Poonam Gupta

COT 5407 – Introduction to Algorithms **Homework Assignment #4**

Problem 1: B-Tree Deleting a Key

Case 1:

In this case, we check for the following conditions in a given node x

a) The key k is in node x;
b) x.leaf is TRUE
c) x.n > t-1

In the following case, we locate the key k from $x.key_{1 \ldots}$ $x.key_{x.n}$. Once the key is located, delete the key and reduce size of the node to reflect the change. At all times, we ensure that all none root nodes have at least t -1 keys.

```
B-Tree-Delete-Key(x,k)
if (x.leaf) && (x.n > t – 1)
    for i=1 to x.n
        if x.key_i == k
            DELETE-Key(x,k)          // Delete key k from node x and adjust the size of node
            DISK-WRITE(x)
            Return true
```

Case 2a:

```
//In this case, the key k is located in an internal nodes i.e. x.leaf is FALSE.
//Check if it has t – 1 keys.
if (!x.leaf) && (x.n > t – 1)
    i = 1                            //Starting with the first key
    while x.key_i < k                //Scan each entry till we reach key that is smaller than k
        i = i + 1
    y = x.c_i                        //Identify index i in x to identify location of k so we can get x.c_i
                                     to identify preceding child y.
    DISK-READ(y)                     // Read preceding child from disk
    if (y.n > t – 1)                 //Check if the preceding child has t -1 keys.
        k'= Find-Predecessor-Key (k, y)  // Locate the predecessor k' of a given key k in a subtree rooted
                                     at y (largest key less than k)
        B-Tree-Delete-Key(y,k')      // Recursively delete k' from the predecessor child
        x.key_i=k'                   // Replace k with k'
        Disk-Write(x)                //Write node x back to disk
        Return true                      //Return
```

Case 2b

```
    z = x.c_i+1                      //Identify index i+1 in x to identify location of k so we can get
                                     x.c_i+1 to identify successor child z.
    DISK-READ(z)                     // Read successor child from disk
```

```
    if (z.n > t – 1)                      //Check if the successor child has t -1 keys.
        k'= Find- Successor-Key (k, y)    // Locate the successor k' of a given key k in a subtree rooted at
                                          y (smallest key less than k)
        B-Tree-Delete-Key(z,k')           // Recursively delete k' from the successor child
        x.key_i=k'                        // Replace k with k'
        Disk-Write(x)                     //Write node x back to disk
        Return true                       //Return
```

Case 2c

```
if (z.n == t-1) && (y.n == t-1) // Check if the preceding as well as successor child has only t-1 keys.
    y.key_{n+1} = k              // Insert key k into y
for tmp1 = 1 to z.n              //for each key in z
    y.key_{n+1+ tmp1} = z.key_{tmp1} // Append key from z to the end of y
y.n = y.n + 1 + z.n              // Update the size of y to reflect the merged node (y and z)
for tmp2 = i + 1 to x.n -1       // Starting from the next key after x.keyi , move each key left
    x.c_{tmp2} = x.c_{tmp2+1}    // Update the node x such that key k is removed from the node
x.n = x.n-1                      // Update the length of node x
free(z)                          // Free the pointer to node z as it has already been merged
B-TREE-DELETE(y,k)               // Recursively delete key k from y.
DISK-WRITE(x)                    // Write nodes x,y,z to disk
DISK-WRITE(y)
DISK-WRITE(z)
Return true
```

Case 3

In this case, we are trying to search for k while traversing down the tree. During this process, we review each internal node to check if it has t-1 keys and if it is, we will need to adjust the number of keys so that it is not minimum. As we are traversing down, we also identify root $x.c_i$ of the subtree that must contain k.

```
x,i = B-Tree-Search(x,k)          //Returns the node and location of key k
x.c_i = Find-Root(x,k)            //Returns Root x.ci
DISK-READ(x.c_i)
if x.c_i.n > t - 1                // If the node contains t – 1 keys, we can continue
    B-TREE-DELETE(x.c_i,k)
DISK-READ(x.c_{i+1})              //
```

Case 3a

```
Else if (x.c_{i+1}.n) >=t          //Check if the right sibling has t or more keys
    x.c_i.key_{n+2} = x.key_i      //Push the key from x down to x.ci
    x.c_i.n = x.c_i.n + 1          //Increment the length of x.ci to reflect the additional key
    x.key_i = x.c_{i+1}.key_1      //Move key from x.ci's immediate right sibling up into x
    x.c_i.c_{n+1} = x.c_{i+1}.c_1  //Move the child pointer from the sibling into x.ci
    for tmp1 = 1 to x.c_{i+1}.n
```

```
        x.c_{i+1}.key_j = x.c_{i+1}.key_{j+1}   //Move each key in sibling left.
    x.c_{i+1}.n = x.c_{i+1}.n - 1               // Reduce the length of the right sibling to reflect the change.
    DISK-WRITE(x)                               // Write all nodes to disk
    DISK-WRITE(x.c_i)
    DISK-WRITE(x.c_{i+1})
    B-TREE-DELETE(x.c_i,k)                      // Recursively delete key k from the Root x.ci
```

Case 3a continued

```
Else if (x.c_{i-1}.n) >=t                       //Check if the left sibling has t or more keys
    x.c_i.key_{n+2} = x.key_i                   //Push the key from x down to x.ci
    x.c_i.n = x.c_i.n + 1                       //Increment the length of x.ci to reflect the additional key
    x.key_i = x.c_{i-1}.key_1                   //Move key from x.ci's immediate left sibling up into x
    x.c_i.c_{n+1} = x.c_{i-1}.c_1               //Move the child pointer from the sibling into x.ci
    for tmp1 = 1 to x.c_{i-1}.n
        x.c_{i-1}.key_j = x.c_{i-1}.key_{j+1}   //Move each key in sibling left.
    x.c_{i-1}.n = x.c_{i-1}.n - 1               // Reduce the length of the left sibling to reflect the change.
    DISK-WRITE(x)                               // Write all nodes to disk
    DISK-WRITE(x.c_i)
    DISK-WRITE(x.c_{i-1})
    B-TREE-DELETE(x.c_i,k)                      // Recursively delete key k from the Root x.ci
```

Case 3b

```
Else                                            //if both nodes have t-1 keys
    i = 1                                       //Starting with the first key
    while x.key_i < k                           //Scan each entry till we reach key that is smaller than k
        i = i + 1
    y = x.c_i                                       //Identify index i in x to identify location of k so we can get x.c_i
                                                    to identify preceding child y.
    z = x.c_{i+1}                                   //Identify index i+1 in x to identify location of k so we can get
                                                    x.c_{i+1} to identify successor child z.
    y.key_{n+1} = k                                 // Insert key k into y
    for tmp1 = 1 to z.n                             //for each key in z
        y.key_{n+1+ tmp1} = z.key_{tmp1}            // Append key from z to the end of y
    y.n = y.n + 1 + z.n                             // Update the size of y to reflect the merged node (y and z)
    for tmp2 = i + 1 to x.n -1                      // Starting from the next key after x.keyi , move each key left
        x.c_{tmp2} = x.c_{tmp2+1}                   // Update the node x such that key k is removed from the node
    x.n = x.n-1                                     // Update the length of node x
    free(z)                                         // Free the pointer to node z as it has already been merged
    B-TREE-DELETE(y,k)                              // Recursively delete key k from y.
    DISK-WRITE(x)                                   // Write nodes x,y,z to disk
    DISK-WRITE(y)
    DISK-WRITE(z)
    Return true
```

Combined code

```
B-Tree-Delete-Key(x,k)
//Case 2a – Identify if internal node, find predecessor child and if it has only t-1 keys
if(x.leaf = FALSE)
    if (!x.leaf) && (x.n > t – 1)
        i = 1
        while x.key_i < k
            i = i + 1
        y = x.c_i
        DISK-READ(y)
        if (y.n > t – 1)
            k'= Find-Predecessor-Key (k, y)
            B-Tree-Delete-Key(y,k')
            x.key_i=k'
            Disk-Write(x)
            Return true
//Case 2b -  Identify successor child and check if successor has t – 1 keys.
        else if (x.c_i+1.n > t-1)
            z = x.c_i+1
            DISK-READ(z)
            k'= Find- Successor-Key (k, y)
            B-Tree-Delete-Key(z,k')
            x.key_i=k'
            Disk-Write(x)
            Return true
//Case 2c – if preceding as well as successor child has only t-1 keys
        else if (z.n == t-1) && (y.n == t-1)
            y.key_n+1 = k
            for tmp1 = 1 to z.n
                y.key_n+1+ tmp1 = z.key_tmp1
            y.n = y.n + 1 + z.n
            for tmp2 = i + 1 to x.n -1
                x.c_tmp2 = x.c_tmp2+1
            x.n = x.n-1
            free(z)
            B-TREE-DELETE(y,k)
            DISK-WRITE(x)
            DISK-WRITE(y)
            DISK-WRITE(z)
            Return true
// Case 1 (k is in leaf node)
else if(x.leaf == True)
    if (x.leaf) && (x.n > t – 1)
```

```
    for i=1 to x.n
        if x.key_i == k
            DELETE-Key(x,k)
            DISK-WRITE(x)
            Return true
//Case 3b
    else
        x,i = B-Tree-Search(x,k)
        x.c_i = Find-Root(x,k)
        DISK-READ(x.c_i)
        if x.c_i.n > t - 1
            B-TREE-DELETE(x.c_i,k)
        DISK-READ(x.c_{i+1})
        if (x.c_{i+1}.n) >=t
            x.c_i.key_{n+2} = x.key_i
            x.c_i.n = x.c_i.n + 1
            x.key_i = x.c_{i+1}.key_1
            x.c_i.c_{n+1} = x.c_{i+1}.c_1
            for tmp1 = 1 to x.c_{i+1}.n
                x.c_{i+1}.key_j = x.c_{i+1}.key_{j+1}
            x.c_{i+1}.n = x.c_{i+1}.n - 1
            DISK-WRITE(x)
            DISK-WRITE(x.c_i)
            DISK-WRITE(x.c_{i+1})
            B-TREE-DELETE(x.c_i,k)
        Else if (x._{ci-1}.n) >=t
            x.c_i.key_{n+2} = x.key_i
            x.c_i.n = x.c_i.n + 1
            x.key_i = x.c_{i-1}.key_1
            x.c_i.c_{n+1} = x.c_{i-1}.c_1
            for tmp1 = 1 to x.c_{i-1}.n
                x.c_{i-1}.key_j = x.c_{i-1}.key_{j+1}
            x.c_{i-1}.n = x.c_{i-1}.n - 1
            DISK-WRITE(x)
            DISK-WRITE(x.c_i)
            DISK-WRITE(x.c_{i-1})
            B-TREE-DELETE(x.c_i,k)
//Case 3b
        Else
            i = 1
            while x.key_i < k
                i = i + 1
            y = x.c_i
            z = x.c_{i+1}
```

```
y.key_{n+1} = k
for tmp1 = 1 to z.n
      y.key_{n+1+ tmp1} = z.key_{tmp1}
y.n = y.n + 1 + z.n
for tmp2 = i + 1 to x.n -1
      x.c_{tmp2} = x.c_{tmp2+1}
x.n = x.n-1
free(z)
B-TREE-DELETE(y,k)
DISK-WRITE(x)
DISK-WRITE(y)
DISK-WRITE(z)
Return true
```

In summary, during the traversal of the tree, we identify, at each node the criteria on where it is an internal node and how many keys exist in this node and possibly its siblings and then ensure that the rules of the structure of B-tree are maintained at all times. At times, we need to adjust keys, either by rearranging the node's children or by backing up and then adjusting the keys.

Complexity analysis: To delete a key from the B-Tree, we have to use the search operation and then adjust the node or siblings. Due to this, the worst-case scenario for delete operation in B-Tree is O(logn) and the average case is Θ(logn).

Problem 2 (3 + 1 points) Disjoint Sets

In the depth-determination problem, we maintain a forest F={Ti} of rooted trees under three operations:
Make−Tree(v) creates a tree whose only node is v,
Find−Depth(v) returns the depth of node v,
Graft (r, v) makes node r (the root of a tree) as a child of node v of a different tree.

(1) (1 point) Suppose a tree representation similar to a disjoint-set forest is used: $v.p$ is the parent of node v, except $v.p=v$ if v is a root. Suppose further that we implement *Graft* (r, v) by setting $r.p=v$ and *Find−Depth*(v) by following the find path up to the root and returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m *Make−Tree* (total n), *Find−Depth* (total n), and *Graft* (total n) operations is Θ(n^2).

Answer: If we use disjoint-set data structure, Make−Tree takes Θ (1) time.
Graft is basically a union operation; thus, it takes Θ (1) times.
The cost of Find−Depth depends on the depth of the given node i.e. linear.
For a sequence of n operations, the depth of a node is O(n), thus, for the worst-case T(n)=n O(n) = O(n^2).
For example, let k = m/3 be an integer, considering a sequence of operations with k + 1 Make−Trees creating k + 1 single-node trees, k Grafts forming a single path, and k − 1 FIND-DEPTH for the leaf node, then the running time of the n operations is

$T(n) = (k + 1) * \Theta(1) + k\,\Theta(1) + (k - 1) * k = \Omega(m^2)$.
Hence the worst-case running time is $\Theta(m^2)$.

(2) (1 point) Give an algorithm of *Find−Depth* by modifying *Find-Set*. Your algorithm should perform path compression and its running time should be linear in the length of the find path. Make sure your algorithm updates the pseudo-distances correctly.

Answer: According to the definition of $v.d$ that the sum of the pseudo-distances along the path from v to root of its set S_i equals to the depth of v in T_i, *Find−Depth* can be implemented by modifying *Find-Set* in such a way: assume the path is composed of $v_0, \ldots, v_k$ where v_k is the root, for every node v_i along the path, update depth of v in T_i is $\sum_{j=0}^{k} v_j \,.\, d$,i.e., with path compression, whenever the parent pointer of a node changes, the pseudo-distances is updated by the sum of its ancestor's pseudo-distances.

```
FIND-SET (v)
If (v ≠ v.p) then
(v.p,d) = FIND-SET(v.p)
v.d = v.d +d
return (v.p, v.d)
else
return (v,0)
end if
```

(3) (1 point) Give an algorithm of *Graft* (r, v), which combines the sets containing r and v, by modifying the *Union* and *Link* procedures. Make sure that your algorithm updates pseudo-distances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

Answer: To implement *Graft* we need to find v 's actual depth and add it to the pseudodistance of the root of the tree S_i which contains r.

```
GRAFT (r, v)
(X, d1) = Find-set(r)            / / find the parent of set
(y, d2) = Find-set(v)
if (x.rank > y.rank) then      //check whose rank is higher
        y.p = x
        x.d = x.d + d2 + y.d
Else
        x.p = y
        x.d = x.d + d2
        if (x.rank == y.rank) then
                y.rank = y.rank + 1
        end if
end if
```

(4) (1 bonus point) Give a tight bound on the worst-case running time of a sequence of *m* *Make−Tree*, *Find−Depth*, and *Graft* operations, *n* of which are *Make−Tree* operations.

Answer: The three implemented operations have the same asymptotic running time as MAKE, FIND, and UNION for disjoint sets, so the worst-case runtime of m such operations, n of which are MAKE-TREE operations, is $O(m\alpha(n))$.