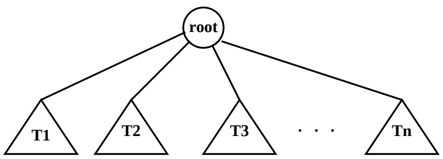
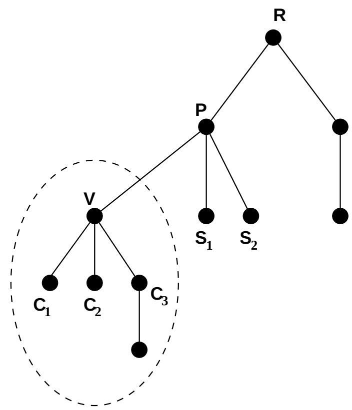
Definitions

- A **tree** is a collection of nodes.
  - ☐ The collection can be empty
  - ☐ Otherwise, a tree consists of
    - ∘ A distinguished node **r**, called the **root**
    - ∘ Zero or more nonempty sub-trees T_1, T_2, ..., T_n, each of whose roots are connected by a directed **edge** from r
- General Tree characteristics
  - ☐ The root of each subtree is a **child** of r
  - ☐ r is the **parent** of each subtree root.
  - ☐ Example using recursive definition:



Definitions (cont.)

- General Tree characteristics
  - ☐ **Out degree**: the number of children of that node
  - ☐ **forest**: a collection of one or more trees.
  - ☐ Binary tree definitions that don't conflict also apply
  - ☐ Example (Figure 6.1):



General Tree Node

What operations must be supported in a general tree?

- The General Tree and Node ADTs:

```
template <class Elem> class GTNode {
public:
  GTNode(const Elem&);       // Constructor
  ~GTNode();                 // Destructor
  Elem value();              // Return node's value
  bool isLeaf();             // TRUE if node is a leaf
  GTNode* parent();          // Return parent
  GTNode* leftmost_child();  // Return first child
  GTNode* right_sibling();   // Return right sibling
  void setValue(Elem&);      // Set node's value
  void insert_first(GTNode<Elem>* n);// Insert 1st child
  void insert_next(GTNode<Elem>* n); // Insert next sib
  void remove_first();       // Remove first child
  void remove_next();        // Remove right sibling
};

template <class Elem> class GenTree {
  public:
    GenTree();
    ~GenTree();
    void clear();            // Free the nodes
    GTNode* root();          // return root
    void newroot(Elem,       // Combine trees
    GTNode<Elem>*, GTNode<Elem>*);
};
```


General Tree Traversals

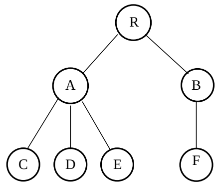
There is no concept of an inorder traversal

- Recursive definitions:
  - ☐ Preorder: visit the root, then perform a preorder traversal of each subtree from left to right
  - ☐ Postorder: perform a preorder traversal of each subtree from left to right, then visit the root
  - ☐ Preorder Example:

```
template <class Elem>
void GenTree<Elem>::
     printhelp(GTNode<Elem>* subroot) {
  if (subroot->isLeaf())
    cout << "Leaf: ";
  else
    cout << "Internal: ";
  cout << subroot->value() << "\n";
  for (GTNode<Elem>* temp = subroot->leftmost_child();
       temp != NULL; temp = temp->right_sibling())
    printhelp(temp);
}
```

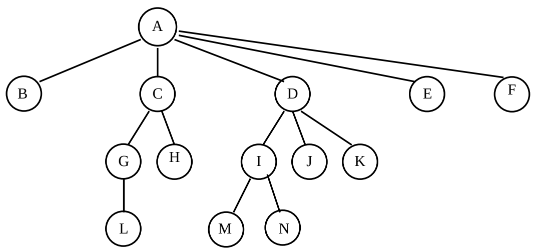
General Tree Example

- Example:



  - ☐ Preorder: R A C D E B F
  - ☐ Postorder: C D E A F B R
  - ☐ Inorder?

- Example:



  - ☐ Preorder:
  - ☐ Postorder:

General Tree Implementations

There are several choices, depending on application

- Array-based implementations:
  - ☐ Parent pointer
  - ☐ Lists of children (hybrid array/link)
  - ☐ Leftmost child/right sibling
- Link-based implementations:
  - ☐ Fixed-size arrays for child pointers
  - ☐ Linked lists of child pointers
- Storage-based
  - ☐ Sequential tree implementation

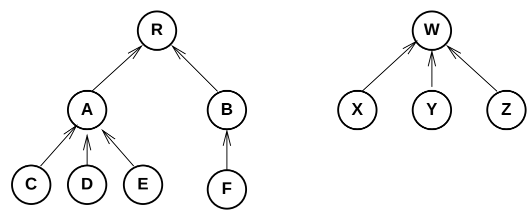
Parent Pointer Implementation

Probably the simplest general tree implementation

- Each node stores only a pointer to its parent
  - ☐ Not very general purpose
  - ☐ Very good for answering whether two nodes are in the same tree
    - ○ Operation is called **FIND**
  - ☐ A Disjoint set problem:
    - ○ Determine if two objects are in the same set (**FIND**)
    - ○ Merge two sets together (**UNION**)
- A useful application is determining equivalence classes

Parent Pointer Implementation

- Nodes are stored in an array:



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

Union/Find

- Are two elements in the same tree?

```
bool Gentree::differ(int a, int b) {
  int root1 = FIND(a);
  int root2 = FIND(b);
  return (root1 != root2);
}
```

- Implementing Union and Find:

```
void Gentree::UNION(int a, int b) {
  int root1 = FIND(a);
  int root2 = FIND(b);
  if (root1 != root2)
    array[root2] = root1;
}

int Gentree::FIND(int curr) const {
  while (array[curr] != ROOT)
    curr = array[curr];
  return curr;
}
```

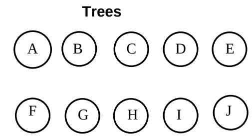
- Keep the depth small using **weighted union rule**
  - ☐ Weighted union rule: join the tree with fewer nodes to the tree with more nodes

Equivalence Processing Example

- Initial:

Array

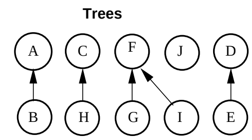
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
A	B	C	D	E	F	G	H	I	J



- After processing (A,B), (C,H), (G,F), (D,E), (I,F):

Array

–1	0	–1	–1	3	–1	5	2	5	–1
A	B	C	D	E	F	G	H	I	J

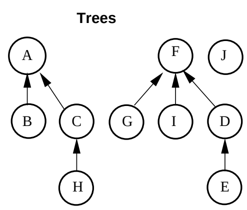


Equivalence Processing Example (cont.)

- After processing (H,A), (E,G)

Array

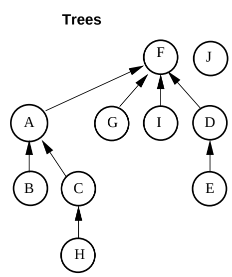
–1	0	0	5	3	–1	5	2	5	–1
A	B	C	D	E	F	G	H	I	J



- After processing (H,E)

Array

5	0	0	5	3	–1	5	2	5	–1
A	B	C	D	E	F	G	H	I	J



Path Compression

Resets the parent of every node on the path from node X to root R

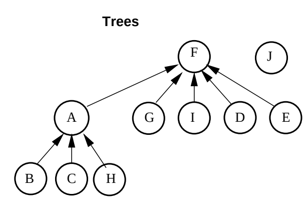
- Code:

```
GTNode* Gentree::FIND(GTNode* curr) const {
  if (array[curr] == ROOT)
    return curr;
  return array[curr] = FIND(array[curr]);
}
```

- Process (H,E):

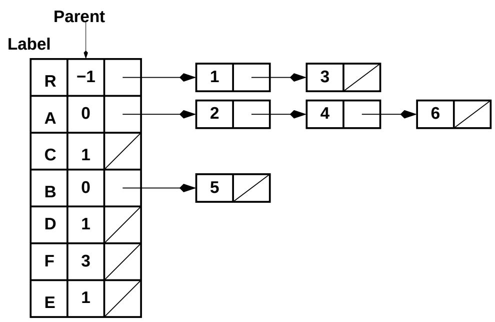
Array

5	0	0	5	5	–1	5	0	5	–1
A	B	C	D	E	F	G	H	I	J



Lists of Children
Hybrid Representation

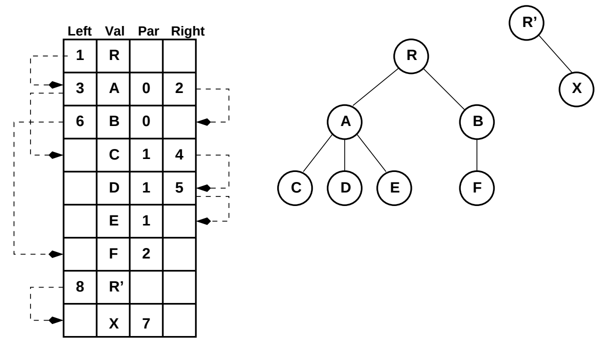
- Key question: how well does a representation perform certain tasks?
  - ☐ find left child and right sibling
  - ☐ find a parent



- What tree does this represents?

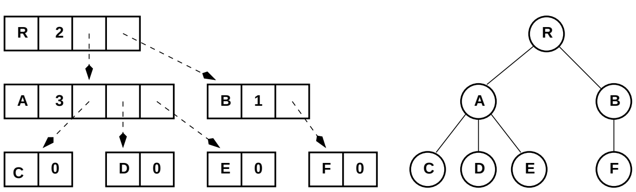
Leftmost Child/Right Sibling
Array Representation

- Array of "pointers" (indices), contains:
  - ☐ index of Left child
  - ☐ label (value)
  - ☐ index of parent
  - ☐ index of right sibling



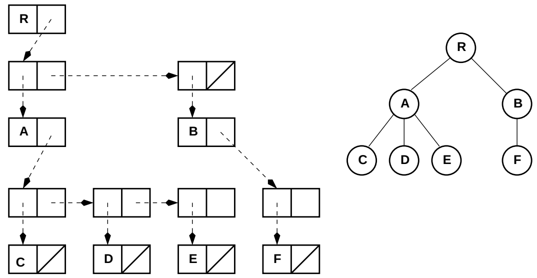
Fixed Size Pointer Array
Linked Representation

- Each parent maintains an array of pointers to children

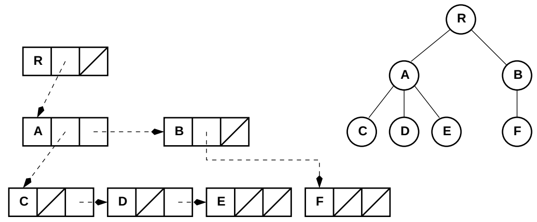


Linked Lists of Child Pointers
Linked Representation

- Each parent maintains an array of pointers to children



- Alternative



K-ary Trees

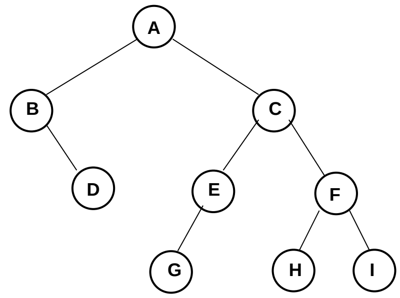
A *k-ary* tree is a tree whose nodes may have up to k children

- A binary tree is the same as a k-ary tree for $k = 2$
- Features and disadvantages
  - ☐ Relatively easy to implement
  - ☐ More wasted space as k grows
- As go FBTs, so go FKTs:
  - ☐ Full 3-ary Tree Theorem: The number of leaves in a non-empty full 3-ary tree is equal to $2^n + 1$ where n is the number of internal nodes
  - ☐ Corollary: The number of empty subtrees in a nonempty 3-ary tree is ...?

Sequential Tree Implementations

Represents an application of the space/time tradeoff principle

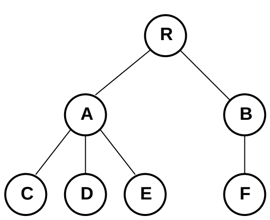
- Goal: store a series of node values with minimum information necessary to reconstruct the tree structure
  - ☐ Advantage: space is saved
  - ☐ Disadvantage: cost to regenerate tree (loss of efficient access to nodes)



- Use a symbol to mark NULL links: AB/D//CEG///FH//I//

Sequential Tree Implementations (cont.)

- FBT implies the list may be stored more efficiently
  - ☐ Mark a leaf or an internal
  - ☐ In a FBT, no '/' characters in the representation
  - ☐ Using the prior example, internals are marked
  - ☐ Representation is A'B'/DC'E'G/F'HI
- General trees require a ‘list end’ indicator



  - ☐ Representation is RAC)D)E))BF)))