Topic 3: Trees and Their Implementations

Read: Chpt. 4, Weiss

Q: Why study trees? Standard linear data structures (arrays, lists, stacks, …) are no longer sufficient for many applications; most advance ADTs are implemented using tree-based data structures.

Recursive definition of (free) tree:
Let \( T \) be a set with \( n \geq 0 \) elements.
(i) If \( n = 0 \), \( T \) is an empty tree,
(ii) If \( n > 0 \), then there is a distinct element \( r \), called the root of \( T \), such that \( T - \{r\} \) can be partitioned into 0 or more disjoint subsets \( T_1, T_2, \ldots \), where each of these subsets also forms a tree.

Basic Concepts of Trees:
Elements of \( T \) are nodes in the tree \( T \).
Each subset \( T_i \) is a subtree of \( r \).
The roots of the subtrees of \( r \) are children of \( r \).
The root \( r \) is the parent of the roots of its subtrees.
Nodes with the same parents are siblings.
Nodes with no children are leaves.
Degree of a node is the number of children of the node.
A path of length \( k \) (in a tree) from node \( x \) to node \( y \) is a sequence of \( k+1 \) nodes \( x = n_0, n_1, n_2, \ldots, n_k = y \) such that \( n_i \) is the parent of \( n_{i+1} \) for all \( i = 0, 1, \ldots, k-1 \).

If there is a path from \( x \) to \( y \), then \( x \) is an **ancestor** of \( y \) and \( y \) is a **descendant** of \( x \).

For any given node \( x \), the **depth** (level) of \( x \) is the length of the (unique) path from the root to \( x \), and the **height** of \( x \) is the length of a longest path from \( x \) to any leaf.

*The height of a tree* is the height of its root.  
*The depth of a tree* is the maximum depth of its nodes.

**Remarks:**

1. There is a unique path from the root to each node in the tree.
2. There is a path of zero length from any node to itself.
3. Depth of the root of a tree is 0 and height of a leaf is also 0.
4. Height of tree = Depth of tree.
5. For convenience, the height of an empty tree is defined to be \(-1\).
Graphical Representation of Tree:

A is the \textit{root} of tree T, 
D is the \textit{parent} of F and G, 
F and G are children of D, 
Degree of F is 2, 
B, C, D are \textit{siblings}, 
C, E, G, H, I are \textit{leaves}, 
D is an \textit{ancestor} of I and I is a \textit{descendant} of D, 
(A,D,F,H) is a \textit{path} of length 3, 
H is of \textit{height} 0 but \textit{depth} 3. 
Height (depth) of the tree is 3.
Some Important Classes of Trees:
1. k-ary trees, $k \geq 2$:
   A k-tree is an unordered tree with each node having at most $k$ children, $k \geq 2$.

2. Binary tree:
   A binary tree is an ordered 2-ary tree.

**Recursive Definition of Binary Tree:**
Let $T$ be a set with $n \geq 0$ elements. $T$ is a binary tree iff

(i) $T$ is empty, or

(ii) if $T$ is not empty, $T$ has a root $r$ such that $T-\{r\}$ can be partitioned into two disjoint binary trees $T_L$ and $T_R$, called the *left subtree* and *right subtree* of $r$.

Example: A binary trees.
3. Extended binary tree:

An **extended binary tree** (EBT) $T_E$ of a given binary tree $T$ with $n$ nodes, $n \geq 0$, is a binary tree constructed from $T$ such that each (original) node in $T$ will have exactly two children in $T_E$. The (original) nodes in $T$ are called **internal nodes** and the (new) nodes in $T_E - T$ are called **external nodes** of the EBT.

**Special Case:**
When $n = 0$, the EBT of an empty tree $T$ consists of a single external node.

**Example:**

```
T
Ø → T_E
```

Diagram: [Binary tree representation]
Given an extended binary tree $T_E$ with $n$ internal nodes.

**Definition:**

*External path length* of $T_E$:

$E(T_E) =$ sum of the level numbers (depth) of all external nodes.

*Internal path length* of $T_E$:

$I(T_E) =$ sum of the level numbers (depth) of all internal nodes.

**Observations:**

1. $\#\text{external nodes} = \#\text{internal nodes} + 1$

2. $E(T_E) = I(T_E) + 2n$.

3. Average distance to an external node $= \frac{E(T_E)}{n + 1}$.

4. Average distance to an internal node $= \frac{I(T_E)}{n}$.

**Q.** What kind of extended binary trees will have maximum (minimum) internal and external path length?
4. Complete binary tree:
   A complete binary tree $T$ with height $h$ is a binary tree such that
   (1) Each node on levels 0 to $h-2$ must have exactly two children, and
   (2) Leaves on level $h$ are left-justified.

Examples:
(a) Not complete binary tree:

(b) Complete binary tree:
5. Balanced binary tree:

A balanced binary tree $T$ is a binary tree such that for any node $x$ in the tree, the height of its left subtree differs by no more than one from the height of its right subtree. Hence, $|h(T_L(x)) - h(T_R(x))| \leq 1$, where $h(T_L(x))$, and $h(T_R(x))$, are the height of the left, and right, subtree rooted at $x$, respectively.

**Example:**
Balanced binary tree showing height of nodes:

```
   3
  /  \
 2    1
 / \   /
0   1  0
   0
```

Not a balanced binary tree showing height of nodes:

```
   4
  /  \
 3    1
 /   /
0   2  0
    /
    1
     /
     0
```
6. Skew tree:
A skew tree $T$ is a binary tree such that each non-leaf node in $T$ is having exactly one child.

**Example:** Some skew trees.

7. Full binary tree:
A full binary tree of height $h$ is a binary tree such that each node on levels $0$ to $h - 1$ is having exactly two children.
Hence, $T$ has exactly $2^{h+1} - 1$ nodes.

**Example:** A full binary tree with $h = 2$. 
**Observation:**
A full tree is a complete binary tree and a complete binary tree is a balanced binary tree.

When implementing an ADT using a (binary) tree, either non-leaf or leaf-nodes can be used to hold the data objects of the ADT and the performance of an ADT operation will usually depend on the height of the tree.

**Nodes, Leaves, and Heights of Binary Trees:**
Given a non-empty binary tree $T$.
1. $T$ with height $h$, $h \geq 0$:
   - Min # nodes = $h+1$.
   - Max # nodes = $2^{h+1} - 1$.
2. $T$ with height $h$, $h \geq 0$:
   - Min # leaves = 1.
   - Max # leaves = $2^h$.
3. $T$ with $n$ nodes, $n \geq 1$:
   - Min height = $\lceil \log_2 n \rceil$.
   - Max height = $n-1$.
4. $T$ with $n$ nodes, $n \geq 1$:
   - Min # leaves = 1.
   - Max # leaves = $\lceil n/2 \rceil$.
5. $T$ with $m$ leaves, $m \geq 1$:
   - Min height = $\lceil \log_2 m \rceil$.
   - Max height = $\infty$.
6. $T$ with $m$ leaves, $m \geq 1$:
   - Min # nodes = $2m-1$.
   - Max # nodes = $\infty$.

**HW:** Prove the above results.
Can you generalize them to $k$-ary trees, $k > 2$?
Traversing a Binary Tree:
1. To explore the underlying hierarchical structures.
2. To retrieve information stored in nodes.
3. To store and restore the topological structure of a tree.

Basic Binary Tree Traversal Algorithms:
1. *Preorder traversal:*
   Traverse/retrieve root,
   Traverse left subtree recursively in preorder,
   Traverse right subtree recursively in preorder.

2. *Postorder traversal:*
   Traverse left subtree recursively in postorder,
   Traverse right subtree recursively in postorder,
   Traverse/retrieve root.

3. *Inorder traversal:*
   Traverse left subtree recursively in inorder,
   Traverse/retrieve root,
   Traverse right subtree recursively in inorder.

4. *Level-order traversal:*
   Starting at level 0, traverse the nodes on each level from left to right level by level.
Example: Binary tree traversals.

![Binary tree](image)

- **Preorder**: A B E F D G H
- **Postorder**: E F B H G D A
- **Inorder**: E B F A D H G
- **Level-order**: A B D E F G H

**HW**: Implement the above tree traversal algorithms.

**More on binary tree traversals:**

Observe that for a given binary tree T, it is easy to traverse T.

**Q**: If the traversal of a binary tree T is given, can we reconstruct the binary tree T?

No!
Consider the following binary trees:

```
    A
   / \
  R   A
```

Observe that preorder, postorder, and level-order traversals for both trees are given by AB, BA, and AB, respectively. Hence, these two binary trees are indistinguishable if only one of the above tree traversals is given!

Q: How about inorder traversal?

Consider the following binary trees:

```
    R
   / \   A
  A   B
```

Again, these two binary trees are indistinguishable if only their inorder traversal is given!
Q: Can we reconstruct the binary tree T from its traversal(s)?
   Must know the root, the left, and the right subtrees.

If we are given any one of the following pairs of traversals of T, T can be reconstructed if exists.
   1. Preorder and inorder traversals.
   2. Postorder and inorder traversals.
   3. Level-order and inorder traversals.

Remark: Preorder and postorder, level order and preorder, and level-order and postorder traversals will not work!

HW. Can you reconstruct a binary tree T having the following pairs of traversals? Algorithms?

Preorder: B A D C E F G I H
Inorder: A F E G C D I B H

Preorder: A C D H I E B F G
Inorder: D I H C E A B G F

Preorder: A C D H I E B F G
LevelOrder: A C B D E F H G I

Postorder: A K J G B D H F I E C
Inorder: A D K G J B C F H E I
Consider given a pair of preorder and inorder traversals of a binary tree T.

Q: How do we reconstruct T if exists?
   Scan preorder traversal from left to right to determine root followed by scanning inorder traversal to determine left and right subtree.

Example: Let preorder = ABDEC and inorder = BEDAC.
General Tree Traversals:
1. Preorder traversal:
   Traverse/retrieve root,
   Traverse subtree $T_1$ recursively in preorder,
   Traverse subtree $T_2$ recursively in preorder,
   \[ \cdots \]
   Traverse subtree $T_k$ recursively in preorder.

2. Postorder traversal:
   Traverse subtree $T_1$ recursively in postorder,
   Traverse subtree $T_2$ recursively in postorder,
   \[ \cdots \]
   Traverse subtree $T_k$ recursively in postorder,
   Traverse/retrieve root.

3. Level order traversal:
   Starting at level 0, traverse the nodes on each level from left to right level by level.

4. Inorder traversal:
   Since the concept of *in-between* is not well-defined for a general tree, we don't usually use inorder traversal for general tree!
Example: General tree traversals.

Preorder: A B E C D F H I J G
Postorder: E B C H I J F G D A
Level order: A B C D E F G H I J
**ADT:** Binary tree.

**UML diagram for the class *BinaryTree*:**

```
Binary Tree

root
leftTree
rightTree

createTree()
destroyTree()
isEmpty()
getRootData()
setRootData()
attachLeftTree()
attachRightTree()
detachLeftTree()
detachRightTree()
getLeftTree()
getRootTree()
preorderTraversal()
postorderTraversal()
inorderTraversal()
levelorderTraversal()
...
```
Binary Tree Implementations:
Depend on speed and memory requirements.

Movement in tree:

- \textbf{up} (require parent info)
- \textbf{down} (require children info)
- \textbf{sideway} (require siblings info)

Q: Array, pointer, or hybrid implementation?

1. Array-based Implementation of (Complete) Binary Tree:
   Given a complete binary tree $T$ with $n$ nodes. $T$ can be implemented using an array $A[0:n-1]$ such that
   - (1) Root of $T$ at $A[0]$,
   - (2) Parent of a node $A[i]$ at $A[(i-1)/2]$ if exists,
   Observe that, for $n \geq 1$, $A[i]$ is a leaf iff $2i \geq n-1$. 
Example:

```
+--- A ---+
|         |
+--- B ---+   +--- C ---+
|         |       |       |
+--- D ---+   +--- F ---+
|         |       |       |
    H      |       |       |
```

last = 9

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

**Advantages:** Fastest, $\Theta(1)$ time, in accessing parent and children locations.

**Disadvantage:** Only useful when tree is complete (or “almost complete”); otherwise, very memory intensive. For a tree with height $h$, it requires an array of size $2^{h+1} - 1$. Hence, a skew tree with 10 nodes with height 9 will require an array of size $2^{10} - 1 = 1023$ for its implementation.

**HW:** What if root of T is stored at A[1] instead? For a node stored in A[i], $i \geq 1$, compute its parent, left child, and right child location.
2. Children-Pointer Implementation of Binary Tree:

**NodeType:**

<table>
<thead>
<tr>
<th>item</th>
<th>Lchild</th>
<th>Rchild</th>
</tr>
</thead>
</table>

The external pointer root points at the root r of the tree. If the tree is empty, root is NULL; otherwise, root→leftChildPtr (root→rightChildPtr) points to the root of the left (right) subtree of r.
TreeNode Class:

typedef string TreeItemType;

class TreeNode  // node in the tree
{
private:
    TreeNode() {};  
    TreeNode(const TreeItemType& nodeItem,
             TreeNode *left = NULL,
             TreeNode *right = NULL) :
        item(nodeItem), leftChildPtr(left),
        rightChildPtr(right) { }
    TreeItemType item;  // data portion
    TreeNode *leftChildPtr;  // pointer to left child
    TreeNode *rightChildPtr; // pointer to right child

    friend class BinaryTree;  // friend class
};  // end TreeNode class

Friends:

In C++, function/class can be declared as a "friend" to another class C. Doing so will allow the function/class to access all of the private and protected members of the class C.
3. Parent-Pointer Implementation of Tree:

Observe that every node in a tree can have at most one parent. Hence, we can implement any tree by using the following hybrid data structure.

**NodeType:**

```
parent

item
```

**Example:**

![Tree Diagram](image)

**Remark:** The array of pointers will be used to access the nodes of the tree.
**General Tree Implementations:**
1. k-tree implementation, k > 2:
   (a) Array-based Implementation of (Complete) k-Tree:
   Given a complete k-tree T with n nodes. T can be implemented using an array $A[0:n-1]$ such that
   (1) Root of T at $A[0]$,
   (2) Parent of a node $A[i]$ at $A[(i-1)/k]$ if exists,
   (3) The jth child of a node $A[i]$, $1 \leq j \leq k$, at $A[ki+j]$ if exist.
   Observe that, for $n \geq 1$, $A[i]$ is a leaf iff $ki \geq n-1$.

**Example:**

![Diagram of a tree with nodes A, B, C, D, E, F, G, H, I and array representation]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>

last = 8
(b) Children-Pointer Implementation:
Same as before except using $k$ children pointers.

NodeType:

```
+-------+-------+
|   item   |
+-------+-------+
| child1 |   ...  |
+-------+-------+
|       | childk |
+-------+-------+
```

2. Parent-Pointer Implementation:
Same as above.

3. Left-Child List-of-Siblings Implementation:
For each node $N$ in $T$, the leftmost child of $N$, say $x$, will become the only child of $N$ and the siblings of $x$ will be linked together to form a chain of siblings of $x$.

NodeType:

```
+-------+-------+
|   item   |
+-------+-------+
|  child  | sibling |
+-------+-------+
```

```
Example:

Left-Child List-of-Siblings Implementation:

Remark: Based on applications, siblings can be linked together using a suitable linked structure such as singly, doubly, or circular doubly linked list.

9/10/14