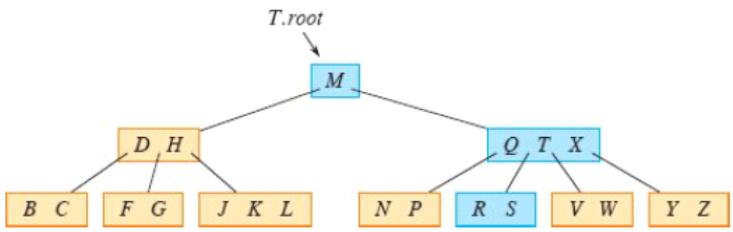
Árboles B (B-trees)

- Árbol de búsqueda balanceado
- Orientado al almacenamiento en memoria secundaría u otros dispositivos
- Busca minimizar los accesos a disco.
- A diferencia de otros tipos de árboles, como los árboles rojinegros, cada nodo pueden tener muchos hijos (incluso miles).

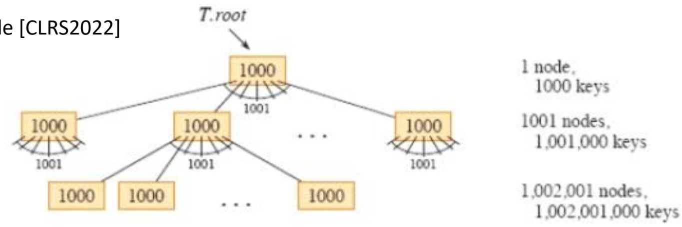


Tomado de [CLRS2022]

Motivación B-trees

- Necesidad de procesar gran cantidad de información (Por ejemplo, historias clínicas, usuarios de facebook, …).
- Dependiendo del tamaño de la información, no se puede tener toda en memoria principal.
- El costo en tiempo de recuperar información de memoria secundaría es alto.
- Se vuelve importante minimizar el número de veces que se accede a memoria secundaria.
- Idea: Crear un árbol donde los nodos sean tan grandes que correspondan a un bloque de memoria, se buscará minimizar la cantidad de nodos que se requieran para hacer las diversas operaciones. Así, se minimiza la cantidad de bloques de memoria a recuperar.

Tomado de [CLRS2022]



Definición B-Trees

Tomado de [CLRS2022]

A ***B-tree*** T is a rooted tree with root $T.root$ having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in monotonically increasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,

   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

   $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}.$$

4. All leaves have the same depth, which is the tree's height h.

5. Nodes have lower and upper bounds on the number of keys they can contain, expressed in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[3]

miércoles, 18 de enero de 2023

Altura de un B-Tree

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

Tomado de [CLRS2022]

Es decir, la altura es O(log_t n) donde t puede ser muy grande.

El máximo número de accesos a memoría sobre un B-tree (inserción, búsqueda, eliminación, mínimo, máximo …) depende de su altura ´por ende son O(log_t n).

miércoles, 18 de enero de 2023

Búsqueda B-Tree

k= 8

3, 7, 10

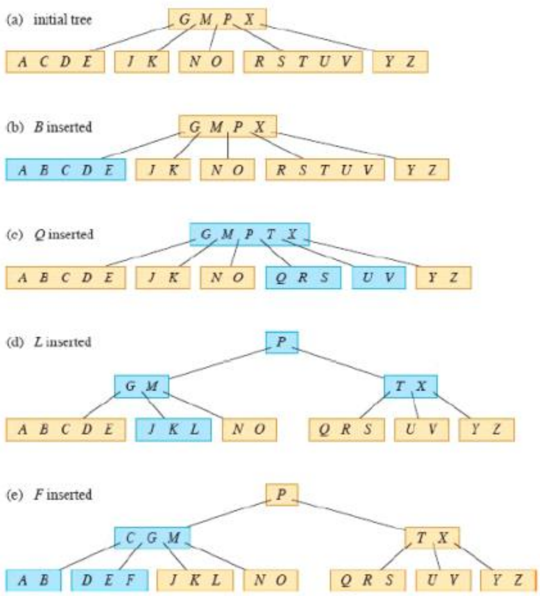
```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Tomado de [CLRS2022]

Accesos O(h)   Uso CPU O(th) Donde h es O(log_t n)

Inserción B-Tree

- Un nuevo elemento se agrega en un nodo hoja (el que corresponda según su valor)
- Si el nodo hoja está completo entonces se divide (split) en dos nodos hoja, donde la mediana del nodo completo pasa a insertarse al nodo padre y el elemento nuevo pasa a insertarse en uno de los dos nodos hoja.

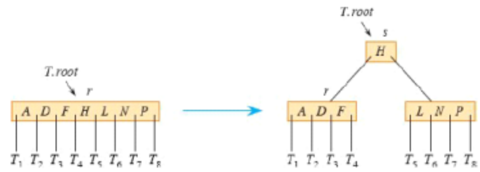


Tomado de [CLRS2022]

Inserción B-Tree (Algoritmos)

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
    s = B-TREE-SPLIT-ROOT(T)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-SPLIT-ROOT(T)
s = ALLOCATE-NODE()
s.leaf = FALSE
s.n = 0
s.c_1 = T.root
T.root = s
B-TREE-SPLIT-CHILD(s, 1)
return s
```

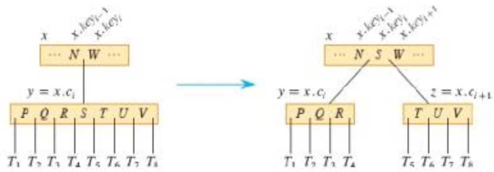



```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf                               // inserting into a leaf?
    while i ≥ 1 and k < x.key_i         // shift keys in x to make room for k
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k                     // insert key k in x
    x.n = x.n + 1                       // now x has 1 more key
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i        // find the child where k belongs
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1                // split the child if it's full
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i                  // does k go into x.c_i or x.c_{i+1}?
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

Tomado de [CLRS2022]

```
B-TREE-SPLIT-CHILD(x, i)
y = x.c_i                               // full node to split
z = ALLOCATE-NODE()                     // z will take half of y
z.leaf = y.leaf
z.n = t − 1
for j = 1 to t − 1                      // z gets y's greatest keys ...
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t                      // ... and its corresponding children
        z.c_j = y.c_{j+t}
y.n = t − 1                             // y keeps t − 1 keys
for j = x.n + 1 downto i + 1            // shift x's children to the right ...
    x.c_{j+1} = x.c_j
x.c_{i+1} = z                           // ... to make room for z as a child
for j = x.n downto i                    // shift the corresponding keys in x
    x.key_{j+1} = x.key_j
x.key_i = y.key_t                       // insert y's median key
x.n = x.n + 1                           // x has gained a child
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```



Ejercicio

Inserte los siguientes elementos en un árbol b vacío (con mínimo grado 2)
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E