B-Trees. [Cormen-Leiserson-Rivest]

1. Search trees designed to minimize IO operations to secondary memory. When database is too large to fit in main memory, some parts will be stored in disk. A single access to disk can be 10^3 to 10^5 times slower than access to memory.
   (Disks rotate at about 7200 RPM; typical range 5K-15K RPM. One rotation takes 8.33 ms, which is about 5 orders slower than a 100 nano sec access for current silicon memory.)

2. In order to amortize the disk access cost, store and fetch in large chunk, instead of single items. Information is divided into large, equal-sized "pages" that are laid out consecutively within each cylinder. Typical page size: 2^{11} to 2^{14} bytes (2K-16K). Often, it takes longer to read one page of information than to examine it (compute). Thus, when dealing with disk-bound data structures, we look at two factors separately:

   a. number of disk accesses,
   b. the CPU time.

3. B-Tree algorithms operate at the granularity of pages. I.e., the unit operations are to READ or WRITE a page. The main memory can only accomodate only so many pages, so older pages will be flushed out as new ones are fetched.

4. Since we want to optimize the number of page accesses, we will choose the size of the B-Tree node to match the page size. That is, each node will store keys for about 50-2000 items, and will have the similar branching factor.
   As an example, with a branching factor of 1001 (each node with 1000 keys), 1 billion keys can be accessed by a tree of height 2. Just 2 disk accesses!

Figure:

5. Definition of a B-Tree.
   A B-Tree is a rooted tree with the following properties:

   1. Every node x has:
      a. n(x): the number of keys stored at x
      b. the n(x) keys themselves sorted, $key_1[x] <= key_2[x] <=...$
      c. leaf(x) boolean, which is true if x is a leaf.

   2. Each non-leaf node contains n(x)+1 pointers, $c_1(x)$, $c_2(x)$, ... to the children of x.

   3. The keys $k_i(x)$ separate the ranges of keys stored in each subtree; suppose k_i is any key stored in the subtree rooted at c_i, then

      $k_1 <= key_1(x) <= k_2 <= key_2(x) <= k_3 .... <= k_{n(x)+1}.$

4. All leaves have the same depth, equal to B-tree's height.

5. There are lower and upper bounds on how many keys a node contains.
   These bounds are expressed by a parameter $t \geq 2$.
   a. Every node except root must have at least $t-1$ keys.
      Every internal node, therefore, has at least t children;
      the root, if non-empty, has at least one key.

   b. Every node has at most $2t-1$ keys. Thus, an internal node has
      at most $2t$ children.
      A node is called FULL if it has $2t-1$ keys.

* The simplest form of B-Tree has $t=2$, which is a 2-3-4 tree.

EXAMPLE:

6. Thm. A B-Tree with n-keys and min deg $t \geq 2$ has height $H \leq \log_t (n+1)/2$.

depth	number of nodes
0	1
1	2
2	$2t$
3	$2t^2$....

(Number of nodes times keys per node).
We get $n \geq 1 + (t-1) \sum_{i=1}^h (2 t^{i-1})$

7. Searching a B-Tree.
   The root of B-Tree always in main memory, so no disk-access
   required there. But if root node is changed, then disk-write
   must be done.

```
B-TREE-SEARCH (x, k)    //search for key k

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

FIG. 1.
search for R.

8. INSERTING into a B-Tree.

Inserting into a B-Tree is more complex. As in binary search trees, we search for the leaf position where to insert, but we simply can't just add a new node---B-Tree requires that all leaves be at the same level, and each node have between t-1 and 2t-1 keys.

We insert k into an existing node. The only problem arises when that node is already full. In this case, we SPLIT the node (with 2t-1 keys) around its t-th key (the median); the median key moves up to parent, and two new children, each with t-1 keys are formed. If the parent was also full, then this splitting step recursively continues up the tree.

In order to perform insert in a single pass, on our way down the search path, we split each full node in preparation for the insert.

```
B-Tree-Split-Child (x, i, y)

z <-= allocateNode();
leaf(z) <- leaf(y);  n(z) <- t-1
for j = 1  to  t-1
       key_j(z) <- key_j+t(y)
if not leaf(y), then
       for j = 1  to  t-1
       c_j(z) <- c_j+t(y)
n(y) <- t-1;
for j = n(x)+1 down to i+1
      c_j+1(x) <- c_j(x)
c_i+1(x) <- z;
for j <- n(x) downto i
       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);
```

Example. Figure 18.5

9. Description:

y is the ith child of x, and is the node being split.
Node y originally has 2t children (and 2t-1 keys), but is reduced to t children (and t-1 keys) by this operation. Node z "adopts" the t largest children of y, and z becomes a new child of x, positioned just after y in x's table. The median key of y moves up to become the key in x that separates y and z.

10. B-TREE-INSERT (T, k)

```
r <- root(T)
if n(r) = 2t-1 then
      s <- allocateNode();
      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)
```

Fig. 18.6

11. 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--
   key_i+1 (x) <- k
   n(x) <- n(x) + 1
   diskWrite(x)
else
   while i >= 1 and k < key_i (x) do
      i--
   i++;
   diskRead(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++
   B-TREE-INSERT-NONFULL (c_i(x), k)
```

end.

12.
Description. First while loop handles the case when x is a leaf.
When x is not a leaf, then insert k into appropriate leaf node in the subtree rooted at x. Second while loop determines the child of x to which recursion descends. The if condition checks if that child is a fullNode or not. If full, the B_TREE_SPLIT splits that into two non-full nodes, and the next if determines which of the children to descend to.

Example: Figure 18.7

Analysis. The number of disk accesses by B-TREE-INSERT is O(h), since only O(1) disk read or writes between calls to B-TREE-NONFULL. The total CPU time is O(th) = O(t log_t n).

13. DELETING a key from B-Tree.

The key may be deleted from any node, not just a leaf. We need to make sure a node doesn't get too small after a deletion. So, if a node has t-1 keys and one of them is deleted, we need to fix it.

Suppose we need to delete k from subtree rooted at x. The proc is structured to ensure that when B-TREE-DELETE is called on a node x, the number of keys in x is at least t. This allows us to perform deletion in one pass, without backing up.

1. If k is in leaf-node x, delete k from x.
2. If k is in non-leaf node x, do:
   a. if the child y that precedes k in node x has t or more keys, then find predecessor k' of k in subtree rooted at y. Recursively delete k', and replace k with k' in x.

   b. Symmetrically, if child z that follows k in node x has >= t keys, find the successor k' of k in subtree rooted at z. Recursively delete k', and replace k with k' in x.

   c. 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 has 2t-1 keys. Free z and recursively delete k from y.

3. If the key k is not present in node x, determine the root c_i(x) of the appropriate subtree tht contains k. If c_i(x) has

only t-1 keys, execute steps 3a or 3b to guarantee we descend to a node with >= t keys. Then finish by recursing on appropriate child of x.

a. If c_i(x) has only t-1 keys but has an immediate sibling with t or more keys, give c_i(x) an extra key by moving a key from x down to c_i(x), moving a key from c_i(x)'s immediate left or right sibling up into x, and moving the appropriate child pointer 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(x) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Figure: 18.8

Since more of the keys in Btree are in the leaves, we expect most delete operations occur at leaves.

Like insertion, deletion also has cost O(t log_t n).