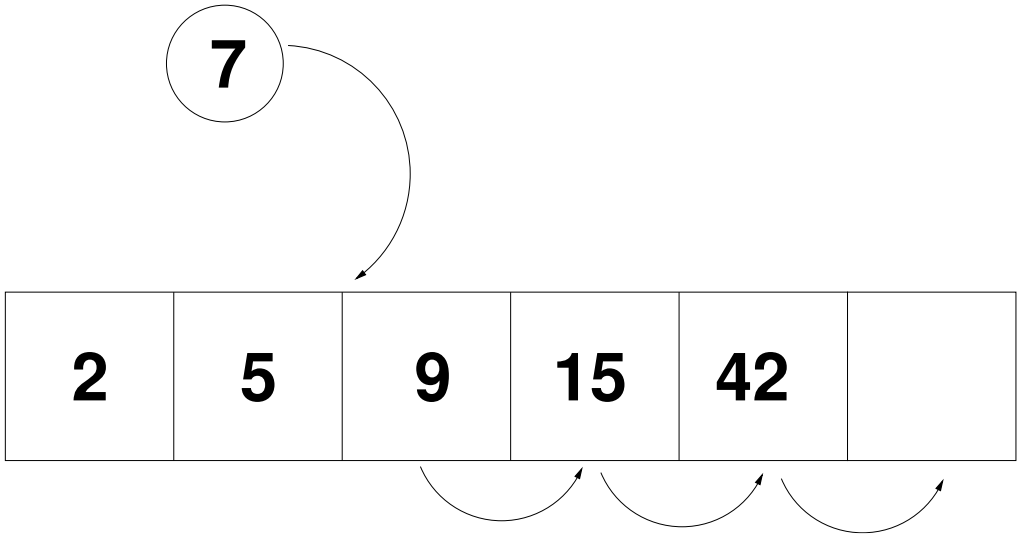
7 Interlude: Is keeping the data sorted worth it?

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 Algorithm Design Technique: Transform and Conquer

Transform and conquer is a design technique that

- First modifies the problem's instance to be more amenable to solution – the transformation stage
- Second the problem is solved – the conquering stage

There are three major variations in the way the instance is transformed

- *Instance simplification*: a simpler or more convenient instance of the same problem
- *Representation change*: a different representation of the same instance
- *Problem reduction*: an instance of a different problem for which an algorihtm is already available.

8.2 Sorting with a heap

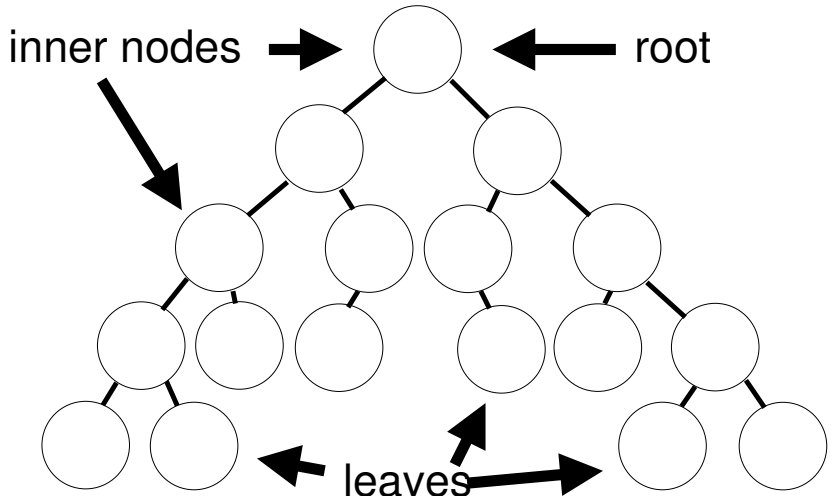
Let's discuss a sorting algorithm HEAPSORT that uses a very important data structure, a *heap*, to manage data during execution.

Binary trees

Before we get our hands on the heap, let's define what a *binary tree* is

- a structure that consists of nodes who each have 0, 1 or 2 chidren
- 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 node*s
- a binary 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 binary tree with the node as the root
- The *height* of a node in a binary 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 binary tree is the height of it's root



- a binary tree is *completely balanced* if the difference between the height of the root's left and right subtrees is atmost one and the subtrees are completely balanced
- the height of a binary tree with n nodes is at least $\lfloor \lg n \rfloor$ and atmost n - 1

  $\Rightarrow O(n)$ and $\Omega(\lg n)$

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

- *preorder*
  - call PREORDER-TREE-WALK($T.root$)
  - the nodes of the example are handled in the following order: 18, 13, 8, 5, 3, 6, 9, 15, 14, 25, 22, 23, 30, 26, 33, 32, 35

```
PREORDER-TREE-WALK(x)
1  if x ≠ NIL then
2      process the element x
3      PREORDER-TREE-WALK(x→left)
4      PREORDER-TREE-WALK(x→right)
```

- *inorder*
  - in the example 3, 5, 6, 8, 9, 13, 14, 15, 18, 22, 23, 25, 26, 30, 32, 33, 35
  - the binary search tree is handled in the ascending order of the keys when processing the elements with inorder

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL then
2      INORDER-TREE-WALK(x→left)
3      process the element x
4      INORDER-TREE-WALK(x→right)
```

- *postorder*
  - in the example 3, 6, 5, 9, 8, 14, 15, 13, 23, 22, 26, 32, 35, 33, 30, 25, 18

```
POSTORDER-TREE-WALK(x)
1  if x ≠ NIL then
2       POSTORDER-TREE-WALK(x→left)
3       POSTORDER-TREE-WALK(x→right)
4       process the element x
```

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