Dynamic Programming

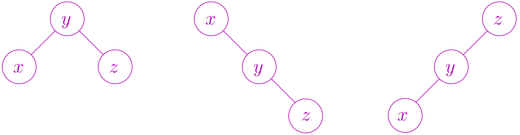
Algorithms: Design and Analysis, Part II

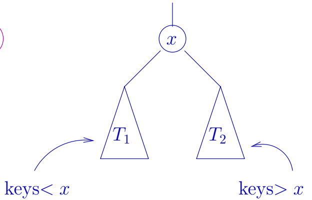
Optimal Binary Search Trees: Problem Definition

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 $= \Theta(\text{height}) = \Theta(\log n)$

Exploiting Non-Uniformity

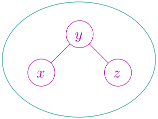
Question: Suppose we have keys $x < y < z$ and we know that:

80% of searches are for x

10% of searches are for y

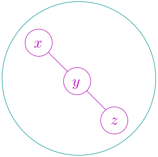
10% of searches are for z

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



$0.8 \cdot 2 + 0.1 \cdot 1 + 0.1 \cdot 2 = 1.9$

and



$0.8 \cdot 1 + 0.1 \cdot 2 + 0.1 \cdot 3 = 1.3$

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, $1 < 2 < \ldots < n$]

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

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

Depth of i in $T + 1$

Example: If T is a red-black tree, then $C(T) = O(\log n)$.
(Assuming $\sum_i p_i = 1$.)

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]