20. Implicit data structures

Learning objectives:

- implicit data structures describe relationships among data elements implicitly by formulas and declarations
- array storage
- band matrices
- sparse matrices
- Buffers eliminate temporary speed differences among interacting producer and consumer processes.
- fifo queue implemented as a circular buffer
- priority queue implemented as a heap
- heapsort

What is an implicit data structure?

An important aspect of the art of data structure design is the efficient representation of the structural relationships among the data elements to be stored. Data is usually modeled as a graph, with nodes corresponding to data elements and links (directed arcs, or bidirectional edges) corresponding to relationships. Relationships often serve a double purpose. Primarily, they define the semantics of the data and thus allow programs to interpret the data correctly. This aspect of relationships is highlighted in the database field: for example, in the entity-relationship model. Secondarily, relationships provide a means of accessing data, by starting at some element and following an *access path* that leads to other elements of interest. In studying data structures we are mainly concerned with the use of relationships for access to data.

When the structure of the data is irregular, or when the structure is highly dynamic (extensively modified at run time), there is no practical alternative to representing the relationships explicitly. This is the domain of list structures, presented in the chapter on “List structures”. When the structure of the data is static and obeys a regular pattern, on the other hand, there are alternatives that compress the structural information. We can often replace many explicit links by a few formulas that tell us where to find the "neighboring" elements. When this approach works, it saves memory space and often leads to faster programs.

We use the term *implicit* to denote data structures in which the relationships among data elements are given implicitly by formulas and declarations in the program; no additional space is needed for these relationships in the data storage. The best known example is the array. If one looks at the area in which an array is stored, it is impossible to derive, from its contents, any relationships among the elements without the information that the elements belong to an array of a given type.

Data structures always go hand in hand with the corresponding procedures for accessing and operating on the data. This is particularly true for implicit data structures: They simply do not exist independent of their accessing procedures. Separated from its code, an implicit data structure represents at best an unordered set of data. With the right code, it exhibits a rich structure, as is beautifully illustrated by the *heap* at the end of this chapter.

Array storage

A two-dimensional array declared as

```
var  A: array[1 .. m, 1 .. n] of elt;
```

is usually written in a rectangular shape:

A[1, 1]	A[1, 2]	…	A[1, n]
A[2, 1]	A[2, 2]	…	A[2, n]
…	…	…	…
A[m, 1]	A[m, 2]	…	A[m, n]

But it is stored in a linearly addressed memory, typically row by row (as shown below) or column by column (as in Fortran) in consecutive storage cells, starting at base address b. If an element fits into one cell, we have

	address
A[1, 1]	b
A[1, 2]	b + 1
…	…
A[1, n]	b + n − 1
A[2, 1]	b + n
A[2, 2]	b + n + 1
…	…
A[2, n]	b + 2 · n − 1
…	…
A[m, n]	b + m · n − 1

If an element of type 'elt' occupies c storage cells, the address α(i, j) of A[i, j] is

$$\alpha(i,j) = b + c \cdot (n \cdot (i-1) + j - 1).$$

This linear formula generalizes to k-dimensional arrays declared as

```
var  A: array[1 .. m₁, 1 .. m₂, … , 1 .. mₖ] of elt;
```

The address $\alpha(i_1, i_2, \ldots, i_k)$ of element $A[i_1, i_2, \ldots, i_k]$ is

$$\alpha(i_1, i_2, \ldots, i_k) = b + c \cdot (\quad (i_1 - 1) \cdot m_2 \cdot \ldots \cdot m_k + (i_2 - 1) \cdot m_3 \cdot \ldots \cdot m_k + \ldots + (i_{k-1} - 1) \cdot m_k + i_k - 1 \quad).$$

The point is that access to an element A[i, j, ...] invokes evaluation of a (linear) formula α(i, j, ...) that tells us where to find this element. A high-level programming language hides most of the details of address computation, except when we wish to take advantage of any special structure our matrices may have. The following types of *sparse matrices* occur frequently in numerical linear algebra.

Band matrices. An n × n matrix M is called a *band matrix of width 2 · b + 1* (b = 0, 1, ...) if $M_{i,j} = 0$ for all i and j with |i – j| > b. In other words, all nonzero elements are located on the main diagonal and in b adjacent minor diagonals on both sides of the main diagonal. If n is large and b is small, much space is saved by storing M in a two-dimensional array A with n · (2 · b + 1) cells rather than in an array with n^2 cells:

```
type  bandm = array[1 .. n, -b .. b] of elt;
var   A: bandm;
```

Each row A[i, ·] stores the nonzero elements of the corresponding row of M, namely the diagonal element $M_{i,i}$, the b elements to the left of the diagonal

$$M_{i,i-b},\ M_{i,i-b+1},\ \ldots,\ M_{i,i-1}$$

and the b elements to the right of the diagonal

$$M_{i,i+1},\ M_{i,i+2},\ \ldots,\ M_{i,i+b}.$$

The first and the last b rows of A contain empty cells corresponding to the triangles that stick out from M in Exhibit 20.1. The elements of M are stored in array A such that A[i, j] contains $M_{i,i+j}$ ($1 \le i \le n$, $-b \le j \le b$). A total of b · (b + 1) cells in the upper left and lower right of A remain unused. It is not worth saving an additional b · (b + 1) cells by packing the band matrix M into an array of minimal size, as the mapping becomes irregular and the formula for calculating the indices of $M_{i,j}$ becomes much more complicated.

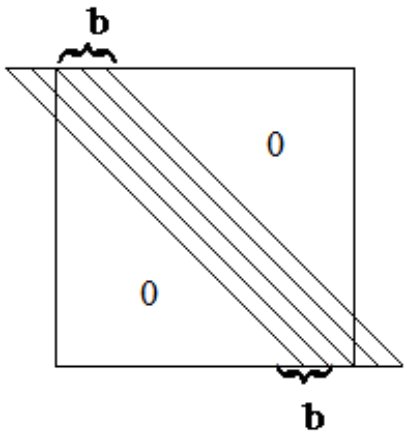

Exhibit 20.1: Extending the diagonals with dummy elements gives the band matrix the shape of a rectangular array.

Exercise: band matrices

(a) Write a procedure add(p, q: bandm; var r: bandm);
which adds two band matrices stored in p and q and stores the result in r.

(b) Write a procedure bmv(p: bandm; v: ... ; var w: ...);
which multiplies a band matrix stored in p with a vector v of length n and stores the result in w.

Solution

(a)
```
procedure add(p, q: bandm; var r: bandm);
var  i: 1 .. n;  j: -b .. b;
begin
   for  i := 1  to  n  do
      for  j := -b  to  b  do
         r[i, j] := p[i, j] + q[i, j]
end;
```

(b)
```
 type  vector = array[1 .. n] of real;

procedure bmv(p: bandm; v: vector; var w: vector);
var  i: 1 .. n;  j: -b .. b;
begin
   for  i := 1  to  n  do  begin
      w[i] := 0.0;
      for  j := -b  to  b  do
         if  (i + j ≥ 1) and (i + j ≤ n)  then  w[i] := w[i] + p[i, j] ·
v[i + j]
      end
   end;
```

Sparse matrices. A matrix is called *sparse* if it consists mostly of zeros. We have seen that sparse matrices of regular shape can be compressed efficiently using address computation. Irregularly shaped sparse matrices, on the other hand, do not yield gracefully to compression into a smaller array in such a way that access can be based on address computation. Instead, the nonzero elements may be stored in an unstructured set of records, where each record contains the pair ((i, j), A[i, j]) consisting of an index tuple (i, j) and the value A[i, j]. Any element that is absent from this set is assumed to be zero. As the position of a data element is stored explicitly as an index pair (i, j), this representation is not an implicit data structure. As a consequence, access to a random element of an irregularly shaped sparse matrix typically requires searching for it, and thus is likely to be slower than the direct access to an element of a matrix of regular shape stored in an implicit data structure.

Exercise: triangular matrices

Let A and B be lower-triangular n × n-matrices; that is, all elements above the diagonal are zero: $A_{i,j} = B_{i,j} = 0$ for $i < j$.

(a) Prove that the inverse (if it exists) and the matrix product of lower-triangular matrices are again lower-triangular.

(b) Devise a scheme for storing two lower-triangular matrices A and B in one array C of minimal size. Write a Pascal declaration for C and draw a picture of its contents.

(c) Write two functions

```
   function A(i, j: 1 .. n): real;
function B(i, j: 1 .. n): real;
```

(d) that access C and return the corresponding matrix elements.

(e) Write a procedure that computes A := A · B in place: The entries of A in C are replaced by the entries of the product A · B. You may use a (small) constant number of additional variables, independent of the size of A and B.

(f) Same as (d), but using A := A^{-1} · B.

Solution

(a) The inverse of an n × n-matrix exists iff the determinant of the matrix is non zero. Let A be a lower-triangular matrix for which the inverse matrix B exists, that is,

$$\det(A) = \prod_{i=1}^{n} A_{i,i} \neq 0 \quad \Rightarrow \quad \forall i, 1 \leq i \leq n: \quad A_{i,i} \neq 0$$

and

$$\sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} \quad = \quad \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

Let 1 ≤ j ≤ n. Then

$$j > 1 \Rightarrow \sum_{k=1}^{n} A_{1,k} \cdot B_{k,j} = A_{1,1} \cdot B_{1,j} = 0 \qquad \Rightarrow B_{1,j} = 0,$$

$$j > 2 \Rightarrow \sum_{k=1}^{n} A_{2,k} \cdot B_{k,j} = A_{2,1} \cdot B_{1,j} + A_{2,2} \cdot B_{2,j} = 0 \Rightarrow B_{2,j} = 0,$$

K

$$j > i \Rightarrow \ldots \qquad \Rightarrow B_{i,j} = 0,$$

and therefore B is a lower-triangular matrix.

Let A and B be lower-triangular, C := A · B:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} = \sum_{k=j}^{i} A_{i,k} \cdot B_{k,j} \cdot$$

If i < j, this sum is empty and therefore Ci,j = 0 (i. e. C is lower-triangular).

(b) A and B can be stored in an array C of size n · (n + 1) as follows (Exhibit 20.2):

```
const  n = … ;
var  C: array [0 .. n, 1 .. n] of real;
```

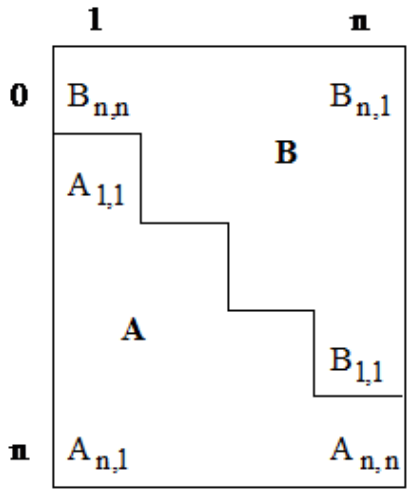

Exhibit 20.2: A staircase separates two triangular matrices

(c) stored in a rectangular array. (graphic does not match)

```
        function A(i, j: 1 .. n): real
  begin  if  i < j  then  return(0.0)  else  return(C[i, j])  end;
     function B(i, j: 1 .. n): real;
  begin  if  i < j  then  return(0.0)  else  return(C[n - i, n + 1 -
j])  end;
```

(d) Because the new elements of the result matrix C overwrite the old elements of A, it is important to compute them in the right order. Specifically, within every row i of C, elements $C_{i,j}$ must be computed from left to right, that is, in increasing order of j.

```
    procedure mult;
  var  i, j, k: integer;  x: real;
  begin
    for  i := 1  to  n  do
      for  j := 1  to  i  do  begin
        x := 0.0;
        for  k := j  to  i  do  x := x + A(i, k) · B(k, j);
        C[i, j] := x
      end
  end;
```

(e)

```
      procedure invertA;
  var  i, j, k: integer;  x: real;
  begin
      for  i := 1  to  n  do  begin
      for  j := 1  to  i - 1  do  begin
        x := 0.0;
        for  k := j  to i - 1  do  x := x - C[i, k] · C[k, j];
```

```
            C[i, j] := x / C[i, i]
          end;
            C[i, i] := 1.0 / C[i, i]
    end
    end;
procedure AinvertedmultB;
begin  invertA;  mult  end;
```

Implementation of the fixed-length fifo queue as a circular buffer

A fifo queue is needed in situations where two processes interact in the following way. A process called *producer* generates data for a process called *consumer*. The processes typically work in bursts: The producer may generate a lot of data while the consumer is busy with something else; thus the data has to be saved temporarily in a buffer, from which the consumer takes it as needed. A keyboard driver and an editor are an example of this producer-consumer interaction. The keyboard driver transfers characters generated by key presses into the buffer, and the editor reads them from the buffer and interprets them (e.g. as control characters or as text to be inserted). It is worth remembering, though, that a buffer helps only if two processes work at about the same speed over the long run. If the producer is always faster, any buffer will overflow; if the consumer is always faster, no buffer is needed. A buffer can equalize only *temporary* differences in speeds.

With some knowledge about the statistical behavior of producer and consumer one can usually compute a buffer size that is sufficient to absorb producer bursts with high probability, and allocate the buffer statically in an array of fixed size. Among statically allocated buffers, a *circular buffer* is the natural implementation of a fifo queue.

A circular buffer is an array B, considered as a ring in which the first cell B[0] is the successor of the last cell B[m – 1], as shown in Exhibit 20.3. The elements are stored in the buffer in consecutive cells between the two pointers 'in' and 'out': 'in' points to the empty cell into which the next element is to be inserted; 'out' points to the cell containing the next element to be removed. A new element is inserted by storing it in B[in] and advancing 'in' to the next cell. The element in B[out] is removed by advancing 'out' to the next cell.

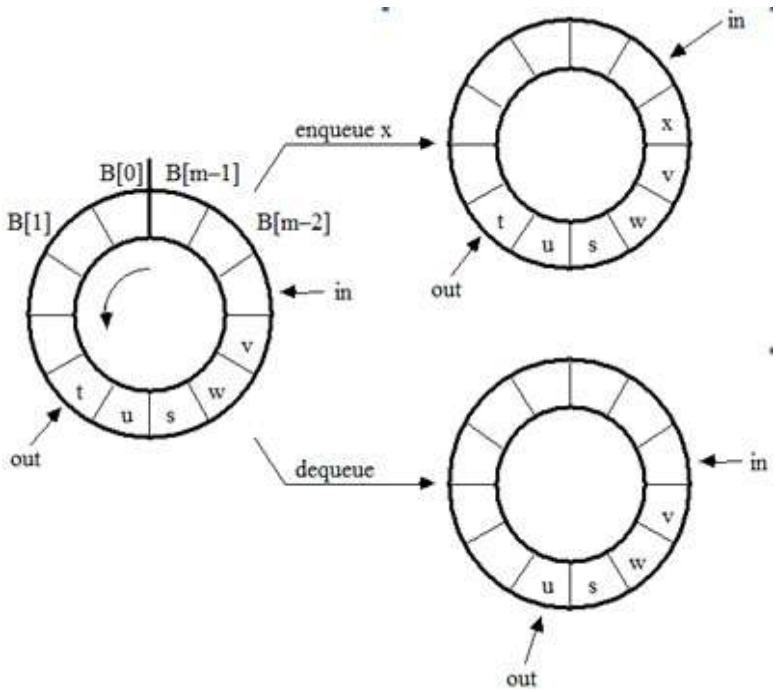

Exhibit 20.3: Insertions move the pointer 'in', deletions the pointer 'out' counterclockwise around the array.

Notice that the pointers 'in' and 'out' meet both when the buffer gets full and when it gets empty. Clearly, we must be able to distinguish a full buffer from an empty one, so as to avoid insertion into the former and removal from the latter. At first sight it appears that the pointers 'in' and 'out' are insufficient to determine whether a circular buffer is full or empty. Thus the following implementation uses an additional variable n, which counts how many elements are in the buffer.

```
const  m = … ;  { length of buffer }
type  addr = 0 .. m − 1;  { index range }
var  B: array[addr] of elt;  {storage}
   in, out: addr;  { access to buffer }
   n: 0 .. m;  { number of elements currently in buffer }

procedure create;
begin  in := 0;  out := 0;  n := 0  end;

function empty(): boolean;
begin  return(n = 0)  end;

function full(): boolean;
begin  return(n = m)  end;

procedure enqueue(x: elt);
{ not to be called if the queue is full }
begin  B[in] := x;  in := (in + 1) mod m;  n := n + 1  end;

function front(): elt;
{ not to be called if the queue is empty }
begin  return(B[out])  end;

procedure dequeue;
{ not to be called if the queue is empty }
begin  out := (out + 1) mod m;  n := n − 1  end;
```

The producer uses only 'enqueue' and 'full', as it deletes no elements from the circular buffer. The consumer uses only 'front', 'dequeue', and 'empty', as it inserts no elements.

The state of the circular buffer is described by its contents and the values of 'in', 'out', and n. Since 'in' is changed only within 'enqueue', only the producer needs write-access to 'in'. Since 'out' is changed only by 'dequeue', only the consumer needs write-access to 'out'. The variable n, however, is changed by both processes and thus is a *shared variable* to which both processes have write-access (Exhibit 20.4 (a)).

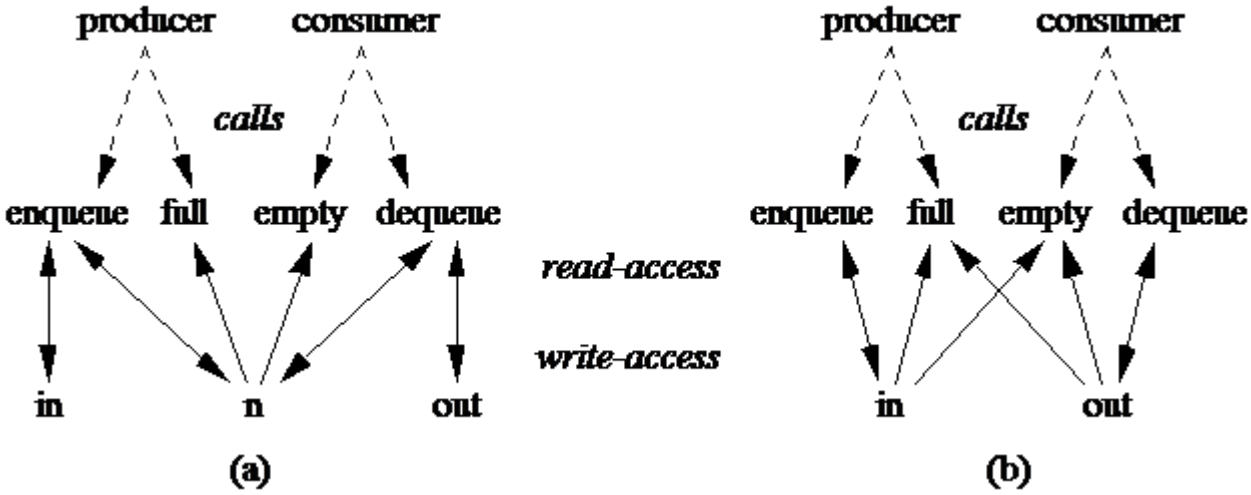

Exhibit 20.4:

(a) Producer and consumer both have write-access to shared variable n.

(b) The producer has read/write-access to 'in' and read-only-access to 'out',
the consumer has read/write-access to 'out' and read-only-access to 'in'.

In a concurrent programming environment where several processes execute independently, access to shared variables must be synchronized. Synchronization is overhead to be avoided if possible. The shared variable n becomes superfluous (Exhibit 20.4 (b)) if we use the time-honored trick of leaving at least one cell free as a *sentinel*. This ensures that 'empty' and 'full', when expressed in terms of 'in' and 'out', can be distinguished. Specifically, we define 'empty' as in = out, and 'full' as (in + 1) mod m = out. This leads to an elegant and more efficient implementation of the *fixed-length fifo queue* by a circular buffer:

```
const  m = … ;  { length of buffer }
type addr = 0 .. m – 1;  { index range }
  fifoqueue = record
       B: array[addr] of elt;  { storage }
       in, out: addr  { access to buffer }
    end;

procedure create(var f: fifoqueue);
begin  f.in := 0;  f.out := 0  end;

function empty(f: fifoqueue): boolean;
begin  return(f.in = f.out)  end;

function full(f: fifoqueue): boolean;
begin  return((f.in + 1) mod m = f.out)  end;

procedure enqueue(var f: fifoqueue; x: elt);
{ not to be called if the queue is full }
begin  f.B[f.in] := x;  f.in := ( f.in + 1) mod m  end;
```

```
function front(f: fifoqueue): elt;
{ not to be called if the queue is empty }
begin  return(f.B[f.out])  end;

procedure dequeue(f: fifoqueue);
{ not to be called if the queue is empty }
begin  f.out := (f.out + 1) mod m  end;
```

Implementation of the fixed-length priority queue as a heap

A *fixed-length priority queue* can be realized by a circular buffer, with elements stored in the cells between 'in' and 'out', and ordered according to their priority such that 'out' points to the element with highest priority (Exhibit 20.5). In this implementation, the operations 'min' and 'delete' have time complexity O(1), since 'out' points directly to the element with the highest priority. But insertion requires finding the correct cell corresponding to the priority of the element to be inserted, and shifting other elements in the buffer to make space. Binary search could achieve the former task in time O(log n), but the latter requires time O(n).

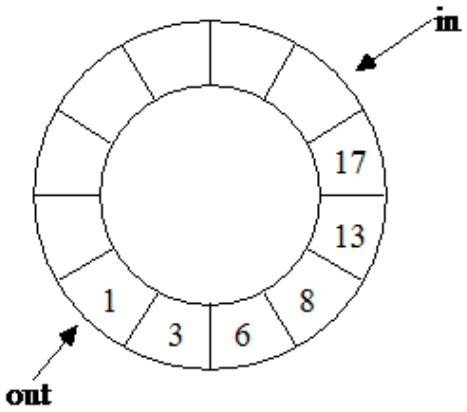

Exhibit 20.5: Implementing a fixed-length priority queue by a circular buffer.
Shifting elements to make space for a new element costs O(n) time.

Implementing a priority queue as a linear list, with elements ordered according to their priority, does not speed up insertion: Finding the correct position of insertion still requires time O(n) (Exhibit 20.6).

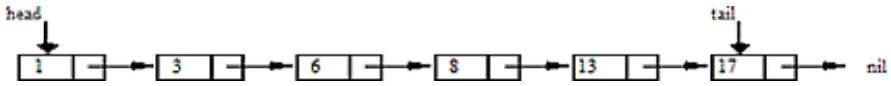

Exhibit 20.6: Implementing a fixed-length priority queue by a linear list. Finding the correct position for a new element costs O(n) time.

The *heap* is an elegant and efficient data structure for implementing a priority queue. It allows the operation 'min' to be performed in time O(1) and allows both 'insert' and 'delete' to be performed in worst-case time O(log n). A heap is a binary tree that:

- obeys a structural property
- obeys an order property
- is embedded in an array in a certain way

Structure: The binary tree is as balanced as possible; all leaves are at two adjacent levels, and the nodes at the bottom level are located as far to the left as possible (Exhibit 20.7).

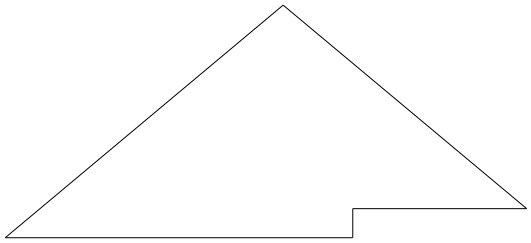

Exhibit 20.7: A heap has the structure of an almost complete binary tree.

Order: The element assigned to any node is ≤ the elements assigned to any children this node may have (Exhibit 20.8).

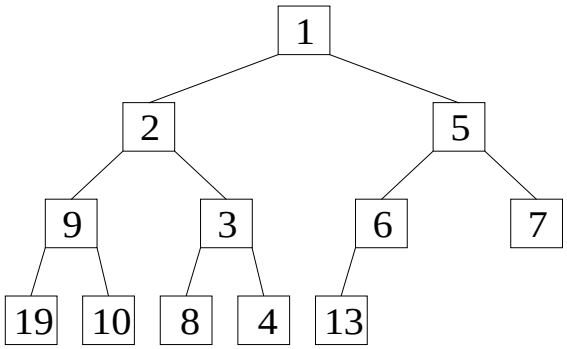

Exhibit 20.8: The order property implies that the smallest element is stored at the root.

The order property implies that the smallest element (the one with top priority) is stored in the root. The 'min' operation returns its value in time O(1), but the most obvious way to delete this element leaves a hole, which takes time to fill. How can the tree be reorganized so as to retain the structural and the order property? The structural condition requires the removal of the rightmost node on the lowest level. The element stored there–13 in our example–is used (temporarily) to fill the vacuum in the root. The root may now violate the order condition, but the latter can be restored by sifting 13 down the tree according to its weight (Exhibit 20.9). If the order condition is violated at any node, the element in this node is exchanged with the smaller of the elements stored in its children; in our example, 13 is exchanged with 2. This *sift-down process* continues until the element finds its proper level, at the latest when it lands in a leaf.

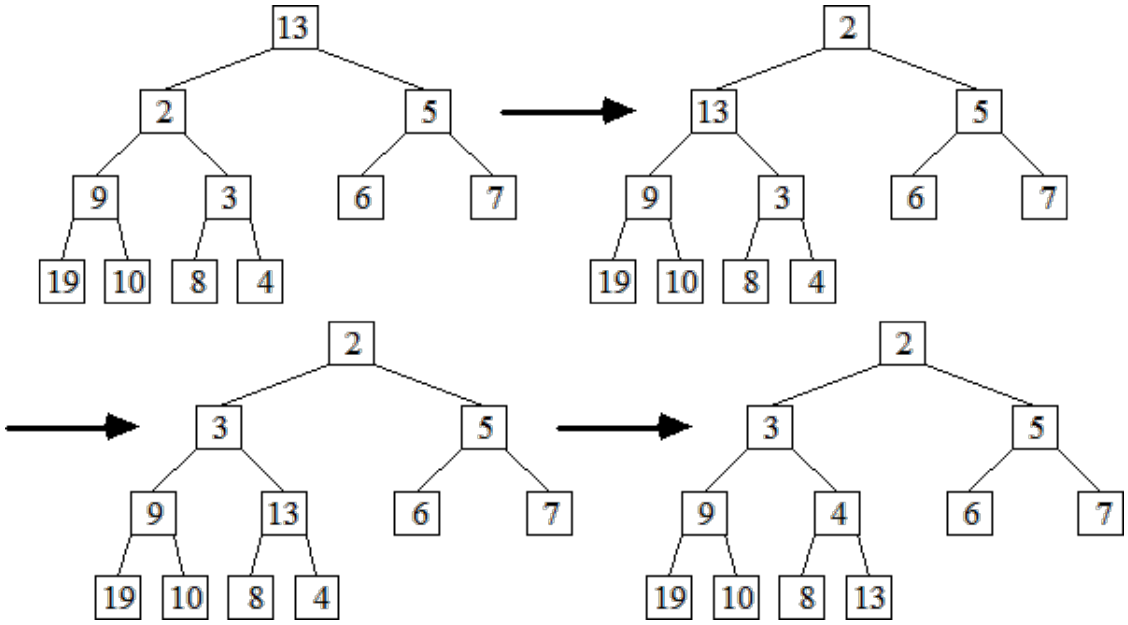

Exhibit 20.9: Rebuilding the order property of the tree in Exhibit 20.8 after 1 has been removed and 13 has been moved to the root.

Insertion is handled analogously. The structural condition requires that a new node is created on the bottom level at the leftmost empty slot. The new element - 0 in our example - is temporarily stored in this node (Exhibit 20.10). If the parent node now violates the order condition, we restore it by floating the new element upward according to its weight. If the new element is smaller than the one stored in its parent node, these two elements - in our example 0 and 6 - are exchanged. This *sift-up process* continues until the element finds its proper level, at the latest when it surfaces at the root.

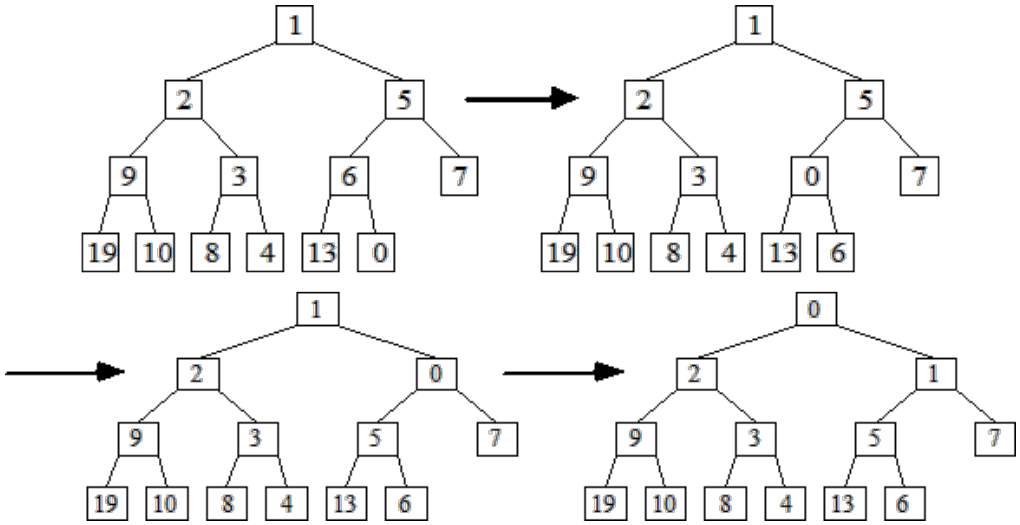

Exhibit 20.10: Rebuilding the order property of the tree in Exhibit 20.8 after 0 has been inserted in a new rightmost node on the lowest level.

The number of steps executed during the sift-up process and the sift-down process is at most equal to the height of the tree. The structural condition implies that this height is $[\log_2 n]$. Thus both 'insert' and 'delete' in a heap work in time O(log n).

A binary tree can be implemented in many different ways, but the special class of trees that meets the structural condition stated above has a particularly efficient array implementation. A *heap* is a binary tree that satisfies the structural and the order condition and is embedded in a linear array in such a way that the children of a node with index i have indices 2 · i and 2 · i + 1 (Exhibit 20.11). Thus the parent of a node with index j has index j div 2. Any subtree of a heap is also a heap, although it may not be stored contiguously. The order property for the heap implies that the elements stored at indices 2 · i and 2 · i + 1 are ≥ the element stored at index i. This order is called the *heap order*.

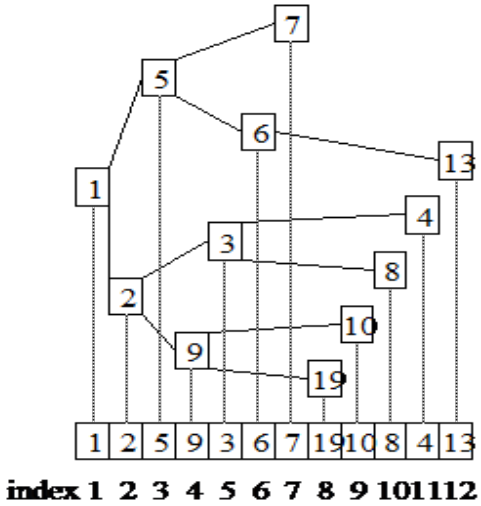

Exhibit 20.11: Embedding the tree of Exhibit 20.8 in a linear array.

The procedure 'restore' is a useful tool for managing a heap. It creates a heap out of a binary tree embedded in a linear array h that satisfies the structural condition, provided that the two subtrees of the root node are already heaps. Procedure 'restore' is applied to subtrees of the entire heap whose nodes are stored between the indices L and R and whose tree structure is defined by the formulas 2 · i and 2 · i + 1.

```
const  m = … ;  { length of heap }
type  addr = 1 .. m;
var  h: array[addr] of elt;

procedure restore(L, R: addr);
var  i, j: addr;
begin
   i := L;
   while  i ≤ (R div 2)  do  begin
      if  (2 · i < R) cand (h[2 · i + 1] < h[2 · i])  then  j := 2 · i +
1  else  j := 2 · i;
      if  h[j] < h[i]  then  { h[i] :=: h[j];  i := j }  else  i := R
   end
end;
```

Since 'restore' operates along a single path from the root to a leaf in a tree with at most R – L nodes, it works in time O(log (R – L)).

Creating a heap

An array h can be turned into a heap as follows: for i := n div 2 down to 1 do restore(i, n);

This is more efficient than repeated insertion of a single element into an existing heap. Since the for loop is executed n div 2 times, and $n - i \le n$, the time complexity for creating a heap with n elements is $O(n \cdot \log n)$. A more careful analysis shows that the time complexity for creating a heap is O(n).

Heap implementation of the fixed-length priority queue

```
const  m = … ;  { maximum length of heap }
type addr = 1 .. m;
  priorityqueue =  record
       h: array[addr] of elt;  { heap storage }
       n: 0 .. m  { current number of elements }
     end;

procedure restore(var h: array[addr] of elt; L, R: addr);
begin  …  end;

procedure create(var p: priorityqueue);
begin  p.n := 0  end;

function empty(p: priorityqueue): boolean;
begin  return(p.n = 0)  end;

function full(p: priorityqueue): boolean;
begin  return(p.n = m)  end;

procedure insert(var p: priorityqueue; x: elt);
{ not to be called if the queue is full }
var  i: 1 .. m;
begin
  p.n := p.n + 1;  p.h[p.n] := x;  i := p.n;
  while  (i > 1) cand (p.h[i] < p.h[i div 2])  do
    { p.h[i] :=: p.h[i div 2];  i := i div 2 }
end;

function min(p: priorityqueue): elt;
{ not to be called if the queue is empty }
begin  return(p.h[1])  end;

procedure delete(var p: priorityqueue);
{ not to be called if the queue is empty }
begin  p.h[1] := p.h[p.n];  p.n := p.n - 1;  restore(p.h, 1, p.n)
end;
```

Heapsort

The heap is the core of an elegant $O(n \cdot \log n)$ sorting algorithm. The following procedure 'heapsort' sorts n elements stored in the array h into decreasing order.

```
procedure heapsort(n: addr);  { sort elements stored in h[1 .. n] }
var  i: addr;
begin  { heap creation phase: the heap is built up }
  for  i := n div 2  downto  1  do  restore(i, n);
  { shift-up phase: elements are extracted from heap in increasing
order }
  for  i := n  downto  2  do  { h[i] :=: h[1];  restore(1, i - 1) }
end;
```

Each of the for loops is executed less than n times, and the time complexity of restore is O(log n). Thus heapsort always works in time $O(n \cdot \log n)$.

Exercises and programming projects

1. Block-diagonal matrices are composed of smaller matrices that line up along the diagonal and have 0 elements everywhere else, as shown in Exhibit 20.12. Show how to store an arbitrary block-diagonal matrix in a minimal storage area, and write down the corresponding address computation formulas.

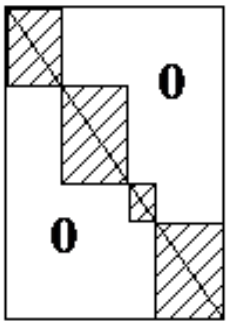

Exhibit 20.12: Structure of a block-diagonal matrix.

2. Let A be an antisymmetric n × n-matrix (i. e., all elements of the matrix satisfy $A_{ij} = -A_{ji}$).
   (a) What values do the diagonal elements A_{ii} of the matrix have?
   (b) How can A be stored in a linear array c of minimal size? What is the size of c?
   (c) Write a

   ```
   function A(i, j: 1 .. n): real;
   ```

   which returns the value of the corresponding matrix element.
3. Show that the product of two n × n matrices of width 2 · b + 1 (b = 0, 1, ...) is again a band matrix. What is the width of the product matrix? Write a procedure that computes the product of two band matrices both having the same width and stores the result as a band matrix of minimal width.
4. Implement a double-ended queue (deque) by a circular buffer.
5. What are the minimum and maximum numbers of elements in a heap of height h?
6. Determine the time complexities of the following operations performed on a heap storing n elements. (a) Searching any element. (b) Searching the largest element (i.e. the element with lowest priority).
7. Implement heapsort and animate the sorting process, for example as shown in the snapshots in “Algorithm animation”. Compare the number of comparisons and exchange operations needed by heapsort and other sorting algorithms (e.g. quicksort) for different input configurations.
8. What is the running time of heapsort on an array h[1 .. n] that is already sorted in increasing order? What about decreasing order?
9. In a k-ary heap, nodes have k children instead of 2 children.
   (a) How would you represent a k-ary heap in an array?
   (b) What is the height of a k-ary heap in terms of the number of elements n and k?
   (c) Implement a priority queue by a k-ary heap. What are the time complexities of the operations 'insert' and 'delete' in terms of n and k?