Algorithms and Data Structures

Chapter 18: B-trees
(based on book "Introduction to Algorithms" of Cormen et al.)

Vincent Van Schependom
KU Leuven Campus Kulak Kortrijk
Academic year 2024–2025

Outline

1. Definition of a B-tree
2. Use-cases
3. Height of a B-tree
4. Operations on B-trees

Outline

1. Definition of a B-tree
2. Use-cases
3. Height of a B-tree
4. Operations on B-trees

Definition of a B-tree

- Rooted tree with root $T.root$

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.
- All leaves have the same depth, i.e. the tree's height h.

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.
- All leaves have the same depth, i.e. the tree's height h.
- Every node x has $x.n$ keys, stored in monotonically increasing order:

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.
- All leaves have the same depth, i.e. the tree's height h.
- Every node x has $x.n$ keys, stored in monotonically increasing order:

$$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.
- All leaves have the same depth, i.e. the tree's height h.
- Every node x has $x.n$ keys, stored in monotonically increasing order:

$$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$

- Each internal node contains $x.n + 1$ pointers to its children:

Definition of a B-tree

- Rooted tree with root $T.root$
- Self-balancing, just like red-black trees.
- All leaves have the same depth, i.e. the tree's height h.
- Every node x has $x.n$ keys, stored in monotonically increasing order:

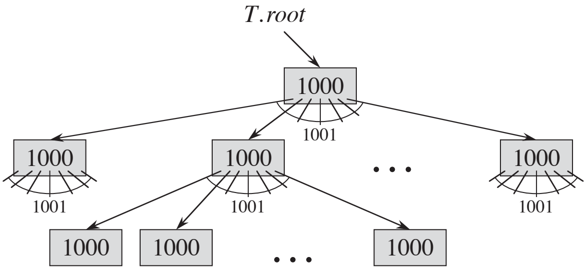
$$x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}$$

- Each internal node contains $x.n + 1$ pointers to its children:

$$x.c_1 \leq x.c_2 \leq \ldots \leq x.c_{x.n+1}$$

Definition of a B-tree

- Rooted tree with troot $T.root$
- All leaves have the same depth, i.e. the tree's height h.
- Every node x has $x.n$ keys, stored in monotonically increasing order.
- Each internal node contains $x.n + 1$ pointers to its children,



1 node,
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

Key seperation

- The keys $x.key_i$ separate the ranges of keys stored in each subtree.

Key seperation

- The keys $x.key_i$ separate the ranges of keys stored in each subtree.
  - Let k_i be any key stored in the subtree with root $x.c_i$

Key seperation

- The keys $x.key_i$ separate the ranges of keys stored in each subtree.
  - Let k_i be any key stored in the subtree with root $x.c_i$
  - So k_i is one of the keys of child $x.c_i$ of node x

Key seperation

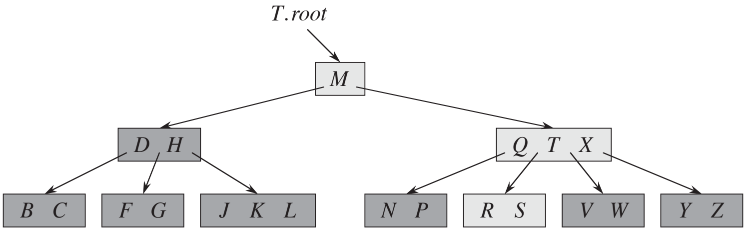
- The keys $x.key_i$ separate the ranges of keys stored in each subtree.
  - Let k_i be any key stored in the subtree with root $x.c_i$
  - So k_i is one of the keys of child $x.c_i$ of node x

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

Key seperation

- The keys $x.key_i$ separate the ranges of keys stored in each subtree.
  - Let k_i be any key stored in the subtree with root $x.c_i$
  - So k_i is one of the keys of child $x.c_i$ of node x

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$



Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least?

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t - 1$ keys.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t - 1$ keys.
  - So, how many children can an internal node have at most?

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t - 1$ keys.
  - So, how many children can an internal node have at most? $2t$ children.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t-1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t-1$ keys.
  - So, how many children can an internal node have at most? $2t$ children.
  - We say that a node is *full* if it contains exactly $2t-1$ keys.

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t-1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t-1$ keys.
  - So, how many children can an internal node have at most? $2t$ children.
  - We say that a node is *full* if it contains exactly $2t-1$ keys.
- Why isn't a minimum degree of $t=1$ allowed?

Minimum degree

- Nodes have lower and upper bounds on the number of keys they can contain.
- Expressed in terms of a fixed integer $t \geq 2$, called the *minimum degree*
- Lower bound:
  - Every node other than the root must have at least $t - 1$ keys.
  - So, how many children does an internal node need at least? t children.
  - If the tree is nonempty, the root must have at least one key.
- Upper bound:
  - Every node may contain at most $2t - 1$ keys.
  - So, how many children can an internal node have at most? $2t$ children.
  - We say that a node is *full* if it contains exactly $2t - 1$ keys.
- Why isn't a minimum degree of $t = 1$ allowed? Trivial.

Outline

1. Definition of a B-tree
2. Use-cases
3. Height of a B-tree
4. Operations on B-trees

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:
    - Fast.

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:
  - Fast.
  - Expensive and limited in capacity.

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:
  - Fast.
  - Expensive and limited in capacity.
- Secondary memory:

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:
  - Fast.
  - Expensive and limited in capacity.
- Secondary memory:
  - Cheaper and much larger.

Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

- Primary memory:
  - Fast.
  - Expensive and limited in capacity.
- Secondary memory:
  - Cheaper and much larger.
  - We need such capacity for storing large amounts of data, like in databases.

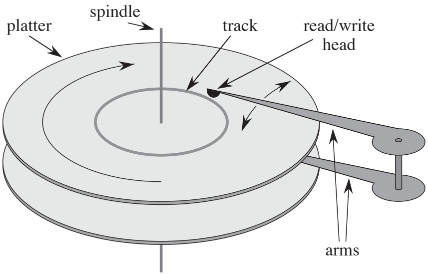
Quick SOCS recap

- Computer systems use a hierarchy of memory technologies.

$$\text{Registers} < \text{Caches} < \text{Main Memory} < \text{Secondary Memory}$$

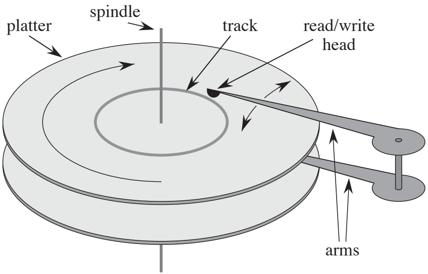
- Primary memory:
  - Fast.
  - Expensive and limited in capacity.
- Secondary memory:
  - Cheaper and much larger.
  - We need such capacity for storing large amounts of data, like in databases.
  - E.g. SSD's and HDD's.

Secondary memory is slow – Example: HDD



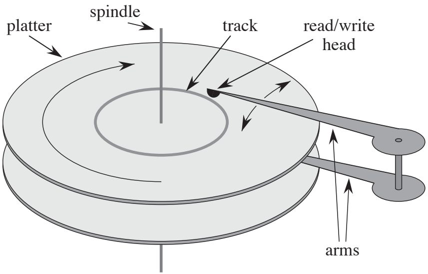
- Because of the moving mechanical parts, secondary memory is slow.

Secondary memory is slow – Example: HDD



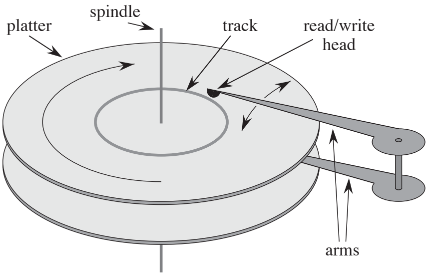
- Because of the moving mechanical parts, secondary memory is slow.
- On average, we need to wait half a turn.

Secondary memory is slow – Example: HDD



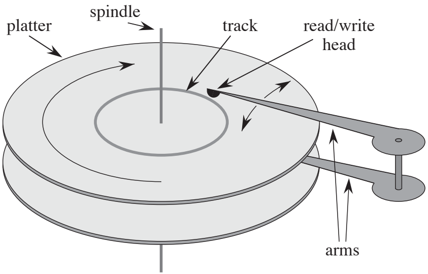
- Because of the moving mechanical parts, secondary memory is slow.
- On average, we need to wait half a turn.
- Moving the arm also takes time.

Secondary memory is slow – Example: HDD



- Because of the moving mechanical parts, secondary memory is slow.
- On average, we need to wait half a turn.
- Moving the arm also takes time.
- SDD are not mechanical, but still significantly slower dan RAM.

Secondary memory is slow – Example: HDD



- Because of the moving mechanical parts, secondary memory is slow.
- On average, we need to wait half a turn.
- Moving the arm also takes time.
- SDD are not mechanical, but still significantly slower dan RAM.

  $\rightarrow$ latency!

Combating latency

- Access not just one item, but several at a time.

Combating latency

- Access not just one item, but several at a time.
  - B-trees are a very common datastructure for handling this.

Combating latency

- Access not just one item, but several at a time.
  - B-trees are a very common datastructure for handling this.
- Information is divided into a number of equal-sized *blocks*.

Combating latency

- Access not just one item, but several at a time.
  - B-trees are a very common datastructure for handling this.
- Information is divided into a number of equal-sized *blocks*.
- Each node is usually as large as a whole disk block.

Combating latency

- Access not just one item, but several at a time.
  - B-trees are a very common datastructure for handling this.
- Information is divided into a number of equal-sized *blocks*.
- Each node is usually as large as a whole disk block.
  - The number of blocks read or written provides a good approximation of the total time spent accessing the disk drive.

Combating latency

- Access not just one item, but several at a time.
  - B-trees are a very common datastructure for handling this.
- Information is divided into a number of equal-sized *blocks*.
- Each node is usually as large as a whole disk block.
  - The number of blocks read or written provides a good approximation of the total time spent accessing the disk drive.
  - So we want the height to be as small as possible.

Outline

1. Definition of a B-tree
2. Use-cases
3. Height of a B-tree
4. Operations on B-trees

Height of a B-tree

Theorem

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$

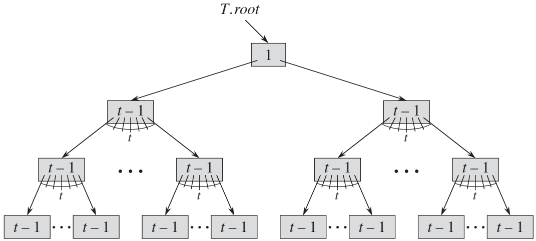
$$h \leq \log_t \left(\frac{n+1}{2}\right)$$

Height of a B-tree

Theorem

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$

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

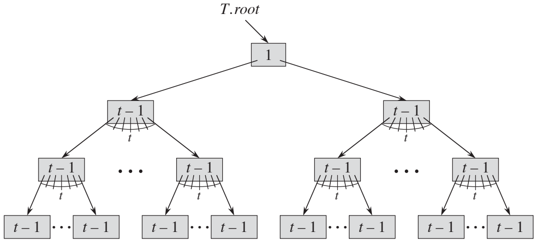


Height of a B-tree

Theorem

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$

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



depth	0	1	2	3	...	h
number of nodes	1	2	$2t$	$2t^2$	...	2^{h-1}

Height of a B-tree

Proof.

Height of a B-tree

Proof. The number of keys n satisfies

$$n \geq \overbrace{\underbrace{1}_{\text{root}} + \underbrace{(t-1)}_{\text{other nodes}}}^{\text{keys per node}} \overbrace{\sum_{i=1}^{h} 2t^{h-1}}^{\#\text{nodes}}$$

Height of a B-tree

Proof. The number of keys n satisfies

$$n \geq \overbrace{\underbrace{1}_{\text{root}} + \underbrace{(t-1)}_{\text{other nodes}}}^{\text{keys per node}} \overbrace{\sum_{i=1}^{h} 2t^{h-1}}^{\#\text{nodes}}$$

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

Height of a B-tree

Proof. The number of keys n satisfies

$$
\begin{aligned}
n &\geq \overbrace{\underbrace{1}_{\text{root}} + \underbrace{(t-1)}_{\text{other nodes}}}^{\text{keys per node}} \overbrace{\sum_{i=1}^{h} 2t^{h-1}}^{\#\text{nodes}} \\
&= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\
&= 2t^h - 1
\end{aligned}
$$

Height of a B-tree

Proof. The number of keys n satisfies

$$n \geq \overbrace{\underbrace{1}_{\text{root}} + \underbrace{(t-1)}_{\text{other nodes}}}^{\text{keys per node}} \overbrace{\sum_{i=1}^{h} 2t^{h-1}}^{\#\text{nodes}}$$

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

$$= 2t^h - 1$$

Thus

$$t^h \leq \frac{n+1}{2}$$

Height of a B-tree

Proof. The number of keys n satisfies

$$
\begin{aligned}
n &\geq \underbrace{1}_{\text{root}} + \overbrace{\underbrace{(t-1)}_{\text{other nodes}}}^{\text{keys per node}} \overbrace{\sum_{i=1}^{h} 2t^{h-1}}^{\#\text{nodes}} \\
&= 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right) \\
&= 2t^h - 1
\end{aligned}
$$

Thus

$$
t^h \leq \frac{n+1}{2} \quad \Longleftrightarrow \quad \boxed{h \leq \log_t\left(\frac{n+1}{2}\right)}
$$

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.
  - Constant t is left out in asymptotic notation.

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.
  - Constant t is left out in asymptotic notation.
  - The base logarithm for B-trees is much larger: t vs 2.

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.
  - Constant t is left out in asymptotic notation.
  - The base logarithm for B-trees is much larger: t vs 2.
- Factor of $\lg t$ saved over red-black trees.

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.
  - Constant t is left out in asymptotic notation.
  - The base logarithm for B-trees is much larger: t vs 2.
- Factor of $\lg t$ saved over red-black trees.
- B-trees are limited in depth but contain an enourmous amount of keys.

Comparison with red-black trees

- Height of the tree grows as $O(\lg n)$ in both cases.
  - Constant t is left out in asymptotic notation.
  - The base logarithm for B-trees is much larger: t vs 2.
- Factor of $\lg t$ saved over red-black trees.
- B-trees are limited in depth but contain an enourmous amount of keys.
  $\rightarrow$ Few memory operations, so minimal memory latency.

Outline

1 Definition of a B-tree

2 Use-cases

3 Height of a B-tree

4 Operations on B-trees

Searching a B-tree

- B-TREE-SEARCH is much like searching a binary tree.

Searching a B-tree

- B-TREE-SEARCH is much like searching a binary tree.
- However, *multiway* (as opposed to binary) branching decisions are made.

Searching a B-tree

- B-TREE-SEARCH is much like searching a binary tree.
- However, *multiway* (as opposed to binary) branching decisions are made.
  - The amount of *ways* is equal to the number of children.

Searching a B-tree

- B-TREE-SEARCH is much like searching a binary tree.
- However, *multiway* (as opposed to binary) branching decisions are made.
  - The amount of *ways* is equal to the number of children.
  - At each internal node x, the search makes an $(x.n + 1)$-way branching decision.

Searching a B-tree: pseudocode

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Searching a B-tree: pseudocode

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- x is a pointer to the root node of a subtree

Searching a B-tree: pseudocode

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- x is a pointer to the root node of a subtree
- k is the key to be searched

Searching a B-tree: pseudocode

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- x is a pointer to the root node of a subtree
- k is the key to be searched
- If k is in the B-tree, B-TREE-SEARCH returns the node-index pair (y, i) such that $y.key_i = k$. Otherwise, the procedure returns NIL.

Searching a B-tree: pseudocode

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
      i = i + 1
if i ≤ x.n and k == x.key_i
      return (x, i)
elseif x.leaf
      return NIL
else DISK-READ(x.c_i)
      return B-TREE-SEARCH(x.c_i, k)
```

- x is a pointer to the root node of a subtree
- k is the key to be searched
- If k is in the B-tree, B-TREE-SEARCH returns the node-index pair (y, i) such that $y.key_i = k$. Otherwise, the procedure returns NIL.

- Recall that $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$ where k_i is any key of child $x.c_i$

Searching a B-tree: memory accesses

- The nodes encountered during the recursion form a simple path downward from the root to the tree.

Searching a B-tree: memory accesses

- The nodes encountered during the recursion form a simple path downward from the root to the tree.
- Thus, B-TREE-SEARCH executes $O(h) = O(\log_t n)$ memory accesses.

Searching a B-tree: memory accesses

- The nodes encountered during the recursion form a simple path downward from the root to the tree.
- Thus, $\textsc{B-Tree-Search}$ executes $O(h) = O(\log_t n)$ memory accesses.
  - Recall that $h \leq \log_t \left(\frac{n+1}{2}\right)$.

Searching a B-tree: memory accesses

- The nodes encountered during the recursion form a simple path downward from the root to the tree.
- Thus, $\textsc{B-Tree-Search}$ executes $O(h) = O(\log_t n)$ memory accesses.
  - Recall that $h \le \log_t \left(\frac{n+1}{2}\right)$.
  - Constants are left out in O-notation.

Searching a B-tree: runtime

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- For every node x, $x.n < 2t$, so the while loop takes $O(1)$ time for each encountered node.

Searching a B-tree: runtime

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- For every node x, $x.n < 2t$, so the while loop takes $O(1)$ time for each encountered node.
- The total runtime is $O(th) = O(t \log_t n)$.

Inserting a key

- Significantly more complicated than inserting a key into a binary tree.

Inserting a key

- Significantly more complicated than inserting a key into a binary tree.
- We cannot simply *create* a new leaf node and insert it, because the resulting tree would fail to be a valid B-tree.

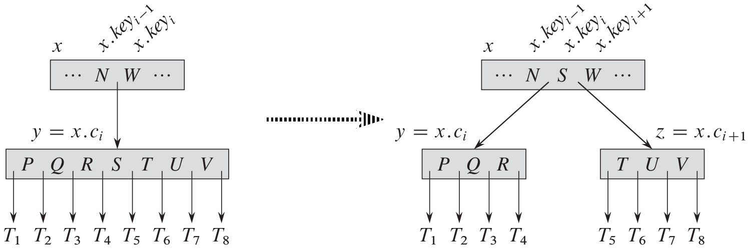
Inserting a key

- Significantly more complicated than inserting a key into a binary tree.
- We cannot simply *create* a new leaf node and insert it, because the resulting tree would fail to be a valid B-tree.
- So, we need to insert the key into an *existing* leaf node.

Inserting a key

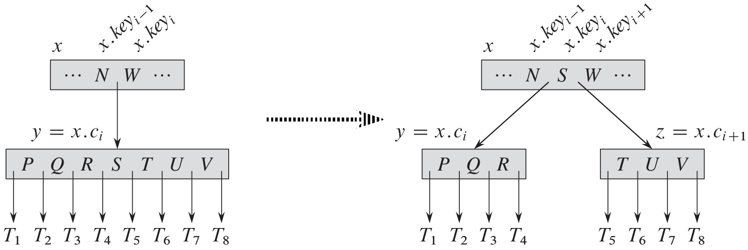
- Significantly more complicated than inserting a key into a binary tree.
- We cannot simply *create* a new leaf node and insert it, because the resulting tree would fail to be a valid B-tree.
- So, we need to insert the key into an *existing* leaf node.
- What if a node y is *full*, i.e. what if $y.n = 2t - 1$?

Splitting a (full) node y with $y.n = 2t - 1$



$\textsc{B-Tree-Split-Child}(x, i)$

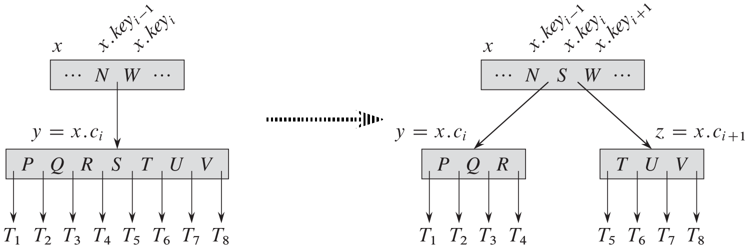
Splitting a (full) node y with $y.n = 2t - 1$



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

- x is a non-full node and $y = x.c_i$ is a full child of x.

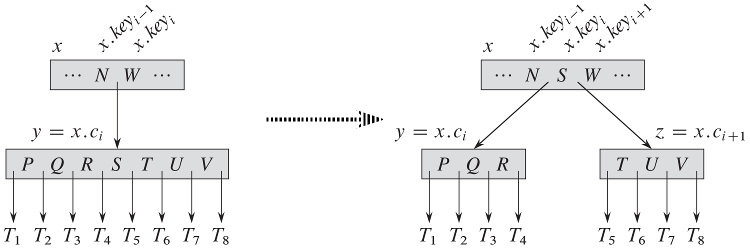
Splitting a (full) node y with $y.n = 2t - 1$



B-TREE-SPLIT-CHILD(x, i)

- x is a non-full node and $y = x.c_i$ is a full child of x.
- Split y about its median key S and move S up into y's parent node x.

Splitting a (full) node y with $y.n = 2t - 1$



B-TREE-SPLIT-CHILD(x, i)

- x is a non-full node and $y = x.c_i$ is a full child of x.
- Split y about its median key S and move S up into y's parent node x.
- Every key $y.key_i$ that is greater than the median S, is placed in a new node z, which is a new child of x.

Inserting a key (continued)

Inserting a key (continued)

B-TREE-INSERT(T, k)

- Checks if the root $r = T.root$ is full. If it is:

KU LEUVEN kulak

Inserting a key (continued)

$\text{B-T{\scriptsize REE}-I{\scriptsize NSERT}}(T, k)$

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.

Inserting a key (continued)

$\text{B-TREE-INSERT}(T, k)$

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.
  - Splits the old root by calling $\text{B-TREE-SPLIT-CHILD}(s, 1)$.

Inserting a key (continued)

$\text{B-T}{\small\text{REE}}\text{-I}{\small\text{NSERT}}(T, k)$

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.
  - Splits the old root by calling $\text{B-T}{\small\text{REE}}\text{-S}{\small\text{PLIT}}\text{-C}{\small\text{HILD}}(s, 1)$.
- Calls $\text{B-T}{\small\text{REE}}\text{-I}{\small\text{NSERT}}\text{-N}{\small\text{ONFULL}}(x, k)$ with either $x = r$ or $x = s$

KU LEUVEN kulak

Inserting a key (continued)

$\text{B-TREE-INSERT}(T, k)$

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.
  - Splits the old root by calling $\text{B-TREE-SPLIT-CHILD}(s, 1)$.
- Calls $\text{B-TREE-INSERT-NONFULL}(x, k)$ with either $x = r$ or $x = s$

Complexity?

Inserting a key (continued)

B-TREE-INSERT(T, k)

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.
  - Splits the old root by calling B-TREE-SPLIT-CHILD$(s, 1)$.
- Calls B-TREE-INSERT-NONFULL(x, k) with either $x = r$ or $x = s$

Complexity?

- Again $O(h) = O(\log_t n)$ disk accesses.

Inserting a key (continued)

$\text{B-TREE-INSERT}(T, k)$

- Checks if the root $r = T.root$ is full. If it is:
  - Make the current root child of a new empty root s.
  - Splits the old root by calling $\text{B-TREE-SPLIT-CHILD}(s, 1)$.
- Calls $\text{B-TREE-INSERT-NONFULL}(x, k)$ with either $x = r$ or $x = s$

Complexity?

- Again $O(h) = O(\log_t n)$ disk accesses.
- Again $O(th) = O(t \log_t n)$ time required.

Questions?