VE281

Data Structures and Algorithms

Binary Tree Traversal;

Priority Queues and Heaps

Outline

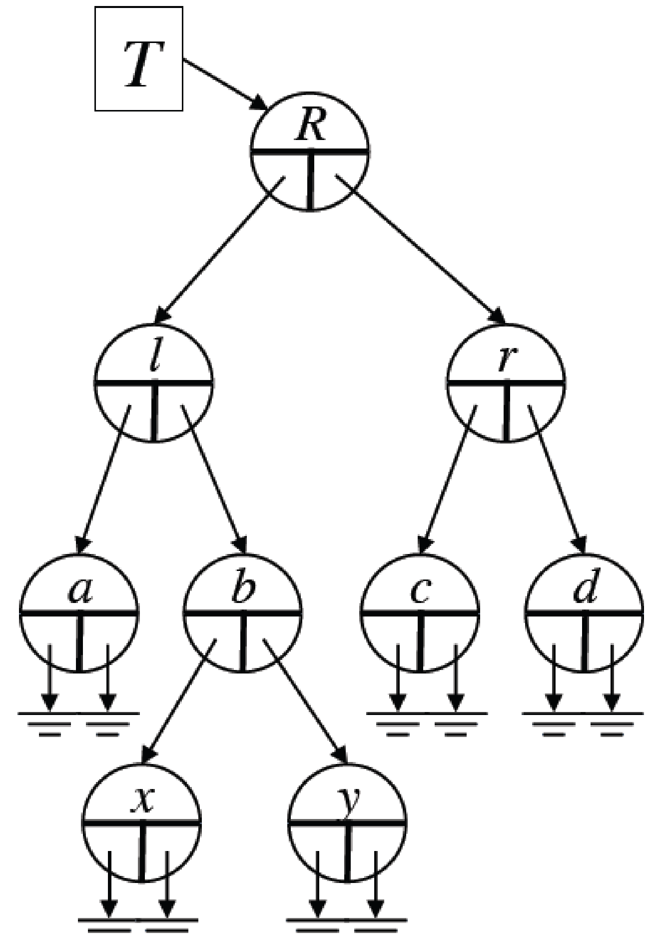
- Binary Tree Traversal

- Priority Queue
- Min Heap and Its Operations

Representing Binary Tree Using Linked Structure

```
struct node {
   Item item;
   node *left;
   node *right;
};
```

- `left/right` points to a left/right **subtree**.
  - If the subtree is an empty one, the pointer points to **NULL**.
- For a leaf node, both its `left` and `right` pointers are NULL.



Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.

- In a traversal, each node of the binary tree is visited **exactly once**.

- During the visit of a node, all actions (making a clone, displaying, evaluating the operator, etc.) with respect to this node are taken.

Binary Tree Traversal Methods

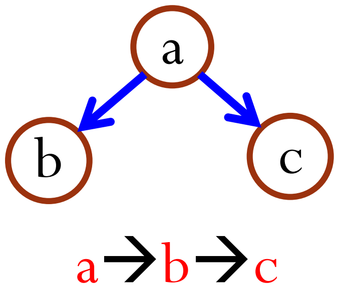
- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order

- Level order traversal

Pre-Order Depth-First Traversal

Procedure

- Visit the node
- Visit its left subtree
- Visit its right subtree

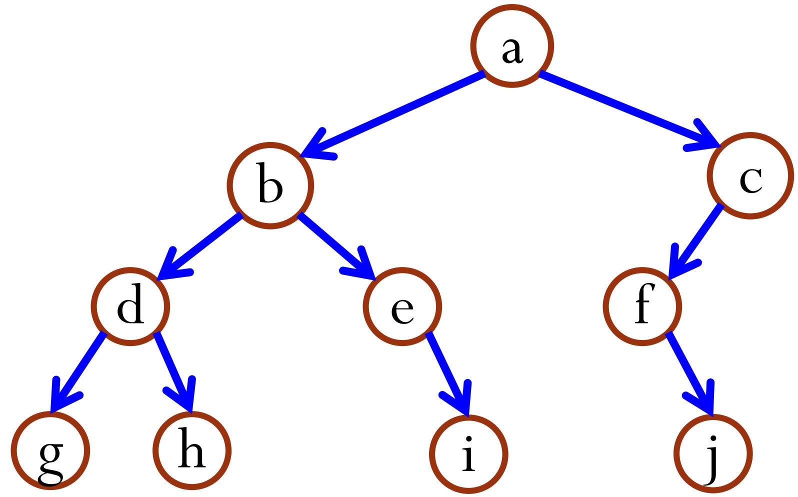



```
void preOrder(node *n) {
  if(!n) return;
  visit(n);
  preOrder(n->left);
  preOrder(n->right);
}
```

Pre-Order Depth-First Traversal

Example

a
b
d
g
h
e
i
c
f
j

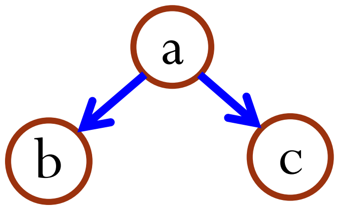


a→b→d→g→h→e→i→c→f→j

Post-Order Depth-First Traversal

Procedure

- Visit the left subtree
- Visit the right subtree
- Visit the node



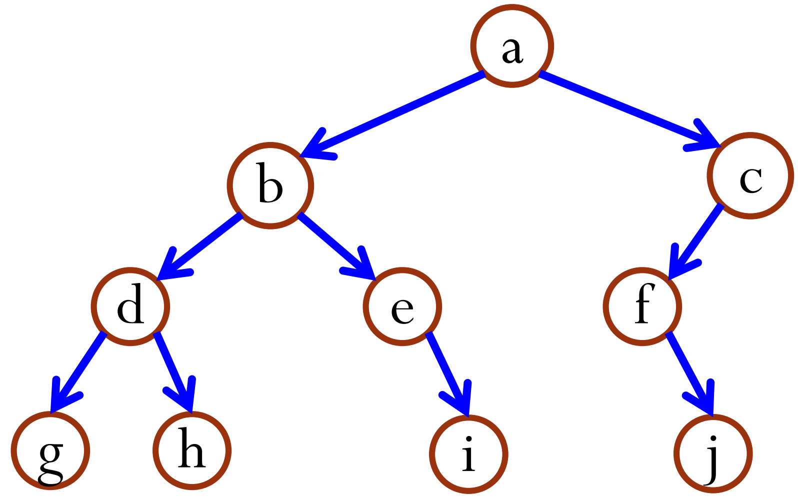
b→c→a

```
void postOrder(node *n) {
  if(!n) return;
  postOrder(n->left);
  postOrder(n->right);
  visit(n);
}
```

Post-Order Depth-First Traversal

Example

g
h
d
i
e
b
j
f
c
a

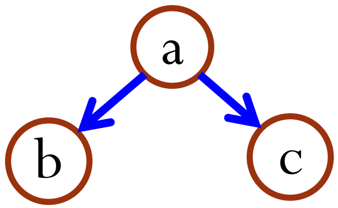


g → h → d → i → e → b → j → f → c → a

In-Order Depth-First Traversal

Procedure

- Visit the left subtree
- Visit the node
- Visit the right subtree

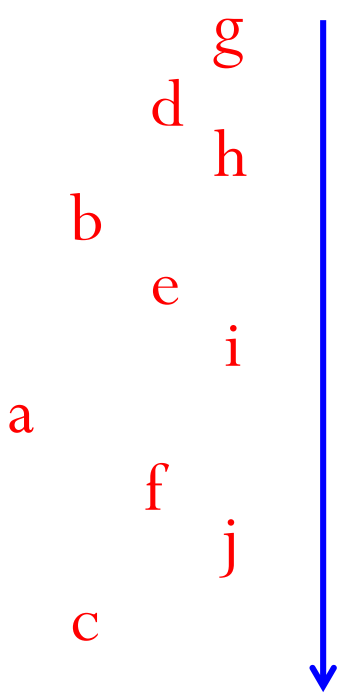


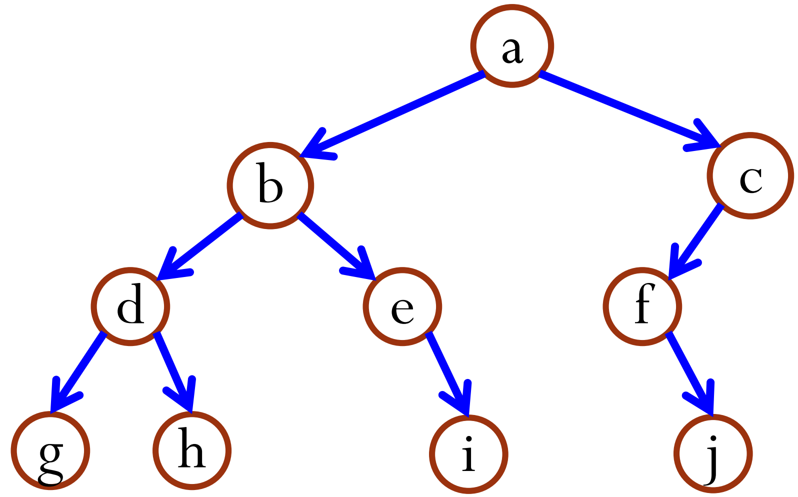
b→a→c

```
void inOrder(node *n) {
  if(!n) return;
  inOrder(n->left);
  visit(n);
  inOrder(n->right);
}
```

In-Order Depth-First Traversal

Example

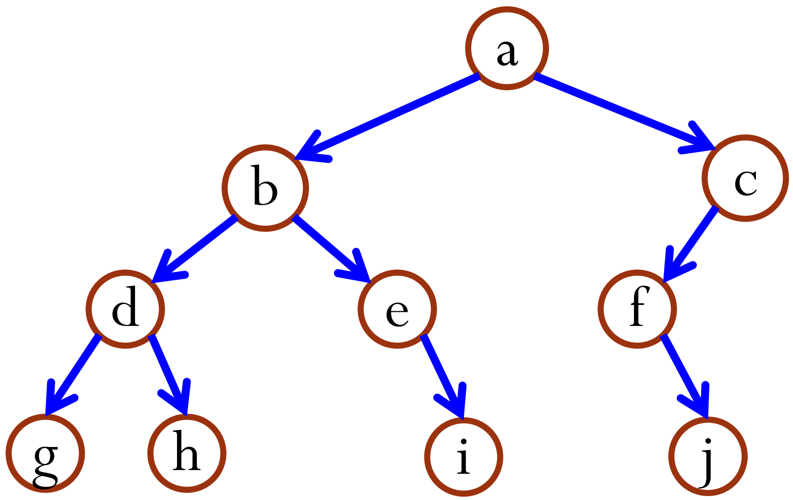




g→d→h→b→e→i→a→f→j→c

Level-Order Traversal

- We want to traverse the tree level by level **from top to bottom**.
- Within each level, traverse **from left to right**.



How can we implement this traversal?

a→b→c→d→e→f→g→h→i→j

Level-Order Traversal

Procedure

- Use a queue!

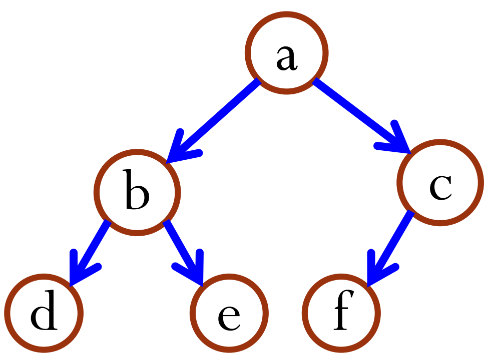
Loop

1. Enqueue the root node into an empty queue.
2. While the queue is not empty, dequeue a node from the front of the queue.
    1. Visit the node.
    2. Enqueue its left child (if exists) and right child (if exists) into the queue.

Level-Order Traversal

Code and Example

```
void levelOrder(node *root) {
  queue q; // Empty queue
  q.enqueue(root);
  while(!q.isEmpty()) {
    node *n = q.dequeue();
    visit(n);
    if(n->left) q.enqueue(n->left);
    if(n->right) q.enqueue(n->right);
  }
}
```



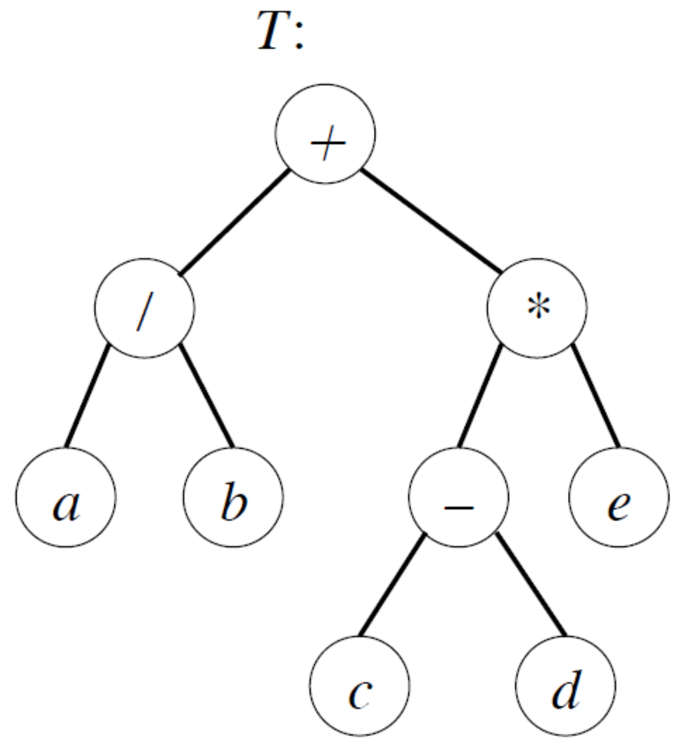
Queue: a b c d e f

Output: a b c d e f

Binary Tree Traversal

Application

- The expression $a/b + (c - d)e$ has been encoded as a tree T.
  - The leaves are **operands**.
  - The internal nodes are **operators**.
- How would you traverse the tree T to print out the expression?
  - In-order depth-first traversal.
- What is the expression printed out by post-order depth-first traversal?
  - $ab/cd - e * +$
  - **Reverse Polish Notation**



Outline

- Binary Tree Traversal

- Priority Queue
- Min Heap and Its Operations

Priority Queues

- Two kinds of priority queues:
  - Min priority queue.
  - Max priority queue.

- We will focus on **min priority queue**.
  - The max priority queue is similar.

What Is Min Priority Queue?

- A collection of items.
- Each item has a key (or "**priority**").

- Support the following operations:
  - **`isEmpty`**
  - **`size`**
  - **`enqueue`**: put an item into the priority queue.
  - **`dequeueMin`**: remove element with **min** key.
  - **`getMin`**: get item with **min** key.

Applications of Priority Queue

- Banking services
  - VIP customer who arrives later gets served first.

- Network bandwidth management
  - The prioritized traffic, such as real-time data, is forwarded with the least delay once it reaches the network router.

- Discrete event simulation
  - One event happening triggers a few others, which are put into a queue.
  - Simulating in the order of the **beginning time** of the events.

Min Priority Queue: Implementation

- A collection of items.
- Each item has a key (or "**priority**").

- Support the following operations:
  - **`isEmpty`**
  - **`size`**
  - **`enqueue`**: put an item into the priority queue.
  - **`dequeueMin`**: remove element with **min** key.
  - **`getMin`**: get item with **min** key.

What's the time complexity for an unsorted array-based implementation?

Priority Queue Implemented with Heap

- Priority queues are most commonly implemented using **Binary Heaps** (will be shown soon).

- Complexity of the operation using heap implementation:
  - **`isEmpty`**, **`size`**, and **`getMin`** are $O(1)$ time complexity in the worst case.
  - **`enqueue`** and **`dequeueMin`** are $O(\log n)$ time complexity in the worst case, where n is the size of the priority queue.

Application of Priority Queue: Sorting

- Sorting elements (in ascending order):
  1. **`enqueue`** elements to be sorted into a min priority queue

     $\boxed{\text{Complexity: } O(n \log n)}$

  2. Repeatedly call **`dequeueMin`** to extract elements out of the queue.

     $\boxed{\text{Complexity: } O(n \log n)}$

- The resulting elements are sorted by their keys.

- What is the time complexity? $O(n \log n)$
- This is known as **heap sort**.

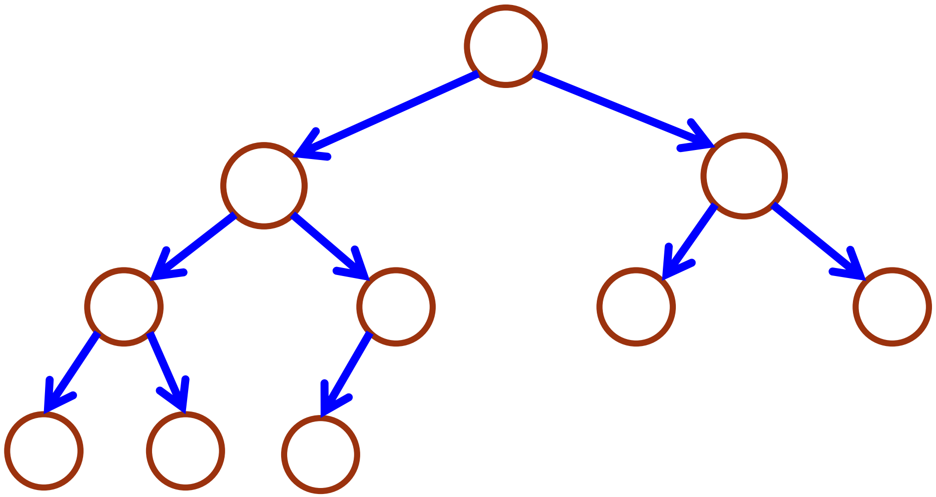
Outline

- Binary Tree Traversal

- Priority Queue
- Min Heap and Its Operations

Binary Heap

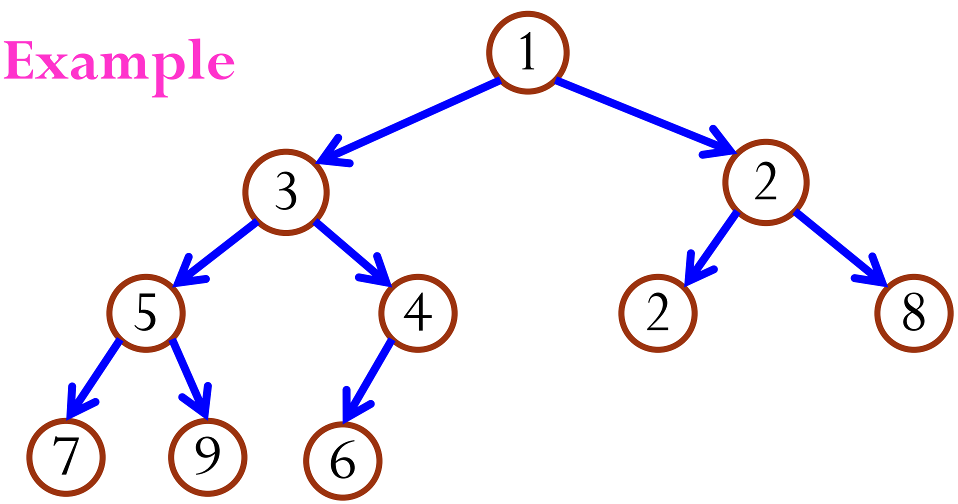
- A **binary heap** is a **complete binary tree**.



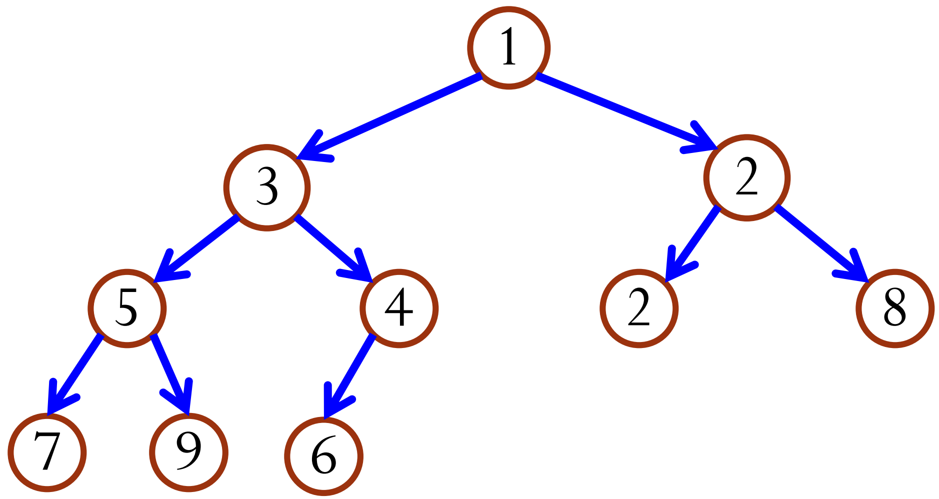
Min Heap

- A min heap is
  - a **binary heap**, and
  - a tree where for **any** node v, the key of v is smaller than or equal to ($\leq$) the keys of any **descendants** of v.
- Property: The key of the root of **any** subtree is always the smallest among all the keys in that subtree.

Example



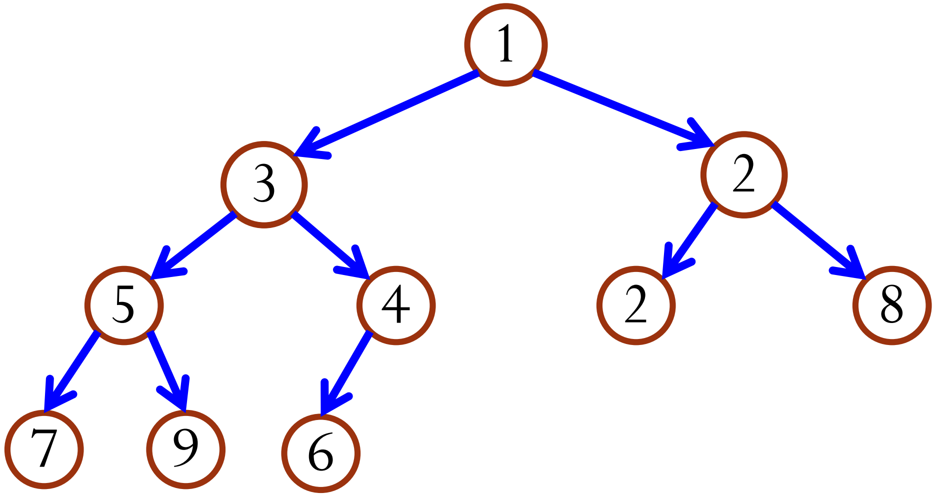
Min Heap



- However, the keys of nodes **across** subtrees have no required relationship.
  - Different from binary search trees, which we will show later.

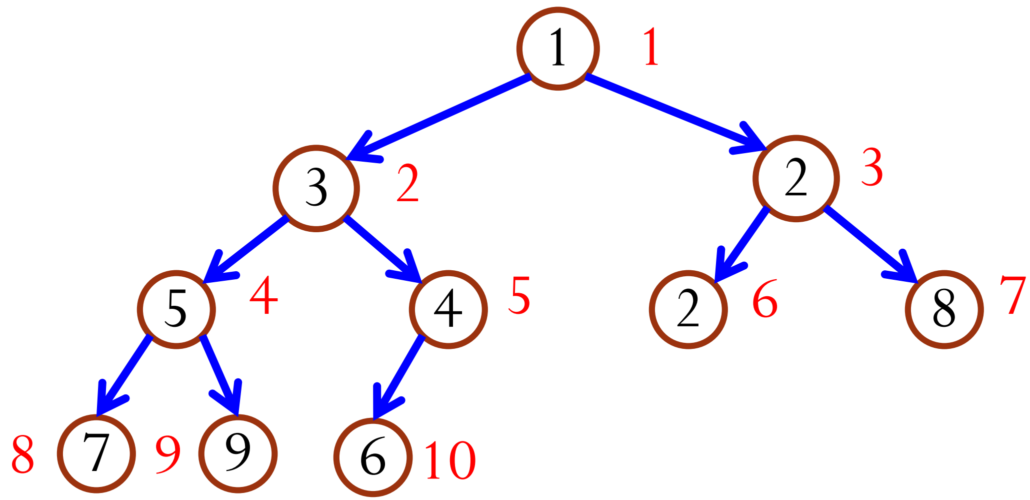
Heap Height

- Assume the heap has n nodes, the height of the heap is
$$\lceil \log_2(n+1) \rceil - 1$$



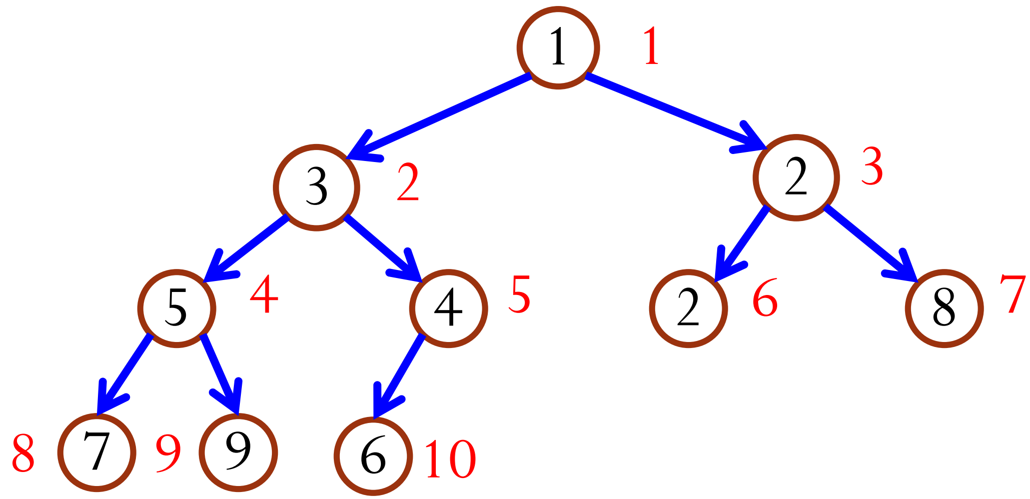
Binary Heap Implementation as an Array

- Store the elements in an array in the order produced by a level-order traversal.
- The first element is stored at index 1.



–	1	3	2	5	4	2	8	7	9	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Index Relation



Index relation allows us to move up and down a heap easily.

- A node at index i ($i \neq 1$) has its parent at index $\lfloor i/2 \rfloor$.
- Assume the number of nodes is n. A node at index i ($2i \leq n$) has its left child at $2i$.
  - If $2i > n$, it has no left child.
- A node at index i ($2i + 1 \leq n$) has its right child at $2i + 1$.
  - If $2i + 1 > n$, it has no right child.

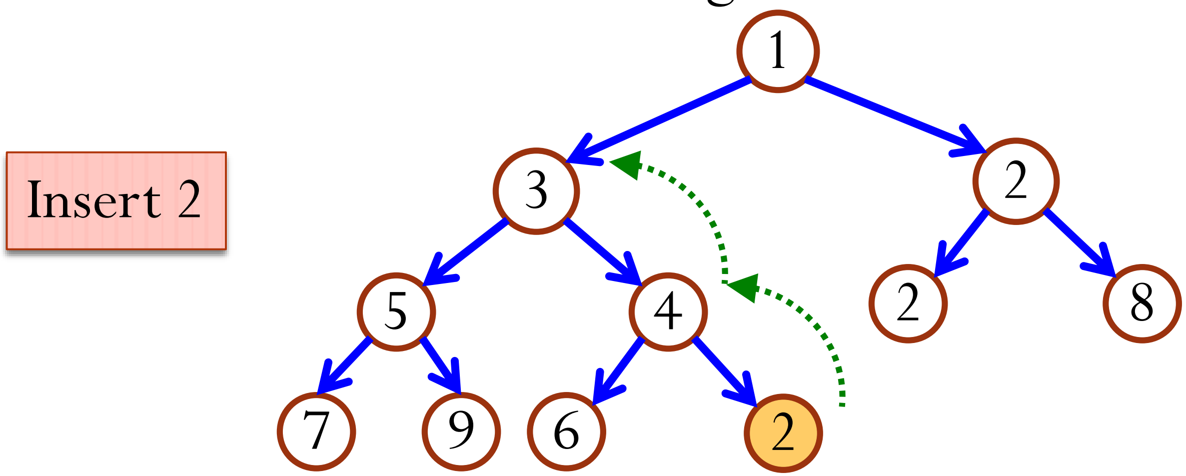
Min Heap Implementation

- We also have a **`size`** variable to keep the number of nodes in the heap.
  - The heap elements are stored in heap[1], heap[2], …, heap[size].

- Operations
  - **`isEmpty: return size==0;`**
  - **`size: return size;`**
  - **`getMin: return heap[1];`**

Procedure of enqueue

- Insert **`newItem`** as the rightmost leaf of the tree.

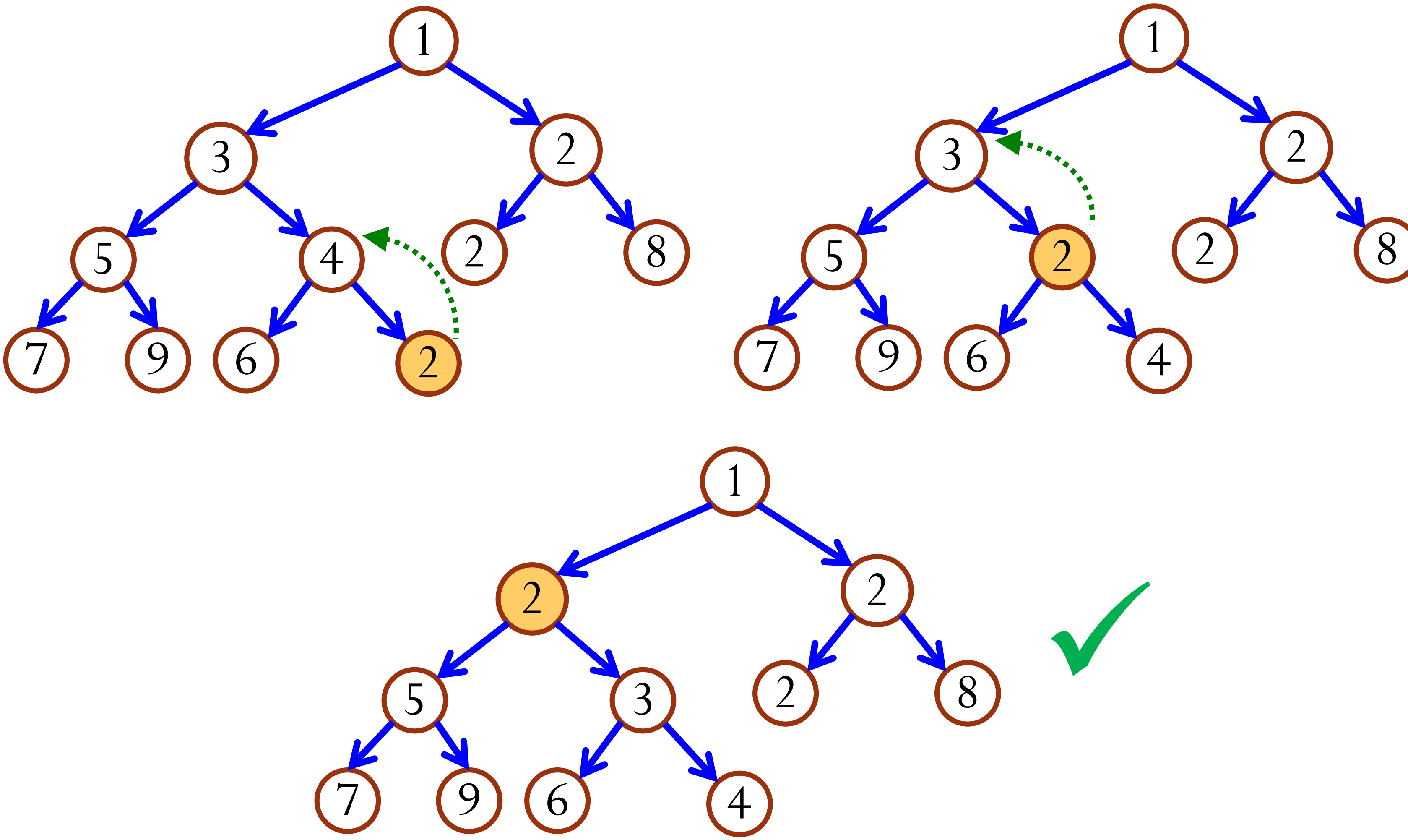



```
heap[++size] = newItem;
```

- The tree may no longer be a heap at this point!
- **Percolate up** **`newItem`** to an appropriate spot in the heap to restore the heap property.

Percolate Up

Illustration



Percolate Up

Code

```
void minHeap::percolateUp(int id) {
  while(id > 1 && heap[id/2] > heap[id]) {
    swap(heap[id], heap[id/2]);
    id = id/2;
  }
}
```

- Pass index (**`id`**) of array element that needs to be percolated up.
- Swap the given node with its parent and move up to parent until:
  - we reach the root at position 1, or
  - the parent has a smaller or equal key.

enqueue

Code

```
void minHeap::enqueue(Item newItem) {
  heap[++size] = newItem;
  percolateUp(size);
}
```

- What is the time complexity?
  - $O(\log n)$