CS271: Project 7

Krystal Ly, Hamy Nguyen, Casey Kotbyul Kim

9 December 2022

1 Inserting and Deleting with BTrees

1. Visually represent the result of inserting into a BTree with t = 2 the following values in the order listed: F, J, D, H, B, C, I, G, A, E, K, L, M

   For each value, show the BTree at the completion of the call to B-Tree-Insert(T, k). Additionally, for each value inserted, list next to the resulting B-Tree, the function calls made throughout the insertion. An example of this format is given at the end of this project description for your reference.

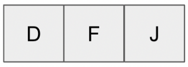
   Insert F

   $B - TREE - INSERT(T, F)$
   $\quad B - TREE - INSERT - NONFULL(x = T.root, F)$

   Insert J

   $B - TREE - INSERT(T, J)$
   $\quad B - TREE - INSERT - NONFULL(x = T.root, J)$

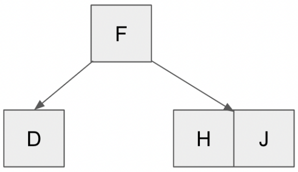
   Insert D



$B - TREE - INSERT(T, D)$

$\quad B - TREE - INSERT - NONFULL(x = T.root, D)$

Insert H



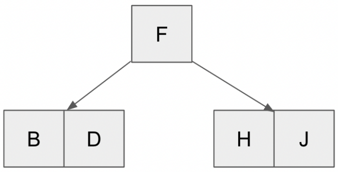
$B - TREE - INSERT(T, H)$

$\quad B - TREE - SPLIT - CHILD(x = s, 1)$

$\quad B - TREE - INSERT - NONFULL(x = T.root, H)$

$\qquad B - TREE - INSERT - NONFULL(x = x.c_2, H)$

Insert B

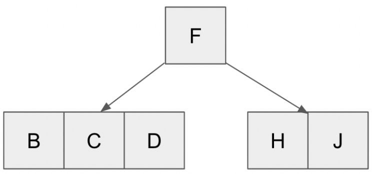


$B - TREE - INSERT(T, B)$

$\quad B - TREE - INSERT - NONFULL(x = T.root, B)$

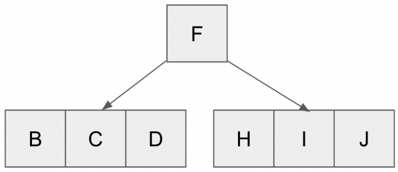
$\qquad B - TREE - INSERT - NONFULL(x = x.c_1, B)$

Insert C



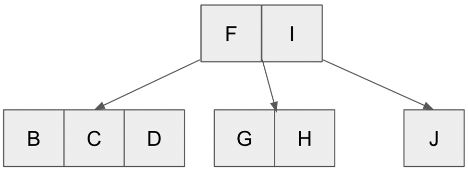
$B - TREE - INSERT(T, C)$
$B - TREE - INSERT - NONFULL(x = T.root, C)$
$B - TREE - INSERT - NONFULL(x = x.c_1, C)$

Insert I



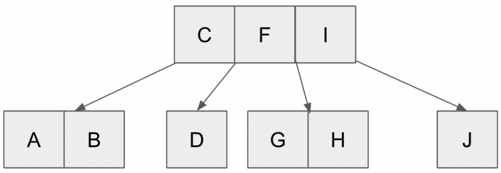
$B - TREE - INSERT(T, I)$
$B - TREE - INSERT - NONFULL(x = T.root, I)$
$B - TREE - INSERT - NONFULL(x = x.c_2, I)$

Insert G



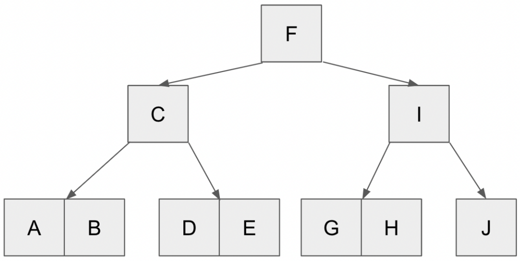
$B - TREE - INSERT(T, G)$
$B - TREE - INSERT - NONFULL(x = T.root, G)$
$B - TREE - SPLIT - CHILD(x, 2)$
$B - TREE - INSERT - NONFULL(x = x.c_2, G)$

Insert A



$B - TREE - INSERT(T, A)$
$B - TREE - INSERT - NONFULL(x = T.root, A)$
$B - TREE - SPLIT - CHILD(x, 1)$
$B - TREE - INSERT - NONFULL(x = x.c_1, A)$

Insert E



$B-TREE-INSERT(T,E)$

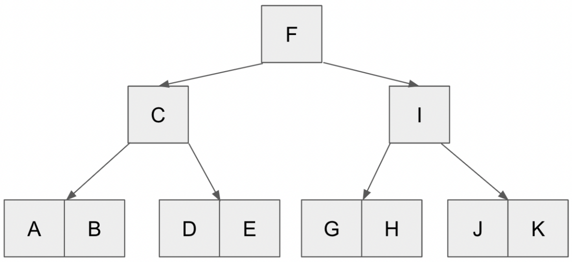
$\quad B-TREE-SPLIT-CHILD(x=s,1)$

$\quad B-TREE-INSERT-NONFULL(x=T.root,E)$

$\quad\quad B-TREE-INSERT-NONFULL(x=x.c_1,E)$

$\quad\quad\quad B-TREE-INSERT-NONFULL(x=x.c_2,E)$

Insert K



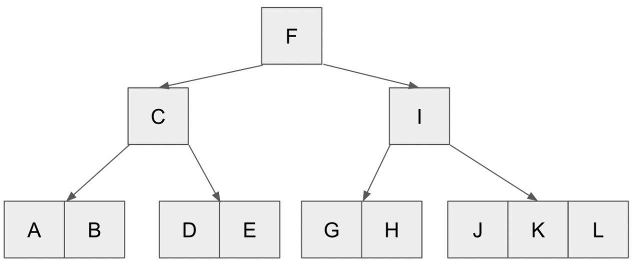
$B-TREE-INSERT(T,K)$

$\quad B-TREE-INSERT-NONFULL(x=T.root,K)$

$\quad\quad B-TREE-INSERT-NONFULL(x=x.c_2,K)$

$\quad\quad\quad B-TREE-INSERT-NONFULL(x=x.c_2,K)$

Insert L



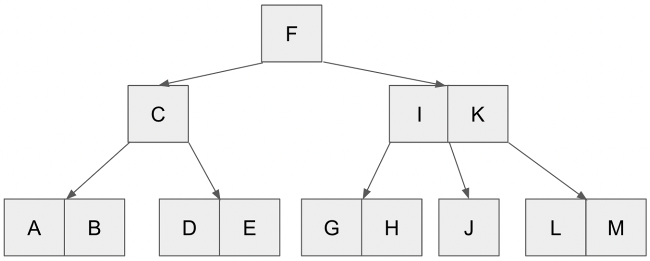
$B-TREE-INSERT(T,L)$

$\quad B-TREE-INSERT-NONFULL(x=T.root,L)$

$\quad\quad B-TREE-INSERT-NONFULL(x=x.c_2,L)$

$\quad\quad\quad B-TREE-INSERT-NONFULL(x=x.c_2,L)$

Insert M



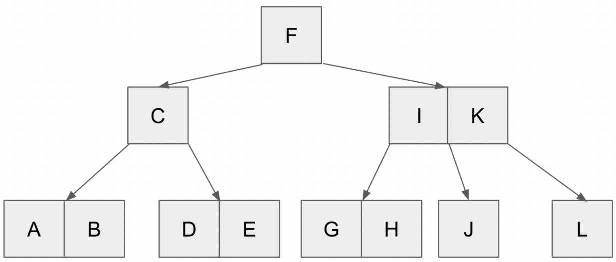
$B-TREE-INSERT(T, M)$
$\quad B-TREE-INSERT-NONFULL(x = T.root, M)$
$\quad\quad B-TREE-INSERT-NONFULL(x = x.c_2, M)$
$\quad\quad\quad B-TREE-SPLIT-CHILD(x, 2)$
$\quad\quad\quad B-TREE-INSERT-NONFULL(x = x.c_3, M)$

2. Visually represent the result of deleting from the BTree created in problem 1 the following values in the order listed:

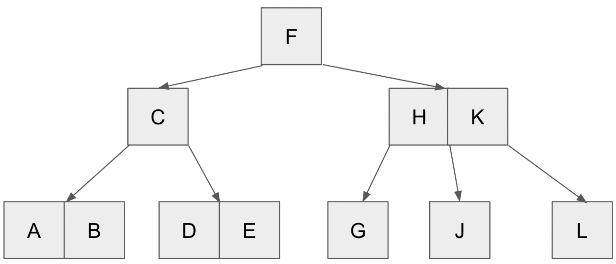
M, I, H, B, E

You are only responsible for showing the final tree after each deletion. No function calls are required.

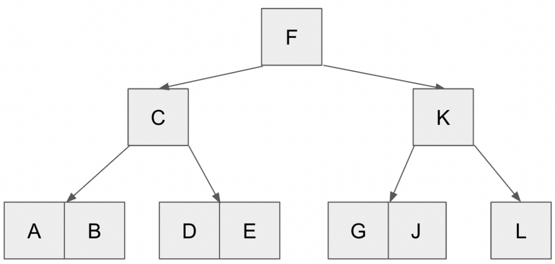
Delete M



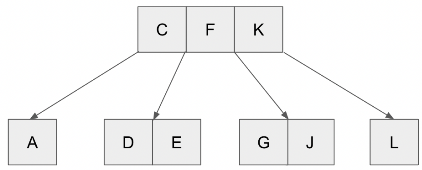
Delete I



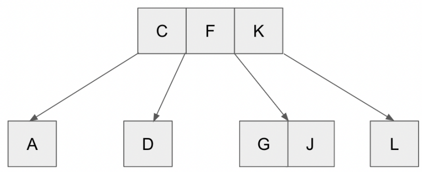
Delete H



Delete B



Delete E



2 C++ Implementation

Consider the following 2 pseudocode options for implementing the Allocate-Node() functionality of the BTree in C++. How would each impact the runtime of the B-Tree-Insert function? Consider both asymptotic analysis as well as real time impacts.

```
Allocate-Node()
    x = Node()
    x.leaf = true
    x.n = 0
    x.keys = new int[2*t-1]      \\ member variable int* keys
    x.c = new Node*[2*t]         \\ member variable Node** c

Allocate-Node()
    x = Node()
    x.leaf = true
    x.n = 0
    x.keys = { }                  \\ member variable vector<int> keys
    x.c = { }                     \\ member variable vector<Node*> c
```

- Asymptotically, the two implementations do not give a different runtime for $B-TREE-INSERT$. Since for the two $allocate-node()$ function, it will create dynamic memory to make sure that the arrays will create enough space for $2t-1$ keys and t pointer holders for each node. Thus, running time to iterate through the arrays to find the correct positions for new inserted keys will be O($2t-1$). Therefore, asymptotic runtime for insert in B-tree of both implementations will remain $O(\log_t n)$. However, in terms of real time impact, the second implementation which uses a vector can cost more time compared to the first one which uses a pre-allocated array. In C++, vectors are implemented with a dynamically allocated array. Therefore, when the array is full, it has to reallocate a bigger array and copy every element from the old array to the new array. This process is expensive in terms of processing time. On the other hand, if we pre-allocate the amount of memory that can store the maximum possible number of elements ($2t-1$) like in the first implementation, there will be no extra cost of resizing the array and moving all elements. Therefore, although the two implementations would not give different asymptotic runtimes, using the first implementation can give a faster runtime in real time.

3 Pseudocode

Write pseudocode for the public B-Tree class method B-Tree-Delete(T , k). You are encouraged to write additional, supporting methods but you may not call any method for which you do not write pseudocode. Your code should be written in a syntax consistent with your textbooks. You may therefore assume the following overall class structure:

Algorithm 1: B-TREE-DELETE(x,k)

```
linenosize=
i = x.n
// Case 1
if x.leaf then
    prev_k = NIL
    while i ≥ 1 and k < x.key_i do
        temp = x.key_i
        x.key_i = prev_k
        prev_k = temp
        i = i − 1
    end
    if x.key_i exists and k == x.key_i then
        x.key_i = prev_k
        x.n = x.n − 1
        // if the B-Tree is empty after deleting
        if x.n == 0 then
            T.root = NIL
        end
    else
        return NIL ;                                   // k doesn't exist
    end
else
    while i ≥ 1 and k < x.key_i do
        i = i − 1
    end
    if x.key_i exists and k == x.key_i then
        // Case 2a
        if x.c_i.n ≥ t then
            predecessor = x.c_i.key_{x.c_i.n}
            B-TREE-DELETE(x.c_i,predecessor)
            x.key_i = predecessor
        // Case 2b
        else if x.c_{i+1}.n ≥ t then
            successor = x.c_{i+1}.key_1
            B-TREE-DELETE(x.c_{i+1},successor)
            x.key_i = successor
        // Case 2c
        else if x.c_i.n == t − 1 and x.c_{i+1}.n == t − 1 then
            B-TREE-MERGE-CHILD(x, i)
            B-TREE-DELETE(x.c_i,k)
    else
        i = i + 1
        if x.c_i.n == t − 1 then
            // Case 3a with right immediate sibling
            if x.c_{i+1} exists and x.c_{i+1}.n ≥ t then
                B-TREE-BORROW-FROM-SIBLING(x, i, i + 1)
                B-TREE-DELETE(x.c_i, k)
            // Case 3a with left immediate sibling
            else if x.c_{i−1} exists and x.c_{i−1}.n ≥ t then
                B-TREE-BORROW-FROM-SIBLING(x, i, i − 1)
                B-TREE-DELETE(x.c_i, k)
            // Case 3b with right immediate sibling
            else if x.c_{i+1} exists and x.c_{i+1}.n == t − 1 then
                B-TREE-MERGE-CHILD(x,i)
                B-TREE-DELETE(x.c_i,k)
            // Case 3b with left immediate sibling
            else if x.c_{i−1} exists and x.c_{i−1}.n == t − 1 then
                B-TREE-MERGE-CHILD(x,i − 1)
                B-TREE-DELETE(x.c_{i−1},k)
        else
            B-TREE-DELETE(x.c_i, k)
        end
    end
end
```

Algorithm 2: B-TREE-MERGE-CHILD(x,i)

```
y = x.c_i ;                                  // y: left child
z = x.c_{i+1} ;                              // z: right child
for j = 1 to t - 1 do
    y.key_{j+t} = z.key_j ;                  // move all keys from z to y
end
if not z.leaf then
    for j = 1 to t do
        y.c_{j+t} = z.c_j ;                  // move all children of z to y
    end
end
// y.leaf = False if y or z has children
y.leaf = y.leaf and z.leaf
y.n = 2t - 1
for j = i + 1 to x.n do
    x.c_j = x.c_{j+1} ;                      // shift children in x and delete the last
end
x.c_{x.n+1} = NIL
y.key_t = x.key_i
for j = i to x.n - 1 do
    x.key_i = x.key_{i+1} ;                  // shift keys in x and delete the last
end
x.key_{x.n} = NIL
x.n = x.n - 1
if x == T.root and x.n == 0 then
    T.root = y
end
```

Algorithm 3: B-TREE-BORROW-FROM-SIBLING(x, i, s)

```
// Borrow from the right sibling
if s == i + 1 then
    y = x.c_i
    z = x.c_{i+1}
    y.key_t = x.key_i
    y.n = t
    // if z is not a leaf, bring the corresponding child from z to y
    if not z.leaf then
        y.c_{t+1} = z.c_1
        // shift the children in z
        for j = 1 to n − 1 do
            z.c_j = z.c_{j+1}
        end
        z.c_{z.n} = NIL
    end
    // change y.leaf if necessary
    if y.leaf == True and y.c_{t+1} ≠ NIL then
        y.leaf = False
    end
    x.key_i = z.key_1
    // shift the keys in z
    for j = 1 to n − 1 do
        z.key_j = z.key_{j+1}
    end
    z.key_n = NIL
    z.n = z.n − 1
    // check whether z is a leaf
    z.leaf = True
    for j = 1 to z.n + 1 do
        if z.c_j! = NIL then
            z.leaf = False
        end
    end
// Borrow from the left sibling
else if s = i − 1 then
    y = x.c_{i+1}
    z = x.c_{i−1}
    // shift keys in y
    for j = y.n + 1 to 2 do
        y.key_j = y.key_{j−1}
    end
    y.key_1 = x.keyi − 1
    // shift children in y
    for j = y.n + 2 to 2 do
        y.c_j = y.c_{j−1}
    end
    // if z is not a leaf, bring the corresponding child from z to y
    if not z.leaf then
        y.c_1 = z.c_{z.n+1}
        z.c_{z.n+1} = NIL
    end
    y.n = t
    // change y.leaf if necessary
    if y.leaf == True and y.c_1 ≠ NIL then
        y.leaf = False
    end
    x.key_{i−1} = z.key_{z.n}
    z.key_{z.n} = NIL
    z.n = z.n − 1
    // check whether z is a leaf
    z.leaf = True
    for j = 1 to z.n + 1 do
        if z.c_j! = NIL then
            z.leaf = False
        end
    end
```