Brian Newsom

CSCI 2270 - Gabe Johnson

Shang Ma

DESIGN DOCUMENT

Info

1. node is a B-tree node that contains information about keys and children.
2. keys are values used to keep data in order. For our assignment a key is just an integer, but in practice they could be any number of data types. They are used to divide the tree into subtrees.
3. children are references to other nodes. For this assignment they are simply in-memory pointers. In a real situation they might be disk addresses.
4. root: this is the top-most node of a B-Tree. Unlike the binary trees we have seen, this may change due to inserting or removing data. We make exceptions for the root node.
5. index nodes: these are nodes that are not leaves and not root. It is called an *index* because its purpose is to guide us to the next layer down.
6. leaves: these are the nodes at the *bottom* of the tree. They do not have children.
7. order: the order of a B-Tree is the maximum number of children a non-leaf may have. This is commonly denoted as m for some reason.

For this assignment we will adhere to the following invariants:

1. A node's keys are kept in ascending order, starting at index 0.
2. A node may have at most m children.
3. Index nodes have at least round\_up(m/2) children.
4. If the root is not a leaf, it has at least two children.
5. All leaves are at the same level.
6. The node pointed to by child[i] holds keys that are less than key[i]. The node pointed to by the final child pointer holds keys that are larger than the final key.
7. Leaves must have at least round\_up(m/2) - 1 keys.

// -------------------------------------------- I N S E R T --------------------------------------------------------------\\

void insert(btree\* &root, int key);

// Insert the given key into a b-tree rooted at 'root'. If the key is

// already contained in the btree this should do nothing.

INPUT: address of the root node, the key being inserted

If the root node is NULL

Create new btree with key

return

If the key is already contained (using function)

Return

If a leaf node

If empty slots

Add\_key(&root, key)

Full

If parent is full of children

Add\_height(&root, key)

If previously no parent

Point to new root of btree

Change is\_leaf to true for new leaves

return

If parent && full of keys

If appropriate child node exists

Recurse to appropriate child

If no appropriate child && not full of children

Create child

return

RETURN: Nothing

On exit:

-- the 'root' pointer should refer to the root of the

--tree. (the root may change when we insert or delete)

-- the btree pointed to by 'root' is valid.

// -------------------------------------------- R E M O V E -------------------------------------------------------------\\

//

// Remove the given key from a b-tree rooted at 'root'. If the key is

// not in the btree this should do nothing.

If contains is false

Return

If leaf node

Search value

Delete node, reassign array

Rebalance(node\* &root)

Else if internal node

Pick replacement node to from leftmost element of right tree

Key = replacement node

Remove(replacement node)

RETURN: NOTHING

// On exit:

// -- the 'root' pointer should refer to the root of the

// tree. (the root may change when we insert or delete)

// -- the btree pointed to by 'root' is valid.

void remove(btree\* &root, int key);

// ----------------------------------------- C O N T A I N S ----------------------------------------------------------------\\

//

// Return true if any node accessed from the given root node contains

// the provided key value.

bool contains(btree\* &root, int key);

if root is null

return false

if x has children

if key is within grouping

return true

Find which child contains key

Recurse on this child

If x has no children

if key is in node

return true

else return false

RETURN: TRUE OR FALSE  
//On exit:

//btree should not have changed

//btree must pass all invariants

//---------------------------------------------------Additional functions-----------------------------------------------------\\

In sorta pseudocode

Init\_node(node\* &root)

int num\_keys=0;

Int keys[BTREE\_ORDER]=[];

bool is\_leaf=true;

btree\* children[BTREE\_ORDER + 1]=0;

Int median(btree\* &root)

Run one iteration of bubble sort

Pick ceil(size/2)

Return key

Void add\_key(btree\* &root, int key)

For(i = 0 ; i < size-1 ; i)

If key > key[i] && key < key[i+1]

For size

Swap elements into order

If key < key[i]

Swap all elements back 1 position

If key > key[i]

I++

Void add\_height(node\* &root, key)

Keys.push\_back(key)

Keys[0] = median(&root)

Initialize node, add < keys to left

Initialize node, add > keys to right

Set up links to children

Void rebalance(node\* &root)

If right has > ceil(m/2)-1

Add key to end of starved node

Replace the key with the replacement node

Fix children by adding first child of right to last child of left

If left has > ceil(m/2)-1

Add key to starved node

Replace key with replacement node

Fix children by adding last child of left to first child of right

Else

Initialize new node, add all elements of starved node and its sibling

Remove replacement node, replace children with combined node

If parent is starved

Rebalance( parent)