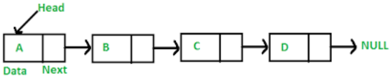
Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

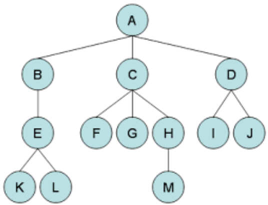
Toward **non-linear** Data Structures

- Elements in a **list** (or in a vector) are totally ordered: predecessor/successor.



- In a **tree** some elements may be uncomparable.

(Informal) Tree = partial ordering + a minimum element (= entry point)

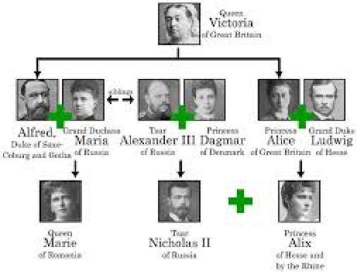


This is achieved by allowing each node to have > 1 successor.

Motivations

- Better representation of unordered/partiall ordered data

Ex: genealogic tree



- Speeding up elementary operations

(insertion/removal/search)

Definitions

- an element in a tree = a **node**
- the predecessor of a node = her **father**
- a successor = a **child**
- a pair (x, y) with x father of y is an **edge**

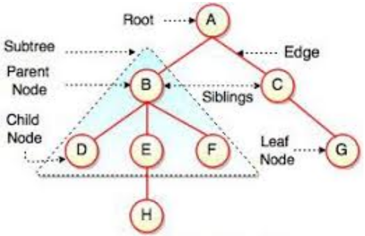


Fig. Structure of Tree

Definitions

- an element in a tree = a **node**
- the predecessor of a node = her **father**
- a successor = a **child**
- a pair (x, y) with x father of y is an **edge**

- x is an **ancestor** of y if it is "before" y: either $x = y$ or $\exists w_1, w_2, \ldots, w_p$ s.t.
  - x is the father of w_1;
  - w_n is the father of y;
  - and $\forall i,\ w_i$ is the father of w_{i+1}.
- y is a **descendent** of x if x is an ancestor of y.

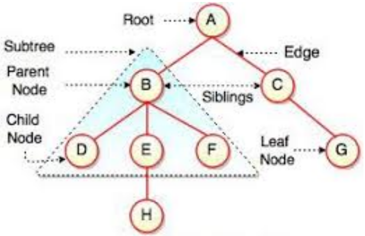


Fig. Structure of Tree

Definitions cont'd

• x and y are **siblings** if they have the same father.

Remark: two siblings are uncomparable

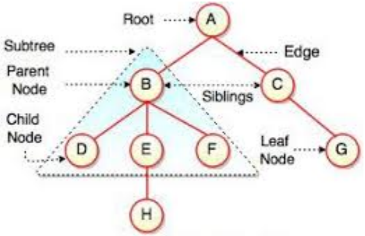


Fig. Structure of Tree

Definitions cont'd

- x and y are **siblings** if they have the same father.

Remark: two siblings are uncomparable

- The **root** is the unique element w/o predecessor (= entry-point)
- A **leaf** is a node w/o successor

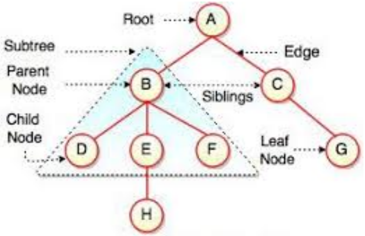


Fig. Structure of Tree

Definitions cont'd

- x and y are **siblings** if they have the same father.

Remark: two siblings are uncomparable

- The **root** is the unique element w/o predecessor (= entry-point)
- A **leaf** is a node w/o successor
- A **subtree** rooted at x = the tree whose nodes are all the descendents of x

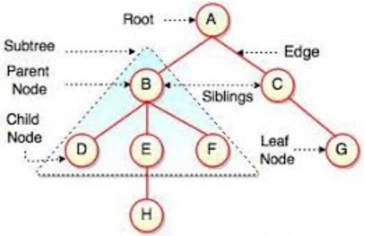


Fig. Structure of Tree

Definitions cont"d

- The **order** of a tree = number of nodes
- The **degree** of a tree = max. # of children for a node

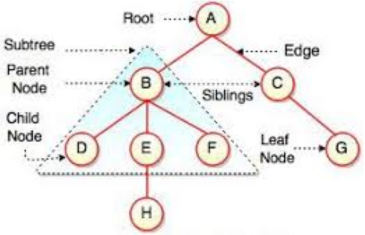


Fig. Structure of Tree

Definitions cont"d

- The **order** of a tree = number of nodes
- The **degree** of a tree = max. # of children for a node
- The **level/depth** of a node = # of ancestors - 1
- The **height** of a tree = max. level of a node

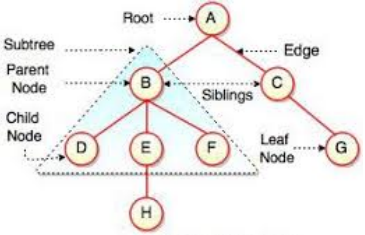


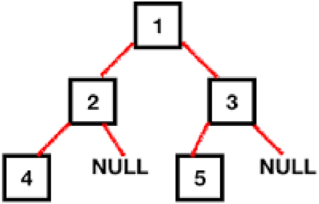
Fig. Structure of Tree

Implementation

Minimal requirements: Access to father node in $\mathcal{O}(1)$; Enumeration of children nodes in $\mathcal{O}(1)$ per child; Access to the root in $\mathcal{O}(1)$.

1) Static tree as two **vectors**:

- `father[i]` $=$ father of i ($i =$ root $\iff$ `father[i] = -1`)
- `child[i]` $=$ first child of i (children must be consecutive).



`father:  [-1,`**0,0**`,1,2]`

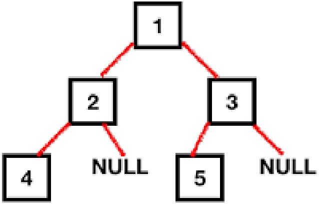
`child:  [`**1**`,3,4,-1,-1]`

Implementation

Minimal requirements: Access to father node in $\mathcal{O}(1)$; Enumeration of children nodes in $\mathcal{O}(1)$ per child; Access to the root in $\mathcal{O}(1)$.

2) Dynamic tree:

```
struct node {
   //information zone
   int value;
   //link zone
   node *father;
   node *child; //first child
   node *next; //next sibling
   node *prev; //previous sibling
}
```



Children of a node are in a (doubly) linked list, whose head is accessed via the `child` pointer.

Special case: **Binary** tree

Definition

k-ary Tree = Degree $\leq k$ Tree (Binary $\iff k = 2$).

```
struct node {
   int value;
   //information zone is slightly different
   node *father;
   node *left;
   node *right;
}
```

Remark: every tree can be encoded as a binary tree

point to the next sibling on the left, and to the first child on the right

Iterators to Trees

A generic problem: enumeration of all elements in a data structure.

- For a list: simple left-to-right scan.
- For a tree?

  $\rightarrow$ Each element should appear before its children

  $\rightarrow$ Two strategies:

  - **Depth-First Search (DFS)**
  - **Breadth-First Search (BFS)**

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

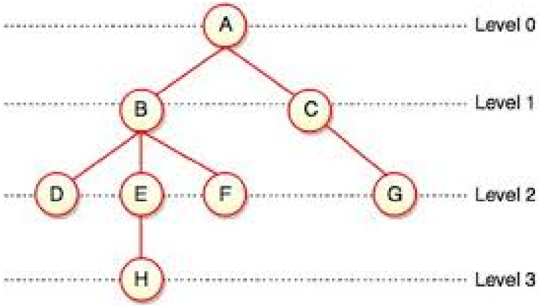


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A

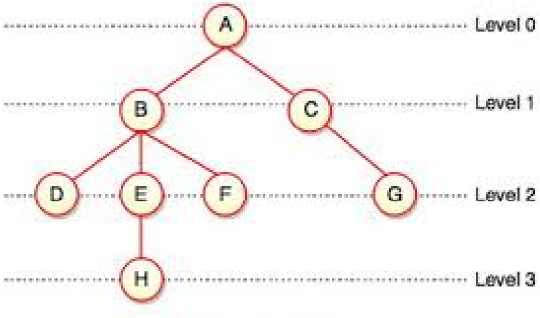


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B

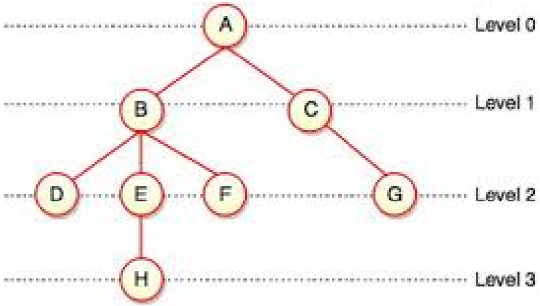


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D

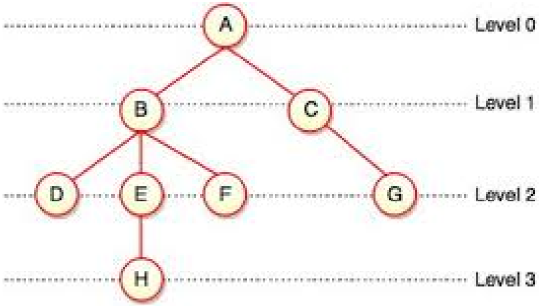


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B

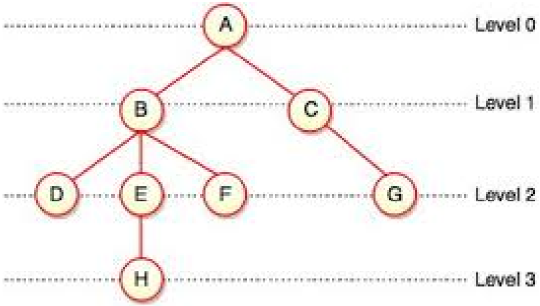


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E

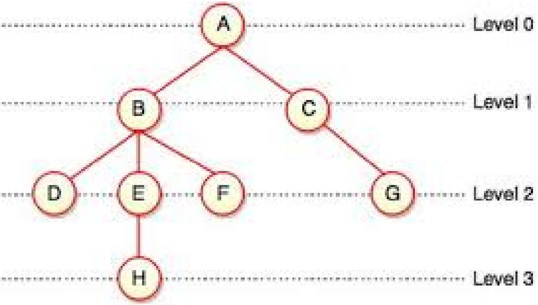


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H

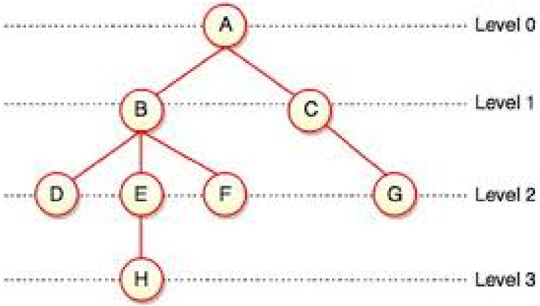


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E

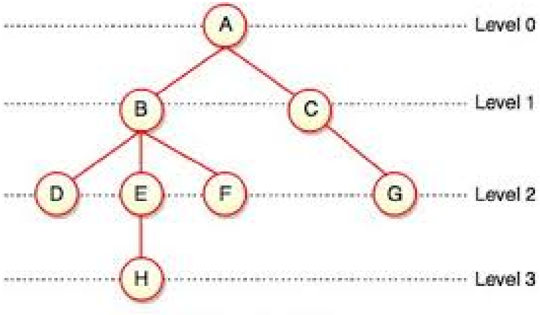


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B

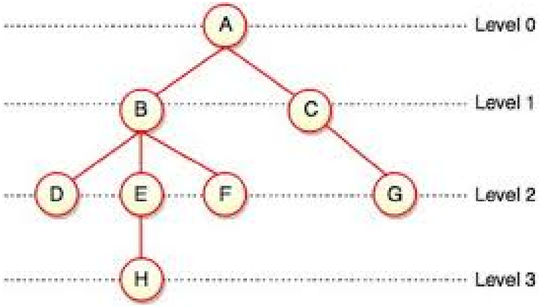


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F

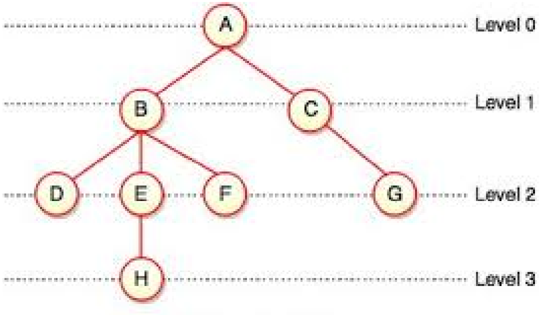


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B

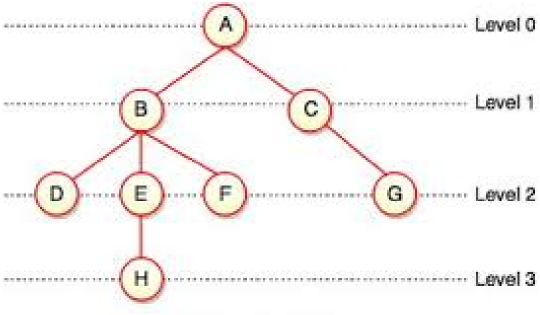


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B
Go Back to A

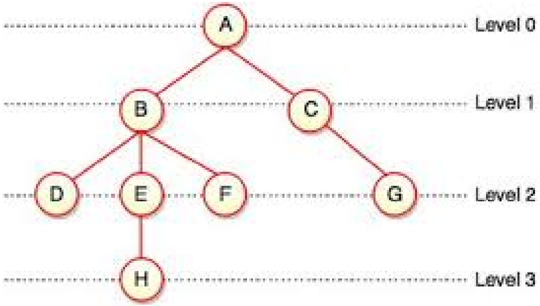


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B
Go Back to A
Visit C

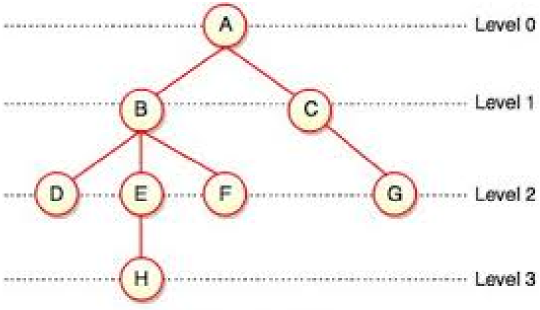


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B
Go Back to A
Visit C
Visit G

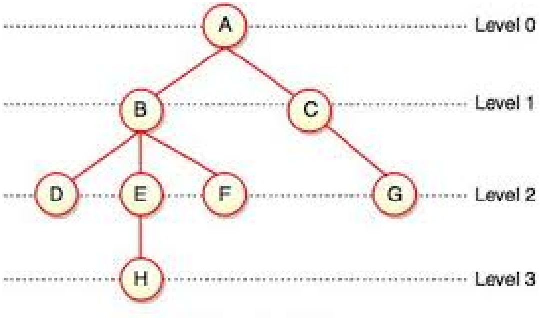


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B
Go Back to A
Visit C
Visit G
Go Back to C

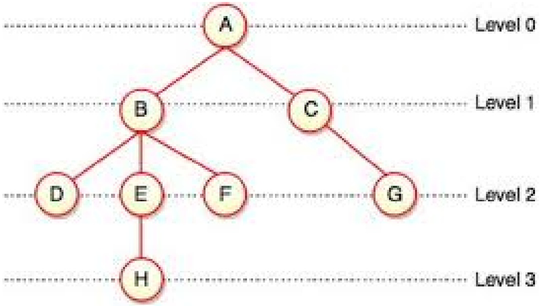


Fig. Levels of Tree

Depth-First Search

Special case of backtracking:

- Start from the root
- When at a node x, go to the first unvisited child (if any) or go back to its father.

Visit A
Visit B
Visit D
Go back to B
Visit E
Visit H
Go Back to E
Go Back to B
Visit F
Go Back to B
Go Back to A
Visit C
Visit G
Go Back to C
Go Back to A

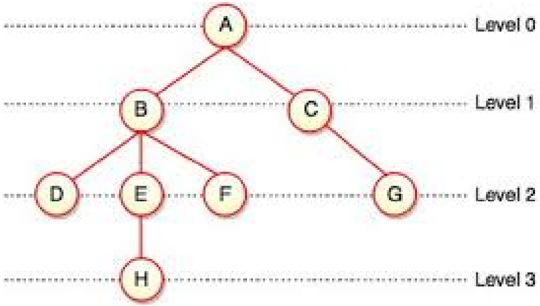


Fig. Levels of Tree

DFS Implementation

- With a **stack**

```
void dfs(node *root) {
  stack<node*> nodes;
  nodes.push(root); //start from the root
  while(!nodes.empty()) {
    node *n = nodes.top(); //current node
    cout << n->value << endl;
    if(n->child != nullptr)
      nodes.push(n->child); //go to the first unvisited child
    else {
      nodes.pop(); //go back to father node
      if(n->next != nullptr)
        nodes.push(n->next); //continue to next sibling
    }
  }
}
```

Complexity: Linear

DFS implementation

- With a **recursive** algorithm

```
void dfs_rec(node *root) {
    cout << root -> value << endl;
    for(node *n = root->child; n != nullptr; n = n->next)
        dfs_rec(n);
}
```

(Simple) Question: What happened to the stack?

Ordering the nodes

Tree Traversal

- **Preorder**: first visit during a DFS

`[A,B,D,E,H,F,C,G]`

- **Postorder**: last visit during a DFS

`[D,H,E,F,B,G,C,A]`

- **Inorder**: first backtracking during a DFS

`[D,B,H,E,F,A,G,C]`

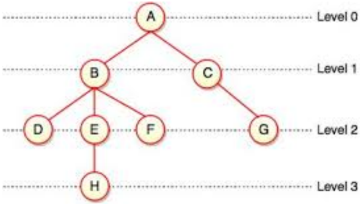


Fig. Levels of Tree

All computable in linear time during a DFS

Traversal Implementation

```
void preorder(node *root){
    cout << root -> value << endl;
    for(node *n = root->child; n != nullptr; n = n->next)
        preorder(n);
}

void postorder(node *root){
    for(node *n = root->child; n != nullptr; n = n->next)
        preorder(n);
    cout << root -> value << endl;
}

void inorder(node *root) {
    if(root->child == nullptr)
        cout << root->value << endl;
    else{
        inorder(root->child);
        cout << root->value << endl;
        for(node *n = root->child->next; n != nullptr; n = n->next)
            inorder(n);
    }
}
```

Binary Tree Traversal

```
void preorder(node *root) {
    cout << root->value << endl;
    if(root->left!=nullptr) preorder(root->left);
    if(root->right!=nullptr) preorder(root->right);
}
```

```
void postorder(node *root) {
    if(root->left!=nullptr) postorder(root->left);
    if(root->right!=nullptr) postorder(root->right);
    cout << root->value << endl;
}
```

```
void inorder(node *root) {
    if(root->left!=nullptr) inorder(root->left);
    cout << root->value << endl;
    if(root->right!=nullptr) inorder(root->right);
}
```

Breadth-First Search

Visit nodes by increasing distance to the root.

Distance = number of nodes to traverse

Distance to the root = **Level**

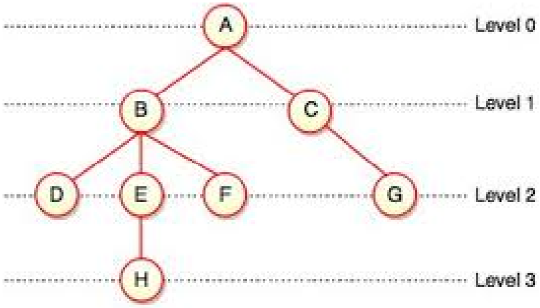


Fig. Levels of Tree

Breadth-First Search

Visit nodes by increasing distance to the root.

Distance = number of nodes to traverse

Distance to the root = **Level**

Visit A

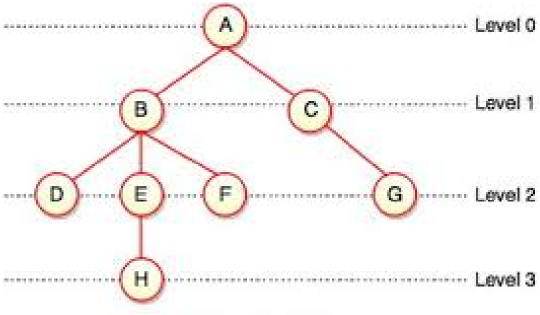


Fig. Levels of Tree

Breadth-First Search

Visit nodes by increasing distance to the root.

Distance = number of nodes to traverse

Distance to the root = **Level**

Visit A
Visit B
Visit C

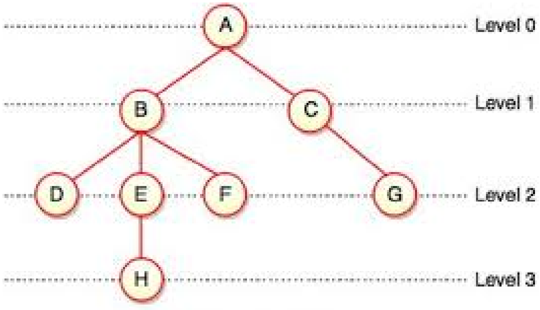


Fig. Levels of Tree

Breadth-First Search

Visit nodes by increasing distance to the root.

Distance = number of nodes to traverse

Distance to the root = **Level**

Visit A
Visit B
Visit C
Visit D
Visit E
Visit F
Visit G

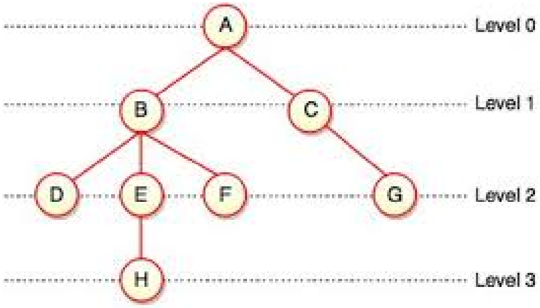


Fig. Levels of Tree

Breadth-First Search

Visit nodes by increasing distance to the root.

Distance = number of nodes to traverse

Distance to the root = **Level**

Visit A
Visit B
Visit C
Visit D
Visit E
Visit F
Visit G
Visit H

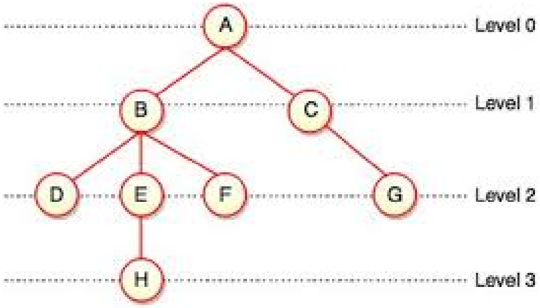


Fig. Levels of Tree

BFS Implementation

With a **queue**.

```
void bfs(node *root) {
   queue<node*> nodes;
   nodes.push(root);
   while(!nodes.empty()){
      node *n = nodes.front(); nodes.pop();
      cout << n->value << endl; //visit node
      for(node *c = n->child; c != nullptr; c = c->next)
         nodes.push(c);
   }
}
```

Complexity: Linear

Basic operations

- **Output the root?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

Basic operations

- **Output the root?**

$\implies$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\implies$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

Basic operations

- **Output the root?**

$\Longrightarrow$ Direct access from the pointer to root.

Complexity? $\mathcal{O}(1)$

- **Output the leaves?**

$\Longrightarrow$ find all nodes n s.t. `n->child` is empty

```
void leaves(node *root) {
    if(root->child == nullptr) cout << root->value << endl;
    else for(node *n = root->child; n != nullptr; n = n->next)
        leaves(n);
}
```

Complexity? $\mathcal{O}(n)$

- **Compute the degree?**

$\Longrightarrow$ Enumeration of all d children in $\mathcal{O}(d)$

$\Longrightarrow$ In $\mathcal{O}(1)$ (same trick as `size()` for lists)

Requires an additional field **int** `degree;`

Computing the order

In $\mathcal{O}(n)$ time by **Tree Traversal**.

$\Longrightarrow$ We can compute the order of *all rooted subtrees* (special case of Dynamic Programming)

```
//addition of a new field int order; to the structure
void compute_orders(node *root){
   root->order = 1;
   for(node *n = root->child; n != nullptr; n = n->next) {
      compute_orders(n);
      root->order += n->order;
   }
}
```

Complexity: Linear

Computing the levels

Observation: if n is not the root, then

`n->level = n->father->level + 1`.

Method #1: using a **preordering**.

```
void compute_levels(node *n) {
   if(n->father == nullptr)
      n->level = 0; //root
   else n->level = n->father->level +1;
   for(node *c = n->child; c != nullptr; c = c->next)
      compute_levels(c);
}
```

Complexity: Linear

Computing the levels

Method #2: using a **BFS**!

```
void compute_levels(node *root) {
   queue<node*> nodes;
   nodes.push(root);
   root->level = 0;
   while(!nodes.empty()){
      node *n = nodes.front(); nodes.pop();
      for(node *c = n->child; c != nullptr; c = c->next){
         nodes.push(c);
         c->level = n->level + 1;
      }
   }
}
```

Complexity: Linear

Computing the heights

- For a leaf node n: `n->height = 0`
- For a node n with children $c_1, c_2, \ldots, c_d$:

$$\texttt{n->height = 1+ } \max_i \texttt{ c\_i->height}$$

```
void compute_heights(node *root) {
   root->height = 0;
   for(node *n = root->child; n != nullptr; n = n->next) {
      compute_heights(n);
      if(n->height >= root->height)
         root->height = 1 + n->height;
   }
}
```

Complexity: Linear

Questions