CS2351
Data Structures

Tutorial :
Optimal Binary Search Tree

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

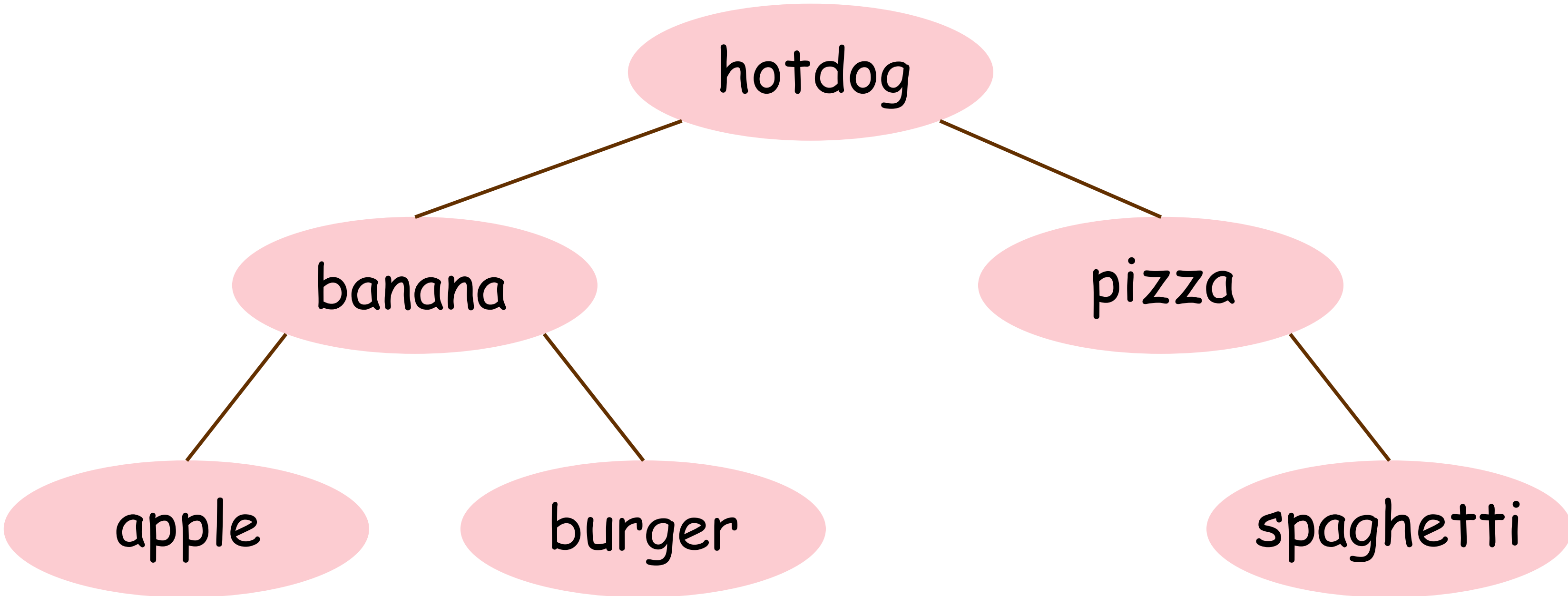
Writing a Translation Program

- 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
- 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

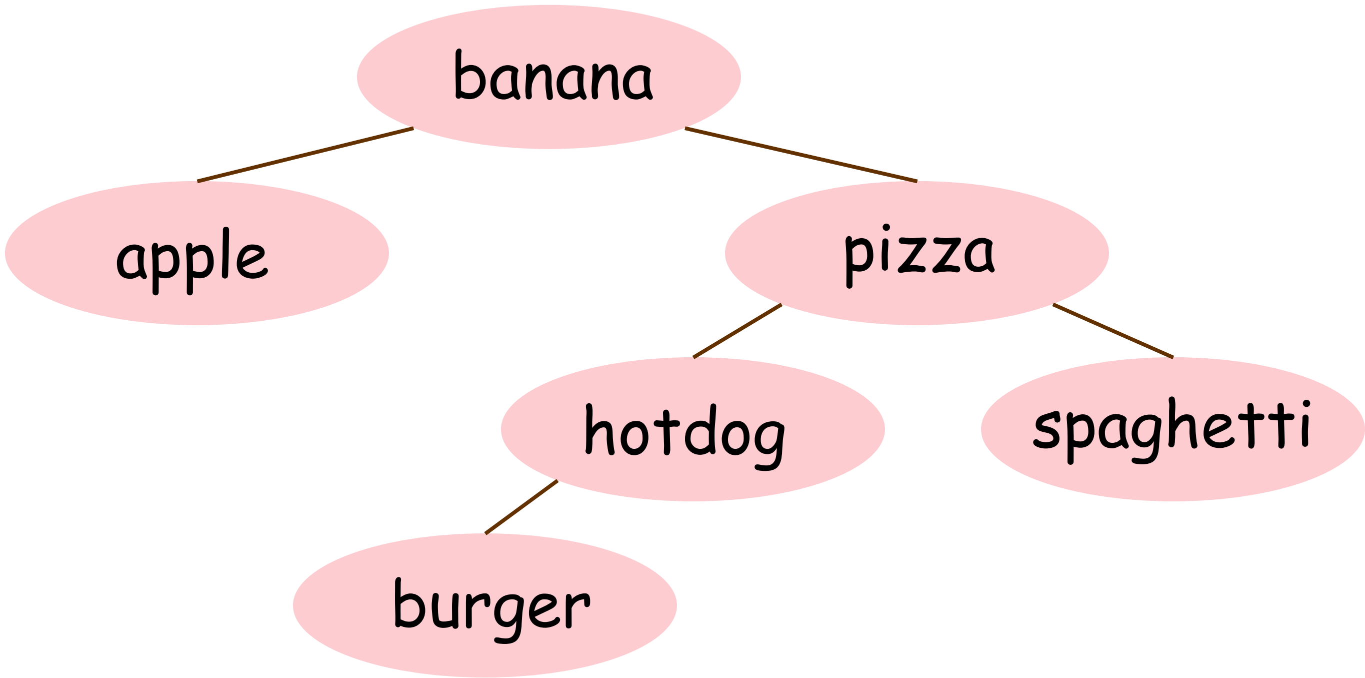
  E.g., we may unluckily search for a word in the range (hotdog, pizza) more often than in the range (spaghetti, $+\infty$)

- 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 ☆		

☆ Frequently searched

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

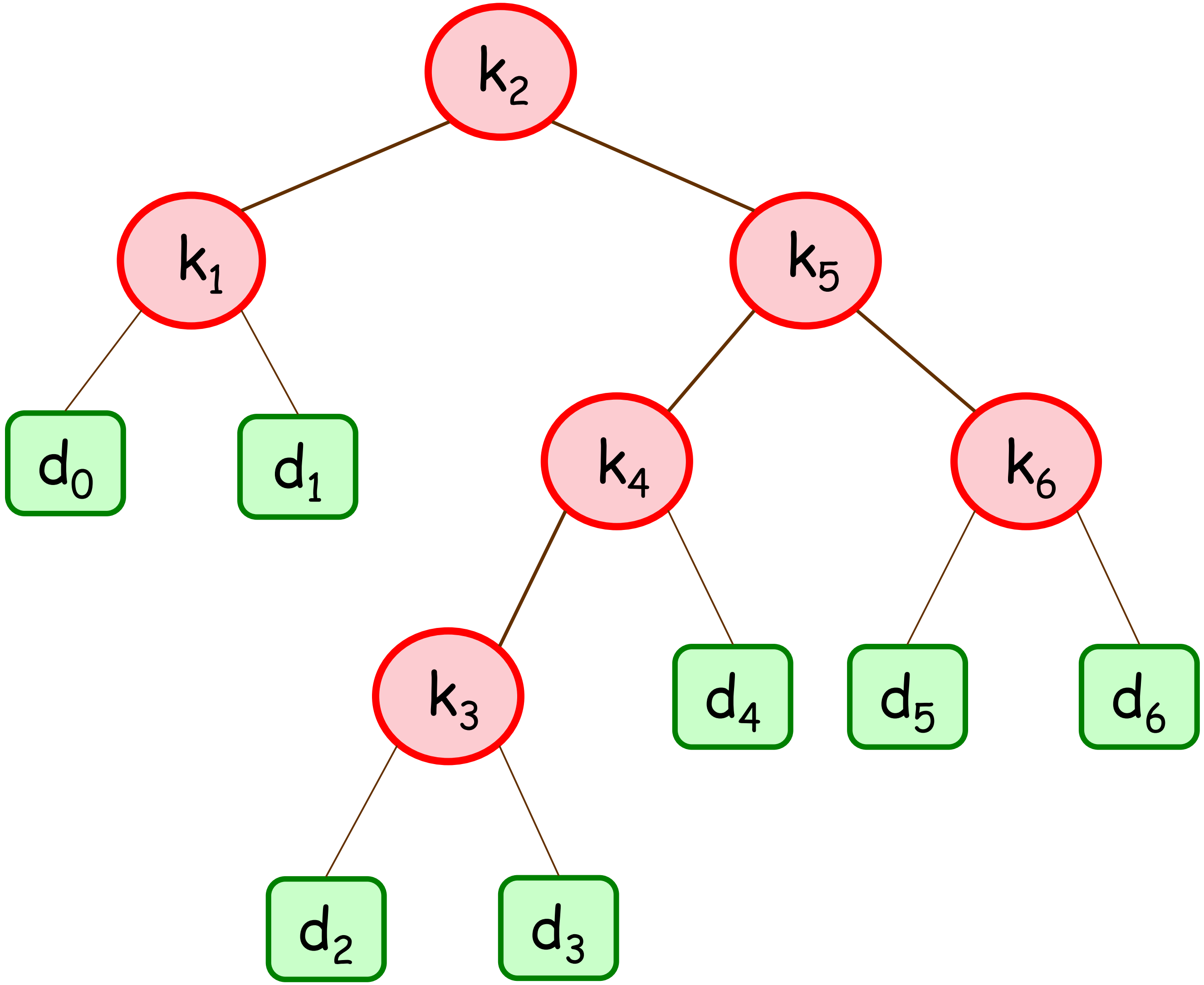


This tree has better expected performance

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

$= \sum_i p_i$ node-depth(k_i) $+ \sum_j q_j$ 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 (OBST)

Property of OBST

Let T = OBST for the keys

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

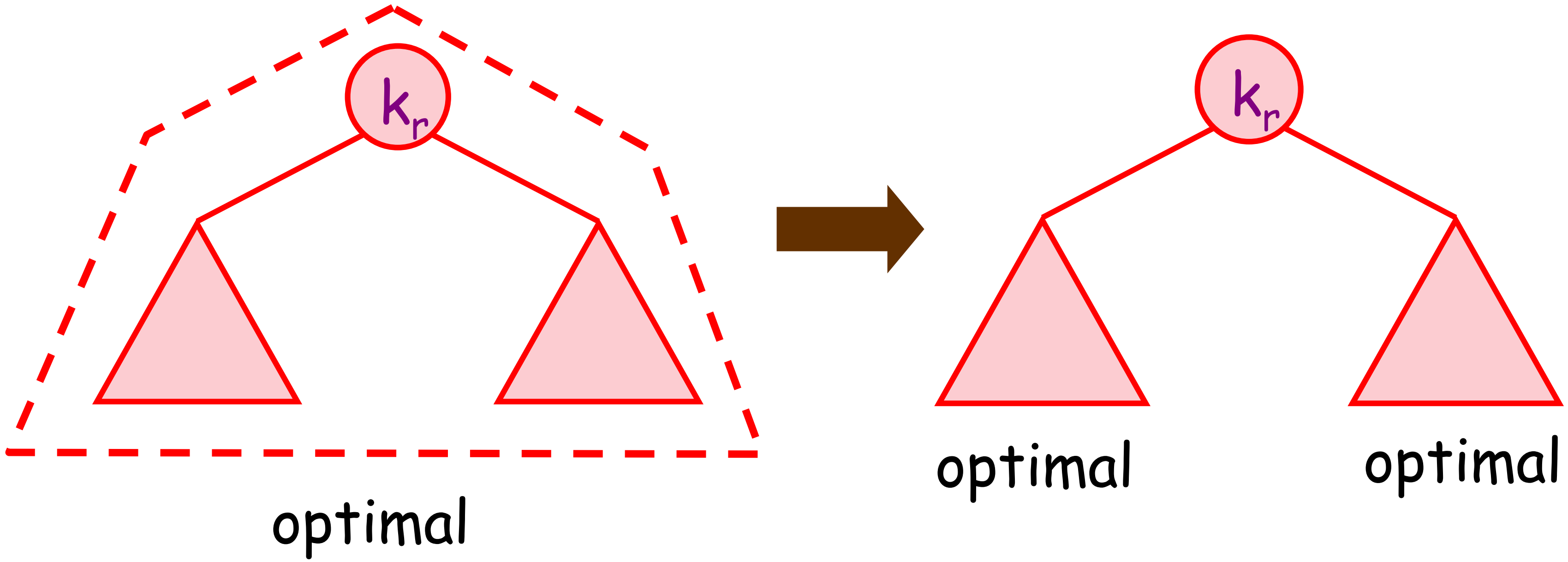
whose root stores k_r

Let L and R be its left and right subtrees

Lemma:

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

Property of OBST



Proof : By contradiction

Deciding Root of OBST

- To find the OBST, our idea is to decide its root, and also the root of each subtree
- To help our discussion, we define :

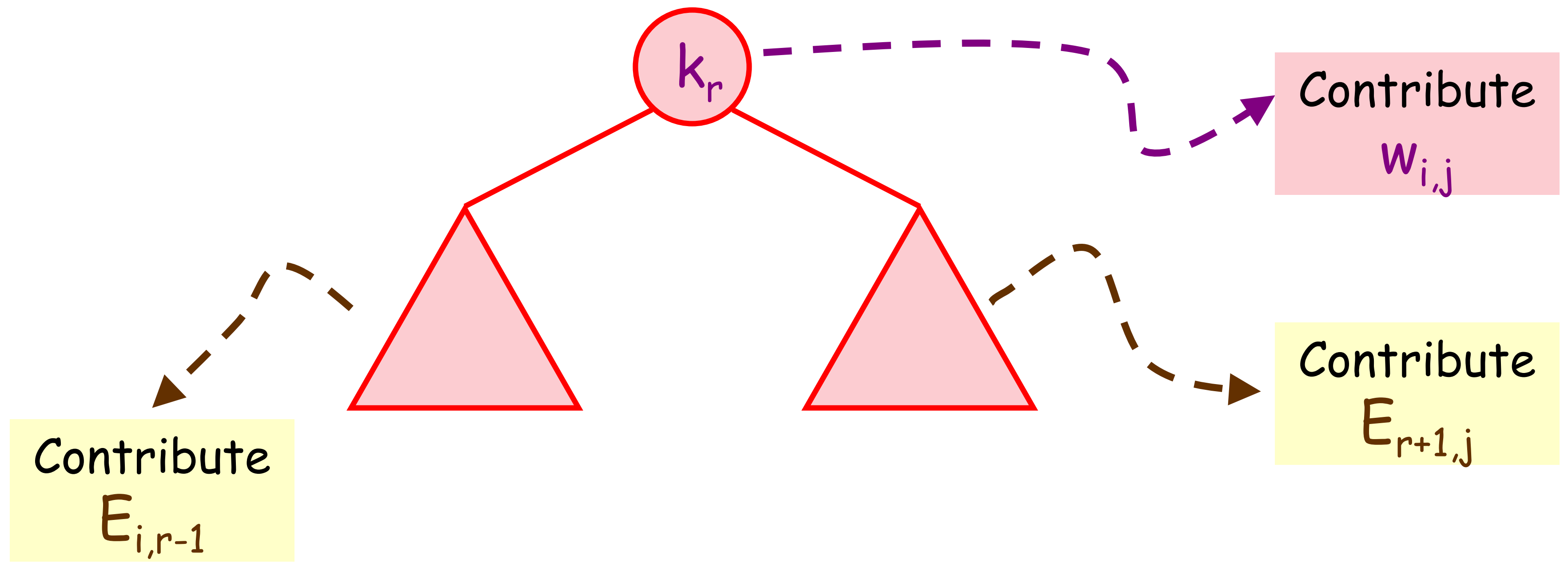
$E_{i,j}$ = expected time searching keys in
(k_i, k_{i+1}, …, k_j; d_{i-1}, d_i, …, d_j)

$w_{i,j}$ = $\sum_{s=i \text{ to } j} p_s + \sum_{t=i-1 \text{ to } j} q_t$
= sum of the probabilities of keys
(k_i, k_{i+1}, … , k_j; d_{i-1}, d_i, … , d_j)

Deciding Root of OBST

Lemma: For any $j \geq i$,

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



Deciding Root of OBST

Corollary:

Let r be the parameter that minimizes

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

Then the root of the OBST for keys

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

should be set to k_r

Computing $E_{i,j}$

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 ;
```

Computing $E_{i,j}$

Question: What is its running time?

- It has a recurrence of the following form :

$$T(i, j) = \sum_r T(i, r-1) + \sum_r T(r+1, j) + 1$$

$$T(i, i) = 1$$

- By substitution, we find $T(i, j) = \Omega(3^{j-i})$

➔ Compute_E(1,n) takes $\Omega(3^n)$ time

Computing $E_{i,j}$ faster

- Recall that when we use recursion to compute the n^{th} Fibonacci number, it runs in exponential time
- But we can speed it up to $O(n)$ time. Why ?

Key idea : avoid redundant computations (by storing computed terms in a table)

- We now apply the same idea to speed up the computation of $E_{i,j}$

Bottom-Up Approach

- We use a 2D table E to store $E_{i,j}$ once they are computed
- We also use a 2D table W to store $w_{i,j}$
- The algorithm works as follows :

1. We first compute all entries in W
   ➔ This is done in $O(n^2)$ time (how?)
2. We compute $E_{i,j}$ for $j-i = 0,1,2,\ldots,n-1$

Bottom-Up Approach

```
BottomUp_E( )      /* Finding E_{i,j} */
  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

- A slight change in the algorithm allows us to get the root of each subtree, and thus the structure of OBST (how?)
- The powerful technique of storing computed terms in a table is called Dynamic Programming
- Knuth observed a further property so that we can compute OBST in $O(n^2)$ time (search wiki for more information)