as a merge, consider the existing heap as one argument, and a tree with only the single new node as the second. Merge the two to produce the new heap.

Unlike the classic heap, a skew heap does not insist that the binary tree is complete. Furthermore, a skew heap makes no attempt to guarantee the balance of its internal tree. Potentially, this means that a tree could become thin and unbalanced. But this is where the first observation is used. During the merge process the left and right children are systematically swapped. The result is that a thin and unbalanced tree cannot remain so. It can be shown (although the details are not presented here) that amortized over time, each operation in a skew heap is no worst than O(log n).

The following illustrates the addition of the value 10 to an existing tree. Notice how a tree with a long right path becomes a tree with a long left path.



The merge algorithm for a skew heap can be described as follows:

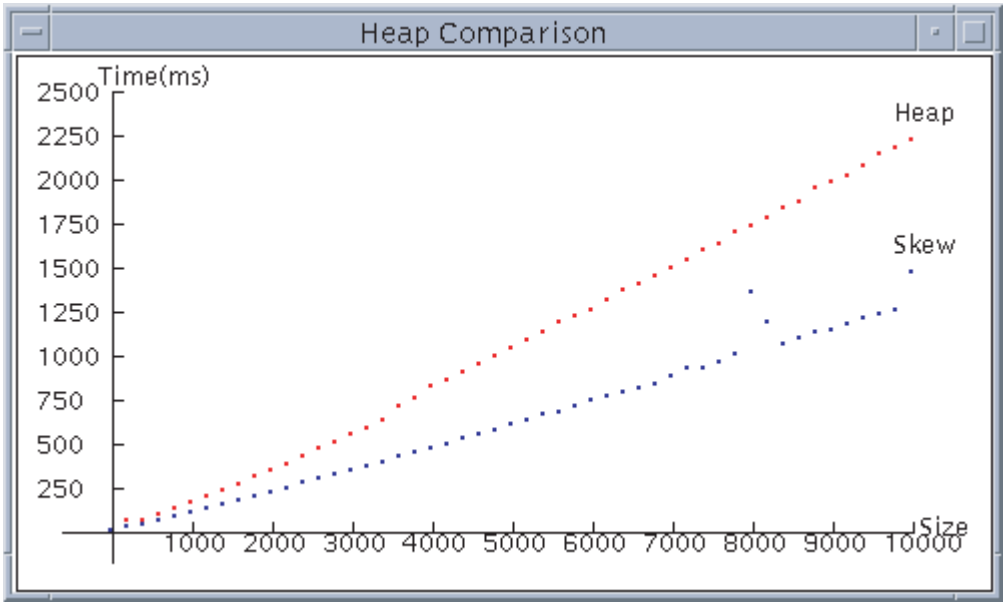
```
Node merge (Node left, Node right)
  if (left is null) return right
  if (right is null) return left
  if (left child value < right child value) {
    Node temp = left.left;
    left.left = merge(left.right, right)
    left.right = temp
    return left;
```

```
} else {
  Node temp = right.right
  right.right = merge(right.left, left)
  right.left = temp
  return right
}
```

In worksheet 35 you will complete the implementation of the SkewHeap based on these ideas.

Like the self organizing list described in Chapter 8, a skew heap is an example of a data structure that tries to optimize future performance based on past operations. It makes no guaranteed that poorly balanced trees cannot arise, but if they do they are quickly taken apart so that they cannot have a lasting impact on execution time.

To measure the relative performance of the Heap and SkewHeap abstractions, an experiment was conducted in which n random integers between 0 and 100 were inserted into a heap and then removed. A plot of the execution times for various values of n was obtained as follows. The result indicates that even through the SkewHeap is performing many more memory allocations than the Heap, the overall execution time is still faster.



Short Exercises

1. Given an example of a priority queue that occurs in a non-computer science situation.

2. Where is the smallest value found in a heap?

3. Where is the 2nd smallest element in a heap? The third smallest element?

4. What is the height of a heap that contains 10 elements?

5. If you interpret a sorted array as a heap, is it guaranteed to have the heap order property?

6. Could you implement the tree sort algorithm using a heap? Recall that the tree sort algorithm simply copies values from a vector into a container (that is, the heap), the repeatedly removes the smallest element and copies it back to the vector. What is the big-oh execution time for this algorithm?

7. Suppose you wanted to test the Heap data structure. What would be some good boundary value test cases?

8. Program a test driver for the Heap data type, and execute the operations using the test cases you identified in the previous question.

9. An operation we have not discussed is the ability to change the value of an item already in the heap. Describe an algorithm that could be used for this purpose. What is the complexity of your algorithm?

Analysis Exercises

1. Show that unless other information is maintained, finding the *maximum* value in a heap must be an O(n) operation. Hint: How many elements in the heap could potentially be the maximum?

2. Imagine a heap that contains 2^n values. This will naturally represent a complete binary tree. Notice that the heap property is retained if the left and right subtrees of any node are exchanged. How many different equivalent heaps can be produced using only such swappings?
3.

Self Study Questions

1. What are the abstract operations that characterize a priority queue?

2. Give an example of a priority queue that occurs in a non-computer science situation.

3. Describe how a priority queue is similar to a bag, and how it is different. Describe how a priority queue is similar to a queue, and how it is different.

4. Why is a priority queue not a true queue in the sense described in Chapter 7?

5. What is discrete event driven simulation? How is it related to priority queues?

6. What property characterizes the nodes in a heap?

7. When a binary tree, such as a heap, is represented as a dynamic array, where are the children of node i? What is the index of the parent of node i?

8. When a heap is represented in an array, why is it important that the tree being represented in complete? What happens if this condition is not satisfied?

9. Illustrate the process of percolating a value down into position. Explain why this operation is O(log n).

10. Illustrate the process of percolating a value up into position. Explain why this operation is O(log n).

11. Place the following values, in the order shown, into an initially empty heap, and show the resulting structure: 5 6 4 2 9 7 8 1 3

12. How is a skew heap different from a heap? What property of the heap data type does this data structure exploit?

13. Perform the same insertions from question 10 into an initially empty skew heap, and show the resulting structure.

Chapter Summary

Key Concepts

- Priority Queue
- Heap
- Heap order property
- Heap sort
- Skew heap

A *priority queue* is not a true queue at all, but is a data structure designed to permit rapid access and removal of the smallest element in a collection. One way to build a priority queue is to use an ordered collection. However, they can also be constructed using an efficient array representation and an idea termed the heap. A heap is a binary tree that supports the heap order property. The heap order property says that the value stored at every node is smaller than the value stored at either of its child node. The classic heap stores elements in a complete binary tree. Because the tree is complete, it can be represented in an array form without any holes appearing in the collection. In addition to providing the basis for implementing a priority queue, the heap structure forms the basis of a very efficient sorting algorithm.

A skew heap is a form of heap that does not have the fixed size characteristic of the vector heap. The skew heap data structure is interesting in that it can potentially have a very poor worst case performance. However, it can be shown that the worst case performance cannot be maintained, and following any occurrence the next several

operations of insertion or removal must be very rapid. Thus, when measured over several operations the performance of a skew heap is very impressive.

Programming Exercises

1. Complete the implementation of the priority queue using a sorted dynamic array as the underlying container. This technique allows you to access both the smallest and the largest element with equal ease. What is the algorithmic execution time for operations with this representation?

2. Consider using a balanced binary tree, such as an AVL tree, for an implementation technique. Recall that as part of the process of removing an element from an AVL tree you needed the ability to find the leftmost child of the right subtree. Show how using the ability to identify the leftmost child can be used to find the smallest element in the tree. Write priority queue operations based on this observation.

3. Complete the implementation of the resturant simulation described in this chapter. Initialize your simulation with a number of arrival events selected at random points over the period of an hour. Assume that patrons take between five and ten minuets (selected randomly) between the time they are seated and the time they order. Assume that patrons take between 30 to 50 minuits (again, selected randomly) to eat. Those patrons who are not able to seat wait on a queue for a table to become available. Print out a message every time an event occurs indicating the type of the event, and the time for the new events generated by the current event. Keep track of the number of patrons serviced in the period of two hours.

4. Program an implementation of an airport. The airport has two runways. Planes arrive and request permission to land and, independently, planes on the ground request permission to talk off.

5. Program a discrete event driven simulation of a hospital emergency room. Upon arrival, a triage doctor assigns each patient a number based on the severity of his or her harm. The room has a fixed number of beds. As each bed is freed, the next most urgent patient is handled. Patients take varying amounts of time to handle depending upon their condition. Collect statistics on the length of time each patient will have to wait.

6. An alternative approach to the skew heap is a leftist heap. In a leftist heap, the height of the left child is always larger than or equal to the height of the right child. As two children in a heap can always be swapped without violating the heap-order property, this is an easy condition to impose. To construct a leftist heap, nodes are modified so as to remember their height. (We saw how to do this in the AVL tree). As with the skew heap, all operations on a leftist heap are implemented using the single task of merging two heaps. The following steps are used to merge heaps T1 and T2:
    a. If the value in T2 is less than the value in T1, merge in the opposite order

b. The root of T1 is the root of the new heap. The right shild is recursively merged with T2
c. If the resulting height of the right child is larger than the left child, the two children are reversed
d. The height of the result is set to one lrger than the height of the left child.

Because the merge is always performed with a right child, which is always has a height smaller than the left child, the result is produced very quickly. Provide an implementation of a heap data structure based on this principle.

On the Web

The wikipedia contains articles explaining the priority queue abstraction, as well as the heap data structure, and the associated heap sort algorithm. The entry for heap contains links to several variations, including the skew heap, as well as others. Wikipedia also contains a good explanation of the concept of Discrete Event Simlation. The on-line *Dictionary of Algorithms and Data Structures* contains explanations of heap, the heapify algorithm, and other topics discussed in this chapter.