Parallel And Sequential Data Structures and Algorithms

Binary Search Trees

Learning Objectives

- Review binary search trees from 15-122 and 15-150
- See how to implement balanced binary search trees in a way that is generic across many rebalancing schemes (AVL Trees, Red-Black Trees, Treaps)

ADT Summary

`dictionary<K,V>`:

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

`set<K>`:

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

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

`SortedDict<K,V>`:

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

`SortedSet<K>`:

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

Today: These, with binary search trees!

Hash Tables Cost Summary

- Recall $\alpha = n/m$ is the **load factor**
- **Dynamic resizing** doubles the capacity when $\alpha > 1$

	Fixed-size table (capacity m)	Dynamically-sized table
`insert`	$\Theta(1+\alpha)$ expected	$\Theta(1)$ amortized expected
`find`	$\Theta(1+\alpha)$ expected	$\Theta(1)$ expected
`delete`	$\Theta(1+\alpha)$ expected	$\Theta(1)$ expected

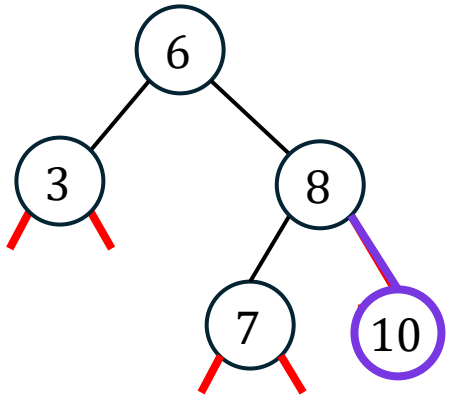
- The expected bounds hold for **any possible sequence of operations**, i.e., they are worst-case bounds

Binary Search Trees

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

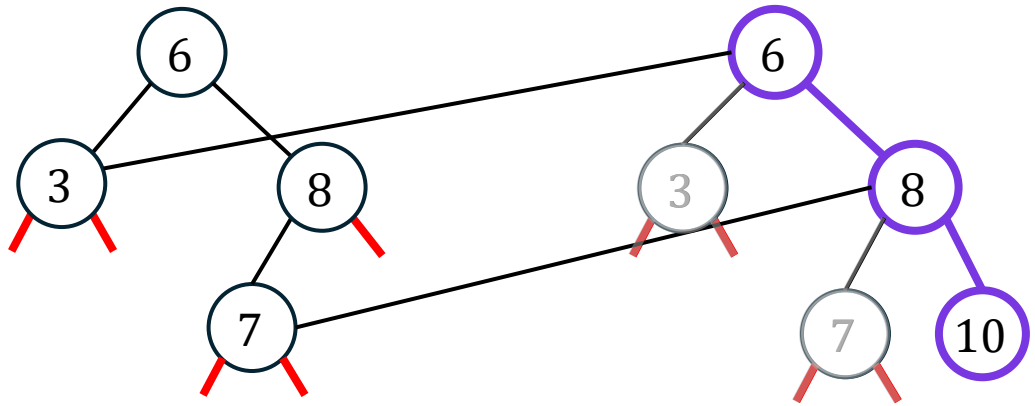


Example: `bst_insert(T, 10)`

- 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, _)`

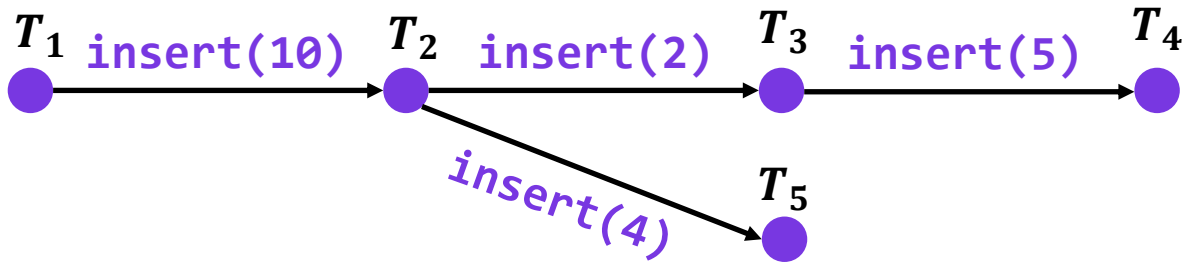
- New nodes are created for each node along the path
- Old nodes not on the insertion path are **reused**

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!



A Generic (Functional) BST Data Type

```
type BST<K : Ordered, V> =
    Empty
  | Node { left: BST,
           key: K,
           value: V,
           size: int,
           right: BST }
```

- A generic BST is parameterized by the type of the **keys** and **values** it stores at the nodes
- The key type K must be **ordered**, meaning we can compare keys, i.e., $k_1 < k_2,\ \ k_1 = k_2,\ k_1 > k_2$.

- The BST type is **recursive**. A BST is either empty, or it's a Node which itself has two BSTs as children/subtrees.
- The implementation in C++ versus SML (or generally, any OOP language versus any functional language), looks quite different.

Ordering

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

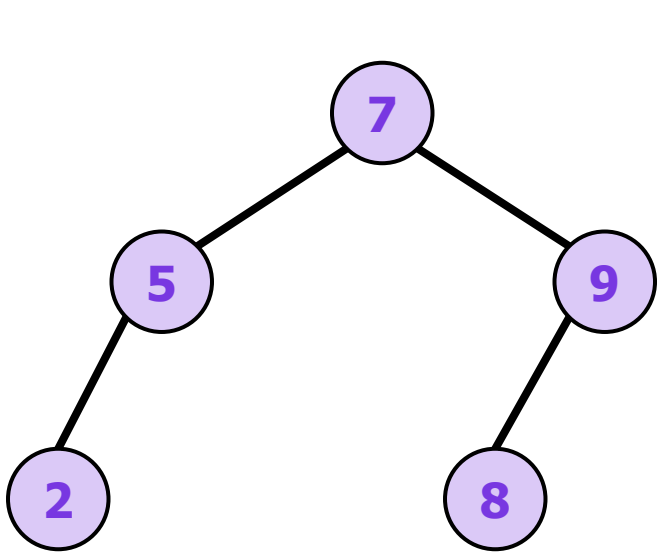
Definition (BST Invariant): A binary tree with root key k and left/right subtrees L and R satisfies the BST invariant if:

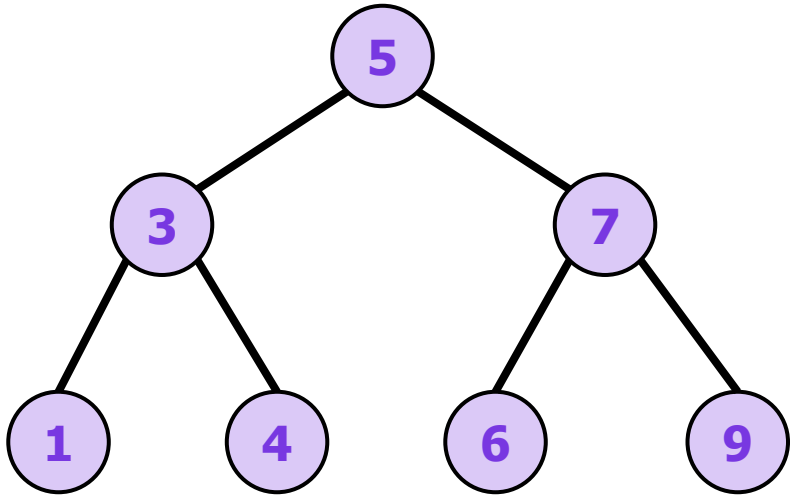
- Every key in L is less than k
- Every key in R is greater than k

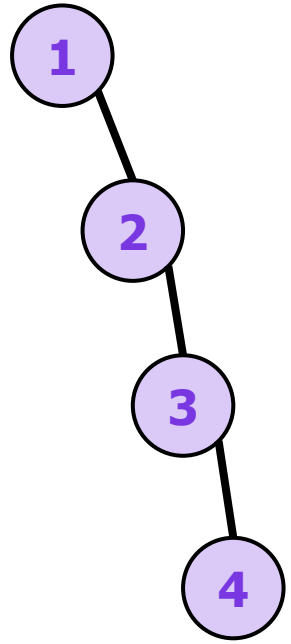
- This is why the BST data type requires the key type K to be orderable, so that we can check $k_1 < k_2,\ k_1 = k_2,\ k_1 > k_2$

Examples

Note: Like hash tables, we often ignore values when describing examples/algorithms since values are just data associated with the key, which is what matters for the algorithms







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!

More Operations on BSTs

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

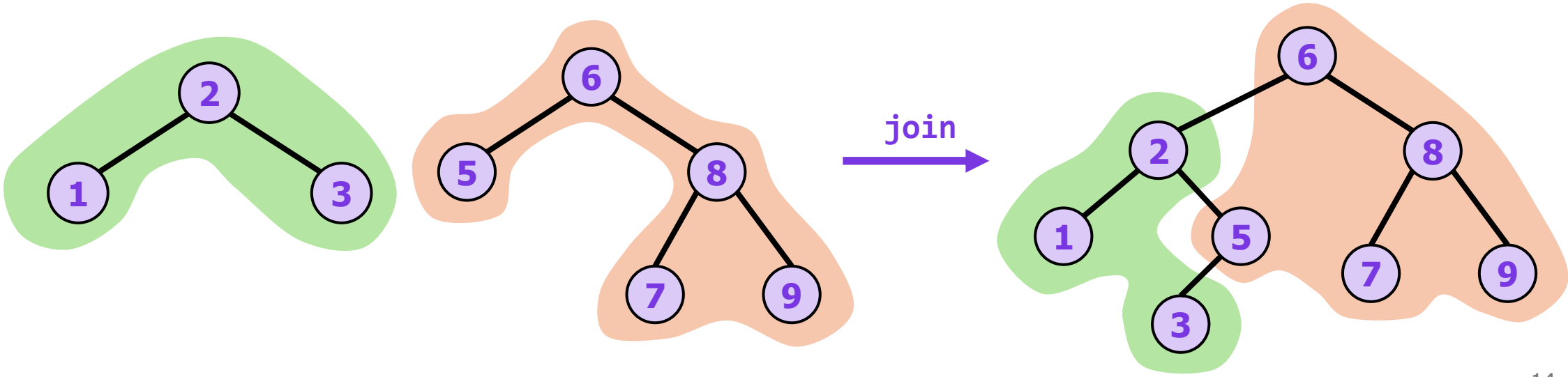
Implements the set and SortedSet ADT!

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!

- 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 not $\Theta(1)$ cost

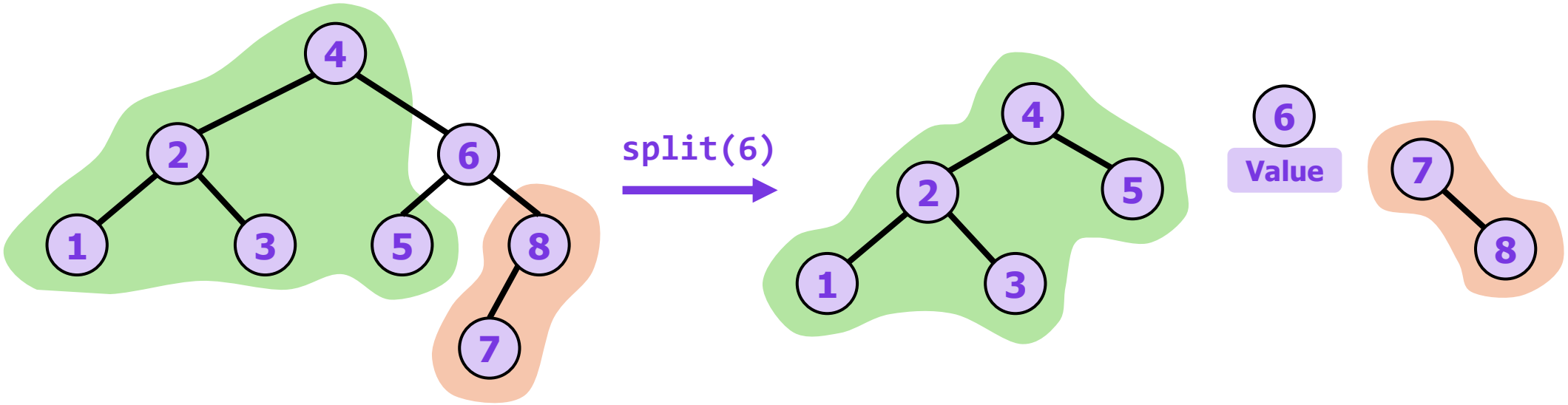
Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
  - **`join(L, R) ->`** `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) ->`** `(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.

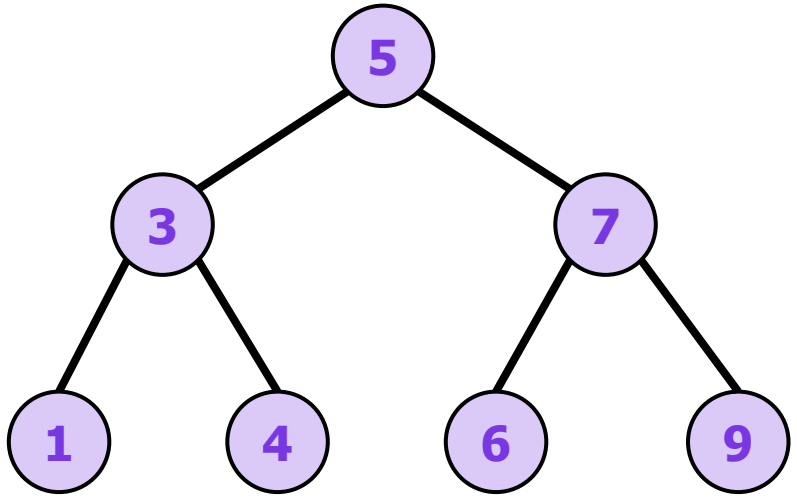


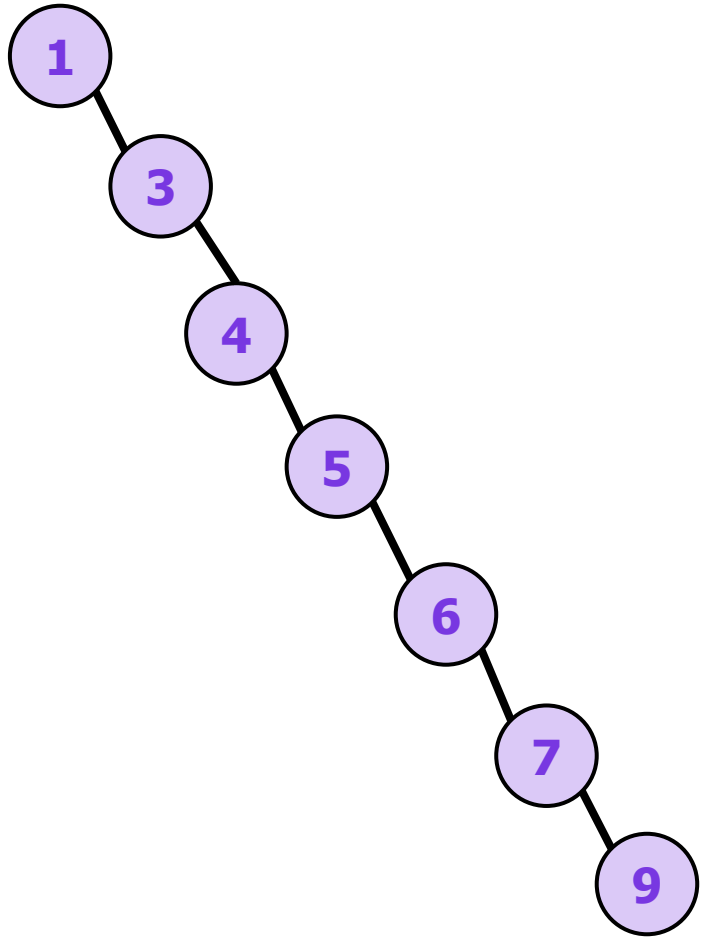
Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
  - **`join(L, R) ->`** `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) ->`** `(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.



Balance





Balance

- We prefer our trees balanced, since this makes most operations cost $O(\log n)$ in the worst-case rather than $O(n)$.
- You learned about **AVL Trees** in 15-122, and **Red-Black Trees** in 15-150.

In this class we will study a set of algorithms for implementing balanced binary search trees that work for **any balancing scheme**.

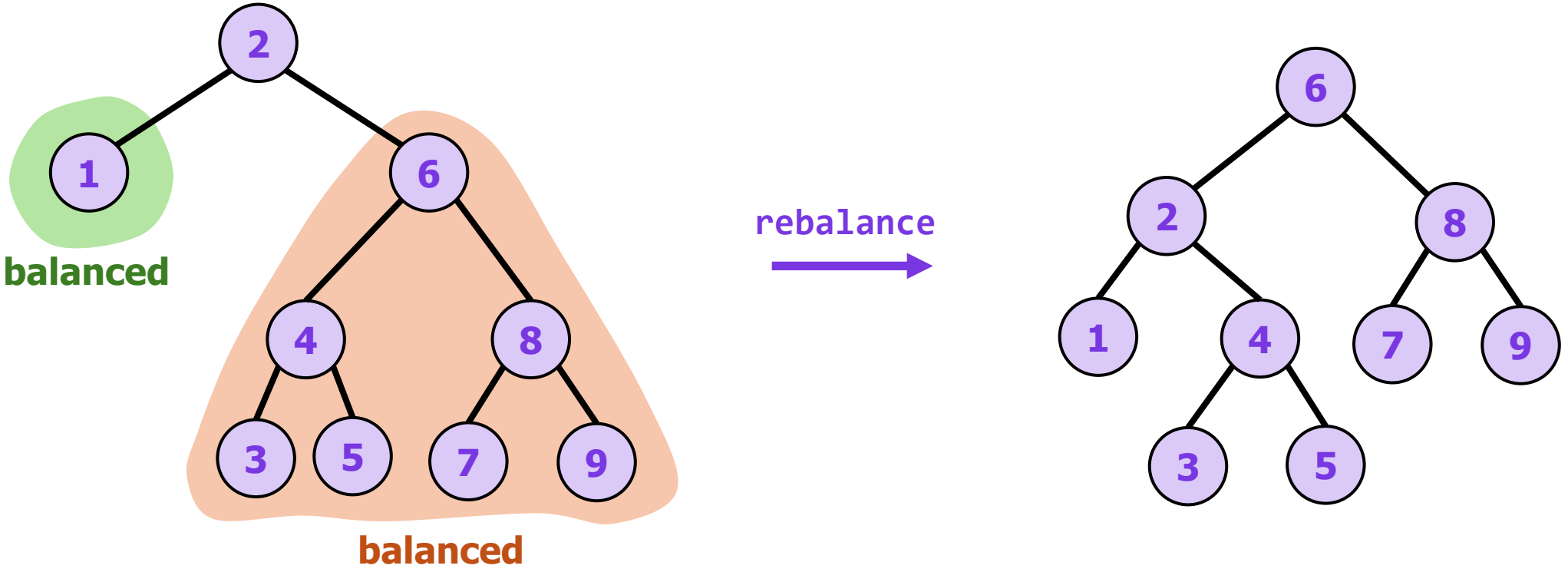
- In other words, we can implement all the algorithms and just "plug in" a balancing scheme without having to change a single line of code in any of the algorithms.

The `rebalance` primitive

- To implement our BST algorithms so that they keep the tree balance, we will assume we have a function **rebalance**
- Next lecture, we will implement rebalance for a particular class of balanced BSTs called **treaps**

Definition (Rebalance): Rebalance takes a BST whose left and right children are balanced (but the root might not be) and returns a new BST that is balanced (now including at the root node).

Rebalance example



rebalance can be implemented using AVL-Tree rotations, Red-Black Tree rotations (with a color field), or next lecture, treaps!

BST Algorithms

Implementing BSTs

```
type BST<K : Ordered, V> =
 Empty | Node { left: BST, key: K, value: V, size: int, right: BST }

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

- `makeNode` is a constructor for nodes
- It should initialize any invariants needed by the algorithm
  - E.g., in a Red-Black Tree, it should initialize the color field
- `makeNode` does not call `rebalance`; algorithms call `rebalance`
  - We choose this convention to make it clearer where rebalancing occurs

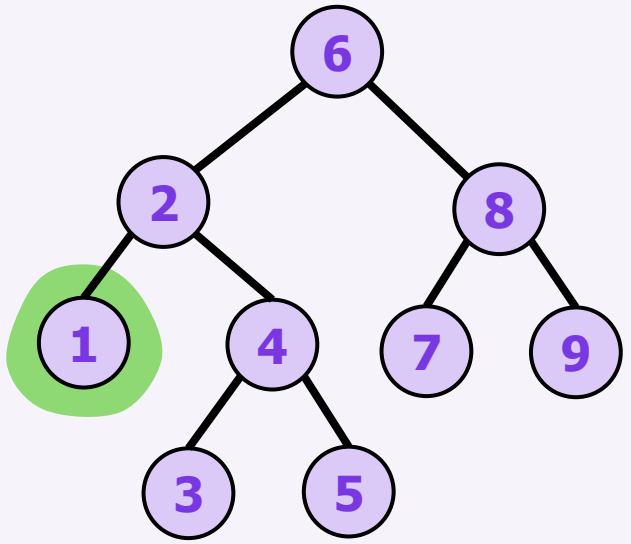
Read-Only Operations

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

```
fun find(T : BST<K,V>, k : K) -> option<V>:
  match T with:
    case Empty: return NONE
    case Node(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 T with:
    case Empty:
      return NONE
    case Node(Empty,k,v,_,_):
      return SOME((k,v))
    case Node(L,_,_,_,_):
      return first(L)
```



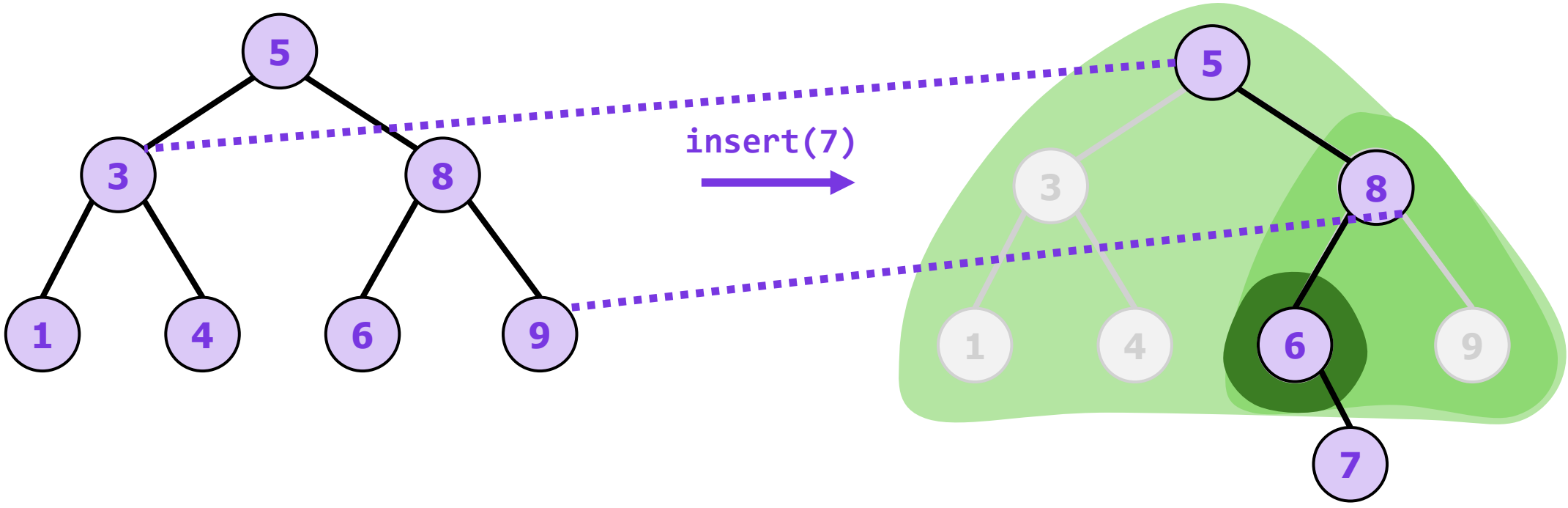
- first finds the node of the tree with the least key
- This is the "leftmost" node in the tree

Single-Update Operations

```
fun insert(T : BST<K,V>, k : K, v : V) -> BST<K,V>:
  match T with:
    case Empty:
      return rebalance(makeNode(Empty, k, v, Empty))
    case Node(L,k’,v',_,R):
      if k < k’:
        return rebalance(makeNode(insert(L, k, v), k’, v’, R))
      else if k > k’:
        return rebalance(makeNode(L, k’, v’, insert(R, k, v))
      else:
        return rebalance(makeNode(L, k, v, R))
```

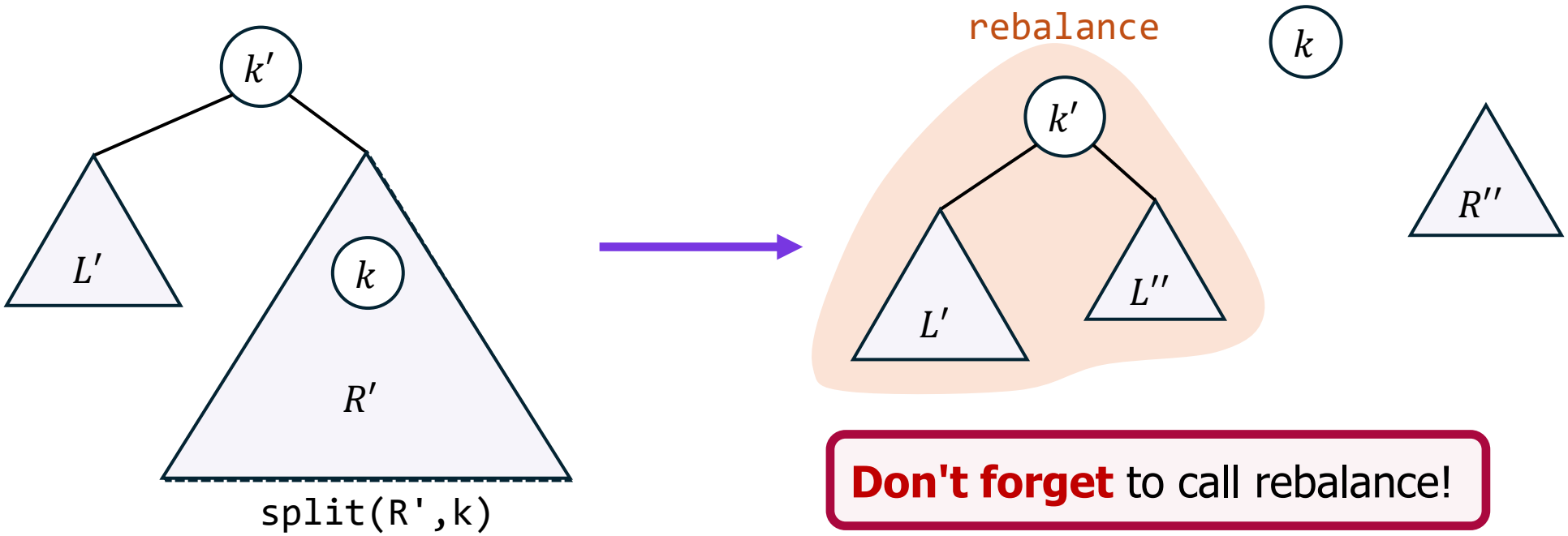
Remember, updates do not mutate anything, they create a new tree. They are **persistent updates**!

Insert Example



Structural Operations: Split

- **`split(T, k) ->`** `(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

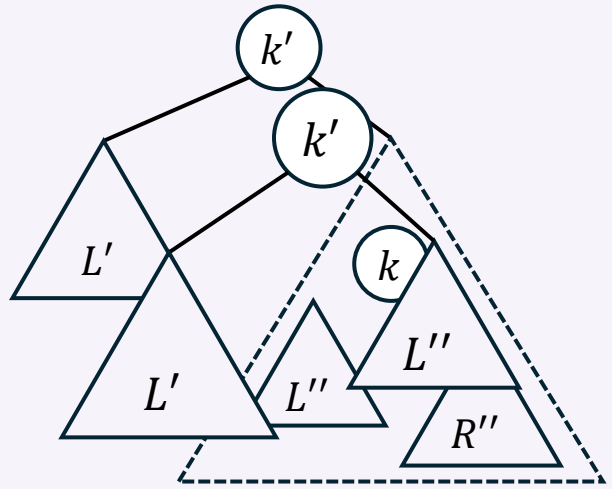


Don't forget to call rebalance!

Structural Operations: Split

- **`split(T, k) ->`** `(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

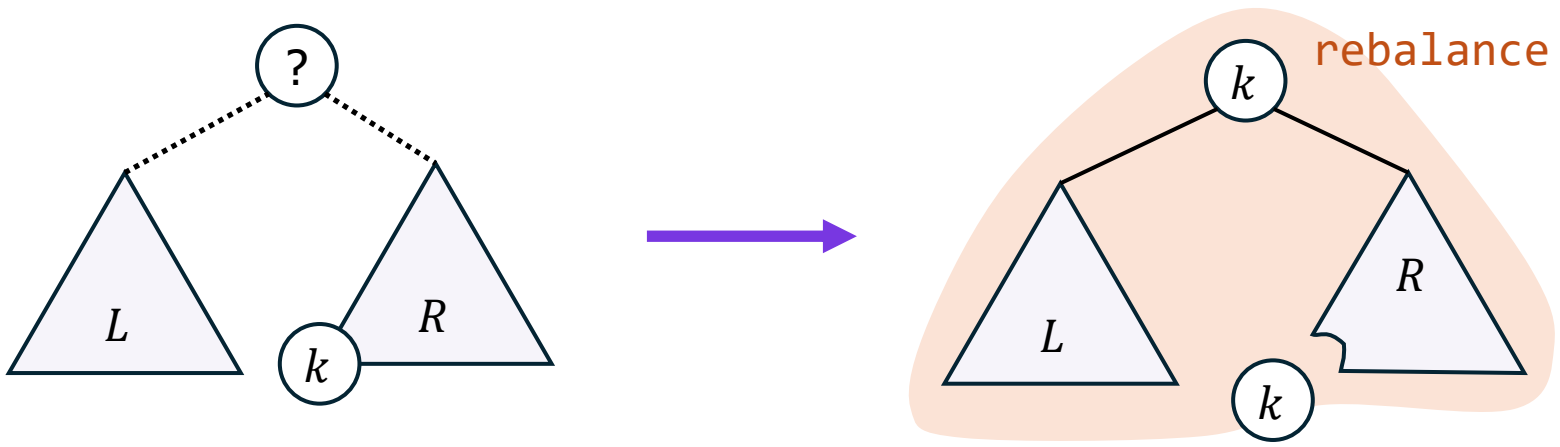
```
fun split(T : BST<K,V>, k : K) -> (BST, option<V>, BST):
  match T with:
    case Empty: return (Empty, NONE, Empty)
    case Node(L',k',v',_,R'):
      if k < k':
        L'', optV, R'' = split(L', k)
        return (L'', optV, rebalance(makeNode(R'', k', v', R')))
      else if k > k':
        L'', optV, R'' = split(R', k)
        return (rebalance(makeNode(L', k', v', L'')), optV, R'')
      else: return (L', SOME(v'), R')
```



Structural Operations: Join

- **`join(L, R) ->`** `BST<K,V>`:
  join the two trees `L` and `R`, where every key in `L` is less than every key in `R`

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

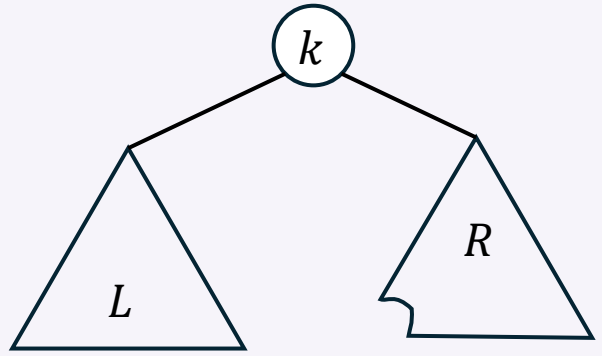


Don't forget to call rebalance!

Structural Operations: Join

- **`join(L, R) ->`** `BST<K,V>`:
  join the two trees `L` and `R`, where every key in `L` is less than every key in `R`

```
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 rebalance(makeNode(L, k, v, R'))
```



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 -> bool, T : BST<K,V>) ->`** `BST<K,V>`:
  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 T with:
    case Empty: return Empty
    case Node(L,k,v,_,R):
      fL, fR = parallel (filter(p, L), filter(p, R))
      if p(v):
        return rebalance(makeNode(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 rebalance costs $O(\log n)$ and assuming $p(v)$ can be evaluated in constant time.

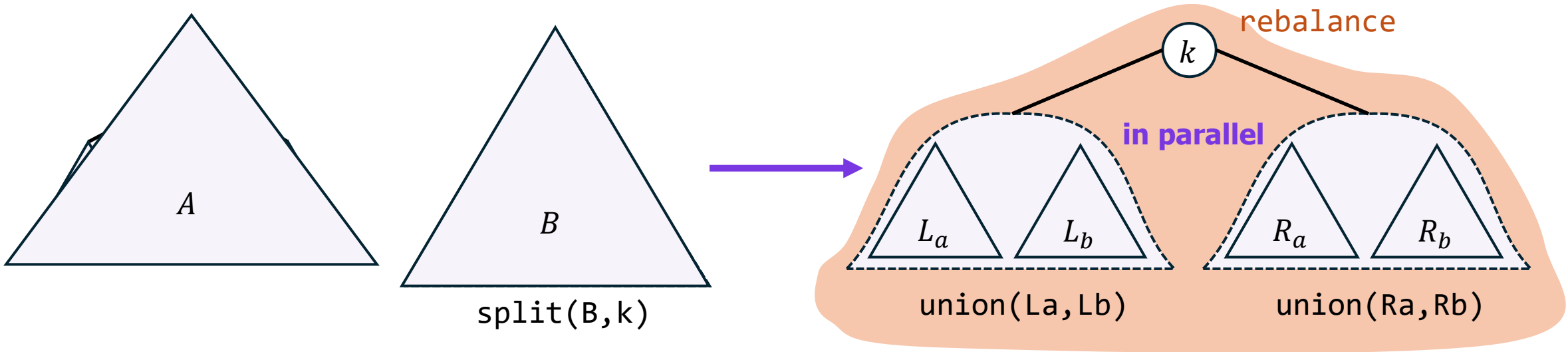
Proof:

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

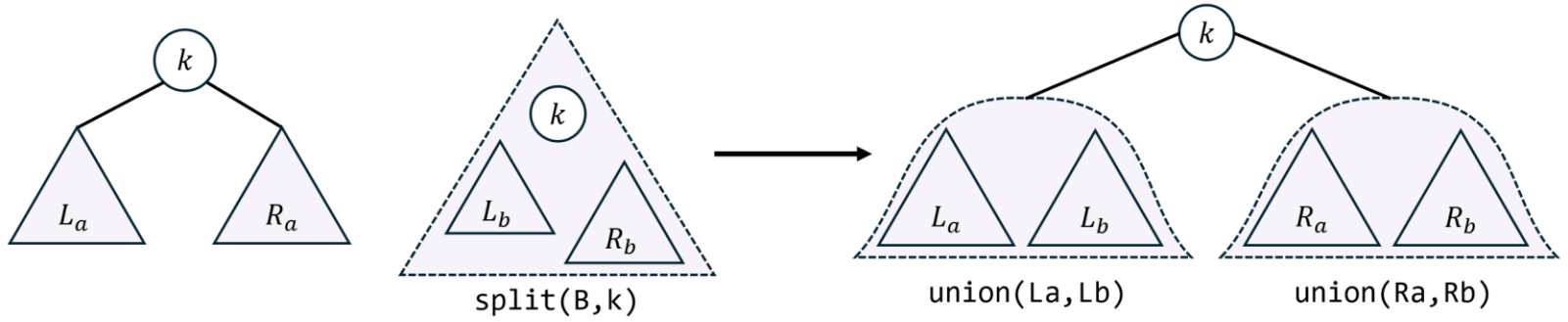
Union

- **`union(A, B) ->`** `BST<K,V>`:
  return the union of `A` and `B`

Caveat: Which value do we keep if a key is in both? For now, just take value from A.



Union



- **union(A, B) ->** BST<K,V>:
  return the union of A and B

```
fun union(A : BST<K,V>, B : BST<K,V>) -> BST:
  match (A, B) with:
    case (Empty, _): return B
    case (_, Empty): return A
    case (Node(La, k, v, _, Ra), _):
      Lb, _, Rb = split(B, k)
      L, R = parallel (union(La, Lb), union(Ra, Rb))
      return rebalance(makeNode(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 implemented **persistent sorted sets** and **persistent sorted dictionaries**
- Balanced binary search trees can be implemented using a generic function **rebalance**, which can be implemented using AVL, Red-Black, or Treap-style rebalancing (next lecture)
- 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