COURS 421-b, COMPOSITION D'INFORMATIQUE

Philippe Jacquet, Luc Maranget

vendredi 25 janvier 2008

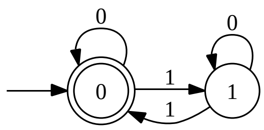
⋆⋆⋆

Partie I, transmission d'un texte binaire.

Un *texte* est défini comme une suite (finie) de 0 et 1. Les entiers 0 et 1 sont les chiffres d'une représentation des entiers en base deux, par la suite on les appelle « *bits* ».

Question 1. Donner un automate fini déterministe (DFA) qui décrit le langage des textes où le bit 1 apparaît un nombre pair de fois.

Réponse : L'automate :



On trouve assez facilement cet automate fini (déterministe) en considérant deux états : S_0 qui signifie « j'ai vu un nombre pair de 1 » et S_1 qui signifie « j'ai vu un nombre impair de 1 » ; puis en détaillant les transitions : lire 0 ne fait pas changer d'état, tandis que lire 1 fait passer de S_0 à S_1, et de S_1 à S_0. Enfin, l'état S_0 est à la fois initial et final. □

En Java nous représentons les bits par les booléens, à savoir **false** (pour 0) et **true** (pour 1). Les textes sont représentés en machine comme des tableaux de bits (type **boolean** `[]`). Enfin, les canaux de transmission sont représentés en machine par des objets qui implémentent l'interface suivante **Channel**.

```
interface Channel {
  public boolean read() ;
  public boolean isOver() ;
}
```

La méthode `read` renvoie les bits dans l'ordre du texte transmis, tandis que la méthode `isOver` permet de déterminer si la transmission est terminée ou pas.

À titre d'exemple de la lecture d'un canal, voici une méthode qui lit le texte transmis sur un canal `c` et l'affiche sur la sortie standard.

```
static void printAll(Channel c) {
  while (!c.isOver()) {
    if (c.read()) {
      System.out.print(1) ;
    } else {
      System.out.print(0) ;
    }
  }
  System.out.println() ;
}
```

Question 2. Nous modélisons l'émetteur du texte comme un objet d'une classe **ArrayChannel** qui implémente l'interface **Channel**. Le constructeur **ArrayChannel** (**boolean** [] t) prend en argument un tableau de bits t dont les éléments pris dans l'ordre définissent le texte. En pratique, on peut partir du squelette suivant :

```
class ArrayChannel implements Channel {
  // Déclaration de quelques variables d'instance.
  ...

  // Constructeur, puis méthodes de l'interface Channel.
  ArrayChannel (boolean [] t) { ... }
  public boolean isOver() { ... }
  public boolean read() { ... }
}
```

Réponse : **Remarque** Nous prions les élèves de bien vouloir excuser la rédaction de cette question. En effet, il manque une question précise « Écrire la classe **ArrayChannel** ».

Nous espérons que notre présence dans la salle d'examen a permis de minimiser les conséquences de cet oubli.

Cela étant dit, voici un code solution.

```
class ArrayChannel implements Channel {
  private boolean [] t ;
  private int idx ;

  ArrayChannel (boolean [] t) {
    this.t = t ; idx = 0 ;
  }

  public boolean isOver() { return idx >= t.length ; }

  public boolean read() {
    if (isOver()) throw new Error ("No bit available") ;
    return t[idx++] ;
  }
}
```

□

Nous modélisons le récepteur comme une méthode **static boolean** [] readAll(**Channel** c) qui prend en argument un canal c, lit le texte transmis sur ce canal, et renvoie le texte reçu. Compte tenu de ce que la taille d'un texte n'est pas connue à l'avance, les bits du texte seront d'abord stockés dans une liste qui sera finalement convertie en tableau.

Question 3. Écrire une classe **List** des cellules de liste de bits.

Réponse :

```
class List {
  boolean val ;
  List next ;

  List (boolean val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

□

Question 4. Écrire une méthode **static boolean** `[] toArray(`**List** `p)` qui transforme une liste de bits en tableau.

Réponse :

```
static boolean [] toArray(List p) {
  // D'abord calculer la longueur de la liste
  int len = 0 ;
  for (List q = p ; q != null ; q = q.next) len++ ;

  // Puis allouer/remplir le tableau
  boolean [] t = new boolean[len] ;
  int k = 0 ;
  for (List q = p ; q != null ; q = q.next) {
    t[k] = q. val ; k++ ;
  }
  return t ;
}
```

□

Question 5. Écrire la méthode **static boolean** `[] readAll(`**Channel** `c)` de réception du texte lu sur le canal `c`.

Réponse : Il faut bien faire attention à construire la liste de bits selon l'ordre de lecture.

```
private static List readRec(Channel c) {
  if (c.isOver()) {
    return null ;
  } else {
    return new List (c.read(), readRec(c)) ;
    // ok, car les arguments sont évalués de la gauche vers la droite.
  }
}

static boolean [] readAll(Channel c) {
  List p = readRec(c) ;
  return toArray(p) ;
}
```

□

En réalité, la transmission d'un texte d'une machine à une autre peut se faire incorrectement. Nous adoptons un modèle simple, où la seule sorte d'erreur possible est l'inversion d'un bit (0 est émis et 1 est reçu, ou le contraire), et où au plus un bit est inversé.

Question 6. Pour modéliser un canal avec erreurs, nous introduisons la classe **UnsafeChannel** :

- La classe **UnsafeChannel** implémente l'interface **Channel**.
- Le constructeur **UnsafeChannel**`(`**Channel** `c)` prend un canal `c` en argument.

Un objet de la classe **UnsafeChannel** lit et retransmet les bits lus sur `c`, et ceci bit par bit. Lors de la retransmission d'un bit, si l'erreur n'est pas encore survenue et dans un cas sur 256, le bit est retransmis inversé.

Écrire la classe **UnsafeChannel**. Pour tirer au hasard un nombre entre 0 (inclus) et n (exclu) on utilisera la méthode `nextInt(`**int** `n)` d'un objet de la classe de bibliothèque **Random** (package `java.util`).

Réponse :

```
import java.util.* ;
```

```
class UnsafeChannel implements Channel {
  private Channel c ;
  private Random rand ;
  private boolean noErrorYet ;

  UnsafeChannel (Channel c) {
    this.c = c ;
    rand = new Random () ;
    noErrorYet = true ;
  }

  public boolean isOver() { return c.isOver() ; }

  public boolean read() {
    boolean b = c.read() ;
    if (noErrorYet && rand.nextInt(256) == 0) {
      noErrorYet = false ;
      return !b ; // ou if (b) return false else return true ;
    } else {
      return b ;
    }
  }
}
```

□

Partie II, détection et correction d'une erreur isolée.

Question 7. Écrire une méthode **static boolean** `add(`**boolean** `b1,`**boolean** `b2)` qui renvoie le bit égal à la somme (modulo 2) des deux bits `b1` et `b2`.

Réponse : Si on connaît le ou exclusif, c'est assez rapide.

```
static boolean add(boolean b1,boolean b2) { return b1 ^ b2 ; }
```

Ou même :

```
static boolean add(boolean b1,boolean b2) { return b1 != b2 ; }
```

Si on a pas vu le truc, ce n'est guère plus difficile.

```
static boolean add(boolean b1,boolean b2) {
  if (b1) {
    if (b2) { return false ; } else { return true ; }
  } else {
    if (b2) { return true ; } else { return true ; }
  }
}
```

□

La *somme de contrôle* d'un texte est définie comme la somme de ses bits (toujours modulo 2).

Question 8. Écrire une méthode **static boolean** `checkSum(`**boolean** `[] t)` qui renvoie la somme de contrôle du texte passé en argument.

Réponse :

```
static boolean ckeckSum(boolean [] t) {
  boolean r = false ;
  for (boolean b : t) r = add(r,b) ;
  return r ;
}
```

□

Question 9. Pour cette question seulement, on considère un canal non-sûr quelconque qui peut inverser plus d'un bit du texte transmis, sans en changer la taille. Que peut-on dire de la la transmission du message quand les sommes de contrôle des textes émis et reçu sont distinctes ?

Réponse : Un nombre impair de bits sont erronés. En effet, changer un bit dans un texte en inverse la somme (modulo 2). □

Nous supposons toujours qu'au plus une erreur surviendra. Dans ces conditions, il est possible de détecter l'existence de l'erreur, puis sa position.

Une *position* dans un texte de taille n est un entier pris entre 0 (compris) et n (exclu) — c'est tout simplement un indice dans le tableau **boolean** `[]`.

La *liste de contrôle d'ordre* k (k entier naturel) est la liste des positions i telles que $i \bmod 2^{k+1} < 2^k$ (mod est l'opérateur « reste de la division euclidienne », noté `%` en Java).

Question 10. Pour $n = 16$, donner les listes de contrôle pour k variant de 0 à 4.

Réponse :

ordre	liste de contrôle
0	$\{0, 2, 4, 6, 8, 10, 12, 14\}$
1	$\{0, 1, 4, 5, 8, 9, 12, 13\}$
2	$\{0, 1, 2, 3, 8, 9, 10, 11\}$
3	$\{0, 1, 2, 3, 4, 5, 6, 7\}$
4	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

□

Afin de lever toute ambiguïté dans les réponses, nous fixons les notations de l'écriture en base deux des entiers. Soit x entier naturel quelconque, et soit m entier tel que $2^m > x$. L'*écriture binaire* (de taille $m+1$) de x est l'unique suite de bits, notée cette fois à l'envers $b_m b_{m-1} \cdots b_1 b_0$, telle que $x = \sum_{k=0}^{k=m} b_k \cdot 2^k$. Par exemple, l'écriture binaire de taille 4 de l'entier 5 est 0101.

Question 11. Reformuler la définition de la liste de contrôle d'ordre k en considérant l'écriture binaire des positions pour m suffisamment grand.

Réponse : La liste de contrôle d'ordre k est l'ensemble des positions dont le bit d'indice k est nul. En effet, si x est représenté par $b_m b_{m-1} \cdots b_1 b_0$, alors $x \bmod 2^{k+1}$ est représenté par une suite de 0, suivie de $b_k b_{k-1} \cdots b_1 b_0$ et est donc strictement inférieur à 2^k si et seulement si le bit b_k est nul. □

On représente les listes de contrôle comme des listes de positions *croissantes*. Voici **CheckList**, une classe des cellules de liste de contrôle.

```
class CheckList {
  int pos ;
  CheckList next;

  CheckList(int p, CheckList c) {
    pos = p ; next = c ;
  }
}
```

Question 12. Écrire une méthode **static CheckList** `buildList(`**int** `k,` **int** `n)` qui renvoie la liste de contrôle d'ordre k pour n donné en argument.

Réponse : Avec la première définition.

```
static CheckList buildList(int k, int n) {
  int dk = 1;
  for (int i = 0 ; i < k ; i++) dk *= 2 ; // dk = 2^k
  int dkp = 2*dk ;                         // dkp = 2^(k+1)

  CheckList c = null;
  for (int i = n-1 ; i >= 0 ; i--) {
    if(i % dkp < dk) c = new CheckList(i,c);
  }
  return c;
}
```

Selon la définition à l'aide de la représentation binaire.

```
static boolean bitUnset(int i, int k) {
  return (i & (1 << k)) == 0 ;
}

static CheckList buildList(int k, int n) {
  CheckList c = null;
  for (int i = n-1; i >= 0 ; i--) {
    if(bitUnset(i,k)) c = new CheckList(i, c);
  }
  return c;
}
```

□

Pour tout entier $n > 0$ on considère l'entier noté $\lceil \log_2 n \rceil$ et défini comme le plus petit entier m tel que $n \leq 2^m$.

Question 13. Écrire une méthode **static int** `ceilLog2(int n)` qui renvoie $\lceil \log_2 n \rceil$.

Réponse :

```
static int ceilLog2(int n) {
  int r = 0, d = 1 ; // NB. d = 2^r
  while (d < n) {
    d *= 2 ; r++ ;   // NB. d = 2^r
  }
  // NB. r = 1 + max{ k | 2^k < n } = min{ k | 2^k >= n }
  return r ;
}
```

□

Les listes de contrôle servent à construire le *vecteur de contrôle* t_c qui est fonction d'un texte t de taille n. Le vecteur de contrôle est un texte de taille $\lceil \log_2 n \rceil + 1$. Le bit en position k dans t_c est appelé *somme de contrôle d'ordre k de t*. Ce bit est défini comme la somme (modulo 2) de tous les bits du texte original dont la position appartient à la liste de contrôle d'ordre k. On remarquera que le dernier élément de t_c est la somme de contrôle de t, qui est donc aussi la somme de contrôle d'ordre $\lceil \log_2 n \rceil$ de t.

Question 14. Écrire une méthode **static boolean** `[] checkVector(boolean [] t)` qui renvoie le vecteur de contrôle du texte t.

Réponse :

```
static boolean [] checkVector(boolean [] t){
```

```
  int n = t.length ;
  int sz = ceilLog2(n)+1 ;

  boolean [] tc = new boolean [sz];
  for(int k = 0; k < sz ; k++){
    tc[k] = false ; // Inutile, mais plus clair.
    for (CheckList p = buildList(k, n) ; p != null ; p = p.next) {
      tc[k] = add(t[p.pos], tc[k]) ;
    }
  }
  return tc ;
}
```

□

Question 15. Évaluer la complexité de la méthode précédente.

Réponse : La taille du vecteur de contrôle de l'ordre de $\log_2 n$. Pour chacun de ses éléments, le coût dominant est celui de la construction de la liste de contrôle, qui est de l'ordre de n. Le coût de la méthode est donc de l'ordre de $n \log_2 n$. □

On précise les conditions de transmission du texte. Le texte est transmis par un canal non-sûr. On note t_s le texte envoyé, et t_r le texte reçu. On se place toujours dans le cas de au plus une erreur, c'est-à-dire de au plus une inversion de bit.

Le vecteur de contrôle du texte original, noté t_c, est envoyé par un canal sûr. Le récepteur connaît donc exactement t_c. Par ailleurs, le récepteur peut calculer le vecteur de contrôle de t_r, noté t'_c,

Question 16. On suppose que le texte reçu t_r contient exactement une erreur. Soit p la position de cette erreur. Montrer que le bit d'indice k de p vaut zéro, si et seulement si $t_c[k]$ et $t'_c[k]$ diffèrent.

Réponse : Inverser un bit dans un paquet quelconque de bits en inverse la somme modulo 2. Le résultat découle ensuite de la réponse à la question 11, à savoir que la liste de contrôle d'ordre k est exactement l'ensemble des positions dont le bit d'indice k est nul. □

Question 17. Écrire **static boolean** `[] readAllCorrected(`**Channel** `c,` **Channel** `cc)`, une méthode qui renvoie le message t_s émis sur le canal non-sûr `c`. Le canal `cc` est sûr et sert à la transmission du vecteur de contrôle t_c.

Réponse :

```
static boolean [] reallAllCorrected(Channel c, Channel cc) {
  boolean [] tr = readAll(c), tc = readAll(cc) ;
  boolean [] tcc = checkVector(tr) ;
  int sz = tc.length ;
  if (tc[sz-1] == tcc[sz-1]) return tr ;

  // Il y a une erreur
  int p = 0 ;
  for (int k = sz-2 ; k >= 0 ; k--) {
    if (tc[k] == tcc[k]) p = p + 1 ; // Ou p |= 1
    p *= 2 ;                         // Ou p <<= 1
  }
  tr[p] = !tr[p] ; // Corriger le bit.
  return tr ;
}
```

On notera que la position p est construite des bits les plus significatifs vers les moins significatifs. On peut très bien écrire aussi.

```
int p = 0, k = 0, d = 1 ; // d est 2^k
while (k < sz-1) {
  if (tc[k] == tcc[k]) p += d ;
  k++ ; d *= 2 ;            // d est toujours 2^k
}
```

□

Partie III, calcul plus efficace.

Lors du calcul des sommes de contrôle, certaines additions sont effectuées plusieurs fois. Pour limiter le nombre d'additions modulo 2, nous introduisons une structure d'arbre, qui mémorise les additions intermédiaires.

Commençons par le cas simple des textes de taille 2^n. L'arbre est un arbre binaire strict, dont chaque sommet porte un bit. À tout texte de taille 2^n on associe un arbre de la façon suivante.

- À un texte de taille 1 (c'est-à-dire 2^0), on associe une feuille qui porte le bit du texte.
- À un texte de taille 2^{n+1}, on associe un arbre dont le sous-arbre gauche représente la première moitié du texte et le sous-arbre droit la seconde moitié du texte. Le bit associé est la somme (modulo 2) du texte représenté.

La figure 1 donne un un exemple d'arbre et du texte correspondant.

Question 18. Combien l'arbre d'un texte de taille 2^n contient-il de sommets ?

Réponse : De toute évidence $2^{n+1} - 1$ sommets, comme on le démontre facilement à partir de la définition, si on y tient. En effet, on a la récursion :

$$N(2^0) = 1, \qquad N(2^{n+1}) = 1 + 2N(2^n),$$

dont $N(2^n) = 2^{n+1} - 1$ est la solution bien connue. □

Question 19. Écrire une méthode **static Tree** `buidTree(`**boolean** `[] t)` qui renvoie l'arbre associé au texte t (dont la taille est de la forme 2^n). La classe des sommets d'arbre est donnée par la figure 2. On prendra soin d'écrire une méthode de complexité linéaire en la taille du texte.

Réponse : Un code sans copie de tableau :

```
// Renvoie l'arbre de t[a...b[.
static Tree buildTree(boolean [] t, int a, int b) {
  if (a+1 == b) {
    return new Tree (t[a]) ;
  } else {
    int m = (a+b)/2 ;
    Tree left = buildTree(t, a, m) ;
    Tree right = buildTree(t, m, b) ;
    return new Tree (left, Bit.add(left.sum, right.sum), right) ;
  }
}

static Tree buildTree(boolean [] t) {
  return buildTree(t, 0, t.length) ;
}
```

La complexité linéaire est atteinte, puisque chaque appel de la méthode `buildTree` ne procède en propre qu'à des opérations de coût constant, et qu'il y a de l'ordre de 2^{n+1} appels au total (compter les sommets de l'arbre produit).

Figure 1: Arbre du texte `00100110`

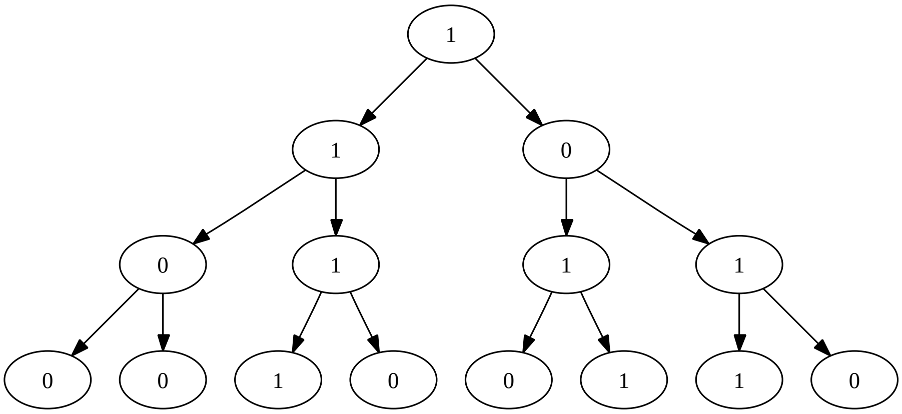


Figure 2: Classe des sommets d'arbre.

```
class Tree {
  boolean sum ;
  Tree left, right ; // Fils gauche et droit.

  Tree (boolean b) { // Constructeur des feuilles
    this.sum = b ;
  }

  Tree (Tree left, boolean b, Tree right) { // Constructeur des sommets internes.
    this.left = left ; this.right = right ; this.sum = b ;
  }
}
```

On notera qu'un code qui calculerait des sous-tableaux serait de complexité de l'ordre de $n2^n$. En effet, chaque appel pour un tableau de taille m procéderait alors à de l'ordre de m opérations (allouer deux tableaux de taille moitié). Pour une profondeur de récursion k donnée, l'ordre de grandeur est donc 2^k fois 2^{n-k}. Soit un coût en 2^n pour chaque profondeur de récursion. Il en irait de même d'une solution qui recalculerait les sommes au lieu de les extraire des sous-arbres. □

Question 20. Calculer (à la main) le vecteur de contrôle du texte de la figure 1 à l'aide de l'arbre, en s'efforçant de limiter le nombre d'additions de bits effectuées.

Réponse : Le vecteur de contrôle contient ici 4 sommes.

On repère les étages de l'arbre par des entiers croissants : de 0 pour les feuilles, à 3 pour la racine, on note que cette numérotations correspond aux indices du vecteur de contrôle.

- À l'ordre 3 (toutes les positions), on prend le bit de la racine, soit 1.
- À l'ordre 2 (liste de positions $\{0, 1, 2, 3\}$), on prend le bit du sous-arbre gauche, soit 1.
- À l'ordre 1 (liste de positions $\{0, 1, 4, 5\}$), on somme un bit sur deux de l'étage numéro 1, soit $0 + 1 = 1$.
- Enfin, à l'ordre 0, on somme une feuille sur deux, soit $0 + 0 + 1 + 0 = 1$.

□

Question 21. Relier le chemin vers une feuille donnée de l'arbre et la représentation binaire de la position dans le texte du bit contenu dans la feuille. Les chemins sont des suites de zéros et de uns, où zéro signifie « à gauche » et un « à droite ». Par exemple le chemin vers la quatrième feuille de l'arbre de la figure 1 est noté 011.

Réponse : Par construction, le chemin est la représentation binaire de la position. □

Question 22. Écrire une méthode **static boolean** `checkSum(`**int** `k,` **int** `n,` **Tree** `t)` qui renvoie la somme de contrôle d'ordre k du texte représenté par l'arbre t, en d'efforçant de limiter les additions de bits effectuées. La taille du texte représenté par t vaut 2^n (avec $0 \leq k \leq n$).

Réponse : La réponse à la question précédente nous permet de déduire que la somme de contrôle d'ordre k est la somme des bits d'un sommet sur deux de l'étage k. Cela vaut également pour la somme de contrôle d'ordre n, en considérant la représentation binaire de taille $n + 2$. Dans le cas $k < n$, il reste à sélectionner un sommet sur deux à partir de l'étage qui précède k.

```
static boolean checkSum(int k, int n, Tree t) {
  if (k == n) { // Ne peut survenir qu'à la racine.
    return t.sum ;
  } else if (k == n-1) {
    return t.left.sum ;
  } else {
    return Bit.add(checkSum(k, n-1, t.left),checkSum(k, n-1, t.right)) ;
  }
}
```

□

Question 23. Quelle est la complexité du calcul du vecteur de contrôle avec la nouvelle technique, en fonction de la taille du texte, notée 2^n.

Réponse : Il faut tenir compte de la construction de l'arbre (d'un ordre de grandeur proportionnel à 2^n, plus exactement de terme dominant 2^{n+1}, car l'arbre possède $2^{n+1} - 1$ sommets) et du calcul de $n + 1$ sommes de contrôle. On peut compter les additions ou les appels à `checkSum`, puisqu'on peut leur affecter un nombre borné d'opérations. Pour $k = n$, un appel. Pour $k < n$, il y a un appel initial, puis 2 appels, ... jusqu'à atteindre les 2^{n-k-1} sommets de l'étage numéroté $k+1$, Soit finalement dans ce cas.

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

Soit au total, pour $k' = n - k$ variant de 0 à $n-1$.

$$\sum_{k'=0}^{n-1} (2^{k'} - 1) = n + 2^n - 1$$

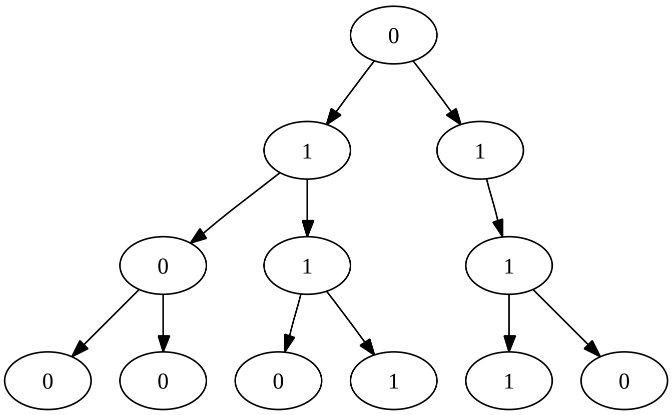
Auquel on peut, si on veut être précis, ajouter l'appel du cas $n = k$.

Bref, au total, un coût dont l'ordre de grandeur est linéaire, en la taille du texte (qui est 2^n). □

On supprime la restriction « le texte est de taille 2^n» et on considère le cas général de textes de taille n (avec $0 < n$). On souhaite maintenir la relation de la question 21, entre représentation binaire d'une position et chemin de l'arbre. Les nouveaux arbres ne sont plus strictement binaires : ils contiennent des sommets à un seul fils et descendre vers ce fils correspond à zéro dans un chemin vers une feuille.

Question 24. Dessiner l'arbre obtenu pour le texte `000110`.

Réponse :



□

Question 25. Formuler une nouvelle définition des arbres dans le cas général. Il faut considérer un nouvel élément : la taille des chemins de l'arbre ciblé. On se placera dans le cadre d'un texte t de taille n, avec une taille de chemin cible k, avec $0 < n \leq 2^k$.

Réponse :

- Si $k = 0$, alors n vaut nécessairement 1 et on construit une feuille.
- Si $k > 0$, alors il y a ensuite deux sous-cas, selon qu'il existe des bits d'indice $k-1$ valant 1 dans les représentations binaires des positions de t ou pas.
  - Si $2^{k-1} < n$, alors on construit les deux arbres représentant respectivement les 2^{k-1} premiers bits du texte, et les bits de l'indice 2^{k-1} à la fin du texte (tous deux avec une taille cible 2^{k-1}). On construit ensuite un sommet à deux fils.
  - Sinon ($n \leq 2^{k-1}$) on ne construit qu'un seul arbre pour tout le texte et une taille cible 2^{k-1}, puis on construit un sommet à un fils.

□