CHAPTER 11 Trees

11.1 Introduction to Trees

【Definition 1】 A *tree* is a connected undirected graph with no simple circuits. *Forest* is an undirected graph with no simple circuits.

Note:

- Any tree must be a simple graph.
- Each connected components of forest is a tree.

【 Theorem 1 】 An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Rooted tree: In many applications of trees a particular vertex of a tree is designated as the *root*. Once we specify a root, we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a *rooted tree*.
- Parent: The *parent of a non-root vertex v* is the *unique* vertex *u* with a directed edge from *u* to *v*.
- Child: When *u* is the parent of *v*, *v* is called a *child* of *u*.
- Sibling: Vertices with the same parent are called *siblings* .
- Ancestors: The *ancestors of a non-root vertex* are all the vertices in the path from root to this vertex.
- Descendants: The *descendants of vertex v* are all the vertices that have *v* as an ancestor.
- Leaf: A vertex is called a *leaf* if it has no children.
- Internal Vertex: A vertex that has children is called an *internal vertex*.
- Subtree: The *subtree at vertex v* is the subgraph of the tree consisting of vertex v and its descendants and all edges incident to these descendants.

Binary Tree

【 Definition 】 A rooted tree is called a *m-ary tree* if every internal vertex has no more than *m* children.

The tree is called a *full m-ary tree* if every internal vertex has exactly *m* children.

Ordered rooted tree

【 Definition 】 An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered.

In an ordered binary tree, the two possible children of a vertex are called the *left child* and the *right child*, if they exist.

The tree rooted at the left child is called the *left subtree*, and that rooted at the right child is called the *right subtree*.

Tree Properties

【 Theorem 2】 A tree with n vertices has n-1 edges.

【 Theorem 3】 A full m-ary tree with i internal vertices contains $n = mi+1$ vertices.

【 Theorem 4】 A full m-ary tree with

- n vertices has $I = (n-1)/m$ internal vertices and $l = [(m-1)n+1]/m$ leaves
- i internal vertices has $n = mi+1$ vertices and $l = (m-1)i+1$ leaves
- l leaves has $n = (ml-1)/(m-1)$ vertices and $I = (l-1)/(m-1)$ internal vertices

Note: For a full binary tree, $l = i+1$, $e = v - 1$.

- Level: The *level of vertex v* in a rooted tree is the length of the unique path from the root to *v*.
- Height: The *height of a rooted tree* is the maximum of the levels of its vertices.

- Balanced: A rooted m-ary tree of height h is called *balanced* if all its leaves are at levels h or h-1.

【 Theorem 5】 There are at most m^h leaves in an m-ary tree of height h.

【 Corallary 】 If an m-ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$.

If the m-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$.

11.2 Applications of Trees

1. Binary Search Trees

- **The Concept of Binary Search Trees**

An ordered rooted binary tree each vertex contains a distinct key value. The key values in the tree can be compared using "greater than" and "less than", and the key value of each vertex in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

- **Construct the binary search tree**

The shape of a binary search tree depends on its key values and their order of insertion.

Insert the elements in order.

The first value to be inserted is put into the root.

Thereafter, each value to be inserted begins by comparing itself to the value in the root, moving left it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.

- **Binary Search Tree Algorithm**

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

```
procedure insertion(T: binary search tree, x: item)
v := root of T
{a vertex not present in T has the value null}
while v ≠ null and label(v) ≠ x
   if x < label(v) then
      if left child of v ≠ null then v := left child of v
      else add new vertex as a left child of v and set v := null
   else
      if right child of v ≠ null then v := right child of v
      else add new vertex as a right child of v and set v := null
if root of T = null then add a vertex v to the tree and label it with x
else if v is null or label(v) ≠ x then label new vertex with x and let v be this new vertex
return v {v = location of x}
```

- **The computational complexity**

Suppose we have a binary search tree T for a list of n items.

We can form a full binary tree U from T by adding unlabeled vertices whenever necessary so that every vertex with a key has two children.

- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- If a binary search tree is balanced, locating or adding an item requires no more than $\lceil \log(n+1) \rceil$ comparisons.

2. Decision Trees

A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices

for each possible outcome of the decision, is called a *decision tree*.

3. Prefix Codes

- **the Concept of Prefix Codes**

To ensure that no bit string corresponds to more than one sequence of letters, the bit string for a letter must never occur as the first part of the bit string for another letter. Codes with this property are called *prefix codes*.

- **How to Construct Prefix Codes**

Using a binary tree.

- the left edge at each internal vertex is labeled by 0.
- the right edge at each internal vertex is labeled by 1.
- the leaves are labeled by characters which are encoded with the bit string constructed using the labels of the edges in the unique path from the root to the leaves.

Huffman Coding

Given symbols and their frequencies, our goal is to construct a rooted binary tree where the symbols are the labels of the leaves. The algorithm begins with a forest of trees each consisting of one vertex, where each vertex has a symbol as its label and where the weight of this vertex equals the frequency of the symbol that is its label. At each step, we combine two trees having the least total weight into a single tree by introducing a new root and placing the tree with larger weight as its left subtree and the tree with smaller weight as its right subtree. Furthermore, we assign the sum of the weights of the two subtrees of this tree as the total weight of the tree. The algorithm is finished when it has constructed a tree, that is, when the forest is reduced to a single tree.

11.3 Tree Traversal

1. Traversal Algorithms

A traversal algorithm is a procedure for systematically visiting every vertex of an ordered rooted tree.

Tree traversals are defined recursively.

Tree traversals are named: preorder, inorder, postorder.

PREORDER Traversal Algorithm

【Definition】 Let T be an ordered tree with root r. If T has only r, then r is the *preorder traversal* of T. Otherwise, suppose T_1, T_2, …, T_n are the subtrees at r from left to right in T. The *preorder traversal* begins by visiting r. Then traverses T_1 in preorder, then traverses T_2 in preorder, and so on, until T_n is traversed in preorder.

Note: Preorder traversal of an binary ordered tree

- Visit the root.
- Visit the left subtree, using preorder.
- Visit the right subtree, using preorder.

INORDER Traversal Algorithm

【Definition】 Let T be an ordered tree with root r. If T has only r, then r is the *inorder traversal* of T. Otherwise, suppose T_1, T_2, …, T_n are the left to right subtrees at r. The *inorder traversal* begins by traversing T_1 in inorder. Then visits r, then traverses T_2 in inorder, and so on, until T_n is traversed in inorder.

Note: Inorder traversal of an binary ordered tree

- Visit the left subtree, using inorder.

- Visit the root.
- Visit the right subtree, using inorder.

POSTORDER Traversal Algorithm

【Definition】 Let T be an ordered tree with root r. If T has only r, then r is the postorder traversal of T. Otherwise, suppose T_1, T_2,…, T_n are the left to right subtrees at r. The postorder traversal begins by traversing T_1 in postorder. Then traverses T_2 in postorder, until T_n is traversed in postorder, finally ends by visiting r.

Note: Postorder traversal of an binary ordered tree

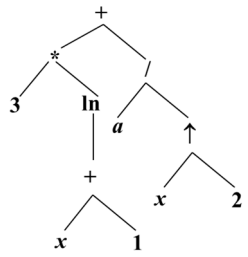
- Visit the left subtree, using postorder.
- Visit the right subtree, using postorder.
- Visit the root.

2. Infix, prefix, and postfix notation

A Binary Expression Tree is a special kind of binary tree in which:

1. Each leaf node contains a single operand,
2. Each nonleaf node contains a single operator,
3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

- Infix Form： The fully parenthesized expression obtained by an inorder traversal of the binary tree is said to be in *infix form*.
- Prefix Form： The expression obtained by an preorder traversal of the binary tree is said to be in *prefix form (Polish notation).*
- Postfix Form： The expression obtained by an postorder traversal of the binary tree is said to be in *postfix form (reverse Polish notation).*



Infix form: $(3 * ln(x + 1)) + (a/(x \uparrow 2))$

Prefix form: $+ * 3\, ln + x1/a \uparrow x2$

Postfix form: $3x1 + ln * a\, x2 \uparrow /+$

Evaluate the binary expression tree

- When a binary expression tree is used to represent an expression, the levels of the nodes in the tree indicate their relative precedence of evaluation.
- Operations at higher levels of the tree are evaluated later than those below them. The operation at the root is always the last operation performed.

11.4 Spanning Trees

【Definition 1】 Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

Find A Spanning Tree of The Simple Graph

Method: Find spanning trees by removing edges from simple circuits.

【 Theorem 1 】 A simple graph is connected if and only if it has a spanning tree.

Algorithms for constructing spanning trees

Depth-first search

Depth-first search (also called *backtracking*) -- this procedure forms a rooted tree, and the underlying undirected graph is a spanning tree.

1. Arbitrarily choose a vertex of the graph as root.
2. Form a path starting at this vertex by successively adding edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path.
3. Continue adding edges to this path as long as possible.
4. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.
5. If the path does not go through all vertices, more edges must be added. Move back to the next to last vertex in the path, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path. Repeat this process.

Breadth-first search

1. Arbitrarily choose a vertex of the graph as a root, and add all edges incident to this vertex.
2. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them.
3. For each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree.
4. Follow the same procedure until all the vertices in the tree have been added.

Backtracking scheme

There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution. The method to find a solution via backtracking

11.5 Minimum Spanning Trees

【Definition 1】 A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Algorithms for minimum spanning trees

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

Kruskal's algorithm

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(G: weighted connected undirected graph with n vertices)
$T :=$ empty graph
for $i := 1$ **to** $n - 1$
  $e :=$ any edge in G with smallest weight that does not form a simple circuit
    when added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}