# Recitation 12 — Tree Annotation and Exam Debriefing

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

*16 November 2011*

**Today's Agenda:**

- Interval Stabbing
- Treaps and Independence
- Dijkstra vs. $A^*$

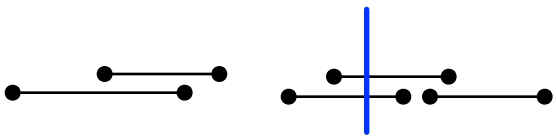
**Announcements:** Assignment 7? Questions? about class, the exam, or anything?

## 1 Interval Stabbing

Consider the following problem, which we'll call the *interval stabbing* problem: We would like to maintain a set $S$ of closed *labeled* intervals (e.g., $S = \{([0,2],1),([1,5],0)\}$). Each of these intervals is labeled with a unique integer. We're interested in supporting BST operations (insert, delete, join, split, etc) and the following operation:

> $\mathsf{stab}(S,x)$ finds an interval (indeed, any interval) in $S$ that is "stabbed" by $x$—that is, $\mathsf{stab}(S,x)$ returns its label $\ell$ such that $(I,\ell) \in S$ and $x \in I$.

Pictorially, we have



For simplicity, we'll assume that all the endpoints are unique.

Aside #1: We could also support delete but to simplify the presentation, we'll ignore it.

Aside #2: You have seen a slight variant of this problem already (on practice exam I). Though, in the current version, you don't know the queries up front and you are allowed to modify the set of intervals.

### 1.1 Digression: Prefix Max Query

Instead of trying to solve this problem directly, we turn to a seemingly unrelated problem, which in this recitation we will call the *prefix-sum query* (PMQ) problem, with the goal of reducing the interval stabbing problem to this new problem. In PMQ, you maintain a set $T$ of ordered pairs $\{(x,(y,p))\}$ while supporting standard BST operations (split, join, etc.) and the query $\mathsf{maxLEQ}(T,x')$, which returns the maximum $y$ value, together with its corresponding payload $p$, of every pair whose $x$ value is at most $x'$; that is, it returns the pair $(y,p)$ such that $y$ is the maximum $y$ value over all ordered pairs whose $x \leq x'$.

**Q:** Any idea how we would solve such a problem?

**A:** Augment binary trees.

**Q:** What exactly do you mean by augmenting binary trees? Can you be more specific?

One way to accomplish this is to store these $(x, (y, p))$ pairs (think (key, (value,payload)) pairs) in a BST as we normally do, but in addition, we associate to each node of our BST the max within the subtree rooted at that node. *Let's write code for this.* In the interest of time, we'll do it for standard BSTs—you can easily adapt this to Treaps. We'll also assume that the payload $p$ has type `payload`.

With this, we can define our BST data type as follows:

```
datatype bst = TL |
  TN of {left:  bst, right:  bst, key:  int, v:  int*payload,
  max:  (int*payload) option} (* <-- this is what we augmented *)
```

Notice that we have augmented each node (TN) with a "max" field, which we'll use to keep the maximum value in the corresponding subtree (as well as its payload). We'll also write a function to extract this max value and a function that returns the bigger of the two inputs.

```
fun getMax T =
  case T
   of TL => NONE
    | TN{max,...} => max

fun greaterOf (maxA, maxB) =
  case (maxA, maxB)
   of (NONE, _) => maxB
    | (_,NONE) => maxA
    | (SOME(vA, _), SOME(vB, _)) =>
      (case Int.compare(vA,vB)
         of GREATER => maxA
         | _ => maxB)
```

Our first task will be to write the join function. To get started, let's look at the simplest case when we have a tree $T_1$ with keys smaller than a key-value pair $(k, v)$ which is smaller than the keys in $T_2$—and we'd like to join them together. Let's write a `mkNode` function:

```
fun mkNode {l:bst,r:bst,k:int,v:int*payload} =
  let val max = greaterOf(getMax l, greaterOf(SOME v, getMax r))
  in TN{left=l,right=r,key=k,v=v,max=max}
  end
```

With `mkNode` in place, the join function is completely identical to what we had before.

```
fun join(T1, T2) =
  case T2
   of TL => T1
  | TN{left,right,key=k,v=v,...} =>
      mkNode{l=join(T1,NONE,left),r=right,k=k,v=v}
```

We can implement split (which internally makes use of join) like before (won't cover here). And we're in a position to implement maxLEQ:

```
fun maxLEQ(T, k) =
  case split(T, k)
   of (lt, NONE, _) => getMax lt
    | (lt, SOME(_,v), _) => greaterOf(getMax lt, SOME v)
```

### 1.2 Implementing Interval Stabbing

We will reduce the interval stabbing problem to the PMQ problem. Before we formally describe the reduction, let's try to get a feel of what the reduction should look like.

First notice that we can stab an interval only if it begins before our stab query point. These are the candidate intervals—we don't know if we'll stab them for sure because they might end before our stab query point.

**Q:** How do we identify the candidate intervals?

**A:** Conveniently, if the tree is ordered by the left endpoint, we will do a split at the stab query point $x'$. This split will give us the set of candidate intervals that start before $x'$.

**Q:** Which candidate interval (among those starting before $x'$) is the most likely $x'$ will stab?

**A:** The one that has the right endpoint that is furthest to the right.

**Q:** How can we identify that interval? In particular, which interval should the algorithm return?

We can take advantage of PMQ and find the maximum right endpoint of the candidate intervals.

**Q:** How do we know whether this right-most endpoint that PMQ returned, say $r$, is stabbed by $x'$?

**A:** There are two possibilities at this point:

1. If $r < x'$ (i.e., the right-most endpoint is before the query point), then you know you didn't stab any interval.
2. Otherwise, you know that you have stabbed an interval—and we can carry in our payload the label of the interval.

This leads to the following code:

```
fun make (S : ((int*int)*payload) seq) =
  let val sortedX = sort cmpX S
      val littleTrees : bst seq = map (fn ((lx,rx),l) => mkSingleton(lx,(rx,l))) sortedS
  in reduce (fn (t1,t2) => join(t1,t2)) TL littleTrees
  end

fun stab Q x' =
  case maxLT(Q, x')
   of NONE => NONE
    | SOME(rx, l) =>
      (case Int.compare (rx, x')
        of LESS => NONE
         | _ => SOME l)
```

## 2 Treaps and Independence

Assume that priorities are generated using a random hash function $h\colon \mathsf{keys} \to [0, 1]$. For keys $1, 2, 3, 4, 5$, assume the corresponding hash values are as follows:

| key | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $h$(key) | 0.4 | 0.1 | 0.5 | 0.2 | 0.6 |

What would the final Treap look like as a result of inserting the keys $1, 4, 2, 5, 3$ in this order?

Many students were confused by this question because it said to "insert .... in this order". In fact, as we proved in class, the order in which we insert these keys doesn't matter, and it is easier to figure out the Treap by considering the set of keys as a whole. We will now discuss this.

**Q:** What is a Treap?

**A:** As you remember, Treap is a wordplay on Binary Search Tree + Heap.

**Q:** Treap must conform to both BST and Heap (either max or min heap) properties. Specifically, a Treap is a tree where each node is associated with a (key, priority) pair. The keys satisfy the BST property and (random) priorities satisty the Heap property.

But what is the BST property and what is the max-Heap property?

**A:** *The BST Property:* a node with a key $k$ has a left subtree and right sub-tree. Everything in the left subtree is less than the key and everything in the right subtree is greater than the key. *Heap:* The priority of a node is greater than any of its sub-trees.

**Q:** Based on these properties, which one of the keys is the root?

**A:** The one with greatest priority, which is "5".

**Q:** Then which keys will be on the left side of the root, and which on the right side?

**A:** We are following the BST property. So all other keys are less than "5", and they will be on the left side.

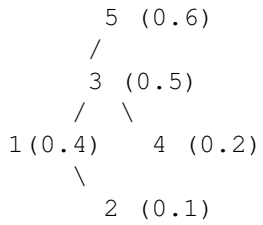
**Q:** How many options did we have in choosing the root?

**A:** We had no other choice.

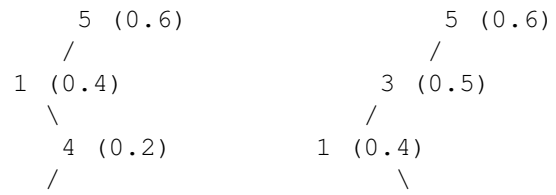
**Q:** Since both the left and right subtrees of the root are Treaps, how many possible different Treaps can we have for a given (key,priority) sequence?

**A:** There is only one choice! For each subtree, there is a unique Treap. This can be shown recursively. That is: there is only one possible treap structure for this secuence—and the order in which we insert these keys does not matter.

Solution: (do step by step)




Those students who did entered the keys one at a time into the treap often made the same error. After entering the first four keys they got the treap on the left. Then they inserted 3 by placing it between 5 and 1 to establish the heap order property and put 1 on the left of 3, as shown on the right.



```
2 (0.1)              4 (0.2)
                     /
                  2 (0.1)
```

**Q:** Where did they go wrong?

**A:** They needed to split the subtree at rooted 1 by the key 3 to get the left and right subtrees of 3.

## 2.1 Analysis – Subtree Size

In our analysis of the expected depth of a key in a Treap, we made use of the following indicator random variable

$$A_{i,j} = \begin{cases} 1 & j \text{ is an ancestor of } i \\ 0 & \text{otherwise} \end{cases}$$

Write an expression for $S_i$—the size of a subtree rooted at key $i$—in terms of $A_{i,j}$.

–

**Solution:**

$$S_i = \sum_{j=1}^{n} A_{j,i}$$

That is, we find all vertices that have $i$ as an ancestor and count them up.

**Q:** So what is the expected value of the subtree size?

**A:** Recall from class when discussing the depth of a Treap that $\mathbf{E}[A_{i,j}] = \frac{1}{|j-i|+1}$. Therefore, by linearity of expectation,

$$\begin{aligned} \mathbf{E}[S_i] &= \sum_{j=1}^{n} \mathbf{E}[A_{j,i}] = \sum_{j=1}^{n} \frac{1}{|i-j|+1} \\ &= \sum_{j=1}^{i} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= \sum_{k=1}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} = \boxed{H_i + (H_{n-i+1} - 1)} \end{aligned}$$

## 2.2 Analysis – Chernoff or not

Unfortunlatey, most people got this wrong.

**Q:** Does the Chernoff bound imply that the size of the subtree rooted at key $i$ is within a constant factor of $\mathbf{E}[S_i]$ with high probability? Justify your answer.

Before we answer this question, let's try to remember the preconditions for the Chernoff bounds:

**Q:** What is the requirement of using Chernoff bounds?

**A:**

**Theorem 2.1** (Chernoff Bound)**.** *Given a set of independent random variables $X_1, \ldots, X_n$, each taking a real value between 0 and 1, with $X = \sum_{i=1}^{n} X_i$ and $\mathbf{E}[X] = \mu$, then for any $\lambda > 0$*

$$\mathbf{Pr}[X > (1+\lambda)\mu] \quad < \quad \exp\left(\frac{-\mu\lambda^2}{2+\lambda}\right).$$

Notice that one important requirement is that the variables $X_i$'s that we sum over is mutually independent. *But what does it even mean for these variables to be mutually independent?* Remember that in 15-251, we said the two events $A$ and $B$ are independent iff. $\mathbf{Pr}[A \wedge B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. Generalizing this definition, we have that two random variables $X$ and $Y$ are independent if and only if for all possibile values $x$ and $y$ that these RVs can take on,

$$\mathbf{Pr}[X = x \wedge Y = y] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y].$$

Generalizing this even further, we have:

The random variables $X_1, \ldots, X_n$ are *mutually independent* if and only if *for all subset $S \subseteq \{1, \ldots, n\}$, the following holds*

$$\mathbf{Pr}\left[\bigwedge_{i \in S} X_i = x_i\right] = \prod_{i \in S} \mathbf{Pr}[X_i = x_i]$$

*for all settings for $x_i$.*

Back to the exam question: Clearly, the indicator RVs $A_{i,j}$'s take on either 0 or 1, so they must be between 0 and 1. This means all that we've left to check is whether the random variables $A_{j,i}$, $j = 1, \ldots, n$ (in the expression of $S_i$) are mutually independent.

It might not be immediately clear whether these are independent RVs or not. But to convince yourself that the high-probability bound cannot possibly hold, let's look at the key at the root of the Treap, say that key is $i^*$. What happens is that the subtree size of this particular key is $n$—it is the whole Treap. For this particular key, with probability 1, $S_{i^*} = n$; this stands in stark constrast with the claim that with high probability $S_{i^*} \in O(\log n)$.

You should convince yourself that the $A_{j,i}$ for $j = 1, \ldots, n$ are not independent. For intuition, the following observation should convince you that the RVs in the summands are **NOT** independent: Suppose you split the sum in two halves; if there were a lot of 1's in the left half, then it is likely that there are many ones in the right sum—because in that case, the node $i$ is a node high up in the tree.

On the other hand, a few lectures ago, we said that the depth of each key $i$ is at most $O(\log n)$ with high probability. What's going on? Let $D_i$ be the depth of the key $i$, so as we argued in class,

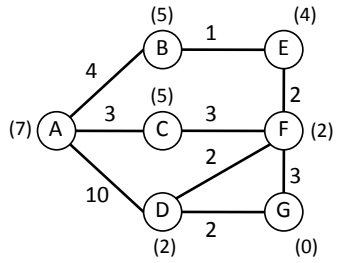
$$D_i = \sum_{j=1}^{n} A_{i,j} \implies \mathbf{E}[D_i] = \sum_{j=1}^{n} \mathbf{E}[A_{i,j}] \leq 2H_n.$$

To get a high probability bound for $D_i$, we need to argue that $A_{i,j}$ for $j = 1, \ldots, n$ are mutually independent. **Exercise:** Show this formally using the definition.

# 3 More Exam Debriefing: Dijkstra and A-star

Dijkstra's algorithm is a fundamental algorithm that we hope to help you get a firm grasp of it before moving on to other classes. We'll review Dijkstra again and contrast it with $A^*$. We take inspiration from misconceptions we

see from the exam's answers. To be concrete, we'll step through Dijkstra's and $A^*$ on the following graph from the exam:




**Q:** Can Dijkstra's algorithm loop over even if we have negative-weight edges?

**A:** If we look closely at the algorithm, we'll realize that Dijkstra does not visit each vertex more than once. In fact, this is a defining characteristic of Dijkstra: It claims that the shortest-path distance to a node $v$ is the distance it gets when it first visits $v$ (i.e., when $v$ is the result of a deleteMin)—and it declares this distance final and never visits this vertex again.

This is different from, for example, Bellman-Ford, which can "visit" a vertex many times (although it has a different notion of visiting). Indeed, in Bellman-Ford, we can get into an infinite loop if we don't put a cap on the number of iterations and keep running as long as edge relaxation changes the distance.

We go now through the case in the exam (draw graph). The question was: what is the order of vertices that Dijkstra will visit if the source vertex of G.

**Q:** Interim question: how would you modify Dijsktra's algorithm to only find path from A to G?

**A:** Simply terminate when G is first visited. (There may still be unvisited vertices in the priority queue that the full Dijkstra algorithm would continue to visit.)

Let's start working on the Dijkstra on the blackboard.

**Q:** On each iteration, what two (2) data structures Dijkstra's algorithm maintains?

**A:** It will contain a shortest path tree (D) and the priority queue (PQ). Crucially: $D$ also defines a set of vertices which have been already visited. More about this soon.

**Q:** On each recursive call which vertex will it consider?

**A:** It will consider the top (min) element of the PQ, say vertex $u$, and delete it from Q.

**Q:** What will it do it with $u$ if $u$ is in $D$?

**A:** If $u$ is in $D$, then it will recurse (and next element from priority queue PQ is considered.).

**Q:** What if $u$ is not in $D$ yet:

**A:** All neighbors of $u$ are *relaxed*. This means: it will consider every neighbor $v$ and the weight of the edge to the neighbor $w \in W(u, v)$ and add the neighbor to the priority queue with priority **Q:** what priority? **A:** with priority $d + w$ where $d$ is the priority of $u$ when it was popped from the priority queue.

At last, $u$ is added to $D$ and dijkstra is recursed.

**Q:** Why does it not work properly with negative edges (in general)?

(Note, SML code for dijkstra can be found from lecture 9).

## 3.1 Dijkstra on the exam graph

| step | visits | D after | PQ after | notes |
|---|---|---|---|---|
| init | | | (A,0) | |
| 2 | A | $(A->0)$ | (D,10), (B,4), (C,3), | |
| 3 | C | $(A,C->3)$ | (D,10), (F,3+3=6), (B,4) | (added A not shown) |
| 4 | B | $(A,C,B->4)$ | (D,10), (F,6), (E,4+1=5) | |
| 5 | E | $(A,C,B,E->5)$ | (D,10), (F,6) | (added F not shown) |
| 6 | F | $(A,C,B,E,F->6)$ | (G, 3+3+3=9), (D,3+3+2=8) | (D got downgraded) |
| 7 | D | $(A,C,B,E,F,D->8)$ | (G,9) | |
| 8 | G | $(A,C,B,E,F,D,G->9)$ | empty | |

**Q:** How can we use $D$ to find shortest path from A to G?

**A:** Backtrack. First choose the neighbor of G which has the shortest distance from A and recurse.

## 3.2 When to choose A-star over Dijsktra?

**A:** A-star can be used only for looking for one path. It is better than Dijkstra when the heuristic is helpful as it can greatly limit the number of vertices being checked. This is particularly true on graphs with a lot of structure (e.g. grids).