CSE515 Advanced Algorithms
Lecture 24: Random Binary Search Trees

Antoine Vigneron
antoine@unist.ac.kr

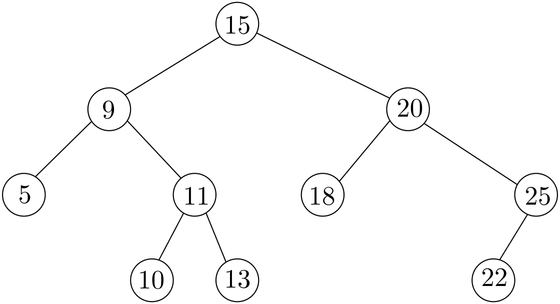
Ulsan National Institute of Science and Technology

May 19, 2021

Introduction

- Assignment 4 due tomorrow.
- In this lecture, I present a *random* binary search trees (BST).
- No particular reference, but you can look at Lecture 5 (review of graph algorithms and data structures).

Binary Search Trees

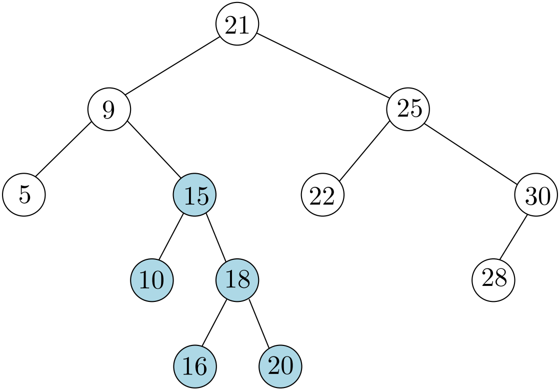


Definition (Binary search tree)

A *binary search tree (BST)* T is a rooted binary tree that records a key at each node. 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 $\{5, 9, 10, 15, 16, 18, 20, 21, 22, 25, 28, 30\}$.

Insertion into a BST

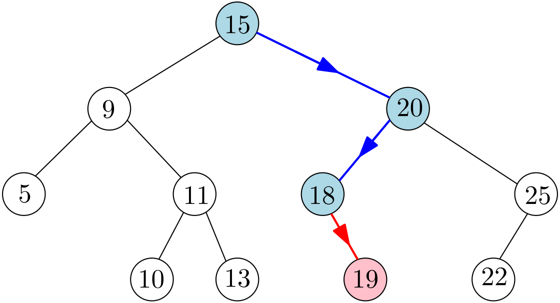
Inserting key k into a BST

procedure $\textsc{Insert}(r, k)$
    if $r = \text{NIL}$ **then**
        $r \leftarrow \textsc{NewNode}(k)$
    else if $k < \text{key}(v)$ **then**
        $\textsc{Insert}(\text{left}(r), k)$
    else
        $\textsc{Insert}(\text{right}(r), k)$

- 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.

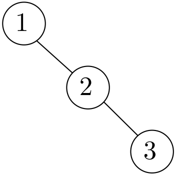
BST Insertion: Example

- Inserting 19 into the tree from Slide 3

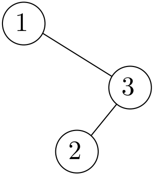


BST Insertion Orders

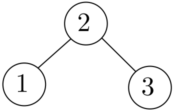
- The shape of a BST depends on the order of insertions.



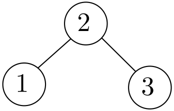
$1 \to 2 \to 3$



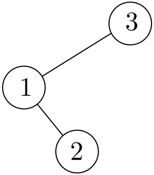
$1 \to 3 \to 2$



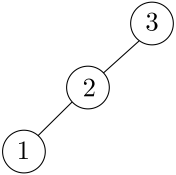
$2 \to 1 \to 3$



$2 \to 3 \to 1$



$3 \to 1 \to 2$



$3 \to 2 \to 1$

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

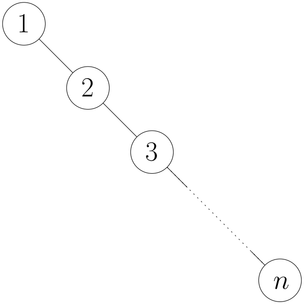
```
procedure SEARCH(v, k)
    if v = NIL then
        return NOTFOUND
    if k < key(v) then
        return SEARCH(left(v), k)
    if k > key(v) then
        return SEARCH(right(v), k)
    return v                                  ▷ k = key(v)
```

Binary Search Trees

- A BST with n nodes has height at least $\lfloor \log n \rfloor$, so the worst case search time is at least $\Omega(\log n)$.

  (Remark: log means $\log_2$ as usual in this course.)

- But the height could be n in the worst case:



- So the search time is $\Omega(n)$ in the worst case.

Random Permutation

Problem

Given an array $A[1 \dots n]$ of n elements, compute a permutation of $A[1 \dots n]$ chosen uniformly at random.

Example

Suppose $A = [a, b, c]$, then the algorithm outputs each of $[a, b, c]$, $[a, c, b]$, $[b, a, c]$, $[b, c, a]$, $[c, a, b]$, $[c, b, a]$ with probability $1/6$.

Permutation by Sorting

Pseudocode

procedure $\textsc{PermuteBySorting}(A[1 \dots n])$
  $P[1 \dots n] \leftarrow$ new array
  for $i \leftarrow 1, n$ **do**
    $P[i] \leftarrow \text{random}(1, n^3)$ ▷ random number in $\{1, 2, \dots, n^3\}$
  Sort A using $P[i]$ as the key of $A[i]$ for all i

- Problem: if two keys are equal, it fails, i.e. the permutation is not chosen uniformly at random.
- The random number is chosen in $\{1, \dots, n^3\}$ to ensure that it happens with probability $\leqslant 1/n$. (left as an exercise).
- In practice just generate a random floating-point number in $[0, 1)$.

Computing a Random Permutation

- Remark: This method can be used with standard spreadsheet programs: generate a column of random numbers and sort according to it.

Theorem

If all keys are distinct, then PERMUTEBYSORTING *produces a uniform random permutation.*

- Proof in textbook (Lemma 5.4 p. 125), not covered.
- Other problems:
  - This algorithm is not *in place* because it uses the auxiliary array $P[1 \dots n]$.
  - It runs in $\Theta(n \log n)$ time if we sort using MERGE SORT.
- How to fix it?

In-Place Computation of a Random Permutation

Pseudocode

```
procedure RandomizeInPlace(A[1...n])
    for i ← 1, n − 1 do
        Exchange A[i] with A[random(i, n)]
                                        ▷ random number in {i, ..., n}
```

Theorem

RandomizeInPlace *computes a uniform random permutation.*

- Proof done in class, see MIT textbook p. 126.

Random BSTs

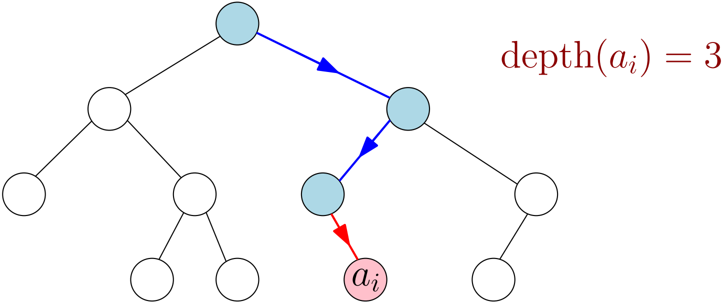
- How can we make the search time logarithmic?
- One approach is to insert the nodes *in a random order*.
- Then we will show that the expected search time is $O(\log n)$.

Pseudocode

```
procedure ConstructRandomBST(A[1...n])
    A ← random permutation of A
    T ← empty BST
    for i ← 1, n do
        insert A[i] into T
    return T
```

Random BSTs

- How much time does it take to construct a random BST?
- Suppose that the insertion procedure takes time c, ignoring recursive calls.



- Then the time taken to insert a_i is $c(1 + \mathrm{depth}(a_i))$, where $\mathrm{depth}(a_i)$ is the length of a path from the root to a_i.

Random BSTs

Definition

The *total path length* of a rooted tree T is the sum of the depths of its nodes

$$P(T) = \sum_{v \in T} \mathrm{depth}(v).$$

- For instance, the tree in previous slide has total path length 22.

Corollary

$\textsc{ConstructRandomBST}$ *constructs a randon binary search tree* $T \neq \emptyset$ *in time* $O(P(T))$.

Random BSTs

Theorem

The expected path length $E[P(T)]$ of a random BST is $O(n \log n)$.

- Proof done in class. See lecture notes.

Consequences

- The expected construction time of a random BST is $O(n \log n)$.
- The expected time to insert one node is $O(\log n)$.
- The expected search time is $O(\log n)$.

Concluding Remarks

- There exist deterministic *balanced binary search trees* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$ in the worst case, and such that is also possible to insert and delete nodes in $\Theta(\log n)$ worst-case time.
  - 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 CSE515, but you should know that they exist. (Covered in CSE221 Data structures.)

Concluding Remarks

- The average construction time and search time for random BST are $O(n \log n)$ and $O(\log n)$, which is the same as balanced BSTs.
- Random BSTs are much simpler than balanced BSTs.
- On the other hand, if we allow the user to insert and delete nodes, the worst-case insertion and deletion time can be $\Omega(n)$ in the worst case. Reason: the tree is not rebalanced, so a long path can be created if we insert nodes by increasing values of their keys, for instance.