Data Structure and Algorithm

Lecture 16

(Heap)

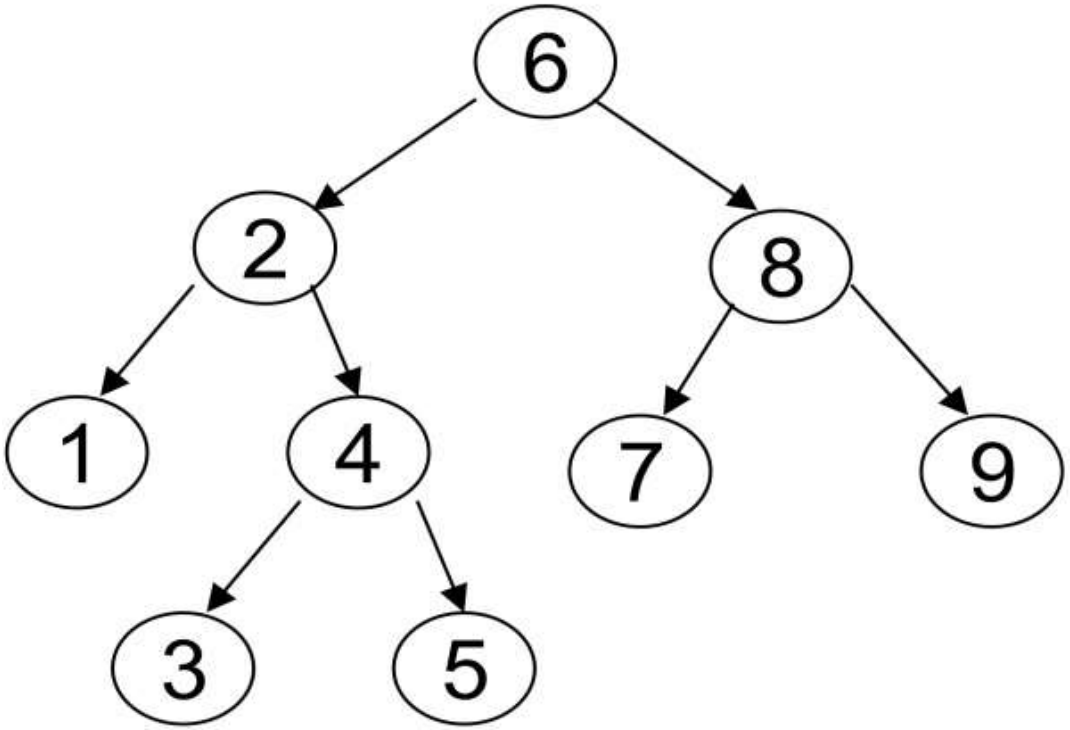
Binary search tree (BST)

Binary search tree (BST) is a node-based binary tree data structure which has the following properties:

- The left sub tree of a node contains only nodes with keys less than the root node's key.
- The right sub-tree of a node contains only nodes with keys greater than the root node's key.
- Both the left and right sub-trees must also be binary search trees.

BST

From the above properties it naturally follows that:



Program: Creating a Binary Search Tree

We assume that every node of a binary search tree is capable of holding an integer data item and that the links can be made to point to the root of the left subtree and the right subtree, respectively. Therefore, the structure of the node can be defined using the following declaration:

Binary Search Tree Class Definition

```
struct node
  {
     int info;
     struct node *left;
     struct node *right;
  };

  class BST
  {
     public:
        void find(int, node **, node **);
        void insert(node *, int);
        void preorder(node *);
        void inorder(node *);
        void postorder(node *);
        void display(node *, int);
        BST();
        node *root;
  }
```

Binary Search Tree Insert

```
void BST::insert(node *tree, int data)
   {
      node *newnode;
      newnode = new node;
      newnode->info=data;

      if (root == NULL)
      {
         root = new node;
         root->info = newnode->info;
         root->left = NULL;
         root->right = NULL;
         cout<<"Root Node is Added"<<endl;
         return;
      }
```

Binary Search Tree Insert

```
 if (tree->info == newnode->info)
        {
           cout<<"Element already in the tree"<<endl;
           return;
        }
 if (tree->info > newnode->info)
        {
           if (tree->left != NULL)
              {
              insert(tree->left, newnode->info);
                  }
              else
                  {
              tree->left = newnode;
              (tree->left)->left = NULL;
              (tree->left)->right = NULL;
              cout<<"Node Added To Left"<<endl;
              return;
               } }
```

Binary Search Tree Insert

```
else
        {
           if (tree->right != NULL)
           {
              insert(tree->right, newnode->info);
           }
           else
           {
              tree->right = newnode;
              (tree->right)->left = NULL;
              (tree->right)->right = NULL;
              cout<<"Node Added To Right"<<endl;
              return;
           }
        }
```

Operation on BST

OPERATIONS

Operations on a binary tree require comparisons between nodes.

These comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values.

This comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.

The following are the operations that are being done in Binary Tree

- Searching.
- Sorting.
- Deletion.
- Insertion.

Applications of Trees

1. Compiler Design.
2. Unix / Linux.
3. Database Management.
4. dynamic dictionary
4. Trees are very important data structures in computing.
5. They are suitable for:

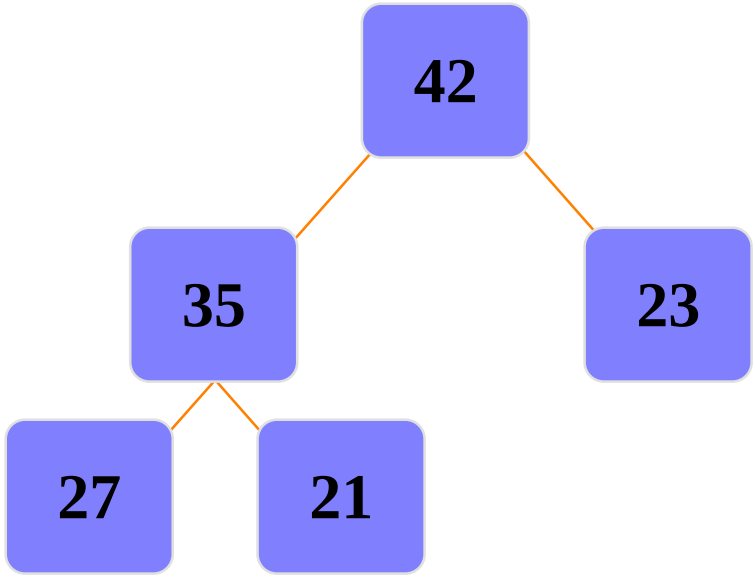
a. Hierarchical structure representation, e.g.,
   i. File directory.
   ii. Organizational structure of an institution.
   iii. Class inheritance tree.

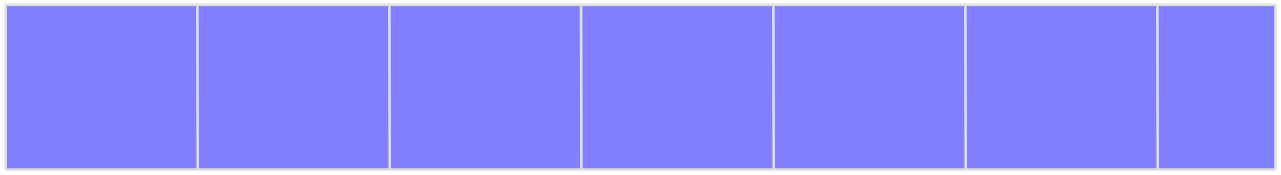
b. Problem representation, e.g.,
   i. Expression tree.
   ii. Decision tree.

C. Efficient algorithmic solutions, e.g.,
   i. Search trees.
   ii. Efficient priority queues via heaps.

Implementing a Heap

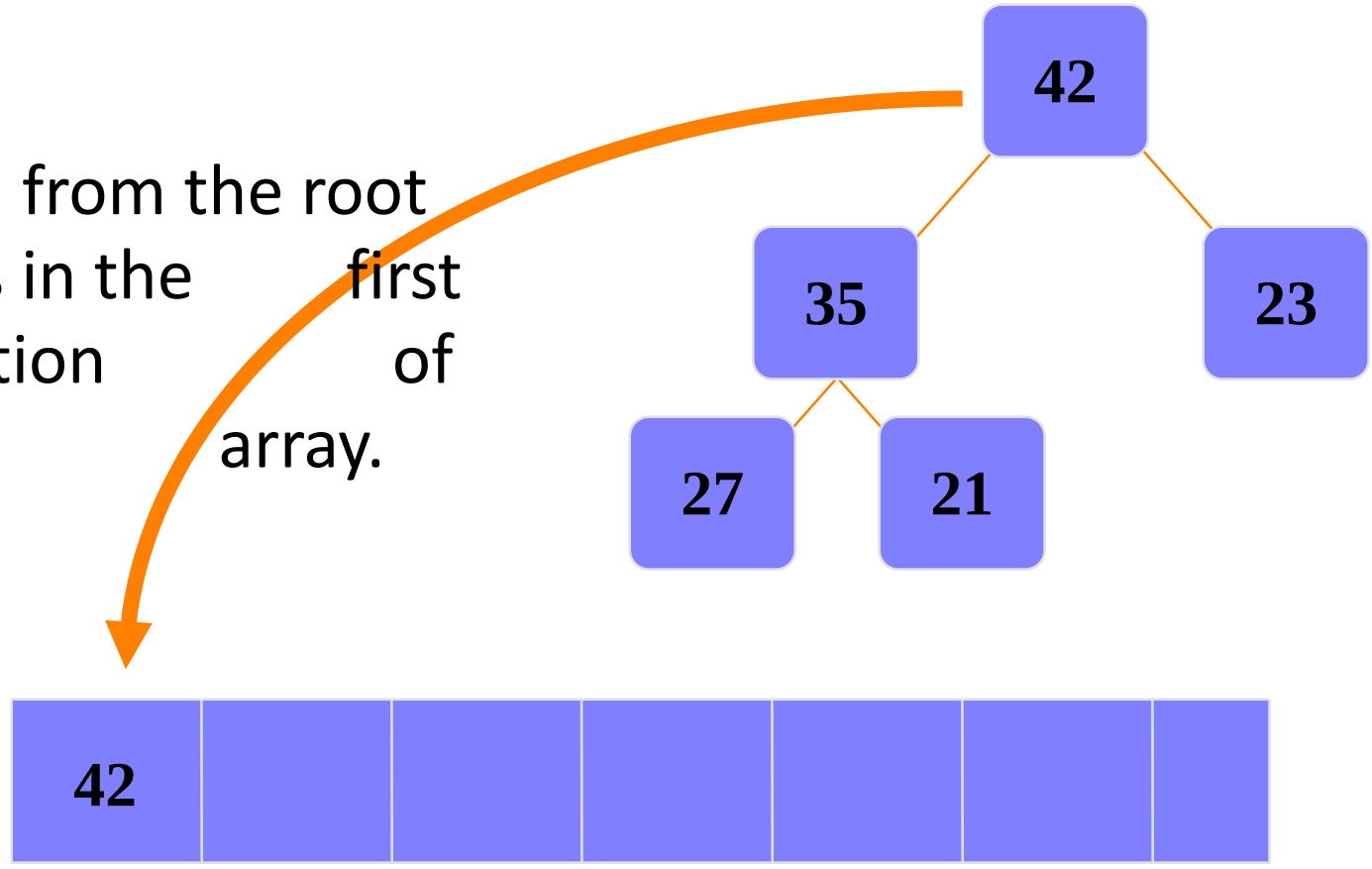
- We will store the data from the nodes in a partially-filled array.





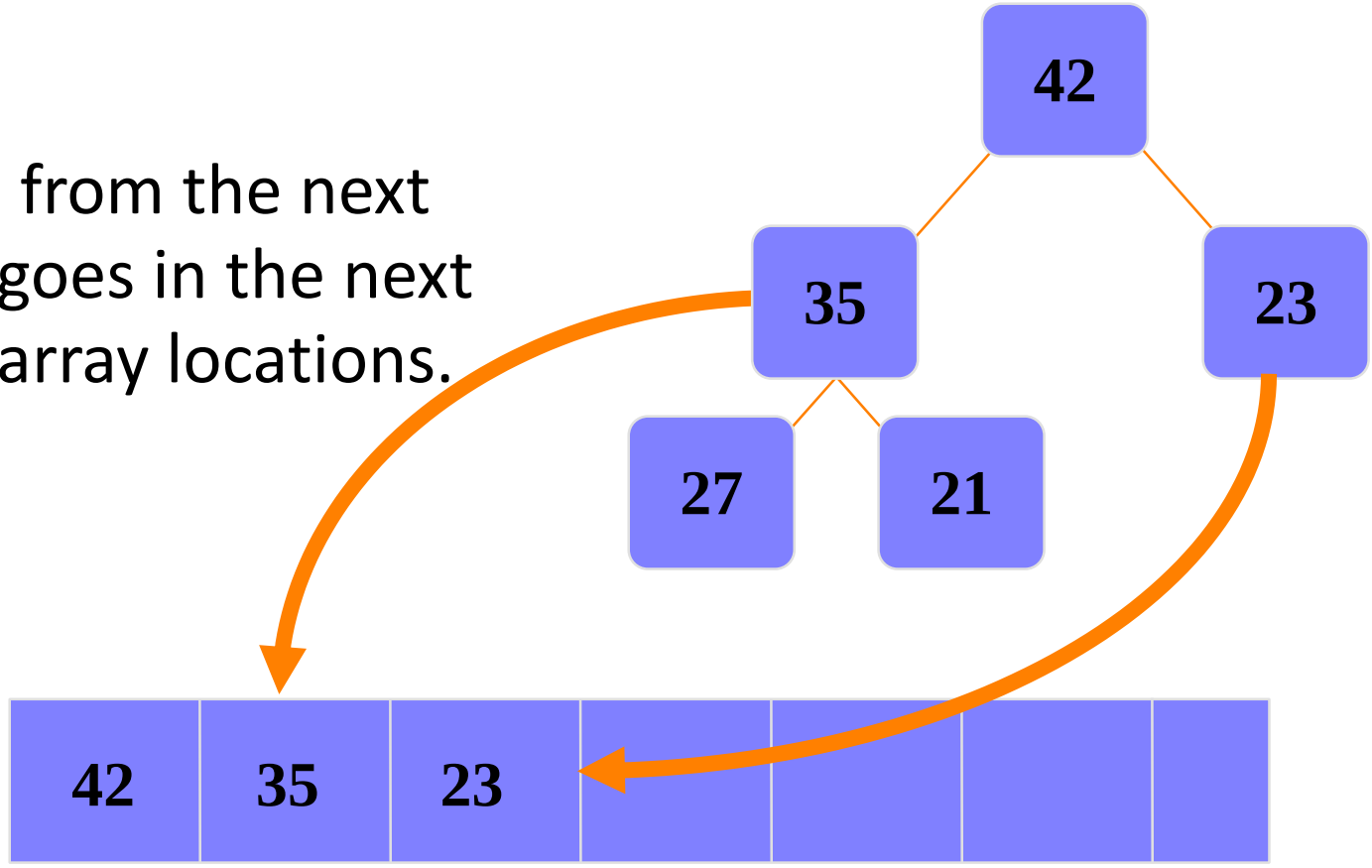
Implementing a Heap

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



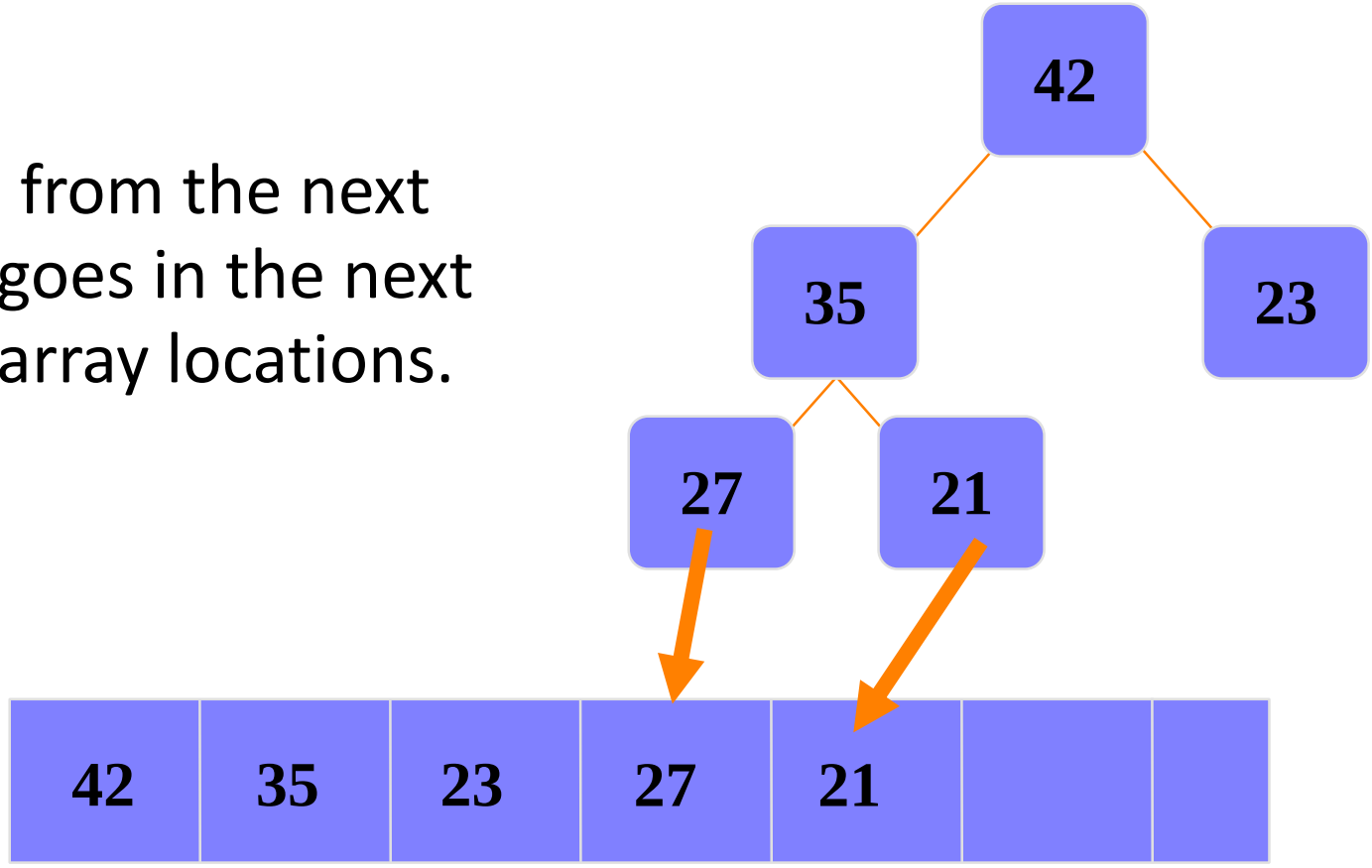
Implementing a Heap

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



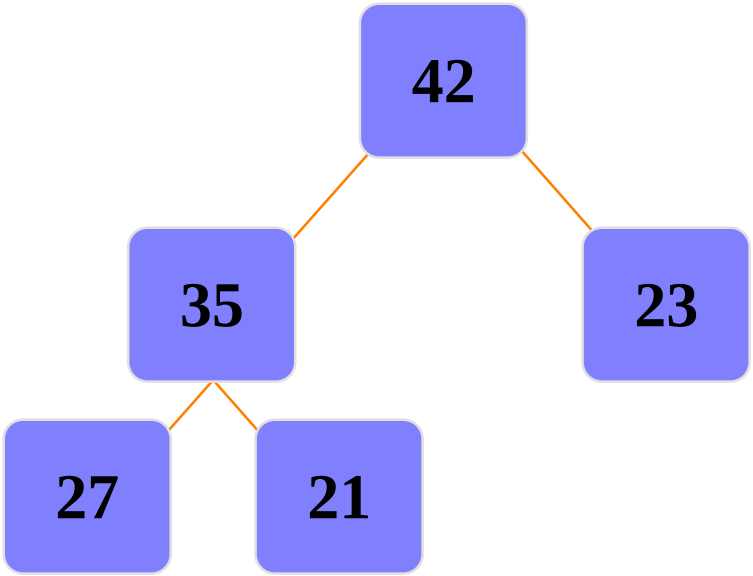
Implementing a Heap

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



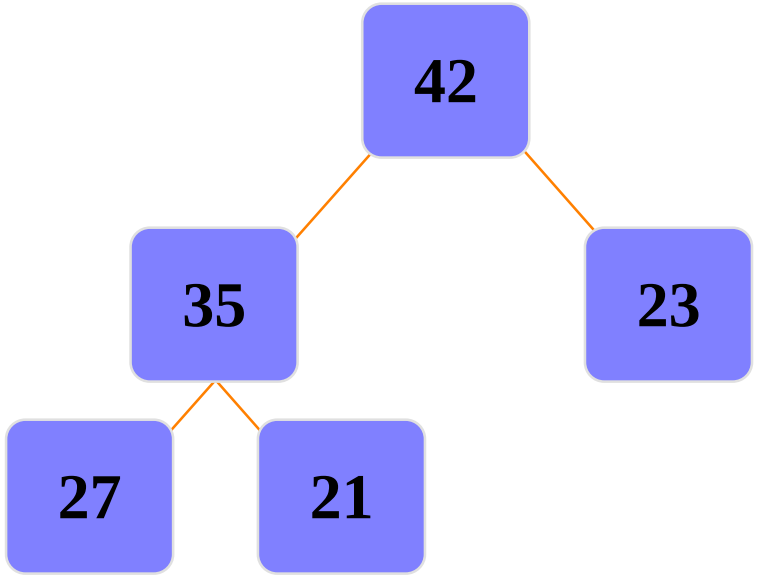
Implementing a Heap

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



Important Points about the Implementation

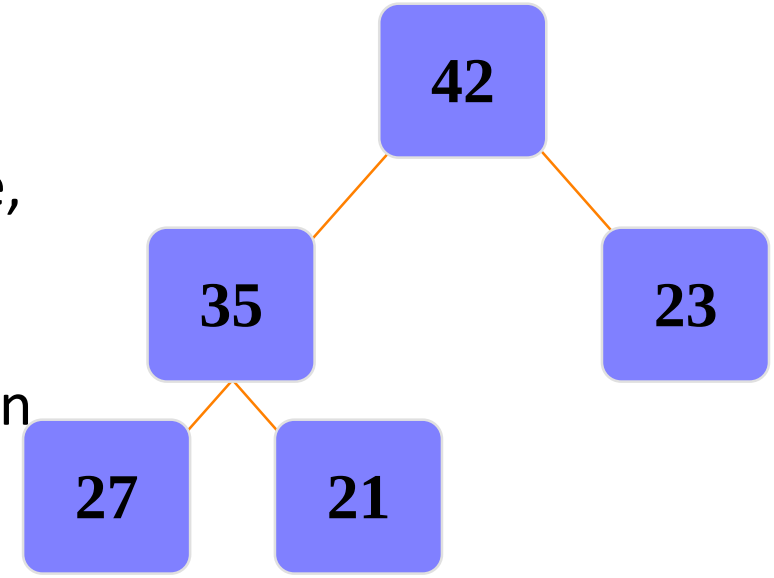
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



42	35	23	27	21		

Important Points about the Implementation

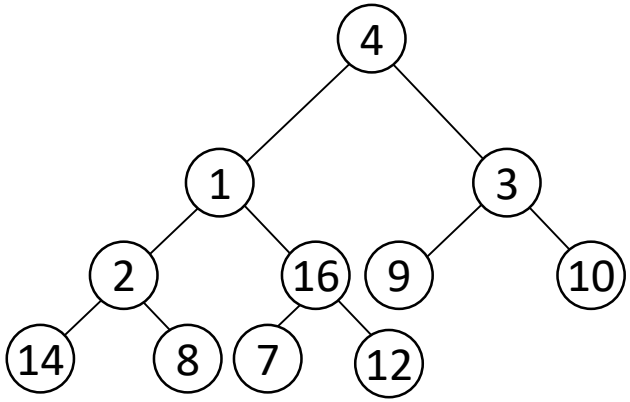
- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.



42	35	23	27	21		

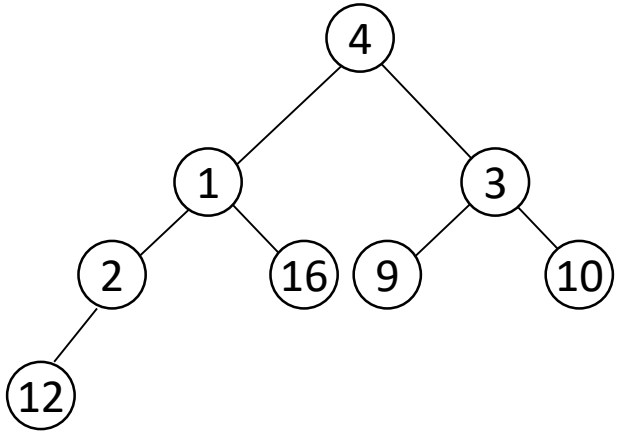
Special Types of Trees

Def: **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

Def: **Complete binary tree** = A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



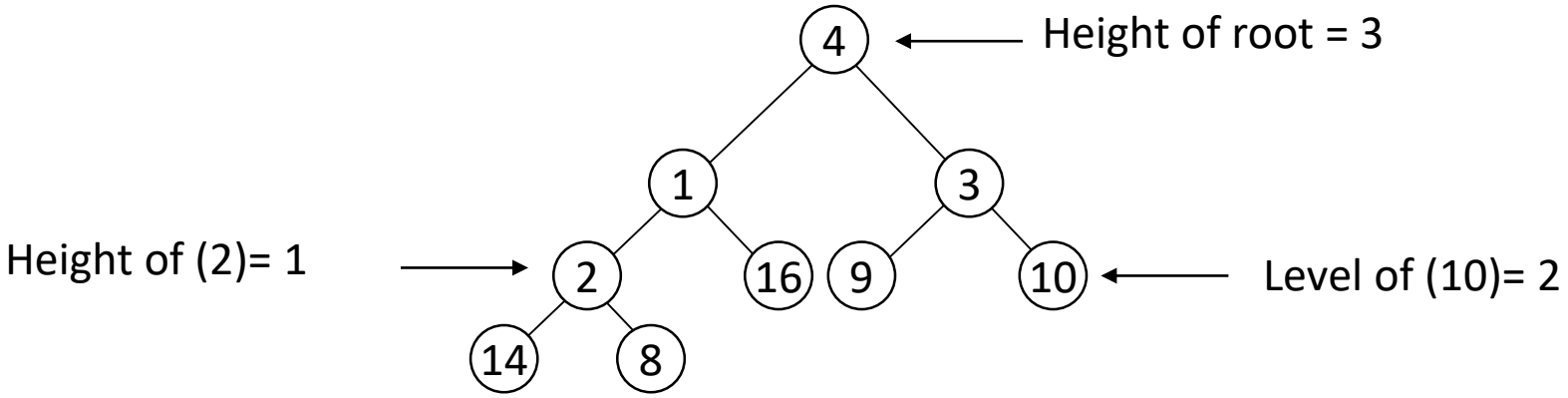
Complete binary tree

Definitions

Height of a node = the number of edges on the longest simple path from the node down to a leaf

Level of a node = the length of a path from the root to the node

Height of tree = height of root node



Useful Properties

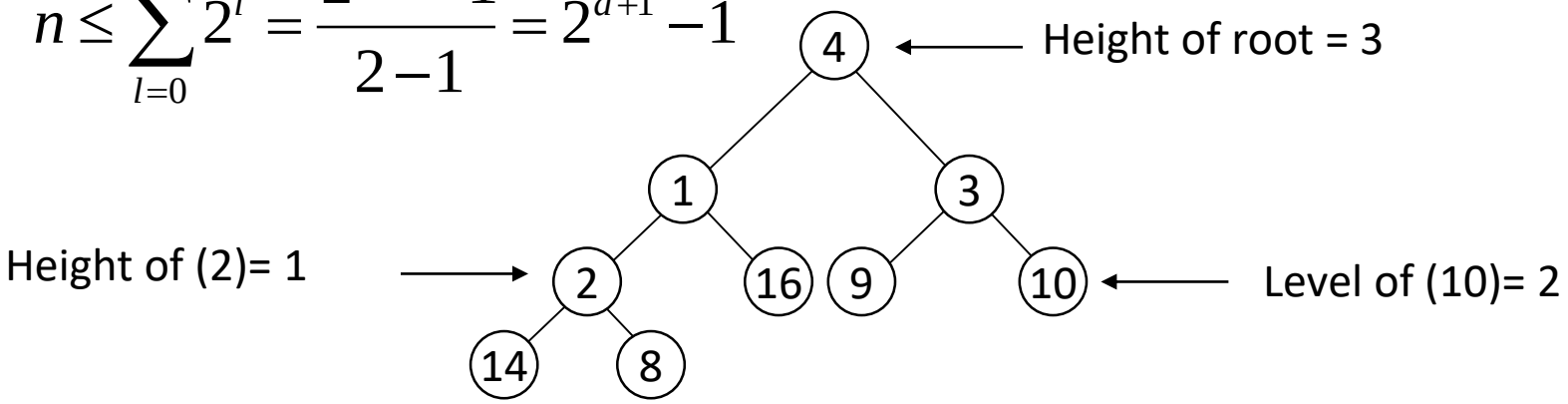
- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with depth d has **at most** $2^{d+1} - 1$ nodes

height

- A binary tree with n nodes has depth **at least** $\lfloor lgn \rfloor$

height

$$n \leq \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$



Heaps

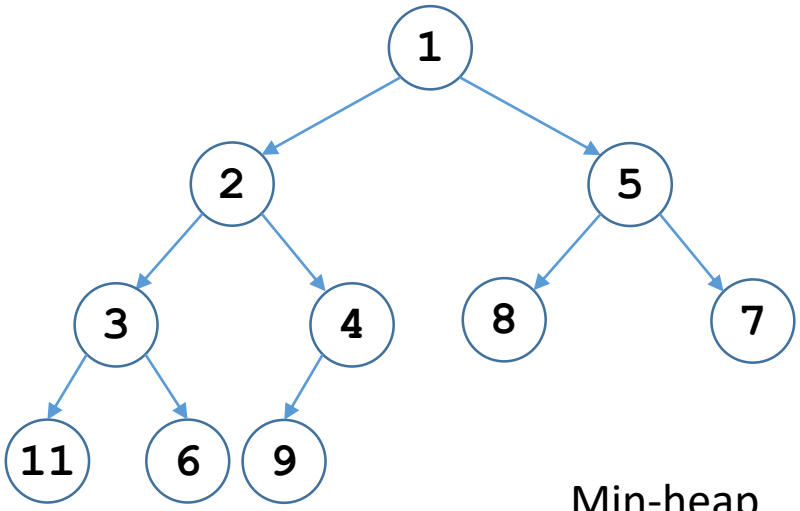
A complete binary tree

- Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)
- Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)

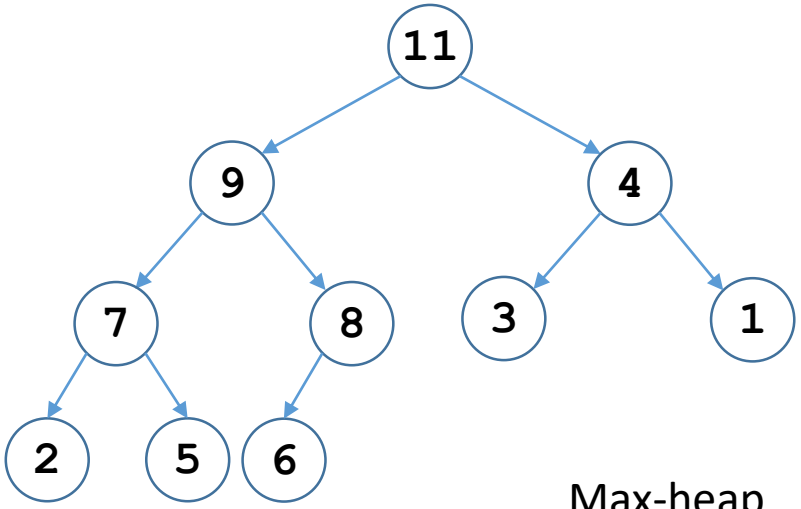
Heaps

A complete binary tree

- Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)
- Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)



Min-heap



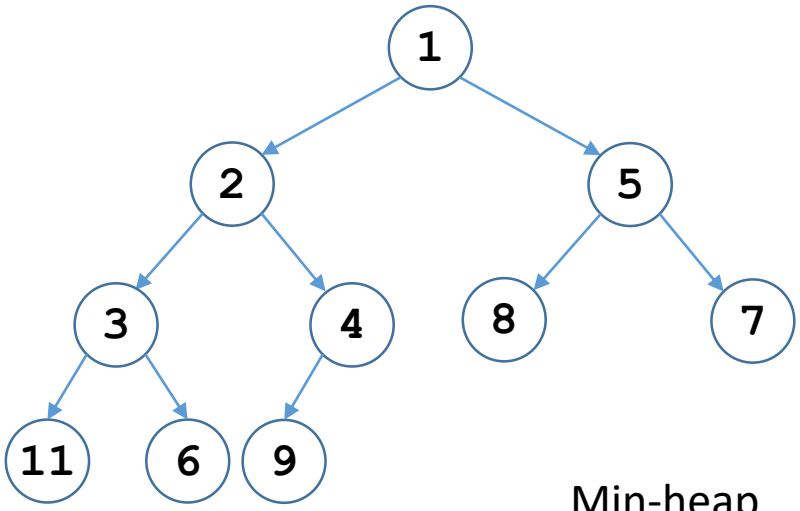
Max-heap

Heaps

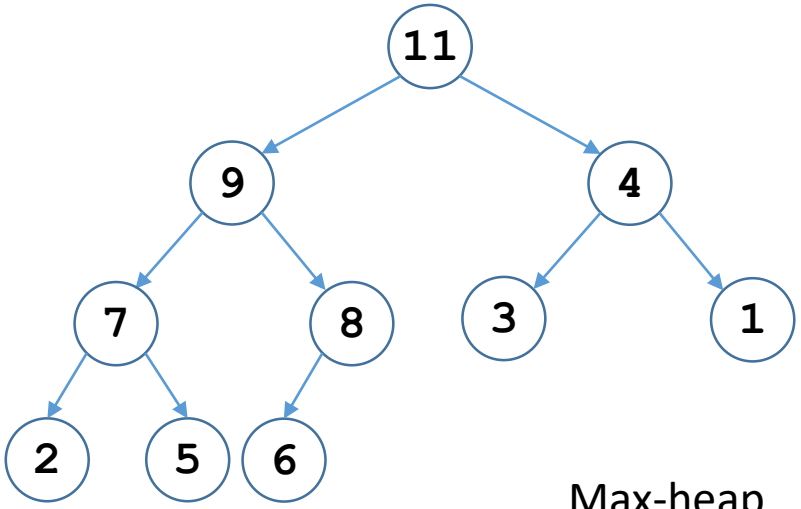
A complete binary tree ← **Shape property**

Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)

Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)



Min-heap



Max-heap

- The shape of all heaps with a given number of elements is the same.

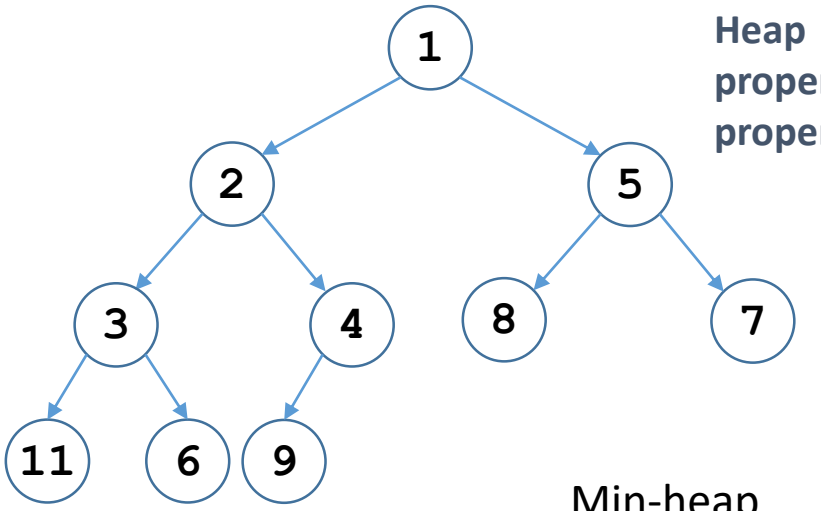
Heaps

A complete binary tree ← **Shape property**

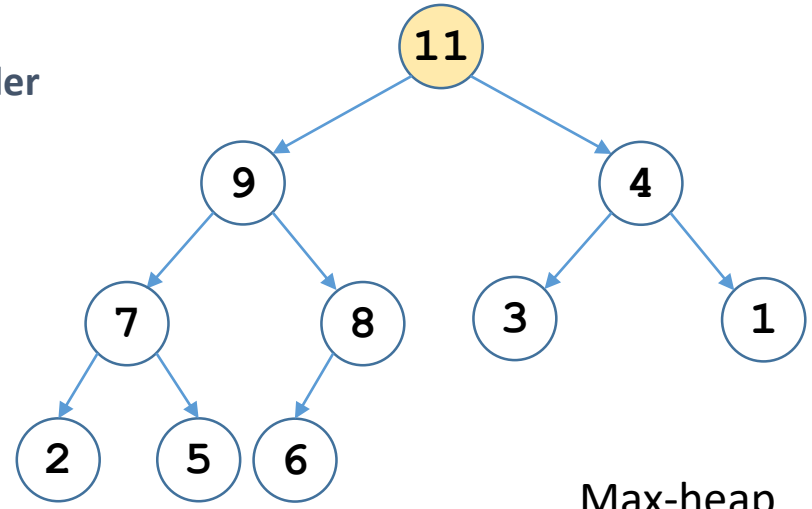
Each of the elements contains a value that is less than or equal to the value of each of its children (Min-heap)

Each of the elements contains a value that is greater than or equal to the value of each of its children (Max-heap)

Heap property/Order property



Min-heap

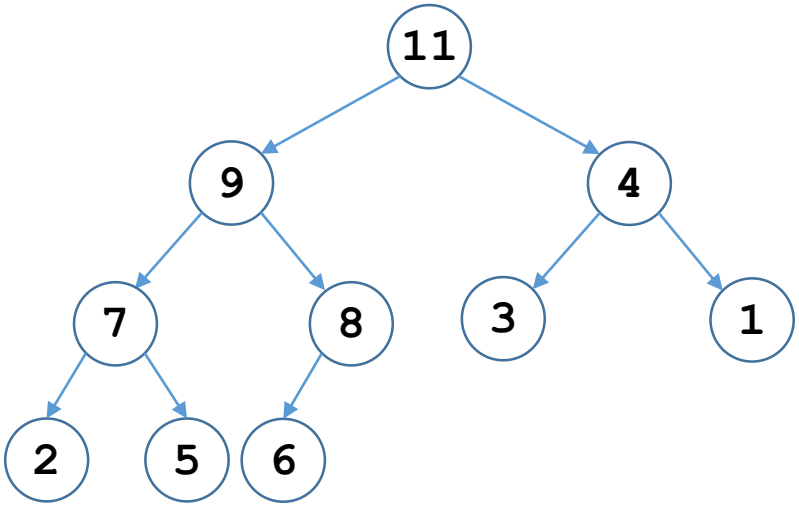


Max-heap

- The shape of all heaps with a given number of elements is the same.
- The root node always contains the largest value in the max-heap (in addition, the subtrees are heaps as well).

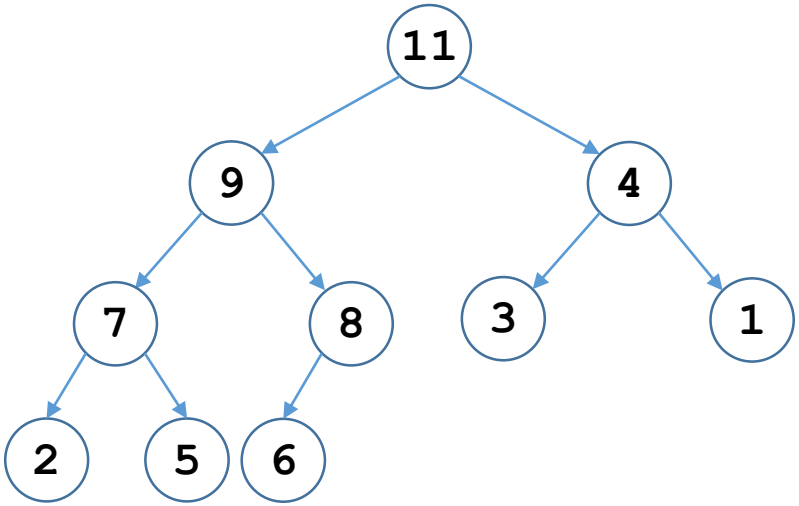
Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete)



Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete, there are not any “holes” in the tree)



Index	Value
1	11
2	9
3	4
4	7
5	8
6	3
7	1
8	2
9	5
10	6

Map from array elements to tree nodes and vice versa

Root – $A[1]$

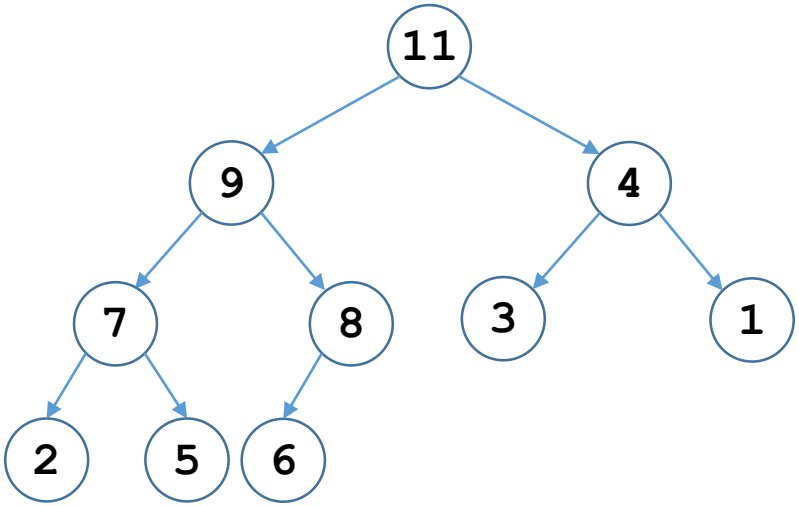
Left[i] – $A[2i]$

Right[i] – $A[2i+1]$

Parent[i] – $A[\lfloor i/2 \rfloor]$

Heaps (Implementation Issue)

Heap elements can be stored as array elements (since the tree is complete, there are not any “holes” in the tree)



Index	Value
1	11
2	9
3	4
4	7
5	8
6	3
7	1
8	2
9	5
10	6

length[*A*] – number of elements in array *A*.

No. of leaves = $\lceil n/2 \rceil$

Height of a heap = $\lfloor \lg n \rfloor$

Operations on Heaps

Maintain/Restore the max-heap property

MAX-HEAPIFY

Create a max-heap from an unordered array

BUILD-MAX-HEAP

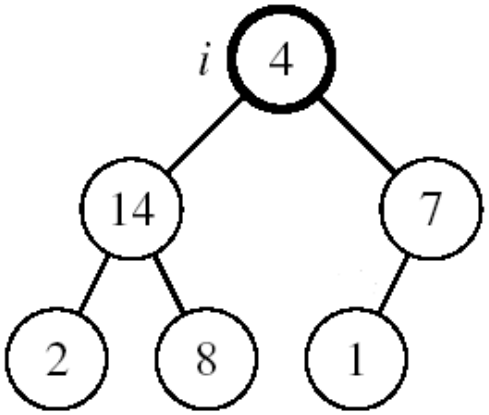
Sort an array in place

HEAPSORT

Priority queues

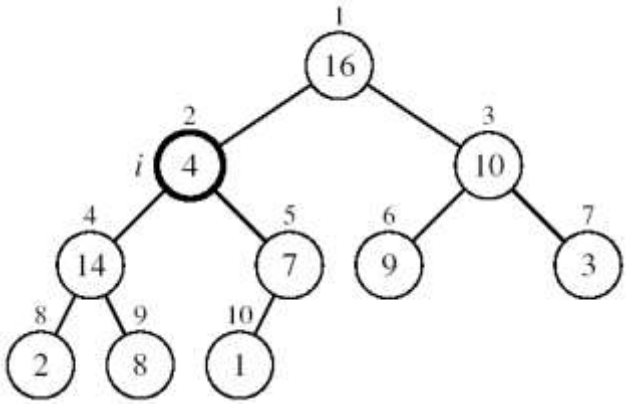
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Assume Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



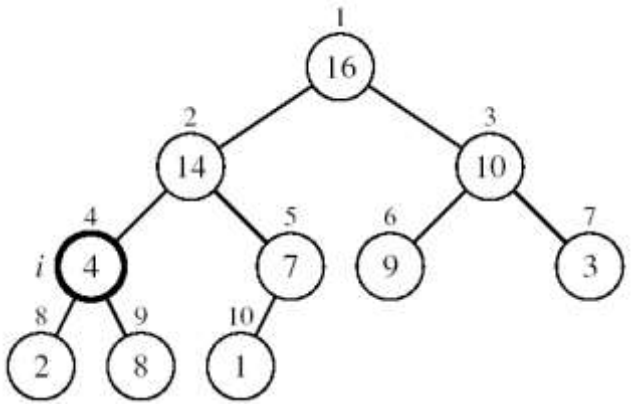
Example

MAX-HEAPIFY(A, 2, 10)



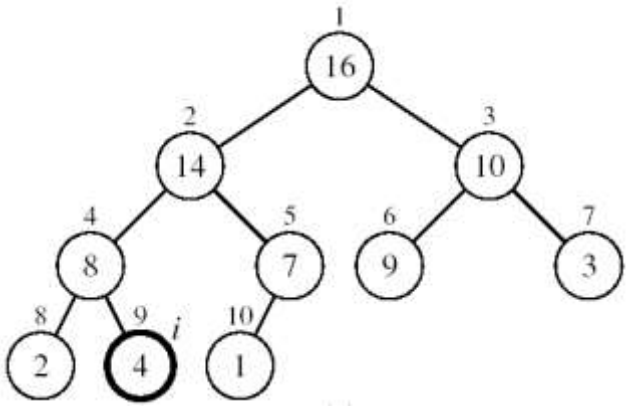
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



Heap property restored

Heap Operations: Heapify()

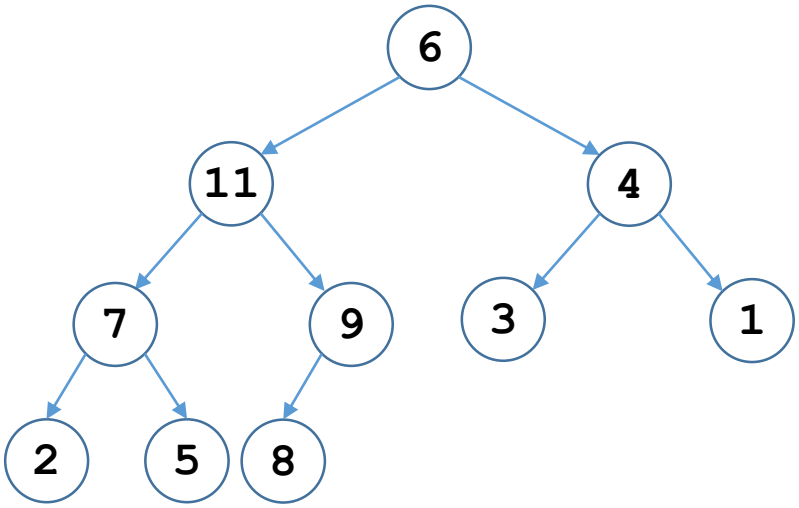
`Heapify()`: maintain the heap property

- **Given:** a node *i* in the heap with children *l* and *r*
- **Given:** two subtrees rooted at *l* and *r*, assumed to be heaps
- **Problem:** The subtree rooted at *i* may violate the heap property
- **Action:** let the value of the parent node "float down" so subtree at *i* satisfies the heap property
  - May lead to the subtree at the child not being a heap.
  - Recursively fix the children until all of them satisfy the max-heap property.

Illustration of Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

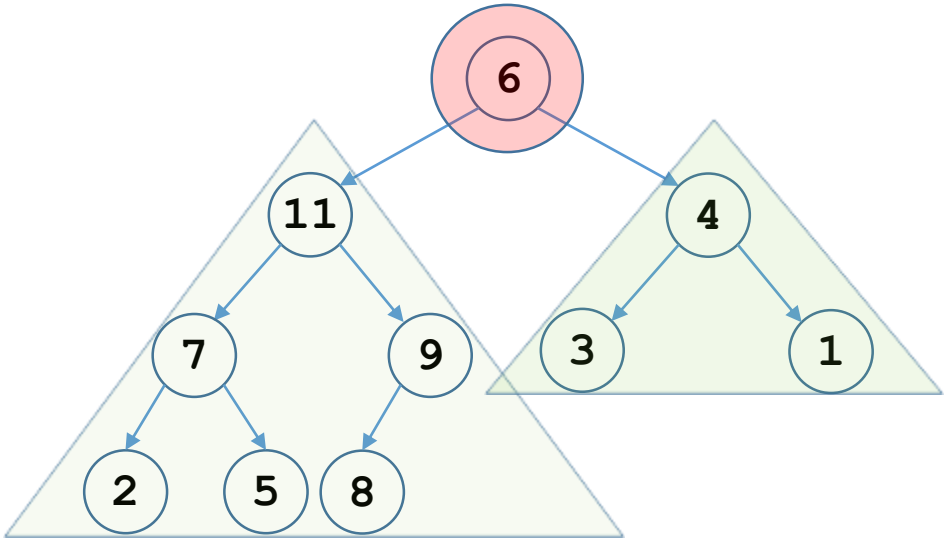
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

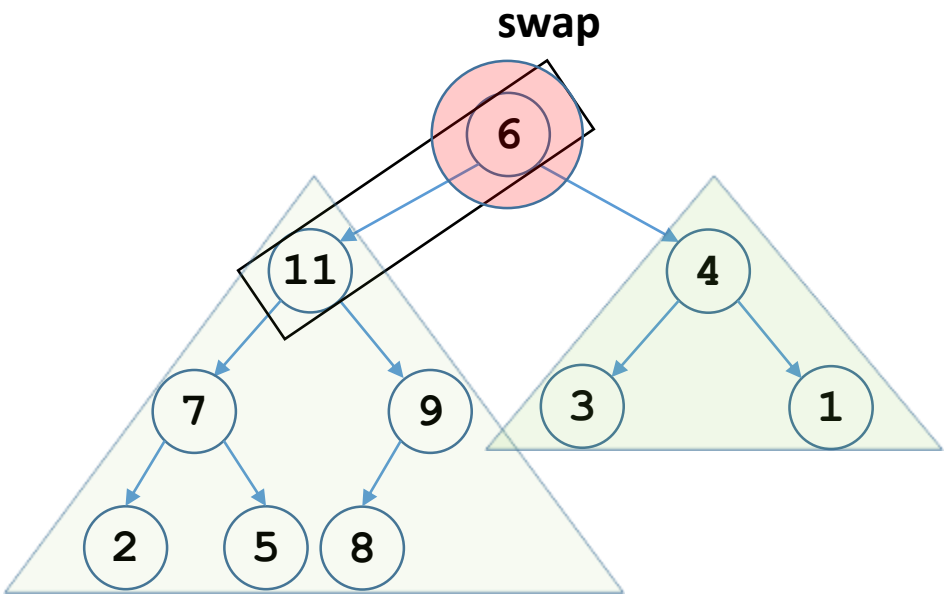
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

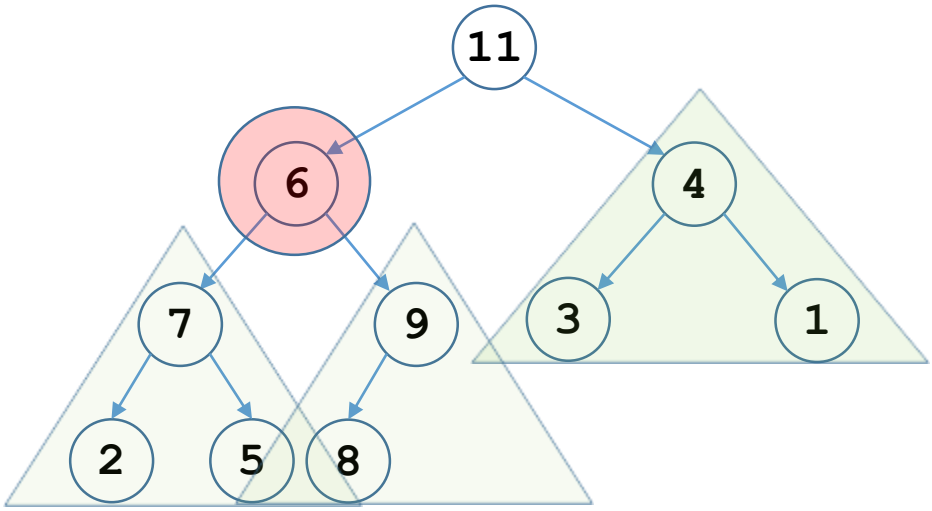
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

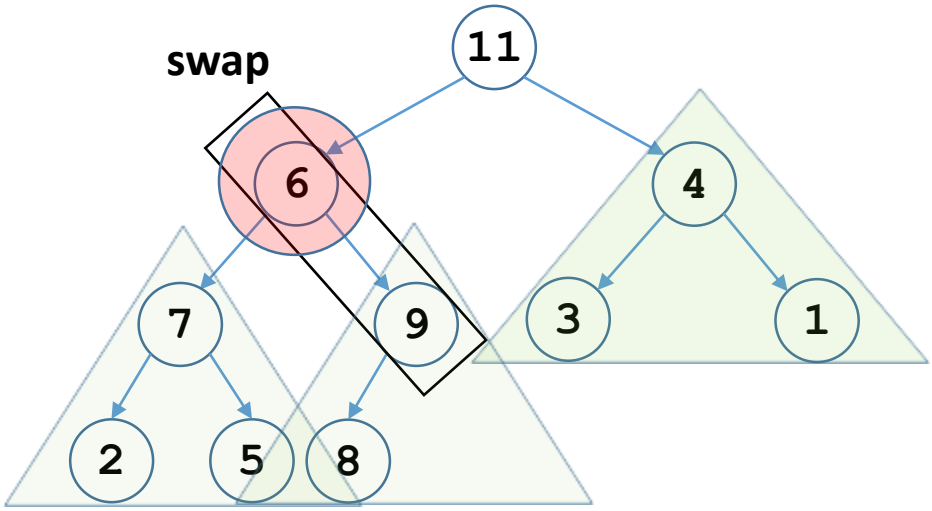
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

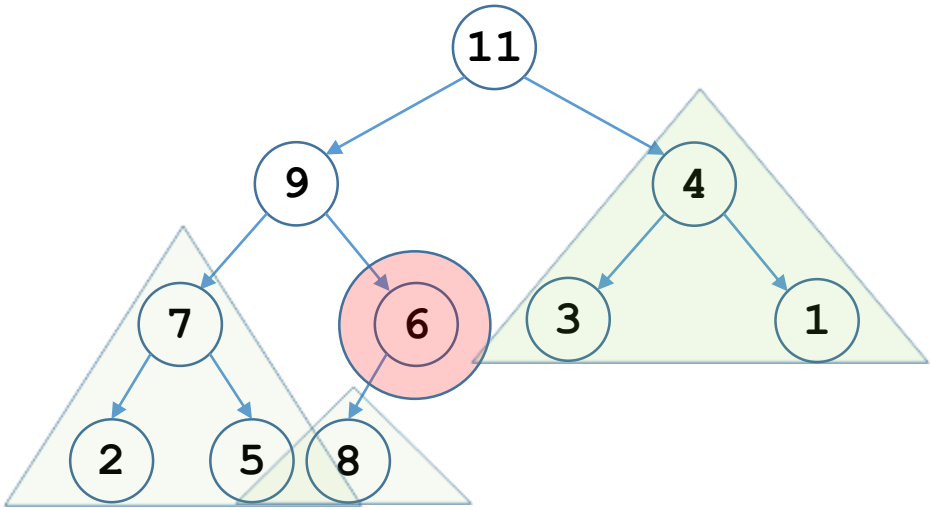
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

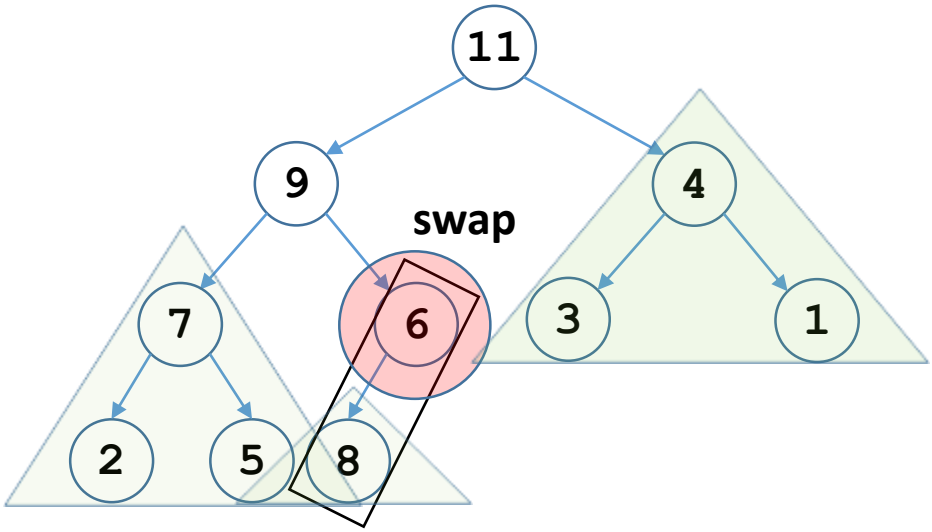
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

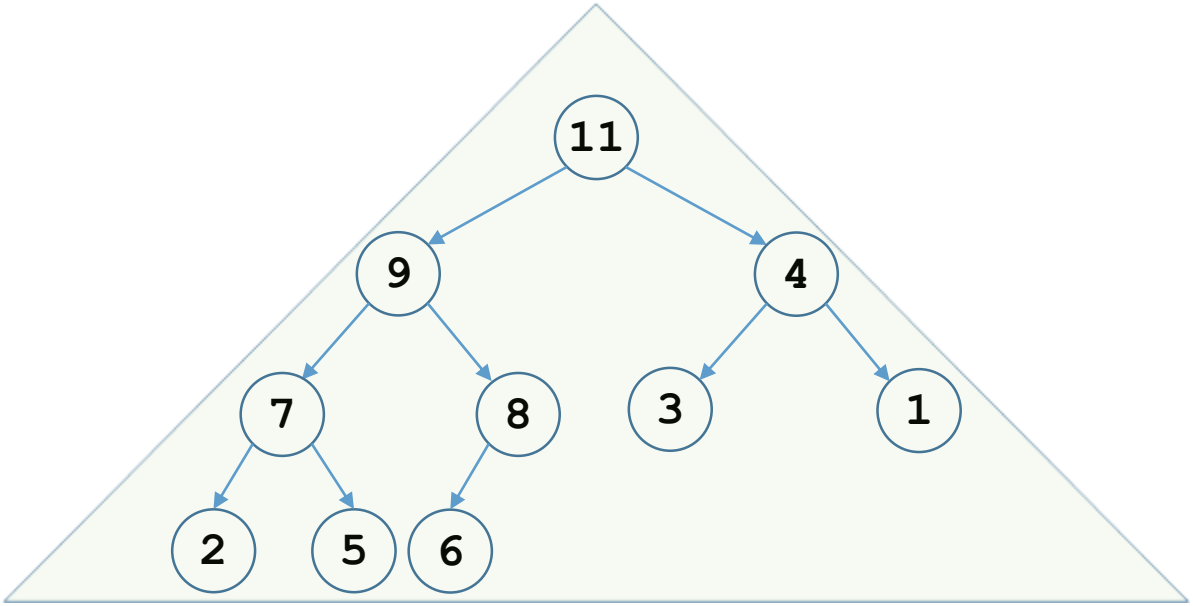
Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



The Heapify operation

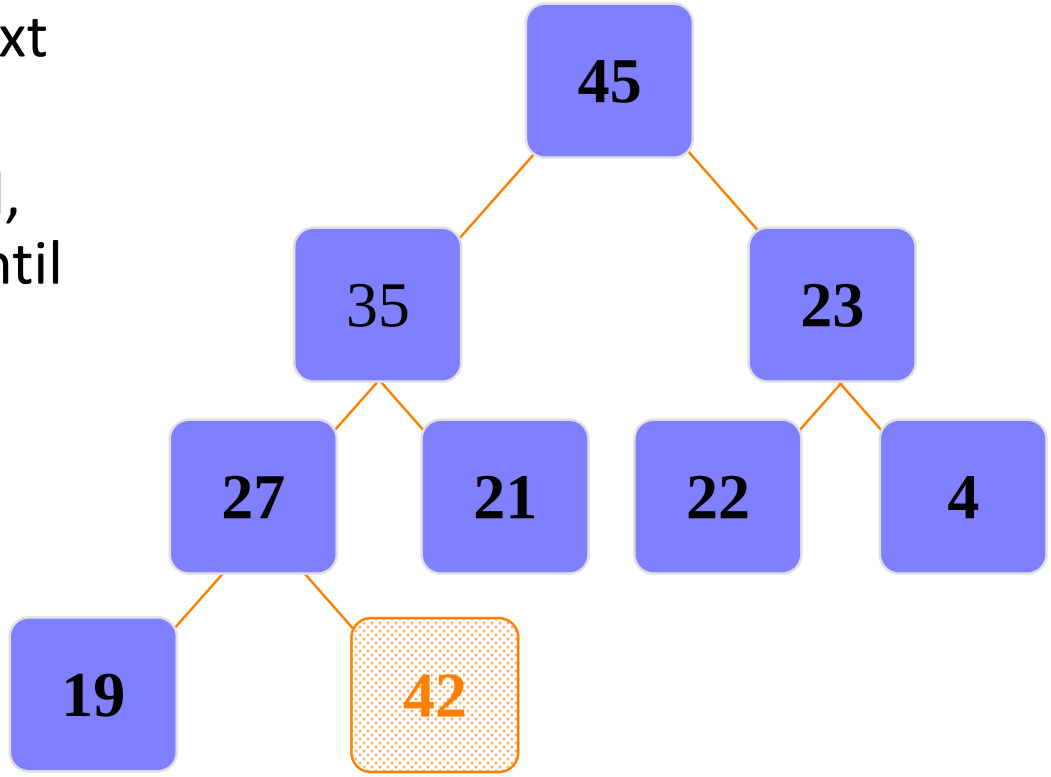
Suppose that, the order property is violated by the **root node** only (not any other node, they are in place)

Repair the structure so that it becomes a heap again (called Heapify operation), that is, move the element down from the root position until it ends up in a position where the heap property is satisfied



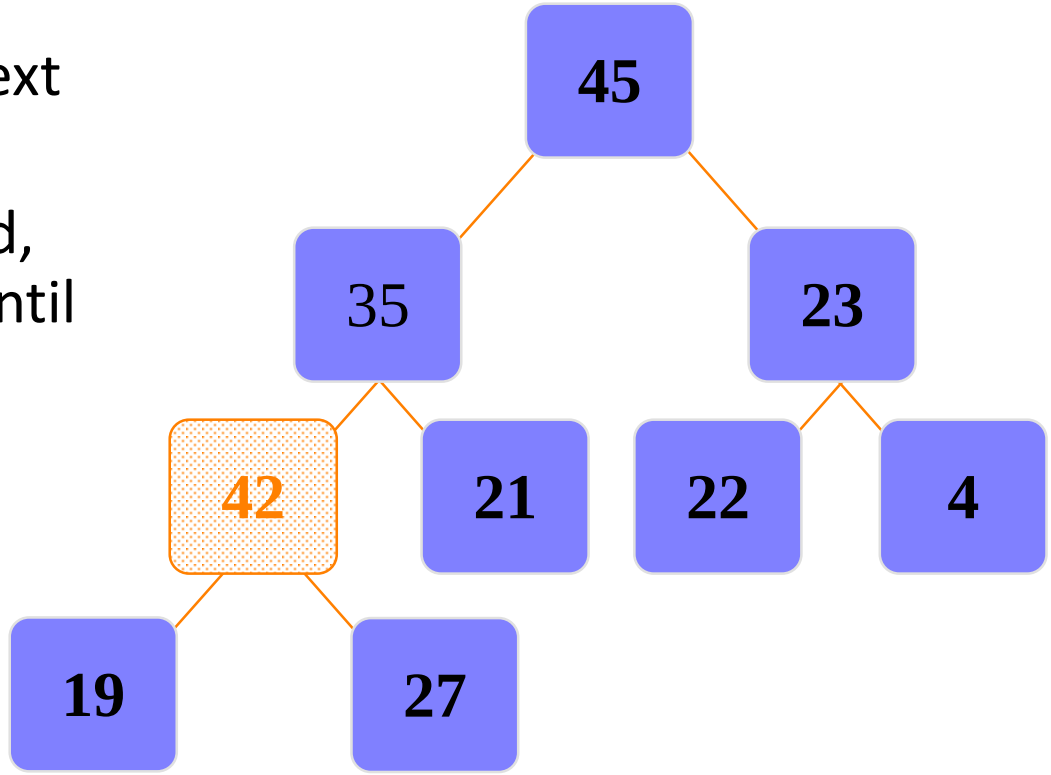
Adding a Node to a Heap

- 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.



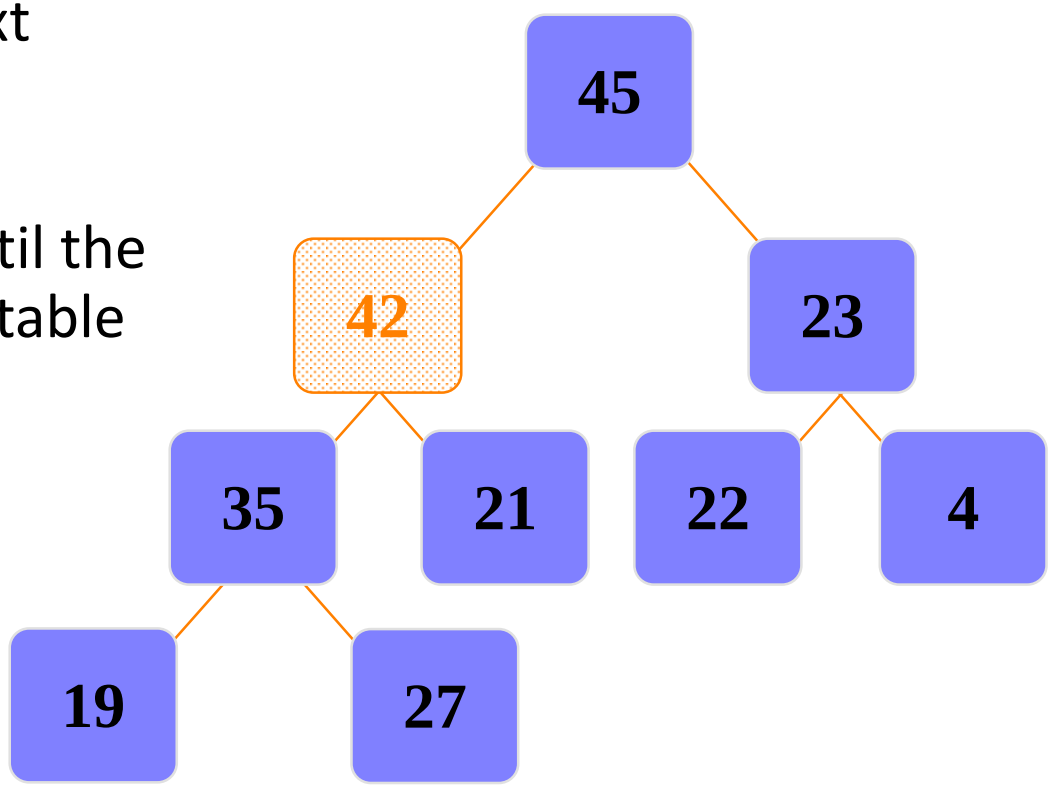
Adding a Node to a Heap

- 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.



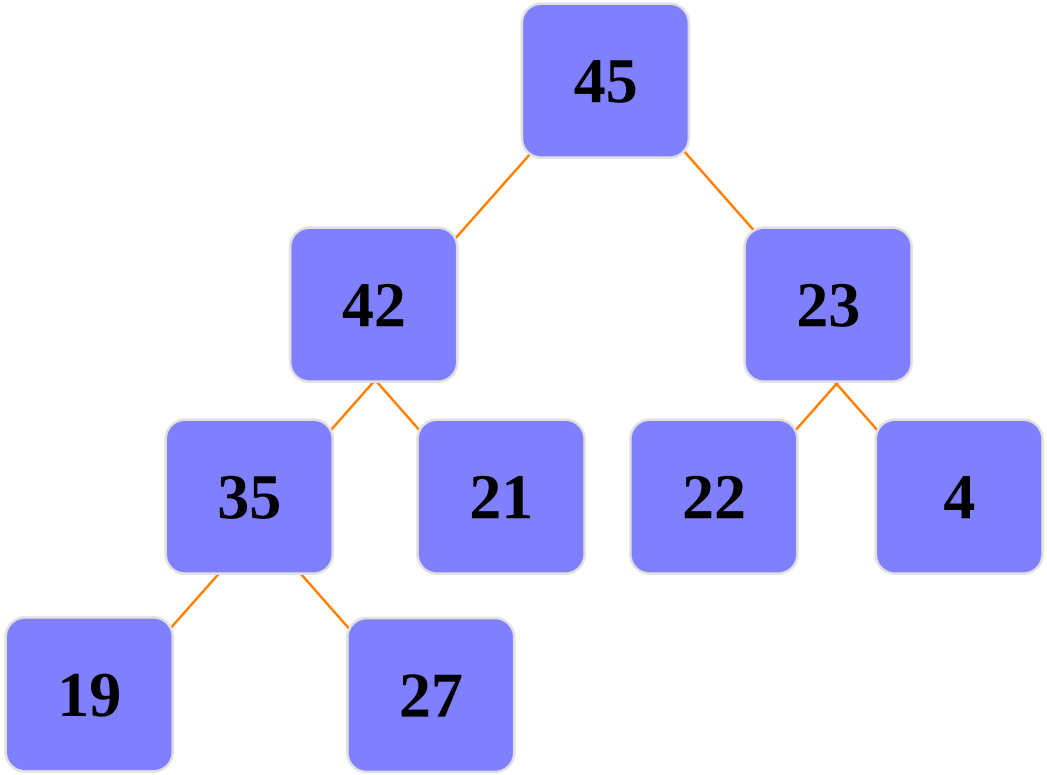
Adding a Node to a Heap

- 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.



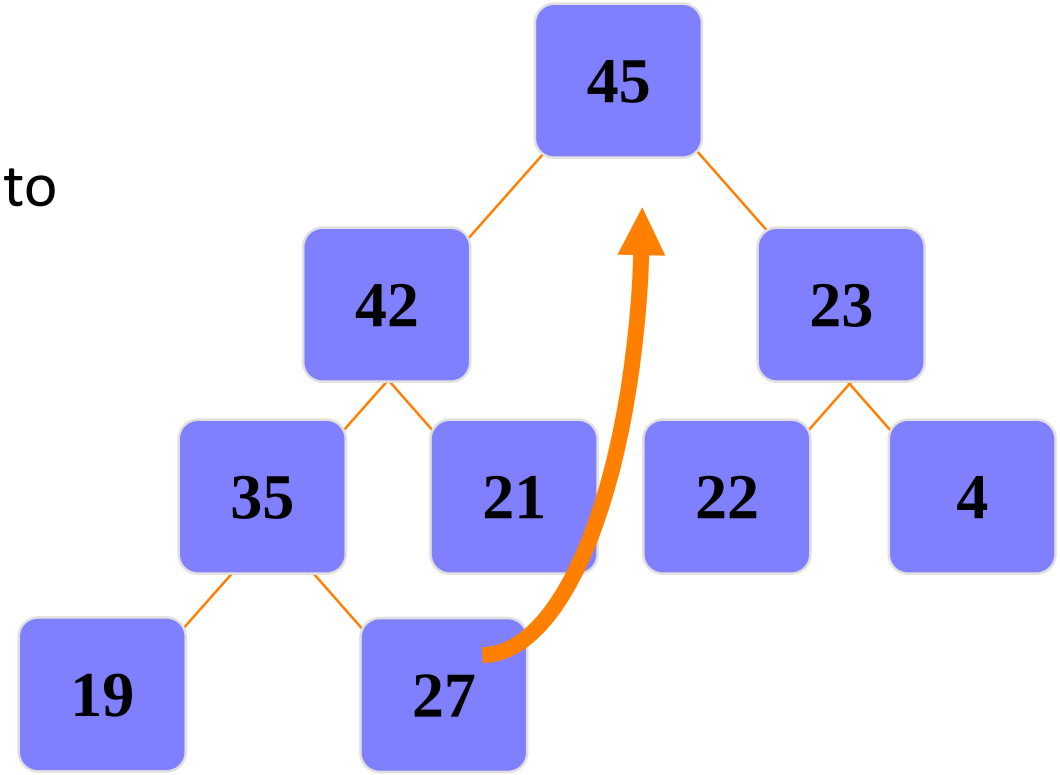
Adding a Node to a Heap

- 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 **reheapup**.



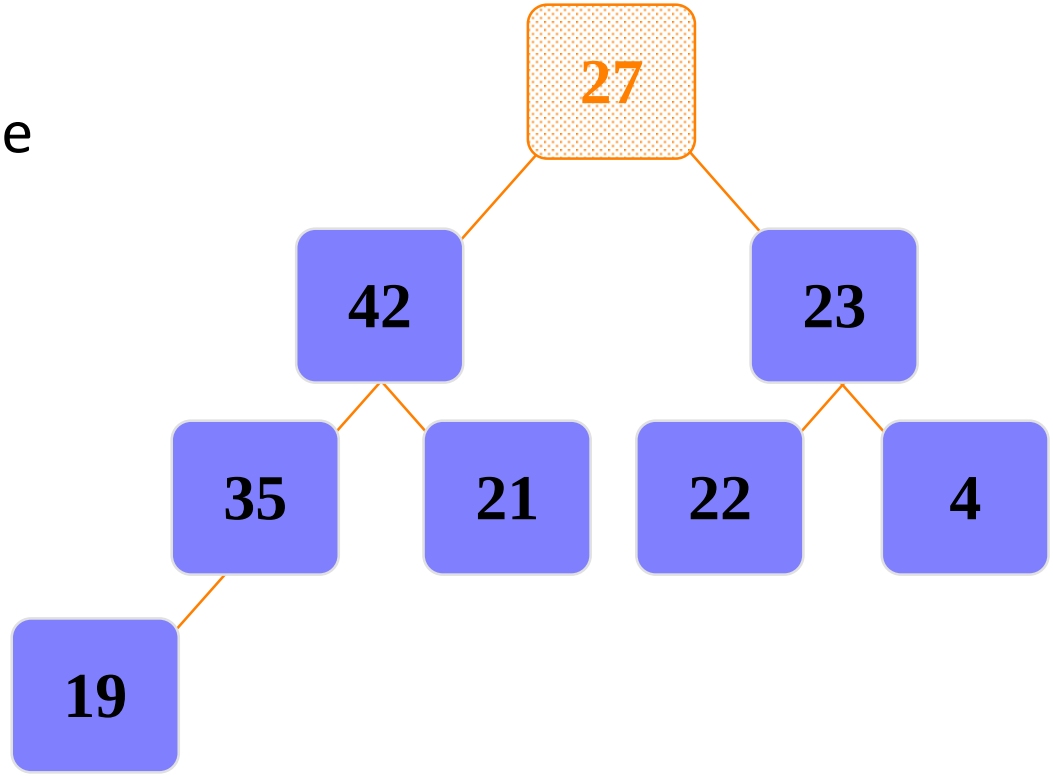
Removing the Top of a Heap

- Move the last node onto the root.



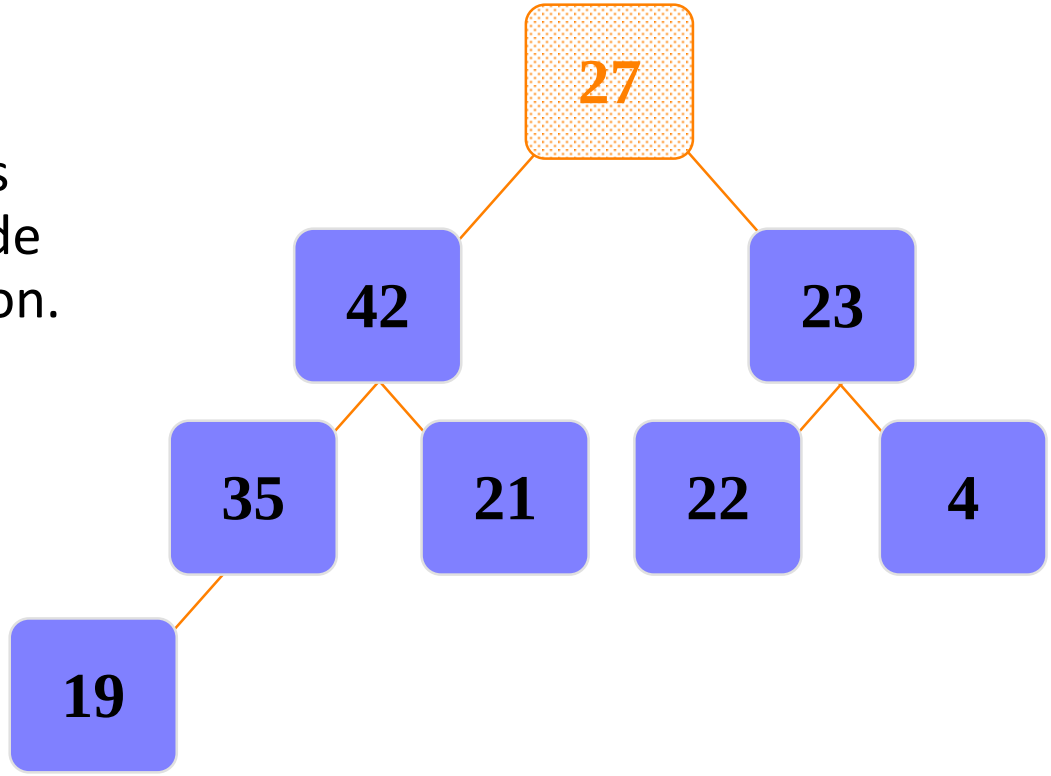
Removing the Top of a Heap

- Move the last node onto the root.



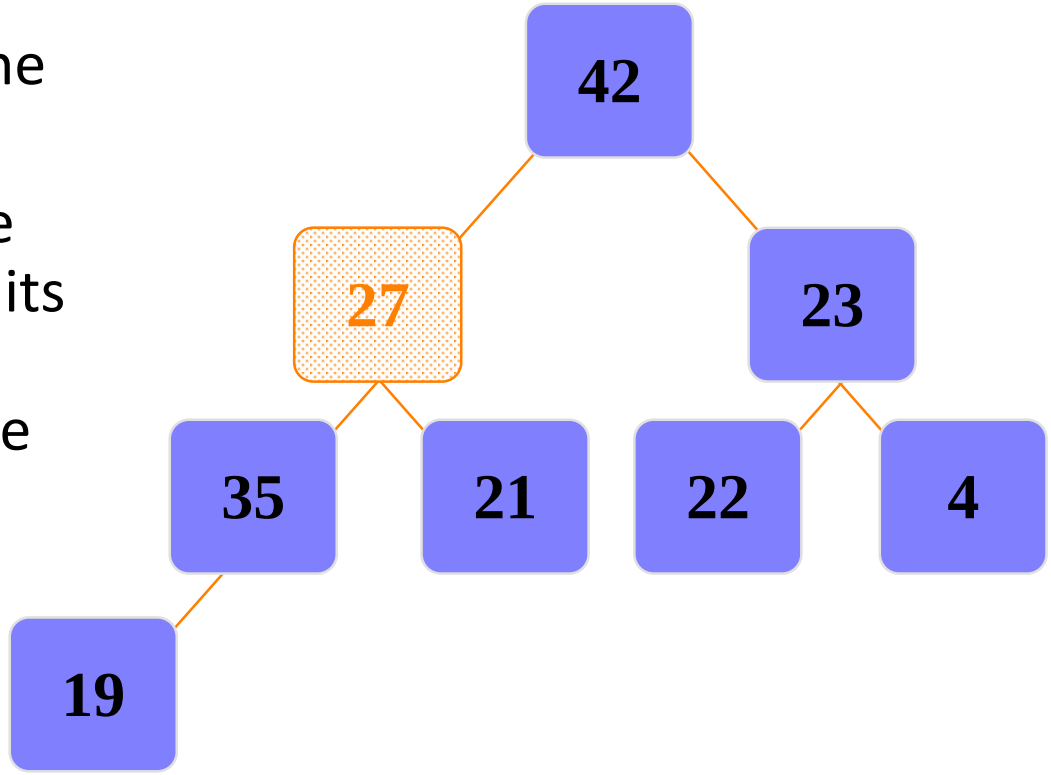
Removing the Top of a Heap

- 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.



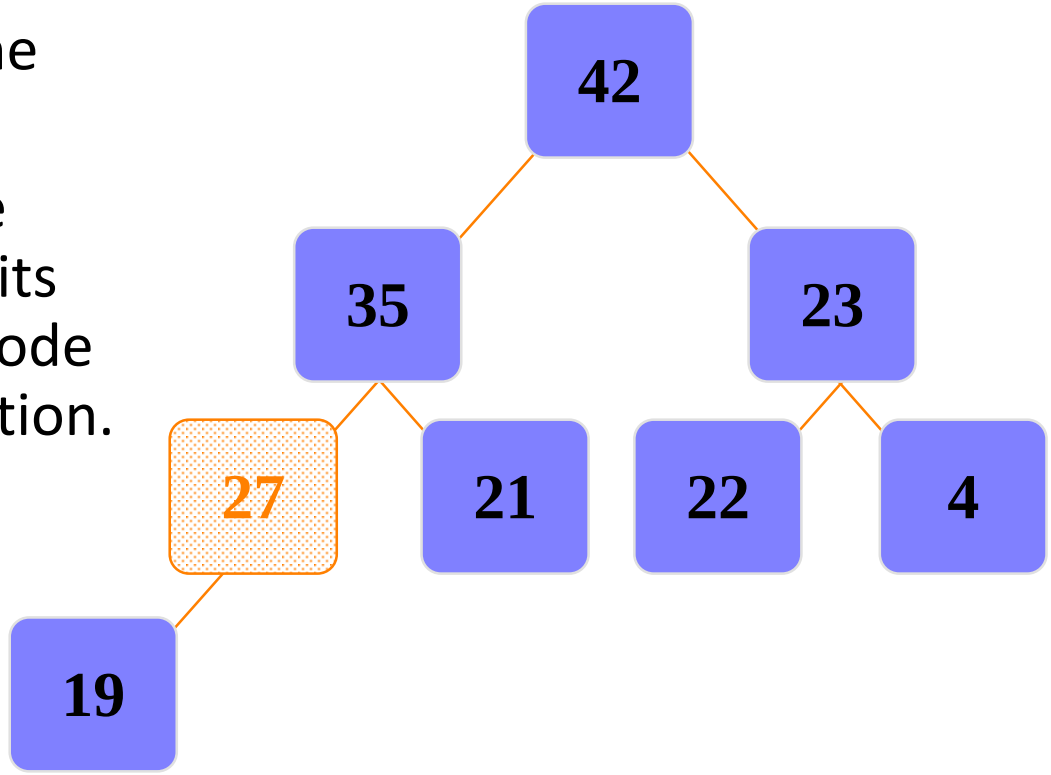
Removing the Top of a Heap

- 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.



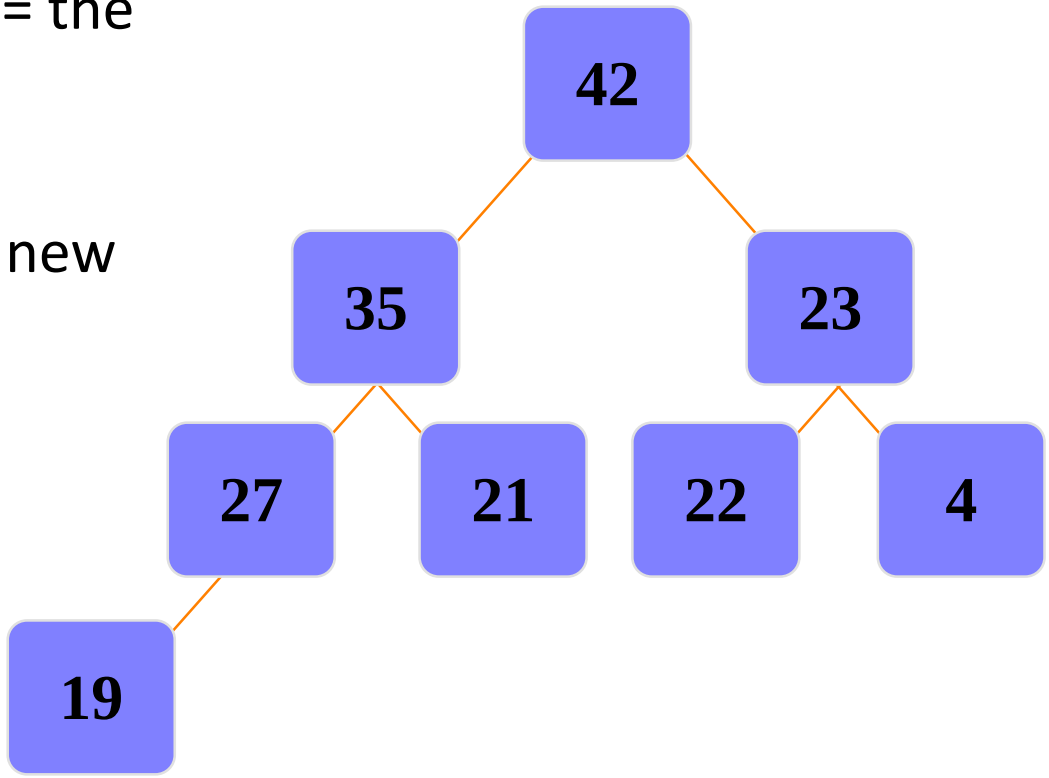
Removing the Top of a Heap

- 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 **reheapdown**.



AVL TREE

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."

Avl tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced.

The balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. This can be done by avl tree rotations

Need for AVL tree

- The disadvantage of a binary search tree is that its height can be as large as N-1

  This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case

We want a tree with small height

A binary tree with N node has height at least Q(log N)

Thus, our goal is to keep the height of a binary search tree O(log N)

Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree

Thus we go for AVL tree.

APPLICATIONS of AVL Tree

AVL trees play an important role in most computer related applications. The need and use of avl trees are increasing day by day. their efficiency and less complexity add value to their reputation.

Some of the applications are

- Contour extraction algorithm
- Parallel dictionaries
- Compression of computer files
- Translation from source language to target language
- Spell checker

DISADVANTAGES OF AVL TREE

one limitation is that the tree might be spread across memory

as you need to travel down the tree, you take a performance hit at every level down

one solution: store more information on the path

Difficult to program & debug ; more space for balance factor.

asymptotically faster but rebalancing costs time.

most larger searches are done in database systems on disk and use other structures

Priority Queues (PQ)

Properties:

- Each element is associated with a priority (key)
- The key with the highest (MAX-PQ)/lowest (MIN-PQ) key is extracted first
- Can be implemented as a Max-Heap/Min-Heap

An application: Schedule jobs on a shared resource

- PQ keeps track of jobs and their relative priorities
- When a job is finished or interrupted, highest priority job is selected from those pending using EXTRACT-MAX function
- A new job can be added at any time using INSERT function

Applications of Priority Queue

Graph algorithms:

- Dijkstra's and A*-search algorithms to compute shortest paths
- Prim's algorithm to compute minimum spanning tree (MST) of a graph

Others:

- Huffman coding for compressing data
- Resource scheduling in operating systems