Lecture 9

Tree Data Structure and Tree Traversals

Dr. Yusuf H. Sahin
Istanbul Technical University

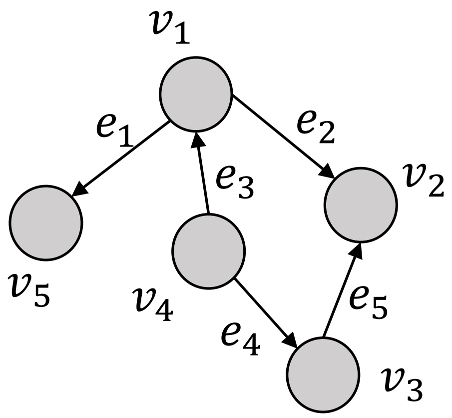
sahinyu@itu.edu.tr

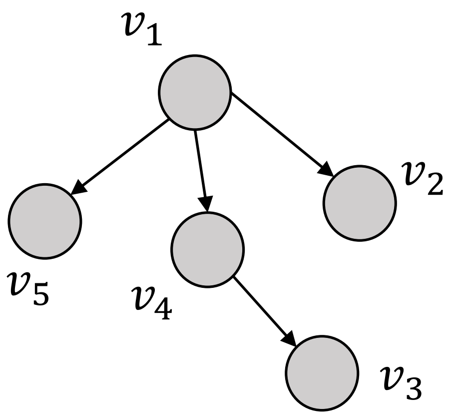
Trees

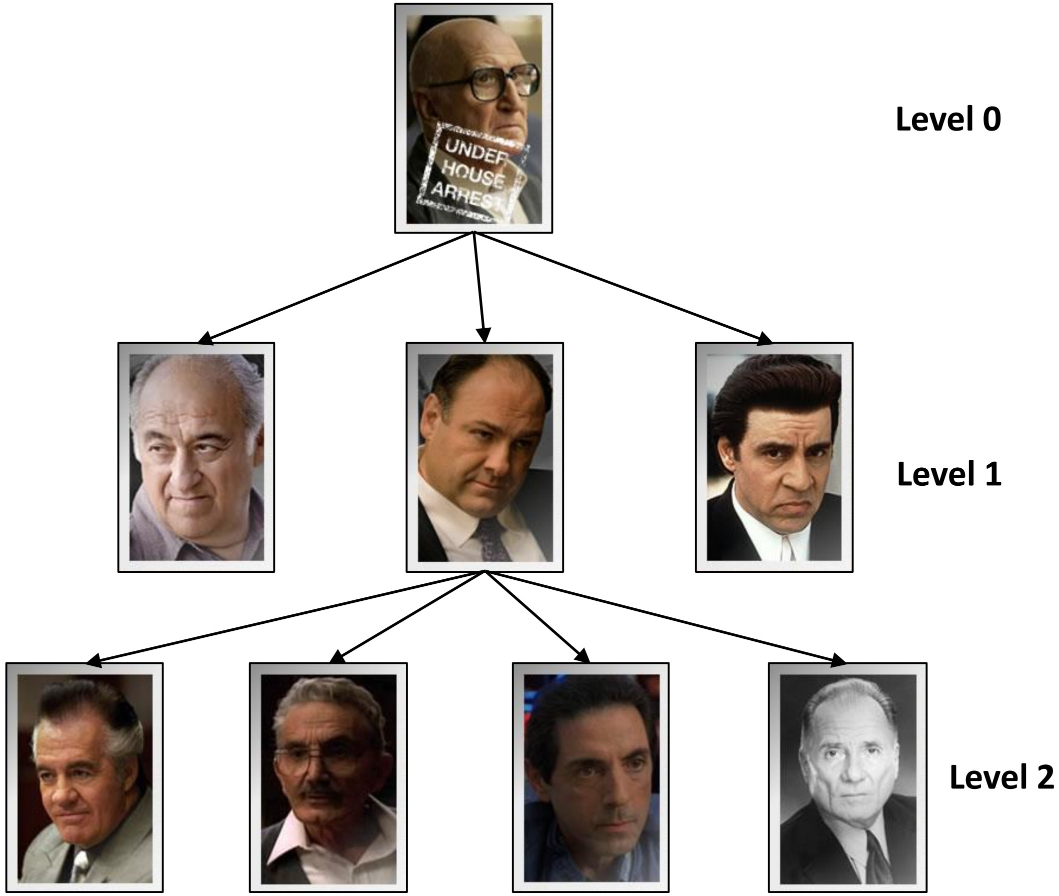
- The most widely used nonlinear data structure.
  - Hierarchical & divided to levels
- A graph is a collection of edges and vertices.

$$G = (V, E)$$

- A tree is a directed graph with no loops.
- If a tree is nonempty, it has a root node which points to other subtrees.
- Each node different than the root node has a unique parent.



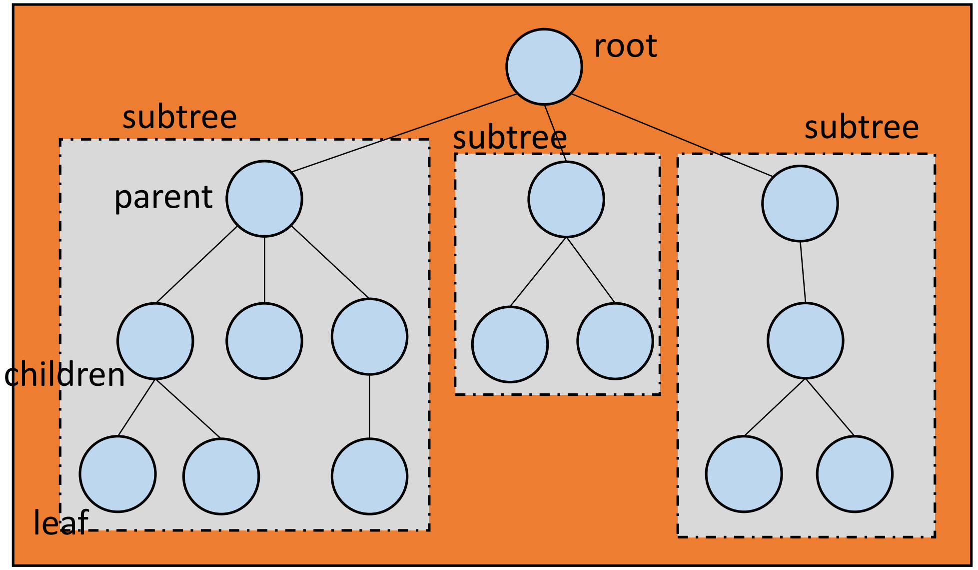




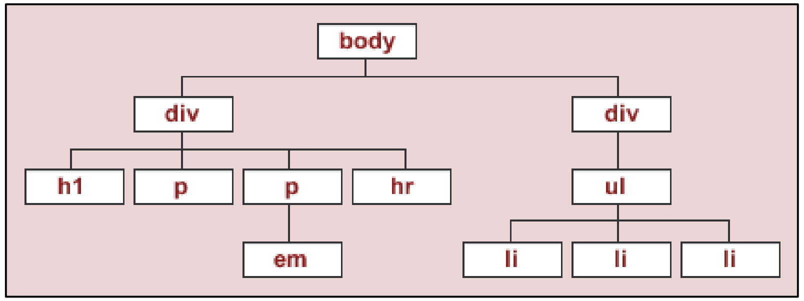
https://www.imdb.com/title/tt0141842/

Trees

- A tree could be recursively defined as the root node which points to other trees.
- **Degree of the tree:** The number of pointed subtress
- **Leaf node:** Node with degree 0
- **Height:** Edge count for the longest path between the root node and a leaf node.
- **Ordered Tree:** A tree where the children of each node has a linear order.

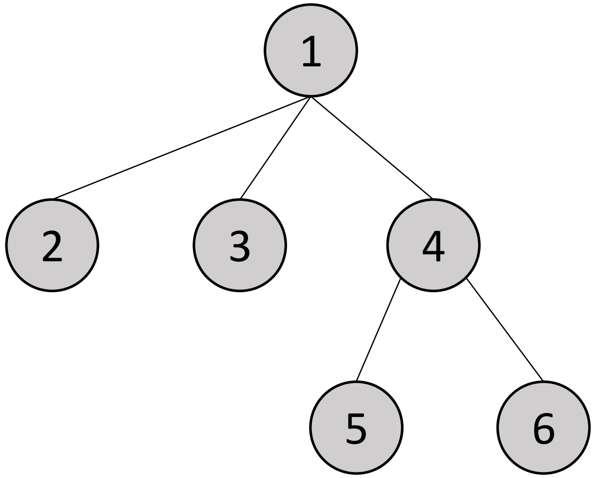



```html
<body>

    <div id="content">
      <h1>Heading here</h1>
      <p>Lorem ipsum dolor sit amet.</p>
      <p>Lorem ipsum dolor <em>sit</em> amet.</p>
      <hr>
    </div>

    <div id="nav">
      <ul>
        <li>item 1</li>
        <li>item 2</li>
        <li>item 3</li>
      </ul>
    </div>

  </body>
```



http://web.simmons.edu/~grabiner/comm244/weekfour/document-tree.html

Trees in C



- A tree should contain pointers to its parent and children.
- A tree is balanced if left and right subtrees are balanced for every subtree.

```c
struct TreeNode {
    int data;
    struct TreeNode* parent;
    struct ListNode* children;
};


struct ListNode {
    struct TreeNode* child;
    struct ListNode* next;
};
```

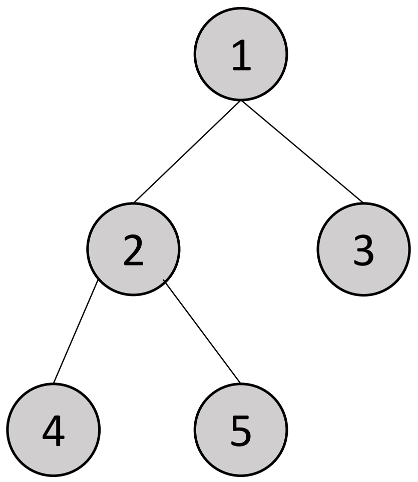
```c
void printTree(struct TreeNode* tree) {
    if (tree == NULL) return;
    printf("%d\n", tree->data);
    struct ListNode* ptr = tree->children;
    while (ptr != NULL) {
        printTree(ptr->child);
        ptr = ptr->next;
    }
}
```

```c
void addFront(struct TreeNode* parent, struct TreeNode* child) {
    struct ListNode* newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
    newNode->child = child;
    newNode->next = parent->children;
    parent->children = newNode;
}
```

```c
struct TreeNode head;
head.parent = NULL;
head.data = 1;
head.children = NULL;

for (int i = 2; i < 5; i++) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = i;
    newNode->parent = &head;
    newNode->children = NULL;
    addFront(&head, newNode);
}

struct TreeNode* firstChild = head.children->child;
for (int i = 5; i < 7; i++) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = i;
    newNode->parent = firstChild;
    newNode->children = NULL;
    addFront(firstChild, newNode);
}

printTree(&head);
```

```
1
4
6
5
3
2
```

Binary Tree



- A binary tree has at most two subtrees: left, right
- Left subtree should be created before the right subtree.

```c
struct treeNode {
    int data;
    struct treeNode* left;
    struct treeNode* right;
    struct treeNode* parent;
};
```

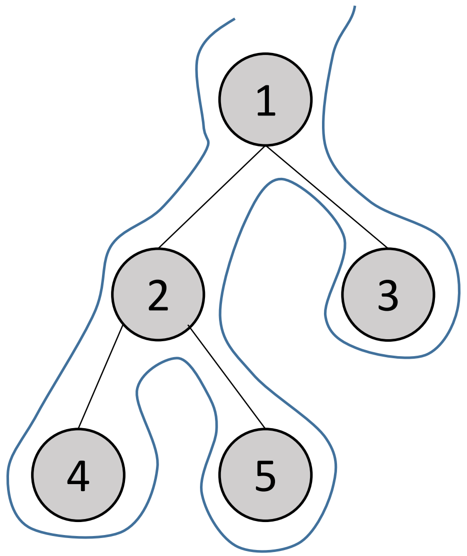
```c
struct treeNode* createNodeWithParent(int data, struct treeNode* parent) {
    struct treeNode* newNode = (struct treeNode*)malloc(sizeof(struct treeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->parent = parent;
    return newNode;
}
```

```c
int main() {
    struct treeNode* root = createNodeWithParent(1, NULL);
    root->left = createNodeWithParent(2, root);
    root->right = createNodeWithParent(3, root);
    root->left->left = createNodeWithParent(4, root->left);
    root->left->right = createNodeWithParent(5, root->left);

    printf("%d\n", root->left->right->parent->parent->data);

    free(root->left->left);
    free(root->left->right);
    free(root->left);
    free(root->right);
    free(root);

    return 0;
}
```

```
1
```

Binary Tree Traversal

- A binary tree could be traversed using three methods.
  - Preorder
  - Inorder
  - Postorder




```c
void preorder(struct treeNode* ptr) {
    if (ptr) {
        printf("%d\n", ptr->data);
        preorder(ptr->left);
        preorder(ptr->right);
    }
}
```

1,2,4,5,3

```c
void inorder(struct treeNode* ptr) {
    if (ptr) {
        inorder(ptr->left);
        printf("%d\n", ptr->data);
        inorder(ptr->right);
    }
}
```

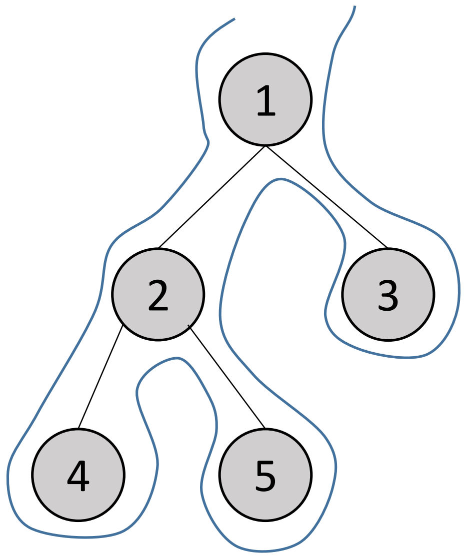
4,2,5,1,3

```c
void postorder(struct treeNode* ptr) {
    if (ptr) {
        postorder(ptr->left);
        postorder(ptr->right);
        printf("%d\n", ptr->data);
    }
}
```

4,5,2,3,1

Binary Tree Traversal

- Finding sum, minimum, maximum and average values could also be done by tree traversal.




```
int preorder_sum(struct treeNode* ptr) {
    if (ptr) {
        return preorder_sum(ptr->left) +  preorder_sum(ptr->right) + ptr->data;
    }
}
```

15

```
int preorder_max(struct treeNode* ptr) {
    if (ptr == NULL) return INT_MIN;
    int leftMax = preorder_max(ptr->left);
    int rightMax = preorder_max(ptr->right);
    int maxVal = ptr->data;
    if (leftMax > maxVal) maxVal = leftMax;
    if (rightMax > maxVal) maxVal = rightMax;
    return maxVal;
}
```

5

```
int preorder_min(struct treeNode* ptr) {
    if (ptr == NULL) return INT_MAX;
    int leftMin = preorder_min(ptr->left);
    int rightMin = preorder_min(ptr->right);
    int minVal = ptr->data;
    if (leftMin < minVal) minVal = leftMin;
    if (rightMin < minVal) minVal = rightMin;
    return minVal;
}
```

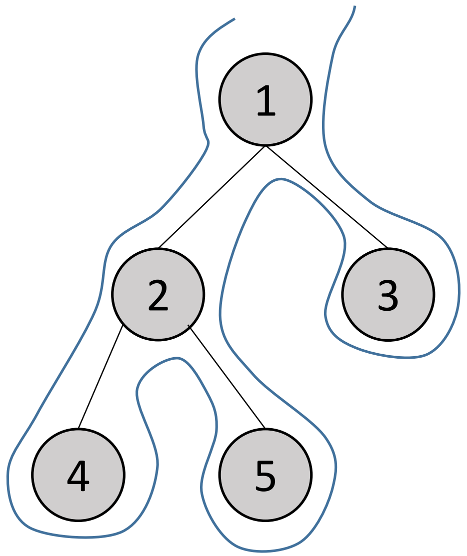
1

```
int count_nodes(struct treeNode* ptr) {
    if (ptr == NULL) return 0;
    return 1 + count_nodes(ptr->left) + count_nodes(ptr->right);
}

float preorder_average(struct treeNode* ptr) {
    int sum = preorder_sum(ptr);
    int nodeCount = count_nodes(ptr);
    if (nodeCount == 0) return 0;
    return (float)sum / nodeCount;
}
```

3

Binary Tree Traversal

- The reference to the result variable could also be kept as a function parameter.




```c
void preorder_sum(struct treeNode* ptr, int* sum) {
    if (ptr) {
        *sum += ptr->data;
        preorder_sum(ptr->left, sum);
        preorder_sum(ptr->right, sum);
    }
}
```

15

```c
void preorder_count(struct treeNode* ptr, int* count) {
    if (ptr) {
        (*count)++;
        preorderCount(ptr->left, count);
        preorderCount(ptr->right, count);
    }
}
```

5

```c
void preorder_max(struct treeNode* ptr, int* max) {
    if (ptr) {
        if (ptr->data > *max) {
            *max = ptr->data;
        }
        preorder_max(ptr->left, max);
        preorder_max(ptr->right, max);
    }
}
```

5

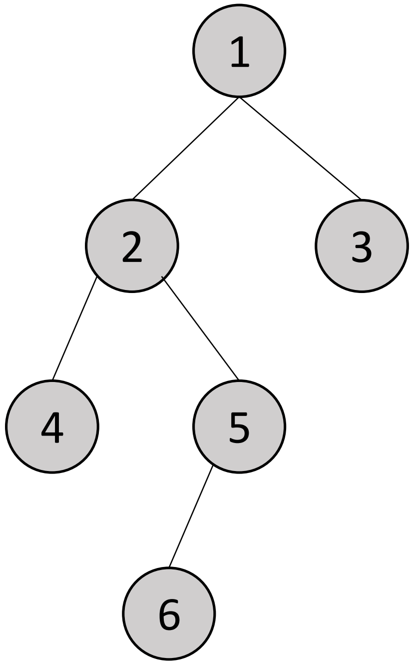
```c
void preorder_min(struct treeNode* ptr, int* min) {
    if (ptr) {
        if (ptr->data < *min) {
            *min = ptr->data;
        }
        preorder_min(ptr->left, min);
        preorder_min(ptr->right, min);
    }
}
```

1

```c
int sum = 0;
preorderSum(&root, &sum);

int max = INT_MIN;
preorderMax(&root, &max);

int min = INT_MAX;
preorderMin(&root, &min);
```

Depth & Height

- For a point p,
  - the count of its ancestors defines its depth.
  - the count of its below layers defined its height




```c
int depth(struct treeNode* node) {
    if (node->parent == NULL) {
        return 0;
    } else {
        return 1 + depth(node->parent);
    }
}
```

```c
int height(struct treeNode* node) {
    if (node == NULL) {
        return 0;
    } else if (node->left == NULL) {
        return height(node->right) + 1;
    } else if (node->right == NULL) {
        return height(node->left) + 1;
    } else {
        int leftHeight = height(node->left);
        int rightHeight = height(node->right);
        if (leftHeight > rightHeight)
            return leftHeight + 1;
        else
            return rightHeight +1;
    }
}
```