Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe und Sarah Winkler

10 June 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 11

Exercise 1 (Building Heaps)

A heap can be constructed from an array A of size n in two ways, either top-down or bottom-up.

The *top-down* method starts by considering $A[1]$ as a heap. In the $i+1$-th iteration $A[1,\ldots,i]$ is assumed to be a heap and $A[i+1]$ is added, i.e. pulled up until the heap condition holds. This yields the extended heap $A[1,\ldots,i+1]$. The approach is iterated until $i = n$, so the whole array is turned into a heap.

In the *bottom-up* method, one starts with $i = \lfloor n/2 \rfloor$. In each iteration all subtrees in $A[i+1,\ldots,n]$ are assumed to satisfy the heap condition. $A[i]$ has one or two children at positions $A[2i]$ and $A[2i+1]$, which are by assumption the roots of valid heaps. After sinking $A[i]$, all subtrees in $A[i,\ldots,n]$ are heaps. By decreasing i this approach is repeated until $i = 1$, so the whole array satisfies the heap condition.

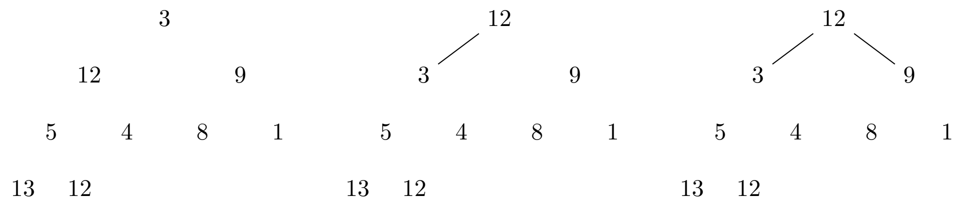
Now consider the example array

$$A = [3, 12, 9, 5, 4, 8, 1, 13, 12]$$

and perform the following operations (on paper):

a) Construct the initial heap for A in a *top-down* fashion.

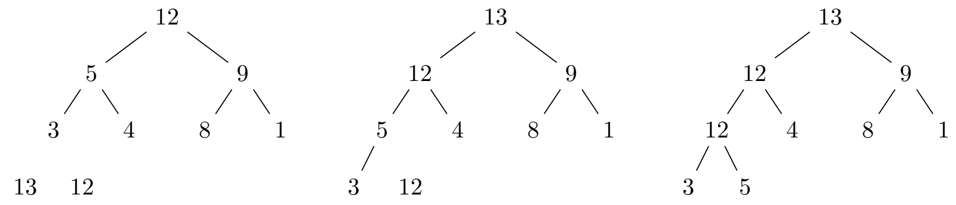
Solution.



initially, only 3 is in the heap

12 is added and pulled to the root

adding 9 does not destroy the heap property



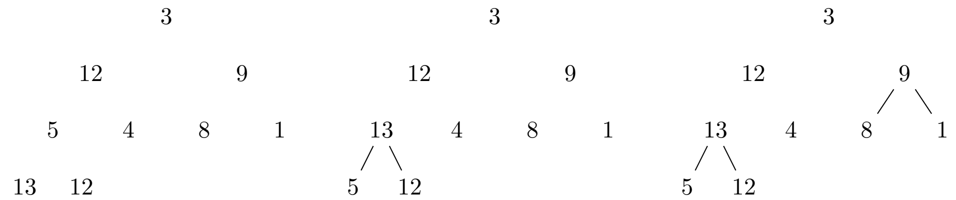
5 is added and pulled up one level; adding 4, 8 and 1 does not affect the heap property

13 gets drawn up to the root

finally, adding 12 requires to pull it up one level

b) Construct the initial heap for A using the *bottom-up* approach.

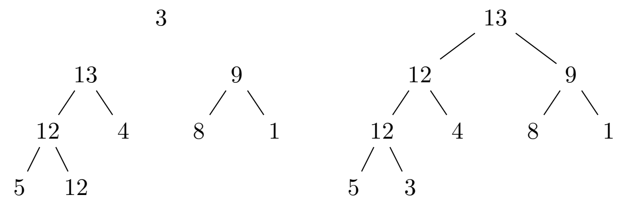
Solution.



initially, all leaves are in the heap

5 is added and sunk down

adding 9 does not destroy the heap property



when adding 12, it sinks one level down

after adding and sinking 3 the heap is established

c) Apply Heapsort to one of the heaps obtained above. Is Heapsort stable?

Solution.

One obtains the following sequence of arrays (the **bold** values are already fixed):

	$[13, 12, 9, 12, 4, 8, 1, 5, 3]$	initial array from heap in b)
$i = 9$	$[12, 12, 9, 5, 4, 8, 1, 3, \mathbf{13}]$	swap 13 and 3, sink 3
$i = 8$	$[12, 5, 9, 3, 4, 8, 1, \mathbf{12}, \mathbf{13}]$	swap 12 and 3, sink 3
$i = 7$	$[9, 5, 8, 3, 4, 1, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 12 and 1, sink 1
$i = 6$	$[8, 5, 1, 3, 4, \mathbf{9}, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 9 and 1, sink 1
$i = 5$	$[5, 4, 1, 3, \mathbf{8}, \mathbf{9}, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 8 and 4, sink 4
$i = 4$	$[4, 3, 1, \mathbf{5}, \mathbf{8}, \mathbf{9}, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 5 and 3, sink 3
$i = 3$	$[3, 1, \mathbf{4}, \mathbf{5}, \mathbf{8}, \mathbf{9}, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 4 and 1, sink 1
$i = 2$	$[1, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{8}, \mathbf{9}, \mathbf{12}, \mathbf{12}, \mathbf{13}]$	swap 3 and 1

The two elements with key 12 are exchanged during sorting, thus Heapsort is not stable.

Exercise 2 (Combining Heaps)

Provide pseudo-code for an algorithm to build one heap that contains all elements of two given heaps with n and m elements, respectively (where n and m are positive). Assume that the heaps are given in a tree representation, i.e. each node has links to its two children. The running time of the algorithm should be $O(\log(n+m))$ in the worst case.

Solution.
The idea is to remove a leaf from h_1 and use it as a root having the two given heaps as children. This requires $O(\log n)$ comparisons. To establish the heap condition, the root element has to be sunk into the heap with $n+m$ elements which takes at most $O(\log(n+m))$ comparisons.

For details, see Listing **??**.

Exercise 3 (Heapsort in C)

Implement functions *sink* and *buildHeap* in C, and use them to incorporate *heapsort* into the framework of Exercise 4 of last week.

Solution.
See *sort.c*.

Exercise 4 (Lower Bound for Searching)

In the lecture you used decision trees to derive an information-theoretic lower bound for comparison-based sorting: Given a comparison operation that can check for two elements a and b whether $a \leq b$ or $a > b$ holds, it was shown that any sorting algorithm using only such an operation requires $\Omega(n \log n)$ comparisons.

Use the same technique to show that searching a value in a sorted array requires $\Omega(\log(n))$ comparisons.

Solution.
When searching a specific key in an array with n elements, there are $n+1$ possible outcomes: either the element at some position between 1 and n matches the key, or no such element is found. Thus a decision tree realizing search in a sorted array needs to have $n+1$ leaves, which requires it to have depth $\Omega(\log(n))$.

Listing 1 Combining Heaps

```
function combine(h1, h2 : ^ heap) : ^ heap
begin
   root := deleteLeaf(&h1);                          /* delete some leaf */
   root^.left := h1;                                 /* place it as new root */
   root^.right := h2;
   sink(root);                                       /* let the root sink into the heap */
   return root;
end

function deleteLeaf(Rp : ^^ heap) : ^ heap
begin
   P := nil;
   R := Rp^;
   while R^.left ≠ nil or R^.right ≠ nil do
      P := R;
      if R^.left ≠ nil then
         R := R^.left;
      else
         R := R^.right;
   if P ≠ nil then
      if P^.left = R then
         P^.left := nil;                             /* delete left child */
      else
         P^.right := nil;                            /* delete right child */
   else
      Rp^ = nil;                                     /* root deleted – heap is empty now */
   return R;
end

procedure sink(R : ^ heap)
begin
   while (R^.left ≠ nil and R^.left^.key > R^.key) or
            (R^.right ≠ nil and R^.right^.key > R^.key) do
      if R^.left = nil or (R^.right ≠ nil and R^.right^.key > R^.left^.key) then
         max := R^.right^.key                        /* sink to the right – swap keys */
         R^.right^.key := R^.key
         R^.key := max
         R := R^.right                               /* continue sinking at R */
      else
         max := R^.left                              /* sink to the left – swap keys */
         R^.left^.key := R^.key
         R^.key := max
         R := R^.left                                /* continue sinking at R */
end
```