Devoir en temps limité n°5 – 4h

Calculatrices interdites

1 Un peu de hachage

1. *Qu'est-ce qu'une fonction de hachage ?*
   Un fonction de hachage est une fonction qui va de l'ensemble des clés dans $[|0, m-1|]$.

2. *Écrire une fonction* `int recherche(couple* table, char* cle)` *qui recherche la cle dans la table et renvoie la valeur associée.*

```c
int recherche(couple* table, char* cle){
  int hache = h1(cle);
  if (strcmp(table[hache].cle,cle)){
      return table[hache].valeur;
  }

  //Si on arrive ici, la clé n'est pas stockée dans la table
  assert(false);
}
```

3. *En supposant que la fonction de hachage s'effectue en temps constant, quelle est la complexité de votre fonction* `recherche` *?*
   On fait un appel à h_1 plus 1 comparaison et 2 accès à une case de tableau et 2 accès à des champs.
   Conclusion, la fonction a une complexité constante.

2 Logique

Exercice tiré et adapté du sujet CCINP option informatique 2017

4. *Rappeler la définition de l'ensemble des formules propositionnelles.*
   C'est une définition inductive, avec $\mathcal{B} = \{\top, \bot\} \cup \mathcal{V}$ avec $\mathcal{V}$ l'ensemble des variables et $C = \{\neg, \wedge, \vee, \implies, \iff\}$. (remarque : une définition sans l'implication, l'équivalence ou top et bottom est aussi acceptable).
5. *Représenter le comportement manichéen des interlocuteurs dans le premier sujet abordé sous la forme d'une formule ϕ du calcul des propositions dépendant des formules propositionnelles X_1 et Z_1 .*
   Soit X_1 et Z_1 sont sémantiquement vraies, soient elles sont toutes deux sémantiquement fausses. La formule suivante est donc toujours vraies dès lors que la règle manichéenne est respectée :
   $\phi = (X_1 \wedge Z_1) \vee (\neg X_1 \wedge \neg Z_1)$
6. *Représenter les informations données par les participants sous la forme de deux formules du calcul des propositions X_1 et Z_1 dépendant des variables V et C.*
   $X_1 = V \vee C$ et $Z_1 = \neg V$
7. *En utilisant des équivalents sémantiques, simplifier ϕ et déterminer dans quelle région vous devez vous rendre pour rejoindre le village.*
   On compbine les formules obtenues dans les deux questions précédentes.

$$\begin{aligned}\phi &= ((V \vee C) \wedge \neg V) \vee (\neg(V \vee C) \wedge \neg\neg V) \\ &\equiv ((V \wedge \neg V) \vee (C \wedge \neg V)) \vee (\neg V \wedge \neg C \wedge V) \\ &\equiv (\bot \vee (C \wedge \neg V)) \vee \bot \\ &\equiv C \wedge \neg V\end{aligned}$$

   En conclusion, la seule valuation satisfaisant ϕ est $V \mapsto F, C \mapsto V$: le village est dans les collines et pas dans la vallée.
8. *Représenter le comportement manichéen des interlocuteurs dans le second sujet abordé sous la forme d'une formule du calcul des propositions ψ dépendant des formules propositionnelles X_2 , Y_2 et Z_2 .*
   Même principe avec 3 formules.
   $\psi = (X_2 \wedge Y_2 \wedge Z_2) \vee (\neg X_2 \wedge \neg Y_2 \wedge \neg Z_2)$

9. *Représenter les informations données par les participants sous la forme de trois formules du calcul des propositions X_2 , Y_2 et Z_2 dépendant des variables G, M et D.*
   $X_2 = G \wedge D$ et $Y_2 = M \implies \neg D$ et $Z_2 = G \wedge \neg M$
10. *En faisant une table de vérité, déterminer quel chemin vous devez suivre pour rejoindre le village.*

G	M	D	X_2	Y_2	Z_2	ψ
V	V	V	V	F	F	F
V	V	F	F	V	F	F
V	F	V	V	V	V	V
V	F	F	F	V	V	F
F	V	V	F	F	F	V
F	V	F	F	V	F	F
F	F	V	F	V	F	F
F	F	F	F	V	F	F

    En conclusion on a deux valuations qui fonctionnent : $v_1 : G \mapsto V, M \mapsto F, D \mapsto V$ et $v_2 : G \mapsto F, M \mapsto V, D \mapsto V$. La seule chose dont on peut être sûr est que le chemin de droite mène au village.
11. *En admettant que les trois participants aient menti, pouviez-vous prendre d'autres chemins ? Si oui, le ou lesquels ?*
    Dans la ligne correspondant à la valuation v_2, les trois participants ont menti, on aurait donc pu prendre le chemin du milieu et de droite.

3 Représentation classiques d'ensembles

Exercice tiré et adapté du sujet Centrale option informatique 2023

1. Avec une liste triée (Ocaml)

12. *Dans cette question **uniquement**, on implémente un ensemble d'entiers positifs par la liste de ses éléments, rangés **dans l'ordre croissant**. Écrire une fonction `succ_list : int list -> int -> int` prenant en arguments une liste d'entiers distincts dans l'ordre croissant et un entier x et renvoyant le successeur de x dans la liste, c'est-à-dire le plus petit entier strictement supérieur à x de la liste (ou -1 si cela n'existe pas).*

```
let rec succ_list l x = match l with
 |[] -> -1
 |t::q when t<= x -> succ_list q x
 |t::q -> t;;
```

13. *Donner sa complexité dans le pire cas.*
    Dans le pire des cas, x est plus grand que le plus grand élément et on parcourt tout le tas avant de s'en rendre compte. La complexité est donc linéaire en la taille de la liste dans le pire cas.

2. Avec un tableau trié (C)

Soit N un entier naturel strictement positif, fixé pour toute cette partie et qui est une variable globale. On choisit de représenter un ensemble d'entiers E de cardinal $n < N$ par un tableau t de taille $N + 1$ dont la case d'indice 0 indique le nombre n d'éléments de E et les cases d'indices 1 à n contiennent les éléments de E rangés dans l'ordre croissant, les autres cases étant non significatives.

Par exemple, le tableau `[3,2,5,7,9,1,14]` (suggestion de présentation) représente l'ensemble à 3 éléments $\{2, 5, 7\}$.

14. *Pour une telle implémentation d'un ensemble E, décrire brièvement des méthodes permettant de réaliser chacune des opérations ci-dessous (on ne demande pas d'écrire des programmes) et donner leurs complexités dans le pire cas :*
    - *déterminer le maximum de E ;* Il est dans la case n du tableau. On récupère la valeur de n dans la case 0. $O(1)$
    - *tester l'appartenance d'un élément x à E ;* On le compare avec les éléments entre 1 à n. Dans le pire cas, $O(n)$.
    - *ajouter un élément x dans E (on suppose la taille du tableau suffisante et que x n'appartient pas à E).* On écrit $n + 1$ dans la case 0. On trouve l'indice i où insérer x et on décale tous les éléments de i à n et enfin on met x dans la case i. $O(n)$

15. *Par une méthode dichotomique, écrire une fonction* `int succ_tab(int* t, int x)` *prenant en arguments un tableau* `t` *codant un ensemble E comme ci-dessus et un entier* `x` *et renvoyant le successeur de* `x` *dans E (-1 si cela n'existe pas).*
Il y avait des points pour écrire une dichotomie classique mais la vraie version nécessite un peu d'astuce pour gérer le cas où il faut renvoyer -1.

```
int succ_tab(int* t, int x){
    int n = t[0];
    int g = 1;
    int d = n+1;

    while(g<d){
      int milieu = (g+d)/2;
      else if (t[milieu]<=x){g=milieu+1;}
      else{d = milieu;}
    }

    if(d<=n){return tab[d];}
    else{return -1;}
}
```

16. *Calculer la complexité dans le pire cas de la fonction* `succ_tab` *en fonction de n.*
C'est une dichotomie, donc c'est en $O(log_2(n))$.
En effet, à chaque tour de boucle, on diminue la taille du sous-tableau dans lequel la recherche s'effectue (entre g et d-1) d'un facteur 2. Le reste des opérations est élémentaire.
17. *Écrire une fonction* `int* union_tab(int* t_1, int* t_2)` *prenant en arguments deux tableaux* `t_1` *et* `t_2`*, de taille N, codant deux ensembles E_1 et E_2 et renvoyant le tableau correspondant à $E_1 \cup E_2$. On supposera que $|E_1 \cup E_2| \leq N$.*

```
int* union_tab(int* t_1, int* t_2){
  //Nouveau tableau
  int* t = malloc(N*sizeof(int));

  int n1 = t_1[0];
  int n2 = t_2[0];
  t[0] = n1+n2;

  //Ensuite on fusionne les deux tableaux comme dans le tri fusion, pour garantir l'ordre

  //indice du premier élément non ajouté dans t_1 et t_2.
  int i1 = 1;
  int i2 = 1;
  for(int i=1; i<=n1+n2;i+=1){
    if(t_2[i2]==n2+1 || t_1[i1]<t_2[i2]){
      t[i] = t_1[i1];
      i1+=1;
    }
    else{
      t[i] = t_2[i2];
      i2+=1;
    }
    i+=1;
  }

  return t;
}
```

3. Avec des arbres binaires de recherche (Ocaml)

```
type abr = Vide | Noeud of abr * int * abr;;
```

18. *Rappeler la propriété d'ordonnancement existant sur les étiquettes d'un ABR, sachant qu'on considèrera dans la suite qu'une étiquette ne peut apparaitre plusieurs fois dans l'ABR.*
L'étiquette d'un noeud est plus grande à celle de son fils gauche et plus petite à celle de son fils droit.
19. *Dessiner un ABR pour l'ensemble de valeurs* $\{2, 3, 5, 8, 13\}$.
Plusieurs possibilités.

20. *Écrire une fonction* `min_abr : abr -> int` *prenant en argument un ABR représentant un ensemble E et renvoyant son étiquette minimale (-1 si l'ensemble est vide).*

```
let rec min_abr a = match a with
  |Vide -> -1
  |Noeud(Vide,x,_) -> x
  |Noeud(g,_,_) -> min_abr g;;
```

21. *Écrire une fonction récursive* `partitionne_abr : abr -> int -> (bool * abr * abr)` *prenant en arguments un arbre binaire de recherche représentant un ensemble E et un entier* `x` *et renvoyant un triplet* `(b, ag, ad)` *où* `b` *vaut* **`true`** *si* `x` *appartient à E et* **`false`** *sinon,* `ag` *est un arbre binaire de recherche codant les éléments de E strictement plus petits que* `x` *et* `ad` *un arbre binaire de recherche codant les éléments de E strictement plus grands que* `x`*.*

```
let rec partitionne_abr a x = match a with
  |Vide -> (false, Vide, Vide)
  |Noeud(g,r,d) when r=x -> (true, g,d)
  |Noeud(g,r,d) when r<x -> let b,dg,dd = partitionne_abr d x in
                            (b,Noeud(g,r,dg),dd)
  |Noeud(g,r,d) when r>x -> let b,gg,gd = partitionne_abr g x in
                            (b,gg,Noeud(gd,r,d));;
```

22. *Écrire une fonction* `insere_racine_abr : abr -> int -> abr` *prenant en arguments un arbre binaire de recherche représentant un ensemble E et un entier* `x` *et renvoyant un arbre binaire de recherche associé à l'ensemble $E \cup \{x\}$ et de racine étiquetée par* `x`*. Calculer sa complexité dans le pire cas en fonction de l'arbre reçu puis en fonction de E.*

```
let insere_racine_abr a x =
   let b, g, d = partitionne_abr abr x in
   Noeud (g, x, d);;
```

Lors de l'union, si x est déjà dans E, alors $E \cup \{x\} = E$. La partition retire x, donc on a à la fin de toute façon qu'une seule occurence de x.

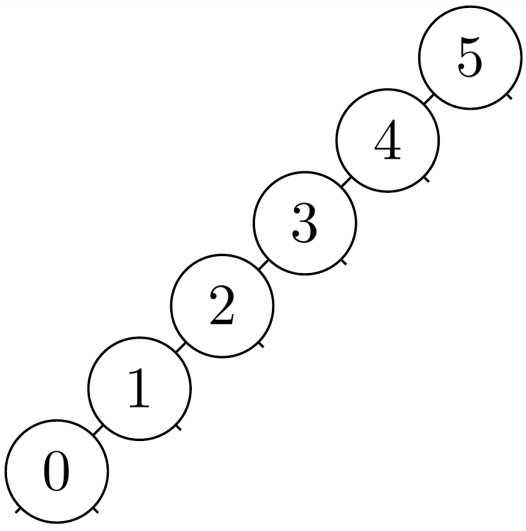
La complexité de cette fonction est la même que celle de `partitionne_abr`.

Dans le pire cas, la fonction `partitionne_abr` suit tout un chemin de la racine jusqu'à un vide. En ne faisant que des opérations élémentaires à chaque appels. On fait alors autant d'appels que la hauteur de l'arbre, donc la fonction est linéaire en la hauteur de l'arbre. (rq : répond à "Calculer sa complexité dans le pire cas en fonction de l'arbre")

Pour estimer la complexité en fonction de E, il faudrait estimer la hauteur de l'arbre en fonction de E, plus précisément en fonction du cardinal de E.

L'énoncé ne précise rien sur l'équilibrage des ABR ou la manière de les créer, et on sait que les ABR peuvent à priori prendre toutes les formes que les arbres binaires peuvent prendre.

Un indice plus précis est la fonction qu'on est en train d'écrire : si cette fonction est celle qui sert à créer les ABR, en ajoutant un par un les éléments de l'ensemble à un arbre initialement vide, alors préserver l'équilibrage est à priori impossible. Par exemple pour l'ensemble $\{0, 1, 2, 3, 4, 5\}$, dont on ajoute les éléments dans l'ordre croissant, on obtiendrait l'arbre suivant :



Cet arbre a une hauteur linéaire en $|E|$. En conclusion notre fonction est linéaire en le cardinal de E.

23. *Écrire une fonction* `union_abr : abr -> abr -> abr` *prenant en arguments deux arbres binaires de recherche représentant deux ensembles E_1 et E_2 et renvoyant un arbre binaire de recherche associé à l'ensemble $E_1 \cup E_2$. On expliquera brièvement la méthode choisie.*

```
let union_abr a1 a2 = match a1 with
   | Nil -> a2
   | Noeud (g1, v1, d1) -> let _, g2, d2 = partitionne_abr a2 v1 in
       Noeud (union_abr g1 g2, v1, union_abr g1 g2)
```

On prend un noeud dans a_1 de valeur v_1. Les éléments de g_1 sont strictement inférieurs à v_1 et les éléments de d_1 sont strictement supérieurs. On partitionne a_2 de la même manière en g_2 et d_2.
Dans $E_1 \cup E_2$, les éléments strictement inférieurs à v_1 sont ceux de g_1 et g_2, dont on fait l'union et les éléments strictement supérieurs sont ceux de d_1 et d_2, dont on fait l'union.
On obtient la formule finale.

4 Représentation par des arbres binaires complets (C)

On considère dans cette partie des arbres binaires complets dont les nœuds sont étiquetés par des booléens. Les nœuds seront numérotés à partir de la racine qui porte le numéro 1 dans l'ordre d'un parcours en largeur (de gauche à droite à chaque profondeur). La Figure 1 donne un exemple de cette numérotation.

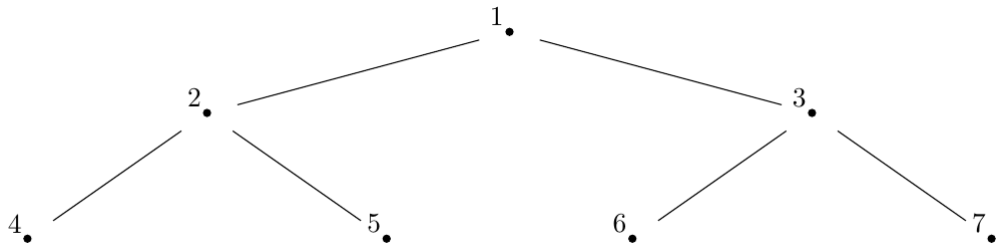


FIGURE 1 – Numérotation des noeuds

24. *Dans un arbre binaire complet de hauteur $p \in \mathbb{N}$, quel numéro peut avoir un noeaud à la profondeur $k \in [|0, p|]$? Combien le sous-arbre dont la racine est i a-t'il de feuilles ? Justifier.*
Il y a 2^k noeuds à la profondeur k. Ainsi, un noeud à la profondeur k a un numéro compris entre

$$1 + \sum_{i=0}^{k-1} 2^i = 1 + 2^k - 1 = 2^k$$

et

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

Le noeud de numéro i est donc à une profondeur $\lfloor \log_2 i \rfloor$, et le sous-arbre de racine numéro i a donc $2^{p-\lfloor \log_2 i \rfloor}$ feuilles.

25. *Dans un arbre binaire complet de hauteur $p \in \mathbb{N}$, pour le nœud numéro i, donner, en les justifiant, les numéros de son fils gauche, de son fils droit et de son père.*
Le nœud de numéro i est à une profondeur $\lfloor \log_2 i \rfloor$ et à cette profondeur, il y a $i - 2^{\lfloor \log_2 i \rfloor}$ noeuds à sa gauche. Son fils gauche est alors à une profondeur $\lfloor \log_2 i \rfloor + 1$ et il y a $2 * (i - 2^{\lfloor \log_2 i \rfloor})$ noeuds à sa gauche. Son numéro est donc $i_g = 2i$, et le fils droit du noeud i est $2i + 1$. De même, son père a pour numéro $\lfloor \frac{i}{2} \rfloor$.

26. *Écrire une fonction `bool appartient(bool* ens, int taille, int x)` qui détermine si un entier `x` quelconque appartient ou non à un ensemble `ens` donné de taille `taille`. Calculer sa complexité.*
Comme indiqué dans l'énoncé, les feuilles ont les numéros partant de 2^p qui n'est autre que la moitié de la taille du tableau.

```
bool appartient(bool* ens, int taille, int x){
 return ens[taille/2+x];
}
```

Complexité en $O(1)$.

27. *Écrire une fonction `bool* fabrique(int* t, int n, int ppuis2)` qui prend en arguments un tableau d'entiers positifs distincts t de taille n et une valeur pour 2^p et renvoie l'arbre associé à l'ensemble E dont les éléments sont ceux du tableau. On supposera que tous les éléments de t appartiennent à $[|0, 2^p - 1|]$. Cette fonction devra s'exécuter en $O(2^p)$.*

```
bool* fabrique(int* t, int n, int ppuis2){
  bool* ens = malloc((ppuis2*2)*sizeof(bool));

  //Remplir les feuilles
  for (int i=0;i<n;i+=1){
    ens[t[i]+ppuis2] = true;
  }

  //Remplir les autres noeuds
  for (int i=ppuis2-1;i>=1;i-=1){
    ens[i] = ens[2*i] || ens[2*i+1];
  }
  return ens;
}
```

28. *Écrire une fonction* `void insere(bool* ens, int taille, int k)` *qui ajoute un entier k à un ensemble E. On suppose k compatible avec la valeur de p associée à E. Cette fonction devra s'exécuter en $O(1)$ dans le meilleur cas.*

    On part de l'indice correspondant à la valeur à ajouter et on mets à true la case d'indice correspondant ainsi que ses parents. Pour avoir une complexité en $O(1)$ dans le meilleur des cas, il faut arrêter la boucle dès que possible.

```
void insere(bool* ens, int taille, int k){
 int ind = taille/2+k;
 while (ind > 0 && !ens[ind]){
   ens[ind]= true;
   ind = ind/2;
 }
}
```

29. *Écrire une fonction* `void supprime(bool* ens, int taille, int k)` *qui retire un entier k d'un ensemble E. On suppose k compatible avec la valeur de p associée à E. Calculer la complexité de cette fonction dans le pire cas.*

```
void supprime(bool* ens, int taille, int k){
   int ind = taille/2+k;

   //Mise à jour de la feuille
   ens[ind] = false;
   ind = ind/2;

   //Mise à jour de tous les ancetres, leur valeur dépend de leur autre enfant
   while(ind > 0){
     ens[ind] = ens[2*ind] || ens[2*ind +1];
     ind = ind/2;
   }
}
```

    Cet algorithme parcourt un chemin d'une feuille à la racine. Il est linéaire en la hauteur, donc logarithmique en le nombre de noeuds.

30. *Écrire une fonction* `int minlocal(bool* ens, int taille, int i)` *qui cherche l'élément de E minimal parmi ceux codés dans le sous-arbre de racine numérotée i dans l'arbre associé à E. Si un tel élément n'existe pas, cette fonction devra renvoyer -1. Calculer la complexité de cette fonction en fonction de p et i.*

    On cherche la feuille valant true la plus à gauche. Pour chaque noeud, si c'est une feuille on a trouvé, sinon on étudie son fils de gauche si celui-ci vaut true et son fils droit sinon.

    Avec l'ajout de la valeur i, l'énoncé pointe clairement vers une résolution récursive.

```
int minlocal(bool* ens, int taille, int i){
 if (i >= taille/2 ){ //Si on est sur une feuille
   if (ens[i]==true){return i-taille/2;}
   else{return -1;}
 }
 else{ //Si on est sur un noeud
   if (ens[2*i]==true){return minlocal(ens, taille, 2*i);}
   else{return minlocal(ens,taille, 2*i+1);}
 }
}
```

    À nouveau, notre algorithme parcourt un chemin du noeud i original (probablement la racine) à une feuille en suivant les true. La complexité est linéaire en p. Plus précisément, elle est linéaire en p moins la profondeur du noeud i initial.

31. *Prouver l'algorithme décrit ci-dessus.*
On remarque tout d'abord que le successeur de x dans l'arbre parfait est la première feuille qui vaut vrai à droite de la feuille représentant x.
S'il existe, la feuille du successeur et la feuille représentant x ont donc un unique ancêtre commun de profondeur maximale, qui est ce que cherche cet algorithme.

Pour montrer la terminaison, on considère comme variant la profondeur du noeud i, qui est un entier positif strictement décroissant dans la boucle car le i suivant est le père du i précédent.

On considère la propriété suivante :
"Toutes les feuilles du sous-arbre enraciné en i qui sont à droite de la feuille représentant x valent `false`".

Montrons qu'il s'agit d'un invariant
À l'initialisation, i représente la feuille x. la propriété est donc vraie. (x n'est pas à droite de x)
Supposons la propriété vraie jusqu'à la fin d'une certaine itération et considérons la suivante. On notera i la valeur de la variable `i` à l'itération précédente et i' sa valeur à l'itération actuelle.
Au vu de la condition du `while`, on sait que la case $i+1$ vaut `false` et que i n'est pas le plus à droite de sa rangée.
On prend alors i' numéro du père de i. Deux choix s'offrent à nous

- i est fils droit de i'. Ainsi toutes les feuilles dans le sous-arbre enraciné dans i' sont soit à gauche de la feuille x, soit des feuilles de l'arbre enraciné en i. En conclusion on a pas pu gagner de feuilles qui valent `true` et qui sont à droite de x. La propriété tient pour i'.
- i est fils gauche de i'. Son frère est $i+1$, dont la valeur est `false`. L'arbre enraciné en $i+1$ ne contient donc aucun `true` et l'arbre enraciné en i' ne contient aucun `true` à droite de x. La propriété tient pour i'.

Notre propriété est donc un invariant.
À la fin de la boucle, on a deux possibilités :

- Si i est le plus à droite de sa rangée, alors on sait que toutes les feuilles à droite de x sont dans l'arbre enraciné en i et d'après l'invariant, valent `false`. En conclusion il n'y a pas de successeur pour x et l'algorithme est correct car il renvoie -1.
- si la case $i+1$ vaut `true`, alors il existe au moins une feuille dans l'arbre enraciné en $i+1$ qui vaut `true`. En particulier on note m la feuille minimale valant `true` de l'arbre enraciné en $i+1$. Cette feuille est à droite de x et toutes les feuilles entre les deux sont soit dans l'arbre enraciné en i et valent `false` par l'invariant, soit dans l'arbre enraciné en $i+1$ et valent `false` par minimalité de m. L'algorithme est correct car il renvoie m qui est bien le successeur de x.

En conclusion, l'algorithme est correct dans tous les cas d'arrête de la boucle `while`.

32. *Écrire une fonction `int successeur(bool* ens, int taille, int x)` prenant en arguments l'ensemble E et un entier x positif et renvoyant son successeur dans E (-1 si cela n'existe pas).*
Pour savoir si i est le plus à droite, on garde en mémoire le numéro du dernier noeud de la ligne actuelle (évolue en divisant par 2 car ces noeuds sont les fils les uns des autres)

```
int successeur(bool* ens, int taille, int x){
  int i = taille/2+x;
  int plusadroite = taille-1;
  while(i!=plusadroite && ens[i+1]==false){
    i = i/2;
    plusadroite = plusadroite/2
  }

  if(i==plusadroite){return -1;}
  else{return minlocal(ens, taille, i+1);}
}
```

33. *Montrer que si $x \in E$ admet bien un successeur dans E, il existe une constante $K > 0$ indépendante de E et p telle que la complexité de `successeur(ens, taille, x)` soit majorée par $K(log_2(\texttt{successeur(ens, taille,x)}-x)+2)$. On admet que le même type de justification montre que si x est le maximum de E, la complexité de `successeur(ens, taille, x)` est majorée par $K(log_2(2^p - x) + 2)$.*
Pour trouver le successeur y de x , il faut remonter à la profondeur $p_m = p - \lfloor \log_2(y-x) \rfloor$ où l'on va calculer le minimum local. La recherche de successeur va implique $p - p_m + 1$ tours de la boucle `while`, qui ne contient que des opérations élémentaires suivi d'un appel à `minlocal` à partir de la profondeur p_m , pour une complexité en

$$O((p - p_m + 1) + (p - p_m + 1)) = O(2\log_2(successeur(x) - x) + 2)$$

34. *En utilisant la fonction `successeur`, écrire une fonction `int cardinal(bool* ens, int taille)` prenant en argument un ensemble et renvoyant son cardinal.*

```
int cardinal(bool* ens, int taille){
  //On trouve le premier élément
  int v = minlocal(ens,taille,1);
  int res=0;

  //Ensuite on applique successeur jusqu'à ce qu'on en puisse plus
  while(v!=-1){
    res+=1;
    v = successeur(ens, taille, v);
  }

  return res;
}
```

35. *Déterminer la complexité de la fonction `cardinal` en fonction de p et $n = |E|$. On rappelle que la fonction log_2 est concave.*
En notant $(x_1, \dots, x_n)$ les éléments de E , la complexité de `cardinal` se décompose en $O(p+1)$ pour déterminer x_1 à l'aide de `minlocal`, suivi de $n-1$ appels à successeur, pour une complexité totale de

$$O(p+1+\sum_{i=1}^{n-1}(2\log_2(x_{i+1}-x_i)+2)$$

Par concavité du logarithme, on a

$$\sum_{i=1}^{n-1}\log_2(x_{i+1}-x_i) \leq (n-1)\log_2(\frac{x_n-x_1}{n-1}) \leq (n-1)\log_2(x_n-x_1)$$

Comme $x_n - x_1 \leq 2^p$, on a finalement une complexité pour `cardinal` en $O(np)$.

36. *Quels sont les intérêts et inconvénients d'une telle structure ? Dans quels cas peut-elle s'avérer plus intéressante que des structures connues ?*
La taille de cette structure de donnée est linéaire en la largeur de l'intervalle dont on veut représenter une partie, et permet d'avoir les fonctions de base (test d'appartenance, ajout, suppression et même successeur) en complexité logarithmique en fonction de cette largeur.
Il en est de même pour, par exemple, un simple tableau de booléens indiquant ou non l'entier correspondant à son indice. Pour un tel tableau, l'appartenance, l'ajout et la suppression se font en temps constant, mais la recherche du sucesseur se fait en temps linéaire (en la taille de l'intervalle) dans le pire des cas, ce qui est moins bien. Par contre, le calcul du cardinal se fait lui aussi en temps linéaire, ce qui est plus efficace.
Comparé à une structure type arbre binaire comme vue précédemment, la taille de la structure n'est pas linéaire en le nombre d'éléments contenus, mais les complexités logarithmiques pour les opérations de base nécessitent d'équilibrer l'arbre puisqu'elles dépendent de la hauteur.

5 Arbres de van Emde Boas (Ocaml)

Soit p un entier positif et $N = 2^{2^p}$. On supposera que tous les entiers manipulés restent représentables par la structure d'entier OCaml ordinaire. On considère le type de structure suivant (appelé arbre *veb* par la suite) implémentant un ensemble E d'entiers positifs strictement inférieurs à N :

```
type veb = {mutable mini : int; mutable maxi : int; table : veb array};;
```

Les champs mutables d'un arbre *veb* sont modifiables : par exemple, pour un arbre *veb* noté `v`, `v.mini <- 1` change la valeur du champ `v.mini`.

Le codage d'un ensemble $E \subset [|0, N-1|]$ par un arbre *veb* dit d'ordre $N = 2^{2^p}$ suit les règles suivantes :

- Les champs `mini` et `maxi` représentent toujours la valeur minimale et maximale de E. Ils sont mis arbitrairement à -1 si l'ensemble est vide.
- Si $N = 2$, i.e. si l'arbre doit coder une partie de $[|0, 1|]$, le champ table est un tableau vide (i.e. `[||]`), les champs `mini` et `maxi` suffisant à coder E. **On veillera à ce que les fonctions demandées traitent correctement ce cas particulier.**
- Pour $N > 2$, on note $\widehat{E} = E \setminus \{min(E)\}$ (où $min(E)$ désigne le minimum de E). On décompose $\widehat{E}$ en $\sqrt{N} = 2^{2^{p-1}}$ ensembles E_k définis par

$$\forall k \in [|0, \sqrt{N}-1|], E_k = \left\{x - k\sqrt{N} | x \in \widehat{E} \cap [|k\sqrt{N}, (k+1)\sqrt{N}-1|]\right\}$$

le champ `table` est alors un tableau de $\sqrt{N}+1$ arbres *veb* d'ordre $\sqrt{N}$:

- pour $k \in [|0, \sqrt{N}-1|]$, l'arbre *veb* stocké dans la case `table.(k)` code l'ensemble E_k ;
- l'arbre *veb* stocké dans la case `table.(`$\sqrt{N}$`)` code l'ensemble $R = \{k \in [|0, \sqrt{N}-1|] | E_k \neq 0\}$.

Par exemple, considérons l'arbre *veb* `ex` codant l'ensemble $E = \{2, 13, 14, 15\}$ avec $p = 2$ i.e. $N = 16$. On a $\widehat{E} = \{13, 14, 15\}$ puisque $min(E) = 2$ et donc les cinq cases du tableau `ex.table` correspondent respectivement aux ensembles

$$E_0 = \emptyset \quad E_1 = \emptyset \quad E_2 = \emptyset \quad E_3 = \{13 - 3\sqrt{16}, 14 - 3\sqrt{16}, 15 - 3\sqrt{16}\} = \{1, 2, 3\} \quad R = \{3\}$$

On obtient la structure représentée en Figure 3.

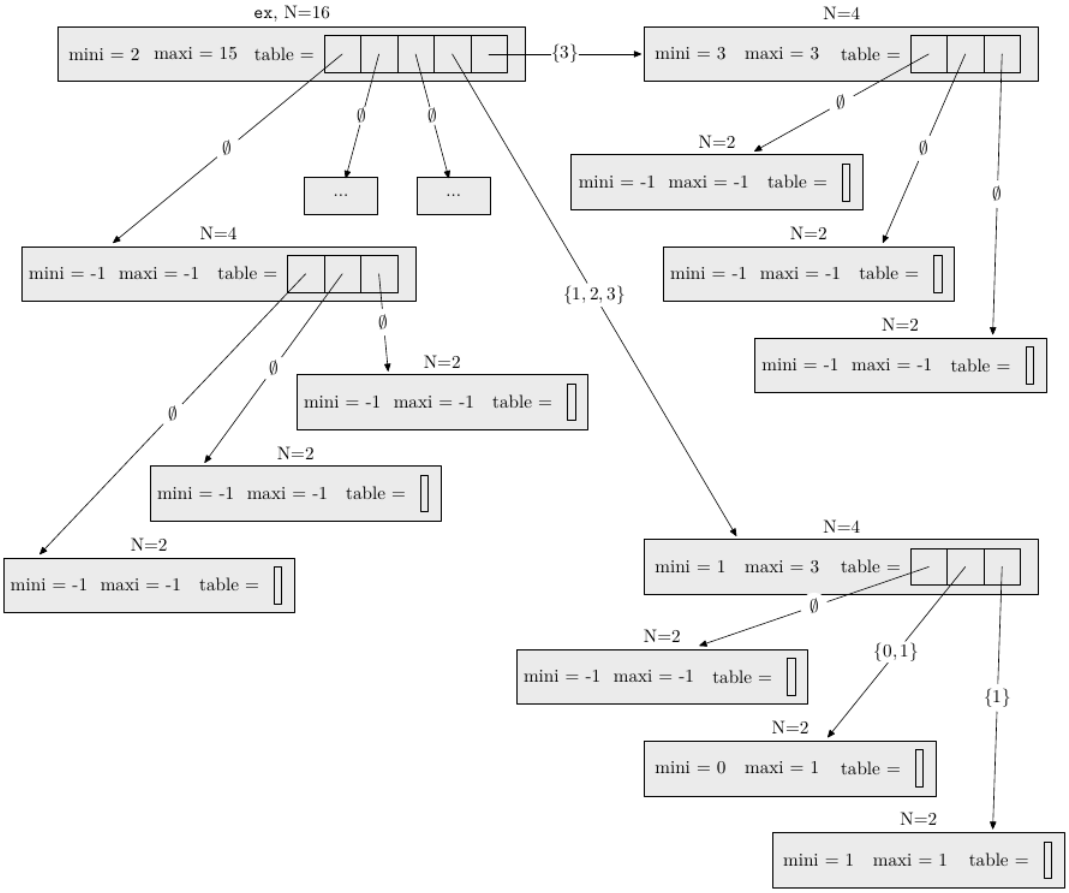


FIGURE 2 – L'arbre *veb* **ex** associé à l'ensemble $\{2, 13, 14, 15\}$ avec $p = 2$

37. *On veut coder l'ensemble* $\{0, 2, 3, 5, 7, 13, 14\}$ *par un arbre veb d'ordre* $N = 16$*, noté* `ex_veb`*. Indiquer quel ensemble code chaque arbre veb stocké dans* `ex_veb.table` *puis préciser* `ex_veb.table.(3)`*.*
On a $E_0 = \{2, 3\}$, $E_1 = \{1, 3\}$, $E_2 = \emptyset$, $E_3 = \{1, 2\}$ et $R = \{0, 1, 3\}$.
`ex_veb.table.(3)` a `mini = 1`, `maxi = 2`, `table = [|t0;t1;t2|]` avec `t0 = {mini=-1;maxi=-1;table=[||]}` l'ensemble vide, `t1 = {mini=0;maxi=0;table=[||]}` l'ensemble représentant $\{2\}$ et `t0 = {mini=1;maxi=1;table=[||]}` l'ensemble représentant $\{1\}$.

38. *Écrire une fonction* `creer_veb : int -> veb` *prenant en argument un entier* p *et créant un arbre veb d'ordre* $N = 2^{2^p}$ *implémentant l'ensemble vide.*

```
let rec puiss2 n = if n = 0 then 1 else 2 * puiss2 (n - 1);; (*Fonction auxiliaire pour les puissances*)

let creer_veb p = (*Crée l'arbre vide pour p*)
   let nombrecases = puiss2(puiss2 (p-1))+1; (*nb de cases de table*)
   let res = {mini = -1; maxi = -1; table = Array.make nbcases {mini=0;maxi=0;table=[||]}} in

   (*On met un arbre vide pour p-1 dans chaque case du tableau*)
```

```
  for i = 0 to nbcases-1 do
    res.table.(i) <- creer_veb (p-1)
  done;

  res;;
```

39. *Résoudre en fonction de q la récurrence $C(q) = C(\sqrt{q}) + O(1)$ dans le cas où $q = 2^{2^s}$ et $C(1) = 1$.*

    On a $q = 2^{2^s}$, donc $C(2^{2^s}) = C(2^{2^{s-1}}) + O(1)$. On pose $D(s) = C(2^{2^s})$. La suite D vérifie la récurrence $D(s) = D(s-1) + O(1)$.
    Par analogie avec les suites arithmétiques, on en déduire $D(s) = O(s)$ et donc $C(q) = \log_2(\log_2(q))$.

40. *Écrire une fonction `appartient_veb : veb -> int -> bool` qui teste l'appartenance d'un entier x quelconque à un ensemble E codé par un arbre veb. Calculer sa complexité dans le pire cas.*

```
let rec appartient_veb veb x =
 let racineN = Array.length veb.table - 1 in
 if racineN=-1 then (veb.mini = x || veb.maxi=x) (*Si N=2, on cherche si x est le min ou le max *)
 else begin
   if x< veb.mini || x>veb.maxi then false (*Hors des bornes *)
   if x = veb.mini then true (* x est le minimum *)
   else let k = x/racineN in (* On cherche le Ek dans lequel x va se trouver *)
   let newx = x-k*racineN in (* On décale x comme la définition d'un veb l'indique *)
   appartient_veb veb.table.(k) newx (* On cherche x dans Ek *)
 end;;
```

    Dans cette fonction, toutes les opérations sont élémentaires, sauf les appels récursif. Le pire cas est donc le cas où on fait le plus d'appel récursifs.
    Ce cas peut arriver si le x recherché est bien dans l'ensemble mais n'est jamais le min des sous-veb avant d'avoir $N = 2$. Dans ce cas on fait autant d'appels que la hauteur de l'arbre veb, qui est p. En effet chaque niveau de profondeur du veb correspond à un N qui est 2^{2^s} avec $s \in [|0, p|]$.

41. *Écrire une fonction `successeur_veb : veb -> int -> int` qui calcule le successeur d'un entier x quelconque dans E (-1 s'il n'y en a pas). Calculer sa complexité dans le pire cas.*

    Si $N = 2$ alors le successeur est mini, maxi ou -1 selon comment x se compare à mini et maxi.
    Si l'ensemble est vide la réponse est immédiate.
    Si x est plus petit que le minimum de l'ensemble, alors mini est le successeur
    Dans tout autre cas, le successeur est dans un des E_k. On identifie le E_k où x se trouve si $x \in E$. Si E_k est vide ou le maximum de E_k est plus petit que x, alors on doit trouver $E_j \neq \emptyset$ avec $j > k$ le plus petit possible et récupérer son minimum.
    Si E_k n'est pas vide et x est plus petit que le maximum, alors on affine la recherche en cherchant le sucesseur dans E_k.
    Attention : les valeurs dans les E_k on subi une transformation ($-k\sqrt{N}$) qu'il faut inverser.

```
let rec successeur_veb veb x =
 let racineN = Array.length veb.table - 1 in
 if racineN = -1 then
     if x<veb.mini then veb.mini
     else if x<veb.maxi then veb.maxi
     else -1
 else if veb.mini = -1 then -1 (* pas de successeur dans l'ensemble vide*)
 else if veb.mini > x then veb.mini (* si mini est le sucesseur *)
 else let k = x/racineN in
      let newx = x-k*racineN in
      if veb.table.(k).min = -1 (* Ek est vide *) ||
         veb.table.(k).max <= newx (* Aucun élément de Ek plus grand que x *) then
                  let kk = successeur veb.table.(racineN) k in (* successeur de k dans R *)
                  if kk = -1 then -1 (* Pas de Ej non vide après Ek*)
                  else veb.table.(kk).min + (kk)*racineN
     else (successeur_veb veb.table.(k) newx) + k*racineN (* Le successeur est dans Ek,
     + k*racineN pour trouver sa valeur au N actuel *)
```

    Complexité dans le pire cas : À chaque appel récursif, on descend d'un niveau dans l'arbre. On s'arrête évidemment pour $N = 2$ (on peut s'arêter avant, mais on s'intéresse au pire cas). Chaque appel, à part un appel récursif, effectue uniquement des opérations élémentaires (comparaisons, opérations arithmétiques et accès aux champs d'une structure/cases d'un tableau)
    La hauteur de l'arbre est p, soit $\log_2(\log_2(N))$. En conclusion, la fonction est en $O(\log_2(\log_2(N)))$

42. *Écrire une fonction `insertion_veb : veb -> int -> unit` qui insère un entier x dans un arbre veb. On suppose que $x \in [|0, N-1|]$. Cette fonction devra avoir une complexité en $O(log_2(log_2(N)))$.*

```
let rec insertion_veb v x =
   (* Si l'arbre est vide *)
   if v.mini = -1 && v.maxi = -1 then begin
     v.mini <- x;
     v.maxi <- x
   end
   else if x < v.mini then begin
     (* La valeur à insérer est le nouveau minimum *)
     let mini = v.mini in
     v.mini <- x;
     (* Il faut ranger l'ancien minimum dans les Ek*)
     insertion_veb v mini
     end
   else begin
     if x > v.maxi then v.maxi <- x; (*Mise à jour du maximum*)
     let racineN = Array.length v.table - 1 in
     (*Si N!=2 alors on insere dans la bonne sous table*)
     if racineN >= 0 then begin
       let k = x/racineN in
       let newx = x-k*racineN in
       insertion_veb v.table.(k) newx;
       (* Enfin, on met à jour R si l'arbre était vide *)
       if table.(k).mini = -1 then
         insertion_veb v.table.(racineN) k
     end
   end;;
```

La complexité est linéaire en la hauteur de l'arbre qui est en $\log_2(\log_2(N))$.