Another popular sorting algorithm is heapsort. It runs in time $O(n \log n)$, even in the worst case, and can be implemented such that it takes only a constant amount of memory in addition to the array it sorts. Heapsort uses a heap, which is a data structure for storing a prioritized queue.

**Priority Queues.** A data structure implements the *priority queue* abstract data type if it supports at least the following operations: INSERT, FINDMIN and DELETEMIN. The operations are applied to a set of items with priorities. The priorities are totally ordered so any two can be compared. To avoid any confusion we will usually refer to the priorities as ranks. We will always use integers as priorities and follow the convention that smaller ranks represent higher priorities. In many applications FINDMIN and DELETEMIN are combined:

```plaintext
item ExtractMin
r = FindMin; DeleteMin; return r.
```

Function ExtractMin removes and returns the item with smallest rank.

**Heap.** A heap is a particularly compact priority queue. We can think of it as a binary tree with items stored in the nodes, as in Figure 6. Each level is full except possibly the last level which is filled from left to right until we run out of items. The items are stored in heap-order: every node $\mu$ has a rank larger than or equal to the rank of its parent. Symmetrically, $\mu$ has a rank less than or equal to the ranks of both its children. As a consequence, the root contains the item with smallest rank.

We store the nodes of the tree in a table, level by level from top down, and each level from left to right, as shown in Figure 7. The embedding in the table saves explicit pointers establishing parent-child relations. Specifically, we can find the children and parent of a node by index computation: the left child of $A[i]$ is $A[2i]$, the right child is $A[2i + 1]$, and the parent is $A[\lfloor i/2 \rfloor]$. The item with minimum rank is stored in the first position:

```plaintext
item FindMin(int n)
assert n >= 1; return A[1].
```

Since the index at least doubles each step, a path can have at most $\log_2 n$ edges.

**Deleting the Minimum.** We first study the problem of repairing the heap-order if it is violated at the root. Let $n$ be the length of the table. We repair the heap-order by a sequence of swaps along a single path. Each swap is done...
between an item \( A[i] \) and the smaller of its children, \( A[k] \), assuming \( A[k] \) is even smaller than \( A[i] \):

```c
void DNEAP(int i, n)
if 2i \leq n then
    if 2i = n then k = 2i
    else if \( A[2i + 1] < A[2i] \) then k = 2i + 1
    else k = 2i
endif
endif
if A[k] < A[i] then SWAP(i, k);
    DNEAP(k, n)
endif
endfor
```

Since a path has at most \( \log_2 n \) edges, the time to repair the heap-order takes time at most \( O(\log n) \). To delete the minimum we overwrite the root with the last item, shorten the heap, and repair the heap-order:

```c
void DELETEMIN(int n)
```

**Inserting an Item.** Consider repairing the heap-order if it is violated at the last position of the heap. In this case the item moves up the heap until it reaches a position where its rank is larger than that of its parent.

```c
void UPEAP(int i)
if i \geq 2 then k = [i/2];
    if A[i] < A[k] then SWAP(i, k);
        UPEAP(k)
endif
endif
```

An item is added by first expanding the heap by one item, placing the new item in the position that just opened up, and repairing the heap-order.

```c
void INSERT(int n, item r)
A[n] = r; UPEAP(n).
```

A heap supports FINDMIN in constant time and INSERT, DELETEMIN in time \( O(\log n) \) each. Since items are stored in a table, we need to know how big the heap can grow ahead of time. Later in this course we will learn about a priority queue that is faster for insertions and that does not require advance knowledge of the number of items.

**Sorting by Selection.** Priority queues can be used for sorting. The first step throws all items into the priority queue, and the second step takes them out in order:

```c
for i = 1 to n do INSERT(i, r) endfor;
for i = 1 to n do PRINT(EXTRACTMIN) endfor.
```

This is sorting by selection: at each step we find the minimum and remove it from the pool. Using a heap we can select in time \( O(\log n) \) and therefore complete the second for-loop in time \( O(n \log n) \). The heap can be constructed by repeated insertion. Assuming the items are already stored in the array this can be done by repeated heap repair:

```c
for i = 1 to n do UPEAP(i) endfor.
```

In the worst case, the \( i \)-th item moves up all the way to the root. The number of exchanges is therefore at most

\[
\sum_{i=1}^{n} \log_2 i \leq n \log_2 n.
\]

The upper bound is asymptotically tight because half the terms in the sum are at least \( \log_2 \frac{n}{2} = \log_2 n - 1 \).

**HeapSort.** It is also possible to construct the initial heap in time \( O(n) \) by building it from bottom to top. We modify the first loop accordingly, and we change the second loop such that items are stored rather than printed in sorted order.

```c
void HEAPSORT(int n)
for i = n downto 1 do DNEAP(i, n) endfor;
for i = n downto 1 do
    SWAP(i, 1); DNEAP(1, i - 1)
endfor.
```

At each step of the first for-loop we consider the subtree with root \( A[i] \). At this moment the items in the left and right subtrees rooted at \( A[2i] \) and \( A[2i + 1] \) are already heaps. We can therefore use one call to Function DNEAP to make the subtree with root \( A[i] \) a heap. We will prove shortly that this bottom-up construction of the heap takes time only \( O(n) \). Figure 8 shows the array after each iteration of the second for-loop. Note how the heap gets smaller by one item each step. A single downheap operation takes time \( O(\log n) \), and in total HeapSort takes time \( O(n \log n) \). In addition to the input table, HeapSort uses a constant number of variables and memory for the
We can save the memory for the stack by writing Function DHEAP as an iteration. The sort can be changed to non-decreasing order by reversing the order of items in the heap.

Three ways to sum. We return to proving that the bottom-up approach to constructing a heap takes only $O(n)$ time. We simplify the analysis by assuming $n = 2^\ell - 1$. Then the heap is a binary tree with $\ell$ full levels. There is no exchange for the roots in the last level, at most one exchange for the roots in the second to the last level, at most two exchanges for the roots in the third to the last level, etc. The total number of exchanges is therefore at most

$$X_\ell = 0 \cdot 2^{\ell-1} + 1 \cdot 2^{\ell-2} + \ldots + (\ell - 1) \cdot 2^0$$

$$= \sum_{i=0}^{\ell-1} i \cdot 2^{\ell-i-1}.$$ 

We prove below that $X_\ell = 2^\ell - \ell - 1$. It follows that the number of exchanges is at most $2^\ell - \ell - 1$, which is less than $n$.

Proof by Induction. Induction is a mechanical way to prove the equation. It requires we know or we conjecture the result. First we establish the equation for the smallest possible value of $\ell$. For $\ell = 1$ we have $X_1 = 0$ and $2^1 - 1 - 1 = 0$, which is correct. Now assume that $X_\ell = 2^\ell - \ell - 1$, which is called the induction hypothesis. To prove that the equation also holds for $\ell + 1$ we note that adding a new level increases the count for each old node by one, while the contribution of every new node is zero. Hence, $X_{\ell+1} = X_\ell + 2^{\ell-1} - 1$. Using the induction hypothesis, we get $X_{\ell+1} = 2^{\ell+1} - (\ell + 1) - 1$, as required. We have thus shown that if the equation is true for $X_\ell$ then it is true for $X_{\ell+1}$. It follows the equation is true for any number of levels.

Index Transformation. But what if we do not know the answer? We may do index transformations to manipulate the sum and hopefully find a closed form expression. The most common such transformation writes $i - 1$ for $i$:

$$x_\ell = \sum_{i=1}^{\ell} (i-1) \cdot 2^{\ell-i}$$

$$= 2 \cdot \sum_{i=1}^{\ell} i \cdot 2^{\ell-i-1} - \sum_{i=1}^{\ell} 2^{\ell-i}$$

$$= 2X_\ell + \ell - (2^\ell - 1).$$

Hence $X_\ell = 2^\ell - \ell - 1$.

Proof by Picture. Finally, we can figure out the answer by drawing the right picture. Draw the heap as a tree with $n = 2^\ell - 1$ nodes. The number of edges is $n - 1 = 2^\ell - 2$. For each node, take the edge to the right child and complete the path by taking edges to left children down to the bottom level. This is illustrated in Figure 9. The paths cover each edge exactly once, except