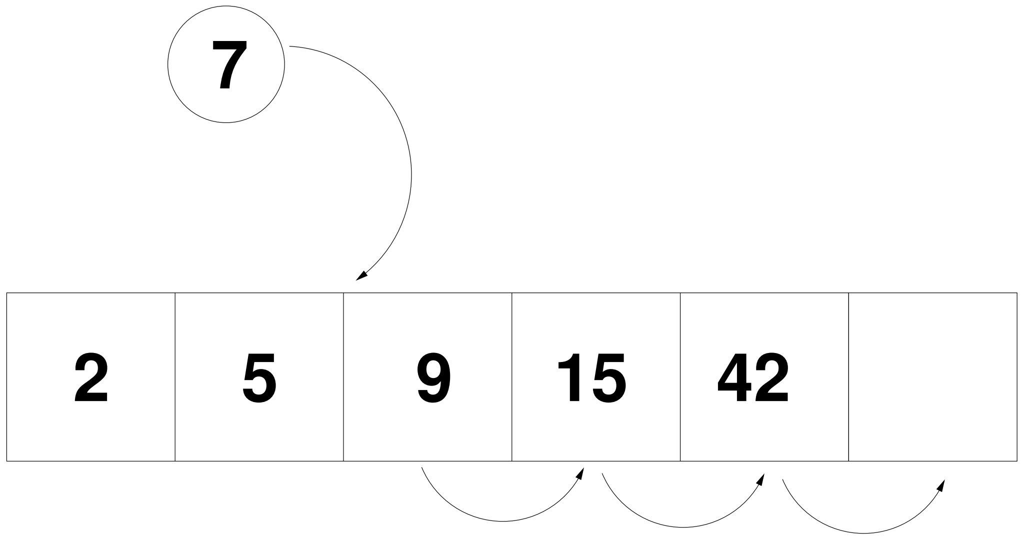
7 Interlude: Is keeping the data sorted worth it?

When a sorted range is needed, one idea that comes to mind is to keep the data stored in the sorted order as more data comes into the structure

Is this an efficient approach to problems needing a sorted range?



- scanning through a sorted data range can be stopped once an element larger than the one under search has been met

  $\Rightarrow$ it's enough to scan through approximately half of the elements

  $\Rightarrow$ scanning becomes more efficient by a constant coefficient

- in the addition, the correct location needs to be found, i.e. the data needs to be scanned through

  $\Rightarrow$ addition of a new element into the middle of the data range is $\Theta(n)$

  $\Rightarrow$ addition becomes $\Theta(n^2)$

$\Rightarrow$ it's usually not worth the effort to keep the data sorted unless it's somehow beneficial to the other purposes and a data structure with constant time insert can be chosen

- if the same key cannot be stored more than once the addition requires searching anyway

  $\Rightarrow$ mainting the order becomes beneficial in a data structure with a constant time insert

8 Tree, Heap and Priority queue

This chapter deals with a design method *transform and conquer*

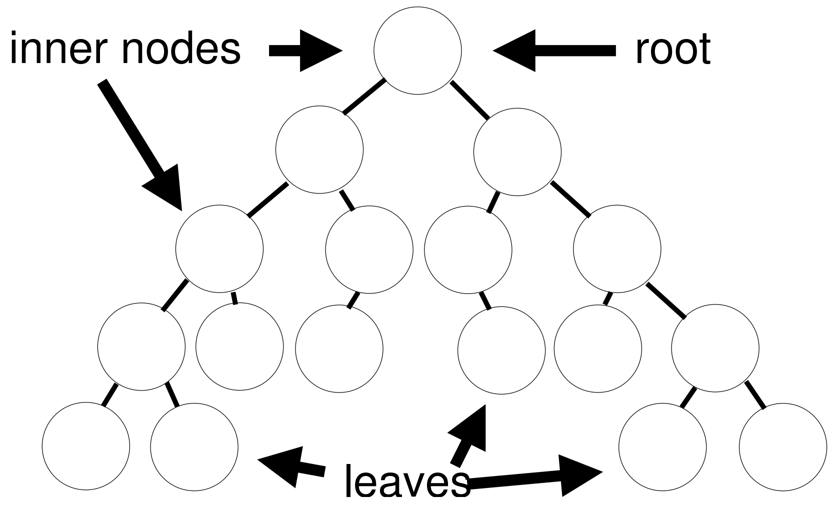
The notion of a *(binary) tree* and a *heap* are introduced

A sort based on the construction of a heap tree (HEAPSORT) is investigated

A *priority queue*, a set of elements with an orderable characteristic – a priority, is discussed.

8.1 Trees

- a structure that consists of nodes who each may have (an arbitrary number of) chidren
- For *binary* trees, the number of children is limited to 0, 1, or 2), and the children are called *left* and *right*



- a node is the *parent* of its children
- a childless node is called a *leaf*, and the other nodes are *internal nodes*

- a tree has at most one node that has no parent, i.e. the *root*
  - all other nodes are the root's children, grandchildren etc.
- the descendants of each node form the *subtree* of the tree with the node as the root
- The *height* of a node in a tree is the length of the longest simple downward path from the node to a leaf
  - the edges are counted into the height, the height of a leaf is 0
- the height of a tree is the height of it's root

- a tree is *completely balanced* if the difference between the height of the root's subtrees is atmost one and the subtrees are completely balanced
- the height of a tree with n nodes is at least $\lfloor \lg n \rfloor$ and atmost n - 1 (the base of the logarithm depends on how many children nodes may have) $\Rightarrow O(n)$ and $\Omega(\lg n)$

The nodes of the tree can be handled in different orders.

- *preorder*
  - call PREORDER-TREE-WALK($T.root$)
  - Handle a node first, only then recursively handle its children

```
PREORDER-TREE-WALK(x)
if x ≠ NIL then
    process the element x
    for child in x→children do
        PREORDER-TREE-WALK(child)
```

- *inorder*
  - Usually only applicable to *binary* trees.
  - First recursively handle the left subtree, the the node, then recursively its right subtree

INORDER-TREE-WALK(x)
if $x \neq$ NIL **then**
    INORDER-TREE-WALK($x \rightarrow left$)
    process the element x
    INORDER-TREE-WALK($x \rightarrow right$)

- *postorder*
  - First recursively handle node's children, only then the node itself

```
POSTORDER-TREE-WALK(x)
if x ≠ NIL then
    for child in x→children do
        POSTORDER-TREE-WALK(child)
    process the element x
```

- running-time $\Theta(n)$
- extra memory consumption = Θ(maximum recursion depth) = $\Theta(h+1)$ = $\Theta(h)$