Heap Sort and Insertion Sort

towards data structures

What really counts

- so far we counted only comparisons
- what really counts: ISA instructions of translated program
- you know the (unoptimized) translation process
    - C0 compiler

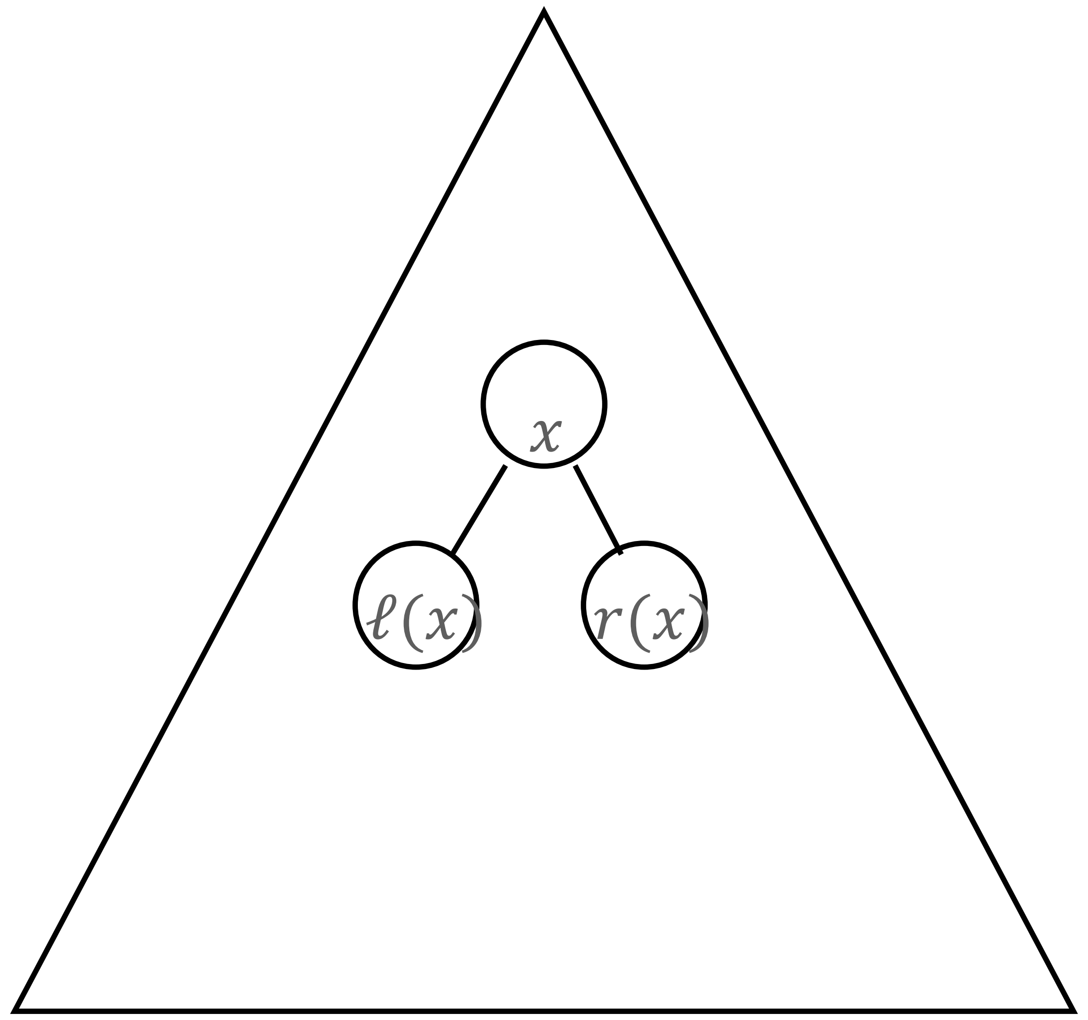
What really counts

- so far we counted only comparisons
- what really counts: ISA instructions of translated program
- you know the (unoptimized) translation process
    - C0 compiler

2 more sorting algorithms with O(n log n) comparisons

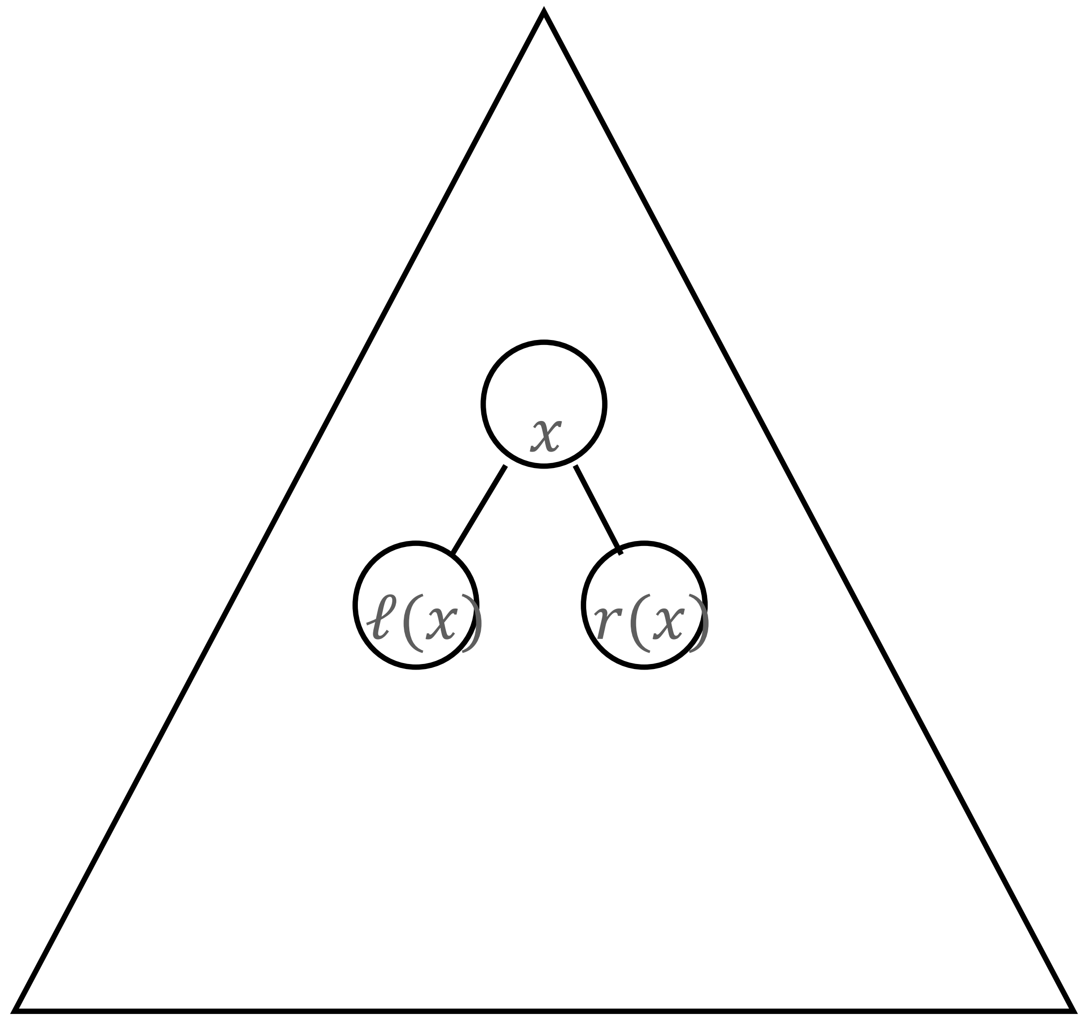
- so far
    - O(1) ISA instructions per comparison
    - 2 arrays of length n + …
- heap sort
    - O(1) ISA instructions/comparison
    - 1 array; only swaps in place
- insertion sort
    - O(n) ISA instructions/comparison with arrays
    - future: data structures; search trees…

storing the elements of a binary tree in an array.



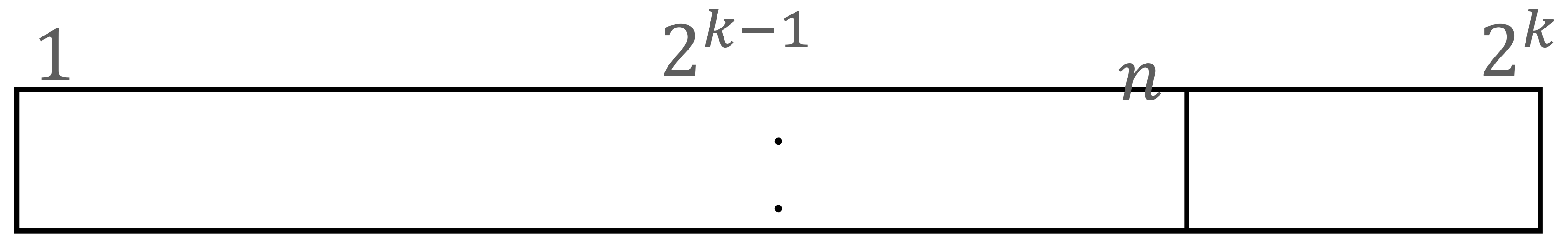
- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

storing the elements of a binary tree in an array.

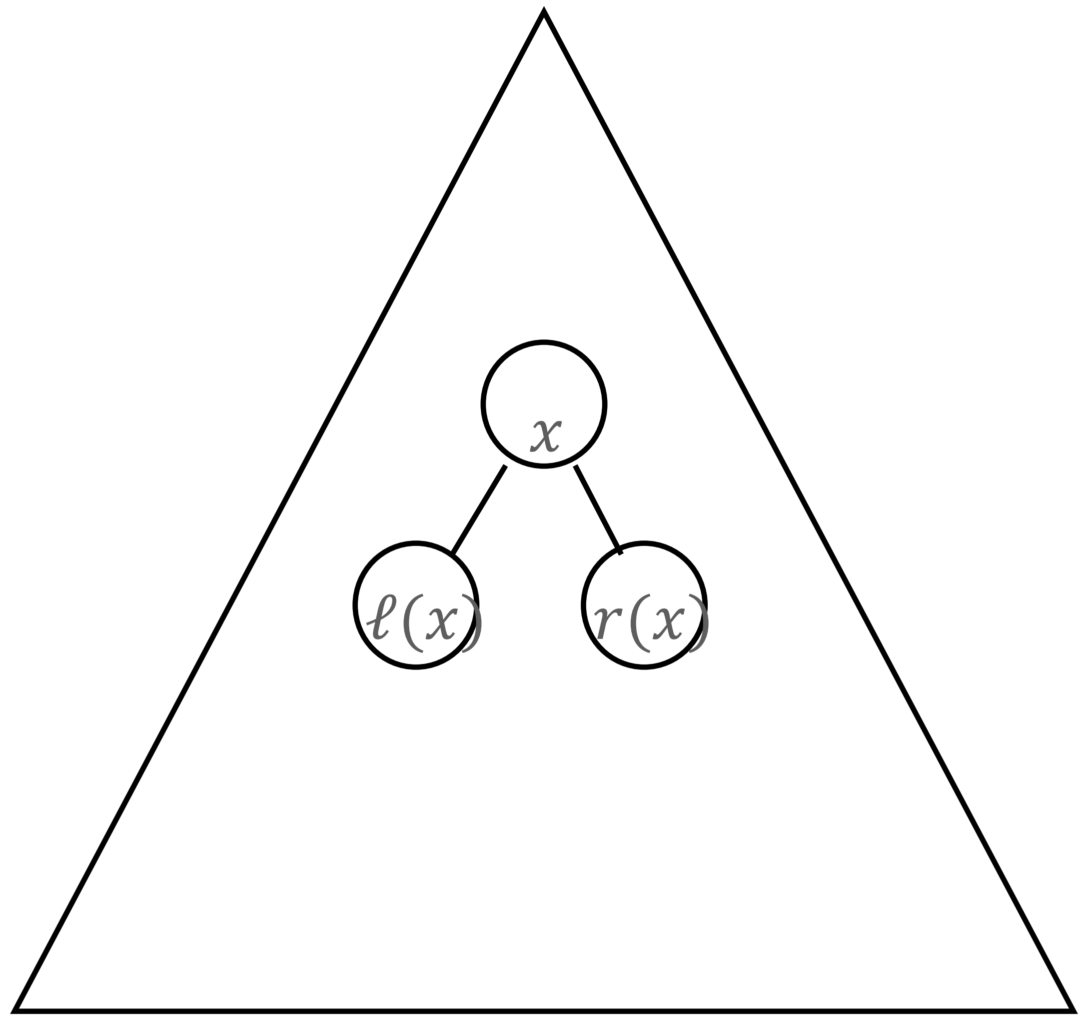


- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

implement in array A of length $2^k = 2^{\lceil \log n \rceil}$

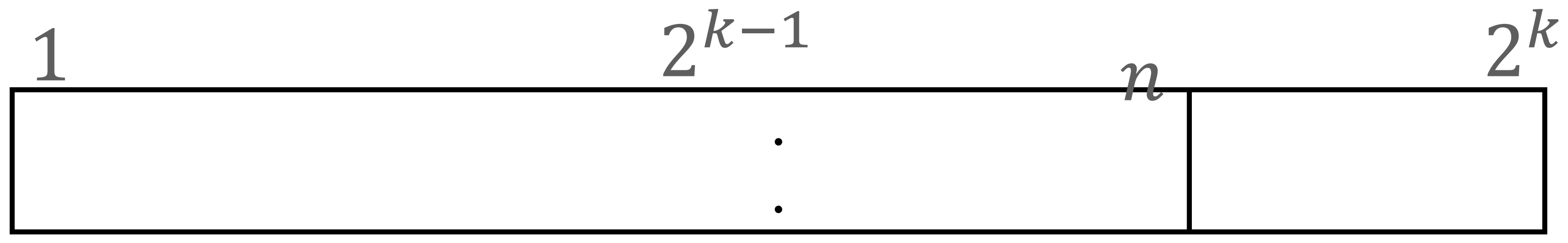


storing the elements of a binary tree in an array.

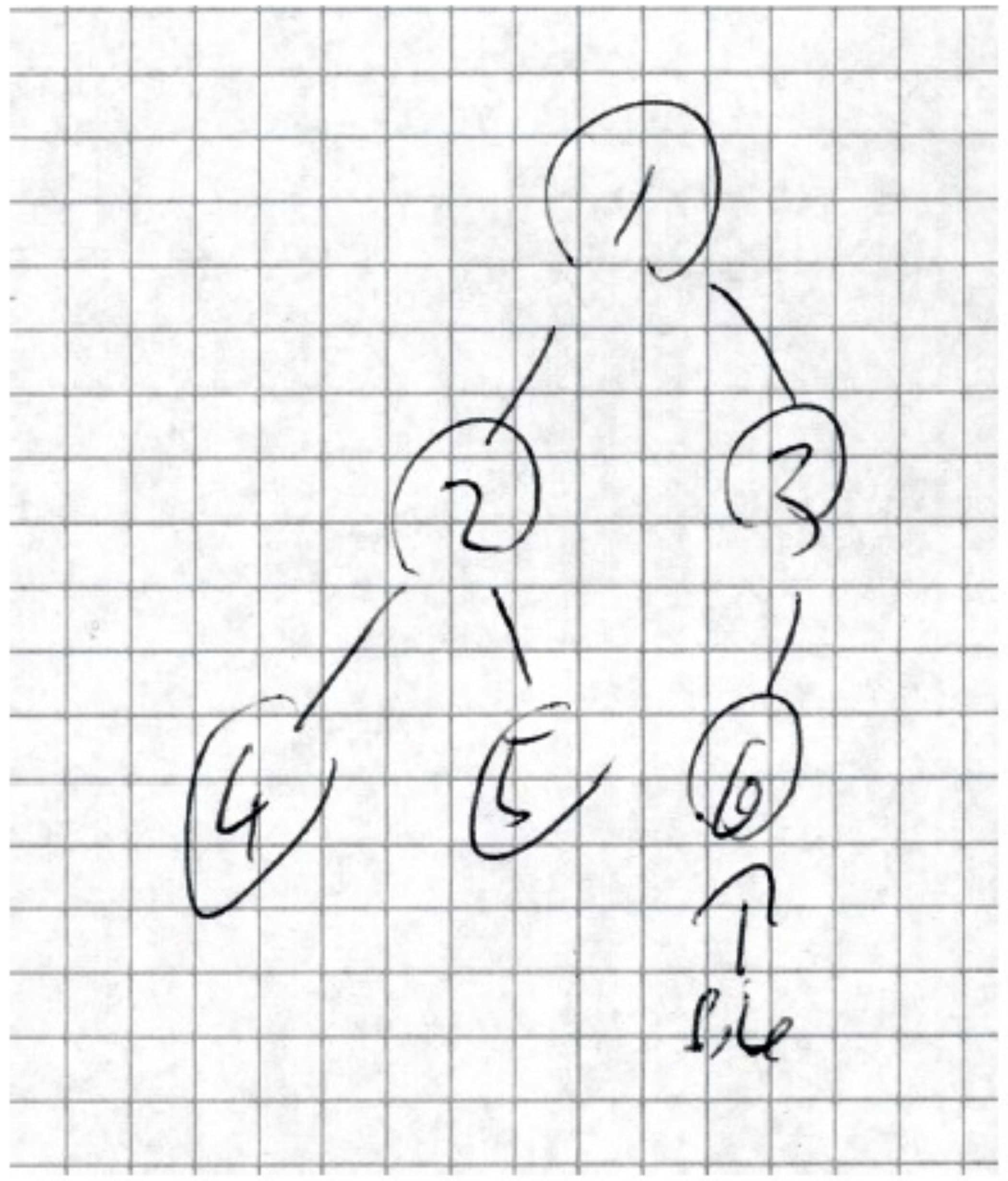


- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

implement in array A of length $2^k = 2^{\lceil \log n \rceil}$



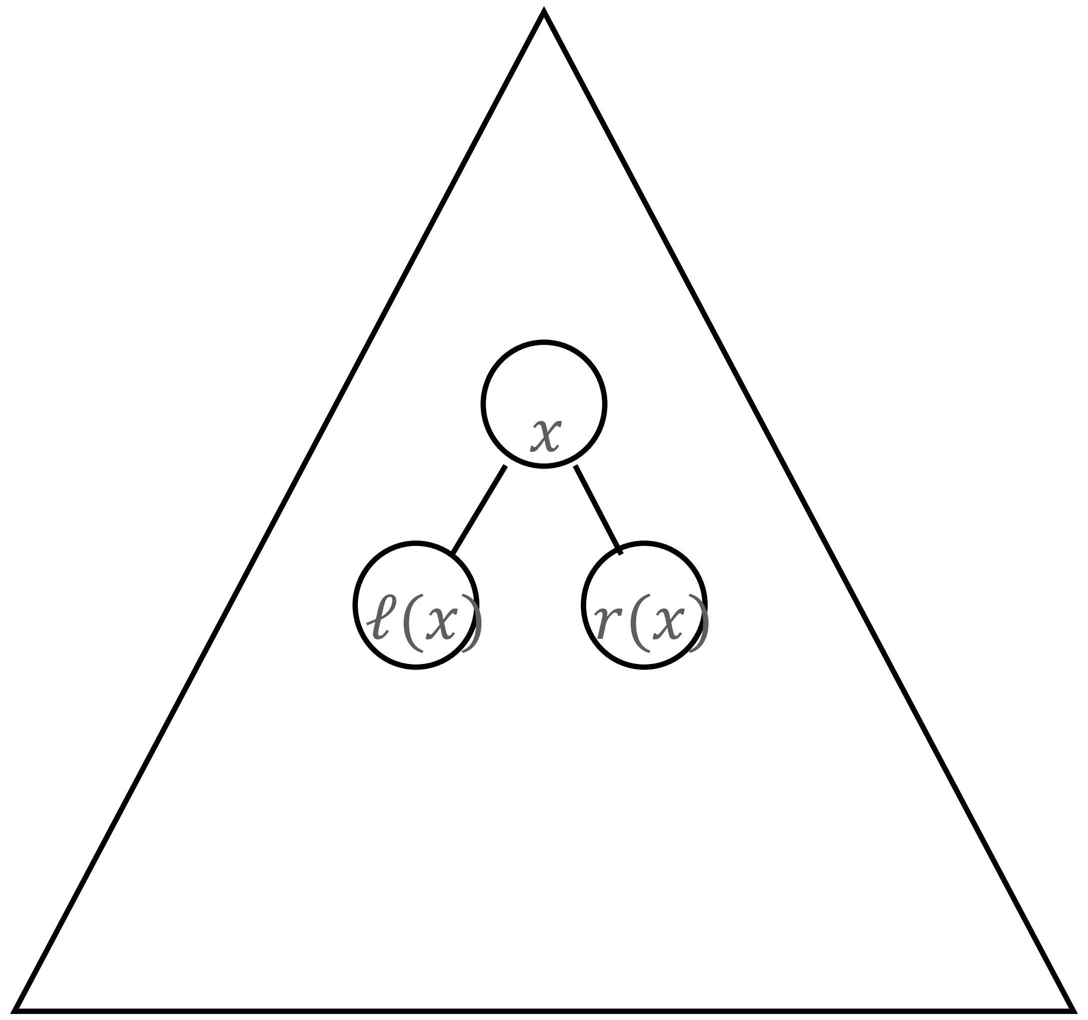
$$
\begin{aligned}
root &= 1 \\
\ell(x) &= 2x \quad \text{left son} \\
r(x) &= 2x+1 \quad \text{right son} \\
p(x) &= \cancel{\lfloor 2/x \rfloor} \quad \text{parent} \qquad \lfloor x/2 \rfloor
\end{aligned}
$$



order of nodes in array

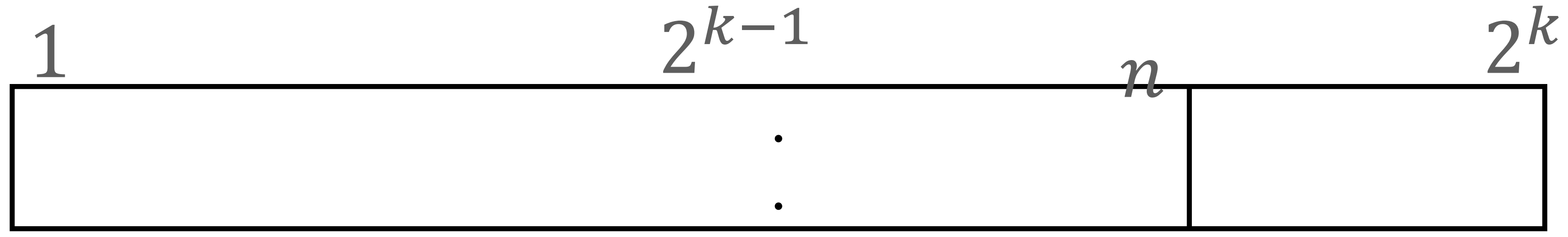
1. top to bottom
2. left to right

storing the elements of a binary tree in an array.



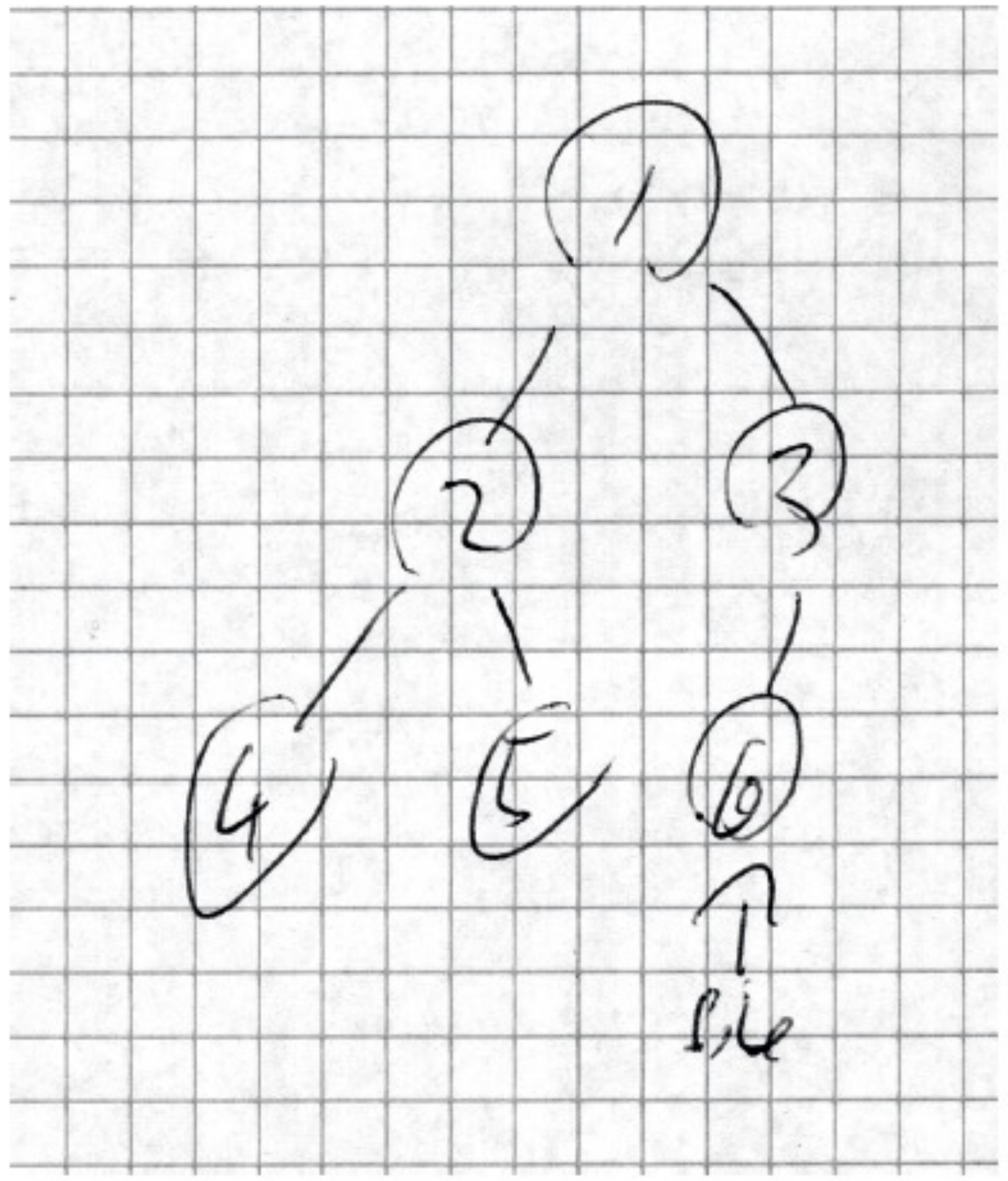
- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

implement in array A of length $2^k = 2^{\lceil \log n \rceil}$



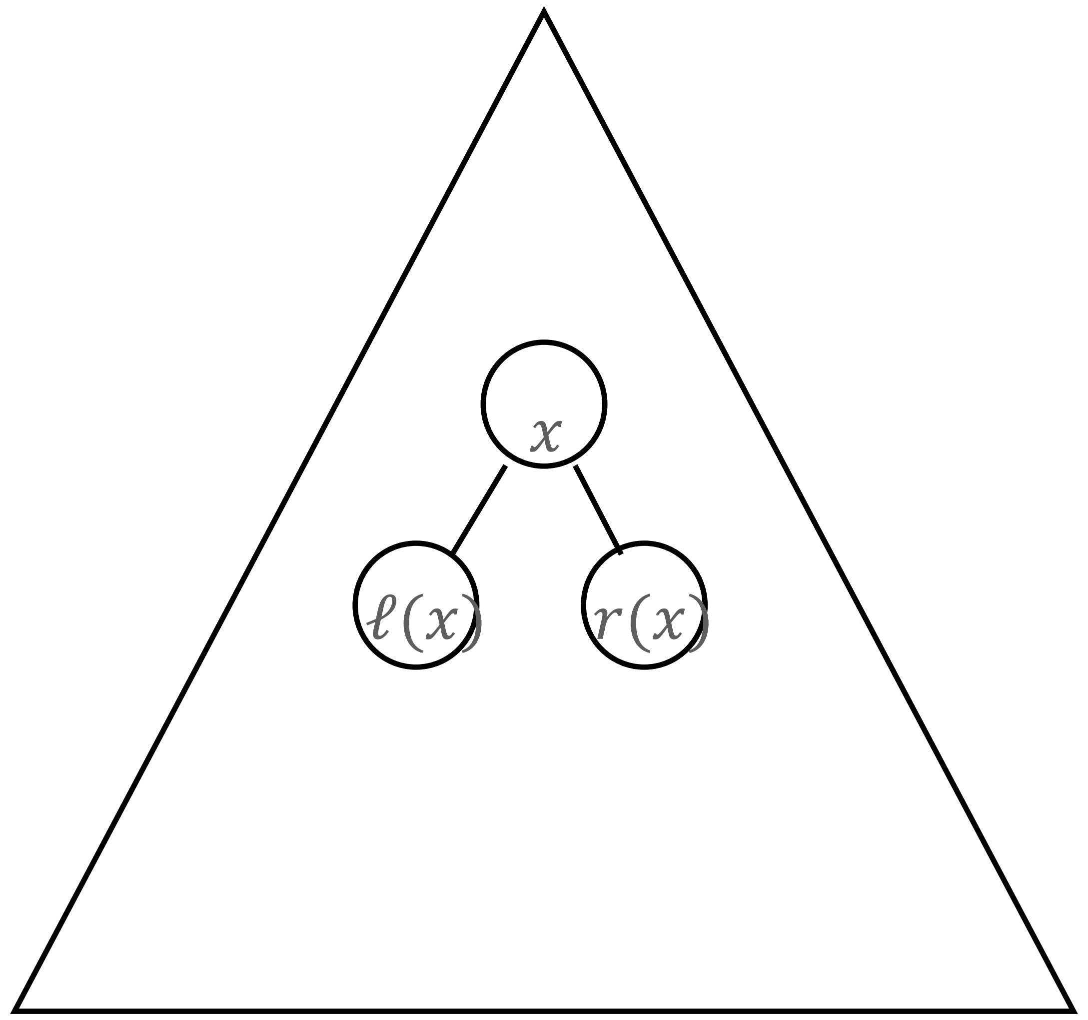
complete binary tree
except right end of bottom layer

$$
\begin{aligned}
root &= 1 \\
\ell(x) &= 2x \quad \text{left son} \\
r(x) &= 2x+1 \quad \text{right son} \\
p(x) &= \cancel{\lfloor 2/x \rfloor} \quad \text{parent} \quad \lfloor x/2 \rfloor
\end{aligned}
$$



order of nodes in array
1. top to bottom
2. left to right

storing the elements of a binary tree in an array.

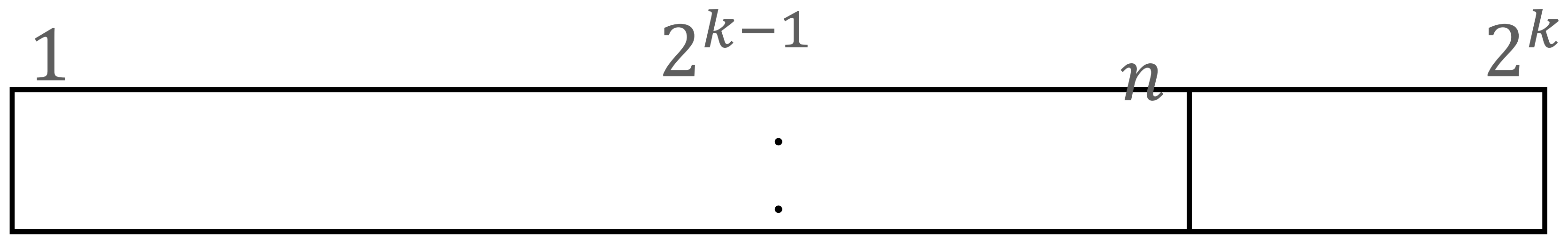


- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

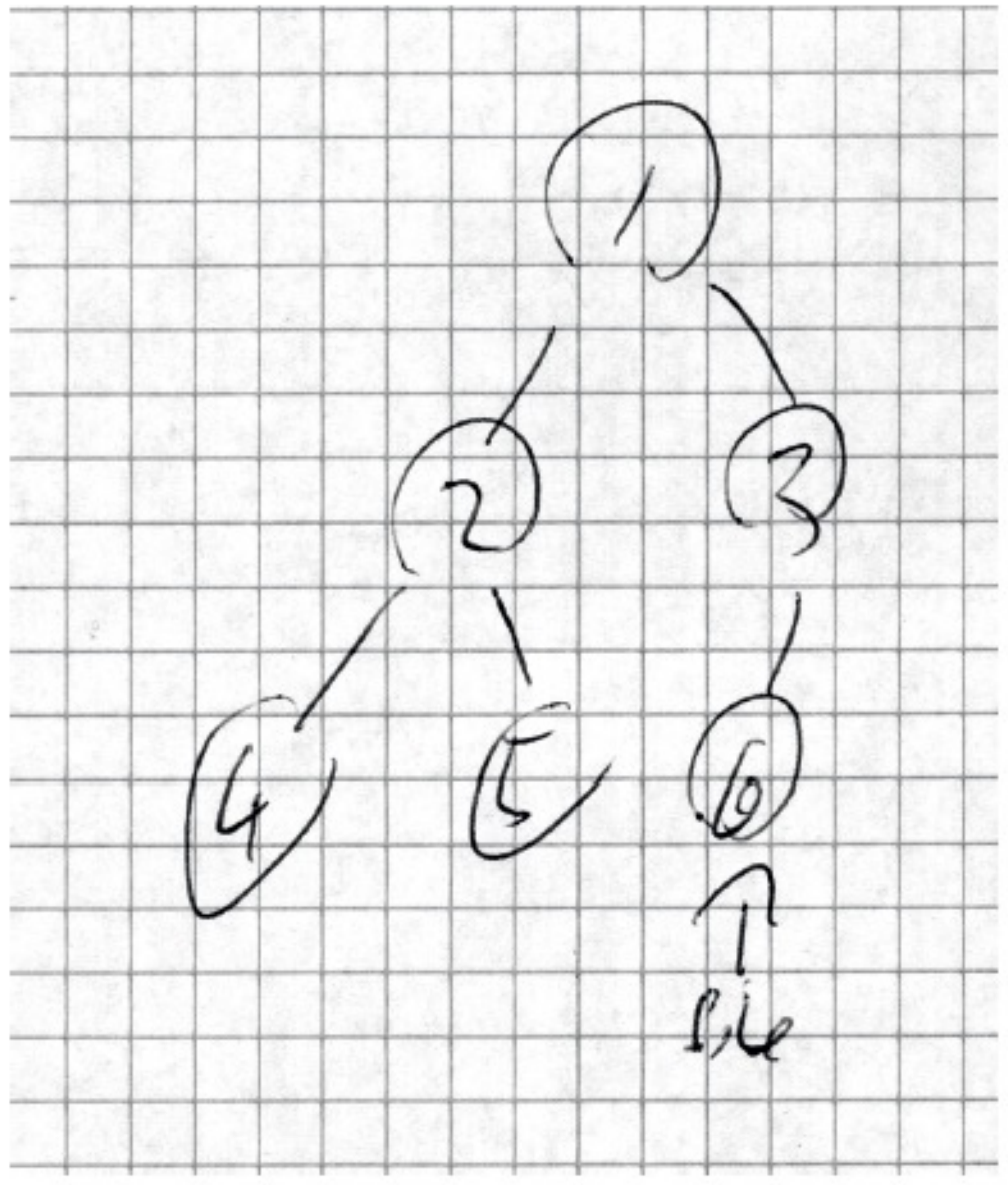
implement in array A of length $2^k = 2^{\lceil log n \rceil}$



complete binary tree
except right end of bottom layer

$$
\begin{array}{lcll}
root & = & 1 & \\
\ell(x) & = & 2x & \text{left son} \\
r(x) & = & 2x+1 & \text{right son} \\
p(x) & = & \cancel{\lfloor 2/x \rfloor} & \text{parent}
\end{array}
$$

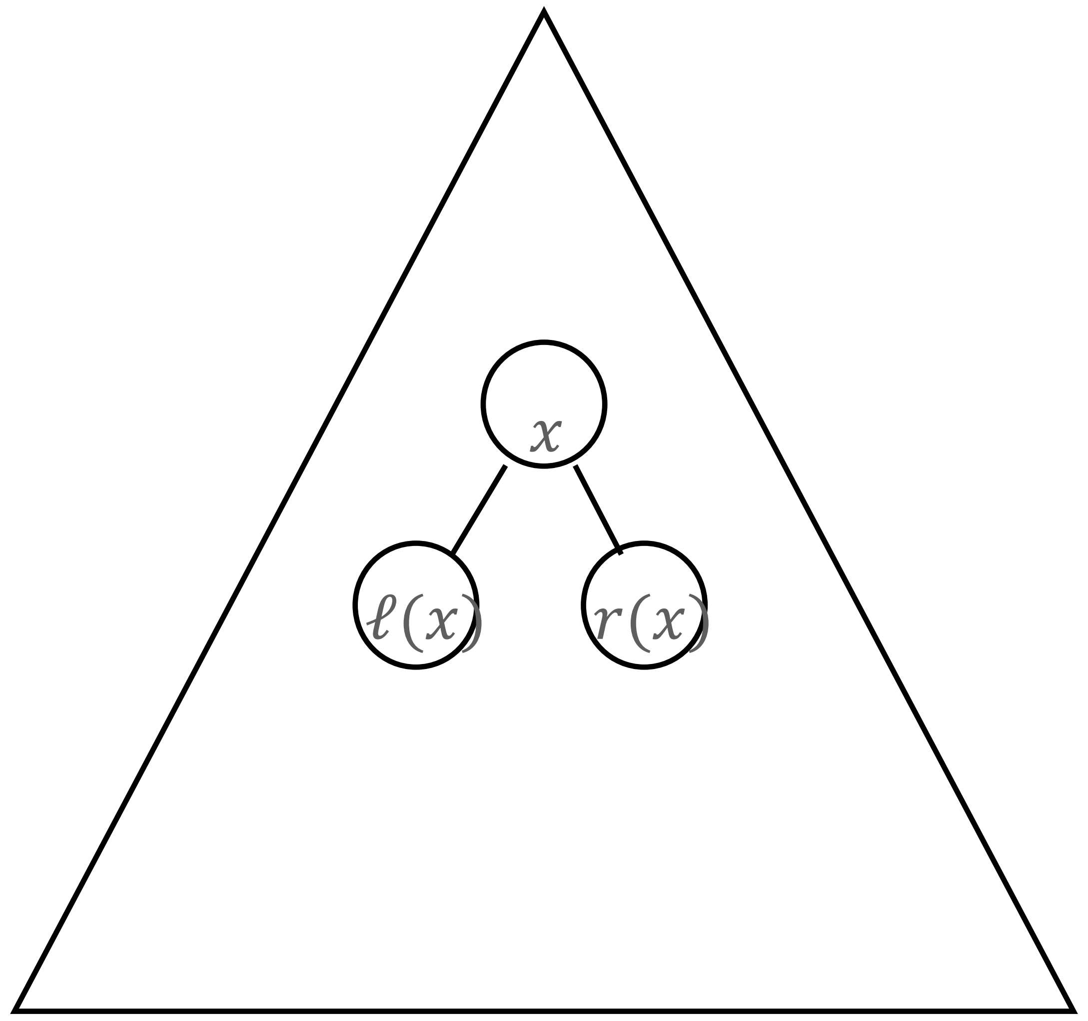
$\lfloor x/2 \rfloor$



order of nodes in array

1. top to bottom
2. left to right

storing the elements of a binary tree in an array.



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

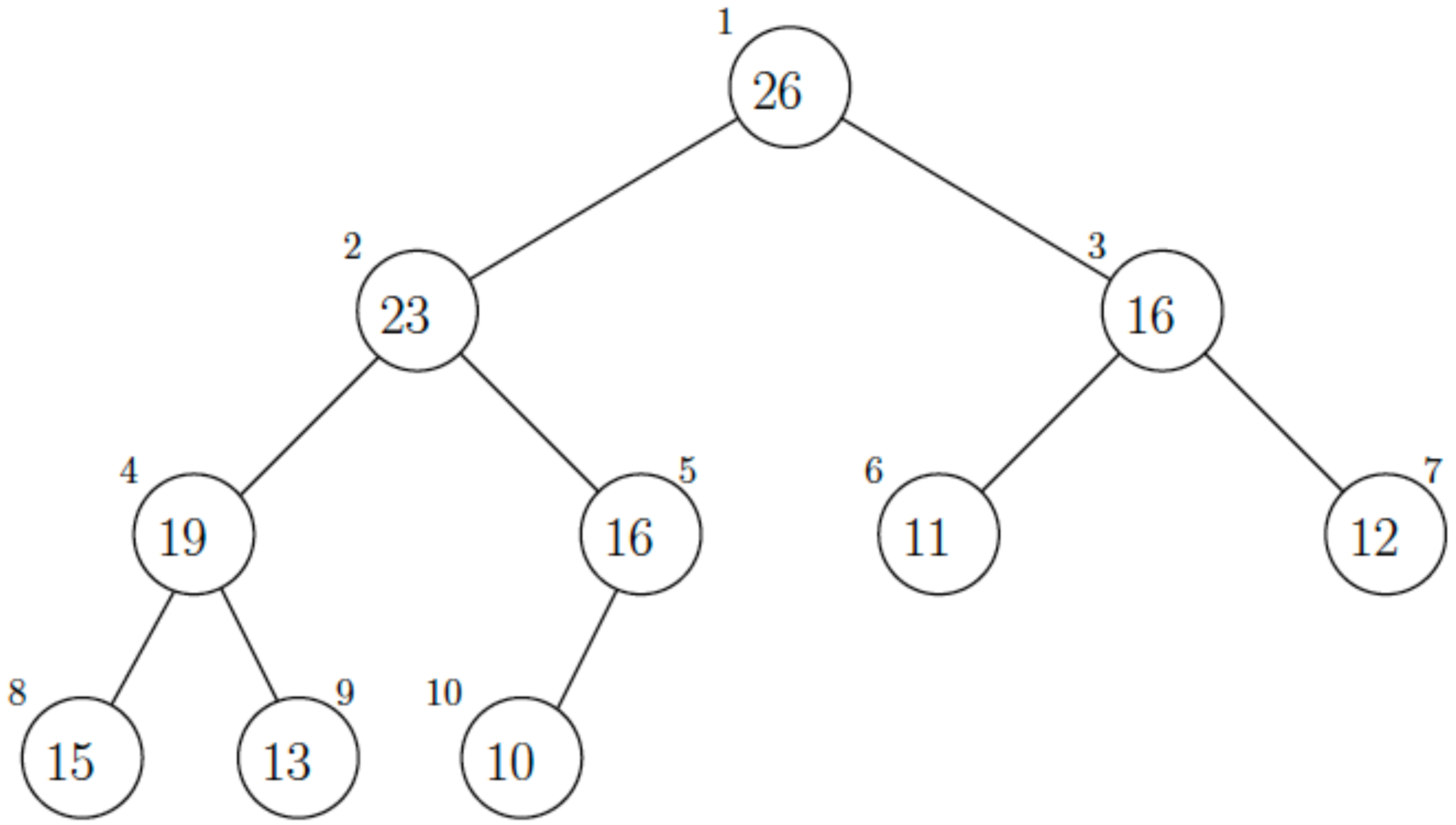
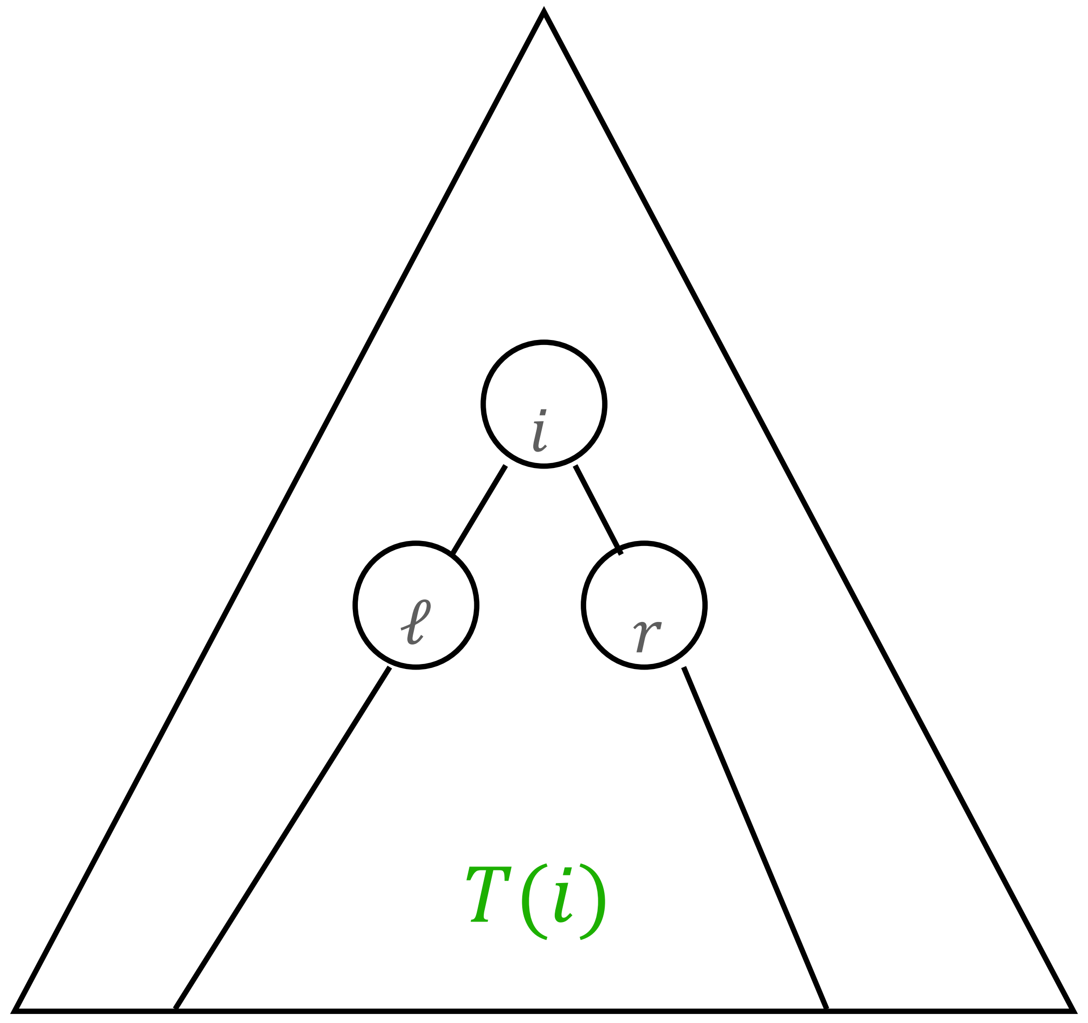


Figure 3.1: A heap. The large numbers in the circles are the elements stored in the heap. The small numbers next to the circle is the corresponding position in the array.

establishing heap property



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

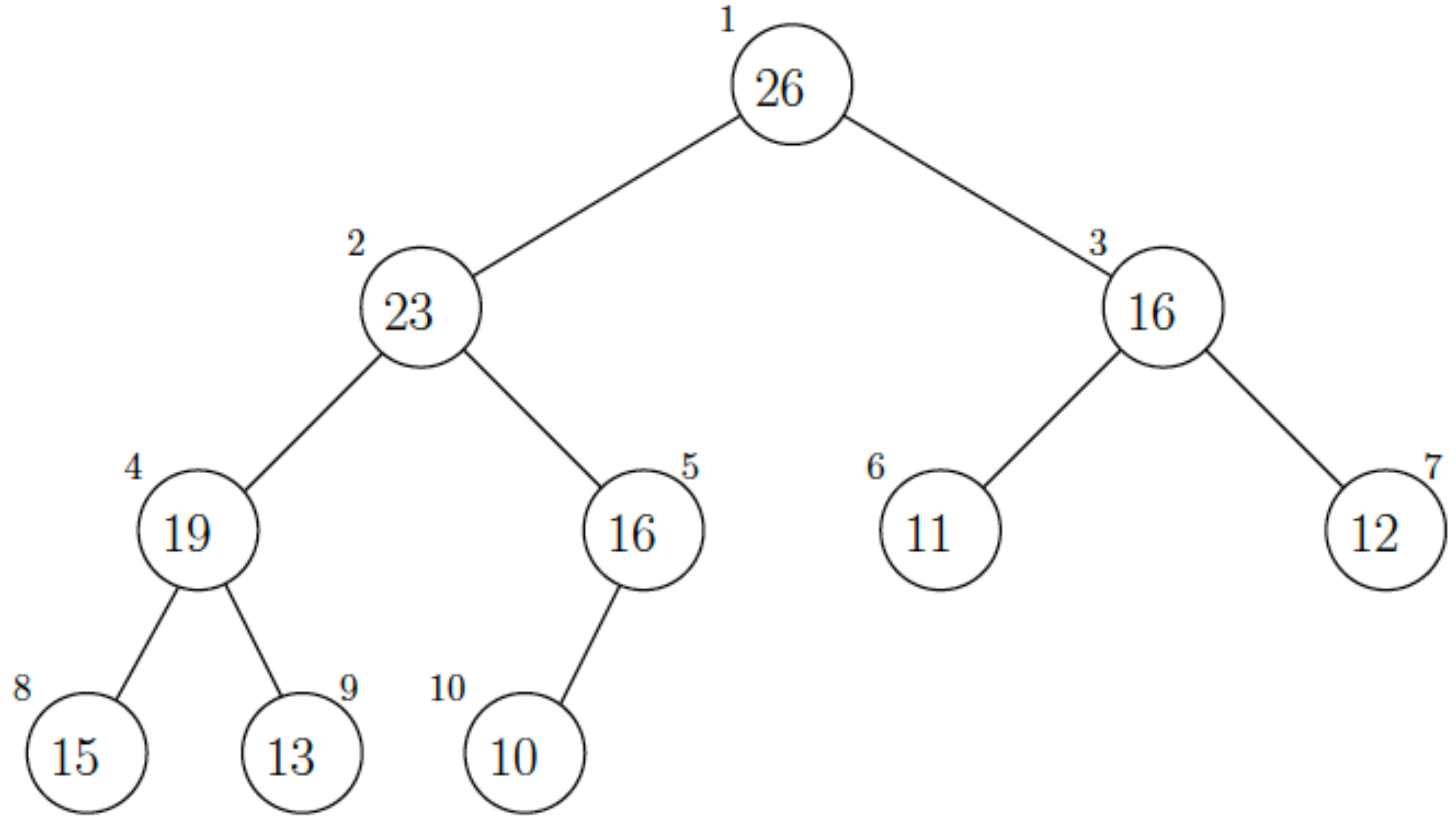
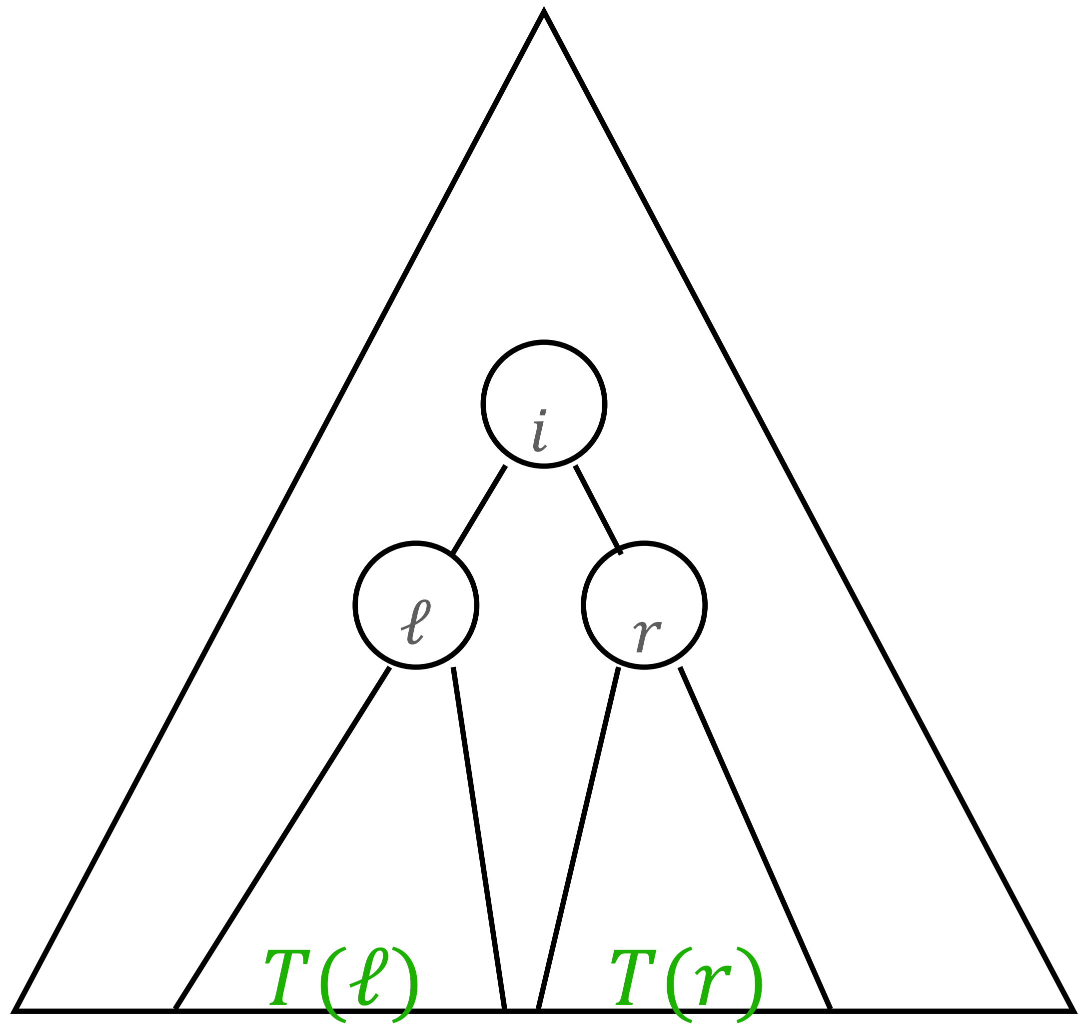


Figure 3.1: A heap. The large numbers in the circles are the elements stored in the heap. The small numbers next to the circle is the corresponding position in the array.

notation: $T(x)$: subtree with root x

establishing heap property



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

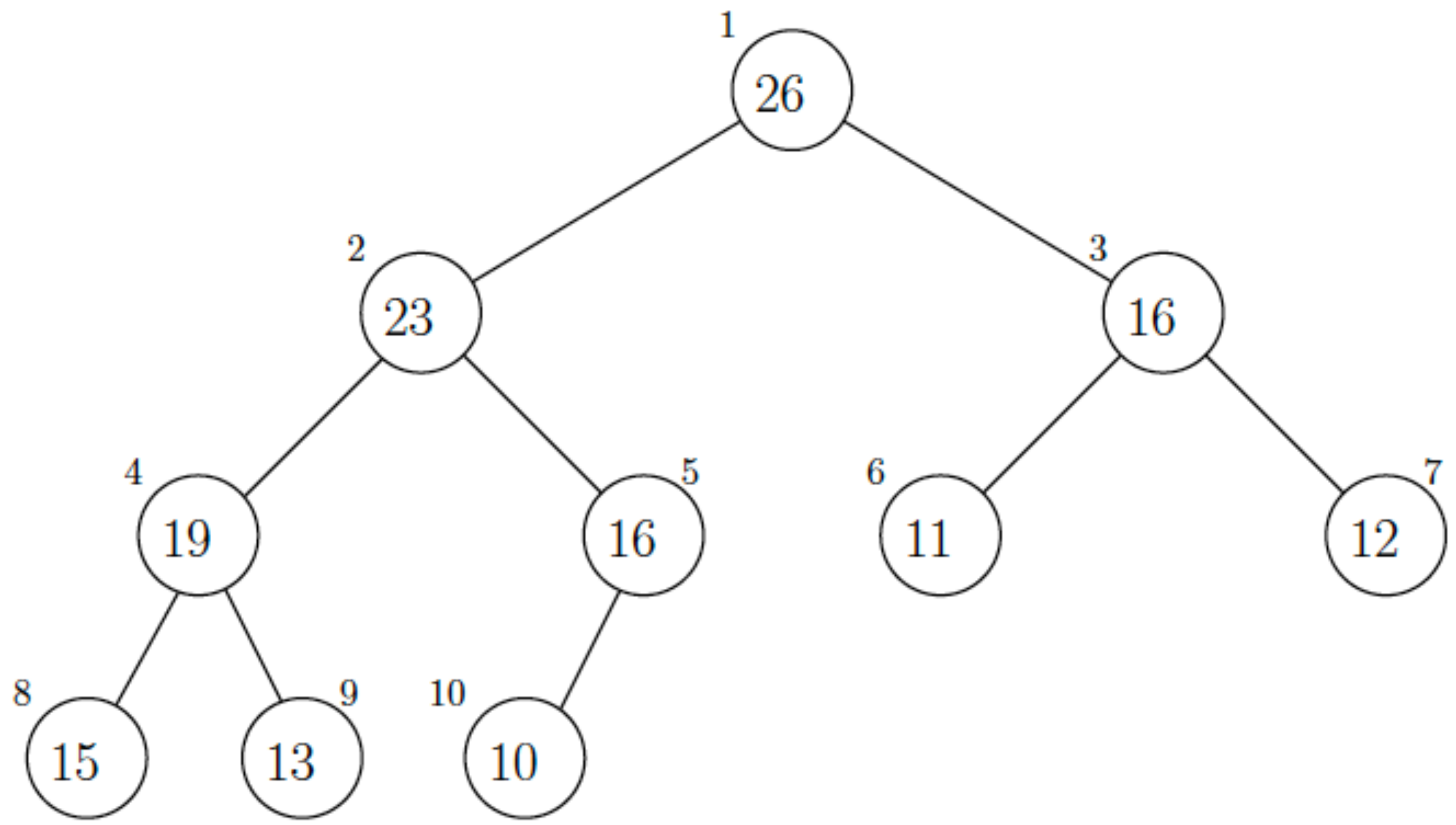


Figure 3.1: A heap. The large numbers in the circles are the elements stored in the heap. The small numbers next to the circle is the corresponding position in the array.

notation: $T(x)$: subtree with root x

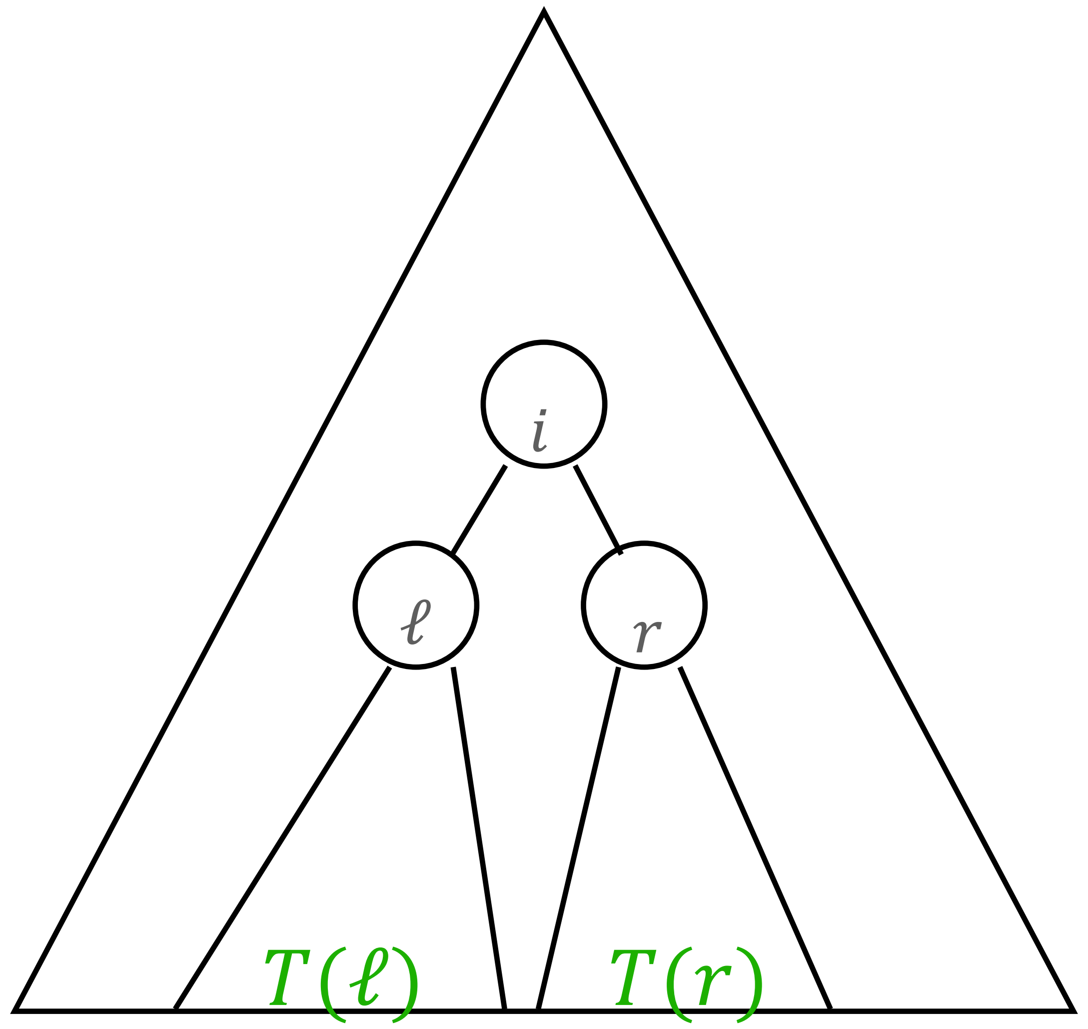
function heapify(A,i)

Input: A, i.

- i: interior node with sons $r = r(i)$, $ell = \ell(i)$
- subtrees $T(\ell), T(r)$ fulfill heap property

Output: afterwards $T(i)$ fulfills heap property.

establishing heap property



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

```
heapify(A,i):

A(h)  = max{A(i), A(l), A(r)}
h=i: done
h !=i: swap(A(i), A(h)) /* y=A(i);A(i)=A(h);A(h)=y*/
if h leaf {done} else {heapify(A,h)}
```

notation: $T(x)$: subtree with root x

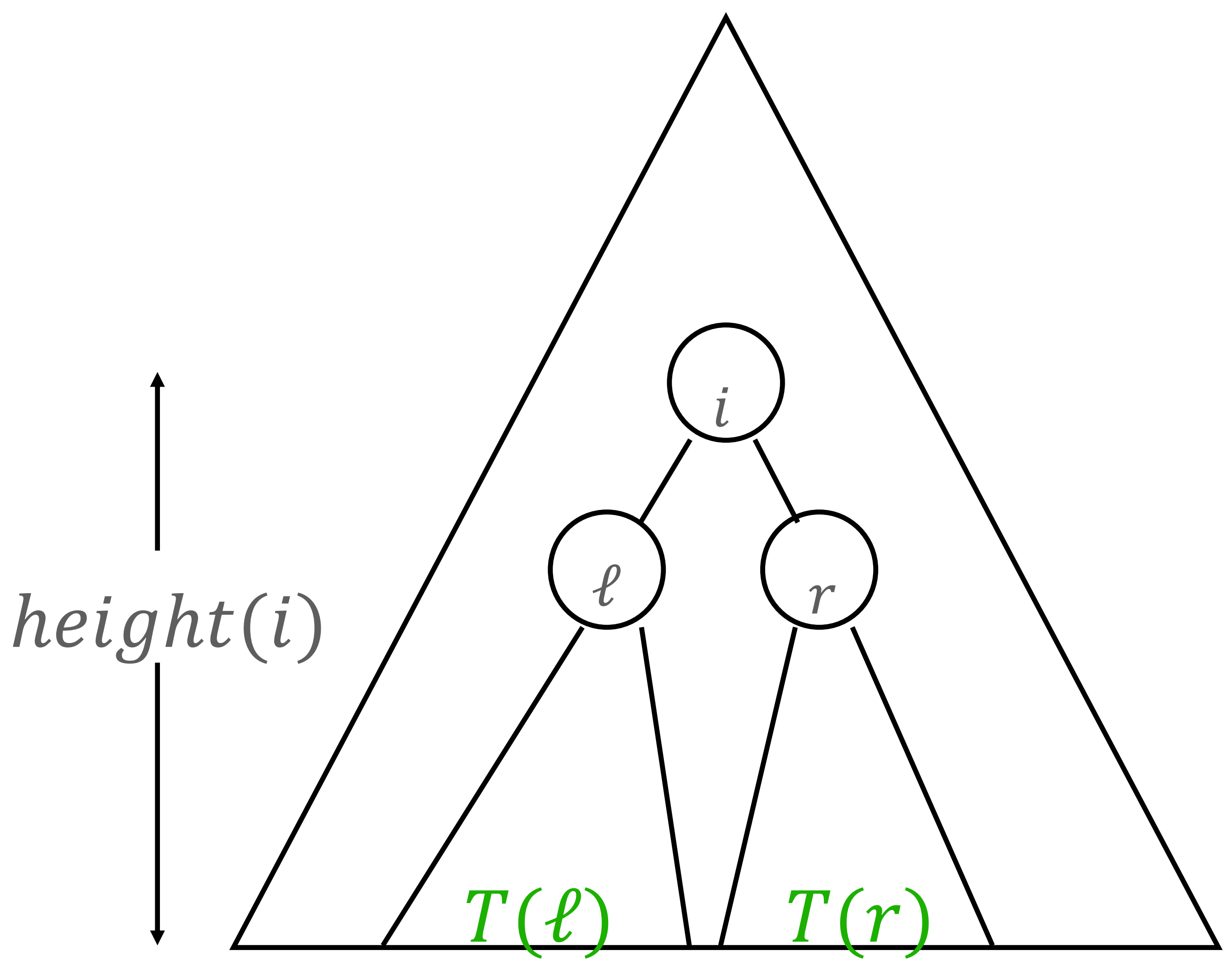
function heapify(A,i)

Input: A, i.

- i: interior node with sons $r = r(i)$, $ell = \ell(i)$
- subtrees $T(\ell), T(r)$ fulfill heap property

Output: afterwards $T(i)$ fulfills heap property.

establishing heap



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

notation: $T(x)$: subtree with root x

function heapify(A,i)

Input: A, i.

- i: interior node with sons $r = r(i)$, $ell = \ell(i)$
- subtrees $T(\ell), T(r)$ fulfill heap property

Output: afterwards $T(i)$ fulfills heap property.

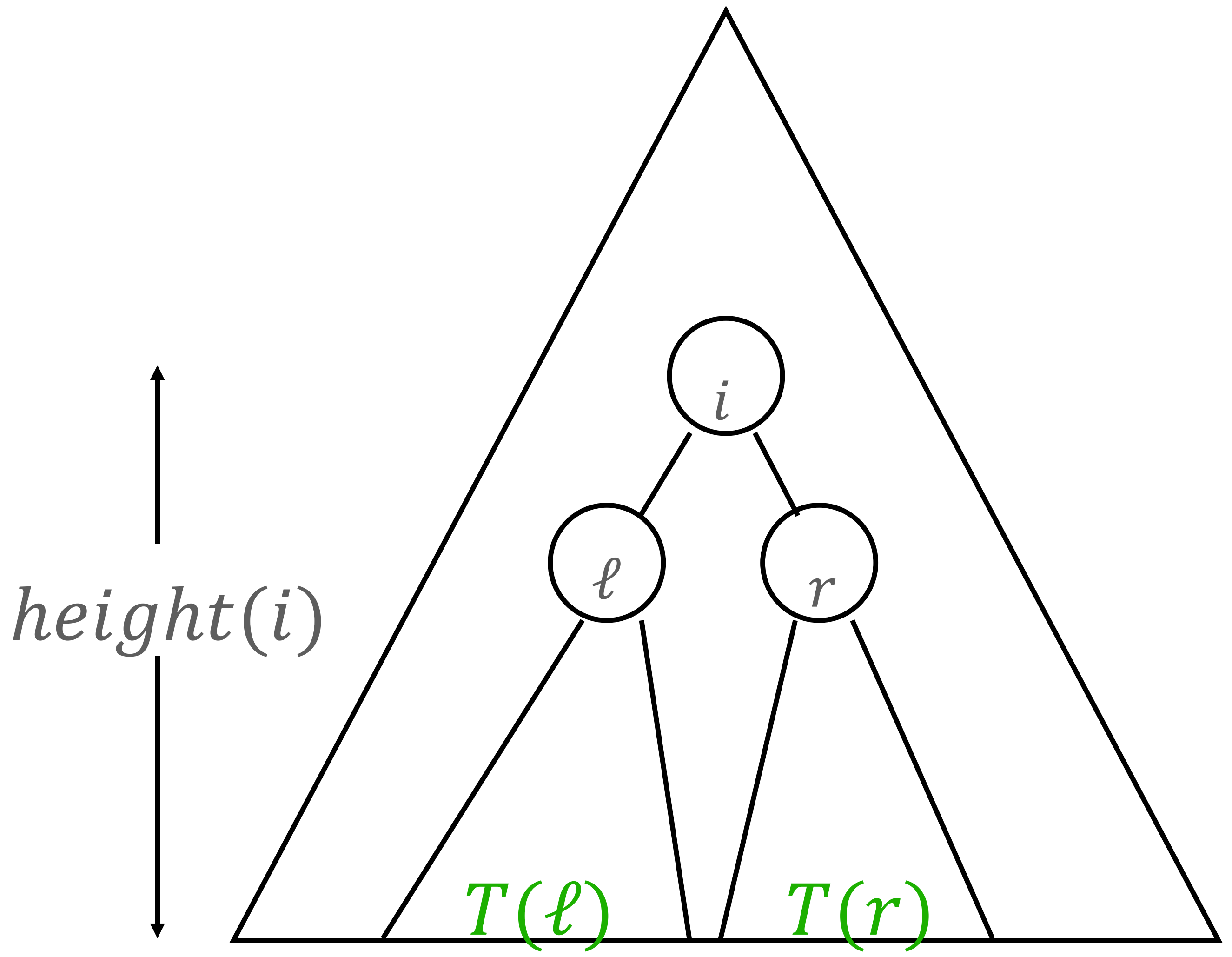
```
heapify(A,i):

A(h)  = max{A(i), A(l), A(r)}
h=i: done
h !=i: swap(A(i), A(h)) /* y=A(i);A(i)=A(h);A(h)=y*/
if h leaf {done} else {heapify(A,h)}
```

run time: O(height(i))

building a heap from scratch

Algorithm 11 Build-heap
Input: array $A[1..n]$
Output: afterwards, A satisfies the heap property
1: ~~*heap-size* := *n*~~
2: **for** $i = \lfloor n/2 \rfloor, \ldots, 1$ **do**
3: Heapify(A, i)

establishing heap



- x: node
  - $\ell(x)$: left son
  - r(x): right son
- # nodes = n

heap property:

$\forall i \neq root. A[p(i)] \geq A[i]$

```
heapify(A,i):

A(h) = max{A(i), A(l), A(r)}
h=i: done
h !=i: swap(A(i), A(h)) /* y=A(i);A(i)=A(h);A(h)=y*/
if h leaf {done} else {heapify(A,h)}
```

run time: O(height(i))

building a heap from scratch

Algorithm 11 Build-heap

Input: array $A[1..n]$

Output: afterwards, A satisfies the heap property

~~1: *heap-size* := n~~

2: **for** $i = \lfloor n/2 \rfloor, \ldots, 1$ **do**

3: Heapify(A, i)

$\lceil \frac{n}{2} \rceil$ nodes are leaves $\Rightarrow$ $x = 0 \xrightarrow{\text{at most}} \lceil \frac{n}{2} \rceil$ subtrees

$x = 1 \xrightarrow{\text{at most}} \lceil \frac{n}{4} \rceil$ subtrees

$\vdots$

runtime $t(n)$: $\left\lceil \frac{n}{2^{x+1}} \right\rceil$

For each height x at most $\cancel{n/2^x}$ subtrees of height x

height(n) = $\lceil \log n \rceil$

$$
\begin{aligned}
t(n) &= O\Big(\sum_{x=0}^{\lceil \log n \rceil} \frac{nx}{2^{x+1}}\Big) \\
&< n \cdot O\Big(\sum_{x=1}^{\lceil \log n \rceil} \frac{x}{2^x}\Big) \\
&< n \cdot O\Big(\sum_{x=1}^{2} \frac{x}{2^x} + \sum_{x=3}^{\infty} \frac{x}{2^x}\Big) \\
&< n \cdot \Big(O(1) + O\big(\sum_{x=3}^{\infty} (\tfrac{3}{4})^x\big)\Big) \\
&= n \cdot O(1) \quad \text{(convergent geometric series)}
\end{aligned}
$$

$\frac{x}{2^x} = \frac{2^x \cdot x}{4^x} < \frac{3^x}{4^x}$

heap sort

Algorithm 12 Heap sort

Input: array $A[1..n]$

Output: afterwards, A is sorted

```
Build-heap(A)
for i := n, ..., 2 do
    Swap(A[1], A[i])
    heap-size := heap-size − 1
    ~~Heapify(A, 1)~~    /* A[i..n] IS SORTED AND CONTAINS THE n − i + 1
    LARGEST ELEMENTS */
```

$heapify(A[1: heap - size], 1)$

heap sort

Algorithm 12 Heap sort

Input: array $A[1..n]$

Output: afterwards, A is sorted

```
Build-heap(A)
for i := n, ..., 2 do
    Swap(A[1], A[i])
    heap-size := heap-size − 1
    ~~Heapify(A, 1)~~   /* A[i..n] IS SORTED AND CONTAINS THE n − i + 1 LARGEST ELEMENTS */
```

$heapify(A[1{:}\, heap - size], 1)$

correctness: exercise

heap sort

Algorithm 12 Heap sort

Input: array $A[1..n]$

Output: afterwards, A is sorted

```
Build-heap(A)
for i := n, ..., 2 do
    Swap(A[1], A[i])
    heap-size := heap-size − 1
    ~~Heapify(A, 1)~~   /* A[i..n] IS SORTED AND CONTAINS THE n − i + 1
    LARGEST ELEMENTS */
```

$heapify(A[1: heap - size], 1)$

correctness: exercise

run time T(n):

- Build-heap: O(n)
- n calls of heapify, each time O(log n)

heap sort

Algorithm 12 Heap sort

Input: array $A[1..n]$
Output: afterwards, A is sorted

```
Build-heap(A)
for i := n, ..., 2 do
    Swap(A[1], A[i])
    heap-size := heap-size − 1
    ~~Heapify(A, 1)~~   /* A[i..n] IS SORTED AND CONTAINS THE n − i + 1 LARGEST ELEMENTS */
```

$heapify(A[1: heap - size], 1)$

correctness: exercise

run time T(n):

- Build-heap: O(n)
- n calls of heapify, each time O(log n)

$$\boxed{T(n) = O(n\log n)}$$

binary locate

an easy variant of binary search

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \ldots, a[n]\} \quad a[1] < x < a[n]$

Output: $\begin{cases} \cancel{m \quad \text{if there is an } 1 \le m \le n \text{ with } a[m] = x} \\ \cancel{-1 \quad \text{otherwise}} \end{cases}$ $\quad i$ with $\quad a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** /* $0 \le \ell < r \le n+1$ AND $a[\ell] < x < a[r]$ */
  $m := \lfloor \frac{\ell + r}{2} \rfloor$;
  if $a[m] = x$ **then**
    return m;
  if $a[m] < x$ **then**
    $\ell := m$
  else
    $r := m$;
return -1;

---

binary locate

an easy variant of binary search

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \dots, a[n]\} \quad a[1] < x < a[n]$

Output: $\cancel{\begin{cases} m & \text{if there is an } 1 \le m \le n \text{ with } a[m] = x \\ -1 & \text{otherwise} \end{cases}}$ $\quad i$ with $a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** /* $0 \le \ell < r \le n + 1$ AND $a[\ell] < x < a[r]$ */
  $m := \lfloor \frac{\ell + r}{2} \rfloor$;
  ~~**if** $a[m] = x$ **then**~~
    ~~return m;~~
  if $a[m] < x$ **then**
    $\ell := m$
  else
    $r := m$;
return ~~-1;~~ ℓ;

binary locate

an easy variant of binary search

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \ldots, a[n]\} \quad a[1] < x < a[n]$

Output: $\begin{cases} \cancel{m \quad \text{if there is an } 1 \leq m \leq n \text{ with } a[m] = x} \\ \cancel{-1 \quad \text{otherwise}} \end{cases}$ $\quad i$ with $a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** $\quad$ /* $0 \leq \ell < r \leq n+1$ AND $a[\ell] < x < a[r]$ */
    $m := \lfloor \frac{\ell + r}{2} \rfloor$;
    ~~**if** $a[m] = x$ **then**~~
        ~~return m;~~
    if $a[m] < x$ **then**
        $\ell := m$
    else
        $r := m$;
return ~~-1;~~ ℓ;

log n comparisons

binary insert

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \ldots, a[n]\} \quad a[1] < x < a[n]$

Output: $\begin{cases} \cancel{m \quad \text{if there is an } 1 \le m \le n \text{ with } a[m] = x} \\ \cancel{-1 \quad \text{otherwise}} \end{cases}$ $\quad i \;$ with $\; a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** $\quad$ /* $0 \le \ell < r \le n+1$ AND $a[\ell] < x < a[r]$ */
$\quad m := \lfloor \frac{\ell + r}{2} \rfloor$;
$\quad$ ~~**if** $a[m] = x$ **then**~~
$\quad\quad$ ~~return m;~~
$\quad$ **if** $a[m] < x$ **then**
$\quad\quad \ell := m$
$\quad$ **else**
$\quad\quad r := m$;
return $\cancel{-1;}$ ℓ;

log n comparisons

binary insert

input: $a[1:n]$ as above sorted, $x \notin A$

```
insert-array(a,x):

if x < a[1] {output (x, a[1],..., a[n])}
if x > a[n] {output (a[1],..., a[n],x)}
i = binary-locate(a,x);
output (a[1],...,a[i],x,a[i+1],..., a[n])
```

insertion sort

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \ldots, a[n]\}$ $\quad a[1] < x < a[n]$

Output: $\begin{cases} \cancel{m \quad \text{if there is an } 1 \leq m \leq n \text{ with } a[m] = x} \\ \cancel{-1 \quad \text{otherwise}} \end{cases}$ $\quad i$ with $a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** /* $0 \leq \ell < r \leq n+1$ AND $a[\ell] < x < a[r]$ */
  $m := \lfloor \frac{\ell+r}{2} \rfloor$;
  ~~**if** $a[m] = x$ **then**~~
    ~~return m;~~
  if $a[m] < x$ **then**
    $\ell := m$
  else
    $r := m$;
return $\cancel{-1;}$ ℓ;

log n comparisons

binary insert

input: $a[1:n]$ as above sorted, $x \notin A$

```
insert-array(a,x):

if x < a[1] {output (x, a[1],..., a[n])}
if x > a[n] {output (a[1],..., a[n],x)}
i = binary-locate(a,x);
output (a[1],...,a[i],x,a[i+1],..., a[n])
```

insertion-sort

input $((a[1], \ldots a[n])$ pairwise different.
output: sorted sequence $b = (b[1], \ldots, b[n])$

```
b=(a[1]);
for i = 2 to n
{b = insert-array(a[i], b}
```

O(n log n) comparisons, time $O(n^2)$

insertion sort

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $\quad x \notin A = \{a[1], \ldots, a[n]\} \quad a[1] < x < a[n]$

Output: $\begin{cases} \cancel{m \quad \text{if there is an } 1 \le m \le n \text{ with } a[m] = x} \\ \cancel{-1} \quad \cancel{\text{otherwise}} \end{cases}$ $\quad i$ with $a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** $\quad$ /* $0 \le \ell < r \le n + 1$ AND $a[\ell] < x < a[r]$ */
$\quad m := \lfloor \frac{\ell + r}{2} \rfloor$;
$\quad$ ~~**if** $a[m] = x$ **then**~~
$\quad\quad$ ~~return m;~~
$\quad$ **if** $a[m] < x$ **then**
$\quad\quad \ell := m$
$\quad$ **else**
$\quad\quad r := m$;
return ~~-1;~~ ℓ;

log n comparisons

binary insert

input: $a[1:n]$ as above sorted, $x \notin A$

```
insert-array(a,x):

if x < a[1] {output (x, a[1],..., a[n])}
if x > a[n] {output (a[1],..., a[n],x)}
i = binary-locate(a,x);
output (a[1],...,a[i],x,a[i+1],..., a[n])
```

shifting

time O(n) (with array)

insertion-sort

input $((a[1], \ldots a[n])$ pairwise different.
output: sorted sequence $b = (b[1], \ldots, b[n])$

```
b=(a[1]);
for i = 2 to n
{b = insert-array(a[i], b}
```

n times n-shifting

O(n log n) comparisons, time $O(n^2)$

insertion sort

Algorithm 1 ~~Binary search~~ binary-locate(a,x)

Input: Sorted array $a[1..n]$, $a[1] < a[2] < \cdots < a[n]$, element x $x \notin A = \{a[1], \ldots, a[n]\}$ $a[1] < x < a[n]$

Output: $\cancel{\begin{cases} m & \text{if there is an } 1 \le m \le n \text{ with } a[m] = x \\ -1 & \text{otherwise} \end{cases}}$ i with $a[i] < x < a[i+1]$

$\ell := 0$; $r := n + 1$;
while $\ell + 1 < r$ **do** /* $0 \le \ell < r \le n+1$ AND $a[\ell] < x < a[r]$ */
  $m := \lfloor \frac{\ell + r}{2} \rfloor$;
  ~~**if** $a[m] = x$ **then**~~
    ~~return m;~~
  if $a[m] < x$ **then**
    $\ell := m$
  else
    $r := m$;
return ~~-1;~~ ℓ;

$\boxed{\text{log n comparisons}}$

binary insert

input: $a[1:n]$ as above sorted, $x \notin A$

```
insert-array(a,x):

if x < a[1] {output (x, a[1],..., a[n])}
if x > a[n] {output (a[1],..., a[n],x)}
i = binary-locate(a,x);
output (a[1],...,a[i],x,a[i+1],..., a[n])
```

time O(n) (with array)

insertion-sort

input $((a[1], \ldots a[n])$ pairwise different.
output: sorted sequence $b = (b[1], \ldots, b[n])$

```
b=(a[1]);
for i = 2 to n
{b = insert-array(a[i], b}
```

O(n log n) comparisons, time $O(n^2)$

let's use balanced trees instead of arrays for this!