Database Structures

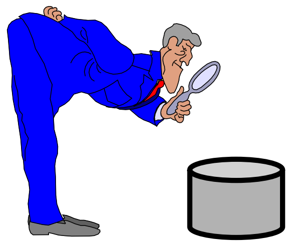
Krzysztof Goczyła

Department of Software Engineering

Faculty of Electronics, Telecommunications and Informatics

Gdańsk University of Technology

krissun@pg.edu.pl

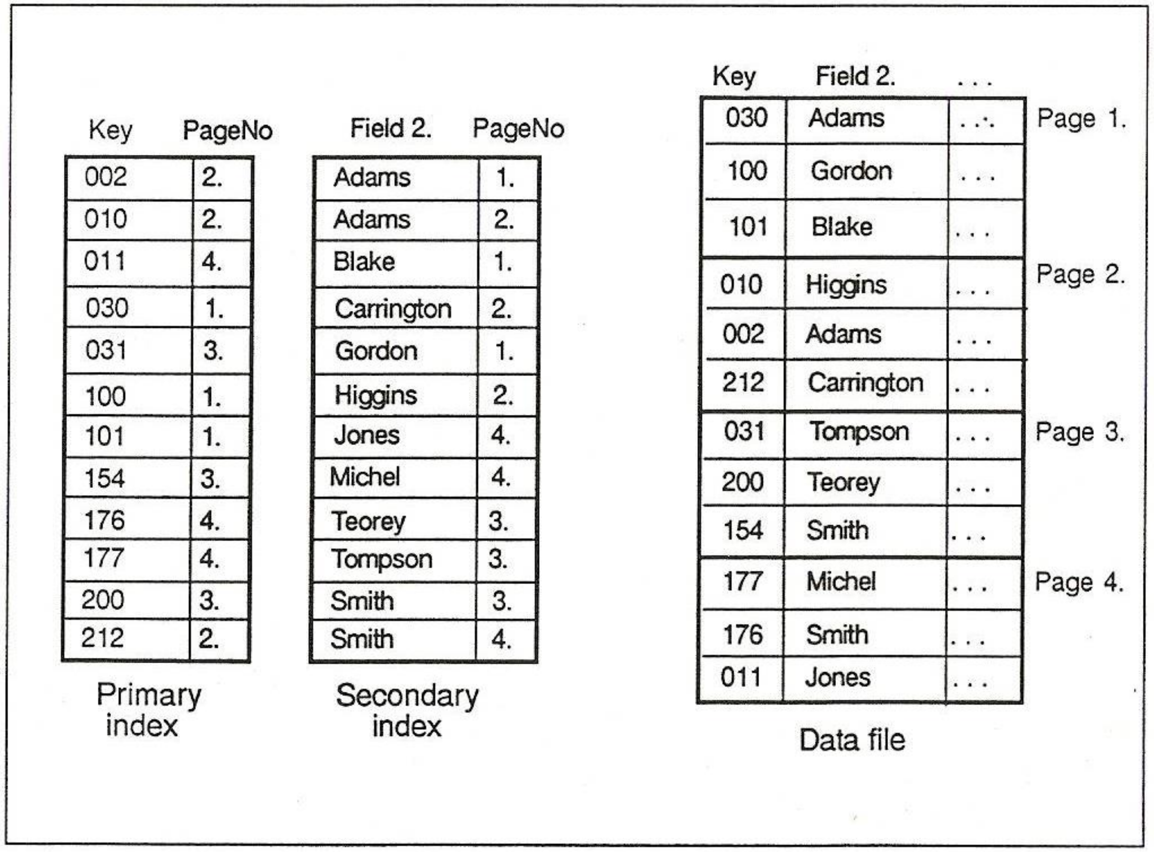


Part II

Index files

In an index file, each record has a corresponding position in the index (**dense index**). Many fields can be indexed, not just the primary key field. The index on the primary key is the **primary index**. An index placed on a different field (fields) is an auxiliary index (**secondary index**). Such an index may contain duplicate keys.

There is no need for an overflow area, because inserting a new record into the file is immediately reflected in the new item in the index.



Key	PageNo
002	2.
010	2.
011	4.
030	1.
031	3.
100	1.
101	1.
154	3.
176	4.
177	4.
200	3.
212	2.

Primary index

Field 2.	PageNo
Adams	1.
Adams	2.
Blake	1.
Carrington	2.
Gordon	1.
Higgins	2.
Jones	4.
Michel	4.
Teorey	3.
Tompson	3.
Smith	3.
Smith	4.

Secondary index

Key	Field 2.	. . .	
030	Adams	. . .	Page 1.
100	Gordon	. . .	
101	Blake	. . .	
010	Higgins	. . .	Page 2.
002	Adams	. . .	
212	Carrington	. . .	
031	Tompson	. . .	Page 3.
200	Teorey	. . .	
154	Smith	. . .	
177	Michel	. . .	Page 4.
176	Smith	. . .	
011	Jones	. . .	

Data file

In order to ensure effective random or sequential access to the file, the index has to be sorted in some way (in the figure on the left, each index is a sequential, static file).

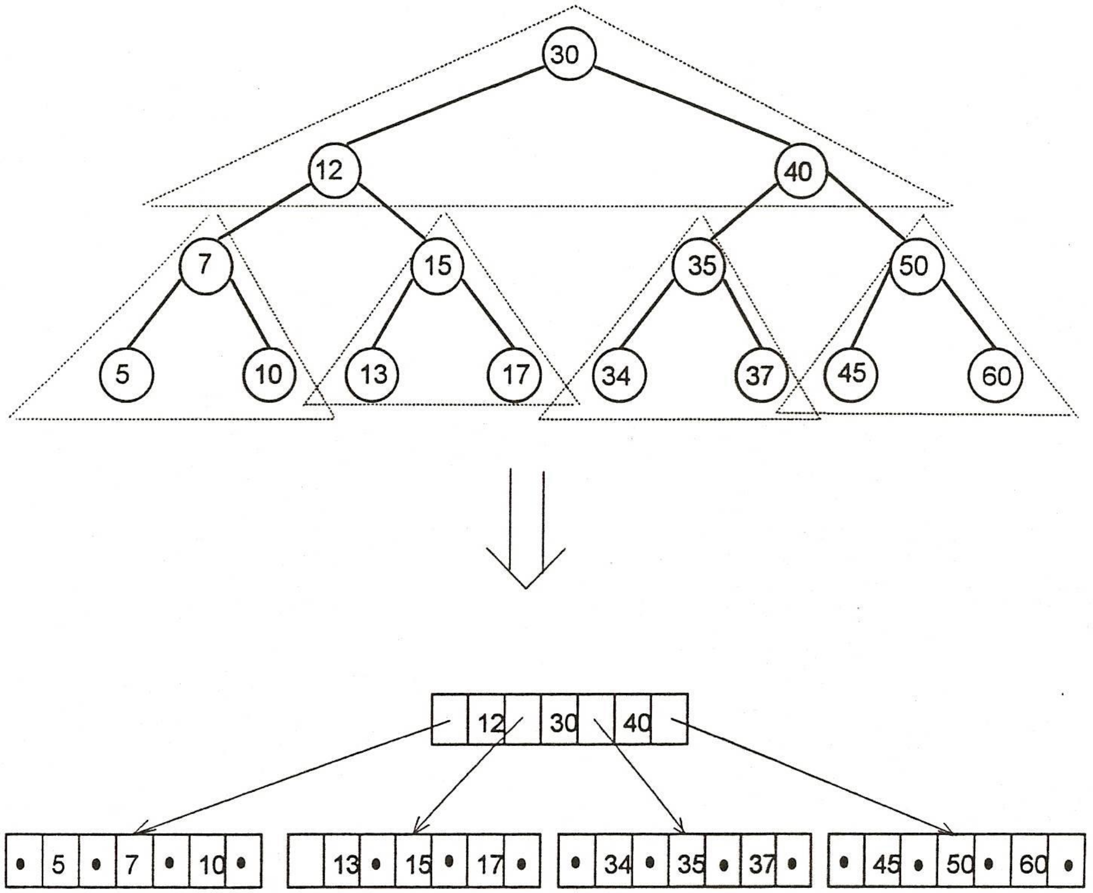
In a dynamic environment (with inserts and updates) index structures that allow modifications must be used.

Trees are such dynamic structures.

B-trees

Idea:

Adaptation of the binary search tree to the specifics of reading/writing in blocks (Bayer and McCreight).



In figure on the left:

In the binary search tree (top), the maximum number of accesses for random reads is 4.

In the B-tree (bottom), the maximum number of disk accesses for random reads is 2.

B-tree - Definition

Definition:

Let $h \geq 0$, $d \geq 1$. B-tree with **height** h and **order** d is an ordered tree that is either empty ($h = 0$), or satisfies the following conditions:

1. All leaves are at the same level equal to h;
2. Each page contains at most $2d$ keys;
3. Each page except the root page contains at least d keys (the root may contain 1 key);
4. If a non-leaf page has m keys, then it has $m+1$ child pages.

Comment:

Condition 1. ensures the tree balancing.

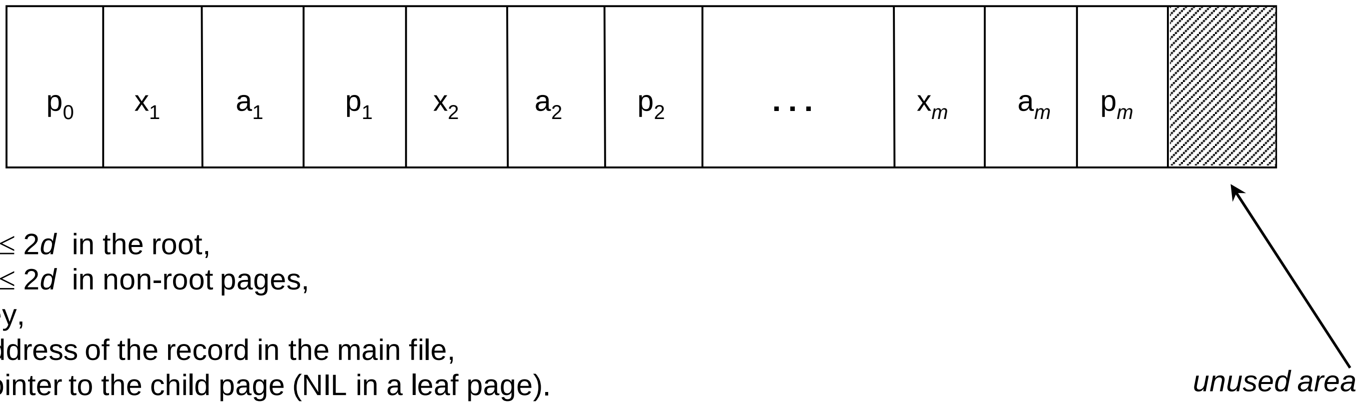
Conditions 2. and 3. guarantee min. 50% of filling level of tree pages, except root.

Condition 4. determines the degree of tree branching.

B-tree – Structure

B-tree records are pairs (*key*, *address*). Between the records, there are pointers to corresponding lower-level pages (if any).

The structure of a page in a B-tree:



$1 \le m \le 2d$ in the root,
$d \le m \le 2d$ in non-root pages,
x_i - key,
a_i - address of the record in the main file,
p_i – pointer to the child page (NIL in a leaf page).

On each page:

$x_1 < x_2 < \ldots < x_m$
p_0 – pointer to the page with keys $< x_1$
p_m - pointer to the page with keys $> x_m$
p_i - pointer to the page with keys that are $> x_i$ and $< x_{i+1}$, $i = 1,2,\ldots,m-1$.

Remark:

To improve the algorithms running on B-trees, the page may have a header containing additional information, e.g.:

- the current number of records on the page,
- pointer to the parent page,
- …

B-tree – Performance

Consider a non-empty B-tree (h, d), $h > 0$. **Minimal** number of keys $N_{\min}$ is:

$$N_{\min} = 1 + 2d + 2(d+1)d + 2(d+1)^2 d + ... + 2(d+1)^{h-2} d = 1 + 2((d+1)^{h-1} - 1)$$

(1 key in the root page and d keys on other pages).

Maximal number of keys $N_{\max}$ is:

$$N_{\max} = 2d + 2d(2d+1) + 2d(2d+1)^2 + ... + 2d(2d+1)^{h-1} = (2d+1)^h - 1$$

(each page is filled with 2d keys).

$N_{\min}$ corresponds to the highest tree with N keys, $N_{\max}$ corresponds to the lowest tree with N keys.
So, the height ***h*** of a **B-tree *h*** with N keys satisfies the following condition (we derive h from the two above formulas):

$$\log_{2d+1}(N+1) \le h \le \log_{d+1}((N+1)/2) + 1$$

So, the cost of searching a key in a B-tree of order d with N keys requires no more than aprox. $\log_d N$ disk accesses (usually d >>1).

The actual value of d depends on the page size and key length. Typical values of d are 10..100, which results in a very small value of $\log_d N$ even for large N.

Typical B-tree is low, but very wide!

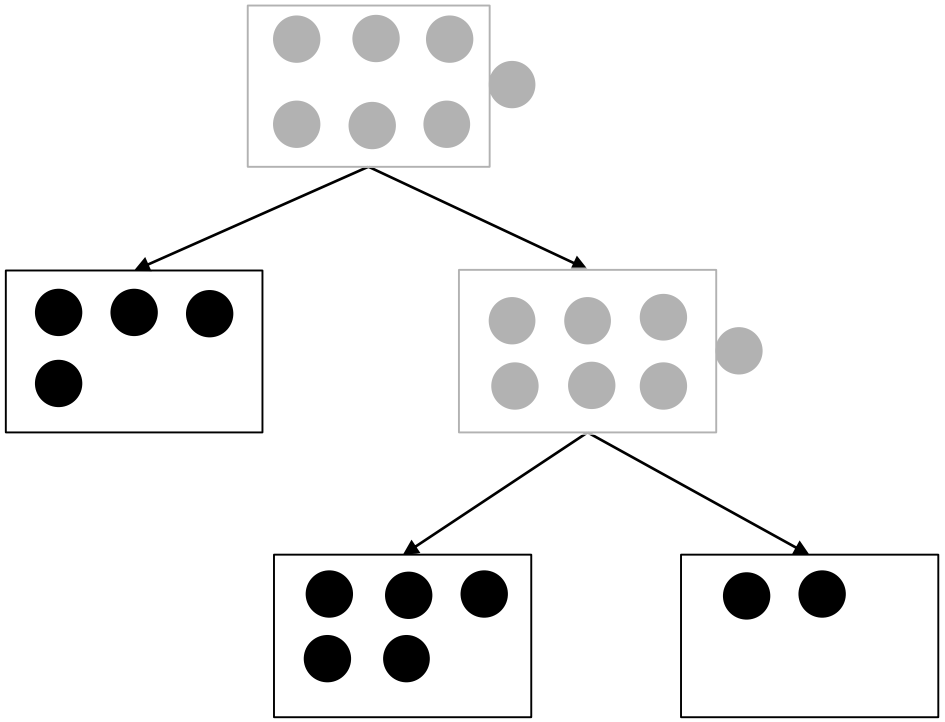
B-tree – Performance

The memory usage in a B-tree varies from ~50% (N_{min} keys) to 100% (N_{max} keys).

It can be shown that upon random insertions and deletions, the expected value of memory usage in a B-tree is

$$\ln 2 \ (\approx 70\%).$$

This is a typical value for so-called *buddy systems:*



Resulting memory usage ratio is

$\alpha = 11/18 = 0{,}61$

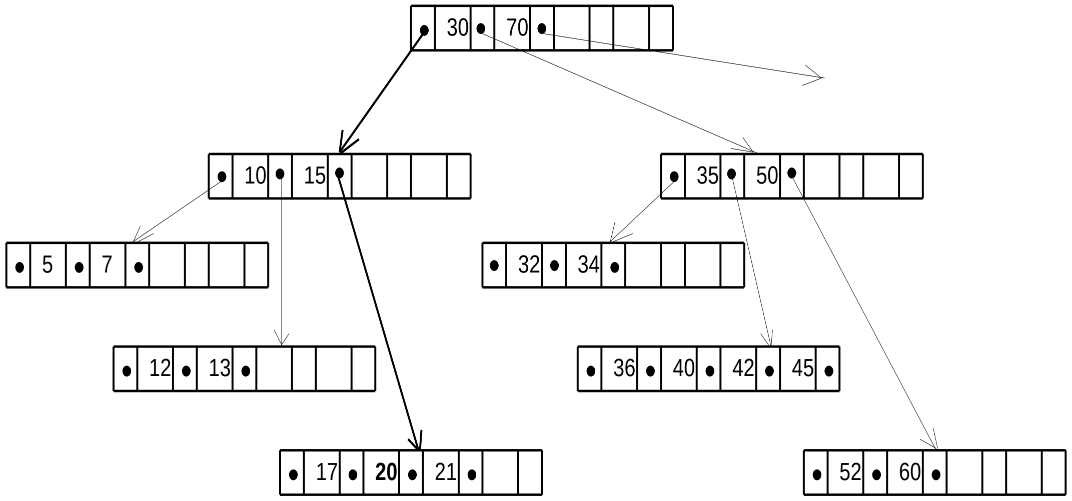
B-tree – Searching

Given the value of the key x. Find the pair (x,a) in a given B-tree.

Algorithm (denotations as in the slide "B-tree - Structure")

1. Let s be the address of the root page.
2. If s = NIL, then RETURN (Not_Found).
3. Fetch the page indicated by s to the main memory.
4. Search for x in this page (eg. using bisection).
5. If found $x_i = x$, then RETURN (x_i, a_i).
6. If $x < x_1$, then $s = p_0$. Go to step 2.
7. If $x > x_m$, then $s = p_m$. Go to step 2.
8. Let $s = p_i$, where $x_i < x < x_{i+1}$. Go to step 2.

<u>Example</u>: x = 20



The search was successful after 3 disk accesses.

B-tree – Inserting

Given a key-address pair (x,a). Insert the pair into a given B-tree.

Algorithm (*schema*)

```
1. Search for x (using the Searching algorithm).
2. If found, then RETURN (Already_Exists).
3. On the current page check if $m < 2d$. If YES,
      then insert (x,a) into this page, RETURN (OK).

                                                  /* OVERFLOW! */
4. Try compensation (description will follow).
5. If compensation possible, then RETURN (OK).

                                                  /* COMPENSATION IMPOSSIBLE! */
6. Make split of the overflown page (description will follow).
7. Make the ancestor page the current page. Go to step 3.
```

Note 1: After successful insert the new pair (x, a) has been inserted into a leaf page.

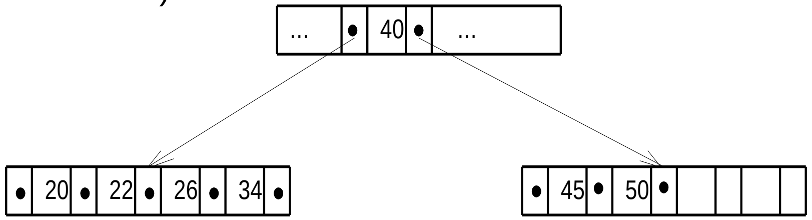
Note 2: After the split the ancestor page may overflow. Then the insertion process propagates towards the root. In an extreme case, the root page will overflow, which causes the root page to be split and the height of the B-tree increases by 1. For this reason, the algorithm's operation is most effective if buffers for the full path from root to the leaf have been reserved in memory (to avoid so-called page thrashing). It is recommended that the buffer capacity in the main memory is h+1 pages, where h is the current height of the B-tree.

B-tree – Inserting

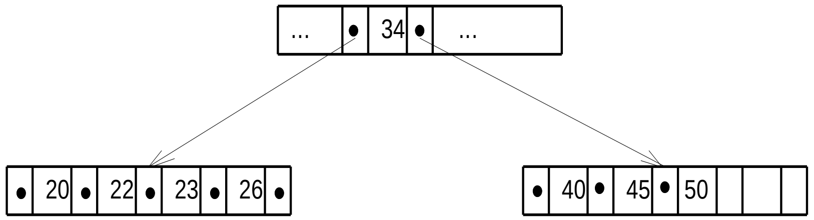
Compensation

If one of the neighboring pages ("siblings") is not full (contains less than 2*d* keys), then we take: all keys (of course with corresponding links) from the overflown page, all keys from this neighboring page and the corresponding key from the page ancestor. Then, we distribute these keys equally to the two pages (the overflown page and the neighbor page), replacing the key taken from the ancestor page with the middle key as to the value of all these keys.

<u>Example</u>: (values of *a* are omitted)



Insert x = 23. Overflow!. Compensation is possible. Result:



Note that there was no need to modify any pointers on the ancestor page.

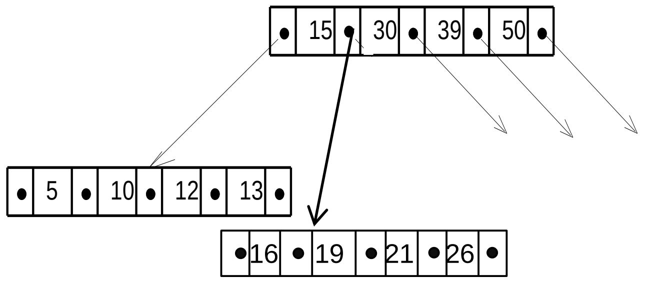
B-tree – Inserting

Split

If compensation is not possible (the overflown page does not have a sibling that is not full), you must:

1. Create (i.e. take from tthe pool of available disk pages) a new page.
2. Distribute all the keys, except the middle one, from the overflown page into two pages: the overflown and newly created.
3. Perform the operation of inserting the middle key into the ancestor page.

<u>Example</u>:

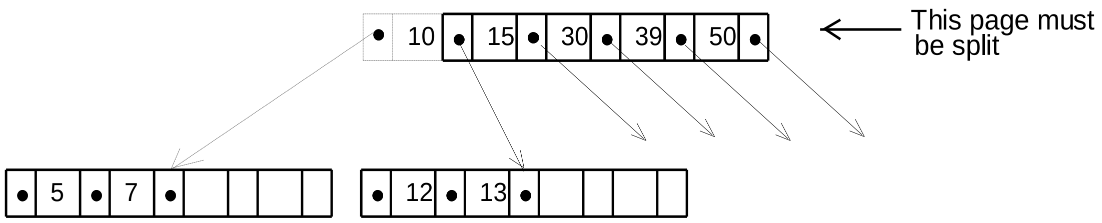


Insert x = 7. Overflow! Compensation is impossible (the only sibling is full).

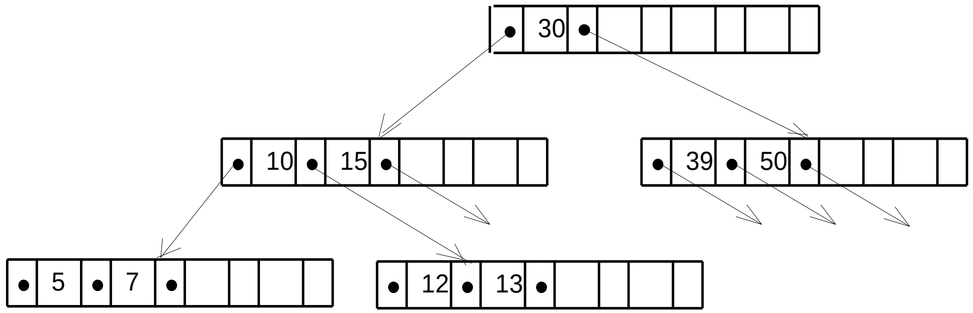
B-tree – Inserting

Split

After the first spli:



After the second split:



The height of the B-tree increased by 1.

B-tree – Deleting

Given the key x. Delete the key, i.e. the pair (x, a), from a given B-tree.

Algorithm (*schema*)

1. Search for x (using the Searching algorithm).
2. if not found, then RETURN (Not_Found).
3. If the key is on a page that is not a leaf, replace it with the largest key from the left subtree or the smallest key from the right subtree (note that this replacing key is always in the leaf). Then remove this replacing key from the leaf.
4. If after this operation in the leaf $m \geq d$, then RETURN (OK).

/* UNDERFLOW! */

5. Try **compensation**. If possible, then RETURN (OK).

/* COMPENSATION IMPOSSIBLE! */

6. Perform **merge** (*description will follow*).
7. If the underflow occurs on the ancestor page after the merge, go to step 5, making the ancestor page the current page. Otherwise, RETURN (OK).

Note:

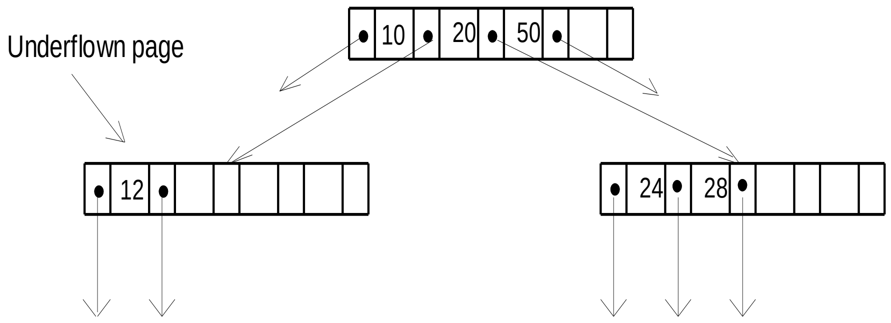
Page merge can propagate from leaves upwards, in extreme cases, up to the root. In this situation, the height of the B-tree may decrease by 1. As with insertion, it is recommended that the buffer capacity in the main memory is h+1 pages.

B-tree – Deleting

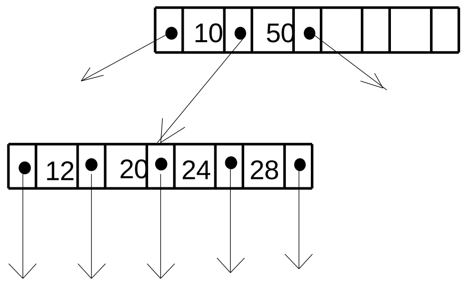
Merging

Assume that the neighbor page to the page with the underflow contains *d* keys, so the total number of keys on both pages is less than 2d. Therefore compensation is impossible. Instead, we can merge these two pages, retrieving one key from the ancestor side.

Example: (we assume that the page with keys <10 also contains only *d* keys)

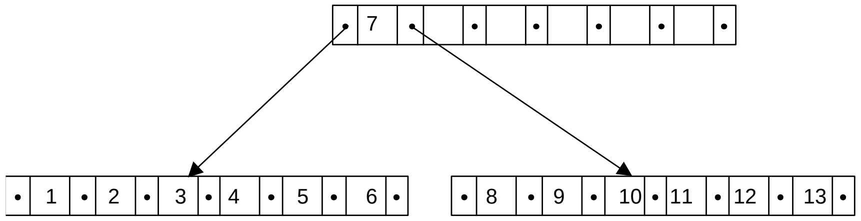


After merge:

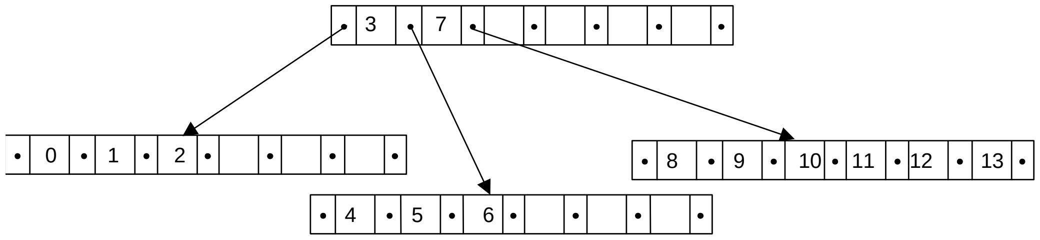


B-tree – Examples

1. Initial tree: (*d* = 3)

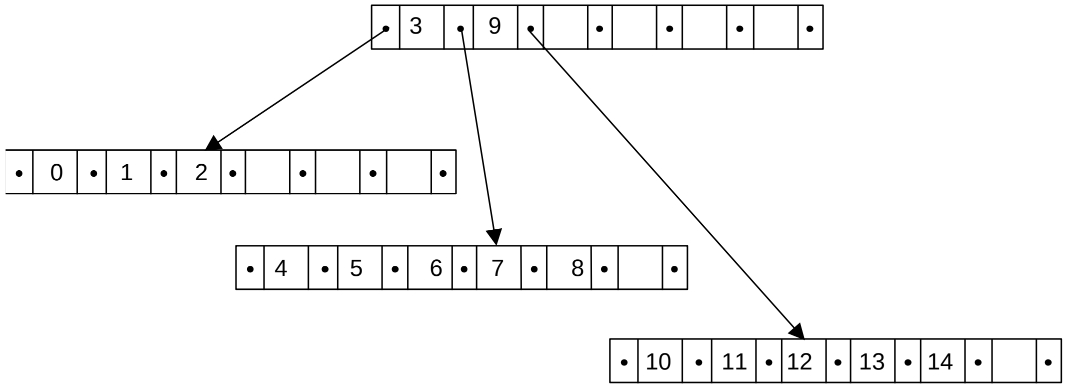


2. Insert x=0; compensation impossible, so we perform a split:

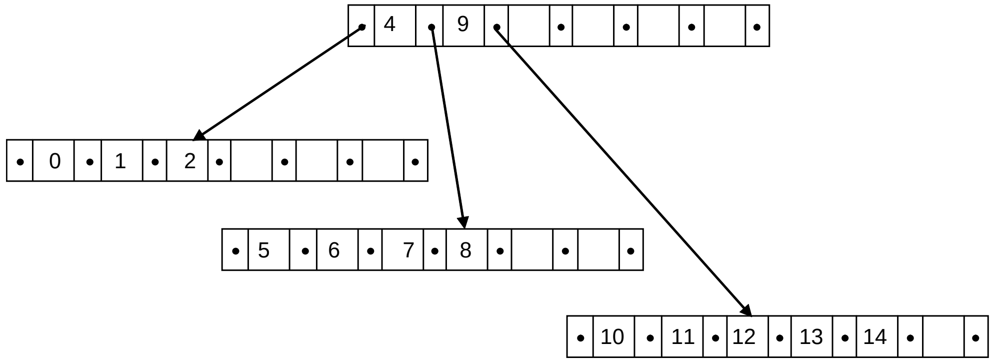


B-tree – Examples

3. Insert x=14; compensation possible:

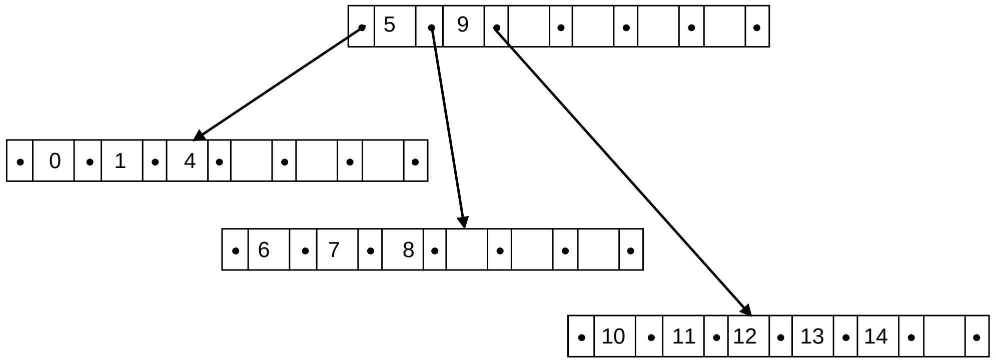


4. Delete x=3:

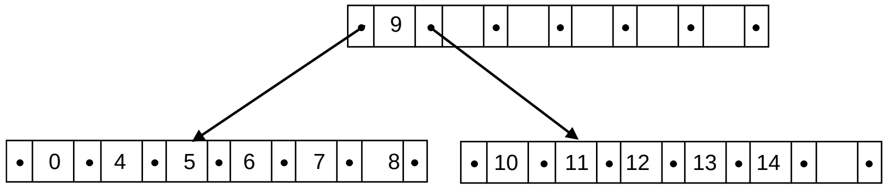


B-tree – Examples

5. Delete x=2; compensation possible:



6. Delete x=1; compensation impossible, perform merge:



GDAŃSK UNIVERSITY OF TECHNOLOGY

B-tree – Other operations

ETI

Get next record is simple if in the header of each page we store a pointer to the ancestor page. Then the operation of fetching the next key costs from min. 0 index page fetches (the next key is on the same page) up to max. h-1 index page fetches (when it is necessary to go back to the root page).

The actual number of index page fetches needed depends on the capacity of the buffer for pages. If we keep in main memory the whole path from the root to the current leaf, then at most 1 disk access is needed to read the index page from a different path.

Of course, it is also necessary to read the disk page from the main area in order to fetch the record pointed to by the address from the pair (*key*, *address*).

Sequential read of the entire file requires fetching all the B-tree pages (if the buffers in the memory have at least h pages, each page is read only once) and at most 1 page from the main file for each read item from the B-tree. If the file contains N records, then the number of pages in the index meets the following condition:

$$\lceil N / (2d) \rceil \leq \mathrm{SI}_N \leq \lceil (N-1)/d + 1 \rceil$$

and the maximum number of disk access needed is $T_E = \mathrm{SI}_N + N$. In practice, it may be much smaller, especially if we use more than 1 buffer for the main file pages, because then we increase the chances that another record from the main file is already in the buffer.

Nevertheless, the actual value of T_E may prevent an efficient execution of some SQL operations (e.g. joins). We will return to this problem when discussing *clustered indexes*.

B-tree with secondary keys

The algorithms given previously assume the uniqueness of keys (e.g. prmary keys in relations). By simple modification of these algorithms, the B-tree structure can also be used for auxiliary (non-unique) keys.

To this end, we extend the key *x* by a certain integer *n*, so that the pair (*x*, *n*) is unique in the whole file. A good choice for *n* is the sequence number of the record in the file or the concatenation of the page number in the main file with the slot number on the page, i.e. (detailed) disk address. The value of *n* is then unique, therefore the pair (*x*, *n*) is also unique. In this way, the key does not have to be physically extended, because the address of the record must be kept in the B-tree anyway.

Problem:

How do you modify the search, insert and delete algorithms?
In particular, how do you find all records with the same key?

For example, suppose that the key in the B-tree is the person's surname. How do you find all persons named Nowak?

B^+-tree - Definition

B^+ -tree is an extension of the indexed-sequential structure with a tree-like index. There are only keys in nodes that are not leaves. In leaf nodes there are stored even whole records (x, f), or main parts of records (e.g. without fields of variable length).

B^+ -tree can be treated as consisting of a B-tree (nodes other than leaves) to which the main file pages have been attached.

Parameters of a non-empty B^+-tree:

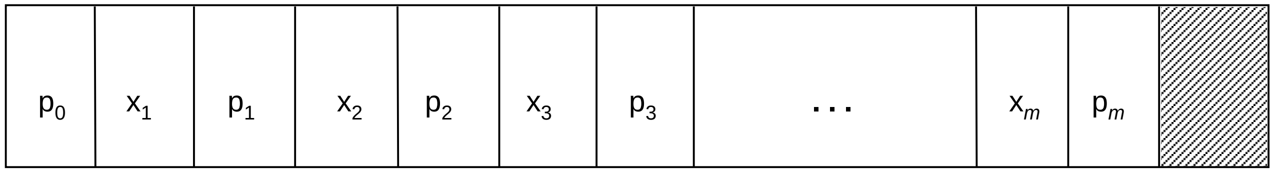
h^* - height of the tree, $h^* \geq 2$,

d^* - order of the B-tree (in the nodes that are no B^+-tree leaves),

r - order of the leaves (a leaf contains $r..2r$ records); normally $r < d^*$.

B⁺-tree – Structure

A non-leaf page:



$1 \le m \le 2d^*$ in the root page,
$d^* \le m \le 2d^*$ in the non-root pages,
x_i - a key,
p_i – a pointer to a child page.

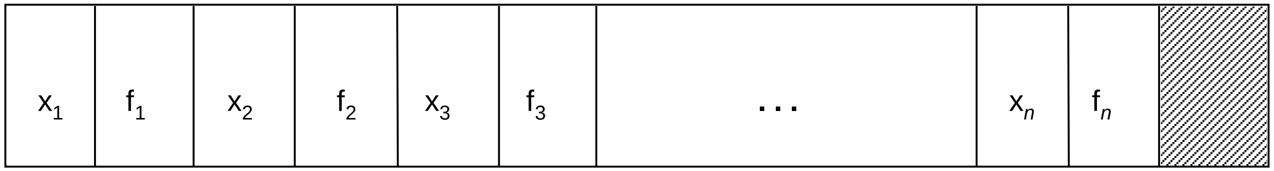
On each page:

$x_1 < x_2 < ... < x_m$
p_0 – pointer to the page with keys $\le x_1$
p_m - pointer to the page with keys $> x_m$
p_i - pointer to the page with keys that are $>x_i$ and $\le x_{i+1}$, $i = 1,2,...,m-1$.

Compare with the B-tree page structure!

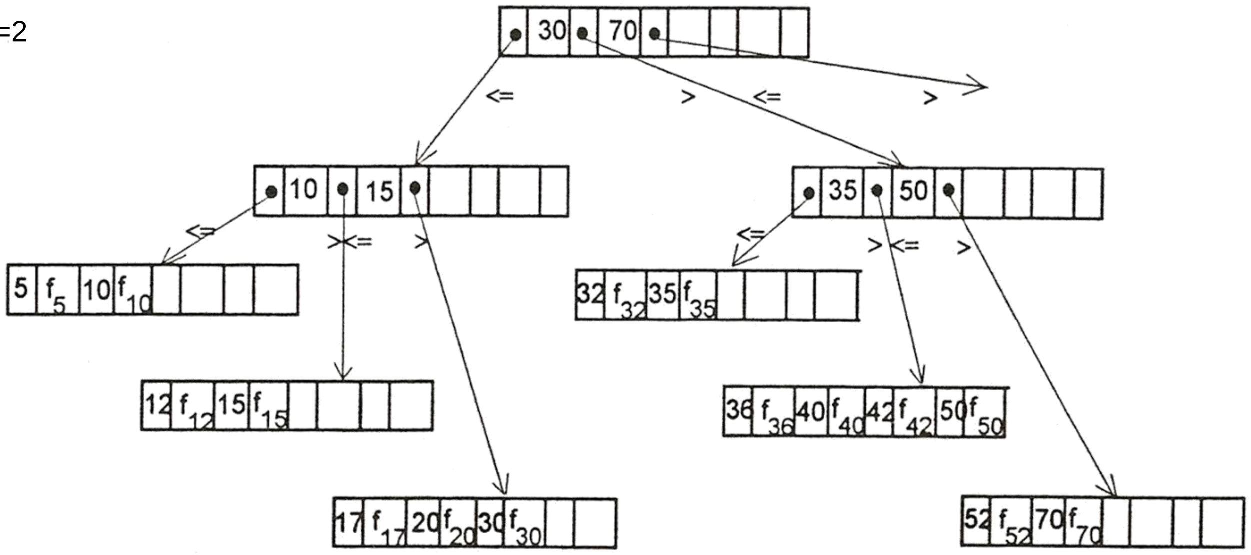
B^+-tree – Structure

A leaf page:



$r \leq n \leq 2r$
x_i - key
f_i - other fields of the record (or pointers to long or variable-length fields)

<u>Example:</u>
B^+-tree: $h^*=3$, $d^*=2$, $r=2$



B⁺-tree – Performance

Consider a B$^+$-tree (h^*,d^*,r), $h^* > 1$. **Minimal** number of different keys N_{min} (minimal number of records in leaves) is:

$$N_{min} = 2r(d^*+1)^{h^*-2}$$

Maximal number of different keys N_{max} (maximal number of records in leaves) wynosi:

$$N_{max} = 2r(2d^*+1)^{h^*-1}$$

So, the height of the **B$^+$-tree** satisfies the following condition:

$$1 + \lfloor \log_{2d^*+1}(N/(2r)) \rfloor \le h^* \le 2 + \lceil \log_{d^*+1}(N/(2r)) \rceil$$

So, it may assumed that searching a record in a B$^+$-tree (h^*,d^*,r) with N keys (and N records) requires no more than approx. $\log_{d^*}(N/2r)+2$ disk accesses (note that normally d^* >>1).

The B$^+$-tree is called the *clustered index*. It is clear that a given file can have only one clustered index. As a rule, RDBMS creates such an index on the primary key, which enables very effective sequential access to the file. Indeed, file records are ordered according to the value of the key, which minimizes the number of pages needed for sequential reading of the entire file according to subsequent key values. This is used not only in sorting operations by primary key values, but also in merge join strategies.

To make sequential access easier, some additional fields may be kept in each leaf page:
e.g, pointers to next (or previous) leaf page that consideralbly accelerate sequential access.