1.27.2017

Splitting a node in the B-tree

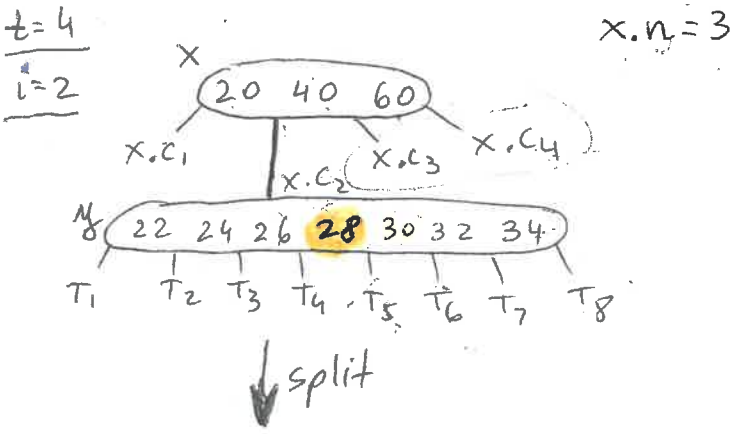
Input x - nonfull internal node (in main memory)
index i s.t. { $x.c_i$ is a full child of x
$x.c_i$ is in the main memory

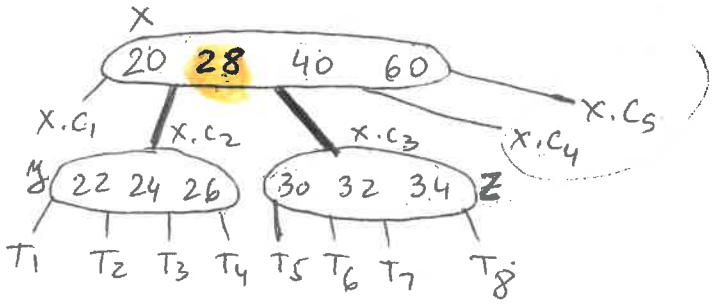
Output split node $x.c_i$ around its median key $x.c_i.key_t$

```
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                      } Θ(t)
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf                           }
 8      for j = 1 to t                      } Θ(t)
 9          z.c_j = y.c_{j+t}               }
10  y.n = t - 1
11  for j = x.n + 1 downto i + 1            } Θ(t)
12      x.c_{j+1} = x.c_j                   }
13  x.c_{i+1} = z
14  for j = x.n downto i                    } Θ(t)
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)
```

$RT = \Theta(t)$
$\Theta(1)$ disk operations

$t = 4$
$i = 2$





lines 5..6, $j = 1 \ldots 3$
$z.key_1 = y.key_5$
$z.key_2 = y.key_6$
$z.key_3 = y.key_7$

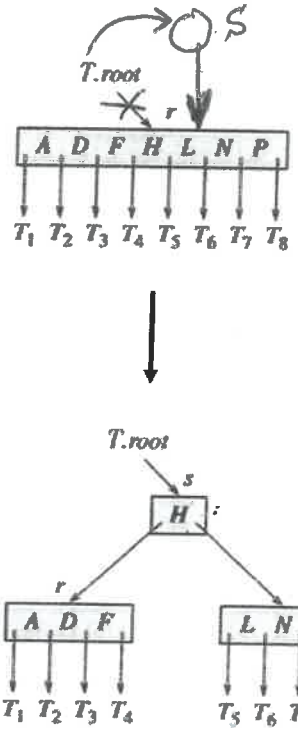
lines 7..9, $j = 1..4$
$z.c_1 = y.c_5$
$z.c_2 = y.c_6$
$z.c_3 = y.c_7$
$z.c_4 = y.c_8$

lines 11..12, $j = 4..3$
$x.c_5 = x.c_4$
$x.c_4 = x.c_3$
$x.c_3 = z$

lines 14..15, $j = 3 \ldots 2$
$x.key_4 = x.key_3$
$x.key_3 = x.key_2$
$x.key_2 = y.key_4$

• If the root r is full, then split r and a new node s becomes the root

t = 4
# keys 3...7



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.c1 = r
8       B-TREE-SPLIT-CHILD(s, 1)        O(t)
9       B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

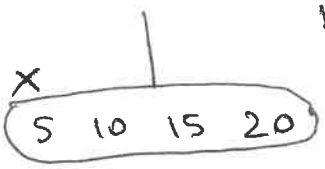
$RT = O(t \cdot h) = O(t \cdot \log_t n)$

if t = constant $\Rightarrow$ $\boxed{RT = O(\lg n)}$

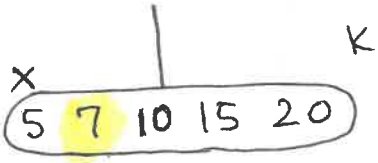
- Inserts Key k into the subtree rooted at the nonfull node x

```
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)
```

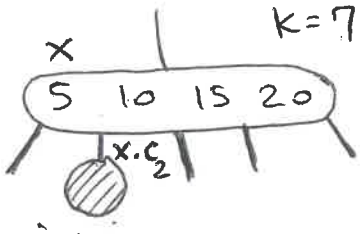
x is a leaf



i=4
...
i=1

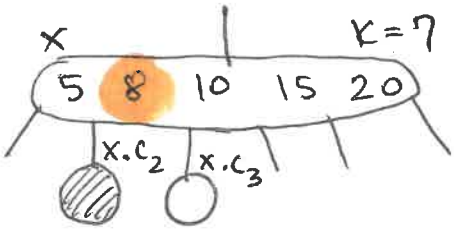


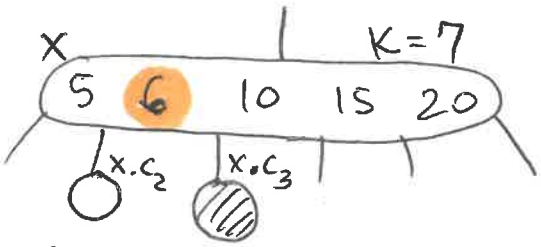
x is NOT a leaf



i=4
...
i=1 after while loop
line 11 ⇒ i=2

if $x.c_i$ is full (lines 13..16)



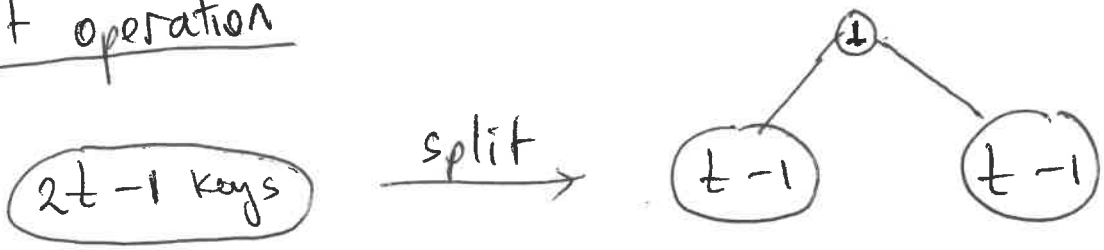


if $k > x.key_i$ ⇒ $i = i+1$

lines 1..16 have RT = O(t)

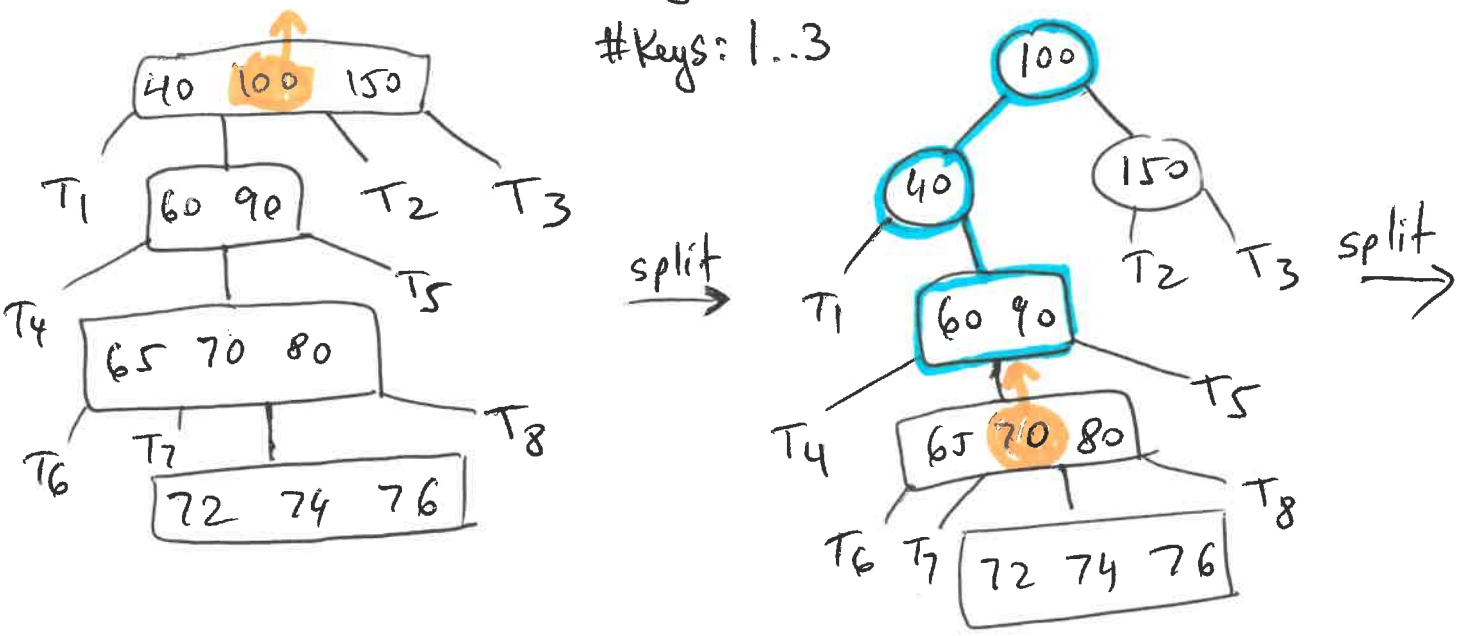
Insert operation

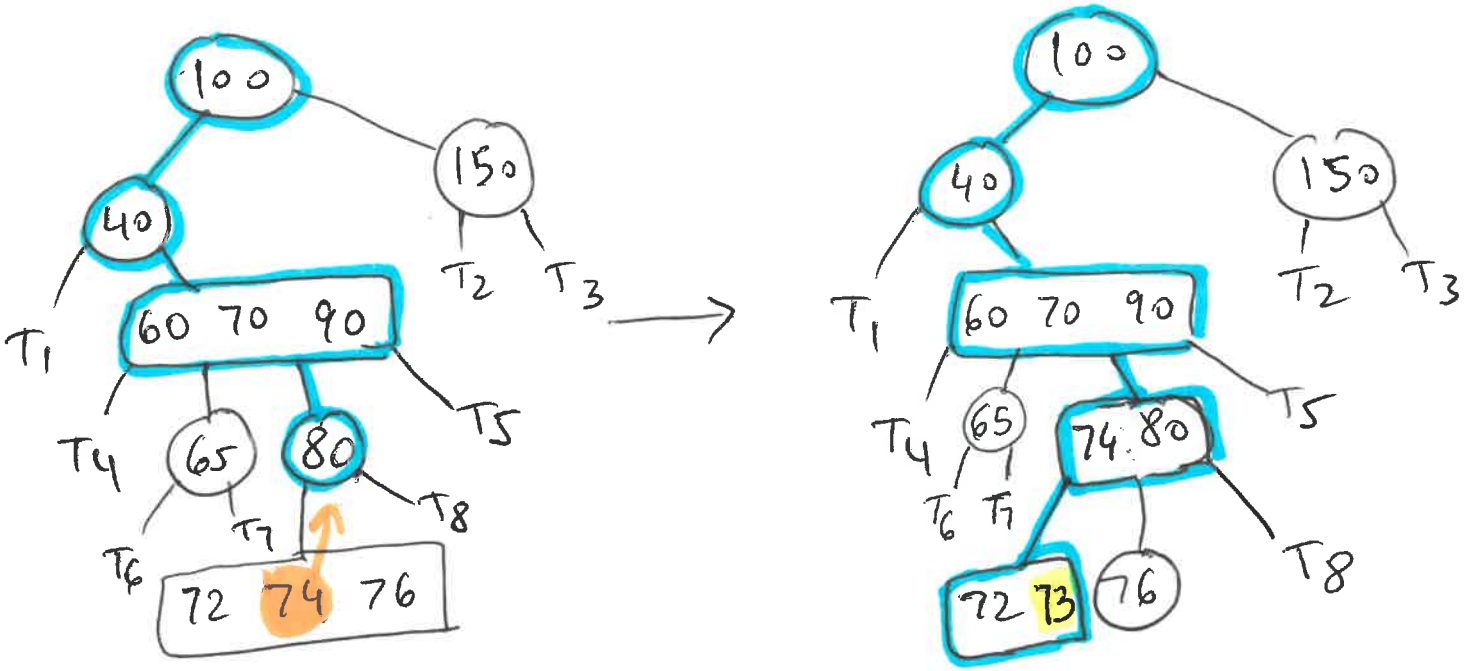
Split operation



Insert : insert the key in a single pass from the root to a leaf
- as we move down the tree, all full nodes are split accordingly

example
let t=2. Insert the key 73 into the B-tree below:



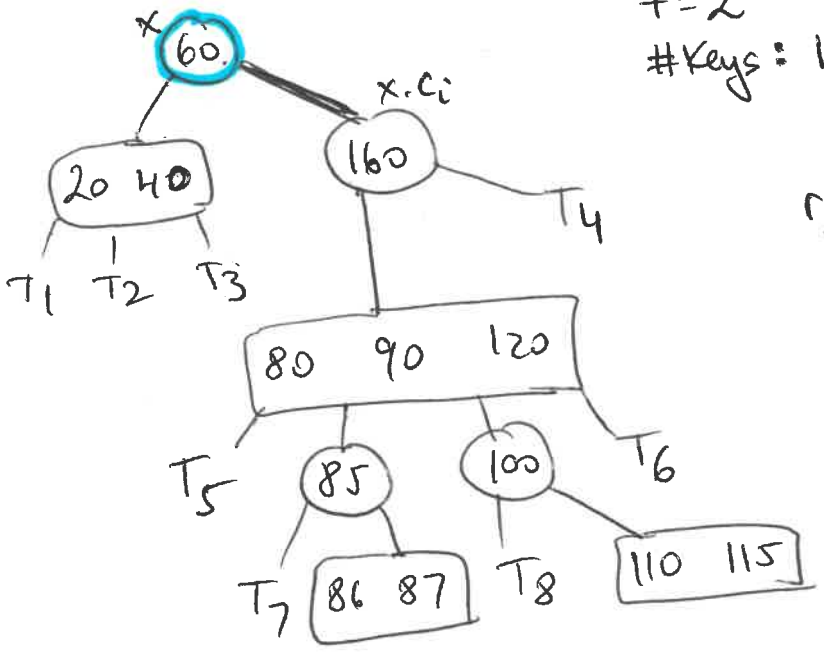


Delete

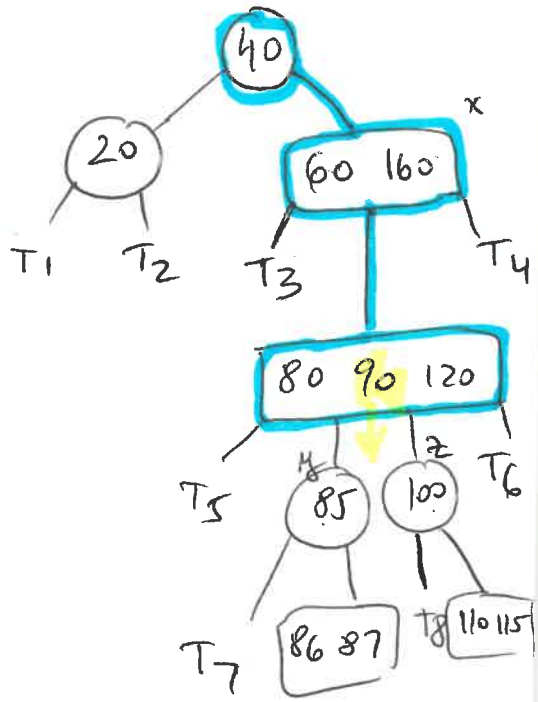
- delete the key 90 from the following B-tree:

$t = 2$

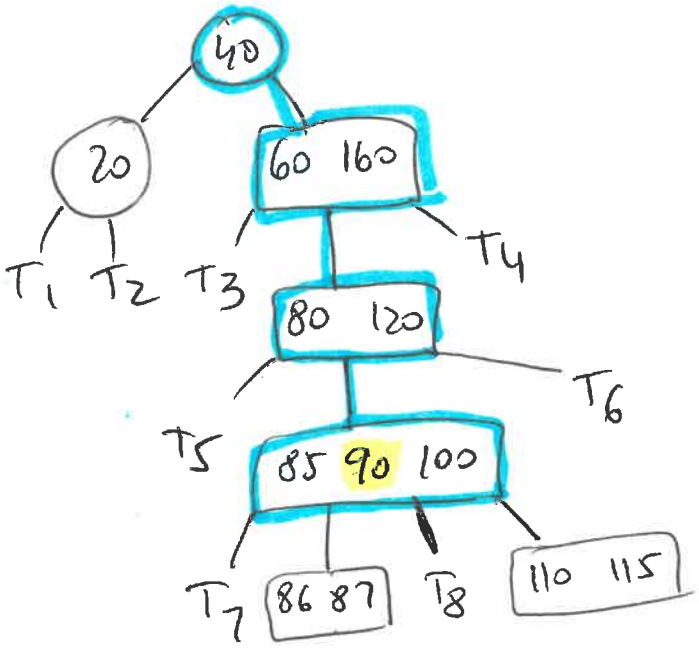
#keys : 1..3



rule 3a →



Rule 2c merge →



(to be continued next class)

Rules for DELETING a key:

Rule 1: If the key $k \in$ to the leaf node x, then delete the key k from x

Rule 2: If the key $k \in$ to the internal node x :

a. if the child y that precedes k in a node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' and replace k by k' in x.
Find and delete k' in a single downward pass.

b. if y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z.
Recursively delete k' and replace k by k' in x.
Find and delete k' in a single downward pass.

c. Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t$-1 keys. Then free z and recursively delete k from y.

Rule 3: If $k \notin$ to the internal node x, take $x.ci$ the root of the subtree that must contain k (if k is in the tree). If $x.ci$ has only t-1 keys, then use 3a or 3b to guarantee we descend to a node with $\geq t$ keys

a. If $x.ci$ has an immediate sibling with $\geq t$ keys, then give $x.ci$ an extra key by:
   - moving a key from x to $x.ci$,
   - moving a key from $x.ci$'s immediate left or right sibling up to x,
   - moving the appropriate child pointer from the sibling into $x.ci$

b. If both $x.ci$'s immediate siblings have t-1 keys, merge $x.ci$ with one sibling, which involves moving a key from x down into the new merged node to become the median for that node