# Red-Black Trees : Shared Memory de-duplication

## Implement

A class BRtree was built for handling all tree-related stuff including attributes and functions.
For each tree object we have:

```
        BRnode* root;              (the root)
        BRnode* leavesBlack;       (the leaves)
```

Functions we need to insert and delete nodes to a red-black tree:

```
        void insert_br(int, int);
        void insert_fixup_br(BRnode*);
        void right_rotate_br(BRnode*);
        void left_rotate_br(BRnode*);
        void TRANSPLANT_RB(BRnode*, BRnode*);
        void DELETE_RB(BRnode*);
        void DELETE_FIXUP_RB(BRnode*);
        BRnode* MINIMUM_TREE(BRnode*);
```

Functions to search nodes in trees:

```
        BRnode* find(int);
        BRnode* find_without_key(int);
        BRnode* recursive_find(BRnode*, int);
```

Other functions were also built for simulating the algorithm:

```
        void Load(nodePage*, int, BRtree);
        void Update(nodePage*, int, BRtree);
```

A few list of data [ <hash of page content>, <page id>] will be generated to simulate pages. The hash values will be the keys for red-black tree. It also describes the content of page, so we know a page has changed if we see a different hash value with the same page id.

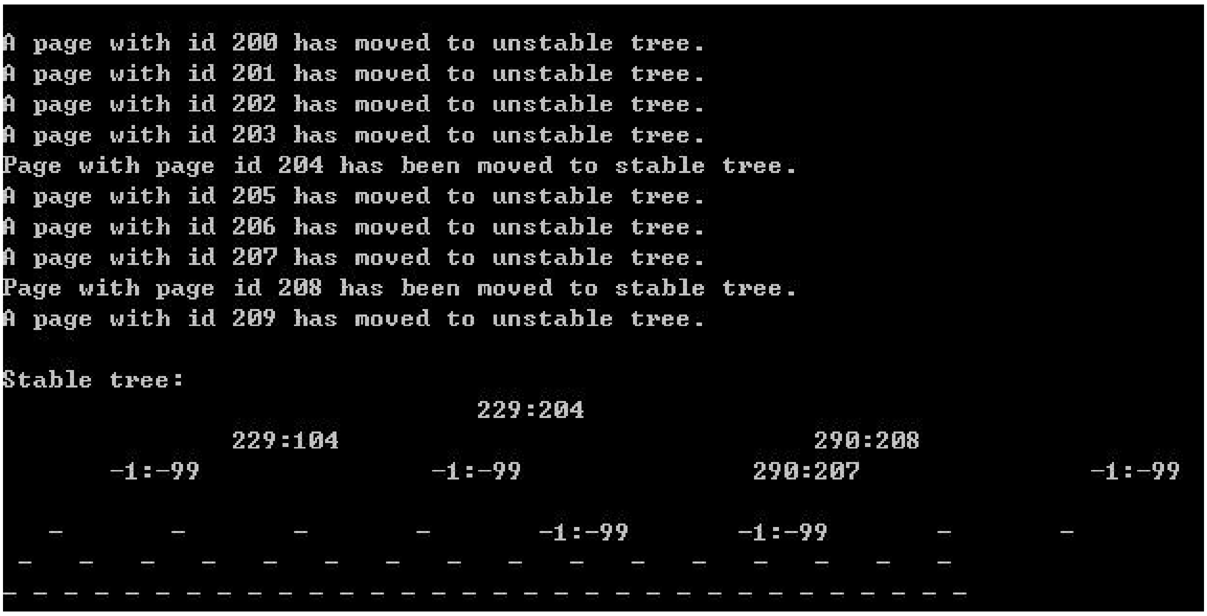
## Time complexity

For insert and delete, it takes O(log n) to traverse to the appropriate position, and O(log n) to fix up. Recolor, rotate and transplant all cost O(1). So it’s O(log n)
For search, it takes O(log n) to search by hash because hash is the key of BST, and O(n) to search “by id” because all nodes are needed to go through.
Load() calls insert functions m times, so the time is O(log n) for one data, and O(m log n) for many data, where m is the size of data and n is the size of the tree.
Deduplicate() calls find(), DELETE_RB() and insert_br(), where their times are O(log n) and O(log n), so the time is O(log n).
Update() calls find_without_key(), DELETE_RB(), insert_br() and Deduplicate(), where their times are O(n), O(log n), O(log n) and O(log n), so the combined time should be O(n). For Update() to process m size data, it takes O(mn).

| (tree size n) | single data | data size m |
|---|---|---|
| Load() | O(log n) | O(m log n) |
| Update() | O(n) | O(mn) |
| Deduplicate() | O(log n) | O(m log n) |

## Example

The sample runs were already included in the cpp file.
3 sets of data are created.
The first set is only for testing Load(), which only loads data of 5 pages into the unstable tree.
The second set (10 pages) is for testing Update() and Deduplicate(). One of them intends to have the same hash value as one page from the first set, and another two have same hash value with each other’s. These two pairs show the function of merging (the will be moved to stable tree for merging).
The third set (5 pages) is for testing Update(). One of them intends to have the same page id as one of pairs mentioned above. This is for testing the function of moving changed pages to unstable tree.
After each function (Load and Update), the trees will show for checking in detail.



```
A page with id 200 has moved to unstable tree.
A page with id 201 has moved to unstable tree.
A page with id 202 has moved to unstable tree.
A page with id 203 has moved to unstable tree.
Page with page id 204 has been moved to stable tree.
A page with id 205 has moved to unstable tree.
A page with id 206 has moved to unstable tree.
A page with id 207 has moved to unstable tree.
Page with page id 208 has been moved to stable tree.
A page with id 209 has moved to unstable tree.

Stable tree:
                               229:204
               229:104                               290:208
       -1:-99         -1:-99              290:207               -1:-99

   -       -       -       -       -1:-99       -1:-99       -       -
 -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```