9.2 DEAPS

9.2.1 Definition

A *deap* is a double-ended heap that supports the double-ended priority queue operations of insert, delete min, and delete max. As in the case of the min-max heap, these operations take logarithmic time on a deap. However, the deap is faster by a constant factor and the algorithms are simpler.

Definition: A *deap* is a complete binary tree that is either empty or satisfies the following properties:

(1) The root contains no element.

(2) The left subtree is a min-heap.

(3) The right subtree is a max-heap.

(4) If the right subtree is not empty, then let i be any node in the left subtree. Let j be the corresponding node in the right subtree. If such a j does not exist, then let j be the node in the right subtree that corresponds to the parent of i. The key in node i is less than or equal to the key in j. □

An example of an 11-element deap is shown in Figure 9.6. The root of the min-heap contains 5, while that of the max-heap contains 45. The min-heap node with key 10 corresponds to the max-heap node with key 25, while the min-heap node with key 15 corresponds to the max-heap node with key 20. For the node containing 9, the node j defined in property (4) of the deap definition is the max-heap node that contains 40.

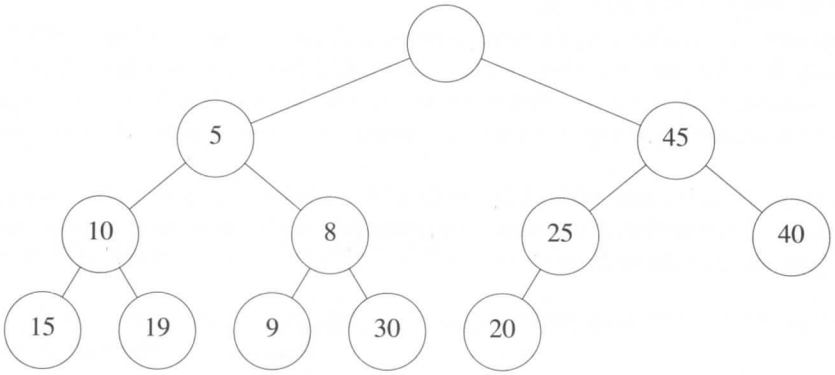

Figure 9.6: An 11 element deap

From the definition of a deap, it is evident that in an n element deap, $n > 1$, the min element is in the root of the min-heap while the max element is in the root of the max-heap. If $n = 1$, then the min and max elements are the same and are in the root of the min-heap. Since a deap is a complete binary tree, it may stored as an implicit data structure in a one dimensional array much the same way as min-, max-, and min-max-heaps are stored. In the case of a deap, position 1 of the array is not utilized (we may simply begin the array indexing at 2 rather than at 1). Let n denote the last occupied position in this array. Then the number of elements in the deap is $n - 1$. If i is a node in the min-heap, then its corresponding node in the max-heap is $i + 2^{\lfloor \log_2 i \rfloor - 1}$. Hence the j defined in property (4) of the definition is given by:

$$j = i + 2^{\lfloor \log_2 i \rfloor - 1};$$
$$\text{if } (j > n)\ j /= 2;$$

Notice that if property (4) of the deap definition is satisfied by all leaf nodes i of the min-heap, then it is satisfied by all remaining nodes of the min-heap too.

The double-ended priority queue operations are particularly easy to implement on a deap. The complexity of each operation is bounded by the height of the deap which is logarithmic in the number of elements in the deap.

9.2.2 Insertion Into A Deap

Suppose we wish to insert an element with key 4 into the deap of Figure 9.6. Following this insertion, the deap will have 12 elements in it and will thus have the shape shown in Figure 9.7. j points to the new node in the deap.

The insertion process begins by comparing the key 4 to the key in j's corresponding node, i, in the min-heap. This node contains a 19. To satisfy property (4), we move the 19 to node j. Now, if we use the min-heap insertion algorithm to insert 4 into position i, we get the deap of Figure 9.8.

If instead of inserting a 4, we were to insert a 30 into the deap of Figure 9.6, then the resulting deap has the same shape as in Figure 9.7. Comparing 30 with the key 19 in the corresponding node i, we see that property (4) may be satisfied by using the max-heap insertion algorithm to insert 30 into position j. This results in the deap of Figure 9.9.

The case when the new node, j, is a node of the min-heap is symmetric to the case just discussed. The function *deap_insert* (Program 9.4) implements the insert operation. The data type, *deap*, is defined as:

```
element deap [MAX_SIZE]
```

The position of the last element in the deap is n and $n = 1$ denotes an empty deap.

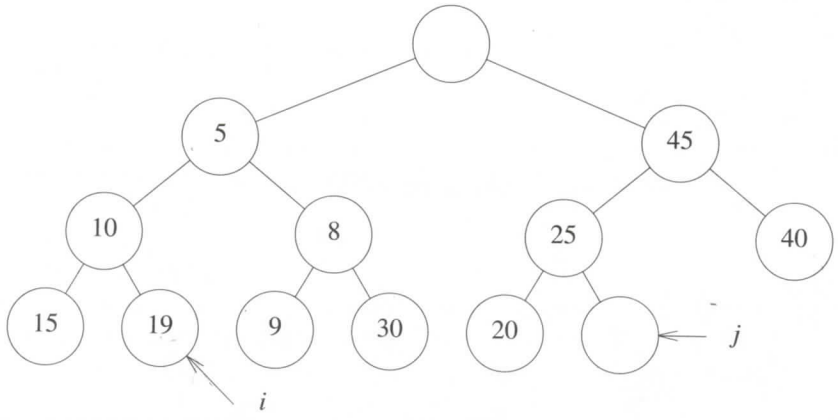

Figure 9.7: Shape of a 12 element deap

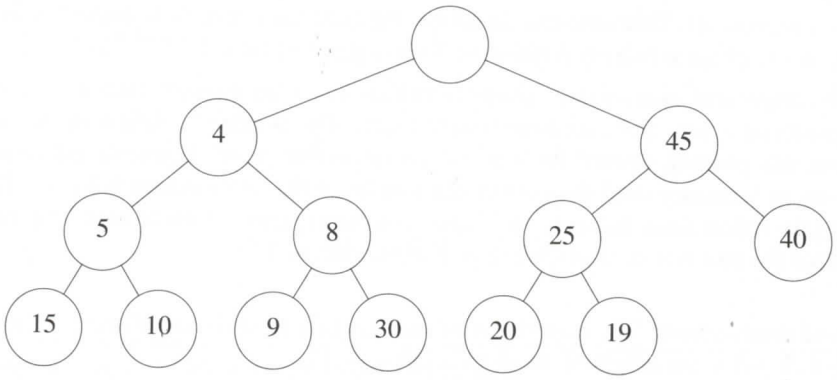

Figure 9.8: Deap of Figure 9.6 following the insertion of 4

The function *deap_insert* uses the following functions whose implementation we leave as exercises:

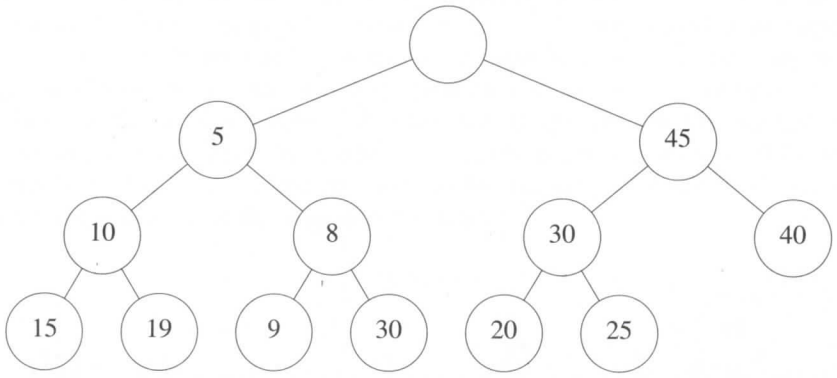

Figure 9.9: Deap of Figure 9.6 following the insertion of 30

(1) *max_heap*(n). This function returns *TRUE iff* n is a position in the max-heap of the deap.

(2) *min_partner*(n). This function computes the min-heap node that corresponds to the max-heap position n. This is given by $n - 2^{\lfloor \log_2 n \rfloor - 1}$

(3) *max_partner*(n). This function computes the max-heap node that corresponds to the parent of the min-heap position n. This is given by $(n + 2^{\lfloor \log_2 n \rfloor - 1}) / 2$.

(4) *min_insert* and *max_insert*. These functions insert an element into a specified position of a min- and max-heap, respectively. This is done by following the path from this position toward the root of the respective heap. Elements are moved down as necessary until the correct place to insert the new element is found. This process differs from that used in Chapter 5 to insert into a min- or max-heap only in that the root is now at position 2 or 3 rather than at 1.

Analysis of *deap_insert*: The correctness of this function is easily established. Its complexity is O(log$n$) as the height of the deap is O(log$n$). □

9.2.3 Deletion Of Min Element

Now consider the delete min operation. A description of the deletion process is given in Program 9.5. The strategy is to first transform the deletion of the element from the root of the min-heap to the deletion of an element from a leaf position in the min-heap. This is done by following a root to leaf path in the min-heap ensuring that the min-heap properties are satisfied on the preceding levels of the heap. This process has the effect of shifting the empty position initially at the min-heap root to a leaf node p. This leaf node

```
void deap_insert(element deap[], int *n, element x)
{
/* insert x into the deap */
   int i;
   (*n) ++;
   if (*n == MAX_SIZE) {
      fprintf(stderr, "The heap is full\n");
      exit(1);
   }
   if (*n == 2)
      deap[2] = x; /* insert into empty deap */
   else switch(max_heap(*n)) {
      case FALSE:  /* *n is a position on min side */
            i = max_partner(*n);
            if (x.key > deap[i].key)  {
               deap[*n] = deap[i];
               max_insert(deap,i,x);
            }
            else
               min_insert(deap,*n,x);
            break;
      case TRUE: /* *n is a position on max side */
            i = min_partner(*n);
            if (x.key < deap[i].key) {
               deap[*n] = deap[i];
               min_insert(deap,i,x);
            }
            else
               max_insert(deap,*n,x);
   }
}
```

Program 9.4: Function to insert an item into a deap

is then filled by the element, t, initially in the last position of the deap. The insertion of t into position p of the min-heap is done as in *deap_insert* except that the specification of *max_partner*(i) is changed to:

$$j = i + 2^{\lfloor \log_2 i \rfloor - 1};$$
$$\text{if } (j > n)\ j /= 2;$$

and the insertion does not increase the size of the deap. Function *modified_deap_insert*

does this insertion. We leave the writing of this function as an exercise.

```
element deap_delete_min(element deap[], int *n)
{
/* delete the minimum element from the heap */
   int i,j;
   element temp;
   if (*n <  2) {
      fprintf(stderr, "The deap is empty\n");
      /* return an error code to user */
      deap[0].key = INT_MAX;
      return deap[0];
   }
   deap[0] = deap[2];  /* save min element */
   temp = deap[(*n)--];
   for (i = 2; i*2 <= *n; deap[i] = deap[j], i = j) {
   /* find node with smaller key */
   j = i*2;
   if (j+1 <= *n) {
      if (deap[j].key > deap[j+1].key)
         j++;
   }
}
modified_deap_insert(deap,i,temp);
return deap[0];
}
```

Program 9.5: Delete min function

For example, suppose that we wish to remove the minimum element from the deap of Figure 9.6. To do this, we first place the last element (the one with key 20) in the deap into a temporary element, *temp*, since the deletion removes this node from the heap structure. Next, we fill the vacancy created in the min-heap root (node 2) by the removal of the minimum element. To fill this vacancy we move along the path from the root to a leaf node. Prior to each move, we place the smaller of the elements in the current node's children into the current node. We then move to the node previously occupied by the moved element. In this example, we first move 8 into node 2. Then we move 9 into the node formerly occupied by 8. Now, we have an empty leaf and proceed to insert 20 into this. We compare 20 with the key 40 in its max partner. Since $20 < 40$, no exchange is needed and we proceed to insert the 20 into the min-heap beginning at the empty position. This operation results in the deap of Figure 9.10.

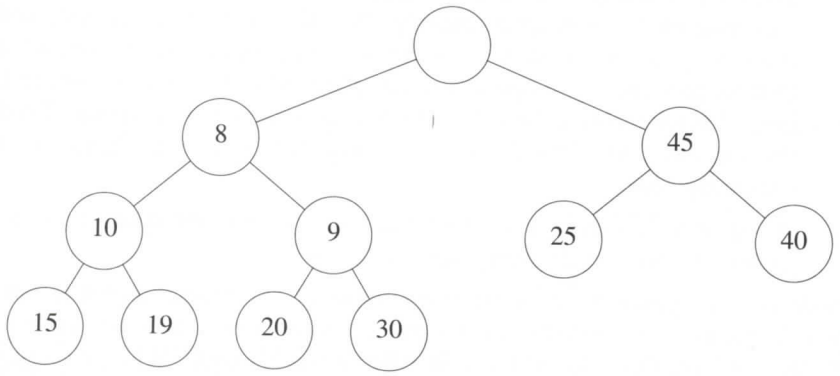

Figure 9.10: Deap of Figure 9.6 following a delete min

Analysis of *deap_delete_min*: We may easily verify that *deap_delete_min* works correctly regardless of whether the last position in the deap is in the min- or max-heap. The complexity is O(log$n$) as the height of a deap is O(log$n$). □

The *deap_delete_max* operation is performed in a similar manner.

EXERCISES

1. Complete the *deap_insert* function (Program 9.4) by writing all the functions that it uses. Test the insertion function by running it on a computer. Generate your own test data.
2. Complete the *deap_delete_min* function (Program 9.5) by writing all the functions that it uses. Test the correctness of your function by running it on a computer using test data of your choice.
3. Write a function to initialize a deap with n elements. Your function must run in $O(n)$ time. Show that it actually has this running time. (Hint: Use a series of adjusts as discussed in Section 5.6.)
4. Write the functions to perform all double-ended priority queue operations for a min-max heap and for a deap.
   (a) Use suitable test data to test the correctness of your functions.