Binary Search Trees

Binary search trees (BSTs) are a versatile data structure for storing sorted data. In particular, they can be used to implement the *sorted dictionary* and *sorted set* interfaces introduced in the previous lecture. Recall that these interfaces specify *what operations* a data structure supports, without prescribing *how* those operations are implemented. Binary search trees are one possible implementation, and their recursive structure also lends itself well to parallel algorithms.

However, the efficiency of a BST depends heavily on its shape. With n keys in a balanced BST, the height is $O(\log n)$, allowing efficient searching and updates. But the same keys can be arranged in a completely unbalanced BST of height $\Theta(n)$.

To address this problem, computer scientists have developed various *balancing schemes*, such as AVL trees, red-black trees, and treaps. These maintain approximately balanced trees, ensuring low height and therefore efficient algorithms on them. In this lecture, we will develop a *generic framework* for implementing BST algorithms that works independently of the particular balancing scheme. This means that our algorithms are correct and efficient for AVL Trees, Red-Black Trees, and more, without having to design and analyze separate version of them.

Then, in the next lecture, we will study treaps, which use randomization to maintain balance.

1 Binary Trees and Generic Algorithms

You have already encountered several kinds of binary search trees, such as AVL trees and red-black trees. Although these data structures use different balancing schemes, many of the algorithms we write on them are essentially identical.

For example, consider searching for a particular key in a BST. We compare the target key with the key at the root, and then recursively search either the left or right subtree. This algorithm does not really care whether the tree is an AVL tree, a red-black tree, or a treap. However, these data structures have different internal representations, so a naive approach would require us to write separate versions of the same algorithm for each one.

For example, consider writing an algorithm for checking whether a particular key is contained in a Red-Black tree. We might naturally write the following code:

```
type Color = RED | BLACK

type RBTree<K> =
    Empty
  | Node { left : RBTree,
           k : K,
           color : Color,
           right : RBTree }
```

```
fun rb_contains(T: RBTree<K>, k: K) -> bool:
  match T with:
    case Empty: return False
    case Node(L, k', _, R):
      if k < k': return rb_contains(L, k)
      else if k > k': return rb_contains(R, k)
      else: return True
```

Then, we are told that we also need to support `contains` on an AVL Tree:

```
type AVLTree<K> =
    Empty
  | Node { left : AVLTree,
           k : K,
           size : int,
           height : int,
           right : AVLTree }
```

```
fun avl_contains(T: AVLTree<K>, k: K) -> bool:
  match T with:
    case Empty: return False
    case Node(L, k', _, _, R):
      if k < k': return avl_contains(L, k)
      else if k > k': return avl_contains(R, k)
      else: return True
```

At which point we realize that we have essentially written the exact same code *twice*. The only difference was the *stuff that didn't matter*, the internal balancing data that the trees hold. Even though our algorithm only needs to know the key and the left and right children, it had to deal with the internal representation being different.

This motivates the idea of **generic algorithms**: algorithms that works for any type satisfying a specified interface. Rather than depending on the concrete representation of a tree, our algorithms should depend only on the operations they actually need.

1.1 The Binary Tree Interface

Looking at the previous examples, our search algorithm only needs to know whether the tree is empty, and if not, what element is stored at its root and what its left and right subtrees are. It does not need to know anything about the internal balancing scheme, such as the colors of red-black tree nodes or the heights of AVL tree nodes. To make our algorithms generic, we can capture these requirements by specifying an *interface* for binary trees.

Interface: Generic Binary Tree Interface

A type `Tree` satisfies the interface `BinaryTree<Elem>`, where `Elem` is the type of elements stored in the tree, if it supports:

- `size(T : Tree) -> int`:
  return the number of elements in T
- `expose(T : Tree) -> optional<NodeView<Tree>>`:
  return `NONE` if `T` is empty, or a *view* of its root element and left and right subtrees
- `empty : Tree`:
  an empty tree
- `join_mid(L : Tree, e : Elem, R : Tree) -> Tree`:
  construct a tree containing the elements of L, e, and the elements of R, in that order

where `NodeView` is defined as:

```
type NodeView<Tree : BinaryTree<Elem>> = {
  left: Tree,
  e: Elem,
  right: Tree
}
```

There are two kinds of operations in this interface:

- **Reading a tree:** `size` and `expose` allow us to inspect its logical structure.
- **Constructing a tree:** `empty` and `join_mid` allow us to create new trees.

Importantly, *both abstract away implementation details* (e.g., balancing). `expose` doesn't return them, and `join_mid` doesn't ask for them! It's all kept secret inside the black box.

Hence, it is important to emphasize that `join_mid` is *entirely responsible* for balancing. This means `join_mid(L, e, R)` does *not* necessarily produce a tree where e is the root. The result of `join_mid` is equivalent to creating a tree with e as the root whose children and L and R and **then rebalancing that tree to restore invariants**.

Here is an implementation of `contains` for BSTs using our generic interface:

```
fun contains(T : BinaryTree<K>, k : K) -> bool:
  match expose(T) with:
    case NONE: return False
    case SOME(NodeView(L, k', R)):
      if k < k': return contains(L, k)
      else if k > k': return contains(R, k)
      else: return True
```

1.2 Implementing the Binary Tree Interface

To use our generic algorithms with a particular binary tree representation, we only need to implement the four operations required by the `BinaryTree<Elem>` interface. For example, consider a simple *unbalanced* binary tree representation:

```
type MyTree<Elem> =
    Empty
  | Node { left: MyTree,
           e: Elem,
           size: int,
           right: MyTree }

empty: MyTree<Elem> = Empty

fun size(T: MyTree<Elem>) -> int:
  match T with:
    case Empty: return 0
    case Node(_, _, s, _): return s

fun join_mid(L: MyTree<Elem>, e: Elem, R: MyTree<Elem>) -> MyTree<Elem>:
  return Node(L, e, 1 + size(L) + size(R), R)

fun expose(T: MyTree<Elem>) -> optional<NodeView<MyTree<Elem>>>:
  match T with:
    case Empty: return NONE
    case Node(L, e, _, R):
      return SOME(NodeView(L, e, R))
```

Notice that `expose` *discards the stored size field*, returning only the element and its left and right subtrees. Conversely, `join_mid` correctly computes the size field when constructing a new node. This is what allows other algorithms to use this tree type generically. They get only what they need to traverse and construct new trees, both without ever knowing what other metadata or invariants might be involved behind the scenes.

For an unbalanced binary tree, `join_mid` is just one line of code. For balanced trees, however, it can require dozens of lines of complicated balancing logic: roughly 30-40 lines for AVL trees and Red-Black trees. However, the payoff is that we only have to write it once, and can then write all of our other algorithms without ever touching the balancing code ever again. If that sounds like too many lines of code, next lecture we will see **Treaps**, a binary search tree that uses **randomization** for balancing, whose `join_mid` is under 10 lines of code!

2 Binary Search Trees

An important observation is that our generic `BinaryTree<Elem>` interface does not require the elements to be sorted, or even to have an ordering! It simply describes how to inspect and construct binary trees, regardless of what their elements represent.

In fact, balanced binary trees can be used to implement data structures other than binary search trees. We will see an example in a few lectures, where we use balanced trees to represent sequences supporting efficient splitting and concatenation.

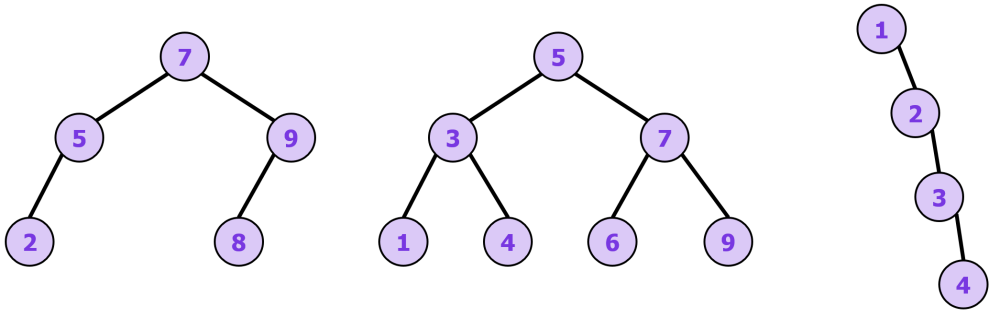
For now, however, we will focus on using binary trees to implement dictionaries and sets. A **binary search tree** (BST) is a binary tree whose elements are arranged according to their keys.

Definition: BST Invariant

A binary tree with root key k and left and right subtrees L, R satisfies the BST invariant if:

- Every key in L is less than k,
- Every key in R is greater than k,
- Both L and R satisfy the BST invariant.

Here are three examples of binary search trees. Note that being a binary search tree doesn't necessarily entail being balanced (e.g., the rightmost tree).



Our generic binary tree interface stores elements of an arbitrary type `Elem`. For binary search trees, we can choose to store either ordered keys (to implement a sorted set) or key-value pairs ordered by their keys (to implement a sorted dictionary). We will focus on the key-value version.

Definition: Binary Search Tree Interface

A type `Tree` satisfies the interface `BST<K,V>` if it satisfies `BinaryTree<(K,V)>`, where `K` is totally ordered, and all of its trees satisfy the BST invariant.

We will focus on BSTs that store key-value pairs, since this allows us to directly implement sorted dictionaries. However, this choice is somewhat arbitrary: we could just as easily focus on BSTs that store only keys, and use them to implement sorted sets. In fact, either representation can be used to implement both sets and dictionaries.

Remark: Two Ways to Implement Sets and Dictionaries

Sets and dictionaries are closely related, so it is natural for their implementations to share the same underlying data structure. There are two common ways to do this:

- **Dictionary as a set of key-value pairs.** Implement a dictionary as a set of pairs (k, v), where the ordering compares only the keys k, ignoring the values. The same tree algorithms can then support both sets of keys and dictionaries of key-value pairs.

  This is essentially the approach used by C++ standard library implementations such as libstdc++ and libc++, which share generic red-black tree machinery between `std::set` and `std::map`. C#'s `SortedDictionary` similarly uses an underlying tree-based set of key-value pairs.

- **Set as a dictionary with empty values.** Implement a set of keys of type K as a dictionary mapping each key to a trivial value, such as `Unit`. For example, Java's `TreeSet` is backed by `TreeMap`, and Rust's `BTreeSet` reuses `BTreeMap` with a zero-sized value type.

Both approaches allow sets and dictionaries to share essentially all of their underlying algorithms. The choice is primarily one of implementation architecture, not asymptotic efficiency. Importantly, the public interfaces of sets and dictionaries remain distinct.

3 Implementing Dictionaries Using BSTs

In this course, we have chosen to distinguish between two interfaces that serve different purposes. The `SortedDict<K,V>` interface specifies the operations we want a dictionary to support, such as `insert`, `find`, and `delete`. Our `BST<K,V>` interface, on the other hand, specifies the structural operations supported by a binary search tree, such as `expose` and `join_mid`.

Although both are interfaces, they play different roles: **a dictionary describes what we want to implement, while a BST describes how we can implement it.** We can use a BST internally to implement a dictionary, without making the BST's structural operations part of the dictionary interface. This separation is a design choice we make; other libraries and textbooks may organize these interfaces differently, but the underlying algorithms are essentially the same.

Our implementation approach. We will now implement dictionary operations using our generic BST interface. This is not just a theoretical abstraction; it is how our actual 15-210 lab libraries are implemented in both SML and C++!

In SML, we use a *functor* that takes a BST implementation (any structure satisfying the `BST` signature) and produces a sorted dictionary implementation. In C++, we similarly use a *class template* taking any BST implementation (any type satisfying the `bst` concept). In both cases, the dictionary is represented internally by a BST and its operations are implemented as algorithms on that tree, while the underlying BST is hidden from users of the dictionary[1].

We will therefore write our algorithms directly on BSTs, just as the implementations do. These algorithms implement the dictionary interface, but work with any BST satisfying our generic interface, regardless of its balancing scheme.

As a reminder, all of our algorithms on BSTs are going to be **persistent**:

Definition: Persistent Data Structures

A persistent data structure preserves previous versions when modified. Operations like `insert` and `remove` return **new trees** while sharing unchanged subtrees with the original. This allows multiple versions to coexist efficiently.

3.1 Search Operations

We begin with some of the simplest but arguably most important operations, since they form the namesake of binary *search* trees, searching!

3.1.1 Find

Let us start with `find`, one of the basic operations of the dictionary interface. Given a key k, it returns the corresponding value v if a key-value pair (k, v) exists in the dictionary, otherwise returning `NONE` if the key is not present.

The algorithm follows the familiar binary search tree search procedure. At each node, we compare the target key with the key at the root. If they match, we return the corresponding value. Otherwise, we recursively search the left or right subtree, depending on the ordering.

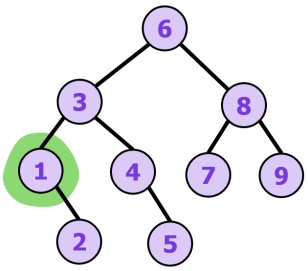
Algorithm: Dictionary: Find (BST Implementation)

```
fun find(T: BST<K,V>, k: K) -> option<V>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k',v), R)):
      if k < k':      return find(L, k)
      else if k > k': return find(R, k)
      else:           return SOME(v)
```

[1] In SML, this is through opaque signature ascription (`:>`); in C++, by the tree being a private data member.

3.1.2 First and Last

The `SortedDict<K,V>` interface also supports order-based operations. For example, `first` returns the key-value pair with the smallest key, or `NONE` if the dictionary is empty. Symmetrically, `last` returns the key-value pair with the greatest key.



By the BST invariant, the smallest key must be the leftmost element of the tree. We therefore follow left subtrees until we reach a node whose left subtree is empty.

Algorithm: Dictionary: First (BST Implementation)

```
fun first(T: BST<K,V>) -> option<(K,V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k,v), R)):
      match first(L) with:
        case SOME(e): return e
        case NONE:    return SOME((k,v))
```

3.1.3 Predecessor and Successor

The `prev` and `next` operations find the predecessor and successor of a given key k: the key-value pairs with the greatest key less than k, and the least key greater than k, respectively.

The `prev` algorithm works similarly to `first`; it follows a path down the tree, and if the current root key k' is at least k, it searches the left subtree. Otherwise, the root is a potential answer, but there might be a better one in the right subtree. It must therefore search the right subtree first, returning its answer if one exists, or the root otherwise. The `next` operation works symmetrically.

Algorithm: Dictionary: First (BST Implementation)

```
fun prev(T: BST<K,V>, k: K) -> option<(K,V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k',v), R)):
      if k <= k': return prev(L, k)
      else:
        match prev(R, k) with:
          case SOME(e): return SOME(e)
          case NONE:    return SOME((k',v))
```

3.2 Persistent Updates

We now turn to operations that modify the contents of a dictionary. Recall that our BST algorithms are *persistent*: they return new trees representing the updated contents, leaving the original trees unchanged.

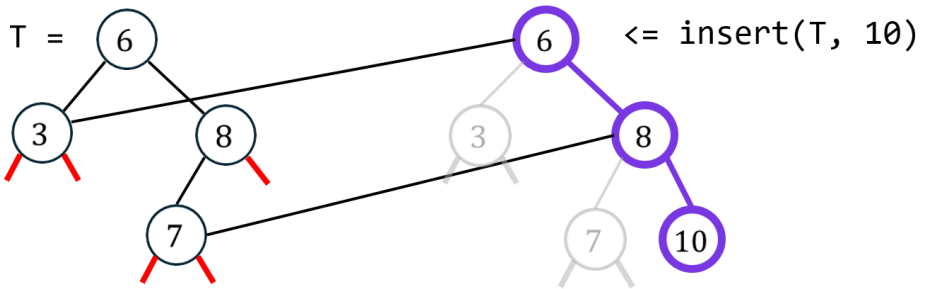
3.2.1 Insert

`insert` follows the same search procedure as `find`. It recursively searches for the location where the key belongs, and then constructs a new tree containing the inserted key-value pair.

Algorithm: Dictionary: Insert (BST Implementation)

```
fun insert(T: BST<K,V>, k: K, v: V) -> BST<K,V>:
  match expose(T) with:
    case NONE:
      return join_mid(empty, (k,v), empty)
    case SOME(NodeView(L, (k',v'), R)):
      if k < k':      return join_mid(insert(L, k, v), (k',v'), R)
      else if k > k': return join_mid(L, (k',v'), insert(R, k, v))
      else:           return join_mid(L, (k,v), R)
```

The idea is that each recursive call creates a new tree using `join_mid`. Importantly, note that at each recursive call, at least one of the subtrees L or R is *reused*. This allows unchanged subtrees to be shared with the old tree, rather than copying the entire data structure!



Also observe that the algorithm contains *no balancing logic whatsoever*! All of that complexity is hidden inside `join_mid`. Lastly, since the original subtrees are never modified, the operation is persistent. The new tree shares subtrees with the old, but never modifies them.

3.2.2 Remove

The final basic dictionary operation is `remove`. We could try to implement it now, but the logic is actually quite complicated. The algorithm has to recursively search for the key k to remove, rebuilding the tree it goes, much like `insert`. But what happens once it finds the node with key k? When it encounters a node `{L, (k,v), R}`, it must build a new tree containing just the contents of `L` and `R`, *and* ensure that the entire resulting tree is balanced.

So, instead of trying to do it now, we will come back to `remove` after we introduce two new powerful operations, `split` and `join`. Once we have them, implementing `remove` will be trivial!

4 Beyond the Basic Dictionary Interface

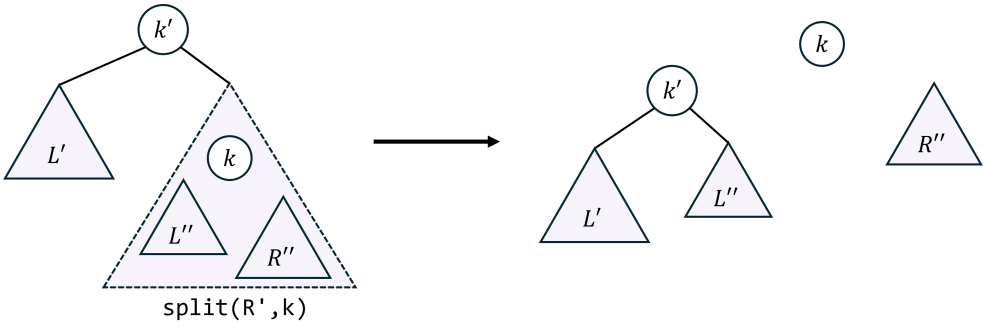
So far, we have focused on implementing the basic dictionary operations using BSTs, ones that take or return a single key or key-value pair. However, BSTs also support more powerful operations that manipulate *entire trees* or sets of keys at once!

We will now explore two very important structural algorithms on BSTs, **split** and **join**. These will first allow us to elegantly implement the remove operation, completing the basic sorted dictionary interface. More importantly, they will enable us to implement more complex *parallel algorithms* for dictionaries and sets, since so far, all of our algorithms have been sequential.

4.1 Split

One of the core functions for bulk and parallel BST operations is `split`. It is a useful building block for many other algorithms, including our generic implementation of `join`.

The `split(T, k)` function takes a tree T and a key k and returns (L, x, R), where L is a BST containing all elements of T whose keys are less than k, and R is a BST containing the elements with keys greater than k. The middle value x is an option type equal to `SOME`(v) if k exists in the tree and is associated with value v, or is equal to `NONE` otherwise.



Algorithm: BST: split

```
fun split(T: BST<K,V>, k: K) -> (BST, option<V>, BST):
  match expose(T) with:
    case NONE:
      return (empty, NONE, empty)
    case SOME(NodeView(L', (k',v'), R')):
      if k < k':
        L'', optV, R'' = split(L', k)
        return (L'', optV, join_mid(R'', (k',v'), R'))
      else if k > k':
        L'', optV, R'' = split(R', k)
        return (join_mid(L', (k',v'), L''), optV, R'')
      else:
        return (L', SOME(v'), R')
```

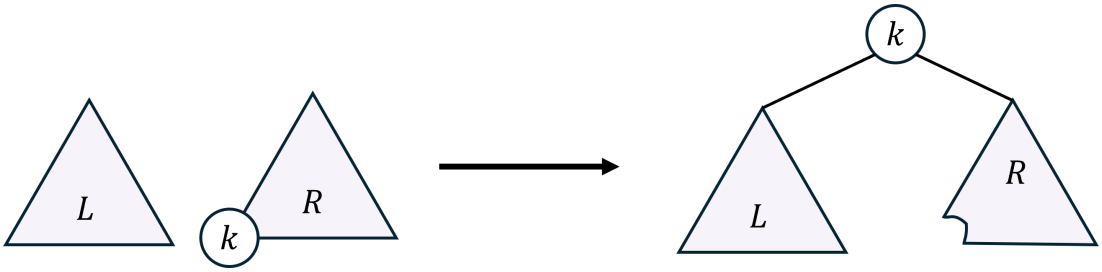
`split` is implemented recursively. Given a tree and a key, it checks whether the root of the tree has a key k' that is less than, greater than, or equal to k. Suppose, as in the picture above, that $k > k'$, then the algorithm has to recursively split the right subtree. After splitting the right subtree, it reconstructs the corresponding tree using `join_mid`, which also handles rebalancing.

When $k < k'$, the mirror opposite happens. The base cases are when it finds that $k = k'$ in which case the split simply returns the left and right subtrees and the value of the root, or the trivial base case of an empty tree.

4.2 Join

join is the counterpart to `split`, the other core operation for parallel BST algorithms; `split` takes a BST and breaks it into two, L and R, while `join` takes two BSTs L and R and glues them together into one BST. Note that it is a precondition of `join` that all the keys in L are less than all the keys in R. Without this precondition, `join` can not be implemented efficiently.

Let us think about the algorithm design idea behind `join`. We are given two trees L and R and we want to make them part of a single BST. Well, if we had a root node to put between them, it would be easy, we would just make a new BST with L and R as the children, but we *don't* have a root node to put between them. So the trick is, let's steal one! We can remove the minimum element of R (or the maximum of L) and use that as the root of the new tree. We can use `first` to do the job of finding the minimum element of R.



We need to do more than just *find* the minimum element of course, we also need to remove it since otherwise we would duplicate the minimum element. We could write a new algorithm from scratch for this, but the easiest way is to realize that we can just reuse the `split` function and split on the minimum! Splitting on the minimum key k means we will always get `(Empty, SOME(v), R')` where R' no longer contains k.

Algorithm: BST: join

```
fun join(L: BST<K,V>, R: BST<K,V>) -> BST:
  match first(R) with:
    case NONE: return L
    case SOME((k,v)):
      _, _, R' = split(R, k)
      return join_mid(L, (k, v), R')
```

The resulting BST must satisfy both the BST invariant and the balancing invariants of its underlying implementation. The precondition on L and R takes care of the former, while `join_mid` handles the latter. Remember, `join_mid` is entirely responsible for rebalancing!

4.3 Applications of Split and Join

Now that we have implemented `split` and `join`, we can use them as building blocks for implementing other dictionary operations.

4.3.1 Remove

We postponed implementing `remove` because it was unclear how to efficiently combine the left and right subtrees after removing a node. But now that we have `split` and `join`, the implementation is almost trivial!

First, we split the BST around the key k that we want to remove. This gives us two trees L and R containing all keys less than and greater than k, respectively, while discarding the key k itself. We can then simply join these two trees back together.

Algorithm: Dictionary: Remove (BST Implementation)

```
fun remove(T: BST<K,V>, k: K) -> BST<K,V>:
  L, _, R = split(T, k)
  return join(L, R)
```

That's it! Both `split` and `join` take care of maintaining the BST and balancing invariants, so `remove` needs no additional logic.

4.3.2 Get Range

The `get_range` operation takes a BST T and two keys k_1 and k_2, where $k_1 \leq k_2$, and returns a new BST containing exactly the keys k satisfying $k_1 \leq k < k_2$. We use the convention of *half-open ranges*, including the lower bound but excluding the upper bound, since this matches our convention for sequence operations like `scan`.

We can implement this using two calls to `split`. First, we need to split before k_1 to discard everything strictly less than k_1, then split the result of that at k_2 to discard elements with keys greater than or equal to k_2. Since `split(T, k)` *removes* an element with key exactly k, to discard the elements strictly less than k_1, the trick is to split on the *predecessor* of k_1.

Algorithm: Dictionary: Get Range (BST Implementation)

```
fun get_range(T: BST<K,V>, k1: K, k2: K) -> BST<K,V>:
  match prev(T, k1) with:
    case SOME((pk, _)): _, _, R = split(T, pk)
    case NONE:                  R = T
  L, _, _ = split(R, k2)
  return L
```

This operation will become **extremely useful** in a few lectures, when we introduce *augmented trees*, which allow us to efficiently answer queries about entire ranges of keys, rather than just individual keys. The combination of `get_range` and augmentation will allow us to solve *range queries* over any associative combining function in $O(\log n)$ cost per operation.

4.3.3 Filter

So far, all of our BST algorithms have followed a single path through the tree. But BSTs also support algorithms that process entire subtrees in parallel! A good example is `filter`, which works similarly to filtering a sequence: given a predicate `p : (K, V) -> bool`, it returns a new BST containing exactly the key-value pairs for which $p(k, v)$ is true.

The key insight is that given a node `{L, (k,v), R}`, we can recursively filter L and R in parallel. All that remains is to decide whether to keep (k, v) or discard. Keeping it corresponds to a `join_mid` while discarding it is simply a `join`. What would otherwise require extremely intricate and complicated balancing logic if implemented totally from scratch becomes nearly trivial again because `join_mid` and `join`!

Algorithm: Dictionary: Filter (BST Implementation)

```
fun filter(p: (K, V) -> bool, T: BST<K,V>) -> BST<K,V>:
  match expose(T) with:
    case NONE: return empty
    case SOME(NodeView(L, (k,v), R)):
      fL, fR = parallel (filter(p, L), filter(p, R))
      if p(k, v): return join_mid(fL, (k,v), fR)
      else:       return join(fL, fR)
```

Theorem: Work and Span of Filtering

`filter` on a balanced[a] BST costs $\Theta(n)$ work and $O(\log^2 n)$ span, assuming that `join_mid` and `join` each cost $O(\log n)$ work and span, and that evaluating p is constant cost.

[a] under the simplifying assumption of roughly even subtree sizes

Proof. For simplicity, suppose the BST is approximately evenly balanced, so each subtree has roughly half the elements. The work recurrence is then

$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n).$$

This is **leaf-dominated**, giving $W(n) = O(n)$. Since the algorithm must visit all n nodes, the work is $W(n) = \Theta(n)$. Since the two recursive calls execute in parallel, the span recurrence is

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n),$$

which is **balanced**, giving $S(n) = O(\log^2 n)$. □

5 Set-Theoretic Operations

We have already seen how BSTs can implement the basic dictionary and sorted dictionary operations. Now, we will consider three powerful operations that work on entire collections *and* are more efficient than their sequence counterparts: `union`, `intersection`, and `difference`.

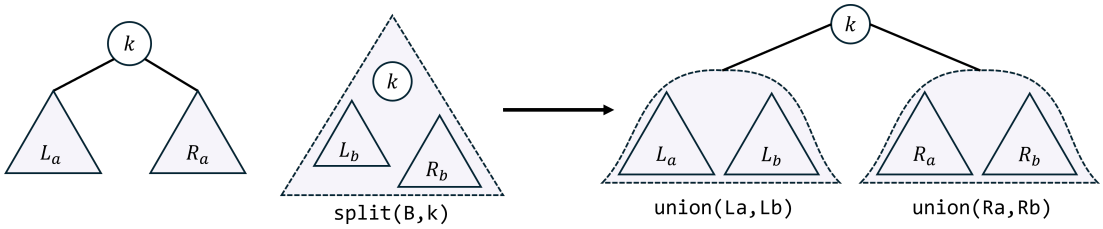
These are traditionally considered *set* operations, and indeed we defined them as part of our set and sorted set interfaces. However, they are equally useful for dictionaries! We can define them in terms of the sets of keys stored in each dictionary, preserving their associated values. Our sorted dictionary libraries support all three operations, and they will be important ingredients in algorithms we cover later, such as parallel breadth-first search (BFS)!

5.1 Union

The **union** operation takes two sets or dictionaries A and B, and returns a collection containing every key that occurs in *either* A *or* B, i.e., it corresponds exactly to the mathematical definition of the union of two sets.

For sets, that's the entire definition. For dictionaries, however, we must also decide which value to associate with a key that appears in both inputs. For simplicity, our implementation gives precedence to A: if a key appears in both dictionaries, we keep the value from A.

This sounds tricky, but just like the other algorithms we've seen, we can implement it in only a few lines of code thanks to `split` and `join_mid`! The algorithm takes the root key k of A and uses `split` to partition B around that key. It can then recursively union of the corresponding left and right subtrees *in parallel*, before constructing the result using `join_mid`.



Algorithm: Dictionary: Union (BST Implementation)

```
fun union(A: BST<K,V>, B: BST<K,V>) -> BST<K,V>:
  match (expose(A), expose(B)) with:
    case (NONE, _): return B
    case (_, NONE): return A
    case (SOME(NodeView(La, (k,v), Ra)), _):
      Lb, _, Rb = split(B, k)
      L, R = parallel (union(La, Lb), union(Ra, Rb))
      return join_mid(L, (k,v), R)
```

The base cases occur when either tree is empty, in which case we simply return the other tree.

Remark: Union with Tiebreaking

For dictionaries, we could instead let the caller provide a function $(V, V) \to V$ to resolve duplicate keys, either choosing one value or combining both (e.g., by addition).

This just adds a few lines of code and doesn't change the algorithm in any meaningful way, so we omit it for simplicity. Note that it is also easy to simulate even if your `union` interface doesn't support it: just take the intersection of the keys first, manually combine the corresponding values, and then union them into the final result.

Remark: Handwaving imminent

In our proof of the cost bounds of `union` below, we are going to make the additional assumption that A is not just balanced, but *perfectly balanced*. That is, we will assume that A is a binary search tree where every internal node has two children and every leaf node has the same depth.

Without this assumption, just under the assumption that the tree has approximate balance as induced by a balancing scheme such as AVL Trees, Red-Black Trees, Treaps, etc., the proof goes from being under two pages to being a 40-page research paper by **Blelloch** et al., which you are welcome to read **here** if interested!

Theorem: Cost of Union

Given two balanced binary search trees A and B of size m and n respectively, assuming WLOG that $m \leq n$ and assuming that `split` costs $O(\log n)$ work, union costs

$$O\left(m + m \log\left(\frac{n}{m}\right)\right)$$

work in $O(\log m \log n)$ span.

Proof. To satisfy the WLOG assumption that $m \leq n$, we can simply swap the two trees before calling union if A is larger than B. So, this assumption is implementable.

The algorithm selects the root key of the first tree A, splits the second tree B at that key, and then recursively unions the left and right subtrees. The work performed locally at each non-base case consists of splitting B and combining the results using `join_mid`. Splitting B costs $O(\log n)$ work, and `join_mid` costs at most $O(\log(m+n))$ work since it builds a tree of size at most $m+n$.

Let $W(m,n)$ denote the work to union trees of sizes m and n. If A is a singleton (i.e., $m = 1$), then the total work is $O(\log(1+n))$ to split B and `join_mid` the element of A into the result. Otherwise, it recurs on the left and right subtrees of A. Under our assumption that A is perfectly balanced, it gets split into two equally sized subtrees, so ignoring constant factors, we write

$$W(m,n) \leq \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0, \\ 1 + \log(1+n) & \text{if } m = 1, \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log(m+n) & \text{otherwise,} \end{cases}$$

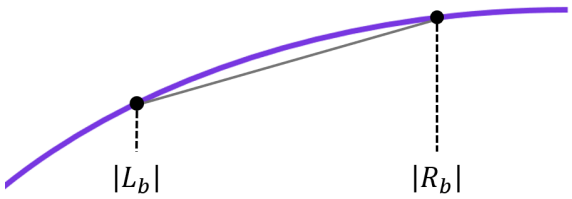
where $|L_b|+|R_b| \le n$.Note that B is *not* necessarily split exactly in half, since the sizes $|L_b|$ and $|R_b|$ depend on how many keys are less or greater than the root of A, which can be arbitrary.

What we will prove, however, is that splitting B evenly is the *worst case*. This is rather counterintuitive; for most divide-and-conquer algorithms, splitting in half is the best case. The reason is that we are paying logarithmic costs for the subproblems, and logarithm is a concave function.

More specifically, let's apply our favourite recurrence-solving technique, the **brick method**, and consider the cost of the children relative to the root of the recurrence. The local cost of the root is $\log(n+m)$, while the combined cost of the children is

$$\log\left(\frac{m}{2}+|L_b|\right)+\log\left(\frac{m}{2}+|R_b|\right),$$

where $|L_b|+|R_b| \le n$. Since log is *concave*, the value of the function $\log\left(\frac{m}{2}+x\right)$ at the midpoint between $|L_b|$ and $|R_b|$ is at least as high as the average of its values at those two points. Therefore, *the cost is highest/worst* when $|L_b| = |R_b| = n/2$. Formally, this is known as **Jensen's Inequality**.



Therefore, the cost of the children is maximized when $|L_b| = |R_b|$, in which case it is at most

$$2\log\left(\frac{m+n}{2}\right) = 2\log(m+n)-2.$$

Asymptotically, this is nearly twice the cost of the root, suggesting that the recurrence is **leaf-dominated**: as we descend the recursion tree, the number of calls doubles, while the cost per call decreases only logarithmically. So by the brick method, we can bound the total work by analyzing *the cost at the leaves*.

The recursion continues until the size of the A subtree reaches 1. Since the total number of nodes in A is m and we assume that A is perfectly balanced, there are $(m+1)/2$ leaves, each of which costs $W(1,b_i)$, where each b_i is the size of the corresponding subtree of B at that leaf. These values satisfy

$$\sum_{i=1}^{(m+1)/2} b_i \le n.$$

From our recurrence, each such leaf costs $1+\log(1+b_i)$. Therefore, the total work at the leaves is

$$\sum_{i=1}^{(m+1)/2} (1+\log(1+b_i)) = \frac{m+1}{2} + \sum_{i=1}^{(m+1)/2} \log(1+b_i).$$

Again, using the fact that the logarithm function is concave, this sum is maximized when the b_i are all equal (*Jensen's inequality*), so the summation is maximized as

$$\sum_{i=1}^{(m+1)/2} \log(1+b_i) \leq \frac{(m+1)}{2} \log\left(1+\frac{2n}{m+1}\right).$$

Since we are in a leaf-dominated recurrence, the work at internal nodes of the recursion tree is asymptotically dominated by the work at the leaves. Therefore, the total work of `union` is

$$O\left(m + m\log\left(\frac{n}{m}\right)\right).$$

For the span, observe that the two recursive calls execute in parallel. Since A is balanced, there are $O(\log m)$ levels of recursion, and at each level the sequential work costs at most $O(\log(n+m))$ where $m \leq n$. Therefore, the total span is $O(\log m \log n)$. □

Remark: Faster Union

It is possible to achieve an even better span of $O(\log n)$ for union using more advanced techniques beyond the scope of this course. If we use `union` in the implementation of future algorithms, we will therefore cite that it costs $O(\log n)$ span, even though we have not implemented or analyzed that version.

Remark: Implementing Intersection and Difference

We can also use similar algorithms for both intersection and set difference. The main difference is deciding whether to keep each key from A: for intersection, we keep k only if it is also found in B; for difference, we keep k only if it is not found in B. Whenever we discard a key, we use `join` rather than `join_mid` to combine the resulting subtrees.