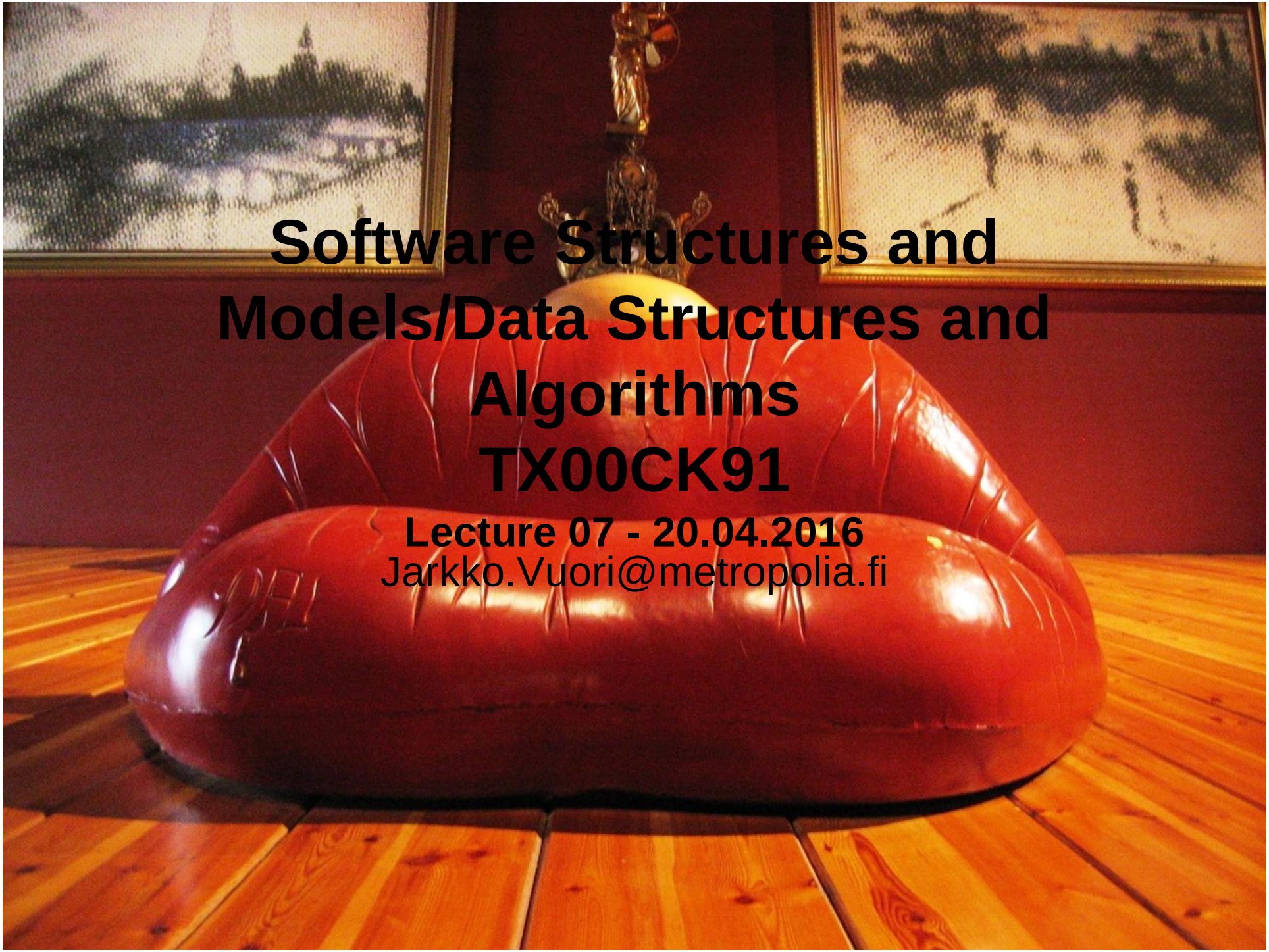


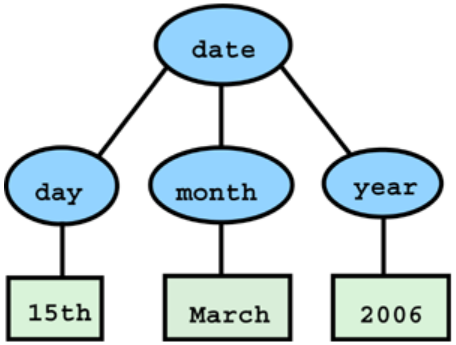
Software Structures and Models/Data Structures and Algorithms TX00CK91

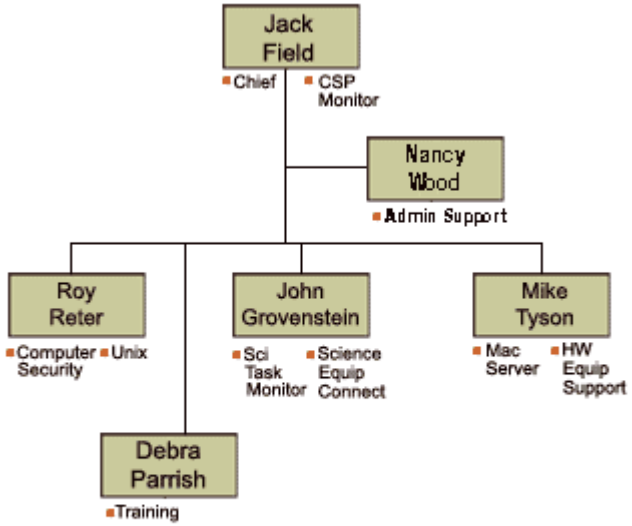
Lecture 07 - 20.04.2016

Jarkko.Vuori@metropolia.fi

Tree (structure)

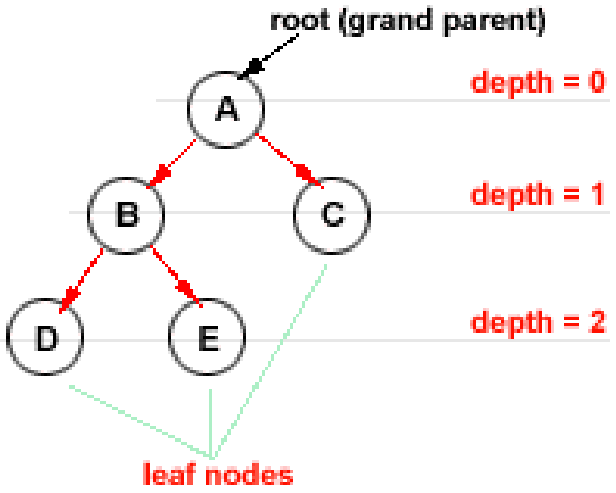
- Tree (structure) is one of the most fundamental data structures in computer science
- It offers a method to search data very efficiently
  - It is used in directories and compilers
- It is more complicated data structure than many other ADTs
- There are many possible ways to implement them
  - Can be based either on arrays or linked structures
- Node of the tree can contain application data
  - As with other containers, implementation of the tree structure can be made independent of item type
- XML (eXtensible Markup Langugage) uses tree structure for storing/representing data
- All the operations are targeted to the nodes of the tree
- Suits well to represent hierarchial data efficiently
  - Book contains chapters, which then contains titles, which contains paragraphs, etc.
  - Organization may have chairman, chief executive officer, department leader, engineer, worker, etc.





Dynamic tree

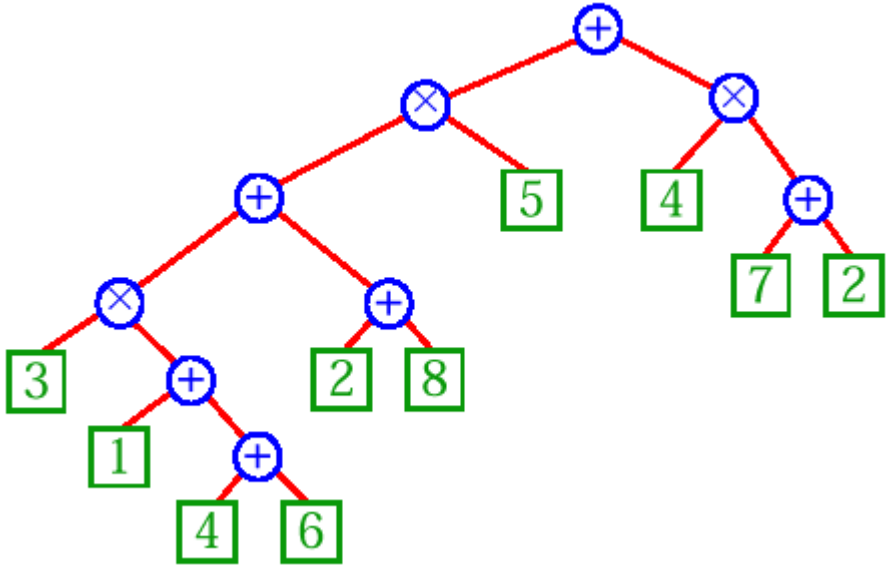
- Queues and lists can be defined recursively
  - E.g. List is
    1. Empty
    2. List and a new item a added to it (aL)
- Tree can also be defined recursively
  1. Empty
  2. Node with type T which has non contiguous subtrees with type T
- In principle a list is a tree whose node is degenerated
  - It has only one subtree
- Tree may have more than two subtrees also
  - If there is only two subtrees, we call this as *binary tree*
    - Binary tree is the simplest tree structure
- First node is called as a root, and those nodes who does not have subtrees are called as leafs
  - When this structure is switched to upside down (root is up and leafs are down), we get the diagram which describes the tree data structure
- Depth of the Node describes how far (how many steps are needed) the node is from the root node




```
private class Node {
    private T data;
    private Node left;
    private Node right;
        …
}
```

Binary tree

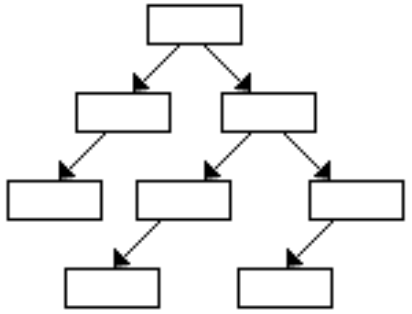
- Arithmetic expression can be represented with a binary tree
  - Transformation to this representation is called as parsing
  - Compilers will do this when they compile the source code
    - To find out what the programmer has had in his/her mind when writing the program



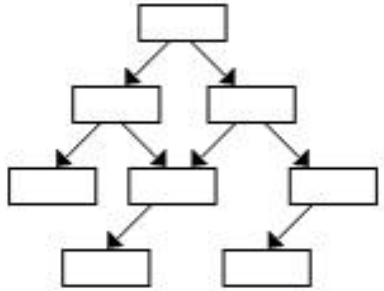
(((4+6)+1)*3+(2+8))*5+(7+2)*4

Binary tree

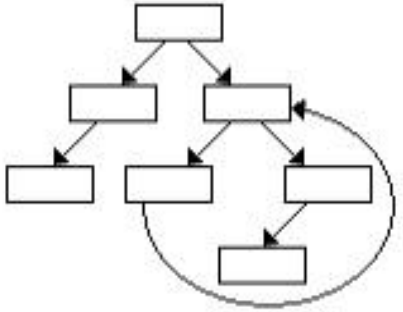
- Every possible binary tree structure is not legal
  - Subtree pointers can point to nodes that belongs to upper nodes
- The requirement that every node has two links is not enough
  - In principle doubly linked list could also be a tree structure if only those links are arranged in a suitable way



A valid binary tree



A valid binary tree with a shared subtree

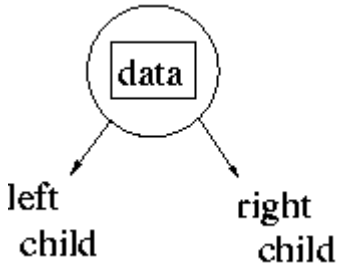


An invalid binary tree

ADT binary tree

- Binary tree consist of nodes
- Every node has an item and left and right children (subtree)
  - Right and left childrens are pointers to subtrees
    - They can point to a node (which is a root of a new subtree) or can have a value of NULL when the subree is empty
- Operations
  - initialization
  - Inserting an element
  - Searching from the tree
  - Deleting an element
  - Traversing all the tree's elements (from the given root)

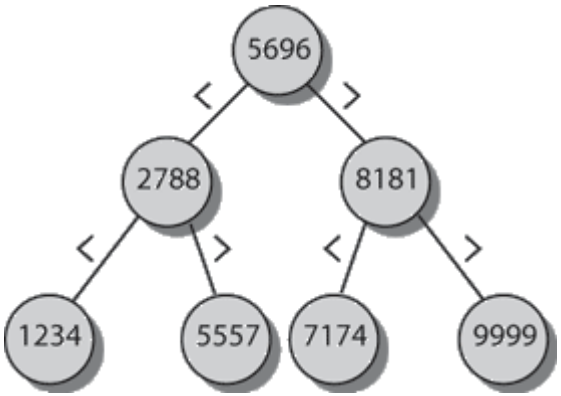
```
private class Node {
    private T data;
    private Node left;
    private Node right;
       …
}
```




```
void putItemToTree(T item);
bool isItemInTree(T item);
void print();
          .
          .
          .
```

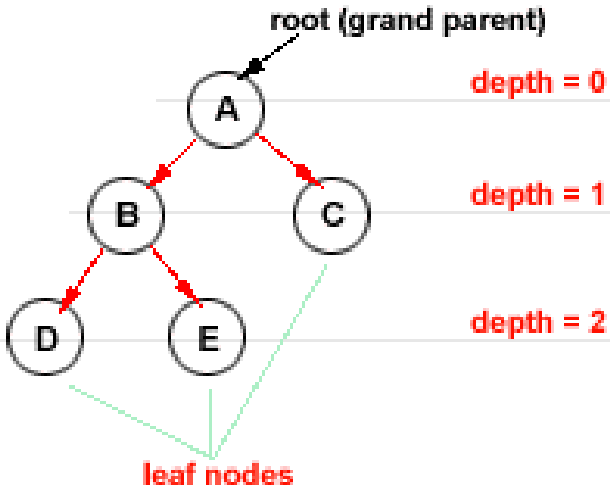
Binary search tree

- It is quite difficult to search data from the linked list
  - We can easily access the preceding and the following node in double linked lists (from the given node)
  - But on the average, we must still check half of the nodes before the needed item has been found
- If the data is stored in such a way that on the left subtree we have smaller items and on the right subtree we have larger (than the current node), we can use binary search algorithm for finding the data
  - This tree structure is called as a *binary **search** tree*
  - Ordering between smaller and larger can be defined by the application programmer (e.g. numerical order, aplhabetical order, etc.)
  - Every subtree selection reduces the number of remaining nodes by half
    - That is the reason for the name bi-nary tree



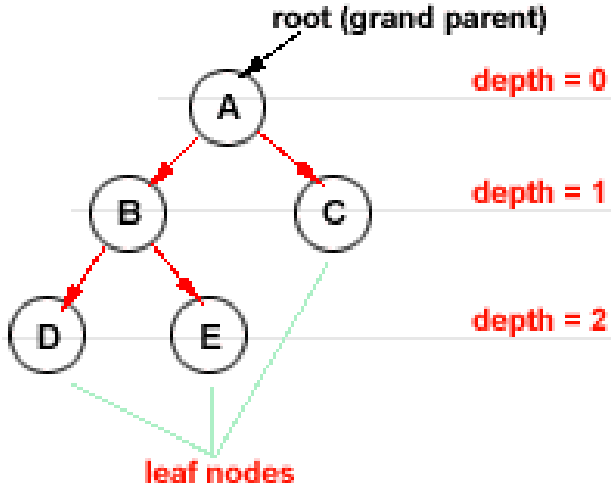
Binary tree, traversing the tree

- Different traversing orders:
  - preorder
    - First we traverse the root, then left subtree and finally the right subtree
    - A B D E C
  - inorder
    - First we traverse the left subtree, then the root and finally the right subtree
    - D B E A C
  - postorder
    - First we traverse the left subtree, then the right subtree and finally the root
    - D E B C A
- All those traversing orders are represented in recursive form



Binary tree, inorder traversing

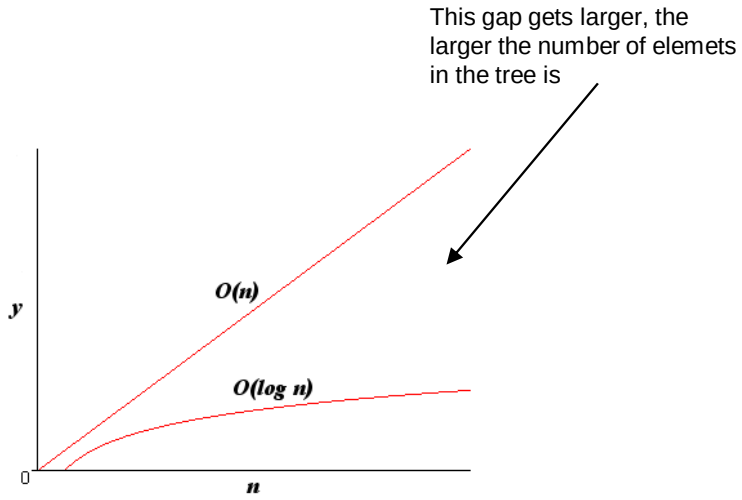
- function in_order_traverse traverses nodes of the binary search tree in order of increasing values
- In this function, we use a pointer stack where we store the addresses of those nodes to where we must return later on




```
void in_order_traverse(Node *tree) {
    stack<Node> pointerstack = new stack<Node>;
    Tree        p;

    p = tree; // tree is a Tree class variable
    while ((p != NULL) || !pointerstack.empty()) {
        while (p != NULL) {
            pointerstack.push(p);
            p = p.left;
        }
        p = pointerstack.top(); pointerstack.pop();
        System.out.print(p.item + " ");
        p = p.right;
    }
}
```

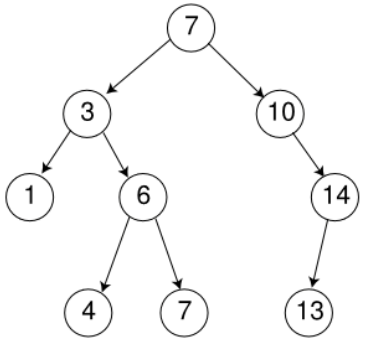
Binary tree, searching

- The searching algorithm is simple
  - If the node to be searched is in the root, searching is terminated
  - If the searched item is ”smaller” than the root, the item will be in the left subtree
  - If the searched item is ”larger” than the root, the item will be in the right subtree
- Because the tree can be defined reursively, also the search algorithm can be defined recursively (recursion will be discussed in the next lecture)
- Every comparision makes the ”nodes to be searched” set smaller, when we select only one of the subtrees
  - Binary search, on the **average** needs $\log_2 n$ operations
- Searching information from the binary search tree (BST) is a very efficient operation
- How the search operation for the item 5557 goes on the given search tree (on the right side)?



Binary search tree, adding an element

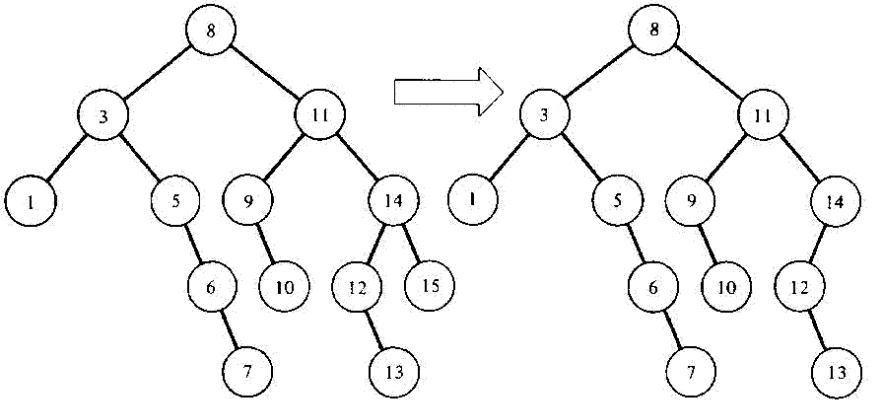
- When inserting a node, we must spesify also the ordering function
  - So that the new item will be inserted to the right place
- We search such a place in the tree, where the item whould have been
  - When this place is founded, we set the item to that place (node)
- Node insertion is an efficient operation; on the average, $\log_2 n$ node travelsals are needed
  - Of course it takes more time that inserting an item to the linked list
- To where (in the given tree) a node with a value of 5 would be inserted?



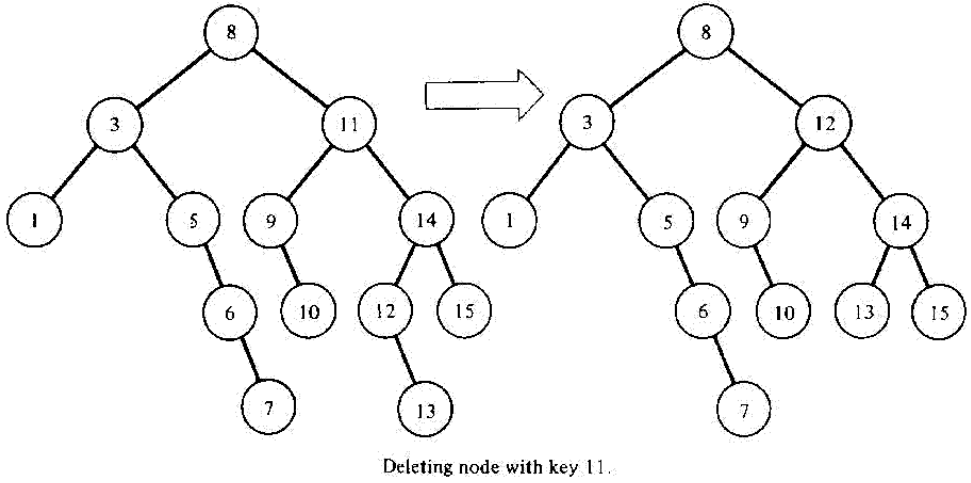

```
Node *insert_node(Titem key) {
    Node *node;

    node = new Node;
    node->item = key;

    return (node);
}

Node *insert_binary_tree(Node *node,
                         int (*compare)(T item1, T item2),
                         T key) {
    if (compare(node->item, key) <= 0) {
        if (node->left == NULL)
            return (node->left = insert_node(key));
        else
            insert_binary_tree(node->left, compare, key);
    } else if (compare(node->item, key) > 0) {
        if (node->right == NULL)
            return (node->right = insert_node(key));
        else
            insert_binary_tree(node->right, compare, key);
    }
}
```

Binary search tree, deleting an element

- Deleting a node is a more difficult operation
  - Leaf removal is simple
  - Node who has only one children is a relatively easy case, move the children to the place where we just removed the node
  - Node who has two childrens is a difficult case; where to place those orphans (the other can be placed to the place where the deleted node was, but where to put the other children)



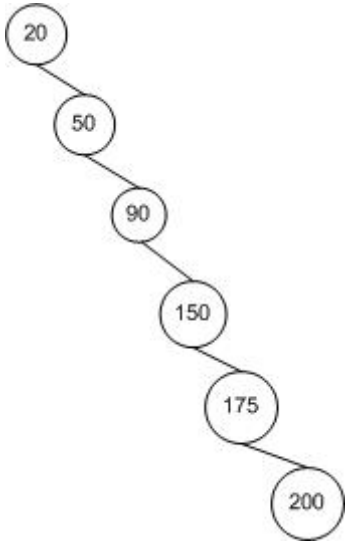
Deleting node with key 15.



Deleting node with key 11.

Binary search tree, balancing

- Binary search tree is called a balanced tree, when right and left subtrees have equal number of nodes, and all subtrees are balanced
- If node were inserted in unoptimal way, the tree could be unoptimally formed regarding search operation
  - In the picture right, we have inserted items with increasing values, therefore the structure of the tree is a linear list (only one node is a leaf)
  - Balanced three has half of the nodes as leafs
- Tree must be periodically balanced, if optimal searches are needed
- There exist algorithms that ensure that the tree is always balanced
  - AVL-tree: first self-balanced tree structure, introduced in 1962
  - Red-black tree : improved version, introduced in 1972



Binary tree where nodes
20, 50, 90, 150,
175, and 200 were inserted in that order

Dynamic and array structures

- Dynamic structure represents the order by special links (= addresses to memory blocks)
- Arrays represent order by the location in memory
  - Items are in successive locations (in memory), the whole array is a contiguous list in memory
- Array and dynamic linked list are implementation options for an ordered set of items

Property	Dynamic linked list	Array
Memory reservation	When needed. All free space of the computer can be used.	All the memory needed must be reserved before we can use it (we can ran out of the memory, or we can reserve too much of it).
Searching	Only sequential search is possible	Random access is also possible when we know the index if the item
Order expression	Order is expressed by links	Order is expressed by the position in memory
Processing power needed	Insertions and deletions between existing items are easy	Insertions and deletions requires copying items in memory
Programming	Needs exercise	Easy

Conclusion of abstract datatypes

- Standard abstract datatypes are general purpose, application independent components
  - lists
  - queue
  - stack
  - set
  - string
  - tree
  - etc.
- In a program development, we can create application dependent abstract datatypes
  - We can utilize, e.g. general abstract datatypes
  - measurement (exercises)
  - triangle
  - Complex number
  - matrix

  - Dynamic memory
  - etc.
- It is useful to use abstract datatypes because
  - Programs are simpler, and writing them is easier
  - Program work can be subdivided to more than one programmer
  - We don’t so easily get errors in our programs
  - Testing of the programs is easier
  - It is easier to produce reusable program code