Traversals

- standard ways to move across each node of tree
- more variety than list since tree nodes have more successors

List, only two ways:

⟶ head to tail

⟵ tail to head

Tree, each node can send you in two directions (more, for a non-binary tree) first three follow same pattern:

```
void _______Order(TreeNode* TN)
{
    in some order:
      - run recursively on left
          subtree of *TN
      - run recursively on right
          subtree of *TN
      - print out element of *TN
}
```

Only difference (besides name) will be exactly where printing element happens

```
void PreOrder(TreeNode* TN)
{
    cout << TN->elem;
    PreOrder(TN->left);
    PreOrder(TN->right);
}
```

w/ Pre Order, printing of element occurs **before** recursive calls

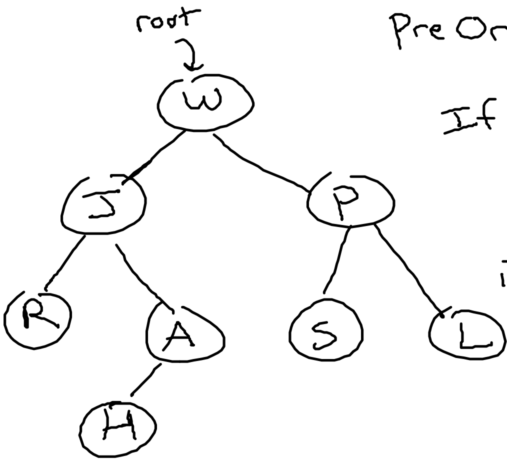
Note the TreeNode* parameter!
We don't want client code having access to TreeNodes (just as the ListNode declaration was hidden from clients in List class). But we do need a TreeNode parameter to make recursion work. The solution is a wrapper function

```
void PreOrder();   // declaration  ]
void PreOrder()    // definition   ] public
{
    PreOrder(root);   <- private member function
}
```

We will often take this approach with trees.

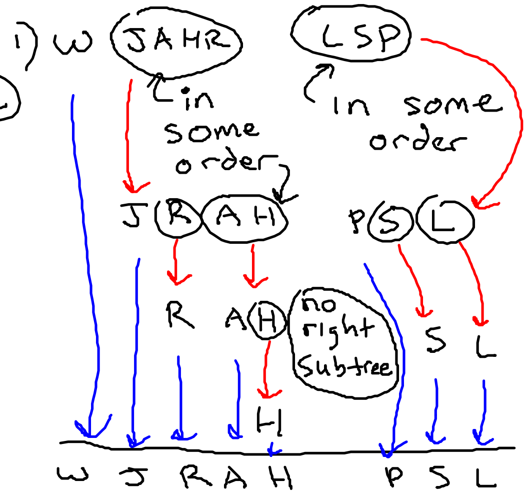
Finally, what if root were NULL? Our code should also work when passed a pointer to a NULL subtree.

```
void PreOrder(TreeNode* TN)
{
    if (TN != NULL)
    {
        cout << TN -> elem;
        PreOrder(TN -> left);
        PreOrder(TN -> right);
    }
}
```



Pre Order: W J R A H P S L

If you can't see it right away, do it recursively yourself:

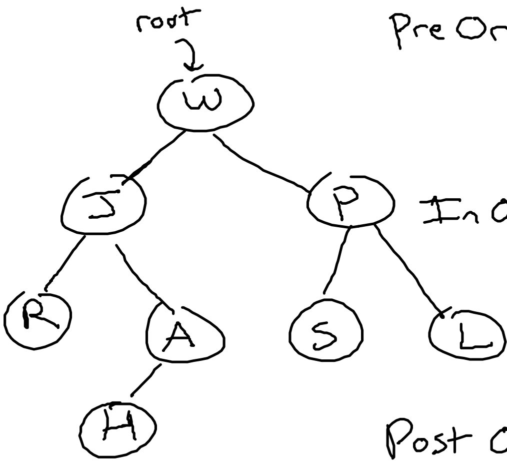


```
void InOrder(TreeNode* TN)
{   if (TN != NULL)
    {
        InOrder(TN->left);
        cout << TN->elem;
        InOrder(TN->right);
    }
}
```

In order prints in between recursive calls

```
void PostOrder(TreeNode * TN)
{   if (TN != NULL)
    {
        PostOrder(TN->left);
        PostOrder(TN->right);
        cout << TN->elem;
    }
}
```

Post order prints after recursive calls



Pre Order: W J R A H P S L

In Order: R J H A W S P L

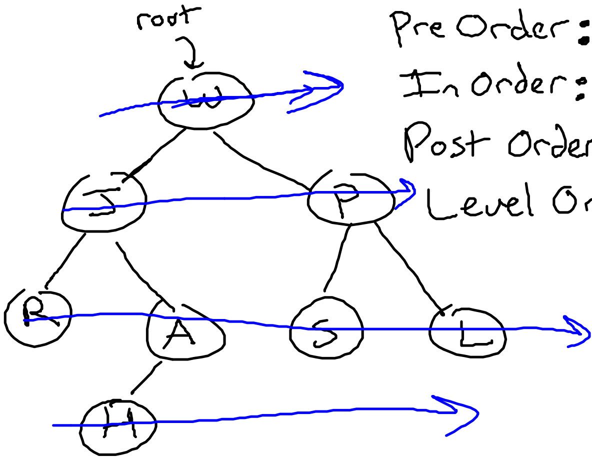
Post Order: R H A J S L P W

— = subtree processed by 2nd-level-deep recursive call

— = subtree processed by 3rd-level-deep recursive call

— = subtree processed by 4th-level-deep recursive call

We can also traverse level-by-level, although that can't be done the same way as the other three traversals, Use a queue to keep order of values in a level.



Pre Order: W J R A H P S L

In Order: R J H A W S P L

Post Order: R H A J S L P W

Level Order: W J P R A S L H

```
void LevelOrder()
{   Queue <TreeNode*> Q;
    Q.Enqueue(root);
    while (!Q.IsEmpty())
    {   TreeNode* temp = Q.Dequeue();
        if (temp != NULL)
        {   cout << temp->elem;
            Q.Enqueue(temp->left);
            Q.Enqueue(temp->right);
        }
    }
}
```

Analysis

Recursive Traversals:

- if you discount actual cost of recursive calls, code is $O(1)$
- So, the running time is $O(1)$ times # of times function is called
- A tree of n nodes has $n+1$ null ptrs, and there is one function call for each ptr-to-a-node and one for each ptr-to-null
- hence $O(2n+1) \equiv O(n)$

Level order

- Again, # of nodes + # of null ptrs is $2n+1$ for any binary tree
- each ptr-to-node and ptr-to-NULL is enqueued and dequeued once.
- Enqueue & dequeue take $O(1)$
- hence $O(2n+1) \equiv O(n)$

i.e. all four traversals are $O(n)$ time

Final note:

You don't need to print, you can do anything else at a node too.

For example, any of the four traversals could be the basis for a search function.