HEAPS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Agenda

1 Introduction

2 Heaps

3 Heapsort

4 Bibliography

Centro de Informática UFPE

Universidade Federal de Pernambuco

Introduction

Priority queues

- Job scheduling
- Algorithms such as Prim and Dijkstra
- Sorting (heapsort)

Priority queue ADT

- E $find\_max(heap\ h)$
- E $remove\_max(heap\ h)$
- $void\ add(heap\ h, Key\ K, Element\ E)$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Introduction

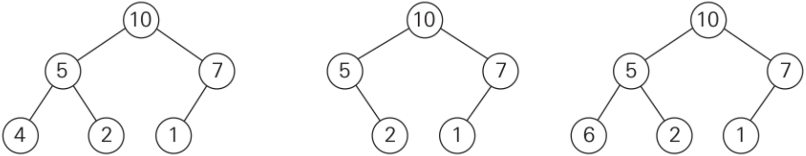
Possible implementations (cons):

- List (sorted/unsorted): single (worst) insertion or deletion in $O(n)$
- BST: n insertions/deletions in $\Theta(n\ log\ n)$ (in average)
- AVL: cost of rotations and space efficiency

Heap: a binary tree where the following two conditions are met

- Shape property: a complete binary tree
- Parental dominance: each node key is $\geq / \leq$ than the childrens key
  - If $\geq$: max-heap
  - If $\leq$: min-heap

Introduction[1]



Left-most tree: is a heap

Middle tree: is not a heap (shape property not met)

Right-most tree: is not a heap (parental dominance not met)

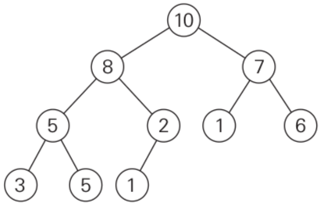
[1] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: an array-based implementation[2]



the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

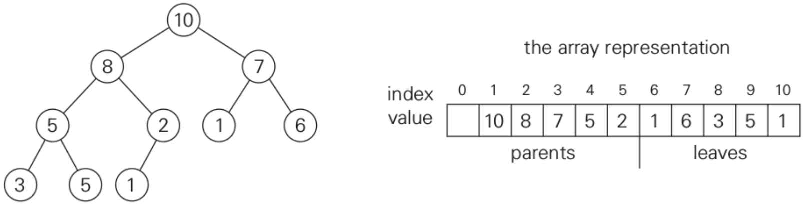
parents | leaves

Properties:

- Height of a heap with n nodes = $\lfloor log_2\ n \rfloor$
- Largest element: the root (in a max-heap)
- Root position: index 1

[2] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Heaps: an array-based implementation[3]



Properties:

- Internal nodes: first $\lfloor n/2 \rfloor$ positions
- Leaves: last $\lceil n/2 \rceil$ positions
- Children of $1 \leq i \leq \lfloor n/2 \rfloor$: $2i$ and $2i + 1$ (if value $\leq n$)
- Parent of $2 \leq i \leq n$: $\lfloor i/2 \rfloor$

[3] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Heaps: an array-based implementation

Creating a heap:

- Top-down
  - Elements not known beforehand
  - Elements inserted one-by-one
  - Heapify after each insertion
  - Worst temporal efficiency of each insertion in $O(log\ n)$
  - Worst temporal efficiency of heap creation in $O(n\ log\ n)$
- Bottom-up
  - Elements known beforehand
  - All elements inserted at once
  - Heapify from the last to the first internal node
  - Worst temporal efficiency of heap creation in $O(n)$

Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10

↓
2

Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10

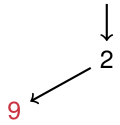
$\downarrow$

2

Heapify process

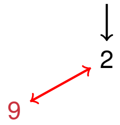
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

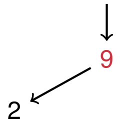
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: top-down creation (tree view)

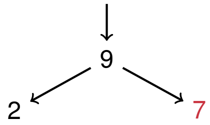
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

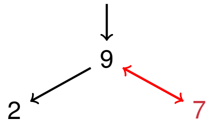
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

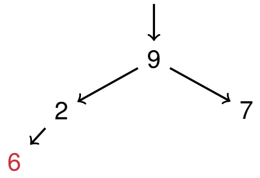
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

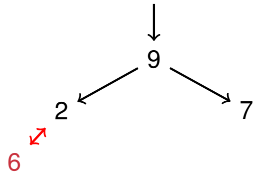
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

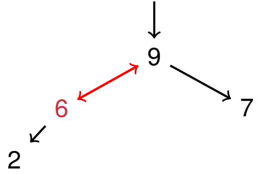
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: top-down creation (tree view)

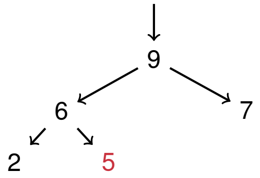
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

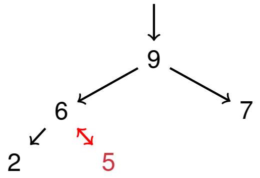
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

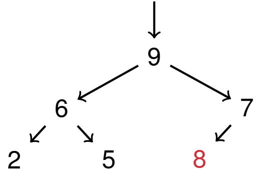
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

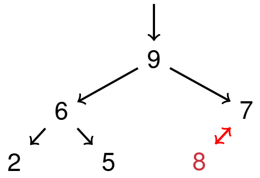
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

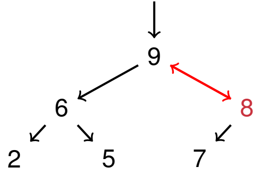
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: top-down creation (tree view)

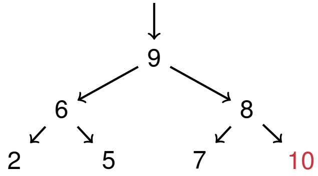
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

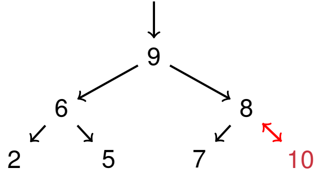
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: top-down creation (tree view)

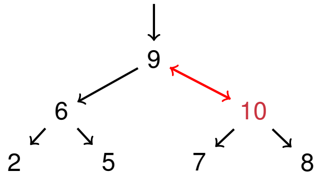
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: top-down creation (tree view)

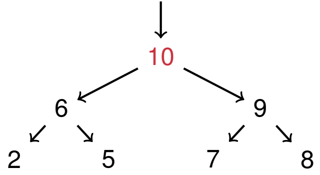
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: top-down creation (tree view)

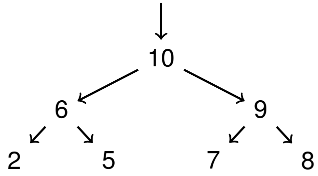
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

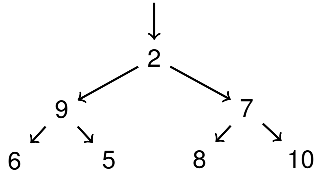
Heaps: top-down creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



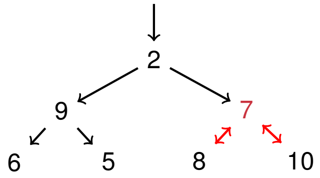
Heaps: bottom-up creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Heaps: bottom-up creation (tree view)

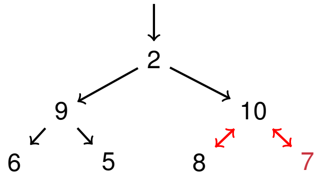
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

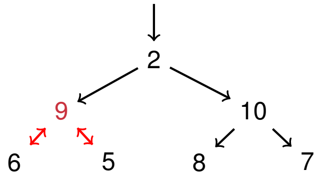
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

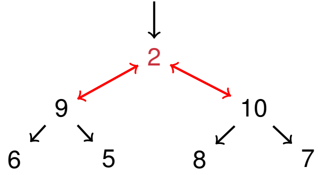
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

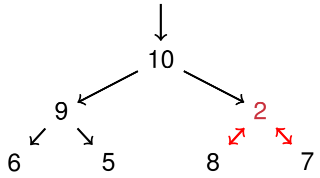
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

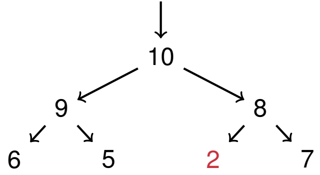
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

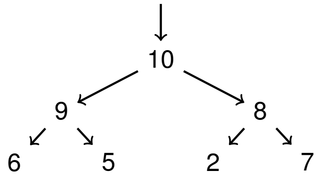
Inserting: 2, 9, 7, 6, 5, 8, 10



Heapify process

Heaps: bottom-up creation (tree view)

Inserting: 2, 9, 7, 6, 5, 8, 10



Even provided the same sequence of numbers, bottom-up and top-down creation algorithms may result in structurally different heaps.

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i;                     // current position of the i-th internal node
    v ← H[k];                  // value of the i-th internal node
    heap ← false;
    // finding the proper place for the i-th internal node
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;             // position of the first child
        if j < n then
            // has two children | finds the largest child
            if H[j] < H[j + 1] then j ← j + 1;
        if v ≥ H[j] then
            // is a heap if v is ≥ than the largest child
            heap ← true;
        else
            // places the largest child in H[k] | updates k
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

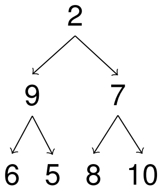
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```

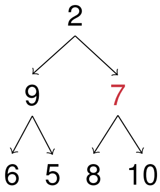


index:	0	1	2	3	4	5	6	7
H:		2	9	7	6	5	8	10

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 * k ≤ n do
        j ← 2 * k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	7	6	5	8	10
				i				
				k				

$v = 7$

$heap = false$

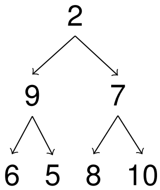
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

for $i \leftarrow \lfloor n/2 \rfloor$ ***downto*** 1 **do**
  $k \leftarrow i$; $v \leftarrow H[k]$;
  $heap \leftarrow$ **false**;
  while $\neg\, heap \wedge 2 * k \leq n$ **do**
    $j \leftarrow 2 * k$;
    if $j < n$ **then**
      if $H[j] < H[j+1]$ **then**
        $j \leftarrow j + 1$
    if $v \geq H[j]$ **then**
      $heap \leftarrow$ **true**;
    else
      $H[k] \leftarrow H[j]$;
      $k \leftarrow j$;
  $H[k] \leftarrow v$;



index:	0	1	2	3	4	5	6	7
H:		2	9	7	6	5	8	10
				i				
				k				
							j	

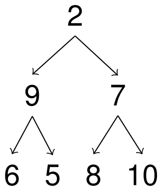
$v = 7$

$heap = false$

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	7	6	5	8	10
				i				
				k				
								j

$v = 7$

$heap = false$

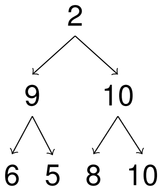
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	10
				i				
								k
								j

$v = 7$

$heap = false$

Heaps: bottom-up creation (algorithm)

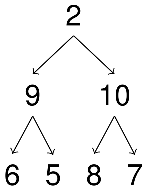
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
				i				
								k

$v = 7$

$heap = false$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

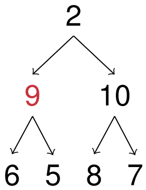
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
			i					
			k					

$v = 9$

$heap = false$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

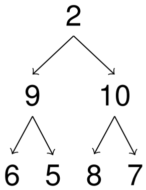
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
			i					
			k					
					j			

$v = 9$

$heap = false$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

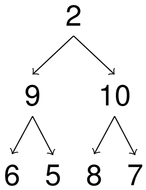
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
			i					
			k					
					j			

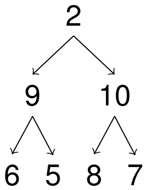
$v = 9$

$heap = true$

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

for $i \leftarrow \lfloor n/2 \rfloor$ ***downto*** 1 **do**
  $k \leftarrow i$; $v \leftarrow H[k]$;
  heap $\leftarrow$ **false**;
  while $\neg$ *heap* $\wedge\ 2 * k \leq n$ **do**
    $j \leftarrow 2 * k$;
    if $j < n$ **then**
      if $H[j] < H[j+1]$ **then**
        $j \leftarrow j + 1$
    if $v \geq H[j]$ **then**
      heap $\leftarrow$ **true**;
    else
      $H[k] \leftarrow H[j]$;
      $k \leftarrow j$;
  $H[k] \leftarrow v$;



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
			i					
			k					

$v = 9$

$heap = true$

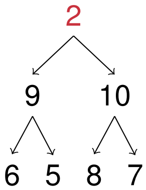
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
		i						
		k						

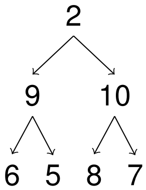
$v = 2$

$heap = false$

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 * k ≤ n do
        j ← 2 * k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
		i						
		k						
			j					

$v = 2$

$heap = false$

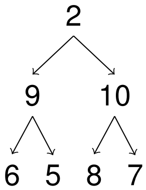
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 * k ≤ n do
        j ← 2 * k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		2	9	10	6	5	8	7
		i						
		k						
				j				

$v = 2$

$heap = false$

Heaps: bottom-up creation (algorithm)

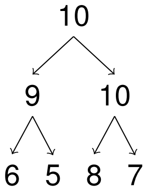
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		10	9	10	6	5	8	7
		i						
				k				
				j				

$v = 2$

$heap = false$

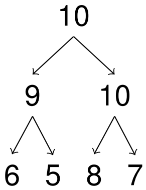
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		10	9	10	6	5	8	7
		i						
				k				
							j	

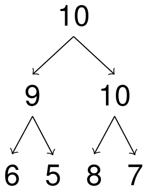
$v = 2$

$heap = false$

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		10	9	10	6	5	8	7
		i						
				k				
							j	

$v = 2$

$heap = false$

Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

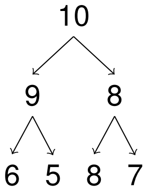
Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1

        if v ≥ H[j] then
            heap ← true;

        else
            H[k] ← H[j];
            k ← j;

    H[k] ← v;
```



index:	0	1	2	3	4	5	6	7
H:		10	9	8	6	5	8	7
		i						
							k	
							j	

$v = 2$

$heap = false$

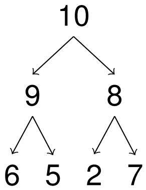
Centro de Informática UFPE

Universidade Federal de Pernambuco

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

for $i \leftarrow \lfloor n/2 \rfloor$ ***downto*** 1 **do**
- $k \leftarrow i$; $v \leftarrow H[k]$;
- *heap* $\leftarrow$ **false**;
- **while** $\neg$ *heap* $\wedge 2 * k \leq n$ **do**
  - $j \leftarrow 2 * k$;
  - **if** $j < n$ **then**
    - **if** $H[j] < H[j+1]$ **then**
      - $j \leftarrow j + 1$
  - **if** $v \geq H[j]$ **then**
    - *heap* $\leftarrow$ **true**;
  - **else**
    - $H[k] \leftarrow H[j]$;
    - $k \leftarrow j$;
- $H[k] \leftarrow v$;



index:	0	1	2	3	4	5	6	7
H:		10	9	8	6	5	2	7
		i						
							k	

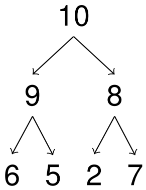
$v = 2$

$heap = false$

Heaps: bottom-up creation (algorithm)

Algorithm: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```

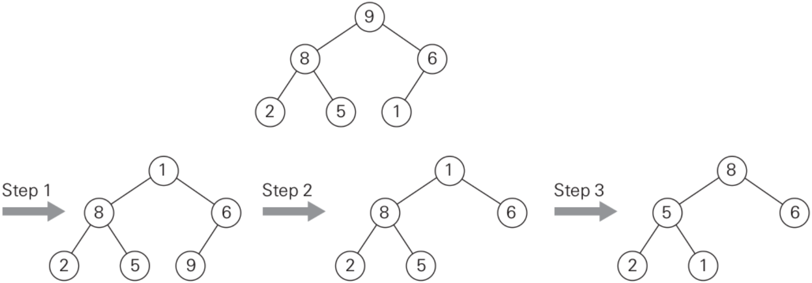


index:	0	1	2	3	4	5	6	7
H:		10	9	8	6	5	2	7
	i							

Heaps: deletion[4]

The largest element is the root (max-heap)

1. Exchange the root's key with the last key K.
2. Decrease the heap's size by 1.
3. Heapify the tree by sifting K down (see bottom-up algorithm).



Deletion is in $O(log\ n)$

[4] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

Heapsort[5]

Sorting algorithm based on heaps

1. Construct a heap for a given array (bottom-up construction).
2. Apply the deletion operations $n - 1$ times to the remaining heap.

Temporal efficiency in $O(n \log n)$

- Bottom-up creation in $O(n)$
- Each deletion in $O(log\ n)$

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

Centro de Informática UFPE

Universidade Federal de Pernambuco

[5] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Asymptotic efficiency of sorting algorithms[6]

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Temporal efficiency: same class of Mergesort (better space efficiency)

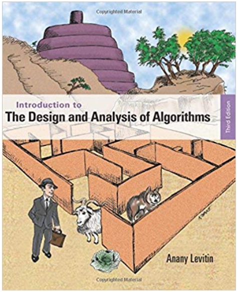
In general, worse temporal efficiency than Quicksort

Finding the k largest elements: $O(n + k\ log\ n)$

[6] Source: http://bigocheatsheet.com/

Agenda

Bibliography



Chapter 6 (pp. 226–232)
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3rd edition. Pearson. 2011.

Chapter 5 (pp. 170–177)
Chapter 7 (pp. 243–244)
Clifford Shaffer.
Data Structures and Algorithm Analysis. Dover, 2013.

Centro de Informática UFPE

Universidade Federal de Pernambuco

HEAPS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil