Randomized Algorithms, Lecture 6

Jacob Holm (jaho@di.ku.dk)

May 13th 2019

Good Morning.

Today's Lecture

Data structures

- The data-structuring problem
- Random Treaps
- Hashing fundamentals
- Hashing with chaining
- Two-level hashing

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

MAKESET(S) Create a new empty set S.

INSERT(i, S) Insert item i with key $k(i)$ into S.

DELETE(k, S) Delete item with key k from S.

FIND(k, S) Get item with key k in S.

JOIN(S_1, i, S_2) Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

CONCAT(S_1, S_2) Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

SPLIT(k, S) Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and $S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

MAKESET(S) Create a new empty set S.

INSERT(i, S) Insert item i with key $k(i)$ into S.

DELETE(k, S) Delete item with key k from S.

FIND(k, S) Get item with key k in S.

JOIN(S_1, i, S_2) Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

CONCAT(S_1, S_2) Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

SPLIT(k, S) Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and $S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

$\textsc{makeset}(S)$ Create a new empty set S.

$\textsc{insert}(i, S)$ Insert item i with key $k(i)$ into S.

$\textsc{delete}(k, S)$ Delete item with key k from S.

$\textsc{find}(k, S)$ Get item with key k in S.

$\textsc{join}(S_1, i, S_2)$ Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

$\textsc{concat}(S_1, S_2)$ Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

$\textsc{split}(k, S)$ Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and $S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

MAKESET(S) Create a new empty set S.

INSERT(i, S) Insert item i with key $k(i)$ into S.

DELETE(k, S) Delete item with key k from S.

FIND(k, S) Get item with key k in S.

JOIN(S_1, i, S_2) Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

CONCAT(S_1, S_2) Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

SPLIT(k, S) Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and
$S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

$\textsc{makeset}(S)$ Create a new empty set S.

$\textsc{insert}(i, S)$ Insert item i with key $k(i)$ into S.

$\textsc{delete}(k, S)$ Delete item with key k from S.

$\textsc{find}(k, S)$ Get item with key k in S.

$\textsc{join}(S_1, i, S_2)$ Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

$\textsc{concat}(S_1, S_2)$ Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

$\textsc{split}(k, S)$ Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and $S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

$\textsc{MakeSet}(S)$ Create a new empty set S.

$\textsc{Insert}(i, S)$ Insert item i with key $k(i)$ into S.

$\textsc{Delete}(k, S)$ Delete item with key k from S.

$\textsc{Find}(k, S)$ Get item with key k in S.

$\textsc{Join}(S_1, i, S_2)$ Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

$\textsc{Concat}(S_1, S_2)$ Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

$\textsc{Split}(k, S)$ Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and
$S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

$\textsc{MakeSet}(S)$ Create a new empty set S.

$\textsc{Insert}(i, S)$ Insert item i with key $k(i)$ into S.

$\textsc{Delete}(k, S)$ Delete item with key k from S.

$\textsc{Find}(k, S)$ Get item with key k in S.

$\textsc{Join}(S_1, i, S_2)$ Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

$\textsc{Concat}(S_1, S_2)$ Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

$\textsc{Split}(k, S)$ Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and
$S_2 = \{j \in S \mid k(j) > k\}$.

The data-structuring problem

Maintain disjoint sets $S_1, S_2, \ldots$ of *items* with keys from totally ordered universe $\mathcal{U}$ under

$\textsc{makeset}(S)$ Create a new empty set S.

$\textsc{insert}(i, S)$ Insert item i with key $k(i)$ into S.

$\textsc{delete}(k, S)$ Delete item with key k from S.

$\textsc{find}(k, S)$ Get item with key k in S.

$\textsc{join}(S_1, i, S_2)$ Given $k(i_1) < k(i) < k(i_2)$ for $i_1 \in S_1, i_2 \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup \{i\} \cup S_2$.

$\textsc{concat}(S_1, S_2)$ Given $k(i) < k(j)$ for $i \in S_1, j \in S_2$.
Replace S_1 and S_2 with $S = S_1 \cup S_2$.

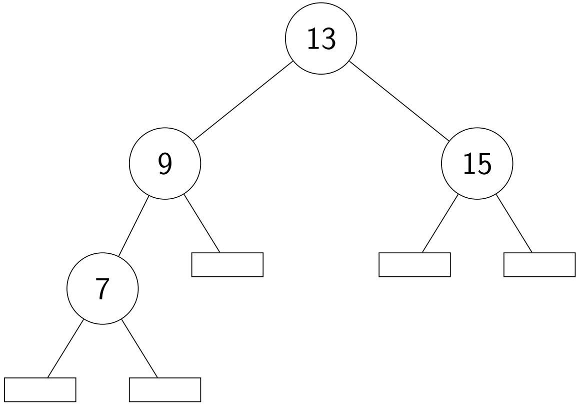
$\textsc{split}(k, S)$ Replace S with $S_1 = \{j \in S \mid k(j) < k\}$ and
$S_2 = \{j \in S \mid k(j) > k\}$.

Binary Search Tree

Suppose we assign a unique *key* $k(i)$ from some totally ordered set to each item of a set S.

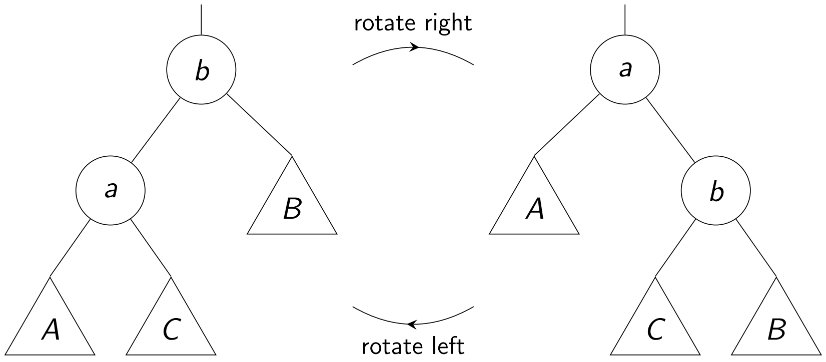
A binary tree whose inner nodes store the items of S is in *symmetric order* and is called a *binary search tree* if, for each node with key k, the left subtree contains $\{j \in S \mid k(j) < k\}$ and the right subtree contains $\{j \in S \mid k(j) > k\}$.

Binary Search Tree, Example



We assume that we only store items in internal nodes.
Leaves are marked on this slide using rectangles, but are left out of the drawing in the following.

Binary Search Tree, Rotations



Binary Search Tree, Algo

function MAKESET(r)
  Initialize r as a leaf.
function FIND(k, r)
  if $k(r.\text{item}) < k$ **then**
    return FIND($k, r.\text{left}$)
  else if $k(r.\text{item}) > k$ **then**
    return FIND($k, r.\text{right}$)
  else
    return r ▷ If $k \notin S_r$ this is a leaf and has no item.
function INSERT(i, r)
  $v \leftarrow$ FIND($k(i), r$)
  Replace leaf v with internal node i having 2 leaf children.

Binary Search Tree, Algo

function $\text{MAKESET}(r)$
  Initialize r as a leaf.
function $\text{FIND}(k, r)$
  if $k(r.\text{item}) < k$ **then**
    return $\text{FIND}(k, r.\text{left})$
  else if $k(r.\text{item}) > k$ **then**
    return $\text{FIND}(k, r.\text{right})$
  else
    return r $\triangleright$ If $k \notin S_r$ this is a leaf and has no item.
function $\text{INSERT}(i, r)$
  $v \leftarrow \text{FIND}(k(i), r)$
  Replace leaf v with internal node i having 2 leaf children.

Binary Search Tree, Algo

function MAKESET(r)
  Initialize r as a leaf.
function FIND(k, r)
  if $k(r.\text{item}) < k$ **then**
    return FIND($k, r.\text{left}$)
  else if $k(r.\text{item}) > k$ **then**
    return FIND($k, r.\text{right}$)
  else
    return r ▷ If $k \notin S_r$ this is a leaf and has no item.
function INSERT(i, r)
  $v \leftarrow$ FIND($k(i), r$)
  Replace leaf v with internal node i having 2 leaf children.

Binary Search Tree, Algo

```
function JOIN(r1, i, r2)
    r ← new internal node having children r1 and r2.
    r.item ← i
    return r
function DELETE(k, r)
    v ← FIND(k, r)
    if  v.left is a leaf  then
        Let v.right replace v as child of v.parent
    else if  v.right is a leaf  then
        Let v.left replace v as child of v.parent
    else
        v′ ← FIND(∞, v.left).parent
        i ← v′.item
        DELETE(k(i), v.left)          ▷ v′ has a leaf as right child
        v.item ← i
```

Binary Search Tree, Algo

function JOIN(r_1, i, r_2)
  $r \leftarrow$ new internal node having children r_1 and r_2.
  r.item $\leftarrow i$
  return r
function DELETE(k, r)
  $v \leftarrow$ FIND(k, r)
  if v.left is a leaf **then**
    Let v.right replace v as child of v.parent
  else if v.right is a leaf **then**
    Let v.left replace v as child of v.parent
  else
    $v' \leftarrow$ FIND(∞, v.left).parent
    $i \leftarrow v'$.item
    DELETE($k(i)$, v.left) $\triangleright$ v' has a leaf as right child
    v.item $\leftarrow i$

Binary Search Tree, Algo

function CONCAT(r_1, r_2)
  $i \leftarrow$ FIND(∞, r_1).parent.item
  DELETE($k(i), r_1$)
  return JOIN(r_1, i, r_2)

function SPLIT(k, r)
  Use rotations to make k the root key.
  return (r.left, r.right).

Binary Search Tree, Algo

function $\text{CONCAT}(r_1, r_2)$
  $i \leftarrow \text{FIND}(\infty, r_1)$.parent.item
  $\text{DELETE}(k(i), r_1)$
  return $\text{JOIN}(r_1, i, r_2)$

function $\text{SPLIT}(k, r)$
  Use rotations to make k the root key.
  return (r.left, r.right).

Binary Search Tree, Balancing

A number of strategies exist that use rotations to keep the BST *balanced*.

- AVL-trees (1962)
- Red/Black trees (1978)
- Splay trees (1985)

AVL-trees explicitly maintain the height of each subtree and ensures the height difference between siblings is at most one. Simple to implement, but uses $\mathcal{O}(\log n)$ rotations per insert/delete in the worst case.

Red/black trees store a single bit of balancing info per node, and use at most 2 (3) rotations per insertion (deletion). They are complicated to implement but good in practice. Time is still worst case $\mathcal{O}(\log n)$ per operation.

Splay trees use *amortized* $\mathcal{O}(\log n)$ rotations per *operation* (including queries). They do not explicitly maintain balance, but are *self-adjusting* in the sense that frequently accessed items will move closer to the root. They are in fact conjectured to be optimal up to a constant factor.

Binary Search Tree, Balancing

A number of strategies exist that use rotations to keep the BST *balanced*.

- AVL-trees (1962)
- Red/Black trees (1978)
- Splay trees (1985)

AVL-trees explicitly maintain the height of each subtree and ensures the height difference between siblings is at most one. Simple to implement, but uses $\mathcal{O}(\log n)$ rotations per insert/delete in the worst case.

Red/black trees store a single bit of balancing info per node, and use at most 2 (3) rotations per insertion (deletion). They are complicated to implement but good in practice. Time is still worst case $\mathcal{O}(\log n)$ per operation.

Splay trees use *amortized* $\mathcal{O}(\log n)$ rotations per *operation* (including queries). They do not explicitly maintain balance, but are *self-adjusting* in the sense that frequently accessed items will move closer to the root. They are in fact conjectured to be optimal up to a constant factor.

Binary Search Tree, Balancing

A number of strategies exist that use rotations to keep the BST *balanced*.

- AVL-trees (1962)
- Red/Black trees (1978)
- Splay trees (1985)

AVL-trees explicitly maintain the height of each subtree and ensures the height difference between siblings is at most one. Simple to implement, but uses $\mathcal{O}(\log n)$ rotations per insert/delete in the worst case.

Red/black trees store a single bit of balancing info per node, and use at most 2 (3) rotations per insertion (deletion). They are complicated to implement but good in practice. Time is still worst case $\mathcal{O}(\log n)$ per operation.

Splay trees use *amortized* $\mathcal{O}(\log n)$ rotations per *operation* (including queries). They do not explicitly maintain balance, but are *self-adjusting* in the sense that frequently accessed items will move closer to the root. They are in fact conjectured to be optimal up to a constant factor.

Binary Search Tree, Balancing

A number of strategies exist that use rotations to keep the BST *balanced*.

- AVL-trees (1962)
- Red/Black trees (1978)
- Splay trees (1985)

AVL-trees explicitly maintain the height of each subtree and ensures the height difference between siblings is at most one. Simple to implement, but uses $\mathcal{O}(\log n)$ rotations per insert/delete in the worst case.

Red/black trees store a single bit of balancing info per node, and use at most 2 (3) rotations per insertion (deletion). They are complicated to implement but good in practice. Time is still worst case $\mathcal{O}(\log n)$ per operation.

Splay trees use *amortized* $\mathcal{O}(\log n)$ rotations per *operation* (including queries). They do not explicitly maintain balance, but are *self-adjusting* in the sense that frequently accessed items will move closer to the root. They are in fact conjectured to be optimal up to a constant factor.

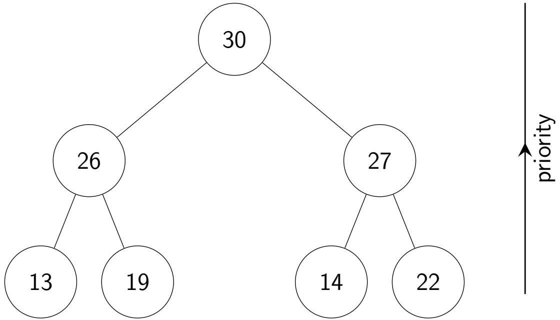
Max-Heap

Suppose we assign a unique *priority* $p(i)$ from some totally ordered set to each item of a set S.

A tree whose inner nodes store the items of S is in *max-heap order* if the priority of each node is less than the priority of its parent.

Max-Heap, Example

$\{13, 26, 19, 30, 14, 27, 22\}$



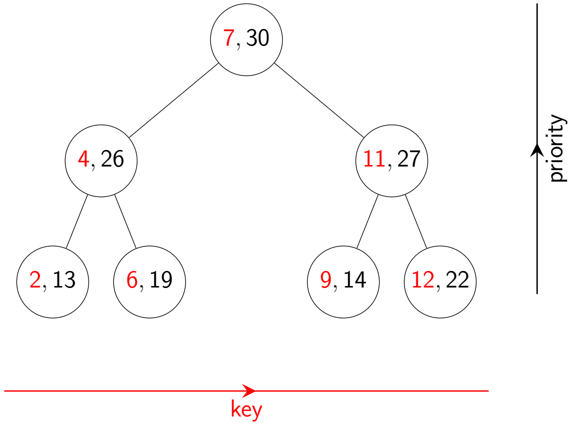
Treap

Suppose we assign both a unique key $k(i)$ and a unique priority $p(i)$ to each item $i \in S$.

A *treap* for S is a binary tree where each node stores an item from S, such that the nodes are in BST order wrt $k(\cdot)$ and in max-heap order wrt $p(\cdot)$.

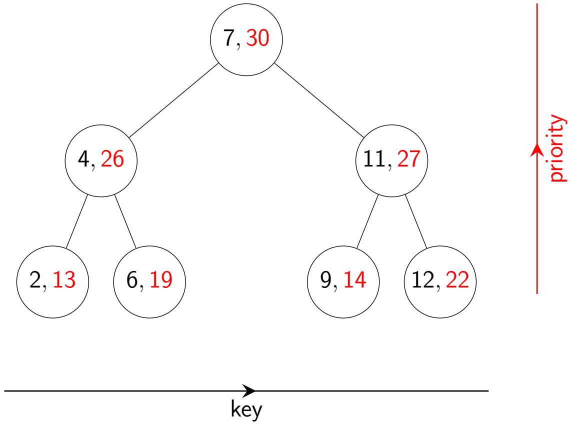
Treap, Example

$$\{(2, 13), (4, 26), (6, 19), (7, 30), (9, 14), (11, 27), (12, 22)\}$$



Treap, Example

$\{(2, 13), (4, 26), (6, 19), (7, 30), (9, 14), (11, 27), (12, 22)\}$



Treap, Uniqueness

Theorem

Let $S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$ *be any set of key-priority pairs such that the keys are distinct and the priorities are distinct. Then there is a unique treap* $T(S)$ *for it.*

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Treap, Uniqueness Proof

By induction on n. Trivial for $n \leq 1$, so let $n > 1$ and suppose it holds for all S' with $|S'| < n$. Let p_i be (unique) highest priority in S. If a treap exists for S, then (k_i, p_i) is the root, $S_1 = \{(k_j, p_j) \mid k_j < k_i\}$ is in the left subtree, and $S_2 = \{(k_j, p_j) \mid k_j > k_i\}$ is in the right subtree. By induction, $T(S_1)$ and $T(S_2)$ exist and are unique, so we can recursively construct the unique $T(S)$ from $T(S_1)$ and $T(S_2)$. $\square$

Random Treap

Idea: We can assign distinct uniformly random priorities on insert and use rotations to maintain the treap invariant.

Theorem

Let $x_{(k)}$ denote the node containing the item with the k'th smallest key. The expected depth of $x_{(k)}$ is $H_k + H_{n-k+1} - 1 \in \mathcal{O}(\log n)$.

We will assume in our analysis that the random priorities are distinct. This can be justified by e.g. using the keys (which are required to be distinct) to disambiguate.

Random Treap

Idea: We can assign distinct uniformly random priorities on insert and use rotations to maintain the treap invariant.

Theorem

Let $x_{(k)}$ denote the node containing the item with the k'th smallest key. The expected depth of $x_{(k)}$ is $H_k + H_{n-k+1} - 1 \in \mathcal{O}(\log n)$.

This means that the time for a successful find is $\mathcal{O}(\log n)$.
Finding the time for an *unsuccessful* find is part of the next assignment.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k+1-i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k+1-i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k+1-i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

Let $x_{(j)}$ be the unique deepest node such that the subtree rooted at $x_{(j)}$ contains both $x_{(i)}$ and $x_{(k)}$. This is also known as the *nearest common ancestor* $x_{(j)} = \text{nca}(x_{(i)}, x_{(k)})$.

By definition, either $j = i$, or $j = k$, or $x_{(i)}$ is in the left subtree and $x_{(k)}$ is in the right subtree.

Since $x_{(j)}$ is the root of this subtree, $p_{(j)} \geq \max\{p_{(i)}, \ldots, p_{(k)}\}$.

Now $X_{ik} = 1$ iff $j = i$ and the result follows.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k+1-i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

All orders equally likely, so each of the $k + 1 - i$ elements in the set are equally likely to be the largest.

Random Treap, Depth Proof

Expectation of indicator variable.

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k+1-i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then for $i < k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{k + 1 - i}$$

$$\mathbb{E}\left[\sum_{i<k} X_{ik}\right] = \sum_{d=2}^{k} \frac{1}{d} = H_k - 1$$

Linearity of expectation and change of variable, setting $d = k + 1 - i$.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Similarly for $i > k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{i + 1 - k}$$

$$\mathbb{E}\left[\sum_{i>k} X_{ik}\right] = \sum_{d=2}^{n+1-k} \frac{1}{d} = H_{n+1-k} - 1$$

Let $x_{(j)}$ be the unique deepest node such that the subtree rooted at $x_{(j)}$ contains both $x_{(k)}$ and $x_{(i)}$. This is also known as the *nearest common ancestor* $x_{(j)} = \text{nca}(x_{(k)}, x_{(i)})$.

By definition, either $j = k$, or $j = i$, or $x_{(k)}$ is in the left subtree and $x_{(i)}$ is in the right subtree.

Since $x_{(j)}$ is the root of this subtree, $p_{(j)} \geq \max\{p_{(k)}, \ldots, p_{(i)}\}$.

Now $X_{ik} = 1$ iff $j = i$ and the result follows.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Similarly for $i > k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(k)}, \dots, p_{(i)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{i+1-k}$$

$$\mathbb{E}\left[\sum_{i>k} X_{ik}\right] = \sum_{d=2}^{n+1-k} \frac{1}{d} = H_{n+1-k} - 1$$

All orders equally likely, so each of the $i + 1 - k$ elements in the set are equally likely to be the largest.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Similarly for $i > k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{i + 1 - k}$$

$$\mathbb{E}\left[\sum_{i>k} X_{ik}\right] = \sum_{d=2}^{n+1-k} \frac{1}{d} = H_{n+1-k} - 1$$

Expectation of indicator variable.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Similarly for $i > k$

$$X_{ik} = 1 \iff p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}$$

$$\mathbb{E}[X_{ik}] = Pr[X_{ik} = 1] = \frac{1}{i + 1 - k}$$

$$\mathbb{E}\left[\sum_{i>k} X_{ik}\right] = \sum_{d=2}^{n+1-k} \frac{1}{d} = H_{n+1-k} - 1$$

Linearity of expectation and change of variable, setting $d = i + 1 - k$.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then

$$\begin{aligned}
\mathbb{E}[\mathrm{depth}(x_{(k)})] &= \mathbb{E}\left[\sum_{i=1}^{n} X_{ik}\right] \\
&= \mathbb{E}\left[\sum_{i<k} X_{ik}\right] + \mathbb{E}\left[\sum_{i>k} X_{ik}\right] + \mathbb{E}[X_{kk}] \\
&= (H_k - 1) + (H_{n+1-k} - 1) + 1 \\
&= H_k + H_{n+1-k} - 1 \qquad \square
\end{aligned}$$

The depth of a node is defined as the number of ancestors, including the node itself.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then

$$\begin{aligned}
\mathbb{E}[\text{depth}(x_{(k)})] &= \mathbb{E}\left[\sum_{i=1}^{n} X_{ik}\right] \\
&= \mathbb{E}\left[\sum_{i<k} X_{ik}\right] + \mathbb{E}\left[\sum_{i>k} X_{ik}\right] + \mathbb{E}[X_{kk}] \\
&= (H_k - 1) + (H_{n+1-k} - 1) + 1 \\
&= H_k + H_{n+1-k} - 1 \qquad \square
\end{aligned}$$

Linearity of expectation.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then

$$\begin{aligned}
\mathbb{E}[\mathrm{depth}(x_{(k)})] &= \mathbb{E}\left[\sum_{i=1}^{n} X_{ik}\right] \\
&= \mathbb{E}\left[\sum_{i<k} X_{ik}\right] + \mathbb{E}\left[\sum_{i>k} X_{ik}\right] + \mathbb{E}[X_{kk}] \\
&= (H_k - 1) + (H_{n+1-k} - 1) + 1 \\
&= H_k + H_{n+1-k} - 1 \qquad \square
\end{aligned}$$

Previous slides.

Random Treap, Depth Proof

Let $p_{(i)}$ denote the priority of $x_{(i)}$, and let X_{ik} indicate that $x_{(i)}$ is ancestor to $x_{(k)}$.

Then

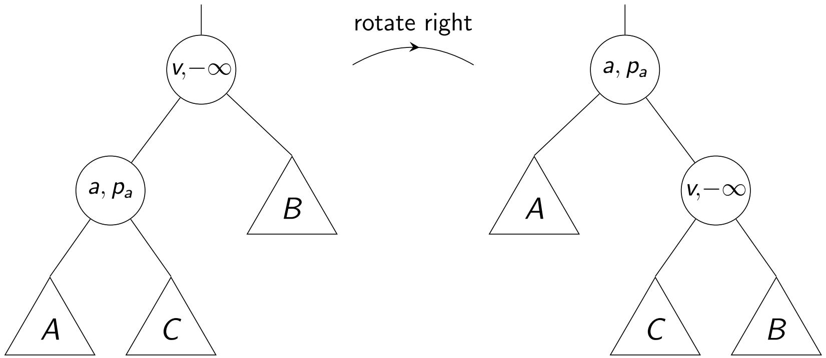
$$\begin{aligned}
\mathbb{E}[\text{depth}(x_{(k)})] &= \mathbb{E}\left[\sum_{i=1}^{n} X_{ik}\right] \\
&= \mathbb{E}\left[\sum_{i<k} X_{ik}\right] + \mathbb{E}\left[\sum_{i>k} X_{ik}\right] + \mathbb{E}[X_{kk}] \\
&= (H_k - 1) + (H_{n+1-k} - 1) + 1 \\
&= H_k + H_{n+1-k} - 1 \qquad \square
\end{aligned}$$

Random Treap, Deletion

We can implement DELETE(k, r) on random treaps as

function RT-DELETE(k, r)
    $v \leftarrow$ FIND(k, r)
    v.priority $\leftarrow -\infty$
    while v has a non-leaf child **do**
        if v.left.priority $>$ v.right.priority **then**
            ROTATE-RIGHT(v)
        else
            ROTATE-LEFT(v)
    Set v.item $\leftarrow$ **nil** and delete v.left and v.right

Random Treap, Deletion



Note that this works, because 1) Setting the priority to $-\infty$ only violates the max-heap invariant for the children of that node, and 2) each rotation moves the violation to a smaller subtree.

Also note that each rotation we do adds exactly one new ancestor to v.

Random Treap, Deletion

Theorem

The expected number of rotations made by each call to RT-DELETE *is less than* 2.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$\begin{aligned} Y_{ik} = 1 &\iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k-1)}, -\infty\} \\ &\qquad \wedge \neg(p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}) \\ &\iff p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \dots, p_{(k-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1} (\tfrac{1}{d} - \tfrac{1}{d+1}) = 1 - \tfrac{1}{k}$$

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$\begin{aligned} Y_{ik} = 1 \iff & p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k-1)}, -\infty\} \\ & \wedge \neg(p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}) \\ \iff & p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \ldots, p_{(k-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1}(\tfrac{1}{d} - \tfrac{1}{d+1}) = 1 - \tfrac{1}{k}$$

For $x_{(i)}$ to become a new ancestor it has to be an ancestor after the RT-DELETE, but not before.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$
\begin{aligned}
Y_{ik} = 1 \iff & p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k-1)}, -\infty\} \\
& \wedge \neg(p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}) \\
\iff & p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \ldots, p_{(k-1)}\}
\end{aligned}
$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1} \left(\tfrac{1}{d} - \tfrac{1}{d+1}\right) = 1 - \tfrac{1}{k}$$

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$\begin{aligned} Y_{ik} = 1 &\iff p_{(i)} = \max\{p_{(i)}, \dots, p_{(k-1)}, -\infty\} \\ &\qquad \wedge \neg(p_{(i)} = \max\{p_{(i)}, \dots, p_{(k)}\}) \\ &\iff p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \dots, p_{(k-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1}\big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \tfrac{1}{k}$$

There are $(k-i+1)(k-i)$ ways to choose a largest and second largest element from the set $\{p_{(i)}, \dots, p_{(k)}\}$.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$\begin{aligned} Y_{ik} = 1 \iff & p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k-1)}, -\infty\} \\ & \wedge \neg(p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}) \\ \iff & p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \ldots, p_{(k-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1}\big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \tfrac{1}{k}$$

A very useful formula:

$$\frac{1}{n} - \frac{1}{n+1} = \frac{n+1}{(n+1)n} - \frac{n}{(n+1)n} = \frac{(n+1)-n}{(n+1)n} = \frac{1}{(n+1)n}$$

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Then for $i < k$

$$\begin{aligned} Y_{ik} = 1 &\iff p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k-1)}, -\infty\} \\ &\qquad \wedge \neg(p_{(i)} = \max\{p_{(i)}, \ldots, p_{(k)}\}) \\ &\iff p_{(k)} > p_{(i)} > \max\{p_{(i+1)}, \ldots, p_{(k-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(k-i+1)(k-i)} = \tfrac{1}{k-i} - \tfrac{1}{k-i+1}$$

$$\mathbb{E}\Big[\sum_{i=1}^{k-1} Y_{ik}\Big] = \sum_{d=1}^{k-1} \big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \tfrac{1}{k}$$

Linearity of expectation, and a change of variable setting $d = k - i$. Finally a telescoping sum.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Similarly for $i > k$

$$Y_{ik} = 1 \iff p_{(i)} = \max\{-\infty, p_{(k+1)}, \dots, p_{(i)}\} \wedge \neg(p_{(i)} = \max\{p_{(k)}, \dots, p_{(i)}\})$$
$$\iff p_{(k)} > p_{(i)} > \max\{p_{(k+1)}, \dots, p_{(i-1)}\}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \frac{1}{(i-k+1)(i-k)} = \frac{1}{i-k} - \frac{1}{i-k+1}$$
$$\mathbb{E}\Big[\sum_{i=k+1}^{n} Y_{ik}\Big] = \sum_{d=1}^{n-k}\big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \frac{1}{n-k+1}$$

For $x_{(i)}$ to become a new ancestor it has to be an ancestor after the RT-DELETE, but not before.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Similarly for $i > k$

$$\begin{aligned} Y_{ik} = 1 &\iff p_{(i)} = \max\{-\infty, p_{(k+1)}, \ldots, p_{(i)}\} \\ &\qquad \wedge \neg(p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}) \\ &\iff p_{(k)} > p_{(i)} > \max\{p_{(k+1)}, \ldots, p_{(i-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(i-k+1)(i-k)} = \tfrac{1}{i-k} - \tfrac{1}{i-k+1}$$

$$\mathbb{E}\Big[\sum_{i=k+1}^{n} Y_{ik}\Big] = \sum_{d=1}^{n-k}\big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \tfrac{1}{n-k+1}$$

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Similarly for $i > k$

$$\begin{aligned} Y_{ik} = 1 \iff & p_{(i)} = \max\{-\infty, p_{(k+1)}, \dots, p_{(i)}\} \\ & \wedge \neg(p_{(i)} = \max\{p_{(k)}, \dots, p_{(i)}\}) \\ \iff & p_{(k)} > p_{(i)} > \max\{p_{(k+1)}, \dots, p_{(i-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(i-k+1)(i-k)} = \tfrac{1}{i-k} - \tfrac{1}{i-k+1}$$

$$\mathbb{E}\Big[\sum_{i=k+1}^{n} Y_{ik}\Big] = \sum_{d=1}^{n-k}\big(\tfrac{1}{d} - \tfrac{1}{d+1}\big) = 1 - \tfrac{1}{n-k+1}$$

There are $(i-k+1)(i-k)$ ways to choose a largest and second largest element from the set $\{p_{(k)}, \dots, p_{(i)}\}$.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Similarly for $i > k$

$$\begin{aligned} Y_{ik} = 1 \iff & p_{(i)} = \max\{-\infty, p_{(k+1)}, \ldots, p_{(i)}\} \\ & \wedge \neg(p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}) \\ \iff & p_{(k)} > p_{(i)} > \max\{p_{(k+1)}, \ldots, p_{(i-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \frac{1}{(i-k+1)(i-k)} = \frac{1}{i-k} - \frac{1}{i-k+1}$$

$$\mathbb{E}\Big[\sum_{i=k+1}^{n} Y_{ik}\Big] = \sum_{d=1}^{n-k}\Big(\frac{1}{d} - \frac{1}{d+1}\Big) = 1 - \frac{1}{n-k+1}$$

A very useful formula:

$$\frac{1}{n} - \frac{1}{n+1} = \frac{n+1}{(n+1)n} - \frac{n}{(n+1)n} = \frac{(n+1)-n}{(n+1)n} = \frac{1}{(n+1)n}$$

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Similarly for $i > k$

$$\begin{aligned} Y_{ik} = 1 &\iff p_{(i)} = \max\{-\infty, p_{(k+1)}, \ldots, p_{(i)}\} \\ &\qquad \wedge \neg(p_{(i)} = \max\{p_{(k)}, \ldots, p_{(i)}\}) \\ &\iff p_{(k)} > p_{(i)} > \max\{p_{(k+1)}, \ldots, p_{(i-1)}\} \end{aligned}$$

$$\mathbb{E}[Y_{ik}] = \Pr[Y_{ik} = 1] = \tfrac{1}{(i-k+1)(i-k)} = \tfrac{1}{i-k} - \tfrac{1}{i-k+1}$$

$$\mathbb{E}\Big[\sum_{i=k+1}^{n} Y_{ik}\Big] = \sum_{d=1}^{n-k} (\tfrac{1}{d} - \tfrac{1}{d+1}) = 1 - \tfrac{1}{n-k+1}$$

Linearity of expectation, and a change of variable setting $d = i - k$. Finally a telescoping sum.

Random Treap, Deletion Proof

Let $v = x_{(k)}$, and let Y_{ik} indicate that $x_{(i)}$ becomes a new ancestor of $x_{(k)}$.

Finally

$$
\begin{aligned}
\mathbb{E}[\#\text{rotations}] &= \mathbb{E}[\#\text{new ancestors to } v] \\
&= \mathbb{E}\left[\sum_{i=1}^{n} Y_{ik}\right] \\
&= \mathbb{E}\left[\sum_{i=1}^{k-1} Y_{ik}\right] + \mathbb{E}\left[\sum_{i=k+1}^{n} Y_{ik}\right] + \mathbb{E}[Y_{kk}] \\
&= (1 - \tfrac{1}{k}) + (1 - \tfrac{1}{n+1-k}) + 0 \\
&< 2 \qquad \square
\end{aligned}
$$

Hash function

Given a (large) universe U of keys, and a positive integer m.

Definition

A hash function $h : U \to [m]$ is a random variable, whose values are functions from $U \to [m]$.
Equivalently, for each $x \in U$, $h(x) \in [m]$ is a random variable.

Hash function

Given a (large) universe U of keys, and a positive integer m.

Definition

A hash function $h : U \to [m]$ is a random variable, whose values are functions from $U \to [m]$. Equivalently, for each $x \in U$, $h(x) \in [m]$ is a random variable.

Hash function

When discussing hash functions, we usually care about

1. Space (*seed size*) needed to represent h.
2. Time needed to calculate $h(x)$ given $x \in U$.
3. Properties of the random variable.

Hash function

When discussing hash functions, we usually care about

1. Space (*seed size*) needed to represent h.
2. Time needed to calculate $h(x)$ given $x \in U$.
3. Properties of the random variable.

Hash function

When discussing hash functions, we usually care about

1. Space (*seed size*) needed to represent h.
2. Time needed to calculate $h(x)$ given $x \in U$.
3. Properties of the random variable.

Hash function types

Definition

A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform.

Impractical, why?

Definition

A hash function $h : U \to [m]$ is c-*universal* if, for all $x \neq y \in U$: $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Hash function types

Definition

A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform.

Impractical, why?

Definition

A hash function $h : U \to [m]$ is c-*universal* if, for all $x \neq y \in U$: $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Hash function types

Definition

A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform.

Impractical, why? Space! Require $|U| \log_2 m$ bits to represent.

Definition

A hash function $h : U \to [m]$ is c-*universal* if, for all $x \neq y \in U$: $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Hash function types

Definition

A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform.

Impractical, why? Space! Require $|U| \log_2 m$ bits to represent.

Definition

A hash function $h : U \to [m]$ is c-*universal* if, for all $x \neq y \in U$: $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Hash function types

Definition

A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform.

Impractical, why? Space! Require $|U| \log_2 m$ bits to represent.

Definition

A hash function $h : U \to [m]$ is *c-universal* if, for all $x \neq y \in U$: $\Pr[h(x) = h(y)] \leq \frac{c}{m}$.

For many purposes c-universal hash functions for some small constant c are enough. We will see examples of such functions a little later today.

Multiply-mod-prime

Let $U = [u]$ and pick prime $p \geq u$. For any $a \in [p]_+ = \{1, \ldots, p-1\}$, $b \in [p]$, and $m < u$, let $h^m_{a,b} : U \to [m]$ be

$$h^m_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

Multiply-mod-prime

Let $U = [u]$ and pick prime $p \geq u$. For any $a \in [p]_+ = \{1, \ldots, p-1\}$, $b \in [p]$, and $m < u$, let $h^m_{a,b} : U \to [m]$ be

$$h^m_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

Is this a hash function?

Multiply-mod-prime

Let $U = [u]$ and pick prime $p \geq u$. For any $a \in [p]_+ = \{1, \ldots, p-1\}$, $b \in [p]$, and $m < u$, let $h^m_{a,b} : U \to [m]$ be

$$h^m_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

Is this a hash function? NO!

Multiply-mod-prime

Let $U = [u]$ and pick prime $p \geq u$. For any $a \in [p]_+ = \{1, \ldots, p-1\}$, $b \in [p]$, and $m < u$, let $h^m_{a,b} : U \to [m]$ be

$$h^m_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

Choose $a \in [p]_+$ and $b \in [p]$ uniformly at random, and let $h(x) := h^m_{a,b}(x)$.
Then $h : U \to [m]$ is a universal hash function.

Multiply-shift

Let $U = [2^w]$ and $m = 2^\ell$. For any odd $a \in [2^w]$ define

$$h_a(x) := \left\lfloor \frac{(ax) \bmod 2^w}{2^{w-\ell}} \right\rfloor$$

Choose odd $a \in [2^w]$ uniformly at random, and let $h(x) := h_a(x)$.
Then $h : U \to [m]$ is a 2-universal hash function.

Multiply-shift, C

For $U = [2^{64}]$ the C code looks like this:

```
#include<stdint.h>
uint64_t hash(uint64_t x, uint64_t l, uint64_t a) {
  return (a*x) >> (64-l);
}
```

Hashing with chaining

Goal is to maintain $S \subseteq U$, $|S| = n$ and for $x \in U$ be able to tell if $x \in S$.

Idea: Let $m \geq n$ and pick *universal* $h : U \to [m]$. Store array L, where $L[i] =$ linked list over $\{y \in S \mid h(y) = i\}$.

Then $x \in S \iff x \in L[h(x)]$.

This can be checked in $\mathcal{O}(|L[h(x)]| + 1)$ time.

Hashing with chaining

Goal is to maintain $S \subseteq U$, $|S| = n$ and for $x \in U$ be able to tell if $x \in S$.

Idea: Let $m \geq n$ and pick *universal* $h : U \to [m]$. Store array L, where $L[i] =$ linked list over $\{y \in S \mid h(y) = i\}$.

Then $x \in S \iff x \in L[h(x)]$.

This can be checked in $\mathcal{O}(|L[h(x)]| + 1)$ time.

Hashing with chaining

Goal is to maintain $S \subseteq U$, $|S| = n$ and for $x \in U$ be able to tell if $x \in S$.

Idea: Let $m \geq n$ and pick *universal* $h : U \to [m]$. Store array L, where $L[i] =$ linked list over $\{y \in S \mid h(y) = i\}$.

Then $x \in S \iff x \in L[h(x)]$.

This can be checked in $\mathcal{O}(|L[h(x)]| + 1)$ time.

Hashing with chaining

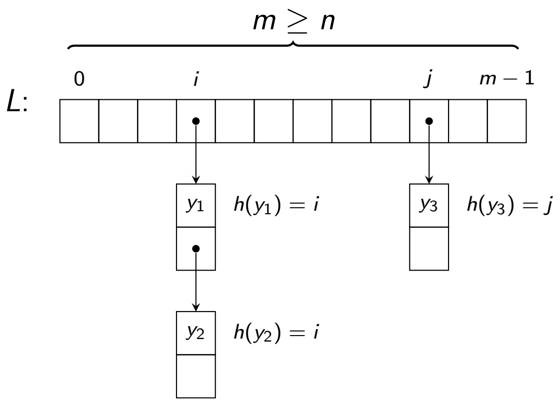
Goal is to maintain $S \subseteq U$, $|S| = n$ and for $x \in U$ be able to tell if $x \in S$.

Idea: Let $m \geq n$ and pick *universal* $h : U \to [m]$. Store array L, where $L[i] =$ linked list over $\{y \in S \mid h(y) = i\}$.

Then $x \in S \iff x \in L[h(x)]$.

This can be checked in $\mathcal{O}(|L[h(x)]| + 1)$ time.

Hashing with chaining



Hashing with chaining

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \leq 1$

Proof.

$$\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\leq |S|\tfrac{1}{m} = \tfrac{n}{m} \leq 1 \qquad \square
\end{aligned}$$

Hashing with chaining

By definition of $L[i] := \{y \in S \mid h(y) = i\}$.

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \le 1$

Proof.

$$
\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\le |S|\tfrac{1}{m} = \tfrac{n}{m} \le 1 \qquad \square
\end{aligned}
$$

Hashing with chaining

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \leq 1$

Proof.

$$\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\leq |S| \tfrac{1}{m} = \tfrac{n}{m} \leq 1
\end{aligned}$$

□

Here we use the *Iverson Bracket* notation

$$[P] := \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{if } P \text{ is false} \end{cases}$$

This can often be used as a shorthand for an indicator variable.
In this case $[h(y) = h(x)]$ becomes an indicator variable for the event $h(y) = h(x)$.

Hashing with chaining

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \leq 1$

Proof.

$$
\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\leq |S| \tfrac{1}{m} = \tfrac{n}{m} \leq 1
\end{aligned}
$$

□

Linearity of expectation.

Hashing with chaining

Expectation of indicator variable

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \leq 1$

Proof.

$$
\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\leq |S| \tfrac{1}{m} = \tfrac{n}{m} \leq 1 \qquad \square
\end{aligned}
$$

Hashing with chaining

Theorem

For $x \notin S$, $\mathbb{E}[|L[h(x)]|] \leq 1$

Proof.

$$
\begin{aligned}
\mathbb{E}[|L[h(x)]|] &= \mathbb{E}\Big[\big|\{y \in S \mid h(y) = h(x)\}\big|\Big] \\
&= \mathbb{E}\Big[\sum_{y \in S} [h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \mathbb{E}\Big[[h(y) = h(x)]\Big] \\
&= \sum_{y \in S} \Pr[h(y) = h(x)] \\
&\leq |S|\frac{1}{m} = \frac{n}{m} \leq 1
\end{aligned}
$$

$\square$

Since $x \notin S$ and $y \in S$, we have $x \neq y$.
Then by definition of a universal hash function $h : U \to [m]$, $\Pr[h(y) = h(x)] \leq \frac{1}{m}$.

Two-level hashing

When storing a *static* set, we can get *worst case* constant time lookup with linear space by using two levels of hashing, as follows:

Step I Pick universal hash function $h : U \to [n]$. Let $S_i = \{x \in S \mid h(x) = i\}$, $s_i = |S_i|$, and $B = \sum_{i=0}^{n-1} \binom{s_i}{2}$. If $B \geq n$, start over with a new h, else continue to Step II.

Step II For each $i \in [n]$, pick a universal hash function $h_i : U \to [s_i(s_i - 1)]$ until h_i has no collisions on S_i.

Two-level hashing

When storing a *static* set, we can get *worst case* constant time lookup with linear space by using two levels of hashing, as follows:

Step I — Pick universal hash function $h : U \to [n]$. Let $S_i = \{x \in S \mid h(x) = i\}$, $s_i = |S_i|$, and $B = \sum_{i=0}^{n-1} \binom{s_i}{2}$. If $B \geq n$, start over with a new h, else continue to Step II.

Step II — For each $i \in [n]$, pick a universal hash function $h_i : U \to [s_i(s_i - 1)]$ until h_i has no collisions on S_i.

Two-level hashing

When storing a *static* set, we can get *worst case* constant time lookup with linear space by using two levels of hashing, as follows:

Step I Pick universal hash function $h : U \to [n]$. Let $S_i = \{x \in S \mid h(x) = i\}$, $s_i = |S_i|$, and $B = \sum_{i=0}^{n-1} \binom{s_i}{2}$. If $B \geq n$, start over with a new h, else continue to Step II.

Step II For each $i \in [n]$, pick a universal hash function $h_i : U \to [s_i(s_i - 1)]$ until h_i has no collisions on S_i.

Two-Level hashing

The expected total time for finding h is $\mathcal{O}(n)$.

$$B = \sum_{\substack{x,y \in S \\ x \neq y}} [h(x) = h(y)]$$

$$\mathbb{E}[B] = \sum_{\substack{x,y \in S \\ x \neq y}} \Pr[h(x) = h(y)] \leq \binom{n}{2} \cdot \frac{1}{n} = \frac{n}{2}$$

$$\Pr[B \geq 2\,\mathbb{E}[B]] \leq \frac{1}{2} \qquad \text{(By Markov)}$$

B is geometrically distributed, so the expected number of trials before $B < n$ is ≤ 2, and each trial takes linear time.

Two-Level hashing

The expected total time for finding h_i is $\mathcal{O}(s_i)$.

$$B_i = \sum_{\substack{x,y \in S_i \\ x \neq y}} [h(x) = h(y)]$$

$$\mathbb{E}[B_i] = \sum_{\substack{x,y \in S_i \\ x \neq y}} \Pr[h(x) = h(y)] \leq \binom{s_i}{2} \cdot \frac{1}{s_i(s_i - 1)} = \frac{1}{2}$$

$$\Pr[B \geq 2\,\mathbb{E}[B]] \leq \frac{1}{2} \qquad \text{(By Markov)}$$

B_i is geometrically distributed, so the expected number of trials before $B_i < 1$ is ≤ 2, and each trial takes linear time.

Summary

We have seen several examples of how randomization can be part of a data structure.

- First as a simple means of balancing a binary search tree by viewing it as a random treap
- Then we defined the concept of a hash function, and saw some actual practical hash functions.
- Finally we saw a simple application of universal hashing, when we looked at hashing with chaining and two-level hashing.
- Next time: More Hashing

Summary

We have seen several examples of how randomization can be part of a data structure.

- First as a simple means of balancing a binary search tree by viewing it as a random treap
- Then we defined the concept of a hash function, and saw some actual practical hash functions.
- Finally we saw a simple application of universal hashing, when we looked at hashing with chaining and two-level hashing.
- Next time: More Hashing

Summary

We have seen several examples of how randomization can be part of a data structure.

- First as a simple means of balancing a binary search tree by viewing it as a random treap
- Then we defined the concept of a hash function, and saw some actual practical hash functions.
- Finally we saw a simple application of universal hashing, when we looked at hashing with chaining and two-level hashing.
- Next time: More Hashing

Summary

We have see several examples of how randomization can be part of a data structure.

- First as a simple means of balancing a binary search tree by viewing it as a random treap
- Then we defined the concept of a hash function, and saw some actual practical hash functions.
- Finally we saw a simple application of universal hashing, when we looked at hashing with chaining and two-level hashing.
- Next time: More Hashing

Summary

We have seen several examples of how randomization can be part of a data structure.

- First as a simple means of balancing a binary search tree by viewing it as a random treap
- Then we defined the concept of a hash function, and saw some actual practical hash functions.
- Finally we saw a simple application of universal hashing, when we looked at hashing with chaining and two-level hashing.
- Next time: More Hashing