Definitions

- A **tree** is a collection of nodes.
  - The collection can be empty
  - Otherwise, a tree consists of
    - A distinguished node \( r \), called the **root**
    - Zero or more nonempty sub-trees \( T_1, T_2, \ldots, T_n \), each of whose roots are connected by a directed **edge** from \( r \)

- **General Tree characteristics**
  - The root of each subtree is a **child** of \( r \)
  - \( r \) is the **parent** of each subtree root.
  - Example using recursive definition:

```
root
   /   \
 T1   T2
   /   \
 T3   Tn
```

Definitions (cont.)

- **General Tree characteristics**
  - **Out degree**: the number of children of that node
  - **Forest**: a collection of one or more trees.
  - Binary tree definitions that don’t conflict also apply
  - Example (Figure 6.1):

General Tree Node

What operations must be supported in a general tree?

- The General Tree and Node ADTs:

  ```
  template <class Elem> class GTreeNode {
      public:
          GTreeNode(const Elem&); // Constructor
          ~GTreeNode(); // Destructor
          Elem value(); // Return node's value
          bool isLeaf(); // TRUE if node is a leaf
          GTreeNode* parent(); // Return parent
          GTreeNode* leftmost_child(); // Return first child
          GTreeNode* right_sibling(); // Return right sibling
          void setValue(Elem&); // Set node's value
          void insert_first(GTreeNode<Elem>* n); // Insert 1st child
          void insert_next(GTreeNode<Elem>* n); // Insert next sibling
          void remove_first(); // Remove first child
          void remove_next(); // Remove right sibling
      }
  
  template <class Elem> class GenTree {
      public:
          GenTree();
          ~GenTree();
          void clear(); // Free the nodes
          GTreeNode* root(); // return root
          void newroot(Elem, // Combine trees
                      GTreeNode<Elem>* root1, GTreeNode<Elem>* root2);
  }
  ```

CSC 375-Turner, Page 2

General Tree Traversals

There is no concept of an inorder traversal

- Recursive definitions:
  - **Preorder**: visit the root, then perform a preorder traversal of each subtree from left to right
  - **Postorder**: perform a preorder traversal of each subtree from left to right, then visit the root
  - **Preorder Example**:

    ```
    template <class Elem>
    void GenTree<Elem>::
    printHelp(GTreeNode<Elem>* subroot) {
        if (subroot->isLeaf())
            cout << "Leaf: ";
        else
            cout << "Internal: ";
        cout << subroot->value() << ":\n";
        for (GTreeNode<Elem>* temp = subroot->leftmost_child();
             temp != NULL; temp = temp->right_sibling())
            printHelp(temp);
    }
    ```

CSC 375-Turner, Page 3
**General Tree Example**

- Example:

  ![Diagram of a general tree example]

  - Preorder: R A C D E B F
  - Postorder: C D E A F B R
  - Inorder?

- Example:

  ![Diagram of another general tree example]

  - Preorder:
  - Postorder:

*CSC 375-Turner, Page 5*

---

**General Tree Implementations**

There are several choices, depending on application

- Array-based implementations:
  - Parent pointer
  - Lists of children (hybrid array/link)
  - Leftmost child/right sibling

- Link-based implementations:
  - Fixed-size arrays for child pointers
  - Linked lists of child pointers

- Storage-based
  - Sequential tree implementation

*CSC 375-Turner, Page 6*

---

**Parent Pointer Implementation**

*Probably the simplest general tree implementation*

- Each node stores only a pointer to its parent
  - Not very general purpose
  - Very good for answering whether two nodes are in the same tree
    - Operation is called FIND
  - A Disjoint set problem:
    - Determine if two objects are in the same set (FIND)
    - Merge two sets together (UNION)

- A useful application is determining equivalence classes

*CSC 375-Turner, Page 7*

---

**Parent Pointer Implementation**

- Nodes are stored in an array:

  ![Diagram of parent pointer implementation]

<table>
<thead>
<tr>
<th>Parent’s Index</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>7</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>R</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>Node Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

*CSC 375-Turner, Page 8*
Union/Find

- Are two elements in the same tree?
  
  ```cpp
  bool G{}::differ(int a, int b) {
    int root1 = FIND(a);
    int root2 = FIND(b);
    return (root1 != root2);
  }
  ```

- Implementing Union and Find:
  
  ```cpp
  void G{}::UNION(int a, int b) {
    int root1 = FIND(a);
    int root2 = FIND(b);
    if (root1 != root2) {
      array[root2] = root1;
    }
  }
  ```

- Keep the depth small using weighted union rule
  - Weighted union rule: Join the tree with fewer nodes to the tree with more nodes

Equivalence Processing Example

- Initial:

<table>
<thead>
<tr>
<th>Array</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 -1 -1 -1 -1 -1</td>
<td>A B C D E</td>
</tr>
<tr>
<td>A B C D E F G H I J</td>
<td></td>
</tr>
</tbody>
</table>

- After processing (A,B), (C,H), (G,F), (D,E), (I,F):

<table>
<thead>
<tr>
<th>Array</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 0 -1 -1 3 -1 5 2 5 -1</td>
<td>A B C D E F G H I J</td>
</tr>
<tr>
<td>A B C D E F G H I J</td>
<td></td>
</tr>
</tbody>
</table>

Equivalence Processing Example (cont.)

- After processing (H,A), (E,G)

<table>
<thead>
<tr>
<th>Array</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 0 5 3 -1 5 2 5 -1</td>
<td>A B C D E F G H I J</td>
</tr>
<tr>
<td>A B C D E F G H I J</td>
<td></td>
</tr>
</tbody>
</table>

- After processing (H,E)

<table>
<thead>
<tr>
<th>Array</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 0 5 3 -1 5 2 5 -1</td>
<td>A B C D E F G H I J</td>
</tr>
<tr>
<td>A B C D E F G H I J</td>
<td></td>
</tr>
</tbody>
</table>

Path Compression

Resets the parent of every node on the path from node X to root R

- Code:
  
  ```cpp
  GNode* G{}::FIND(GNode* curr) const {
    if (array[curr] == ROOT)
      return curr;
    return array[cur] = FIND(array[cur]);
  }
  ```

- Process (H,E):

<table>
<thead>
<tr>
<th>Array</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 0 5 5 -1 5 0 5 -1</td>
<td>A B C D E F G H I J</td>
</tr>
<tr>
<td>A B C D E F G H I J</td>
<td></td>
</tr>
</tbody>
</table>
Lists of Children
Hybrid Representation

- Key question: how well does a representation perform certain tasks?
  - find left child and right sibling
  - find a parent

Parent

Label

R -1
A 0
C 1
B 0
D 1
F 3
E 1

- What tree does this represents?

CSC 375-Turner, Page 13

Leftmost Child/Right Sibling Array Representation

- Array of “pointers” (indices), contains:
  - index of left child
  - label (value)
  - index of parent
  - index of right sibling

CSC 375-Turner, Page 14

Fixed Size Pointer Array
Linked Representation

- Each parent maintains an array of pointers to children

CSC 375-Turner, Page 15

Linked Lists of Child Pointers
Linked Representation

- Each parent maintains an array of pointers to children

CSC 375-Turner, Page 16
K-ary Trees

A k-ary tree is a tree whose nodes may have up to k children

- A binary tree is the same as a k-ary tree for k = 2
- Features and disadvantages
  - Relatively easy to implement
  - More wasted space as k grows
- As go FBTs, so go FKTs:
  - Full 3-ary Tree Theorem: The number of leaves in a non-empty full 3-ary tree is equal to 2^n + 1 where n is the number of internal nodes
  - Corollary: The number of empty subtrees in a nonempty 3-ary tree is ...

Sequential Tree Implementations

Represents an application of the space/time tradeoff principle

- Goal: store a series of node values with minimum information necessary to reconstruct the tree structure
  - Advantage: space is saved
  - Disadvantage: cost to regenerate tree (loss of efficient access to nodes)

CSC 375-Turner, Page 17

Sequential Tree Implementations (cont.)

- FBT implies the list may be stored more efficiently
  - Mark a leaf or an internal
  - In a FBT, no ‘/’ characters in the representation
  - Using the prior example, internals are marked
  - Representation is A'B'/D'E'G/F'H'I
- General trees require a ‘list end’ indicator

CSC 375-Turner, Page 18

CSC 375-Turner, Page 19