Algorithms and Data Structures

Module 2

Lecture 9

Priority queue implementations. 2-3 trees. Binary heaps.

Priority queue: definition

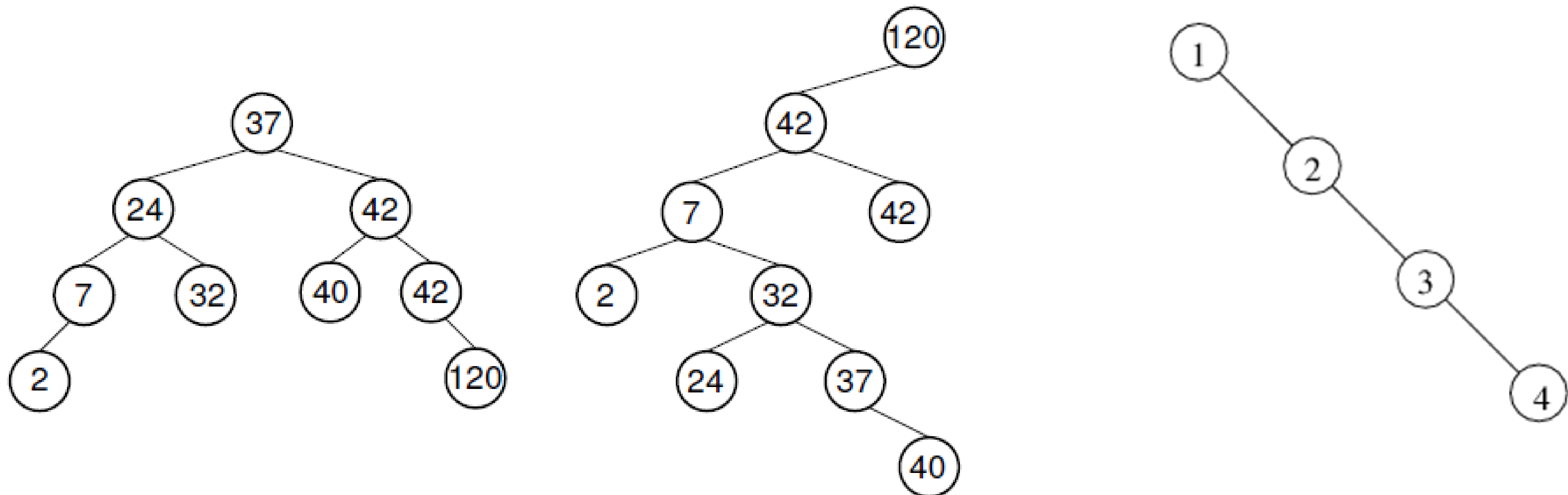
Priority queue is an abstract data structure which efficiently implements operations:

- `Init(n)` – initialize an empty priority queue with *n* possible items.
- `Build(S)` – build priority queue containing items of S.
- `Add(x, prior)` – add item *x* with priority *prior* to the priority queue.
- `GetMin()` / `GetMax()` – get the item with the highest priority.
- `DelMin()` / `DelMax()` – delete the item with the highest priority.
- `ChangePriority(x, new_prior)` – change the priority of *x* to *new_prior*.

Priority queue: binary search tree

Summary of time complexity for BST: `GetMin`, `DelMin`, `Add` have time complexity $O(h)$, where h is the height of the BST.

Height is $O(\log n)$ on average but $O(n)$ in the worst case ☹



Priority queue: tree-based implementations

Conclusion: since the complexity of priority queue implementations using trees is $O(h)$, we need balanced trees implementation to achieve $O(\log n)$ worst case complexity for priority queue operations.

A tree is called *balanced* iff its height is $O(\log n)$.

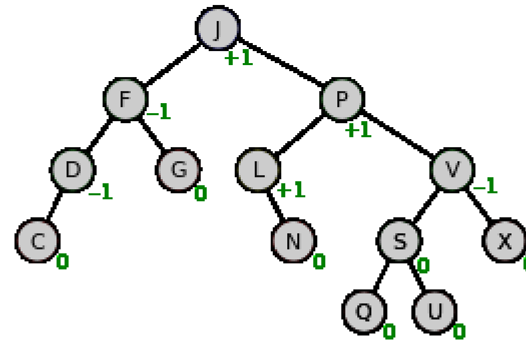
In order to build a balanced tree, we need to

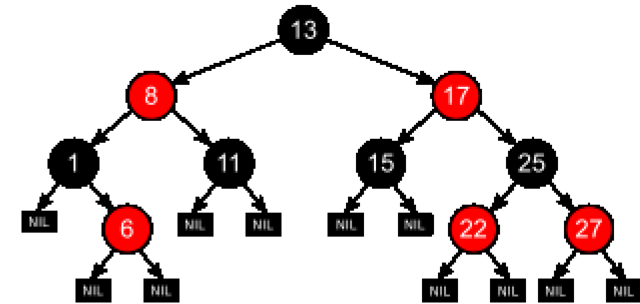
1) Keep additional information (subtrees' heights) for the vertices of the tree.
2) Rebuild (*rebalance*) the tree when it becomes unbalanced after some operations.

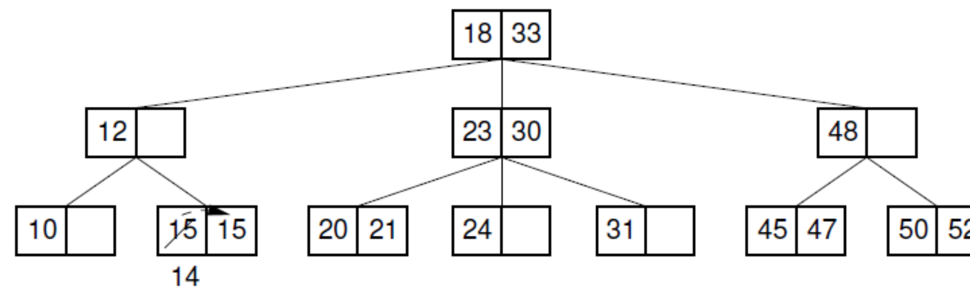
Priority queue: tree-based implementations

Balanced trees types:

- AVL trees
- Red-black trees
- <u>2-3 trees</u>
- …







2-3 trees

2-3 trees have the properties

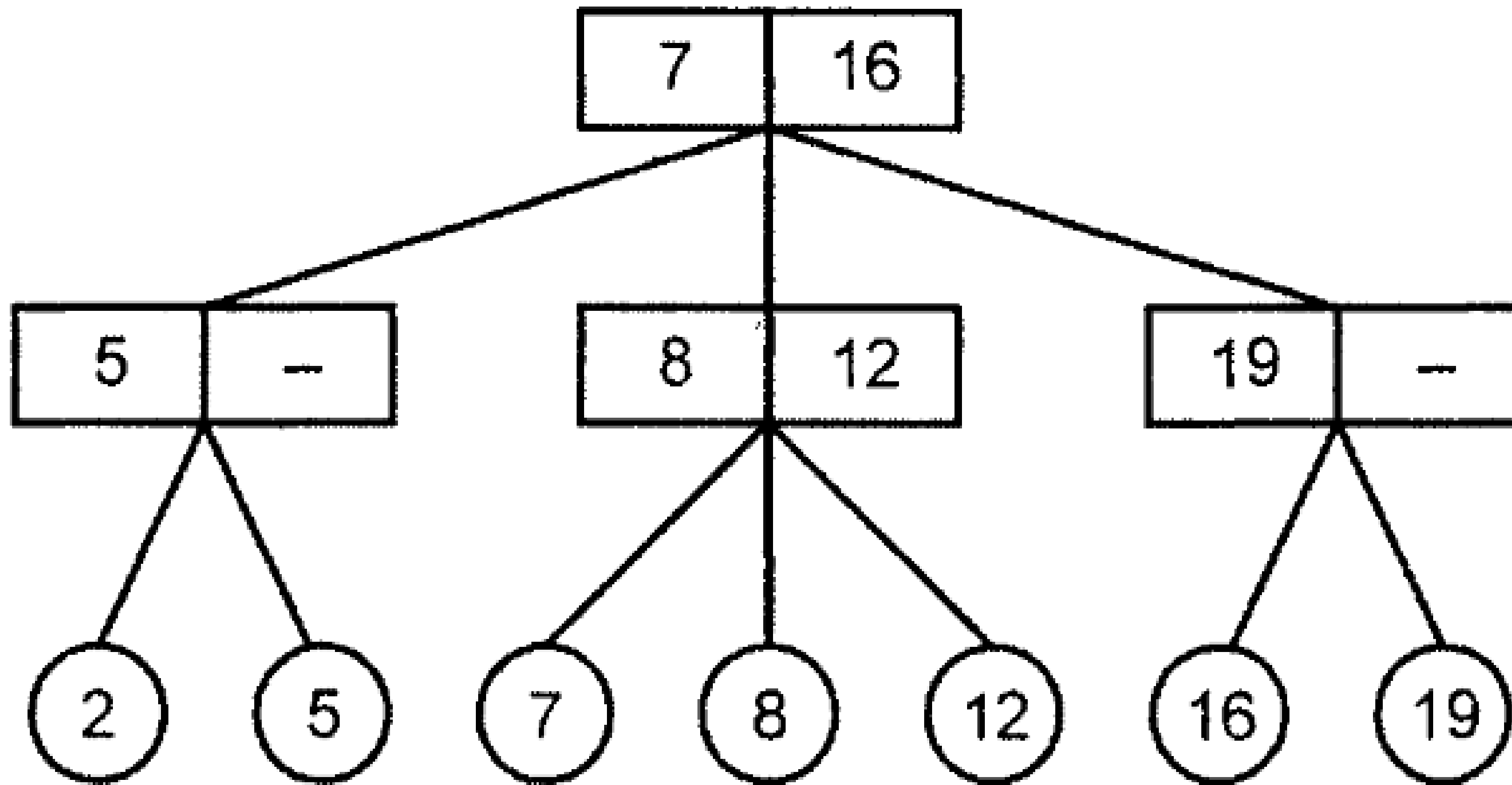
a) Each non-leaf vertex has 2 or 3 child vertices.
b) The depths (=lengths of the paths from the root) of all leaves are equal.

The structure of a 2-3 tree:

a) The items are stored in leaves.
b) Each non-leaf vertex stores two labels
    1. The least key stored in the 2nd subtree.
    2. The least key stored in the 3rd subtree (or NULL, if there is no 3rd subtree).

The height of a 2-3 tree is $O(\log n)$.

2-3 trees



2-3 trees

A helper function `Find(K)`:

1. Start from the root (current vertex = root of the 2-3 tree).
2. If K is less than the 1st label of the current vertex then move to the left child (current vertex = 1st child).
3. Else if K is less than the 2nd label of the current vertex then move to the middle child (current vertex = 2nd child).
4. Else move to the right child (current vertex = 3rd child).
5. Repeat steps 2-4 until a leaf is reached.
6. Return ‘true’ and the position of the found vertex or ‘false’ and the position where the vertex would be located.

Time complexity: $O(h) = O(\log n)$.

2-3 trees

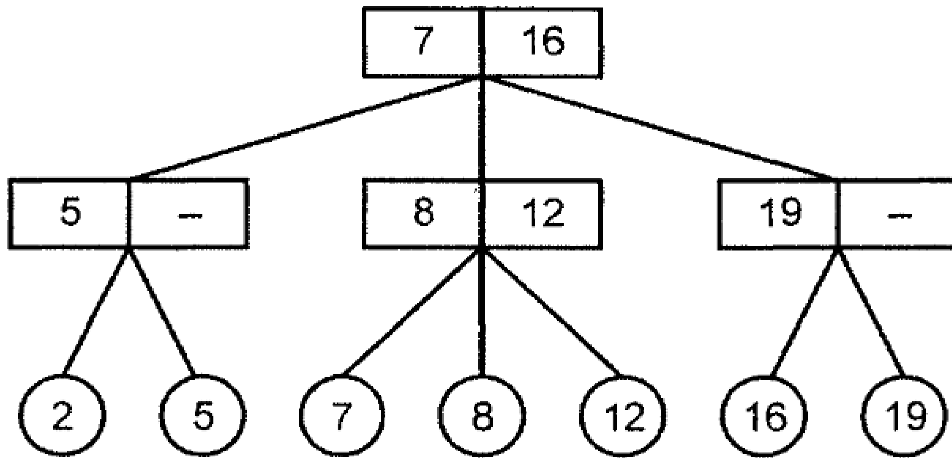
`GetMin()` : start from the root and move to the leftmost vertex, i.e. stop when the current vertex has no left child. Time complexity: $O(h) = O(\log n)$.

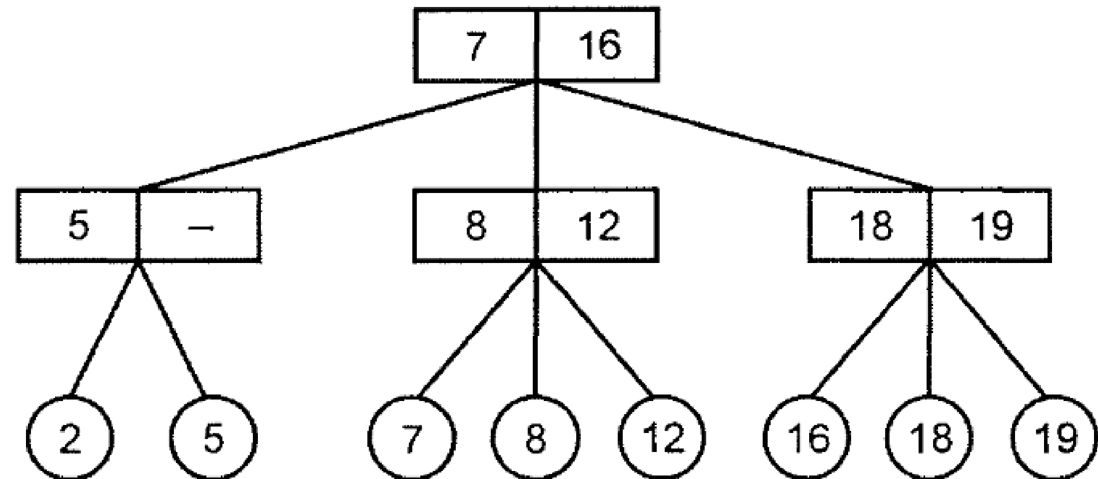
`Add(x, key)` : search for the position at which *x* would be located in the 2-3 tree. We stop at the non-leaf vertex w.

- If w has only 2 children, we just add the new leaf v as the proper child of w, then recalculate labels of w.

2-3 trees

Adding an item with key 18:





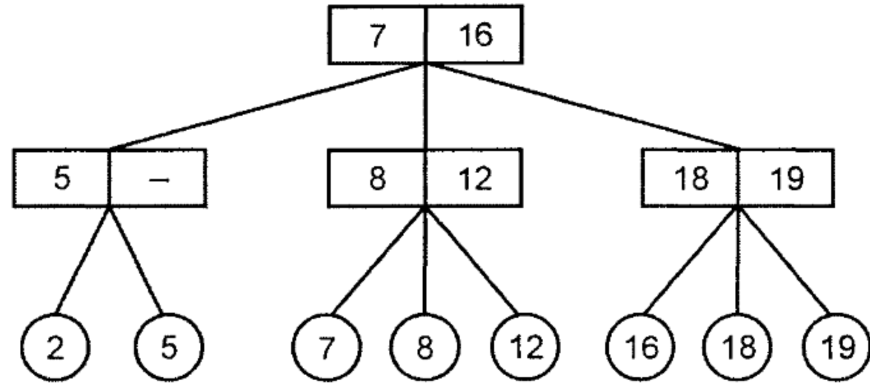
2-3 trees

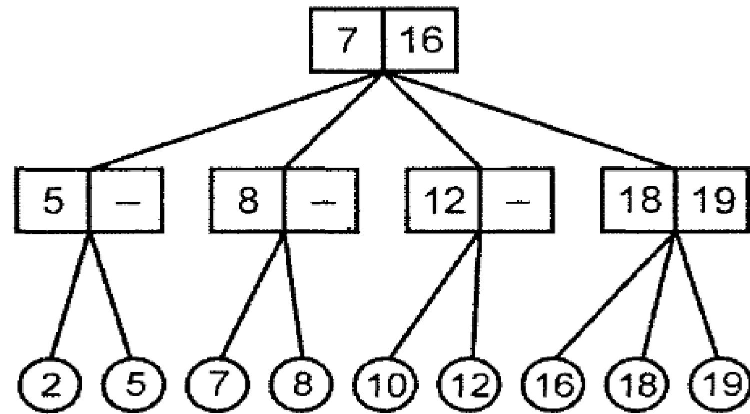
`Add(x,key)`: search for the position at which *x* would be located in the 2-3. We stop at the non-leaf vertex w.

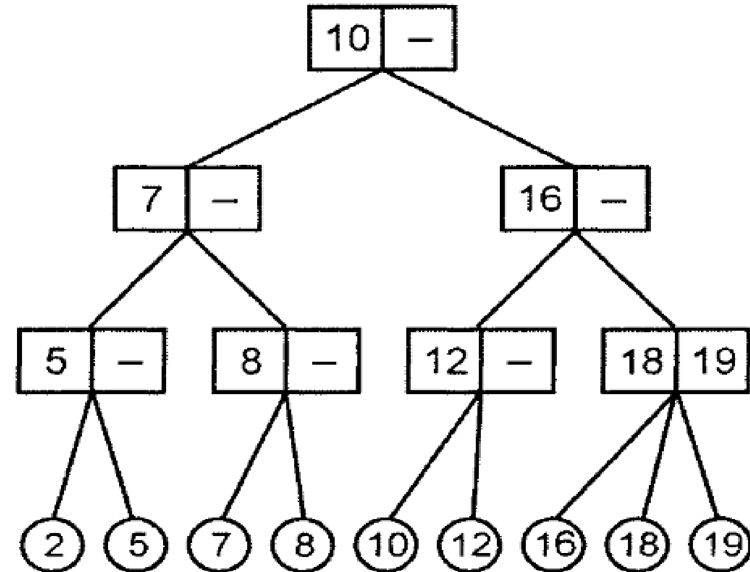
- …
- If w has 3 children, we add the new leaf v as the 4th and then reconstruct the subtree at w :
  1) Split w into two vertices w' and $w"$.
  2) Vertex w' receives the leaves with the least two keys, vertex $w"$ receives the leaves with the greatest two keys. Assign appropriate labels to w' and $w"$.
  3) If the parent of w' and $w"$ has 4 children, repeat the reconstruction.
  4) If the root has been split, create a new root of the 2-3 tree.

2-3 trees

Adding an item with key 10:







2-3 trees

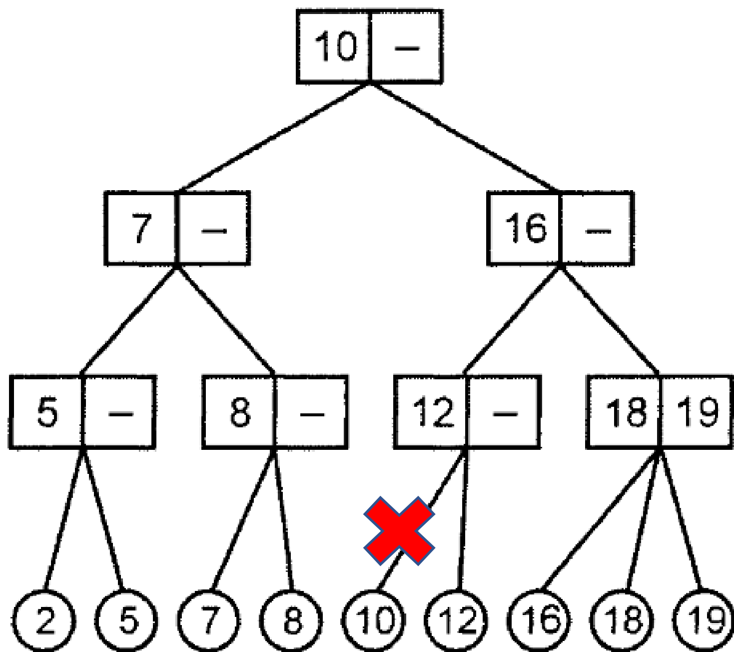
`DelMin()`: delete the leftmost leaf of the tree.

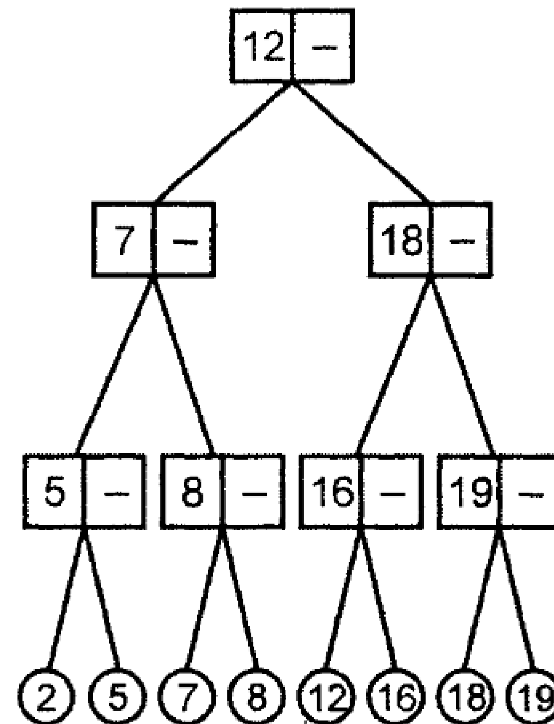
Deleting a leaf v from the 2-3 tree:

- Let w be the parent of v.
- Remove v.
- If w has 2 children then stop the operation.
- If w has 1 child then
  1) If w has a sibling u with 3 children then move one of $u's$ children to be a child of w. Update the labels of w , u and their parent.

2-3 trees

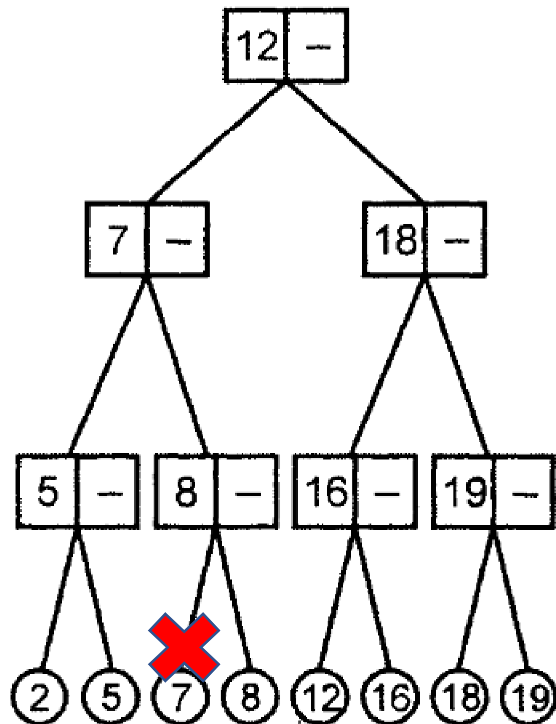
- If w has 1 child then
  1) If w has a sibling u with 3 children then move one of $u's$ children to be a child of w. Update the labels of w , u and their parent.

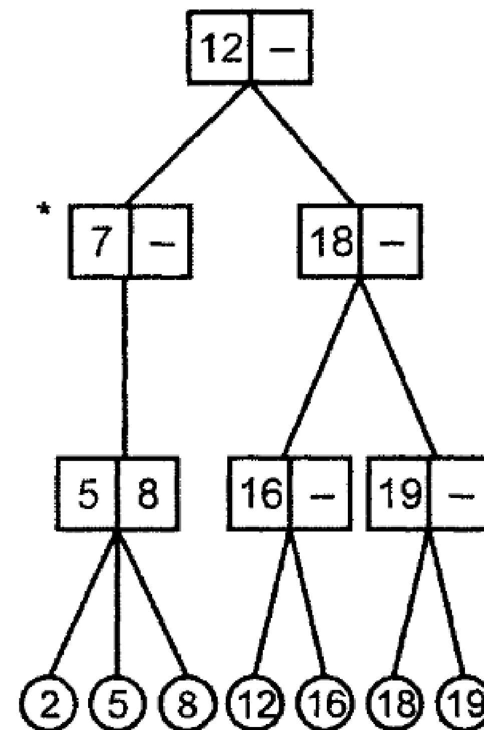




2-3 trees

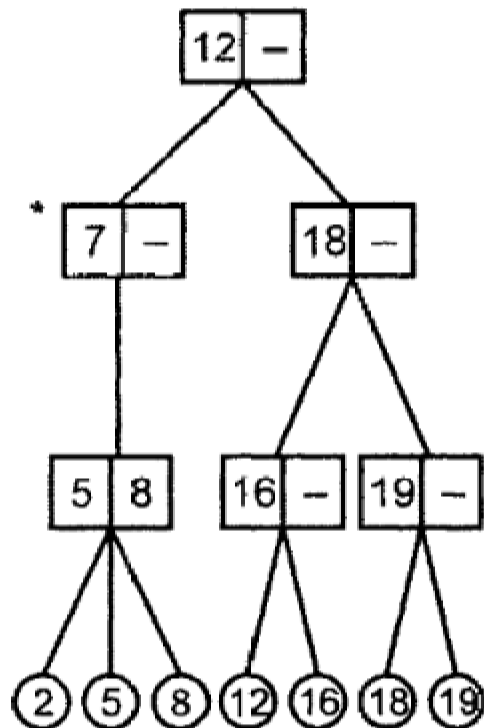
- If w has 1 child then
  2) If w has only a sibling u with 2 children then move the child w to be a child of u. Delete w . Update the labels of u and its parent.

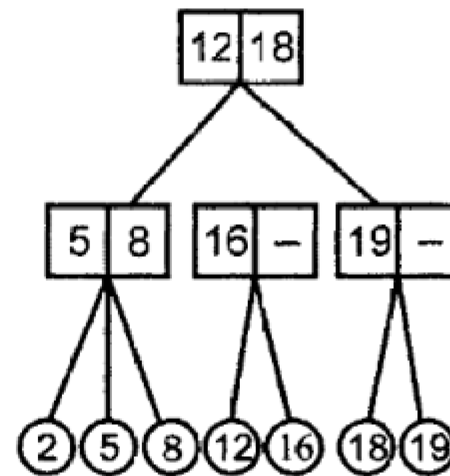




2-3 trees

- Repeat this procedure recursively.
- If we reach the root and the root has only 1 child => remove the root and make its child the new root.





2-3 trees

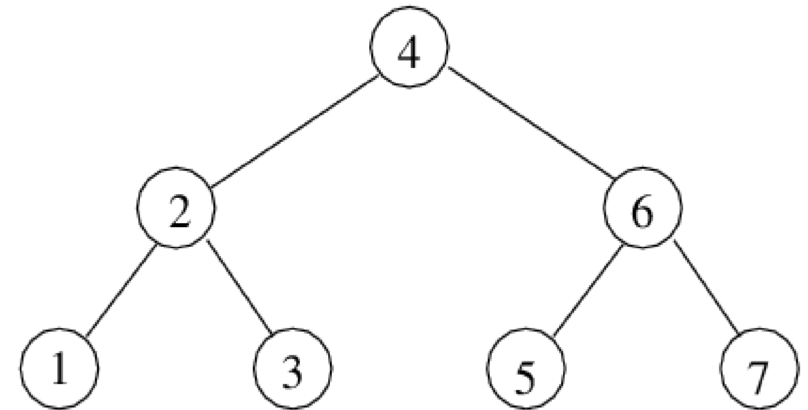
Summary of time complexity for 2-3 trees: `GetMin`, `DelMin`, `Add` have time complexity $O(h)$, where h is the height of the tree.

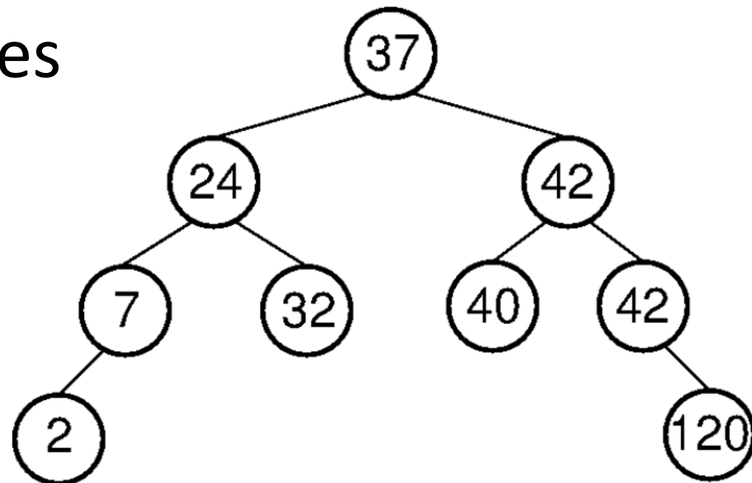
Height of a 2-3 tree is $O(\log n)$ - both on the average and in the worst case.

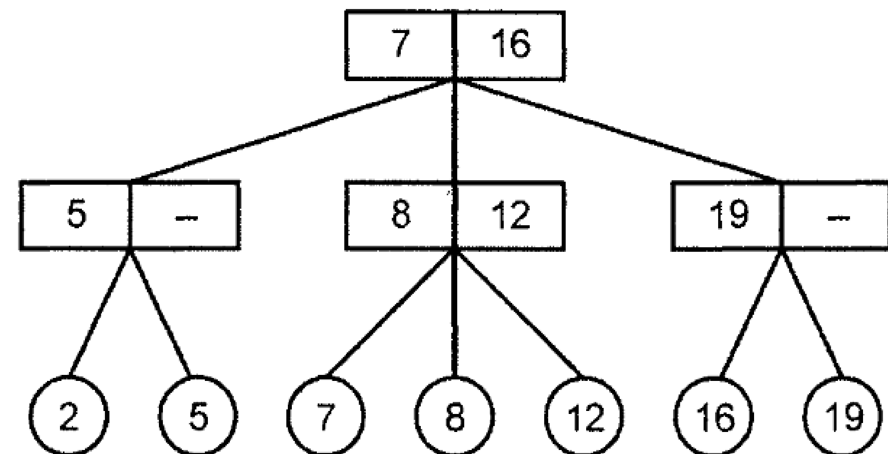
Tree-based data structures

These tree-based data structures keep items in *sorted* order:

- Binary search tree (unbalanced)
- AVL trees
- Red-black trees
- 2-3 trees
- …





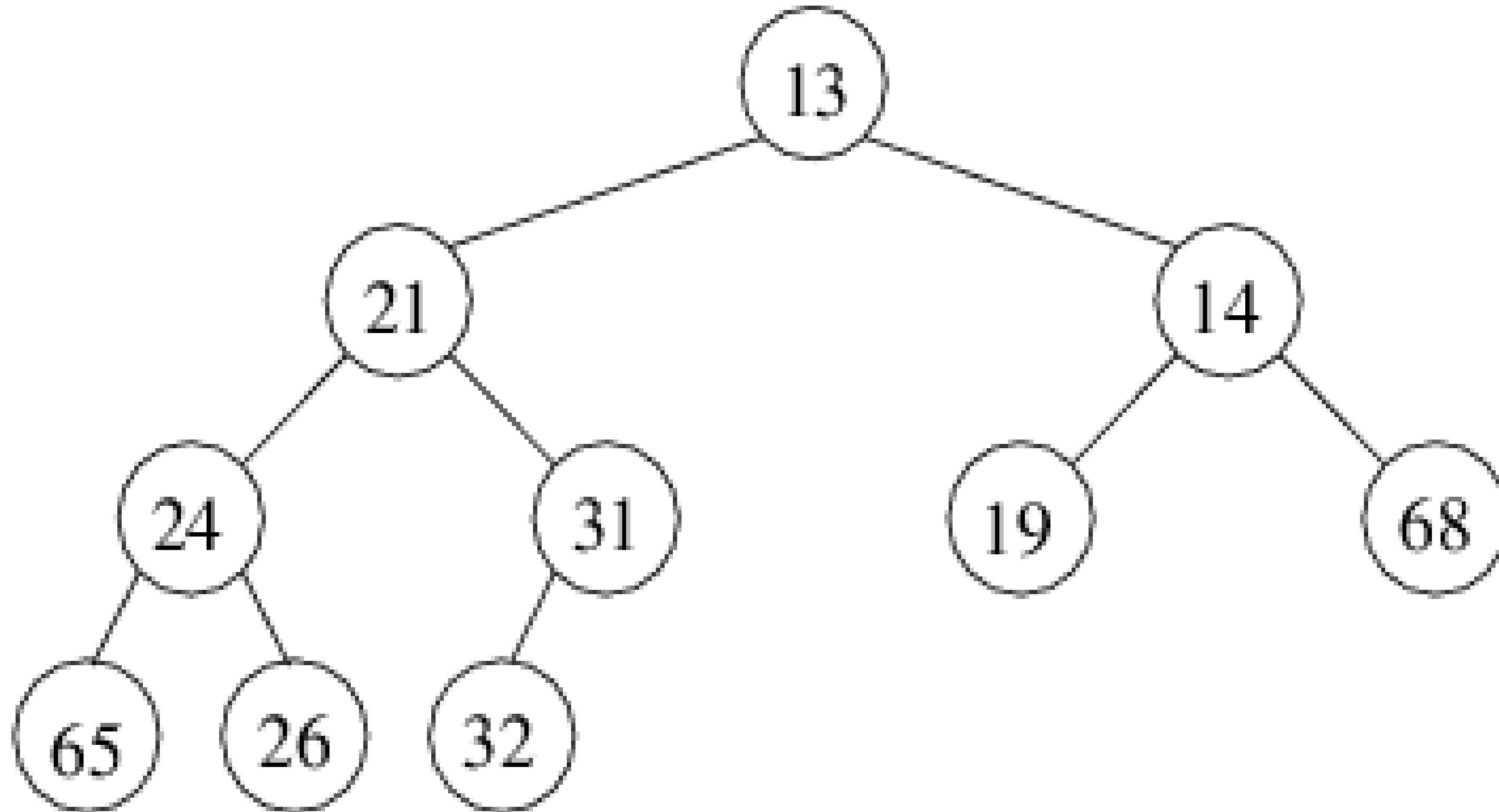


Heaps

A *heap* is a data structure which efficiently implements a priority queue with $O(1)$ time complexity for `GetMin()` and $O(\log n)$ time complexity for `DelMin()`.

Heaps are implemented as tree-based data structures for which all vertices store item+key pairs and the following *heap condition* holds: *the key of any non-root vertex is not less (not greater, for maximizing heaps) than the key of its parent*. Hence the minimum key item is always stored in the root.

Heaps



Binary heap

A *binary heap* is implemented as a *complete* binary tree represented as a linear array.

A binary tree is called *complete* iff every *level* of this tree, except possibly the last, is completely filled. If the last level is incomplete, the vertices at the last level are situated as far left as it is possible.

Binary heap