24 + 9 + 44 + 6 = 83%

# CS271: Data Structures

Name: Bao Luu, Thanh Do, Khoa Nguyen

Instructor: Dr. Stacey Truex

## Project #5

## Exercises

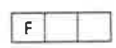
### Inserting and Deleting with BTrees

24/26 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.

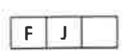
Insert F:




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

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

Insert J:




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

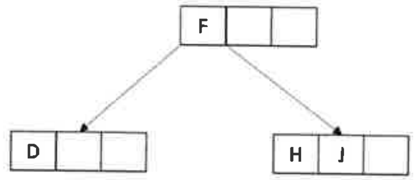
B-TREE-INSERT-NONFULL($x = T.root$, J)

Insert D:


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

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

Insert H:

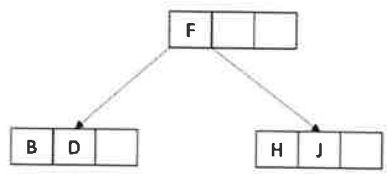



B-TREE-INSERT($T$, H)

B-TREE-SPLIT-CHILD($x = T.root$, 1)

B-TREE-INSERT-NONFULL($\cancel{x}$ s, H) this call is to insert into subtree rooted @ s = T.root
– will trigger an additional call to insert @ 2nd child

Insert B:

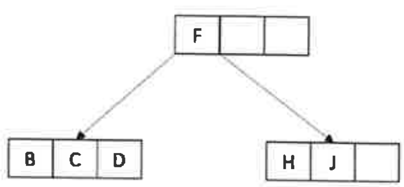



B-TREE-INSERT($T$, B)

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

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

Insert C:

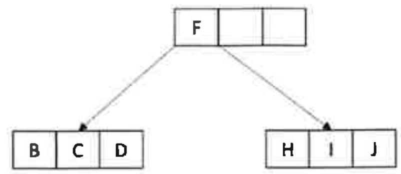



B-TREE-INSERT($T$, C)

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

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

Insert I:

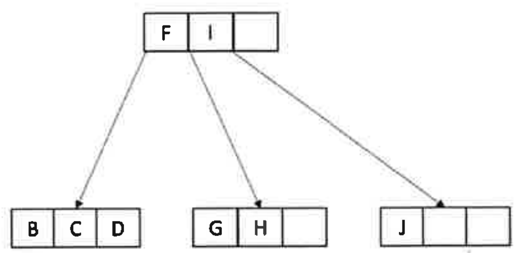



B-Tree-Insert($T$, I)

  B-Tree-Insert-Nonfull($x = T.root$, I)

    B-Tree-Insert-Nonfull($x = x.c_2$, I)

Insert G:



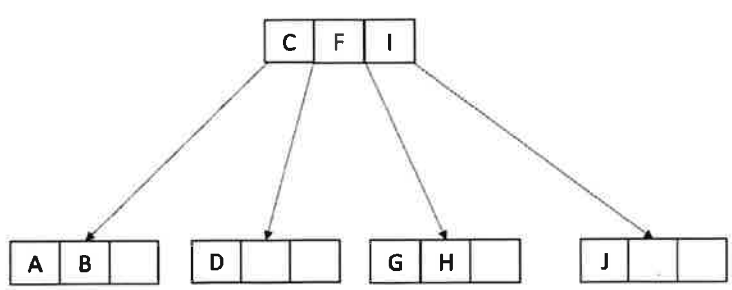

B-Tree-Insert($T$, G)

  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:



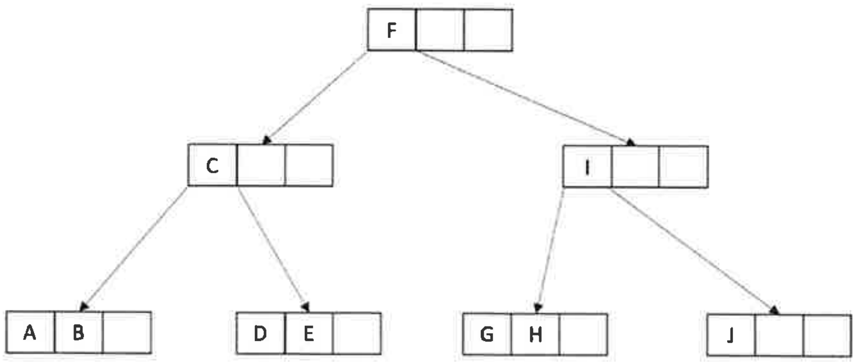
B-Tree-Insert($T$, A)

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)

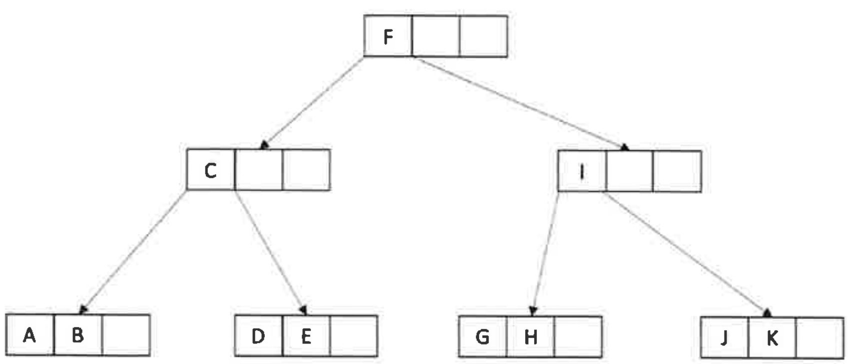
B-Tree-Split-Child($x$, 1) undefined in call to B-Tree-Insert

B-Tree-Insert-Nonfull($x = T.root$, E)

B-Tree-Insert-Nonfull($x = x.c_1$, E)

B-Tree-Insert-Nonfull($x = x.c_2$, E)

Insert K:

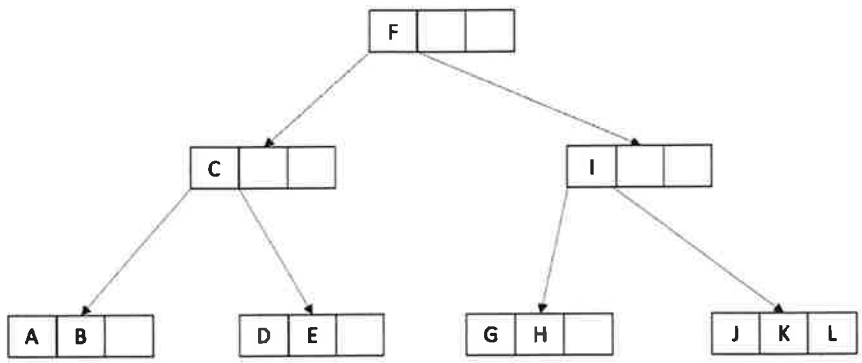



B-Tree-Insert($T$, K)

B-Tree-Insert-Nonfull($x = T.root$, K)

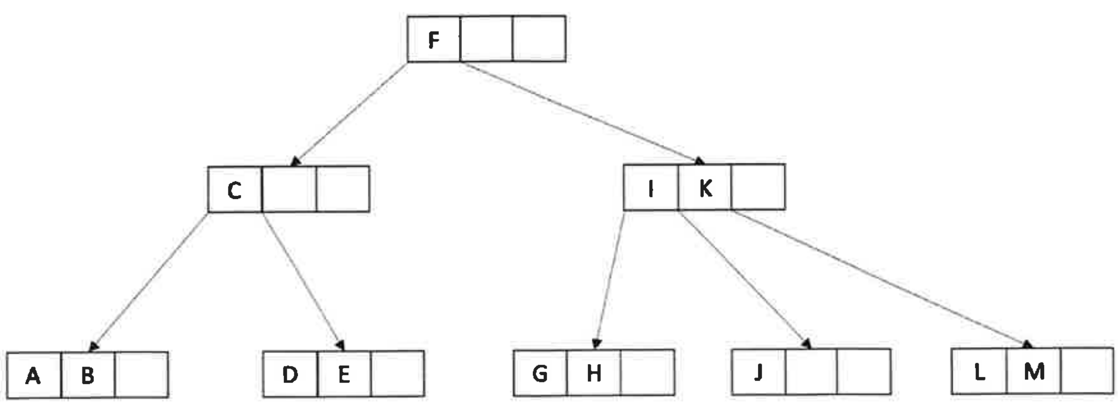
B-Tree-Insert-Nonfull($x = x.c_2$, K)
B-Tree-Insert-Nonfull($x = x.c_2$, K)

Insert L:




B-Tree-Insert($T$, L)
B-Tree-Insert-Nonfull($x = T.root$, L)
B-Tree-Insert-Nonfull($x = x.c_2$, L)
B-Tree-Insert-Nonfull($x = x.c_2$, L)

Insert M:




B-Tree-Insert($T$, M)

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

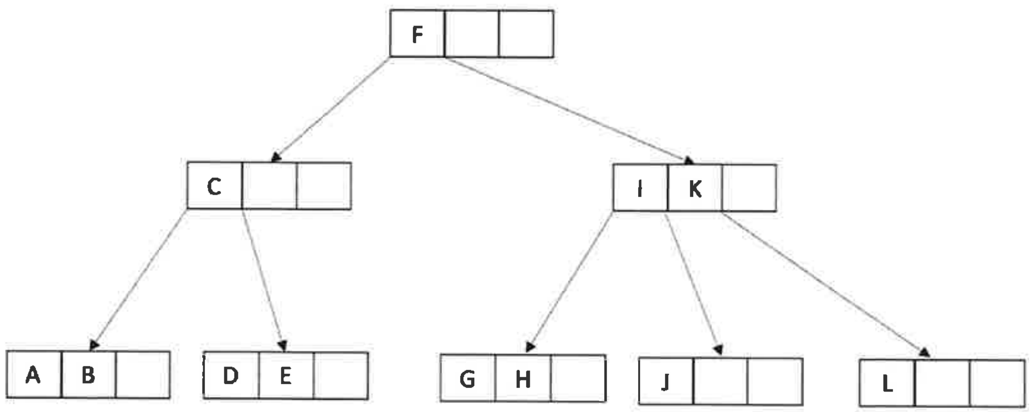
9/15

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

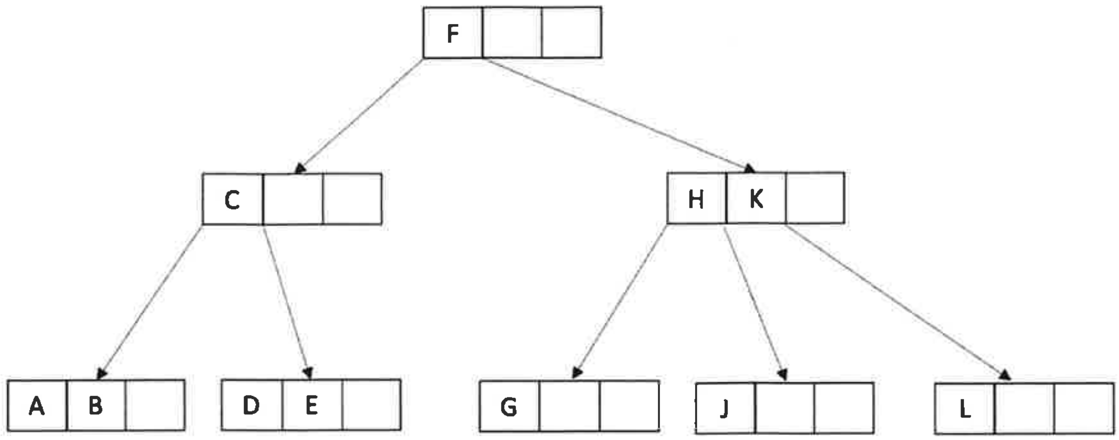
$$M, I, H, B, E$$

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

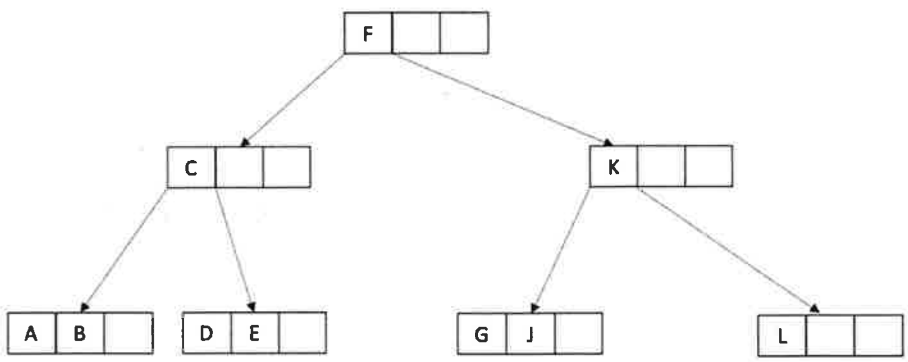
Delete M:



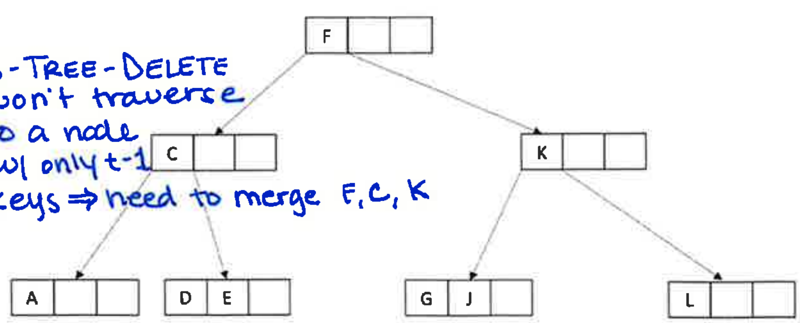

Delete I:




Delete H:

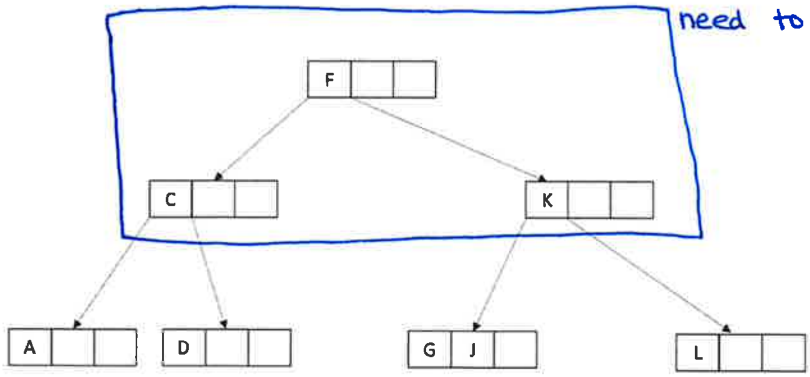



Delete B:




B-TREE-DELETE won't traverse to a node w/ only t-1 keys ⇒ need to merge F, C, K

Delete E:




need to merge

## Pseudocode

49/53

3. Write pseudocode for the `public B-Tree` class method B-Tree-Delete($T$, $k$).

   Pre-condition: The B-tree is valid with order t and the key to be deleted exists in the tree.
   Post-condition: The key is deleted from the B-tree, and the B-tree remains balanced.

**Algorithm 1** Deleting a key $k$ from a BTree $T$ +2

**procedure** B-Tree-Delete($T$, $k$)
    **if** $T.root \neq NIL$ **then**
        B-Tree-Delete($T.root$, $k$)
    **end if**
**end procedure**

when do we need to update T.root?

Pre-condition: The node $x$ is valid and the key to be deleted exists in the tree.
Post-condition: The key is deleted from the B-tree rooted at node $x$, and the B-tree rooted at node $x$ remains balanced.

**Algorithm 2** Deleting a key $k$ from a BTree rooted at a node $x$ +5

```
procedure B-TREE-DELETE(x, k)
    if x == NIL then
        return
    end if
    i = FIND-K(x, k)
    if i ≤ x.n and x.keys_i == k then
        if x.leaf == True then
            REMOVE-LEAF-KEY(x, i)
        else
            REMOVE-INTERNAL-KEY(x, i, i + 1)
        end if
    else
        if x.leaf == True then
            return
        end if
        y = x.c_i
        if y.n == t − 1 then
            if i > 1 and x.c_{i−1}.n ≥ t then
                SWAP-LEFT(x, y, x.c_{i−1}, i+ 1
            else if i < x.n + 1 and x.c_{i+1}.n ≥ t then
                SWAP-RIGHT(x, y, x.c_{i+1}, i)
            else if i > 1 then
                MERGE-LEFT(x.c_{i−1}, y, x.keys_i),1
                DELETE(x.keys_i)-1
            else            not a node - need to move down all keys and children to account for x
                MERGE-RIGHT(x.c_{i+1}, y, x.keys_i) having 1 fewer child/key
                DELETE(x.keys_i)
            end if
            B-TREE-DELETE(y, k)
        end if
    end if
end procedure
```

Pre-condition: The node $x$ is valid, and the $k$ is the key to be searched in the node $x$.
Post-condition: If there exists an index $i$ such that $k \leq x.keys_i$, then $i$ is the smallest index for which this condition holds, else $i$ is set to $x.n+1$.

**Algorithm 3** Determine the index $i$ of the first key in a B-Tree node $x$ where $k \leq x.key_i$ ($i = x.n+1$ if no such key exists) + 4

```
procedure FIND-K(x, k)
    if x == NIL then
        return NIL
    end if
    i = 1
    while i ≤ x.n and k > x.keys_i do
        i = i + 1
    end while
    return i
end procedure
```

Pre-condition: The node $x$ is a valid leaf node and the index $i \leq x.n$.
Post-condition: The key at index $i$ is deleted from the node $x$.

**Algorithm 4** Remove the key at index $i$ from a B-Tree leaf node $x$ + 4

```
procedure REMOVE-LEAF-KEY(x, i)
    if x == NIL or i > x.n then
        return x
    end if
    for j = i; j < x.n; j = j + 1 do
        x.keys_j = x.keys_{j+1}
    end for
    DELETE(x.keys_j)
    x.n = x.n − 1
end procedure
```

Pre-condition: The node $x$ is a valid node and $i \leq x.n$, $j \leq x.n + 1$.
Post-condition: The key at index $i$ is deleted from the node $x$, the child at index $j$ is modified accordingly, and the B-tree rooted at node $x$ remains satisfied.

**Algorithm 5** Remove the key at index $i$ and child at index $j$ from a B-Tree internal node $x$ +4

```
procedure REMOVE-INTERNAL-KEY(x, i, j)
    if x == NIL or i > x.n or j > x.n + 1 then
        return x
    end if
    k = x.keys_i
    y = x.c_i
    z = x.c_j
    if y.n ≥ t then
        pred = MAX(y)
        B-TREE-DELETE(y, pred)
        x.keys_i = pred
    else if z.n ≥ t then
        suc = MIN(z)
        B-TREE-DELETE(z, suc)
        x.keys_i = suc
    else
        MERGE-LEFT(y, z, k)
        for l = i; l < x.n; l = l + 1 do        need to also shift children down
            x.keys_l = x.keys_{l+1}              now that contents of z are in y
        end for
        ~~DELETE(x.keys_l)~~
        DELETE(z)
        B-TREE-DELETE(y, k)
    end if
    x.n = x.n − 1    not necessarily if k was replaced by pred or suc
end procedure
```

Pre-condition: The node $x$ is a valid node.
Post-condition: The maximum key of the sub-tree rooted at x is returned and the tree is unchanged.

**Algorithm 6** Return the maximum key in the B-Tree rooted at $x$ +3

**procedure** MAX($x$)
  **if** $x == NIL$ **then**
    **return** $x$
  **end if**
  **while** not $x.leaf$ ~~≠ True~~ **do**
    $x = x.c_{x.n+1}$ can directly use booleans :)
  **end while**
  **return** $x.keys_{x.n}$
**end procedure**

Pre-condition: The node $x$ is a valid node.
Post-condition: The minimum key of the sub-tree rooted at x is returned and the tree is unchanged.

**Algorithm 7** Return the minimum key in the B-Tree rooted at $x$ +3

**procedure** MIN($x$)
  **if** $x == NIL$ **then**
    **return** $x$
  **end if**
  **while** not $x.leaf$ ~~≠ True~~ **do**
    $x = x.c_1$
  **end while**
  **return** $x.keys_1$
**end procedure**

Pre-condition: The node $x$ and its right sibling $y$ are valid nodes that have $t-1$ keys, and $k$ is a valid key value.
Post-condition: The key $k$ and node $y$ is merged into $x$, and the B-tree properties are maintained.

**Algorithm 8** Merge key $k$ and all keys and children from $y$ into $y$'s left sibling $x$ + 6

**procedure** MERGE-LEFT($x$, $y$, $k$)
  **if** $x == NIL$ or $x.n \neq t-1$ or $y == NIL$ or $y.n \neq t-1$ **then**
    **return**
  **end if**
  $x.n = x.n + 1$
  $x.keys_{x.n} = k$
  **for** $i = 1$; $i \leq y.n$; $i = i + 1$ **do**
    $x.keys_{x.n+i} = y.keys_i$
  **end for**
  **for** $i = 1$; $i \leq y.n + 1$; $i = i + 1$ **do**
    $x.c_{x.n+i} = y.c_i$
  **end for**
  $x.n = 2t - 1$ fair, cleaner to use x.n+= y.n
**end procedure**

Pre-condition: The node $x$ and its left sibling $y$ are valid nodes that have $t-1$ keys, and $k$ is a valid key value.
Post-condition: The key $k$ and node $y$ is merged into $x$, and the B-tree properties are maintained.

**Algorithm 9** Merge key $k$ and all keys and children from $y$ into $y$'s right sibling $x$ + 4

**procedure** MERGE-RIGHT($x$, $y$, $k$)
  **if** $x == NIL$ or $x.n \neq t-1$ or $y == NIL$ or $y.n \neq t-1$ **then**
    **return**
  **end if**
  $x.n = x.n + 1$
  $x.keys_{x.n} = k$ should be smaller than all keys already in $x$
  **for** $i = 1$; $i < x.n$; $i = i + 1$ **do**
    $x.keys_{x.n+i} = x.keys_i$
  **end for**
  **for** $i = 1$; $i \leq y.n$; $i = i + 1$ **do**
    $x.keys_i = y.keys_i$
  **end for**
  **for** $i = 1$; $i \leq x.n$; $i = i + 1$ **do**
    $x.c_{x.n+i} = x.c_i$
  **end for**
  **for** $i = 1$; $i \leq y.n + 1$; $i = i + 1$ **do**
    $x.c_i = y.c_i$
  **end for**
  $x.n = 2t - 1$ fair
**end procedure**

Pre-condition: The node $y$, its left sibling $z$ and its parent $x$ are valid nodes. $i$ is the index dividing children $y$ and $z$ in $x$. Node y is at least 1 key away from being full and Node z is at least 1 key away from being min-ed out.
Post-condition: Node $y$ has an extra element that belonged to node $x$ and Node $x$ has an extra element that belonged to node $z$. B-tree property is maintained.

**Algorithm 10** Give $y$ an extra key by moving a key from its parent $x$ down into $y$, moving a key from $y$'s left sibling $z$ up into $x$, and moving the appropriate child pointer from $z$ into $y$.
Let $i$ be the index of the key dividing $y$ and $z$ in $x$. +5

**procedure** SWAP-LEFT($x$, $y$, $z$, $i$)
    **if** $z.n < t - 2$ or $y.n > 2t - 2$ **then**
        **return**
    **end if**
    $k = y.n$
    **while** $k > 0$ **do**
        $y.keys_{k+1} = y.keys_k$
        $k = k - 1$
    **end while**
    **if** not $y.leaf$ $\cancel{== False}$ **then**
        $j = y.n + 1$
        **while** $j > 0$ **do**
            $y.c_{j+1} = y.c_j$
            $j = j - 1$
        **end while**
    **end if**
    $y.keys_1 = x.keys_i$
    **if** not $y.leaf$ $\cancel{== False}$ **then**
        $y.c_{\cancel{0}\,1} = z.c_{z.n+1}$ careful. we index starting @ 1
    **end if**
    $x.keys_i = z.keys_{z.n}$
    $y.n = y.n + 1$
    $z.n = z.n - 1$
**end procedure**

Pre-condition: The node $y$, its right sibling $z$ and its parent $x$ are valid nodes. $i$ is the index dividing children $y$ and $z$ in $x$. Node y is at least 1 key away from being full and Node z is at least 1 key away from being min-ed out.
Post-condition: Node $y$ has an extra element that belonged to node $x$ and Node $x$ has an extra element that belonged to node $z$. B-tree property is maintained.

**Algorithm 11** Give $y$ an extra key by moving a key from its parent $x$ down into $y$, moving a key from $y$'s right sibling $z$ up into $x$, and moving the appropriate child pointer from $z$ into $y$. Let $i$ be the index of the key dividing $y$ and $z$ in $x$. ← 4

**procedure** SWAP-RIGHT($x$, $y$, $z$, $i$)
    **if** $z.n < t - 2$ or $y.n > 2t - 2$ **then**
        **return**
    **end if**
    $y.keys_{y.n+1} = x.keys_i$
    **if** not $y.leaf$ ~~== False~~ **then**
        $x.c_{x.n+1} = z.c_1$ child should be added to y
    **end if**
    $x.keys_i = z.keys_{\cancel{0}\,1}$
    $k = 2$
    **while** $k \leq z.n$ **do**
        $z.keys_{k-1} = z.keys_k$
        $k = k + 1$
    **end while**
    **if** not $z.leaf$ ~~== False~~ **then**
        $j = 2$
        **while** $j \leq z.n + 1$ **do**
            $z.c_{j-1} = z.c_j$
            $j = j + 1$
        **end while**
    **end if**
    $y.n = y.n + 1$
    $z.n = z.n - 1$
**end procedure**

6/6

## C++ Implementation nice, thorough work here!

4. How would each pseudocode option impact the runtime of the B-TREE-INSERT function? Consider both asymptotic analysis as well as real-time impacts.

The main difference between the two ALLOCATE-NODE options is that the one that uses a vector implementation does not initialize the size of the vector while the one that uses an array does. Because an array is sized dynamically, this does not pose a problem with compilation, but does with runtime. This is because whenever we try to insert into a full vector, the vector would move to another space in memory and create a copy of itself there as a way to dynamically allocate more space. This results in a slower runtime, as shown below.

The function for B-Tree-Insert(T,k) is as follows:

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t - 1
3      s = ALLOCATE-NODE()
4      T.root = s
5      s.leaf = FALSE
6      s.n = 0
7      s.c_1 = r
8      B-TREE-SPLIT-CHILD(s, 1)
9      B-TREE-INSERT-NONFULL(s, k)
10 else B-TREE-INSERT-NONFULL(r, k)
```

Lines 1-7 of B-Tree-Insert all take up constant time O(1). The different implementations of Allocate-Node are both O(1) time since assigning values to variables takes up O(1) time and the vector initialized originally is NULL.

Lines 8-10 is where we can see the differences in implementation. The code for B-Tree-Split-Child(s,1) called in line 8 is as follows:

```
B-TREE-SPLIT-CHILD(x, i)
1  z = ALLOCATE-NODE()
2  y = x.c_i
3  z.leaf = y.leaf
4  z.n = t - 1
5  for j = 1 to t - 1
6      z.key_j = y.key_{j+t}
7  if not y.leaf
8      for j = 1 to t
9          z.c_j = y.c_{j+t}
10 y.n = t - 1
11 for j = x.n + 1 downto i + 1
12     x.c_{j+1} = x.c_j
13 x.c_{i+1} = z
14 for j = x.n downto i
15     x.key_{j+1} = x.key_j
16 x.key_i = y.key_t
17 x.n = x.n + 1
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

As stated above, when a vector overflows, it is copied into another memory location. Because the number of children of a B-tree Node can range from t to 2t and the number of keys in a B-tree Node can range from t-1 to 2t-1, we can say that this copying operation takes O(t) time for the implementation of `ALLOCATE-NODE` that uses a vector, where t is the min-degree of the B-tree. On the other hand, because the array is intiialized with size 2t-1, inserting into an array will consistently take O(1) time. This is reflected in lines 9, 12 and 15 of B-TREE-SPLIT-CHILD. If pushing a key or a child rightwards overflows the vector, it will take O(t) time, while lines 9, 12 and 15 will only take O(1) time if implemented with the array implementation.

```
B-TREE-INSERT-NONFULL(x, k)
1  i = x.n
2  if x.leaf
3      while i ≥ 1 and k < x.key_i
4          x.key_{i+1} = x.key_i
5          i = i − 1
6      x.key_{i+1} = k
7      x.n = x.n + 1
8      DISK-WRITE(x)
9  else while i ≥ 1 and k < x.key_i
10         i = i − 1
11     i = i + 1
12     DISK-READ(x.c_i)
13     if x.c_i.n == 2t − 1
14         B-TREE-SPLIT-CHILD(x, i)
15         if k > x.key_i
16             i = i + 1
17     B-TREE-INSERT-NONFULL(x.c_i, k)
```

The while loop from lines 3-5 also follow this same logic. if $x.key_{i+1}$ is larger than the current size of the vector, this will overflow the vector and cause it to dynamically copy itself to a new memory location. This copying will take O(t) time, while an array initialized first with the size will take O(1) time to insert into.

Therefore, with both implementations, it will take O(th) time asymptotically, but the multiple on t will be a lot larger if using the vector version of the implementation. Thus, while the two may seem the same in asymptotic analysis, the array implementation that initializes 2t-1 sized arrays will be a lot faster in real life.