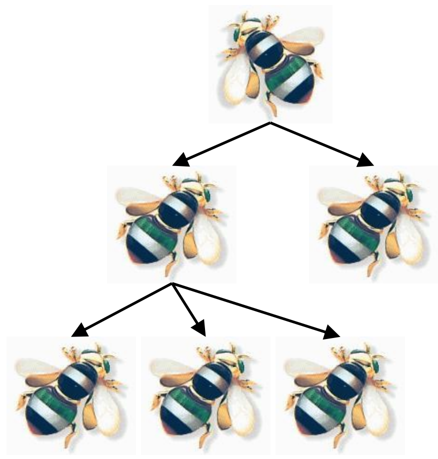
B- Trees



COL 106

Shweta Agrawal, Amit Kumar

Slide Credit : Yael Moses, IDC Herzliya

Animated demo: http://ats.oka.nu/b-tree/b-tree.html

https://www.youtube.com/watch?v=coRJrcIYbF4

Motivation

- Large differences between time access to disk, cash memory and core memory
- Minimize expensive access
 (e.g., disk access)
- B-tree: Dynamic sets that is optimized for disks

B-Trees

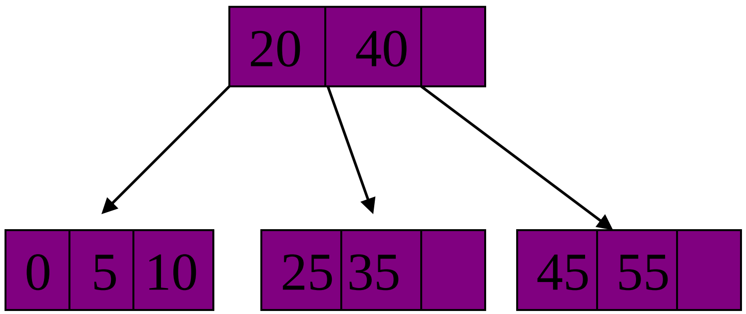
A B-tree is an M-way search tree with two properties :

1. It is perfectly balanced: every leaf node is at the same depth
2. Every internal node other than the root, is at least half-full, i.e. $M/2-1 \leq \#keys \leq M-1$
3. Every internal node with k keys has $k+1$ non-null children

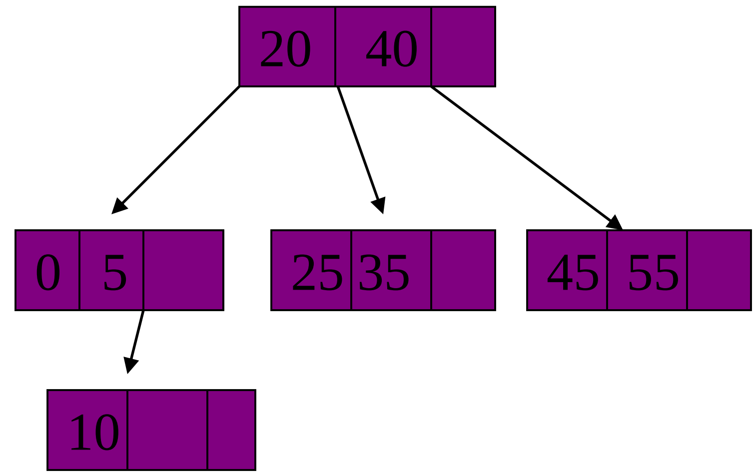
For simplicity we consider M even and we use t=M/2:

2.* Every internal node other than the root is at least half-full, i.e. $t-1 \leq \#keys \leq 2t-1$, $t \leq \#children \leq 2t$

Example: a 4-way B-tree



B-tree



4-way tree

B-tree

1. It is perfectly balanced: every leaf node is at the same depth.
2. Every node, except maybe the root, is at least half-full

   $t\text{-}1 \leq \#keys \leq 2t\text{-}1$

3. Every internal node with k keys has $k+1$ non-null children

B-tree Height

Claim: any B-tree with *n* keys, height *h* and minimum degree *t* satisfies:

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

Proof:

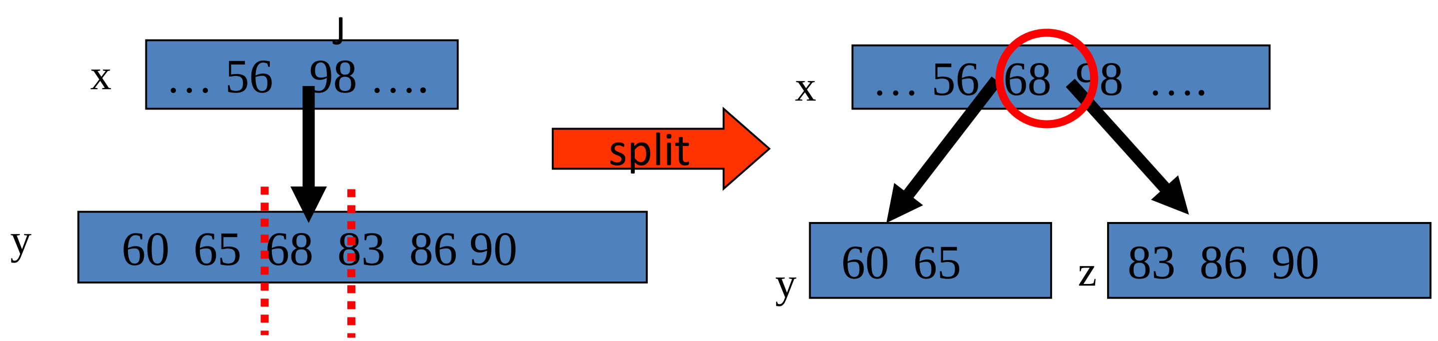
- The minimum number of KEYS for a tree with height *h* is obtained when:
  - The root contains one key
  - All other nodes contain *t-1* keys

B-Tree: Insert X

1. As in *M-way* tree find the leaf node to which *X* should be added
2. Add *X* to this node in the appropriate place among the values already there
   (there are no subtrees to worry about)
3. Number of values in the node after adding the key:
   - Fewer than *2t-1:* done
   - Equal to *2t: overflowed*
4. Fix overflowed node

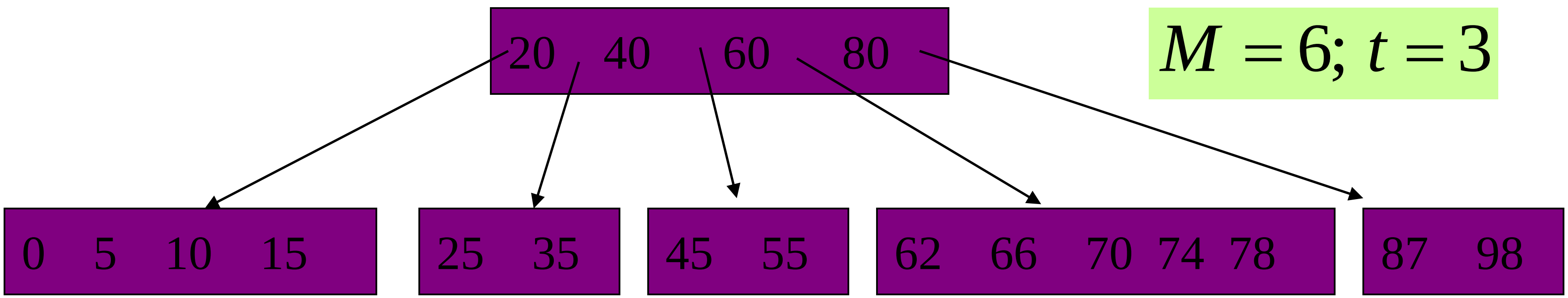
Fix an Overflowed

1. Split the node into three parts, *M=2t*:
   - Left: the first *t* values, become a left child node
   - Middle: the middle value at position *t*, goes up to parent
   - Right: the last *t-1* values, become a right child node
2. Continue with the parent:
   1. Until no overflow occurs in the parent
   2. If the root overflows, split it too, and create a new root node

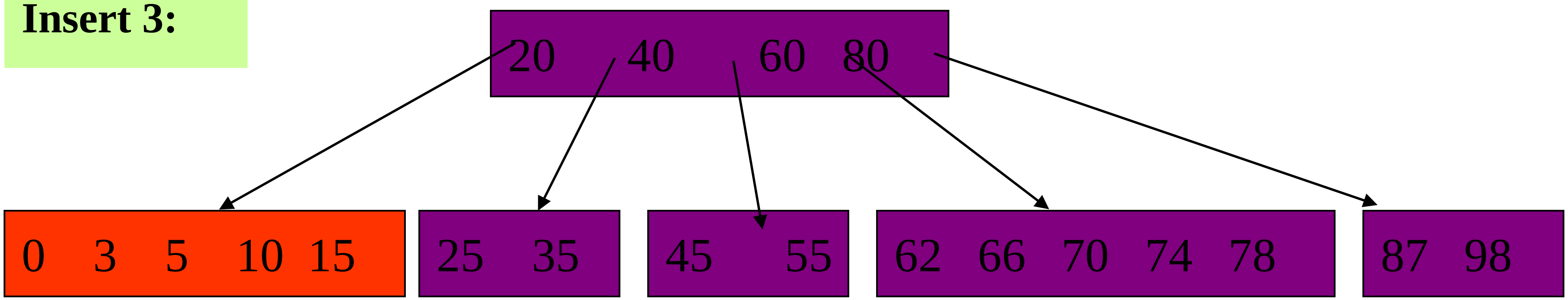


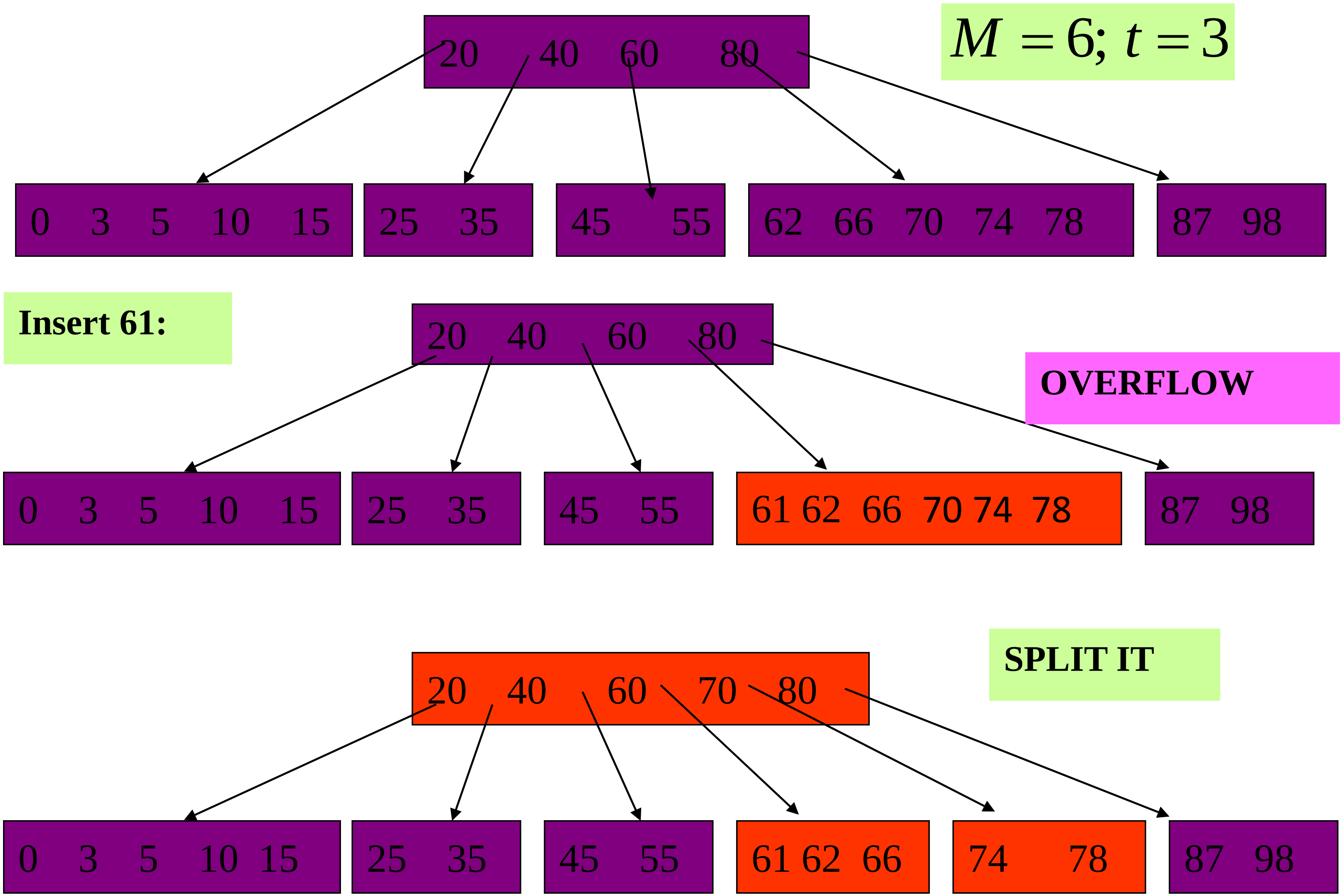
Insert example

$M = 6;\ t = 3$



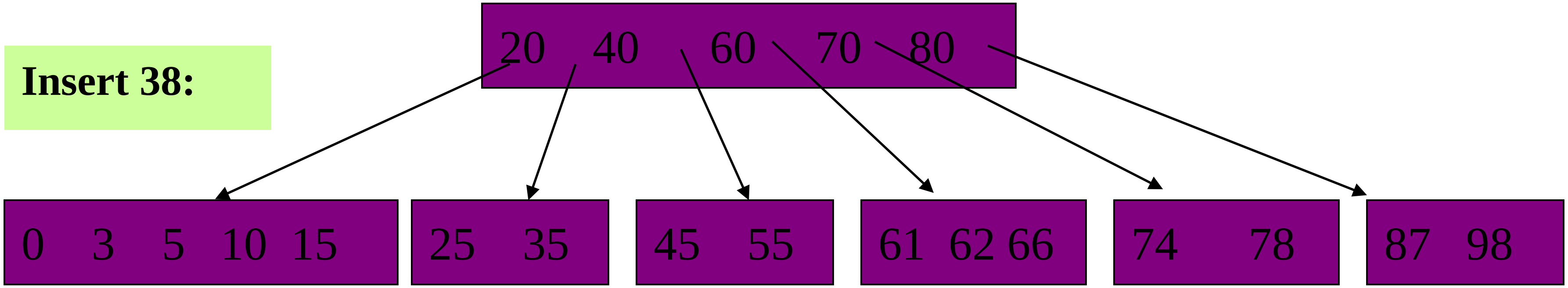
Insert 3:

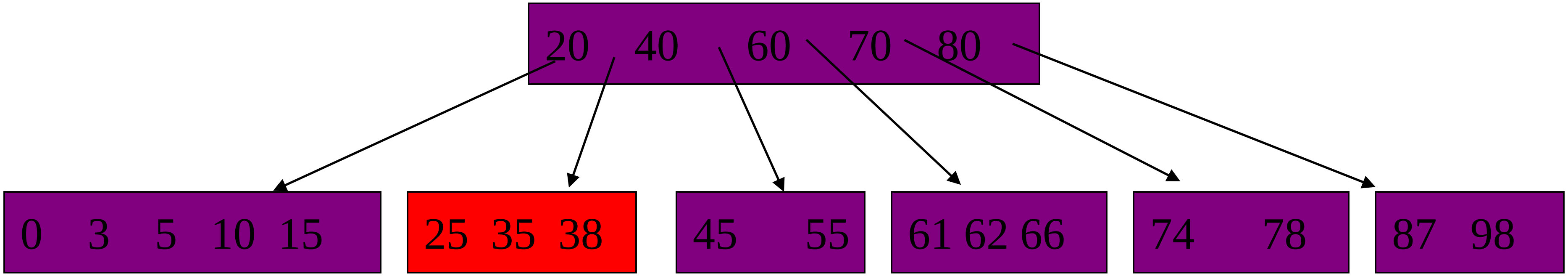



$M = 6; t = 3$
20 40 60 80
0 3 5 10 15
25 35
45 55
62 66 70 74 78
87 98
Insert 61:
20 40 60 80
OVERFLOW
0 3 5 10 15
25 35
45 55
61 62 66 70 74 78
87 98
SPLIT IT
20 40 60 70 80
0 3 5 10 15
25 35
45 55
61 62 66
74 78
87 98

$M = 6; t = 3$

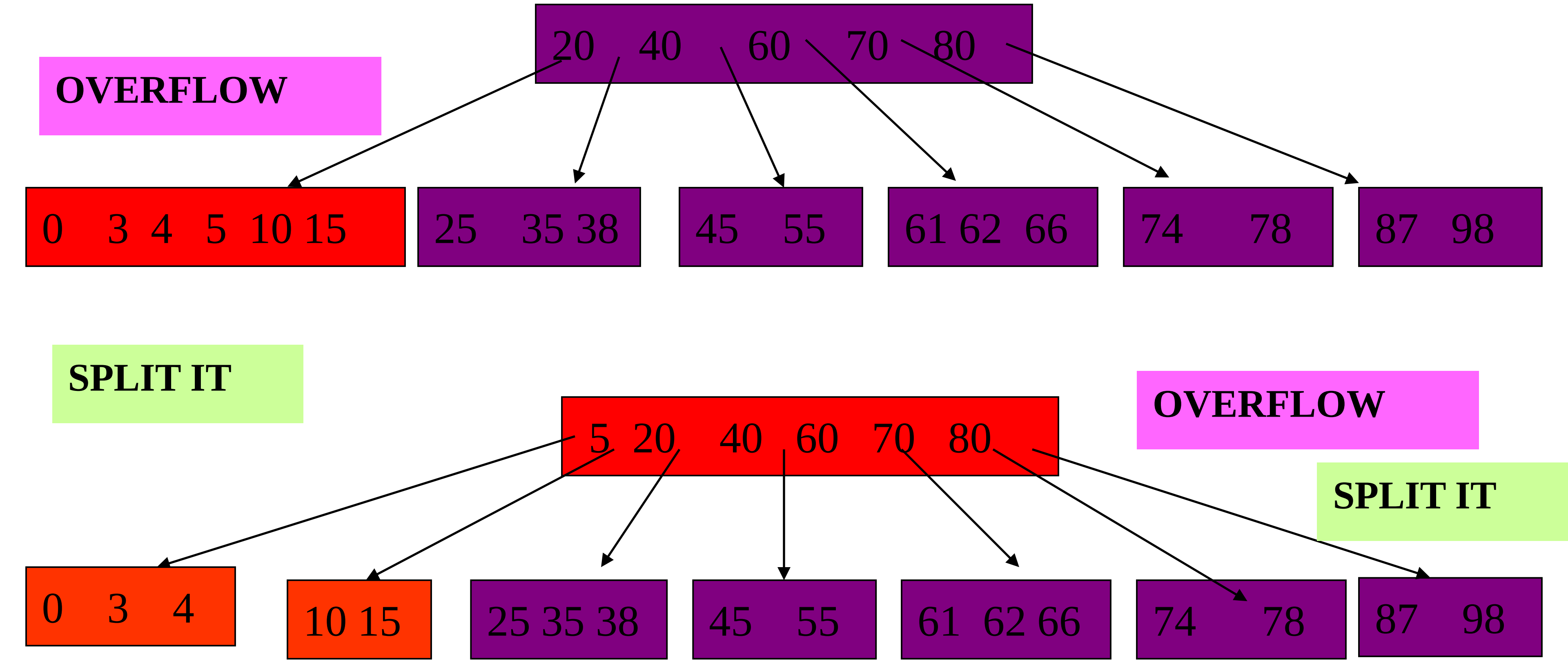
Insert 38:



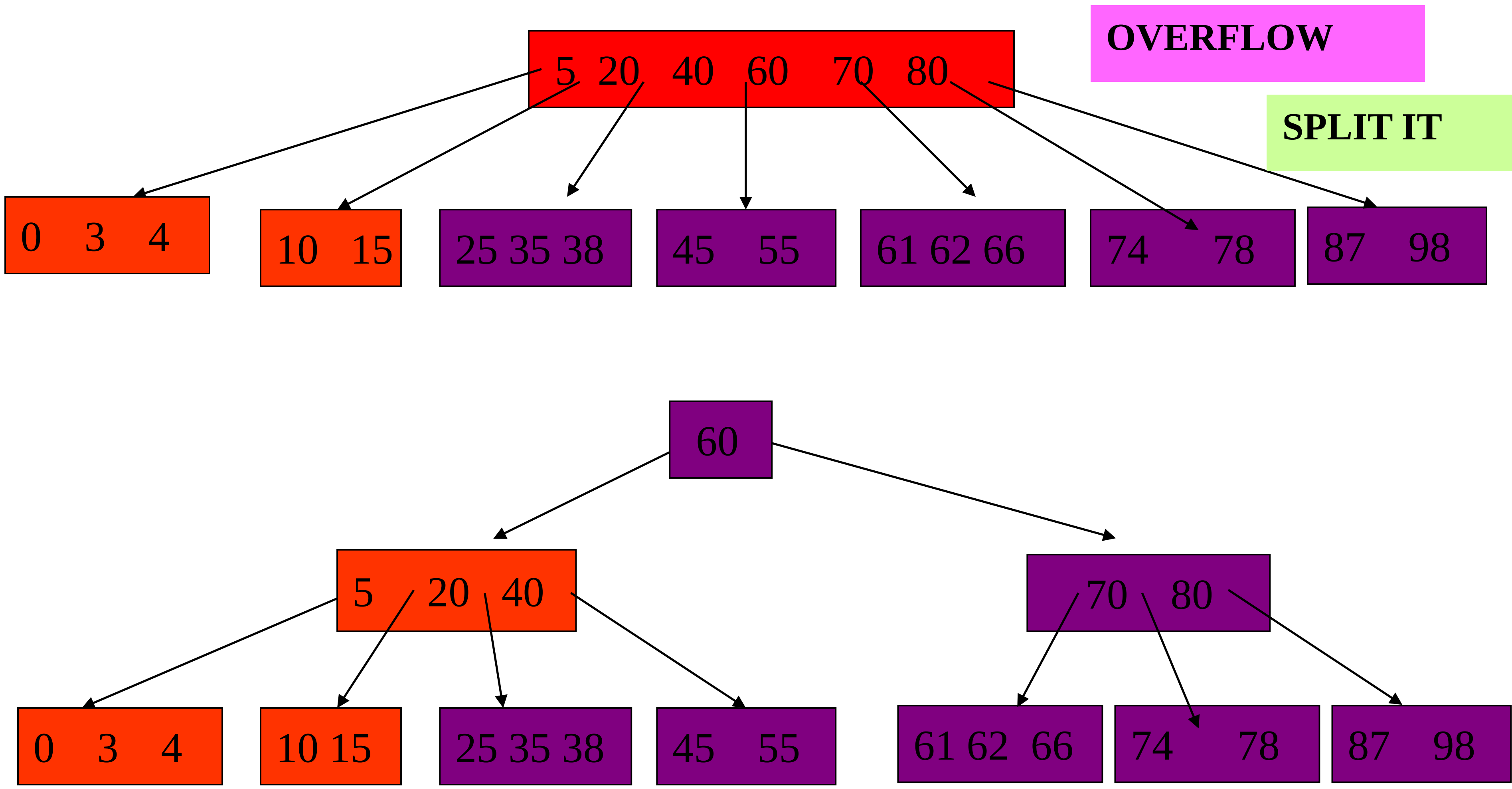


Insert 4:

$M = 6; t = 3$



$M = 6;\ t = 3$



5 20 40 60 70 80
OVERFLOW
SPLIT IT
0 3 4
10 15
25 35 38
45 55
61 62 66
74 78
87 98
60
5 20 40
70 80
0 3 4
10 15
25 35 38
45 55
61 62 66
74 78
87 98

Complexity Insert

- Inserting a key into a B-tree of height *h* is done in a single pass down the tree and a single pass up the tree

Complexity: $$O(h) = O(\log_t n)$$

B-Tree: Delete X

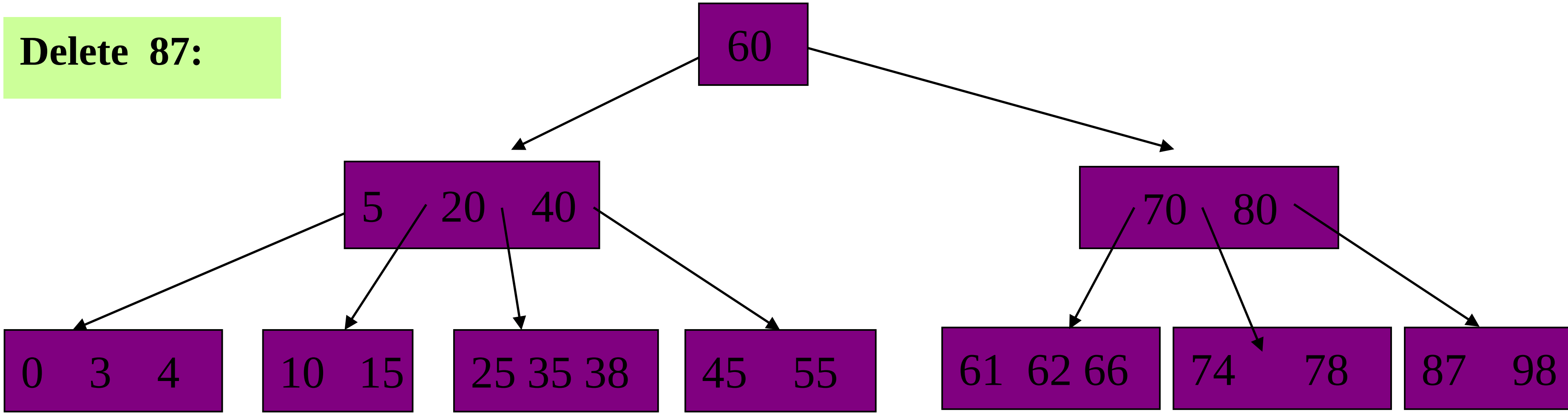
- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node < t-1

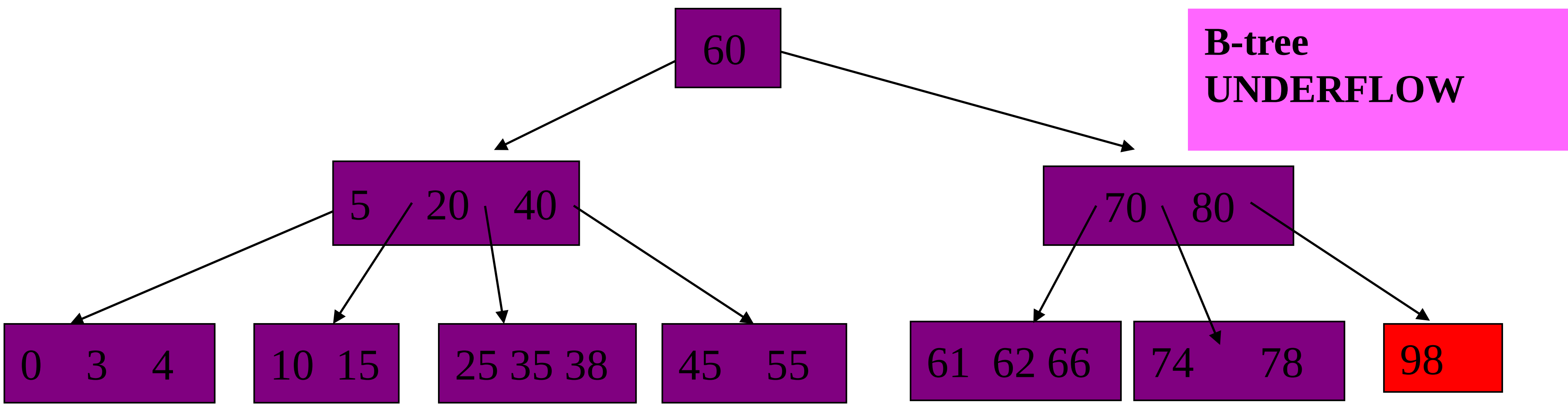
Recall: The root should have at least 1 value in it, and all other nodes should have at least t-1 values in them

$M = 6; t = 3$

Underflow Example

Delete 87:





B-tree UNDERFLOW

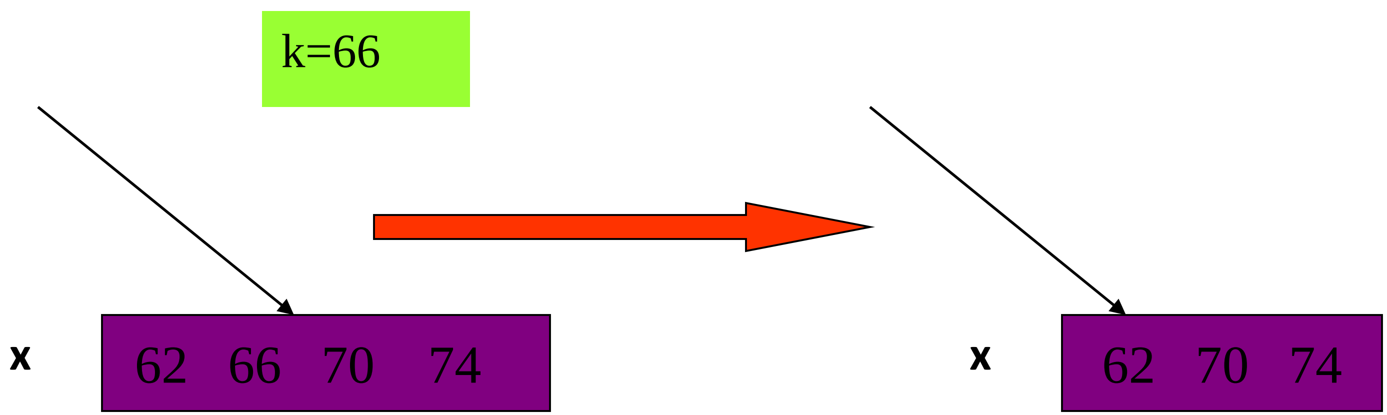
B-Tree: Delete X,k

- *Delete as in M-way tree*
- *A problem:*
  - might cause *underflow*: the number of keys remain in a node < t-1
- Solution:
  - make sure a node that is visited has at least t instead of t-1 keys.
  - If it doesn't have k
    - (1) either take from sibling via a rotate, or
    - (2) merge with the parent
  - If it does have k
    - See next slides

Recall: The root should have at least 1 value in it, and all other nodes should have at least t-1 (at most 2t-1) values in them

B-Tree-Delete(x,k)

1st case: *k* is in *x* and *x* is a *leaf* → delete *k*



How many keys are left?

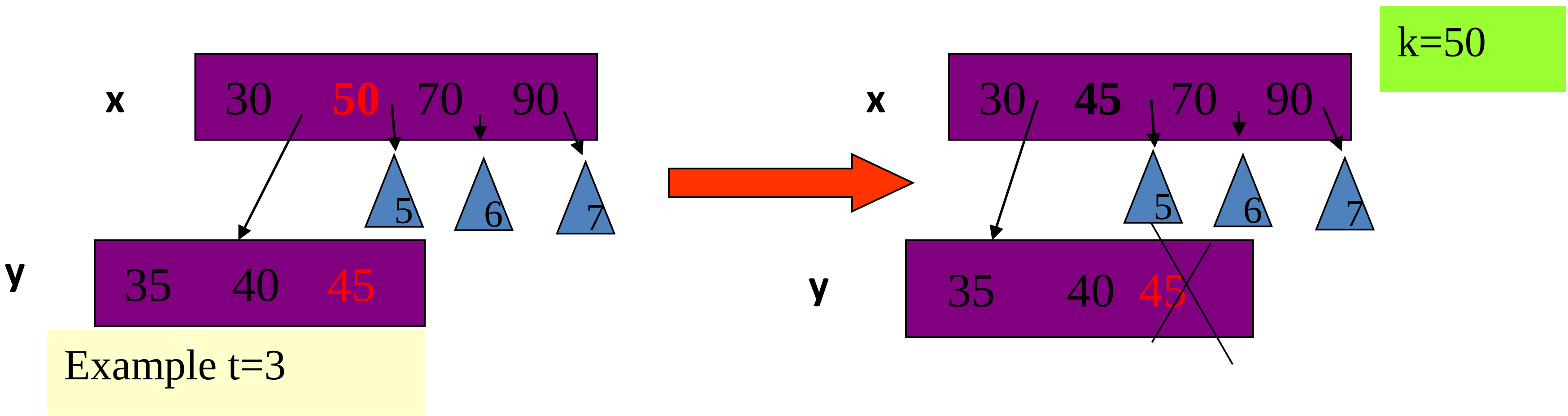
Example t=3

2nd case: *k* in the internal node *x*, *y* and z are the preceding and succeeding nodes of the key k∈ x

a. If *y* has at least *t* keys:

- ▷ Replace *k* in *x* *k'* ∈ *y*, where *k'* is the predecessor of *k* in *y*
- ▷ Delete *k'* recursively

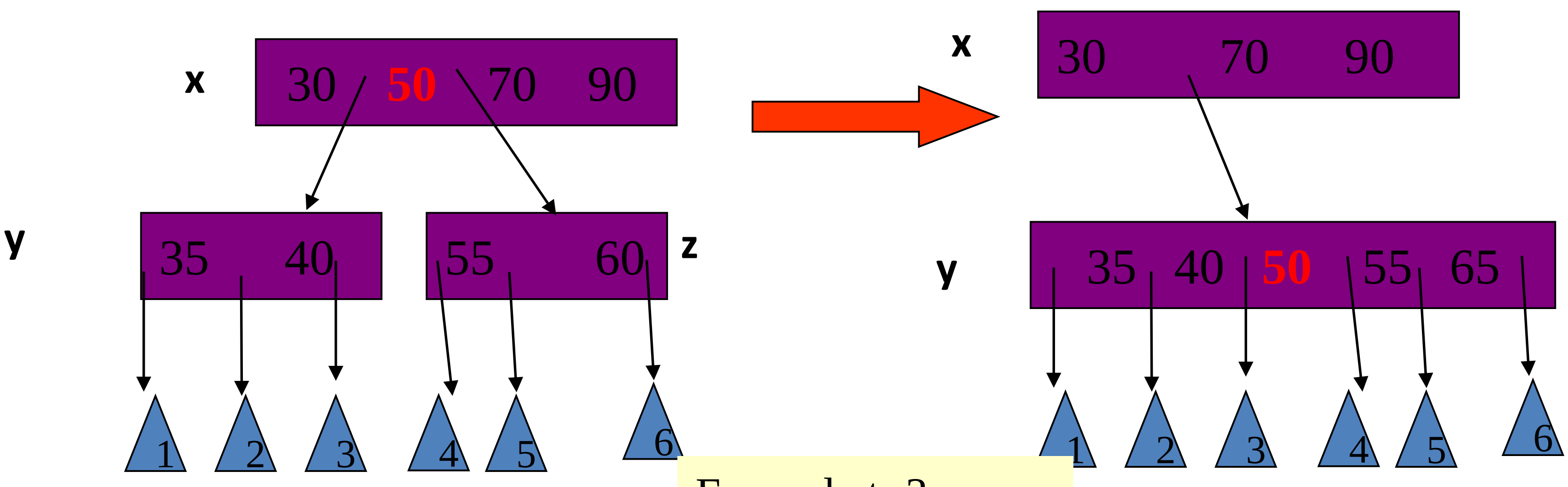
b. Similar check for successor case



2nd case cont.:

c. Both a and b are not satisfied: y and z have *t-1* keys

- Merge the two children, y and z
- Recursively delete *k* from the merged cell



Questions

- When does the height of the tree shrink?
- Why do we need the number of keys to be at least *t* and not *t-1* when we proceed down in the tree?

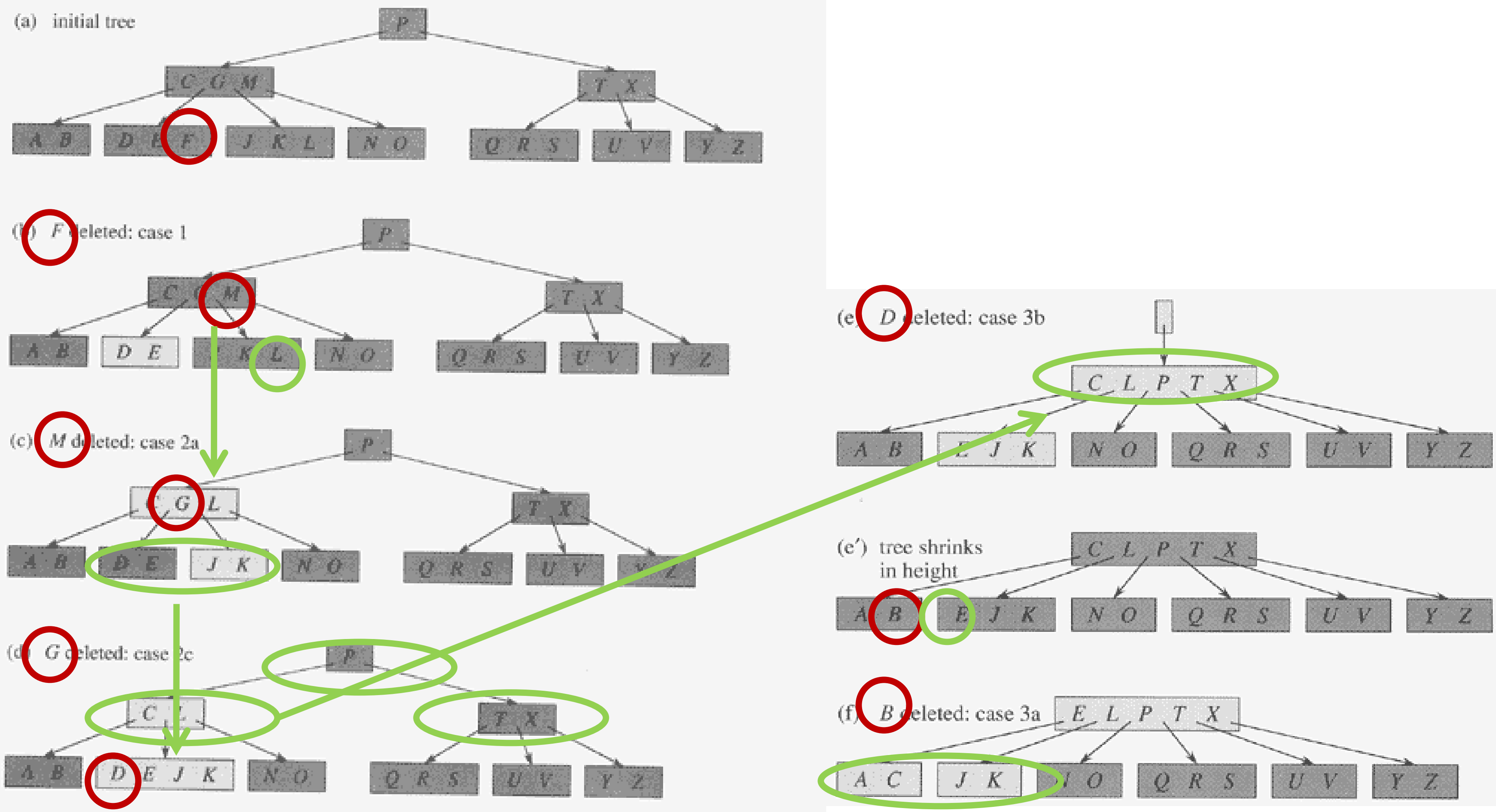


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M is moved up to take M's position. **(d)** Deletion of G. This is case 2c: G is pushed down to make node $DEGJK$, and then G is deleted from this leaf (case 1). **(e)** Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form $CLPTX$; then, D is deleted from a leaf (case 1). **(e′)** After (d), the root is deleted and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.

Delete Complexity

- Basically downward pass:
  - Most of the keys are in the leaves – one downward pass
  - When deleting a key in internal node – may have to go one step up to replace the key with its predecessor or successor

Complexity $$O(h) = O(\log_t n)$$

Run Time Analysis of B-Tree Operations

- For a B-Tree of order *M=2t*
  - #keys in internal node: *M-1*
  - #children of internal node: between *M/2* and *M*
  - ➔ Depth of B-Tree storing *n* items is $O(\log_{M/2} N)$
- Find run time is:
  - *O(log M)* to binary search which branch to take at each node, since *M* is constant it is *O(1).*
  - Total time to find an item is $O(h*\log M) = O(\log n)$
- Insert & Delete
  - Similar to find but update a node may take : *O(M)=O(1)*

Note: if M is >32 it worth using binary search at each node

A typical B-Tree

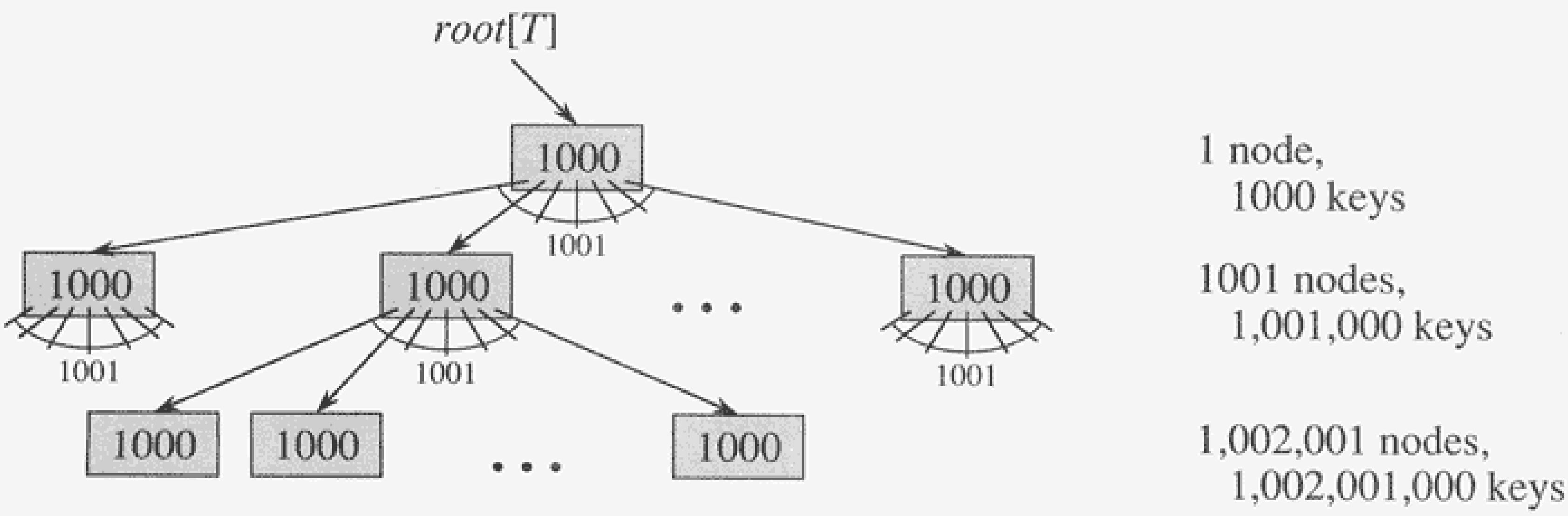


Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Why B-Tree?

- B-trees is an implementation of dynamic sets that is optimized for disks
  - The memory has an hierarchy and there is a tradeoff between size of units/blocks and access time
  - The goal is to optimize the number of times needed to access an “expensive access time memory”
  - The size of a node is determined by characteristics of the disk – block size – page size
  - The number of access is proportional to the tree depth