I. Tas

⚠ Les algorithmes sont écrits pour des tableaux d'indice 1 à n.

1. Tas binaires

Function PARENT(i)
  return $\lfloor i/2 \rfloor$

Function LEFT(i)
  return $2i$

Function RIGHT(i)
  return $2i+1$

Procedure PERCOLATEUP(T, i)

```
Require: T[1..(i−1)] tas avec 1 ≤ i ≤ T.LENGTH indice du noeud à percoler)
Ensure: T[1..i] tas
    j ← PARENT(i)
    while (j > 0) and (T[j] ≺ T[i]) do                ≺ : 'moins prioritaire que'
        PERMUT(T[i], T[j])
        i ← j
        j ← PARENT(i)
```

L'opération d'ajout d'un élément e est donc réalisée par l'algorithme suivant :

Procedure ADD(T, e)

```
Require: T.SIZE < T.LENGTH
Ensure: T is a heap.
    T.SIZE ← T.SIZE + 1
    T[T.SIZE] ← e
    PERCOLATEUP(T, T.SIZE)
```

Procedure PERCOLATEDOWN(T, i)

```
Require: T[LEFT(i)..T.SIZE] et T[RIGHT(i)..T.SIZE] sont des tas et
         1 ≤ i ≤ T.SIZE indice du noeud à percoler
Ensure: T[i..T.SIZE] is a heap.
    l ← LEFT(i)
    r ← RIGHT(i)
    if (l ≤ T.SIZE) and (T[i] ≺ T[l]) then            ≺ : 'moins prioritaire que'
        m ← l
    else
        m ← i
    if (r ≤ T.SIZE) and (T[m] ≺ T[r]) then m ← r
    if m ≠ i then
        PERMUT(T[i], T[m])
        PERCOLATEDOWN(T, m)
```

Suppression d'un élément :

Function REMOVE(T)

```
Require: T.SIZE ≥ 1
Ensure: T is a heap.
    m ← T[1]
    T[1] ← T[T.SIZE]
    T.SIZE ← T.SIZE - 1
    PERCOLATEDOWN(T, 1)
    return m
```

Construction d'un tas binaire :

Procedure BUILDHEAP(T)

```
Require: T.LENGTH is the actual
         number of elements of T
         and is ≥ 1
Ensure: T is a heap.
    T.SIZE ← T.LENGTH
    for i = ⌊T.SIZE/2⌋ downto 1 do
        PERCOLATEDOWN(T, i)
```

2. Tas binomiaux

Lier deux arbres d'ordre $o \rightarrow$ nouvel arbre d'ordre $o+1$.

Procedure BinomialLink(y, z)

p(y) $\leftarrow z$
next(y) $\leftarrow$ child(z)
child(z) $\leftarrow y$
order(z) $\leftarrow$ order(z) + 1

Création d'un Tas Binomial vide.

Procedure CreateBinomial-Heap

$H \leftarrow$ Allocate-Heap()
head(H) $\leftarrow$ NIL
return (H)

Pour fusionner deux tas binomiaux, il faut d'abord créer une liste de toutes les racines des deux tas classées par "ordre" croissant.

Procedure RootMerge(H1,H2)

$H \leftarrow$ CreateBinomialHeap()
if $order(head(H2)) < order(head(H1))$ **then**
  $head(H) \leftarrow head(H2)$;$current2 \leftarrow next(head(H2))$;$current1 \leftarrow head(H1)$
else
  $head(H) \leftarrow head(H1)$;$current1 \leftarrow next(head(H1))$; $current2 \leftarrow head(H2)$
$current \leftarrow head(H)$
while $current1 \neq NIL$ *and* $current2 \neq NIL$ **do**
  if $order(current1) > order(current2)$ **then**
    $next(current) \leftarrow current2$; $current \leftarrow next(current)$;
    $current2 \leftarrow next(current2)$
  else
    $next(current) \leftarrow current1$; $current \leftarrow next(current)$;
    $current1 \leftarrow next(current1)$
if $current1 = NIL$ **then** $tail \leftarrow current2$ **else** $tail \leftarrow current1$
while $tail \neq NIL$ **do**
  $next(current) \leftarrow tail$; $current \leftarrow next(current)$; $tail \leftarrow next(tail)$
return (head(H))

Procedure MergeBinomialHeap(H_1, H_2)

```
H ← CreateBinomialHeap()
head(H) ← RootMerge (H1, H2)
if head(H) = NIL then return H
prev_x ← NIL
x ← head(H)
next_x ← next(x)
while next_x ≠ NIL do
    if ( order(x) ≠ order(next_x) ) or ( (next(next_x) ≠ NIL) and (order(x) = order(next(next_x)) ) then
        prev_x ← x
        x ← next_x
    else
        if key(x) ≤ key(next_x) then
            next(x) ← next(next_x)
            BinomialLink(next_x, x)
        else
            if prev_x = NIL then
                head(H) ← next_x
            else
                next(prev_x) ← next_x
            BinomialLink(x, next_x)
            x ← next_x
    next_x ← next(x)
return H
```

II. Arbres de recherche

1. Abres binaires de recherche BST

Procedure InOrderTreeWalk(x)

```
if x ≠ NIL then
    InOrderTreeWalk(left(x))
    PrintKey(key(x))
    InOrderTreeWalk(right(x))
```

Function TreeSearch(x, k) /* récursif */

```
if (x = NIL) or (k = key(x)) then return x
if (k < key(x)) then
    return TreeSearch(left(x), k)
else
    return TreeSearch(right(x), k)
```

Function TreeSearch(x, k) /*itératif*/

```
while (x ≠ NIL) or (k ≠ key(x)) do
    if (k < key(x)) then x ← left(x)
    else x ← right(x)
return x
```

Function TreeMinimum(x)

```
while left(x) ≠ NIL do x ← left(x)
return x
```

Function TreeMaximum(x)

```
while right(x) ≠ NIL do x ← right(x)
return x
```

Function TreeSuccessor(x)

```
if right(x) ≠ NIL then return TreeMinimum(right(x))
y ← parent(x)
while (y ≠ NIL) and (x == right(y)) do
    x ← y
    y ← parent(y)
return y
```

Function TreePredecessor(x)

```
if left(x) ≠ NIL then return TreeMaximum(left(x))
y ← parent(x)
while (y ≠ NIL) and (x == left(y)) do
    x ← y
    y ← parent(y)
return y
```

Procedure TreeInsert(T, z)

$y \leftarrow NIL$
$x \leftarrow root(T)$
while $x \neq NIL$ **do**
    $y \leftarrow x$
    if $key(z) < key(x)$ **then** $x \leftarrow left(x)$
    else $x \leftarrow right(x)$
$parent(z) \leftarrow y$
if $y = NIL$ **then** $root(T) \leftarrow z$
else if $key(z) < key(y)$ **then** $left(y) \leftarrow z$
else $right(y) \leftarrow z$

Procedure Transplant(T, u, v)

if $parent(u) = NIL$ **then** $root(T) \leftarrow v$
else if $u = left(parent(u))$ **then** $left(parent(u)) \leftarrow v$
else $right(parent(u)) \leftarrow v$
if $v \neq NIL$ **then** $parent(v) \leftarrow parent(u)$

Procedure TreeDelete(T, z)

if $left(z) = NIL$ **then** Transplant($T, z, right(z)$)
else if $right(z) = NIL$ **then** Transplant($T, z, left(z)$)
else
    $y \leftarrow$ TreeMinimum($right(z)$)
    if $parent(y) \neq z$ **then**
        Transplant($T, y, right(y)$)
        $right(y) \leftarrow right(z)$
        $parent(right(y)) \leftarrow y$
    Transplant(T, z, y)
    $left(y) \leftarrow left(z)$
    $parent(left(y)) \leftarrow y$

Procedure LeftRotate(T, x)

$y \leftarrow right(x)$
$right(x) \leftarrow left(y)$
if $left(y) \neq nil(T)$ **then** $parent(left(y)) \leftarrow x$
$parent(y) \leftarrow parent(x)$
if $parent(x) = nil(T)$ **then** $root(T) \leftarrow y$
else if $x = left(parent(x))$ **then** $left(parent(x)) \leftarrow y$
else $right(parent(x)) \leftarrow y$
$left(y) \leftarrow x$
$parent(x) \leftarrow y$

2. Arbres rouge/noir

Procedure RBTreeInsert(T, z)

```
y ← nil(T)
x ← root(T)
while x ≠ nil(T) do
    y ← x
    if key(z) < key(x) then x ← left(x)
    else x ← right(x)
parent(z) ← y
if y = nil(T) then root(T) ← z
else if key(z) < key(y) then left(y) ← z
else right(y) ← z
left(z) ← nil(T)
right(z) ← nil(T)
color(z) ← RED
RBTreeInsertFixup(T, z)
```

Procedure RBTreeInsertFixup(T, z)

```
while color(parent(z)) = RED do
    if parent(z) = left(parent(parent(z))) then
        y ← right(parent(parent(z)))
        if color(y) = RED then
            color(parent(z)) ← BLACK
            color(y) ← BLACK
            color(parent(parent(z))) ← RED
            z ← parent(parent(z))
        else
            if z = right(parent(z)) then z ← parent(z) ; LeftRotate(T, z)
        color(parent(z)) ← BLACK
        color(parent(parent(z))) ← RED
        RightRotate(T, parent(parent(z)))
    else
        Comme dans la clause "then" en échangeant TOUS les "right" et "left"
color(root(T)) ← BLACK
```

Procedure RBTreeDelete(T, z)

```
y ← z
ycolor ← color(y)
if left(z) = nil(T) then
    x ← right(z)
    Transplant(T, z, right(z))
else if right(z) = nil(T) then
    x ← left(z)
    Transplant(T, z, left(z))
else
    y ← TreeMinimum(right(z))
    ycolor ← color(y)
    x ← right(y)
    if parent(y) = z then
        parent(x) ← y
    else
        Transplant(T, y, left(y))
        right(y) ← right(z)
        parent(right(y)) ← y
    Transplant(T, z, y)
    left(y) ← left(z)
    parent(left(y)) ← y
    color(y) ← color(z)
if ycolor = BLACK then RBTreeDeleteFixup(T, x)
```

Procedure RBTreeDeleteFixup(T, x)

```
while x ≠ root(T) and color(x) = BLACK do
    if x = left(parent(x)) then
        w ← right(parent(x))
        if color(w) = RED then
            color(w) ← BLACK
            color(parent(x)) ← RED
            LeftRotate(T, parent(x))
            w ← right(parent(x))
        if (color(left(w)) = BLACK) and (color(right(w)) = BLACK) then
            color(w) ← RED
            x ← parent(x)
        else
            if color(right(w)) = BLACK then
                color(left(w)) ← BLACK
                color(w) ← RED
                RightRotate(T, w)
                w ← right(parent(x))
            color(w) ← color(parent(x))
            color(parent(x)) ← BLACK
            color(right(w)) ← BLACK
            LeftRotate(T, parent(x))
            x ← root(T)
    else
        Comme dans la clause "then" en échangeant TOUS les "right" et "left"
color(x)) ← BLACK
```

III. B-Arbres

Function BTreeSearch(x, k)

```
i ← 1
while i ≤ numkeys(x) and key(x,i) < k do
    i ← i + 1
if i ≤ numkeys(x) and k = key(x,i) then return (x,i)
else if leaf(x) then return NIL
else
    DiskRead(child(x,i))
    return BTreeSearch(child(x,i), k)
```

La création d'un arbre vide se fait avec BTreeCreate de complexité $O(1)$:

Function BTreeCreate(T)

```
x ← AllocateNode()
leaf(x) ← TRUE
numkeys(x) ← 0
DiskWrite(x)
root(T) ← x
```

La fonction BTreeSplitChild prend en entrée un nœud interne x non plein et un index i tel que $child(x, i)$ est un fils plein de x. Les deux nœuds sont supposé résider en mémoire centrale. La fonction découpe alors ce fils en deux et met à jour x pour qu'il référence le nouveau fils. Si le nœud devant être coupé est la racine, on crée d'abord une nouvelle racine vide qui sera le parent de l'ancienne racine. La hauteur du B-Tree est alors augmentée de 1. Cette manière d'insérer une clé dans un B-Tree fait que, contrairement aux autres arbres, un B-Tree grandi par la racine et non pas par les feuilles.

Function BTreeSplitChild(x, i)

```
z ← AllocateNode()
y ← child(x,i)
leaf(z) ← leaf(y)
numkeys(z) ← t − 1
for j = 1 to t − 1 do key(z,j) ← key(y,j + t)

if not leaf(y) then
    for j = 1 to t do
        child(z,j) ← child(y,j + t)
numkeys(y) ← t − 1
for j = numkeys(x) + 1 downto i + 1 do
    child(x,j + 1) ← child(x,j)
child(x,i + 1) ← z
for j = numkeys(x) downto i do
    key(x,j + 1) ← key(x,j)
key(x,i) ← key(y,t)
numkeys(x) ← numkeys(x) + 1
DiskWrite(y)
DiskWrite(z)
DiskWrite(x)
```

A l'aide de cette fonction, nous pouvons écrire l'algorithme suivant qui insère une clé k dans l'arbre T en effectuant un seul parcours de l'arbre.

Function BTREEINSERT(T, k)

```
r ← root(T)
if numkeys(r) = 2t − 1 then
    s ← ALLOCATENODE()
    root(T) ← s
    leaf(s) ← FALSE
    numkeys(s) ← 0
    child(s, 1) ← r
    BTREESPLITCHILD(s, 1)
    BTREEINSERTNONFULL(s, k)
else BTREEINSERTNONFULL(r, k)
```

Function BTREEINSERTNONFULL(x, k)

```
i ← numkeys(x)
if leaf(x) then
    while (i ≥ 1) and (k < key(x, i)) do key(x, i + 1) ← key(x, i) ; i ← i − 1
    key(x, i + 1) = k
    numkeys(x) ← numkeys(x) + 1
    DISKWRITE(x)
else
    while (i ≥ 1) and (k < key(x, i)) do i ← i − 1
    i ← i + 1
    DISKREAD(child(x, i))
    if numkeys(child(x, i)) = 2t − 1 then BTREESPLITCHILD(x, i)
    if k > key(x, i) then i ← i + 1
    BTREEINSERTNONFULL(child(x, i), k)
```

BTREEDELETE :

1. Si la clé k est dans le nœud x et que x est une feuille, supprimer la clé de la feuille.
2. Si la clé k est dans le nœud x et que x est un nœud interne
   (a) Si le fils y qui précède k dans le nœud x possède au moins t clés, trouver le prédécesseur k' de k dans le sous-arbre enraciné en y. Supprimer récursivement la clé k' dans y et remplacer k par k' dans x. La recherche et suppression du prédécesseur peut se faire en une seule passe.
   (b) Si y possède un nombre de clé inférieur à t, examiner le fils z qui suit la clé k dans x. Si z possède au moins t clés, trouver le successeur k' de k dans le sous-arbre enraciné en z. Supprimer récursivement k' et remplacer k par k'.
   (c) Si y et z ont tous deux exactement $t - A$ clé, fusionner k et les clés de z dans $y$de telle sorte que x perde à la fois la clé k et la référence au fils z. y contient maintenant $2t - 1$ clés. le nœud z peut être détruit et la suppression de k continue récursivement sur y
3. Si la clé k n'est pas présente dans le nœud interne x, déterminer la racine $child(x, i)$ du sous-arbre devant contenir la clé. Si $child(x, i)$ ne possède que $t - 1$ clés, exécuter une des deux sous étapes suivantes pour garantir que l'on puisse descendre dans un nœud possédant au moins t clés. Continuer la suppression récursive sur le bon fils de x.
   (a) Si $child(x, i)$ possède exactement $t - 1$ clé mais à un frère immédiat possédant au moins t clés, ajouter une clé à $child(x, i)$ en descendant une clé de x dans $child(x, i)$, en montant une clé du frère immédiat gauche ou droite de $child(x, i)$ dans le nœud x et en déplaçant le fils concerné du frère dans $child(x, i)$.
   (b) Si $child(x, i)$ et ses deux frères immédiats ont $t - 1$ clés, fusionner $child(x, i)$ avec un de ses frères, ce qui nécessite de faire descendre un clé de x en tant que clé médiane dans $child(x, i)$