Data Structures (in C++)

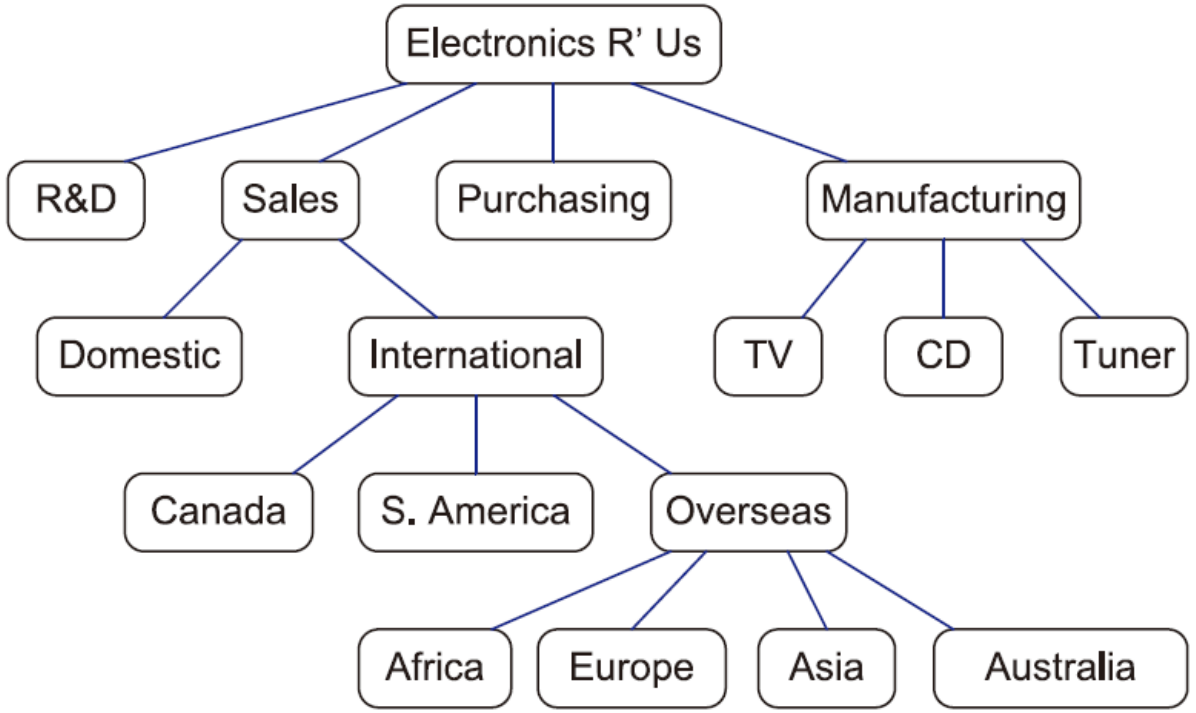
- Trees, Tree Traversal Algorithms, and Binary Trees -

부산대학교 정보·의생명공학대학
정보컴퓨터공학부

Trees

Trees

- **Tree**
  - A data type that stores elements hierarchically (*Above* and *below*)
  - Each element has a *parent* and zero or more *children* elements (except for the top element)
  - The top element is called the *root* of the tree



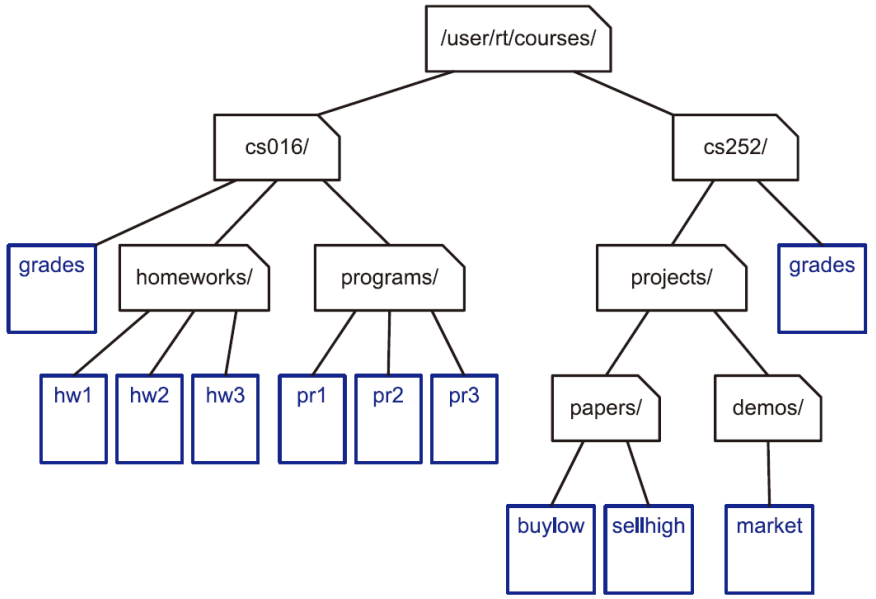
Trees

- **Formal Tree Definition**
  - We define tree T to be a set of *nodes* storing elements in a *parent-child* relationship with the following properties:

    - If T is nonempty, it has a special node, called the ***root*** of T, that has no parent.
    - Each node v of T different from the root has a unique ***parent*** node w; every node with parent w is a ***child*** of w.

  - Two nodes with the same parent are *siblings*
  - A node is *external* (or a *leaf*) if it has no children
  - A node is *internal* if it has one or more children

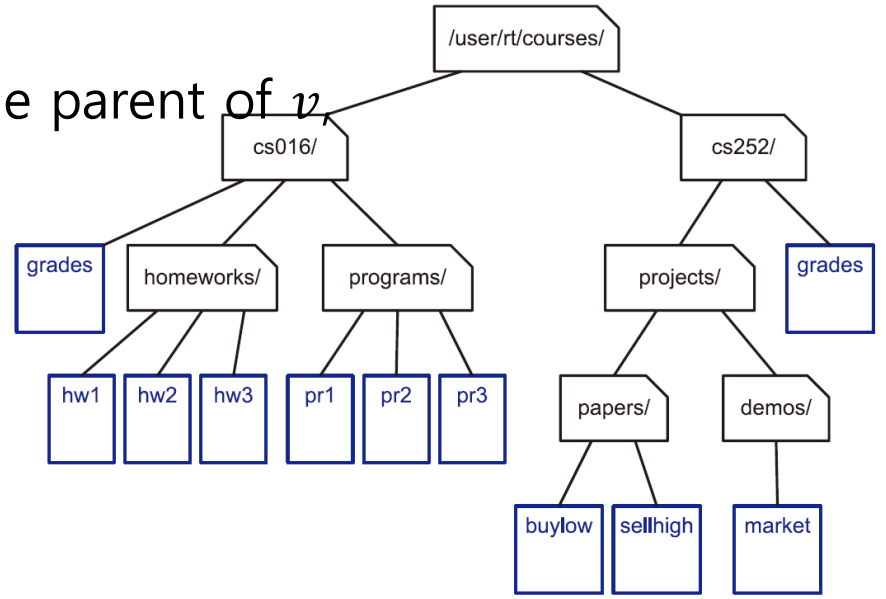
UNIX system root directory : /



Trees

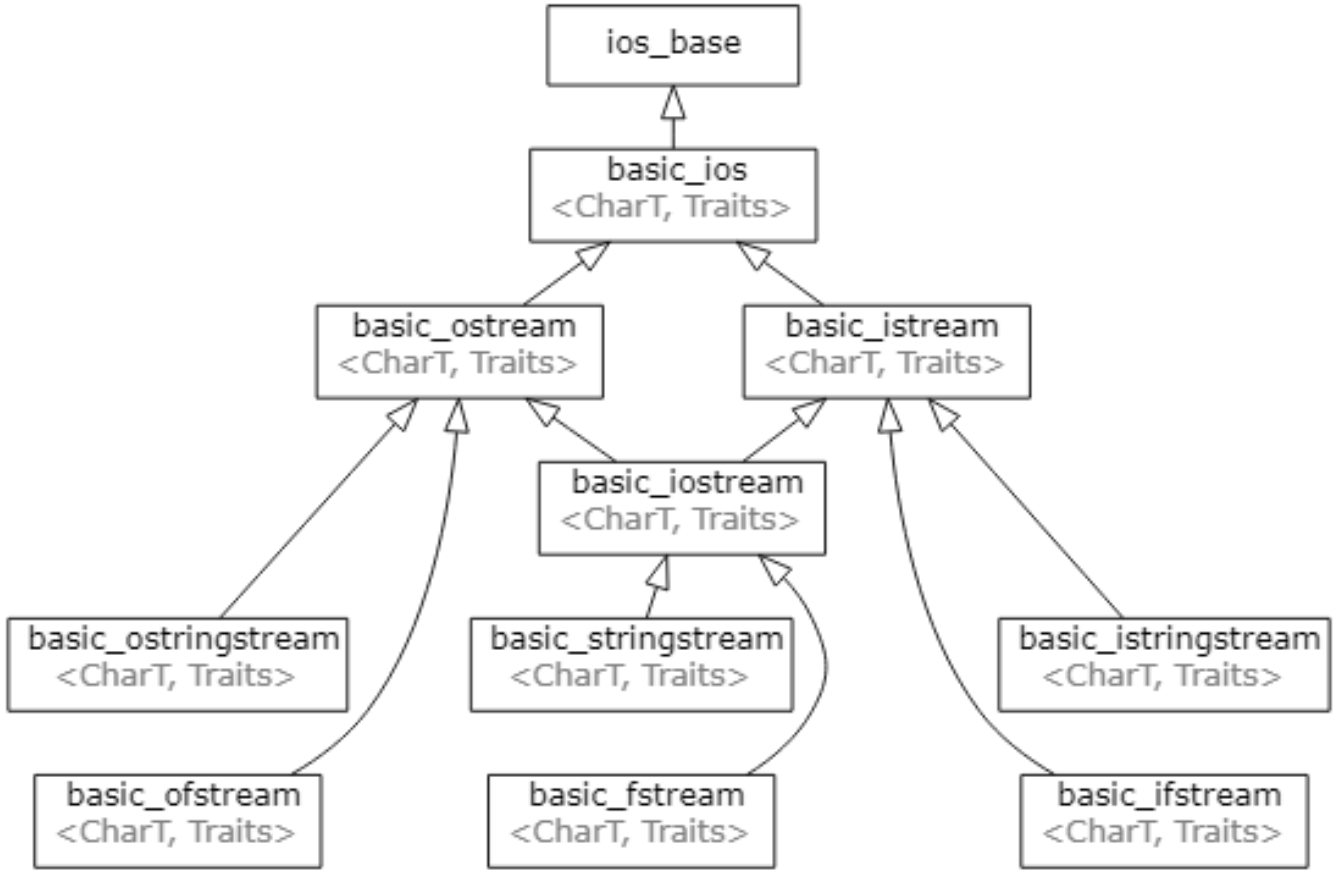
- **Formal Tree Definition**
  - A node u is an *ancestor* of a node v if $u = v$ or u is an ancestor of the parent of v
    - Any other node on the path from the node to the root
  - A node v is a *descendant* of a node u if u is an ancestor of v
  - The *subtree* of T rooted at a node v is the tree consisting of all the descendants of v (including v itself)
  - An *edge* of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa.
  - A *path* of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge

UNIX system root directory : /



Trees

- **Tree Example: C++ Inheritance**



https://en.cppreference.com/w/cpp/io#Stream-based_I.2FO

Trees

- **Ordered Tree**
  - There is a linear ordering defined for the children of each node
  - Children of a node can be identified as the first, second, third and so on (*i.e.*, left to right)

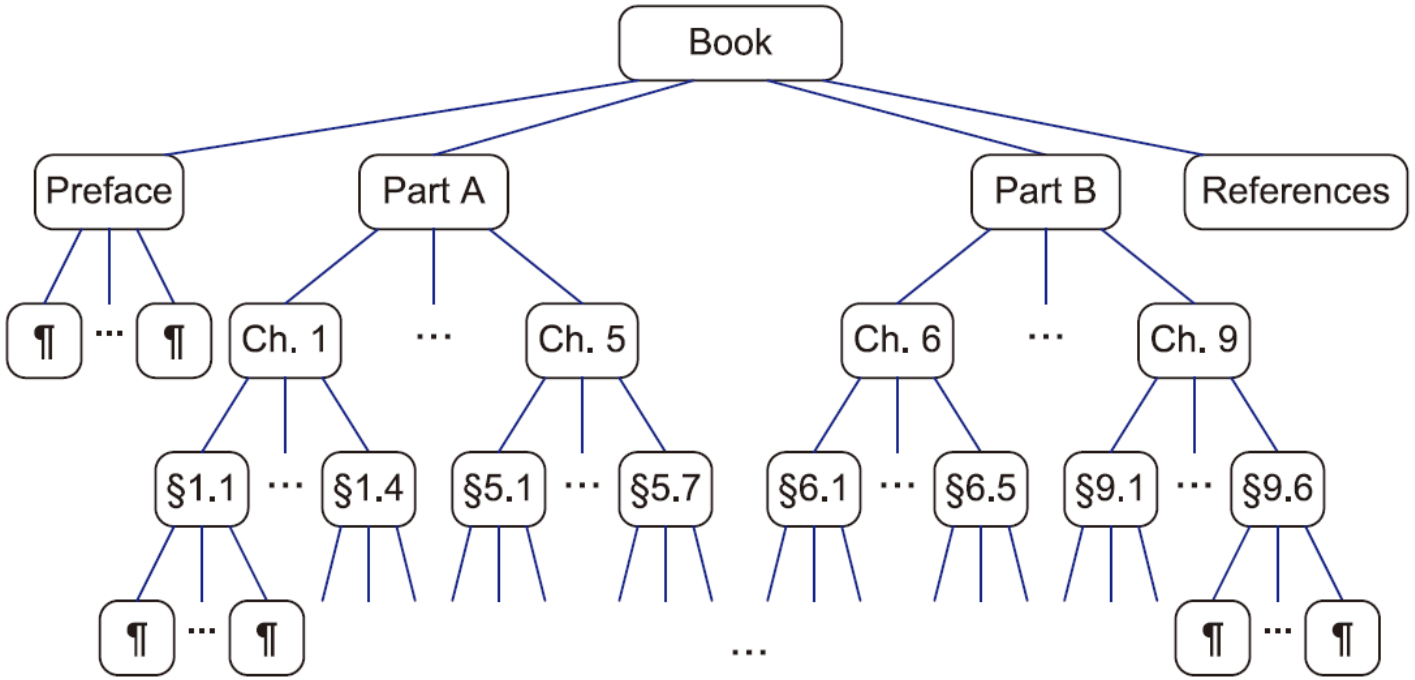


Figure 7.4: An ordered tree associated with a book.

Tree ADT

- It stores elements at the nodes
- Each node of the tree is associated with a *position* object
  - Provides public access to nodes
- The tree ADT supports the following operations given a position p of tree T:

p.parent(): Return the parent of p; an error occurs if p is the root.

p.children(): Return a position list containing the children of node p.

p.isRoot(): Return true if p is the root and false otherwise.

p.isExternal(): Return true if p is external and false otherwise.

- Additional utility functions:

size(): Return the number of nodes in the tree.

empty(): Return true if the tree is empty and false otherwise.

root(): Return a position for the tree's root; an error occurs if the tree is empty.

positions(): Return a position list of all the nodes of the tree.

Tree ADT

- **C++ Implementation**
  - Position implementation

std::list<Position>

```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position parent() const;               // get parent
  PositionList children() const;         // get node's children
  bool isRoot() const;                   // root node?
  bool isExternal() const;               // external node?
};
```

Code Fragment 7.1: An informal interface for a position in a tree (not a complete C++ class).

Tree ADT

- **C++ Implementation**
  - Tree implementation

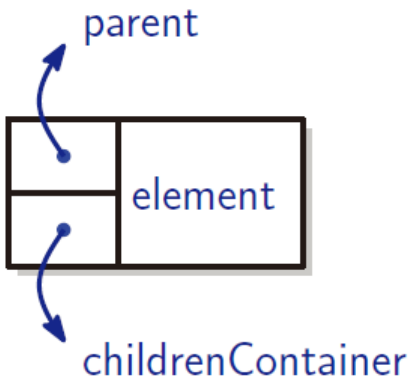
```
template <typename E>                    // base element type
class Tree<E> {
public:                                  // public types
  class Position;                        // a node position
  class PositionList;                    // a list of positions
public:                                  // public functions
  int size() const;                      // number of nodes
  bool empty() const;                    // is tree empty?
  Position root() const;                 // get the root
  PositionList positions() const;        // get positions of all nodes
};
```

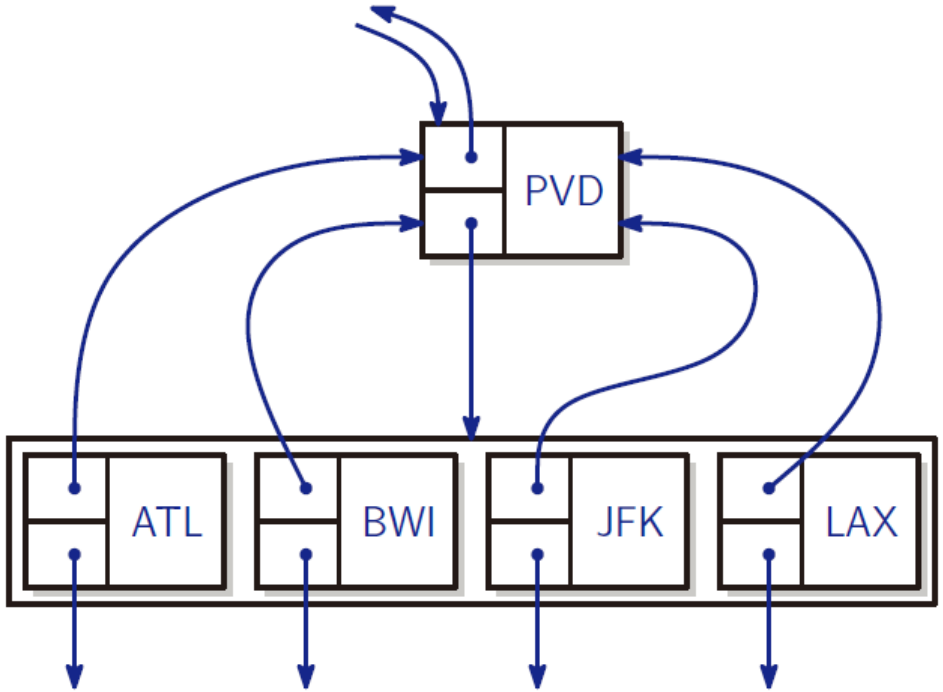
Code Fragment 7.2: An informal interface for the tree ADT (not a complete class).

  - There is no C++ STL container *std::tree* but *std::map* is implemented as a red/black tree

Tree ADT

- **A Linked Structure for General Trees**
  - Each node of T by a position object p has:
    - A reference to the node's element
    - A link to the node's parent
    - A collection to store links to the node's children (*e.g.*, PositionList)





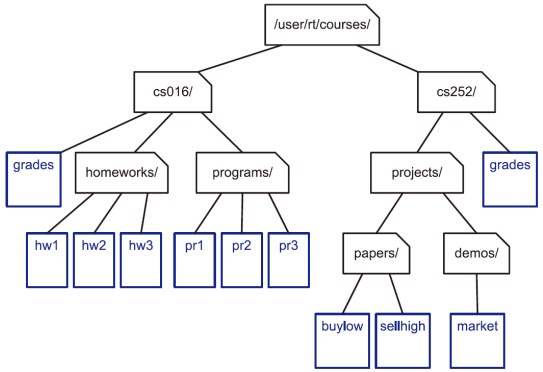
using iterator

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Tree Traversal Algorithms

Tree Traversal Algorithms

- **Depth**
  - The depth of p is the number of ancestors of p, excluding p itself
  - The number of edges from the node to the root node



- If p is the root, then the depth of p is 0
- Otherwise, the depth of p is one plus the depth of the parent of p

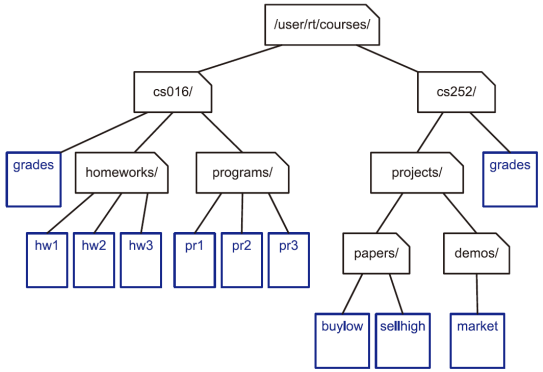
Algorithm depth(T, p):
  if p.isRoot() **then**
    return 0
  else
    return $1 +$ depth$(T, p$.parent$())$

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                          // root has depth 0
  else
    return 1 + depth(T, p.parent());   // 1 + (depth of parent)
}
```

- Time complexity of $depth(T, p) = O(d_p)$

Tree Traversal Algorithms

- **Height**
  - The number of edges on the longest path from the node to a leaf



    - If p is external, then the height of p is 0
    - Otherwise, the height of p is one plus the maximum height of a child of p

  - The height of a tree is the height of the root

  - The height of a tree is equal to the maximum depth of its external nodes

Tree Traversal Algorithms

- **Height Computation**

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Algorithm height1(T):
  $h = 0$
  for each $p \in T$.positions() **do**
    if p.isExternal() **then**
      $h = \max(h, \text{depth}(T, p))$
  return h

```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions();          // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
    if (q->isExternal())
      h = max(h, depth(T, *q));                // get max depth among leaves
  }
  return h;
}
```

$$O\left(n + \sum_p (1 + d_p)\right)$$

Tree Traversal Algorithms

- **Height Computation**

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Algorithm height2(T, p):
  if p.isExternal() **then**
    return 0
  else
    $h = 0$
    for each $q \in p$.children() **do**
      $h = \max(h, \text{height2}(T, q))$
    return $1 + h$

```
int height2(const Tree& T, const Position& p) {
  if (p.isExternal()) return 0;                  // leaf has height 0
  int h = 0;
  PositionList ch = p.children();                // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q)
    h = max(h, height2(T, *q));
  return 1 + h;                                  // 1 + max height of children
}
```

Recursive call

Proposition 7.5: *Let T be a tree with n nodes, and let c_p denote the number of children of a node p of T. Then $\sum_p c_p = n - 1$.*

Justification: Each node of T, with the exception of the root, is a child of another node, and thus contributes one unit to the above sum. ■

$$O\left(\sum_p (1 + c_p)\right) = O(n)$$

Tree Traversal Algorithms

- **Traversal**
  - A systematic way of visiting all the nodes of a tree

- **Preorder Traversal**
  - The root of T is visited first and then the subtrees rooted at its children are traversed recursively
  - The subtrees are traversed according to the order of the children if the tree is ordered
  - Parents always come before their children

Algorithm preorder(T, p):
perform the "visit" action for node p
for each child q of p **do**
recursively traverse the subtree rooted at q by calling preorder(T, q)

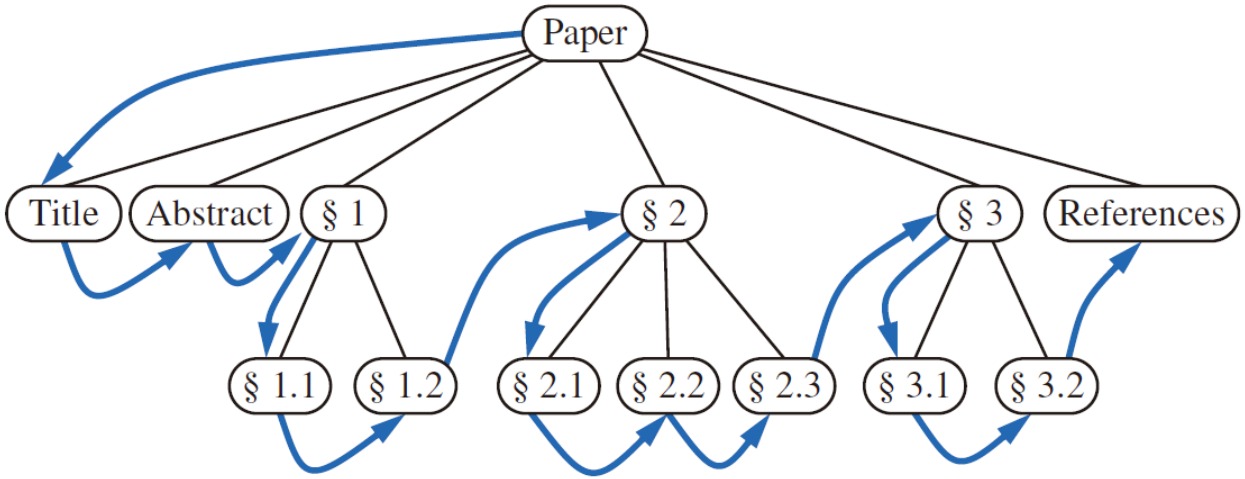


Figure 7.6: Preorder traversal of an ordered tree, where the children of each node are ordered from left to right.

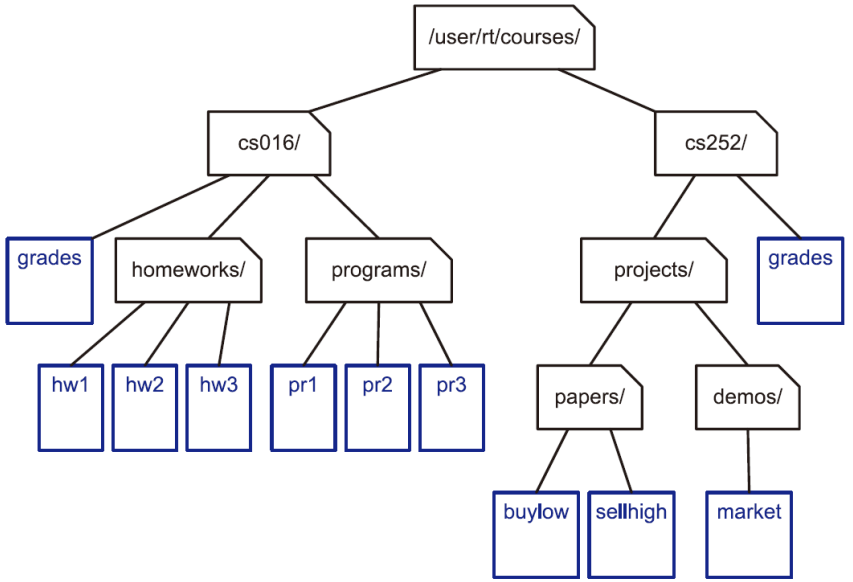
Tree Traversal Algorithms

- **Preorder Traversal**
  - Assume that visiting a node takes $O(1)$

$$O\left(\sum_p (1 + c_p)\right)$$

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                                  // print element
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

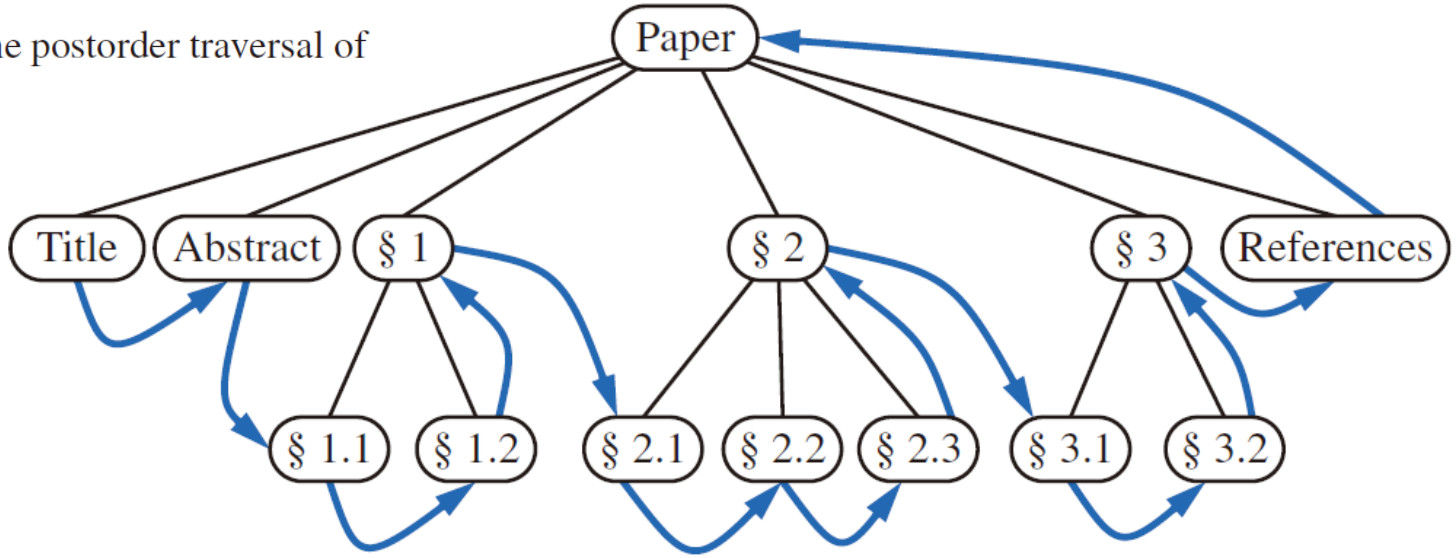


Tree Traversal Algorithms

- **Postorder Traversal**
  - Recursively traverses the subtrees rooted at the children of the root first, and then visits the root
  - Visits a node p after it has visited all the other nodes in the subtree rooted at p

Algorithm postorder(T, p):
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling postorder(T, q)
  perform the "visit" action for node p

Code Fragment 7.12: Algorithm postorder for performing the postorder traversal of the subtree of a tree T rooted at a node p.



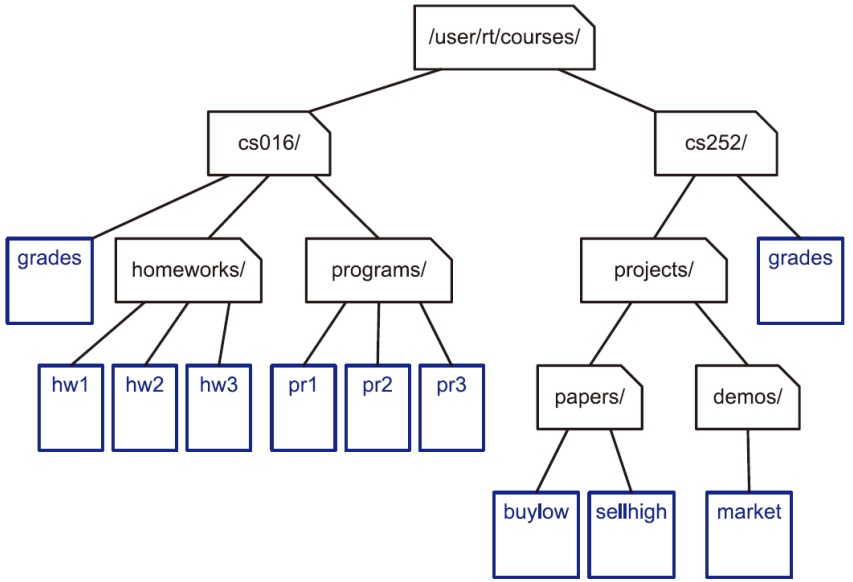
Tree Traversal Algorithms

- **Postorder Traversal**
  - Assume that visiting a node takes $O(1)$

$$O\left(\sum_p (c_p + 1)\right)$$

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

```
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                  // print element
}
```



Tree Traversal Algorithms

- **Postorder Traversal Example: File System Disk Space**

```
int diskSpace(const Tree& T, const Position& p) {
  int s = size(p);                              // start with size of p
  if (!p.isExternal()) {                        // if p is internal
    PositionList ch = p.children();             // list of p's children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      s += diskSpace(T, *q);                    // sum the space of subtrees
    cout << name(p) << ": " << s << endl; // print summary
  }
  return s;
}
```

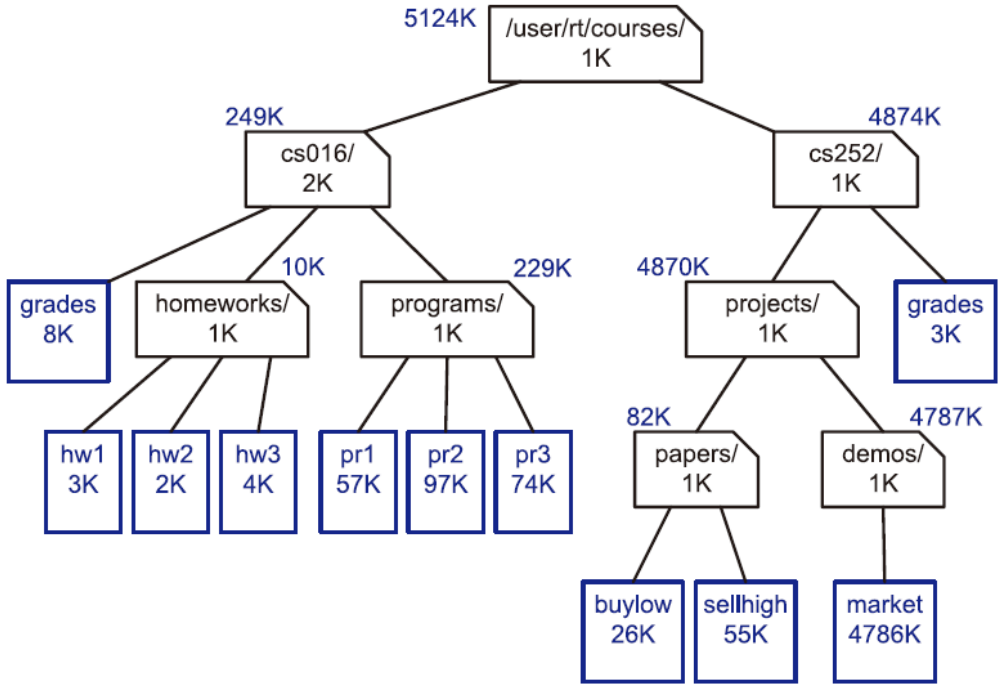


Figure 7.9: The tree of Figure 7.3 representing a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.

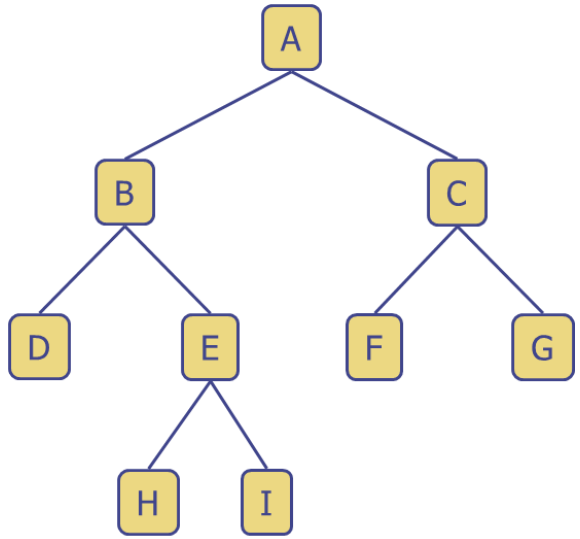
Binary Trees

Binary Trees

- **Binary Tree**
  - An *ordered tree* in which every node has *at most two children*

    1. Every node has at most two children.
    2. Each child node is labeled as being either a ***left child*** or a ***right child***.
    3. A left child precedes a right child in the ordering of children of a node.

  - The subtree rooted at a left (right) child of an internal node is called the *left (right) subtree*
  - If each node has either zero or two children, a binary tree is called a *proper* (or *full*) binary tree
  - Otherwise, it is called *improper*



- **Recursive Binary Tree Definition**
  - A node r, called the ***root*** of T and storing an element
  - A binary tree, called the ***left subtree*** of T
  - A binary tree, called the ***right subtree*** of T

Binary Trees

- **Example: Binary Decision Tree**
  - Follows an edge from a parent to a child with each decision
  - A decision tree is a proper binary tree

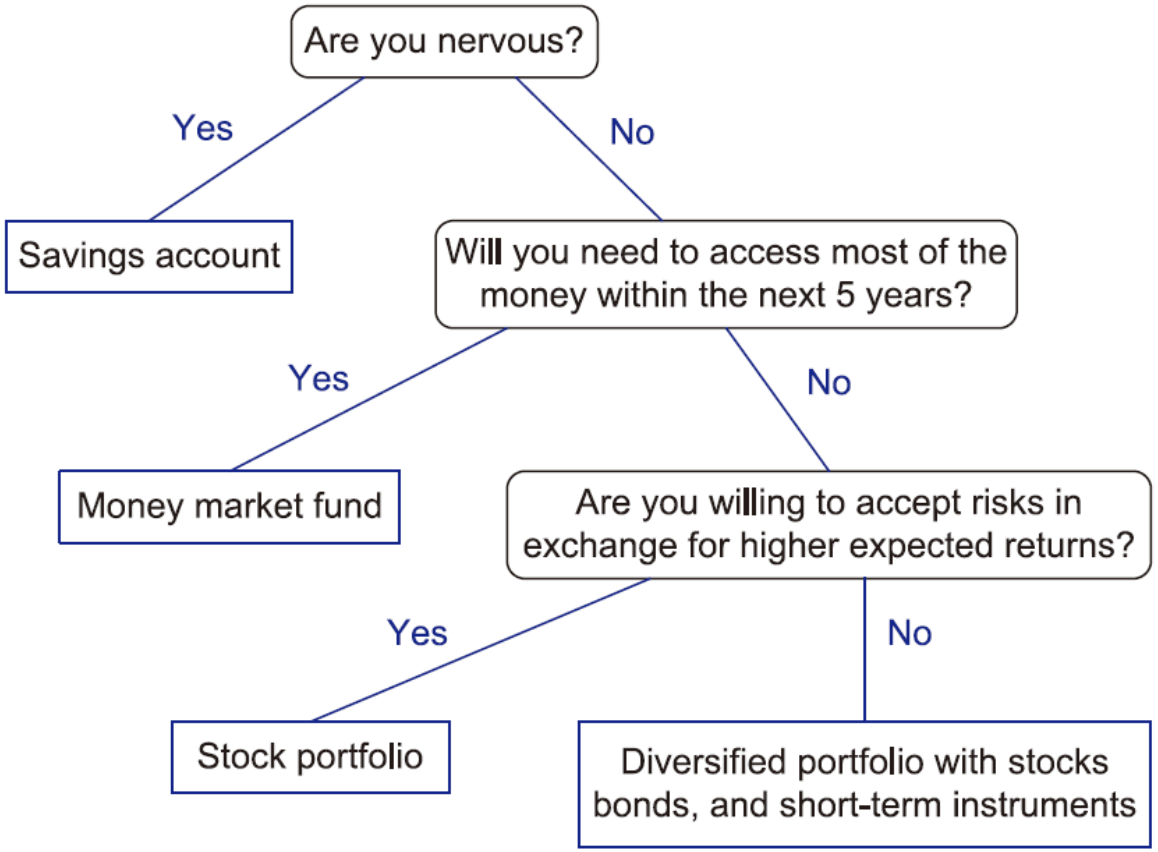


Figure 7.10: A decision tree providing investment advice.

Binary Trees

- **Example: Arithmetic Expression Tree**
  - An arithmetic expression can be represented by a tree
  - External nodes: variables or constants
  - Internal nodes: arithmetic operations

- *If a node is external, then its value is that of its variable or constant.*
- *If a node is internal, then its value is defined by applying its operation to the values of its children.*

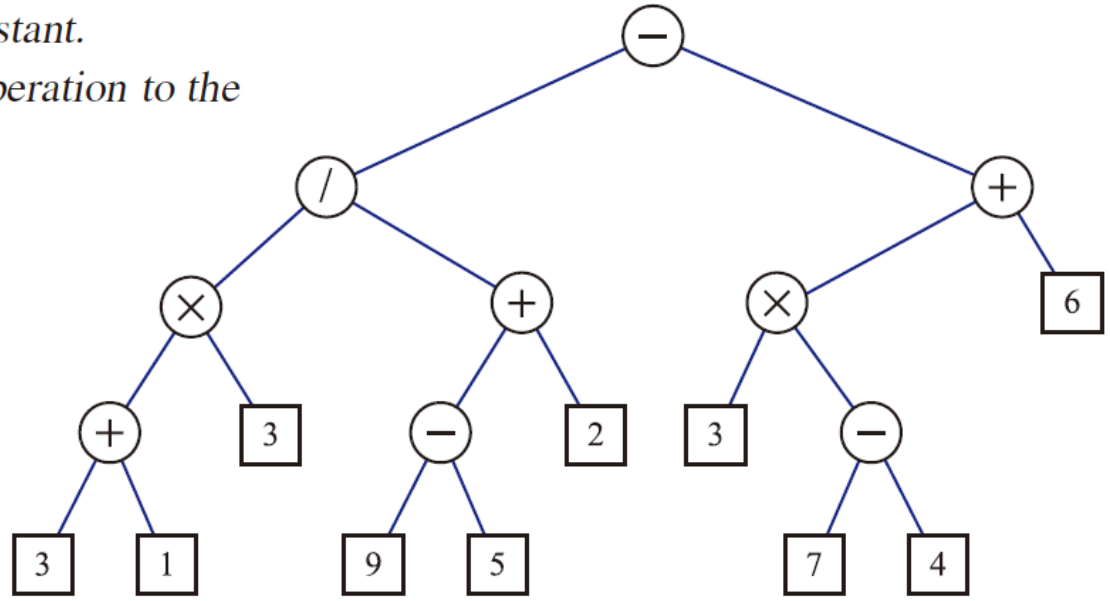


Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Binary Tree ADT

- Each node of the tree stores an element
- Each node is associated with a *position* object

p.left():	Return the left child of p; an error condition occurs if p is an external node.
p.right():	Return the right child of p; an error condition occurs if p is an external node.
p.parent():	Return the parent of p; an error occurs if p is the root.
p.isRoot():	Return true if p is the root and false otherwise.
p.isExternal():	Return true if p is external and false otherwise.
size():	Return the number of nodes in the tree.
empty():	Return true if the tree is empty and false otherwise.
root():	Return a position for the tree's root; an error occurs if the tree is empty.
positions():	Return a position list of all the nodes of the tree.

Binary Tree ADT

- **C++ Implementation**
  - *left()* and *right()* replace *children()* of the *Position* of a tree

```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position left() const;                 // get left child
  Position right() const;                // get right child
  Position parent() const;               // get parent
  bool isRoot() const;                   // root of tree?
  bool isExternal() const;               // an external node?
};
```

Position ADT of a binary tree

```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position parent() const;               // get parent
  PositionList children() const;         // get node's children
  bool isRoot() const;                   // root node?
  bool isExternal() const;               // external node?
};
```

Code Fragment 7.1: An informal interface for a position in a tree (not a complete C++ class).

Position ADT of a tree

Binary Tree ADT

- **C++ Implementation**

```
template <typename E>                    // base element type
class BinaryTree<E> {                    // binary tree
public:                                  // public types
  class Position;                        // a node position
  class PositionList;                    // a list of positions
public:                                  // member functions
  int size() const;                      // number of nodes
  bool empty() const;                    // is tree empty?
  Position root() const;                 // get the root
  PositionList positions() const;        // list of nodes
};
```

Code Fragment 7.16: An informal interface for the binary tree ADT (not a complete C++ class).

Properties of Binary Trees

- **Level**
  - The set of all nodes of a tree at the same depth
  - In general, level d has 2^d nodes at most

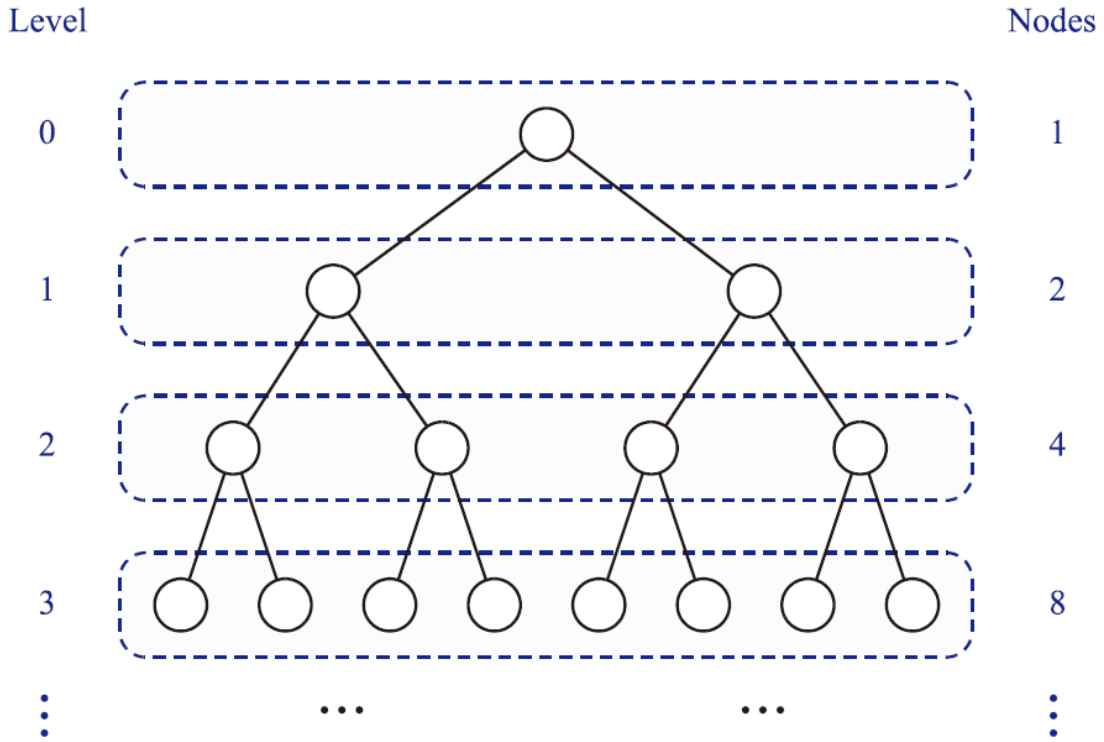


Figure 7.12: Maximum number of nodes in the levels of a binary tree.

Properties of Binary Trees

Proposition 7.10: *Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:*

1. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq n-1$

Also, if T is proper, then it has the following properties:

1. $2h+1 \leq n \leq 2^{h+1}-1$ Geometric progression
2. $h+1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq (n-1)/2$

n: The number of nodes
n_E: The number of external nodes
n_I: The number of internal nodes
h: The height of the tree

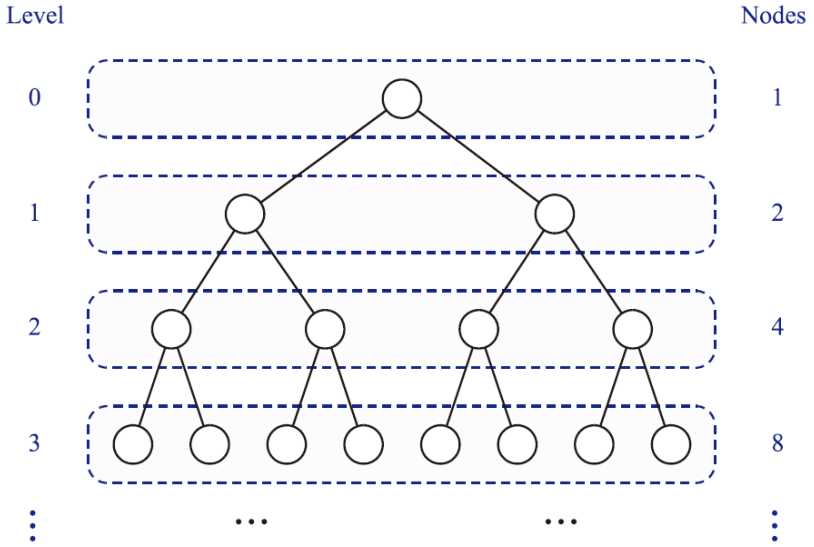


Figure 7.12: Maximum number of nodes in the levels of a binary tree.

Properties of Binary Trees

Proposition 7.11: *In a nonempty proper binary tree T, the number of external nodes is one more than the number of internal nodes.*

Justification: We can see this using an argument based on induction. If the tree consists of a single root node, then clearly we have one external node and no internal nodes, so the proposition holds.

If, on the other hand, we have two or more, then the root has two subtrees. Since the subtrees are smaller than the original tree, we may assume that they satisfy the proposition. Thus, each subtree has one more external node than internal nodes. Between the two of them, there are two more external nodes than internal nodes. But, the root of the tree is an internal node. When we consider the root and both subtrees together, the difference between the number of external and internal nodes is $2 - 1 = 1$, which is just what we want. ■

n: The number of nodes
n_E: The number of external nodes
n_I: The number of internal nodes
h: The height of the tree

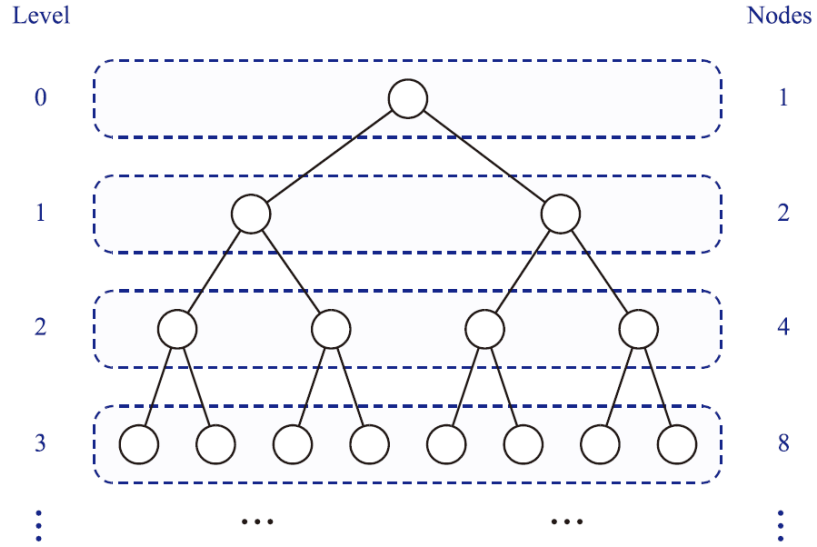
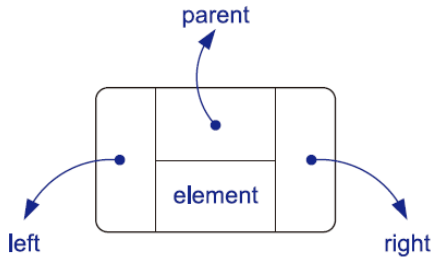


Figure 7.12: Maximum number of nodes in the levels of a binary tree.

Binary Trees: Linked Structure

- **Linked Binary Tree**
  - A binary tree can be implemented by a *linked structure*
  - Assume the tree is *proper*



A node in a linked binary tree

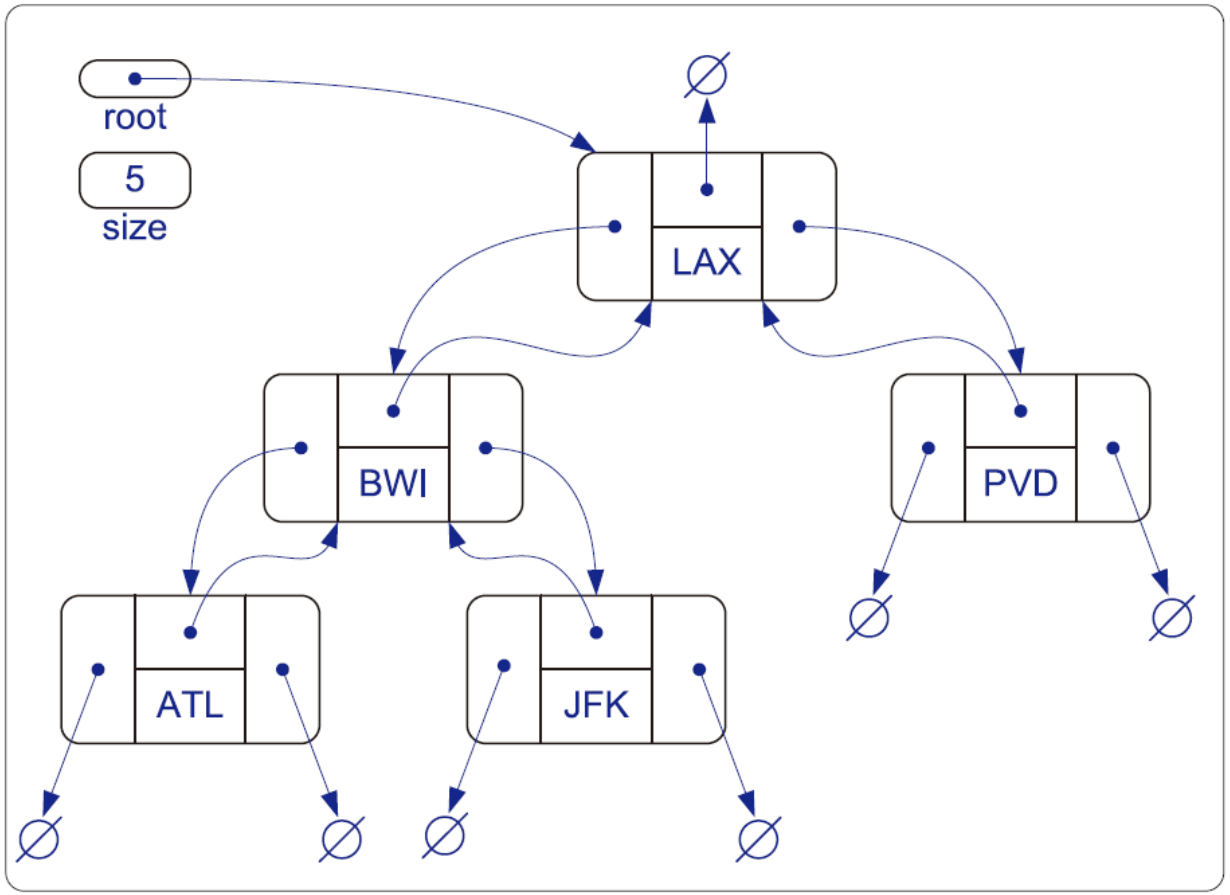


Figure 7.14: An example of a linked data structure for representing a binary tree.

Binary Trees: Linked Structure

- **C++ Implementation**

```
struct Node {                                   // a node of the tree
  Elem    elt;                                  // element value
  Node*   par;                                  // parent
  Node*   left;                                 // left child
  Node*   right;                                // right child
  Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
};
```

```
class Position {                                // position in the tree
private:
  Node* v;                                      // pointer to the node
public:
  Position(Node* _v = NULL) : v(_v) { }         // constructor
  Elem& operator*()                             // get element
    { return v->elt; }
  Position left() const                         // get left child
    { return Position(v->left); }
  Position right() const                        // get right child
    { return Position(v->right); }
  Position parent() const                       // get parent
    { return Position(v->par); }
  bool isRoot() const                           // root of the tree?
    { return v->par == NULL; }
  bool isExternal() const                       // an external node?
    { return v->left == NULL && v->right == NULL; }
  friend class LinkedBinaryTree;                // give tree access
};
typedef std::list<Position> PositionList;       // list of positions
```

Code Fragment 7.18: Class Position implementing a position in a binary tree. It is nested in the public section of class LinkedBinaryTree.

Binary Trees: Linked Structure

- **C++ Implementation**

```
typedef int Elem;                                  // base element type
class LinkedBinaryTree {
protected:
  // insert Node declaration here...
public:
  // insert Position declaration here...
public:
  LinkedBinaryTree();                              // constructor
  int size() const;                                // number of nodes
  bool empty() const;                              // is tree empty?
  Position root() const;                           // get the root
  PositionList positions() const;                  // list of nodes
  void addRoot();                                  // add root to empty tree
  void expandExternal(const Position& p);          // expand external node
  Position removeAboveExternal(const Position& p); // remove p and parent
  // housekeeping functions omitted...
protected:                                         // local utilities
  void preorder(Node* v, PositionList& pl) const;  // preorder utility
private:
  Node* _root;                                     // pointer to the root
  int n;                                           // number of nodes
};
```

public Node implementation is protected

```
LinkedBinaryTree::LinkedBinaryTree()               // constructor
  : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const                 // number of nodes
  { return n; }
bool LinkedBinaryTree::empty() const               // is tree empty?
  { return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
  { return Position(_root); }
void LinkedBinaryTree::addRoot()                   // add root to empty tree
  { _root = new Node; n = 1; }
```

Code Fragment 7.20: Simple member functions for class LinkedBinaryTree.

Binary Trees: Linked Structure

- **Binary Tree Update Functions**

expandExternal(p): Transform p from an external node into an internal node by creating two new external nodes and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.

removeAboveExternal(p): Remove the external node p together with its parent q, replacing q with the sibling of p (see Figure 7.15, where p's node is w and q's node is v); an error condition occurs if p is an internal node or p is the root.

Removing a child alone results in an improper tree

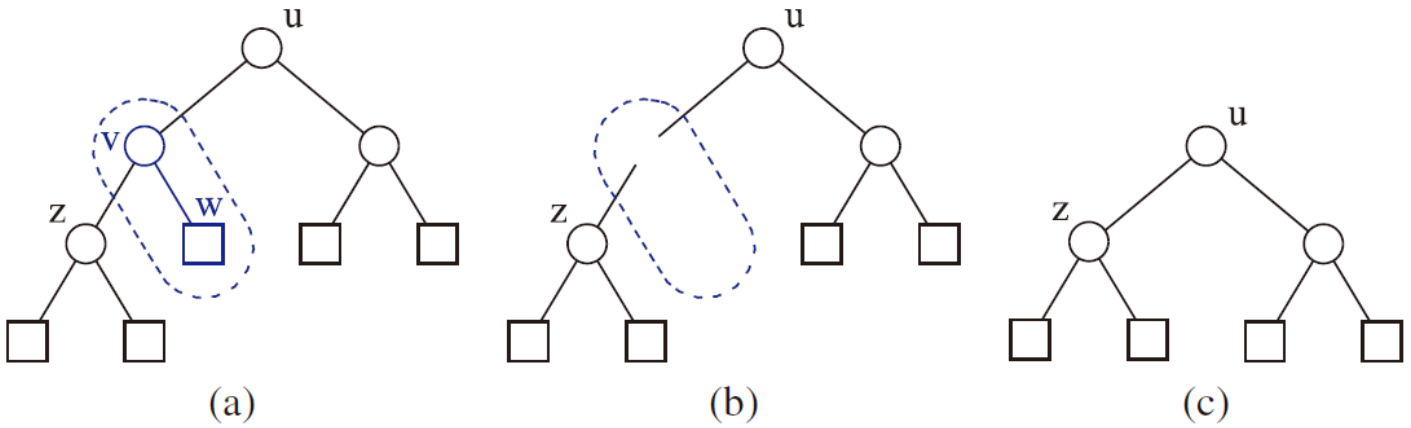


Figure 7.15: Operation removeAboveExternal(p), which removes the external node w to which p refers and its parent node v.

Binary Trees: Linked Structure

- **Binary Tree Update Functions**
  - Expansion: Create left and right children
  - Removal: Replace a node's parent with its sibling

```
                                                    // expand external node
void LinkedBinaryTree::expandExternal(const Position& p) {
  Node* v = p.v;                                    // p's node
  v->left = new Node;                               // add a new left child
  v->left->par = v;                                 // v is its parent
  v->right = new Node;                              // and a new right child
  v->right->par = v;                                // v is its parent
  n += 2;                                           // two more nodes
}
```

Code Fragment 7.21: The function expandExternal(p) of class LinkedBinaryTree.

```
LinkedBinaryTree::Position                          // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p) {
  Node* w = p.v; Node* v = w->par;                  // get p's node and parent
  Node* sib = (w == v->left ? v->right : v->left);
  if (v == _root) {                                 // child of root?
    _root = sib;                                    // ...make sibling root
    sib->par = NULL;
  }
  else {
    Node* gpar = v->par;                            // w's grandparent
    if (v == gpar->left) gpar->left = sib;          // replace parent by sib
    else gpar->right = sib;
    sib->par = gpar;
  }
  delete w; delete v;                               // delete removed nodes
  n -= 2;                                           // two fewer nodes
  return Position(sib);
}
```

Code Fragment 7.22: An implementation of the function removeAboveExternal(p).

Binary Trees: Linked Structure

- **Binary Tree Preorder Traversal**

```
                                                        // list of all nodes
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder(_root, pl);                                  // preorder traversal
  return PositionList(pl);                              // return resulting list
}

                                                        // preorder traversal
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const {
  pl.push_back(Position(v));                            // add this node
  if (v->left != NULL)                                  // traverse left subtree
    preorder(v->left, pl);
  if (v->right != NULL)                                 // traverse right subtree
    preorder(v->right, pl);
}
```

Preorder because a binary tree is ordered

Binary Trees: Linked Structure

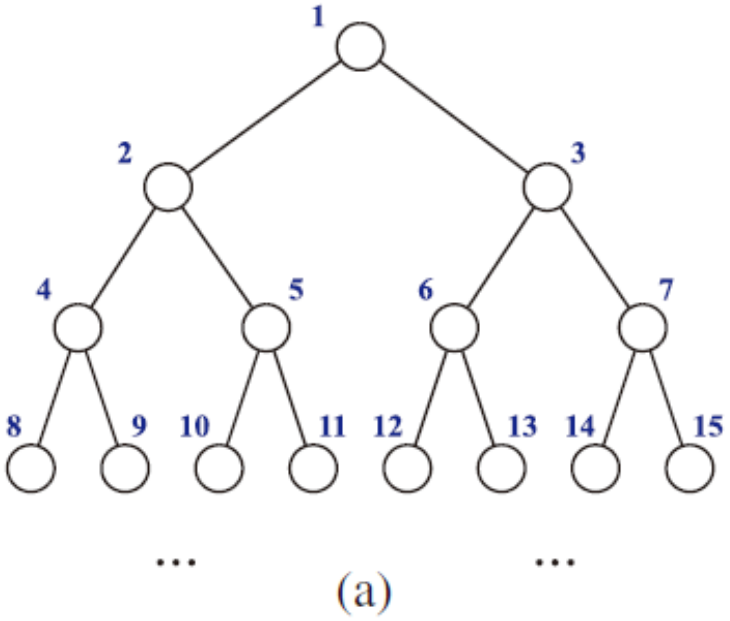
- **Linked Structure Performance**
  - Space complexity is $O(n)$

- Each of the position functions left, right, parent, isRoot, and isExternal takes $O(1)$ time.
- By accessing the member variable n, which stores the number of nodes of T, functions size and empty each run in $O(1)$ time.
- The accessor function root runs in $O(1)$ time.
- The update functions expandExternal and removeAboveExternal visit only a constant number of nodes, so they both run in $O(1)$ time.
- Function positions is implemented by performing a preorder traversal, which takes $O(n)$ time. (We discuss three different binary-tree traversals in Section 7.3.6. Any of these suffice.) The nodes visited by the traversal are each added in $O(1)$ time to an STL list. Thus, function positions takes $O(n)$ time.

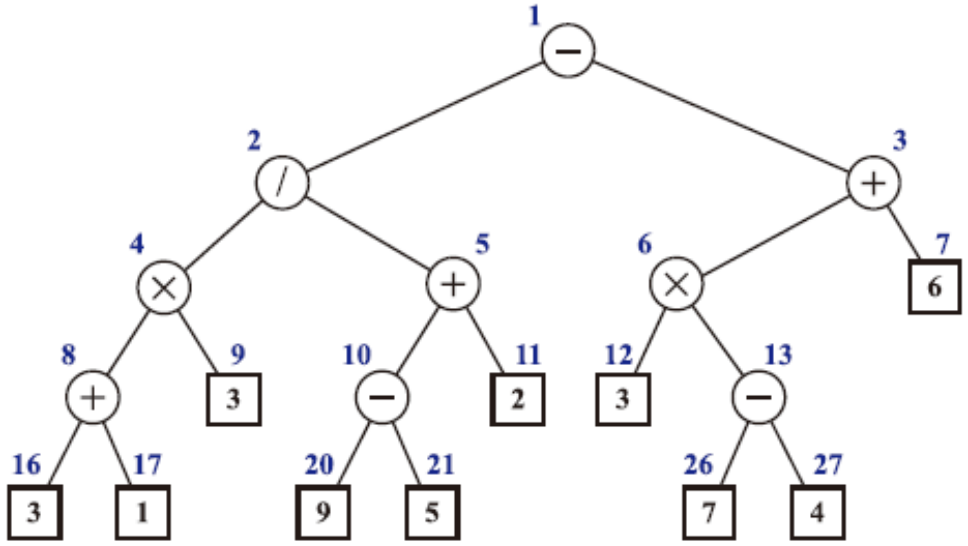
Operation	*Time*
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

Binary Trees: Vector-based Structure

- **Vector-Based Binary Tree**
  - For every node v of T, let $f(v)$ be the integer defined as follows:
    - If v is the root of T, then $f(v) = 1$
    - If v is the left child of node u, then $f(v) = 2f(u)$
    - If v is the right child of node u, then $f(v) = 2f(u) + 1$
  - f is known as a *level numbering* of the nodes



(a)



(b)

Binary Trees: Vector-based Structure

- **Vector-Based Binary Tree**
  - The node v of T is associated with the element of a vector S at rank $f(v)$
  - Let n be the number of nodes of T, f_M be the maximum value of $f(v)$ over all the nodes of T
  - The vector S has size $N = f_M + 1$ (*cf.* no element with index 0)
  - For a tree of height h, $N = O(2^h)$ ($2^n - 1$ for the worst case)

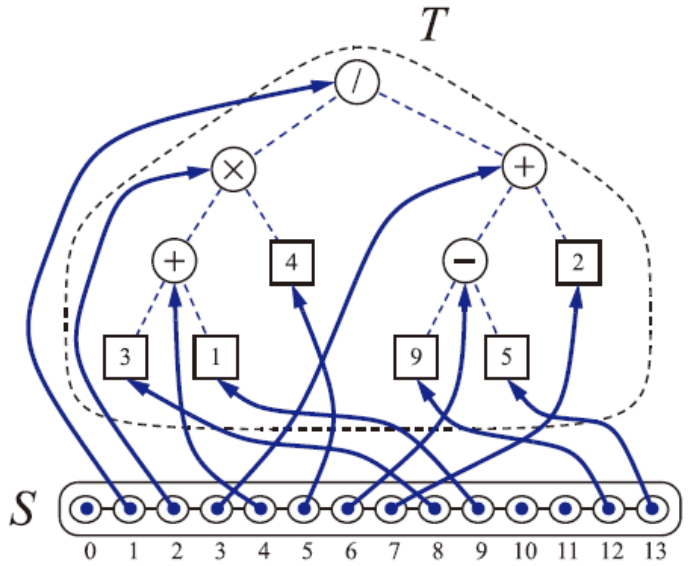


Figure 7.17: Representation of a binary tree T by means of a vector S.

Operation	*Time*
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

Binary Trees Traversals

- **Preorder Traversal**

Algorithm binaryPreorder(T, p):
  perform the "visit" action for node p
  if p is an internal node **then**
    binaryPreorder(T, p.left()) {recursively traverse left subtree}
    binaryPreorder(T, p.right()) {recursively traverse right subtree}

Code Fragment 7.24: Algorithm binaryPreorder, which performs the preorder traversal of the subtree of a binary tree T rooted at node p.

- **Postorder Traversal**

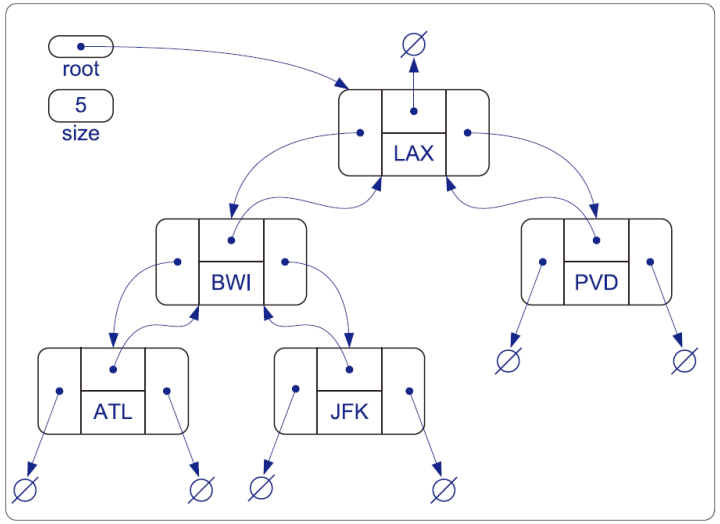
Algorithm binaryPostorder(T, p):
  if p is an internal node **then**
    binaryPostorder(T, p.left()) {recursively traverse left subtree}
    binaryPostorder(T, p.right()) {recursively traverse right subtree}
  perform the "visit" action for the node p

Code Fragment 7.25: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node p.

Algorithm postorder(T, p):
  for each child q of p **do**
    recursively traverse the subtree rooted at q by calling postorder(T, q)
  perform the "visit" action for node p

Code Fragment 7.12: Algorithm postorder for performing the postorder traversal of the subtree of a tree T rooted at a node p.

Tree Traversal Algorithm



Binary Trees Traversals

- **Postorder Traversal**
  - Can be applied to various "*bottom-up*" evaluation problems

Algorithm evaluateExpression(T, p):
  if p is an internal node **then**
    $x \leftarrow$ evaluateExpression$(T, p.\text{left}())$
    $y \leftarrow$ evaluateExpression$(T, p.\text{right}())$
    Let $\circ$ be the operator associated with p
    return $x \circ y$
  else
    return the value stored at p

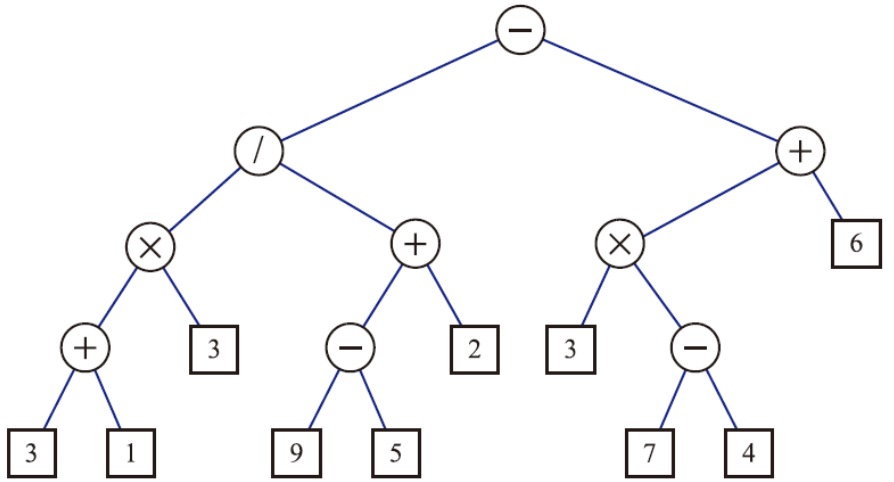


Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Code Fragment 7.26: Algorithm evaluateExpression for evaluating the expression represented by the subtree of an arithmetic-expression tree T rooted at node p.

Binary Trees Traversals

- **Inorder Traversal**
  - Visit a node *between* the recursive traversals of its left and right subtrees
  - Traverse from the left to the right

```
Algorithm inorder(T, p):
  if p is an internal node then
    inorder(T, p.left())          {recursively traverse left subtree}
  perform the "visit" action for node p
  if p is an internal node then
    inorder(T, p.right())         {recursively traverse right subtree}
```

Code Fragment 7.27: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node p.

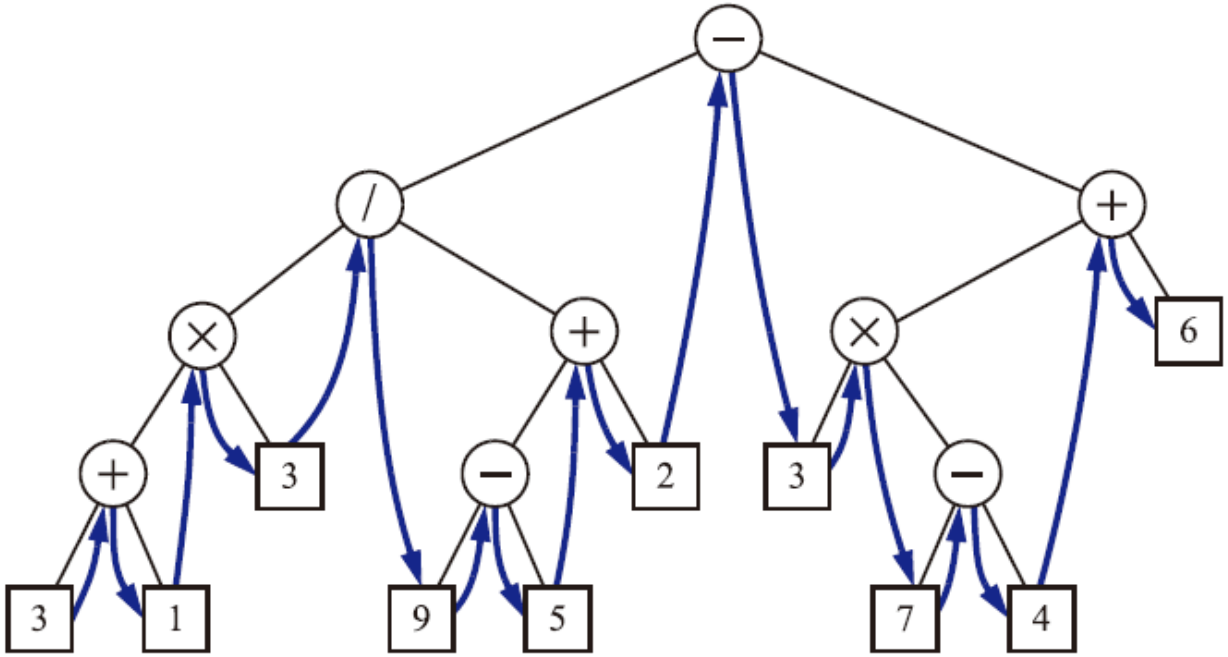


Figure 7.18: Inorder traversal of a binary tree.

Binary Trees Traversals

- **Example: Drawing a Tree using Inorder Traversal**
  - Two rules to assign (x, y) coordinates:
    - $x(p)$ is the number of nodes visited before p in the inorder traversal of T.
    - $y(p)$ is the depth of p in T.

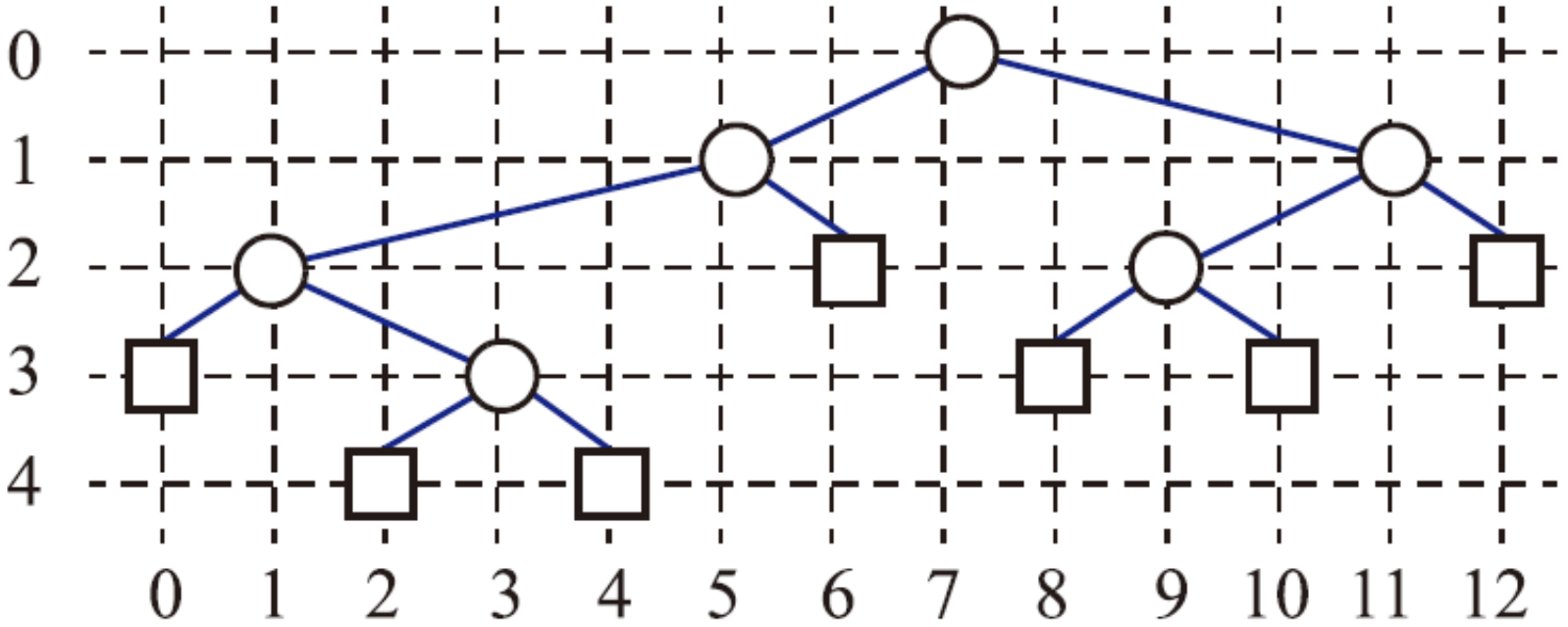


Figure 7.20: The inorder drawing algorithm for a binary tree.

Binary Trees Traversals

- **Binary Search Trees**
  - Let S be a set whose elements have an order relation
  - A *binary search tree* for S is a proper binary tree T such that:
    - Each internal node p of T stores an element of S, denoted with $x(p)$
    - For each internal node p of T, the elements stored in the left subtree of p are less than or equal to $x(p)$ and the elements stored in the right subtree of p are greater than or equal to $x(p)$
    - The external nodes of T do not store any element
  - Search time is proportional to the height h of T

$$\log(n+1) - 1 \leq h \leq n - 1 \;\rightarrow\; O(\log(n)) \sim \Omega(n)$$

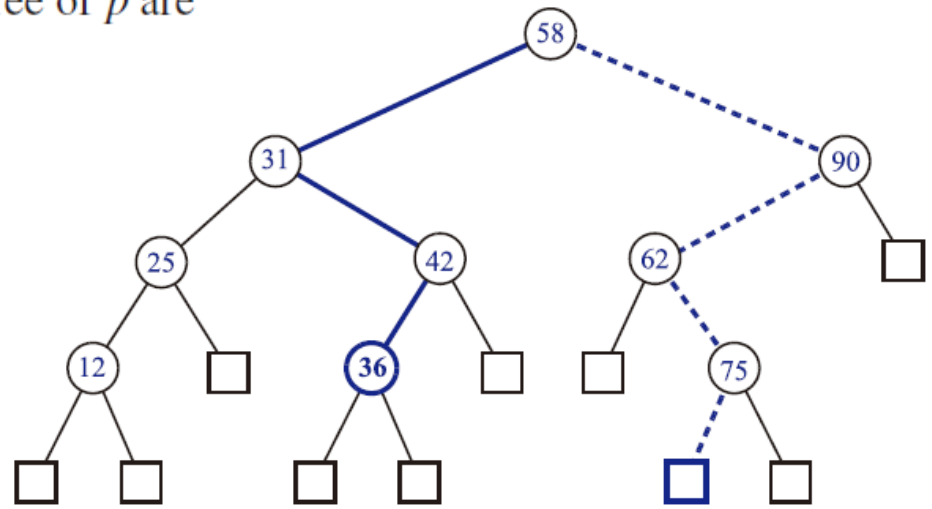


Figure 7.19: A binary search tree storing integers. The blue solid path is traversed when searching (successfully) for 36. The blue dashed path is traversed when searching (unsuccessfully) for 70.

Binary Trees Traversals

- **Euler Tour Traversal**
  - Generalization of preorder, inorder, and postorder traversal algorithms
  - Informally defined as a *walk* around a tree T
  - Regard *edges* of T as *walls on the walker's left*
  - Each node p of T is encountered three times by the Euler tour:
    - "On the left" (before the Euler tour of p's left subtree)
    - "From below" (between the Euler tours of p's two subtrees)
    - "On the right" (after the Euler tour of p's right subtree)

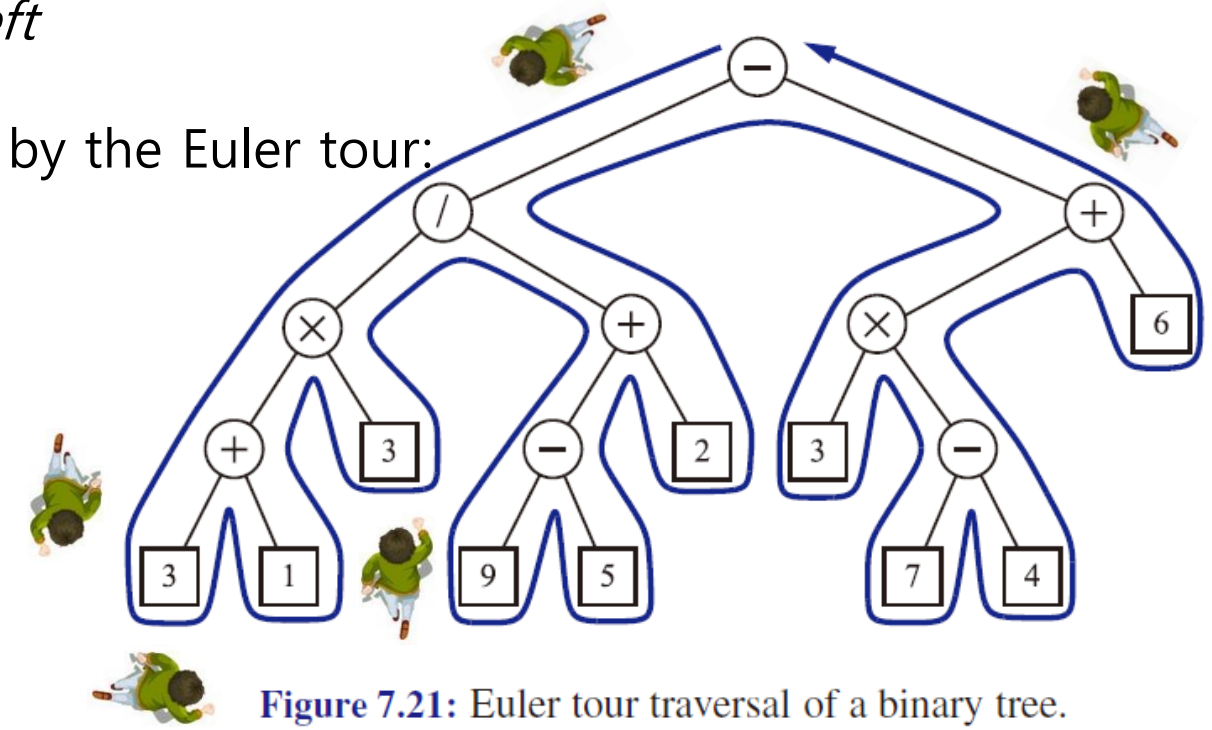


Figure 7.21: Euler tour traversal of a binary tree.

https://www.vecteezy.com/vector-art/292391-top-view-of-people-doing-activities

Binary Trees Traversals

- **Euler Tour Traversal**

Algorithm eulerTour(T, p): $O(n)$

perform the action for visiting node p on the left
if p is an internal node **then**
  recursively tour the left subtree of p by calling eulerTour$(T, p.\text{left}())$
perform the action for visiting node p from below
if p is an internal node **then**
  recursively tour the right subtree of p by calling eulerTour$(T, p.\text{right}())$
perform the action for visiting node p on the right

Code Fragment 7.28: Algorithm eulerTour for computing the Euler tour traversal of the subtree of a binary tree T rooted at a node p.

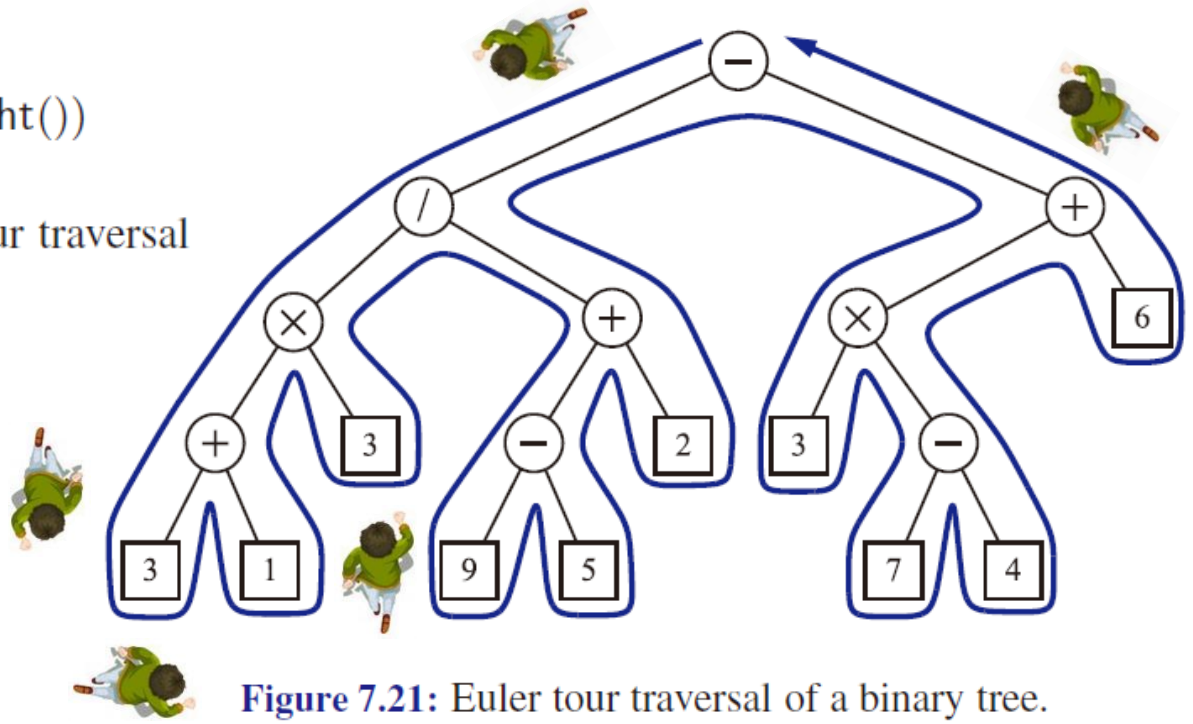


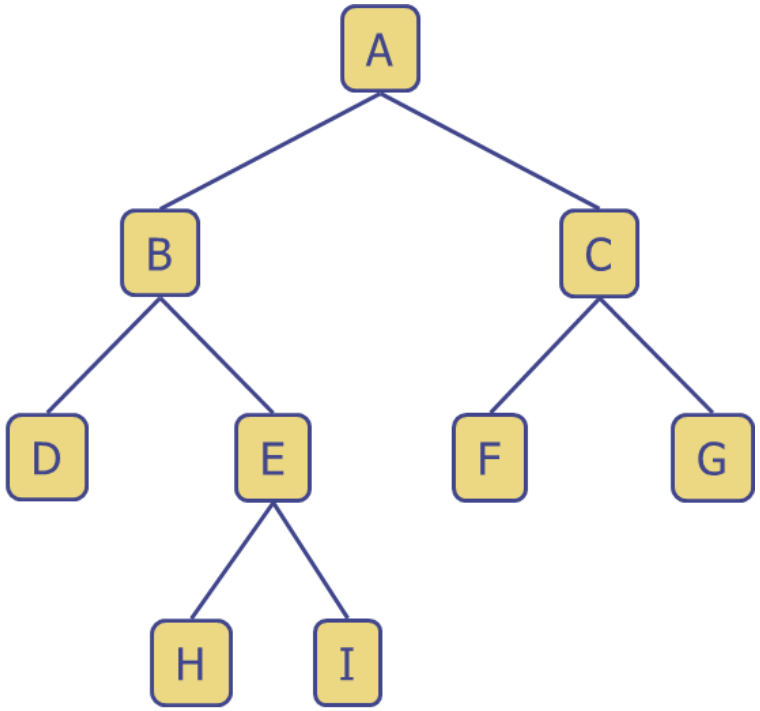
Figure 7.21: Euler tour traversal of a binary tree.

https://www.vecteezy.com/vector-art/292391-top-view-of-people-doing-activities

Binary Trees Traversals

- **Example: The Number of Descendants**
  - Wish to compute the number of descendants of each node p in a n node binary tree T
    1. Initialize a counter to 0
    2. Each node holds counter values from the left and the right
    3. Increment the counter with each visit on the left
    4. Use counter values from the left and the right to get the number of descendants

$O(n)$



Binary Trees Traversals

- **Example: Printing Arithmetic Expression**

  - "On the left" action: if the node is internal, print "("
  - "From below" action: print the value or operator stored at the node
  - "On the right" action: if the node is internal, print ")"

```
Algorithm printExpression(T, p):
    if p.isExternal() then
        print the value stored at p
    else
        print "("
        printExpression(T, p.left())
        print the operator stored at p
        printExpression(T, p.right())
        print ")"
```

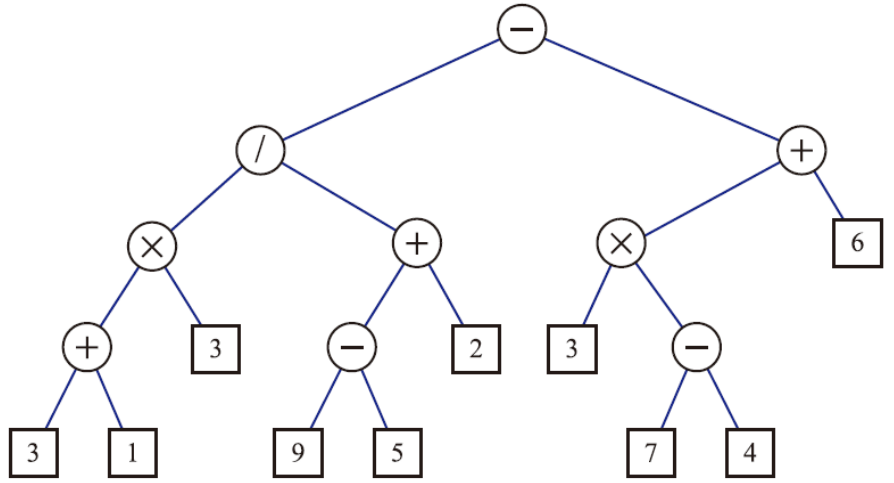


Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Representing General Trees with Binary Trees

- Transform a general tree T into a binary tree T'
- Assume that a general tree T is ordered or that it has been arbitrarily ordered
  - For each node u of T, there is an internal node u' of T' associated with u
  - If u is an external node of T and does not have a sibling immediately following it, then the children of u' in T' are external nodes
  - If u is an internal node of T and v is the first child of u in T, then v' is the left child of u' in T
  - If node v has a sibling w immediately following it, then w' is the right child of v' in T'
- External nodes of T' serve only as place holders
- Takes each set of siblings $\{v_1, v_2, \ldots, v_k\}$ in T with parent v and replaces it with a chain of right children rooted at v_1, then it becomes the left child of v