DATA STRUCTURES AND ALGORITHMS

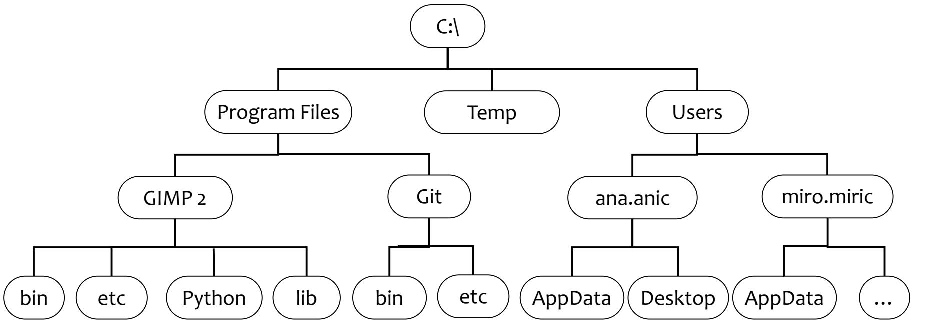
Lecture 09

Learning outcome 3

TREES

Introduction

- All previous data structures have been linearly arranged
  - Can we store the following data in vector or list:



  - We can't because the data is hierarchical in nature
    - We need a new structure - a tree

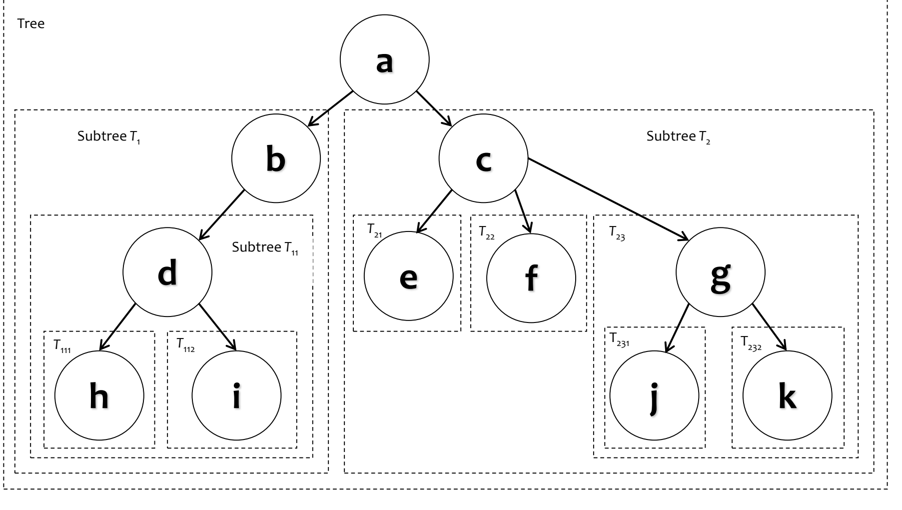
Application of trees

- Trees are suitable for:
  - Storage of hierarchical data
    - Family tree
    - Sports competitions
    - File system
    - Organizational chart of the company
    - Organization chart of the army
  - Storing non-hierarchical data in a searchable form
    - Indexes in databases

Simplified tree definition

- A tree is a group of connected nodes with the following properties:
  - Each node contains one or more values
  - Nodes are hierarchically organized (parent - children)
  - There is exactly one node that has no parents and is called the tree root
  - Each node is also a subtree root, and this subtree can be complex (composed of several nodes) or trivial (composed of only 1 node)

An example of a tree



Basic terms (1/4)

- Nodes that are located directly below a node are called its children
  - For example, nodes e, f, and g are the children of node c
- Except for the root of the tree, each node has exactly one parent, and that is the node directly above it
  - For example, the parent of node h is node d
  - Each node can have several children, but at most one parent
- Nodes with the same parent are called siblings
  - For example, nodes e, f, and g are siblings

Basic terms (2/4)

- The path from node x to node y is a series of nodes that are traversed when going from x to y (where each node on the path is the parent of the next node on that path)
  - For example, the path from a to k is: a, c, g, k; the path from e to g does not exist
- If a path consists of n nodes, then the length of that path is equal to n - 1
  - For example, the path length of a path a, c, g, k is 3
- If we look at some node x:
  - The descendants of the node x are all nodes in the tree to which there is a path from x
  - The ancestors of node x are all nodes in the tree from which there is a path to x

Basic terms (3/4)

- A leaf is a node that has no children
  - For example, nodes h, i, e, f, j, k are leaves
- An internal node is a node that has children
  - For example, nodes a, b, c, d, g are internal
- Node level or node depth represents its distance (path length) from the root
  - Root has level 0, his children have level 1, their children have level 2, and so on
- The depth of a tree is equal to the maximum node level in the tree
  - For example, the depth of our tree is 3

Basic terms (4/4)

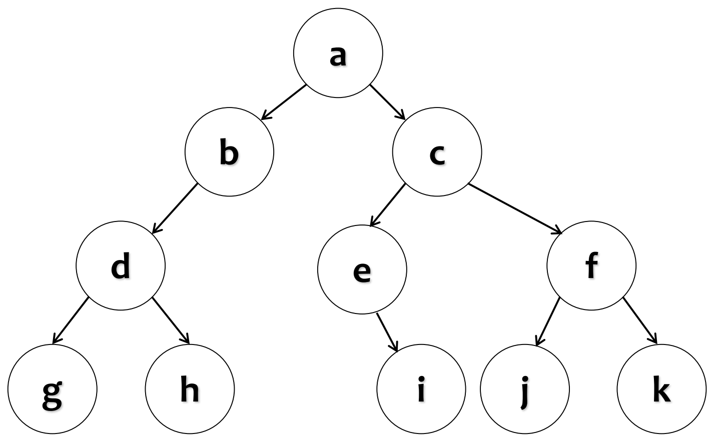
- The degree of the node is equal to the number of his children
  - For example, degree of node a is 2, degree of node c is 3
- The degree of the tree is equal to the degree of the node with the most children
  - For example, the degree of our tree is 3

BINARY TREES

Introduction

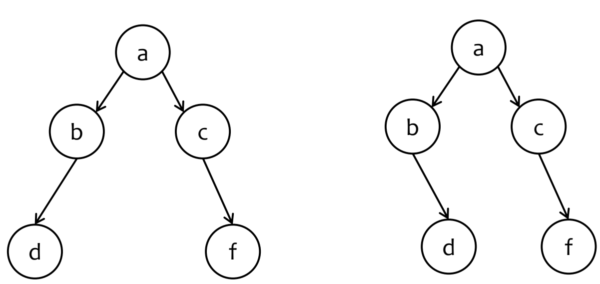
- A binary tree is a tree whose degree can be a maximum of 2
  - This means that each node can have a maximum of two children
- Binary trees are a subset of general trees
  - Usually working with them is easier than working with general trees
- The terms introduced for general trees are used in the same way for binary trees

An example of a binary tree



The difference between a left and a right child

- We distinguish the left and right child of each node
- If a node has only one child, it does matter if it is a left or a right child
    - The next two binary trees are not equal

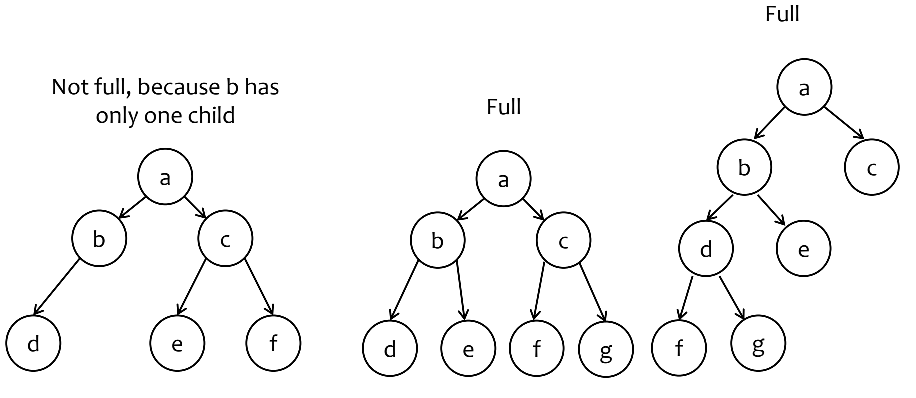


Left child Right child

Types of binary trees

- We are interested in the following types of binary trees:
  - A **full** binary tree is one in which each non-leaf node has exactly 2 children
  - A **perfect** binary tree is one that is full and in which all the leaves are at the same level
  - A **complete** binary tree is one in which all levels (except perhaps the last) are completely filled, and the last level has all the nodes filled from the left
    - This means that nodes are added to the tree as follows:
      - We start from the root and fill each level from left to right
      - When there is no more room in current level, we move on to the next level and start filling it from the left

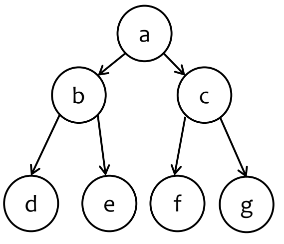
Examples of full binary trees

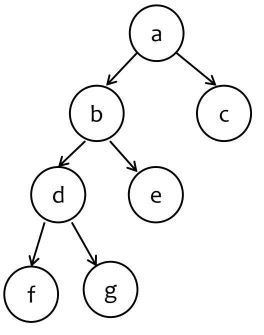


Examples of perfect binary trees

Full, but not perfect because the level of nodes f and g is equal to 3, node e is equal to 2, and node c is equal to 1

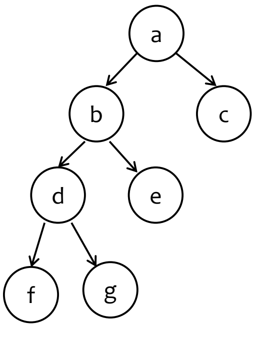
Perfect



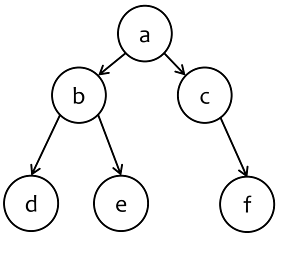


Examples of complete binary trees

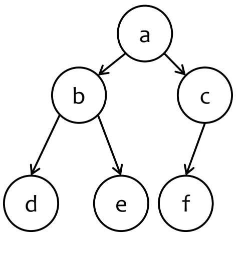
Full, but not complete



Not complete



Complete



RECURSION

Recursion

- A problem-solving method in which a function calls itself with different parameter values
  - Each function call works on a part of the problem
  - There must be a stop condition (base case)
- Each execution of the same function is called an iteration
  - Each iteration is independent of the previous one
  - In each iteration:
    - Let's solve a small part of the problem
    - We recursively call on ourselves to solve the rest of the problem
  - Check out https://www.cs.usfca.edu/~galles/visualization/RecFact.html

Example of recursion

- The task of each iteration is to display only one letter

```
void display(string name, int i) {
  if (i == name.size()) {
     return;
  }
  cout << name[i] << endl;
  display(name, i + 1);
}
int main() {
  string name = "Marko";
  display(name, 0);
  return 0;
}
```

Base case check

Solving a piece of a problem

Recursive call

Initial call

Strana ▪ 21

Problem

- Let's write a program that will display all the subfolders within the given folder.
  - We will use dirent.h
    - Available in any Linux distribution
    - For Windows: github.com/tronkko/dirent
  - Recursive function design:
    - Which part of the problem is solved by one iteration?
      - Displays the name of the current folder
    - What recursive calls we make?
      - One for each subfolder

Strana ▪ 22

The basic outline of the solution

```
void process_folder(const char* parent, const char* name, int lvl) {

    // Do this folder.

    // Create a full path.

    // Do the subfolders.
}

int main() {
    process_folder("D:\\", "Temp", 0);

    return 0;
}
```


ALGEBRA

Solution details (1/2)

```
// Do this folder.
for (int i = 0; i < lvl; i++) {
    cout << " ";
}
cout << name << endl;

// Create a full path.
stringstream sstr;
sstr << parent << name << "\\";
char full[256];
sstr >> full;

// Do the subfolders.
```


ALGEBRA

Solution details (2/2)

```
// Do the subfolders.
DIR* dir;
dirent* ent;

if ((dir = opendir(full)) == NULL) {
    return;
}

while ((ent = readdir(dir)) != NULL) {
    if (ent->d_name[0] == '.') {
        continue;
    }

    if (S_ISDIR(ent->d_type) == true) {
        process_folder(full, ent->d_name, lvl + 1); // REC!
    }
}

closedir(dir);
```


TREE TRAVERSALS

Introduction

- Tree traversal is the process of visiting all nodes of a tree under conditions:
  - We will visit every element in the tree
  - We will not visit any element two or more times
- The most common reasons for visiting are:
  - Read the contents of the element
  - Change the content of an element

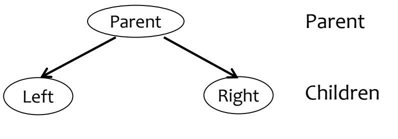
Traversal of linear structures

- Traversal of linear structures (for example, lists) is simple: we start from the first element and go to the last
- For example, if we have a list of 50 integers
  - If we want to calculate the sum of all elements, we will go from the beginning to the end of the list and read the contents of each element
  - If we want to multiply each number by 2, we will go from the beginning to the end of the list and change the content of each element

Traversal of the binary tree

- Traversal of hierarchical structures is more complex and there are several ways to visit them
- The most well-known binary tree traversal algorithms are:
  - DFS algorithms (*depth-first search*)
    - INORDER
    - PREORDER
    - POSTORDER
  - BFS algorithm (*breadth-first search*)
- All algorithms start from the root and differ in the order in which nodes are visited
- DFS algorithms are recursive

DFS traversal algorithms



INORDER	Traversal principle: **Left, Parent, Right** The traversal starts from the left subtree and goes to the parent and then to the right subtree. Each subtree is traversed in the same way (recursive).
PREORDER	Traversal principle: **Parent, Left, Right** The traversal starts from the parent and goes to the left and then to the right subtree. Each subtree is traversed in the same way (recursive).
POSTORDER	Traversal principle: **Left, Right, Parent** The traversal starts from the left subtree and goes to the right subtree and finally to the parent. Each subtree is traversed in the same way (recursive).

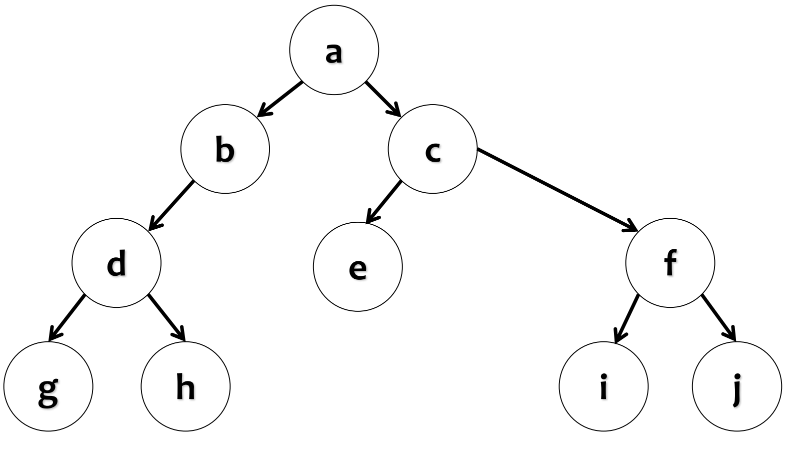
Applications of traversal algorithms

- Applications of traversal algorithms:
  - INORDER
    - It is often used on binary search trees (BST) as it returns values in sorted form (defined by the BST itself)
  - PREORDER
    - It is often used to duplicate a tree because it first visits the parents and only then the children
  - POSTORDER
    - It is often used to erase nodes and destroy trees because it visits children first and then parents

Strana ▪ 31

INORDER example (1/11)

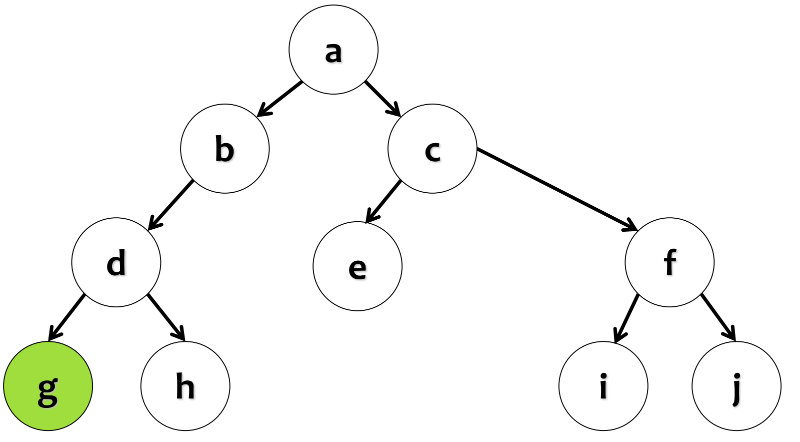
- Lets consider the binary tree:



Strana ▪ 32

INORDER example (2/11)

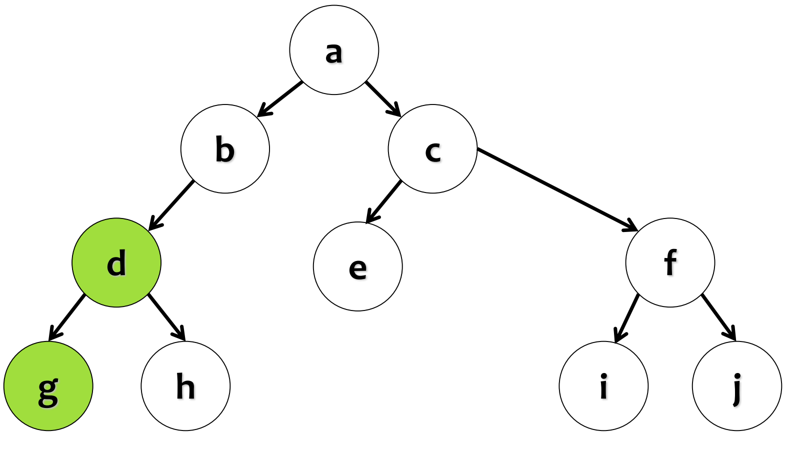
Traversal order: g



ALGEBRA

INORDER example (3/11)

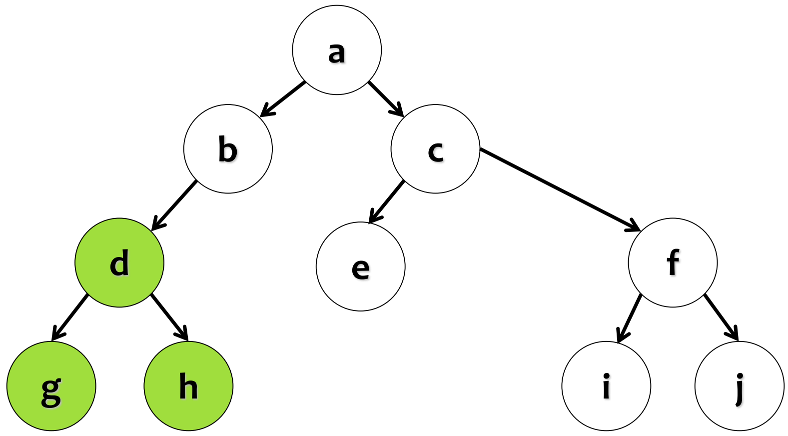
Traversal order: g, d



ALGEBRA

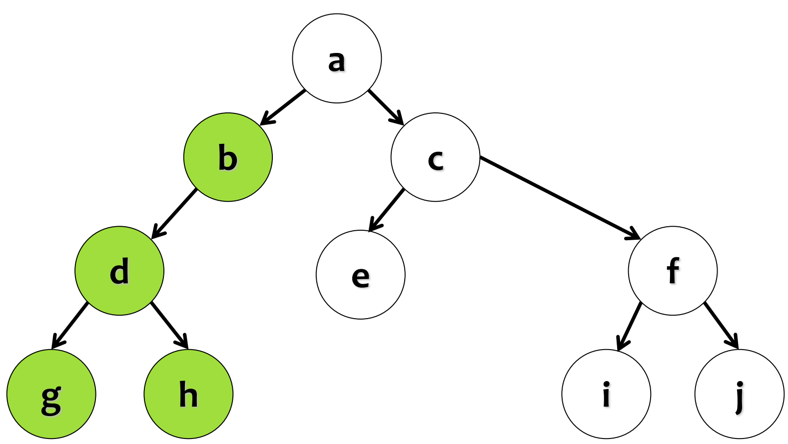
INORDER example (4/11)

Traversal order: g, d, h



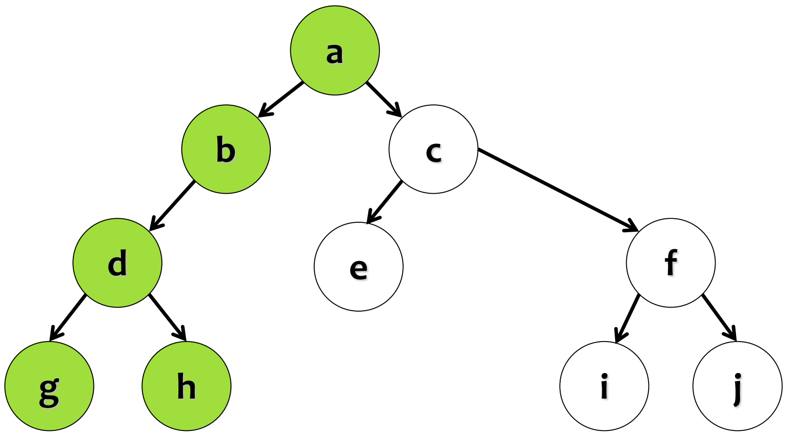
INORDER example (5/11)

Traversal order: g, d, h, b



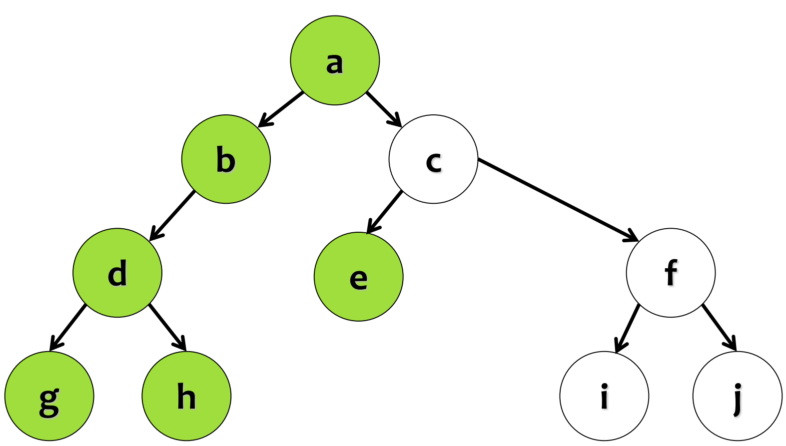
INORDER example (6/11)

Traversal order: g, d, h, b, a



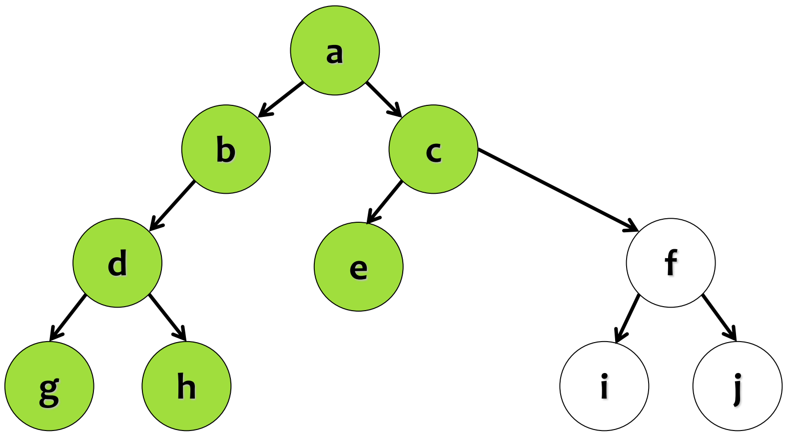
INORDER example (7/11)

Traversal order: g, d, h, b, a, e



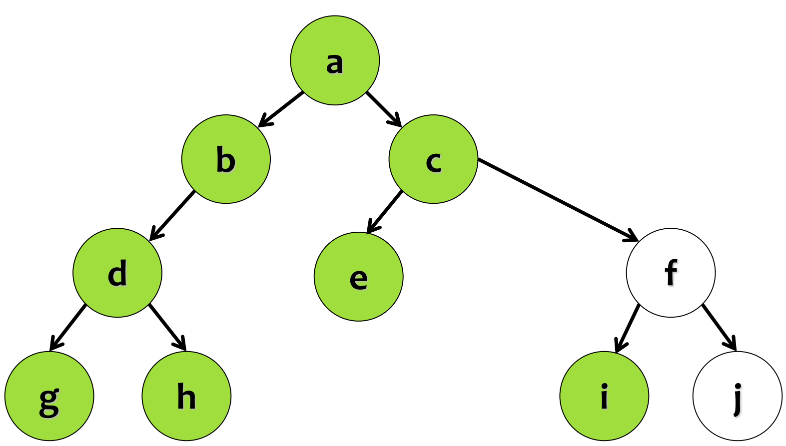
INORDER example (8/11)

Traversal order: g, d, h, b, a, e, c



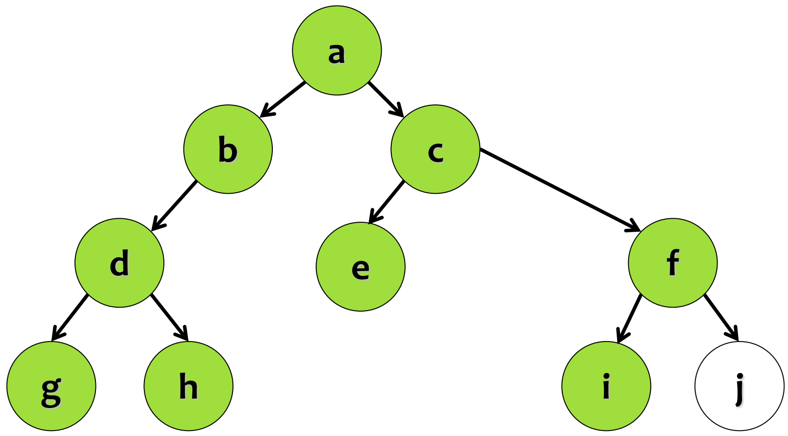
INORDER example (9/11)

Traversal order: g, d, h, b, a, e, c, i



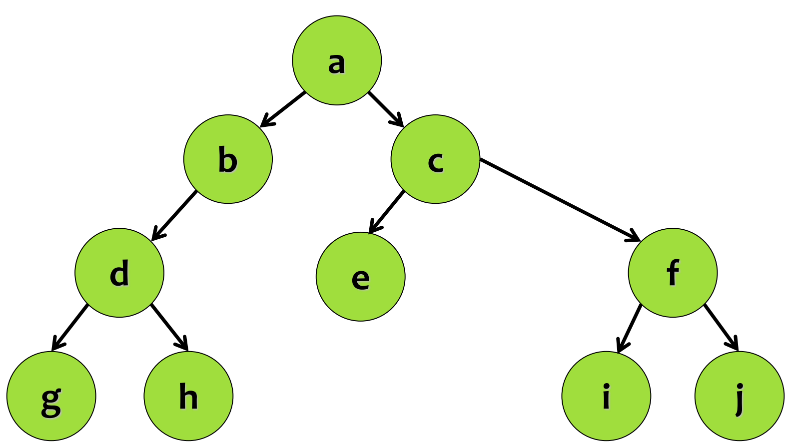
INORDER example (10/11)

Traversal order: g, d, h, b, a, e, c, i, f



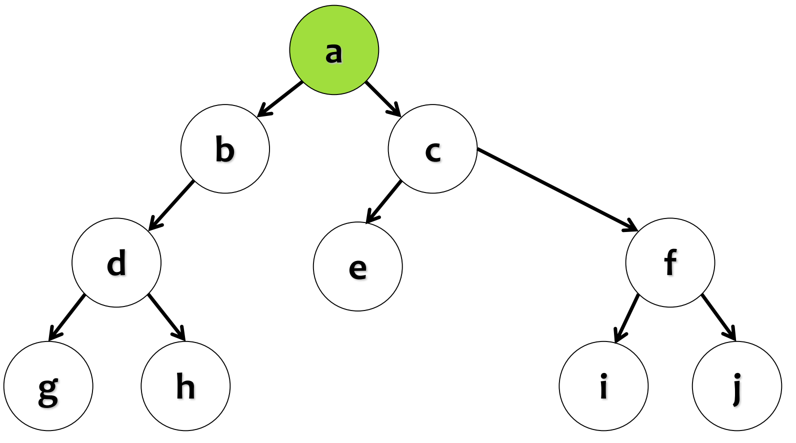
INORDER example (11/11)

Traversal order: g, d, h, b, a, e, c, i, f, j



PREORDER example (1/10)

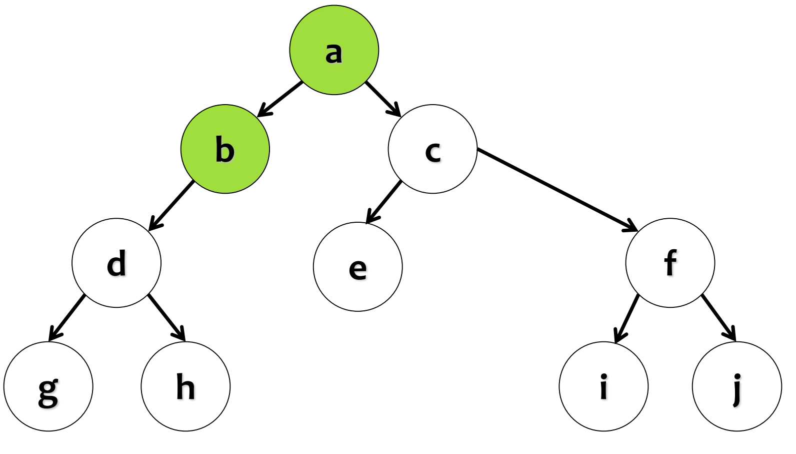
Traversal order: a



ALGEBRA

PREORDER example (2/10)

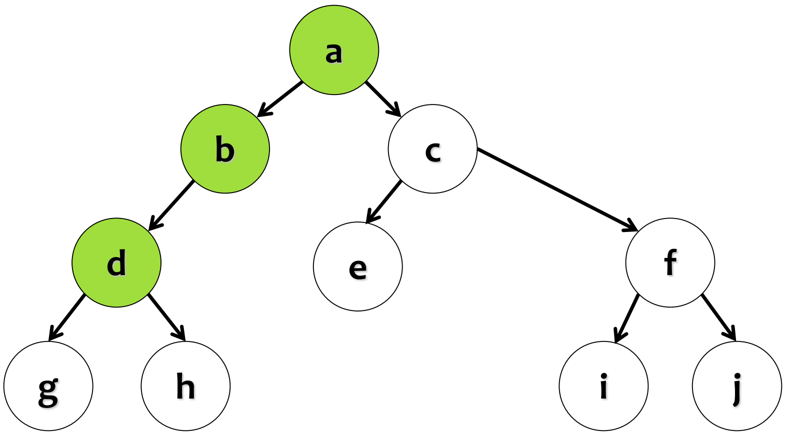
Traversal order: a, b



ALGEBRA

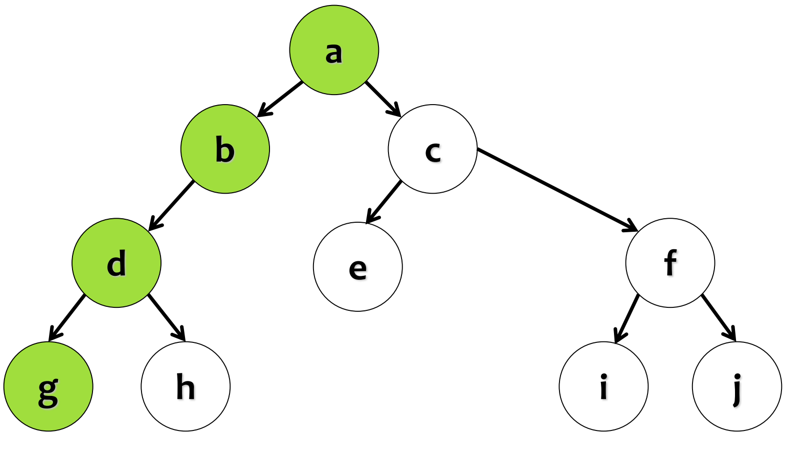
PREORDER example (3/10)

Traversal order: a, b, d



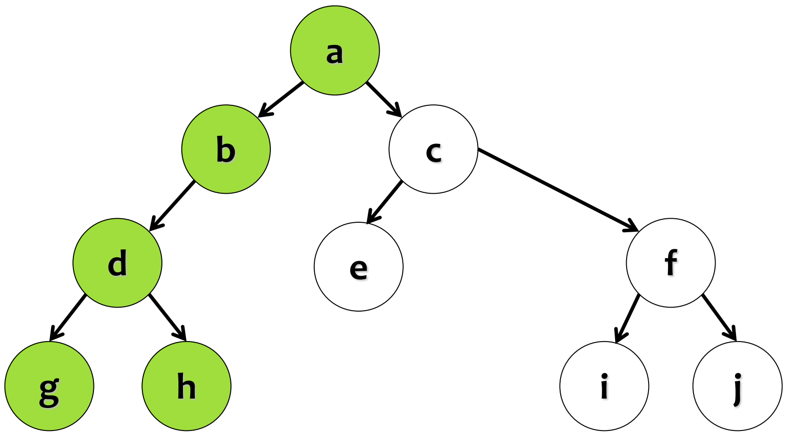
PREORDER example (4/10)

Traversal order: a, b, d, g



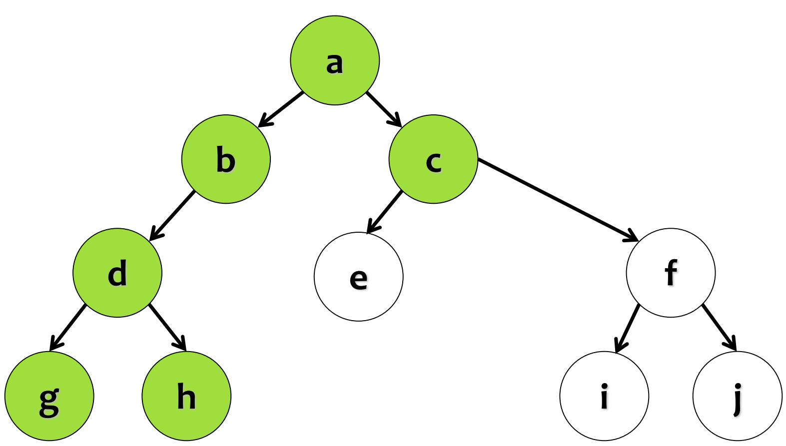
PREORDER example (5/10)

Traversal order: a, b, d, g, h



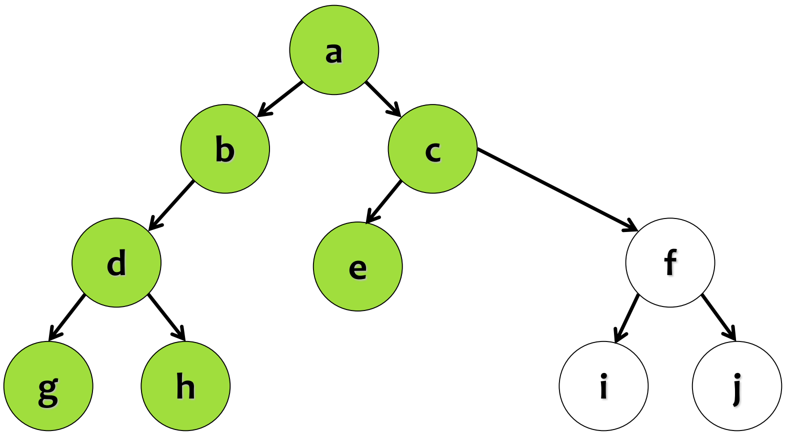
PREORDER example (6/10)

Traversal order: a, b, d, g, h, c



PREORDER example (7/10)

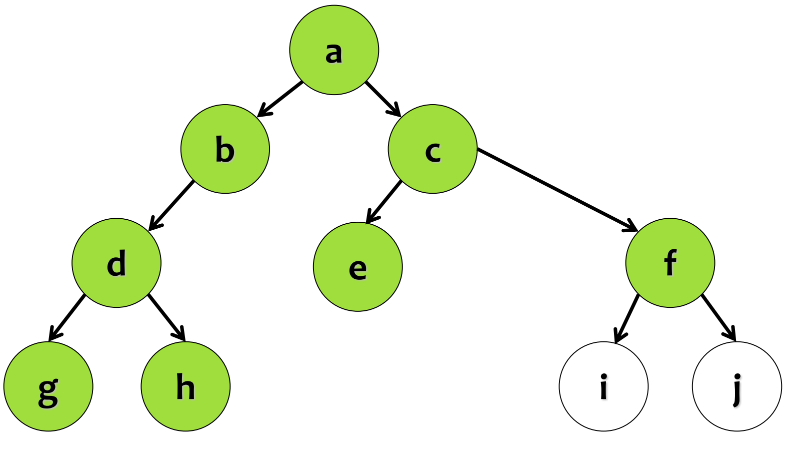
Traversal order: a, b, d, g, h, c, e



ALGEBRA

PREORDER example (8/10)

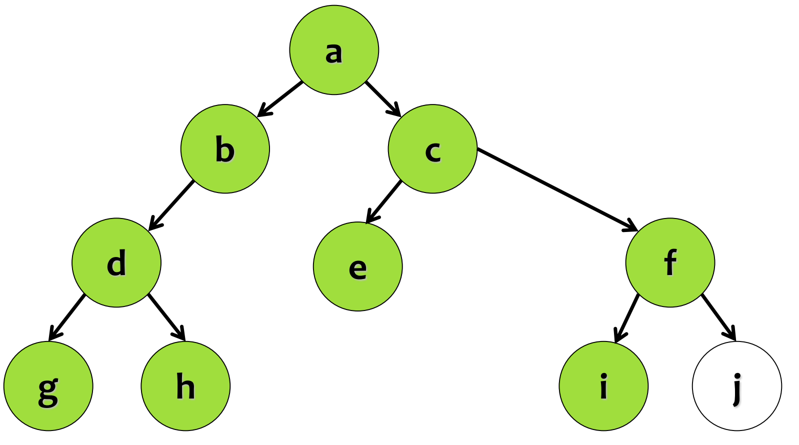
Traversal order: a, b, d, g, h, c, e, f



ALGEBRA

PREORDER example (9/10)

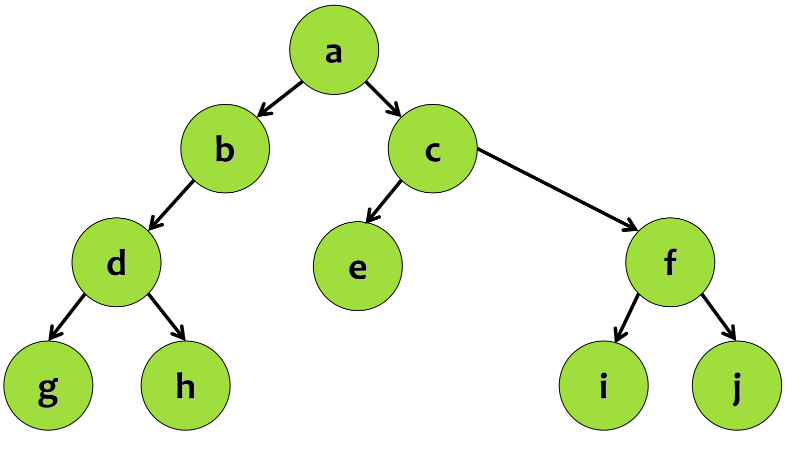
Traversal order: a, b, d, g, h, c, e, f, i



ALGEBRA

PREORDER example (10/10)

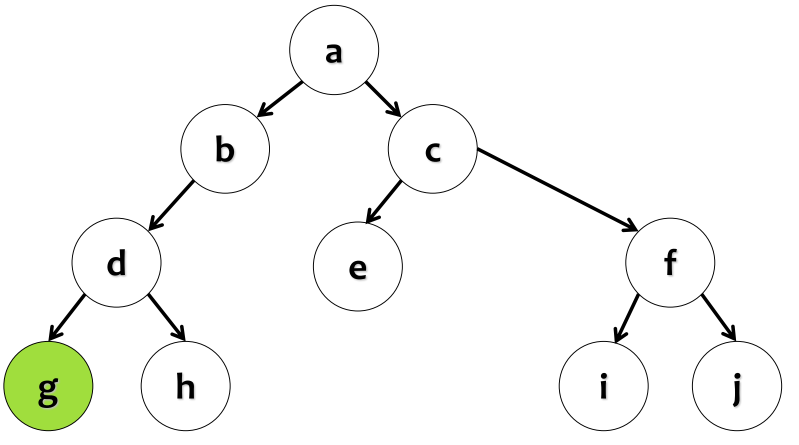
Traversal order: a, b, d, g, h, c, e, f, i, j



ALGEBRA

POSTORDER example (1/10)

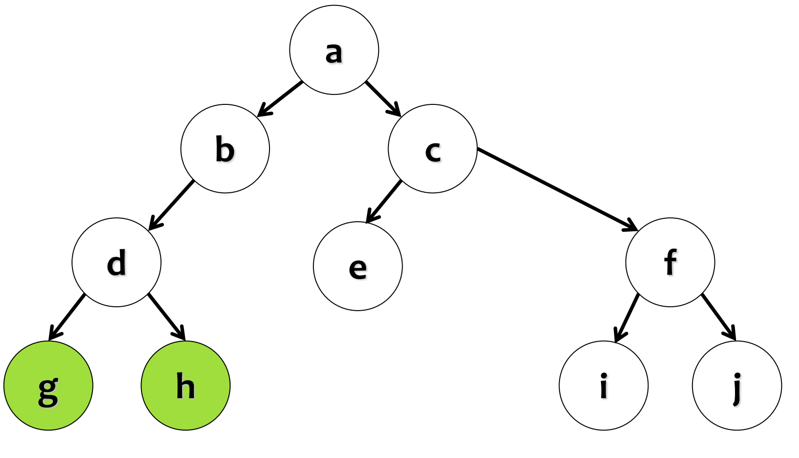
Traversal order: g



ALGEBRA

POSTORDER example (2/10)

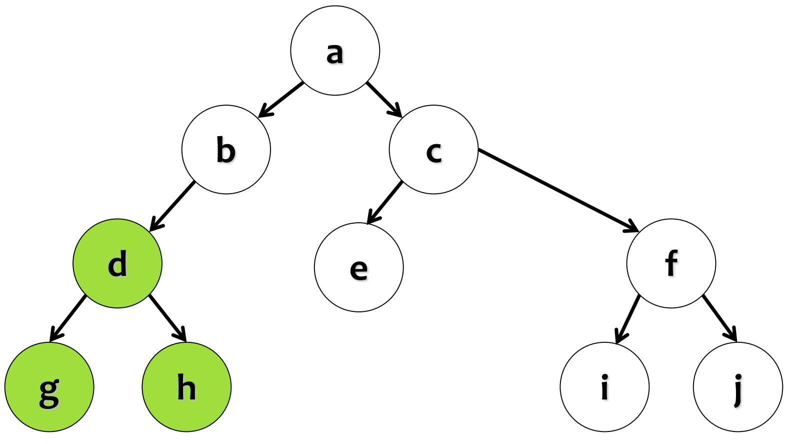
Traversal order: g, h



ALGEBRA

POSTORDER example (3/10)

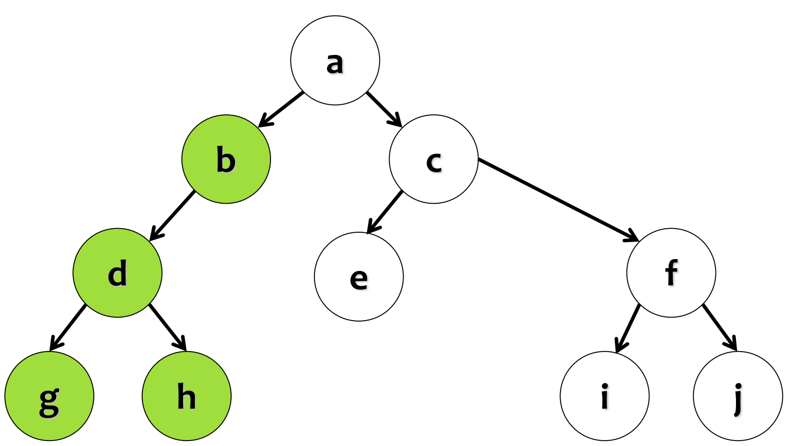
Traversal order: g, h, d



ALGEBRA

POSTORDER example (4/10)

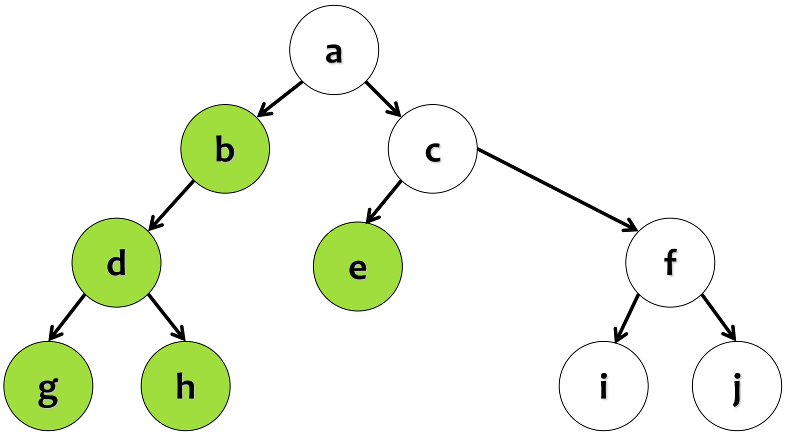
Traversal order: g, h, d, b



ALGEBRA

POSTORDER example (5/10)

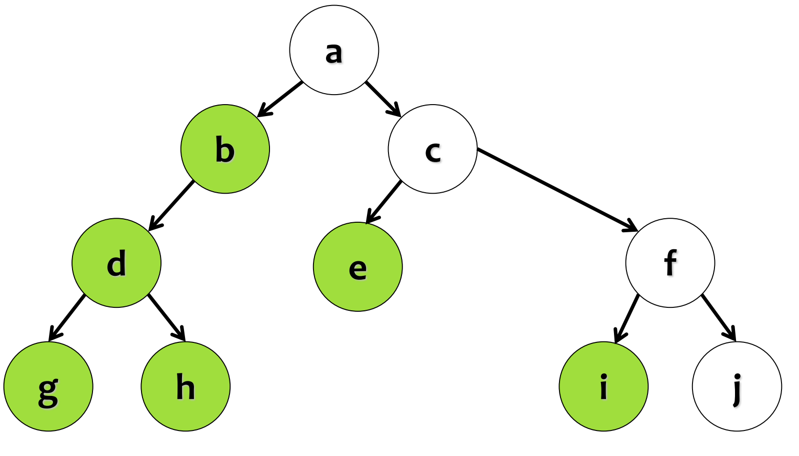
Traversal order: g, h, d, b, e



Strana ▪ 57

POSTORDER example (6/10)

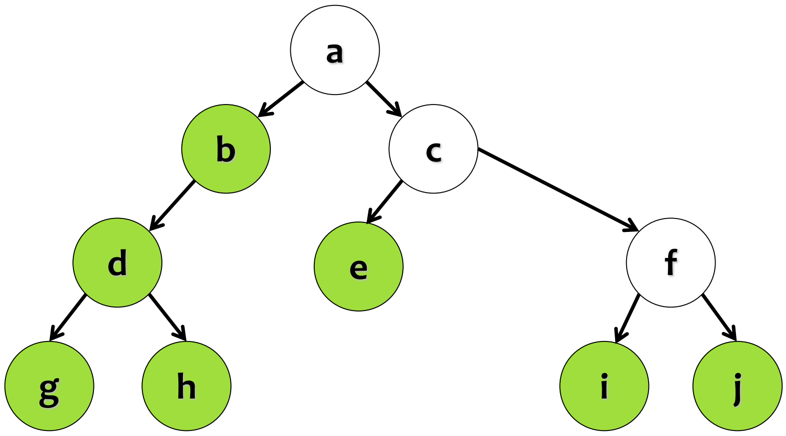
Traversal order: g, h, d, b, e, i



Strana ▪ 58

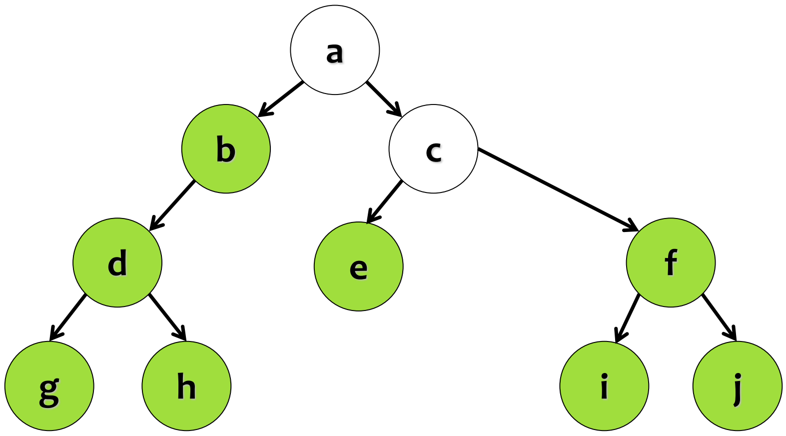
POSTORDER example (7/10)

Traversal order: g, h, d, b, e, i, j



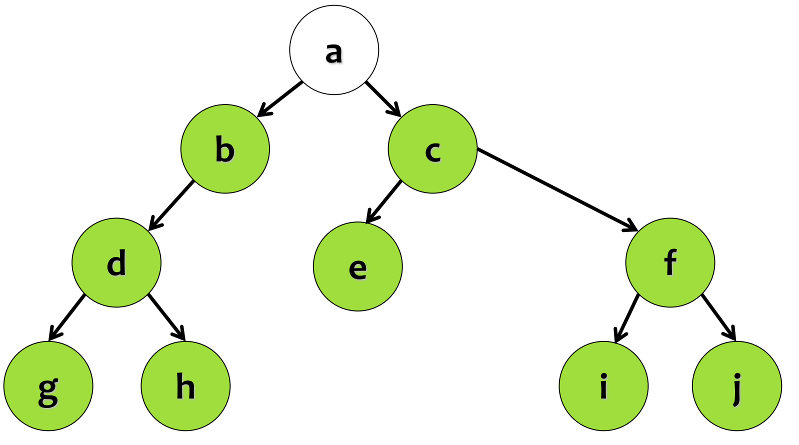
POSTORDER example (8/10)

Traversal order: g, h, d, b, e, i, j, f



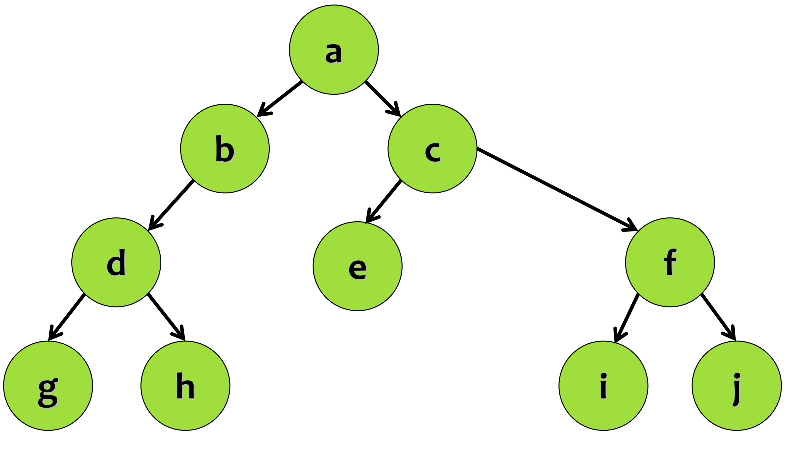
POSTORDER example (9/10)

Traversal order: g, h, d, b, e, i, j, f, c



POSTORDER example (10/10)

Traversal order: g, h, d, b, e, i, j, f, c, a



The order of traversal of all three algorithms

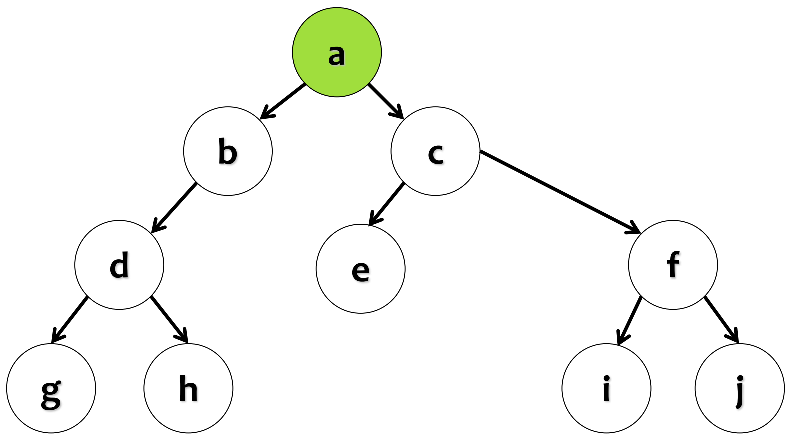
- Order of traversal for INORDER:
  - g, d, h, b, a, e, c, i, f, j
- Order of traversal for PREORDER:
  - a, b, d, g, h, c, e, f, i, j
- Order of traversal for POSTORDER:
  - g, h, d, b, e, i, j, f, c, a

BFS algorithm

- The BFS algorithm visits nodes level by level, from left to right:
  1. Take a queue
  2. Add root to the queue
  3. Take the next element A from the queue and display its value
  4. Add the children of node A to the queue
  5. If the queue is not empty, go to step 3
- For example, if the tree displays a hierarchy, then this traversal method first displays those at the top of the hierarchy
- The usage of this algorithm is not so common in practice

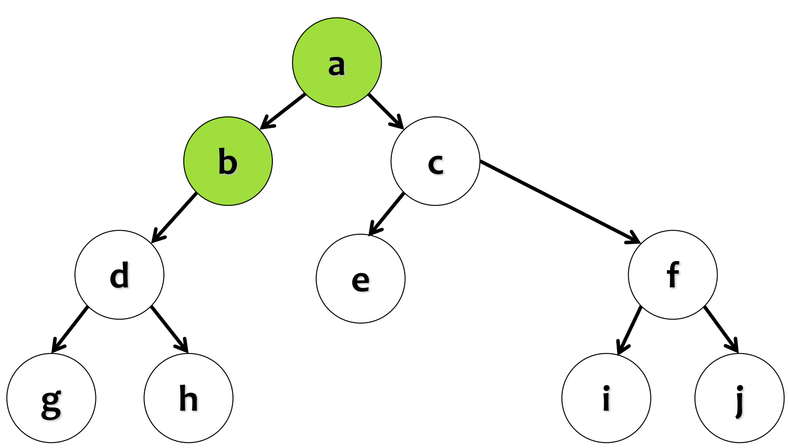
BFS example (1/10)

Traversal order: a



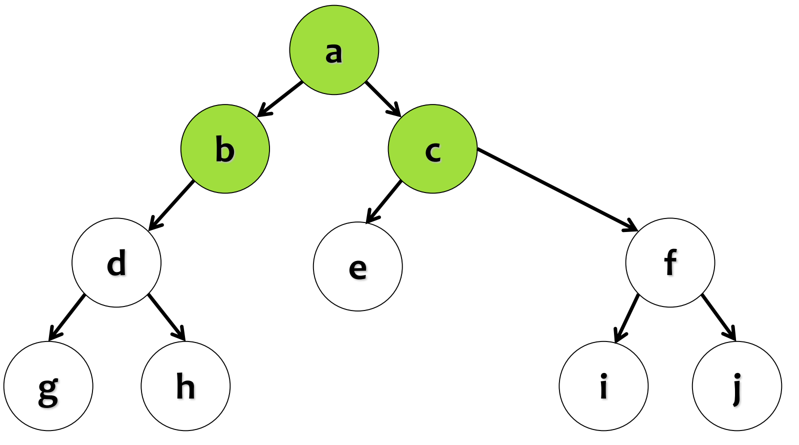
BFS example (2/10)

Traversal order: a, b



BFS example (3/10)

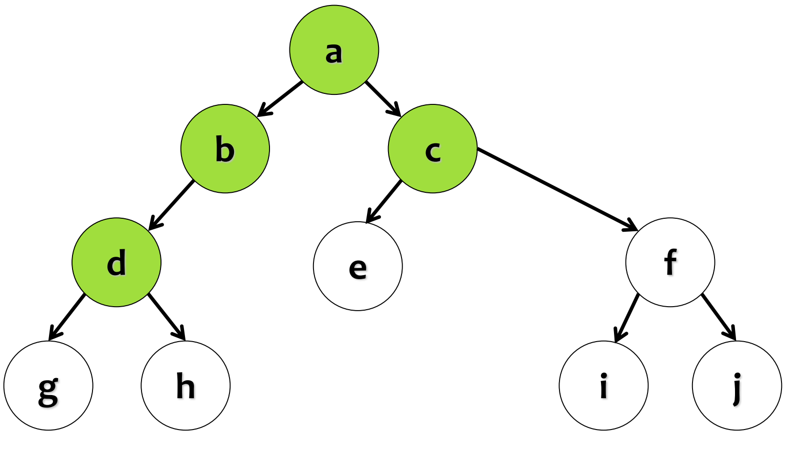
Traversal order: a, b, c



ALGEBRA

BFS example (4/10)

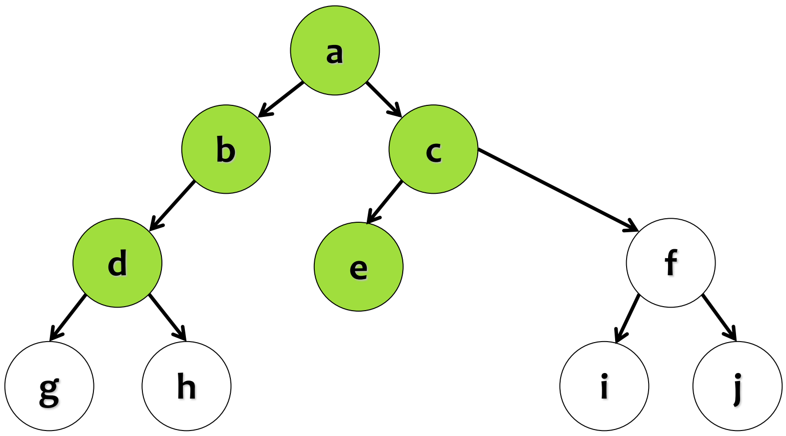
Traversal order: a, b, c, d



ALGEBRA

BFS example (5/10)

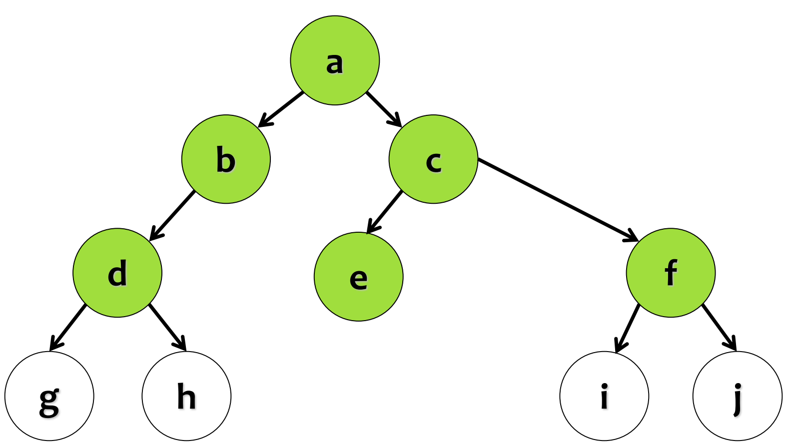
Traversal order: a, b, c, d, e



ALGEBRA

BFS example (6/10)

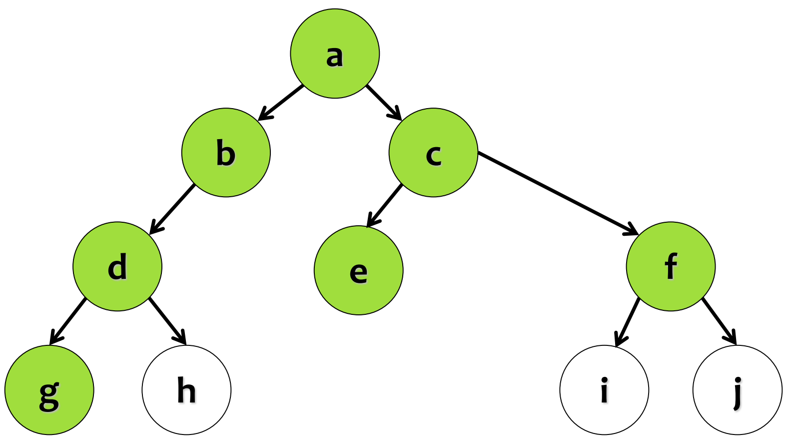
Traversal order: a, b, c, d, e, f



ALGEBRA

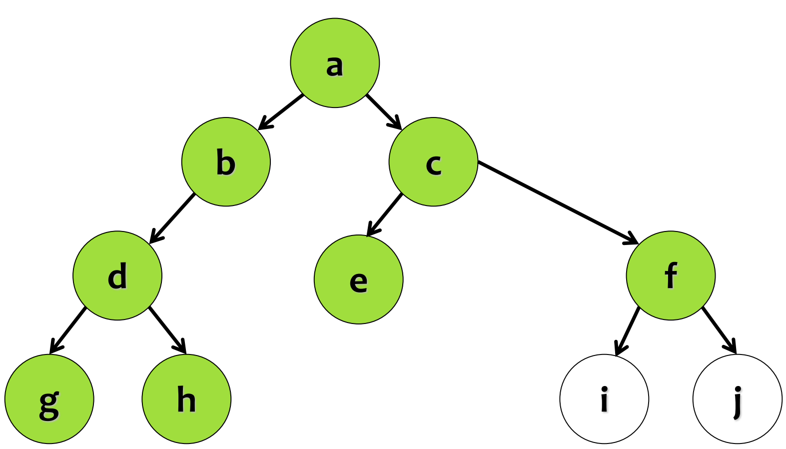
BFS example (7/10)

Traversal order: a, b, c, d, e, f, g



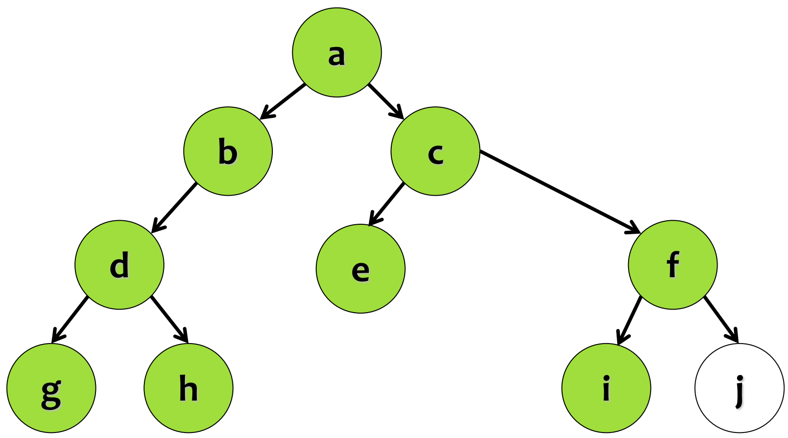
BFS example (8/10)

Traversal order: a, b, c, d, e, f, g, h



BFS example (9/10)

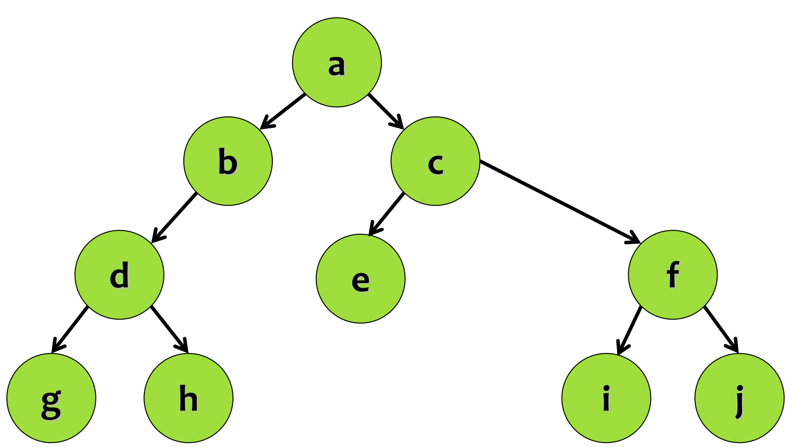
Traversal order: a, b, c, d, e, f, g, h, i



ALGEBRA

BFS example (10/10)

Traversal order: a, b, c, d, e, f, g, h, i, j



ALGEBRA

TREE IMPLEMENTATION

Introduction

- STL does not contain a "pure" tree implementation
  - The tree is used internally as a support for some containers
- Good implementation is available at https://github.com/kpeeters/tree.hh
- We will use our simple implementation
  - Tree traversal methods are missing
  - The destructor is missing

Implementation – btree.h

```
struct node {
    string element;
    node* left_child;
    node* right_child;
};

class btree {
private:
    node* root_node;
    node* create_new_node(string element);
public:
    btree(string element);
    void insert_left(node* parent, string element);
    void insert_right(node* parent, string element);
    node* root();
    node* get_left_child(node* parent);
    node* get_right_child(node* parent);
};
```


Implementation – btree.cpp

```
node* btree::create_new_node(string element) {
      node* novi = new node;
      novi->element = element;
      novi->left_child = nullptr;
      novi->right_child = nullptr;
      return novi;
}

btree::btree(string element) {
      root_node = create_new_node(element);
}

void btree::insert_left(node* parent, string element) {
      parent->left_child = create_new_node(element);
}
```

Implementation – btree.cpp

```
void btree::insert_right(node* parent, string element) {
	parent->right_child = create_new_node(element);
}

node* btree::root() {
	return root_node;
}

node* btree::get_left_child(node* parent) {
	return parent->left_child;
}

node* btree::get_right_child(node* parent) {
	return parent->right_child;
}
```


Usage example

```
btree t("A");

node* node_a = t.root();
t.insert_left(node_a, "B");

node* node_b = t.get_left_child(node_a);
t.insert_left(node_b, "C");

node* node_c = t.get_left_child(node_b);
t.insert_left(node_c, "D");

node* node_d = t.get_left_child(node_c);
t.insert_left(node_d, "E");
```