# Data Structures – Week #6

## Special Trees

# Outline

- Adelson-Velskii-Landis (AVL) Trees
- Splay Trees
- B-Trees

# AVL Trees



## Motivation for AVL Trees

- Accessing a node in a BST takes *$O(\log_2 n)$* in average.
- A BST can be structured so as to have an average access time of *$O(n)$*. *Can you think of one such BST?*
- Q: Is there a way to *guarantee a worst-case access time of $O(\log_2 n)$ per node* or can we find a way to *guarantee a BST depth of $O(\log_2 n)$*?
- A: ***AVL Trees***

# Definition

An *AVL tree* is a *BST* with the following *balance condition*:
*for each node in the BST, the height of left and right sub-trees can differ by at most 1*, or

$$\left| h_{N_L} - h_{N_R} \right| \leq 1.$$



# Remarks on Balance Condition

- *Balance condition must be easy to maintain*:
  - This is the reason, for example, for the balance condition's not being as follows: the height of left and right sub-trees of each node have the same height.
- *It ensures the depth of the BST is $O(\log_2 n)$.*
- The *height information is stored* as an additional field in BTNodeType.

## Structure of an AVL Tree

```
struct BTNodeType {
    infoType *data;
    unsigned int height;
    struct BTNodeType *left;
    struct BTNodeType *right;
}
```



## Rotations

Definition:

- *Rotation* is the operation performed on a BST to restore its AVL property lost as a result of an insert operation.

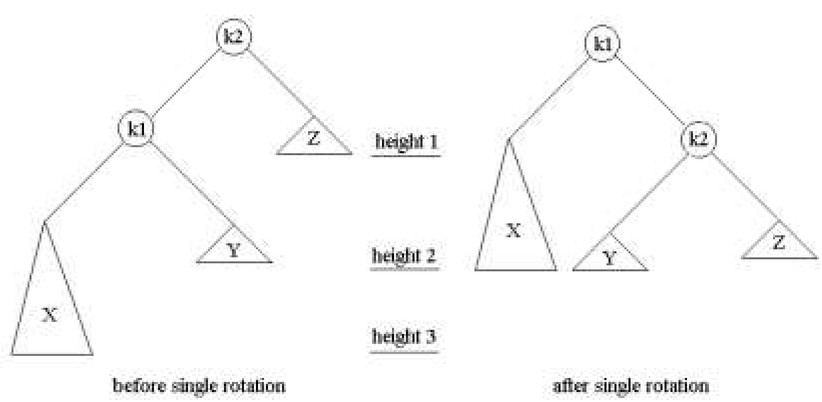
- We consider the node α whose new balance violates the AVL condition.

# Rotation

- Violation of AVL condition
- The AVL condition violation may occur in four cases:
  - Insertion into *left subtree of the left child (L/L)*
  - Insertion into *right subtree of the left child (R/L)*
  - Insertion into *left subtree of the right child (L/R)*
  - Insertion into *right subtree of the right child (R/R)*
- The outside cases 1 and 4 (i.e., L/L and R/R) are fixed by a *single rotation*.
- The other cases (i.e., R/L and L/R) need two rotations called *double rotation* to get fixed.
- These are fundamental operations in balanced-tree algorithms.

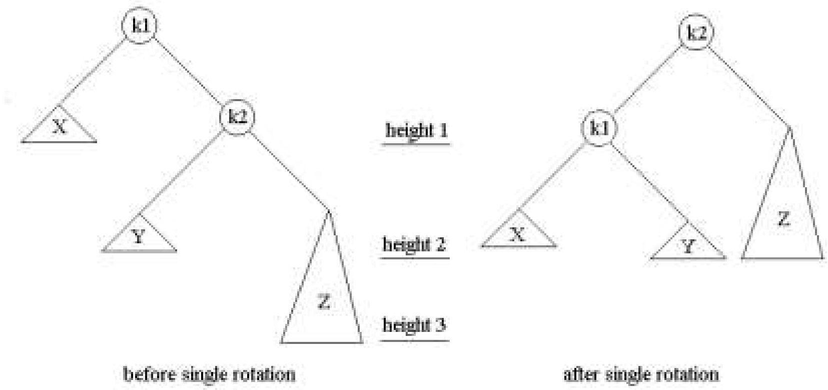


# Single Rotation (L/L)




$\alpha \equiv$ k2 node

## Single Rotation (R/R)

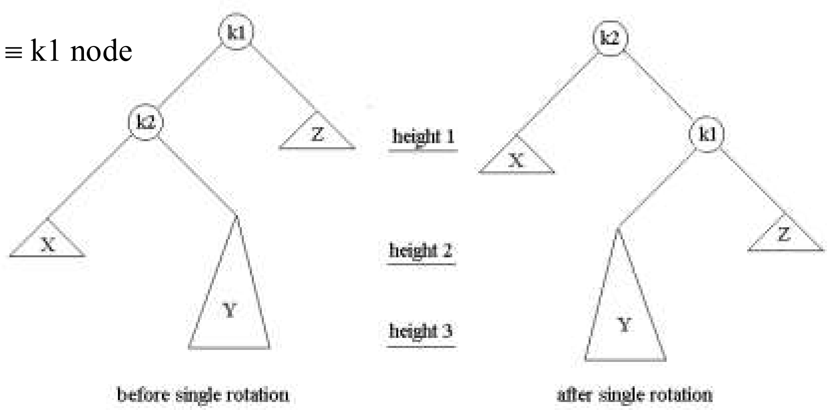



α ≡ k1 node



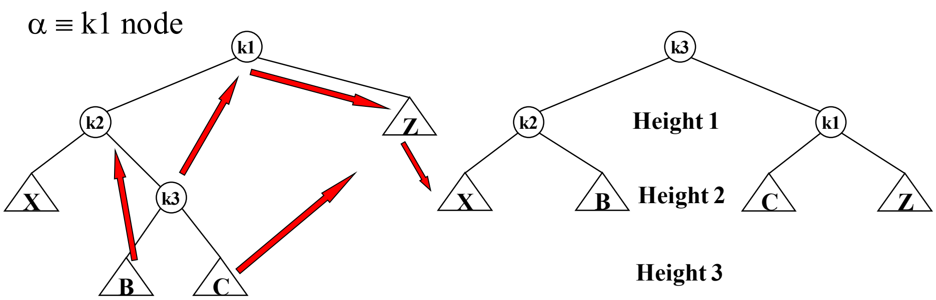
## Double Rotation (R/L)

α ≡ k1 node




**Single rotation cannot fix the AVL condition violation!!!**

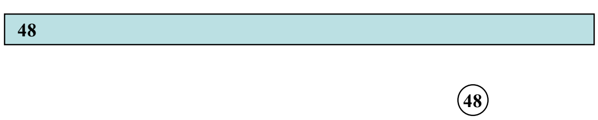
# Double Rotation (R/L)




**The symmetric case (L/R) is handled similarly left as an exercise to you!**

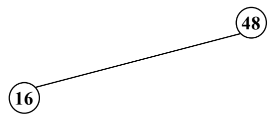


# Constructing an AVL Tree – Animation



# Constructing an AVL Tree – Animation

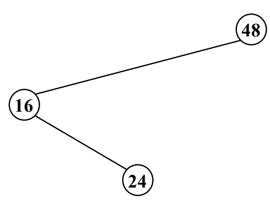
48 16





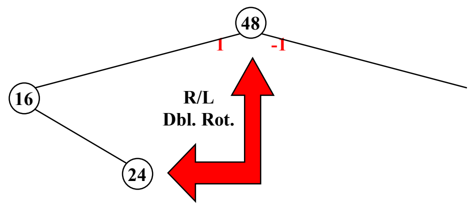

# Constructing an AVL Tree – Animation

48 16 24



## Constructing an AVL Tree – Animation

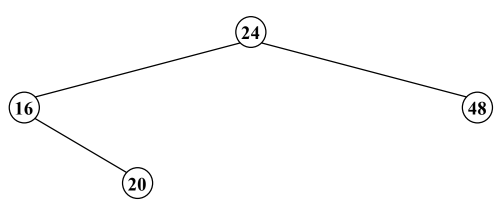
**48 16 24**





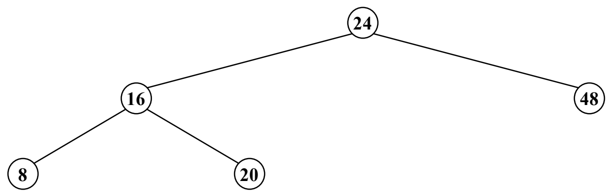

## Constructing an AVL Tree – Animation

**48 16 24 20**



# Constructing an AVL Tree – Animation

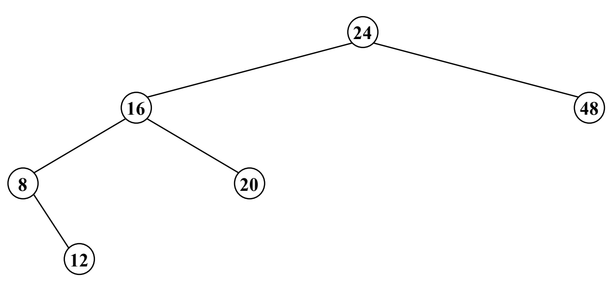
**48 16 24 20 8**





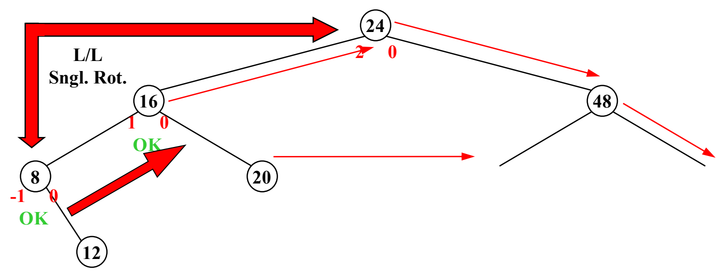

# Constructing an AVL Tree – Animation

**48 16 24 20 8 12**



# Constructing an AVL Tree – Animation

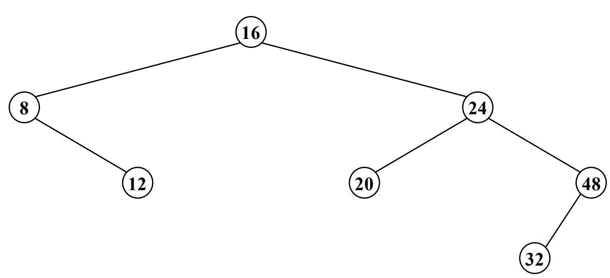
**48 16 24 20 8 12**





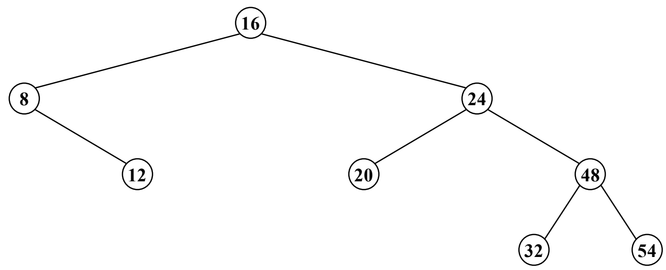

# Constructing an AVL Tree – Animation

**48 16 24 20 8 12 32**



# Constructing an AVL Tree – Animation

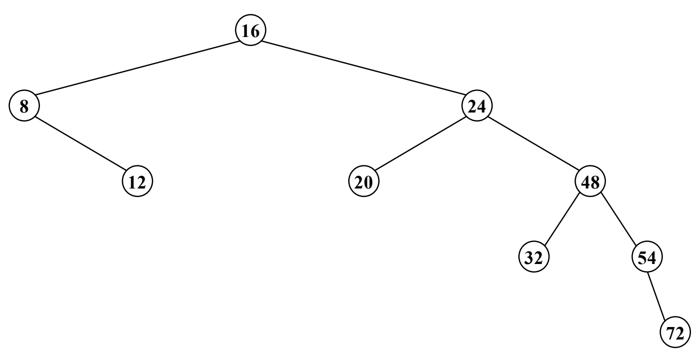
**48 16 24 20 8 12 32 54**





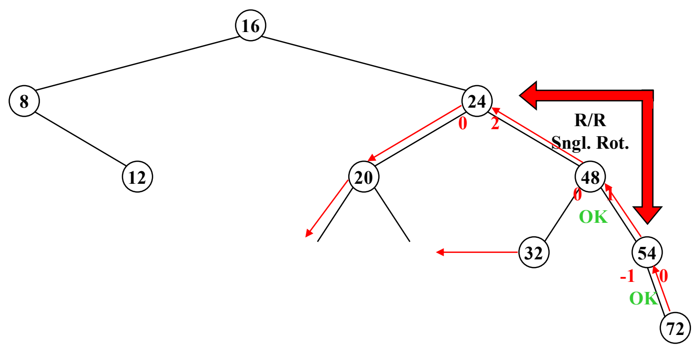

# Constructing an AVL Tree – Animation

**48 16 24 20 8 12 32 54 72**



# Constructing an AVL Tree – Animation

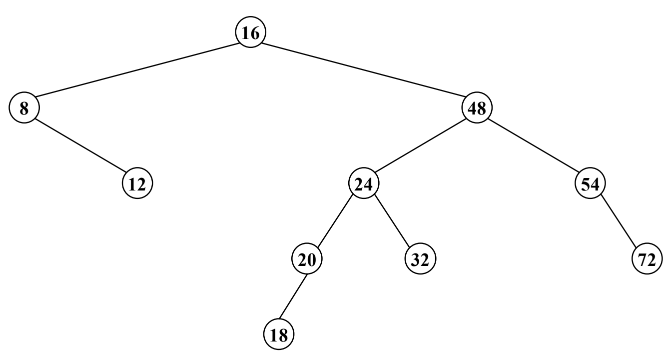
**48 16 24 20 8 12 32 54 72**





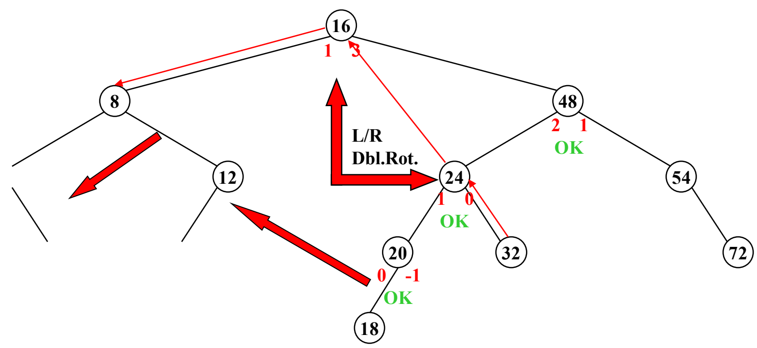

# Constructing an AVL Tree – Animation

**48 16 24 20 8 12 32 54 72 18**



# Constructing an AVL Tree – Animation

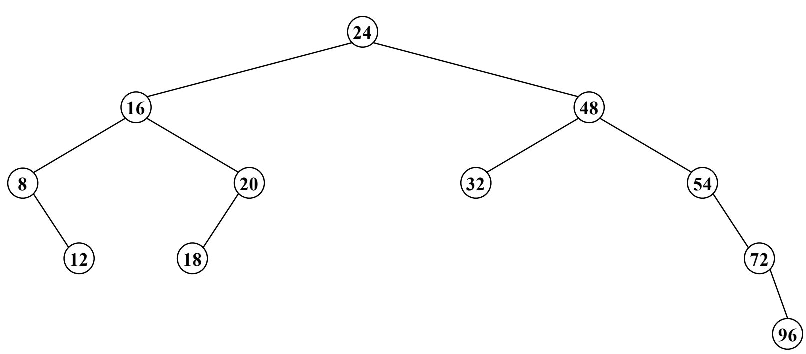
**48 16 24 20 8 12 32 54 72 18**





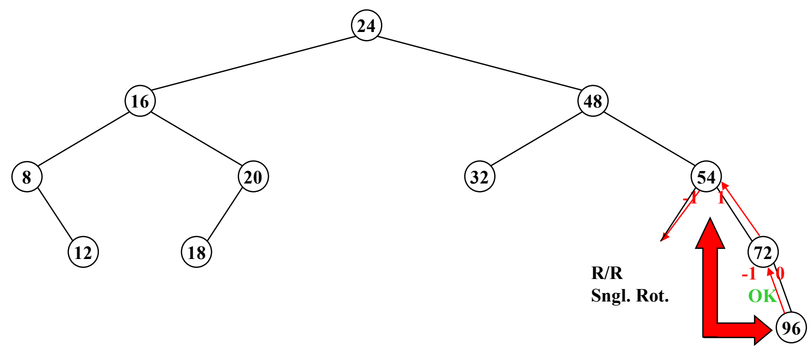

# Constructing an AVL Tree – Animation

**48 16 24 20 8 12 32 54 72 18 96**



# Constructing an AVL Tree – Animation

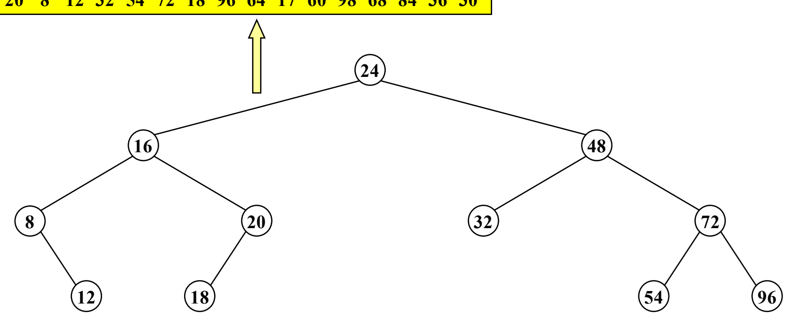
**48 16 24 20 8 12 32 54 72 18 96**






# Constructing an AVL Tree – Animation

**48 16 24 20 8 12 32 54 72 18 96 64 17 60 98 68 84 36 30**



## Height versus Number of Nodes

- The *minimum number* of nodes in an AVL tree recursively relates to the height of the tree as follows:

$$S(h) = S(h-1) + S(h-2) + 1;$$

*Initial Values*: $S(0)=1;\ S(1)=2$

Homework: Solve for $S(h)$ as a function of $h$!



## Splay Trees

## Motivation for Splay Trees

- We are looking for a data structure where, *even though some worst case (O(n)) accesses may be possible*, *m consecutive tree operations starting from an empty tree (inserts, finds and/or removals) take* $O(m*\log_2 n)$.
- Here, the main idea is to assume that, *O(n) accesses* are not bad as long as they *occur relatively infrequently*.
- Hence, we are looking for *modifications of a BST per tree operation that attempts to minimize O(n) accesses*.



## Splaying

- The underlying idea of splaying is to *move a deep node accessed upwards to the root*, assuming that it will be accessed in the near future again.
- While doing this, other deep nodes are also carried up to smaller depth levels, making the average depth of nodes closer to $O(\log_2 n)$.

# Splaying

- Splaying is similar to bottom-up AVL rotations
- If a node *X is the child of the root* R,
  - then we *rotate only X and R, and this is the last rotation performed*.

    else consider *X*, its *parent P* and *grandparent G*.

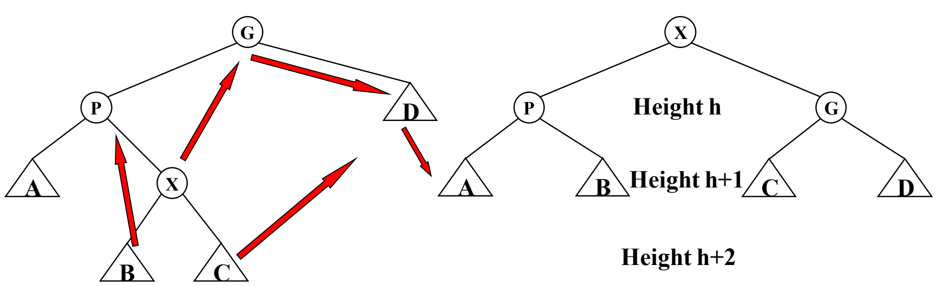
    Two cases and their symmetries to consider

    *Zig-zag case,* and

    *Zig-zig case*.



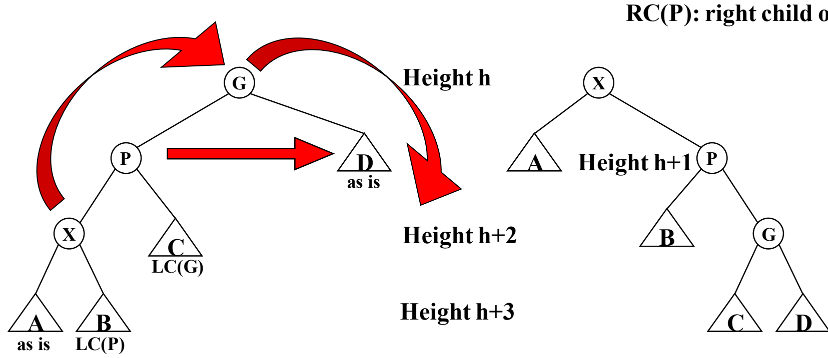
# Zig-zag case




**This is the same operation as an AVL double rotation in an R/L violation.**

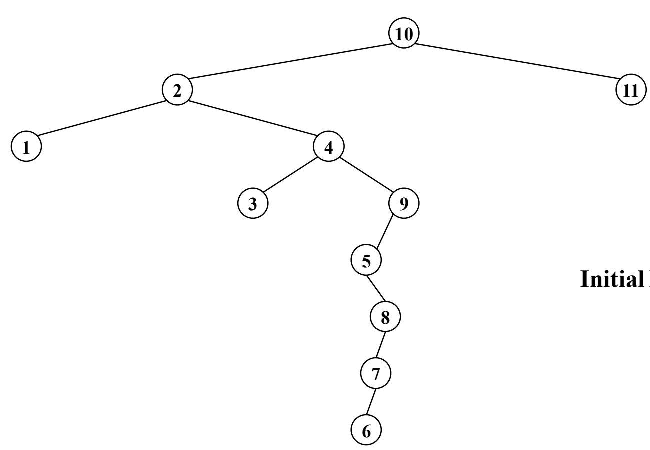
# Zig-zig case

LC(P): left child of node P
RC(P): right child of node P



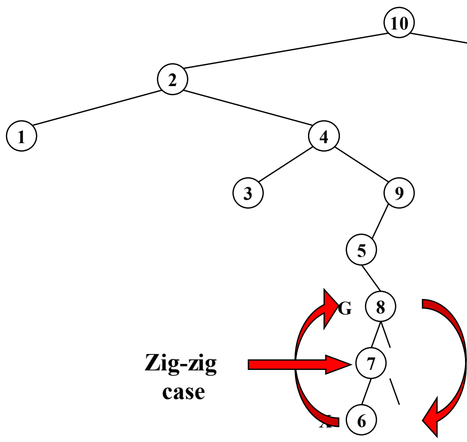



# Animated Example




Initial BST

# Animated Example

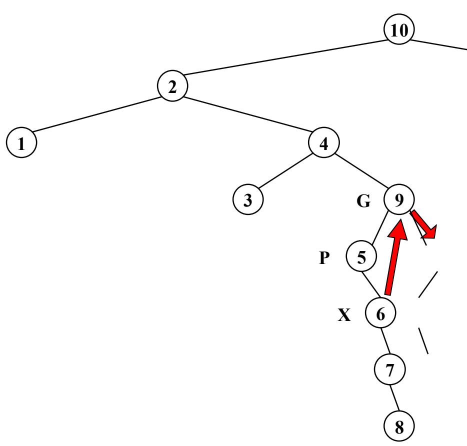



**Node with 6 accessed!**

**G** stands for **<u>G</u>randparent**
**P** stands for **<u>P</u>arent**
**X** denotes the ***current node***



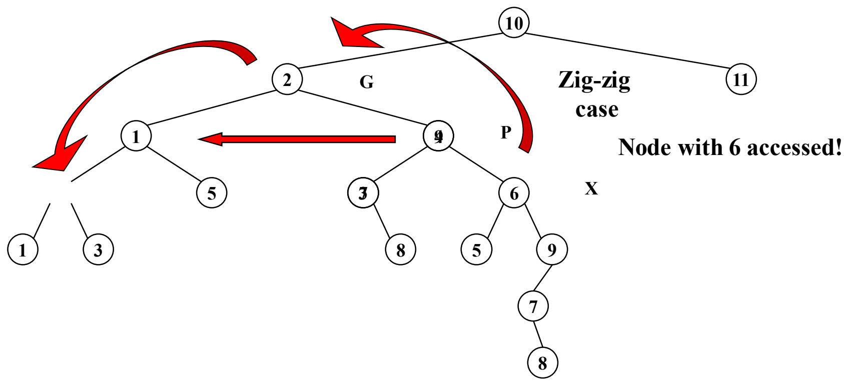
# Animated Example




**Node with 6 accessed!**

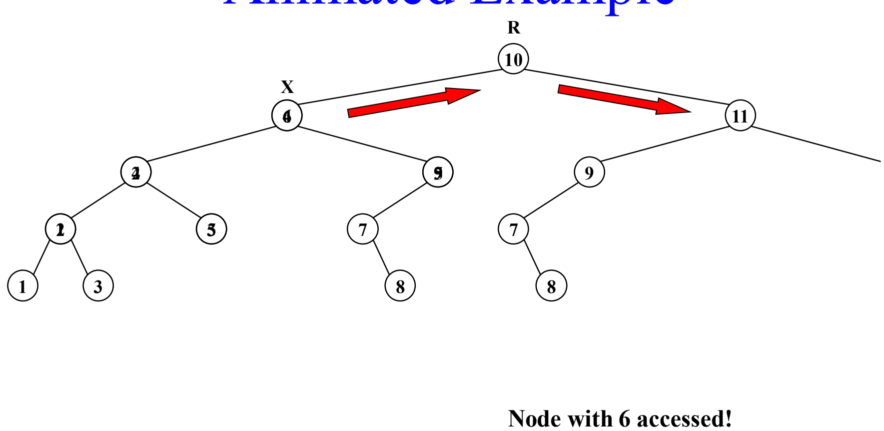
**Zig-zag case**

# Animated Example

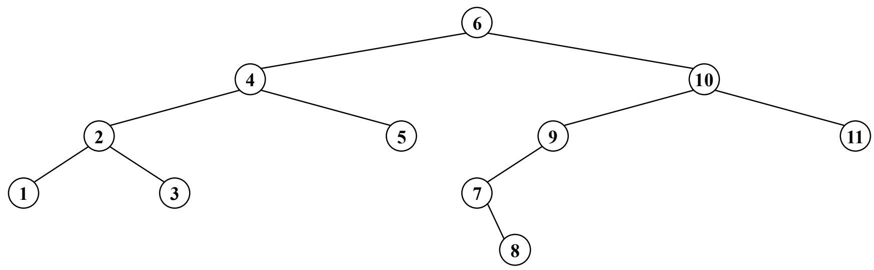





# Animated Example



# Animated Example




**Node with 6 accessed!**



# B-Trees

# Motivation for B-Trees

- Two technologies for providing memory capacity in a computer system
  - *Primary* (main) *memory (silicon chips)*
  - *Secondary storage (magnetic disks)*
- Primary memory
  - 5 orders of magnitude (i.e., about $10^5$ times) *faster*,
  - 2 orders of magnitude (about 100 times) *more expensive*, and
  - by at least 2 orders of magnitude *less in size*

  than secondary storage due to mechanical operations involved in magnetic disks.



# Motivation for B-Trees

- During one disk read or disk write (4-8.5msec for 7200 RPM disks), MM can be accessed about $10^5$ times (100 nanosec per access).
- To reimburse (compensate) for this time, at each disks access, *not a single item*, but one or more ***equal-sized*** *pages* of items (each page $2^{11}$-$2^{14}$ bytes) are accessed.
- We need some data structure to store these ***equal sized*** *pages* in MM.
- *B-Trees*, with their ***equal-sized leaves*** *(as big as a page),* are suitable data structures for storing and performing regular operations on paged data.

# B-Trees

- A *B-tree* is a rooted tree with the following properties:
- Every node $x$ has the following fields:
  - $n[x]$, the number of keys currently stored in $x$.
  - the $n[x]$ keys themselves, in *non-decreasing order*, so that

    $$key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x] ,$$
  - $leaf[x]$, a boolean value, true if $x$ is a leaf.



# B-Trees

- *Each internal node* has $n[x]+1$ pointers, $c_1[x],\ldots, c_{n[x]+1}[x]$, to its children. *Leaf nodes* have *no children*, hence no pointers!
- The keys separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $c_i[x]$, then

  $$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1} .$$
- *All leaves have the same depth*, $h$, equal to the *tree's height*.

# B-Trees

- There are lower and upper bounds on the number of keys a node may contain. These bounds can be expressed in terms of a fixed integer *$t \geq 2$* called the *minimum degree* of the B-Tree.
  - Lower limits
    - All *nodes but the root* has *at least t-1* keys.
    - Every *internal node but the root* has *at least **t** children*.
    - A non-empty tree's **root** must have *at least one key*.



# B-Trees

- Upper limits
  - Every *node* can contain *at most **2t-1 keys***.
  - Every *internal node* can have *at most **2t children***.
  - A node is defined to be full if it has exactly *2t-1 keys*.
- For a *B-tree* of minimum degree *$t \geq 2$* and *$n$* nodes

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

## Basic Operations on B-Trees

- *B-tree search*
- *B-tree insert*
- *B-tree removal*



## Disk Operations in B-Tree operations

- Suppose *x* is a pointer to an object.
- It is accessible if it is in the main memory.
- If it is on the disk, it needs to be transferred to the main memory to be accessible. This is done by *DISK_READ(x)*.
- To save any changes made to any field(s) of the object pointed to by *x*, a *DISK_WRITE(x)* operation is performed.

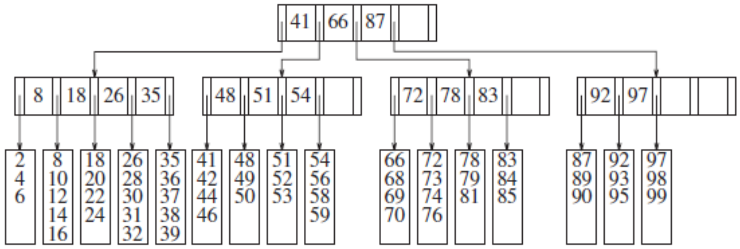



**Figure 4.63** B-tree of order 5

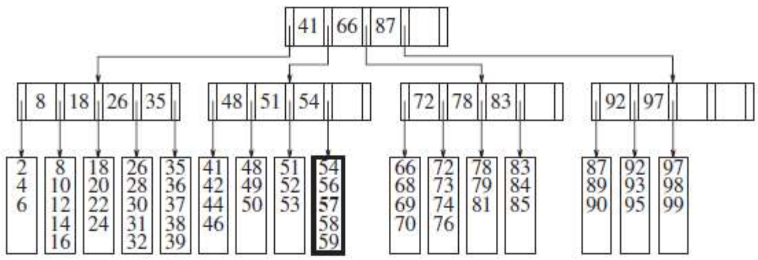



**Figure 4.64** B-tree after insertion of 57 into the tree in Figure 4.63



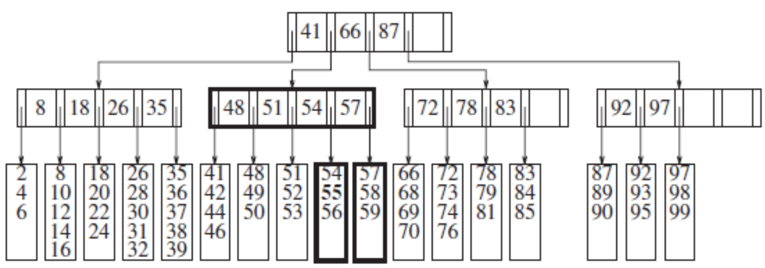



**Figure 4.65** Insertion of 55 into the B-tree in Figure 4.64 causes a split into two leaves


**Figure 4.66** Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

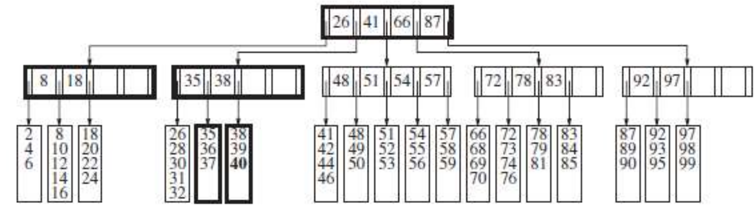



**Figure 4.66** Insertion of 40 into the B-tree in Figure 4.65 causes a split into two leaves and then a split of the parent node

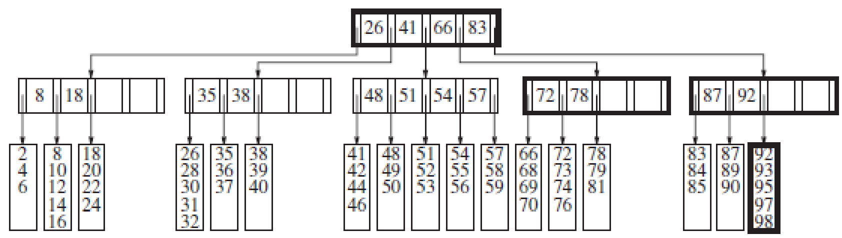
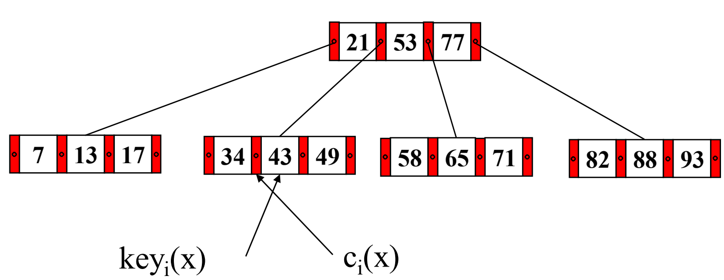



**Figure 4.67** B-tree after the deletion of 99 from the B-tree in Figure 4.66

March 10, 2021 Borahan Tümer, Ph.D. 55

# Search in B-Trees

- Similar to search in BSTs with the exception that instead of a binary, a multi-way (*n[x]*+1-way) decision is made.




March 10, 2021 Borahan Tümer, Ph.D. 56

# Search in B-Trees

```
B-tree-Search(x, k)
{  i=1;
   while (i ≤ n[x] and k > key_i[x]) i++;
   if (i ≤ n[x] and k = key_i[x])           // if key found
        return (x,i);
   if (leaf[x])                    // if key not found at a leaf
        return NULL;
   else {DISK_READ(c_i[x]);              // if key < key_i[x]
          return B-tree-Search(c_i[x],k);}
}
```



# Insertion in B-Trees

- Insertion into a B-tree is more complicated than that into a BST, since the creation of a new node to place the new key may violate the B-tree property of the tree.
- Instead, the key is put *into a leaf node x if it is not full*.
- If full, a *split* is performed, which splits a full node (with *2t-1* keys) at its *median key*, $key_t[x]$, into two nodes with *t-1* keys each.
- $key_t[x]$ moves up into the parent of *x* and identifies the split point of the two new trees.

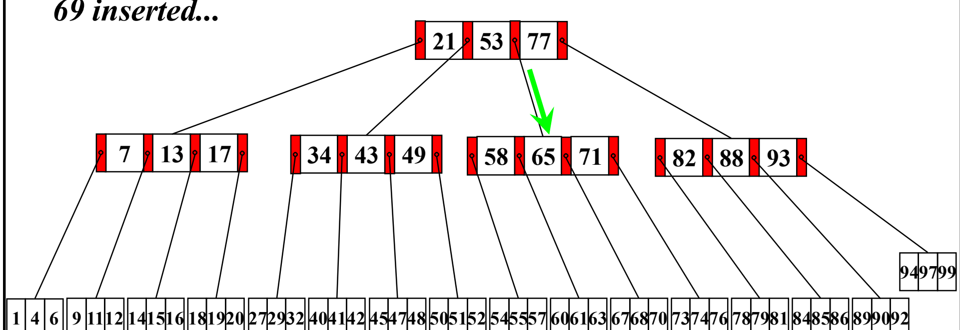
# Insertion in B-Trees

- A *single-pass insertion* starts at the root traversing *down to the leaf* into which the key is to be inserted.
- On the path down, *all full nodes are split* including a full leaf that also guarantees a parent with an available position for the median key of a full node to be placed.



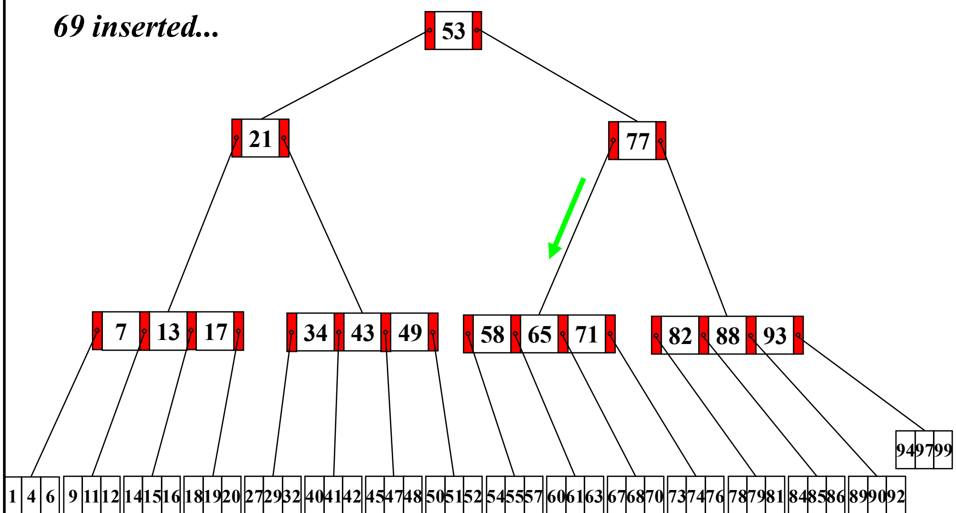
# Insertion in B-Trees: Example

***69 inserted...***



# Insertion in B-Trees: Example

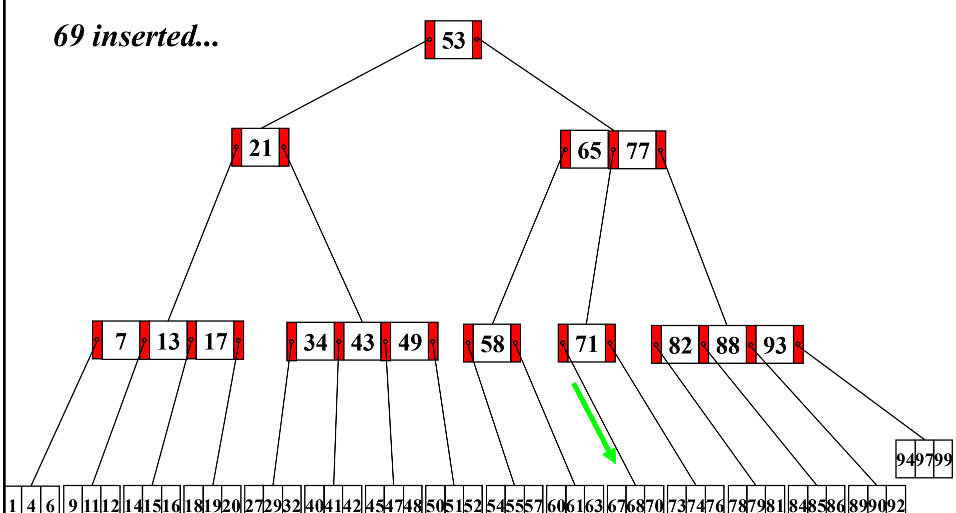
***69 inserted...***





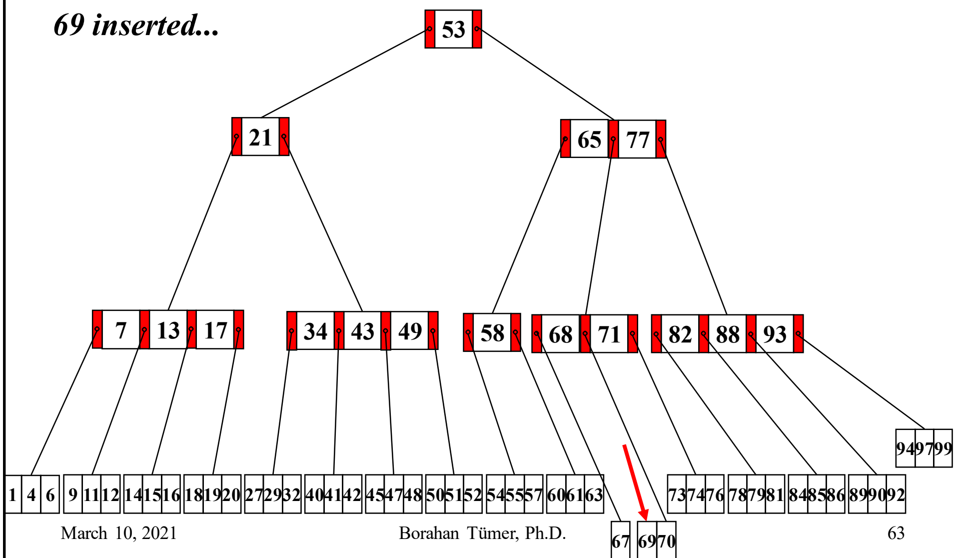

# Insertion in B-Trees: Example

***69 inserted...***



# Insertion in B-Trees: Example

***69 inserted...***






# Insertion in B-Trees: B-tree-Insert

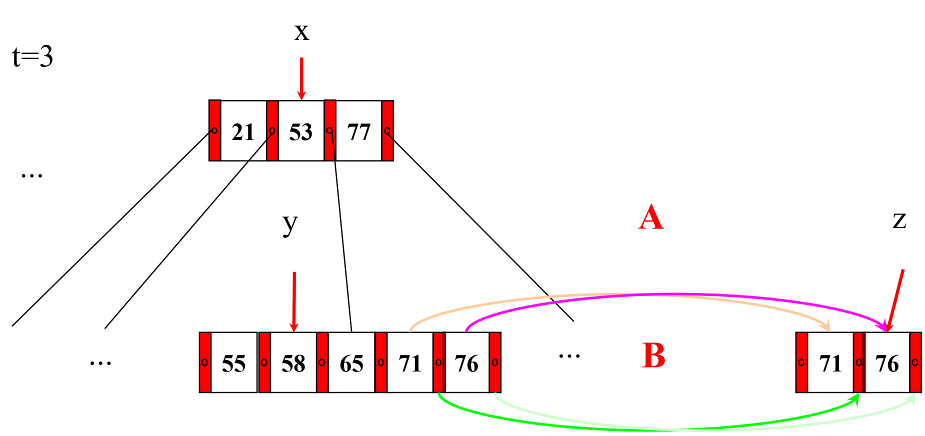
```
B-tree-Insert(T,k)
{  r=root[T];
   if (n[r] == 2t-1) {
         s=malloc(new-B-tree-node);
         root[T]=s;
         leaf[s]=false;
         n[s]=0;
         c1[s]=r;
         B-tree-Split-Child(s,1,r);
         B-tree-Insert-Nonfull(s,k); }
   else B-tree-Insert-Nonfull(r,k);
}
```

# Insertion in B-Trees: B-tree-Split-Child

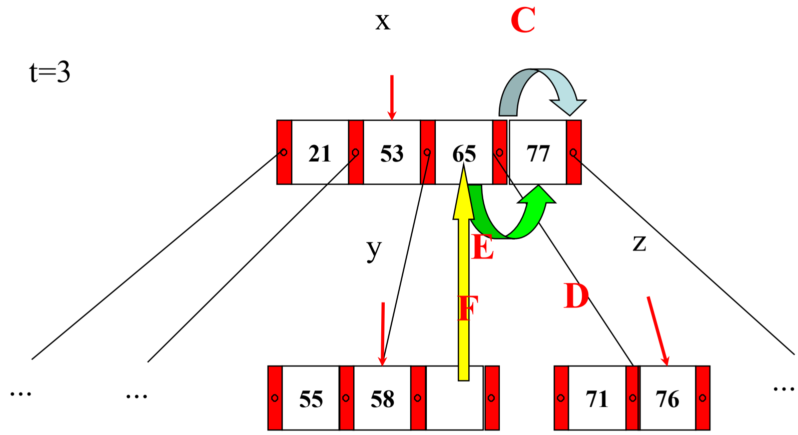
B-tree-Split-Child(x,i,y)
{    z=malloc(new-B-tree-node);
    leaf[z]=leaf[y];
    n[z]=t-1;
    for (j = 1; j < t) $key_j[z]=key_{j+t}[y]$; **A**
    if (!leaf[y])
        for (j = 1; j <= t;j++) $c_j[z]=c_{j+t}[y]$; **B**
    n[y]=t-1;
    for (j=n[x]+1; j>=i+1; j--) $c_{j+1}[x]=c_j[x]$; **C**
    $c_{i+1}[x]=z$; **D**
    for (j=n[x]; j>=i; j--) $key_{j+1}[x]=key_j[x]$; **E**
    $key_i[x]=key_t[y]$; n[x]++; **F**
    DISK_WRITE(y);
    DISK_WRITE(z);
    DISK_WRITE(x);
}



# B-tree-Split-Child: Example

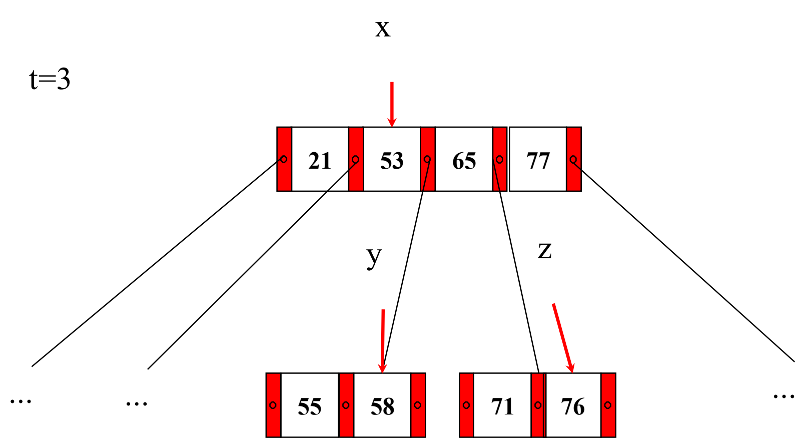


# B-tree-Split-Child: Example






# B-tree-Split-Child: Example



# Insertion in B-Trees: B-tree-Insert-Nonfull

```
B-tree-Insert-Nonfull(x,k)
{   i=n[x];
    if (leaf[x]) {
          while (i≥1 and k < key_i[x]) {key_i+1[x]=key_i[x]; i--;}
           key_i+1[x]=k;
           n[x]++;
           DISK_WRITE(x);
    }
    else {
           while (i≥1 and k < key_i[x]) i--;
           i++;
           DISK_READ(c_i[x]);
           if (n[c_i[x]]==2t-1) {
                     B-tree-Split-Child(x,i, c_i[x]);
                     if (k > key_i[x]) i++;
           }
           B-tree-Insert-Nonfull(c_i[x],k);
         }
}
```

if x is a leaf
then place key in x;
write x on disk;
else find the node (root of subtree) key goes to;
read node from disk;
if node full
split node at key's position;
recursive call with node split and key;



# Removing a key from a B-Tree

- Removal in B-trees is different than insertion only in that *a key may be removed from any node, not just from a leaf*.
- As the insertion algorithm splits any full node down the path to the leaf to which the key is to be inserted, a recursive removal algorithm may be written to ensure that for any call to removal on a node $x$, the number of keys in $x$ is at least the minimum degree $t$.

## Various Cases of Removing a key from a B-Tree

1. If the key $k$ is in node $x$ and $x$ is a leaf, remove the key $k$ from $x$.
2. If the key $k$ is in node $x$ and $x$ is an internal node, then
   a. If the child $y$ that precedes $k$ in 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 $x$. Finding $k'$ and deleting it can be performed in a single downward pass.



## Various Cases of Removal a key from a B-Tree

b. Symmetrically, if the child $z$ that follows $k$ in node $x$ 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 $x$. Finding $k'$ and deleting it can be performed 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. Free $z$ and recursively delete $k$ from $y$.

## Various Cases of Removal a key from a B-Tree

3. If $k$ is not present in internal node $x$, determine root $c_i[x]$ of the subtree that must contain $k$, if $k$ exists in the tree. If $c_i[x]$ has only $t$-$1$ keys, execute step *3a* or *3b* as necessary to guarantee that we descend to a node containing at least $t$ keys. Then finish by recursing on the appropriate child of $x$.



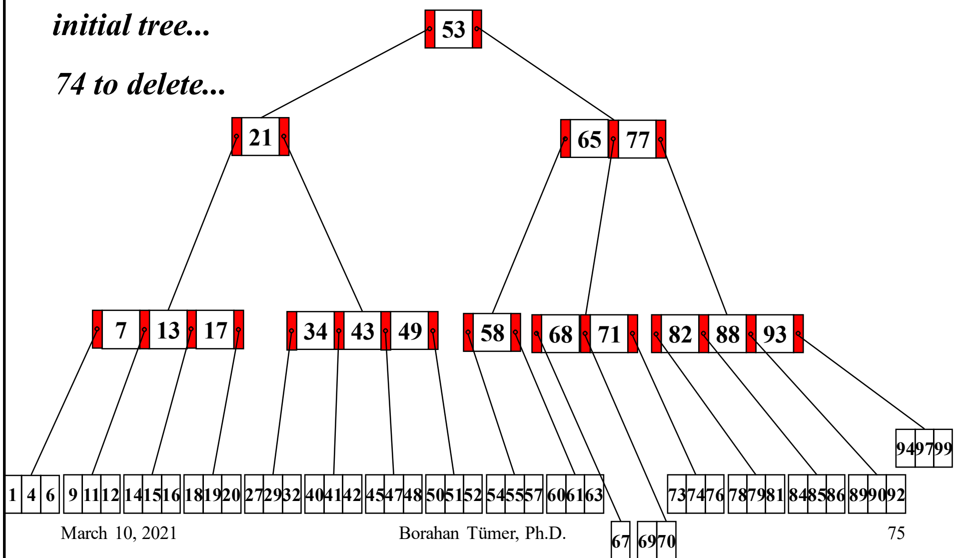
## Various Cases of Removal a key from a B-Tree

a. If $c_i[x]$ has only $t$-$1$ keys but has an immediate sibling with at least $t$ keys, give $c_i[x]$ an extra key by moving a key from $x$ down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.
b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t$-$1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from $x$ down into the new merged node to become the *median key* for that node.

# Removal in B-Trees: Example

***initial tree...***

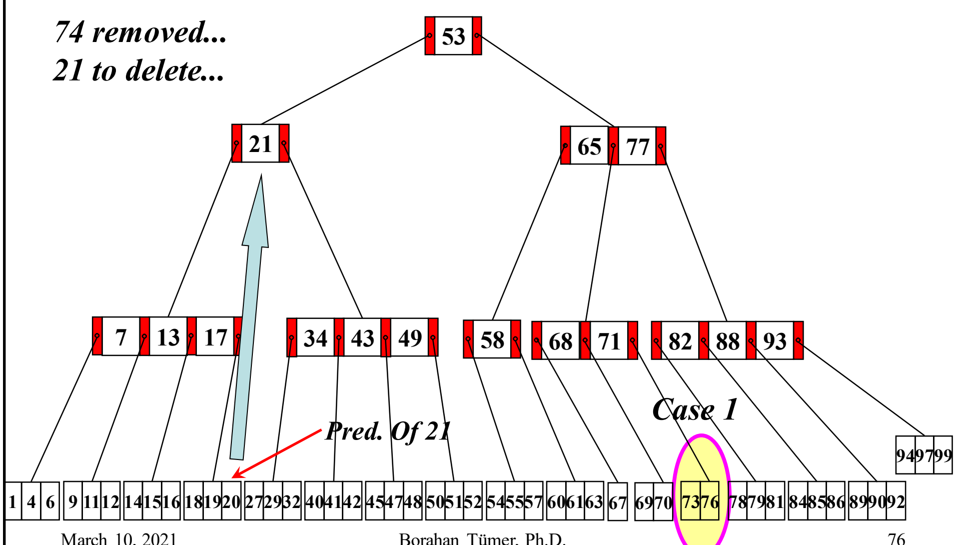
***74 to delete...***




March 10, 2021 Borahan Tümer, Ph.D. 75

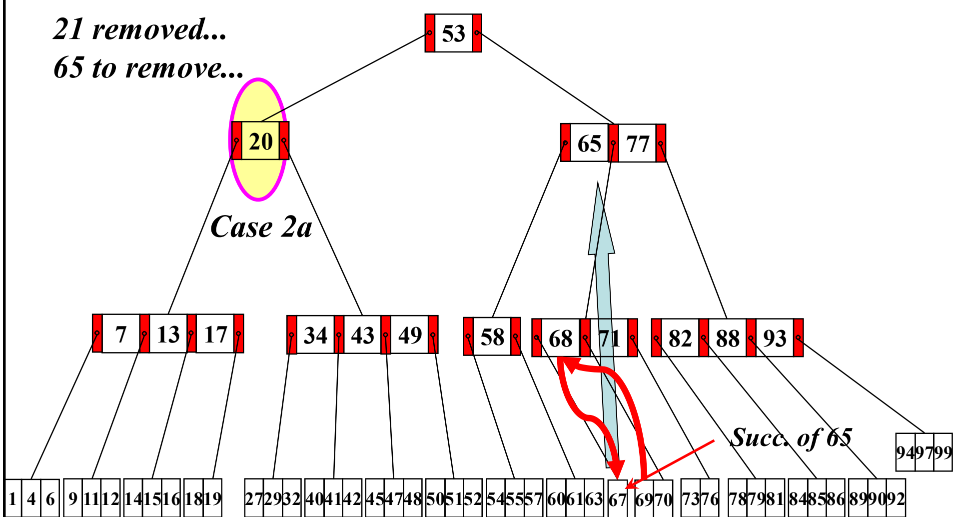
# Removal in B-Trees: Example

***74 removed...***
***21 to delete...***



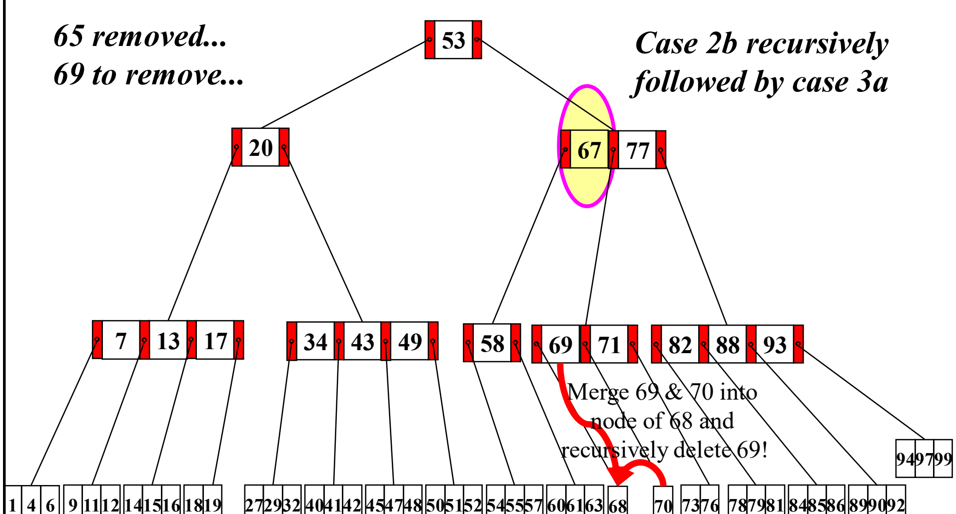

March 10, 2021 Borahan Tümer, Ph.D. 76

# Removal in B-Trees: Example





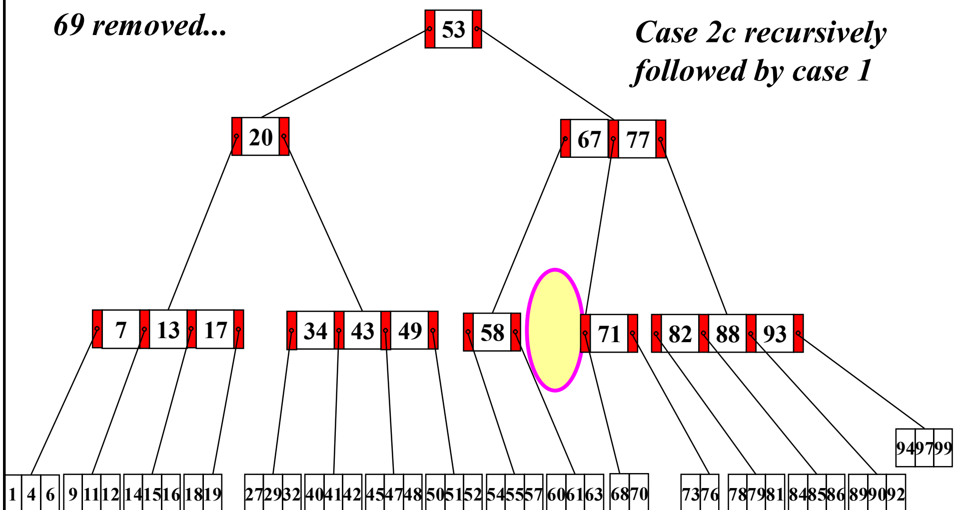

# Removal in B-Trees: Example



# Removal in B-Trees: Example

*69 removed...*

*Case 2c recursively followed by case 1*




March 10, 2021 Borahan Tümer, Ph.D. 79