Augmented Binary Search Trees

1 Overview

In the last few lectures we presented the binary search tree data structure. One of its uses is to store sets of ordered items, as well as map them to values, since the nodes of a binary tree are well-suited to storing key-value pairs. The data structure keeps the keys in sorted order, and thus supports order-based operations such as `split`, `first`, `last`, and `join`. We also discussed balancing schemes, focusing specifically on treaps. Now, we turn our attention to ways in which their functionality can be enhanced.

2 Incorporating a Combining Function into BSTs

Our first generalization will be to imbue the values stored in the binary search trees with an associative function, allowing the tree to always automatically have reduced values on hand. It will be called an **Augmented Binary Search Tree**.

When the augmented BST is created, an associative combining function, along with its identity is specified[1]. We denote the combining function by `combine : (V,V) -> V`, and its corresponding identity value as `identity : V`.

The nodes of the augmented BST are defined here:

Definition: Augmented Binary Search Tree

```
type AugBST<K : Ordered, V> =
    Empty
  | Node { left: AugBST<K,V>,
           key: K,
           value: V,
           reduced: V,    // new!
           size: int,
           right: AugBST<K,V>  }
```

The only change from the earlier definition is that we have included a new field called `reduced` of type `V`. It will contain the result of applying the combine function `combine` to the sequence of `V` values represented by this node of the tree in increasing order by key. (See the examples later in this lecture.)

[1] The way this is specified in code is very programming language dependent, and will not be discussed in this lecture. When defining an augmented BST in pseudocode, we will therefore specify the combine function and identity in writing or as a comment.

The only other change to our previous definition of BSTs is in the `makeNode` function. This is the function that is used to create all nodes of the BST. And it is where all the fields of a new BST node are computed. Here is our new version, along with some functions that it depends on.

```
fun makeNode(L: AugBST<K,V>, k: K, v: V, R: AugBST<K,V>) -> AugBST<K,V>:
  return Node(L, k, v,
              combine(combine(reduced_val(L),v),reduced_val(R)),
              size(L)+1+size(R), R)

fun reduced_val(T: AugBST<K,V>) -> V:
  match T with:
    case Empty: return identity
    case Node(_,_,_,reduced,_,_): return reduced

fun size(T: AugBST<K,V>) -> int:
  match T with:
    case Empty: return 0
    case Node(_,_,_,_,s,_): return s
```

As you can see, the way in which the `reduced` is computed is perfectly analogous to the way in which the `size` is computed. The only difference is that the user-specified combine function is used instead of addition, and that the identity element `identity` is used instead of `0`.

This inductively maintains the invariant that the `reduced` at a node n is the result of applying the combining function to all the values in the subtree rooted at n in left-to-right order. (The order is determined by the keys.)

These are all the changes in the code that are required to implement augmented binary search trees. All of the other functions we developed for BSTs (`insert`, `delete`, `split`, `union`, `join`, etc.) now support augmented values, with no changes.

2.1 An Example

Suppose I had a sequence of numbers $[v_0, v_1, \ldots, v_{n-1}]$. I want to be able to quickly handle queries of the form `findMax`(i, j), which returns the maximum of $v_i, \ldots, v_j$. To do this, create a binary search tree where the combining function is `combine`$(a, b) = \max(a, b)$ and the identity `identity`$= -\infty$.

Then I create a BST `T` by inserting the key-value pairs $(0, v_0), \ldots, (n-1, v_{n-1})$ into an initially empty tree. The following code suffices to handle the queries.

```
fun findMax(T: BST<int,int>, i : int, j : int) -> int:
  match split(T, i) with:
    case (_, SOME(v), R):
      if (i == j) then return v
      match split(R,j-i-1) with:
        case(L, some(v'), _):
          return max(v, max(getRV(L), v'))
```

This is nice, but it's just the beginning of what can be done. Note that splits and joins are not supported on this data structure because the keys are specified in a very inflexible way. In

particular, keys are tied to fixed indices; they do not support efficient rank-based changes.

For example, if I wanted to insert a new element at position (rank) 0, I would have to add one to every key in the tree, which would cost $O(n)$ work! We will see a way to get around this in the next section.

3 Rank-based Operations on BSTs

Let us number the elements in a BST of n nodes from least to greatest with the numbers $[0, 1, \ldots, n-1]$. These are the **ranks** of the nodes. So the rank of the least node in the tree is 0, and the rank of the rightmost node is $n-1$. This will give us an alternative (and very useful) way to access the nodes in the tree. This will give us the power that sequences have to access the ith element, while also allowing efficient splitting and joining and insertion and deletion.

The operations that depend on ranks will make use of the size fields that we store in all the nodes of the tree. The ranks, like the keys, increase as we traverse a tree from left to right.

With this in mind, we can implement some of the most important operations on BSTs using ranks instead of keys. The problem of determining the ranks of given keys, or the keys with a given rank is often called the *order statistics problem*, so a dynamic tree data structure supporting them is often called an *order statistics tree*.

Interface: Order Statistics Tree

Stores a set of key-value pairs (K, V) in sorted order and supports the operations of a binary search tree in addition to the following operations:

- `find_by_rank(T,i)`: return the key-value pair corresponding to the key with rank `i` in `T`. The rank `i` must be in $[0, |T|-1]$.
- `rank_of_key(T,k)`: Return the rank of the key k in T, i.e., the number of keys in T that are less than k
- `split_rank(T,i)`: returns `(L,k,v,R)` where `L` is the BST representing the first `i` elements of `T`, and `(k,v)` is the rank `i` element of `T`, and `R` contains the remaining elements of `T`.
- `join_rank(L,R)`: returns the BST representing the sequence of values represented by `L` followed by those represented by `R`. It does not make use of the keys. But to preserve the key-order property, all the keys in `L` must be less than all the keys in `R`.
- `assign_rank(T,i,v)`: Replaces the value of the element in `T` of rank `i` with `v`. Returns the new tree.
- `delete_rank(T,i)`: Delete the key-value pair with rank `i`, and return the new tree.

An order statistics tree can also be augmented with a combine function and identity value to enable support for quering for reduced values in constant time.

Algorithm: BST: split_rank

```
fun split_rank(T: BST<K,V>, i: int) -> (BST, k, V, BST):
  match T with:
    case Node(L,k,v,_,_,R):
      if i = size(L):
        return(L, k, v, R)
      else if i < size(L):
        (L', k', v', R') = split_rank(L, i)
        return (L', k', v', rebalance(makeNode(R', k, v, R)))
      else:
        (L', k', v', R') = split_rank(R, i-size(L)-1)
        return (rebalance(makeNode(L, k, v, L')), k', v', R')
```

Algorithm: BST: join_rank

```
fun join_rank(L: BST<K,V>, R: BST<K,V>) -> BST:
  if size(R) = 0:
    return L
  else:
    (_, k, v, R') = split_rank(R,0)
    return rebalance(makeNode(L,k,v,R'))
```

Algorithm: BST: assign_rank

```
fun assign_rank(T: BST<K,V>, i: int, v: V) -> BST:
  (L,k,_,R) = split_rank(T,i)
  return rebalance(makeNode(L, k, v, R))
```

Algorithm: BST: delete_rank

```
fun delete_rank(T: BST<K,V>, i: int) -> BST:
  (L,_,_,R) = split_rank(T,i)
  return join_rank(L,R)
```

Algorithm: BST: find_by_rank

```
fun find_by_rank(T: BST<K,V>, i: int) -> (K,V):
  match T with:
    case Node(L,k,v,_,_,R):
      if i < size(L): return find_by_rank(L, i)
      else if i = size(L): return (k,v)
      else: return find_by_rank(R, i-size(L)-1)
```

Algorithm: BST: rank_of_key

```
fun rank_of_key(T : BST<K,V>, k : K) -> int:
  match T with:
    case Empty: return 0
```

```
    case Node(L,k',_,_,R):
      if k == k': return size(L)
      else if k < k': return rank_of_key(L, k)
      else: return size(L) + 1 + rank_of_key(R, k)
```

3.1 An Example

Consider the following problem:

> A sequence of n distinct keys $k_0, k_1, \ldots, k_{n-1}$ is inserted into a set at times $0, \ldots, n-1$. After this you are confronted by a sequence of m queries of the form: `query(i,j)` which means: In the set, after the insertion at time i, return the key that had rank j. The goal is to give a solution to this problem that uses $O(n \log n)$ space and $O(\log n)$ time per query. You have to process all of the insertions before you are allowed to see any of the queries.

For example, the keys inserted are $7, 5, 9, 1, 2$. Then for `query(2,1)` the answer is 7.

The solution involves taking advantage of the persistence of BSTs, while also making use of order statistics operations. Here's the pseudo-code for processing the insertions:

```
t = [⊥ for _ in 0...n-1]
B = BST<int,int>::empty()  // An empty BST
for i in 0...n-1:
  B = insert(B, k_i, k_i)
  t[i] = B           // t[i] is the tree after inserting k_i
```

And here's the pseudo-code for answering the queries:

```
fun query(i,j):
  (answer,_) = find_by_rank(t[i], j)
  return answer
```

Each insertion only uses an additional $O(\log n)$ space. Each `find_rank` also does at most $O(\log n)$ work. Therefore this solution satisfies the required time and space bounds.

4 Dynamic Sequences

Notice that using rank-based operations means that we don't need the keys to navigate the tree. So you can see that it's possible, and in fact quite useful, to deploy BSTs with no keys in the nodes. In this case the BST just represents a sequence of values, in the same way that `sequences` do, while supporting more operations at the cost of being more expensive (operations cost $O(\log n)$ rather than $O(1)$).

An augmented BST with no keys is essentially a fully persistent dynamic sequence supporting O(log n) split, join, insert, delete, and update. All of these operations are $O(\log n)$ work on a tree with n nodes, and consume $O(\log n)$ additional space due to persistence. And they're all purely functional and persistent. Since the keys don't exist there is no longer any restriction on the use of `join_rank` to preserve the key-order.

Because these sequences are fully persistent, you can, for example, create a new sequence by joining a sequence to itself. Consider the following mind-blowing scenario. Start with a sequence of length one. Then join it to itself. Then join that one to itself, etc. Do it k times. You would end up with a sequence of length 2^k. The space used by the sequence is $O(k^2)$. You could then continue to operate on this huge sequence with splits, insertions, assignments, deletions etc. The cost would be $O(k)$ work and space per operation.

This key-less version of BSTs is not supported in either the SML or the C++ libraries.[2]

[2] An adjustment is needed to fit this into the framework of the previous lectures. The `rebalance` function as previously described uses the keys as a way to record the priorities of the nodes.

The actual way in which the SML implementation of treaps works is to generate a random priority for a node the moment you create the node, and store it in the node. So keys are not used. But this would fail in the case of joining a tree with itself, because the priorities would be identical.

Alternatively, AVL trees which keep the depth of the tree around, are also well suited for this.