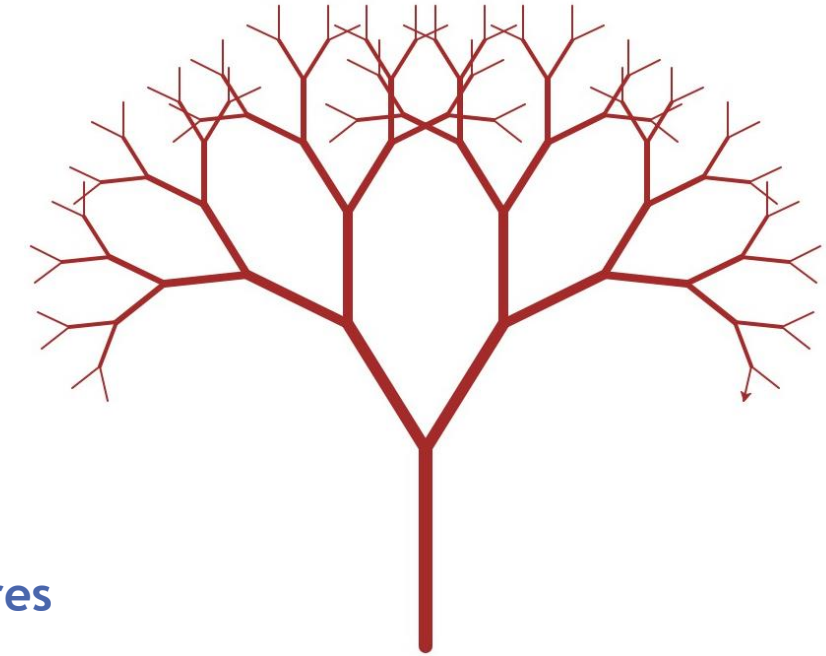
INFDEV026A - Algoritmiek
Unit 4

G. Costantini, F. Di Giacomo, G. Maggiore

costg@hr.nl, giacf@hr.nl, maggg@hr.nl – Office H4.204

Today

- ~~Why is my code slow?~~
  - ~~**Empirical and complexity analysis**~~
- ~~How do I order my data?~~
  - ~~**Sorting algorithms**~~
- How do I structure my data?
  - Linear, tabular, **recursive data structures**
- How do I represent relationship networks?
  - **Graphs**



BINARY TREES

Detailed agenda

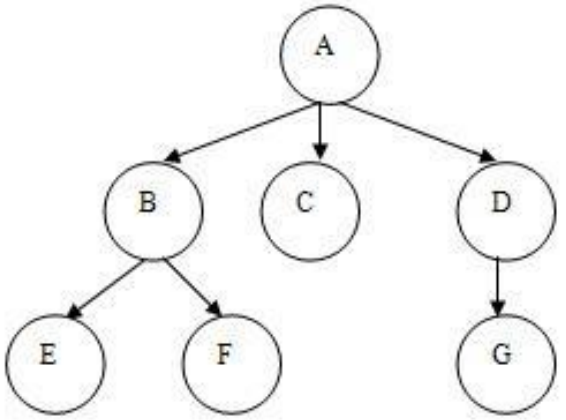
- What is a tree?
- What is a binary tree?
- What is a **binary search tree**?

- What is a 2-3 tree? [only short hint to basic idea]
- What is a **k-d tree**? [to do in the assignment, exercise 2]

Tree definition

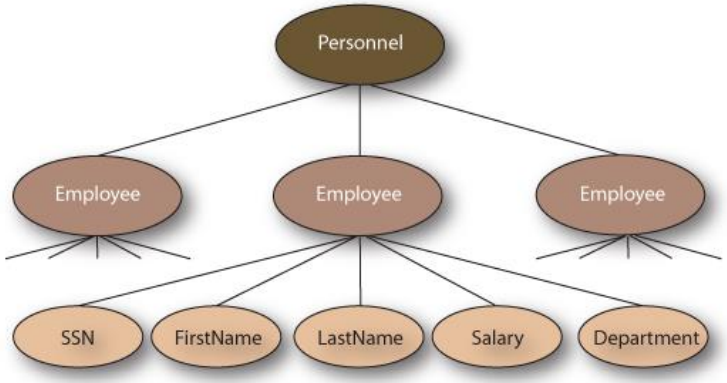
- Tree ⇒ nonlinear data structure made of *nodes* that models a hierarchical organization
- Very common structure in computer science
  - file systems, inheritance structure of Java classes, classification of Java types, etc…
- Tree with no nodes ⇒ *null* or *empty* tree
- Tree that is not empty
  - *root* node and potentially many levels of additional nodes that form a hierarchy

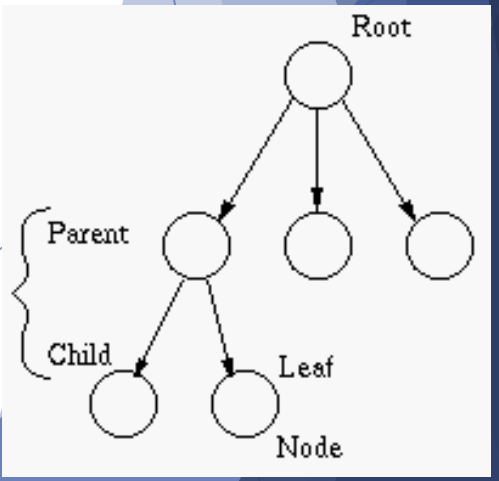
Tree definition

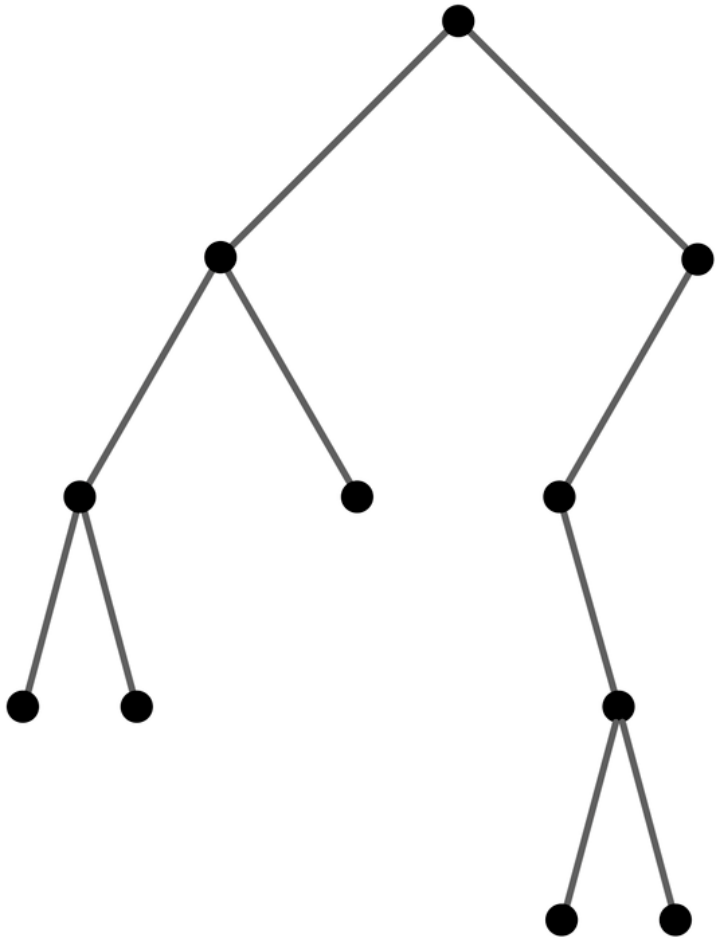


- Recursive definition
  - A tree is a pair (r, S), where r is a node and S is a set of disjoint trees, none of which contains r
    - r is the root of the tree
    - Elements of S are the subtrees; S can be empty

- Each element may have several successors (called its "*children*") and every element except one (called the "*root*") has a unique predecessor (called its "*parent*")





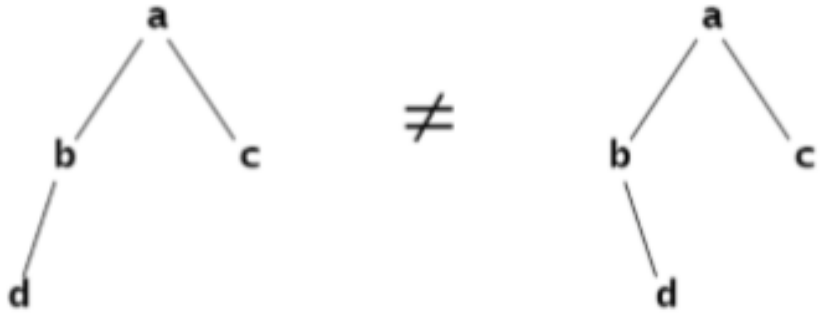


Binary Trees

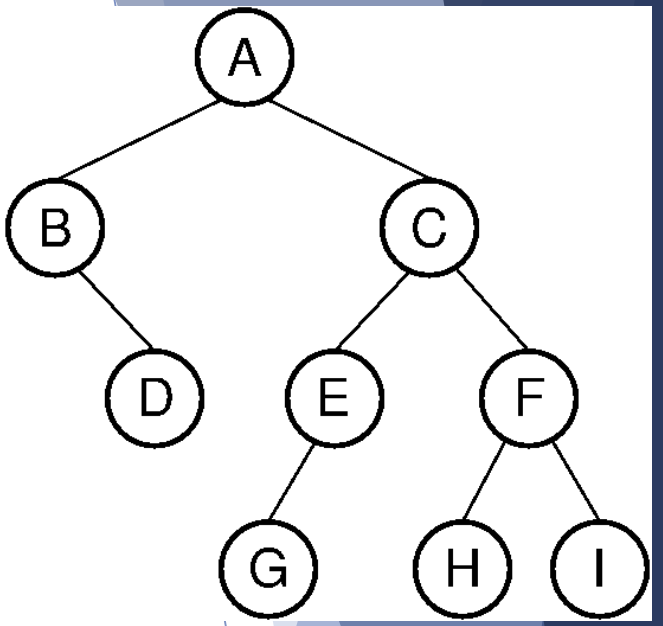
Binary tree definition

- Basic definition
  - Binary tree ⇒ tree data structure in which each node has at most two children (*left* child and *right* child)

- Recursive definition (using just set theory notions)
  - Binary tree ⇒ either the empty set or a **triple T** $= (x, L, R)$, where x is a node and L and R are disjoint binary trees (not containing x)
  - x is the root of the tree T
  - L is the left subtree of T
  - R is the right subtree of T

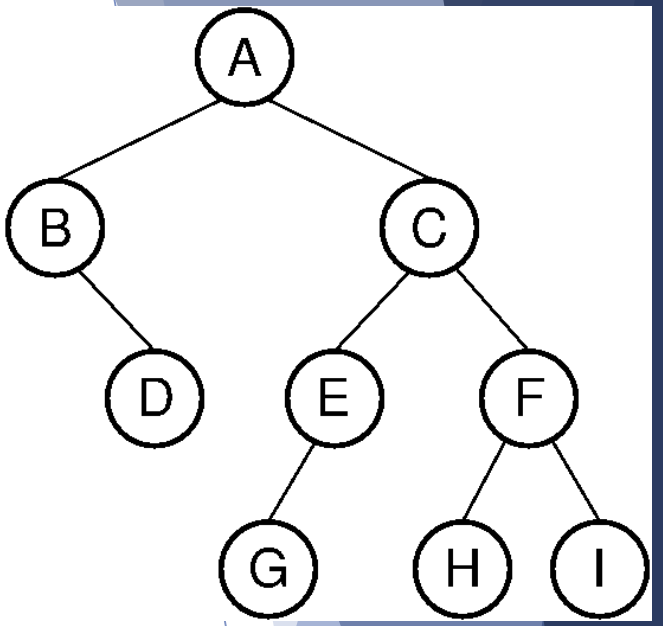


Binary trees properties (1/3)



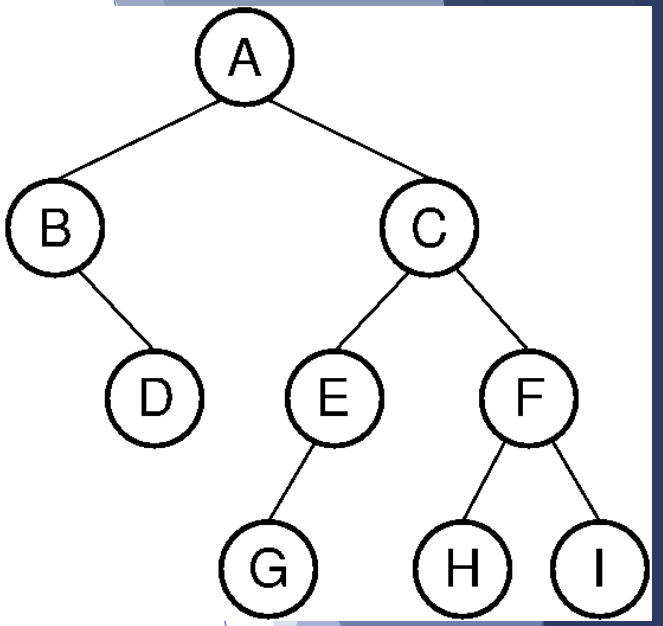
- **Size**
  - Number of nodes it contain
  - Singleton = tree of size 1
- **Parent** and **children**
  - Given the tree $T = (x, L, R)$, if x_L is the root of L and x_R is the root of R, then:
    - x is the parent of both x_L and x_R, which are its children
- **Leaf**
  - Node with no children
- **Internal node**
  - Node with at least one child

Binary trees properties (2/3)



- **Adjacent nodes**
  - One is the parent of the other
- **Path**
  - Sequence of nodes where each one is adjacent to the following one in the sequence
  - Length of a path ⇒ number of adjacent pairs (≈ arrows)
  - Trees are acyclic ⇒ No path can contain the same node more than once
- **Root path** for a node x_0
  - Path starting from x_0 and which ends in the root of the tree
- **Depth** of a node
  - Length of its root path

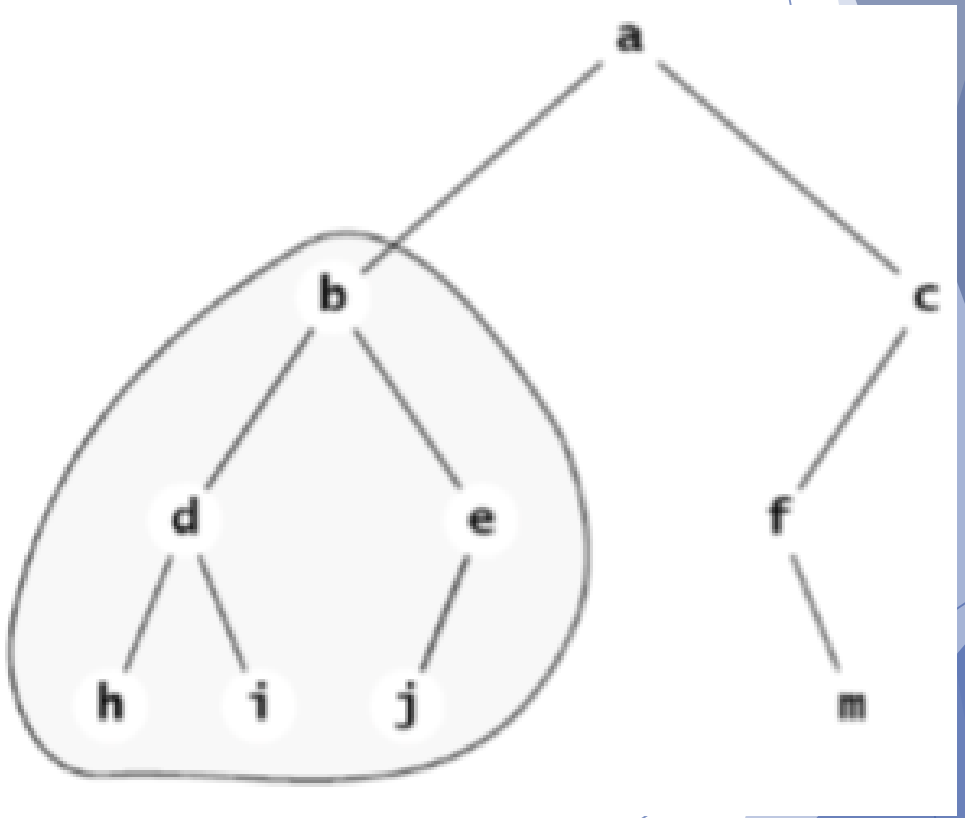
Binary trees properties (3/3)



- **Level**
  - Set of all nodes at a given depth
- **Height**
  - Greatest depth among its nodes
    - Height of a singleton is 0
    - Height of the empty tree is -1

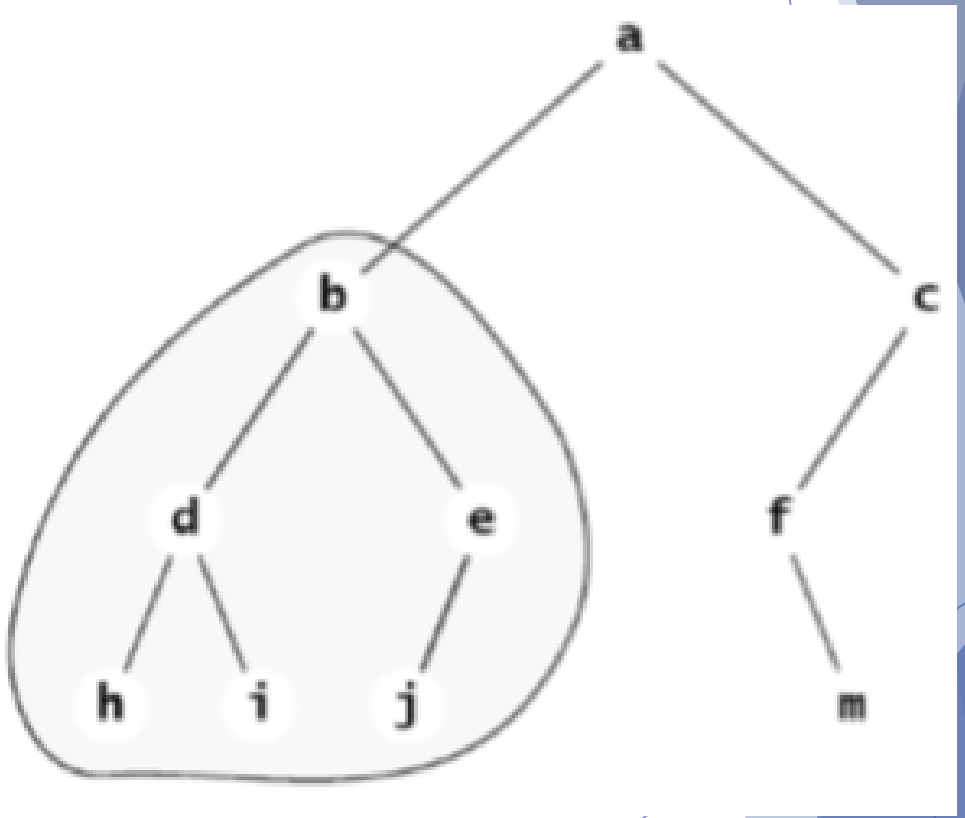
Binary trees properties – Example

- Size?
  - 10
- Height?
  - 3
- Root node?
  - a
- Path (a, b, e, i) is valid?
  - No

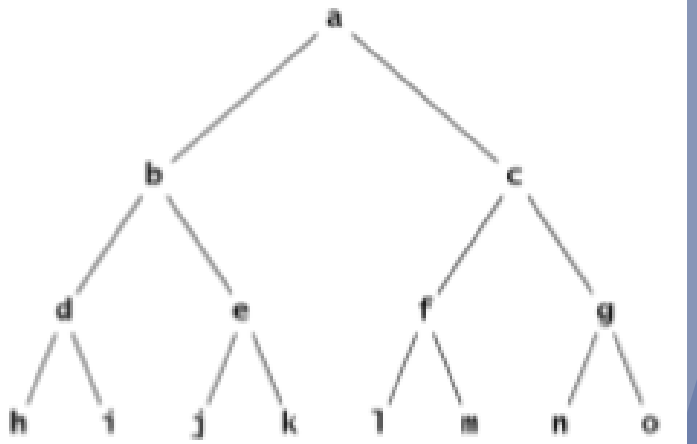


Binary trees properties – Example

- Path (a, b, d) is valid?
  - Yes. Length of the path?
    - 2
- Level 2 of the tree?
  - $\{ d, e, f \}$
- Depth of node c?
  - 1
- Subtree rooted at c?
  - $c \rightarrow f \rightarrow m$



Full binary trees



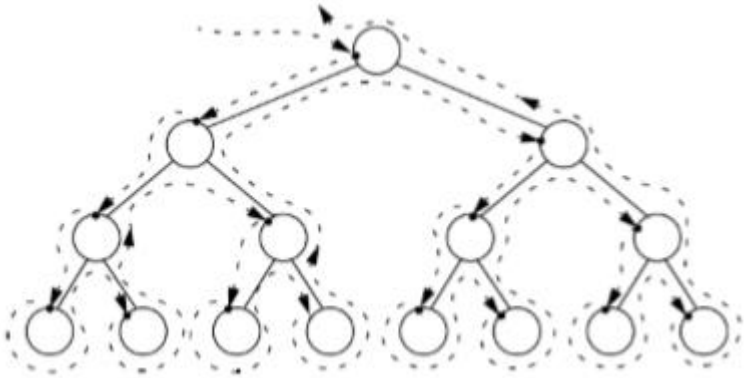
- Consider the full binary tree of height $h = 3$
  - **FULL** = all *leaves* are at the *same level* and every interior node has two children
- The full binary tree of height h has $l = 2^h$ leaves and $m = 2^h - 1$ internal nodes
  - $l = 2^h = 2^3 = 8$ leaves
  - $m = 2^h - 1 = 2^3 - 1 = 7$ internal nodes
  - Total of $n = 2^{h+1} - 1 = 2^4 - 1 = 16 - 1 = 15$ nodes
- The full binary tree with n nodes has height $h = \log_2(n+1) - 1$
  - Height $h = \log_2(n+1) - 1 = \log_2(15+1) - 1 = \log_2(16) - 1 = 4 - 1 = 3$

Traversing a binary tree

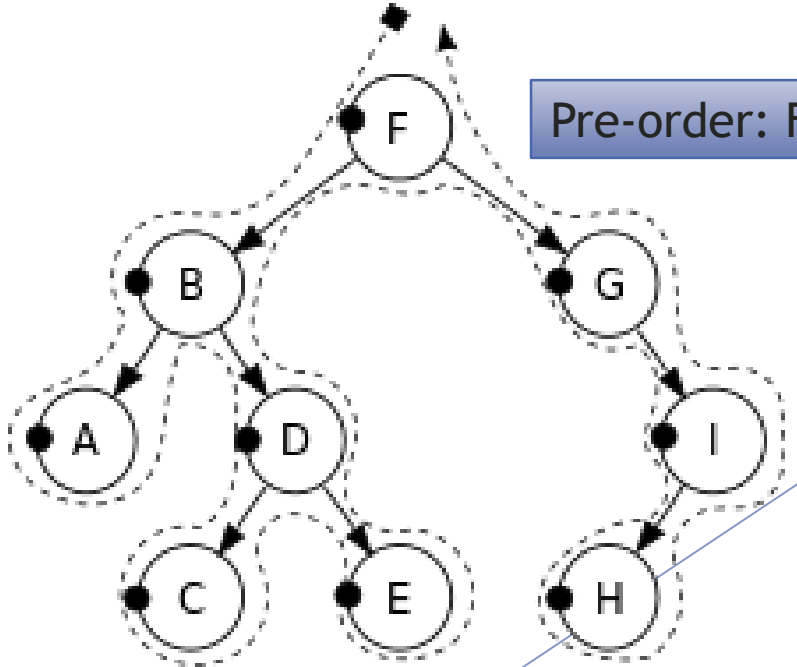
- **Tree traversal** ⇒ process of visiting (examining and/or updating, printing, …) each node in a tree data structure, exactly once, in a systematic way

- Possible traversal algorithms (classified by the order in which nodes are visited)
  - Pre-order
  - In-order (symmetric)
  - Post-order

Traversing a binary tree



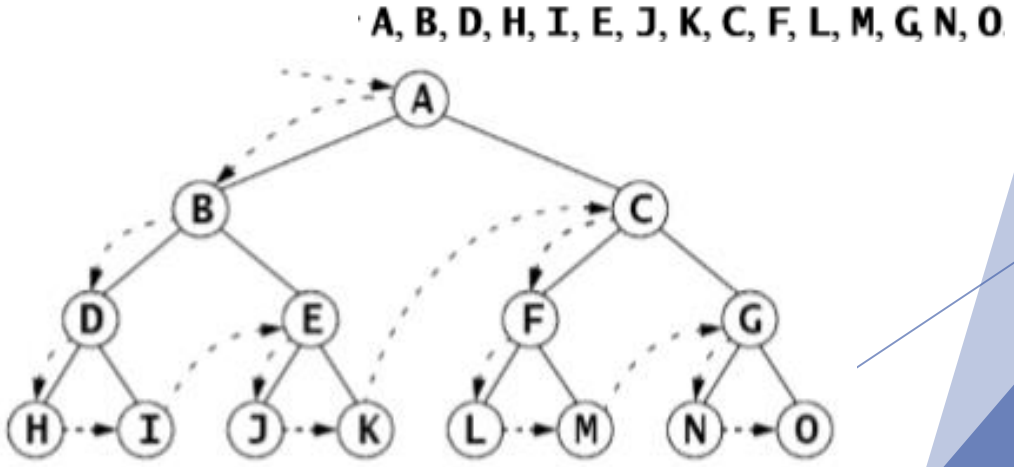
- **Pre-order**
  - **Visit the root**
  - **Traverse the left subtree**
  - **Traverse the right subtree**
- In-order
  - Traverse the left subtree
  - Visit the root
  - Traverse the right subtree
- Post-order
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the root



Pre-order: F, B, A, D, C, E, G, I, H

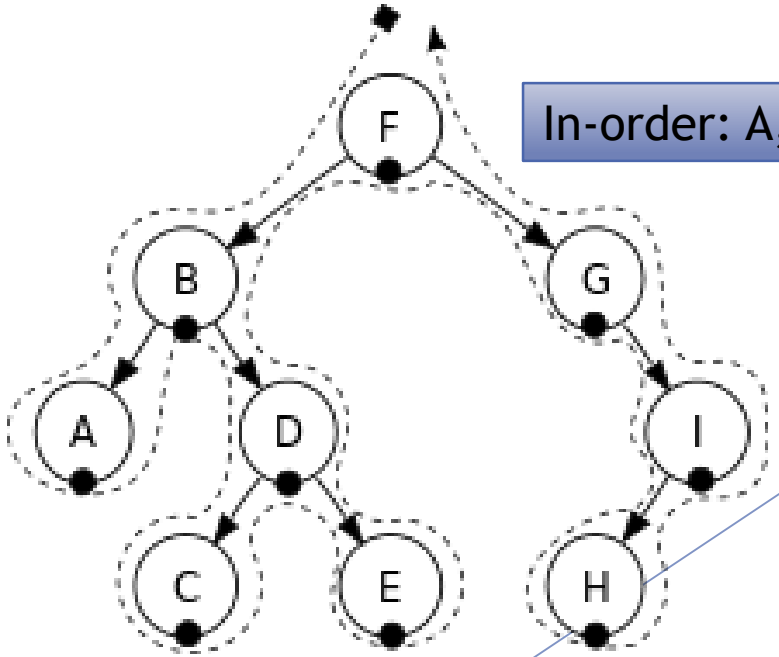
Traversing a binary tree

- **Pre-order**
  - **Visit the root**
  - **Traverse the left subtree**
  - **Traverse the right subtree**
- In-order
  - Traverse the left subtree
  - Visit the root
  - Traverse the right subtree
- Post-order
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the root



Traversing a binary tree

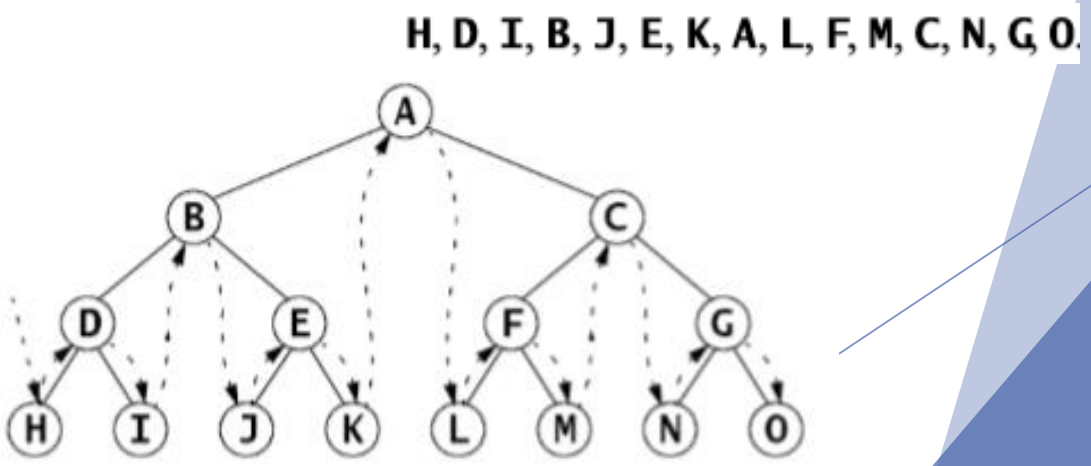
- Pre-order
  - Visit the root
  - Traverse the left subtree
  - Traverse the right subtree
- **In-order**
  - **Traverse the left subtree**
  - **Visit the root**
  - **Traverse the right subtree**
- Post-order
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the root



In-order: A, B, C, D, E, F, G, H, I

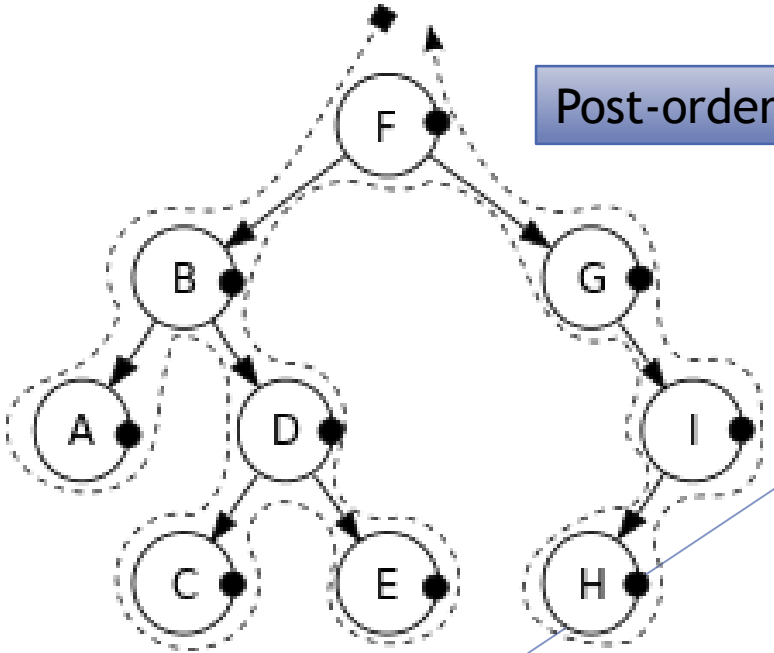
Traversing a binary tree

- Pre-order
  - Visit the root
  - Traverse the left subtree
  - Traverse the right subtree
- Post-order
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the root
- **In-order**
  - **Traverse the left subtree**
  - **Visit the root**
  - **Traverse the right subtree**



Traversing a binary tree

- Pre-order
  - Visit the root
  - Traverse the left subtree
  - Traverse the right subtree
- In-order
  - Traverse the left subtree
  - Visit the root
  - Traverse the right subtree
- **Post-order**
  - **Traverse the left subtree**
  - **Traverse the right subtree**
  - **Visit the root**

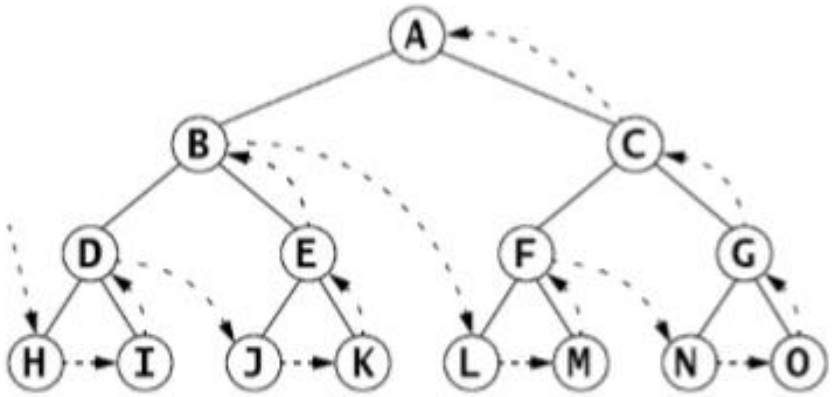


Post-order: A, C, E, D, B, H, I, G, F

Traversing a binary tree

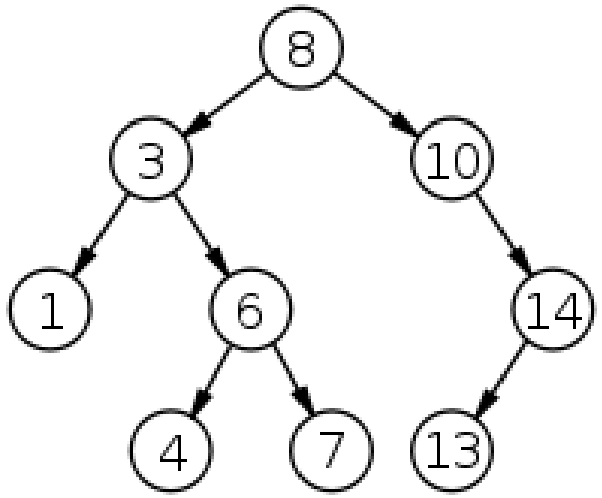
- Pre-order
  - Visit the root
  - Traverse the left subtree
  - Traverse the right subtree
- In-order
  - Traverse the left subtree
  - Visit the root
  - Traverse the right subtree
- **Post-order**
  - **Traverse the left subtree**
  - **Traverse the right subtree**
  - **Visit the root**

H, I, D, J, K, E, B, L, M, F, N, O, G, C, A.



Traversing a binary tree

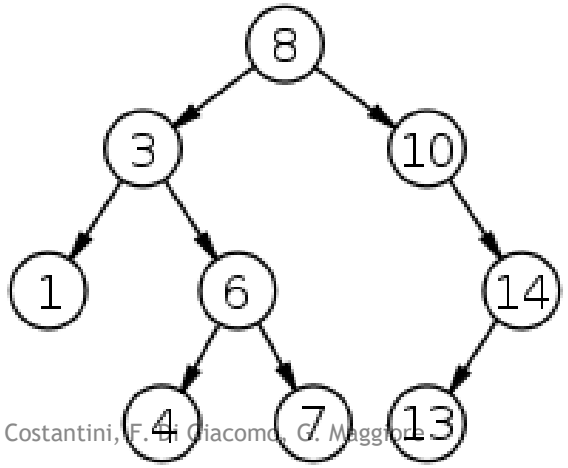
- Exercise
  - Show the resulting sequence of numbers obtained by traversing the following binary tree using the following traversal algorithms
    - Preorder
    - Inorder
    - Postorder

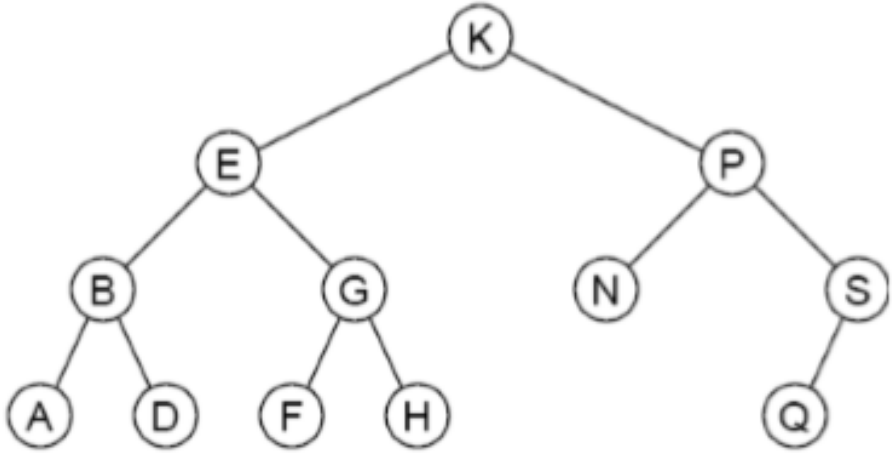


Binary Search Trees

Binary search tree definition

- A **binary search tree** is a binary tree whose elements include a key field of some *ordinal* type and which has this property:
  - If k is the key value at any node, then $k \geq x$ for every key x in the node's left subtree and $k \leq y$ for every key y in the node's right subtree
  - Also called "BST property"



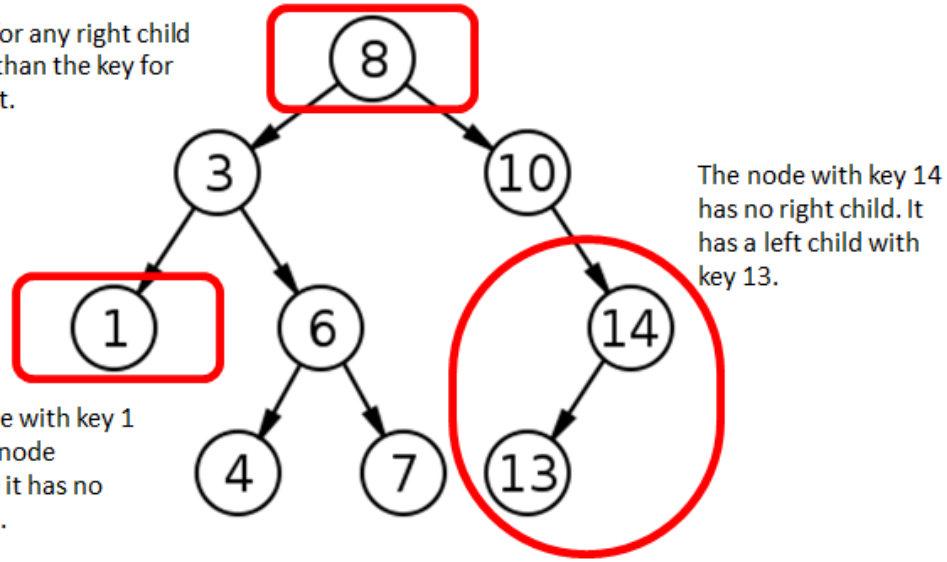


Binary search tree

The key for any left child is smaller than the key for its parent.

The key for any right child is larger than the key for its parent.

The "root" node is at the top of the tree. Here, its key is 8.

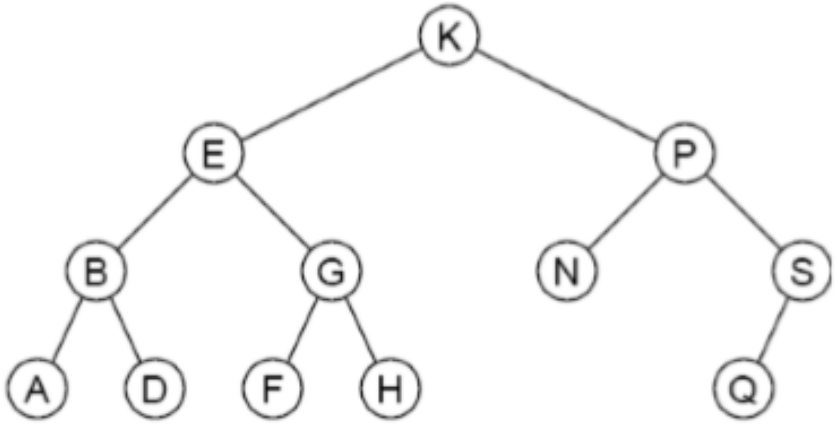


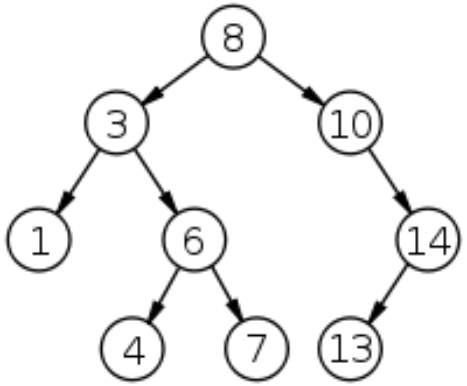
The node with key 14 has no right child. It has a left child with key 13.

The node with key 1 is a leaf node because it has no children.

Binary search tree property

- Because of the BST property…
  - in-order traversal ⇒ **sorted sequence** of elements in increasing order





- $A, B, D, E, F, G, H, K, N, P, Q, S$

$1, 3, 4, 6, 7, 8, 10, 13, 14$

Binary search tree operations

- What can we do with a binary search tree?
  - **Search** if it contains a specific element
  - **Insert** a new element
  - **Delete** a specific element

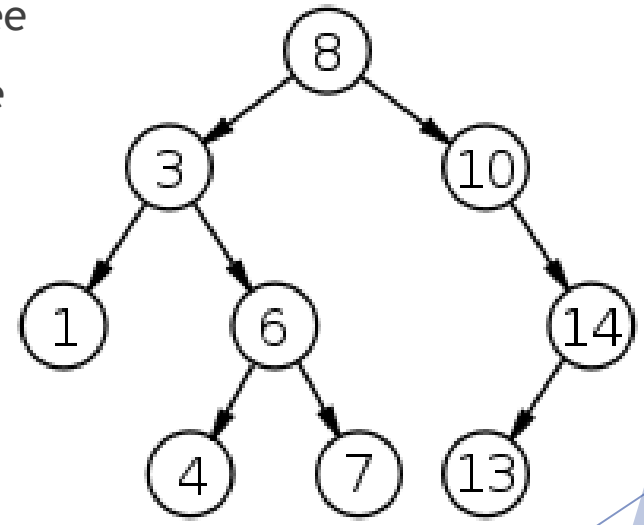
Binary search tree – Search

- We look for a specific value v
- Recursive definition (starting from the root node)

- Examine the current node n
  - If it's null → value v not found in the tree
  - If its value is equal to v → return the node n
  - Otherwise…
    - if its value is greater than v → continue the search (recursive call) on the *left* subtree
    - if its value is smaller than v → continue the search (recursive call) on the *right* subtree

Binary search tree – Search

- Example, looking for 13
  - Examine the root: $8 < 13$ → search in the right subtree
  - Examine the root: $10 < 13$ → search in the right subtree
  - Examine the root: $14 > 13$ → search in the left subtree
  - Examine the root: $13 = 13$ → value found!



- Example, looking for 2
  - Examine the root: $8 > 2$ → search in the left subtree
  - Examine the root: $3 > 2$ → search in the left subtree
  - Examine the root: $1 < 2$ → search in the right subtree
  - Examine the root: null → value not found!

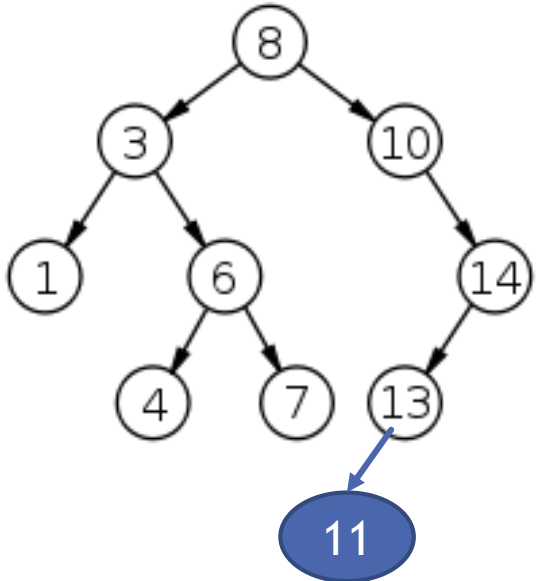
Binary search tree – Insertion

- We want to insert the value v in the tree, maintaining the BST property
- Recursive definition, similar to the search

- Examine the current node n (starting with the root)
  - If its value is greater than v
    - If n has a left child n_L → continue the insertion (recursive call) on the *left* subtree
    - Otherwise → insert the new node (with value v) as n's left child
  - If its value is smaller than v
    - If n has a right child n_R → continue the insertion (recursive call) on the *right* subtree
    - Otherwise → insert the new node (with value v) as n's right child

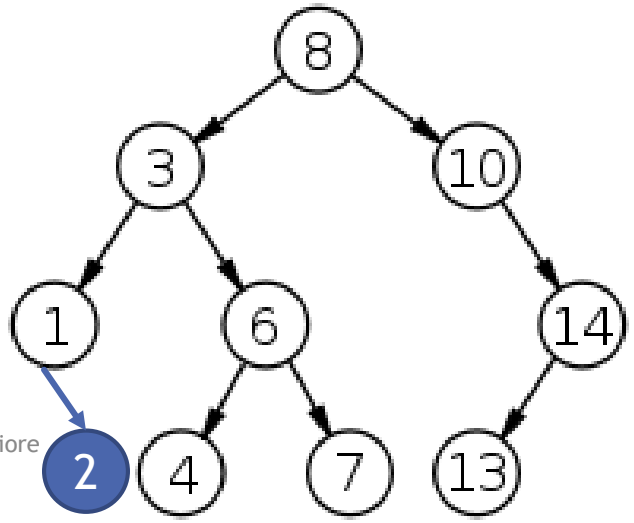
Binary search tree – Insertion

- Example, inserting 11
  - Examine the root: $8 < 11$ → there is a right child → insert in the right subtree
  - Examine the root: $10 < 11$ → there is a right child → insert in the right subtree
  - Examine the root: $14 > 11$ → there is a left child → insert it in the left subtree
  - Examine the root: $13 > 11$ → there is *no* left child → create the new node as left child of 13



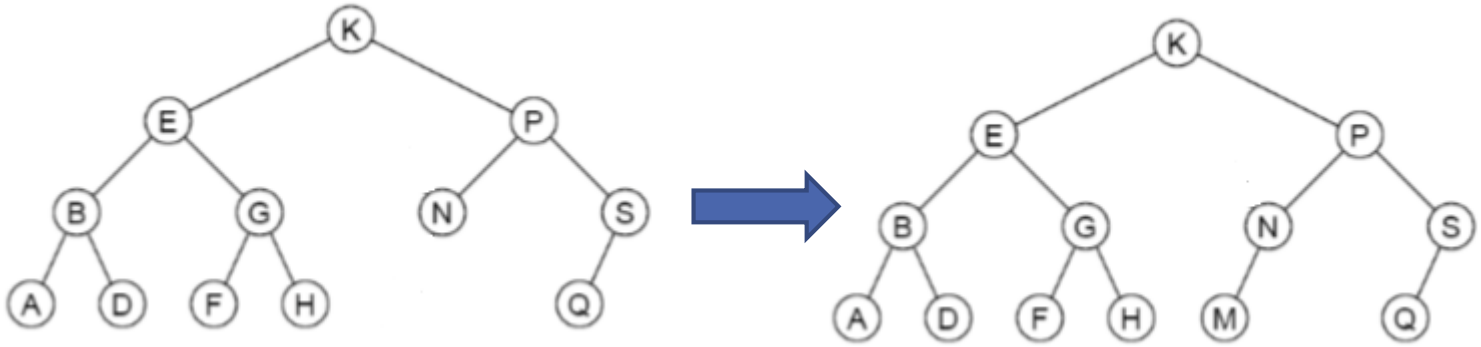
Binary search tree – Insertion

- Example, inserting 2
  - Examine the root: $8 > 2$ → there is a left child → insert in the left subtree
  - Examine the root: $3 > 2$ → there is a left child → insert in the left subtree
  - Examine the root: $1 < 2$ → there is *no* right child → create the new node as right child of 1



Binary search tree – Insertion

- Example, inserting M
  - Examine the root: $K < M$ → there is a right child → insert in the right subtree
  - Examine the root: $P > M$ → there is a left child → insert in the left subtree
  - Examine the root: $N > M$ → there is *no* left child → create the new node as left child of N



Binary search tree – Insertion

- Exercise
  - Given the following sequence of numbers, draw a binary search tree by inserting such numbers from left to right.

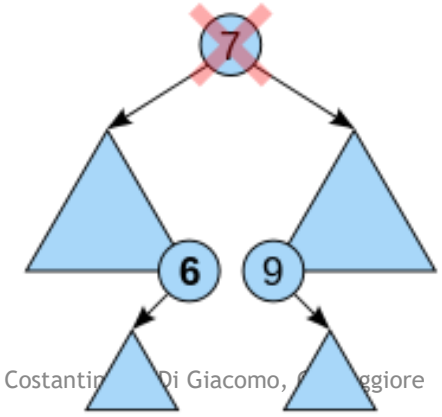
$$11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31$$

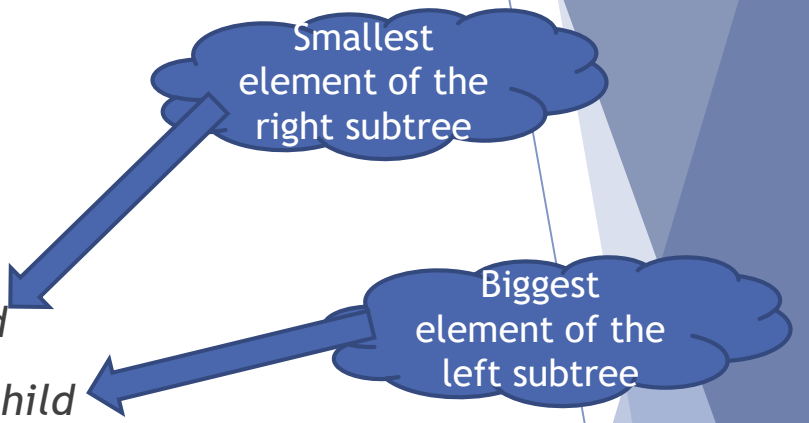
Binary search tree – Deletion

- We want to remove the node with value v
- Three cases
  1. **Deleting a leaf** → we can simply remove it from the tree (easy!)
  2. **Deleting a node with one child** → remove the node and replace it with its child
  3. **Deleting a node with two children** → more complicated recursive procedure
     - call the node to be deleted n but do not delete n
     - choose either its in-order successor node or its in-order predecessor node, r
     - copy the value of r to n, then recursively call delete on r until reaching one of the first two cases

Binary search tree – Deletion

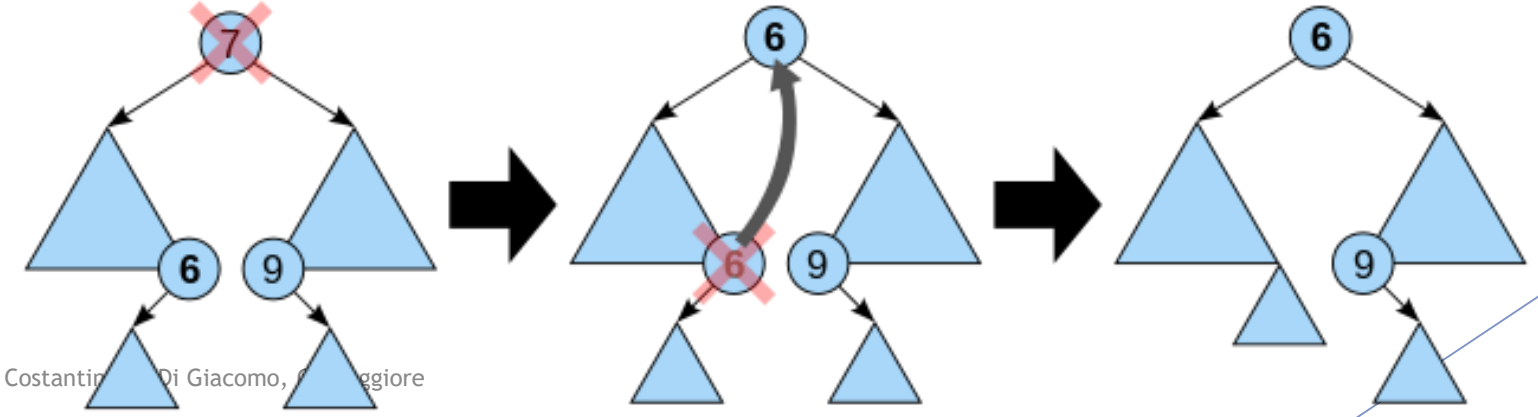
- Case 3 (node with two children) is the most difficult
- As with all binary trees…
  - a node's *in-order successor* is its *right subtree's left-most child*
  - a node's *in-order predecessor* is the *left subtree's right-most child*





Binary search tree – Deletion

- Case 3 (node with two children) is the most difficult
- As with all binary trees…
  - a node's *in-order successor* is its *right subtree's left-most child*
  - a node's *in-order predecessor* is the *left subtree's right-most child*
- In either case, this node will have zero or one children. Delete it according to one of the two simpler cases.



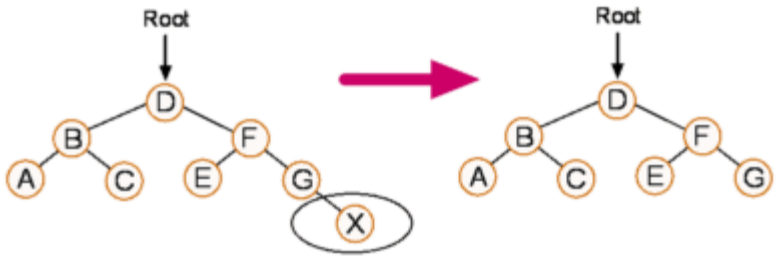
Smallest element of the right subtree

Biggest element of the left subtree

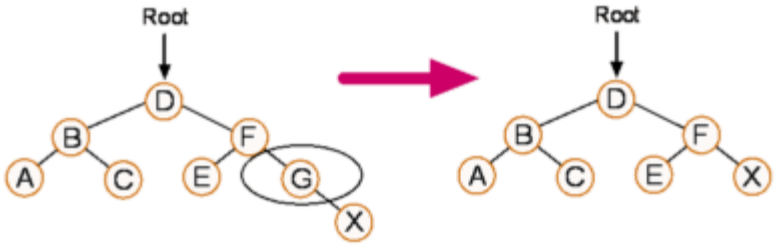
Binary search tree – Deletion

- Examples

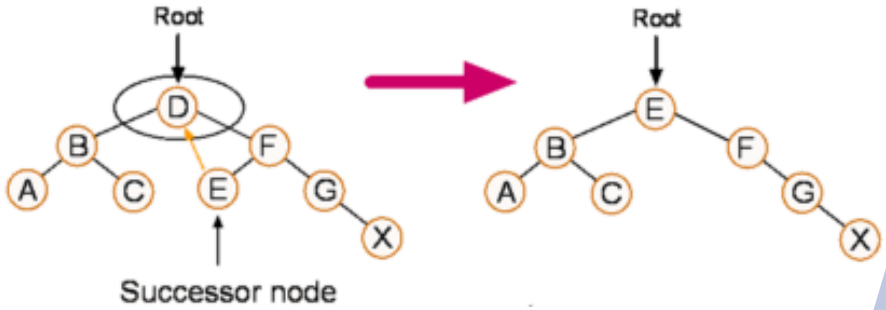
Leaf Deletion



Deleting a node with a single child

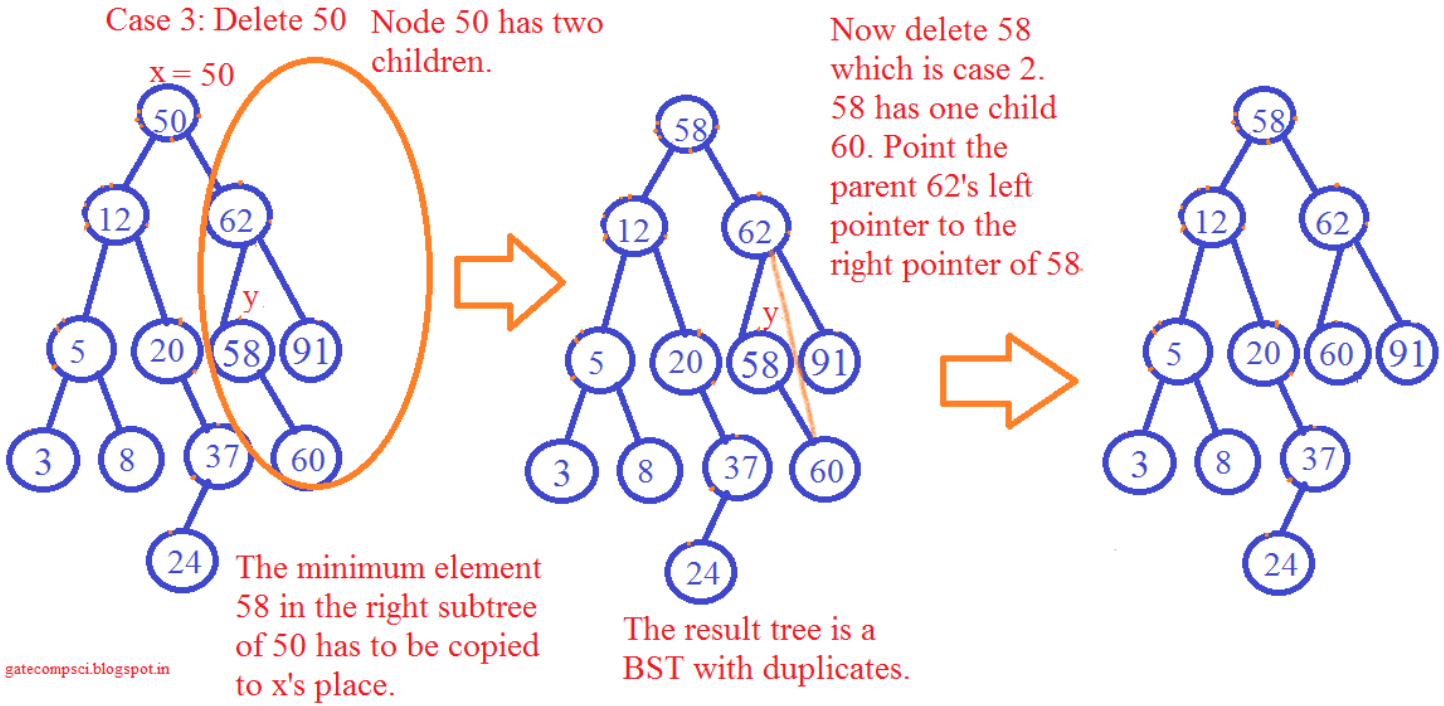


Deleting a node with two children, locate the successor node on the right-hand side (or predecessor on the left) and replace the deleted node (D) with the successor (E). Finally remove the successor node.



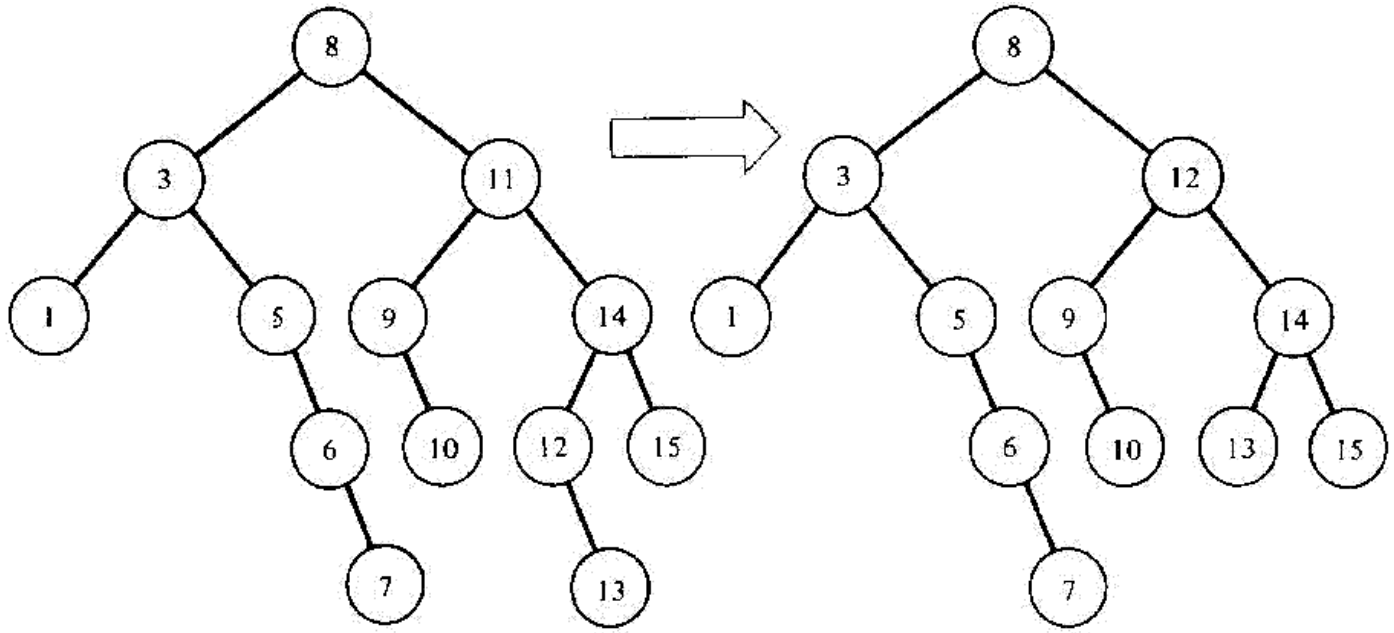
Binary search tree – Deletion

- Example, deleting 50



Binary search tree – Deletion

- Example, deleting 11



Binary search tree – Deletion

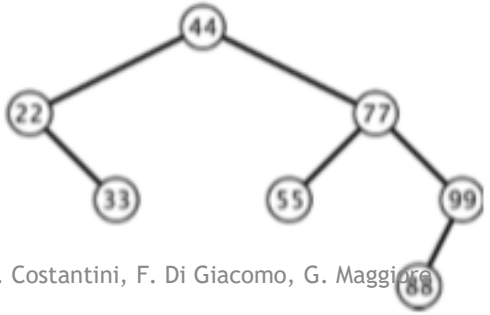
- Exercise
  - Given the following sequence of numbers, draw a binary search tree by inserting such numbers from left to right and then show the two trees that can be the result after the removal of 11.

$$11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31$$

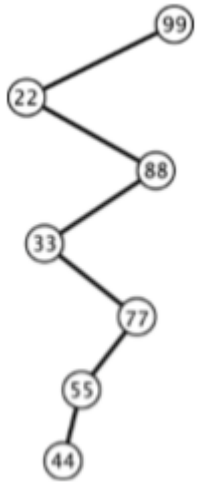
Binary search tree – Balanced vs unbalanced

- Very efficient searching if a binary search tree is balanced
- Without further restrictions, a binary search tree may grow to be very unbalanced
  - worst case when the elements are inserted in sorted order → the tree becomes almost linear

- Inserting the input sequence
  - 44, 22, 77, 55, 99, 88, 33



- Inserting the input sequence
  - 99, 22, 88, 33, 77, 55, 44



Binary search tree – Performance

- Computational complexity of the operations
  - *Insertion* & *search* → begin at the root of the tree and proceed down toward the leaves, making one comparison at each level of the tree
  - *Deletion* → this operation does not always traverse the tree down to a leaf, but it is always a possibility
- … Thus, the time required to execute each algorithm is proportional to the height h of the tree
  - Height of a binary search tree?
    - On average, binary search trees with n nodes have $O(\log n)$ height
    - In the worst case, binary search trees can have $O(n)$ height (the most unbalanced tree is like a linked list)

Operation	Average case	Worst case
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$

BST in .NET

- http://msdn.microsoft.com/en-us/library/ms379572(v=vs.80).aspx

- C#: **SortedSet**
  - http://msdn.microsoft.com/en-us/library/dd412070.aspx
  - It is implemented using a self-balancing red-black tree that gives a performance complexity of $O(\log n)$ for insert, delete, and lookup. It is used to keep the elements in sorted order, to get the subset of elements in a particular range, or to get the Min or Max element of the set.

2-3 trees

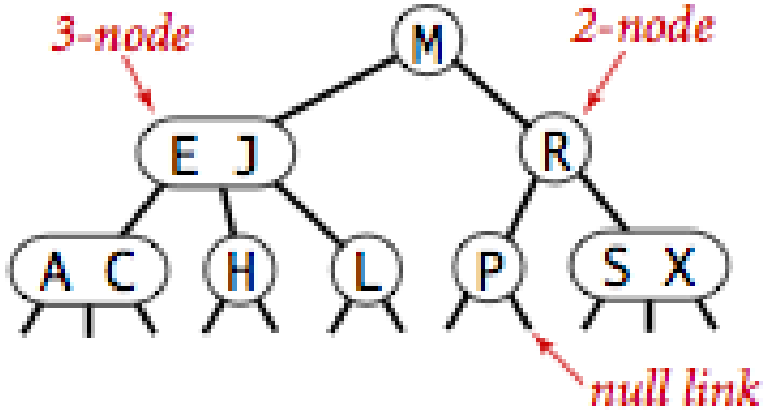
2-3 trees

- As we saw before, BST could be very unbalanced ⇒ bad for performance!

- **2-3 trees**
  - a type of binary search tree where costs are *guaranteed* to be logarithmic
  - near-perfect balance achieved through allowing nodes to *hold more than one key*

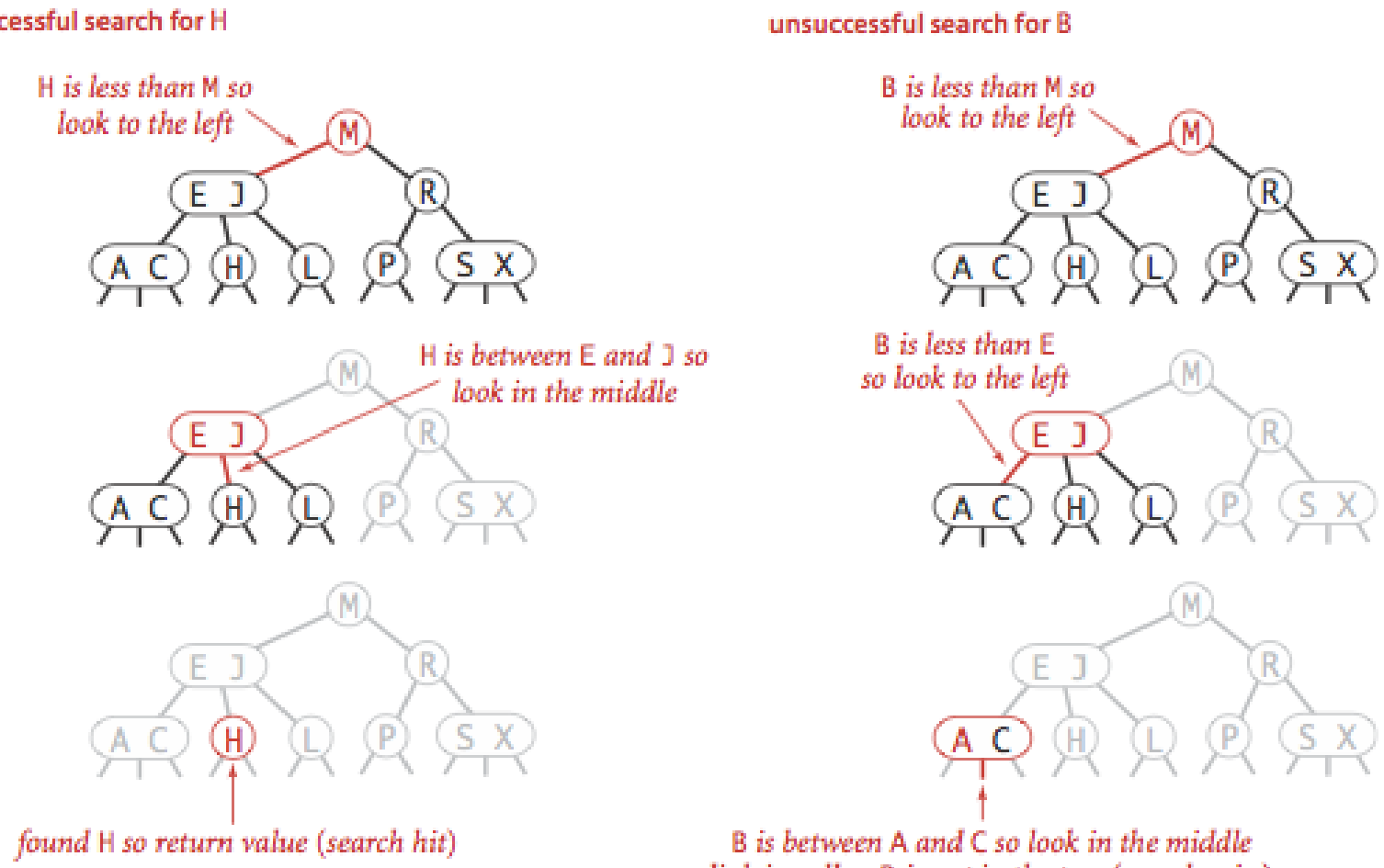
2-3 trees

- *Definition*: a 2-3 search tree is a tree that either is empty or…
  - A **2-node**, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
  - A **3-node**, with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys and a right link to a 2-3 search tree with larger keys.
  - A *perfectly balanced* 2-3 search tree (or 2-3 tree for short) is one whose null links are all the same distance from the root.



Anatomy of a 2-3 search tree

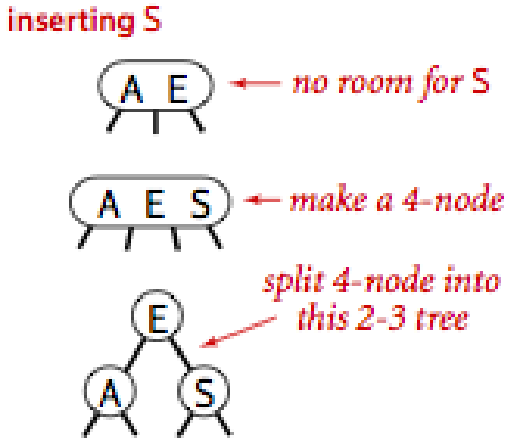
2-3 trees search



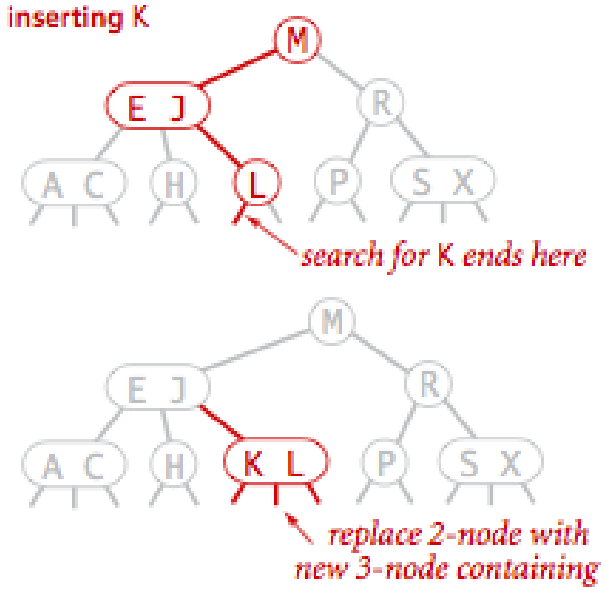
Search hit (left) and search miss (right) in a 2-3 tree

2-3 trees insertion

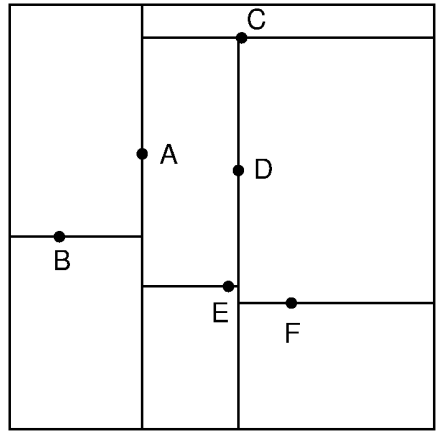
- A bit more complicated (more cases to consider)
- The simplest two cases …
  - *Insert into a 2-node*
  - *Insert into a tree consisting of a single 3-node*



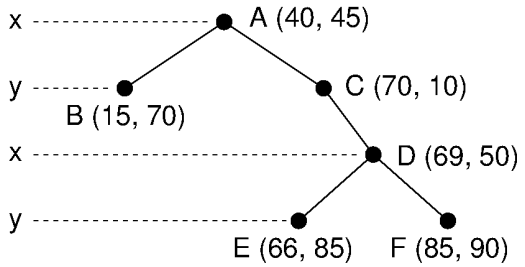
Insert into a single 3-node



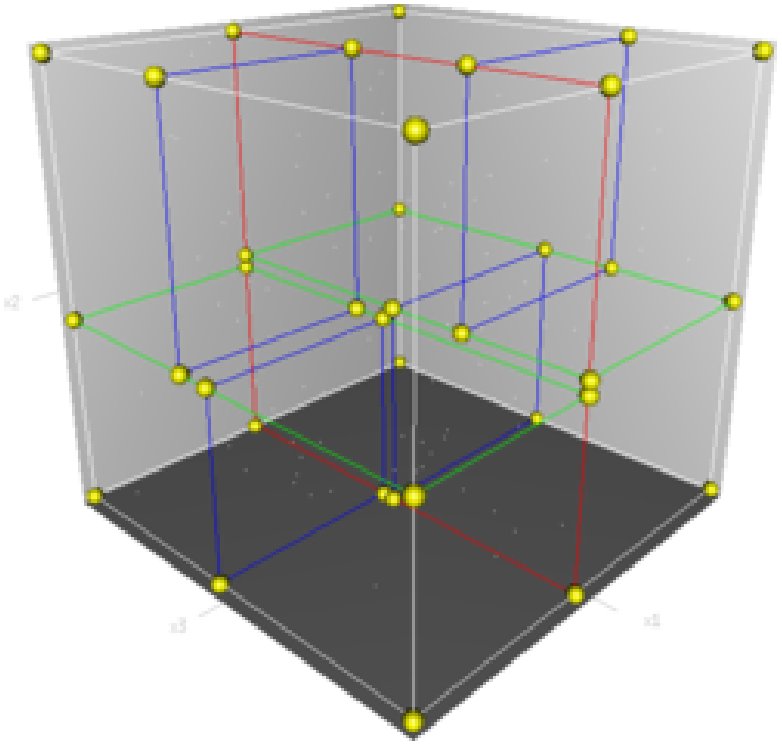
Insert into a 2-node



(a)



(b)



k-d trees

K-dimensional trees

k-d trees

- What is a k-d tree?
  - special case of binary space partitioning trees
  - space-partitioning data structure for organizing points in a k-dimensional space
  - useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches)
    - Range search == Exercise 2 of the assignment!

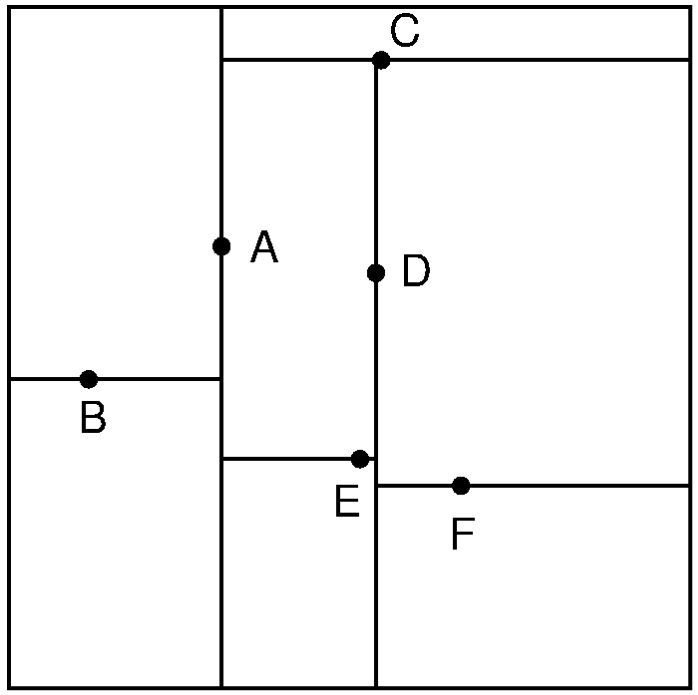
k-d trees informal description

- Binary tree in which every node is a k-dimensional point
- Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces
  - Points to the left of this hyperplane are represented by the left subtree of that node
  - Points right of the hyperplane are represented by the right subtree
- Every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis
  - for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree

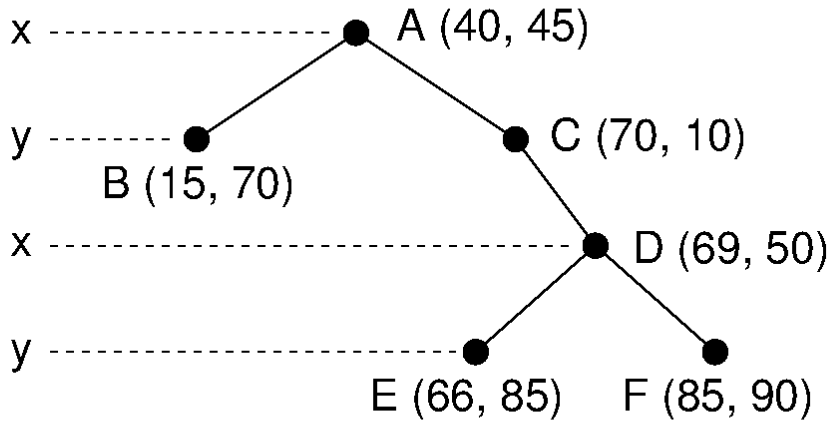
k-d trees construction

- The canonical method of k-d tree construction has the following constraints:
  - As one moves down the tree, one cycles through the axes used to select the splitting planes
    - Example: in a 2-d tree, the root has an **x**-aligned plane, the root's children would both have a **y**-aligned plane, the root's grandchildren would have all **x**-aligned planes, etc…
  - Points are inserted by selecting the *median* of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane
- This method leads to a *balanced k-d tree* (i.e., each leaf node is approximately at the same distance from the root)
  - Note: if you do not choose the median, there is no guarantee that the tree will be balanced. It is *not required* in the assignment to choose the median.

2-d trees example

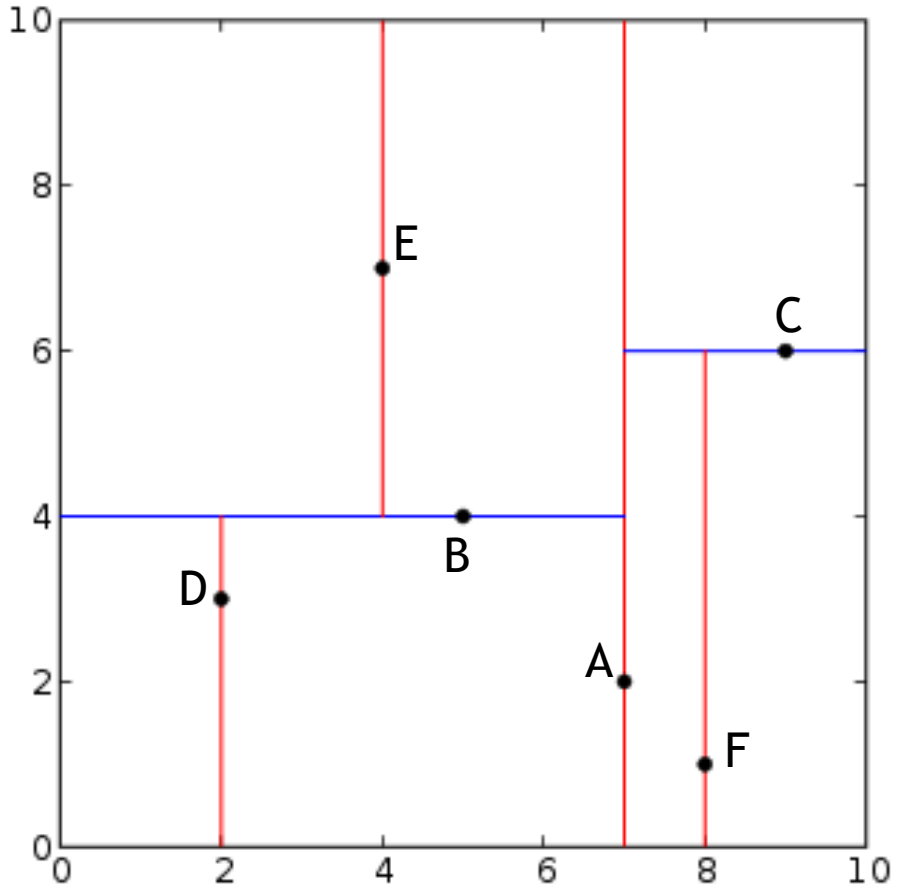


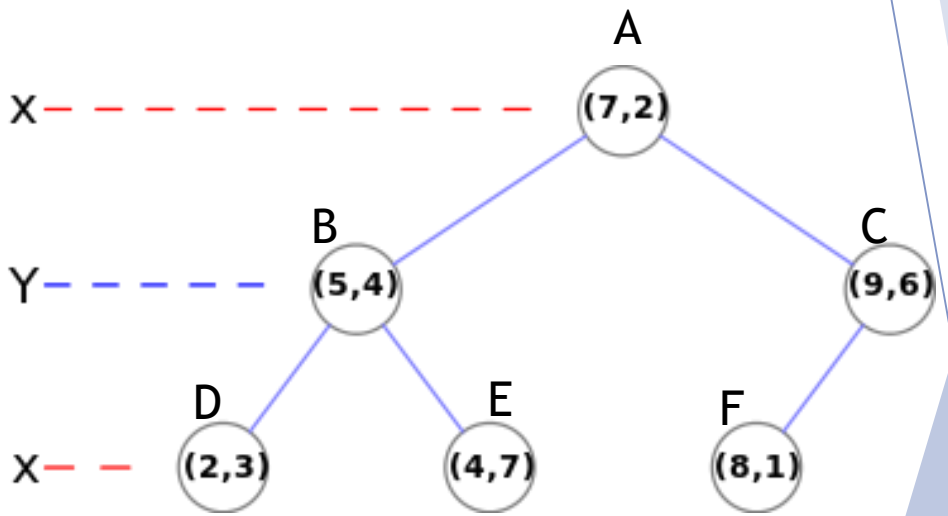
(a)



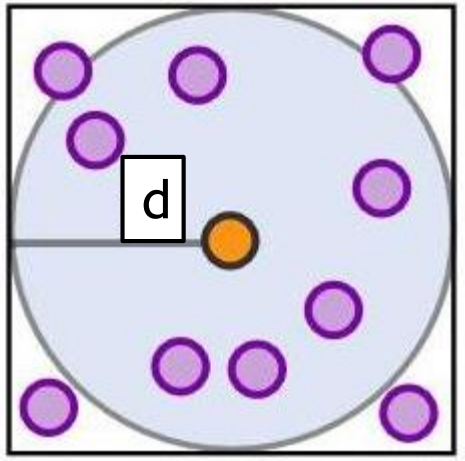
(b)

2-d trees example





k-d trees in the assignment



- A range search searches for *ranges of parameters*
  - Example: if a tree is storing values corresponding to income and age, then a range search might be something like looking for all members of the tree which have an age between 20 and 50 years and an income between 50,000 and 80,000
- Since k-d trees divide the range of a domain in half at each level of the tree, they are useful for performing range searches
  - In the assignment you have to look for special buildings within a certain distance d from a house… this is exactly a range search, where the x coordinate of the building should be between $[x_h - d, x_h + d]$ and the y coordinate between $[y_h - d, y_h + d]$ (supposing that x_h and y_h are the coordinates of the house). Before returning them, check if the buildings found are really within radius d from the house (using the Euclidean distance).
  - Create the tree only *once* and then use it for all the searches to do (one for each given house).

Homework

- ***GO ON WITH THE ASSIGNMENT!!!***
  - ***FINISH EXERCISE 1***
  - ***START EXERCISE 2***