

# B-Trees

Antonio Carzaniga

Faculty of Informatics  
Università della Svizzera italiana

April 30, 2024

- Search in secondary storage

- B-Trees

- ▶ properties
- ▶ search
- ▶ insertion



- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - ▶ all basic operations have the same cost
    - ▶ even this is a rough approximation, since the main-memory system is not at all “flat”

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - ▶ all basic operations have the same cost
    - ▶ even this is a rough approximation, since the main-memory system is not at all “flat”
- However, some applications require more storage than what fits in main memory
  - ▶ we must use data structures that reside in *secondary storage* (i.e., disk)

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - ▶ all basic operations have the same cost
    - ▶ even this is a rough approximation, since the main-memory system is not at all “flat”
- However, some applications require more storage than what fits in main memory
  - ▶ we must use data structures that reside in *secondary storage* (i.e., disk)

***Disk is 10,000–100,000 times slower than RAM***



**Memory access/transfer**

**CPU cycles ( $\approx 1\text{ns}$ )**

---

---

*Register*

---

***1***

<b>Memory access/transfer</b>	<b>CPU cycles (<math>\approx 1\text{ns}</math>)</b>
<b><i>Register</i></b>	<b><i>1</i></b>
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
<b><i>Remote DRAM (main memory)</i></b>	<b><i>100</i></b>

<b>Memory access/transfer</b>	<b>CPU cycles (<math>\approx 1\text{ns}</math>)</b>
<b><i>Register</i></b>	<b><i>1</i></b>
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
<b><i>Remote DRAM (main memory)</i></b>	<b><i>100</i></b>
<b><i>SSD seek</i></b>	<b><i>20,000</i></b>

<b>Memory access/transfer</b>	<b>CPU cycles (<math>\approx 1\text{ns}</math>)</b>
<b><i>Register</i></b>	<b><i>1</i></b>
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
<b><i>Remote DRAM (main memory)</i></b>	<b><i>100</i></b>
<b><i>SSD seek</i></b>	<b><i>20,000</i></b>
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within a datacenter	500,000

<b>Memory access/transfer</b>	<b>CPU cycles (<math>\approx 1\text{ns}</math>)</b>
<b><i>Register</i></b>	<b><i>1</i></b>
L1 cache	4
L2 cache	10
Local L3 cache	40–75
Remote L3 cache	100–300
Local DRAM	60
<b><i>Remote DRAM (main memory)</i></b>	<b><i>100</i></b>
<b><i>SSD seek</i></b>	<b><i>20,000</i></b>
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within a datacenter	500,000
<b><i>HDD seek</i></b>	<b><i>10,000,000</i></b>
Read 1 MB sequentially from network	10,000,000
Read 1 MB sequentially from disk	30,000,000
Round-trip time USA–Europe	150,000,000

# Modeling Disk Access

- Let  $x$  be a pointer to some (possibly complex) object

- Let  $x$  be a pointer to some (possibly complex) object
- When the object is in memory,  $x$  can be used directly as a reference to the object
  - ▶ e.g.,  $\ell = x.size$  or  $x.root = y$

- Let  $x$  be a pointer to some (possibly complex) object
- When the object is in memory,  $x$  can be used directly as a reference to the object
  - ▶ e.g.,  $\ell = x.size$  or  $x.root = y$
- When the object is on disk, we must first perform a disk-read operation  
**DISK-READ**( $x$ ) reads the object into memory, allowing us to refer to it (and modify it) through  $x$

- Let  $x$  be a pointer to some (possibly complex) object
- When the object is in memory,  $x$  can be used directly as a reference to the object
  - ▶ e.g.,  $\ell = x.size$  or  $x.root = y$
- When the object is on disk, we must first perform a disk-read operation  
**DISK-READ**( $x$ ) reads the object into memory, allowing us to refer to it (and modify it) through  $x$
- Any changes to the object in memory must be eventually saved onto the disk  
**DISK-WRITE**( $x$ ) writes the object onto the disk (if the object was modified)

- Assume each node  $x$  is stored on disk

- Assume each node  $x$  is stored on disk

## **ITERATIVE-TREE-SEARCH**( $T, k$ )

```
1  $x = T.root$ 
2 while  $x \neq NIL$ 
3     DISK-READ( $x$ )
4     if  $k == x.key$ 
5         return  $x$ 
6     elseif  $k < x.key$ 
7          $x = x.left$ 
8     else  $x = x.right$ 
9 return  $x$ 
```

- Assume each node  $x$  is stored on disk

## **ITERATIVE-TREE-SEARCH**( $T, k$ )

```
1  $x = T.root$ 
2 while  $x \neq NIL$ 
3     DISK-READ( $x$ )
4     if  $k == x.key$ 
5         return  $x$ 
6     elseif  $k < x.key$ 
7          $x = x.left$ 
8     else  $x = x.right$ 
9 return  $x$ 
```

cost

- Assume each node  $x$  is stored on disk

	<i>cost</i>
<b>ITERATIVE-TREE-SEARCH</b> ( $T, k$ )	
1 $x = T.root$	$c$
2 <b>while</b> $x \neq NIL$	$c$
3 <b>DISK-READ</b> ( $x$ )	$100000c$
4 <b>if</b> $k == x.key$	$c$
5 <b>return</b> $x$	$c$
6 <b>elseif</b> $k < x.key$	$c$
7 $x = x.left$	$c$
8 <b>else</b> $x = x.right$	$c$
9 <b>return</b> $x$	$c$



- Assume we store the nodes of a search tree on disk
  1. node accesses should be reduced to a minimum
  2. spending more than a few basic operations for each node is not a problem

- Assume we store the nodes of a search tree on disk
  1. node accesses should be reduced to a minimum
  2. spending more than a few basic operations for each node is not a problem
- Rationale
  - ▶ basic in-memory operations are much cheaper
  - ▶ the bottleneck is with node accesses, which involve **DISK-READ** and **DISK-WRITE** operations

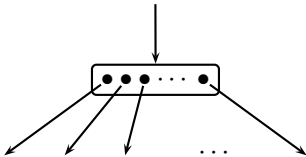


- In a balanced *binary* tree,  $n$  keys require a tree of height  $h = \lfloor \log_2 n \rfloor$ 
  - ▶ all the important operations require access to  $O(h)$  nodes
  - ▶ each one accounting for *one or very few* basic operations

- In a balanced *binary tree*,  $n$  keys require a tree of height  $h = \lfloor \log_2 n \rfloor$ 
  - ▶ all the important operations require access to  $O(h)$  nodes
  - ▶ each one accounting for *one or very few* basic operations
  
- **Idea:** store several keys and pointers to children nodes in a single node

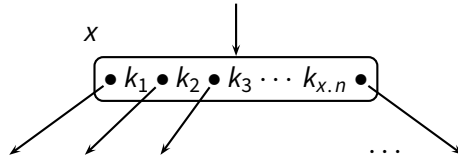
- In a balanced *binary* tree,  $n$  keys require a tree of height  $h = \lfloor \log_2 n \rfloor$ 
  - ▶ all the important operations require access to  $O(h)$  nodes
  - ▶ each one accounting for *one or very few* basic operations
  
- **Idea:** store several keys and pointers to children nodes in a single node
  - ▶ in practice we **increase the degree** (or *branching factor*) of each node up to  $d > 2$ , so  $h = \lfloor \log_d n \rfloor$ 
    - ▶ in practice  $d$  can be as high as a few thousands

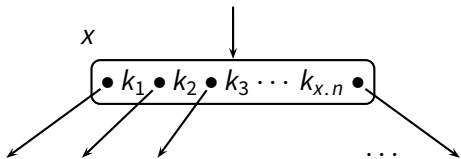
- In a balanced *binary* tree,  $n$  keys require a tree of height  $h = \lfloor \log_2 n \rfloor$ 
  - ▶ all the important operations require access to  $O(h)$  nodes
  - ▶ each one accounting for *one or very few* basic operations
- **Idea:** store several keys and pointers to children nodes in a single node
  - ▶ in practice we **increase the degree** (or *branching factor*) of each node up to  $d > 2$ , so  $h = \lfloor \log_d n \rfloor$ 
    - ▶ in practice  $d$  can be as high as a few thousands



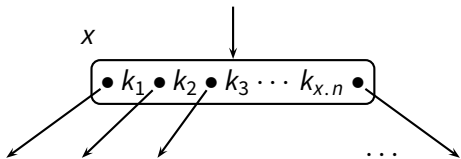
E.g., if  $d = 1000$ , then  
**only three accesses** ( $h = 2$ )  
cover **up to one billion keys**

# Definition of a B-Tree



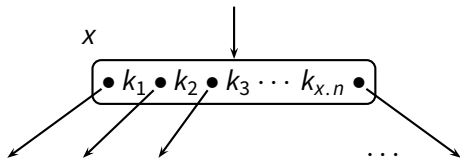


- Every node  $x$  has the following fields
  - ▶  $x.n$  is the number of keys stored at each node



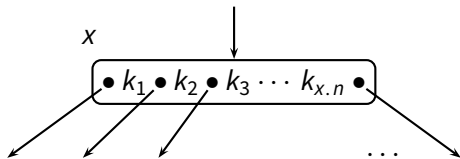
■ Every node  $x$  has the following fields

- ▶  $x.n$  is the number of keys stored at each node
- ▶  $x.key[1] \leq x.key[2] \leq \dots x.key[x.n]$  are the  $x.n$  keys stored in nondecreasing order



■ Every node  $x$  has the following fields

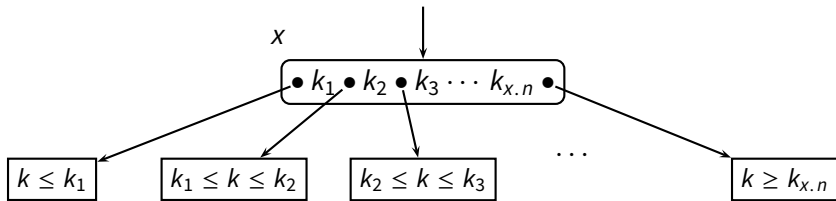
- ▶  $x.n$  is the number of keys stored at each node
- ▶  $x.key[1] \leq x.key[2] \leq \dots x.key[x.n]$  are the  $x.n$  keys stored in nondecreasing order
- ▶  $x.leaf$  is a Boolean flag that is TRUE if  $x$  is a *leaf node* or FALSE if  $x$  is an *internal node*



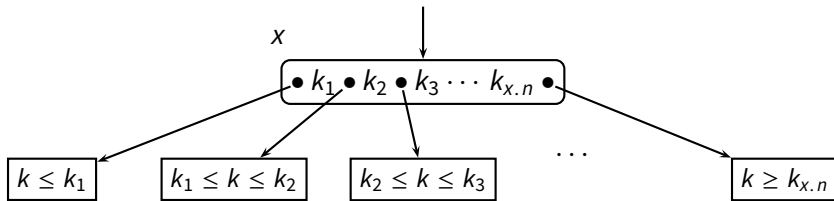
■ Every node  $x$  has the following fields

- ▶  $x.n$  is the number of keys stored at each node
- ▶  $x.key[1] \leq x.key[2] \leq \dots x.key[x.n]$  are the  $x.n$  keys stored in nondecreasing order
- ▶  $x.leaf$  is a Boolean flag that is TRUE if  $x$  is a *leaf node* or FALSE if  $x$  is an *internal node*
- ▶  $x.c[1], x.c[2], \dots, x.c[x.n + 1]$  are the  $x.n + 1$  pointers to its children, if  $x$  is an *internal node*

## Definition of a B-Tree (2)

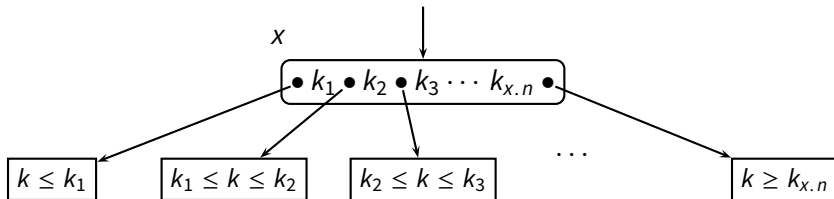


## Definition of a B-Tree (2)



- The keys  $x.key[i]$  delimit the ranges of keys stored in each subtree

## Definition of a B-Tree (2)



- The keys  $x.key[i]$  delimit the ranges of keys stored in each subtree

$x.c[1]$   $\longrightarrow$  subtree containing keys  $k \leq x.key[1]$

$x.c[2]$   $\longrightarrow$  subtree containing keys  $k, x.key[1] \leq k \leq x.key[2]$

$x.c[3]$   $\longrightarrow$  subtree containing keys  $k, x.key[2] \leq k \leq x.key[3]$

...

$x.c[x.n + 1]$   $\longrightarrow$  subtree containing keys  $k, k \geq x.key[x.n]$

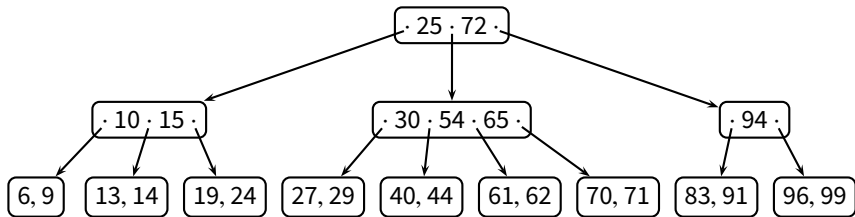
## Definition of a B-Tree (3)

## Definition of a B-Tree (3)

- *All leaves have the same depth*

## Definition of a B-Tree (3)

- **All leaves have the same depth**
- Let  $t \geq 2$  be the **minimum degree** of the B-tree
  - ▶ every node other than the root must have **at least  $t - 1$  keys**
  - ▶ every node must contain **at most  $2t - 1$  keys**
    - ▶ a node is *full* when it contains exactly  $2t - 1$  keys
    - ▶ a full node has  $2t$  children





## **B-TREE-SEARCH**( $x, k$ )

```
1   $i = 1$ 
2  while  $i \leq x.n$  and  $k > x.key[i]$ 
3       $i = i + 1$ 
4  if  $i \leq x.n$  and  $k == x.key[i]$ 
5      return  $(x, i)$ 
6  if  $x.leaf$ 
7      return NIL
8  else DISK-READ( $x.c[i]$ )
9      return B-TREE-SEARCH( $x.c[i], k$ )
```

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

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

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

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

*Proof:*

- ▶  $n \geq 1$ , so the root has at least one key (and therefore two children)

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

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

*Proof:*

- ▶  $n \geq 1$ , so the root has at least one key (and therefore two children)
- ▶ every other node has at least  $t$  children

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

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

*Proof:*

- ▶  $n \geq 1$ , so the root has at least one key (and therefore two children)
- ▶ every other node has at least  $t$  children
- ▶ in the worst case, there are two subtrees (of the root) each one containing a total of  $(n-1)/2$  keys, and each one consisting of  $t$ -degree nodes, with each node containing  $t-1$  keys

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

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

*Proof:*

- ▶  $n \geq 1$ , so the root has at least one key (and therefore two children)
- ▶ every other node has at least  $t$  children
- ▶ in the worst case, there are two subtrees (of the root) each one containing a total of  $(n-1)/2$  keys, and each one consisting of  $t$ -degree nodes, with each node containing  $t-1$  keys
- ▶ each subtree contains  $1 + t + t^2 \cdots + t^{h-1}$  nodes, each one containing  $t-1$  keys

- **Theorem:** the height of a B-tree containing  $n \geq 1$  keys and with a minimum degree  $t \geq 2$  is

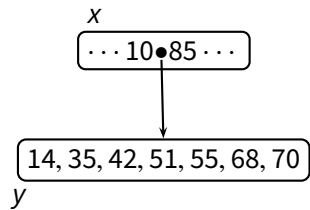
$$h \leq \log_t \frac{n+1}{2}$$

*Proof:*

- ▶  $n \geq 1$ , so the root has at least one key (and therefore two children)
- ▶ every other node has at least  $t$  children
- ▶ in the worst case, there are two subtrees (of the root) each one containing a total of  $(n-1)/2$  keys, and each one consisting of  $t$ -degree nodes, with each node containing  $t-1$  keys
- ▶ each subtree contains  $1 + t + t^2 \cdots + t^{h-1}$  nodes, each one containing  $t-1$  keys, so

$$n \geq 1 + 2(t^h - 1)$$





$x$   
... 10 • 85 ...



14, 35, 42, 51, 55, 68, 70

$y$

---

$x$   
... 10 • 51 • 85 ...



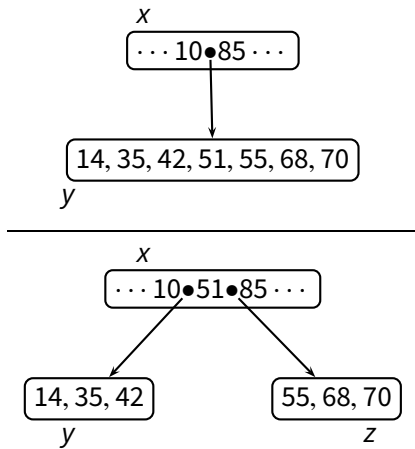
14, 35, 42

$y$



55, 68, 70

$z$

**B-TREE-SPLIT-CHILD**( $x, i, y$ )

```

1   $z = \text{ALLOCATE-NODE}()$ 
2   $z.\text{leaf} = y.\text{leaf}$ 
3   $z.n = t - 1$ 
4  for  $j = 1$  to  $t - 1$ 
5       $z.\text{key}[j] = y.\text{key}[j + t]$ 
6  if not  $y.\text{leaf}$ 
7      for  $j = 1$  to  $t$ 
8           $z.c[j] = y.c[j + t]$ 
9   $y.n = t - 1$ 
10 for  $j = x.n + 1$  downto  $i + 1$ 
11      $x.c[j + 1] = x.c[j]$ 
12 for  $j = x.n$  downto  $i$ 
13      $x.\text{key}[j + 1] = x.\text{key}[j]$ 
14  $x.\text{key}[i] = y.\text{key}[t]$ 
15  $x.n = x.n + 1$ 
16 DISK-WRITE( $y$ )
17 DISK-WRITE( $z$ )
18 DISK-WRITE( $x$ )

```

# Complexity of B-TREE-SPLIT-CHILD

- What is the complexity of **B-TREE-SPLIT-CHILD**?

# Complexity of B-TREE-SPLIT-CHILD

- What is the complexity of **B-TREE-SPLIT-CHILD**?
- $\Theta(t)$  basic CPU operations

# Complexity of B-TREE-SPLIT-CHILD

- What is the complexity of **B-TREE-SPLIT-CHILD**?
- $\Theta(t)$  basic CPU operations
- 3 **DISK-WRITE** operations

```
B-TREE-SPLIT-CHILD( $x, i, y$ )
1   $z = \mathbf{ALLOCATE-NODE}()$ 
2   $z.leaf = y.leaf$ 
3   $z.n = t - 1$ 
4  for  $j = 1$  to  $t - 1$ 
5       $x.key[j] = x.key[j + t]$ 
6  if not  $x.leaf$ 
7      for  $j = 1$  to  $t$ 
8           $z.c[j] = y.c[j + t]$ 
9   $y.n = t - 1$ 
10 for  $j = x.n + 1$  downto  $i + 1$ 
11      $x.c[j + 1] = x.c[j]$ 
12 for  $j = x.n$  downto  $i$ 
13      $x.key[j + 1] = x.key[j]$ 
14  $x.key[i] = y.key[t]$ 
15  $x.n = x.n + 1$ 
16 DISK-WRITE( $y$ )
17 DISK-WRITE( $z$ )
18 DISK-WRITE( $x$ )
```

# Insertion Under Non-Full Node

# Insertion Under Non-Full Node

```
B-TREE-INSERT-NONFULL( $x, k$ )
1   $i = x.n$                                 // assume  $x$  is not full
2  if  $x.leaf$ 
3      while  $i \geq 1$  and  $k < x.key[i]$ 
4           $x.key[i+1] = x.key[i]$ 
5           $i = i - 1$ 
6       $x.key[i+1] = k$ 
7       $x.n = x.n + 1$ 
8      DISK-WRITE( $x$ )
9  else while  $i \geq 1$  and  $k < x.key[i]$ 
10      $i = i - 1$ 
11      $i = i + 1$ 
12     DISK-READ( $x.c[i]$ )
13     if  $x.c[i].n == 2t - 1$                 // child  $x.c[i]$  is full
14         B-TREE-SPLIT-CHILD( $x, i, x.c[i]$ )
15         if  $k > x.key[i]$ 
16              $i = i + 1$ 
17     B-TREE-INSERT-NONFULL( $x.c[i], k$ )
```

# Insertion Procedure

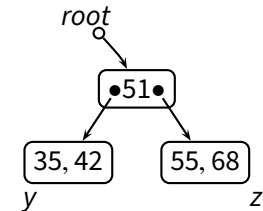
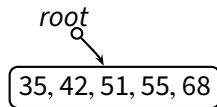
## **B-TREE-INSERT**( $T, k$ )

```
1   $r = T.root$ 
2  if  $r.n == 2t - 1$ 
3       $s = \mathbf{ALLOCATE-NODE}()$ 
4       $T.root = s$ 
5       $s.leaf = \mathbf{FALSE}$ 
6       $s.n = 0$ 
7       $s.c[1] = r$ 
8      B-TREE-SPLIT-CHILD( $s, 1, r$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-TREE-INSERT-NONFULL( $r, k$ )
```

# Insertion Procedure

## **B-TREE-INSERT**( $T, k$ )

```
1   $r = T.root$ 
2  if  $r.n == 2t - 1$ 
3       $s = \mathbf{ALLOCATE-NODE}()$ 
4       $T.root = s$ 
5       $s.leaf = \mathbf{FALSE}$ 
6       $s.n = 0$ 
7       $s.c[1] = r$ 
8      B-TREE-SPLIT-CHILD( $s, 1, r$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-TREE-INSERT-NONFULL( $r, k$ )
```



- What is the complexity of **B-TREE-INSERT**?

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$  basic CPU steps operations

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$  basic CPU steps operations
- $O(h) = O(\log_t n)$  disk-access operations

- What is the complexity of **B-TREE-INSERT**?
- $O(th) = O(t \log_t n)$  basic CPU steps operations
- $O(h) = O(\log_t n)$  disk-access operations
- The best value for  $t$  can be determined according to
  - ▶ the ratio between CPU (RAM) speed and disk-access time
  - ▶ the *block-size* of the disk, which determines the maximum size of an object that can be accessed (read/write) in one shot