Priority Queue & Binary Heap

Introduction

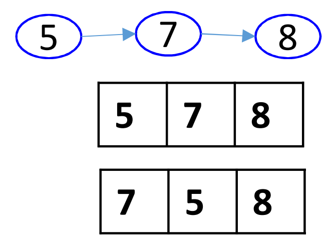
- simple queues doesn’t work in some instances
  - Prim’s algorithm
  - Dijkstra’s algorithm

Priority Queue

is a data structure of items with keys (priorities) that supports two basic operations: insert a new item, and delete the item with the largest (smallest) key.

Model of a Priority Queue

- Several possible implementations are possible:
  - Simple linked list
  - A sorted contiguous list
  - An unsorted list
  - Binary search tree



- What will be the complexity of *insert*, *delmax (or delmin)* and other operations if the above data structures are used?

Priority Queue ADT

- In practice, several other operations needed to maintain the queues under all the conditions.
- A more complete set of operations:
  - Construct a priority queue from *n* given items.
  - Insert a new item
  - Delete the maximum/minimum item
  - Change the priority of an arbitrarily specified item
  - Delete an arbitrarily specified item
  - Join two priority queues into one large one.

Priority Queue Implementations

- Implementations of PQ ADT have widely varying performance:

	insert	delmax	delete	findmax	change	join
ordered array	n	1	n	1	n	n
ordered list	n	1	n	1	n	n
unordered array	1	n	n	n	n	n
unordered list	1	n	n	n	n	1
heap	$\log n$	$\log n$	$\log n$	1	$\log n$	n

Binary Heap (or Just Heap)

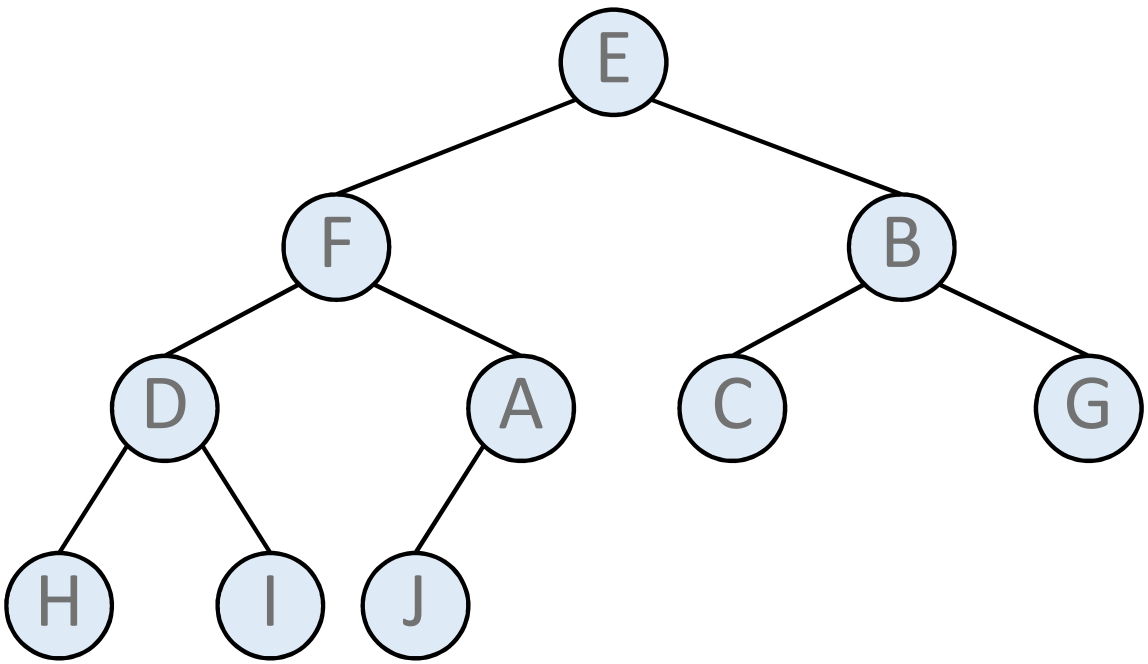
- Heaps have two properties
  - Structure property
  - Heap-order property
- An operation on a heap can destroy one of the properties,
  - A heap operation must not terminate until all heap properties are restored.

Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is always filled from left to right.

Such a tree is known as a complete binary tree.

Structure Property: a heap is a binary tree that is completely filled.



h : height

n : number of nodes

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

h : 3

n : 10

$8 \le 10 < 15$

A Complete binary tree

Height of Heaps

- A complete binary tree of height h has at least 2^h and at most $2^{h+1} - 1$ nodes.
- This implies that the height of a complete binary tree is

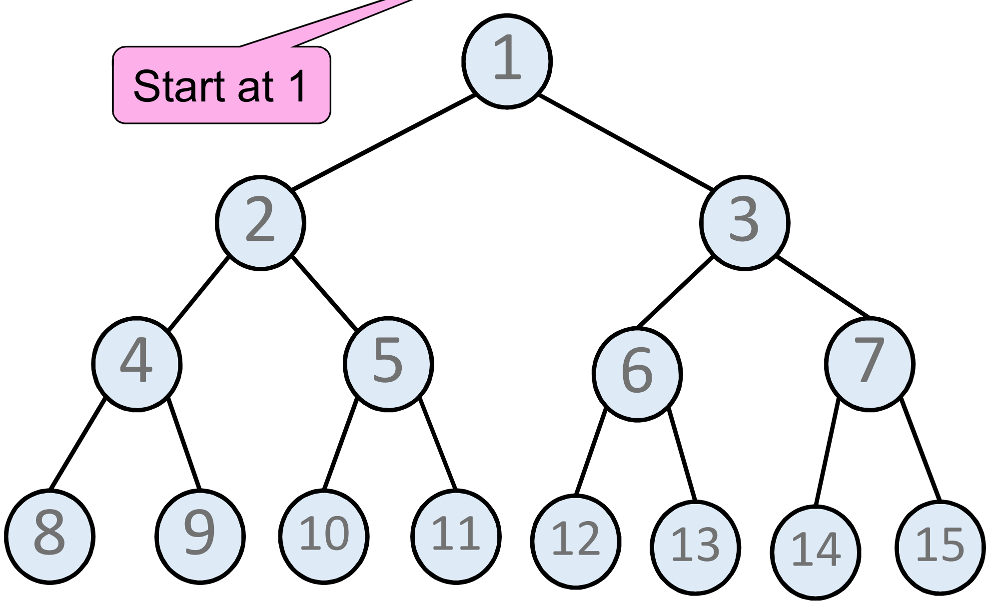
$$\lfloor \log n \rfloor = O(\log n)$$

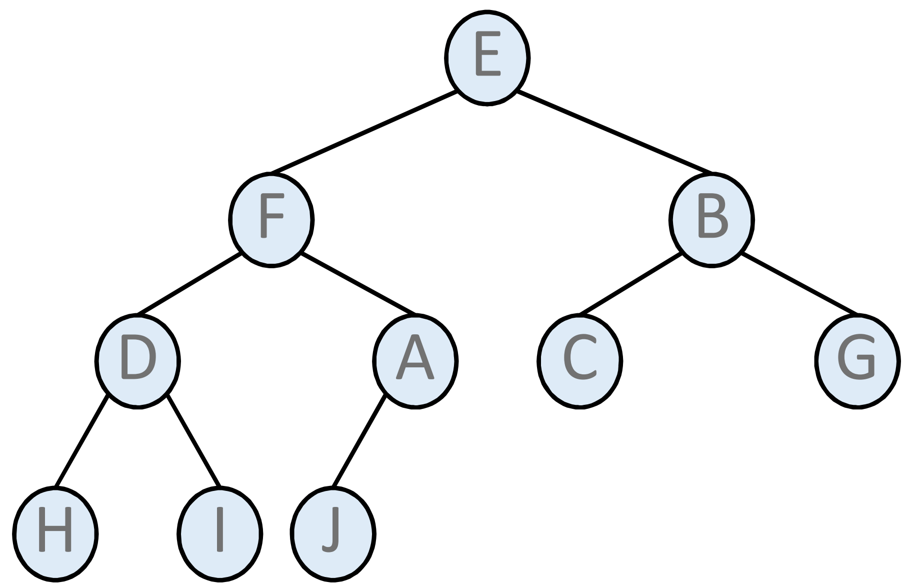
- Because a complete binary tree is so regular, it can be represented in an array.
  - This encourages a straight-forward pointer-free implementation.

Array Implementation of Heap

elements		E	F	B	D	A	C	G	H	I	J			
Node index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Start at 1

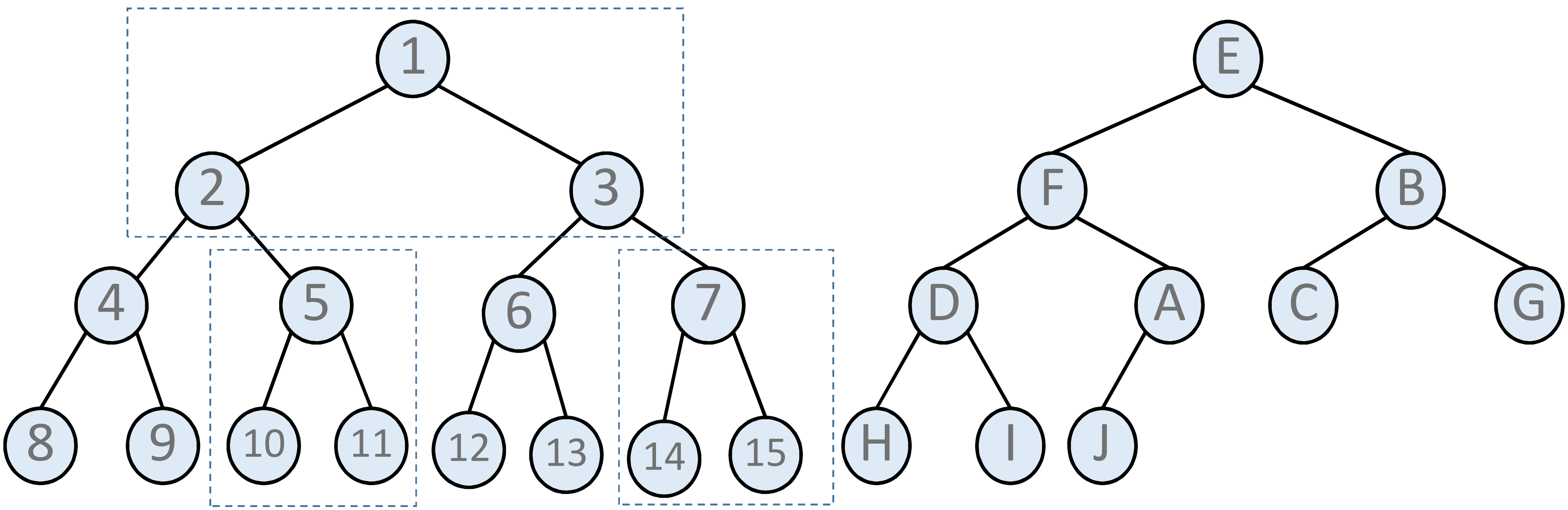




Array Implementation of Heap

- left child: position $2i$
- right child: position $(2i + 1)$,
- parent: position $\lfloor i/2 \rfloor$

	E	F	B	D	A	C	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13



Array Implementation of Heap

	E	F	B	D	A	C	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- For any element in array position *i*,
  - left child: position $2i$
  - right child: position $(2i + 1)$,
  - parent: position $\lfloor i/2 \rfloor$
- No pointers are required, and the operations required to traverse the tree are extremely simple. (*Note: bit shifting can be used :*

  001101 → 000110)
- The only problem is the estimation of the maximum heap size required in advance.

Heap Order Property

- The other trick that enables operations to be performed quickly is the heap order property.
- the largest/smallest element is placed at the root => we can find it in constant time.
- Thus, *findmax*/*findmin*, now in constant time O(1).
- In addition, the heap order property is slightly less strict than the search order in binary search tree.

Heap Order Property

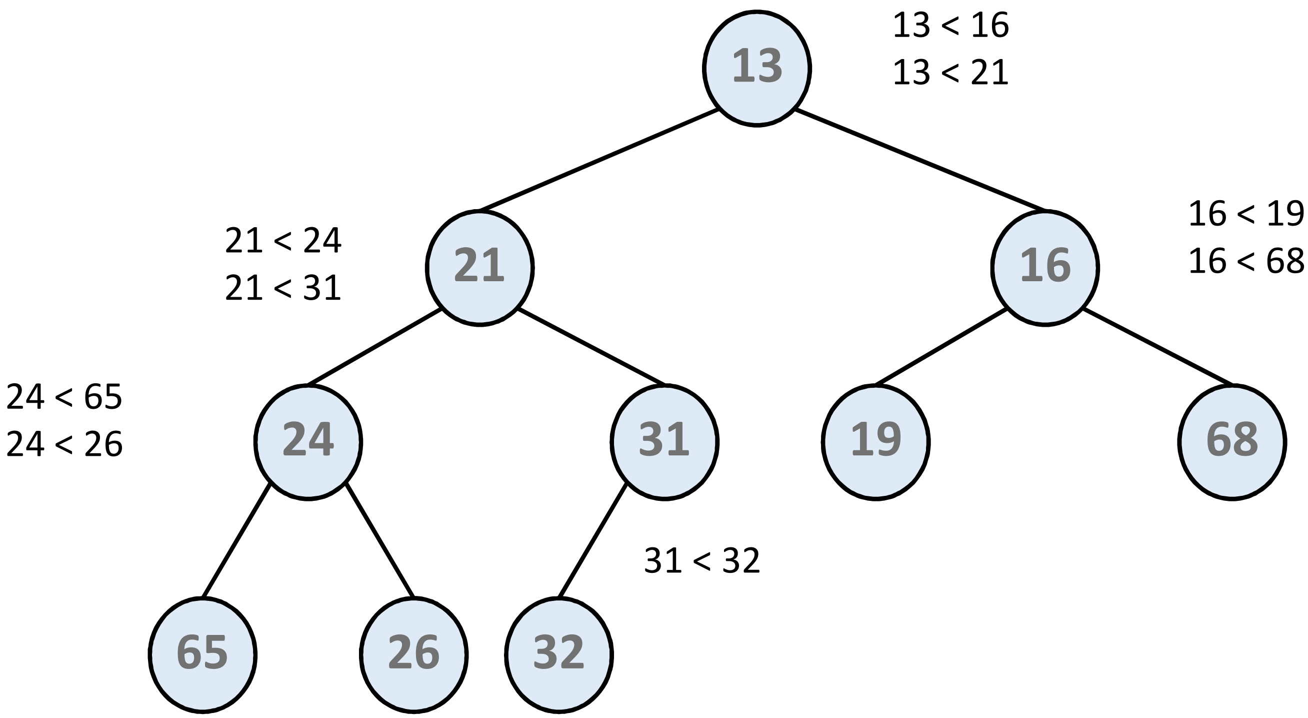
Heap Order Property

Each node is larger(smaller) than or equal to the keys in all of that node's children (if any).

Equivalently, the key in each node of a heap-ordered tree is smaller(larger) than or equal to the key in that node's parent (if any).

- If the parent is larger than its children, the heap is known as a max-heap.
- If the parent is smaller than its children, the heap is known as a min-heap.
- In the following, we consider min-heaps.

Min-Heap: Example



Exercise: Write a max-heap with the same set of keys.

Heap: *Insert* (to insert an element x into the heap)

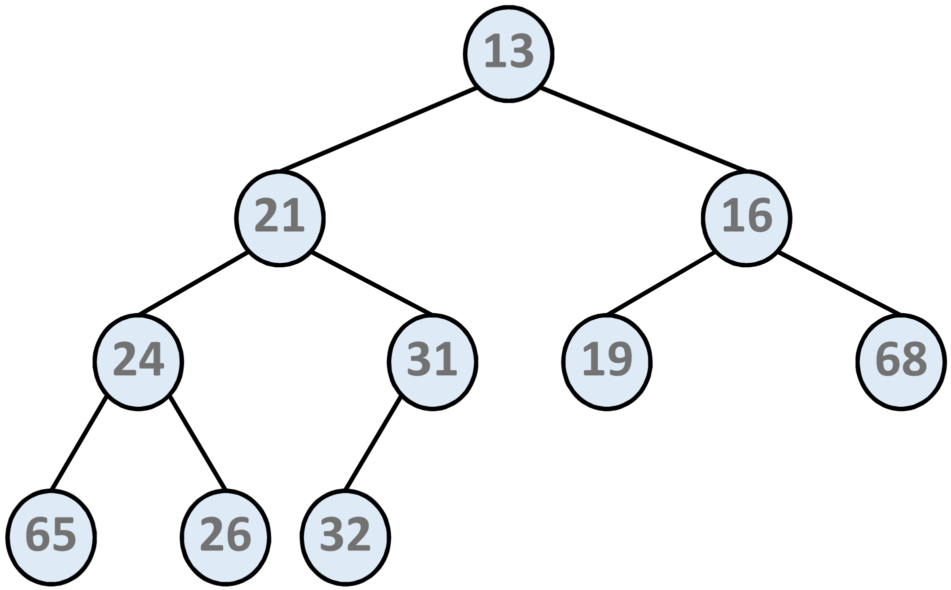
Step 1 : Create a hole in the next available location and put x in the hole.

Step 2 : Compare x with its parent. If heap order is not preserved, swap x with its parent. Repeat this step toward the root, until heap order is preserved.

This process is called

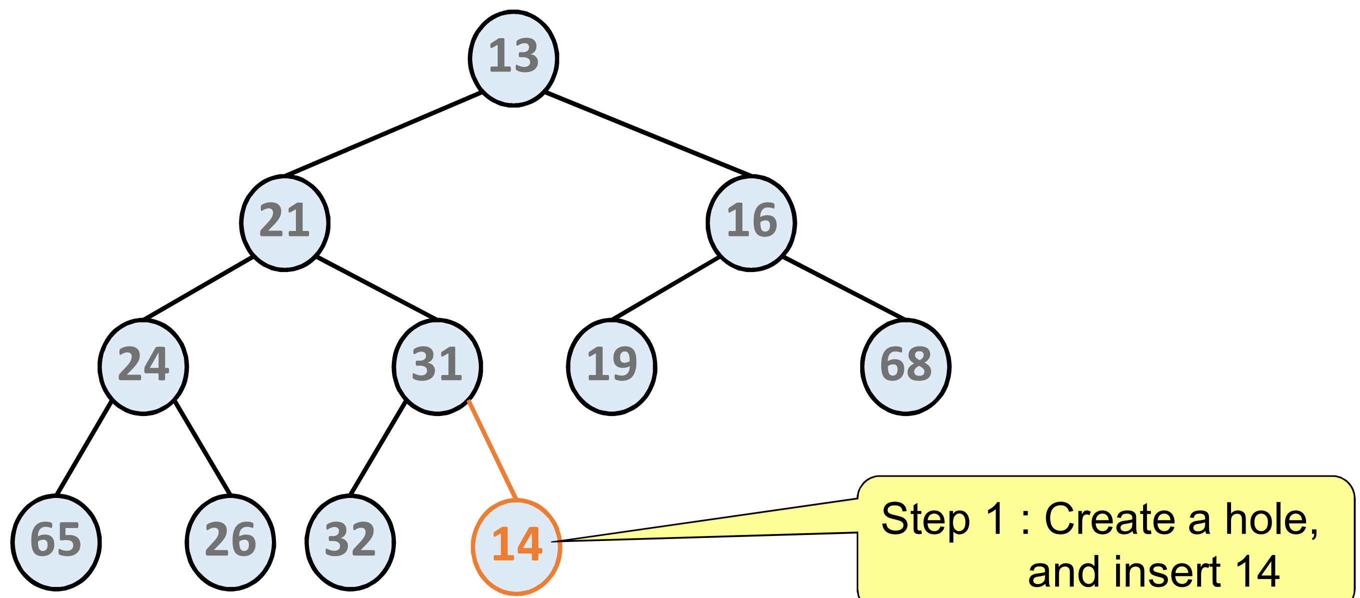
bubble-up/percolate-up/heapify-up/trickle-up.

Insert: Example (insert 14 into the heap)



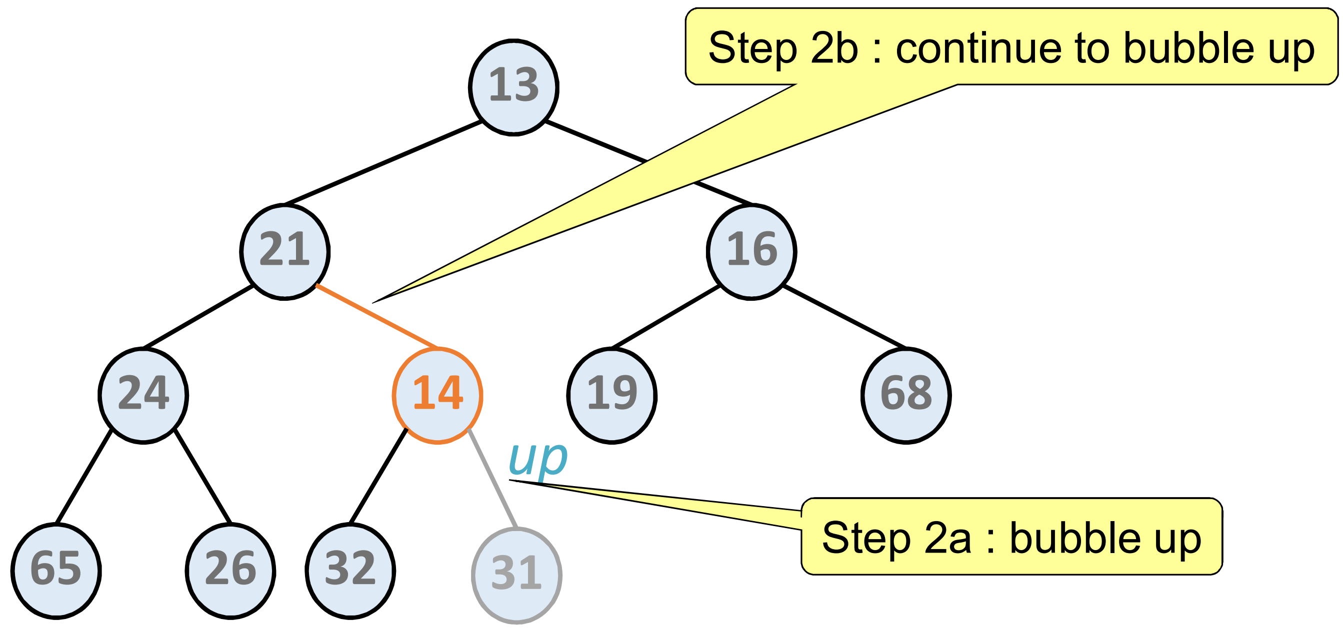
	13	**21**	**16**	**24**	**31**	**19**	**68**	**65**	**26**	**32**			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Insert: Example (insert 14 into the heap)



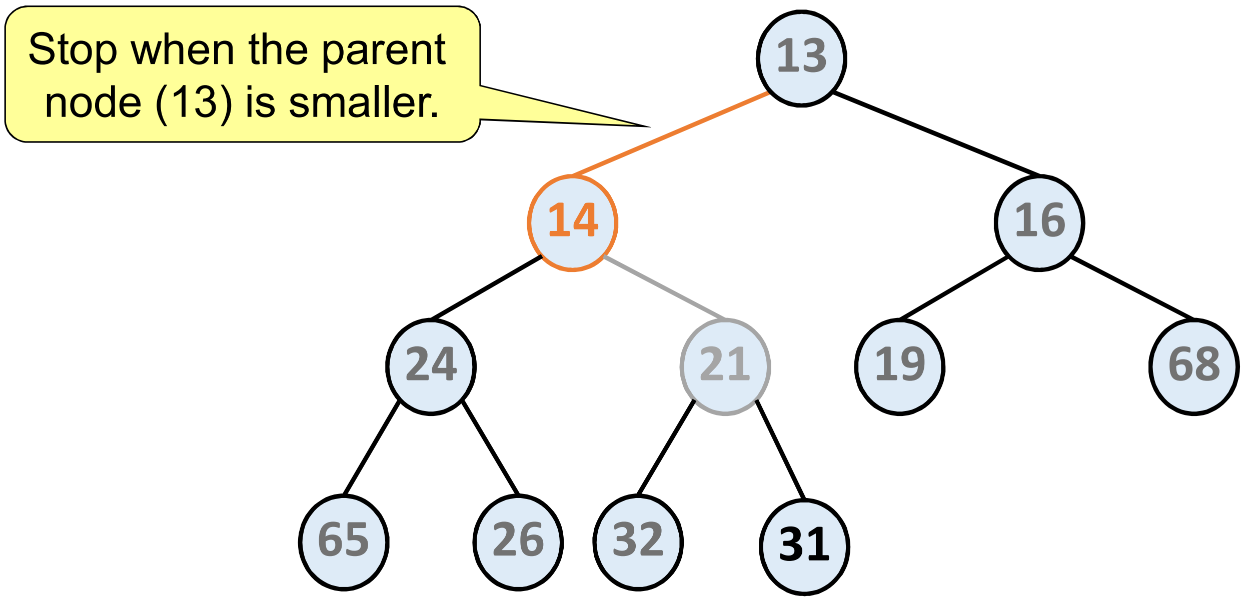
	13	**21**	**16**	**24**	**31**	**19**	**68**	**65**	**26**	**32**	**14**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Insert: Example (insert 14 into the heap)



	13	**21**	**16**	**24**	**~~31~~** **14**	**19**	**68**	**65**	**26**	**32**	**~~14~~** **31**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Insert: Example (insert 14 into the heap)



	13	**~~21~~** **14**	**16**	**24**	**~~14~~** **21**	**19**	**68**	**65**	**26**	**32**	**31**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Observations

- The number of comparisons during insert is $O(\log n)$ if the element is the new minimum and is bubbled all the way up to the root.
- It has been shown that 2.6 comparisons are required on average to perform an *insert*.
  - The average *insert* moves an element up 1.6 levels.

Heap: *delmin* (to removing the minimum which is located at the root.)

Step 1 : Remove the root and leave a hole.

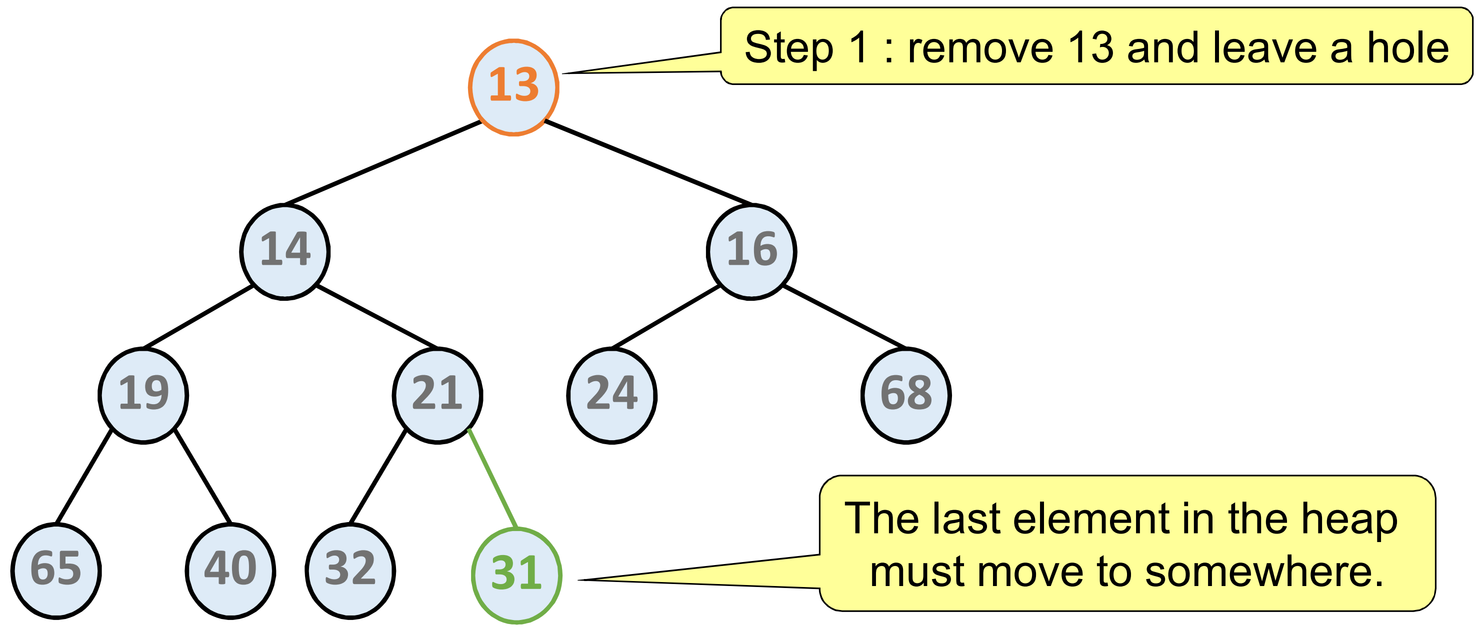
Step 2 : Delete the last element, x, of the heap.

Step 3 : Repeatly heapify the hole until heap order is preserved if x is placed to the hole.

This process is called

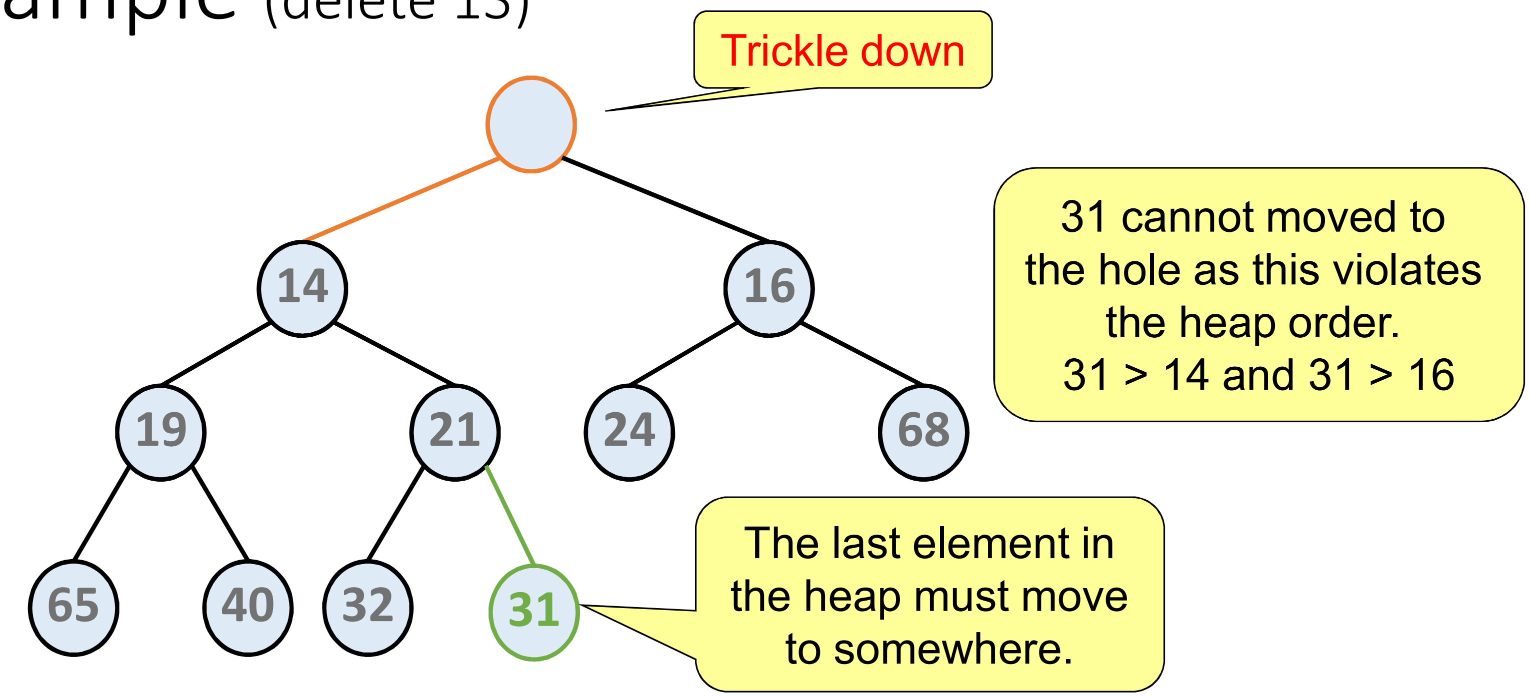
bubble-down/percolate-down/heapify-down/trickle-down.

delmin: Example (delete 13)



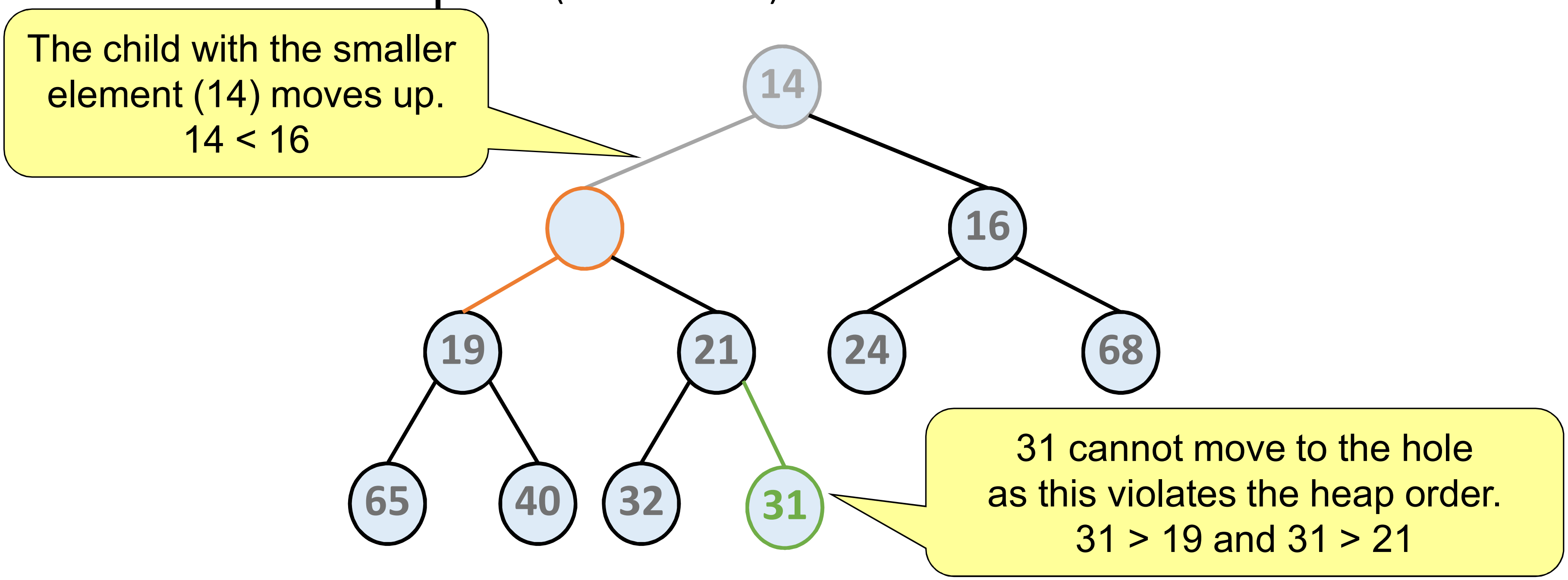
	~~13~~	14	16	19	21	24	68	65	40	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (delete 13)



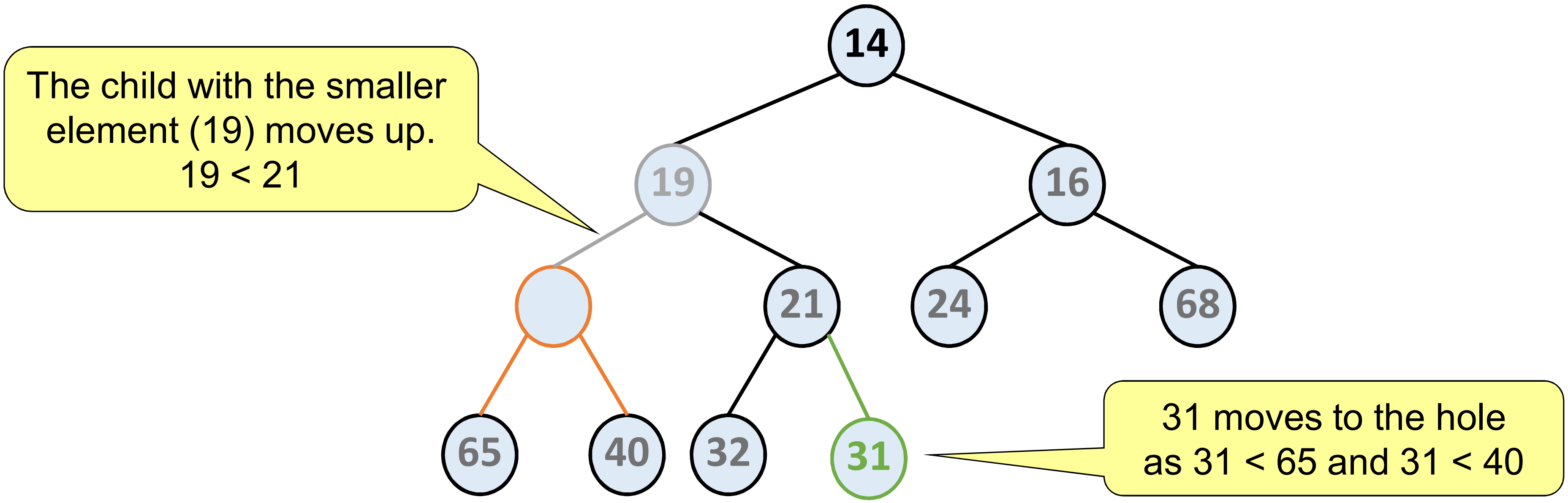
		14	**16**	**19**	**21**	**24**	**68**	**65**	**40**	**32**	**31**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (delete 13)



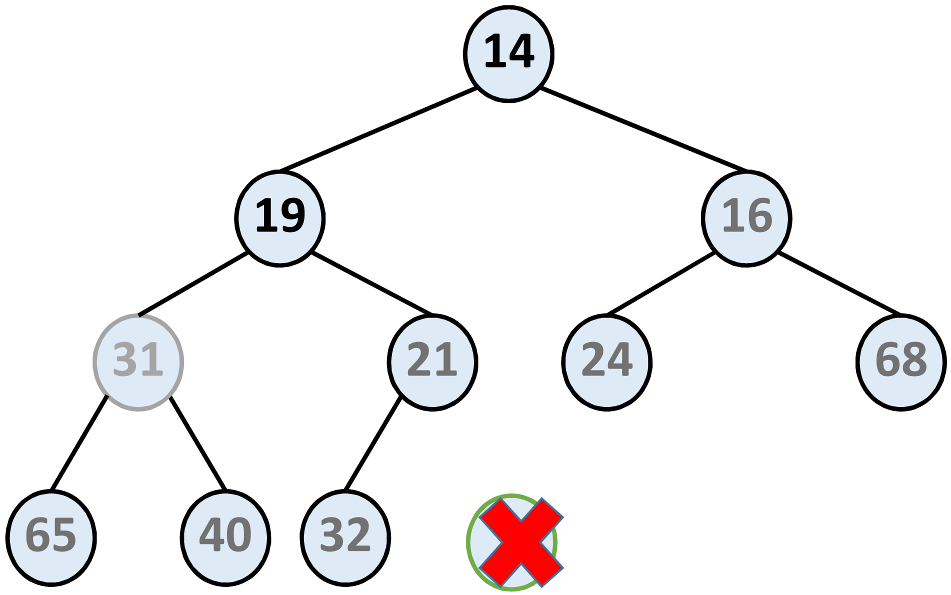
	14		16	19	21	24	68	65	40	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (delete 13)



	14	**19**	**16**		**21**	**24**	**68**	**65**	**40**	**32**	**31**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (delete 13)



	14	**19**	**16**	**31**	**21**	**24**	**68**	**65**	**40**	**32**			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap: Implementation (1)

Structure Declaration

Max capacity of the heap

Current size of the heap

Array to store the elements of the heap

```
typedef struct bst_s bst_t;
typedef struct heap_s {
    int capacity;
    int size;
    int *e;
} heap_t;
```

Set of common operations

```
heap_t *heap_init(int max_e);
void    heap_free(heap_t *h);
void    heap_make_empty(heap_t *h);
void    heap_insert(heap_t *h, int x);
int     heap_delete_min(heap_t *h);
int     heap_find_min(heap_t *h);
int     heap_is_full(heap_t *h);
int     heap_is_empty(heap_t *h);
void    heap_print(heap_t *h);
```

Heap: Implementation (2)

```
heap_t *heap_init(int max_e){
        heap_t *h = (heap_t *)malloc(sizeof(heap_t));
        h->e = (int *)malloc((max_e + 1)* sizeof(int));
        h->size = 0;
        h->capacity = max_e;
        h->e[0] = INT_MIN;
        return h;
}
```

The smallest value

Heap: Implementation (3)

	13	21	16	24	~~31~~ 14	19	68	65	26	32	~~14~~ 31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
void heap_insert(heap_t *h, int x){
        int i;
        if (heap_is_full(h)){
                fprintf(stderr, "The heap is full.\n");
                exit(1);
        }
        for (i = ++h->size; h->e[i / 2] > x; i /= 2)
                h->e[i] = h->e[i / 2];
        h->e[i] = x;
}
```

++size

Violation of heap order

Assign x to the hole

Swap with the parent

Move up one level

	~~13~~	14	16	19	21	24	68	65	40	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap: Implementation (4)

```
int heap_delete_min(heap_t *h){
        int i, child, min_e, last_e;
        if (heap_is_empty(h)){
                fprintf(stderr, "The heap is empty.\n");
                exit(1);
        }
        min_e = h->e[1];
        last_e = h->e[h->size--];
        for (i = 1; i * 2 <= h->size; i = child){
                /* Find smaller child */
                child = i * 2;
                if (child != h->size &&
                        h->e[child + 1] < h->e[child])
                        child++;
                /* trickle the hole down one level */
                if (last_e > h->e[child])
                        h->e[i] = h->e[child];
                else    break;
        }
        h->e[i] = last_e;
        return min_e;
}
```

Return the min

size --
last_e

Move down a level until reaching a leave node

If violate heap order, swap

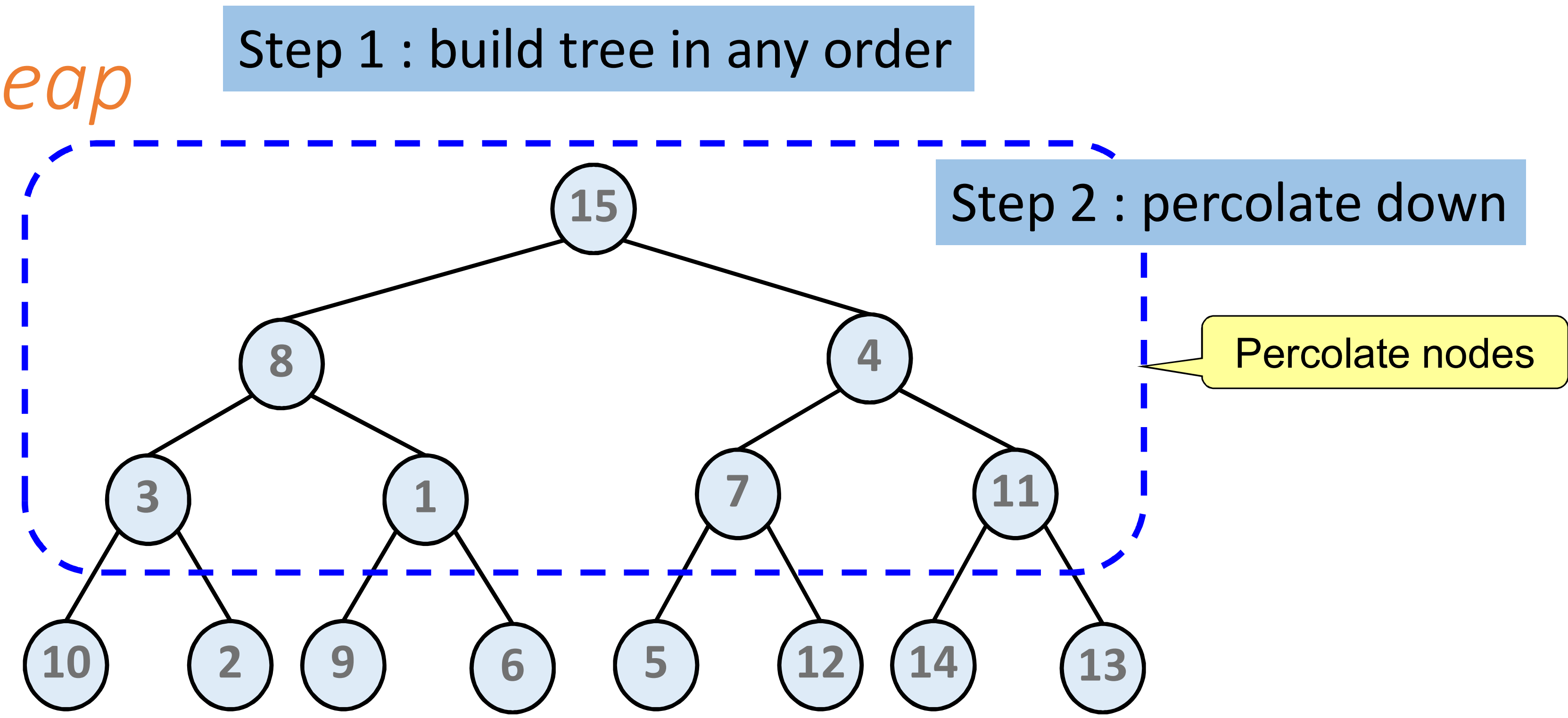
Other Heap Operations (min-heap)

- *findmin*: Finding the minimum can be performed in constant time.
- *findmax*: No help in finding the maximum
- *sort*: There is no strict ordering information
  - But can be used for sorting. (see heapsort)
- *decrease_key*(P, Δ): fixed by *bubble_up*
- *increase_key*(P, Δ): fixed by *trickle_down*
- *delete*: fixed by bubble up and trickle down
- *build_heap*

Observation on *build_heap*

- Method 1 :
  - Create an empty heap,
  - and perform n successive inserts.
  - This will take $O(n)$ average but $O(n \log n)$ worst-case.
- Method 2 :
  - Place the n keys into the tree in any order,
  - For node i = $\lfloor n/2 \rfloor$ down to 1, perform *trickle_down*.

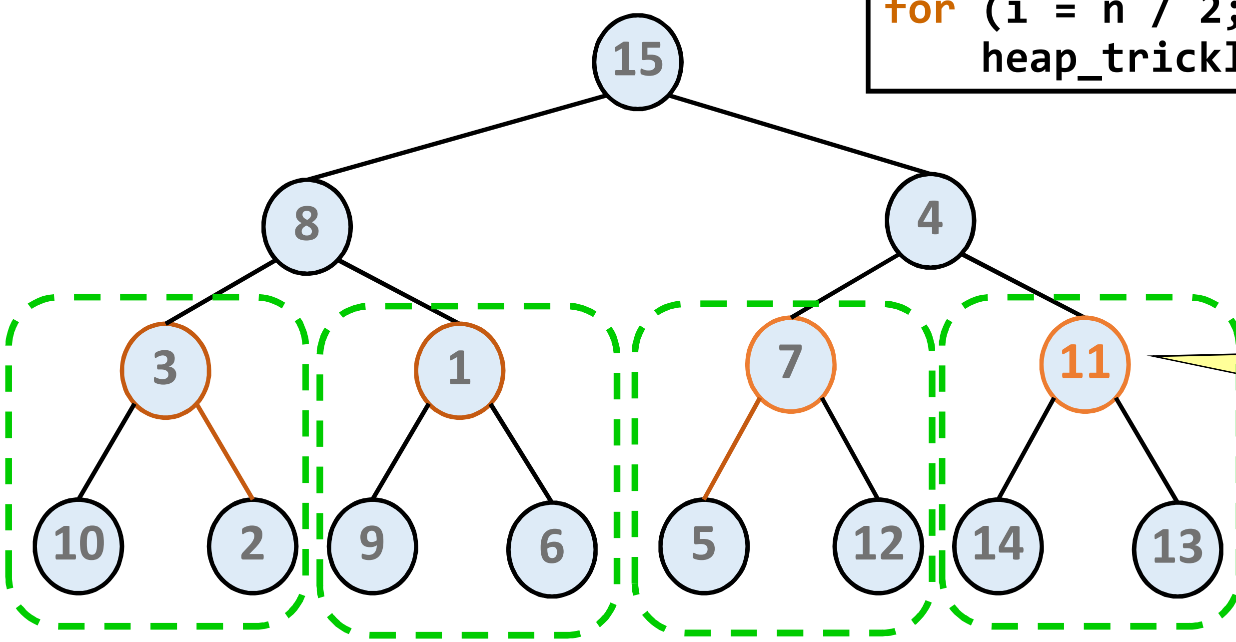
build_heap



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

build_heap: trickle_down

```
for (i = n / 2; i > 0; i--)
    heap_trickle_down(i);
```

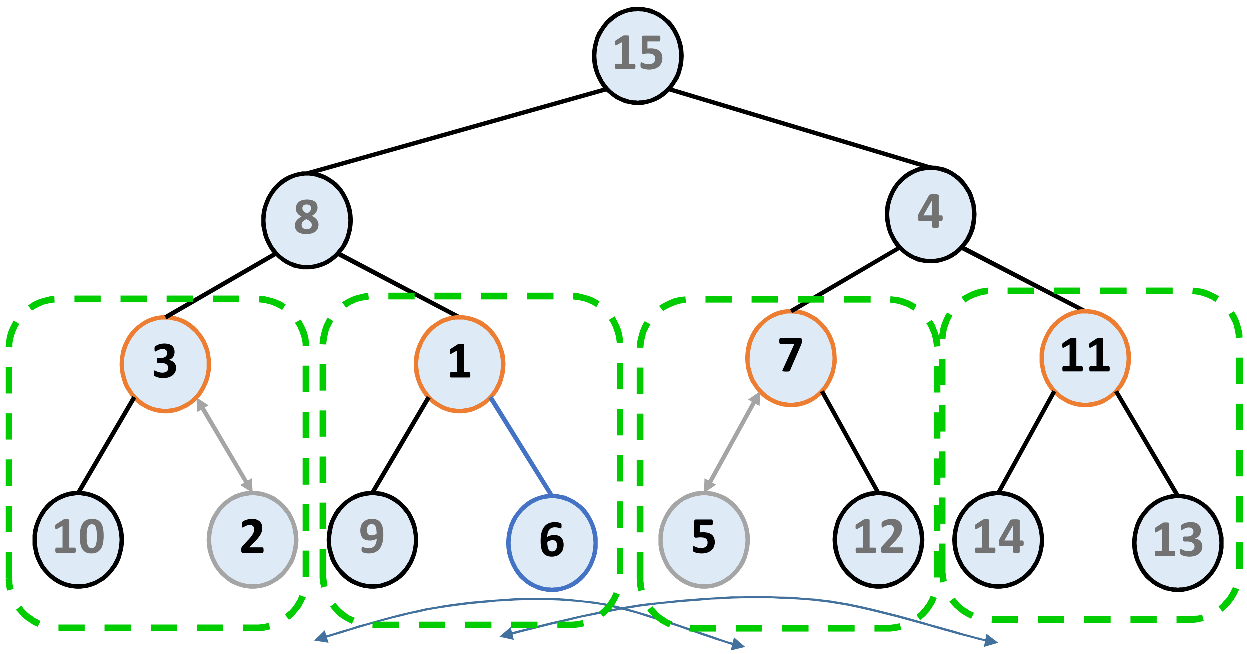


Percolate nodes

If the heap order is **not** preserved, swap with the smaller child.

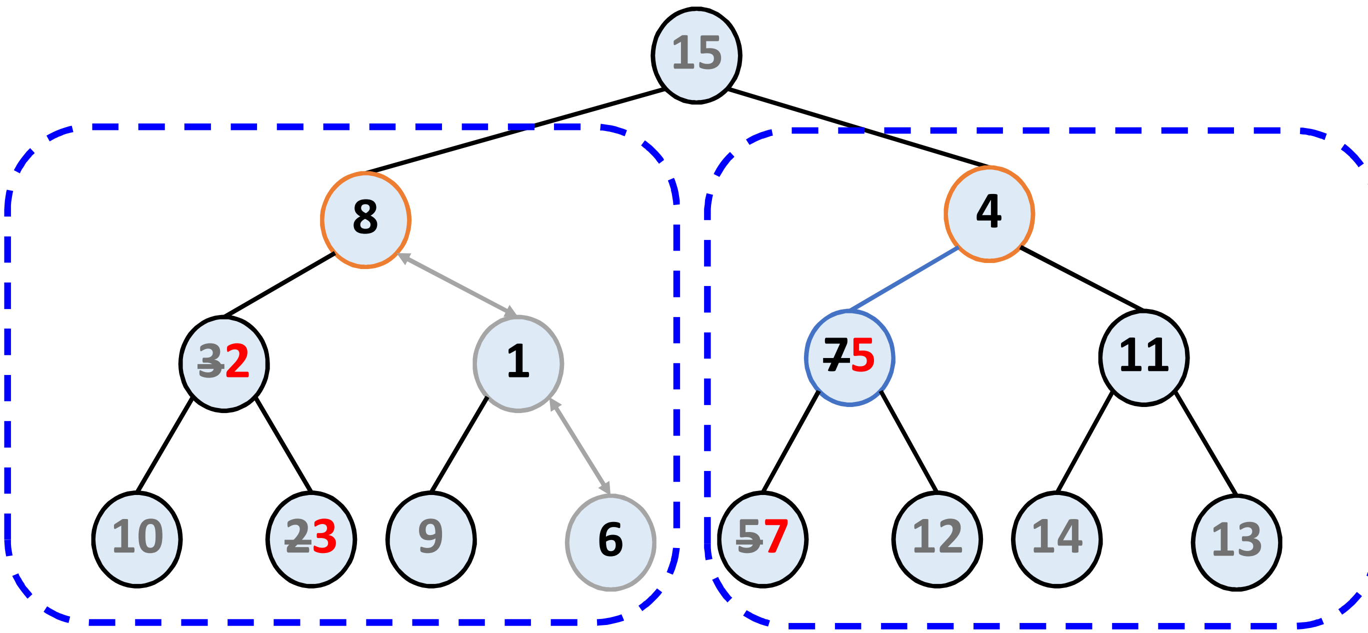
	15	**8**	**4**	**3**	**1**	**7**	**11**	**10**	**2**	**9**	**6**	**5**	**12**	**14**	**13**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

build_heap: trickle_down



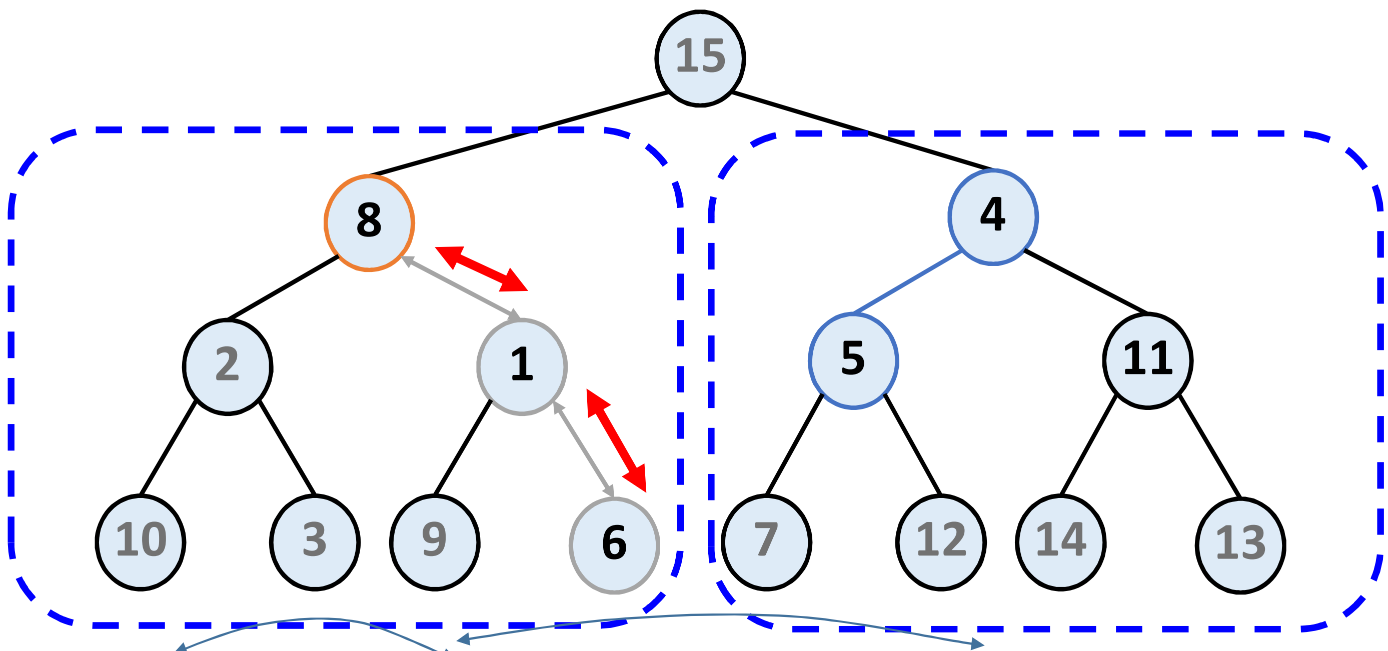
	15	**8**	**4**	**3**	**1**	**7**	**11**	**10**	**2**	**9**	**6**	**5**	**12**	**14**	**13**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

build_heap: trickle_down



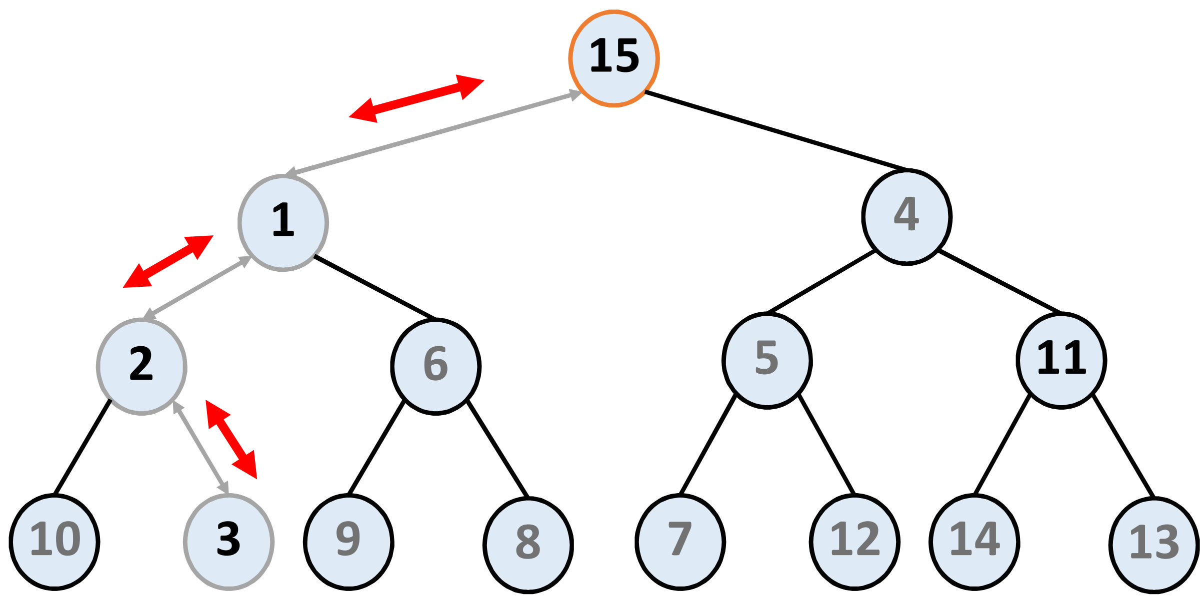
	15	**8**	**4**	**2**	**1**	**5**	**11**	**10**	**3**	**9**	**6**	**7**	**12**	**14**	**13**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

build_heap: trickle_down



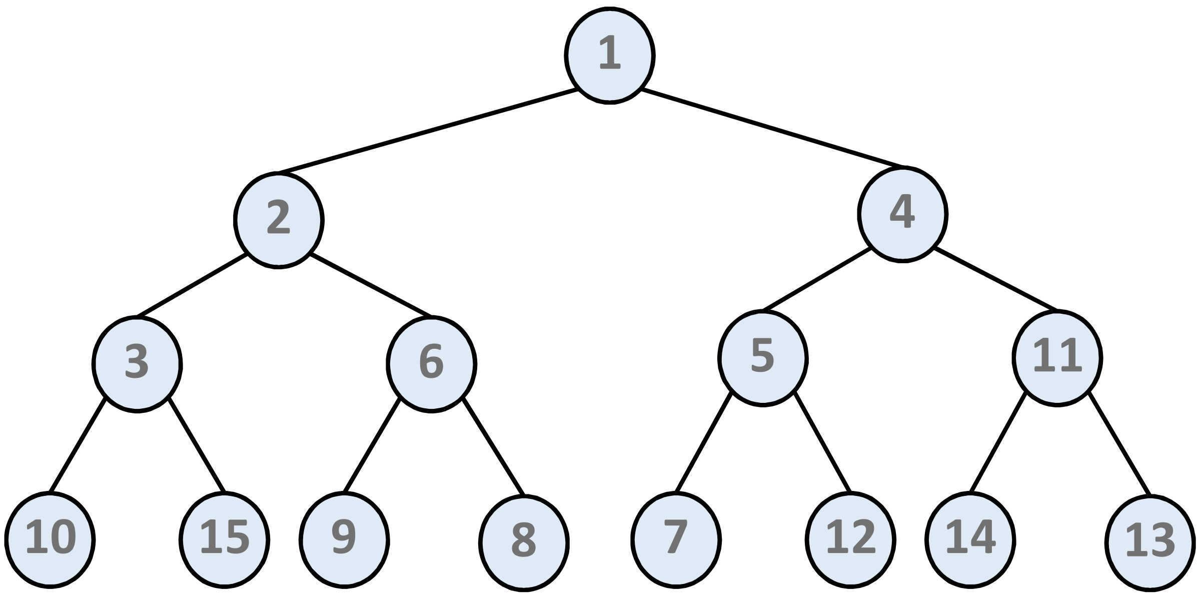
	15	**8**	**4**	**2**	**1**	**5**	**11**	**10**	**3**	**9**	**6**	**7**	**12**	**14**	**13**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

build_heap: trickle_down



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

build_heap: Final



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

Complexity of *build_heap*

- To analyze the running time of *build_heap*, the simplest analysis is that *n/2* nodes trickle down the tree *O*(*height-of-tree*) = *O*(*log n*) times, and thus the complexity is *O*(*n log n*)
- However, we observe that every node at height *h* actually trickles down at most *h* times, instead of trickling down *O*(*height-of-tree*) times
- This gives a tighter bound *O*(*n*)

Applications of Heaps

- Heapsort
- K- selection problem

Heapsort

- Priority queues can be used to sort in $O(n \log n)$ time.
- The basic strategy is to
  - (1) build a binary heap of n elements in $O(n)$ time
  - (2) perform n *delete_min*.
- We record the minimum elements that leaves in a second array and copy the array back to complete the sorting.
- Total running time is $O(n) + n \times O(\log n) = O(n \log n)$.

Tricks in Implementing Heapsort

- The memory requirement is doubled since we need an extra array.
- Avoid the second array by making use of the last cell in the array to store the value returned by *delete_min*.
  - Using this strategy the array will contain the elements in decreasing sorted order after the last *delete_min*.
- Suppose we stick to the more typical increasing order, we can change the heap ordering property so that the parent has a larger key than the child.
  - We use a maxheap with a *delete_max* operation.

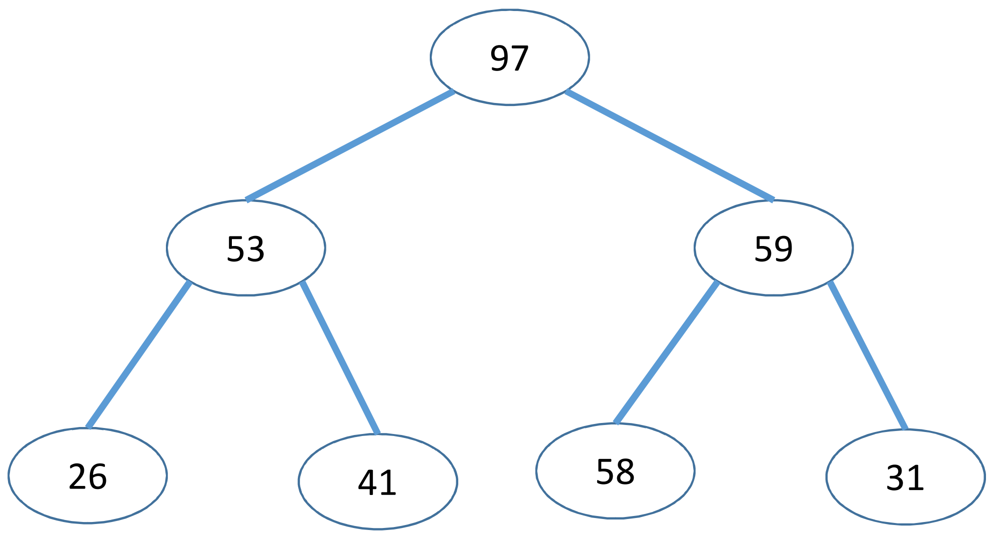
	53	41	31	26	58	59	97

heap

Sorted values

Heapsort Example (1)

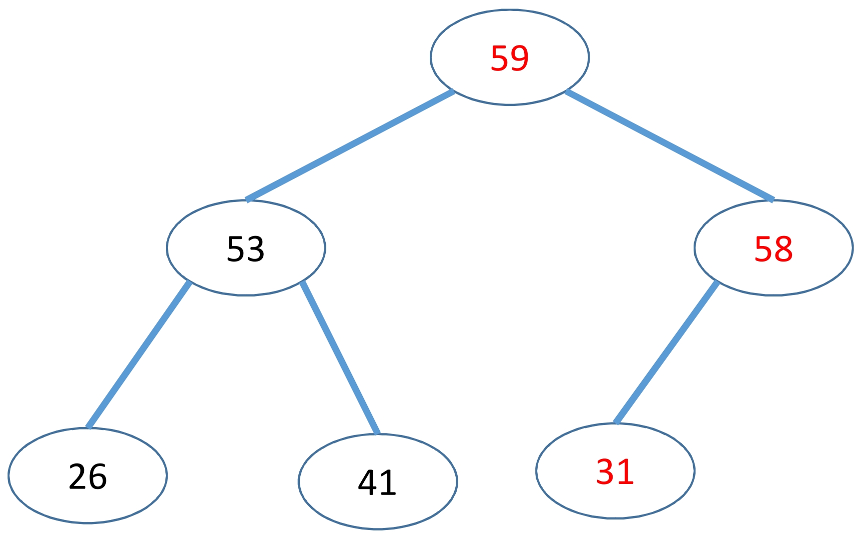
- Maxheap with its array representation. Execute *delete_max*.



	97	53	59	26	41	58	31

Heapsort Example (2)

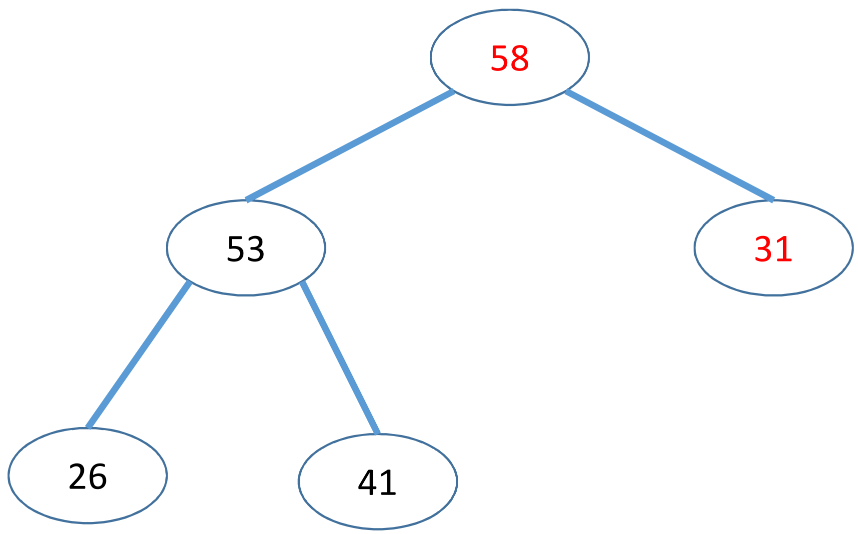
- The maxheap after *delete_max*.



	59	53	58	26	41	31	97

Heapsort Example (3)

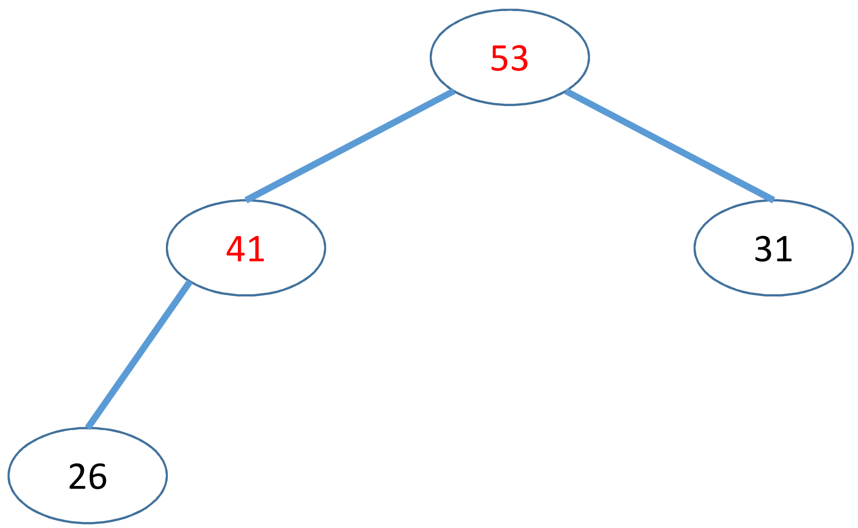
- The maxheap after *delete_max*.



	58	53	31	26	41	59	97

Heapsort Example (4)

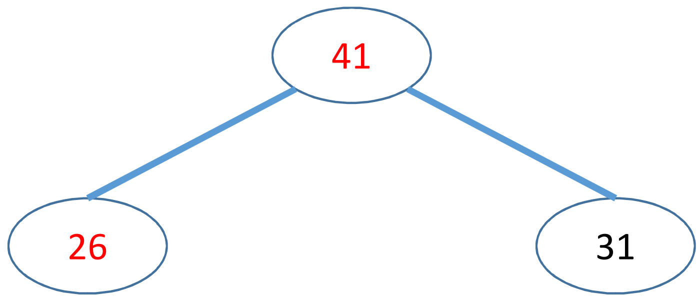
- The maxheap after *delete_max*.



	53	41	31	26	58	59	97

Heapsort Example (5)

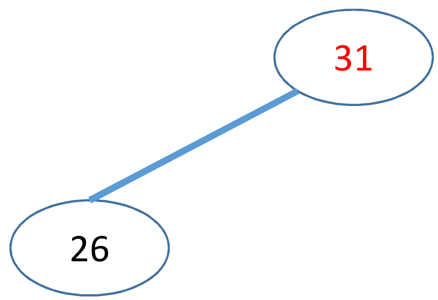
- The maxheap after *delete_max*.



	41	26	31	53	58	59	97

Heapsort Example (6)

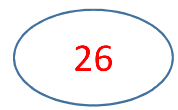
- The maxheap after *delete_max*.



	31	26	41	53	58	59	97

Heapsort Example (7)

- The maxheap after *delete_max*.



	26	31	41	53	58	59	97

Heapsort Example (8)

- The maxheap after *delete_max*.

	26	31	41	53	58	59	97

The k-selection Problem

- Problem: Suppose you have a group of n numbers and would like to determine the k-th largest.
- First Algorithm
  - Build a max-heap for all numbers and it takes $O(n)$.
  - Keep *delmax* until we get the k-th value returned. k x $O(\log n)$.
  - The total running time is $O(n + k \log n)$.
- For small k then the running time dominated by the heap building operation and is $O(n)$.
- For larger values of k, the running time is $O(k \log n)$ time.

The *k*-selection Problem (2)

- Second Algorithm
  1. Build a smaller min-heap of k elements.
  2. Then compare the remaining (n - k) numbers against the heap.
  3. If the new element is larger, it replaces the root, otherwise it is discarded.
  4. When the algorithm teminates, the heap contains the k largest numbers from the set.
- To build a k-element the heap takes $O(k)$.
- The time for step 2 is
  - $O(1)$: to test if the element goes into the heap
  - + $O(\log k)$: to delete the root and insert the new element if this is necessary
- The total time is $O(k + (n - k)\log k) = O(n \log k)$.

Summary

- Priority Queue ADT:
  - Pick largest/smallest element + insert
- Binary heap: structure & order properties
  - Efficient array implementation
  - *findmax*/*findmin* in constant time
  - fixing heap properties in $O(\log n)$ time.
  - $O(n)$ heap construction
- Heapsort : O(n log n) comparisons sorting
- Application: k-selection problem
  - reveal theoretical bound of $O(n \log n)$ in finding the median of a set of n numbers.