B-Trees

B-trees are balanced search trees designed to work well on magnetic disks or direct-access secondary storage devices. B-trees differ significantly from red-black trees in that B-tree nodes may have many children, from a handful to thousands.

It often takes more time to access a page of information and read it from a disk than it takes for the computer to examine all the information read. For this reason, we look at the two principal components of the running time:

- the number of disk accesses, and
- the CPU time.

We model these disk operations in our pseudo-code as follows. Let x be a pointer to an object. DISK-READ(x) to read object x into main memory; DISK-WRITE(x) to write object x to the disk.

1 Definition

A B-tree is a rooted tree having the following properties.

1. Every node x has the following fields:
    - $n[x]$, the number of keys currently stored in node x
    - the $n[x]$ keys themselves, stored in nondecreasing order: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$, and

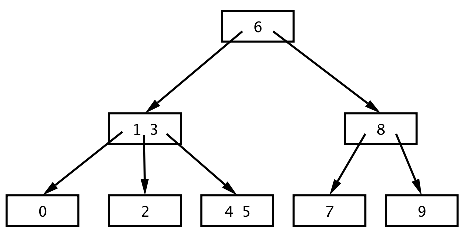

Figure 1: B-Tree where $t = 2$ and the sequence 9,0,8,1,7,2,6,3,5,4 inserted.

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

2. If x is an internal node, it also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

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

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

4. Every leaf has the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:

   - 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 non-empty, the root must have at least one key.
   - Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children. We say that a node is **full** if it contains exactly $2t-1$ keys.

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

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

2 Operations

The operations for B-trees include B-Tree-Search, B-Tree-Create, B-Tree-Insert, B-Tree-Delete, etc. We assume that:

- the root of the B-tree is always in main memory.
- any nodes that are passed as parameters must already have had a Disk-Read operation performed on them.

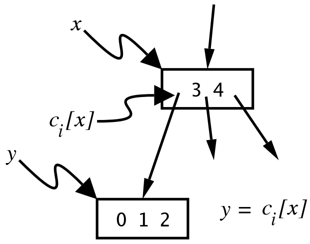

Figure 2: The arguments for the B-TREE-SPLIT-CHILD algorithm.

2.1 Search

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

2.2 Creation

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE()
leaf[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
root[T] ← x
```

2.3 Insertion

A fundamental operation used during insertion is the **splitting** of a full node y (having $2t-1$ keys) around its **median** key $key_t[y]$ into two nodes having $t-1$ keys each. The median key moves up into y's parent, which must be nonfull prior to the splitting of y.

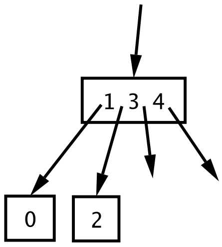

Figure 3: The result of child splitting.

```
B-Tree-Split-Child(x, i, y)
z ← Allocate-Node()
leaf[z] ← leaf[y]
n[z] ← t − 1
for j ← 1 to t − 1
     do key_j[z] ← key_{j+t}[y]
if not leaf[y]
   then for j ← 1 to t
             do c_j[z] ← c_{j+t}[y]
n[y] ← t − 1
 for j ← n[x] + 1 downto i + 1
      do c_{j+1}[x] ← c_j[x]
 c_{i+1}[x] ← z
 for j ← n[x] downto i
      do key_{j+1}[x] ← key_j[x]
 key_i[x] ← key_t[y]
 n[x] ← n[x] + 1
 Disk-Write(y)
 Disk-Write(z)
 Disk-Write(x)
```

Inserting a key k into a B-tree T of height h is done in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$.

Figure 4: See Fig 18.6 (page 446) – create a new node for root if root is $2t-1$

```
B-Tree-Insert(T, k)
r ← root[T]
if n[r] = 2t − 1
   then s ← Allocate-Node()
        root[T] ← s
        leaf[s] ← FALSE
        n[s] ← 0
        c_1[s] ← r
        B-Tree-Split-Child(s, 1, r)
        B-Tree-Insert-Nonfull(s, k)
   else B-Tree-Insert-Nonfull(r, k)
```

```
B-Tree-Insert-Nonfull(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < key_i[x]
              do key_{i+1}[x] ← key_i[x]
                 i ← i − 1
        key_{i+1}[x] ← k
        n[x] ← n[x] + 1
        Disk-Write(x)
   else while i ≥ 1 and k < key_i[x]
              do i ← i − 1
          i ← i + 1
          Disk-Read(c_i[x])
          if n[c_i[x]] = 2t − 1
             then B-Tree-Split-Child(x, i, c_i[x])
                  if k > key_i[x]
                     then i ← i + 1
          B-Tree-Insert-Nonfull(c_i[x], k)
```

2.4 Deletion

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node , do the following:
   - If the child y that precedes k in node x and x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x.

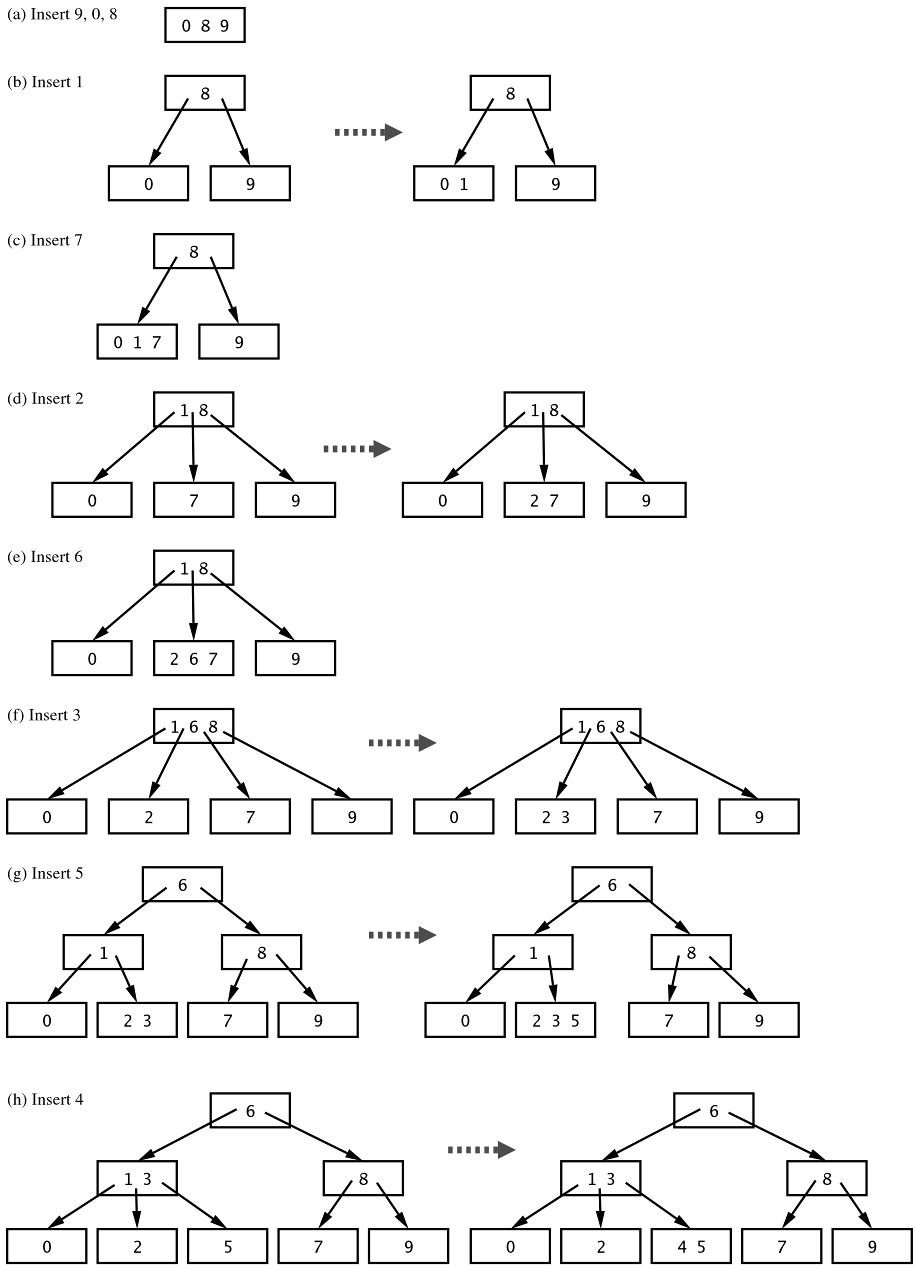

Figure 5: Inserting the sequence 9,0,8,1,7,2,6,3,5,4 into a B-Tree.

- Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x.
- Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.

   (a) If $c_i[x]$ has only $t-1$ keys but has a sibling with t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child from the sibling into $c_i[x]$.

   (b) If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys, merge c_i with one sibling., which involves moving a key from x down into the new merged node to become the median key from that node.