Learning Goals

- Definition of a Treap and its motivating ideas
- Definition of a Heap
- Implementation of Treap insertion
- Analysis of the expected performance of a Treap

Treap: Motivating Ideas

- A binary search tree's shape depends on the arrival order of the nodes.

Treap: Motivating Ideas

- A binary search tree's shape depends on the arrival order of the nodes.
  - If the nodes $1, 2, \ldots, n$ arrive in this increasing order and are added with the naïve BST Insert, the resulting tree will be a linked list.

Treap: Motivating Ideas

- A binary search tree's shape depends on the arrival order of the nodes.
  - If the nodes $1, 2, \ldots, n$ arrive in this increasing order and are added with the naïve BST INSERT, the resulting tree will be a linked list.
  - Intuitively, for less adversarial arrival orders, the tree should be somewhat balanced.
- In fact, it can be shown that, if the nodes arrive in a uniformly random order, the expected height of the resulting BST is $O(\log n)$.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.
- If the first node (the root) is the i-th largest, then the height of the resulting tree is $1 + \max\{h_{i-1}, h_{n-i}\}$.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.
- If the first node (the root) is the i-th largest, then the height of the resulting tree is $1 + \max\{h_{i-1}, h_{n-i}\}$.
- When we take the expectation of $\max\{h_{i-1}, h_{n-i}\}$, $\mathbf{E}[\cdot]$ cannot be moved into $\max\{\cdot\}$.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.
- If the first node (the root) is the i-th largest, then the height of the resulting tree is $1 + \max\{h_{i-1}, h_{n-i}\}$.
- When we take the expectation of $\max\{h_{i-1}, h_{n-i}\}$, $\mathbf{E}[\cdot]$ cannot be moved into $\max\{\cdot\}$.
  - In fact, $\mathbf{E}[\max\{X_1, X_2\}] \geq \max\{\mathbf{E}[X_1], \mathbf{E}[X_2]\}$, a consequence of *Jensen's inequality*: $\mathbf{E}_X[f(X)] \geq f(\mathbf{E}_X[X])$ for convex f.
- A common trick to deal with this is to say $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1} + h_{n-i}] = \mathbf{E}[h_i] + \mathbf{E}[h_{n-i}]$.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.
- If the first node (the root) is the i-th largest, then the height of the resulting tree is $1 + \max\{h_{i-1}, h_{n-i}\}$.
- When we take the expectation of $\max\{h_{i-1}, h_{n-i}\}$, $\mathbf{E}[\cdot]$ cannot be moved into $\max\{\cdot\}$.
  - In fact, $\mathbf{E}[\max\{X_1, X_2\}] \geq \max\{\mathbf{E}[X_1], \mathbf{E}[X_2]\}$, a consequence of *Jensen's inequality*: $\mathbf{E}_X[f(X)] \geq f(\mathbf{E}_X[X])$ for convex f.
- A common trick to deal with this is to say $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1} + h_{n-i}] = \mathbf{E}[h_i] + \mathbf{E}[h_{n-i}]$.
- But here such a bound would be too loose.

Proof Sketch for Randomly Built BST

- Rough idea: it is natural to formulate the problem as a recursion, and let h_n be the random variable for the height of the BST formed by n nodes when they arrive in a random order.
- If the first node (the root) is the i-th largest, then the height of the resulting tree is $1 + \max\{h_{i-1}, h_{n-i}\}$.
- When we take the expectation of $\max\{h_{i-1}, h_{n-i}\}$, $\mathbf{E}[\cdot]$ cannot be moved into $\max\{\cdot\}$.
    - In fact, $\mathbf{E}[\max\{X_1, X_2\}] \geq \max\{\mathbf{E}[X_1], \mathbf{E}[X_2]\}$, a consequence of *Jensen's inequality*: $\mathbf{E}_X[f(X)] \geq f(\mathbf{E}_X[X])$ for convex f.
- A common trick to deal with this is to say $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1} + h_{n-i}] = \mathbf{E}[h_i] + \mathbf{E}[h_{n-i}]$.
- But here such a bound would be too loose.
    - If we expect h_i and h_{n-i} differ not much, then we'd lose a factor of 2 each time we apply this.

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.
- Let H_n be 2^{h_n}.

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.
- Let H_n be 2^{h_n}.
- Then $H_n = \frac{1}{n}\sum_{i=1}^{n} 2 \cdot \max\{H_{i-1}, H_{n-i}\}$.

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.
- Let H_n be 2^{h_n}.
- Then $H_n = \frac{1}{n}\sum_{i=1}^{n} 2 \cdot \max\{H_{i-1}, H_{n-i}\}$.

$$\mathbf{E}\left[H_n\right] \leq \frac{2}{n}\sum_{i=1}^{n} \mathbf{E}\left[H_{i-1}\right] + \mathbf{E}\left[H_{n-i}\right] = \frac{4}{n}\sum_{i=1}^{n-1} \mathbf{E}\left[H_i\right].$$

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.
- Let H_n be 2^{h_n}.
- Then $H_n = \frac{1}{n}\sum_{i=1}^{n} 2 \cdot \max\{H_{i-1}, H_{n-i}\}$.

$$\mathbf{E}[H_n] \leq \frac{2}{n}\sum_{i=1}^{n} \mathbf{E}[H_{i-1}] + \mathbf{E}[H_{n-i}] = \frac{4}{n}\sum_{i=1}^{n-1} \mathbf{E}[H_i].$$

- This is a tractable recursion, and one can show that $\mathbf{E}[H_n]$ is polynomial in n. Therefore $\mathbf{E}[h_n]$ is $O(\log n)$.

Proof Sketch for Randomly Built BST (Cont.)

- Another clever idea: we can amplify the quantity we are interested in, so that the estimation error caused by $\mathbf{E}[\max\{h_{i-1}, h_{n-i}\}] \leq \mathbf{E}[h_{i-1}] + \mathbf{E}[h_{n-i}]$ is more negligible.
- Let H_n be 2^{h_n}.
- Then $H_n = \frac{1}{n}\sum_{i=1}^{n} 2 \cdot \max\{H_{i-1}, H_{n-i}\}$.

$$\mathbf{E}[H_n] \leq \frac{2}{n}\sum_{i=1}^{n} \mathbf{E}[H_{i-1}] + \mathbf{E}[H_{n-i}] = \frac{4}{n}\sum_{i=1}^{n-1} \mathbf{E}[H_i].$$

- This is a tractable recursion, and one can show that $\mathbf{E}[H_n]$ is polynomial in n. Therefore $\mathbf{E}[h_n]$ is $O(\log n)$.
  - The latter is another consequence of Jensen's inequality: $2^{\mathbf{E}[h_n]} \leq \mathbf{E}[2^{h_n}] = \mathbf{E}[H_n] = O(n^c)$.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
    - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
  - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.
  - This does not require knowing the set of nodes before they arrive.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
  - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.
  - This does not require knowing the set of nodes before they arrive.
    - When a node arrives, we just sample its position in π by choosing uniformly at randm its position w.r.t. the node that have arrived.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
  - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.
  - This does not require knowing the set of nodes before they arrive.
    - When a node arrives, we just sample its position in π by choosing uniformly at randm its position w.r.t. the node that have arrived.
- If we write $\pi(i)$ on node i to denote its position in our hypothetical ordering π, then the node with the smallest $\pi(\cdot)$ should be the root.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
    - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.
    - This does not require knowing the set of nodes before they arrive.
        - When a node arrives, we just sample its position in π by choosing uniformly at randm its position w.r.t. the node that have arrived.
- If we write $\pi(i)$ on node i to denote its position in our hypothetical ordering π, then the node with the smallest $\pi(\cdot)$ should be the root.
    - The same is true for each of the subtrees.

Back to Reality..

- However, we cannot assume the nodes arrive in uniformly random order.
- Idea: we sample a uniformly random arrival order π for the nodes, then when an node arrives, we insist on treating it as if its position in the arrival order is the one in π and not the one actually observed.
  - For example, if node i arrives first according to π, then we make i the root of the tree upon its arrival, even if it arrives late.
  - This does not require knowing the set of nodes before they arrive.
    - When a node arrives, we just sample its position in π by choosing uniformly at randm its position w.r.t. the node that have arrived.
- If we write $\pi(i)$ on node i to denote its position in our hypothetical ordering π, then the node with the smallest $\pi(\cdot)$ should be the root.
  - The same is true for each of the subtrees.
  - The resulting tree has the property that, for any two nodes x and y, if x is an ancestor of y, then $\pi(x) < \pi(y)$.

Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.

Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.
- It supports the operation of Extract-Max.

Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.
- It supports the operation of EXTRACT-MAX.
  - The algorithm HeapSort uses this for sorting.

Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.
- It supports the operation of Extract-Max.
  - The algorithm HeapSort uses this for sorting.
  - Some graph algorithms use this, e.g. Dijkstra's algorithm for shortest path in graphs with nonnegative weights.

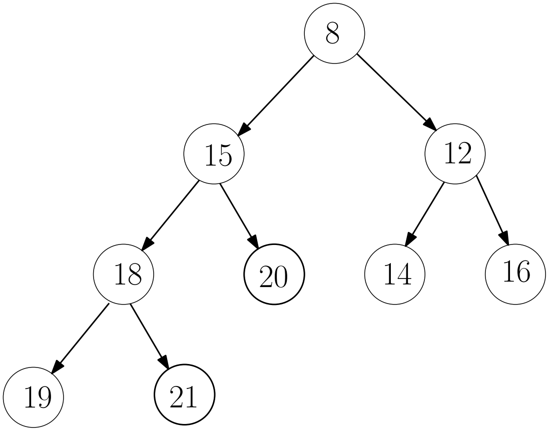
Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.
- It supports the operation of Extract-Max.
  - The algorithm HeapSort uses this for sorting.
  - Some graph algorithms use this, e.g. Dijkstra's algorithm for shortest path in graphs with nonnegative weights.
- Insert(x): insert the new node as a leaf; this may violate the heap property — let it "swim" up the tree by swapping it with the parent as long as the heap property is still violated.

Heaps

- A *heap* is a tree that satisfies the heap property: for any two nodes x and y, if y is a child of x, then $key(x) \leq key(y)$.
- It supports the operation of Extract-Max.
  - The algorithm HeapSort uses this for sorting.
  - Some graph algorithms use this, e.g. Dijkstra's algorithm for shortest path in graphs with nonnegative weights.
- Insert(x): insert the new node as a leaf; this may violate the heap property — let it "swim" up the tree by swapping it with the parent as long as the heap property is still violated.
- Side remark: It is often useful to implement a heap in an array. One need not keep pointers for parents or children.

Heap: An Illustration



Treap

- Idea to simulate random arrival order in building a BST: in addition to the key values, give each node x a random *priority* value $\pi(x) \in [n] := \{1, \ldots, n\}$.

Treap

- Idea to simulate random arrival order in building a BST: in addition to the key values, give each node x a random *priority* value $\pi(x) \in [n] := \{1, \ldots, n\}$.
- Maintain the BST property on the key values, and maintain the heap property on the priority values.

Treap

- Idea to simulate random arrival order in building a BST: in addition to the key values, give each node x a random *priority* value $\pi(x) \in [n] := \{1, \ldots, n\}$.
- Maintain the BST property on the key values, and maintain the heap property on the priority values.
- The resulting data structure is a *Treap*.

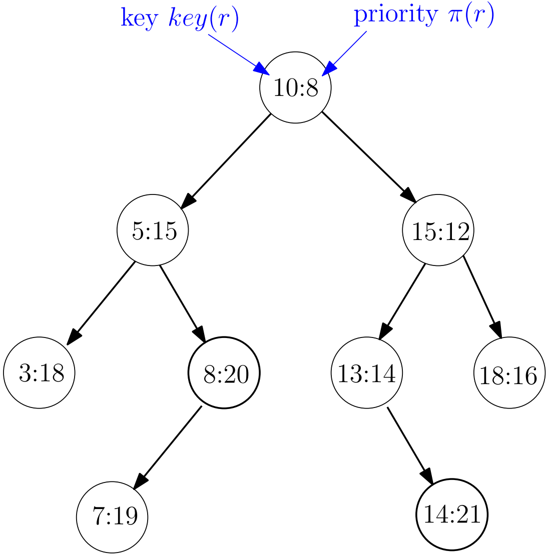
Treap

- Idea to simulate random arrival order in building a BST: in addition to the key values, give each node x a random *priority* value $\pi(x) \in [n] := \{1, \ldots, n\}$.
- Maintain the BST property on the key values, and maintain the heap property on the priority values.
- The resulting data structure is a *Treap*.
- Operation $\textsc{Find}(x)$ is the same as in BST.

Treap

- Idea to simulate random arrival order in building a BST: in addition to the key values, give each node x a random *priority* value $\pi(x) \in [n] := \{1, \ldots, n\}$.
- Maintain the BST property on the key values, and maintain the heap property on the priority values.
- The resulting data structure is a *Treap*.
- Operation $\textsc{Find}(x)$ is the same as in BST.
- Operation $\textsc{Insert}(x, r)$ first does the BST insertion using key values, and then assigns a uniformly random priority value to the new node and lets it swim (using tree rotations!) to restore the heap property on the priority values.

Treap: An Illustration



Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies Find and Insert both take $O(\log n)$ time in expectation.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies Find and Insert both take $O(\log n)$ time in expectation.
  - For Insert, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
  - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of $\textsc{Insert}(x)$: it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
  - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of INSERT(x): it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.
  - Walking down the path from the root to the leaf that is x: $v_1 = r, v_2, \ldots, v_h = x$, let's look at the size of the shrinking subtree rooted at each v_i.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
    - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of INSERT(x): it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.
    - Walking down the path from the root to the leaf that is x: $v_1 = r, v_2, \ldots, v_h = x$, let's look at the size of the shrinking subtree rooted at each v_i.
    - Start with the root, the size of the (sub)tree is n.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
  - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of INSERT(x): it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.
  - Walking down the path from the root to the leaf that is x: $v_1 = r, v_2, \ldots, v_h = x$, let's look at the size of the shrinking subtree rooted at each v_i.
  - Start with the root, the size of the (sub)tree is n.
  - One step down to node v_2. Wlog let's say v_2 is the left child of v_1.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
    - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of INSERT(x): it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.
    - Walking down the path from the root to the leaf that is x: $v_1 = r, v_2, \ldots, v_h = x$, let's look at the size of the shrinking subtree rooted at each v_i.
    - Start with the root, the size of the (sub)tree is n.
    - One step down to node v_2. Wlog let's say v_2 is the left child of v_1.
    - With probability $\frac{3}{4}$, the size of the subtree shrinks by a factor of $\frac{3}{4}$ when we go from $v_0 = r$ to v_1.

Analysis

- The proof that a BST for n randomly arriving nodes has expected height $O(\log n)$ in fact already implies FIND and INSERT both take $O(\log n)$ time in expectation.
  - For INSERT, the number of "swim" steps is bounded by the height of the tree, which is again $O(\log n)$ in expectation.
- An alternative analysis of INSERT(x): it suffices to show that the ordinary tree insertion puts x at a leaf of depth $O(\log n)$.
  - Walking down the path from the root to the leaf that is x: $v_1 = r, v_2, \ldots, v_h = x$, let's look at the size of the shrinking subtree rooted at each v_i.
  - Start with the root, the size of the (sub)tree is n.
  - One step down to node v_2. Wlog let's say v_2 is the left child of v_1.
  - With probability $\frac{3}{4}$, the size of the subtree shrinks by a factor of $\frac{3}{4}$ when we go from $v_0 = r$ to v_1.
  - The same reasoning applies at every step down the path.

Analysis of Insert Cont.

- Let's say a step down the path is *successful* if the size of the subtree shrinks by a factor of $\frac{3}{4}$.

Analysis of INSERT Cont.

- Let's say a step down the path is *successful* if the size of the subtree shrinks by a factor of $\frac{3}{4}$.
- We can have at most $\log_{\frac{4}{3}} n$ successful steps before the subtree becomes a singleton.

Analysis of INSERT Cont.

- Let's say a step down the path is *successful* if the size of the subtree shrinks by a factor of $\frac{3}{4}$.
- We can have at most $\log_{\frac{4}{3}} n$ successful steps before the subtree becomes a singleton.
- This is very similar to the analysis of Quicksort.

Analysis of INSERT Cont.

- Let's say a step down the path is *successful* if the size of the subtree shrinks by a factor of $\frac{3}{4}$.
- We can have at most $\log_{\frac{4}{3}} n$ successful steps before the subtree becomes a singleton.
- This is very similar to the analysis of Quicksort.
- An application of Chernoff bound shows that w.h.p. INSERT takes $O(\log n)$ time.

Analysis of INSERT Cont.

- Let's say a step down the path is *successful* if the size of the subtree shrinks by a factor of $\frac{3}{4}$.
- We can have at most $\log_{\frac{4}{3}} n$ successful steps before the subtree becomes a singleton.
- This is very similar to the analysis of Quicksort.
- An application of Chernoff bound shows that w.h.p. INSERT takes $O(\log n)$ time.
- In fact, another use of union bound shows that, w.h.p. the height of the Treap is $O(\log n)$.