Binary Heap

Joy Mukherjee

Assistant Professor
Computer Science & Engineering
IIT Bhubaneswar

Applications

- Job scheduling in operating systems (priority queue)
- Efficient sorting of elements (heap sort)
- IMDB: Finding out top-rated movies in a particular category

- The definition of priority is varied across applications, where top priority may either indicate the smallest or the largest.

Binary Heap

A Binary Heap is a Binary Tree with following properties.

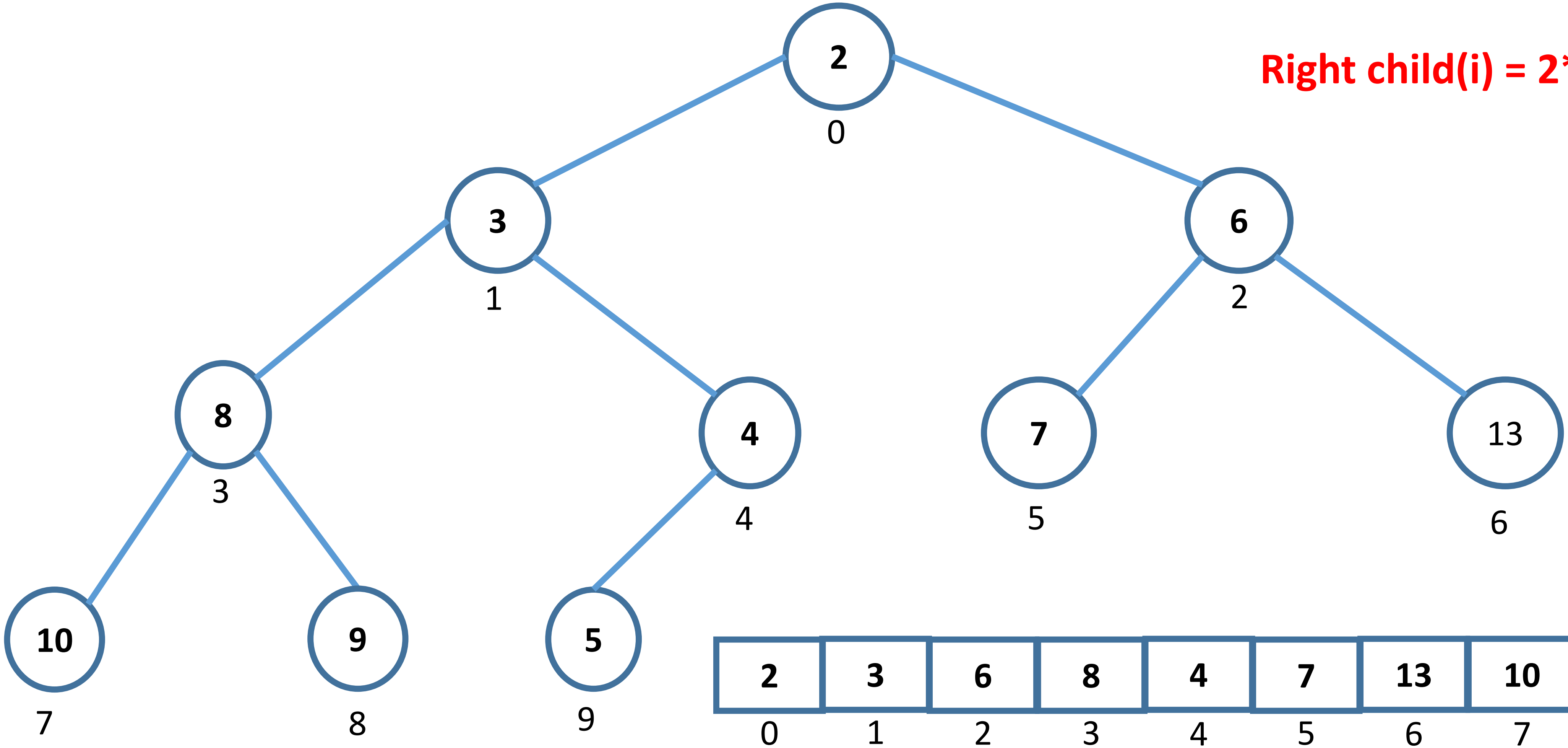
1. It is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes it suitable to be stored in an array.
2. A Binary Heap is either a Min Binary Heap or a Max Binary Heap.
   I. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for its left and right subtree.
   II. In a Max Binary Heap, the key at root must be maximum among all keys present in Binary Heap. The same property must be recursively true for its left and right subtree.

Min Binary Heap

Parent(i) = (i-1)/2

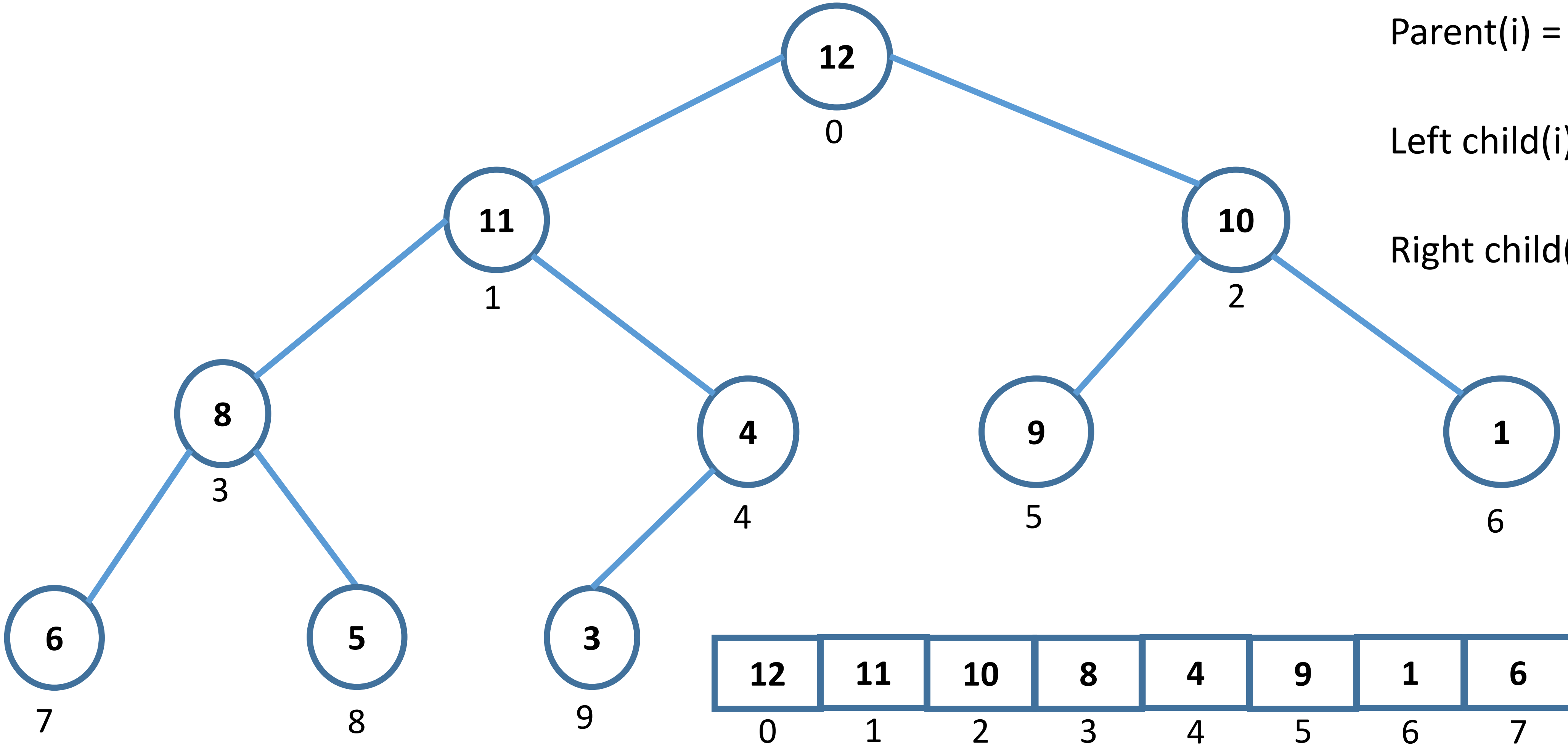
Left child(i) = 2*i + 1

Right child(i) = 2*i + 2



2	3	6	8	4	7	13	10	9	5
0	1	2	3	4	5	6	7	8	9

Max Binary Heap



Parent(i) = (i-1)/2

Left child(i) = 2*i + 1

Right child(i) = 2*i + 2

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

Binary Heap Data Structure

- Discussion: maximum binary heap
- An array A which is used to store the elements in the heap
- An integer n which gives the number of elements currently in the heap
- An integer maxsize which gives the maximum size of the heap
- n <= maxsize

Move Down

- Suppose you update a key value at index i in the heap A
- Assumption: Left subtree and right subtree if A[i] maintain heap property.
- Objective: How to restore the heap property at A[i]

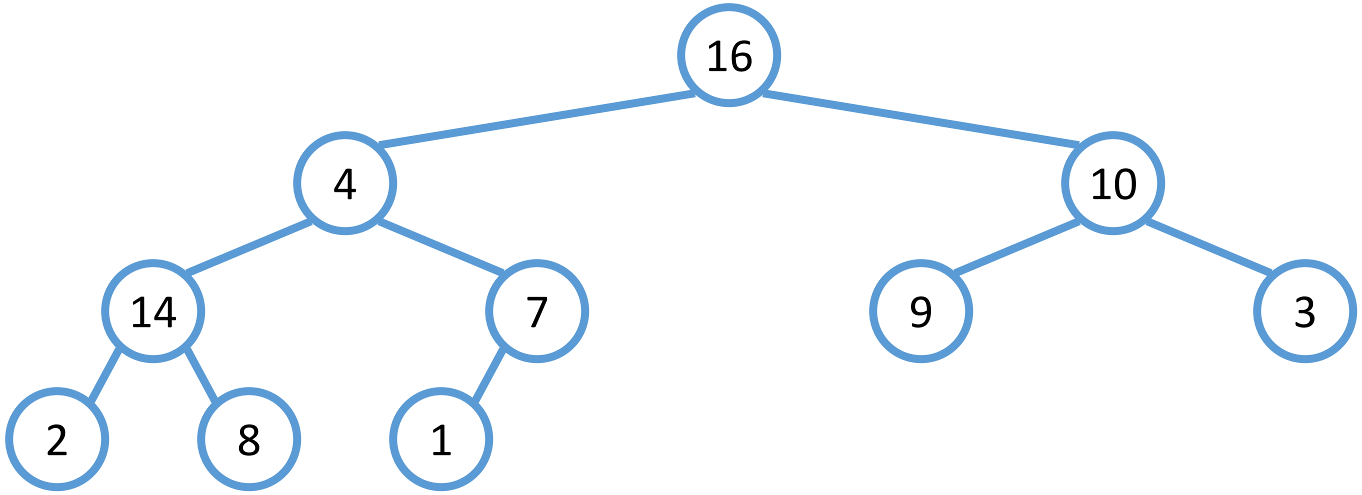
Move Down

```
void movedown(int a[], int n, int i)
{
        int max = i;
        int l = 2*i+1;
        int r = 2*i+2;
        if(l < n && a[l] > a[max])
                max = l;
        if(r < n && a[r] > a[max])
                max = r;
        if(max != i) {
                swap(a, i, max);
                movedown(a, n, max);
        }
}
```

Movedown() operation applies only if at index i the heap property is violated. However, the left and right subtree of i must be proper heaps.

In such scenario, movedown() operation restores the heap property at index i.

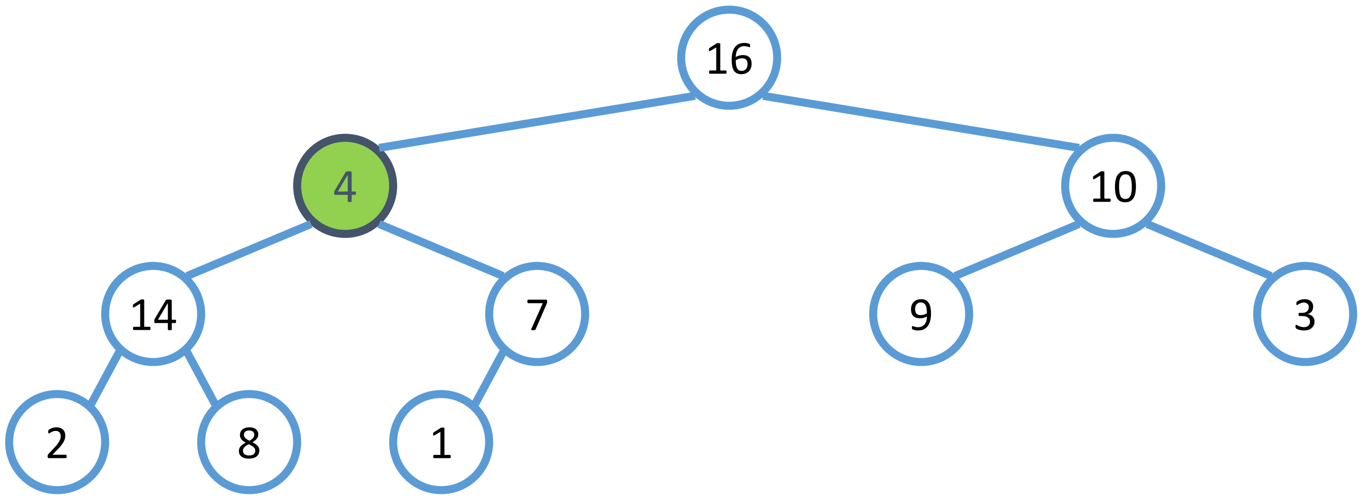
Move Down(A, 10, 1)



A =

16	4	10	14	7	9	3	2	8	1

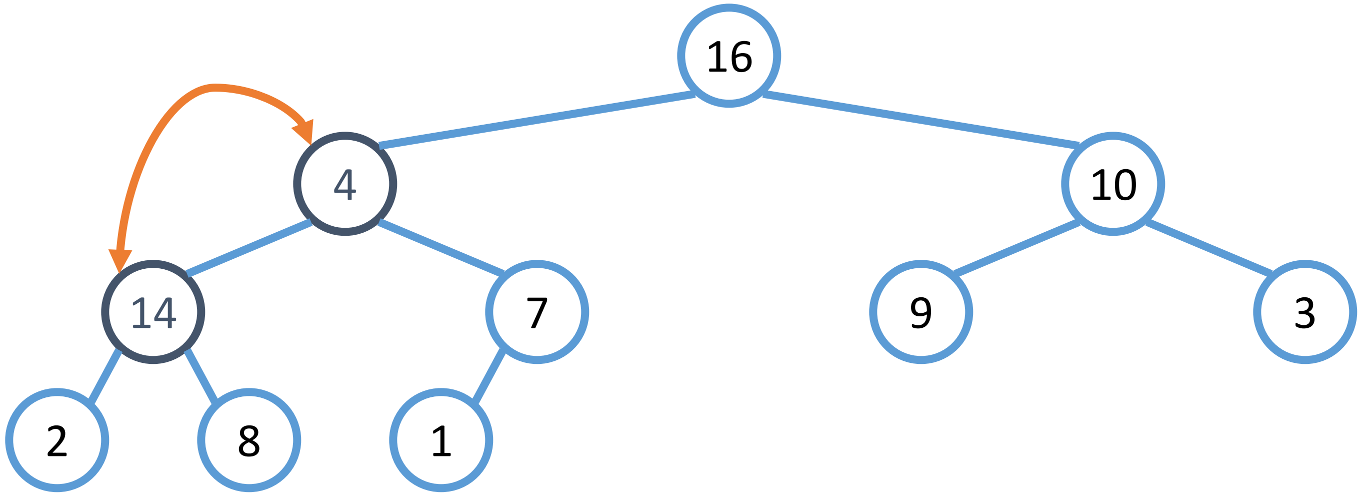
Move Down(A, 10, 1)



A =

16	4	10	14	7	9	3	2	8	1

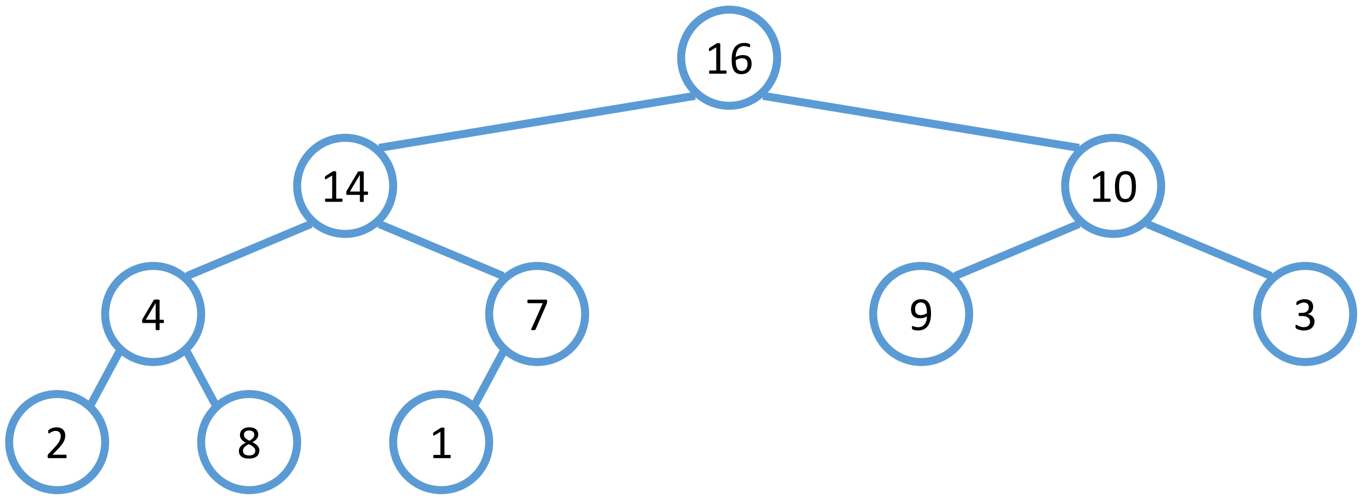
Move Down(A, 10, 1)



A =

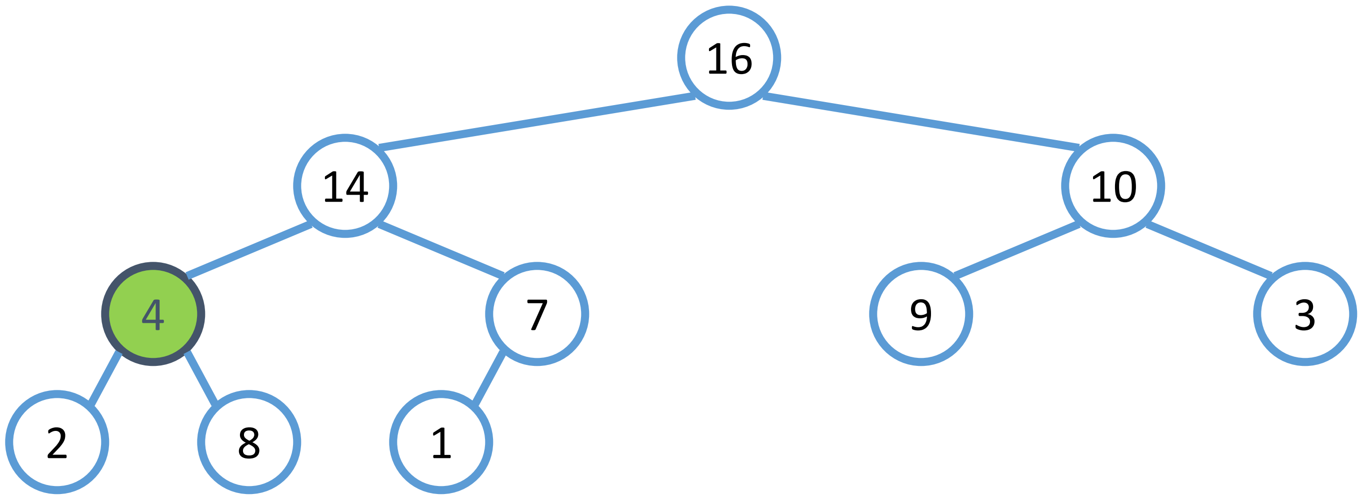
16	4	10	14	7	9	3	2	8	1

Move Down(A, 10, 3)



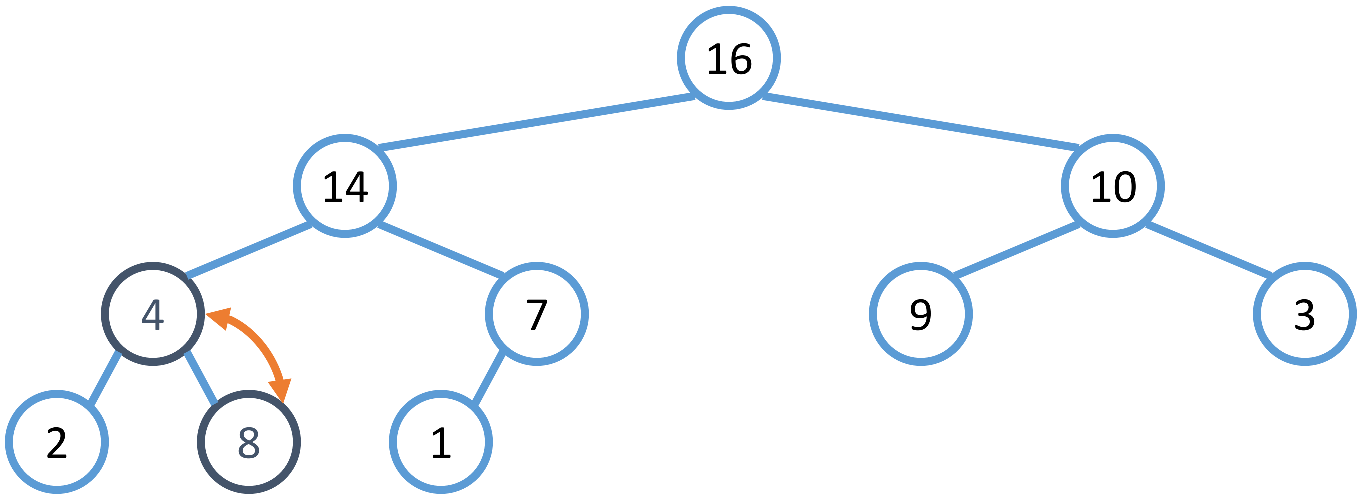
A =	16	14	10	4	7	9	3	2	8	1

Move Down(A, 10, 3)



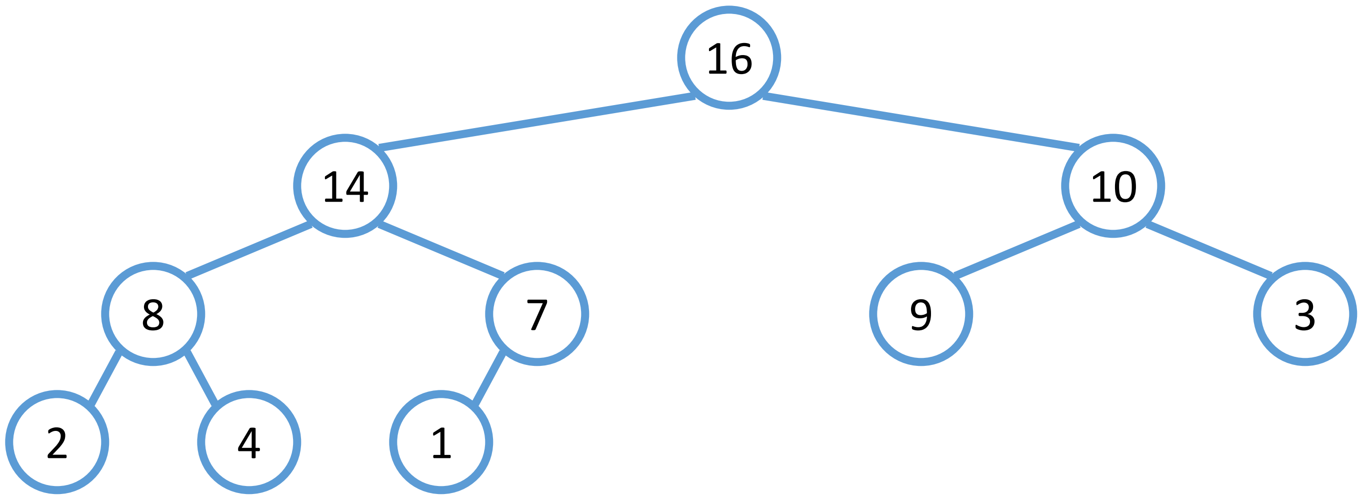
A =	16	14	10	4	7	9	3	2	8	1

Move Down(A, 10, 3)



A =	16	14	10	4	7	9	3	2	8	1

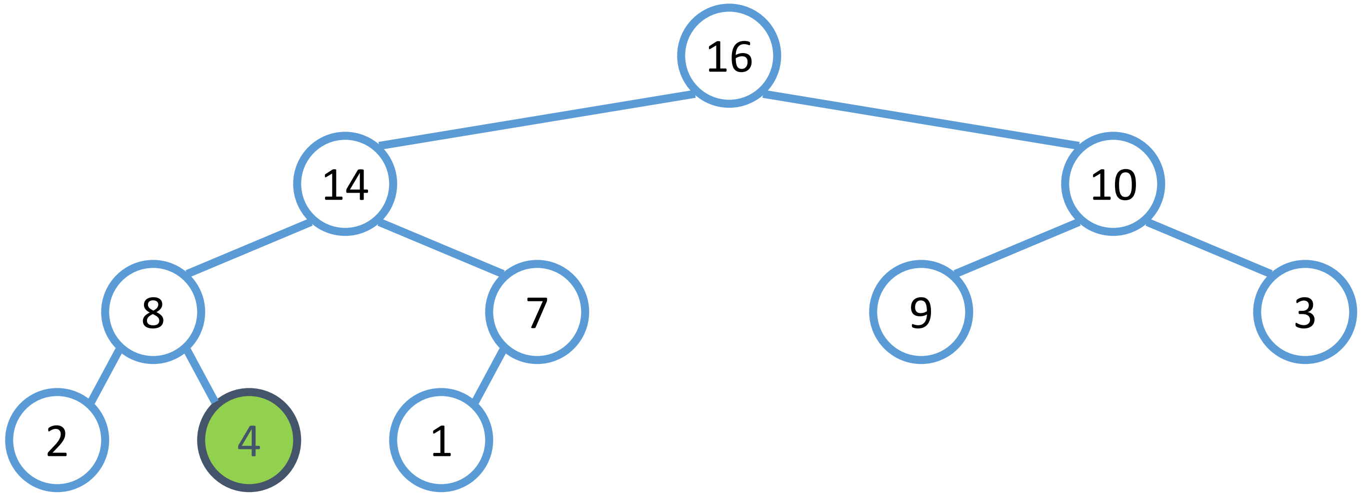
Move Down(A, 10, 8)



A =

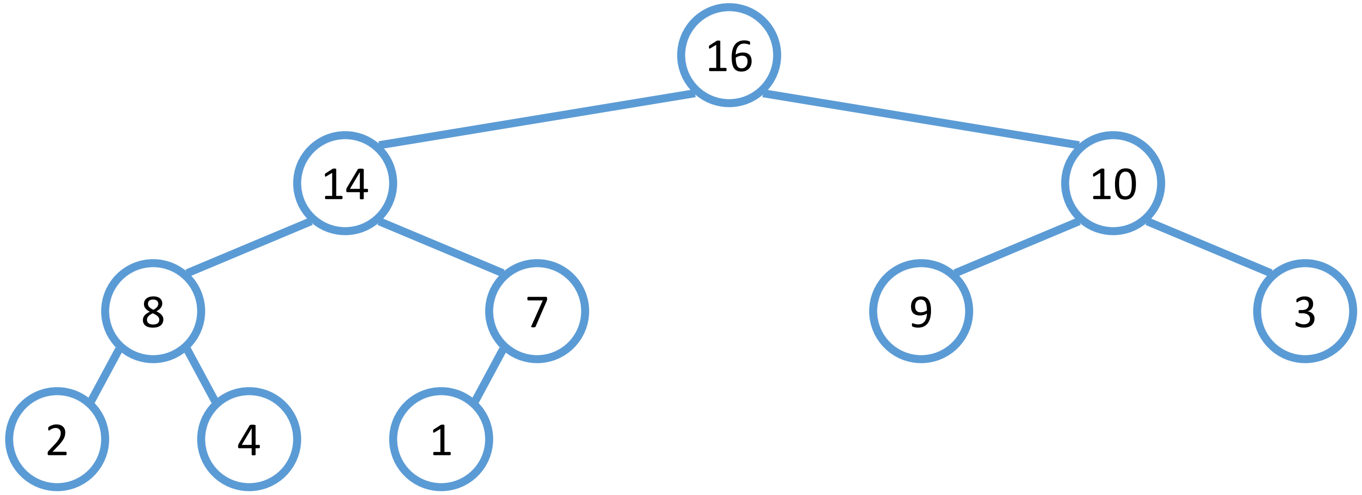
16	14	10	8	7	9	3	2	4	1

Move Down(A, 10, 8)



A =	16	14	10	8	7	9	3	2	4	1

Final Heap



A =

16	14	10	8	7	9	3	2	4	1

Analyzing Move Down(A, n, i)

- Level-0 has 1 element
- Level-1 has 2 elements
- Level-2 has 4 elements
- Level-h has at least 1 element and at most 2^{h} elements
- The binary heap of height h has at least 2^{h} elements and at most $2^{h+1} – 1$ elements
- $2^{h} \le n \le 2^{h+1} – 1$
- $h \le \log_2 n \le h+1$
- **Time Complexity of MoveDown() is $O(h) = O(\log_2 n)$.**

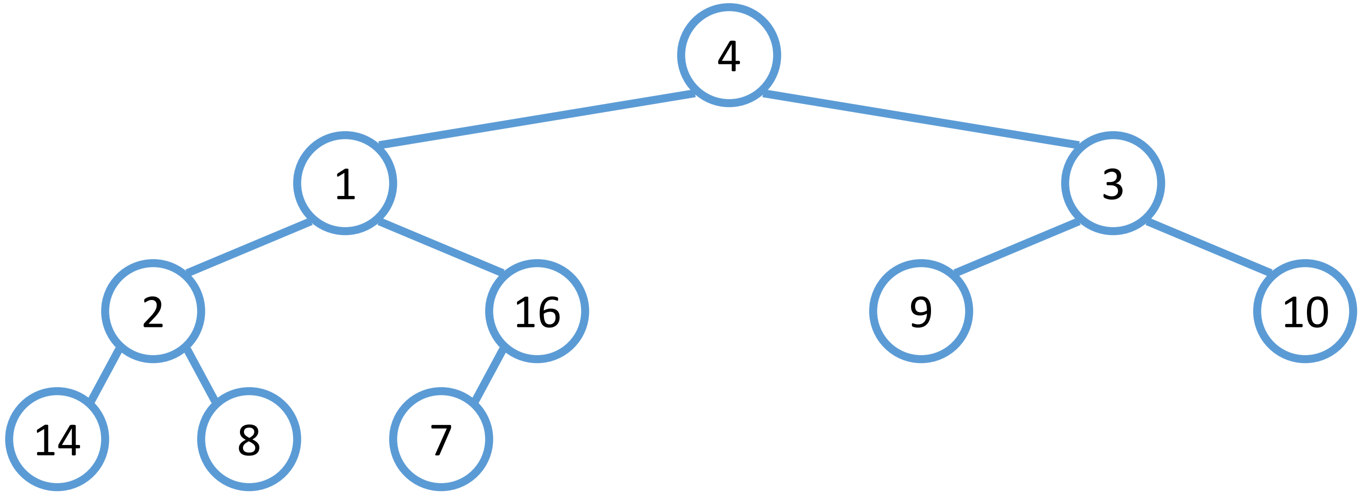
Build Binary Heap

Parent(i) = (i-1)/2

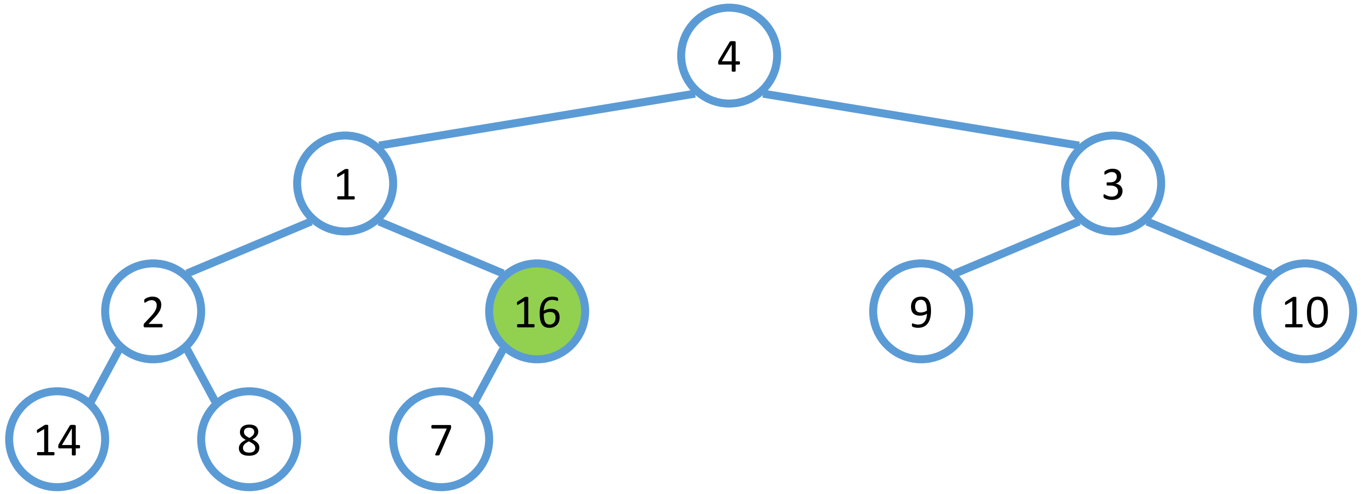
```
void buildBinaryHeap(int a[], int n)
{

        int i, start;
        start = (n-2)/2; // start is the index of the parent of the last leaf a[n-1]
        for(i = start; i >= 0; i--)
                movedown(a, n, i);
}
```

BuildHeap(A, 10)

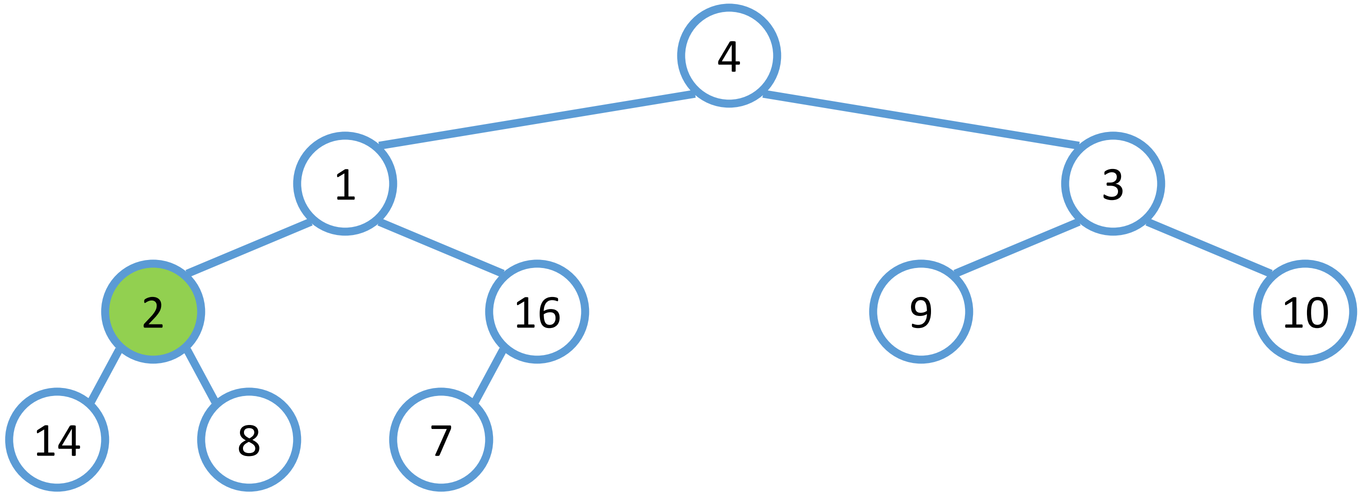
- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



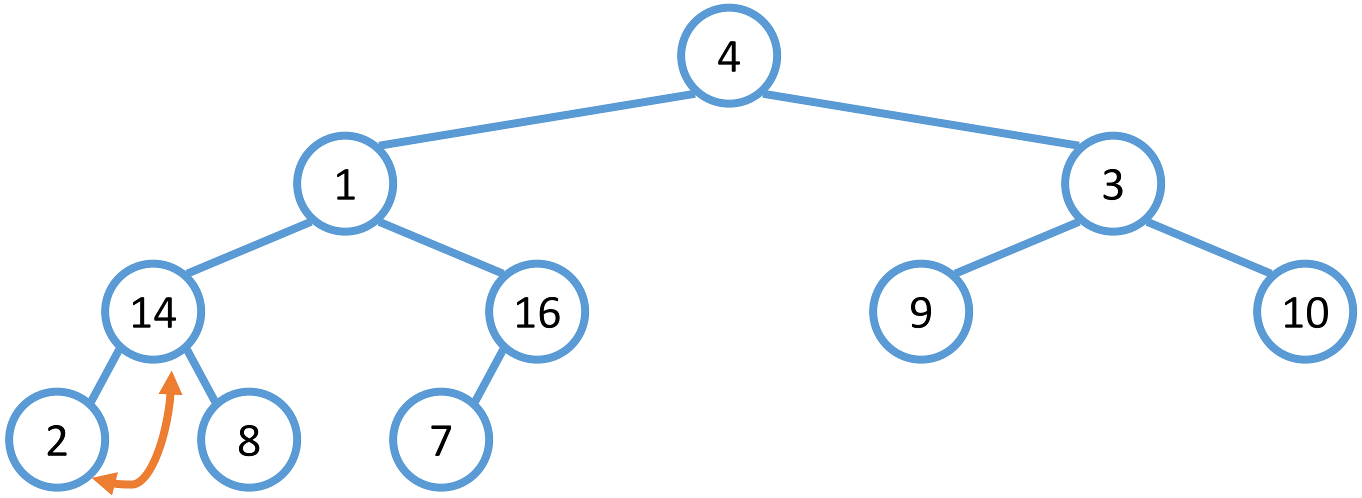
Build Heap(): Move Down(a, 10, 4)



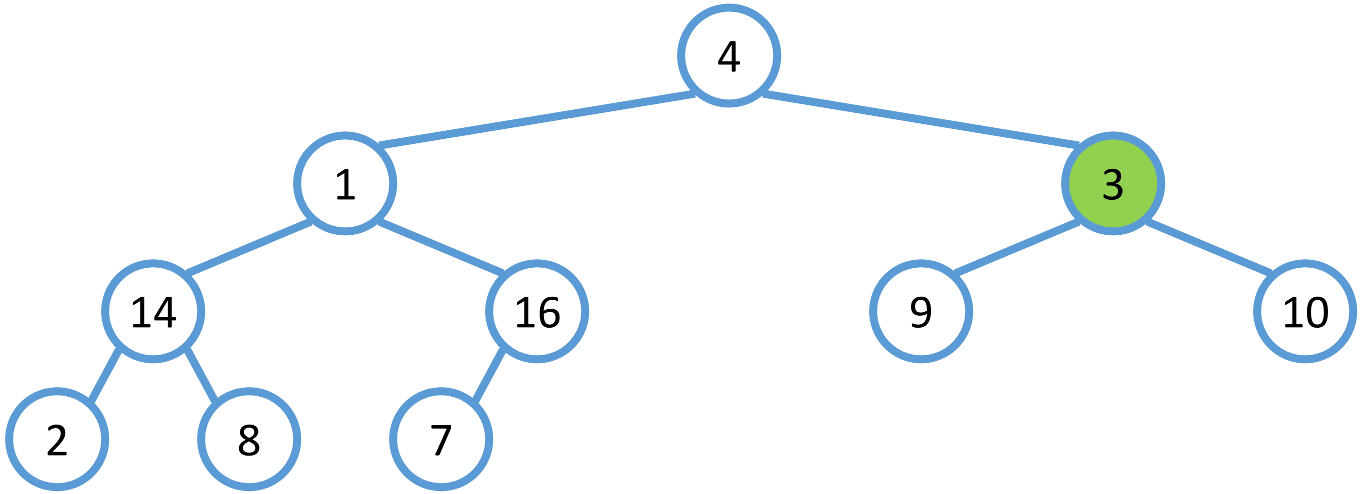
Build Heap(): Move Down(a, 10, 3)



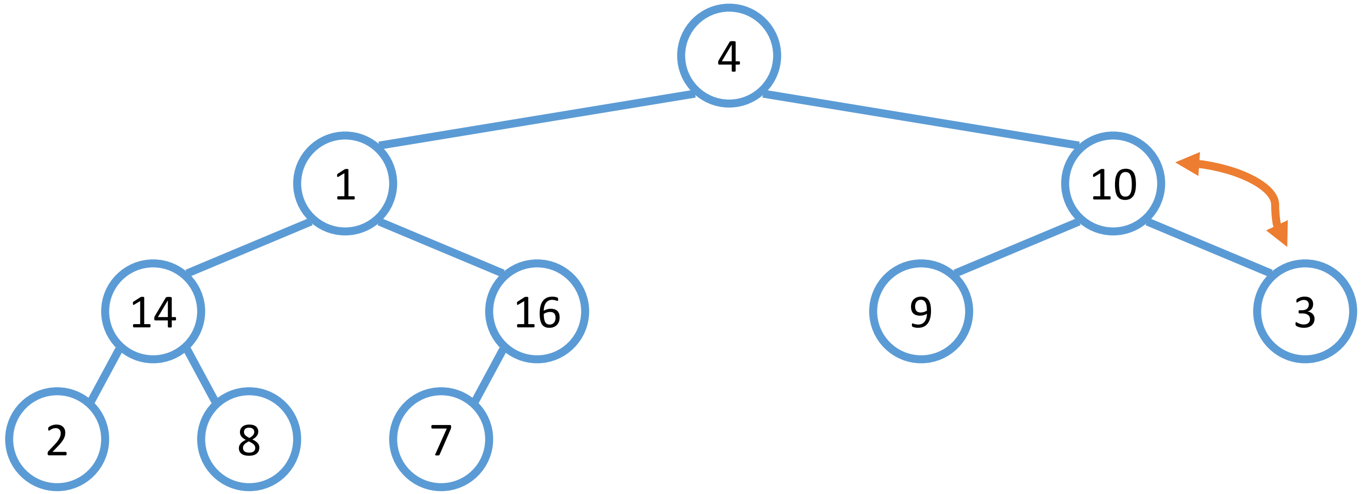
Build Heap(): Move Down(a, 10, 3)



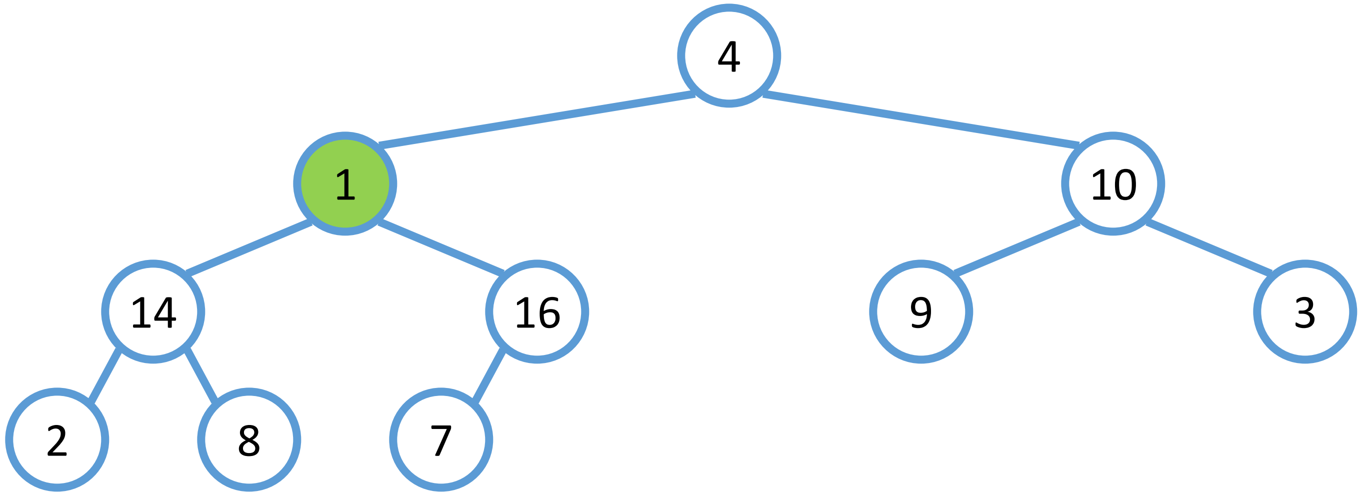
Build Heap(): Move Down(a, 10, 2)



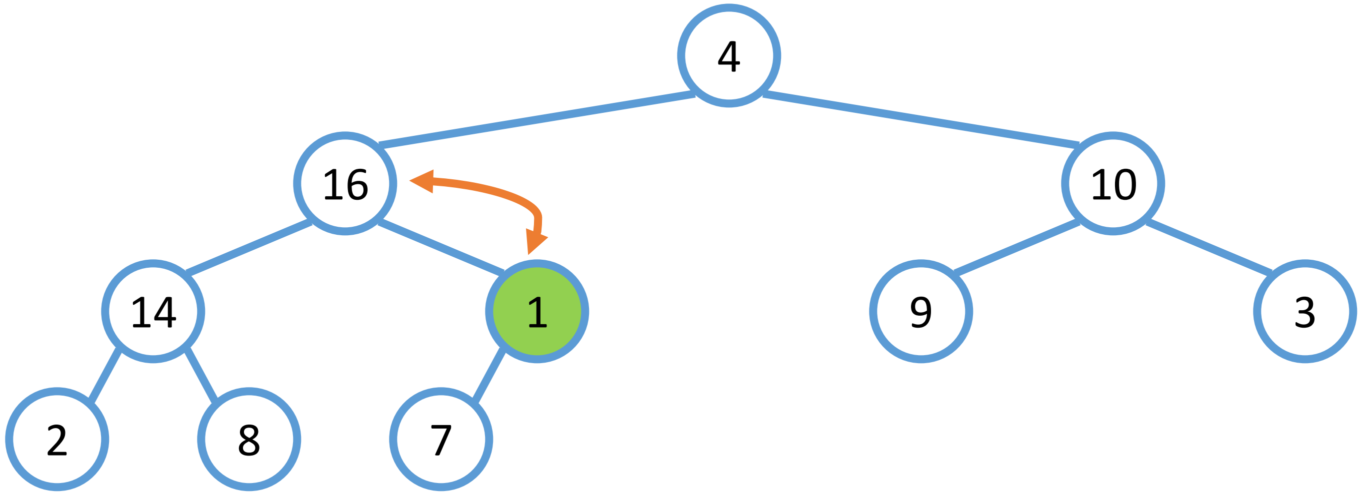
Build Heap(): Move Down(a, 10, 2)



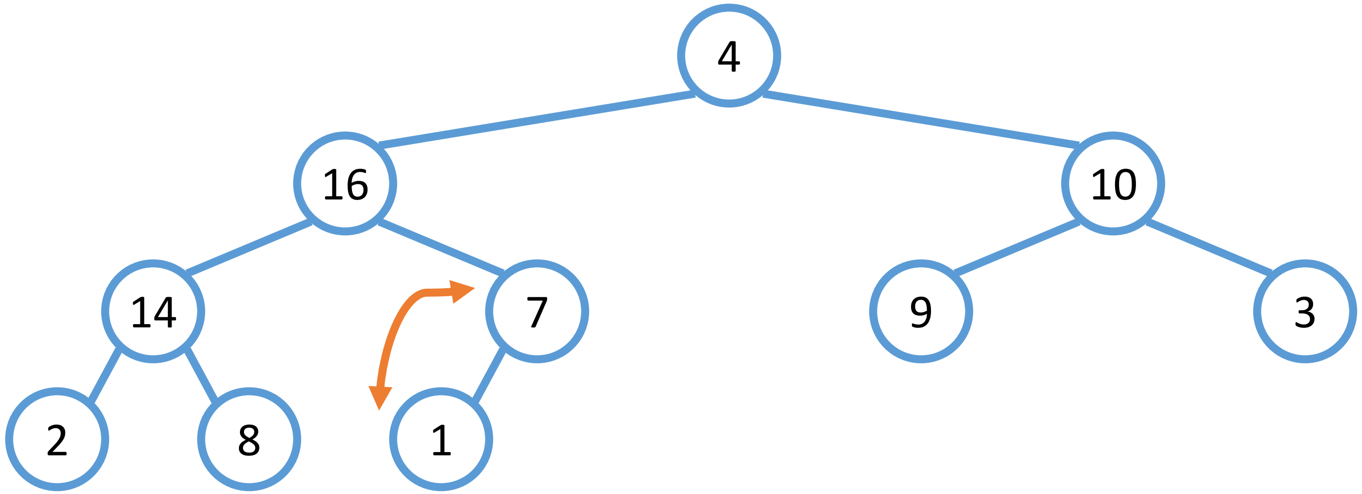
Build Heap(): Move Down(a, 10, 1)



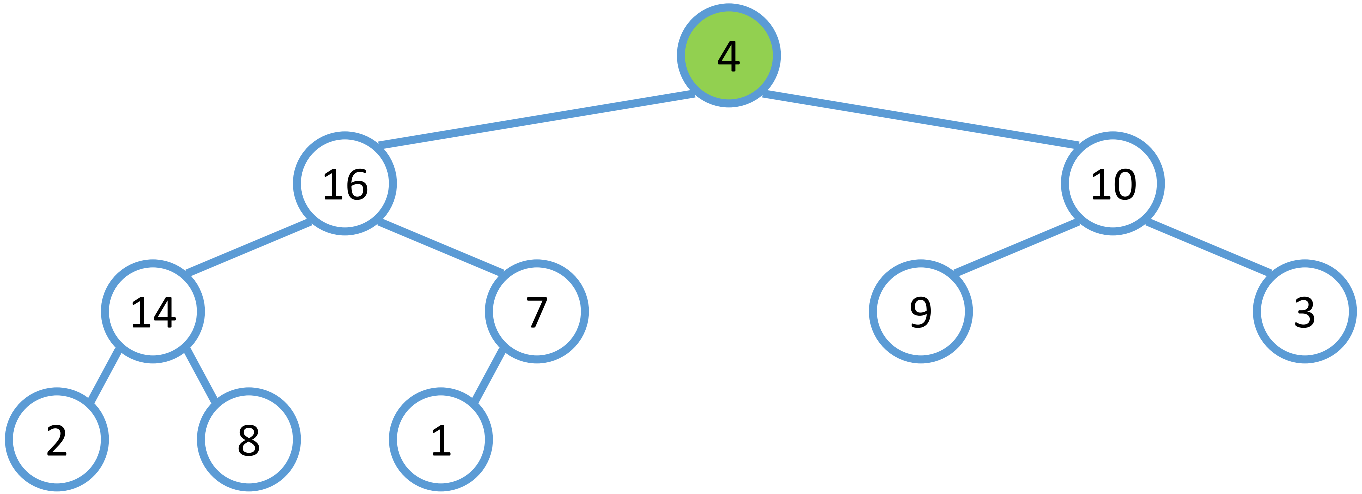
Build Heap(): Move Down(a, 10, 4)



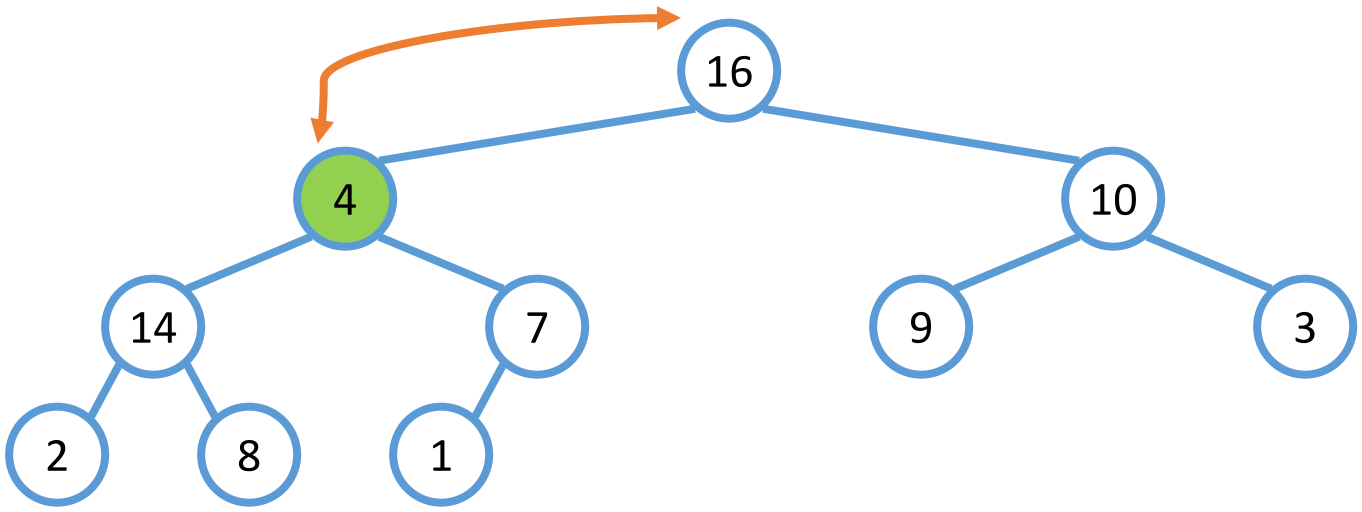
Build Heap(): Move Down(a, 10, 9)



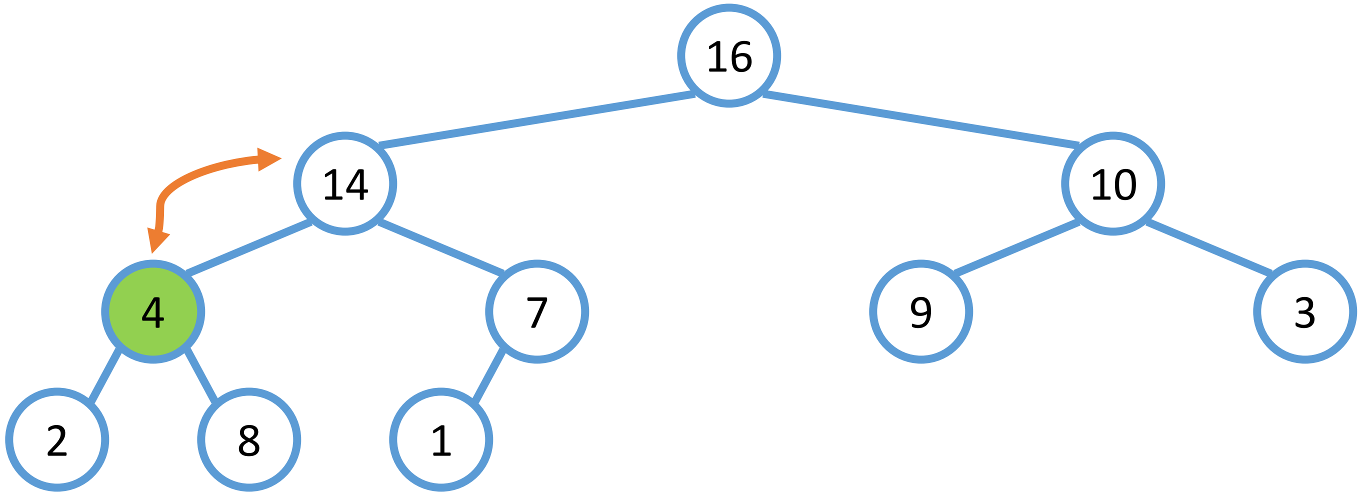
Build Heap(): Move Down(a, 10, 0)



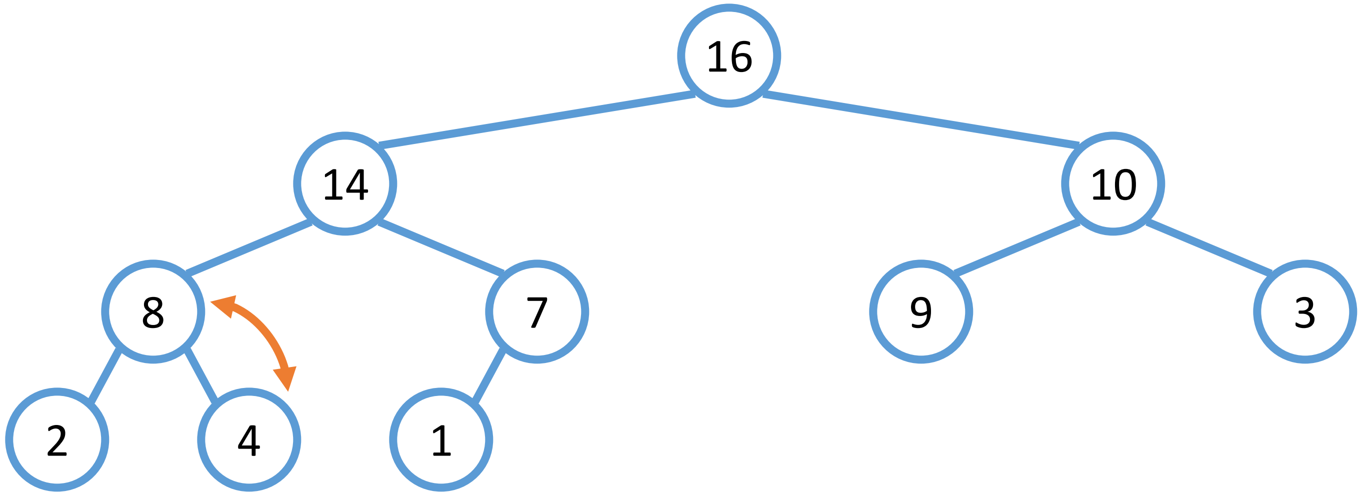
Build Heap(): Move Down(a, 10, 1)



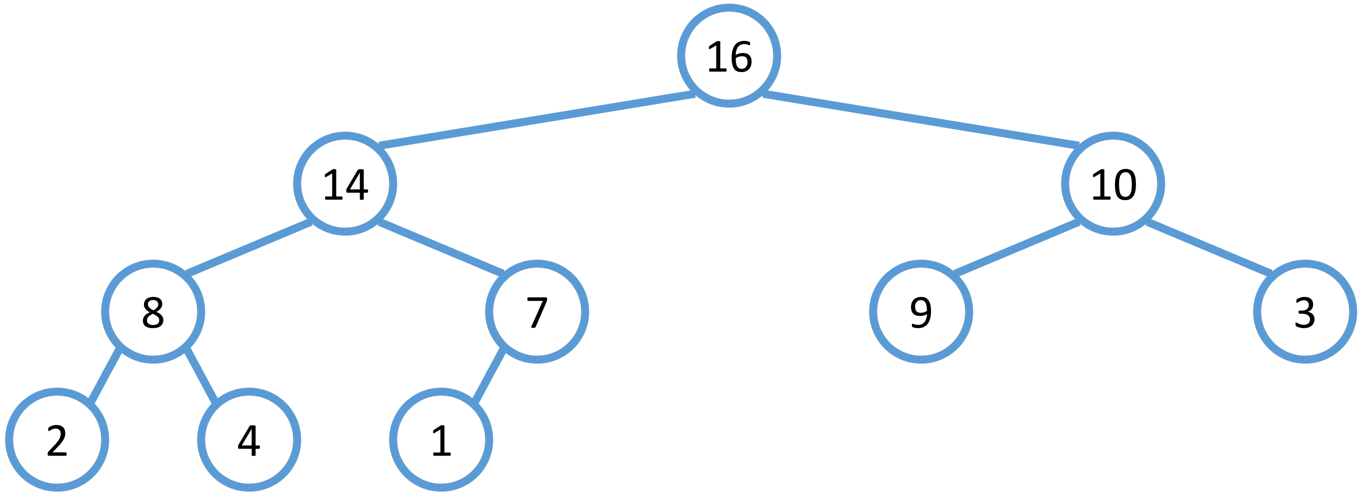
Build Heap(): Move Down(a, 10, 3)



Build Heap(): Move Down(a, 10, 8)



Build Heap(): Move Down(a, 10, 8)



Analyzing Build Heap()

- Each call to **`MoveDown()`** takes O(lg n) time
- There are O(n) such calls (specifically, $\lfloor(n-1)/2\rfloor$)
- Thus the running time is O(n lg n)
  - Is this a correct asymptotic upper bound?
  - Is this an asymptotically tight bound?
- A tighter bound is O(n)
  - How can this be? Is there a flaw in the above reasoning?

Tighter Analysis of BuildHeap()

Level	No of Elements	Time for each MoveDown()
0	1	h
1	2	h-1
2	2^2	h-2
h-1	2^{h-1}	1

Total time

$= h * 1 + (h-1) * 2 + (h-2) * 2^2 + \ldots + (h - (h-1)) * 2^{h-1}$

$= (h + 2h + 2^2h + 2^{h-1}h) - (1*2 + 2*2^2 + \ldots + (h-1) * 2^{h-1})$

$= h(2^h - 1) - S$

Tighter Analysis of BuildHeap()

$S = 1*2 + 2*2^2 + \ldots + (h\text{-}1) * 2^{h-1}$

$2S = \quad\quad 1*2^2 + \ldots + (h\text{-}2) * 2^{h-1} + (h\text{-}1) * 2^h$

$-S = 2 + 2^2 + \ldots + 2^{h-1} - (h\text{-}1) * 2^h$

$-S = 2(2^{h-1} - 1) - (h\text{-}1) * 2^h$

Total time

$= h(2^h - 1) - S$

$= h(2^h - 1) + 2(2^{h-1} - 1) - (h\text{-}1) * 2^h$

$= h2^h - h + 2^h - 2 - h\,2^h + 2^h$

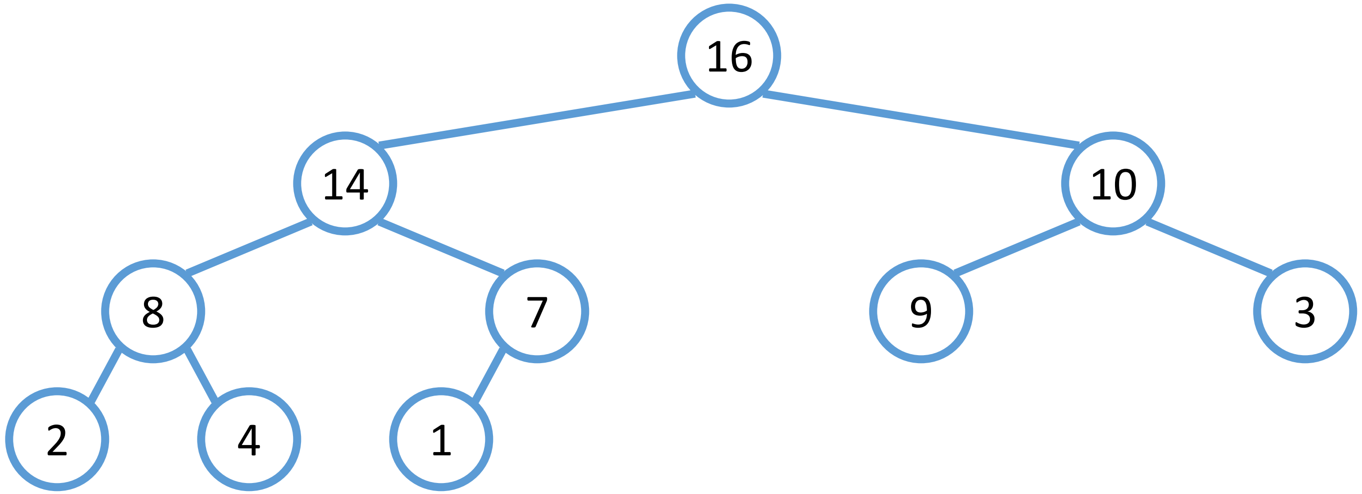
$= 2^{h+1} - h - 2 = 2^{h+1} - 1 - h - 1 \leq 2^{h+1} - 1 = O(n)$

Heap Sort

```
void heapsort(int a[], int n)
{
        int i;
        buildBinaryHeap(a, n);
        for(i = n-1; i >= 1; i--) {
                swap(a, 0, i);
                n--;
                movedown(a, n, 0);
        }
}
```

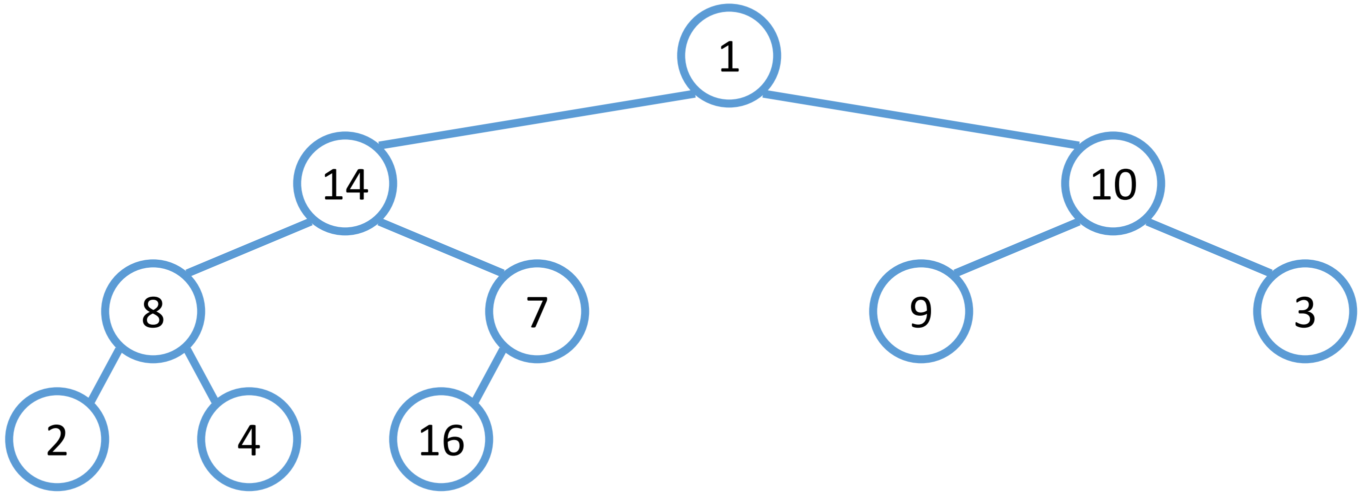
Heap Sort

N = 10



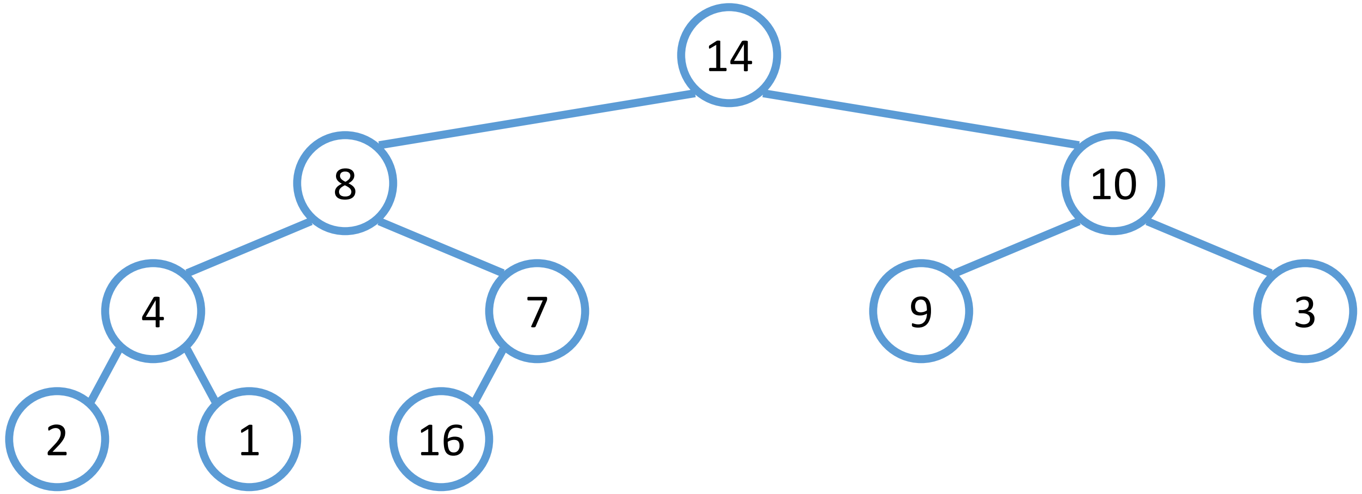
Heap Sort

N = 9



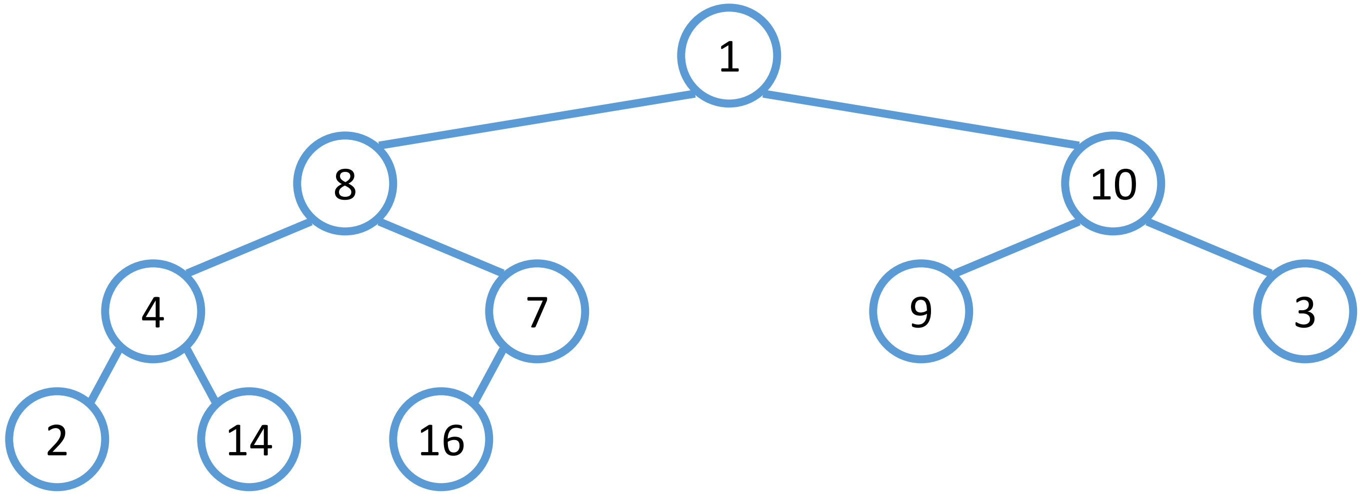
Heap Sort

N = 9



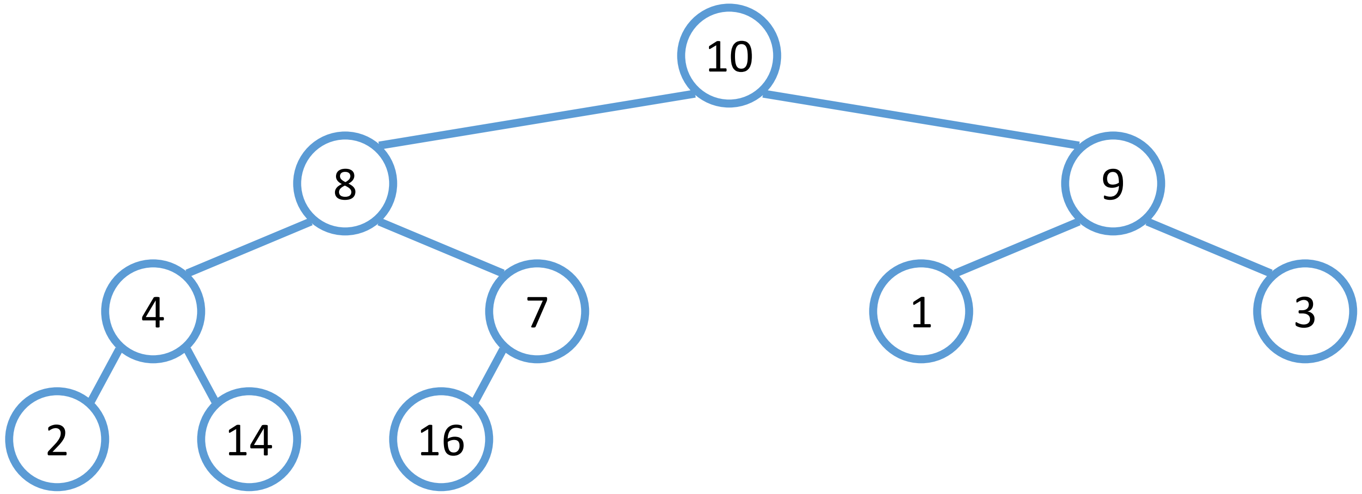
Heap Sort

N = 8



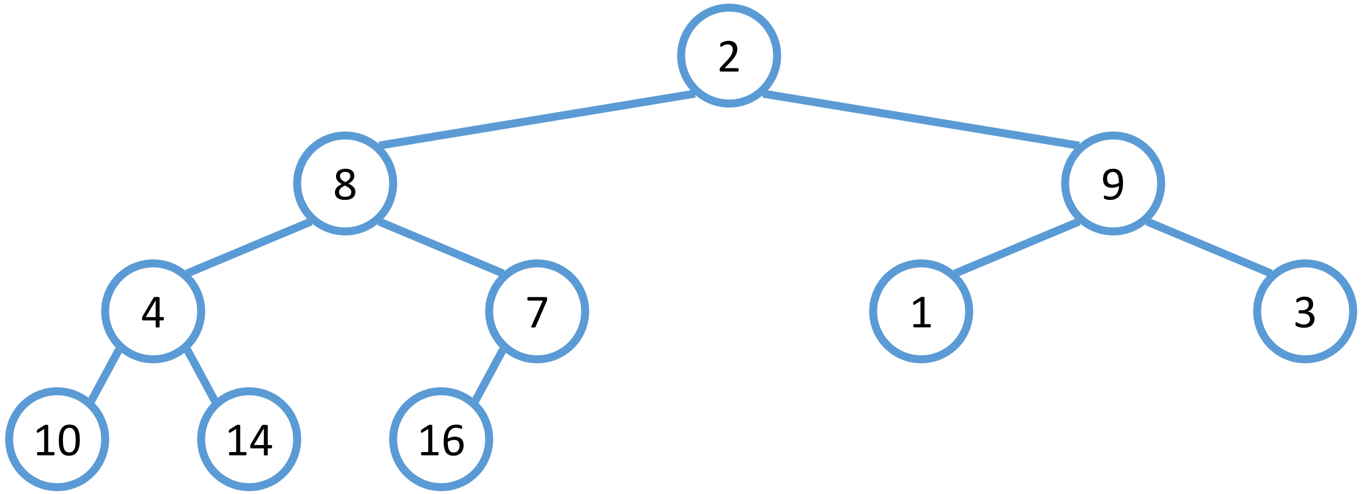
Heap Sort

N = 8



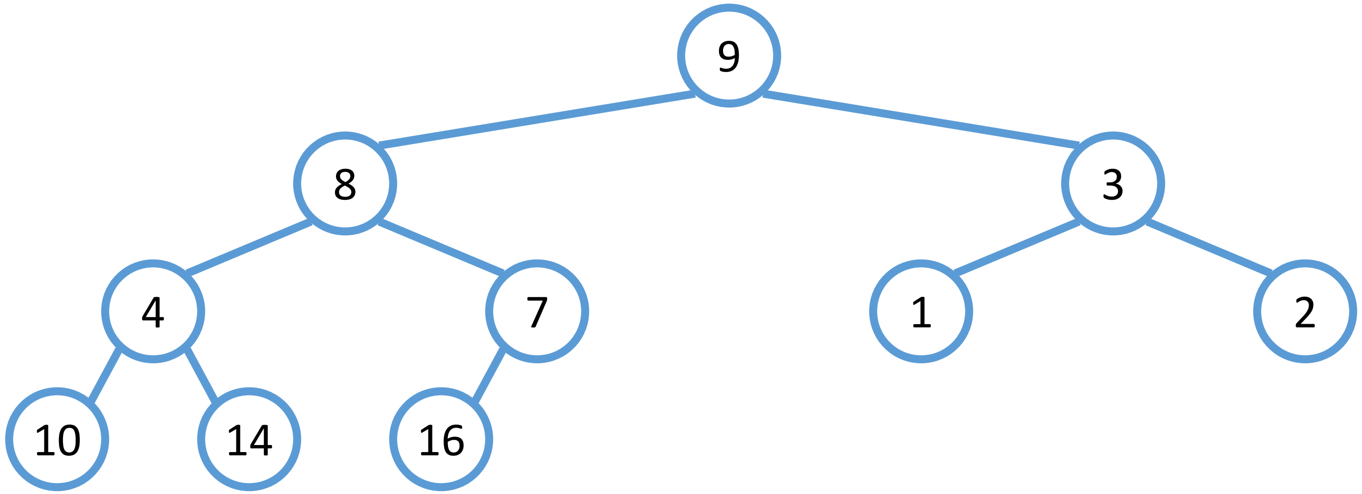
Heap Sort

N = 7



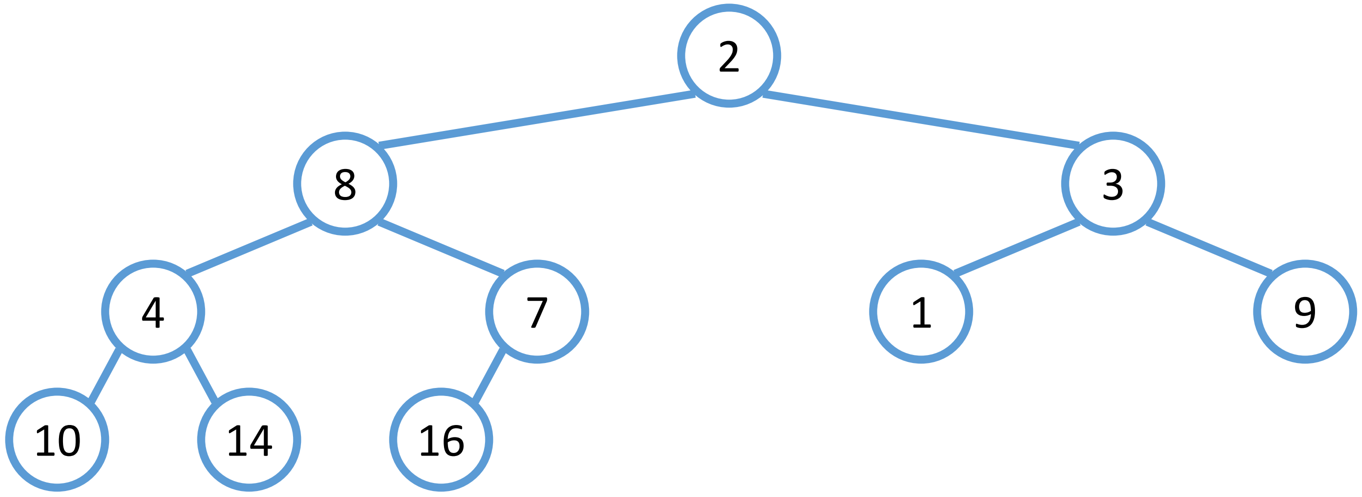
Heap Sort

N = 7



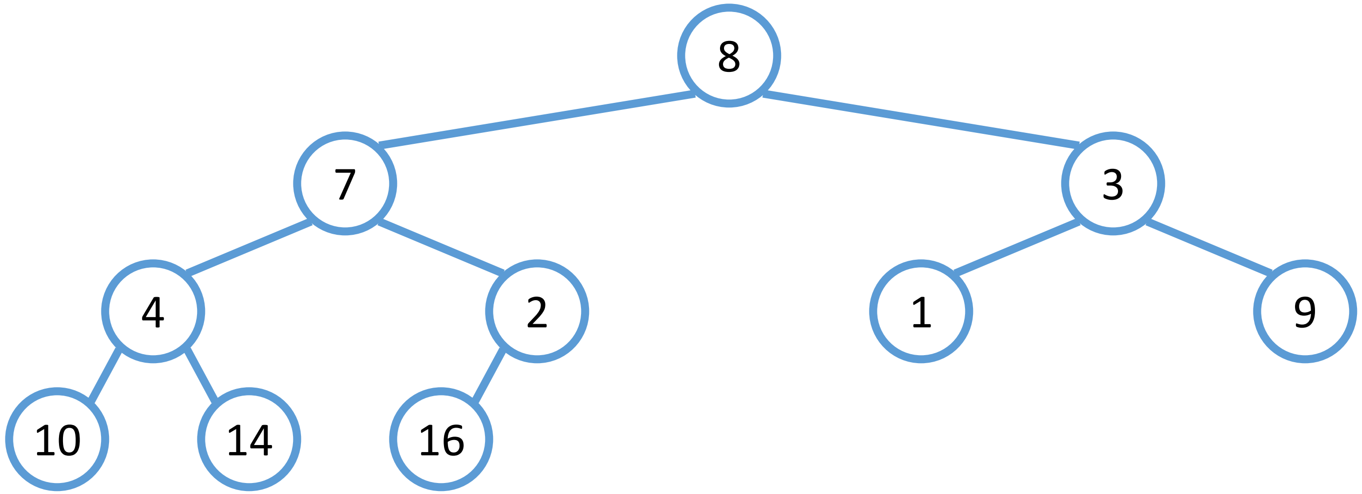
Heap Sort

N = 6



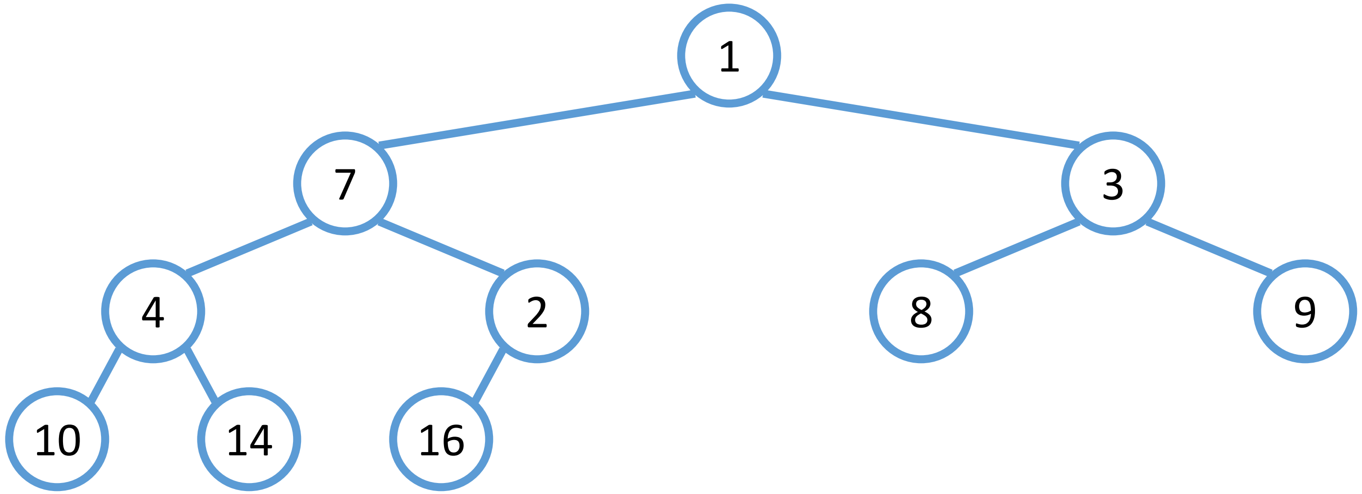
Heap Sort

N = 6



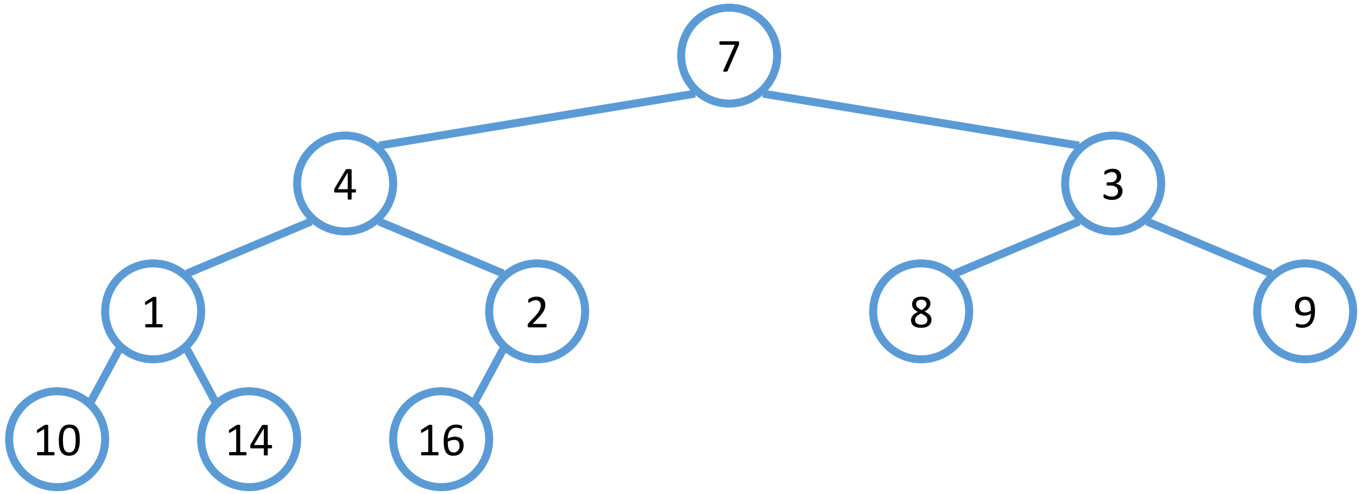
Heap Sort

N = 5



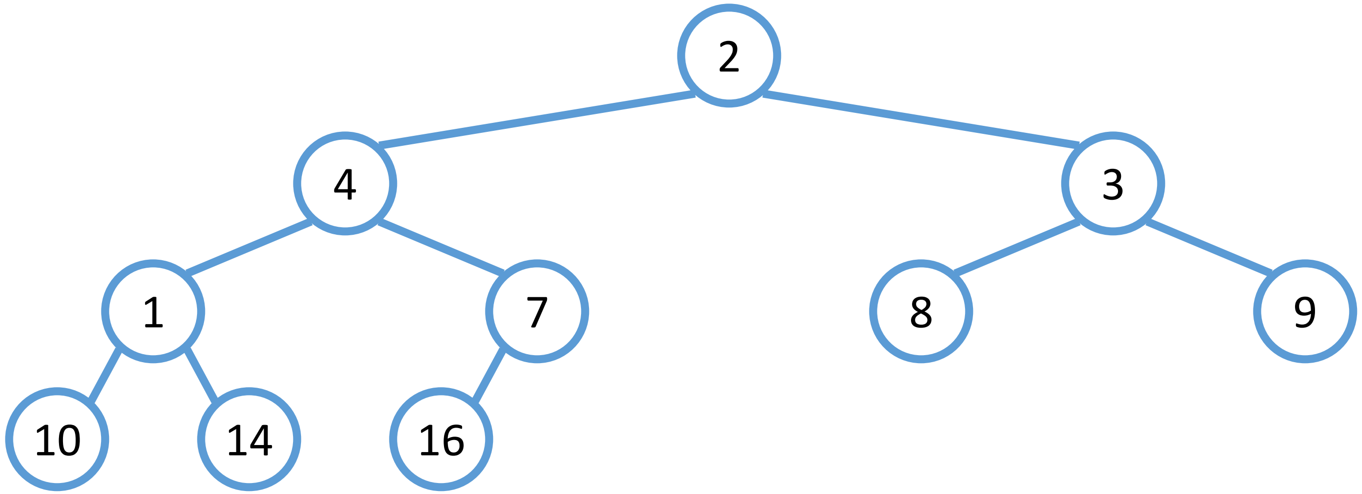
Heap Sort

N = 5



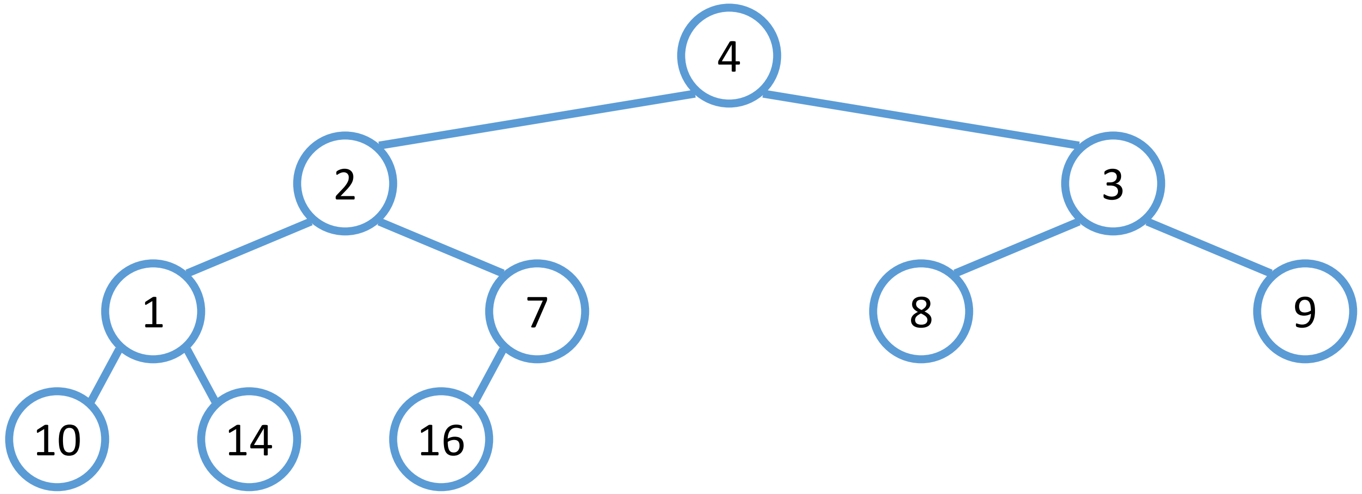
Heap Sort

N = 4



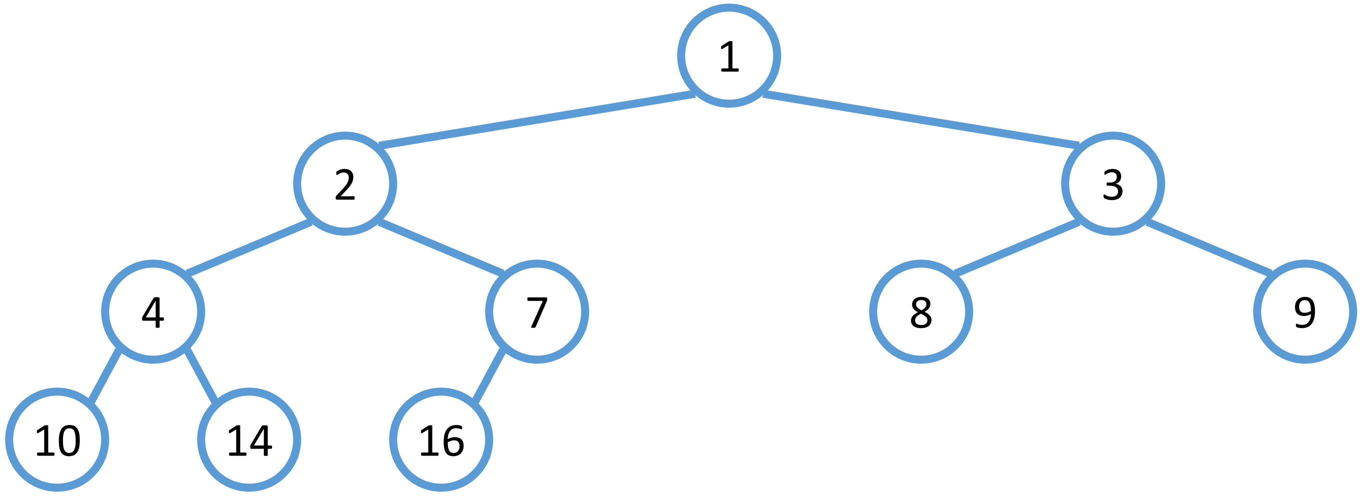
Heap Sort

N = 4



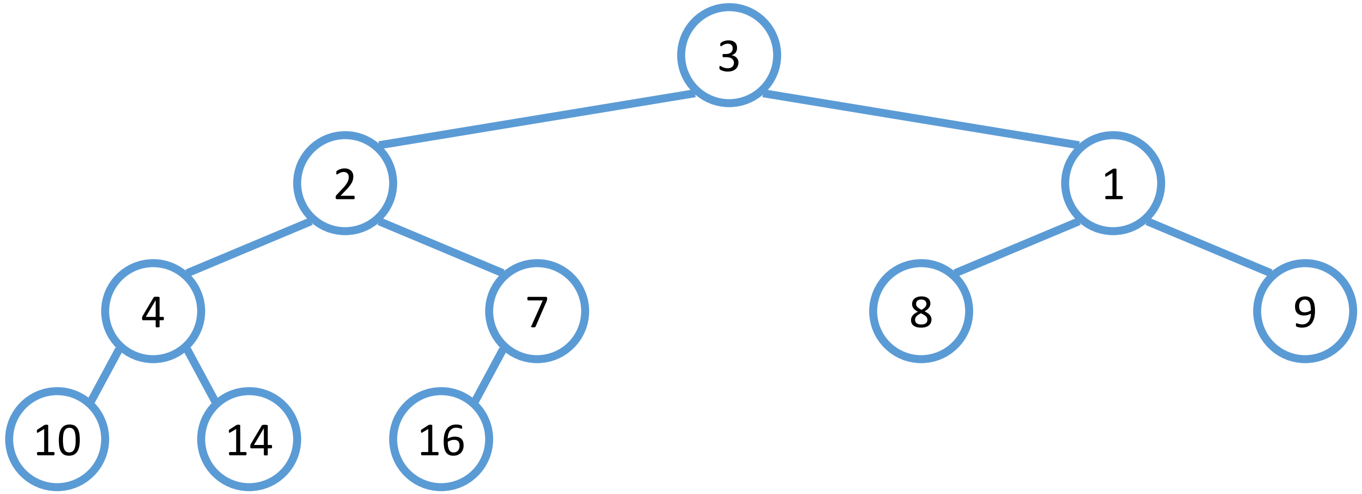
Heap Sort

N = 3



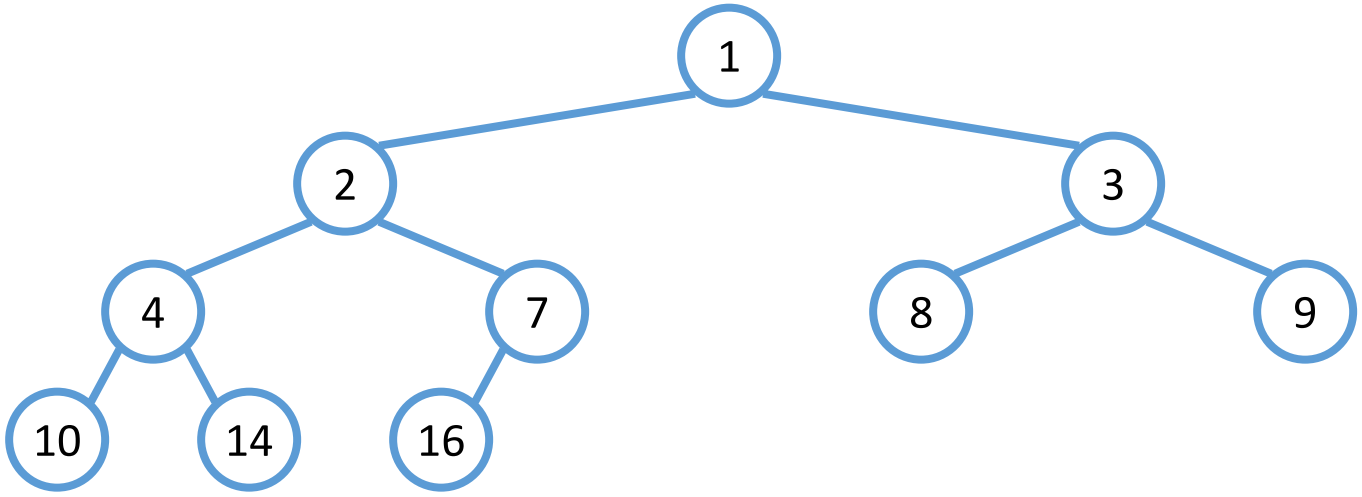
Heap Sort

N = 3



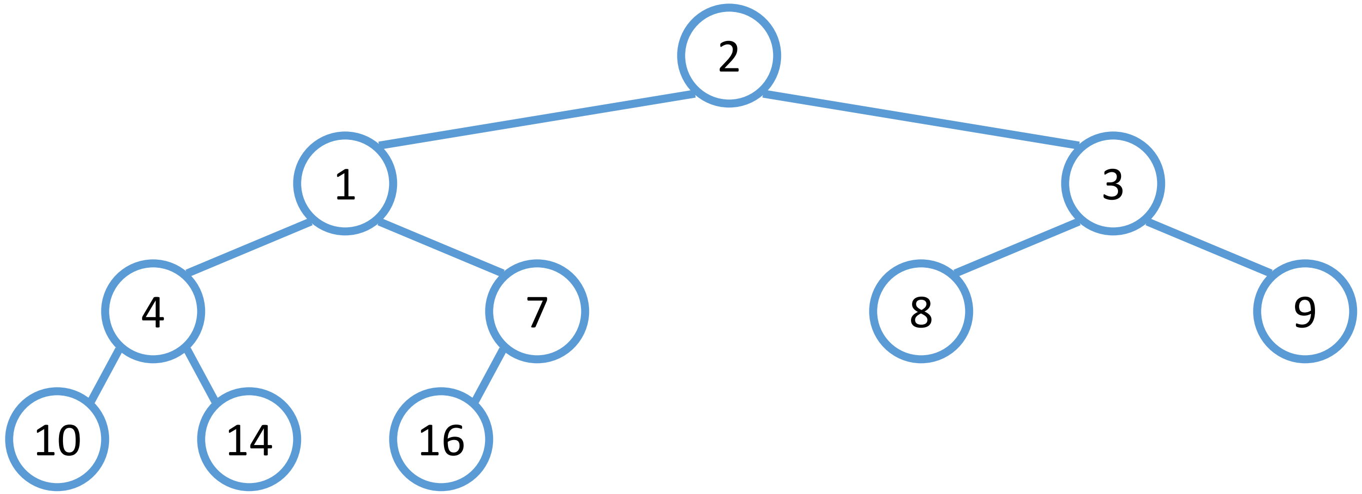
Heap Sort

N = 2



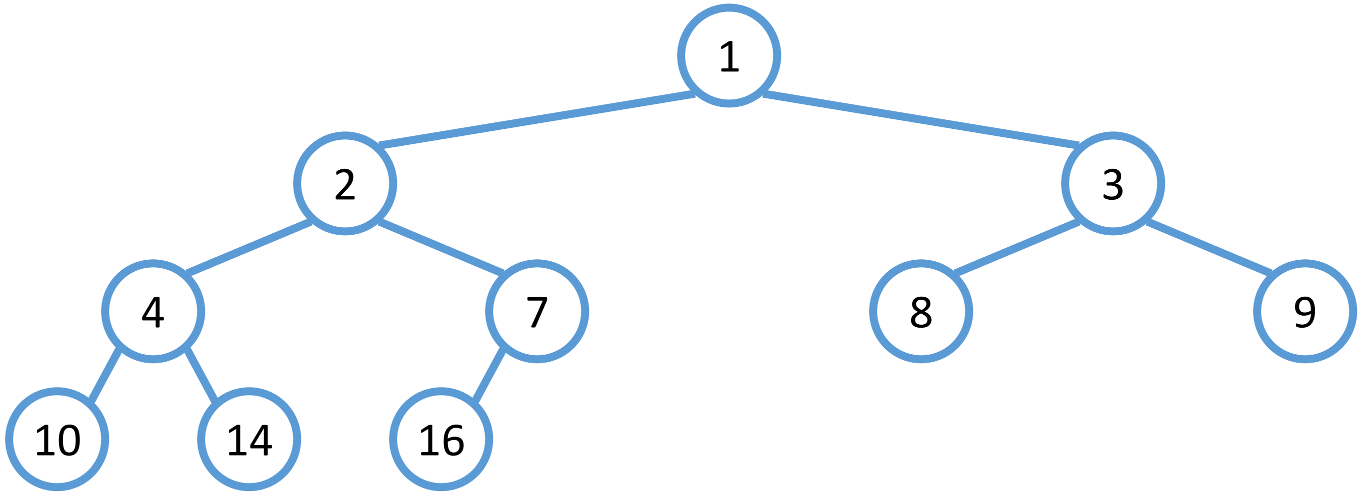
Heap Sort

N = 2



Heap Sort

N = 1



Analyzing Heapsort

- The call to **`BuildHeap()`** takes O(n) time
- Each of the n - 1 calls to **`MoveDown()`** takes O(lg n) time
- Thus the total time taken by **`HeapSort()`**
  = O(n) + (n - 1) O(lg n)
  = O(n) + O(n lg n)
  = O(n lg n)

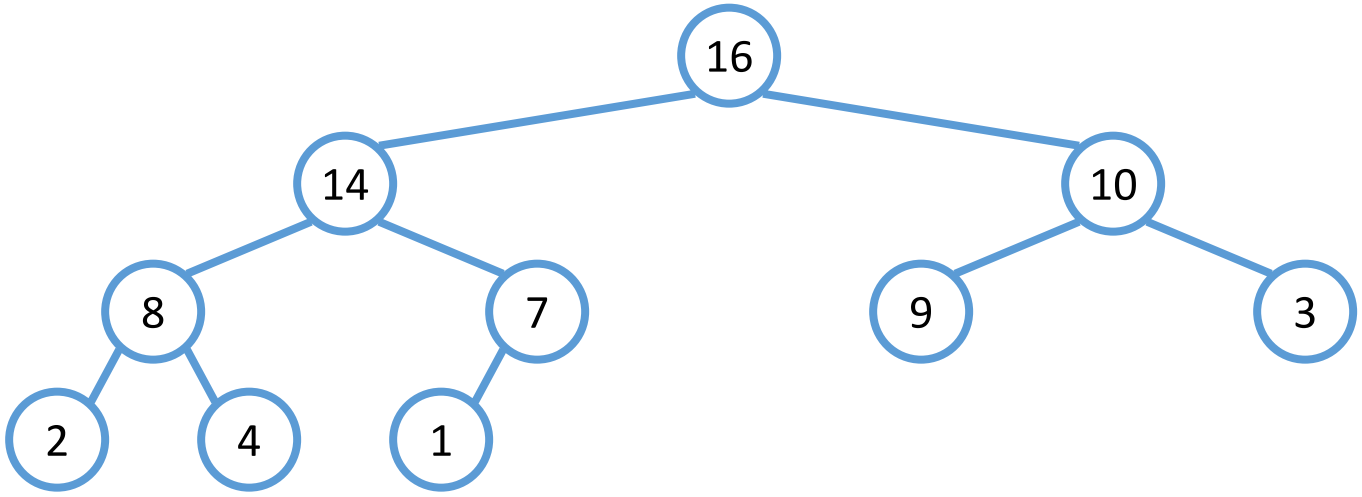
Extract Maximum: O(lg n)

```
int extractMaximum(int a[], int *n)
{
        if(*n <= 0) {
                printf("\nBinary Heap is empty.\n");
                return -1 * MAX_SIZE;
        }
        int max = a[0];
        a[0] = a[(*n)-1];
        (*n)--;
        movedown(a, *n, 0);
        return max;
}
```

Extract Maximum

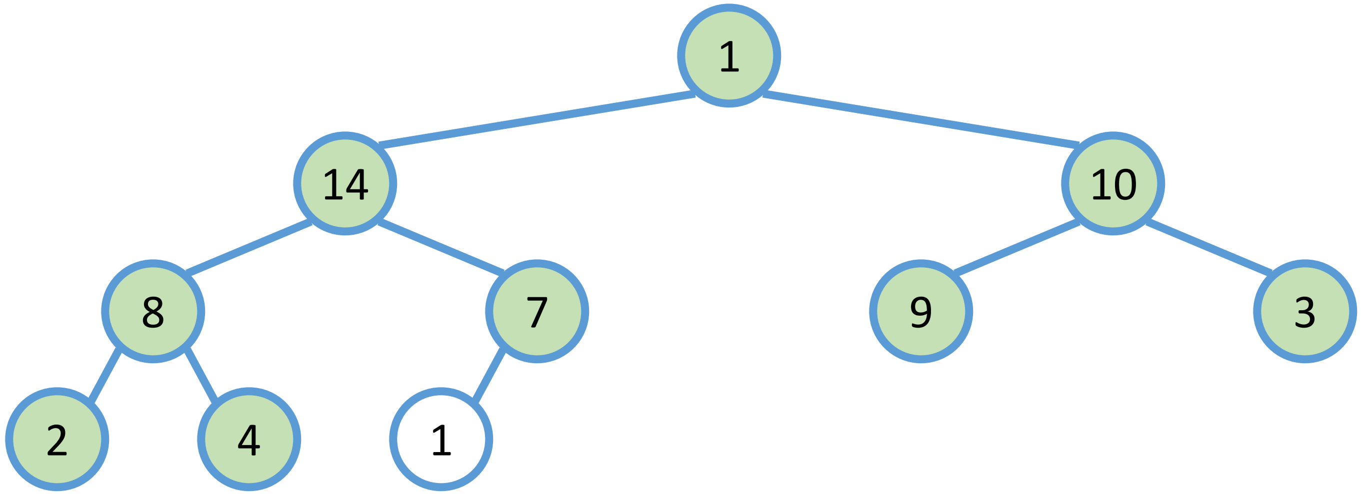
N = 10

Max = a[0] = 16



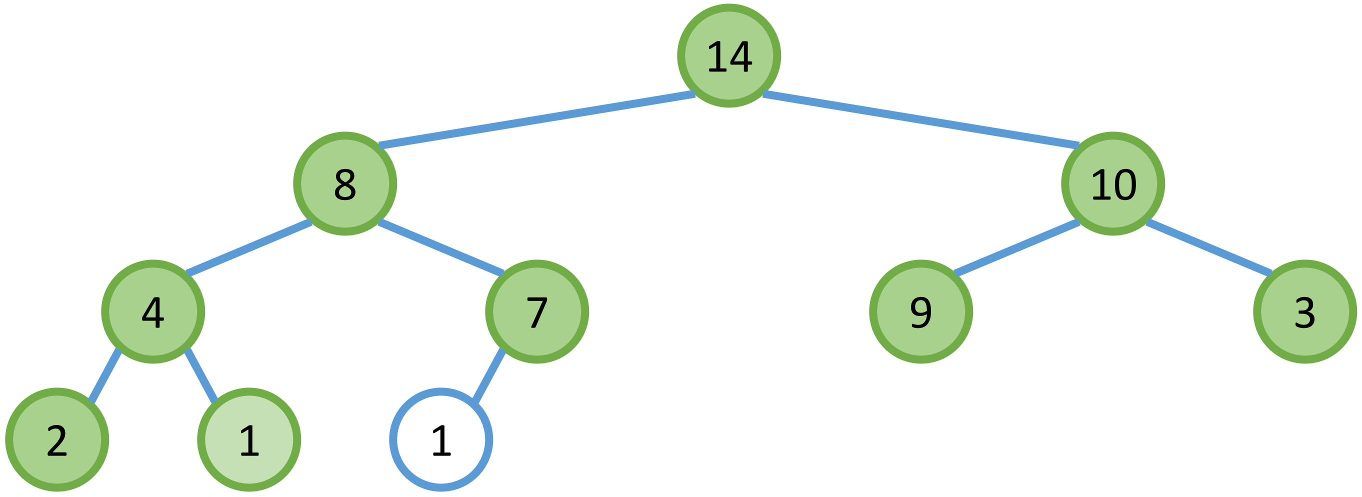
Extract Maximum

N = 9



Extract Maximum

N = 9



Move Up: O(lg n)

Moveup() is used to restore the heap property at index i. It checks whether a[i] > a[parent] in the max heap. If so, swap a[i] and a[parent], and recursively apply Moveup() operation on a[parent].

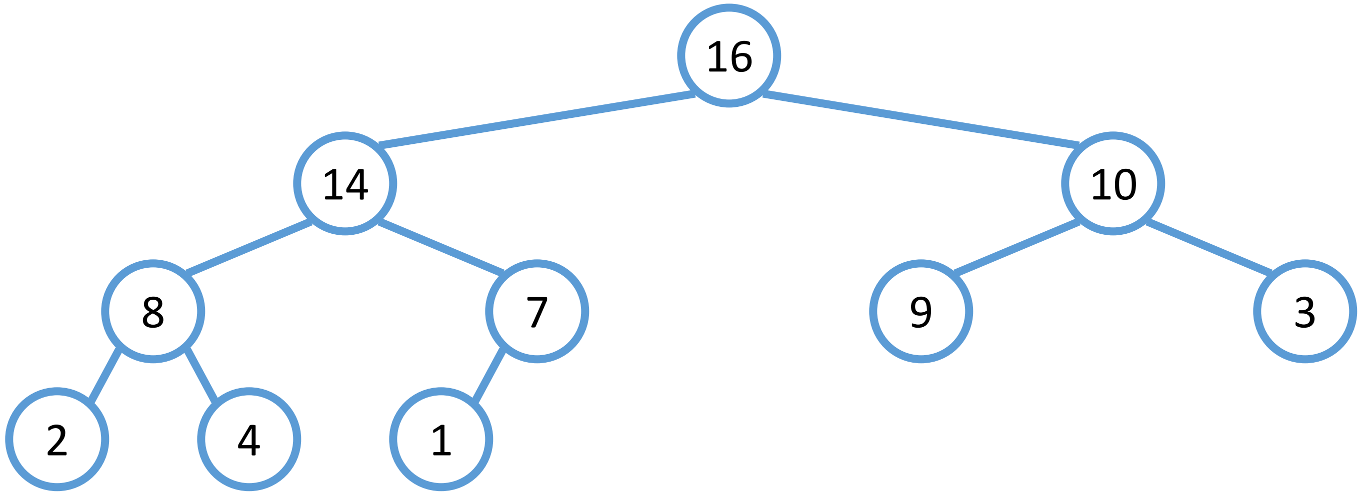
```
void moveup(int a[], int i)
{
        int p = (i-1)/2;

        if(p >= 0 && a[i] > a[p]) {
                swap(a, i, p);
                moveup(a, p);
        }
}
```

Insert into Binary Heap: O(lg n)

```
void insertBinaryHeap(int a[], int *n, int x)
{
        if((*n) == MAX_SIZE) {
                printf("\nBinary Heap is full.\n");
                return;
        }
        (*n)++;
        a[(*n)-1] = x;
        moveup(a, (*n)-1);
}
```

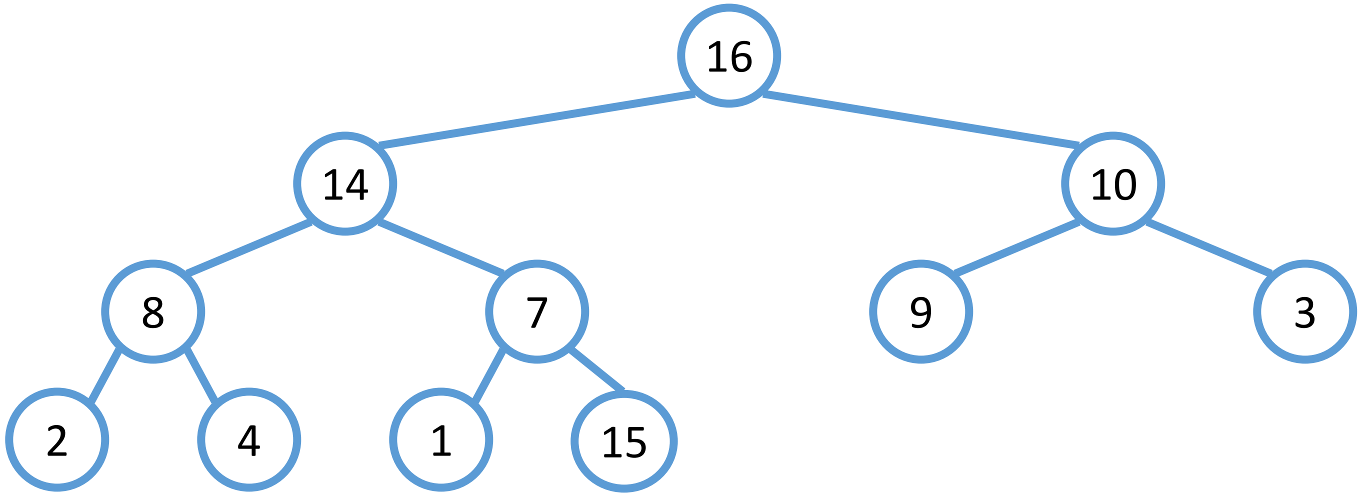
Insert 15 into Binary Heap

N = 10



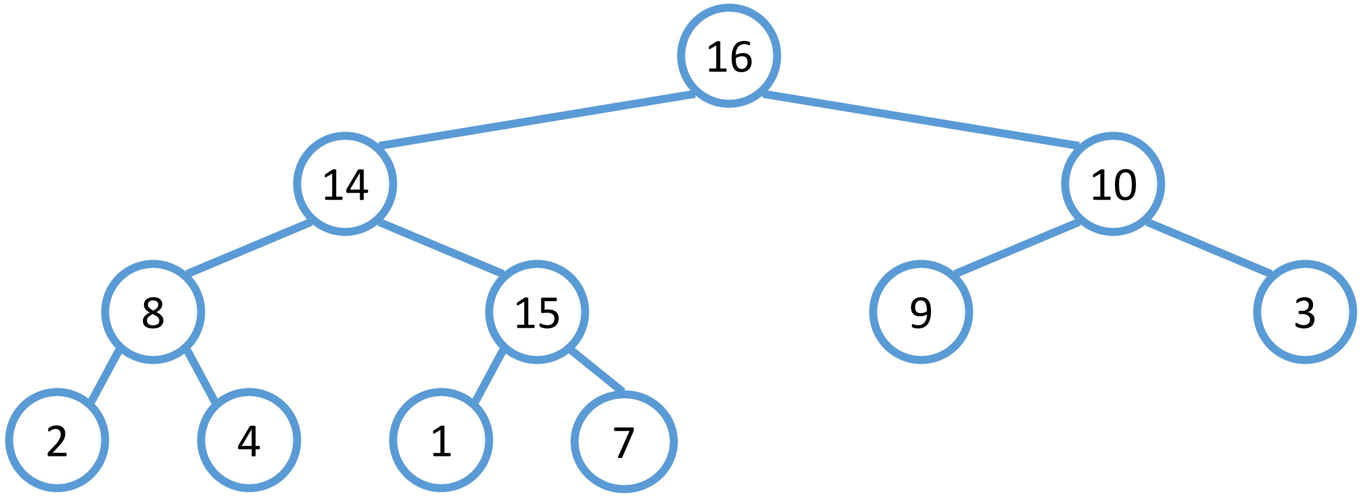
Insert into Binary Heap

N = 11



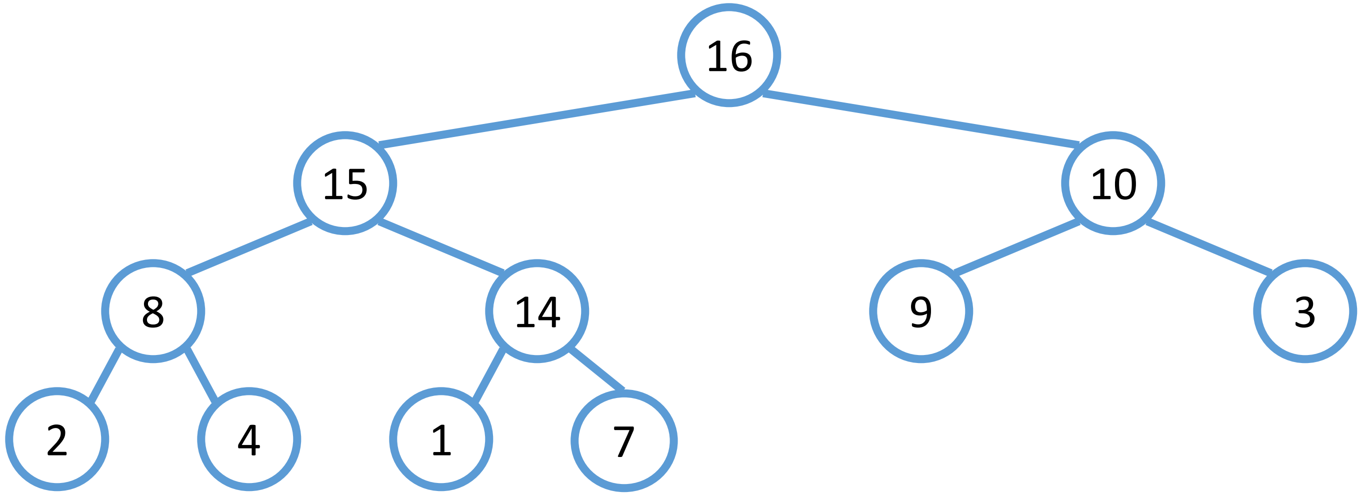
Insert into Binary Heap

N = 11



Insert into Binary Heap

N = 11

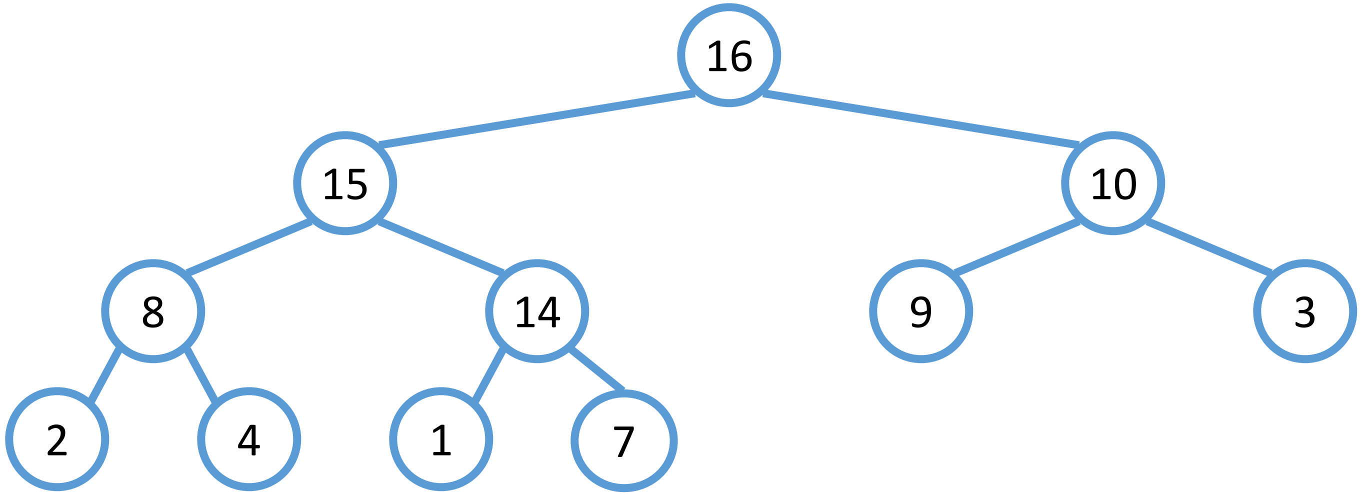


Increase Key: O(lg n)

```
void increaseKey (int a[], int n, int i, int x)
{
        if(x > a[i]) {
                 a[i] = x;
                moveup(a, i);
        }
}
```

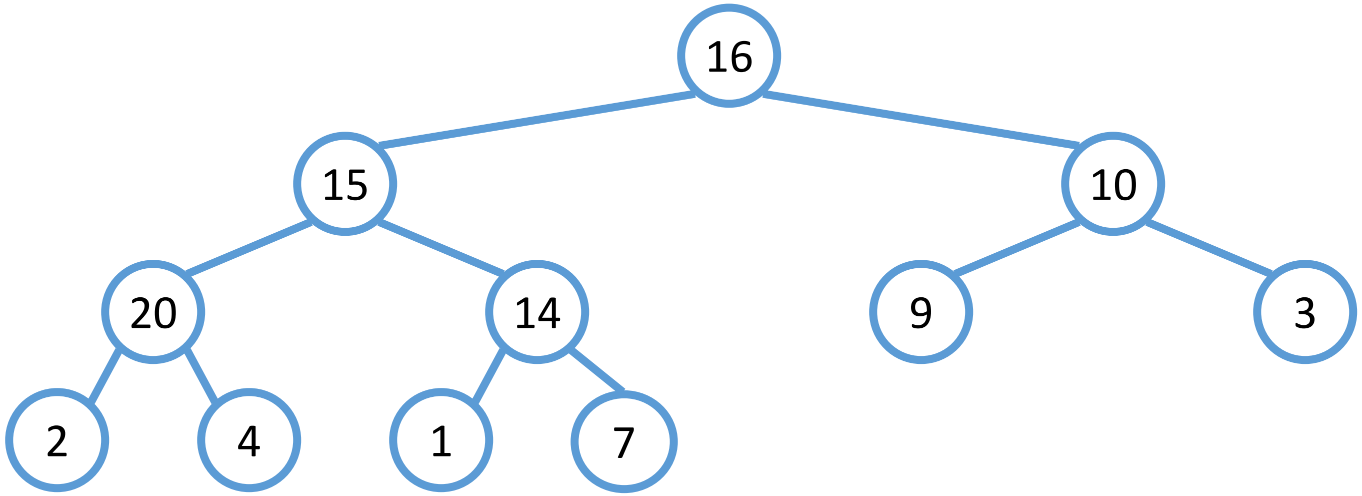
Increase Key at index 3 to 20

N = 11



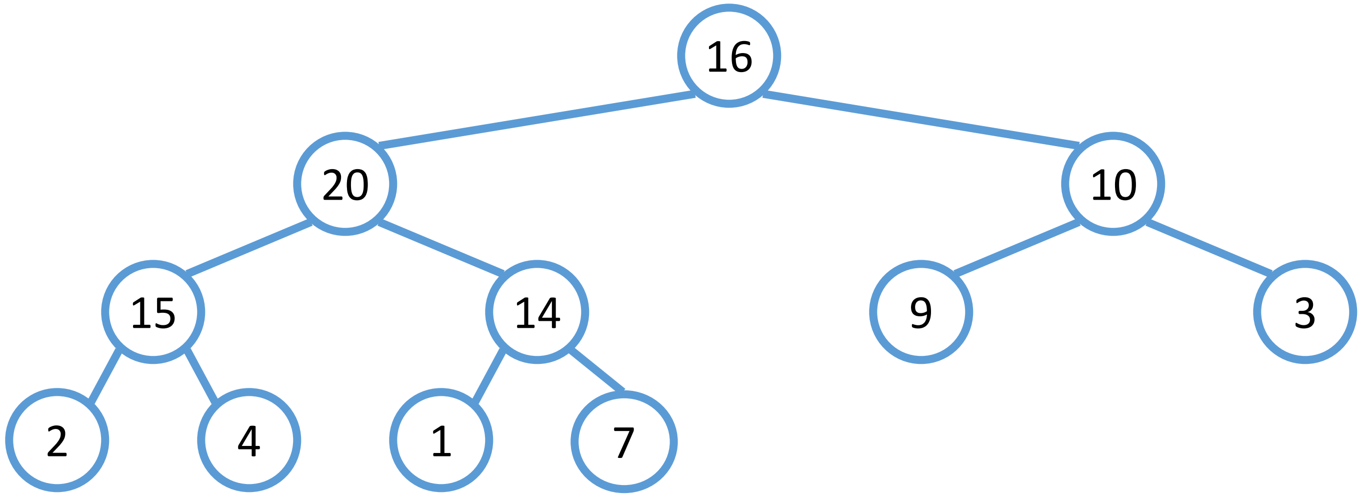
Increase Key at index 3 to 20

N = 11



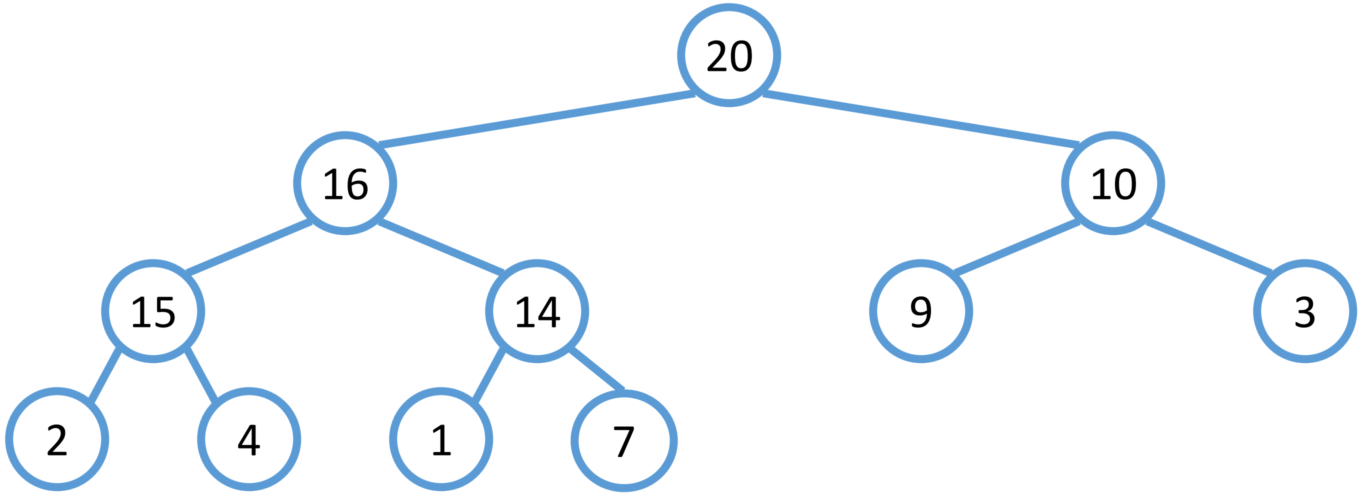
Increase Key at index 3 to 20

N = 11



Increase Key at index 3 to 20

N = 11

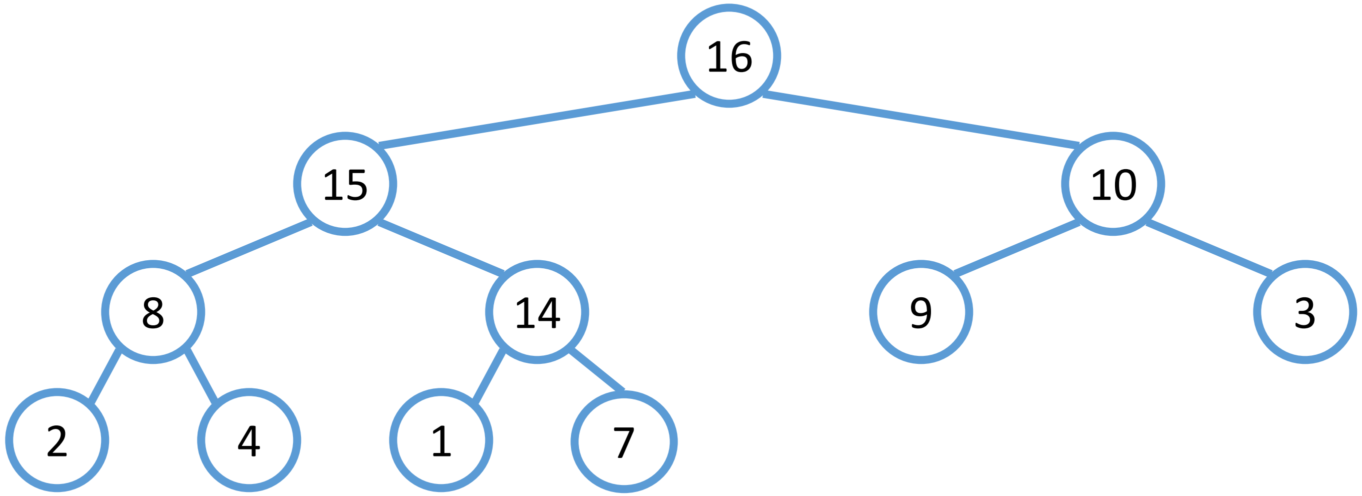


Decrease Key: O(lg n)

```
void decreaseKey (int a[], int n, int i, int x)
{
        if(x < a[i]) {
                a[i] = x;
                movedown(a, n, i);
        }
}
```

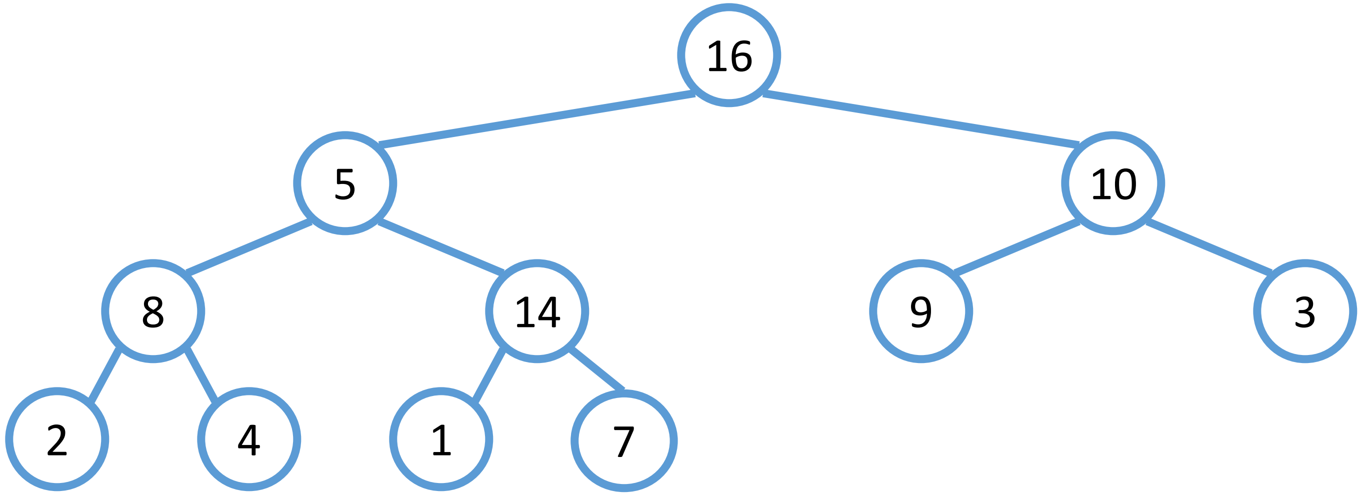
Decrease Key at index 1 to 5

N = 11



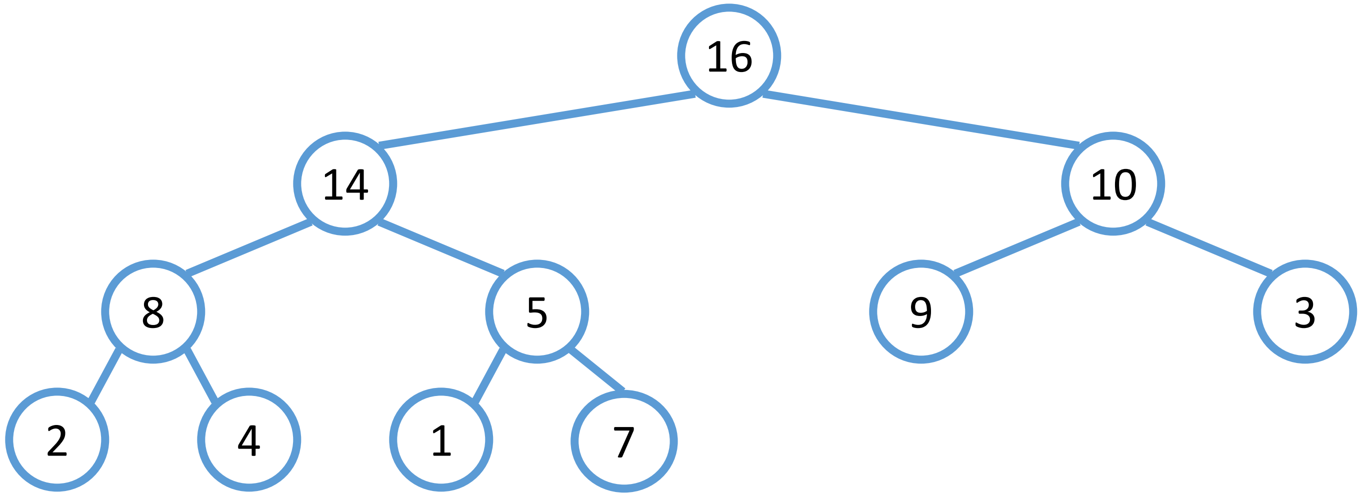
Decrease Key at index 1 to 5

N = 11



Decrease Key at index 1 to 5

N = 11



Decrease Key at index 1 to 5

N = 11