# CS3223 Tutorial 1: Storage & Indexing

**Wk 3, Sem 2, 2022/23**

**Questions to be discussed: 2, 4, 5 & 8.**

1. (Exercise 9.6, R&G)

   Consider a disk with a sector size of 512 bytes, 2000 tracks per surface, 50 sectors per track, five double-sided platters, and an average seek time of 10 msec. Assume that the disk platters rotate at 5400 rpm (revolutions per minute), the disk can read/write from only one head at a time, and a block size of 1024 bytes is chosen.

   Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

   (a) How many records fit onto a block?

   (b) How many blocks are required to store the entire file? If the file is arranged sequentially on the disk, how many surfaces are needed?

   (c) How many records of 100 bytes can be stored using this disk?

   (d) If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what page is stored on block 1 of track 1 on the next disk surface?

   (e) What time is required to read a file containing 100,000 records of 100 bytes each sequentially?

   (f) What is the time required to read a file containing 100,000 records of 100 bytes each in a random order? To read a record, the block containing the record has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

**Solution:**

In this question, we assume that only one disk head can read/write at any one time; i.e., reading from multiple tracks in parallel is not supported.

(a) $\frac{1024}{100} = 10$ records per block

(b) Number of blocks to store file = 100,000/10 = 10,000 blocks. Number of blocks per track $= \frac{50 \times 512}{1024} = 25$. Number of blocks per cylinder $= 25 \times 5 \times 2 = 250$.

Storing a file sequentially on a disk has the following properties: (1) All the blocks in a cylinder are sequentially accessed before the blocks in the next contiguous cylinder are accessed. (2) Within a cylinder, all the blocks on a cylinder track (i.e., surface) are sequentially accessed before the blocks on the next cylinder track. (3) Within a cylinder track, the blocks are sequentially accessed by reading contiguous sectors.

Since the size of the file exceeds the capacity of a cylinder, the file needs to be stored in more than one cylinder. Therefore, 10 surfaces are needed to store the file.

(c) Number of blocks for disk $= 250 \times 2000 = 500,000$. Number of records for disk $= 500,000 \times 10 = 5$ million.

(d) Each track can store 25 blocks. So page 26 is stored on block 1 of track 1 on the next surface.

(e) From (b), we know that file requires a storage space of 10,000 blocks and each cylinder can store 250 blocks. Therefore, the file requires $\frac{10000}{250} = 40$ cylinders (or 400 tracks). Since this access seeks the tracks 40 times, the total seek time is $40 \times 0.01 = 0.4$ sec.

Assuming that rotational delay is incurred only for the first block, the total rotational delay $= \frac{60}{5400} \times 0.5 = 0.006$ sec.

The transfer time of one track is $\frac{60}{5400} = 0.011$ sec. Therefore, time taken to transfer 400 tracks is $400 \times 0.011$ sec $= 4.4$ sec.

Therefore, total access time is 0.4 + 0.006 + 4.4 = 4.806 sec.

(f) Average seek time = 10 msec. Average rotational delay = 6msec. Since each track can store 25 block, transfer time for one block $= \frac{0.011 \times 1000}{25} = 0.44$ msec.

Average access time for one record = 10 + 6 + 0.44 = 16.44 msec.

Since the file consists of 100,000 records, the total access time $= (100,000 \times 16.44 \times 10^{-3})$ sec = 1644 sec.

2. Consider a buffer pool with a total of 10 buffer frames (numbered 0 to 9). Suppose that an application needs to make three repeated sequential scans of a file that has a total of 11 pages ($p_1$ to $p_{11}$) resulting in the following sequence of page requests:

$$\underbrace{p_1, p_2, \cdots, p_{11}}_{1^{st} \text{ scan}}, \underbrace{p_1, p_2, \cdots, p_{11}}_{2^{nd} \text{ scan}}, \underbrace{p_1, p_2, \cdots, p_{11}}_{3^{rd} \text{ scan}}.$$

Assume the following for this question:

1. All the buffer frames are initially in the free list.
2. Access to a page p requires a disk page read if p is not in the buffer pool.
3. For each accessed page $p$, the application always unpins $p$ before making the next page request (i.e., there are no pinned pages in the buffer pool when a page request is issued).

(a) How many disk page reads are incurred under the LRU replacement policy?

(b) How many disk page reads are incurred under the Clock replacement policy?

(c) How many disk page reads are incurred under the MRU replacement policy?

**Solution:**

(a) **LRU:** The first scan requires 11 page I/Os. The reading of $p_{11}$ replaces $p_1$; so after the first scan, the buffer contains $\{p_2, p_3, \cdots, p_{11}\}$. Each subsequent scan also requires 11 page I/Os. In both the second and third scans, the reading of $p_i$ replaces $p_{i+1}$ (for i = 1 to 10); and the reading of $p_{11}$ replaces $p_1$. After the second/third scan, the buffer contains $\{p_2, \cdots, p_{11}\}$. Total number of disk page reads = 11+11+11 = 33. The above phenomenon where the scan of a file requires each page of the file to be read is known as *sequential flooding*. LRU performs very poorly for this access pattern.

(b) **Clock:** Assume current = 0 and that all the buffer frames are initially in the free list. Suppose that after the first scan of pages $p_1$ to $p_{10}$, we have the following state after $p_{10}$ is unpinned:

current = 0

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| referencedBit | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Note that the initial reading of first ten pages do not involve using the clock replacement algorithm to look for replacement buffer pages as all the buffer frames are in the free list. The referenced bit for a page is set to 1 whenever its pin count drops to zero after an unpin operation.

The reading of $p_{11}$ replaces $p_1$ and we have the following state after $p_{11}$ is unpinned:

current = 0

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 11 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| referencedBit | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In the second scan of pages $p_1$ to $p_9$, the reading of $p_i$ replaces $p_{i+1}$; and we have the following state after $p_9$ is unpinned:

current = 9

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 11 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| referencedBit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

The second scan of page $p_{10}$ replaces $p_{11}$ and we have the following state after $p_{10}$ is unpinned:

current = 0

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| referencedBit | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The second scan of page $p_{11}$ replaces $p_1$ and we have the following state after $p_{11}$ is unpinned:

current = 1

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 10 | 11 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| referencedBit | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In the third scan of pages $p_1$ to $p_8$, the reading of $p_i$ replaces $p_{i+1}$; and we have the following state after $p_8$ is unpinned:

current = 9

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 10 | 11 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| referencedBit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

After the third scan of pages $p_9$ to $p_{11}$, we have the following state:

current = 2

| frame | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| page | 9 | 10 | 11 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| referencedBit | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Total number of disk page reads = 11+11+11 = 33.

(c) **MRU:** The first scan requires 11 page I/Os and the reading of $p_{11}$ replaces $p_{10}$. The buffer contains $\{p_1, p_2, \cdots, p_9, p_{11}\}$ after the first scan. The second scan requires one page I/O for $p_{10}$ (which replaces $p_9$). The buffer contains $\{p_1, p_2, \cdots, p_8, p_{10}, p_{11}\}$ after the second scan. The third scan requires one page I/O for $p_9$ (which replaces $p_8$). Total number of disk page reads = 11+1+1 = 13.

For this application's access pattern, both LRU and Clock performed equally poorly; indeed, the same performance would be obtained if the buffer pool had only one frame! MRU turns out to be the optimal replacement policy for this access pattern.

3. Consider a B$^+$-tree index with a page size of 8192 bytes, 4-bytes key and 8-bytes disk page address.
   (a) What is the maximum order of the index?
   (b) Assume that the order of the index is given by your answer for part (a) and the height of the index is 3 (i.e., the index has 3 levels of internal nodes).
      i. What is the minimum number of leaf nodes in the index?
      ii. What is the maximum number of leaf nodes in the index?

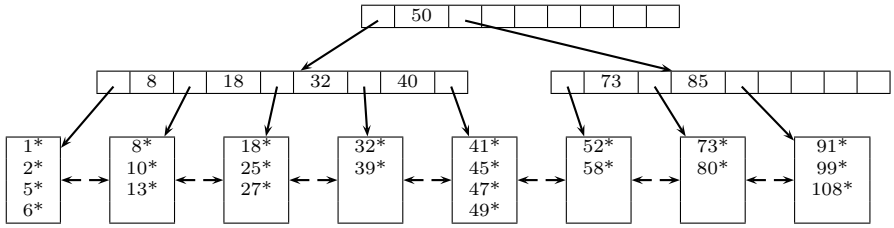
**Solution:**

(a) Let d be the order of the index. Solving $2d(4) + (2d + 1)(8) \leq 8192$, we have $d \leq 341$. Therefore, the maximum order is 341.

(b) i. To minimize the number of leaf nodes, each internal node has the minimum number of keys (i.e., 1 for the root node and d for each non-root node).
The minimum number of nodes at level i is $2 \times (d + 1)^{i-1}$, for $i \geq 1$.
Therefore, the minimum number of leaf nodes is $2 \times (d + 1)^2 = 233{,}928$.

ii. To maximize the number of leaf nodes, each internal node has the maximum number of keys (i.e., 2d).
The maximum number of nodes at level i is $(2d + 1)^i$.
Therefore, the maximum number of leaf nodes is $(2d + 1)^3 = 318{,}611{,}987$.

4. Consider the $B^+$-tree index of order d=2 shown below.



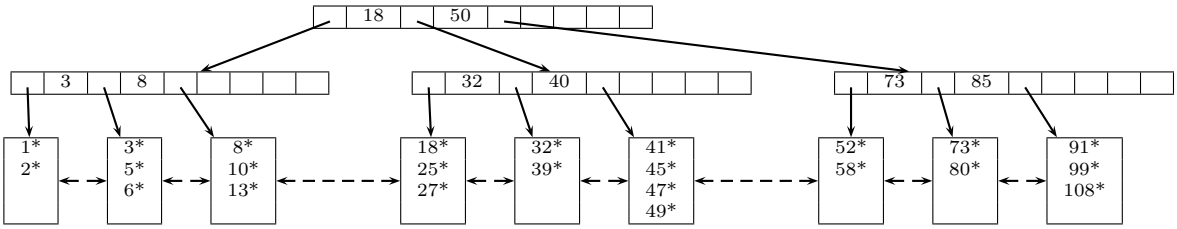

In this question, assume that an overflowed node is always split.

(a) Show the $B^+$-tree index that would result from inserting a data entry with key 3 into this index.

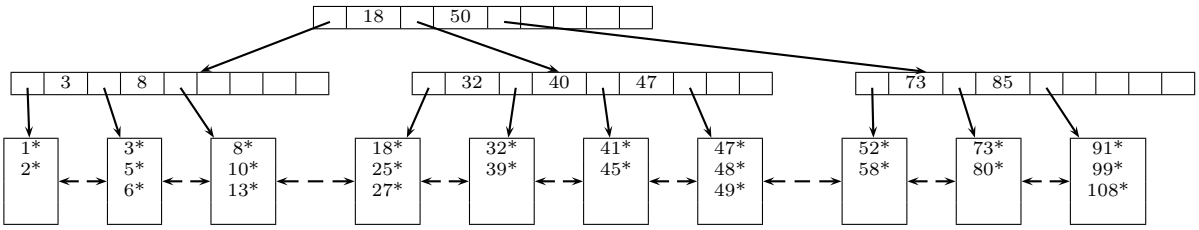
(b) Show the $B^+$-tree index that would result from inserting a data entry with key 48 into the $B^+$-tree index obtained from part (a).

**Solution:**

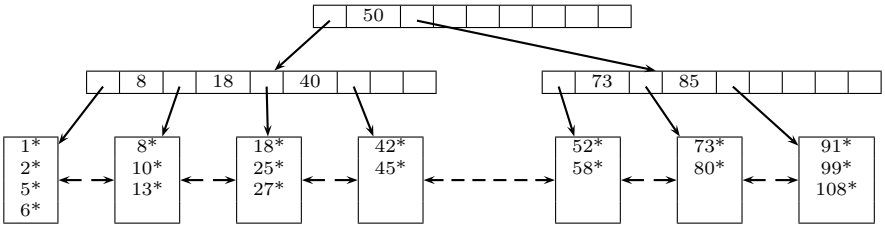
(a)




(b)



5. Consider the B$^+$-tree index of order d=2 shown below.




Show the B$^+$-tree index that would result from deleting the data entry with key 80 from this index under each of the following scenarios:

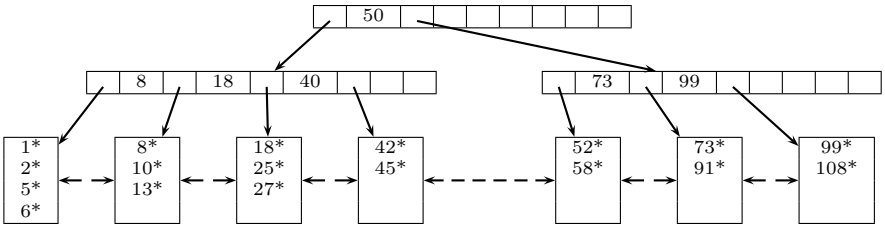
(a) Redistribution is used whenever possible.

(b) Redistribution is not used for leaf nodes but it is used whenever possible for internal nodes.

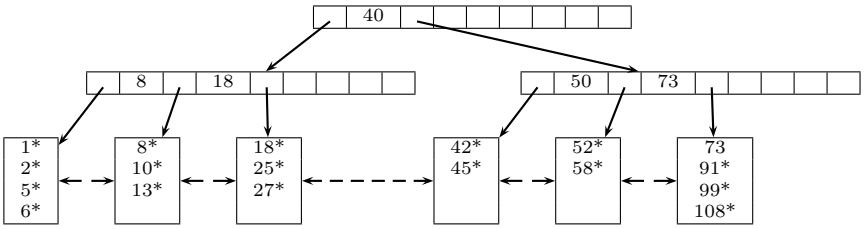
(c) Redistribution is not used for both leaf and internal nodes.

**Solution:**

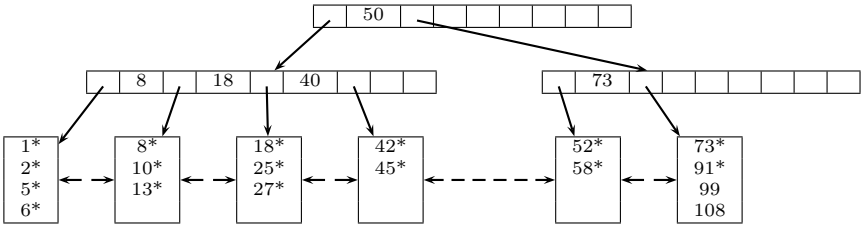
(a)




(b)




(c) The underflowed internal node can't be resolved! Here, we do not perform merging to create an overflowed node followed by splitting the overflowed node.



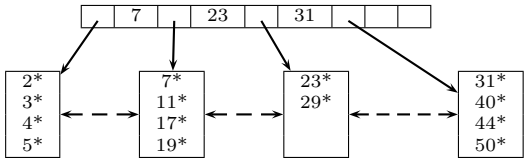
6. In this question, assume that an overflowed node is always split.
   (a) Show the $B^+$-tree index (of order d = 2) constructed by inserting the following sequence of 14 data entries one at a time:

$$11^*, 5^*, 3^*, 31^*, 19^*, 2^*, 7^*, 17^*, 23^*, 29^*, 4^*, 50^*, 40^*, 44^*$$

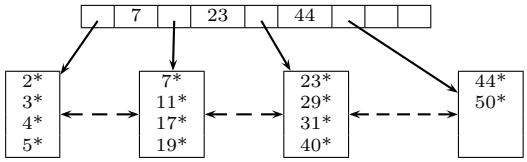
   (b) Show the $B^+$-tree index (of order d = 2) constructed by bulk-loading the same set of data entries in (a).

**Solution:**

(a) **Top-down insertion**
The following solution assumes that redistribution is performed whenever possible.




(b) **Bulk-loading**



7. This question considers the update cost for handling overflowed nodes in $B^+$-tree indexes.
   (a) What is the maximum number of nodes that need to be updated when an overflowed leaf node is resolved using redistribution?
   (b) What is the maximum number of nodes that need to be updated when an overflowed leaf node is split without any overflow propagation?
   (c) What is the maximum number of nodes that need to be updated when an overflowed internal node is resolved using redistribution?
   (d) What is the maximum number of nodes that need to be updated when an overflowed internal node is split without any overflow propagation?

**Solution:**

(a) 3 (overflowed leaf node, adjacent sibling leaf node used for redistribution, parent node of updated leaf nodes).

(b) 4 (overflowed leaf node, new leaf node, adjacent sibling of overflowed leaf node, parent node of updated leaf nodes).

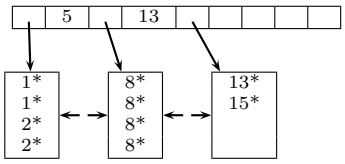
(c) 3 (overflowed internal node, adjacent sibling internal node used for redistribution, parent node of overflowed node).

(d) 3 (overflowed internal node, new internal node, parent node of overflowed node).

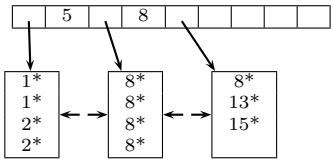
Based on the above analysis, the following strategy is generally adopted to handle overflowed nodes in $B^+$-trees:

1. Whenever a leaf node overflows, we first attempt to resolve the overflow using redistribution with one adjacent sibling node N. If redistribution with N is not feasible, we split the overflowed node. By performing splitting only as a last resort, the update cost of resolving overflowed leaf nodes is kept low.
2. Whenever an internal node overflows, we simply resolve the overflow by splitting the overflowed node (i.e., redistribution is not performed for internal nodes). Since the chance of overflow propagation is low and both redistribution and splitting (without overflow propagation) incur the same update cost, the preference is to resolve overflows with splitting as the indexed relation is expected to grow over time with new tuples and splitting is inevitable.

8. This question considers an issue with non-unique B$^+$-tree indexes. Consider the following non-unique B$^+$-tree index with order = 2 shown below.




Suppose that we need to insert a new data entry with a key value of 8 into this index. Using redistribution to handle the overflowed leaf node caused by the insertion will result in the following index:




However, the search algorithm discussed in class is no longer sound if the same key value could be stored in multiple leaf nodes (e.g., searching for k=8 will search only the third leaf node); thus, we will need to modify the logic of the search algorithm for non-unique B$^+$-tree indexes.

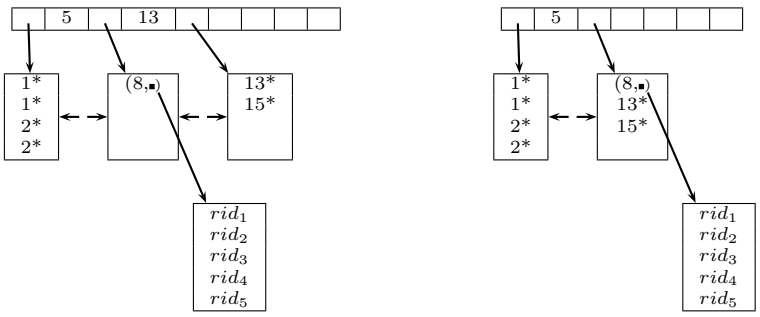
Suggest a way to handle duplicate key values that does not require modifying the search algorithm.

**Solution:**

One idea is to use Format 3 for non-unique indexes so that each key value is stored exactly once together with its RID-list; however, this does not quite solve the problem if the data entry (key,RID-list) exceeds the disk page capacity.

For a key k that has a very large RID-list, one solution is not to store the RID-list in the leaf node itself but to store the RID-list in a separate linked list of disk pages (referred to as overflow pages). Thus, the data entry (k,RID-list) in the leaf node is replaced by a special data entry (k,p) where p is the disk address of first page of a possibly long linked list of overflow pages to store the large RID-list.

For the given example, the left figure below shows the storage of the RID-list for key 8 using a single overflow page that is linked to the data entry for 8 in the leaf page. To resolve the resultant underflowed leaf page, we could merge with its right sibling node as shown in the right figure below.




Another solution is to support only unique B$^+$-tree indexes. For each non-unique index (on a search key k) that we want to create, we instead create a unique index on the composite search key (k,x), where x is some attribute with unique values (e.g., x could be a candidate key or RID). Using this composite-key unique index, an equality search query (say k=10) is transformed to a range search query (k=10 and x $\geq -\infty$), where $-\infty$ denote the minimum value for the domain of $x$.