Data Structures and Algorithms (CEN3016)

Dr. Muhammad Umair Khan

Assistant Professor

Department of Computer Engineering

National University of Technology

Heap

- A heap is a type of data structure. One of the interesting things about heaps is that they allow you to find the largest element in the heap in O(1) time. (Recall that in certain other data structures, like arrays, this operation takes O(n) time.)
- Furthermore, extracting the largest element from the heap (i.e. finding and removing it) takes O(log n) time.
- These properties make heaps very useful for implementing a “priority queue," which we’ll get to later.
- They also give rise to an O(n log n) sorting algorithm, “heapsort," which works by repeatedly extracting the largest element until we have emptied the heap.

Two Special Heaps

A **heap** is a certain kind of complete binary tree.

- ► A heap is a **kind of** **tree** that offers both insertion and deletion in $O(\log_2 n)$ time.
- ► Fast for insertions; not so fast for deletions.

- Max-Heap
- Min-Heap

Max-Heap

In a max-heap, every node i other than the root satisfies the following property:

$$A[\text{Parent}(i)] \geq A[i].$$

Min-Heap

In a min - heap, every node i other than the root satisfies the following property :

$$A[\text{Parent}(i)] \le A[i].$$

Heaps

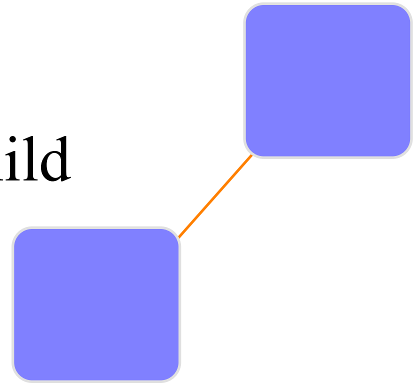
Root

When a complete binary tree is built, its first node must be the root.

Heaps

Almost Complete binary tree.

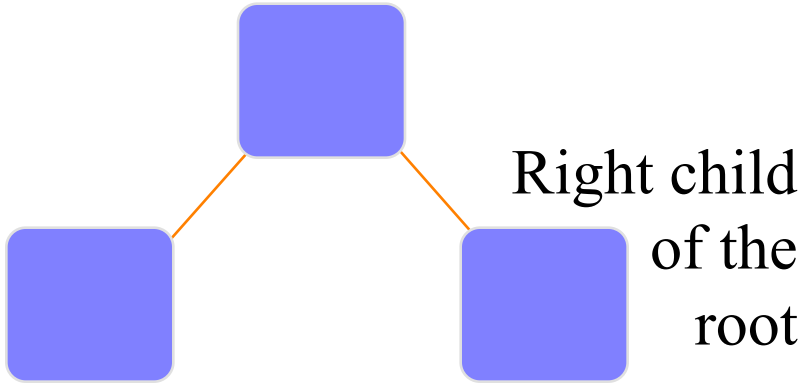
Left child of the root



The second node is always the left child of the root.

Heaps

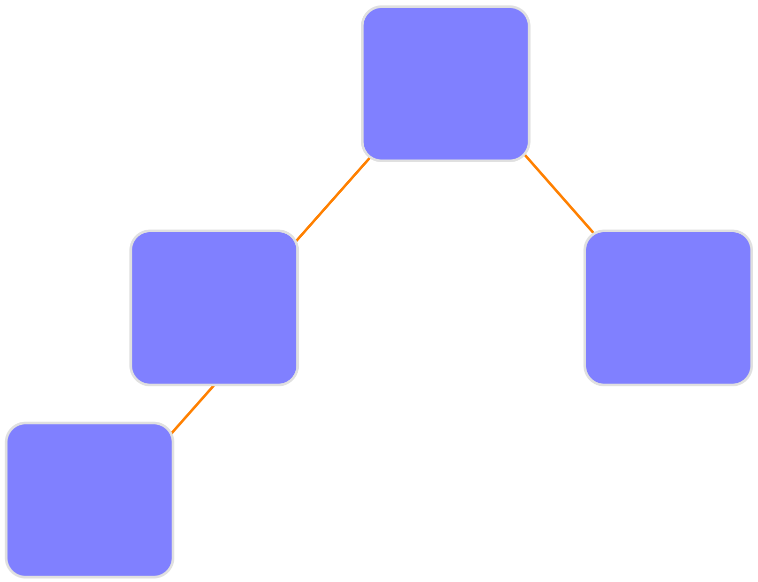
Almost Complete binary tree.



The third node is always the right child of the root.

Heaps

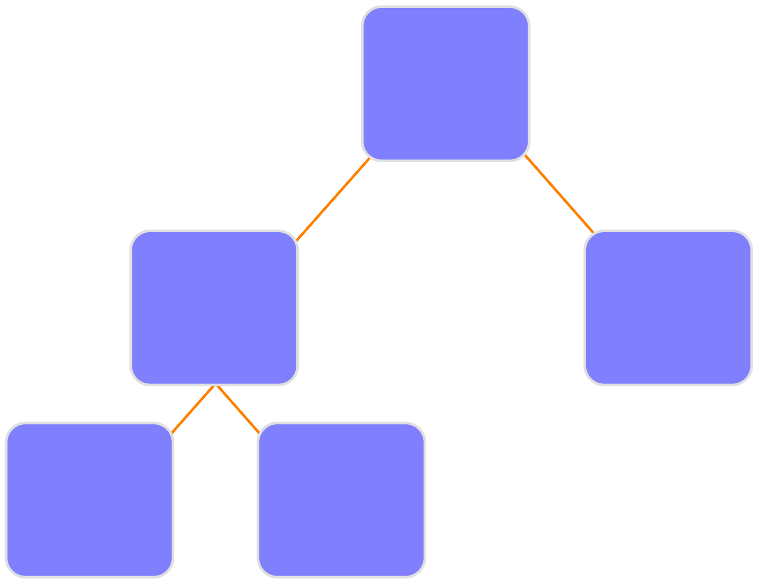
Almost Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

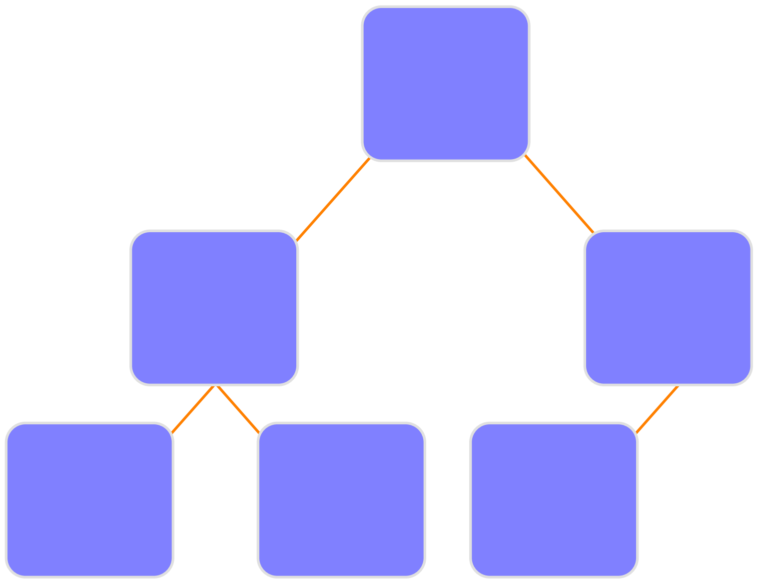
Almost Complete
binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

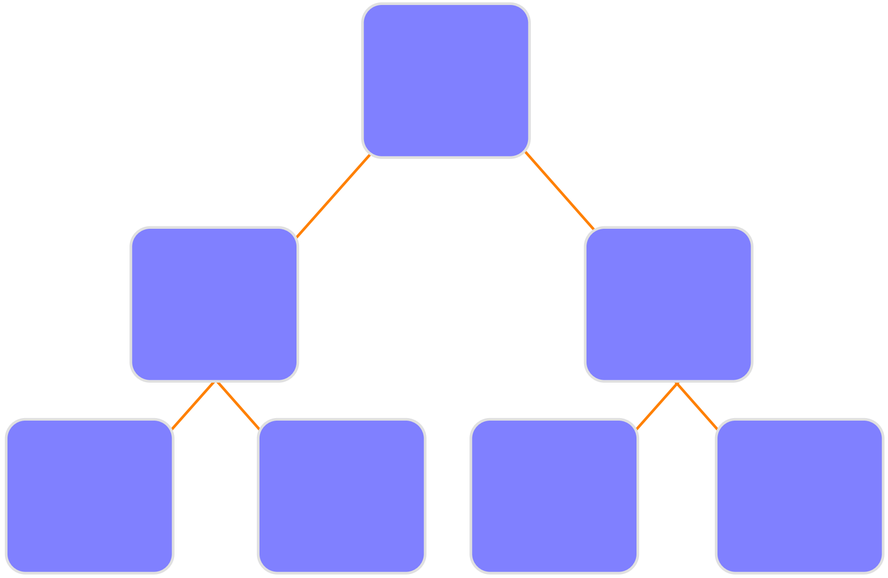
Almost Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

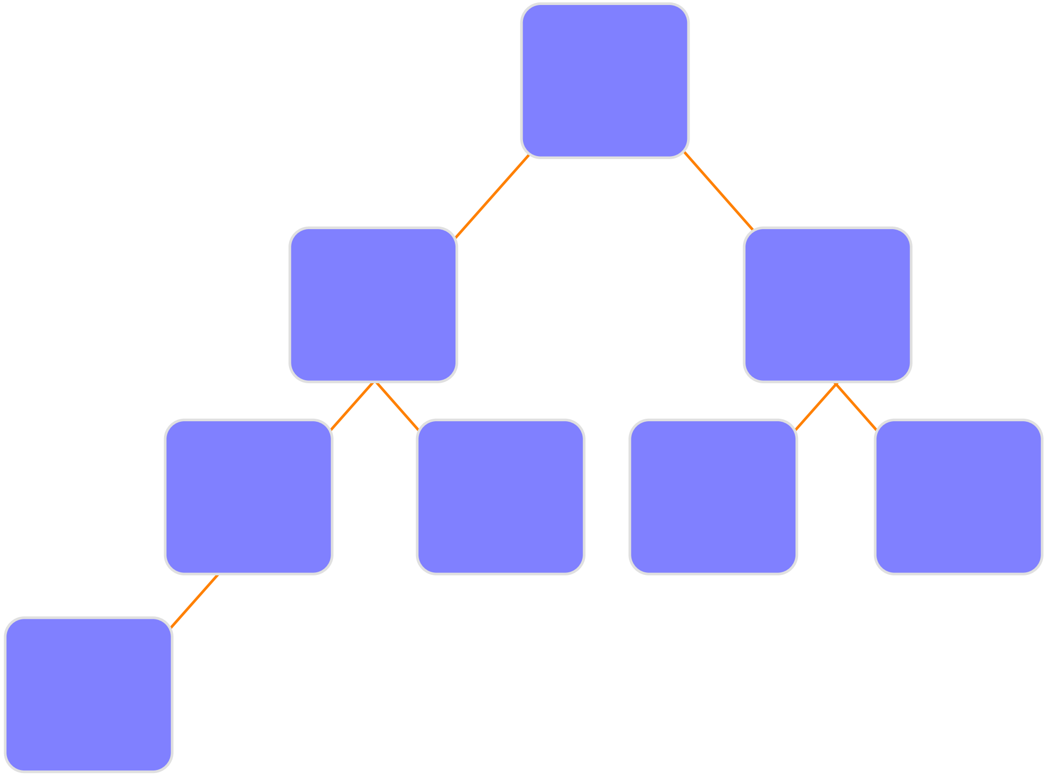
Almost Complete binary tree.



The next nodes always fill the next level from left-to-right.

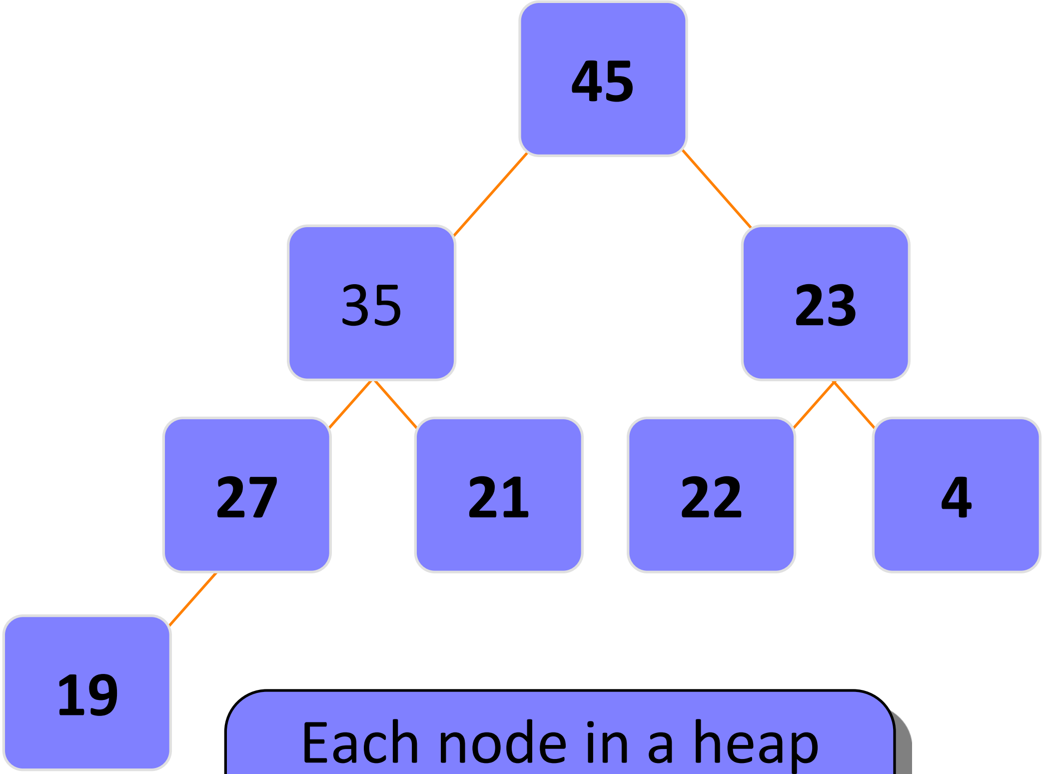
Heaps

Almost Complete binary tree.



Heaps

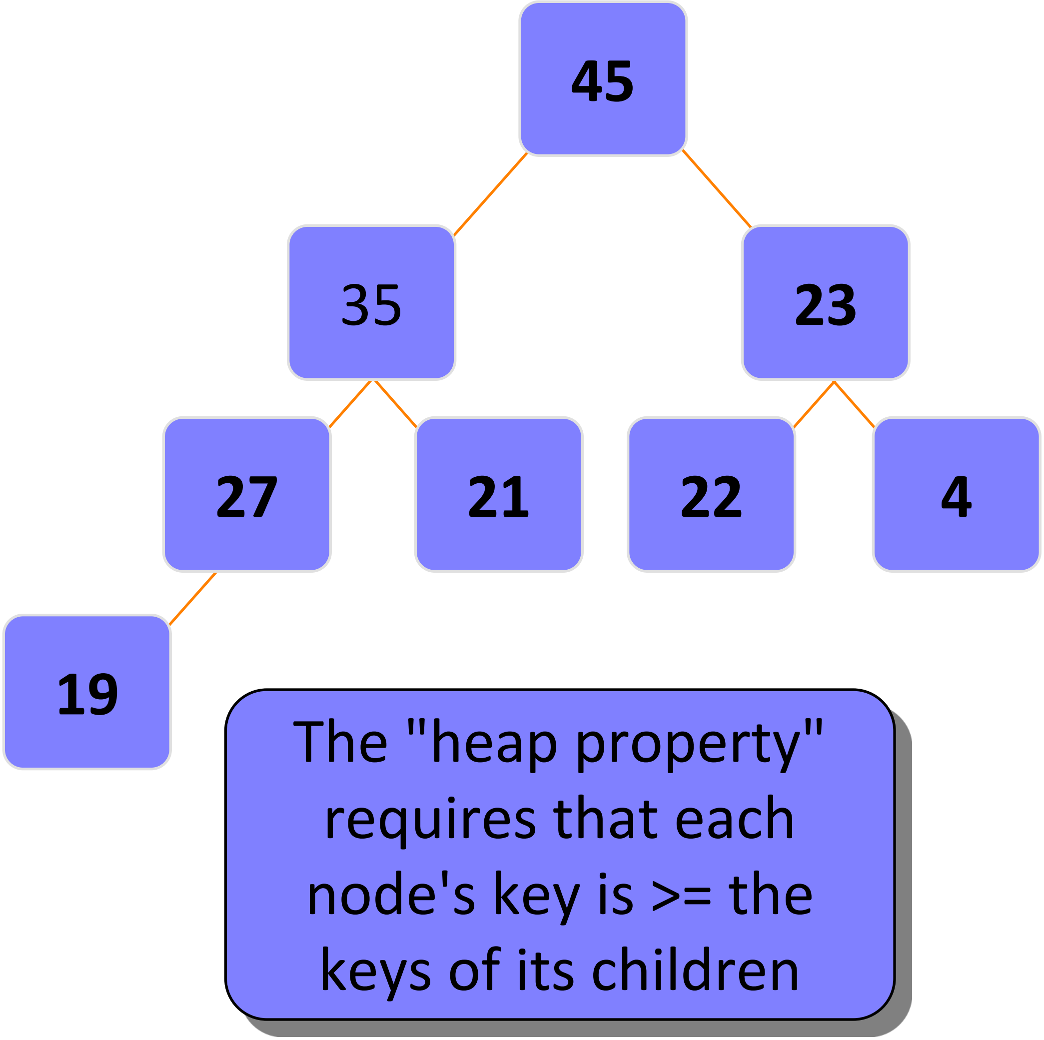
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

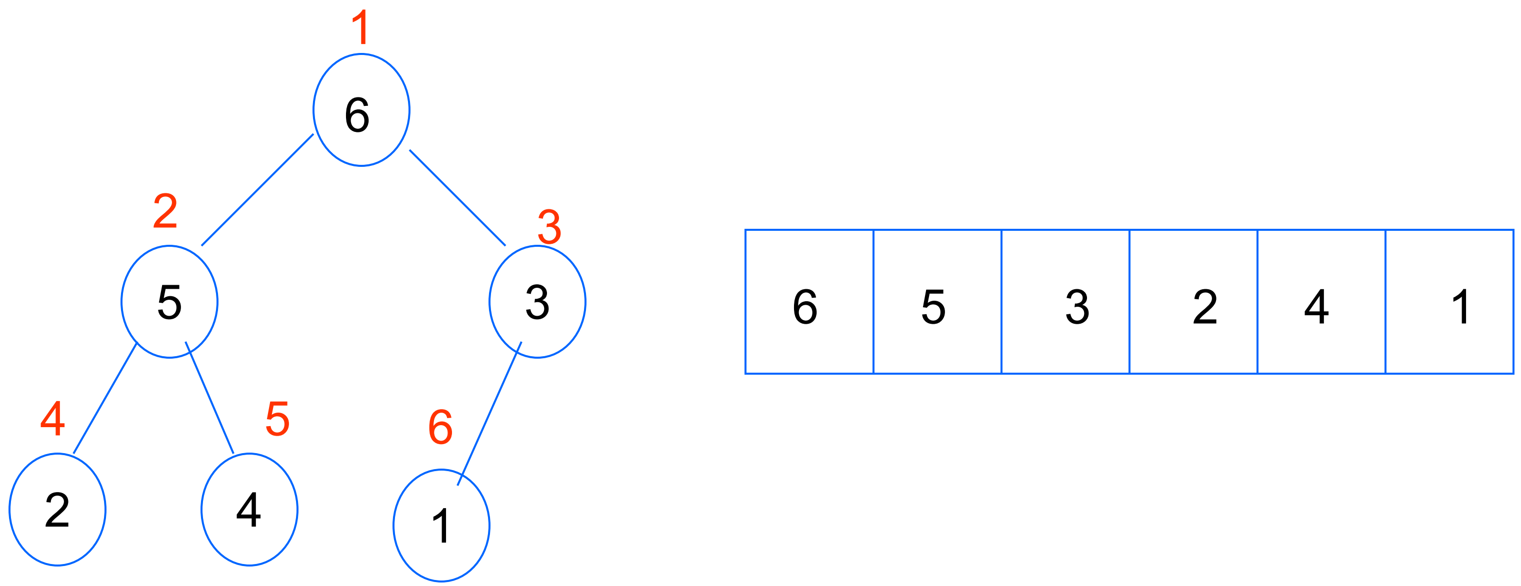
Heaps

A heap is a **certain** kind of complete binary tree.



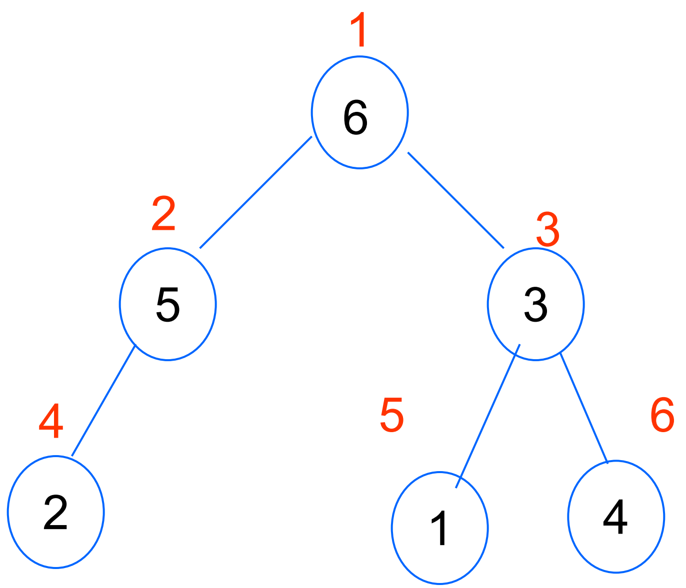
A Data Structure Heap

- A heap is a nearly complete binary tree which can be easily implemented on an array.

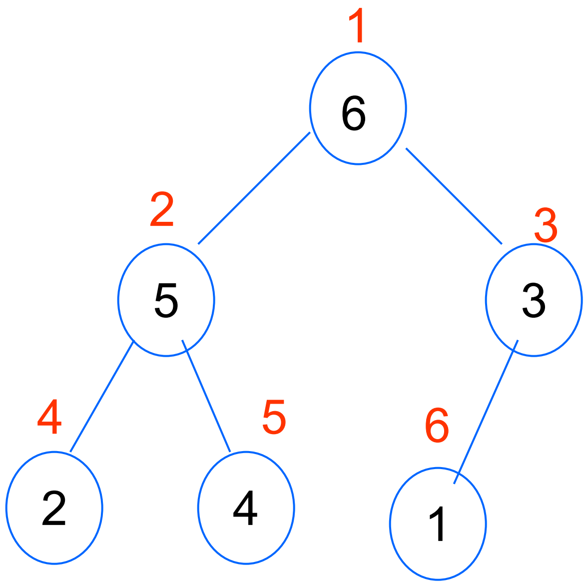


Nearly complete binary tree

- Every level except bottom is complete.
- On the bottom, nodes are placed as left as possible.



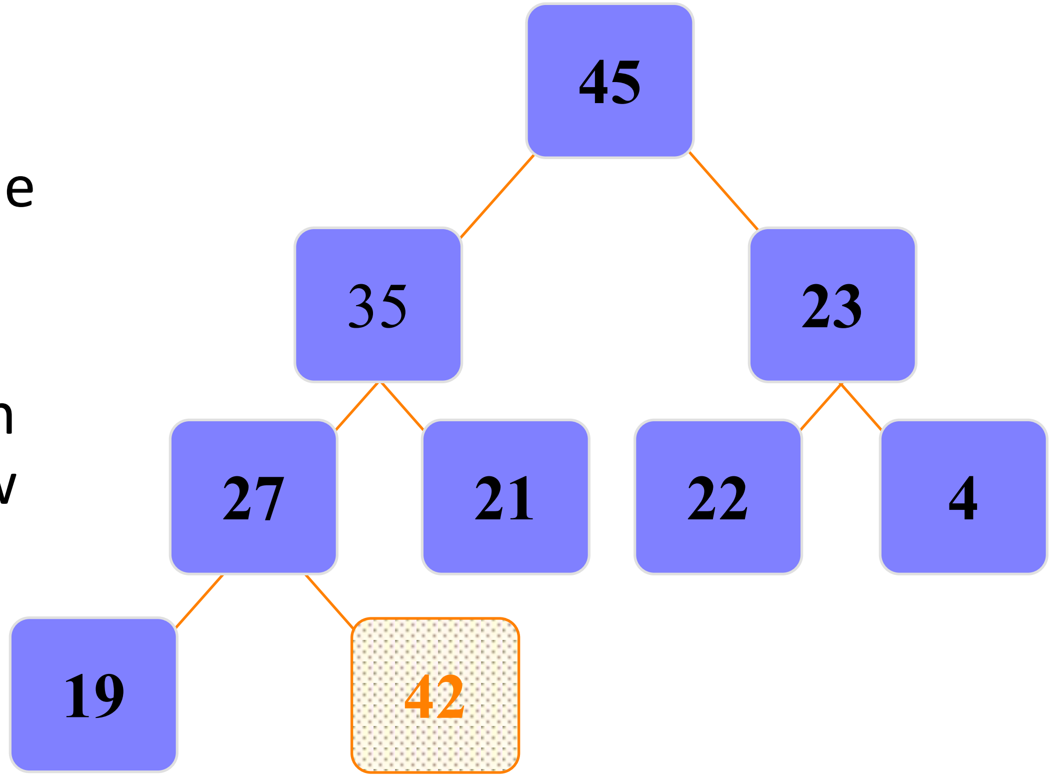
No!



Yes!

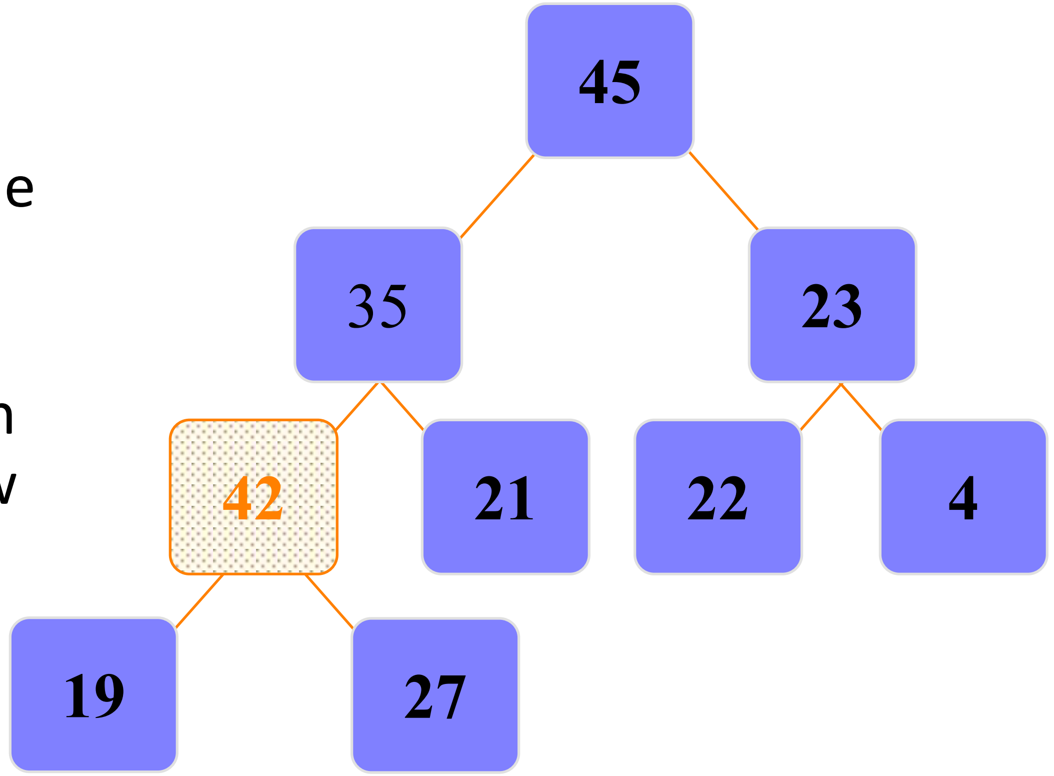
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



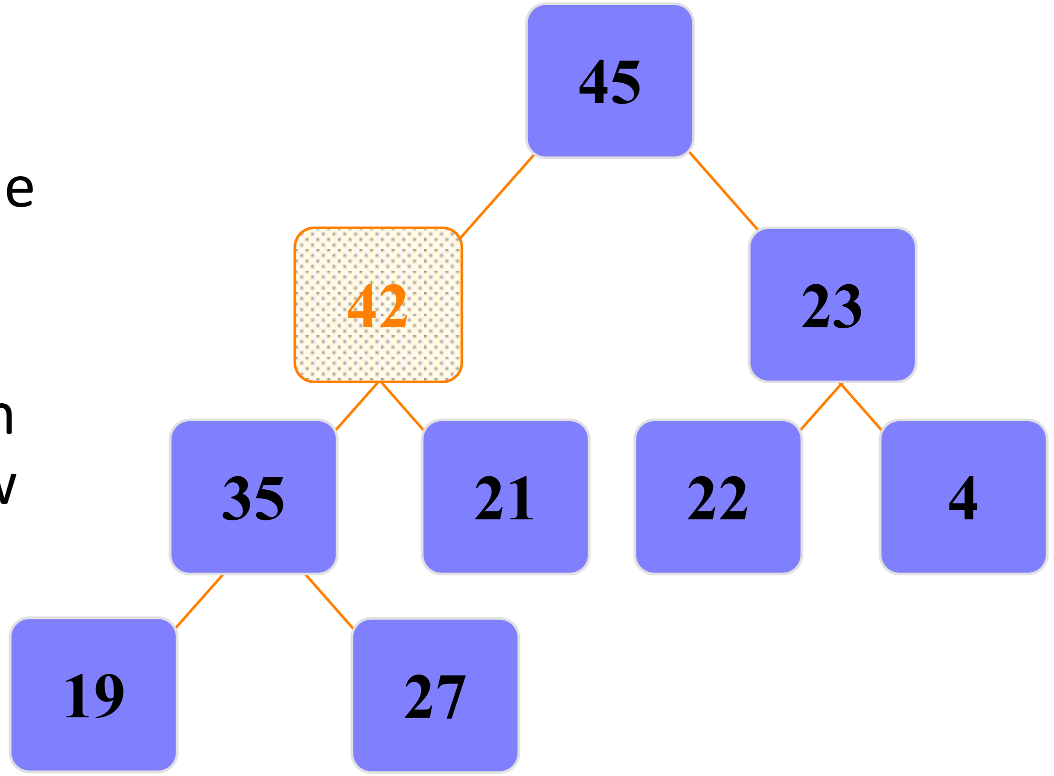
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



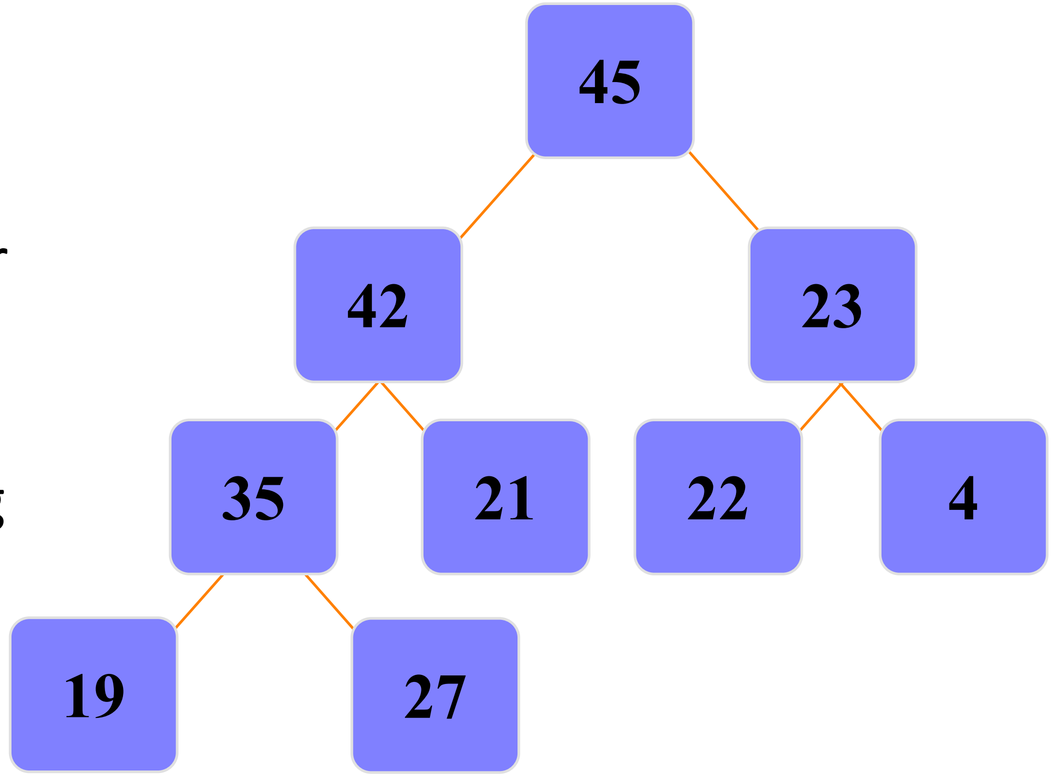
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



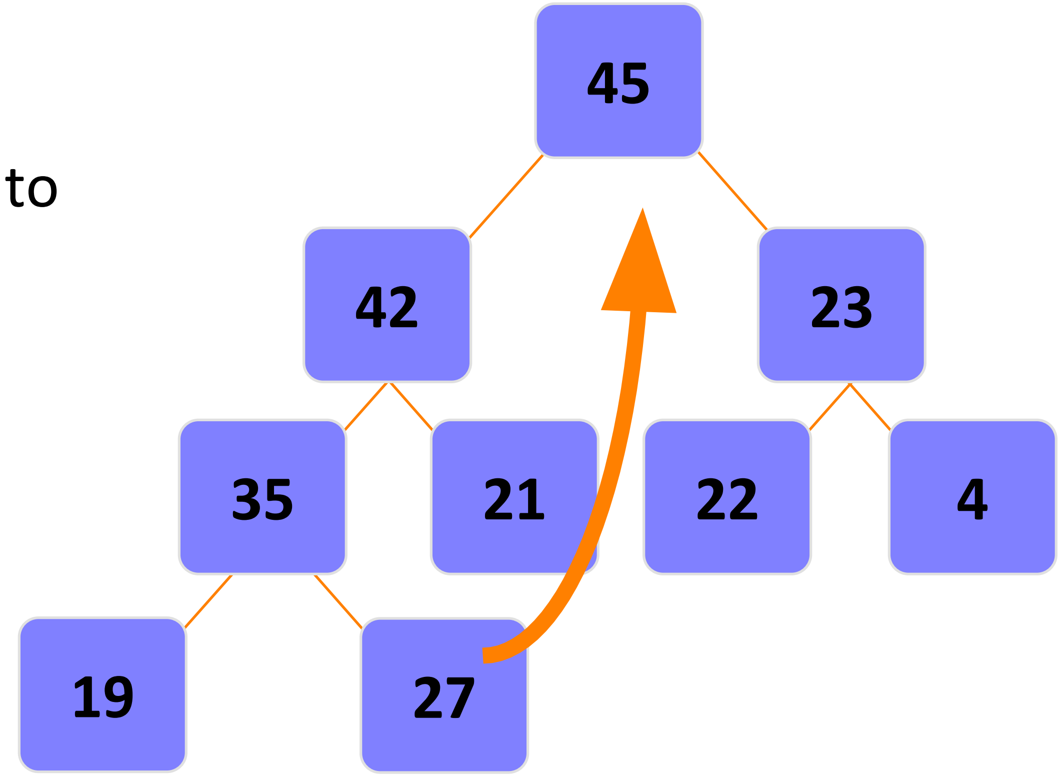
Adding a Node to a Heap

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



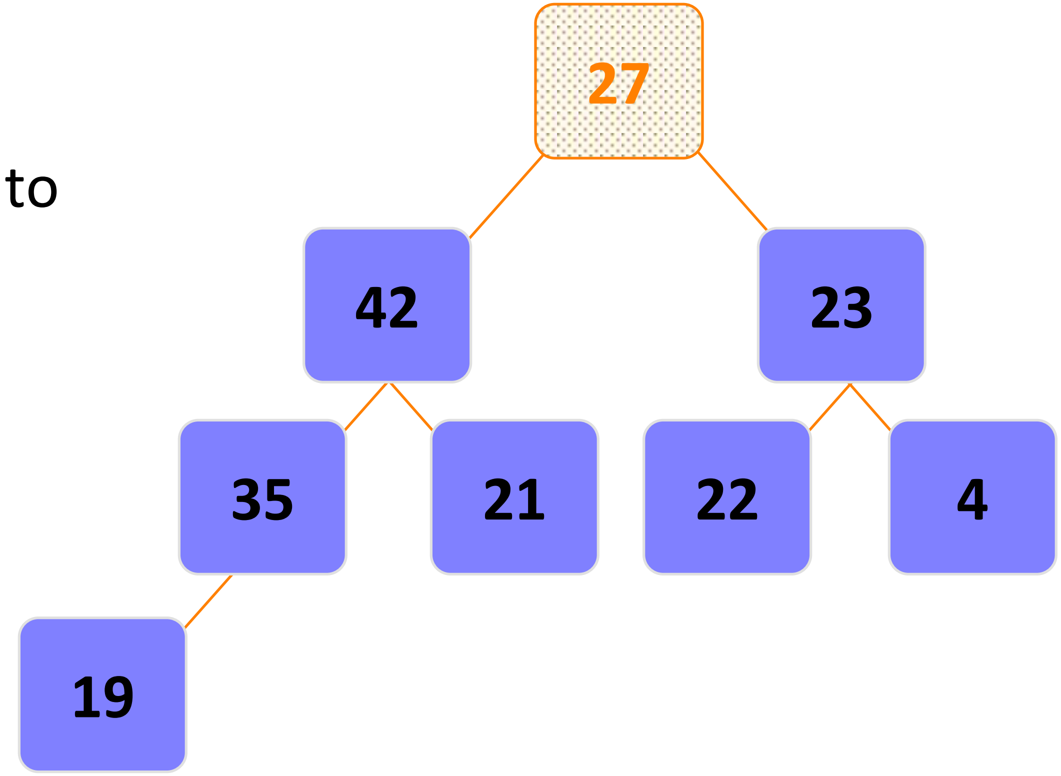
Removing the Top of a Heap

- Move the last node onto the root.



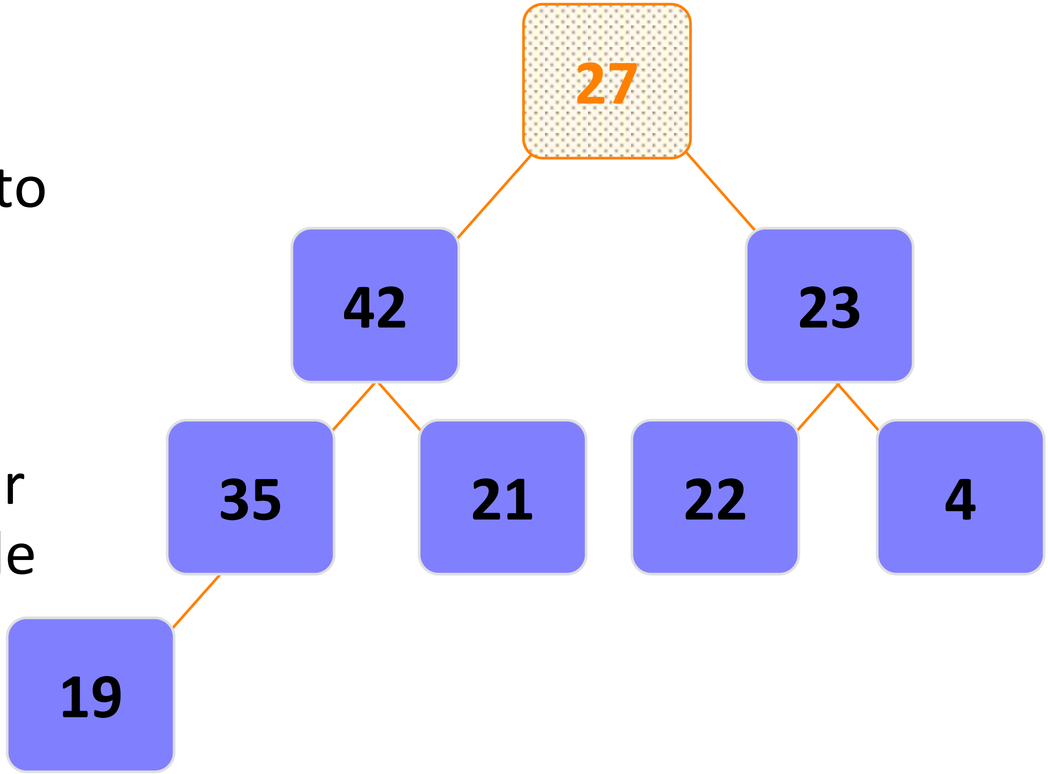
Removing the Top of a Heap

- Move the last node onto the root.



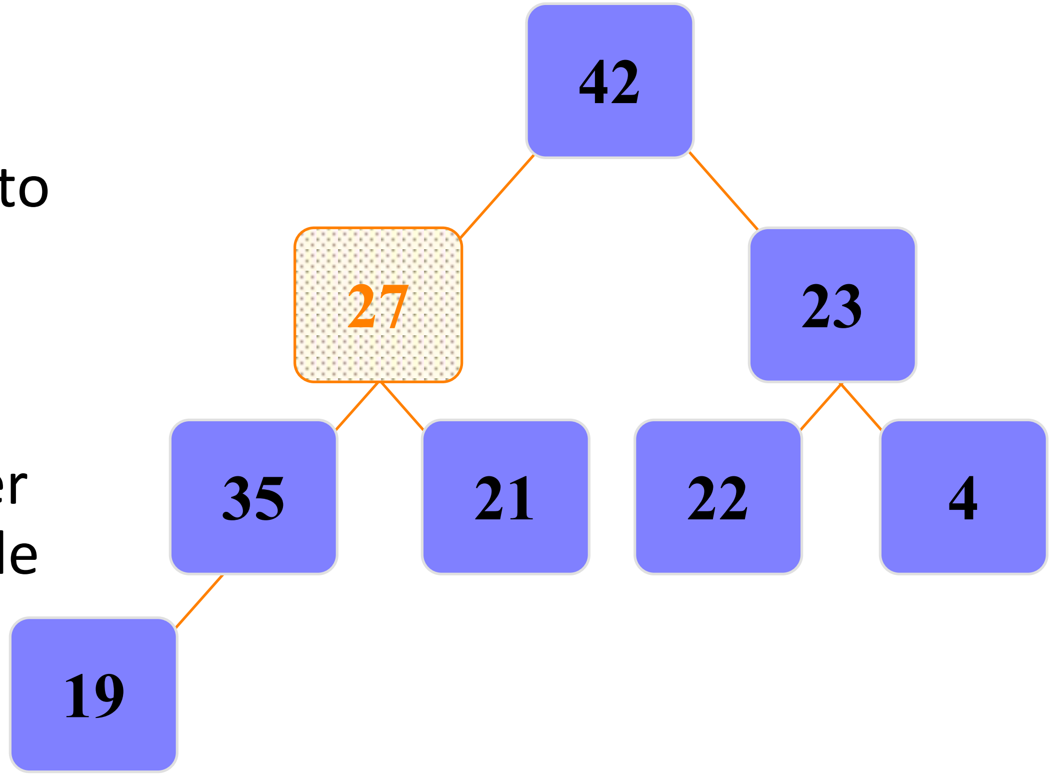
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



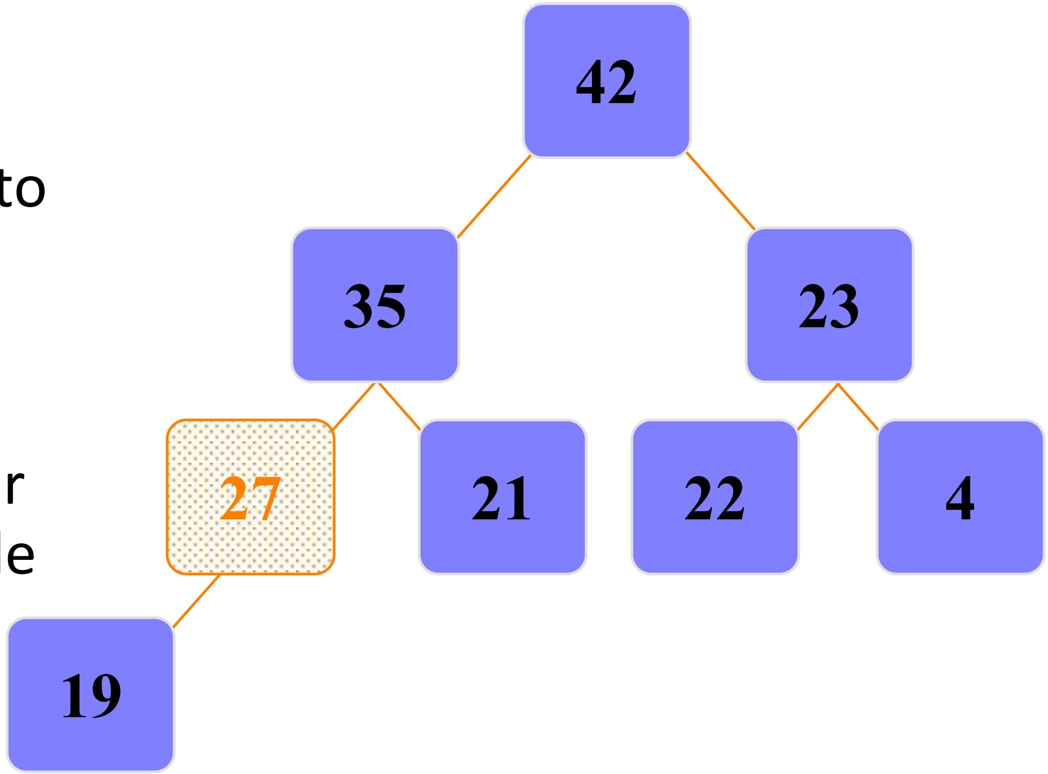
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



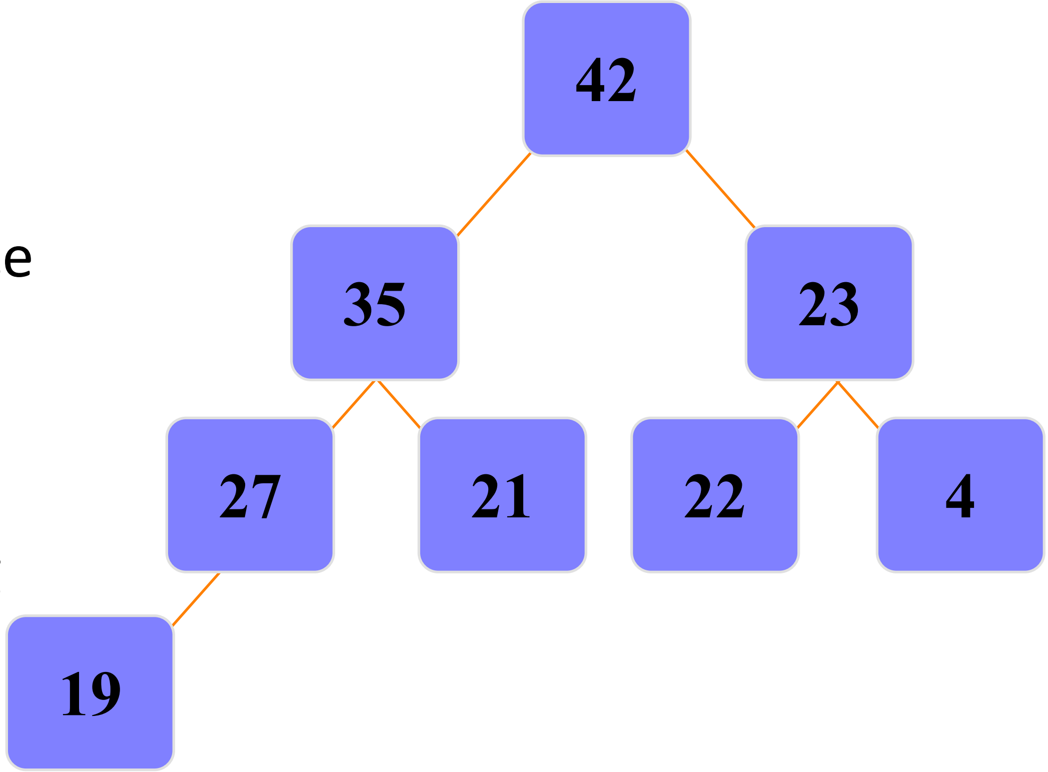
Removing the Top of a Heap

- ❑ Move the last node onto the root.
- ❑ Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



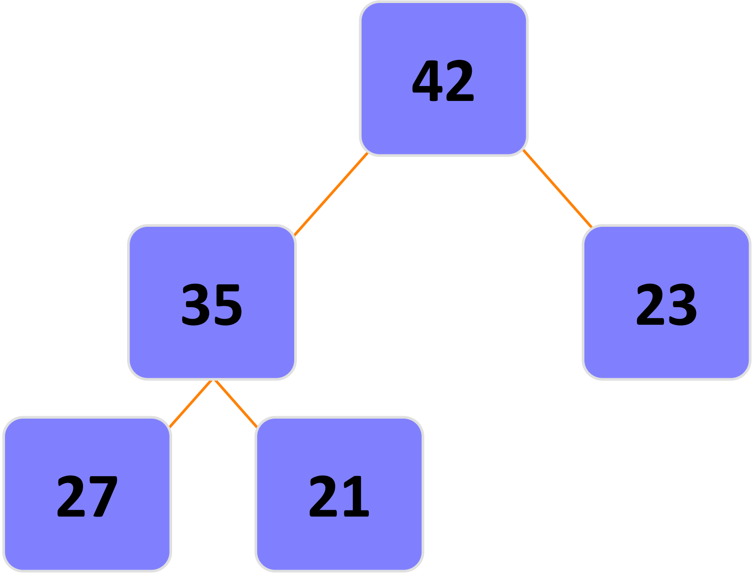
Removing the Top of a Heap

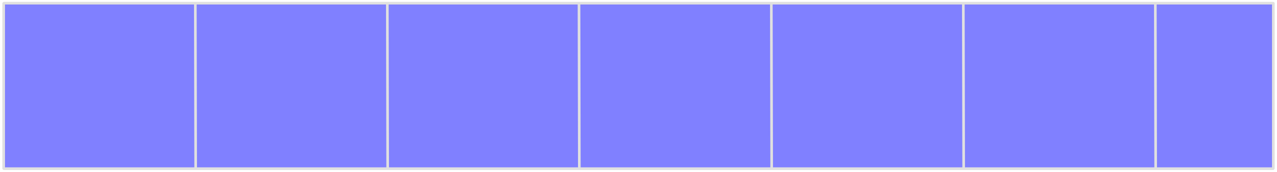
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

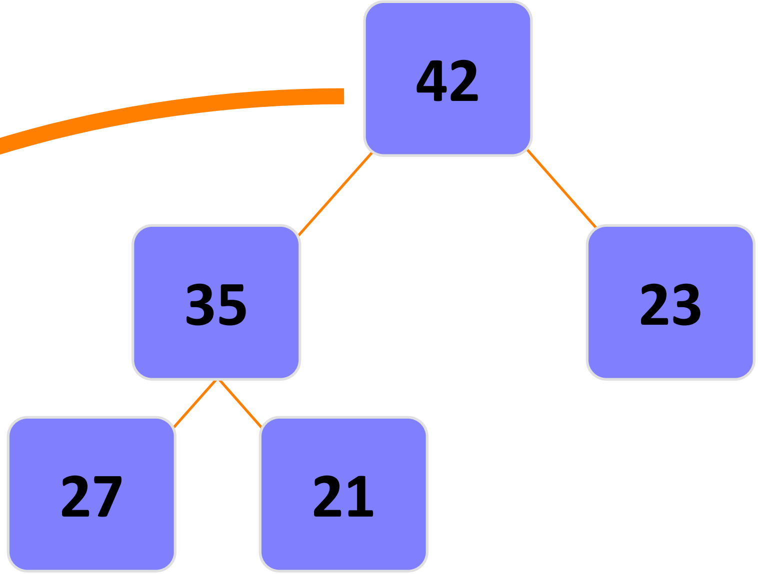
- We will store the data from the nodes in a partially-filled array.



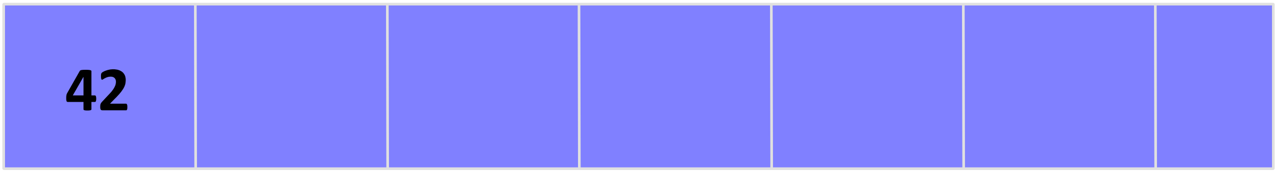


An array of data

Implementing a Heap



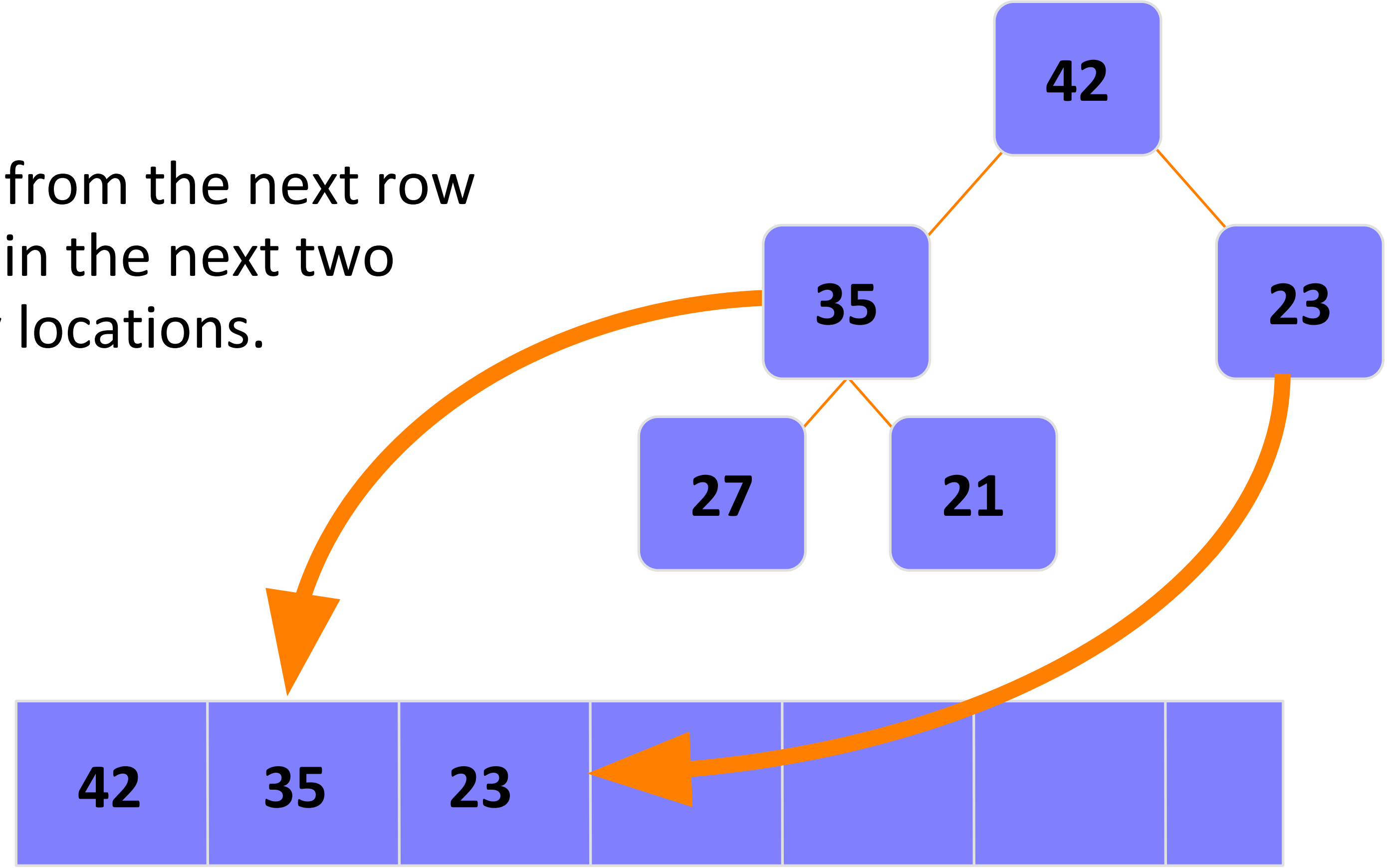
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

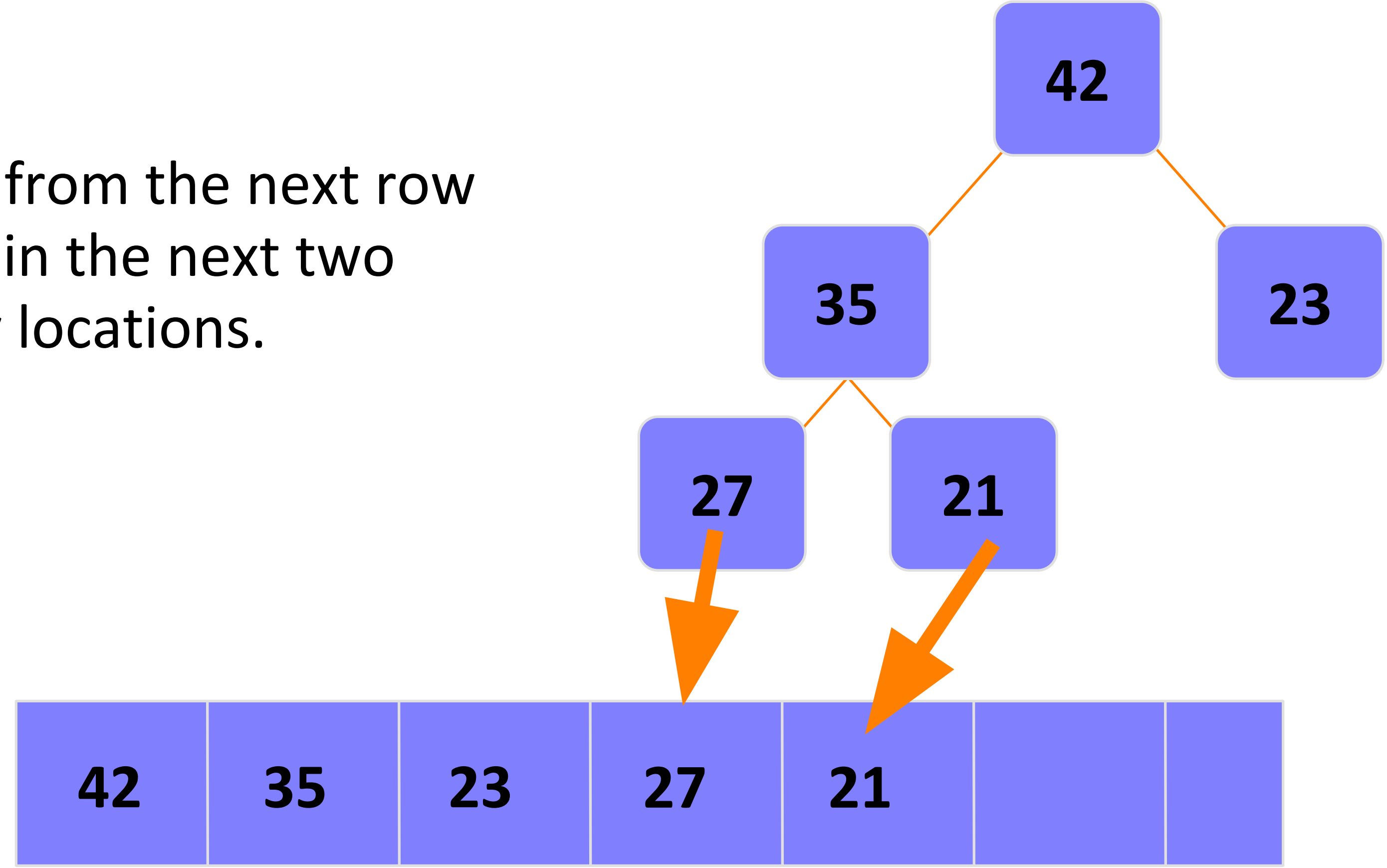
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

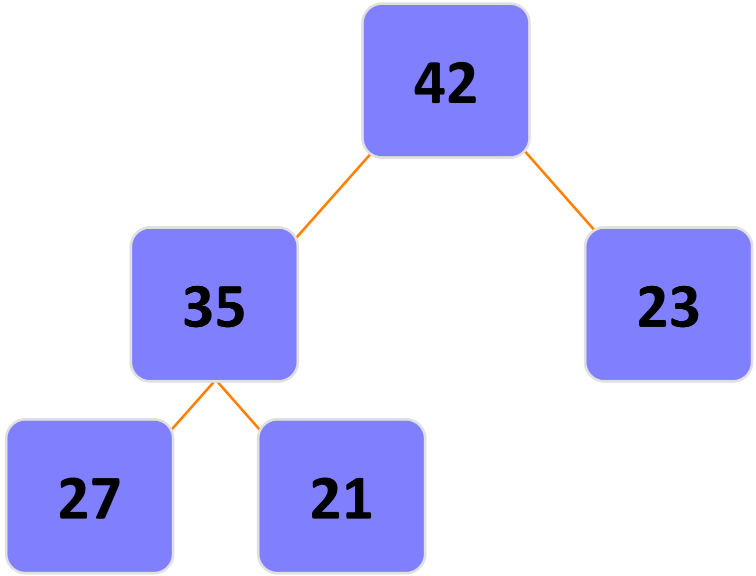
- Data from the next row goes in the next two array locations.

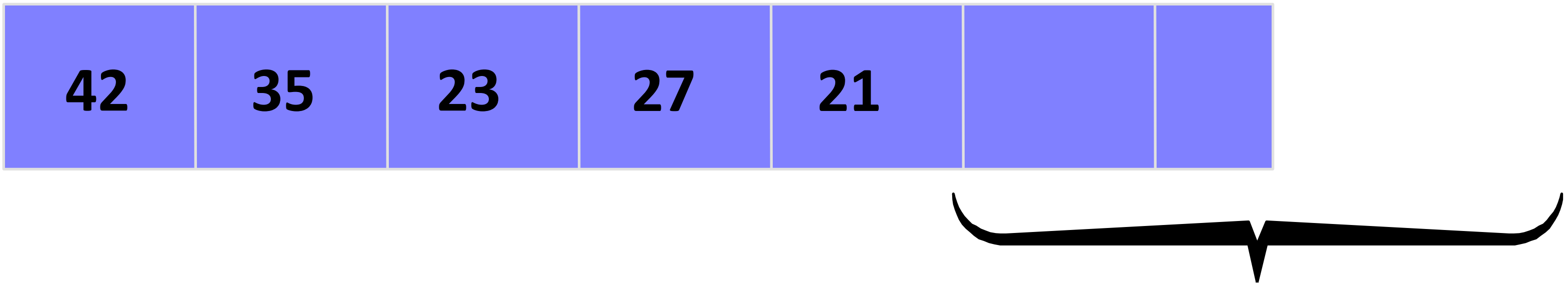


An array of data

Implementing a Heap

- Data from the next row goes in the next two array locations.

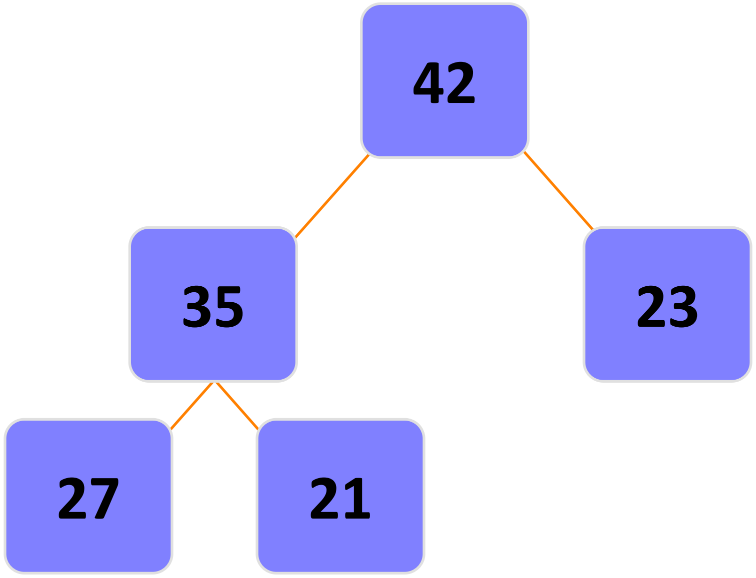




We don't care what's in this part of the array.

Important Points about the Implementation

- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

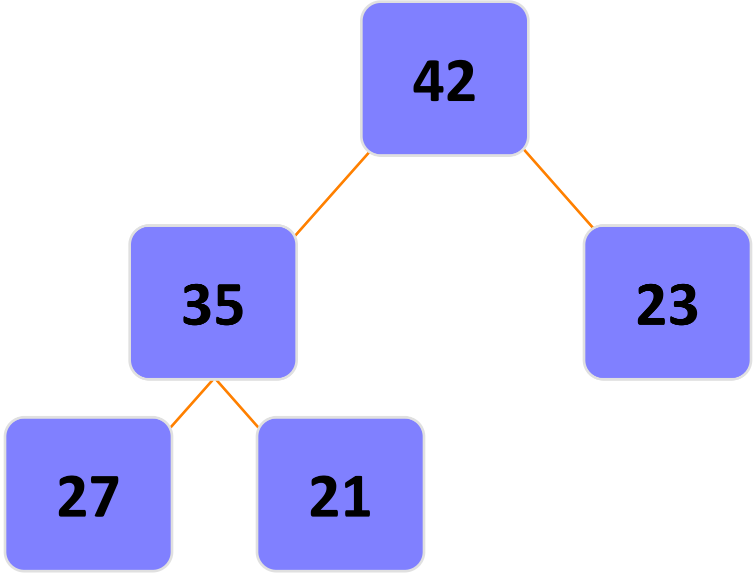


42	35	23	27	21		

An array of data

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given



Programming representation:

Parent: (i-1)/2

Left: 2*i + 1

Right: 2*i + 2

42	**35**	**23**	**27**	**21**		
[1]	[2]	[3]	[4]	[5]		

Logical representation:

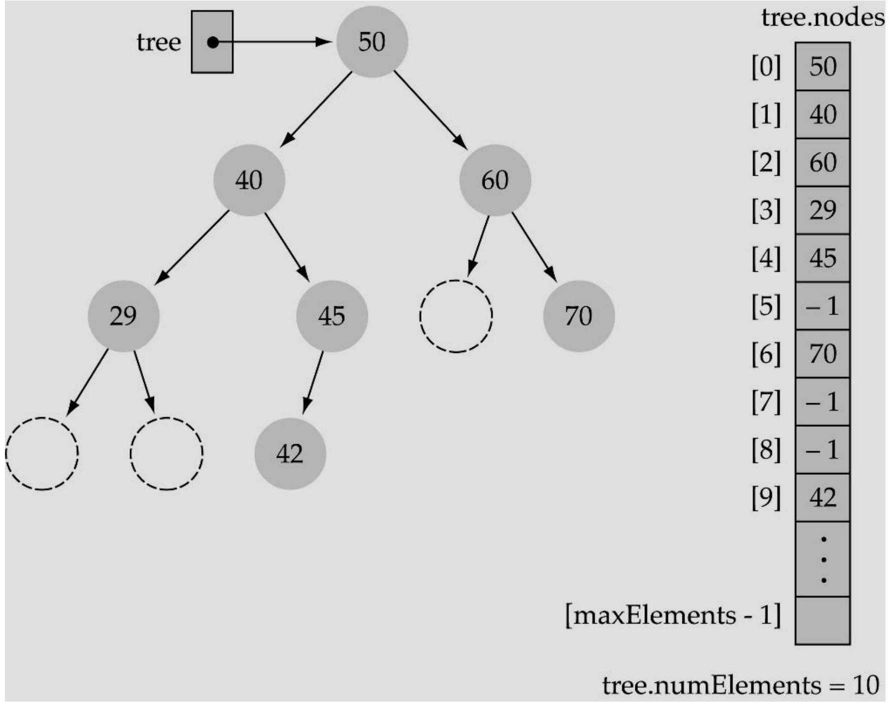
Parent: (i)/2

Left: 2*i

Right: 2*i + 1

Array-based representation of binary trees (cont.)

- Full or complete trees can be implemented easily using an array-based representation (elements occupy contiguous array slots)
- "Dummy nodes" are required for trees which are not full or complete



Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

Implementation in C++

```
#include <iostream>
using namespace std;
#define MAX 5
int heap_size=0;
int harr[MAX];
int parent(int i) { return (i-1)/2; }
// to get index of left child of node at index i
int left(int i) { return (2*i + 1); }
// to get index of right child of node at index i
int right(int i) { return (2*i + 2); }
```

Implementation in C++

```
// Inserts a new key 'k'
void insertKey_min(int k)
{
if (heap_size == MAX)
{
cout << "\nOverflow: Could not insertKey\n";
return;
}
// First insert the new key at the end
heap_size++;
int i = heap_size - 1;
harr[i] = k;
// Fix the min heap property if it is violated
while (i != 0 && harr[parent(i)] > harr[i])
{
int temp = harr[i];
harr[i] = harr[parent(i)];
harr[parent(i)] = temp;
i = parent(i);
}}
```

Implementation in C++

```
// Inserts a new key 'k'
void insertKey_max(int k)
{
if (heap_size == MAX)
{
cout << "\nOverflow: Could not insertKey\n";
return;}
// First insert the new key at the end
heap_size++;
int i = heap_size - 1;
harr[i] = k;
// Fix the max heap property if it is violated
while (i != 0 && harr[parent(i)] < harr[i])
{
int temp = harr[i];
harr[i] = harr[parent(i)];
harr[parent(i)] = temp;
i = parent(i);
}
}
```

Implementation in C++

```
// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeapify(int i)
{
int l = left(i);
int r = right(i);
int smallest = i;
if (l < heap_size && harr[l] < harr[i])
smallest = l;
if (r < heap_size && harr[r] < harr[smallest])
smallest = r;
if (smallest != i)
{
int temp = harr[i];
harr[i] = harr[smallest];
harr[smallest] = temp;
MinHeapify(smallest);}}
```

Implementation in C++

```
void MaxHeapify(int i)
{
int l = left(i);
int r = right(i);
int largest = i;
if (l < heap_size && harr[l] > harr[i])
largest = l;
if (r < heap_size && harr[r] > harr[largest])
largest = r;
if (largest != i)
{
int temp = harr[i];
harr[i] = harr[largest];
harr[largest] = temp;
MaxHeapify(largest);
}
}
```

Implementation in C++

```
int delete_key()
{
if (heap_size <= 0)
return 0;
if (heap_size == 1)
{
heap_size--;
return harr[0];
}
// Store the minimum value, and remove it from heap
int root = harr[0];
harr[0] = harr[heap_size-1];
heap_size--;
// MinHeapify(0);
MaxHeapify(0);
return root;
}
```

Implementation in C++

```
void display(){
for(int i=0 ; i<heap_size ; i++){
cout<<" "<<harr[i]<<" ,";
}
cout<<"\b \b";
}
```

Implementation in C++

```
int main() {
/*insertKey_min(3);
insertKey_min(1);
insertKey_min(2);
insertKey_min(15);
insertKey_min(5);*/
insertKey_max(3);
insertKey_max(2);
insertKey_max(1);
insertKey_max(15);
insertKey_max(5);
cout<<"Inserted!!";
```

Implementation in C++

```
display();
cout<<endl;
int temp = delete_key();
if(temp == 0)
{
cout<<"\nHeap is Empty!!"<<endl;
}else{
cout<<temp<<" Deleted!!"<<endl;
}
```

```
display();
return 0;
}
```

Max-Heapify

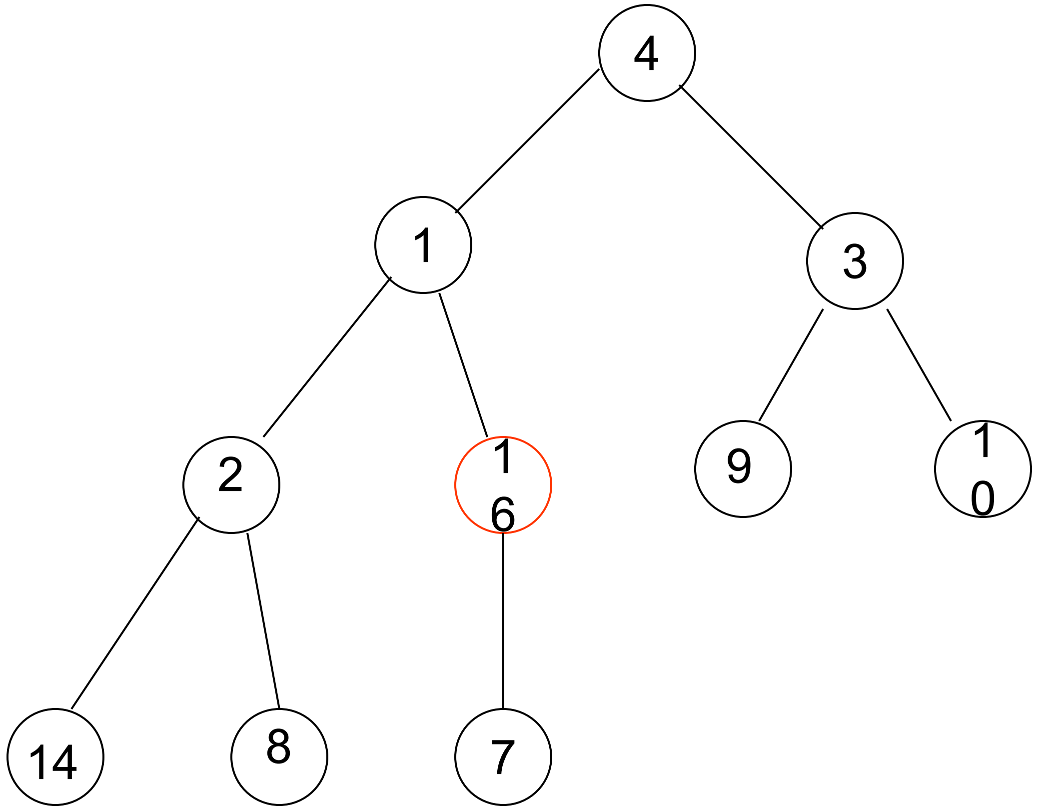
- Max-Heapify(A,i) is a subroutine.
- When it is called, two subtrees rooted at Left(i) and Right(i) are max-heaps, but A[i] may not satisfy the max-heap property.
- Max-Heapify(A,i) makes the subtree rooted at A[i] become a max-heap by letting A[i] “float down”.

Building a Max-Heap

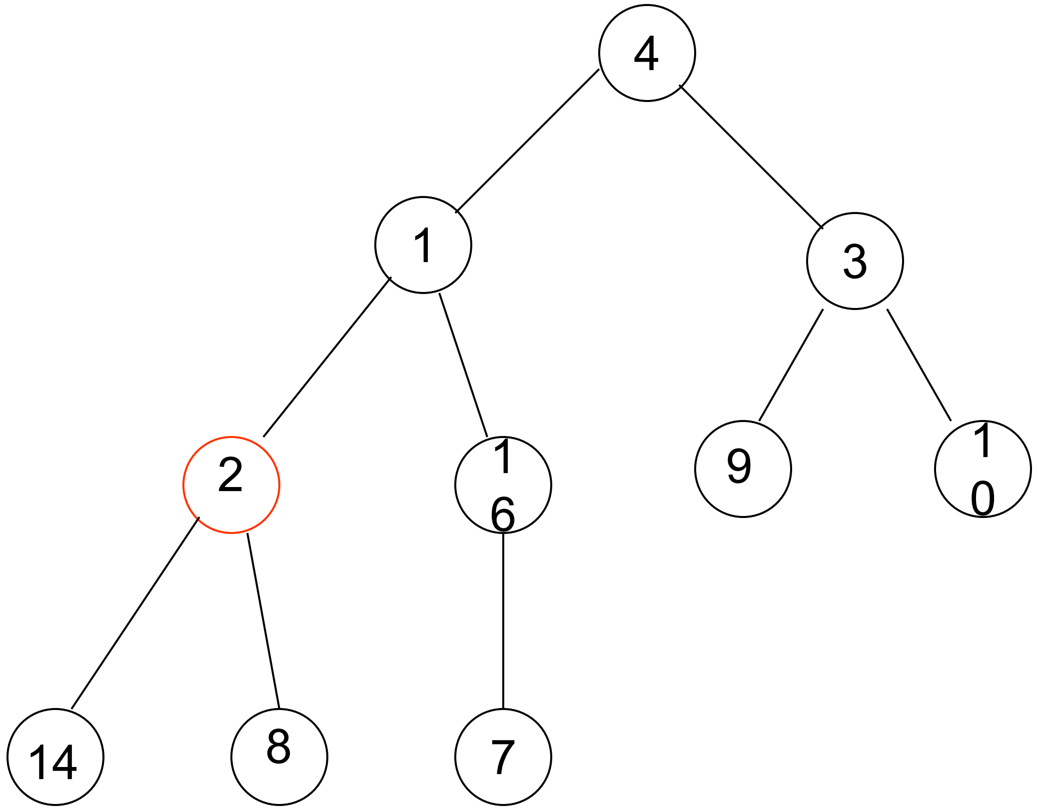
$$\text{Build-Max-Heap}(A)$$
$$heap\text{-}size[A] \leftarrow length[A];$$
$$\text{for } i \leftarrow \lfloor length[A]/2 \rfloor \text{ downto } 1$$
$$\quad \text{do Max-Heapify}(A, i);$$

e.g., 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

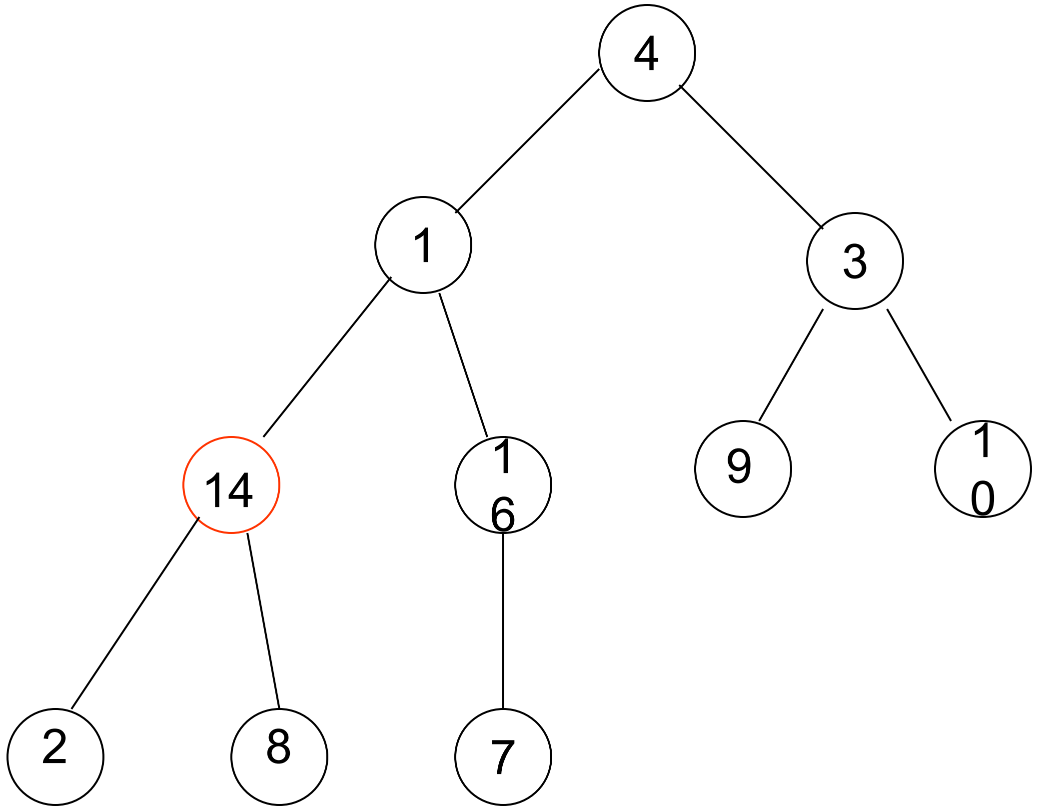
Adding a Node to a Heap



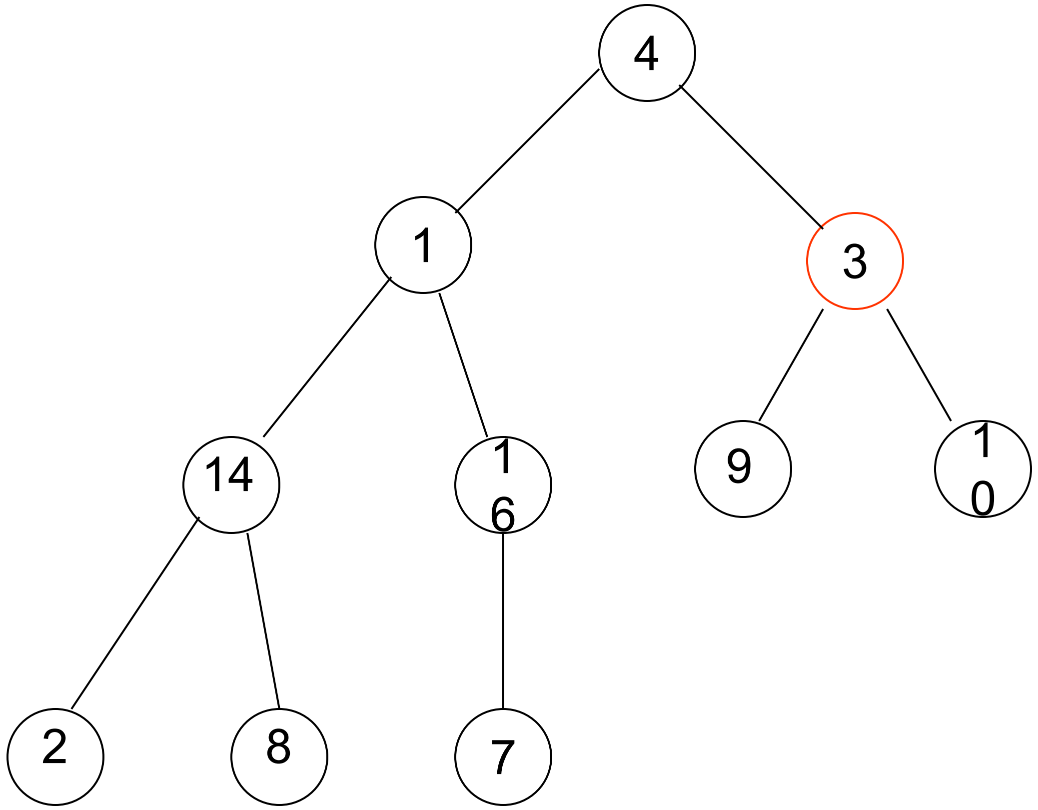
Adding a Node to a Heap



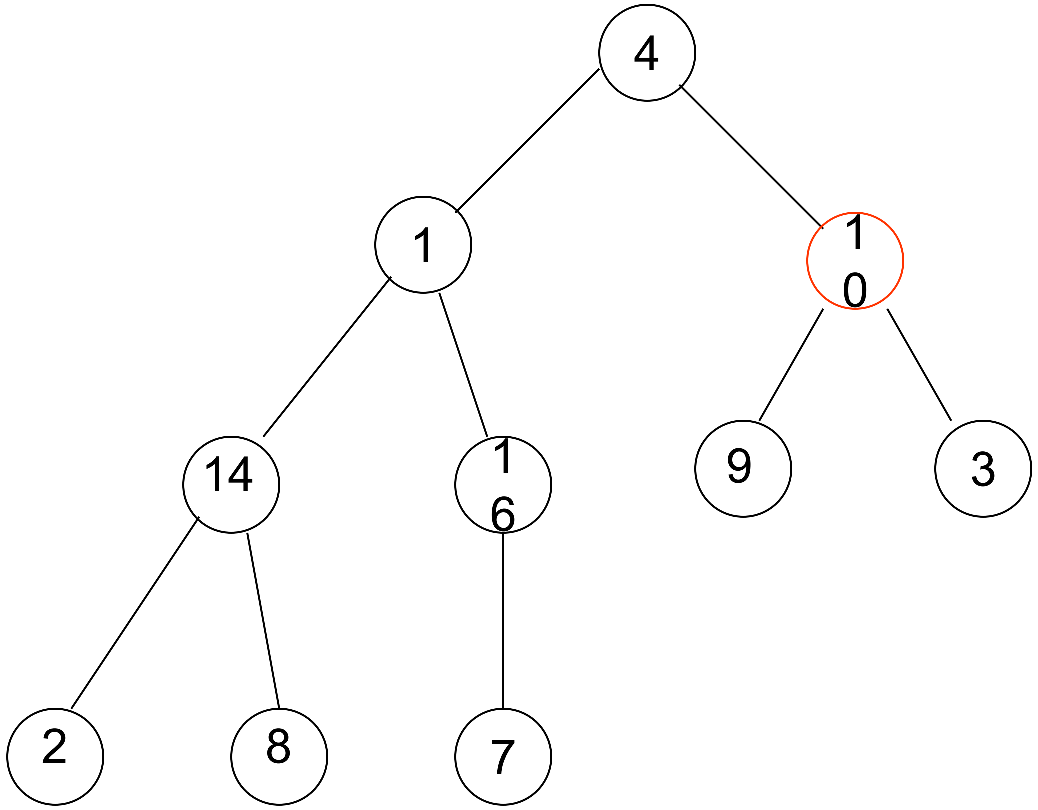
Adding a Node to a Heap



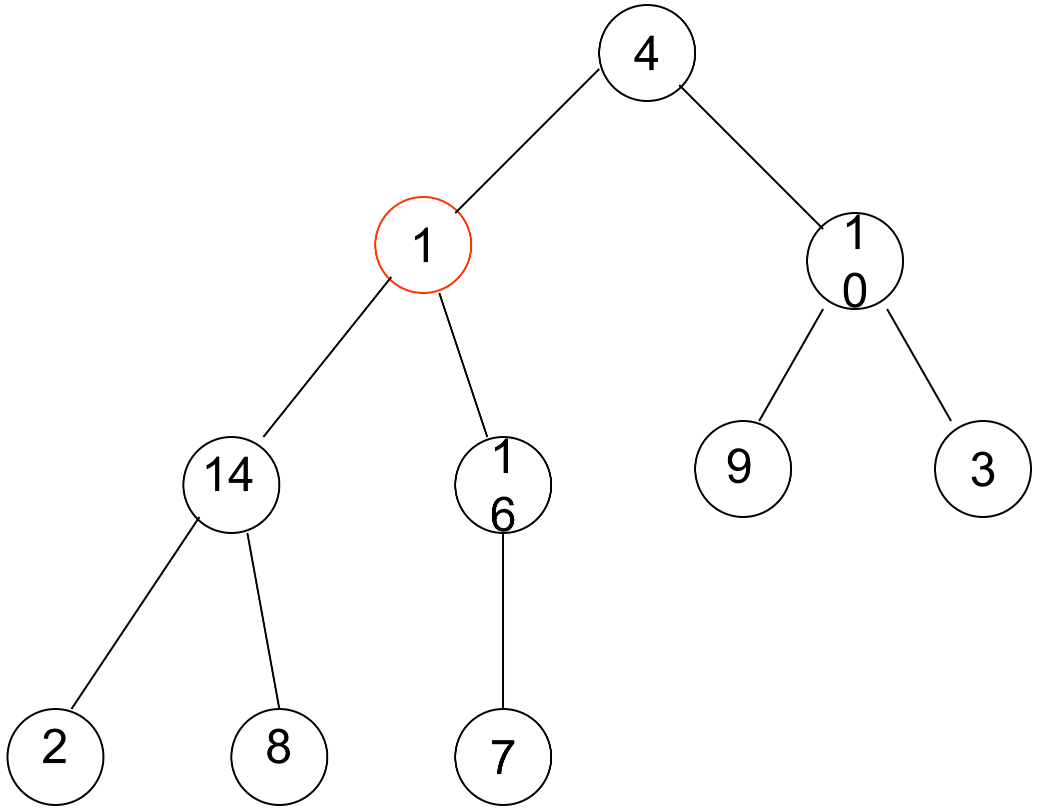
Adding a Node to a Heap



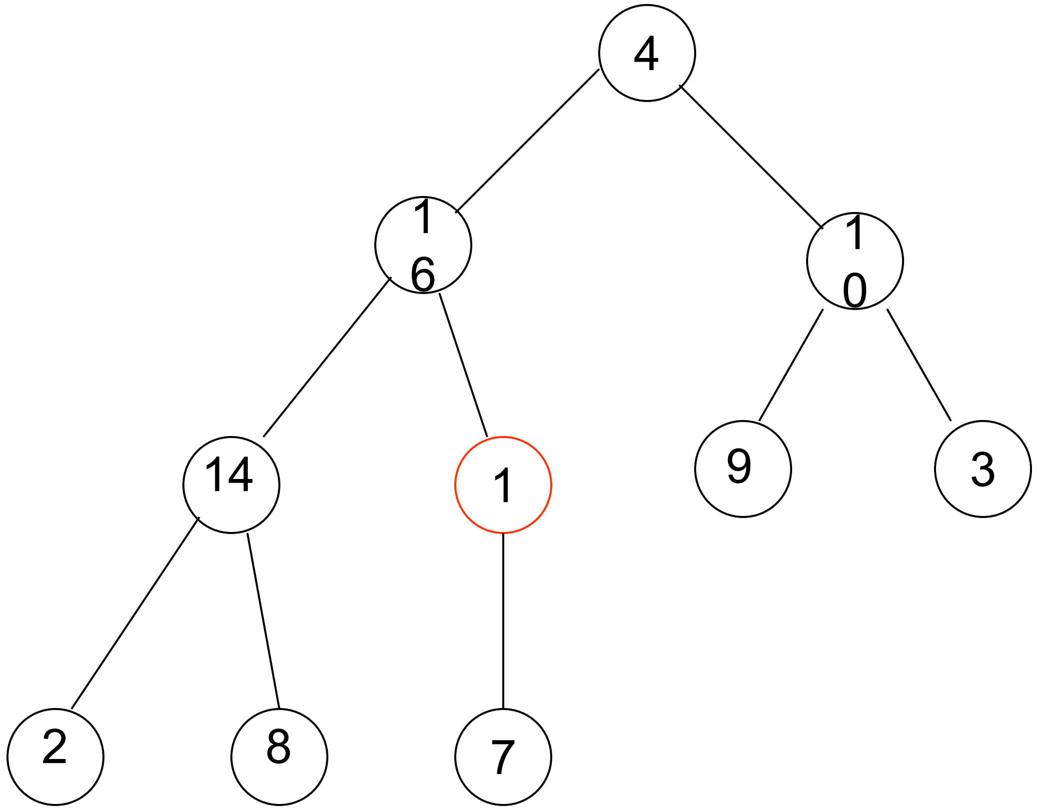
Adding a Node to a Heap



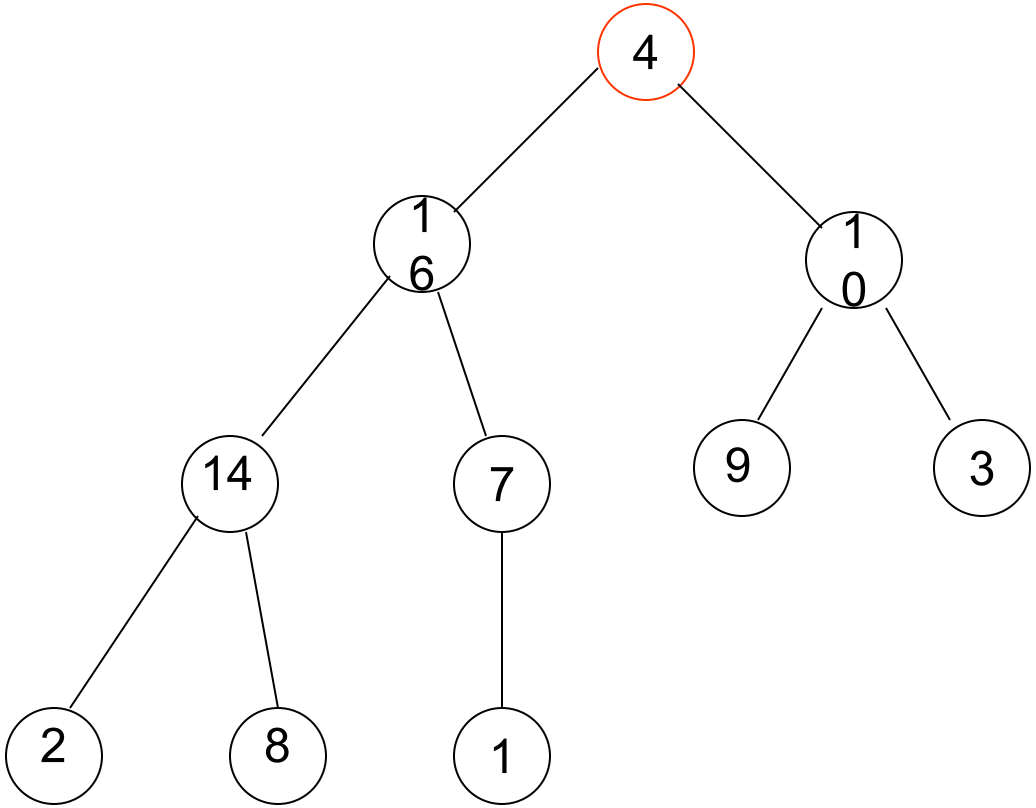
Adding a Node to a Heap



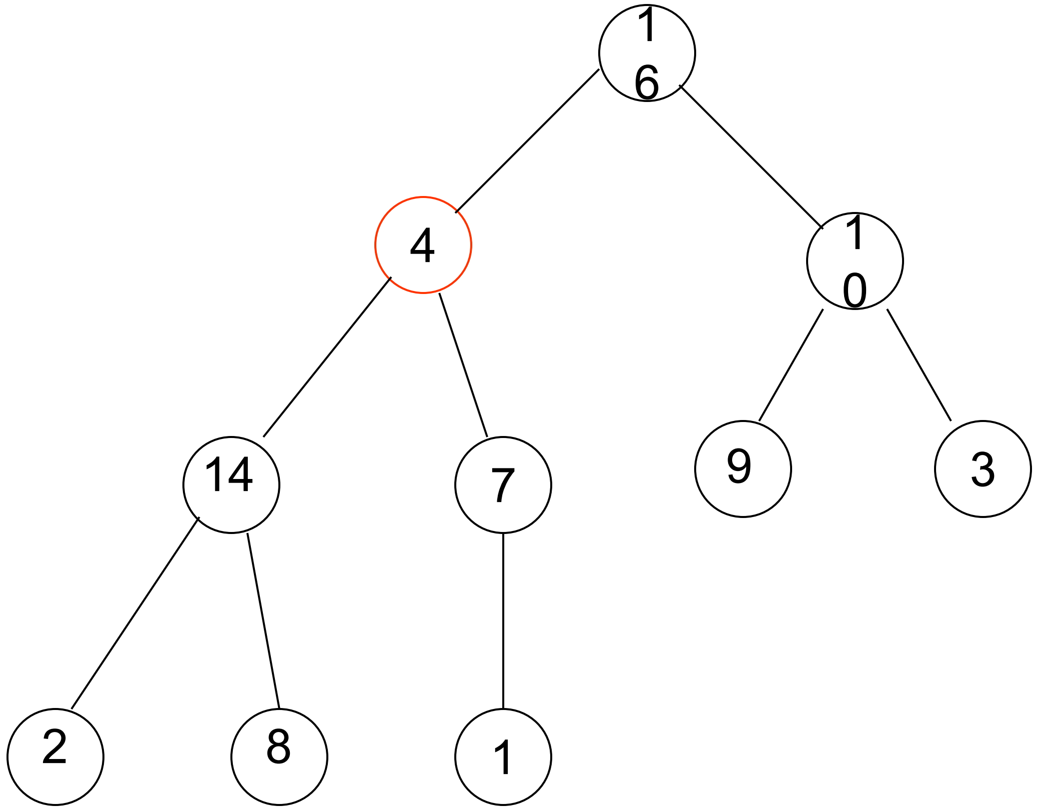
Adding a Node to a Heap



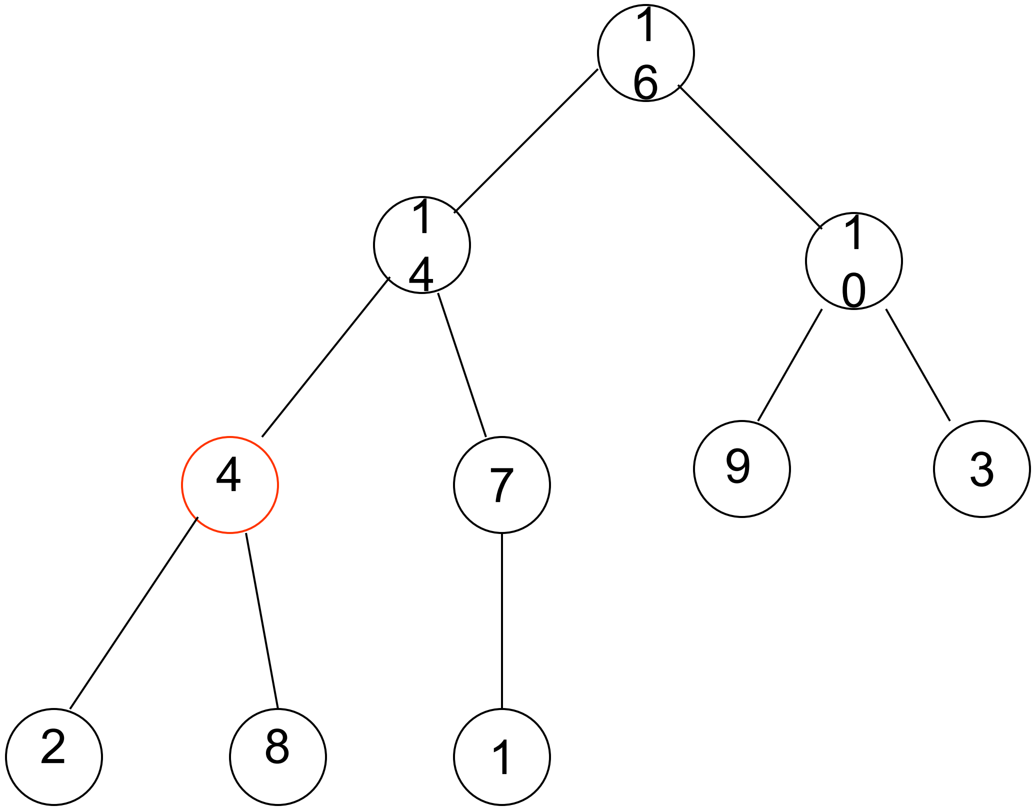
Adding a Node to a Heap



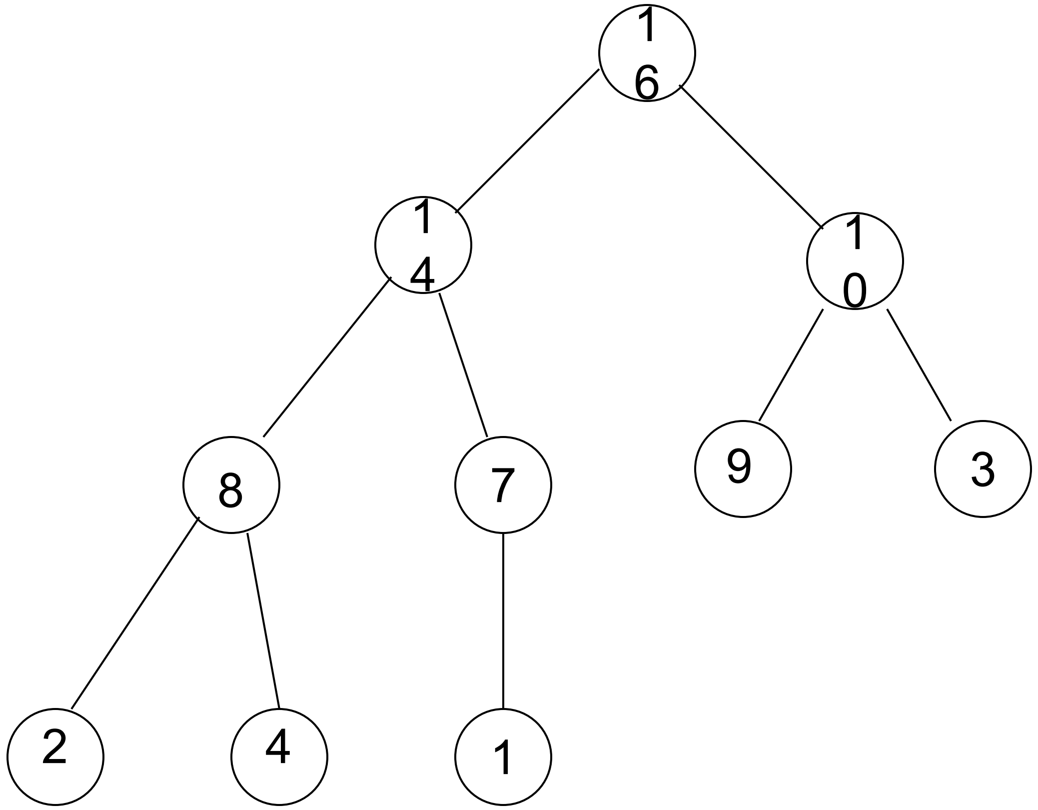
Adding a Node to a Heap



Adding a Node to a Heap



Adding a Node to a Heap

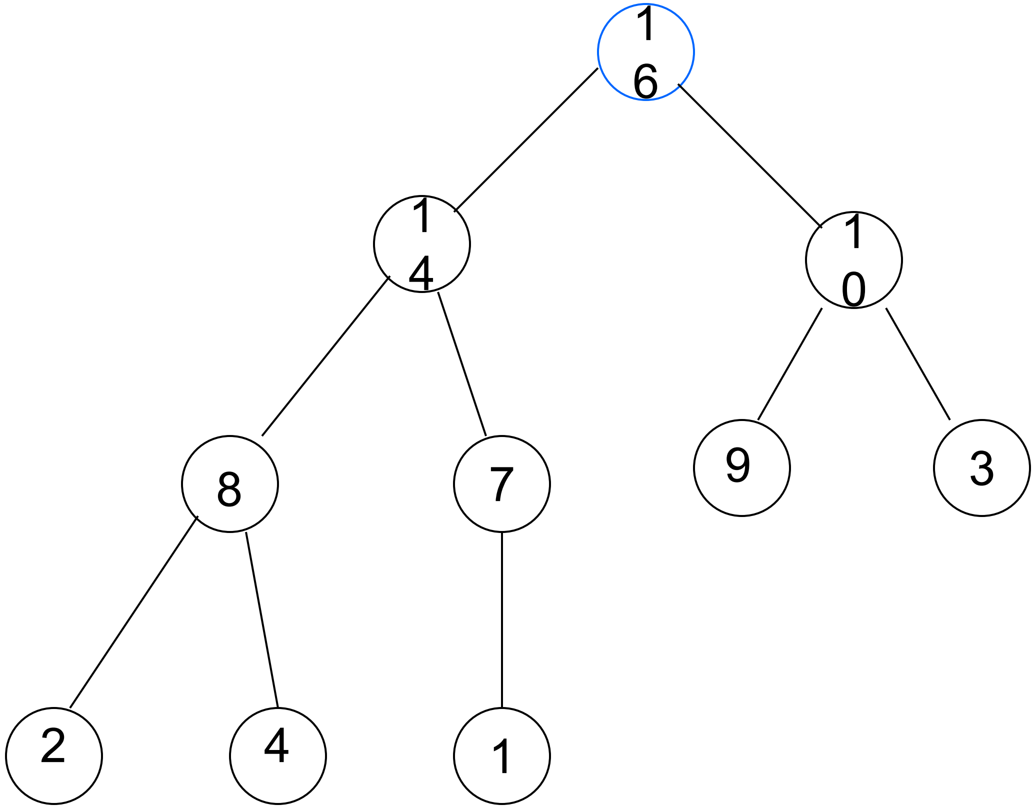


Heapsort

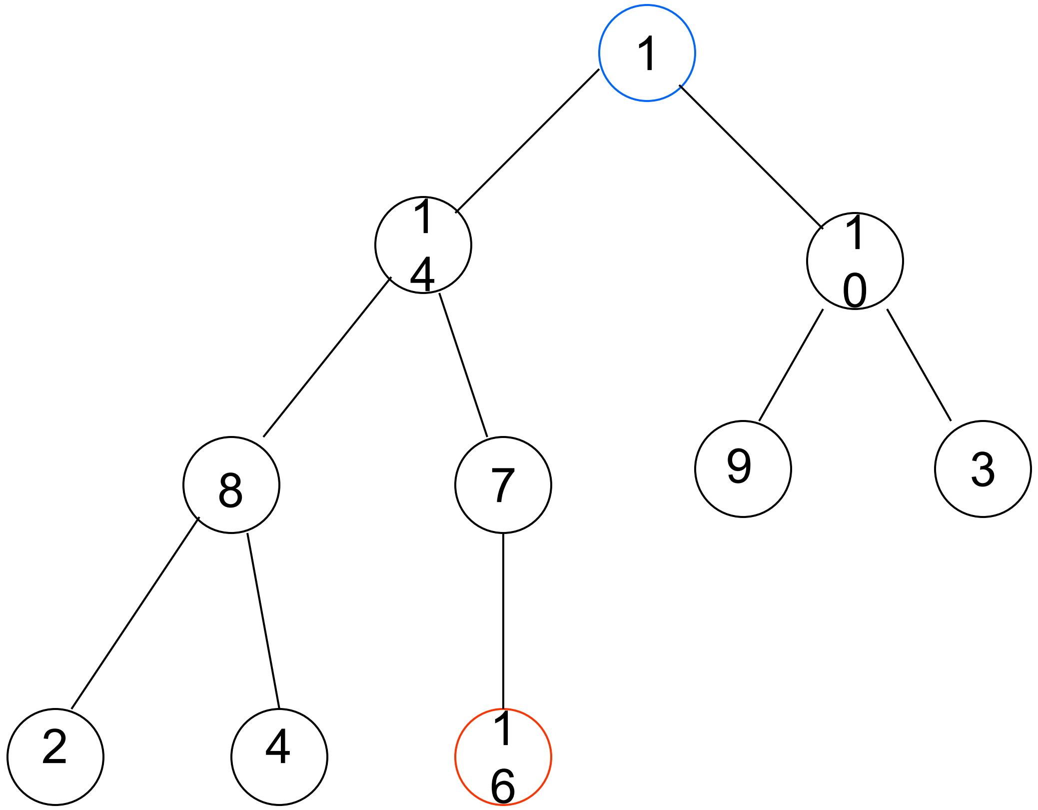
$$
\begin{array}{l}
\text{Heapsort}(A) \\
\quad \text{Buid - Max - Heap}(A); \\
\quad \text{for } i \leftarrow \mathit{length}[A] \ \text{ downto } 2 \\
\quad\quad \text{do begin} \\
\quad\quad\quad \text{exchange } \ A[1] \leftrightarrow A[i]; \\
\quad\quad\quad \mathit{heap}\text{ - }\mathit{size}[A] \leftarrow \mathit{heap}\text{ - }\mathit{size}[A] - 1; \\
\quad\quad\quad \text{Max - Heapify}(A,1); \\
\quad\quad \text{end - for}
\end{array}
$$

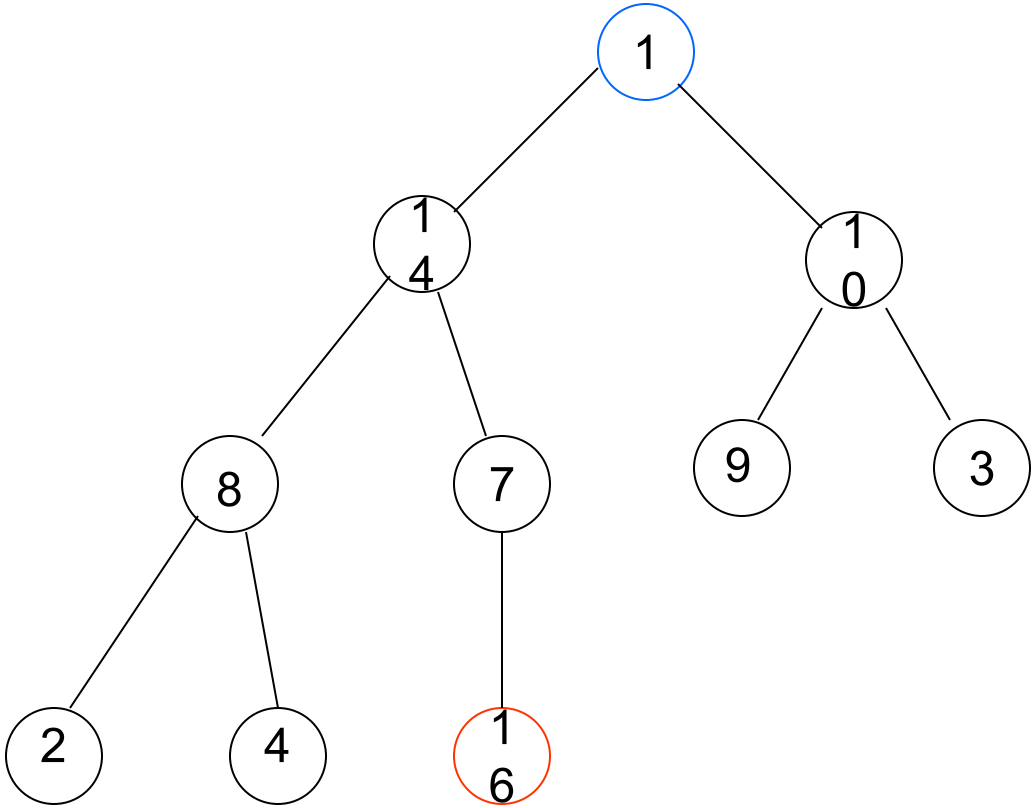
Input: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

Build a max-heap

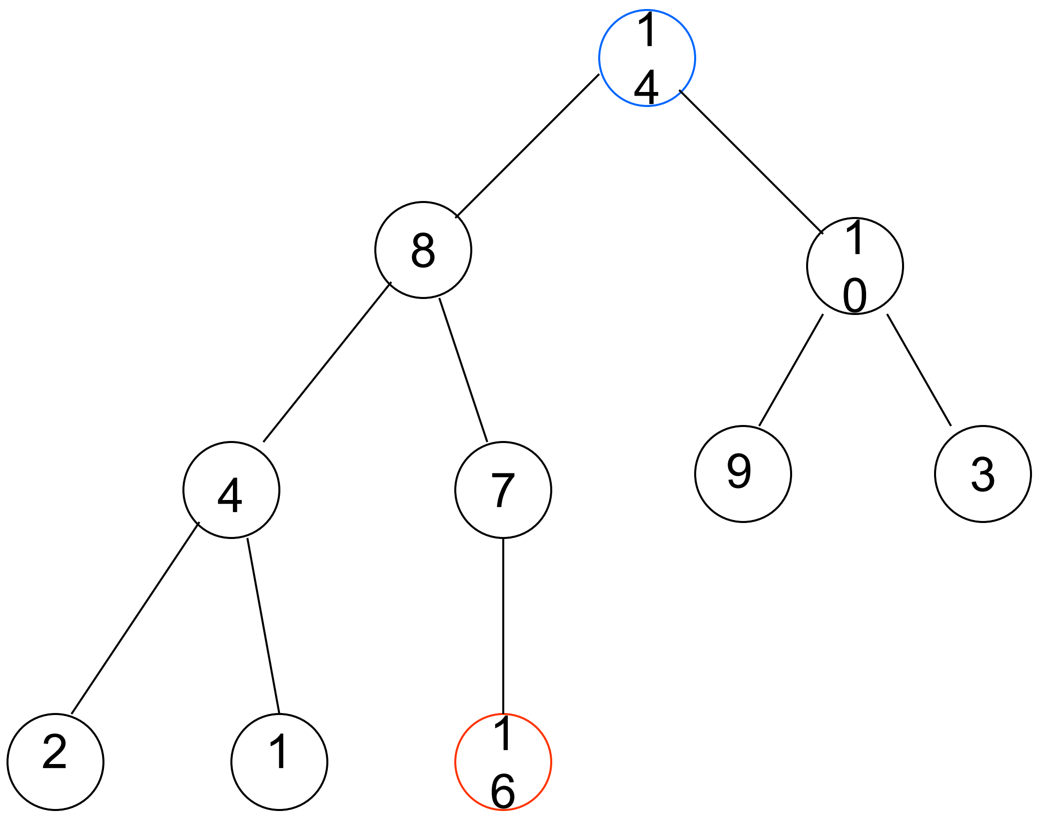


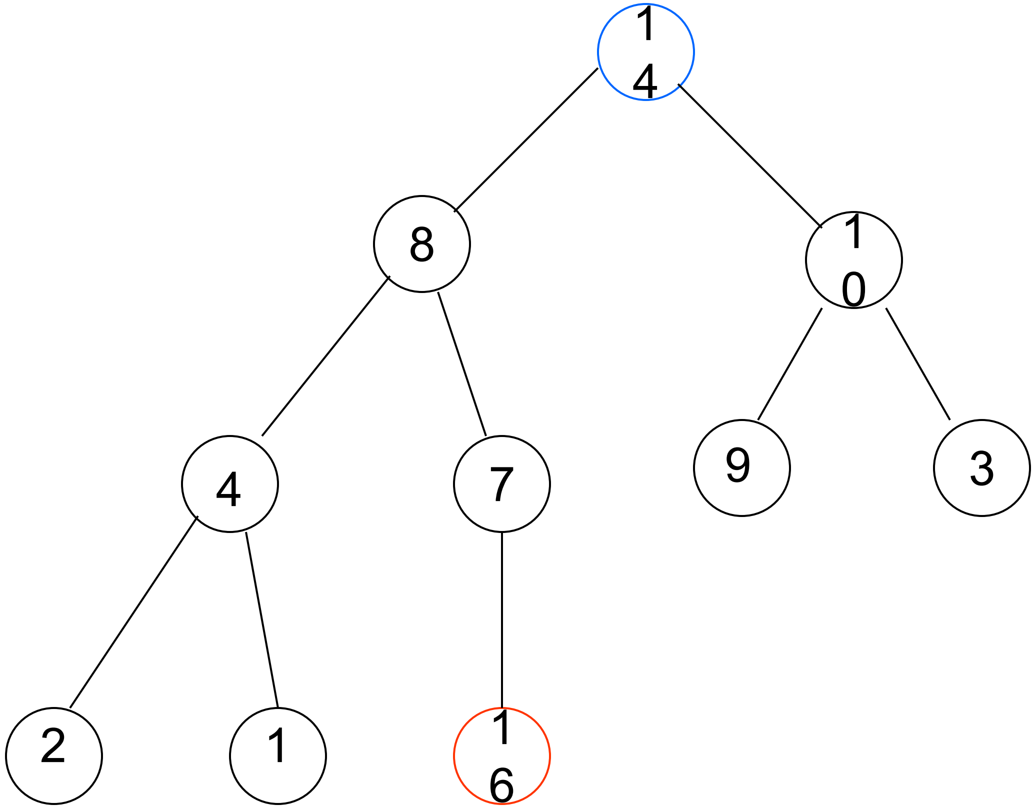
16, 14, 10, 8, 7, 9, 3, 2, 4, 1.


1
14
10
8
7
9
3
2
4
16

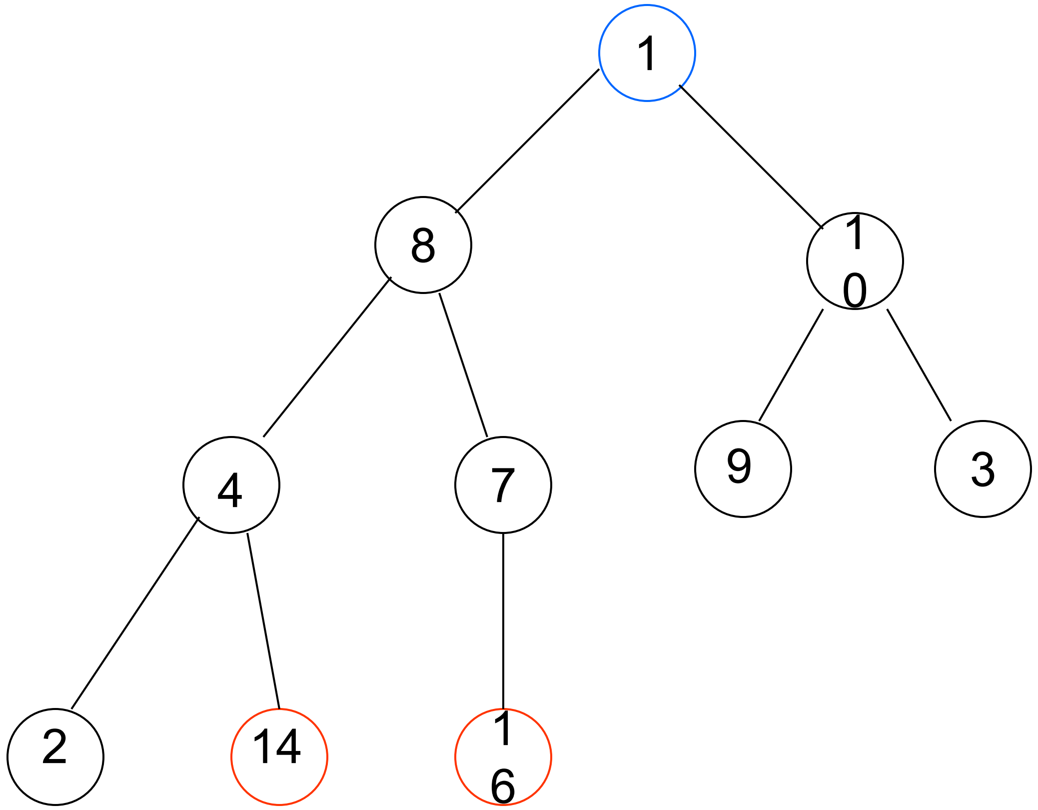


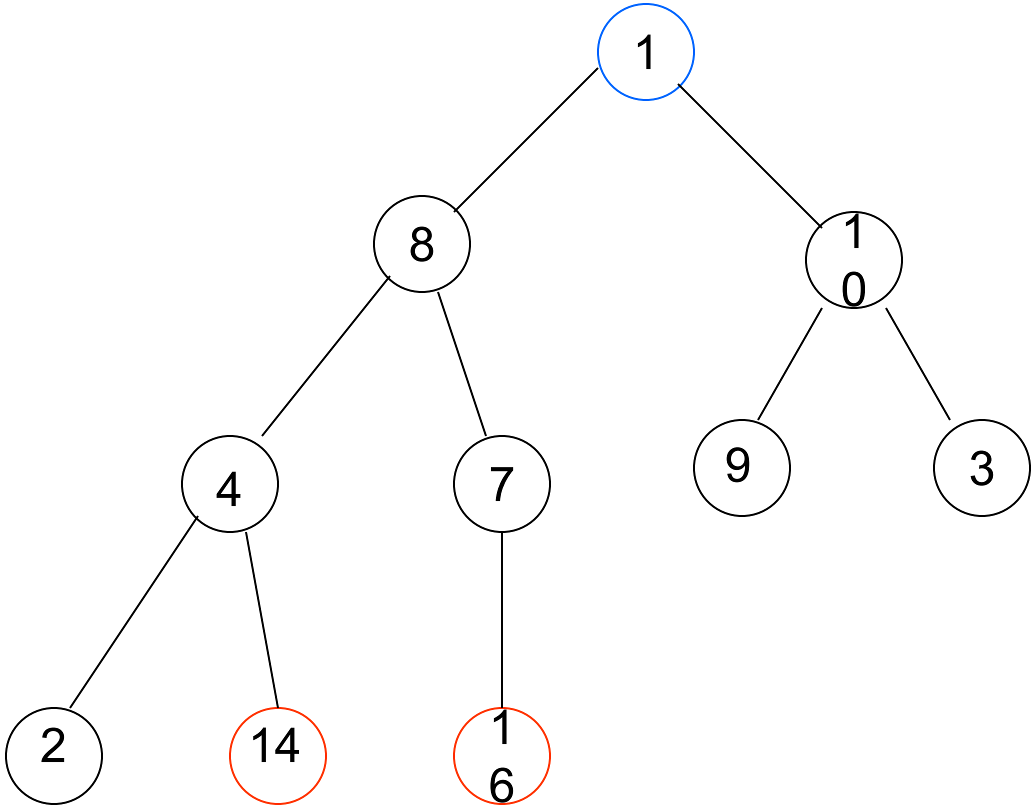
1, 14, 10, 8, 7, 9, 3, 2, 4, 16.


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

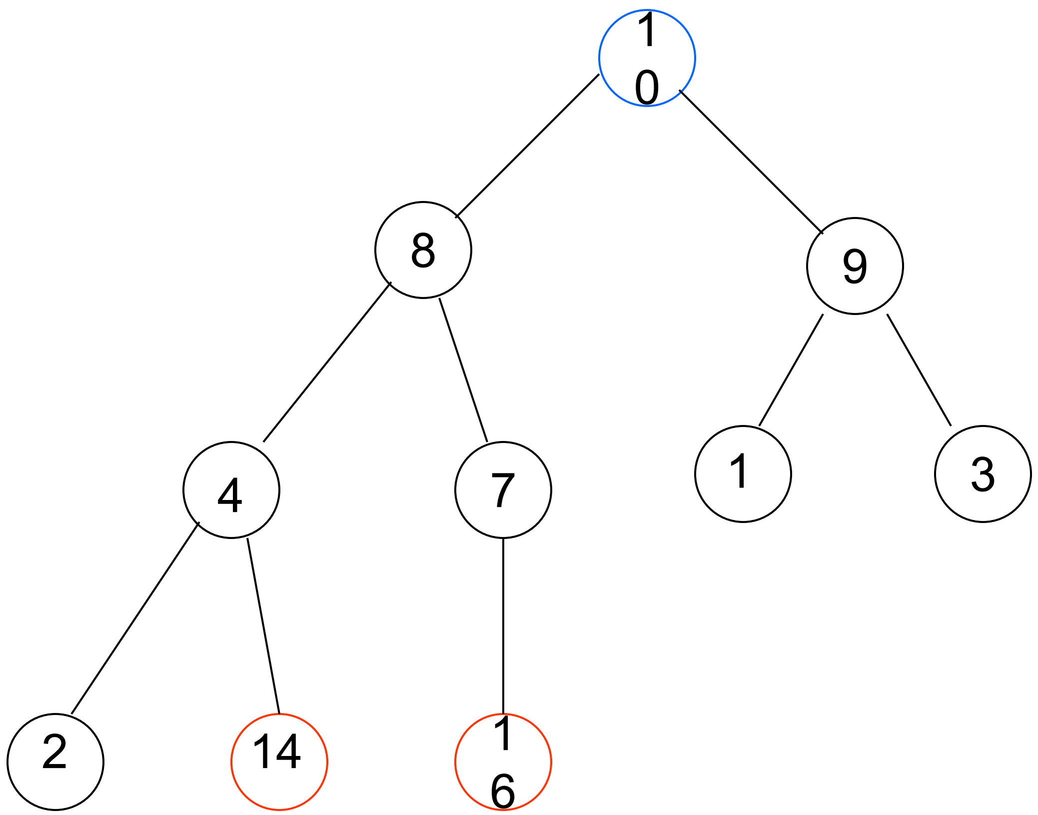


14, 8, 10, 4, 7, 9, 3, 2, 1, 16.

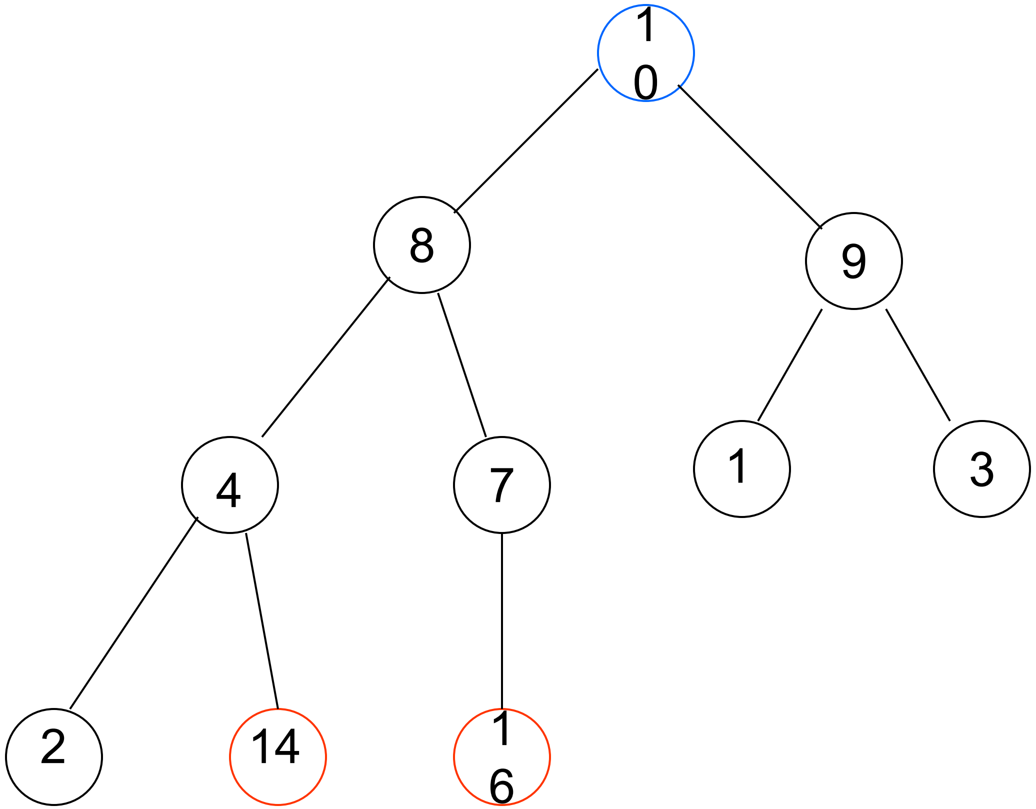

1
8
1
0
4
7
9
3
2
14
1
6



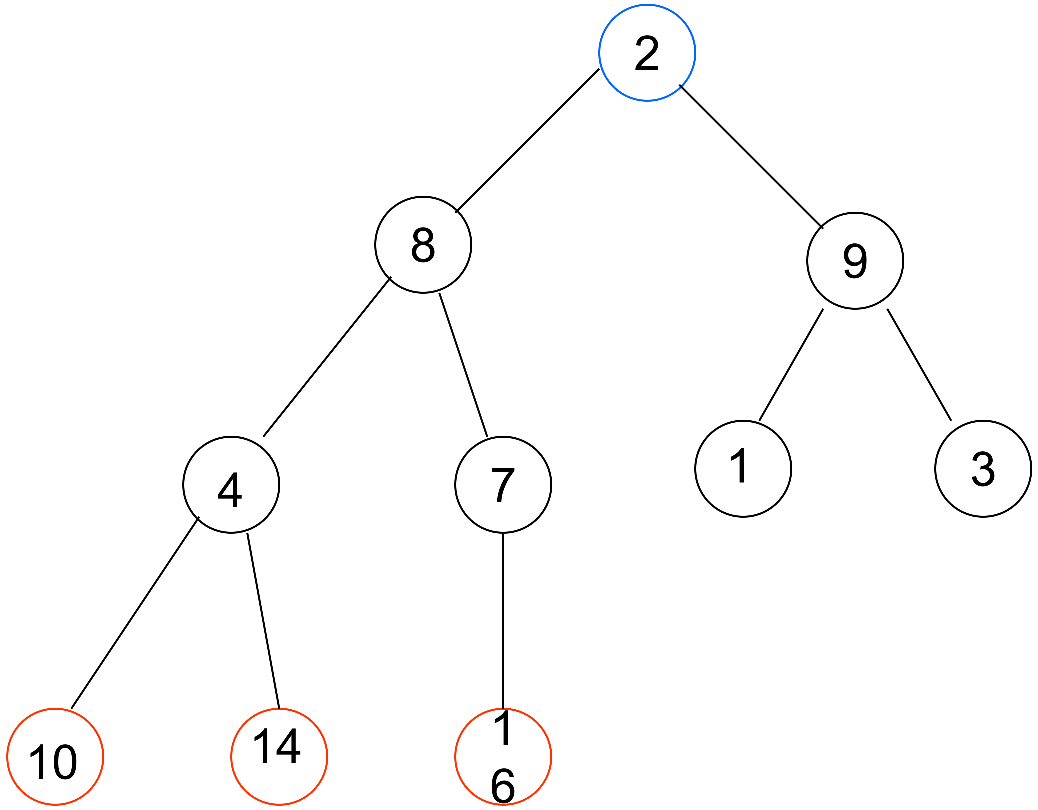
1, 8, 10, 4, 7, 9, 3, 2, 14, 16.

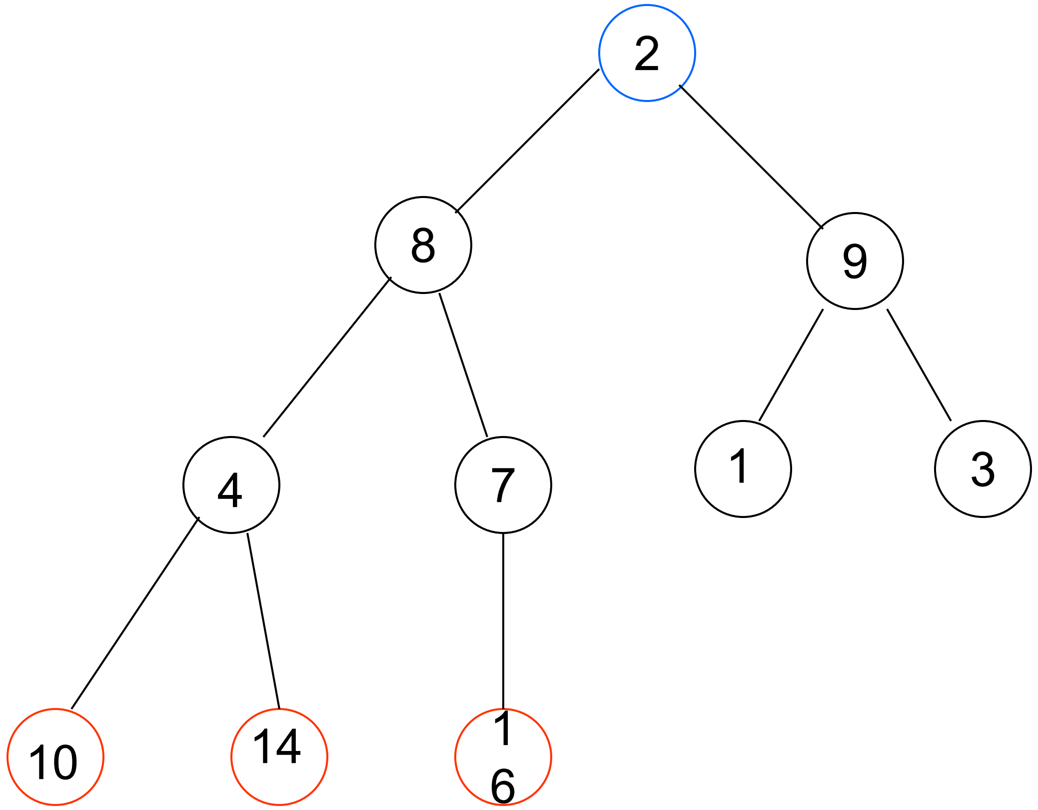


1
0
8
9
4
7
1
3
2
14
1
6

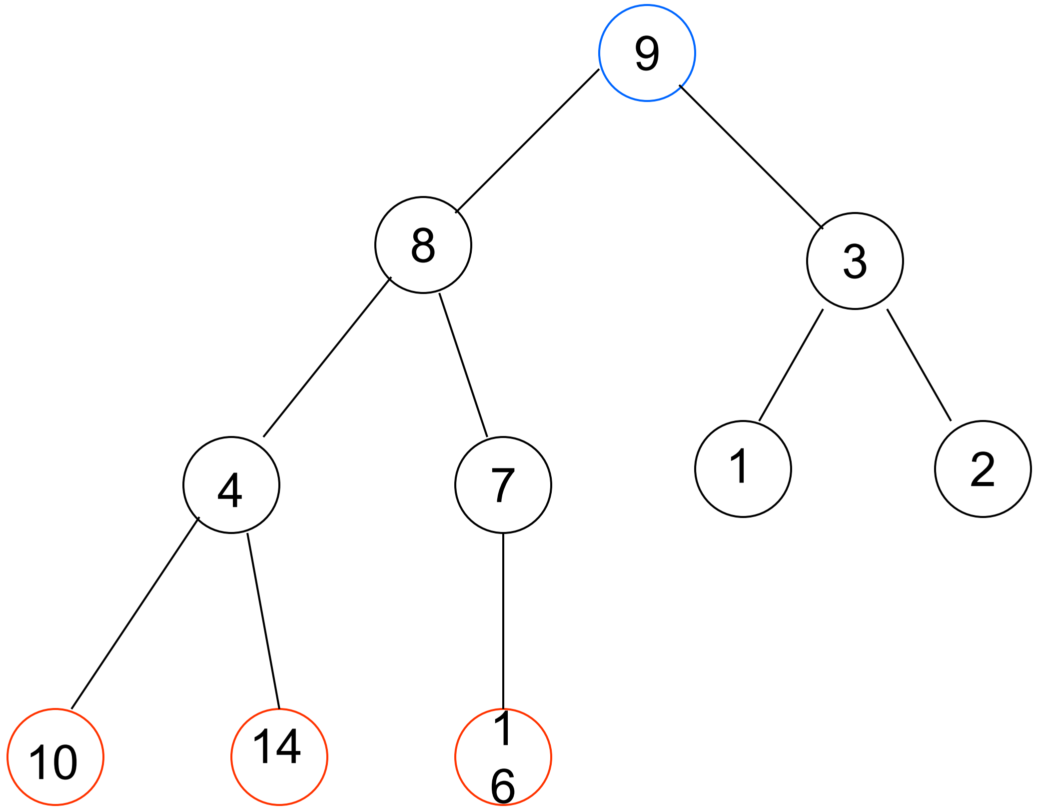


10, 8, 9, 4, 7, 1, 3, 2, 14, 16.

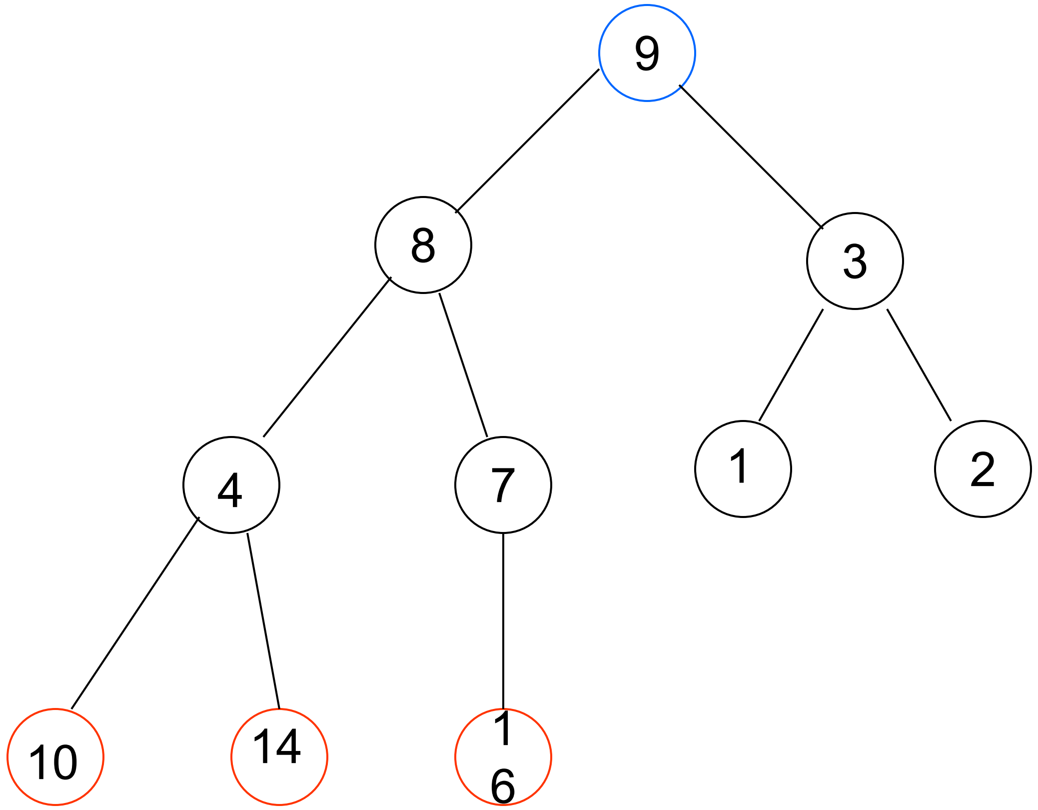

2
8
9
4
7
1
3
10
14
1
6



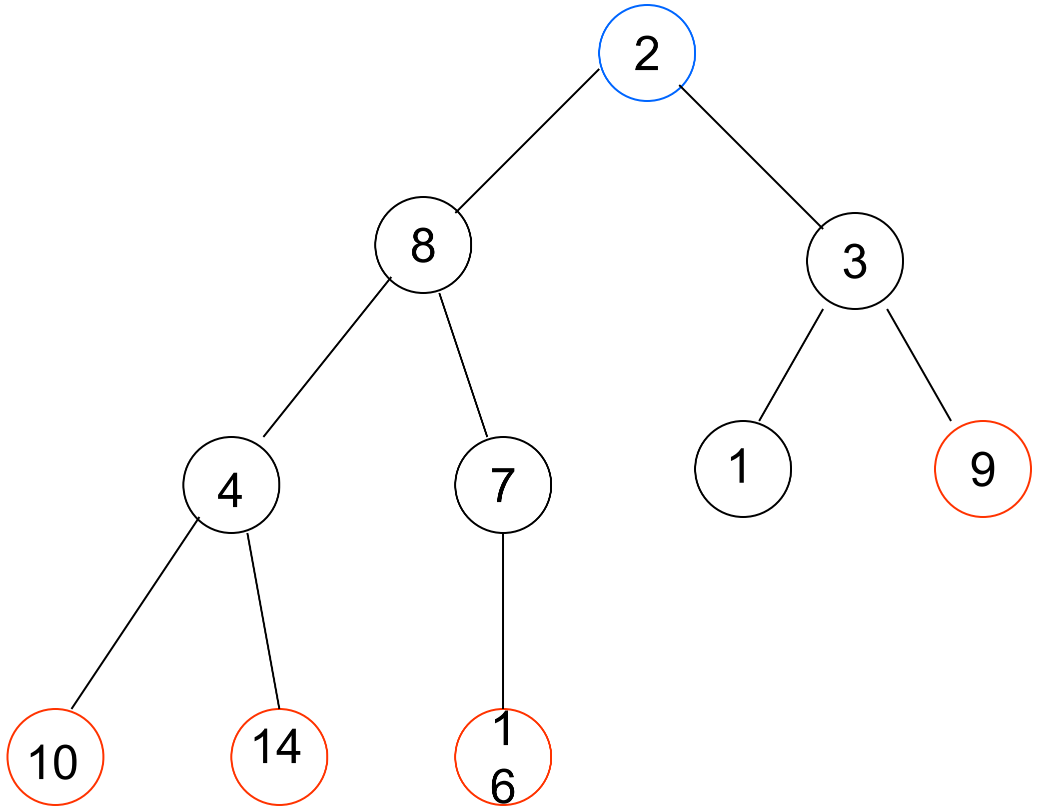
2, 8, 9, 4, 7, 1, 3, 10, 14, 16.



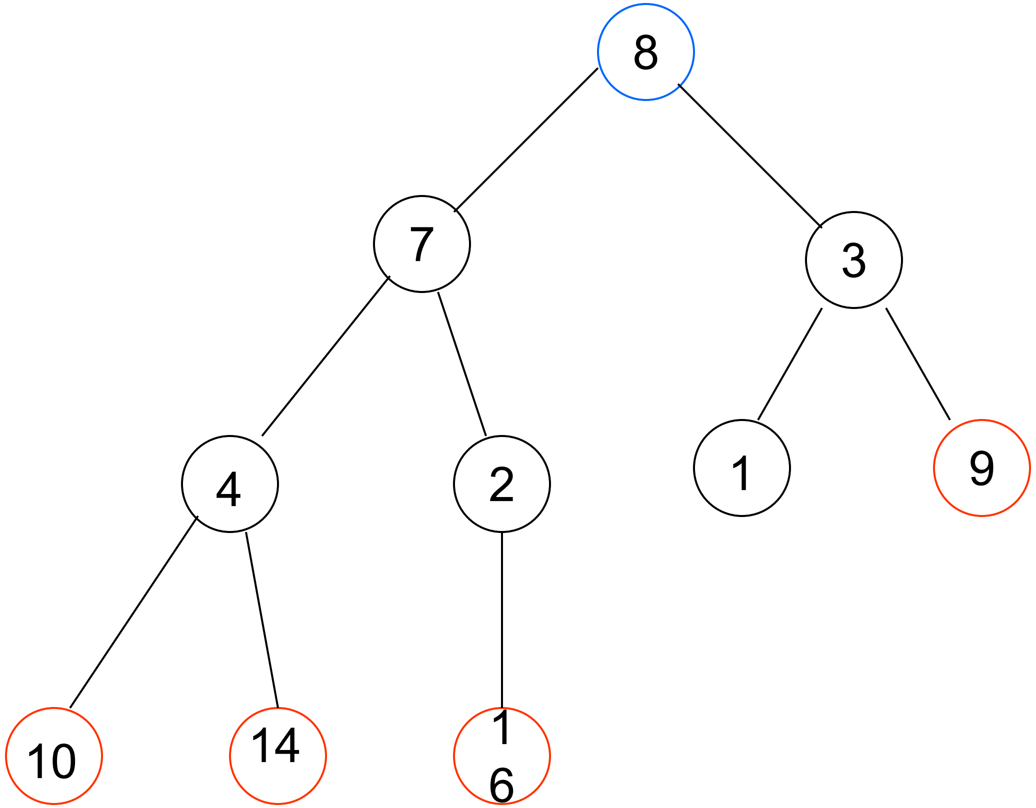
9, 8, 3, 4, 7, 1, 2, 10, 14, 16.



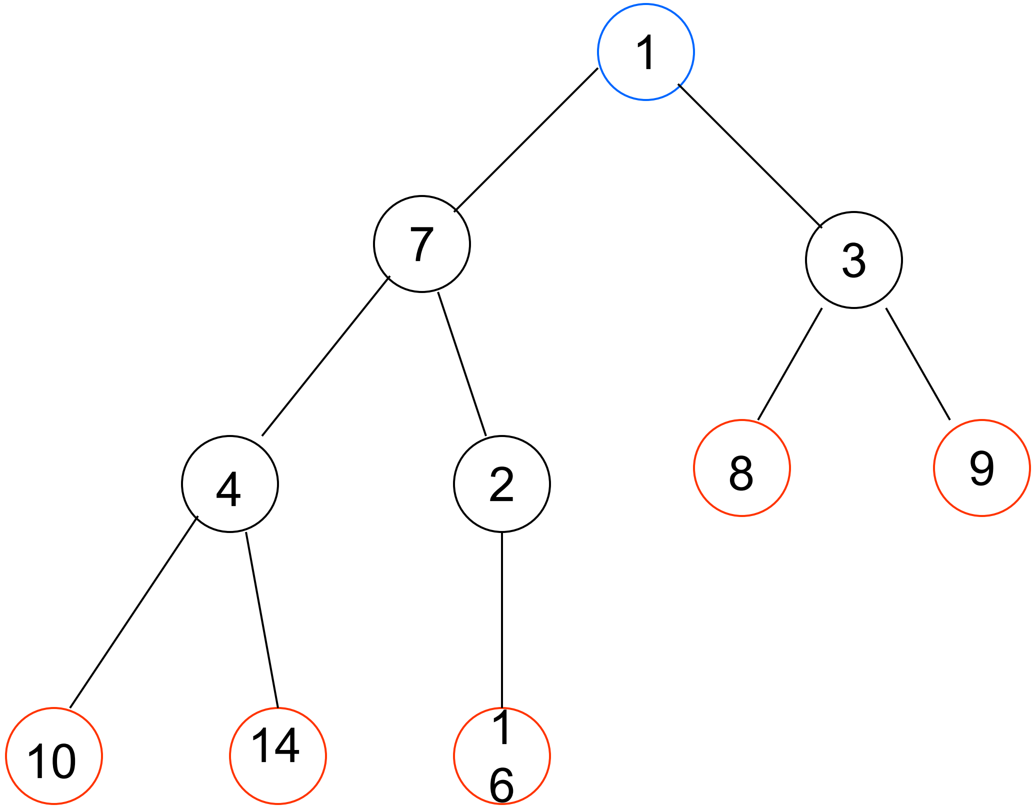
9, 8, 3, 4, 7, 1, 2, 10, 14, 16.



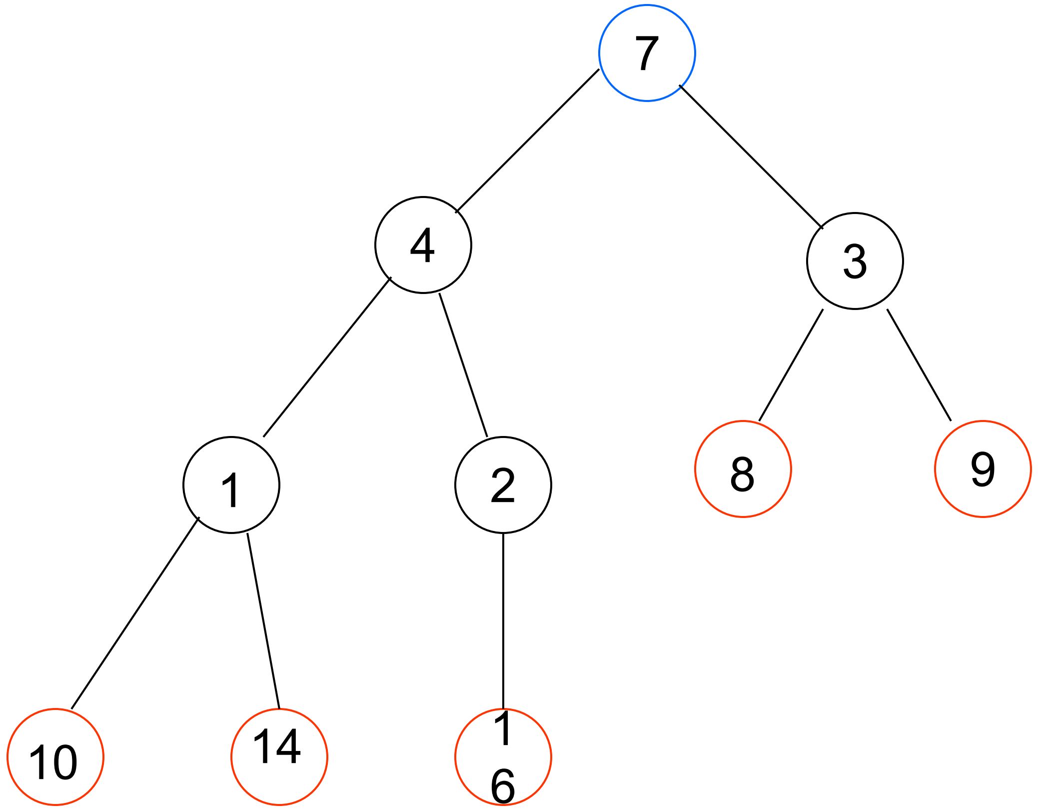
2, 8, 3, 4, 7, 1, 9, 10, 14, 16.

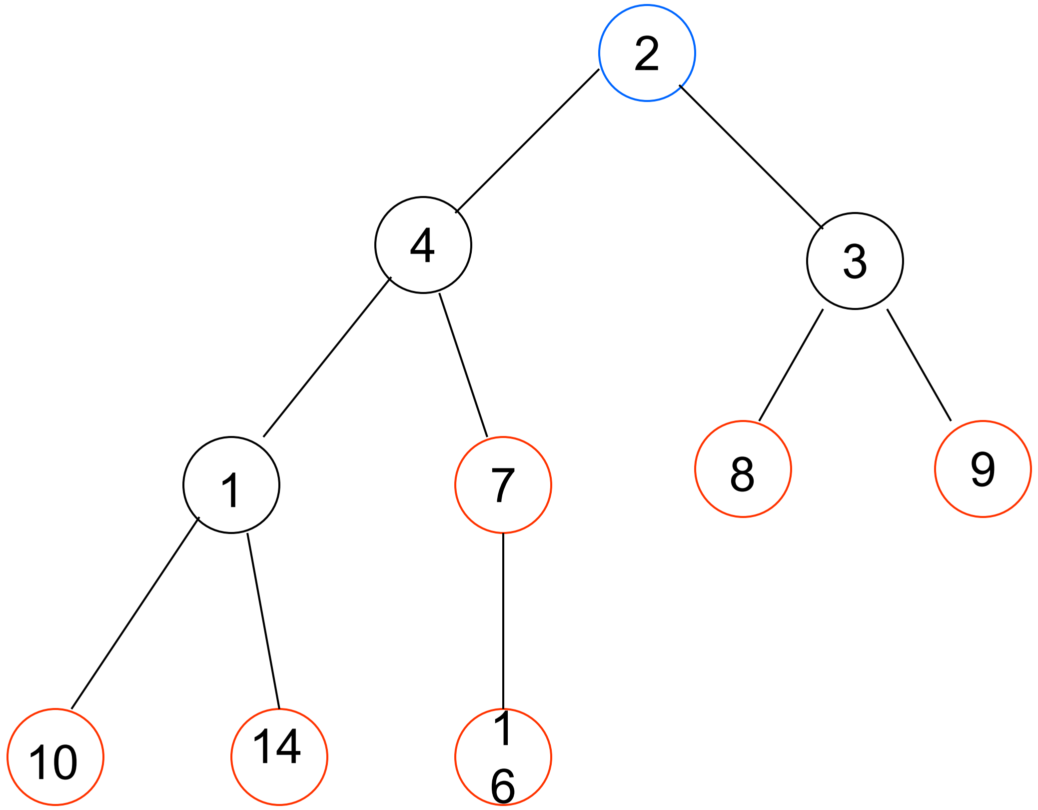


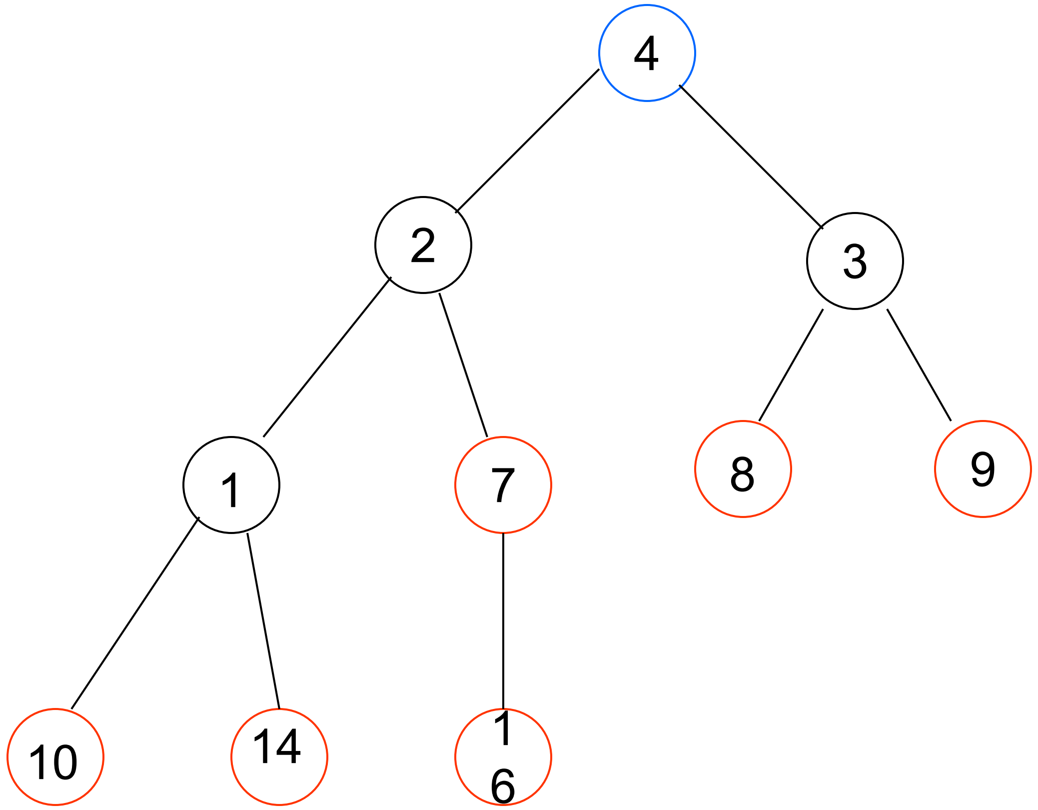
2, 8, 3, 4, 7, 1, 9, 10, 14, 16.

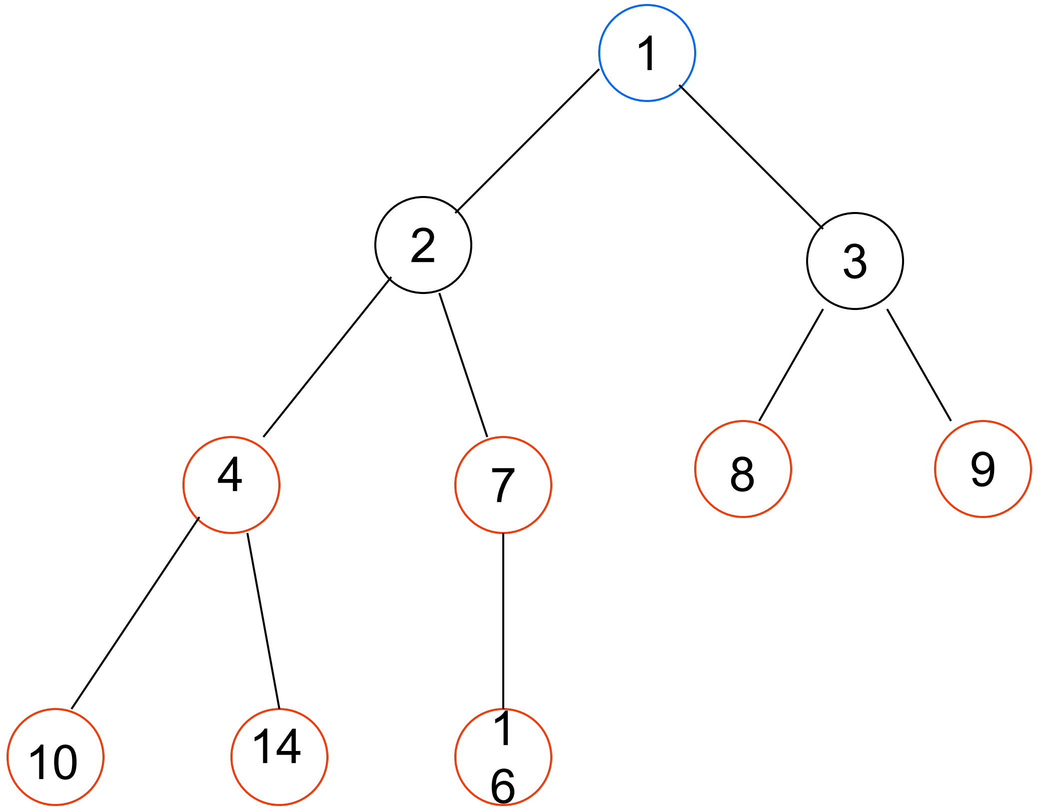


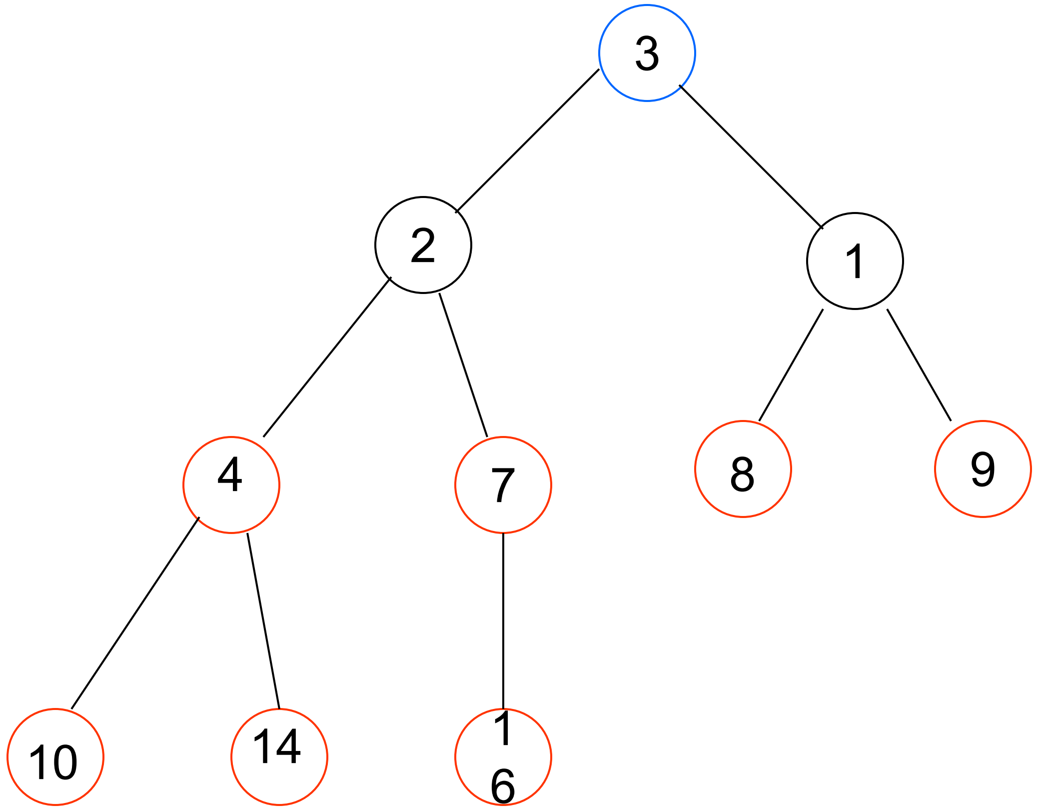
1, 7, 3, 4, 2, 8, 9, 10, 14,16.

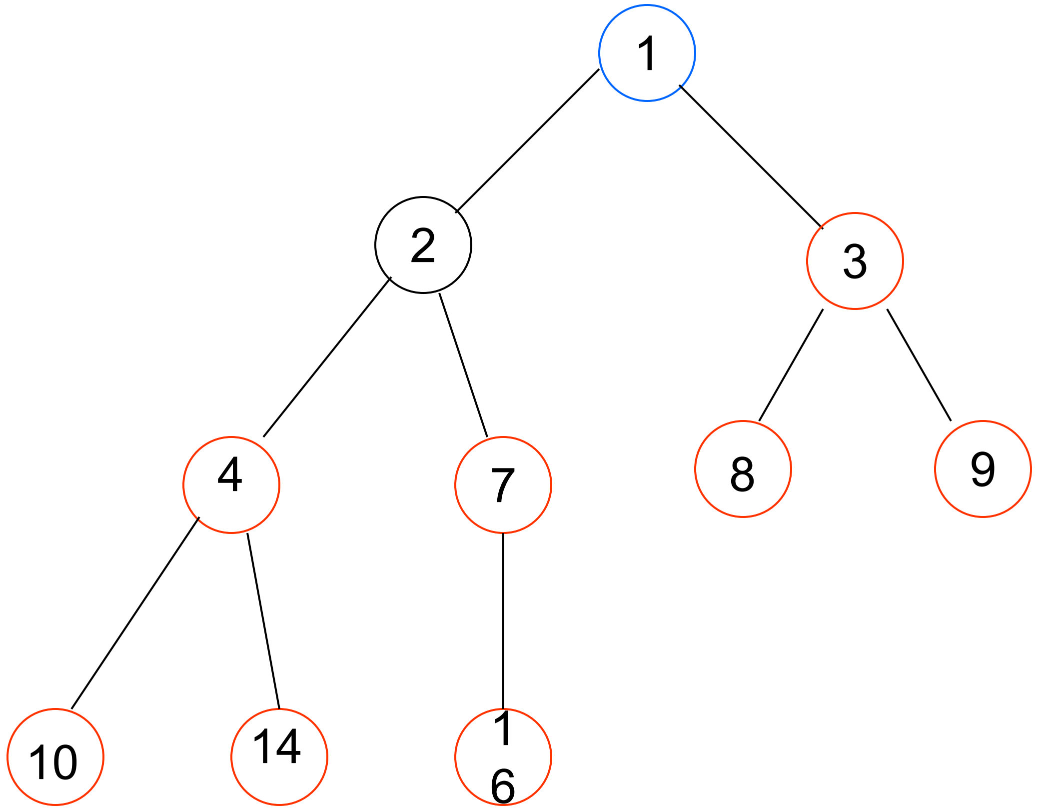

7
4
3
1
2
8
9
10
14
1
6

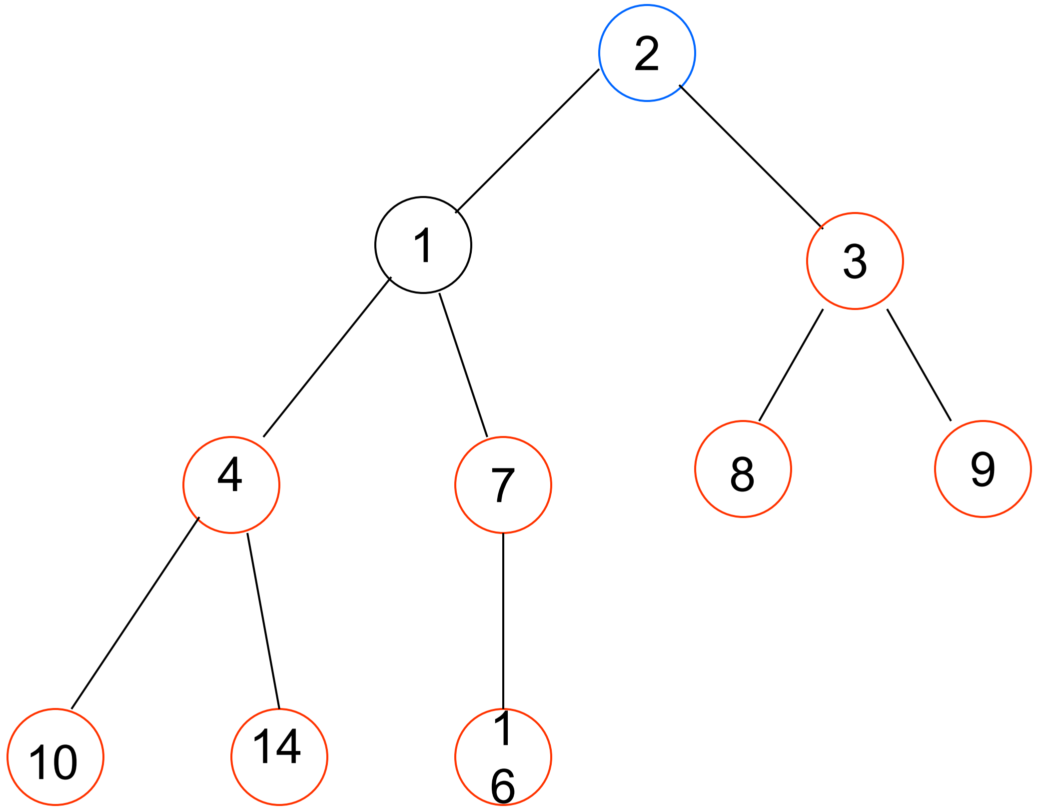

2
4
3
1
7
8
9
10
14
1
6

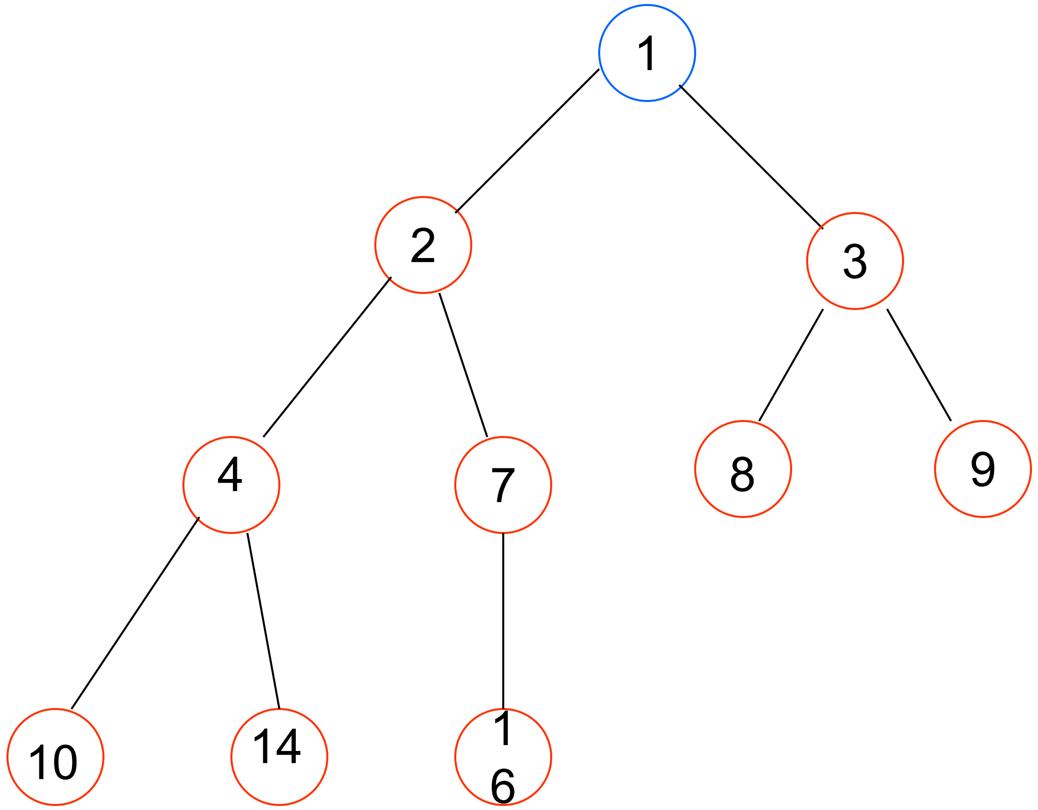

4
2
3
1
7
8
9
10
14
1
6


1
2
3
4
7
8
9
10
14
1
6


3
2
1
4
7
8
9
10
14
1
6


1
2
3
4
7
8
9
10
14
1
6


2
1
3
4
7
8
9
10
14
1
6



1, 2, 3, 4, 7, 8, 9, 10, 14,16.

Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.
We will only consider a max-priority queue.
If we give the key a meaning, such as priority, so that elements with the highest priority have the highest value of key, then we can use the heap structure to extract the element with the highest priority