Tree & Binary Trees (3)

College of Computer Science, CQU

Outline

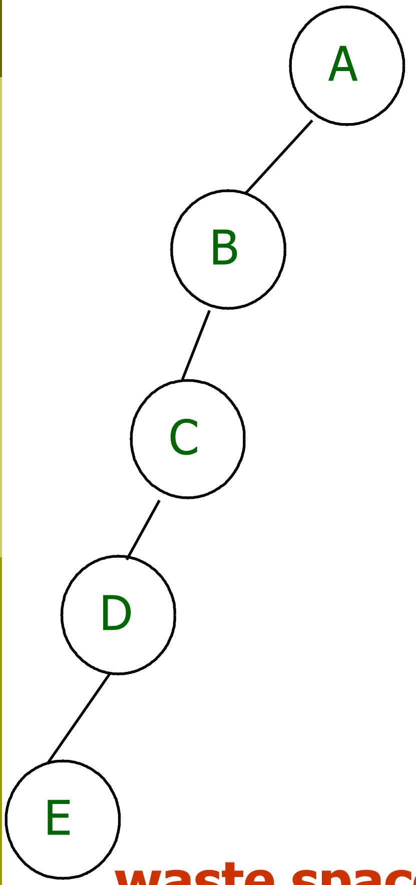
- Binary Tree Implementations
  - Array-Based Implementation
  - Pointer-Based Implementation
- General Tree Traversals
- General Tree Implementations
- Converting forest to binary tree

Binary Tree Implementations

- There are two implementations:
  - array-based implementation
  - pointer-based implementation

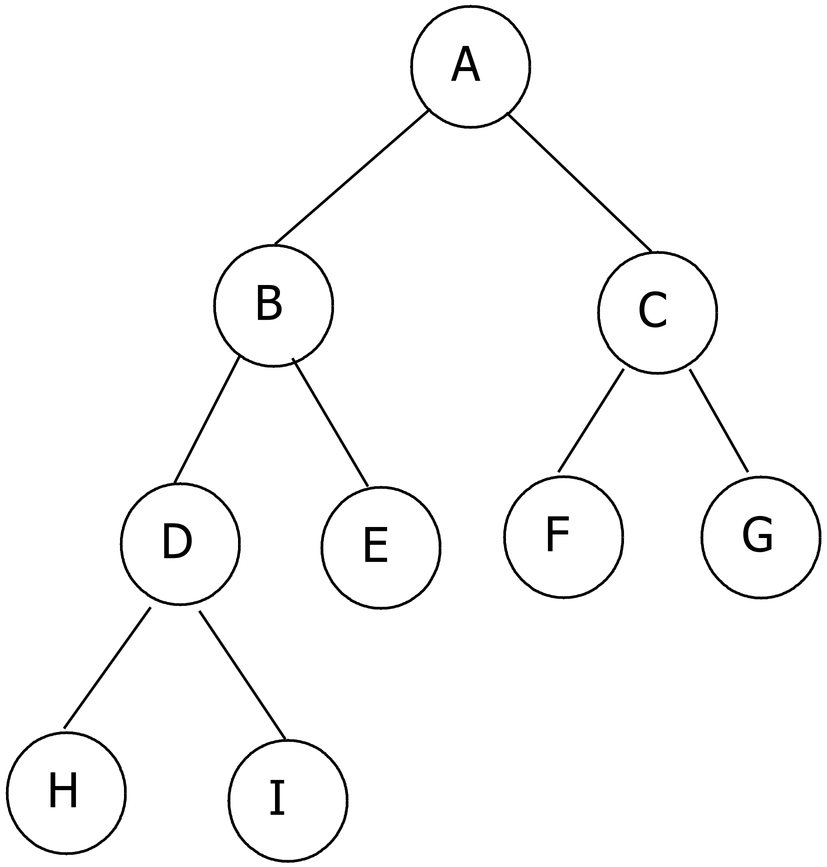
Array-Based Implementation

- How to use an array representation for binary trees?



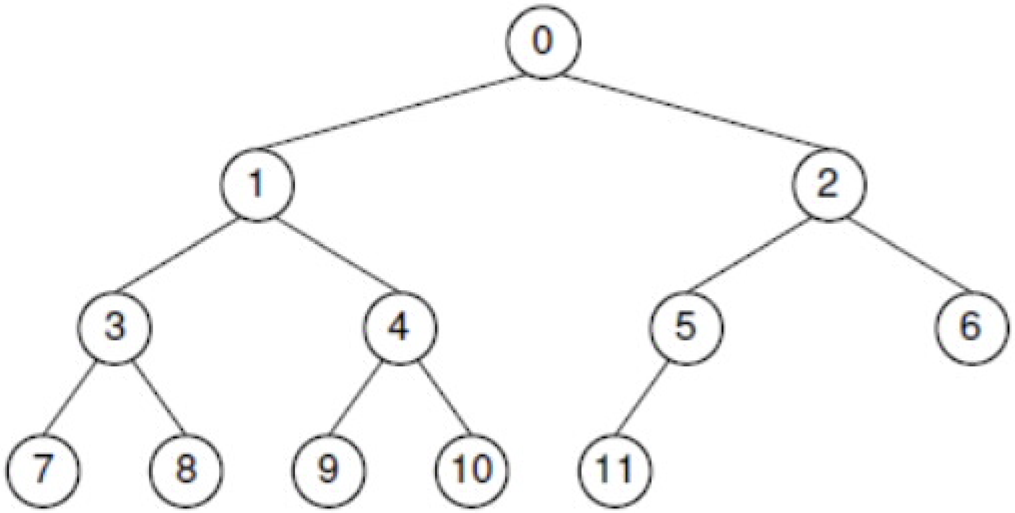
[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

waste space



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Array-Based Implementation for Complete Binary Trees



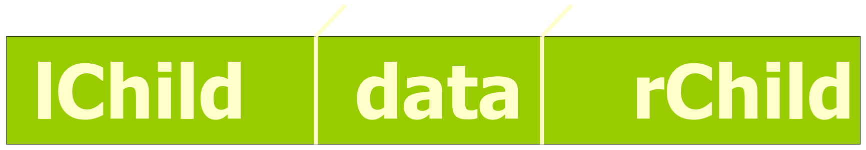
Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

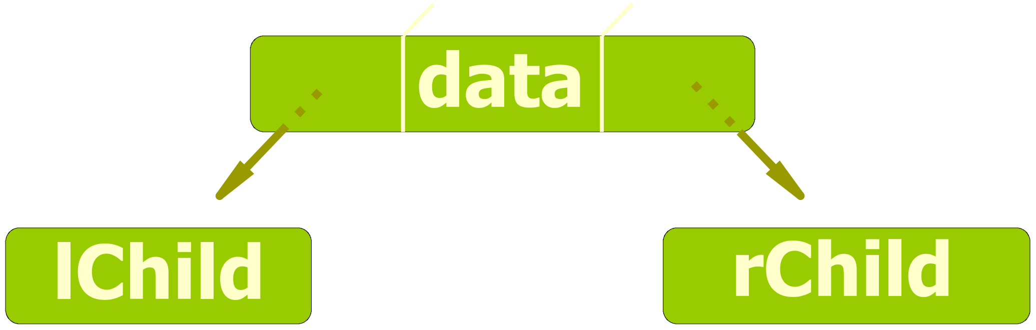
Array-Based Implementation for Complete Binary Trees

- The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to n-1.
  - Parent(r) = $(r - 1) / 2$ if $r \neq 0$.
  - Left child(r) = $2r + 1$ if $2r + 1 < n$.
  - Right child(r) = $2r + 2$ if $2r + 2 < n$.
  - Left sibling(r) = $r - 1$ if r is even.
  - Right sibling(r) = $r + 1$ if r is odd and $r + 1 < n$.

Pointer-Based Implementation

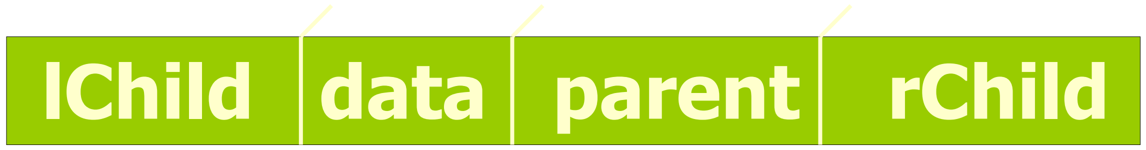
- All binary tree nodes have two children.
- The most common node implementation includes a value field and pointers to the two children

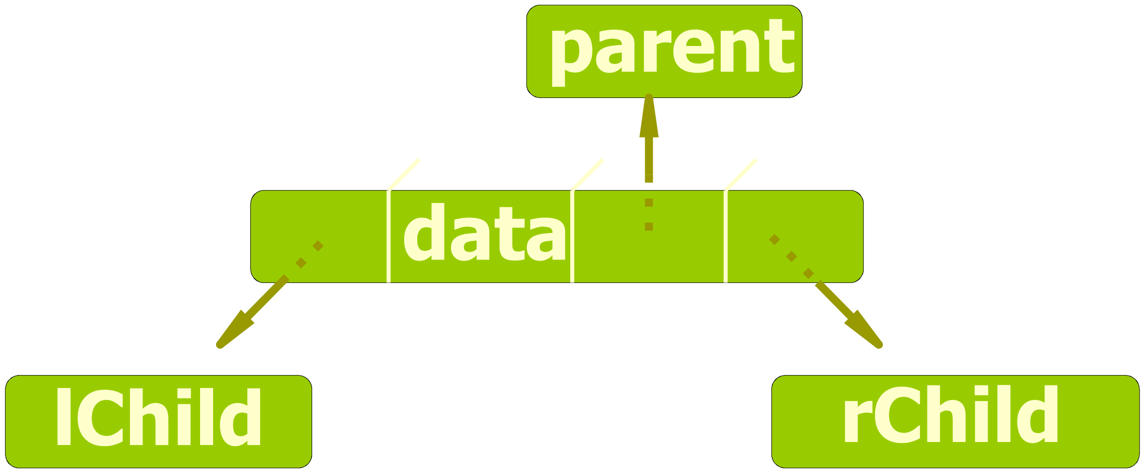




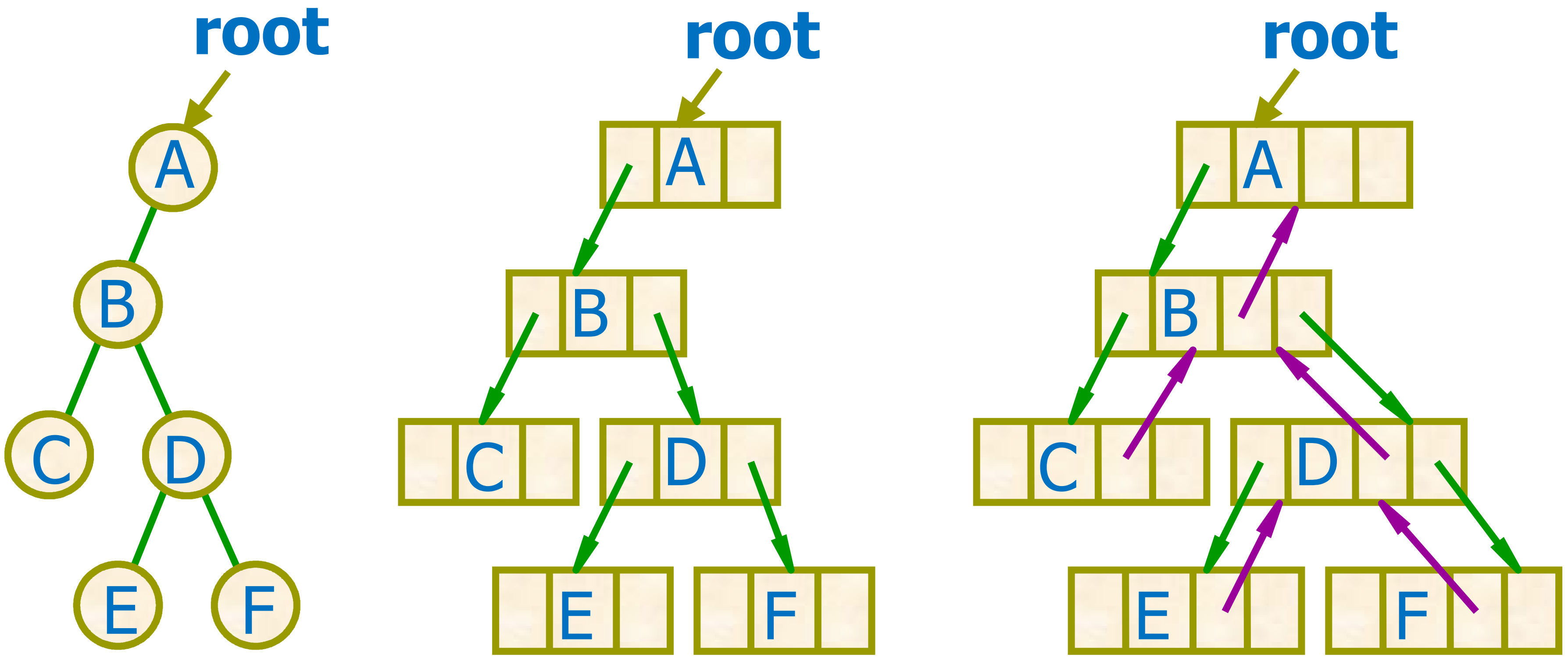
Pointer-Based Implementation

- All binary tree nodes have two children and one parent, except root.
- An alternate node implementation includes a value field and pointers to the two children and one parent.





Example



Pointer-Based Node Implementation

```
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
    Key k; // The node’s key
    E it; // The node’s value
    BSTNode* lc; // Pointer to left child
    BSTNode* rc; // Pointer to right child

public:
    // Two constructors -- with and without initial values
    BSTNode() { lc = rc = NULL; }
    BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
        { k = K; it = e; lc = l; rc = r; }
    ~BSTNode() {} // Destructor
```

Pointer-Based Node Implementation

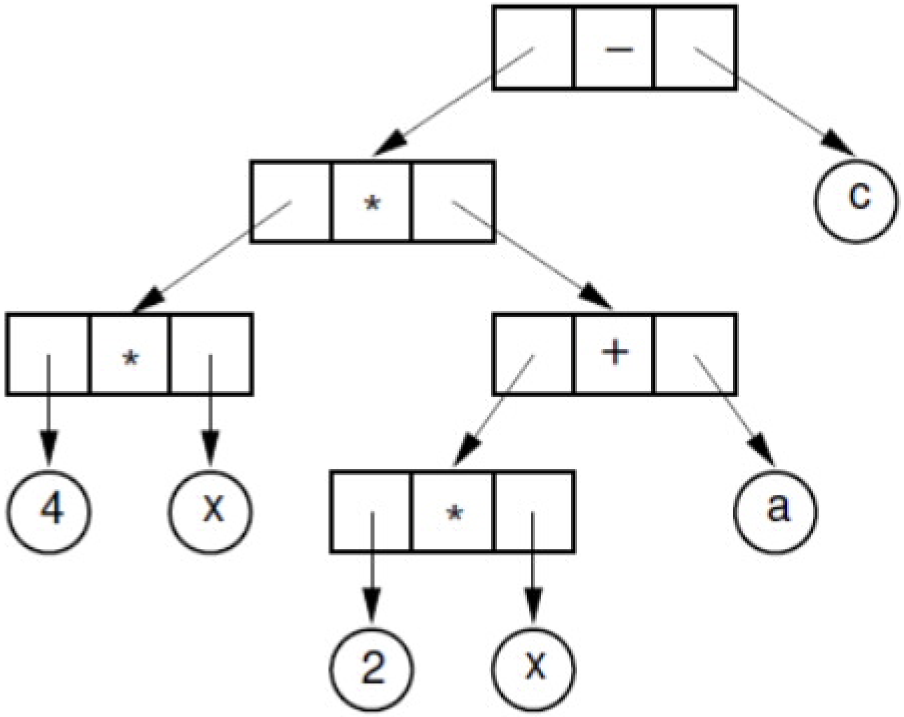
```
  // Functions to set and return the value and key
  E& element() { return it; }
  void setElement(const E& e) { it = e; }
  Key& key() { return k; }
  void setKey(const Key& K) { k = K; }

  // Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

Distinguish leaf and internal nodes

- Why?
  - Some applications require data values only for the leaves.
  - Other applications require one type of value for the leaves and another for the internal nodes.
- By definition, only internal nodes have non-empty children. It can save space to have separate implementations for internal and leaf nodes.

Example: Expression Tree

- Expression : 4x(2x + a) - c



Internal nodes store operators, while the leaves store operands.

Approach

- **Approach 1**: to use class inheritance
    - A base class can be declared for binary tree nodes in general, with subclasses defined for the internal and leaf nodes.
- **Approach 2**: to use the composite design pattern
    - using a virtual base class and separate node classes for the two types.

Approach1
--leaf node representation

```
// Node implementation with simple inheritance
class VarBinNode { // Node abstract base class
public:
    virtual ˜VarBinNode() {}
    virtual bool isLeaf() = 0; // Subclasses must implement
};

class LeafNode : public VarBinNode { // Leaf node
private:
    Operand var; // Operand value

public:
    LeafNode(const Operand& val) { var = val; } // Constructor
    bool isLeaf() { return true; } // Version for LeafNode
    Operand value() { return var; } // Return node value
};
```

Approach1

--internal node representation

```
class IntlNode : public VarBinNode { // Internal node
private:
    VarBinNode* left; // Left child
    VarBinNode* right; // Right child
    Operator opx; // Operator value

public:
    IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
        { opx = op; left = l; right = r; } // Constructor
    bool isLeaf() { return false; } // Version for IntlNode
    VarBinNode* leftchild() { return left; } // Left child
    VarBinNode* rightchild() { return right; } // Right child
    Operator value() { return opx; } // Value
};
```

Approach1--Function traverse

```
void traverse(VarBinNode *root) { // Preorder traversal
    if (root == NULL) return; // Nothing to visit
    if (root->isLeaf()) // Do leaf node
        cout << "Leaf: " << ((LeafNode *)root)->value() << endl;
    else { // Do internal node
        cout << "Internal: "
               << ((IntlNode *)root)->value() << endl;
        traverse(((IntlNode *)root)->leftchild());
        traverse(((IntlNode *)root)->rightchild());
    }
}
```

Approach2
--leaf node implementation

```
// Node implementation with the composite design pattern
class VarBinNode { // Node abstract base class
public:
    virtual ˜VarBinNode() {} // Generic destructor
    virtual bool isLeaf() = 0;
    virtual void traverse() = 0;
};
class LeafNode : public VarBinNode { // Leaf node
private:
    Operand var; // Operand value
public:
    LeafNode(const Operand& val) { var = val; } // Constructor
    bool isLeaf() { return true; } // isLeaf for Leafnode
    Operand value() { return var; } // Return node value
    void traverse() { cout << "Leaf: " << value() << endl; }
};
```

Approach2 --internal node implementation

```
class IntlNode : public VarBinNode { // Internal node
private:
    VarBinNode* lc; // Left child
    VarBinNode* rc; // Right child
    Operator opx; // Operator value
public:
    IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
        { opx = op; lc = l; rc = r; } // Constructor
    bool isLeaf() { return false; } // isLeaf for IntlNode
    VarBinNode* left() { return lc; } // Left child
    VarBinNode* right() { return rc; } // Right child
    Operator value() { return opx; } // Value
    void traverse() { // Traversal behavior for internal nodes
        cout << "Internal: " << value() << endl;
        if (left() != NULL) left()->traverse();
        if (right() != NULL) right()->traverse();
    }
};
```

General Tree Traversals

- For general trees, preorder and postorder traversals are defined with meanings similar to their binary tree counterparts.
- Preorder traversal of a general tree first visits the root of the tree, then performs a preorder traversal of each subtree from left to right.
- A postorder traversal of a general tree performs a postorder traversal of the root's subtrees from left to right, then visits the root.
- Inorder traversals are generally not useful with general trees.

General Tree Traversals

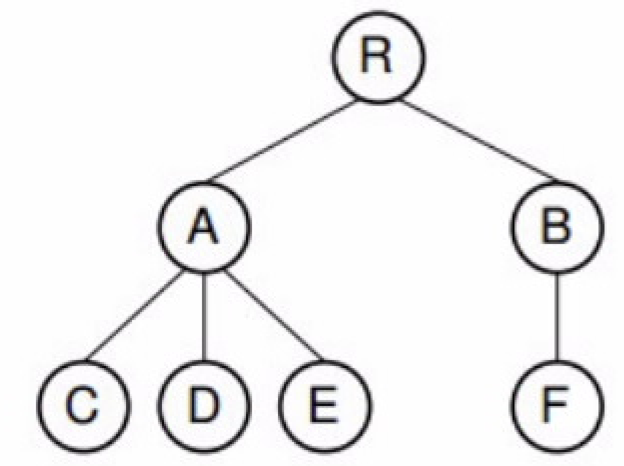
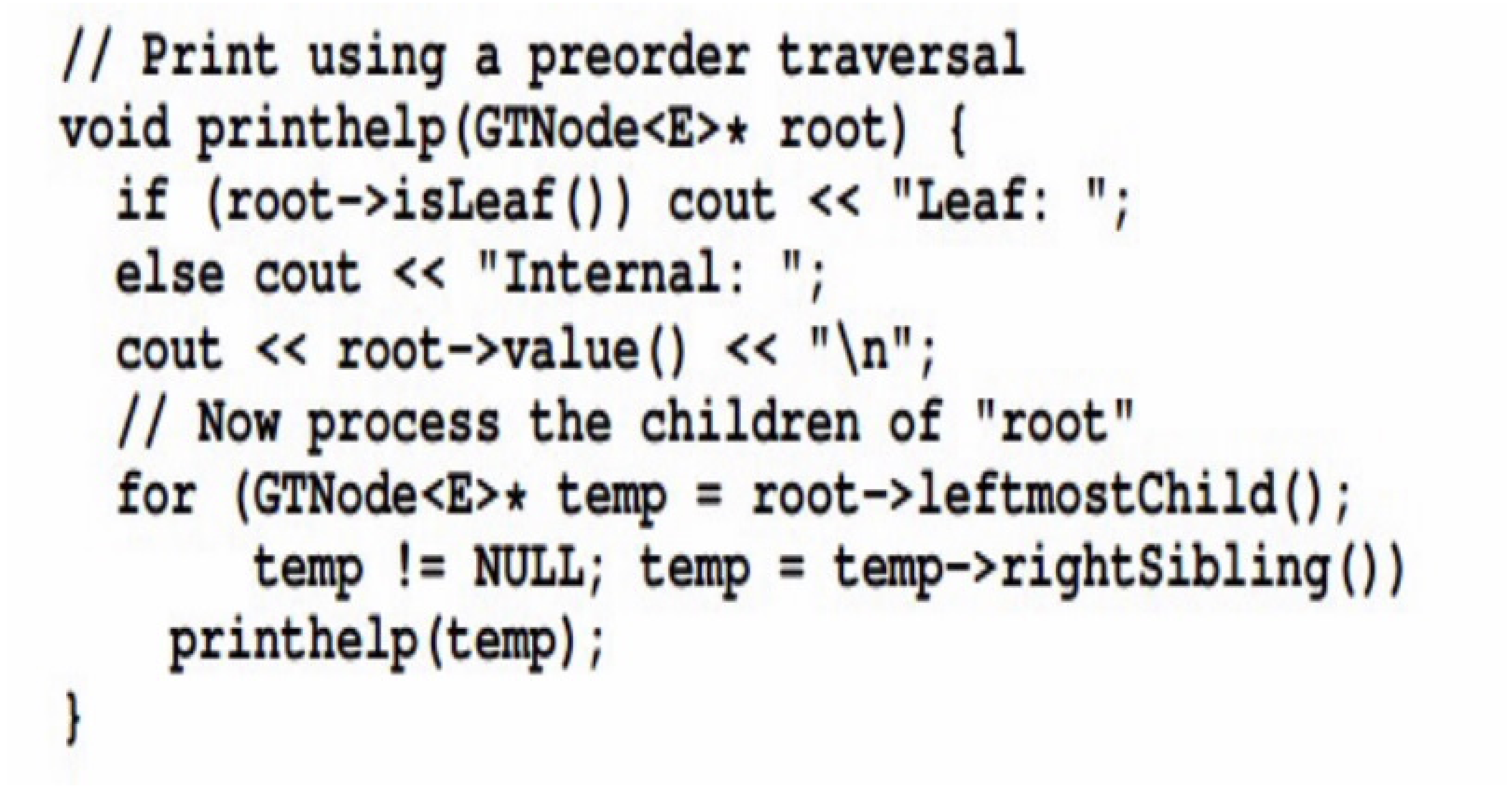


Figure 6.3 An example of a general tree.

- preorder traversal : RACDEBF.
- postorder : CDEAFBR.

General Tree Traversals



```
// Print using a preorder traversal
void printhelp(GTNode<E>* root) {
  if (root->isLeaf()) cout << "Leaf: ";
  else cout << "Internal: ";
  cout << root->value() << "\n";
  // Now process the children of "root"
  for (GTNode<E>* temp = root->leftmostChild();
           temp != NULL; temp = temp->rightSibling())
    printhelp(temp);
}
```

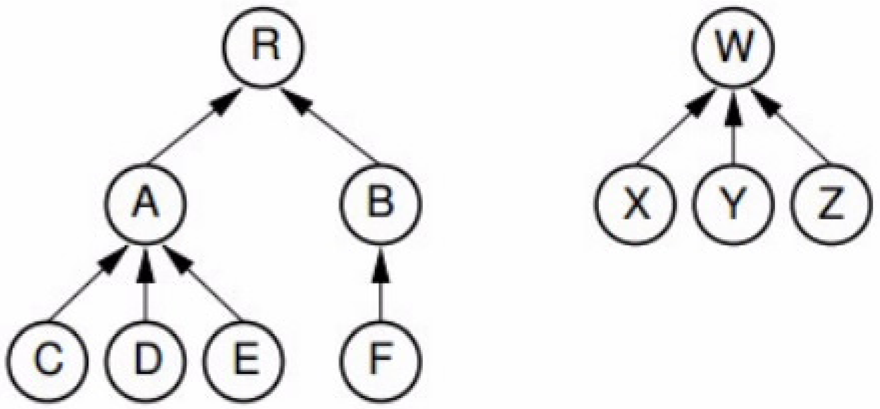
General Tree Implementations

- The simplest general tree implementation is to store for each node only a pointer to that node’s parent. We will call this the parent pointer implementation.
- The parent pointer implementation stores precisely the information

required to answer the following, useful question: “Given two nodes, are they in the same tree?”

To answer the question, we need only follow the series of parent pointers from each node to its respective root. If both nodes reach the same root, then they must be in the same tree. If the roots are different, then the two nodes are not in the same tree.

Parent Pointer Implementations



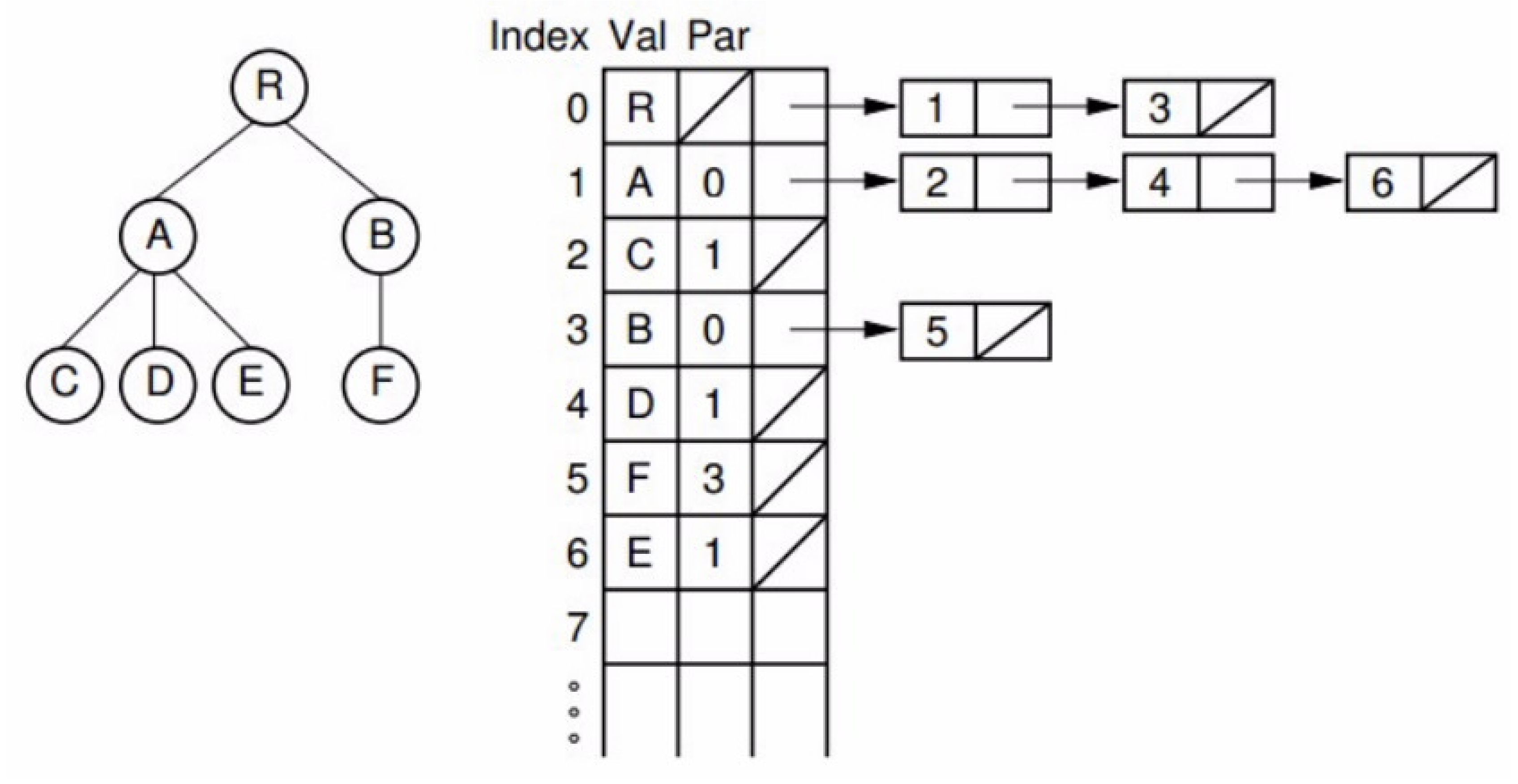
Parent's Index		0	0	1	1	1	2		7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

- This implementation is not general purpose, because it is inadequate for such important operations as finding the leftmost child or the right sibling for a node.

List of Children Implementations

- It simply stores with each internal node a linked list of its children.
- Each node contains a value, a pointer (or index) to its parent, and a pointer to a linked list of the node's children, stored in order from left to right.
- Each linked list element contains a pointer to one child.
- Thus, the leftmost child of a node can be found directly because it is the first element in the linked list.
- However, to find the right sibling for a node is more difficult.

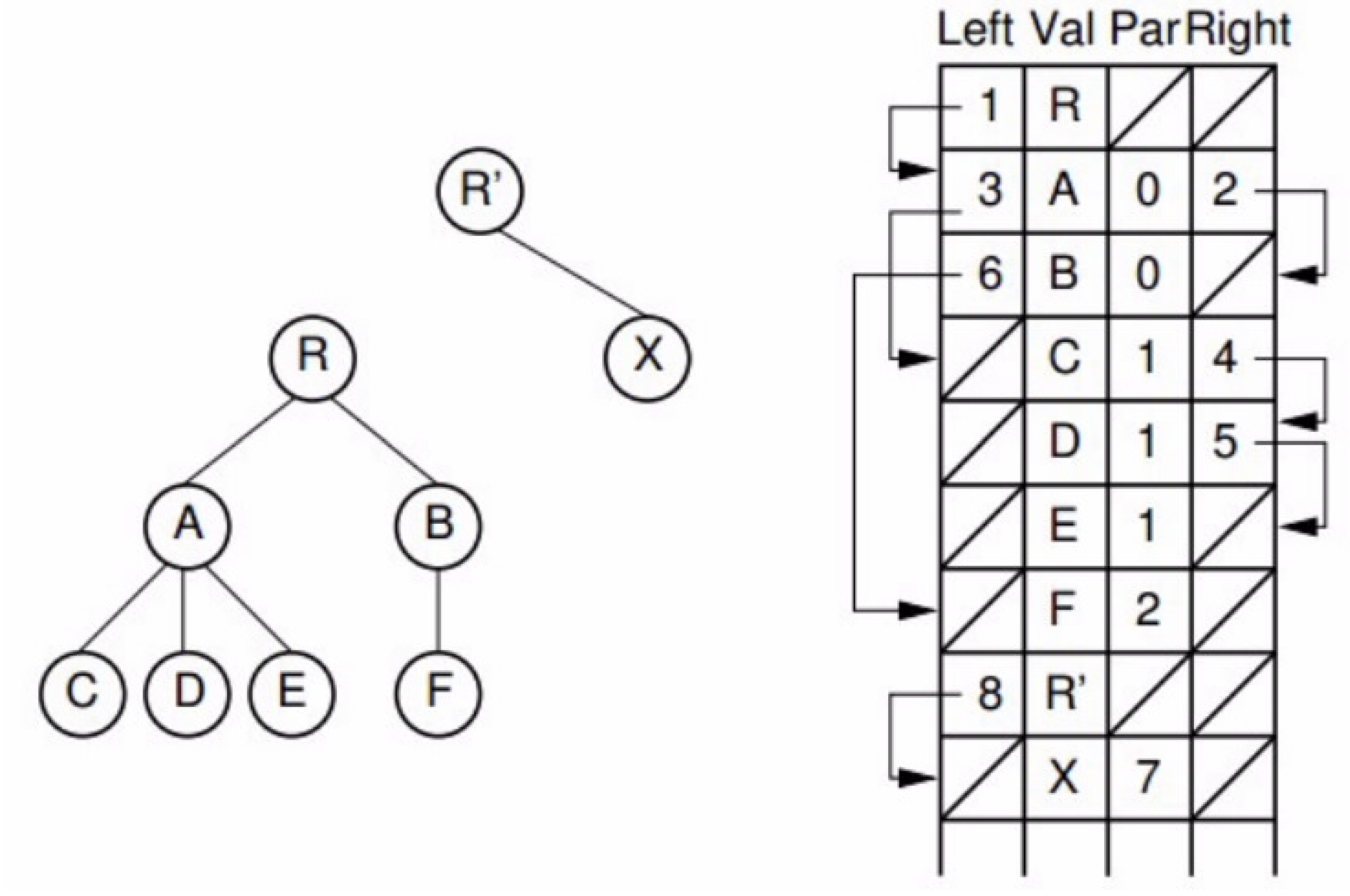
List of Children Implementations



Left-Child/Right-Sibling Implementation

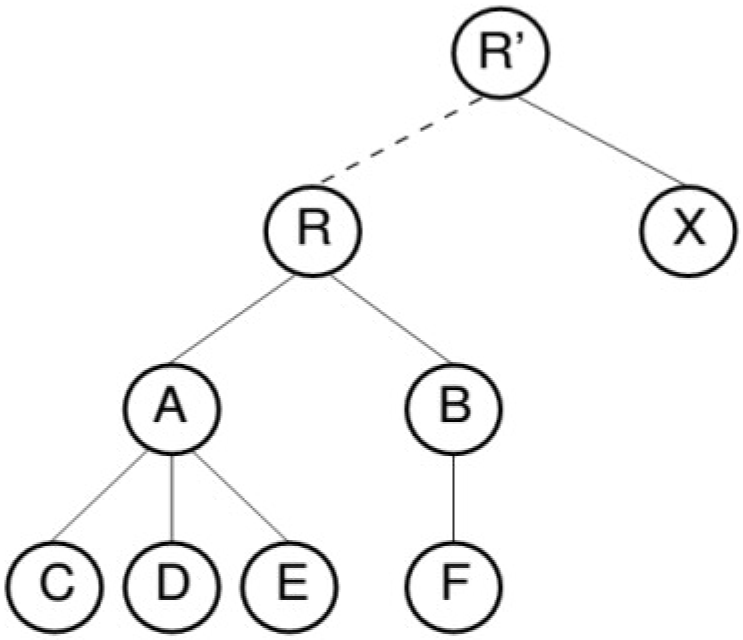
- Each node stores its value and pointers to its parent, leftmost child, and right sibling.
- Thus, each of the basic ADT operations can be implemented by reading a value directly from the node.
- If two trees are stored within the same node array, then adding one as the subtree of the other simply requires setting three pointers
- This implementation is more space efficient than the “list of children” implementation, and each node requires a fixed amount of space in the node array

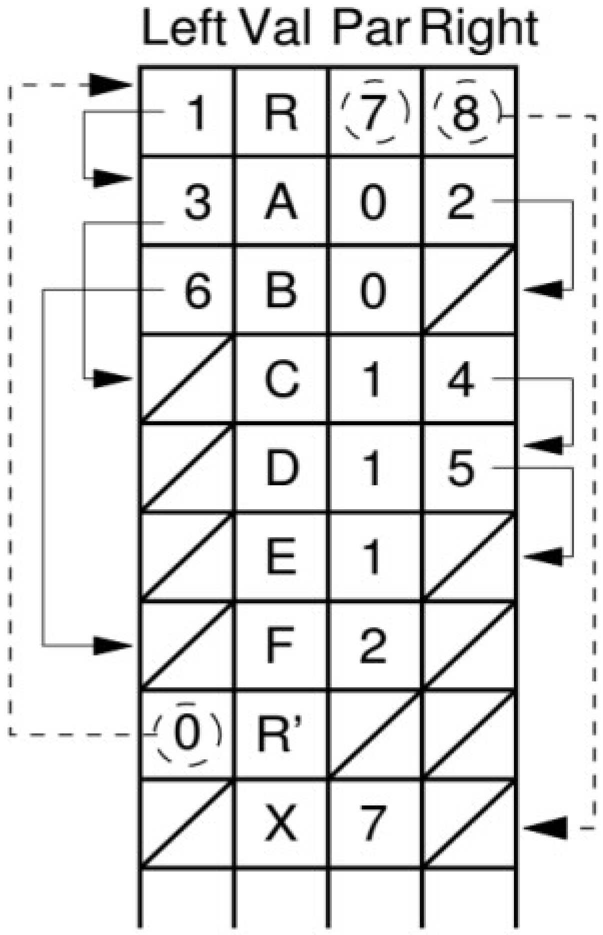
Left-Child/Right-Sibling Implementation



Left	Val	Par	Right
1	R	/	/
3	A	0	2
6	B	0	/
/	C	1	4
/	D	1	5
/	E	1	/
/	F	2	/
8	R'	/	/
/	X	7	/

Left-Child/Right-Sibling Implementation





Left	Val	Par	Right
1	R	7	8
3	A	0	2
6	B	0	
	C	1	4
	D	1	5
	E	1	
	F	2	
0	R'		
	X	7	

Dynamic Node Implementations

- The two general tree implementations just described use an array to store the collection of nodes. In contrast, our standard implementation for binary trees stores each node as a separate dynamic object containing its value and pointers to its two children.
- Unfortunately, nodes of a general tree can have any number of children, and this number may change during the life of the node.
- A general tree node implementation must support these properties.
- One solution is simply to limit the number of children permitted for any node and allocate pointers for exactly that number of children.

Dynamic Node Implementations

- There are two major objections to this.
  - First, it places an undesirable limit on the number of children, which makes certain trees un-representable by this implementation.
  - Second, this might be extremely wasteful of space
- The alternative is to allocate variable space for each node. There are two basic approaches. One is to allocate an array of child pointers as part of the node. In essence, each node stores an array-based list of child pointers.

Dynamic Node Implementations

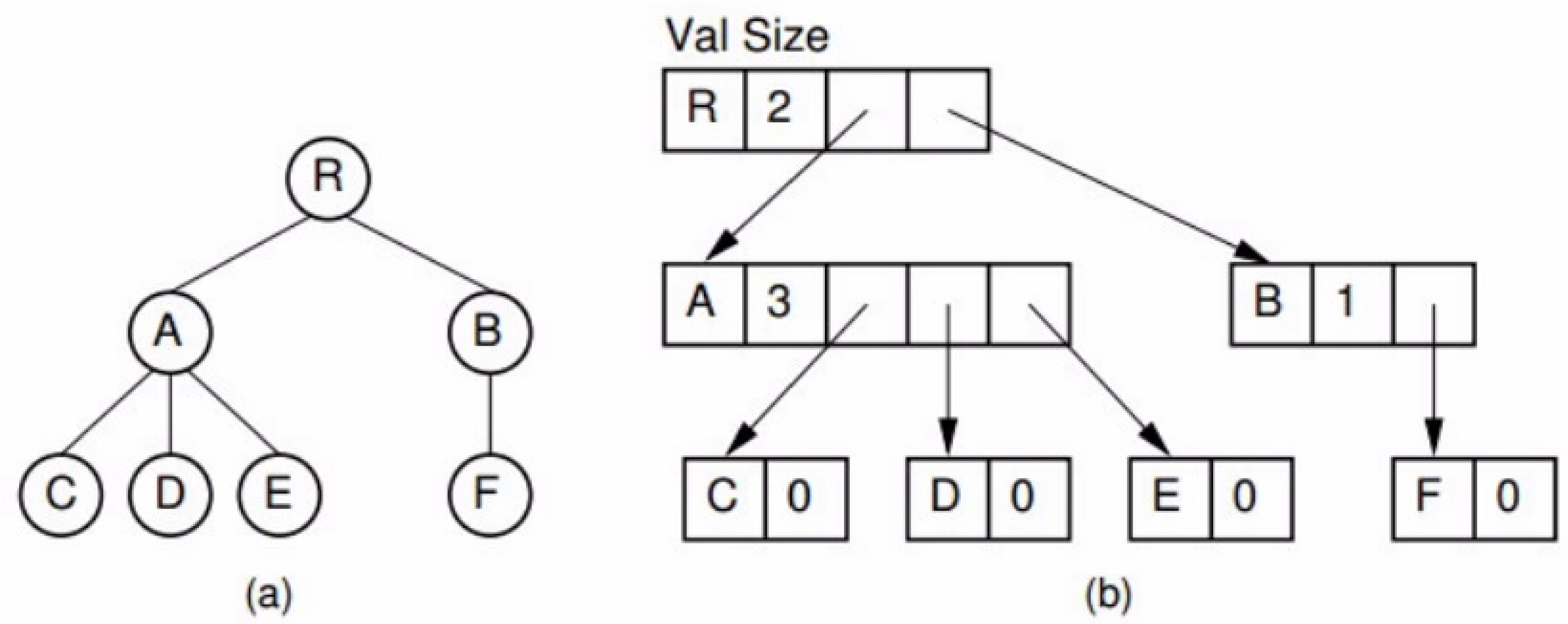


Figure 6.12 A dynamic general tree representation with fixed-size arrays for the child pointers. (a) The general tree. (b) The tree representation. For each node, the first field stores the node value while the second field stores the size of the child pointer array.

Dynamic Node Implementations

- This approach assumes that the number of children is known when the node is created, which is true for some applications but not for others.
- It also works best if the number of children does not change. If the number of children does change (especially if it increases), then some special recovery mechanism must be provided to support a change in the size of the child pointer array.
- One possibility is to allocate a new node of the correct size from free store and return the old copy of the node to free store for later reuse. This works especially well in a language with built-in garbage collection such as Java.

Dynamic Node Implementations

- Another approach that is more flexible, but which requires more space, is to store a linked list of child pointers with each node .
- This implementation dynamically allocated nodes rather than storing the nodes in an array.

Dynamic Node Implementations

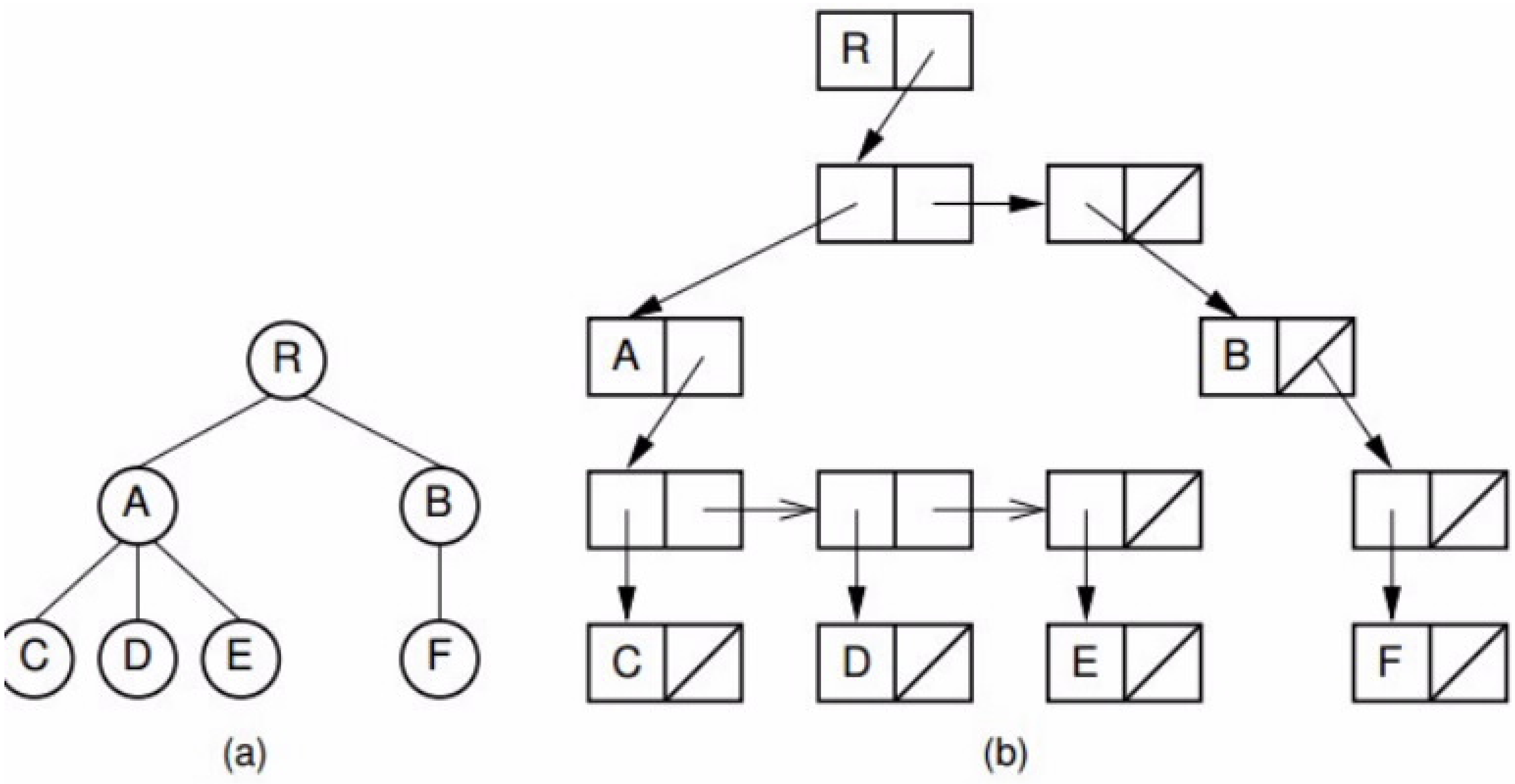
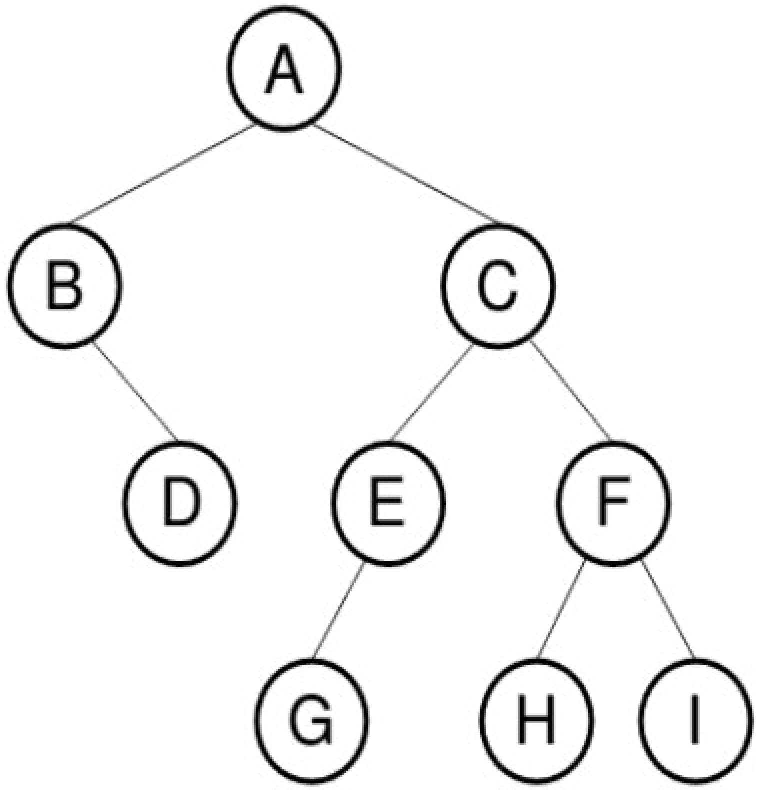


Figure 6.13 A dynamic general tree representation with linked lists of child pointers. (a) The general tree. (b) The tree representation.

Sequential Tree Implementations

- Sequential tree implementation stores a series of node values with the minimum information needed to reconstruct the tree structure.
- This approach has the advantage of saving space because no pointers are stored. It has the disadvantage that accessing any node in the tree require sequentially processing all nodes that appear before it in the node list. In other words, node access must start at the beginning of the node list, processing nodes sequentially in whatever order they are stored until the desired node is reached.
- Thus, one primary virtue of the other implementations discussed in this section is lost: efficient access to arbitrary nodes in the tree.

Sequential Tree Implementations



the corresponding sequential representation would be as follows (assuming that '/' stands for NULL):

AB/D//CEG///FH//I//

Sequential Tree Implementations

- Sequential tree implementations are ideal for archiving trees on disk for later use because they save space, and the tree structure can be reconstructed as needed for later processing.
- Sequential tree implementations can be used to serialize a tree structure.
- Serialization is the process of storing an object as a series of bytes, typically so that the data structure can be transmitted between computers.
- This capability is important when using data structures in a distributed processing environment.

Converting forest to binary tree

- The “left-child/right-sibling” implementation stores a fixed number of pointers with each node. This can be readily adapted to a dynamic implementation. In essence, we substitute a binary tree for a general tree. Each node of the “left-child/right-sibling” implementation points to two “children” in a new binary tree structure.
- The left child of this new structure is the node’s first child in the general tree. The right child is the node’s right sibling. We can easily extend this conversion to a forest of general trees, because the roots of the trees can be considered siblings.

Converting forest to binary tree

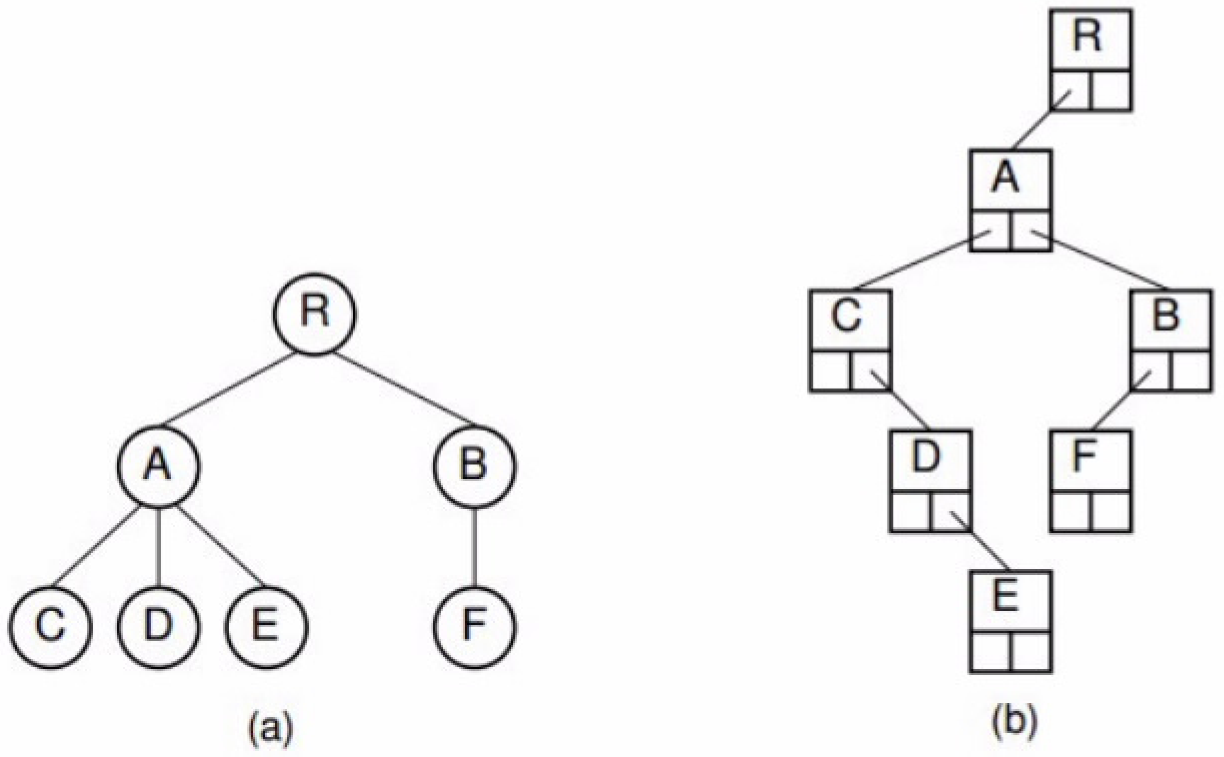


Figure 6.15 A general tree converted to the dynamic "left-child/right-sibling" representation. Compared to the representation of Figure 6.13, this representation requires less space.

Converting forest to binary tree

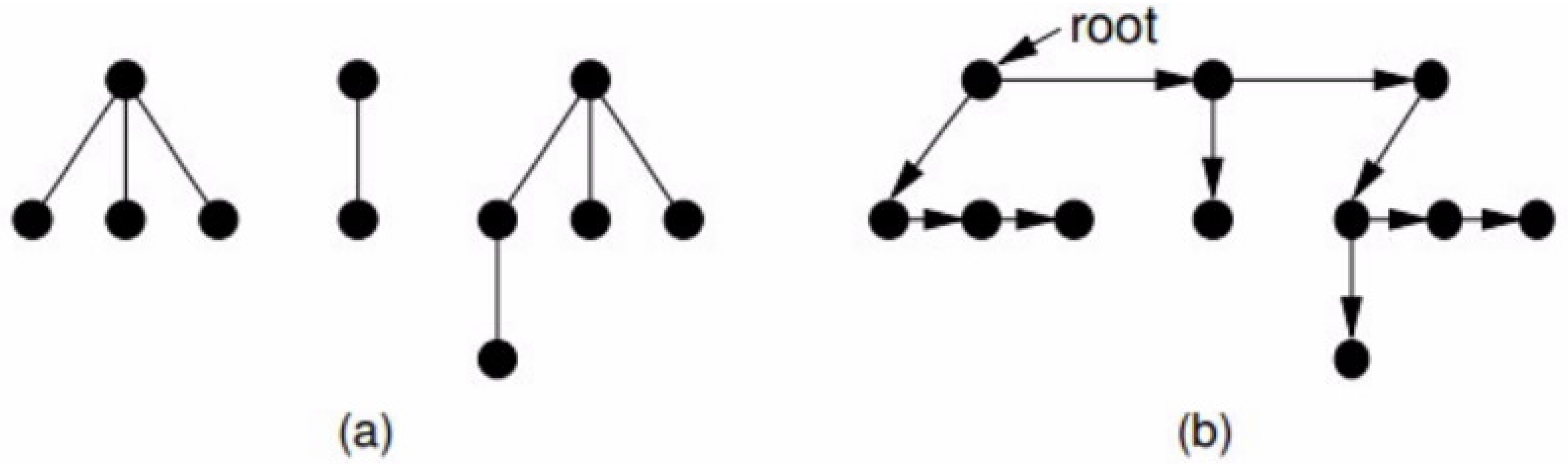


Figure 6.14 Converting from a forest of general trees to a single binary tree. Each node stores pointers to its left child and right sibling. The tree roots are assumed to be siblings for the purpose of converting.

Converting forest to binary tree

- Here we simply include links from each node to its right sibling and remove links to all children except the leftmost child.
- Figure 6.15 shows how this might look in an implementation with two pointers at each node. The implementation of Figure 6.15 only requires two pointers per node.
- The representation of Figure 6.15 is likely to be easier to implement, space efficient, and more flexible than the other implementations presented in this section.

-End-