# Lecture 5 Notes (February 24, 2025)

Sam Westrick

s.westrick@nyu.edu

## Sequential BSTs and Path Copying

The traditional "sequential" interface to a binary search tree structure typically consists of functions such as `insert`, `remove`, and `lookup`, for example as follows.

```
type key
val key_compare: key * key -> order  (* LESS, EQUAL, or GREATER *)
datatype tree = Leaf | Node of tree * key * tree
val insert: tree * key -> tree
val remove: tree * key -> tree
val lookup: tree * key -> bool  (* is the key in the tree? *)
```

### Path Copying By Default

In this particular presentation, updates on the tree return "new" trees as output, and the original tree is left unmodified. The actual mechanism we'll use to accomplish this is known as **path copying**, which falls out naturally from a purely functional programming style. All of the algorithms we develop in this lecture are based on this idea.

To illustrate, here an implementation of `insert` which (for now) completely ignores tree balance.

```
(* Note: this implementation ignores rebalancing *)
fun insert (t, x) =
  case t of
    Leaf => Node (Leaf, x, Leaf)
  | Node (l, k, r) =>
      case key_compare (x, k) of
        LESS => Node (insert (l, x), k, r)
      | EQUAL => t
      | GREATER => Node (l, k, insert (r, x))
```

This implementation walks down the tree to find a good place to insert a node, and then makes new `Node`s on its way back out. These new nodes are the "copied path", which mirrors the path that was traversed for the insertion.

For example, see the figure below. The initial tree, $T$, is drawn in black. Note that we don't draw the `Leaf` elements, only the `Node`, and we label each node with the key stored at that node. The keys in this case are integers. We then call `insert(T,5)`, and the new tree is drawn in red. The original tree $T$ is preserved, and the new tree shares many of its nodes with $T$. For a tree of height $h$, the cost of this insertion is only $O(h)$.

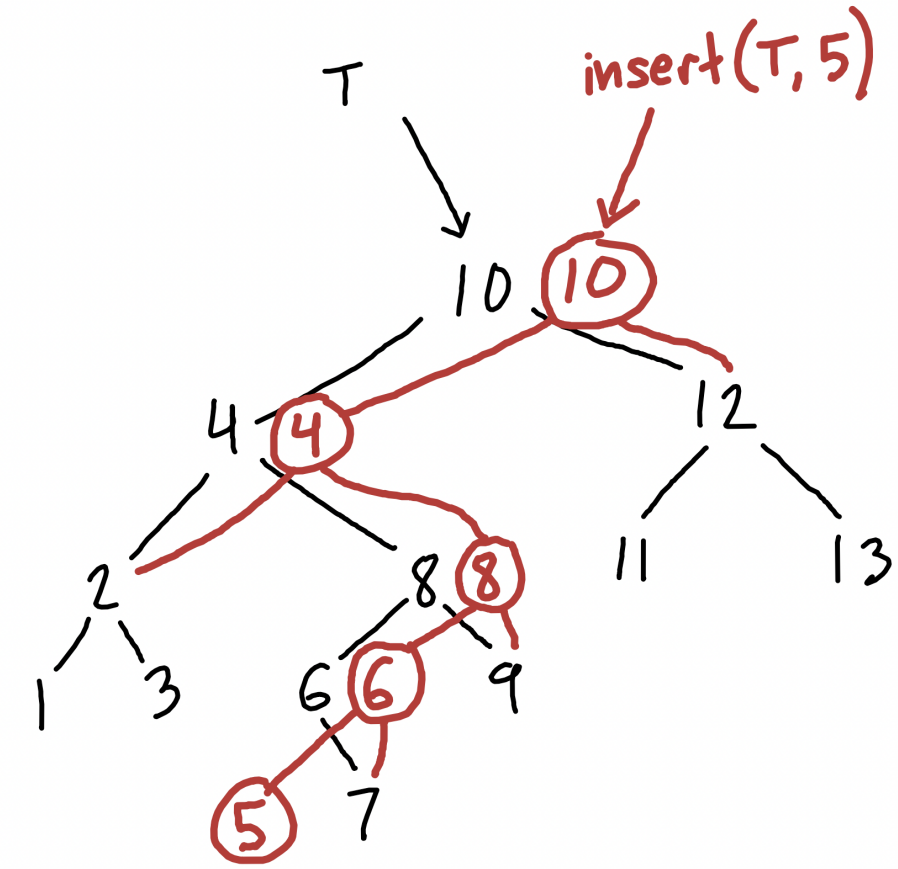


Figure 1: Example path copying insertion

### Sequential Bulk Operations

With functions such as `insert`, `remove`, and `lookup`, we can easily implement "bulk" operations such "union", "intersection", and "difference". The terminology is taken from sets, where we think of binary search trees as implementing a set of keys.

Below are sequential implementations of union, intersection, and difference. Each of these is implemented by walking through one tree and accumulating an output tree. Specifically, we use a sequential function, `iterate`, which walks through a tree and applies a function at each key to accumulate a final value.

```
fun sequential_union(t1, t2) =
  iterate t1 t2 (fn (tu, x) => insert (tu, x))
fun sequential_intersection(t1, t2) =
  iterate t1 Leaf (fn (ti, x) =>
    if lookup (t2, x) then insert (ti, x) else ti)
fun sequential_difference(t1, t2) =
  iterate t1 t1 (fn (td, x) =>
    if lookup (t2, x) then remove (td, x) else td)

(* sequential "iterator" over a tree t, with accumulator a *)
fun iterate (t: tree) (a: 'a) (f: 'a * key -> 'a) =
  case t of
    Leaf => a
  | Node (l, k, r) =>
      let
        val a' = iterate l a f
        val a'' = f(a', k)
        val a''' = iterate r a'' f
      in
        a'''
      end
```

These implementations are not parallel, and also not particularly work efficient: even with implementations of `insert` and `remove` which maintain balance, the total work for each of these implementations will be $O(m \log n)$ for two trees of size $n$ and $m$ where $n \geq m$. (You can always choose to iterate over the smaller tree, and perform lookups/insertions/removals on the larger tree.)

In this lecture, we will develop parallel versions of these bulk operations. The parallel code happens to also perform asymptotically less work than the sequential code described above.

## Splitting Trees

Many parallel algorithms are based on the idea of splitting something in half, to process two halves in parallel. In order to apply this idea to binary search trees,

we need a way to split a tree.

Of course, the `Node`s of a tree automatically suggest reasonable places to split. The balancing strategy for the tree will ensure that the height of the tree is asymptotically logarithmic, so if we simply process the two children of a `Node` in parallel, we end up with parallel traversals that are highly parallel: linear work (to touch a linear number of nodes), and logarithmic span.

For example, we could perform a parallel reduction over the tree. Here, we supply a function `f` which is applied to each key to produce one element of the reduction, and then these elements are combined via the associative function `g`.

```
(* Parallel reduction over a tree. Assuming the tree is balanced,
 * and that the work and span of `f` and `g` are constant, this
 * requires O(n) work and O(log n) span, where n is the number of keys.
 *)
fun reduce (g: 'a * 'a -> 'a) (z: 'a) (f: key -> 'a) (t: tree) : 'a =
  case t of
    Leaf => z
  | Node (l, k, r) =>
      let
        val (x, y) = ForkJoin.par (fn () => reduce g z f t,
                                   fn () => reduce g z f t)
        val m = f k
      in
        g(g(x,m),y)
      end
```

**An attempt at parallel union**. Let's run with the idea of taking advantage of natural splits in the tree to attempt to implement a bulk operation over two trees. Specifically, let's try `union`.

In the previous lecture, we saw an algorithm for divide-and-conquer merge of two sorted sequences. If we think of the in-order traversal of a binary search tree as implicitly defining a sorted sequence, perhaps we can reuse this algorithm to union two binary search trees in parallel.

The high-level algorithm for parallel merging was to use the median element of one sequence to split the other sequence; this makes it possible to perform two merges in parallel.

We can start running with a similar idea for tree union, but we immediately get stuck.

```
fun union(t1, t2) =
  case (t1, t2) of
    (Leaf, _) => t2
  | (_, Leaf) => t1
  | (Node (l1, k, r1), _) =>
      let
```

```
    (* Problem 1: what do we do here? Split t2 at key k? *)
    val (l2, r2) = ???
    val (l, r) =
      ForkJoin.par (fn () => union (l1, l2),
                    fn () => union (r1, r2))
  in
    (* Problem 2: what about rebalancing? *)
    Node (l, k, r)
  end
```

There are two problems to solve:

1. We'd like to split `t2` into two pieces `(l2, r2)` such that the key `k` lies between `l2` and `r2` in the sorted order. This would make it possible to perform two unions in parallel. However, there's no guarantee that the root of `t2` is equal to `k`. Perhaps the key `k` is buried deep in a subtree of `t2`, or perhaps it is not present in the tree at all.
2. After performing the unions in parallel, how do we stitch the results together? Creating `Node(l,k,r)` is correct in terms of preserving the "left-to-right" invariant of the binary search tree, but this result may not be balanced.

## Split and Join

The solution to our problems will be two key functions, named `split` and `join`, which are inverses of each other. The former lets us pull trees apart, and the second puts them back together again.

Let's begin with `split`. The function `split(t,k)` splits a tree `t` "at" a key `k`. It returns a tuple `(l,b,r)`, where:

- `l` is a tree containing all keys less than `k`
- `b` is a boolean indicating whether or not `k` was in the tree
- `r` is a tree containing all keys greater than `k`

Ignoring balance for a moment, let's implement this function.

```
fun split_no_rebalancing(t, k) =
  case t of
    Leaf => (Leaf, false, Leaf)
  | Node(l, k', r) =>
      case key_compare (k, k') of
        EQUAL => (l, true, r)
      | LESS =>
          let val (ll, b, lr) = split_no_rebalancing(l, k)
          in (ll, b, Node(lr, k', r))
          end
      | GREATER =>
          let val (rl, b, rr) = split_no_rebalancing(r, k)
```

```
    in (Node(l, k', rl), b, rr)
    end
```

The code is remarkably similar to insertion, except we keep track of which elements were less and greater than the key `k`. Again, we utilize the path copying idea to construct new nodes on the way back out. For a tree of height $h$, this requires at most $O(h)$ work.

It will be helpful to visualize an execution on an actual tree. The figure below shows the before and after of splitting a tree $T$ at key 6.

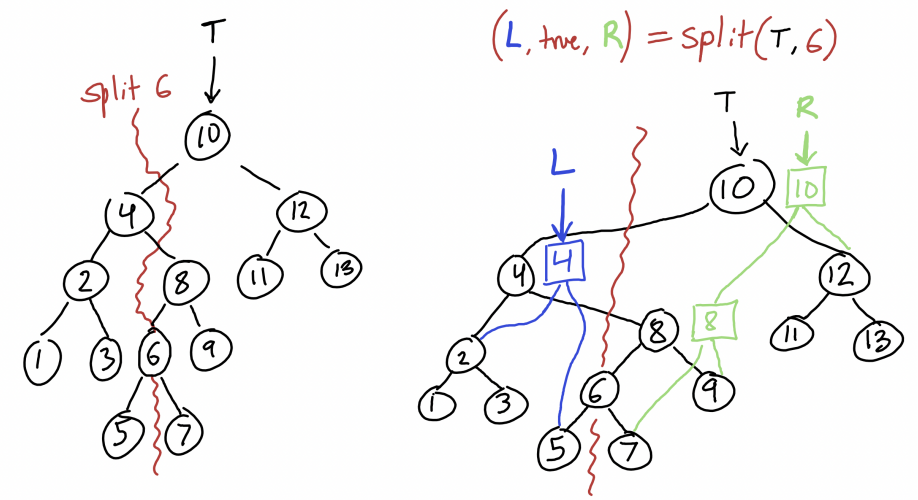


Figure 2: Example split (before and after)

### Restoring Balance with Join

The above code for splitting does not attempt to restore balance of the output trees. This can be remedied with a corresponding function called `join` which stitches two balanced trees together, ensuring that the final tree is also balanced.

```
join: tree * key * tree -> tree
```

When we call `join(l,k,r)`, we have a few requirements:

1. **Balance requirement**: `l` and `r` must each be already balanced.
2. **Order requirement**: the maximum key of `l` must be smaller than `k`, and minimum key of `r` must be greater than `k`, i.e., $\max(\texttt{l}) < \texttt{k} < \min(\texttt{r})$.

Assuming these requirements are met of the input, `join` will guarantee that the output is balanced, with keys that are appropriately ordered.

For the implementation of `join`, we refer the reader to the paper Just Join for Parallel Ordered Sets which describes four possible implementations, based

on four popular balancing criteria: red-black, AVL, weight-balanced trees, and treaps.

The surprising result from this paper is that the cost of join is (essentially) $O(\log n - \log m)$, where $n$ is the size (number of keys) of the bigger tree and $m$ is the size of the smaller tree.

(The specifics are a bit trickier than this, in particular requiring a criterion-specific notion of "rank" which takes the place of the logarithm in this bound. The rank of a tree is closely related to its height, but the specifics differ across balancing criteria.)

Using `join`, we can implement `split` in a way that guarantees that the results are balanced. Essentially, we just replace each `Node` constructor with a call to `join`.

```
fun split(t, k) =
  case t of
    Leaf => (Leaf, false, Leaf)
  | Node (l, k', r) =>
      case key_compare (k, k') of
        EQUAL => (l, true, r)
      | LESS =>
          let val (ll, b, lr) = split(l, k)
          in (ll, b, join(lr, k', r))
          end
      | GREATER =>
          let val (rl, b, rr) = split(r, k)
          in (join(l, k', rl), b, rr)
          end
```

In the Just Join paper, they prove that this particular implementation of `split` requires at most $O(\log n)$ work and span for all of the balancing criteria considered.

With `split` and `join`, we can easily implement our previous "sequential" interface, all with $O(\log n)$ work and span.

```
fun lookup(t,k) =
  let val (_, b, _) = split(t,k) in b end
fun insert(t,k) =
  let val (l, _, r) = split(t,k) in join(l,k,r) end
fun remove(t,k) =
  let val (l, _, r) = split(t,k) in join2(l,r) end
```

## Parallel Union, Intersection, and Difference with Split and Join

With `split` and `join`, we are now ready to finish implementing the `union` function. We can also similarly implement `intersection` and `difference`, which utilize a variant of `join(l,k,r)` called `join2(l,r)` which does not require a key in the middle (see the Just Join paper for more details.)

**A key result from the Just Join paper is that all of the following implementations have $O(m \log(1 + n/m))$ work and $O(\log n \log m)$ span, where $n$ is the size of the larger tree and $m$ is the size of the smaller tree.** Note that the work is asymptotically smaller than the sequential algorithms we described at the beginning of lecture.

```
fun union(t1, t2) =
  case (t1, t2) of
    (Leaf, _) => t2
  | (_, Leaf) => t1
  | (Node (l1, k, r1), _) =>
      let
        val (l2, _, r2) = split(t2, k)
        val (l, r) =
          ForkJoin.par (fn () => union(l1, l2),
                        fn () => union(r1, r2))
      in
        join(l, k, r)
      end

fun intersection(t1, t2) =
  case (t1, t2) of
    (Leaf, _) => Leaf
  | (_, Leaf) => Leaf
  | (Node (l1, k, r1), _) =>
      let
        val (l2, b, r2) = split(t2, k)
        val (l, r) =
          ForkJoin.par (fn () => intersection(l1, l2),
                        fn () => intersection(r1, r2))
      in
        if b then join(l, k, r) else join2(l, r)
      end

fun difference(t1, t2) =
  case (t1, t2) of
    (Leaf, _) => Leaf
  | (_, Leaf) => t1
  | (Node (l1, k, r1), _) =>
      let
        val (l2, b, r2) = split(t2, k)
        val (l, r) =
          ForkJoin.par (fn () => difference(l1, l2),
                        fn () => difference(r1, r2))
      in
        if b then join2(l, r) else join(l, k, r)
      end
```

## Key-Value Maps

Often, rather than storing only a key at each node, we also want to associate a value with each key. This is straightforward: we just store the values in each node. Lookup on such a structure typically returns the value associated with a key. Similarly, `split(t,k)` can return the value associated with key `k`. To handle the cases where the key is not in the tree, we can use the `option` type.

```
type key
type value
type tree = Leaf | Node of tree * key * value * tree

val lookup: tree * key -> value option
val split: tree * key -> tree * (value option) * tree
val join: tree * key * value * tree -> tree
val join2: tree * tree -> tree
```

As an excercise, we recommend adapting algorithms such as `union`, `intersection`, and `difference` to key-value maps. (An interesting question is what exactly to do with the values, and there are many options here in the design space.)

## Augmentation

We've seen before that it easy to store information about every subtree within nodes, such as (for example) the size of the subtree. The idea is to be careful to correctly compute the size at the moment a `Node` is constructed:

```
datatype tree = Leaf | Node of int * tree * key * tree
fun size Leaf = 0
  | size (Node (n, _, _, _)) = n

(* Instead of using `Node` directly, call this instead, and the
 * sizes will always be correct. *)
fun make_node(l,k,r) =
  Node(size l + size r + 1, l, k, r)
```

We can take this idea further to store all kinds of interesting data at each node. In particular, we can **augment** the tree with an associative combining function, `g`, similar to the combining function of a parallel reduction, and use this function to compute arbitrary interesting properties of all subtrees.

```
type key
type value
type avalue  (* augmented values *)

val f: (key * value) -> avalue     (* singleton avalues, from key-value pairs *)
val g: avalue * avalue -> avalue   (* associative combining *)
val z: avalue                      (* identity for g *)
```

```
datatype tree = Leaf | Node of tree * key * value * avalue * tree

fun aval Leaf = z
  | aval (Node (_, _, _, a, _)) = a

(* Instead of using `Node` directly, call this instead, and the
 * augmented values will always be correct. *)
fun make_node(l,k,v,r) =
  let val a = g(g(aval(l), f(k,v)), aval(r))
  in Node(l, k, v, a, r)
  end
```

For example, we can augment nodes with the largest value within each subtree:

```
type value = int
type avalue = int
fun f(k, v) = v
fun g(a1, a2) = Int.max (a1, a2)
val z = valOf Int.minInt
```

Now, imagine we reimplement all of our tree algorithms described earlier in lecture, using this augmentation. Now we can efficiently query the augmented value for ranges of keys:

```
(* compute the augmented value only for keys k in the range k1 < k < k2 *)
fun query_range(t, k1, k2) =
  let
    val (_, _, t') = split(t, k1)
    val (t'', _, _) = split(t', k2)
  in
    aval(t'')
  end
```

This would tell us the maximum value $v$ across all $(k, v)$ pairs where $k_1 < k < k_2$. The cost of the query is only two splits, which comes out to only $O(\log n)$ work and span. Many such queries can be safely executed in parallel.

**As long as the functions `f: key * value -> avalue` and `g: avalue * avalue -> avalue` require only $O(1)$ work and span, then augmentation has no affect on the asymptotic cost of any purely functional tree algorithm.** Intuitively, augmentation only slightly increases the (constant) cost of constructing a node.