ECE 368

Week 6 (Notes)

Expression Trees

Binary trees provide an efficient data structure for representing expressions with binary operators.

- Root contains the operator
- Left and right children contain the operands - which may each be expression trees

Traversals ?

Generation ?

Tree Traversal Operation (Section 5.6)

(1) Preorder traversal (dept-first search)

(2) Inorder traversal (symmetric order)

(3) Postorder traversal

Tree Traversal Operation

(1) Preorder traversal (dept-first search)

(1) Visit the root

(2) Traverse the left subtree in preorder

(3) Traverse the right subtree in preorder

(NLR traversal)

Example of Tree Traversal Operation (In-class)

(2) Inorder Traversal (Symmetric Order)

(1) Traverse the left subtree in inorder (LNR)

(2) Visit the root

(3) Traverse the right subtree in inorder

--> infix expression (without parentheses) insert parentheses

(3) Postorder Traversal (LRN)

(1) Traverse left subtree

(2) Traverse right subtree

(3) Visit root

-→ postfix expression

Creation of Expression Trees

(Using Postfix Expression)

Algorithm uses a stack. It uses the same mechanism that is used for the evaluation of a postfix expression.

Let P(n) represent a postfix expression, with n tokens, labeled from 1 to n.

Let H(i) represent the stack height after the processing of i-th token.

Let LCHILD(i), RCHILD(i) be the left and the right child of a node representing the i-th token.

Algorithm for Building Expression Trees

(Using Postfix Expressions)

1. If i-th token is an operand, Stack it. It does not have any child.

   LCHILD (i) = RCHILD (i) = Null

2. If i-th token is a unary operator, then its right child is popped from the stack.

   RCHILD (i) = Token with index (i – 1)

   No left child. LCHILD(i) = Null

   Push the i-th token (operator) on the stack.

3. If i-th token is a binary operator, then its children are popped from the stack.

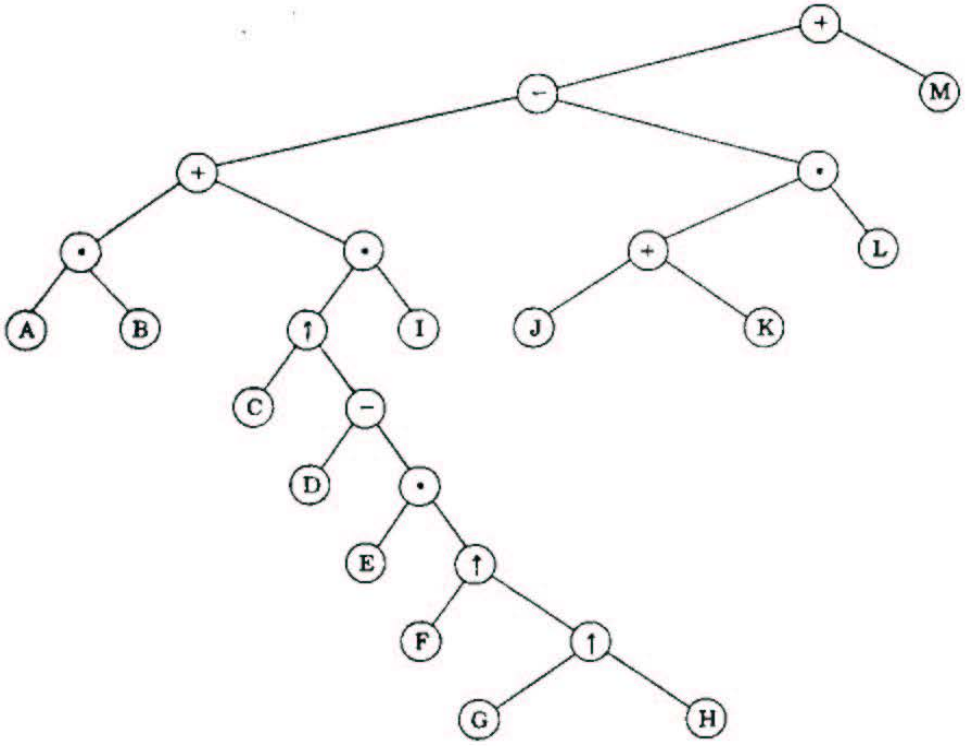
   RCHILD (i) = Token with (i – 1)

   LCHILD (i) = Token with index = largest j , (j < i) such that H(j) = H(i)

   Push the i-th token (operator) on the stack.

Example

Prefix	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P	A	B	•	C	D	E	F	G	H	↑	↑	•	−	↑	I	•	+	J	K	+	L	•	−	M	+
W	1	1	−1	1	1	1	1	1	1	−1	−1	−1	−1	−1	1	−1	−1	1	1	−1	1	−1	−1	1	−1
H	1	2	1	2	3	4	5	6	7	6	5	4	3	2	3	2	1	2	3	2	3	2	1	2	1
RCHILD	0	0	2	0	0	0	0	0	0	9	10	11	12	13	0	15	16	0	0	19	0	21	22	0	24
LCHILD	0	0	1	0	0	0	0	0	0	8	7	6	5	4	0	14	3	0	0	18	0	20	17	0	23



Complexity Analysis and Sorting

Read Chapter 6, Section 6.1

Time complexity

- A Priori Estimate
- A Posteriori Testing (Measurement)

Space Complexity

Time Complexity

Definition: Amount of Work Done by a Program

Two Possible ways:

1. Count the number of arithmetic and logical operations performed in a program. Obtain a weighted sum of these operations (weights being proportional to the time a computer takes to perform the operations.)

$$t_{add}, t_{sub}\ t_{mul}, t_{div}, \ldots.$$

$$T_C = t_{add}\ C_{add} + t_{sub}\ C_{sub} + t_{mul}\ C_{mul} + \ldots.$$

Limitations:

Practically not desirable.

Analysis is limited to a specific machine.

Time Complexity

2. Count the number of **“program steps”**

A program step is a meaningful segment of a program that has an execution time associated with it. It does not depend on the run time.

For a given language number of program steps can be determined for each statement.

Time Complexity

A further abstraction is needed, since we do not want complexity to be dependent upon the language and style of programming.

We need to isolate operation fundamental to the problem under study (ignore the bookkeeping).

The amount of time is then proportional to fundamental operations.

What if the total number of operations performed are substantially higher that the fundamental operations?

Define classes of algorithms!!

Some Examples of Reasonable Choices of Basic Operations

Problem	*Operation*
1. Find *X* in a list of names.	Comparison of *X* with an entry in the list
2. Multiply two matrices with entries.	Multiplication of two real numbers real (or multiplication and addition of real numbers.)
3. Sort a list of numbers.	Comparison of two list entries.
4. Traverse a binary tree represented as a linked structure where each node contains pointers to its left and right children.)	Traversing a link. (Here, setting a pointer would be considered a basic operation rather than bookkeeping.)

How to Present Analytical Results Concisely

Algorithm Performance Depends on:

- size of the input data
- characteristics of the input data

Example:

How performance results should be expressed?

How to Present Analytical Results Concisely

- Average Case Analysis
- Worst-Case Analysis

How about the “best” (optimal in the worst sense) algorithm?

Average Case Analysis

Compute the number of operations performed for each input of size *n* and then take the average.

In practice, some inputs occur more frequently than others.

Let D_n be the set of inputs of size *n*, for the problem under consideration.

Let *p(I)* be the probability that input *I* occurs and let *t(I)* be the number of basic operations performed by the algorithm on input *I*.

Average Case Analysis (cont.)

Average Complexity:

$$A(n) = \sum_{I \in D_n} p(I) \cdot t(I)$$

We need a careful evaluation of *t(I)* and need to know function *p(I)*, through experience.

Worst-Case Analysis

The maximum number of operations performed on any input of size *n*.

Provides an upper bound.

Worst-Case Complexity:

$W(n) = \max_{I \in D_n} t(I)$

Easy to compute!!

Example: Sequential Search

Assume *k* is an array of *n* records. We want to find the first *i* such that *k[i]* = KEY. If search is successful, return i else *-1*.

```
for (i = 0; I < n; i++)
if (KEY = = k(i))
return (-1);
```

Worst-Case Analysis of the Search Example

Note: Input for which an algorithm behaves worst depends on the particular algorithm, not on the problem.

If an algorithm searches the list from the bottom up, then the worst case would occur if **search key** is only in the first position (or is not present at all).

Some Merits of Selected Measure of Complexity

Notion of fundamental operation is acceptable if the total number of operations performed is roughly proportional to the number of fundamental operations.

If an algorithm performs *t(I)* operations, the total number of operations (or total execution time) will be *ct(I).*

c is a constant and depends on the computer and algorithm.

Similarly, we can describe *cA(n)* and *cW(n).*

Some Merits of Selected Measure of Complexity (cont.)

c may need to be determined to find the execution time.

Various algorithm with great difference in performance exist, so that *c* becomes unimportant.

Complexity Terminology

We need a language to describe time complexity of algorithms –

Big O notation.

We can compare the growth function of an algorithm with the grown function of another directly.

Example: # of program steps

→ Algorithm 1: $.01\ n^2 + f(n)$

→ Algorithm 2: $10\ n^2\ h(n)$

For small *n*, Algorithm 1 runs more slowly then Algorithm 2, but as *n* gets larger Algorithm 2 becomes slower.

Complexity Terminology

Interested mostly with the asymptotic bound of a growth function, or the order of a growth function, *g(n);* denoted *O(g(n)).*

Definition: Given two functions: *f(n)* and *g(n),*

f(n) is on the order of *g(n),* if there exists positive integers *a*, *b* such that:

$f(n) \leq a.\ g(n)$ for all $n \geq b$

f(n) is *0(g(n))*

Example:

$f(n) = .01\ n^2 + 100^n$

$g(n) = n^2$

Is $f(n)\ 0(g(n))$

$0(n^2)$?

$f(n) \leq a\ g(n)$ for all $n \geq b$

$.01\ n^2 + 100n \leq a\ n^2$?

$a = 1$

$n \geq 102 = b$

Complexity Terminology

if $f(n) = n^2 + 100n$ ls $f(n)$ $0(n^2)$

$n^2 + 100n \leq an^2$ for all $n \geq b$

$a = 2$

$b = 100$

$n^2 + 100n \leq 2n^2$ $n \geq 100$

Also $f(n)$ is $0(h(n))$ where $h(n) = n^3$ because:

$n^2 + 100n \leq an^3$ for all $n \geq b$

$a = 1$

$b = 11$

$f(n)$ is $0(n^3)$

Also $f(n)$ is $0(t(n))$ where $t(n) = 2^n$ etc.

Complexity Terminology

If *f(n)* is *0(g(n))* then eventually *f(n) < g(n);*

f(n) is bounded by *g(n)*

(asymptotically upper bounded).

Transitivity Property:

If *f(n)* is *0(g(n))* and *g(n)* is *0(h(n))* then *f(n)* is *0(h(n))*

Efficiency Terminology (Summary)

0(1) means a constant computing time.

0(log n) is called logarithmic

0(√n)

0(n) is called linear.

0(n log n)

0(n√n)

0(n^2) is quadratic.

0(n^3) is cubic.

0(2^n) is exponential.

0(n^n)

log n	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Table 1.7 Function values

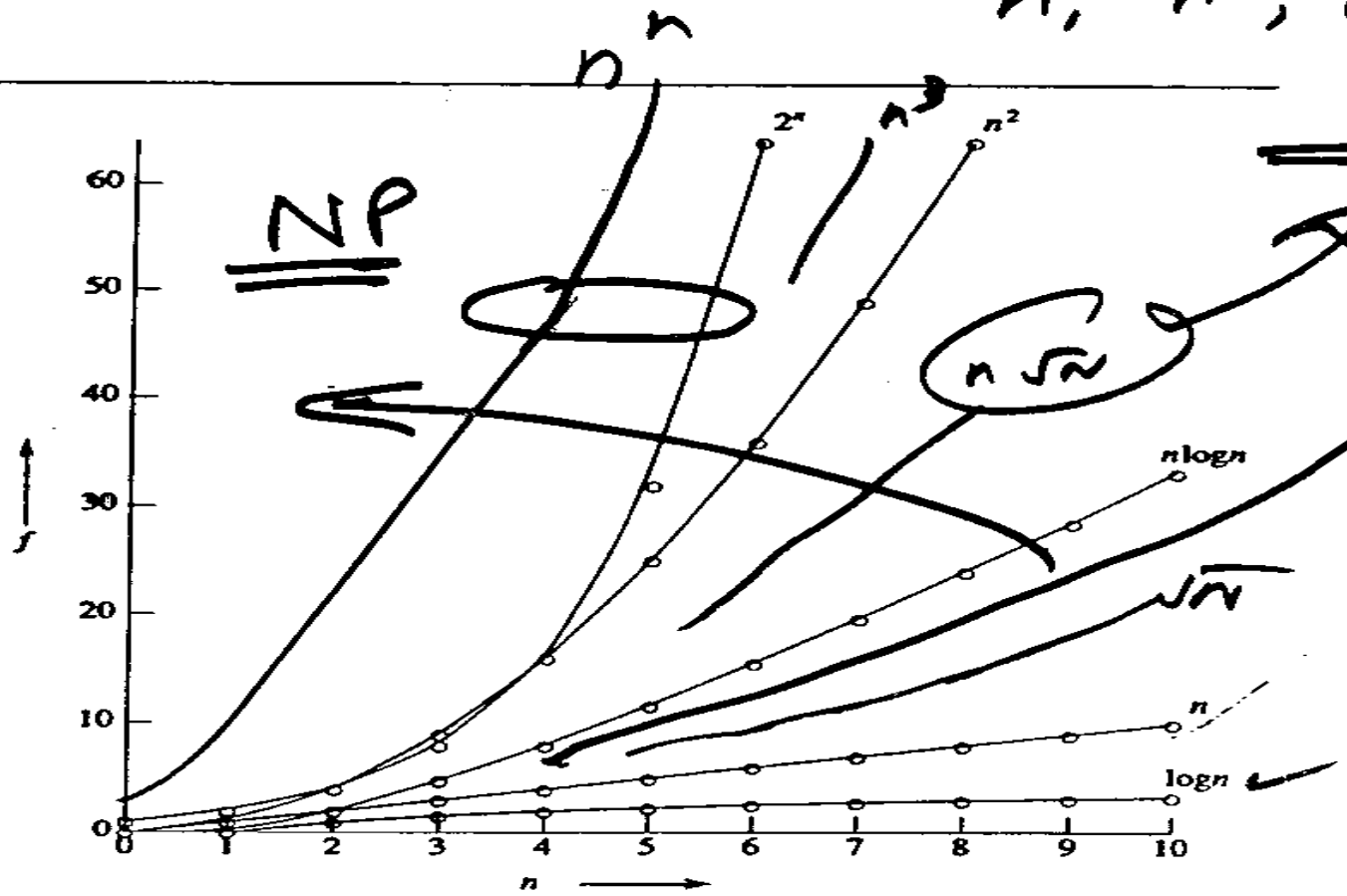


Figure 1.4 Plot of function values