B-Trees

David Kauchak

cs302

Spring 2013

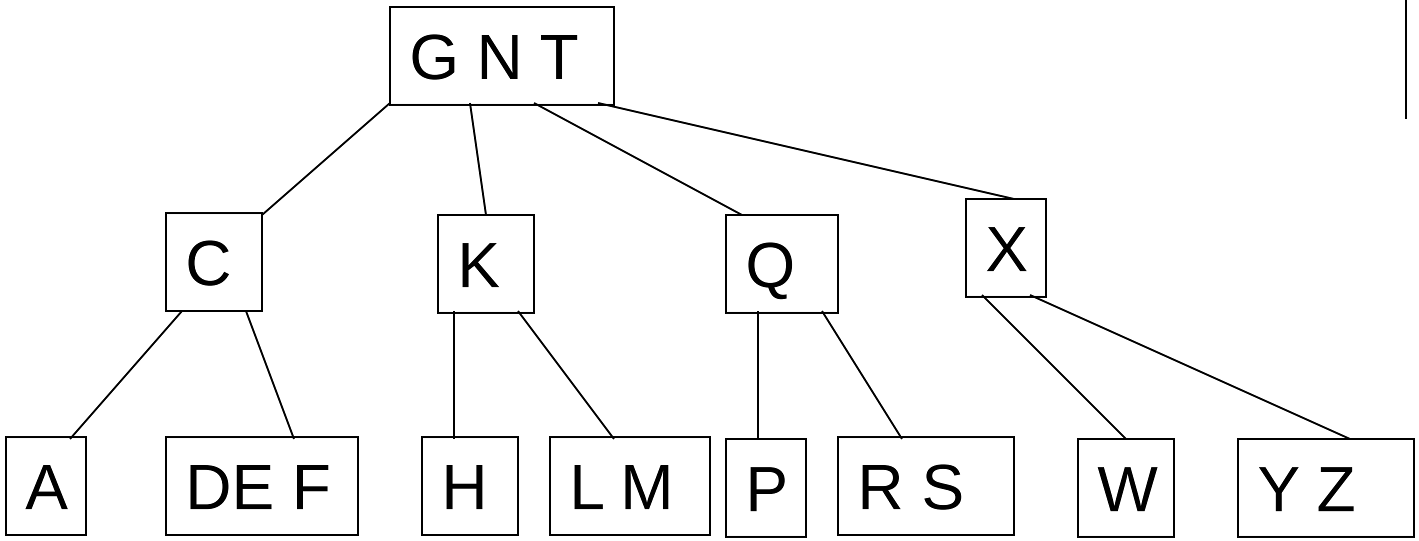
Admin

- Homework 10 out today
- Midterm out Monday/Tuesday
  - Available online
  - 2 hours
  - Will need to return it to me within 3 hours of downloading
  - Must take by Friday at 6pm
- Review on Tuesday
  - E-mail if you have additional topics you'd like covered

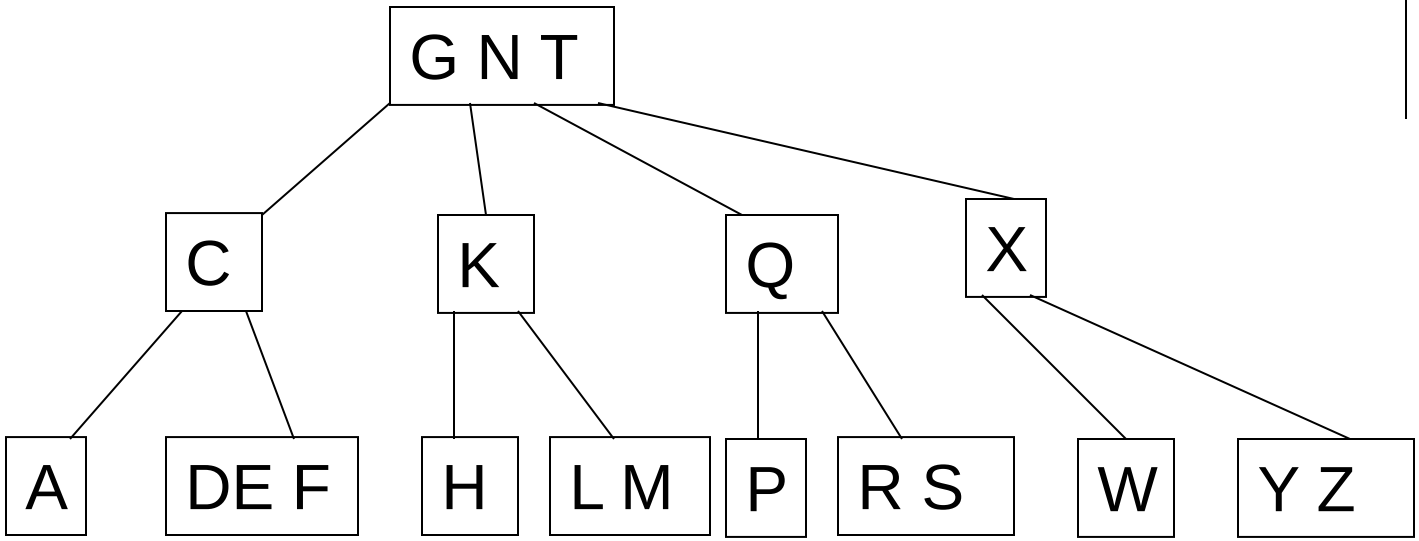
B-tree

- Defined by one parameter: *t*
- Balanced n-ary tree
- Each node contains between *t*-1 and *2t*-1 keys/data values (i.e. multiple data values per tree node)
  - keys/data are stored in **sorted order**
  - one exception: root can have < t-1 keys
- Each internal node contains between *t* and 2*t* children
  - the keys of a parent **delimit** the values of the children keys
  - For example, if key_i = 15 and key_{i+1} = 25 then child *i* + 1 must have keys between 15 and 25
- all leaves have the same depth

Example B-tree: t = 2

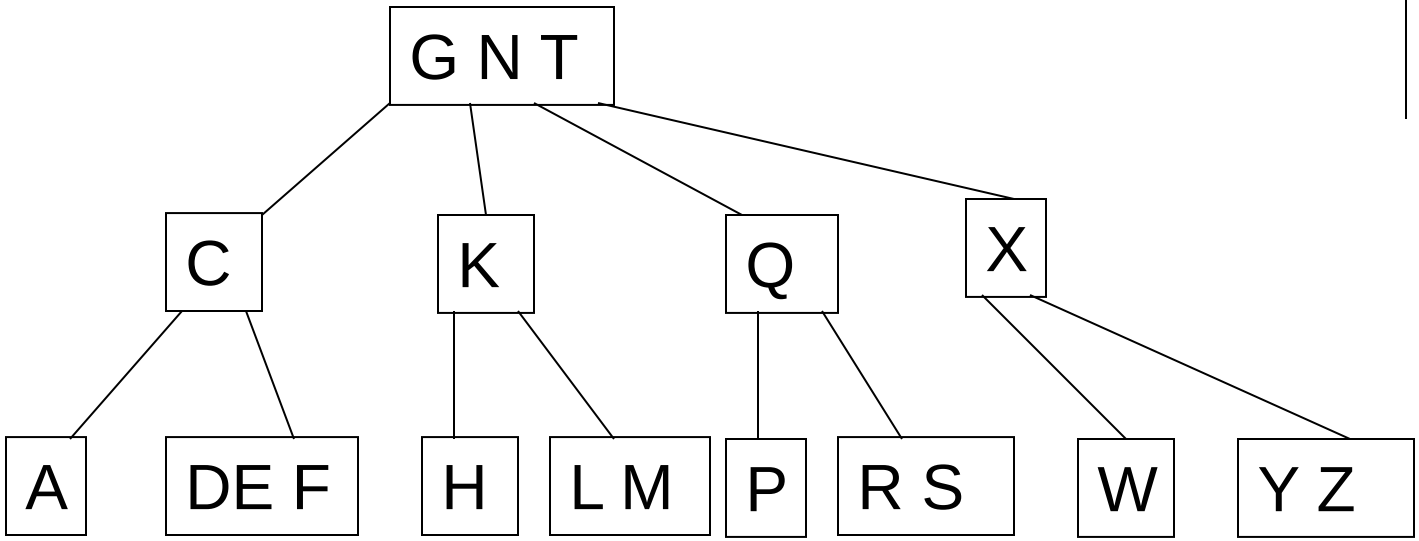


Example B-tree: t = 2



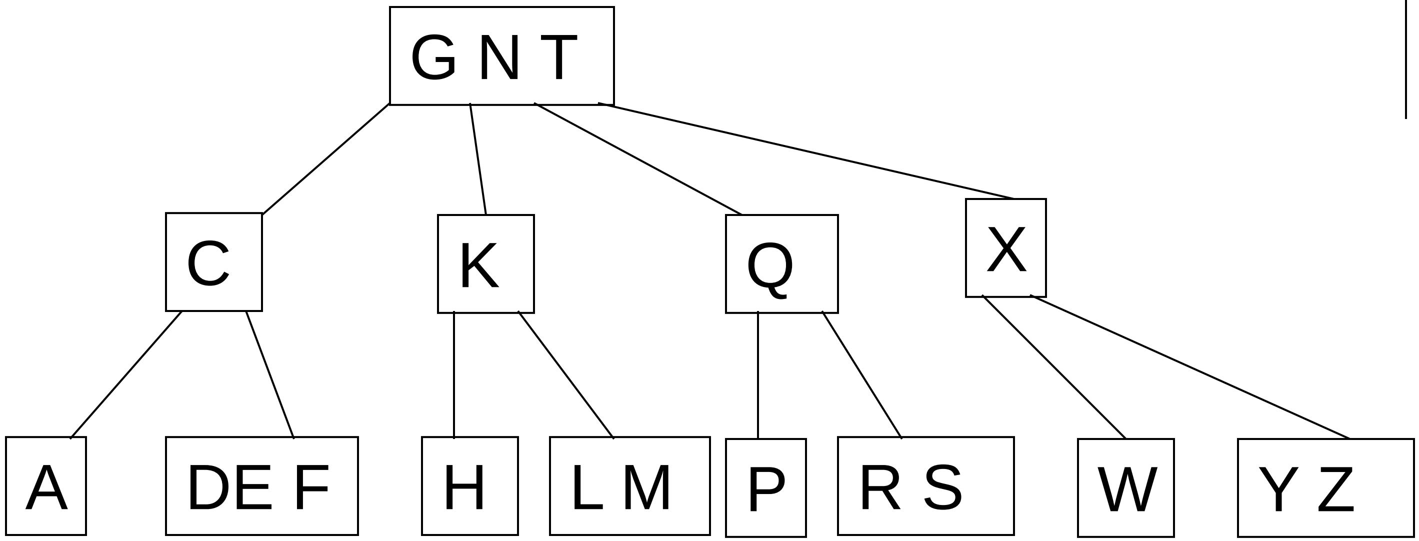
Balanced: all leaves have the same depth

Example B-tree: t = 2



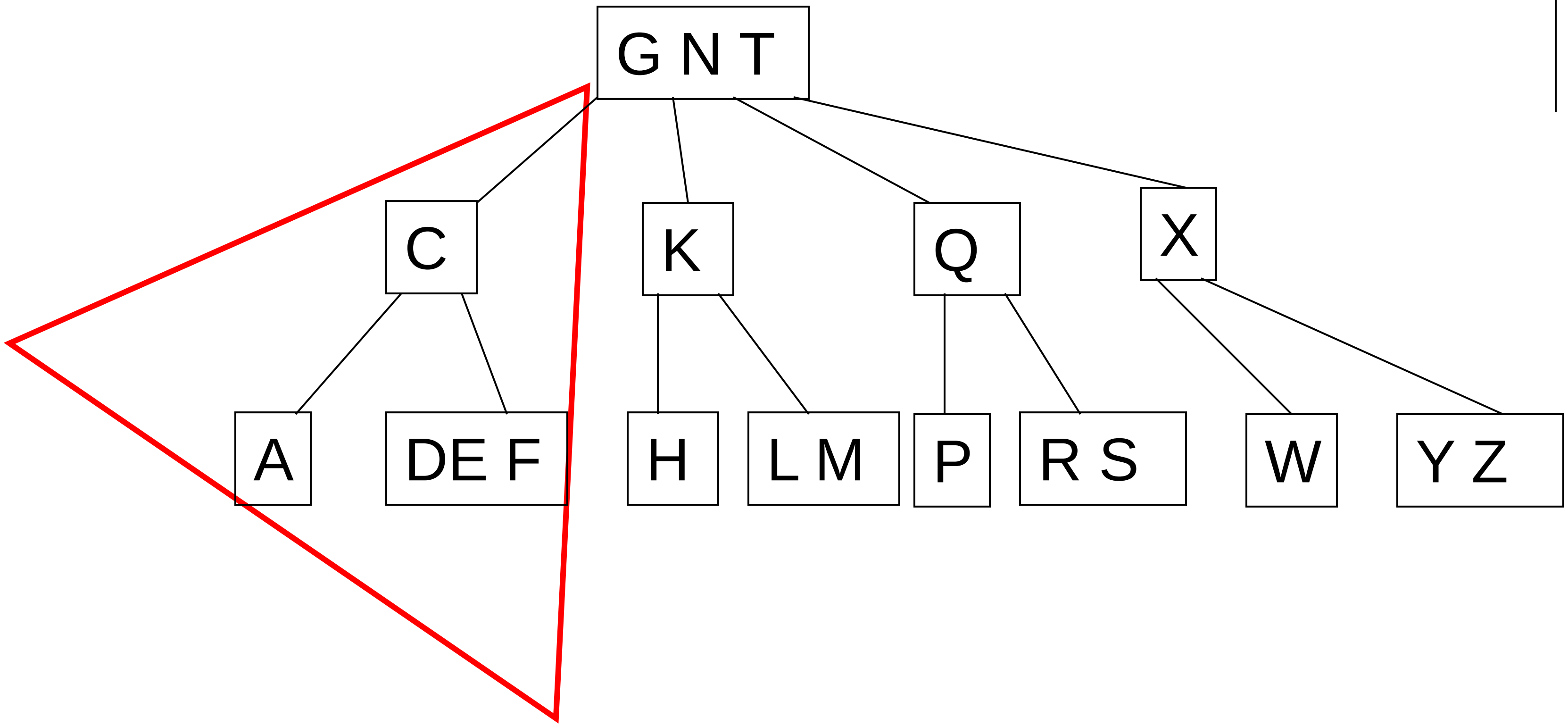
Each node contains between *t-1* and *2t* – 1 keys stored in increasing order

Example B-tree: t = 2



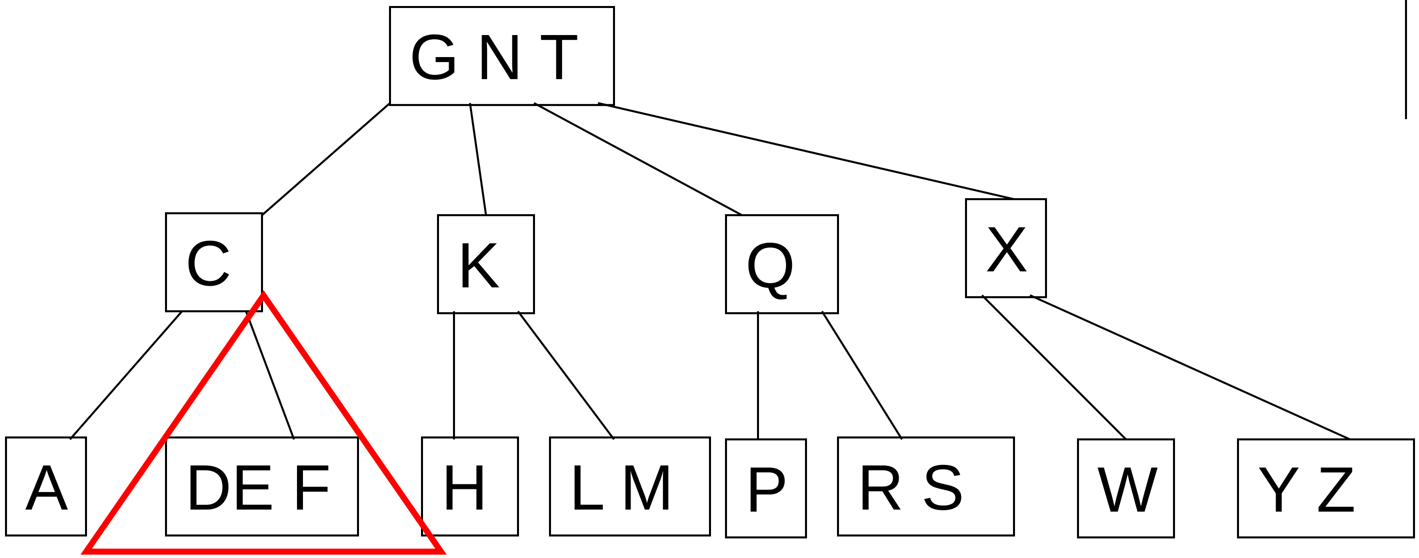
Each node contains between *t* and *2t* children

Example B-tree: t = 2



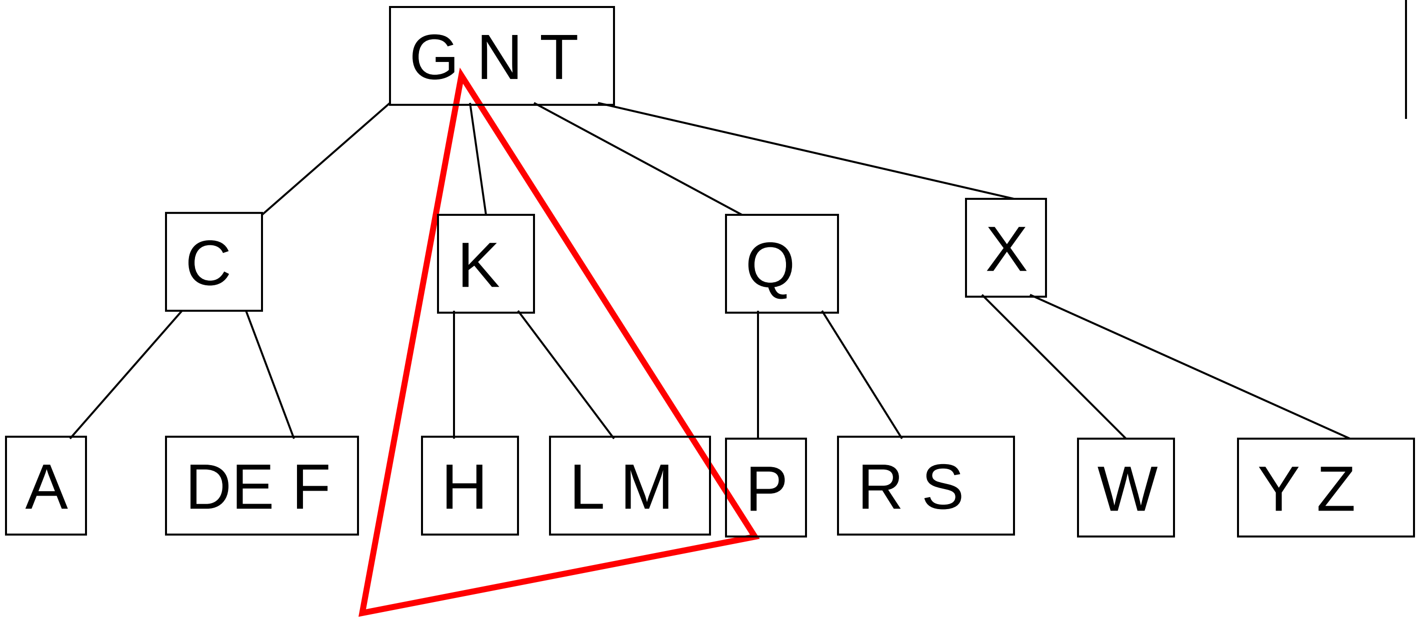
The keys of a parent delimit the values that a child's keys can take

Example B-tree: t = 2



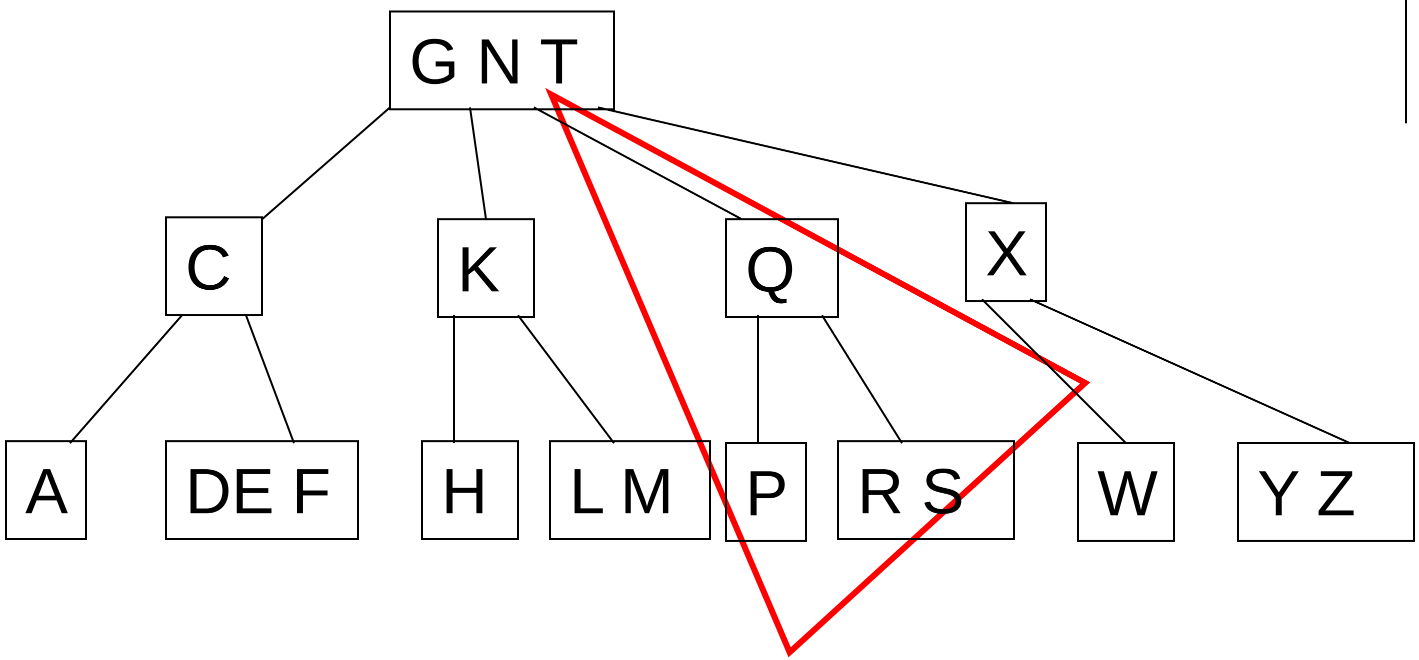
The keys of a parent delimit the values that a child's keys can take

Example B-tree: t = 2



The keys of a parent delimit the values that a child’s keys can take

Example B-tree: t = 2



The keys of a parent delimit the values that a child's keys can take

When do we use B-trees over other balanced trees?

B-trees are generally an **on-disk** data structure

Memory is limited or there is a large amount of data to be stored

In the extreme, only one node is kept in memory and the rest on disk

Size of the nodes is often determined by a page size on disk. Why?

Databases frequently use B-trees

Notes about B-trees

Because *t* is generally large, the height of a B-tree is usually small

t = 1001 with height 2, how many values can we have?

Each internal node contains between *t* and 2*t* children

Each node contains between *t*-1 and 2*t*-1 keys/data values (i.e. multiple data values per tree node)

root level 1 level 2

2001+2002 * 2001 + 2002*2002*2001 = 8,024,024,007

(over 8 billion keys!!!)

Notes about B-trees

Because *t* is generally large, the height of a B-tree is usually small

We will count both run-time as well as the number of disk accesses. Why?

Height of a B-tree

B-trees have a similar feeling to BSTs

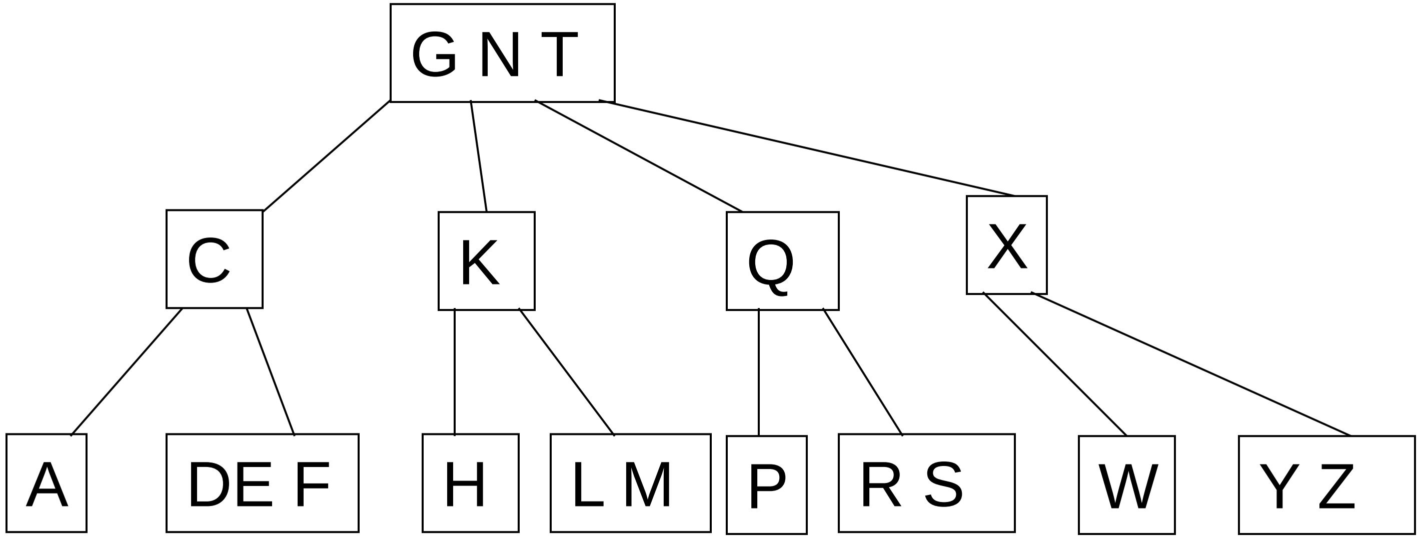
We saw for BSTs that most of the operations depended on the height of the tree

How can we bound the height of the tree?

We know that nodes must have a minimum number of keys/data items (t-1)

For a tree of height *h*, what is the smallest number of keys?

Minimum number of nodes at each depth?



1 root ☺

2 children

2t children

$2t^{h-1}$ children

In general?

Minimum number of keys/values

root

min. keys per node

min. number of nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

Minimum number of nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

$$= 2t^h - 1$$

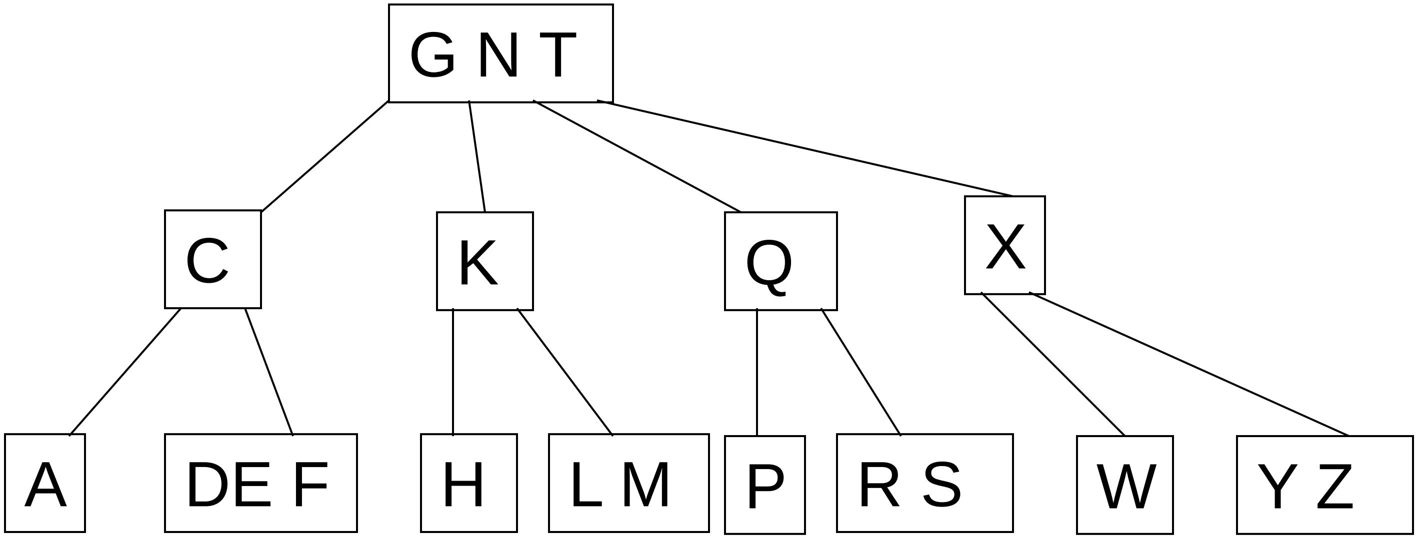
so,

$$t^h \leq (n+1)/2$$

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

Searching B-Trees

Find value *k* in B-Tree



Searching B-Trees

Find value *k* in B-Tree node *x*

number of keys

key[i]

B-TreeSearch(x, k)

$i \leftarrow 1$
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$(C_x[i], k)$

child[i]

Searching B-Trees

```
B-TreeSearch(x, k)
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)
```

make disk reads explicit

Searching B-Trees

```
B-TreeSearch(x, k)
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)
```

iterate through the sorted keys
and find the correct location

Searching B-Trees

$\text{B-TREESEARCH}(x, k)$

$i \leftarrow 1$
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 $\text{LEAF}(x)$
    return $null$
else
    $\text{DISKREAD}(C_x[i])$
    return $\text{B-TREESEARCH}(C_x[i], k)$

if we find the value
in this node, return it

Searching B-Trees

```
B-TreeSearch(x, k)
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)
```

if it’s a leaf and we didn’t find it, it’s not in the tree

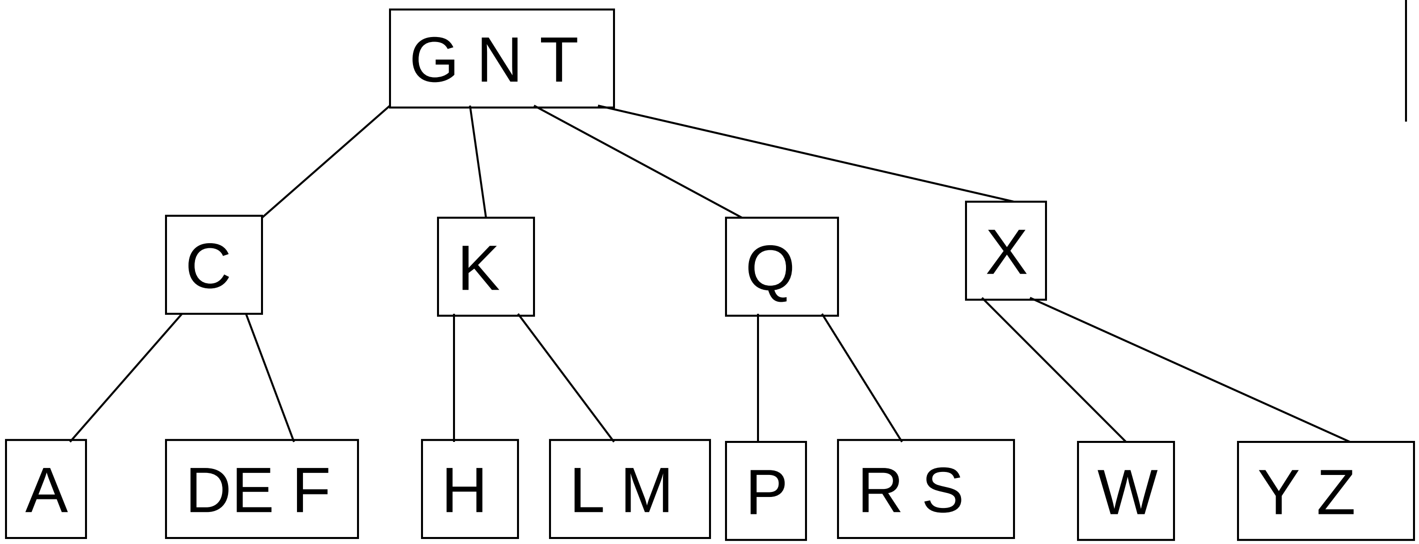
Searching B-Trees

B-TreeSearch(x, k)

$i \leftarrow 1$
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$(C_x[i], k)$

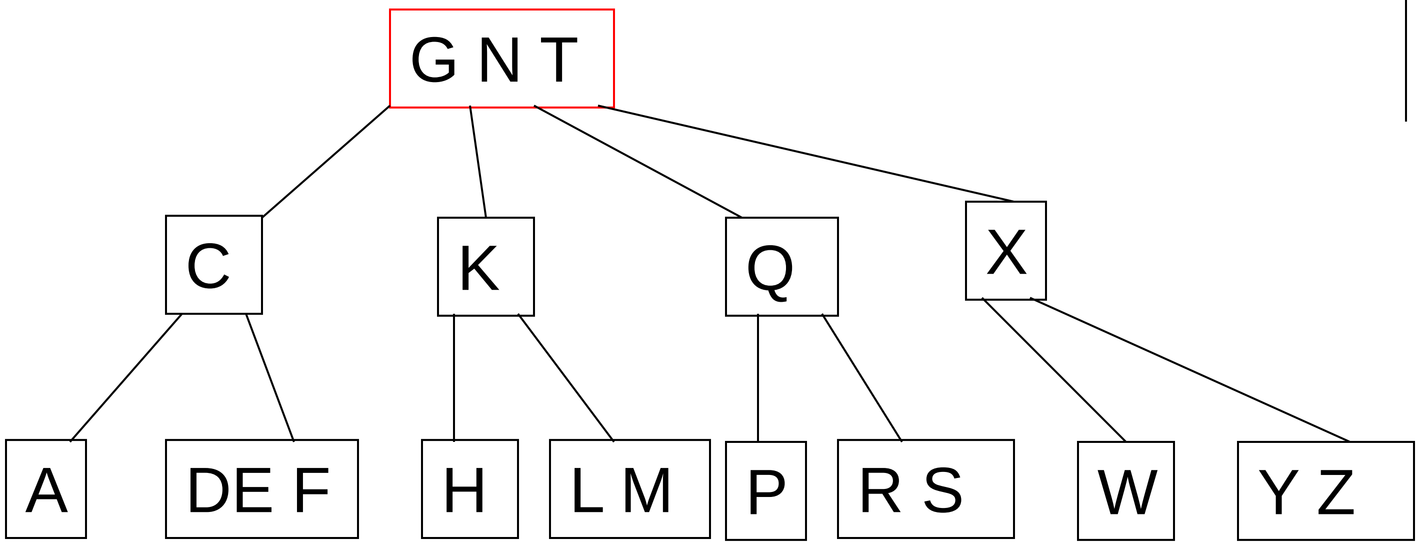
Recurse on the proper child where the value is between the keys

Search example: R




```
B-TreeSearch(x, k)
 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)
```

Search example: R



B-TreeSearch(x, k)

$i \leftarrow 1$

while $i \leq n(x)$ and $k > K_x[i]$

$\quad i \leftarrow i + 1$

if $i \leq n(x)$ and $k = K_x[i]$

$\quad$ **return** (x, i)

if Leaf(x)

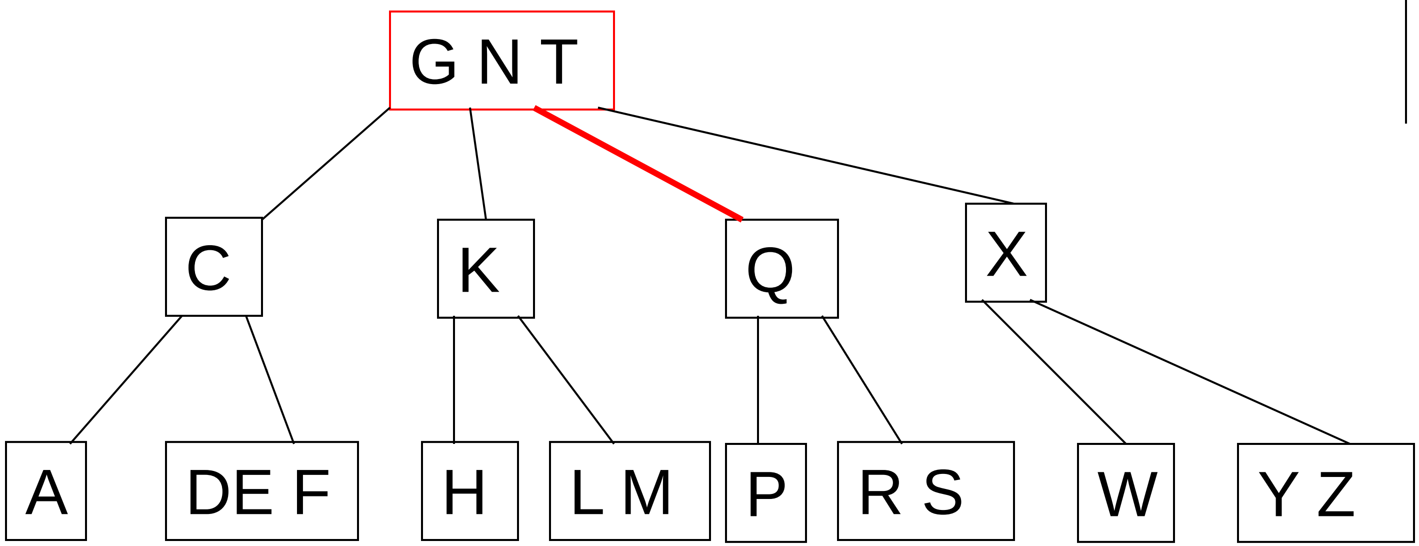
$\quad$ **return** *null*

else

$\quad$ DiskRead$(C_x[i])$

$\quad$ **return** B-TreeSearch$(C_x[i], k)$

Search example: R



B-TreeSearch(x, k)

$i \leftarrow 1$

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*

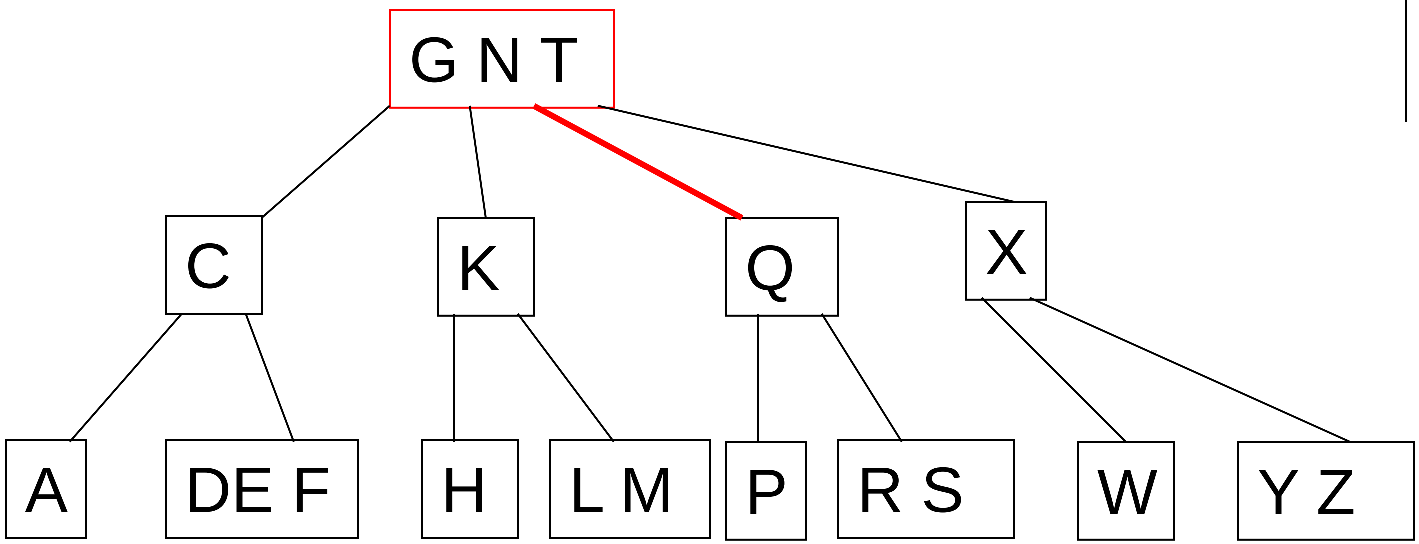
else

    DiskRead$(C_x[i])$

    return B-TreeSearch$(C_x[i], k)$

find the correct location

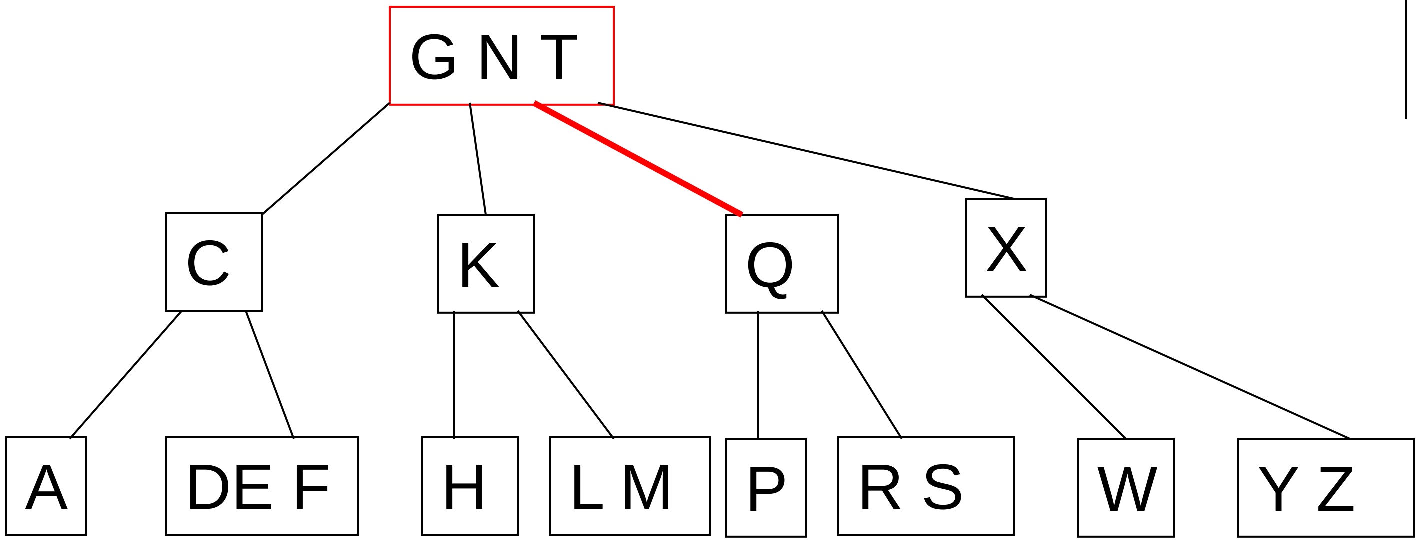
Search example: R




```
B-TreeSearch(x, k)
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)
```

the value is not in this node

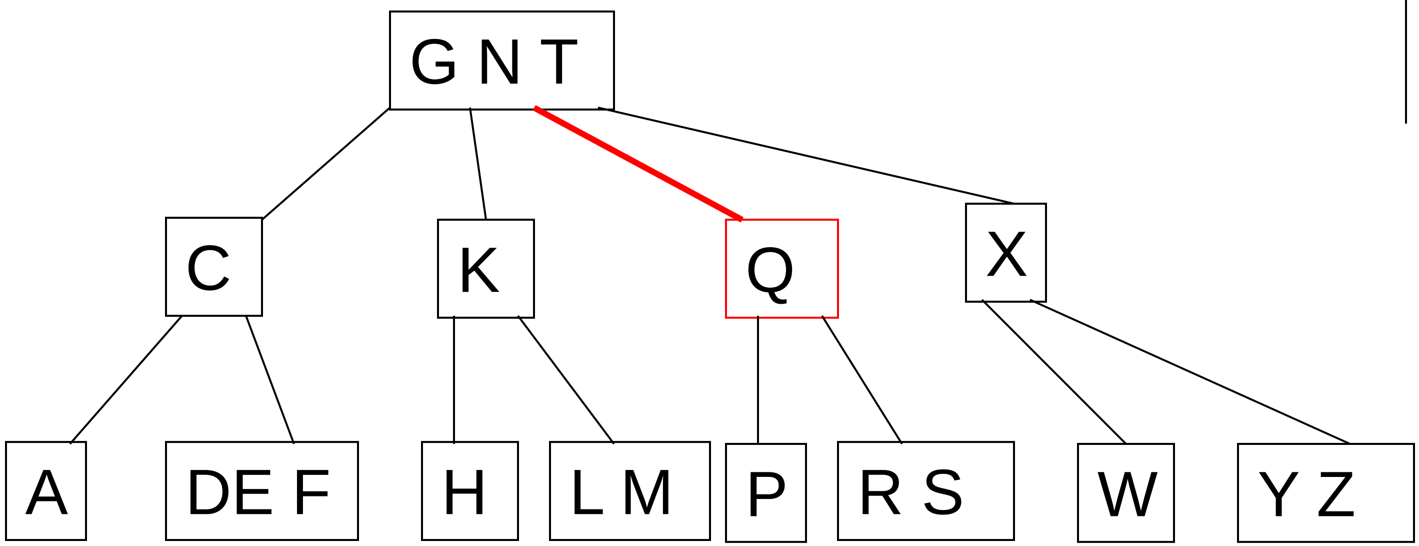
Search example: R




```
B-TREESEARCH(x, k)
  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)
```

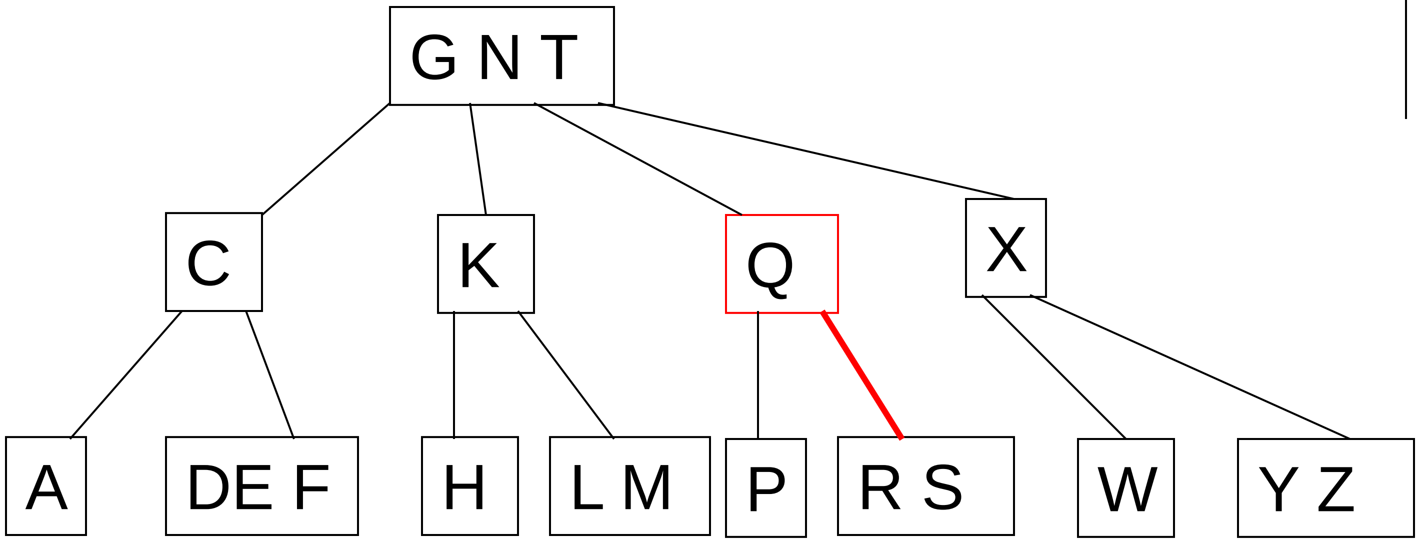
this is not a leaf node

Search example: R




```
B-TreeSearch(x, k)
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)
```

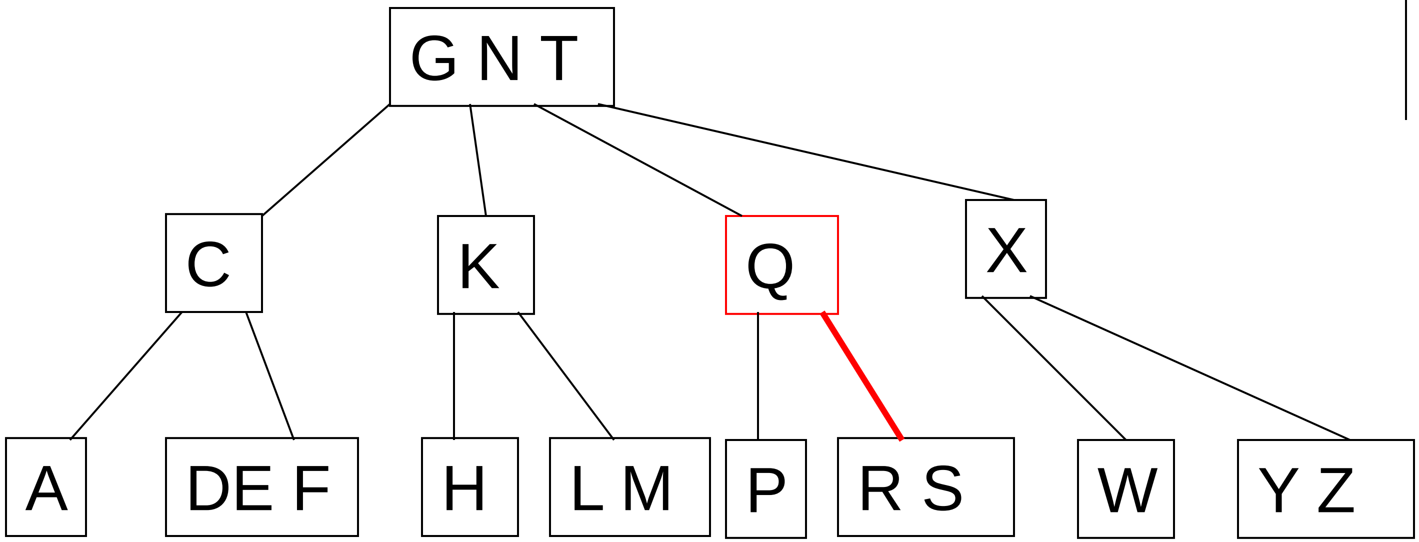
Search example: R




```
B-TreeSearch(x, k)
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

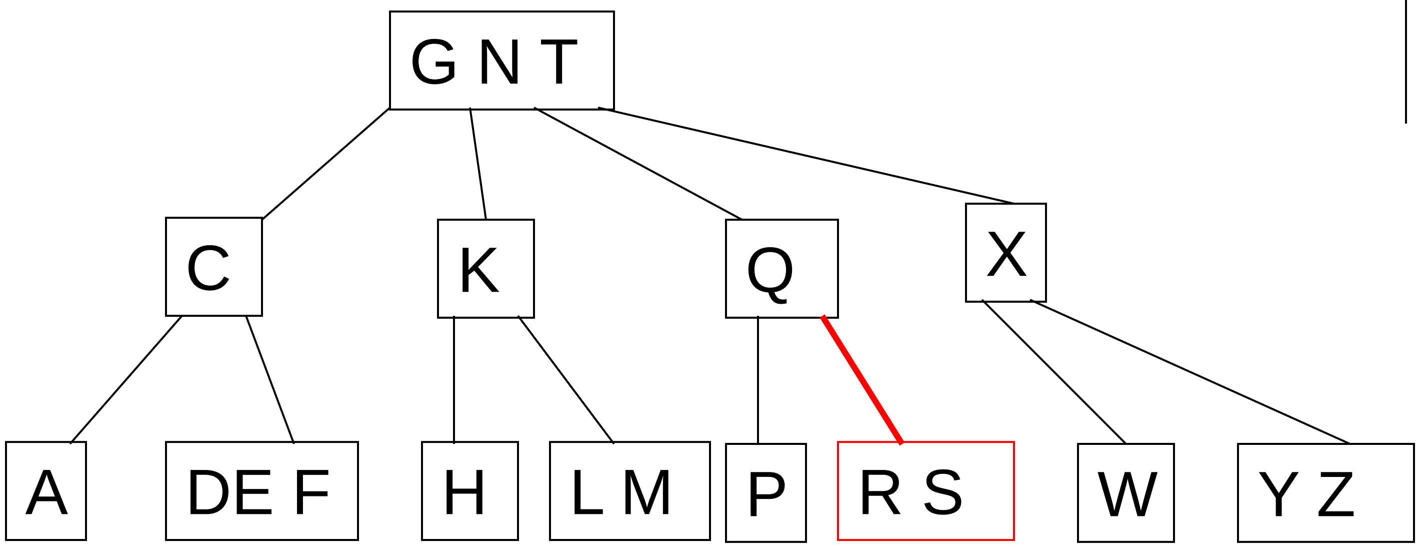
Search example: R




```
B-TreeSearch(x, k)
  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

Search example: R



B-TreeSearch(x, k)

$i \leftarrow 1$

while $i \le n(x)$ and $k > K_x[i]$

    $i \leftarrow i + 1$

if $i \le n(x)$ and $k = K_x[i]$

    return (x, i)

if Leaf(x)

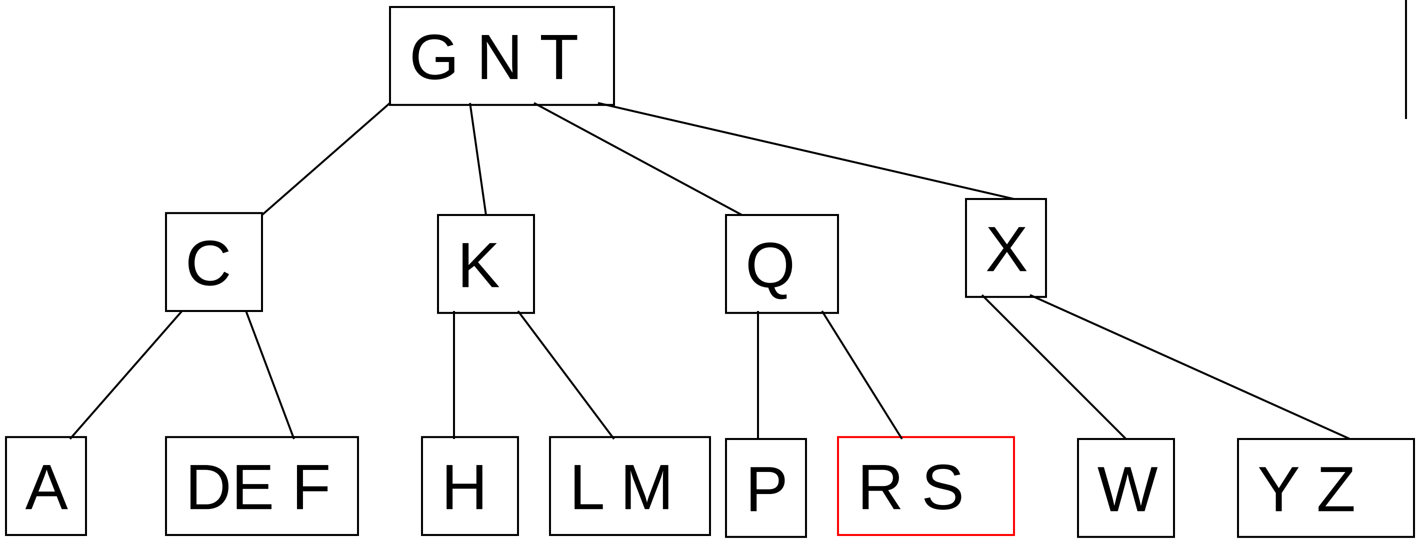
    return *null*

else

    DiskRead$(C_x[i])$

    return B-TreeSearch$(C_x[i], k)$

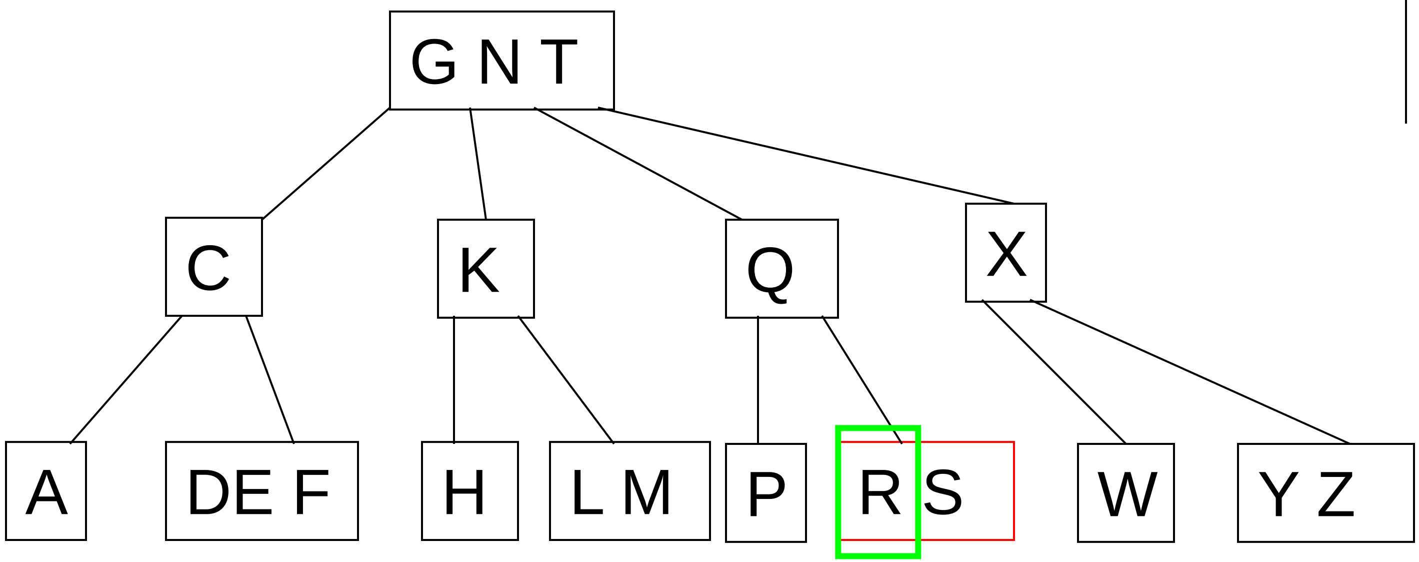
Search example: R




```
B-TreeSearch(x, k)
    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

Search example: R




```
B-TreeSearch(x, k)
  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)
```

Search running time

How many calls to BTreeSearch?

- O(height of the tree)
- $O(\log_t n)$

Disk accesses?

- One for each call – $O(\log_t n)$

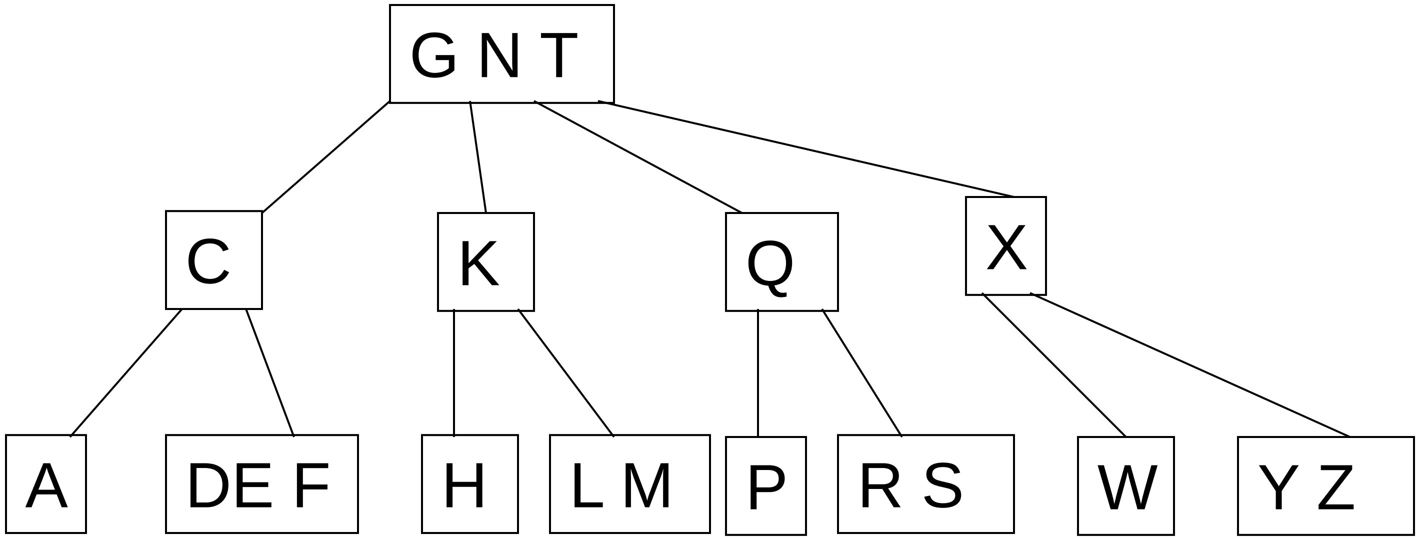
Computational time?

- O(t) keys per node
- linear search
- $O(t \log_t n)$

Why not binary search to find key in a node?

```
B-TREESEARCH(x, k)
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)
```

BST-Insert



B-Tree insert

Starting at root, follow the *search* path down the tree

- If the node is full (contains 2*t* - 1 keys)
  - split the keys into two nodes around the median value
  - add the median value to the parent node
- If the node is a leaf, insert it into the correct spot

Observations

- Insertions **always** happens in the leaves
- When does the height of a B-tree grow?
- Why do we know it's always ok when we're splitting a node to insert the median value into the parent?

Insertion: t = 2

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

Insertion: t = 2

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

$\boxed{G}$

Insertion: t = 2

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

C G

Insertion: t = 2

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

$\boxed{\text{C G N}}$

Insertion: t = 2

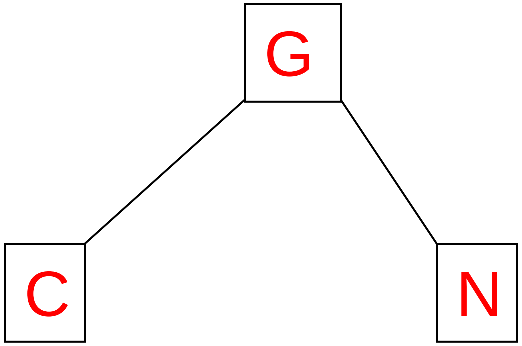
G C N A H E K Q M F W L T Z D P R X Y S

C G N

Node is full, so split

Insertion: t = 2

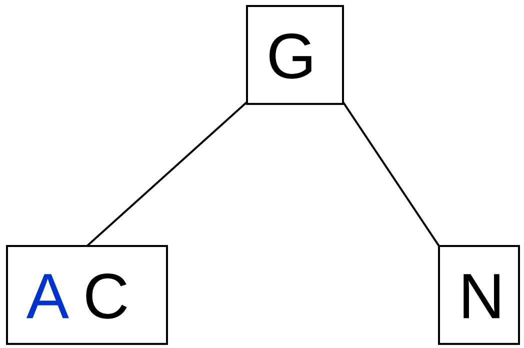
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

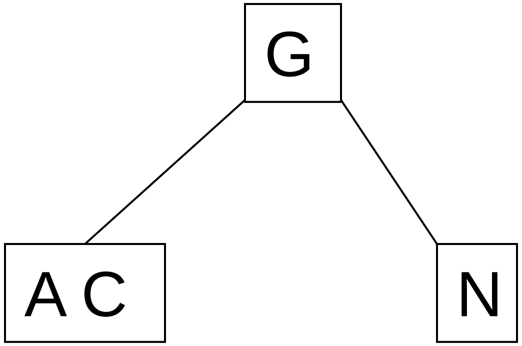
Insertion: t = 2

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



Insertion: t = 2

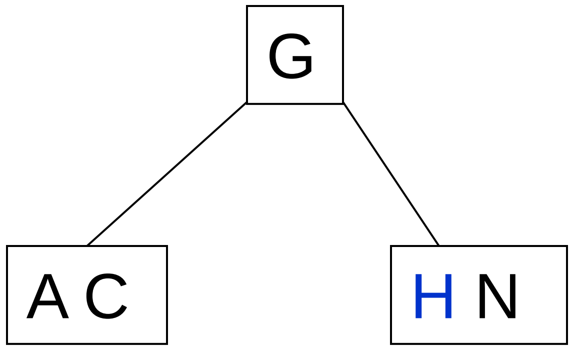
G C N A H E K Q M F W L T Z D P R X Y S



?

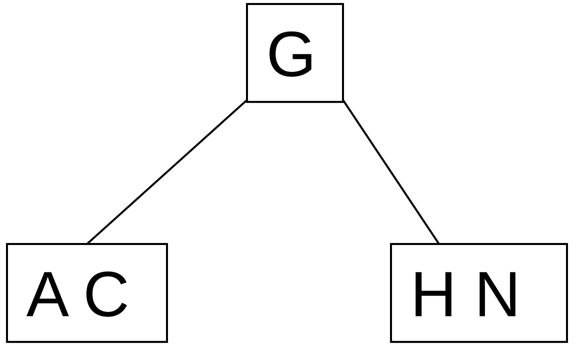
Insertion: t = 2

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



Insertion: t = 2

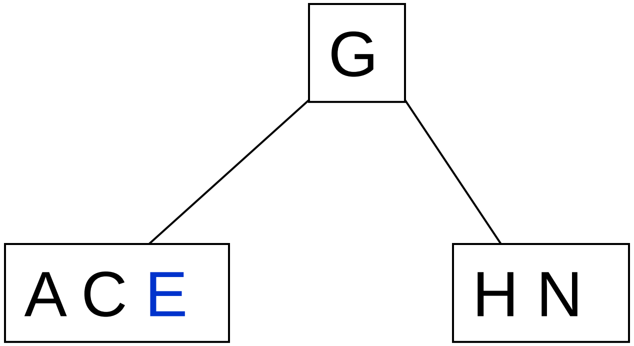
G C N A H E K Q M F W L T Z D P R X Y S



?

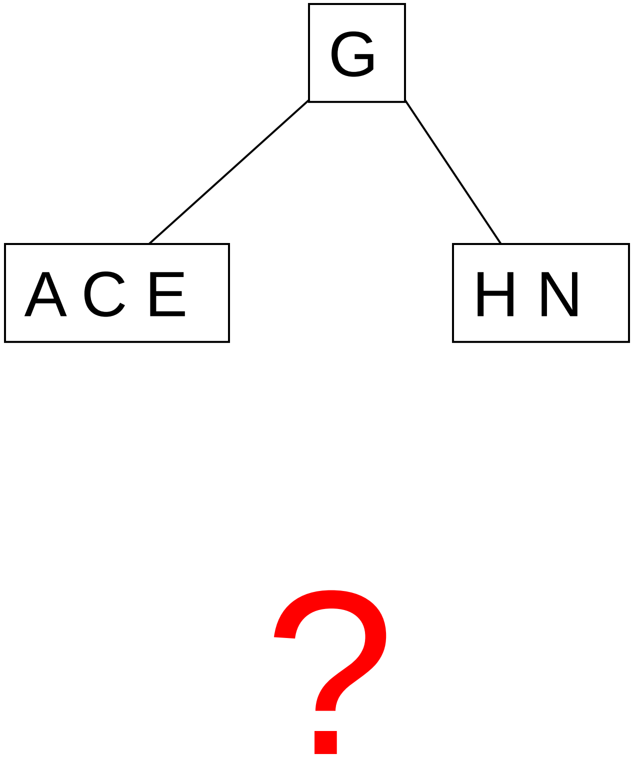
Insertion: t = 2

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



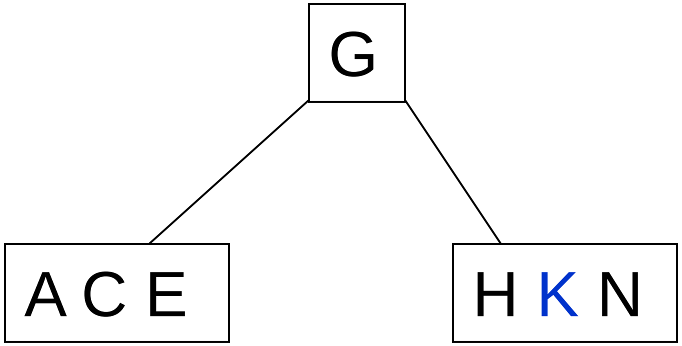
Insertion: t = 2

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



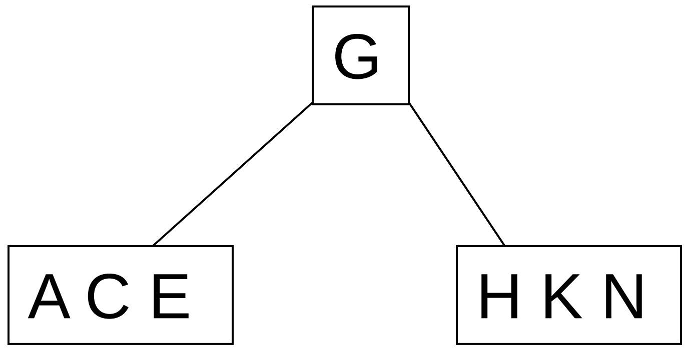
Insertion: t = 2

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



Insertion: t = 2

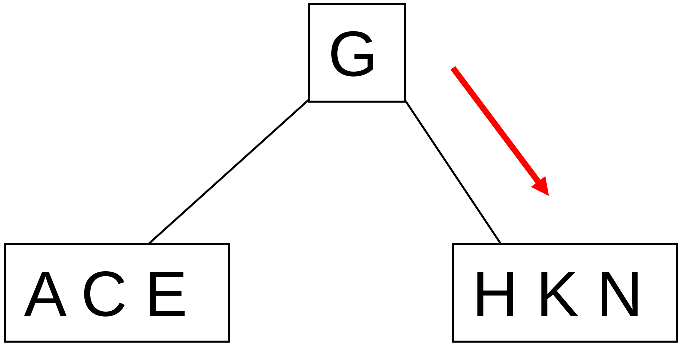
G C N A H E K Q M F W L T Z D P R X Y S



?

Insertion: t = 2

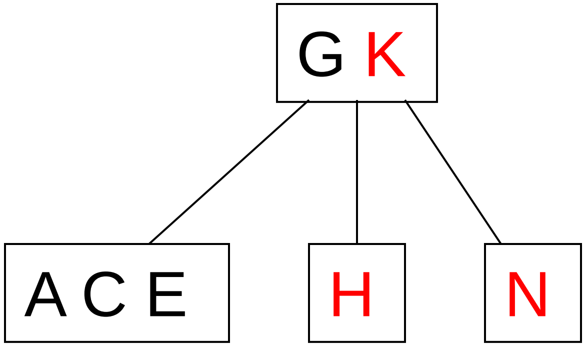
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

Insertion: t = 2

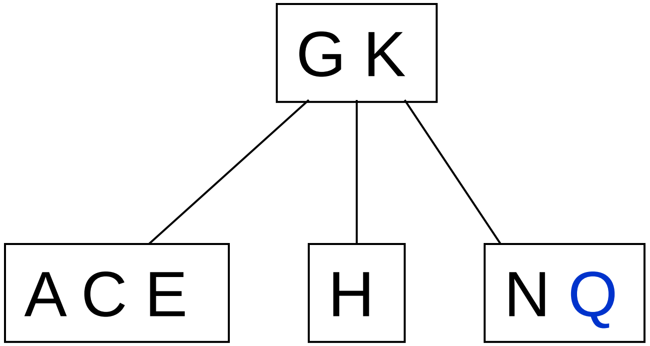
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

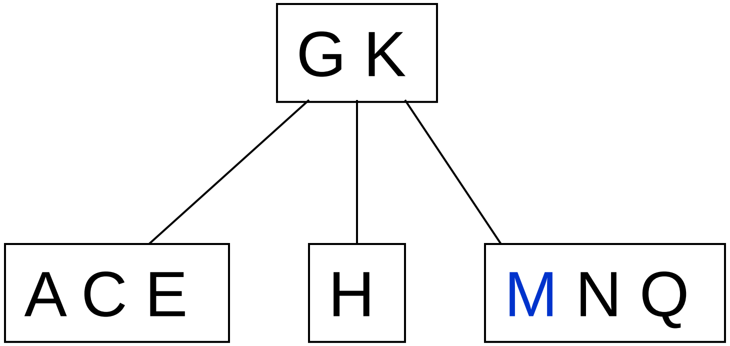
Insertion: t = 2

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



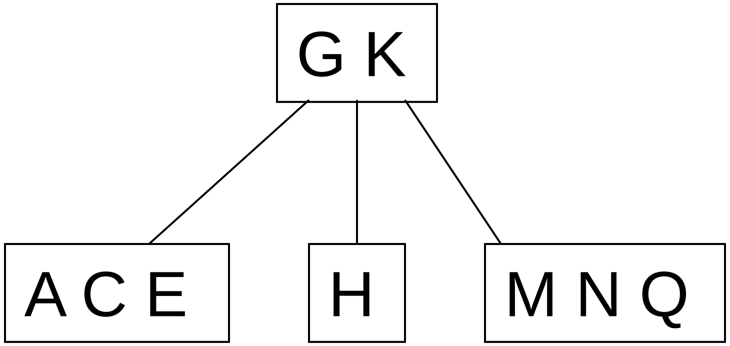
Insertion: t = 2

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



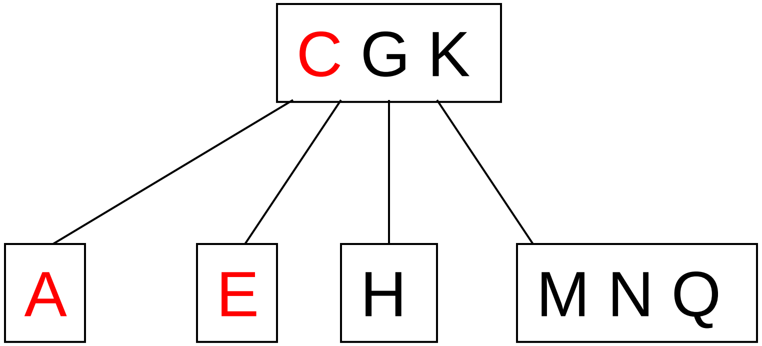
Insertion: t = 2

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



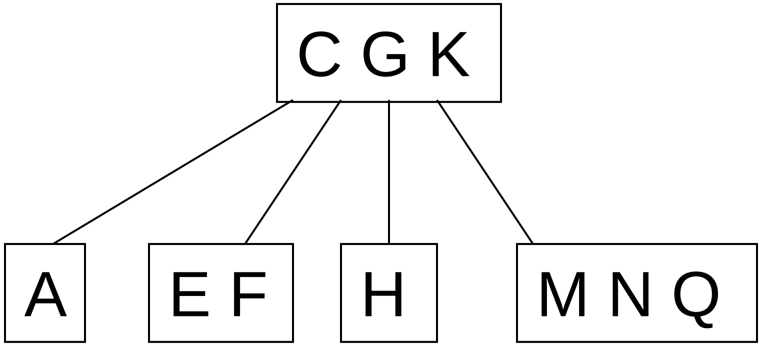
Insertion: t = 2

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



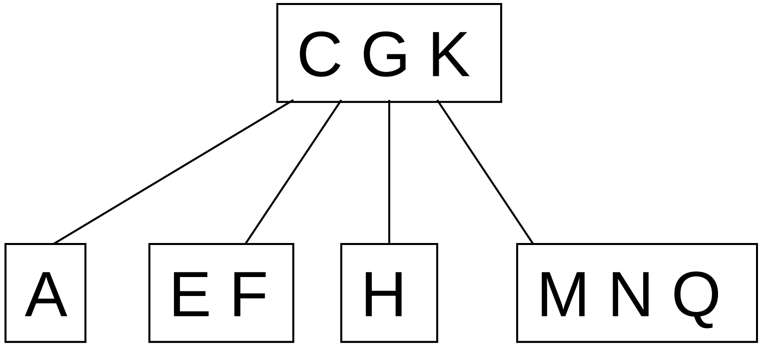
Insertion: t = 2

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



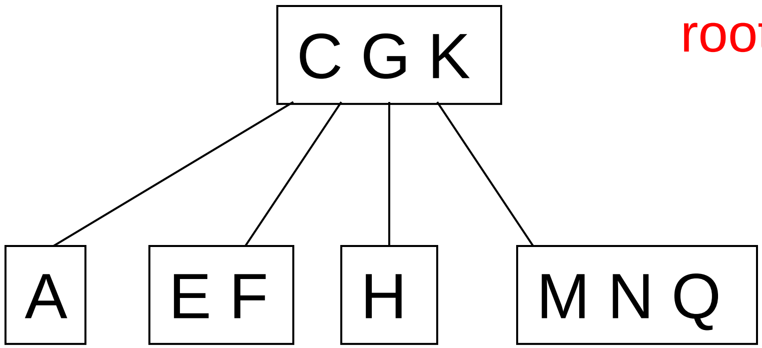
Insertion: t = 2

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



Insertion: t = 2

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

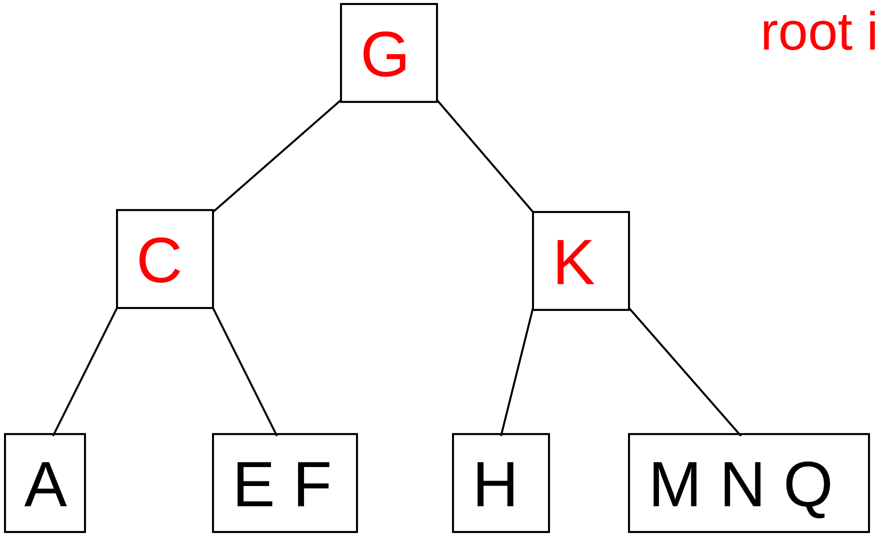


root is full, so split

?

Insertion: t = 2

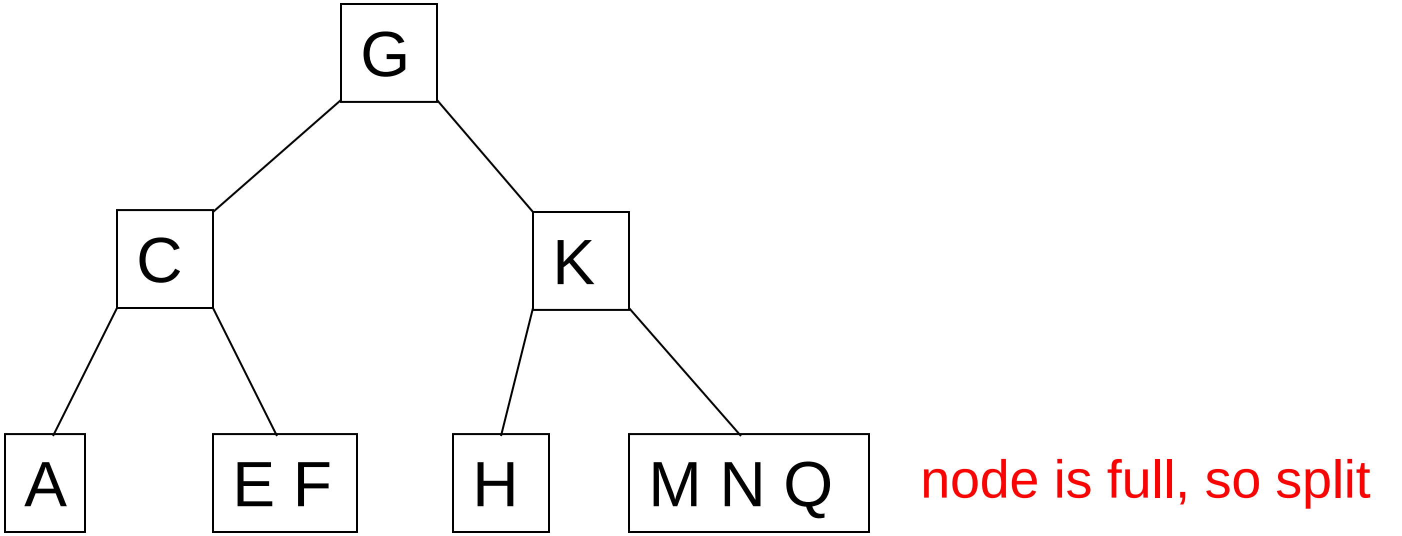
G C N A H E K Q M F W L T Z D P R X Y S



root is full, so split

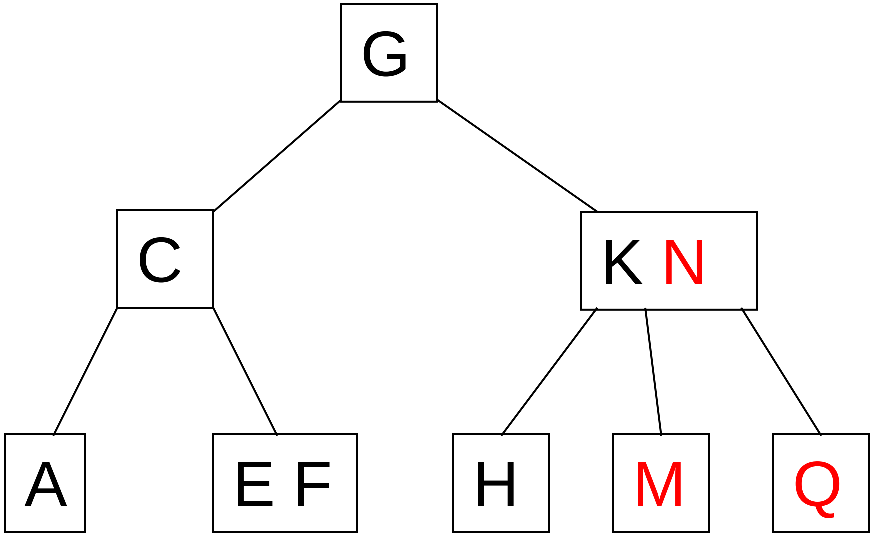
Insertion: t = 2

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



Insertion: t = 2

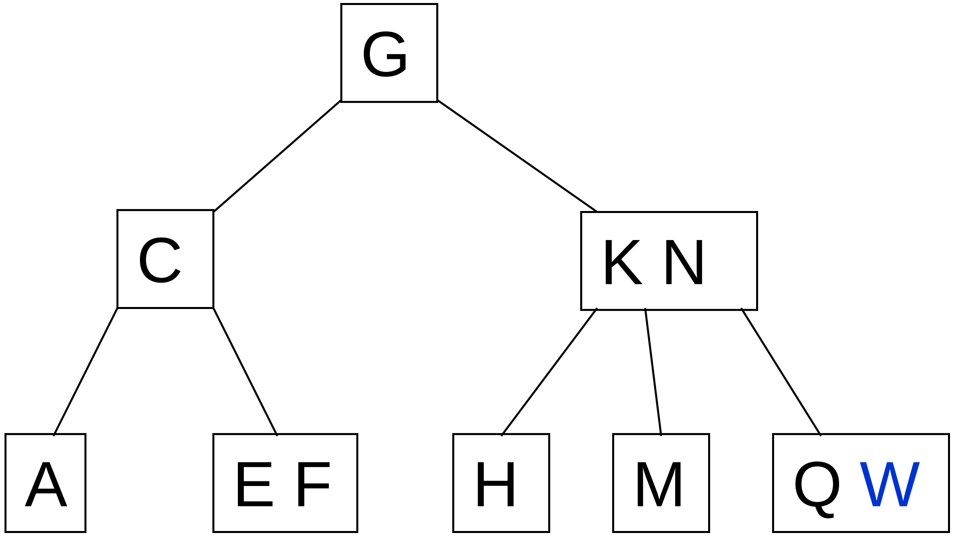
G C N A H E K Q M F W L T Z D P R X Y S



node is full, so split

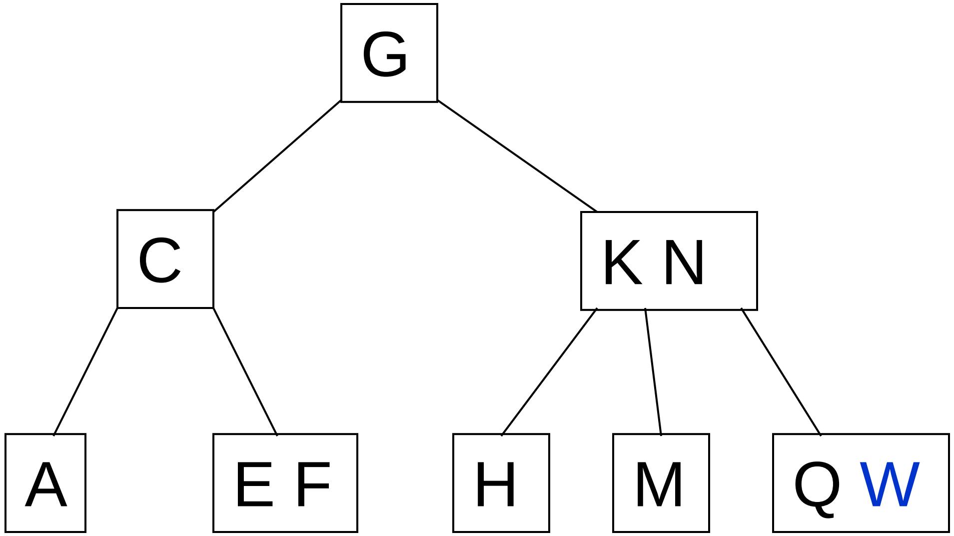
Insertion: t = 2

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



Insertion: t = 2

G C N A H E K Q M F W ...



Correctness of insert

Starting at root, follow *search* path down the tree

- If the node is full (contains $2t - 1$ keys), split the keys around the median value into two nodes and add the median value to the parent node
- If the node is a leaf, insert it into the correct spot

Does it add the value in the correct spot?

- Follows the correct *search* path
- Inserts in correct position

Correctness of insert

Starting at root, follow *search* path down the tree

- If the node is full (contains $2t$ - 1 keys), split the keys around the median value into two nodes and add the median value to the parent node
- If the node is a leaf, insert it into the correct spot

Do we maintain a proper B-tree?

- Maintain t-1 to 2t-1 keys per node?
  - Always split full nodes when we see them
  - Only split full nodes
- All leaves at the same level?
  - Only add nodes at leaves

Insert running time

Without any splitting?

- Similar to BTreeSearch, with one extra disk write at the leaf
- $O(\log_t n)$ disk accesses
- $O(t \log_t n)$ computation time

When a node is split

How many disk accesses?

- 3 disk write operations
  - 2 for the new nodes created by the split (one is reused, but must be updated)
  - 1 for the parent node to add median value

Runtime to split a node?

- O(t) – iterating through the elements a few times since they’re already in sorted order

Maximum number of nodes split for a call to insert?

- O(height of the tree)

Running time of insert

$O(\log_t n)$ disk accesses

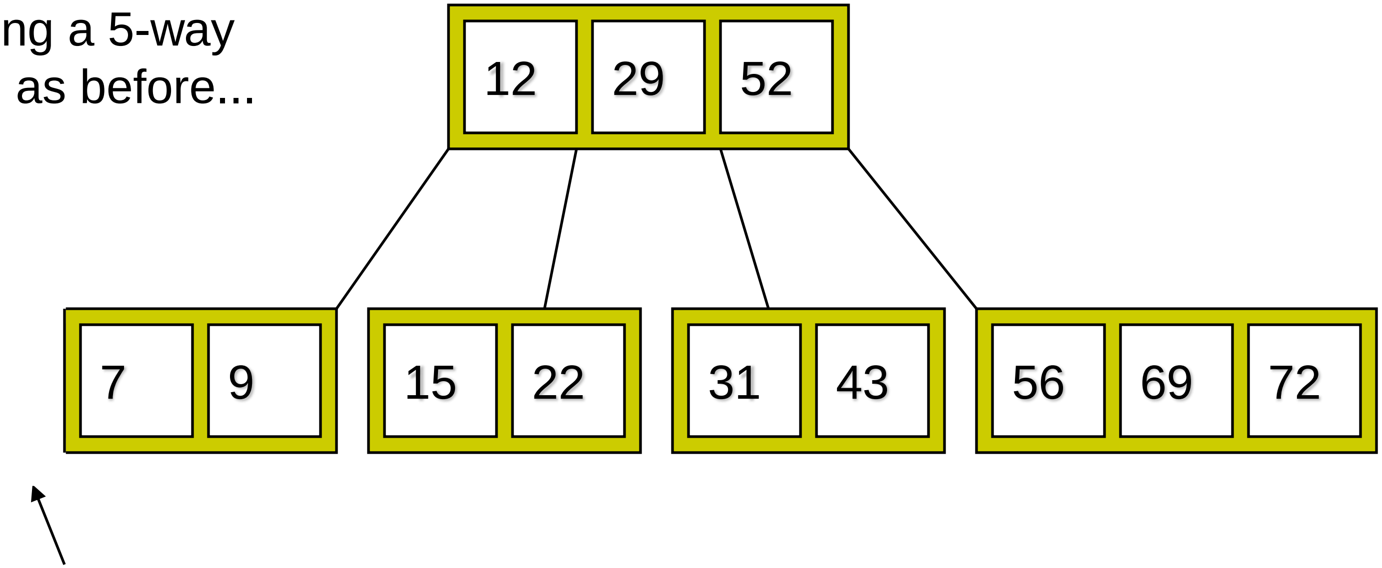
$O(t \log_t n)$ computational costs

Removal from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn’t cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case can we delete the key and promote the predecessor or successor key to the non-leaf deleted key’s position.

Type #1: Simple leaf deletion

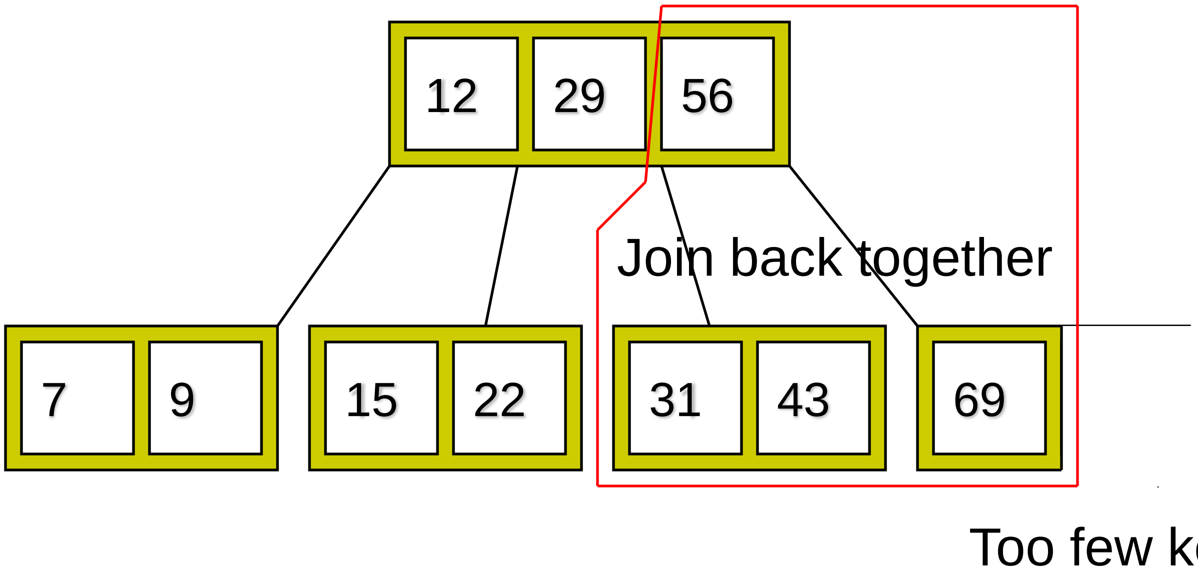
Assuming a 5-way B-Tree, as before...



Delete 2: Since there are enough keys in the node, just delete it

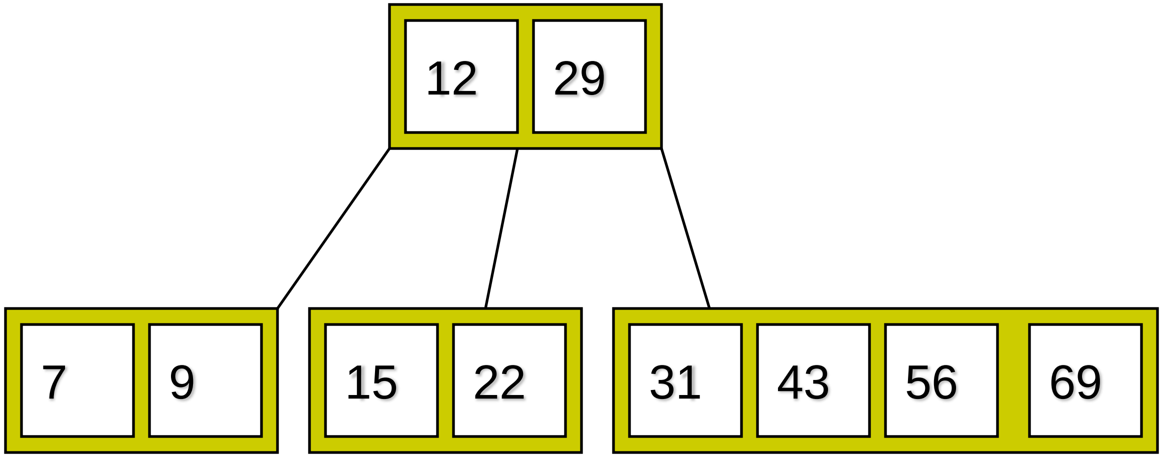
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



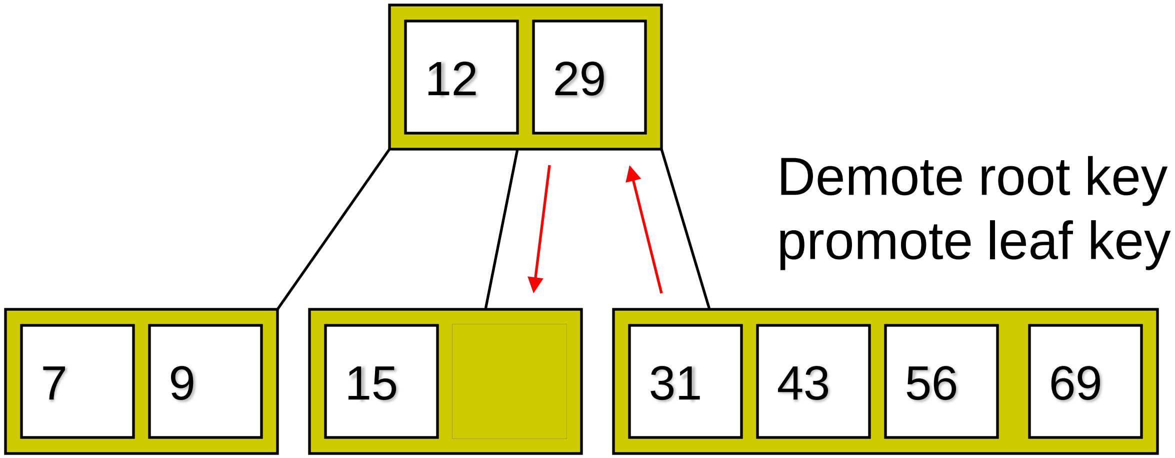
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



Note when printed: this slide is animated

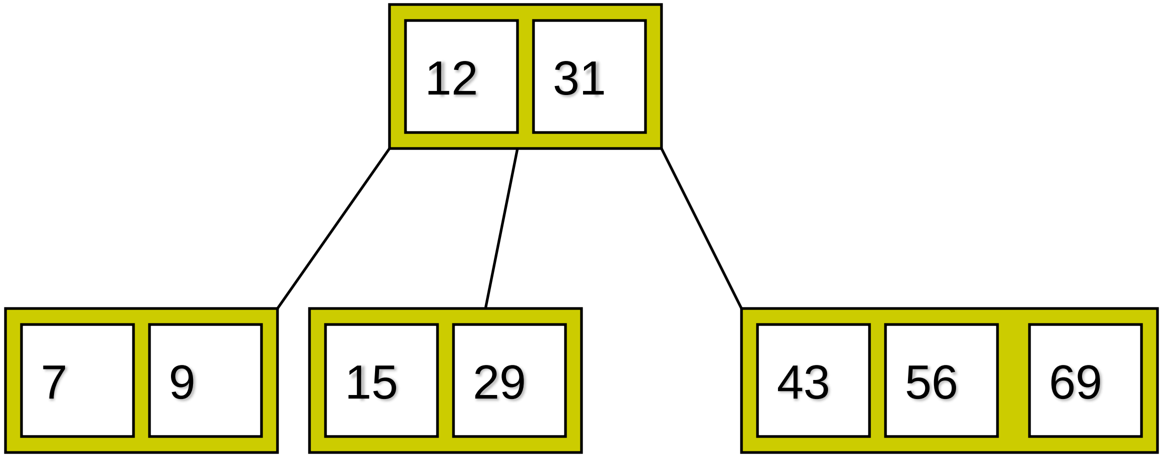
Type #3: Enough siblings



Demote root key and promote leaf key

Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Exercise in Removal from a B-Tree

- Create a B-tree t=3 or 5-way using these:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6,12
- Delete these keys: 4, 5, 7, 3, 14

Summary of operations

Search, Insertion, Deletion

- disk accesses: $O(\log_t n)$
- computation: $O(t \log_t n)$

Max, Min

- disk accesses: $O(\log_t n)$
- computation: $O(\log_t n)$