Algorithms and Data Structures (II)

Gabriel Istrate

May 27, 2020

Where are we:

The last topics of this course

- Data Structures for external memory: B-Trees.
- A bit of data compression
- Tries.

Next: B-Trees

Outline:

- Search in secondary storage
- B-Trees
  - properties
  - search
  - insertion

Complexity Model

- 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”

Complexity Model

- 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

Idea

Idea

- 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

- 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

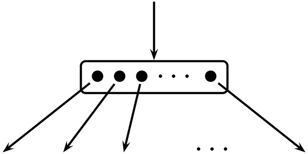
- **Idea:** store several keys and pointers to children nodes in a single node

- 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

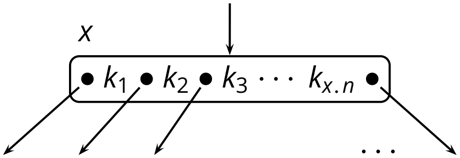
- In a balanced *binary* tree, n keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

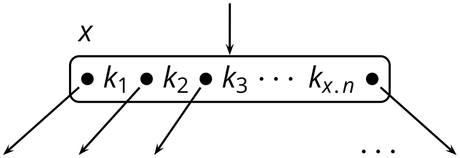
- **Idea:** store several keys and pointers to children nodes in a single node
  - in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - in practice d can be as high as a few thousands



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

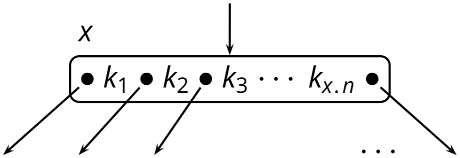
Definition of a B-Tree





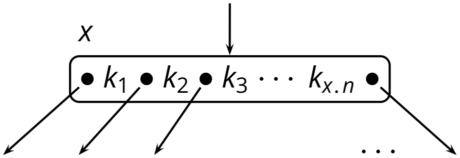
- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node

Definition of a B-Tree

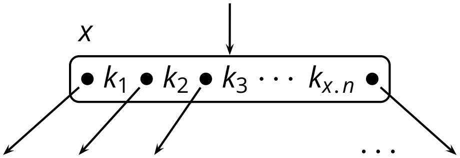


- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order

Definition of a B-Tree

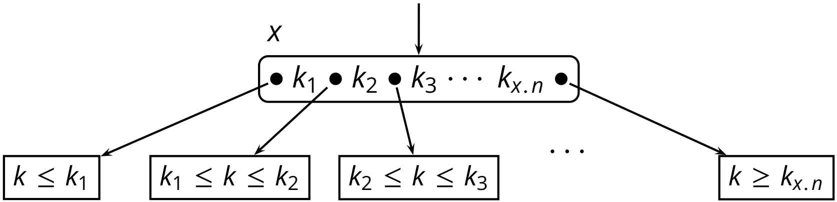


- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*

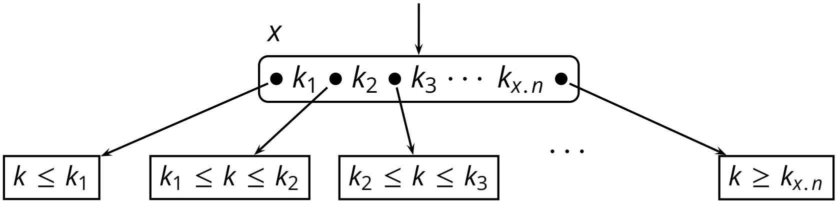


- Every node x has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if x is a *leaf node* or FALSE if x is an *internal node*
  - $x.c[1],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

Definition of a B-Tree (2)

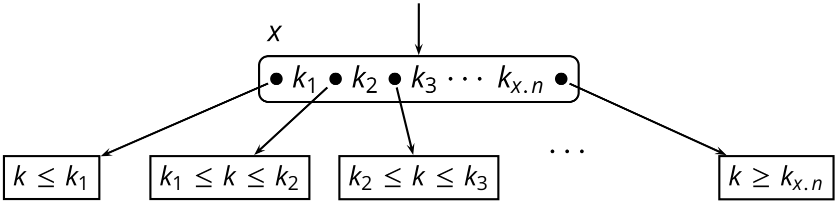


Definition of a B-Tree (2)



- 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, $x.key[2] \leq k \leq x.key[3]$

  $\ldots$

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

Defining properties of a B-Tree

Defining properties of a B-Tree

- ***All leaves have the same depth***

Defining properties of a B-Tree

- ***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
- ***Nonleaf nodes: one more child than values, arranged like in a BST.***

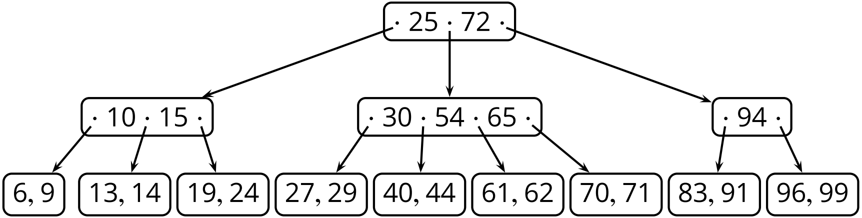
Defining properties of a B-Tree

- ***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

- ***Nonleaf nodes: one more child than values, arranged like in a BST.***

Example



· 25 · 72 ·
· 10 · 15 ·
· 30 · 54 · 65 ·
· 94 ·
6, 9
13, 14
19, 24
27, 29
40, 44
61, 62
70, 71
83, 91
96, 99

Search in B-Trees

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)
if x.leaf
    return NIL
else DISK-READ(x.c[i])
    return B-TREE-SEARCH(x.c[i], k)
```

Height of a B-Tree

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

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

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

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

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

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

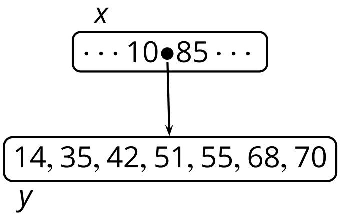
Proof:

- $n \geq 1$, 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

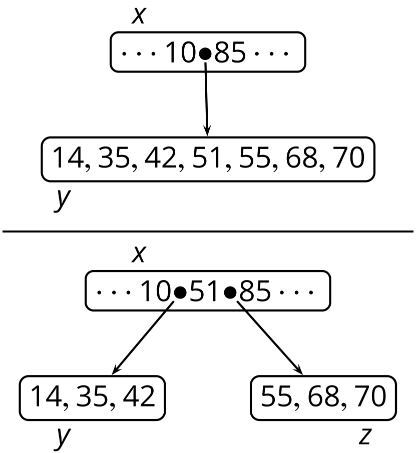
$$n \geq 1 + 2(t^h - 1)$$

Splitting

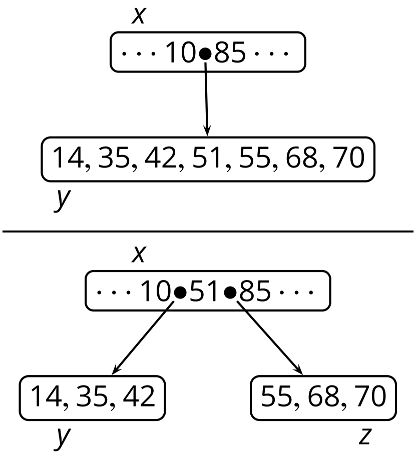
Splitting



Splitting



Splitting




```
B-TREE-SPLIT-CHILD(x, i, y)
  z = ALLOCATE-NODE()
  z.leaf = y.leaf
  z.n = t − 1
  for j = 1 to 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]
  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)
```

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)
z = ALLOCATE-NODE()
z.leaf = y.leaf
z.n = t − 1
for j = 1 to t − 1
        x.key[j] = x.key[j + t]
if not x.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]
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)
```

Insertion Under Non-Full Node

Insertion Under Non-Full Node

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n                                   // assume x is not full
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                 // child x.c[i] is full
        B-TREE-SPLIT-CHILD(x, i, x.c[i])
        if k > x.key[i]
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c[i], k)
```

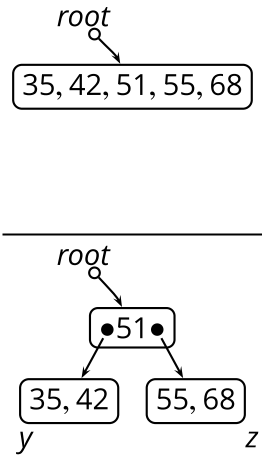
Insertion Procedure

Insertion Procedure

```
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-CHILD(s, 1, r)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Insertion Procedure

```
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-CHILD(s, 1, r)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```



Complexity of Insertion

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

Complexity of Insertion

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

Complexity of Insertion

- 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

Complexity of Insertion

- 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

Deletion

Mirror image of insertion.

- In insertion: key always goes to leaf. Before inserting new key check if node to insert is full
- If so: first split node, to make it non-full.
- : Deletion: want to delete from leaf. But key may not be in leaf.
- Before deleting key check if node to delete from is minimal ($t - 1$ keys).

Deletion

- Case 1: key is in a leaf, delete key.
- Case 2: key is **not** in a leaf. Then its predecessor/successor are in leaf. Delete key, promote pred/succ.
- Cases 1/2 may cause leaf node to become defficient (too few keys). Have to make it nonminimal.
- Look at the immediately adjacent siblings of this node. Several cases:
  - If there is a non-minimal sibling, then take a key/child pointer from that sibling to the parent, and one key/child from parent to defficient leaf.
  - If both siblings are minimal: merge node with one sibling (doesn’t matter which) and one node from parent. If this makes the parent have too few nodes repeat recursively.