# CS4311
# Design and Analysis of Algorithms

## Lecture 11: Dynamic Programming III

# Writing a Translation Program

- Suppose we want to design a program to translate English texts on food to Chinese
- First problem to solve:

Given an English word, can we quickly search for its Chinese equivalent?

E.g., Apple → 蘋果, Banana → 香蕉,
Pizza → 比薩, Burger → 漢堡,
Hotdog → 熱狗, Spaghetti → 意大利麵

# Writing a Translation Program

- However, some English words may not have a Chinese equivalent
  - In this case, we report not found

- E.g., Biryani (a South Asian dish)
  Burrito (a common Mexican food)
  Jambalaya (a famous Louisiana dish)
  Okonomiyaki (a kind of Japanese pizza)

# Writing a Translation Program

- Let $n$ = # of English words in our database with Chinese equivalent

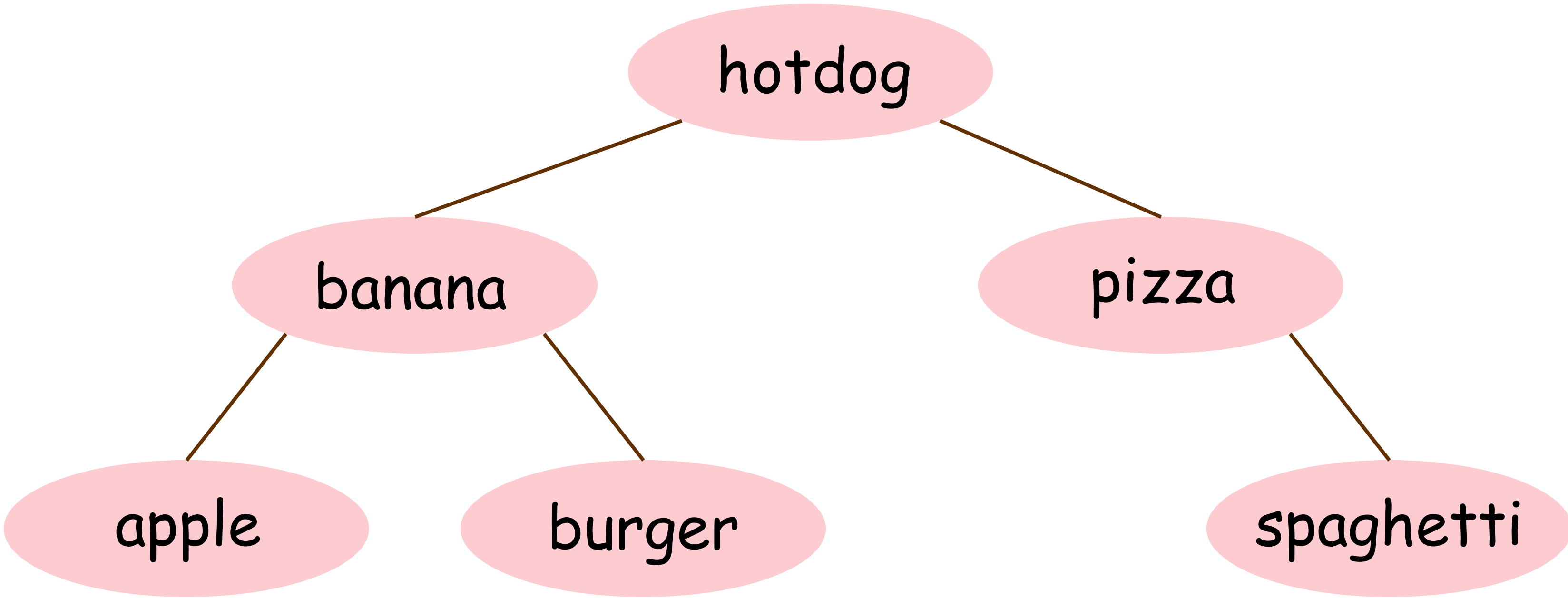
Solution 1: Hashing

- Good, but need a good hash function

Solution 2: Balanced Binary Search Tree

- worst-case $O(\log n)$ time per query

# Balanced Binary Search Tree




Keys = words in the database

# Writing a Translation Program

- In real life, different words may be searched with different frequencies

  E.g., apple may be more often than pizza

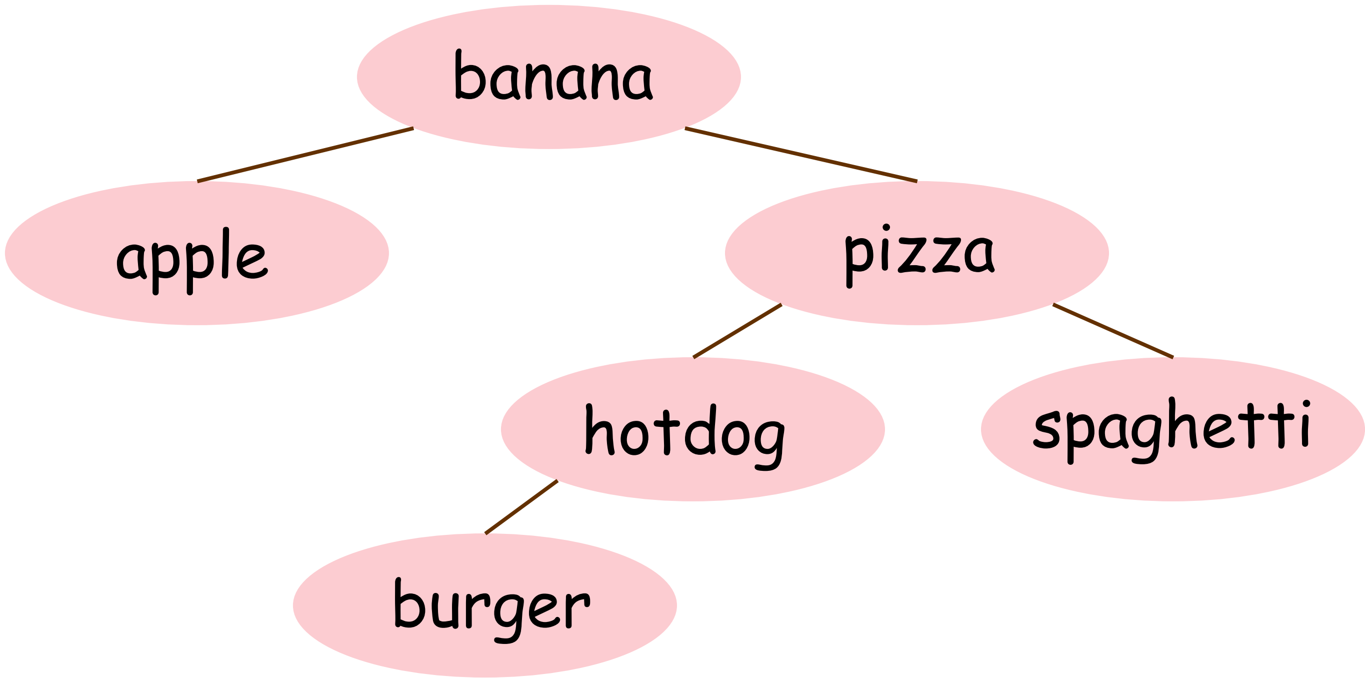
- Also, there may be different frequencies for the unsuccessful searches

  E.g., we may unluckily search for a word in the range (hotdog, pizza) more often than in the range (spaghetti, $+\infty$)

- Suppose your friend in Google gives you the probabilities of what a search will be:

| | | | |
|---|---|---|---|
| < apple | 0.01 | = hotdog | 0.02 |
| = apple | 0.21 | (hotdog, pizza) | 0.04 |
| (apple, banana) | 0.10 | = pizza | 0.04 |
| = banana | 0.18 | (pizza, spaghetti) | 0.11 |
| (banana, burger) | 0.05 | = spaghetti | 0.07 |
| = burger | 0.01 | > spaghetti | 0.04 |
| (burger, hotdog) | 0.12 | | |

- Given these probabilities, we may want words that are searched more frequently to be nearer the root of the search tree

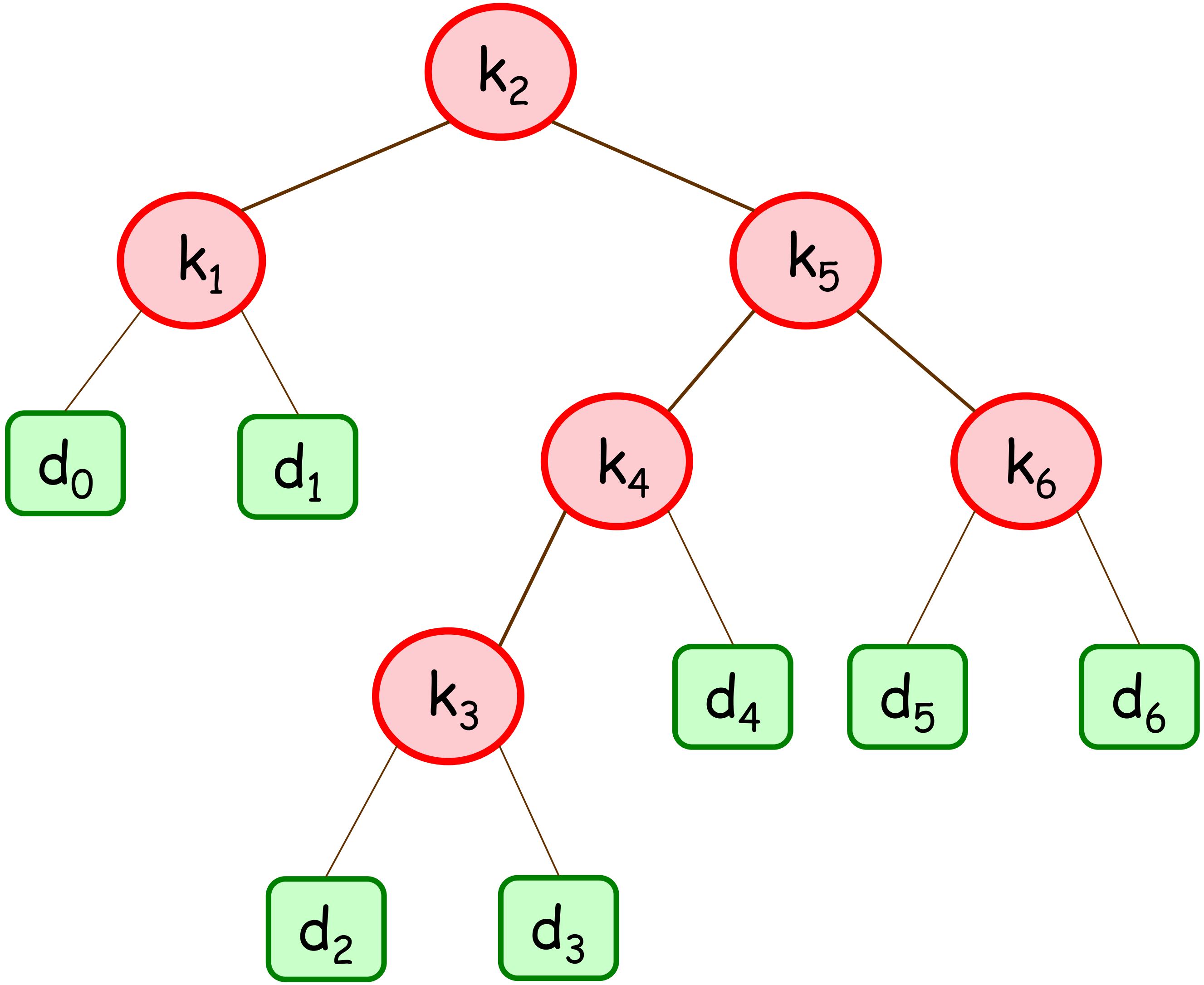



This tree has better expected performance

# Expected Search Time

- To handle unsuccessful searches, we can modify the search tree slightly (by adding dummy leaves), and define the expected search time as follows:
- Let $k_1 < k_2 < \ldots < k_n$ denote the $n$ keys, which correspond to the internal nodes
- Let $d_0 < d_1 < d_2 < \ldots < d_n$ be dummy keys for ranges of the unsuccessful search

  → dummy keys correspond to leaves




Search tree of Page 9 after modification

# Search Time

Lemma: Based on the modified search tree:

- when we search for a word $k_i$,

  search time = node-depth($k_i$)

- when we search for a word in range $d_j$,

  search time = node-depth($d_j$)

# Expected Search Time

- Let $p_i$ = Pr( $k_i$ is searched )
- Let $q_j$ = Pr( word in $d_j$ is searched )

So, $$\sum_i p_i + \sum_j q_j = 1$$

Expected search time

$$= \sum_i p_i \text{ node-depth}(k_i) + \sum_j q_j \text{ node-depth}(d_j)$$

# Optimal Binary Search Tree

Question:

Given the probabilities $p_i$ and $q_j$,
can we construct a binary search tree
whose expected search time is minimized?

Such a search tree is called an
Optimal Binary Search Tree

# Optimal Substructure

Let $T$ = optimal BST for the keys

$$( k_i, k_{i+1}, \ldots , k_j; d_{i-1}, d_i, \ldots , d_j ).$$

Let $L$ and $R$ be its left and right subtrees.

Lemma: Suppose $k_r$ is the root of $T$. Then,

- $L$ must be an optimal BST for the keys
  $$(k_i , k_{i+1}, \ldots , k_{r-1}; d_{i-1}, d_i, \ldots , d_{r-1})$$
- $R$ must be an optimal BST for the keys
  $$(k_{r+1}, k_{r+2}, \ldots , k_j; d_r, d_{r+1}, \ldots , d_j)$$

# Optimal Substructure

Let $E_{i,j}$ = expected time spent with the keys ( $k_i$, $k_{i+1}$, …, $k_j$; $d_{i-1}$, $d_i$, …, $d_j$ ) in optimal BST

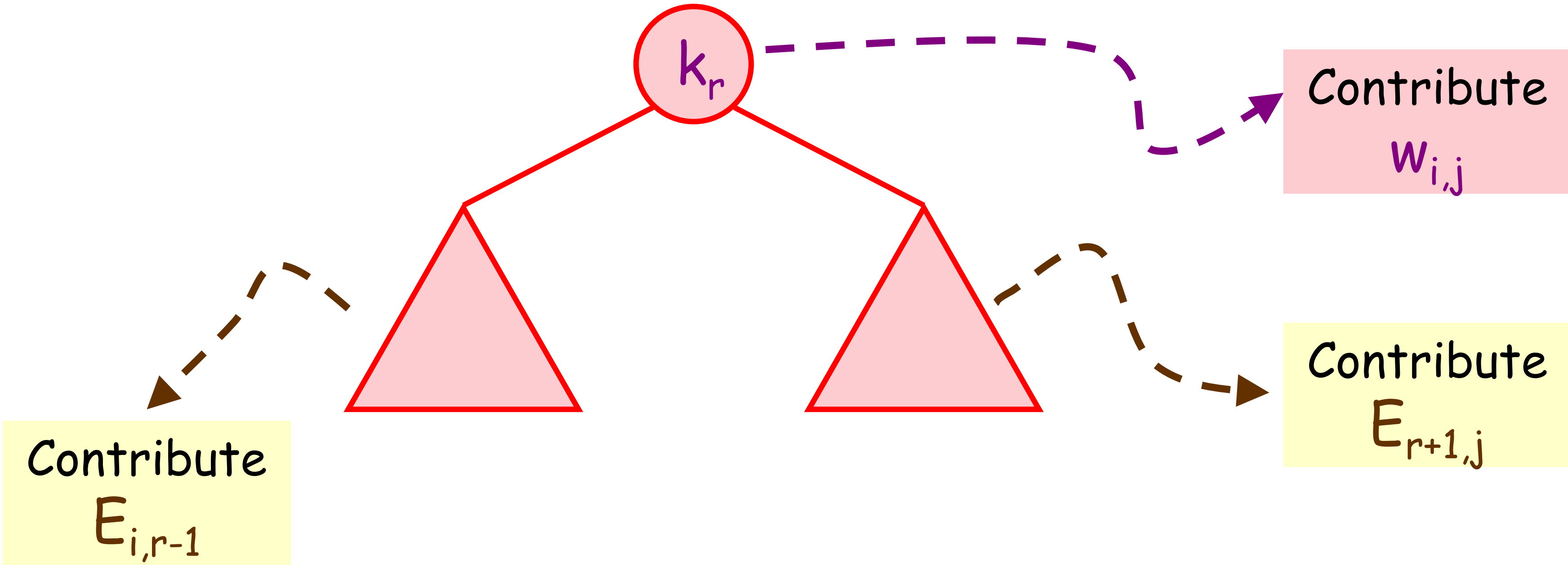
Let $w_{i,j} = \sum_{s=i \text{ to } j} p_s + \sum_{t=i-1 \text{ to } j} q_t$

= sum of the probabilities of keys ( $k_i$, $k_{i+1}$, … , $k_j$; $d_{i-1}$, $d_i$, … , $d_j$ )

# Optimal Substructure

**Lemma:** For any $j \geq i$,

$$E_{i,j} = \min_r \{ E_{i,r-1} + E_{r+1,j} + w_{i,j} \}$$



# Optimal Binary Search Tree

Define a function Compute_E(i,j) as follows:

```
Compute_E(i, j)   /* Finding e_{i,j} */
  1. if (i == j+1) return q_j;   /* Exp time with key d_j */
  2. min = ∞;
  3. for (r = i, i+1, ..., j) {
        g = Compute_E(i,r-1) + Compute_E(r+1,j) + w_{i,j} ;
        if (g < min) min = g;
     }
  4. return min ;
```

# Optimal Binary Search Tree

Question: We want to get Compute_E(1,n)
What is its running time?

- Similar to Matrix-Chain Multiplication, the recursive function runs in $\Omega(3^n)$ time

- In fact, it will examine at most once for all possible binary search tree → Running time = O(C(2n-2,n-1)/n)

Catalan Number

# Overlapping Subproblems

Here, we can see that :

To Compute_E(i,j) and Compute_E(i,j+1), there are many COMMON subproblems:

Compute_E(i,i+1), …, Compute_E(i,j-1)

So, in our recursive algorithm, there are many redundant computations !

Question: Can we avoid it ?

# Bottom-Up Approach

- Let us create a 2D table $E$ to store all $E_{i,j}$ values once they are computed
- Let us also create a 2D table $W$ to store all $w_{i,j}$

We first compute all entries in $W$.

Next, we compute $E_{i,j}$ for $j-i = 0,1,2,\ldots,n-1$

# Bottom-Up Approach

BottomUp_E( ) `/* Finding min #operations */`

1. Fill all entries of W
2. for j = 1, 2, …, n, set E[j+1,j] = $q_j$ ;
3. for (length = 0,1,2,…, n-1)

    Compute E[i,i+length] for all i;

    `// From W and E[x,y] with |x-y| < length`

4. return E[1,n] ;

Running Time = $\Theta(n^3)$

# Remarks

- Again, a slight change in the algorithm allows us to get the exact structure of the optimal binary search tree
- Also, we can make minor changes to the recursive algorithm and obtain a memoized version (whose running time is $O(n^3)$)