Discrete Mathematics: Lecture 14

- Last time:
  - Chap 11.1: Introduction to trees
  - Chap 11.3: Tree traversal
- Today:
  - Chap 11.2: Applications of trees
- Assignment 5 due in two weeks

Review of last time

- Trees, forests, rooted trees, ordered rooted tress
- m-ary tree, full m-ary tree, balanced trees
- Tree terminology
- Basic properties of trees
  - there is a unique simple path between any two of its vertices
  - a tree with n vertices has $n - 1$ edges
  - A full m-ary tree with i internal vertices has $n = mi + 1$ vertices
  - There are at most m^h leaves in an m-ary tree of height h
- Universal address system for ordered rooted trees
- Preorder, inorder, postorder traversal of ordered rooted trees

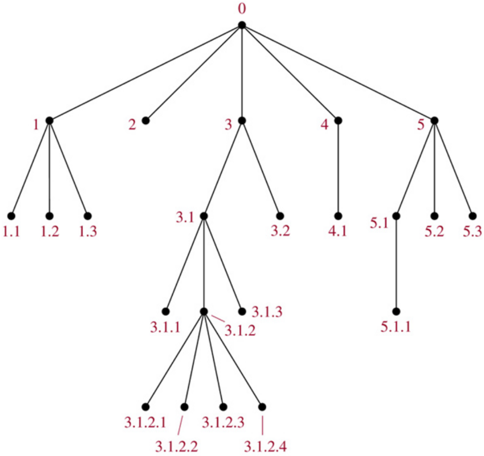
10.3: Tree travesal

- Ordered rooted trees are often used to store information.
- We will discuss several important algorithms for visiting each vertex of an ordered rooted tree.

Universal address systems (通用地址系统)

- We label all the vertices of an ordered rooted tree as follows:
  - Label the root with the integer 0. Then label its k children (at level 1) from left to right with $1, 2, \ldots, k$.
  - For each vertex v with label A, label its k_v children, as they are drawn from left to right, with $A.1, A.2, \ldots, A.k_v$.
- The labeling is called the universal address system of the ordered rooted tree.
- Then a vertex v at level n, is labeled $x_1.x_2.\ldots.x_n$, where the unique path from the root to v goes through the x_1st vertex at level 1, the x_2nd vertex at level 2, and so on.
- We can totally order the vertices using the lexicographic ordering of their labels.

An example



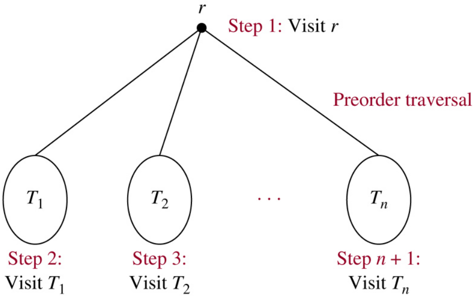
$0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2 < 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3$

Traversal algorithms

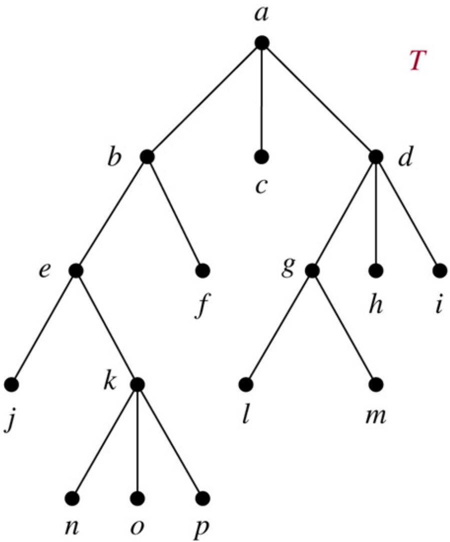
- Ordered rooted trees are often used to store information.
- Traversal (遍历) algorithms are procedures for systematically visiting every vertex of an ordered rooted tree.
- Three commonly used algorithms: preorder (前序) traversal, inorder (中序) traversal, and postorder (后序) traversal

Preorder traversal

- Let T be an ordered rooted tree with root r.
- If T consists only of r , then r is the preorder traversal of T.
- Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right.
- The preorder traversal begins by visiting r. It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.

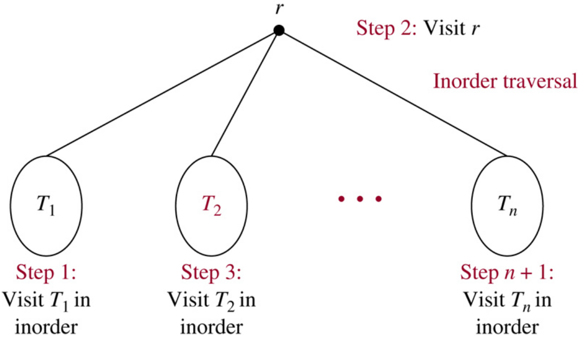


An example



Inorder traversal

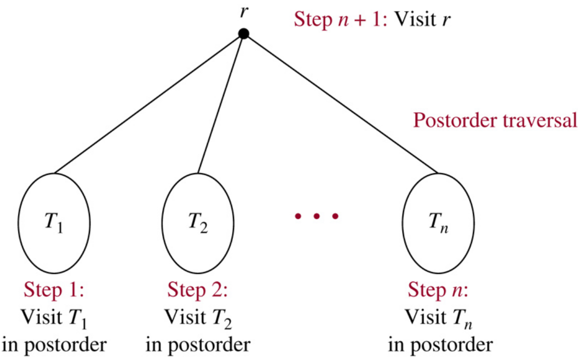
Definition: ... The inorder traversal begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, and so on, until T_n is traversed in inorder.



Example:

Postorder traversal

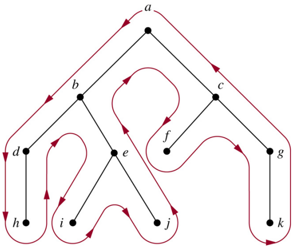
Definition: ... The postorder traversal begins by traversing T_1 in postorder, then T_2 in postorder, ..., then T_n in postorder, and ends by visiting r.



Example:

A shortcut for traversing an ordered rooted tree

- First draw a curve around the ordered rooted tree starting at the root, moving along the edges.
- Preorder: list each vertex the first time the curve passes it
- Inorder: list a leaf when the curve passes it and list each internal vertex the second time the curve passes it
- Postorder: list a vertex the last time the curve passes it

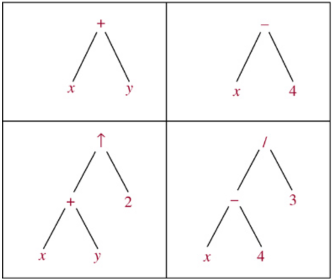


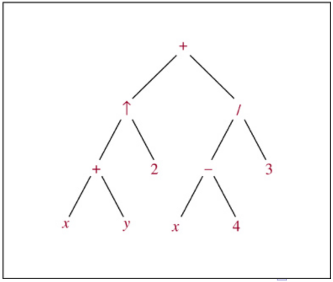
A recursive algorithm for inorder traversal

procedure inorder(T: ordered rooted tree)
r := root of T
if r is a leaf then list r
else
  l := first child of r from left to right
  $T(l)$:= subtree with l as its root
  $inorder(T(l))$
  list r
  for each child c of r except for l from left to right
    $T(c)$:= subtree with c as its root
    $inorder(T(c))$

Binary tree representation of expressions

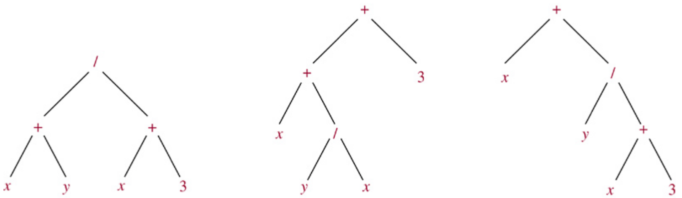
- We can represent expressions using ordered rooted trees.
- The internal vertices represent operations.
- The leaves represent the variables or numbers.
- Each binary operation operates on its left and right subtrees, each unary operation operates on its single subtree.
- Example: $((x + y) \uparrow 2) + ((x - 4)/3)$





Infix (中缀) notation

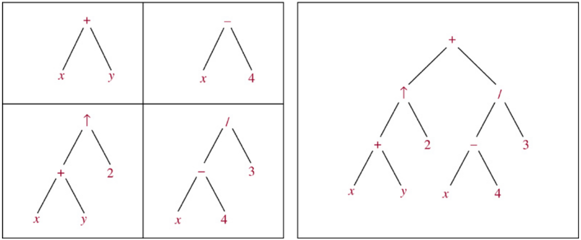
- When there are only binary operations, an inorder traversal of the binary tree representing an expression produces the original expression without parentheses.
- Example: inorder traversals of the following binary trees all lead to $x + y/x + 3$



- To avoid ambiguity, we should include parentheses for operations.
- The fully parenthesized expression is said to be in infix form.

Prefix (前缀) notation

- We obtain the prefix form of an expression when we traverse its rooted tree in preorder.
- Expression written in prefix form is said to be in Polish notation.
- An expression in prefix form is unambiguous, so no parentheses are needed.
- Example: the prefix form for $((x + y) \uparrow 2) + ((x - 4)/3)$



Evaluating an expression in prefix form

- In the prefix form, a binary operator (操作符) precedes its two operands (操作数).
- We can evaluate an expression in prefix form by working from right to left.
- When we see an operator, we perform the operation with the two operands immediately to the right of the operator.
- Whenever an operation is performed, we consider the result a new operand.
- Example: evaluating $+ - *235/\uparrow 234$

Postfix (后缀) notation

- We obtain the postfix form of an expression when we traverse its rooted tree in postorder.
- Expression written in postfix form is said to be in reverse Polish notation.
- An expression in postfix form is unambiguous, so no parentheses are needed.
- We can evaluate an expression in postfix form similarly as for prefix form, except that we work from left to right.

11.2: Introduction

- How should items in a list be stored so that an item can be easily located?
- What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type?
- How should a set of characters be efficiently coded by bit strings?

Binary search trees (搜索树)

- Searching for items in a list is one of the most important tasks that arises in computer science
- Our goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered
- This can be accomplished via the use of a binary search tree
    - a binary tree where each vertex is labeled with a key and the key of each vertex is larger (resp. smaller) than the keys of all vertices in its left (resp. right) subtree

Form a binary search tree

- Start with a single-vertex tree, and assign the first item in the list as the key of the root
- To add a new item, start at the root and move to the left (resp. right) if the item is less (resp. greater) than the key of the vertex and this vertex has a left (resp. right) child
- When the item is less (resp. greater) than the key of the vertex and this vertex has no left (resp. right) child, a new vertex with this item as its key is inserted as a new left (resp. right) child

Example 1

Form a binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetic order)

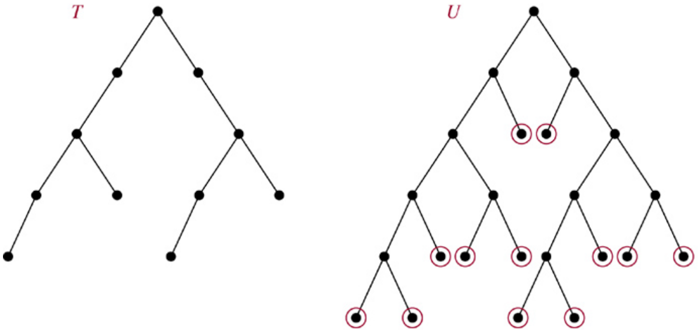
ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(T: binary search tree, x: item)
v := root of T
{a vertex not present in T has the value *null*}
while $v \neq null$ and $label(v) \neq x$
    if $x < label(v)$ **then**
        if left child of $v \neq null$ **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 $v \neq null$ **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 $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {v = location of x}

Example 2: Use Algorithm 1 to insert the word oceanography into the binary search tree in Example 1

The computational complexity of Algorithm 1

- 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



Unlabeled vertices circled

The computational complexity of Algorithm 1

- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf
- The internal vertices of U are the vertices of T, so U has n internal vertices.
- By Theorem 4 (ii) in Chap 11.1, U has $n+1$ leafs
- By Corollary 1 in Chap 11.1, the height of U is $\geq \lceil \log(n+1) \rceil$.
- Hence, it is necessary to perform $\geq \lceil \log(n+1) \rceil$ comparisons to add some item.
- Note that if U is balanced, its height is $\lceil \log(n+1) \rceil$.

Decision trees (决策树)

- Rooted trees can be used to model problems in which a series of decisions leads to a solution.
- Each internal vertex corresponds to a decision, with a subtree for each possible outcome of the decision.
- The possible solutions of the problem correspond to the paths to the leaves.

An example

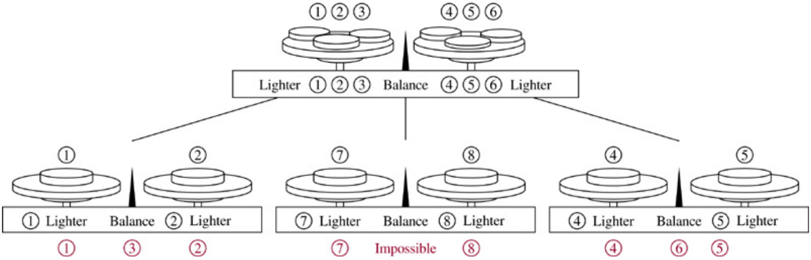
- There are 7 coins, all with the same weight, and a counterfeit coin that weighs less than the others.
- How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one?
- Give an algorithm for finding the counterfeit coin.

An example: solution

- There are 3 possibilities for each weighing. Hence the decision tree is a 3-ary tree.
- There are 8 possible outcomes. Hence the tree has 8 leaves.
- By Corollary 1 of Chap 11.1, the height of the tree is at least $\lceil \log_3 8 \rceil = 2$.

An example: solution

- There are 3 possibilities for each weighing. Hence the decision tree is a 3-ary tree.
- There are 8 possible outcomes. Hence the tree has 8 leaves.
- By Corollary 1 of Chap 11.1, the height of the tree is at least $\lceil \log_3 8 \rceil = 2$.



Introduction to prefix codes

- Consider using bit strings to encode the letters of the English alphabet
- Each letter can be represented with a bit string of length 5
- Is it possible to find a coding scheme of these letters such that when data are coded, fewer bits are used?
- If so, we can save memory and reduce transmittal time

Introduction to prefix codes

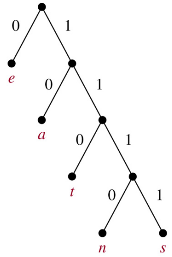
- Consider using bit strings of different lengths to encode letters
- Letters that occur more frequently should be encoded using short bit strings
- Then some method must be used to determine where the bits for each character start and end
- *e.g.*, if e - 0, a - 1, t - 01, then what does 0101 stand for?

Introduction to prefix codes

- One way to ensure that no bit string stands for more than one sequence of letters: encode letters so that the bit string for a letter never occurs as a prefix of the bit string for another one
- Codes with this property are called prefix codes (前缀码)
- *e.g.*, e - 0, a - 10, t - 11
- A word can be recovered from the unique bit string that encodes its letters
- *e.g.*, 10110

Prefix codes

- A prefix code can be represented using a binary tree
- The characters are the labels of leaves in the tree
- Left edges are labeled with 0, and right edges 1
- The bit strings used to encode a character is the sequence of labels of edges in the unique path from the root to the leaf labeled with the character
- Use the tree to decode a bit string, *e.g.*, 11111011100



Huffman coding

- An algorithm with input: the frequencies of symbols in a string, and output: a prefix code that encodes the string using the fewest possible bits
- This algorithm, known as Huffman coding, was developed by David Huffman in a term paper written in 1951 while he was a graduate student at MIT
- A fundamental algorithm in data compression, the subject devoted to reducing the number of bits required to represent information

ALGORITHM 2 Huffman Coding.

procedure *Huffman*(C: symbols a_i with frequencies w_i, $i = 1, \ldots, n$)
$F :=$ forest of n rooted trees, each consisting of the single vertex a_i and assigned weight w_i
while F is not a tree
    Replace the rooted trees T and T' of least weights from F with $w(T) \geq w(T')$ with a tree having a new root that has T as its left subtree and T' as its right subtree. Label the new edge to T with 0 and the new edge to T' with 1.
    Assign $w(T) + w(T')$ as the weight of the new tree.
{the Huffman coding for the symbol a_i is the concatenation of the labels of the edges in the unique path from the root to the vertex a_i}

Example: A: 0.08, B: 0.10, C: 0.12, D: 0.15, E: 0.20, F:0.35.
What is the average number of bits used to encode a character?