Parallel And Sequential Data Structures and Algorithms

Extensions/Augmentations of BSTs and Dictionaries

Learning Objectives

- Motivate augmentation by considering order statistics queries
- See how trees can be used to efficiently support sequence appends
- Implement and apply reduced-value augmentation to problems from 15-210

Review of BSTs

Review: An Interface for Binary Trees

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

```
size(T : Tree) -> int
expose(T : Tree) -> optional<NodeView<Tree>>

empty : Tree
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*

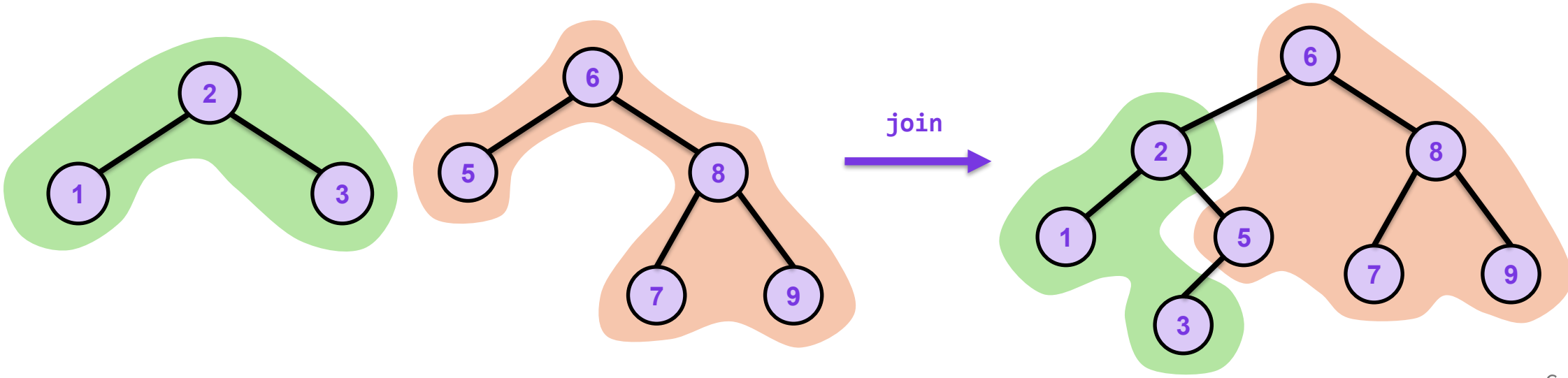
Review: Writing Generic Algorithms

```
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

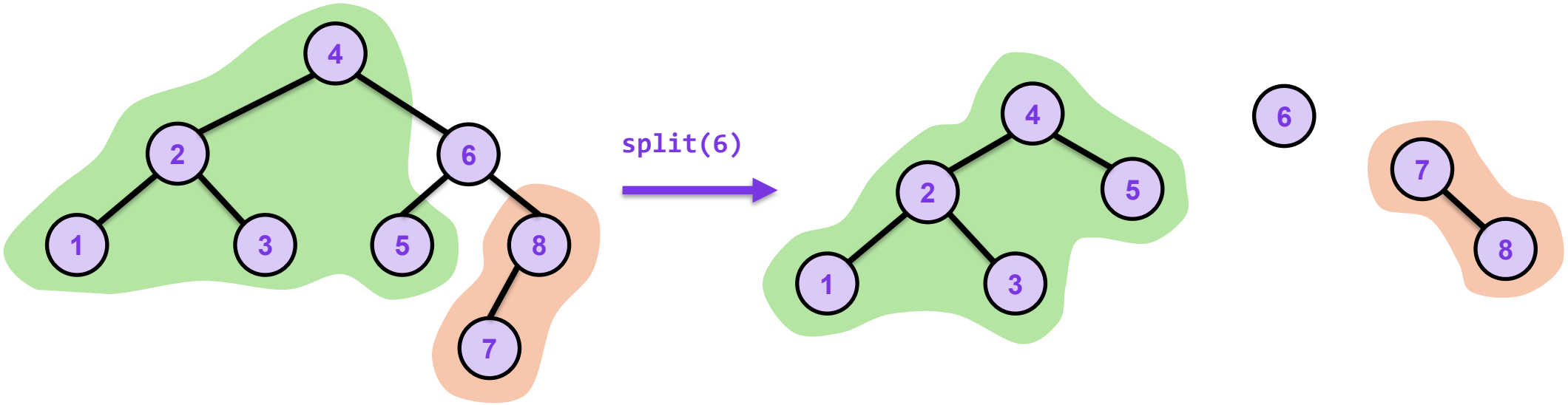
Review: Structural Operations on BSTs

- 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.



Review: 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.



Order Statistics Queries

Motivation for Generic Augmentation

`dictionary<K,V>`:
- Stores key-value pairs
- `insert`, `find`, `delete`

```
D = {13:'e',5:'g',3:'d',9:'a',7:'b',1:'c',11:'f'}
```

`SortedDict<K,V>`:
- Stores sorted key-value pairs
- `insert`, `find`, `delete`
- Also, `first`, `last`, prev, next

```
D = {1:'c',3:'d',5:'g',7:'b',9:'a',11:'f',13:'e'}
```

- We have access to the first and last elements of a sorted dictionary.
- Maybe access to the **median** may be useful.
- In fact, wouldn't it be cool to have rank-based access to ***any*** arbitrary element?

Order Statistics Sorted Dictionary

OrdSortedDict<K,V>:
- SortedDict<K,V>
- select
- rank

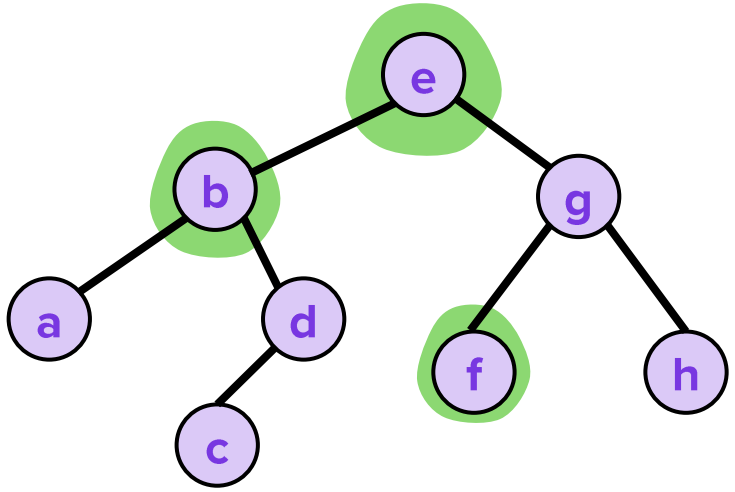
select(T: BST<K, V>, i: int) -> option<(K, V)>

returns NONE if no key with rank i exists; otherwise returns an option containing the key-value pair corresponding to the key with rank i in T.

rank(T: BST<K, V>, k: K) -> int

returns the rank of the key k in T, i.e., the number of keys in T that are less than k.

Order Statistics Sorted Dictionary




```
select(T, 4) =
```

```
select(T, 1) =
```

```
select(T, 5) =
```

Access to a node's size helps us find the correct key!

`size()` Implementation 1:

- Idea 1: Traverse entire tree and count

```
fun size(T: BST<K,V>) -> int:
  match (expose(T)) with:
    case NONE: return 0
    case (SOME(NodeView(L, (k,v), R))):
      return ______
```

`size()` Implementation 1:

- Idea 1: Traverse entire tree and count

```
fun size(T: BST<K,V>) -> int:
  match (expose(T)) with:
    case NONE: return 0
    case (SOME(NodeView(L, (k,v), R))):
      return 1 + size(L) + size(R)
```



$$W = O(n)$$

This is not very efficient!!

- Can we make this quicker?

  Yes! By storing the size information at the node itself.

Augmentation: Storing pre-computed information at the nodes of a BST

`size()` Implementation 2:

- Idea 2: *Augment* the node with size
  - Store size as a field at the node
  - Access in $O(1)$; `size()` also $O(1)$

Not a generic BST transformation!
Must directly add to your implementation.

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

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

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

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return ______
    case SOME(NodeView(L, (k, V), R)):
      if ______: return ______
      else if ______: return ______
      else: return ______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if ______: return ______
      else if ______: return ______
      else: return ______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return ______
      else if ______: return ______
      else: return ______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return ______
      else if i == size(L): return ______
      else: return ______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return select(L, i)
      else if i == size(L): return _______
      else: return _______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return select(L, i)
      else if i == size(L): return SOME(k, V)
      else: return ______
```

`select` Implementation

- Let's implement **`select`** and `rank` now!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return select(L, i)
      else if i == size(L): return SOME(k, V)
      else: return select(R, i-size(L)-1)
```

rank Implementation

- Let's implement `select` and **`rank`** now!

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

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if ______: return ______
      else if ______: return ______
      else: return ______
```

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k': return ______
      else if ______: return ______
      else: return ______
```

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k': return ______
      else if k < k': return ______
      else: return ______
```

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k': return size(L)
      else if k < k': return _______
      else: return _______
```

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k': return size(L)
      else if k < k': return rank(L, k)
      else: return ______
```

rank Implementation

- Let's implement `select` and **`rank`** now!

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE: return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k': return size(L)
      else if k < k': return rank(L, k)
      else: return size(L) + 1 + rank(R, k)
```

Order Statistics Functions

```
fun select(T: BST<K,V>, i: int)
    -> Option<(K, V)>:
  match expose(T) with:
    case NONE:
      return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L):
        return select(L, i)
      else if i == size(L):
        return SOME(k, V)
      else:
        return select(R, i-size(L)-1)
```

```
fun rank(T: BST<K,V>, k: K) -> int:
  match expose(T) with:
    case NONE:
      return 0
    case SOME(NodeView(L, (k', V), R)):
      if k == k’:
        return size(L)
      else if k < k’:
        return rank(L, k)
      else:
        return size(L) + 1 + rank(R, k)
```

We call `size()` a lot, so having it precomputed makes these functions efficient!

Quick Recap:

- We know what ***augmentation*** is: keeping pre-computed information at the node to not recompute every time, provides constant time access, updates as the tree goes through structural changes
- We used the idea of size augmentation to build a dictionary that supports order statistics operations (`rank`, `select`)
- Let's move on to another similar but different example of how size augmentation can be useful

Tree Sequences

One quick observation!

```
fun select(T: BST<K,V>, i: int) -> Option<(K, V)>:
  match expose(T) with:
    case NONE: return NONE
    case SOME(NodeView(L, (k, V), R)):
      if i < size(L): return select(L, i)
      else if i == size(L): return SOME(k, V)
      else: return select(R, i-size(L)-1)
```

select does not actually depend on the BST invariant!
It's just the i^{th} item in the in-order traversal

⟶ **Generalizable!!!!**

ArraySequence vs. TreeSequence

- We have been using `ArraySequences` for cost bounds (array-based).
- We'll now explore the `TreeSequence` interface (tree-based).

	`ArraySequence`	`TreeSequence`
`nth`	$O(1)$	$O(\log n)$
`subseq`	$O(1)$	$O(\log n)$
`length`	$O(1)$	$O(1)$
`update*`	$O(n)$	$O(\log n)$
`append`	$O(n)$	$O(\log n)$

* Update is O(1) if using a mutable seq; O(n) if a persistent seq

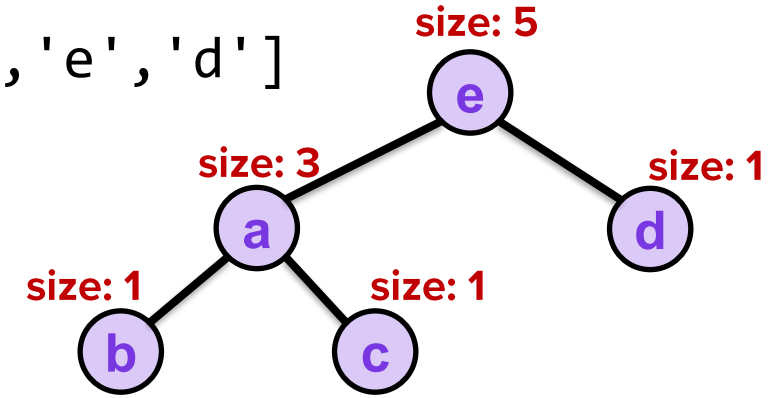
`TreeSequence<T>`:

- Satisfies the `Sequence` interface with `nth`, `subseq` in $O(\log n)$; `length` in $O(1)$
- Supports `append` efficiently in $O(\log n)$

Append is much cheaper!
Use `TreeSequence` for dynamic, append-heavy use cases

TreeSequence Implementations

- Sequences are inherently ordered: How can we represent a `TreeSequence`?
- We can use a size-augmented `BinaryTree` to store a sequence (`TreeSequence`).
  - `S = ['b','a','c','e','d']`



- We will now implement some generic algorithms using a size-augmented binary tree.

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if ______:
        return _______
      else if ______:
        return _______
      else:
        return ______
```

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        return ______
      else if ______:
        return ______
      else:
        return ______
```

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        return _______
      else if i > size(L):
        return _______
      else:
        return _______
```

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        return nth(L, i)
      else if i > size(L):
        return ______
      else:
        return ______
```

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        return nth(L, i)
      else if i > size(L):
        return nth(R, i-size(L)-1)
      else:
        return ______
```

nth, size

```
fun nth(T: BinaryTree<Elem>, i: int) -> Elem:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        return nth(L, i)
      else if i > size(L):
        return nth(R, i-size(L)-1)
      else:
        return e
```

nth is just `select`!

`size()` is the `BinaryTree` size function

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if ______: return ______
    elif ______: return ______
    else:
      mid = ______
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return ______
    elif ______: return ______
    else:
      mid = ______
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif ______: return ______
    else:
      mid = ______
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return ______
    else:
      mid = ______
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return singleton(f(i))
    else:
      mid = ______
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return singleton(f(i))
    else:
      mid = (i+j)/2
      return join_mid(______, ______, ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return singleton(f(i))
    else:
      mid = (i+j)/2
      return join_mid(______, f(mid), ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return singleton(f(i))
    else:
      mid = (i+j)/2
      return join_mid(build(i, mid), f(mid), ______)
  return build(0, n)
```

tabulate

```
fun tabulate(f: int -> Elem, n: int) -> BinaryTree<Elem>:
  fun build(i: int, j: int):
    if i >= j: return Empty
    elif i == j+1: return singleton(f(i))
    else:
      mid = (i+j)/2
      return join_mid(build(i, mid), f(mid), build(mid+1, j))
  return build(0, n)
```

Similar to building a treap or quicksort!!

update

```
fun update(T: BinaryTree<Elem>, i: int, k: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, k', R)):
      if ______:
        return join_mid(______, ______, ______)
      else if ______:
        return join_mid(______, ______, ______)
      else: return ______
```

Very similar to BST `insert`!!

update

```
fun update(T: BinaryTree<Elem>, i: int, e: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e', R)):
      if i < size(L):
        return join_mid(______, ______, ______)
      else if ______:
        return join_mid(______, ______, ______)
      else: return ______
```

Very similar to BST `insert`!!

update

```
fun update(T: BinaryTree<Elem>, i: int, e: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e', R)):
      if i < size(L):
        return join_mid(______, ______, ______)
      else if i > size(L):
        return join_mid(______, ______, ______)
      else: return ______
```

Very similar to BST `insert`!!

update

```
fun update(T: BinaryTree<Elem>, i: int, e: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e', R)):
      if i < size(L):
        return join_mid(update(L, (i, e)), e', R)
      else if i > size(L):
        return join_mid(______, ______, ______)
      else: return ______
```

Very similar to BST `insert`!!

update

```
fun update(T: BinaryTree<Elem>, i: int, e: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e', R)):
      if i < size(L):
        return join_mid(update(L, (i, k)), e', R)
      else if i > size(L):
        return join_mid(L, e', update(R, (i-size(L)-1, e)))
      else: return _______
```

Very similar to BST `insert`!!

update

```
fun update(T: BinaryTree<Elem>, i: int, e: Elem) -> BinaryTree<Elem>:
  match expose(T) with:
    case NONE: raise OutOfBounds
    case SOME(NodeView(L, e', R)):
      if i < size(L):
        return join_mid(update(L, (i, e)), e', R)
      else if i > size(L):
        return join_mid(L, e', update(R, (i-size(L)-1, e)))
      else: return join_mid(L, e, R)
```

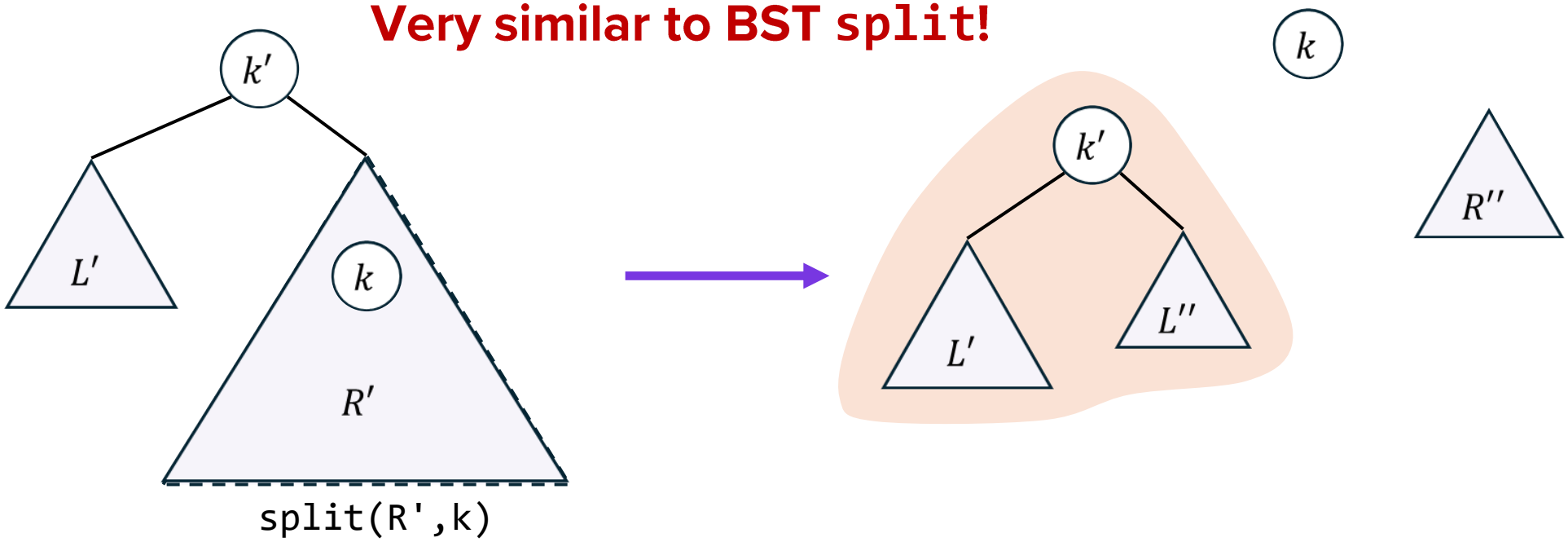
split_at

- `split_at(T, i) -> (BinaryTree<Elem>, BinaryTree<Elem>)`: splits T into two sequences, one containing the elements at index less `i`, the other containing elements with index `>= i`

split_at

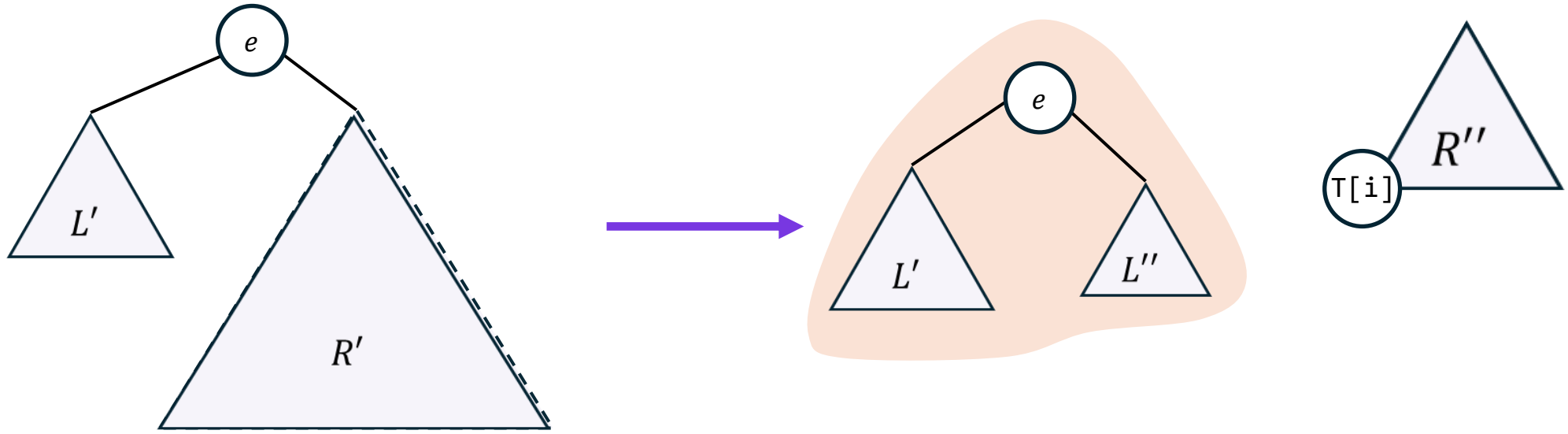
- `split_at(T, i) -> (BinaryTree<Elem>, BinaryTree<Elem>)`: splits T into two sequences, one containing the elements at index less `i`, the other containing elements with index `>= i`

Very similar to BST split!



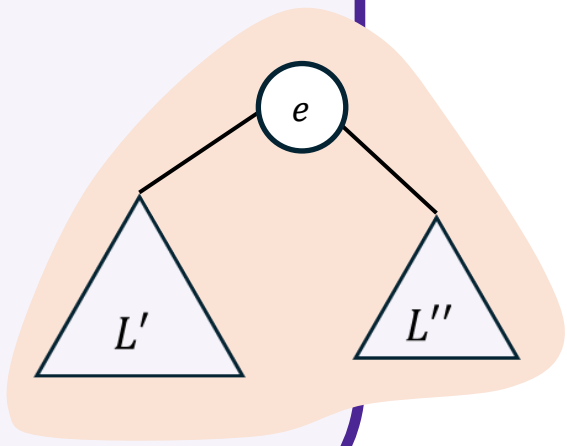
split_at

- `split_at(T, i) -> (BinaryTree<Elem>, BinaryTree<Elem>)`: splits T into two sequences, one containing the elements at index less `i`, the other containing elements with index `>= i`

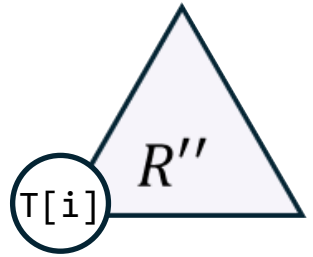


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (______, ______)
    case SOME(NodeView(L, e, R)):
      if ______:
        (_, _) = ______
        return (______, ______)
      else if ______:
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

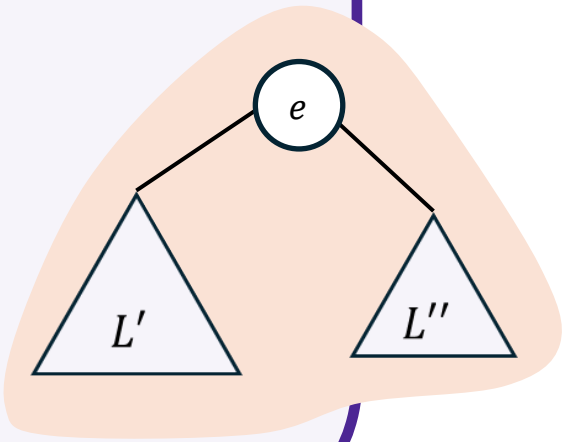


By convention, we add the `i`th value to the right sequence

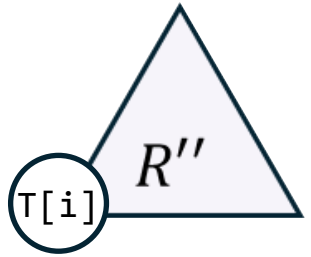


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if ______:
        (_, _) = ______
        return (______, ______)
      else if ______:
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

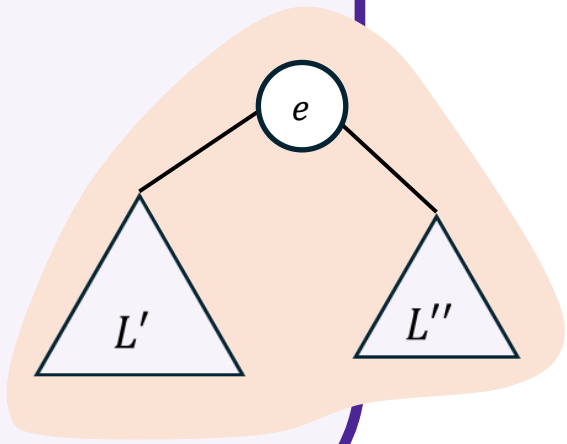


By convention, we add the current value to the right sequence

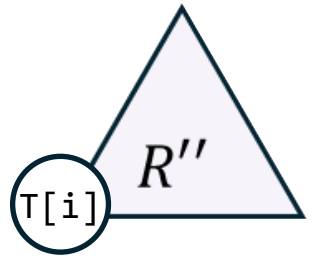


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (_, _) = ______
        return (______, ______)
      else if ______:
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

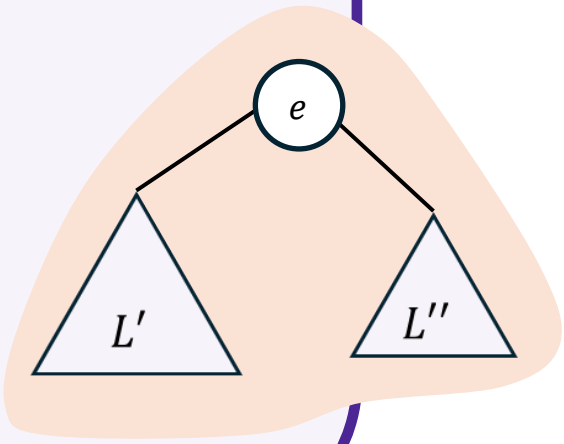


By convention, we add the current value to the right sequence

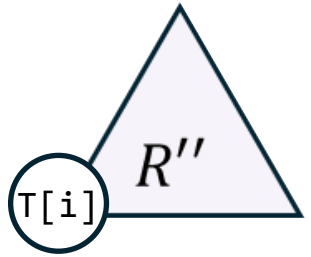


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (_, _) = ______
        return (______, ______)
      else if i > size(L):
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

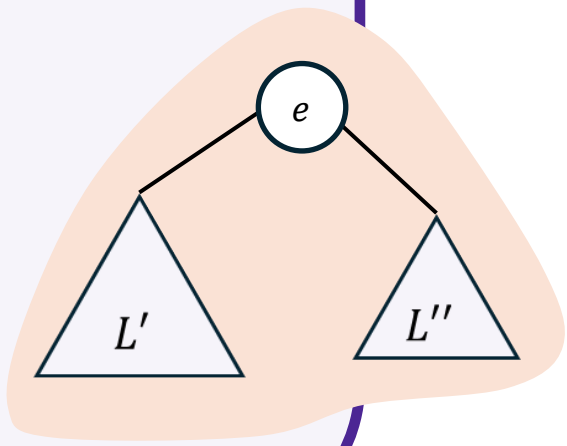


By convention, we add the current value to the right sequence

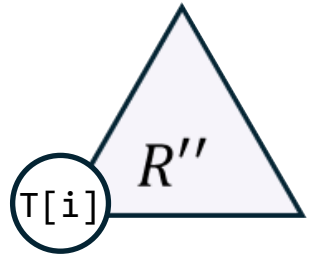


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (______, ______)
      else if i > size(L):
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

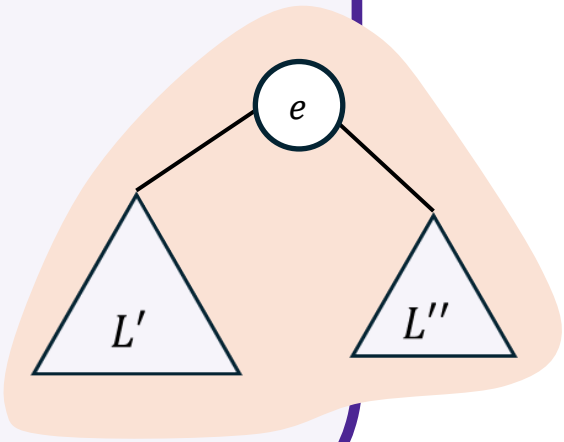


By convention, we add the current value to the right sequence

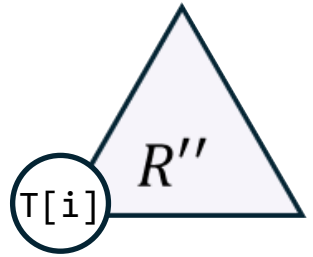


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', ______)
      else if i > size(L):
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```

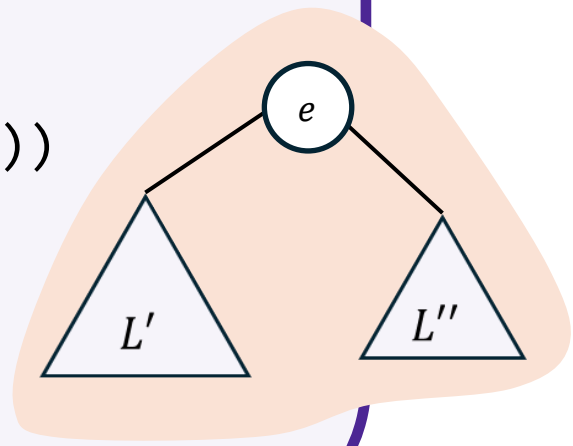


By convention, we add the current value to the right sequence

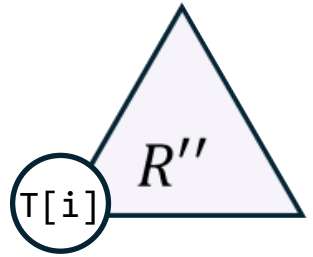


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (_, _) = ______
        return (______, ______)
      else: return (______, ______)
```



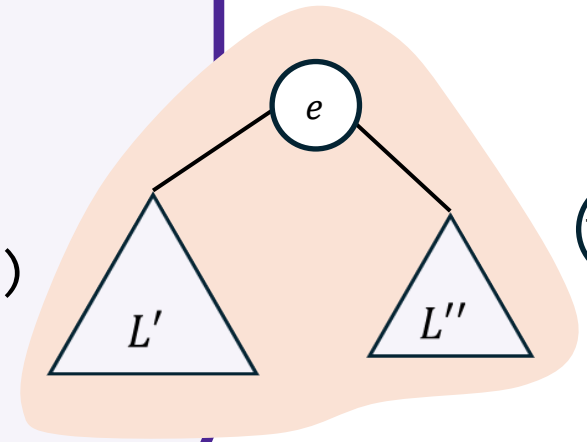
By convention, we add the current value to the right sequence

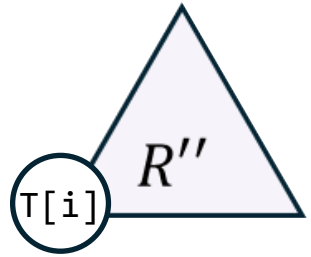


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (L', R') = split_at(R, i-size(L)-1)
        return (______, ______)
      else: return (______, ______)
```

By convention, we add the current value to the right sequence

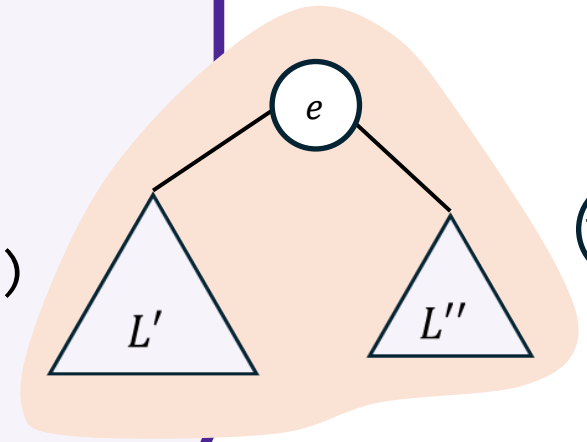


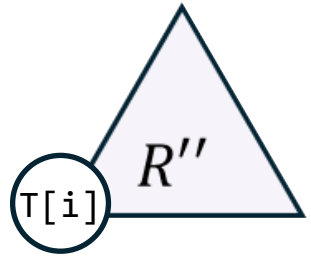


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (L', R') = split_at(R, i-size(L)-1)
        return (______, R')
      else: return (______, ______)
```

By convention, we add the current value to the right sequence

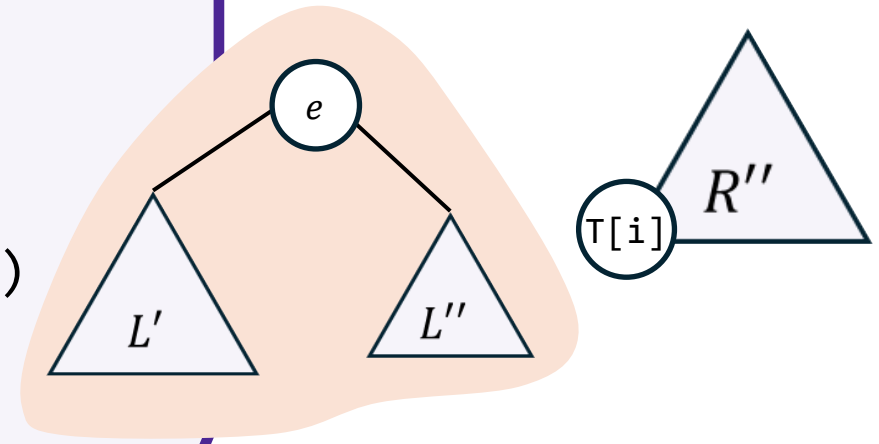




split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (L', R') = split_at(R, i-size(L)-1)
        return (join_mid(L, k, L'), R')
      else: return (______, ______)
```

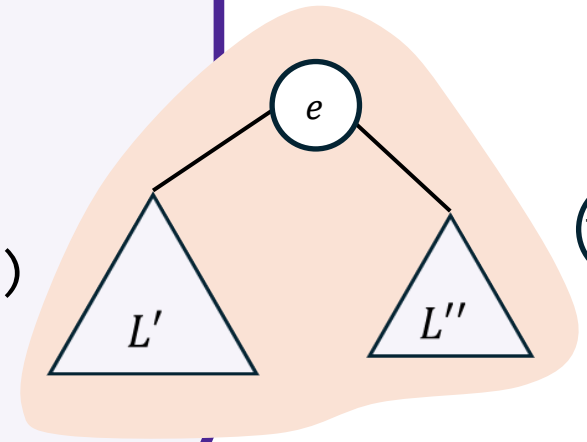
By convention, we add the current value to the right sequence

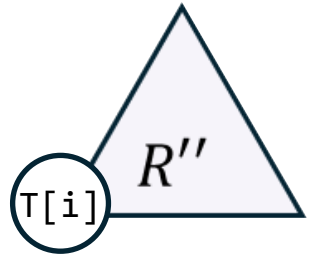


split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (L', R') = split_at(R, i-size(L)-1)
        return (join_mid(L, e, L'), R')
      else: return (L, ______)
```

By convention, we add the current value to the right sequence





split_at

```
fun split_at(T: BinaryTree<Elem>, i: int)
    -> (BinaryTree<Elem>, BinaryTree<Elem>):
  match expose(T) with:
    case NONE: return (Empty, Empty)
    case SOME(NodeView(L, e, R)):
      if i < size(L):
        (L', R') = split_at(L, i)
        return (L', join_mid(R', e, R))
      else if i > size(L):
        (L', R') = split_at(R, i-size(L)-1)
        return (join_mid(L, e, L'), R')
      else: return (L, join_mid(Empty, e, R))
```

By convention, we add the current value to the right sequence

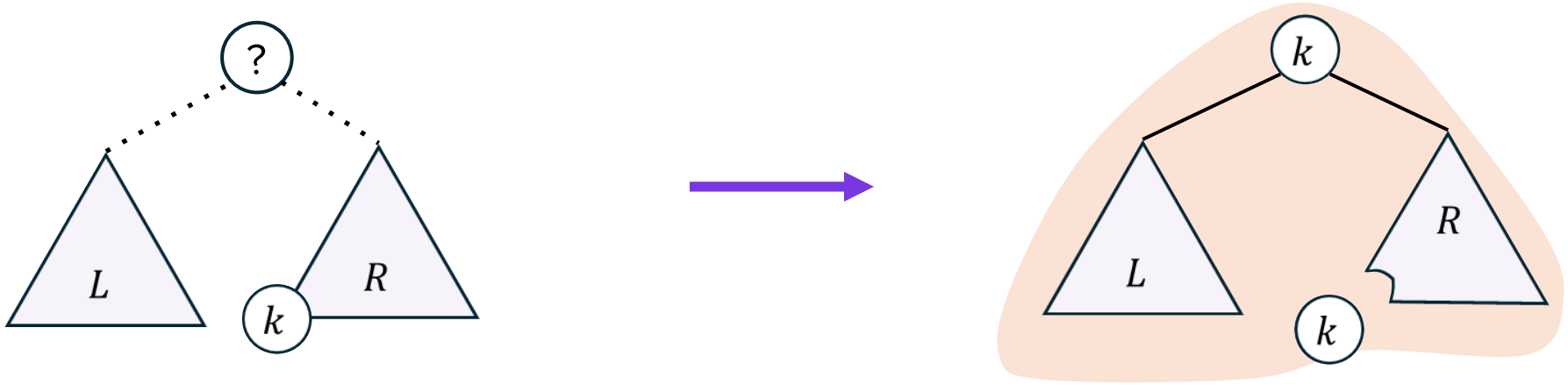
subseq

```
fun subseq(T: BinaryTree<Elem>, i: int, k: int) -> BinaryTree<Elem>:
  (T', _) = split_at(T, i+k)
  (_, T'') = split_at(T', i)
  return T''
```

append

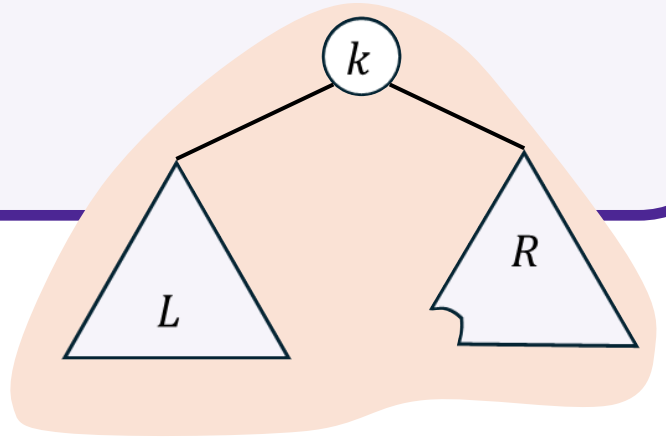
Very similar to BST `join`!!

- `append(A, B) -> BinaryTree<Elem>`: appends sequence B to sequence A and returns the new sequence



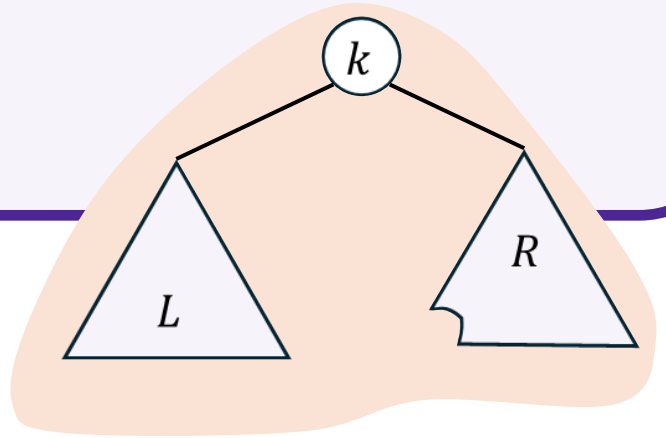
append

```
fun append(A: BinaryTree<Elem>, B: BinaryTree<Elem>)
    -> BinaryTree<Elem>:
  if _____:
    return _____
  else:
    (_, _) = _____
    return _____
```



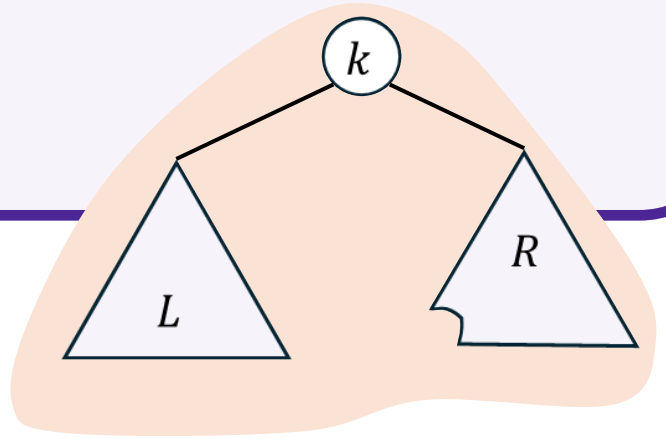
append

```
fun append(A: BinaryTree<Elem>, B: BinaryTree<Elem>)
    -> BinaryTree<Elem>:
  if size(B) == 0 :
    return _____
  else:
    (_, _) = _____
    return _____
```



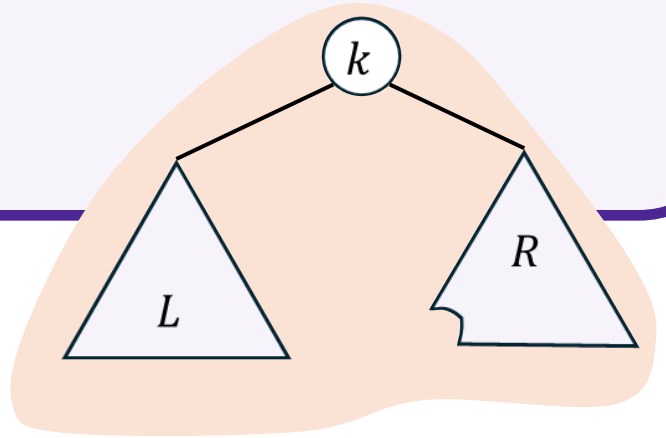
append

```
fun append(A: BinaryTree<Elem>, B: BinaryTree<Elem>)
    -> BinaryTree<Elem>:
  if size(B) == 0 :
    return A
  else:
    (_, _) = _____
    return _____
```



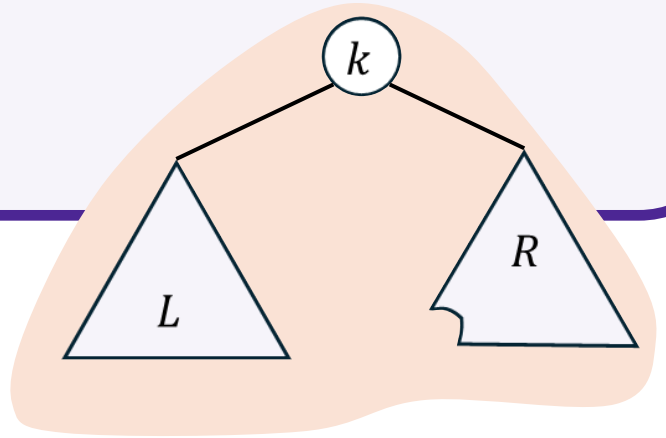
append

```
fun append(A: BinaryTree<Elem>, B: BinaryTree<Elem>)
    -> BinaryTree<Elem>:
  if size(B) == 0 :
    return A
  else:
    (k, R) = split_at(B, 1)
    return _____
```



append

```
fun append(A: BinaryTree<Elem>, B: BinaryTree<Elem>)
    -> BinaryTree<Elem>:
  if size(B) == 0:
    return A
  else:
    (k, R) = split_at(B, 1)
    return join_mid(A, k[0], R)
```



Augmenting with Reduced Values

Motivation

- So far, we've seen many problems that use reduce: MCSS, ParenMatch, copy-left/copy-right problems, Waffle.
- What if we wanted to make them *dynamic*? Consider a code editing program:

```
val ((m1, p1, s1, t1), (m2, p2, s2, t2))
```

- In this case, representing these problems using a **tree-based** data structure, where you can efficiently support insertion and deletion, may be a good idea.

Reduce over Tree Values

- Reducing over the values in a tree takes $O(n)$ work

```
fun reduce(T: AugBST<K,V>, f: (V, V) -> V, I: V):
  match expose(T):
    case NONE: return I
    case SOME(L, (k, V), R)):
      Lred, Rred = parallel(reduce(L), reduce(R))
      return f(f(Lred, V), Rred)
```

- But is there a way to make this much more efficient?

If we store the reduced value at a tree (***augmentation***), access takes $O(1)$.

Introducing AugBSTs

- In this course, we've seen that reduced values are useful for many problems.
- Maybe we store them as augmented values.
- So, we define a general interface for this pattern, parameterized by the combining function.
- This works because the node definition and `makeNode` function look similar no matter the combining function.

AugBSTs Interface

A type `Tree` satisfies the interface `AugBST<K,V,f,I>` if it satisfies the interface `BST<K,V>`, `f: (V,V) -> V` is an associative function, with identity `I: V`, and it supports the following operation:

- `reduced_val(T) -> V`: equivalent to `reduce(f, I, values)`, where values is the values of T ordered by increasing order of their keys.

- The specific AugBST implementation (e.g. AugTreap, AugRBTree, etc.) depends on which BST it is (e.g. Treap, RBTree, etc.)
- Changes to code are simple, similar for all

AugTreap Implementation

New!!

```
type AugTreap<K,V,f,I> =
    Empty
  | Node { left: AugTreap,
           (k,v): (K,V),
           size: int,
           reduced: V,
           right: AugTreap }
```

```
fun reduced_val(T: AugTreap<K,V,f,I>) -> V:
  match T with:
    case Empty:
      return I
    case Node(_,_,_,reduced,_):
      return reduced
```

```
empty: AugTreap<K,V,f,I> = Empty
```

```
fun expose(T: AugTreap<K,V,f,I>)
    -> optional<NodeView<AugTreap<K,V,f,I>>>:
  match T with:
 case Empty: return NONE
 case Node(L,e,_,_,R): return SOME(NodeView(L,e,R))
```

`join_mid` and `size` ignore **`reduced`** (add one more `_` in their `Node(L,e,s,_,R)` pattern)

AugTreap Implementation

```
type AugTreap<K,V,f,I> =
    Empty
  | Node { left: AugTreap,
           (k,v): (K,V),
           size: int,
           reduced: V,
           right: AugTreap }
```

```
fun make_node(L: AugTreap<K,V,f,I>, (k,v): (K,V), R: AugTreap<K,V,f,I>)
    -> AugTreap<K,V,f,I>:
  return Node(L, (k,v), 1 + size(L) + size(R),
                        ________________________, R)
```

AugTreap Implementation

```
type AugTreap<K,V,f,I> =
    Empty
  | Node { left: AugTreap,
           (k,v): (K,V),
           size: int,
           reduced: V,
           right: AugTreap }
```

Computing the size and the reduced value are analogous!

```
fun make_node(L: AugTreap<K,V,f,I>, (k,v): (K,V), R: AugTreap<K,V,f,I>)
    -> AugTreap<K,V,f,I>:
  return Node(L, (k,v), 1 + size(L) + size(R),
                        f(f(reduced_val(L), v), reduced_val(R)), R)
```

Dynamic ParenMatch

- We are given a char sequence `chars` of length n, and we want to support:
- `matched(chars, i, j)`: Is `chars[i:j)` balanced?
- `update(chars, i, c)`: Set `chars[i]` to `c`

```
chars = ab((cd))
chars = a(((cd))
chars = a(((cd))
```

```
matched(chars, 0, 7)   True
update(chars, 1, '(')
matched(chars, 1, 4)   False
```

Dynamic ParenMatch

- Recall our divide-and-conquer recitation strategy:
  - Map `'('` to `(1, 0)`, `')'` to `(0, 1)`
  - Store `(excessLeft, excessRight)` for subseq
  - Combining function:

```
fun combine_paren((i, j), (k, l)) -> (int, int):
  return (i+k-j, l) if j <= k else (i, l+j-k)
```

- How should we define the keys, values, `f`, and `I` for our AugBST?

Dynamic ParenMatch

- How should we define the keys, values, f, and I for our AugBST?
  - Keys: index in the string
  - Values: Mapped value
    - For `'('`: `(1, 0)`
    - For `')'`: `(0, 1)`
    - For everything else: `(0, 0)`
  - `f`: same as before
  - `I`: `(0, 0)`

```
fun combine_paren((i, j), (k, l))
    -> (int, int):
  if j <= k: return (i, l+j-k)
  else: return (i+k-j, l)
```

Dynamic ParenMatch

```
type pm_treap = AugTreap<int,(int,int),combine_parens,(0,0)>
```

- Let's implement `matched` now!
  - We need the range `[i..j)`: What BST function can we use?
    - `ij_treap = get_range(chars, i-1, j)`
    - returns to us an AugTreap solely consisting of values in that range
  - Then what do we need to check?
    - Whether the reduced value at the root is (0, 0)
    - `reduced_val(ij_treap) == (0, 0)`

Dynamic ParenMatch

```
type pm_treap = AugTreap<int, (int,int), combine_parens, (0,0)>

fun matched(chars: pm_treap, i: int, j: int) -> bool:
  ij_treap = get_range(chars, i-1, j)
  return reduced_val(ij_treap) == (0, 0)
```

Dynamic ParenMatch

- `pm_treap = AugTreap<int,(int,int),combine_parens,(0,0)>`
- Let's implement `update` now!
  - Need to update index `i` to character `c`
  - Ideas:
    - Using dictionary `insert` function

Dynamic ParenMatch

```
type pm_treap = AugTreap<int, (int,int), combine_parens, (0,0)>

fun matched(chars: pm_treap, i: int, j: int) -> bool:
  ij_treap = get_range(chars, i-1, j)
  return reduced_val(ij_treap) == (0, 0)

fun update(chars: pm_treap, i, c) -> bool:
  return insert(chars, (i, c))
```

Summary

- Size-augmentation allows us to implement efficient order statistics operations
- Size-augmentation also allows us to implement sequences using trees (TreeSequence)
- Reduced value augmentation can be generalized into **AugBSTs**
- AugBSTs are useful for dynamic versions of problems that use `reduce` with custom combining functions