# Fmlib

**Helmut Brandl**

**Feb 14, 2024**

# CONTENTS:


**1 Data Structures** **3**
- 1.1 Array . . . 3
  - 1.1.1 Binary Search . . . 3
- 1.2 Radix Balanced Array . . . 6
  - 1.2.1 Basic Idea . . . 6
  - 1.2.2 Data Structure . . . 7
  - 1.2.3 Invariant . . . 8
  - 1.2.4 Element Retrieval . . . 9
  - 1.2.5 Element Replacement . . . 10
  - 1.2.6 Element Insertion at the Rear End . . . 10
  - 1.2.7 Element Deletion at the Rear End . . . 12
- 1.3 B Tree . . . 13
  - 1.3.1 General . . . 13
  - 1.3.2 Search . . . 16
  - 1.3.3 Insertion . . . 16
  - 1.3.4 Deletion . . . 25

**2 Combinator Parsing** **33**
- 2.1 Layout Parsing . . . 33
  - 2.1.1 Parsing Expression Grammar . . . 33
  - 2.1.2 Formal Semantics . . . 34
  - 2.1.3 Implementation . . . 38

**Bibliography** **41**

This document describes the design and implementation of some datastructures and algorithms of `Fmlib`. It does not describe the API from a user's perspective. Find the documentation of the API from a user's perspective here

CHAPTER

ONE

# DATA STRUCTURES

## 1.1 Array

### 1.1.1 Binary Search

#### Specification

Binary search is an algorithm to find elements in an array or array like structure which contains the elements in ascending order. In an array like structure elements can be retrieved very fast by providing the index of the element. The first index is `0`, the second is `1` and the last is `length arr - 1`.

The array must not contain duplicates, otherwise the algorithm does not work as expected. I.e.

```
k0 < k1 < ... < km
```

must be satisfied where `ki` is the `i`th key in the array and the array has length `m + 1`.

The binary search algorithm has the interface:

```
let binsearch
    (compare: 'key -> 'key -> int)      (* Comparison function. *)
    (key_of:  'item -> 'key)            (* Extract a key from an item. *)
    (key:     'key)                     (* The key to be found. *)
    (arr:     'item array)              (* The array to be search in. *)
    : int * bool
    =
    ...
```

The search key is in many cases only a part of the items in the array. Therefore we need a function `key_of` to extract the search key from an item. The comparison function `compare` defines the order of the keys with the standard semantics used in ocaml

```
compare a b < 0                 if and only if  a < b
compare a b = 0                 if and only if  a = b
compare a b > 0                 if and only if  a > b
```

It is required that the comparison function defines a linear order for the keys.

The function `binsearch` returns the pair `(i, exact)`, where `i` is the index of the key and the flag `exact` indicates that an exact match has been found.

The function might return `(len, false)` where `len` is the length of the array. Note that `len` is not a valid index into the array because the index of the last element is `len - 1`. For our reasoning we assume a fictitious element `arr.(len)` which has the value `+infinity`. Then the return value `i` satisfies the specification:

```
if exact then
    key = arr.(i)
else
    key < arr.(i)

arr.(j) < key       (* for all j < i *)
```

If an exact match has been found, then `i` is a valid index into the array. If no exact match has been found, then `i` is the position where the search key can be inserted without violating the order.

### Basic Algorithm

The basic search algorithm works between two valid indices `lower` and `upper` satisfying the invariant:

```
0 <= lower < upper < length arr

arr.(lower) < key < arr.(upper)
```

If `lower + 1 = upper`, then no exact match can be found, because there is no other element between the indices `lower` and `upper`. Therefore the pair `(upper, false)` satisfies the specification.

If `lower + 1 < upper`, then there is at least one index between `lower` and `upper` i.e. `2 <= upper - lower` and `(upper - lower) / 2` is at least `1`.

The middle value `mid = lower + (upper - lower) / 2` lies strictly between the indices `lower` and `upper` and is a valid index into the array. There are 3 possibilities:

- `key < arr.(mid)`: We have to continue the search between `lower` and `mid`.
- `key = arr.(mid)`: We have found an exact match and return `(mid, true)`.
- `arr.(mid) < key`: We have to continue the search between `mid` and `upper`.

We define the tail recursive helper function as follows:

```
let rec search lower upper =
    (* Invariant:

        0 <= lower < upper < len

        arr.(lower) < key < arr.(upper)
    *)
    if lower + 1 = upper then
        upper, false
    else
        let mid = lower + (upper - lower) / 2 in
        let cmp = compare key (key_of arr.(mid)) in
            if cmp < 0 then
                search lower mid
            else if cmp = 0 then
                mid, true
            else
                search mid upper
```

### Complete Algorithm

The basic algorithm requires an array length of at least 2 and that the search key lies strictly between the first and the last element of the array. In the complete algorithm we have to cover the corner cases there these conditions are not satisfied i.e. the lengths zero and one and cases that the search key is strictly less than or greater than all elements of the array.

```
let binsearch
    (compare: 'key -> 'key -> int)      (* Comparison function. *)
    (key_of:  'item -> 'key)            (* Extract a key from an item. *)
    (key:     'key)                     (* The key to be found. *)
    (arr:     'item array)              (* The array to be search in. *)
    : int * bool
    =
    let len = length arr
    in
    if len = 0 then
        len, false

    else if len = 1 then
        let cmp = compare key (key_of arr.(0)) in
        if cmp <= 0 then
            0, cmp = 0
        else
            len, false

    else
        (** length is at least 2! *)
        let rec search lower upper =
            ... (* see above *)
        in
        let lower, upper = 0, len - 1 in
        let cmp = compare key (key_of arr.(lower)) in
        if cmp <= 0 then
            (* key is less or equal the first element *)
            lower, cmp = 0
        else
            (* key is greater than the first element *)
            let cmp = compare key (key_of arr.(upper)) in
            if cmp < 0 then
                (* invariant for [search] satisfied. *)
                search lower upper
            else if cmp = 0 then
                (* exact match with the last element *)
                upper, true
            else
                (* key is greater than all elements *)
                len, false
```

## 1.2 Radix Balanced Array

As opposed to lists, arrays allow random access. You can get any element of an array in constant time. Lists need linear time if you want to access the `i`th element of the list.

However inserting elements to an array is expensive. It requires to allocate a new array which has room for the additional element. Then you have to copy the original array into the new array and add the additional element to the array.

Inserting elements into lists is a cheap operation as long as you insert the new element to the front of the list.

Radix balanced arrays (or trees) are a good compromise. They offer fast random access, fast appending of new elements to the end of the array and fast removal of elements from the end of the array. They are based on the idea that a long array can be split into chunks which fit into the cache line of modern computers. As long as the array segments fit into a cache line, inserting is a relatively cheap operation. If we organize the chunks into a radix tree, then random access to specific elements needs only time proportional to the height of the tree which is very small even for very long arrays.

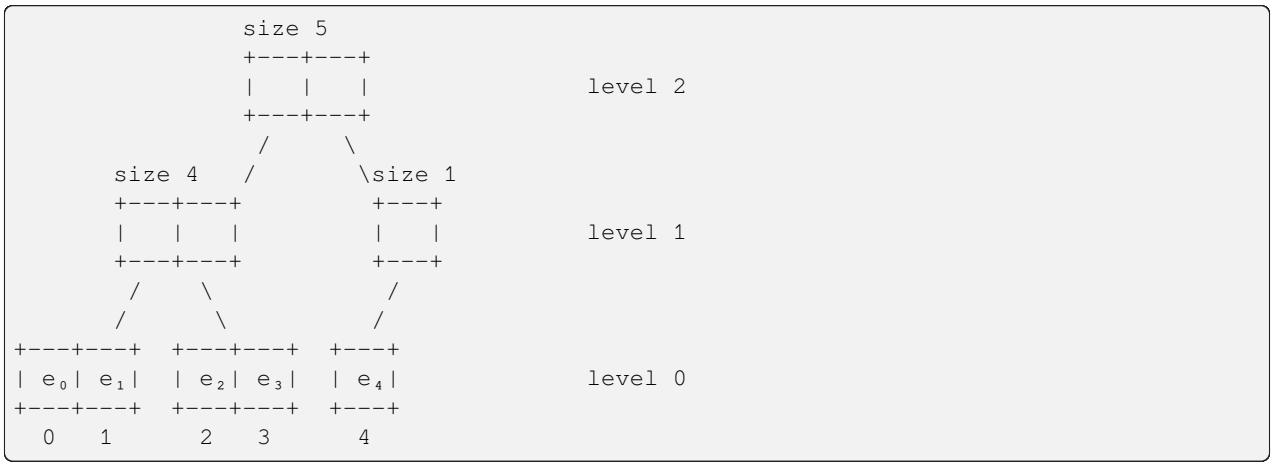
### 1.2.1 Basic Idea

We work with a branching factor $B$ which is a power of two. The exponent of the branching factor is $b$ such that $B = 2^b$. On current microprocessors the size of a cache line is 32 machine words or more. Therefore $b = 5$ and $B = 2^5 = 32$ are good choices. In order to illustrate the radix balanced arrays and keep the graphics reasonably sized we use a branching factor of 2 in the examples.

We use arrays of size $B$ to store the elements of the whole structure. If we want to store 5 elements with branching factor 2 we need 3 arrays to store the chunks




The complete tree to store 5 elements looks like




At level 0 we have all the leaves which store the elements. In this specific case we have intermediate nodes at level 1 and at level 2. Each node at level $l$ has a slot size of $2^{lb}$ and is capable to address $2^{(l+1)b}$ elements. The following table illustrates the slot sizes and maximal sizes for the branching factor 2 and the branching factor 32.

| level | slot size (B = 2) | max size (B = 2) | slot size (B = 32) | max size (B = 32) |
|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 32 |
| 1 | 2 | 4 | 32 | 1024 |
| 2 | 4 | 8 | 1024 | 32768 |
| 3 | 8 | 16 | 32768 | 1048576 |

Note that with a branching factor 32 which fits well into the cache line of modern microprocessors we are able to store more than 1 million elements in a tree with only 3 intermediate levels.

The tree is balanced in the sense that each subtree of a tree has the same depth and all subtrees except the rightmost are full.

Each subtree is a valid radix balanced tree. The counting of its elements starts at zero.

If we want to find an element at index $i$ we first have to find the valid slot $s$ in the tree. This can be done by integer division $s = i/2^{lb}$.

The relative index $o$ (i.e. its offset) in the subtree at slot $s$ can be found by the formula $o = i - 2^{lb}s$.

Integer divisions and multiplications by powers of 2 can be performed very efficiently by using bitshifts. The indices are always non-negative integers. Therefore logical shift does the same thing as arithmetic shift.

```
let bitsize: int = 32

let slot (i: int) (l: int): int =
    (* The slot of index [i] at level [l].

        i / 2^(l * bitsize)
     *)
    i lsr (l * bitsize)


let offset (i: int) (s: int) (l: int): int =
    (* The offset of index [i] in slot [s] in a tree at level [l].

        i- s * 2^(l * bitsize)
     *)
    i - s lsl (l * bitsize)
```

A radix balanced array at level $l$ is full if it has $2^{(l+1)b}$ elements.

```
let full_size (l: int): int =
    (* The size of a full radix balanced array at level [l]. *)
    assert (0 <= l);
    1 lsl ((l + 1) * bitsize)
```

### 1.2.2 Data Structure

We use an algebraic data type to define the type of a radix balanced tree. The leaf node is just an array of the element type. An intermediate node is an array whose elements are other radix balanced trees and has information about its level and its total number of elements.

```
type 'a t =
    | Leaf of
        'a array
```

(continues on next page)

(continued from previous page)

```
    | Node of {
        size:  int; (* Total number of elements, including subtrees. *)
        level: int;
        nodes: 'a t array}
```

Since the total number of elements is available in all intermediate nodes, the function to calculate the number is elements i.e. the length of the radix balanced array is straightforward and doesn't need any recursion.

```
let length: 'a t -> int =
    (* The length of the radix balanced array. *)
    function
    | Leaf arr ->
        Array.length arr
    | Node node ->
        node.size
```

The functions to compute the level, the fullness and the emptyness of a radix balanced array are straightforward as well.

```
let level: 'a t -> int = function
    | Leaf _ ->
        0
    | Node node ->
        node.level

let is_full: 'a t -> bool = function
    | Leaf arr ->
        Array.length arr = full_size 0
    | Node node ->
        node.size = full_size node.level

let is_empty (t: 'a t): bool =
    length t = 0

let has_some (t: 'a t): bool =
    length t > 0
```

### 1.2.3 Invariant

The following invariant is maintained by all operations where $B$ is the branching factor:

- Each leaf node has at most $B$ elements.
- Each leaf node which is not the root node has at least one element.
- Each interior node has at most $B$ children.
- Each interior node has at least one child.
- If the root node is an interior node it has at least two children (a root node with only one child makes no sense and can be removed and the child can be used as the root).
- All children of an interior node except the last child are full.
- All children of an interior node have the same level and the level of the interior node is one higher than the level of its children.
- The size of an interior node is the sum of the size of the children.

### 1.2.4 Element Retrieval

The following function retrieves the element at a certain index $i$ i.e. it implements the random access.

```
let rec element (i: int) (t: 'a t): 'a =
    (* The element at index [i] in the radix balanced array [t]. *)
    assert (0 <= i);
    assert (i < length t);
    match t with
    | Leaf arr ->
        arr.(i)
    | Node node ->
        let s = slot i node.level in
        let o = offset i s node.level in
        element o node.nodes.(s)
```

Note that $i$ is the relative index within the subtree. Before going down one level we have to find the slot and the relative offset within the slot.

In a nonempty array the first and last element of the array can be retrieved by using the function `element`. However more efficient functions can be defined which do not need any slot and offset computations. The first element of an intermediate node is always in the first child (i.e. first slot) and the last element of an intermediate node is always in the last child (i.e. last slot).

```
let first (t: 'a t): 'a =
    (* The first element of the non empty radix balanced array [t]. *)
    assert (has_some t);
    let rec fst = function
        | Leaf arr ->
            Array.first arr
        | Node node ->
            fst (Array.first node.nodes)
    in
    fst t


let last (t: 'a t): 'a =
    (* The last element of the non empty radix balanced array [t]. *)
    assert (has_some t);
    let rec fst = function
        | Leaf arr ->
            Array.last arr
        | Node node ->
            fst (Array.last node.nodes)
    in
    fst t
```

### 1.2.5 Element Replacement

Replacing an element within the radix balanced array affects only nodes on the path from the root to the corresponding leaf. In the leaf the element has to be replaced by the new element. In each intermediate node the child at the corresponding slot has to be replaced by the new child.

```
let rec replace (i: int) (e: 'a) (t: 'a t): 'a t =
    (* Replace the element at index [i] by the element [e] within the radix
       balanced array [t]. *)
    assert (0 <= i);
    assert (i < length t);
    match t with
    | Leaf arr ->
        Leaf (Array.replace i e arr)
    | Node node ->
        let s = slot i node.level in
        let o = offset i s node.level in
        Node
            {node with
                nodes =
                    Array.replace
                        s
                        (replace o e node.nodes.(s))
                        node.nodes
            }
```

Each replacement within an elementary array requires an array copy. But remember that the elementary arrays always fit into a cache line such that the operations are very fast.

### 1.2.6 Element Insertion at the Rear End

Radix balanced arrays are *balanced* and *packed* which means that all subtrees have the same height and are full except the rightmost subtrees. Therefore cheap insertion is possible only at the rear end of the array. It is possible to relax the invariant like it is done in *radix relaxed balanced trees*. However this has a cost for the runtime performance. The implementation described here has no such relaxation because it tries to be as efficient as possible and keep all subtrees packed and fully balanced.

Remember that a radix balanced tree at the level $l$ is full if it has $2^{(l+1)b}$ elements.

We want to insert an element $e$ at the rear end of the tree $t$

If the balanced tree is full, it is not possible to insert an element into the tree. The only thing we can do is to construct a new tree at the same level with only one element.

```
+---+
|   |               level 3
+---+
  |
+---+
|   |               level 2
+---+
  |
+---+
|   |               level 1
+---+
  |
+---+
```

(continues on next page)

(continued from previous page)

```
| e |               level 0
+---+
```

```
let rec singleton_tree (lev: int) (e: 'a): 'a t =
    (* Construct tree at level [lev] with the element [e]. *)
    if lev = 0 then
        Leaf [| e |]
    else
        Node {
            size = 1;
            level = lev;
            nodes = [| singleton_tree (lev - 1) e |]
        }
```

If the tree $t$ has a parent, then it is the last child of the parent. Since $t$ is full and the parent is not full, we can append the singleton tree to the nodes of the parent.

If the tree $t$ is the root, then we need a new root node with $t$ as the first child and the singleton tree as the second child.

Since we have constructed the singleton tree at the same level as the tree $t$, the new tree is still balanced.

Insertion into a not full tree can be done by the following recursive function:

```
let rec push_not_full (e: 'a) (t: 'a t): 'a t =
    (* Append the element [e] at the rear end of the radix balanced array
       [t] which is not full. *)
    assert (not (is_full t));
    match t with
    | Leaf arr ->
        Leaf (Array.push e arr)

    | Node node ->
        let slot = Array.length node.nodes - 1
        assert (0 <= slot);
        in
        let nodes =
            if is_full node.nodes.slot then
                Array.push
                    (singleton_tree (node.level - 1))
                    node.nodes
            else
                Array.replace
                    slot
                    (push_not_full e node.nodes.(slot))
                    node.nodes
        in
        Node
            {node with nodes; size = node.size + 1}
```

Finally we get the complete insertion function.

```
let push (e: 'a) (t: 'a t): 'a t =
    (* Append the element [e] at the rear end of the radix balanced array
       [t]. *)
    let lev = level t
    and len = length t
    in
```

(continues on next page)

(continued from previous page)

```
    if len = full_size lev then
        Node {
            size  = len + 1;
            level = lev + 1;
            nodes = [| t; singleton_tree lev e|]
        }
    else
        push_not_full e t
```

### 1.2.7 Element Deletion at the Rear End

The structure of the radix balanced tree allows fast removal of elements on the rear end. This is possible only if the array is not empty. We want to write a function with the following signature

```
val pop (t:'a t): 'a * 'a t
(* Pop the last element off the array [t] and return it together with
   the array where the last element has been removed.

   Precondition: [t] must not be empty i.e. [has_some t]
*)
```

The basic algorithm is not complicated.

If the tree is a leaf node we just remove the last element from the non empty array.

If the tree is an interior node, then we take the last child and remove recursively the last element from the child and replace the last child by the child where the last element has been removed. However we have to consider the following corner case:

**The last child has only one element:**
In that case the child disappears completely. I.e. the new interior node has one child less than the old interior node. We have to distinguish two cases:

**The parent node is the root node:**
The root node has at least two children. Since the last child has only one element, we remove the last child completely. Since there remains only one child in the root node, we can replace the root node by its first child.

**The parent node is not the root node:**
The parent node has at least two children. Otherwise it would have only one element and would have been removed completely. Since the last child has only one element, we remove the last child completely.

The following helper function removes the last element from a nonempty tree.

```
let rec pop_aux (is_root: bool) (t: 'a t): 'a * 'a t =
    (* Remove the last element from a nonempty tree. *)
    assert (has_some t);
    match t with
    | Leaf arr ->
        Array.(last arr, Leaf (remove_last arr))
    | Node node ->
        let j = Array.length node.nodes - 1 in
        assert (0 <= j);
        let child = node.nodes.(j) in
        let len   = length child in
        if is_root && j = 1 && len = 1 then
            (* Last child of the root node has only one element. *)
```

(continues on next page)

(continued from previous page)

```
            last child,
            node.nodes.(0)
        else
            let e, nodes =
                if len = 1 then
                    (* Last child has only one element. *)
                    last child,
                    Array.remove_last node.nodes
                else
                    (* Normal case. *)
                    let e, child = pop_aux false child in
                    e,
                    Array.replace j child node.nodes
            in
            e,
            Node {
                node with
                size = node.size - 1;
                nodes
            }
```

With the helper function is straightforward. We can write two versions of the removal function. The first assumes that the tree is not empty. The second one works on empty trees as well (returning an option).

```
let pop (t: 'a t): 'a * 'a t =
    assert (has_some t);
    pop_aux true t

let pop_opt (t: 'a t): ('a * 'a t) option =
    if is_empty t then
        None
    else
        Some (pop_aux true t)
```

## 1.3 B Tree

### 1.3.1 General

A B-tree has interior nodes and leaf nodes. The leaf nodes have no children. The interior nodes have children. The root node is either an interior node (if it has children) or a leaf node (if it doesn't have children).

Each node carries keys (or key value pairs). The keys are sorted. Each key separates two children in the interior nodes. The child to the left of a key has only keys strictly lower than the key and the child to the right of a key has only keys strictly greater than the key.

Each interior node has one child more than the number of keys. The maximum number `m` of children in a B tree is called its *order*. The minimal order of a B tree is 3.

```
            Example: B-tree of order 3

                +---+
                | 4 |                              root node / interior node
                +---+
             /         \
```

(continues on next page)

(continued from previous page)

```
    +---+                +---+---+
    | 2 |                | 7 | 9 |                interior nodes
    +---+                +---+---+
    |   |               /    |    \
+---+   +---+   +---+---+  +---+   +---+
| 1 |   | 3 |   | 5 | 6 |  | 8 |   | 10|          leaf nodes
+---+   +---+   +---+---+  +---+   +---+
```

A B-tree of order `m` satisfies the following invariant:

- Every node has at most `m - 1` keys and `m` children in case of an interior node.
- Every node (except the root node) has at least `ceil (m / 2) - 1` keys and `ceil (m / 2)` children in case of an iterior node.
- The tree is balanced in the sense that all leaves appear at the same level.
- The tree is sorted in the sense that all keys of the child to the left of a key are strictly smaller than the key and all keys of the child to the right of a key are strictly greater than the key.

Note that `m / 2` is not an integral value if `m` is an odd number. Therefore `ceil (m / 2)` rounds the value up to its nearest integral number. Example: If the order is 3, then `3 / 2 = 1.5` and `ceil (3 / 2) = 2`. I.e. a B tree of order 3 has at least 2 children and 1 key and at most 3 children and 2 keys.

| order | keys | children |
|---|---|---|
| 3 | 1 - 2 | 2 - 3 |
| 4 | 1 - 3 | 2 - 4 |
| 5 | 2 - 4 | 3 - 5 |
| 6 | 2 - 5 | 3 - 6 |
| 32 | 15 - 31 | 16 - 32 |

According to the invariant each node except the root node has at least the minimal number of keys. Any interior node except the root node has at least the miminal number of children.

The keys (or key value pairs) and the pointers to the children are usually stored in arrays. If we choose a power of two for the order which represents the caches size of a processor (e.g. 32), then the arrays can be stored within a cache line which speeds up insertion an deletion and results in better data locality.

B trees can be used to implement finite maps i.e. to store and retrieve key value pairs or finite sets i.e. to store keys. The keys have to be sortable. In this section we describe the the implementation of finite map via B trees.

In ocaml we use a module functor of the following form:

```
module Map (Key: SORTABLE) =
struct

    let order = ...                                (* usually 32 for cache efficiency *)

    let odd_order: bool =                          (* is the order odd? *)
        assert (3 <= order);                       (* minimal order 3 is a must *)
        order / 2 * 2 < order

    let max_keys: int = order - 1                  (* the maximal number of keys *)

    let min_keys: int =                            (* the minimal number of keys *)
        if odd_order then
            (order - 1) / 2
```

(continues on next page)

(continued from previous page)

```
        else
            order / 2 - 1

    type 'a pairs = (Key.t * 'a) array

    type 'a t =
        | Leaf of 'a pairs                  (* key value pairs *)
        | Node of 'a pairs * 'a t array     (* key value pairs + children *)

    ...
    (* Search, insert and delete see below ... *)
end
```

with the module type `SORTABLE` which is defined in the module `Fmlib_std.Interfaces`.

```
module type SORTABLE = sig
    type t

    val compare: t -> t -> int
    (** [compare a b]

        compare a b < 0         if and only if a < b
        compare a b = 0         if and only if a = b
        compare a b > 0         if and only if a > b
    *)
```

A leaf consists of an array of keys (or key value pairs)

```
   0    1    2                                len
+----+----+--------------------------------+
| k0 | k1 | ...                            |
+----+----+--------------------------------+
```

where `k0`, `k1`, ... `k(len-1)` are the keys of the leaf.

An interior node consists of an array of keys and an array of children where the array of children has one more element than the array of keys.

```
   0    1    2                                len
+----+----+--------------------------------+
| k0 | k1 | ...                            |
+----+----+--------------------------------+
|    |    |                                |
c0   c1   c2                              c(len)
```

`c0`, `c1`, `c2`, ... `c(len)` are the children with the property that all keys in the child `c1` are strictly less than the key `k1` and all the keys in the child `c2` are strictly greater than the key `k1`.

### 1.3.2 Search

Since the keys in a B tree are sorted we can search within a B tree using binary search. Let's assume we have a binary search function of the form

```
let bsearch (key: Key.t) (arr: 'a pairs): int * bool =
    ...
```

We can search within the leaf and the interior nodes the corresponding array of key value pairs for the position of the key `key`. The function returns the position and an exact flag. The exact flag indicates if an exact match has been found. If the flag is not set, then the search key is strictly smaller than the key at the position. If the length is returned as the position, then we know that all keys in the array are strictly smaller than the search key.

If no exact match can be found in a leaf node, then the key is not in the leaf.

If no exact match can be found in an interior node, then the search key is not in the interior node. However it can be in the child at the corresponding position. Note that the array of the children has always one more element than the array of key value pairs. Therefore there is a valid child at the position `length pairs`.

The search algorithm can be implemented by a straightforward recursive function

```
let rec find_opt (key: Key.t) (map: 'a t): 'a option =
    match map with
    | Leaf pairs ->
        let i, exact = bsearch key pairs in
        if exact then
            Some (snd pairs.(i))
        else
            None

    | Node (pairs, children) ->
        let i, exact = bsearch key pairs in
        if exact then
            Some (snd pairs.(i))
        else
            find_opt key children.(i)
```

### 1.3.3 Insertion

All nodes have at most `m - 1` keys and `m` children where `m` is the order of the B tree. A node which has exactly `m - 1` keys (and `m` children in case of an interior node) is full.

Insertion always starts in a leaf node. If the leaf node is full, then the insertion causes an overflow. The overflow condition might pop up to the root. In that situation the height of the tree grows.

An insertion of a key value pair in a tree which has already a key value pair with the same key causes a nondestructive overwrite of the old value by the new value. The structure of the B tree remains the same, just the value of the corresponding key value pair will be updated.

A proper insertion of a new key value pair starts by searching the leaf node and the position in the leaf node where to insert the new pair. The result of the insertion is described by the data type

```
type 'a insert =
    | Normal_insert of 'a t
    | Split_insert of 'a t * (Key.t * 'a) * 'a t
```

I.e. we insert the new pair into a leaf node and either return a new leaf node if there is enough room in the node or a splitted leaf node with a new key value pair which has to be inserted into the corresponding parent or a splitted leaf node which consist of a left leaf node, a popup key value pair and a right leaf node.

The insertion into the parent can either end in a normal insert or in a split insert.

During insertion the invariant is maintained that the popup key separates all the key value pairs of the left tree from the key value pairs of the right tree.

The basic insertion function looks like

```
let add (key: Key.t) (value: 'a) (map: 'a t): 'a t =
    match add_aux key value map with
    | Normal_insert map ->
        map
    | Split_insert (left, popup_key, right) ->
        (* tree grows at the root *)
        Node ([| popup_key |], [| left; right |]
```

If the splitting reaches the root, then a new root is created with one key value pair (the popup pair) and two children.

The basic insertion function uses the auxiliary function `add_aux` which implements the recursive algorithm.

```
let rec add_aux (key: Key.t) (value: 'a) (map: 'a t): 'a insert =
    match map with
    | Leaf pairs ->
        add_in_leaf key value pairs

    | Node (pairs, children) ->
        let i, exact = bsearch key pairs in
        if exact then
            (* An exact match has been found. Therefore update the value. *)
            let pairs = Array.replace i (key,value) pairs in
            Normal_insert (Node (pairs, children))
        else
            (** Add the key value pair into the [i]th child. *)
            match add_aux key value children.(i) with
            | Normal_insert child ->
                let children = Array.replace i child children in
                Normal_insert (Node (pairs, children))
            | Split_insert (u, y, v) ->
                add_in_node i u y v pairs children
```

The function `add_aux` uses the two helper functions `add_in_leaf` and `add_in_node` to insert the key value pair either into a leaf node or an interior node.

### Insertion into a leaf node

The insertion into a leaf node is easy, if the leaf is not full. In case of overflow we have to distinguish several cases.

In the following we assume that we want to insert a key value pair `y` at position `i` into a full leaf node. As a result we want to get a triple `(left, popup_key, right)` where `left` is the left tree after the split, `right` is the right key after the split and the key in the key value pair `popup_key` separates the keys in `left` from the keys in `right`.

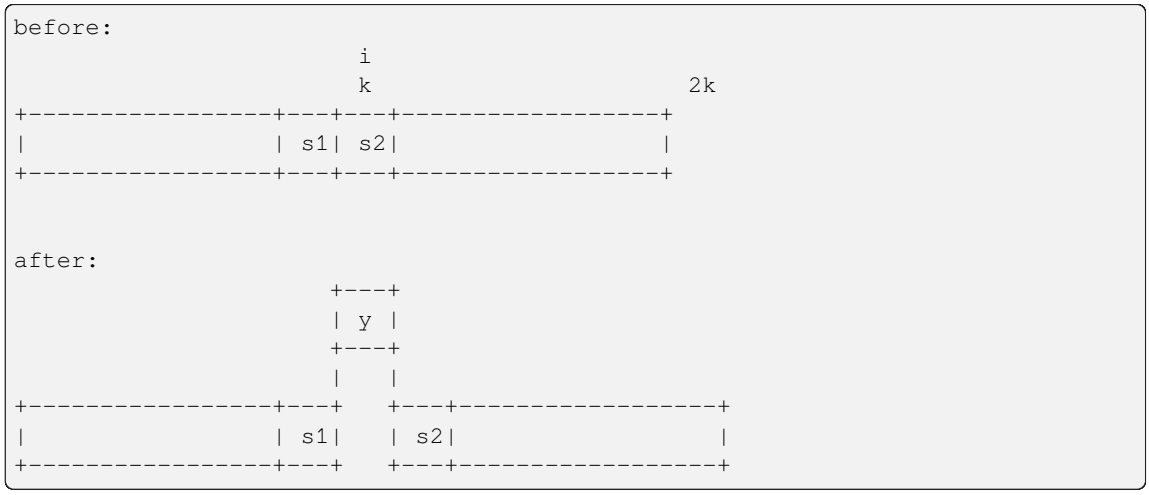
Now we analyze the different cases.

**Overflow, odd order:**

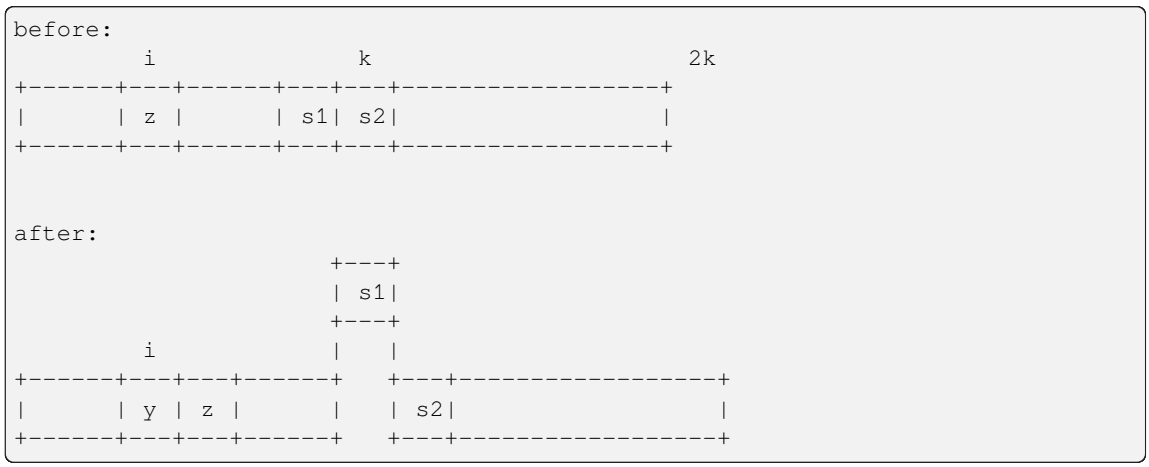
`m = 2 * k + 1`, i.e. we have `2 * k` keys and the array of the key value pairs consists of two subarrays of size `k`. At the right end of the left subarray there is a key `s1` and at the left end of the right subarray there is a key `s2`.

We have to distinguish the cases `i = k`, `i < k` and `i > k`.

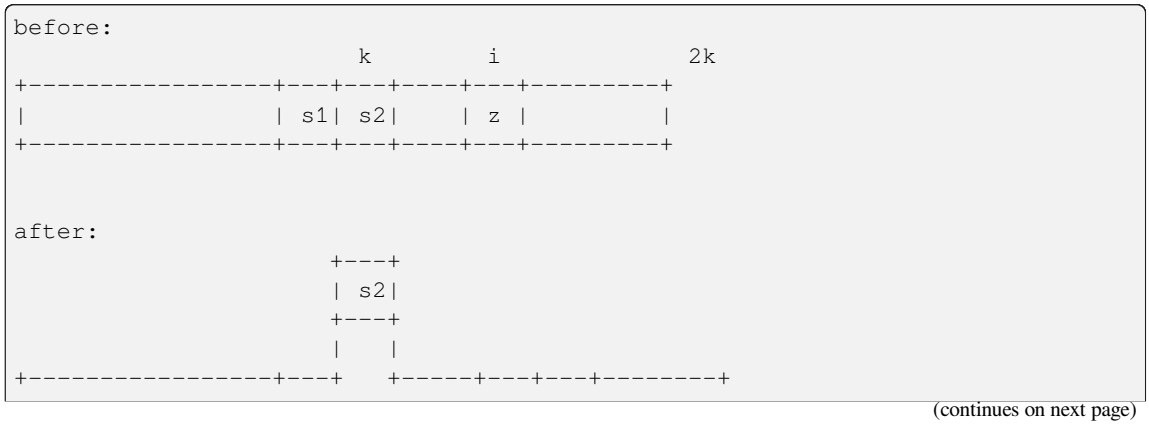
**`i = k`:**
In that case we have `s1 < y < s2`. We split the array into the two subarrays of size `k` and use `y` as the popup key.




**`i < k`:**
In that case we have `y < s1` and we have to insert `y` into the left subarray and use `s1` as the popup key.




**`i > k`:**
In that case we have `s2 < y` and we have to insert `y` into the right subarray and use `s2` as the popup key.




(continues on next page)

(continued from previous page)

```
|                  | s1|    |      | y | z |         |
+------------------+---+    +------+---+---+---------+
```

**Overflow, even order:**

`m = 2 * k`, i.e. we have `2 * k - 1` key value pairs and the array of key value pairs consists of two subarrays of size `k - 1` with a single separator pair `s` which separates the two subarrays.

We have to distinguish the cases `i < k` and `i >= k`. In both cases `s` can be used as the popup key. The insertion of `y` happens either in the left or in the right subarray.

`i < k`:

```
before:
             i             k
+---------+---+----+---+------------------+
|         | z |    | s |                  |
+---------+---+----+---+------------------+


after:

                         +---+
                         | s |
                         +---+
             i           |   |
+---------+---+---+------+   +------------------+
|         | y | z |      |   |                  |
+---------+---+---+------+   +------------------+
```

`i >= k`:

```
before:
                           k    i
+------------------+---+----+---+---------+
|                  | s |    | z |         |
+------------------+---+----+---+---------+


after:

                   +---+
                   | s |
                   +---+
                   |   |
+------------------+   +------+---+---+---------+
|                  |   |      | y | z |         |
+------------------+   +------+---+---+---------+
```

The following function does the insertion into a leaf node.

```
let add_in_leaf (key: Key.t) (value: 'a) (pairs: 'a pairs): 'a insert =
    let len = Array.length pairs in
    let i, exact = bsearch key pairs in
    if exact then
        Normal_insert (Leaf (Array.replace i (key, value) pairs))

    else if len < max_keys then
```

(continues on next page)

(continued from previous page)

```
        (* Leaf is not full. *)
        Normal_insert (Leaf (Array.insert i (key, value) pairs))

    else
        (* Leaf is full *)
        let insert_subarray = insert_subarray pairs i (key, value)
        and k = order / 2
        in
        if odd_order then
            if i = k then
                let left  = subarray pairs 0 k
                and right = subarray pairs k len
                in
                Split_insert (Leaf left, (key, value), Leaf right)
            else if i < k then
                let left  = insert_subarray 0 (k - 1)
                and right = subarray pairs k len
                in
                Split_insert (Leaf left, pairs.(k - 1), Leaf right)
            else
                let left  = subarray pairs 0 k
                and right = insert_subarray (k + 1) len
                in
                Split_insert (Leaf left, pairs.(k), Leaf right)
        else begin
            (* even order *)
            if i < k then
                let left  = insert_subarray 0 (k - 1)
                and right = subarray pairs k len
                in
                Split_insert (Leaf left, pairs.(k - 1), Leaf right)
            else
                let left  = subarray pairs 0 (k - 1)
                and right = insert_subarray k len
                in
                Split_insert (Leaf left, pairs.(k - 1), Leaf right)
        end
```

Using the two helper functions `subarray` and `insert_subarray`.

```
subarray (arr: 'a array) (start: int) (beyond: int): 'a t =
    (* The subarray of [arr] starting at [start] and ending one before [beyond]. *)
    assert (0 <= start);
    assert (start <= beyond);
    assert (beyond <= Array.length arr);
    Array.sub arr start (beyond - start)


let insert_subarray
        (arr: 'a array) (i: int) (x: 'a) (start: int) (beyond: int)
    : 'a array
    =
    (* The subarray of [arr] starting at [start] and ending one before [beyond]
       with [x] inserted at position [i]. *)
    assert (0 <= start);
    assert (start <= i);
    assert (i <= beyond);
```

(continues on next page)

(continued from previous page)

```
    assert (beyond <= Array.length arr);
    let arr2 = Array.make (beyond - start + 1) x in
    Array.blit arr start arr2 0 (i - start);
    Array.blit arr i arr2 (i - start + 1) (beyond - i);
    arr2
```

### Insertion into an interior node

In this subsection we treat the situation that a key value pair has been inserted into the `i` th child `t` of an interior node and the insertion into the child `t` caused an overflow and resulted in the triple `(u, y, v)` where `u` and `v` are two valid B trees separated by the popup key value pair `y`. Furthermore the interior node is full and must be splitted as well.

The case distinctions are the same as in the insertion into a full leaf node with the additional complexity that the child nodes have to be treated.

**Overflow, odd order:**

`m = 2 * k + 1`. The array of the key value pairs consists of two subarray of equal size `k` with the key value pair `s1` ending the left subarray and `s2` beginning the right subarray.

We have to distinguish the cases `i = k`, `i < k` and `i > k`.

**`i = k`:**

`s1` is a strict lower bound of all keys in `t` and therefore for all keys in `(u, y, v)`. `s2` is a strict upper bound. We split the interior node into the two subarrays and use `u` as the last child in the left part and `v` as the first child in the right part. We use `y` as the popup key.

```
before:
                        i
                        k                       2k
+-----------------+---+---+-------------------+
|                 | s1| s2|                   |
+-----------------+---+---+-------------------+
                      |
                      t


after:
                      +---+
                      | y |
                      +---+
                      |   |
+-----------------+---+   +---+-------------------+
|                 | s1|   | s2|                   |
+-----------------+---+   +---+-------------------+
                      |   |
                      u   v
```

**`i < k`:**

In that case we have to insert the popup key `y` at position `i` of the key value pairs of the interior node, replace the `i``th child ``t` by `u` and use `v` as the additionally needed child.

```
before:
         i              k                       2k
+------+---+------+---+---+-------------------+
|      | z |      | s1| s2|                   |
+------+---+------+---+---+-------------------+
```

(continues on next page)

(continued from previous page)

```
       |
       t


after:
                      +---+
                      | s1|
                      +---+
         i            |   |
+------+---+---+------+   +---+------------------+
|      | y | z |      |   | s2|                  |
+------+---+---+------+   +---+------------------+
       |   |
       u   v
```

**i > k:**
The same as before just with insertion into the right part of the interior node.

```
before:
                          k         i              2k
+-----------------+---+---+----+---+---------+
|                 | s1| s2|    | z |         |
+-----------------+---+---+----+---+---------+
                                |
                                t


after:
                      +---+
                      | s2|
                      +---+
                      |   |
+-----------------+---+   +-----+---+---+--------+
|                 | s1|   |     | y | z |        |
+-----------------+---+   +-----+---+---+--------+
                                |   |
                                u   v
```

**Overflow, even order:**
`m = 2 * k`.

We have to distinguish the cases `i < k` and `i >= k`.

`i < k`:

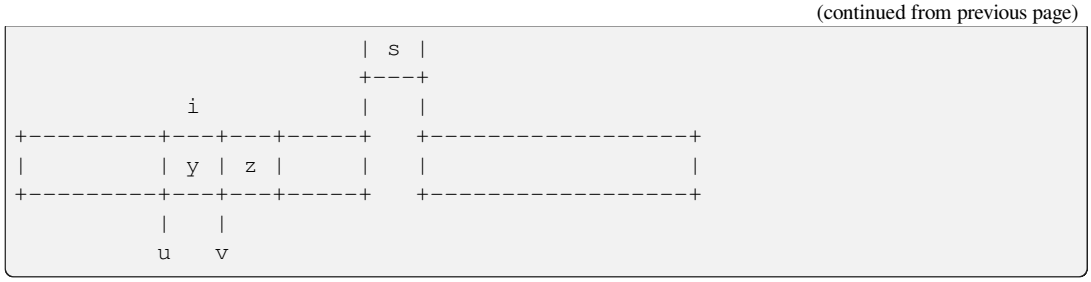
```
before:
            i            k
+---------+---+----+---+-----------------+
|         | z |    | s |                 |
+---------+---+----+---+-----------------+
          |
          t


after:

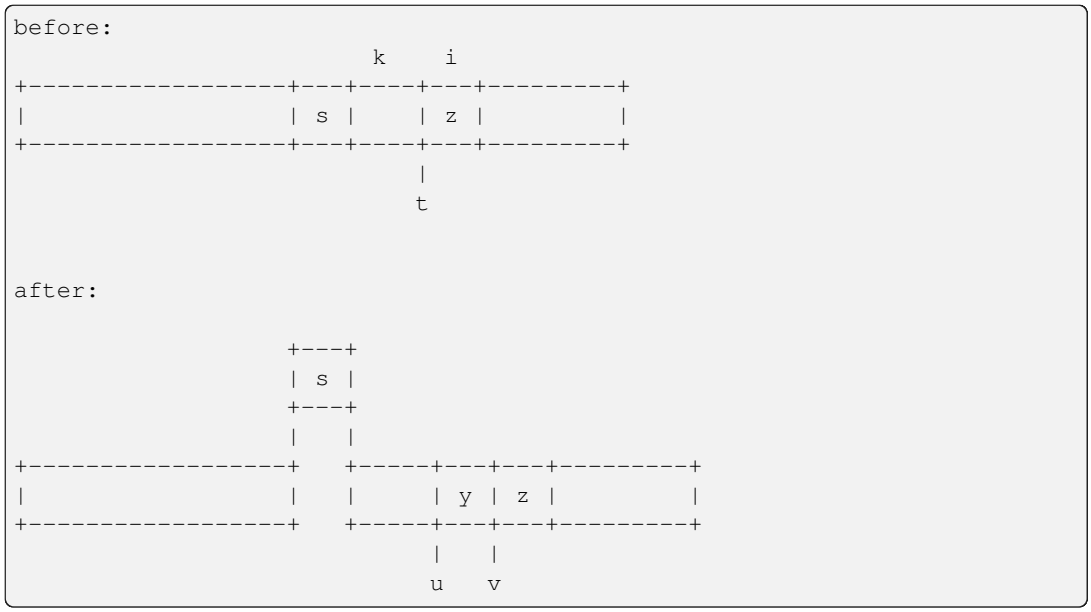
                        +---+
```

(continues on next page)

(continued from previous page)




`i >= k`:




The insertion function which inserts into an interior node reads like

```
let add_in_node
        (i: int)
        (left: 'a t)
        (pair: Key.t * 'a)
        (right: 'a t)
        (pairs: 'a pairs)
        (children: 'a t array)
    : 'a insert
    =
    let len = Array.length pairs in
    if len < max_keys then
        let pairs = Array.insert i pair pairs
        and children = Array.insert i left children
        in
        children.(i + 1) <- right;
        Normal_insert (Node (pairs, children))
    else
        (* Node is full. *)
        let k = order / 2
        and insert_subarray = insert_subarray pairs i pair
        and split_subarray start beyond =
            split_subarray children i left right start beyond
        in
```

(continues on next page)

(continued from previous page)

```
        if odd_order then
            if i = k then
                let left_pairs     = subarray pairs    0 k
                and left_children  = subarray children 0 (k + 1)
                and right_pairs    = subarray pairs    k len
                and right_children = subarray children k (len + 1)
                in
                left_children.(k)  <- left;
                right_children.(0) <- right;
                Split_insert (
                    Node (left_pairs, left_children),
                    pair,
                    Node (right_pairs, right_children))
            else if i < k then
                let left_pairs     = insert_subarray 0 (k - 1)
                and left_children  = split_subarray  0 k
                and right_pairs    = subarray pairs    k len
                and right_children = subarray children k (len + 1)
                in
                Split_insert (
                    Node (left_pairs, left_children),
                    pairs.(k - 1),
                    Node (right_pairs, right_children))
            else begin
                let left_pairs     = subarray pairs    0 k
                and left_children  = subarray children 0 (k + 1)
                and right_pairs    = insert_subarray (k + 1) len
                and right_children = split_subarray  (k + 1) (len + 1) in
                Split_insert (
                    Node (left_pairs, left_children),
                    pairs.(k),
                    Node (right_pairs, right_children))
            end
        else begin
            (* even order *)
            if i < k then
                let left_pairs     = insert_subarray 0 (k - 1)
                and left_children  = split_subarray  0 k
                and right_pairs    = subarray pairs    k len
                and right_children = subarray children k (len + 1)
                in
                Split_insert (
                    Node (left_pairs, left_children),
                    pairs.(k - 1),
                    Node (right_pairs, right_children))
            else
                let left_pairs     = subarray pairs    0 (k - 1)
                and left_children  = subarray children 0 k
                and right_pairs    = insert_subarray k len
                and right_children = split_subarray  k (len + 1)
                in
                Split_insert (
                    Node (left_pairs, left_children),
                    pairs.(k - 1),
                    Node (right_pairs, right_children))
        end
```

with the additional helper function

```
let split_subarray
        (arr: 'a array) (i: int) (x: 'a) (y: 'a) (start: int) (beyond: int)
    : 'a array
    =
    (* The subarray of [arr] starting at [start] and ending one before [beyond]
       with [x] inserted at position [i] and the original value at position
       [i] replaced by [y]. *)
    assert (i < beyond);
    let arr = insert_subarray arr i x start beyond in
    arr.(i - start + 1) <- y;
    arr
```

### 1.3.4 Deletion

#### Basic Deletion

As with insertion, an actual deletion can only be done on a leaf node.

If the to be deleted key value pair is located in an interior node, then it cannot be deleted directly. In order to delete a key value pair in an interior node, we have to find a direct neighbour (predecessor or successor with respect to the order), which is always in a leaf node and delete the direct neighbour. Since the deleted key value pair is a direct neighbour of the to be deleted key value pair we can substitute the deleted key value pair for the key value pair in the interior node without disturbing the order.

Furthermore we have to handle a possible underflow condition. The invariant of a B tree requires that each leaf node or interior node which is not the root of the tree must have a minimal number of keys (and a minimal number of children in case of an interior node).

If a node underflows because of the deletion of a key value pair, the missing key value pair can be pulled from the parent and the missing child from the sibling. This can require to merge the underflowing child with its sibling (details see below).

The reparation might cause the parent to underflow. The reparation of the underflow condition of the parent can be done with the help of the parent of the parent etc. The underflow condition might popup recursively to the root.

An underflow condition in the root node is no problem except the case that the root becomes empty (i.e. no key value pairs and just one child). In that case the height of the tree shrinks at the root and the child becomes the new root.

In order to handle deletion we use the datastructure:

```
type 'a delete = {
    tree:  'a t;         (* The tree with the deleted key value pair. *)
    pair:   Key.t * 'a; (* The deleted key value pair. *)
    underflow: bool;     (* one key less than the minimal number *)
}
```

The basic deletion looks like:

```
let remove (key: Key.t) (map: 'a t): 'a t =
    match remove_aux key map with
    | None ->
        map
    | Some d ->
        match d.tree with
        | Node (pairs, children) when Array.is_empty pairs ->
            (* tree shrinks at the root *)
```

(continues on next page)

(continued from previous page)

```
            children.(0)
        | _ ->
            d.tree
```

It uses an auxiliary function `remove_aux` which returns an optional `'a delete` structure. The auxiliary function returns `None` if the tree does not have a key value pair with the desired key. In case that the tree contains a key value pair with the desired key, a tree is returned with the key value pair deleted. The function `remove` has to check, if the root node is an interior node with no key value pairs. In that case the tree shrinks in height and the only child becomes the new root.

The function `remove_aux` implements the recursion.

```
let rec remove_aux (key: Key.t) (map: 'a t): 'a delete option =
    match map with
    | Leaf pairs ->
        let i, exact = bsearch key pairs in
        if exact then
            let pair =  pairs.(i)
            and pairs = Array.remove i pairs
            and underflow = Array.length pairs <= min_keys
            in
            Some {
                tree = Leaf pairs;
                pair;
                underflow
            }
        else
            None

    | Node (pairs, children) ->
        let i, exact = bsearch key pairs in
        if exact then
            let d = remove_last children.(i) in
            let pair  = pairs.(i)
            and pairs = Array.replace i d.pair pairs in
            Some (handle_delete i pair pairs children d)
        else
            Option.map
                (fun d -> handle_delete i d.pair d pairs children)
                (remove_aux key children.(i))
```

The function searches in a leaf node and in an interior node for the key value pair which has to be deleted.

In case of an exact match, the key is deleted. In a leaf node the deletion is straightforward. In an interior node the deletion cannot be done directly. The last key value pair in the child to the left of the to be deleted key value pair is deleted with the help of the function `remove_last` and the last key value pair is substituted for the to be deleted key value pair. Remember that the last key value pair in the child to the left of the to be deleted key value pair are direct neighbours.

The function `handle_delete` checks for an underflow condition and does the reparation if needed.

If the search does not find an exact match in a leaf node, then no key value pair with the to be deleted key exists and nothing has to be done.

If the search does not find an exact match in an interior node, then the deletion continues in the child to the left of the key.

The code of the function `handle_delete` is straightforward and uses the helper function `handle_underflow` to handle an underflow condition.

```
let handle_delete
        (i: int)                (* Index of the child where the deletion occurred. *)
        (pair: Key.t * 'a)      (* The deleted key value pair. *)
        (d: 'a delete)          (* The new tree with the key value pair deleted. *)
        (pairs: 'a pairs)       (* The key value pairs of the parent. *)
        (children: 'a t array)  (* The children of the parent. *)
    : 'a delete
    =
    if not d.underflow then
        {
            tree = Node (pairs, Array.replace i d.tree children);
            pair;
            underflow = false
        }
    else
        let len = Array.length pairs in
        if i < len then
            handle_underflow i true d.tree children.(i + 1) pair pairs children
        else
            let i = i - 1 in
            handle_underflow i false children.(i) d.tree pair pairs children
```

If the deletion in the child has not caused an underflow in the chid, the new child is substituted for the old child. This cannot case an underflow in the parent either.

If the deletion in the child has caused an underflow, then a sibling has to be used to repair the underflow.

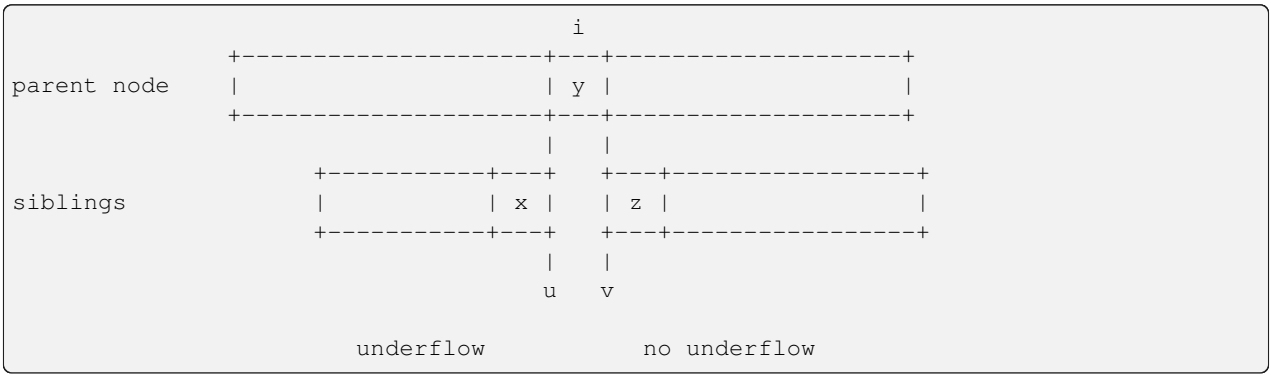
As long as the child is not the last child of the parent, there is a right sibling. The new child and its right sibling are passed to the function `handle_underflow` with the indication that the underflow happened in the first tree of the two siblings.

If the child is the last child of the parent, then we use its left sibling and hand it over to the function `handle_underflow`.

Since a valid B tree interior node has at least one key value pair and two children (even the root!) there is always either a left or a right sibling.

### Handling of Underflow

In this section we treat the case that the deletion in one of the children of a parent node caused an underflow and how this underflow in the child can be repaired with the help of the parent and a sibling. We assume that the underflow happend in the left child. The situation where the underflow happened in the right child is symmetrical.



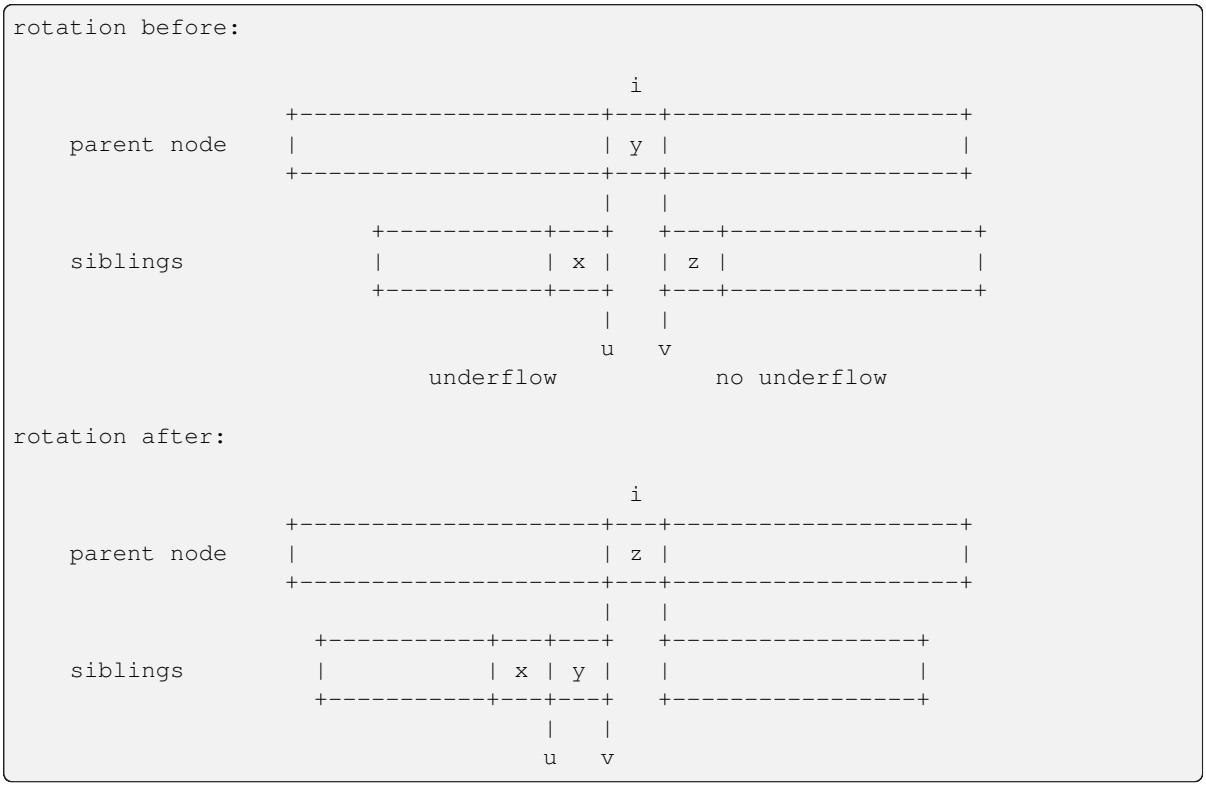

In the picture it is assumed that the siblings are interior nodes and have children. If the children are leaf nodes, then the algorithm is the same. Just ignore the children of the siblings.

The left sibling has exactly one missing key value pair and exactly one missing child. Reason: Since both are not the root node they have at least the minimal number of keys and children before the deletion. Deletion happened in the left sibling i.e. the left sibling had the minimal number of keys and children before the deletion. Otherwise no underflow would have happened.

We have to distinguish two cases: The right sibling is not minimal of the right sibling is minimal.

**Right sibling is not minimal:**

In that case we can chop off the first child `v` and the first key value pair `z` from the right sibling without violating the B tree invariant. The key value pair `z` can be pushed up to the parent and key value pair `y` from the parent and the B tree `v` can be appended to the underflowing chid to repair the situation.



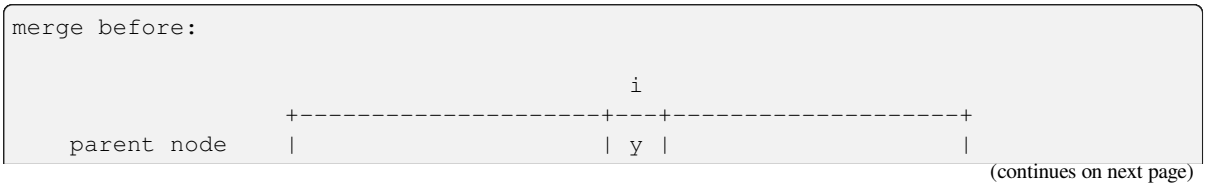

**Right sibling is minimal:**

In that case we cannot chop off a key value pair and a child from the right sibling without violating the B tree invariant. The only possibility is to merge the two siblings.

In order to merge the two sibling successfully, we have to push down the separator key value pair from the parent. This might cause an underflow in the parent depending on the size of the parent. A possible underflow in the parent must be repaired at the level of the parent of the parent.

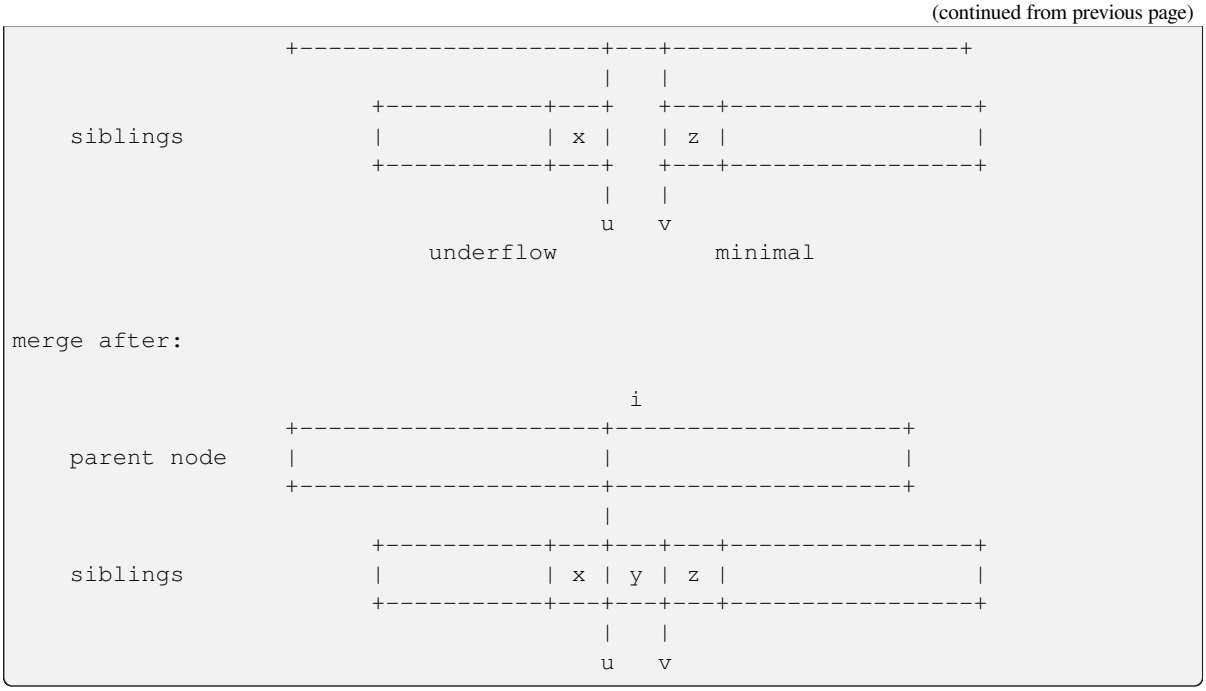
The parent looses one key value pair and one child with the merge. If the parent is the root node, the key value pair might be the last one leaving the parent with the merged child as the only child. In that case we can throw away the parent and use the merged child as the new root.

The merge can never create an overflow. The merged node has `2 * min_keys` key value pairs which is either the maximal number of keys or one less than the maximal number of keys.




(continues on next page)

(continued from previous page)




The following function handles the underflow condition properly in all cases.

```
let handle_underflow
        (i: int)                (* Index of the child where the deletion occurred. *)
        (underflow_left: bool)  (* Underflow happend in the left child? *)
        (left_child: 'a t)
        (right_child: 'a t)
        (pair: Key.t * 'a)      (* The deleted key value pair. *)
        (pairs: 'a pairs)       (* The key value pairs of the parent. *)
        (children: 'a t array)  (* The children of the parent. *)
    : 'a delete
    =
    let not_minimal pairs1 pairs2 =
        if underflow_left then
            not_minimal pairs2
        else
            not_minimal pairs1
    in
    match left_child, right_child with
    | Leaf pairs1, Leaf pairs2 when not_minimal pairs1 pairs2 ->
        (* Right sibling is not minimal, rotate *)
        let pairs1, pairs, pairs2 =
            rotate_keys underflow_left i pairs1 pairs pairs2
        in
        let children =
            replace2 i (Leaf pairs1) (Leaf pairs2) children
        in
        {tree = Node (pairs, children); pair; underflow = false}

    | Leaf pairs1, Leaf pairs2 ->
        (* Sibling is minimal, merge *)
        merge_leaves i pair pairs1 pairs2 pairs children

    | Node (pairs1, children1), Node (pairs2, children2)
```

(continues on next page)

(continued from previous page)

```
        when not_minimal pairs1 pairs2
        ->
        (* Sibling is not minimal, rotate *)
        let pairs1, pairs, pairs2 =
            rotate_keys underflow_left i pairs1 pairs pairs2
        and children1, children2 =
            rotate_children underflow_left children1 children2
        in
        let children =
            replace2
                i
                (Node (pairs1, children1))
                (Node (pairs2, children2))
                children
        in
        {tree = Node (pairs, children); pair; underflow = false}

    | Node (pairs1, children1), Node (pairs2, children2) ->
        (* Sibling is minimal, merge *)
        merge_nodes
            i pair
            pairs1 children1
            pairs2 children2
            pairs children

    | _, _ ->
        assert false (* Cannot happen, tree is balanced. *)
```

The function uses the following helper functions:

```
let not_minimal (pairs: 'a pairs): bool =
    min_keys < Array.length pairs


let replace2
        (i: int) (left: 'a t) (right: 'a t) (children: 'a t array)
    : 'a t array
    =
    let children = Array.copy children in
    assert (Array.valid_index i children);
    assert (Array.valid_index (i + 1) children);
    children.(i)     <- left;
    children.(i + 1) <- right;
    children


let rotate_keys
        (to_left: bool)
        (i: int) (left: 'a pairs) (parent: 'a pairs) (right: 'a pairs)
    : 'a pairs * 'a pairs * 'a pairs
    =
    let open Array in
    assert (valid_index i parent);
    if to_left then
        push parent.(i) left,
        replace i (first right) parent,
        remove_first right
```

(continues on next page)

(continued from previous page)

```
    else
        remove_last left,
        replace i (last left) parent,
        push_front parent.(i) right


let rotate_children
        (to_left: bool)
        (left: 'a t array) (right: 'a t array)
    : 'a t array * 'a t array
    =
    let open Array in
    if to_left then
        push (first right) left,
        remove_first right
    else
        remove_last left,
        push_front (last left) right



let merge_keys
        (i: int) (left: 'a pairs) (parent: 'a pairs) (right: 'a pairs)
    : 'a pairs * 'a pairs
    =
    assert (Array.valid_index i parent);
    let len_left  = Array.length left
    and len_right = Array.length right
    in
    let merged = Array.make (len_left + 1 + len_right) parent.(i)
    and parent = Array.remove i parent
    in
    Array.blit left  0 merged 0 len_left;
    Array.blit right 0 merged (len_left + 1) len_right;
    merged, parent


let merge_leaves
        (i: int)
        (pair: Key.t * 'a)
        (pairs1: 'a pairs) (pairs2: 'a pairs)
        (pairs: 'a pairs) (children: 'a t array)
    : 'a delete
    =
    assert (i + 1 < Array.length children);
    let merged, pairs = merge_keys i pairs1 pairs pairs2
    and children      = Array.remove i children
    and underflow     = Array.length pairs <= min_keys
    in
    children.(i) <- Leaf merged;
    {tree = Node (pairs, children); pair; underflow}



let merge_nodes
        (i: int)
        (pair: Key.t * 'a)
```

(continues on next page)

(continued from previous page)

```
        (pairs1: 'a pairs) (children1: 'a t array)
        (pairs2: 'a pairs) (children2: 'a t array)
        (pairs: 'a pairs) (children: 'a t array)
    : 'a delete
    =
    assert (i + 1 < Array.length children);
    let pairs_new, pairs = merge_keys i pairs1 pairs pairs2
    and children      = Array.remove i children
    and underflow     = Array.length pairs <= min_keys
    and children_new  = Array.append children1 children2
    in
    children.(i) <- Node (pairs_new, children_new);
    {tree = Node (pairs, children); pair; underflow}
```

CHAPTER

# TWO

# COMBINATOR PARSING

## 2.1 Layout Parsing

The design of layout parsing in `Fmlib` is based on the work of Adams and Ağacan [AAugacan14] with the following modifications:

- Tokens can by default appear to the right of the leftmost token of a construct. This modification makes the default handling more practical but does not change the semantics.
- The transition relation to describe the formal semantics has been extended to make parsing expressions not backtracking by default. A backtracking expression has to be annotated explicitly.

### 2.1.1 Parsing Expression Grammar

A parsing expression is either empty, a terminal, a nonterminal, a sequence of two expressions or a biased choice i.e. an expression is defined by the grammar

$$
\begin{aligned}
& e ::= \\
& \epsilon \text{ empty} \\
& \quad | \\
& a \text{ terminal} \\
& \quad | \\
& A \text{ nonterminal} \\
& \quad | \\
& ee \text{ sequence} \\
& \quad | \\
& e/e \quad \text{biased choice} \\
& \quad | \\
& \overleftarrow{e} \text{ backtrack}
\end{aligned}
$$

A parsing expression grammar is a four tuple $(T, N, S, \delta)$ which consists of a set of terminals (aka tokens), a set of nonterminals, the start symbol which is a special nonterminal and a function mapping each nonterminal to an expression.

The function $\delta$ can be extended to expressions such that

$$
\begin{aligned}
\delta(Ae) &:= \delta(A)e \\
\delta(e_1 \,/\, e_2) &:= \delta(e_1) \,/\, e_2 \\
\delta(ae) &:= ae \\
\delta(\epsilon) &:= \epsilon
\end{aligned}
$$

where $A$ is the nonterminal at the start of an expression. I.e. if an expression does not start with a nonterminal, then the function $\delta$ does not change the expression.

**Restriction**: Left recursion is forbidden. I.e. for all expressions $e$ there has to be a natural number $n$ such that $\delta^n(e)$ either is the empty expression or starts with a terminal (or is a biased choice where the first alternative is either empty or starts with a terminal).

$$\begin{aligned} \delta^n(e) &= \epsilon \\ & \quad a e_1 \\ \epsilon &\,/ \ldots \\ (a e_1) &\,/ \ldots \end{aligned}$$

In order to specify a certain layout we introduce the following additional expressions.

$$\begin{aligned} e ::= & \\ & \\ & \ldots \\ & \mid \\ e^n \quad & e \text{ indented by } n \text{ columns} \\ & \mid \\ |e| & \text{vertically aligned at first token} \\ & \mid \\ e^{\circledast} & \text{detached} \end{aligned}$$

where $e^n$ describes the same structure as $e$ but indented by $n$ columns relative to its parent, $|e|$ describes the same structure as $e$ but using its first token for vertical alignment and $e^{\circledast}$ describes the structure $e$ detached from any indentation and alignment requirements.

### 2.1.2 Formal Semantics

#### Indentation of a Sequence of Tokens

An input stream $u$ consists of a finite sequence of tokens appearing at a certain column. We notate the token $a$ appearing at column $i$ as $a_i$. Therefore we have

$$u = a_i \; b_j \; c_k \; \ldots \; z_l$$

as a valid input stream.

- The indentation of a token is the column at which the token appears in the input stream.
- The indentation of a nonempty subsequence of the input stream is the column of its leftmost token.
- In an aligned nonempty subsequence of the input stream the leftmost token must be the first token.

### Transition Relation

A parsing expression $p$ is executed within a state. The state consists of the not yet consumed part of the input stream, an indentation set and an alignment flag. The execution either succeeds (notated by $\top w$) by consuming a part of the input stream $w$ or fails after consuming a part of the input stream (notated by $\bot\ w$). In the success case we can have a modified indentation set and a modified alignment flag. Therefore the right hand side of the success case is annotated as $\top^g_J w$.

$$
\begin{array}{c}
(p, wu, I, f) \Rightarrow \\
\top^g_J w \\
(p, wu, I, f) \Rightarrow \\
\bot\ w
\end{array}
$$

In all transitions we maintain the invariant that the indentation set on the right side of the transition is a subset of the indentation set on the left side of the transition.

An indentation set is the set of allowed indentations for the parsing expression.

Indentation sets are sets of natural numbers which can be described by pairs $[a, b]$ where $a$ is the lower bound and $b$ is the upper bound. The upper bound can be infinity $\infty$. A valid indentation set is never empty, i.e. we have $a \leq b$.

If the indentation set if $[j, k]$ where $k \neq \infty$ then $k$ is the column of the (up to now) encountered leftmost token. If the uppper bound is infinite, then no token has yet been encountered in the construct.

The indentation set of the start expression is $[0, \infty]$.

A parsing expression grammar successfully parses an input stream $u$, if and only if

$$(A, u, [0, \infty], -) \ \Rightarrow\ \top^f_{[j,k]} u$$

where $A$ is the start symbol of the grammar. The indentation of $u$ is $k$.

### Empty Expression

The empty parsing expression $\epsilon$ always succeeds by not consuming any token an does not change the state.

$$(\epsilon, u, I, f) \Rightarrow \top^f_I \epsilon$$

### Terminal

Let us first consider the case that the aligment flag is not set. Then the parsing expression $a$ succeeds if it encounters the token $a_i$ at an allowed position $i$. If $[j, k]$ is the indentation set then $j \leq i$ has to be satisfied for an allowed position. If the next token on the input stream is not the expected token or it is offside, then the expression fails.

$$\frac{j \leq i \wedge a = b}{(a, b_i u, [j, k], -) \Rightarrow \top^-_{[j, \min\, i\, k]} a} \qquad \frac{i < j \vee a \neq b}{(a, b_i u, [j, k], -) \Rightarrow \bot\ \epsilon}$$

The input indentation set $[j, k]$ means that up to now the leftmost token has been encountered at column $k$ (or no token has been encountered in the surrounding construct and $k = \infty$) and the minimal allowed column is $j$. The token $a$ might be the new leftmost token. Therefore on success the upper bound of the indentation set might have to be updated.

Now we consider the case that the alignment flag is set. This means that we are trying to align a construct and have not yet encountered its first token. Because we are trying to align a construct within some indentation set and the next token is the first and leftmost token of the construct, the next token must be within this indentation set.

$$\frac{i \in I \wedge a = b}{(a, b_i u, I, +) \Rightarrow \top^-_{[i,i]} a} \qquad \frac{i \notin I \vee a \neq b}{(a, b_i u, I, +) \Rightarrow \bot\ \epsilon}$$

In the success case the token is consumed, the indentation set consists only of the column of the token and the aligment flag is reset. In case of failure nothing is consumed.

### Nonterminal

If a parsing expression starts with a nonterminal, then the nonterminal has to be mapped to its parsing expression by using the function $\delta$.

$$\frac{(\delta(A)e, u, I, f) \quad \Rightarrow \quad o}{(Ae, u, I, f) \quad \Rightarrow \quad o}$$

Remember that left recursion is not allowed in a parsing expression grammar. Therefore finally some terminal will appear as the first subexpression (or whole expression becomes the empty expression).

### Sequence

For the sequence of two parsing expressions $p_1 p_2$ we have to distinguish three cases:

- The first expression fails. This implies that the whole expression fails.

$$\frac{(p_1, w_1 w_2 u, I, f) \quad \Rightarrow \quad \bot\, w_1}{(p_1 p_2, w_1 w_2 u, I, f) \quad \Rightarrow \quad \bot\, w_1}$$

- The first expression succeeds, but the second fails. This implies that the whole expression fails as well.

$$\frac{\begin{array}{ll}(p_1, w_1 w_2 u, I, f) & \Rightarrow \quad \top_J^g w_1 \\ (p_2, w_2 u, J, g) & \Rightarrow \quad \bot\, w_2\end{array}}{(p_1 p_2, w_1 w_2 u, I, f) \quad \Rightarrow \quad \bot\, (w_1 w_2)}$$

- Both expressions succeed. In that case the whole expression succeeds. The second expression uses the output state of the first as the input state. The final state of the whole expression is the final state of the second expression.

$$\frac{\begin{array}{ll}(p_1, w_1 w_2 u, I, f) & \Rightarrow \quad \top_J^g w_1 \\ (p_2, w_2 u, J, g) & \Rightarrow \quad \top_K^h w_2\end{array}}{(p_1 p_2, w_1 w_2 u, I, f) \quad \Rightarrow \quad \top_K^h (w_1 w_2)}$$

### Biased Choice

For the biased choice $p_1 \,/\, p_2$ we have to distinguish some cases:

- The first expression succeeds. In that case the whole expression succeeds with the same output.

$$\frac{(p_1, wu, I, f) \quad \Rightarrow \quad \top_J^g w}{(p_1 \,/\, p_2, wu, I, f) \quad \Rightarrow \quad \top_J^g w}$$

- The first expression fails by consuming some tokens. In that case the whole expression fails with the same result.

$$\frac{\begin{array}{ll}(p_1, wu, I, f) & \Rightarrow \quad \bot\, w \\ w \neq \epsilon & \end{array}}{(p_1 \,/\, p_2, wu, I, f) \quad \Rightarrow \quad \bot\, w}$$

- The first expression fails by not consuming any token. In that case the result of the whole expression is the result of the second expression.

$$\frac{\begin{array}{lcl}(p_1, u, I, f) & \Rightarrow & \bot\ \epsilon \\ (p_2, u, I, f) & \Rightarrow & o\end{array}}{(p_1\ /\ p_2, u, I, f) \Rightarrow o}$$

### Backtrack

The backtracking operator has no effect in the case of success. A failure with consuming tokens is converted to a failure without consuming tokens.

$$\frac{(p, uw, I, f) \Rightarrow \top^g_j w}{(\overleftarrow{p}, u, I, f) \Rightarrow \top^g_j w} \qquad \frac{(p, uw, I, f) \Rightarrow \bot\ u}{(\overleftarrow{p}, u, I, f) \Rightarrow \bot\ \epsilon}$$

### Alignment

The expression $|p|$ describes an input sequence according to $p$ where the first token is the leftmost token in the sequence and the sequence is vertically aligned according to the first token.

Clearly alignment only makes sense if there are at least two vertically aligned expressions. The expression $|p|\ |q|$ aligns the input sequences for botch expressions vertically by using the first token of each sequence for the alignment.

A sequence of aligned expressions have to be decoupled from the surrounding part of the input stream by indentation. The expression

$$(|p|\ |q|\ \ldots)^n$$

aligns the input streams described by $p$, $q$, $\ldots$ vertically and indents the whole block by $n$ columns relative to the surrounding input stream (note: the indentation can be zero). The decoupling by indentation guarantees that the effect of the alignment is only local to the vertically aligned blocks.

The transition of an aligned block is described by

$$\frac{(p, wu, I, +) \Rightarrow o}{(|p|, wu, I, f) \Rightarrow o \quad \text{adapt flag}}$$

It might be necessary to adapt the alignment flag in the output state to cover the corner case $p = \epsilon$. If the parsed sequence is not empty then it has a last token. Since each token clears the alignment flag, the initial alignment flag is cleared at the end. This is not the case for an empty sequence.

Adaption: If the alignment flag is cleared at the end, no adaption is necessary. If the alignment flag is not cleared at the end (only possible for an empty sequence of tokens) then the alignment flag is set to its initial value $f$. This makes sure that an empty aligned sequence has no effect.

### Indentation

Indentation has no effect if the alignment flag is set.

$$\frac{(p, wu, I, +) \quad \Rightarrow \quad o}{(p^n, wu, I, +) \quad \Rightarrow \quad o}$$

If the alignment flag is not set, then the transition is described by the rules

$$\frac{(p, wu, [i+n, \infty], -) \quad \Rightarrow \quad \top^f_{[k,l]} w}{(p^n, wu, [i, j], -) \quad \Rightarrow \quad \top^f_{[i,\, \min\, j\, (l-n)]} w} \qquad \frac{(p, wu, I^n, -) \quad \Rightarrow \quad \bot\, w}{(p^n, wu, I, -) \quad \Rightarrow \quad \bot\, w}$$

where $i + n \le k \le l$ and therefore $i \le l - n$ is guaranteed because of the invariant.

### Detachment

If a parsing expression has some output in a completely unrestricted environment, then the corresponding detached expression has the same output in any environment except that the initial indentation and alignment state is preserved. I.e. a detached expression runs independently from the indentation and aligment requirements.

$$\frac{(p, wu, [0, \infty], -) \quad \Rightarrow \quad \top^g_J w}{(p^{\circledast}, wu, I, f) \quad \Rightarrow \quad \top^f_I w} \qquad \frac{(p, wu, [0, \infty], -) \quad \Rightarrow \quad \bot\, w}{(p^{\circledast}, wu, I, f) \quad \Rightarrow \quad \bot\, w}$$

## 2.1.3 Implementation

In order to implement layout parsing with combinators we need an indentation set and an aligment flag in the state.

```
module Indent = struct
    type t = {
        lb:  int;           (* lower bound *)
        ub:  int option;    (* upper bound or infinity *)
        abs: bool;          (* aligment flag *)
    }

    let initial: t =
        {lb = 0; ub = None; align = false}
    ...
end
```

For each token arriving at a certain column `i` we can check, if the token is allowed at that column.

```
let check_column (i: int) (ind: t): bool =
    ind.lb <= i
    &&
    (
        match ub with
        | Some ub when ind.abs ->
            i <= ub
        | _ ->
            true
    )
```

This function only checks the correct indentation. After this check it has to be verified as usual if the token is the expected one.

If the token is in an allowed column and is an expected token, then the token can be consumed.

```
let consume (i: int) (ind: t): t =
    assert (check_column i ind);
    if not ind.abs then
        (* The token might be the new leftmost token. *)
        match ind.ub with
        | Some ub when ub <= i ->
            ind
        | _ ->
            {ind with ub = Some i}
    else
        (* First token in an aligned structure *)
        {
            lb  = i;
            ub  = Some i;
            abs = false;
        }
```

Remember: An upper bound, if present, marks the column of the leftmost token up to now. The consumed token might be the new leftmost token. If this is the case, the structure has to be updated.

The function `align` sets the alignment flag and the function `end_align` handles the corner case of an empty aligned structure.

```
let align (ind: t): t =
    {ind with abs = true}

let end_align (ind0: t) (ind: t): t =
    (* [ind0] is the indentation state at the start *)
    if not ind.abs then
        (* flag is cleared, the aligned sequence is not empty. *)
        ind
    else
        (* the aligned sequence is empty and therefore must not have any
           effect *)
        {ind with abs = ind0.abs}
```

In order to handle indentation properly we need functions to start and end an indented block.

```
let start_indent (i: int) (ind: t): t =
    assert (0 <= i);
    if ind.abs then
        (* No effect on aligned structures which have not yet received
           a first token. *)
        ind
    else
        match ind.ub with
        | None ->
            (* It does not make sense to indent relative to something
               which does not yet have any token. *)
            ind
        | Some ub ->
            {
                lb  = ub + i;
                ub  = None;
                abs = false;
            }

let end_indent (ind0: t) (ind: t): t =
```

(continues on next page)

(continued from previous page)

```
    if ind0.abs || ind0.ub = None then
        ind
    else
        ind0
```

# BIBLIOGRAPHY

[AAugacan14] Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, 121–132. New York, NY, USA, September 2014. ACM. URL: https://michaeldadams.org/papers/layout_parsing_2/LayoutParsing2-2014-haskell-authors-copy.pdf, doi:10.1145/2633357.2633369.