Lecture 6.
Knuth Morris Pratt algorithm
Binary heap

Algorithms and Data Structures
Ivan Solomatin
MIPT 2020

Outline

- Knuth Morris Pratt (KMP) algorithm
  - String-searching problem
  - Prefix-function
  - KMP algorithm
- Binary heap
  - Heap invariant
  - Restoration of invariant (if element updated)
  - Implementation on vector
  - Push/pop/remove
  - Building heap
  - Implementation

Knuth Morris Pratt algorithm

- **String-searching problem**
- Prefix-function
- KMP algorithm

String-searching problem

Problem statement

Given string s: $|s| = N$ and pattern p: $|p| = K$.

$$s = s_0 s_1 \dots s_{N-1}, \qquad s_i \in \Sigma$$

$$p = p_0 p_1 \dots p_{K-1}, \qquad p_i \in \Sigma$$

We need to find all substrings of s, which are equal to p

$$i: \begin{cases} s_i = p_0 \\ s_{i+1} = p_1 \\ \dots \\ s_{i+K-1} = p_{K-1} \end{cases} \Leftrightarrow \ \forall j \in [0, K): s_{i+j} = p_j$$

String-searching problem

Problem statement

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c

	0	1	2
p:	a	b	a

Substrings:

0, 2, 6

String-searching problem

Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c

p:	a	b	a

String-searching problem

Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c
p:		a	b	a							

String-searching problem

Naïve solution

```python
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c
p:			a	b	a						

String-searching problem

Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c
p:				a	b	a					

String-searching problem

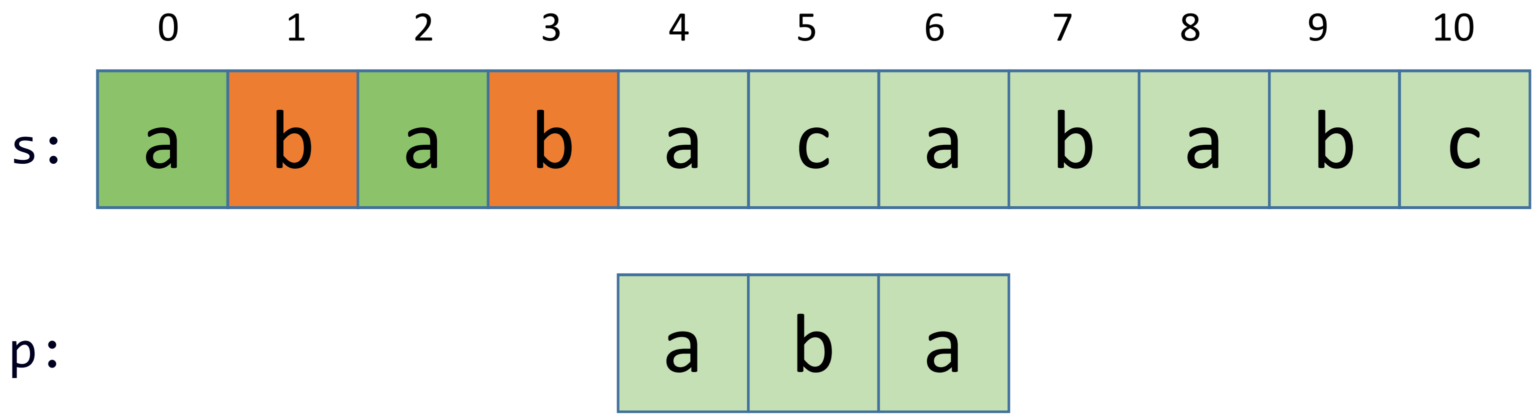
Naïve solution

```python
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2



String-searching problem

Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c
p:						a	b	a			

String-searching problem

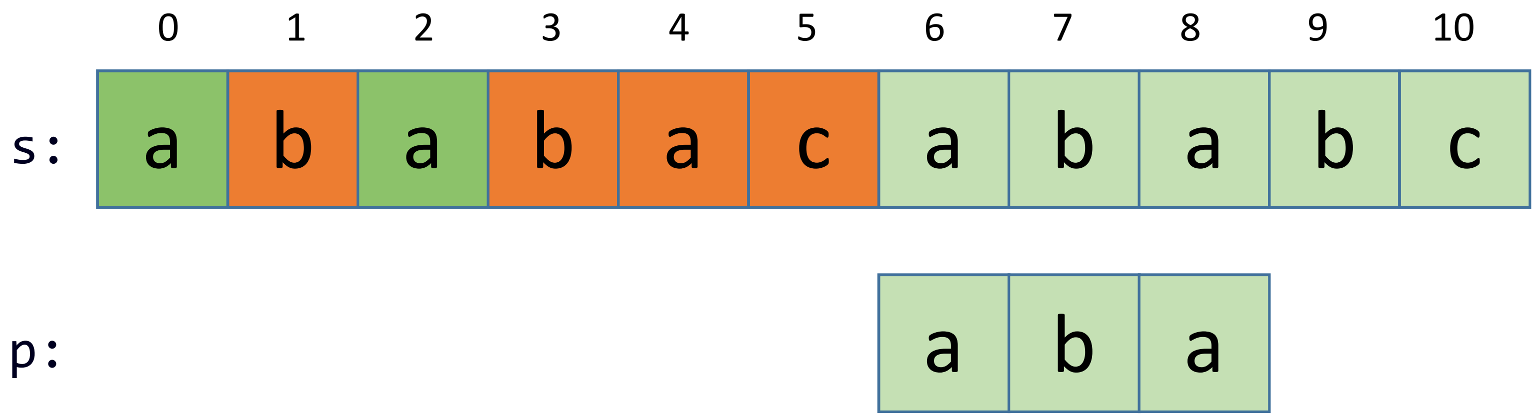
Naïve solution

```python
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2, 6



String-searching problem

Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2, 6

	0	1	2	3	4	5	6	7	8	9	10
s:	a	b	a	b	a	c	a	b	a	b	c
p:								a	b	a	

String-searching problem

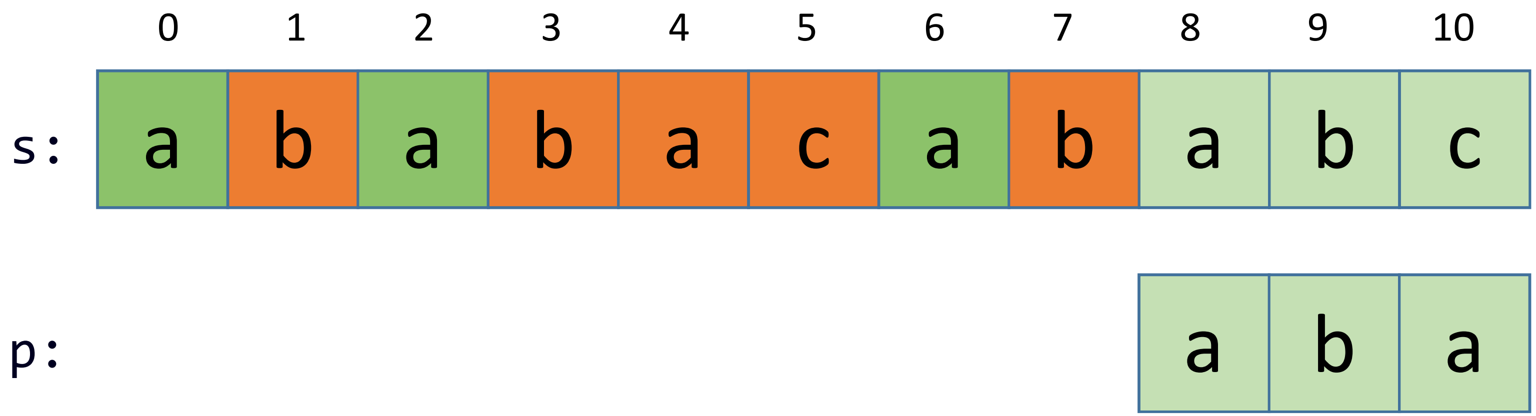
Naïve solution

```
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2, 6



String-searching problem

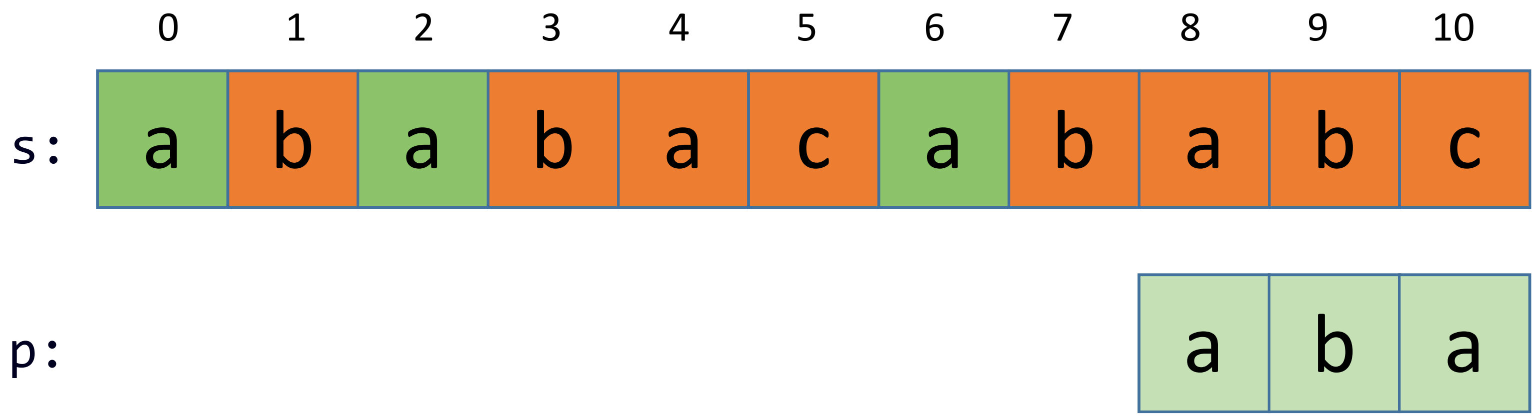
Naïve solution

```python
N = len(s)
K = len(p)
substrings = []
for i in range(N - K + 1):
    if all([s[i + j] == p[j] for j in range(K)]):
        substrings.append(i)
```

Substrings:

$$O(NK)$$

0, 2, 6



Knuth Morris Pratt algorithm

- String-searching problem
- **Prefix-function**
- KMP algorithm

Prefix-function

Definition

$\pi(s)$ – length of longest prefix which equals suffix, which is not a whole string

$$\pi(s) =$$
$$\max\{i:\ i < N\ \cap\ s[:i] = s[-i:]\} =$$
$$\max\{i:\ i < N\ \cap\ \forall j \in [0;i)\ s[j] = s[N-1-i+j]\}$$

$\pi(\text{‘abacaba’}) = 3$

$\pi(\text{‘aaaaaaa’}) = 6$

$\pi(\text{‘abcdefg’}) = 0$

$\pi(\text{‘abcabcabc’}) = 6$

Prefix-function

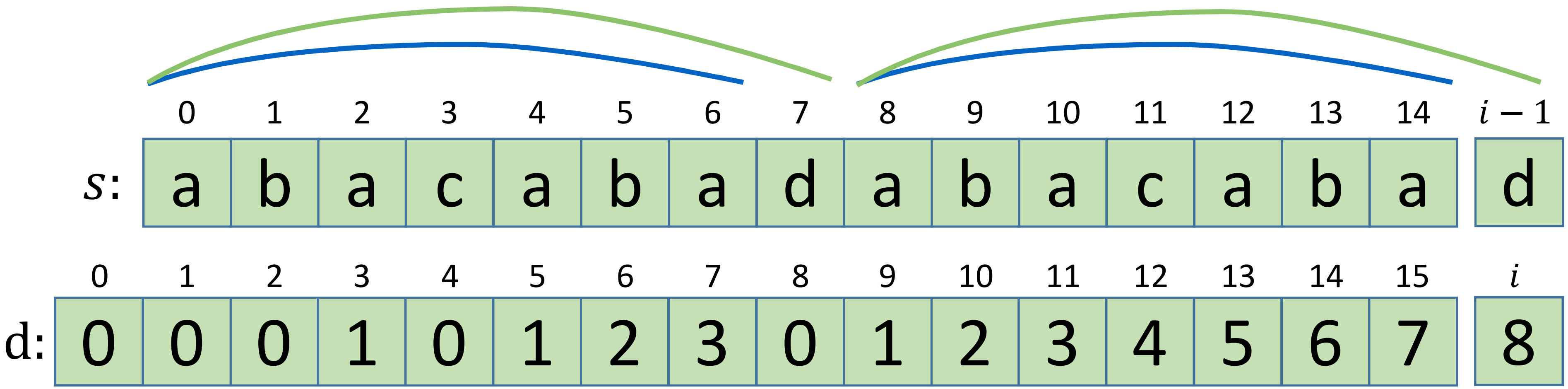
DP approach for π calculation

Let's calculate π function using DP approach.

1. $\mathrm{d[i]} = \pi(s[:i])$
2. $\mathrm{d[0]} = \pi(s[:0]) = \pi('') = 0$
   $\mathrm{d[1]} = \pi(s[:1]) = \pi(\mathrm{s}_0) = 0$
3. $\mathrm{d[i]} = ?$

Prefix-function

DP approach for π calculation



$$s[i-1] = s\big[d[i-1]\big]$$

$$d[i] = d[i-1] + 1$$

Prefix-function

DP approach for π calculation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	$i-1$
s:	a	b	a	c	a	b	a	d	a	b	a	c	a	b	a	c

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	i
d:	0	0	0	1	0	1	2	3	0	1	2	3	4	5	6	7	4

$$s[i-1] \neq s\big[d[i-1]\big]$$

$$s[i-1] = s\Big[d\big[d[i-1]\big]\Big]$$

$$d[i] = d\big[d[i-1]\big] + 1$$

Prefix-function

DP approach for π calculation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	$i-1$
s:	a	b	a	c	a	b	a	d	a	b	a	c	a	b	a	?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	i
d:	0	0	0	1	0	1	2	3	0	1	2	3	4	5	6	7	?

$$s[i-1] \neq s\left[d[i-1]\right]$$

$$s[i-1] \neq s\left[d\left[d[i-1]\right]\right]$$

$$s[i-1] \neq s\left[d\left[d\left[d[i-1]\right]\right]\right]$$

...

$$d[i] = \begin{cases} d[\dots] + 1 \\ 0 \end{cases}$$

Prefix-function

DP approach for π calculation

Let's calculate π function using DP approach.

1. $\mathrm{d[i]} = \pi(s[:i])$
2. $\mathrm{d[0]} = \pi(s[:0]) = \pi('') = 0$

   $\mathrm{d[1]} = \pi(s[:1]) = \pi(\mathrm{s}_0) = 0$
3.
```
d[i] = d[i - 1]
while s[i - 1] != s[d[i]] and d[i] > 0:
    d[i] = d[d[i]]
if s[i - 1] == s[d[i]]:
    d[i] += 1
```
4. ?

Knuth Morris Pratt algorithm

- String-searching problem
- Prefix-function
- **KMP algorithm**

KMP algorithm

Idea

We have a string s and pattern p, both with elements from Σ. Let's use DP approach to calculate $\pi(\mathrm{p\$s})$, where $\$ \notin \Sigma$ – any symbol not from Σ.

	0	1	2	3	4	5	6	7	8	9	10	11	12
s:	a	b	a	$	a	b	a	c	a	b	a	b	a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
d:	0	0	0	1	0	1	2	3	0	1	2	3	2	3

If in any index i : $d[i] = |p|$, this means, that i is end of matched substring.

KMP algorithm

Implementation

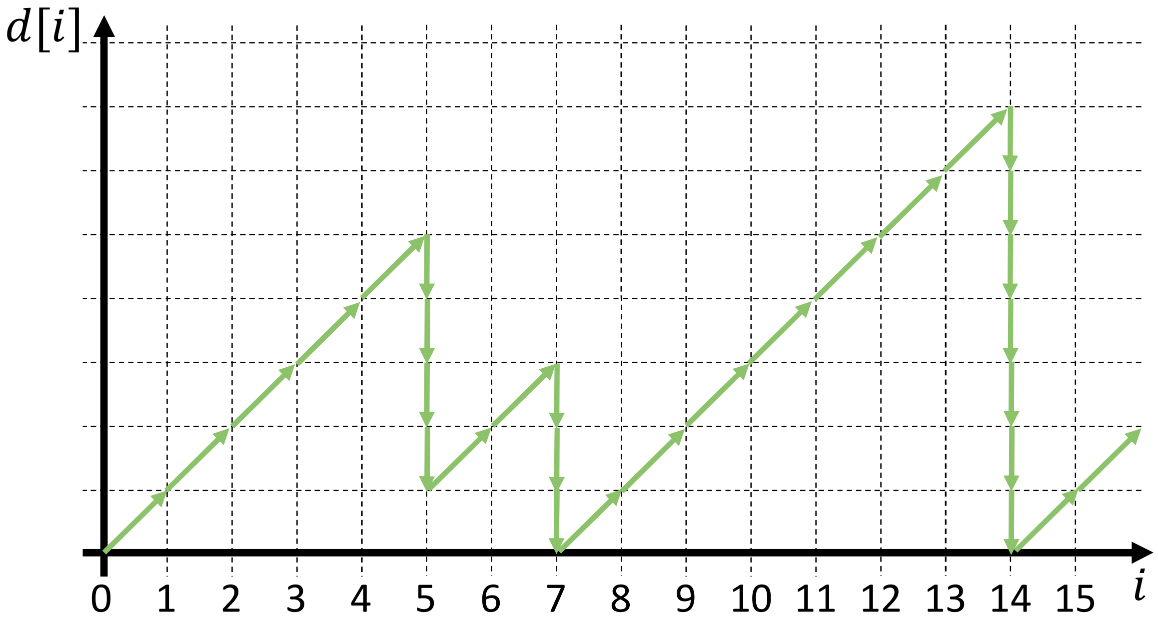
```python
def prefix_function(s):
    d = [0] * (len(s) + 1)
    for i in range(2, len(d)):
        d[i] = d[i - 1]
        while s[i - 1] != s[d[i]] and d[i] > 0:
            d[i] = d[d[i]]
        if s[i - 1] == s[d[i]]:
            d[i] += 1
    return d

def find_substrings(s, p):
    substrings = []
    d = prefix_function(p + '$' + s)
    for i in range(len(p) + 1, len(d)):
        if d[i] == len(p):
            substrings.append(i - 2 * len(p) - 1)
    return substrings
```

KMP algorithm
Complexity

Let’s analyze complexity of `prefix_function(s)`.

On each step we either assign d[i] = d[i-1]+1, or iterate over candidates d[i-1], d[d[i-1]], …, until we find suitable one, or reach 0. In worst case, each such iteration decrease d[i] by 1, and it may take $O(N)$ operations. But let’s calculate total number of operations:



N_+ – number of +1 operations
N_- – number of –1 operations
$N_+ \leq N$
$N_- \leq N_+ \leq N$

Complexity:
$T = N_+ + N_- \leq 2N = O(N)$

Conclusion

Python built-ins

```python
s = 'abacababa'
p = 'aba'
# first substring:
print(s.find(p))


# all substrings
# (may be a bit slower than KMP if lots of matches):
i = -1
substrings = []
while True:
    i = s.find(p, i + 1)
    if i >= 0:
        substrings.append(i)
    else:
        break

# Best practice: regular expressions (re module)
```

Binary heap

- **Heap invariant**
- Restoration of invariant (if element updated)
- Implementation on vector
- Push/pop/remove
- Building heap
- Implementation

Binary heap

Heap invariant

Imagine that we need to obtain minimum of the elements present in data structure. Vector (python list), linked list and doubly linked list will use $O(N)$ operations for that. If we keep list sorted, we can minimum is 0-th element, so it'll take $O(1)$, but adding and removing elements will take $O(N)$

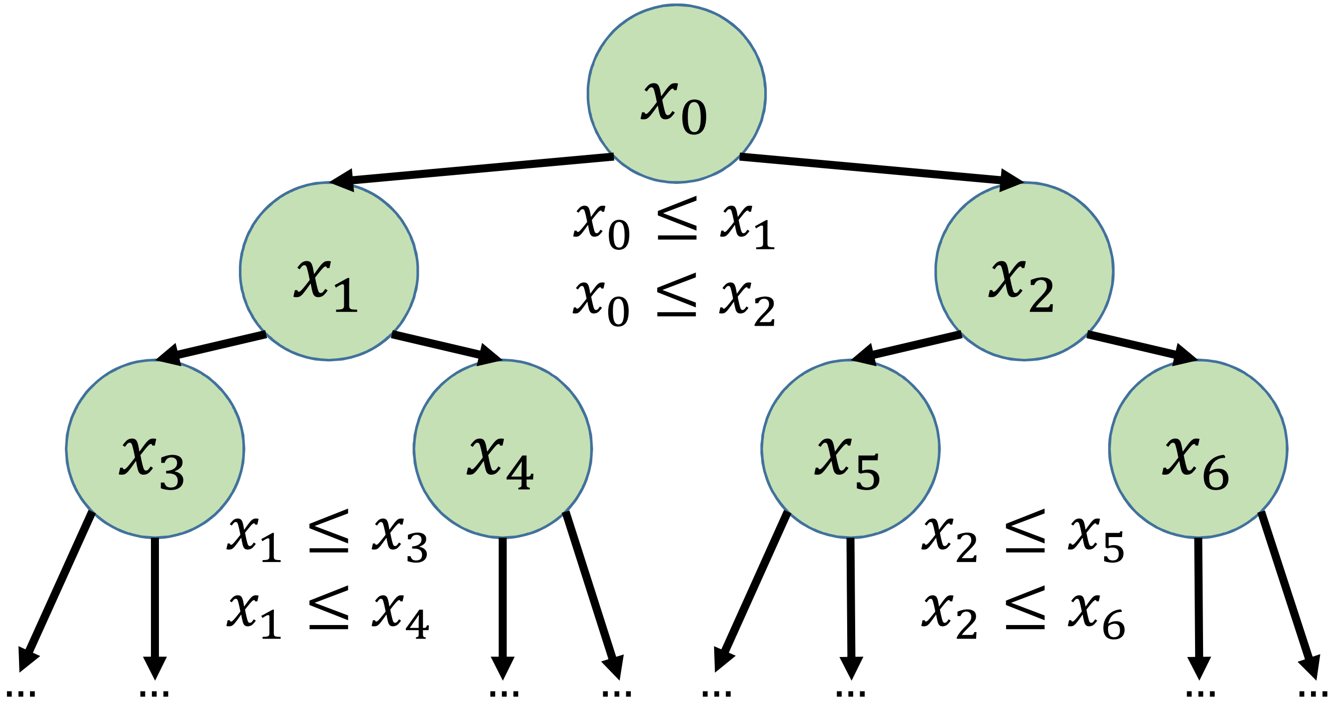
E.g. we need to support the following operations:

- `push(x)` – add value x to data structure (order doesn't matter).
- `remove(i)` – remove element, if you know its actual index (or node) i.
- `find_min()` – returns node with minimum value.
- `len()` – returns number of elements currently present in data structure.

Operation	Vector (python list)	Linked list	Doubly Linked list	Sorted vector (sorted python list)
`push(x)`	$O(1)$	$O(1)$	$O(1)$	$O(N)$
`remove(i)`	$O(N)$	$O(1)$	$O(1)$	$O(N)$
`find_min()`	$O(N)$	$O(N)$	$O(N)$	$O(1)$
`len()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Binary heap

Heap invariant



Value in parent node should be $\leq$ than values of his children.

$\Downarrow$

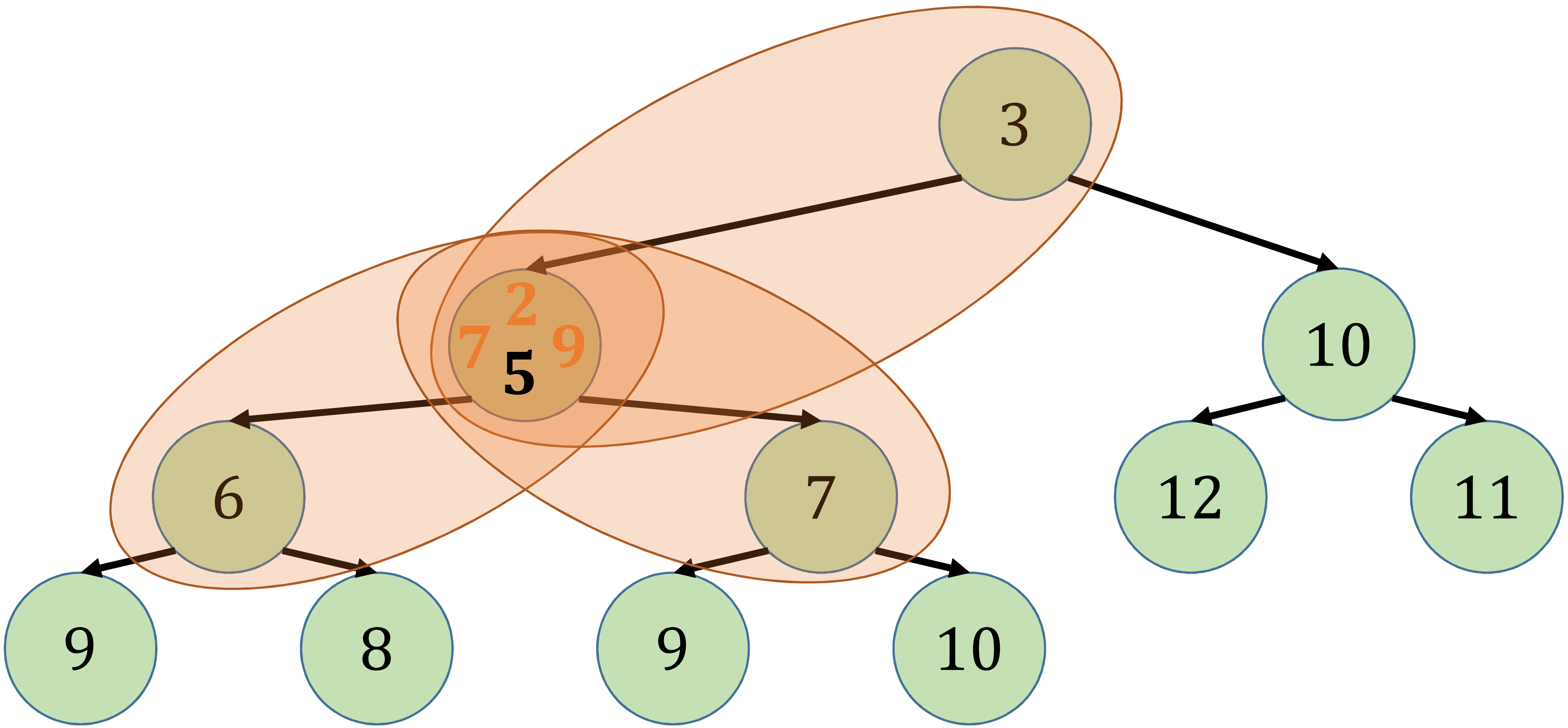
Root of the tree is minimum element.

Binary heap

- Heap invariant
- **Restoration of invariant (if element updated)**
- Implementation on vector
- Push/pop/remove
- Building heap
- Implementation

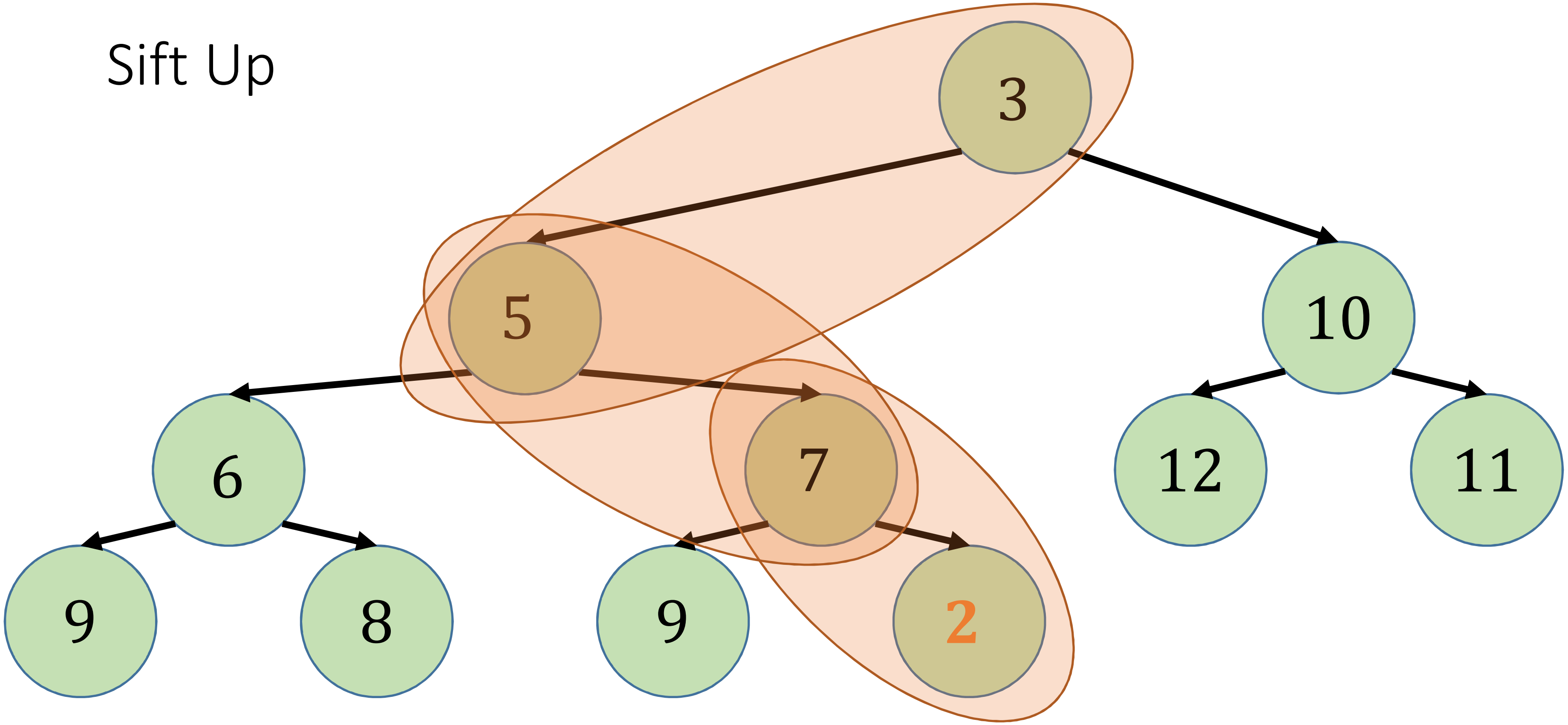
Restoration of invariant

Possible cases of invariant violation



Restoration of invariant

Sift Up



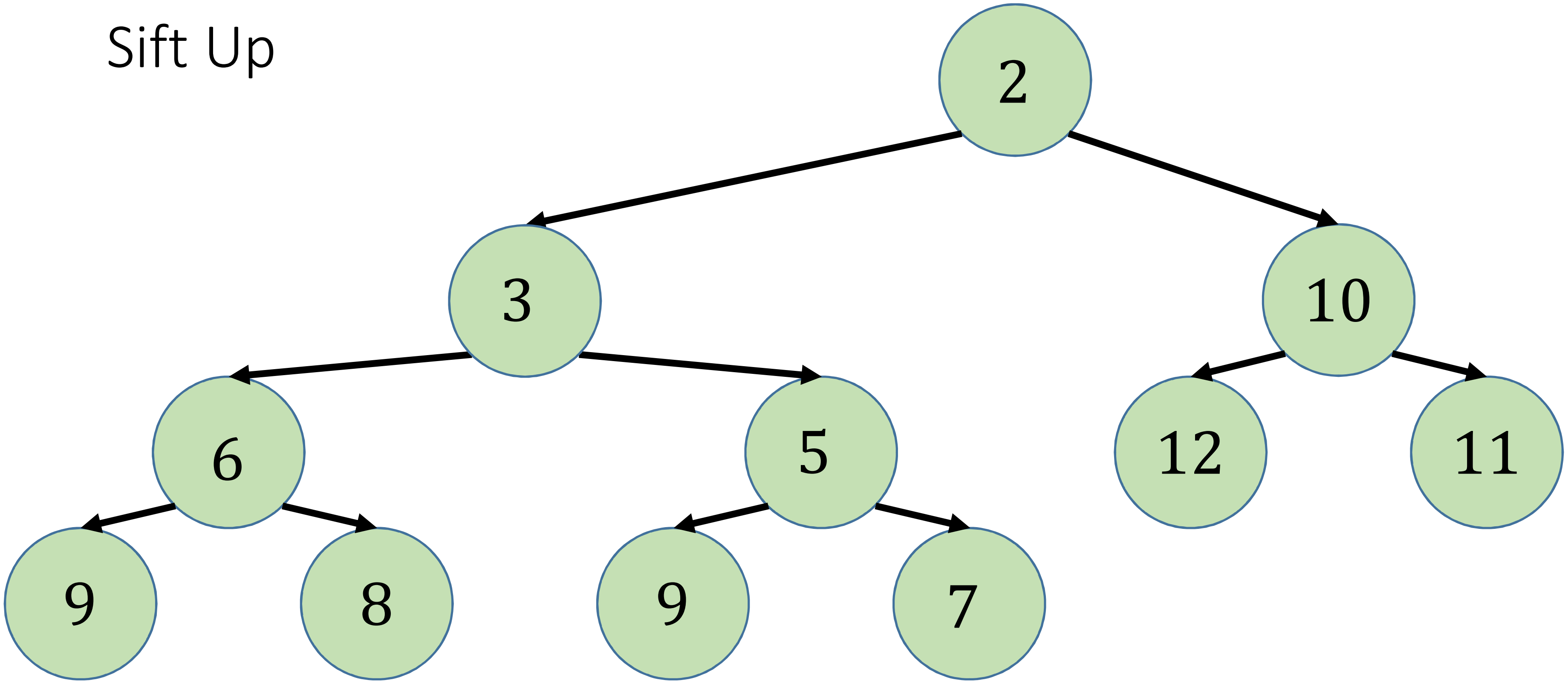
Pseudocode:

$O(H)$

```
def sift_up(data, i):
    if not is_root(i) and (data[i] < data[parent(i)]):
        swap(data[i], data[parent(i)])
        sift_up(data, parent(i))
```

Restoration of invariant

Sift Up



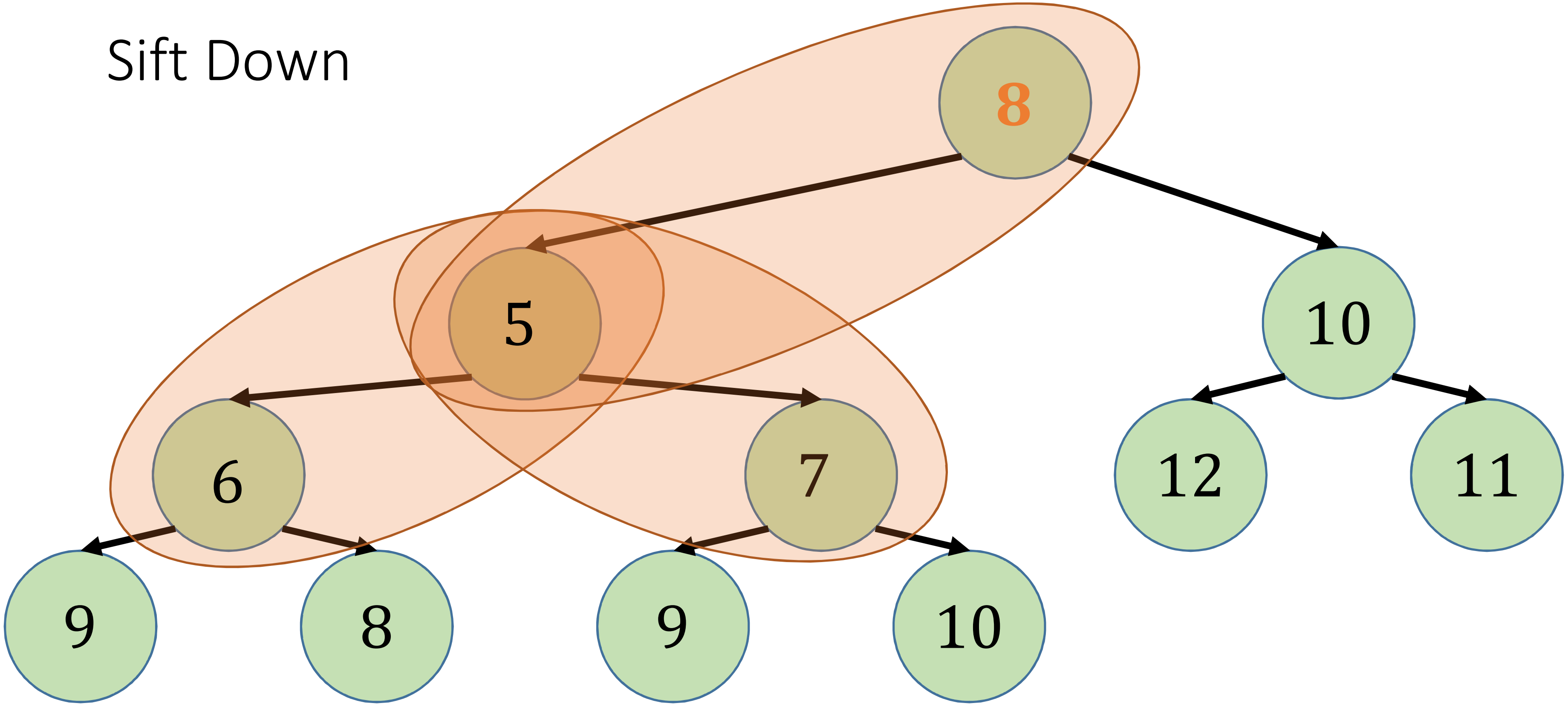
Pseudocode:

$O(H)$

```
def sift_up(data, i):
    if not is_root(i) and (data[i] < data[parent(i)]):
        swap(data[i], data[parent(i)])
        sift_up(data, parent(i))
```

Restoration of invariant

Sift Down



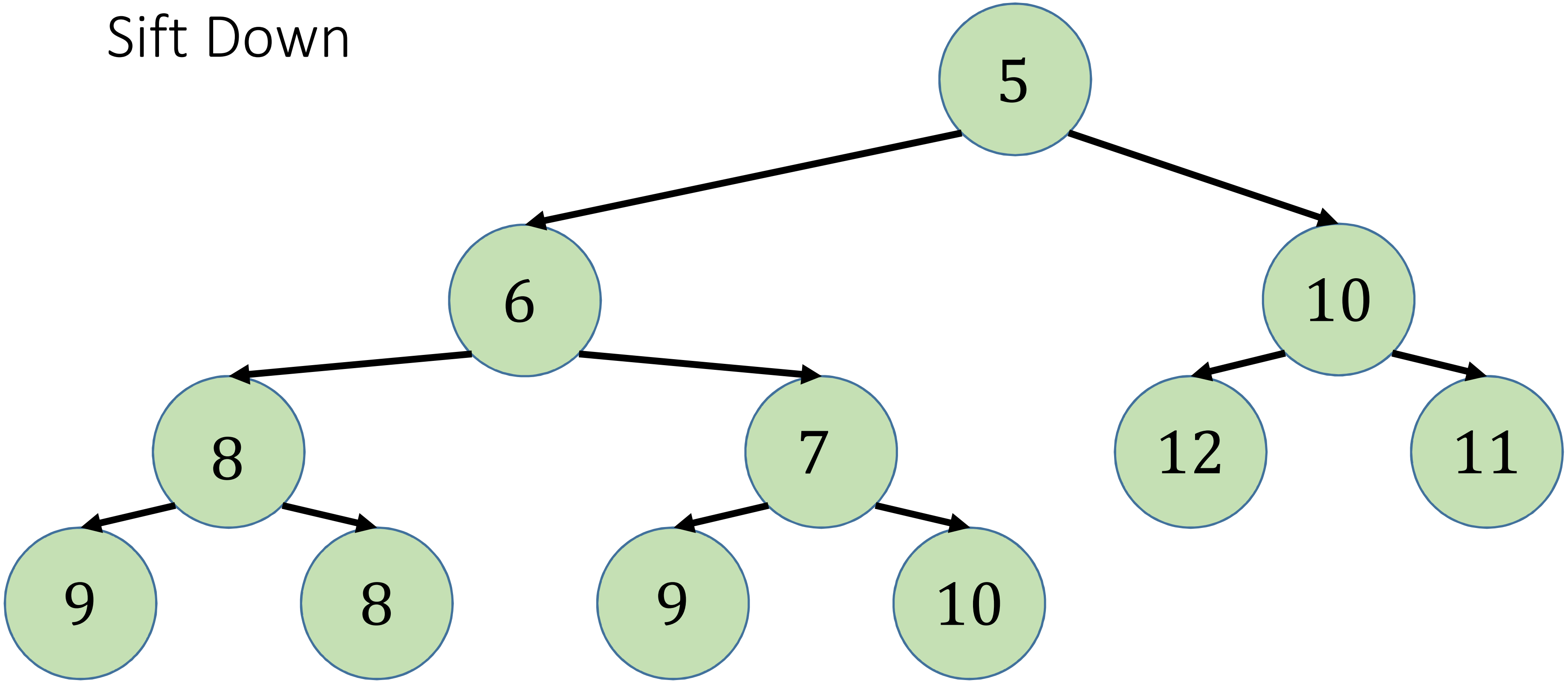
Pseudocode:

$O(H)$

```
def sift_down(data, i):
    i1, i2 = children(i)
    # node with minimum value (be careful, one or both children may not exist):
    i_min = i1 if data[i1] < data[i2] else i2
    if data[i_min] < data[i]:
        swap(data[i], data[i_min])
        sift_down(data, i_min)
```

Restoration of invariant

Sift Down



Pseudocode:

$O(H)$

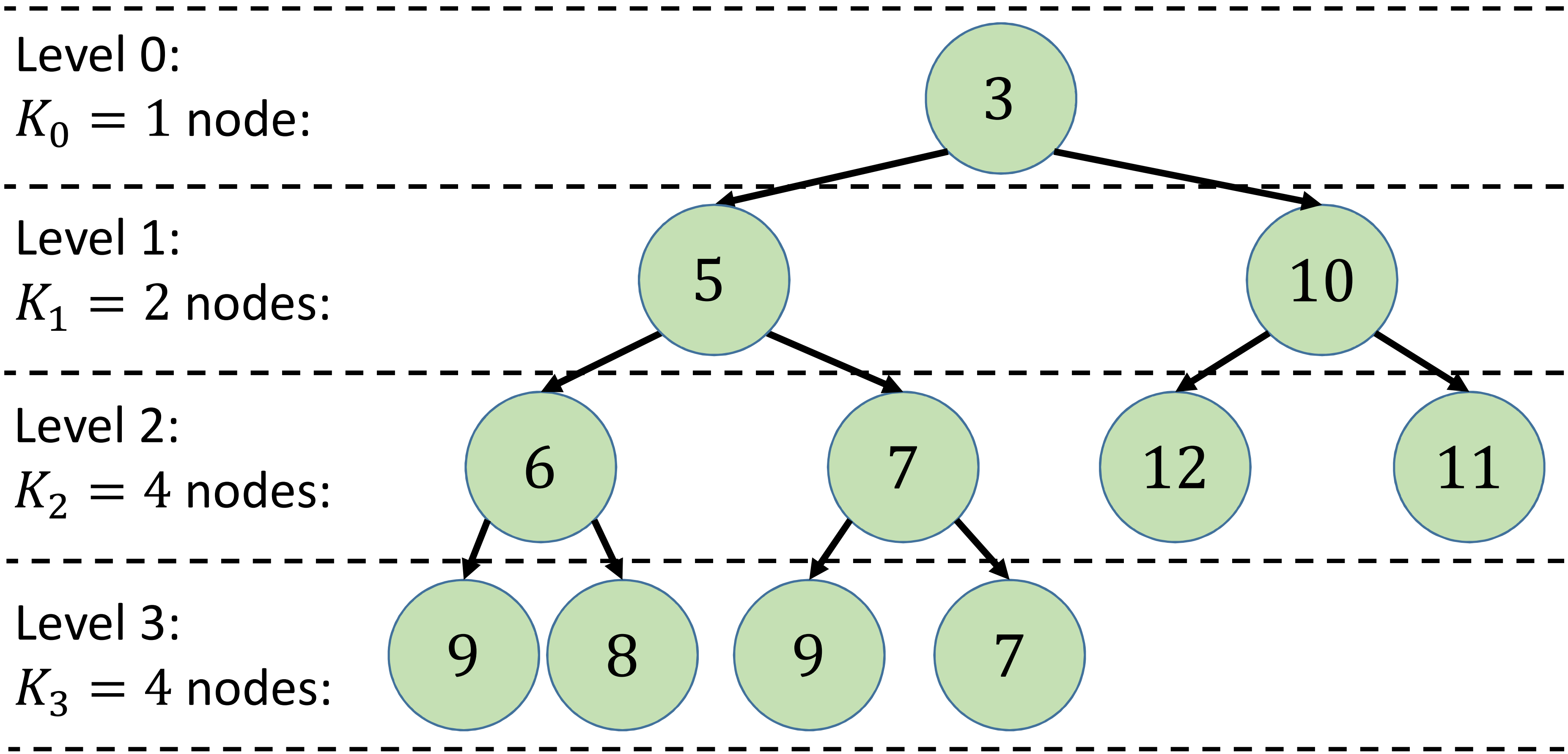
```
def sift_down(data, i):
    i1, i2 = children(i)
    # node with minimum value (be careful, one or both children may not exist):
    i_min = i1 if data[i1] < data[i2] else i2
    if data[i_min] < data[i]:
        swap(data[i], data[i_min])
        sift_down(data, i_min)
```

Binary heap

- Heap invariant
- Restoration of invariant (if element updated)
- **Implementation on vector**
- Push/pop/remove
- Building heap
- Implementation

Implementation on vector

Binary tree structure



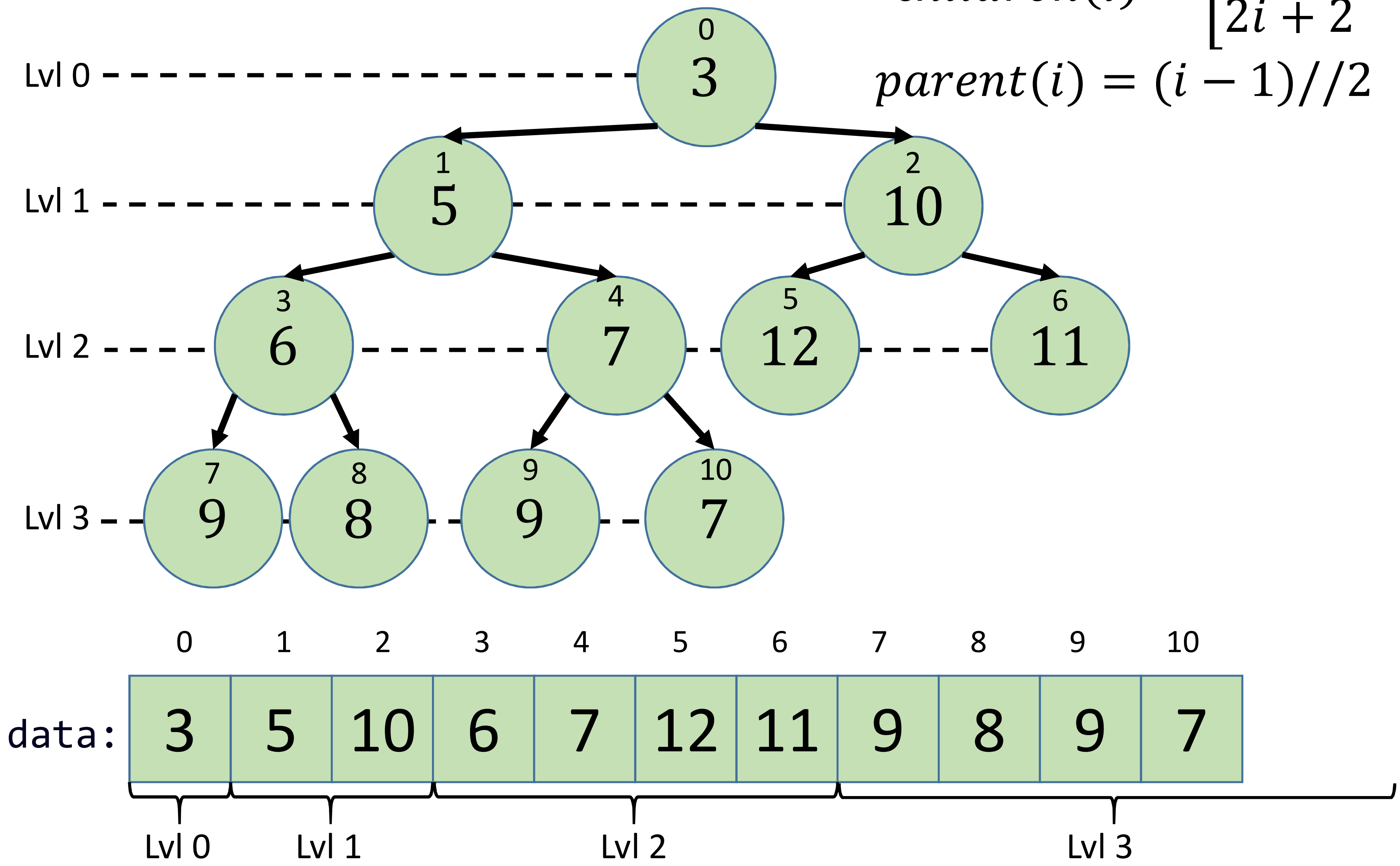
$H = \lfloor \log_2 N \rfloor$, $K_i = 2^i$ nodes, $i \in [0; H)$, $K_H \in (0, 2^H]$

Implementation on vector

Nodes enumeration

$$children(i) = \begin{bmatrix} 2i+1 \\ 2i+2 \end{bmatrix}$$

$$parent(i) = (i-1)//2$$

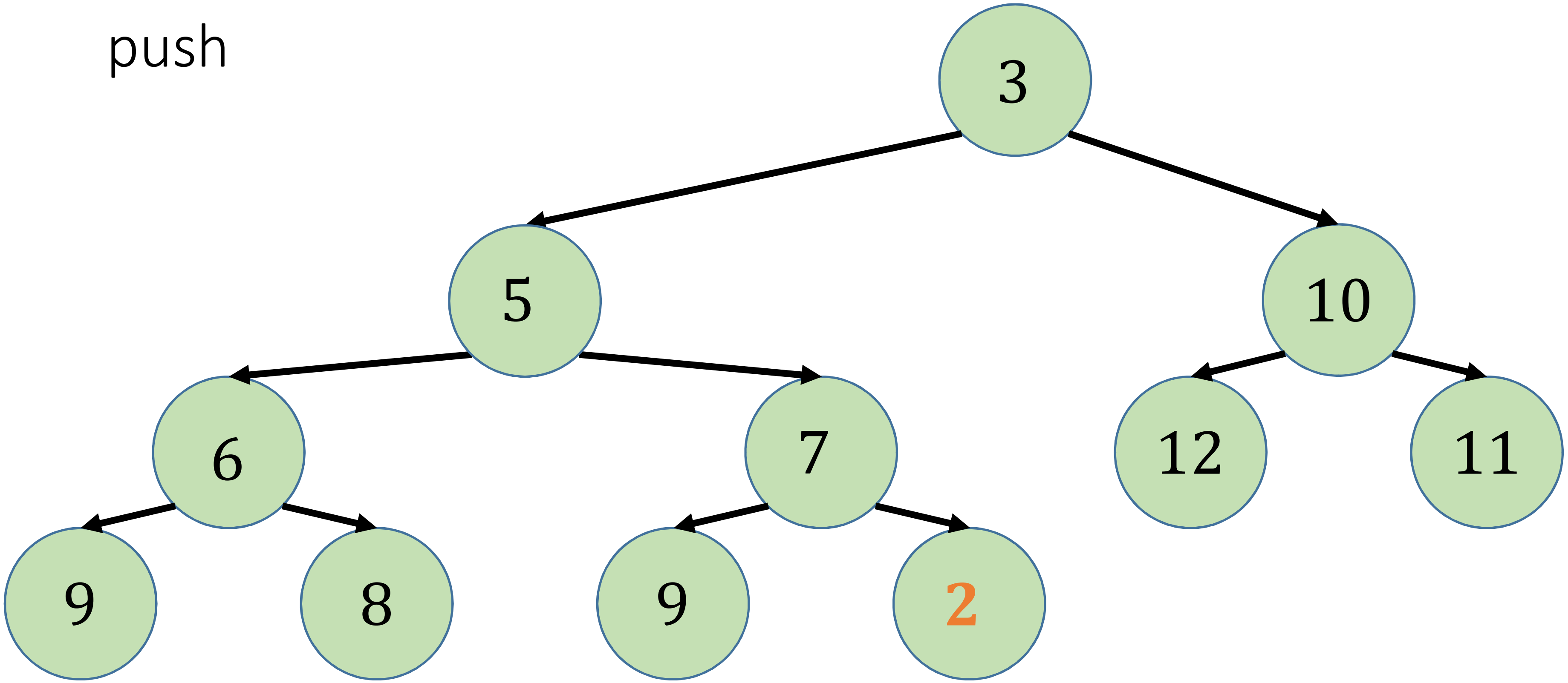


Binary heap

- Heap invariant
- Restoration of invariant (if element updated)
- Implementation on vector
- **Push/pop/remove**
- Building heap
- Implementation

Heap push/pop/remove

push

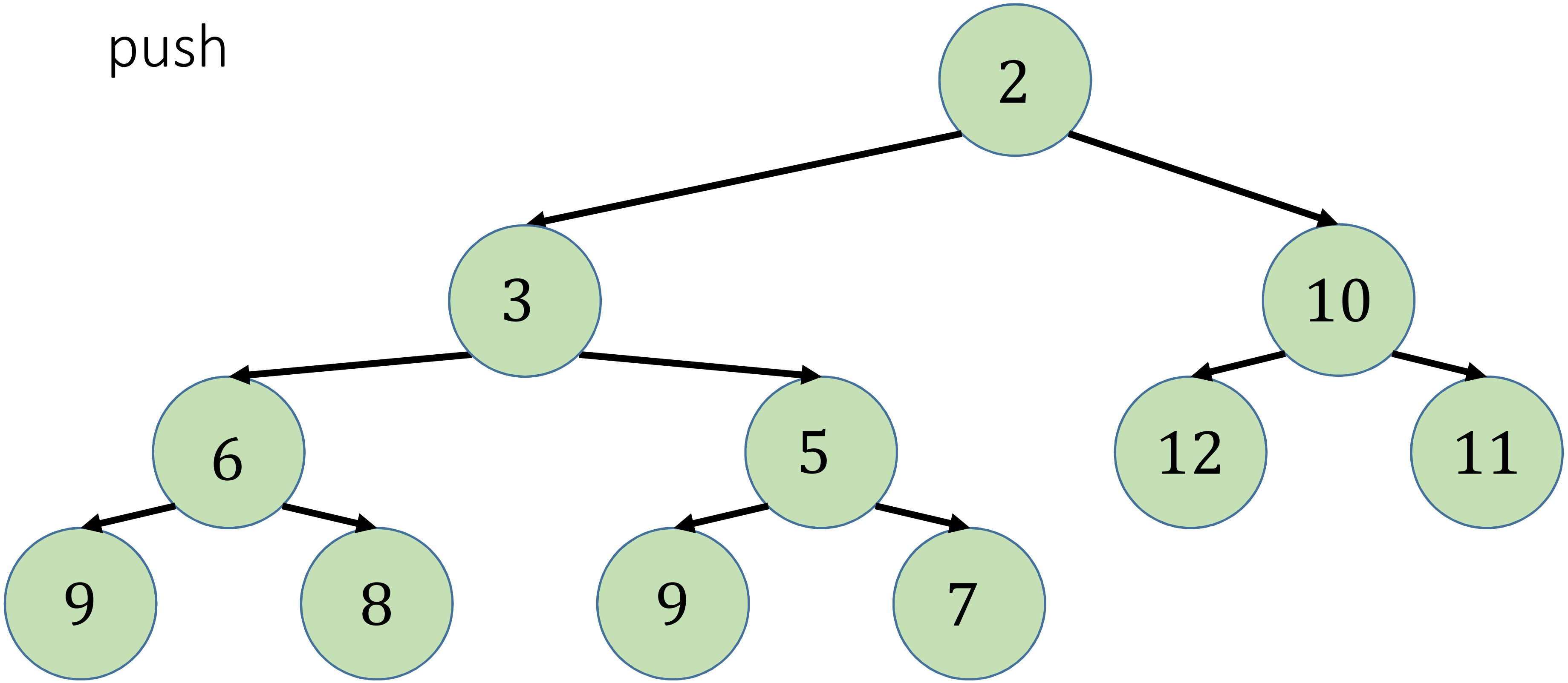


$O(H) = O(\log N)$

```python
def heappush(data, x):
    data.append(x)
    sift_up(data, len(data) - 1)
```

Heap push/pop/remove

push

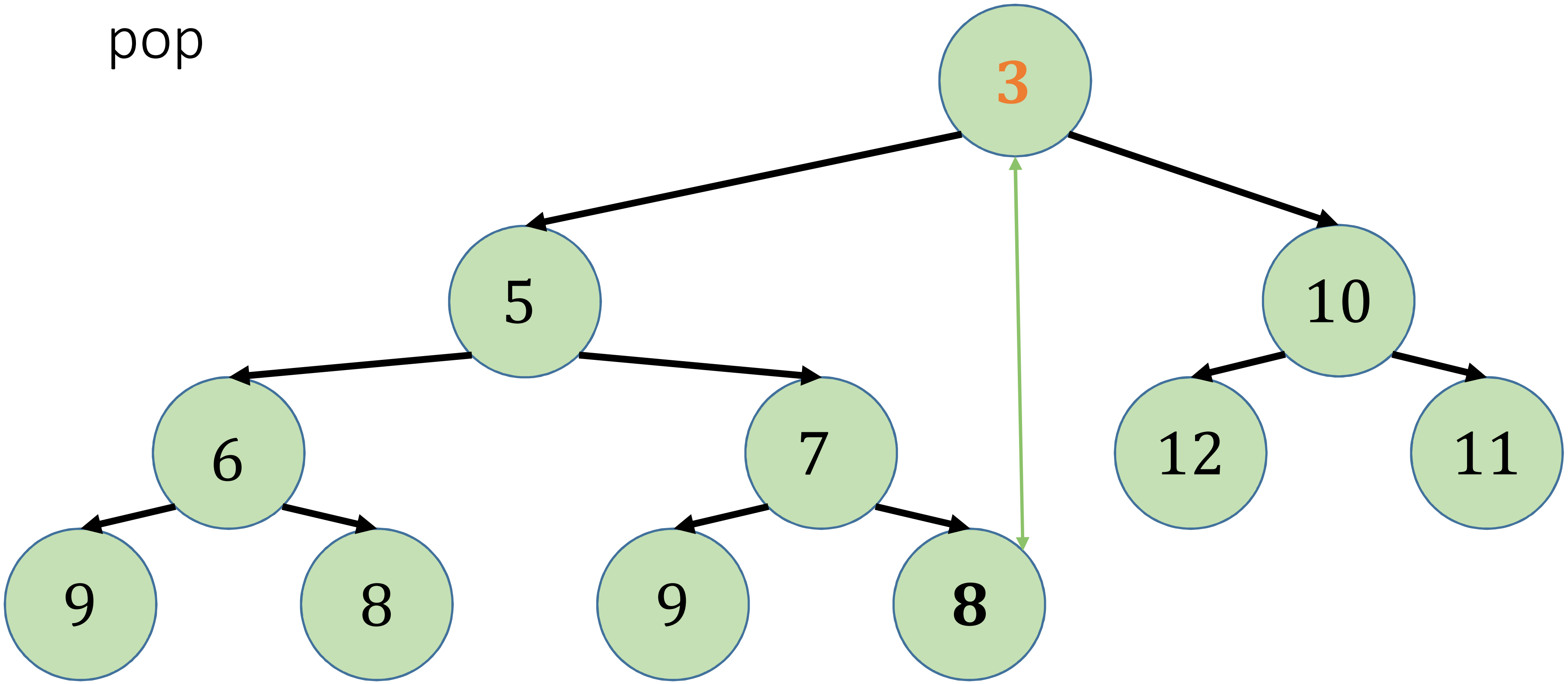


$$O(H) = O(\log N)$$

```
def heappush(data, x):
    data.append(x)
    sift_up(data, len(data) - 1)
```

Heap push/pop/remove

pop

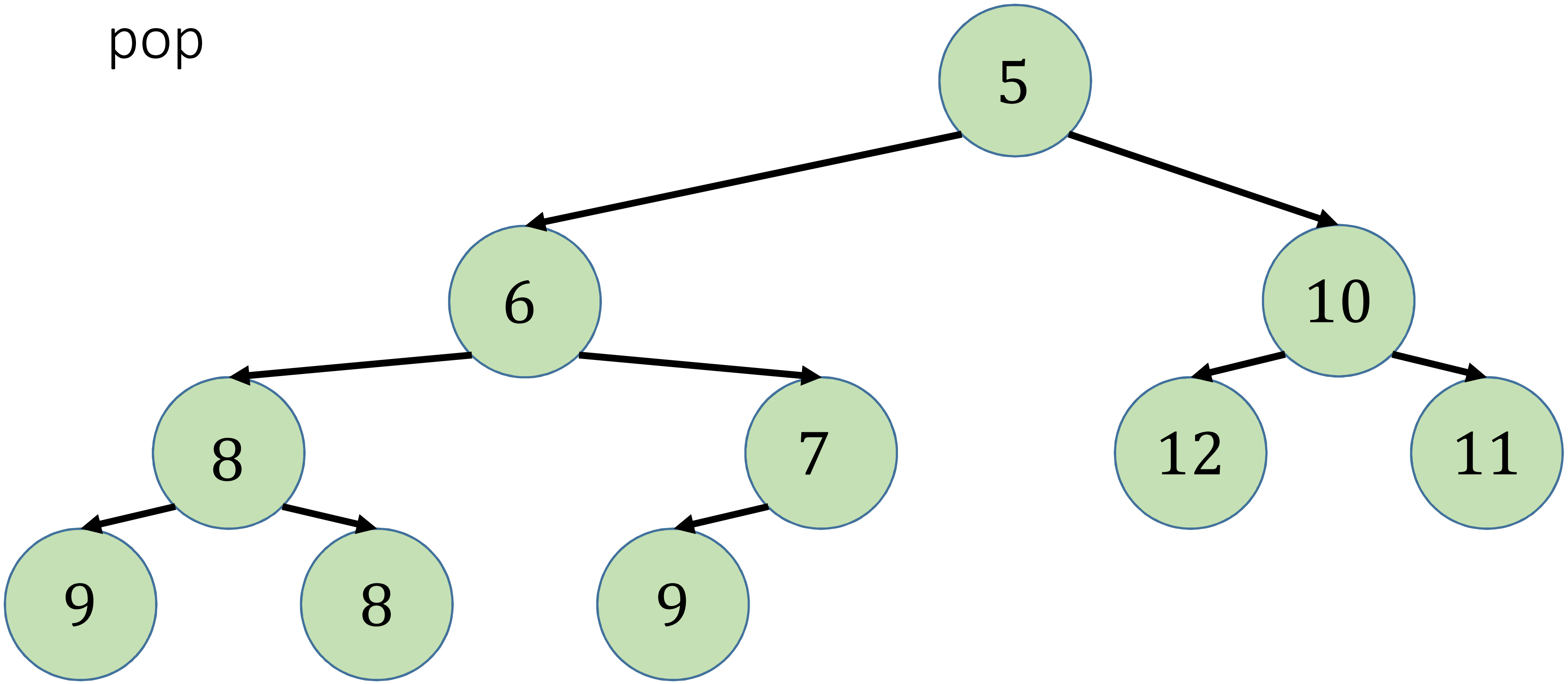


$$O(H) = O(\log N)$$

```
def heappop(data):
    swap(data[0], data[len(data) - 1])
    res = data.pop()
    sift_down(data, 0)
    return res
```

Heap push/pop/remove

pop

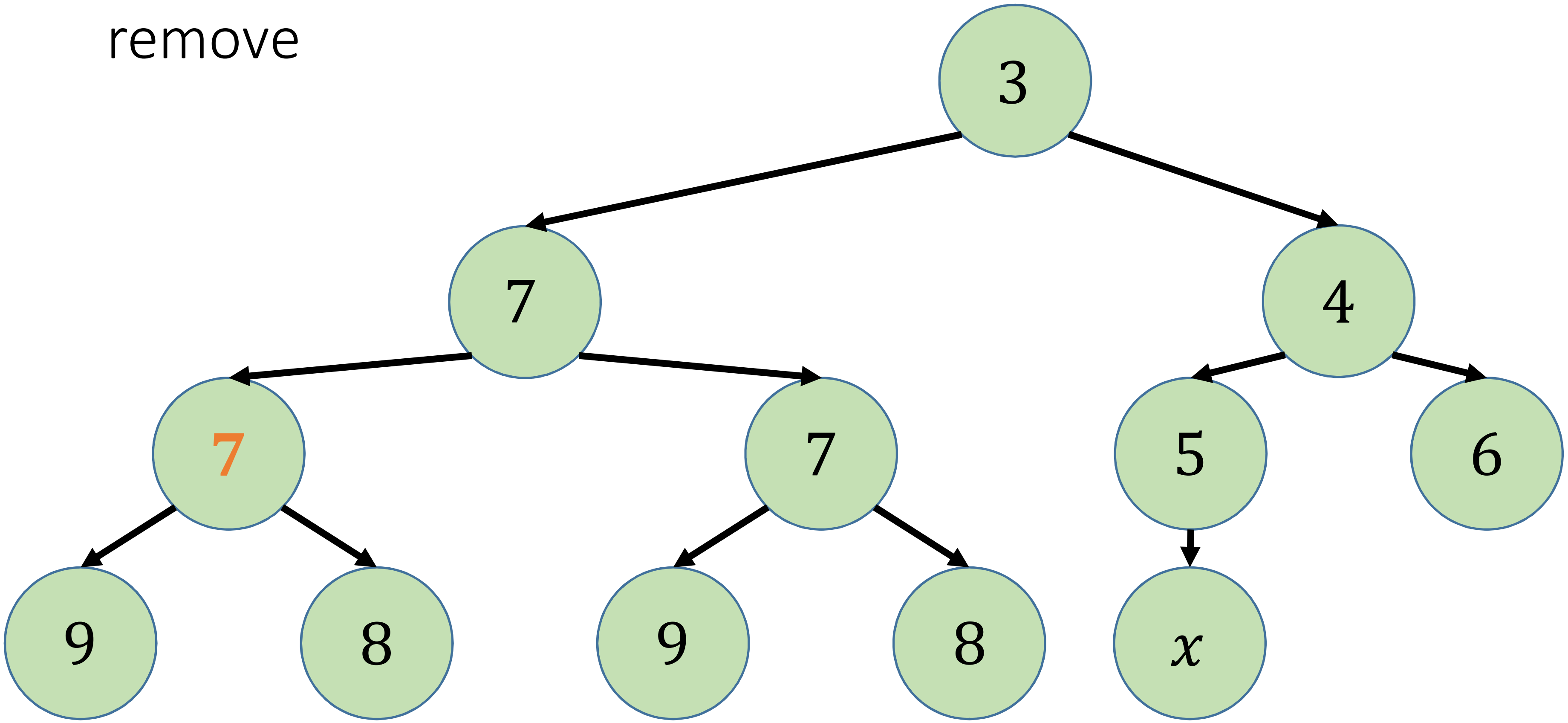


$$O(H) = O(\log N)$$

```
def heappop(data):
    swap(data[0], data[len(data) - 1])
    res = data.pop()
    sift_down(data, 0)
    return res
```

Heap push/pop/remove

remove

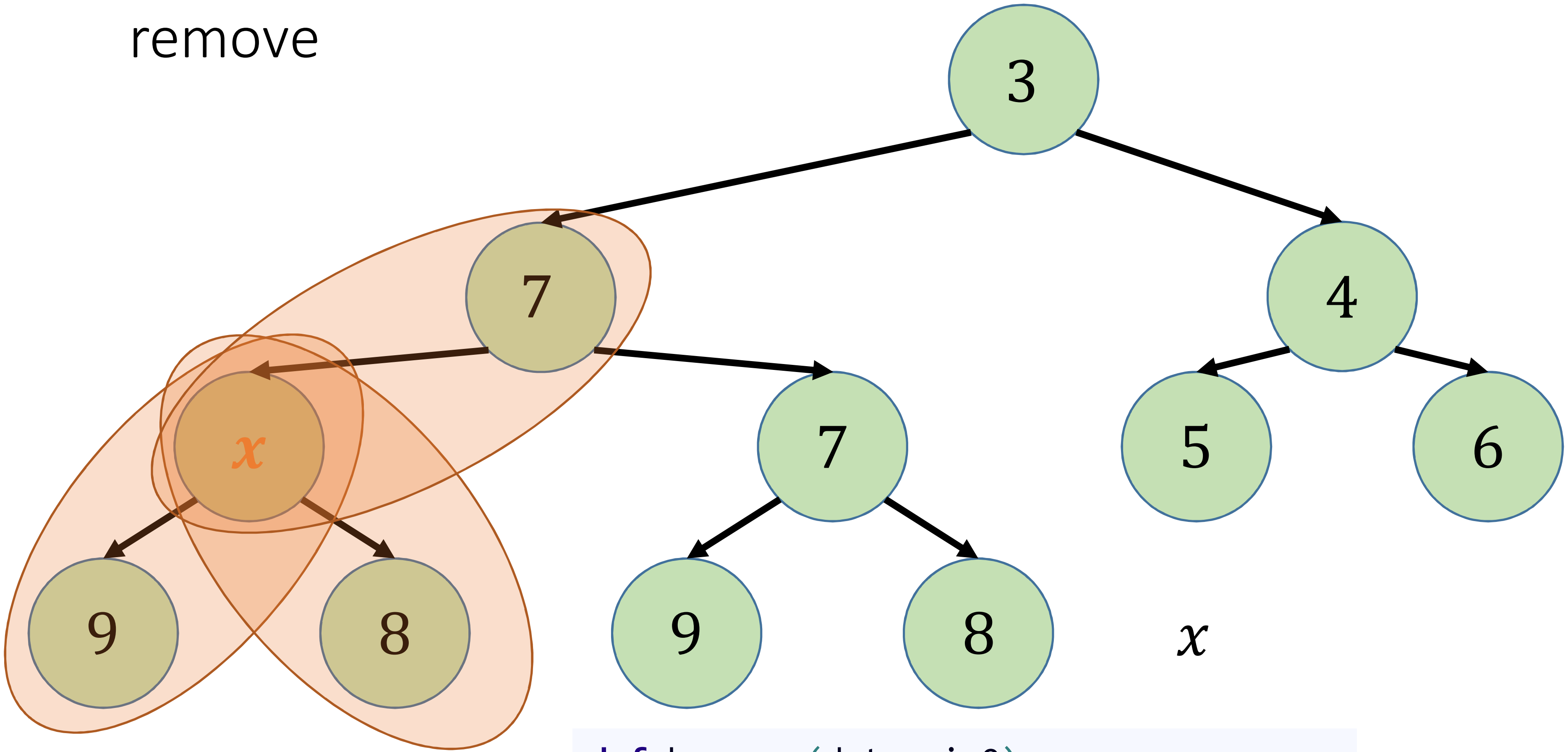


$$O(H) = O(\log N)$$

```
def heappop(data, i=0):
    swap(data[i], data[len(data) - 1])
    res = data.pop()
    sift_up(data, i)
    sift_down(data, i)
    return res
```

Heap push/pop/remove

remove

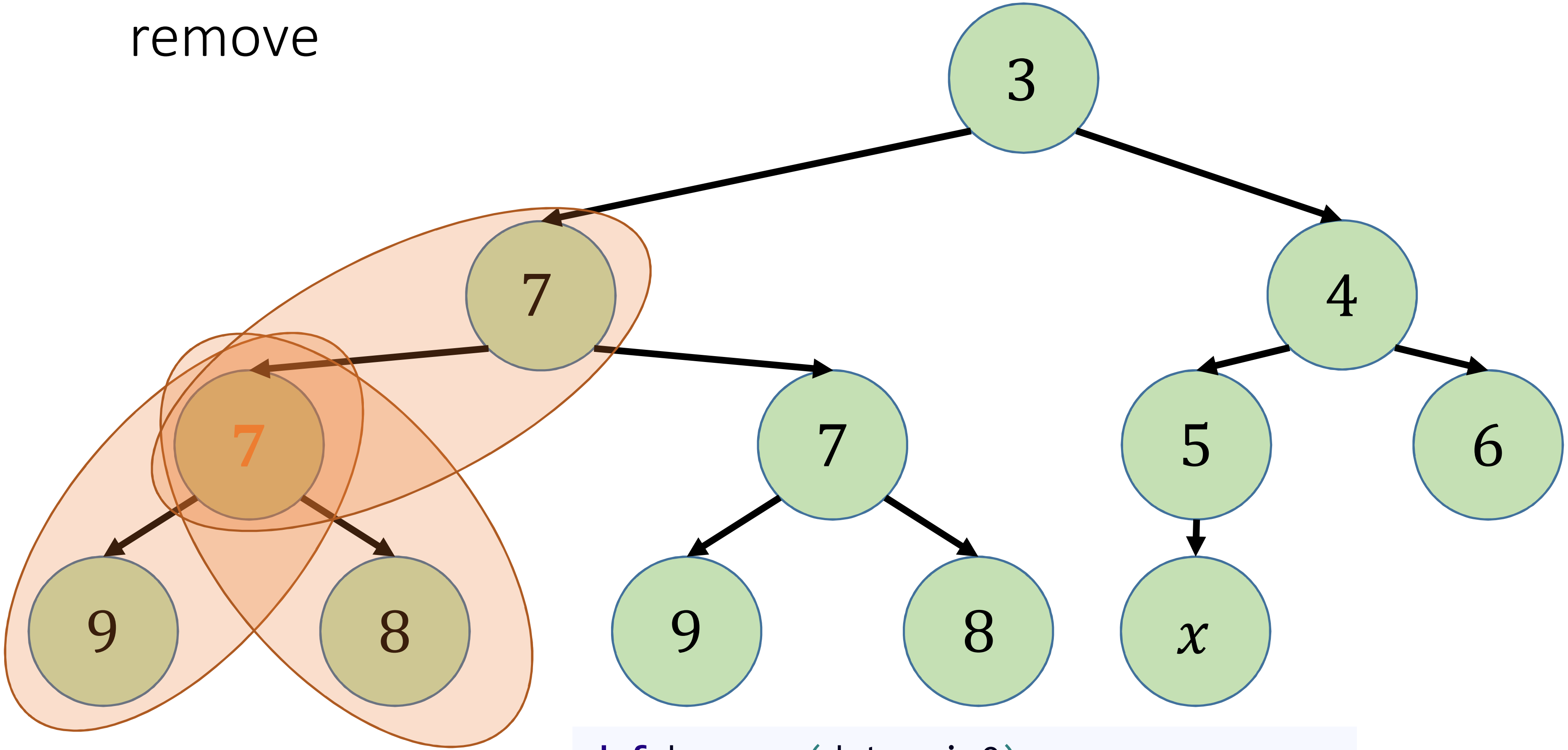



```python
def heappop(data, i=0):
    swap(data[i], data[len(data) - 1])
    res = data.pop()
    sift_up(data, i)
    sift_down(data, i)
    return res
```

$$O(H) = O(\log N)$$

Heap push/pop/remove

remove




```python
def heappop(data, i=0):
    swap(data[i], data[len(data) - 1])
    res = data.pop()
    sift_up(data, i)
    sift_down(data, i)
    return res
```

$$O(H) = O(\log N)$$

Heap push/pop/remove

Complexity comparison

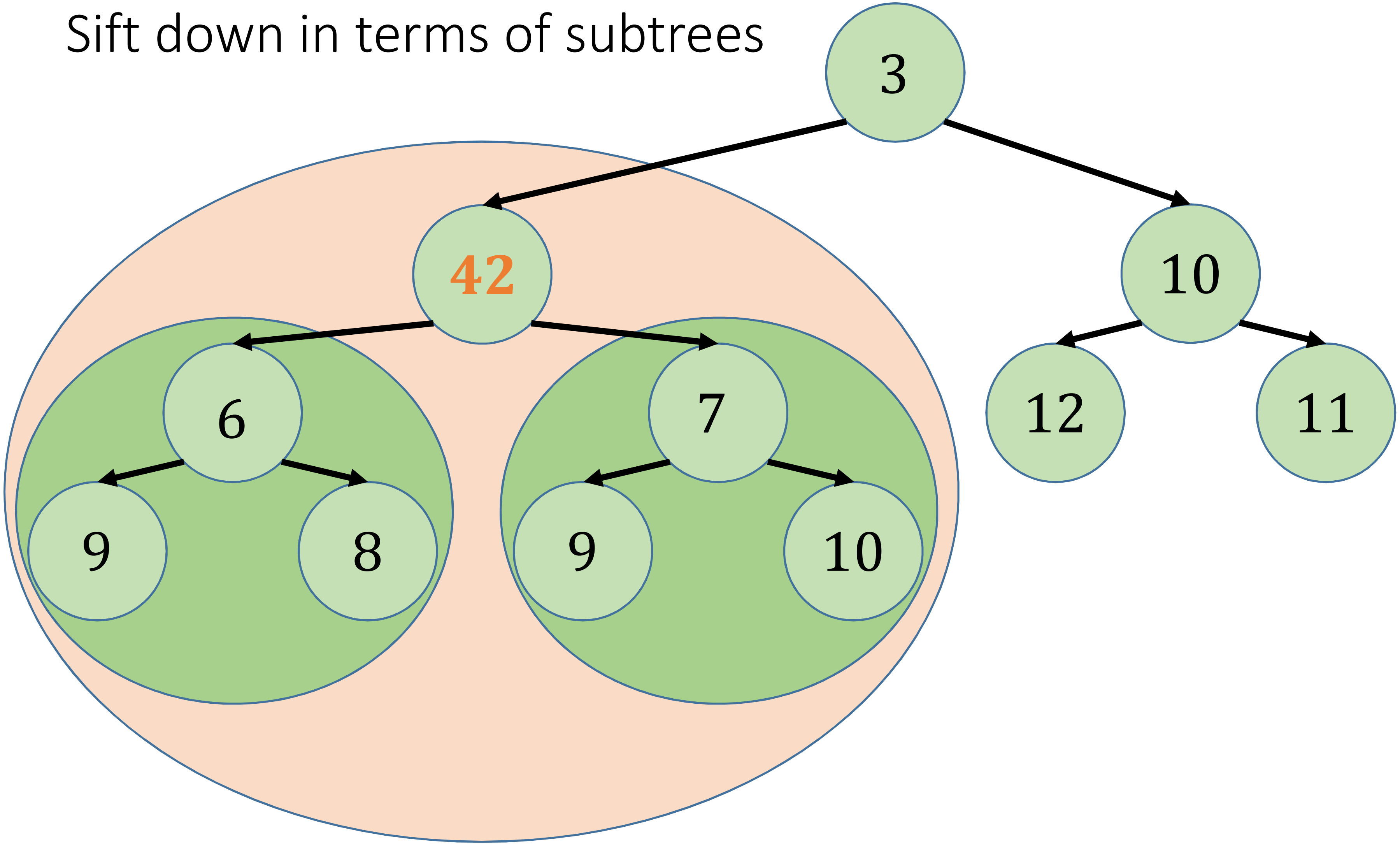
Operation	Vector (python list)	Linked list	Doubly Linked list	Sorted vector (sorted python list)	Heap
`push(x)`	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$\boldsymbol{O(\log N)}$
`remove(i)`	$O(N)$	$O(1)$	$O(1)$	$O(N)$	$\boldsymbol{O(\log N)}$
`find_min()`	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$\boldsymbol{O(1)}$
`len()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$\boldsymbol{O(1)}$

Binary heap

- Heap invariant
- Restoration of invariant (if element updated)
- Implementation on vector
- Push/pop/remove
- **Building heap**
- Implementation

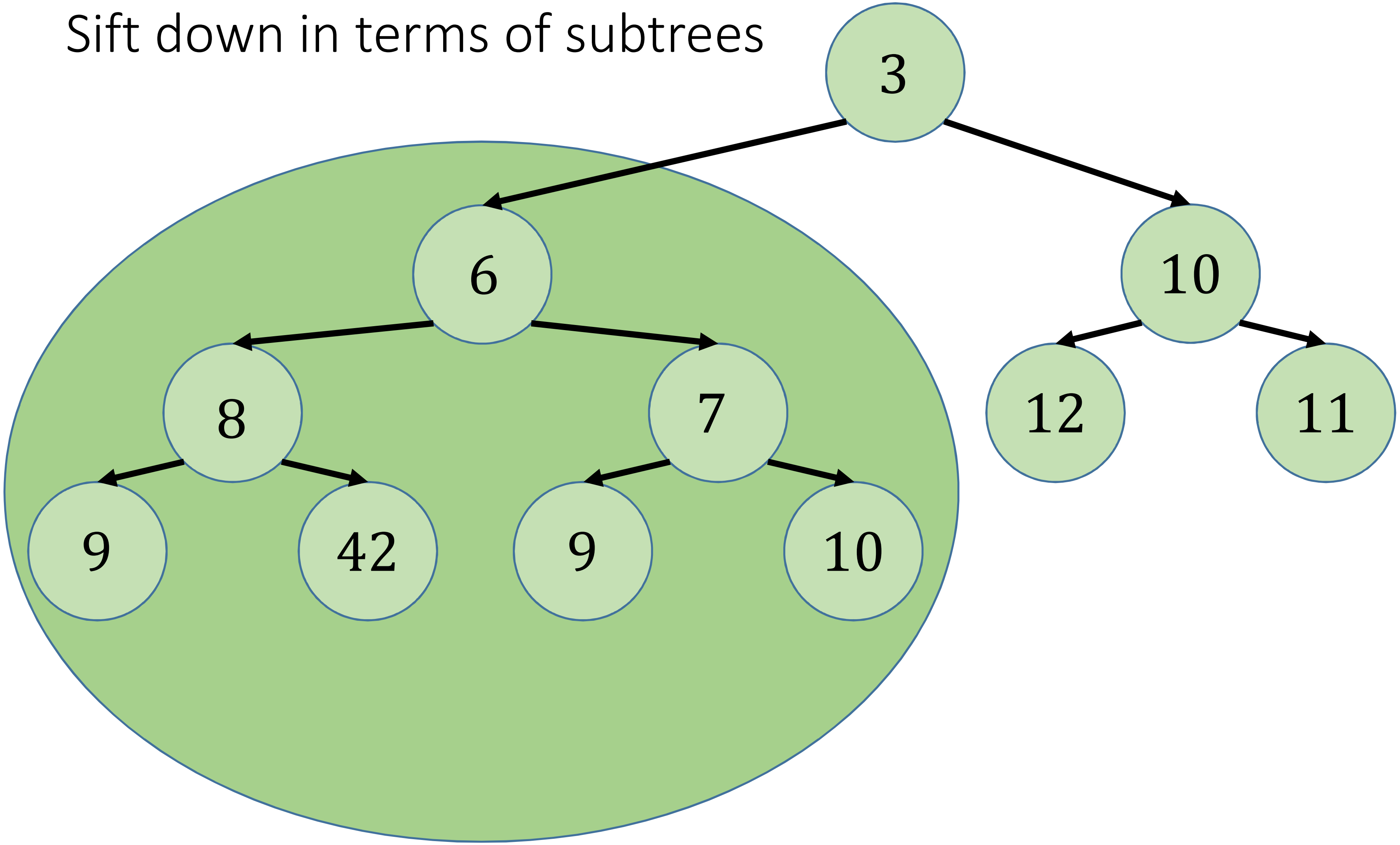
Building heap

Sift down in terms of subtrees



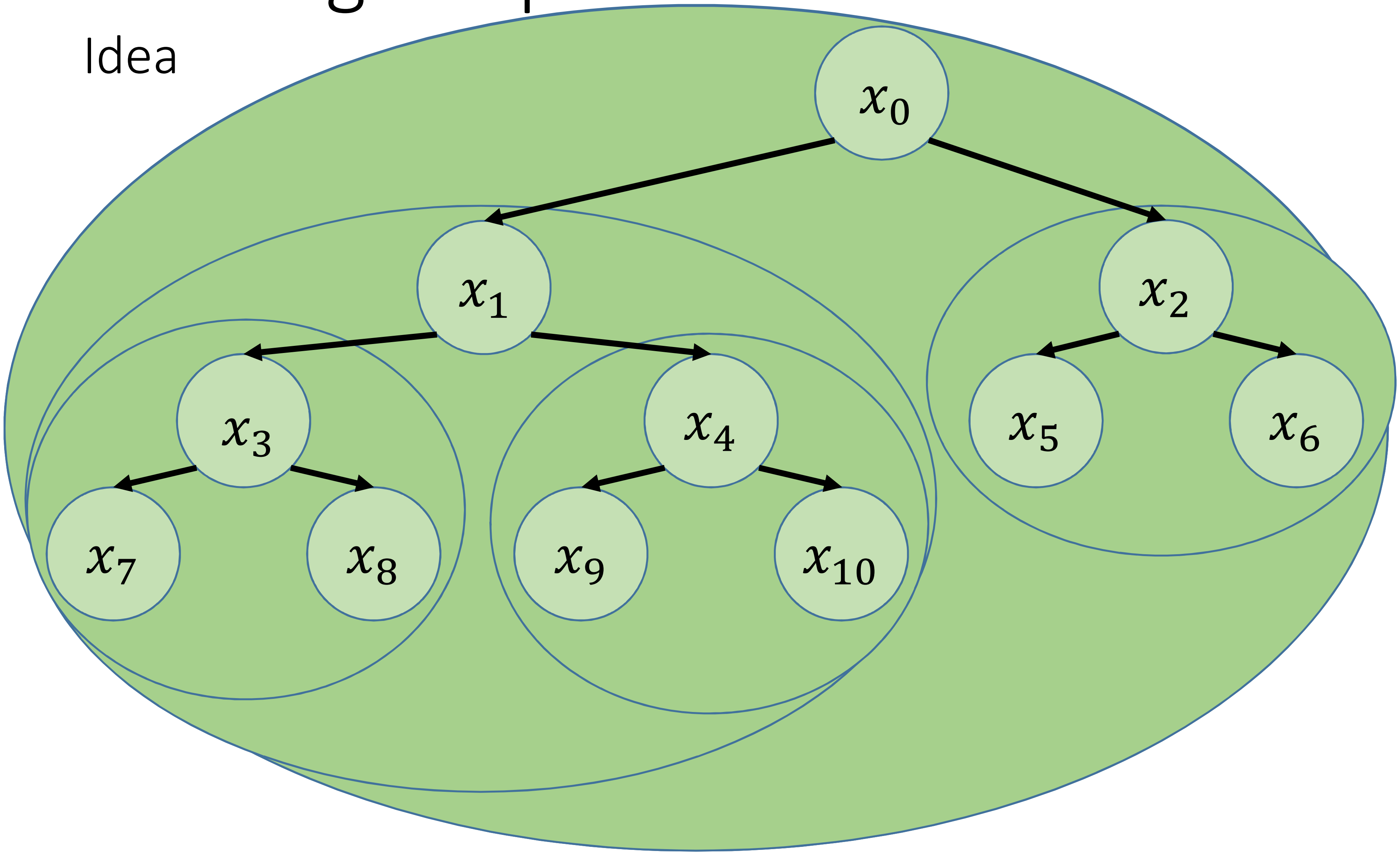
Building heap

Sift down in terms of subtrees



Building heap

Idea



Call `sift_down` in backward order starting from last node which has a child.

Building heap

Complexity

Call `sift_down` in backward order starting from last node which has a child.

Last node: $N-1$
Last node with child: $parent(N-1) = (N-2)//2$

```
def heapify(data):
    for i in range ((N - 2) // 2, -1, -1):
        sift_down(data, i)
```

Height of subtree for nodes on level i: $h(i) = H - i$
Number of nodes on level i: $2^i = 2^{H-h(i)} \leq \frac{N}{2^{h(i)}}$

Complexty of heapify():

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

$$T = \sum_{h=0}^{H} h \frac{N}{2^h} = N \boxed{\sum_{h=0}^{H} \frac{h}{2^h}} \leq 2N = O(N)$$

Binary heap

- Heap invariant
- Restoration of invariant (if element updated)
- Implementation on vector
- Push/pop/remove
- Building heap
- **Implementation**

Binary heap

Implementation

```
def sift_up(data, i):
    if i == 0:
        return
    parent = (i - 1) // 2
    if data[parent] > data[i]:
        data[parent], data[i] = data[i], data[parent]
        sift_up(data, parent)
```

```
def sift_down(data, i):
    child1 = i * 2 + 1
    child2 = i * 2 + 2
    if child1 >= len(data):
        return
    if child2 >= len(data):
        child_min = child1
    else:
        child_min = child1 if data[child1] < data[child2] else child2
    if data[child_min] < data[i]:
        data[i], data[child_min] = data[child_min], data[i]
        sift_down(data, child_min)
```

Binary heap

Implementation

```
def heapify(data):
    for i in range(len(data) - 1, -1, -1):
        sift_down(data, i)

def heappush(data, x):
    data.append(x)
    sift_up(data, len(data) - 1)

def heappop(data, i=0):
    data[i], data[-1] = data[-1], data[i]
    res = data.pop()
    sift_up(data, i)
    sift_down(data, i)
    return res
```

Binary heap

HeapSort

Let's implement a selection sort idea, but use heap for obtaining minimum value on each step instead of $O(N)$ minimum search:

```python
def heap_sort(x):
    heapify(x)
    return [heappop(x) for i in range(len(x))]
```

$$O(N) + O(N \log N) = O(N \log N)$$

Conclusion

Python built-ins

```
from heapq import heapify, heappush, heappop

data = [random.randint(0, 10000) for i in range(100)]
heapify(data)
heappush(data, x)
print(heappop(data))
```

Thank you for watching!