CSE331 Introduction to Algorithms
Lecture 13
Optimal Binary Search Trees

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

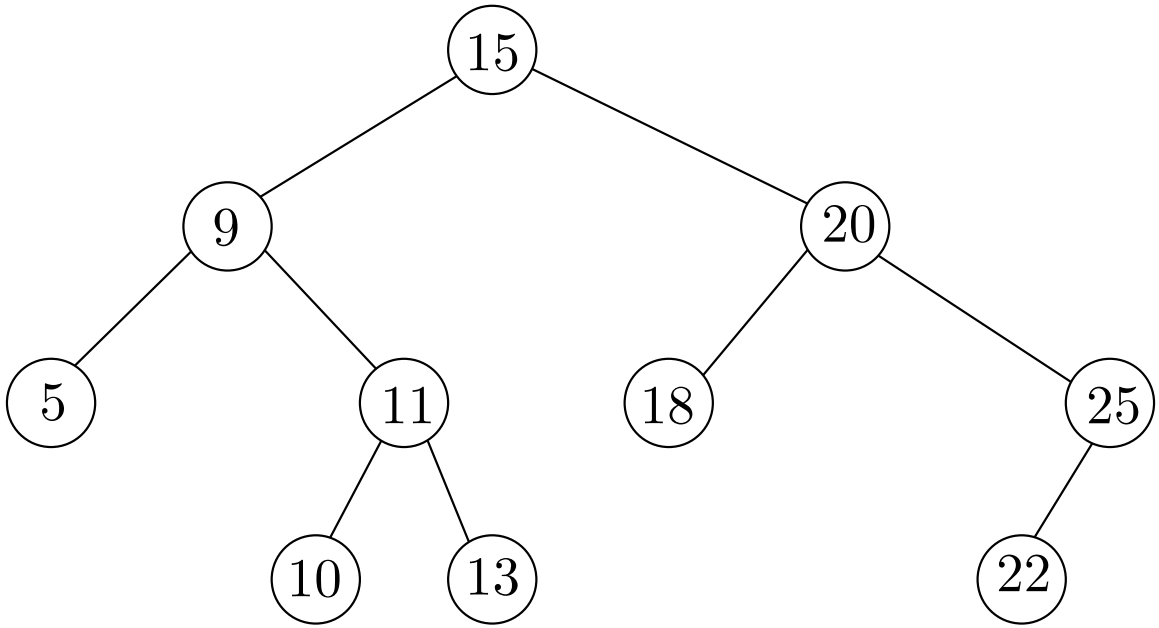
July 11, 2017

1. Introduction
2. Binary search trees
3. Optimal binary search trees
4. Structure of an Optimal BST
5. Recursive Solution
6. Dynamic programming
7. Computing an optimal solution
8. Conclusion

Introduction

- In this lecture, we will show how to compute optimal *binary search trees*.
- We will solve it using dynamic programming.
- The lecture will start a short review of binary search trees.
- **Reference**: Section 12 (p. 286) and Section 15.5 of the textbook (p. 397)
  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

Binary Search Trees

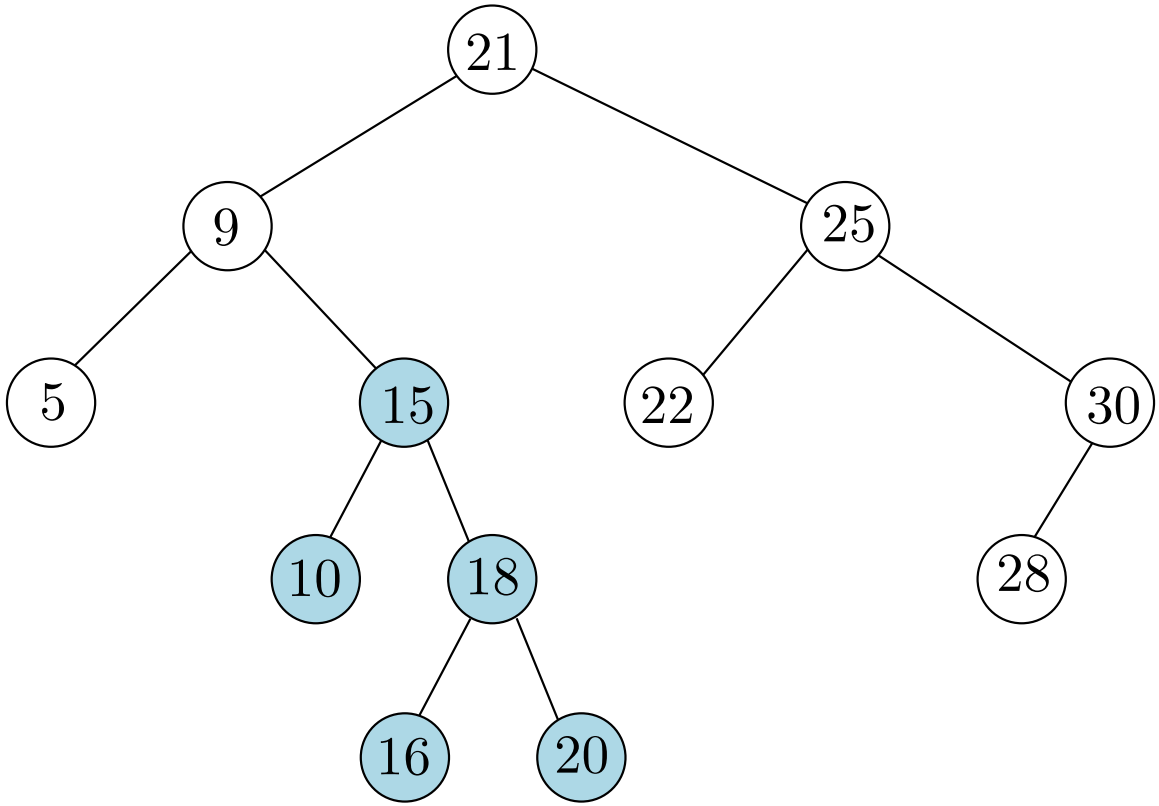


Definition (Binary search tree)

A *binary search tree (BST)* T is a binary tree that records at each node v a key $\text{key}(v)$. Every node v of T has the following properties.

- For every node u in the left subtree of v, we have $\text{key}(u) \leqslant \text{key}(v)$.
- For every node w in the right subtree of v, we have $\text{key}(w) \geqslant \text{key}(v)$.

Subtrees of a BST



- BST with set of keys $\{5, 9, 10, 15, 16, 18, 20, 21, 22, 25, 28, 30\}$.

Subtrees of a BST

Proposition

The keys stored in a subtree T' of a binary search tree T are consecutive. So if the keys of T are $k_1 < k_2 < \cdots < k_n$, then T' records $k_i < k_{i+1} < \cdots < k_j$ for some $i, j \in \{1, \ldots, n\}$.

Proof.

Done in class. □

Binary Search Trees

Implementation

A node v of a BST records the following fields:

- key(v) — the key of v
- left(v) — pointer to the left child of v
- right(v) — pointer to the right child of v

The pointer left(v) or right(v) is set to NIL if the corresponding child does not exist.

- Node v may also record satellite data
- For instance, if T records points (x, y, z), the key could be x and (y, z) could be the satellite data.
- In this lecture we do not use satellite data.

Insertion into a BST

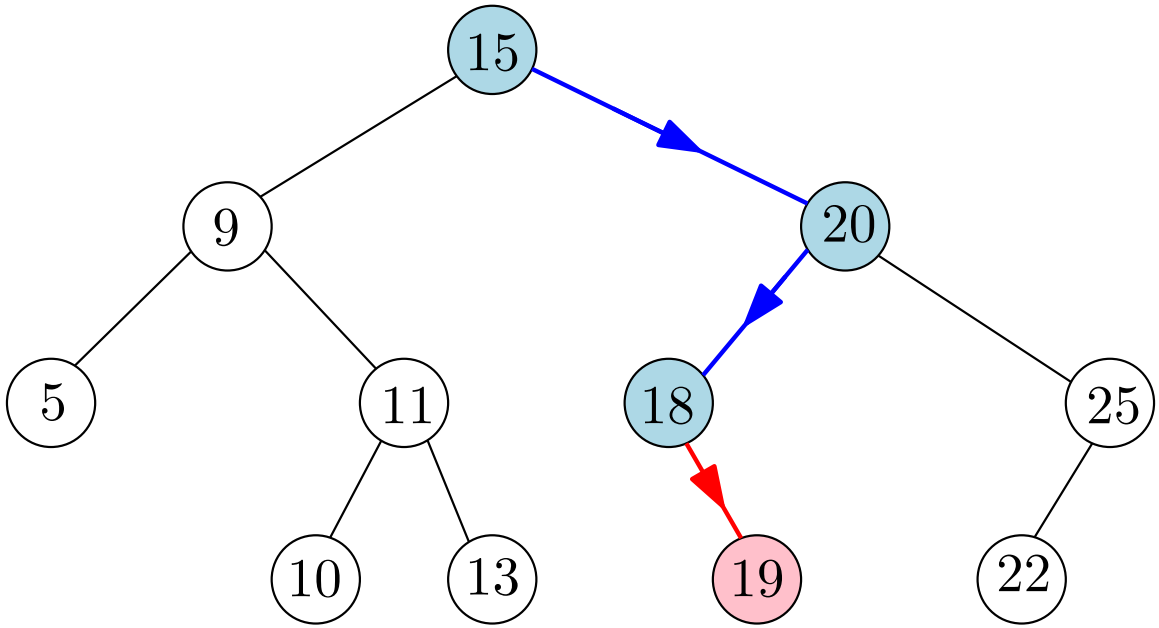
Inserting key k into a BST

```
1: procedure INSERT(r, k)
2:     if r = NIL then
3:         r ← NEWNODE(k)
4:     else if k < key(r) then
5:         INSERT(left(r), k)
6:     else
7:         INSERT(right(r), k)
```

- The new key k is inserted from the *root* node r of the tree T.
- The root node is the only node without parent.
- Insertion takes $O(h+1)$ time, where h is the height of the tree.

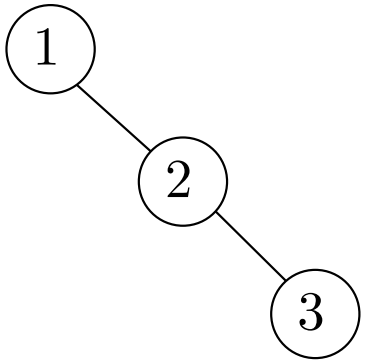
BST Insertion: Example

- Inserting 19 into the tree from Slide 4

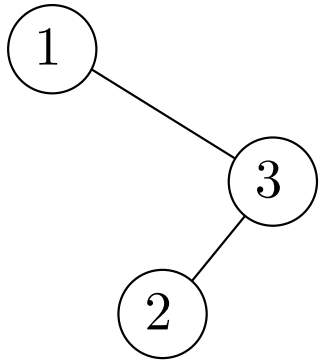


BST Insertion Orders

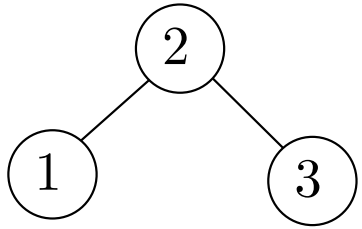
- The shape of a BST depends on the order of insertions.



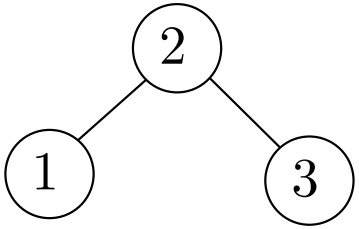
$1 \to 2 \to 3$



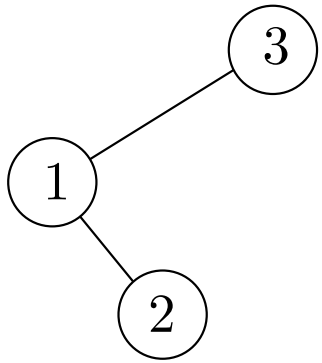
$1 \to 3 \to 2$



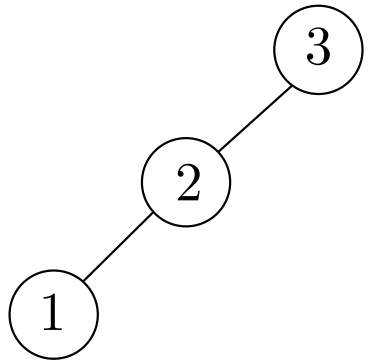
$2 \to 1 \to 3$



$2 \to 3 \to 1$



$3 \to 1 \to 2$



$3 \to 2 \to 1$

In-Order Traversal

- The keys of a binary search tree T can be printed in nondecreasing order by calling the following procedure, called *in-order traversal*, from the root of T.

Pseudocode

```
1: procedure IN-ORDER(v)
2:     if v = NIL then
3:         return
4:     IN-ORDER(left(v))
5:     Print key(v)
6:     IN-ORDER(right(v))
```

- On the BST from Slide 4, it prints:

5 9 10 11 13 15 18 20 22 25

Searching in a BST

Problem (Searching)

Given a binary search tree T and a key k, the searching problem is to decide whether k is the key of a node v of T, and if so, return v.

- The procedure on next slide allows to search in a BST in $O(h+1)$ time, where h is the height of the tree.

Searching in a BST

Pseudocode

```
1: procedure SEARCH(v, k)
2:     if v = NIL then
3:         return NOTFOUND
4:     if k < key(v) then
5:         return SEARCH(left(v), k)
6:     if k > key(v) then
7:         return SEARCH(right(v), k)
8:     return v                                  ▷ k = key(v)
```

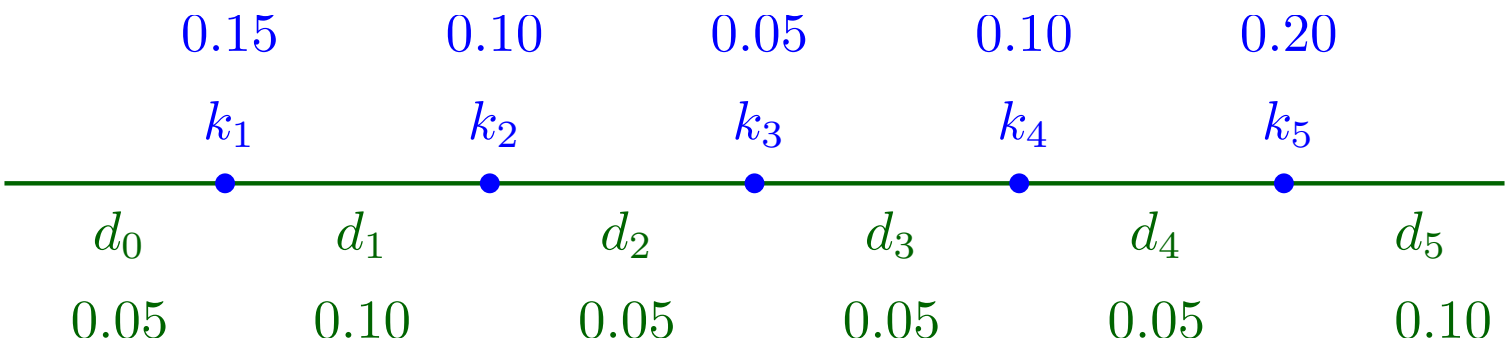
Balanced Binary Search Trees

- A BST with n nodes has height at least $\lfloor \log n \rfloor$, so the (worst case) search time is $\Omega(\log n)$.
- There exist *balanced binary search trees* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is also possible to insert and delete nodes in $\Theta(\log n)$ time in a balanced BST.
  - It requires to rebalance (change the structure) of the BST while inserting/deleting.
- So balanced BST have the same asymptotic search time as a sorted array, and allow efficient insertion/deletion. Sorted arrays, on the other hand, do not allow efficient insertion/deletion.
- Balanced binary search trees are not covered in CSE331.

  (Covered in CSE221 Data structures.)

Optimal Binary Search Trees

- We are given a set of keys $k_1 < k_2 < \cdots < k_n$.
- We want to build a BST on this set of keys, so as to answer search queries efficiently.
- We know in advance the probability distribution of the queries. Example:



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal Binary Search Trees

Input

- A list $k_1 < k_2 < \cdots < k_n$ of n keys.
- The probability p_i that a query falls on k_i, for $i = 1, \ldots, n$.
- The probability q_i that a query is between k_i and k_{i+1}, for $i = 1, \ldots, n$.
- The probability q_0 that a query is less than k_0.
- The probability q_n that a query is more than k_n.

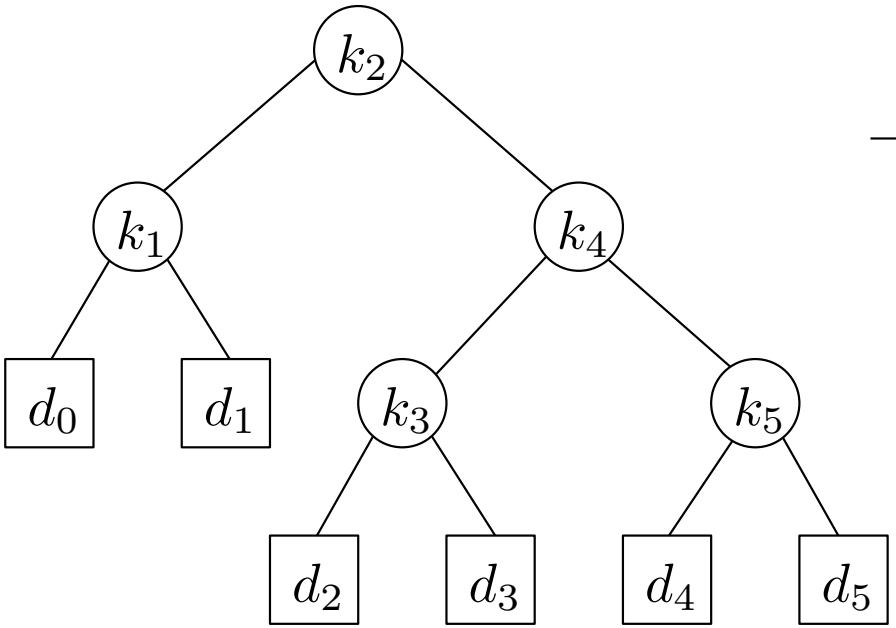
- The probabilities should sum to 1, so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$$

Optimal Binary Search Trees

- Several different BST can record the keys.
- Goal: Construct a BST such that the expected query time is minimized.
- The BST will contain dummy keys d_i at the leaves, each d_i corresponding to the interval that has probability q_i.

Example



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Input table

- Is it a good solution?
- No. Intuitively, we would like k_5 to be closer to the root, as it has the highest probability.

Problem Formulation

- The *depth* of a node is the length of the path from the root.
- We assume that the search time for a node v is proportional to $\text{depth}_T(v) + 1$, which is the number of nodes on the path from the root to v.
- Then the expected search time in a tree T is proportional to

$$\sum_{i=1}^{n} p_i(\text{depth}_T(k_i) + 1) + \sum_{i=0}^{n} q_i(\text{depth}_T(d_i) + 1)$$

$$= 1 + \sum_{i=1}^{n} p_i \,\text{depth}_T(k_i) + \sum_{i=0}^{n} q_i \,\text{depth}_T(d_i).$$

- So our goal is to construct a tree T that minimizes

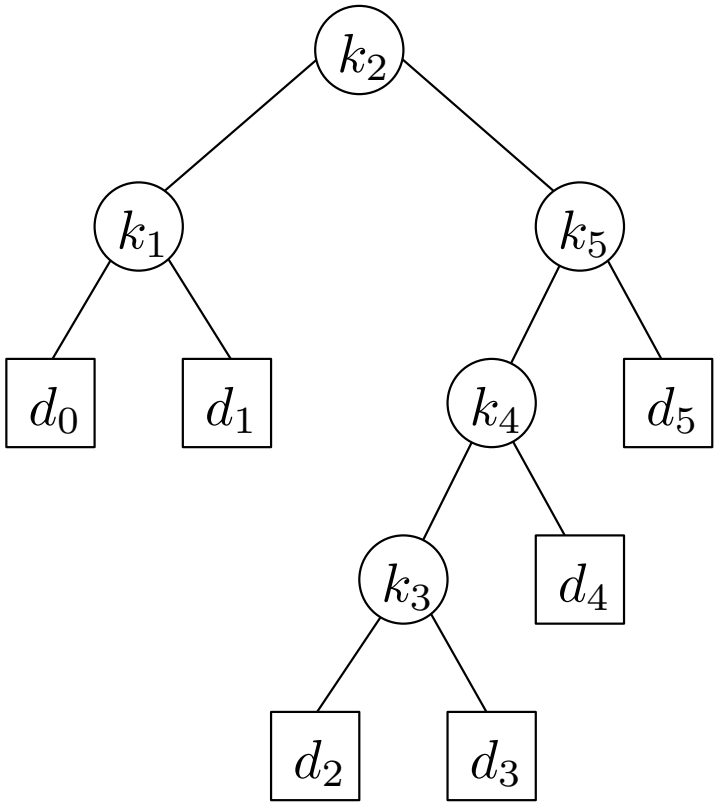
$$\text{cost}(T) = \sum_{i=1}^{n} p_i \,\text{depth}_T(k_i) + \sum_{i=0}^{n} q_i \,\text{depth}_T(d_i). \tag{1}$$

Example

node	depth	probability	contribution
k_1	1	0.15	0.15
k_2	0	0.10	0
k_3	2	0.05	0.10
k_4	1	0.10	0.10
k_5	2	0.20	0.40
d_0	2	0.05	0.10
d_1	2	0.10	0.20
d_2	3	0.05	0.15
d_3	3	0.05	0.15
d_4	3	0.05	0.15
d_5	3	0.10	0.30
Total			1.80

Table: Calculation of cost(T) for the tree T on Slide 18. According to our model, the expected search time will be proportional to $1 + 1.80 = 2.80$

Example



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

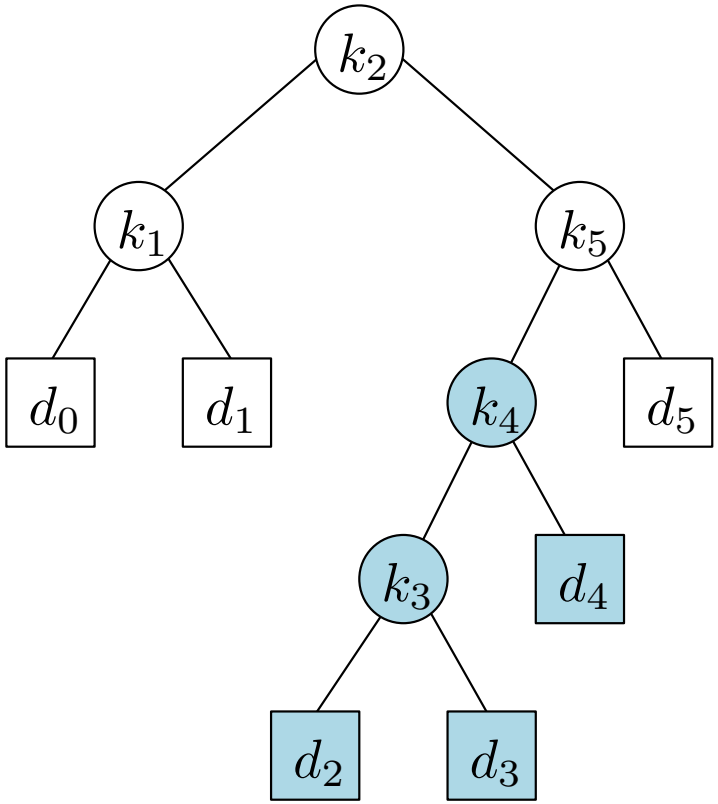
Input table

- An optimal solution with cost 1.75, improving on Slide 18 which has a cost of 1.80. The key k_5, which has highest probability, is closer to the root.

Brute Force Approach

- A brute force approach would enumerate all possibles BSTs over $k_1, \ldots, k_n$, compute their costs, and return the smallest.
- It would take exponential time because there is an exponential number of such BSTs.
  - Proof?
- So we will try dynamic programming.

Structure of an Optimal BST



- The subtree rooted at k_4 contains keys k_3, k_4 and d_2, d_3, d_4.

Structure of an Optimal BST

Lemma

Every subtree is either a single dummy node, or it contains consecutive keys $k_i < k_{i+1} < \cdots < k_j$ and dummy nodes $d_{i-1}, d_i, \ldots, d_j$, for $1 \leqslant i \leqslant j \leqslant n$.

Proof.

Follows from Slide 6. □

Structure of an Optimal BST

- From the lemma above, each subtree of T is a subtree $T[i,j]$ where $j \geqslant i-1$ such that
  - $T[i,j]$ contains keys $k_i, \ldots, k_j$ and $d_{i-1}, \ldots, d_j$ if $i \leqslant j$.
  - $T[i,i-1]$ only contains the dummy key d_{i-1}.
- The *weight* of the subtree $T[i,j]$ is the sum of the probabilities of its nodes. So

$$w[i,j] = \sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell \qquad \text{whenever } 1 \leqslant i \leqslant j \leqslant n$$

$$w[i,i-1] = q_{i-1} \qquad \text{for } i = 1, 2, \ldots, n+1$$

Structure of an Optimal BST

Lemma

If $1 \leqslant i \leqslant j \leqslant n$ *and* $T[i,j]$ *is a subtree of an optimal binary search tree* T*, then* $T[i,j]$ *must be optimal for the subproblem with keys* $k_i, \ldots, k_j$ *and cost function*

$$\text{cost}(T[i,j]) = \sum_{\ell=i}^{j} p_\ell \,\text{depth}_{T[i,j]}(k_\ell) + \sum_{\ell=i-1}^{j} q_\ell \,\text{depth}_{T[i,j]}(d_\ell).$$

Proof

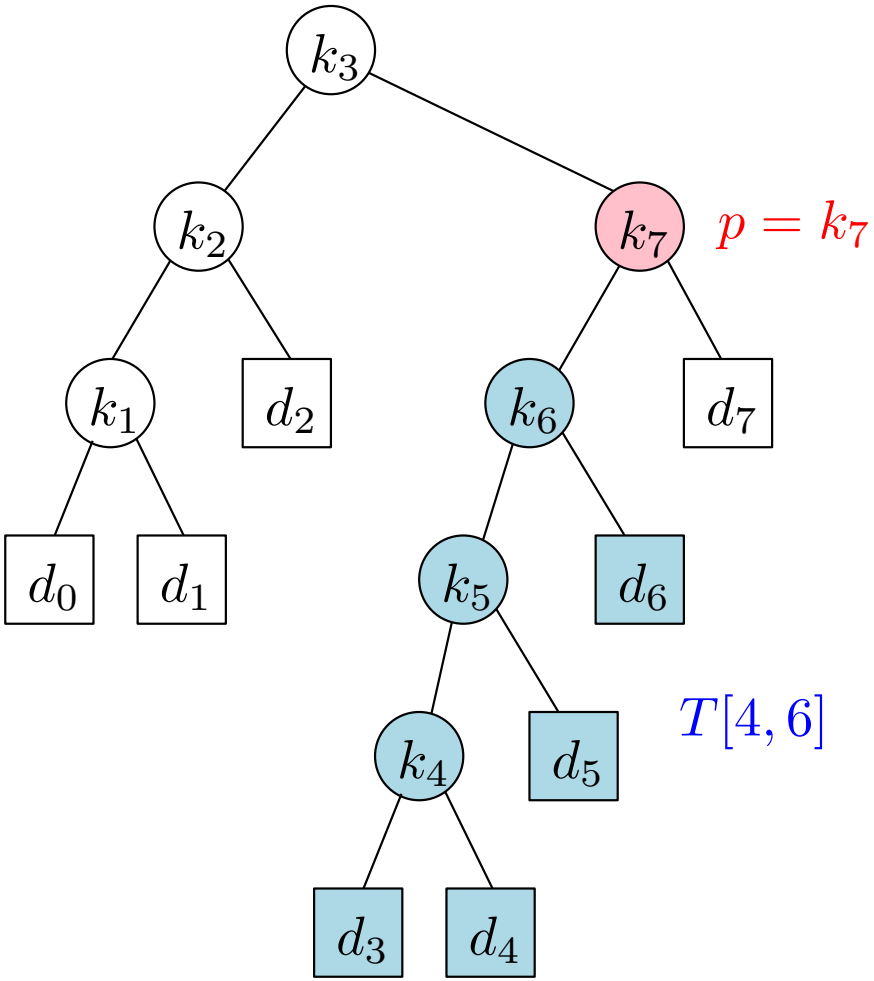


Figure: Tree T and subtree $T[4,6]$

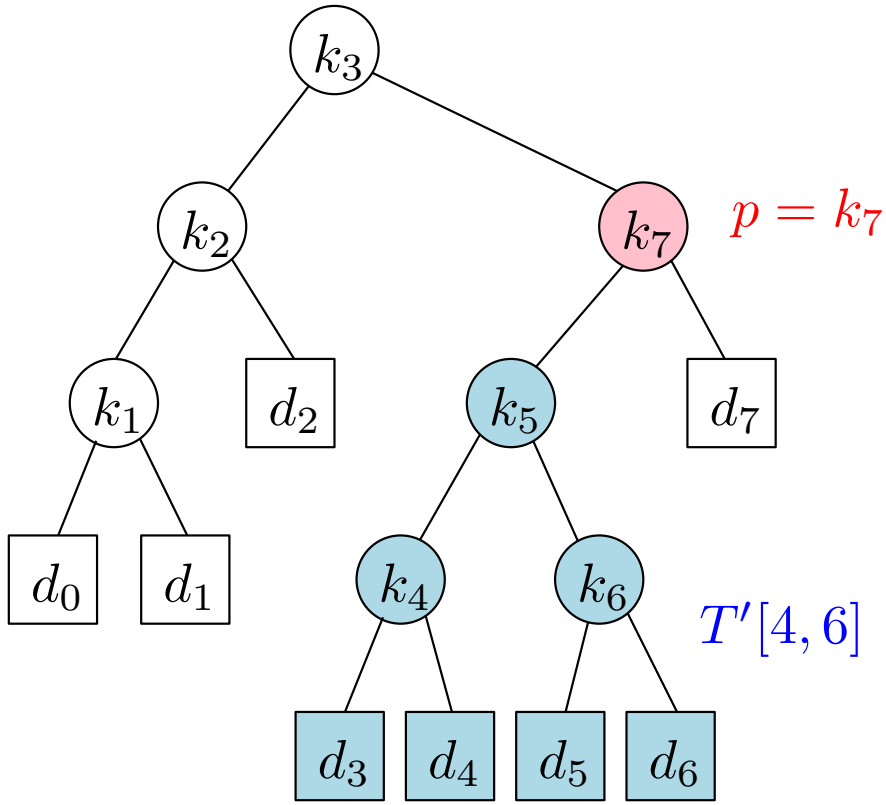


Figure: Tree T' obtained by replacing $T[4,6]$ with $T'[4,6]$

Proof

- We make a proof by contradiction.
- So we assume that $T[i,j]$ is not optimal.
- Then there is a subtree $T'[i,j]$ over the same set of keys such that

$$\text{cost}(T'[i,j]) < \text{cost}(T[i,j]).$$

- Let T' be the tree obtained from T by replacing $T[i,j]$ with $T'[i,j]$.
- Let p be the parent of the root of $T[i,j]$.
- Then p is the parent of the root of $T'[i,j]$.
- For every node v of $T[i,j]$

$$\text{depth}_T(v) = \text{depth}_T(p) + 1 + \text{depth}_{T[i,j]}(v)$$

- We denote $\delta = \text{depth}_T(p) + 1$.

Proof

- Then the contributions of the nodes of $T[i,j]$ to $\text{cost}(T)$ is

$$\sum_{\ell=i}^{j} p_\ell \,\text{depth}_T(k_\ell) + \sum_{\ell=i-1}^{j} q_\ell \,\text{depth}_T(d_\ell)$$

$$= \sum_{\ell=i}^{j} p_\ell \left(\delta + \text{depth}_{T[i,j]}(k_\ell)\right) + \sum_{\ell=i-1}^{j} q_\ell(\delta + \text{depth}_{T[i,j]}(d_\ell))$$

$$= \delta \cdot \left(\sum_{\ell=i}^{j} p_\ell + \sum_{\ell=i-1}^{j} q_\ell\right) + \text{cost}(T[i,j])$$

$$= \delta \cdot w[i,j] + \text{cost}(T[i,j])$$

Proof

- As the other nodes do not move, we have

$$\begin{aligned} \text{cost}(T') &= \text{cost}(T) + \text{cost}(T'[i,j]) - \text{cost}(T[i,j]) \\ &< \text{cost}(T). \end{aligned}$$

- It contradicts the lemma hypothesis that T is optimal.

Recursive Solution

- Let $c[i,j] = \text{cost}(T[i,j])$ be the cost of an optimal subtree $T[i,j]$ where $j \geqslant i-1$.
- As $T[i,i-1]$ has only one node with depth 0, we have $c[i,i-1] = 0$.
- If k_r is the root of an optimal subtree containing $k_i, \ldots, k_j$, then we have

$$c[i,j] = c[i,r-1] + w[i,r-1] + c[r+1,j] + w[r+1,j] \quad (2)$$

because for each node v of a subtree $T[i,r-1]$

$$\text{depth}_{T[i,j]}(v) = 1 + \text{depth}_{T[i,r-1]}(v),$$

and for each node v of a subtree $T[r+1,j]$

$$\text{depth}_{T[i,j]}(v) = 1 + \text{depth}_{T[r+1,j]}(v).$$

Recursive Solution

As we seek to minimize $c[i, j]$, it follows from Equation (2) that $c[i, j]$ is given by the relation

$$c[i, j] = \min_{i \leqslant r \leqslant j} (c[i, r-1] + w[i, r-1] + c[r+1, j] + w[r+1, j]) \qquad (3)$$

whenever $1 \leqslant i \leqslant j \leqslant n$.

As mentioned in previous slide, we also have

$$c[i, i-1] = 0$$

whenever $1 \leqslant i \leqslant n+1$.

Dynamic Programming

- Equation 3 requires to know the weight $w[i, j]$ of each possible subtree.
- We first compute it in time $\Theta(n^2)$.

Pseudocode

```
1: procedure ComputeWeights(p, q, n)
2:     w[1...n+1][0...n] ← new array
3:     for i ← 1, n+1 do
4:         w[i, i−1] ← q[i−1]
5:         for j ← i, n do
6:             w[i, j] ← w[i, j−1] + p_j + q_j
7:     return w
```

Dynamic Programming

- Equation 3 shows how to compute the optimal cost $c[i, j]$ as a function of $c[i, r-1]$ and $c[r+1, j]$ where $i \leqslant r \leqslant j$.
- It means that $c[i, j]$ only depends on smaller intervals, that is, we only need costs $c[i', j']$ such that $j' - i' < j - i$.
- So we will compute $c[i, j]$ by increasing length $\ell = j - i + 1$.
- Pseudocode on next slide. Comments:
  - The array w in Line 1 is the output of the procedure on Slide 33.
  - Lines 3–4 set cost 0 to all subtrees formed by a single leaf. (See Slide 25.)
  - The outer loop (Lines 5–12) goes by increasing size $\ell = j - i + 1$ of the subproblem.
  - Line 8–12 implement Equation 3.

Dynamic Programming

Pseudocode

```
1:  procedure OptimalCost(p, q, n, w)
2:      c[1...n+1, 0...n] ← new array
3:      for i ← 1, n+1 do
4:          c[i, i−1] ← 0
5:      for ℓ ← 1, n do
6:          for i ← 1, n−ℓ+1 do
7:              j ← i+ℓ−1
8:              c[i, j] ← ∞
9:              for r ← i, j do
10:                 t ← c[i, r−1] + w[i, r−1] + c[r+1, j] + w[r+1, j]
11:                 c[i, j] ← min(t, c[i, j])
12:     return c[1,n]
```

Analysis

- It is easy to see that the running time is $O(n^3)$ because of the three nested loops that are iterated at most n times each.
- As each subproblem $T[i,j]$ is solved in $\Theta(j-i+1)$ time, the total running time is $\Theta(f(n))$ where

$$f(n) = \sum_{1 \leqslant i \leqslant j \leqslant n} j - i + 1.$$

- On next slide, we argue that $f(n) = \Omega(n^3)$, which completes the proof that the running time is $\Theta(n^3)$.

Analysis

- First assume that n is a multiple of 4.

$$f(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} j - i + 1 \geqslant \sum_{i=1}^{n/4} \sum_{j=3n/4}^{n} j - i + 1$$

$$\geqslant \sum_{i=1}^{n/4} \sum_{j=3n/4}^{n} \frac{n}{2} \geqslant \frac{n}{4} \cdot \frac{n}{4} \cdot \frac{n}{2} = \frac{n^3}{32} = \Omega(n^3)$$

- If n is not a multiple of 4: Clearly f is an increasing function so

$$f(n) \geqslant f(4\lfloor n/4 \rfloor) \geqslant \frac{(4\lfloor n/4 \rfloor)^3}{32} \geqslant \frac{(n-4)^3}{32} = \Omega(n^3).$$

Computing an Optimal Solution

- We only computed the optimal *cost* of a BST.
- We did not show how to compute the optimal BST itself.
- As usual, it can be done quickly using the tables that we computed. (Left as an exercise.)
- Next slide shows a modification of the pseudocode that allows to easily recover an optimal BST by constructing an auxiliary table.
- The reconstruction procedure is left as an exercise.

Computing an Optimal Solution

Pseudocode

```
1:  procedure OptimalCost2(p, q, n, w)
2:      c[1...n+1, 0...n] ← new array
3:      r[1...n, 1...n] ← new array
4:      for i ← 1, n+1 do
5:          c[i, i−1] ← 0
6:      for ℓ ← 1, n do
7:          for i ← 1, n−ℓ+1 do
8:              j ← i+ℓ−1
9:              c[i, j] ← ∞
10:             for r ← i, j do
11:                 t ← c[i, r−1] + w[i, r−1] + c[r+1, j] + w[r+1, j]
12:                 if t < c[i, j] then
13:                     c[i, j] ← t
14:                     r[i, j] ← r
15:     return arrays c and r
```

Concluding Remarks

- The brute force approach takes exponential time, and dynamic programming takes cubic time.
- So once again dynamic programming improved the running time from exponential to polynomial.
- A running time of $\Theta(n^3)$ suggests that we can solve the problem up to roughly $n = 1000$. The brute force approach probably fails for n less than 50, as we saw in the case of binomial coefficients.
- Difference with the textbook: The textbook uses

$$e[i, j] = c[i, j] + w[i, j].$$

in the dynamic programming approach.