Dynamic Programming

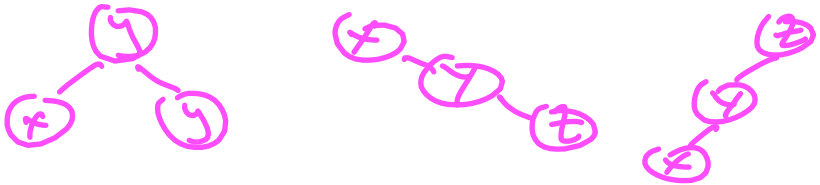
Optimal Binary Search Trees: Problem Definition

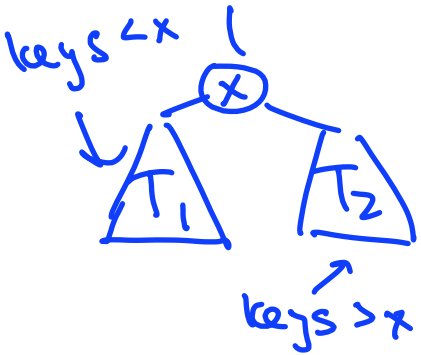
Algorithms: Design and Analysis, Part II

A Multiplicity of Search Trees

Recall: for a given set of keys, there are lots of valid search trees.

[say $x < y < z$]





the search tree property

Question: what is the "best" search tree for a given set of keys?

A good answer: a balanced search tree, like a red-black tree.

(recall Part I)

$\Rightarrow$ worst-case search time = $O(\text{height}) = O(\log n)$

Exploiting Non-Uniformity

Question: Suppose we have keys $x < y < z$ and we know that: 80% of searches are for x, 10% of searches are for y, 10% of searches are for z.

$.8 \cdot 2 + .1 \cdot 1 + .1 \cdot 2 = 1.9$

What is the average search time (i.e., number of nodes looked at) in the trees  and 

respectively?

(A) 2 and 3 (B) 2 and 1

(C) 1.9 and 1.2 (D) 1.9 and 1.3

Problem Definition

Input: frequencies $p_1, p_2, \ldots, p_n$ for items $1, 2, \ldots, n$.

[assume items in sorted order, $1 < 2 < 3 < \cdots < n$]

Goal: Compute a valid search tree that minimizes the weighted (average) search time.

$$C(T) = \sum_{\text{items } i} p_i \cdot [\text{search time for } i \text{ in } T]$$

[search time for i in T] = depth of i in T + 1

(assuming $\sum p_i = 1$)

Example: if T is a red-black tree, then $C(T) = O(\log n)$.

Comparison with Huffman Codes

Similarities

- output = a binary tree
- goal is (essentially) to minimize average depth with respect to given probabilities

Differences:

- with Huffman codes, constraint was prefix-freeness [i.e., symbols only at leaves]
- here, constraint = search tree property [seems harder to deal with]