HEAP DATA STRUCTURE

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

Spalio 20 d., 2023

Heap

Heap is a specialized binary tree-based data structure that satisfies the following conditions:

1. A heap is an almost complete binary tree: the first element is stored in the root.

   Then the next level is filled from the left to the right.

   The new level is started only when the previous level is fully filled.

Heap

Heap is a specialized binary tree-based data structure that satisfies the following conditions:

1. A heap is an almost complete binary tree: the first element is stored in the root.

   Then the next level is filled from the left to the right.

   The new level is started only when the previous level is fully filled.
2. A heap can be regarded as being partially ordered. Children of each vertex are not larger than the vertex itself.

Heap

Heap is a specialized binary tree-based data structure that satisfies the following conditions:

1. A heap is an almost complete binary tree: the first element is stored in the root.

   Then the next level is filled from the left to the right.

   The new level is started only when the previous level is fully filled.
2. A heap can be regarded as being partially ordered. Children of each vertex are not larger than the vertex itself.

Heap

Heap is a specialized binary tree-based data structure that satisfies the following conditions:

1. A heap is an almost complete binary tree: the first element is stored in the root.

   Then the next level is filled from the left to the right.

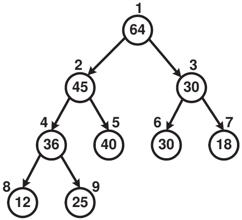
   The new level is started only when the previous level is fully filled.
2. A heap can be regarded as being partially ordered. Children of each vertex are not larger than the vertex itself.

It is easy to show that the highest priority element is always stored at the root.

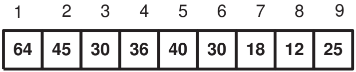
Since a heap is a complete binary tree, it is very convenient to store its data in an array.

Since a heap is a complete binary tree, it is very convenient to store its data in an array.

An example of a heap is given in the following figure:



a)

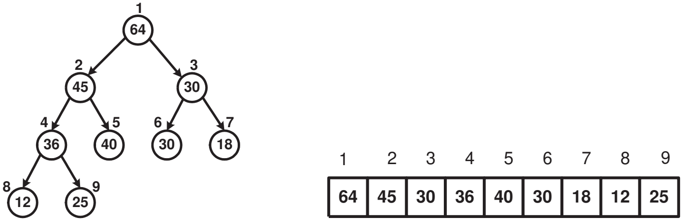


1	2	3	4	5	6	7	8	9
64	45	30	36	40	30	18	12	25

b)

Since a heap is a complete binary tree, it is very convenient to store its data in an array.

An example of a heap is given in the following figure:



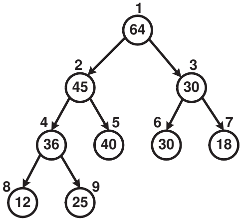
1	2	3	4	5	6	7	8	9
64	45	30	36	40	30	18	12	25

a) b)

Given a node a_i at index i, its children are at indices $2i$ and $2i + 1$ (elements a_{2i} and a_{2i+1}).

Since a heap is a complete binary tree, it is very convenient to store its data in an array.

An example of a heap is given in the following figure:



a)

1	2	3	4	5	6	7	8	9
64	45	30	36	40	30	18	12	25

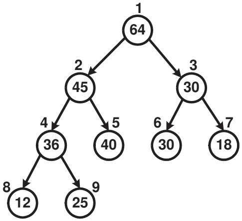
b)

Given a node a_i at index i, its children are at indices $2i$ and $2i+1$ (elements a_{2i} and a_{2i+1}).

A parent of vertex a_i is at index $j = \lfloor i/2 \rfloor$ (element a_j).

Since a heap is a complete binary tree, it is very convenient to store its data in an array.

An example of a heap is given in the following figure:



a)

1	2	3	4	5	6	7	8	9
64	**45**	**30**	**36**	**40**	**30**	**18**	**12**	**25**

b)

Given a node a_i at index i, its children are at indices $2i$ and $2i+1$ (elements a_{2i} and a_{2i+1}).

A parent of vertex a_i is at index $j = \lfloor i/2 \rfloor$ (element a_j).

This simple indexing scheme makes it efficient to move "up" or "down" the binary tree.

Construction of a heap.

We have data $e_1, e_2, \ldots, e_N$. Initially, these elements are stored in array A without any changes.

Construction of a heap.

We have data $e_1, e_2, \ldots, e_N$. Initially, these elements are stored in array A without any changes.

Balancing of a heap is done by sift-down operation (swapping elements which are out of order).

Construction of a heap.

We have data $e_1, e_2, \ldots, e_N$. Initially, these elements are stored in array A without any changes.

Balancing of a heap is done by sift-down operation (swapping elements which are out of order).

All leaves of the binary tree are already ordered. Such elements are at indices $(\frac{N}{2} + 1), \ldots, N$.

Then we take elements $\frac{N}{2}, \frac{N}{2} - 1, \dots, 1$ and apply recursive checking of the ordering condition.

If this condition is not satisfied sift-down transformation is used to restore heap condition, i.e. the given element is swapped with its largest child.

Then we take elements $\frac{N}{2}, \frac{N}{2} - 1, \ldots, 1$ and apply recursive checking of the ordering condition.

If this condition is not satisfied sift-down transformation is used to restore heap condition, i.e. the given element is swapped with its largest child.

This checking step is continued in a new place till heap condition is satisfied (recursion technique).

Heap algorithm

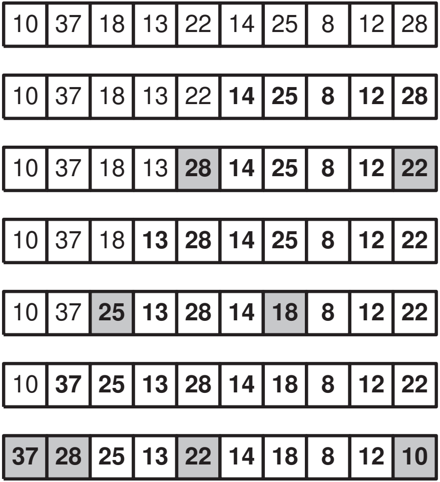
```
MakeHeap ()
begin
    for ( i=1; i $\leqslant$ N; i++ ) do
        $a_i = e_i$;
    end do
    for ( i=N/2; i > 0; i - - ) do
        HeapDownOrder ( i, N );
    end do
end MakeHeap
```

This function is iterative, since at each step only one sub-tree is considered.

```
HeapDownOrder ( p, N )
begin
    i=p;    j = 2i;
    while ( j $\leqslant$ N ) do
        k = j;
        if ( (j+1) $\leqslant$ N ) then
            if ( $a_{j+1} > a_j$ ) k = j+1;
        if ( $a_i < a_k$ ) then
            swap ($a_i$, $a_k$);
            i = k;    j= 2i;
        else
            j = N+1;
end HeapDownOrder
```

Make a heap from array

$$A = (10, 37, 18, 13, 22, 14, 25, 10, 12, 28).$$



10	37	18	13	22	14	25	8	12	28

10	37	18	13	22	**14**	**25**	**8**	**12**	**28**

10	37	18	13	**28**	**14**	**25**	**8**	**12**	**22**

10	37	18	**13**	**28**	**14**	**25**	**8**	**12**	**22**

10	37	**25**	**13**	**28**	**14**	**18**	**8**	**12**	**22**

10	**37**	**25**	**13**	**28**	**14**	**18**	**8**	**12**	**22**

37	**28**	**25**	**13**	**22**	**14**	**18**	**8**	**12**	**10**

FIGURE: Construction of heap.

Now we estimate the complexity of this algorithm.

Since a heap is balanced binary tree, its height is equal to $\log N$.

For each activation of HeapDownOrder algorithm the number of comparisons is not larger than $2 \log N$ and the number of swappings is not larger than $\log N$.
Thus total costs can be bounded from above as

$$L(N) \le N \log N, \quad S(N) \le \frac{1}{2} N \log N\,.$$