Homework 8 due Tues 11/16

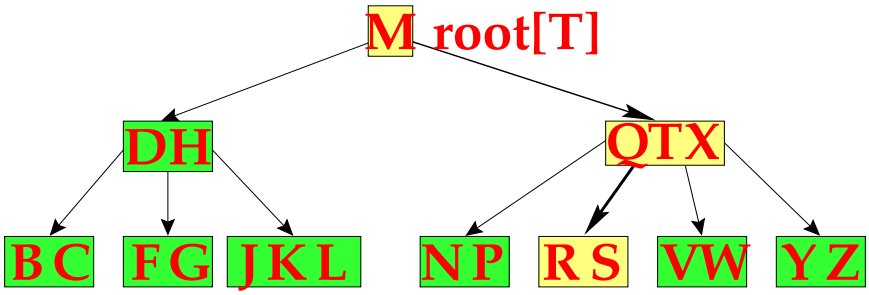
- CLRS 18.1-5 (red-black vs. BTrees)
- CLRS 18.2-6 (complexity in t)

Chapter 18: B-Trees

A B-tree is a balanced tree scheme in which balance is achieved by permitting the nodes to have multiple keys and more than two children.

Definition Let $t \geq 2$ be an integer. A tree T is called a **B-tree** having **minimum degree** t if the leaves of T are at the same depth and each node u has the following properties:

1. u has at most $2t - 1$ keys.
2. If u is not the root, u has at least $t - 1$ keys.
3. The keys in u are sorted in the increasing order.
4. The number of u's children is precisely one more than the number of u's keys.
5. For all $i \geq 1$, if u has at least i keys and has children, then every key appearing in the subtree rooted at the i-th child of u is less than the i-th key and every key appearing in the subtree rooted at the $(i + 1)$-st child of u is greater than the i-th key.


M root[T]
DH
QTX
B C
F G
J K L
N P
R S
V W
Y Z

Notation

Let u be a node in a B-tree. By $n[u]$ we denote the number of keys in u. For each i, $1 \leq i \leq n[u]$, $key_i[u]$ denotes the i-th key of u. For each i, $1 \leq i \leq n[u] + 1$, $c_i[u]$ denotes the i-th child of u.

Terminology

We say that a node is **full** if it has $2t - 1$ keys and we say that a node is **lean** if it has the minimum number of keys, that is $t - 1$ keys in the case of a non-root and 1 in the case of the root.

2-3-4 Trees

B-trees with maximum degree 2 are called **2-3-4 trees** to signify that the number of children is two, three, or four.

Depth of a B-tree

Theorem A Let $t \geq 2$ and n be integers. Let T be an arbitrary B-tree with minimum degree t having n keys. Let h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$.

Proof The height is maximized when all the nodes are lean. If T is of that form, the number of keys in T is

$$1 + \sum_{i=1}^{h} 2(t-1)t^{i-1} = 2(t-1)\frac{t^h - 1}{t-1} + 1 = 2t^h - 1.$$

Thus the depth of a B-tree is at most $\frac{1}{\lg t}$ of the depth of an RB-tree.

Searching for a key k in a B-tree

Start with $x = root[T]$.

1. If $x = \mathbf{nil}$, then k does not exist.
2. Compute the smallest i such that the i-th key at x is greater than or equal to k.
3. If the i-th key is equal to k, then the search is done.
4. Otherwise, set x to the i-th child.

The number of examinations of the key is

$$O((2t-1)h) = O(t \log_t (n+1)/2).$$

Binary search may improve the search within a node

How do we search for a predecessor?

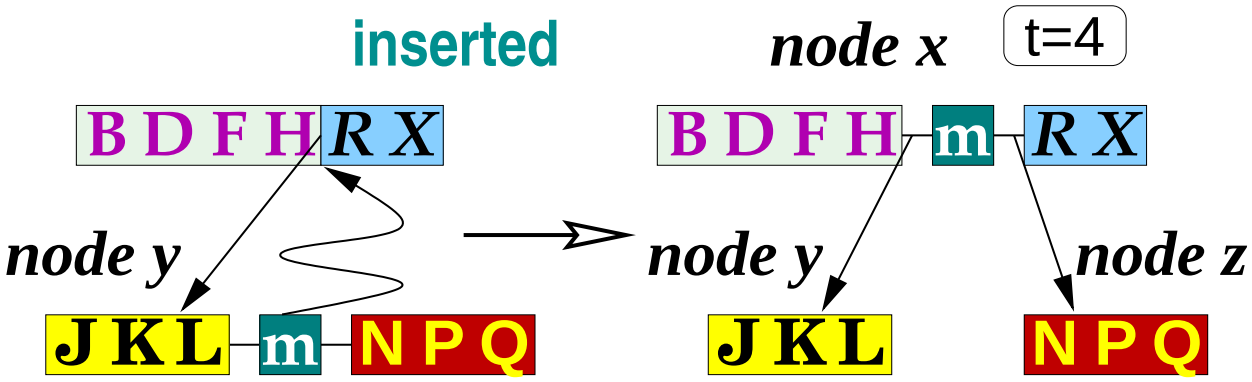
B-Tree-Predecessor(T, x, i)

```
 1: ▷ Find a pred. of key_i[x] in T
 2: if i ≥ 2 then
 3:     if c_i[x] = nil then return key_{i−1}[x]
 4: ▷ If i ≥ 2 & x is a leaf
 5: ▷ return the (i − 1)st key
 6:     else {
 7: ▷ If i ≥ 2 & x is not a leaf
 8: ▷ find the rightmost key in the i-th child
 9:         y ← c_i[x]
10:         repeat
11:             z ← c_{n[y]+1}
12:             if z ≠ nil then y ← z
13:         until z = nil
14:         return key_{n[y]}[y]
15:     }
```

```
16: else {
17: ▷ Find y and j ≥ 1 such that
18: ▷ x is the leftmost key in c_j[y]
19:     while y ≠ root[T] and c_1[p[y]] = y do
20:         y ← p[y]
21:     j ← 1
22:     while c_j[p[y]] ≠ y do j ← j + 1
23:     if j = 1 then return "No Predecessor"
24:     return key_{j−1}[p[y]]
25: }
```

Basic Operations, Split & Merge

Split takes a full node x as part of the input. If x is not the root, then its parent should not be full. The input node is split into three parts: the middle key, a node that has everything to the left of the middle key, and a node that has everything to the right of the middle key. Then the three parts replace the pointer pointing to x. As a result, in the parent node the number of children and the number of keys are both increased by one.


inserted
node x
t=4
B D F H
R X
B D F H
m
R X
node y
node y
node z
J K L
m
N P Q
J K L
N P Q

B-Tree-Split(T, x)

```
 1: if n[x] < 2t − 1 then return "Can't Split"
 2: if x ≠ root[T] and n[p[x]] = 2t − 1 then
 3:     return "Can't Split"
 4: ▷ Create new nodes y and z
 5: n[y] ← t − 1
 6: n[z] ← t − 1
 7: for i ← 1 to t − 1 do {
 8:     key_i[y] ← key_i[x]
 9:     key_i[z] ← key_{i+t}[x]
10: }
11: for i ← 1 to t do {
12:     c_i[y] ← c_i[x]
13:     c_i[z] ← c_{i+t}[x]
14: }
15: ▷ If x is the root then create a new root
16: if x = root[T] then {
17:     ▷ Create a new node v
18:     n[v] ← 0
19:     c_1[v] ← x
20:     p[x] ← v
21:     root[T] ← v
22: }
```

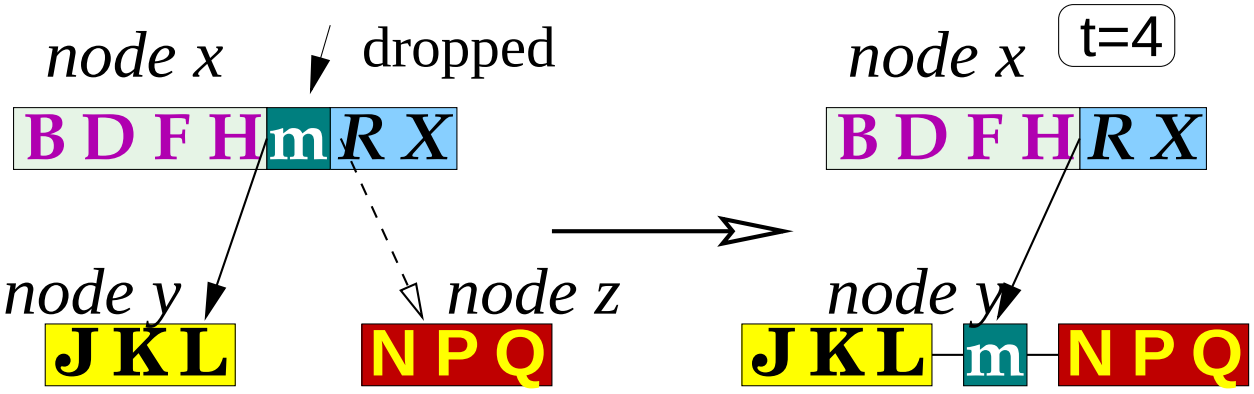
```
23: ▷ Find the spot for insertion
24: j ← 1
25: while c_j[p[x]] ≠ x do j ← j + 1
26: ▷ Open up space for insertion
27: if j ≤ n[p[x]] then
28:     for i ← n[p[x]] downto j do {
29:         key_{i+1}[p[x]] ← key_i[p[x]]
30:         c_{i+2}[p[x]] ← c_{i+1}[p[x]]
31:     }
32: ▷ Insertion
33: key_j[p[x]] ← key_t[x]
34: c_j[p[x]] ← y
35: c_{j+1}[p[x]] ← z
36: n[p[x]] ← n[p[x]] + 1
37: p[y] ← p[x]
38: p[z] ← p[x]
39: ▷ Return the pointer to the parent
40: return p[x]
```

Merge takes as input a node and the position of a key. Then it merges the key and the pair of children flanking the key into one node.

What kind of properties must the input node and the children satisfy for such an operation be possible?

What kind of properties must the input node and the children satisfy for such an operation be possible?

The input node must not be lean, and the two children must be lean.


node x
dropped
B D F H m R X
node y
J K L
node z
N P Q
node x
t=4
B D F H R X
node y
J K L
m
N P Q

B-Tree-Merge(T, x, i)

```
 1: ▷ Merge the i-th key of x and the two
 2: ▷ children flanking the i-th key
 3: y ← c_i[x]                          ▷ y is the left child
 4: z ← c_{i+1}[x]                      ▷ z is the right child
 5: if (n[y] > t − 1 or n[z]] > t − 1) then
 6:     return "Can't Merge"
 7: key_t[y] ← key_i[x]                 ▷ Append the middle key
 8: for j ← 1 to t − 1 do               ▷ Copy keys from z
 9:     key_{t+j}[y] ← key_j[z]
10: for j ← 1 to t do {
11:     c_{t+j}[y] ← c_j[z]            ▷ Copy children from z
12:     p[c_j[z]] ← y                  ▷ Fix the parent pointers
13: }
14: n[x] ← n[x] − 1                    ▷ Fix the n-tag
15: if (n[x] = 0) then {               ▷ If x was the root
16:     root[T] ← y                    ▷ and was lean, then
17:     p[y] ← nil                     ▷ y becomes the root
18: }
```

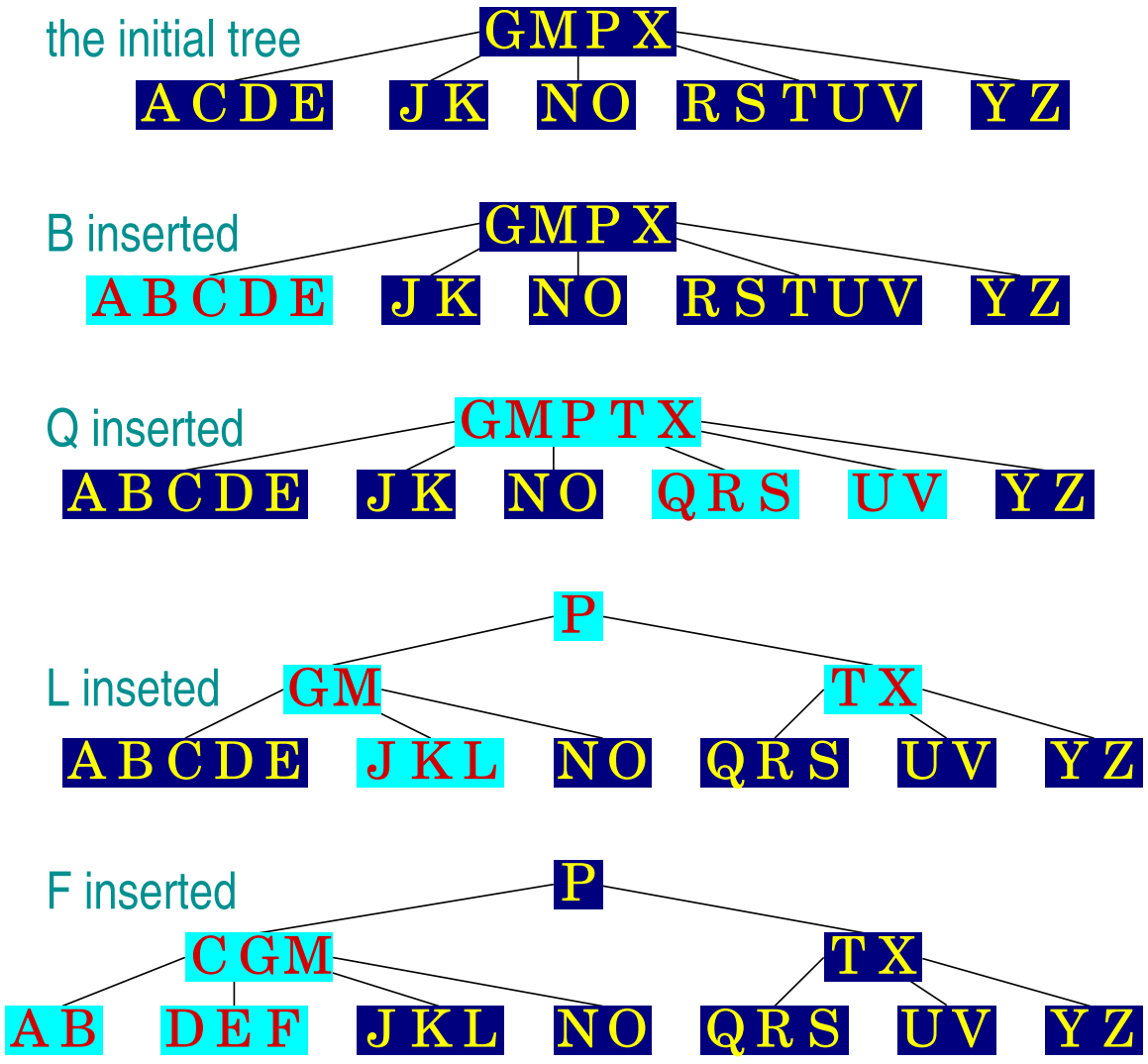
```
19: ▷ If the middle key is not the last key
20: ▷ Fill the gap by moving things
21: else if i ≤ n[x] then {
21:     for j ← i to n[x] do
22:         key_j[x] ← key_{j+1}[x]
23:     for j ← i to n[x] do
24:         c_j[x] ← c_{j+1}[x]
25: }
```

Insertion of a key

Suppose that a key k needs to be inserted in the subtree rooted at y in a B-tree T.

Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the subtree are full. Since visiting all the nodes in the subtree is very costly, we will make sure only that y is not full.

If y is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with y set to the child.


the initial tree
G M P X
A C D E
J K
N O
R S T U V
Y Z
B inserted
G M P X
A B C D E
J K
N O
R S T U V
Y Z
Q inserted
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z
L inseted
P
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z
F inserted
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z

```
B-Tree-Insert(T, y, k)
 1: z ← y
 2: f ← false
 3: while f = false do {
 3:     if n[z] = 2t − 1 then
 4:         z ← B-Tree-Split(T, z)
 5:     j ← 1
 6:     while key_j[z] < k and j ≤ n[z] do
 7:         j ← j + 1
 8:     if c_j[z] ≠ nil then z ← c_j[z]
 9:     else f ← true
10: }
11: for i ← n[z] downto j do
12:     key_{j+1}[z] ← key_j[z]
13: key_j ← k
14: n[z] ← n[z] + 1
15: return z
```

Deletion

The task is to receive a key k and a B-tree T as input and eliminate it from T if it is in the tree. To accomplish this, we will take an approach similar to that we took for binary search trees.

- Search for k. If the node containing k is a leaf, eliminate k.
- Otherwise, search for the predecessor k in the subtree immediate to the right of k. Relocate the predecessor to the position of k.

  What should we be careful about?

We should avoid removing a key from a lean leaf.

To avoid such a case, we can take a strategy similar to that we took in Insertion, that is, when a node is about to be visited, make sure that the node is not lean.

Strategy

When a lean node x is about to be visited, do the following:

- In the case when x is not the first child, if its immediate left sibling is not lean move the last key and the last child of the sibling to x; otherwise merge the sibling, x, and the key between them into one.
- In the case when x is the first child, if its immediate right sibling is not lean move the first key and the first child of the sibling to x; otherwise merge the sibling, x, and the key between them into one.

We can then assume that if x is not the root then its parent is not lean.

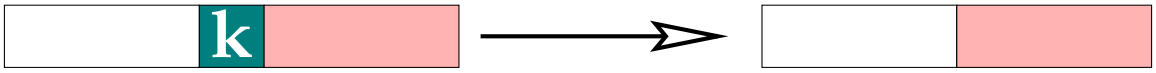
As we go down the tree, at each level there are three cases:

1. found k, this is a leaf

2. found k, this is not a leaf

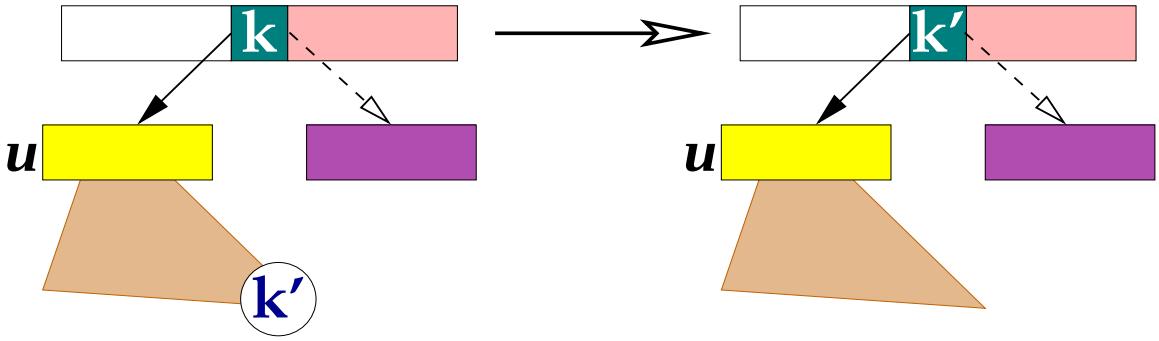
3. did not find k

Case 1: k is in a leaf node: remove k

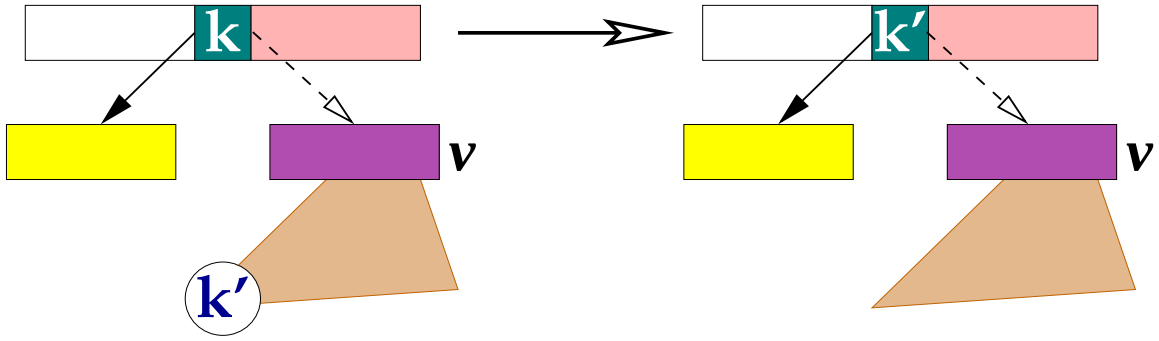


Case 2: k is in an internal node:

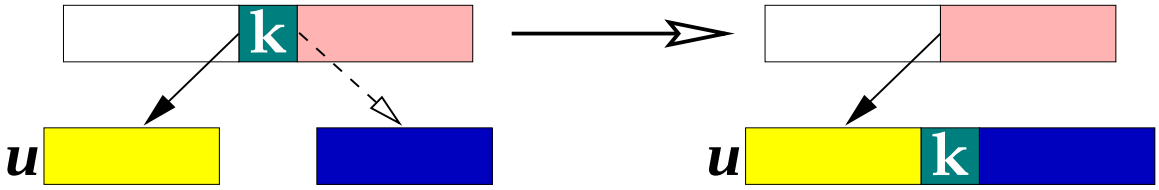
2a: left child has at least t nodes: take predecessor



2a: right child has at least t nodes: take successor

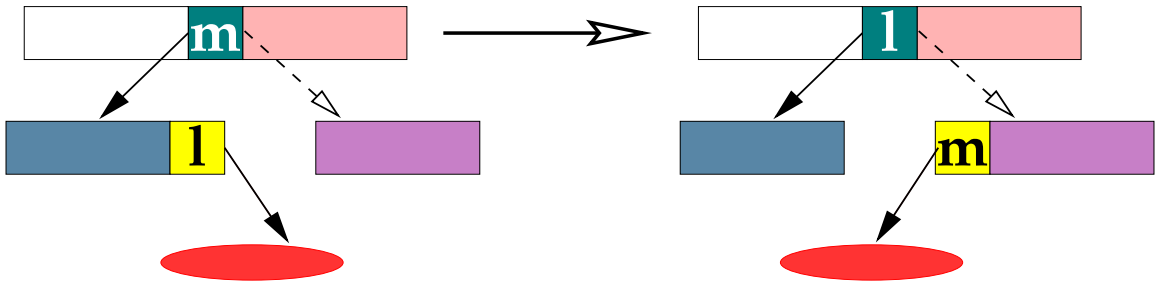


2c: both children are lean: merge children

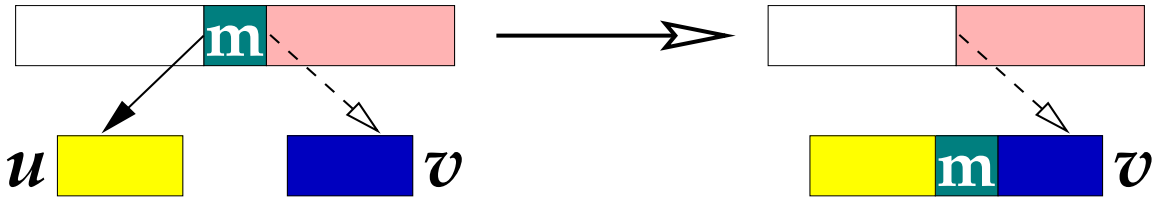


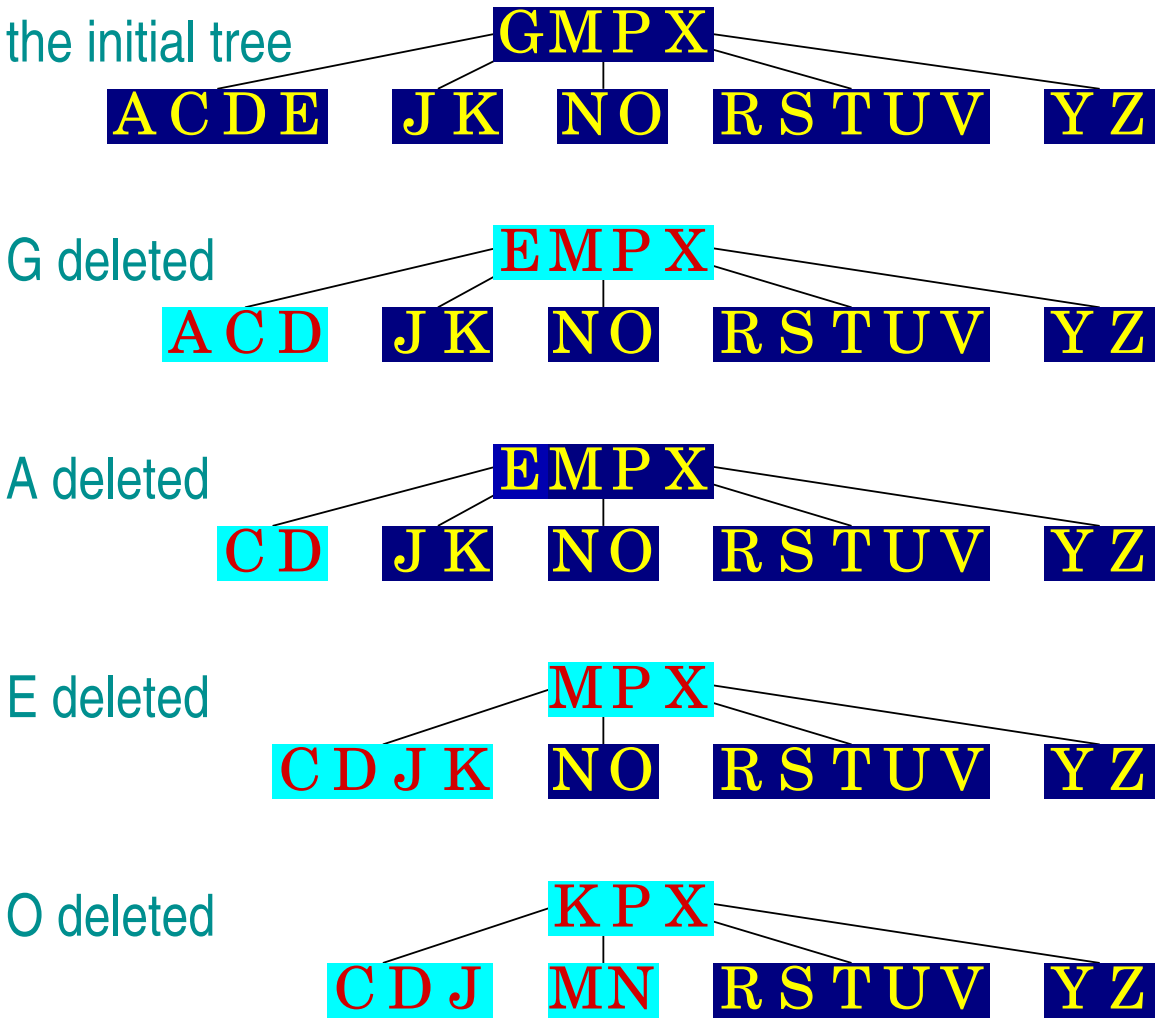
Case 3: not found yet: find subtree containing k

Case 3a: either sibling has at least t keys: move key



Case 3b: both siblings lean: merge with one sibling




the initial tree
G M P X
A C D E
J K
N O
R S T U V
Y Z
G deleted
E M P X
A C D
J K
N O
R S T U V
Y Z
A deleted
E M P X
C D
J K
N O
R S T U V
Y Z
E deleted
M P X
C D J K
N O
R S T U V
Y Z
O deleted
K P X
C D J
M N
R S T U V
Y Z