Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Binary Heap & Priority Queue

Last updated: **16/03/2011**

Tang Wai Chung, Matthew

Binary Heap (or Just Heap)

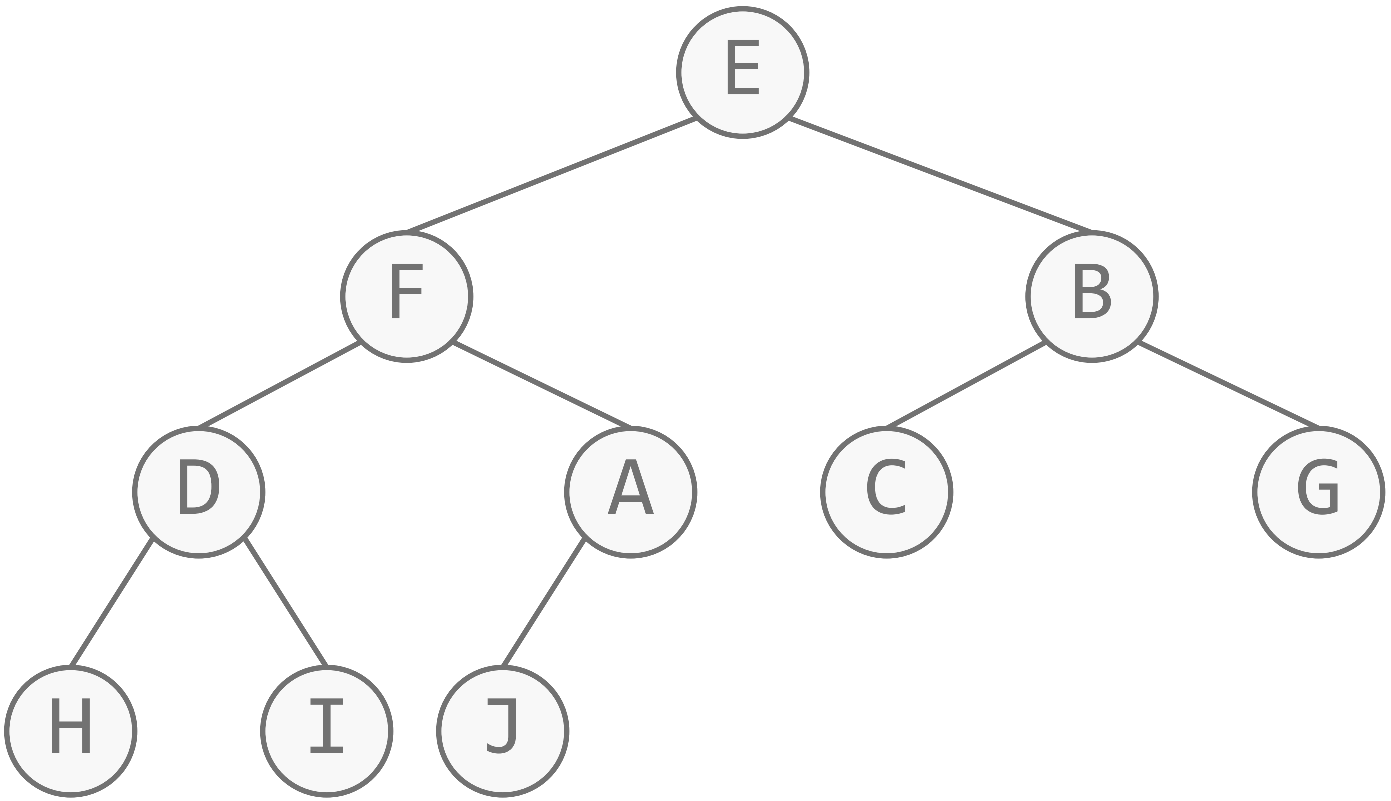
- Heaps have two properties
  - **Structure** property & **Heap-order** property
- An operation on a heap can **destroy** one of the properties,
  - A heap operation must not terminate until all heap properties are restored.

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: Illustration



A Complete binary tree

Height of Heaps

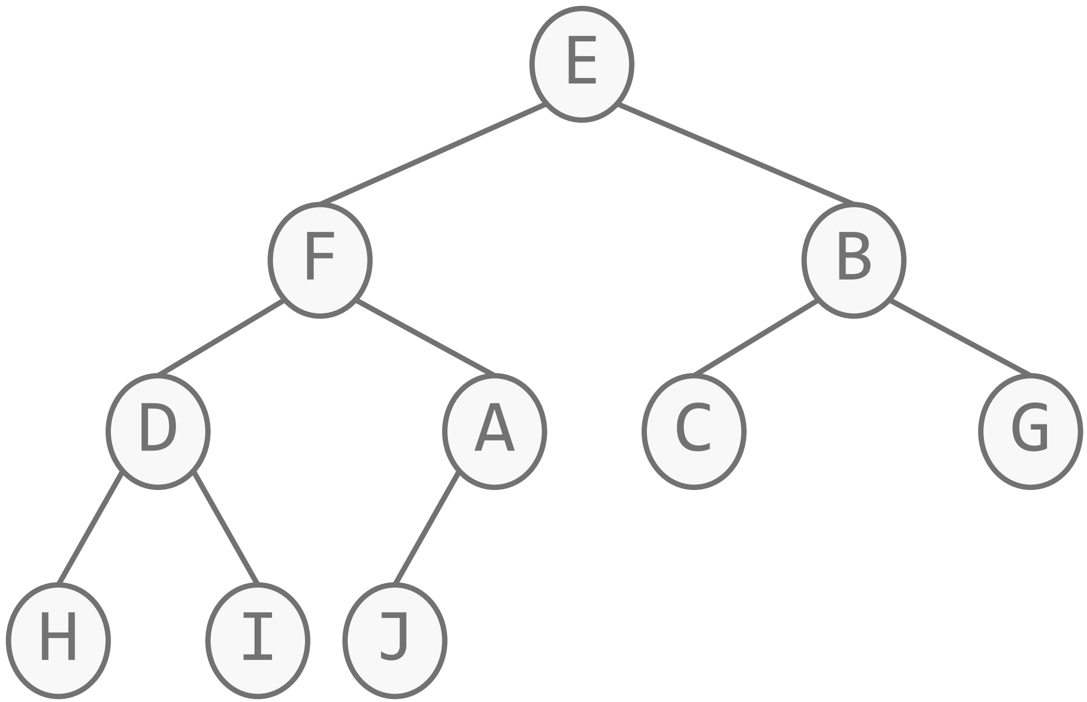
- A complete binary tree of height H at least 2^H and at most 2^{H+1} - 1 nodes.
- This implies that the height of a complete binary tree is

$$\lfloor \lg N \rfloor = O(\lg N)$$

- Because a complete binary tree is so regular, it can be represented in an **array**.
  - This encourages a straight-forward **pointer-free** implementation.
  - We may **heapify** an ordinary array to turn it into a priority queue.

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



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 **2*i***
  - **right** child: position **(2*i* + 1)**,
  - **parent**: position $\lfloor i/2 \rfloor$
- No pointers are required, and the operations required to traverse the tree are extremely **simple**.
  - **Bit shifting** can be used to replace multiplications of 2.
- 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.
- For a heap, the largest/smallest element is placed at the root so that we can find it in constant time.
- Thus, we get the extra operation, ***findmax***/***findmin***, 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

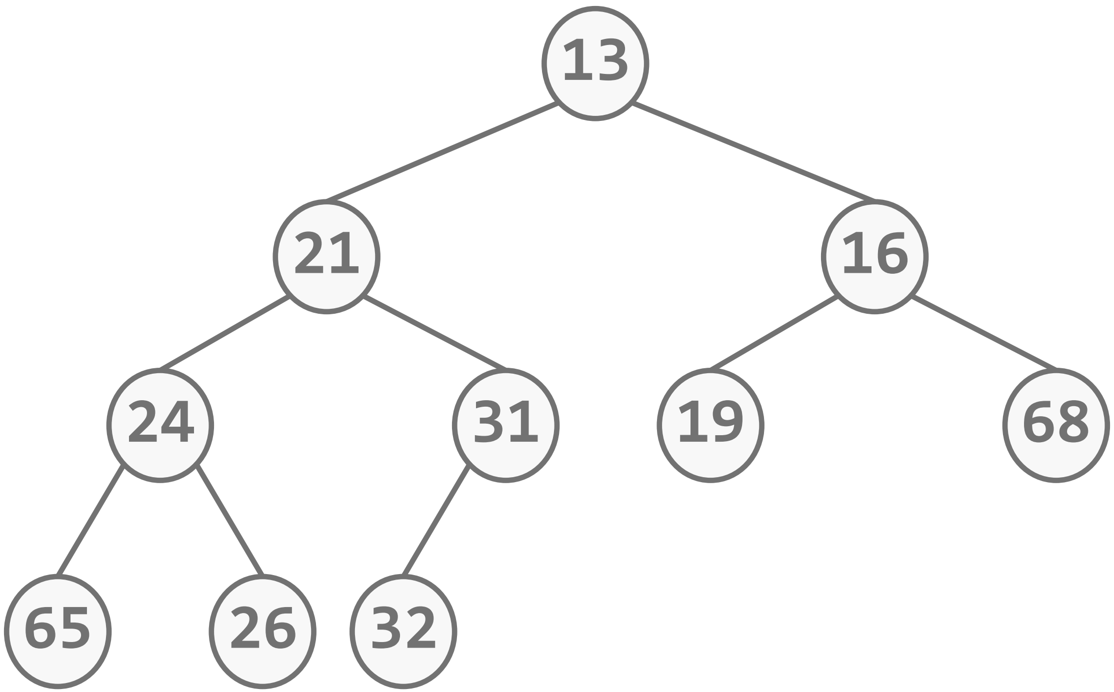
Heap Order Property

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**.
- We will first begin our discussions with min-heaps.

Heap: Example

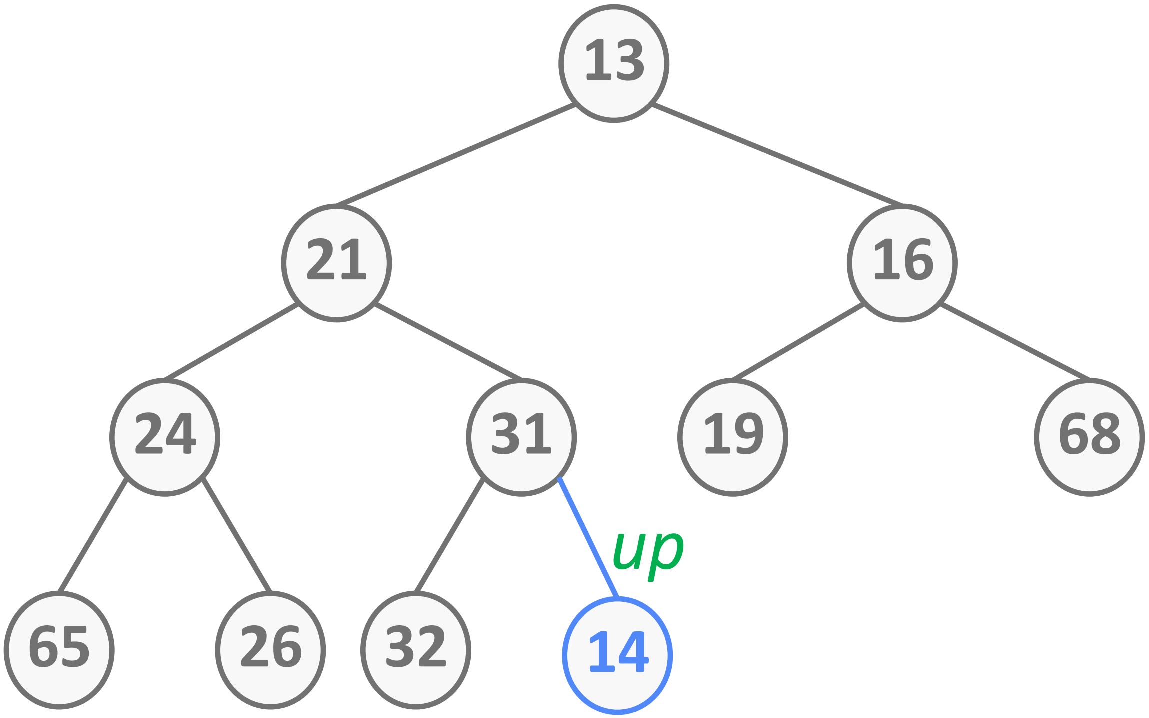


*Can you give a **max-heap** with the same set of keys?*

Heap: *insert*

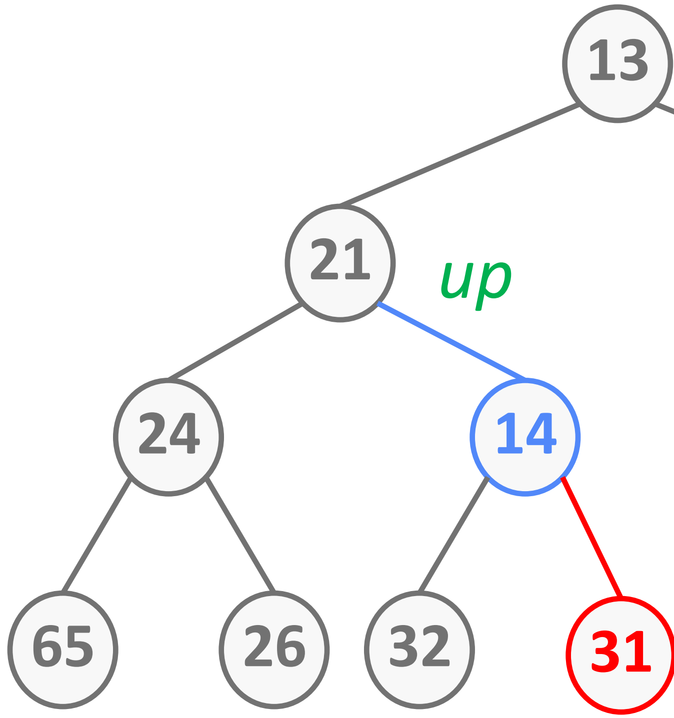
- We create a **hole** in the next available location.
  - If *x* can be placed in the hole without violating the heap order, just do it.
  - Otherwise we slide the element that is in the holes parent node into the hole, thus bubbling the hole **up** toward the root.
- We continue this process until *x* can be placed in the hole.
- This strategy is known as a **percolate up**/**upward heapify**/**upward fixing**/**promotion**.

insert: Example



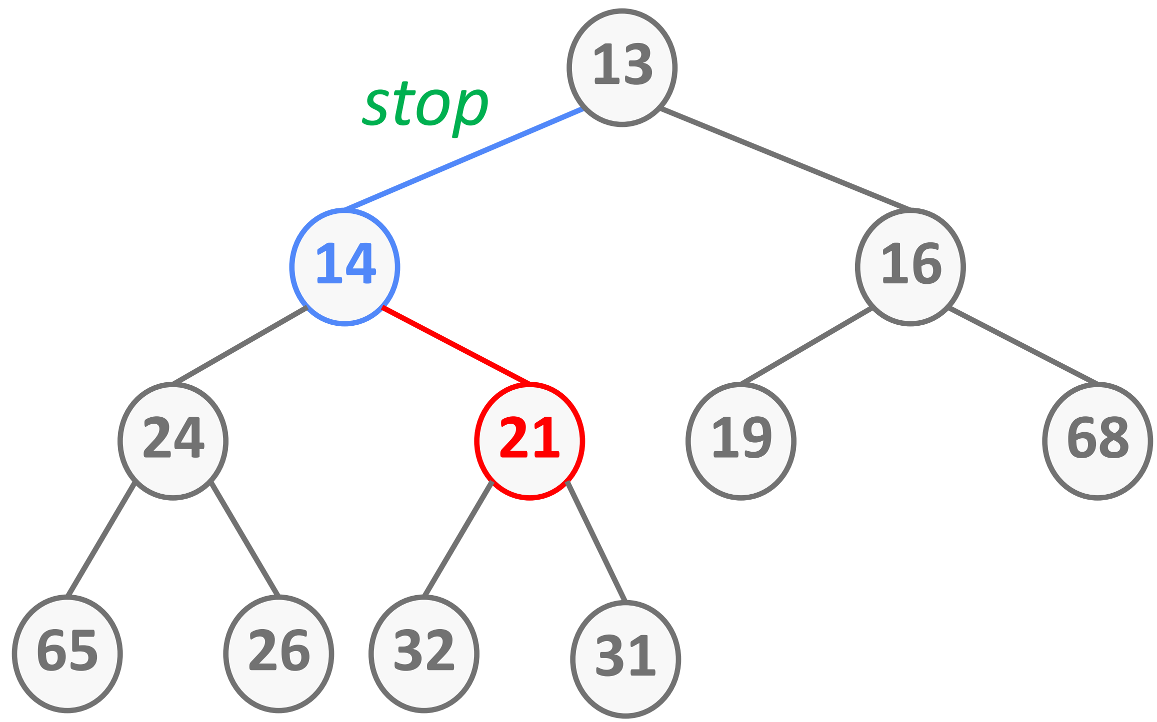
	13	21	16	24	31	19	68	65	26	32	14		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

insert: Example (2)



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

insert: Example (3)



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

Observations

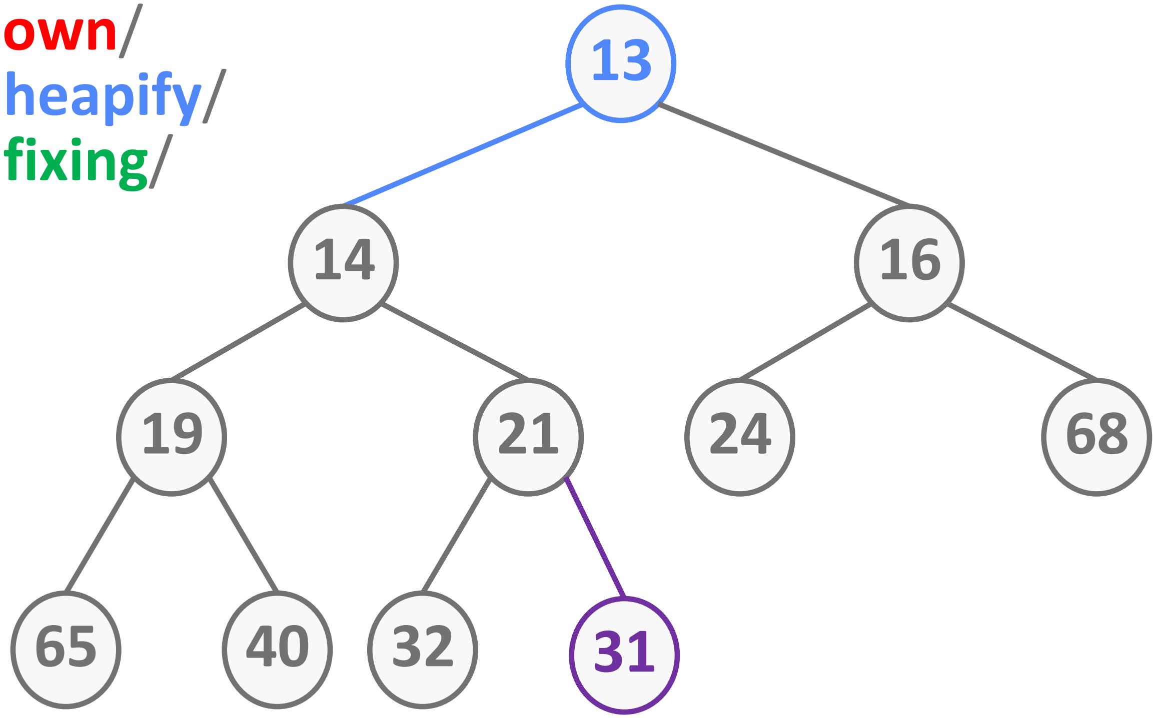
- The number of comparisons during insert is **$O(\lg N)$** if the element is the new **minimum** and is percolated all the way up to the **root**.
- It has been shown that **2.6** comparisons are required on **average** to perform an ***insert***.
  - The average ***insert*** moves an element up **1.6**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

Heap: *delmin*

- Finding the minimum is easy; the hard part is removing it.
- When the root is removed, a **hole** is created.
- To maintain **structure** property, we need to slide **down** the hole until we can put the **last element** (x) in, maintaining the **heap-order** property.
  - Or we may swap the root with the last element, then begin the sliding down.
- We then need to slide the **smaller** children of the node up into the hole, then push the hole down one level.
- We repeat this step until x can be placed in the hole.

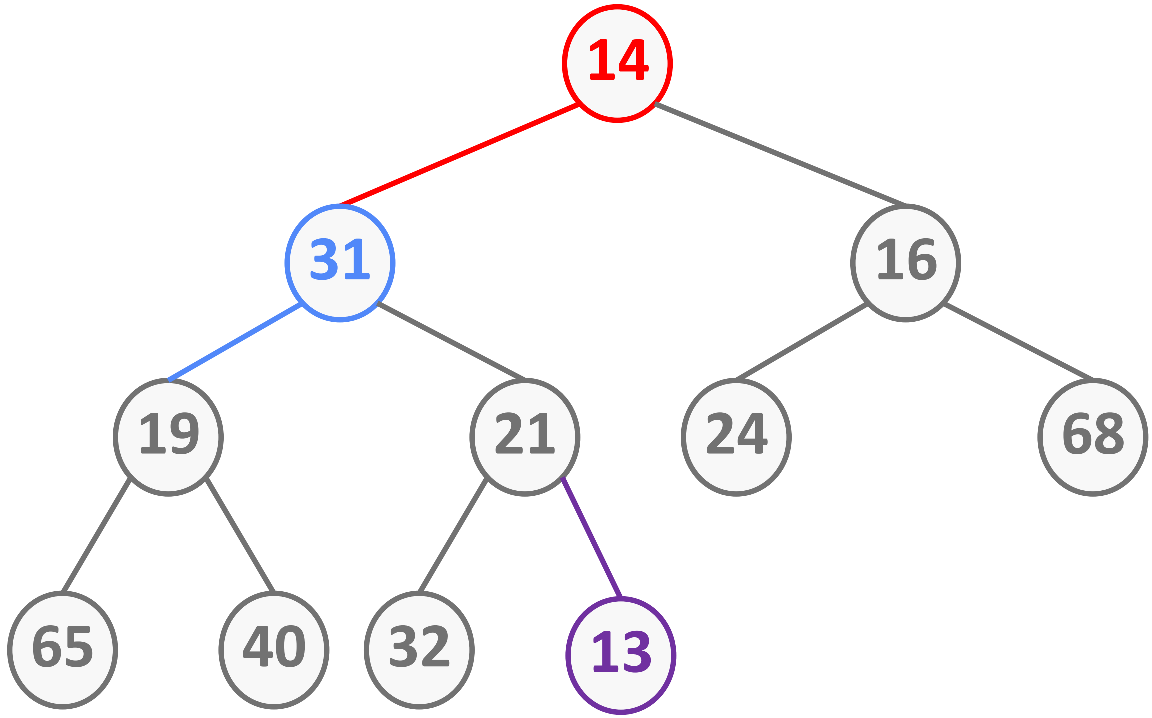
delmin: Example

percolate down/
downward heapify/
downward fixing/
demotion



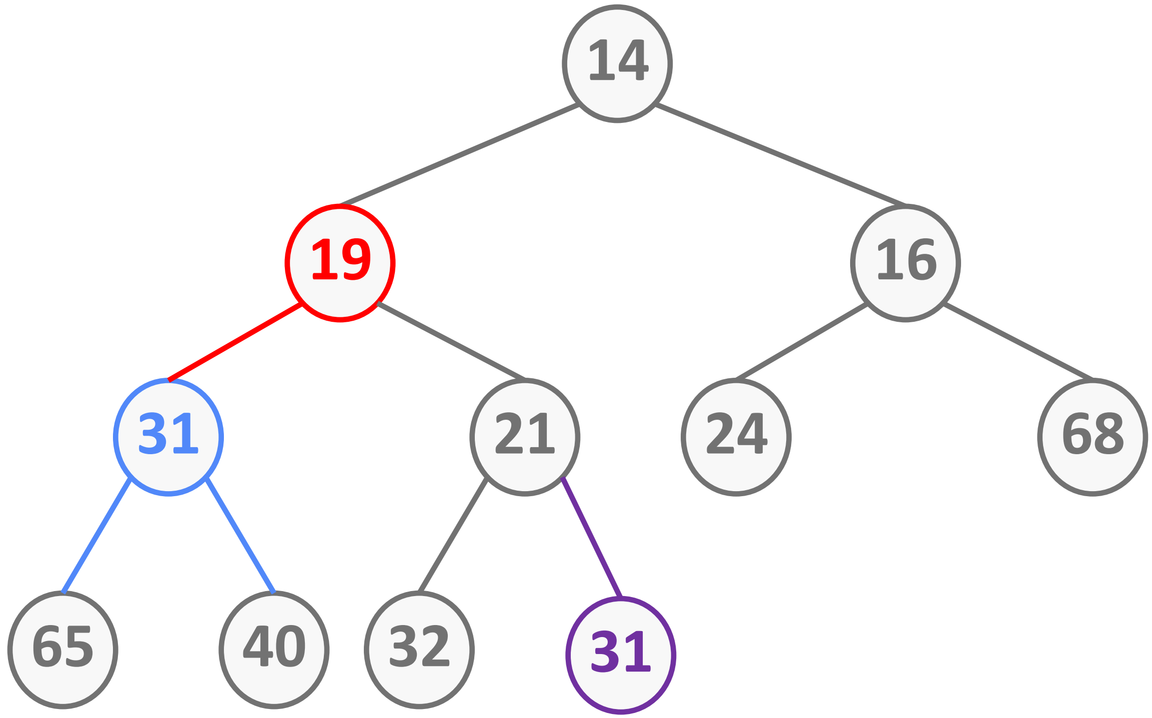
	13	14	16	19	21	24	68	65	40	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (2)



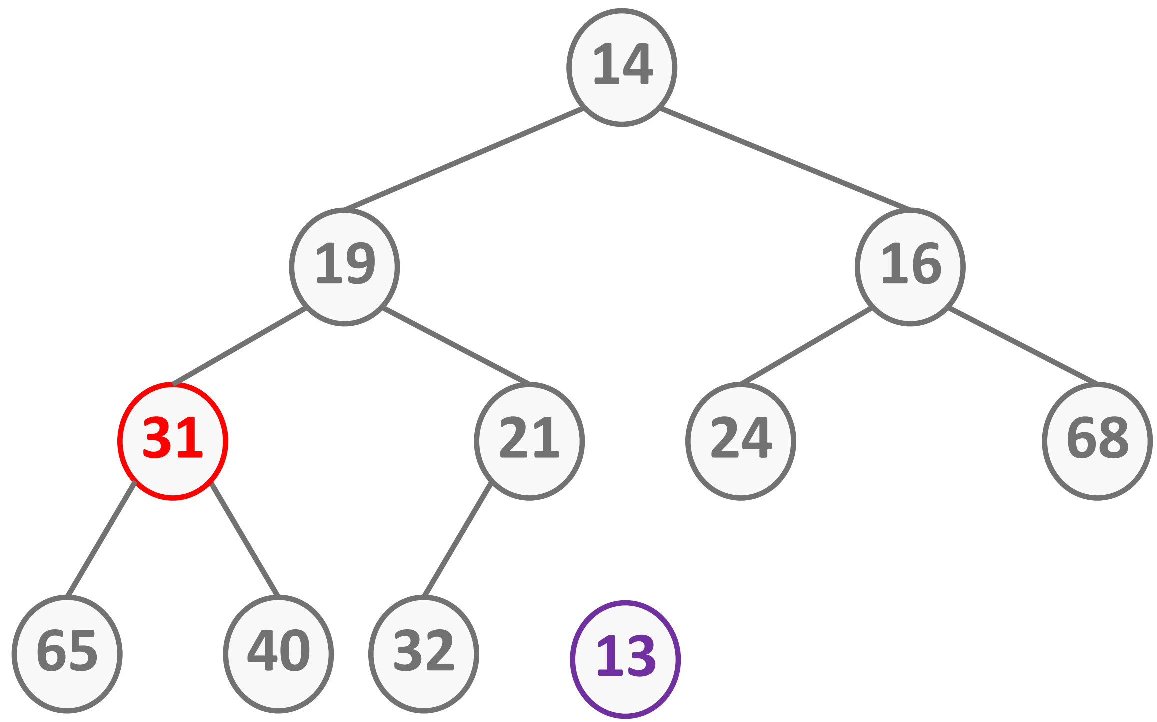
	14	31	16	19	21	24	68	65	40	32	13		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

delmin: Example (3)



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

delmin: Example (4)



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

Fixing Up/down: Code (min-heap)

Travel along the ancestors and swap if out of order.

```
void fix_up(int a[], int i){
    for ( ; a[i / 2] >= a[i]; i /= 2)
        swap(&a[i / 2], &a[i]);
}
```

Travel along the decentants and swap if out of order.

```
void fix_down(int a[], int n, int i){
    int j; /* child */

    for ( ; i * 2 <= n; i = j){
        j = i * 2; /* find smaller child */
        if (j < n && a[j + 1] < a[j]) j++;
        if (a[i] <= a[j])
            break;
        /* swap the values */
        swap(&a[i], &a[j]);
    }
}
```

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): update key and then ***fix_up***
- ***increase_key***(P, Δ): update key and then ***fix_down***
- ***delete***: fixed by fixing up then down
- ***build_heap***

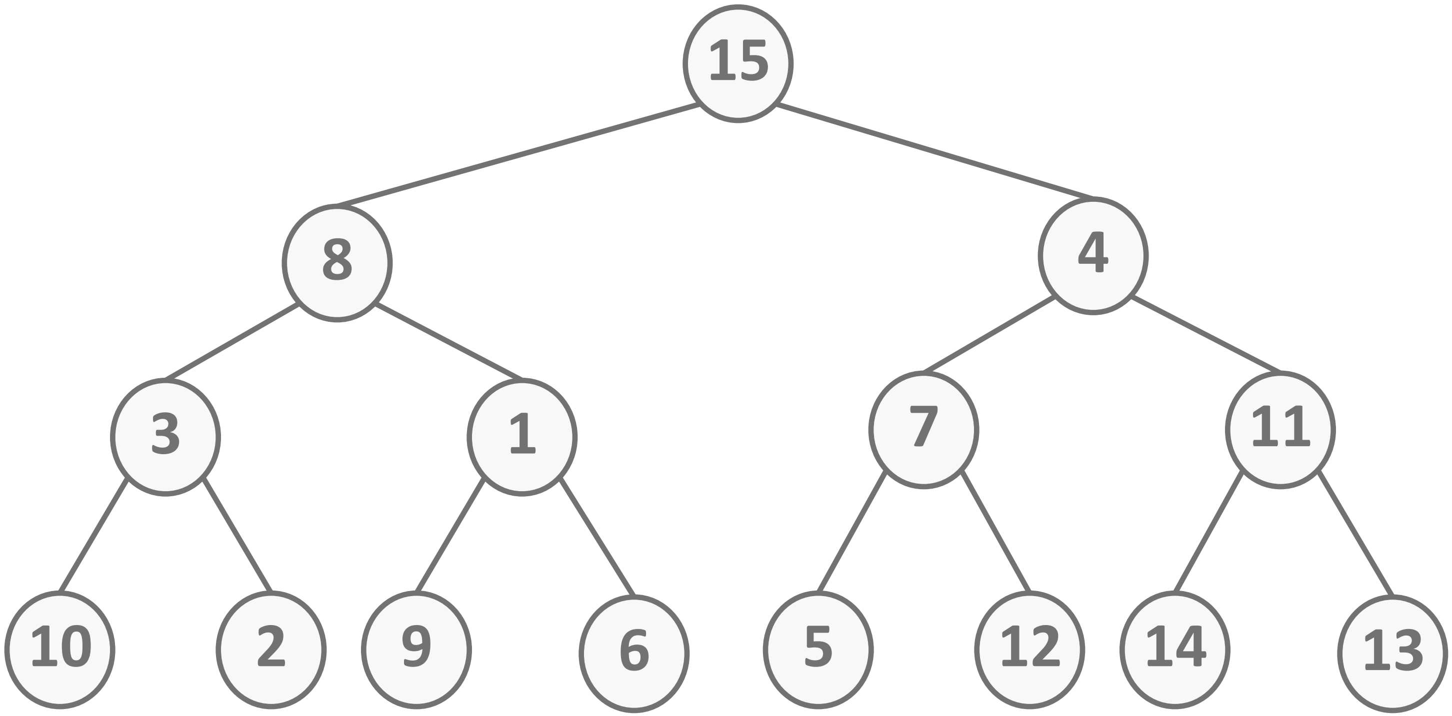
Building a Heap

- Takes N keys and places them into an empty heap.
- Perform N **successive** inserts.
  - This will take $O(N)$ average but $O(N \lg N)$ worst-case.
- Another **smarter** way is to place the N keys into the tree in any order (or just take the initial order) ...
- then perform ***fix_down*** on half of the nodes.
  - since there is no need to fix leaves.
- We call this procedure: ***heapify***

```
for (i = n / 2; i > 0; i--)
    fix_down(a, n, i);
```

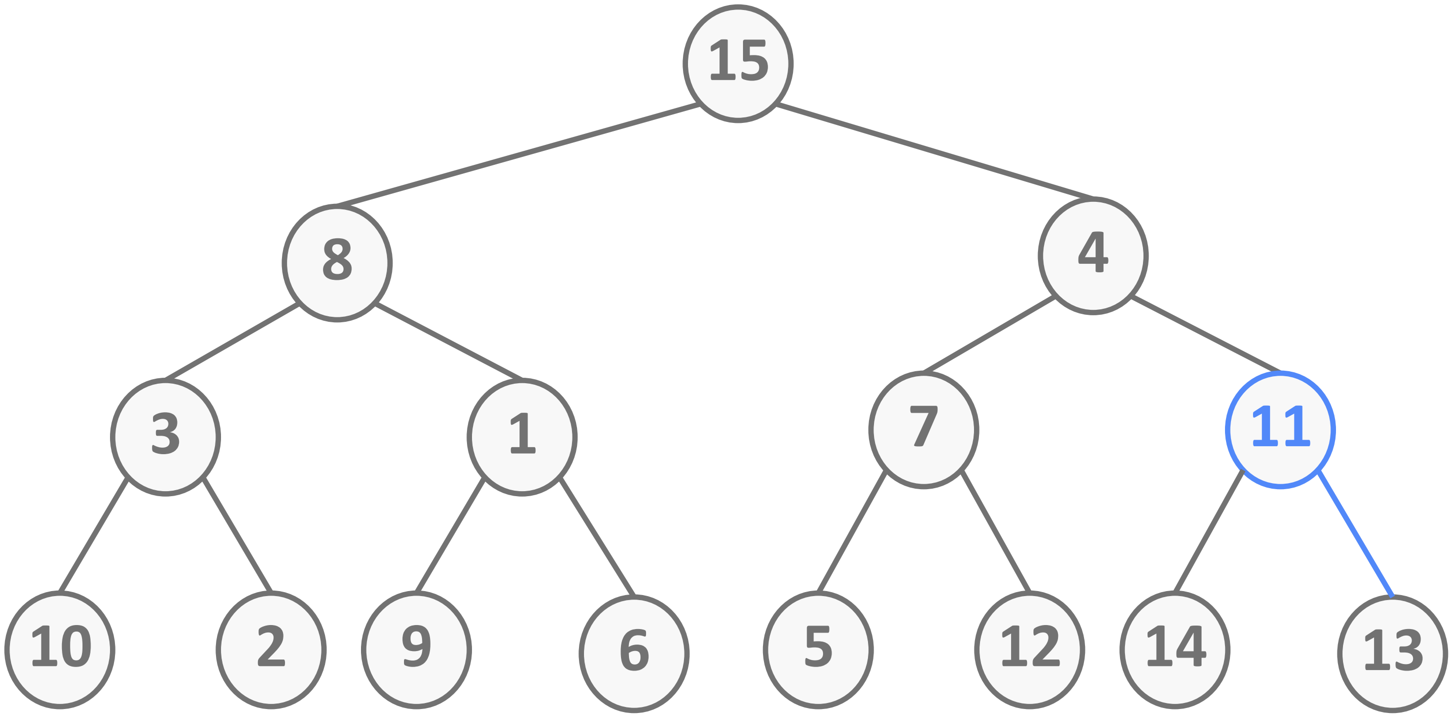
What is the worse-case complexity of this algorithm?

heapify: Initial



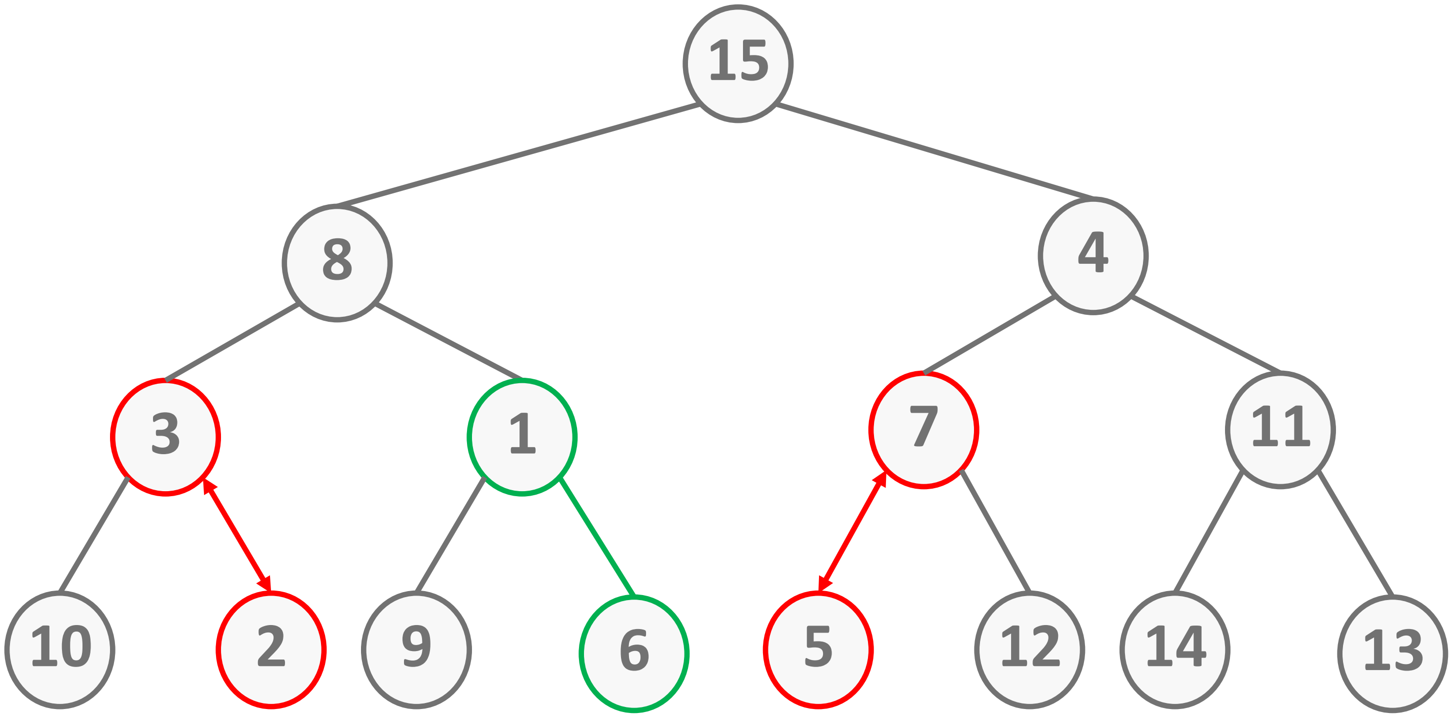
	15	8	4	3	1	7	11	10	2	9	6	5	12	14	13
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

heapify: fix_down(7)



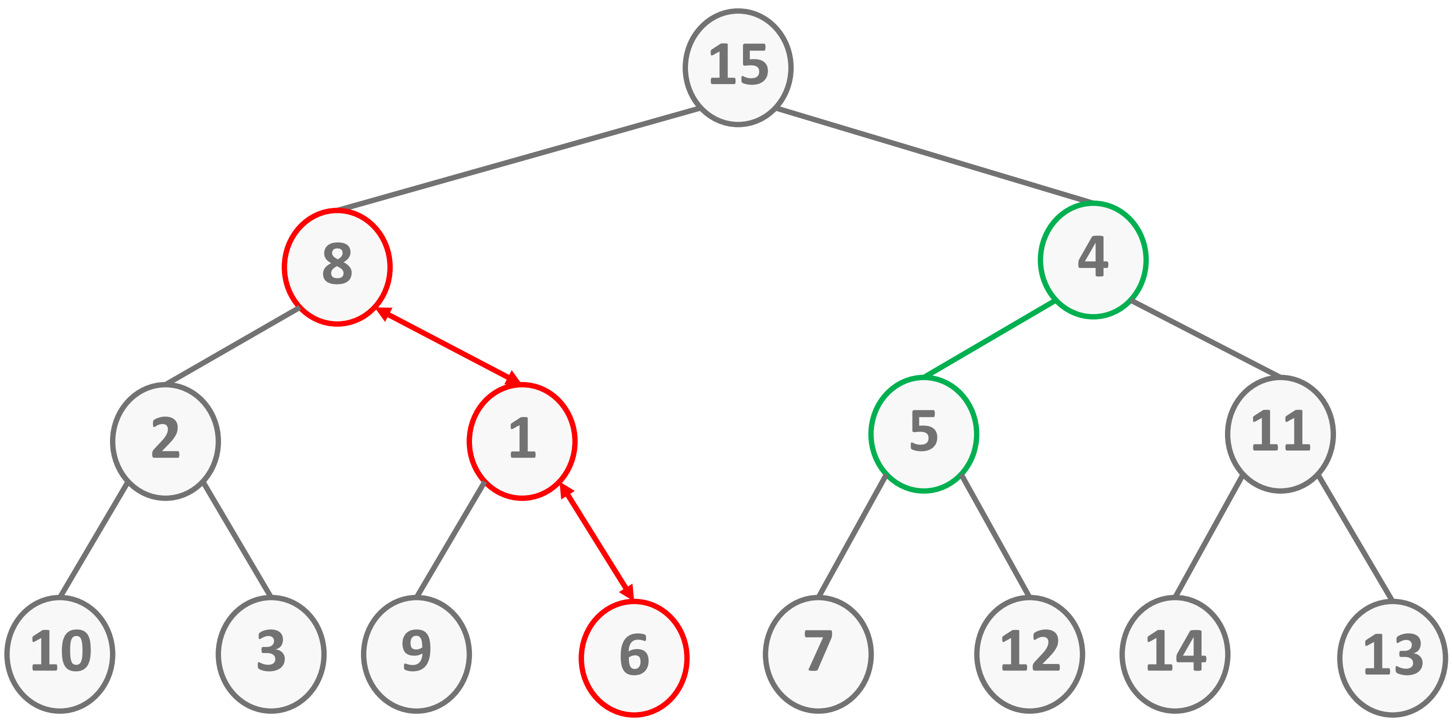
	15	8	4	3	1	7	11	10	2	9	6	5	12	14	13
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

heapify: fix_down(6,5,4)



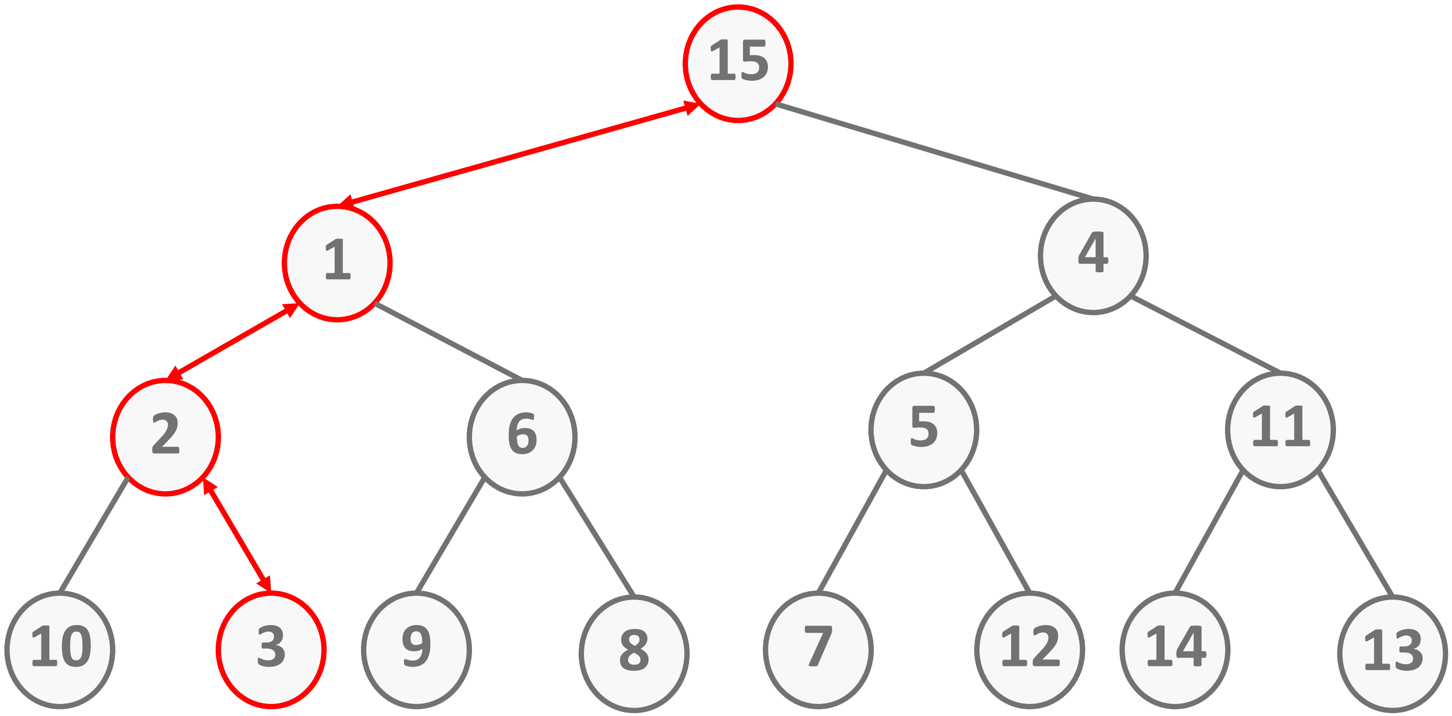
	15	8	4	3	1	7	11	10	2	9	6	5	12	14	13
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

heapify: fix_down(3,2)



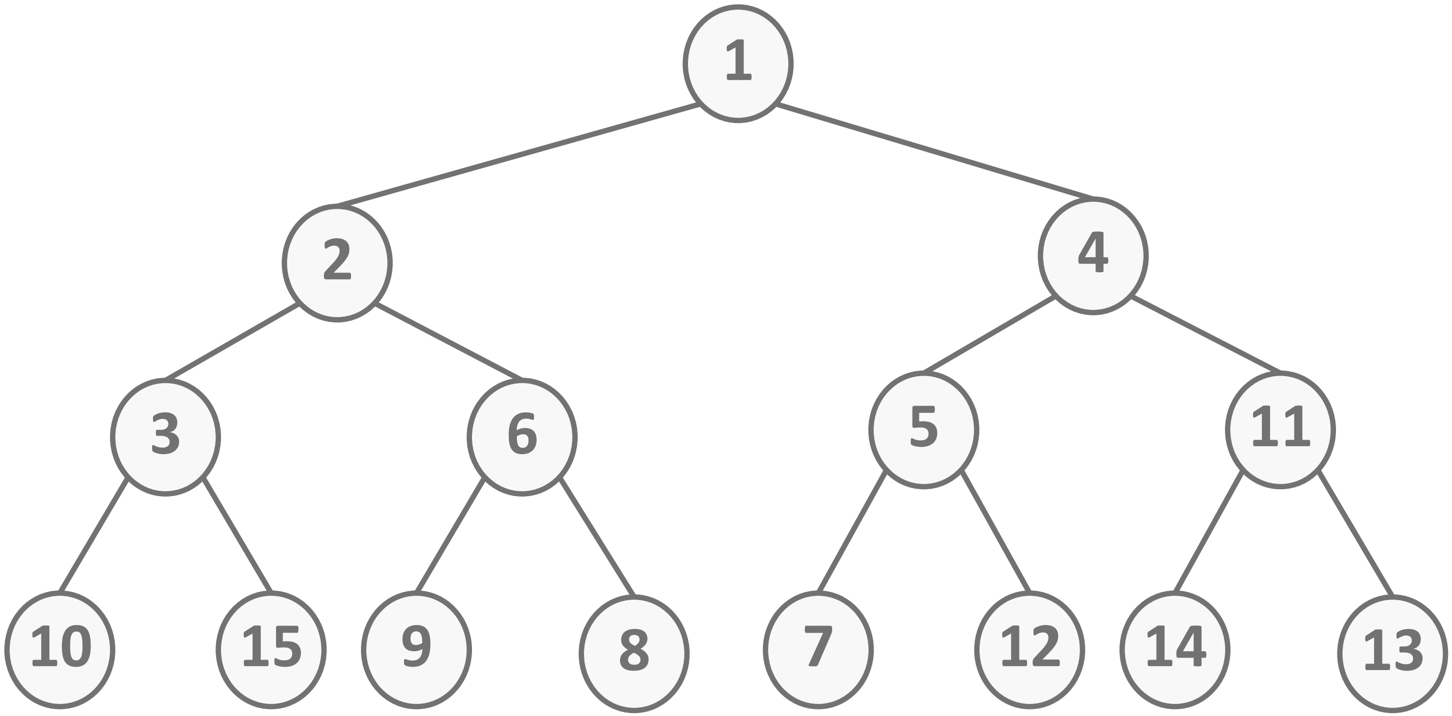
	15	**8**	4	2	**1**	5	11	10	3	9	**6**	7	12	14	13
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

heapify: fix_down(1)



	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

heapify: 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 ***heapify***

- To bound the running time of ***heapify***, we must bound the number of swaps during the construction.
  - Every node at height i percolates down at most (H - i) times.
  - Consider the **worse** case: the heap is a complete binary tree with (2^{H+1} - 1) nodes.
- We can show the sum is ***O*(*N*)**.
  - ***heapify*** is more efficient than our intuitive idea by a factor of **lg *N***.

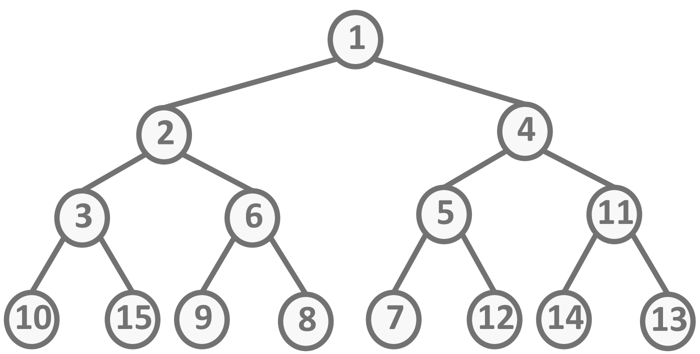
Complexity of *heapify* (Cont')

There are 2^i nodes at level *i*
Suppose all node **percolates** down to the leaves, then

$$S = \sum_{i=0}^{H} 2^i (H - i) \qquad 2S = \sum_{i=0}^{H} 2^{i+1}(H - i) = \sum_{i=1}^{H} 2^i (H - i + 1)$$

$$S = 2S - S = \sum_{i=1}^{H} 2^i - H = (2^{H+1} - 1) - 1 - H$$

$$\begin{aligned} S &= (2^{H+1} - 1) - H - 1 \\ &= N - \lfloor \lg N \rfloor - 1 \\ &= O(N) \end{aligned}$$



Priority Queue

- In some applications, a simple queue may not be the best strategy to complete jobs.
  - Printer queue
  - Multiprocessing queue
- Sometimes it seems that small jobs take longer
- Important jobs can't be done first

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.

Priority Queue (PQ) ADT

- In practice, priority queues are more **complex** than the simple definition.
- There are several other operations 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 arbitrary specified item
  - **Delete** an arbitrary specified item
  - **Join** two priority queues into one large one.

PQ ADT: Implementations

- Several possible implementations are possible:
  - Simple linked list
  - A sorted contiguous list
  - **An unsorted array/list**
  - **Binary heap**
- What will be the **complexity** of ***insert*** and ***delmax*** if the above data structures are used?

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	1	1	N	N
unordered array	1	N	1	N	1	N
unordered list	1	N	1	N	1	1
(binary) heap	$\lg N$	$\lg N$	$\lg N$	1	$\lg N$	N
binomial queue	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$
best in theory	1	$\lg N$	$\lg N$	1	1	1

Let's see how to implement **max-heap** using unordered array and binary heap.

PQ ADT: Unsorted List Implm.

The implementation is straight-forward.

```
typedef struct {
    int k;
    char *s;
} e_t;

typedef struct {
    int n;
    e_t *a;
} pq_t;
```

```
void pq_insert(pq_t *pq, e_t e){
    pq->a[pq->n++] = e;
}

e_t pq_delmax(pq_t *pq){
    int i, m = 0;
    e_t max_e = {INT_MAX, NULL};
    if (pq->n == 0) return max_e;

    for (i = 1; i < pq->n; i++)
        if (pq->a[m].k < pq->a[i].k)
            m = i;
    max_e = pq->a[m];
    pq->n--;
    for (i = m; i < pq->n; i++)
        pq->a[i] = pq->a[i + 1];

    return max_e;
}
```

find the max.

remove and shift down

PQ ADT: Binary Heap Implm.

Note the extra work to maintain the heap structure.

```
void pq_insert(pq_t *pq, e_t e){
    pq->a[++pq->n] = e;
    fix_up(pq->a, pq->n);
}

e_t pq_delmax(pq_t *pq){
    swap(&pq->a[1], &pq->a[pq->n]);
    fix_down(pq->a, pq->n - 1, 1);
    return pq->a[pq->n--];
}
```

insert: extra fix_up in ***O*(lg *N*)** time.

delmax: reduced workload.
fix_down in ***O*(lg *N*)** time.

Be careful with the extra initialization that makes your life easier.

```
pq_t *pq_create(){
    pq_t *pq = malloc(sizeof *pq);
    pq->a = malloc((MAX_SIZE + 1) * sizeof(pq_t));
    pq->a[0].k = INT_MAX;
    pq->n = 0;
    return pq;
}
```

Heapsort

- Heaps can be used to sort in $O(N \lg N)$ time.
- The basic strategy is to
  - build a binary heap of N elements in $O(N)$ time
  - perform N ***delmin / delmax***.
- We record the **minimum** elements that leaves in a second array and copy the array back to complete the sorting.
- Total running time is $O(N) + N \times O(\lg N) = \boldsymbol{O(N \lg N)}$.

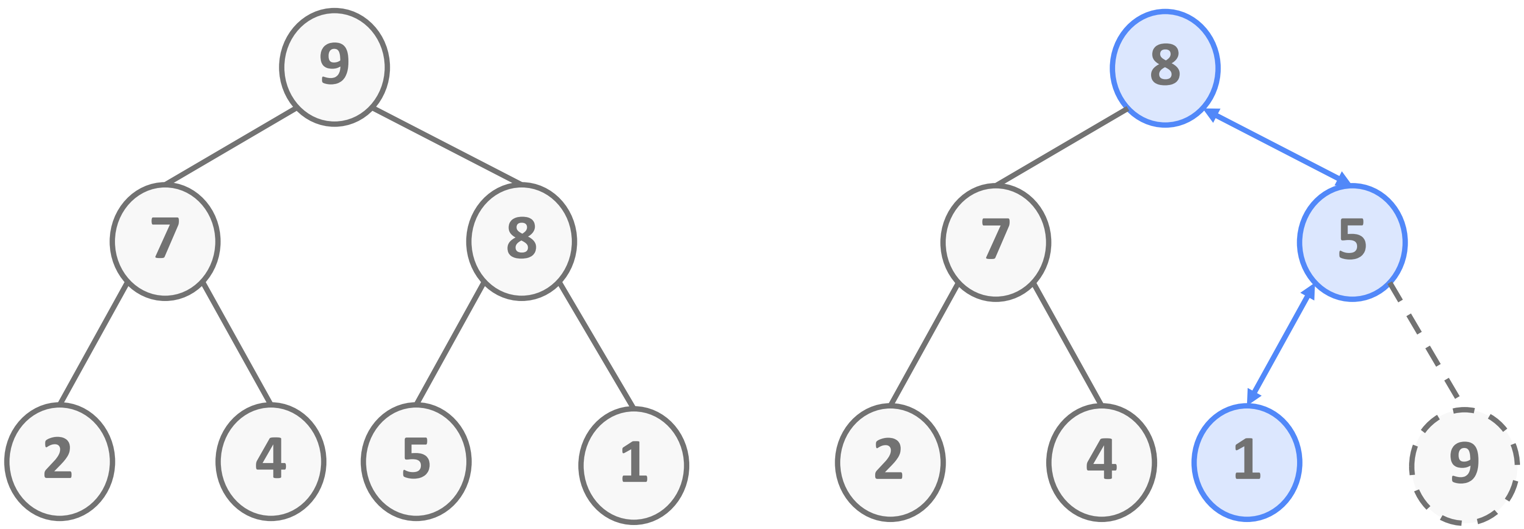
```
void heapsort(int n, int a[]){
    pq_t *pq = pq_create();
    for (i = 0; i < n; i++)
        pq_insert(pq, a[i]);
    for (i = 0; i < n; i++)
        a[i] = pq_delmin(pq);
}
```

Tricks in Implementing Heapsort

- The memory requirement is doubled since we need an extra array stored inside the priority queue.
  - **Overhead**: $O(N)$ time to copy data back to the original array
  - The **trick** to avoid the second array is to make use of the last cell in the array to store the value returned by ***delmin***.
  - Using this strategy the array will contain the elements in decreasing sorted order after the last ***delmin***.
- Suppose we want to sort in **increasing** order, we can use a maxheap with a ***delmax*** operation.
  - The maximum element will move towards the end of the array.

Heapsort: Example

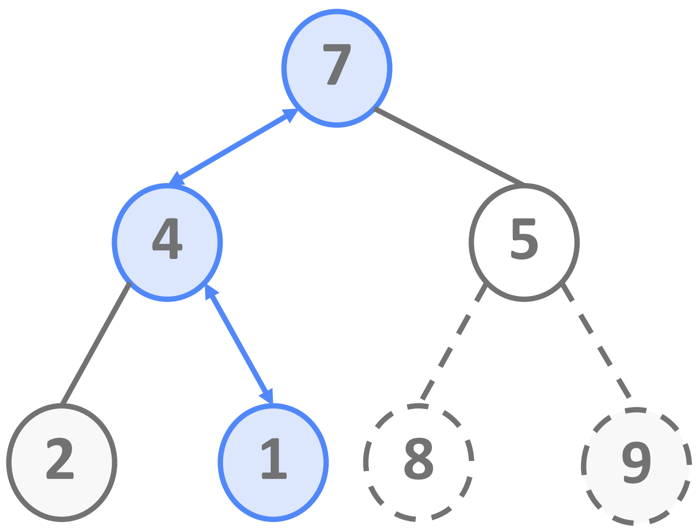
- Maxheap with its array representation. Execute ***delmax***.



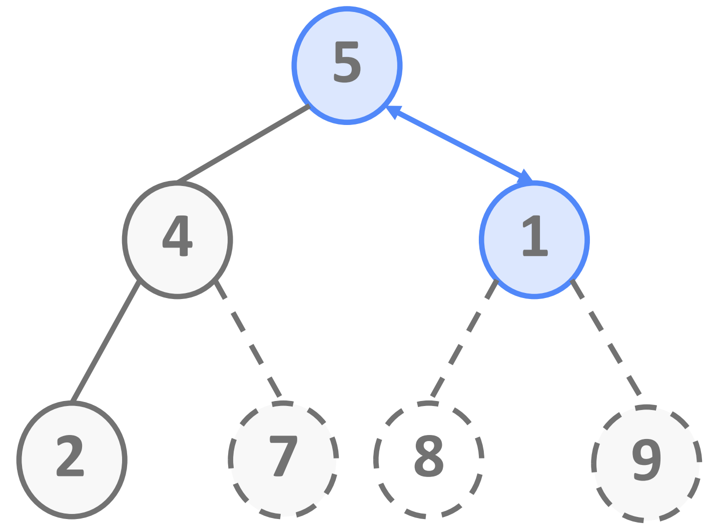
-	9	7	8	2	4	5	1

-	**8**	7	**5**	2	4	**1**	**9**

Heapsort: Example (Cont')



-	7	4	5	2	1	8	9



-	5	4	1	2	7	8	9

Heapsort: Code

- Remember that we start the first element at index 1 in the binary heap.
  - Sometimes you might want to avoid this by adding index checking in fix_down()

```
void heapsort(int n, e_t *a){
    int i;
    for (i = n / 2; i >= 1; i--)
        fix_down(a, n, i);
    while (n > 1){
        swap(&a[1], &a[n]);
        fix_down(a, --n, 1);
    }
}
```

Heapsort: Analysis

- ***heapify***: at most 2*N* comparisons.
- ***delmax***: the *i*-th operation uses at most $2\lfloor \lg i \rfloor$ comparisons.

Total no. of comparisons

$$
\begin{aligned}
&= 2N + \sum_{i=1}^{N-1} 2\lfloor \lg i \rfloor \\
&< 2N - 2\lg N + 2\sum_{i=1}^{N} \lg i \\
&= 2N - 2\lg N + 2\lg(N!) \\
&\approx 2N - 2\lg N + 2(N \lg N - 1.44N) \\
&= O(N \lg N)
\end{aligned}
$$

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. $K \times O(\lg N)$.
  - The total running time is $O(N + K \lg N)$.
- 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 $O(K \lg N)$ time.

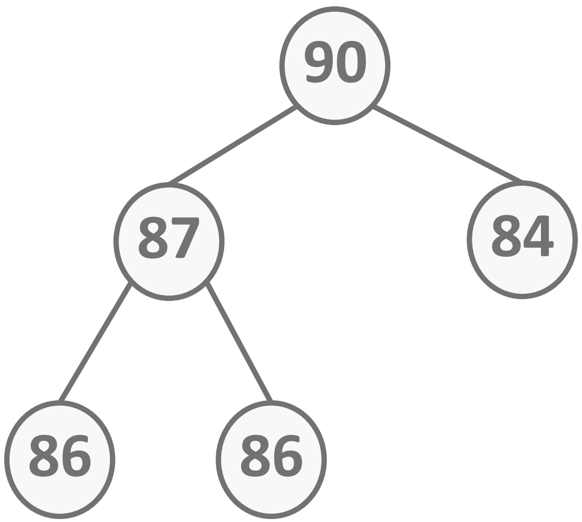
The *K*-selection Problem (2)

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

Priority Queue for Index Items

- Suppose that the records to be processed in a PQ are in an **existing** array.
  - Then the PQ can refer to the items through **array index**.
- The array index can also be used to control PQ routines like ***change*** & ***delete***.
- Technically, the PQ ADT cannot move the **external** actual data, but have to keep track of the positions of the indices(keys) internally.
- Another array is allocated to allow **fast lookup** of the positions of the keys in the PQ array.

Index Heap Data Structures



-	3	2	4	9	1		

Such implementation is sometimes called the index heap, which is usually in many graph algorithms.

k	qp[k]	pq[k]	data[k]
0			Wilson / 63
1	5	3	Johnson / 86
2	2	2	Jones / 87
3	1	4	Smith / 90
4	3	9	Washington / 84
5		1	Thompson / 65
6			Brown / 82
7			Jackson / 61
8			White / 76
9	4		Adams / 86
10			Black / 71

Index Heap: Implm.

```
static int n, pq[MAX + 1], qp[MAX + 1];
void exch(int i, int j){
    int t;
    t = qp[i]; qp[i] = qp[j]; qp[j] = t;
    pq[qp[i]] = i; pq[qp[j]] = j;
}

void pq_insert(int k){
    qp[k]= ++N;
    pq[N] = k;
    fix_up(pq, n);
}

int pq_delmax(){
    exch(pq[1], pq[n]);
    fix_down(pq, --n, 1);
    return pq[n + 1];
}

void pq_change(int k){
    fix_up(pq, qp[k]);
    fix_down(pq, n, qp[k]);
}
```

Summary

- Binary heap: structure & order properties
  - **Efficient** array implementation
  - ***findmax*/*findmin*** in constant time
  - fixing heap properties in $O(\lg N)$ time.
  - $O(N)$ ***heapify*** construction
- **Priority Queue ADT** - various implementations
- **Heapsort**
- Application: K-selection problem
  - reveal theory bound of $O(N \lg N)$ in finding the **median** of a set of N numbers.