Analysis of Algorithms

B-Trees

Andres Mendez-Vazquez

November 5, 2018

Outline

Outline

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Disk-based Environments

Something Notable

We have the following hierarchy of data access speed

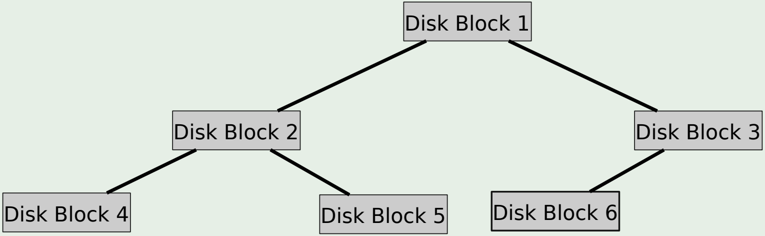
1. CPU
2. Cache
3. Main Memory
4. Secondary Storage: Magnetic Disks and SSD
5. Tertiary Storage: Tapes

We know the following

- Data is stored on disk in units called blocks or pages.
- Every disk access has to read/write one or multiple blocks.
- Even if we need to access a single integer stored in a disk block which contains thousands of integers, **we need to read the whole block in.**

Now, What if you use a binary tree

In this structure the nodes are disk blocks

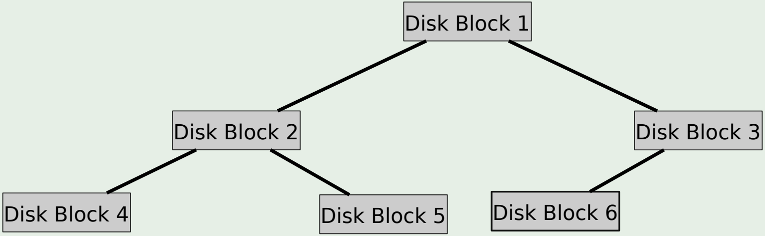


Still, We have the following problem

- If a disk block is 8K (8192 bytes)

Now, What if you use a binary tree

In this structure the nodes are disk blocks

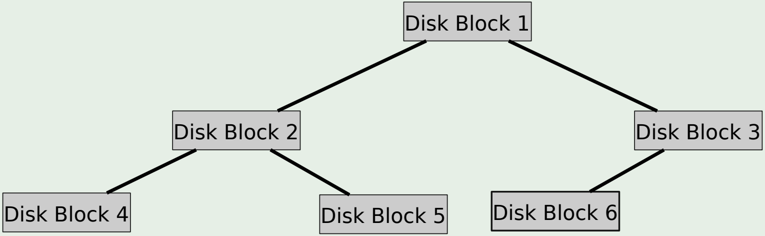


Still, We have the following problem

- If a disk block is 8K (8192 bytes)
- Problem the necessary information for a node is
  - A key = 4 bytes
  - A value = 4 bytes
  - Two Children = 8 bytes

Now, What if you use a binary tree

In this structure the nodes are disk blocks



Still, We have the following problem

- If a disk block is 8K (8192 bytes)
- Problem the necessary information for a node is
  - A key = 4 bytes
  - A value = 4 bytes
  - Two Children = 8 bytes

Problem!!!

Then

We use only 0.2% of the block is full

Even

If we store multiple tree nodes in a disk!!!

Problem!!!

Then

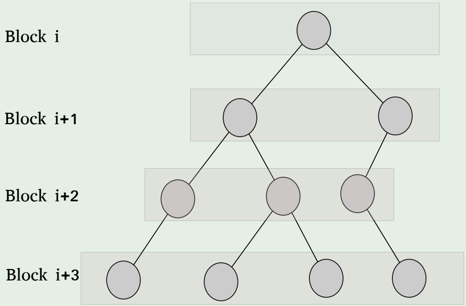
We use only 0.2% of the block is full

Even

If we store multiple tree nodes in a disk!!!

However

The query and update need to access $O(\log_2 n)$ nodes



Worst Case $O(\log_2 n)$ accesses to disk!!!

Increase the branching

With a large B

$$\log_B n \ll log_2 n \quad (1)$$

Ok

- We can minimize the number of disk access by increasing the branching!!!

Increase the branching

With a large B

$$\log_B n \ll log_2 n \quad (1)$$

Ok

- We can minimize the number of disk access by increasing the branching!!!
- We need a way to access elements in the new branching.

Motivation for B-Trees

Some facts!

- Index structures for large datasets cannot be stored in main memory (Actually, not anymore the case!!!).
- Storing it on disk requires different approach to efficiency.
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7 ms.
- Crudely speaking, one disk access takes about the same time as 200,000 instructions!

Motivation for B-Trees

Some facts!

- Index structures for large datasets cannot be stored in main memory (Actually, not anymore the case!!!).
- Storing it on disk requires different approach to efficiency.
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7 ms.
- Crudely speaking, one disk access takes about the same time as 200,000 instructions!

Motivation for B-Trees

Some facts!

- Index structures for large datasets cannot be stored in main memory (Actually, not anymore the case!!!).
- Storing it on disk requires different approach to efficiency.
- Assuming that a disk spins at 3600 RPM, one revolution occurs in $1/60$ of a second, or 16.7 ms.
- Crudely speaking, one disk access takes about the same time as 200,000 instructions!

Motivation for B-Trees

Some facts!

- Index structures for large datasets cannot be stored in main memory (Actually, not anymore the case!!!).
- Storing it on disk requires different approach to efficiency.
- Assuming that a disk spins at 3600 RPM, one revolution occurs in $1/60$ of a second, or 16.7 ms.
- Crudely speaking, one disk access takes about the same time as 200,000 instructions!

Motivation for B-Trees

Now

- Assume that we use a binary tree to store about 20 million records.
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20 \times 10^6$ is about 24, so this takes about 0.2 seconds.
- We know we can't improve on the $\log_2 n$ lower bound on search for a binary tree.
- However, the solution is to use more branches and thus reduce the height of the tree! As branching increases, depth decreases.

Motivation for B-Trees

Now

- Assume that we use a binary tree to store about 20 million records.
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20 \times 10^6$ is about 24, so this takes about 0.2 seconds.
- We know we can't improve on the $\log_2 n$ lower bound on search for a binary tree.
- However, the solution is to use more branches and thus reduce the height of the tree! As branching increases, depth decreases.

Motivation for B-Trees

Now

- Assume that we use a binary tree to store about 20 million records.
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20 \times 10^6$ is about 24, so this takes about 0.2 seconds.
- We know we can't improve on the $\log_2 n$ lower bound on search for a binary tree.
- However, the solution is to use more branches and thus reduce the height of the tree! As branching increases, depth decreases.

Motivation for B-Trees

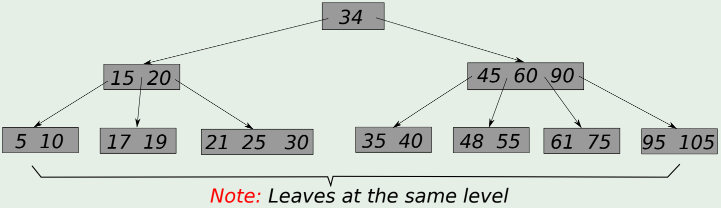
Now

- Assume that we use a binary tree to store about 20 million records.
- We end up with a very deep binary tree with lots of different disk accesses; $\log_2 20 \times 10^6$ is about 24, so this takes about 0.2 seconds.
- We know we can't improve on the $\log_2 n$ lower bound on search for a binary tree.
- However, the solution is to use more branches and thus reduce the height of the tree! As branching increases, depth decreases.

Outline

B-Trees definition

Example

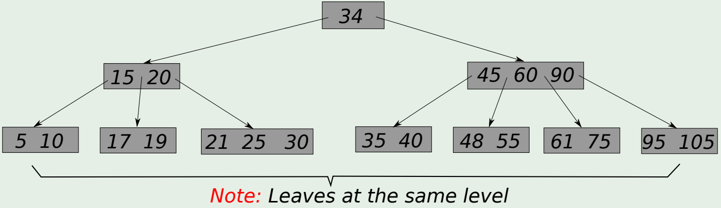


Definitions

- Every node x has the following attributes:

B-Trees definition

Example

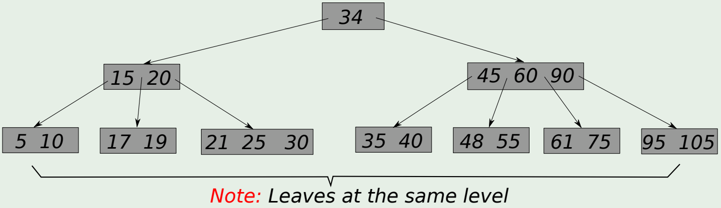


Definitions

- Every node x has the following attributes:
  - $x.n$ number of keys stored at node x.
    - Each key has an associated payload (Pointer, values, etc).
  - The keys are sorted $key_1 \leq key_2 \leq \ldots \leq key_{x.n}$.
  - $x.leaf$ is a boolean value and denotes a leaf when is set to TRUE.

B-Trees definition

Example

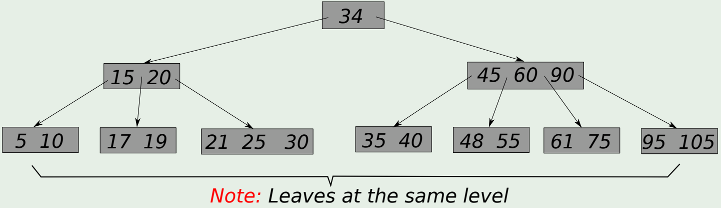


Definitions

- Every node x has the following attributes:
  - $x.n$ number of keys stored at node x.
    - Each key has an associated payload (Pointer, values, etc).
  - The keys are sorted $key_1 \leq key_2 \leq ... \leq key_{x.n}$.
  - $x.leaf$ is a boolean value and denotes a leaf when is set to TRUE.

B-Trees definition

Example

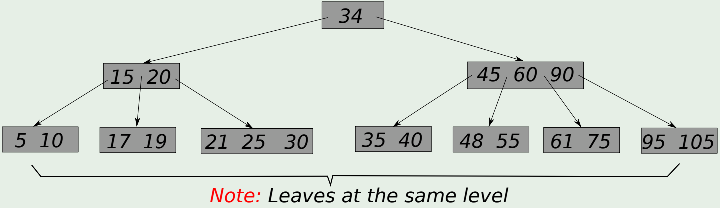


Definitions

- Every node x has the following attributes:
  - $x.n$ number of keys stored at node x.
    - Each key has an associated payload (Pointer, values, etc).
  - The keys are sorted $key_1 \leq key_2 \leq ... \leq key_{x.n}$.
  - $x.leaf$ is a boolean value and denotes a leaf when is set to TRUE.

B-Trees definition

Example



B-Trees definition

In addition

- Every node x has the following attributes:
  - It contains $x.n + 1$ pointers to its children:

$$x.c_1, x.c_2, ..., x.c_{n+1}$$

    - ★ Leaf nodes do not have children then they leave this field undefined.
    - ★ The keys are used to separate the keys stored at the B-Tree. For example, if k_i is any key stored in the subtree stored at tree with root $x.c_i$ then

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

B-Trees definition

In addition

- Every node x has the following attributes:
  - It contains $x.n+1$ pointers to its children:

$$x.c_1, x.c_2, ..., x.c_{n+1}$$

    - ★ Leaf nodes do not have children then they leave this field undefined.
    - ★ The keys are used to separate the keys stored at the B-Tree. For example, if k_i is any key stored in the subtree stored at tree with root $x.c_i$ then

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

B-Trees definition

In addition

- Every node x has the following attributes:
  - It contains $x.n + 1$ pointers to its children:

$$x.c_1, x.c_2, ..., x.c_{n+1}$$

    - ★ Leaf nodes do not have children then they leave this field undefined.
    - ★ The keys are used to separate the keys stored at the B-Tree. For example, if k_i is any key stored in the subtree stored at tree with root $x.c_i$ then

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

B-Trees definition

In addition

- Every node x has the following attributes:
  - It contains $x.n+1$ pointers to its children:

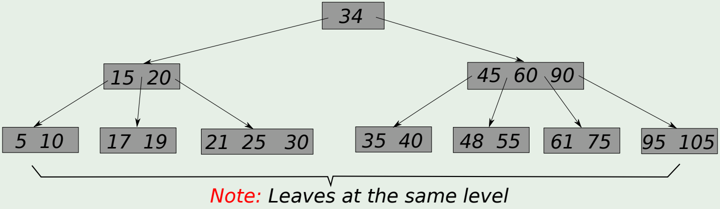
$$x.c_1, x.c_2, ..., x.c_{n+1}$$

    - ★ Leaf nodes do not have children then they leave this field undefined.
    - ★ The keys are used to separate the keys stored at the B-Tree. For example, if k_i is any key stored in the subtree stored at tree with root $x.c_i$ then

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

B-Trees definition

Example

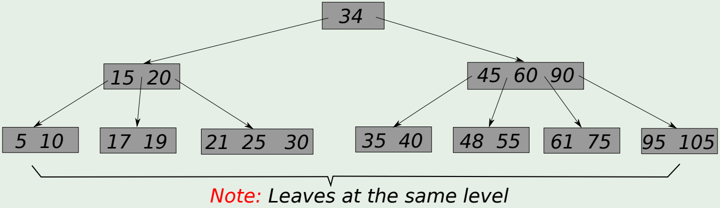


Minimum Degree

- A fixed integer $t \geq 2$ is called the minimum degree or branching of the tree:
  - if $x \neq root \to t-1 \leq x.n \leq 2t-1$
  - If $x = root \to 1 \leq x.n \leq 2t-1$

B-Trees definition

Example



Minimum Degree

- A fixed integer $t \geq 2$ is called the minimum degree or branching of the tree:
  - if $x \neq root \rightarrow t-1 \leq x.n \leq 2t-1$
  - If $x = root \rightarrow 1 \leq x.n \leq 2t-1$

Outline

We want to store large sets of indexes

First

We assume that the set is so voluminous that only a small part can be kept in main memory!!!

Thus

We want to minimize the number of access to hard drive by using the locality principle!!!

We want to store large sets of indexes

First

We assume that the set is so voluminous that only a small part can be kept in main memory!!!

Thus

We want to minimize the number of access to hard drive by using the locality principle!!!

Application: Minimizing disk access when looking for indexes in databases

Each node is stored as a page

Page size determines t. Since t is usually large, this implies a large branching factor, so height is small.

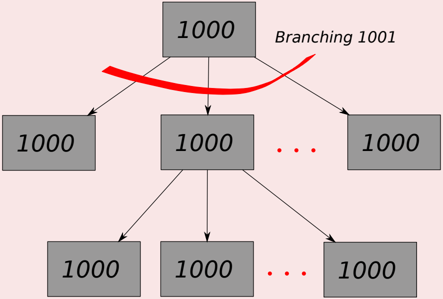
Example with $t = 1001$, we have 1000 (key, elements) per node

Application: Minimizing disk access when looking for indexes in databases

Each node is stored as a page

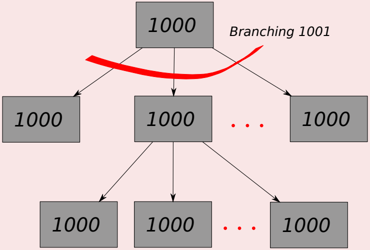
Page size determines t. Since t is usually large, this implies a large branching factor, so height is small.

Example with $t = 1001$, we have 1000 (key, elements) per node



Application: Minimizing disk access when looking for indexes in databases

Example with $(2t-1)+1 = 1001$, we have 1000 (key, elements) per node



Application: Minimizing disk access when looking for indexes in databases

The example above

- It can hold over one billion keys.
  - the **height is only** 2 (Assuming root at height 0), so we can find any key with only two disk accesses (Compared to red-black trees, where the branching factor is 2).
  - Then, disk accesses are minimal!!!

Application: Minimizing disk access when looking for indexes in databases

The example above

- It can hold over one billion keys.
  - the **height is only** 2 (Assuming root at height 0), so we can find any key with only two disk accesses (Compared to red-black trees, where the branching factor is 2).
  - Then, disk accesses are minimal!!!

Outline

Height of a B-Tree

Theorem 18.1

Let n be the number of keys in T, $n \geq 1, t \geq 2$, and h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$

Proof

- The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys.

Height of a B-Tree

Theorem 18.1

Let n be the number of keys in T, $n \geq 1, t \geq 2$, and h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$

Proof

- The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys.
- Thus, T , whose height is h,
  - It has at least 2 nodes at depth 1.
  - At least $2t$ nodes at depth 2.
  - At least $2t^2$ nodes at depth 3.
  - Then, depth h has at least $2t^{h-1}$ nodes.

Height of a B-Tree

Theorem 18.1

Let n be the number of keys in T, $n \geq 1, t \geq 2$, and h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$

Proof

- The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys.
- Thus, T , whose height is h,
  - It has at least 2 nodes at depth 1.
  - At least $2t$ nodes at depth 2.
  - At least $2t^2$ nodes at depth 3.
  - Then, depth h has at least $2t^{h-1}$ nodes.

Height of a B-Tree

Theorem 18.1

Let n be the number of keys in T, $n \geq 1, t \geq 2$, and h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$

Proof

- The root of a B-tree T contains at least one key, and all other nodes contain at least $t - 1$ keys.
- Thus, T , whose height is h,
  - It has at least 2 nodes at depth 1.
  - At least $2t$ nodes at depth 2.
  - At least $2t^2$ nodes at depth 3.
  - Then, depth h has at least $2t^{h-1}$ nodes.

Height of a B-Tree

Theorem 18.1

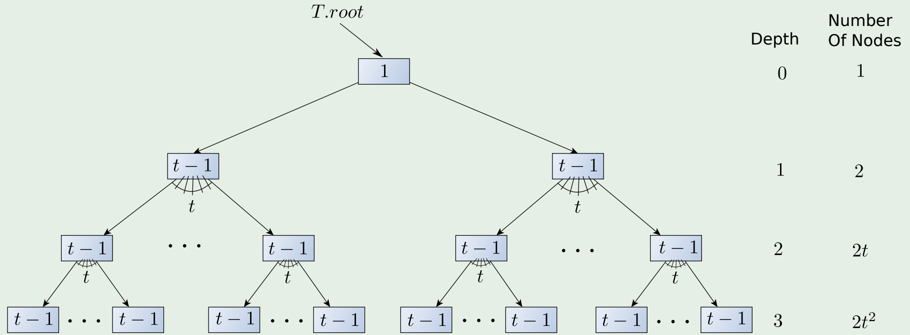
Let n be the number of keys in T, $n \geq 1, t \geq 2$, and h be the height of T. Then $h \leq \log_t \frac{n+1}{2}$

Proof

- The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys.
- Thus, T , whose height is h,
  - It has at least 2 nodes at depth 1.
  - At least $2t$ nodes at depth 2.
  - At least $2t^2$ nodes at depth 3.
  - Then, depth h has at least $2t^{h-1}$ nodes.

For example

We have the following



Height of a B-Tree

We have at least

1. Depth 0 - One key
2. Depth 1 - $2t^0(t-1)$
3. Depth 2 - $2t^1(t-1)$
4. Depth 3 - $2t^2(t-1)$
5. ...

Height of a B-Tree

We have at least

1. Depth 0 - One key
2. Depth 1 - $2t^0(t-1)$
3. Depth 2 - $2t^1(t-1)$
4. Depth 3 - $2t^2(t-1)$
5. ...

Thus

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

Height of a B-Tree

We have at least

1. Depth 0 - One key
2. Depth 1 - $2t^0(t-1)$
3. Depth 2 - $2t^1(t-1)$
4. Depth 3 - $2t^2(t-1)$
5. ...

Thus

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

Height of a B-Tree

We have at least

1. Depth 0 - One key
2. Depth 1 - $2t^0(t-1)$
3. Depth 2 - $2t^1(t-1)$
4. Depth 3 - $2t^2(t-1)$
5. ...

Thus

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

Height of a B-Tree

We have at least

1. Depth 0 - One key
2. Depth 1 - $2t^0(t-1)$
3. Depth 2 - $2t^1(t-1)$
4. Depth 3 - $2t^2(t-1)$
5. ...

Thus

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

Height of a B-Tree

Finally

$$n \geq 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right) = 2t^h - 1 \quad (3)$$

Height of a B-Tree

Finally

$$n \geq 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1 \tag{3}$$

Therefore

$$t^h \leq \frac{n+1}{2} \tag{4}$$

Height of a B-Tree

Finally

$$h \leq \log_t \frac{n+1}{2} \tag{5}$$

Outline

Constraints on the Operations

The root of the B-tree is always in main memory

1. Disk-Read are never performed on it.
2. Only When is written, we use a Disk-Write.

Constraints on the Operations

The root of the B-tree is always in main memory

1. Disk-Read are never performed on it.
2. Only When is written, we use a Disk-Write.

If a node is passed as parameter

It has already had all the necessary Disk-Read operations performed on it before hand.

- Disk-Write: To move node from memory to disk.

Constraints on the Operations

The root of the B-tree is always in main memory

1. Disk-Read are never performed on it.
2. Only When is written, we use a Disk-Write.

If a node is passed as parameter

It has already had all the necessary Disk-Read operations performed on it before hand.

In the code that follows, we use:

- **Disk-Read**: To move node from disk to memory.
- **Disk-Write**: To move node from memory to disk.

Constraints on the Operations

The root of the B-tree is always in main memory

1. Disk-Read are never performed on it.
2. Only When is written, we use a Disk-Write.

If a node is passed as parameter

It has already had all the necessary Disk-Read operations performed on it before hand.

In the code that follows, we use:

- **Disk-Read**: To move node from disk to memory.
- **Disk-Write**: To move node from memory to disk.

Constraints on the Operations

The root of the B-tree is always in main memory

1. Disk-Read are never performed on it.
2. Only When is written, we use a Disk-Write.

If a node is passed as parameter

It has already had all the necessary Disk-Read operations performed on it before hand.

In the code that follows, we use:

- **Disk-Read**: To move node from disk to memory.
- **Disk-Write**: To move node from memory to disk.

Outline

Search operation

Pseudo-Code

B-Tree-Search(x, k)

- $i = 1$
- **while** $i \leq x.n$ **and** $k > x.key\left[i\right]$
    - $i = i + 1$
- **if** $i \leq x.n$ **and** $k == x.key\left[i\right]$
    - **return** (x, i)
- **elseif** $x.leaf$
    - **return** NIL
- **else** Disk-Read$(x.c\left[i\right])$
    - **return** B-Tree-Search$(x.c\left[i\right], k)$

Search operation

Pseudo-Code

B-Tree-Search(x, k)

1. $i = 1$
2. **while** $i \leq x.n$ and $k > x.key\left[i\right]$
3. $\quad i = i + 1$
4. **if** $i \leq x.n$ and $k == x.key\left[i\right]$
5. $\quad$ **return** (x, i)
6. **elseif** $x.leaf$
7. $\quad$ **return** NIL
8. **else** Disk-Read$(x.c\left[i\right])$
9. $\quad$ **return** B-Tree-Search$(x.c\left[i\right], k)$

Search operation

Pseudo-Code

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)
```

Search operation

Pseudo-Code

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)
```

Search operation

Pseudo-Code

```
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)
```

Using **recursion** to make the search easier

So, we use line 1 to 5

1. Move to the key $x.key[i]$ such that $k \leq x.key[i]$
2. To return the value if stored at the node by the sorted keys!!!

Using **recursion** to make the search easier

So, we use line 1 to 5

1. Move to the key $x.key[i]$ such that $k \leq x.key[i]$
2. To return the value if stored at the node by the sorted keys!!!

If the node is a leaf

Return NIL == "That key is not in the B-Tree"

Using **recursion** to make the search easier

So, we use line 1 to 5

1. Move to the key $x.key\left[i\right]$ such that $k \leq x.key\left[i\right]$
2. To return the value if stored at the node by the sorted keys!!!

If the node is a leaf

Return NIL $==$ "That key is not in the B-Tree"

The key could be in the next level

Then, Disk-Read($x.c\left[i\right]$) and call the recursion in the children node already in memory.

Using **recursion** to make the search easier

So, we use line 1 to 5

1. Move to the key $x.key[i]$ such that $k \leq x.key[i]$
2. To return the value if stored at the node by the sorted keys!!!

If the node is a leaf

Return NIL $==$ "That key is not in the B-Tree"

The key could be in the next level

Then, Disk-Read$(x.c[i])$ and call the recursion in the children node already in memory.

Search operation

Note

$Search(root[t], k)$ **returns** (x, i) **or** NIL **if no such key.**

Cost of Search

Worst Cost

- $O(h) = O(\log_t n)$ disk reads when going through the entire tree.
- $x.n < 2t \Rightarrow O(t)$ for searching the key at each node
- Finally, we have that $O(th) = O(t \log_t n)$ CPU time.

Cost of Search

Worst Cost

- $O(h) = O(\log_t n)$ disk reads when going through the entire tree.
- $x.n < 2t \Rightarrow O(t)$ for searching the key at each node
- Finally, we have that $O(th) = O(t \log_t n)$ CPU time.

Cost of Search

Worst Cost

- $O(h) = O(\log_t n)$ disk reads when going through the entire tree.
- $x.n < 2t \Rightarrow O(t)$ for searching the key at each node
- Finally, we have that $O(th) = O(t \log_t n)$ CPU time.

Outline

Creating an empty tree

Pseudo-Code

B-Tree-Create(T)

1. $x =$Allocate-Node()
2. $x.leaf =$TRUE
3. $x.n = 0$
4. Disk-Write(x)
5. $T.root = x$

Note

- To create a nonempty tree, first create an empty tree and then insert nodes.

Creating an empty tree

Pseudo-Code

B-Tree-Create(T)

1. x =Allocate-Node()
2. $x.leaf$ =TRUE
3. $x.n = 0$
4. Disk-Write(x)
5. $T.root = x$

Note

- To create a nonempty tree, first create an empty tree and then insert nodes.

Cost of Create

Worst Cost

- $O(1)$ **disk accesses.**
- $O(1)$ **CPU time.**

Outline

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t-1$ keys, something needs to be done in order to make space in the node.

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t-1$ keys, something needs to be done in order to make space in the node.

Process

1. Split the node around the median key.
2. You finish with two nodes of size $t-1$ and the median key y.
3. Promote the median key to the father node to identify the new ranges.
4. If the father is full **recursively** split the father to make room.

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t - 1$ keys, something needs to be done in order to make space in the node.

Process

1. Split the node around the median key.
2. You finish with two nodes of size $t - 1$ and the median key y.
3. Promote the median key to the father node to identify the new ranges.
4. If the father is full **recursively** split the father to make room.

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t - 1$ keys, something needs to be done in order to make space in the node.

Process

1. Split the node around the median key.
2. You finish with two nodes of size $t - 1$ and the median key y.
3. Promote the median key to the father node to identify the new ranges.
4. If the father is full **recursively** split the father to make room.

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t-1$ keys, something needs to be done in order to make space in the node.

Process

1. Split the node around the median key.
2. You finish with two nodes of size $t-1$ and the median key y.
3. Promote the median key to the father node to identify the new ranges.
4. If the father is full **recursively** split the father to make room.

Insertion

Something Notable

Here is where the things become interesting!!!

- Insertions can only be done in non-full nodes.
- The holding data structures for keys and pointers are arrays!!!

What?

This means that if a node has $2t-1$ keys, something needs to be done in order to make space in the node.

Process

1. Split the node around the median key.
2. You finish with two nodes of size $t-1$ and the median key y.
3. Promote the median key to the father node to identify the new ranges.
4. If the father is full **recursively** split the father to make room.

Important!!!

We always insert at...

THE LEAF LEVEL!!!

Therefore

What if the leaf child becomes full?

Important!!!

We always insert at...

THE LEAF LEVEL!!!

Therefore

What if the leaf child becomes full?

Splitting

Splitting

Applied to a full child of a non-full parent when full$\equiv 2t-1$ keys.

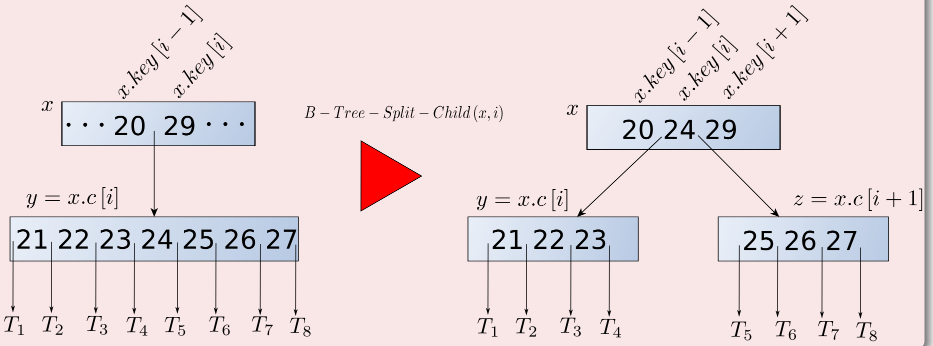
Example with $t=4$

Splitting

Splitting

Applied to a full child of a non-full parent when full$\equiv 2t-1$ keys.

Example with $t=4$



Split-Child

Algorithm

```
B-Tree-Split-Child(x, i)
z = Allocate-Node()
y = x.c_i
z.leaf = y.leaf
z.n = t − 1
for j = 1 and t − 1
        z.key[j] = y.key[j + t]
if not y.leaf
        for j = 1 to t
                z.c[j] = y.c[j + t]
y.n = t − 1
```

```
for j = x.n + 1 downto i + 1
        x.c[j + 1] = x.c[j]
x.c[i + 1] = z
for j = x.n downto i
        x.key[j + 1] = x.key[j]
x.key[i] = y.key[t]
x.n = x.n + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

Explanation

First

- The code works as follow:
  - the element y has $2t$ children ($2t - 1$ keys) but is reduced to t children.
  - For this, the new node z takes the t largest children from y, and z becomes a new child of x.

Explanation

First

- The code works as follow:
  - the element y has $2t$ children $(2t-1$ keys) but is reduced to t children.
  - For this, the new node z takes the t largest children from y, and z becomes a new child of x.

Explanation

First

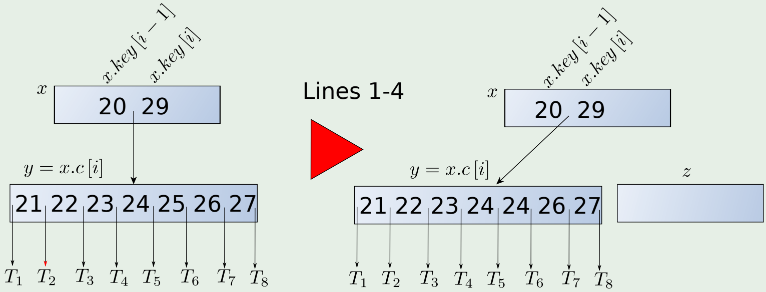
- The code works as follow:
  - the element y has $2t$ children $(2t-1$ keys) but is reduced to t children.
  - For this, the new node z takes the t largest children from y, and z becomes a new child of x.

Detailed Explanation

First

Lines 1-4 creates node z

1. $z =$ **Allocate-Node()**
2. $y = x.c_i$
3. $z.leaf = y.leaf$
4. $z.n = t - 1$

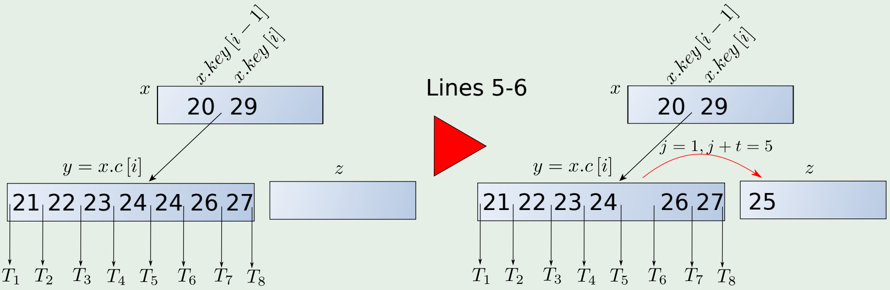


Detailed Explanation

First

Lines 5-6 copies the keys from position $j+1$ in the y node to position j in node z:

5. **for** $j = 1$ and $t - 1$
6. $\quad z.key\,[j] = y.key\,[j+t]$

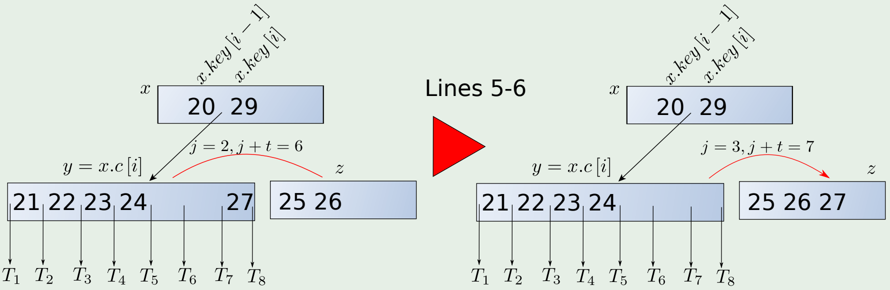


Detailed Explanation

First

Lines 5-6 copies the keys from position $j+1$ in the y node to position j in node z:

5. **for** $j = 1$ and $t-1$
6. $z.key\left[j\right] = y.key\left[j+t\right]$

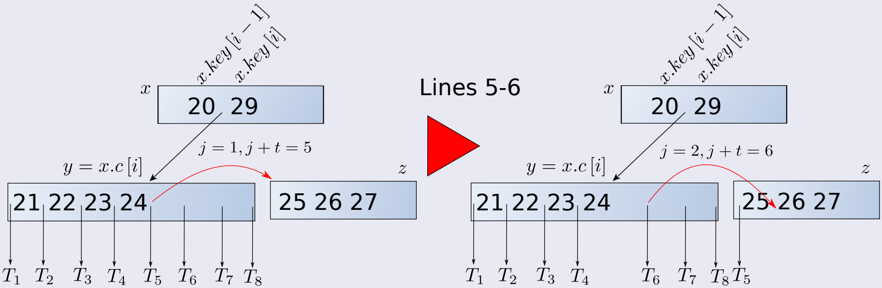


Detailed Explanation

Then

Lines 7-8 are used to copy the children if you are not a leaf

7. **if not** y.leaf
8. **for** $j = 1$ **to** t
9. $z.c[j] = y.c[j+t]$

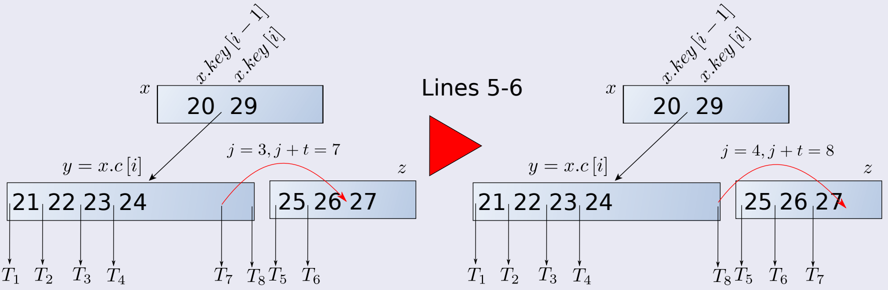


Detailed Explanation

Then

Lines 7-8 are used to copy the children if you are not a leaf

7. **if not** y.leaf
8. **for** $j = 1$ **to** t
9. $z.c[j] = y.c[j+t]$

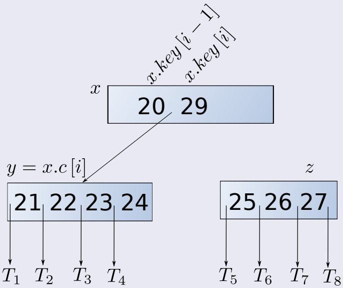


Detailed Explanation

Then

Lines 7-8 are used to copy the children if you are not a leaf

7. **if not** y.leaf
8. **for** $j = 1$ **to** t
9. $z.c\left[j\right] = y.c\left[j+t\right]$



Detailed Explanation

Then

Line 10 adjust the count for y.

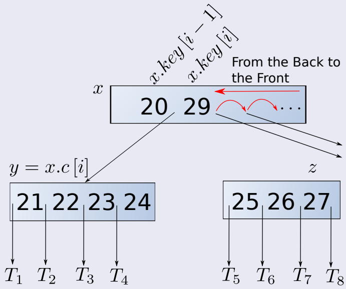
10. $y.n = t - 1$

Detailed Explanation

Then

Line 11-13 make space to the pointer for the z node

11. **for** $j = x.n + 1$ **downto** $i + 1$
12. $\quad x.c[j+1] = x.c[j]$
13. $x.c[i+1] = z$

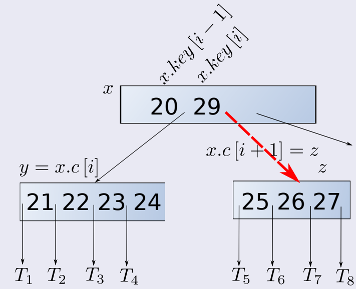


Detailed Explanation

Then

Line 11-13 make space to the pointer for the z node

11. **for** $j = x.n + 1$ **downto** $i + 1$
12. $\quad x.c[j+1] = x.c[j]$
13. $x.c[i+1] = z$



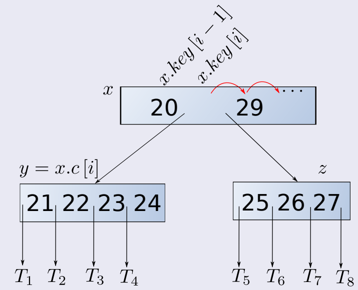
Detailed Explanation

Then

Line 14-15 make space to key from the z node to the node x

14. **for** $j = x.n$ **downto** i

15. $\qquad x.key\,[j+1] = x.key\,[j]$

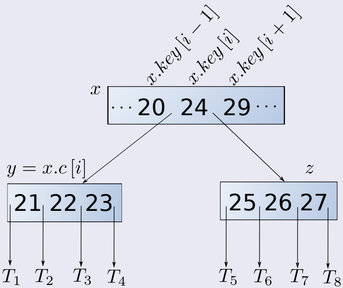


Detailed Explanation

Then

Line 16-17 copy the key to the correct place and increase the counter of x

16. $x.key[i] = y.key[t]$
17. $x.n = x.n + 1$



Detailed Explanation

Then

Line 18-20 Write everything to the hard drive

18. **Disk-Write**(y)
19. **Disk-Write**(z)
20. **Disk-Write**(x)

Cost of Split-Child

Complexity

- $\Theta(t)$ CPU time the for loop to go through the keys
- $O(1)$ disk writes.

Insert

Code

B-Tree-Insert(T, k)

1. $r = T.root$
2. **if** $r.n == 2t - 1$
3. $\quad s =$**Allocate-Node()**
4. $\quad T.root = s$
5. $\quad s.leaf =$**FALSE**
6. $\quad s.n = 0$
7. $\quad s.c\,[1] = r$
8. $\quad$**B-Tree-Split-Childs**$(s, 1)$
9. $\quad$**B-Tree-Insert-Nonfull**(s, k)
10. **else** **B-Tree-Insert-Nonfull**(s, k)

cinvestav

Insert

Code

B-Tree-Insert(T, k)

```
r = T.root
if r.n == 2t − 1
        s = Allocate-Node()
        T.root = s
        s.leaf = FALSE
        s.n = 0
        s.c[1] = r
        B-Tree-Split-Childs(s, 1)
        B-Tree-Insert-Nonfull(s, k)
else B-Tree-Insert-Nonfull(r, k)
```

Insert

Code

B-Tree-Insert(T, k)

```
r = T.root
if r.n == 2t − 1
        s = Allocate-Node()
        T.root = s
        s.leaf = FALSE
        s.n = 0
        s.c[1] = r
        B-Tree-Split-Childs(s, 1)
        B-Tree-Insert-Nonfull(s, k)
else B-Tree-Insert-Nonfull(s, k)
```

Insert

Code

B-Tree-Insert(T, k)

```
r = T.root
if r.n == 2t − 1
        s = Allocate-Node()
        T.root = s
        s.leaf = FALSE
        s.n = 0
        s.c[1] = r
        B-Tree-Split-Childs(s, 1)
        B-Tree-Insert-Nonfull(s, k)
else B-Tree-Insert-Nonfull(s, k)
```

Explanation

First

Insert using the root of T and the key k to be inserted.

Explanation

First

Insert using the root of T and the key k to be inserted.

Second

1. Use a a temporary variable r to look at the root
2. If $r.n == 2t - 1$ Then prepare to split by creating an alternate s father node.
   1. Then Split the node s using Split-Child
   2. Insert using the **Insert-Non full** operation.
3. else Insert using the **Insert-Non full** operation.

Explanation

First

Insert using the root of T and the key k to be inserted.

Second

1. Use a a temporary variable r to look at the root
2. If $r.n == 2t - 1$ Then prepare to split by creating an alternate s father node.
    1. Then Split the node s using Split-Child
    2. Insert using the **Insert-Non full** operation.
3. else Insert using the **Insert-Non full** operation.

Explanation

First

Insert using the root of T and the key k to be inserted.

Second

1. Use a a temporary variable r to look at the root
2. If $r.n == 2t - 1$ Then prepare to split by creating an alternate s father node.
    1. Then Split the node s using Split-Child
    2. Insert using the **Insert-Non full** operation.
3. else Insert using the **Insert-Non full** operation.

Explanation

First

Insert using the root of T and the key k to be inserted.

Second

1. Use a a temporary variable r to look at the root
2. If $r.n == 2t - 1$ Then prepare to split by creating an alternate s father node.
   1. Then Split the node s using Split-Child
   2. Insert using the **Insert-Non full** operation.
3. else Insert using the **Insert-Non full** operation.

Insert-Full

Note

First, modify tree (if necessary) to create room for new key. Then, call Insert-Nonfull()

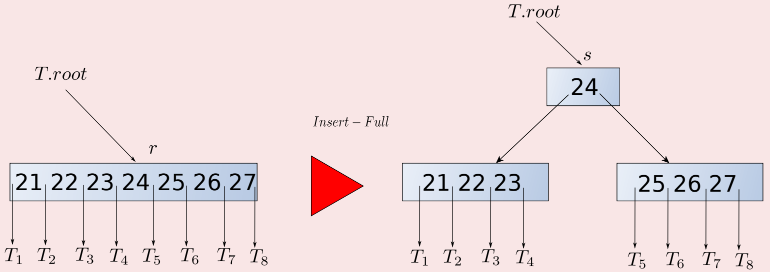
Example

Insert-Full

Note

First, modify tree (if necessary) to create room for new key. Then, call Insert-Nonfull()

Example



Insert-Nonfull

Algorithm

```
B-Tree-Insert-Nonfull(x, k)
i = x.n
if x.leaf
        while i ≥ 1 and k < x.key[i]
                x.key[i + 1] = x.key[i]
                i = i − 1
        x.key[i + 1] = k
        x.n = x.n + 1
        Disk-Write(x)
else while i ≥ 1 and k < x.key[i]
                i = i − 1
        i = i + 1
        Disk-Read(x.c[i])
        if x.c[i].n == 2t − 1
                B-Tree-Split-Child(x, i)
                if k > x.key[i]
                        i = i + 1
        B-Tree-Insert-Nonfull(x.c[i], k)
```

Explanation

Line 1

it gets the rightmost key of the B-Tree

1. $i = x.n$

if $x.leaf == TRUE$

We make space on the key array because we have space for it.

3. **while** $i \geq 1$ **and** $k < x.key[i]$
4. $x.key[i+1] = x.key[i]$
5. $i = i - 1$

Explanation

Line 1

it gets the rightmost key of the B-Tree

1. $i = x.n$

if $x.leaf == TRUE$

We make space on the key array because we have space for it.

3. **while** $i \geq 1$ **and** $k < x.key[i]$
4. $\quad x.key[i+1] = x.key[i]$
5. $\quad i = i - 1$

Explanation

Insert the key with the payload at the correct position and increase the counter of x

6. $x.key\,[i+1] = k$
7. $x.n = x.n + 1$

Write everything to the disk

8. **Disk-Write**(x)

Explanation

Insert the key with the payload at the correct position and increase the counter of x

6. $x.key\left[i+1\right] = k$
7. $x.n = x.n + 1$

Write everything to the disk

8. **Disk-Write**(x)

Explanation

if $x.leaf! = TRUE$

Get into the correct child and bring it from the hard drive

```
else while i ≥ 1 and k < x.key[i]
        i = i − 1
    i = i + 1
    Disk-Read(x.c[i])
```

if the child $x.c[i]$ is full split it

```
    if x.c[i].n == 2t − 1
        B-Tree-Split-Child(x, i)
```

Explanation

if $x.leaf! = TRUE$

Get into the correct child and bring it from the hard drive

```
else while i ≥ 1 and k < x.key[i]
            i = i − 1
      i = i + 1
      Disk-Read(x.c[i])
```

if the child $x.c[i]$ is full split it

```
      if x.c[i].n == 2t − 1
            B-Tree-Split-Child(x, i)
```

Explanation

Now we need to decide

if $k == x.key[i]$

- Then, we take the left child of $x.key[i]$

If not,

- we take the right child of $x.key[i]$

```
        if k > x.key [i]
            i = i + 1
```

Explanation

Now we need to decide

if $k == x.key[i]$

- Then, we take the left child of $x.key[i]$

If not,

- we take the right child of $x.key[i]$

```
if k > x.key[i]
    i = i + 1
```

After that, we insert in a non-full element

```
B-Tree-Insert-Nonfull(x.c[i], k)
```

Explanation

Now we need to decide

if $k == x.key[i]$

- Then, we take the left child of $x.key[i]$

If not,

- we take the right child of $x.key[i]$

```
if k > x.key[i]
    i = i + 1
```

After that, we insert in a non-full element

```
B-Tree-Insert-Nonfull(x.c[i], k)
```

Explanation

Now we need to decide

if $k == x.key[i]$

- Then, we take the left child of $x.key[i]$

If not,

- we take the right child of $x.key[i]$

if $k > x.key[i]$
$i = i + 1$

After that, we insert in a non-full element

B-Tree-Insert-Nonfull$(x.c[i], k)$

Explanation

Now we need to decide

if $k == x.key[i]$

- Then, we take the left child of $x.key[i]$

If not,

- we take the right child of $x.key[i]$

```
if k > x.key[i]
    i = i + 1
```

After that, we insert in a non-full element

```
B-Tree-Insert-Nonfull(x.c[i], k)
```

Cost of Insertion

Worst case

- $\Theta(\log_t n)$ disk writes.
- $\Theta(t \log_t n)$ CPU time.

Example of Constructing a B-Tree by Insertion

Proceed as follows

Suppose we start with an empty B-Tree and keys arrive in the following order:

- 1, 12, 8, 2, 25, 6 ,14, 28, 19, 20, 17, 7, 52, 16, 48, 60, 68, 3, 26, 29, 53, 55, 24, 23, 22, 11.

Example of Constructing a B-Tree by Insertion

Proceed as follows

Suppose we start with an empty B-Tree and keys arrive in the following order:

- 1, 12, 8, 2, 25, 6 ,14, 28, 19, 20, 17, 7, 52, 16, 48, 60, 68, 3, 26, 29, 53, 55, 24, 23, 22, 11.

Something Notable

- We want to build a B-Tree with at most 5 keys. Thus:

$$2t - 1 = 5$$

$$2t = 6$$

$$t = 3$$

Example of Constructing a B-Tree by Insertion

Proceed as follows

Suppose we start with an empty B-Tree and keys arrive in the following order:

- 1, 12, 8, 2, 25, 6 ,14, 28, 19, 20, 17, 7, 52, 16, 48, 60, 68, 3, 26, 29, 53, 55, 24, 23, 22, 11.

Something Notable

- We want to build a B-Tree with at most 5 keys. Thus:

$$
\begin{aligned}
2t - 1 &= 5 \\
2t &= 6 \\
t &= 3
\end{aligned}
$$

Example of Constructing a B-Tree by Insertion

Proceed as follows

Suppose we start with an empty B-Tree and keys arrive in the following order:

- 1, 12, 8, 2, 25, 6 ,14, 28, 19, 20, 17, 7, 52, 16, 48, 60, 68, 3, 26, 29, 53, 55, 24, 23, 22, 11.

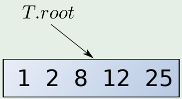
Something Notable

- We want to build a B-Tree with at most 5 keys. Thus:

$$
\begin{aligned}
2t - 1 &= 5 \\
2t &= 6 \\
t &= 3
\end{aligned}
$$

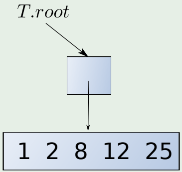
First

We insert the first 5 elements in the root node



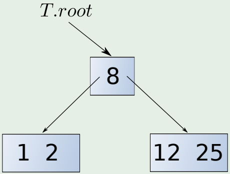
Constructing a B-Tree

Then, we want to insert 6 and for this we split promoting 8



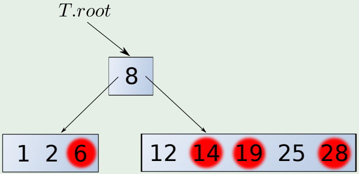
Constructing a B-Tree

Then, we want to insert 6 and for this we split promoting 8



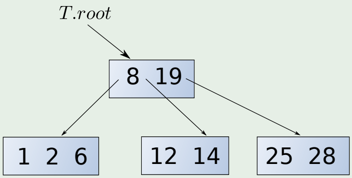
Constructing a B-Tree

6, 14, 28, 19 get added to the leaf nodes



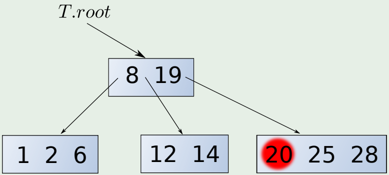
Constructing a B-Tree

Add 20, Split necessary by promoting 19



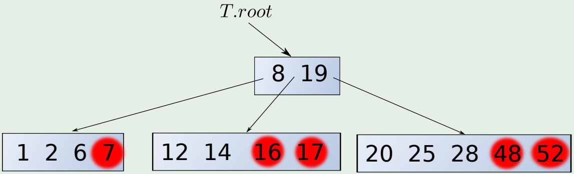
Constructing a B-Tree

Add 20 to the leaf node



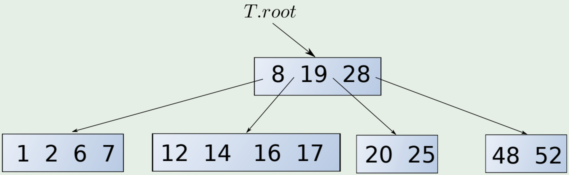
Constructing a B-Tree

Add 17, 7, 52, 16, 48 to the leaf nodes



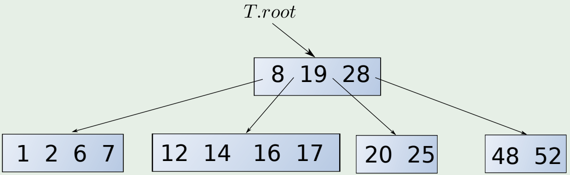
Constructing a B-Tree

Add 60 to a leaf node, it is necessary to split by promoting 28 to the root



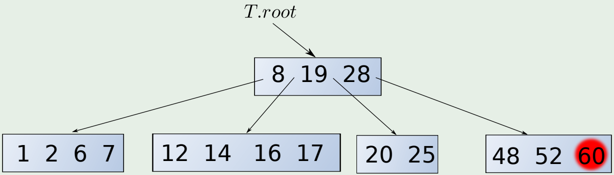
Constructing a B-Tree

Add 60



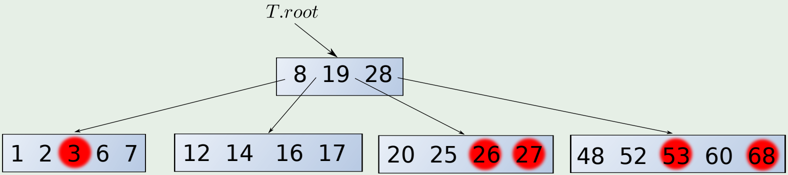
Constructing a B-Tree

Add 68, 3, 26, 27, 53 to the leaf nodes



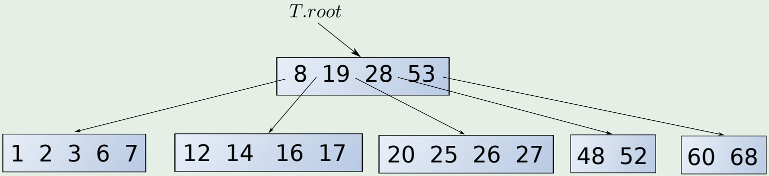
Constructing a B-Tree

Add 68, 3, 26, 27, 53 to the leaf nodes



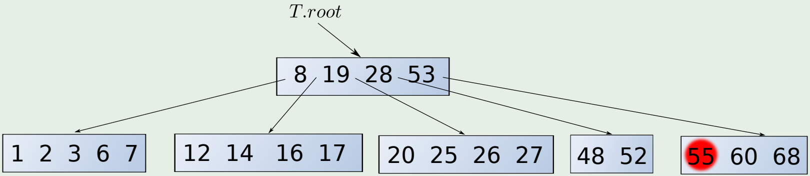
Constructing a B-Tree

Add 55 by splitting a leaf node and promoting 54



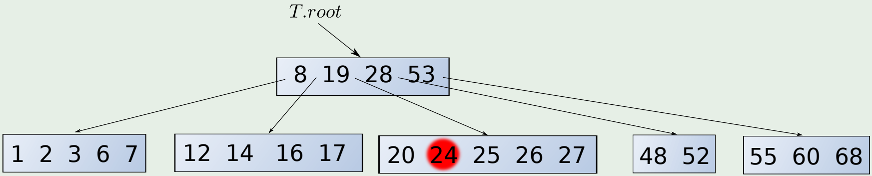
Constructing a B-Tree

Add 55 to the leaf



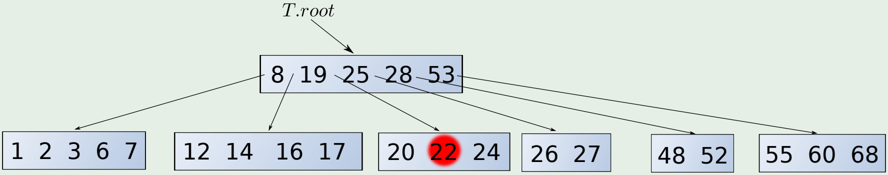
Constructing a B-Tree

Add 24 to the leaf



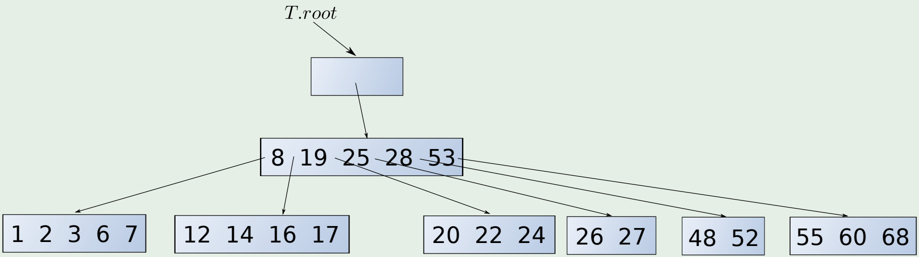
Constructing a B-Tree

Add 22 by splitting a leaf node and promoting 25



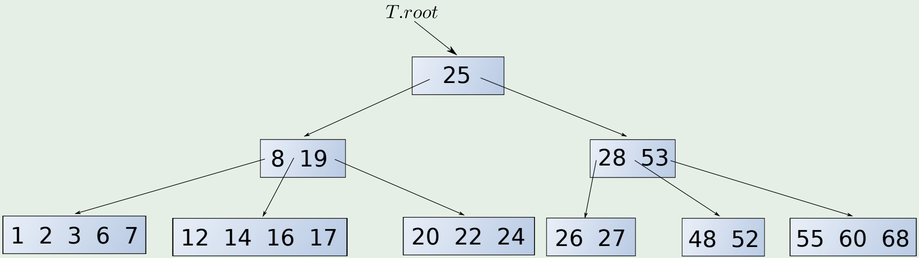
Constructing a B-Tree

Add 11 to the leaf node by adding a empty root node



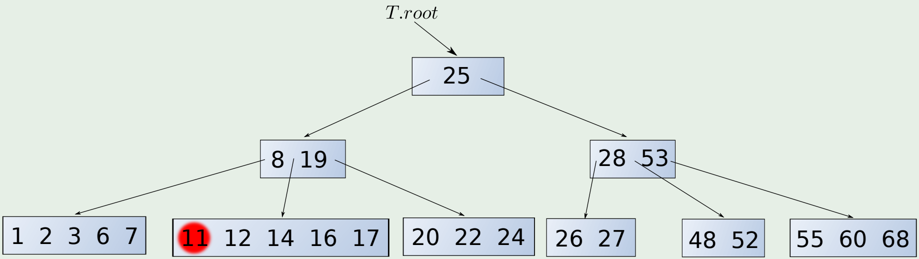
Constructing a B-Tree

Split the old root by promoting 25



Constructing a B-Tree

Add 11 to the leaf



Outline

Deletion

Main idea

Recursively descend the tree.

Ensure

Ensure any non-root node x that is considered for deletion has at least t keys.

Note that

May have to move a key down from parent.

Deletion

Main idea

Recursively descend the tree.

Ensure

Ensure any non-root node x that is considered for deletion has at least t keys.

Note that

May have to move a key down from parent.

Deletion

Main idea

Recursively descend the tree.

Ensure

Ensure any non-root node x that is considered for deletion has at least t keys.

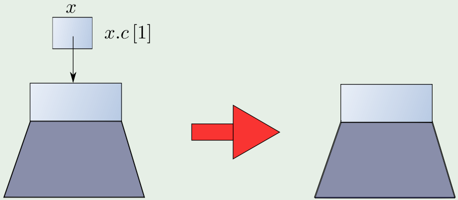
Note that

May have to move a key down from parent.

Deletion Cases

Case 0: You delete the only key at the root ≈ Empty root

Then, you make root's only child the new root:

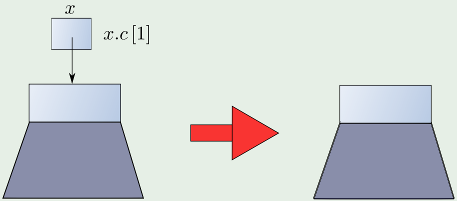


Case 1: k in x and $x.leaf == TRUE$, then delete k from x.

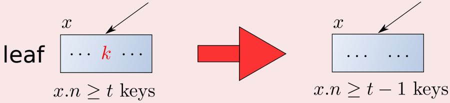
Deletion Cases

Case 0: You delete the only key at the root ≈ Empty root

Then, you make root's only child the new root:

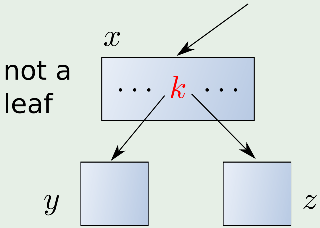


Case 1: k in x and $x.leaf == TRUE$, then delete k from x.



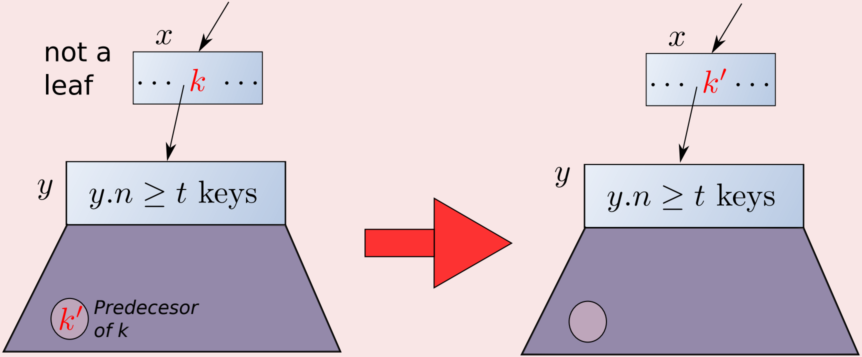
Deletion Cases

Case 2: k in x, x internal



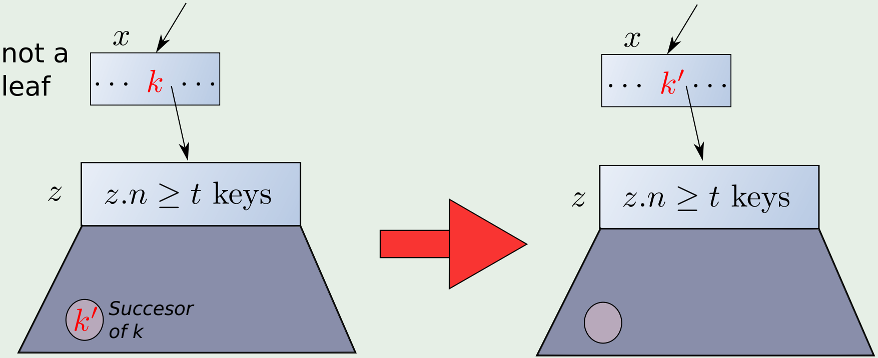
Deletion Cases

Subcase A: 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.



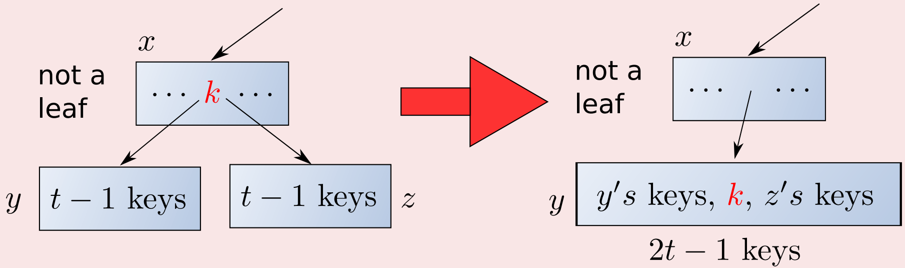
Deletion Cases

Subcase B: z has at least t keys; find successor k' in subtree rooted at z, recursively delete k', replace k by k' in x.



Deletion Cases

Subcase C: y and z both have $t-1$ keys; merge k and z into y, free z, recursively delete k from y.



Deletion cases

Case 3

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

Deletion cases

Case 3

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

Deletion cases

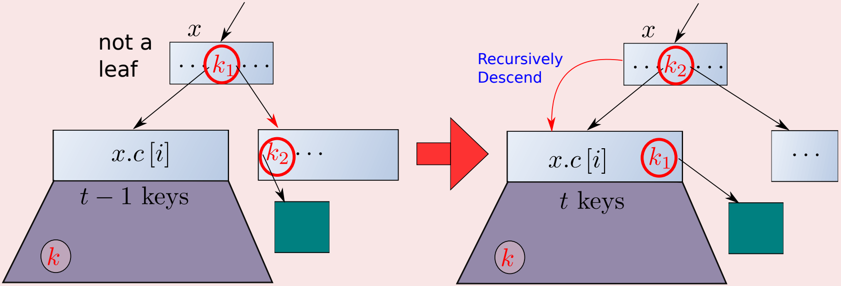
Case 3

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

Case 3.A

Subcase A

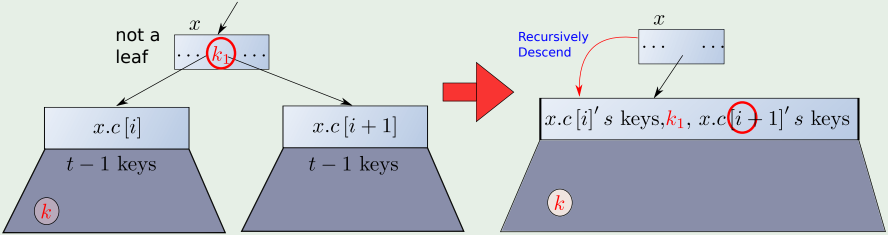
If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.



Case 3.B

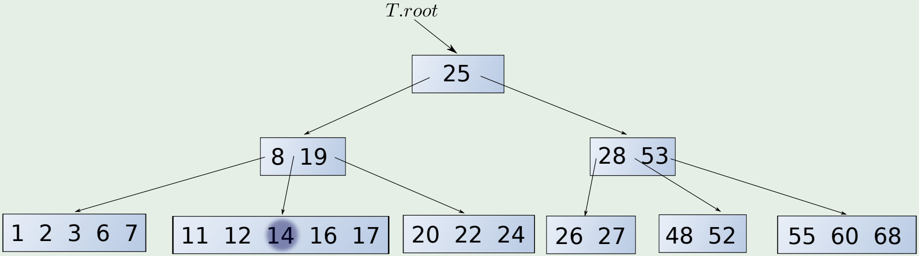
Subcase B

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



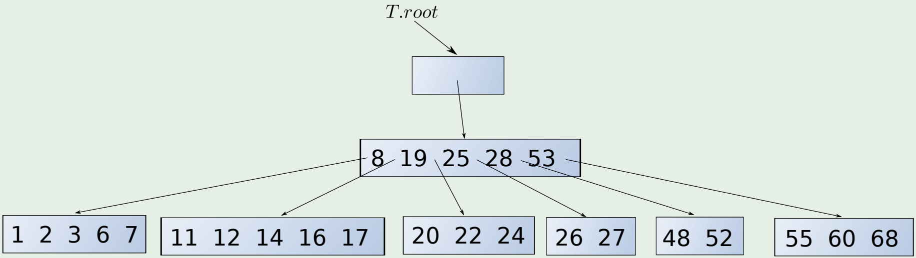
Delete Example

Delete 14 - Case 3.B



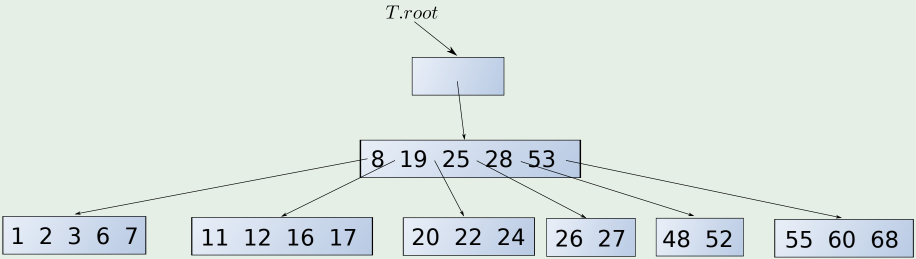
Delete Example

Delete 14 - move 25 down from the root and join the children nodes



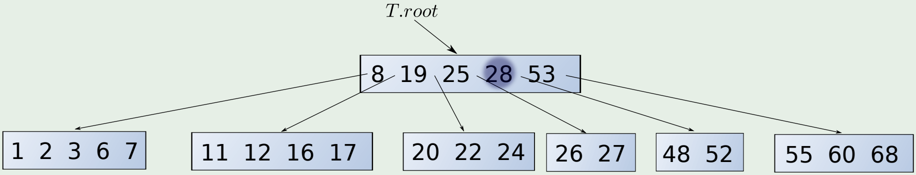
Delete Example

Delete 14



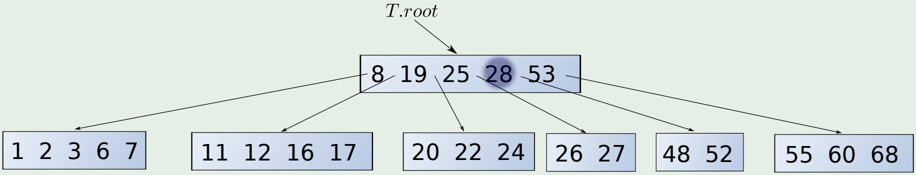
Delete Example

Case 0



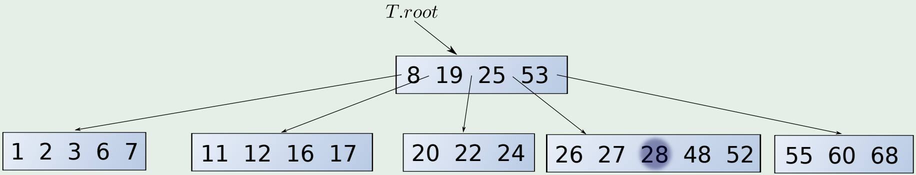
Delete Example

Delete 28 - Case 2.C



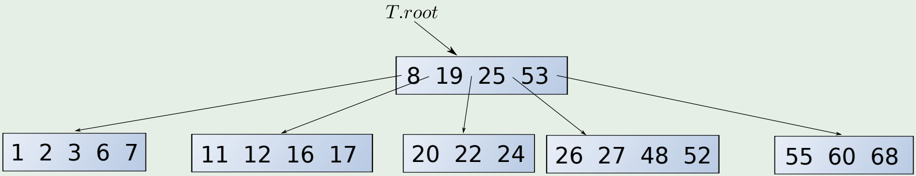
Delete Example

Join the left and right children of 28 and move it down



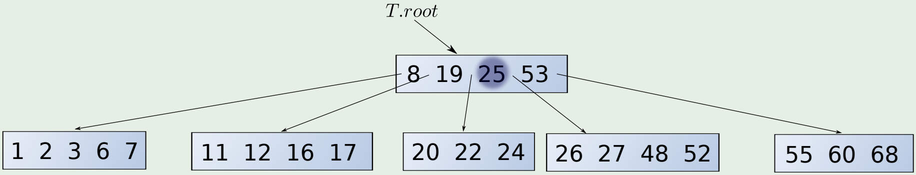
Delete Example

Recursively Delete 28



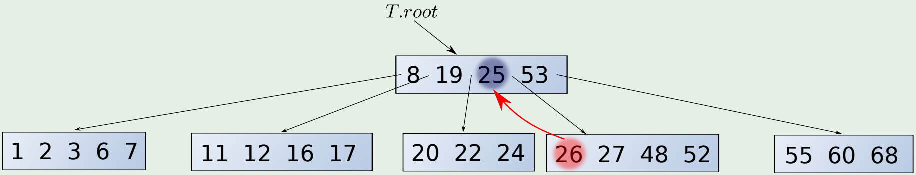
Delete Example

Delete 25 - Case 2.B



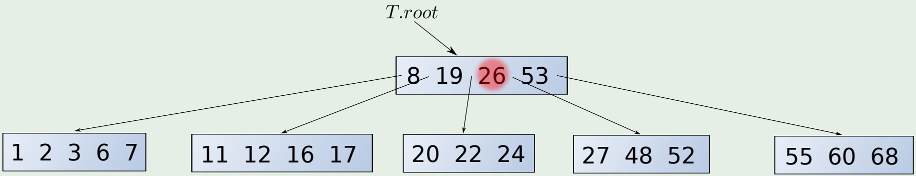
Delete Example

Move 26 to the position of 25



Delete Example

Move 26 to the position of 25



Outline

Reasons for using B-Trees

Justification

When searching tables held on disc, the cost of each disc transfer is high, but does not depend much on the amount of data transferred, especially if consecutive items are transferred.

Reasons for using B-Trees

Justification

When searching tables held on disc, the cost of each disc transfer is high, but does not depend much on the amount of data transferred, especially if consecutive items are transferred.

Example

- If we use a B-Tree of order 101, say, we can transfer each node in one disc read operation.
- A B-Tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory).

Reasons for using B-Trees

Justification

When searching tables held on disc, the cost of each disc transfer is high, but does not depend much on the amount of data transferred, especially if consecutive items are transferred.

Example

- If we use a B-Tree of order 101, say, we can transfer each node in one disc read operation.
- A B-Tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory).

Comparing trees

Binary trees

- They can become unbalanced and lose their good time complexity (big O).
- AVL trees are strict binary trees that overcome the balance problem.
- Heaps remain balanced, but only prioritize (not order) the keys.

Comparing trees

Binary trees

- They can become unbalanced and lose their good time complexity (big O).
- AVL trees are strict binary trees that overcome the balance problem.
- Heaps remain balanced, but only prioritize (not order) the keys.

Multi-way trees

- B-Trees can be m-way, they have any even number of children.
- The 2-3 (or 3 way) approximates a permanently balanced binary tree.

Comparing trees

Binary trees

- They can become unbalanced and lose their good time complexity (big O).
- AVL trees are strict binary trees that overcome the balance problem.
- Heaps remain balanced, but only prioritize (not order) the keys.

Multi-way trees

- B-Trees can be m-way, they have any even number of children.
- The 2-3 (or 3 way) approximates a permanently balanced binary tree.

Comparing trees

Binary trees

- They can become unbalanced and lose their good time complexity (big O).
- AVL trees are strict binary trees that overcome the balance problem.
- Heaps remain balanced, but only prioritize (not order) the keys.

Multi-way trees

- B-Trees can be m-way, they have any even number of children.
- The 2-3 (or 3 way) approximates a permanently balanced binary tree.

Comparing trees

Binary trees

- They can become unbalanced and lose their good time complexity (big O).
- AVL trees are strict binary trees that overcome the balance problem.
- Heaps remain balanced, but only prioritize (not order) the keys.

Multi-way trees

- B-Trees can be m-way, they have any even number of children.
- The 2-3 (or 3 way) approximates a permanently balanced binary tree.

Outline

Extending the B-Tree Structure: B+ Trees

B+ Tree

A B+ Tree is like a B-tree except that the interior and leaf nodes have a different structure.

Actually

A B+ tree can be viewed as a B-tree in which each node contains only keys and pointers to the children.

Finally

At leaves level you have the real data items(They could be pointers to specific data).

Node

This allows to pack more information in each node.

Extending the B-Tree Structure: B+ Trees

B+ Tree

A B+ Tree is like a B-tree except that the interior and leaf nodes have a different structure.

Actually

A B+ tree can be viewed as a B-tree in which each node contains only keys and pointers to the children.

Finally

At leaves level you have the real data items(They could be pointers to specific data).

Node

This allows to pack more information in each node.

Extending the B-Tree Structure: B+ Trees

B+ Tree

A B+ Tree is like a B-tree except that the interior and leaf nodes have a different structure.

Actually

A B+ tree can be viewed as a B-tree in which each node contains only keys and pointers to the children.

Finally

At leaves level you have the real data items(They could be pointers to specific data).

Node

This allows to pack more information in each node.

Extending the B-Tree Structure: B+ Trees

B+ Tree

A B+ Tree is like a B-tree except that the interior and leaf nodes have a different structure.

Actually

A B+ tree can be viewed as a B-tree in which each node contains only keys and pointers to the children.

Finally

At leaves level you have the real data items(They could be pointers to specific data).

Node

This allows to pack more information in each node.

In the paper

Something Notable

"Modularizing B+-Trees: Three-Level B+-Trees Work Fine" by Shigero Sasaki and Takuya Araki from NEC

NEC

NEC Corporation (Nippon Denki Kabushiki Gaisha) is a Japanese multinational provider of information technology (IT) services and products, with its headquarters in Minato, Tokyo, Japan.NEC provides information technology (IT) and network solutions to business enterprises, communications services providers and to government agencies.

In the paper

Something Notable

"Modularizing B+-Trees: Three-Level B+-Trees Work Fine" by Shigero Sasaki and Takuya Araki from NEC

NEC

NEC Corporation (Nippon Denki Kabushiki Gaisha) is a Japanese multinational provider of information technology (IT) services and products, with its headquarters in Minato, Tokyo, Japan.NEC provides information technology (IT) and network solutions to business enterprises, communications services providers and to government agencies.

Outline

Exercises

You can try the following ones

1. 18.1-3
2. 18.1-4
3. 18.2-3
4. 18.2-5
5. 18.2-4
6. 18.2-6
7. 18.2-7
8. 18.3-1