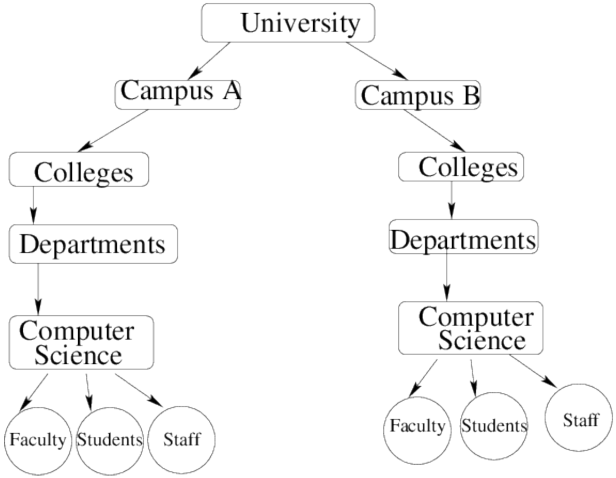
Trees

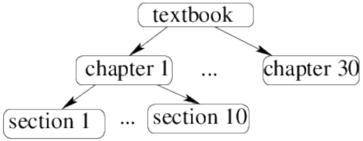
Trees are used to represent a hierarchical relation between data.

- Structure of an organization

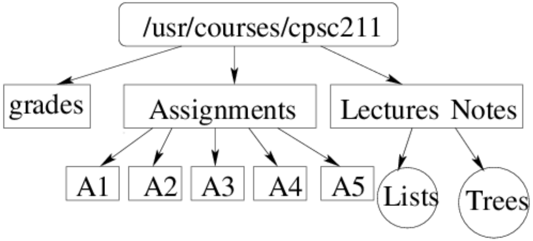


Trees – Examples

- Table of contents of a book



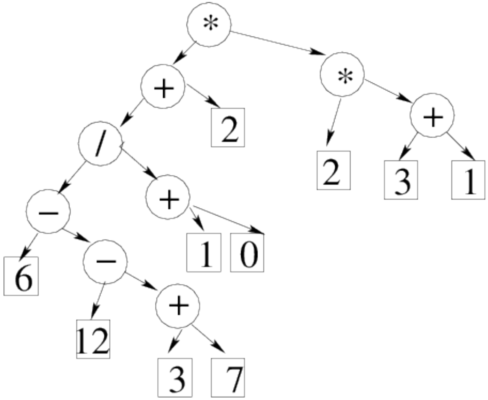
- Most operating systems use tree structures to organize their file systems. Here is an example of a Linux/UNIX file system tree



Data Organization

- Expression tree for the arithmetic expression

$$((6-(12-(3+7)))/(1+0)+2)*(2*(3+1))$$



Constructing an Expression Tree

- The following algebraic expression:

  $$a/b*(c+(d-e))$$

  was converted to its postfix form

  $$ab/cde-+*$$

  by using an algorithm based on queue and stack.
- A binary tree is a convenient data structure to store an algebraic expression. The following algorithm on the next slide is used to generate such a tree from a given postfix form.

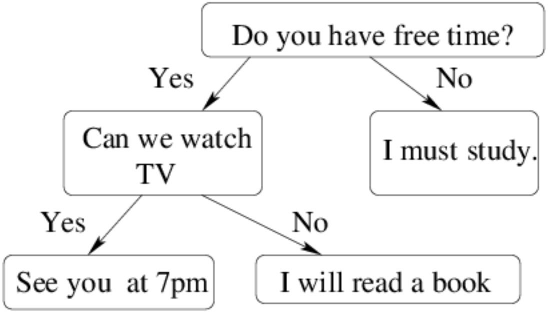
Constructing an Expression Tree (cont)

Algorithm

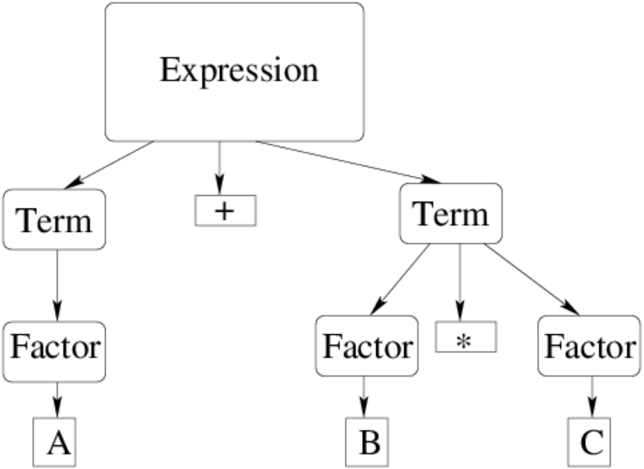
- Read postfix form of the algebraic expression token by token and initialize a new node with the token.
- Test the token:

  If the token is an operand, put it on a stack.
  If the token is an operator, pop from the stack two values and make the first one the right child and the second one the left child of the operator. Push the operator on the stack.
- Continue two previous steps until no token left in the postfix form.

Decision Tree



Parse Tree for the Algebraic Expression $A + B * C$



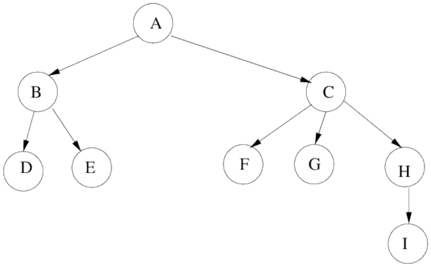
Tree Terminology

- A tree is a set of nodes and directed edges (or branches) connecting pairs of nodes.
- *Size of a tree* is the total number of nodes of this tree.
- If there is a path from the node u to the node v then u is an *ancestor* of v and v is a *descendant* of u.
- *Size of a node* or *degree of a node* is the number of descendants.
- *Root* is the distinguished node—it is the ancestor of all nodes in tree.
- *Siblings* are nodes with the same parent.
- Nodes that have no children are called *leaves* or *external nodes*. All other nodes are called *internal nodes*.

General Rooted Tree

- *Depth* of a node in a tree is the length of the path from the root to the node.
- *Height* of a node is the length of the longest path from a given node to the deepest leaf. The height of a tree is equal to the height of the root.
- A tree with n nodes must have $n-1$ edges.
- *A general rooted tree* is a set of nodes that has a designated node called the *root*, from which zero or more subtrees descend.
- Each subtree itself satisfies the definition of a tree.
- Every node (except the root) is connected by an edge from exactly one other node.
- There is a unique path from the root to each node.
- An empty tree has no nodes.

General Rooted Tree (cont)

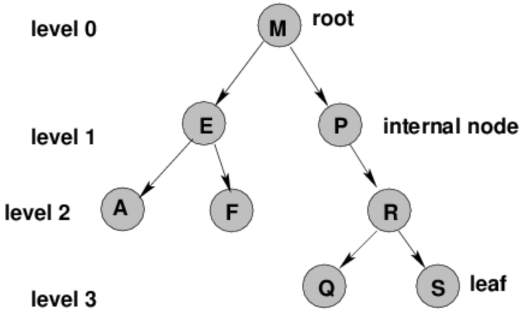


A is a root node. B is a parent of D and E. C is the sibling of B. D and E are children of B.
D, E, F, G, I are external nodes, or leaves. A, B, C, H are internal nodes.
The depth (level) of E is 2. The height of the tree is 3. The degree of node B is 2.
Tree property: The number of edges is equal to the number of nodes minus one.

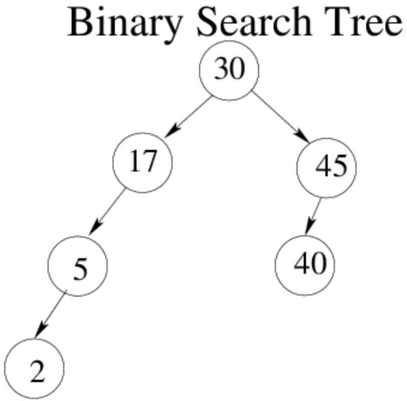
Node	Height	Depth
A	3	0
B	1	1
C	2	1
D	0	2
E	0	2
F	0	2
G	0	2
H	1	2
I	0	3

Binary Trees

- *A binary tree* is a tree in which each node has exactly two subtrees: the left subtree and right subtree, either or both of which may be empty.
- The recursive definition of the binary tree: it is either empty or consists of a root, a left tree, and a right tree.

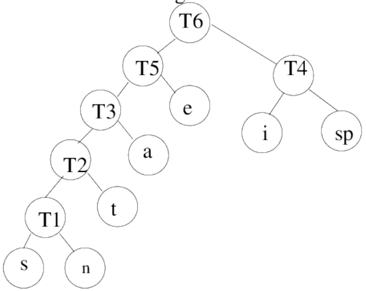


Examples of Binary Trees



File-Compression: Huffman Coding Tree

Huffman's algorithm after the final merge



- Characters are coded – ASCII characters are represented using 8 bits for each character.
- This is a fixed code length (= 8 bits) representation.
- The general strategy of the Huffman's algorithm is to allow the character code length to vary from character to character and to ensure that frequently occurring characters have *short* codes.

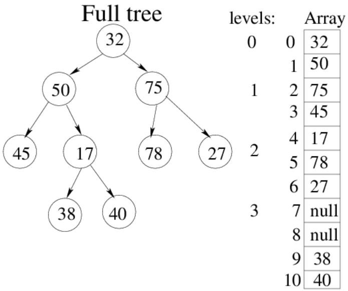
A Proper/Full Binary Tree

Definition

The set of proper/full binary trees can be defined recursively.

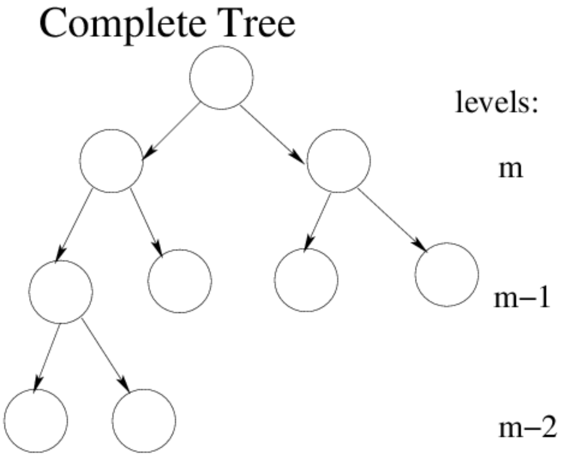
A single node is a full binary tree.

If T_1 and T_2 are full binary trees, there is a full binary tree T consisting of a root r together with edges connecting the root r to each of the roots of left subtree T_1 and right subtree T_2.

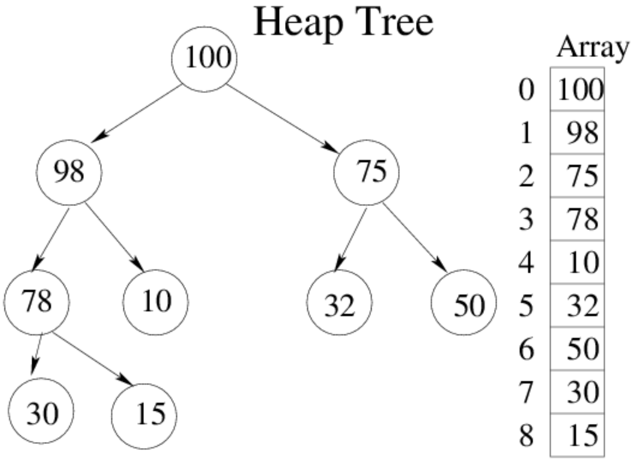


levels:		Array
0	0	32
	1	50
1	2	75
	3	45
2	4	17
	5	78
	6	27
3	7	null
	8	null
	9	38
	10	40

Dense or Complete Binary Tree



Heap Tree


Heap Tree
100
98
75
78
10
32
50
30
15
Array
0 100
1 98
2 75
3 78
4 10
5 32
6 50
7 30
8 15

Properties of Proper Binary Trees

- Notation used:

n_e	= number of external nodes (leaves)
n_i	= number of internal nodes
n	= number of nodes
h	= height of a tree

- The number of nodes at level k is at most 2^k $(k = 0, 1, \ldots)$
- Main properties:
  - $n_e = n_i + 1$
  - $n_e \leq 2^h$ which is equivalent to $h \geq \log_2(n_e)$
  - $h \geq \log_2(n+1) - 1$ since $n_e = (n+1)/2$
  - $h \leq n_i \leq (n-1)/2$ since $n_i = (n-1)/2$

`BinaryNode` Type

- First we describe a node in a binary tree. This is done in the struct (or class) `BinaryNode`.
  - A node in this representation has two `BinaryNode` members, one for the left child and one for the right child and data of object type.
  - When a node has no children, the corresponding members are `nullptr`.
- Because trees can be defined recursively, many tree functions are easily implemented by using recursion and the recursive functions are compact comparing with their non-recursive implementation.

A Size and Height of a Binary Tree

- To find the size of a tree we use the formula:

  $$S_T = 1 + S_L + S_R,$$

  where S_L is the size of the left tree, and S_R is the size of the right tree.
- To find the height of a tree we use the formula:

  $$H_T = 1 + \max(H_L, H_R)$$

  where H_L is the height of the left tree, and H_R is the height of the right tree.

BinaryNode Type (cont)

```
struct BinaryNode {
   int element;
   BinaryNode *left, *right;
   //constructor
   BinaryNode(int el=0, BinaryNode *lt=nullptr,
           BinaryNode *rt=nullptr) :
           element(el), left(lt), right(rt) { }
   // member functions
   int size(BinaryNode *t);
   int height(BinaryNode *t);
   BinaryNode *copy();
};
```

Implementation of `BinaryNode`

```
int BinaryNode::size(BinaryNode *t) { //recursive
   if (t == nullptr)
      return 0;
   else
      return 1 + size(t->left) + size(t->right);
}
int BinaryNode::height(BinaryNode *t) { //recursive
   if (t == nullptr)
      return -1;
   else {
      int hlf = height(t->left);
      int hrt = height(t->right);
      return (hlf > hrt) ? 1+hlf : 1+hrt;
   }
}
```

Implementation of `BinaryNode` (cont)

```
BinaryNode *BinaryNode::copy( ) { // recursive
   BinaryNode *root = new BinaryNode(element);
   if (left != nullptr)
      root->left = left->copy();
   if (right != nullptr)
      root->right = right->copy();
   return root;
}
```

The Class BinaryTree

```
#include "BinaryNode.h"
class BinaryTree {
 private:
   BinaryNode *root;
   void deleteTree(BinaryNode *root);
 public: // class BinaryTree (cont)
   BinaryTree( ) { root = nullptr; }
   BinaryTree(int el) { root = new BinaryNode(el); }
   ~BinaryTree() { deleteTree(root); root = nullptr; }
```

The Class BinaryTree (cont)

```
   // functions
   BinaryNode *getRoot( ) { return root; }
   bool isEmpty( ) { return root == nullptr; }
   int size( ) { return (root == nullptr) ?
                              0 : root->size(root); }
   int height( ) {  return (root == nullptr) ?
                              -1 : root->height(root); }
   void copy(BinaryTree& rhs) {
      if (this != &rhs) {
         deleteTree(root); // make tree empty
         if (rhs.root != nullptr)
            root = rhs.root->copy();
      }
   }
   void merge(int rootItem, BinaryTree& t1,
                 BinaryTree& t2);
};
```

Binary Tree Functions

```
// merge t1, t2 and root (with rootItem)
void BinaryTree::merge(int rootItem, BinaryTree& t1,
                       BinaryTree& t2)
{
   if (t1.root == t2.root && t1.root != nullptr) {
      cerr << "Left tree == right tree; "
           << " merge aborted" << endl;
      return;
   }
   if (this != &t1 && this != &t2) deleteTree(root);
   root = new BinaryNode(rootItem, t1.root,
                         t2.root);
   //remove aliases
   if (this != &t1) t1.root = nullptr;
   if (this != &t2) t2.root = nullptr;
}
```

Binary Tree Functions (cont)

```
// delete a tree rooted at "root"
void BinaryTree::deleteTree(BinaryNode *root)
{  // postorder traversal
   if (root == nullptr) return; // nothing to delete
   if (root->left != nullptr)
      deleteTree(root->left);
   if (root->right != nullptr)
      deleteTree(root->right);
   delete root;
}
```

Binary Tree Traversals

- A tree traversal requires an algorithm that visits (processes) each node of a tree exactly once.
- By processing we mean whatever operation to be performed on the element at a given node (examples: to print it, to update it).
- To implement traversal functions we need the class `TreeOperation`, which contains only one pure virtual function: `processNode()`.
  - This function must be defined in a subclass of the class `TreeOperation` before we use the traversal functions.

```
class TreeOperation {
 public:
   virtual void processNode(int elem) = 0;
};
```

Preorder Traversal (recursive)

Process the root node.

Recursively visit all nodes in the left subtree.

Recursively visit all nodes in the right subtree.

```
//this is a continuation of the struct BinaryNode
//TreeOperation is a class
//with one function: processNode()
void BinaryNode::preOrderTraversal(TreeOperation& op)
{
   op.processNode(element);
   if (left != nullptr) left->preOrderTraversal(op);
   if (right != nullptr) right->preOrderTraversal(op);
}
```

Inorder Traversal (recursive)

Recursively visit all nodes in the left subtree.

Process the root node.

Recursively visit all nodes in the right subtree.

```
void BinaryNode::inOrderTraversal(TreeOperation& op)
{
   if (left != nullptr) left->inOrderTraversal(op);
   op.processNode(element);
   if (right != nullptr) right->inOrderTraversal(op);
}
```

Postorder Traversal (recursive)

Recursively visit all nodes in the left subtree.

Recursively visit all nodes in the right subtree.

Process the root node.

```
void BinaryNode::postOrderTraversal(TreeOperation& op)
{
   if (left != nullptr) left->postOrderTraversal(op);
   if (right != nullptr) right->postOrderTraversal(op);
   op.processNode(element);
}
```

Binary Tree Traversal (cont.)

- This shows how we can use the traversal functions.
  - We need to define a subclass of the class `TreeOperation` where the function `processNode` is defined.
  - In this case we simply print out the element of the node.

```
class PrintTreeNode : public TreeOperation
{
 public:
   void processNode(int element) {
      cout << element << " ";
   }
};
```

Printing a Tree

- In the main function, we generate a tree (we will discuss this later), and then we can print out the elements of the tree depending on our chosen traversal.

```
int main()
{
   BinaryTree bt;
   // here generate the tree bt with
   // integer elements
   ...
   PrintTreeNode ptn;
   bt.getRoot()->preOrderTraversal(ptn);
   cout << endl;
   ...
}
```

- Or we could implement the traversal functions in the class `BinaryTree` and then we would simply call `bt.preOrderTraversal(ptn)`.

Binary Search Tree

Definition

The set of **extended binary trees** can be defined recursively.

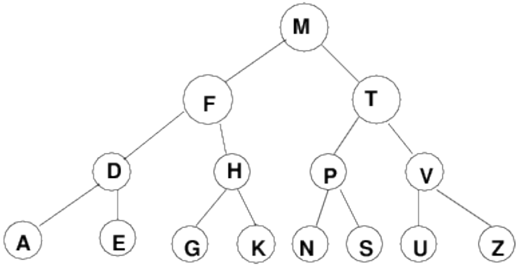
- The empty set is an extended binary tree.
- If T_1 and T_2 are extended binary trees, there is an extended binary tree T consisting of a root r together with edges connecting the root r to each of the roots of T_1 and T_2.

Sorting property for extended binary tree:

- For any given data item X in the extended nonempty binary tree, every node in the left nonempty subtree of X contains only items that are less than X with respect to a given sorting.
- Every node in the right nonempty subtree of X contains only items that are greater than or equal to X with respect to the same sorting.

Definition of a Binary Search Tree

- A binary search tree is an extended binary tree with sorting property.



The Class BinarySearchTree

```
class BinarySearchTree { // uses struct BinaryNode
 private:
   BinaryNode *root;
   // some private functions will be here
 public:
   BinarySearchTree( ) { root = nullptr; }
   ~BinarySearchTree( ) { deleteTree(root);
                          root = nullptr; }
   int size() { return (root==nullptr) ?
                  0 : root->size(root);}
   int height() { return (root==nullptr) ?
                  -1 : root->height(root);}
   void insert(int x)
   {  root = insert(x, root); }
   void remove(int x)
   {  root = remove(x, root);  }
   BinaryNode *find(int x)
   { return find(x, root); }
   bool isEmpty( ){ return root == nullptr; }
};
```

Finding Nodes

- The function `findMin` looks for the node containing the smallest item.
  - Such a node is the most left node of the tree.
  - For the function `find` (see the next slide) as long as a `nullptr` reference has not been reached, we either have found the item or need to branch left or right depending on the result of comparison.

```
// private function
// it finds a node containing the smallest item
BinaryNode *BinarySearchTree::findMin(BinaryNode *t)
{
   if (t == nullptr) throw EmptyTree();
   while (t->left != nullptr) t = t->left;
   return t;
}
```

Implementation of `BinarySearchTree` Functions

```
// private function
// it finds a node containing element x
BinaryNode *BinarySearchTree::find(int x,
                                   BinaryNode *t)
{
   while (t != nullptr) {
      if (x < t->element) t = t->left;
      else if (x > t->element) t = t->right;
      else return t; // found
   }
   throw ItemNotFound();
}
```

Inserting Nodes

- Insertion of a new node always occurs at the leaf nodes of a tree: there are no data movement.
- To insert a node we need traverse $O(\log_2 N)$ nodes, where N is the number of nodes in the tree, assuming that the tree is full
    - if m is the deepest level of the tree then all the nodes on levels from 1 to $m-1$ have children
- If the tree is not empty, there are three possibilities:
    - if the item to be inserted is smaller than the item in node t, then we need to call insert recursively on the left subtree.
    - if the item to be inserted is larger than the item in node t, then we need to call insert recursively on the right subtree.
    - the item to insert matches the item in t. Since it is a duplicate item, we throw an exception.

An Insert Node Function

```
// private function
// it inserts a new node containing the element x
BinaryNode *BinarySearchTree::insert(int x,
                                      BinaryNode *t)
{
   if (t == nullptr) t = new BinaryNode(x);
   else if (x < t->element)
      t->left = insert(x, t->left);
   else if (x > t->element)
      t->right = insert(x, t->right);
   else
      throw DuplicateItem();
   return t;
}
```

Removing Nodes

- The function `removeMin` removes the node containing the smallest item, that is, the most left node of the tree.
- Since this node has no left child, it is simply bypassed (by `t = t->right`).

```
BinaryNode *BinarySearchTree::removeMin(BinaryNode *t)
{
   if (t == nullptr) throw ItemNotFound();
   if (t->left != nullptr)
      t->left = removeMin(t->left);
   else {
      BinaryNode *node = t;
      t = t->right;
      delete node;
   }
   return t;
}
```

A Remove Function

- If we want to remove a node we need to do so in order to preserve the sorting property of the tree.
  - If there are two children, we replace the node with the minimum element in the right subtree and then remove the right subtree's minimum.
  - Otherwise, there is only one child or zero children. If there is a left child, we set *t* to its left child.
  - Otherwise, we know there is no left child, so we can set *t* to its right child.

A Remove Function (cont)

```
BinaryNode *BinarySearchTree::remove(int x,
                                     BinaryNode *t)
{
   if (t == nullptr) throw ItemNotFound();
   if (x < t->element)
      t->left = remove(x, t->left);
   else if (x > t->element)
      t->right = remove(x, t->right);
   else if (t->left != nullptr && t->right != nullptr){
      // item x is found; t has two children
      t->element = findMin(t->right)->element;
      t->right = removeMin(t->right);
   } else  { //t has only one child
      BinaryNode *node = t;
      t = (t->left != nullptr) ? t->left : t->right;
      delete node;
   }
   return t;
}
```