I. Heap

1. Binary heap

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)] heap with 1 ≤ i ≤ T.LENGTH index of the node to
         percolate)
Ensure: T[1..i] heap
  j ← PARENT(i)
  while (j > 0) and (T[j] ≺ T[i]) do                  ≺ : ' has less priority than'
      PERMUT(T[i], T[j])
      i ← j
      j ← PARENT(i)
```

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] and T[RIGHT(i)..T.SIZE] are heaps and
         1 ≤ i ≤ T.SIZE index of the node to percolate
Ensure: T[i..T.SIZE] is a heap.
  l ← LEFT(i)
  r ← RIGHT(i)
  if (l ≤ T.SIZE) and (T[i] ≺ T[l]) then              ≺ : 'has less priority than'
      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)
```

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
```

Procedure BUILDHEAP(T)

Require: T.LENGTH ≥ 1
Ensure: T `is a heap.`

```
T.SIZE ← T.LENGTH
for i = [T.SIZE/2] downto 1 do
    PERCOLATEDOWN(T, i)
```

2. Binomial Heap

Procedure BINOMIALLINK(y, z)

```
p(y) ← z
next(y) ← child(z)
child(z) ← y
order(z) ← order(z) + 1
```

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. Search Tree

1. Binary Search Tree

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. Red-Black Tree

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
        Like in the "then" part but swapping "right" and "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
        Like in the "then" part but swapping "right" and "left"
color(x)) ← BLACK
```

III. B-Tree

Function BTreeSearch(x, k)

```
i ← 1
while i ≤ numkeys(x) and k > key(x, i) 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)
```

Building an empty tree (complexity $O(1)$):

Function BTreeCreate(T)

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

The function BTreeSplitChild take an intern node x which is not full and an index i such that $child(x, i)$ is a full son of x. The two nodes are supposed to be in central memory. The function split the son into two nodes and update x in order that x refers to the new son. If the node that should be cut is the root, we first create a new empty root that will be the parent of the old root. The height of a B-Tree is then incremented. This way to insert a key makes B-Tree grow by the root and not by the leaves.

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)
```

Thanks to this function we can write the algorithm that inserts a key k into the tree T by traversing the tree only once downward.

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)
```

BTreeRemove:

1. If the key k is in the node x and if x is a leaf, de-let the key of the leaf.
2. If the key k is in the node x and if x is an intern node
   (a) If the child y preceding k in the node x has at least t keys, find the predecessor k' of k in the sub-tree rooted in y. Delete recursively the key k' in y and replace k by k' in x. Searching and deleting the predecessor can be done in a single pass.
   (b) If y has less than t keys, consider the child z that follows the key k in x. If z has at least t keys, find the successor k' of k in the sub-tree rooted in z. Delete recursively k' and replace k by k'.
   (c) If y and z have both exactly $t-1$ keys, merge k and the keys of z in y such that x lose both the key k and the link to the child z. y contains now $2t-1$ keys. The node z can be destroyed and the deletion of k goes on recursively on y.
3. If the key k is not in the intern node x, find the root $child(x, i)$ of the sub-tree that should contain the key. If $child(x, i)$ has only $t-1$ keys, do one of the two following steps in order to guarantee that it is possible to go down inside a node that has at least t keys. Continue the deletion recursively on the appropriate child of x.
   (a) If $child(x, i)$ has exactly $t-1$ key but an immediate brother that has at least t keys, add a key to $child(x, i)$ by moving down a key of the immediate (left or right) brother of $child(x, i)$ in the node x and by moving the corresponding child of the brother into $child(x, i)$.
   (b) If $child(x, i)$ and one of its two immediate brothers has $t-1$ keys, merge $child(x, i)$ with one of its brothers, which requires to move down one key of x into the median place of $child(x, i)$.