Lecture 13
Trees and Tree Algorithms

EECS 281: Data Structures & Algorithms

Informal Definition: Tree

Mathematical abstraction that plays a central role in the design and analysis of algorithms

- Build and use explicit data structures that are concrete realizations of trees
- Describe the dynamic properties of algorithms

Concrete Implementation

- A node contains some information, and points to its left child node and right child node
- Efficient for moving *down* a tree from parent to child

Node in a Binary Tree

```
template <typename KEY>
struct Node {
    KEY datum;
    Node *left, *right;
};
```

Formal Definition: Tree

Tree: set of nodes storing elements in a parent-child relationship with the following properties:

- T has a special node r, called the root of T, with no parent node;
- Each node v of T, such that $v \neq r$, has a unique parent node u

Note: A tree *can* be empty [CLRS]

Some Tree Terminology

- Root: “top-most” vertex in the tree
  - The initial call
- Parent/Child: direct links in tree
- Siblings: children of the same parent
- Ancestor: predecessor in tree
  - Closer to root along path
- Descendent: successor in tree
  - Further from root along path

Some Tree Terminology

- Internal node: a node with children
- External (Leaf) node: a node without children
- Ordered Tree: linear ordering for the children of each node
- Binary Tree: ordered tree in which every node has at most two children

Some Tree Terminology

Depth:

depth(empty) = 0;

depth(node) = depth(parent) + 1;

Height:

height(empty) = 0;

height(node) = max(height(children)) + 1;

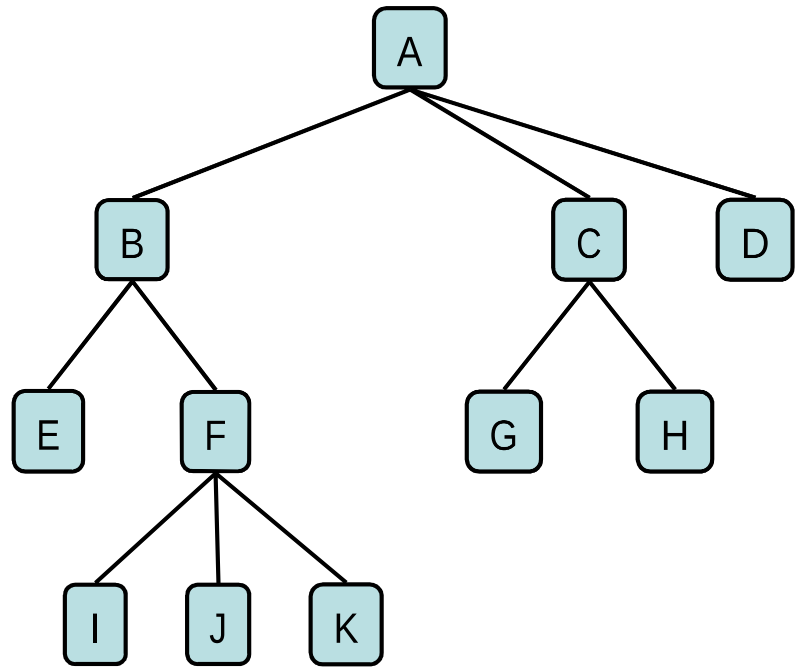
Max Height/Depth:

maximum height/depth of tree’s nodes

Tree Terminology

- Root:
- Internal nodes:
- External nodes (a.k.a. leaf):

- Max depth:
- Label one subtree
- Is this a binary tree?



Proper Binary Tree Property

Definition: ***proper binary tree***

- A binary tree where
  - All external nodes have zero children
  - All internal nodes have two children

Complete Binary Tree Property

Definition: ***complete binary tree***

- A binary tree with depth *d* where
  - Tree depths 1, 2, …, *d* have the max number of nodes possible
  - All internal nodes are to the left of the external nodes at depth *d*
  - That is, all leaves are leftmost at depth *d*

Proper and Complete

- Draw a binary tree that is neither proper nor complete
- Draw a binary tree that is proper, but not complete
- Draw a binary tree that is complete, but not proper
- Draw a binary tree that is both proper and complete

Binary Tree Implementation

Array Binary Tree Implementation

- Root at index 1
- Left child of node *i* at 2 * *i*
- Right child of node *i* at 2 * *i* + 1
- Some indices may be skipped
- Can be space prohibitive for sparse trees

Binary Tree Implementation

Complexity of array implementation

• Insert key (best case)	$O(\ )$
• Insert key (worst case)	$O(\ )$
• Delete key (worst case)	$O(\ )$
• Parent	$O(\ )$
• Child	$O(\ )$
• Space (best case)	$O(\ )$
• Space (worst case)	$O(\ )$

Binary Tree Implementation

- Pointer-based binary tree implementation

```
<typename T>
struct Node {
    T datum;
    Node *left, *right;
};
```

- A node contains some information, and points to its left child node and right child node
- Efficient for moving *down* a tree from parent to child

Binary Tree Implementation

Complexity of pointer implementation

• Insert key (best case)	$O(\ )$
• Insert key (worst case)	$O(\ )$
• Delete key (worst case)	$O(\ )$
• Parent	$O(\ )$
• Child	$O(\ )$
• Space (best case)	$O(\ )$
• Space (worst case)	$O(\ )$

Trees: Data Structures

- Another way to do it (not common)

```
<typename T>
struct Node {
    T datum;
    Node *parent, *left, *right;
};
```

- If node is root, then **`*parent`** is `nullptr`
- If node is external, then `*left` and `*right` are `nullptr`

Translating General Trees into Binary Trees

T: General tree

T': Binary tree

Intuition:

- Take set of siblings $\{v_1, v_2, \ldots, v_k\}$ in *T* that are children of *v*
- v_1 becomes left child of *v* in *T'*
- $v_2, \ldots v_k$ become chain of right children of v_1 in *T'*
- Recurse from v_2

Left: new “generation”; Right: sibling

Summary of Trees

- Trees have intuitive definitions
  - Think family tree
- Trees can be implemented
  - As an array (vector)
  - With linked structs (or classes)
- General trees can be converted to binary trees

Tree Traversal

Systematic method to process every node in a tree

- Preorder:
  - Visit node,
  - Recursively visit left subtree,
  - Recursively visit right subtree
- Inorder:
  - Recursively visit left subtree,
  - Visit node,
  - Recursively visit right subtree

Tree Traversal

Systematic method to process every node in a tree

- Postorder:
  - Recursively visit left subtree,
  - Recursively visit right subtree,
  - Visit node
- Level order:
  - Visit nodes in order of increasing depth in tree

Recursive Implementation

Preorder

```
Algorithm preorder(T, v)
    visit node v
    for (node c : v.children())
        preorder(T, c)
```

Recursive Implementation

Postorder

```
Algorithm postorder(T, v)
    for (node c : v.children())
        postorder(T, c)
    visit node v
```

Recursive Implementation

Inorder (assumes tree with left/right)

```
Algorithm inorder(T, v)
    inorder(T, leftchild(v))
    visit node v
    inorder(T, rightchild(v))
```

Recursive Implementations

```
void preorder(Node *p) {
    if (!p) return;
    visit(p->datum);
    preorder(p->left);
    preorder(p->right);
}
```

```
void inorder(Node *p) {
    if (!p) return;
    inorder(p->left);
    visit(p->datum);
    inorder(p->right);
}
```

```
void postorder(Node *p) {
    if (!p) return;
    postorder(p->left);
    postorder(p->right);
    visit(p->datum);
}
```

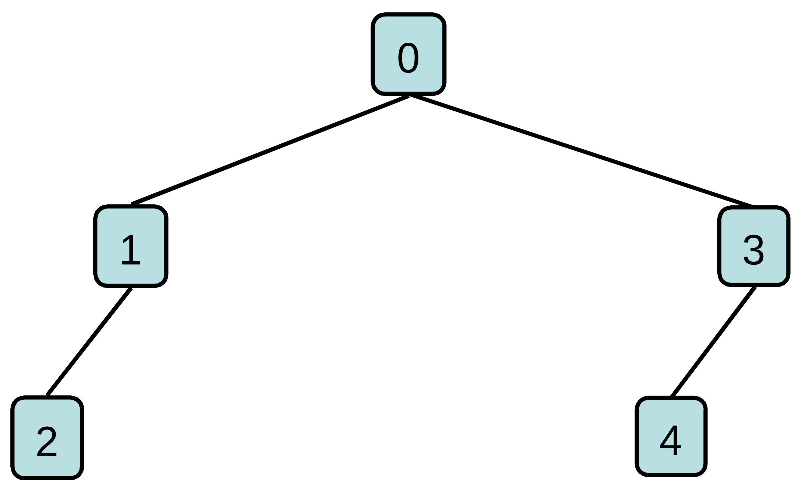
Summary of Tree Algorithms

- Definitions of depth and height
- Methods of tree traversal
  - Preorder
  - Inorder
  - Postorder
  - Level order (breadth-first search)

All are depth-first search

Exercise

- In what order are nodes visited?
  - Preorder
  - Inorder
  - Postorder
  - Level order



Exercise

- Draw the binary tree from traversals
- Preorder: 7 3 6 9 8 13 27
- Inorder: 3 6 7 8 9 13 27

Exercise

- Write a function to do a level order traversal, printing the datum at each node

```
void levelorder(Node *p) {




}
```