Project #2

B+tree Practice

Basic B+tree

- In-memory 형식의 B+tree를 실행해보며 구조를 파악해보시길 바랍니다

```
$ ./main
...
Enter any of the following commands after the prompt > :
        i <k> -- Insert <k> (an integer) as both key and value).
        f <k> -- Find the value under key <k>.
        p <k> -- Print the path from the root to key k and its associated value.
...
> i 1
1 |
> i 2
1 2 |
> i 3
1 2 3 |
> i 4
3 | 1 2 | 3 4 |
>
```

각각의 commands에 따라
다음과 같은 결과를 얻을 수 있습니다

Basic B+tree

- In-memory 형식의 B+tree를 실행해보며 구조를 파악해보시길 바랍니다

```
$ ./main 5
...
Enter any of the following commands after the prompt > :
        i <k> -- Insert <k> (an integer) as both key and value).
        f <k> -- Find the value under key <k>.
        p <k> -- Print the path from the root to key k and its associated value.
...
3 |
1 2 | 3 4 5 6 |
> i 7
3 5 |
1 2 | 3 4 | 5 6 7 |
>
```

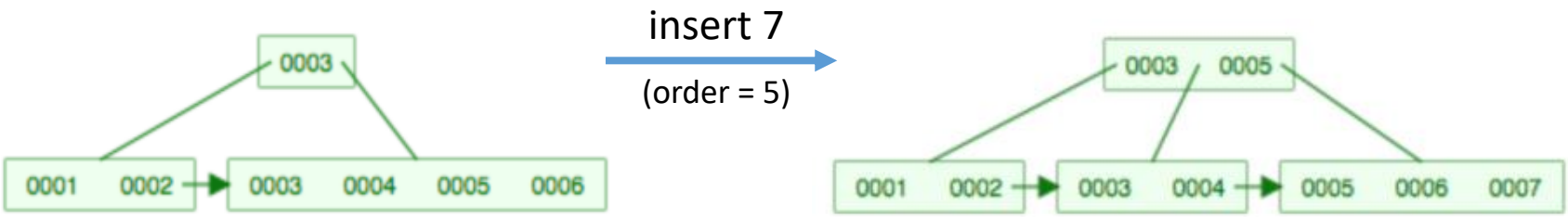
주어진 argument에 따라 order 값을 조정할 수 있습니다
(usage()function을 참고하세요)

key가 1~6까지 sequential하게 insert되었다는 전제 하에
key 7을 insert하면 정해진 order값에 따라
node가 분리된 것을 확인 할 수 있습니다

Basic B+tree

- 주어진 In-memory 형식의 B+tree code를 모두 이해하시면, 이번 프로젝트 On-disk 형식의 B+tree 구조 역시 이해할 수 있을 것 입니다

- B+tree의 동작 과정을 가시적으로 볼 수 있는 사이트를 참고하세요
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html



Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- Implement 4 commands : open / insert / find / delete
- There should be an appropriate **data file** in your system (you can call it a very simple database), maintaining disk-based B+tree after serving those commands.

1. **open <pathname>**
   - Open existing data file using ‘pathname’ or create one if not existed.
   - All other 3 commands below should be handled after open data file.
2. **insert <key> <value>**
   - Insert input ‘key/value’ (record) to data file at the right place.
   - Same key should not be inserted (no duplicate).
3. **find <key>**
   - Find the record containing input ‘key’ and return matching ‘value’.
4. **delete <key>**
   - Find the matching record and delete it if found.

Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- library (libbpt.a) should provide those API servies.

1. **int open_table (char *pathname);**
   - Open existing data file using ‘pathname’ or create one if not existed.
   - If success, return the unique table_id, which represents the own table in this database. Otherwise, return negative value.

2. **int db_insert (int64_t key, char * value);**
   - Insert input ‘key/value’ (record) to data file at the right place.
   - If success, return 0. Otherwise, return non-zero value.

3. **char * db_find (int64_t key);**
   - Find the record contatining input ‘key’.
   - If found matching ‘key’, return matched ‘value’.

4. **int db_delete (int64_t key);**
   - Find the matching record and delete it if found.
   - If success, return 0. Otherwise, return non-zero value.

Disk-based B+tree

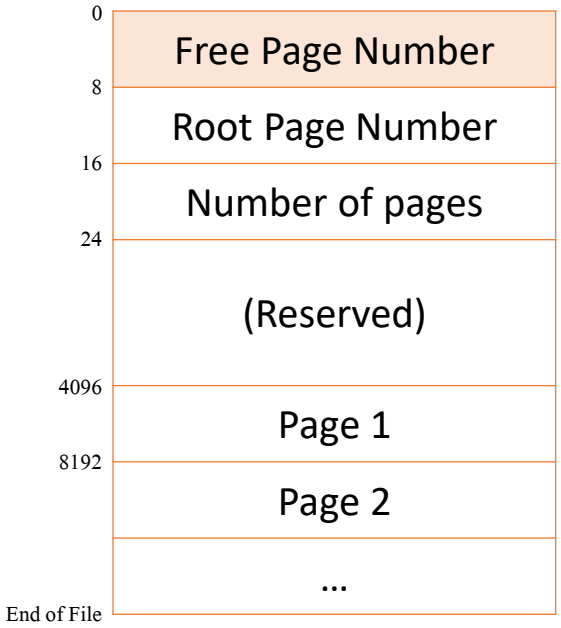
❖ 현재 아래의 명세를 만족하고 있습니다

- All update operation (insert / delete) should be applied to your data file as an operation unit. That means one update operation should change the data file layout correctly.
- Note that your code must be worked as other students' data file. That means, your code should handle open(), find(), insert(), and delete() API with other students' data file as well.
- So follow the data file layout described from next slides.
  - We fixed the on-disk page size with 4096 Bytes.
  - We fixed the record (key + value) size with 128 (8 + 120) Bytes.
    - Type : (key ➔ integer) & (value ➔ string)
  - There are 4 types of page
    1. Header page (special, containing metadata)
    2. Free page (maintained by free page list)
    3. Leaf page (containing records)
    4. Internal page (indexing internal/leaf page)

Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

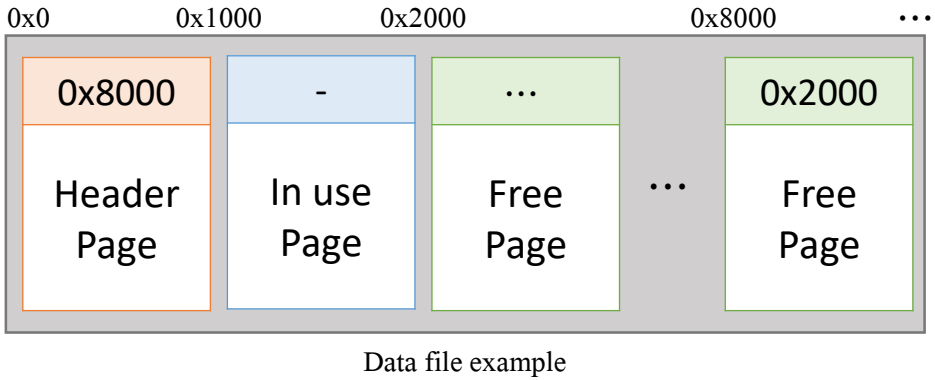
- **Header Page (Special)**
  - Header page is the first page (offset 0-4095) of a data file, and contains metadata.
  - When we open the data file at first, initializing disk-based B+tree should be done using this header page.
    - Free page number : [0-7]
      - points the first free page (head of free page list)
      - 0, if there is no free page left.
    - Root page number : [8-15]
      - pointing the root page within the data file.
    - Number of pages : [16-23]
      - how many pages exist in this data file now.



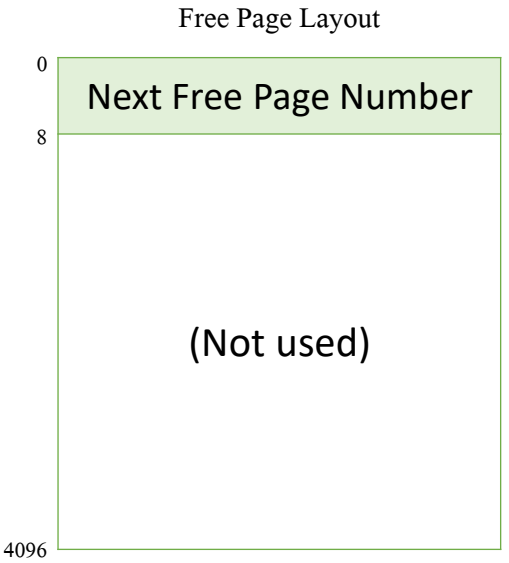
Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- **Free Page**
  - In previous slide, header page contains the position of the first free page.
  - Free pages are linked and allocation is managed by the free page list.
    - Free page number : [0-7]
      - points the first free page (head of free page list)
      - 0, if there is no free page left.



Data file example



Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

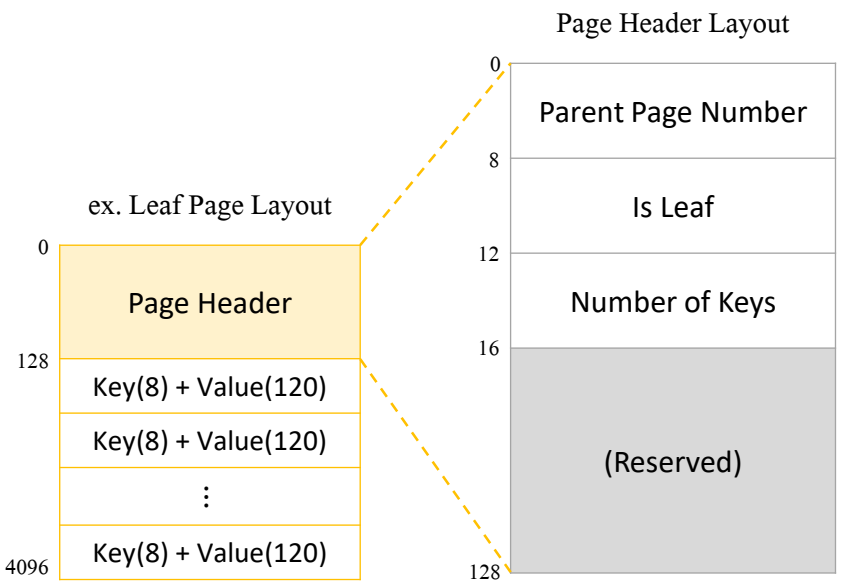
- **Page Header**
  - Leaf / Internal page have first 128 bytes as a page header.
  - Leaf / Internal page should contain those data

    (see the *node* structure in include/bpt.h)
    - Parent page number : [0-7]

      - If Leaf / internal page, this field points the position of parent page.
    - Is Leaf : [8-11]

      - 0 is internal page, 1 is leaf page.
    - Number of keys : [12-15]

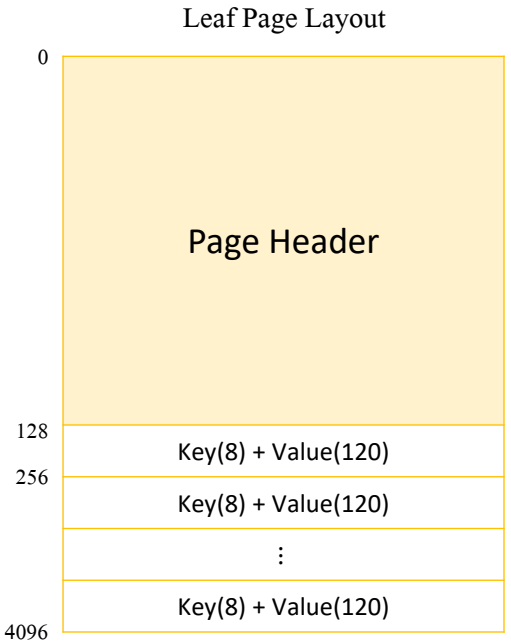
      - the number of keys within this page.



Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- **Leaf Page**
  - Leaf page contains the key/value records. (Keys are sorted in the page)
  - One record size is 128 bytes and we contain maximum 31 records per one data page.
  - First 128 bytes will be used as a page header for other types of pages.
  - Branching factor (order) = 32

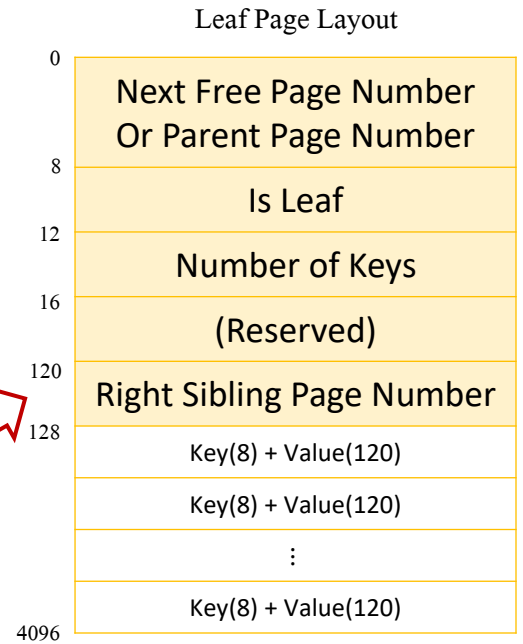


Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- **Leaf Page**
  - Leaf page contains the key/value records. (Keys are sorted in the page)
  - One record size is 128 bytes and we contain maximum 31 records per one data page.
  - First 128 bytes will be used as a page header for other types of pages.
  - Branching factor (order) = 32
    - ➢ We can say that the order of leaf page in disk-based B+tree is 32, but there is a minor problem.
    - ➢ There should be one more page number added to store right sibling page number for leaf page.

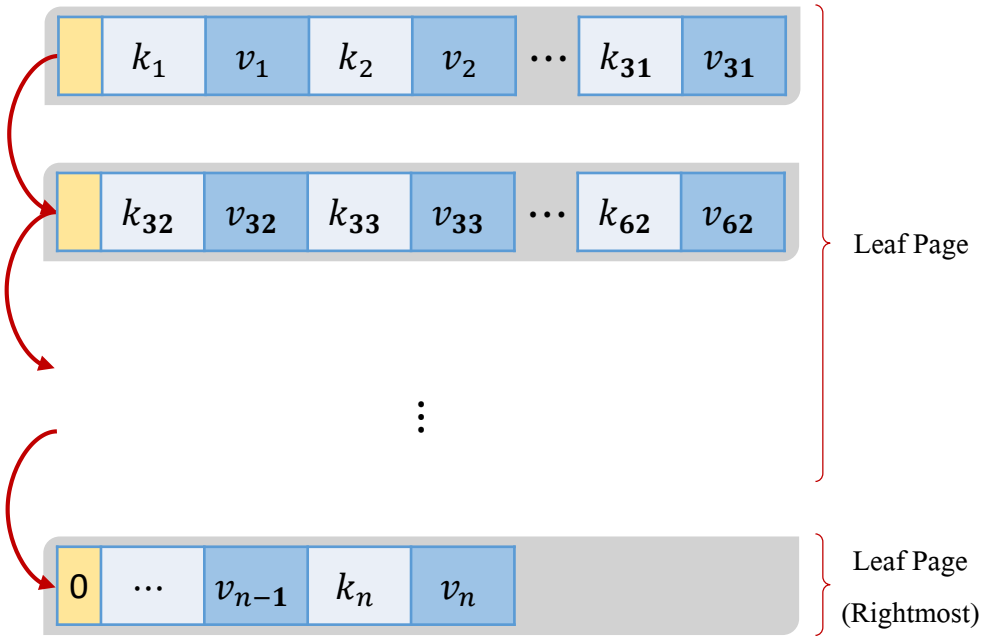
      (see the comments of $node$ structure in include/bpt.h)
    - ➢ So we define one special page number at the end of page header.
    - ➢ If rightmost leaf page, right sibling page number field is 0.

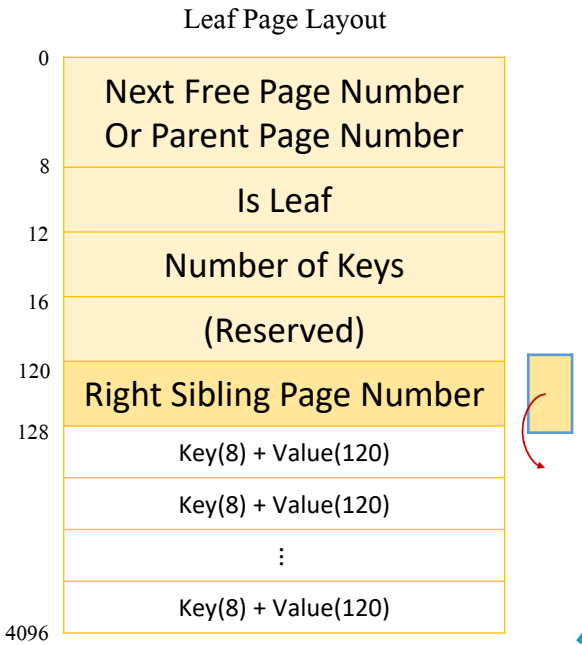


Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- **Leaf Page**

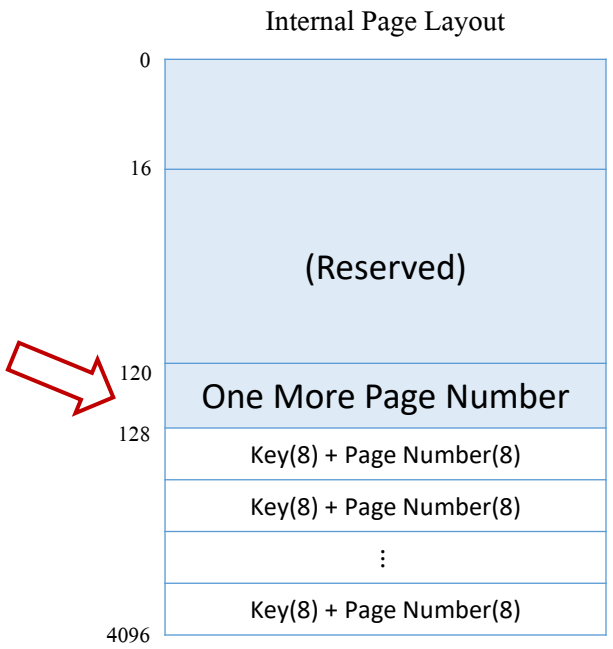




Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

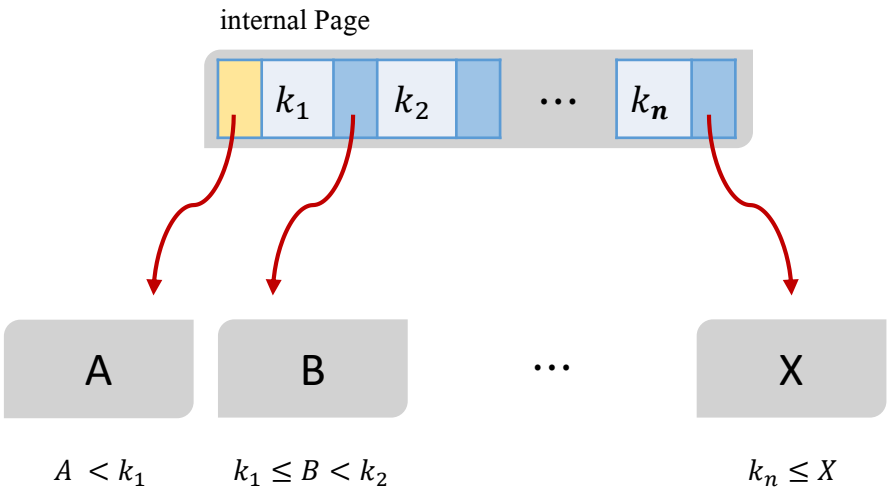
- **Internal Page**
  - Internal page is similar to leaf page, but instead of containing 120 bytes of values, it contains 8 bytes of another page (internal or leaf) number.
  - Internal page also needs one more page number to interpret key ranges and we use the field which is specially defined in the leaf page for indicating right sibling.
  - Branching factor (order) = 249
    - Internal page can have maximum 248 entries (We learned in LAB class)
    - (4096 - 128) / (8+8) = 248.

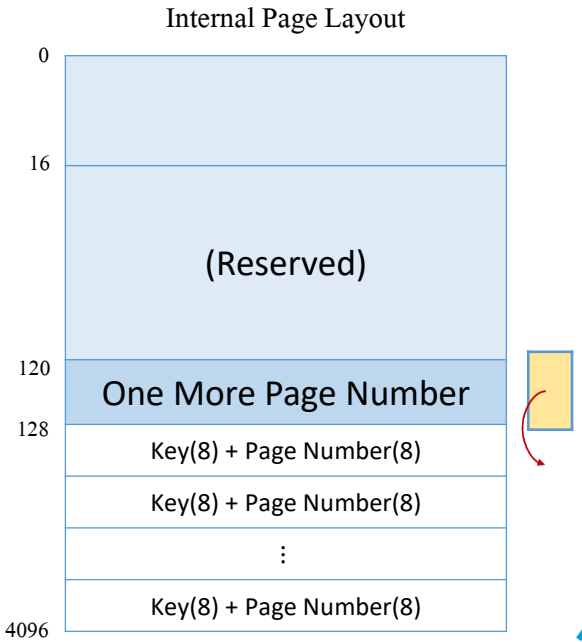


Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- **Internal Page**





Disk-based B+tree

❖ 현재 아래의 명세를 만족하고 있습니다

- Implement those APIs for managing a database file and synchronizing between in-memory pages and on-disk pages.

```
typedef uint64_t pagenum_t;

struct page_t {
        // in-memory page structure
};

// Allocate an on-disk page from the free page list
pagenum_t file_alloc_page();

// Free an on-disk page to the free page list
void file_free_page(pagenum_t pagenum);

// Read an on-disk page into the in-memory page structure(dest)
void file_read_page(pagenum_t pagenum, page_t* dest);

// Write an in-memory page(src) to the on-disk page
void file_write_page(pagenum_t pagenum, const page_t* src);
```