Data Structure
Chapter 6

Non-Binary Trees

Dr. Patrick Chan
School of Computer Science and Engineering
South China University of Technology

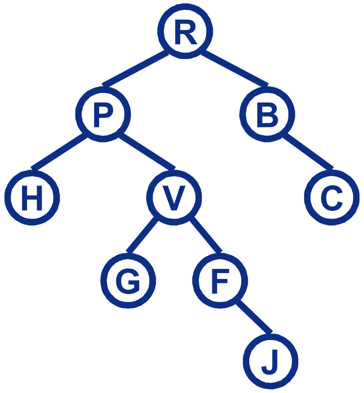
Outline

- Non-Binary (General) Tree (Ch 6.1)
- Parent Pointer Implementation (Ch 6.2)
- List of Children Implementation (Ch 6.3.1)
- Left-Child/Right-Sibling Implementation (Ch 6.3.2)
- Dynamic Left-Child/Right-Sibling Implementation (Ch 6.3.4)
- Dynamic Node Implementation (Ch 6.3.3)
- K-ary Trees (Ch 6.4)
- Sequential Tree Implementation (Ch 6.5)

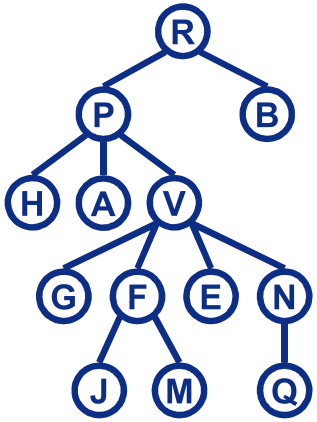
General Tree (Non-binary Tree)

- 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
- The remaining nodes in $(\mathrm{T} - \{R\})$ are partitioned into $n \geq 0$ disjoint subsets $\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_k$, each of which is a tree, and whose roots $R_1, R_2, \ldots, R_k$, respectively, are children of R

General Tree



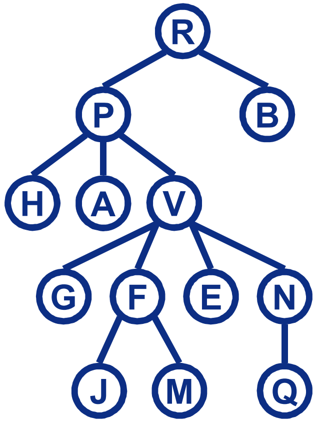
Binary Tree
- Two, one or zero child



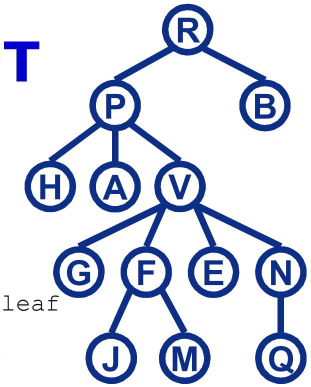
General Tree
- Any number of child

General Tree

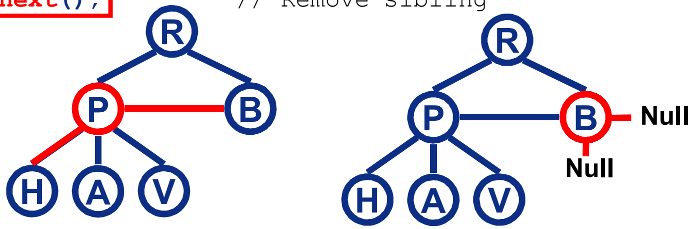
- To maintain the structure of **Binary Tree**, each node has
  - Left child pointer
  - Right child pointer
- How about **General Tree**?



General Tree Node: ADT



```
// General tree node ADT
template <class Elem> class GTNode {
public:
  GTNode(const Elem&);                // Constructor
  ~GTNode();                          // Destructor
  Elem value();                       // Return value
  bool isLeaf();                      // TRUE if is a leaf
  GTNode* leftmost_child();           // First child
  GTNode* right_sibling();            // Right sibling
  void setValue(Elem&);               // Set value
  void insert_first(GTNode<Elem>* n);
  void insert_next(GTNode<Elem>* n);
  void remove_first();                // Remove first child
  void remove_next();                 // Remove sibling
};
```

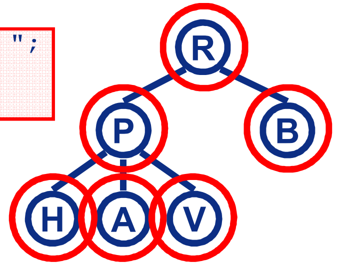


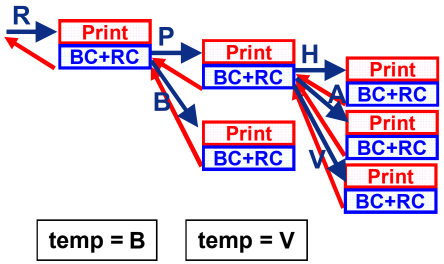
General Tree: ADT

```
// General Tree ADT
template <class Elem> class GenTree {

private:
  void printhelp(GTNode*);     // Print helper function

public:
  GenTree();                   // Constructor
  ~GenTree();                  // Destructor

  void clear();                // Send nodes to free store
  GTNode* root();              // Return the root

                               // Combine two subtrees
  void newroot(ELEM, GTnode *, GTnode *);
  Void print();                // Print a tree
};
```

General Tree: Traversal 1

```
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);
}
```

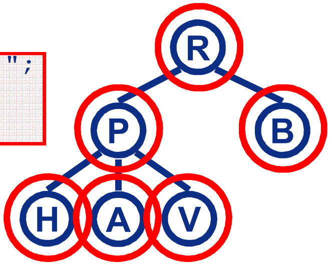


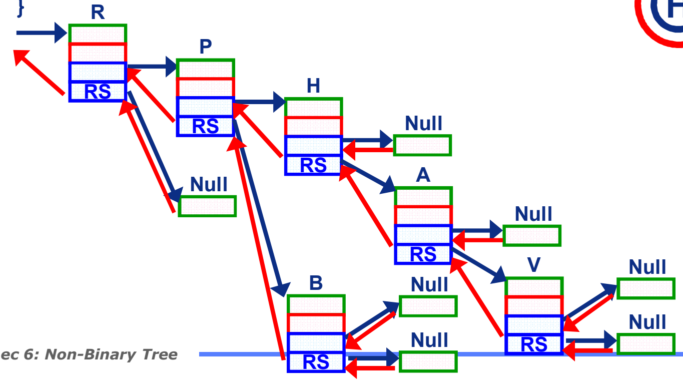


```
Internal: R
Internal: P
Leaf: H
Leaf: A
Leaf: V
Leaf: B
```

General Tree: Traversal 2

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





```
Internal: R
Internal: P
Leaf: H
Leaf: A
Leaf: V
Leaf: B
```

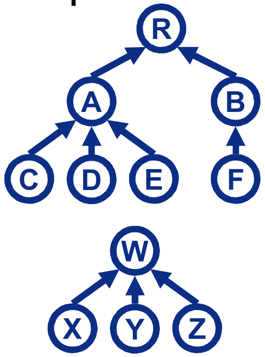
General Tree: Implementation

- Parent Pointer Implementation
- List of Children Implementation
- Left-Child/Right-Sibling Implementation
- Dynamic Left-Child/Right-Sibling Implementation
- Dynamic Node Implementation

Parent Pointer Implementation

- **Only storing pointer** may be the simplest general tree implementation

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



- Good for answering the question

  Are these two nodes in the same tree?

Parent Pointer Implementation

Equivalence Class

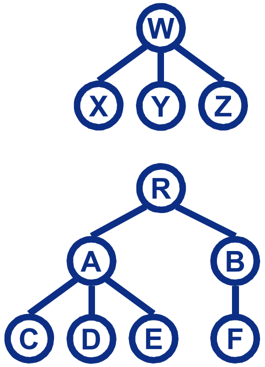
- Assigning the members of a set to disjoint subsets called **equivalence classes**
  - E.g.
    Object A and B are equivalent
    Object B and C are equivalent
    Object A and C must be equivalence
- **UNION/FIND implementation**
  - Check if two objects are equivalent: **differ**
  - Set "two objects are equivalent": **UNION**

Parent Pointer Implementation

```
class Gentree {   // Gentree for UNION/FIND
private:
  int* array;                 // Node array
  int size;                   // Size of node array
  int FIND(int) const;        // Find root

public:
  Gentree(int);               // Constructor
  ~Gentree() { delete [] array; }  // Destructor
  void UNION (int, int);      // Merge equivalences
  void differ (int, int);     // TRUE if not in same tree
}
```

Parent Pointer Implementation

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

// Return TRUE if nodes in different trees
bool Gentree::differ(int a, int b) {
  int root1 = FIND(a);    // Find root for a
  int root2 = FIND(b);    // Find root for b
  return root1 != root2; // Compare roots
}

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



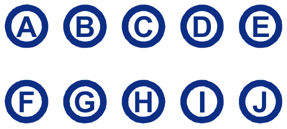
array →

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

Parent Pointer Implementation

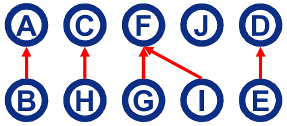
Equivalence Class: Example

0	1	2	3	4	5	6	7	8	9
/	/	/	/	/	/	/	/	/	/
A	B	C	D	E	F	G	H	I	J



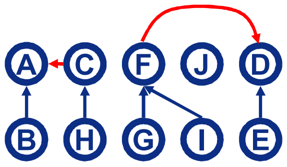
(A, B) (C, H) (F, G) (F, I) (D, E)

0	1	2	3	4	5	6	7	8	9
/	0	/	/	3	/	5	2	5	/
A	B	C	D	E	F	G	H	I	J



(A, H) (E, G)

0	1	2	3	4	5	6	7	8	9
/	0	0	/	3	3	5	2	5	/
A	B	C	D	E	F	G	H	I	J

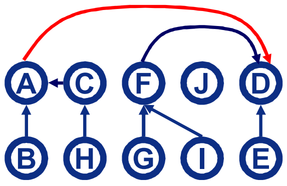


Parent Pointer Implementation

Equivalence Class: Example

(E, H)

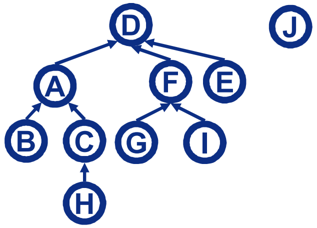
0	1	2	3	4	5	6	7	8	9
3	0	0	/	3	3	5	2	5	/
A	B	C	D	E	F	G	H	I	J



Is (A, B)? Yes

Is (H, D)? Yes

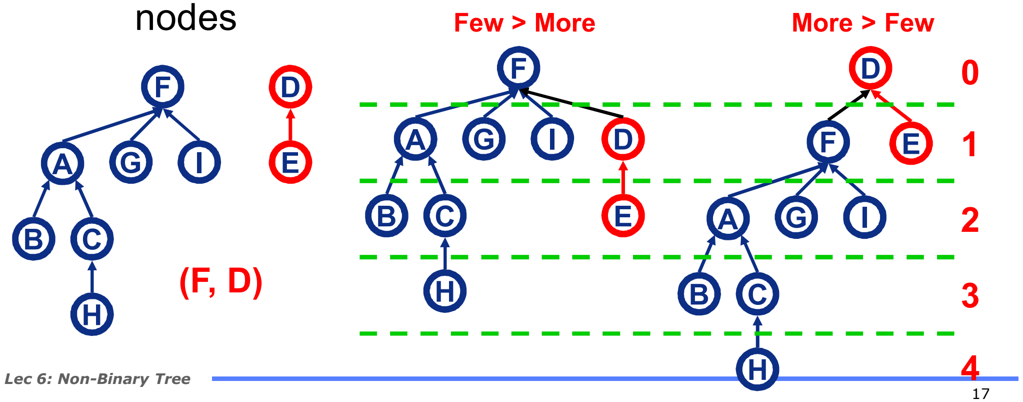
Is (J, I)? No



Parent Pointer Implementation

Equivalence Class: Reduce the cost

- The search cost can decrease by reducing the height of the tree
  - **Weighted Union Rule**
    - Join the tree with fewer nodes to the tree with more nodes



Parent Pointer Implementation

Equivalence Class: Reduce the cost

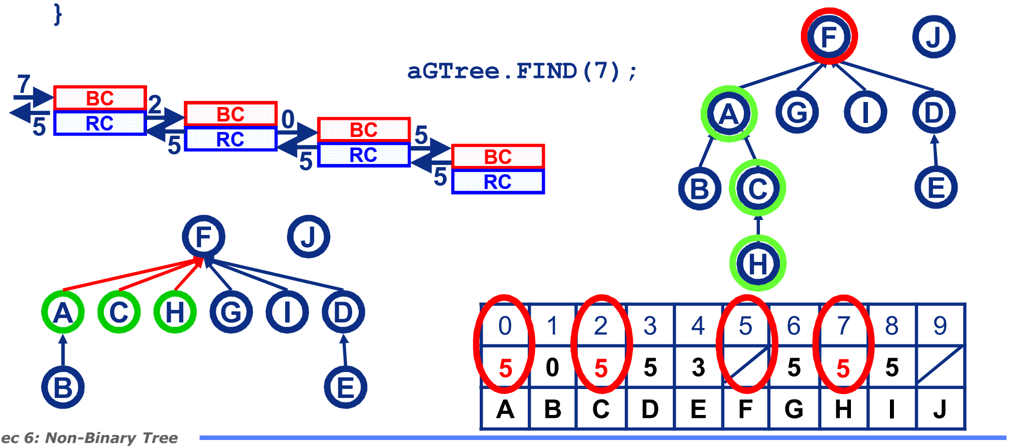
- **Path Compression**

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

Base Case

Recursive Call

aGTree.FIND(7);



0	1	2	3	4	5	6	7	8	9
5	0	5	5	3	/	5	5	5	/
A	B	C	D	E	F	G	H	I	J

☺ Small Exercise!!!! ☺

(A, C)

(D, F)

(L, A)

(H, A)

(I, L)

(F, B)

(G, H)

(J, K)

If (A, I)?

If (J, B)?

If (K, J)?

If (L, H)?
(path compression)

0	1	2	3	4	5	6	7	8	9	10	11
/	/	/	/	/	/	/	/	/	/	/	/
A	B	C	D	E	F	G	H	I	J	K	L

☺ Small Exercise!!!! ☺

(A, C)

(D, F)

(L, A)

(H, A)

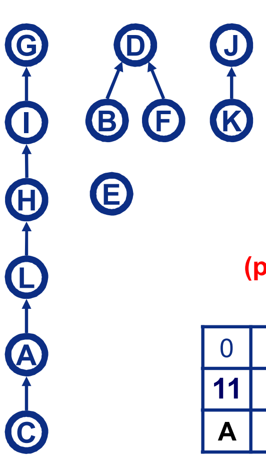
(I, L)

(F, B)

(G, H)

(J, K)

0	1	2	3	4	5	6	7	8	9	10	11
11	3	0	/	/	3	/	8	6	/	9	7
A	B	C	D	E	F	G	H	I	J	K	L

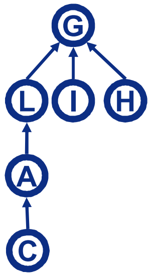


If (A, I)? Yes

If (J, B)? No

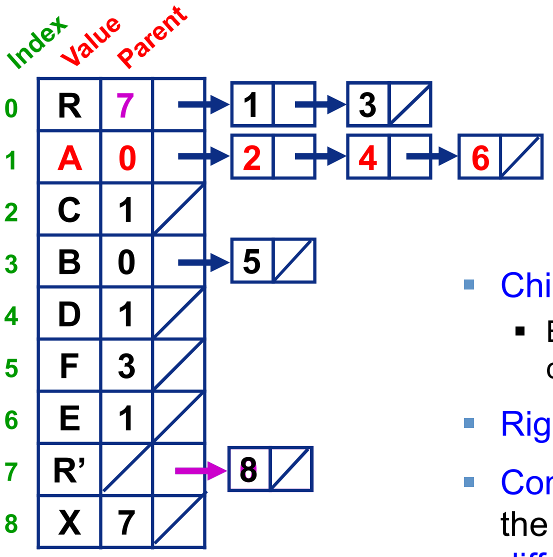
If (K, J)? Yes

If (L, H)? Yes
(path compression)

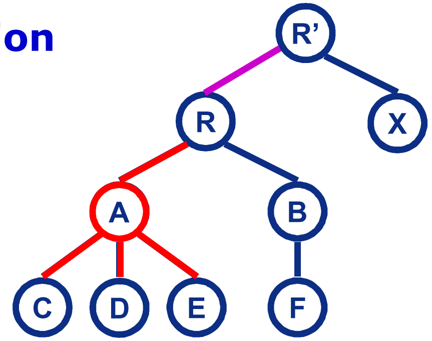


0	1	2	3	4	5	6	7	8	9	10	11
11	3	0	/	/	3	/	6	6	/	9	6
A	B	C	D	E	F	G	H	I	J	K	L

General Tree Implementation
List of Children

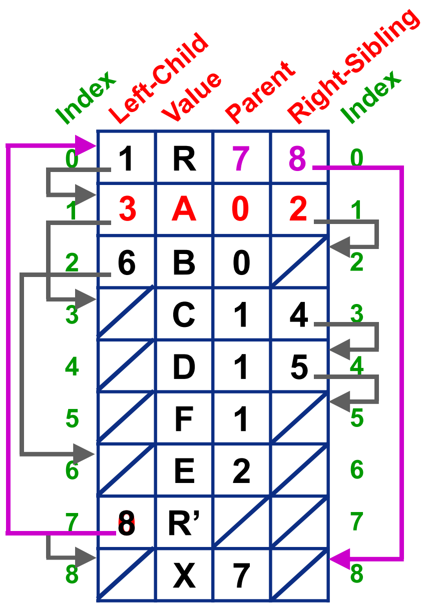


Index	Value	Parent	
0	R	7	→ 1 → 3
1	A	0	→ 2 → 4 → 6
2	C	1	
3	B	0	→ 5
4	D	1	
5	F	3	
6	E	1	
7	R'		→ 8
8	X	7	

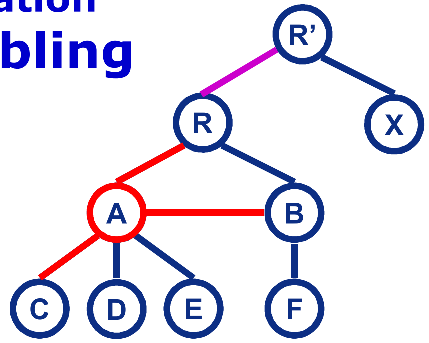


- Children can be found easily
  - Especially for the leftmost child
- Right sibling is more difficult
- Combining trees is difficult if the trees are stored in different array

General Tree Implementation
Left-Child/Right-Sibling

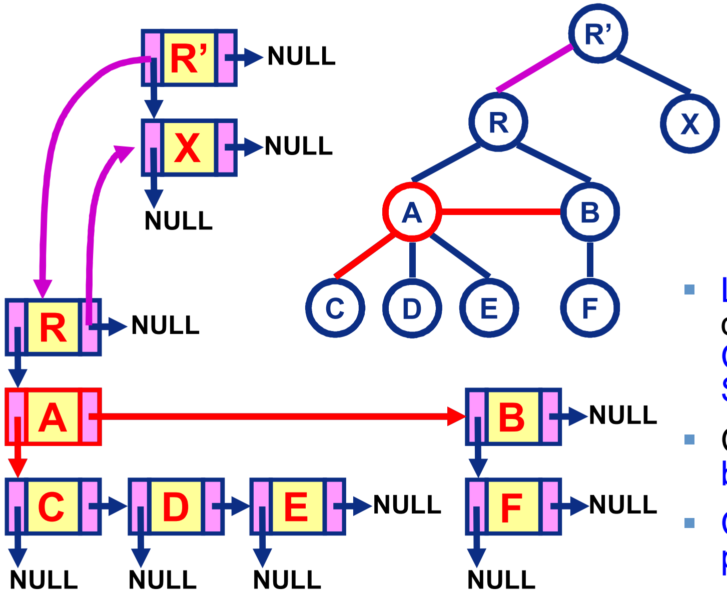


Index	Left-Child	Value	Parent	Right-Sibling	Index
0	1	R	7	8	0
1	3	A	0	2	1
2	6	B	0		2
3		C	1	4	3
4		D	1	5	4
5		F	1		5
6		E	2		6
7	8	R'			7
8		X	7		8



- Improved version of "List of Children"
  - **Right sibling** pointer is added
  - **More space efficient** as each node requires a fixed amount of space
  - Combining trees is difficult if the trees are stored in different array

General Tree Implementation
Dynamic "Left-Child/Right-Sibling"



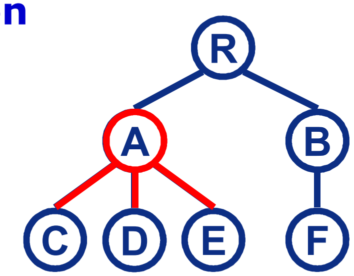
- Linked version of "Left-Child/Right-Sibling"
- Convert as a binary tree
- Cannot find the parent of a node

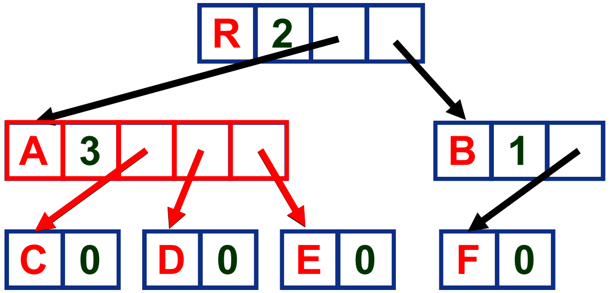
General Tree Implementation
Dynamic Node

- Allocate **variable space** for each node
- Two implementation methods:
  - Array-based List
  - Linked List

General Tree Implementation
Dynamic Node

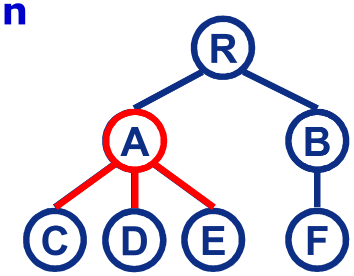
- Allocate an array of child pointers as part of the node
- Assume the number of children is known when the node is created

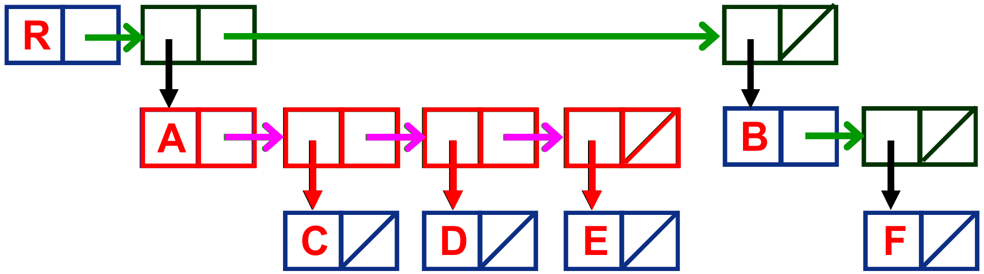




General Tree Implementation
Dynamic Node

- Store a linked list of child pointers with each node
- More flexible (no assumption on number of child) but require more space

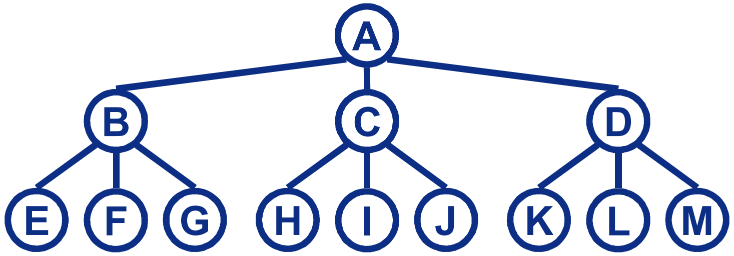




K-ary Trees

- ***K*-ary Trees** are trees with nodes have at most K children
  - e.g. Binary Tree, $\boldsymbol{K} = 2$

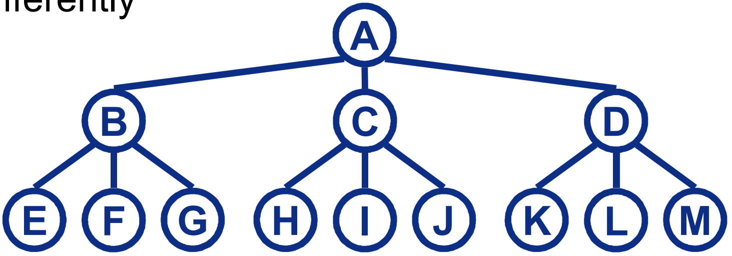
    General Tree, $\boldsymbol{K} = \text{inf}$



3-ary Tree

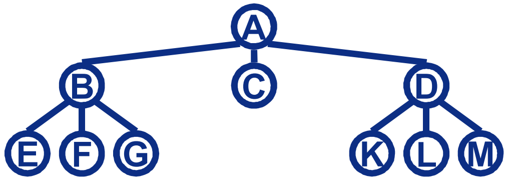
K-ary Trees

- Easy to implement relatively
  - Many properties of binary trees can be extended
  - When $\boldsymbol{K}$ becomes large, the potential number of NULL pointers increase
    - Internal and leaf nodes should be implemented differently

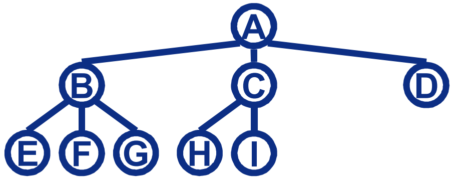


3-ary Tree

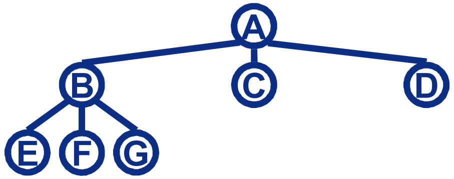
K-ary Trees



Full 3-ary Tree
(not complete)



Complete 3-ary Tree
(not Full)



Full and complete 3-ary Tree

Sequential Tree Implementations

- **Fundamentally different approach** to implementing trees
- Store a series of node values with the minimum information needed to reconstruct the tree structure
  - **Preorder traversal** is used

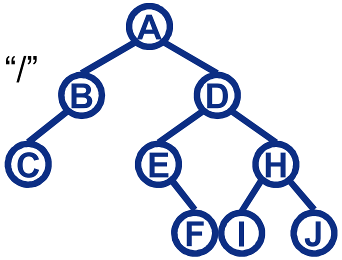
Sequential Tree Implementations

- For **Binary Trees (preorder traversal)**,

  ~~**ABCDEFHIJ**~~
  - Do not have enough information to reconstruct the tree

  A B C / / / D E / F / / H I / / J / /
  - NULL pointer should also be added “/”

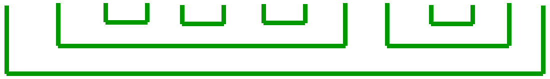
  A’ B’ C / D’ E’ / F H’ I J
  - Add ‘ to the internal node
  - Remove the “/” (NULL pointer) of the leaf node

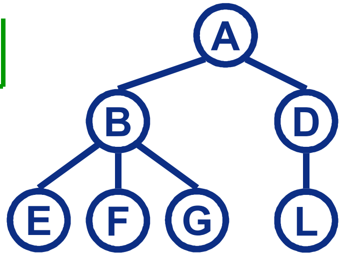


Sequential Tree Implementations

- For **General Tree**,
  - “)” indicates when a node’s child list has come to an end

A B E) F) G)) D L)))





Sequential Tree Implementations

- Space/Time Tradeoff
  - Space saving
    - No pointer is needed
  - Lost the benefit of tree
    - **Tree:** Efficient access O($\log_2 n$)
    - **Sequential Tree:** Sequential access O(n)