1 The Plan

These notes will guide you through the basic ideas. You should also read (at least) Sections 1-5 of Seidel and Aragon's paper on *randomized search trees*, linked to from the course home page. This week, student A will report on the most interesting aspects of the problems and coding assignment, while student B will summarize the key elements the Seidel and Aragon paper.

2 The Predecessor Problem

Let S be a set of distinct numbers[1]. Given some number x, the *predecessor* of x in S is the largest element in S no exceeding x. Mathematically, we write

$$pred_S(x) = \max\{y \in S \,|\, y \leq x\}.$$

Another way to think about predecessors is to imagine the infinite real number line. Moving left gives smaller numbers while moving right gives larger numbers. Suppose you make a mark on the line at every position given in S. Now, given a number x, find x on the line and move left until you hit a mark. There is no need to move if you're already on a mark. This mark gives the predecessor of x in S.

Given a set of numbers S the *predecessor problem* asks you to preprocess S to efficiently answer predecessor queries for any number x. That is, organize S so that $\text{pred}_S(x)$ is efficient for all x.

Question 1. *What is a good way to solve the predecessor problem for a fixed set of numbers S?*

Question 2. *What does* efficient *mean with respect to the predecessor problem? Why is your definition of efficient good?*

Often we consider the predecessor problem when the set of numbers is dynamic. In this case we allow the insertion and deletion of elements in S. *Binary search trees* provide a solution to the predecessor problem on dynamic sets. A binary search tree for a set of elements S is a binary tree whose internal nodes contain the elements of S and whose leaves (for convenience) contain the null value. In addition, a binary search tree exhibits the *search tree property* which says that if t is an internal node of the the search tree with value x then all elements in the left subtree of t have value less than x and all elements in the right subtree of t have value greater than x. Note here that we are relying on our assumption that S is always a set of distinct numbers. Binary search trees support *insert*, *delete*, and *search* operations that take time linear in the *height* of the tree. Of course, a sequence of these operations may cause the tree to become very imbalanced so that the height is proportional to the size of S.

Question 3. *What is the maximum height of a binary search tree after n numbers have been inserted into it? What sequence of numbers causes this worst-case behavior?*

Because of this poor-worst case performance, the predecessor problem on dynamic sets is usually solved with *balanced binary search trees* like AVL-trees, red-black trees and B-trees. These trees use tricks like rotations to keep the tree balanced. You probably have seen some of these trees previously in a data structures course. Balanced binary trees are pervasive. For example, red-black trees serve as the underlying data structure for the `set` class in the Standard Template Library in C++ and the `TreeMap` and `TreeSet` class in the Java Collections Framework.

Balanced binary trees support $O(\log n)$-time insertion, deletion, successor, predecessor, and search operations. However, most of these data structures are tedious to implement and the constant factors hidden inside the asymptotic notation can sometimes be quite large.

The good news is that if we use a small of amount of randomness, we can design a data structure for the predecessor problem that maintains the logarithmic bounds on the operations in expectation with very high probability. High probability here means that is is more likely for the computer to exhibit hardware failure while simultaneously being hit by an astroid than it is for the operation to take non-logarithmic time. Well, not actually, what it really

[1]This will make our analysis easier later

means is that the probability the tree has depth larger than $c \log n$ is less than n^{-d} where d is a constant that depends on c.

Finally, a note on S. We assume that S is a set of numbers. We do this because it's easy. Often S is a set of objects where each $x \in S$ has some key $key(x)$ associated with it. We use the keys to index the elements in the search trees instead of the elements themselves. However, for convenience we often use x in place of $key(x)$ without confusion.

Question 4. *Given a binary search tree T for the set S and some number x, how does one find $pred_S(x)$ in T? How about the successor?*

Question 5. *Suppose that you were not interested in finding the predecessor or the successor of an item, and instead, only interested in search, insertion, and deletion. This is known as the* dynamic dictionary problem. *What is a good way to solve the dynamic dictionary problem?*

3 Treaps

Many of you might have seen Skip Lists as a solution to the predecessor problem on dynamic sets. A Skip List is a randomized data structure that supports dictionary operations and predecessor / successor searches in $O(\log n)$ time with high probability. This week, we will look at another randomized data structure, the Treap, which also supports dynamic dictionary operations and predecessor / successor searches in $O(\log n)$ time with high probability.

To start, recall that a binary tree T exhibits the *heap property* if, whenever t is an internal node with value x, all the elements in both the left and right subtrees of t have values larger than x. Treaps are binary search trees that maintain the heap property. Thus, a treap is a *portmanteau*[2] of tree and heap. The idea is that each element $x \in S$ has a distinct priority, denoted $priority(x)$, drawn uniformly at random from the reals.[3] A treap maintains the search tree property with respect to the keys of S and the heap property with respect to the priorities of S. Colloquially we say that if x has higher priority than y that $priority(x) < priority(y)$. This is annoying since it can create confusion when doing comparisons. Here when we say x has a higher priority than y we literally mean that $priority(x) > priority(y)$. This means that the element with smallest priority occupies the root of the tree.

Problem 1. *Let S be a set of elements where each element has a distinct key and a distinct priority. Use induction to show that there is exactly one treap for S.*

3.1 Insertion and Deletion

Insertion and deletion in treaps is pretty easy. To insert a new element, we first assign it a random priority, then we use the standard binary search tree insertion algorithm: walk down the tree, moving left if the key value is less than the current node and moving right if the key value is larger than the current node, finally inserting the new element at a leaf in the appropriate spot. Of course, this breaks the heap property, so now you need to rotate the node up with respect to its priority while still maintaining the search-tree property of the tree. You may want to review binary search tree insertion and floating heap elements up if this does not make sense. Notice that the insertion operation is linear in the height of the tree.

Problem 2. *Define a deletion operation for the treap data structure. Its running time should be proportional to the height of the tree. As a hint, think about performing insertion backwards.*

Problem 3. *Let T be a treap on a set of n elements. Describe an operation $split(T, k)$ that splits T into two treaps T_1 and T_2 where T_1 contains all the elements from T with key values at most k and T_2 contains all the elements from T with key values larger than k. Prove that your algorithm runs in time proportional to the height of T.*

[2]Other examples of portmanteaus are *TomKat*, *Brangelina*, and *gerrymander*

[3]In reality, one can use any subinterval of the reals like (0,1)

3.2 Analysis

Since a treap is just a binary search tree, we might not expect it to behave any better than an unbalanced binary search tree—it might have bad worst-case performance. However, this is not the case in expectation. Here's some intuition: Imagine that, given a set of distinct numbers S, you insert the numbers from S into a standard binary search tree, but that you do so by choosing a number uniformly at random from the remaining elements in S. In this case, the binary search tree is very likely balanced. In fact, the expected height of the tree is $O(\log n)$ with high probability. This is not surprising given that this process precisely mimics the process of randomized quicksort—the random element behaves like a random pivot. Assigning a random priority to each element before inserting it into the treap means that each element is equally likely to be the root. So it too mimics the randomized quicksort algorithm. The analysis below is inspired by this observation.

We will show that a treap is well-balanced in expectation[4]. Given a treap T and an element x, let $depth(x)$ be the depth of x in T. The depth of an element is its distance from the root. So the root has depth 0, the children of the root each have depth 1, and so on. Order the elements in S by their rank so that $x_1 < x_2 < x_3 < \cdots < x_n$. We will show that the expected depth of any element x_k is at most $2 \log n$. Note that $depth(x_k)$ is a random variable since its insertion is governed by a random process.

Lemma 1. $E[depth(x_k)] < 2 \log n$

Proof. The value of $depth(x_k)$ is determined by the number of elements that are ancestors of it in T. Let X_i be an indicator random variable so that $X_i = 1$ if and only if element x_i is an ancestor of x_k in T. Then

$$E[depth(x_k)] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} Pr[X_i = 1]$$

where the penultimate equality uses the linearity of expectation (i.e., the expectation of a sum is the sum of the expectations) and the final equality follows from the definition of expectation. This leaves us with the problem of computing the probability that x_i is an ancestor of x_k. Order the elements by their rank so $x_1 < x_2 < \ldots x_n$. We consider two cases: when $i < k$ and when $i > k$.

Question 6. *Why do we ignore the case where $i = k$?*

Claim 1. *If $i < k$ then x_i is an ancestor of x_k if and only if $priority(x_i) < priority(x_j)$ for all $i < j \le k$. If $i > k$ then x_i is an ancestor of x_k if and only if $priority(x_i) < priority(x_j)$ for all $k \le j < i$.*

Problem 4. *Prove Claim 1.*

Assuming that Claim 1 holds then the probability that x_i is an ancestor of x_k is always $1/(k - i + 1)$ when $i < k$ and $1/(i - k + 1)$ when $i > k$. This is true because the priorities are assigned uniformly at random so each element has an equal chance of having the smallest priority. Now we have

[4]This analysis is derived from Mark de Berg's notes and Motwani and Raghavan's Randomized Algorithms text. M&R is on reserve in Schow Library.

$$\begin{aligned}
E[depth(x_k)] &= \sum_{i=1}^{n} Pr[X_i = 1] \\
&= \sum_{i=1}^{k-1} Pr[x_i \text{ is an ancestor of } x_k] + \sum_{i=k+1}^{n} Pr[x_i \text{ is an ancestor of } x_k] \\
&= \sum_{i=1}^{k-1} 1/(k-i+1) + \sum_{i=k+1}^{n} 1/(i-k+1) \\
&= \sum_{i=2}^{k} 1/i + \sum_{i=2}^{n-k+1} 1/i \\
&= H_k + H_{n-k+1} \qquad \text{where } H_t = \textstyle\sum_{j=1}^{t} 1/j = \text{the } t^{th} \text{ harmonic number} < \ln t + 1 \\
&< 2 + \ln k + \ln(n-k+1) \\
&< 2\log n
\end{aligned}$$

□

Unfortunately, the fact that the expected depth of an element is $O(\log n)$ does not imply that the expected depth of all the elements is $O(\log n)$ simultaneously. For this, we need to quantify the actual probability that a node has depth $O(\log n)$. In other words, we will show that the expected depth of an element occurs with high probability. Remember that we expressed the depth of a node x_k by the sum of its indicator random variables X_i where $X_i = 1$ if and only if x_i is an ancestor of x_k.

Question 7. *Argue that the indicator random variables for a fixed x_k are independent of one another.*

A Bernoulli trial is a coin flip where the probability of heads is p and the probability of tails is $1 - p$. If p is allowed to change from one independent Bernoulli trial to the next, then we have a sequence of Poisson trials.

Question 8. *Why do the indicator random variables represent Poisson trials and not Bernoulli trials?*

Since the random indicator variables above represent n Poisson trials, we can apply the Chernoff bound from Lemma 1.2[5] of the de Berg probability handout. We restate the Lemma here for completeness.

Lemma 2 (Chernoff Bound)**.** *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \le i \le n$, $Pr[X_1 = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = E[X] = \sum_{x=1}^{n} p_i$ and any $\delta > 0$,*

$$Pr[X > (1+\delta)\mu] \le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}.$$

If $1 + \delta$ is a sufficiently large constant then the Chernoff bound says that the probability that X deviates largely from its expectation decreases exponentially in the expectation.

Problem 5. *Choose $\delta = e^2 - 1$ and use the fact that $H_k + H_{n-k+1} \ge H_n \ge \ln n$ to show that*

$$Pr[depth(x_k) \le 32 \ln n] \ge 1 - (1/n^6).$$

Hint: use the Chernoff bound from above to show that $Pr[depth(x_k) > 32 \ln n] \le 1/n^6$.

[5] This is Theorem 4.1 in Motwani and Raghavan

We have now shown that every $x_i \in S$ has depth $O(\log n)$ with probability at least $1 - (1/n^6)$. However, we want this to be true for every node in S. For this, we use a probabilistic tool called the *union bound*. The union bound says that, given a bunch of events, the probability of at least one of them happening is at most the sum of the probabilities of any of them happening. In other words, if $Y_1, Y_2, \ldots, Y_n$ are events, then

$$Pr[Y_1 \cup Y_2 \cup \cdots \cup Y_n] \leq Pr[Y_1] + Pr[Y_2] + \cdots + Pr[Y_n].$$

We often use the union bound to bound bad events, so if each Y_i is a bad event that occurs with some small probability, then the probability that any of bad events occurs is still pretty small. So, if we take Y_i to be the event that $depth(x_i) > 32 \ln n$ then the probability that at least one of the nodes has depth larger than $32 \ln n$ is at most $n \cdot (1/n^6) = 1/n^5$. Thus, the probability that the whole tree has depth at most $32 \ln n$ is $1 - 1/n^5$. That's pretty awesome.

4 Theory vs. Practice

Often there is a disconnect between theory and practice. We will very much try to remedy this gap by discussing the feasibility of certain algorithm and data structure implementations. Our first foray into this chasm is some empirical work to both confirm our analysis from above and experimentally validate the claim that a Treap is a competitive alternative to the standard, deterministic balanced binary search trees.

Problem 6. *You will implement a treap data structure in Java and compare it with the the* `TreeSet` *class in Java which, as we already mentioned, is a red-black tree implementation of a set. Read the description of the Java* `TreeSet` *at* `http://download.oracle.com/javase/6/docs/api/`*. Notice that its interface includes the method* `add` *for insertion, the method* `ceiling` *for successor, the method* `floor` *for predecessor, the method* `remove` *for delete, and the method* `contains` *which is essentially search since this set class assumes (rightfully so) that two elements with the same key are actually the same. You'll notice that the* `TreeMap` *class (on which the* `TreeSet` *class is based) has a* `get` *method which retrieves the* value *for a* key. *We will base our treap interface on the interface for* `TreeSet`. *You should perform the following:*

- *Implement a Treap data structure, parameterized by type* `E`*, in a file called* `Treap.java`*. The class should have at least three constructors, a default constructor which constructs a new treap, sorted according to the natural ordering of its elements, one which accepts a* `Comparator`*, and one which accepts a Java* `Collection` *and constructs a new treap containing the elements in the specified collection, sorted according to the natural ordering of the elements. You'll see these are essentially the same constructors as the first three listed on the Java doc page.*
- *Provide methods for* `add`, `ceiling`, `remove`, `floor`, *and* `contains`.
- *Perform the following experiment: Generate 50000 random integers and insert them it into both the treap and the* `TreeSet`*. Record the time it took to perform the insertion (over groups of, say, 1000) for both data structures. Making sure to not delete the integers from the sets, repeat this process until both data structures contain the same random integers. Now create a graph where size is on the horizontal axis and time is on the vertical axis and the performance of the two data structures is compared. How did the treap perform? Does the running time scale logarithmically with the size?*
- *Perform one more experiment of your own choosing. Provide a clear, complete explanation of your results. Please include graphs and figures where appropriate.*
- *Implement your $split(T, k)$ operation from Problem 3 and provide some empirical evidence that it runs in time proportional to the height of the tree.*
- *Please turn in your code using the* `turnin` *command on the unix machines.*