CSE 100: TREAPS AND RANDOMIZED SEARCH TREES
Announcements

- PA2 is released and it is about RSTs.
Today: Treaps as a backbone for RSTs

- AVL trees are difficult to implement
- Standard BSTs are likely to be horribly imbalanced and therefore slow
- Solution: A tree that is expected to be balanced: a Randomized Search Tree.
  - A “treap” is the underlying structure of an RST.
Trees + Heaps = Treaps!

Treaps maintain the following properties:

- Data keys are organized as in a binary search tree
  - All nodes in left subtree are less than the root of that subtree, all nodes in right are greater
- Priorities are organized as in a heap
  - The priority of a node is always greater than the priorities of its children.
Treaps are not necessarily balanced!

A bad match between keys and priorities can lead to a very imbalanced treap.

We will look at ways to ensure this doesn’t happen on average… when we discuss RSTs.
Treap insert

Letters = keys
Numbers = priorities

Insert (F, 40)
Treap insert

Letters = keys
Numbers = priorities

Step 1: Insert using BST insert

Insert (F, 40)
Step 1: Insert using BST insert

Insert (F, 40)
Step 1: Insert using BST insert
Step 2: Fix heap ordering

Idea 1: “Bubble” F up until it’s in the right place…

Letters = keys
Numbers = priorities
Step 1: Insert using BST insert
Step 2: Fix heap ordering

Idea 1: “Bubble” F up until it’s in the right place…
Treap insert

Step 1: Insert using BST insert
Step 2: Fix heap ordering

Idea 1: “Bubble” F up until it’s in the right place…
Does this work?
A. Yes
B. No
AVL rotation to the rescue!

Step 1: Insert using BST insert
Step 2: Fix heap ordering

Letters = keys
Numbers = priorities

Insert (F, 40)
AVL rotation to the rescue!

Step 1: Insert using BST insert
Step 2: Fix heap ordering

Insert (F, 40)
AVL rotation to the rescue!

Step 1: Insert using BST insert
Step 2: Fix heap ordering

Letters = keys
Numbers = priorities

Insert (F, 40)
AVL rotation to the rescue!

What should we do next?
A. Rotate right at F to get C, F, and E “in a line”
B. Simply rotate left at C
Treaps don’t require balance

Unlike an AVL tree, the series of rotations performed in a treap serve only to propagate notes up or down the tree while maintaining the BST property. We do not care about keeping the tree balanced.
Treaps don’t require balance

Letters = keys
Numbers = priorities
Treaps don’t require balance

Letters = keys
Numbers = priorities
Treaps don’t require balance

Letters = keys
Numbers = priorities
Tips for coding your treap rotations

• Study the generalized AVL tree single rotations carefully.

• Divide into two cases:
  • 1. Rotation right (i.e. node that wants to move up is the left child of its parent)
  • 2. Rotate left (i.e. node that wants to move up is the right child of its parent)

• For each case determine:
  • Which links to “cut” (i.e., pointers that will need to be reassigned)
    • Be sure to consider all cases—think about all the children and parents of all of the nodes involved.
  • Where to reattach these links (i.e. what their new values will be)

  *It is simply the cutting and restoring of links that creates the rotation.*
Graphical depiction of the general case of a treap rotation
Graphical depiction of the general case of a single rotation
Graphical depiction of the general case of a single rotation

Rotate left
Graphical depiction of the general case of a single rotation

Rotate left
Graphical depiction of the general case of a single rotation

Rotate left
Graphical depiction of the general case of a single rotation

(Of course the modifications to the links can be done in any order)
How would you delete in a treap?

Delete D?
Delete C??
How would you delete in a treap? Rotate down!

Which node would I rotate C with to delete it?
A. Node E  
B. Node B  
C. Node F  
D. None of these
How would you delete in a treap? Rotate down!

Delete C
How would you delete in a treap? Rotate down!

Delete C
How would you delete in a treap? Rotate down!

Delete C
How would you delete in a treap? Rotate down!

Delete C
Warm up with Treaps

Insert (R, 45) into the treap
Warm up with Treaps

Letters = keys
Numbers = priorities

Insert (R, 45) into the treap
Warm up with Treaps

Insert (R, 45) into the treap
Warm up with Treaps

Insert \((R, 45)\) into the treap
Warm up with Treaps

Insert (R, 45) into the treap

Letters = keys
Numbers = priorities
Warm up with Treaps

Letters = keys
Numbers = priorities

Insert (R, 45) into the treap
Why Treaps?

• Treaps are worth studying because...
  • they permit very easy implementations of split and join operations, as well as pretty simple implementations of insert, delete, and find
  • they are the basis of randomized search trees, which have performance comparable to balanced search trees but are simpler to implement
  • they also lend themselves well to more advanced tree concepts, such as weighted trees, interval trees, etc.

• We will look at the first two of these points
Tree splitting

• The tree splitting problem is this:
  • Given a tree and a key value K not in the tree, create two trees:
    One with keys less than K, and one with keys greater than K
• This is easy to solve with a treap, once the insert operation has been implemented:
  • Insert (K, INFINITY) in the treap
  • Since this has a higher priority than any node in the heap, it will become the root of the treap after insertion
  • Because of the BST ordering property, the left subtree of the root will be a treap with keys less than K, and the right subtree of the root will be a treap with keys greater than K. These subtrees then are the desired result of the split
• Since insert can be done in time O(H) where H is the height of the treap, splitting can also be done in time O(H)

How could you do a join? For the answer, see http://cseweb.ucsd.edu/users/kube/cls/100/Lectures/lec5/lec5.pdf
Worst case time to find in a treap

• The worst case time to find an element in a treap is $O(H)$, where $H$ is the height of the treap.
• Unfortunately, a standard treap can become very unbalanced.
• Is it possible to develop a “balanced treap” data structure (while still maintaining the two essential treap relationships)?
Randomized Search Trees

- Randomized search trees were invented by Cecilia Aragon and Raimund Seidel, in early 1990’s
- RST’s are treaps in which priorities are assigned randomly by the insert algorithm when keys are inserted
- To implement a randomized search tree:
  - Adapt a treap implementation and its insert method that takes a (key,priority) pair as argument
  - To implement the RST insert method that takes a key as argument:
    - call a random number generator to generate a uniformly distributed random priority (a 32-bit random int is more than enough in typical applications; fewer bits can also be made to work well) that is independent of the key
    - call the treap insert method with the key value and that priority
- That’s all there is to it: none of the other treap operations need to be changed at all
- (The RST implementation should take care to hide the random priorities, however)

In your PA you are implementing RST insert!
Analysis of RSTs

• How many steps are required, on average, to find that the key you’re looking for is in the tree? (Average case analysis of a “successful find”)

• You can read about this here: http://cseweb.ucsd.edu/users/kube/cls/100/Lectures/lec5/lec5.pdf

• Punch line: The average number of comparisons for a successful find in an RST is exactly the same as the average number of comparisons in a BST!

\[ D_{avg}(N) = \frac{2(N+1)}{N} \sum_{i=1}^{N} \frac{1}{i} - 3 \]

So what have we gained by using an RST??
BST Probabilistic Assumptions

Which of the following is/are the probabilistic assumptions we made in our average case successful find in a BST?

A. All keys are equally likely to be searched for
B. The tree is approximately balanced
C. All orders of data are equally likely to occur
D. A&B
E. A&C
RST Probabilistic Assumptions

• Which of the probabilistic assumptions from the BST is NOT included in the RST analysis?

A. All keys are equally likely to be searched for
B. All orders of data are equally likely to occur

Why not?
What second assumption is it replaced with? Why is that important?
RST Probabilistic Assumptions

• Suppose you have a RST with \(N\) nodes \(x_1, \ldots, x_N\), holding keys \(k_1, \ldots, k_N\) and priorities \(p_1, \ldots, p_N\), such that \(x_i\) is the node holding key \(k_i\) and priority \(p_i\).

• We make the following 2 probabilistic assumptions:
  - **Assumption #1**: Each key in the tree is equally likely to be searched for.
  - **Assumption #2**: The priorities are randomly uniformly generated independently of each other and of the keys.

• For convenience we will assume that:
  - keys are listed in sorted order: for all \(0 < i < N\) (though keys can be inserted in any order),
  - all priorities are distinct.
Recall our BST analysis…

\[ D(N) = \sum_{\text{all BSTs with } N \text{ nodes}} \left( \frac{1}{N!} \right) \left( \sum_{i=1}^{N} d(x_i) \right) \]

What is \( d(x_i) \)?
A. The average depth of a node in a specific BST with \( N \) nodes
B. The average depth of a node in any BST with \( N \) nodes
C. The depth of the node \( x_i \) in the average BST with \( N \) nodes
D. The depth of the node \( x_i \) in a specific BST with \( N \) nodes
Recall our BST analysis…

\[ D(N) = \sum_{\text{all BSTs with } N \text{ nodes}} \left( \frac{1}{N!} \right) \left( \sum_{i=1}^{N} d(x_i) \right) \]

What does the inner sum do (boxed)?
A. Finds the average depth of a node in a specific BST with \( N \) nodes
B. Finds the total depth of the nodes in a specific BST with \( N \) nodes
C. Finds the average depth of a node in the average BST with \( N \) nodes
Recall our BST analysis…

\[
D(N) = \sum_{\text{all BSTs with } N \text{ nodes}} \left( \frac{1}{N!} \right) \left( \sum_{i=1}^{N} d(x_i) \right)
\]

What does the outer sum do?
A. Finds the average total depth of all BSTs with N nodes
B. Finds the total depth of all BSTs with N nodes
C. Finds the average time to find a node in any BST with N nodes
Expected node depth in an RST

• Now, \( d(x_i) \) is the depth of the node with the \( i \)th smallest key (recall that we assume keys are ordered from smallest to largest).

• We want to find the expected depth of this specific node across all RSTs with \( N \) nodes (similar to the outer sum before, but for one specific node).

• Before, we averaged over all possible key insertion sequences. Should we do this again?

  • A. Yes
  • B. No
  • C. I have no idea…
Expected node depth in an RST

• We will average over all ways of generating priorities during insertion, because that’s what will affect the position of this node in the treap given our assumption that the keys are in sorted order
• Let $Pr(p_1, ..., p_N)$ be the probability of generating $N$ priority values $p_1, ..., p_N$
• So the expected value (i.e. average) of $d(x_i)$ is:

$$E[d(x_i)] = \sum_{p_1, ..., p_N} Pr(p_1, ..., p_N)d(x_i)$$
Expected depth and ancestors

- Define $A_{ij}$ to be the “indicator function” for the RST’s ancestor relation:

$$A_{ij} = \begin{cases} 
1, & \text{if } x_i \text{ is an ancestor of } x_j \\
0, & \text{otherwise}
\end{cases}$$

$$A_{ii} = 1$$

- Which of the following correctly represents $d(x_i)$ in terms of $A_{ij}$

A. $d(x_i) = \sum_{m=1}^{N} A_{mi}$  
B. $d(x_i) = \sum_{m=1}^{N} A_{im}$  
C. $d(x_i) = \sum_{i=1}^{N} A_{ii}$
Expected depth and ancestors

- Define $A_{ij}$ to be the “indicator function” for the RST’s ancestor relation:
  \[
  A_{ij} = \begin{cases} 
  1, & \text{if } x_i \text{ is an ancestor of } x_j \\
  0, & \text{otherwise} 
  \end{cases}
  \]

  \[
  d(x_i) = \sum_{m=1}^{N} A_{mi} 
  \]
  \[
  A_{ii} = 1 
  \]

  So, the expected value of the depth of node $x_i$ is (using the fact that the expectation of a sum is the sum of the expectations):

  \[
  E[d(x_i)] = \sum_{p_1, \ldots, p_N} Pr(p_1, \ldots, p_N) d(x_i) 
  \]

  \[
  E[d(x_i)] = \sum_{p_1, \ldots, p_N} Pr(p_1, \ldots, p_N) \sum_{m=1}^{N} A_{mi} = \sum_{m=1}^{N} E[A_{mi}] 
  \]
Probability of being an ancestor

- $A_{ij}$ is a random variable that takes values 0, 1
- Its expected value is just equal to the probability that it has value 1, i.e.
  \[ E[A_{mi}] = Pr[A_{mi} = 1] \]
- What is $Pr[A_{mi} = 1]$, in English?
- A. The probability that node $x_m$ is an ancestor of node $x_i$
- B. The average depth of node $x_i$
- C. The average difference in depth between nodes $x_m$ and $x_i$

\[
E[d(x_i)] = \sum_{m=1}^{N} E[A_{mi}]
\]
Probability of being an ancestor

- $A_{ij}$ is a random variable that takes values 0, 1
- Its expected value is just equal to the probability that it has value 1, i.e.
  \[ E[A_{mi}] = Pr[A_{mi} = 1] \]

Lemma (by Seidel and Aragon): $x_m$ is an ancestor of $x_i$ if and only if among all priorities $priority_h$ such that $h$ lies between the indices $m$ and $i$ inclusive, $priority_m$ is the largest

Convince yourself you believe it.
Probability of being an ancestor

- $A_{ij}$ is a random variable that takes values 0, 1
- Its expected value is just equal to the probability that it has value 1, i.e.
  $$E[A_{mi}] = Pr[A_{mi} = 1]$$  The probability that node $x_m$ is an ancestor of node $x_i$

Lemma (by Seidel and Aragon): $x_m$ is an ancestor of $x_i$ if and only if among all priorities $p_h$ such that $h$ lies between the indices $m$ and $i$ inclusive, $p_m$ is the largest

- So, probability that node $x_m$ is an ancestor of $x_i$ is just the probability that the random priority generated for $x_m$ is higher than the other priorities generated for nodes with indexes between $m$ and $i$ inclusive
- In total, there are $|m - i| + 1$ of these nodes. What is the probability that any one of these nodes has the highest priority?

  A. It depends on the node  
  B. $1/(|m - i| + 1)$  
  C. $1/N$  
  D. $1/(m \times i)$
Expected depth of $x_i$

$$E[A_{mi}] = \frac{1}{|m-i| + 1}$$

$$E[d(x_i)] = \sum_{m=1}^{N} E[A_{mi}] = \sum_{m=1}^{N} \frac{1}{|m-i| + 1}$$

This is the expected depth of a single node in any RST. So where do we go from here?

A. We’re done  
B. We need to average over all nodes in the tree  
C. We need to average over all insertion orders of the keys  
D. We need to average over all probability sequences
Expected depth of $x_i$

$$E[A_{mi}] = \frac{1}{|m-i| + 1}$$

$$E[d(x_i)] = \sum_{m=1}^{N} E[A_{mi}] = \sum_{m=1}^{N} \frac{1}{|m-i| + 1}$$

This is the expected depth of a single node in any RST. Below is the expected depth of any node in any RST:

$$D_{avg}(N) = \frac{1}{N} \sum_{i=1}^{N} E[d(x_i)] = \frac{1}{N} \sum_{i=1}^{N} \sum_{m=1}^{N} \frac{1}{|m-i| + 1}$$
The solution

\[
D_{\text{avg}}(N) = \sum_{i=1}^{N} \frac{1}{N} (E[d(x_i)]) = \frac{1}{N} \sum_{i=1}^{N} \sum_{m=1}^{N} \frac{1}{|m-i|+1}
\]

Can be simplified to

\[
D_{\text{avg}}(N) = \frac{2(N+1)}{N} \sum_{i=1}^{N} \frac{1}{i} - 3
\]

Which is exactly the same as the average time to find an element in a normal BST!

(Why does this make sense?)

So what have we gained??
Comparing RST’s and vanilla BST’s

- We have seen that in the average depth of a node in a N-node RST is the same as in a N-node BST: for large N it is approximately \(2 \ln(N) = 1.386 \log_2 N\), which is \(O(\log N)\).

- This seems very good, but the analysis for the BST depended on the assumption that all sequences of key insertions were equally likely.

- Often in practice “bad” sequences of BST key insertions (in which the keys are somewhat sorted) can in fact be more likely than others.

- Also, if a particular sequence is “bad”, it will be bad (leading to much worse than \(2\ln N\) average depth) every time a BST is built with that sequence.

- However, in a randomized search tree, the average case analysis is independent of the sequence of key insertions.

- If a good random number generator is used to generate the treap priorities, the probability of constructing a “bad” randomized search tree is very low, no matter what the sequence of key insertions is.

- That’s why RST’s are better than vanilla BST’s!