COP 5536 Spring 2019 Programming Project Report

Due Date: Apr14th, 2019, 11:59pm EST

Xinghua Pan*
Instructor: Dr. Sartaj K Sahni

April 14, 2019

1 Overview

This project implemented the B+ tree using C++11. According to the lecture, the B+ tree node is an array partitioned into indexed red-black tree node, this has to maintain two array, one for dictionary pair, one for index. This is not space effective and with out any computing benefits. So in this implementation, an indexed red-black tree (with the idea of left size of each red black tree node) is implemented to support the B+ tree node structure.

1.1 File structure

The file organizations is very simple. The B+ tree implementation is in the `bp_tree.h`, and the red-black tree implementation is in the `rb_tree.h`. In the `main.cc`, simply read the input file line by line, call the B+ tree function accordingly.

There are two files, `rb_main.cc` and `bp_main.cc`, run a test for red black tree and B+ tree.

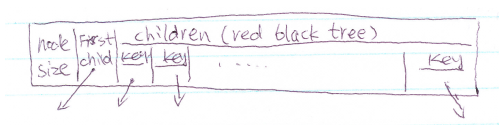
Documentation generated by **Doxygen** is in the directory `doc`.

2 B Plus Tree

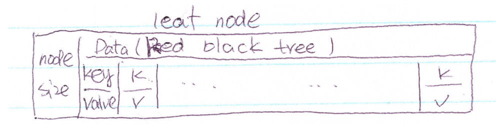
B+ plus tree has two type of nodes, index(or internal) node and leaf node. Both types of nodes have a data member `node_size`, which tracking the number of elements in the node. The leaf node have a data member, `data`, which is a red-black tree, in which are the red black tree node stored the real dictionary pair, in this project, an [int, float] pair. The index node also have a red black tree in its node, `children`, but the red-black tree node store the dictionary pair of key and pointer of its children (b+ tree node), i.e. [int, bp node*], also, there is a pointer (`first_child`) to a b+ tree node as its first child.

*xi.pan@ufl.edu, UFID: 95160902

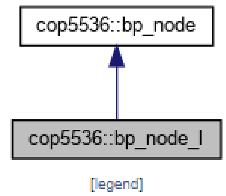
Figure 1: The structures of B+ tree node

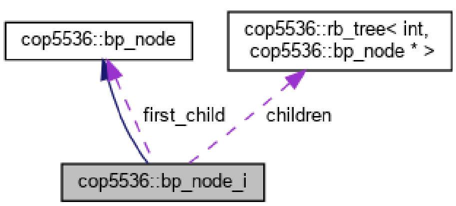


(a) index node

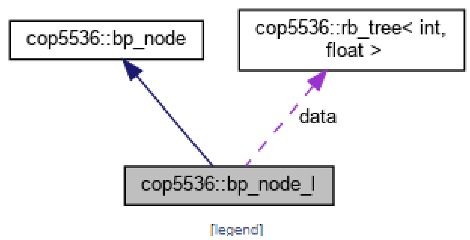


(b) leaf node





(c) index node



(d) leaf node

Note that in this implementation, the leaf node are not doubly linked to each other.

The B+ tree support three operations, **Insert, Search/Search_In_Range, Delete**, as the assignment required.

There is a helper function **split**, which call the red black tree's split function and return a new index node.

The **Search** function is simple, if the searched node is leaf, search in its red black tree, if it is a index node, search for the child with greatest but smaller than the search key, this function is implemented into the red-black tree **search_eq_or_gt** function, with the returned pointer, recursively search the child.

The **Search_In_Range** is simple, too, using the **search_range** function in the red-black tree. If the node is leaf, search its red-black tree; if its an index node, find all the children, recursively all the children.

For **Insert** and **Delete**, if the node is full or empty, it need to split or merge with its sibling, this need some tracking mechanism (using stack or parent pointer) from the inserted or deleted node back to the root. But in this implementation, the c++ *future* is used for parent to get the result from its child node. Depending on the result of child's insert or delete, the parent decide to split and return new index node to its own parent, or merge its children and let its own parent know a deficient node need to take care, all the decision making job back to the root, and the B+ tree's height increase or decrease respectively.

	bp_node ()
	bp_node (unsigned short ns)
virtual	**~bp_node** ()
virtual void	**destory_** ()=0
virtual void	**print** ()=0
virtual bool	**is_leaf** ()=0
bool	**is_full** (unsigned short degree)
virtual float	**search** (int key)=0
virtual std::vector< float > *	**search** (int low, int high)=0
virtual bool	**remove** (int key)=0
virtual bool	**splitable** ()
virtual **rb_tree_node**< int, **bp_node** * > *	**split** (bool is_left)=0
virtual **rb_tree_node**< int, **bp_node** * > *	**insert_** (int key, float value, unsigned short degree)=0

Public Attributes

unsigned short	**node_size**

(a) functions of b+ tree node

2.1 Insert

All the data dictionary pair need to inserted to the leaf node, so the insert algorithm as follow, from B+ tree's root, if it is a leaf node, insert the dictionary pair into its red black tree, if it is a index node, search the red black tree by the "great or equal to", which return the child which key is less or equal to the search key. If the return key greater than the search key, then search the first child, or search the child node given by the returned pointer. Note that, to implement this algorithm, if the node is leaf, the code of first *if* between line 1 and 7 will goes into leaf node's insert function, and the *else* part will go to index node's insert function. So as the **Delete** algorithm.

Algorithm 0: B+ tree insert algorithm

Input: B+ tree's node, (key, value)
Output: new (key, b plus tree node) pair or nullptr

```
if node is a leaf then
    // insert dictionary into leaf node
    insert(key, value);
    node_size++;
    if node is full then
        return node.split();
    else
        return nullptr;
else
    // insert dictionary into index node
    child_node ← search_eq_or_gt(key);
    if child_node.key > key then
        return_node ← insert(node's first child, key, value);
    else
        return_node ← insert(child_node, key, value);
    if return_node is not nullptr then
        insert return_node to the children red black tree;
        if node is full then
            return node.split()
    return nullptr
```

2.2 Delete

Since all the data dictionary are in the leaf node, the delete operation should go to leaf node too. To keep the leaf node at the same level of the tree, if the node is empty after delete, there is a "borrow" mechanism to keep the tree's height as "long" as possible. If there is a deletion on a index node, there must be the root node. To keep this in mind, observe that, for index node and leaf node, there are different definition of empty node. For a leaf node, if the inside red black tree is empty, the leaf node is empty too, we can delete the lead node by its parent; but for a index node, if the inside red black tree is empty, we call the index node is empty, but there is a "first child" pointer still point to its child, we can not delete the node now. The empty index node will be the deficient index node, and should merged with its sibling by its parent node.

Algorithm 1: B+ tree delete algorithm

```
Input: B+ tree's node, key
Output: bool value indicate child is dificient or not after deletion
if node is a leaf then
    delete the key in the red black tree;
    if red black is empty then
        return true
    else
        return false
else
    child_node ← search_eq_or_gt(key);
    if child_node.key > key then
        child_deficient ← delete(node's first child, key);
    else
        child_deficient ← delete(child_node, key);
    if child_deficient then
        if sibling can split then
            split the sibling, insert the split element into its child;
        else if can merge with other then
            merge with sibling;
        else
            // no child can split or can be merged
            return true
    return false
```

3 Red Black Tree

The red black tree implemented all the operation described in the lecture. C++ template is used to support both type of B+ tree node.

The function `nth` support access the red black tree nodes by index, and the data member `left_size` is help to tracking the number of nodes in its left child.

The function `insert_, delete_key` support the insert and delete operation, and `fix_insert_` and `fix_delete_` is called when the tree have two consecutive red nodes or a sub-tree become deficient. The function `left_rotate` and `right_rotate` is called when the fix operation need to rotate the sub-tree.

The function `split_` and `join_`, respectively split the tree into two trees or join two trees into one, are implemented with the help of data member `rank`.

key_type	key
data_type	data
node_color	color
short int	left_size
short int	rank
Public Attributes inherited from cop5536::rb_base_< rb_tree_node< key_type, data_type > >	
rb_tree_node< key_type, data_type > *	left
rb_tree_node< key_type, data_type > *	right
rb_tree_node< key_type, data_type > *	parent

(a) data structure of red black tree node

	rb_tree ()
	rb_tree (node_type< key_type, data_type > *o)
	rb_tree (const rb_tree &o)
virtual	~rb_tree ()
bool	insert (key_type key, data_type data)
void	print_tree ()
void	print_node (node_type< key_type, data_type > *node_, int space)
void	get_keys (std::vector< key_type > &vec)
void	get_keys__ (node_type< key_type, data_type > *node_, std::vector< key_type > &vec)
bool	delete_key (key_type key)
node_type< key_type, data_type > *	nth (short int n)
void	join_ (node_type< key_type, data_type > *m_, rb_tree< key_type, data_type > *b_)
short int	get_size_of (node_type< key_type, data_type > *node_)
node_type< key_type, data_type > *	split_ (short int n_, rb_tree< key_type, data_type > *b_)
void	split_ (node_type< key_type, data_type > *node_, rb_tree< key_type, data_type > *b_)
data_type	search (key_type key)
node_type< key_type, data_type > *	search_eq_or_gt (key_type key)
std::vector< node_type< key_type, data_type > * > *	search_range (key_type low, key_type high)
bool	insert_ (key_type key, data_type data)
bool	insert_ (node_type< key_type, data_type > *inode)
node_type< key_type, data_type > *	sibling_of_ (node_type< key_type, data_type > *node_)
node_type< key_type, data_type > *	next_ (node_type< key_type, data_type > *node_)
node_type< key_type, data_type > *	prev_ (node_type< key_type, data_type > *node_)
void	cut_out_ (node_type< key_type, data_type > *cnode, node_type< key_type, data_type > *child_of_)
void	swap_parent_ (node_type< key_type, data_type > *old_node, node_type< key_type, data_type > *new_node)
void	left_rotate_ (node_type< key_type, data_type > *lnode)
void	right_rotate_ (node_type< key_type, data_type > *rnode)
virtual void	copy_data_ (node_type< key_type, data_type > *from, node_type< key_type, data_type > *to)
virtual node_type< key_type, data_type > *	search_ (key_type key)
virtual bool	delete_ (key_type key)
void	destory_ (node_type< key_type, data_type > *node_)
void	update_left_size_ (node_type< key_type, data_type > *node_, int sz=1)
void	fix_insert_ (node_type< key_type, data_type > *node_)
bool	is_red_ (node_type< key_type, data_type > *node_)
void	fix_delete_ (node_type< key_type, data_type > *dnode, node_type< key_type, data_type > *pod, node_type< key_type, data_type > *cod)
void	search_range_ (std::vector< node_type< key_type, data_type > * > *rs, node_type< key_type, data_type > *node_, key_type low, key_type high)
void	split__ (node_type< key_type, data_type > *node_, bool was_left, rb_tree< key_type, data_type > *s_tree, rb_tree< key_type, data_type > *b_tree)

(b) functions of red black tree