2.5. dict.ak

//// A module for working with bytearray dictionaries.

////

//// ### Important

////

//// Dictionaries are \*\*ordered sets\*\* of key-value pairs, which thus

//// preserve some invariants. Specifically, each key is only present once in

//// the dictionary and all keys are stored in ascending lexicographic order.

////

//// These invariants allow for more optimized functions to operate on `Dict`,

//// but as a trade-offs, prevent `Dict` from being serializable. To recover a `Dict`

//// from an unknown `Data`, you must first recover an `Pairs<k, v>` and use

//// `dict.from\_ascending\_list`.

use aiken/builtin

/// An opaque `Dict`. The type is opaque because the module maintains some

/// invariant, namely: there's only one occurrence of a given key in the dictionary.

///

/// Note that the `key` parameter is a phantom-type, and only present as a

/// means of documentation. Keys can be any type, yet will need to comparable

/// to use functions like `insert`.

///

/// See for example:

///

/// ```aiken

/// pub type Value =

/// Dict<PolicyId, Dict<AssetName, Int>>

/// ```

pub opaque type Dict<key, value> {

inner: Pairs<ByteArray, value>,

}

/// Create a new empty Dict

/// ```aiken

/// dict.to\_pairs(dict.new()) == []

/// ```

pub fn new() -> Dict<key, value> {

Dict { inner: [] }

}

const foo = #"666f6f"

const bar = #"626172"

const baz = #"62617a"

fn fixture\_1() {

new()

|> insert(foo, 42)

|> insert(bar, 14)

}

/// Remove a key-value pair from the dictionary. If the key is not found, no changes are made.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 100)

/// |> dict.insert(key: "b", value: 200)

/// |> dict.delete(key: "a")

/// |> dict.to\_pairs()

///

/// result == [Pair("b", 200)]

/// ```

pub fn delete(self: Dict<key, value>, key: ByteArray) -> Dict<key, value> {

Dict { inner: do\_delete(self.inner, key) }

}

fn do\_delete(

self: Pairs<ByteArray, value>,

key k: ByteArray,

) -> Pairs<ByteArray, value> {

when self is {

[] ->

[]

[Pair(k2, v2), ..rest] ->

if builtin.less\_than\_equals\_bytearray(k, k2) {

if k == k2 {

rest

} else {

self

}

} else {

[Pair(k2, v2), ..do\_delete(rest, k)]

}

}

}

test delete\_1() {

delete(new(), foo) == new()

}

test delete\_2() {

let m =

new()

|> insert(foo, 14)

delete(m, foo) == new()

}

test delete\_3() {

let m =

new()

|> insert(foo, 14)

delete(m, bar) == m

}

test delete\_4() {

let m =

new()

|> insert(foo, 14)

|> insert(bar, 14)

!has\_key(delete(m, foo), foo)

}

test delete\_5() {

let m =

new()

|> insert(foo, 14)

|> insert(bar, 14)

has\_key(delete(m, bar), foo)

}

test delete\_6() {

let m =

new()

|> insert("aaa", 1)

|> insert("bbb", 2)

|> insert("ccc", 3)

|> insert("ddd", 4)

|> insert("eee", 5)

|> insert("fff", 6)

|> insert("ggg", 7)

|> insert("hhh", 8)

|> insert("iii", 9)

|> insert("jjj", 10)

delete(m, "bcd") == m

}

/// Keep only the key-value pairs that pass the given predicate.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 100)

/// |> dict.insert(key: "b", value: 200)

/// |> dict.insert(key: "c", value: 300)

/// |> dict.filter(fn(k, \_v) { k != "a" })

/// |> dict.to\_pairs()

///

/// result == [Pair("b", 200), Pair("c", 300)]

/// ```

pub fn filter(

self: Dict<key, value>,

with: fn(ByteArray, value) -> Bool,

) -> Dict<key, value> {

Dict { inner: do\_filter(self.inner, with) }

}

fn do\_filter(

self: Pairs<ByteArray, value>,

with: fn(ByteArray, value) -> Bool,

) -> Pairs<ByteArray, value> {

when self is {

[] ->

[]

[Pair(k, v), ..rest] ->

if with(k, v) {

[Pair(k, v), ..do\_filter(rest, with)]

} else {

do\_filter(rest, with)

}

}

}

test filter\_1() {

filter(new(), fn(\_, \_) { True }) == new()

}

test filter\_2() {

let expected =

new()

|> insert(foo, 42)

filter(fixture\_1(), fn(\_, v) { v > 14 }) == expected

}

test filter\_3() {

let expected =

new()

|> insert(bar, 14)

filter(fixture\_1(), fn(k, \_) { k == bar }) == expected

}

/// Finds a value in the dictionary, and returns the first key found to have that value.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 42)

/// |> dict.insert(key: "b", value: 14)

/// |> dict.insert(key: "c", value: 42)

/// |> dict.find(42)

///

/// result == Some("a")

/// ```

pub fn find(self: Dict<key, value>, value v: value) -> Option<ByteArray> {

do\_find(self.inner, v)

}

fn do\_find(self: Pairs<ByteArray, value>, value v: value) -> Option<ByteArray> {

when self is {

[] -> None

[Pair(k2, v2), ..rest] ->

if v == v2 {

Some(k2)

} else {

do\_find(rest, v)

}

}

}

test find\_1() {

find(new(), foo) == None

}

test find\_2() {

find(

new()

|> insert(foo, 14),

14,

) == Some(foo)

}

test find\_3() {

find(

new()

|> insert(foo, 14),

42,

) == None

}

test find\_4() {

find(

new()

|> insert(foo, 14)

|> insert(bar, 42)

|> insert(baz, 14),

14,

) == Some(baz)

}

/// Fold over the key-value pairs in a dictionary. The fold direction follows keys

/// in ascending order and is done from right-to-left.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 100)

/// |> dict.insert(key: "b", value: 200)

/// |> dict.insert(key: "c", value: 300)

/// |> dict.foldr(0, fn(\_k, v, r) { v + r })

///

/// result == 600

/// ```

pub fn foldr(

self: Dict<key, value>,

zero: result,

with: fn(ByteArray, value, result) -> result,

) -> result {

do\_foldr(self.inner, zero, with)

}

fn do\_foldr(

self: Pairs<ByteArray, value>,

zero: result,

with: fn(ByteArray, value, result) -> result,

) -> result {

when self is {

[] -> zero

[Pair(k, v), ..rest] -> with(k, v, do\_foldr(rest, zero, with))

}

}

test foldr\_1() {

foldr(new(), 14, fn(\_, \_, \_) { 42 }) == 14

}

test foldr\_2() {

foldr(fixture\_1(), zero: 0, with: fn(\_, v, total) { v + total }) == 56

}

/// Fold over the key-value pairs in a dictionary. The fold direction follows keys

/// in ascending order and is done from left-to-right.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 100)

/// |> dict.insert(key: "b", value: 200)

/// |> dict.insert(key: "c", value: 300)

/// |> dict.foldl(0, fn(\_k, v, r) { v + r })

///

/// result == 600

/// ```

pub fn foldl(

self: Dict<key, value>,

zero: result,

with: fn(ByteArray, value, result) -> result,

) -> result {

do\_foldl(self.inner, zero, with)

}

fn do\_foldl(

self: Pairs<ByteArray, value>,

zero: result,

with: fn(ByteArray, value, result) -> result,

) -> result {

when self is {

[] -> zero

[Pair(k, v), ..rest] -> do\_foldl(rest, with(k, v, zero), with)

}

}

test fold\_1() {

foldl(new(), 14, fn(\_, \_, \_) { 42 }) == 14

}

test fold\_2() {

foldl(fixture\_1(), zero: 0, with: fn(\_, v, total) { v + total }) == 56

}

/// Construct a dictionary from a list of key-value pairs. Note that when a key is present

/// multiple times, the first occurrence prevails.

///

/// ```aiken

/// let pairs = [Pair("a", 100), Pair("c", 300), Pair("b", 200)]

///

/// let result =

/// dict.from\_pairs(pairs)

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

/// ```

pub fn from\_pairs(self: Pairs<ByteArray, value>) -> Dict<key, value> {

Dict { inner: do\_from\_pairs(self) }

}

fn do\_from\_pairs(xs: Pairs<ByteArray, value>) -> Pairs<ByteArray, value> {

when xs is {

[] ->

[]

[Pair(k, v), ..rest] -> do\_insert(do\_from\_pairs(rest), k, v)

}

}

test from\_list\_1() {

from\_pairs([]) == new()

}

test from\_list\_2() {

from\_pairs([Pair(foo, 42), Pair(bar, 14)]) == from\_pairs(

[Pair(bar, 14), Pair(foo, 42)],

)

}

test from\_list\_3() {

from\_pairs([Pair(foo, 42), Pair(bar, 14)]) == fixture\_1()

}

test from\_list\_4() {

from\_pairs([Pair(foo, 42), Pair(bar, 14), Pair(foo, 1337)]) == fixture\_1()

}

test bench\_from\_pairs() {

let dict =

from\_pairs(

[

Pair("bbba", 8),

Pair("bbab", 12),

Pair("aabb", 13),

Pair("aaab", 9),

Pair("bbbb", 16),

Pair("aaaa", 1),

Pair("aaba", 5),

Pair("abab", 10),

Pair("baba", 7),

Pair("baab", 11),

Pair("abaa", 2),

Pair("baaa", 3),

Pair("bbaa", 4),

Pair("babb", 15),

Pair("abbb", 14),

Pair("abba", 6),

],

)

size(dict) == 16

}

/// Like ['from\_list'](from\_list), but from an already sorted list by ascending

/// keys. This function fails (i.e. halt the program execution) if the list isn't

/// sorted.

///

/// ```aiken

/// let pairs = [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

///

/// let result =

/// dict.from\_ascending\_pairs(pairs)

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

/// ```

///

/// This is meant to be used to turn a list constructed off-chain into a `Dict`

/// which has taken care of maintaining interval invariants. This function still

/// performs a sanity check on all keys to avoid silly mistakes. It is, however,

/// considerably faster than ['from\_list'](from\_list)

pub fn from\_ascending\_pairs(xs: Pairs<ByteArray, value>) -> Dict<key, value> {

let Void = check\_ascending\_list(xs)

Dict { inner: xs }

}

fn check\_ascending\_list(xs: Pairs<ByteArray, value>) {

when xs is {

[] -> Void

[\_] -> Void

[Pair(x0, \_), Pair(x1, \_) as e, ..rest] ->

if builtin.less\_than\_bytearray(x0, x1) {

check\_ascending\_list([e, ..rest])

} else {

fail @"keys in associative list aren't in ascending order"

}

}

}

/// Like [`from\_ascending\_pairs`](#from\_ascending\_list) but fails if \*\*any\*\*

/// value doesn't satisfy the predicate.

///

/// ```aiken

/// let pairs = [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

///

/// dict.from\_ascending\_pairs\_with(pairs, fn(x) { x <= 250 }) // fail

/// ```

pub fn from\_ascending\_pairs\_with(

xs: Pairs<ByteArray, value>,

predicate: fn(value) -> Bool,

) -> Dict<key, value> {

let Void = check\_ascending\_pairs\_with(xs, predicate)

Dict { inner: xs }

}

fn check\_ascending\_pairs\_with(

xs: Pairs<ByteArray, value>,

predicate: fn(value) -> Bool,

) {

when xs is {

[] -> Void

[Pair(\_, v)] ->

if predicate(v) {

Void

} else {

fail @"value doesn't satisfy predicate"

}

[Pair(x0, v0), Pair(x1, \_) as e, ..rest] ->

if builtin.less\_than\_bytearray(x0, x1) {

if predicate(v0) {

check\_ascending\_pairs\_with([e, ..rest], predicate)

} else {

fail @"value doesn't satisfy predicate"

}

} else {

fail @"keys in pairs aren't in ascending order"

}

}

}

test bench\_from\_ascending\_pairs() {

let dict =

from\_ascending\_pairs(

[

Pair("aaaa", 1),

Pair("aaab", 9),

Pair("aaba", 5),

Pair("aabb", 13),

Pair("abaa", 2),

Pair("abab", 10),

Pair("abba", 6),

Pair("abbb", 14),

Pair("baaa", 3),

Pair("baab", 11),

Pair("baba", 7),

Pair("babb", 15),

Pair("bbaa", 4),

Pair("bbab", 12),

Pair("bbba", 8),

Pair("bbbb", 16),

],

)

size(dict) == 16

}

/// Get a value in the dict by its key.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: "Aiken")

/// |> dict.get(key: "a")

///

/// result == Some("Aiken")

/// ```

pub fn get(self: Dict<key, value>, key: ByteArray) -> Option<value> {

do\_get(self.inner, key)

}

fn do\_get(self: Pairs<ByteArray, value>, key k: ByteArray) -> Option<value> {

when self is {

[] -> None

[Pair(k2, v), ..rest] ->

if builtin.less\_than\_equals\_bytearray(k, k2) {

if k == k2 {

Some(v)

} else {

None

}

} else {

do\_get(rest, k)

}

}

}

test get\_1() {

get(new(), foo) == None

}

test get\_2() {

let m =

new()

|> insert(foo, "Aiken")

|> insert(bar, "awesome")

get(m, key: foo) == Some("Aiken")

}

test get\_3() {

let m =

new()

|> insert(foo, "Aiken")

|> insert(bar, "awesome")

get(m, key: baz) == None

}

test get\_4() {

let m =

new()

|> insert("aaa", "1")

|> insert("bbb", "2")

|> insert("ccc", "3")

|> insert("ddd", "4")

|> insert("eee", "5")

|> insert("fff", "6")

|> insert("ggg", "7")

|> insert("hhh", "8")

|> insert("iii", "9")

|> insert("jjj", "10")

get(m, "bcd") == None

}

test get\_5() {

let m =

new()

|> insert("aaa", "1")

|> insert("bbb", "2")

|> insert("ccc", "3")

|> insert("ddd", "4")

|> insert("eee", "5")

|> insert("fff", "6")

|> insert("ggg", "7")

|> insert("hhh", "8")

|> insert("iii", "9")

|> insert("jjj", "10")

get(m, "kkk") == None

}

/// Check if a key exists in the dictionary.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: "Aiken")

/// |> dict.has\_key("a")

///

/// result == True

/// ```

pub fn has\_key(self: Dict<key, value>, key k: ByteArray) -> Bool {

do\_has\_key(self.inner, k)

}

fn do\_has\_key(self: Pairs<ByteArray, value>, key k: ByteArray) -> Bool {

when self is {

[] -> False

[Pair(k2, \_), ..rest] ->

if builtin.less\_than\_equals\_bytearray(k, k2) {

k == k2

} else {

do\_has\_key(rest, k)

}

}

}

test has\_key\_1() {

!has\_key(new(), foo)

}

test has\_key\_2() {

has\_key(

new()

|> insert(foo, 14),

foo,

)

}

test has\_key\_3() {

!has\_key(

new()

|> insert(foo, 14),

bar,

)

}

test has\_key\_4() {

has\_key(

new()

|> insert(foo, 14)

|> insert(bar, 42),

bar,

)

}

/// Insert a value in the dictionary at a given key. If the key already exists, its value is \*\*overridden\*\*. If you need ways to combine keys together, use (`insert\_with`)[#insert\_with].

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert(key: "a", value: 1)

/// |> dict.insert(key: "b", value: 2)

/// |> dict.insert(key: "a", value: 3)

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 3), Pair("b", 2)]

/// ```

pub fn insert(

self: Dict<key, value>,

key k: ByteArray,

value v: value,

) -> Dict<key, value> {

Dict { inner: do\_insert(self.inner, k, v) }

}

fn do\_insert(

self: Pairs<ByteArray, value>,

key k: ByteArray,

value v: value,

) -> Pairs<ByteArray, value> {

when self is {

[] ->

[Pair(k, v)]

[Pair(k2, v2), ..rest] ->

if builtin.less\_than\_bytearray(k, k2) {

[Pair(k, v), ..self]

} else {

if k == k2 {

[Pair(k, v), ..rest]

} else {

[Pair(k2, v2), ..do\_insert(rest, k, v)]

}

}

}

}

test insert\_1() {

let m1 =

new()

|> insert(foo, 42)

let m2 =

new()

|> insert(foo, 14)

insert(m1, foo, 14) == m2

}

test insert\_2() {

let m1 =

new()

|> insert(foo, 42)

let m2 =

new()

|> insert(bar, 14)

insert(m1, bar, 14) == insert(m2, foo, 42)

}

/// Insert a value in the dictionary at a given key. When the key already exist, the provided

/// merge function is called. The value existing in the dictionary is passed as the second argument

/// to the merge function, and the new value is passed as the third argument.

///

/// ```aiken

/// let sum =

/// fn (\_k, a, b) { Some(a + b) }

///

/// let result =

/// dict.new()

/// |> dict.insert\_with(key: "a", value: 1, with: sum)

/// |> dict.insert\_with(key: "b", value: 2, with: sum)

/// |> dict.insert\_with(key: "a", value: 3, with: sum)

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 4), Pair("b", 2)]

/// ```

pub fn insert\_with(

self: Dict<key, value>,

key k: ByteArray,

value v: value,

with: fn(ByteArray, value, value) -> Option<value>,

) -> Dict<key, value> {

Dict {

inner: do\_insert\_with(self.inner, k, v, fn(k, v1, v2) { with(k, v2, v1) }),

}

}

test insert\_with\_1() {

let sum =

fn(\_k, a, b) { Some(a + b) }

let result =

new()

|> insert\_with(key: "foo", value: 1, with: sum)

|> insert\_with(key: "bar", value: 2, with: sum)

|> to\_pairs()

result == [Pair("bar", 2), Pair("foo", 1)]

}

test insert\_with\_2() {

let sum =

fn(\_k, a, b) { Some(a + b) }

let result =

new()

|> insert\_with(key: "foo", value: 1, with: sum)

|> insert\_with(key: "bar", value: 2, with: sum)

|> insert\_with(key: "foo", value: 3, with: sum)

|> to\_pairs()

result == [Pair("bar", 2), Pair("foo", 4)]

}

test insert\_with\_3() {

let with =

fn(k, a, \_b) {

if k == "foo" {

Some(a)

} else {

None

}

}

let result =

new()

|> insert\_with(key: "foo", value: 1, with: with)

|> insert\_with(key: "bar", value: 2, with: with)

|> insert\_with(key: "foo", value: 3, with: with)

|> insert\_with(key: "bar", value: 4, with: with)

|> to\_pairs()

result == [Pair("foo", 1)]

}

/// Efficiently checks whether a dictionary is empty.

/// ```aiken

/// dict.is\_empty(dict.new()) == True

/// ```

pub fn is\_empty(self: Dict<key, value>) -> Bool {

when self.inner is {

[] -> True

\_ -> False

}

}

test is\_empty\_1() {

is\_empty(new())

}

/// Extract all the keys present in a given `Dict`.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert("a", 14)

/// |> dict.insert("b", 42)

/// |> dict.insert("a", 1337)

/// |> dict.keys()

///

/// result == ["a", "b"]

/// ```

pub fn keys(self: Dict<key, value>) -> List<ByteArray> {

do\_keys(self.inner)

}

fn do\_keys(self: Pairs<ByteArray, value>) -> List<ByteArray> {

when self is {

[] ->

[]

[Pair(k, \_), ..rest] ->

[k, ..do\_keys(rest)]

}

}

test keys\_1() {

keys(new()) == []

}

test keys\_2() {

keys(

new()

|> insert(foo, 0)

|> insert(bar, 0),

) == [bar, foo]

}

/// Apply a function to all key-value pairs in a Dict.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert("a", 100)

/// |> dict.insert("b", 200)

/// |> dict.insert("c", 300)

/// |> dict.map(fn(\_k, v) { v \* 2 })

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 200), Pair("b", 400), Pair("c", 600)]

/// ```

pub fn map(self: Dict<key, a>, with: fn(ByteArray, a) -> b) -> Dict<key, b> {

Dict { inner: do\_map(self.inner, with) }

}

fn do\_map(

self: Pairs<ByteArray, a>,

with: fn(ByteArray, a) -> b,

) -> Pairs<ByteArray, b> {

when self is {

[] ->

[]

[Pair(k, v), ..rest] ->

[Pair(k, with(k, v)), ..do\_map(rest, with)]

}

}

test map\_1() {

let result =

fixture\_1()

|> map(with: fn(k, \_) { k })

get(result, foo) == Some(foo)

}

test map\_2() {

let result =

fixture\_1()

|> map(with: fn(\_, v) { v + 1 })

get(result, foo) == Some(43) && size(result) == size(fixture\_1())

}

/// Get the inner list holding the dictionary data.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert("a", 100)

/// |> dict.insert("b", 200)

/// |> dict.insert("c", 300)

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

/// ```

pub fn to\_pairs(self: Dict<key, value>) -> Pairs<ByteArray, value> {

self.inner

}

test to\_list\_1() {

to\_pairs(new()) == []

}

test to\_list\_2() {

to\_pairs(fixture\_1()) == [Pair(bar, 14), Pair(foo, 42)]

}

/// Return the number of key-value pairs in the dictionary.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert("a", 100)

/// |> dict.insert("b", 200)

/// |> dict.insert("c", 300)

/// |> dict.size()

///

/// result == 3

/// ```

pub fn size(self: Dict<key, value>) -> Int {

do\_size(self.inner)

}

fn do\_size(self: Pairs<key, value>) -> Int {

when self is {

[] -> 0

[\_, ..rest] -> 1 + do\_size(rest)

}

}

test size\_1() {

size(new()) == 0

}

test size\_2() {

size(

new()

|> insert(foo, 14),

) == 1

}

test size\_3() {

size(

new()

|> insert(foo, 14)

|> insert(bar, 42),

) == 2

}

/// Combine two dictionaries. If the same key exist in both the left and

/// right dictionary, values from the left are preferred (i.e. left-biaised).

///

/// ```aiken

/// let left\_dict = dict.from\_pairs([Pair("a", 100), Pair("b", 200)])

/// let right\_dict = dict.from\_pairs([Pair("a", 150), Pair("c", 300)])

///

/// let result =

/// dict.union(left\_dict, right\_dict) |> dict.to\_pairs()

///

/// result == [Pair("a", 100), Pair("b", 200), Pair("c", 300)]

/// ```

pub fn union(

left: Dict<key, value>,

right: Dict<key, value>,

) -> Dict<key, value> {

Dict { inner: do\_union(left.inner, right.inner) }

}

fn do\_union(

left: Pairs<ByteArray, value>,

right: Pairs<ByteArray, value>,

) -> Pairs<ByteArray, value> {

when left is {

[] -> right

[Pair(k, v), ..rest] -> do\_union(rest, do\_insert(right, k, v))

}

}

test union\_1() {

union(fixture\_1(), new()) == fixture\_1()

}

test union\_2() {

union(new(), fixture\_1()) == fixture\_1()

}

test union\_3() {

let left =

new()

|> insert(foo, 14)

let right =

new()

|> insert(bar, 42)

|> insert(baz, 1337)

union(left, right) == from\_pairs(

[Pair(foo, 14), Pair(baz, 1337), Pair(bar, 42)],

)

}

test union\_4() {

let left =

new()

|> insert(foo, 14)

let right =

new()

|> insert(bar, 42)

|> insert(foo, 1337)

union(left, right) == from\_pairs([Pair(foo, 14), Pair(bar, 42)])

}

/// Like [`union`](#union) but allows specifying the behavior to adopt when a key is present

/// in both dictionaries. The first value received correspond to the value in the left

/// dictionnary, whereas the second argument corresponds to the value in the right dictionnary.

///

/// When passing `None`, the value is removed and not present in the union.

///

/// ```aiken

/// let left\_dict = dict.from\_pairs([Pair("a", 100), Pair("b", 200)])

/// let right\_dict = dict.from\_pairs([Pair("a", 150), Pair("c", 300)])

///

/// let result =

/// dict.union\_with(

/// left\_dict,

/// right\_dict,

/// fn(\_k, v1, v2) { Some(v1 + v2) },

/// )

/// |> dict.to\_pairs()

///

/// result == [Pair("a", 250), Pair("b", 200), Pair("c", 300)]

/// ```

pub fn union\_with(

left: Dict<key, value>,

right: Dict<key, value>,

with: fn(ByteArray, value, value) -> Option<value>,

) -> Dict<key, value> {

Dict { inner: do\_union\_with(left.inner, right.inner, with) }

}

fn do\_union\_with(

left: Pairs<ByteArray, value>,

right: Pairs<ByteArray, value>,

with: fn(ByteArray, value, value) -> Option<value>,

) -> Pairs<ByteArray, value> {

when left is {

[] -> right

[Pair(k, v), ..rest] ->

do\_union\_with(rest, do\_insert\_with(right, k, v, with), with)

}

}

fn do\_insert\_with(

self: Pairs<ByteArray, value>,

key k: ByteArray,

value v: value,

with: fn(ByteArray, value, value) -> Option<value>,

) -> Pairs<ByteArray, value> {

when self is {

[] ->

[Pair(k, v)]

[Pair(k2, v2), ..rest] ->

if builtin.less\_than\_bytearray(k, k2) {

[Pair(k, v), ..self]

} else {

if k == k2 {

when with(k, v, v2) is {

Some(combined) ->

[Pair(k, combined), ..rest]

None -> rest

}

} else {

[Pair(k2, v2), ..do\_insert\_with(rest, k, v, with)]

}

}

}

}

test union\_with\_1() {

let left =

new()

|> insert(foo, 14)

let right =

new()

|> insert(bar, 42)

|> insert(foo, 1337)

let result = union\_with(left, right, with: fn(\_, l, r) { Some(l + r) })

result == from\_pairs([Pair(foo, 1351), Pair(bar, 42)])

}

/// Extract all the values present in a given `Dict`.

///

/// ```aiken

/// let result =

/// dict.new()

/// |> dict.insert("a", 14)

/// |> dict.insert("b", 42)

/// |> dict.insert("c", 1337)

/// |> dict.values()

///

/// result == [1337, 42]

/// ```

pub fn values(self: Dict<key, value>) -> List<value> {

do\_values(self.inner)

}

fn do\_values(self: Pairs<key, value>) -> List<value> {

when self is {

[] ->

[]

[Pair(\_, v), ..rest] ->

[v, ..do\_values(rest)]

}

}

test values\_1() {

values(new()) == []

}

test values\_2() {

values(

new()

|> insert(foo, 3)

|> insert(bar, 4),

) == [4, 3]

}