Projet de

Spécifications Formelles

–

Structures de données efficaces

Réalisé par

Dalbert Jérôme
Foulfoin Anthony
Pastor Harper Javier

Table des matières

1. Présentation

L'objectif de ce projet est de développer des implantations d'une structure de données ordonnée efficace pour laquelle on dispose d'opérations de type Liste (insertion d'élément en tête, consultation ou retrait de l'élément de tête) et de type Tableau (consultation ou modification du k-ième élément).

Ce type de structure de données dynamique est communément appelé Random Access List.

Avant l'étape de réalisation proprement-dite, nous avons fait des recherches bibliographiques sur les différentes solutions permettant des opérations arithmétiques rapides. Ces recherches ont porté sur deux domaines : les systèmes arithmétiques alternatifs et les circuits additionneurs. Elles nous ont notamment servi à choisir un type d'implantation pour un raffinement donné.

2. Bibliographie

2.1 Redundant number system

2.1.1 Principe

Un système binaire redondant (redundant binary representation (RBR)) est un système numérique dans lequel chaque nombre possède plusieurs représentations possibles.

Dans le système binaire classique, la valeur entière d'un nombre se calcule en effectuant une somme pondérée, à chaque digit est associé un poids. Le poids démarre à 1 pour la position la plus à droite, le poids faible, et est multiplié par 2 pour chaque position suivante. Le système binaire possède deux digits : 0 et 1. Pour un nombre donné, une seule représentation binaire est possible.

Le RBR utilise un digit supplémentaire : -1. Ce système peut permettre de modéliser des nombres négatifs. Ce système utilisera donc les 3 digits {-1,0,1}. Avec ce système, plusieurs représentations sont possibles pour le même nombre. Essayons par exemple représenter le nombre 7.

- En binaire, la représentation sera : 0111 = 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 7
- En RBR, la représentation sera : 0111 = 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 7

  OU 100-1 = 1*2^3 + 0*2^2 + 0*2^1 -1*2^0 = 8 - 1 = 7

  OU 1-111 = 1*2^3 -1*2^2 + 1*2^1 + 1*2^0 = 8 - 4 + 2 + 1 = 7

2.1.1 Avantages

Imaginons que l'on souhaite incrémenter le nombre 7 et la valeur 1 pour obtenir 8. La représentation de 7 en binaire est **0111**, et 8 est représenté par **1000,** on remarque donc qu'il y aura propagation d'une retenue tout au long de la représentation binaire. Le temps de calcul sera donc fonction de la longueur en bit du nombre à calculer.
Avec la représentation en MBR, nous représentons 7 par **100-1** auquel nous souhaitons ajouter la valeur 1 **0001**. Le résultat du calcul est **1000** = 8, et l'on remarque qu'il n'y a pas eu ici d'usage de retenue.

Cette représentation permet donc d'effectuer des additions en temps constant, d'une complexité en O(1).

2.1.2 Inconvénients

Si l'on souhaite convertir un nombre RBR vers sa représentation en complément un 2, cela ne peut s'effectuer que via un algorithme dont la complexité au mieux est en O(log(n)).

2.2 Fibonacci number system

2.2.1 Principe

La suite de Fibonacci est une suite d'entiers très connue. Celle-ci est définie par récurrence grâce à la formule suivante :

$F_0 = 1$ $F_1 = 2$ $F_{n+2} = F_n + F_{n+1}$

Voici par exemple les 10 premiers nombres de la suite de Fibonacci :

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9
0	1	1	2	3	5	8	13	21	34

Le système binaire de Fibonacci (FNS) est très proche du système binaire classique. Celui-ci considère chaque nombre de Fibonacci comme étant un poids, coefficienté par 1 ou 2. F0 constitue le poids faible.

Ainsi un entier peut être exprimé par la formule $\sum_{j=1}^{k-1} a_j F_j$ où k représente la longueur du nombre en système binaire de Fibonacci, a_j un digit égal à 1 ou 2, et F_j le $j^{ème}$ nombre de Fibonacci.

Prenons un exemple, avec le nombre 5.

En binaire celui-ci est représenté par **0101**.

En FNS, celui-ci est représenté par **121** = 1*F1 + 2*F2 + 1*F3 = 1 + 2 + 3 = 5

OU par **211** = 2*F1 + 1*F2 + 1*F3 = 2 + 1 + 3 = 5

Deux représentations sont ici possibles pour le nombre 5. On s'aperçoit donc que le FNS est un système numérique **redondant**.

2.2.2 Avantages

Il s'agit d'un système redondant, on peut donc trouver une représentation de chaque nombre permettant de limiter la propagation des retenues en cas d'incrémentation.

2.2.3 Inconvénients

Il est nécessaire de construire la suite de Fibonacci pour pouvoir calculer la valeur décimale d'un nombre. Ce système numérique redondant n'offre aucun avantage supplémentaire par rapport au système exposé précédemment.

2.3 1-2 binary system

2.3.1 Principe

Dans le système binaire classique, la valeur entière d'un nombre se calcule en effectuant une somme pondérée, à chaque digit est associé un poids. Le poids démarre à 1 pour la position la plus à droite, le poids faible, et est multiplié par 2 pour chaque position suivante. Le système binaire possède deux digits : 0 et 1. Pour un nombre donné, une seule représentation binaire est possible.

Dans le système binaire 1-2, les deux digits 0 et 1 sont remplacés par 1 et 2.

Ainsi le nombre 5 en binaire s'écrit **101**, alors qu'en binaire 1-2 celui-ci s'écrira **21.** Il ne s'agit pas d'un système redondant, il n'y a qu'une seule représentation possible de chaque nombre.

2.3.2 Avantages

Cette méthode n'offre aucun avantage apparent par rapport au système binaire classique.

2.3.3 Inconvénients

Cette méthode n'offre aucun inconvénient apparent par rapport au système binaire classique, mais est cependant moins efficace que le système redondant, dans la mesure où il peut il y avoir des propagations de retenues.

2.4 0-1-2 system

2.4.1 Principe

Le système 0-1-2 est un système en base 3 où les digits sont 0, 1 et 2.
Ainsi, 19 en système 0-1-2 s'écrit **201** (alors qu'il s'écrit **10011** en binaire).
Voici le détail de la décomposition en base 10 : 2*3^2 + 0 + 1*3^0 = 18 + 1 = 19.
Pour un nombre donné, une seule représentation binaire est possible.

2.4.2 Avantages

La formule générale de décomposition en base 10 du nombre $d_3d_2d_1d_0$ (où d représente un digit) est :

$d_3*b^3 + d_2*b^2 + d_1*b^1 + d_0*b^0$ (où b est la base).

Soit l la longueur d'un nombre (i.e. son nombre de digits).

Le coût de la représentation d'un nombre est fonction de l et de b. Pour minimiser ce coût, on veut donc minimiser le produit de l*b tout en gardant b^l constant. Il a été prouvé que la base optimale est e (environ 2,718). Comme 3 est l'entier le plus proche de e, c'est la meilleure base entière : elle permet de représenter les nombres de la façon la plus économique possible.

Le système 0-1-2 est donc notamment meilleur que le système binaire, en termes de coût.

2.4.3 Inconvénients

Dans le marché courant, il n'y a quasiment pas de composants à 3 états qui pourraient permettre de manipuler des nombres dans cette base.

2.5 Skew number system

2.5.1 Principe

Contrairement au système binaire, où le poids de chaque digit à la position n est 2^n, le skew number system a le poids de chaque digit à 2^{n+1}-1.

De plus, les digits permis sont 0, 1 et 2. Mais un 2 est uniquement autorisé à l'emplacement du premier bit de poids faible non nul. 2 ne peut donc apparaître **qu'une seule fois**.

Ainsi, dans ce système, 92 s'écrit **101200** (alors qu'il s'écrit **1011100** en binaire).

Voici le détail de la décomposition en base 10 :
1*(2^6-1) + 0 + 1*(2^4-1) + 2*(2^3-1) + 0 + 0 = 63 + 15 + 14 = 92.

Pour un nombre donné, une seule représentation binaire est possible.

Les poids des bits évoluent de la façon suivante (en partant du poids faible) :
1, 3, 7, 15, 31, 63, 127, 255, 1023, 2047, etc.
Donc on compte comme ceci (attention, jusqu'à la fin de cette sous-partie, les poids faibles sont dorénavant à gauche) :
1, 2, 01, 11, 21, 02, 001, 101, 201, 011, 111, 211, 021, 002, 0001, 1001, 2001, etc.
On remarque que pour incrémenter un de ces nombres :

- soit le poids faible est 0, alors on lui ajoute 1 (le 0 devient donc un 1)
- soit le poids faible est 1, alors on lui ajoute 1 (le 1 devient donc un 2)
- soit le poids faible est 2, alors on le remplace par un 0 et on ajoute 1 au chiffre suivant.

Or, dans le 3ème cas, le chiffre suivant ne peut être que 0 ou 1. En lui ajoutant 1, il deviendra donc soit 1, soit 2.

L'incrémentation est donc terminée. Il n'y a rien d'autre à faire.

2.5.2 Avantages

Dans le système binaire, l'addition de deux chiffres dans un nombre engendre éventuellement une retenue qui peut se propager jusqu'au dernier chiffre.

Dans le skew number system, l'addition de deux chiffres engendre éventuellement une retenue pour le chiffre suivant, mais c'est tout : il n'y a plus de propagation après. L'incrémentation est donc beaucoup plus rapide : elle se fait **en temps constant** (O(1)).

2.5.3 Inconvénients

Cette méthode n'offre aucun inconvénient apparent par rapport au système binaire classique.

2.6 Fast two-operand adders

L'inconvénient d'un additionneur ripple-carry (qui n'est constitué que de plusieurs modules additionneurs complets mis bout à bout) est sa lenteur. En effet, il dépend du temps de propagation de la retenue de module en module.

Le but d'un additionneur rapide est donc de manipuler les retenues de telle sorte que la vitesse soit améliorée. Les additionneurs (tels que carry lookahead) présentés dans les sous-parties suivantes sont des exemples d'additionneurs rapides.

2.7 Carry look-ahead

2.7.1 Principe

La méthode « carry look-ahead » augmente la vitesse d'exécution en réduisant le temps de calcul du bit de retenue. En effet, cette méthode permet d'anticiper la valeur qu'il va prendre.

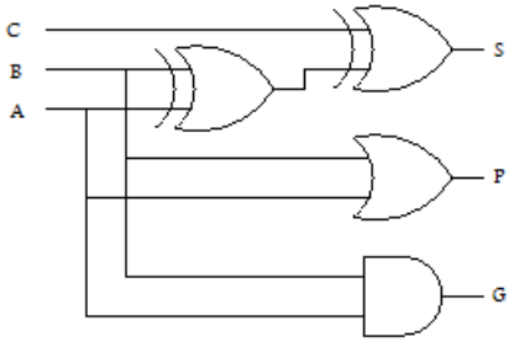
Dans un additionneur classique (sans carry look-ahead, ie : avec propagation de retenue), le calcul du bit de retenue (C) est calculé en parallèle à la valeur du bit somme (S). Ainsi, pour chaque bit on doit attendre que le bit de retenue précédent soit calculé avant de pouvoir commencer le calcul du bit S et C suivant, la propagation de retenue peut donc prendre beaucoup de temps même pour un additionneur 16 bits. Comme cette lenteur est due au temps nécessaire à la propagation de la retenue, il a été impératif d'anticiper les retenues. L'additionneur utilisant le principe du « carry look-ahead » calcule un ou plusieurs retenues avant de calculer le bit somme, ce qui réduit le temps de calcul final.

Exemple :

On introduit les valeurs intermédiaires G et P pour augmenter la lisibilité du calcul des retenues.

Etape i :	Etape i +1 :	Etape i+2 :
$G_i = A_i$ et B_i , $P_i = A_i$ ou B_i	$C_{i+1} = G_i$ ou (P_i et C_i)	$C_{i+2} = G_{i+1}$ ou (P_{i+1} et C_{i+1})

De cette manière on peut calculer les retenues d'une étape « i » plusieurs étapes « i - k » à l'avance.



2.7.2 Avantages

L'anticipation des retenues permet de diminuer de manière importante le temps de calcul final car il n'y a pas de propagation de la retenue, on supprime ainsi du temps d'attente.

2.7.3 Inconvénients

Pour ne pas propager les bits de retenue on rajoute des portes logiques, ce qui complexifie rapidement le circuit.

2.8 Carry-free addition

2.8.1 Principe :

Les additionneurs « carry-free addition » sont découpés en bloc, dans lesquels on peut prévoir la retenue de sortie en fonction de la retenue en entrée. En effet, chaque bloc peut : supprimer une retenue (carry à 0), ou créer une retenue (carry à 1), ou propager la retenue en entrée (carry à 1 ou 0).

Exemple :

Si $A_i = B_i = 0$, $C_{i+1} = 0$: pas de retenue quelque soit C_i

Si $A_i = B_i = 1$, $C_{i+1}=1$: il y a forcément une retenue, quelque soit C_i

Si $A_i \neq B_i$, , $C_{i+1} = C_i$: la retenue sortante de l'étage i est égale à la retenue entrante C_i

Exemple sur des blocs 4 bits:

Propagation de retenue:	Suppression de retenue :	Création de retenue:
A : 0101	A : 0101	A : 1101
B : 1010	B : 0010	B : 1010

2.8.2 Avantages

L'avantage pour un additionneur n bits, est qu'on aura rapidement les bits de retenue pour calculer les bits de poids fort. Ce qui diminue fortement le temps de calcul final.

2.8.3 Inconvénients

Le découpage des n bits en k blocs est arbitraire et peut ne pas être optimal. De plus, si l'on doit faire une propagation de retenue entre chaque bloc, cette méthode ne présente aucun intérêt. Cette méthode complexifie le circuit.

2.9 Conditional carry

2.9.1 Principe :

Afin de diminuer de moitié le temps de propagation (du pire cas) d'un additionneur, on peut diviser l'additionneur en deux additionneurs de longueur égale. Ainsi, pour une addition sur 8 bits, on aurait deux additionneurs de 4 bits; l'un pour la partie de poids faible et l'autre pour la partie de poids fort. Pour éviter que l'addition de la partie de poids fort soit sujette au délai de propagation de la retenue à travers la partie de poids faible, on duplique l'additionneur de poids fort, et on effectue immédiatement l'addition pour les deux cas: une retenue entrante de 1 et une retenue entrante de 0. Lorsque la retenue sortante du bloc de poids faible devient disponible, les deux résultats de poids forts sont également stables et il suffit de choisir le bon via un multiplexeur.

2.9.2 Avantages

Les additionneurs « conditional carry » permettent de faire des circuits plus rapides, car le temps de stabilisation du signal à la sortie peu être divisé par deux.

2.9.3 Inconvénients

Le circuit peu devenir rapidement complexe (occuper un espace important) car on duplique des parties du circuit. En effet, pour un additionneur 16 bit on part de 4 additionneur 4 bit pour arriver à 1 + (3 * 2) = 7 additionneur 4 bit.

2.10 Radix-2 adders

Les « radix-2 adders » sont utilisés pour des systèmes embarqués qui veulent être certain d'avoir un résultat stable en un temps borné. Ce sont donc des additionneurs qui donnent un résultat stable à chaque top d'horloge.

2.11 Pipelined adders

Les « pipelined adders » ont un mécanisme permettant d'accroître la vitesse de calcul. L'idée générale est d'appliquer le principe du travail à la chaîne. Dans un additionneur sans pipeline, le calcul est fait du bit de poids le plus faible, au bit de poids le plus fort. Le calcul d'un bit ne commence que lorsque le bit précédent est complètement stable. Avec un pipeline, l'additionneur commence le calcul du bit suivant avant d'avoir fini le précédent. Plusieurs bits se trouvent donc simultanément en cours de calcul dans l'additionneur. Le temps de calcul d'un seul bit n'est pas réduit. Par contre, le temps de calcul, c'est-à-dire le nombre de bit additionné par unité de temps, est augmenté. Il est multiplié par le nombre de bits qui sont calculés simultanément.

3. Réalisation

3.1 ralR1

ralR1 est le premier raffinement d'une Random Access List, qui étend la représentation binaire des entiers, et le type Liste.

Pour cela, ralR1 dispose des opérations implantées dans compteur_binaire, c'est-à-dire :

Init(etat) : permet d'initialiser la structure

Act_Insert(param, etat, etat_p, result) : permet d'incrémenter le compteur

Act_Remove(param, etat, etat_p, result) : décrémente le compteur

Act_Cardinal(param, etat, etat_p, result) : retourne la valeur entière du compteur

ralR1 dispose également des opérations supplémentaires implantées dans Liste, c'est-à-dire :

Act_Put(param, etat, etat_p, result) : insère l'élément à l'indice indiqué dans la liste

Act_Get(param, etat, etat_p, result) : retourne l'élément à l'indice indiqué dans la liste

Act_Occurrence(param, etat, etat_p, result) : retourne le nombre d'occurrences de l'élément dans la liste.

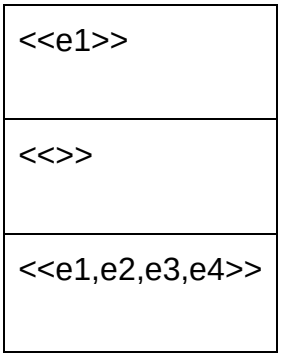
3.1.1 La structure de données

La structure de donnée implantée permet de représenter à la fois un compteur binaire et une liste.

Ainsi, la représentation du compteur binaire se fait via l'utilisation d'un tableau à N cases, N étant le nombre de bits du compteur.

Chacune de ces cases contiendra une liste d'éléments (représenté par la structure power2liste) dont le cardinal est égal au « poids » de la case. La première case possède un poids de 1, et chaque case suivante possède un poids deux fois plus important que la précédente.

Ainsi si l'on souhaite représenter le nombre 101 avec cette structure, cela correspond à 1*2^2 + 0*2^1 + 1*2^0 = 4 + 0 + 1. 4, 0 et 1 représentent donc le nombre d'éléments dont devront disposer les listes de chacune des cases du tableau :



Nous obtenons ainsi une liste à 5 éléments représentant également un entier binaire correspondant au cardinal de cette liste.

3.1.2 Détail des opérations :

Init(etat) : permet d'initialiser la structure, c'est-à-dire un tableau contenant N cases. Chaque case contenant une structure power2liste.

Act_Insert(param, etat, etat_p, result) : permet d'insérer un élément dans la liste et d'incrémenter le compteur. Le fonctionnement de cette opération est similaire à l'opération définie dans compteur_binaire.

Act_Remove(param, etat, etat_p, result) : permet de retirer un élément de la liste et de décrémenter le compteur. Le fonctionnement de cette opération est similaire à l'opération définie dans compteur_binaire.

Act_Cardinal(param, etat, etat_p, result) : retourne le nombre d'éléments dans la liste, correspondant à la valeur du compteur binaire.

Act_Put(param, etat, etat_p, result) : remplace le $i^{ème}$ élément inséré dans la liste, par l'élément indiqué en paramètre, de telle manière qu'après l'exécution de insert(a), insert(b), put(1,d), la structure devienne : << <<>> <<b,d>> >>.

Pour cela, l'algorithme commence par rechercher quel est le bloc, la case, qui contient l'élément que l'on souhaite modifier. Une fois ce bloc identifié, une sous-fonction permet de remplacer l'élément en question dans le bloc en question.

Ainsi par exemple nous disposons de la liste << <<a>> <<b,c>> >>. Nous souhaitons modifier le 1^{er} élément ayant été inséré dans liste, ici représenté par le c, par la valeur d.

L'algorithme va donc identifier le bloc <<b,c>> comme étant celui qui contient la valeur à modifier. Le sous algorithme va prendre en charge ce bloc pour en modifier l'élément en question, le c, et le remplacer par le d.

Act_Get(param, etat, etat_p, result) : retourne le $i^{ème}$ élément ayant été inséré dans la liste, de telle manière qu'après l'exécution de insert(a) et insert(b), get(1) retourne la valeur a. L'implantation de l'algorithme est similaire au Put, on commence par identifier le bloc contenant l'élément, puis une sous-fonction renvoie l'élément demandé dans le bloc.

Act_Occurrence(param, etat, etat_p, result) : retourne le nombre d'occurrences de l'élément dans la liste.

3.1.3 Raffinement de ralR1 avec compteur binaire

L'invariant de liaison est défini par :

Liaison(etatA, etatC) == A!BinaryToInteger[etatA] = C!Cardinal(etatC)

A représentant le compteur_binaire et C ralR1.

Cet invariant vérifie que le nombre entier représenté par les deux structures est identique.

Ce raffinement fonctionne.

3.1.4 Raffinement de ralR1 avec Liste

L'invariant de liaison est définit par :

Liaison(etatA, etatC) ==

/\ A!Cardinal(etatA) = C!Cardinal(etatC)

/\ \A d \in D : \A o \in 0..C!Cardinal(etatC) :

A!Act_Occurrence(d, etatA, etatA, o) = C!Act_Occurrence(d, etatC, etatC, o)

A représentant la liste et C ralR1.

On vérifie ici que les deux structures possèdent le même nombre d'éléments et les mêmes éléments.

Cependant, ce raffinement ne fonctionne pas. Effectivement, nous avons implanté ralR1 de la manière la plus naturelle qui soit pour l'utilisateur.

Ainsi si celui-ci effectue insert(a,liste), insert(b,liste), il s'attend, a priori, à ce que le premier élément inséré, c'est-à-dire le a, possède l'indice 1, et le second élément inséré, le b, possède l'indice 2. Ainsi get(1,liste) retourne a, et get(2,liste) retourne b.

Or, le module Liste se comporte d'une manière qui ne parait pas naturelle pour l'utilisateur, le dernier élément inséré dans la liste est effectivement considéré comme se trouvant à l'indice 1, et le premier élément inséré se trouve à l'indice N où N représente le

nombre d'éléments de la liste. L'indice du premier élément inséré augmente ainsi à chaque insertion d'un nouvel élément.

Après avoir remarqué ce comportement, nous nous apprêtions à modifier nos algorithmes Put et Get, qui s'en seraient trouvés simplifiés, mais le raffinement échoue à cause d'une autre erreur que voici :

STATE 6: <Action line 17, col 1 to line 18, col 133 of module run_ralR1_liste>

/\ Result = "<NO_DATA>"

/\ Tour = "module"

/\ EtatA = <<"b", "a">>

/\ EtatC = << << >>, << "b", "a" >>, << >> >>

/\ Choix = "Put"

/\ Param = [indice |-> 1, valeur |-> "a"]

/\ RaffOk = TRUE

"Condition de raffinement des actions du module invalide !" TRUE

STATE 7: <Action line 17, col 1 to line 18, col 133 of module run_ralR1_liste>

/\ Result = "__NO_DATA"

/\ Tour = "client"

/\ EtatA = "<NO_DATA>"

/\ EtatC = << << >>, << "b", "a" >>, << >> >>

/\ Choix = "<NO_DATA>"

/\ Param = "<NO_DATA>"

/\ RaffOk = FALSE

Les précédentes étapes ont consisté à insérer a et b.

L'étape 6 cherche à remplacer l'élément d'indice 1 par a. Comme indiqué précédemment dans notre structure l'indice 1 est ici représenté par a dans EtatC, alors qu'il est représenté par b dans EtatA.

En tout logique et en prenant cette petite différence en considération, nous devrions donc obtenir :

/\ EtatA = <<"a", "a">>

/\ EtatC = << << >>, << "b", "a" >>, << >> >>

Or nous obtenons

/\ EtatA = "<NO_DATA>"

/\ EtatC = << << >>, << "b", "a" >>, << >> >>

L'EtatC obtenu, correspondant à ralR1 correspond bien à ce qui était attendu, mais EtatA correspondant à Liste devient "<NO_DATA>" après l'opération. Etant donné que l'invariant de liaison compare la cardinalité des deux structures, cette comparaison échoue.

Nous ne sommes pas parvenus à identifier l'origine du problème, après vérification notre algorithme semble correct, mais celui de liste également. Etant donné ce problème, nous n'avons pas modifié Put et Get, afin de nous concentrer sur la résolution de ce dernier.

3.2 ralR2

Le module ralR2 est un raffinement de ralR1 qui doit modifier la structure de compteur binaire utilisée jusqu'alors.

C'est notamment ici que la recherche bibliographique intervient : pour cette implantation, nous avons choisi d'utiliser le skew number system.

Basiquement, les fonctions principales à modifier par rapport à ralR1 sont le cardinal, l'incrémentation et la décrémentation.

Voici leur pseudo-code, qui est assez concis. Dans ce pseudo code, pour simplifier, le nombre est une liste de poids de bits.

```
Fonction Cardinal(nombre : liste de poids)
Début
     Si nombre = liste_vide alors Retourner 0
     Sinon Retourner Tête(nombre) + Cardinal(Queue(nombre))
Fin

Fonction Incrémenter(nombre : liste de poids)
Début
     Match (nombre) with
     | (poidsBit1 :: poidsBit2 :: reste) ->
          Si poidsBit1 = poidsBit2 alors
                    Retourner (1+poidsBit1+poidsBit2) :: reste
               Sinon
                    Retourner 1 :: nombre
     | _ -> Retourner 1 :: nombre
Fin
```

Le même poids répété deux fois indique un 2.

```
Fonction Décrémenter(nombre : liste de poids)
Début
     Match (nombre) with
     | (1 :: reste) ->
          Retourner reste
     | (poidsBit1 :: reste) ->
          Retourner (poidsBit1/2) :: (poidsBit1/2) :: reste
Fin
```

Ces fonctions impliquent qu'on compte comme ceci :

[1], [1,1], [3], [1,3], [1,1,3], [3,3], [7], [1,7], [1,1,7], [3,7], [1,3,7], [1,1,3,7], [3,3,7], etc.

Il suffit d'appliquer la fonction Cardinal pour retrouver la valeur du nombre. Par exemple,

[1,1,3] = 1 + 1 + 3 = 5.

Ces fonctions marchent pour le module ralR2. Cependant, on n'est pas arrivé pas à valider le raffinage par ralR2 du module compteur et/ou ralR1, car on s'est rendu compte que la structure pour stocker un nombre évolue différemment par rapport à ralR1.

En effet, dans ralR1, la liste de nombre a une taille fixe de L (si le nombre est 0, alors il y aura L éléments vides dans la liste). Dans ralR2, on voit que la taille de la liste est variable.

Par conséquent, il nous faut aussi adapter les fonctions Put et Get associées à ralR1, et non pas simplement les copier/coller.

Nous n'avons pas pu le faire par manque de temps.

3.3 ralR3

Grâce au skew number system, notre Random Access List ralR2 a une complexité en O(1) pour les opérations insert/incrémentation et remove/decrémentation.

Mais rechercher un élément d'indice n a une complexité O(n) car on doit possiblement tout parcourir.

L'idée de ralR3 est donc de raffiner la structure des éléments des cases, de manière à réduire cette complexité. Une bonne solution est, au lieu d'avoir une liste dans une case, d'avoir un arbre binaire de recherche dans une case. On réduit ainsi la complexité en passant en O(log n).

Nous n'avons pas pu faire ralR3 par manque de temps.

4. Conclusion

L'accumulation des projets de fin d'année ne nous a pas permis de consacrer suffisamment de temps à la réalisation de ce projet, même si la prolongation d'une semaine nous a été bénéfique.
Nous avons aussi mis du temps pour nous lancer efficacement dans le codage TLA. On aurait aimé avoir quelques séances de TP supplémentaires de suivi, comme en J2EE ou en Intergiciels, pour résoudre plus rapidement certaines difficultés.
Par conséquent, nous n'avons pas pu finir le test de raffinement de ralR2 et faire ralR3.

Cependant, ce projet nous a incités à bien nous organiser. En effet, pour gérer les plusieurs projets en même temps, il a fallu travailler en équipe efficacement et bien se répartir les tâches.

De plus, ce projet nous a permis de mieux comprendre le fonctionnement du langage TLA.