Scuola universitaria professionale
della Svizzera italiana
SUPSI

Università
della
Svizzera
italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures

B Trees

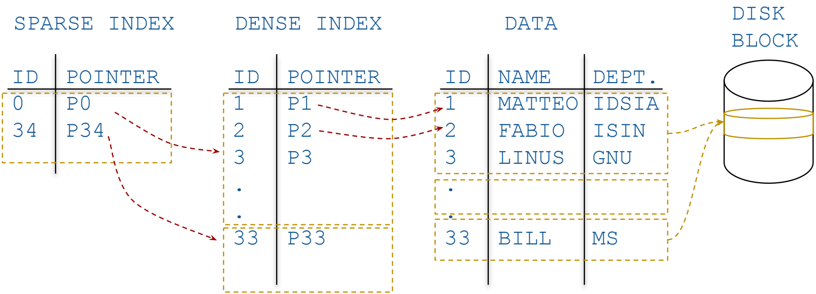
Matteo Salani
matteo.salani@idsia.ch

Indexing and Databases

The need of indexing:

- In databases, the amount of data to be stored is very high.
- Data cannot be stored in the main memory, hence it is commonly stored in the disk.
- Data access from the disk is much slower compared to the main memory access.
- Data is usually accessed in the form of blocks.

Indexing Example



Goal of multi-level indexing:

- Reduce the number of blocks to process to access data
- Challenge: levels must grow and shrink dinamically

B-Trees

- B-trees are balanced search trees designed to work with blocks of data.
- Many database systems use B-trees, or variants, to store information.
- B-tree nodes may have many children (from a few to thousands)
- The height of a B-tree can be considerably less than that of a BST

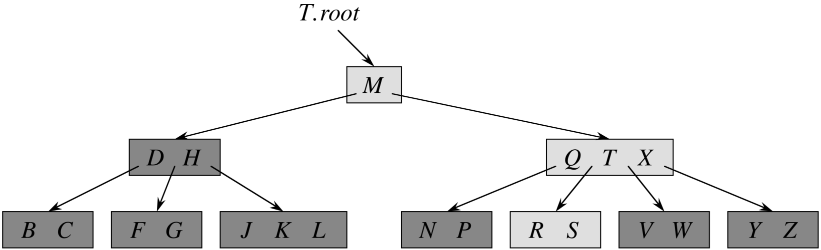


Figure: Example of a B-Tree

B-Trees

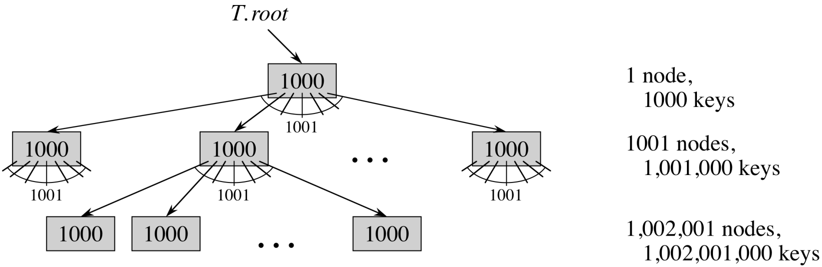


Figure: Example of a B-Tree with over one billion keys

Definition of B-Trees

A B-tree T is a rooted tree (whose root is $T.root$), let x be any node:

1. $x.n$ is the number of keys stored in node x
2. keys $x.k_i$ $i \in \{1 \dots x.n\}$ are stored in non decreasing order ($x.k_1 \leq x.k_2 \leq \cdots \leq x.k_{x.n}$)
3. Each node has $n+1$ pointers $x.c_i$ $i \in \{1 \dots x.n+1\}$ (when x is a leaf, pointers have no meaning)
4. Keys separate the range of keys stored in subtrees $k_{11} \dots k_{1n_1} \leq x.k_1 \leq k_{21} \dots k_{2n_2} \leq x.k_2 \leq \cdots \leq x.k_{x.n} \leq k_{n1} \dots k_{nn_n}$
5. All leaves have the same depth

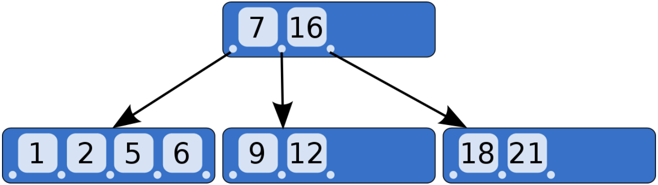


Figure: Example of a B-Tree

Minimum degree or Order of B-Trees

Nodes have **lower** and **upper** bounds on the number of keys they can contain. Let t be the **minimum-degree** or **order** of a B-tree.

1. Every node (except the root) must have at least $t-1$ keys (thus at least t children).
2. Every node may contain at most $2t-1$ keys (thus at most $2t$ children).

Height of a B-Trees

Theorem If $N \geq 1$, the height h of a B-Tree T of degree t satisfies:

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

Proof The root has at least 1 key, the others nodes have at least $t-1$ keys. T has 2 nodes at level 1, $2 \cdot t$ nodes at level 2, $2 \cdot t^2$ nodes at level 3, $2 \cdot t^{h-1}$ nodes at level h,

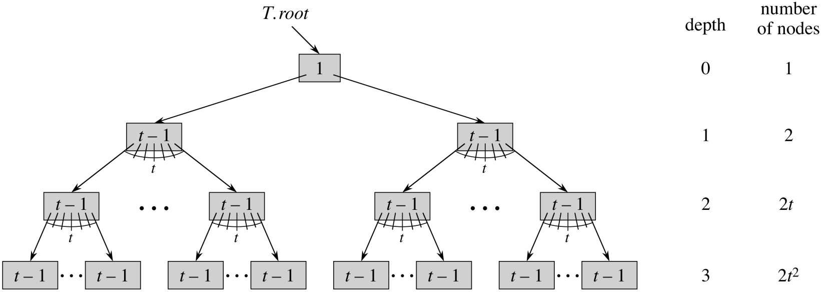


Figure: Minimum number of nodes

Height of a B-Trees

Number of nodes

$$1 + 2 + 2t + 2t^2 + \cdots + 2t^{h-1} = 1 + \sum_{i=1}^{h} 2 \cdot t^{i-1}$$

Number of keys

$$\begin{aligned} N \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2 \cdot t^{i-1} \quad &= 1 + 2 \cdot (t-1) \cdot \sum_{i=0}^{h-1} t^i \\ = 1 + 2 \cdot (t-1) \left(\frac{1-t^h}{1-t} \right) &= 1 + 2 \cdot (t-1) \left(\frac{t^h - 1}{t-1} \right) \\ = 1 + 2 \cdot (t^h - 1) \qquad &= 2 \cdot t^h - 1 \end{aligned}$$

$$N \geq 2 \cdot t^h - 1 \Rightarrow t^h \leq (N+1)/2 \Rightarrow \log_t t^h \leq \log_t (N+1)/2$$

$$h \leq \log_t (N+1)/2$$

Searching in a B-Tree

```
def create(T):
    x = allocateNode()
    x.n = 0
    T.root = x
```

Listing 1: Create method

```
def search(x, key):

    # Similar to BST
    i = 1
    while i <= x.n and key > x.key[i]:
      i = i + 1
    if i <= x.n and key == x.key[i]:
      return x,i
    else if isLeaf(x):
      return NULL
    else
      search(x.c[i], key)
```

Listing 2: Search method

Inserting in a B-Tree

Inserting keys in a B-Tree is more complex than in BST.

- Keys are inserted in **existing** leaf nodes
- If the leaf node is **full** the node is **split** around the median key
- The median key moves to the parent node

```
def split(x, i):
  z = allocateNode()
  y = x.c[i]
  z.n = t-1
  for j = 1 to t-1:
    z.key[j] = y.key[j+t]
  if not isLeaf(y)
    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
```

Listing 3: Split child method

Split Example

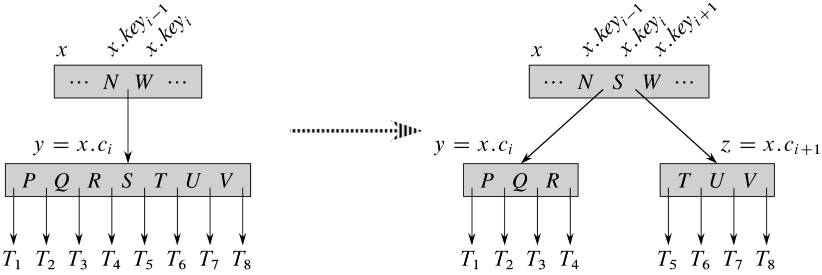


Figure: Split operation, median element "S" goes up one level

Inserting in a B-Tree

We insert a key k into a B-tree T of height h. It requires $O(h)$ block accesses and $O(t \cdot h) = O(t \cdot log_t n)$

```
def insert(T, k):
  r = T.root
  # Case split root
  if r.n == 2t-1
    s = allocateNode()
    T.root = s
    s.n = 0
    s.c[1] = r
    split(s,1)
    insert_nonfull(s, k)
  else
    insert_nonfull(r, k)

```

Listing 4: Insert method

B-Tree growing

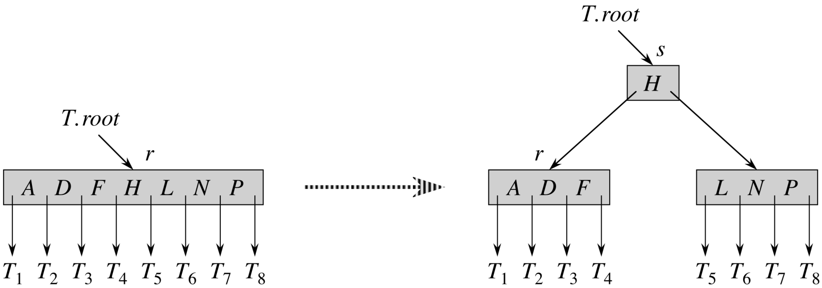


Figure: A B-Tree grows from the top, splitting the root

Inserting in a B-Tree

We insert a key k into a B-tree T of height h. It requires $O(h)$ block accesses and $O(t \cdot h) = O(t \cdot log_t n)$

```
def insert_nonfull(x, k):
  i = x.n
  if isLeaf(x)
    while i >= 1 and k < k.key[i]
      x.key[i+1] = x.key[i]
      i = i-1
    x.key[i+1] = k
    x.n = x.n+1
  else
    while i >= 1 and k < x.key[i]
      i = i-1
    i = i+1
    if x.c[i].n == 2t-1
      split(x,i)
      if k > x.key[i]
        i = i+1
    insert_nonfull(x.c[i], k)

```

Listing 5: Insert non full method

B-Tree example - Order 3

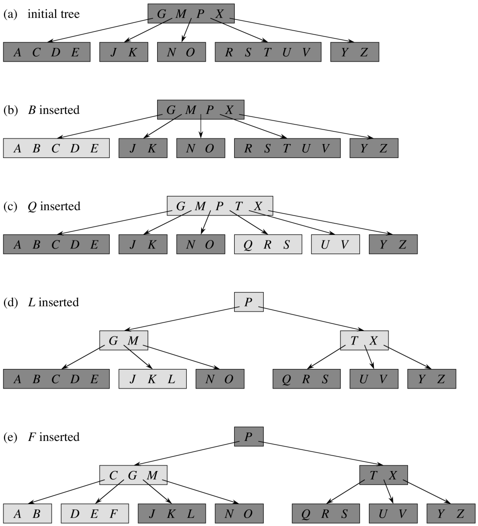


Figure: Sequence of insertion, B, Q, L, F

Deleting in a B-Tree

- We can delete a key from any node
- We may have to rearrange the node's children.
- We must respect B-Tree properties

Deleting in a B-Tree

Sketch of the algorithm, not a real pseudo come

```
def delete(x, k):
  if k in x and isLeaf(x) # case 1
    deleteElement(x, k)
  if k in x and not isLeaf(x)
    y = findPred(k,x) # Return the pred. node of k
    if y.n >= t # case 2.a
      # return the pred. key of k
      k1 = findPredKey(k,y)
      swap(k,k1)
      delete(y, k)
    else
      z = findNext(k,x) # Return the next node of k
      if z.n >= t # case 2.b
        # return the next key of k
        k1 = findNextKey(k,z)
        swap(k,k1)
        delete(z, k)
      else  # case 2.c
        merge(y, k, z)
        freeze(z)
        delete(k, y)

```

Listing 6: Delete method

Deleting in a B-Tree

Sketch of pseudo code (continued)

```
else # k is not in x
  # find the subtree containig k
  i = findChild(x, k)
  if x.c[i].n == t-1
    if i > 1 and x.c[i-1].n >= t # case 3.a
      exchange(x.c[i-1], x)
      delete(x.c[i], k)
    else if i < x.n and x.c[i+1].n >= t # case 3.a
      exchange(x.c[i+1], x)
      delete(x.c[i], k)
    else # case 3.b, index i must be checked
      merge(x.c[i-1], x.key[i-1], x.c[i])
      or merge(x.c[i], x.key[i], x.c[i+1])
      delete(x, k)
      # Take care of root node if it gets empty
  else
    delete(x.c[i], k)

```

Listing 7: Delete method

B-Tree example - Order 3

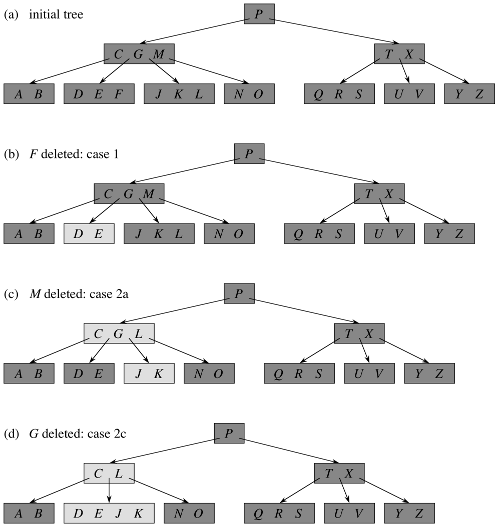


Figure: Deletion of F, M, G

B-Tree example - Order 3

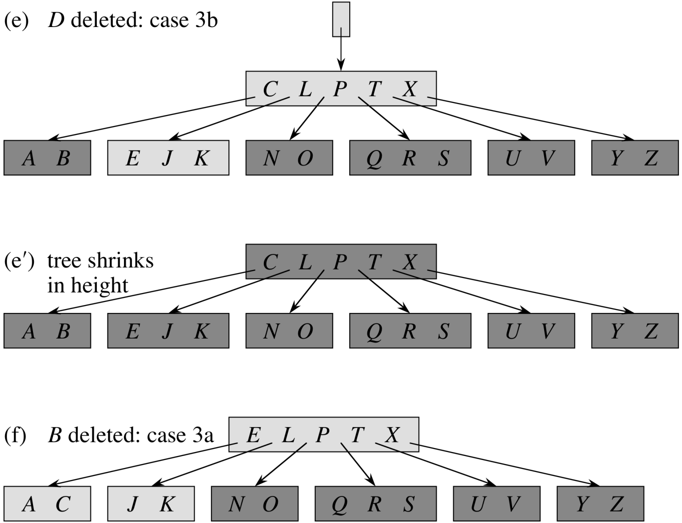


Figure: Deletion of D, B