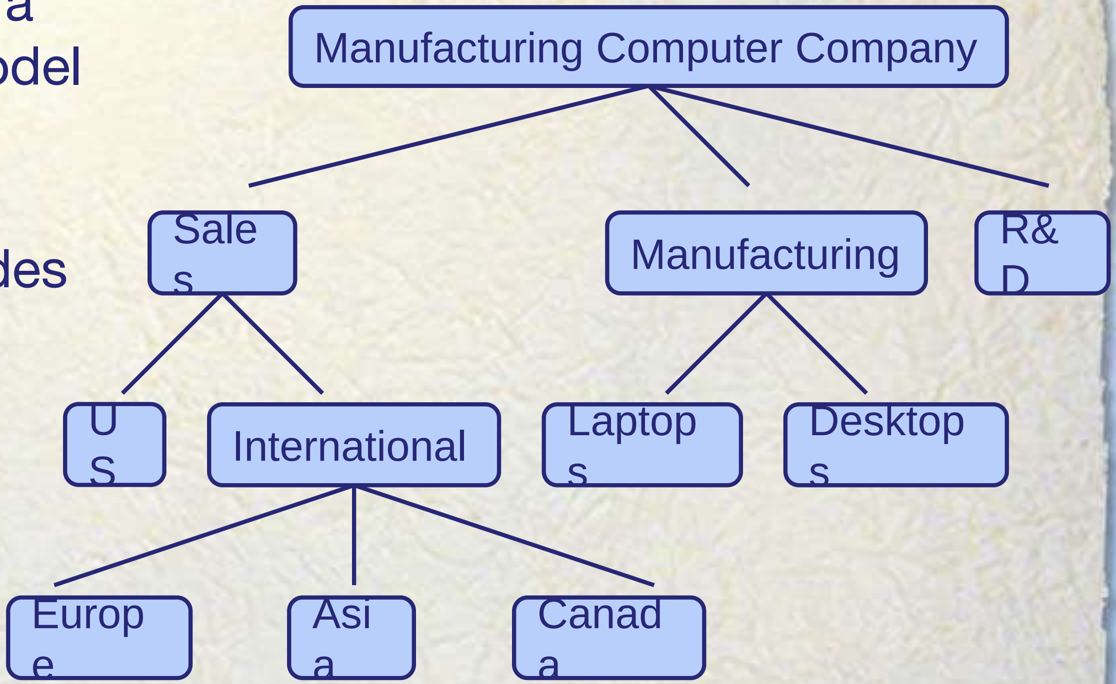
Data Structures and Algorithms

Trees – Part 2

University of Technology and Engineering
Vietnam National University Hanoi

What is a Tree

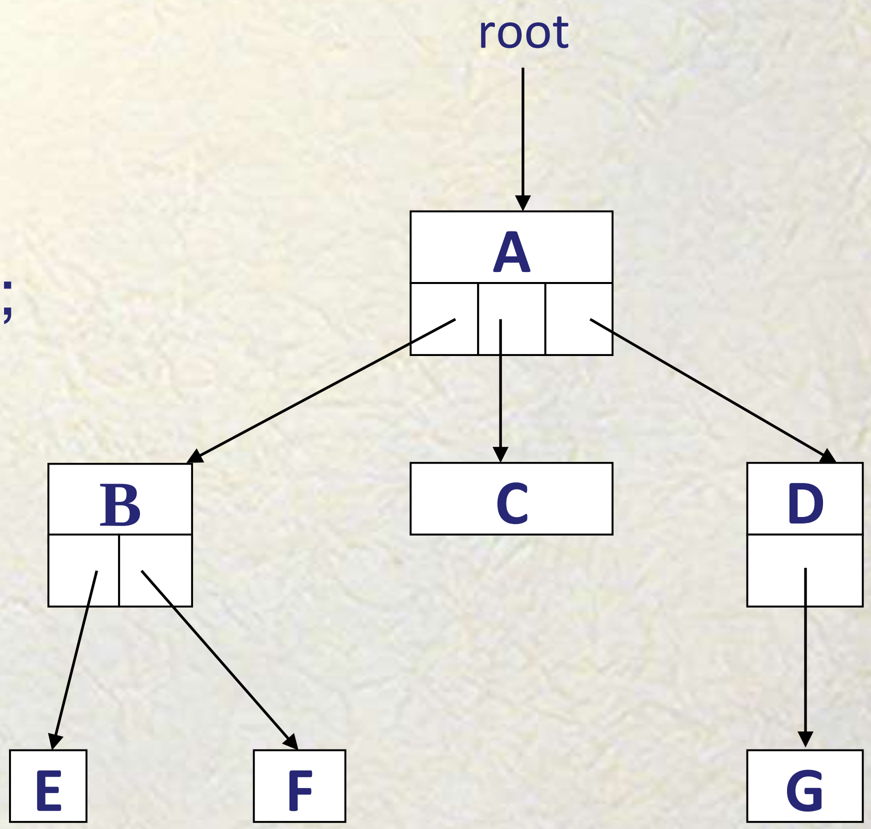
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



List of Children Tree Presentation

```
Template <class Item>
class Node {
    Item data;
    List<Node*> children;
}

Node<Item>* root;
```



Priority Queue

- A priority queue stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMax()
    removes and returns the entry with smallest key

- Example:

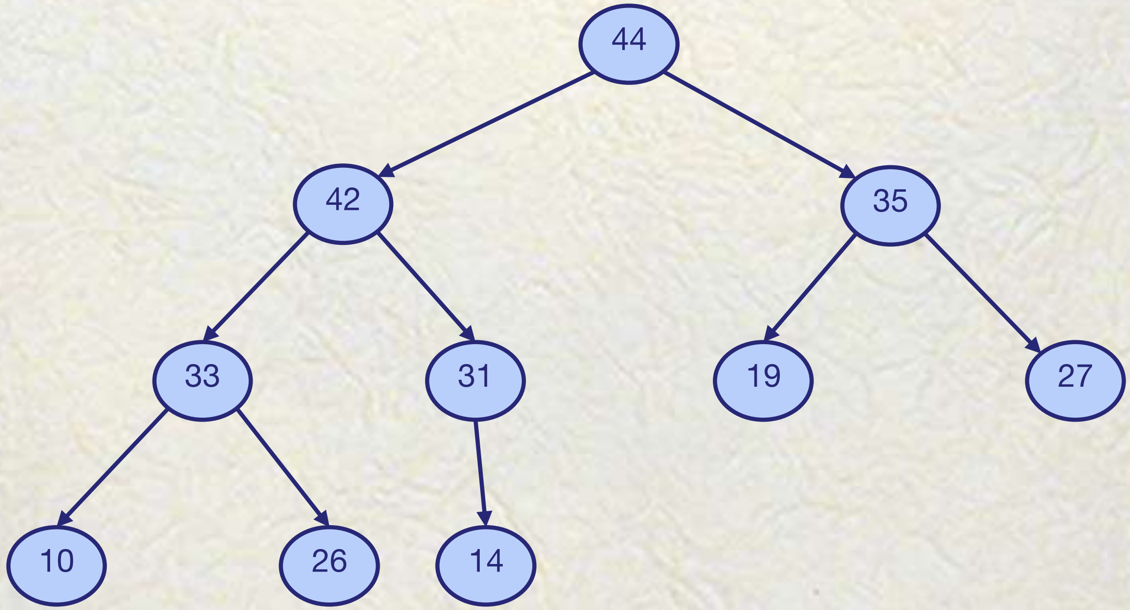
Priority Queue

- Additional methods
  - max()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

- Applications:
  - Standby flyers
  - Auctions
  - Stock market

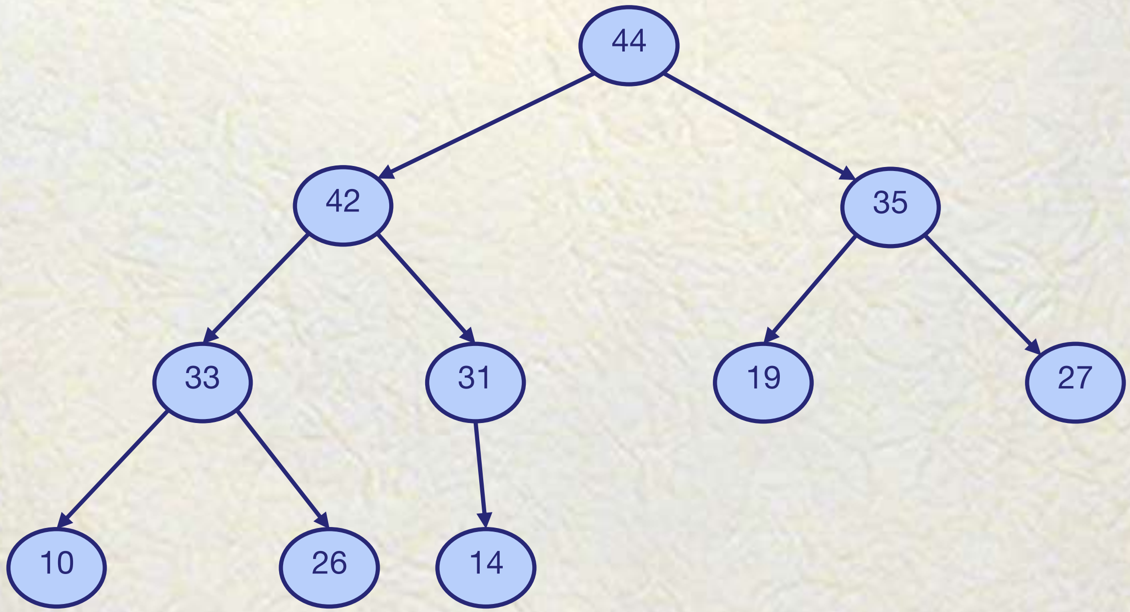
Heap tree

- Heap tree is a binary tree where the value of any internal node is greater or equal to theirs children
- Application: Build the priority queue



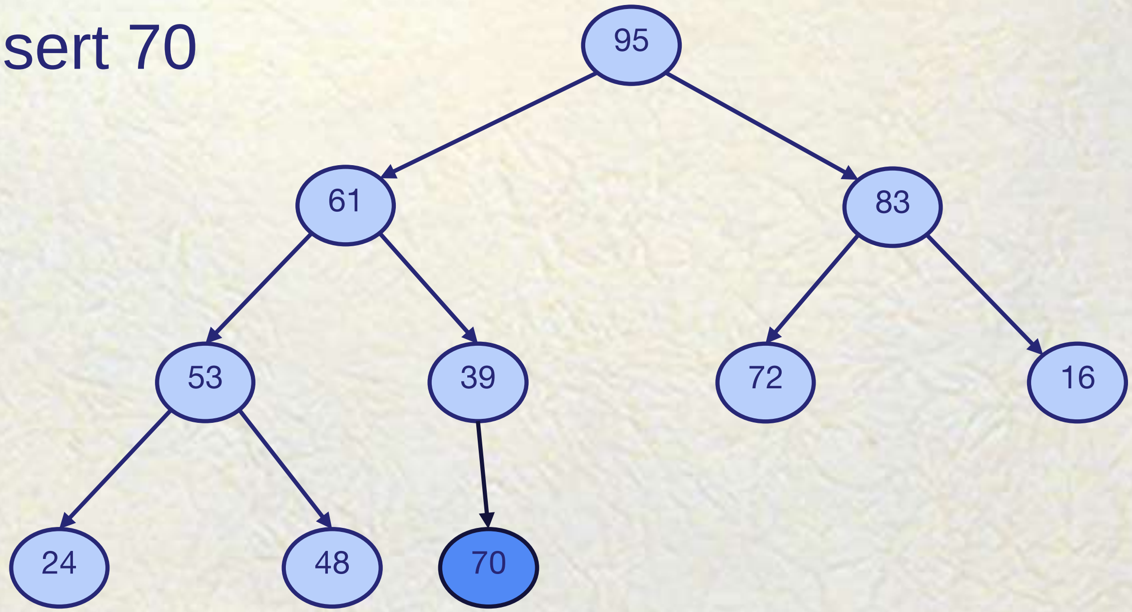
Heap tree

Max operation: get the node with maximum value (the root)



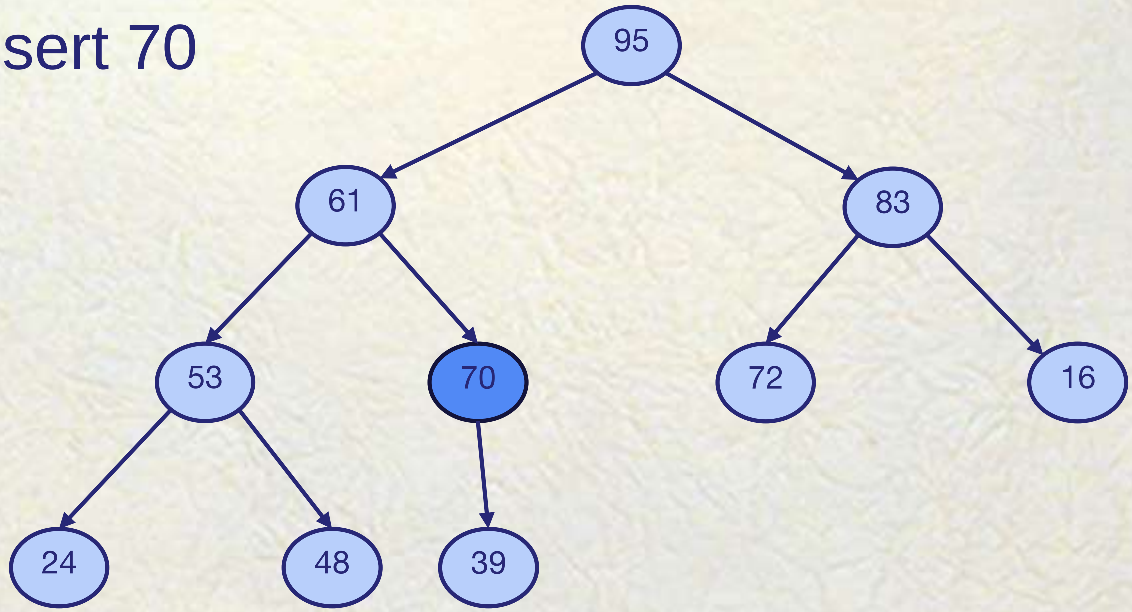
Heap tree insertion

Insert 70



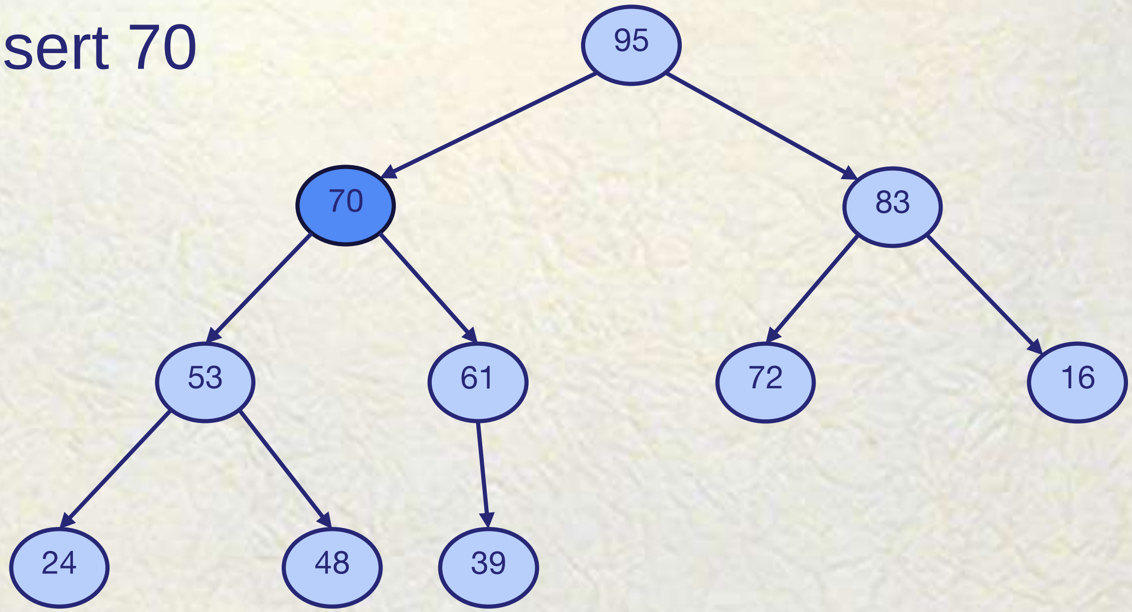
Heap tree insertion

Insert 70



Heap tree insertion

Insert 70



Heap tree insertion

Algorithm *insert*(*v*):

➢ Step 1 – Create a new node at the end of heap.

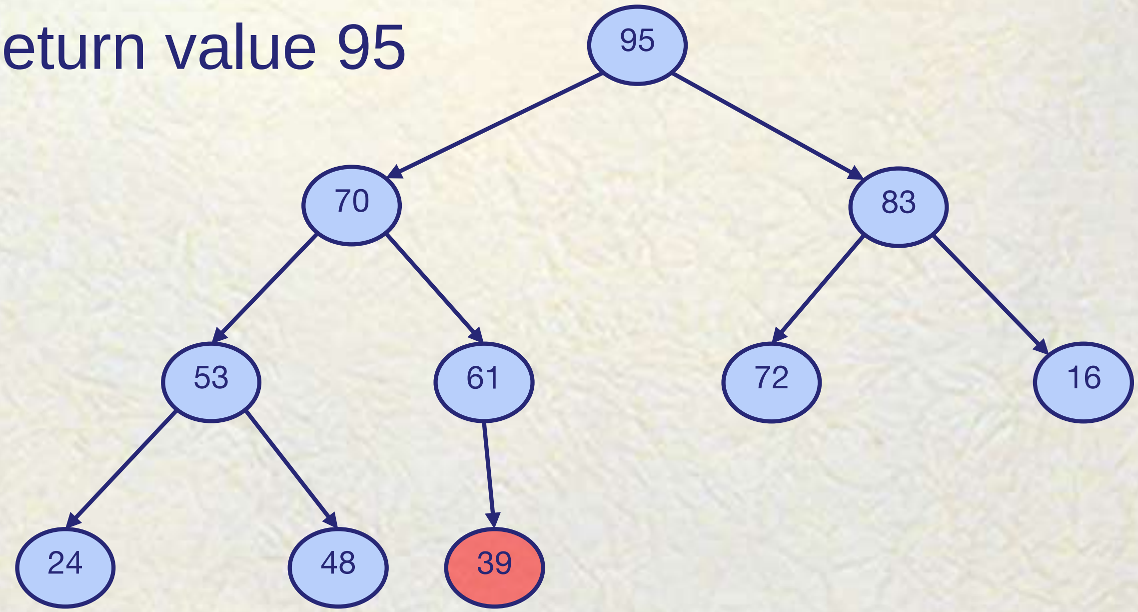
➢ Step 2 – Start the new node, compare the value at this node to its parent. If it is larger than its parent, swap them, move up and continue Step 2.

Exercise 2

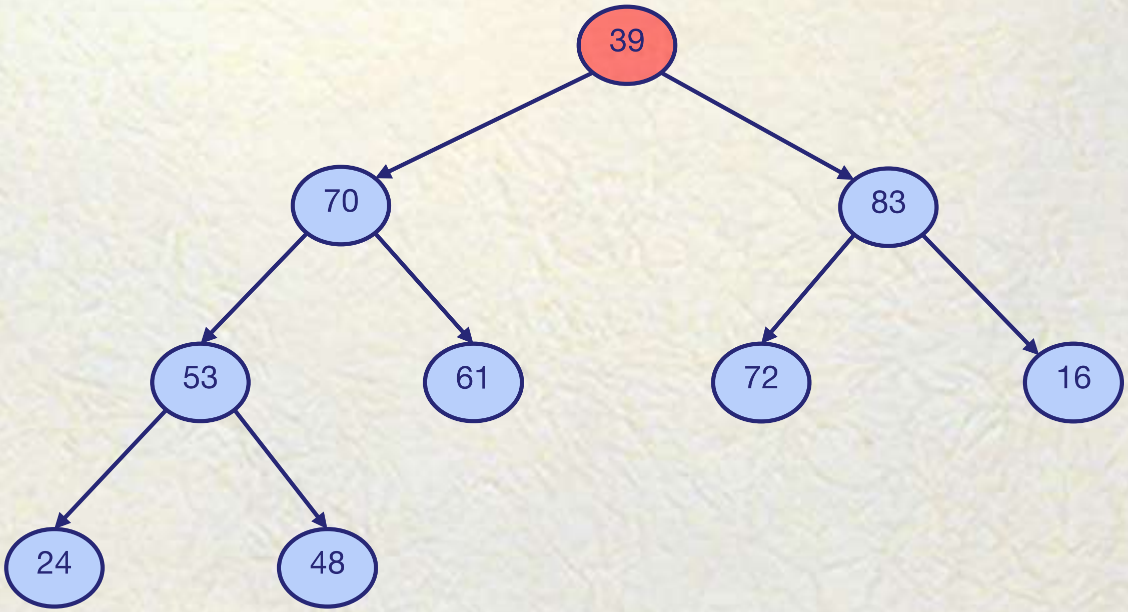
- Construct a max heap tree including: 52, 69, 38, 79, 66, 64, 72, 3, 16, 89, 15, 37, 0, 28, 73, 95.
- Insert the following numbers into the above max heap tree: 5, 3, 9, 7, 2, 4, 6, 1, 8.

Remove max

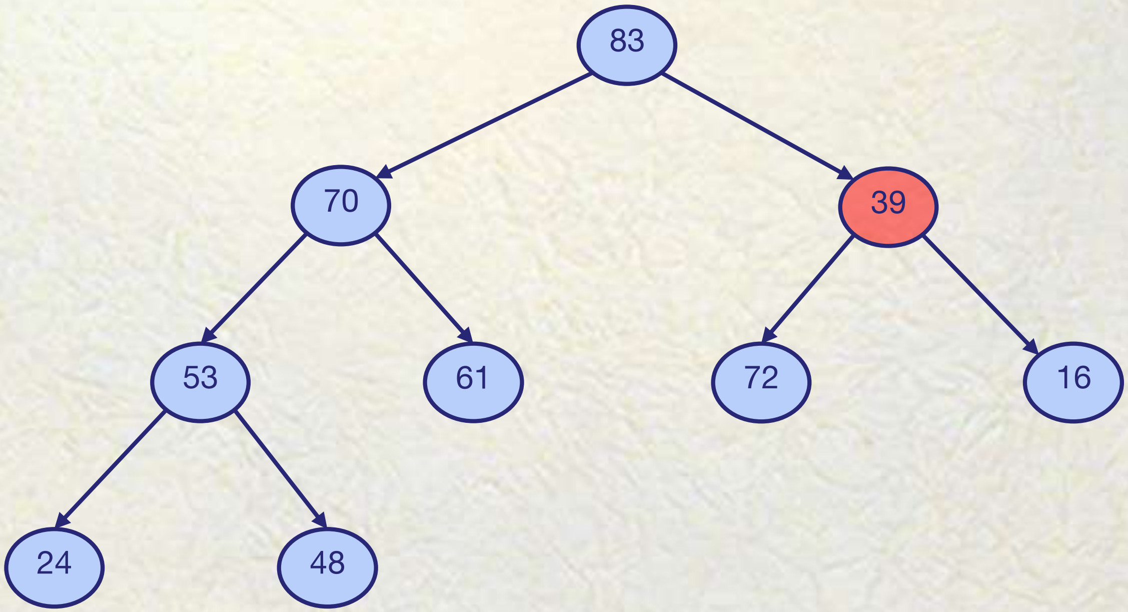
Return value 95



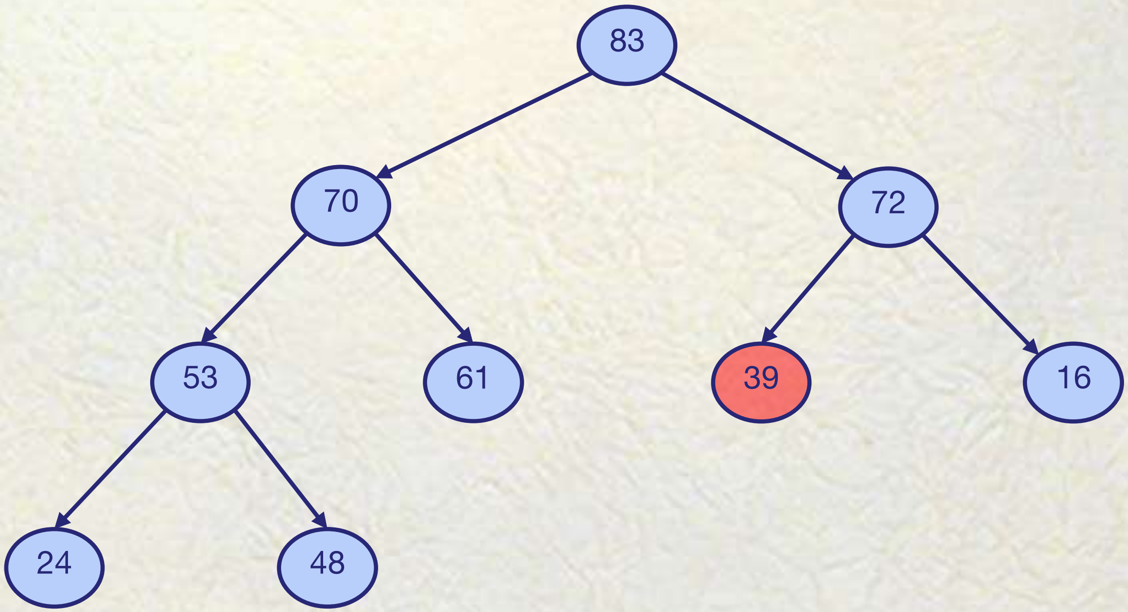
Remove max



Heap tree deletion



Remove max



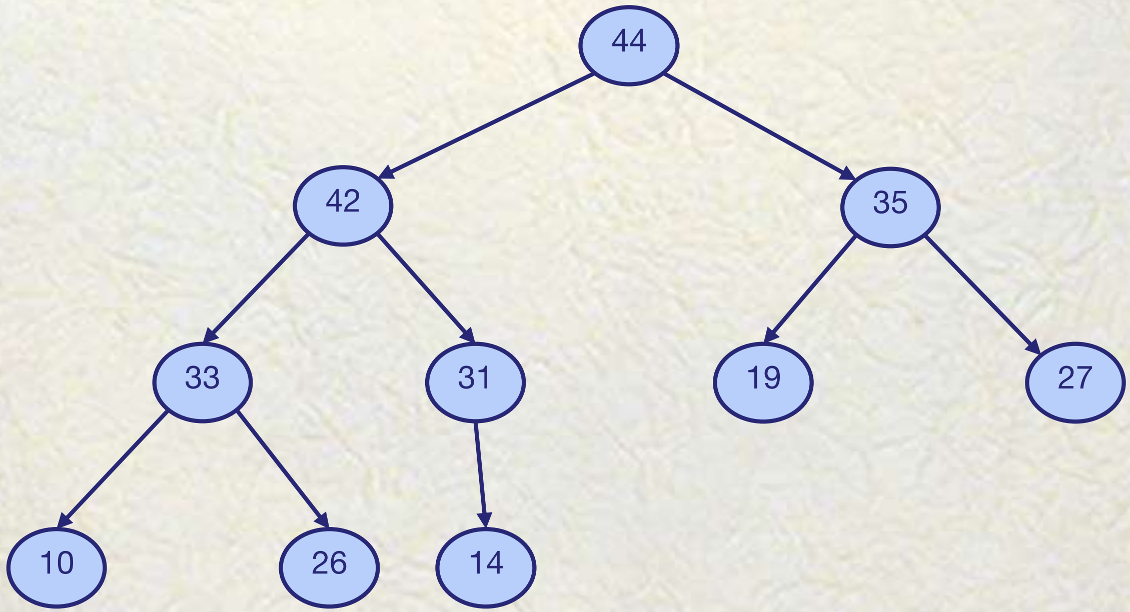
Remove max

Algorithm *remove_max* (*T*)

- Step 1 – Remove the root node.
- Step 2 – Move the last element of the heap to the root
- Step 3 – Start from the root, compare the value at the node to their children. If it is smaller than the largest child, swap them, move down and continue Step 2.

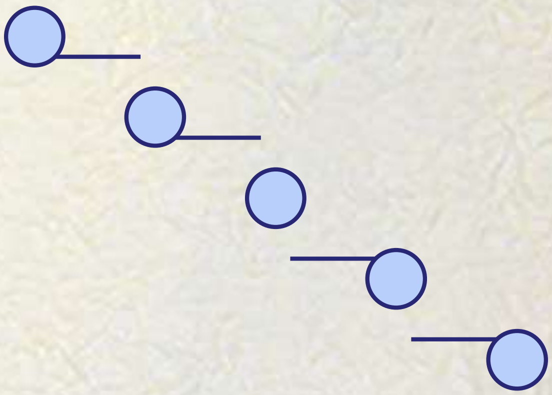
Exercise 3

Describe step by step of removing max from the following max heap tree



Heap tree performance

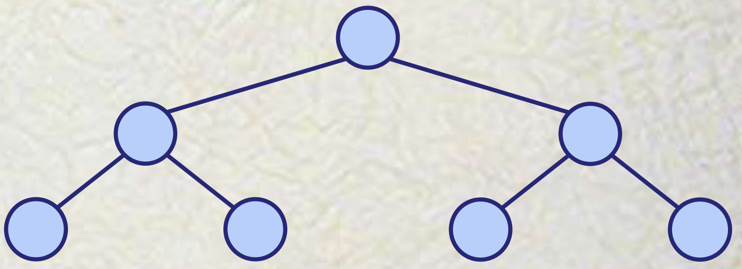
- Max operation: O(1)
- Insertion operation: The height of the tree
- Deletion operation: The height of the tree



The height of the tree:

❖ Worse case: O(n)

❖ Best case: O(log n) when the heap tree is balanced



Balance the heap tree: If insertions and deletions make the heap tree unbalanced, perform balancing operations to make balanced again.

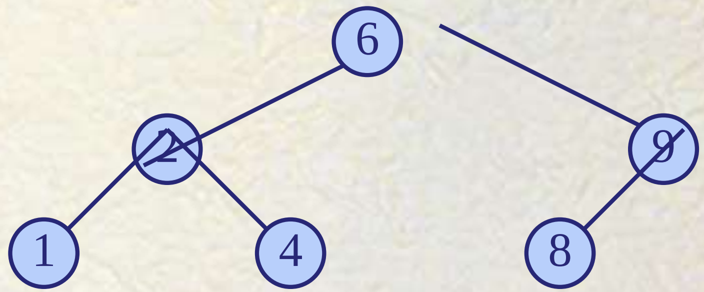
Using heap tree library

Using make_heap, pop_heap, push_heap, and sort_heap from the algorithm library of C++ programming language to implement the following tasks:

- ❖ Make a heap tree
- ❖ Remove the max from the heap tree
- ❖ Insert a node to the heap tree
- ❖ Sort elements in the heap tree

Binary Search Trees

- A binary search tree is a binary tree such that the value at the parent node is larger or equal to values of the left child, and smaller or equal to values of the right child.

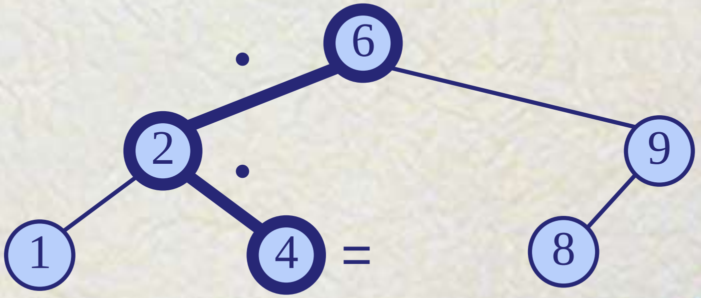


- An inorder traversal of a binary search trees visits the keys in increasing order

Search

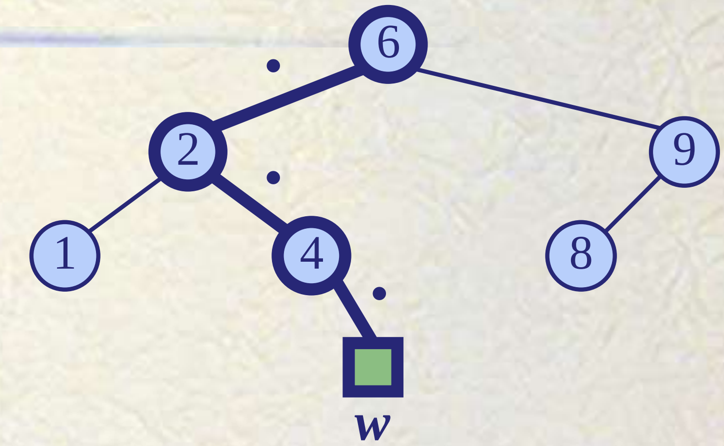
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return null
- Example: find(4):
  - Call TreeSearch(4,root)

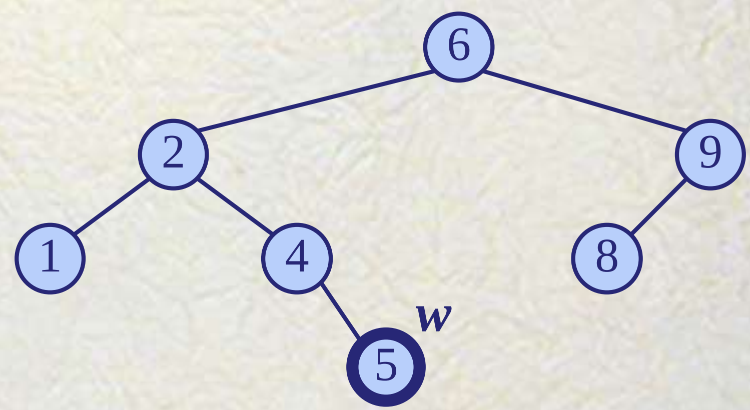
```
Algorithm TreeSearch(k, v)
        if T.isExternal (v)
        return v;
    if k • key(v)
        return TreeSearch(k, T.left(v));
    else if k = key(v)
        return v;
    else { k • key(v); }
        return TreeSearch(k, T.right(v));
```



Insertion

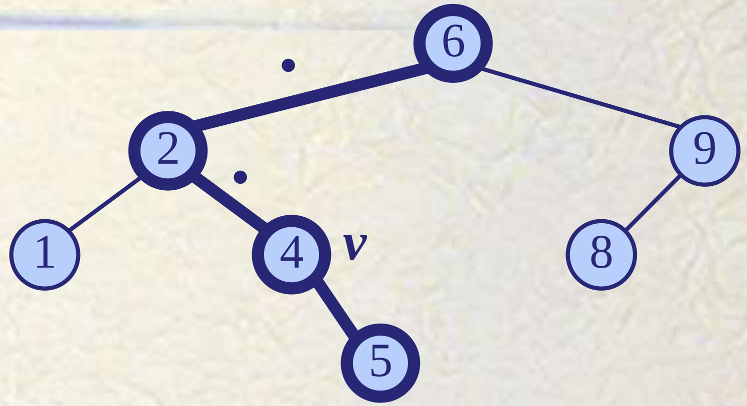
- Insert a value k into the binary tree.
- Algorithm: Start from the root, compare k to the value at this node. If k is smaller, insert k into the left tree, otherwise insert k into the right tree.
- Example: insert 5

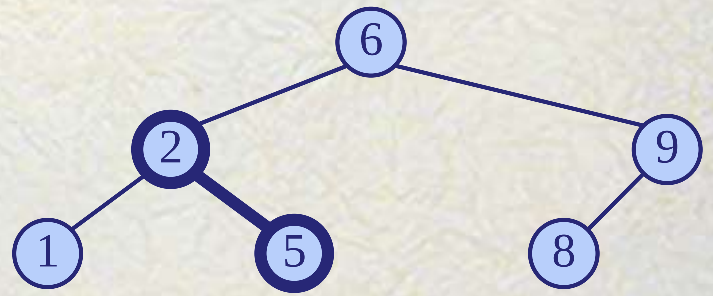




Deletion

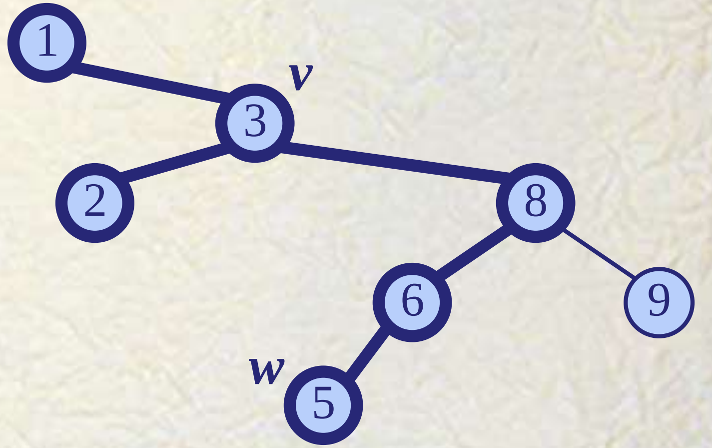
- To perform operation remove(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let let ***v*** be the node storing ***k***
- If node ***v*** is a leaf, remove v. If v has one child, remove v and connect its child to its parent.
- Example: remove 4

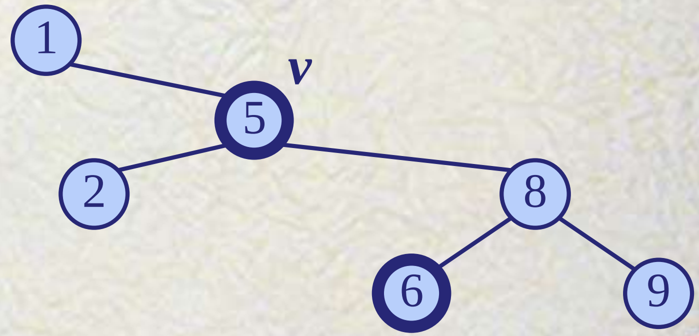




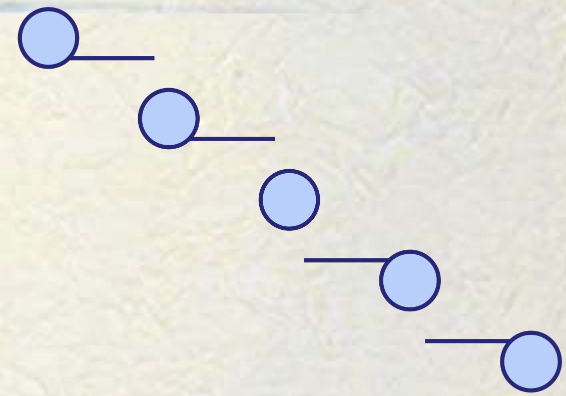
Deletion (cont.)

- We consider the case where the key ***k*** to be removed is stored at a node ***v*** with two children:
  - we find the node ***w*** that follows ***v*** in an inorder traversal (the most-left leaf of the right child of v).
  - we replace node ***v* by *w***
  - we remove node ***w***
- Example: remove 3





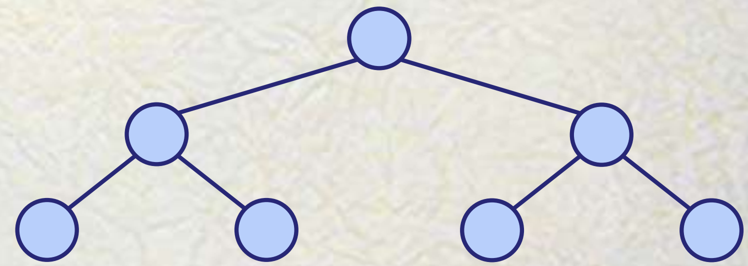
Performance



Operations: Searching, insertion, deletion: The height of the tree

The height of the tree:

- ➢ Worse case: O(n)
- ➢ Best case: O(log n) when the heap tree is balanced



Balance the binary search tree: If insertions and deletions make the binary search tree unbalanced, perform balancing operations to make it balanced again.

Exercise 4

- Create a binary search tree from following numbers: 34, 15, 65, 62, 69, 42, 40, 80, 50, 59, 23, 46, 57, 3, 29
- Draw BSTs after deleting keys 62, 42 and 3 from the above tree.

Exercise 5

- Draw the BST for items with keys
  E A S Y Q U E S T I O N

- Draw the BST for items with keys
  D A T A S T R U C T R E S A N D A L G O R I T H M