Random Binary Search Trees

CS 214, Fall 2019
The necessity of balance
The necessity of balance

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\lceil \log_2 n \rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>
DSSL2 data definition

# An rndbst? (randomized BST of numbers) is either:
# - None
# - _node(key?, nat?, rndbst?, rndbst?)
let rndbst? = OrC(_node?, NoneC)

struct _node:
  let key: key?
  let size: nat?
  let left: rndbst?
  let right: rndbst?
Size maintenance

def empty?(t: rndbst?) -> bool?:
    not _node?(t)

def size(t: rndbst?) -> nat?:
    t.size if _node?(t) else 0

def _fix_size(n: _node?) -> _node?:
    n.size = 1 + size(n.left) + size(n.right)
    n

def _new_node(k: key?) -> rndbst?:
    _node(k, 1, None, None)
Leaf insertion in DSSL2

The easy way to add elements to a tree—at the leaves:

```python
def leaf_insert(t: rndbst?, k: key?) -> rndbst?:
    if empty?(t):
        return _new_node(k)
    elif k < t.key:
        t.left = leaf_insert(t.left, k)
        return _fix_size(t)
    elif k > t.key:
        t.right = leaf_insert(t.right, k)
        return _fix_size(t)
    else:
        return t
```
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
Leaf insertion
The permutation distribution

Can we characterize how sequences of insertions produce (un)balanced trees?
The permutation distribution

Can we characterize how sequences of insertions produce (un)balanced trees?

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 — severely unbalanced (degenerate)

In fact, the only sequence to produce the right-branching degenerate tree is 0, …, 14

There are 21,964,800 sequences that produce the same perfectly balanced tree
Can we characterize how sequences of insertions produce (un)balanced trees?

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 — severely unbalanced (degenerate)
- 7, 3, 1, 0, 2, 5, 4, 6, 11, 9, 8, 10, 13, 12, 14 — balanced
The permutation distribution

Can we characterize how sequences of insertions produce (un)balanced trees?

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 — severely unbalanced (degenerate)
- 7, 3, 1, 0, 2, 5, 4, 6, 11, 9, 8, 10, 13, 12, 14 — balanced
- 7, 11, 3, 13, 9, 5, 1, 14, 12, 10, 8, 6, 4, 2, 0 — balanced
The permutation distribution

Can we characterize how sequences of insertions produce (un)balanced trees?

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 — severely unbalanced (degenerate)
- 7, 3, 1, 0, 2, 5, 4, 6, 11, 9, 8, 10, 13, 12, 14 — balanced
- 7, 11, 3, 13, 9, 5, 1, 14, 12, 10, 8, 6, 4, 2, 0 — balanced

In fact, the only sequence to produce the right-branching degenerate tree is 0, ..., 14

There are 21,964,800 sequences that produce the same perfectly balanced tree
A random BST tends to be balanced

If you generate a tree by leaf-inserting a random permutation of its elements, it will probably be balanced

In particular, the expected length of a search path is

\[ 2 \ln n + \mathcal{O}(1) \]
A random BST tends to be balanced

If you generate a tree by leaf-inserting a random permutation of its elements, it will probably be balanced.

In particular, the expected length of a search path is

\[ 2 \ln n + \mathcal{O}(1) \]

Unfortunately, we usually can’t do that, but we can simulate it.
A tool: tree rotations

Note that order is preserved
In DSSL2

def _rotate_right(d):
    let b = d.left
    d.left = b.right
    b.right = d
    return b

def _rotate_left(b):
    let d = b.right
    b.right = d.left
    d.left = b
    return d
In DSSL2

```python
def _rotate_right(d):
    let b = d.left
d.left = b.right
b.right = _fix_size(d)
return _fix_size(b)

def _rotate_left(b):
    let d = b.right
b.right = d.left
d.left = _fix_size(b)
return _fix_size(d)
```
Another random little helper

```python
def _k_in_n(k: nat?, n: nat?) -> bool?:
    return random(n) < k
```

def _k_in_n(k: nat?, n: nat?) -> bool?:
    return random(n) < k

Pr\left[ _k\text{\_in\_n}(k, n) \right] = k/n
Another random little helper

def _k_in_n(k: nat?, n: nat?) -> bool?:
    return random(n) < k

\[ Pr\left[ _\k_in_n(k, n) \right] = \frac{k}{n} \]

def _one_in_n(n: nat?) -> bool?:
    return random(n) == 0

\[ Pr\left[ _\text{one_in_n}(n) \right] = \frac{1}{n} \]
Root insertion

Using rotations, we can insert at the root:

- To insert into an empty tree, create a new node
- To insert into a non-empty tree, if the new key is greater than the root, then root-insert (recursively) into the right subtree, then rotate left
- By symmetry, if the key belongs to the left of the old root, root insert into the left subtree and then rotate right
def _root_insert(t: rndbst?, k: key?) -> rndbst?:
    if empty?(t):
        return _new_node(k)
    elif k < t.key:
        t.left = _root_insert(t.left, k)
        return _rotate_right(t)
    elif k > t.key:
        t.right = _root_insert(t.right, k)
        return _rotate_left(t)
    else:
        return t
Randomized insertion

We can now build a randomized insertion function that maintains the random shape of the tree:

- Suppose we insert into a subtree of size $k$, so the result will have size $k + 1$
- If the tree were random, the new element would be the root with probability $\frac{1}{k+1}$
- So we root insert with that probability, and otherwise recursively insert into a subsubtree
Randomized insertion in DSSL2

def insert(t: rndbst?, k: key?) -> rndbst?:
    if _one_in_n(size(t) + 1):
        return _root_insert(t, k)
    elif k < t.key:
        t.left = insert(t.left, k)
        return _fix_size(t)
    elif k > t.key:
        t.right = insert(t.right, k)
        return _fix_size(t)
    else:
        return t
Deletion idea

To delete a node, we join its subtrees recursively, randomly selecting which contributes the root (based on size):

```
B
/   \
/     \
A     C

+ E ⇒

B
/   \
/     \
A     C+E
```
def _join(t1: rndbst?, t2: rndbst?) -> rndbst?:
    if empty?(t1): return t2
    elif empty?(t2): return t1
    elif _k_in_n(size(t1), size(t1) + size(t2)):
        t1.right = _join(t1.right, t2)
        return _fix_size(t1)
    else:
        t2.left = _join(t1, t2.left)
        return _fix_size(t2)
def delete(t: rndbst?, k: key?) -> rndbst?:
    if empty?(t):
        return t
    elif k < t.key:
        t.left = delete(t.left, k)
        return _fix_size(t)
    elif k > t.key:
        t.right = delete(t.right, k)
        return _fix_size(t)
    else:
        return _join(t.left, t.right)
Next: guaranteed balance