Overview

- Review basic abstract data structures
  - Sets
  - Lists
  - Trees
  - Graphs
- Review basic concrete data structures
  - Linked-List and variants
  - Trees and variants
- Examine key properties
- Discuss usage for solving important problems (search, sort, selection).

Sets and Multisets

- Common operations
  - Fixed Sets
    - Contains (search)
    - Is empty
    - Size
    - Enumerate
  - Dynamic Sets add:
    - Add
    - Remove
- Other operations (not so common)
  - Intersection
  - Union
  - Sub-set
  - Note, these can, and probably should, be implemented statically (outside of the class).

Set – Language Support

- .NET Framework Support (C#, VB, C++, …)
  - IEnumerable interface
  - ICollection interface
- Java Framework Support
  - Collection
  - Set
- STD library (C++)
  - Set and Multiset classes and their iterators.
List

- Common Queries
  - Enumerate
  - Number of items in the list
  - Return element at index $i$.
  - Search for an item in the list (contains)

- Common Commands
  - Add element
  - Set element at index $i$.
  - Remove element?
  - Insert before index $i$?

List – Language Support

- Arrays – fixed size.
- .NET Framework Support (C#, VB, C++, …)
  - IList interface
  - List<T> class
- Java Framework Support
  - List interface
  - ArrayList<T> and Vector<T> classes
- STD library (C++)
  - std::vector<T> class.

Concrete Implementations

- Set
  - What might you use to implement a concrete set?
  - What are the pro’s and con’s of each approach?

- List
  - Other than arrays, could you implement a list with any other data structure?

Rooted Trees

- A tree is a collection of nodes and directed edges, satisfying the following properties:
  - There is one specially designated node called the root, which has no edges pointing to it.
  - Every node except the root has exactly one edge pointing to it.
  - There is a unique path (of nodes and edges) from the root to each node.
Basic Tree Concepts

- Node – user-defined data structure that contains pointers to data and pointers to other nodes:
  - Root – Node from which all other nodes descend
  - Parent – has child nodes arranged in subtrees.
  - Child – nodes in a tree have 0 or more children.
  - Leaf – node without descendants
  - Degree – number of direct children a tree/subtree has.

Height and Level of a Tree

- Height – # of edges on the longest path from the root to a leaf.
- Level – Root is at level 0, its direct children are at level 1, etc.
- Recursive definition for height:
  \[ 1 + \max(\text{height}(T_L), \text{height}(T_R)) \]

Rooted Trees

- If an edge goes from node \( a \) to node \( b \), then \( a \) is called the parent of \( b \), and \( b \) is called a child of \( a \).
- Children of the same parent are called siblings.
- If there is a path from \( a \) to \( b \), then \( a \) is called an ancestor of \( b \), and \( b \) is called a descendant of \( a \).
- A node with all of its descendants is called a subtree.
- If a node has no children, then it is called a leaf of the tree.
- If a node has no parent (there will be exactly one of these), then it is the root of the tree.

Rooted Trees: Example

- A is the root
- D, E, G, H, J & K are leaves
- B is the parent of D, E & F
- D, E & F are siblings and children of B
- I, J & K are descendants of B
- A & B are ancestors of I
Intuitively, a **binary tree** is a tree in which each node has no more than two children. (These two binary trees are distinct.)

A **binary search tree** is a binary tree in which each node, $n$, has a value $v(n)$ satisfying the following properties:

- $v(n)$'s value is $>$ all values in its left subtree $T_L$.
- $v(n)$'s value is $<$ all values in its right subtree $T_R$.
- $T_L$ and $T_R$ are both binary search trees.

This term is ambiguous, some indicate that each node is either full or empty.

A binary tree is **full** if it has no missing nodes.

- It is either empty.
- Otherwise, the root's subtrees are full binary trees of height $h - 1$.
- If not empty, each node has 2 children, except the nodes at level $h$ which have no children.
- Contains a total of $2^{h+1} - 1$ nodes (how many leaves?)

A binary tree of height $h$ is **complete** if it is full down to level $h - 1$, and level $h$ is filled from left to right.

- All nodes at level $h - 2$ and above have 2 children each.
- If a node at level $h - 1$ has children, all nodes to its left at the same level have 2 children each, and
- If a node at level $h - 1$ has 1 child, it is a left child.
A binary tree is **balanced** if the difference in height between any node’s left and right subtree is ≤ 1.

Note that:
- A full binary tree is also complete.
- A complete binary tree is not always full.
- Full and complete binary trees are also balanced.
- Balanced binary trees are not always full or complete.

---

**Complete & Balanced Trees**

- Not Balanced, Why?

---

**Binary Tree: Pointer-Based Representation**

```c
struct TreeNode; // Binary Tree nodes are struct's
typedef string TreeItemType; // items in TreeNodes are string's
class BinaryTree
{
private:
    TreeNode *root; // pointer to root of Binary Tree
};
struct TreeNode
{
    TreeItemType item; // node in a Binary Tree:
    // place in Implementation file
    TreeNode *leftChild; // pointer to TreeNode's left
    child
    TreeNode *rightChild; // pointer to TreeNode's right child
};
```

---

**Binary Tree: Table-Based Representation**

**Basic Idea:**
- Instead of using *pointers* to the left and right child of a node, use *indices* into an array of nodes representing the binary tree.
- Also, use variable *free* as an index to the first position in the array that is available for a new entry. Use either the *left* or *right child* indices to indicate additional, available positions.
- Together, the list of available positions in the array is called the *free list*. 
**Binary Tree: Table-Based Representation**

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>Left Child</th>
<th>Right Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>-1</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

**Index Item Left Child Right Child**

- **root**: 0
- **free**: 6

**Binary Tree**

- **Jane**
  - **Bob**
  - **Tom**
- **Alan**
- **Ellen**
- **Nancy**

**Binary Tree: Table-Based Representation**

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>Left Child</th>
<th>Right Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>-1</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

**Index Item Left Child Right Child**

- **root**: 0
- **free**: 6

**Binary Tree**

- **Jane**
  - **Bob**
  - **Tom**
- **Alan**
- **Ellen**
- **Nancy**

**Binary Tree: Table-Based Representation**

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>Left Child</th>
<th>Right Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>-1</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

**Index Item Left Child Right Child**

- **root**: 0
- **free**: 6

**Binary Tree**

- **Jane**
  - **Bob**
  - **Tom**
- **Alan**
- **Ellen**
- **Nancy**
- **Mary**

**Binary Tree: Table-Based Representation**

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>Left Child</th>
<th>Right Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>-1</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

**Index Item Left Child Right Child**

- **root**: 0
- **free**: 6

**Binary Tree**

- **Jane**
  - **Bob**
  - **Tom**
- **Alan**
- **Ellen**
- **Nancy**
- **Mary**

**Binary Tree: Table-Based Representation**

- Ellen deleted.

**Binary Tree**

- **Jane**
  - **Bob**
  - **Tom**
- **Alan**
- **Ellen**
- **Nancy**
- **Mary**

**Binary Tree: Table-Based Representation**

```cpp
const int MaxNodes = 100; // maximum size of a Binary Tree
typedef string TreeItemType; // items in TreeNodes are string’s
type TreeNode // node in a Binary Tree
{
    TreeItemType item;
    int leftChild; // index of TreeNode’s left child
    int rightChild; // index of TreeNode’s right child
};

class BinaryTree
{
    private:
        TreeNode node[MaxNodes]; // index of root of Binary Tree
        int root; // index of root of Binary Tree
        int free; // index of free list, linked by rightChild
};
```
Let $i, 1 \leq i \leq n$, be the number assigned to an element of a complete binary tree.

- Array-based representations allow for efficient traversal. Consider the node at index $i$.
  - Left Child is at index $2i+1$.
  - Right Child is at index $2i+2$.
  - Parent is at $\text{floor}(\frac{i-1}{2})$. 
Array-Based Representation

- **Drawback** of array-based trees: Example has only 3 nodes, but uses \(2^{h+1}-1\) array cells to store it.
- Generally use Array-based only if data set exhibits **complete** binary tree behavior.

![Array-Based Representation Diagram]

Traversing a Binary Tree

- Depth-first Traversal
  - Preorder
  - Inorder
  - Postorder
- Breadth-First Traversal
  - Level order

![Traversing a Binary Tree Diagram]

Preorder Traversal

**Basic Idea:**

1) **Visit** the root.
2) Recursively invoke **preorder** on the left subtree.
3) Recursively invoke **preorder** on the right subtree.

**Preorder Result:** 60, 20, 10, 40, 30, 50, 70

![Preorder Traversal Diagram]
**Inorder Traversal**

**Basic Idea:**
1) Recursively invoke \textit{inorder} on the \textit{left} subtree.
2) \textit{Visit} the \textit{root}.
3) Recursively invoke \textit{inorder} on the \textit{right} subtree.

**Postorder Traversal**

**Basic Idea:**
1) Recursively invoke \textit{postorder} on the \textit{left} subtree.
2) Recursively invoke \textit{postorder} on the \textit{right} subtree.
3) \textit{Visit} the \textit{root}.
Level order traversal

- Visit the tree in left-to-right, by level, order:
  - Visit the root node and put its children in a queue (left to right).
  - Dequeue, visit, and put dequeued node’s children into the queue.
  - Repeat until the queue is empty.

```plaintext
    f
c  a  d  h  j
     \    /  \
     i  k
```

Pointer-Based, Preorder Traversal in C++

```plaintext
// FunctionType is a pointer to a function with argument
// (TreeItemType &) that returns void.
typedef void (*FunctionType) (TreeItemType &treeItem);

// Public member function
define void BinaryTree::preorderTraverse( FunctionType visit )
{
    preorder( root, visit );
}

// Private member function
define void BinaryTree::preorder( TreeNode *treePtr,
    FunctionType visit )
{
    if( treePtr != NULL )
    {
        visit( treePtr -> item );
        preorder( treePtr -> leftChild, visit );
        preorder( treePtr -> rightChild, visit );
    }
}
```

Suppose that we define the function
```plaintext
void printItem( TreeItemType &treeItem )
{
    cout << treeItem << endl;
}
```

Then,
```plaintext
// create myTree
    BinaryTree myTree;
// load data into myTree
    ... 
// print TreeItems encountered in preorder traversal of myTree
    myTree.preorderTraverse( &printItem );
```
**Nonrecursive Traversal of a Binary Tree**

**Basic Idea** for a Nonrecursive, *Inorder* Traversal:
1) Push a pointer to the *root* of the binary tree onto a stack.
2) Follow *leftChild* pointers, pushing each one onto the stack, until a *NULL* *leftChild* pointer is found.
3) Process (visit) the item in this node.
4) Get the node’s *rightChild* pointer:
   - If it is not *NULL*, then push it onto the stack, and return to step 2 with the *leftChild* pointer of this *rightChild*.
   - If it is *NULL*, then pop a node pointer from the stack, and return to step 3. If the stack is empty (so nothing could be popped), then stop — the traversal is done.

**Other Binary Tree Properties**

- The number of edges in a tree is $n-1$.
- The number of nodes $n$ in a full binary tree is: $n = 2^h + 1 - 1$ where $h$ is the height of the tree.
- The number of nodes $n$ in a complete binary tree is:
  - minimum: $n = 2^h$
  - maximum: $n = 2^h + 1 - 1$ where $h$ is the height of the tree.
- The number of nodes $n$ in a full or perfect binary tree is:
  - $n = 2L - 1$ where $L$ is the number of leaf nodes in the tree.
- The number of leaf nodes $n$ in a full or perfect binary tree is:
  - $n = 2^h$ where $h$ is the height of the tree.
- The number of leaf nodes in a Complete Binary Tree with $n$ nodes is $UpperBound(n/2)$.
- For any non-empty binary tree with $n_0$ leaf nodes and $n_2$ nodes of degree 2, $n_0 = n_2 + 1$.

**N-ary Trees**

- We can encode an n-ary tree as a binary tree, by have a list of linked-list of children. Hence still two pointers, one to the first child and one to the next sibling.
- Kinda rotates the tree.

**Graphs**

- **Graph $G = (V, E)$**
  - $V$ = set of vertices
  - $E$ = set of edges $\subseteq (V \times V)$
- **Types of graphs**
  - Undirected: edge $(u, v) = (v, u)$; for all $v$, $(v, v) \notin E$ (No self loops.)
  - Directed: $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbb{R}$.
- Dense: $|E| = |V|^2$.
- Sparse: $|E| \ll |V|^2$.
- $|E| = O(|V|^2)$
Graphs

- If \((u, v) \in E\), then vertex \(v\) is adjacent to vertex \(u\).
- Adjacency relationship is:
  - Symmetric if \(G\) is undirected.
  - Not necessarily so if \(G\) is directed.
- If \(G\) is connected:
  - There is a path between every pair of vertices.
  - \(|E| \geq |V| - 1\).
  - Furthermore, if \(|E| = |V| - 1\), then \(G\) is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed.

Representation of Graphs

- Two standard ways.
  - Adjacency Lists.
  - Adjacency Matrix.

Adjacency Lists

- Consists of an array \(Adj\) of \(|V|\) lists.
- One list per vertex.
- For \(u \in V\), \(Adj[u]\) consists of all vertices adjacent to \(u\).

Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is
    \[ \sum_{v \in V} \text{out-degree}(v) = |E| \]
  - Total storage: \(\Theta(|V| + |E|)\)
- For undirected graphs:
  - Sum of lengths of all adj. lists is
    \[ \sum_{v \in V} \text{degree}(v) = 2|E| \]
  - Total storage: \(\Theta(|V| + |E|)\)
Pro and Cons: adj list

Pros
- Space-efficient, when a graph is sparse.
- Can be modified to support many graph variants.

Cons
- Determining if an edge \((u, v) \in G\) is not efficient.
  - Have to search in \(u\)'s adjacency list. \(\Theta(\text{degree}(u))\) time.
  - \(\Theta(V)\) in the worst case.

Adjacency Matrix

- \(|V| \times |V|\) matrix \(A\).
- Number vertices from 1 to \(|V|\) in some arbitrary manner.
- \(A\) is then given by:
  \[
  A[i, j] = a_{ij} = \begin{cases} 
  1 & \text{if } (i, j) \in E \\
  0 & \text{otherwise}
  \end{cases}
  \]

Some graph operations

<table>
<thead>
<tr>
<th>adjacency matrix</th>
<th>adjacency lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertEdge</td>
<td>(O(1))</td>
</tr>
<tr>
<td>isEdge</td>
<td>(O(1))</td>
</tr>
<tr>
<td>#successors?</td>
<td>(O(V))</td>
</tr>
<tr>
<td>#predecessors?</td>
<td>(O(V))</td>
</tr>
</tbody>
</table>

\(A = A^T\) for undirected graphs.

Space and Time

- **Space:** \(\Theta(V^2)\).
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to \(u\): \(\Theta(V)\).
- **Time:** to determine if \((u, v) \in E\): \(\Theta(1)\).
- Can store weights instead of bits for weighted graph.
C# Interfaces

using System;
using System.Collections.Generic;
using System.Security.Permissions;
[assembly: CLSCompliant(true)]
namespace OhioState.Collections.Graph {
    /// <summary>
    /// The Graph interface
    /// <summary>
    public interface IGraph<N, E> {
        /// <summary>
        /// The type of the nodes in the graph.
        /// <summary>
        /// The type of the data on an edge. Also called the edge label.
        /// <summary>
        public interface IEdge<N, E> {
            /// <summary>
            /// The type associated at each node.
            /// Called a node or node label.<typeparamname=N/>
            /// <summary>
            /// The type associated at each edge. Also called the edge label.<typeparamname=E/>
            /// <summary>
            public interface IFiniteGraph<N, E> : IGraph<N, E> {
                /// <summary>
                /// Get the number of edges in the graph.
                /// <summary>
                int NumberOfEdges { get; }
                /// <summary>
                /// Get the number of nodes in the graph.
                /// <summary>
                int NumberOfNodes { get; }
            }
        }
    }
}

C# Interfaces

using System;
using System.Collections.Generic;
namespace OhioState.Collections.Graph {
    /// <summary>
    /// The Graph interface
    /// <summary>
    public interface IGraph<N, E> {
        /// <summary>
        /// The type of the nodes in the graph.
        /// <summary>
        /// The type of the data on an edge. Also called the edge label.<typeparamname=N/>
        /// <summary>
        /// The type associated at each edge. Also called the edge label.<typeparamname=E/>
        /// <summary>
        public interface IEdge<N, E> {
            /// <summary>
            /// The type associated at each node.
            /// Called a node or node label.<typeparamname=N/>
            /// <summary>
            /// The type associated at each edge. Also called the edge label.<typeparamname=E/>
            /// <summary>
            public interface IFiniteGraph<N, E> : IGraph<N, E> {
                /// <summary>
                /// Get the number of edges in the graph.
                /// <summary>
                int NumberOfEdges { get; }
                /// <summary>
                /// Get the number of nodes in the graph.
                /// <summary>
                int NumberOfNodes { get; }
            }
        }
    }
}

C# Interfaces

using System;
using System.Collections.Generic;
namespace OhioState.Collections.Graph {
    /// <summary>
    /// The Graph interface
    /// <summary>
    public interface IGraph<N, E> {
        /// <summary>
        /// The type of the nodes in the graph.
        /// <summary>
        /// The type of the data on an edge. Also called the edge label.<typeparamname=N/>
        /// <summary>
        /// The type associated at each edge. Also called the edge label.<typeparamname=E/>
        /// <summary>
        public interface IEdge<N, E> {
            /// <summary>
            /// The type associated at each node.
            /// Called a node or node label.<typeparamname=N/>
            /// <summary>
            /// The type associated at each edge. Also called the edge label.<typeparamname=E/>
            /// <summary>
            public interface IFiniteGraph<N, E> : IGraph<N, E> {
                /// <summary>
                /// Get the number of edges in the graph.
                /// <summary>
                int NumberOfEdges { get; }
                /// <summary>
                /// Get the number of nodes in the graph.
                /// <summary>
                int NumberOfNodes { get; }
            }
        }
    }
}