Carleton University

Alina Shaikhet

COMP 2402

Heaps

part 1

Priority Queue

Main operations:

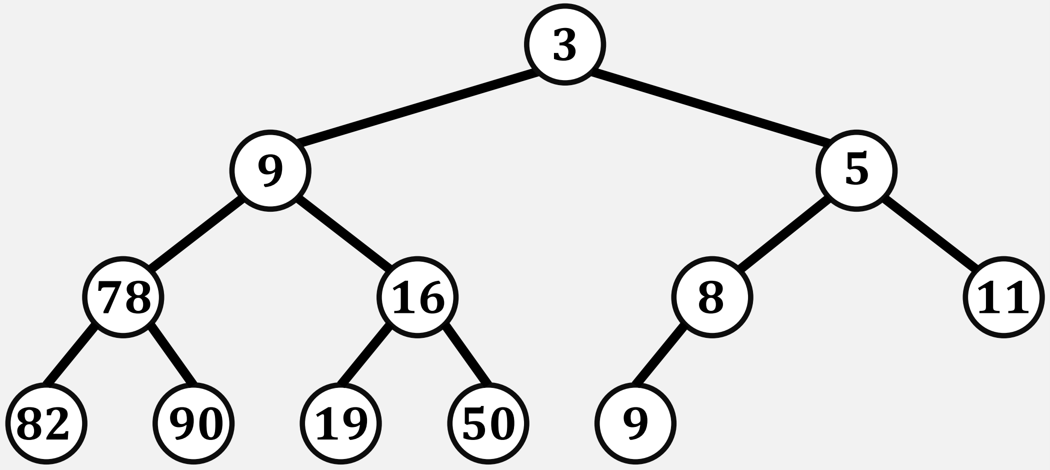
- add($\boldsymbol{x}$)
- removeMin()
- findMin()

Additional operations:

- remove($\boldsymbol{u}$)
- decreaseKey($\boldsymbol{u}$)

$\boldsymbol{u}$ is the pointer to a node/element

Binary Heap



every level (except possibly the bottom) is completely full

Binary Heap is a **complete** binary tree with the **heap property.**

The minimum value is stored at the root.

Heap/(priority queue) **allows duplicate values.**

for each non-root node u:
$u.\mathrm{x} \geq u.\mathrm{parent}.x$

Min-Heap

Supports:

- **insert** an element x to the heap ← add(x)
- **extract minimum** element:
  - return the smallest element
  - remove it from the heap

$\left.\right\}$ removeMin()

Heap is an implementation of a **priority queue**

Eytzinger Method

This method allows for the numbering of ancestors beginning with a descendant.

Allows to present family trees and pedigree charts in text format.


146 TABVLA GENERALIS, 147

Ad Exemplum vnici Henrici 3 regis Galliæ omnibus deinceps reliquis applicanda.

1. FILIVS Henric⁹ 3.

2. PATER Henric⁹.

3. MATER Catharina.

4. Auus Paternus. Francisc⁹.

5. Auia paterna. Claudia.

6. Auus maternus. Laurenti⁹.

7. Auia materna. Magdalena

8. Proauus paternus a parte aui paterni Carolus.

9. Proauia paterna parte aui paterni Ludouica

10. Proau⁹ paternus a parte auiæ paternæ Ludouicus

11. Proauia paterna a parte auiæ paternæ Anna

12. Proau⁹ matern⁹ a parte aui materni Petrus.

13. Proauia materna a parte aui materni Alphonsina.

14. Proau. matern⁹ a parte auiæ maternæ Ioannes.

15. Proauia materna a parte auiæ maternæ Ioanna.

PATERNI progenitores.

16. Abauus paternus a parte Aui paterni (Ioannes.)

17. Abauia paterna a parte Aui paterni (Margareta.)

18. Abauus paternus 2 a parte Aui paterni (Philippus.)

19. Abauia paterna 2. a parte Aui paterni (Margareta.)

20. Abauus paternus 3 a parte Auiæ paternæ (Carolus.)

21. Abauia paterna 3 a parte Auiæ paternæ (Maria)

22. Abauus Paternus 4. a parte Auiæ paternæ (Franciscus.)

23. Abauia paterna 4. a parte Auiæ paternæ (Margareta.)

MATERNI progenitores.

24. Abauus maternus a parte Aui materni (Laurentius.)

25. Abauia materna a parte Aui materni (Clarixa.)

26. Abauus maternus 1. a parte Aui materni. N.

27. Abauia materna 2 a parte Aui materni. N.

28. Abauus maternus 3. a parte Auiæ maternæ (Bertrandus.)

29. Abauia materna 3 a parte Auiæ maternæ (Ludouica)

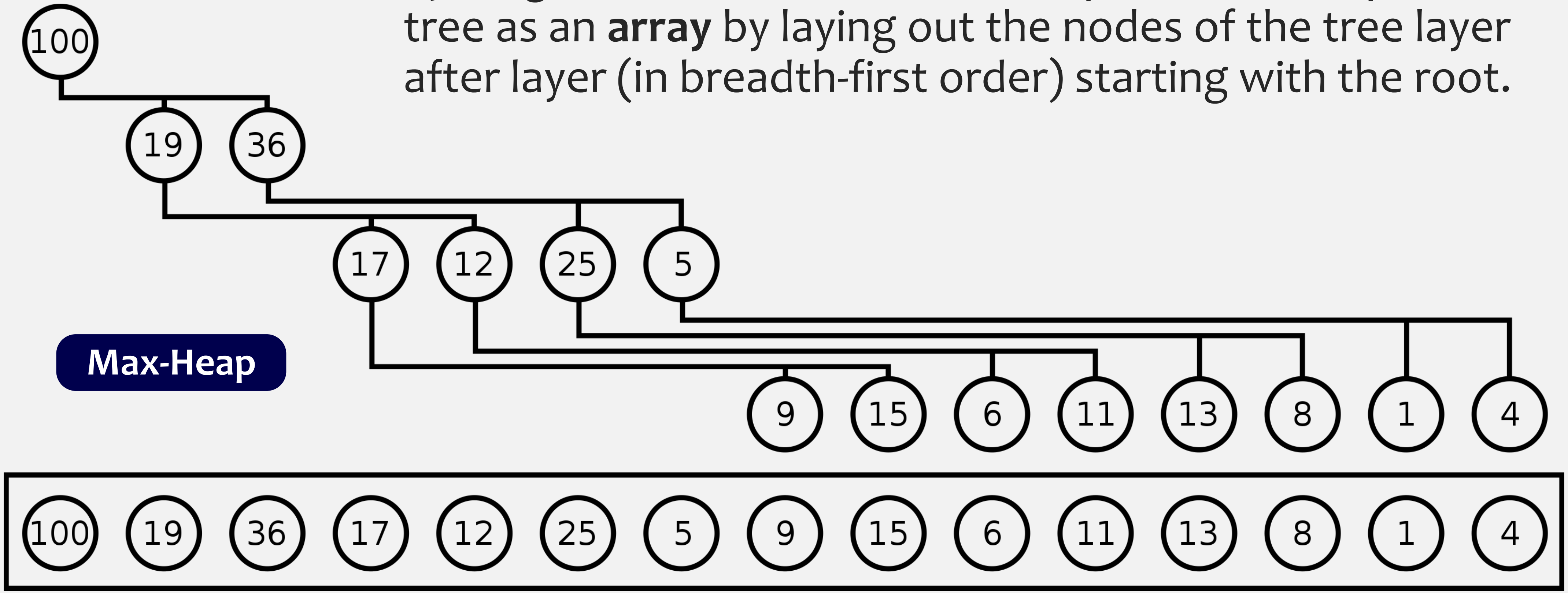
30. Abauus maternus 4. a parte Auiæ maternæ (Ioannes.)

31. Abauia materna 4. a parte Auiæ maternæ (N. filia præsignani.)

https://en.wikipedia.org/wiki/Genealogical_numbering_systems

Heap using an Array

Eytzinger's method allows us to represent a complete binary tree as an **array** by laying out the nodes of the tree layer after layer (in breadth-first order) starting with the root.



https://en.wikipedia.org/wiki/Heap_(data_structure)

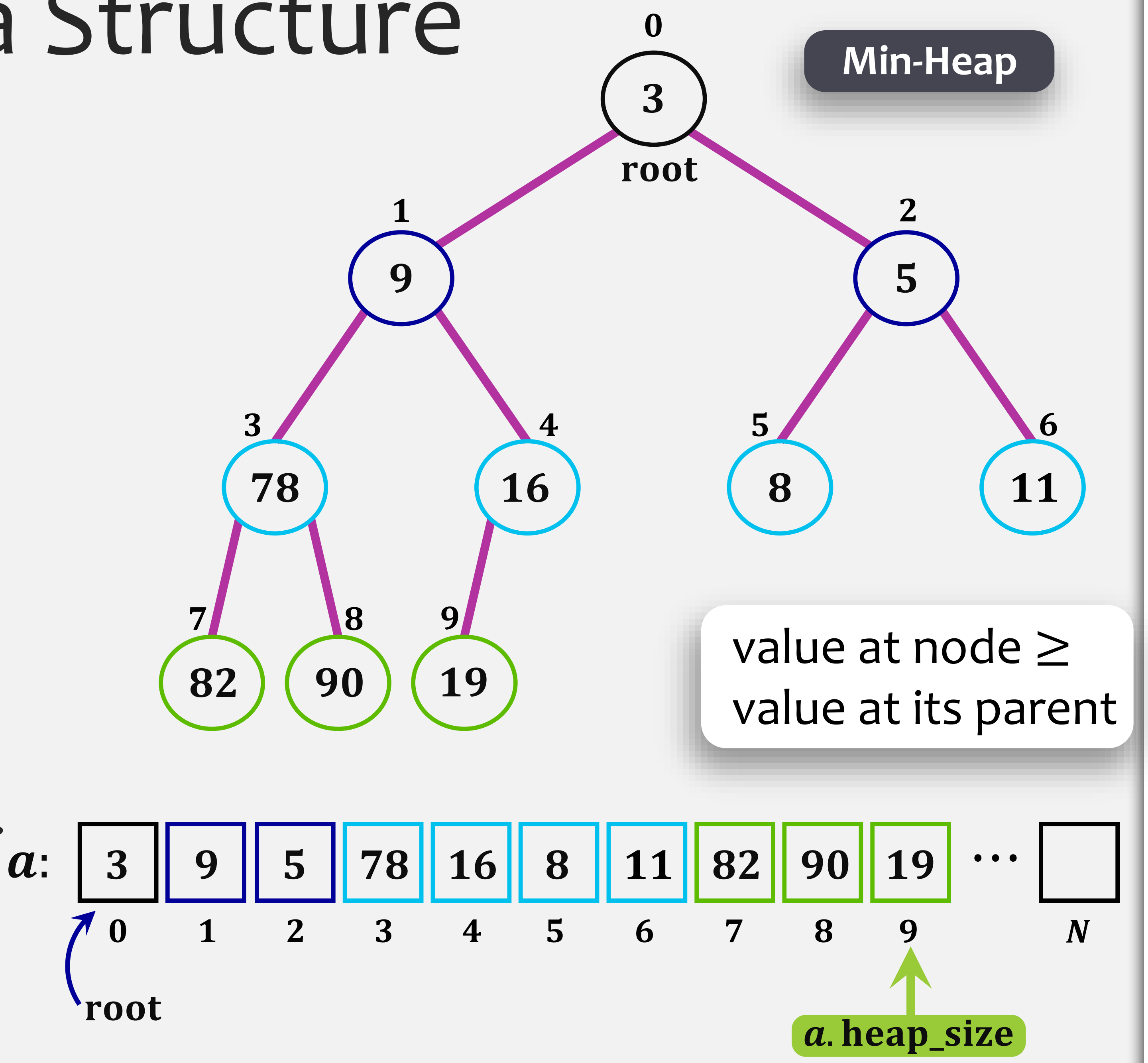
Binary Heap Data Structure

It is an array that we can view as a nearly complete binary tree.

Do not confuse heap with BST.

Notice, a heap is not a sorted structure; it can be regarded as being **partially ordered**.

Heap can be used to **sort** elements.

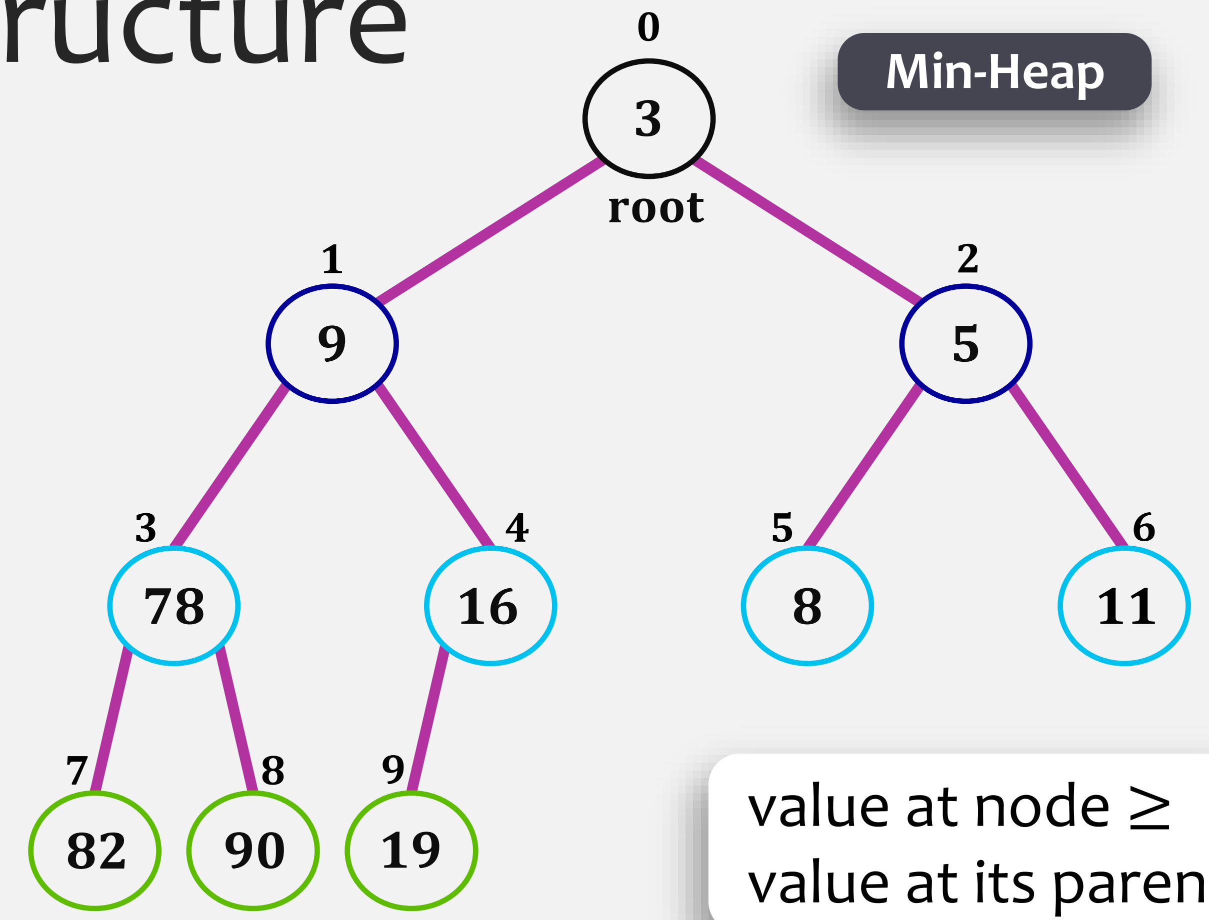


Binary Heap Data Structure

Min-Heap

Array $\boldsymbol{a}[\mathbf{0}..\boldsymbol{n}-\mathbf{1}]$ is a **heap** if for all $\mathbf{0} < \boldsymbol{i} \leq \boldsymbol{n}-\mathbf{1}$:

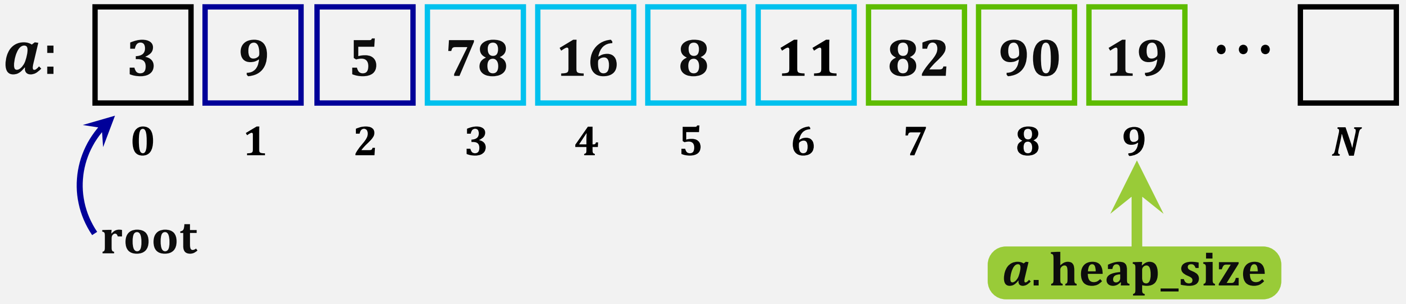
$$\boldsymbol{a}[\text{parent}(\boldsymbol{i})] \leq \boldsymbol{a}[\boldsymbol{i}]$$



value at node $\geq$ value at its parent

Node with index $\boldsymbol{i}$:

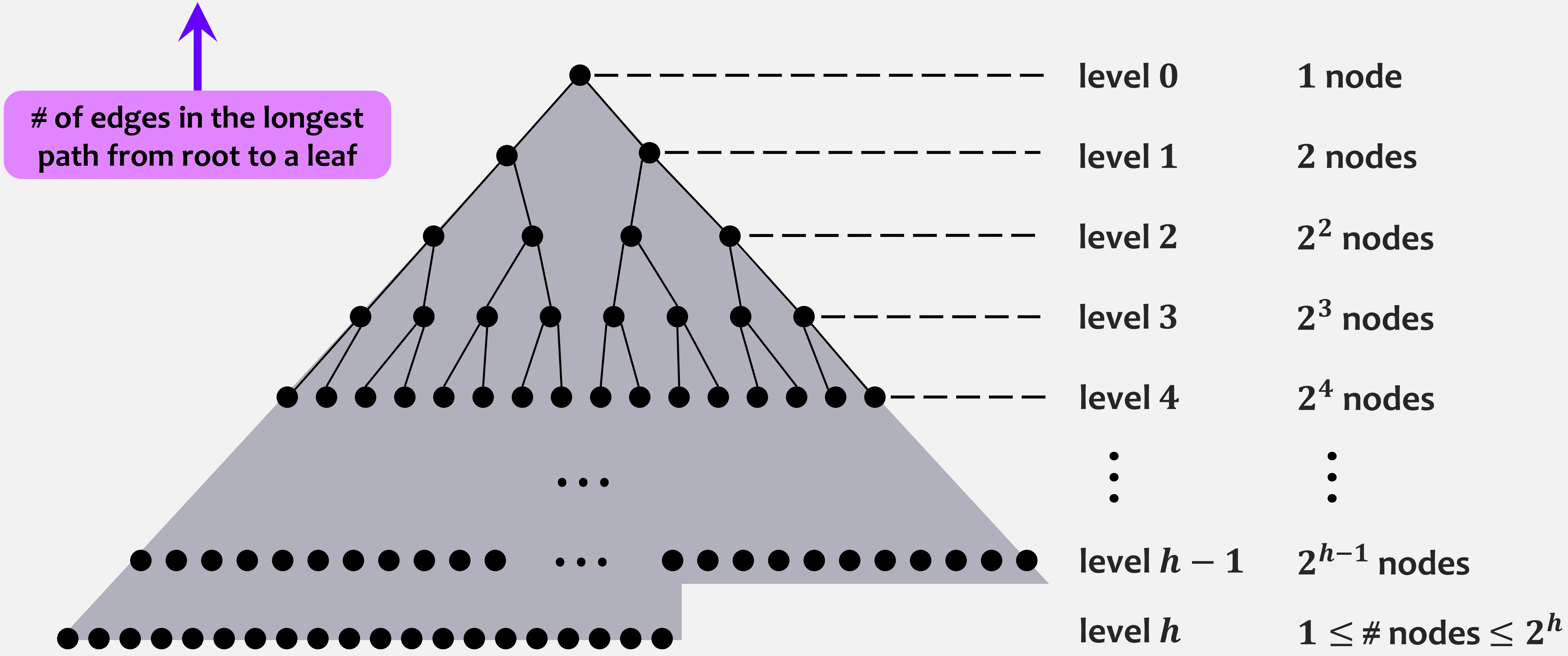
- Parent of $\boldsymbol{i}$ has index $\left\lfloor \frac{\boldsymbol{i}-\mathbf{1}}{\mathbf{2}} \right\rfloor$
- Left child of $\boldsymbol{i}$ has index $\mathbf{2}\boldsymbol{i}+\mathbf{1}$
- Right child of $\boldsymbol{i}$ has index $\mathbf{2}\boldsymbol{i}+\mathbf{2}$



Alina Shaikhet – COMP 2402 – Carleton University

Binary Heap Data Structure

What is the **height** (let us call it $\boldsymbol{h}$) of a heap with $\boldsymbol{n}$ nodes?



Binary Heap Data Structure

$$2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$$

What is the **height** (let us call it $\boldsymbol{h}$) of a heap with $\boldsymbol{n}$ nodes?

$$1 + 2 + 2^2 + \cdots + 2^{h-1} + 1 \le n \le 1 + 2 + 2^2 + \cdots + 2^{h-1} + 2^h$$

$$\left(2^h - 1\right) + 1 \le n \le 2^{h+1} - 1$$

$$2^h \le n < 2^{h+1}$$

$$\underbrace{h \le \overbrace{\log n}}\overbrace{< h + 1}$$

$$\log n - 1 < h \le \log n$$

$$h \le \log n$$

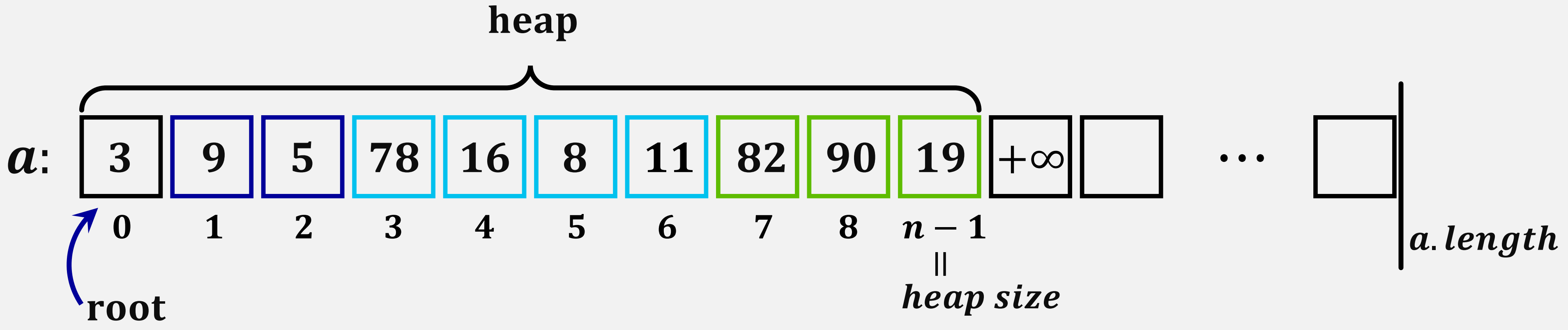
$$\log n < h + 1$$

$$-1 + \log n < h$$

$$h = \lfloor \log n \rfloor$$

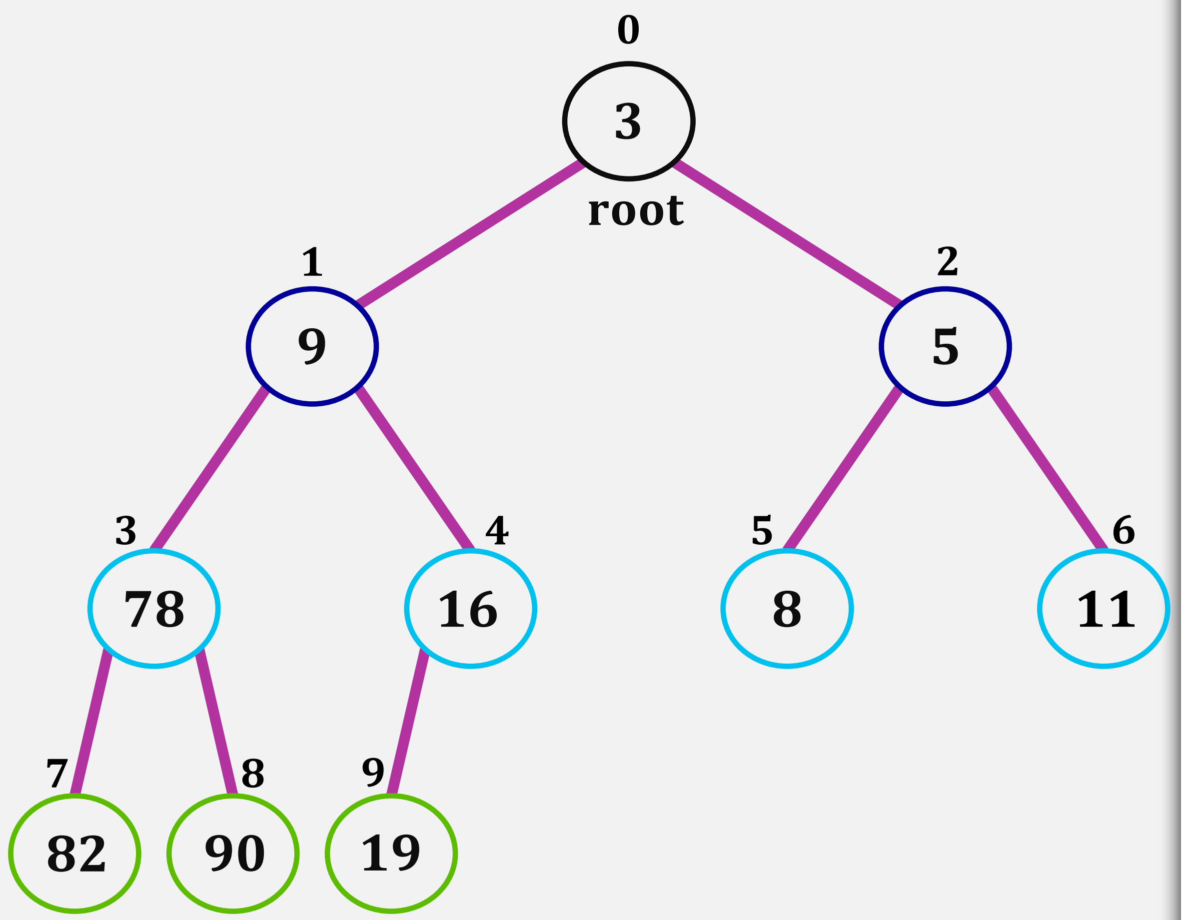
Heaps – Basic Procedures

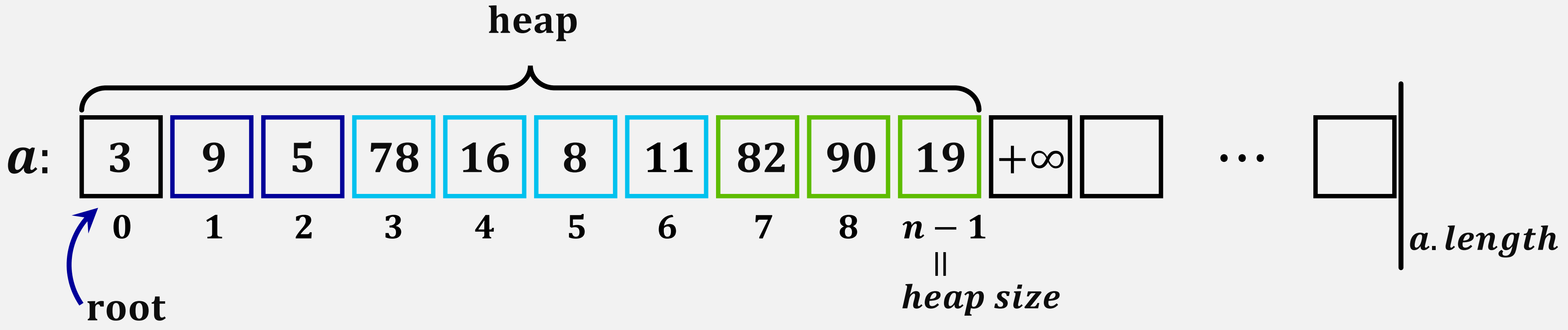
$\boxed{O(1)}$	**findMin()**	– return the smallest element, which is $a[0]$
$\boxed{O(\log n)}$	**decreaseKey(i, x)**	– for $\mathbf{x} \le a[i]$ decrease $a[i]$ to $\mathbf{x}$ and restore the heap property.
$\boxed{O(\log n)}$	**add(x)**	– insert element $\mathbf{x}$ and restore the heap property.
$\boxed{O(\log n)}$	**minHeapify(i)**	– restore the heap property for element $a[i]$.
$\boxed{O(n)}$	**buildMinHeap(a)**	– build a heap from an unordered input array a.
$\boxed{O(\mathrm{n}\log n)}$	**HeapSort(a)**	– sort an array a in place.
$\boxed{O(\log n)}$	**removeMin()**	– return and delete the smallest element, restore the heap property.



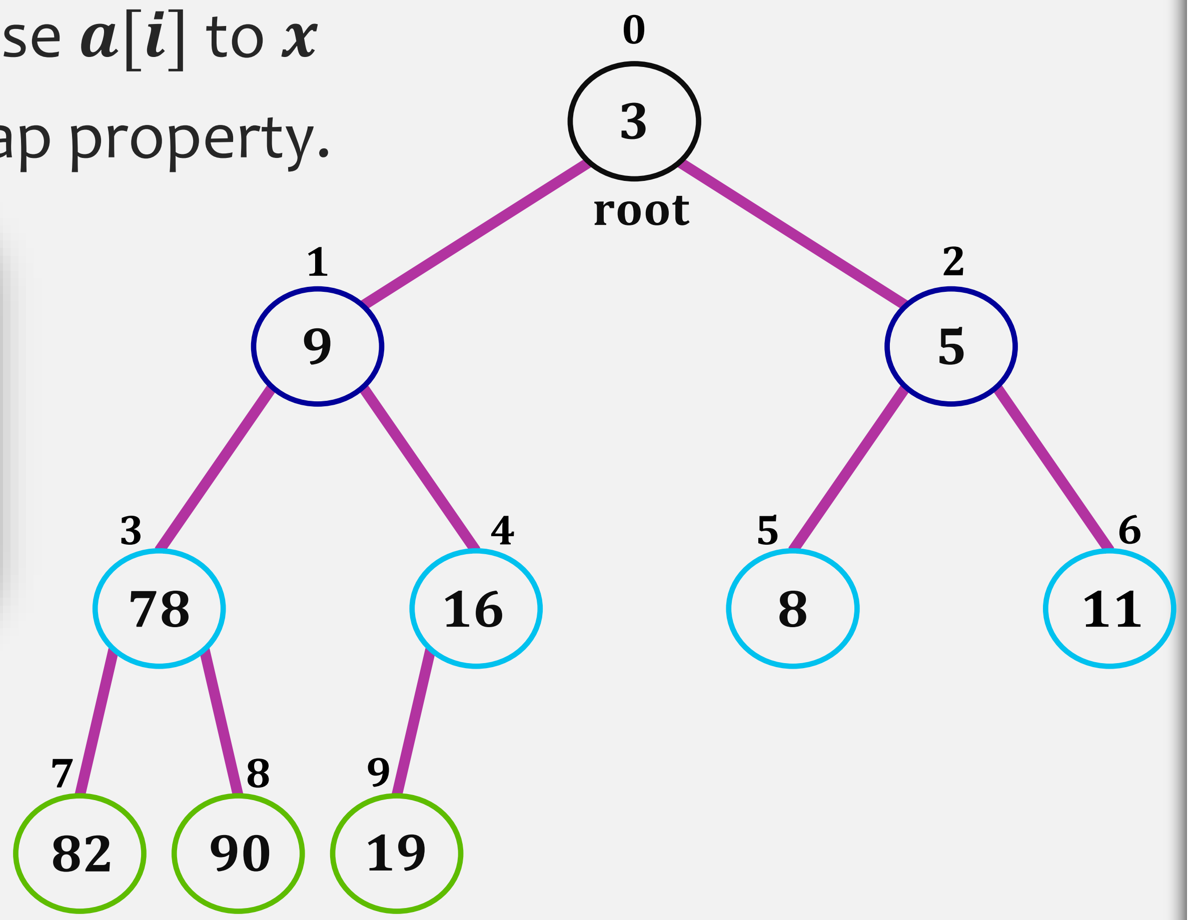
findMin() – return $a[0]$

$\boxed{O(1)}$



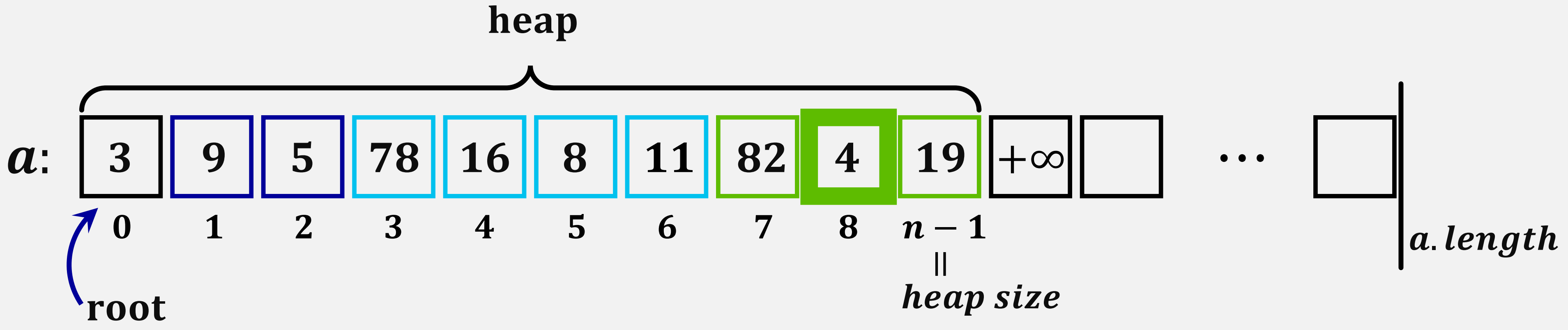


decreaseKey(i, x) – for $x \leq a[i]$ decrease $a[i]$ to x and restore the heap property.



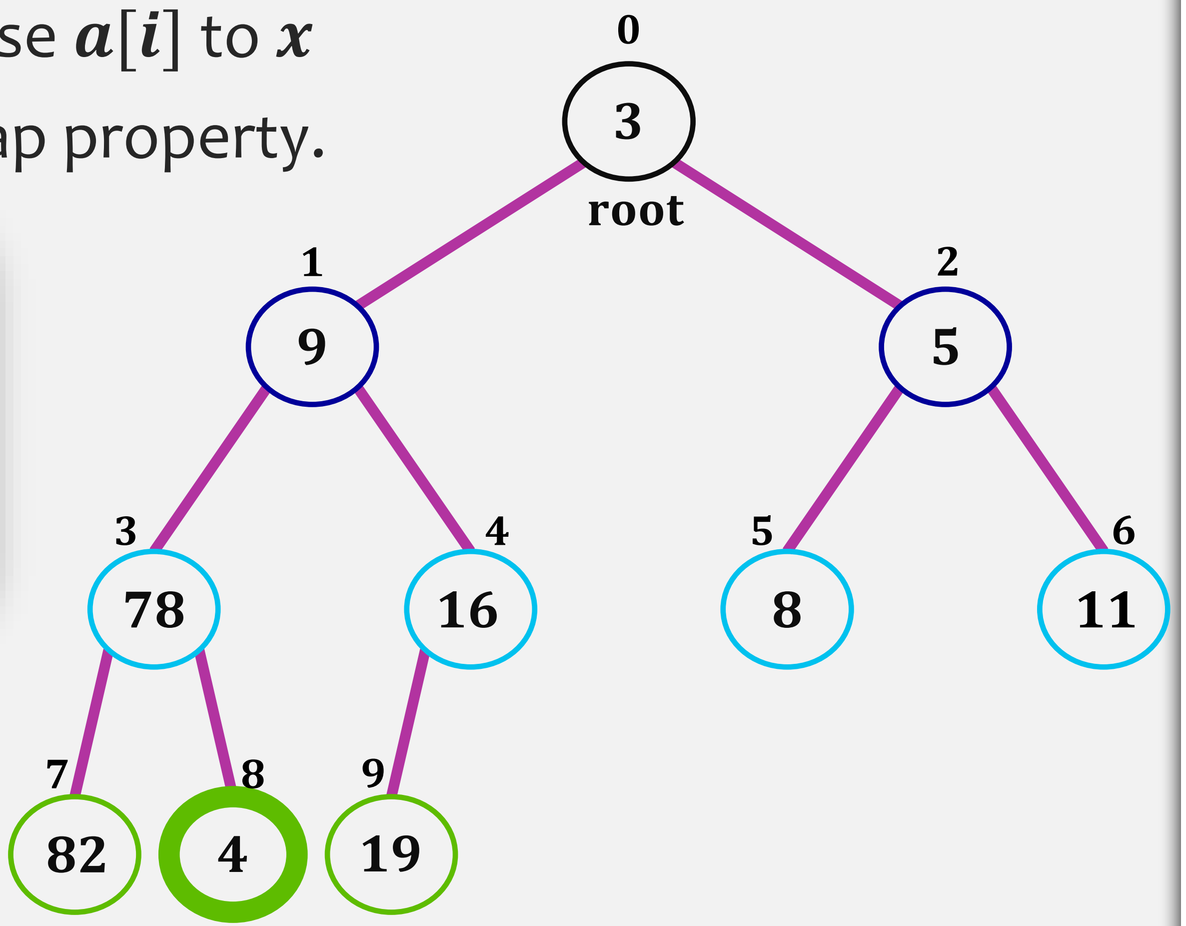
$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;

decreaseKey(8, 4)

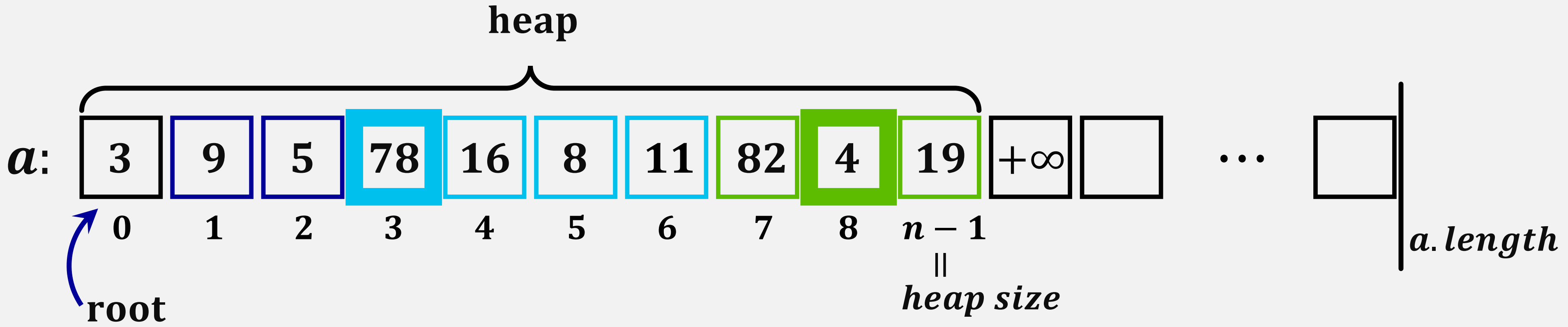


decreaseKey(i, x) – for $x \le a[i]$ decrease $a[i]$ to x and restore the heap property.

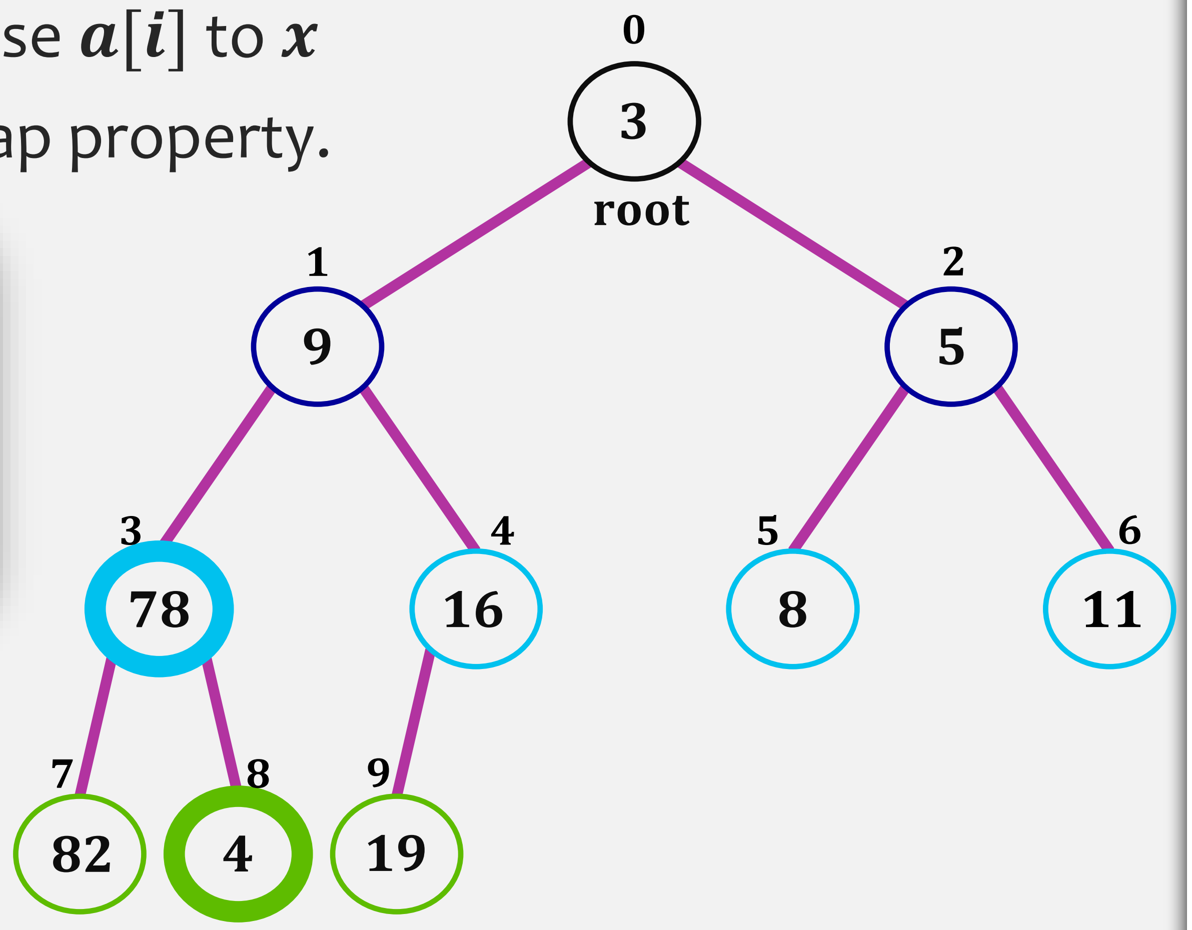
$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;



decreaseKey(8, 4)

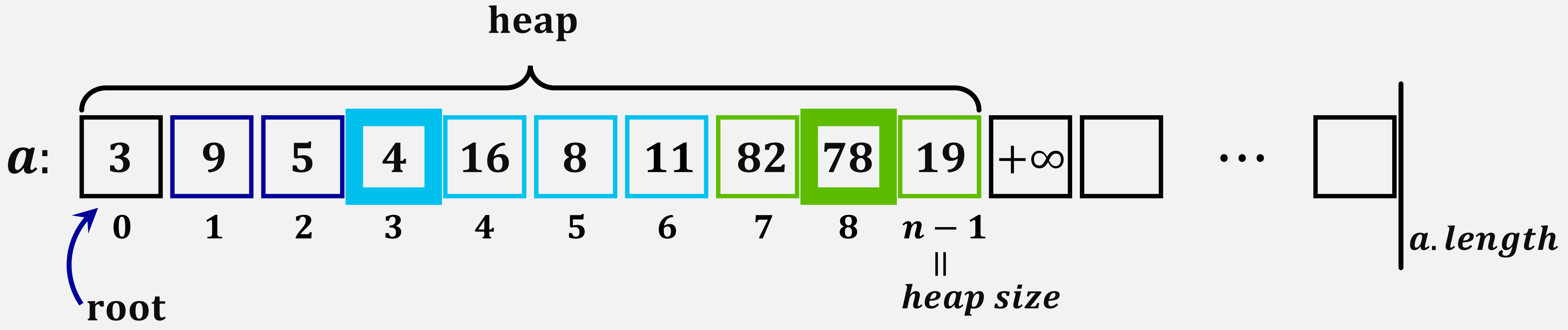


decreaseKey(i, x) – for $x \leq a[i]$ decrease $a[i]$ to x and restore the heap property.



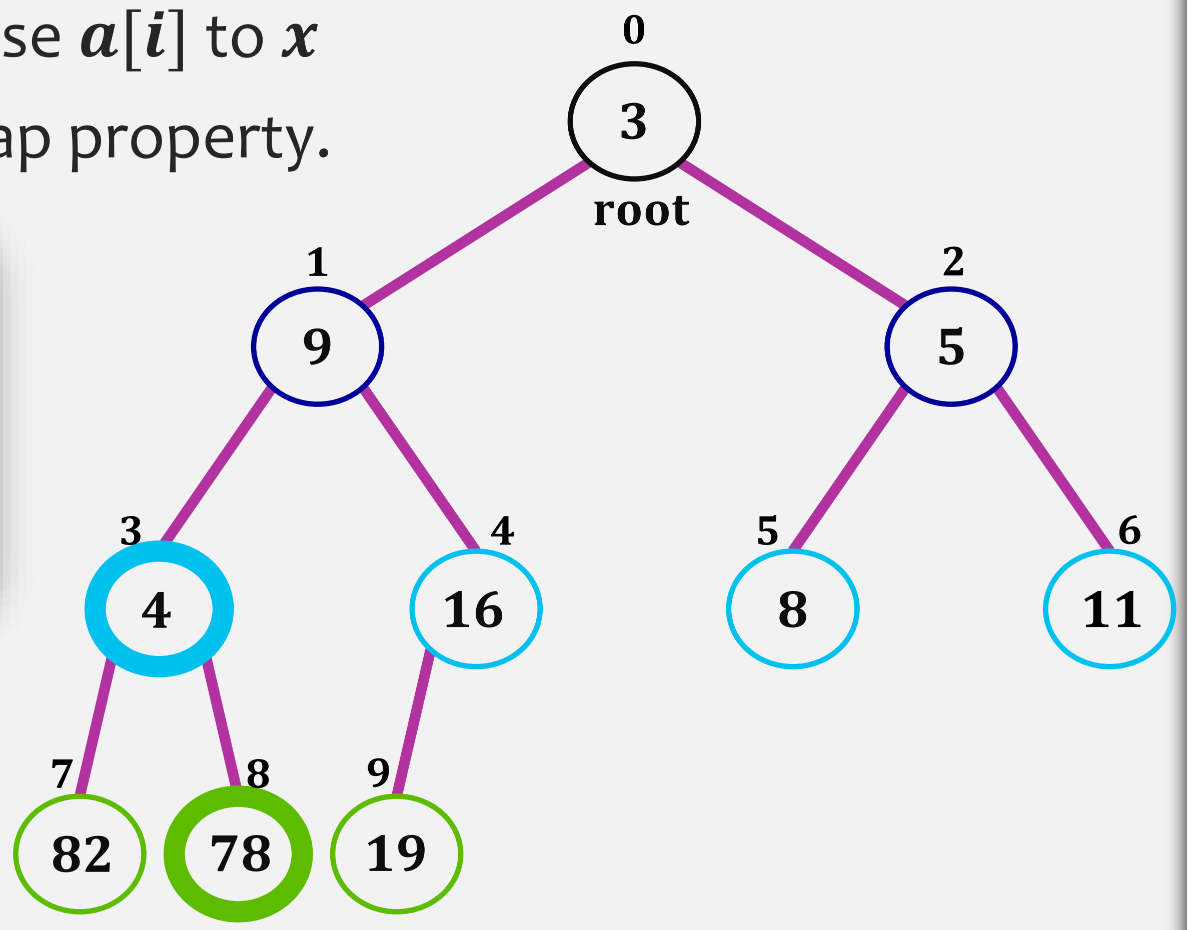
$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;

decreaseKey(8, 4)

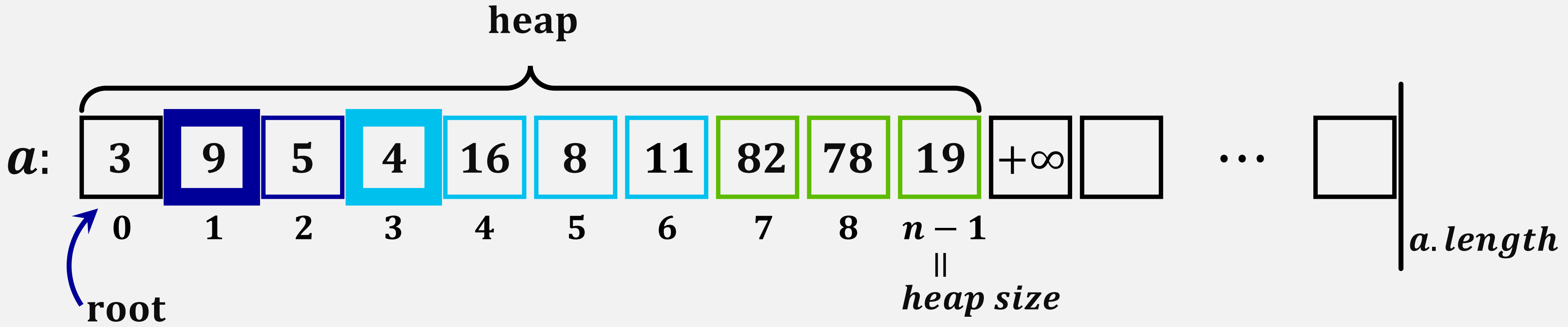


decreaseKey(i, x) – for $x \le a[i]$ decrease $a[i]$ to x and restore the heap property.

$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;



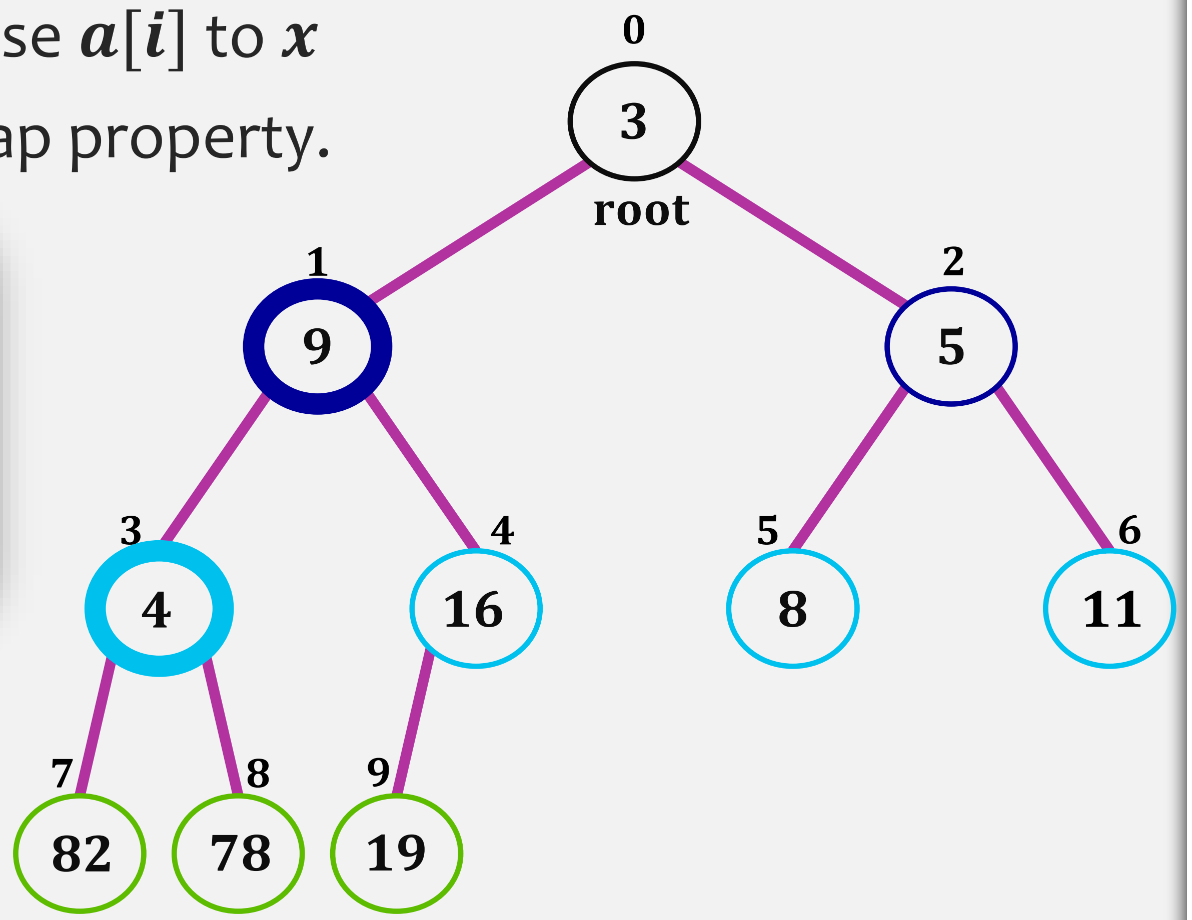
decreaseKey(8, 4)

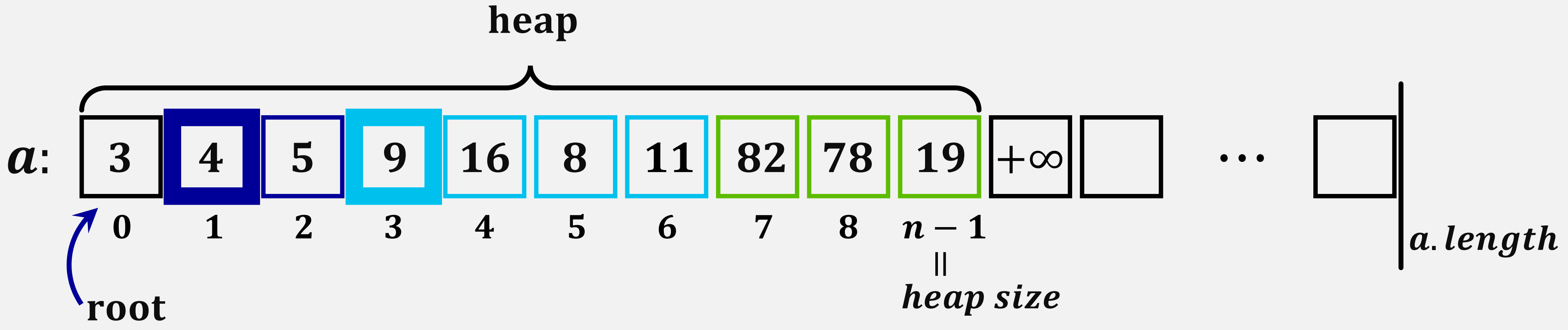


decreaseKey(i, x) – for $x \leq a[i]$ decrease $a[i]$ to x and restore the heap property.

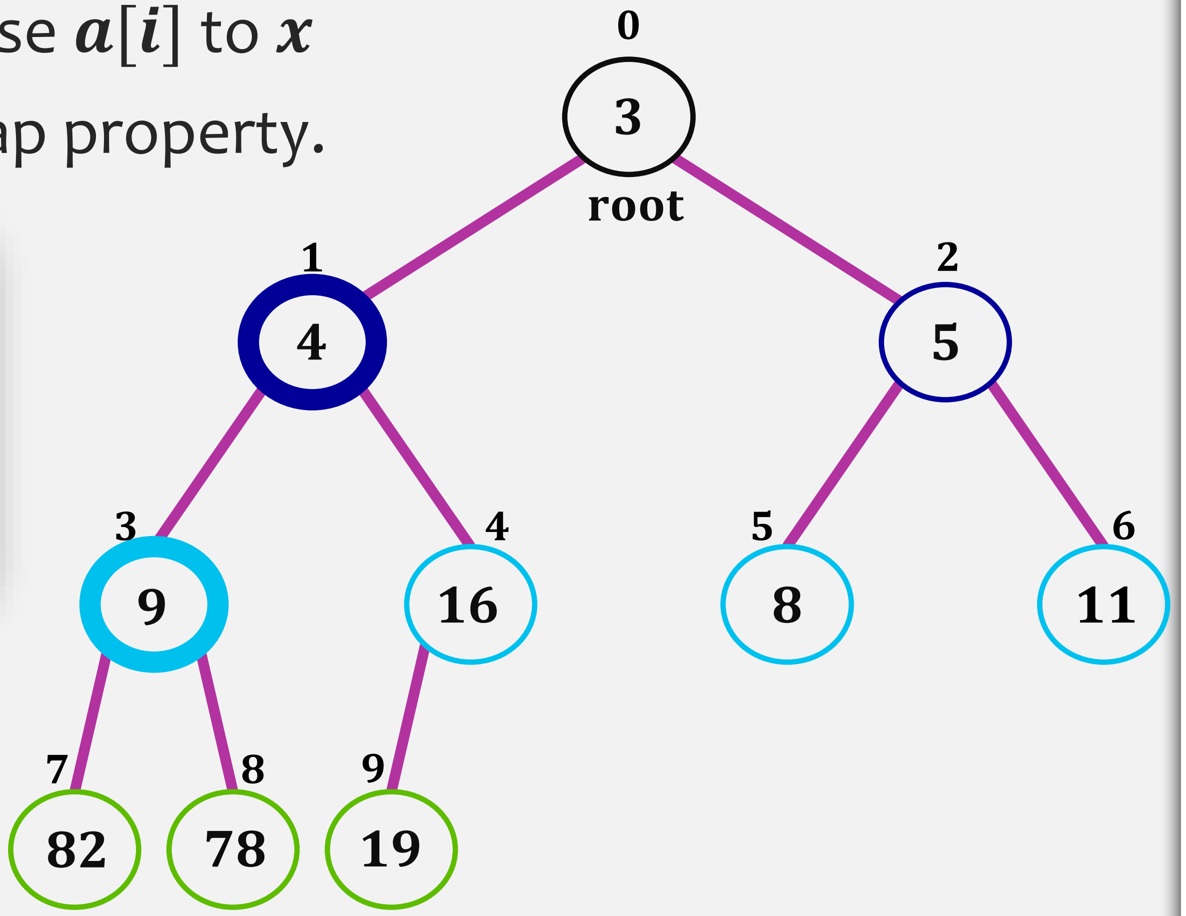
$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;

decreaseKey(8, 4)



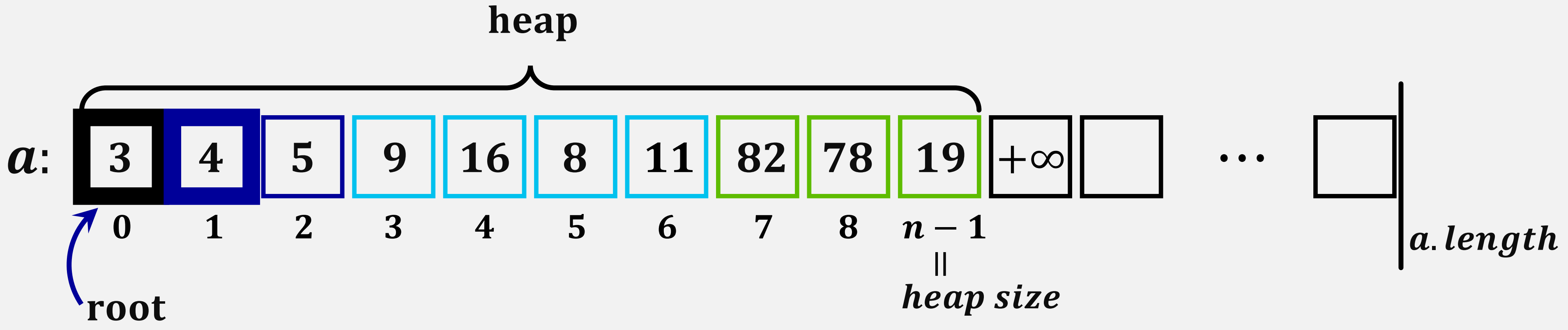


decreaseKey(i, x) – for $x \leq a[i]$ decrease $a[i]$ to x and restore the heap property.



$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;

decreaseKey(8, 4)

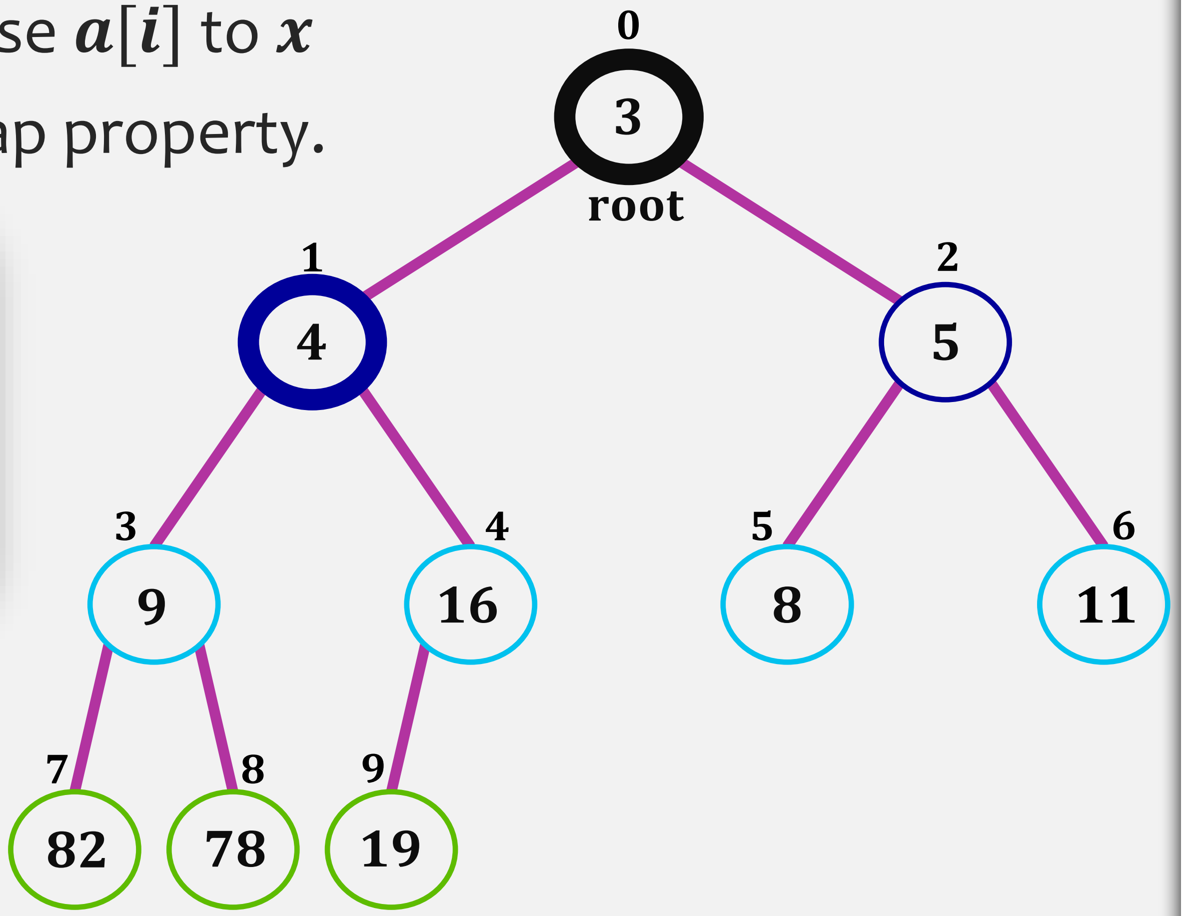


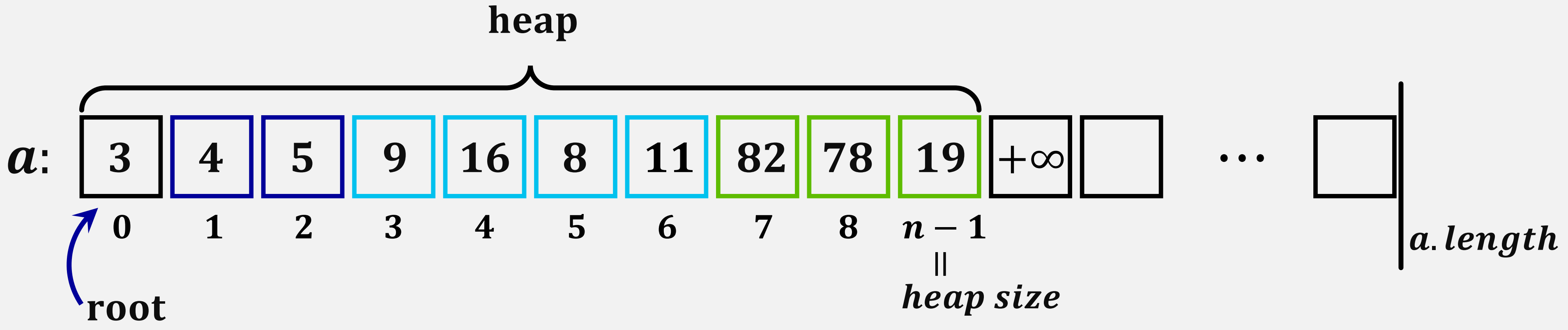
decreaseKey(i, x) – for $x \le a[i]$ decrease $a[i]$ to x and restore the heap property.

$a[i] = x$;
while $i > 0$ and $a[\text{parent}(i)] > a[i]$ do:
    swap $a[i]$ and $a[\text{parent}(i)]$;
    $i = \text{parent}(i)$;

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

decreaseKey(8, 4)



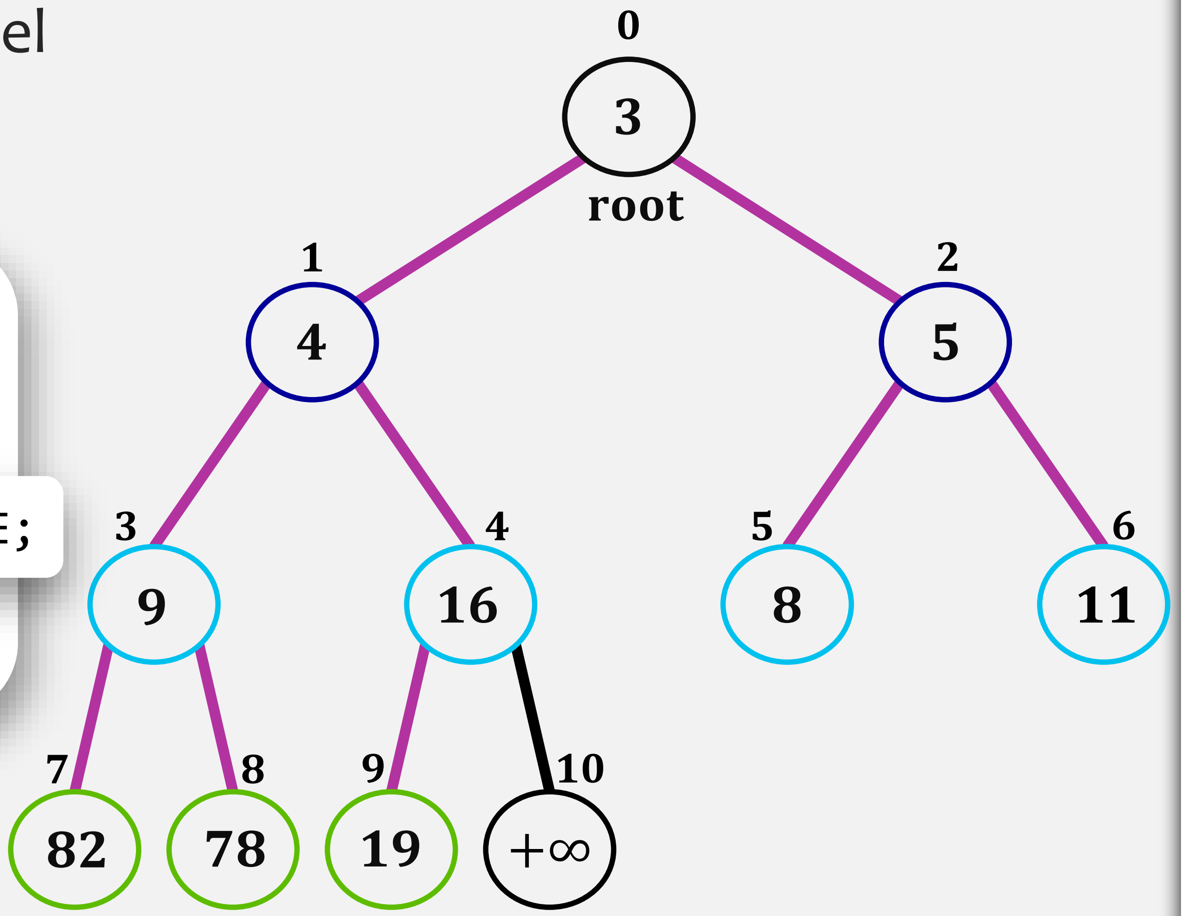


add(x) – add element x to the bottom level of the heap at the leftmost open space and restore the heap property.

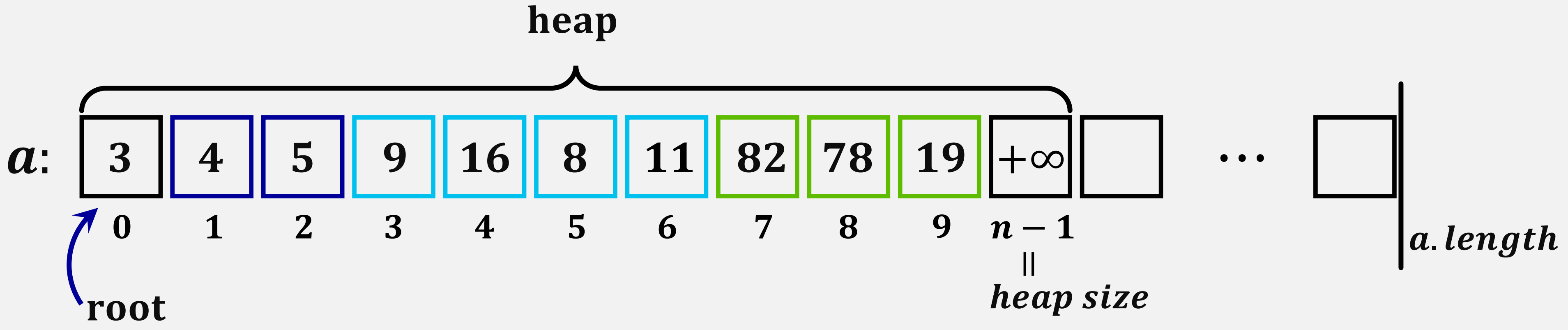
if $(n + 1 > a.\text{length})$ then resize();
$n++$;
$a[n] = +\infty$;
decreaseKey($n - 1, x$);

```
a[n] = Integer.MAX_VALUE;
```

add(12)



Alina Shaikhet – COMP 2402 – Carleton University

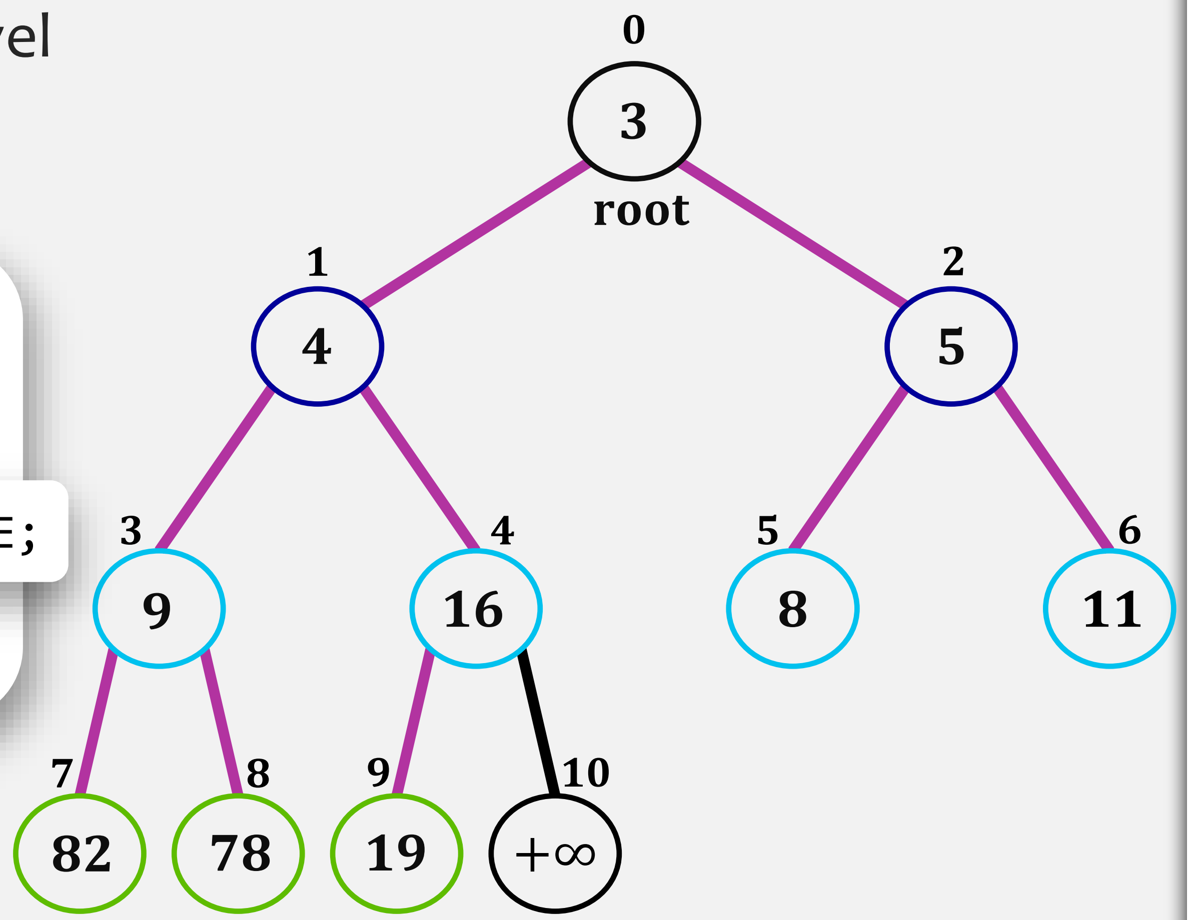


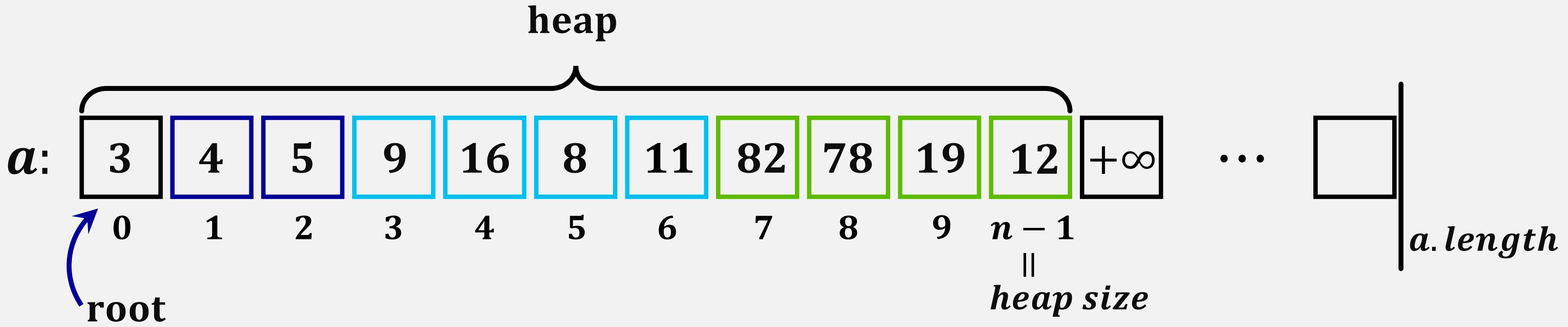
add(x) – add element x to the bottom level of the heap at the leftmost open space and restore the heap property.

if ($n + 1 >$ a.length) then resize();
$n++$;
$a[n] = +\infty$;
decreaseKey($n - 1, x$);

```
a[n] = Integer.MAX_VALUE;
```

add(12)



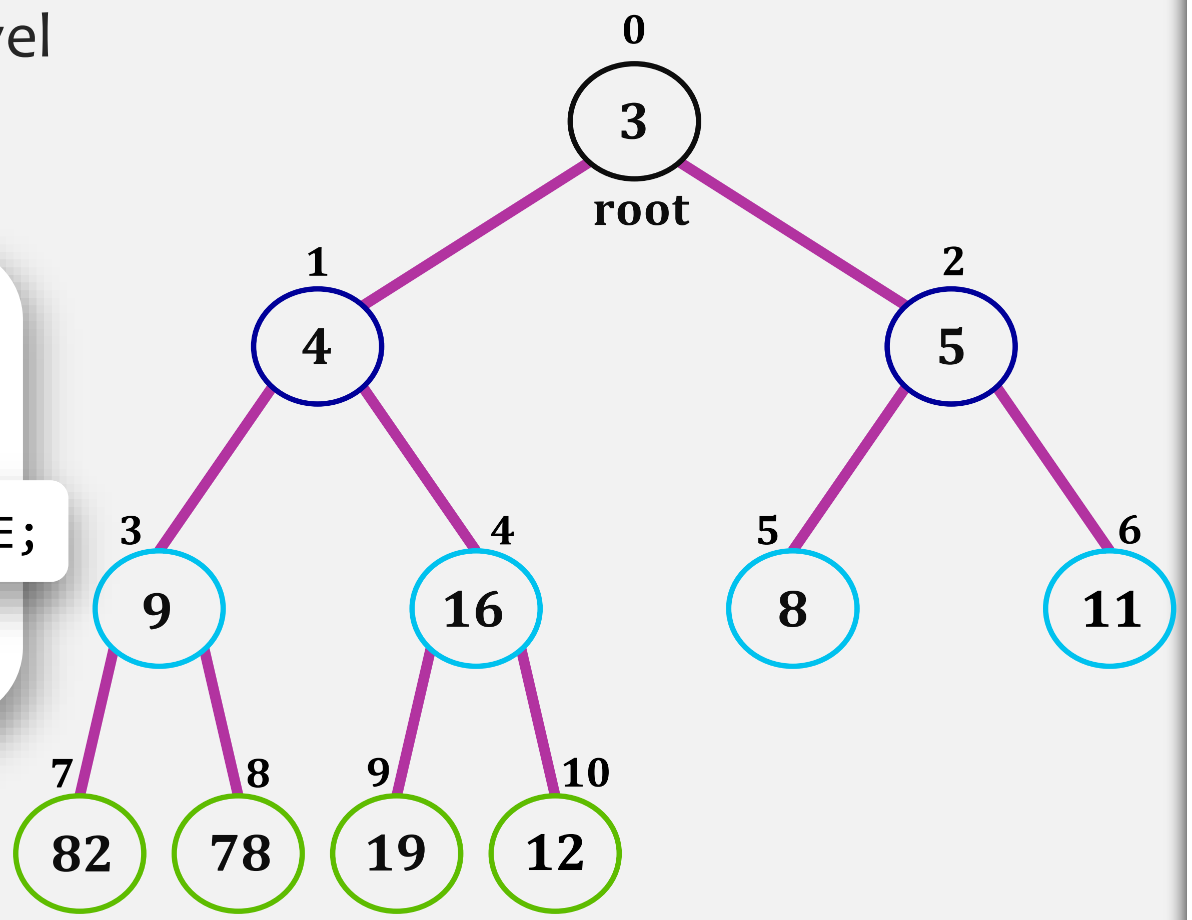


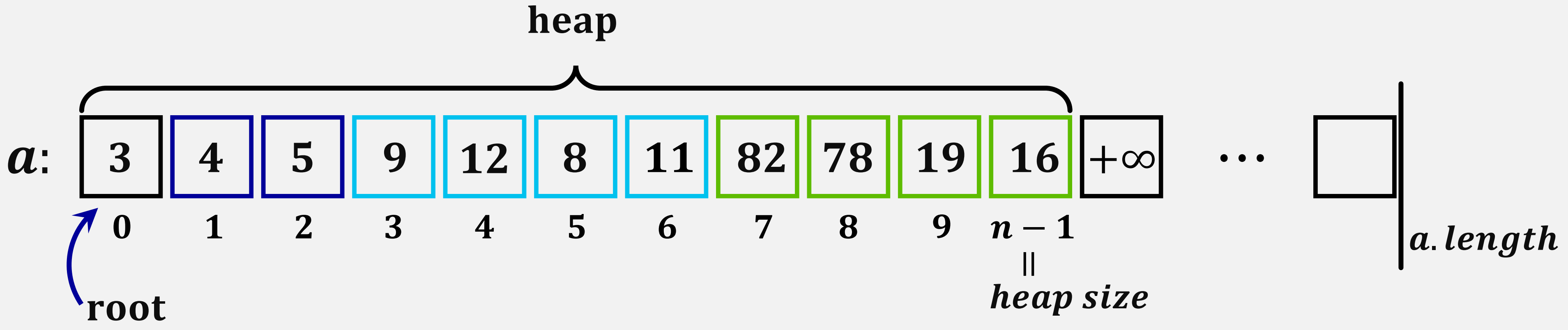
add(x) – add element x to the bottom level of the heap at the leftmost open space and restore the heap property.

if ($n + 1 >$ a.length) then resize();
$n++$;
$a[n] = +\infty$;
decreaseKey($n - 1, x$);

```
a[n] = Integer.MAX_VALUE;
```

add(12)



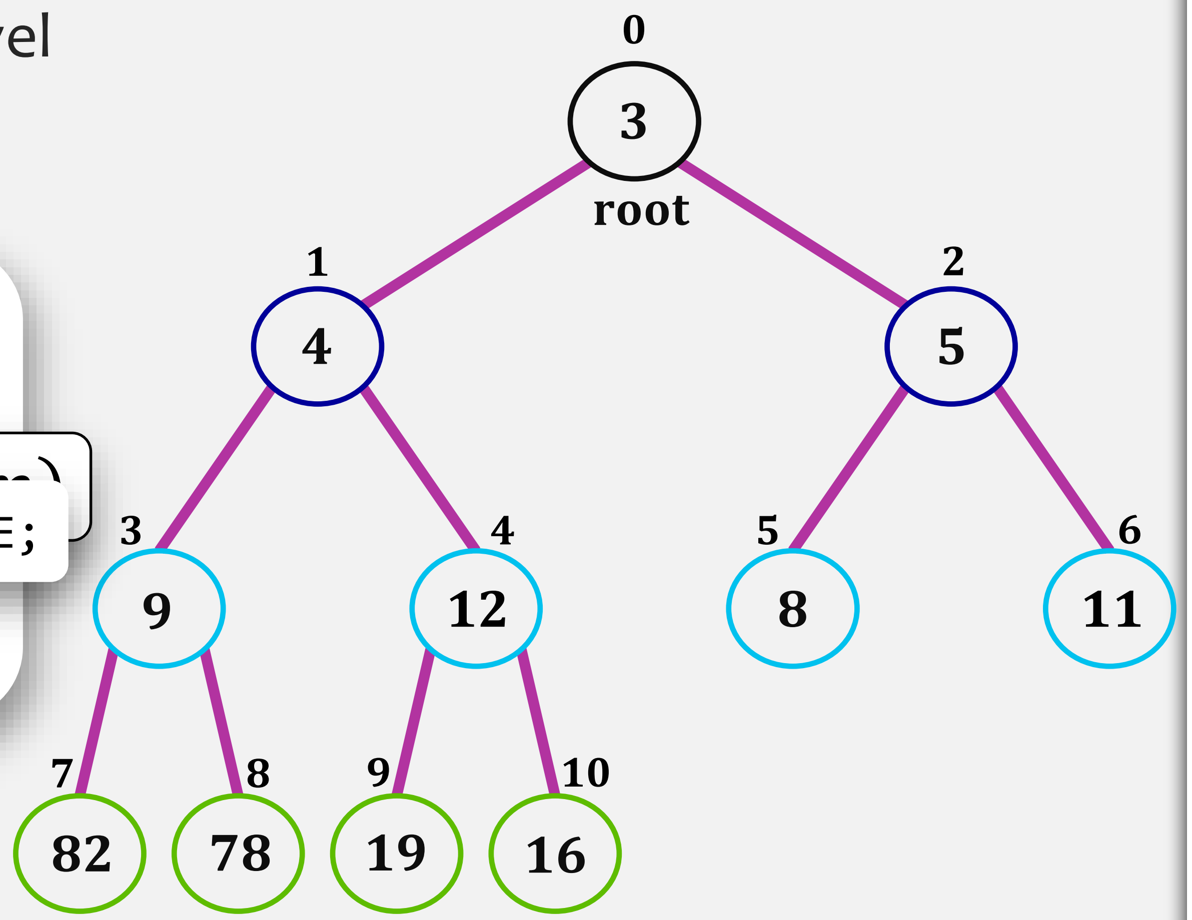


add(x) – add element x to the bottom level of the heap at the leftmost open space and restore the heap property.

if $(n + 1 > \ a$.length) then resize();
$n++;$
$a[n] = +\infty;$
decreaseKey($n - 1, x$);

$O(\log n)$

```
a[n] = Integer.MAX_VALUE;
```



We can build a heap using **add(x)** operation.

$O(n \log n)$

How to build a heap?

Input: array $\boldsymbol{a}[\mathbf{0}..\boldsymbol{N}-\mathbf{1}]$.

Output: heap $\boldsymbol{a}[\mathbf{0}..\boldsymbol{N}-\mathbf{1}]$, containing the same elements

$\boldsymbol{n} = \mathbf{0}$;
while $\boldsymbol{n} < \boldsymbol{N}$:
    add(a[$\boldsymbol{n}$]);

$a[0..n-1]$ **is a heap**

if ($\boldsymbol{n}+\mathbf{1} > \boldsymbol{a}$.length) then resize();
$\boldsymbol{n}++$;
$\boldsymbol{a}[\boldsymbol{n}] = +\infty$;
decreaseKey($\boldsymbol{n}-\mathbf{1}, \boldsymbol{x}$);

We can build a heap using add(x) operation.

$\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$

Time required to build a heap $= \log 1 + \log 2 + \ldots + \log\left(\frac{N}{2}-1\right) + \log\left(\frac{N}{2}\right) + \ldots + \log N = \sum_{n=1}^{N} \log n$

Upper Bound:

$$\sum_{n=1}^{N} \log n \le \sum_{n=1}^{N} \log N = N \cdot \log N = O(N \log N)$$

Lower Bound:

$$\sum_{n=1}^{N} \log n \geqslant \sum_{n=N/2}^{N} \log n \geqslant \sum_{n=N/2}^{N} \log \frac{N}{2} = \log \frac{N}{2} \cdot \left(N - \frac{N}{2} + 1\right) \geqslant$$

$$\geqslant \frac{N}{2} \log \frac{N}{2} = \frac{N}{2}\left(\log N - \log 2\right) = \frac{N}{2} \log N - \frac{N}{2}$$

$$= \frac{N}{4} \log N + \boxed{\frac{N}{4} \log N - \frac{N}{2}} \geqslant$$

positive when $N \geqslant 4$

$$\geqslant \frac{1}{4} N \log N = \Omega(N \log N)$$

$\frac{N}{4} \log N - \frac{N}{2} \geqslant 0$

$\log N \geqslant 2$

$N \geqslant 4$

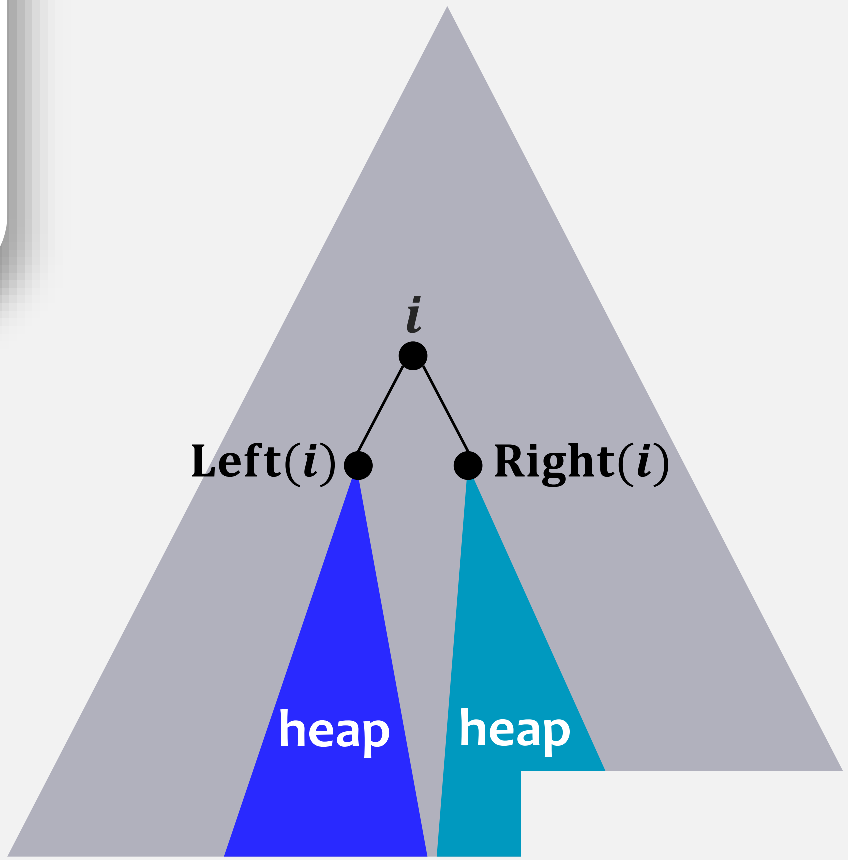
Alina Shaikhet – COMP 2402 – Carleton University

minHeapify(i)

ods book calls this function **trickleDown()**

When **minHeapify(i)** terminates, the subtree rooted at i is a heap.

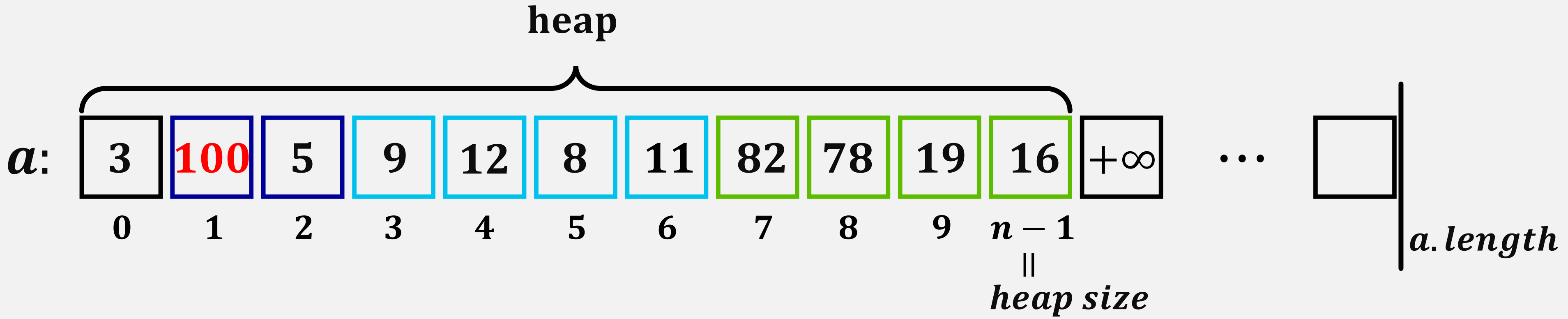
find the smallest of $a[i]$, $a[\mathbf{Left}(i)]$ and $a[\mathbf{Right}(i)]$;
store its index in min;
if $min \neq i$ then
    swap $a[i]$ and $a[min]$;
    minHeapify(min);



minHeapify(i) – restore the heap property for element $a[i]$.

This operation assumes:

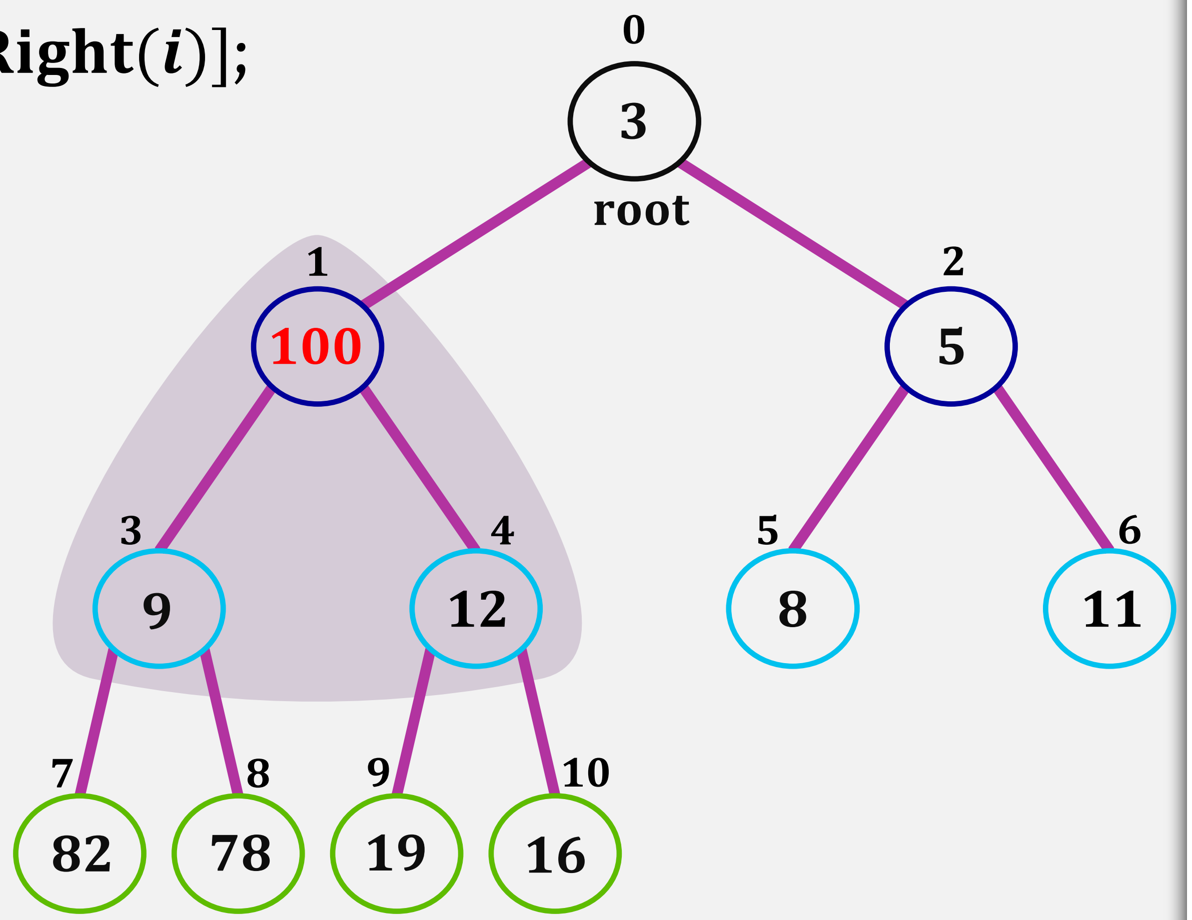
- $0 \leq i \leq n-1$;
- Subtree rooted at $\mathbf{Left}(i)$ is a heap;
- Subtree rooted at $\mathbf{Right}(i)$ is a heap;

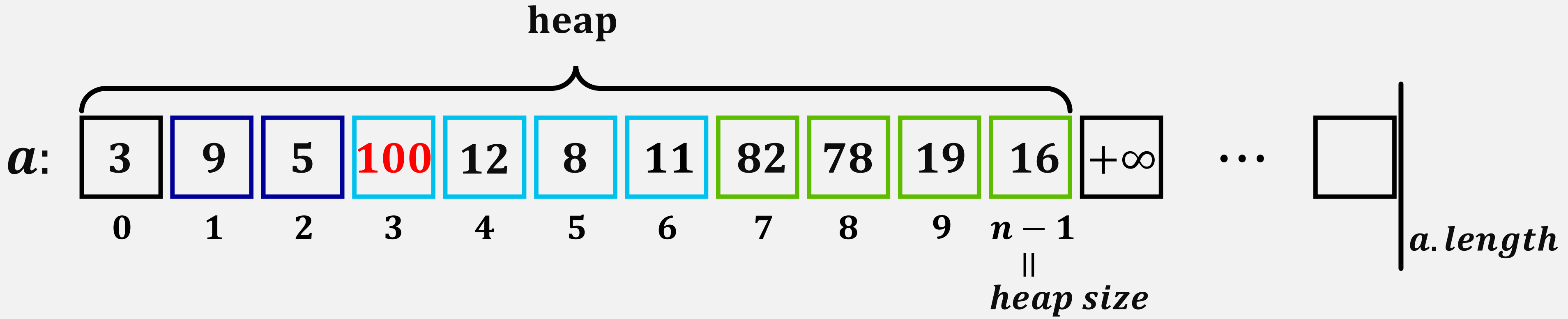


minHeapify(i):

find the smallest of $a[i]$, $a[\mathbf{Left}(i)]$ and $a[\mathbf{Right}(i)]$;
store its index in min;
if $min \neq i$ then
    swap $a[i]$ and $a[min]$;
    minHeapify(min);

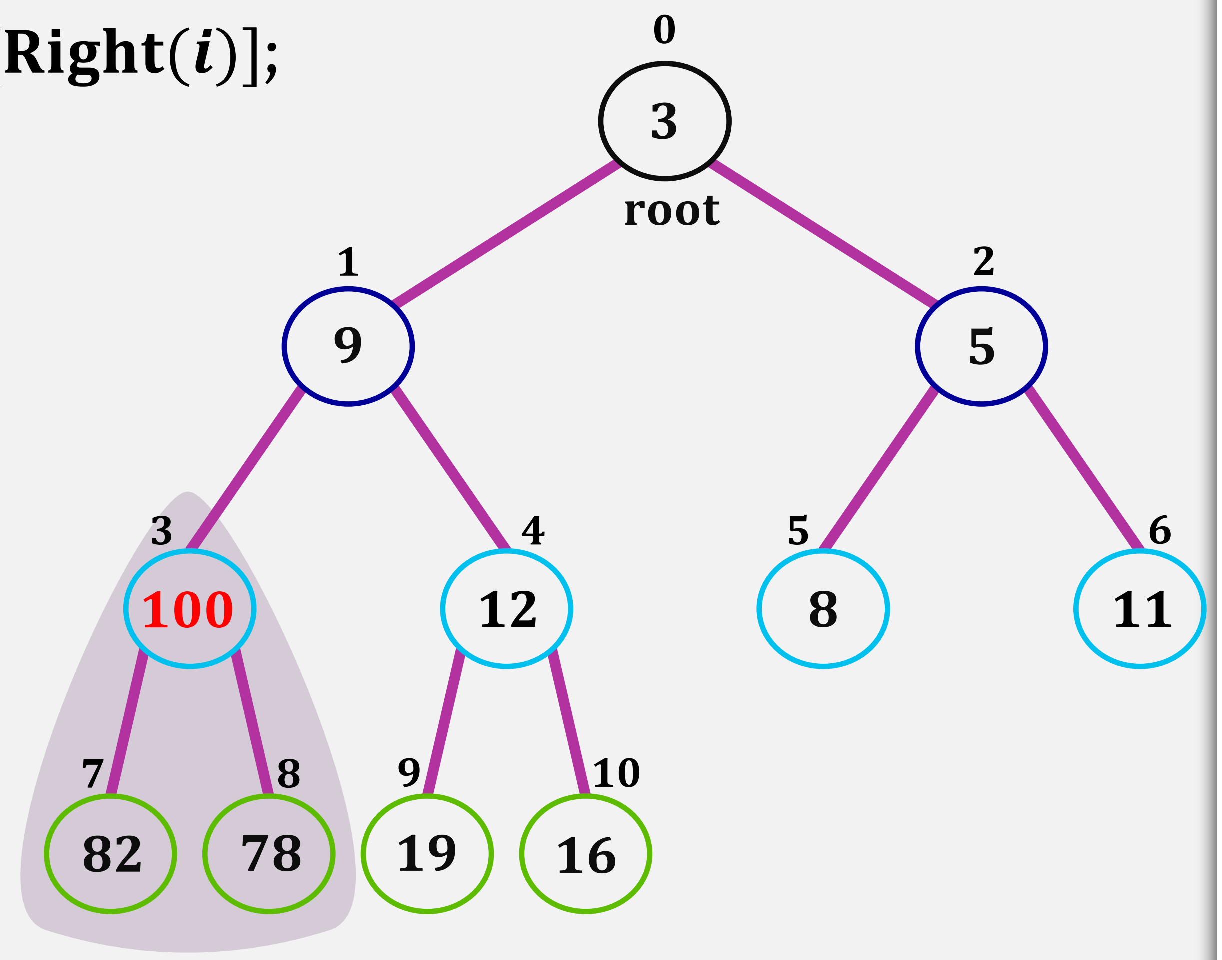
minHeapify(1)



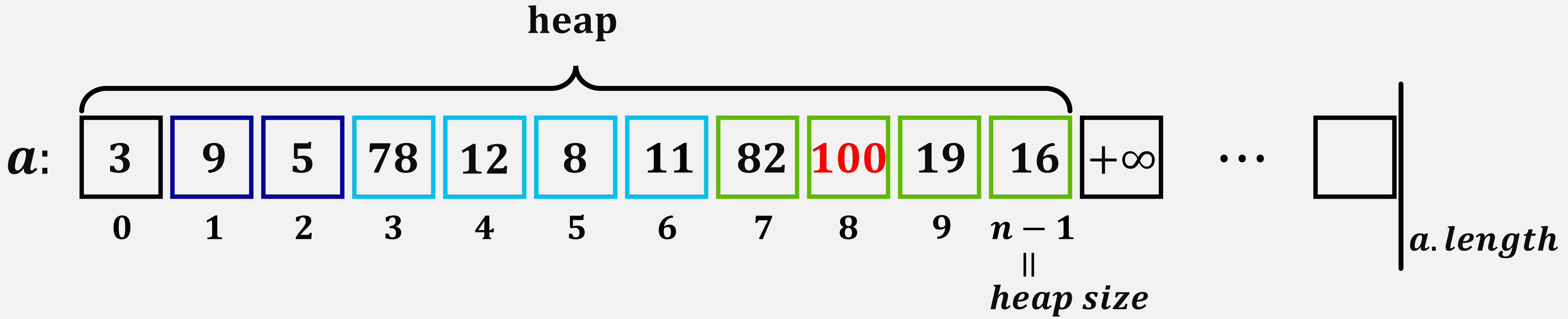


minHeapify(i):

find the smallest of $a[i]$, $a[\mathbf{Left}(i)]$ and $a[\mathbf{Right}(i)]$;
store its index in min;
if $min \neq i$ then
    swap $a[i]$ and $a[min]$;
    minHeapify(min);



minHeapify(1)



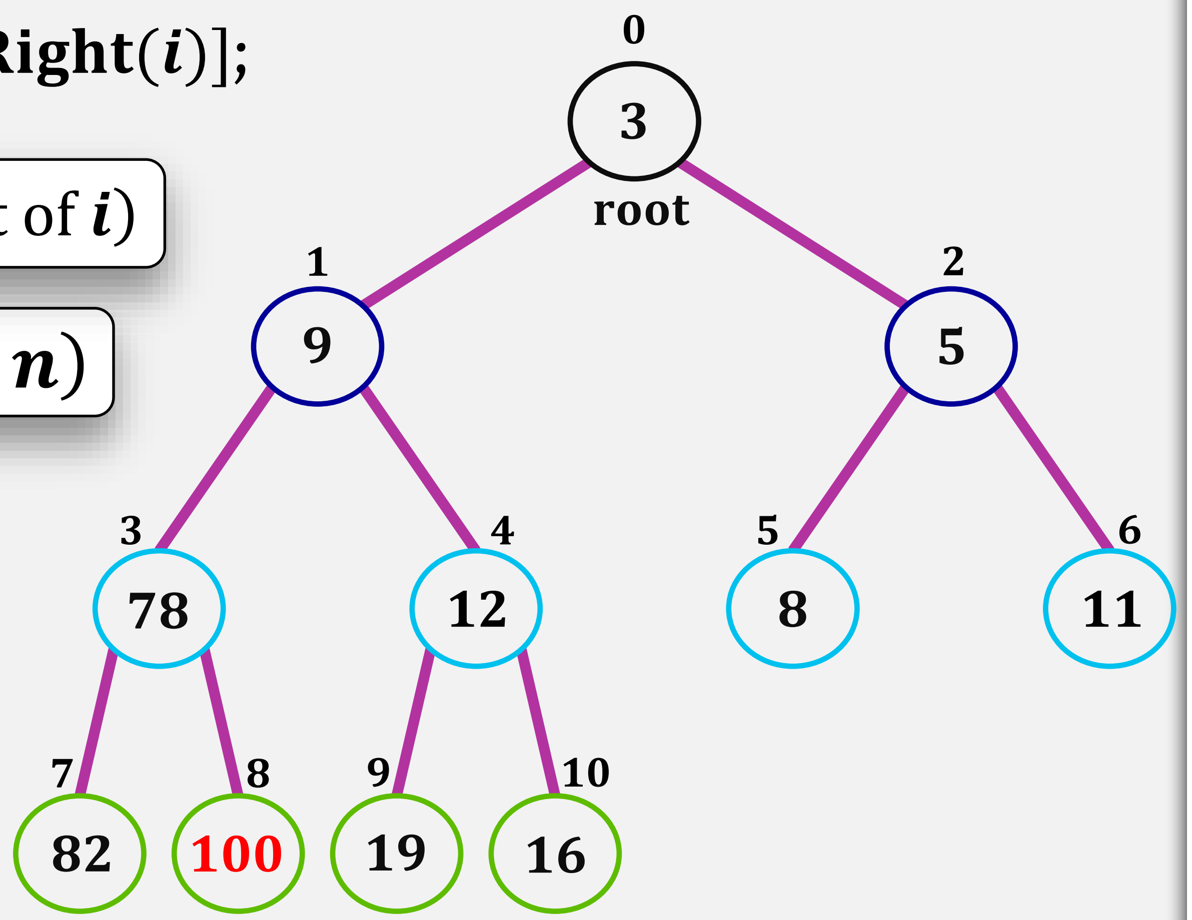
minHeapify(i):

find the smallest of $a[i]$, $a[\mathbf{Left}(i)]$ and $a[\mathbf{Right}(i)]$;
store its index in min;
if $min \neq i$ then
swap $a[i]$ and $a[min]$;
minHeapify(min);

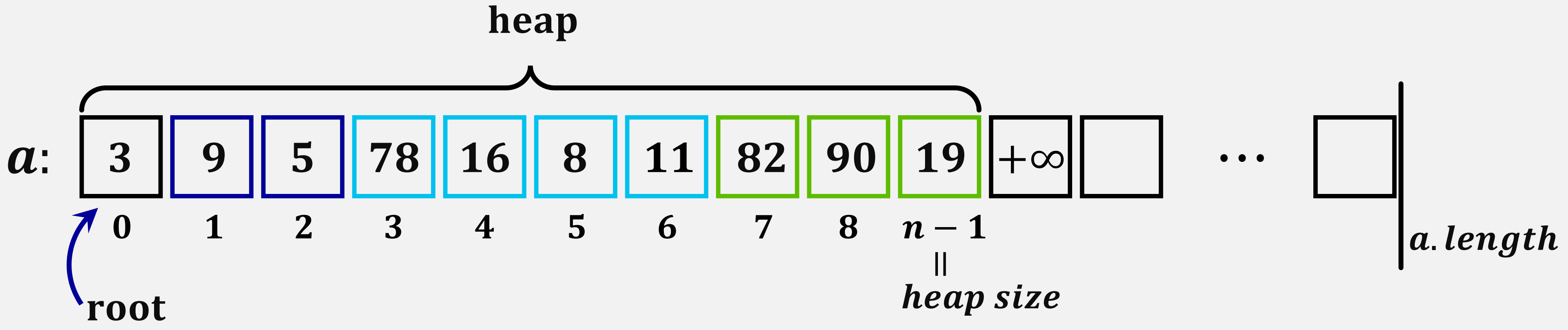
$\boxed{O(\text{height of } i)}$

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

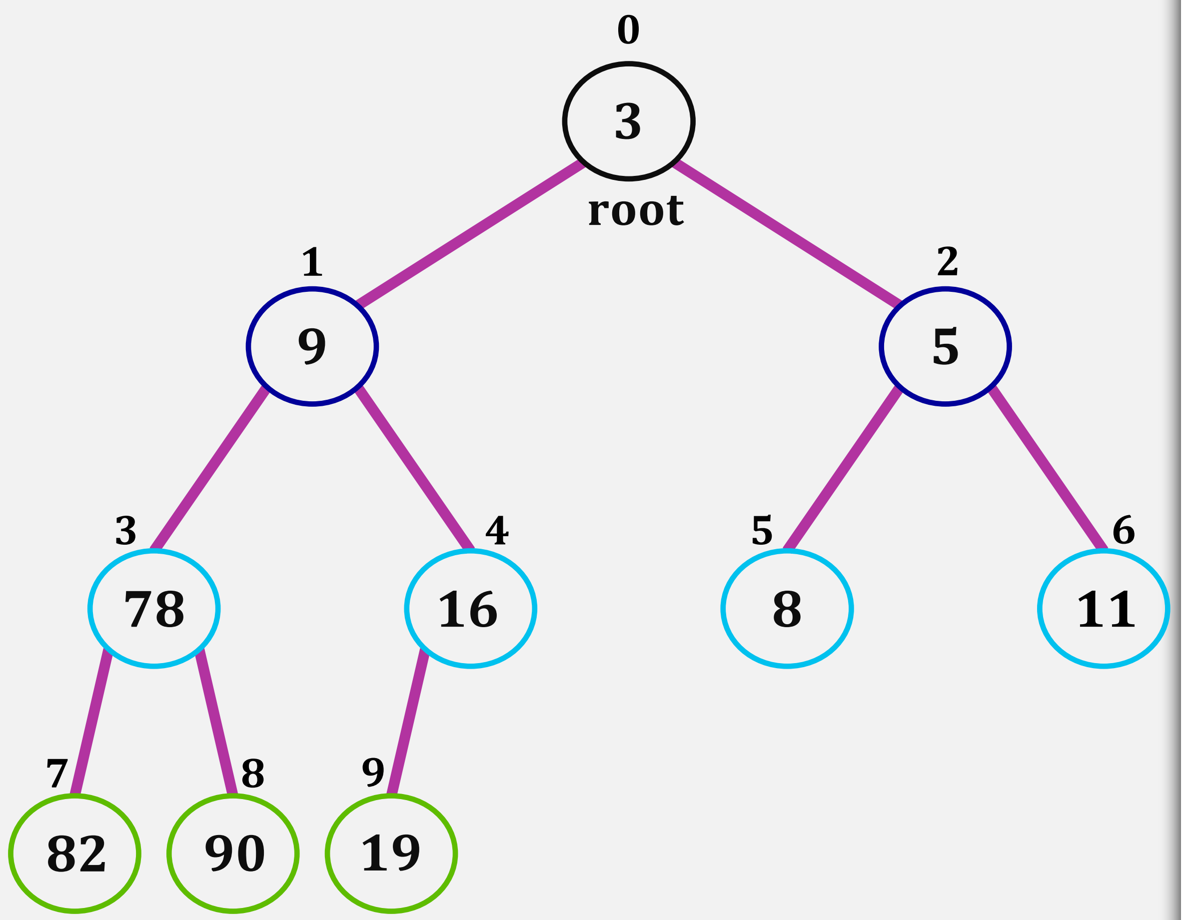
minHeapify(1)



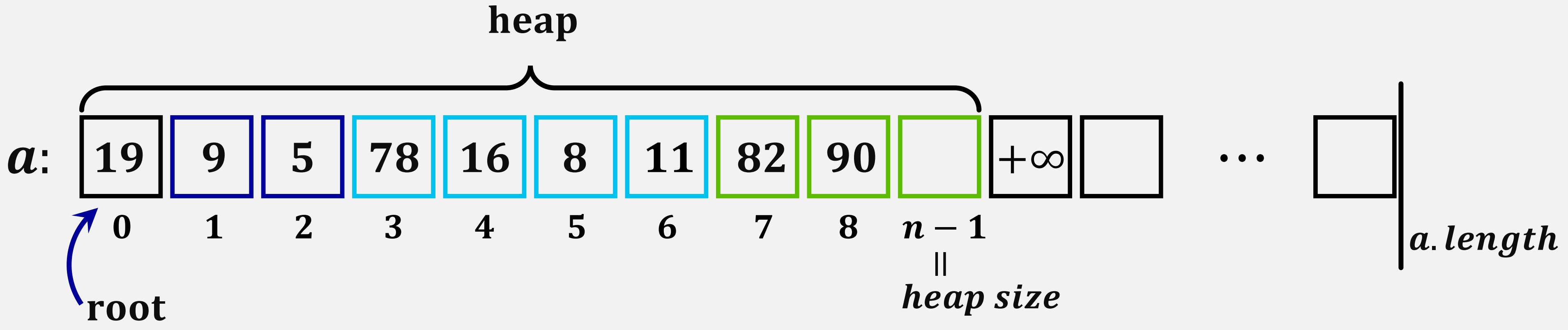
Alina Shaikhet – COMP 2402 – Carleton University



removeMin()

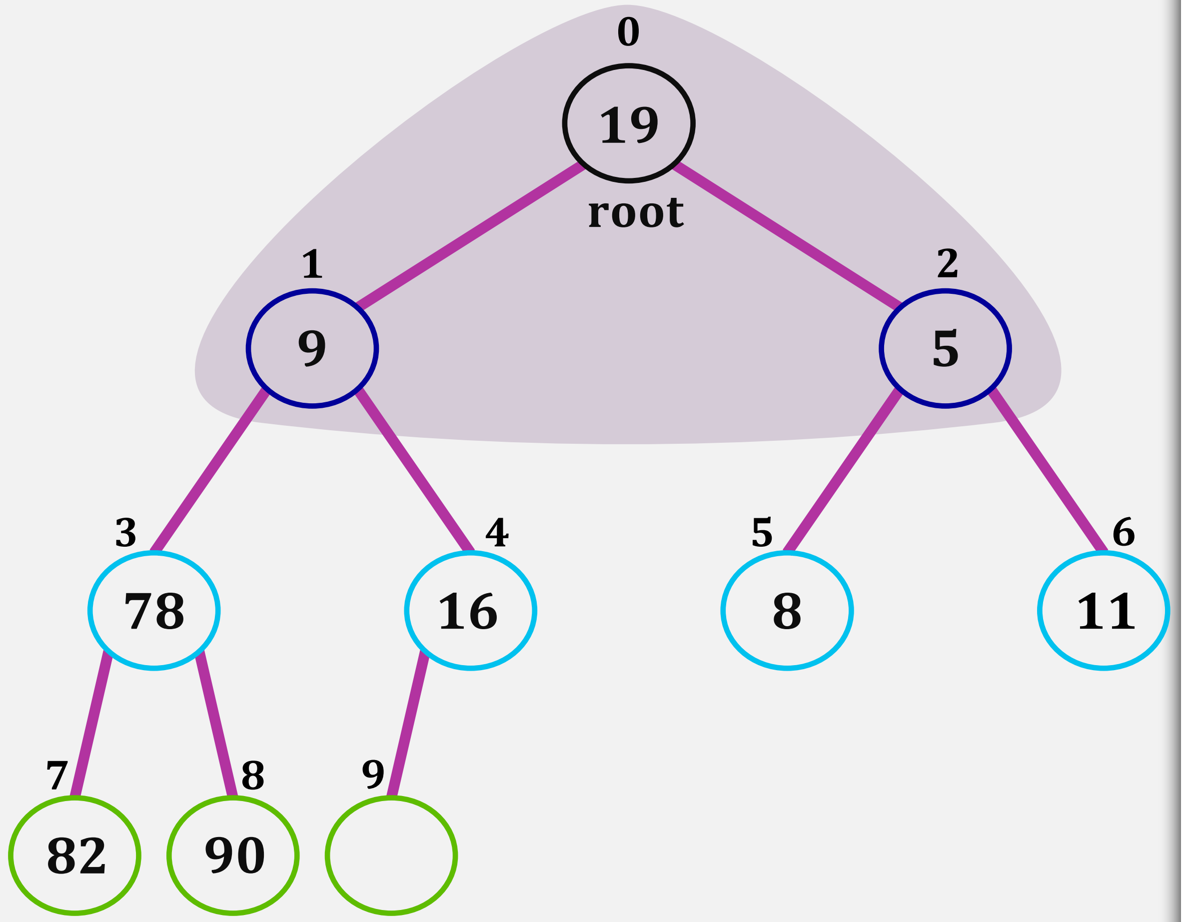


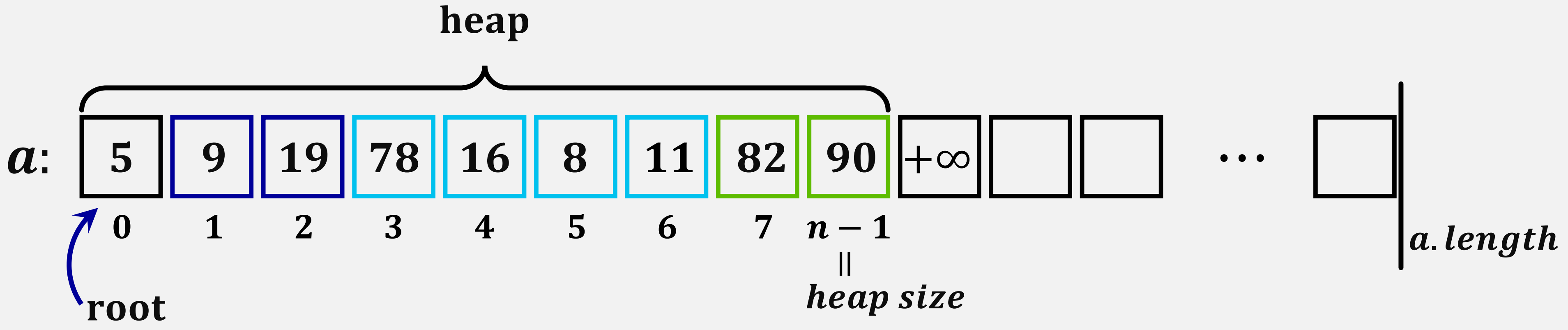
Alina Shaikhet – COMP 2402 – Carleton University

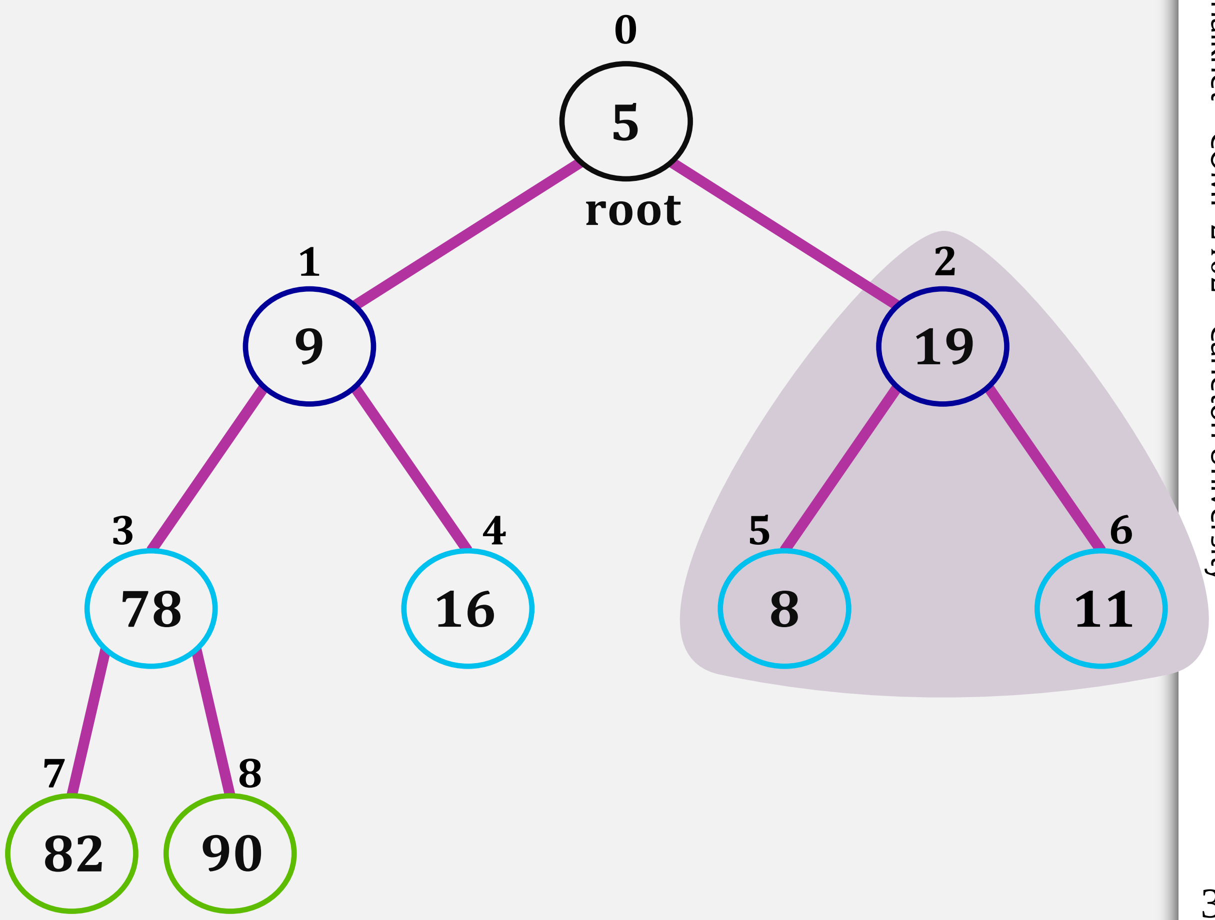


removeMin()

run **minHeapify(0);**

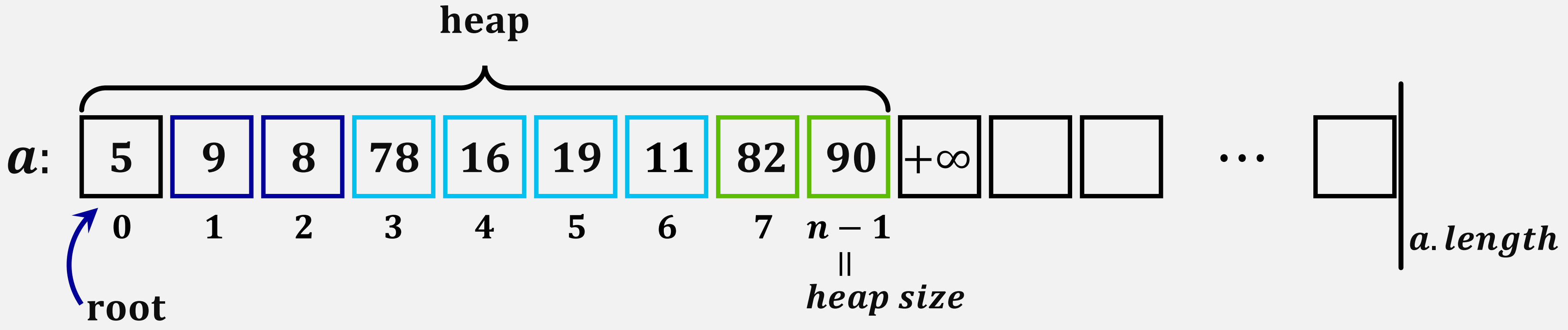






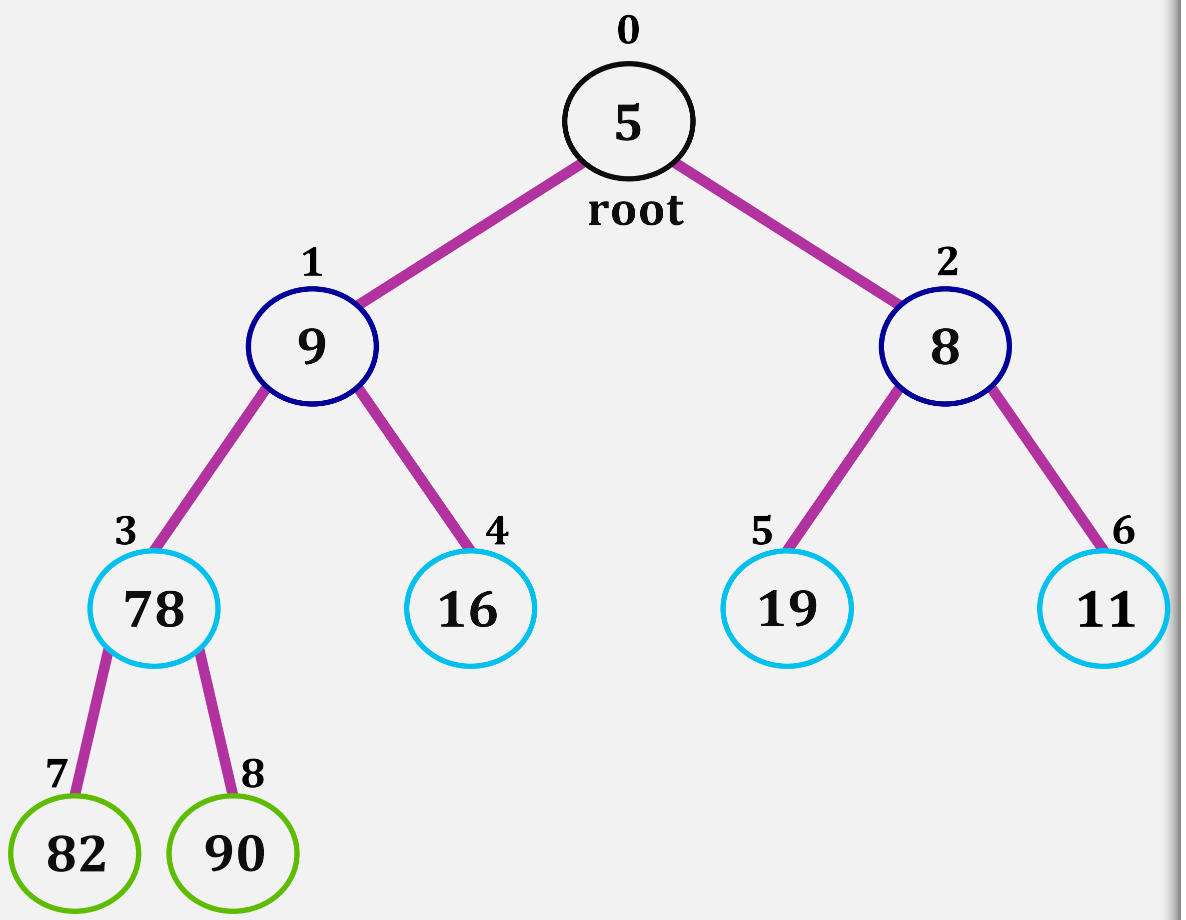
removeMin()

run **minHeapify(0);**



removeMin()

run **minHeapify(0);**



removeMin()

removeMin() – return and delete the smallest element in heap $a[0..n-1], n \geq 0$ and restore the heap property.

$\mathbf{min} = a[0];$
$a[0] = a[n-1];$
$n--;$
minHeapify$(0);$
if $(3n < a$.length) then resize();
return $\mathbf{min};$

$$O(1) + \underbrace{O(\log n)}_{\text{minHeapify}(0)} = O(\log n)$$

Theorem 10.1

A **BinaryHeap** implements the (priority) **Queue** interface. Ignoring the cost of calls to resize(), a **BinaryHeap** supports the operations add($\boldsymbol{x}$) and removeMin() in $\boldsymbol{O}(\log \boldsymbol{n})$ time per operation, and findMin() in $\boldsymbol{O}(\mathbf{1})$ time per operation.
Furthermore, beginning with an empty **BinaryHeap**, any sequence of $\boldsymbol{m}$ add($\boldsymbol{x}$) and removeMin() operations results in a total of $\boldsymbol{O}(\boldsymbol{m})$ time spent during all calls to resize().