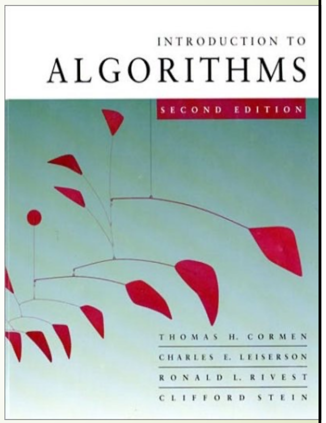
B- TREES

SUBJECT-CODE : KCS-503

Dr. Ragini Karwayun



1

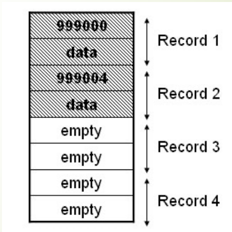
Introduction

- B-trees are balanced binary search trees designed to work well on magnetic disks or other direct-access secondary storage devices.
- B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.
- B-trees differ from red-black trees in that B-tree nodes can have more than one key, in fact from a handful to thousands.
- That is, the "branching factor" of a B-tree can be quite large, although it is usually determined by characteristics of the disk unit used.
- B- trees are similar to red-black trees in that every n-node B-tree has height $O(\lg n)$, although the height of a B-tree can be considerably less than that of a red-black tree because its branching factor can be much larger. Therefore, B-trees can also be used to implement many dynamic-set operations in time $O(\lg n)$.

2

Data Storage

- Data is stored on disk (i.e., secondary memory) in blocks.
- A block is the smallest amount of data that can be accessed on a disk.
- Each block has a fixed number of bytes – typically 512, 1024, 2048, 4096 or 8192 bytes
- Each block may hold many data records.



DATA STRUCTURES ON SECONDARY STORAGE

- Most common secondary storage device is Magnetic disk.
- Large data do not fit into operational memory.
- Disks are very cheap and have higher memory capacity.
- They are very slow due to the mechanical movements of two of its components: platter rotation (Latency time) and arm movement (Seek time).
- In order to minimize the time spent waiting for mechanical movements, disks access not just one item but several at a time (page or block).
- 1 disk access ~ 13 000 000 instructions!!!!
- Number of disk accesses dominates the computational time
- Disk access = Disk-Read, Disk-Write
- Disk divided into blocks. (512, 2048, 4096, 8192 bytes)
- To minimize the overhead ,whole block containing the requested instruction is transferred.

Motivation for studying Multi-way and B-trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.

5

MOTIVATION FOR B-TREES

Disc rotation speed: 7200 RPM, typical
Arm movement: few milli sec.
Total access time for data on disc: 3-9 milli sec., typical
Versus access time in RAM: < 100 ns.
B-tree objective: to optimize disc access by using the full amount of information retrieved per disc access, i.e., one page. So:
Size of 1 node of the B-tree = 1 page / 1 Block

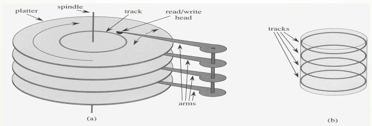


Figure 18.2 **(a)** A typical disk drive. It is composed of several platters that rotate around a spindle. Each platter is read and written with a head at the end of an arm. The arms are ganged together so that they move their heads in unison. Here, the arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when it is stationary. **(b)** A cylinder consists of a set of covertical tracks.

6

Motivation for Multi-Way Trees

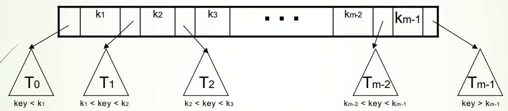
- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach, as access time of secondary storage devices is too large.
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions
- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2$ 20,000,000 is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree

Motivation for Multi-Way Trees

- But, the solution is to use more branches and thus reduce the height of the tree!
  - **As branching increases, depth decreases**
- For example:
  - Suppose we want to access the driving records of the citizens of a city.
    - 10 million items.
    - Assume doesn't fit in main memory.
    - Assume in 1 sec, can execute 25 million instructions or perform 6 disk accesses.
- The Unbalanced Binary tree would be a disaster.
  - In the worst case, it has linear depth and could require 10 mil disk accesses.
- An AVL Tree
  - In the typical case, it has a depth close to log N, log 10 mil ≈ 24 disk accesses, requiring 4 sec.

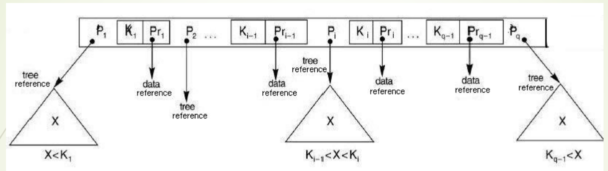
What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most m subtrees, where the subtrees may be empty.
  - Each node consists of at least 1 and at most m-1 distinct keys
  - The keys in each node are sorted.



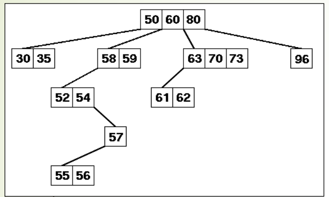
- **The keys and subtrees of a non-leaf node are ordered as:**
- **T_0, k_1, T_1, k_2, T_2, . . . , k_{m-1}, T_{m-1} such that:**
  - All keys in subtree T0 are less than k1.
  - All keys in subtree T_i , 1 <= i <= m - 2, are greater than k_i but less than k_{i+1}.
  - All keys in subtree T_{m-1} are greater than k_{m-1}

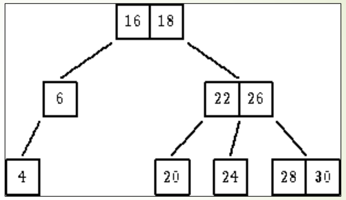
The node structure of a Multi-way tree



- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.
  - The literature contains other node representations that we will not discuss.

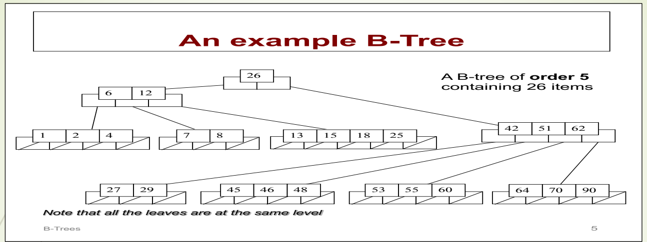
Examples of Multi-way Trees





- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with n keys may contain less than n + 1 non-empty subtrees.

11



- B-trees generalize binary search trees in a natural manner.
- If a B-tree node x contains $n[x]$ keys, then x has $n[x] + 1$ children.
- The keys in node x are used as dividing points separating the range of keys handled by x into $n[x] + 1$ subranges, each handled by one child of x.
- When searching for a key in a B-tree, we make an $(n[x] + 1)$-way decision based on comparisons with the $n[x]$ keys stored at node x.

12

B - Trees

- **A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which**:
- The number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a binary search tree.
- All leaves are on the same level. That is the tree is perfectly balanced.
- All non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children and atmost m children.
- The root is either a leaf node, or it has at least two non-empty subtrees and at most m non empty subtrees
- A leaf node contains no more than $m - 1$ keys.

13

B - Trees

- **Relation between order and min. degree t :**
- **Order of a tree = max. no of children a node can have. = 2t**

For a non-empty B-tree of order m:

	Root node	Non-root node
Minimum number of keys	1	$\lceil m / 2 \rceil - 1$
Minimum number of non-empty subtrees	2	$\lceil m / 2 \rceil$
Maximum number of keys	m - 1	m – 1
Maximum number of non-empty subtrees	m	m

14

Why B - Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:
  1. With a large branching factor m, the height of a B-tree is low resulting in fewer disk accesses.

     Note: As m increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.
  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.
  3. The most common data structure used for database indices is the B-tree. An index is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and *quickly* finds all records with that property

15

Disk Opeations

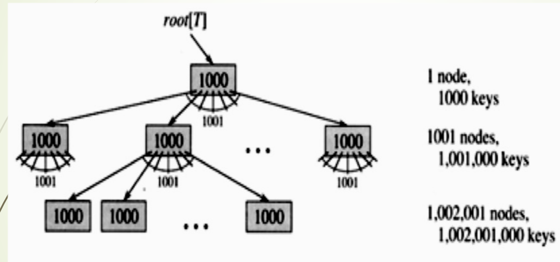
- We model disk operations in our pseudocode as follows.
- Let ***x*** = a pointer to an object.
- If **x** is currently in the computer's main memory, then we can refer to the fields of x as : *key*[*x*], for example.
- If *x* resides on disk, however, then we must perform the operation DISK-READ(*x*) to read object *x* into main memory before its fields can be referred to.
- Similarly, the operation DISK-WRITE(*x*) is used to save any changes that have been made to the fields of object *x*.
- The typical pattern for working with an object x is as follows.
  1. …….
  2. x ← pointer to some object
  3. **DISK-READ(x)**
  4. Operations that access and modify fields of x.
  5. **DISK-WRITE(x)** /* Omitted if no fields of x were changed */
  6. other operations that access but do not modify fields of x
  7. ……….

16

Disk Opeations

- The running time of a B-tree algorithm is determined mainly by the number of DISK-READ and DISK-WRITE operations it performs, so their use should be as little as possible.
- Thus, a B-tree node is usually as large as a whole disk page. The number of children a B-tree node can have is therefore limited by the size of a disk page.
- For a large B-tree stored on a disk, branching factors between 50 and 2000 are often used, depending on the size of a key relative to the size of a page.
- A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key .

A B-tree of height 2 containing over one billion keys



B-tree Properties

A B-tree is a rooted tree with the following properties:

1. Every node *x* has the following fields:
   a. *n[x]*, the number of keys stored in *x*.
   b. *n[x]* keys stored in non-decreasing order: $key_1[x]$ <= $key_2[x]$ <= ... <= $key_{n[x]}[x]$
   c. *leaf[x]* = *true* if *x* is a leaf, and *false* otherwise.
2. Each internal node *x* contains *n[x]+1* pointers to its children: $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$
3. The keys keyi[x] separate the ranges stored in each subtree: if ki is any key stored in the subtree with root ci[x], then
   k1 <= key1[x] <= k2 <= key2[x] <= ...<= keyn[x][x] <= kn[x]+1
4. All leaves have the same depth, i.e. the tree's height h.
5. There are lower and upper bounds on the number of keys in a node. They are expressed in terms of an integer t >= 2 called the minimum degree:
   a. Every node other than the root must have at least t-1 keys.
   b. Every node can contain at most (2t-1) keys. We say the node is full if it contains exactly (2t-1) keys.

19

B-tree of degree *t*

- Definition:
  - Root must have 1 key
  - Internal node has at least *t*-1 keys but at most 2*t*-1 keys, i.e. has at least *t* but at most 2*t* children.
- Theorem: $h \leq \log_t (n+1)/2$
- Insertion and deletions:
  - More complicate but still log (n)
  - Split and merge operation.

The simplest B-tree occurs when *t* = 2. Every internal node then has either 2, 3, or 4 children, and we have a *2-3-4 tree*. In practice, however, much larger values of *t* are typically used.

if t = 2 ⇒ 2-3-4 B-Tree

20

Height of the Tree

- The number of disk accesses for a B-tree is proportional to the height of the tree.
- Theorem 18.1

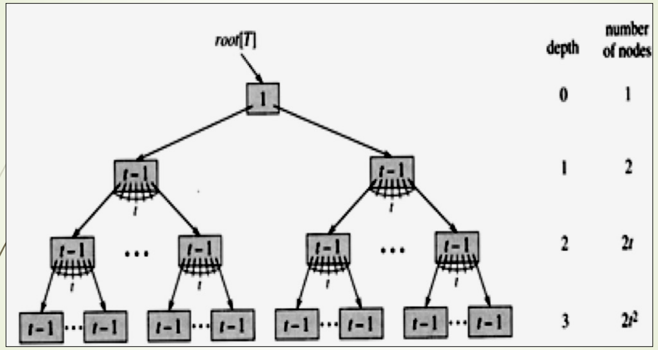
  If n >= 1, then for any n-key B-tree T of height h and minimum degree t >= 2:

  $h \le \log_t (n+1)/2$

- Proof :-

  If a B-tree has height h, the root contains at least one key and all other nodes contain at least (t-1) keys. Thus there are at least 2 nodes at depth 1, at least 2t nodes at depth 2, and so on, until $2t^{h-1}$ nodes at depth h.

Height of the Tree



Height of the Tree

- Above Figure illustrates such a tree for h = 3.

The number of *n* keys satisfies inequality:

$$n >= \overbrace{1}^{\text{root}} + \overbrace{(t-1)}^{\text{min. keys per node}} \overbrace{\sum_{i=1,h} 2t^{i-1}}^{\text{min. number of nodes}}$$

$$= 1+2(t-1)(t^h-1)/(t-1) = 2\,t^h-1$$

$$\mathbf{n >= 2\,t^h-1}$$

$$t^h <= (n+1)/2$$

$$h <= \log_t(n+1)/2$$

Hence the height of the B-tree grows as $O(\log_t n)$, which is significantly slower than the growth of the height of the red-black tree, -- $O(\lg n)$. This means that the number of disk accesses is substantially reduced for most tree operations.

Height of the Tree

RB-Tree	B-Tree
$h \le 2 \lg(n+1)$.	$h <= \log_t(n+1)/2$

- Here we see the power of B-trees, as compared to red-black trees.
- Although the height of the tree grows as $O(\lg n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger.
- Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations.
- Since examining an arbitrary node in a tree usually requires a disk access, the number of disk accesses is substantially reduced.

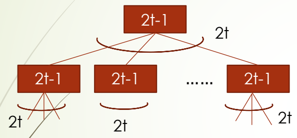
Height of the Tree

- **Why don't we allow a minimum degree of $t = 1$?**
- If we allow t = 1 then, minimum no. of keys in a B-Tree will be t-1 = 0, which is not possible.
- **Show all legal B-trees of minimum degree 2 that represent {1, 2, 3, 4, 5}.**

25

Height of the Tree

- **Derive a tight upper bound on the number of keys that can be stored in a B-tree of height *h* as a function of the minimum degree *t*.**
- For a B-Tree with minimum degree t,



depth	No. of nodes
0	1
1	2t
2	$(2t)^2$
3	$(2t)^3$
-	-
h	$(2t)^h$

- The maximum number of keys in a node is 2t – 1 and the maximum number of keys in a B-Tree of height h →

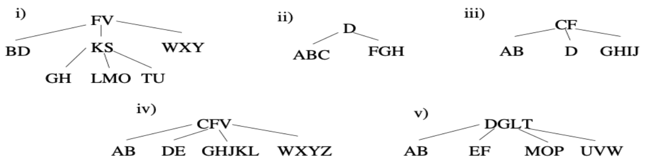
$$n \le (2t-1)\sum_{i=0}^{h}(2t)^i \rightarrow n \le (2t-1) * ((2t)^{h+1} - 1)/(2t-1)$$

$$\rightarrow \boxed{n \le (2t)^{h+1} - 1}$$

26

EXAMPLE

Which of the following are legal B-trees for when the minimum branching factor $t = 3$? For those that are not legal, give one or two sentence very clearly explaining what property was violated.



Basic Operations

- In the basic operations on B-Tree, we adopt two conventions:
- The root of the B-tree is always in main memory, so that a DISK-READ on the root is never required; a DISK-WRITE of the root is required, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.
- All procedures "**one-pass**" algorithms that proceed downward from the root of the tree, without having to back up.

Basic Operations

- The searching algorithm takes as input a pointer to the root node *x* of a subtree, and a key *k*. It returns a pair *(y, i)* such that $key_i[y] = k$.

B-Tree-Search(x,k)

```
i = 1
while i <= n[x] and k > key_i[x]
    i = i + 1
if i <= n[x] and k = key_i[x]
    return (x,i)
if leaf[x]
    return NIL
else
    Disk-Read (c_i[x])     // read i^th child of x
    return B-Tree-Search(c_i[x],k)
```

Start at the leftmost key in the node, and go to the right until you go too far.

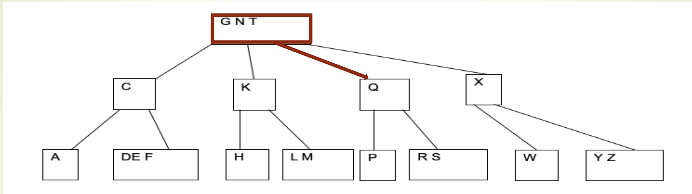
If it is a leaf node, then you are done, as there is no leaf to inspect

Otherwise, retrieve the child node from the disk, and put it into memory

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

29

Search example: S



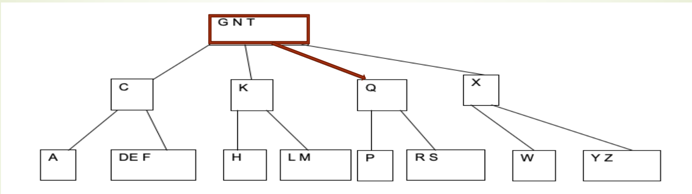

```
i ← 1
while i ≤ n(x) and k > K_x[i]
    i ← i + 1
if i ≤ n(x) and k = K_x[i]
    return (x, i)
if LEAF(x)
    return null
else
    DISKREAD(C_x[i])
    return B-TREESEARCH(C_x[i], k)
```

find the correct location

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

30

Search example: S

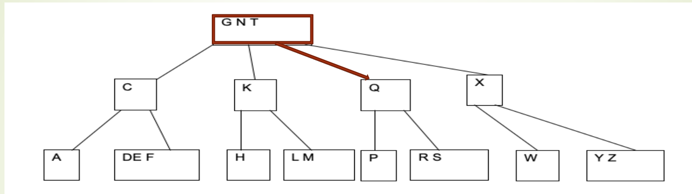


$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if LEAF(x)
    return *null*
else
    DISKREAD$(C_x[i])$
    return B-TREESEARCH$(C_x[i], k)$

This is not a leaf node

31

Search example: S

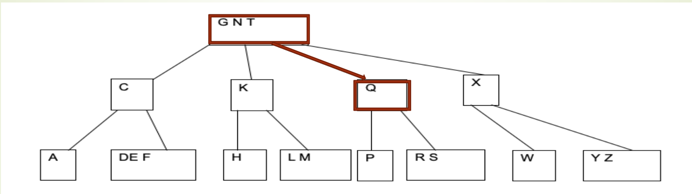


$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if LEAF(x)
    return *null*
else
    DISKREAD$(C_x[i])$
    return B-TREESEARCH$(C_x[i], k)$

This is not a leaf node

32

Search example: S



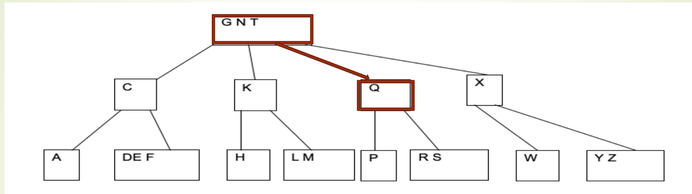

```
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

find the correct location N<S<T

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

33

Search example: S

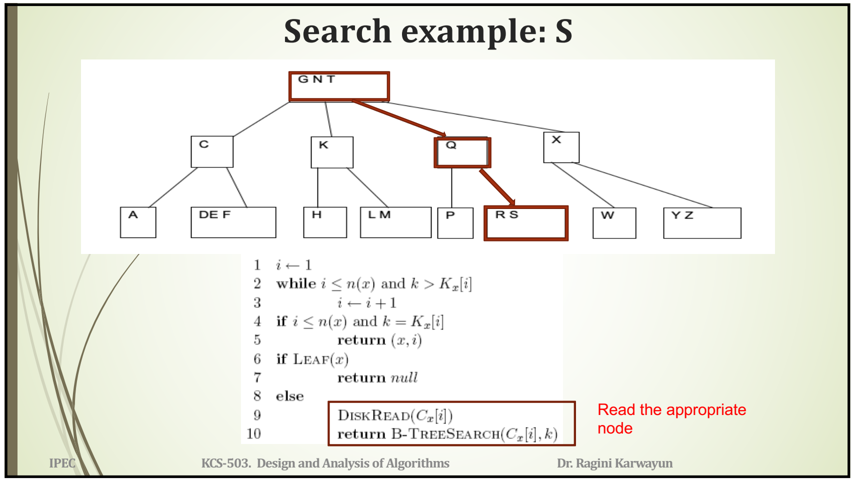



```
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

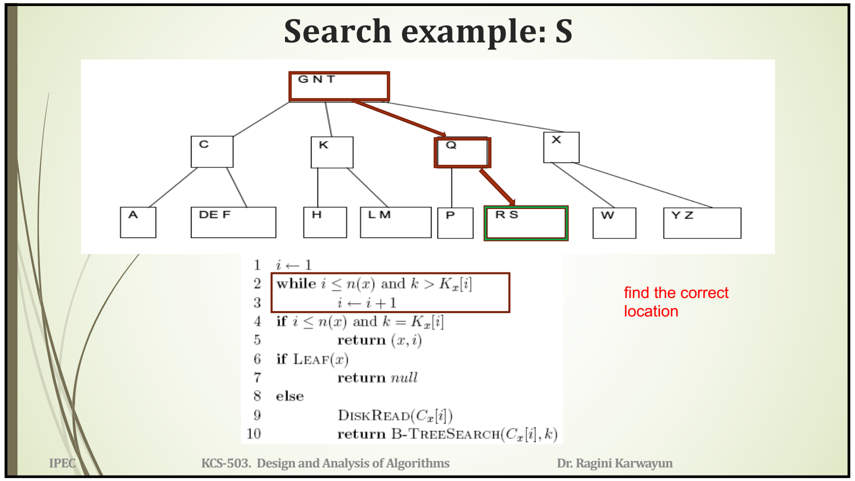
not in this node and this is not a leaf

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

34

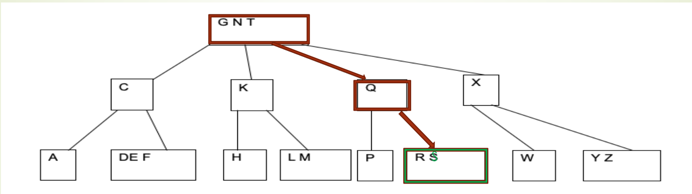

Search example: S
G N T
C
K
Q
X
A
DE F
H
L M
P
R S
W
Y Z
$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
$i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
return (x, i)
if LEAF(x)
return *null*
else
DISKREAD$(C_x[i])$
return B-TREESEARCH$(C_x[i], k)$
Read the appropriate node
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

35


Search example: S
G N T
C
K
Q
X
A
DE F
H
L M
P
R S
W
Y Z
$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
$i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
return (x, i)
if LEAF(x)
return *null*
else
DISKREAD$(C_x[i])$
return B-TREESEARCH$(C_x[i], k)$
find the correct location
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

36

Search example: S




```
1   i ← 1
2   while i ≤ n(x) and k > K_x[i]
3           i ← i + 1
4   if i ≤ n(x) and k = K_x[i]
5           return (x, i)
6   if LEAF(x)
7           return null
8   else
9           DISKREAD(C_x[i])
10          return B-TREESEARCH(C_x[i], k)
```

Key Found

37

Searching: Performance

- The nodes encountered during the recursion form a path downward from the root of the tree.
- The number of disk pages accessed by B-Tree-Search is $O(h) = O(\log_t n)$.
- For each node, n[x] < 2t, hence the while loop 2-3 takes $O(t)$ time.
- Therefore the total CPU time is $O(th) = O(\log_t n)$.

38

Creating an empty B tree

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE()
leaf[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
root[T] ← x
```

B-TREE-CREATE requires *O(1)* disk operations and *O(1)* CPU time.

Inserting in a B tree

- General algorithm:
  - Search for the leaf node *y* at which the new key is to be inserted
- If the node *y* is full (having *2t-1* keys):
  - Split the full node around its median key: $key_t[y]$:
    - Create two nodes with *(t-1)* keys each.
    - Move the median key up to *y*'s parent.
    - If *y*'s parent is also full, make the split again.
- The key is inserted in a single path down the tree.
  - Each full node is split along the way.
  - This assures that when the *y* node needs to be split, its parent cannot be full.

Inserting in a B tree

Every full node encountered in the path from the root to node y is split before proceeding further.

41

Splitting the node

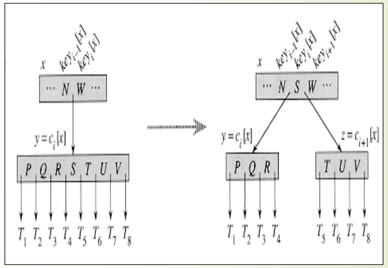
- A fundamental operation used during insertion is the *splitting* of a full node *y* (having 2*t* - 1 keys) around its *median key* $key_{[y]}$ into two nodes having *t* - 1 keys each.
- The median key moves up into *y*'s parent--which must be non-full prior to the splitting of *y*--to identify the dividing point between the two new trees;
- if *y* has no parent, then the tree grows in height by one.
- Splitting, is the means by which the tree grows.
- **The procedure B-TREE-SPLIT-CHILD takes as input a *non-full* internal node *x* (assumed to be in main memory), an index *i*, and a node *y* such that $y = c_i[x]$ is a *full* child of *x*. The procedure then splits this child in two and adjusts *x* so that it now has an additional child**

42

Splitting the node

Figure illustrates this process.

The full node *y* is split about its median key *S*, which is moved up into *y*'s parent node *x*. Those keys in *y* that are greater than the median key are placed in a new node *z*, which is made a new child of *x*.



Splitting the node

B-Tree-Split-Child(x,i,y)

```
z = Allocate-Node()      // allocate a disk page
leaf[z] = leaf[y]
n[z] = t-1
for j = 1 to t-1
    key_j[z] = key_{j+t}[y]
if not leaf[y]
    for j = 1 to t
        c_j[z] = c_{j+t}[y]
n[y] = t-1
for j = n[x]+1 downto i+1  // shift children to the right
    c_{j+1}[x] = c_j[x]
c_{i+1}[x] = z              // add z as a new child
```

Splitting the node (cont.)

// make room for the median

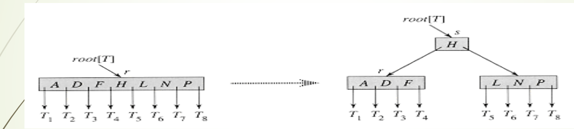
```
13 for j = n[x] downto i
14    key_{j+1}[x] = key_j[x]
15 key_i[x] = key_t[y]
16 n[x]++
17 Disk-Write(y)
18 Disk-Write(z)
19 Disk-Write(x)
```

- The CPU time is determined by loops 4-5 and 7-8, which is Θ(t). Note that other loops perform O(t) iterations. The procedure performs Θ(1) disk operations.

Inserting a key into a B-tree

Inserting a key k into a B-tree T of height h is done in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$.

The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.



Splitting the root with t = 4. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

Splitting a full node encountered before proceeding further ensures that whenever we split a node and move the median up into the parent, it is never full. **Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.**

Inserting a Key: Algorithm

```
B-Tree-Insert(T,k)
   r = root[T]
   if n[r] = 2t-1 // full node
      s = Allocate-Node()
      root[T] = s
      leaf[s] = FALSE
      n[s] = 0
      c1[s] = r
      B-Tree-Split-Child(s,1,r)      // split the old root
      B-Tree-Insert-Nonfull(s,k)
   else
      B-Tree-Insert-Nonfull(r,k)
```

If the node has 2t-1 keys, it can't accept any more keys, so you need to split it into 2 nodes Before doing the insert.

Otherwise, call Nonfull()

Inserting a Key: Algorithm (cont.)

// Insert key *k* into a non-full node *x*

```
B-Tree-Insert-Nonfull(x,k)
  i = n[x]
  if leaf[x] // k is inserted in the ordered list
  { 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)
  }
```

If x is the leaf node

All the keys greater than k are shifted one place towards the right.

key k is inserted in the proper place, no of keys on node x are incremented by 1, and the node is written onto the disk.

Inserting a Key: Algorithm (cont.)

```
else // search the leaf to insert into
{ while i >= 1 and k < key_i[x]
        i--
  i++
  Disk-Read(c_i[x])
  if n[c_i[x]] = 2t-1 // full node
     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 not a leaf node, descend to the appropriate node into memory

If the node selected is full then call split-node function.

Inserting a Key: Performance

- The number of disk accesses performed by B-Tree-Insert is $O(h)$ for a B-tree of height h.
  - Only a $O(1)$ of Disk-Read and Disk-Write operations are performed at each level in the B-Tree-Insert-Nonfull.
- The total CPU time is $O(t\,h) = O(\log_t n)$
  - At each level of the tree the number of CPU operations are determined by *while* loops in B-Tree-Insert-Nonfull.
  - The maximum number of iterations in these loops are $2t-1$, hence the total time at each level is $O(t)$.

Inserting a Key: Example (t = 2)

G C N A H E K Q M F W L T Z D P R X Y S

G

51

Inserting a Key: Example (t = 2)

G C N A H E K Q M F W L T Z D P R X Y S

C G

52

Inserting a Key: Example (t = 2)

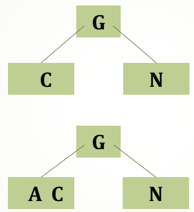
G C N A H E K Q M F W L T Z D P R X Y S

C G N

53

Inserting a Key: Example (t = 2)

G C N A H E K Q M F W L T Z D P R X Y S



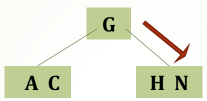
Node is full so split

Now insert A

54

Inserting a Key: Example (t = 2)

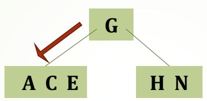
G C N A H E K Q M F W L T Z D P R X Y S



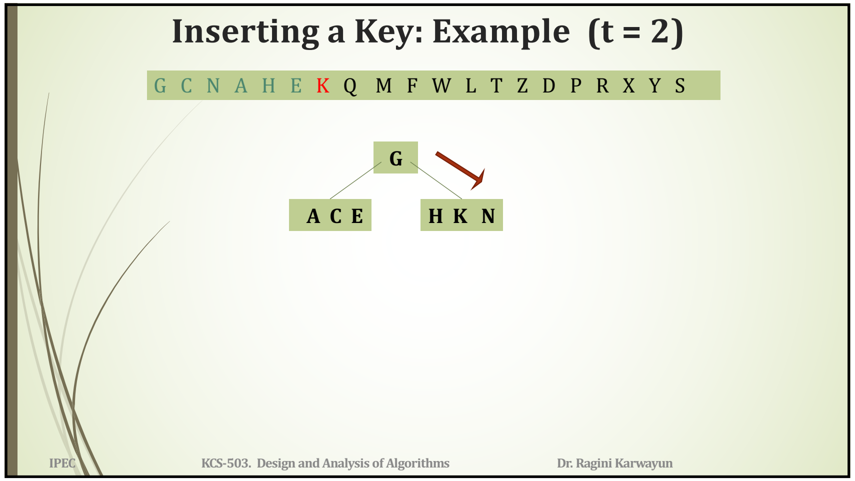
55

Inserting a Key: Example (t = 2)

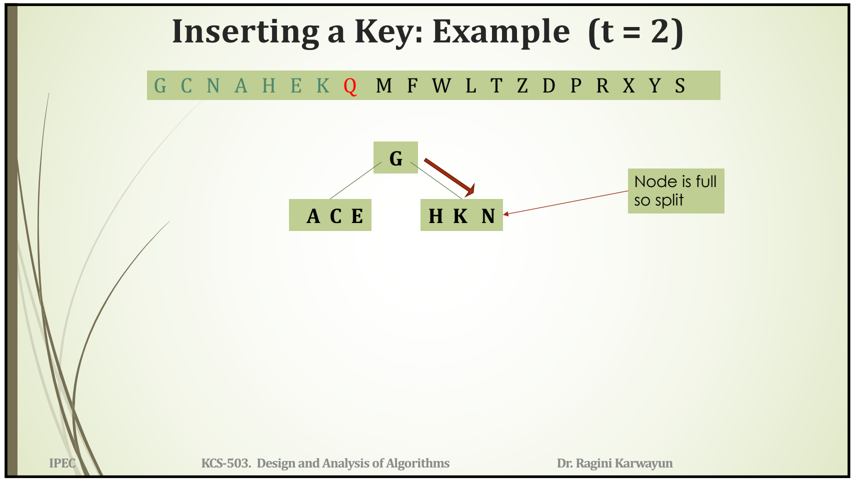
G C N A H E K Q M F W L T Z D P R X Y S



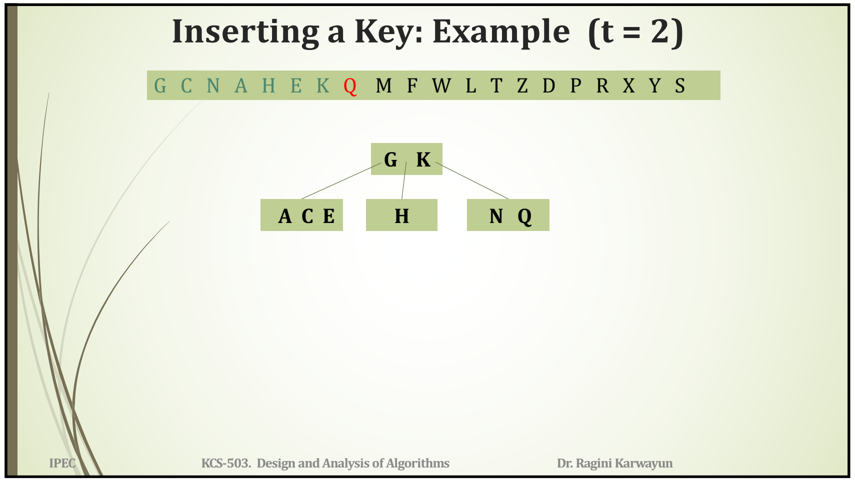
56


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G
A C E
H K N
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

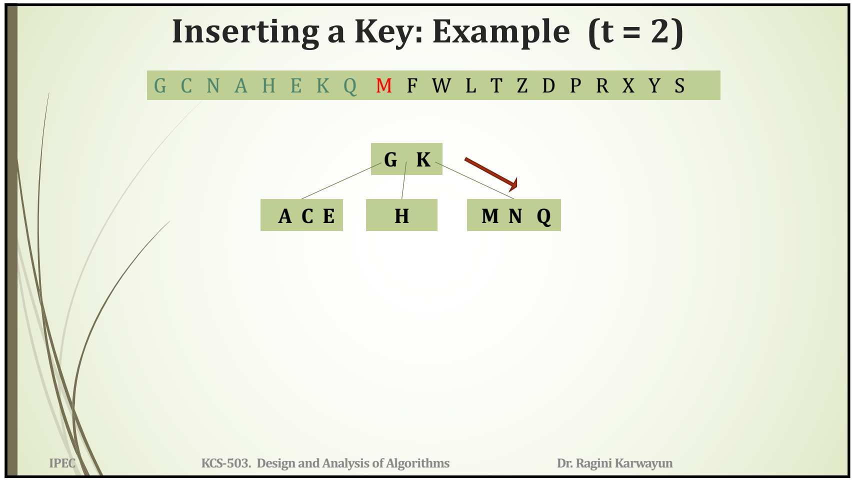
57


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G
A C E
H K N
Node is full so split
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

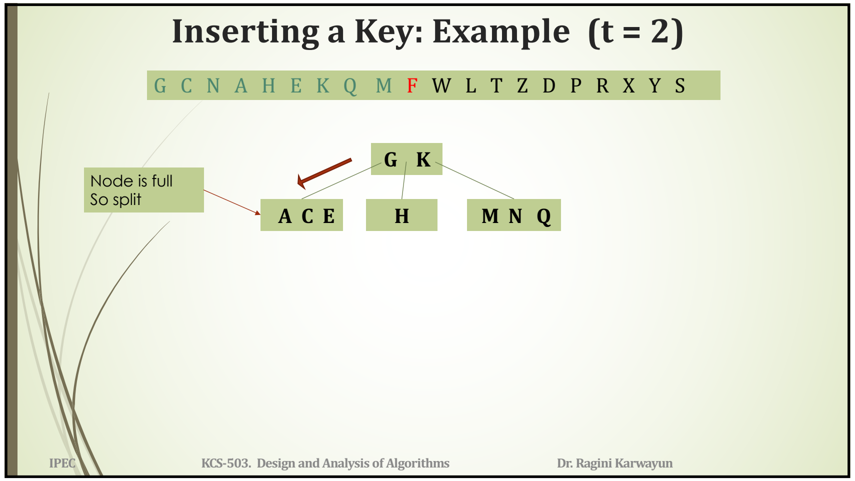
58


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G K
A C E
H
N Q
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

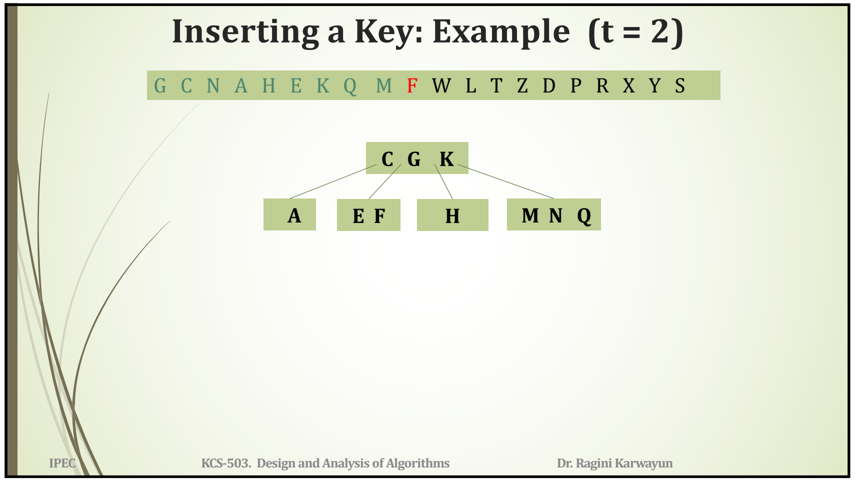
59


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G K
A C E
H
M N Q
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

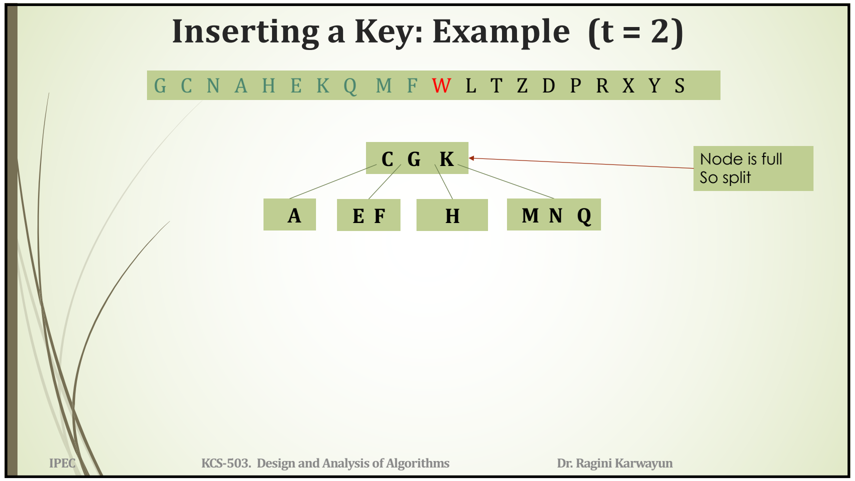
60


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G K
Node is full
So split
A C E
H
M N Q
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

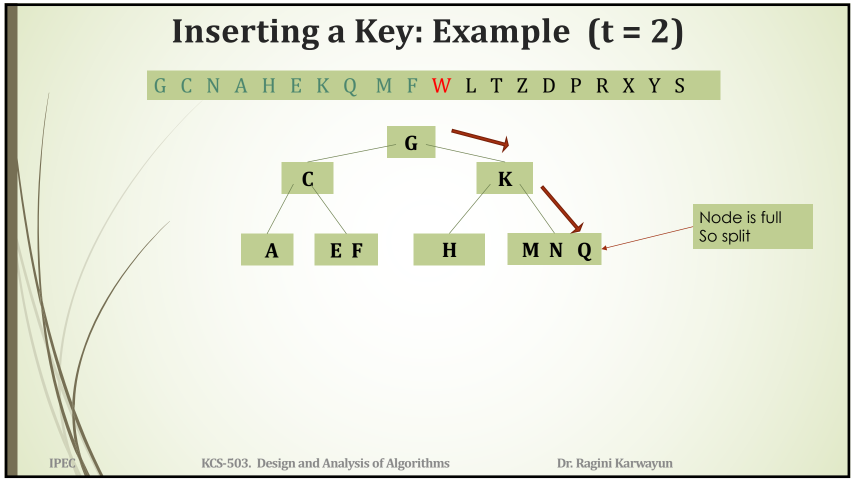
61


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
C G K
A
E F
H
M N Q
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

62


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
C G K
A
E F
H
M N Q
Node is full
So split
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

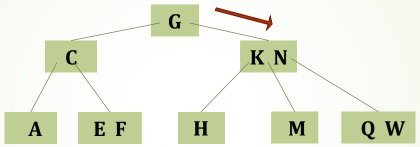
63


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G
C
K
A
E F
H
M N Q
Node is full
So split
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

64

Inserting a Key: Example (t = 2)

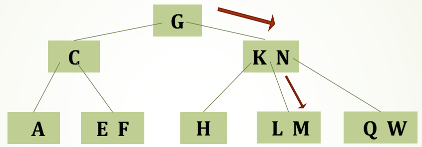
G C N A H E K Q M F W L T Z D P R X Y S



65

Inserting a Key: Example (t = 2)

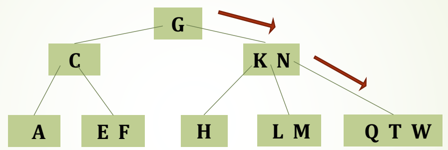
G C N A H E K Q M F W L T Z D P R X Y S



66

Inserting a Key: Example (t = 2)

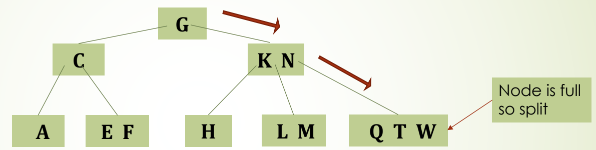
G C N A H E K Q M F W L T Z D P R X Y S



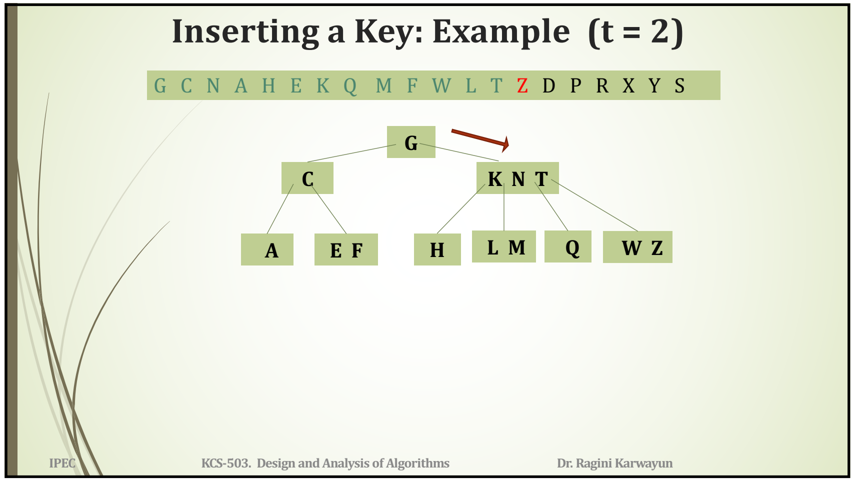
67

Inserting a Key: Example (t = 2)

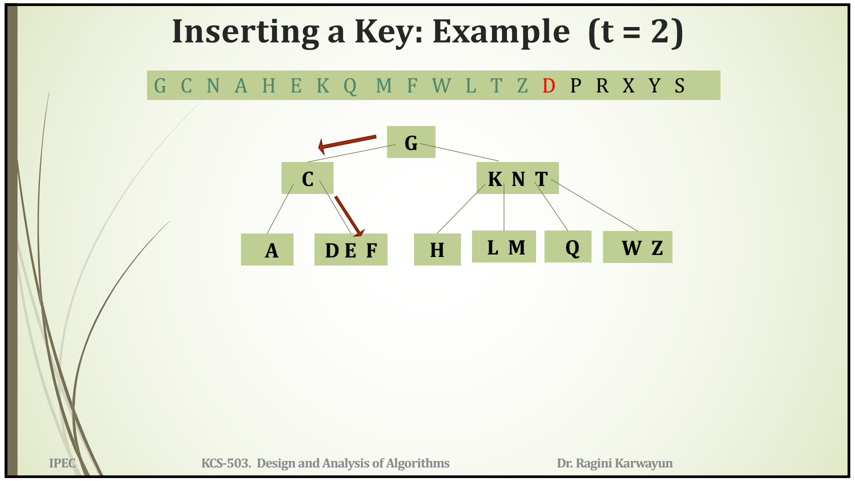
G C N A H E K Q M F W L T Z D P R X Y S



68


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G
C
K N T
A
E F
H
L M
Q
W Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

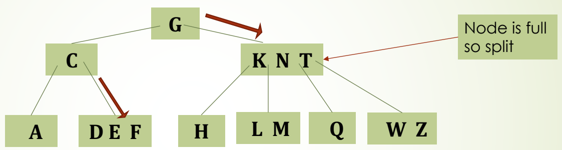
69


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G
C
K N T
A
D E F
H
L M
Q
W Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

70

Inserting a Key: Example (t = 2)

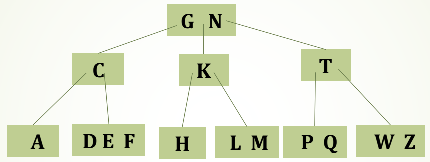
G C N A H E K Q M F W L T Z D P R X Y S



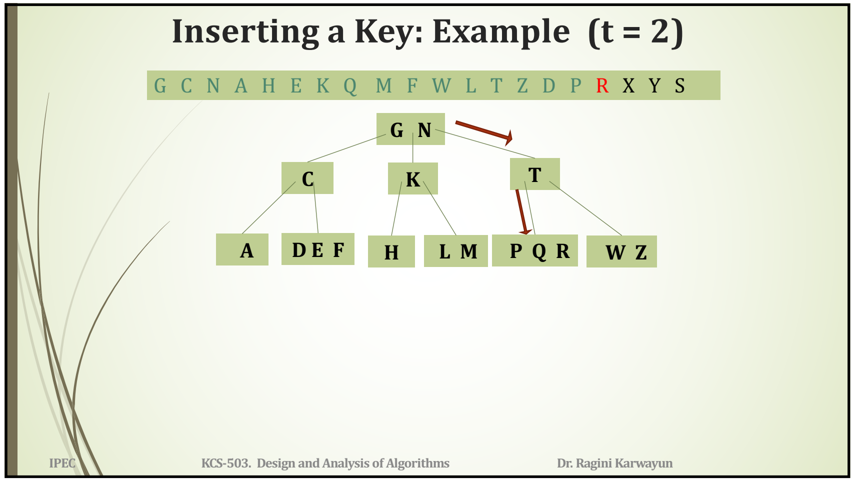
71

Inserting a Key: Example (t = 2)

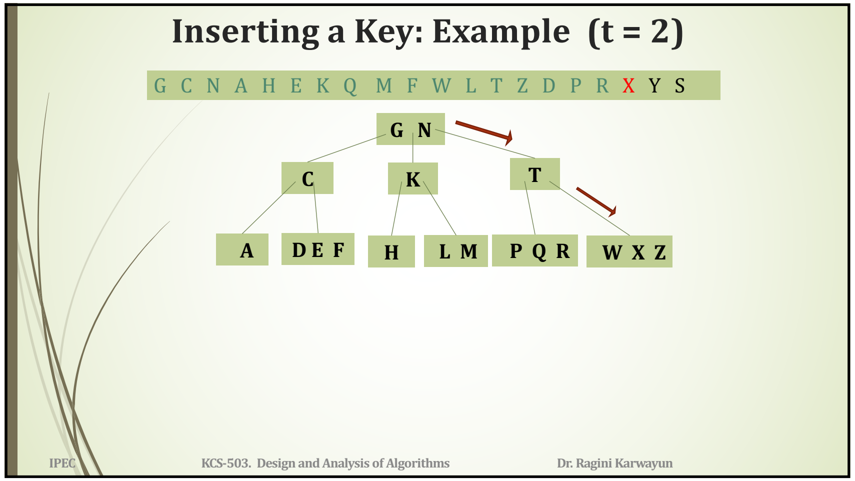
G C N A H E K Q M F W L T Z D P R X Y S



72

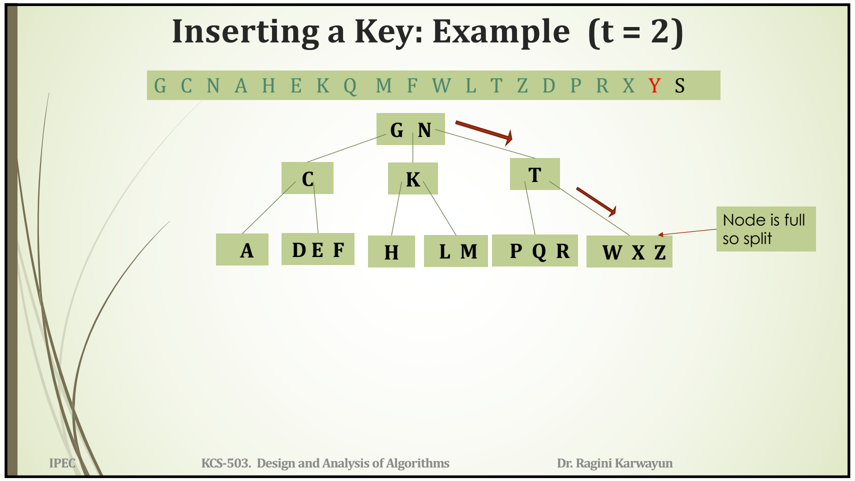

Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G N
C
K
T
A
DE F
H
L M
P Q R
W Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

73


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G N
C
K
T
A
DE F
H
L M
P Q R
W X Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

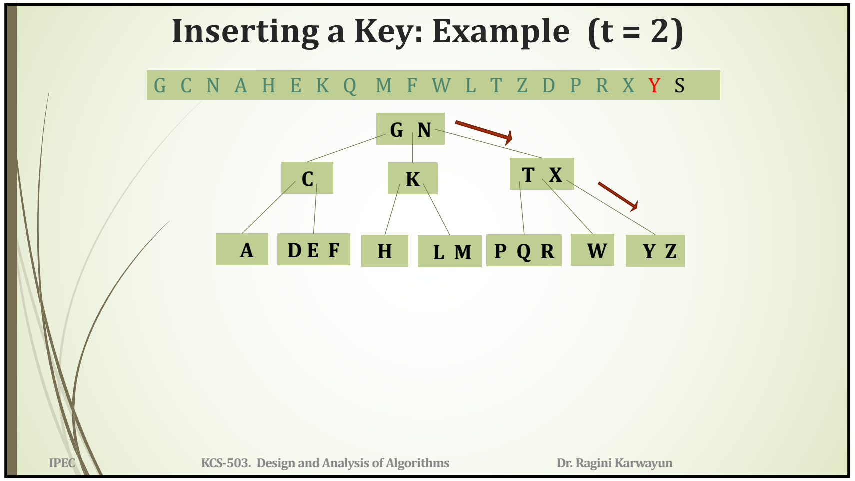
74

Inserting a Key: Example (t = 2)

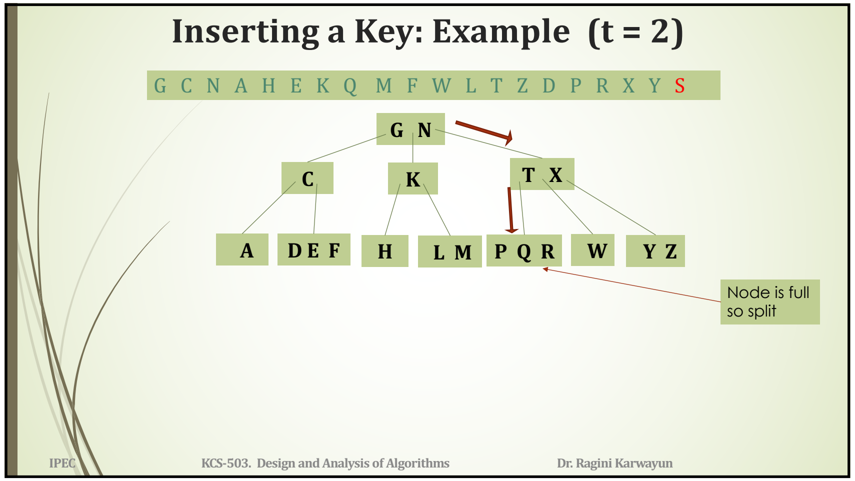


75

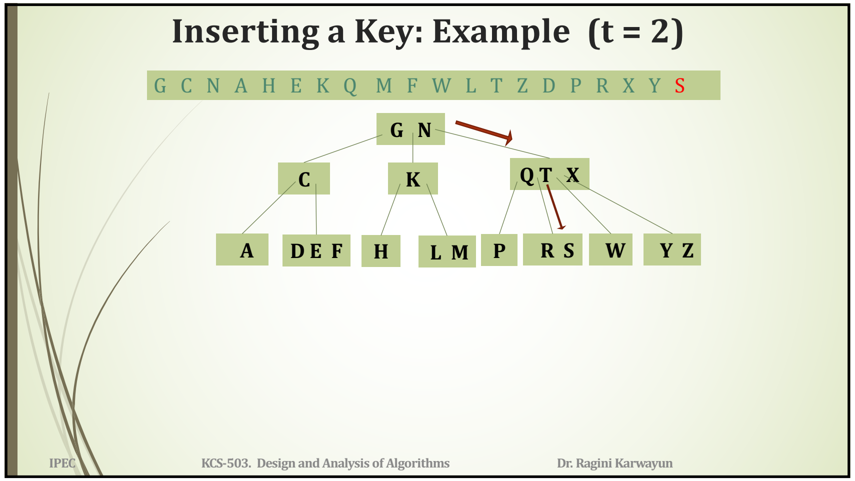
Inserting a Key: Example (t = 2)



76


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G N
C
K
T X
A
D E F
H
L M
P Q R
W
Y Z
Node is full so split
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

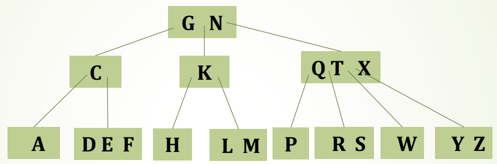
77


Inserting a Key: Example (t = 2)
G C N A H E K Q M F W L T Z D P R X Y S
G N
C
K
Q T X
A
D E F
H
L M
P
R S
W
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

78

Inserting a Key: Example (t = 2)

G C N A H E K Q M F W L T Z D P R X Y S

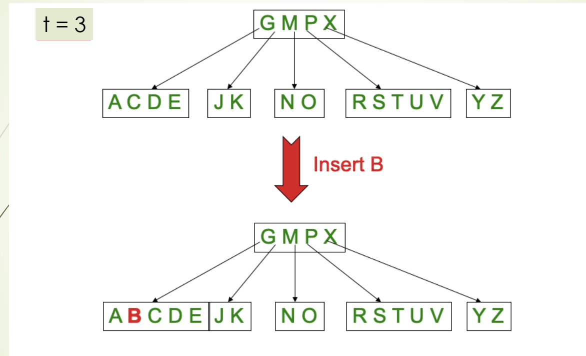


FINAL TREE

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

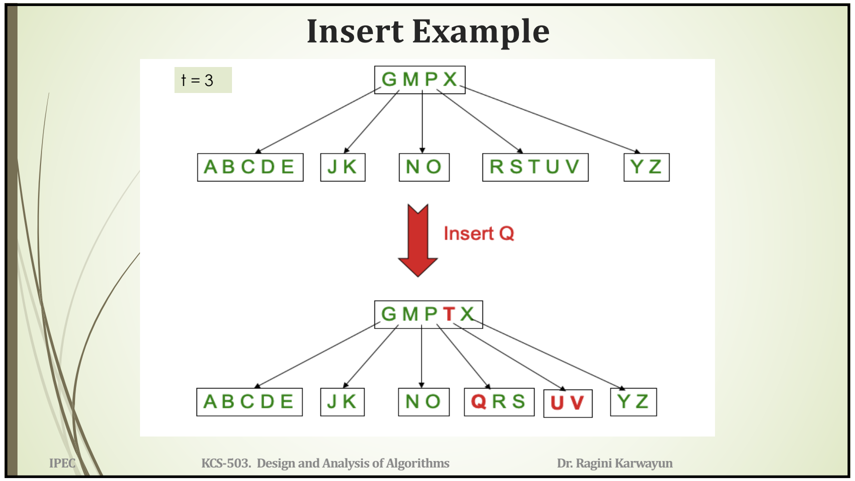
79

Insert Example

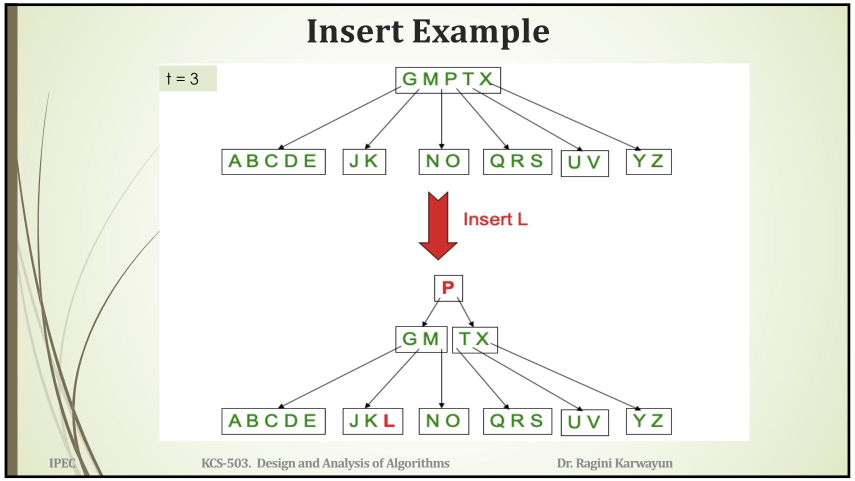


IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

80

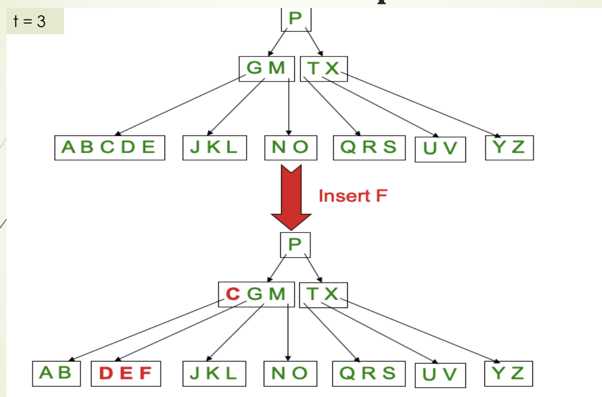

Insert Example
t = 3
G M P X
A B C D E
J K
N O
R S T U V
Y Z
Insert Q
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

81


Insert Example
t = 3
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z
Insert L
P
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

82

Insert Example



83

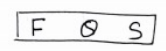
Solved Example

Insert the following keys in an initially empty B-Tree with (i) t=2 (ii) t=3.

F S Q K C L H T V W M R N P A B X Y D Z E.

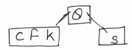
(i) t=2 Max. no. of keys = 2t-1 = 3.
Min no. of keys = t-1 = 1

* F-S-Q



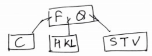
* K → Node is full → so split & then insert K in its proper place.

* C

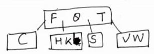


* L → Node full → so split & then insert

*H, T, V →

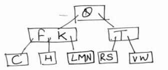


* W → Node full so split & then insert

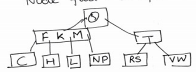


* M → As root is full, so split root & then descend to the appropriate node. If it is also full then split it too & proceed.

*RN

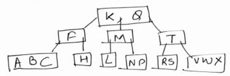


* P → Node full - so split - - .



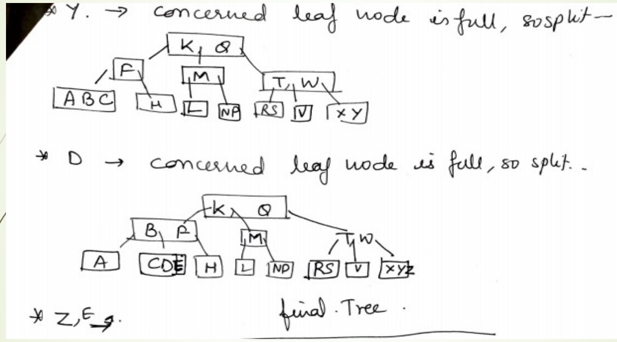
* A → Even though the leaf node where A is to be inserted is non-full, but along the way from root to the leaf the node FKM is full, so first split that node & proceed.

*B, X →



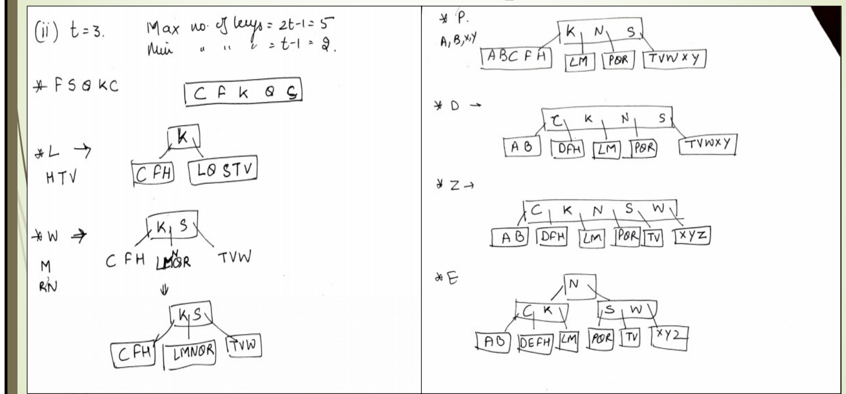
84

Solved Example



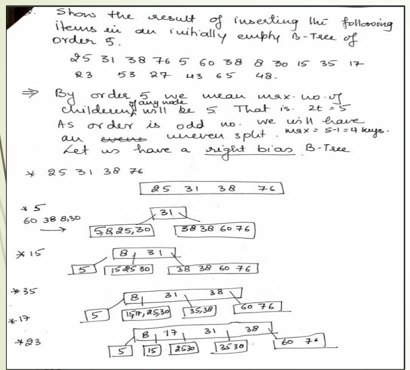
85

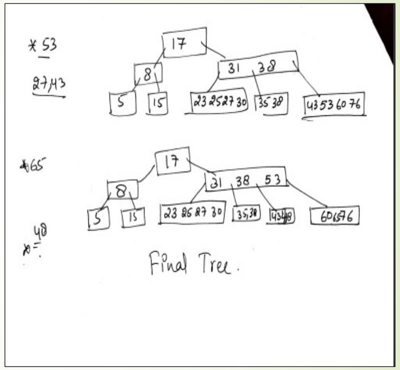
Solved Example



86

Solved Example





87

Important Points

- Insertion in B-Tree can be performed in two ways :
  - Proactive approach : With aggressive splitting, this technique splits any full node as soon as it is encountered in the search for the location in which the item is to be added. The proactive technique has the benefit that the algorithm will visit the nodes only once in cases where split(s) are required, thus giving better performance. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.
  - Reactive approach : Without aggressive splitting, this algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. If we don't split a node before going down to it and split it only if a new key is inserted, we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from the root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because the parent was already full). This cascading effect never happens in the proactive insertion algorithm.

88

Deletion in B-Tree

- Let us assume that procedure B-TREE-DELETE is asked to delete the key k from the subtree rooted at x.
- **This procedure is structured to guarantee that whenever B-TREE-DELETE is called recursively on a node x, the number of keys in x is at least the minimum degree t.**
- Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up".

Deletion in B-Tree

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.

   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.)

   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. Then, free z and recursively delete k from y.

Deletion in B-Tree

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. 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.

   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 $c_i[x]$.

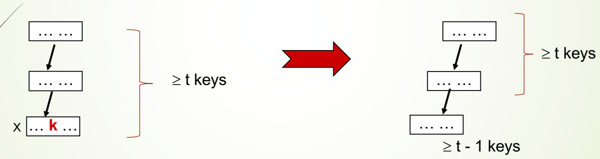
   b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

91

B-Tree Deletion Cases

Case 1:

k is in node x, x is a leaf and all the nodes passed through from the root to node x have at least t keys → delete k from x.

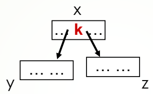


92

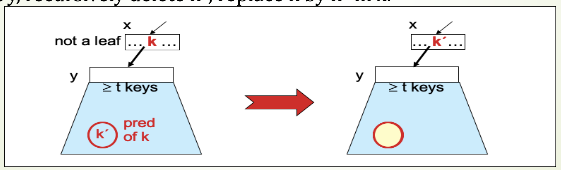
B-Tree Deletion Cases

Case 2:

k is in node x, x is an internal node. We can't simply delete k even if n[x] ≥ t
– we have to replace k with another key to separate the child of x before k and the child after k



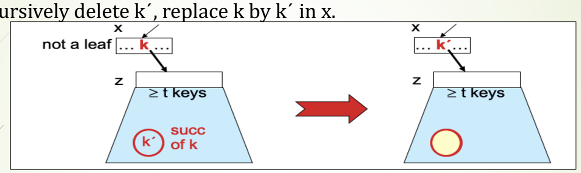
Case 2 a : node y has at least t keys -- find predecessor k´ of k in subtree rooted at y, recursively delete k´, replace k by k´ in x.



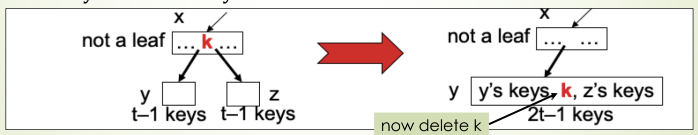
93

B-Tree Deletion Cases

Case 2 b : node z has at least t keys -- find successor k´ of k in subtree rooted at z, recursively delete k´, replace k by k´ in x.



Case 2 c : nodes y and z both have t–1 keys -- merge k and z into y, free z, recursively delete k from y.



94

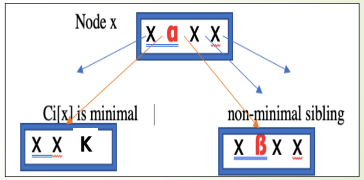
B-Tree Deletion Cases

Case 3:

k is not in an internal node. Let $c_i[x]$ be the root of the subtree that must contain k, if k is in the tree. If ci[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.

3a : $c_i[x]$ has t–1 keys, some sibling (left or right) has at least t keys.

1. Move α down from the node x (parent of $c_i[x]$) to $c_i[x]$.
2. Move β from the non-minimal sibling of $c_i[x]$ up to the parent to take the place of α.
3. Move the child ptr., if there is one, next to β over to $c_i[x]$ (note if $c_i[x]$ is a leaf, then it and its siblings will contain no ptrs.)

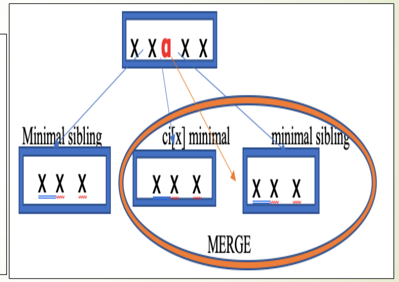


95

B-Tree Deletion Cases

3b: $c_i[x]$ and sibling both have t–1 keys. If each sibling is minimal, then the node is merged with one sibling (doesn't matter which) together with the separating key from the parent

1. Copy data (keys/ptrs) from one sibling to $c_i[x]$.
2. Move the separating key α from the parent down to the median position in $c_i[x]$ so that it separates the original keys in $c_i[x]$ from the new ones copied in.
3. *Note*: We can move α down if node *x* is not the root as *x* then is guaranteed not minimal; the only exceptional case is when x is the root *and* α is its *only* key, in which case the root is deleted and $c_i[x]$ becomes the new root and the B-tree becomes shorter.
4. Delete the chosen sibling. Delete the pointer to the chosen sibling.The above merges $c_i[x]$ with its sibling and α.



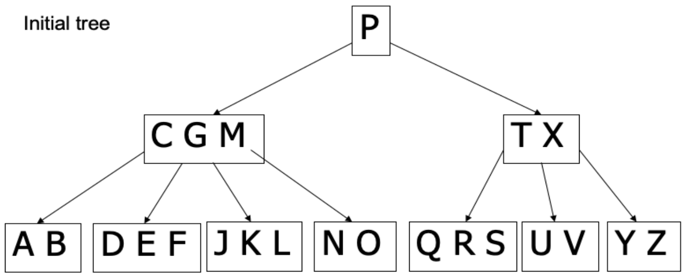
96

t = 3

B-Tree Deletion Example

Delete F: case 1

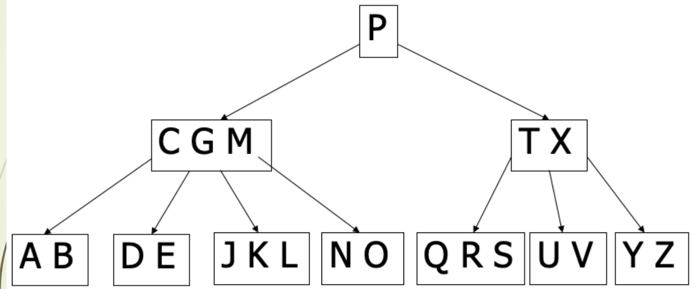
Initial tree



97

t = 3

B-Tree Deletion Example

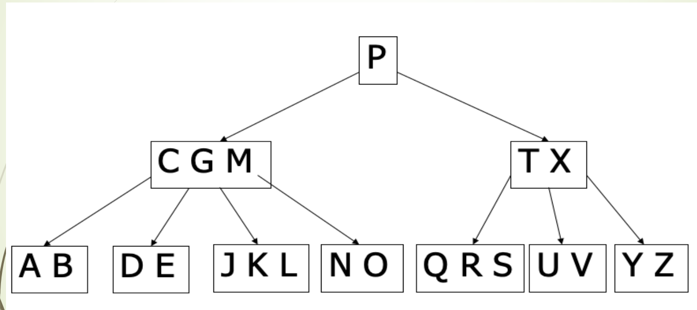


98

t = 3

B-Tree Deletion Example

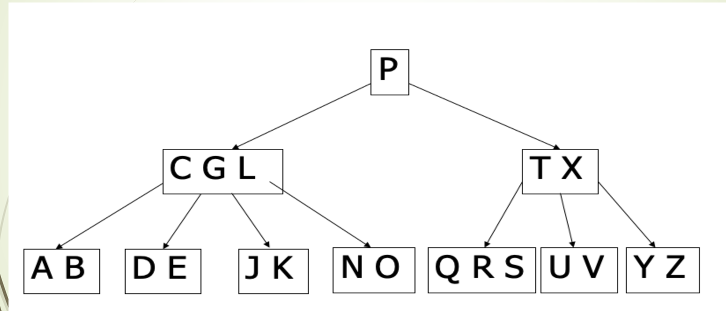
Delete M : Case 2a



99

t = 3

B-Tree Deletion Example

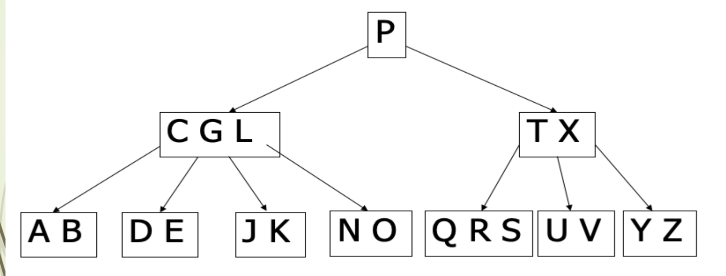


100

t = 3

B-Tree Deletion Example

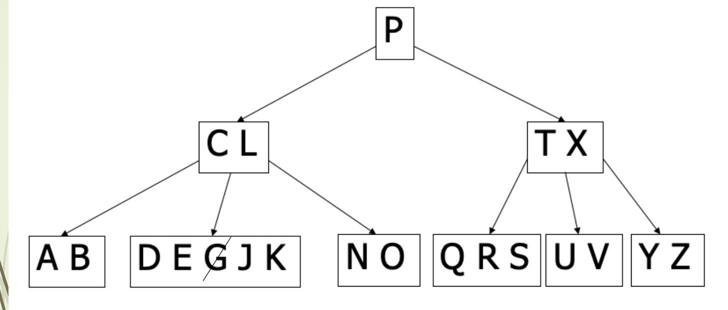
Delete G : Case 2c



101

t = 3

B-Tree Deletion Example

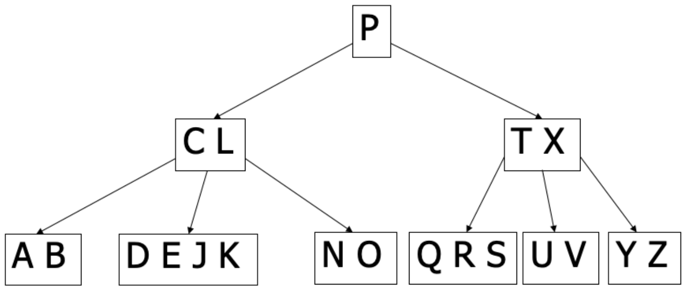


102

t = 3

B-Tree Deletion Example

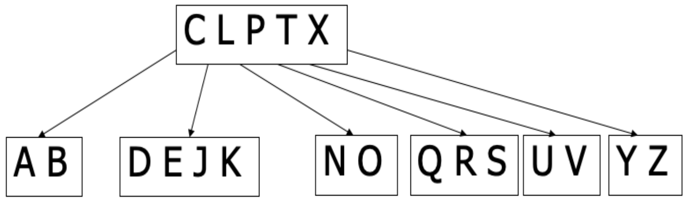
Delete D : Case 3b



103

t = 3

B-Tree Deletion Example

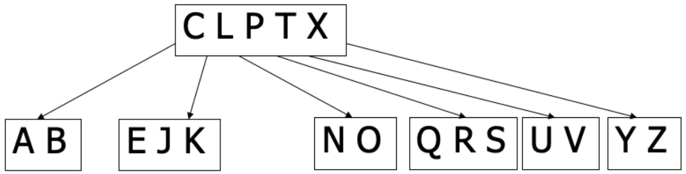


104

t = 3

B-Tree Deletion Example

Delete B: Case 3a

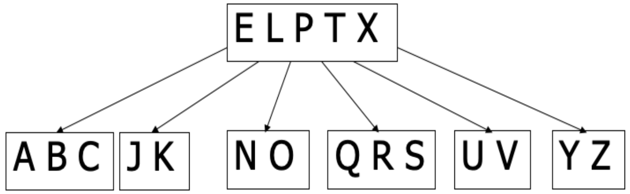


105

t = 3

B-Tree Deletion Example

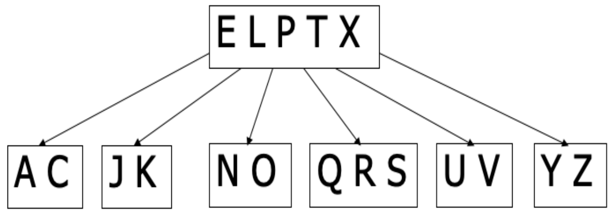
Delete B: Case 3a



106

t = 3

B-Tree Deletion Example

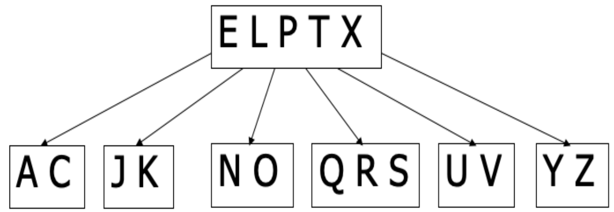


107

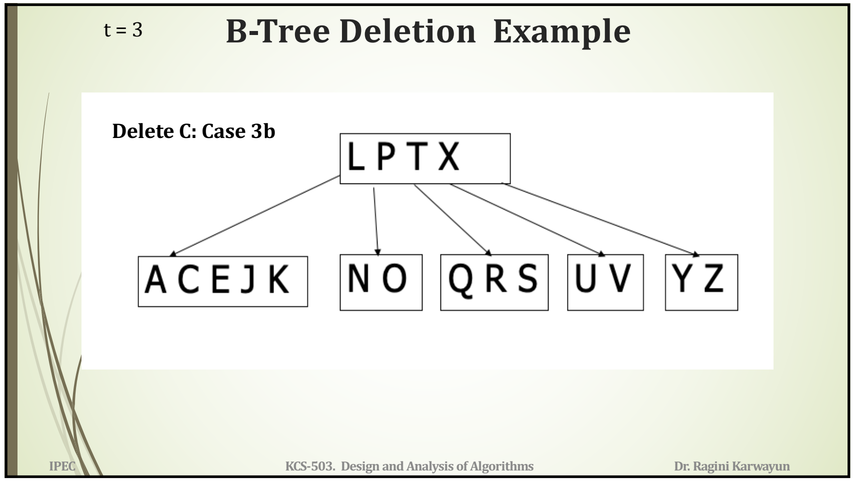
t = 3

B-Tree Deletion Example

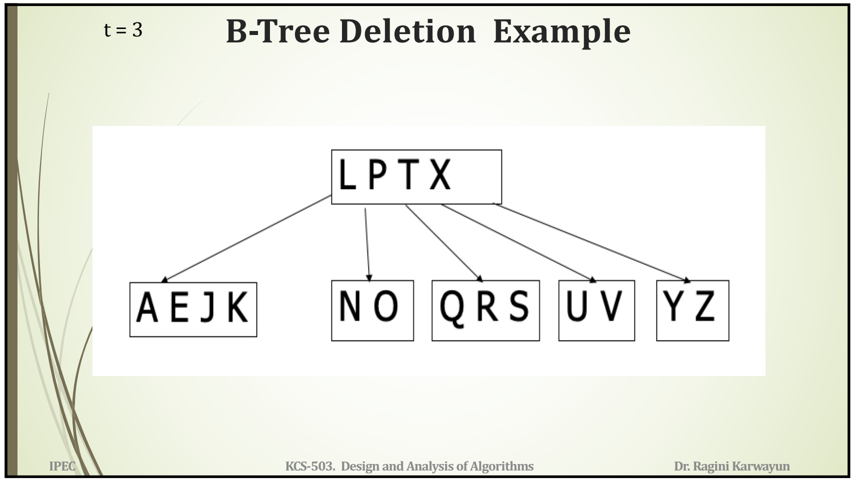
Delete C: Case 3b



108


t = 3
B-Tree Deletion Example
Delete C: Case 3b
L P T X
A C E J K
N O
Q R S
U V
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

109

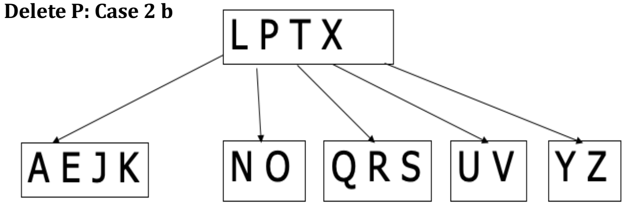

t = 3
B-Tree Deletion Example
L P T X
A E J K
N O
Q R S
U V
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

110

t = 3

B-Tree Deletion Example

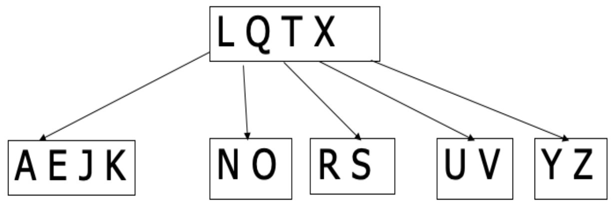
Delete P: Case 2 b



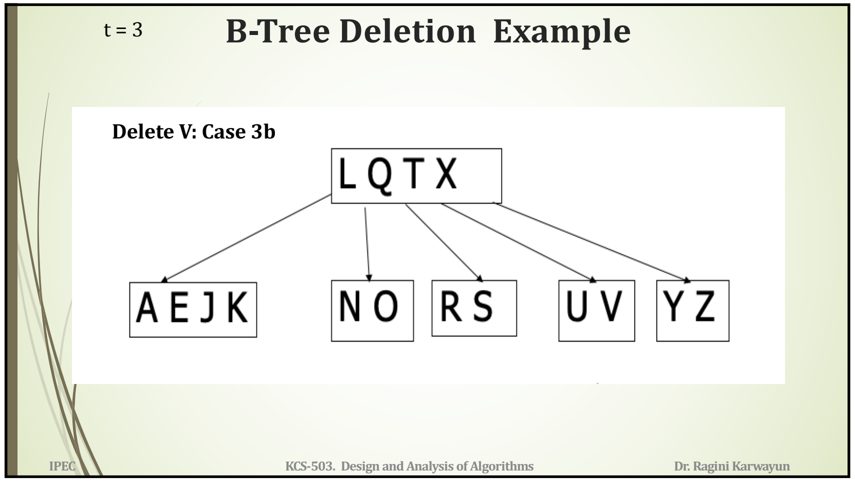
111

t = 3

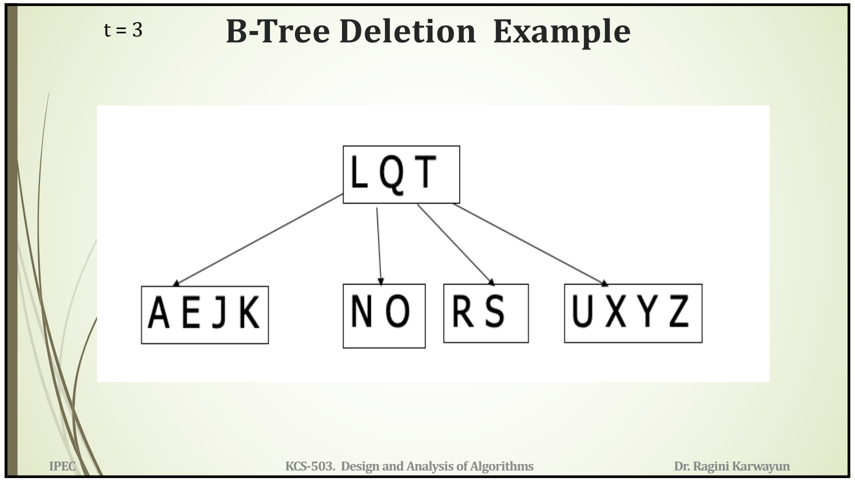
B-Tree Deletion Example



112


t = 3
B-Tree Deletion Example
Delete V: Case 3b
L Q T X
A E J K
N O
R S
U V
Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

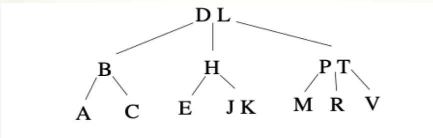
113


t = 3
B-Tree Deletion Example
L Q T
A E J K
N O
R S
U X Y Z
IPEC
KCS-503. Design and Analysis of Algorithms
Dr. Ragini Karwayun

114

B-Tree Deletion Example

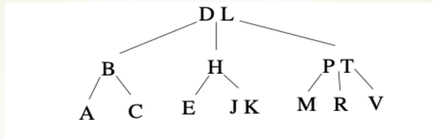
Show the B-tree the results when deleting A, V and P successively from the following B-tree with a minimum branching factor of t = 2.



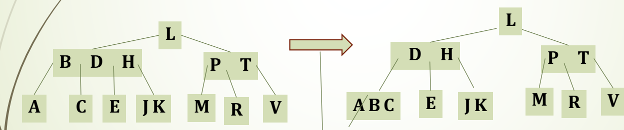
115

B-Tree Deletion Example

Delete A



- As node B has t-1 keys we cannot proceed further.
- Its sibling H is also having t-1 keys, So apply case 3b

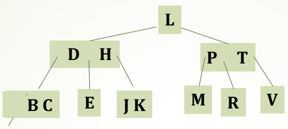


As node A has t-1 keys and its sibling also has t-1 keys so apply case 3-b

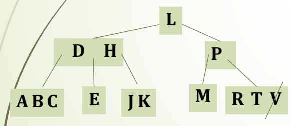
116

B-Tree Deletion Example

Delete V



As node V has t-1 keys and its sibling also has t-1 keys so apply case 3-b



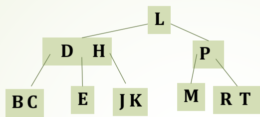
Now delete V

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

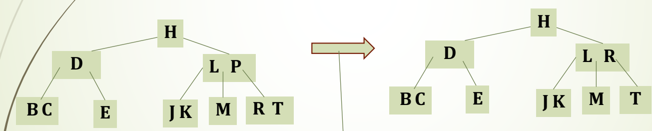
117

B-Tree Deletion Example

Delete P



As node P has t-1 keys and its sibling has t keys so borrow from it & apply case 3-a



Now P in in internal node & has a right child with t keys, so borrow from it and apply case 2b

IPEC KCS-503. Design and Analysis of Algorithms Dr. Ragini Karwayun

118