Problem Set 2: Storage and Indexing

COMP_SCI 339

Due: Tuesday, October 10, 2023

Name: Zhuoyuan Li

1 Data Storage: Heap Files

1. (4 points) Consider a relation S with the following schema:

```
S(a int, b char(10), c char(20))
```

Consider the following small instance of S:

a	b	c
1	Orange	First
2	Red	Second
3	Blue	Third

Assume that S is stored in a heap file on disk, a page size of 8KB (8192 bytes), and 4-byte integers.

Say that we are designing a heap file for a DBMS. We will have one heap file for each table in a database. The heap file supports tuples of fixed length. Each page in a heap file is arranged as a set of slots, each of which can hold one tuple. In addition to these slots, each page has a header that consists of a bitmap with one bit per tuple slot. If the bit corresponding to a particular tuple is 1, it indicates that the tuple is valid; if it is 0, the tuple is invalid (e.g., has been deleted or was never initialized).

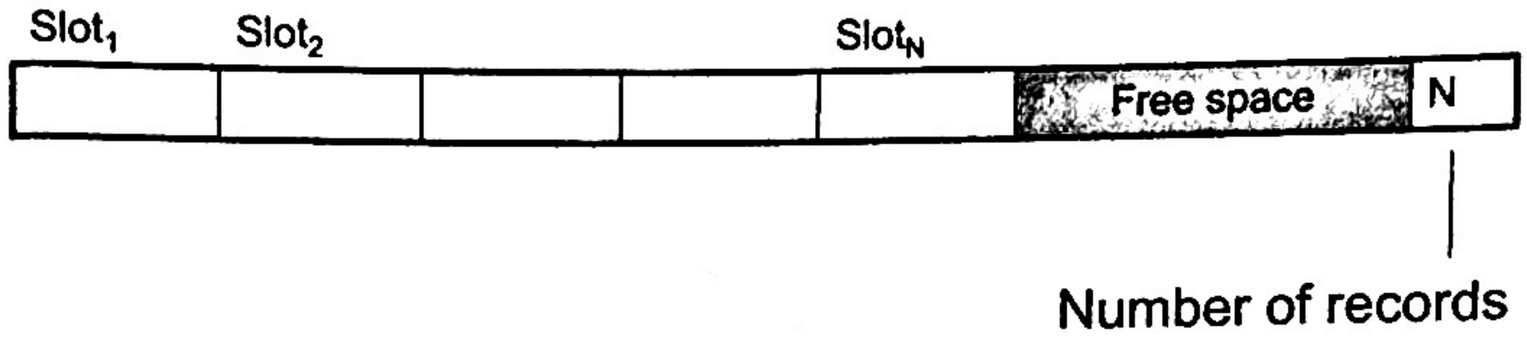
Each heap file consists of page data arranged consecutively on disk. Each page consists of: 1) one or more bytes representing the header, followed by 2) the *page_size*- # header bytes bytes of actual page content or tuples in their slots. Each tuple requires tuple size * 8 bits for its content and 1 bit for the header. Thus, the number of tuples that can fit in a single page is:

$$tuplesPerPage = \lfloor (page\_size * 8)/(tuple\_size * 8 + 1) \rfloor$$

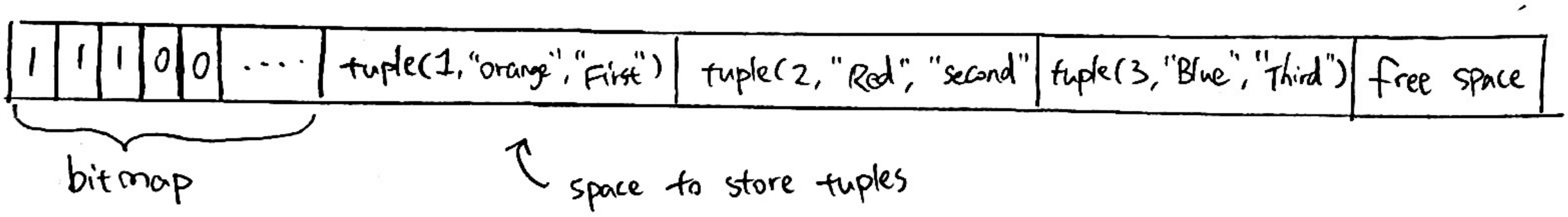
where tuple size is the size of a tuple in the page in bytes. The idea here is that each tuple requires one additional bit of storage in the header. We compute the number of bits in a page (by multiplying page_size by 8), and divide this quantity by the number

of bits in a tuple (including this extra header bit) to get the number of tuples per page. The floor operation rounds down to the nearest integer number of tuples (we don't want to store partial tuples on a page!).

(a) (1 point) Draw a *schematic* representation of the heap file page on disk storing the S instance. A example of this format for a page is:



Fill in the fields to show the three tuples on the page. Show where the empty space is on the page (if any). Show padding/unused space (if any). Do not worry about endianness nor about getting all proportions right. If you want, you may specify byte offsets for the records but you do NOT have to show that information.



(b) (2 points) The size of each S tuple in bytes is $4 + 10 + 20$ or 34 bytes. How many S tuples fit on one page? What is the size of the page *header* in bytes? How many pages are necessary to hold an instance of S with 1000 tuples (remember the space necessary for the page header)?

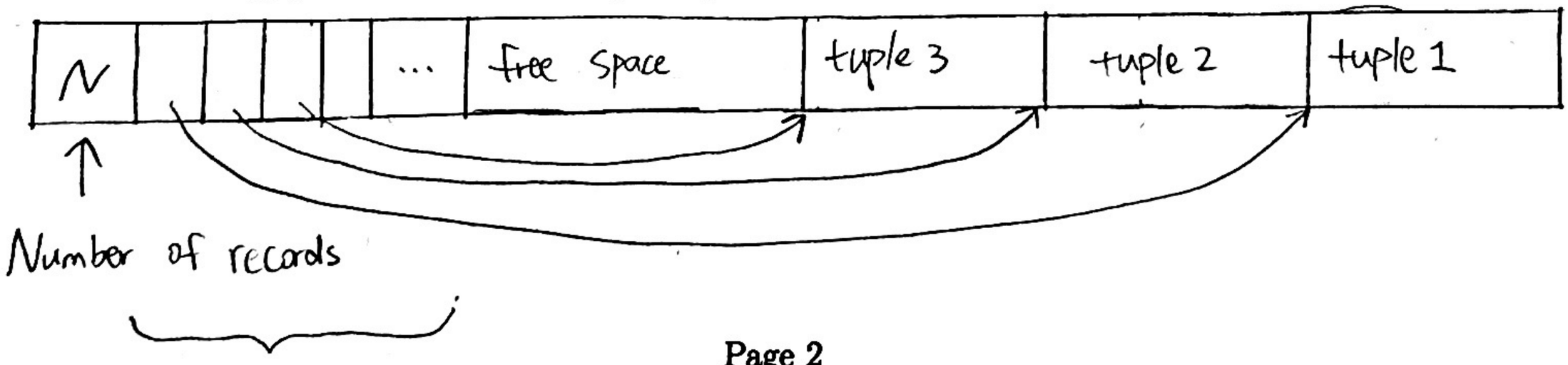
each tuple takes 34 bytes space, and for each page is in 8192 bytes size.

Therefore, tuples per page $= \lfloor (8192 \cdot 8)/(34 \cdot 8 + 1) \rfloor = 240$ tuples

The size of the page header is $240/8 = 30$ bytes

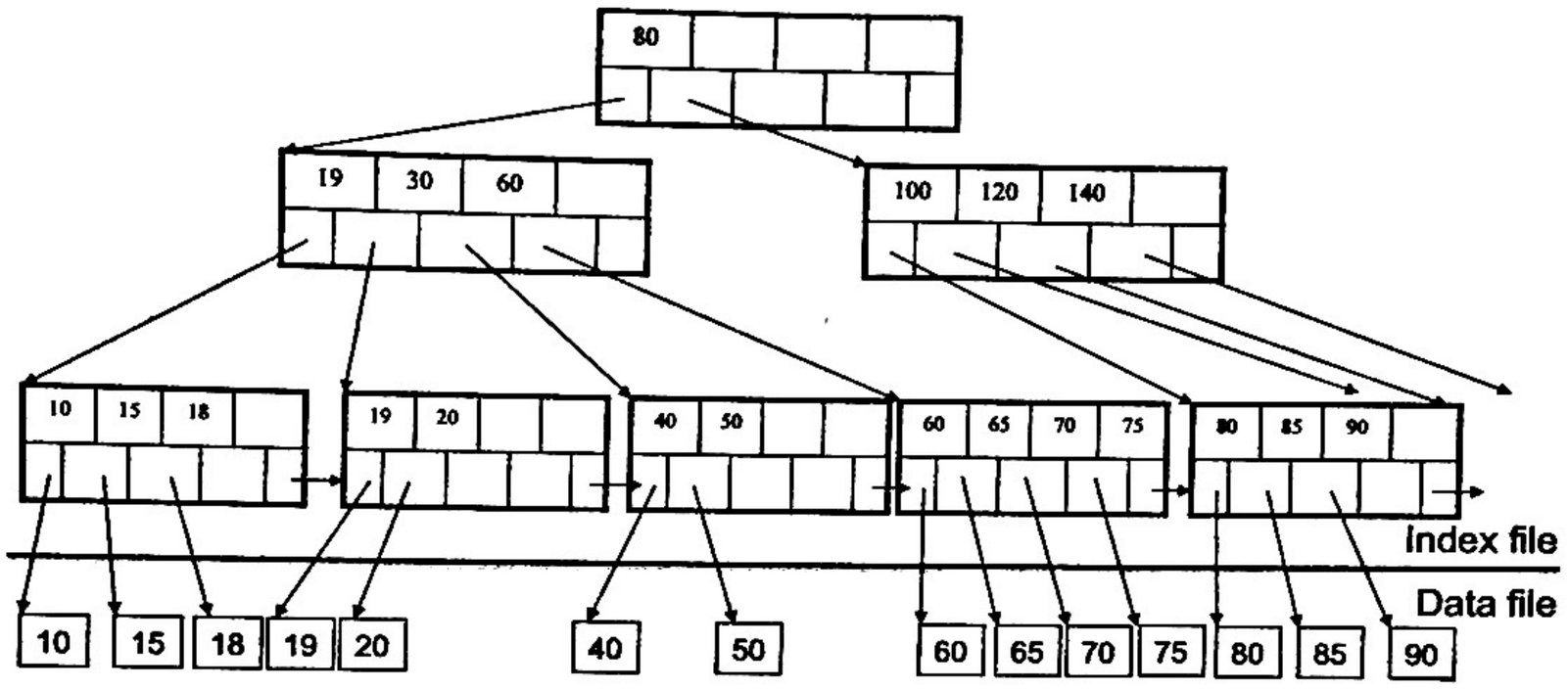
To hold 1000 tuples, we need $\lceil \frac{1000}{240} \rceil = 5$ pages

(c) (1 point) Now imagine that we wanted to extend scheme above to support *variable-length* tuples, draw a *schematic* representation of the modified Heap file page on disk storing the S instance (the one with the three tuples). Assume the strings are now variable lengths (VARCHAR). Do not worry about the detailed representation of the records themselves, though.

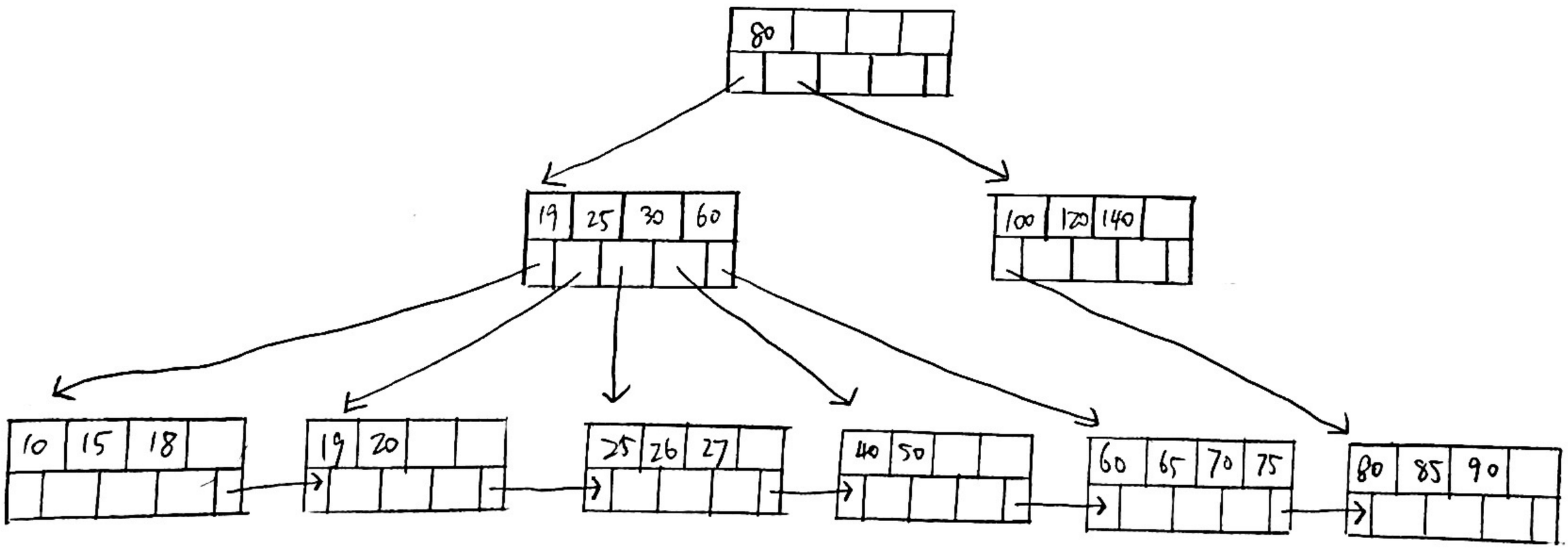


2 B+ Trees

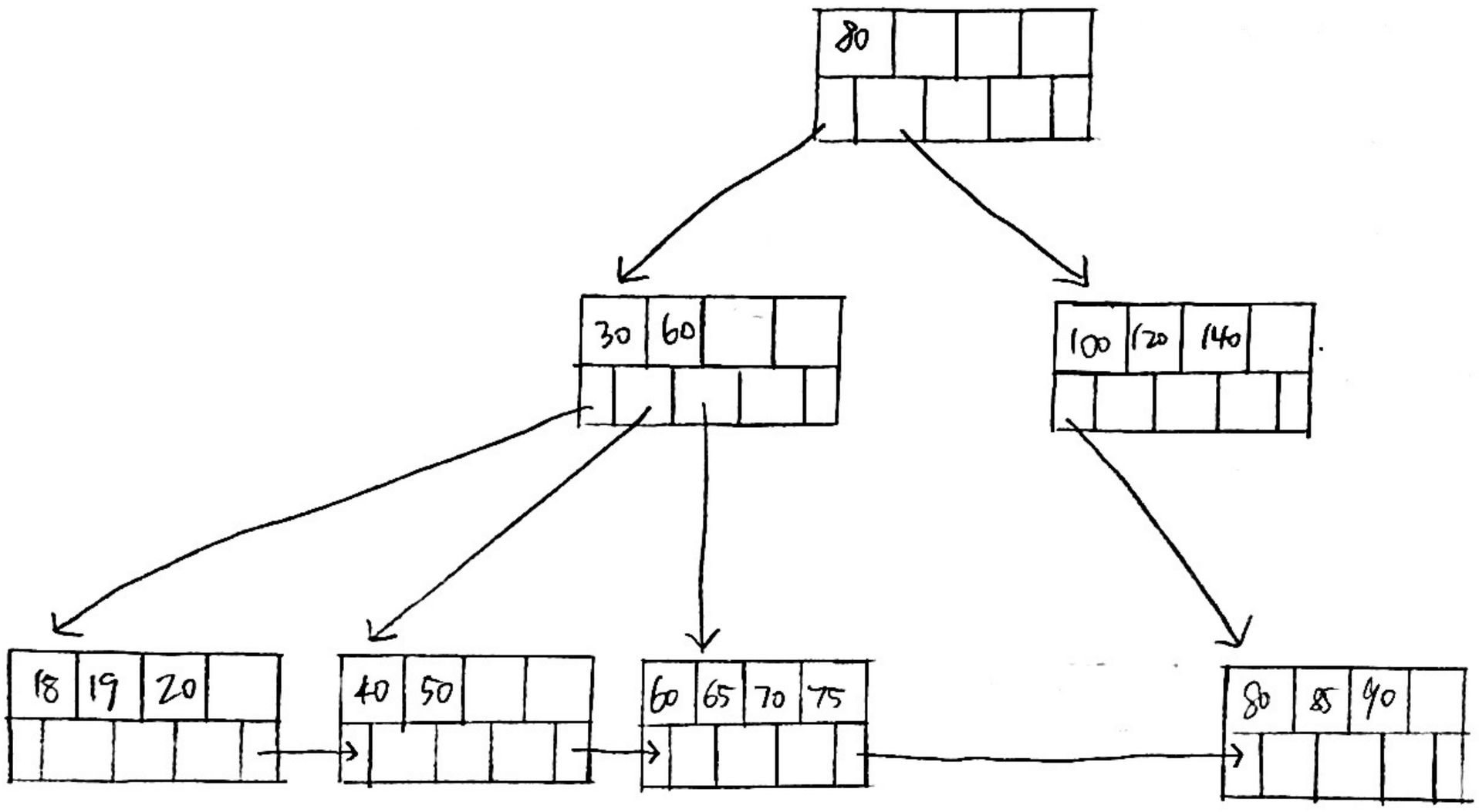
2. (4 points) Consider the following B+ tree index:



(a) (1 point) Draw the modified tree after the insertion of tuples with search key values 25, 26, and 27 into the base relation and index. Draw the final tree only.



(b) (1 point) Consider the *original* tree again. Draw the modified tree after the deletion of tuples with search key values 10 and 15. Draw the final tree only.



(c) (2 points) Consider the original tree again. How many pages will be read from disk to answer each of the following queries. Assume that the buffer pool is empty before each query executes, and that it is large enough to hold the entire database. Assume also that 4 tuples from the base relation fit on one page in the data file and that all pages are full:

- Look up all tuples with search key value of 40 if the index is *clustered.*
- Look up all tuples with search key value of 40 if the index is *unclustered.*
- Look up all tuples with search key value in the range $[60, 90]$ if the index is *clustered.*
- Look up all tuples with search key value in the range $[60, 90]$ if the index is *unclustered.* Assume the worst-case scenario in terms of how the keys are stored in pages.

1). Because the index is clustered, we start with 80 and store it into one page, then we look down and save 19, 30 into one page, then look down to store 40 into one page. At this point we can read 40 from the data file, which is another page. Therefore, we need 4 pages for searching key 40.

2). Even though the index is now unclustered, we still follow the same path in clustered. Since the index file can lead us to 40 in the data file directly, we need 4 pages for searching key 40.

3). In this case, we start from 80, then 19, 30, 60, now we detected the starting point, then we load 60, 65, 70, 75 into one page, we don't reach 90 yet, so we load 80, 85, 90 into another page. Now, we have covered from 60 to 90. Then, we load data from data file, since the index is clustered. we can find 60 on the second page on the data file, and then load 65, 70, 75, 80 into another page, then 85, 90 as another page. At this point, we have covered 60 to 90 in the data file. Therefore, we need to read 7 pages to read key values in the range 60 to 90.

4). In the worst case, to cover 60 to 90 in the index file is still the same, but to locate each value in the data file is not easy. Therefore, we need to read one page at a time for 60, 65, 70, 75, 80, 85, 90 to cover the range 60 to 90.
The total page we need then will be 11 pages.

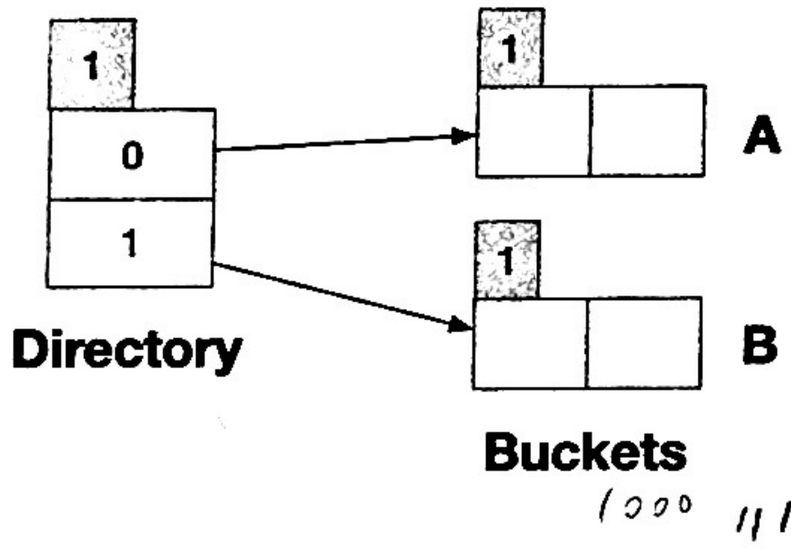
3 Extendible Hashing

3. (2 points) Let's examine extendible hash indexes.

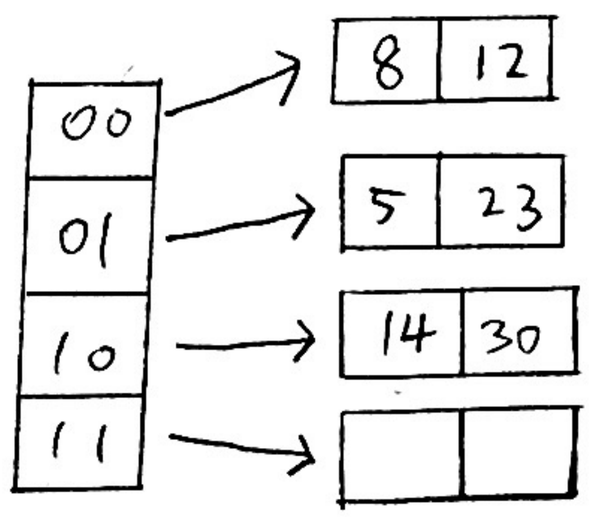
   Consider a small extendible hashing structure where:

   - Each bucket can hold up to 2 records.
   - It has just two primary pages or pages that are unconditionally allocated
   - Uses the least-significant bits for the hash function (i.e., records in a bucket of local depth d agree on the **rightmost** d bits.) For example, key 4 (0100) and key 12 (1100) agree on their rightmost 3 bits (100).

   An example of an empty instance of this index is:



   (a) (2 points) Draw this hash index after inserting keys 8, 5, 14, 23, 12, and 30 (in that order).



(b) (2 points) Consider the extendible hash index you drew above. What is the smallest, unique, positive (> 0) key you can insert into it that will result in the directory doubling in size twice, giving it a global depth of 4?

Since we need to double the size twice, we need to trigger data overflow twice. Now, buckets 00, 01, 10 are fully occuppied. So adding one value to any of them will result in data overflow. Haveve, redistribution will assign values with different suffix into different buckets. Therefore, we can only use 14 and 30 to extend the depth, since they both end with 110. Then, add 6 to the hash, it will cause overflow along with 14 and 30 and double the directory from depth 2 to 3, from 3 to 4.

Therefore, the answer will be 6 (110)