B-Trees

Chapter 18

Secondary storage

Direct-access secondary storage devices, such as disk drives, often have access times that are at least two orders of magnitude slower than memory.

To somewhat mitigate this slowness, each access to secondary storage typically reads a page of data rather than just a single value.

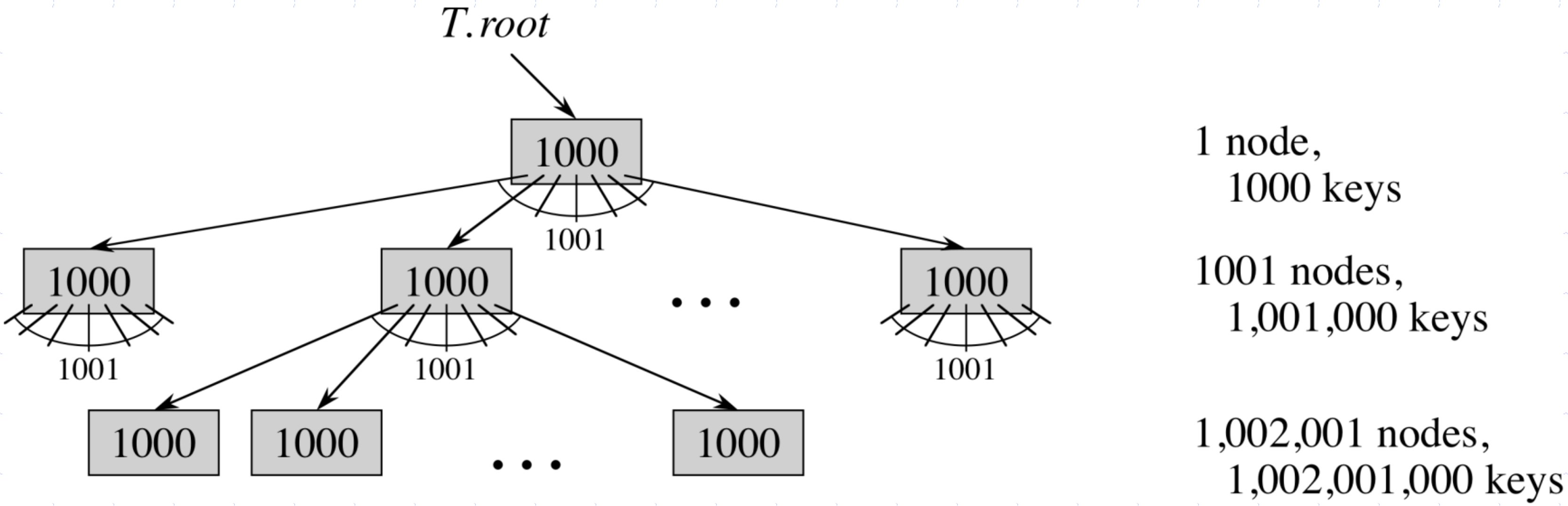
Often reading a page is slower than processing the data in the page.

Although standard analysis still applies, it is sometimes best to consider the number of accesses to the secondary storage (number of pages read), rather than elementary operations, as the measure of time.

B-Trees

B-trees are a generalization of binary trees that are good under this model. A single node of the B-tree is designed to occupy an entire page.

B-trees are used in most database systems.



B-Trees

The definition of a B-tree is quite involved:

- Every B-tree has a fixed minimum degree $t \geq 2$.
- Every node of a B-tree stores a value n, which is the number of keys currently stored in the node. $t-1 \leq n \leq 2t-1$, except for the root, where only the upper bound holds.
- Every node stores keys $key_1, key_2, \ldots, key_n$, and child pointers $c_1, c_2, \ldots, c_{n+1}$.
- The keys separate the ranges of keys stored in each subtree. If k_i is any key stored in the subtree with root c_i, then
  $k_1 \leq key_1 \leq k_2 \leq key_2 \leq \ldots \leq key_n \leq k_{n+1}$
- All leaves have the same depth.

The height of a B-tree

Theorem: For any n-key B-tree of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}$$

Proof: count nodes p_i on each level i. The root contains at least one key and other nodes contain at least t-1 keys.

$p_0 = 1$; $p_1 \geq 2$; $p_2 \geq 2t$; $p_3 \geq 2t^2$; … $p_h \geq 2t^{h-1}$. Therefore

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1, \text{or } t^h \leq \frac{n+1}{2}.$$

Search in a B-tree

```
**B-TREE-SEARCH**(k)
    i = 1
    while(i < n && k> $key_i$)
        i = i + 1
    if(i ≤ n and k = $key_i$)
        return (this, i)
    if(this is a leaf)
        return NIL
    else
        DISK-READ($c_i$)
        return $c_i$.B-TREE-SEARCH(k)
```

Again, I'm more object-oriented than the text. The text's x is the receiver (this pointer) of this method. n, key_i, and c_i are member variables.

Call root.B-TREE-SEARCH(key) to begin.

Note: this is linear search! It takes O(t) time.

Total # of DISK-READS is O(height) = $O(\log_t N)$.
Total CPU time is
O(t) * height = $O(t \log_t N)$

Creating a B-tree

```
B-TREE-CREATE()
    x = ALLOCATE-NODE()
    x.leaf = TRUE;
    x.n = 0;
    DISK-WRITE(x)
    return x;
```

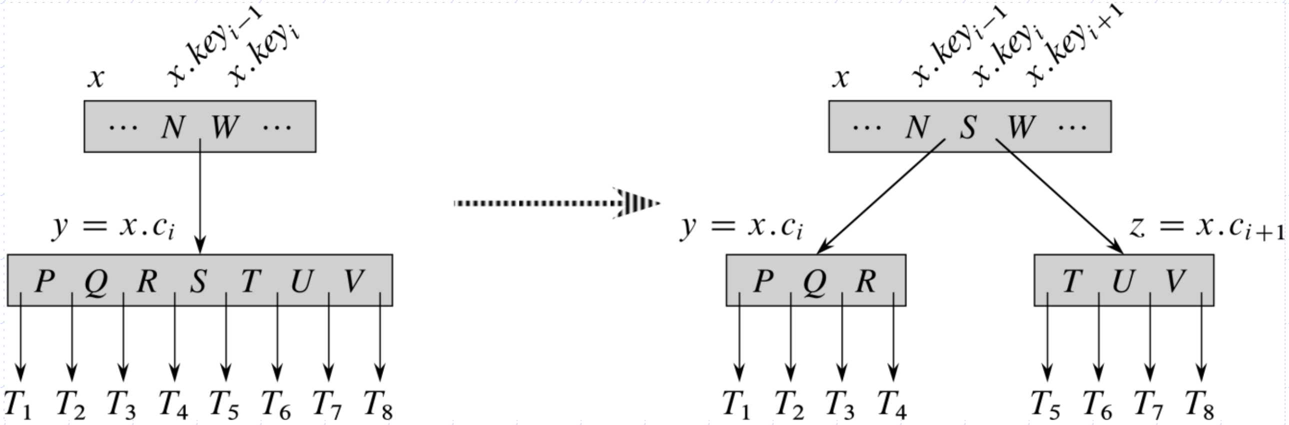
leaf is an optional member variable of a node that tells you if that node is a leaf. You can also tell by checking if the first child pointer is NULL.

```
BTree() {
    leaf = TRUE;
    n = 0;
    DISK-WRITE(this)
}
```

You can also think of B-TREE-CREATE as a constructor, where it would be written more like this.

Inserting into a B-tree

1. Do a modified search in the tree for the key. This should return a position in a leaf node.
2. Insert key at returned position.
3. If this makes the node have too many keys, split the node. This splitting inserts a key into the node's parent.
4. If the parent has too many keys, split it (and so on up the tree to the root). Splitting the root gives a new root.



Inserting into a B-tree

Analysis of this method shows that in the worst case it will have to do 2*height – 1 DISK-READ operations. This is because it searches down the tree and then splits nodes going back up the tree.

We can do better in real life (as opposed to asymptotic analysis) by splitting nodes as we go down the tree. We then reduce it to height DISK-READ operations.

The root has to be handled separately in the following algorithm, because when the root is full (2t-1 keys) we need to create a new root with two children.

Inserting into a B-tree (better)

```
B-TREE-INSERT(T, k)
    r = T.root
    if r.n = 2t -1
        s = ALLOCATE-NODE()
        T.root = s
        s.n = 0
        $s.c_1$ = r
        r.SPLIT()   <-- splits r in half, adding one child to s
        s.B-TREE-INSERT-NONFULL(k)
    else
        r.B-TREE-INSERT-NONFULL(k)
```

Inserting into a B-tree (better)

```
B-TREE-INSERT-NONFULL(k)
    if this is a leaf
        insert k in the right place in the sorted sequence $key_i$.
        n = n+1
        DISK-WRITE(this)
    else
        find i with $key_i < k < key_{i+1}$
        DISK-READ($c_{i+1}$)
        if $c_{i+1}$ is full of keys
            $c_{i+1}$.SPLIT()
            if $k > key_{i+1}$
                i = i + 1
        $c_{i+1}$.B-TREE-INSERT-NONFULL(k)
```

Again, the text's x is the receiver here.

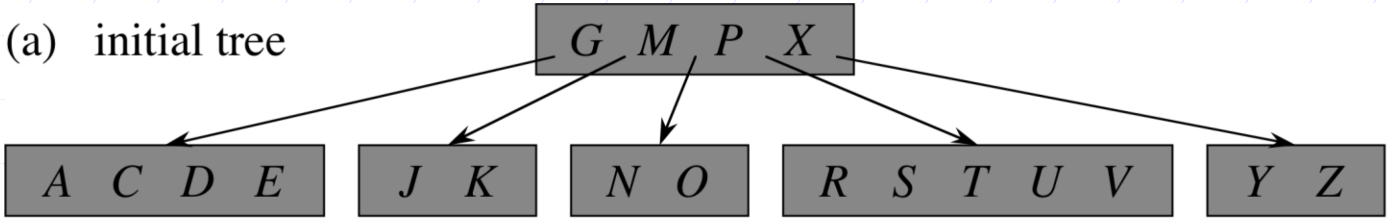
i could be 0 or n.

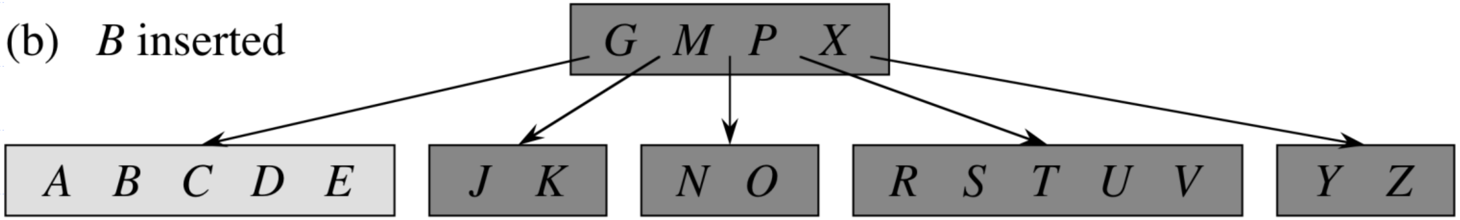
the split could have done this.

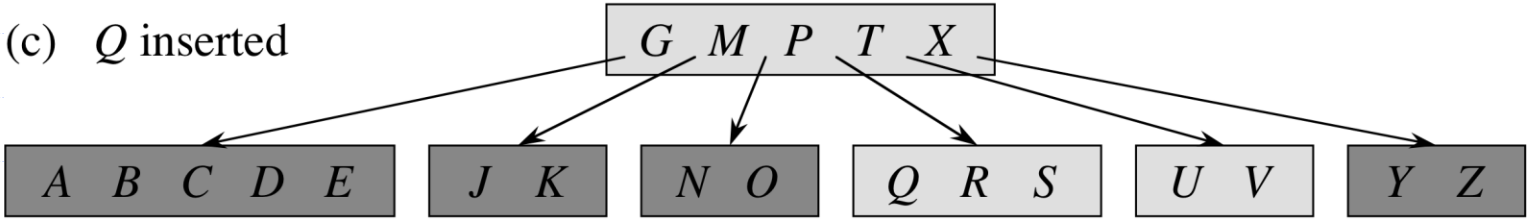
Inserting into a B-tree (better)

- B-TREE-INSERT-NONFULL never recurses on a full node.
- The number of disk accesses is O(h) for a B-Tree of height h: each call to B-TREE-INSERT-NONFULL has O(1) disk accesses. (SPLIT will require O(1) disk accesses itself).
- The total CPU time used is $O(th) = O(t \log_t N)$.

Inserting into a B-tree

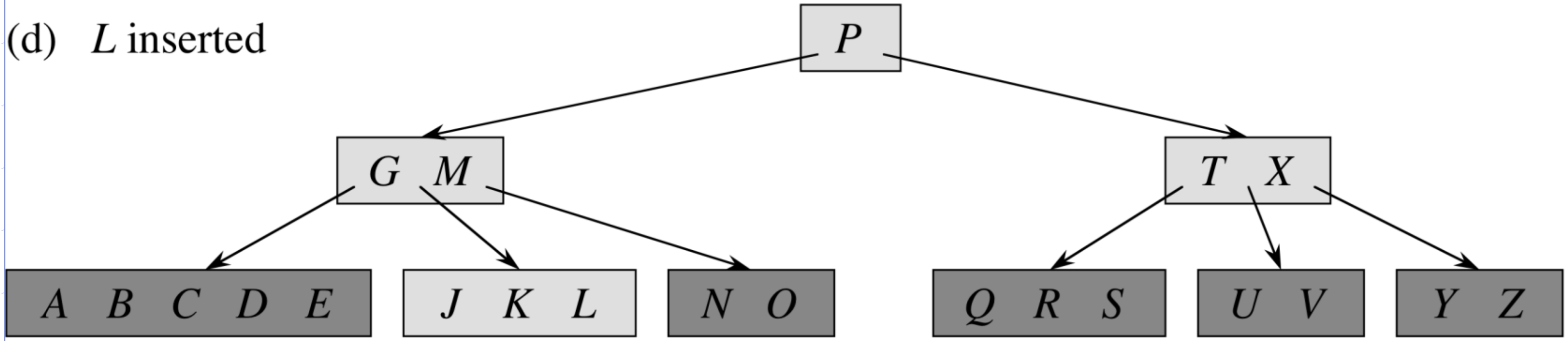

(a) initial tree


(b) *B* inserted

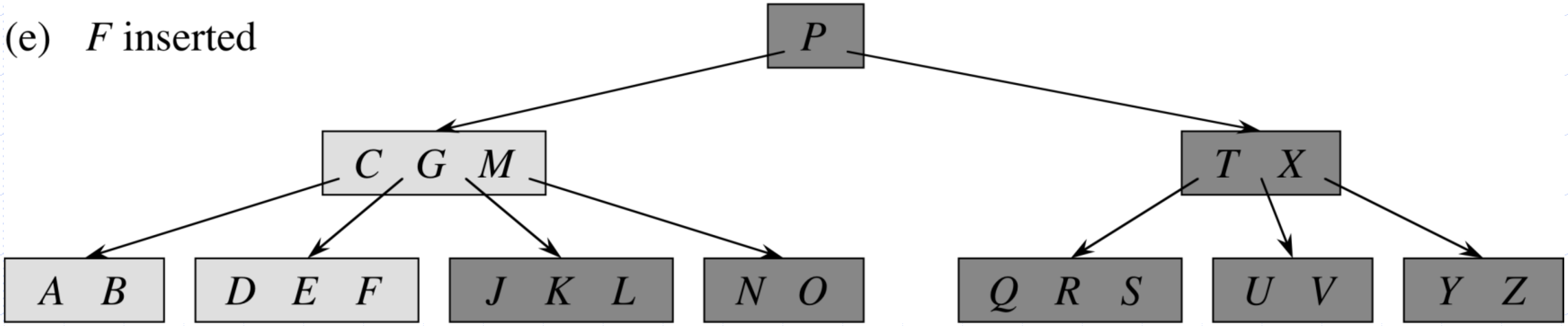

(c) *Q* inserted

Inserting into a B-tree

(d) *L* inserted



(e) *F* inserted



Deleting from a B-tree

Deletion is more complicated, because we can delete a key from any node, not just from a leaf.

Deleting from an internal node means that that node's children must be restructured.

We must ensure that a node does not contain too few keys as a result of the deletion. To do this, we never allow the procedure to step down the tree to a node with the minimum number t-1 of keys.

Deleting from a B-tree

The cases of deletion are as follows. x is the current node in our downward search of the tree for key k.

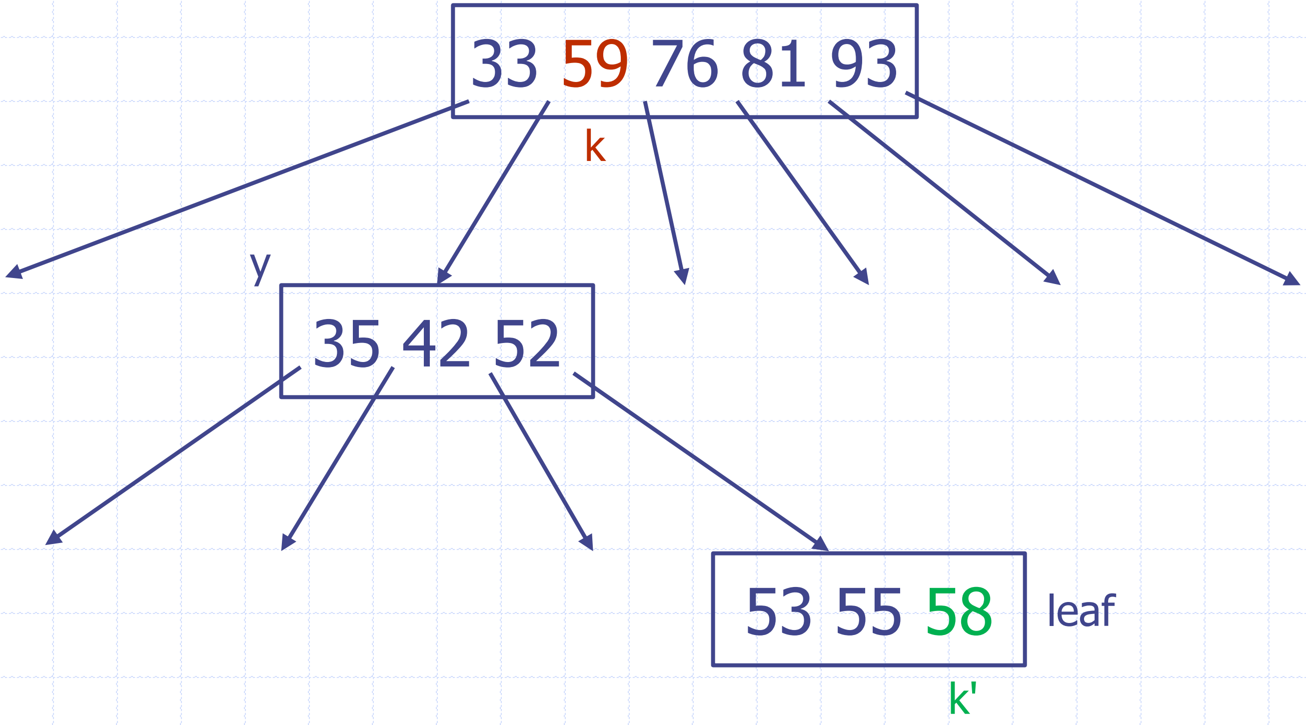
case 1: x is a leaf. Simply delete k from x.

case 2: k is in internal node x.

- 2a: If the child y that precedes k in x has ≥ t keys, find the predecessor k' of k in the subtree rooted at y. Delete k' from its position and replace k with it.

Deleting from a B-tree

t = 3



33 59 76 81 93

k

y

35 42 52

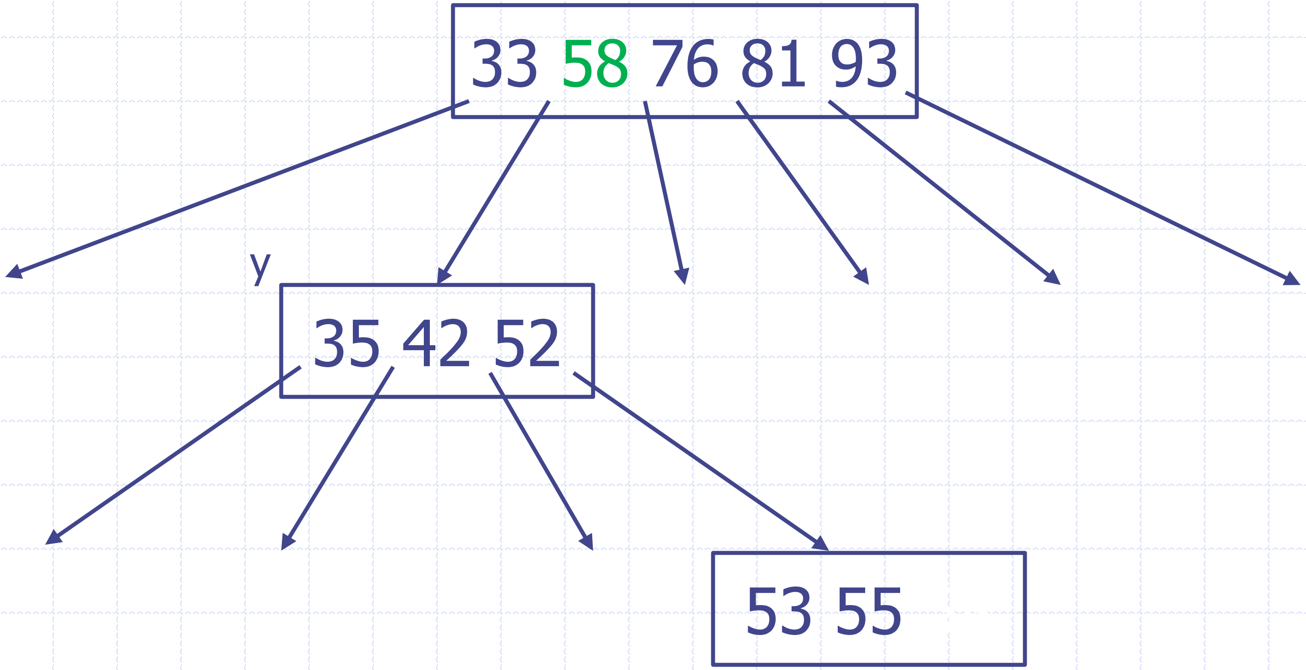
53 55 58

leaf

k'

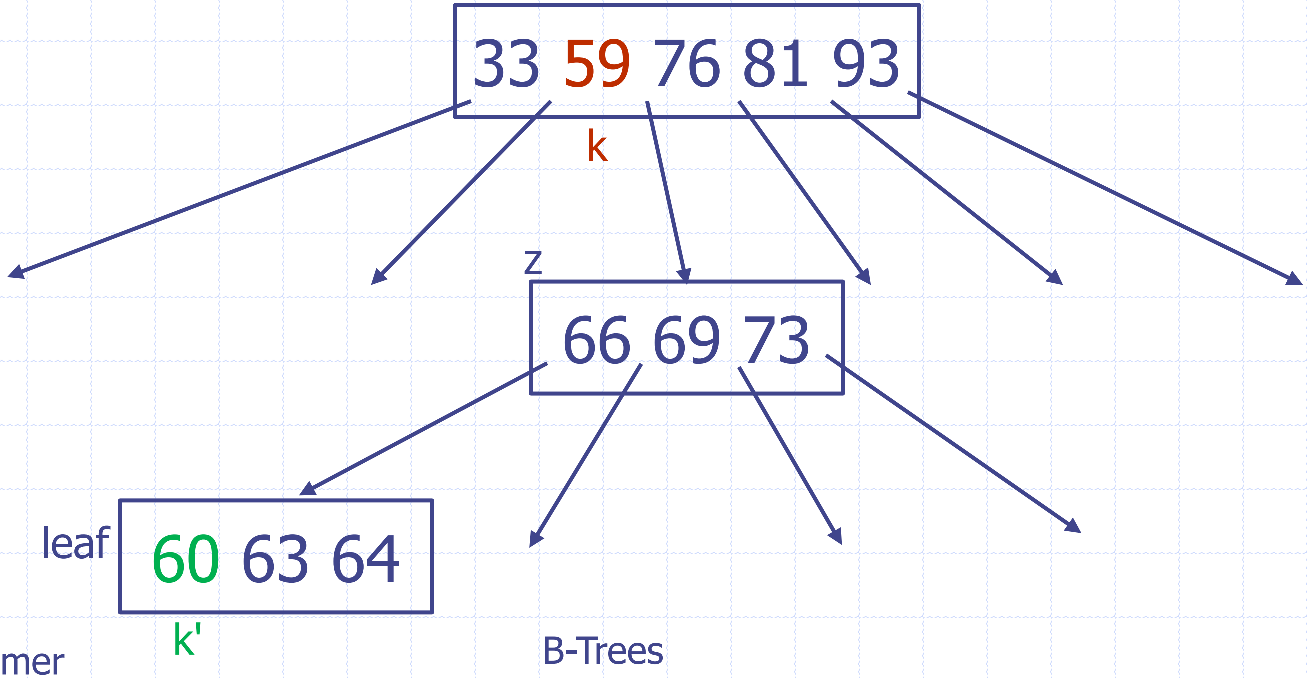
Deleting from a B-tree

t = 3



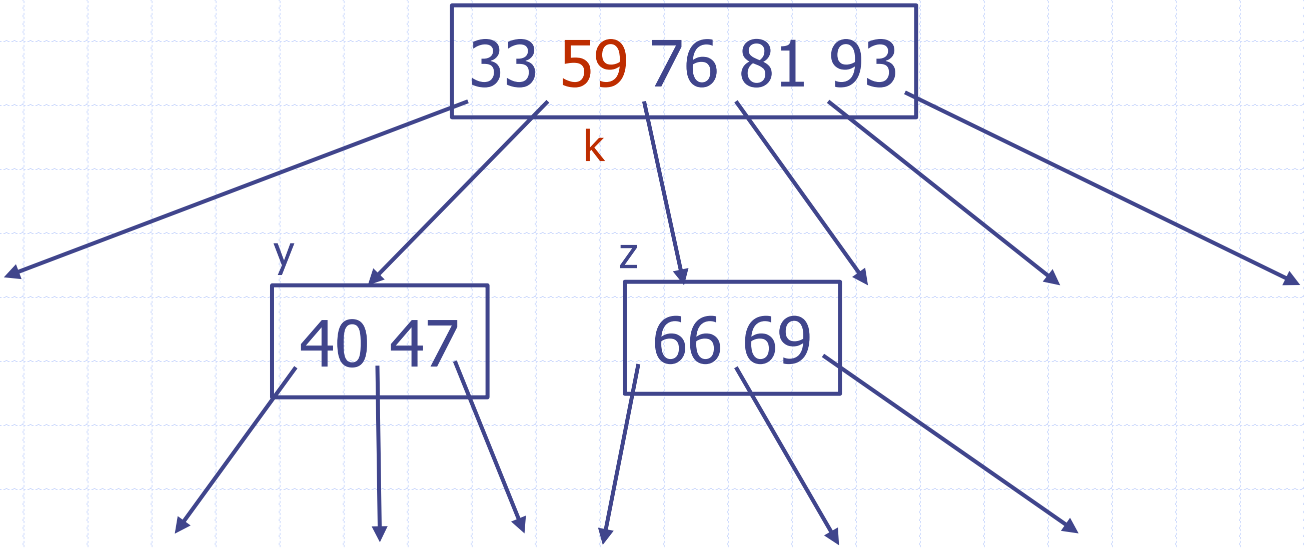
Deleting from a B-tree

- 2b: If the child z that follows k in x has ≥ t keys, find the successor k' of k in the subtree rooted at z. Delete k' from its position and replace k with it.



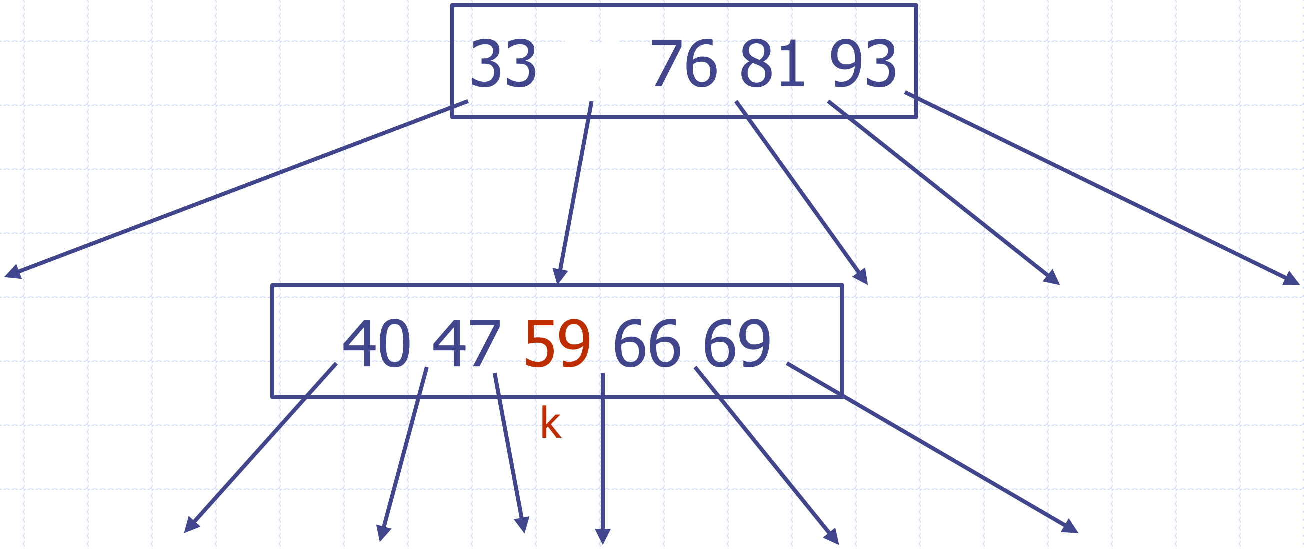
Deleting from a B-tree

- 2c: Otherwise, both y and z have t-1 keys. Delete k from x and merge y and z by putting k inbetween their keys. Recurse on this new node, deleting k.



Deleting from a B-tree

2c: Otherwise, both y and z have t-1 keys. Delete k from x and merge y and z by putting k inbetween their keys. Recurse on this new node, deleting k.

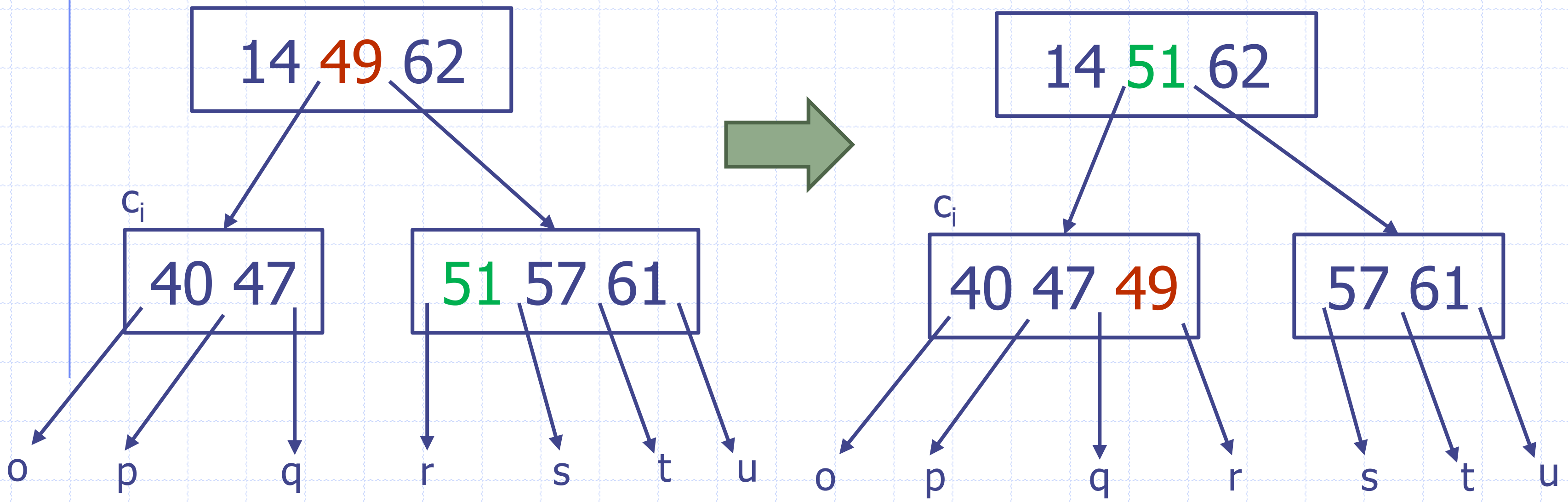


Deleting from a B-tree

case 3: k is not in internal node x. Find child c_i whose subtree must contain k. If c_i has only t-1 keys, execute 3a or 3b. Then recurse on the child containing k.

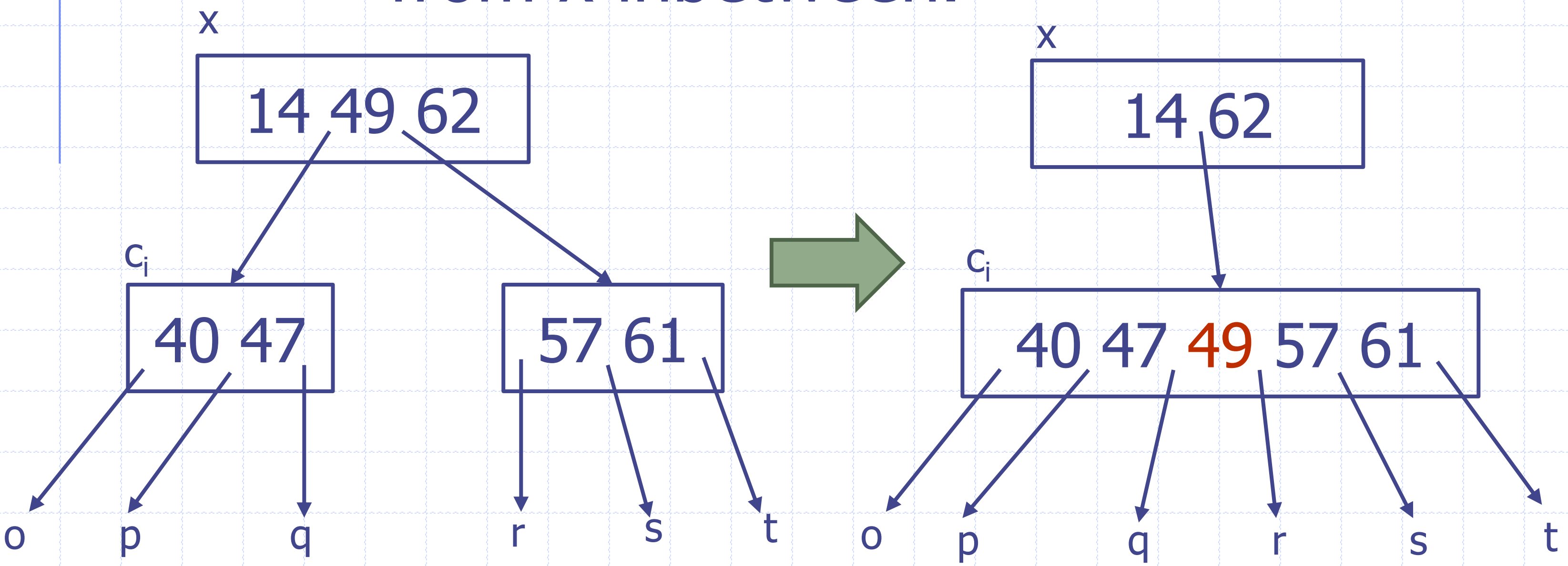
3a: If c_i has an adjacent sibling with at least t keys, then take a key from that sibling for c_i

Deleting from a B-tree



Deleting from a B-tree

3b: If c_i has a sibling with t-1 keys, merge c_i with that sibling with a key from x inbetween.



x

14 49 62

c_i

40 47

57 61

o p q r s t

x

14 62

c_i

40 47 49 57 61

o p q r s t

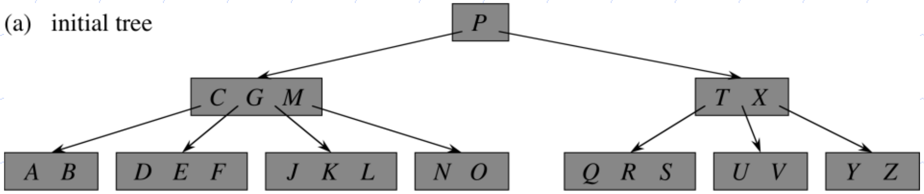
Deleting from a B-tree

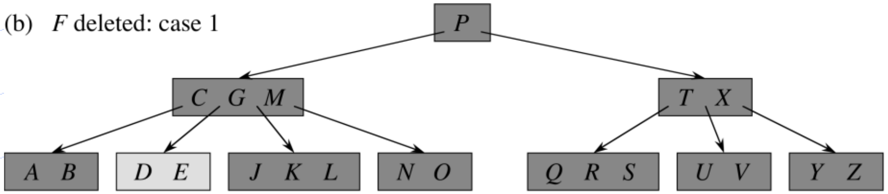
Most keys in a B-tree are stored in the leaves. This is especially true with t = 50 or 100 or 1000.

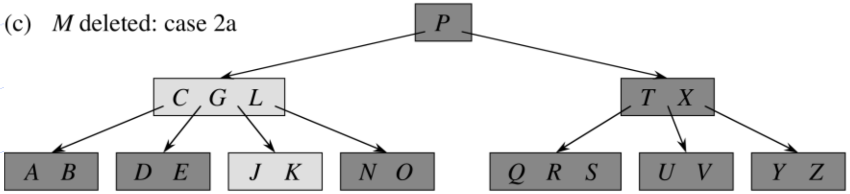
The B-TREE-DELETE just described takes one downward path in the tree when the key is stored at a leaf. In addition to the downward path, it may have to go back to the node containing the key if it is an intermediate node.

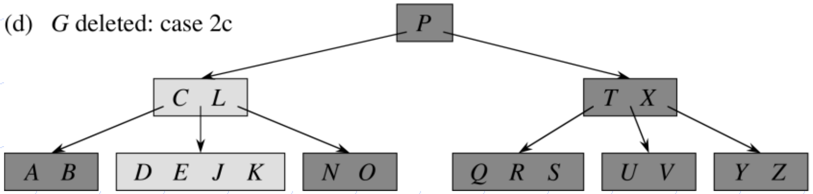
Either way the operation uses O(h) disk accesses, and $O(th) = O(t \log_t N)$ CPU time.

Deleting from a B-tree

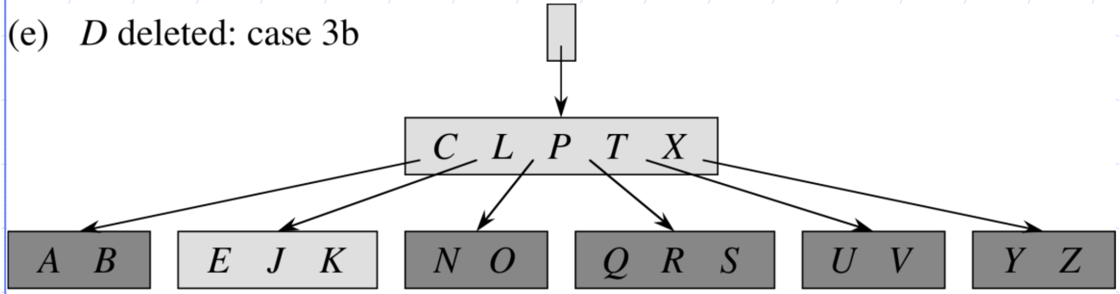

(a) initial tree


(b) *F* deleted: case 1

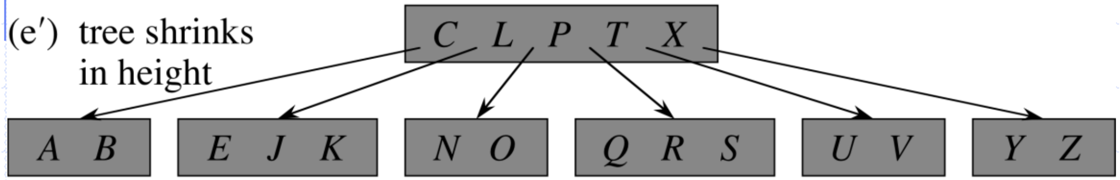

(c) *M* deleted: case 2a


(d) *G* deleted: case 2c

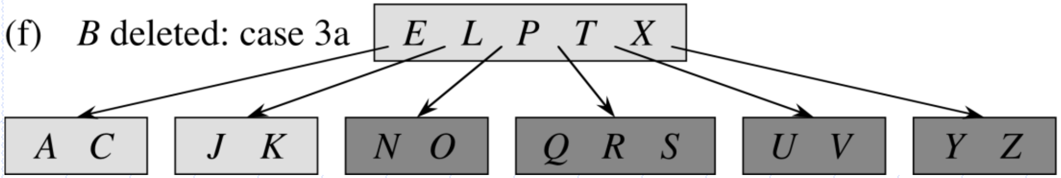
Deleting from a B-tree



(e) *D* deleted: case 3b



(e′) tree shrinks in height



(f) *B* deleted: case 3a