

Parallel And Sequential Data Structures and Algorithms

Binary Search Trees

Learning Objectives

- Review binary search trees and how they can be used to implement **persistent sorted sets and sorted dictionaries**
- Understand how to design tree algorithms **generically**, independent of the specific balancing scheme
- Use structural operations such as **split** and **join** to build persistent BST algorithms
- Analyze the work and span of parallel BST operations such as **filter** and **union**

Review: Dictionaries and Sets

ADT Summary

dictionary<K,V>:

- Stores key-value pairs
- insert, find, delete

Can be implemented with a hash table in $\Theta(1)$ expected time (amortized insert).

set<K>:

- Stores a set of keys
- insert, contains, delete
- Also, union, intersection, and difference

SortedDict<K,V>:

- Stores sorted key-value pairs
- insert, find, delete
- Also, first, last, prev, next

Today: These, with binary search trees!

SortedSet<K>:

- Stores a set of sorted keys
- insert, contains, delete
- Also, union, intersection, and difference
- Also, first, last, prev, next

Hash Tables Cost Summary

Definition: With capacity m and n keys, $\alpha = n/m$ is the **load factor**

Table of size m

- insert: $O(1 + \alpha)$ in expectation
- find: $O(1 + \alpha)$ in expectation
- delete: $O(1 + \alpha)$ in expectation

Dynamically-sized table

- insert: $\Theta(1)$ **amortized** in expectation
- find: $\Theta(1)$ in expectation
- delete: $\Theta(1)$ in expectation

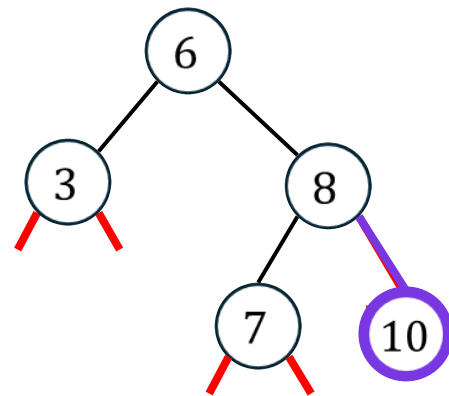
Costs of Randomized Data Structures: maximize, over **all possible operation sequences**, the expected value over the data structure's random choices of:

- **(Worst-case) Expected Cost:** the cost of a **single operation**
- **Amortized Expected Cost:** the **average cost per operation**

Review: Binary Search Trees and Persistence

Imperative BSTs

```
// source: 15-122
tree* bst_insert(tree* T, entry e) {
    if (T == NULL) return leaf(e);
    int cmp = key_compare(entry_key(e), entry_key(T->data));
    if (cmp == 0) T->data = e;
    else if (cmp < 0) T->left = bst_insert(T->left, e);
    else T->right = bst_insert(T->right, e);
    return T;
}
```



- Insertion into an imperative-style (mutable) data structure **modifies** the existing data structure

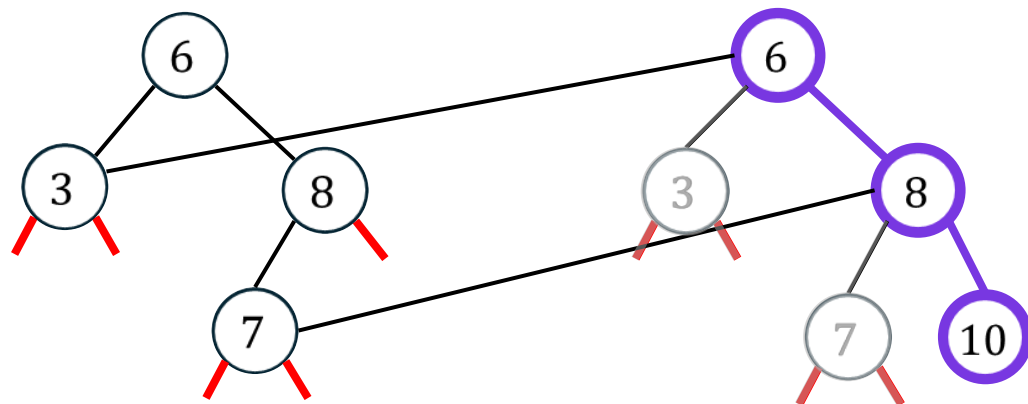
Example: `bst_insert(T, 10)`

- The right child of the 8-node is ***mutated*** to point to a new node containing 10



Functional BSTs

```
(* source: 15-150 *)  
fun insert (k, v) Empty =  
  Node (Empty, (k, v), Empty)  
| insert (k, v) (Node (L, (k', v'), R)) =  
  case Key.compare (k, k') of  
    EQUAL   => Node (L, (k, v), R)  
  | LESS    => Node (insert (k, v) L, (k', v'), R)  
  | GREATER => Node (L, (k', v'), insert (k, v) R)
```



- Insertion into a functional data structure returns a **new data structure**
- The existing data structure remains unmodified

Example: `insert(10, _)`

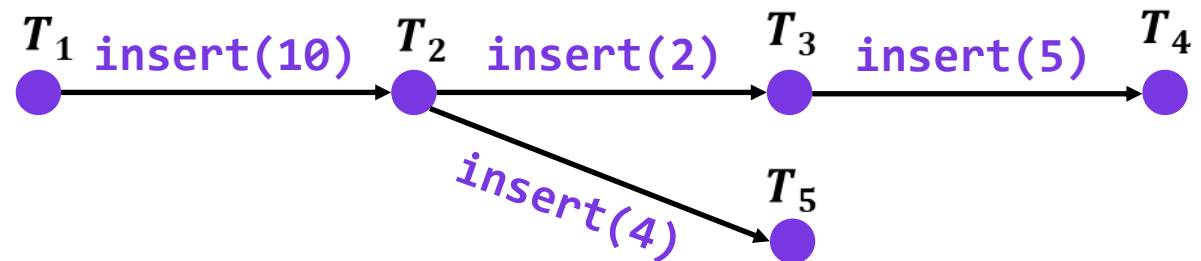
- New nodes are created for each node along the path
- Old nodes not on the insertion path are **reused**

Persistence

- Functional data structures provide a useful property for free:

Definition (Persistent data structure): A persistent data structure is a data structure that preserves the old versions of itself when it is updated.

Persistence is like having version-control history of your data structure!



Binary Trees

You've Seen Many Kinds of Trees

- E.g., (imbalanced) BST, Red-Black Tree, AVL Tree

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

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

```
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
```

This Looks Fun!

- Let's write some more!

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

```
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
```

// Coming next lecture!

```
type Treap<K> =  
  Empty  
| Node { left: Treap,  
         k : K,  
         priority : int,  
         right: Treap }
```



Designing Generic Algorithms

Definition: A **generic algorithm** is one that works for any type satisfying a specified interface.

- E.g., `contains` does not care whether the tree is AVL, Red-Black, a Treap, or even a sorted array. It only needs to know
 - How many elements are in the tree / is it empty?
 - What is the element at the root?
 - What are the left and right subtrees?

An Interface for Binary Trees

A type `Tree` is a `BinaryTree<Elem>` if:

```
size(T : Tree) -> int           For reading/traversing a tree  
expose(T : Tree) -> optional<NodeView<Tree>>
```

```
empty : Tree                    For creating new trees  
join_mid(L : Tree, e : Elem, R : Tree) -> Tree
```

`expose` asks to see the root element and its left and right subtrees.

*Irrelevant internal data such as balancing invariants (height, color, etc.) are **not** returned.*

`join_mid` creates a new tree with `e` at the root and `L` and `R` as its subtrees (**and then balances that tree**)

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

expose returns *only what the algorithm needs to see*

Writing Generic Algorithms

```
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
```

Same algorithm, any binary-tree representation. contains now only depends on expose, so it works for AVL trees, red-black trees, treaps, etc, once we implement expose for each of them

Example: Implementation

To implement different underlying tree types, we only need to write **four things**, then **every** generic tree algorithm can be used on it!

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

```
empty : MyTree<Elem> = Empty
```

join_mid **must maintain invariants**.
E.g., correctly setting the size field. For a proper implementation, **it must balance**.

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

Example: Implementation

To implement different underlying tree types, we only need to write **four things**, then **every** generic tree algorithm can be used on it!

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

expose **doesn't return internal invariant metadata**; only returns generic information

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

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

More Generic Than You Think

Observe: Our generic binary tree interface (size, expose, empty, join_mid) does **not** assume the tree is a binary **search** tree.

- We can use balanced binary trees to implement useful data structures other than binary search trees
- We will see an example two lectures from now, where we will implement a kind of **sequence** that supports being split and appended in $\Theta(\log n)$ cost per operation

Binary Search Trees

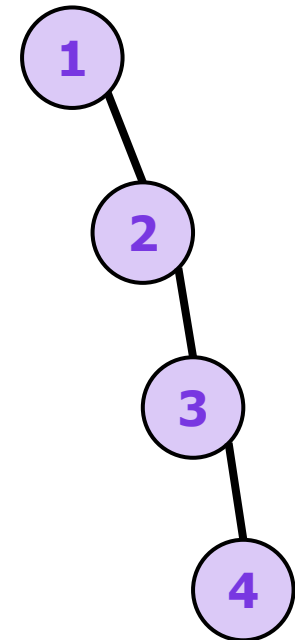
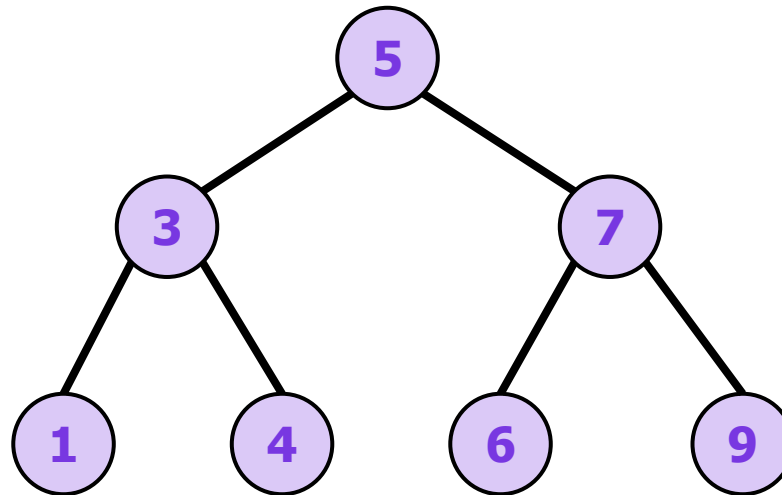
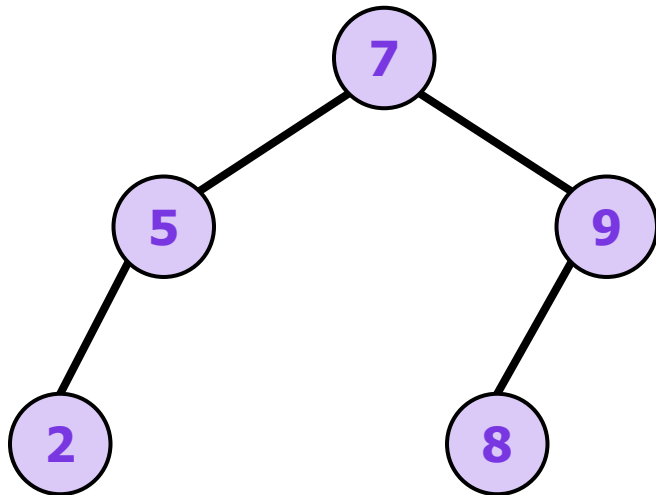
Ordering

- The keys in a binary search tree are kept in **sorted order** via maintaining the **BST Invariant**

Definition (BST Invariant): A binary tree with root key k and left/right subtrees L and R satisfies the BST invariant if:

- Every key in L is less than k
 - Every key in R is greater than k
- This is why the BST data type requires the key type K to be orderable, so that we can check $k_1 < k_2$, $k_1 = k_2$, $k_1 > k_2$

Examples



Keys or Key-Value Pairs

- Our generic binary tree type just stores generic **elements**
- Binary search trees may store either
 - **ordered keys**: to implement a `SortedSet`
 - **key-value pairs ordered by key**: to implement a `SortedDictionary`

We will describe how to implement BSTs whose **elements are key-value pairs** since the keys-only version is a special case and is simpler.

A **BST<K,V>** is a `BinaryTree<(K,V)>` whose keys satisfy the **BST Invariant**.

Common Operations on BSTs

- **size(T) -> int:**
return the number of elements in T
- **find(T, k : K) -> option<V>:**
return the value associated with the key k
(or NONE if it doesn't exist)
- **insert(T, k : K, v : V) -> BST<K,V>:**
return a new tree resulting from inserting
the key-value pair (k,v)
- **delete(T, k : K) -> BST<K,V>:**
return a new tree resulting from deleting the
item with key k

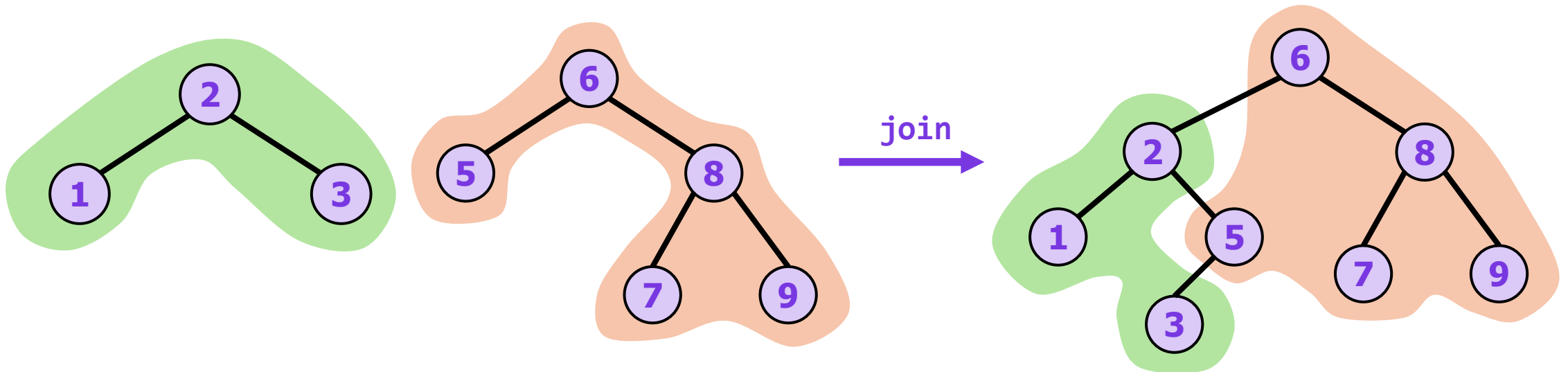
Implements the dictionary ADT!

- **first(T) -> option<(K,V)>:**
return the key-value pair with the least key,
or NONE if T is empty
- **last(T) -> option<(K,V)>:**
return the key-value pair with the greatest
key, or NONE if T is empty
- **next(T, k : K) -> option<(K,V)>:**
return the key-value pair with the least key
greater than k , or NONE if k is greatest
- **prev(T, k : K) -> option<(K,V)>:**
return the key-value pair with the greatest
key less than k , or NONE if k is least

Implements the SortedDict ADT!

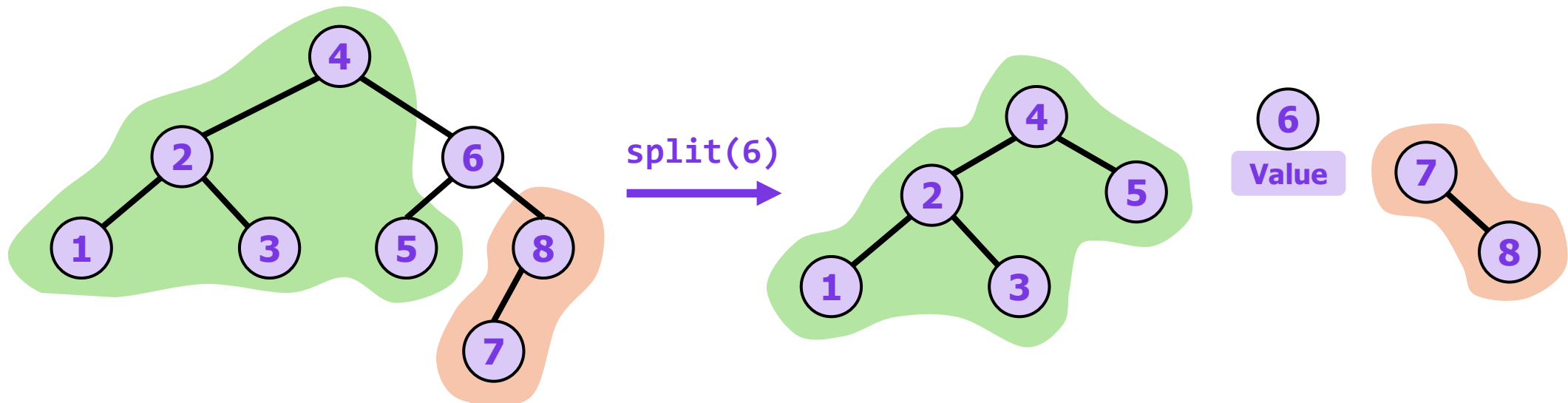
Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
 - **join**(L, R) $\rightarrow$ BST<K,V>:
join the two trees L and R, where every key in L is less than every key in R
 - **split**(T, k) $\rightarrow$ (BST<K,V>, option<V>, BST<K,V>):
splits T into two trees, one containing the keys less than k , one containing the keys greater than k , and if k is present, its value.



Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
 - **join**(L, R) $\rightarrow$ BST<K,V>:
join the two trees L and R, where every key in L is less than every key in R
 - **split**(T, k) $\rightarrow$ (BST<K,V>, option<V>, BST<K,V>):
splits T into two trees, one containing the keys less than k , one containing the keys greater than k , and if k is present, its value.



More Operations on BSTs

- **union**(T1, T2) $\rightarrow$ BST<K,V>:
return the union of T1 and T2
- **intersection**(T1, T2) $\rightarrow$ BST<K,V>:
return the intersection of T1 and T2
- **difference**(T1, T2) $\rightarrow$ BST<K,V>:
return the set difference of T1 and T2

Notice that every "update" operation we defined on BSTs **returns a new tree**, i.e., all these operations are purely functional. Hence, this is an interface for a persistent BST!

Implements the set and SortedSet ADT!

- Implementing this therefore gives us **persistent** dictionaries, sorted dictionaries, sets, and sorted sets!
- More versatile than hash tables, which are neither sorted nor persistent
- The downside is that BST operations will not be $\Theta(1)$ cost

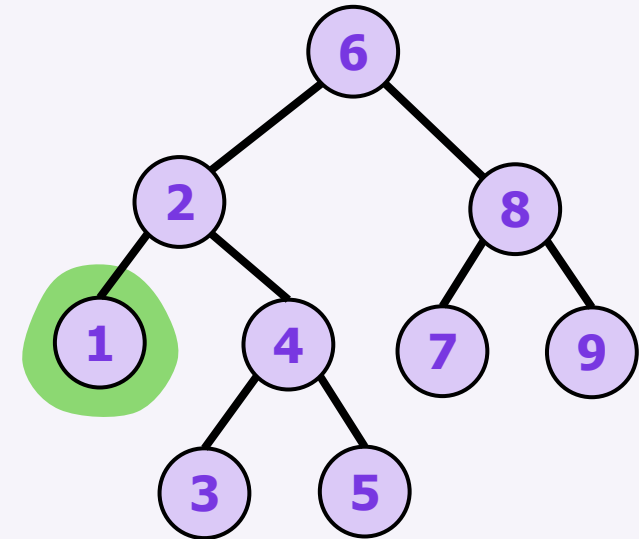
BST Algorithms

Read-Only Operations

```
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)
```

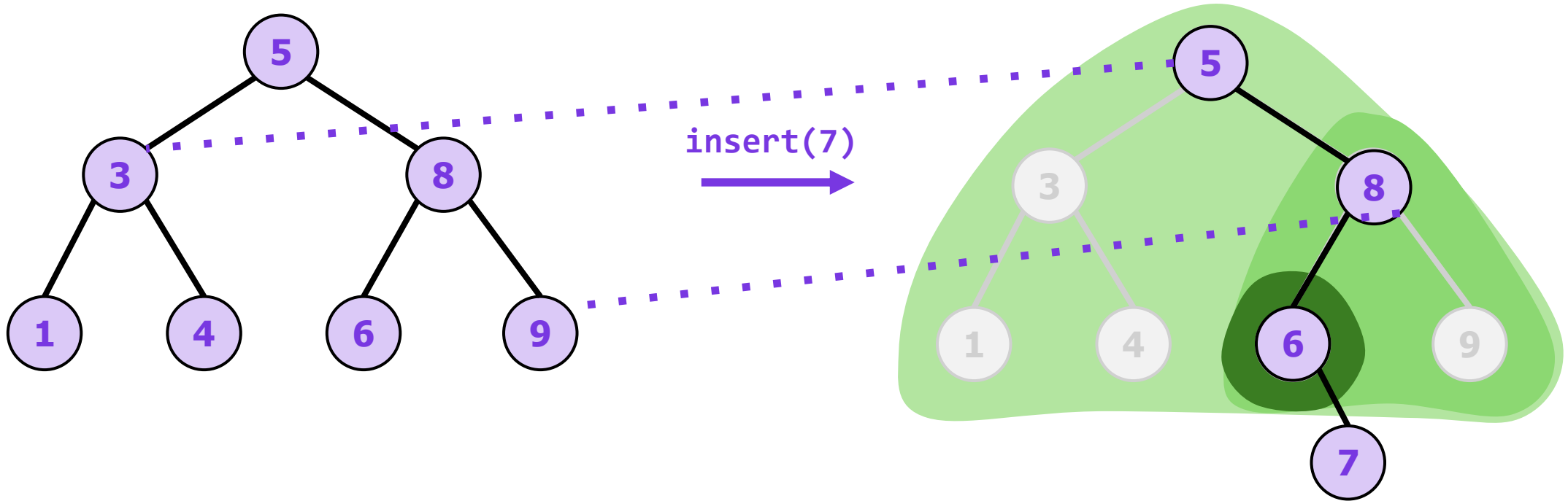
Read-Only Operations

```
fun first(T : BST<K,V>) -> option<(K,V)>:  
  match expose(T) with:  
    case NONE:  
      return NONE  
    case SOME(NodeView(L, (k,v), _)):  
      if size(L) == 0: return SOME((k,v))  
      else: return first(L)
```



- first finds the node of the tree with the least key
- This is the "leftmost" node in the tree

Insert



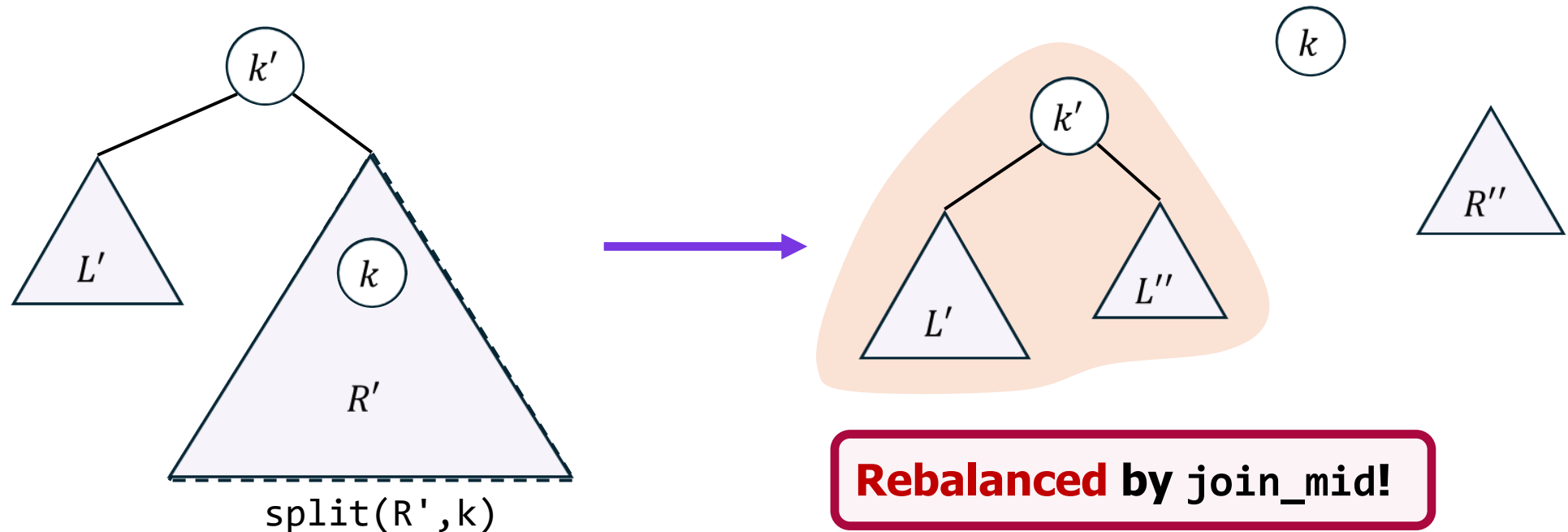
Insert

```
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)
```

Remember, updates do not mutate anything, they create a new tree.
They are **persistent updates**!

Structural Operations: Split

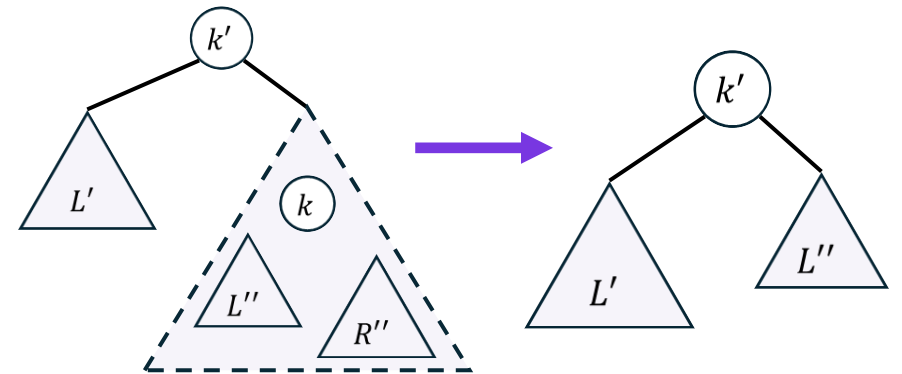
- **split**(T, k) $\rightarrow$ ($\text{BST}\langle K, V \rangle$, $\text{option}\langle V \rangle$, $\text{BST}\langle K, V \rangle$): splits T into two trees, one containing the keys less than k , one containing the keys greater than k , and if k is present, its value



Structural Operations: Split

- **split**(T, k) $\rightarrow$ ($\text{BST}\langle K, V \rangle, \text{option}\langle V \rangle, \text{BST}\langle K, V \rangle$):
splits T into two trees, one containing the keys less than k , one containing the keys greater than k , and if k is present, its value

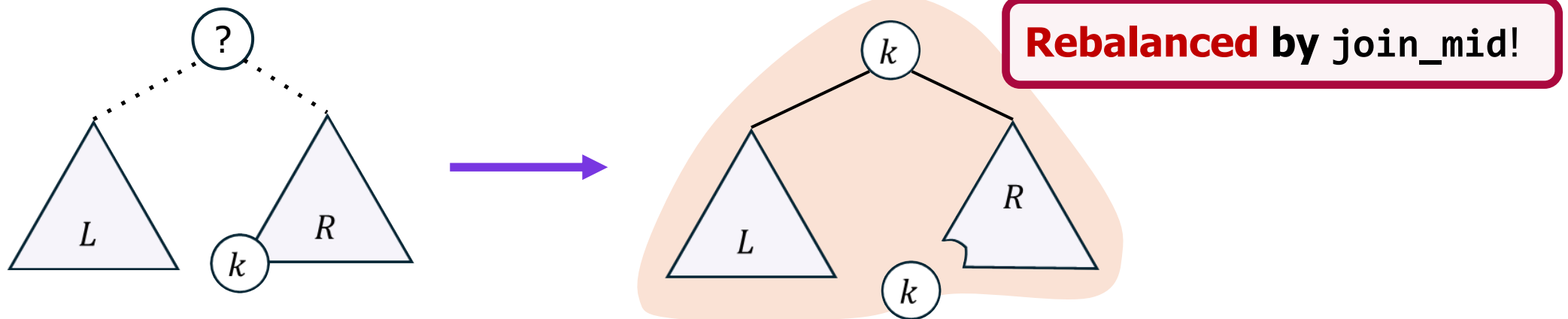
```
fun split(T : BST<K,V>, k : K) -> (BST, option<V>, BST):  
  match expose(T) with:  
  case NONE: return (empty, NONE, empty)  
  case SOME(NodeView(L', (k',v'), R')):  
    if k < k':  
      L'', optV, R'' = split(L', k)  
      return (L'', optV, join_mid(R'', (k',v'), R'))  
    else if k > k':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



Structural Operations: Join

- $\text{join}(L, R) \rightarrow \text{BST}\langle K, V \rangle$:
Merges two BSTs whose key ranges do not overlap, with all keys in L smaller than all keys in R

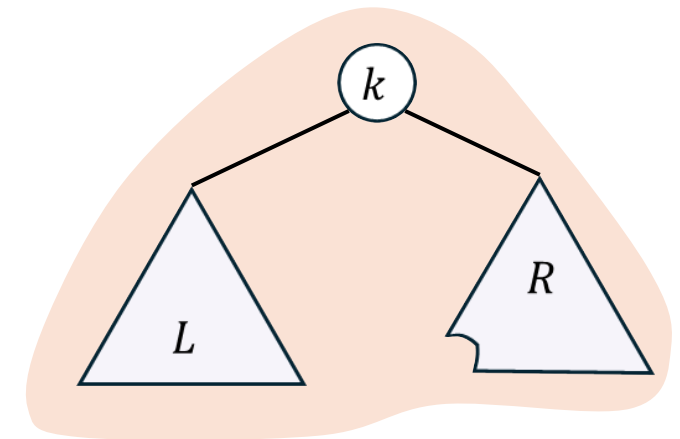
Note: semantically, join does the same thing as union. The difference is because join assumes a precondition (that all of L is less than all of R), it is **much more** efficient



Structural Operations: Join

- `join(L, R) -> BST<K,V>`:
Merges two BSTs whose key ranges do not overlap, with all keys in L smaller than all keys in R

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



Rebalanced by join_mid!

Filtering Operations

- `getRange(T : BST<K,V>, k1 : K, k2 : K) -> BST<K,V>:`
Return a tree containing the keys k in T for which $k_1 < k < k_2$

Hint: make use of `split`

```
fun getRange(T : BST<K,V>, k1 : K, k2 : K) -> BST:  
  _, _, R = split(T, k1)  
  M, _, _ = split(R, k2)  
  return M
```

Filtering Operations

- **filter**($p : V \rightarrow \text{bool}$, $T : \text{BST}\langle K, V \rangle$) $\rightarrow \text{BST}\langle K, V \rangle$:
Return a tree containing the key-value pairs with values v for which $p(v)$ is true

```
fun filter(p : V -> bool, T: BST<K,V>) -> BST:
  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(v):
      return join_mid(fL, (k,v), fR)
    else:
      return join(fL, fR)
```

Filter Analysis

Theorem (Work and Span of Filter): The filter function on a BST costs $O(n)$ work and $O(\log^2 n)$ span assuming a balanced binary tree implementation and assuming $p(v)$ can be evaluated in constant time.

Proof:

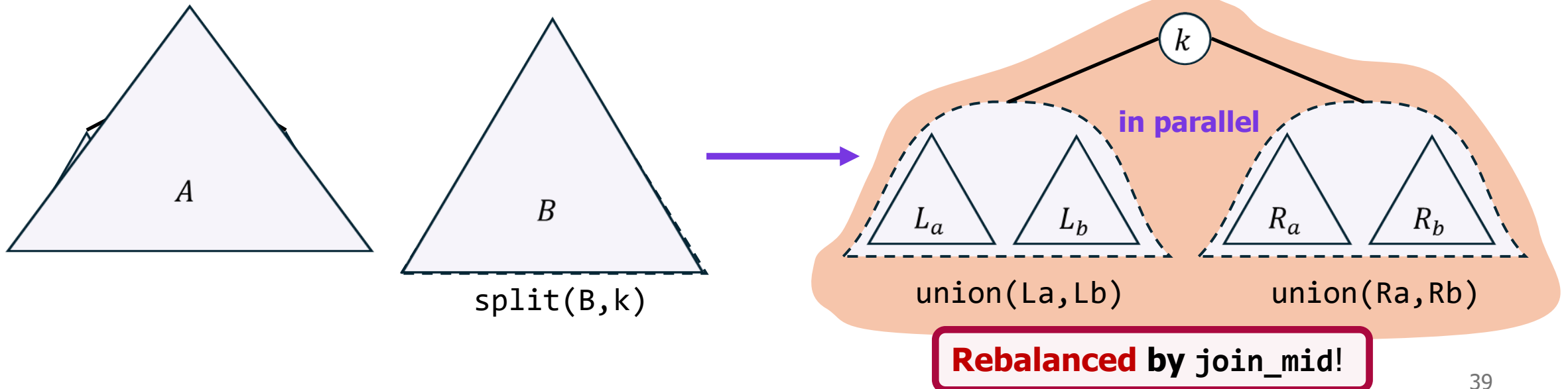
- $W(n) = 2W\left(\frac{n}{2}\right) + O(\log n)$ Leaf Dominated $\Rightarrow W(n) = O(n)$
- $S(n) = S\left(\frac{n}{2}\right) + O(\log n)$ Balanced $\Rightarrow S(n) = O(\log^2 n)$

Union

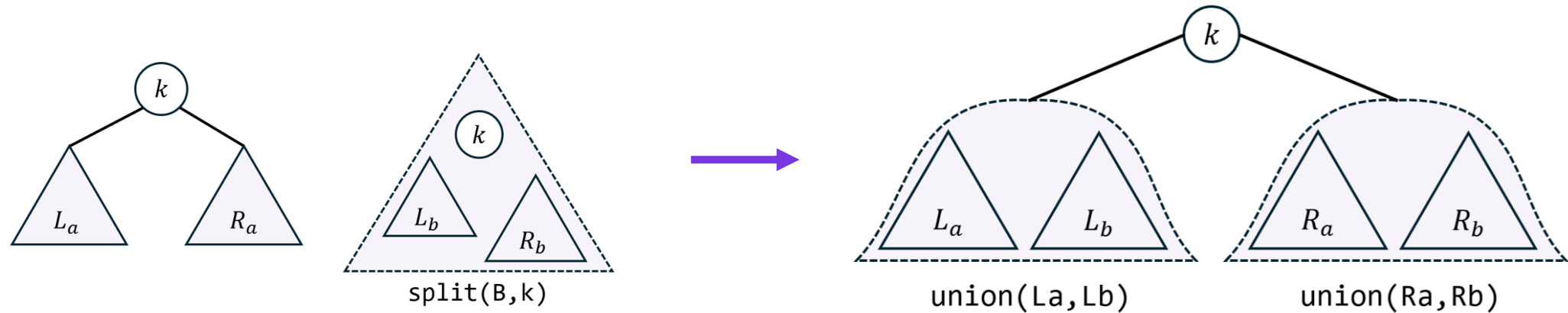
- $\text{union}(A, B) \rightarrow \text{BST}\langle K, V \rangle$:
return the union of A and B

Caveat: Which value do we keep if a key is in both? Doesn't matter for **sets**, but necessary for a dictionary.

Simplest choice: just use value from A.



Union



```
fun union(A : BST<K,V>, B : BST<K,V>) -> BST:  
  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)
```

Cost of Union

Theorem (Cost of Union): Given two balanced trees of size m and n , where WLOG we assume that $m \leq n$, union costs

$$O\left(m \log\left(1 + \frac{n}{m}\right)\right)$$

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

Proof: Next lecture

Summary

- Binary search trees can be used to implement **persistent sorted sets** and **persistent sorted dictionaries**
- Balanced binary trees algorithms don't have to depend on the specific tree implementation (i.e., balancing scheme)
- Structural operations like **split** and **join** are useful for designing other algorithms on trees
- We can implement **parallel algorithms** for operations like **union**, which achieve better cost bounds than on sequences