DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 6

Non-Binary Trees

Outlines of Class (Lecture 6)

- Introduction
- General Tree Definitions and Terminology
- General Tree Traversals
- The Parent Pointer Implementation
- General Tree Implementations
- K-ary Trees

Introduction

- Many organizations are hierarchical in nature
- Trees provide a natural organization for data
- Some organization cannot easily be represented by a binary tree
- Need instead to use a tree whose nodes have an arbitrary number of children
- To distinguish them from binary trees, use the term general tree.
- K-ary trees: The trees whose internal nodes have a fixed number K of children where K is something other than two

General Tree Definitions and Terminology

- The main terminology for tree data structures comes from family trees, with the terms "parent," "child," "ancestor," and "descendant"
- A tree T is a finite set of one or more nodes such that there is one designated node R, called the root of T.
- If the set (T- {R}) is not empty, these nodes are partitioned into n > 0 disjoint subsets T_0, T_1, ..., T_{n-1}, each of which is a tree, and whose roots R_1, R_2, ..., R_n, respectively, are children of R.
- The subsets T_i ($0 \leq i < n$) are said to be subtrees of T.

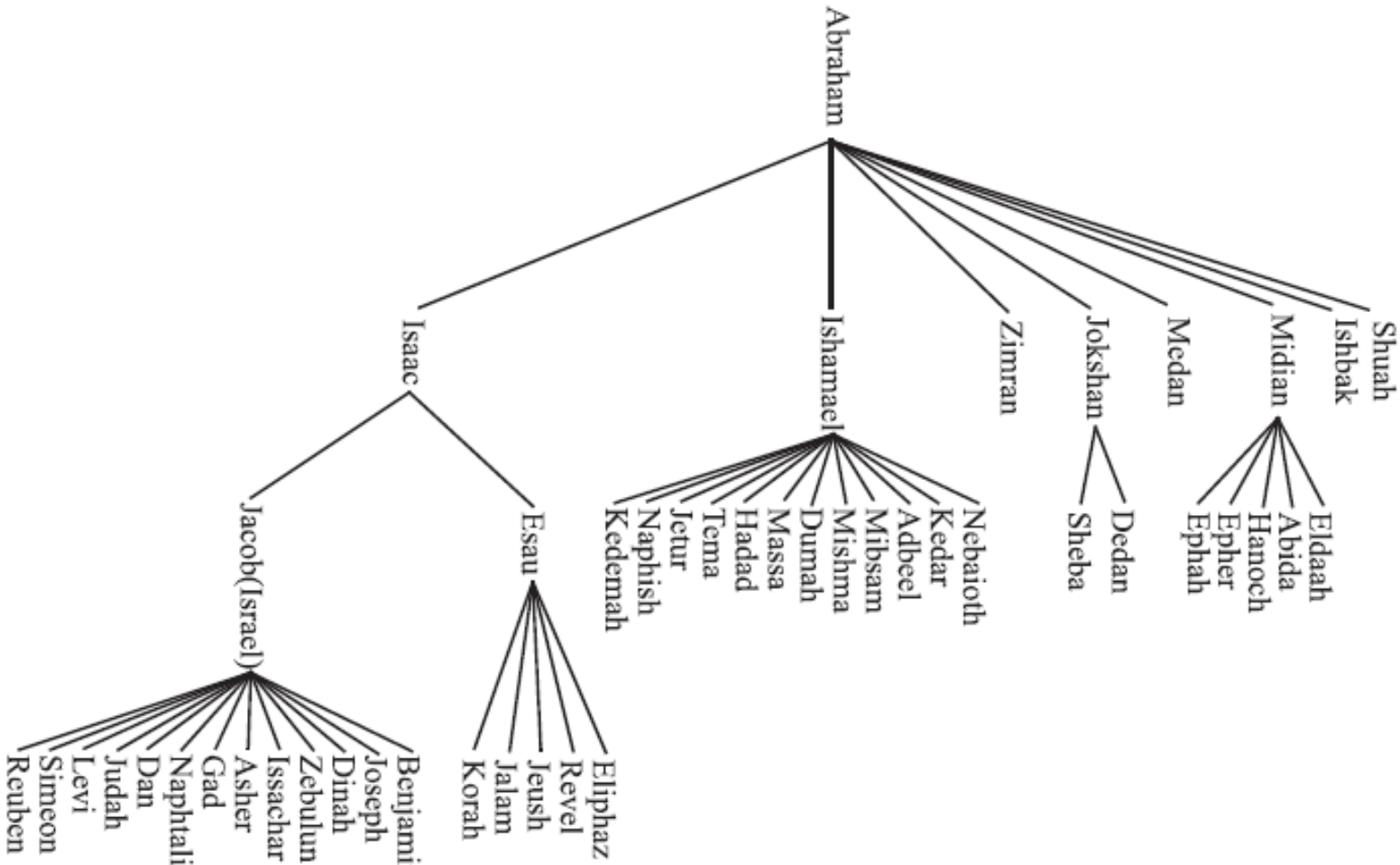


Figure: A family tree showing some descendants of Abraham

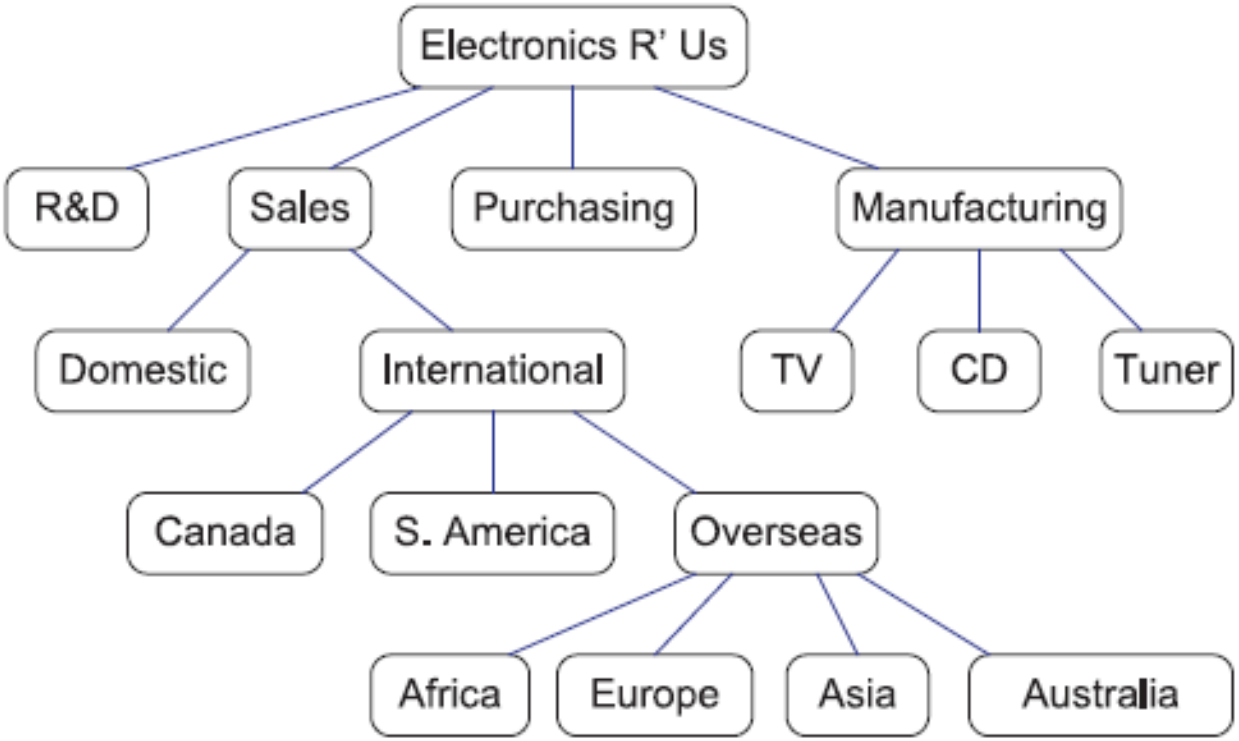


Figure: A tree with 17 nodes representing the organizational structure of a fictitious corporation

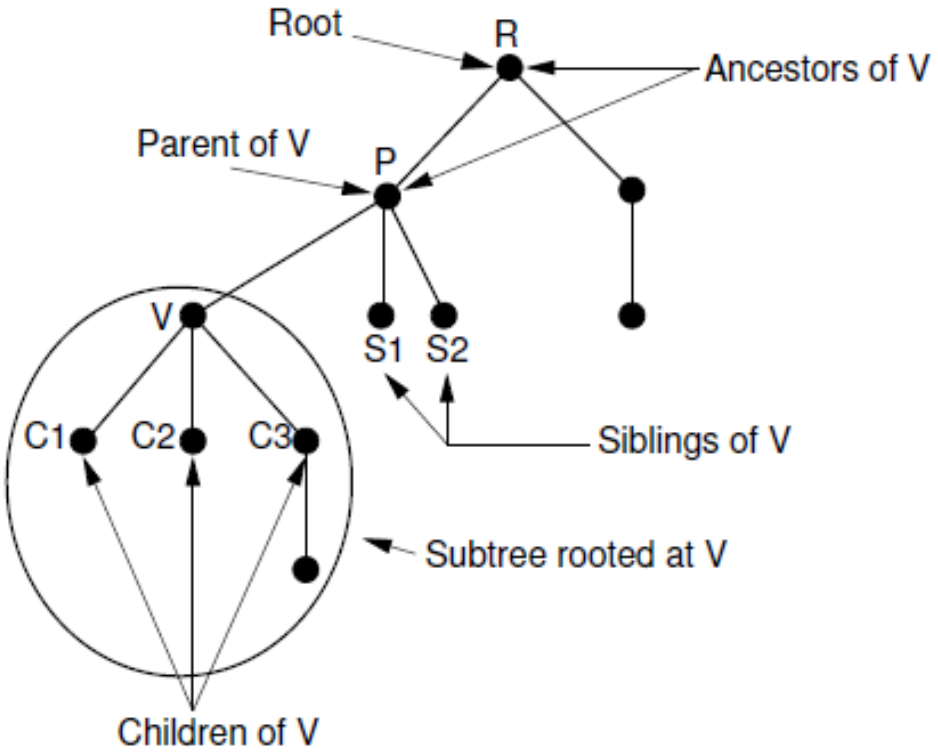


Figure: Notation for general trees

Differences between General Tree and Binary Tree

General Tree	Binary Tree
Each node can have infinite number of childern	Each node can has at most two nodes (left child and right child)
Can't be empty	Can be empty
Subtrees are not ordered	Subtrees are ordered
No limit on the degree of node	Cannot have more than degree 2
Root have in-degree 0 and maximum out-degree n.	Root have in-degree 0 and maximum out-degree 2.

Definitions for the general tree and general tree node

```
// General tree node ADT
template <typename E> class GTNode {
public:
  E value();                          // Return node's value
  bool isLeaf();                      // True if node is a leaf
  GTNode* parent();                   // Return parent
  GTNode* leftmostChild();            // Return first child
  GTNode* rightSibling();             // Return right sibling
  void setValue(E&);                  // Set node's value
  void insertFirst(GTNode<E>*); // Insert first child
  void insertNext(GTNode<E>*);  // Insert next sibling
  void removeFirst();                 // Remove first child
  void removeNext();                  // Remove right sibling
};

// General tree ADT
template <typename E> class GenTree {
public:
  void clear();                 // Send all nodes to free store
  GTNode<E>* root();     // Return the root of the tree
  // Combine two subtrees
  void newroot(E&, GTNode<E>*, GTNode<E>*);
  void print();                 // Print a tree
};
```

Formal Tree Definition

- 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.
- A tree can be empty (doesn't have any nodes)
- Two nodes that are children of the same parent are siblings.
- A node v is external if v has no children.
- A node v is internal if it has one or more children.
- External nodes are also known as leaves.

Tree Functions

- The tree ADT stores elements at the nodes of the tree.
- Each node of the tree is associated with a ***position*** object, which provides public access to nodes.
- Use the notation p (rather than v) to clarify that the argument to the function is a position and not a node.
- 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.
  - 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.

An informal interface for the tree ADT

```
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
};
```

A Linked Structure for General Trees

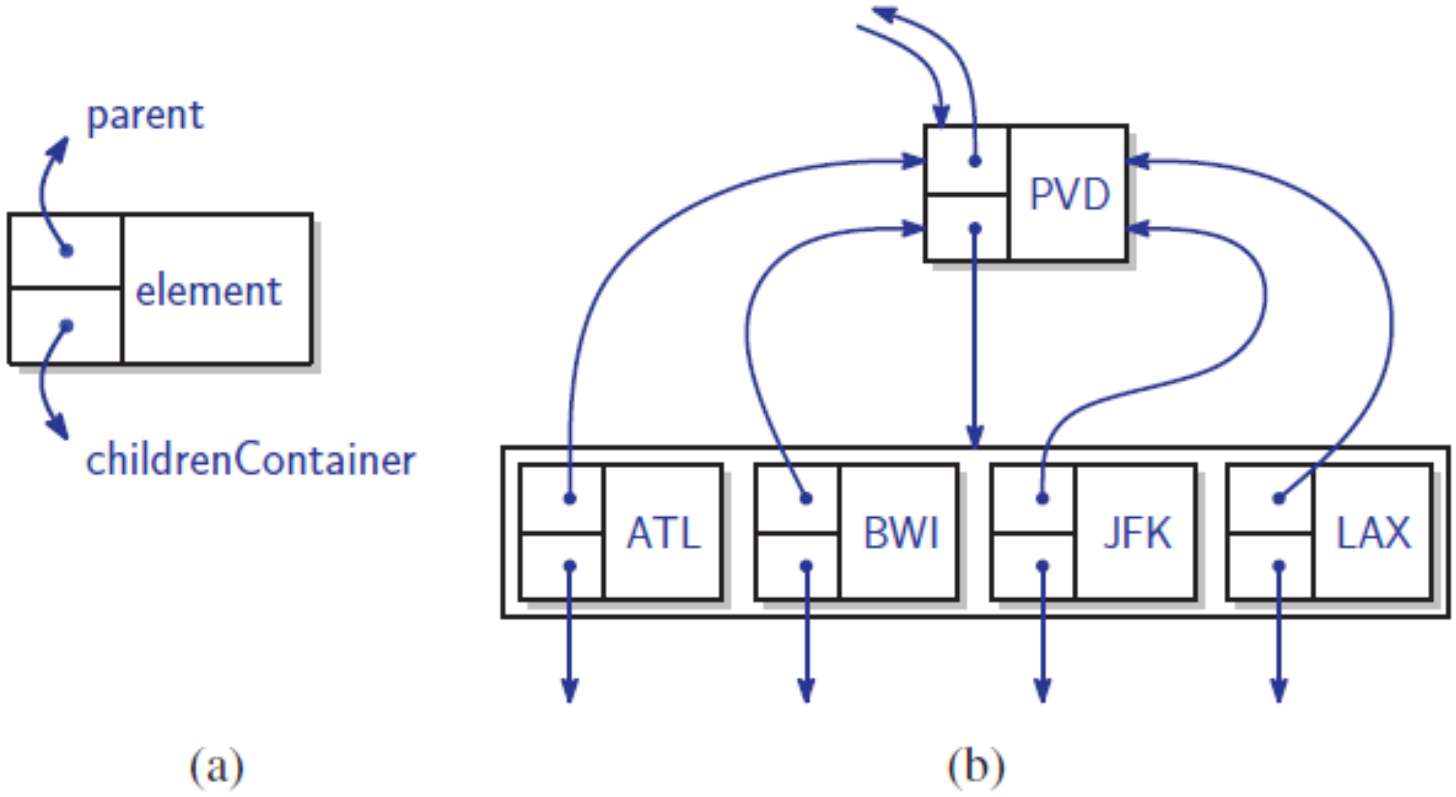


Figure: The linked structure for a general tree: (a) the node structure; (b) the portion of the data structure associated with a node and its children.

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

Table: Running times of the functions of an n-node linked tree structure. Let c_p denote the number of children of a node p. The space usage is $O(n)$.

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 (Cont.)

- A preorder traversal:
  RACDEBF

- A postorder traversal:
  CDEAFBR

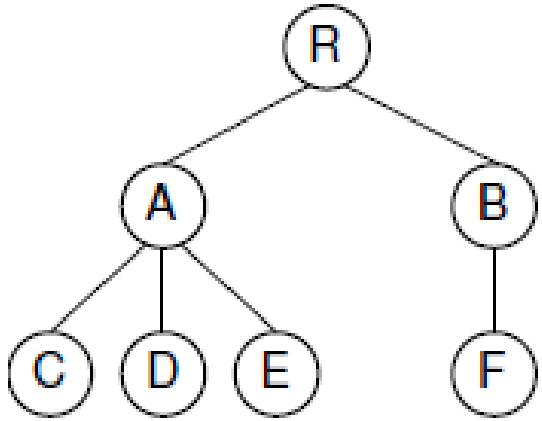


Figure: An example of a general tree

The Parent Pointer Implementation

- To store for each node only a pointer to that node's parent (parent pointer implementation)
- Inadequate for such important operations as finding the leftmost child or the right sibling for a node
- The parent pointer implementation stores precisely the information required to answer the following, useful question:
  - "Given two nodes, are they in the same tree?"
  - Ans: 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.
- FIND: The process of finding the ultimate root for a given node

The Parent Pointer Implementation (Cont.)

- Two basic operations

  (1) determine if two objects are in the same set, and

  (2) merge two sets together.

- Because two merged sets are united, the merging operation is called UNION and the whole process of determining if two objects are in the same set and then merging the sets goes by the name "UNION/FIND."

- Single array is being used to implement a collection of trees
  - ✓ easy to merge trees together with UNION operations

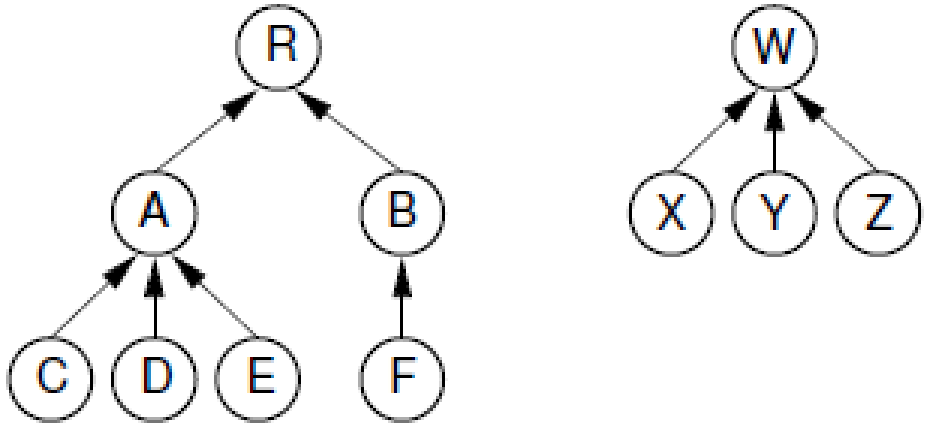
The Parent Pointer Implementation (Cont.)

```
// General tree representation for UNION/FIND
class ParPtrTree {
private:
  int* array;                          // Node array
  int size;                            // Size of node array
  int FIND(int) const;                 // Find root
public:
  ParPtrTree(int);                       // Constructor
  ~ParPtrTree() { delete [] array; } // Destructor
  void UNION(int, int);                // Merge equivalences
  bool differ(int, int);               // True if not in same tree
};

int ParPtrTree::FIND(int curr) const { // Find root
  while (array[curr] != ROOT) curr = array[curr];
  return curr;  // At root
}
```

Figure: General tree implementation using parent pointers for the UNION/ FIND algorithm

The Parent Pointer Implementation (Cont.)

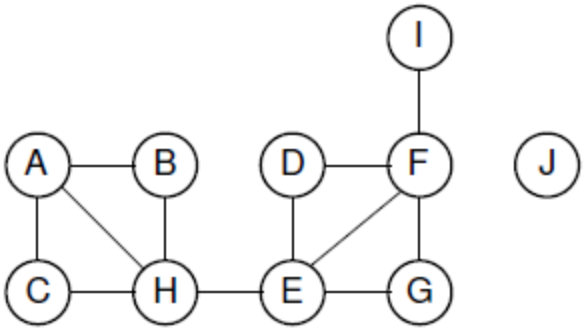


Parent's Index	/	0	0	1	1	1	2	/	7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

Figure: The parent pointer array implementation

Equivalence Processing

- A graph of ten nodes labeled A through J
- For nodes A through I, there is some series of edges that connects any pair of the nodes
- Node J is disconnected from the rest of the nodes
- Two nodes of the graph to be equivalent if there is a path between them
- Nodes A, H, and E would be equivalent in Figure, but J is not equivalent to any other
- Used to represent connections such as wires between components on a circuit board, or roads between cities



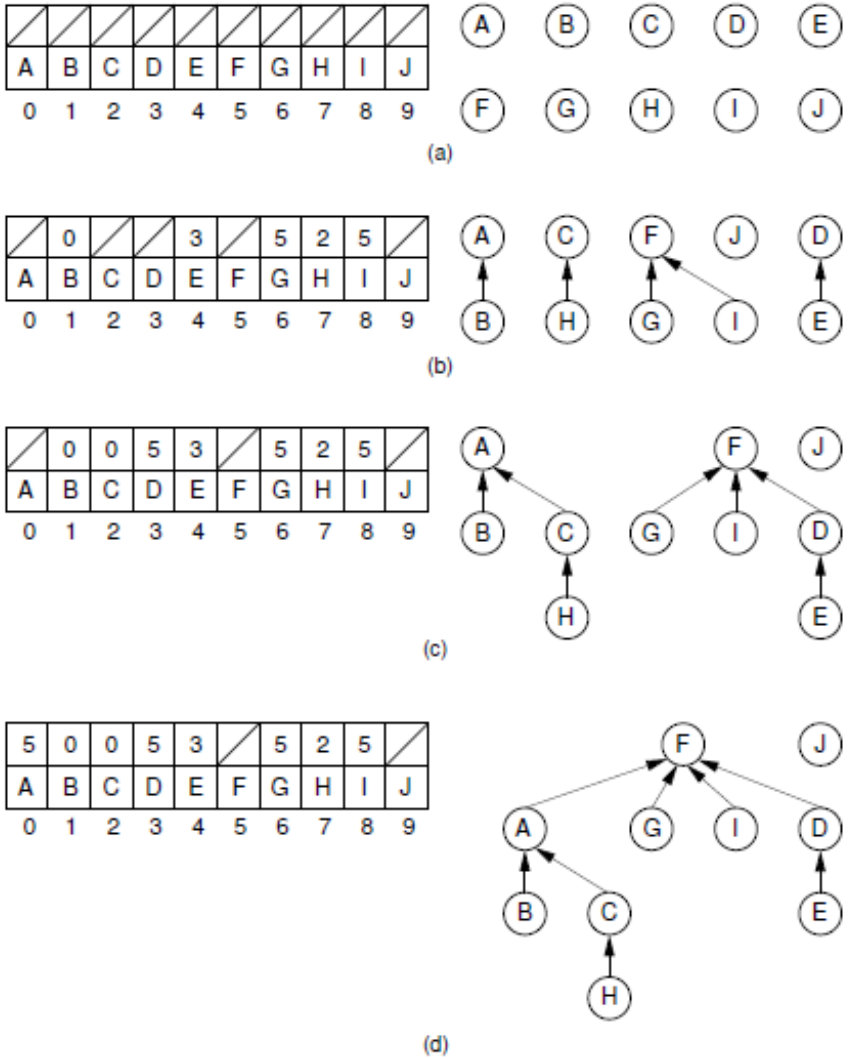


Figure: An example of equivalence processing. (a) Initial configuration for the ten nodes of the graph. The nodes are placed into ten independent equivalence classes. (b) The result of processing five edges: (A, B), (C, H), (G, F), (D, E), and (I, F). (c) The result of processing two more edges: (H, A) and (E, G). (d) The result of processing edge (H, E).

General Tree Implementations

- Several approaches to implementing general trees
- Advantages and disadvantages
  - the amount of space required to store a node
  - the relative ease with which key operations can be performed
- No restriction on how many children a node may have
- Once a node is created the number of children never changes (a fixed amount of space can be allocated for the node when it is created)

List of Children Implementation

- 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
- The leftmost child of a node can be found directly (the first element in the linked list)

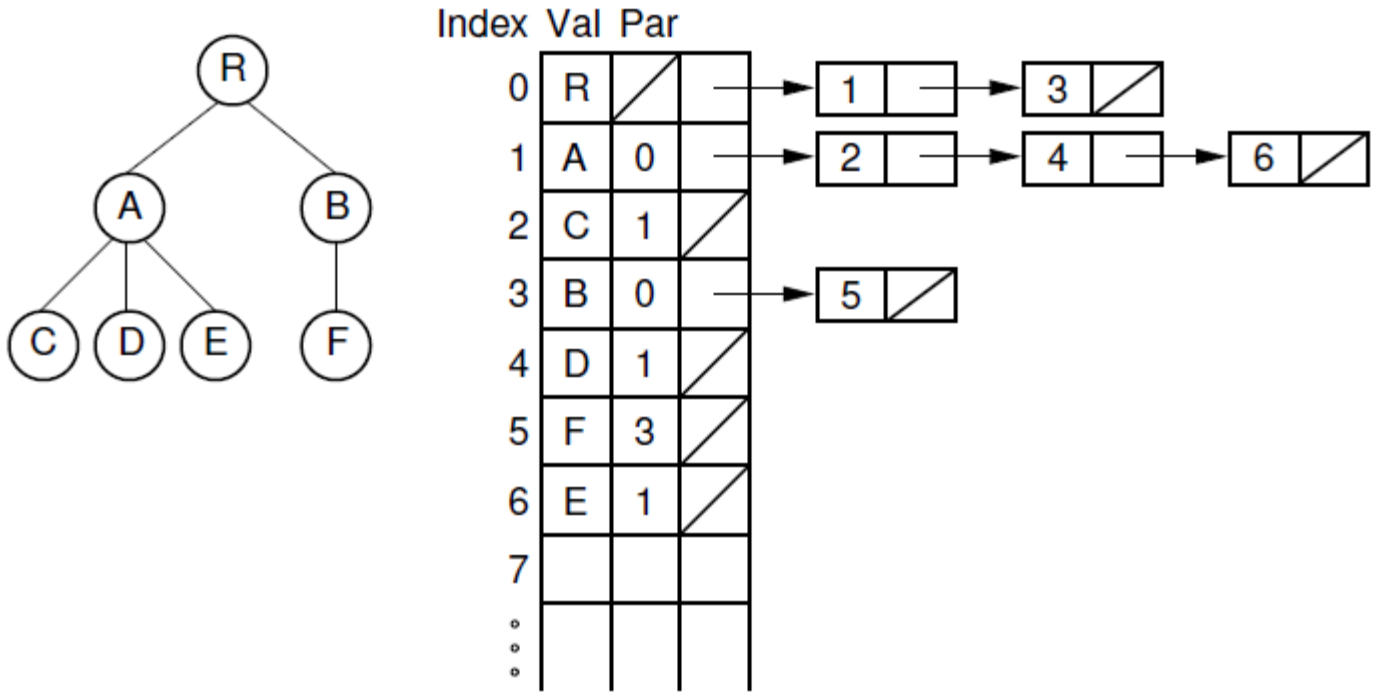


Figure: The “list of children” implementation for general trees. The column of numbers to the left of the node array labels the array indices. The column labeled “Val” stores node values. The column labeled “Par” stores indices (or pointers) to the parents. The last column stores pointers to the linked list of children for each internal node. Each element of the linked list stores a pointer to one of the node’s children (shown as the array index of the target node).

The Left-Child/Right-Sibling Implementation

- List of children implementation: difficult to access a node's right sibling
- The Left-Child/Right-Sibling Implementation: improvement of the "list of children" implementation
- Each node stores its value and pointers to its parent, leftmost child, and right sibling
- More space efficient than the "list of children" implementation,
- Each node requires a fixed amount of space in the node array

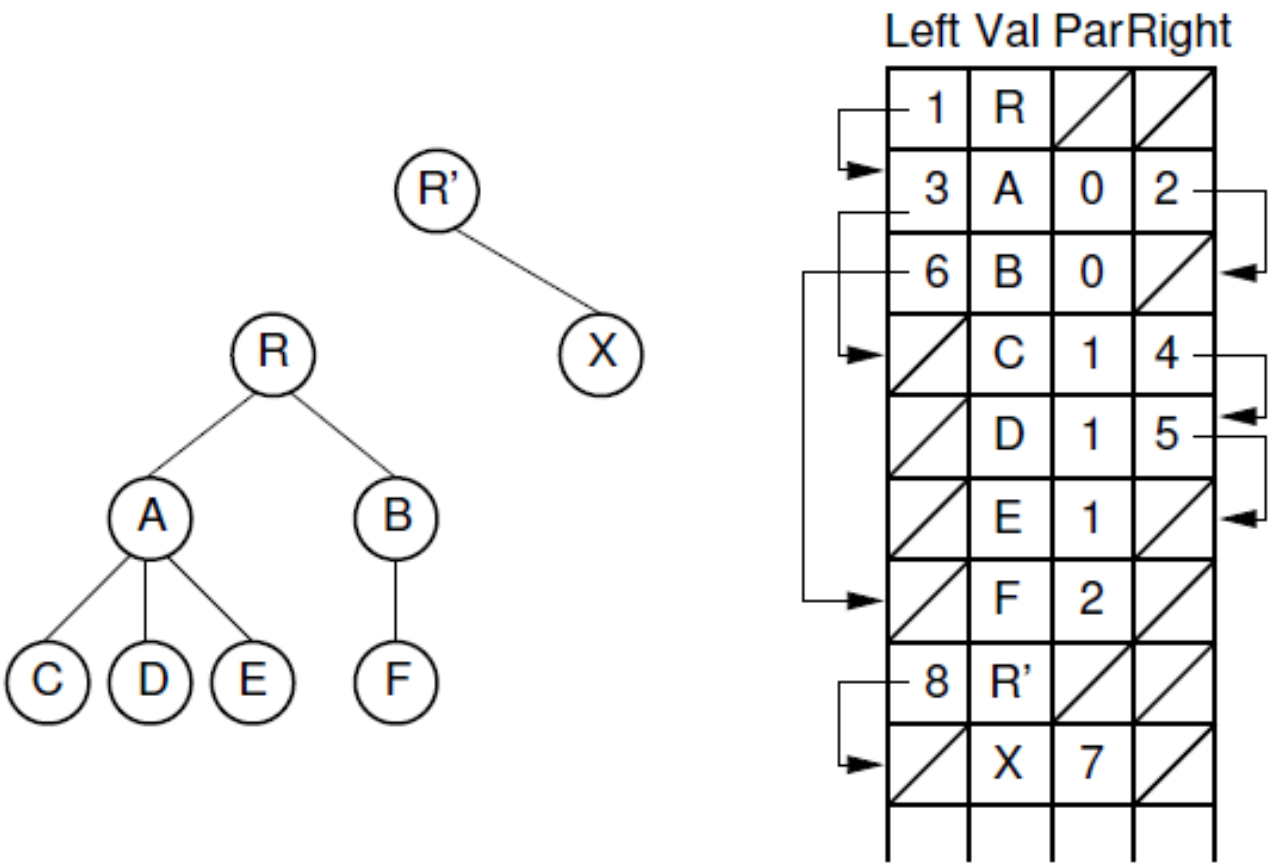


Figure: The “left-child/right-sibling” implementation

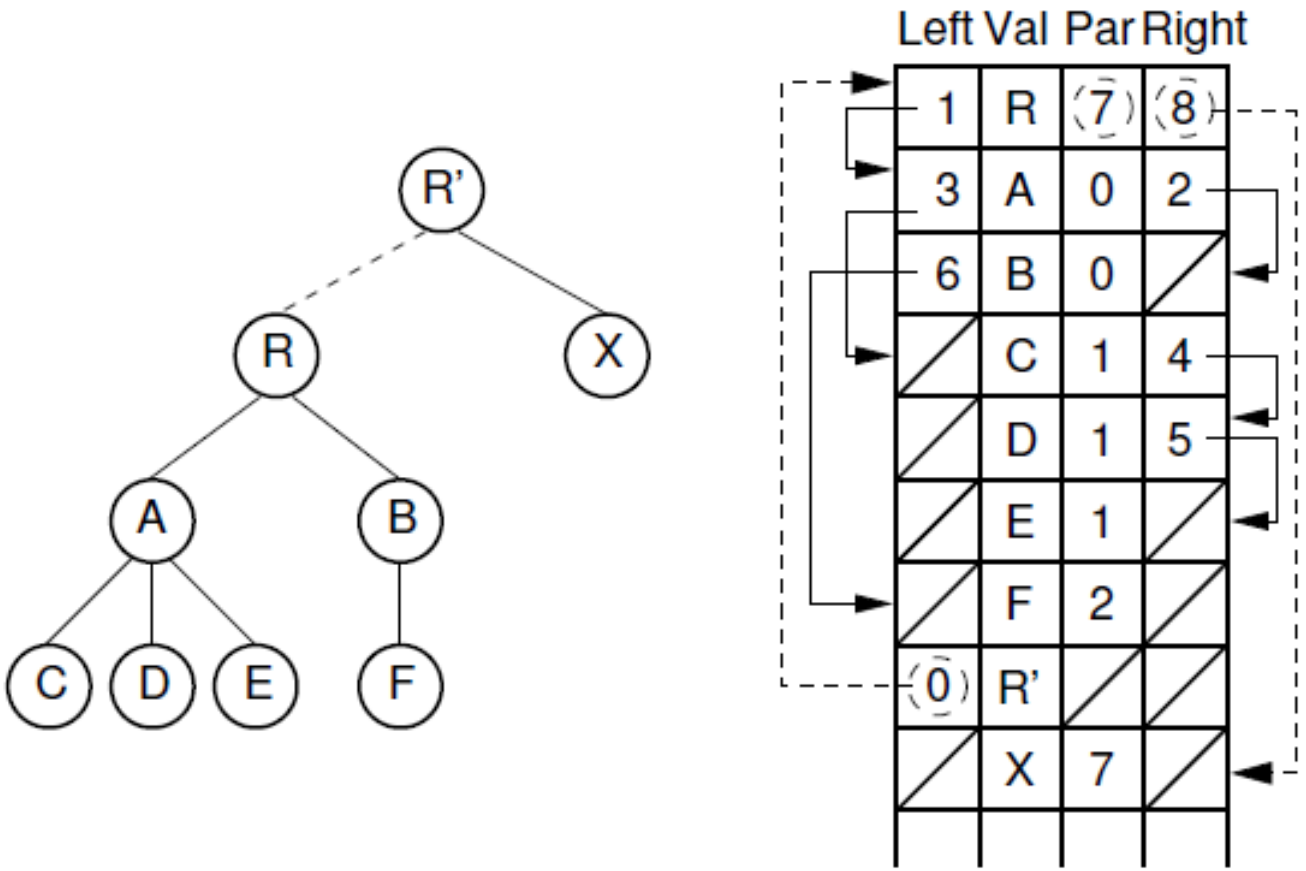


Figure: Combining two trees that use the “left-child/right-sibling” implementation

Dynamic Node Implementations

- Two basic approaches : to allocate variable space for each node
- First:
  - to allocate an array of child pointers as part of the node.
  - each node stores an array-based list of child pointers
  - assumes that the number of children is known when the node is created
  - works best if the number of children does not change.
- Second:
  - more flexible, requires more space to store a linked list of child pointers with each node
  - essentially the same as the “list of children” implementation
  - with dynamically allocated nodes rather than storing the nodes in an array

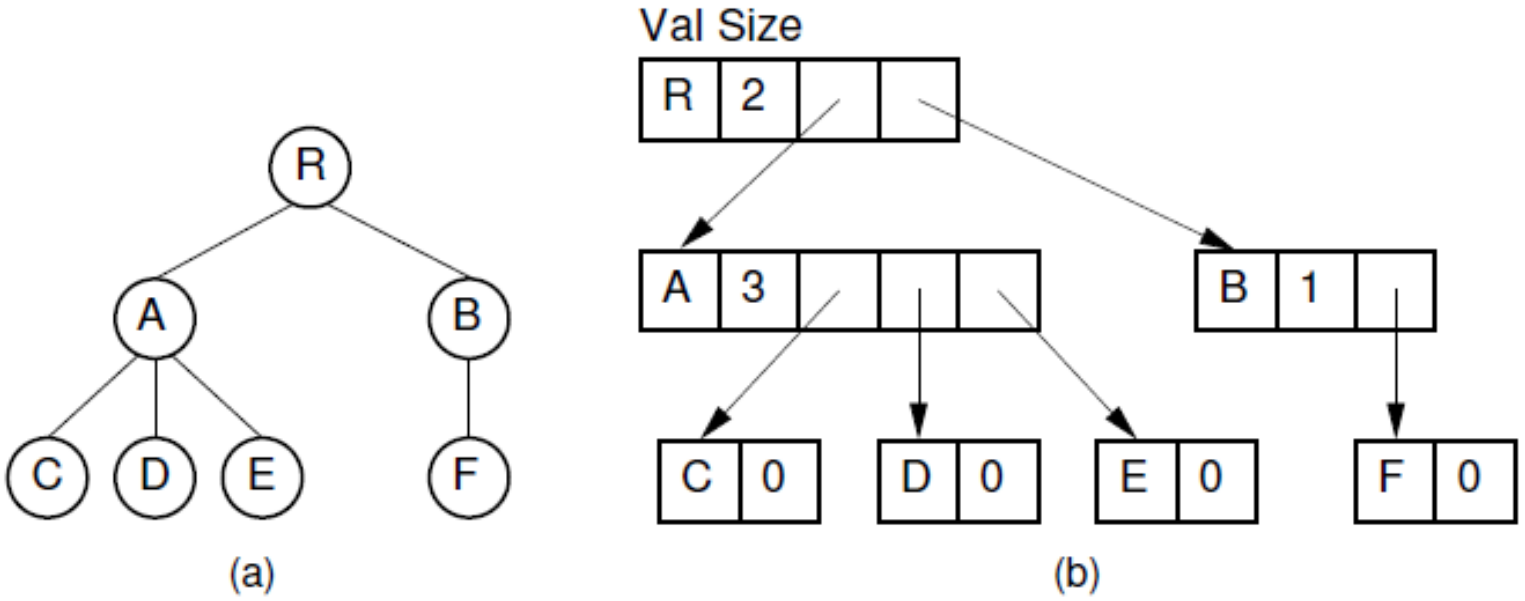


Figure: A dynamic general tree representation with fixed-size arrays for the child pointers. (a) The general tree. (b) The tree representation. For each node, the first field stores the node value while the second field stores the size of the child pointer array

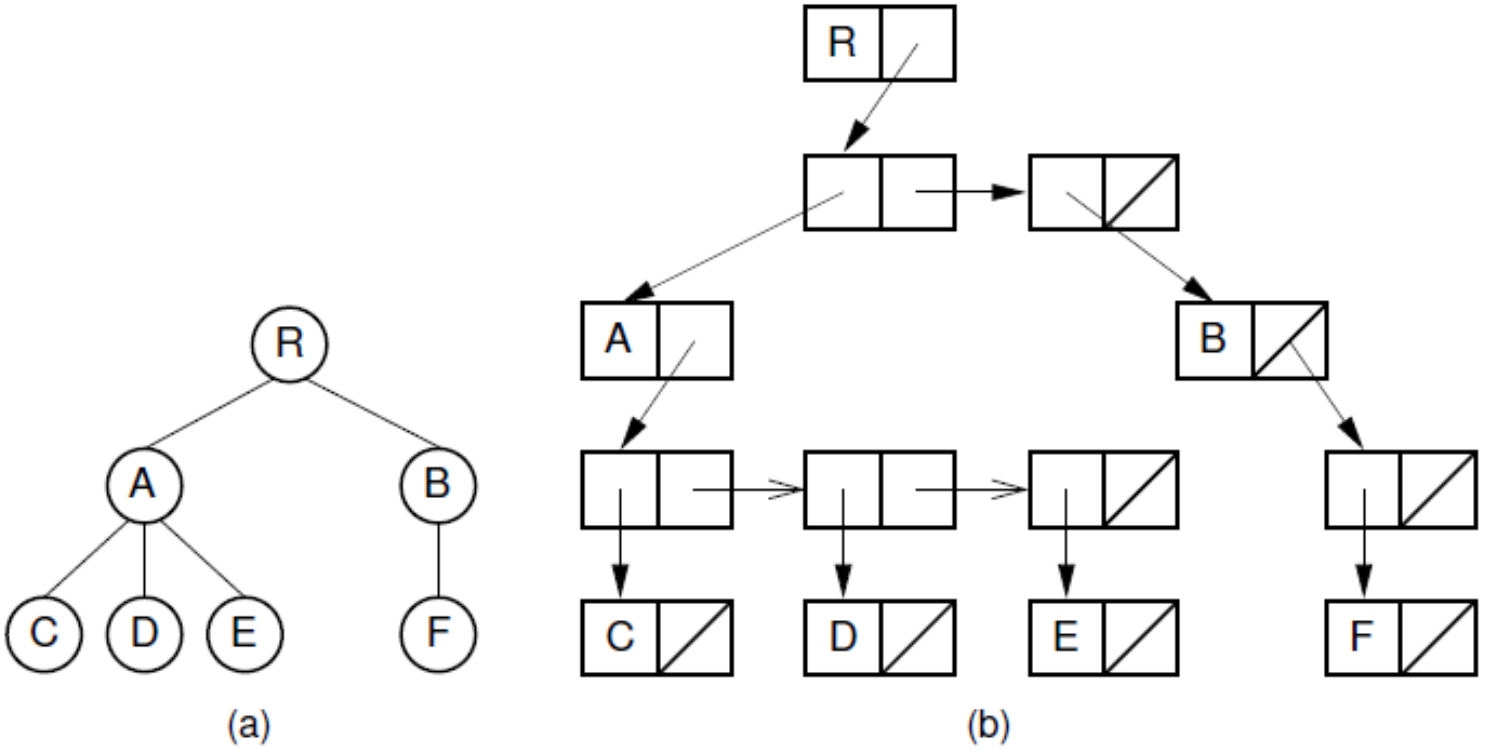


Figure: A dynamic general tree representation with linked lists of child pointers. (a) The general tree. (b) The tree representation.

Dynamic Left-Child/Right-Sibling Implementation

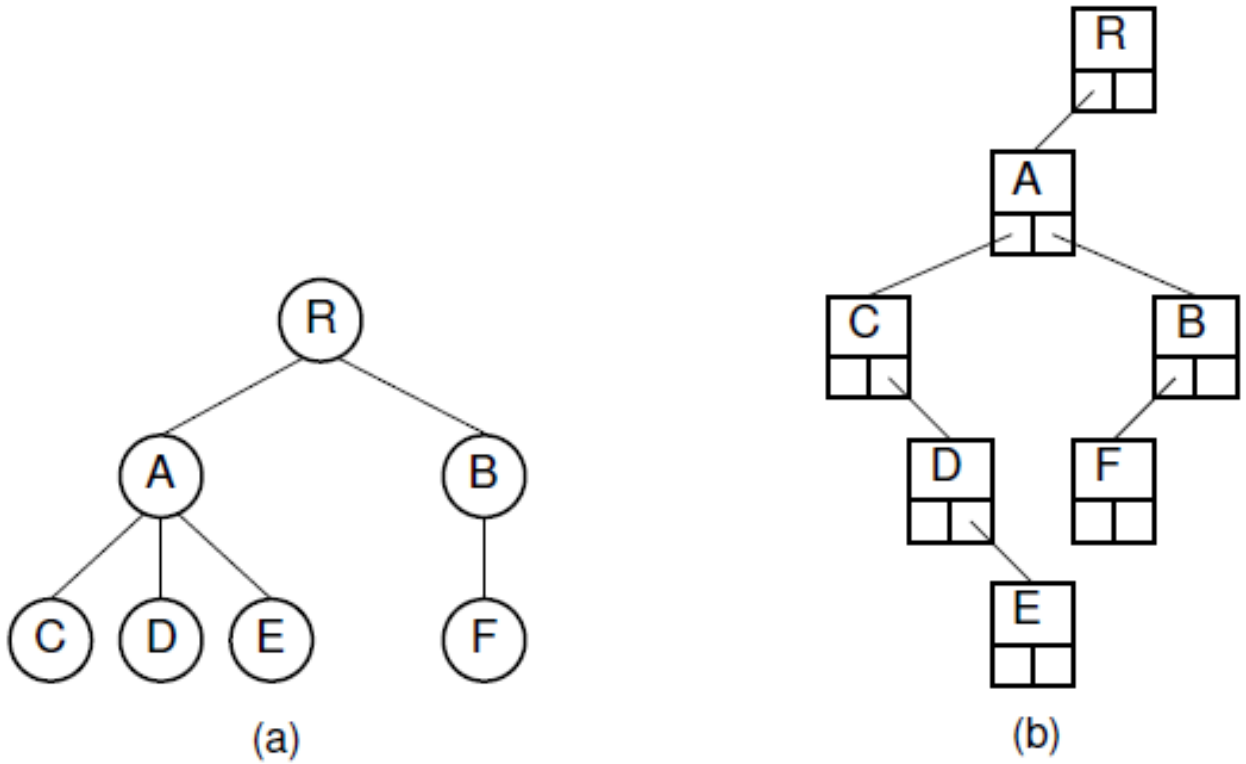


Figure: A general tree converted to the dynamic “left-child/right-sibling” representation

K-ary Trees

- The trees whose internal nodes all have exactly K children
- A full binary tree is a 2-ary tree
- Unlike general trees, K-ary tree nodes have a fixed number of children
- Easy to implement
- As K becomes large, the potential number of NULL pointers grows and the difference between the required sizes for internal nodes and leaf nodes increases
- As K becomes larger, the need to choose separate
- Implementations for the internal and leaf nodes becomes more pressing

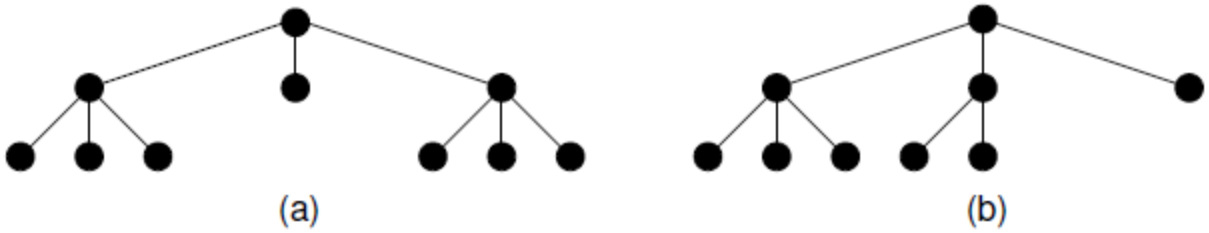


Figure: Full and complete 3-ary trees.
(a) This tree is full (but not complete).
(b) This tree is complete (but not full)

Next Week Lecture (Week 7)

Lecture 7: Internal Sorting

Thank you!