Chapter 11: Priority Queues and Heaps

Data Structures

The priority queue ADT | List-based implementations | Binary heaps | Array representation | Up-heap / down-heap | Building a heap | Heap sort | Priority scheduler

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_11_Priority_Queues_and_Heaps.html

INTRODUCTION

Learning Outcomes

The ADT

Explain the core operations of a priority queue and where it is used.

Binary heaps

Relate heap order and the complete-tree property to the array representation.

Operations

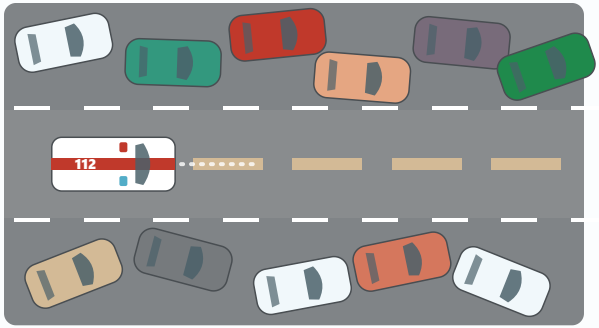
Carry out insertion, removal of the root, and heap construction by hand.

Cost analysis

Analyze the cost of each operation and of heap sort.

The Priority Queue

- Every item has a **priority** (its key), and items leave in priority order.
- Priority, not arrival order, decides who goes next: an ambulance overtakes the traffic.
- Typical uses: task scheduling, emergency rooms, event-driven simulation.
- Costs depend on the implementation: with a heap, **looking at** the top item is O(1) and **removing** it is O(log n); with an unsorted list even looking is O(n).



Not first come, first served: the highest priority goes first.

Basic Concepts

Entry = (key, value)

The key carries the priority; the value carries the actual data.

`(1, "heart attack")` · `(4, "sprain")`

Priority

Keys must be comparable (<, ≤). Usually a *smaller key means higher priority* (a min-priority queue).

Highest priority

The item at the "front": the smallest key in a min-queue, the largest in a max-queue.

Equal priority

Items with equal keys are *not guaranteed* to leave in FIFO order; if that matters, add an arrival sequence number to the key.

Like Goodrich, we define the ADT as a min-priority queue (removeMin), while the heap code uses a max-heap (insert / removeMax). The choice only flips the direction of the comparisons.

Queue or Priority Queue?

Four patients arrive at the emergency room in this order (a smaller number is more urgent):

Arrival	1st	2nd	3rd	4th
Patient (priority)	Alan (3)	Eve (1)	Carl (2)	Nina (1)
Queue (FIFO)	Alan	Eve	Carl	Nina
Priority queue	Eve	Nina	Carl	Alan

Queue
Leaving order = arrival order: first come, first served.

Priority queue
Leaving order = priority order. Eve and Nina tie at 1; here we kept their arrival order.

Applications

Operating Systems

Process scheduling: a higher-priority process gets the CPU first.

`interrupt handler > UI > backup`

Graph Algorithms

Dijkstra's shortest paths and A* search take the next closest vertex from a priority queue.

Emergency Management

Patient er and incident handling: the most urgent case is handled first.

Data Compression

Huffman coding merges the two lowest-frequency nodes at every step.

Core Operations

Operation	Meaning
`insert(k, v)`	**Insert**: the entry takes its place according to its priority.
`removeMin()`	**Remove**: removes and returns the highest-priority entry.
`min()`	**Peek**: returns the entry without removing it.
`size()`	Number of entries.
`isEmpty()`	Is the queue empty?

```
public interface PriorityQueue<K, V> {
    void insert(K key, V value);
    Entry<K, V> min();         // null if empty
    Entry<K, V> removeMin();   // null if empty
    int size();
    boolean isEmpty();
}
```

In a max-priority queue the same operations are called `max()` and `removeMax()`. The heap code in this chapter uses a **max-heap** (`insert`, `removeMax`).

A Sample Sequence of Operations (min-priority)

Operation	Returns	Queue contents
`insert(5, A)`		{ (5,A) }
`insert(9, C)`		{ (5,A), (9,C) }
`insert(3, B)`		{ (3,B), (5,A), (9,C) }
`min()`	(3,B)	{ (3,B), (5,A), (9,C) }
`removeMin()`	(3,B)	{ (5,A), (9,C) }
`insert(7, D)`		{ (5,A), (7,D), (9,C) }
`removeMin()`	(5,A)	{ (7,D), (9,C) }
`removeMin()`	(7,D)	{ (9,C) }
`removeMin()`	(9,C)	{ }
`removeMin()`	null	{ } empty queue

Unsorted List vs. Sorted List

Unsorted list



min needs a full scan

- `insert` : append at the end → **O(1)**
- `min` , `removeMin` : scan the whole list → **O(n)**

Sorted list

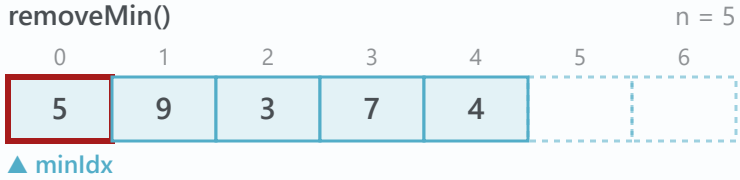


min find a slot for 4, shift the rest

- `insert` : find the right slot → **O(n)**
- `min` , `removeMin` : the minimum is always at one end → **O(1)**

One is fast at inserting, the other at removing. The **binary heap** balances the two (both O(log n)).

Code Trace: `removeMin()` on an unsorted array



Unsorted array a = [5, 9, 3, 7, 4]: insertions go at the end, so the minimum could be anywhere. First candidate: **minIdx = 0**.

Step 1 / 9

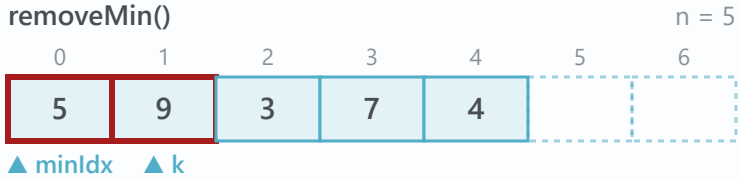
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
▶   int minIdx = 0;                 // index of the
  minimum
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `minIdx = 0`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



k = 1: a[1] = 9 ≥ a[0] = 5 → minIdx stays.

Step 2 / 9

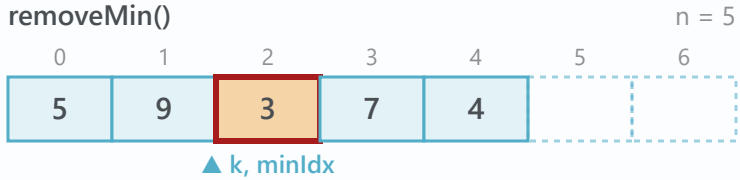
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `k = 1` `minIdx = 0`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



k = 2: a[2] = 3 < a[0] = 5 → new minimum, **minIdx = 2**.

Step 3 / 9

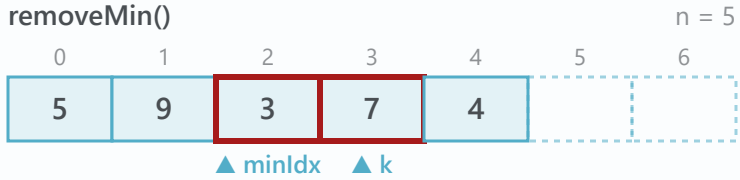
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `k = 2` `minIdx = 2`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



k = 3: a[3] = 7 ≥ a[2] = 3 → minIdx stays.

Step 4 / 9

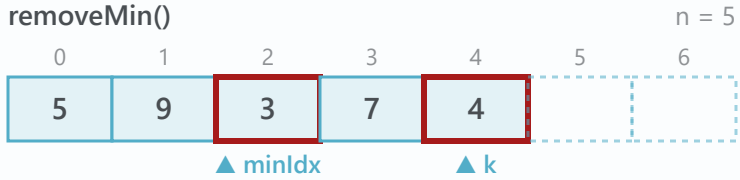
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `k = 3` `minIdx = 2`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



k = 4: a[4] = 4 ≥ a[2] = 3 → minIdx stays.

Step 5 / 9

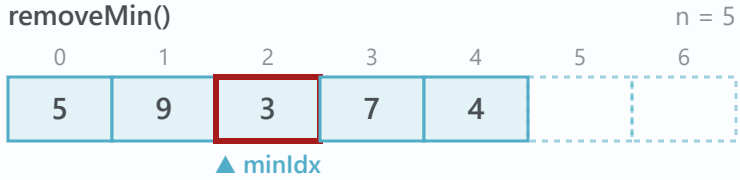
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `k = 4` `minIdx = 2`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



k = 5 = n → scan done: 4 comparisons. Every removeMin scans the whole array → **O(n)**.

Step 6 / 9

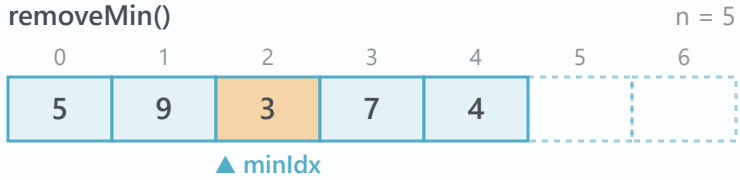
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
▶   for (int k = 1; k < n; k++)
5     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `k = 5` `minIdx = 2`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



min = a[2] = 3 is saved.

Step 7 / 9

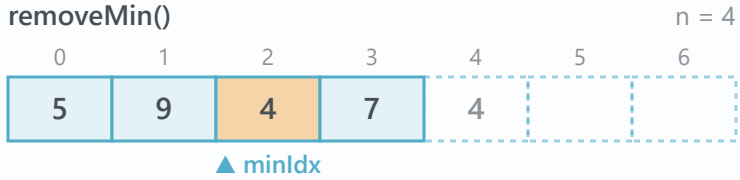
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[minIdx]) minIdx = k;
▶   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 5` `minIdx = 2` `min = 3`

`a = [5, 9, 3, 7, 4]`

Code Trace: `removeMin()` on an unsorted array



To fill the gap without shifting, the last item (4) moves to index 2; **n = 4**. Order doesn't matter, so this is O(1).

Step 8 / 9

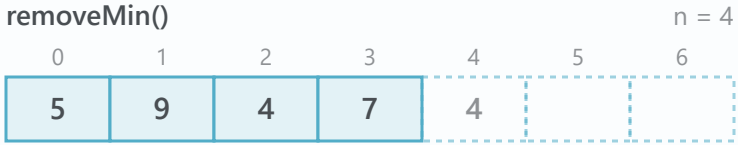
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
▶   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
8   return min;
9 }
```

VARIABLES

`n = 4` `minIdx = 2` `min = 3`

`a = [5, 9, 4, 7 | 4]`

Code Trace: `removeMin()` on an unsorted array



return 3. The remaining array [5, 9, 4, 7] is still unsorted.

Step 9 / 9

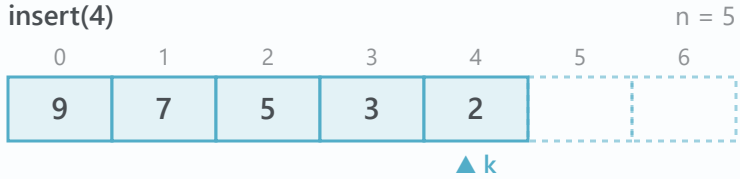
```
1 // unsorted array: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // requires n > 0
3   int minIdx = 0;                 // index of the
  minimum
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[minIdx]) minIdx = k;
6   int min = a[minIdx];
7   a[minIdx] = a[n - 1];  n--;    // last item fills
  the gap
▶   return min;
9 }
```

VARIABLES

`n = 4` `min = 3`

`a = [5, 9, 4, 7 | 4]`

Code Trace: `insert(4)` on a sorted array



The array is sorted from largest to smallest, so the highest-priority (smallest) item is always last. **k = n − 1 = 4**: search for the slot from the back.

Step 1 / 10

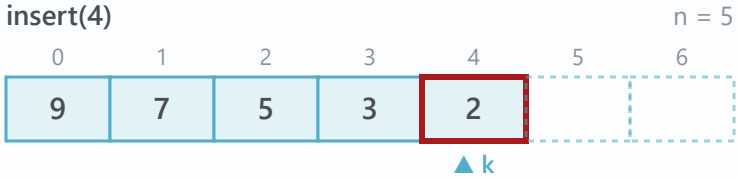
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
▶    int k = n - 1;
4    while (k >= 0 && a[k] < x) {     // items smaller than x
5      a[k + 1] = a[k];               // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2]`

Code Trace: `insert(4)` on a sorted array



a[4] = 2 < 4 → 2 belongs after x, so it shifts.

Step 2 / 10

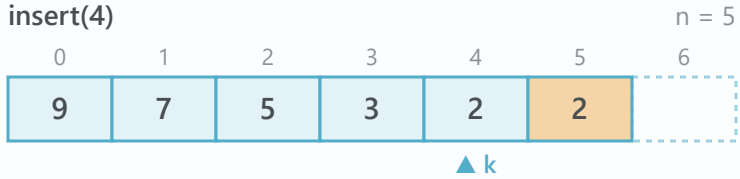
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶    while (k >= 0 && a[k] < x) {    // items smaller than x
5      a[k + 1] = a[k];              // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2, None]`

Code Trace: `insert(4)` on a sorted array



a[5] = a[4] = 2: shifted one to the right (a[4] is a stale copy for now and will be overwritten).

Step 3 / 10

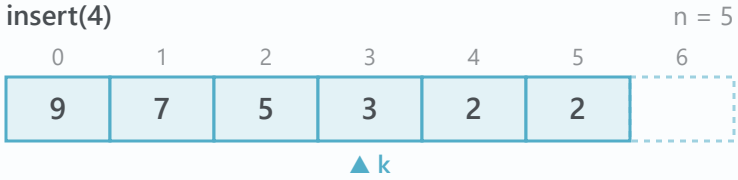
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // items smaller than x
▶      a[k + 1] = a[k];              // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2, 2]`

Code Trace: `insert(4)` on a sorted array



k = 3: move one to the left.

Step 4 / 10

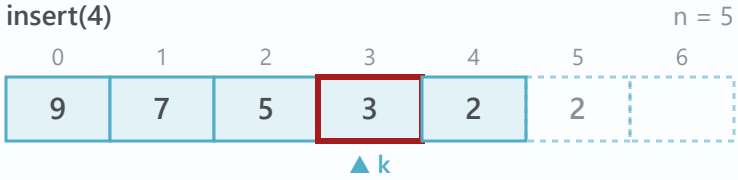
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // items smaller than x
5      a[k + 1] = a[k];              // shift one to the right
▶      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

x = 4 | n = 5 | k = 3

a = [9, 7, 5, 3, 2, 2]

Code Trace: `insert(4)` on a sorted array



a[3] = 3 < 4 → 3 belongs after x, so it shifts.

Step 5 / 10

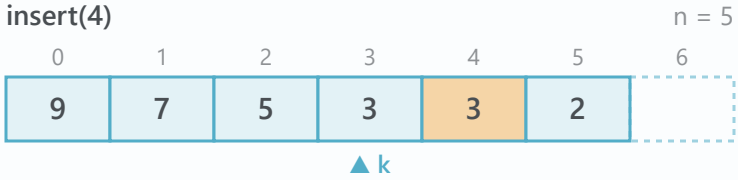
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶    while (k >= 0 && a[k] < x) {    // items smaller than x
5      a[k + 1] = a[k];              // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 3`

`a = [9, 7, 5, 3, 2]`

Code Trace: `insert(4)` on a sorted array



a[4] = a[3] = 3: shifted one to the right (a[3] is a stale copy for now and will be overwritten).

Step 6 / 10

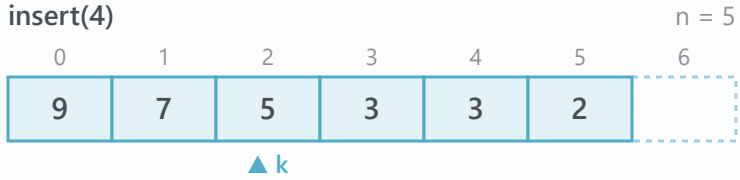
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // items smaller than x
▶      a[k + 1] = a[k];              // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 3`

`a = [9, 7, 5, 3, 3, 2]`

Code Trace: `insert(4)` on a sorted array



k = 2: move one to the left.

Step 7 / 10

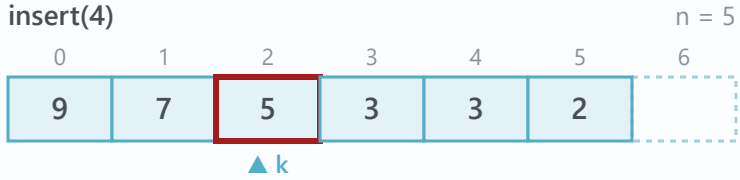
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // items smaller than x
5      a[k + 1] = a[k];              // shift one to the right
▶      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 2`

`a = [9, 7, 5, 3, 3, 2]`

Code Trace: `insert(4)` on a sorted array



a[2] = 5 ≥ 4 → the loop ends. 4 goes at **k + 1 = 3**.

Step 8 / 10

```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶    while (k >= 0 && a[k] < x) {     // items smaller than x
5      a[k + 1] = a[k];               // shift one to the right
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 5` `k = 2`

`a = [9, 7, 5, 3, 3, 2]`

Code Trace: `insert(4)` on a sorted array



a[3] = 4, n = 6. The array is still sorted. In the worst case (x is the largest) every item shifts → **O(n)**.

Step 9 / 10

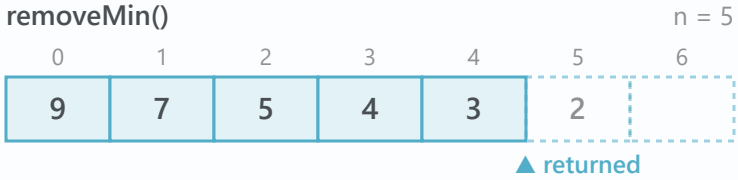
```
1  // sorted array: largest to smallest, min always at a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {     // items smaller than x
5      a[k + 1] = a[k];               // shift one to the right
6      k--;
7    }
▶    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

VARIABLES

`x = 4` `n = 6` `k = 2`

`a = [9, 7, 5, 4, 3, 2]`

Code Trace: `insert(4)` on a sorted array



In return, **removeMin()** is cheap: the minimum (2) is already last → n = 5, **return 2**. No shifting → **O(1)**.

Step 10 / 10

```
1 // sorted array: largest to smallest, min always at a[n-1]
2 public void insert(int x) {
3   int k = n - 1;
4   while (k >= 0 && a[k] < x) {    // items smaller than x
5     a[k + 1] = a[k];              // shift one to the right
6     k--;
7   }
8   a[k + 1] = x;  n++;
9 }
▶ public int removeMin() { n--; return a[n]; }
```

VARIABLES

`n = 5` `a = [9, 7, 5, 4, 3 | 2]`

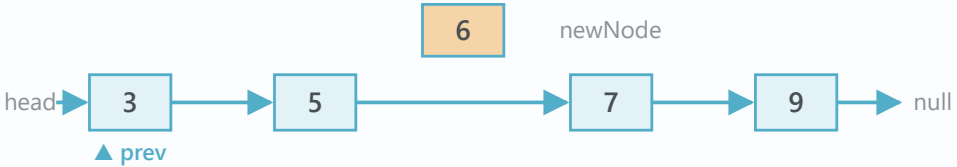
Array and Linked-List Representations

Array representation

- Items live in an array; keeping it sorted means shifting items on every insertion (O(n)).
- If the top-priority item sits at the *front*, every removal shifts the whole array left: O(n).
- Better: keep the top-priority item at the **end** → removal is O(1).

Linked-list representation

- Each item is linked in at the position its priority calls for: finding it is O(n), linking is O(1).
- The top-priority item is at the head, so removal = deleting the head, O(1).
- No shifting, but every node carries an extra reference.



This list runs **from smallest to largest** (min-priority), so the highest-priority item, 3, is at the head. A new node 6 has been created.

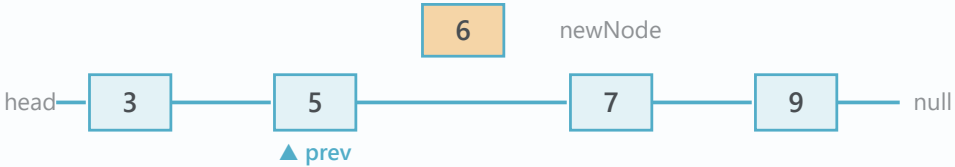
Array and Linked-List Representations

Array representation

- Items live in an array; keeping it sorted means shifting items on every insertion (O(n)).
- If the top-priority item sits at the *front*, every removal shifts the whole array left: O(n).
- Better: keep the top-priority item at the **end** → removal is O(1).

Linked-list representation

- Each item is linked in at the position its priority calls for: finding it is O(n), linking is O(1).
- The top-priority item is at the head, so removal = deleting the head, O(1).
- No shifting, but every node carries an extra reference.



Finding the slot: advance until `prev.next.value = 7 > 6` → **prev = 5**. This scan takes O(n).

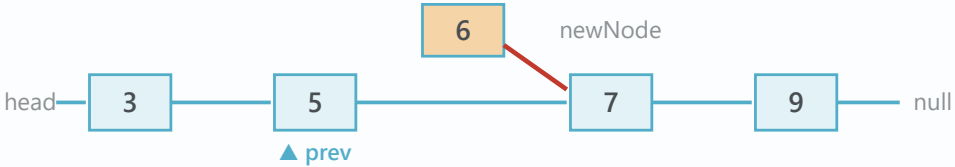
Array and Linked-List Representations

Array representation

- Items live in an array; keeping it sorted means shifting items on every insertion (O(n)).
- If the top-priority item sits at the *front*, every removal shifts the whole array left: O(n).
- Better: keep the top-priority item at the **end** → removal is O(1).

Linked-list representation

- Each item is linked in at the position its priority calls for: finding it is O(n), linking is O(1).
- The top-priority item is at the head, so removal = deleting the head, O(1).
- No shifting, but every node carries an extra reference.



First `newNode.next = prev.next` (6 → 7). Order matters: without this link in place, the path to 7 would be lost.

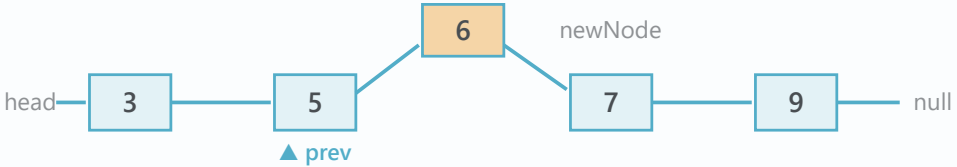
Array and Linked-List Representations

Array representation

- Items live in an array; keeping it sorted means shifting items on every insertion (O(n)).
- If the top-priority item sits at the *front*, every removal shifts the whole array left: O(n).
- Better: keep the top-priority item at the **end** → removal is O(1).

Linked-list representation

- Each item is linked in at the position its priority calls for: finding it is O(n), linking is O(1).
- The top-priority item is at the head, so removal = deleting the head, O(1).
- No shifting, but every node carries an extra reference.



Then `prev.next = newNode` (5 → 6). Linking is O(1) with no shifting, and removal always happens at the head: O(1).

A Sorted-Array Priority Queue: Code

```
class Item<T> {
    T data;
    int priority;
    Item(T d, int p) { data = d; priority = p; }
}
```

```
// fields of SortedArrayPQ<E>
private Item<E>[] items =
    (Item<E>[]) new Item[10];
private int size = 0;
public boolean isEmpty() { return size == 0; }
```

```
public void insert(E data, int p) {
    if (size == items.length)
        items = Arrays.copyOf(items, 2 * size);
    int i = size;
    while (i > 0 && items[i-1].priority < p) {
        items[i] = items[i - 1];  // shift right
        i--;
    }
    items[i] = new Item<>(data, p);
    size++;
}
```

```
public E remove() {
    if (isEmpty()) return null;
    E result = items[0].data;
    for (int i = 1; i < size; i++)
        items[i - 1] = items[i];
    items[--size] = null;
    return result;
}
```

Items are stored as (data, priority) pairs, with the highest priority first, in items[0]. This simple version ignores the advice on the previous slide: insert is O(n) (shifting right) and so is remove (shifting left). Keeping the items in ascending order would shrink remove to one line (`items[--size]`) and O(1). The strict `<` comparison preserves arrival order among equal priorities.

Code Trace: `insert(D, 5)` with a Tie



After insert(A, 2), insert(B, 5), insert(C, 1) the array is [B:5, A:2, C:1]. Now `insert(D, 5)`: two items will shift.

Step 1 / 10

```
▶ public void insert(E data, int p) {
2   if (size == items.length)
3     items = Arrays.copyOf(items, 2 * size);
4   int i = size;
5   while (i > 0 && items[i-1].priority < p) {
6     items[i] = items[i - 1];  // shift right
7     i--;
8   }
9   items[i] = new Item<>(data, p);
10  size++;
11 }
```

VARIABLES

`data = –` `p = –` `i = –`

`size = 3` `items = [B:5, A:2, C:1]`

CALL STACK

`main()`

Code Trace: `insert(D, 5)` with a Tie



`insert(D, 5)` : size 3 == 10? No, so no need to grow.

Step 2 / 10

```
1  public void insert(E data, int p) {
▶    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = –`

`size = 3` `items = [B:5, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



i = size = 3: start at the first free slot and search leftward.

Step 3 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 3`

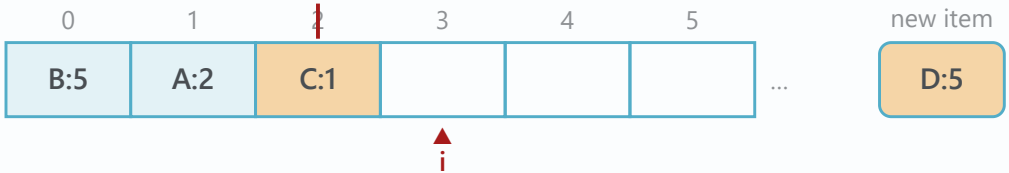
`size = 3` `items = [B:5, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



i > 0 and items[2].priority = 1 < 5 → **true**: the item on the left has lower priority.

Step 4 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
▶    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 3`

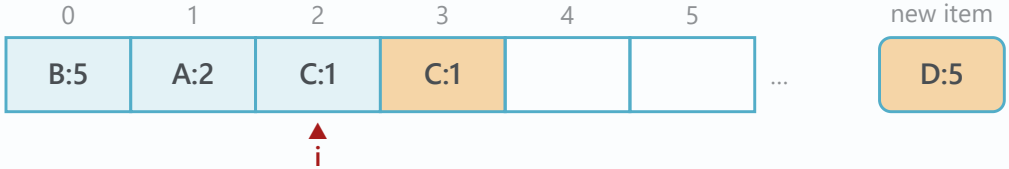
`size = 3` `items = [B:5, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



items[3] = items[2]: C shifts one to the right; **i = 2**.

Step 5 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
▶      items[i] = items[i - 1];  // shift right
▶      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 2`

`size = 3`

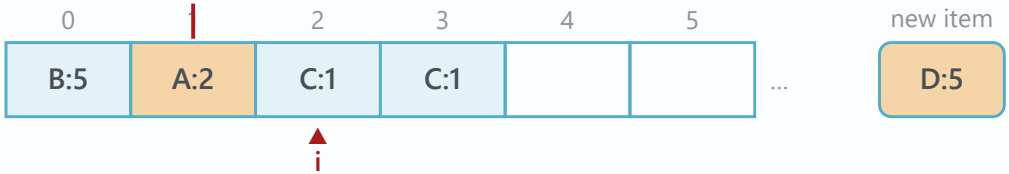
`items = [B:5, A:2, C:1, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



i > 0 and items[1].priority = 2 < 5 → **true**: the item on the left has lower priority.

Step 6 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
▶    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 2` `size = 3`

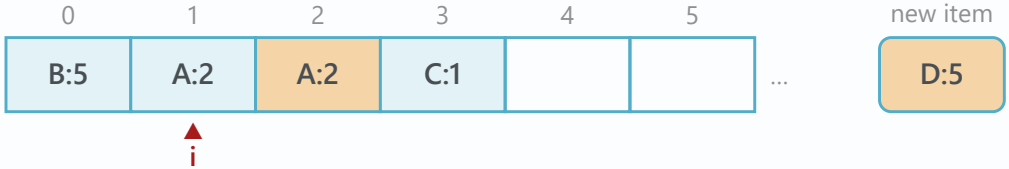
`items = [B:5, A:2, C:1, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



items[2] = items[1]: A shifts one to the right; **i = 1**.

Step 7 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
▶      items[i] = items[i - 1];  // shift right
▶      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 1`

`size = 3`

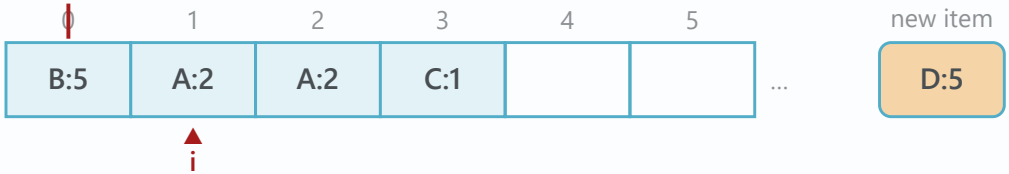
`items = [B:5, A:2, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



items[0].priority = 5 < 5 → **false** (on a tie the earlier arrival stays ahead: FIFO), so the loop ends.

Step 8 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
▶    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

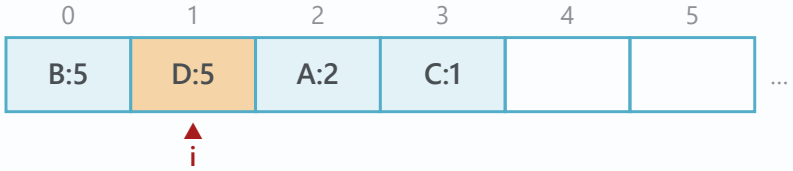
`data = D` `p = 5` `i = 1` `size = 3`

`items = [B:5, A:2, A:2, C:1]`

CALL STACK

- **`insert(D, 5)`**
- `main()`

Code Trace: `insert(D, 5)` with a Tie



`items[1] = (D, 5)` : D fills the gap.

Step 9 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
▶    items[i] = new Item<>(data, p);
10   size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 1`

`size = 3`

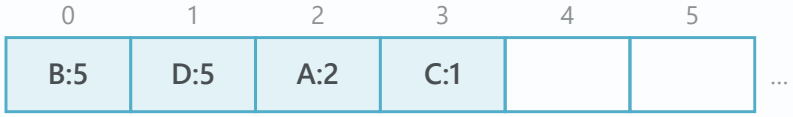
`items = [B:5, D:5, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Code Trace: `insert(D, 5)` with a Tie



`size++` → 4. The array is still sorted from largest to smallest.

Step 10 / 10

```
1  public void insert(E data, int p) {
2    if (size == items.length)
3      items = Arrays.copyOf(items, 2 * size);
4    int i = size;
5    while (i > 0 && items[i-1].priority < p) {
6      items[i] = items[i - 1];  // shift right
7      i--;
8    }
9    items[i] = new Item<>(data, p);
▶    size++;
11 }
```

VARIABLES

`data = D` `p = 5` `i = 1`

`size = 4`

`items = [B:5, D:5, A:2, C:1]`

CALL STACK

`insert(D, 5)`

`main()`

Comparing Costs

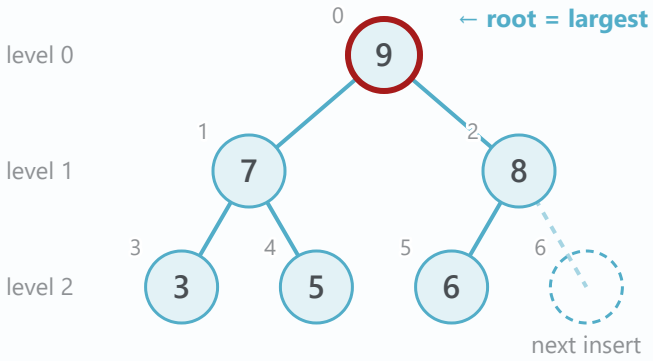
Operation	Unsorted list	Sorted list	Binary heap
`size`, `isEmpty`	O(1)	O(1)	O(1)
`insert`	**O(1)**	O(n)	O(log n)
`min`	O(n)	**O(1)**	**O(1)**
`removeMin`	O(n)	**O(1)**	O(log n)
Filling with n items	O(n)	O(n log n)*	O(n)**

* By sorting first. ** With bottom-up heap construction.

n insertions + n removals: **$O(n^2)$** with either list, **O(n log n)** with a heap. For $n = 10^6$ that is a factor of about 50,000.

The Binary Heap

The binary heap is the most common way to implement a priority queue. It is a binary tree that satisfies two conditions at once:



Siblings are unordered: swapping 7 and 8 still gives a heap.

1. Heap-order property

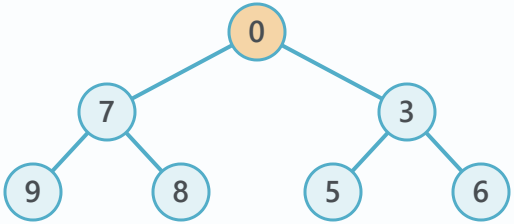
Every node *comes before* its children: in a max-heap, parent ≥ child. The highest-priority item is at the **root**.

2. Complete binary tree property

Every level except the last is full, and the last fills from left to right → height $\lfloor \log_2 n \rfloor$, and no gaps in the array.

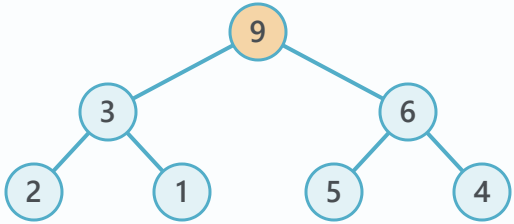
Insertion and removal work along a *single path* from the root to a leaf → **O(log n)**. A heap is *not* a binary search tree.

Min-Heaps and Max-Heaps



Min-heap: smallest key at the root

`key(parent) ≤ key(child)`

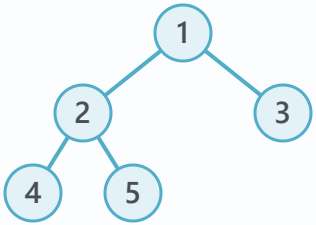


Max-heap: largest key at the root

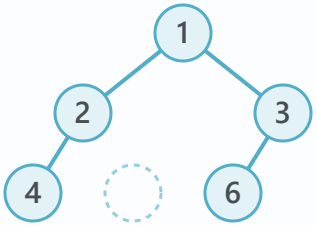
`key(parent) ≥ key(child)`

- The order holds only between **parent and child**: in the min-heap, 7 and 3 are siblings, so 7 > 3 is fine.
- Every path from the root down to a leaf is sorted (increasing in a min-heap, decreasing in a max-heap).

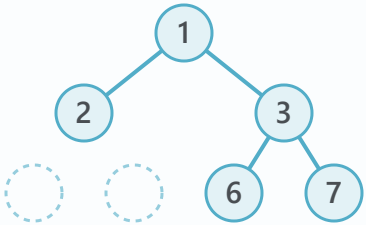
Complete Binary Trees



Complete ✓
last level filled from the left



Not complete ✗
a gap in the last level



Not complete ✗
last level not left-aligned

Height

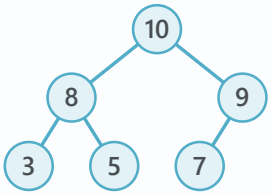
Level d holds at most 2^d nodes, so a complete binary tree with n nodes has height **h = $\lfloor \log_2 n \rfloor$**.

`n = 7 → h = 2 · n = 1000 → h = 9`

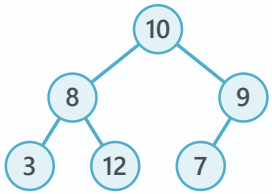
Why it matters

A new item always goes into "the leftmost free spot on the last level", and in the array that is exactly **index n**.

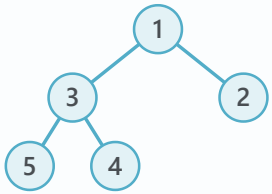
Heap or Not?



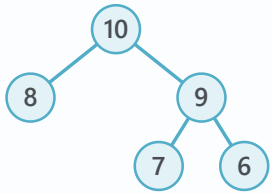
(a)



(b)



(c)

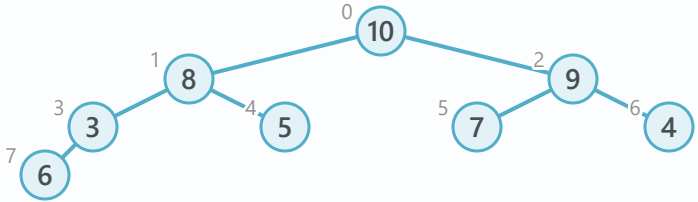


(d)

For each tree, decide: max-heap, min-heap, or not a heap at all? Which property fails?

▶ **Show answers**

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`



isHeap(a) n = 8

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

isHeap(a) is called on an 8-item array. Every node except the root (index 0) is compared once with its **parent**.

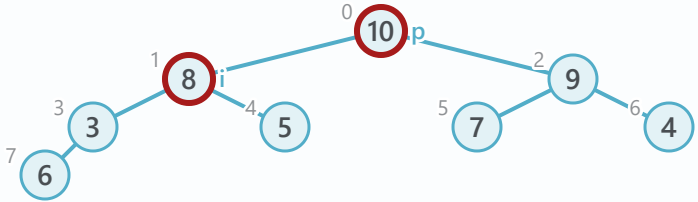
Step 1 / 9

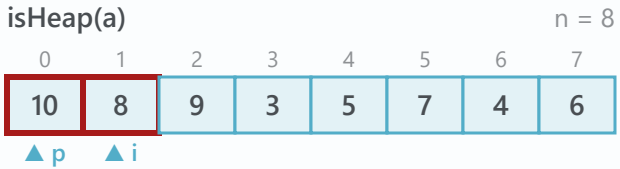
```
▶ static boolean isHeap(int[] a) {  // max-heap?
2   for (int i = 1; i < a.length; i++) {
3     int p = (i - 1) / 2;           // parent
4     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`





i = 1: parent p = (1 − 1) / 2 = 0. a[0] = 10 ≥ a[1] = 8 ✓ → continue.

Step 2 / 9

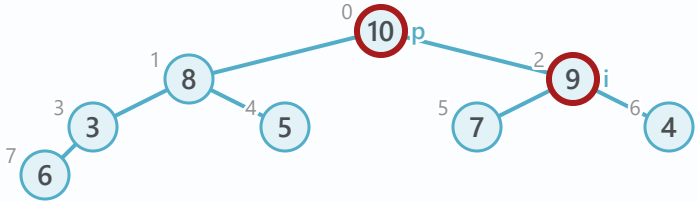
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 1` `p = 0`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`



isHeap(a) n = 8

0	1	2	3	4	5	6	7
10	8	9	3	5	7	4	6
▲ p		▲ i					

i = 2: parent p = (2 − 1) / 2 = 0. a[0] = 10 ≥ a[2] = 9 ✓ → continue.

Step 3 / 9

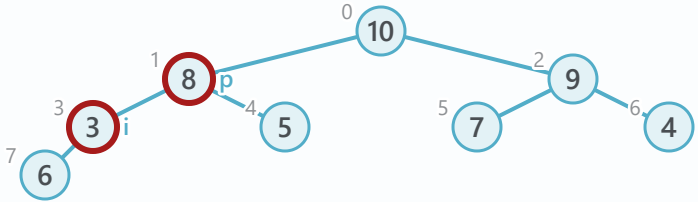
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

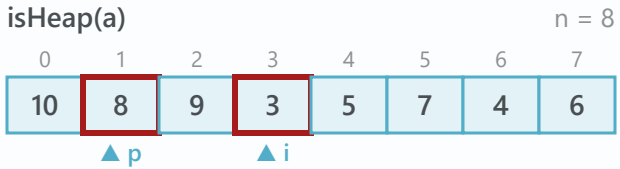
VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 2` `p = 0`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`





i = 3: parent p = (3 − 1) / 2 = 1. a[1] = 8 ≥ a[3] = 3 ✓ → continue.

Step 4 / 9

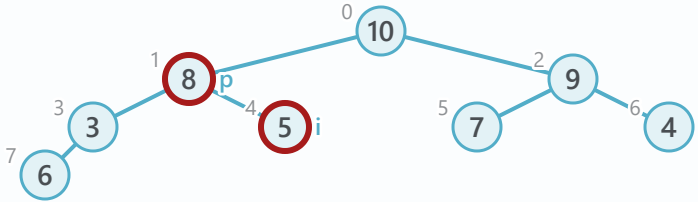
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

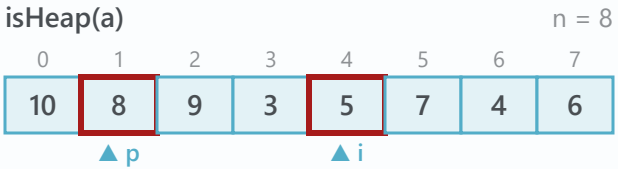
VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 3` `p = 1`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`





i = 4: parent p = (4 − 1) / 2 = 1. a[1] = 8 ≥ a[4] = 5 ✓ → continue.

Step 5 / 9

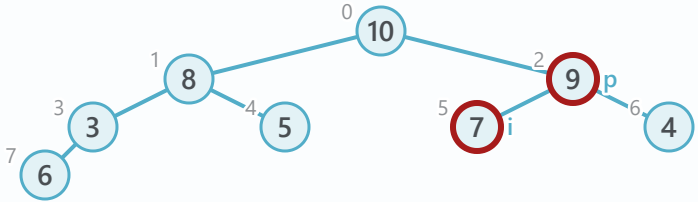
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

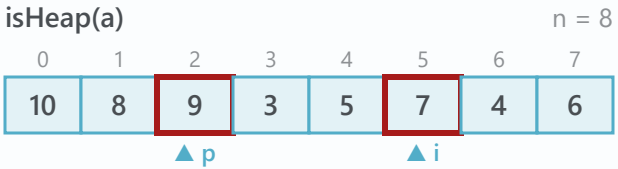
VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 4` `p = 1`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`





i = 5: parent p = (5 − 1) / 2 = 2. a[2] = 9 ≥ a[5] = 7 ✓ → continue.

Step 6 / 9

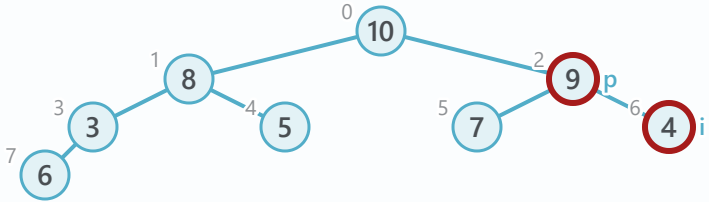
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

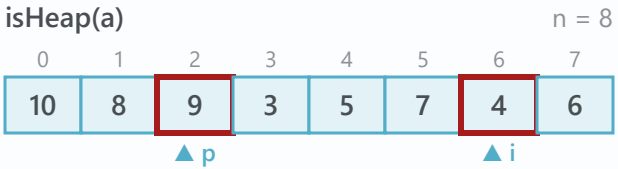
VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 5` `p = 2`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`





i = 6: parent p = (6 − 1) / 2 = 2. a[2] = 9 ≥ a[6] = 4 ✓ → continue.

Step 7 / 9

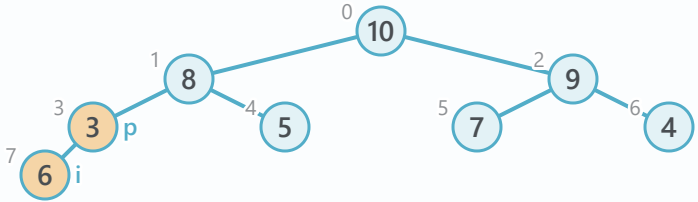
```
1 static boolean isHeap(int[] a) {  // max-heap?
▶   for (int i = 1; i < a.length; i++) {
▶     int p = (i - 1) / 2;           // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 6` `p = 2`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`



isHeap(a) false

0	1	2	3	4	5	6	7
10	8	9	3	5	7	4	6
			▲ p				▲ i

i = 7: p = 3. a[3] = 3 < a[7] = 6 ✗ → the child is larger than its parent: **return false**.

Step 8 / 9

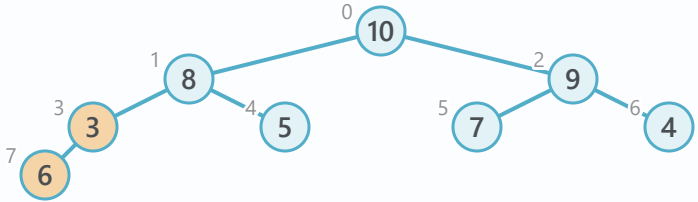
```
1 static boolean isHeap(int[] a) {  // max-heap?
2   for (int i = 1; i < a.length; i++) {
      int p = (i - 1) / 2;           // parent
      if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 7` `p = 3`

Code Trace: `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`



isHeap(a) false

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

Result: **false**. The first seven items form a valid heap, but 6 is larger than its parent 3. The worst case is n − 1 comparisons → **O(n)**.

Step 9 / 9

```
1 static boolean isHeap(int[] a) {  // max-heap?
2   for (int i = 1; i < a.length; i++) {
3     int p = (i - 1) / 2;            // parent
▶     if (a[p] < a[i]) return false; // out of order
5   }
6   return true;
7 }
```

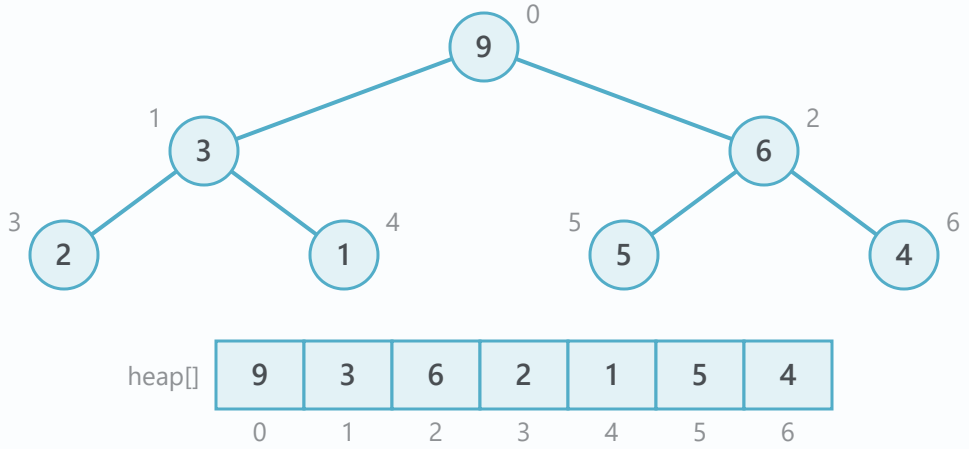
VARIABLES

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 7` `p = 3`

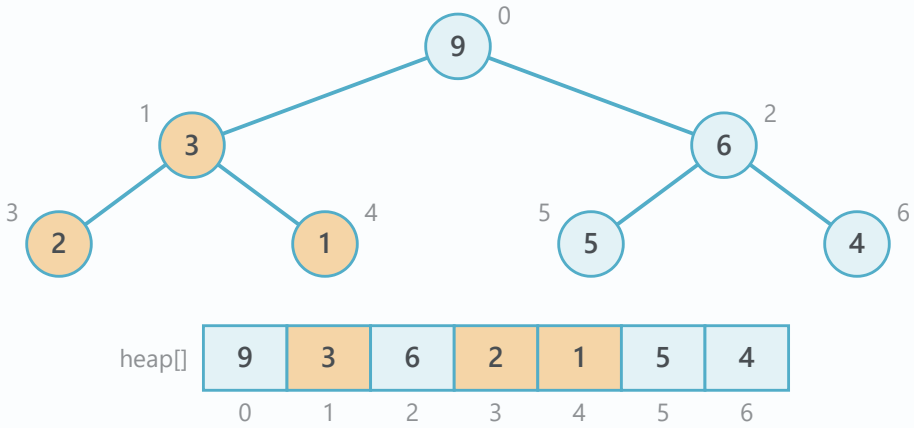
Array Representation

- Starting at the root, nodes are written into the array **level by level, left to right** (level order).
- Because the tree is complete, the array has no gaps and needs no pointers (references).



The small numbers next to the nodes are their array indices. In these slides indices start at **0**, so the root is heap[0].

Parent and Child Indices



Children

`left(i) = 2i + 1` · `right(i) = 2i + 2`

`i = 1 → 3 and 4` · `i = 2 → 5 and 6`

Parent

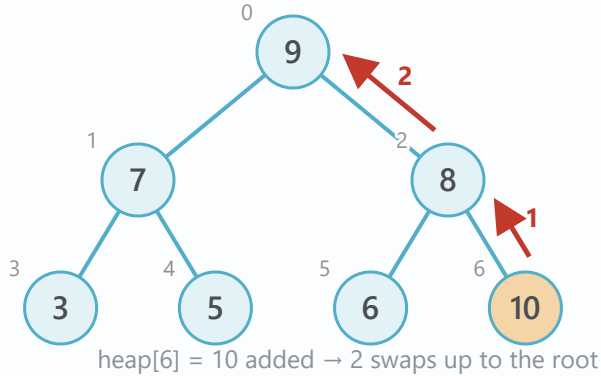
`parent(i) = ⌊(i − 1) / 2⌋`

`i = 6 → 2` · `i = 5 → 2` · `i = 4 → 1`

- Java's integer division already rounds down: `(i - 1) / 2`. A node has no children when 2i + 1 ≥ n.
- The 1-based alternative leaves heap[0] empty: children 2k and 2k + 1, parent ⌊k/2⌋. The two are equivalent; pick one and use it consistently.

Insertion: Up-Heap (Sift-Up)

1. Put the new item in the **first free slot** (heap[n]). The tree stays complete.
2. Heap order may now be broken: if the new item is larger than its parent (max-heap), swap them.
3. Keep moving the item **up** until it reaches the root or its parent is greater than or equal to it.



heap[6] = 10 added → 2 swaps up to the root

Cost

At most $h = \lfloor \log_2 n \rfloor$ swaps → **O(log n)**.

Why it works

The parent that moves down in a swap was already larger than its other child.

Names

up-heap, bubble-up, sift-up, swim; the code in this chapter calls it `swim`.

Inserting into a Max-Heap: Java Code

```
public class MaxHeap {
  private int[] heap = new int[4];
  private int n = 0;                 // number of items

public void insert(int x) {
  if (n == heap.length) resize(2 * heap.length);
  heap[n] = x;  n++;               // first free slot
  swim(n - 1);
}
private void swim(int i) {         // up-heap
  while (i > 0) {
    int parent = (i - 1) / 2;
    if (heap[parent] >= heap[i]) return;
    swap(i, parent);  i = parent;
  }
}
private void swap(int a, int b) {
  int t = heap[a];  heap[a] = heap[b];  heap[b] = t;
}
}  // MaxHeap
```

resize

When the array fills up, its capacity doubles (amortized O(1)).

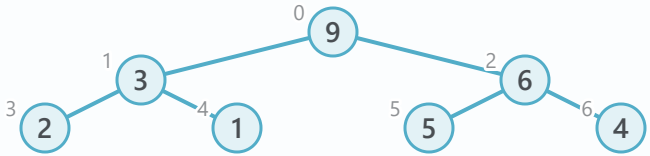
swap

Exchanges heap[a] and heap[b].

Stopping

At the root (i = 0) or once parent ≥ item.

Code Trace: `insert(10)`, `insert(8)`



start — n = 7 · heap.length = 16

0	1	2	3	4	5	6	7	8
9	3	6	2	1	5	4		

Initial max-heap: every parent ≥ its children, and the largest (9) is at the root. Capacity is 16, so there is room.

Step 1 / 14

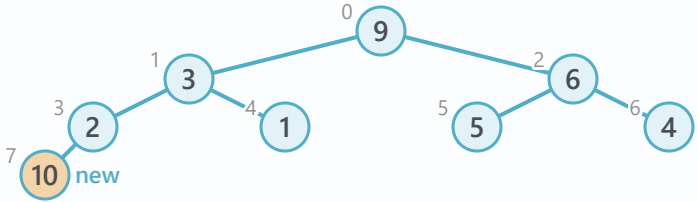
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

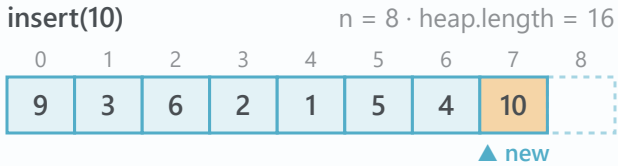
VARIABLES

`n = 7` `heap.length = 16`

`heap = [9, 3, 6, 2, 1, 5, 4]`

Code Trace: `insert(10)`, `insert(8)`





insert(10): 10 goes into the first free slot, **heap[7]**; n = 8. The tree stays complete, but heap order may break.

Step 2 / 14

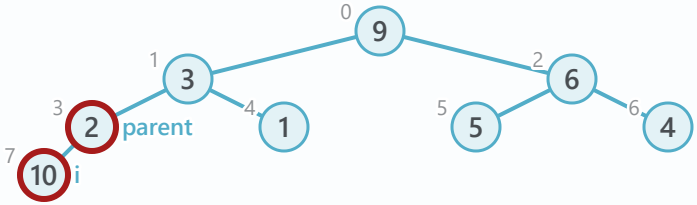
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

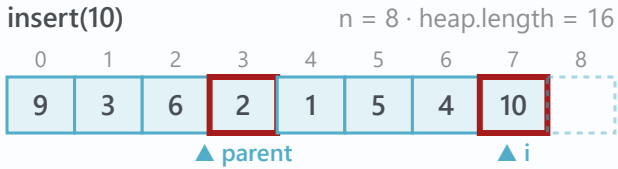
VARIABLES

`x = 10` `n = 8` `heap.length = 16`

`heap = [9, 3, 6, 2, 1, 5, 4, 10]`

Code Trace: `insert(10)`, `insert(8)`





i = 7 > 0 → **parent = (7 − 1) / 2 = 3**. Compare: heap[3] = 2 < heap[7] = 10.

Step 3 / 14

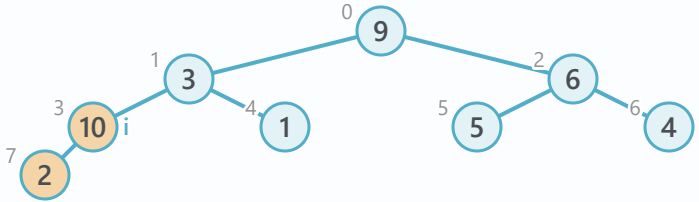
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

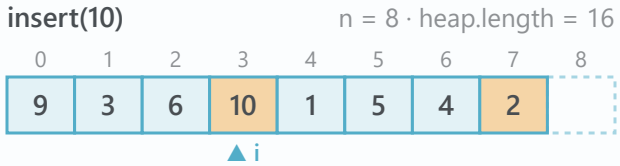
VARIABLES

`i = 7` `parent = 3` `n = 8`

`heap = [9, 3, 6, 2, 1, 5, 4, 10]`

Code Trace: `insert(10)`, `insert(8)`





10 > 2 → **swap** (index 7 ↔ 3); the item moves up one level, **i = 3**.

Step 4 / 14

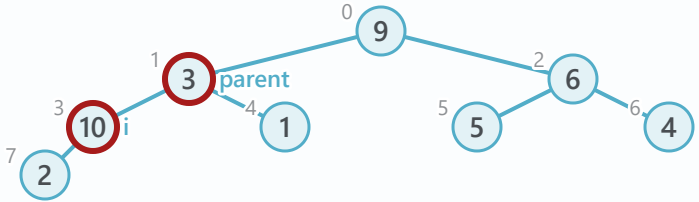
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

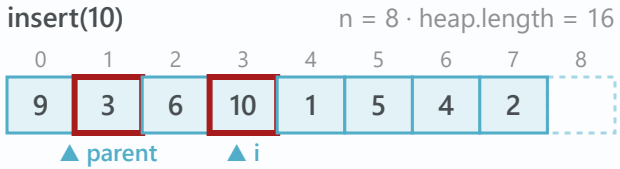
VARIABLES

`i = 3` `parent = 3` `n = 8`

`heap = [9, 3, 6, 10, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`





i = 3 > 0 → **parent = (3 − 1) / 2 = 1**. Compare: heap[1] = 3 < heap[3] = 10.

Step 5 / 14

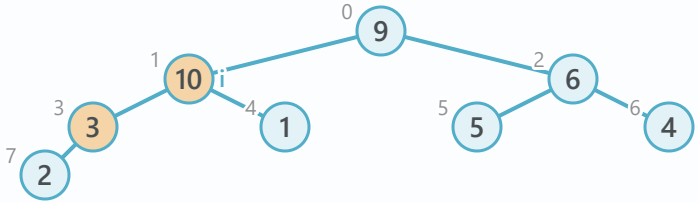
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                 // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {           // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 3` `parent = 1` `n = 8`

`heap = [9, 3, 6, 10, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`



insert(10) n = 8 · heap.length = 16

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

▲ i

10 > 3 → **swap** (index 3 ↔ 1); the item moves up one level, **i = 1**.

Step 6 / 14

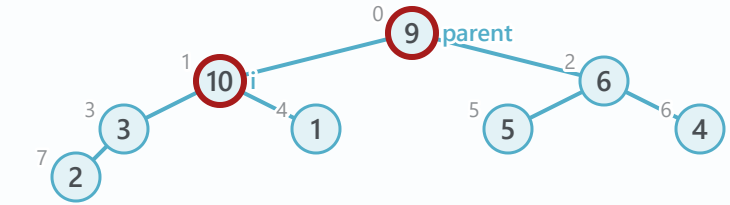
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

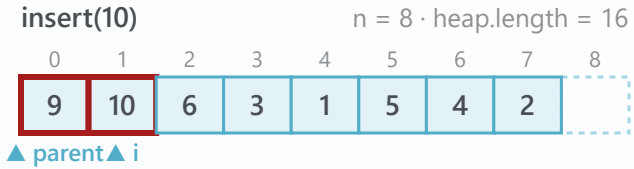
VARIABLES

`i = 1` `parent = 1` `n = 8`

`heap = [9, 10, 6, 3, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`





i = 1 > 0 → **parent = (1 − 1) / 2 = 0**. Compare: heap[0] = 9 < heap[1] = 10.

Step 7 / 14

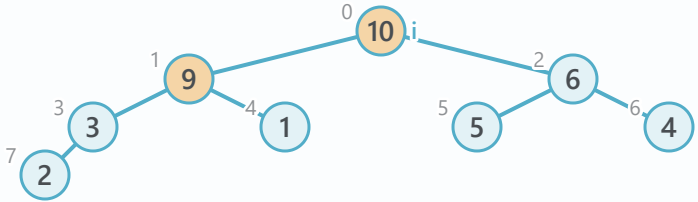
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 1` `parent = 0` `n = 8`

`heap = [9, 10, 6, 3, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`



insert(10) n = 8 · heap.length = 16

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

▲ i

10 > 9 → **swap** (index 1 ↔ 0); the item moves up one level, **i = 0**.

Step 8 / 14

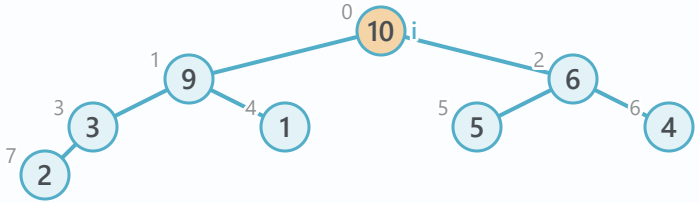
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

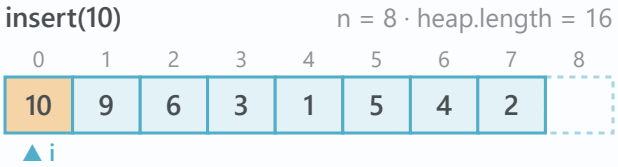
VARIABLES

`i = 0` `parent = 0` `n = 8`

`heap = [10, 9, 6, 3, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`





i = 0: the item is at the **root**; the loop test (i > 0) fails and swim ends. 10 is in place.

Step 9 / 14

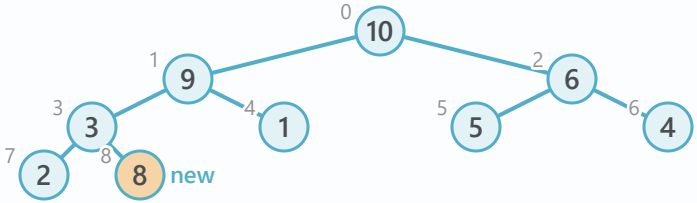
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

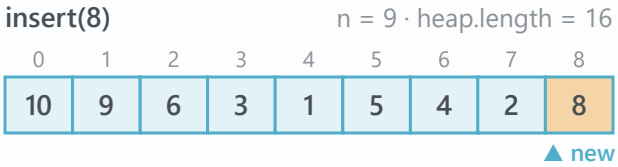
VARIABLES

`i = 0` `n = 8`

`heap = [10, 9, 6, 3, 1, 5, 4, 2]`

Code Trace: `insert(10)`, `insert(8)`





insert(8): 8 goes into the first free slot, **heap[8]**; n = 9. The tree stays complete, but heap order may break.

Step 10 / 14

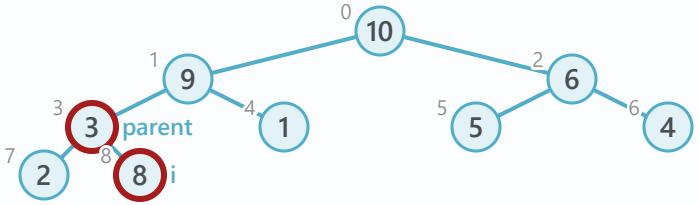
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

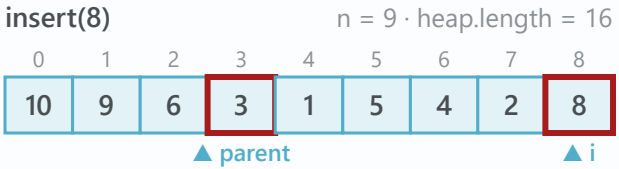
VARIABLES

`x = 8` `n = 9` `heap.length = 16`

`heap = [10, 9, 6, 3, 1, 5, 4, 2, 8]`

Code Trace: `insert(10)`, `insert(8)`





i = 8 > 0 → **parent = (8 − 1) / 2 = 3**. Compare: heap[3] = 3 < heap[8] = 8.

Step 11 / 14

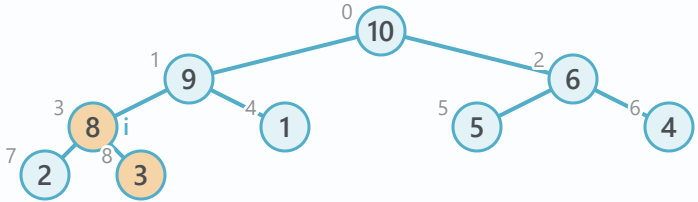
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 8` `parent = 3` `n = 9`

`heap = [10, 9, 6, 3, 1, 5, 4, 2, 8]`

Code Trace: `insert(10)`, `insert(8)`



insert(8) n = 9 · heap.length = 16

0	1	2	3	4	5	6	7	8
10	9	6	8	1	5	4	2	3
			▲ i					

8 > 3 → **swap** (index 8 ↔ 3); the item moves up one level, **i = 3**.

Step 12 / 14

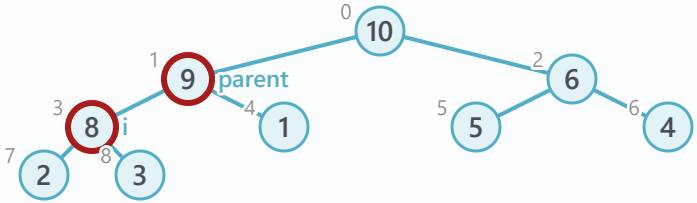
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

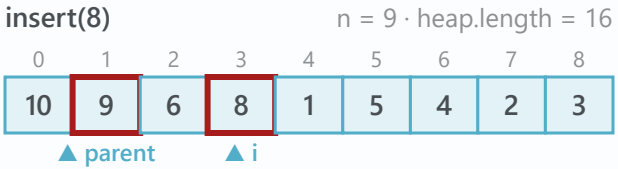
VARIABLES

`i = 3` `parent = 3` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

Code Trace: `insert(10)`, `insert(8)`





i = 3 > 0 → **parent = (3 − 1) / 2 = 1**. Compare: heap[1] = 9 ≥ heap[3] = 8.

Step 13 / 14

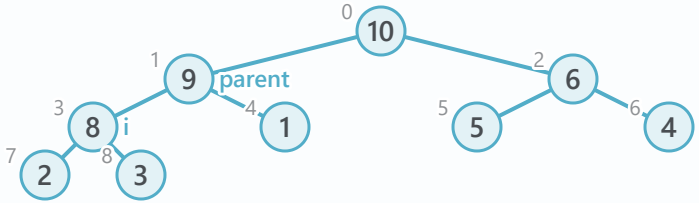
```
1 public void insert(int x) {
2   if (n == heap.length) resize(2 * heap.length);
3   heap[n] = x;  n++;                // first free slot
4   swim(n - 1);
5 }
6 private void swim(int i) {          // up-heap
▶   while (i > 0) {
▶     int parent = (i - 1) / 2;
9     if (heap[parent] >= heap[i]) return;
10    swap(i, parent);  i = parent;
11  }
12 }
```

VARIABLES

`i = 3` `parent = 1` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

Code Trace: `insert(10)`, `insert(8)`



insert(8) n = 9 · heap.length = 16

0	1	2	3	4	5	6	7	8
10	9	6	8	1	5	4	2	3
	▲ parent		▲ i					

9 ≥ 8: the parent is already ≥ the item → heap order holds, **return**. 8 is in place.

Step 14 / 14

```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
▶      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

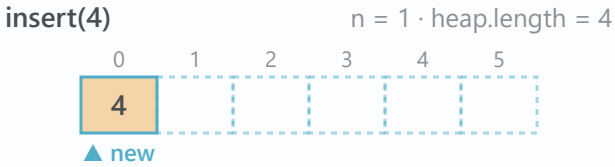
VARIABLES

`i = 3` `parent = 1` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





insert(4): 4 goes into the first free slot, **heap[0]**; n = 1. The tree stays complete, but heap order may break.

Step 1 / 15

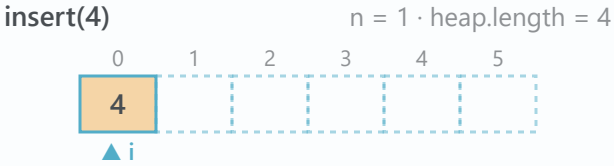
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`x = 4` `n = 1` `heap.length = 4` `heap = [4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 0: the item is at the **root**; the loop test (i > 0) fails and swim ends. 4 is in place.

Step 2 / 15

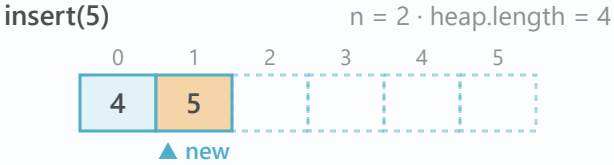
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;   n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);   i = parent;
11   }
12 }
```

VARIABLES

`i = 0` `n = 1` `heap = [4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





insert(5): 5 goes into the first free slot, **heap[1]**; n = 2. The tree stays complete, but heap order may break.

Step 3 / 15

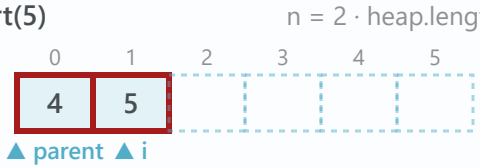
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`x = 5` `n = 2` `heap.length = 4` `heap = [4, 5]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 1 > 0 → **parent = (1 − 1) / 2 = 0**. Compare: heap[0] = 4 < heap[1] = 5.

Step 4 / 15

```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

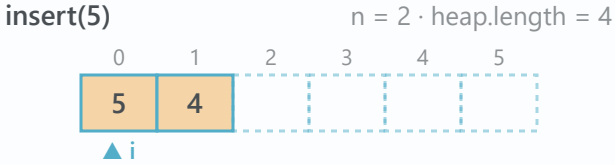
VARIABLES

`i = 1` `parent = 0` `n = 2`

`heap = [4, 5]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





5 > 4 → **swap** (index 1 ↔ 0); the item moves up one level, **i = 0**.

Step 5 / 15

```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

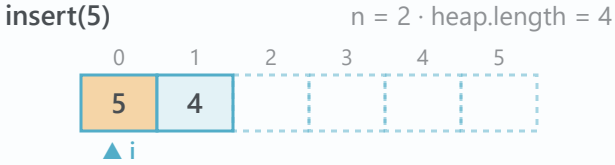
VARIABLES

`i = 0` `parent = 0` `n = 2`

`heap = [5, 4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 0: the item is at the **root**; the loop test (i > 0) fails and swim ends. 5 is in place.

Step 6 / 15

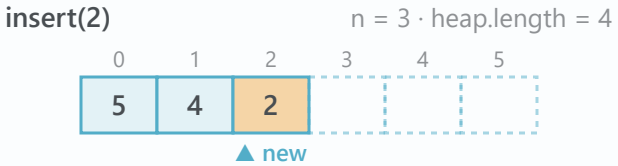
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 0` `n = 2` `heap = [5, 4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





insert(2): 2 goes into the first free slot, **heap[2]**; n = 3. The tree stays complete, but heap order may break.

Step 7 / 15

```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

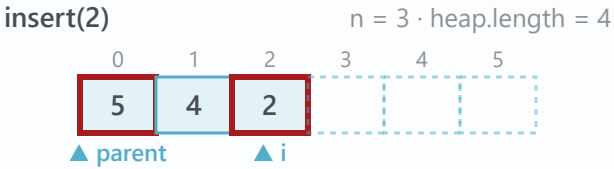
VARIABLES

`x = 2` `n = 3` `heap.length = 4`

`heap = [5, 4, 2]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 2 > 0 → **parent = (2 − 1) / 2 = 0**. Compare: heap[0] = 5 ≥ heap[2] = 2.

Step 8 / 15

```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

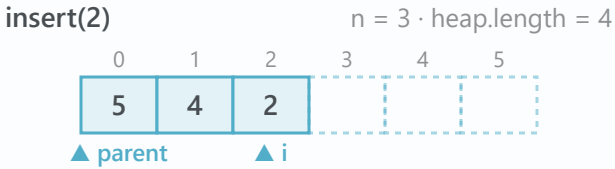
VARIABLES

i = 2 · parent = 0 · n = 3

heap = [5, 4, 2]

Code Trace: inserting 4, 5, 2, 6 into an empty heap





5 ≥ 2: the parent is already ≥ the item → heap order holds, **return**. 2 is in place.

Step 9 / 15

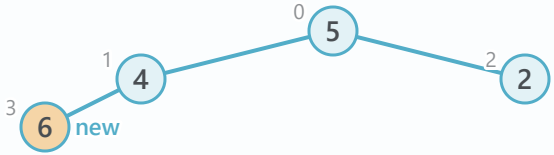
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
▶      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

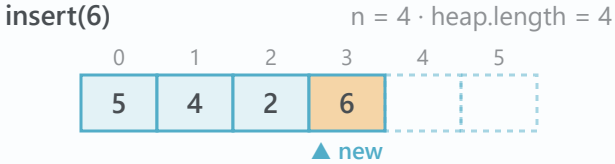
VARIABLES

`i = 2` `parent = 0` `n = 3`

`heap = [5, 4, 2]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





insert(6): 6 goes into the first free slot, **heap[3]**; n = 4. The tree stays complete, but heap order may break.

Step 10 / 15

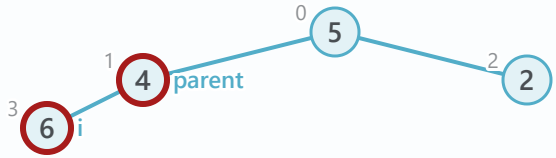
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

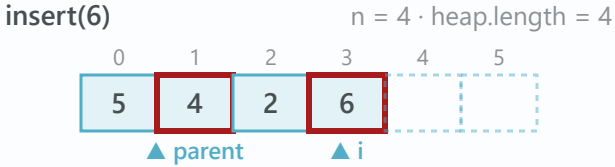
VARIABLES

`x = 6` `n = 4` `heap.length = 4`

`heap = [5, 4, 2, 6]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 3 > 0 → **parent = (3 − 1) / 2 = 1**. Compare: heap[1] = 4 < heap[3] = 6.

Step 11 / 15

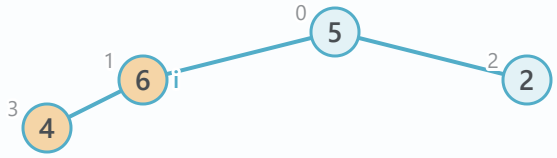
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

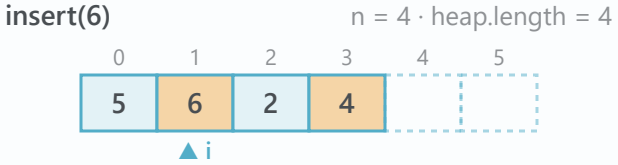
VARIABLES

`i = 3` `parent = 1` `n = 4`

`heap = [5, 4, 2, 6]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





6 > 4 → **swap** (index 3 ↔ 1); the item moves up one level, **i = 1**.

Step 12 / 15

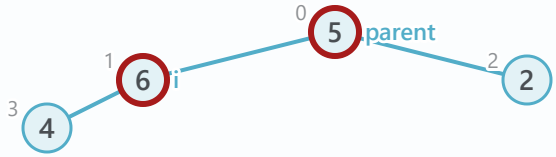
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

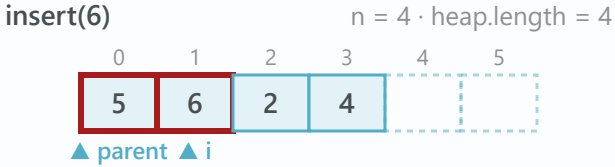
VARIABLES

`i = 1` `parent = 1` `n = 4`

`heap = [5, 6, 2, 4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap





i = 1 > 0 → **parent = (1 − 1) / 2 = 0**. Compare: heap[0] = 5 < heap[1] = 6.

Step 13 / 15

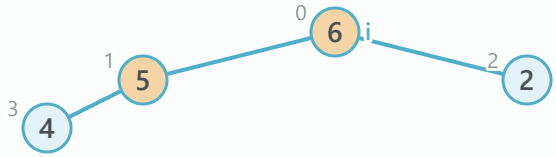
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

i = 1 | parent = 0 | n = 4

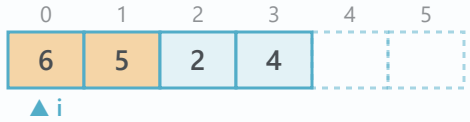
heap = [5, 6, 2, 4]

Code Trace: inserting 4, 5, 2, 6 into an empty heap



insert(6)

n = 4 · heap.length = 4



6 > 5 → **swap** (index 1 ↔ 0); the item moves up one level, **i = 0**.

Step 14 / 15

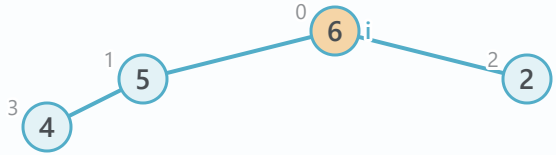
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 0` `parent = 0` `n = 4`

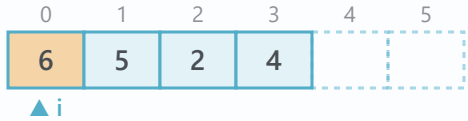
`heap = [6, 5, 2, 4]`

Code Trace: inserting 4, 5, 2, 6 into an empty heap



insert(6)

n = 4 · heap.length = 4



i = 0: the item is at the **root**; the loop test (i > 0) fails and swim ends. 6 is in place.

Step 15 / 15

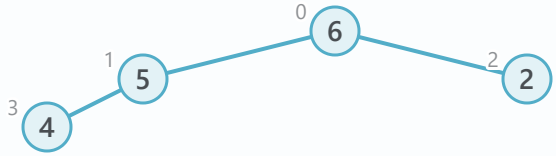
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 0` `n = 4`

`heap = [6, 5, 2, 4]`

Code Trace (continued): insert(1), insert(3)



state

n = 4 · heap.length = 4

0	1	2	3	4	5
6	5	2	4		

Start: heap = [6, 5, 2, 4], n = 4 and capacity 4 → the array is **completely full**.

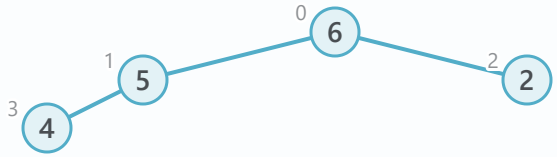
Step 1 / 10

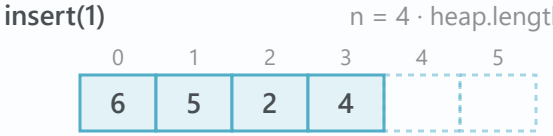
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`n = 4` `heap.length = 4` `heap = [6, 5, 2, 4]`

Code Trace (continued): insert(1), insert(3)





insert(1): n == heap.length (4) → the array is full. **resize(8)** doubles the capacity and copies the items into a new array.

Step 2 / 10

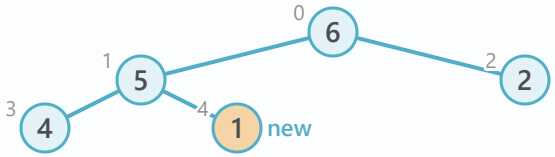
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

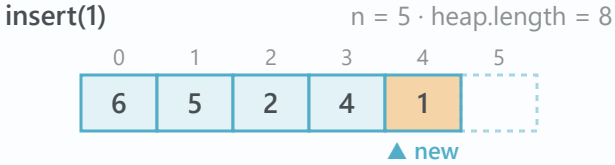
VARIABLES

x = 1 | n = 4 | heap.length = 8

heap = [6, 5, 2, 4]

Code Trace (continued): insert(1), insert(3)





1 goes into the first free slot, **heap[4]**; n = 5. The tree stays complete, but heap order may break.

Step 3 / 10

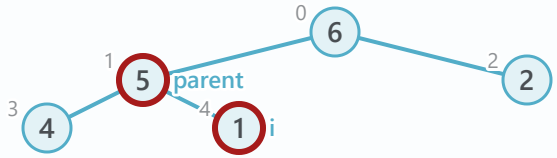
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

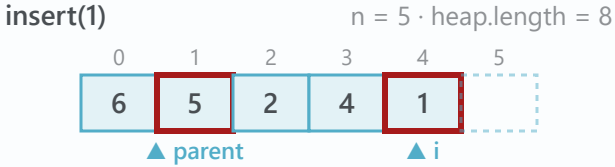
VARIABLES

`x = 1` `n = 5` `heap.length = 8`

`heap = [6, 5, 2, 4, 1]`

Code Trace (continued): insert(1), insert(3)





i = 4 > 0 → **parent = (4 − 1) / 2 = 1**. Compare: heap[1] = 5 ≥ heap[4] = 1.

Step 4 / 10

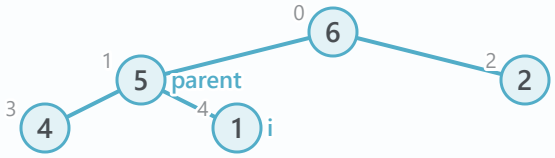
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;   n++;                // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {          // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);   i = parent;
11   }
12 }
```

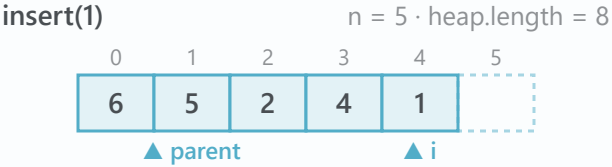
VARIABLES

i = 4 | parent = 1 | n = 5

heap = [6, 5, 2, 4, 1]

Code Trace (continued): insert(1), insert(3)





5 ≥ 1: the parent is already ≥ the item → heap order holds, **return**. 1 is in place.

Step 5 / 10

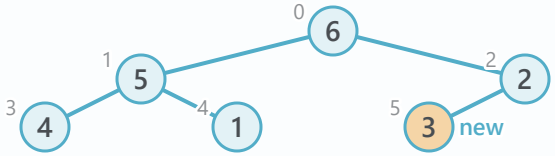
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
▶      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

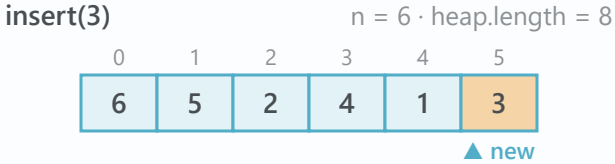
VARIABLES

`i = 4` `parent = 1` `n = 5`

`heap = [6, 5, 2, 4, 1]`

Code Trace (continued): insert(1), insert(3)





insert(3): 3 goes into the first free slot, **heap[5]**; n = 6. The tree stays complete, but heap order may break.

Step 6 / 10

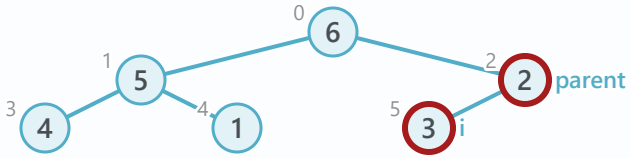
```
1  public void insert(int x) {
▶    if (n == heap.length) resize(2 * heap.length);
▶    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

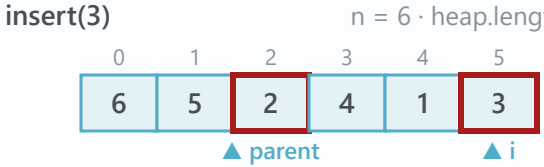
VARIABLES

`x = 3` `n = 6` `heap.length = 8`

`heap = [6, 5, 2, 4, 1, 3]`

Code Trace (continued): insert(1), insert(3)





i = 5 > 0 → **parent = (5 − 1) / 2 = 2**. Compare: heap[2] = 2 < heap[5] = 3.

Step 7 / 10

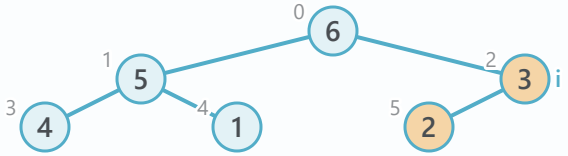
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

i = 5 | parent = 2 | n = 6

heap = [6, 5, 2, 4, 1, 3]

Code Trace (continued): insert(1), insert(3)



insert(3) n = 6 · heap.length = 8

0	1	2	3	4	5
6	5	3	4	1	2
		▲ i			

3 > 2 → **swap** (index 5 ↔ 2); the item moves up one level, **i = 2**.

Step 8 / 10

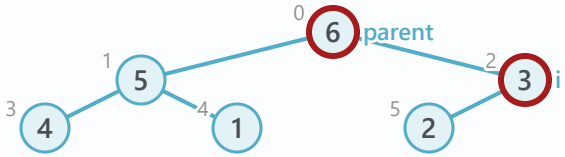
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
11   }
12 }
```

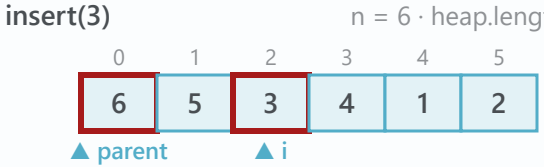
VARIABLES

i = 2 | parent = 2 | n = 6

heap = [6, 5, 3, 4, 1, 2]

Code Trace (continued): insert(1), insert(3)





i = 2 > 0 → **parent = (2 − 1) / 2 = 0**. Compare: heap[0] = 6 ≥ heap[2] = 3.

Step 9 / 10

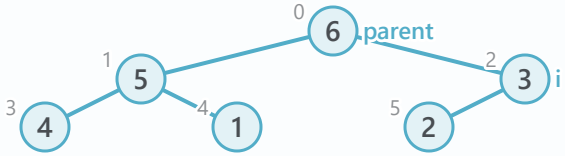
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
9      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

i = 2 | parent = 0 | n = 6

heap = [6, 5, 3, 4, 1, 2]

Code Trace (continued): insert(1), insert(3)



insert(3) n = 6 · heap.length = 8

0	1	2	3	4	5
6	5	3	4	1	2
▲ parent		▲ i			

6 ≥ 3: the parent is already ≥ the item → heap order holds, **return**. 3 is in place.

Step 10 / 10

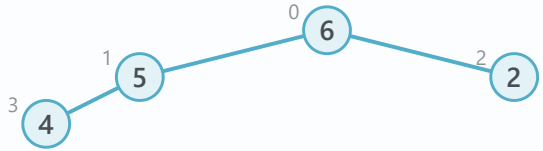
```
1  public void insert(int x) {
2    if (n == heap.length) resize(2 * heap.length);
3    heap[n] = x;  n++;              // first free slot
4    swim(n - 1);
5  }
6  private void swim(int i) {        // up-heap
7    while (i > 0) {
8      int parent = (i - 1) / 2;
▶      if (heap[parent] >= heap[i]) return;
10     swap(i, parent);  i = parent;
11   }
12 }
```

VARIABLES

`i = 2` `parent = 0` `n = 6`

`heap = [6, 5, 3, 4, 1, 2]`

Code Trace: `resize(8)`



resize(8)

heap	6	5	2	4

insert(1) is called with n == heap.length = 4 → **resize(8)**. An array's length is fixed, so growing means a new array plus copying.

Step 1 / 9

```
▶ private void resize(int capacity) {
2   int[] newArr = new int[capacity];
3   for (int k = 0; k < n; k++)
4     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4`

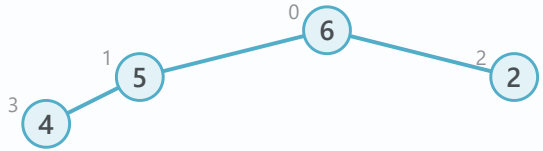
`heap = [6, 5, 2, 4]`

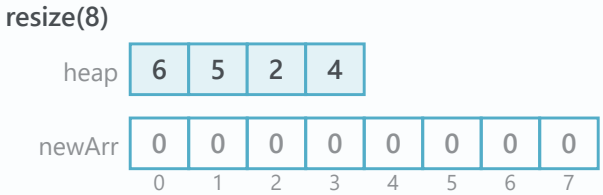
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





A new 8-cell array, **newArr**, is allocated; Java fills an int array with 0s.

Step 2 / 9

```
1 private void resize(int capacity) {
▶   int[] newArr = new int[capacity];
3   for (int k = 0; k < n; k++)
4     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4`

`heap = [6, 5, 2, 4]`

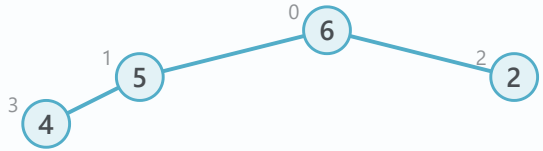
`newArr = [0, 0, 0, 0, 0, 0, 0, 0]`

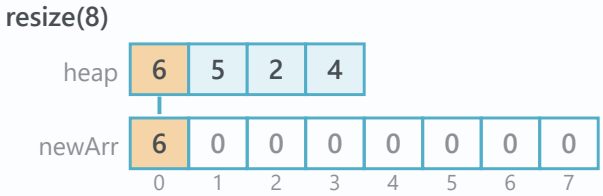
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





k = 0 < n → **newArr[0] = heap[0] = 6**.

Step 3 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
▶   for (int k = 0; k < n; k++)
▶     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4` `k = 0`

`heap = [6, 5, 2, 4]`

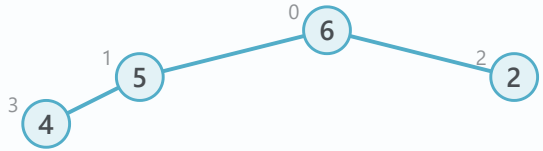
`newArr = [6, 0, 0, 0, 0, 0, 0, 0]`

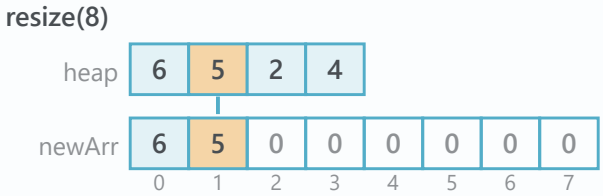
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





k = 1 < n → **newArr[1] = heap[1] = 5**.

Step 4 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
▶   for (int k = 0; k < n; k++)
▶     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4` `k = 1`

`heap = [6, 5, 2, 4]`

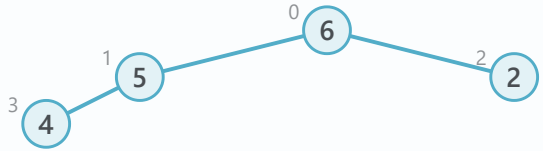
`newArr = [6, 5, 0, 0, 0, 0, 0, 0]`

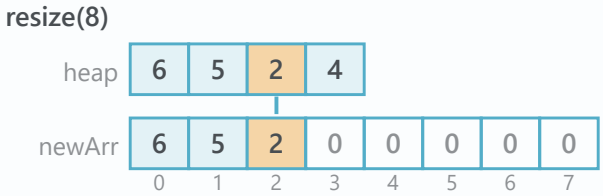
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





k = 2 < n → **newArr[2] = heap[2] = 2**.

Step 5 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
▶   for (int k = 0; k < n; k++)
▶     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4` `k = 2`

`heap = [6, 5, 2, 4]`

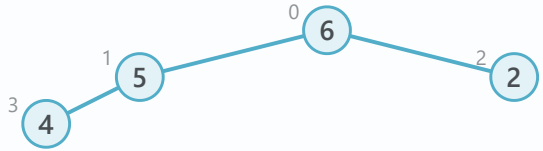
`newArr = [6, 5, 2, 0, 0, 0, 0, 0]`

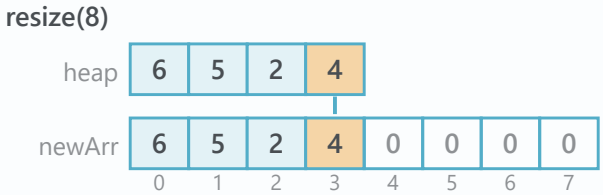
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





k = 3 < n → **newArr[3] = heap[3] = 4**.

Step 6 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
▶   for (int k = 0; k < n; k++)
▶     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4` `k = 3`

`heap = [6, 5, 2, 4]`

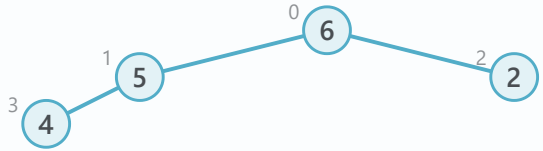
`newArr = [6, 5, 2, 4, 0, 0, 0, 0]`

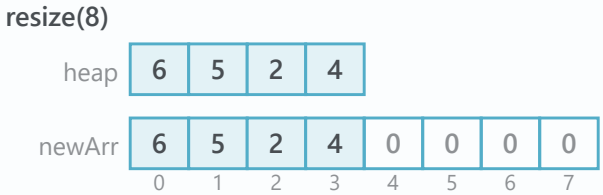
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





k = 4 = n → the loop ends. n items were copied, so this resize costs **O(n)**.

Step 7 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
▶   for (int k = 0; k < n; k++)
4     newArr[k] = heap[k];          // copy
5   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4` `k = 4`

`heap = [6, 5, 2, 4]`

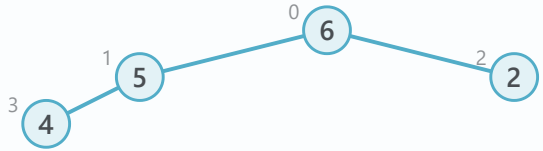
`newArr = [6, 5, 2, 4, 0, 0, 0, 0]`

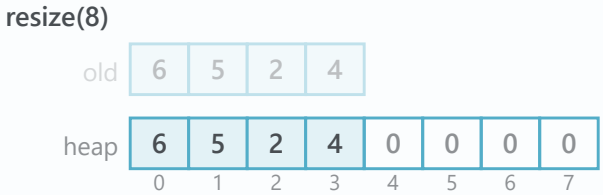
CALL STACK

`resize(8)`

`insert(1)`

Code Trace: `resize(8)`





heap = newArr: the field now refers to the new array, and the old one is left to the garbage collector. Same tree; only the capacity is now 8.

Step 8 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
3   for (int k = 0; k < n; k++)
4     newArr[k] = heap[k];          // copy
▶   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

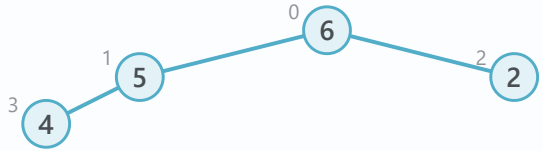
`capacity = 8` `n = 4`

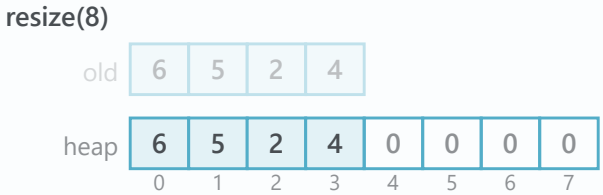
`heap = [6, 5, 2, 4, 0, 0, 0, 0]`

CALL STACK

- **`resize(8)`**
- `insert(1)`

Code Trace: `resize(8)`





resize returns and insert(1) continues. Because the capacity **doubles** each time, n insertions copy fewer than 2n items in total → **amortized O(1)** per insertion.

Step 9 / 9

```
1 private void resize(int capacity) {
2   int[] newArr = new int[capacity];
3   for (int k = 0; k < n; k++)
4     newArr[k] = heap[k];          // copy
▶   heap = newArr;                  // old array becomes
  garbage
6 }
```

VARIABLES

`capacity = 8` `n = 4`

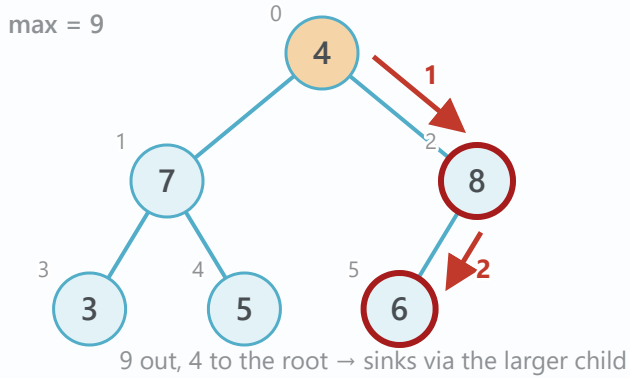
`heap = [6, 5, 2, 4, 0, 0, 0, 0]`

CALL STACK

`insert(1)`

Removing the Maximum: Down-Heap (Sift-Down)

1. Save the item at the root (the largest); it is the return value.
2. Move the **last item** (heap[n − 1]) to the root and decrease n by one.
3. Swap the root with its **larger child**.
4. Keep moving the item **down** until it reaches a leaf or is greater than or equal to both children.



9 out, 4 to the root → sinks via the larger child

Why the larger child?

If the smaller child moved up, it would become the parent of a larger sibling. In a min-heap, pick the *smaller* child instead.

Cost

Two comparisons per level, at most h levels → **O(log n)**.

Removing the Maximum: Java Code

```
public int removeMax() {          // requires n > 0
  int max = heap[0];
  swap(0, n - 1);  n--;           // last item to the root
  sink(0);
  return max;
}
private void sink(int i) {          // down-heap
  while (2 * i + 1 < n) {         // has a left child?
    int j = 2 * i + 1;              // j: larger child
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    swap(i, j);  i = j;             // down one level
  }
}
```

Empty heap

Real code first throws an exception `if (n == 0)`.

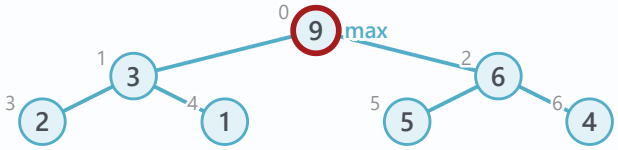
shrink

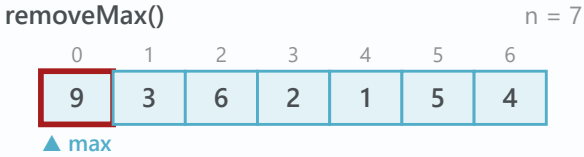
Optional: halve the array once n drops to ¼ of the capacity.

Swap

The maximum stays at the end of the array, which is what heap sort relies on.

Code Trace: `removeMax()` and `sink`





removeMax(): the largest item is always at the root → **max = heap[0] = 9**.

Step 1 / 8

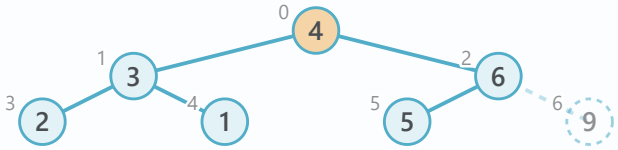
```
1  public int removeMax() {          // requires n > 0
▶    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

VARIABLES

`max = 9` `n = 7`

`heap = [9, 3, 6, 2, 1, 5, 4]`

Code Trace: `removeMax()` and `sink`





The root and the last item swap (0 ↔ 6); **n = 6**, so 9 leaves the heap (faded). The tree is still complete; the new root 4 may break the order.

Step 2 / 8

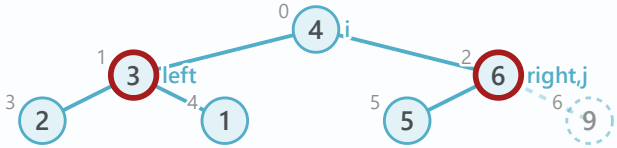
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
▶    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

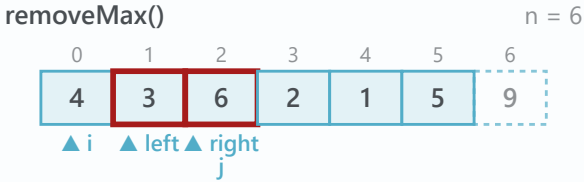
VARIABLES

`max = 9` `n = 6`

`heap = [4, 3, 6, 2, 1, 5 | 9]`

Code Trace: `removeMax()` and `sink`





Left child 1 (3), right child 2 (6) → the larger is **6**, **j = 2**.

Step 3 / 8

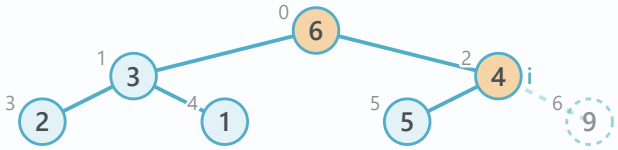
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
▶    sink(0);
5    return max;
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;            // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;           // down one level
13   }
14 }
```

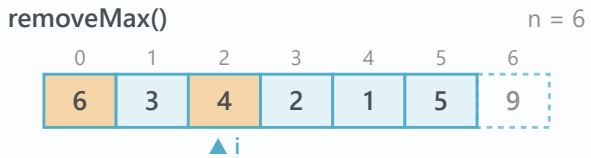
VARIABLES

`max = 9` `i = 0` `j = 2` `n = 6`

`heap = [4, 3, 6, 2, 1, 5 | 9]`

Code Trace: `removeMax()` and `sink`





4 < 6 → **swap** (index 0 ↔ 2); the larger child rises, **i = 2**.

Step 4 / 8

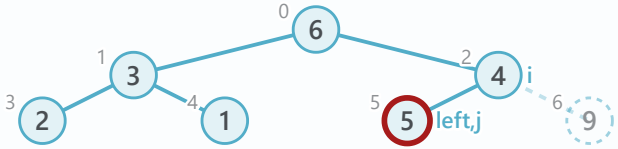
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

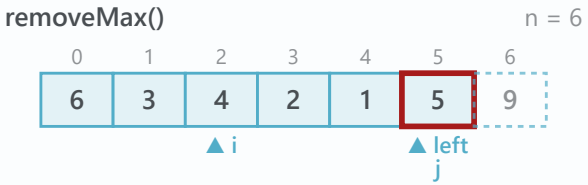
VARIABLES

`max = 9` `i = 2` `j = 2` `n = 6`

`heap = [6, 3, 4, 2, 1, 5 | 9]`

Code Trace: `removeMax()` and `sink`





Only a left child: index 5 (5) → **j = 5**.

Step 5 / 8

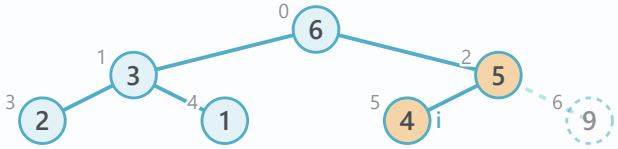
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;              // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

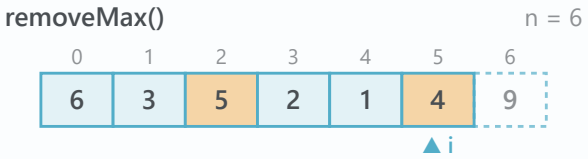
VARIABLES

`max = 9` `i = 2` `j = 5` `n = 6`

`heap = [6, 3, 4, 2, 1, 5 | 9]`

Code Trace: `removeMax()` and `sink`





4 < 5 → **swap** (index 2 ↔ 5); the larger child rises, **i = 5**.

Step 6 / 8

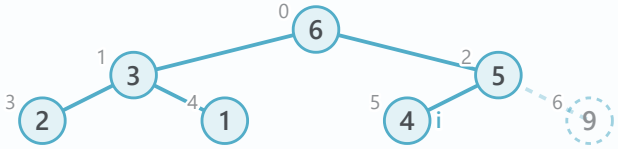
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

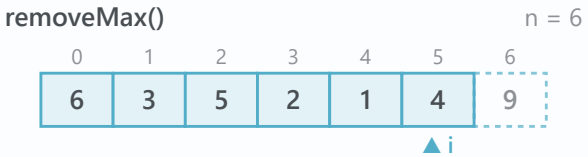
VARIABLES

`max = 9` `i = 5` `j = 5` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

Code Trace: `removeMax()` and `sink`





i = 5: 2·5 + 1 = 11 ≥ n = 6 → no children (a leaf), so the loop ends.

Step 7 / 8

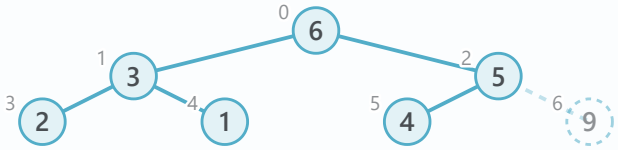
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;             // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;            // down one level
13   }
14 }
```

VARIABLES

`max = 9` `i = 5` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

Code Trace: `removeMax()` and `sink`





sink is done, and the array is a valid max-heap again. **return 9**. The path spans at most $\lfloor \log_2 n \rfloor$ levels → **O(log n)**.

Step 8 / 8

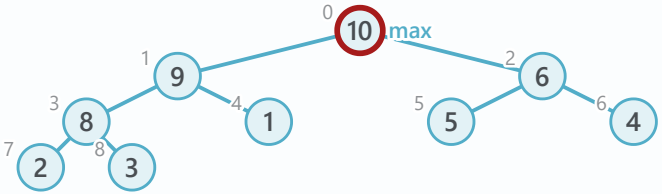
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
▶    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

VARIABLES

`max = 9` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

Code Trace: two `removeMax()` calls in a row



removeMax() #1 n = 9

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

▲ max

removeMax() #1: the largest item is always at the root → **max = heap[0] = 10**.

Step 1 / 17

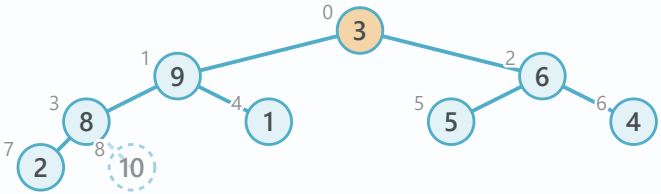
```
1  public int removeMax() {          // requires n > 0
▶    int max = heap[0];
3    swap(0, n - 1);  n--;          // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

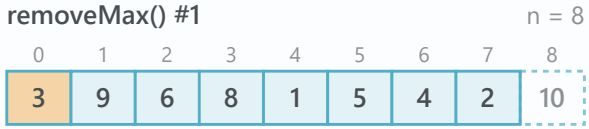
VARIABLES

`max = 10` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

Code Trace: two `removeMax()` calls in a row





The root and the last item swap (0 ↔ 8); **n = 8**, so 10 leaves the heap (faded). The tree is still complete; the new root 3 may break the order.

Step 2 / 17

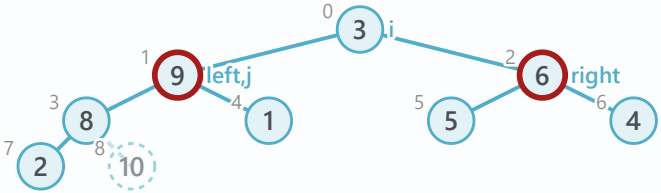
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
▶    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

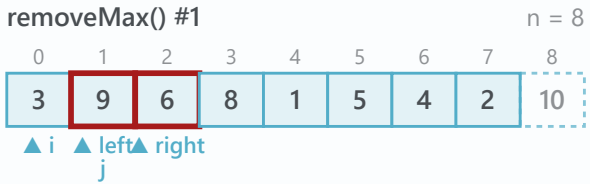
VARIABLES

`max = 10` `n = 8`

`heap = [3, 9, 6, 8, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





Left child 1 (9), right child 2 (6) → the larger is **9**, **j = 1**.

Step 3 / 17

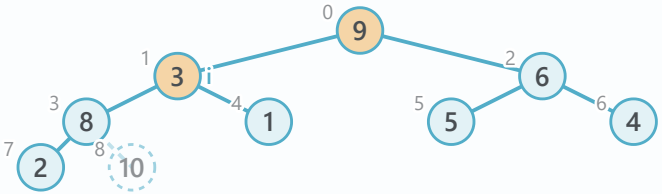
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
▶    sink(0);
5    return max;
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;            // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;           // down one level
13   }
14 }
```

VARIABLES

`max = 10` `i = 0` `j = 1` `n = 8`

`heap = [3, 9, 6, 8, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row



removeMax() #1 n = 8

0	1	2	3	4	5	6	7	8
9	3	6	8	1	5	4	2	10
	▲ i							

3 < 9 → **swap** (index 0 ↔ 1); the larger child rises, **i = 1**.

Step 4 / 17

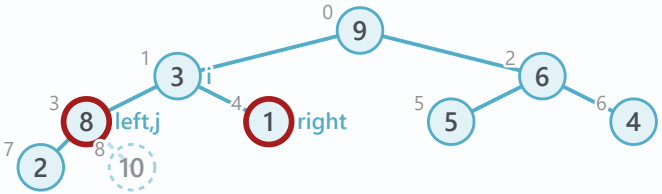
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

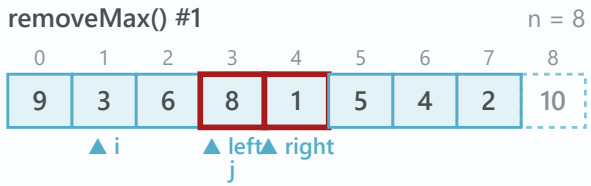
VARIABLES

`max = 10` `i = 1` `j = 1` `n = 8`

`heap = [9, 3, 6, 8, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





Left child 3 (8), right child 4 (1) → the larger is **8**, **j = 3**.

Step 5 / 17

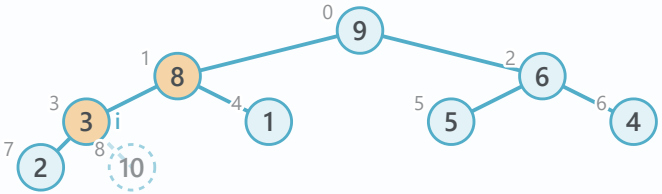
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;              // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

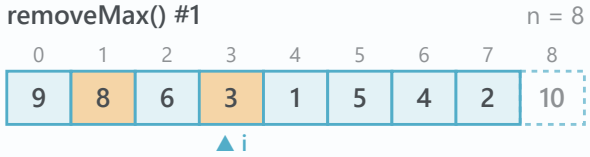
VARIABLES

`max = 10` `i = 1` `j = 3` `n = 8`

`heap = [9, 3, 6, 8, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





3 < 8 → **swap** (index 1 ↔ 3); the larger child rises, **i = 3**.

Step 6 / 17

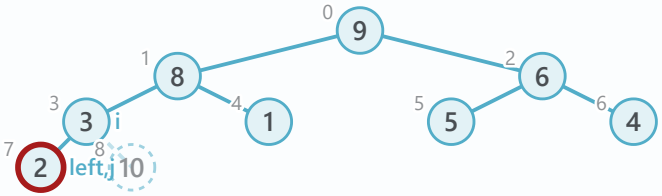
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

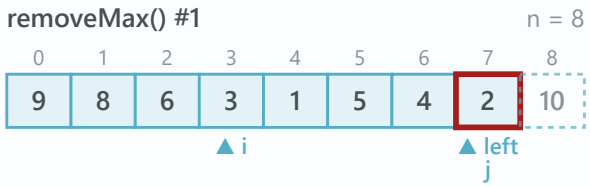
VARIABLES

`max = 10` `i = 3` `j = 3` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





Only a left child: index 7 (2) → **j = 7**.

Step 7 / 17

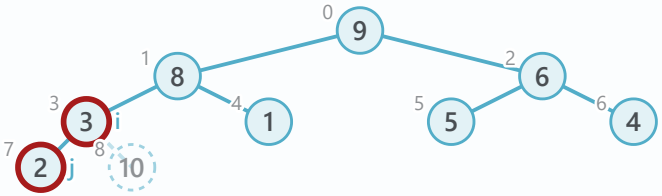
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;              // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

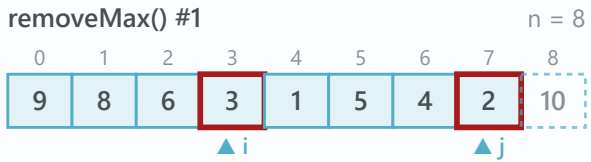
VARIABLES

`max = 10` `i = 3` `j = 7` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





heap[3] = 3 ≥ heap[7] = 2 → no smaller than either child, so heap order holds: **return**.

Step 8 / 17

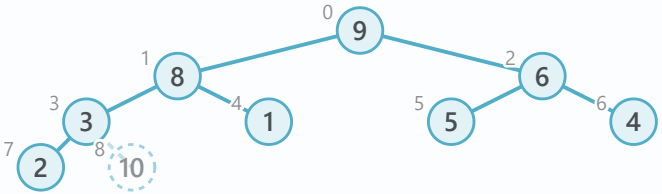
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
▶      if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

VARIABLES

`max = 10` `i = 3` `j = 7` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row



removeMax() #1 n = 8

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

sink is done, and the array is a valid max-heap again. **return 10**. The path spans at most $\lfloor \log_2 n \rfloor$ levels → **O(log n)**.

Step 9 / 17

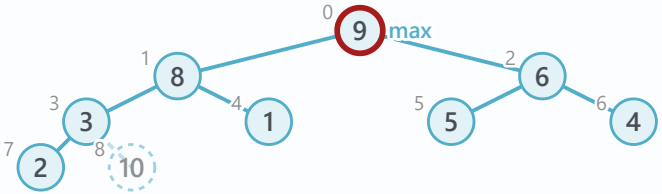
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
▶    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

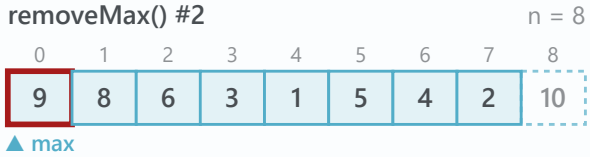
VARIABLES

`max = 10` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





removeMax() #2: the largest item is always at the root → **max = heap[0] = 9**.

Step 10 / 17

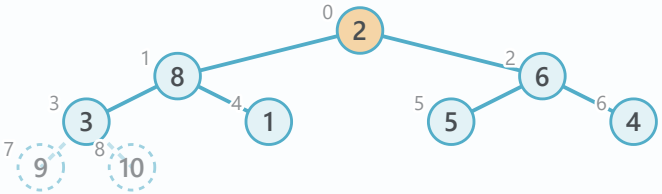
```
1  public int removeMax() {          // requires n > 0
▶    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

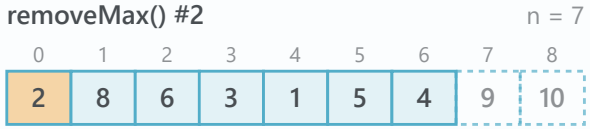
VARIABLES

`max = 9` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

Code Trace: two `removeMax()` calls in a row





The root and the last item swap (0 ↔ 7); **n = 7**, so 9 leaves the heap (faded). The tree is still complete; the new root 2 may break the order.

Step 11 / 17

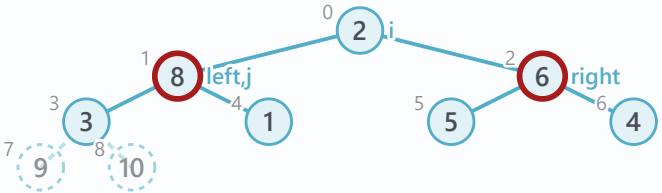
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
▶    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

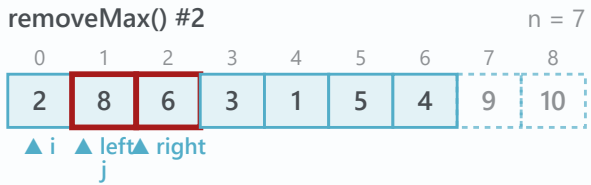
VARIABLES

`max = 9` `n = 7`

`heap = [2, 8, 6, 3, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row





Left child 1 (8), right child 2 (6) → the larger is **8**, **j = 1**.

Step 12 / 17

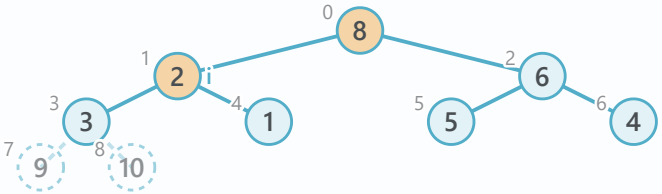
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
▶    sink(0);
5    return max;
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;            // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;           // down one level
13   }
14 }
```

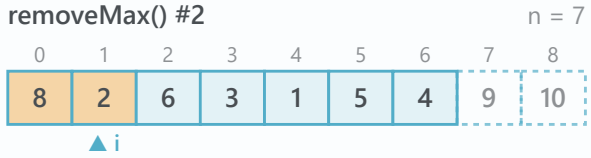
VARIABLES

`max = 9` `i = 0` `j = 1` `n = 7`

`heap = [2, 8, 6, 3, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row





2 < 8 → **swap** (index 0 ↔ 1); the larger child rises, **i = 1**.

Step 13 / 17

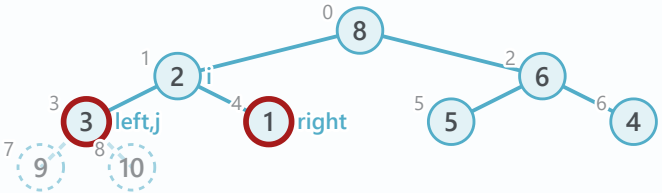
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

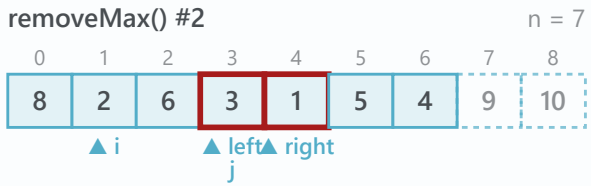
VARIABLES

`max = 9` `i = 1` `j = 1` `n = 7`

`heap = [8, 2, 6, 3, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row





Left child 3 (3), right child 4 (1) → the larger is **3**, **j = 3**.

Step 14 / 17

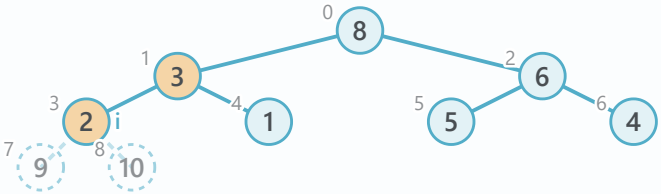
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
▶      int j = 2 * i + 1;              // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

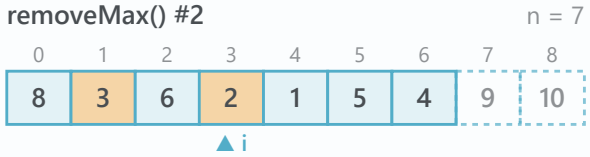
VARIABLES

`max = 9` `i = 1` `j = 3` `n = 7`

`heap = [8, 2, 6, 3, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row





2 < 3 → **swap** (index 1 ↔ 3); the larger child rises, **i = 3**.

Step 15 / 17

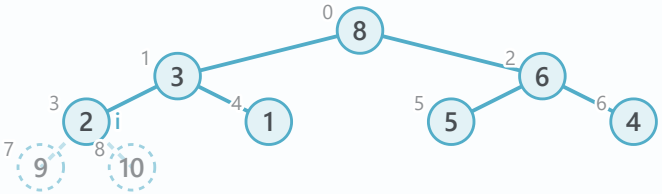
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;             // down one level
13   }
14 }
```

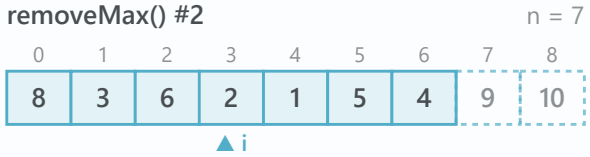
VARIABLES

`max = 9` `i = 3` `j = 3` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row





i = 3: 2·3 + 1 = 7 ≥ n = 7 → no children (a leaf), so the loop ends.

Step 16 / 17

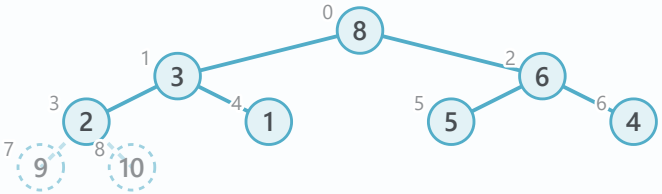
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
5    return max;
6  }
7  private void sink(int i) {          // down-heap
▶    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

VARIABLES

`max = 9` `i = 3` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

Code Trace: two `removeMax()` calls in a row



removeMax() #2 n = 7

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

sink is done, and the array is a valid max-heap again. **return 9**. The path spans at most $\lfloor \log_2 n \rfloor$ levels → **O(log n)**.

Step 17 / 17

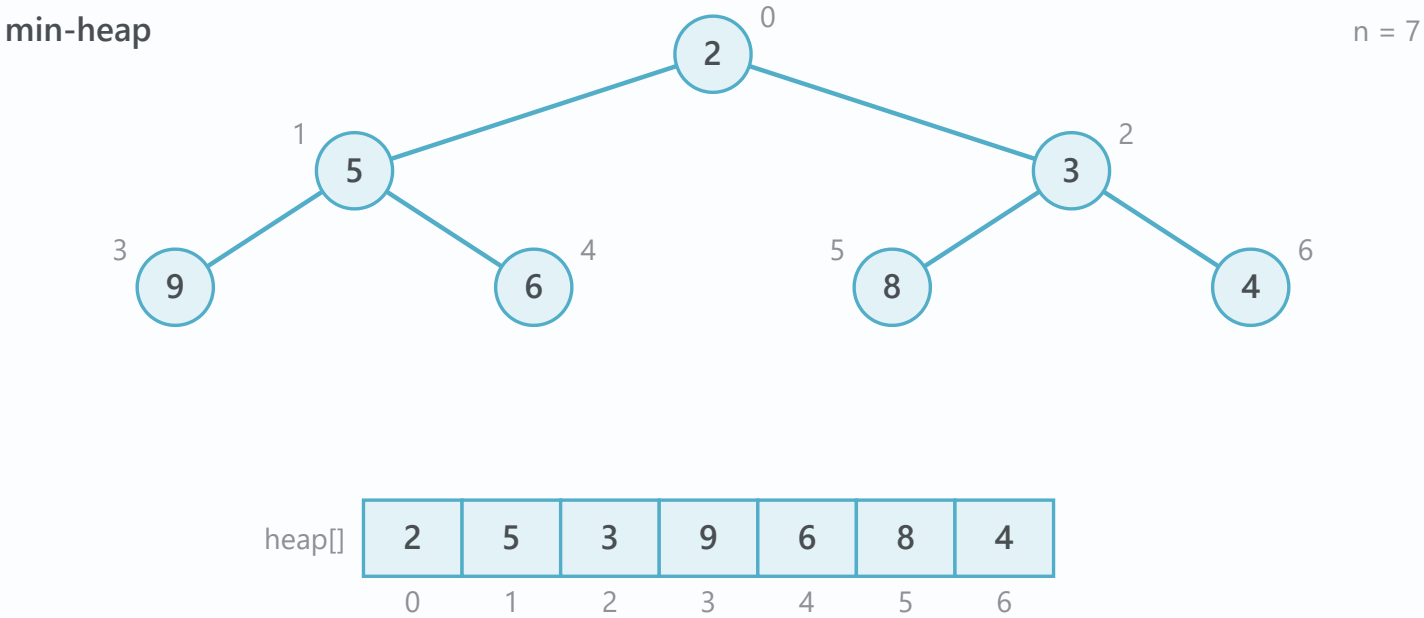
```
1  public int removeMax() {          // requires n > 0
2    int max = heap[0];
3    swap(0, n - 1);  n--;           // last item to the root
4    sink(0);
▶    return max;
6  }
7  private void sink(int i) {          // down-heap
8    while (2 * i + 1 < n) {         // has a left child?
9      int j = 2 * i + 1;              // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;             // down one level
13   }
14 }
```

VARIABLES

`max = 9` `n = 7`

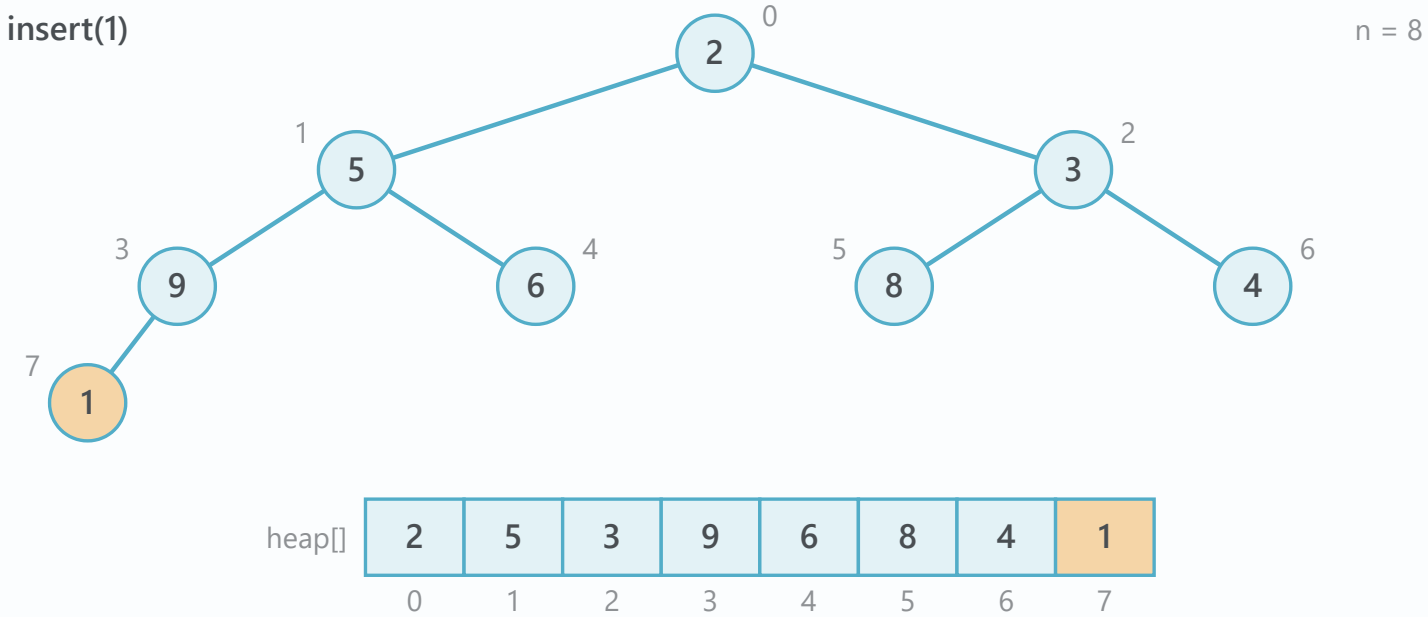
`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

Min-Heap: insert(1) and removeMin()



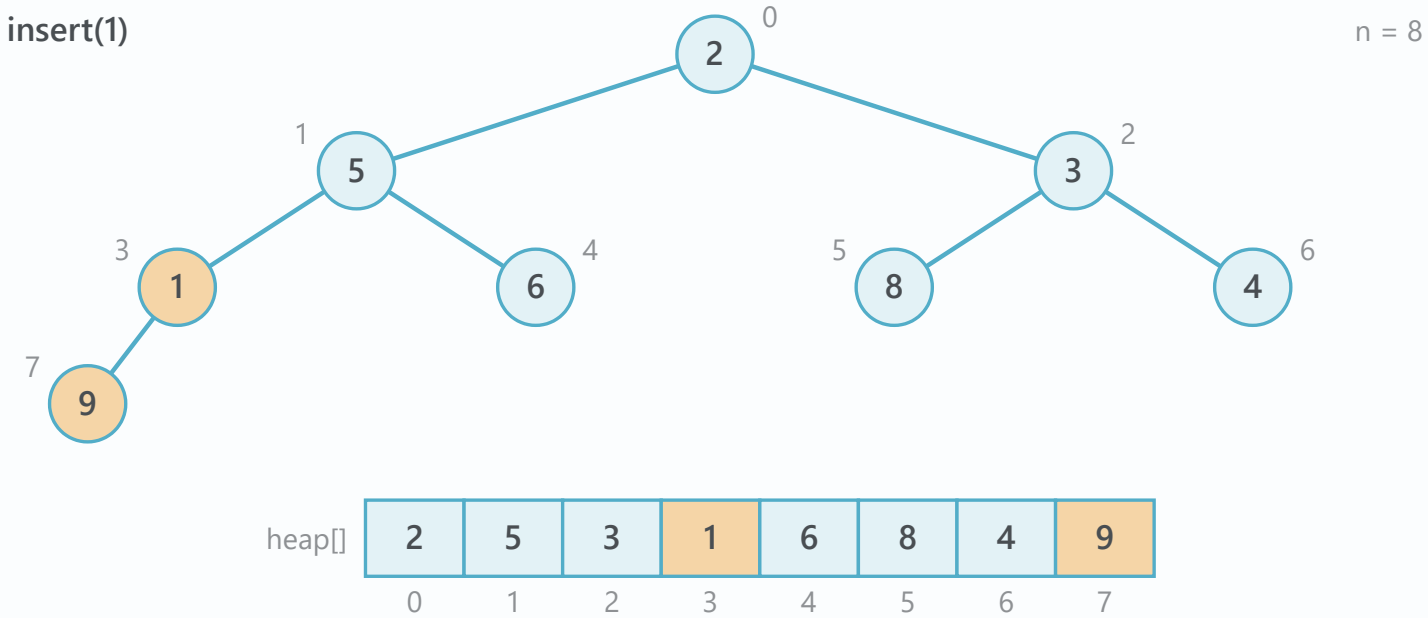
Initial min-heap: every parent ≤ its children, and the smallest item (2) is at the root.

Min-Heap: insert(1) and removeMin()



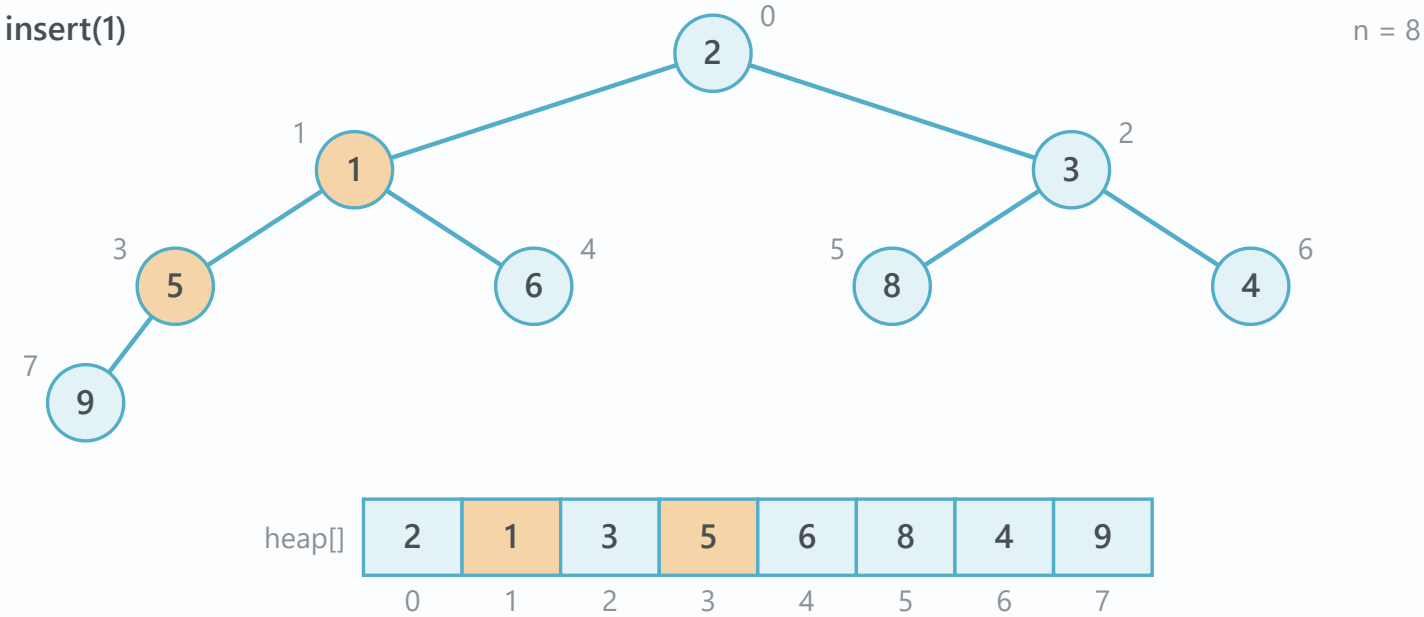
insert(1): 1 goes into the first free slot (index 7). Its parent: (7−1)/2 = 3 → 9.

Min-Heap: insert(1) and removeMin()



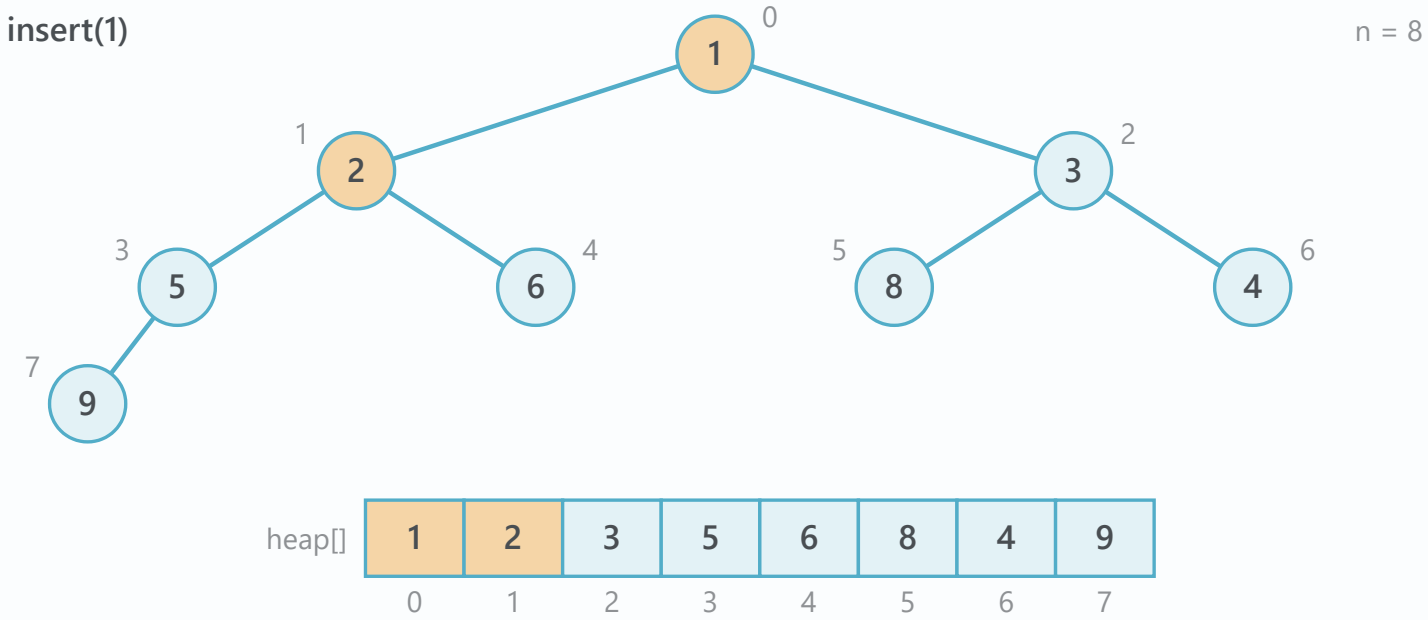
1 < 9 → swap (index 7 ↔ 3).

Min-Heap: insert(1) and removeMin()



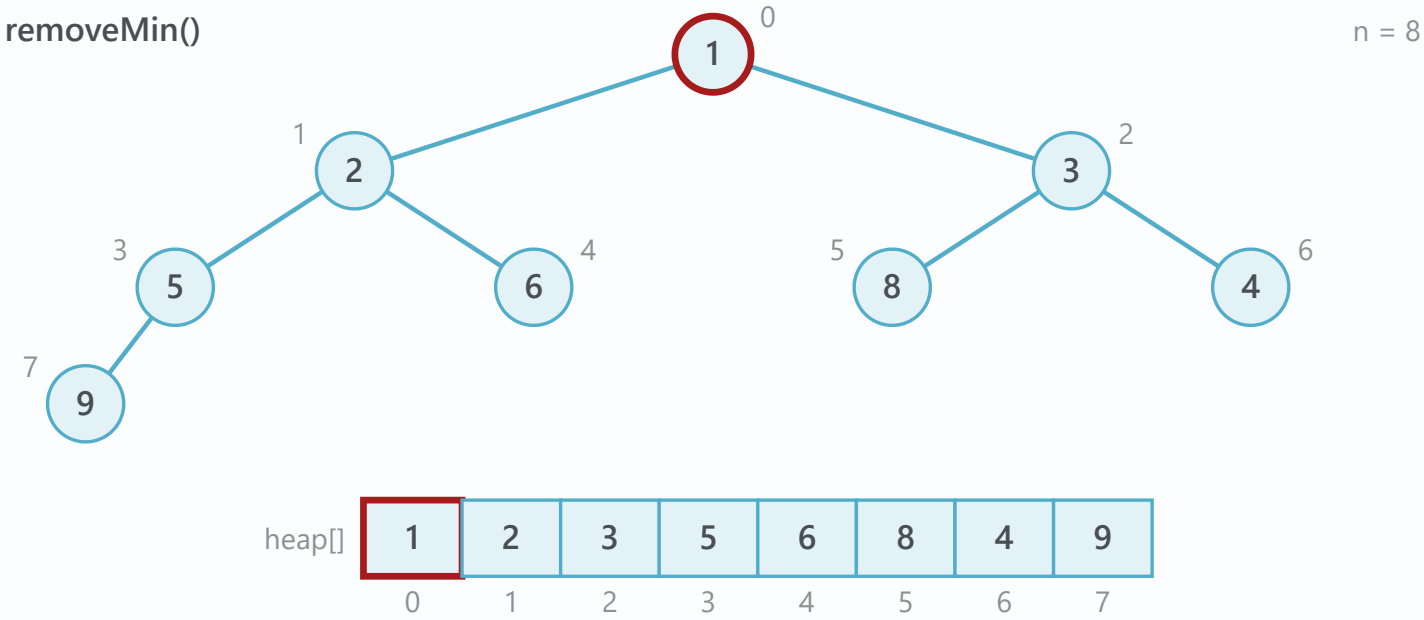
1 < 5 → swap (index 3 ↔ 1).

Min-Heap: insert(1) and removeMin()



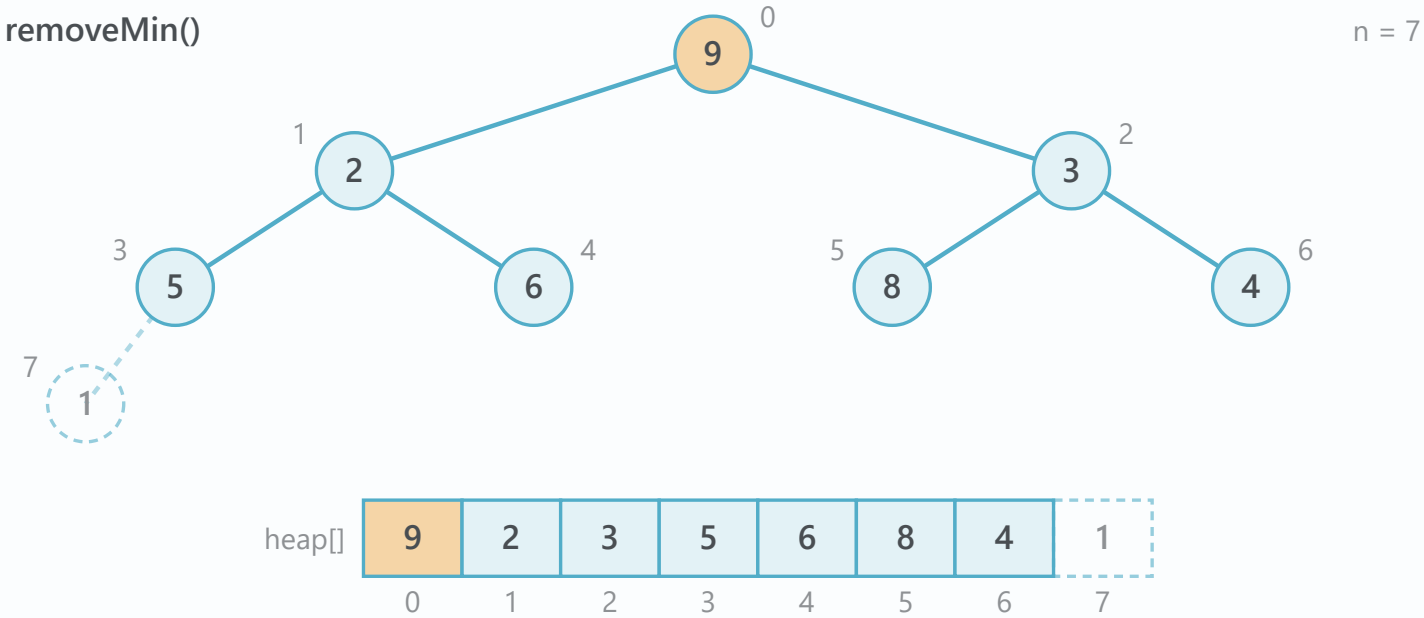
1 < 2 → swap (index 1 ↔ 0). At the root → stop.

Min-Heap: insert(1) and removeMin()



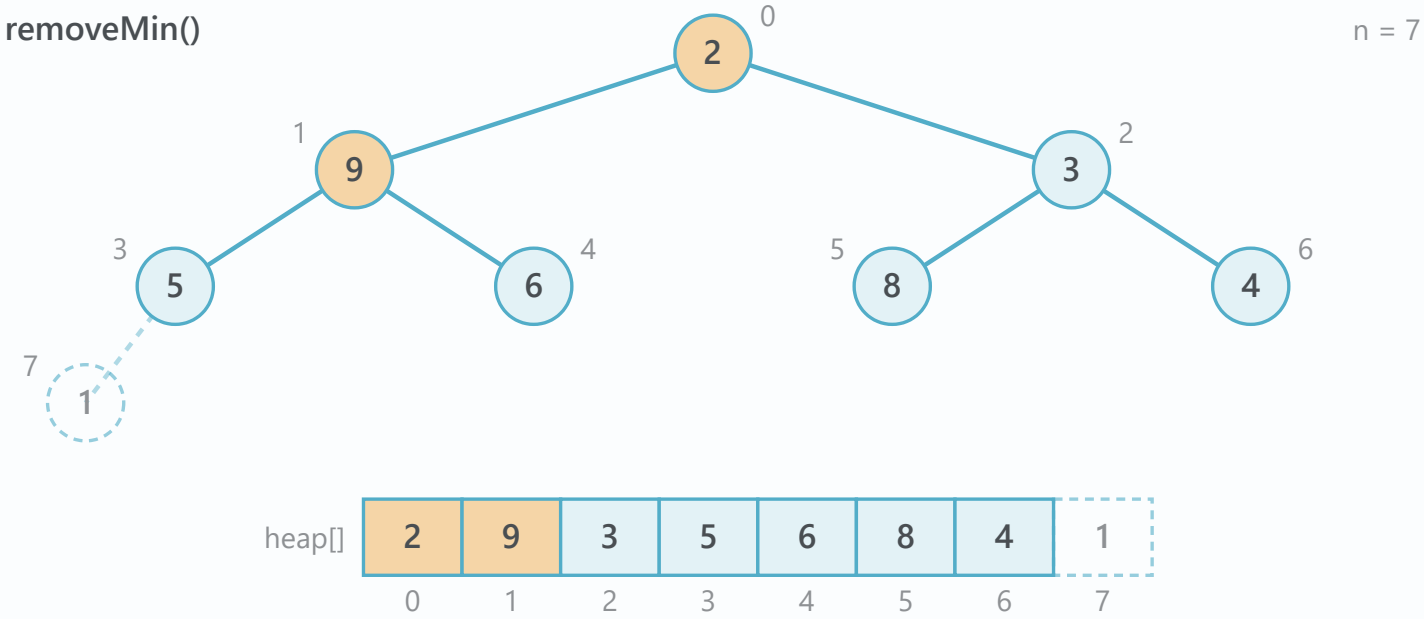
removeMin(): min = heap[0] = 1 (the root) is the value to return.

Min-Heap: insert(1) and removeMin()



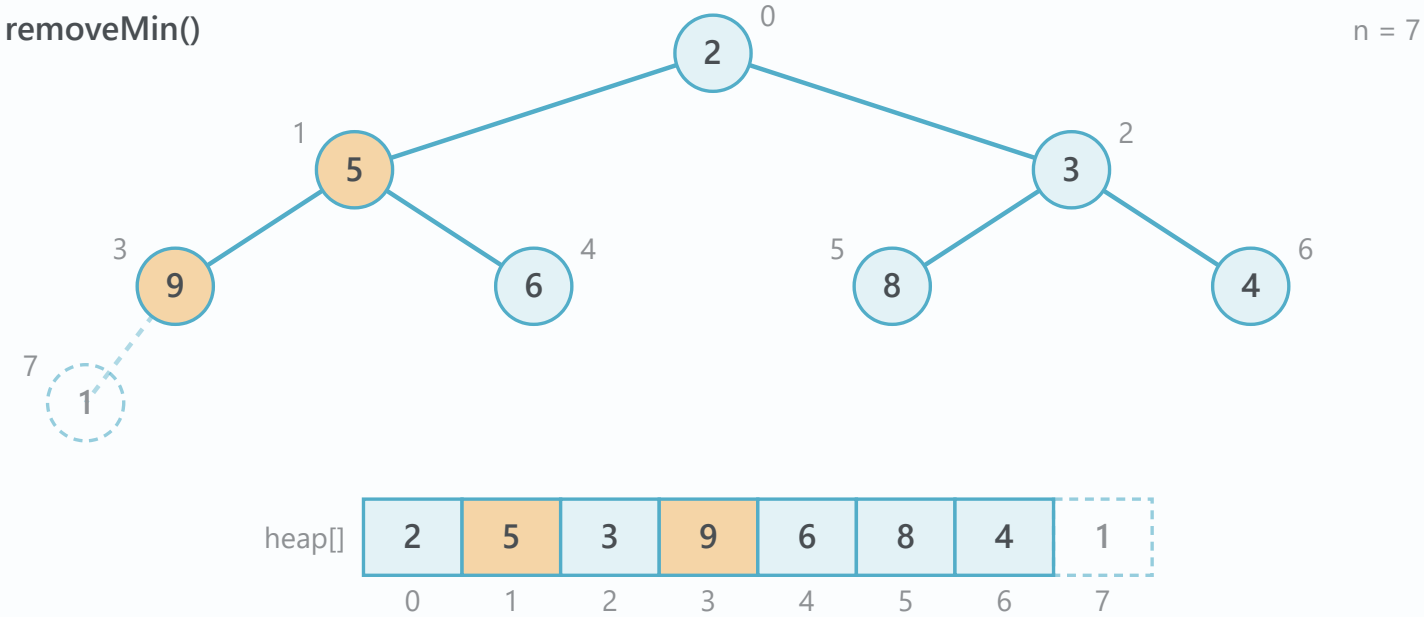
Swap the root with the last item (0 ↔ 7); n = 7, and 1 is outside the heap. The new root 9 may break heap order → down-heap.

Min-Heap: insert(1) and removeMin()



Children: 2 and 3 → the smaller is 2. 9 > 2 → swap (0 ↔ 1).

Min-Heap: insert(1) and removeMin()



Children: 5 and 6 → the smaller is 5. 9 > 5 → swap (1 ↔ 3). Index 3 is a leaf → stop.

Cost of Heap Operations

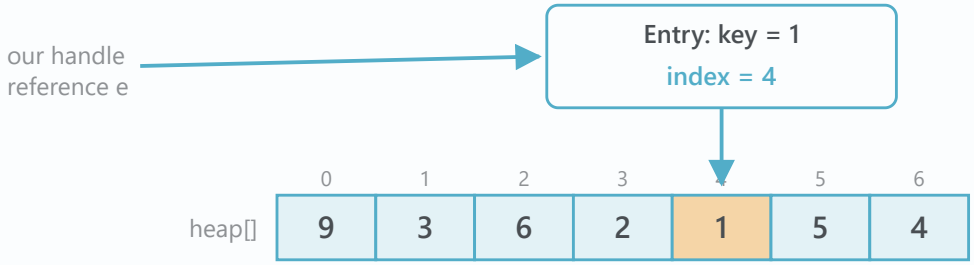
Operation	Cost	Why?
`size`, `isEmpty`	O(1)	a counter n is kept
`max` / `min` (peek)	O(1)	the item is always at heap[0]
`insert`	O(log n)*	Up-heap: one path from a leaf to the root
`removeMax` (removeMin)	O(log n)*	Down-heap: one path from the root to a leaf
Searching for an arbitrary item	O(n)	A heap is not sorted, so it is no good for searching

* Amortized, including growing and shrinking the array. With height $\lfloor \log_2 n \rfloor$, a heap of $n = 10^6$ items needs at most 19 swaps.

INSERTION AND REMOVAL

Adaptable Priority Queues

Sometimes an item's **priority changes** while it waits (a distance shrinks in Dijkstra's algorithm, a patient gets worse), or the item has to be **removed** from the middle. Goodrich defines two extra operations for this:



Locator: each entry knows its position in the heap array, updated on every swap.

replaceKey(e, k)

Change `heap[e.index]`; if the key grew, **up-heap**; if it shrank, **down-heap** (max-heap).

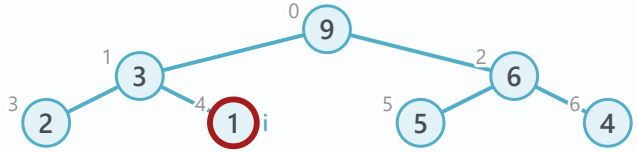
remove(e)

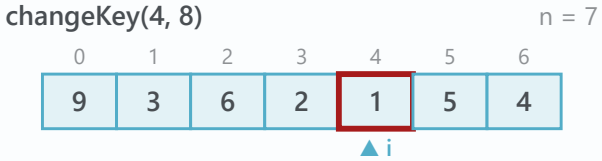
Swap with the last item, n − 1, then up-heap or down-heap the item that took its place.

Why a locator?

Without one, finding the entry is O(n); with one it is O(1), so both operations are **O(log n)**.

Code Trace: `changeKey(4, 8)`





changeKey(4, 8): the key at index 4 is 1. Save the old value: **old = 1**.

Step 1 / 7

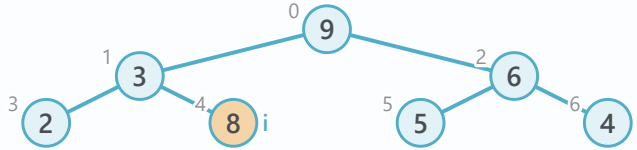
```
1  public void changeKey(int i, int key) {
▶    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {        // up-heap
8    while (i > 0) {
9      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

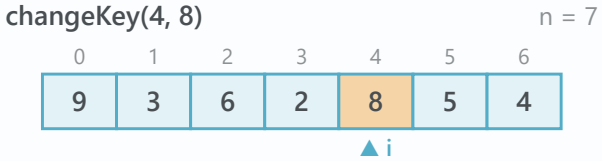
VARIABLES

`i = 4` `key = 8` `old = 1` `n = 7`

`heap = [9, 3, 6, 2, 1, 5, 4]`

Code Trace: `changeKey(4, 8)`





heap[4] = 8. The tree is still complete, but heap order may now be broken around this node.

Step 2 / 7

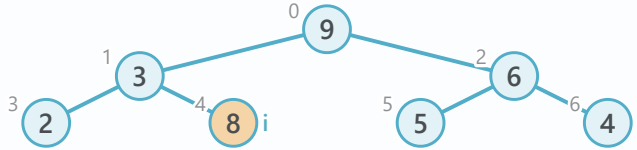
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
▶    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {        // up-heap
8    while (i > 0) {
9      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

VARIABLES

`i = 4` `key = 8` `old = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

Code Trace: `changeKey(4, 8)`



changeKey(4, 8) n = 7

0	1	2	3	4	5	6
9	3	6	2	8	5	4
				▲ i		

8 > 1: the key **increased**, so it can only conflict with its parent → call **swim(4)**.

Step 3 / 7

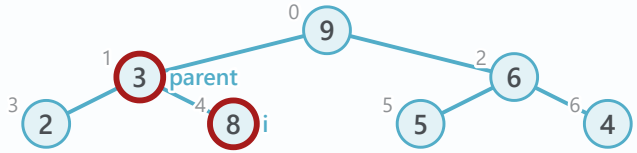
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
▶    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {       // up-heap
8    while (i > 0) {
9      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

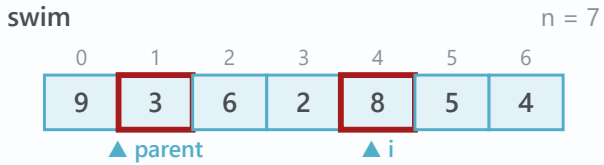
VARIABLES

`i = 4` `key = 8` `old = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

Code Trace: `changeKey(4, 8)`





Parent (4 − 1) / 2 = 1: heap[1] = 3 < 8 → swap needed.

Step 4 / 7

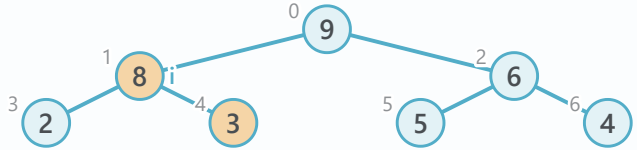
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

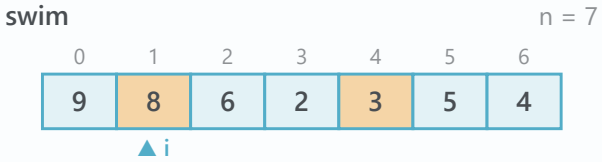
VARIABLES

`i = 4` `parent = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

Code Trace: `changeKey(4, 8)`





Swap (index 4 ↔ 1); **i = 1**. The 3 that moved down was already larger than its other child.

Step 5 / 7

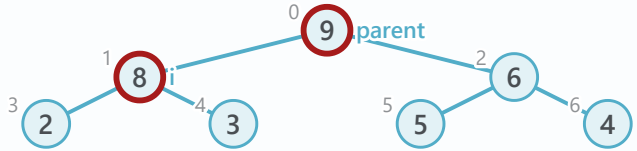
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {        // up-heap
8    while (i > 0) {
9      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
▶      swap(i, parent);  i = parent;
12   }
13 }
```

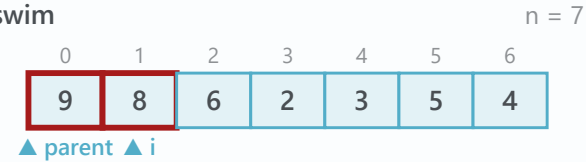
VARIABLES

`i = 1` `parent = 1` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

Code Trace: `changeKey(4, 8)`





Parent 0: heap[0] = 9 ≥ 8 → order holds, **return**.

Step 6 / 7

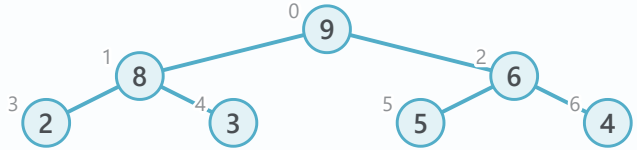
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void swim(int i) {        // up-heap
▶    while (i > 0) {
▶      int parent = (i - 1) / 2;
▶      if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

VARIABLES

`i = 1` `parent = 0` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

Code Trace: `changeKey(4, 8)`



changeKey(4, 8) n = 7

0	1	2	3	4	5	6
9	8	6	2	3	5	4

changeKey is done: 8 moved up one level. Only the path 4 → 1 → 0 was examined: **O(log n)**.

Step 7 / 7

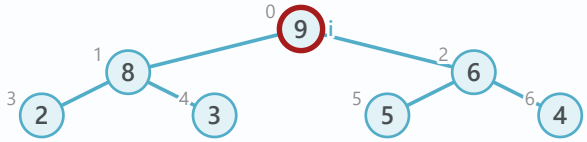
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
▶  }
7  private void swim(int i) {        // up-heap
8    while (i > 0) {
9      int parent = (i - 1) / 2;
10     if (heap[parent] >= heap[i]) return;
11     swap(i, parent);  i = parent;
12   }
13 }
```

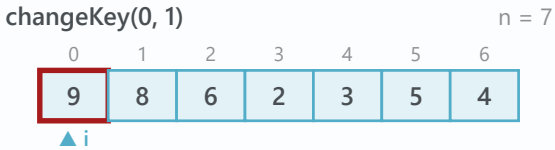
VARIABLES

`n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`





changeKey(0, 1): the key at index 0 is 9. Save the old value: **old = 9**.

Step 1 / 9

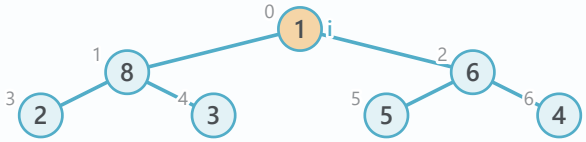
```
1  public void changeKey(int i, int key) {
▶    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

VARIABLES

`i = 0` `key = 1` `old = 9` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`



changeKey(0, 1) n = 7

0	1	2	3	4	5	6
1	8	6	2	3	5	4
▲ i						

heap[0] = 1. The tree is still complete, but heap order may now be broken around this node.

Step 2 / 9

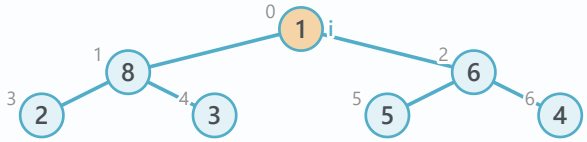
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
▶    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

VARIABLES

`i = 0` `key = 1` `old = 9` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`



changeKey(0, 1) n = 7

0	1	2	3	4	5	6
1	8	6	2	3	5	4
▲ i						

1 < 9: the key **decreased**, so it can only conflict with its children → call **sink(0)**.

Step 3 / 9

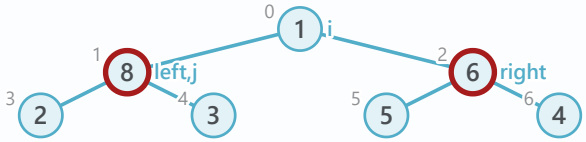
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
▶    if (key > old) swim(i);       // increased: move up
▶    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

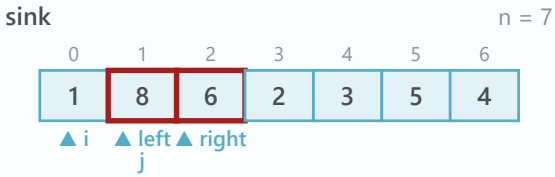
VARIABLES

`i = 0` `key = 1` `old = 9` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`





Children 8 and 6 → the larger is **8**, **j = 1**.

Step 4 / 9

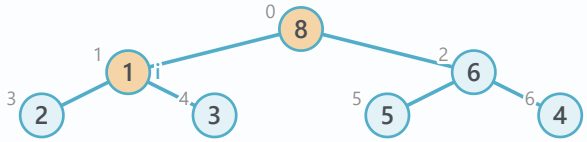
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {
▶      int j = 2 * i + 1;                // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

VARIABLES

`i = 0` `j = 1` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`



sink n = 7

0	1	2	3	4	5	6
8	1	6	2	3	5	4
	▲ i					

1 < 8 → **swap** (0 ↔ 1); the larger child moves up, **i = 1**.

Step 5 / 9

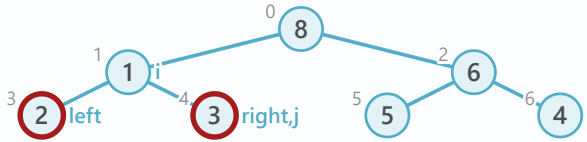
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;
13   }
14 }
```

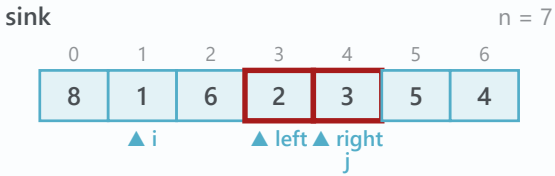
VARIABLES

`i = 1` `j = 1` `n = 7`

`heap = [8, 1, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`





Children 2 and 3 → the larger is **3**, **j = 4**.

Step 6 / 9

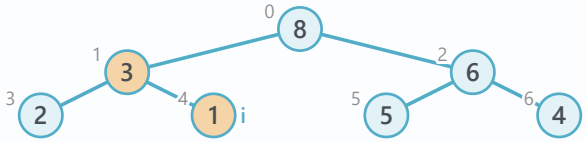
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {
▶      int j = 2 * i + 1;               // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

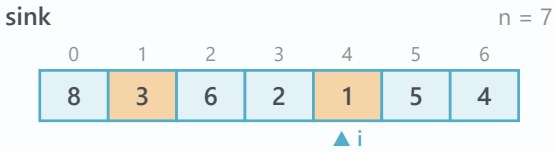
VARIABLES

`i = 1` `j = 4` `n = 7`

`heap = [8, 1, 6, 2, 3, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`





1 < 3 → **swap** (1 ↔ 4); the larger child moves up, **i = 4**.

Step 7 / 9

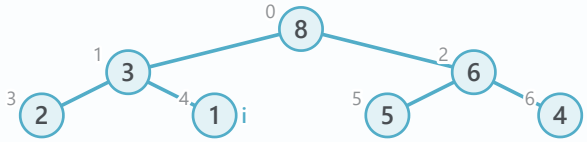
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;
13   }
14 }
```

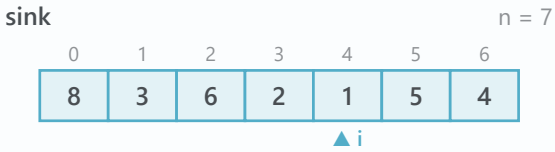
VARIABLES

`i = 4` `j = 4` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`





i = 4: 2·4 + 1 = 9 ≥ n → a leaf, so the loop ends.

Step 8 / 9

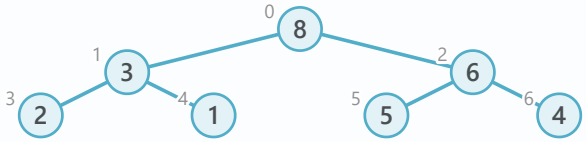
```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
6  }
7  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

VARIABLES

`i = 4` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4]`

Code Trace (continued): `changeKey(0, 1)`



changeKey(0, 1) n = 7

0	1	2	3	4	5	6
8	3	6	2	1	5	4

Done: 1 sank two levels and 8 rose to the root. `changeKey(0, x)` with a small x does exactly what the sink step of removeMax does.

Step 9 / 9

```
1  public void changeKey(int i, int key) {
2    int old = heap[i];
3    heap[i] = key;
4    if (key > old) swim(i);       // increased: move up
5    else sink(i);                 // decreased: move down
▶  }
7  private void sink(int i) {         // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;               // j: larger child
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     swap(i, j);  i = j;
13   }
14 }
```

VARIABLES

`n = 7` `heap = [8, 3, 6, 2, 1, 5, 4]`

Building a Heap from n Items

Option 1: n insertions

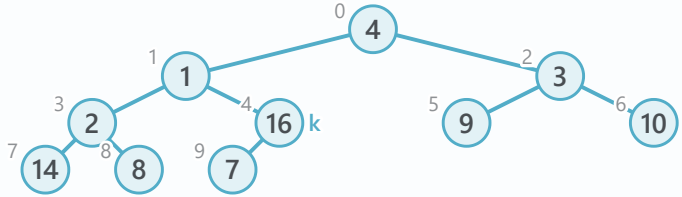
- – Each insertion does an up-heap: O(log n).
- – Total **O(n log n)**.
- – The only option when items arrive one at a time (online).

Option 2: bottom-up

- + If all the items are available up front, take the array as it is.
- + Run **down-heap** on every node, from the last internal node (⌊n/2⌋ − 1) back to the root.
- + Total **O(n)**.

```
for (int k = n / 2 - 1; k >= 0; k--)   // from the last internal node to the root
    sink(k);                             // sink k; its subtrees are already heaps
```

Code Trace: `buildHeap()` (k = 4, 3, 2)



buildHeap() n = 10

0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

▲ k

An arbitrary array (not a heap). Indices 5…9 are leaves, and each is already a heap on its own. Start at **k = n/2 − 1 = 4** (the last internal node).

Step 1 / 9

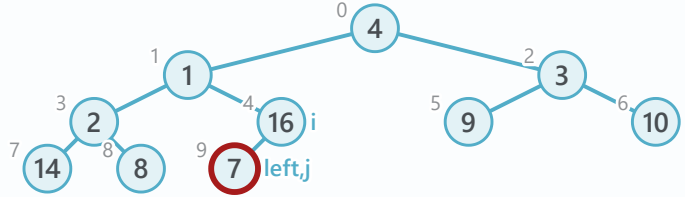
```
1  public void buildHeap() {        // arbitrary array
▶    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

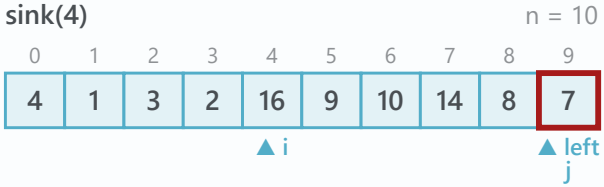
VARIABLES

k = 4 n = 10

heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

Code Trace: `buildHeap()` (k = 4, 3, 2)





k = 4 → sink(4). Only a left child: index 9 (7) → **j = 9**.

Step 2 / 9

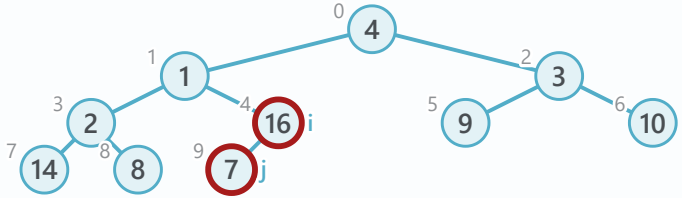
```
 1 public void buildHeap() {        // arbitrary array
 ▶   for (int k = n / 2 - 1; k >= 0; k--)
 ▶     sink(k);                     // subtrees are heaps
 4 }
 5 private void sink(int i) {         // down-heap
 ▶   while (2 * i + 1 < n) {        // has a left child?
 ▶     int j = 2 * i + 1;             // j: larger child
 ▶     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
 9     if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

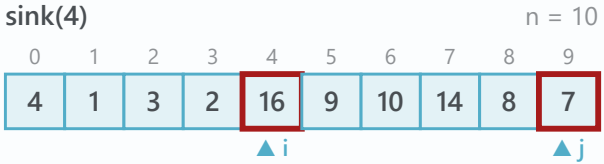
VARIABLES

`k = 4` `i = 4` `j = 9` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

Code Trace: `buildHeap()` (k = 4, 3, 2)





heap[4] = 16 ≥ heap[9] = 7 → no smaller than either child, so heap order holds: **return**.

Step 3 / 9

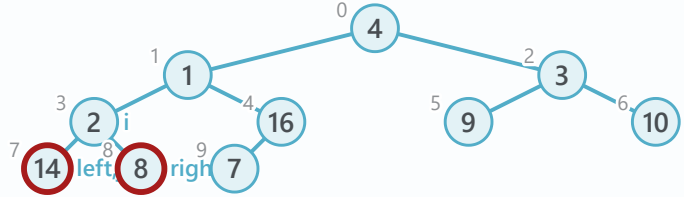
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
▶      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

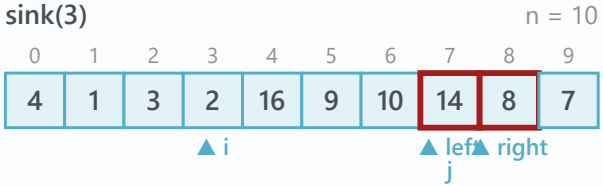
VARIABLES

`k = 4` `i = 4` `j = 9` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

Code Trace: `buildHeap()` (k = 4, 3, 2)





k = 3 → sink(3). Left child 7 (14), right child 8 (8) → the larger is **14**, **j = 7**.

Step 4 / 9

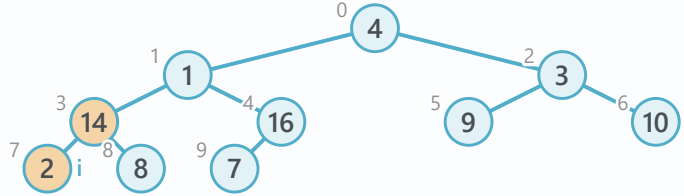
```
 1 public void buildHeap() {        // arbitrary array
 ▶   for (int k = n / 2 - 1; k >= 0; k--)
 ▶     sink(k);                     // subtrees are heaps
 4 }
 5 private void sink(int i) {         // down-heap
 ▶   while (2 * i + 1 < n) {        // has a left child?
 ▶     int j = 2 * i + 1;             // j: larger child
 ▶     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
 9     if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

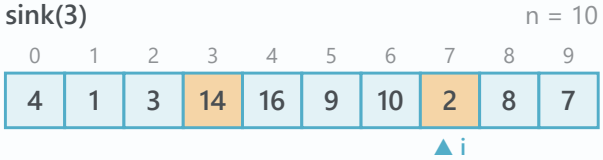
VARIABLES

`k = 3` `i = 3` `j = 7` `n = 10`

heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

Code Trace: `buildHeap()` (k = 4, 3, 2)





2 < 14 → **swap** (index 3 ↔ 7); the larger child rises, **i = 7**.

Step 5 / 9

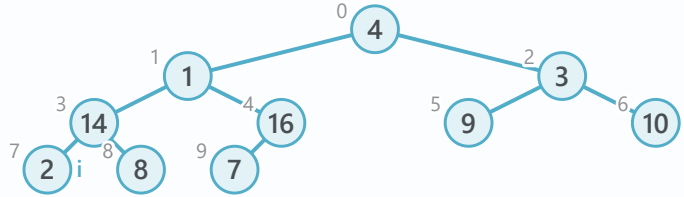
```
1  public void buildHeap() {          // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                       // subtrees are heaps
4  }
5  private void sink(int i) {          // down-heap
6    while (2 * i + 1 < n) {          // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

`k = 3` `i = 7` `j = 7` `n = 10`

`heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]`

Code Trace: `buildHeap()` (k = 4, 3, 2)



sink(3) n = 10

0	1	2	3	4	5	6	7	8	9
4	1	3	14	16	9	10	2	8	7
							▲ i		

i = 7: 2·7 + 1 = 15 ≥ n = 10 → no children (a leaf), so the loop ends.

Step 6 / 9

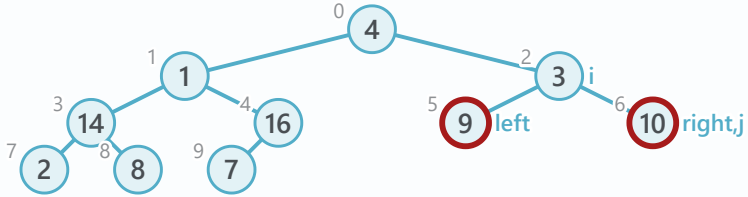
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

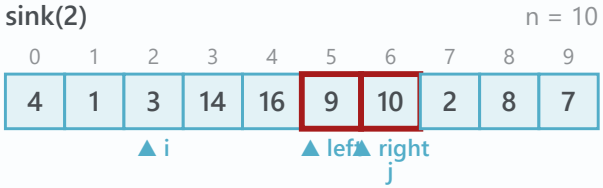
VARIABLES

`k = 3` `i = 7` `n = 10`

`heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]`

Code Trace: `buildHeap()` (k = 4, 3, 2)





k = 2 → sink(2). Left child 5 (9), right child 6 (10) → the larger is **10**, **j = 6**.

Step 7 / 9

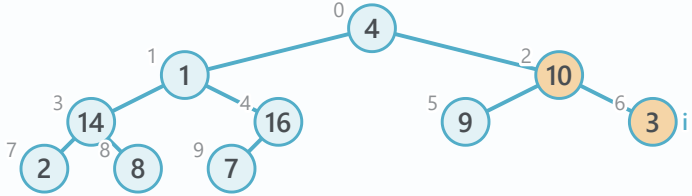
```
1  public void buildHeap() {        // arbitrary array
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

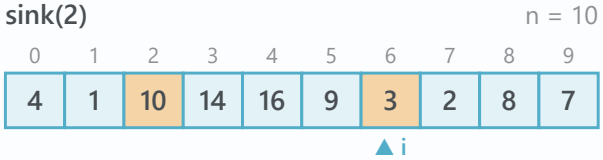
VARIABLES

`k = 2` `i = 2` `j = 6` `n = 10`

heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]

Code Trace: `buildHeap()` (k = 4, 3, 2)





3 < 10 → **swap** (index 2 ↔ 6); the larger child rises, **i = 6**.

Step 8 / 9

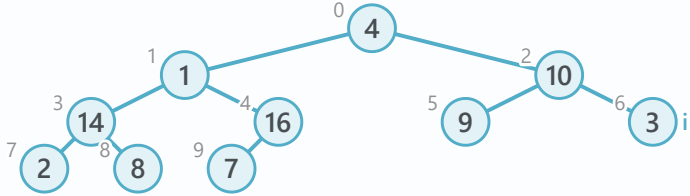
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

`k = 2` `i = 6` `j = 6` `n = 10`

`heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]`

Code Trace: `buildHeap()` (k = 4, 3, 2)



sink(2) n = 10

0	1	2	3	4	5	6	7	8	9
4	1	10	14	16	9	3	2	8	7
						▲ i			

i = 6: 2·6 + 1 = 13 ≥ n = 10 → no children (a leaf), so the loop ends.

Step 9 / 9

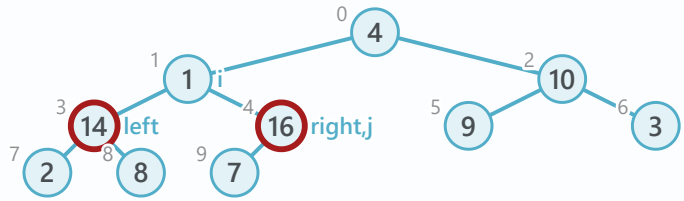
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

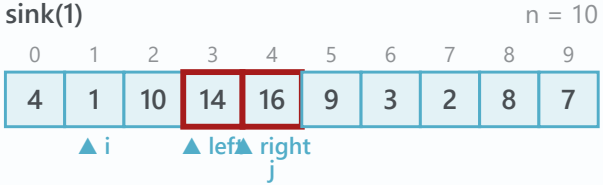
VARIABLES

`k = 2` `i = 6` `n = 10`

`heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]`

Code Trace (continued): k = 1, 0





k = 1 → sink(1). Left child 3 (14), right child 4 (16) → the larger is **16**, **j = 4**.

Step 1 / 13

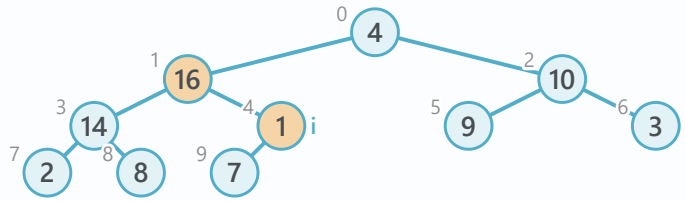
```
1  public void buildHeap() {        // arbitrary array
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 1 | i = 1 | j = 4 | n = 10

heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]

Code Trace (continued): k = 1, 0



sink(1) n = 10

0	1	2	3	4	5	6	7	8	9
4	16	10	14	1	9	3	2	8	7
				▲ i					

1 < 16 → **swap** (index 1 ↔ 4); the larger child rises, **i = 4**.

Step 2 / 13

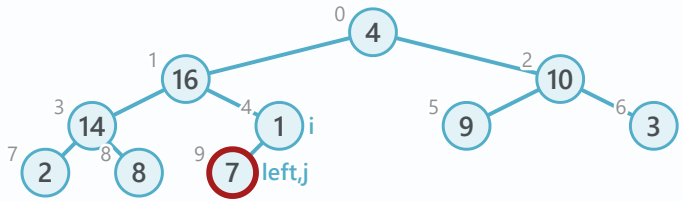
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

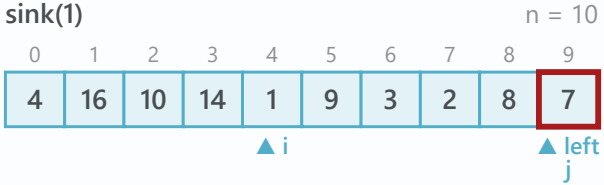
VARIABLES

k = 1 | i = 4 | j = 4 | n = 10

heap = [4, 16, 10, 14, 1, 9, 3, 2, 8, 7]

Code Trace (continued): k = 1, 0





Only a left child: index 9 (7) → **j = 9**.

Step 3 / 13

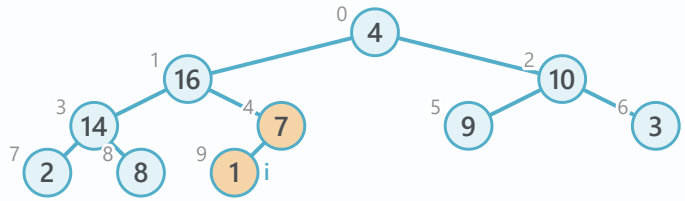
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 1 | i = 4 | j = 9 | n = 10

heap = [4, 16, 10, 14, 1, 9, 3, 2, 8, 7]

Code Trace (continued): k = 1, 0



sink(1) n = 10

0	1	2	3	4	5	6	7	8	9
4	16	10	14	7	9	3	2	8	1
									▲ i

1 < 7 → **swap** (index 4 ↔ 9); the larger child rises, **i = 9**.

Step 4 / 13

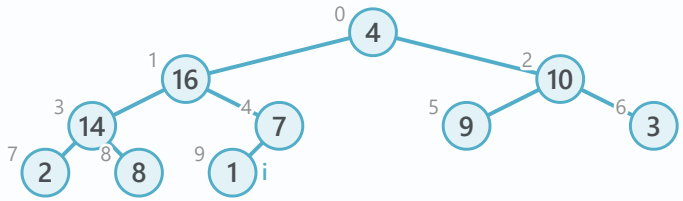
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 1 | i = 9 | j = 9 | n = 10

heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0



sink(1) n = 10

0	1	2	3	4	5	6	7	8	9
4	16	10	14	7	9	3	2	8	1
									▲ i

i = 9: 2·9 + 1 = 19 ≥ n = 10 → no children (a leaf), so the loop ends.

Step 5 / 13

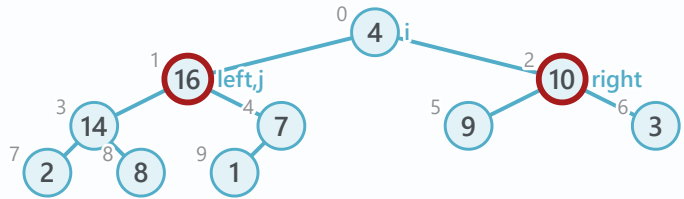
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

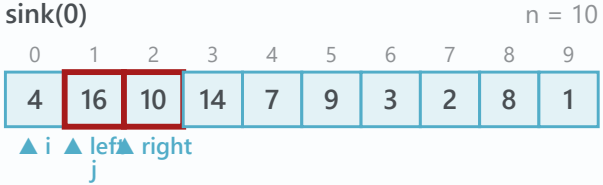
VARIABLES

k = 1 | i = 9 | n = 10

heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0





k = 0 → sink(0). Left child 1 (16), right child 2 (10) → the larger is **16**, **j = 1**.

Step 6 / 13

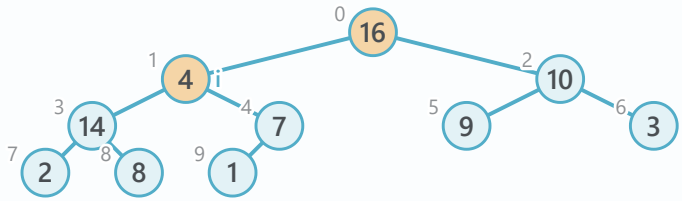
```
1  public void buildHeap() {        // arbitrary array
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 0 | i = 0 | j = 1 | n = 10

heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0



sink(0) n = 10

0	1	2	3	4	5	6	7	8	9
16	4	10	14	7	9	3	2	8	1
	▲ i								

4 < 16 → **swap** (index 0 ↔ 1); the larger child rises, **i = 1**.

Step 7 / 13

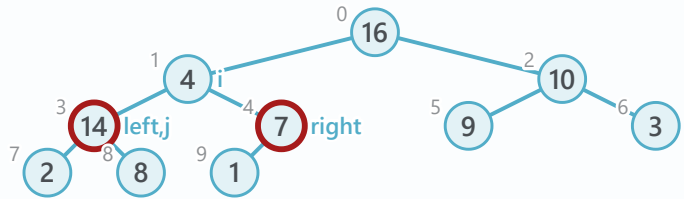
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

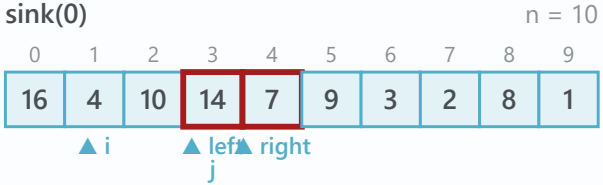
VARIABLES

k = 0 | i = 1 | j = 1 | n = 10

heap = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0





Left child 3 (14), right child 4 (7) → the larger is **14**, **j = 3**.

Step 8 / 13

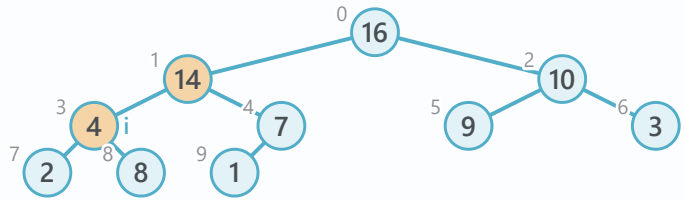
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

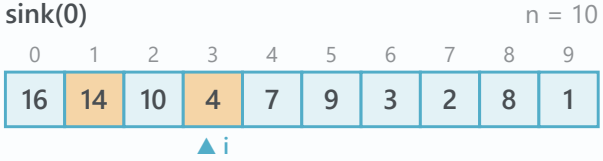
VARIABLES

k = 0 | i = 1 | j = 3 | n = 10

heap = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0





4 < 14 → **swap** (index 1 ↔ 3); the larger child rises, **i = 3**.

Step 9 / 13

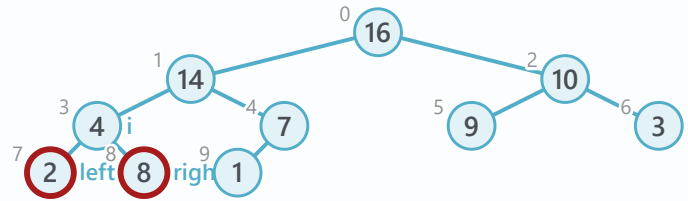
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

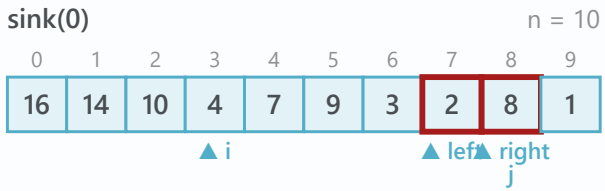
VARIABLES

k = 0 | i = 3 | j = 3 | n = 10

heap = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0





Left child 7 (2), right child 8 (8) → the larger is **8**, **j = 8**.

Step 10 / 13

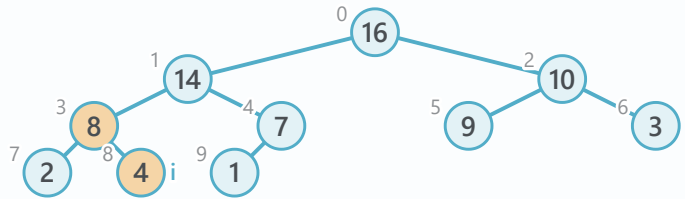
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
▶      int j = 2 * i + 1;             // j: larger child
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 0 | i = 3 | j = 8 | n = 10

heap = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]

Code Trace (continued): k = 1, 0



sink(0) n = 10

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1
								▲ i	

4 < 8 → **swap** (index 3 ↔ 8); the larger child rises, **i = 8**.

Step 11 / 13

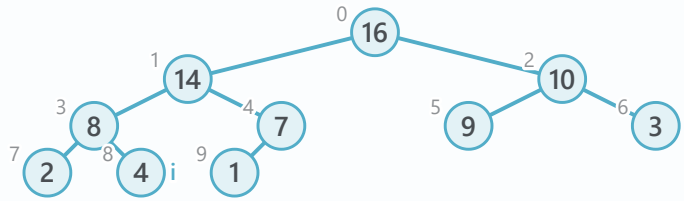
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 0 | i = 8 | j = 8 | n = 10

heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Code Trace (continued): k = 1, 0



sink(0) n = 10

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1
								▲ i	

i = 8: 2·8 + 1 = 17 ≥ n = 10 → no children (a leaf), so the loop ends.

Step 12 / 13

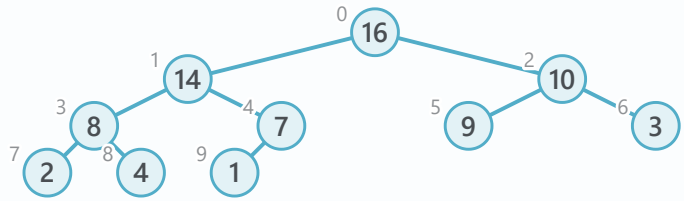
```
1  public void buildHeap() {        // arbitrary array
2    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
▶    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = 0 | i = 8 | n = 10

heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Code Trace (continued): k = 1, 0



buildHeap() n = 10

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

k = −1 → the loop ends. The array is now a **max-heap**: every parent ≥ its children, and the largest (16) is at the root. Total work: O(n).

Step 13 / 13

```
1  public void buildHeap() {        // arbitrary array
▶    for (int k = n / 2 - 1; k >= 0; k--)
3      sink(k);                     // subtrees are heaps
4  }
5  private void sink(int i) {         // down-heap
6    while (2 * i + 1 < n) {        // has a left child?
7      int j = 2 * i + 1;             // j: larger child
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     swap(i, j);  i = j;            // down one level
11   }
12 }
```

VARIABLES

k = −1 n = 10

heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Why Is Bottom-Up Construction O(n)?

Node height	Nodes	Max swaps	Total
0 (leaves)	8	0	0
1	4	1	4
2	2	2	4
3 (root)	1	3	3
n = 15			**11**

- Half the nodes are leaves and do no work at all; the few tall nodes are the only ones that do much.
- In general: $\Sigma_h \, (n / 2^{h+1}) \cdot h \leq n \rightarrow$ **O(n)**.
- With n insertions, the many deep nodes walk long paths instead: in the same tree the worst case is $0 + 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 =$ **34** swaps.

The cost of down-heap depends on a node's *height*; the cost of up-heap depends on its *depth*. Most nodes are near the bottom.

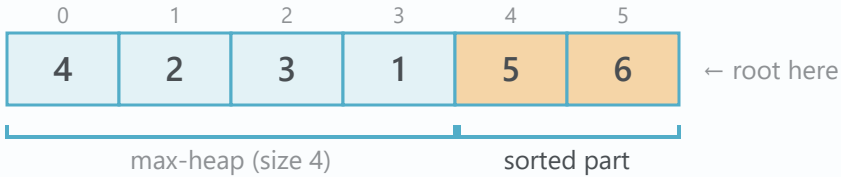
Heap Sort

Phase 1: build the heap

Turn the array into a max-heap bottom-up. **O(n)**

Phase 2: n − 1 removeMax steps

Swap the root (the largest) with the item at the end, shrink the heap by one, and down-heap the root. **O(n log n)**



- The sorted part grows at the end of the array while the heap shrinks at the front; **no extra array** is needed (in place, O(1) extra memory).
- **O(n log n)** even in the worst case: no $O(n^2)$ worst case as in quick sort.
- **Not stable**: equal keys may change their relative order.
- A max-heap sorts in ascending order, a min-heap in descending order.

Heap Sort: Java Code

```java
public static void heapSort(int[] a) {
  int n = a.length;
  // 1) build: from the last internal node to the root
  for (int k = n / 2 - 1; k >= 0; k--)
    sink(a, k, n);
  // 2) sort: move the largest to the end
  for (int last = n - 1; last > 0; last--) {
    swap(a, 0, last);
    sink(a, 0, last);           // heap size = last
  }
}
static void sink(int[] a, int i, int n) {
  while (2 * i + 1 < n) {
    int j = 2 * i + 1;
    if (j + 1 < n && a[j + 1] > a[j]) j++;
    if (a[i] >= a[j]) return;
    swap(a, i, j);  i = j;
  }
}
```

The same sink

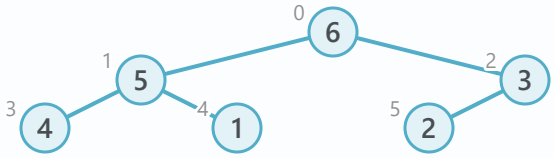
Same as in MaxHeap, except that the array and the heap size are passed as parameters.

Try it

{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

→ {1, 2, 3, 4, 7, 8, 9, 10, 14, 16}

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



heapSort(a) heap size = 6

0	1	2	3	4	5
6	5	3	4	1	2

Input a = [6, 5, 3, 4, 1, 2]. The whole array is the heap region; the sorted part on the right is still empty.

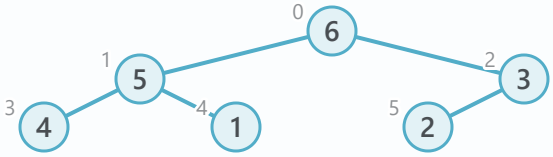
Step 1 / 15

```
1  public static void heapSort(int[] a) {
▶    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
9      sink(a, 0, last);           // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `a = [6, 5, 3, 4, 1, 2]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



1) build heap size = 6

0	1	2	3	4	5
6	5	3	4	1	2

Phase 1: sink is called for k = 2, 1, 0. This array is already a max-heap, so each call returns after its first comparison; no swaps.

Step 2 / 15

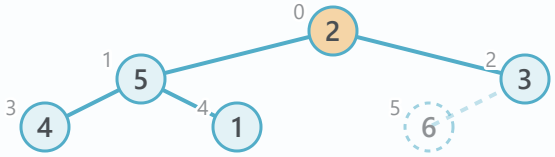
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
9      sink(a, 0, last);           // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `k = 0`

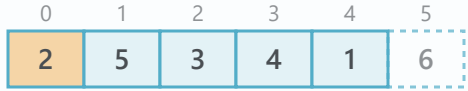
`a = [6, 5, 3, 4, 1, 2]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



last = 5

heap size = 5



faded: sorted part

last = 5: the root (the largest, 6) swaps with a[5] = 2 → 6 is in its final place; heap size 5.

Step 3 / 15

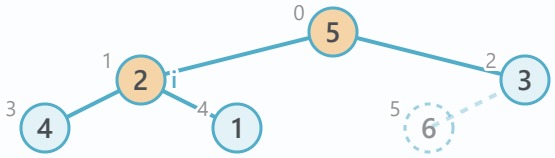
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
▶      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

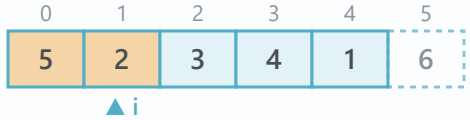
`n = 6` `last = 5`

`a = [2, 5, 3, 4, 1 | 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 5) heap size = 5



faded: sorted part

sink(a, 0, 5): at i = 0, the larger child a[1] = 5 > 2 → swap (0 ↔ 1).

Step 4 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

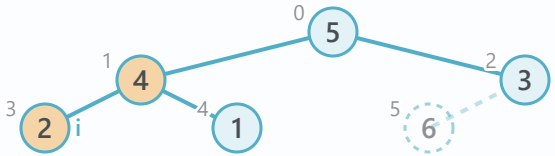
VARIABLES

`n = 6` `last = 5`

`sink: i = 1, j=1`

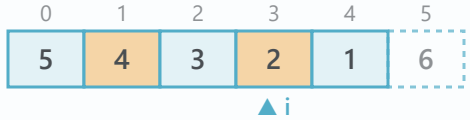
`a = [5, 2, 3, 4, 1 | 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 5)

heap size = 5



faded: sorted part

sink(a, 0, 5): at i = 1, the larger child a[3] = 4 > 2 → swap (1 ↔ 3). Index 3 is a leaf → sink ends.

Step 5 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

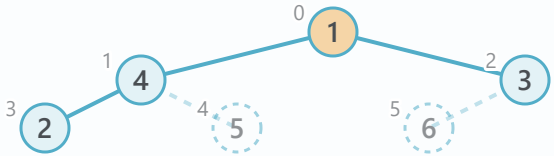
VARIABLES

`n = 6` `last = 5`

`sink: i = 3, j=3`

`a = [5, 4, 3, 2, 1 | 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



last = 4

heap size = 4



faded: sorted part

last = 4: the root (the largest, 5) swaps with a[4] = 1 → 5 is in its final place; heap size 4.

Step 6 / 15

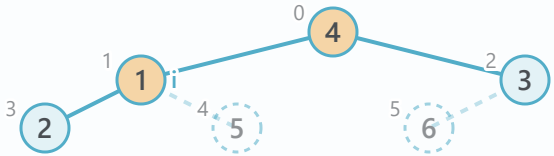
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
▶      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

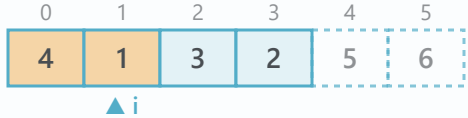
`n = 6` `last = 4`

`a = [1, 4, 3, 2 | 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 4) heap size = 4



faded: sorted part

sink(a, 0, 4): at i = 0, the larger child a[1] = 4 > 1 → swap (0 ↔ 1).

Step 7 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

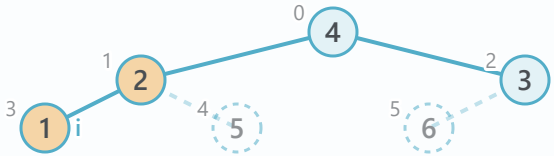
VARIABLES

`n = 6` `last = 4`

`sink: i = 1, j=1`

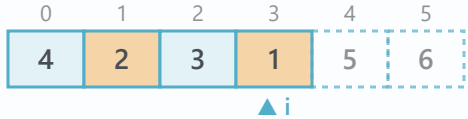
`a = [4, 1, 3, 2 | 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 4)

heap size = 4



faded: sorted part

sink(a, 0, 4): at i = 1, the larger child a[3] = 2 > 1 → swap (1 ↔ 3). Index 3 is a leaf → sink ends.

Step 8 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

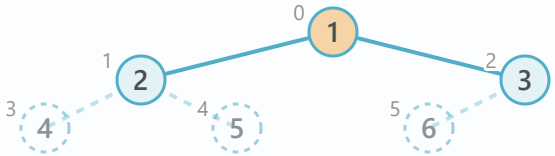
VARIABLES

`n = 6` `last = 4`

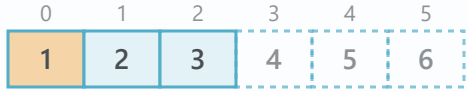
`sink: i = 3, j=3`

`a = [4, 2, 3, 1 | 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



last = 3 heap size = 3



faded: sorted part

last = 3: the root (the largest, 4) swaps with a[3] = 1 → 4 is in its final place; heap size 3.

Step 9 / 15

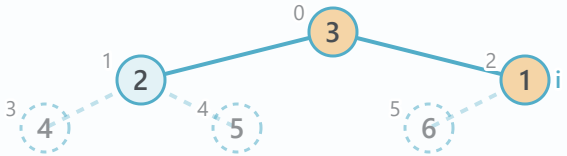
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
▶      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

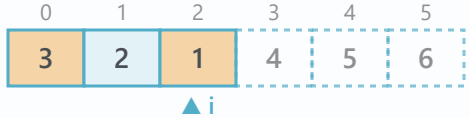
`n = 6` `last = 3`

`a = [1, 2, 3 | 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 3) heap size = 3



faded: sorted part

sink(a, 0, 3): at i = 0, the larger child a[2] = 3 > 1 → swap (0 ↔ 2). Index 2 is a leaf → sink ends.

Step 10 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

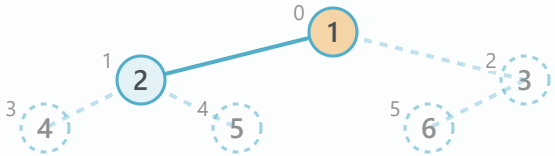
VARIABLES

`n = 6` `last = 3`

`sink: i = 2, j=2`

`a = [3, 2, 1 | 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



last = 2

heap size = 2



faded: sorted part

last = 2: the root (the largest, 3) swaps with a[2] = 1 → 3 is in its final place; heap size 2.

Step 11 / 15

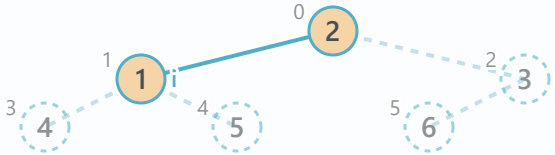
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
▶      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `last = 2`

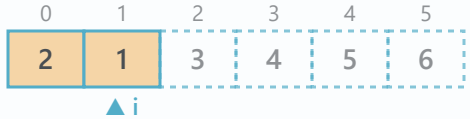
`a = [1, 2 | 3, 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 2)

heap size = 2



faded: sorted part

sink(a, 0, 2): at i = 0, the larger child a[1] = 2 > 1 → swap (0 ↔ 1). Index 1 is a leaf → sink ends.

Step 12 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

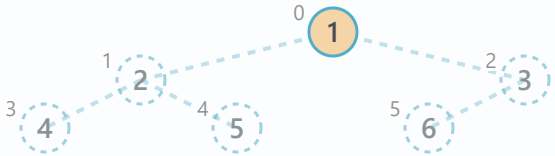
VARIABLES

`n = 6` `last = 2`

`sink: i = 1, j=1`

`a = [2, 1 | 3, 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



last = 1

heap size = 1



faded: sorted part

last = 1: the root (the largest, 2) swaps with a[1] = 1 → 2 is in its final place; heap size 1.

Step 13 / 15

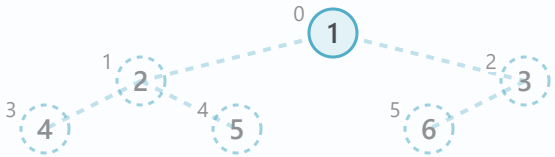
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
▶      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `last = 1`

`a = [1 | 2, 3, 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sink(a, 0, 1) heap size = 1



faded: sorted part

sink(a, 0, 1): heap size 1, the root has no children → nothing to do.

Step 14 / 15

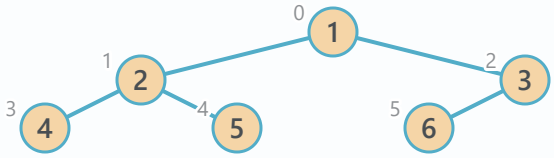
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
7    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
▶      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `last = 1`

`a = [1 | 2, 3, 4, 5, 6]`

Code Trace: `heapSort([6, 5, 3, 4, 1, 2])`



sorted

heap size = 0

0	1	2	3	4	5
1	2	3	4	5	6

last = 0 → the loop ends. The array is in **ascending order**: [1, 2, 3, 4, 5, 6]. No extra array was used (in place); total **O(n log n)**.

Step 15 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) build: from the last internal node to the
   root
4    for (int k = n / 2 - 1; k >= 0; k--)
5      sink(a, k, n);
6    // 2) sort: move the largest to the end
▶    for (int last = n - 1; last > 0; last--) {
8      swap(a, 0, last);
9      sink(a, 0, last);            // heap size = last
10   }
11 }
```

VARIABLES

`n = 6` `last = 0`

`a = [1, 2, 3, 4, 5, 6]`

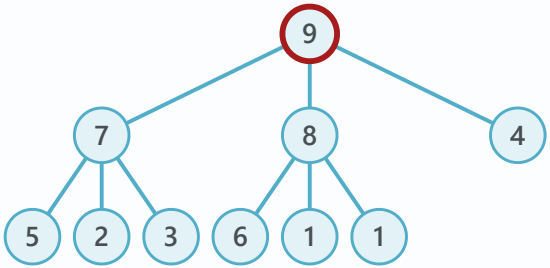
Sorting with a Priority Queue

The general idea: insert every item into a priority queue, then call `removeMin` n times. Which sorting algorithm we get depends on the implementation:

Implementation	n × insert	n × removeMin	Resulting algorithm
Unsorted list	O(n)	$O(n^2)$	Selection sort, $O(n^2)$
Sorted list	$O(n^2)$	O(n)	Insertion sort, $O(n^2)$
Binary heap	O(n log n)	O(n log n)	Heap sort, **O(n log n)**

Same ADT, different data structure: the running time drops from $O(n^2)$ to O(n log n). A concrete case of the ADT vs. data structure distinction.

Kinds of Heaps: d-ary, Binomial, Fibonacci



3-ary max-heap: children 3i+1, 3i+2, 3i+3

- **d-ary heap**: each node has d children; parent (i − 1) / d. Height $\log_d n$, so insertion gets faster, but removal compares d children per level.
- **Binomial / Fibonacci heaps**: a forest of trees; *merging* two queues and decreasing a key are fast (Dijkstra, Prim).

Min-heap type	insert	removeMin	decrease key	merge
Binary heap	O(log n)	O(log n)	O(log n)	O(n)
d-ary heap	$O(\log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	O(n)
Binomial heap	O(log n)	O(log n)	O(log n)	O(log n)
Fibonacci heap	O(1)	O(log n)*	O(1)*	O(1)

* amortized. In practice the binary heap is often the fastest, thanks to its simplicity and cache-friendly array layout.

Java's Built-In PriorityQueue

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5); pq.offer(1); pq.offer(8);
pq.peek();   // 1  (min-heap, O(1))
pq.poll();   // 1  (O(log n))

// reverse comparator for a max-heap
PriorityQueue<Integer> maxPq =
    new PriorityQueue<>(Comparator.reverseOrder());

// patients: lower priority number goes first
PriorityQueue<Patient> er = new PriorityQueue<>(
    Comparator.comparingInt(p -> p.priority));
```

- Internally it is an array-based **binary min-heap**.
- `offer` / `poll` are O(log n); `peek` and `size` are O(1); `remove(obj)` and `contains` are O(n).
- **Not thread-safe**; for multithreaded code, use `PriorityBlockingQueue`.
- No FIFO guarantee among equal priorities.

Code Trace: what `PriorityQueue` does inside

new PriorityQueue

size = 0



An empty PriorityQueue. Internally it keeps a **binary min-heap** in an array (`queue`), with the smallest item at the root.

Step 1 / 11

```
PriorityQueue<Integer> pq =
    new PriorityQueue<>();        // min-heap
pq.offer(5);
pq.offer(1);
pq.offer(8);
System.out.println(pq.peek());    // look
System.out.println(pq.poll());    // remove
System.out.println(pq.poll());
System.out.println(pq.size());
```

VARIABLES

`pq.size() = 0` `queue = []`

OUTPUT

Code Trace: what `PriorityQueue` does inside



offer(5)

size = 1



offer(5): the heap was empty, so 5 becomes the root (index 0).

Step 2 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
▶ pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

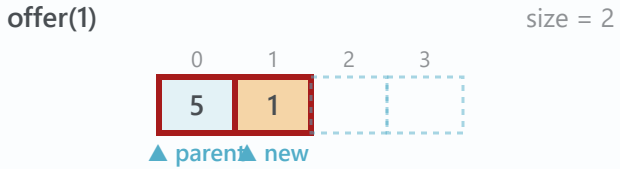
VARIABLES

`pq.size() = 1` `queue = [5]`

OUTPUT

Code Trace: what `PriorityQueue` does inside





offer(1): added at index 1; its parent 5 > 1 → min-heap order is broken.

Step 3 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
▶ pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

VARIABLES

`pq.size() = 2` `queue = [5, 1]`

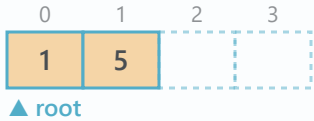
OUTPUT

Code Trace: what `PriorityQueue` does inside



offer(1)

size = 2



siftUp (up-heap): 1 moves up to the root → queue = [1, 5]. Cost: O(log n).

Step 4 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
▶ pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

VARIABLES

`pq.size() = 2` `queue = [1, 5]`

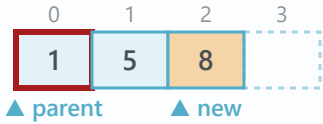
OUTPUT

Code Trace: what `PriorityQueue` does inside



offer(8)

size = 3



offer(8): added at index 2; parent 1 ≤ 8 → no up-heap needed.

Step 5 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
▶ pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

VARIABLES

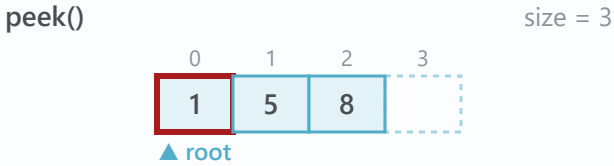
`pq.size() = 3`

`queue = [1, 5, 8]`

OUTPUT

Code Trace: what `PriorityQueue` does inside





peek(): returns the root, 1, but does **not remove** it → O(1).

Step 6 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
▶ System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

VARIABLES

`pq.size() = 3`

`queue = [1, 5, 8]`

OUTPUT

```
1
```

Code Trace: what `PriorityQueue` does inside



poll()

size = 2



poll(): takes the root, 1; the last item, 8, moves to the root; size = 2.

Step 7 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
▶ System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

VARIABLES

`pq.size() = 2` `queue = [8, 5]` `returned = 1`

OUTPUT

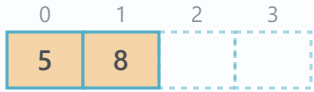
```
1
```

Code Trace: what `PriorityQueue` does inside



poll()

size = 2



siftDown (down-heap): 8 swaps with its smaller child → queue = [5, 8]. 1 is printed.

Step 8 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
▶ System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

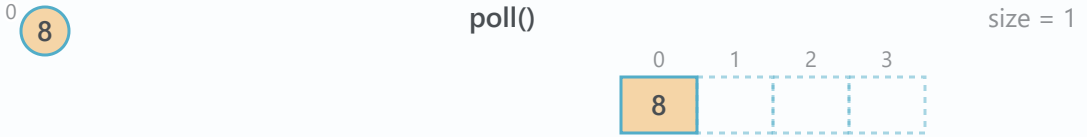
VARIABLES

`pq.size() = 2` `queue = [5, 8]` `returned = 1`

OUTPUT

```
1
1
```

Code Trace: what `PriorityQueue` does inside



poll(): takes the root, 5; the last item, 8, moves to the root; size = 1.

Step 9 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
▶ System.out.println(pq.poll());
9 System.out.println(pq.size());
```

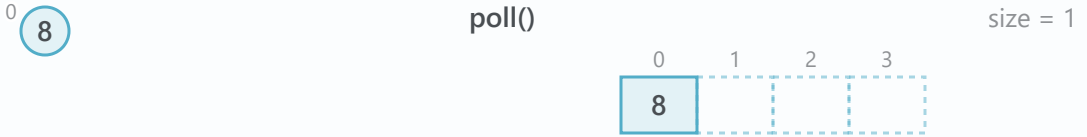
VARIABLES

`pq.size() = 1` `queue = [8]` `returned = 5`

OUTPUT

```
1
1
```

Code Trace: what `PriorityQueue` does inside



One item is left, so no siftDown is needed. 5 is printed.

Step 10 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
▶ System.out.println(pq.poll());
9 System.out.println(pq.size());
```

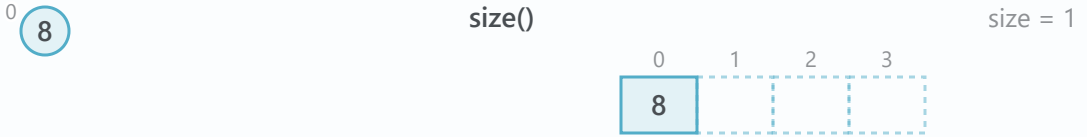
VARIABLES

`pq.size() = 1` `queue = [8]` `returned = 5`

OUTPUT

```
1
1
5
```

Code Trace: what `PriorityQueue` does inside



size() = 1 (only 8 is left) → O(1). Console output: 1 (peek), 1, 5 (poll), 1 (size).

Step 11 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // look
7 System.out.println(pq.poll());    // remove
8 System.out.println(pq.poll());
▶ System.out.println(pq.size());
```

VARIABLES

`pq.size() = 1` `queue = [8]`

OUTPUT

```
1
1
5
1
```

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`

a

0	1	2	3	4	5
5	1	9	3	7	6

pq (min-heap)

(empty)

queue[] size = 0

0	1	2	3

root = smallest of the k largest

k = 3: we keep a **min-heap** of at most k items. Its root is the *smallest* of the k largest items seen so far.

Step 1 / 14

```
1  static int kthLargest(int[] a, int k) {
▶    PriorityQueue<Integer> pq = new PriorityQueue<>();
3    for (int x : a) {
4      pq.offer(x);                    // O(log k)
5      if (pq.size() > k) pq.poll();  // drop the
   smallest
6    }
7    return pq.peek();       // smallest of the k
   largest
8  }
```

VARIABLES

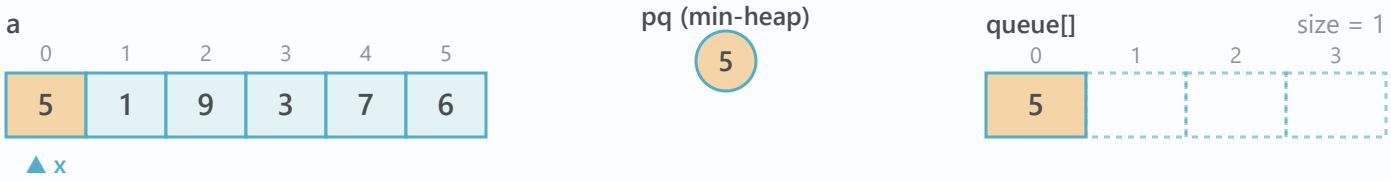
`k = 3` `pq = []`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



x = 5: **offer(5)** → added at the end, up-heap. size = 1.

Step 2 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();       // smallest of the k
  largest
8 }
```

VARIABLES

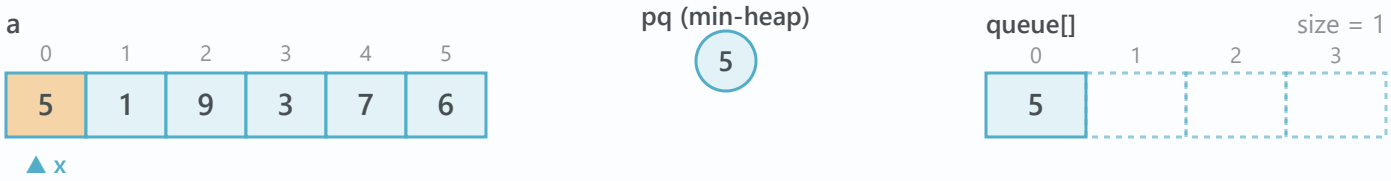
`k = 3` `x = 5` `pq = [5]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



size = 1 ≤ k = 3 → nothing to drop yet.

Step 3 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the smallest
6   }
7   return pq.peek();        // smallest of the k largest
8 }
```

VARIABLES

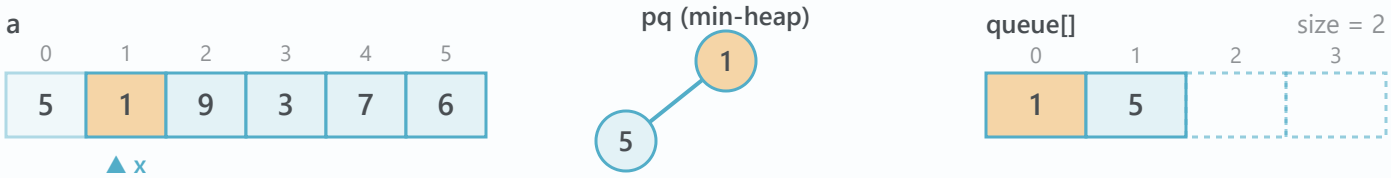
`k = 3` `x = 5` `pq = [5]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



root = smallest of the k largest

x = 1: **offer(1)** → added at the end, up-heap (it swam up: 1 swap). size = 2.

Step 4 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

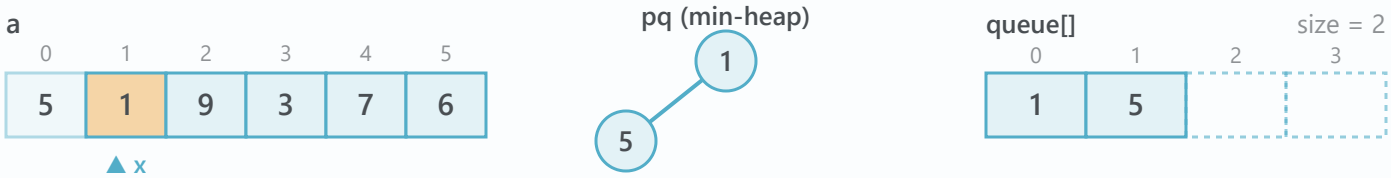
`k = 3` `x = 1` `pq = [1, 5]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



root = smallest of the k largest

size = 2 ≤ k = 3 → nothing to drop yet.

Step 5 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

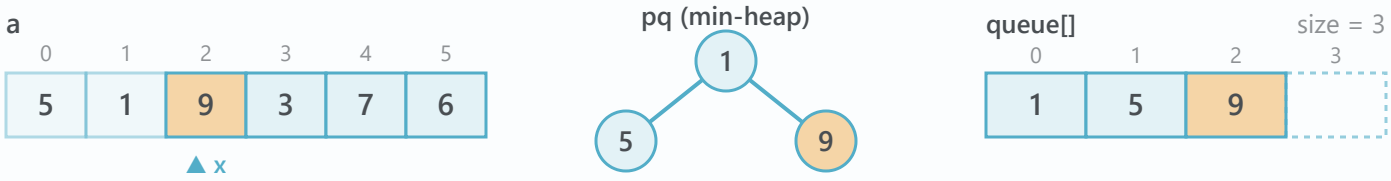
`k = 3` `x = 1` `pq = [1, 5]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



x = 9: **offer(9)** → added at the end, up-heap. size = 3.

Step 6 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
    for (int x : a) {
      pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

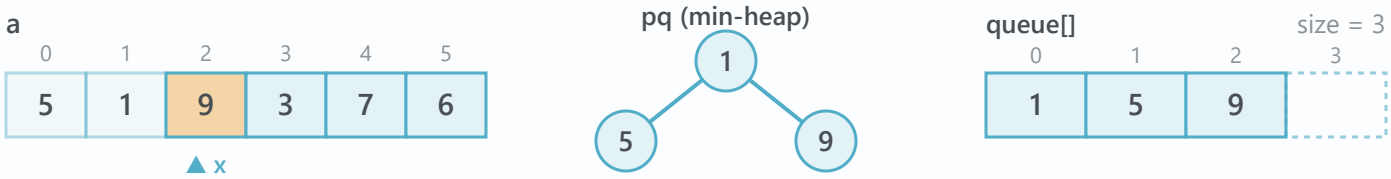
`k = 3` `x = 9` `pq = [1, 5, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



size = 3 ≤ k = 3 → nothing to drop yet.

Step 7 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

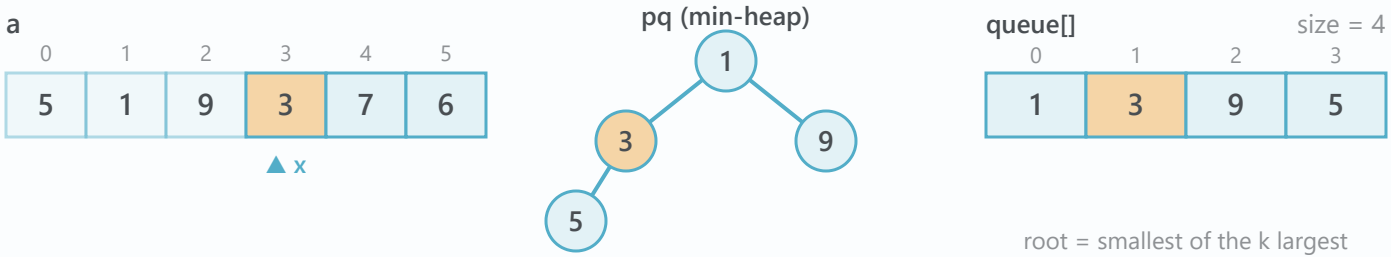
`k = 3` `x = 9` `pq = [1, 5, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



x = 3: **offer(3)** → added at the end, up-heap (it swam up: 1 swap). size = 4.

Step 8 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

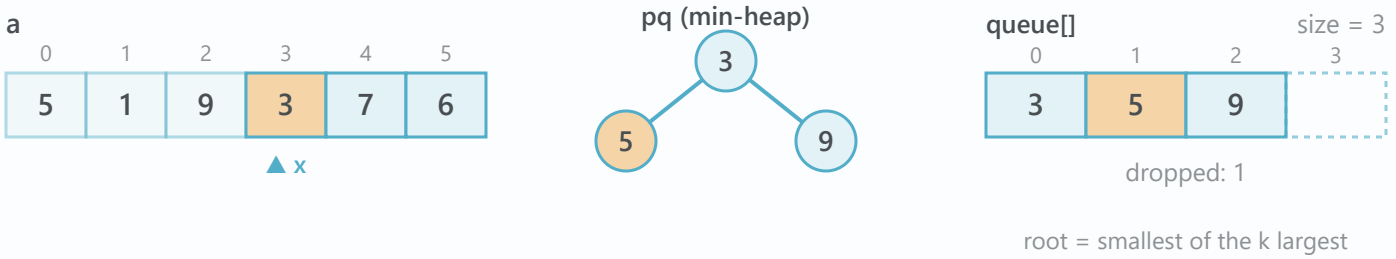
`k = 3` `x = 3` `pq = [1, 3, 9, 5]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



size = 4 > k → **poll()**: the smallest, 1, is dropped (it cannot be among the k largest). Last item to the root, down-heap.

Step 9 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the smallest
6   }
7   return pq.peek();        // smallest of the k largest
8 }
```

VARIABLES

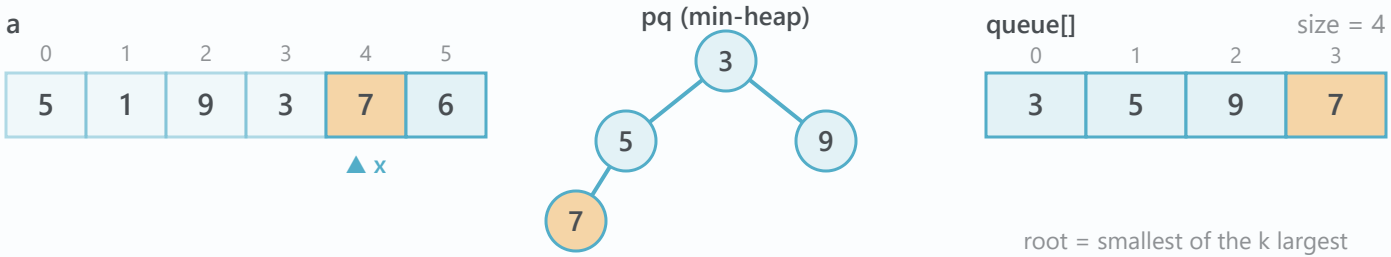
`k = 3` `x = 3` `pq = [3, 5, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



x = 7: **offer(7)** → added at the end, up-heap. size = 4.

Step 10 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
  smallest
6   }
7   return pq.peek();        // smallest of the k
  largest
8 }
```

VARIABLES

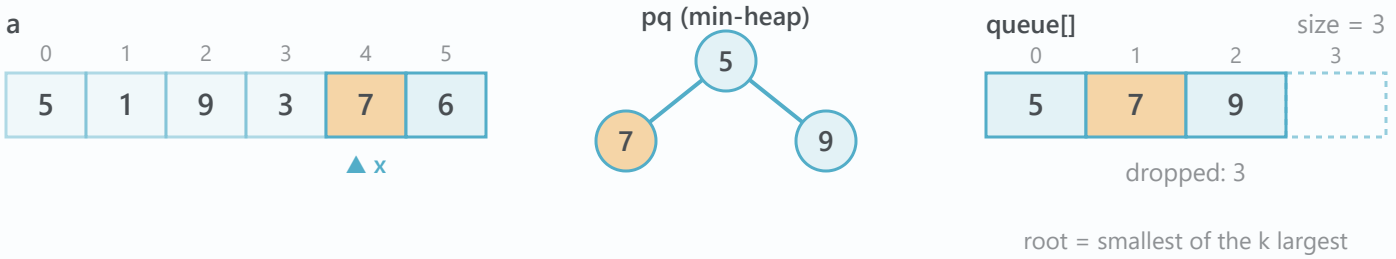
`k = 3` `x = 7` `pq = [3, 5, 9, 7]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



size = 4 > k → **poll()**: the smallest, 3, is dropped (it cannot be among the k largest). Last item to the root, down-heap.

Step 11 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the smallest
6   }
7   return pq.peek();       // smallest of the k largest
8 }
```

VARIABLES

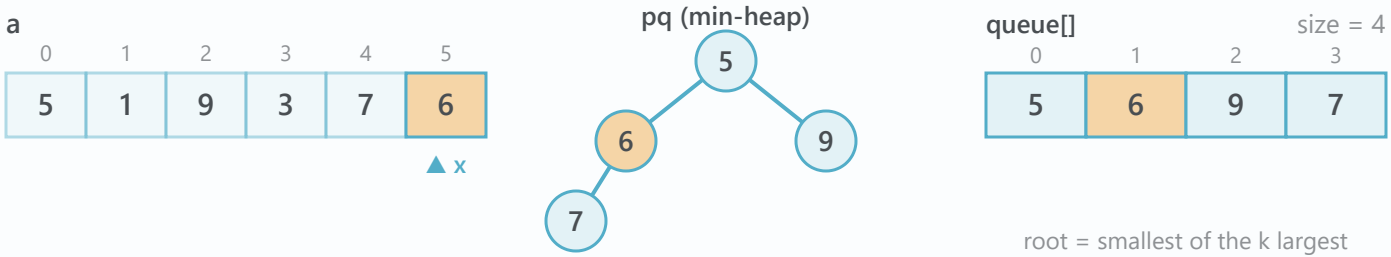
`k = 3` `x = 7` `pq = [5, 7, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



x = 6: **offer(6)** → added at the end, up-heap (it swam up: 1 swap). size = 4.

Step 12 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the
    smallest
6   }
7   return pq.peek();        // smallest of the k
    largest
8 }
```

VARIABLES

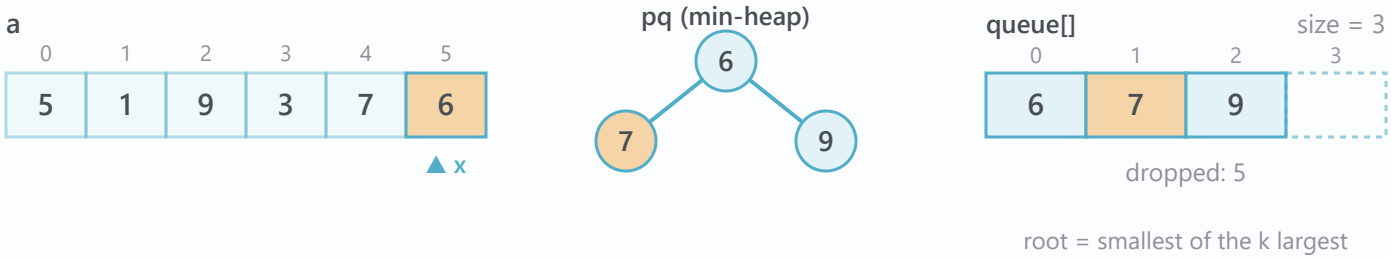
`k = 3` `x = 6` `pq = [5, 6, 9, 7]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



size = 4 > k → **poll()**: the smallest, 5, is dropped (it cannot be among the k largest). Last item to the root, down-heap.

Step 13 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
▶     if (pq.size() > k) pq.poll();  // drop the smallest
6   }
7   return pq.peek();       // smallest of the k largest
8 }
```

VARIABLES

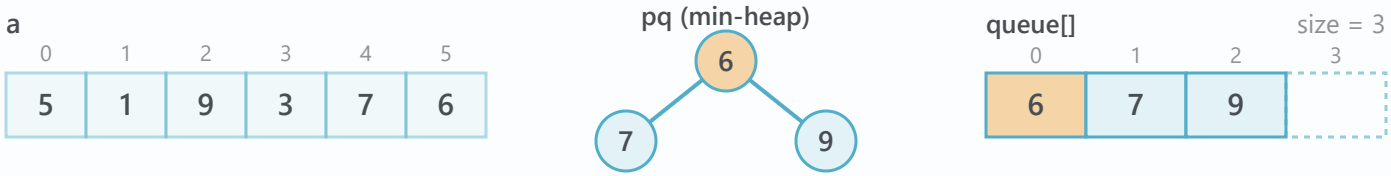
`k = 3` `x = 6` `pq = [6, 7, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: `kthLargest([5, 1, 9, 3, 7, 6], 3)`



Loop done: pq holds the three largest items {6, 7, 9}. **peek() = 6** → the 3rd largest. Total **O(n log k)**, memory O(k).

Step 14 / 14

```
1 static int kthLargest(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // drop the smallest
6   }
▶   return pq.peek();       // smallest of the k largest
8 }
```

VARIABLES

`k = 3` `pq = [6, 7, 9]`

CALL STACK

`kthLargest(a, 3)`

`main()`

Application: A Priority Scheduler



Strict priority

The CPU always picks the process at the front of the highest-priority non-empty queue.

FIFO within a level

Processes with the same priority run in arrival order in their own queue (round robin if needed).

Starvation and aging

Low-priority processes may never get to run (starvation). Raising a process's priority as it waits (aging) fixes this.

Simulator: Priority Scheduler Simulator (three levels, a time quantum, optional aging).

SUMMARY

Summary

Priority queue
insert, min, removeMin. Items leave in priority order.

Heap property
Complete binary tree + parent ≥ child (max) or ≤ child (min).

Array representation
`children 2i+1, 2i+2 · parent ⌊(i−1)/2⌋`

Operation	Heap	How?
peek (max/min)	O(1)	heap[0]
insert	O(log n)	put at heap[n], up-heap
removeMax / removeMin	O(log n)	move the last item to the root, down-heap
building a heap	O(n)	down-heap from ⌊n/2⌋ − 1 down to 0
heap sort	O(n log n)	build + move the root to the end n − 1 times

SUMMARY

Interactive Simulators

Binary Heap Simulator

Insert, remove the max or min, and watch up-heap and down-heap in both the tree and the array view.

Priority Queue Simulator

The ADT's behavior as you insert and remove prioritized items.

Priority Scheduler Simulator

Operating-system-style priority scheduling.

SUMMARY

Common Mistakes

"A heap is sorted"

It is not; only the root is determined. After inserting 3, 1, 2 into a `PriorityQueue`, `println(pq)` → `[1, 3, 2]`.

Index formulas

0-based: 2i+1, 2i+2, (i−1)/2. 1-based: 2i, 2i+1, i/2. Mixing the two scrambles the tree.

Right-child bound

Forget the `j + 1 < n` check and you read an item outside the heap (one already removed).

Wrong child

In a max-heap, down-heap swaps with the *larger* child; picking the smaller one breaks the order.

Searching a heap

`contains` and `remove(obj)` are O(n). Fast access needs a locator or a map.

Wrong direction

`PriorityQueue` is a *min*-heap by default; for a max-heap, use `Comparator.reverseOrder()`.

SUMMARY

Sample Problems

1. Insert 5, 12, 8, 3, 15, 10 in this order into an empty max-heap. What is the final array?
2. Apply `removeMax()` once to the heap from problem 1. What is returned, and what is the new array?
3. Turn the array {3, 1, 6, 5, 2, 4} into a max-heap bottom-up.
4. In a 0-based array, at which indices are the parent and the children of the node at index 13?
5. Is {10, 7, 9, 8, 2, 5} a max-heap? If not, why not?
6. What does it cost to find the *largest* item in a min-heap?

▶ **Show answers**

SUMMARY

Exercises

1. Apply heap sort to [4, 10, 3, 5, 1]. What is the array after each step?
2. What is the height of a binary heap with 100 items? At most how many swaps can one insertion make?
3. In a max-heap with distinct keys, at which indices can the 2nd and 3rd largest items be?
4. Rewrite `isHeap` for a min-heap. What does it return for an empty array and for a one-item array?
5. How would you merge k sorted lists with N items in total using a min-heap? What does it cost?
6. In the max-heap [9, 7, 8, 3, 5, 6, 4], what is the array after `changeKey(5, 10)`?

▶ **Answers and hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.
- Oracle, Java SE 17 `PriorityQueue` API documentation.

The End

Chapter 11: Priority Queues and Heaps

Next deck: Chapter 12: Maps and Hash Tables · Study questions · Simulators: Binary Heap Simulator, Priority Queue Simulator, Priority Scheduler Simulator · Glossary