```
l!=create⇒length(l) ≡ length(rest(l)) + 1 // A6
nth(l,1) ≡ first(l) // A7
i>1⇒nth(l,i) ≡ nth(rest(l),i-1) // A8
add(l,1,e) ≡ cons(e,l) // A9
i>1⇒add(l,i,e) ≡
       cons(first(l),add(rest(l),i-1,e) // A10
remove(l,1) = rest(l) // A11
i>1⇒remove(l,i) ≡
       cons(first(l),remove(rest(l),i-1) // A12
```

Partial operations *remove, add* and *nth* are as usual completed with a precondition.

A few notes about axioms:

- A5: the length of a newly created list is 0 (it's empty).
- A6: When an element is added, the length is increased by 1
- A7: The element at position 1 is the first (!)
- A8: The element at position i in a list is the one at position $i-1$ in the rest of that list
- A9: Adding an element at position 1 is "consing" the element (head-addition)
- A10: Adding an element at position i in a list is adding it at position $i-1$ in the rest of that list
- A11: removing the first element of a list results in getting the rest of that list
- A12: Removing an element at position i in a list is removing it at position $i-1$ in the rest of that list

4 Binary trees

Binary trees are one of the simplest forms of tree structures. They are of great importance because they offer a rather simple way to obtain a good level of performance (in average) when searching, adding or removing information.

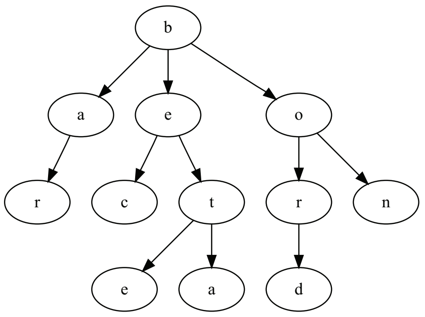

Figure 2: A (part of french) dictionary tree

After an informal definition, we open this section with a convincing (at least we hope so) example of usage of trees for information storage and retrieval. Then the particular case of binary trees is defined more formally. Several properties on binary trees (BT) are then proved using an inductive approach. The rest of the section presents an important type of BT, the binary search trees (BST).

4.1 Principle of tree-like structures

A tree is composed of a node (called its root) containing information, and of a (possibly empty) set of trees (called its children). This definition is clearly recursive, as will be most of the definitions of operations on trees.

4.2 Why use trees? For performance!

Let's go back to the introductory example, *i.e.* searching for a word in a dictionary. We saw that, in the case where the dictionary is stored in a sorted array, a search by induction leads to a logarithmic complexity (in terms of the dictionary size). However, if we want to add a new word in the dictionary, we'll have to move a set of elements (namely all elements that are after the new word in the alphabet) to make room for the new word in the array. That process is clearly linear (in average, we have to move half the array).

If, instead of an array, we choose a tree as the underlying structure for the dictionary, we can associate a character to each node of the tree. For instance, if we want to add the word "bond" (*jump*) to the dictionary represented in figure 2, we follow the link that leads from the root node to the node storing "b", then go down to "o", then to "n". We just then have to add a child node to "n" that contain "d". It is important to notice that the process described here is *independent* from the size of the dictionary: it only depends (linearly) on the size of the word to insert!

4.3 The particular case of binary trees - equivalence

Binary trees are a category of trees for which the number of children is less than or equal to two. This constraint on the structure has two advantages:

1. It allows to dramatically simplify the specification and implementation of most operations.
2. Any (general) tree can be transformed (using a simple rule) into a binary tree, so meeting this constraint does not reduce the generality of the model.

4.4 Definitions

A binary tree (BT) is either empty, or made of a *root* node containing an element and two BT called *left child* and *right child*.

The minimum specification is then rather straightforward:

```
ADT BinaryTree
Uses Element
operations
  create  : → BinaryTree
  <_,_,_> : Element × BinaryTree × BinaryTree  → BinaryTree
  root    : BinaryTree ⇀ Element
  left    : BinaryTree ⇀ BinaryTree
  right   : BinaryTree ⇀ BinaryTree
pre-conditions
  root(a)) iff a ≠ create
  left(a)) iff a ≠ create
  right(a)) iff a ≠ create
axioms
  root(<r,L,R>) ≡ r
  left(<r,L,R>) ≡ L
```

```
right(<r,L,R>) ≡ R
```

The above definition is very simple because incomplete. Any operation added involves an extension to this minimum ADT.

Let's conclude this section with a focus on terminology:

1. A node without any child is called a *leaf*; all other nodes are called *internal nodes.*
2. A BT is called *degenerate* (or *stringy*) if it contains only one leaf.
3. A BT is called *complete* if all its internal nodes have two children and all leaves are at the same level.
4. The *size* of a tree is the number of nodes. An empty tree is then of size 0.
5. The *height* (or *depth*) of a node is the number of links in a path going from the root to this node. The height of a BT is the maximum value of the heights of its nodes. By convention, an empty tree is of height -1.
6. The *path length* (resp. *internal*, *external* path lengths) of a tree is the sum of the heights of its nodes (resp. its internal nodes, its leaves)

4.5 Important properties on binary trees

1. For a binary tree of size n and height h, we have

$$\lfloor \log_2 n \rfloor \leq h \leq n-1 \tag{1}$$

2. For a binary tree of size n and path length PL, we have

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor \leq PL \leq \frac{n(n-1)}{2} \tag{2}$$

Those properties (the reader is urged to try to prove them as an exercise) are important since they show that it is possible, for a certain category of binary trees (namely the binary search trees), to search for and find an element in the tree in a time proportional to the *logarithm* of the tree size. That makes the main advantage of tree structures. Of course, inequation 1 also shows that this time can become linear in the worst case (degenerate trees). Nonetheless, it can be shown for BSTs that, in average, the height is proportional to the logarithm of the size.

4.6 Binary tree paths

There are numerous ways to visit all the elements of a tree once and only once. Here we focus on a particular kind of traversal, the depth-first left-hand path. Such an algorithm involves three possible states when visiting a node (first time we meet the node, come back from left child exploration and come back from right child exploration), hence three possible visit operations on nodes. This algorithm looks like this:

1. If tree is empty, we stop
2. otherwise,
   (a) call $visit_1$ on root
   (b) recursively call procedure for left child
   (c) call $visit_2$ on root
   (d) recursively call procedure for right child
   (e) call $visit_3$ on root

There are three important particular cases of this algorithm:

1. *preorder* traversal, where only $visit_1$ is non-empty;
2. *inorder* traversal, where only $visit_2$ is non-empty;
3. *postorder* traversal, where only $visit_3$ is non-empty.

As we'll see in an exercise, in certain cases (BST) those traversals do not carry the same information.

4.6.1 Specification for `size`, `height` and `PL` operations

```
extension ADT BinaryTree
Uses Natural
operations
  size  : BinaryTree → Natural
  height: BinaryTree → Natural
  PL    : BinaryTree → Natural
axioms
  size(create) ≡ 0
  size(<r,L,R>) ≡ 1 + size(L) + size(R)
  height(create) ≡ -1
  height(L) ≤ height(R) ⇒
    height(<r,L,R>) ≡ 1 + height(R)
  height(L) > height(R) ⇒
    height(<r,L,R>) ≡ 1 + height(L)
  PL(create) ≡ 0
  PL(<r,L,R>) ≡ PL(L) + PL(R) + size(L) + size(R)
```

4.6.2 Example of proof by induction on a binary tree

Let a be a complete binary tree of height h. Compute the number of leaves in a.

We are going to apply the same kind of induction approach as we used in high school to prove properties on natural numbers: *(i)* we prove that the property is true for one / several cases of trees, and *(ii)* we prove that if the property is true for any tree of size smaller than n, then it's true for a tree of size n.

Let $nl(a)$ be the number of leaves we want to compute. If $h = 0$, we have $nl(a) = 1$. Pour $h = 1$, we get $nl(a) = 2$. For $h = 2$, $nl(a) = 4$. Can we prove that, in the general case, we have $nl(a) = 2^h$?

Proposition is true for values 0, 1 and 2 of h. We will suppose that it is true for any value until $h - 1 > 1$ and show that it is still true for h.

If a is complete of height h, then *left(a)* is complete of height $h - 1$. We then have $nl(left(a)) = 2^{h-1}$.

Same thing applies to right subtree of a. Since the number of leaves of a

BT is the sum of that of its two children, we have

$$\begin{aligned} & nl(a) = nl(left(a)) + nl(right(a)) \\ & = 2^{h-1} + 2^{h-1} \\ & = 2^h \end{aligned}$$

5 Binary search trees (BST)

In this section, we consider a particular type of binary tree that relies on the existence of an order relation on the elements (we will represent this relation by a *key()* integer function, defined on the whole *Element* domain) to choose a place for them in the tree during insertion, lookup and removal.

In a BST, elements are stored in such a way that, for any node n,

- all elements in the left child have a key value less than that of n,
- all elements in the right child have a key value greater than that of n.

To simplify, we'll suppose there is no two elements with the same key value in the tree. We start from the base given by the *BinaryTree* ADT and add peculiar operations for searching, adding and removing an element.

5.1 Searching for an element

The algorithm is simple: we compare the element to the root of the tree; if they are equal, the element is found; if the element is greater than (resp. less than) the root, we search for it in the right (resp. left) child. The algorithm stops when either the tree is empty (failure) or the element is equal to the root (success).

The extension can be defined as follows:

```
extension ADT BST
Uses Element, boolean
operations
  contains: BST × Element → boolean
axioms
  contains(create,e) ≡ false                 // A1
  contains(<r,L,R>,r) ≡ true                 // A2
  key(x) < key(r) ⇒
```

```
    contains(<r,L,R>,x) ≡ contains(L,x)   // A3
  key(x) > key(r) ⇒
    contains(<r,L,R>,x) ≡ contains(R,x)   // A4
```

A few notes about these axioms:

- A1 states that an empty tree does not contain any element;
- A2 says that if a BST is non-empty, it contains at least its root;
- A3 and A4 deal with the division of the search space according to the relation between the root and the searched element.

5.2 Adding an element to a BST

The insertion technique described here is called "leaf insertion", because every new element becomes a leaf of the tree. There are other approaches (for instance root insertion, where the new element becomes the root of the tree), we let them in exercise for the reader.

The idea of the algorithm is simple:

- If the initial tree is empty, the result is a BST reduced to its root containing the new element.
- If the element is already in the tree, we do nothing (unicity of the keys / elements).
- Otherwise the addition takes place either in the left subtree (if the element's key is less than than of the root), or in the right one (if the element's key is greater than that of the root).

```
extension ADT BST
Uses Element
operations
  add: BST × Element → BST
axioms
  add(create,e) = <e,create,create> // A5
  add(<r,L,R>,r) = <r,L,R> // A6
  key(x) < key(r) ⇒
    add(<r,L,R>,x) = <r,add(L,x),R> // A7
  key(x) > key(r) ⇒
    add(<r,L,R>,x) = <r,L,add(R,x)> // A8
```

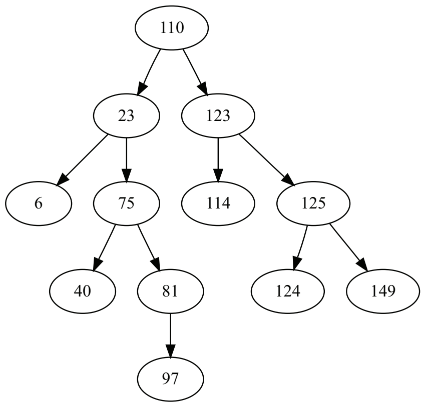

Figure 3: The initial sample BST

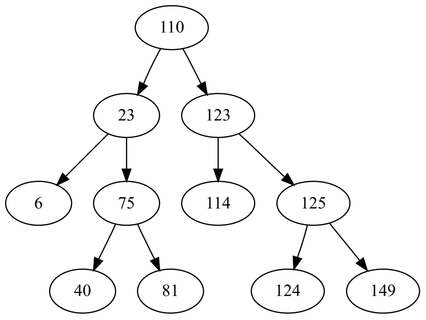

Figure 4: The initial BST after the removal of a leaf (here, 97)

5.3 Removing an element

The removal is more complex since it must not involve any disorganization in the structure of the tree: some nodes may have to be moved to keep the BST property.

Several possible situations must be dealt with. To simplify things for the reader, we identify the elements of the tree with their key, as shown in figure 3:

1. The element to remove is a leaf of the tree. It is the simplest case: no reorganisation. This simple situation is described in figure 4, which show the result after removing the element whose key is 97.

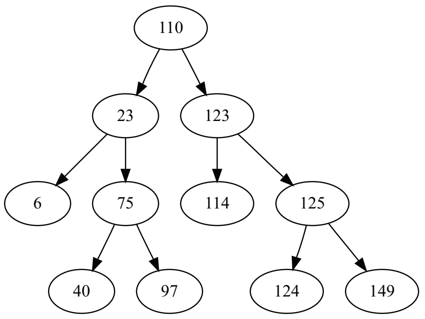

Figure 5: The initial BST after removal of an element with only one child (here 81)

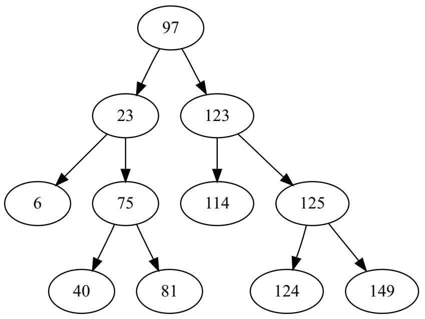

Figure 6: The BST after removal of a two-children node (here 110)

2. The element to remove has only one child: we just have to make this child replace the element to remove, it is clear that the BST property is preserved. This situation is illustrated in figure 5, where we removed the element whose key is 81 from the initial BST.

3. The element to remove has two children. We then have only two possibilities to preserve the BST property: we replace it with either the greatest element of the left child, or the smallest of the right child. Let's choose for instance the greatest element of the left child, as shown in figure 6, where we removed the root from the initial BST (key 110).

To implement this removal operation, we are going to use two auxiliary operations that must then be specified too:

- the `max` operation that returns the element of maximum key in the BST;
- the `delMax` operation that returns the BST minus its max element.

Thus, we get the following specification for remove:

```
extension ADT BST
Uses Element
operations
  max   : BST ⇀ Element
  delMax: BST → BST
  remove: BST × Element → BST
pre-conditions
  max(a)) ⇒ a ≠ create
axioms
  max(<r,L,create>) ≡ r                      // A9
  R≠create ⇒ max(<r,L,R>) ≡ max(R)        // A10
  delMax(create) ≡ create                     // A11
  delMax(<r,L,create>) ≡ L                    // A12
  L≠create ⇒
delMax(<r,L,R>) ≡ <r,L,delMax(R)>             // A13
  remove(create,e) ≡ create                   // A14
  key(x) < key(r) ⇒
    remove(<r,L,R>,x) ≡ <r,remove(L,x),R>     // A15
  key(x) > key(r) ⇒
    remove(<r,L,R>,x) ≡ <r,L,remove(R,x)      // A16
  remove(<r,create,R>,r) ≡ R                  // A17
  L≠create ⇒
    remove(<r,L,R>,r) ≡ <max(L),delMax(L),R> // A18
```

A few notes about the axioms:

- A9 and A10 formalize the fact that, by construction, the element with maximum key is always the rightmost element in a BST: the root if no right child (A9), and the max of the right child if any (A10).
- A11, A12 and A13 define the resulting BST when removing the max element in a given BST: if the tree is empty, nothing changes (A11); if

there is no right child, the result is the left child (since the root happens to be the max element, A12); finally, in the general case, removal takes place in the right child (A13).

- A14 to A18 use the previous two operations to specify general remove operation: if the tree is empty, nothing happens (A14); go down into right (*resp.* left) child if the element to remove has a greater (*resp.* lesser) key than the root (A15, A16); removal of the root, which returns the right child if the left child is empty (A17); finally, replacement of the root by the righmost element in the left child if not empty (A18).

5.4 Complexity of main BST operations

The study of those operations on BST shows that recursive calls are almost always made on a child of the tree, thus reducing the height of the parameter tree by 1. That is why most of those operations have a complexity proportional to the *height* of the tree.

The goal of this section is to show that the height of a BST is, in average, proportional to the logarithm of its size. We start showing this is true in the optimal case of a complete BST, but false in the (worst) case of a degenerate BST. We then conclude in the general case.

5.4.1 Minimum depth of a BST

For a given height h, the biggest possible BST is obviously complete. Let's try to prove (using induction) that the maximum size is then $N = 2^{h+1} - 1$.

The property is obviously true for $h = 0$ (tree reduced to its root). Let's assume it's true for any BST of height less than h. For a BST of height h, we have:

$$\begin{aligned} N &= N_l + N_r + 1 \\ &= 2^{(h-1)+1} - 1 + 2^{(h-1)+1} - 1 + 1 \\ &= 2^h + 2^h - 1 \\ &= 2^{h+1} - 1 \end{aligned}$$

which gives

$$2^{h+1} = N + 1$$

$$\begin{aligned} h+1 &= \log(N+1) \\ h &= \log(N+1)-1 \end{aligned}$$

The depth of the tree is logarithmic, as expected.

5.4.2 Maximum depth of a BST

The worst case is reached when the tree is degenerate (stringy): the depth is then of the same order as the size. Let's prove that, more precisely, we have $N = h + 1$.

The property is obviously true for the tree reduced to its root ($h = 0, N = 1$). Let's suppose it is true for any degenerate tree of height less than h. Let's consider a degenerate tree of height h, and suppose its left child is empty. Its right child being degenerated, its height is $h - 1$ and, by hypothesis, its size is $N - 1 = h$. Then our new tree has a size of $h + 1 = N$.

5.4.3 Average depth for a BST

We consider a random BST. This means that all possible sequences of insertions have the same probability to occur. Let's compute the internal path length *ipl* of such an "average" BST of size N.

We clearly have $ipl(1) = 0$. For any $N > 0$ we have i nodes in the left child and $N - i - 1$ in the right child. All nodes but the root have a depth in the tree equal to the one they have in a child, plus 1. We can then write

$$ipl(N) = ipl(i) + ipl(N - i - 1) + N - 1$$

All possible values for i being equiprobable (probability is $1/N$), the average value for $ipl(i)$ (as that of $ipl(N - i - 1)$ by the way, for symmetry reason) is

$$\frac{1}{N}\sum_{j=0}^{N-1} ipl(j)$$

which gives

$$\begin{aligned} ipl(N) &= \frac{2}{N}\left(\sum_{j=0}^{N-1} ipl(j)\right) + N - 1 \\ N.ipl(N) &= 2\left(\sum_{j=0}^{N-1} ipl(j)\right) + N(N-1) \end{aligned}$$

$$\begin{aligned}
(N-1)ipl(N-1) &= 2\left(\sum_{j=0}^{N-2} ipl(j)\right) + (N-1)(N-2) \\
N.ipl(N) - (N-1)ipl(N-1) &= 2ipl(N-1) + 2(N-1)
\end{aligned}$$

By rearranging the terms of this equation and removing constant 2 (negligible in the right term), we get

$$N.ipl(N) = (N+1)ipl(N-1) + 2N$$

which, by dividing both sides by $N(N+1)$, becomes

$$\begin{aligned}
\frac{ipl(N)}{N+1} &= \frac{ipl(N-1)}{N} + \frac{2}{N+1} \\
\frac{ipl(N-1)}{N} &= \frac{ipl(N-2)}{N-1} + \frac{2}{N} \\
&\vdots \\
\frac{ipl(2)}{3} &= \frac{0}{2} + \frac{2}{3}
\end{aligned}$$

Summing all left members, then all right members, we get

$$\frac{ipl(N)}{N+1} = 2\sum_{i=3}^{N+1} \frac{1}{i}$$

The sigma is about $\ln(N+1) + \gamma - \frac{3}{2}$, where $\gamma \simeq 0,577$ (the Euler constant). This proves that $ipl(N) \in N \log N$, and then that the average depth of a node is in $O(\log N)$. The average number of recursive calls for search, insertion and removal of elements in a BST will then be logarithmic.