Chapter 13: Priority Queues and Heaps

Data Structures and Algorithms with C

Priority queue · Max / min heap · Complete binary tree in an array · Insert: sift up · Delete: sift down · Comparison · Heapify · d-ary

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_13_Priority_Queues_and_Heaps.html

Learning Outcomes

Define

Define priority queues and max/min heaps; tell a heap apart from a binary search tree.

Build the representation

Store a complete binary tree in a 1-based array: parent `i/2`, children `2i`, `2i+1`.

Insert and delete

Trace `insert_max_heap` and `delete_max_heap` step by step.

Analyze

Show why insert and delete are O(lg n) and why the best case is Ω(1).

Compare

Compare heaps with unsorted/sorted arrays and lists using a cost table.

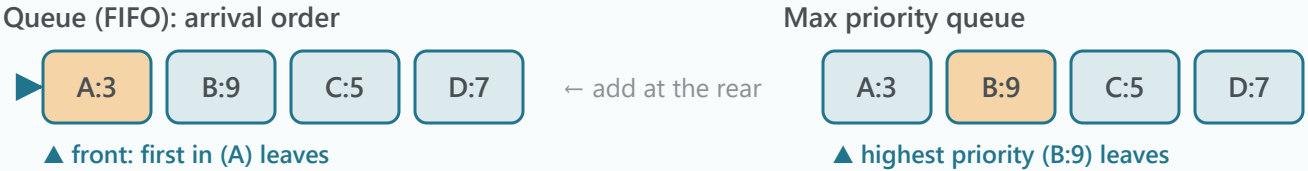
Go further

Explain why bottom-up heap construction (heapify) is O(n); meet the d-ary heap.

Priority Queue: The Highest-Priority Item Leaves First

The item removed first is chosen not by arrival order but by the **highest** or **lowest** value.

- **Max priority queue**: the highest value is served first.
- **Min priority queue**: the lowest value is served first.



Instructor *"We say priority queue; let's see how well we know the queue. Sometimes knowing things can get in the way. In the queue we know, you add at the rear and take from the front. This one is different..."*

Why Do We Need Priority Queues?

1 · Photocopier

Every user pays a flat fee (3 pages or 30, same price). To maximize revenue, serve the **shortest** job first → a **min** priority queue.

2 · Airport

Among planes waiting to land, the one with the **least fuel** must go first → a **min** priority queue.

Real systems work the same way: an operating system, for example, keeps processes in a priority queue.

Instructor *"We want to maximize our revenue. Giving priority to the shortest usage time maximizes it."*

Instructor *"We can't say 'You're at the back of the queue. You're low on fuel, but that's the queue. What can we do?'"*

Abstract Data Type: Priority Queue

```
/* max priority queue interface */
bool is_empty(void);
bool is_full(void);
void insert(element item);      /* O(lg n) */
bool pop_max(element *out); /* pop + return */
bool peek_max(element *top);  /* no removal */
```

Insert

The item goes into the structure together with its priority.

Delete

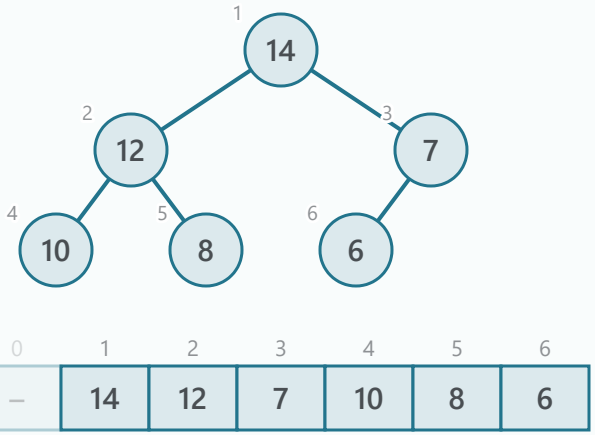
The highest-priority item is returned and removed from the structure.

A priority queue is an **interface**; many data structures can implement it. The **heap** is just one, but it balances both: insert and delete in O(lg n).

Instructor *"Once you grasp where and how to use it, the heap is a very, very important data structure."*

HEAP STRUCTURE

A Heap Lives in an Array as a Complete Binary Tree



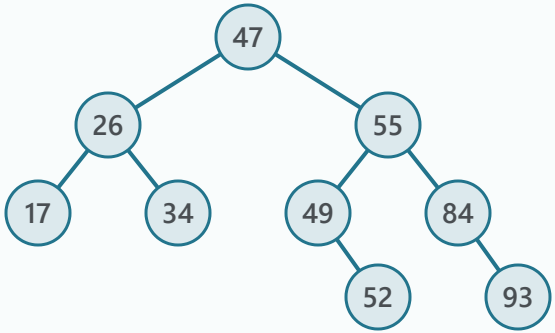
Index 0 of the array is unused; the root is `heap[1]`.

Node	Index
root	`1`
parent of i	`i / 2`
left child	`2i`
right child	`2i + 1`
last item	`n`

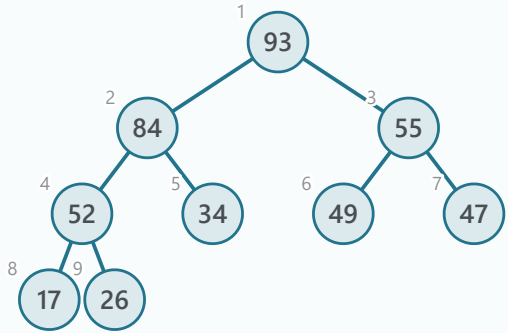
In a 0-based array: parent `(i−1)/2`, children `2i+1`, `2i+2`.

The array is **not sorted**; the only guarantee is that the item at index 1 has the highest priority (is removed first). A complete binary tree has no gaps, so no pointers are needed.

A Heap Means a Complete Binary Tree, Not a Search Tree



Binary search tree



A max heap with the same keys

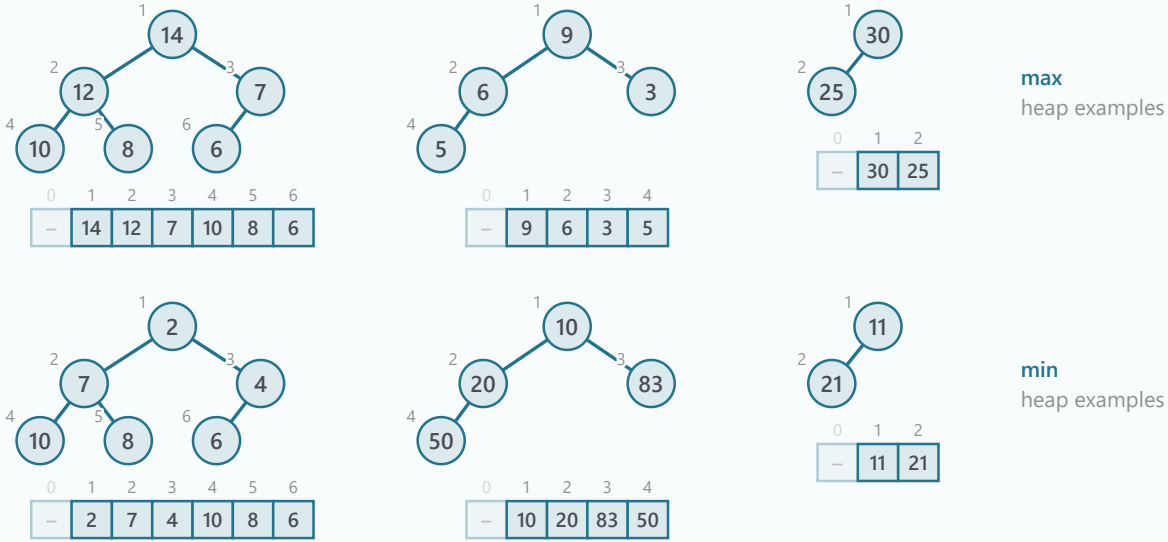
- **Complete binary tree:** every level is full, and the last level fills from left to right.
- In a search tree *left* < *root* < *right*; in a heap only *parent* ≥ *child* (no order between left and right).

Instructor *"'Sir, could you draw a binary search tree?' By popular demand, let's draw one; we are not confusing the two!"*

Definition: Max Heap and Min Heap

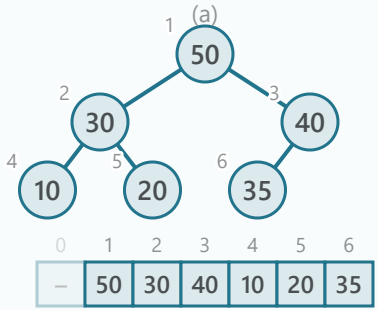
Max heap: a complete binary tree in which every node's key is **no smaller** than its children's.

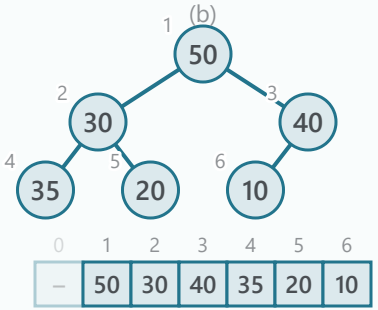
Min heap: a complete binary tree in which every node's key is **no larger** than its children's.

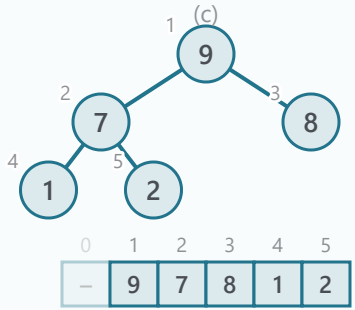


These definitions let us implement max/min priority queues.

Quick Check: Which Is a Max Heap?







Is `heap[k/2] ≥ heap[k]` for every `k ≥ 2`? Only parent–child pairs matter; no order is required between siblings or cousins.

▶ **Answer**

Definitions in C: Array, Counter and Macros

```c
#define MAX_ELEMENTS 200   /* max items + 1 */
#define HEAP_FULL(n) ((n) == MAX_ELEMENTS - 1)
#define HEAP_EMPTY(n)  (!(n))

typedef struct {
  int x;          /* priority (key) */
  /* other fields */
} element;

element heap[MAX_ELEMENTS];  /* [0] unused */
int n = 0;                   /* item count */
```

Capacity

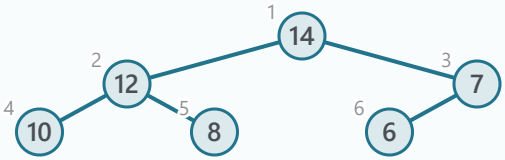
Indices 1..199 are used; the heap is full when `n == 199`.

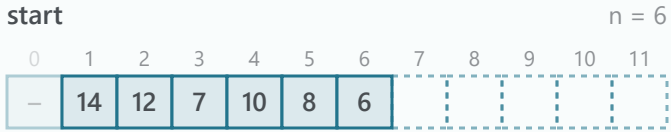
Parentheses in macros

`(n)` prevents precedence bugs in calls like `HEAP_EMPTY(a+b)`.

Common mistake: declaring the array as `element heap[MAX_ELEMENTS - 1]` (199 items, indices 0..198). Before the heap counts as full, `n` can reach 199 and `heap[199]` writes past the end; the size must be `MAX_ELEMENTS`.

Inserting into a Max Heap: Take the New Slot, Then Sift



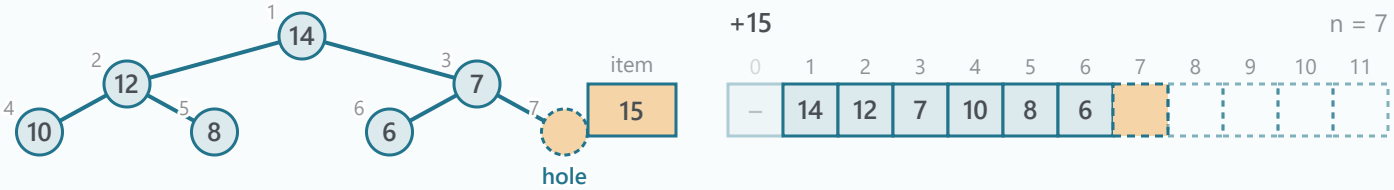


Start: `14, 12, 7, 10, 8, 6`. We will insert 15, 109, 107, 3, 15 in turn.

Step 1 / 5

Instructor *"The slot for the new item is certainly there. No doubt about it. It goes there first. Then, uh-oh! It had to be larger than its child. So we move it to the right position."*

Inserting into a Max Heap: Take the New Slot, Then Sift

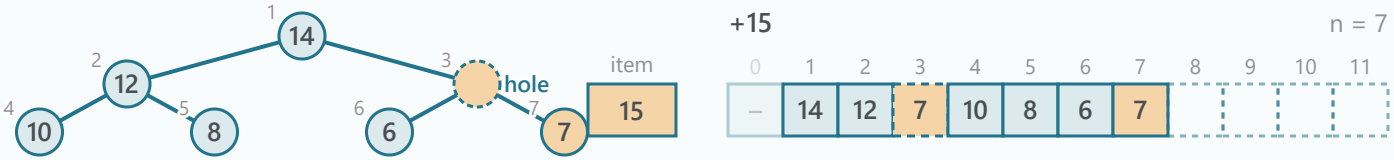


First a new slot opens: the first free position of the complete tree (**7**).

Step 2 / 5

Instructor *"The slot for the new item is certainly there. No doubt about it. It goes there first. Then, uh-oh! It had to be larger than its child. So we move it to the right position."*

Inserting into a Max Heap: Take the New Slot, Then Sift

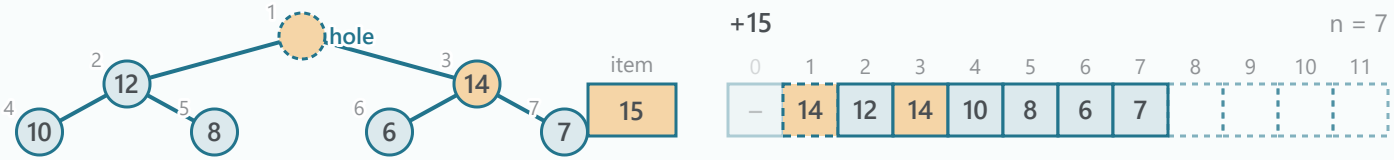


15 > 7: parent **7** moves down; the hole rises to position 3.

Step 3 / 5

Instructor *"The slot for the new item is certainly there. No doubt about it. It goes there first. Then, uh-oh! It had to be larger than its child. So we move it to the right position."*

Inserting into a Max Heap: Take the New Slot, Then Sift

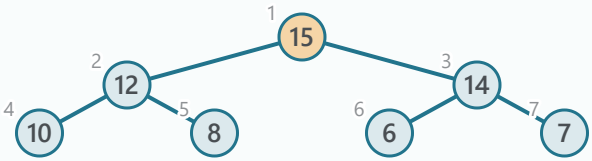


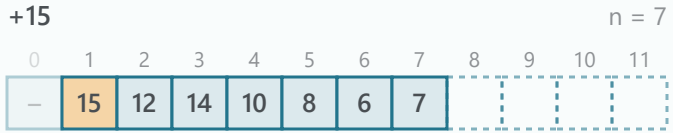
15 > 14: parent **14** moves down; the hole rises to position 1.

Step 4 / 5

Instructor *"The slot for the new item is certainly there. No doubt about it. It goes there first. Then, uh-oh! It had to be larger than its child. So we move it to the right position."*

Inserting into a Max Heap: Take the New Slot, Then Sift





15 is written at position 1. The heap property holds again.

Step 5 / 5

Instructor *"The slot for the new item is certainly there. No doubt about it. It goes there first. Then, uh-oh! It had to be larger than its child. So we move it to the right position."*

`insert_max_heap`: Shifting, Not Swapping

```
void insert_max_heap(element item) {
  if (HEAP_FULL(n)) {
    heap_full();              /* error message */
    return;
  }
  int i = ++n;
  while (i != 1) {             /* i == 1: at root */
    if (item.x <= heap[i / 2].x) break;
    heap[i] = heap[i / 2];  /* move parent down */
    i /= 2;                    /* go up a level */
  }
  heap[i] = item;
}
```

The hole method

No swaps: larger ancestors **shift down** one level each, and the item is written once at the end (a swap costs 3 assignments, a shift 1).

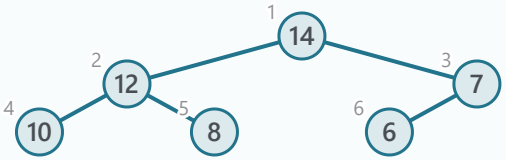
Loop condition

`i ≠ 1` : the root has no parent. When `item.x ≤ parent` , the slot is found.

Full heap

`heap_full()` is an error-message routine; we return without inserting.

Code Trace: `insert_max_heap(15)`



insert(15) n = 6

0	1	2	3	4	5	6	7	8	9	10	11
–	14	12	7	10	8	6					

insert(15) is called; the heap holds 6 items: `14, 12, 7, 10, 8, 6`.

Step 1 / 13

```
▶ void insert_max_heap(element item) {
2   if (HEAP_FULL(n)) {
3     heap_full();              /* error message */
4     return;
5   }
6   int i = ++n;
7   while (i != 1) {            /* i == 1: at root */
8     if (item.x <= heap[i / 2].x) break;
9     heap[i] = heap[i / 2];  /* move parent down */
10    i /= 2;                   /* go up a level */
11  }
12  heap[i] = item;
13 }
```

VARIABLES

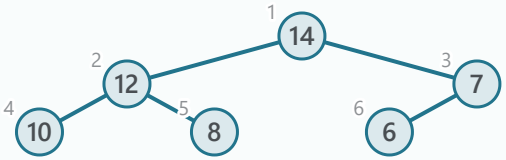
`item.x = 15` `n = 6`

CALL STACK

`insert_max_heap(15)`

`main()`

Code Trace: `insert_max_heap(15)`



insert(15) n = 6

0	1	2	3	4	5	6	7	8	9	10	11
–	14	12	7	10	8	6					

`HEAP_FULL(6)` is false: 6 ≠ 199, there is room.

Step 2 / 13

```
1  void insert_max_heap(element item) {
▶    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

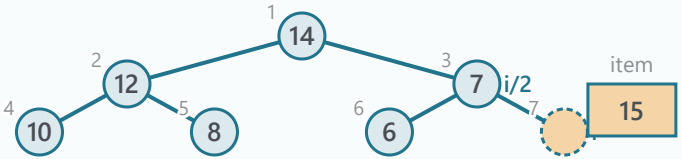
`item.x = 15` `n = 6`

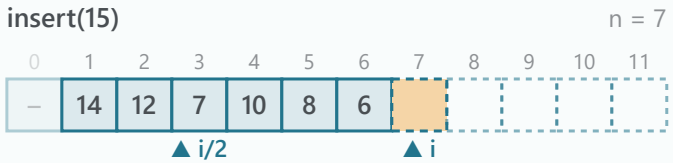
CALL STACK

`insert_max_heap(15)`

`main()`

Code Trace: `insert_max_heap(15)`





i = ++n = 7: the first free position of the complete tree. The slot is certainly there, but first we compare with the parent; `item` is not written yet.

Step 3 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
▶    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

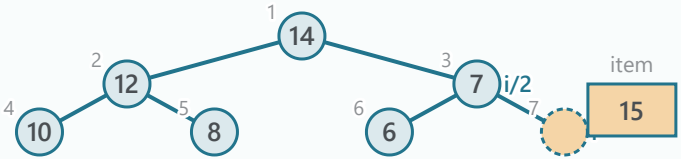
VARIABLES

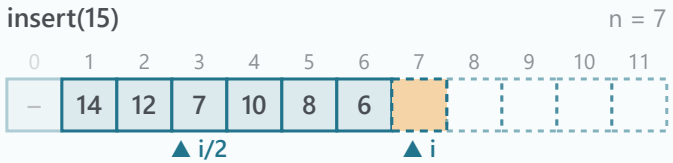
`item.x = 15` `n = 7` `i = 7` `i/2 = 3` `heap[i/2].x = 7`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





i = 7 ≠ 1: not at the root yet; enter the loop.

Step 4 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

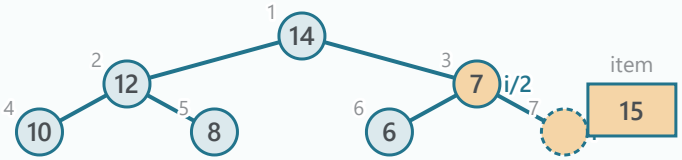
`item.x = 15` `n = 7` `i = 7` `i/2 = 3` `heap[i/2].x = 7`

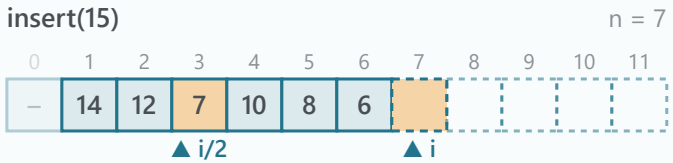
CALL STACK

`insert_max_heap(15)`

`main()`

Code Trace: `insert_max_heap(15)`





15 ≤ 7? No. The new item is larger than its parent: the parent must move down.

Step 5 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
▶      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

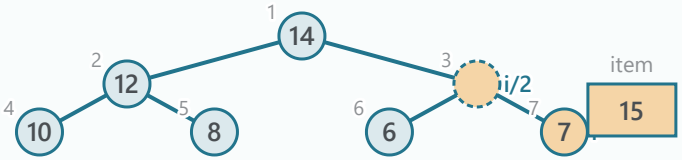
VARIABLES

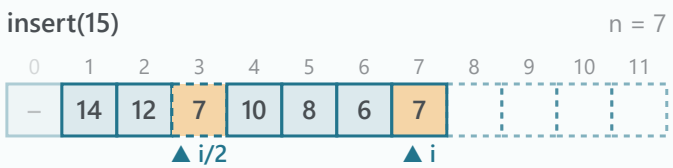
`item.x = 15` `n = 7` `i = 7` `i/2 = 3` `heap[i/2].x = 7`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





heap[7] = heap[3]: the parent (7) shifted down one level. No swap; the hole (orange) moved up.

Step 6 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
▶      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

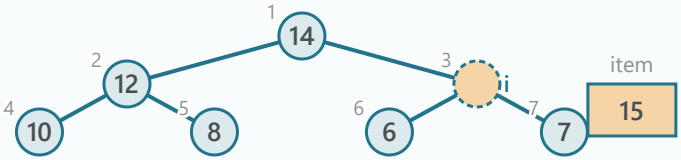
VARIABLES

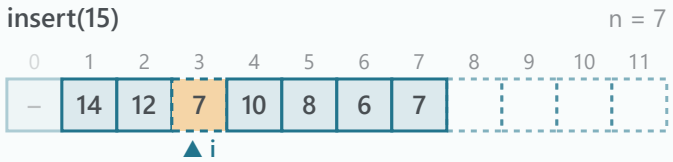
`item.x = 15` `n = 7` `i = 7` `i/2 = 3` `heap[i/2].x = 7`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





i /= 2 → i = 3: move to the parent above.

Step 7 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
▶      i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

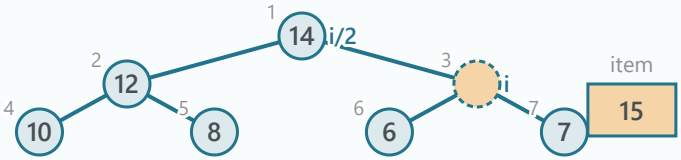
VARIABLES

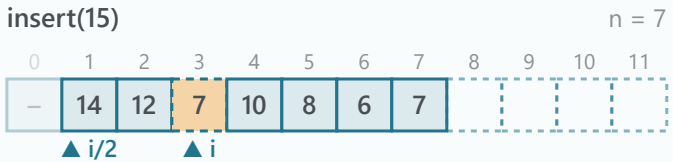
`item.x = 15` `n = 7` `i = 3` `i/2 = 1` `heap[i/2].x = 14`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





i = 3 ≠ 1: not at the root yet; enter the loop.

Step 8 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                    /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

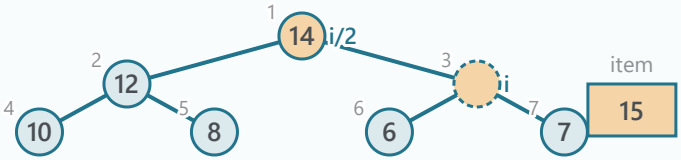
`item.x = 15` `n = 7` `i = 3` `i/2 = 1` `heap[i/2].x = 14`

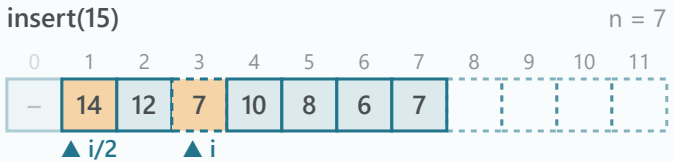
CALL STACK

`insert_max_heap(15)`

`main()`

Code Trace: `insert_max_heap(15)`





15 ≤ 14? No. The new item is larger than its parent: the parent must move down.

Step 9 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
▶      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

`item.x = 15` `n = 7` `i = 3`

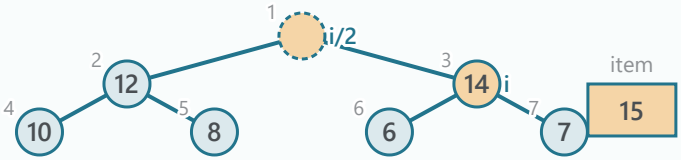
`i/2 = 1` `heap[i/2].x = 14`

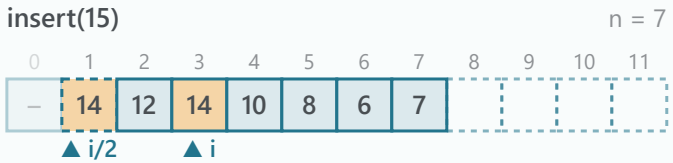
CALL STACK

`insert_max_heap(15)`

`main()`

Code Trace: `insert_max_heap(15)`





heap[3] = heap[1]: the parent (14) shifted down one level. No swap; the hole (orange) moved up.

Step 10 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
▶      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

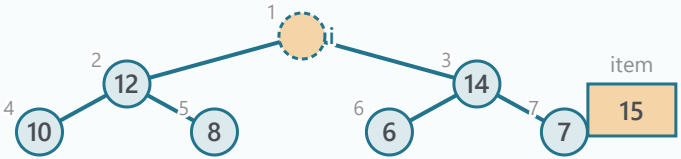
VARIABLES

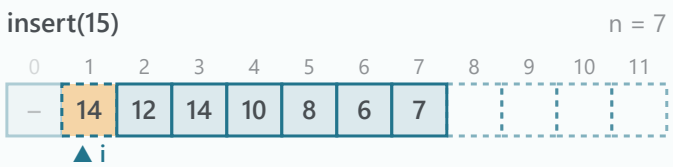
`item.x = 15` `n = 7` `i = 3` `i/2 = 1` `heap[i/2].x = 14`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





i /= 2 → i = 1: move to the parent above.

Step 11 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
▶      i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

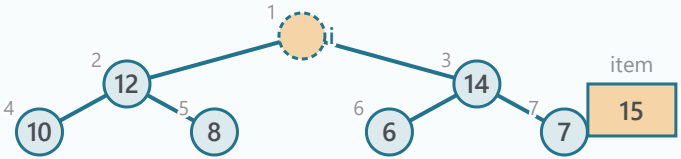
VARIABLES

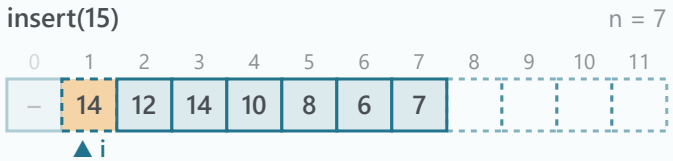
`item.x = 15` `n = 7` `i = 1`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





i == 1: reached the root; the loop ends.

Step 12 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

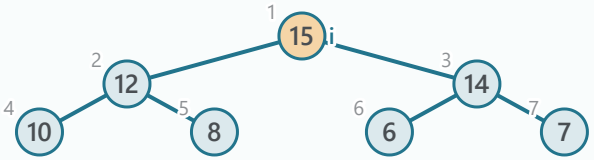
VARIABLES

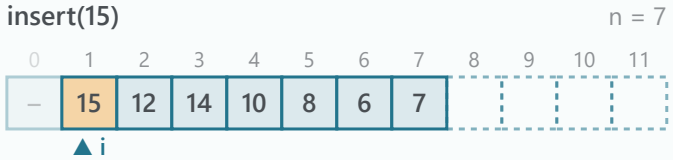
`item.x = 15` `n = 7` `i = 1`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(15)`





heap[1] = item: 15 is in place. Heap: `15, 12, 14, 10, 8, 6, 7` .

Step 13 / 13

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {             /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                    /* go up a level */
11   }
▶    heap[i] = item;
13 }
```

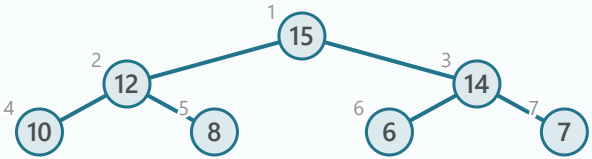
VARIABLES

`item.x = 15` `n = 7` `i = 1`

CALL STACK

- `insert_max_heap(15)`
- `main()`

Code Trace: `insert_max_heap(109)`



insert(109) n = 7

0	1	2	3	4	5	6	7	8	9	10	11
–	15	12	14	10	8	6	7				

insert(109): larger than every item. Worst case: it will climb to the root.

Step 1 / 17

```
▶ void insert_max_heap(element item) {
2   if (HEAP_FULL(n)) {
3     heap_full();              /* error message */
4     return;
5   }
6   int i = ++n;
7   while (i != 1) {            /* i == 1: at root */
8     if (item.x <= heap[i / 2].x) break;
9     heap[i] = heap[i / 2];  /* move parent down */
10    i /= 2;                   /* go up a level */
11  }
12  heap[i] = item;
13 }
```

VARIABLES

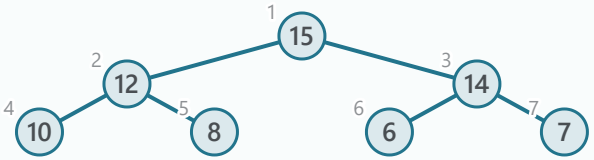
`item.x = 109` `n = 7`

CALL STACK

`insert_max_heap(109)`

`main()`

Code Trace: `insert_max_heap(109)`



insert(109) n = 7

0	1	2	3	4	5	6	7	8	9	10	11
–	15	12	14	10	8	6	7				

`HEAP_FULL(7)` is false: 7 ≠ 199, there is room.

Step 2 / 17

```
1  void insert_max_heap(element item) {
▶    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

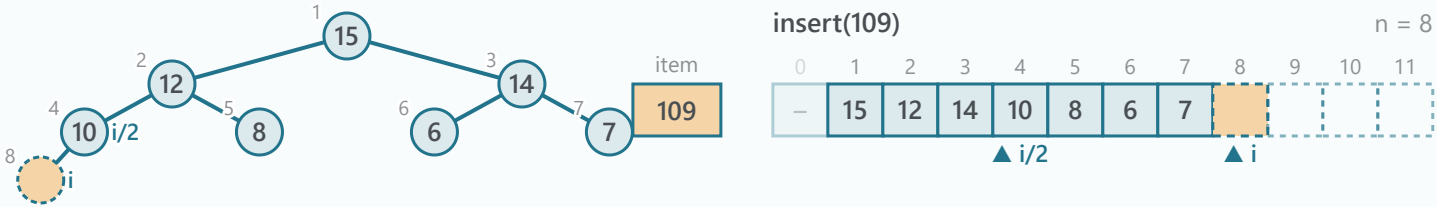
`item.x = 109` `n = 7`

CALL STACK

`insert_max_heap(109)`

`main()`

Code Trace: `insert_max_heap(109)`



i = ++n = 8: the first free position of the complete tree. The slot is certainly there, but first we compare with the parent; `item` is not written yet.

Step 3 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
▶    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

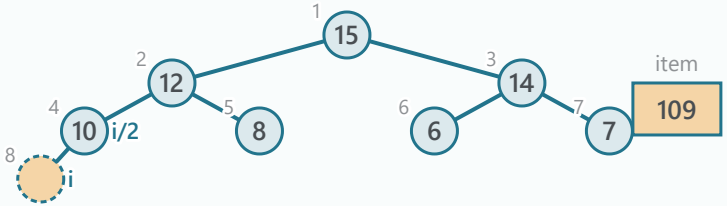
VARIABLES

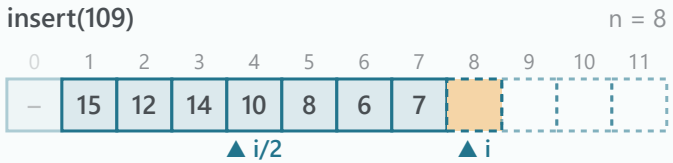
`item.x = 109` `n = 8` `i = 8` `i/2 = 4` `heap[i/2].x = 10`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





i = 8 ≠ 1: not at the root yet; enter the loop.

Step 4 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

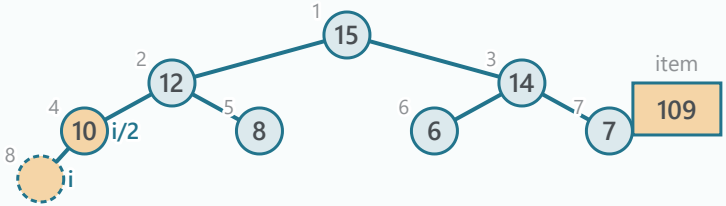
VARIABLES

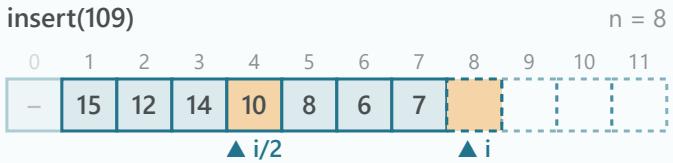
`item.x = 109` `n = 8` `i = 8` `i/2 = 4` `heap[i/2].x = 10`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





109 ≤ 10? No. The new item is larger than its parent: the parent must move down.

Step 5 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
▶      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

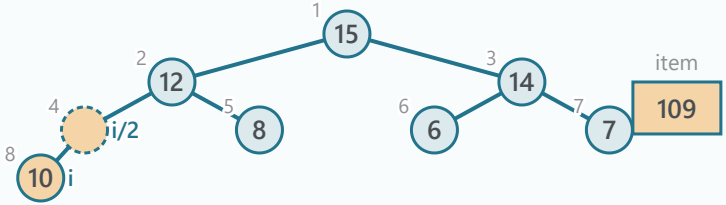
VARIABLES

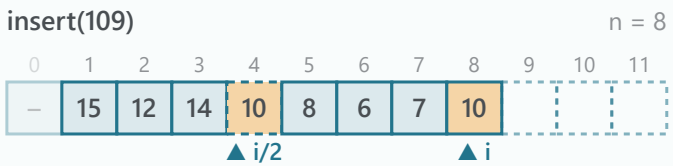
`item.x = 109` `n = 8` `i = 8` `i/2 = 4` `heap[i/2].x = 10`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





heap[8] = heap[4]: the parent (10) shifted down one level. No swap; the hole (orange) moved up.

Step 6 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
▶      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

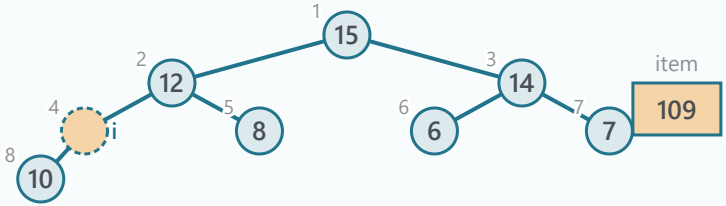
VARIABLES

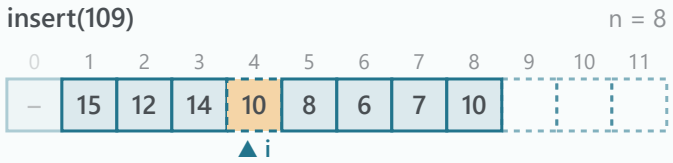
`item.x = 109` `n = 8` `i = 8` `i/2 = 4` `heap[i/2].x = 10`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





i /= 2 → i = 4: move to the parent above.

Step 7 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
▶      i /= 2;                  /* go up a level */
11   }
12   heap[i] = item;
13 }
```

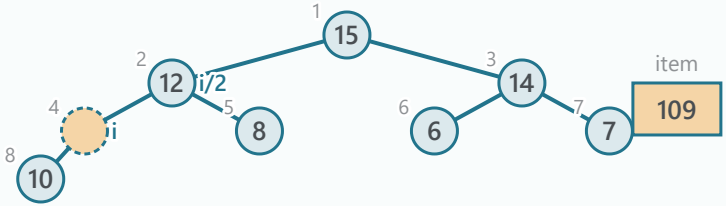
VARIABLES

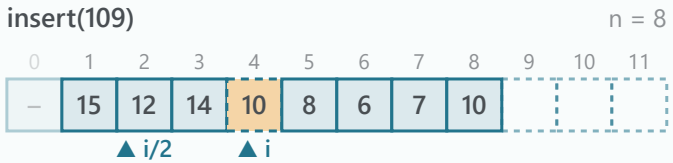
`item.x = 109` `n = 8` `i = 4` `i/2 = 2` `heap[i/2].x = 12`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





i = 4 ≠ 1: not at the root yet; enter the loop.

Step 8 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

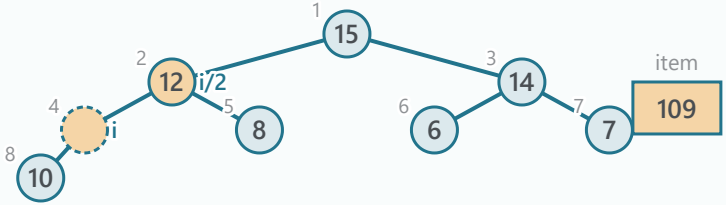
VARIABLES

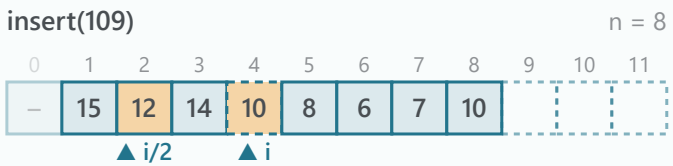
`item.x = 109` `n = 8` `i = 4` `i/2 = 2` `heap[i/2].x = 12`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





109 ≤ 12? No. The new item is larger than its parent: the parent must move down.

Step 9 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
▶      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

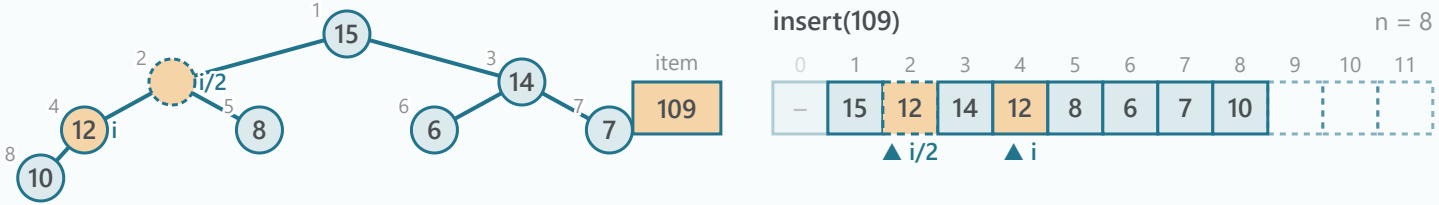
VARIABLES

`item.x = 109` `n = 8` `i = 4` `i/2 = 2` `heap[i/2].x = 12`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`



heap[4] = heap[2]: the parent (12) shifted down one level. No swap; the hole (orange) moved up.

Step 10 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
▶      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

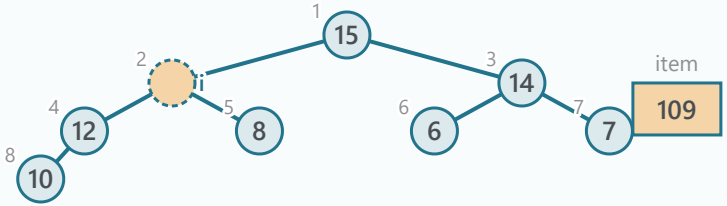
VARIABLES

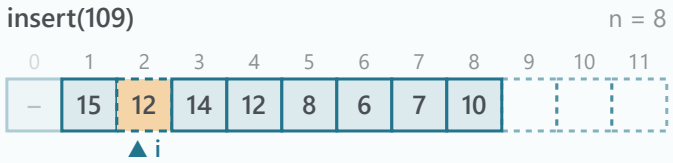
`item.x = 109` `n = 8` `i = 4` `i/2 = 2` `heap[i/2].x = 12`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





i /= 2 → i = 2: move to the parent above.

Step 11 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
▶      i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

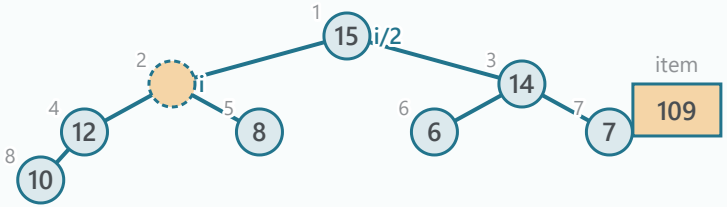
`item.x = 109` `n = 8` `i = 2` `i/2 = 1` `heap[i/2].x = 15`

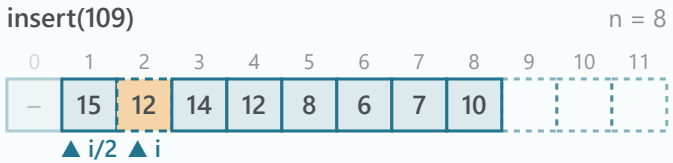
CALL STACK

`insert_max_heap(109)`

`main()`

Code Trace: `insert_max_heap(109)`





i = 2 ≠ 1: not at the root yet; enter the loop.

Step 12 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

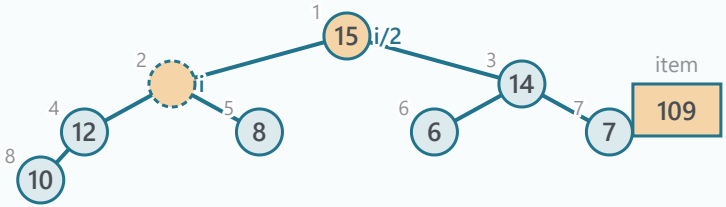
`item.x = 109` `n = 8` `i = 2` `i/2 = 1` `heap[i/2].x = 15`

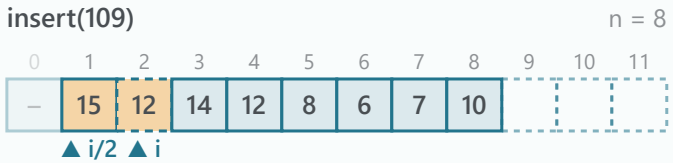
CALL STACK

`insert_max_heap(109)`

`main()`

Code Trace: `insert_max_heap(109)`





109 ≤ 15? No. The new item is larger than its parent: the parent must move down.

Step 13 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
▶      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

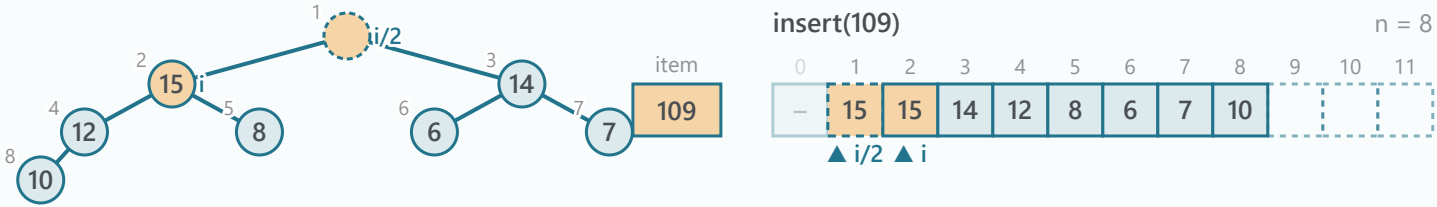
VARIABLES

`item.x = 109` `n = 8` `i = 2` `i/2 = 1` `heap[i/2].x = 15`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`



heap[2] = heap[1]: the parent (15) shifted down one level. No swap; the hole (orange) moved up.

Step 14 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
▶      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

VARIABLES

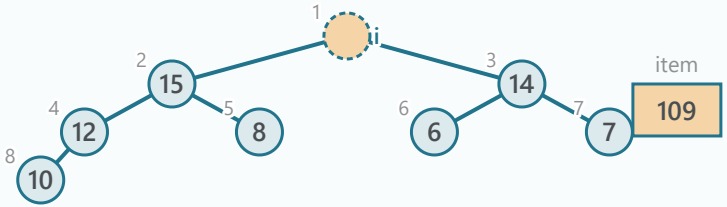
`item.x = 109` `n = 8` `i = 2` `i/2 = 1` `heap[i/2].x = 15`

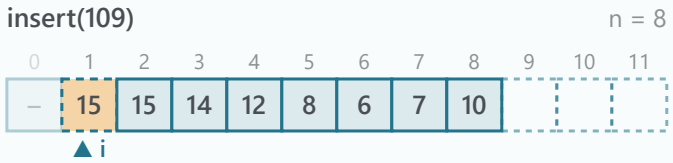
CALL STACK

`insert_max_heap(109)`

`main()`

Code Trace: `insert_max_heap(109)`





i /= 2 → i = 1: move to the parent above.

Step 15 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
▶      i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

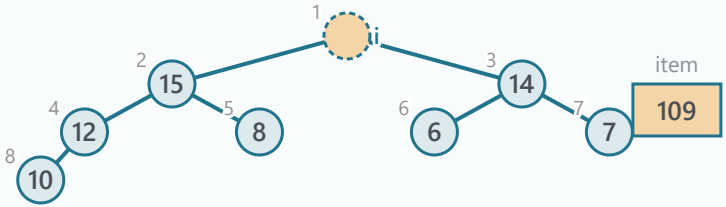
VARIABLES

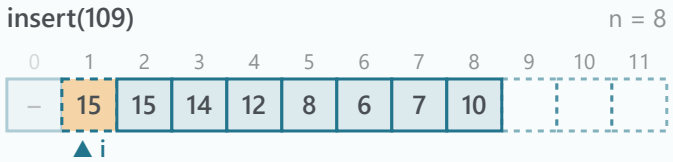
`item.x = 109` `n = 8` `i = 1`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





i == 1: reached the root; the loop ends.

Step 16 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
▶   while (i != 1) {             /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];   /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
12   heap[i] = item;
13 }
```

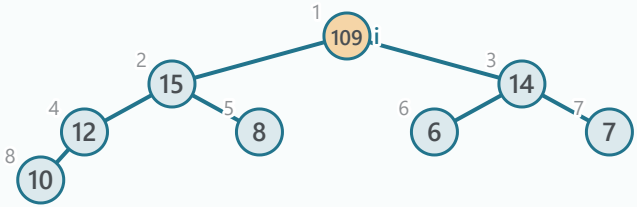
VARIABLES

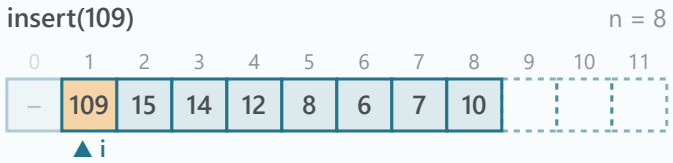
`item.x = 109` `n = 8` `i = 1`

CALL STACK

- `insert_max_heap(109)`
- `main()`

Code Trace: `insert_max_heap(109)`





heap[1] = item: 109 is in place. Heap: `109, 15, 14, 12, 8, 6, 7, 10`.

Step 17 / 17

```
1  void insert_max_heap(element item) {
2    if (HEAP_FULL(n)) {
3      heap_full();              /* error message */
4      return;
5    }
6    int i = ++n;
7    while (i != 1) {            /* i == 1: at root */
8      if (item.x <= heap[i / 2].x) break;
9      heap[i] = heap[i / 2];  /* move parent down */
10     i /= 2;                   /* go up a level */
11   }
▶    heap[i] = item;
13 }
```

VARIABLES

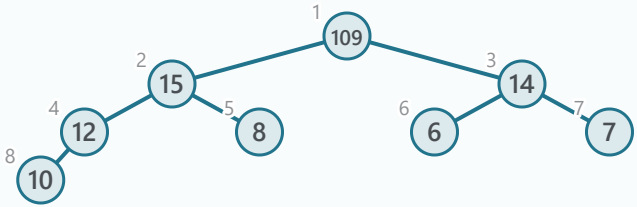
`item.x = 109` `n = 8` `i = 1`

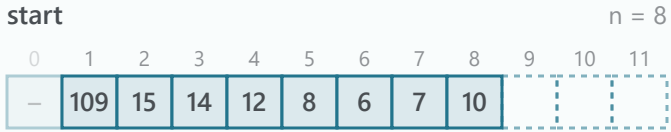
CALL STACK

`insert_max_heap(109)`

`main()`

Successive Insertions: +107, +3, +15





Now: `109, 15, 14, 12, 8, 6, 7, 10`. Next come 107, 3 and 15.

Step 1 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



First a new slot opens: the first free position of the complete tree (**9**).

Step 2 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



107 > 12: parent **12** moves down; the hole rises to position 4.

Step 3 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



107 > 15: parent **15** moves down; the hole rises to position 2.

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

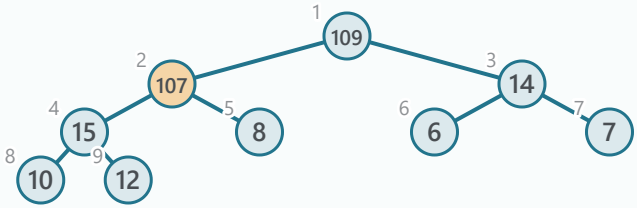
Successive Insertions: +107, +3, +15

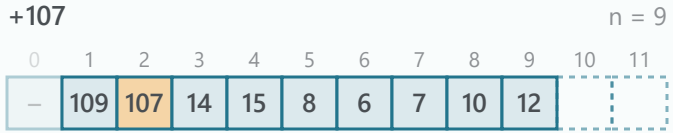


107 ≤ 109: the parent is large enough; stop.

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15

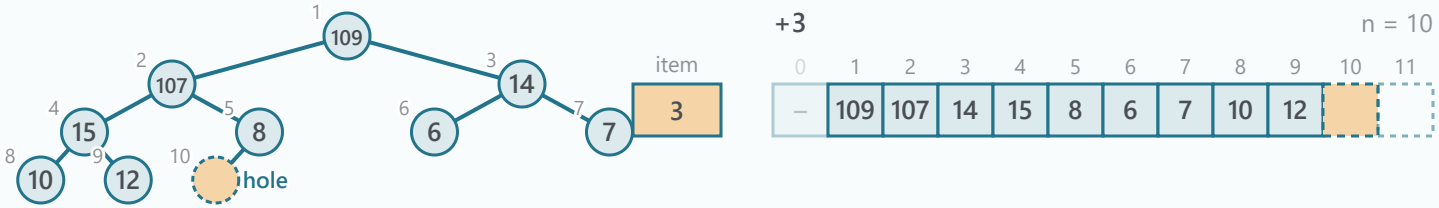




107 is written at position 2. The heap property holds again.

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15

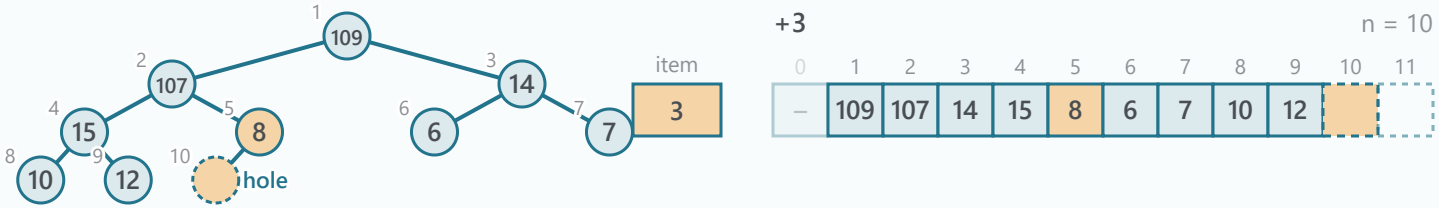


First a new slot opens: the first free position of the complete tree (**10**).

Step 7 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15

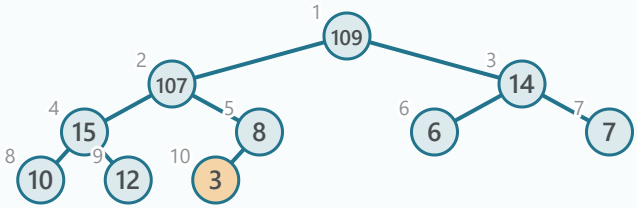


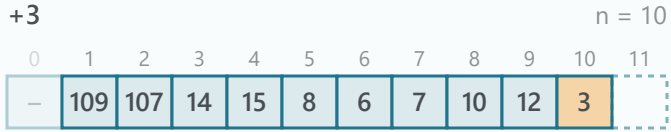
3 ≤ 8: the parent is large enough; stop.

Step 8 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15





3 is written at position 10. The heap property holds again.

Step 9 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



First a new slot opens: the first free position of the complete tree (**11**).

Step 10 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



15 > 8: parent **8** moves down; the hole rises to position 5.

Step 11 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15

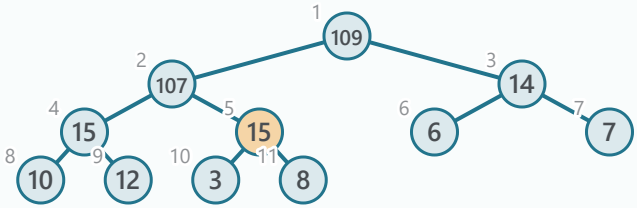


15 ≤ 107: the parent is large enough; stop.

Step 12 / 13

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Successive Insertions: +107, +3, +15



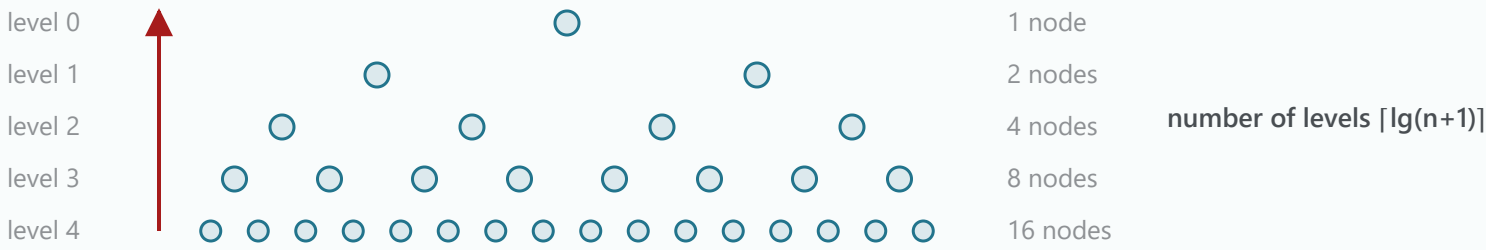
+15 n = 11

0	1	2	3	4	5	6	7	8	9	10	11
–	109	107	14	15	15	6	7	10	12	3	8

15 is written at position 5. The heap property holds again.

Inserting 3 needs no shifts at all (best case, Ω(1)); 109 climbed all the way to the root (worst case, O(lg n)).

Insertion Cost: At Most the Height of the Tree

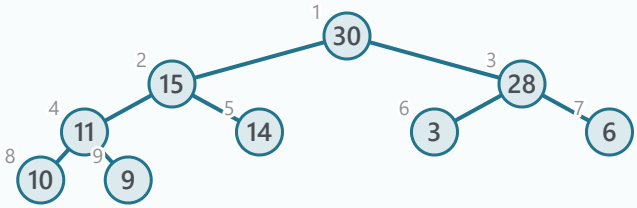


Case	Example	Shifts	Cost
Best	inserting 3	0	Ω(1)
Worst	inserting 109	the height	O(lg n)
General	—	0 … ⌊lg n⌋	O(lg n)

A complete binary tree with n items has ⌈lg(n+1)⌉ levels. The loop climbs one level per iteration, so the cost is at most lg n and at least 1: time complexity **O(lg n)**.

DELETION

Deleting from a Max Heap: Take the Root, Place the Last Item



start n = 9

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	9

Data: `30, 15, 28, 11, 14, 3, 6, 10, 9`. The first item out is 30 at the root.

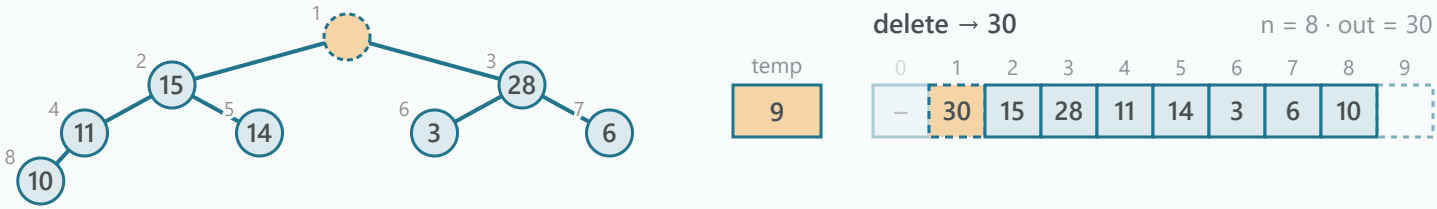
Step 1 / 4

At each step three values matter: the two children and the last item. The largest goes to the slot i points to; `i` marks the slot to fill, `j` the children.

Instructor *"Just taking an item is easy! But the job doesn't end there. We must delete it and keep the max priority queue structure intact."*

DELETION

Deleting from a Max Heap: Take the Root, Place the Last Item



The root (30) is taken. The last item **9** is removed and becomes the candidate for the hole at the root.

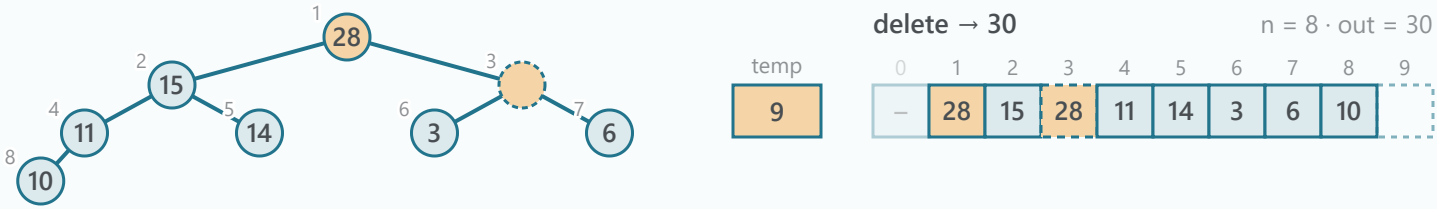
Step 2 / 4

At each step three values matter: the two children and the last item. The largest goes to the slot i points to; `i` marks the slot to fill, `j` the children.

Instructor *"Just taking an item is easy! But the job doesn't end there. We must delete it and keep the max priority queue structure intact."*

DELETION

Deleting from a Max Heap: Take the Root, Place the Last Item



The larger of the two children (28) is larger than temp (9) → moved up.

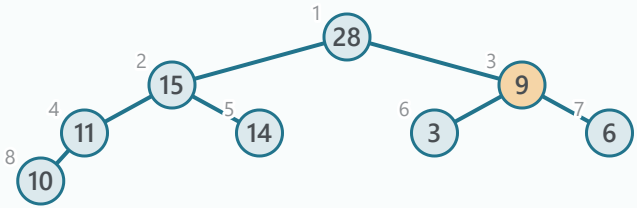
Step 3 / 4

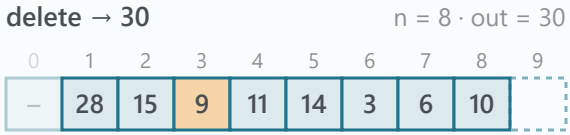
At each step three values matter: the two children and the last item. The largest goes to the slot i points to; `i` marks the slot to fill, `j` the children.

Instructor *"Just taking an item is easy! But the job doesn't end there. We must delete it and keep the max priority queue structure intact."*

DELETION

Deleting from a Max Heap: Take the Root, Place the Last Item





9 settles at position 3.

Step 4 / 4

At each step three values matter: the two children and the last item. The largest goes to the slot i points to; `i` marks the slot to fill, `j` the children.

Instructor *"Just taking an item is easy! But the job doesn't end there. We must delete it and keep the max priority queue structure intact."*

`delete_max_heap`: The Corrected Version

```
bool delete_max_heap(element *out) {
  if (HEAP_EMPTY(n)) return false;
  *out = heap[1];           /* root is returned */
  element temp = heap[n--];  /* find its slot  */
  int i = 1, j = 2;
  while (j <= n) {
    if (j < n && heap[j].x < heap[j + 1].x) j++;
    if (temp.x >= heap[j].x) break;
    heap[i] = heap[j];      /* move child up    */
    i = j; j *= 2;
  }
  heap[i] = temp;
  return true;
}
```

Common mistake

Writing `return 0;` for an empty heap is a **compile error** (an integer does not convert to a struct).

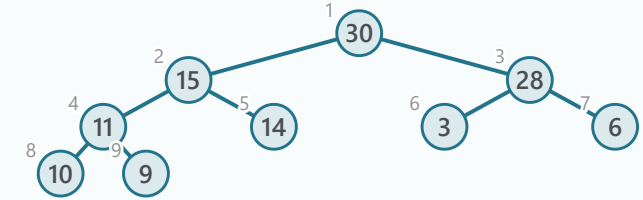
The fix

The suggested solution: return a `bool` (success/failure) and pass the removed item back through a **pointer parameter**.

Why `j < n`?

It guarantees the right child exists before `heap[j+1]` is read.

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 9

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	9

The call: the heap holds 9 items; the root is 30. The removed item goes into `e`.

Step 1 / 15

```
▶ bool delete_max_heap(element *out) {
2   if (HEAP_EMPTY(n)) return false;
3   *out = heap[1];           /* root is returned */
4   element temp = heap[n--];  /* find its slot  */
5   int i = 1, j = 2;
6   while (j <= n) {
7     if (j < n && heap[j].x < heap[j + 1].x) j++;
8     if (temp.x >= heap[j].x) break;
9     heap[i] = heap[j];       /* move child up    */
10    i = j; j *= 2;
11  }
12  heap[i] = temp;
13  return true;
14 }
```

VARIABLES

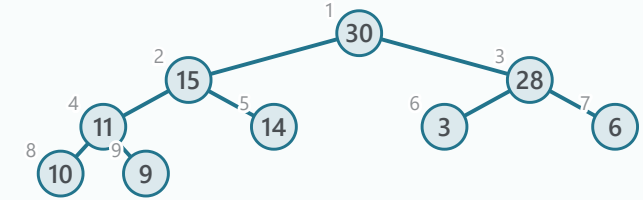
`n = 9`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 9

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	9

`HEAP_EMPTY(9)` is false; there is an item to remove.

Step 2 / 15

```
1  bool delete_max_heap(element *out) {
▶    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

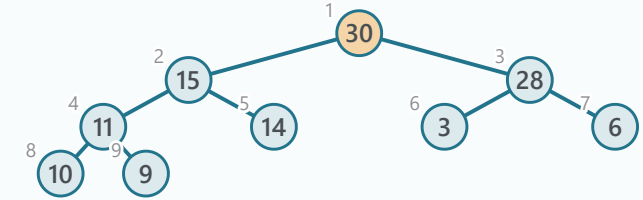
`n = 9`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 9 · out = 30

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	9

***out = heap[1]** = 30: by definition the first item out is the first item of the array.*

Step 3 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
▶    *out = heap[1];          /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

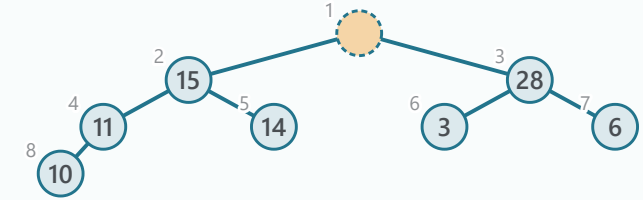
`n = 9` `out→x = 30`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 8 · out = 30

temp: 9

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	

temp = heap[n--] = 9, n = 8. The last item is removed so the tree stays complete; we search for its slot starting at the root.

Step 4 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];          /* root is returned */
▶    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];      /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

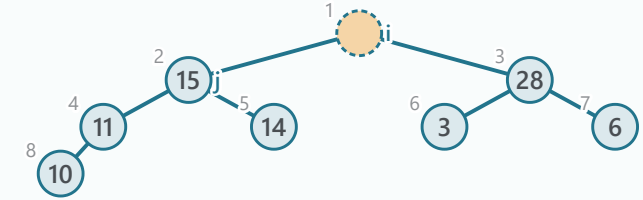
VARIABLES

`n = 8` `out→x = 30` `temp.x = 9`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)



temp: 9

delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	
	▲i	▲j							

i = 1 (candidate slot for temp), **j = 2** (i's left child).

Step 5 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
▶    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

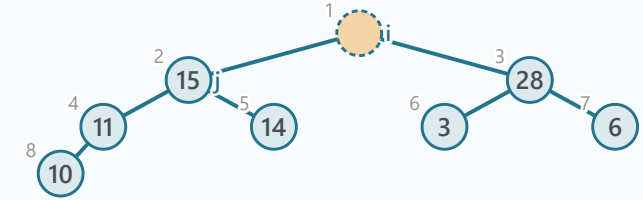
VARIABLES

`n = 8` `out→x = 30` `temp.x = 9` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)



temp: 9

delete_max_heap() — n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	
	▲i	▲j							

j = 2 ≤ n = 8: i has a child; enter the loop.

Step 6 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
▶    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

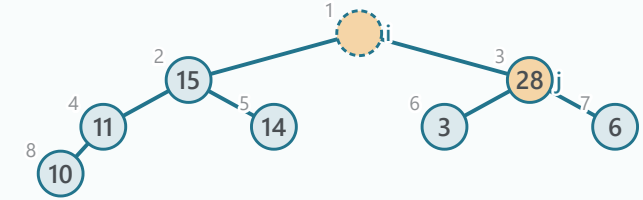
VARIABLES

`n = 8` `out→x = 30` `temp.x = 9` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)



temp: 9

delete_max_heap() — n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	
	▲i		▲j						

The right child is larger (15 < 28) → **j++**, j = 3.

Step 7 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
▶      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

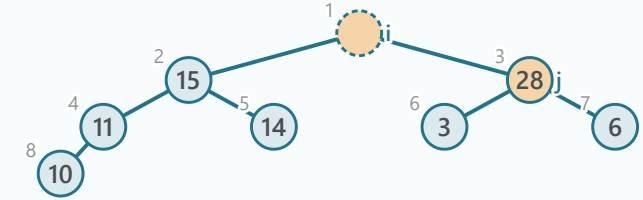
VARIABLES

`n = 8` `out→x = 30` `temp.x = 9` `i = 1` `j = 3`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)



temp

9

delete_max_heap()

n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	30	15	28	11	14	3	6	10	
	▲i		▲j						

9 ≥ 28? No: the larger child must move up.

Step 8 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
▶      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

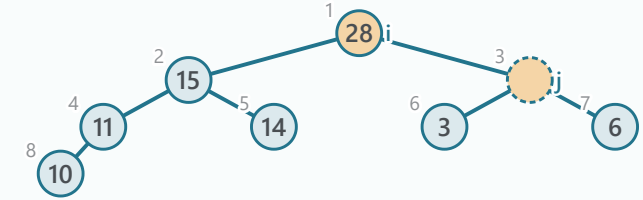
`n = 8` `out→x = 30` `temp.x = 9` `i = 1` `j = 3`

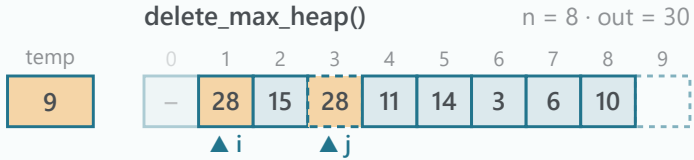
CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: `delete_max_heap` (30)





heap[1] = heap[3]: 28 moved up one level; the hole sank to position 3.

Step 9 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
▶      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

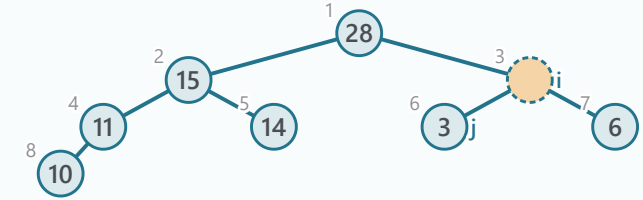
`n = 8` `out→x = 30` `temp.x = 9` `i = 1` `j = 3`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

DELETION · RUN THE CODE STEP BY STEP

Code Trace: `delete_max_heap` (30)



temp

9

delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	28	15	28	11	14	3	6	10	
			▲i			▲j			

i = 3, **j = 6**: continue from the moved value's old slot.

Step 10 / 15

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 8     if (temp.x >= heap[j].x) break;
 9     heap[i] = heap[j];      /* move child up    */
▶      i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

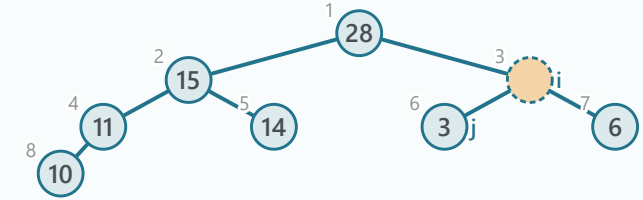
`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 6`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: `delete_max_heap` (30)



temp

9

delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	28	15	28	11	14	3	6	10	
			▲i			▲j			

j = 6 ≤ n = 8: i has a child; enter the loop.

Step 11 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
▶    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

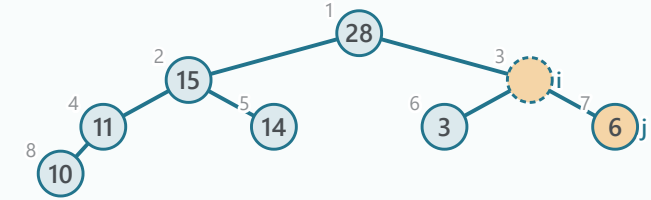
VARIABLES

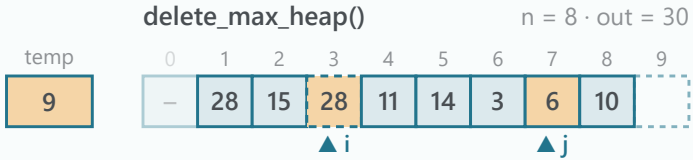
`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 6`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)





The right child is larger (3 < 6) → **j++**, j = 7.

Step 12 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
▶      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

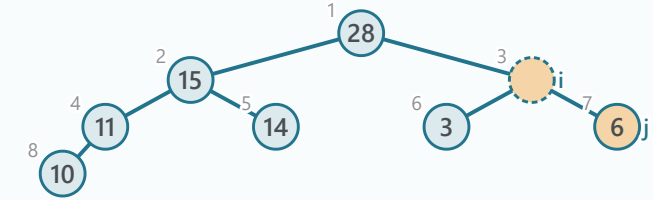
VARIABLES

`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 7`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: `delete_max_heap` (30)



temp: 9

delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	28	15	28	11	14	3	6	10	
			▲i				▲j		

9 ≥ 6? Yes: temp is ≥ both children → `break`.

Step 13 / 15

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 ▶     if (temp.x >= heap[j].x) break;
 9     heap[i] = heap[j];      /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

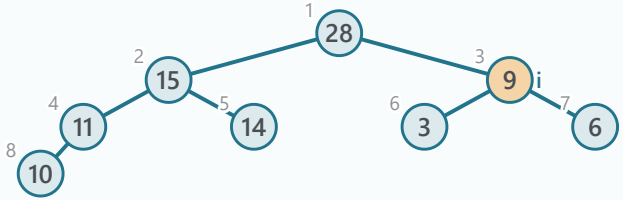
`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 7`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

DELETION · RUN THE CODE STEP BY STEP

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	28	15	9	11	14	3	6	10	

▲ i

heap[3] = temp (9). Heap: `28, 15, 9, 11, 14, 3, 6, 10` .

Step 14 / 15

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
▶    heap[i] = temp;
13   return true;
14 }
```

VARIABLES

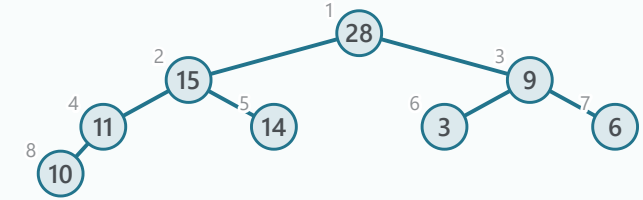
`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 7`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

DELETION · RUN THE CODE STEP BY STEP

Code Trace: `delete_max_heap` (30)



delete_max_heap() n = 8 · out = 30

0	1	2	3	4	5	6	7	8	9
–	28	15	9	11	14	3	6	10	

return true; the caller gets **30** in `out`. The path has ≤ ⌊lg n⌋ levels → O(lg n).

Step 15 / 15

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 8     if (temp.x >= heap[j].x) break;
 9     heap[i] = heap[j];      /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
▶    return true;
14 }
```

VARIABLES

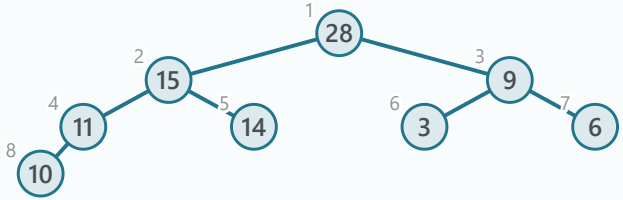
`n = 8` `out→x = 30` `temp.x = 9` `i = 3` `j = 7`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

DELETION · RUN THE CODE STEP BY STEP

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 8

0	1	2	3	4	5	6	7	8
–	28	15	9	11	14	3	6	10

The second call: `28, 15, 9, 11, 14, 3, 6, 10`. This time the last item (10) sinks all the way to a leaf.

Step 1 / 18

```
▶ bool delete_max_heap(element *out) {
2   if (HEAP_EMPTY(n)) return false;
3   *out = heap[1];          /* root is returned */
4   element temp = heap[n--];  /* find its slot  */
5   int i = 1, j = 2;
6   while (j <= n) {
7     if (j < n && heap[j].x < heap[j + 1].x) j++;
8     if (temp.x >= heap[j].x) break;
9     heap[i] = heap[j];       /* move child up    */
10    i = j; j *= 2;
11  }
12  heap[i] = temp;
13  return true;
14 }
```

VARIABLES

n = 8

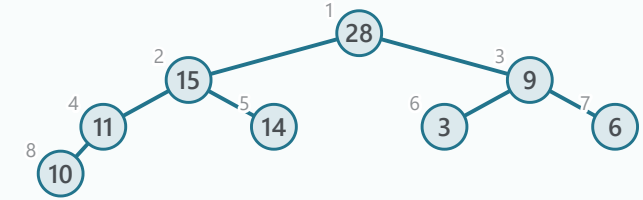
CALL STACK

delete_max_heap(&e)

main()

DELETION · RUN THE CODE STEP BY STEP

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 8

0	1	2	3	4	5	6	7	8
–	28	15	9	11	14	3	6	10

`HEAP_EMPTY(8)` is false; there is an item to remove.

Step 2 / 18

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 8     if (temp.x >= heap[j].x) break;
 9     heap[i] = heap[j];      /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

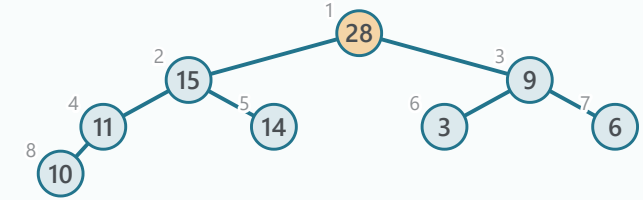
`n = 8`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 8 · out = 28

0	1	2	3	4	5	6	7	8
–	28	15	9	11	14	3	6	10

***out = heap[1]** = 28: by definition the first item out is the first item of the array.

Step 3 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
▶    *out = heap[1];          /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

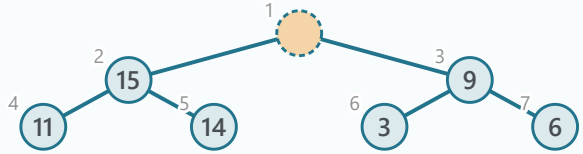
`n = 8` `out→x = 28`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

temp: 10

0	1	2	3	4	5	6	7	8
–	28	15	9	11	14	3	6	

temp = heap[n--] = 10, n = 7. The last item is removed so the tree stays complete; we search for its slot starting at the root.

Step 4 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];          /* root is returned */
▶    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];      /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

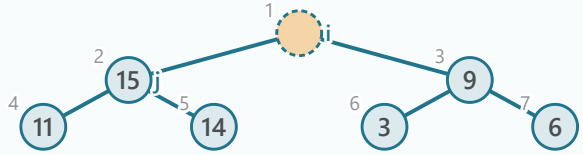
VARIABLES

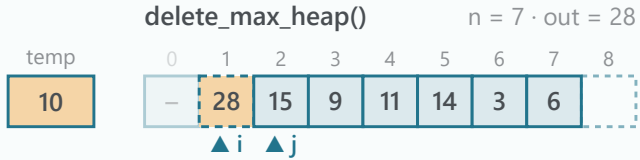
`n = 7` `out→x = 28` `temp.x = 10`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





i = 1 (candidate slot for temp), **j = 2** (i's left child).

Step 5 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
▶    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

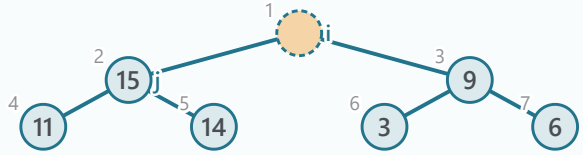
VARIABLES

`n = 7` `out→x = 28` `temp.x = 10` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

temp: 10

0	1	2	3	4	5	6	7	8
–	28	15	9	11	14	3	6	
	▲i	▲j						

j = 2 ≤ n = 7: i has a child; enter the loop.

Step 6 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
▶    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

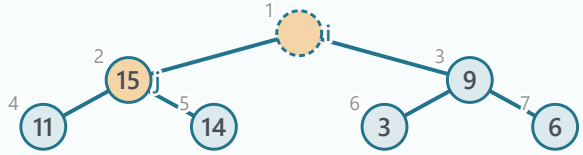
VARIABLES

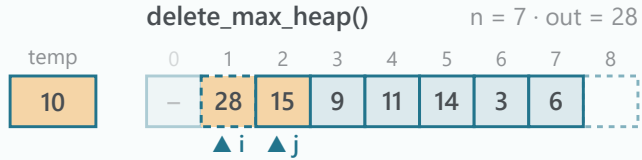
`n = 7` `out→x = 28` `temp.x = 10` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





Left child (15) ≥ right child (9): j stays 2.

Step 7 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
▶      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

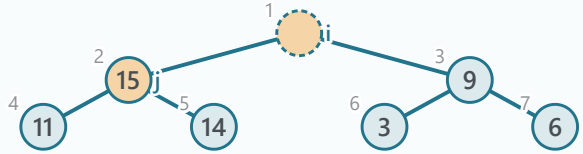
VARIABLES

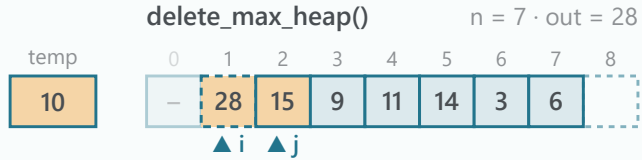
`n = 7` `out→x = 28` `temp.x = 10` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





10 ≥ 15? No: the larger child must move up.

Step 8 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
▶      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

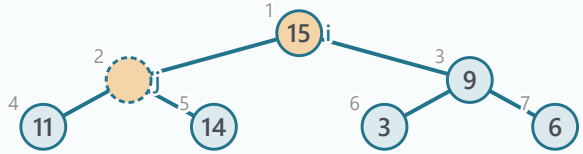
VARIABLES

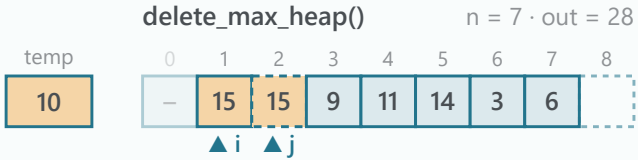
`n = 7` `out→x = 28` `temp.x = 10` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





heap[1] = heap[2]: 15 moved up one level; the hole sank to position 2.

Step 9 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
▶      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

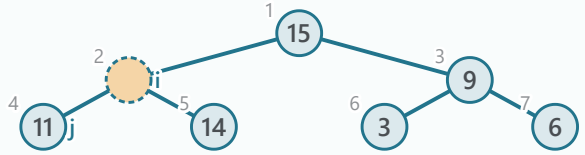
VARIABLES

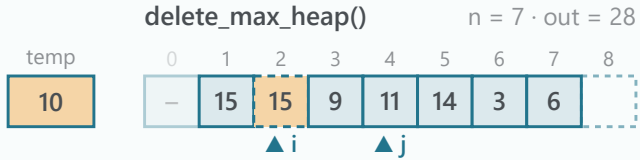
`n = 7` `out→x = 28` `temp.x = 10` `i = 1` `j = 2`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





i = 2, **j = 4**: continue from the moved value's old slot.

Step 10 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
▶      i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

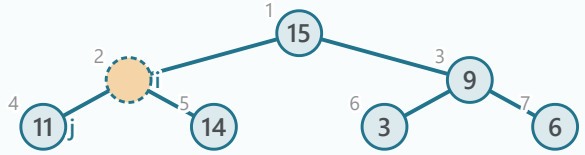
VARIABLES

`n = 7` `out→x = 28` `temp.x = 10` `i = 2` `j = 4`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

temp: 10

0	1	2	3	4	5	6	7	8
–	15	15	9	11	14	3	6	
		▲i		▲j				

j = 4 ≤ n = 7: i has a child; enter the loop.

Step 11 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
▶    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

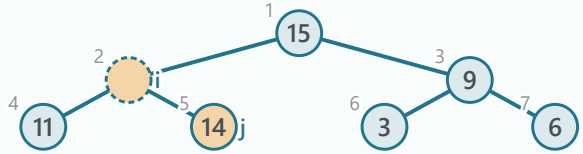
VARIABLES

`n = 7` `out→x = 28` `temp.x = 10` `i = 2` `j = 4`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

temp: 10

0	1	2	3	4	5	6	7	8
–	15	15	9	11	14	3	6	
		▲ i			▲ j			

The right child is larger (11 < 14) → **j++**, j = 5.

Step 12 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
▶      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

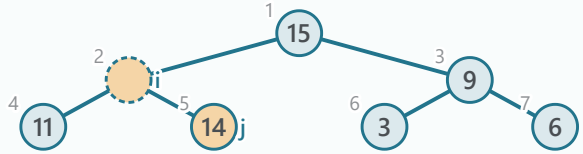
VARIABLES

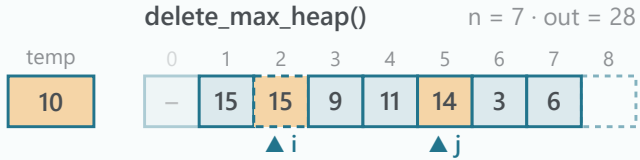
`n = 7` `out→x = 28` `temp.x = 10` `i = 2` `j = 5`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





10 ≥ 14? No: the larger child must move up.

Step 13 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
▶      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

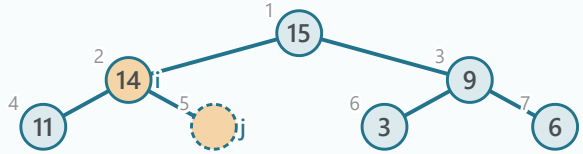
`n = 7` `out→x = 28` `temp.x = 10` `i = 2` `j = 5`

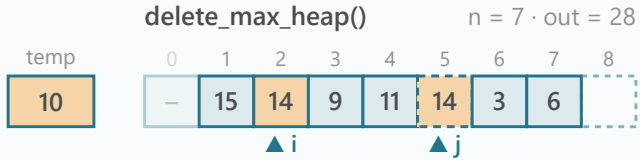
CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: The Second `delete_max_heap` (28)





heap[2] = heap[5]: 14 moved up one level; the hole sank to position 5.

Step 14 / 18

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 8     if (temp.x >= heap[j].x) break;
 ▶     heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

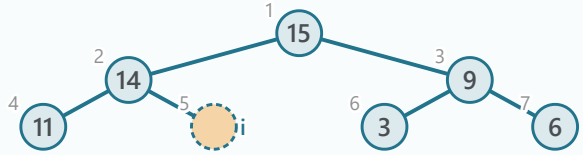
VARIABLES

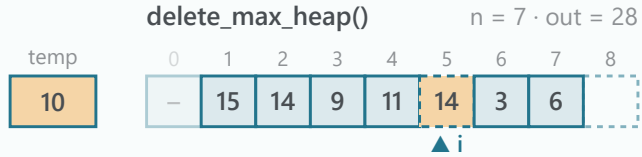
`n = 7` `out→x = 28` `temp.x = 10` `i = 2` `j = 5`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





i = 5, **j = 10**: continue from the moved value's old slot.

Step 15 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
▶      i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

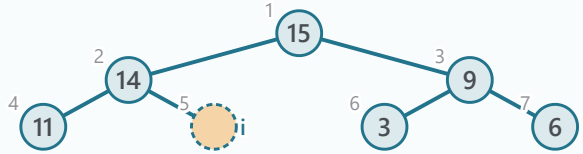
VARIABLES

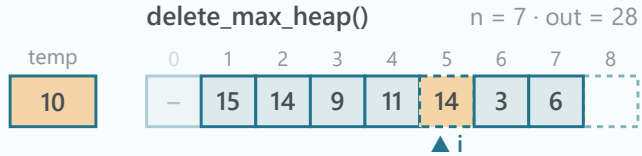
`n = 7` `out→x = 28` `temp.x = 10` `i = 5` `j = 10`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)





j = 10 > n = 7: i is a leaf; leave the loop.

Step 16 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];           /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
▶    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
13   return true;
14 }
```

VARIABLES

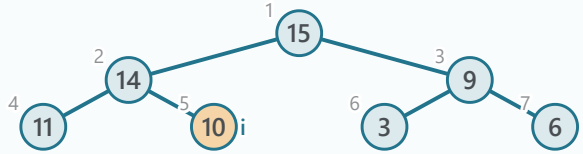
`n = 7` `out→x = 28` `temp.x = 10` `i = 5` `j = 10`

CALL STACK

`delete_max_heap(&e)`

`main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

0	1	2	3	4	5	6	7	8
–	15	14	9	11	10	3	6	

▲ i

heap[5] = temp (10). Heap: `15, 14, 9, 11, 10, 3, 6` .

Step 17 / 18

```
1  bool delete_max_heap(element *out) {
2    if (HEAP_EMPTY(n)) return false;
3    *out = heap[1];          /* root is returned */
4    element temp = heap[n--];  /* find its slot  */
5    int i = 1, j = 2;
6    while (j <= n) {
7      if (j < n && heap[j].x < heap[j + 1].x) j++;
8      if (temp.x >= heap[j].x) break;
9      heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
▶    heap[i] = temp;
13   return true;
14 }
```

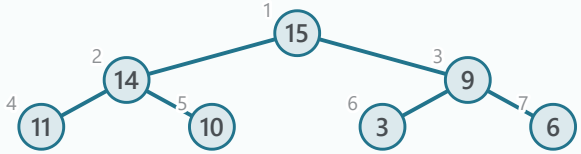
VARIABLES

`n = 7` `out→x = 28` `temp.x = 10` `i = 5` `j = 10`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Code Trace: The Second `delete_max_heap` (28)



delete_max_heap() n = 7 · out = 28

0	1	2	3	4	5	6	7	8
–	15	14	9	11	10	3	6	

return true; the caller gets **28** in `out`. The path has ≤ ⌊lg n⌋ levels → O(lg n).

Step 18 / 18

```
 1 bool delete_max_heap(element *out) {
 2   if (HEAP_EMPTY(n)) return false;
 3   *out = heap[1];           /* root is returned */
 4   element temp = heap[n--];  /* find its slot  */
 5   int i = 1, j = 2;
 6   while (j <= n) {
 7     if (j < n && heap[j].x < heap[j + 1].x) j++;
 8     if (temp.x >= heap[j].x) break;
 9     heap[i] = heap[j];       /* move child up    */
10     i = j; j *= 2;
11   }
12   heap[i] = temp;
▶    return true;
14 }
```

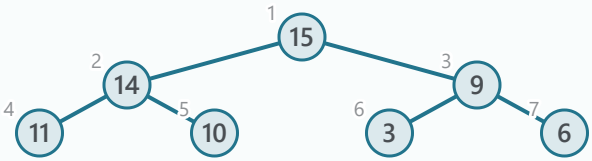
VARIABLES

`n = 7` `out→x = 28` `temp.x = 10` `i = 5` `j = 10`

CALL STACK

- `delete_max_heap(&e)`
- `main()`

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



state n = 7

0	1	2	3	4	5	6	7	8	9
–	15	14	9	11	10	3	6		

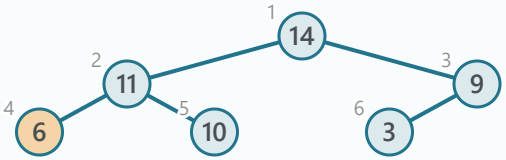
Now: `15, 14, 9, 11, 10, 3, 6`. Removed so far: 30, 28.

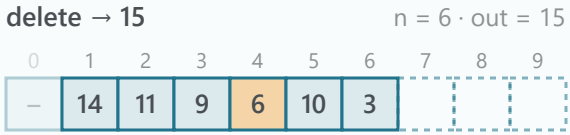
Step 1 / 8

Instructor *“Now it's a matter of pride! ☺”*

DELETION

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



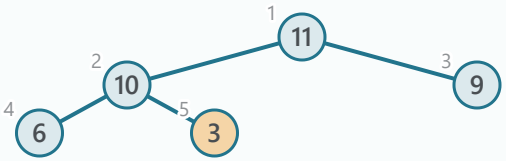


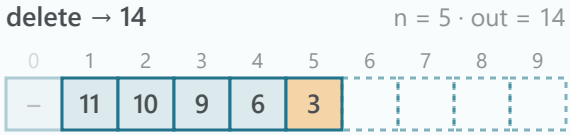
15 removed; the last item 6 settled at position 4. Output order: 30, 28, 15

Step 2 / 8

Instructor *“Now it's a matter of pride! ☺”*

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



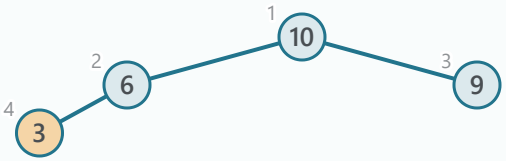


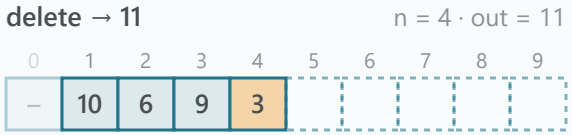
14 removed; the last item 3 settled at position 5. Output order: 30, 28, 15, 14

Instructor *“Now it's a matter of pride! ☺”*

DELETION

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order





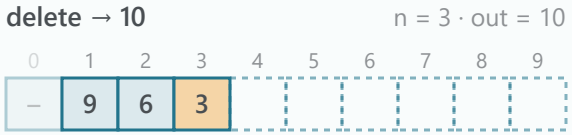
11 removed; the last item 3 settled at position 4. Output order: 30, 28, 15, 14, 11

Step 4 / 8

Instructor *“Now it's a matter of pride! ☺”*

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order

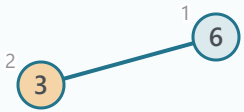


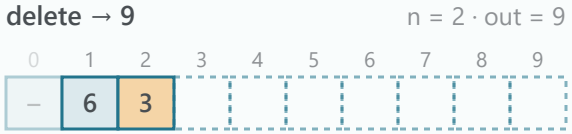


10 removed; the last item 3 settled at position 3. Output order: 30, 28, 15, 14, 11, 10

Instructor *"Now it's a matter of pride! ☺"*

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



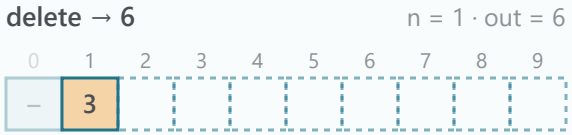


9 removed; the last item 3 settled at position 2. Output order: 30, 28, 15, 14, 11, 10, 9

Instructor *“Now it's a matter of pride! ☺”*

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



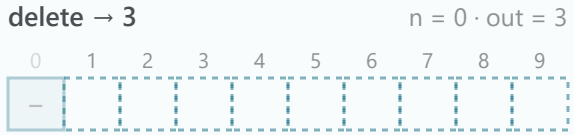


6 removed; the last item 3 settled at position 1. Output order: 30, 28, 15, 14, 11, 10, 9, 6

Instructor *“Now it's a matter of pride! ☺”*

DELETION

Delete Until the Heap Is Empty: Items Come Out in Decreasing Order



3 removed; 3 was the last item, and the heap is empty. Output order: 30, 28, 15, 14, 11, 10, 9, 6, 3

Step 8 / 8

Instructor *"Now it's a matter of pride! ☺"*

Deletion Cost and the Journey of the Last Item

It may stop midway

In the 1st deletion, 9 settled at position 3 without sinking three levels.

It may go to the bottom

In the 2nd deletion, 10 sank to a leaf (position 5): the worst case.

Cost

The loop goes down one level per iteration: at most $\lfloor \lg n \rfloor$ iterations, 2 comparisons each → **O(lg n)**.

A by-product

Deleting n times yields the items in **decreasing order**: this is the idea behind heap sort.

Deletion's best case is $\Omega(1)$ too: if the last item is ≥ both children of the root it settles at once (e.g. when 2 items are left).

Writing the Routines Better: a struct Instead of Globals

```
typedef struct { element *a; int n, capacity; } Heap;

Heap *heap_create(int capacity) {
  Heap *h = malloc(sizeof *h);
  if (h == NULL) return NULL;
  h->a = malloc((capacity + 1) * sizeof *h->a);
  if (h->a == NULL) { free(h); return NULL; }
  h->n = 0; h->capacity = capacity;
  return h;
}
void heap_destroy(Heap *h) { if (h) { free(h->a); free(h); } }
```

Suggestion

Report success with `TRUE` / `FALSE`; pass the item and `n` by reference.

Here

The array and `n` live in a struct: there can be several heaps; `malloc` is always checked.

Instructor *"The code in the book is not very correct anyway. But since the definitions are correct, as far as I can see it is an ideal book for learning data structures..."*

The Min Versions: Rename and Flip the Comparisons

```
void insert_min_heap(element item) {
  /* ... same full check ... */
  int i = ++n;
  while (i != 1) {
    if (item.x >= heap[i / 2].x) break; /* >= ! */
    heap[i] = heap[i / 2];
    i /= 2;
  }
  heap[i] = item;
}
```

Where	Max	Min
insert: stop	`item ≤ parent`	`item ≥ parent`
delete: child	larger	smaller
delete: stop	`temp ≥ [j]`	`temp ≤ [j]`

Instructor *"Change the routine's name and flip the less-than and greater-than signs, and you have successfully written a min heap."*

When Priority Changes over Time: The Planes' Fuel

Observation

Each plane burns fuel at a different rate: fuel levels must be updated every second by an interrupt.

In the heap

An item whose priority rises moves **up**, one whose priority falls moves **down**: O(lg n). Position k is kept in an index table.

```
/* priority of the item at k rose: sift it up */
void increase_key(int k, int key) {
  element e = heap[k];
  e.x = key;
  while (k != 1 && e.x > heap[k / 2].x) {
    heap[k] = heap[k / 2];
    k /= 2;
  }
  heap[k] = e;
}
```

Instructor *"One plane may burn 10 gallons (37.85 liters) of fuel a second, while another can perhaps fly an hour on 10 gallons."*

COMPARISON

Why a Heap? Wouldn't a Simple Array or List Do?

#	Structure	Insert	Delete (max)	Why
1	Unsorted array	Θ(1)	Θ(n)	Append at the end; finding the max scans everything.
2	Unsorted list	Θ(1)	Θ(n)	Insert at the front; still a full scan.
3	Sorted array (ascending)	O(n)	Θ(1)	binsearch O(lg n) + n shifts; the max is at the end.
4	Sorted linked list (descending)	O(n)	Θ(1)	Find the slot; delete from the front, update the head.
5	Max heap	O(lg n)	O(lg n)	Just one root–leaf path.

When best (Ω) and worst (O) cases have the same order, the bound is tight and written with Θ. Without binsearch, insertion into a sorted array is Θ(n), not just O(n).

Instructor *"'What if we keep the largest value in a max variable?' What happens when that plane lands? Then you would need a sorted array."*

COMPARISON

Let's Put Numbers into the Cost Table

#	Insert	Delete
1 · unsorted array	1	2,000,000
2 · unsorted list	1	2,000,000
3 · sorted array	2,000,000	1
4 · sorted list	2,000,000	1
5 · max heap	≈ 21	≈ 21

Assume one instruction = 1 second; lg 2,000,000 ≈ 20.9 (rounded to 21).

Instructor *"I analyze 5 different approaches without writing any code, without touching a key."*

Instructor *"I love the number 2 million because its logarithm is easy to take. […] Just for fun, I run the program built on a max heap 40,000 times. It takes as long as running the other structures once."*

COMPARISON

“Who Will Notice My Algorithm Is Bad?”

Instructor *“You may say ‘Instructions take microseconds, not seconds.’ […] but once it is part of an operating system, what a difference! ‘Come on! A plane arrives at the airport every half hour, every hour. Who will notice my algorithm is bad?’ […] In work that demands expertise, code written that way has no academic or scientific value.”*

Instructor *“The problem should be like this: insert an item, give me the smallest right away. […] If anyone still says ‘Sir, I like the sorted linked list,’ there is nothing I can do.”*

Choose the data structure by the mix of operations. For a problem with frequent inserts and frequent “give me the smallest”, a heap doing both in $O(\lg n)$ beats structures that do one in $\Theta(1)$ and the other in $\Theta(n)$.

COMPARISON

Going Further: Binomial and Fibonacci Heaps

Operation	Binary heap	Binomial heap	Fibonacci heap*
insert	O(lg n)	O(lg n)	O(1)
find min	O(1)	O(lg n)	O(1)
delete min	O(lg n)	O(lg n)	O(lg n)
decrease key	O(lg n)	O(lg n)	O(1)
merge two heaps	O(n)	O(lg n)	O(1)

* Fibonacci heap times are **amortized** (averaged over a sequence of operations) (CLRS). They give a theoretical gain in algorithms such as Dijkstra and Prim that do many "decrease key" operations.

Instructor *"This structure is the binary heap. Beyond it there are Fibonacci heap and binomial heap algorithms. Even faster. Those people did not settle for lg n. — Citius! Altius! Fortius! (Faster! Higher! Stronger!)"*

Heapify: Turning an Array into a Heap in Place

```
void build_heap(element a[], int n) {
  for (int k = n / 2; k >= 1; k--) {
    element temp = a[k];
    int i = k, j = 2 * k;
    while (j <= n) {
      if (j < n && a[j].x < a[j + 1].x) j++;
      if (temp.x >= a[j].x) break;
      a[i] = a[j];
      i = j; j *= 2;
    }
    a[i] = temp;
  }
}
```

The idea

Leaves (k > n/2) are already one-item heaps. From `k = n/2` up to the root, each node **sifts down**: a root whose two children are heaps gets fixed.

What do we gain?

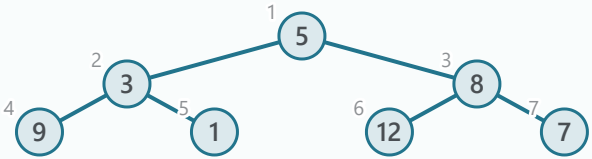
n calls to `insert` take O(n lg n); bottom-up construction takes **O(n)**.

The inner loop

Identical to the loop inside `delete_max_heap` (`adjust` in Horowitz).

HEAPIFY · RUN THE CODE STEP BY STEP

Code Trace: `build_heap` — k = 3 and k = 2



build_heap(a, 7)

0	1	2	3	4	5	6	7
–	5	3	8	9	1	12	7

Input: `5, 3, 8, 9, 1, 12, 7` (not a heap yet). Start at n/2 = 3.

Step 1 / 15

```
▶ void build_heap(element a[], int n) {
2   for (int k = n / 2; k >= 1; k--) {
3     element temp = a[k];
4     int i = k, j = 2 * k;
5     while (j <= n) {
6       if (j < n && a[j].x < a[j + 1].x) j++;
7       if (temp.x >= a[j].x) break;
8       a[i] = a[j];
9       i = j; j *= 2;
10    }
11    a[i] = temp;
12  }
13 }
```

VARIABLES

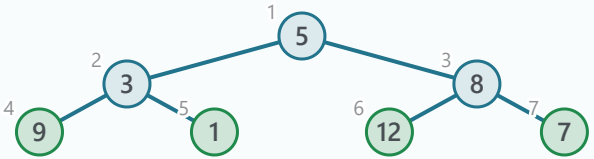
n = 7

CALL STACK

build_heap(a, 7)

main()

Code Trace: `build_heap` — k = 3 and k = 2



build_heap(a, 7) k = 3

0	1	2	3	4	5	6	7
–	5	3	8	9	1	12	7
			▲ k				

k = 3: fix the subtree rooted at 3. Green nodes are already roots of heaps (leaves and those fixed earlier).

Step 2 / 15

```
1  void build_heap(element a[], int n) {
▶    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

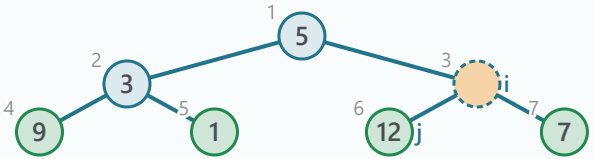
n = 7 k = 3

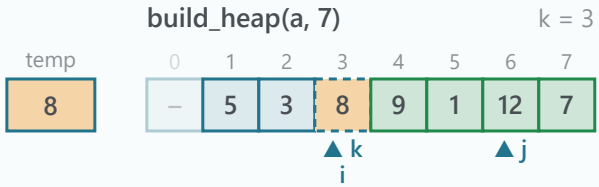
CALL STACK

build_heap(a, 7)

main()

Code Trace: `build_heap` — k = 3 and k = 2





temp = a[3] = 8; i = 3, j = 6. The same sift-down as in deletion begins.

Step 3 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
▶      element temp = a[k];
▶      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

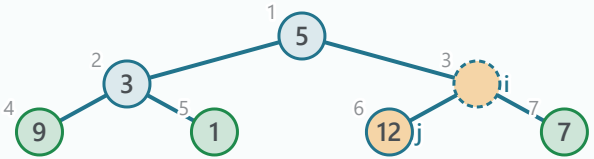
VARIABLES

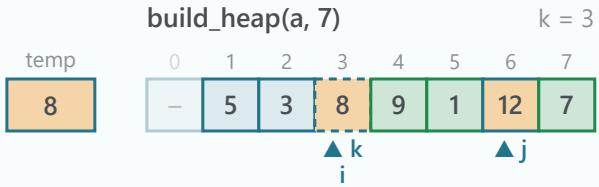
`n = 7` `k = 3` `i = 3` `j = 6` `temp.x = 8`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





Children 12 and 7: the larger is **a[6] = 12**.

Step 4 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
▶        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

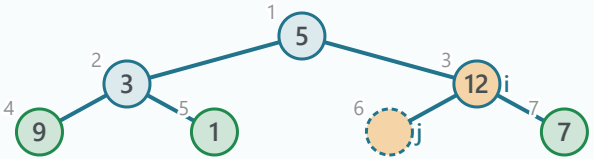
`n = 7` `k = 3` `i = 3` `j = 6` `temp.x = 8`

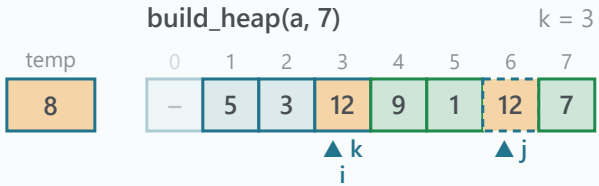
CALL STACK

`build_heap(a, 7)`

`main()`

Code Trace: `build_heap` — k = 3 and k = 2





8 < 12 → **a[3] = a[6]**; the hole sinks to position 6.

Step 5 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
▶        if (temp.x >= a[j].x) break;
▶        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

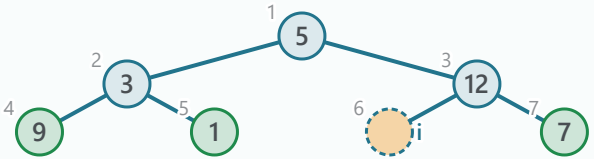
VARIABLES

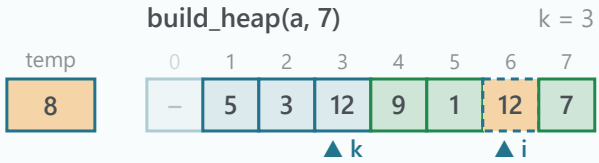
`n = 7` `k = 3` `i = 3` `j = 6` `temp.x = 8`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





i = 6, **j = 12**: continue from the hole's new position.

Step 6 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
▶        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

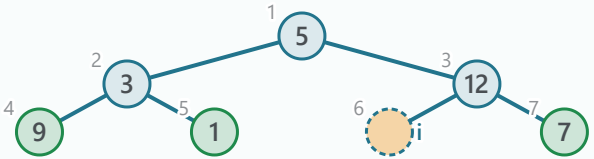
`n = 7` `k = 3` `i = 6` `j = 12` `temp.x = 8`

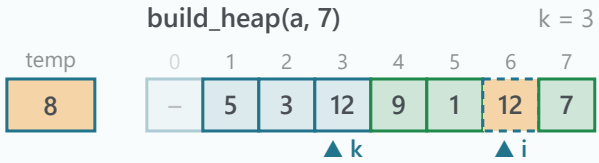
CALL STACK

`build_heap(a, 7)`

`main()`

Code Trace: `build_heap` — k = 3 and k = 2





j = 12 > n: i = 6 is a leaf; the loop ends.

Step 7 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

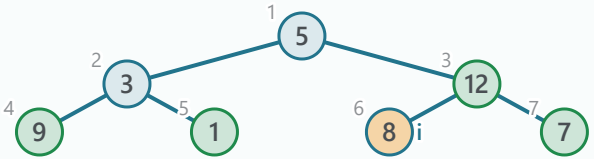
`n = 7` `k = 3` `i = 6` `j = 12` `temp.x = 8`

CALL STACK

`build_heap(a, 7)`

`main()`

Code Trace: `build_heap` — k = 3 and k = 2



build_heap(a, 7) k = 3

0	1	2	3	4	5	6	7
–	5	3	12	9	1	8	7
			▲ k			▲ i	

a[6] = temp (8). The subtree rooted at 3 is now a heap: `5, 3, 12, 9, 1, 8, 7`.

Step 8 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
▶      a[i] = temp;
12   }
13 }
```

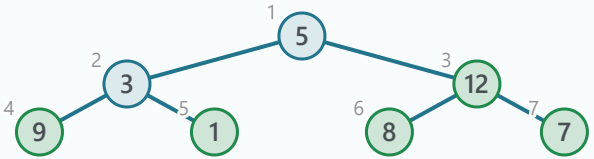
VARIABLES

`n = 7` `k = 3` `i = 6` `j = 12` `temp.x = 8`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2



build_heap(a, 7) k = 2

0	1	2	3	4	5	6	7
–	5	3	12	9	1	8	7
		▲ k					

k = 2: fix the subtree rooted at 2. Green nodes are already roots of heaps (leaves and those fixed earlier).

Step 9 / 15

```
1  void build_heap(element a[], int n) {
▶    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

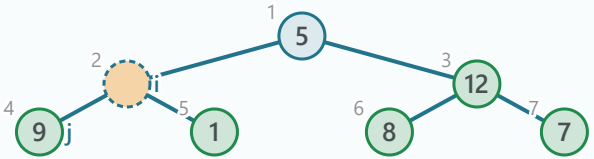
VARIABLES

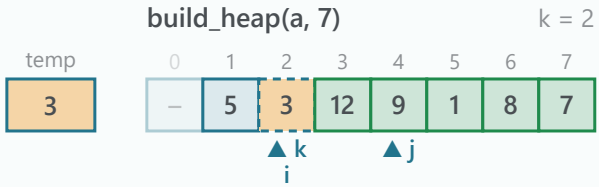
n = 7 k = 2

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





temp = a[2] = 3; i = 2, j = 4. The same sift-down as in deletion begins.

Step 10 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
▶      element temp = a[k];
▶      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

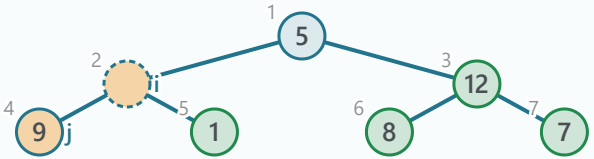
VARIABLES

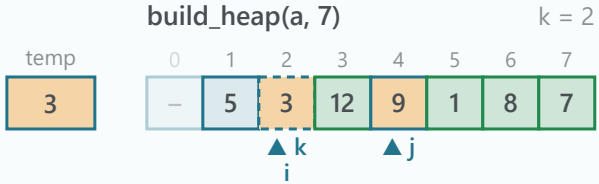
`n = 7` `k = 2` `i = 2` `j = 4` `temp.x = 3`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





Children 9 and 1: the larger is **a[4] = 9**.

Step 11 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
▶        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

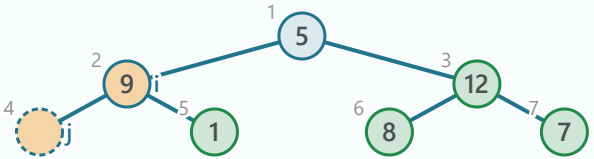
VARIABLES

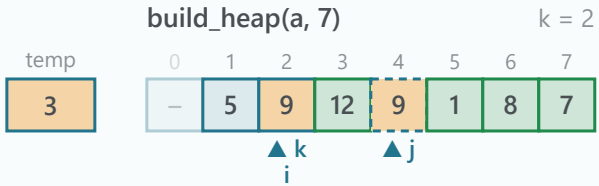
`n = 7` `k = 2` `i = 2` `j = 4` `temp.x = 3`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





3 < 9 → **a[2] = a[4]**; the hole sinks to position 4.

Step 12 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
▶        if (temp.x >= a[j].x) break;
▶        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

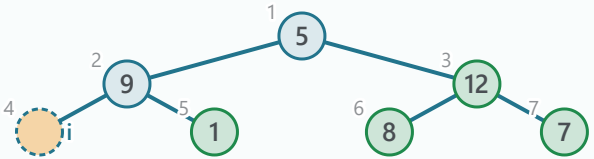
VARIABLES

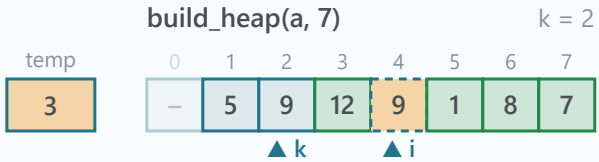
`n = 7` `k = 2` `i = 2` `j = 4` `temp.x = 3`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





i = 4, **j = 8**: continue from the hole's new position.

Step 13 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
▶        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

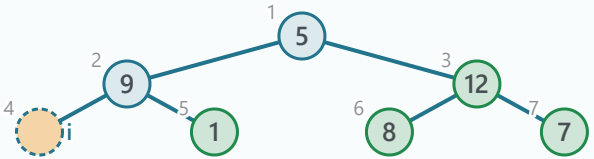
VARIABLES

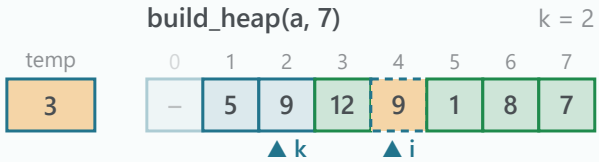
`n = 7` `k = 2` `i = 4` `j = 8` `temp.x = 3`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 3 and k = 2





j = 8 > n: i = 4 is a leaf; the loop ends.

Step 14 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

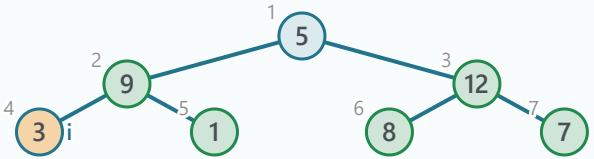
`n = 7` `k = 2` `i = 4` `j = 8` `temp.x = 3`

CALL STACK

`build_heap(a, 7)`

`main()`

Code Trace: `build_heap` — k = 3 and k = 2



build_heap(a, 7) k = 2

0	1	2	3	4	5	6	7
–	5	9	12	3	1	8	7
		▲ k		▲ i			

a[4] = temp (3). The subtree rooted at 2 is now a heap: `5, 9, 12, 3, 1, 8, 7`.

Step 15 / 15

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
▶      a[i] = temp;
12   }
13 }
```

VARIABLES

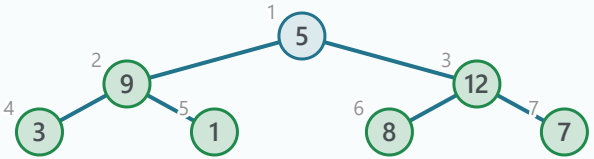
`n = 7` `k = 2` `i = 4` `j = 8` `temp.x = 3`

CALL STACK

- `build_heap(a, 7)`
- `main()`

HEAPIFY · RUN THE CODE STEP BY STEP

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

0	1	2	3	4	5	6	7
–	5	9	12	3	1	8	7

▲ k

k = 1: fix the subtree rooted at 1. Green nodes are already roots of heaps (leaves and those fixed earlier).

Step 1 / 11

```
1  void build_heap(element a[], int n) {
▶    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

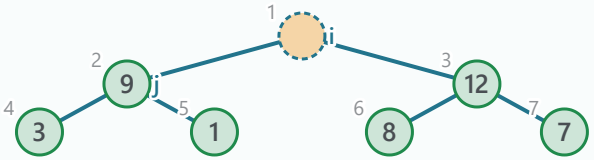
n = 7 k = 1

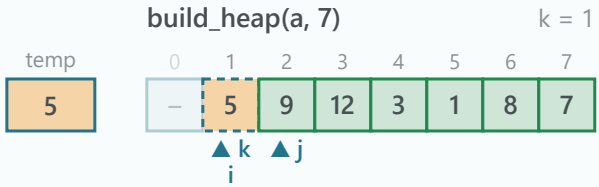
CALL STACK

build_heap(a, 7)

main()

Code Trace: `build_heap` — k = 1





temp = a[1] = 5; i = 1, j = 2. The same sift-down as in deletion begins.

Step 2 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
▶      element temp = a[k];
▶      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

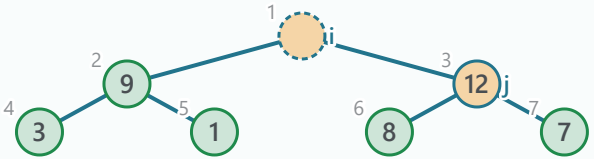
VARIABLES

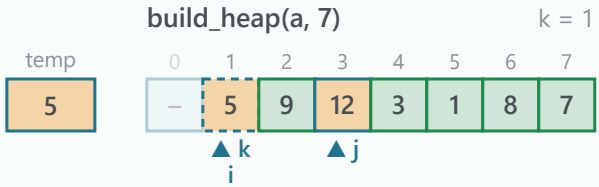
`n = 7` `k = 1` `i = 1` `j = 2` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1





Children 9 and 12: the larger is **a[3] = 12**.

Step 3 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
▶        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

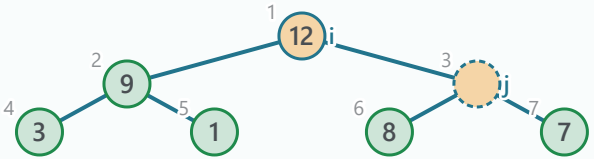
`n = 7` `k = 1` `i = 1` `j = 3` `temp.x = 5`

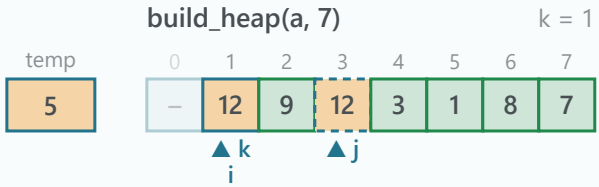
CALL STACK

`build_heap(a, 7)`

`main()`

Code Trace: `build_heap` — k = 1





5 < 12 → **a[1] = a[3]**; the hole sinks to position 3.

Step 4 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
▶        if (temp.x >= a[j].x) break;
▶        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

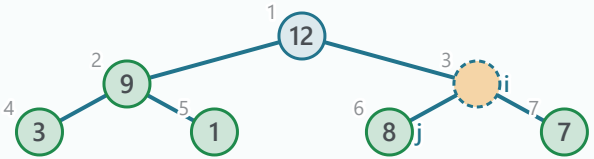
VARIABLES

`n = 7` `k = 1` `i = 1` `j = 3` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

temp
5

0	1	2	3	4	5	6	7
–	12	9	12	3	1	8	7
	▲k		▲i			▲j	

i = 3, **j = 6**: continue from the hole's new position.

Step 5 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
▶        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

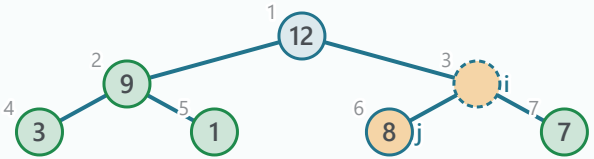
VARIABLES

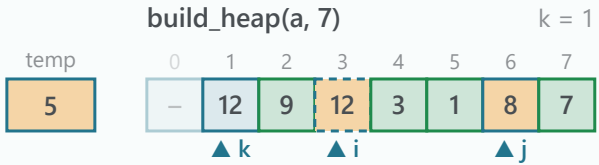
`n = 7` `k = 1` `i = 3` `j = 6` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1





Children 8 and 7: the larger is **a[6] = 8**.

Step 6 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
▶        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

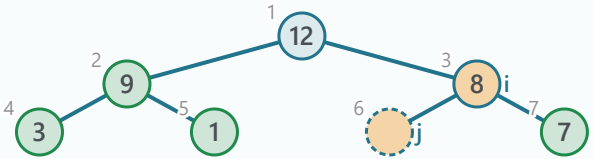
VARIABLES

`n = 7` `k = 1` `i = 3` `j = 6` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

temp
5

0	1	2	3	4	5	6	7
–	12	9	8	3	1	8	7
	▲k		▲i			▲j	

5 < 8 → **a[3] = a[6]**; the hole sinks to position 6.

Step 7 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
▶        if (temp.x >= a[j].x) break;
▶        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

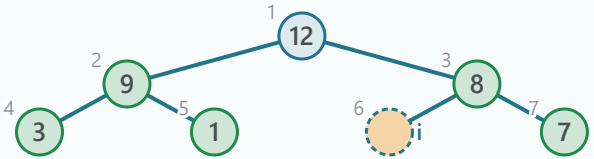
VARIABLES

`n = 7` `k = 1` `i = 3` `j = 6` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

temp
5

0	1	2	3	4	5	6	7
–	12	9	8	3	1	8	7
	▲ k					▲ i	

i = 6, **j = 12**: continue from the hole's new position.

Step 8 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
▶        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

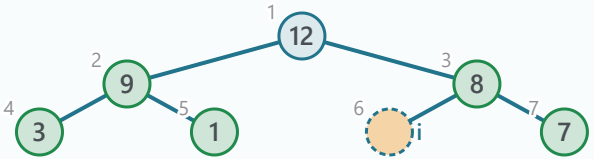
VARIABLES

`n = 7` `k = 1` `i = 6` `j = 12` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

temp: 5

0	1	2	3	4	5	6	7
–	12	9	8	3	1	8	7
	▲ k					▲ i	

j = 12 > n: i = 6 is a leaf; the loop ends.

Step 9 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
▶      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

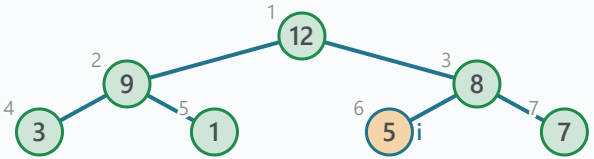
VARIABLES

`n = 7` `k = 1` `i = 6` `j = 12` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7) k = 1

0	1	2	3	4	5	6	7
–	12	9	8	3	1	5	7
	▲ k					▲ i	

a[6] = temp (5). The subtree rooted at 1 is now a heap: `12, 9, 8, 3, 1, 5, 7`.

Step 10 / 11

```
1  void build_heap(element a[], int n) {
2    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
▶      a[i] = temp;
12   }
13 }
```

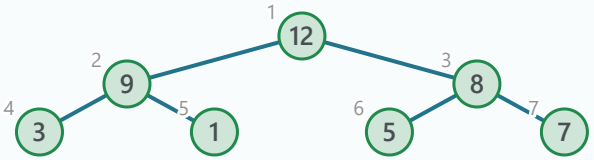
VARIABLES

`n = 7` `k = 1` `i = 6` `j = 12` `temp.x = 5`

CALL STACK

- `build_heap(a, 7)`
- `main()`

Code Trace: `build_heap` — k = 1



build_heap(a, 7)

0	1	2	3	4	5	6	7
–	12	9	8	3	1	5	7

k = 0: the loop is done. The whole array is a max heap: `12, 9, 8, 3, 1, 5, 7`.

Step 11 / 11

```
1  void build_heap(element a[], int n) {
▶    for (int k = n / 2; k >= 1; k--) {
3      element temp = a[k];
4      int i = k, j = 2 * k;
5      while (j <= n) {
6        if (j < n && a[j].x < a[j + 1].x) j++;
7        if (temp.x >= a[j].x) break;
8        a[i] = a[j];
9        i = j; j *= 2;
10     }
11     a[i] = temp;
12   }
13 }
```

VARIABLES

n = **7** k = **–**

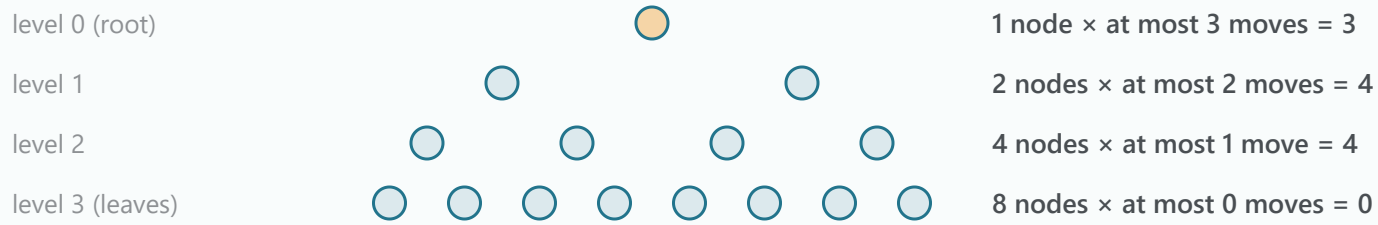
CALL STACK

build_heap(a, 7)

main()

ADVANCED TOPICS

Why Is Heapify O(n)?



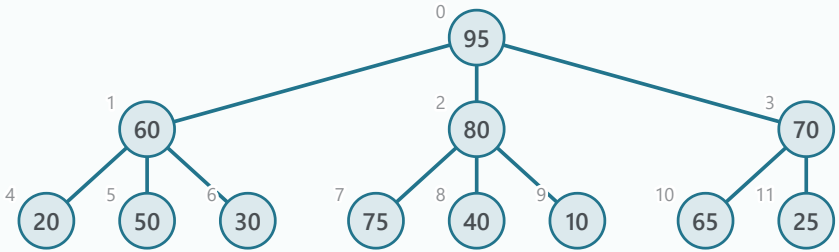
Half the nodes are leaves and never move; a quarter move at most 1 level, an eighth at most 2… Total work:

$$\Sigma_{h \geq 0} \lceil n / 2^{h+1} \rceil \cdot O(h) = O(n \cdot \Sigma\, h/2^{h}) = O(2n) = O(n)$$

For the 15-node complete tree in the example, the total is at most 0·8 + 1·4 + 2·2 + 3·1 = 11 < n. Inserting one by one may instead lift low nodes all the way to the root: n lg n.

ADVANCED TOPICS

d-Ary Heap: Every Node Has d Children



	Binary (d = 2)	d-ary (0-based)
parent	`(i−1)/2`	`(i−1)/d`
children	`2i+1, 2i+2`	`d·i+1 … d·i+d`
height	≈ lg n	≈ $\log_d n$
insert / increase key	O(lg n)	O($\log_d n$) — faster
delete root	O(lg n)	O(d · $\log_d n$) — compares d children

With many inserts/decrease-keys and few deletes (e.g. Dijkstra on dense graphs), choosing d > 2 pays off; a shallow tree is cache-friendly.

ADVANCED TOPICS

Where Are Priority Queues Used?

Operating systems

Scheduling processes and interrupts, timer events.

Graph algorithms

Dijkstra's shortest paths and Prim's MST: picking the nearest vertex.

Heap sort

Build a heap, remove the root n times: O(n lg n), in place.

Event simulation

Events are pulled from a min heap by timestamp.

k largest / smallest

A min heap of size k (a max heap for the smallest): O(n lg k) on streaming data.

Huffman coding

Repeatedly merges the two smallest frequencies.

Lab: Max Heap

insert(15) and two pops

Input (number = insert, pop) `14, 12, 7, 10, 8, 6, 15, pop, pop` Apply

CODE: **insert_max_heap**

```
1 void insert_max_heap(element item) {
2   if (HEAP_FULL(n)) {
3     heap_full();  /* error message */
4     return;
5   }
6   int i = ++n;
7   while (i != 1) {          /* 1: at root */
8     if (item.x <= heap[i / 2].x) break;
9     heap[i] = heap[i / 2];
10    i /= 2;                 /* go up */
11  }
12  heap[i] = item;
13 }
```

empty tree (NULL)



`n 0`

OUTPUT (none yet)

Max heap: no node is smaller than its children; `heap[1]` is the root and the children of `heap[i]` are `2i` and `2i + 1`. The empty slot has a yellow frame.

SUMMARY

Complexity Summary (Binary Heap with n Items)

Operation	Time	Extra space	Note
peek at the max	O(1)	O(1)	`heap[1]`
insert	Ω(1) … O(lg n)	O(1)	sift up
delete the max	O(lg n)	O(1)	sift down
change a priority	O(lg n)	O(1)	if the position is known
bottom-up construction	O(n)	O(1)	in place
building by n inserts	O(n lg n)	O(1)	
search for an arbitrary item	O(n)	O(1)	a heap is not a search structure

SUMMARY

Common Mistakes

1. Mixing up the 1-based (`i/2`, `2i`) and 0-based (`(i-1)/2`, `2i+1`) formulas.
2. Declaring the array one smaller than the capacity: `heap[199]` writes out of bounds.
3. Writing `return 0;` in a routine that returns an `element` (compile error).
4. Forgetting the `j < n` test and reading `heap[j+1]` (outside the heap; outside the array in `build_heap`).
5. Picking the first child instead of the larger one when deleting: the heap property breaks.
6. Treating a heap as a sorted array or a binary search tree.

Exam-Style Question: Insert, Insert, … Delete, Delete

WITH SAMPLE DATA

Into the max heap `heap[1..6] = 14, 12, 7, 10, 8, 6` we insert **15, 109, 107, 3, 15** in order; then `delete_max_heap` is called twice.

(i) What does the array hold after the insertions? (ii) Which items are removed, and what is the final array? (iii) Which insertion shifted least, and which most?

▶ **Solution**

SUMMARY

Exercises

1. Insert 20, 5, 30, 10, 40 into an empty max heap; write the array after each insertion.
2. Write `delete_max_heap` for a min heap. Which lines change?
3. At most how many shifts does an insertion make in a heap of n = 1000 items?
4. In a 1-based heap, at which indices are the parent and the children of node 13?
5. If successive `insert`s are applied to `5, 3, 8, 9, 1, 12, 7` instead of build_heap, is the result the same?
6. Where can the smallest item of a heap be? What does finding it cost?

▶ **Show answers**

SUMMARY

Chapter Summary

Priority queue

The first item out has the highest (max) or the lowest (min) value.

Heap

Complete binary tree + parent ≥ child. In a 1-based array the parent is i/2, the children 2i and 2i+1.

Insertion

Open the first free slot, shift larger ancestors down: O(lg n).

Deletion

Take the root, find a slot for the last item starting at the root, move larger children up: O(lg n).

Why a heap?

Other structures push one operation to Θ(n); a heap keeps both logarithmic.

Beyond

Heapify in O(n), d-ary heaps, binomial and Fibonacci heaps.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Section 5.6 (Heaps), Chapter 7 (Heap Sort).
- T. H. Cormen et al., *Introduction to Algorithms* (CLRS), Chapter 6 (Heapsort, priority queues) and the chapter on Fibonacci heaps.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 9 (Priority Queues and Heapsort).

← Chapter 12: Threaded Binary Trees

Course page

Chapter 14: Hash Tables

The End

Chapter 13: Priority Queues and Heaps

Next chapter: Chapter 14: Hash Tables · Course page