Overview

- Binary Tree traversals
  - Preorder, postorder, inorder
- Binary Tree Data Structures
  - Vector, Linked List
- General Tree Data Structures
- Converting General Trees to Binary Trees

Binary Tree Traversals: (1)

- **Preorder Traversal**:
  - Since a binary tree is also “regular tree”, can use preorder traversal for general trees. However we can simplify it!

  **Algorithm**

  ```
  binaryPreorder(T, v)
  if v is an internal node then
    binaryPreorder(T, T.leftChild(v))
    binaryPreorder(T, T.rightChild(v))
  perform visit action on node v
  ```

Binary Tree Traversals: (2)

- **Postorder Traversal**:
  - Can also simplify the postorder traversal for binary trees.

  **Algorithm**

  ```
  binaryPostorder(T, v)
  if v is an internal node then
    binaryPostorder(T, T.leftChild(v))
    binaryPostorder(T, T.rightChild(v))
  perform visit action on node v
  ```

  - Can be used to solve the expression evaluation problem.

Binary Tree Traversals: (3)

- Specialization of a postorder traversal

  **Algorithm**

  ```
  evaluateExpression(v)
  if v is an internal node
    return the variable stored at v
  else
    let o be the operator stored at v
    x ← evaluateExpression(leftChild(v))
    y ← evaluateExpression(rightChild(v))
    return x o y
  ```
Binary Tree Traversals: (4)

- **Inorder** Traversal:
  - Visit a node between the recursive traversals of its left and right subtrees.

**Algorithm** inorder(T, v)

if v is an internal node then
  inorder(T, T.leftChild(v))
perform “visit” for node v
if v is an internal node then
  inorder(T, T.rightChild(v))


Binary Tree Traversals: (5)

- Visit the nodes of T “from left to right”.
- Visits v after all nodes in its left subtree and before the nodes of its right subtree.
- Many Applications:
  - Inorder traversal of a *binary search tree* visits the elements in a non-decreasing order.
  - Tree Drawing


Binary Tree Traversals: (6)

- Example: Printing an Arithmetic Expression

- specialization of an inorder traversal
- print “(” before traversing the left subtree
- print “)” after traversing the right subtree


Binary Search Tree: (1)

- Definition:
  - Each internal node v stores an element e such that:
    - Elements stored in the left subtree of v are less than or equal to e.
    - Elements stored in the right subtree of v are greater than or equal to e.
Binary Tree Data Structures: (1)

- Vector Based Implementation:
  - **Level Ordering:** For every node $v$ of $T$, let $p(v)$ be the integer defined as follows:
    - If $v$ is the root, $p(v) = 1$.
    - If $v$ is left child of node $u$, $p(v) = 2p(u)$.
    - If $v$ is right child of node $u$, $p(v) = 2p(u) + 1$
    - Numbers the nodes of each level of $T$ in increasing order from left to right (but may skip some nodes!)

Binary Tree Data Structures: (2)

- Representation of binary tree $T$ using a vector $S$ such that node $v$ of $T$ is associated with element of $S$ at rank $p(v)$.
- Simple and efficient implementation.
  - Perform the methods root, parent, leftChild, rightChild, sibling, isInternal, isExternal and isRoot using simple math on the numbers $p(v)$.

Binary Tree Data Structures: (4)

- Let $n$ be number of nodes of $T$, $p_M$ max. value of $p(v)$ over all nodes of $T$.
  - Vector size $N = p_M + 1$
  - No element at rank 0!
- Vector method is fast and easy representation but can be very space inefficient if the height of the tree is large!

Binary Tree Data Structures: (5)

- Representation of a Binary tree $T$ with a Vector $S$. 

![Diagram of binary tree with vector representation](image)
Running Times of the Methods of a Binary Tree Implemented with a Vector:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions, elements</td>
<td>O(n)</td>
</tr>
<tr>
<td>swapElements, replaceElements</td>
<td>O(1)</td>
</tr>
<tr>
<td>root, parent, children</td>
<td>O(1)</td>
</tr>
<tr>
<td>leftChild, rightChild, sibling</td>
<td>O(1)</td>
</tr>
<tr>
<td>isInternal, isExternal, isRoot</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Represent each node v of tree T by an object with reference to:

- Element stored at v.
- Position objects associated with the children and parent of v.

If v is the root of T, reference to parent is null.

If v is an external node of T, references to children are null.

To save space, when external nodes are empty, can have references to external nodes be null.

Can use a special object, NULL_NODE & every external node reference is instead to this object.
**Binary Tree Data Structures: (10)**

- Using the *NULL_NODE* we have to be prepared to throw an exception if the parent method is passed such an object as an argument.

**Binary Tree Data Structures: (11)**

- Example of a Linked Data Structure for a Binary Tree:

![Binary Tree Diagram](image)

**General Tree Data Structures: (1)**

- Linked Structure for General Trees:
  - Can extend the linked structure for binary trees to represent general trees.
  - No limit to the number of children a node can have, use a container (e.g. list, vector) to store the children of node \( v \) instead of using instance variables.

**General Tree Data Structures: (2)**

- Linked Structure for General Trees:

![General Tree Diagram](image)
General Tree Data Structures: (3)

- Can implement method children($v$) by simply calling elements() method of the container.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size, isEmpty</td>
<td>O(1)</td>
</tr>
<tr>
<td>Positions, elements</td>
<td>O(n)</td>
</tr>
<tr>
<td>swapElements,</td>
<td>O(1)</td>
</tr>
<tr>
<td>replaceElements</td>
<td>O(1)</td>
</tr>
<tr>
<td>Root, parent</td>
<td>O(1)</td>
</tr>
<tr>
<td>isInternal, isExternal, isRoot</td>
<td>O(1)</td>
</tr>
<tr>
<td>Children($v$)</td>
<td>O(c_v)</td>
</tr>
</tbody>
</table>

Converting a General Tree to a Binary Tree: (1)

- Representing General Trees with Binary Trees. Transform T into Binary Tree T’ as follows:
  - For each node $u$ of T, there is an internal node $u’$ of T’ associated with $u$.
  - If $u$ is an external node of T and doesn’t have a sibling immediately following it, then the children $u’$ of T’ are external nodes.

Converting a General Tree to a Binary Tree: (2)

- If $u$ is an internal node of T and $v$ is the first child of $u$ in T, then $v’$ is the left child of $u’$ in T.
- If node $v$ has a sibling $w$ immediately following it, then $w’$ is the right child of $v’$ in T’.

- External nodes of T’ are not associated with nodes T and serve only as placeholders.

Converting a General Tree to a Binary Tree: (3)

- Can be seen as a conversion of T into T’ that takes each set of siblings \{ $v_1$, $v_2$, ..., $v_k$ \} in T with parent $v$ and replaces it with a chain of right children rooted at $v_1$, which then becomes the left child of $v$. 
Converting a General Tree to a Binary Tree: (4)

- Example of the Conversion from a General Tree to a Binary Tree:

Priority Queue: (1)

- What is a Priority Queue?
  - An Abstract Data storing a collection of prioritized elements.
  - Supports arbitrary element insertion but supports removal of elements in order of priority.
    - The element with the highest priority can be removed at any time.

Priority Queue: (2)

- A priority queue stores elements in order of priority only!
  - No notion of position as with some other ADTs (sequences, lists etc.)

- Priority Queue ADT:
  - Each element in the priority queue has a corresponding “key” Object.
    - Key Object represents the elements priority.

Priority Queue: (3)

- “Key” Object - Definition:
  - An Object assigned to some element which can be used to rank, identify or weight the element.
  - Assigned to the element by the user or the application.
  - Maybe changed by the application if needed.
Priority Queue: (4)

- Does not need to be a single numerical value.
- Can sometimes be more complex and cannot be quantified by a single number.

Priority Queue: (4)

- In a priority queue, the key is used to assign a priority to each element:
  - A Priority Queue ranks its elements by key with a total order relation.
  - Keys:
    - Every element has its own key
    - Keys are not necessarily unique
  - Total Order Relation:
    - Denoted by ≤
    - Reflexive: \( k \leq k \)
    - Antisymmetric: if \( k_1 \leq k_2 \) and \( k_2 \leq k_1 \), then \( k_1 = k_2 \)
    - Transitive: if \( k_1 \leq k_2 \) and \( k_2 \leq k_3 \), then \( k_1 \leq k_3 \)

Priority Queue: (5)

**Sorting with a Priority Queue**

- A Priority Queue \( P \) can be used for sorting a sequence \( S \) by:
  - Inserting the elements of \( S \) into \( P \) with a series of `insertItem(e, k)` operations
  - Removing the elements from \( P \) in increasing order and putting them back into \( S \) with a series of `removeMin()` operations

**Algorithm PriorityQueueSort(S, P):**

**Input:** A sequence \( S \) of \( n \) elements, on which a total order relation is defined, and a Priority Queue \( P \) that compares keys with the same relation.

**Output:** The sequence \( S \) sorted by the total order relation.

```
while !S.isEmpty() do
    e ← S.removeFirst()
    P.insertItem(e, k)
while P is not empty do
    e ← P.removeMin()
    S.insertLast(e)
```

Priority Queue: (6)

**The Priority Queue ADT**

- A priority queue \( P \) supports the following methods:
  - `size()` Return the number of elements in \( P \)
  - `isEmpty()` Test whether \( P \) is empty
  - `insertItem(e, k)` Insert a new element \( e \) with key \( k \) into \( P \)
  - `minElement()` Return (but don’t remove) an element of \( P \) with smallest key; an error occurs if \( P \) is empty.
  - `minKey()` Return the smallest key in \( P \); an error occurs if \( P \) is empty
  - `removeMin()` Remove from \( P \) and return an element with the smallest key; an error condition occurs if \( P \) is empty.