Data structures and Algorithms
Trees

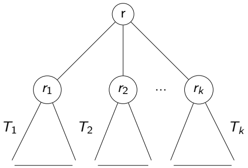
Pham Quang Dung

Hanoi, 2012

Outline

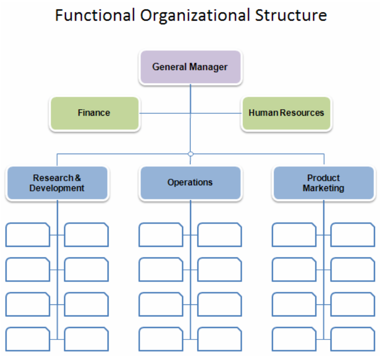
Definitions

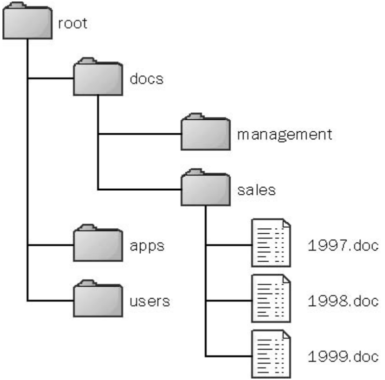
- Abstract model of hierarchical structures
- A tree consists of nodes with a parent-child relation with a special node called root of the tree
  - Basic step: A node r is a tree with root r
  - Recursive step: Suppose $T_1, \dots, T_k$ are trees with root $r_1, \dots, r_k$, and a node r.
    - Construct a tree by making r as the parent of $r_1, \dots, r_k$
    - New constructed tree has r as root
    - $r_1, \dots, r_k$ are children of r



Applications

- Organization charts
- File systems





source: http://www.vertex42.com/ExcelTemplates/organizational-chart.html

source: http://www.ibiblio.org/gdunc/netone/ms_netency/netencyhtml/c0F613788.htm

Tree terminology

- Path: a sequence of nodes $x_1, \ldots, x_k$ where x_i is parent of x_{i+1} $(\forall 1 \leq i < k)$ is a path from node x_1 to node x_k with length $k-1$
- Ancestor: node u is ancestor of node v if there exists a path from u to v
- Descendant: node u is descendant of node v if v is ancestor of u
- Sibling: node u and v are sibling if they have the same parent
- Leaf: node has no children
- Height of a node v is the length of the longest path from v to a leaf on the tree plus 1
- Depth of a node v is the length of the unique path from root to v plus 1
- Internal nodes are nodes having children
- Label of a node: information stored in the node for further computation, e.g., numerical value, record, etc.

Outline

1. Definitions
2. Tree ADT
3. Preorder, inorder, and postorder traversals
4. Data structures for trees
5. Binary Trees

Tree ADT

- parent(n, T)
- leftmostChild(n, T)
- rightSibling(n, T)
- create($x, T_1, \ldots, T_k$): create a new node with label x, make roots of trees $T_1, \ldots, T_k$ as children of x from left to right. Return the tree with root r
- root(T)
- makeNull(T)

Outline

Preorder traversal

Suppose a tree T has subtrees $T_1, \ldots, T_k$ from left to right, preorder of node of T is defined as follows:

- Root of T
- Nodes of T_1 in preorder
- Nodes of T_2 in preorder
- ...
- Nodes of T_k in preorder

Algorithm 1: preorder(T)

```
visit(r);
foreach i ∈ {1, ..., k} do
    preorder(T_i)
```

Inorder traversal

Suppose a tree T has subtrees $T_1, \ldots, T_k$ from left to right, inorder of node of T is defined as follows:

- Nodes of T_1 in inorder
- Root of T
- Nodes of T_2 in inorder
- ...
- Nodes of T_k in inorder

Inorder traversal

Algorithm 2: inorder(T)

```
inorder(T_1);
visit(r);
foreach i ∈ {2, . . . , k} do
    inorder(T_i)
```

Postorder traversal

Suppose a tree T has subtrees $T_1, \ldots, T_k$ from left to right, postorder of node of T is defined as follows:

- Nodes of T_1 in postorder
- Nodes of T_2 in postorder
- ...
- Nodes of T_k in postorder
- Root of T

Postorder traversal

Algorithm 3: postorder(T)

```
foreach i ∈ {1, ..., k} do
    postorder(T_i)
visit(r);
```

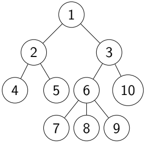
Outline

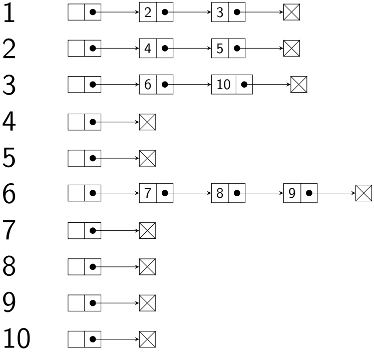
Data structures for trees - Array

- Suppose $\{1, \ldots, n\}$ are nodes of a tree T
- $A[i] = j$ if node j is parent of node i
- $A[i] = 0$ if i is the root of T
- Drawback
  - ineffective for manipulating with children, height, depth
  - do not present the order of children

Data structures for trees - List of children

- Each node is associated with a linked list of its children (with order)





Data structures for trees: leftmost-child and right-sibling representation

```
struct Node{
  int value; // store information
  Node* leftMostCchild; // pointer to the leftmost child of
      the node
  Node* rightSibling; // pointer to the right sibling of the
      node
};

Node* root; // root of the tree
```

Listing 1: structure of a node

Find a node of the given tree

```
Node* find(Node* r, int v){
  if(r == NULL) return NULL;
  if(r->id == v) return r;
  Node* p = r->leftMostChild;
  while(p != NULL){
    Node* pv = find(p,v);
    if(pv != NULL) return pv;
    p = p->rightSibling;
  }
  return NULL;
}
```

Add a node to the end of the list of children of a given node

```
void addChild(Node* p, int v){
  // create new node pv with id = v
  // make pv as a child of p
  Node* pv = new Node;
  pv->id = v;
  pv->rightSibling = NULL;
  pv->leftMostChild = NULL;
  Node* pi = p->leftMostChild;// head of the children list
  if(pi == NULL){
    p->leftMostChild = pv;
  }else{
    while(pi->rightSibling != NULL){
      pi = pi->rightSibling;
    }
    pi->rightSibling = pv;
  }
}
```

pre-order traversal

```
void preorder(Node* r){
  if(r == NULL) return;
  printf("%d, ",r->id);
  Node* p = r->leftMostChild;
  while(p != NULL){
    preorder(p);
    p = p->rightSibling;
  }
}
```

count the number of nodes of the tree

```
int count(Node* r){
  if(r == NULL) return 0;
  int c = 1;
  Node* p = r->leftMostChild;
  while(p != NULL){
    int cp = count(p);
    c = c + cp;
    p = p->rightSibling;
  }
  return c;
}
```

Compute the height of a node of the tree

```
int height(Node* p){
  if(p == NULL) return 0;
  int h = 0;
  Node* pi = p->leftMostChild;
  while(pi != NULL){
    int hi = height(pi);
    h = h > hi ? h : hi;
    pi = pi->rightSibling;
  }
  return h + 1;
}
```

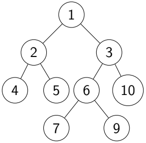
Find the parent of a node of the tree

```
Node* parent(Node* r, Node* p){
  if(r == NULL) return NULL;
  Node* q = r->leftMostChild;
  while(q != NULL){
    if(q == p) return r;
    Node* h = parent(q,p);
    if(h != NULL) return h;
    q = q->rightSibling;
  }
  return NULL;
}
```

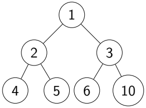
Outline

Binary trees

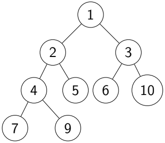
- Each node has at most two children
- Separation of left child and right child



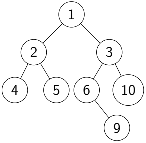
full binary tree



perfect binary tree



complete binary tree



balanced binary tree

Data structures for binary trees

```
struct Node{
int value; // information
Node* leftChild; // pointer to the left child
Node* rightChild; // pointer to the right child
};

Node* root; // root of the tree
```

Listing 2: data structure of a node of a binary tree