CMPS 340 File Processing
B-trees: Definition of and Algorithms for Insertion/Deletion

A **B-tree** is a rooted tree in which each node contains a sequence of keys. (With each key there may be a collection of associated information.) To say that a B-tree has order m means that

(a) No node contains more than $m-1$ keys.

(b) No node, except possibly the root, contains fewer than $\lceil \frac{m}{2} \rceil - 1$ keys. The root may have as few as one key.

(c) Each non-leaf node has exactly one more children than keys.

(d) All leaves are equidistant from the root.

(e) Letting the keys in a node be $x_1, x_2, \cdots, x_r$ (where $x_1 < x_2 < \cdots < x_r$) and letting the subtrees rooted at its children be $s_0, s_1, \cdots, s_r$, respectively, it is the case that, for all i satisfying $0 < i \leq r$, all the keys in s_{i-1} are less than x_i and all the keys in s_i are greater than x_i. (If duplicate keys are allowed, we have to modify these conditions.)

From *(b)* and *(c)*, it follows that every non-leaf node, except possibly the root, has at least $\lceil \frac{m}{2} \rceil$ children but no more than m children.

Below we describe algorithms for the insertion and deletion of a key into/from a B-tree of order m. To simplify the description, we assume that each node is (temporarily) capable of holding up to m keys and $m+1$ pointers to children. A node containing m keys is said to be *overflowing*. A non-root node containing fewer than $\lceil \frac{m}{2} \rceil - 1$ keys is said to be *underflowing*. During the insertion process, a node may be temporarily overflowing. During the deletion process, a node may be temporarily underflowing. A node is said to be *full* if it contains the maximum allowable number of keys ($m-1$). A node is said to be *on the verge of underflowing* if it contains the minimum allowable number of keys ($\lceil \frac{m}{2} \rceil - 1$). The functions `Left_Sibling_of()`, `Right_Sibling_of()`, and `Parent_of()` have the obvious meanings.

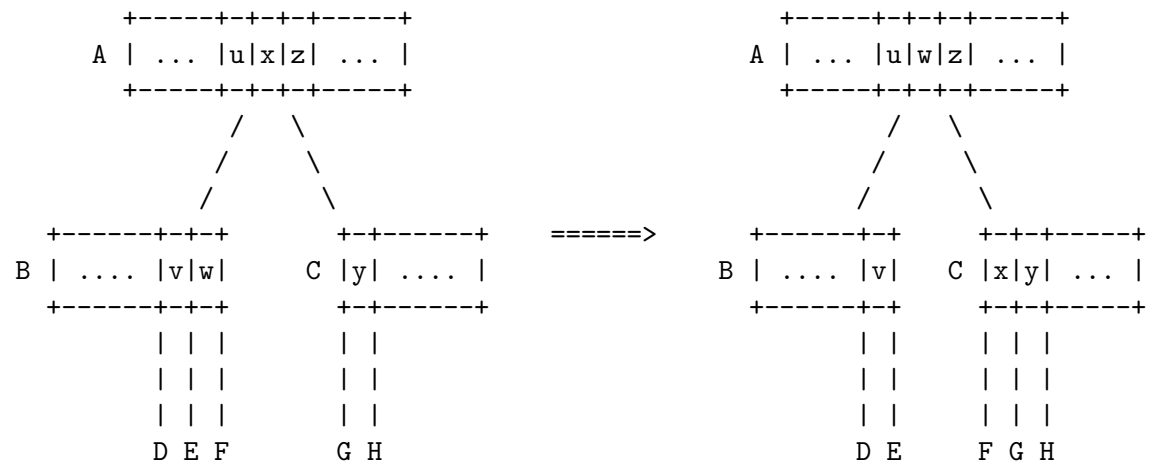
Note: In any of the places where either of the two `Redistribute` procedures are called, it may be beneficial to move as many keys as necessary out of the "giving" node and into the "receiving" node so as to result in the two of them having (as close as possible to) an equal number of keys. To accomplish this, simply nest the call to the appropriate `Redistribute` procedure inside a loop that iterates an appropriate number of times. **End of note.**

The subprograms `LR-Redistribute()`, `RL-Redistribute()`, `Split()`, and `Concatenate()` are described via the pictures appearing below. Each one takes a node as its lone parameter. In the cases of `LR-Redistribute()` and `RL-Redistribute()`, the parameter corresponds to the node that is giving away one of its keys. In the case of `Split()`, the parameter corresponds to the node that gets split into two nodes. As for `Concatenate()`, the parameter corresponds to the sibling on the left among the two adjacent siblings that are combined into a single node.

```
Insert(z):
  BEGIN
    find leaf node M where z "belongs";
    place z into proper place within M;
    Adjust(M);
  END Insert;

Delete(z):
  BEGIN
    find node M where z "belongs";
    IF z is not in M THEN
       do nothing (or report error, if appropriate);
    ELSE
       IF M is a leaf THEN
          remove z from M;
          Adjust(M);
       ELSE
          find N, the leftmost leaf in the right subtree of z;
          let z' be the smallest key in N;
          remove z' from N;
          replace z in M by z';
          Adjust(N);
       ENDIF
    ENDIF
  END Delete;
```

```
Adjust(M):
  BEGIN
    IF M is overflowing THEN
      IF Right_Sibling_of(M) exists and is not full THEN
        LR-Redistribute(M);
      ELSIF Left_Sibling_of(M) exists and is not full THEN
        RL-Redistribute(M);
      ELSE --all of M's immediate siblings are full
        Split(M);
        Adjust(Parent_of(M));
      ENDIF;

    ELSIF M is underflowing THEN
      IF Left_Sibling_of(M) exists and is not on verge of underflowing THEN
        LR-Redistribute(Left_Sibling_of(M));
      ELSIF Right_Sibling_of(M) exists and is not on verge of underflowing THEN
        RL-Redistribute(Right_Sibling_of(M));
      ELSIF M is the root THEN
        IF M has only one child THEN
          child of M becomes root;
          dispose of M;
        ENDIF;
      ELSE
        --To have arrived here, M must be underflowing, M must have
        --at least one sibling, and all of M's immediate siblings are on
        --the verge of underflowing.  It follows that concatenation
        --(or "merging", if you prefer) is called for
        IF Right_Sibling_of(M) exists THEN
          Concatenate(M);
        ELSE --left sibling must exist
          Concatenate(Left_Sibling_of(M));
        ENDIF;
      ENDIF;

    ELSE --M is neither overflowing nor underflowing
      do nothing;
    ENDIF;
  END Adjust;
```

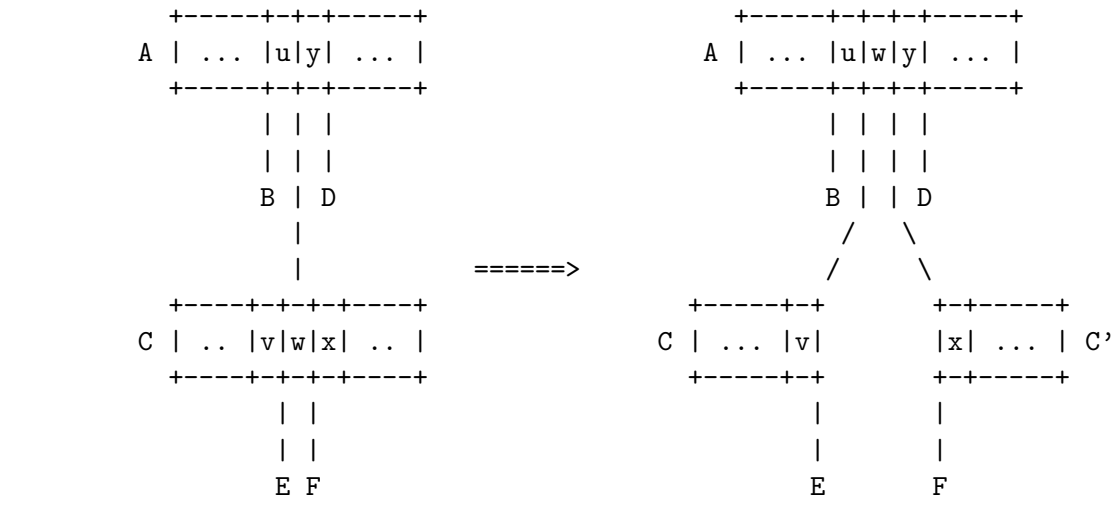
LR-Redistribute(B):



In the pictures, capital letters (e.g., A, B, etc.) refer to nodes and small letters (u,v, etc.) refer to keys.

RL-Redistribute(C): Reverse the direction of the arrow in the picture above.

Split(C):



Concatenate(C): Reverse the direction of the arrow in the picture above.