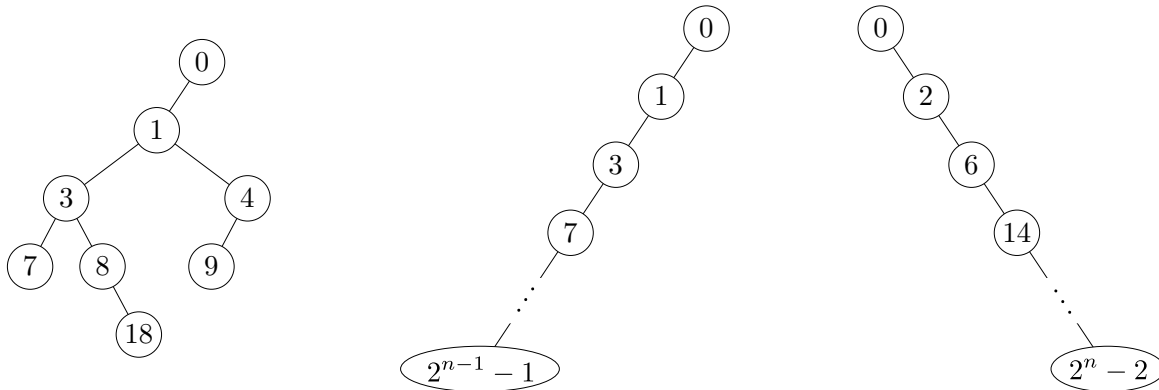


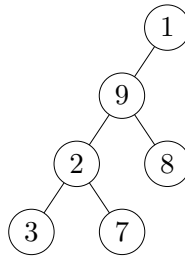
Question 1 –



Question 2.a – Cet arbre est représenté par le tableau :

[|E 9; E 8; E 6; Abs; E 7; Abs; E 5; Abs; Abs; Abs; Abs; Abs; Abs; E 4; E 3|]

Question 2.b – Ce tableau représente l'arbre



Question 3.a –

```

let lastE tab =
  let res = ref (Array.length tab - 1) in
  while !res >= 0 && tab.(!res) = Abs do
    decr res
  done;
  !res;;

```

Question 3.b –

```

let supprAbs (tab: 'a etiq array): 'a sosa =
  let taille = lastE tab + 1 in
  let res = Array.make taille Abs in
  for i = 0 to taille - 1 do
    res.(i) <- tab.(i)
  done;
  res;;

```

Question 4.a – Le tableau suivant est non valide :

[|Abs; E 0|]

En effet, le noeud dont le numéro de Sosa vaut 0 est absent, c'est à dire que la racine est absente. Ainsi, aucun arbre ne correspond à ce tableau.

Question 4.b – Pour qu'un tableau non vide soit valide, il faut que tout noeud puisse être atteint à partir de la racine. Le plus simple est de vérifier que si une case ne contient pas **Abs**, alors son père ne contient pas **Abs**.

```
let valide (arb: 'a sosa): bool =
  let n = Array.length arb in
  let s = ref 1 in
  let res = ref true in
  while !s < n && !res do
    if arb.(!s) <> Abs && arb.(!s-1)/2 = Abs then res := false;
    incr s
  done;
  !res;;
```

Question 5 –

```
let supprE (li: 'a etiq list): 'a list =
  let f (e: 'a etiq): 'a = match e with
    | E e -> e
    | Abs -> failwith "supprE: construteur Abs dans la liste"
  in
  List.map f li;;
```

Question 6 –

```
let ancetres (arb: 'a sosa) (s0: int): 'a list =
  if s0 < 0 || s0 >= Array.length arb then
    failwith "ancetres: numéro de Sosa invalide";
  let rec aux s = match s with
    | 0 -> []
    | s -> let pere = (s-1)/2 in
            arb.(pere)::(aux pere)
  in
  supprE (aux s0);;
```

Question 7 – La hauteur d'un arbre binaire peut être définie par induction :

- Si l'arbre est vide, sa hauteur est (-1) .
- Sinon, l'arbre est de la forme (e, G, D) . Sa hauteur est alors $1 + \max(h_G, h_D)$ où h_G est la hauteur de G et h_D est la hauteur de D .

Question 8.a –

```
1 let rec hautAux (arb: 'a sosa) (i: int): int =
2   if i >= Array.length arb || i < 0 then -1
3   else if arb.(i) = Abs then -1
4   else 1 + max (hautAux arb (2*i+1)) (hautAux arb (2*i+2));;
```

Question 8.b –

```
let haut (arb: 'a sosa): int =  
    hautAux arb 0;;
```

Question 8.c – Comme dans l'énoncé, on fixe un arbre A et « `arb: 'a sosa` » le tableau qui le représente. Lors de l'évaluation de « `haut arb` », le nombre d'appels à la fonction `hautAux` est :

$$2n + 1 \text{ où } n \text{ est le nombre de noeuds dans l'arbre } A.$$

Pour le prouver, on montre par récurrence sur $m \in \mathbb{N}$ la proposition :

$(\mathcal{P}_m)$: $\left\{ \begin{array}{l} \text{Pour tout } i \in \mathbb{N}, \text{ si le sous-arbre d'indice } i \text{ de } A \text{ contient } m \text{ noeuds, alors lors de l'éva-} \\ \text{luation de « } \text{hautAux arb } i \text{ », la fonction } \text{hautAux} \text{ est appelée } 2m + 1 \text{ fois.} \end{array} \right.$

Initialisation. Pour $m = 0$, si le sous-arbre d'indice i possède m noeuds, c'est qu'il est vide, c'est à dire qu'on est dans le cas de la ligne 2 ou de la ligne 3 (dans la fonction `hautAux`). Il n'y a donc aucun appel récursif et le nombre d'appels à la fonction `hautAux` est bien $2m + 1 = 1$ (ne pas oublier de compter l'appel principal).

Hérédité. Soit $m \in \mathbb{N}^*$. On suppose $(\mathcal{P}_{m'})$ pour tout $m' < m$ et on montre $(\mathcal{P}_m)$. Soit $i \in \mathbb{N}$ tel que le sous-arbre d'indice i noté B possède m noeuds. Alors :

- Le sous-arbre gauche de B noté G est le sous-arbre d'indice $2i + 1$ de A et possède $m_G < m$ noeuds. Par hypothèse de récurrence, lors de l'évaluation de « `hautAux arb (2*i+1)` », la fonction `hautAux` est appelée $2m_G + 1$ fois.
- De même, lors de l'évaluation de « `hautAux arb (2*i+2)` », la fonction `hautAux` est appelée $2m_D + 1$ fois avec m_D le nombre de noeuds dans la sous-arbre droit de B .

Finalement, lors de l'évaluation « `hautAux arb i` », on dénombre : un appel principal, un appel récursif sur G et un appel récursif sur D . Au total, le nombre d'appels à la fonction `hautAux` est :

$$\begin{aligned} 1 + (2m_G + 1) + (2m_D + 1) &= 2(m_G + m_D + 1) + 1 \\ &= 2m + 1 \end{aligned}$$

Conclusion. Lors de l'évaluation de « `haut arb` », on évalue « `hautAux arb 0` ». Le sous-arbre d'indice 0 de A est égal à A , donc par $(\mathcal{P}_n)$, il y a $2n + 1$ appels à `hautAux`.

Question 8.d – D'après la question précédente, lors de l'évaluation de « `haut arb` », il y a $2n + 1 = \Theta(n)$ appels à la fonction `hautAux`. De plus, sans compter les appels récursifs, la fonction `hautAux` s'exécute en temps constant. En conclusion :

La complexité de `haut` est en $\Theta(n)$ avec n le nombre de noeuds dans l'arbre.

Question 9 –

```
let larg1 (arb: 'a bin): 'a list =  
    let file = Queue.create() in  
    Queue.push arb file;  
    let rec aux (): 'a list =  
        if Queue.is_empty file then [] else  
            match Queue.pop file with  
            | Vide -> aux()  
            | N(e, g, d) -> Queue.push g file;  
                               Queue.push d file;  
                               e :: aux()  
    in aux();;
```

Question 10 – Pour calculer le parcours en largeur d'un arbre « arb: 'a sosa », il suffit de récupérer dans l'ordre du tableau les étiquettes des noeuds.

```
let larg2 (arb: 'a sosa): 'a list =
  let res = ref [] in
  for i = Array.length arb - 1 downto 0 do
    match arb.(i) with
    | Abs -> ()
    | E e -> res := e :: !res
  done;
  !res ;;
```

Question 11 –

```
(* La fonction aux renvoie le sous arbre d'indice s de arb *)
let binOfSosa (arb: 'a sosa): 'a bin =
  let rec aux (s: int): 'a bin =
    if s < 0 then failwith "binOfSosa: s < 0"
    else if s >= Array.length arb then Vide
    else match arb.(s) with
         | Abs -> Vide
         | E e -> N (e, aux (2*s+1), aux (2*s+2))
  in
  aux 0;;
```

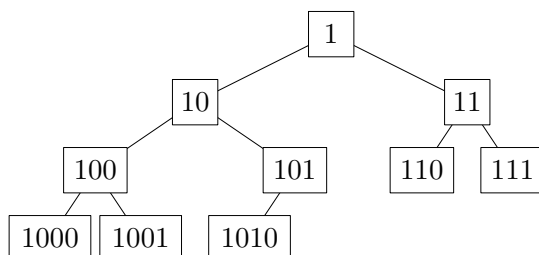
Question 12.a –

```
let pref (f: 'a -> int -> unit) (arb0: 'a bin): unit =
  let rec aux (arb: 'a bin) (s: int): unit = match arb with
    | Vide -> ()
    | N (e, g, d) -> f e s;
                      aux g (2*s+1);
                      aux d (2*s+2);
  in
  aux arb0 0;;
```

Question 12.b –

```
(* sosaMax est le numéro de Sosa maximal pour un noeud de arb *)
let sosaOfBin (arb: 'a bin): 'a sosa =
  let sosaMax = ref (-1) in
  pref (fun _ s -> if !sosaMax < s then sosaMax := s) arb;
  let (arb2: 'a sosa) = Array.make (!sosaMax + 1) Abs in
  pref (fun e s -> arb2.(s) <- E e) arb;
  arb2;;
```

Question 13.a – On obtient l'arbre :



Question 13.b –

Pour obtenir l'arbre B à partir de l'arbre obtenu dans la question précédente, il suffit de supprimer les bits de poids fort (qui valent 1), de remplacer les 0 par des G et de remplacer les 1 par des D .

Soit A un arbre binaire. Pour tout noeud x de A , on note $I(x)$ l'entier $s + 1$ où s est le numéro de Sosa de x . Il s'agit de montrer que le chemin à emprunter pour accéder à un noeud x s'obtient à partir de l'écriture en base 2 de $I(x)$ en utilisant la transformation décrite ci-dessus. On le montre par récurrence forte sur $I(x) \in \mathbb{N}^*$.

Initialisation. Si $I(x) = 1$ alors le noeud x est la racine et la proposition est vraie.

Hérédité. On suppose $I(x) \geq 2$ et que la propriété est vraie pour tout noeud y tel que $I(y) < I(x)$. Soit y le père de x . On a deux cas :

→ Si x est le fils gauche de y , alors $I(x) - 1 = 2(I(y) - 1) + 1$. Donc $I(x) = 2I(y)$.

De plus, pour accéder à x , il faut accéder à y puis aller dans le sous-arbre gauche. Il faudrait donc que l'écriture en base 2 de $I(x)$ soit égal à l'écriture en base 2 de $I(y)$ à laquelle on a concaténé le bit 0. C'est effectivement le cas car multiplier par 2 consiste à ajouter le bit 0 à la fin de la représentation binaire.

→ Sinon, x est le fils droit de y . La preuve est similaire.

Question 14 – On écrit d'abord une fonction intermédiaire `getListBits` qui renvoie la liste des bits de l'entier n donné en entrée. Le bit de poids fort (qui vaut 1) n'apparaît pas dans la liste. Les autres bits apparaissent de gauche à droite (le bit de poids faible est à la fin de la liste).

```
(* Hypothèse: n >= 1 *)
let getListBits n0 =
  let rec aux acc n = match n with
    | 1 -> acc
    | n -> aux (n mod 2 :: acc) (n/2)
  in
  aux [] n0;;

let etiq (arb: 'a bin) (s: int): 'a =
  let rec etiqFromListBits arb li = match arb,li with
    | Vide, _ -> failwith "etiq: numero de Sosa invalide"
    | N(e,_,_), [] -> e
    | N(_,g,_), 0::q -> etiqFromListBits g q
    | N(_,_,d), 1::q -> etiqFromListBits d q
    | _ -> failwith "etiq: ne devrait pas arriver"
  in
  if s < 0 then failwith "Numero de Sosa invalide";
  etiqFromListBits arb (getListBits (s+1));;
```

Question 15 – Pour tout $p \in \mathbb{N}$, on note $S_p = \llbracket 2^p - 1, 2^{p+1} - 2 \rrbracket$. Montrons par récurrence forte sur $p \in \mathbb{N}$:

(Q_p) : S_p est l'ensemble des numéros de Sosa possibles pour un noeud à profondeur p .

- Si $p = 0$ alors un noeud à profondeur p est forcément la racine dont le numéro de Sosa est 0. On a bien $S_0 = \{0\}$.

- Soit $p \geq 1$. On suppose $\mathcal{Q}_{p-1}$ et on montre $\mathcal{Q}_p$.

Soit x un noeud à profondeur p , alors le père de x est à profondeur $p - 1$. Par l'hypothèse de récurrence, le numéro de Sosa du père de x appartient à S_{p-1} . Donc l'ensemble des numéros de Sosa possibles pour x est :

$$\{2s + 1 : s \in S_{p-1}\} \cup \{2s + 2 : s \in S_{p-1}\} = S_p$$

On remarque que pour tout $s \in S_p$, on a $\lfloor \log_2(s+1) \rfloor = p$. Lorsqu'on considère le tableau « **arb**: 'a sosa » de taille $n \in \mathbb{N}^*$, les numéros de Sosa des noeuds appartiennent à $\llbracket 0; n-1 \rrbracket$. En particulier, il existe un noeud dont le numéro de Sosa est $n - 1$ puisqu'on a supposé que la dernière case de **arb** ne contient pas **Abs**. La hauteur de l'arbre est le maximum des profondeurs des noeuds donc :

$$h = \lfloor \log_2(n) \rfloor$$

Question 16.a – Pour tout $n \in \mathbb{N}^*$:

Le nombre de bits dans la représentation en base 2 de n est $\lfloor \log_2(n) \rfloor + 1$.

En effet, si n possède k bits alors :

$$\begin{aligned} 2^{k-1} &\leq n < 2^k \\ k-1 &\leq \log_2(n) < k \\ k &\leq \log_2(n) + 1 < k+1 \\ k &= \lfloor \log_2(n) \rfloor + 1 \end{aligned}$$

Question 16.b –

```

(* Suppose n >= 0, renvoie 0 pour 0 *)
let rec nbBits n = match n with
| 0 -> 0
| n -> 1 + nbBits (n/2);;

```

Question 17.a –

```

let hautBis (arb: 'a sosa): int =
  nbBits (Array.length arb) - 1;;

```

Question 17.b – La fonction **hautBis** a une complexité en :

$\Theta(\log_2(n))$ où n est la taille du tableau **arb**.

En effet, la fonction **nbBits** est de complexité linéaire en le nombre de bits dans n , c'est à dire que son temps d'exécution est en $\Theta(\log_2(n))$.

Question 18 – On fixe $k \in \mathbb{N}^*$ et on définit le numéro de Sosa d'un noeud de la manière suivante :

- Le numéro de Sosa de la racine est 0.
- Étant donné un noeud dont le numéro de Sosa est s , pour tout $i \in \llbracket 1, k \rrbracket$ le fils numéro i a pour indice $ks + i$.

Comme dans le cas binaire, un tableau « **arb: 'a sosa** » de taille $\mathbf{n} \in \mathbb{N}$ représente un arbre binaire A lorsque pour tout $\mathbf{i} \in \llbracket 0, \mathbf{n} - 1 \rrbracket$, les conditions suivantes sont respectées :

- (i) Si A possède un noeud dont le numéro de Sosa est $\mathbf{i}$, alors **arb.**($\mathbf{i}$) contient « **E e** » où **e** est l'étiquette du noeud.
- (ii) Si A ne possède pas de noeud dont le numéro de Sosa est $\mathbf{i}$, alors **arb.**($\mathbf{i}$) vaut **Abs**.
- (iii) Si $\mathbf{i} = \mathbf{n} - 1$ alors **arb.**($\mathbf{i}$) ne vaut pas **Abs**.