Project2

Disk Based B+Tree

B+Tree

- B+Tree source code (originated http://www.amittai.com/prose/bpt.c)
  - Download the basic source code uploaded in piazza (bpt.zip)
  - Unzip the file.

```
[hyeongwon@dev project]$ ls
bpt.zip
[hyeongwon@dev project]$ unzip bpt.zip
Archive:  bpt.zip
   creating: bpt/include/
  inflating: bpt/include/bpt.h
   creating: bpt/lib/
  inflating: bpt/Makefile
   creating: bpt/src/
  inflating: bpt/src/bpt.c
  inflating: bpt/src/main.c
[hyeongwon@dev project]$ ls
bpt  bpt.zip
```

B+Tree

- B+Tree soruce code (originated http://www.amittai.com/prose/bpt.c)
  - Compile the source file using ‘Makefile’
    - If you don’t have make util, type ‘sudo apt-get install make’
    - Don’t change the makefile unless you add another source file.
    - **If Makefile doesn’t make libbpt.a library file at the exact path, you’ll get zero score. (your_git_repo/project2/lib/libbpt.a)**
  - Your project hierarchy will be like this.
    - Your_git_repo
      - project2
        - include /
        - lib /
        - Makefile
        - src /

Basic trial

- Execute it and try some basic commands.

```
$ ./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 |
>
```

you can type commands like this and see the results!

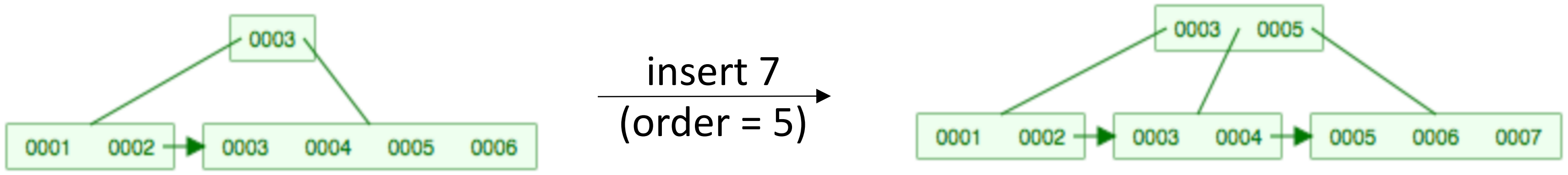
Basic trial

- You can adjust order by giving argument. (see the usage() functions)

```
$ ./main 5
…
3 |
1 2 | 3 4 5 6 |
> i 7
3 5 |
1 2 | 3 4 | 5 6 7 |
>
```

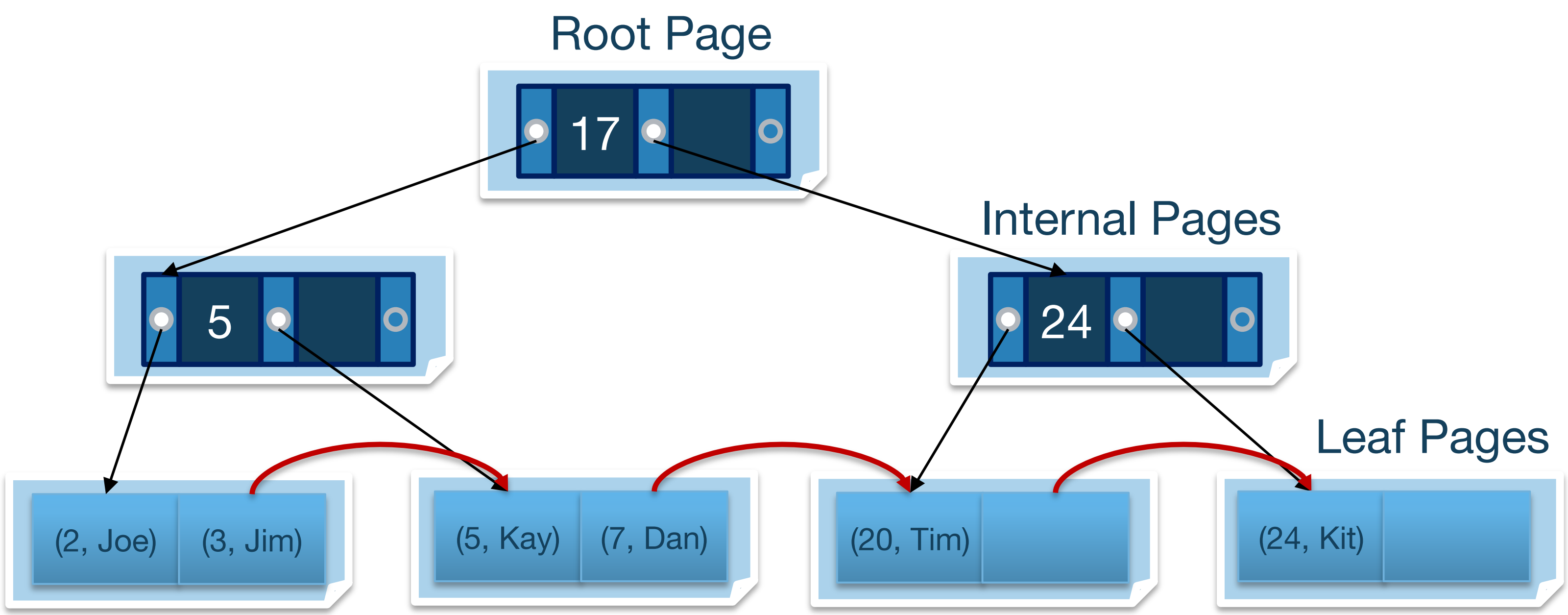
insert key 7 after inserting key 1 to 6 (sequentially)

- You'd better understand the code fully before implementing the project.
- You can get some help from https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html



Disk-based B+tree

- Note that current design only considers in-memory b+tree.
- Our goal
  1. Implement **disk-based** b+ tree. (like below example)
  2. Delayed merge (details in the next slide)



Project Specification

- 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.

A "record" means a <key/value> pair

Project Specification

- Your library (libbpt.a) should provide those API services.
  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. (This table id will be used for future assignment.)
  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. **int db_find (int64_t key, char * ret_val);**
     - Find the record containing input ‘key’.
     - If found matching ‘key’, store matched ‘value’ string in ret_val and return 0. Otherwise, return non-zero value.
     - Memory allocation for record structure(ret_val) should occur in caller function.
  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.

Project Specification

- 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(), insert(), find() and delete() API with other students' data file as well.
- So **follow the data file layout** described from next slides.

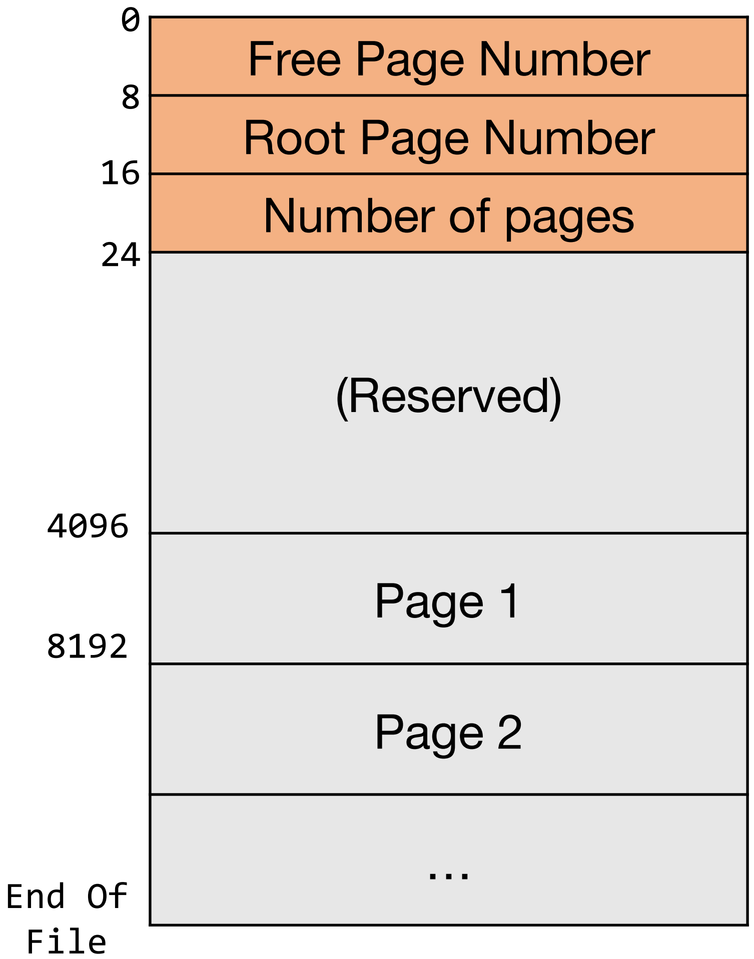
Project Specification

- ➢ 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. (detail next slides..)
  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)

Header Page (Special)

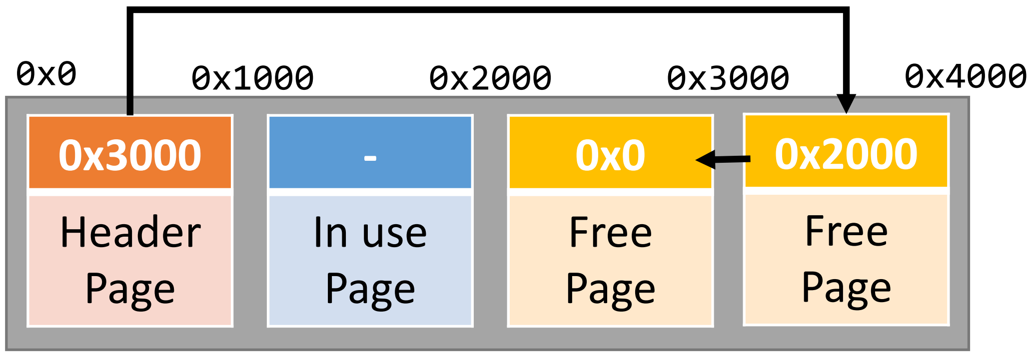
- ➢ Header page is the **first page (offset 0-4095)** of a data file, and contains metadata.
- ➢ When we open the data file at first, initializing disk-based b+tree should be done using this header page.

- 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.

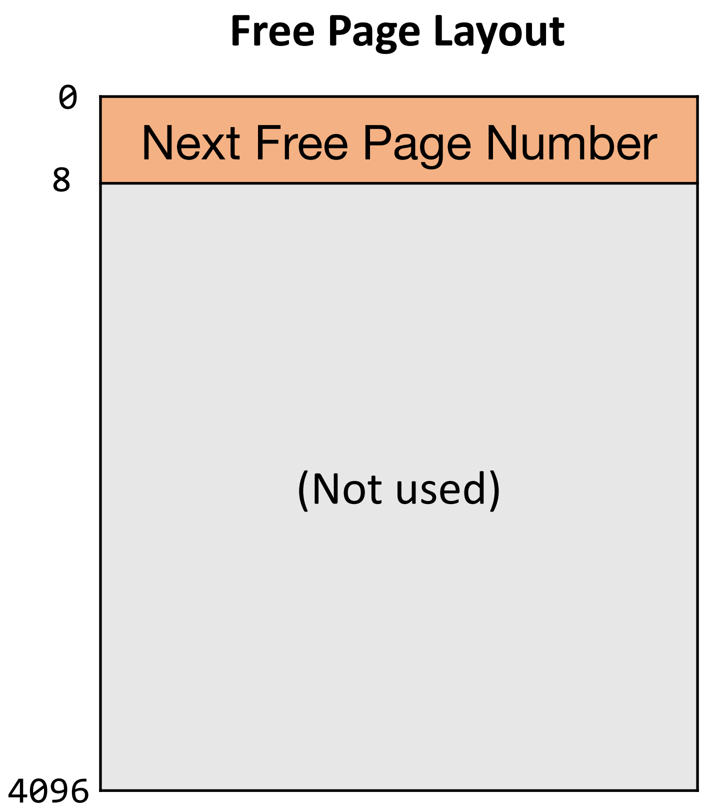


Free Page

- ➢ 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.
- Next free page Number: [0-7]
  - points the next free page.
  - 0, if end of the free page list.



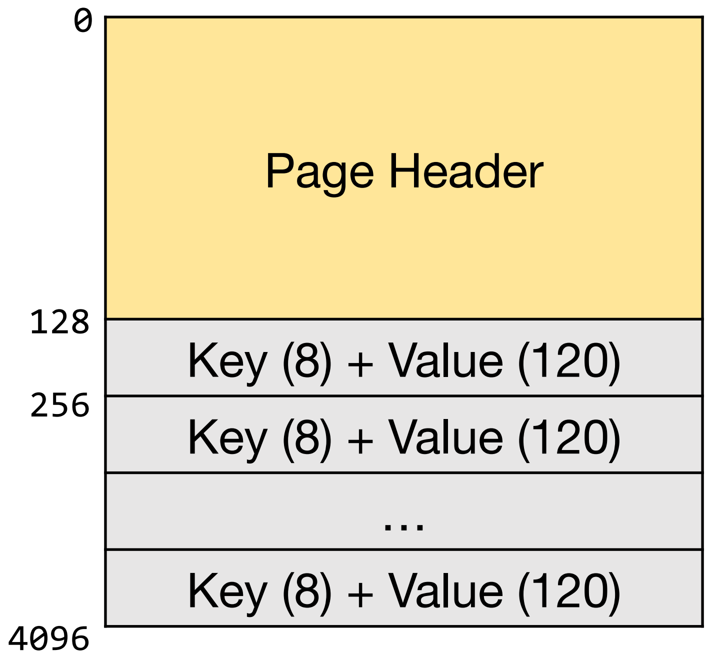
Data file example



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. (see next slides)
- Branching factor (order) = 32

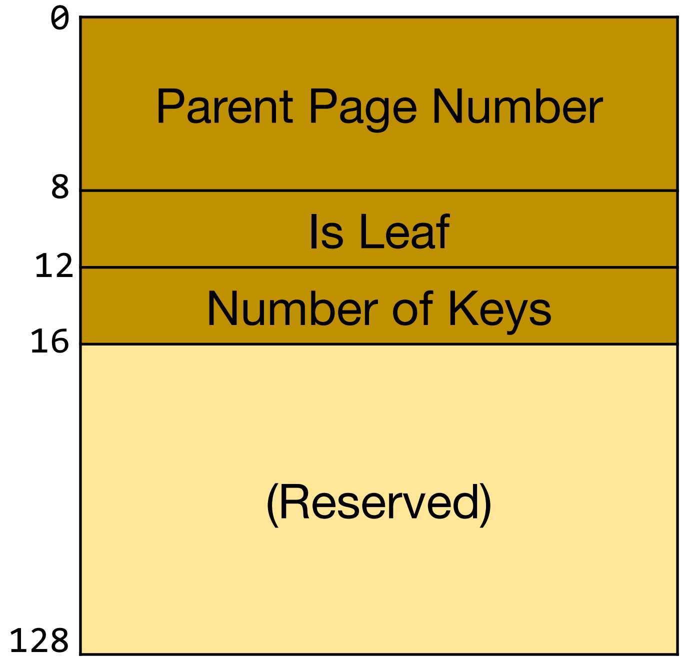
Leaf Page Layout



Page Header

- Internal/Leaf 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 internal/leaf 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.

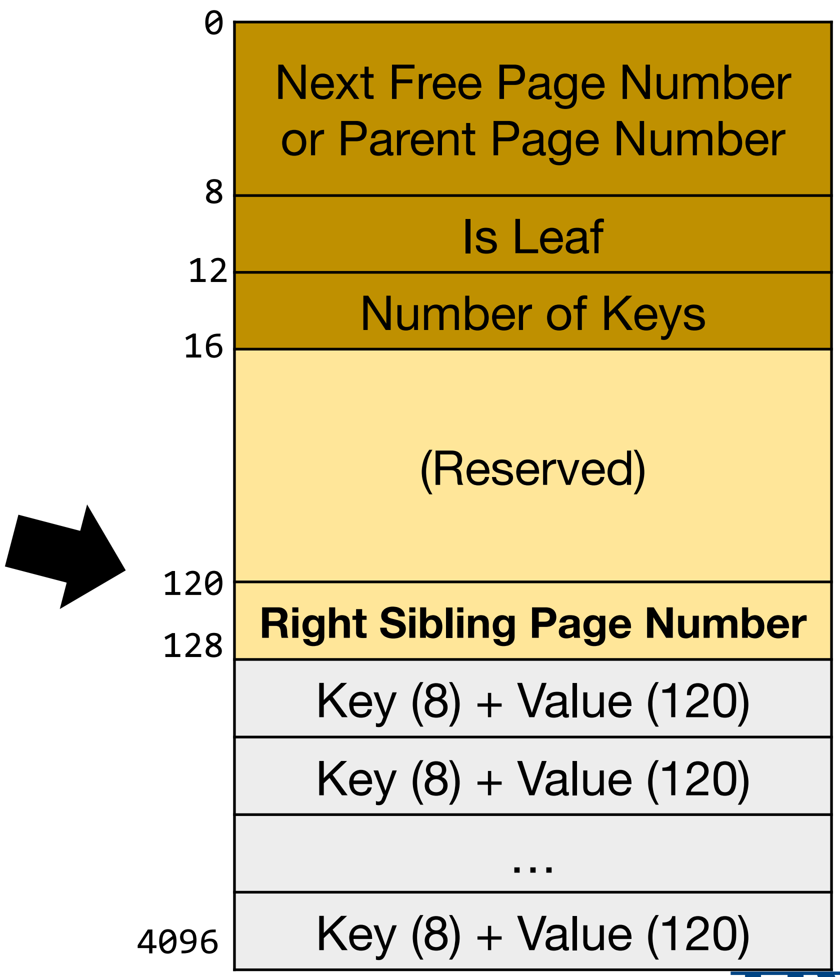
Page Header Layout



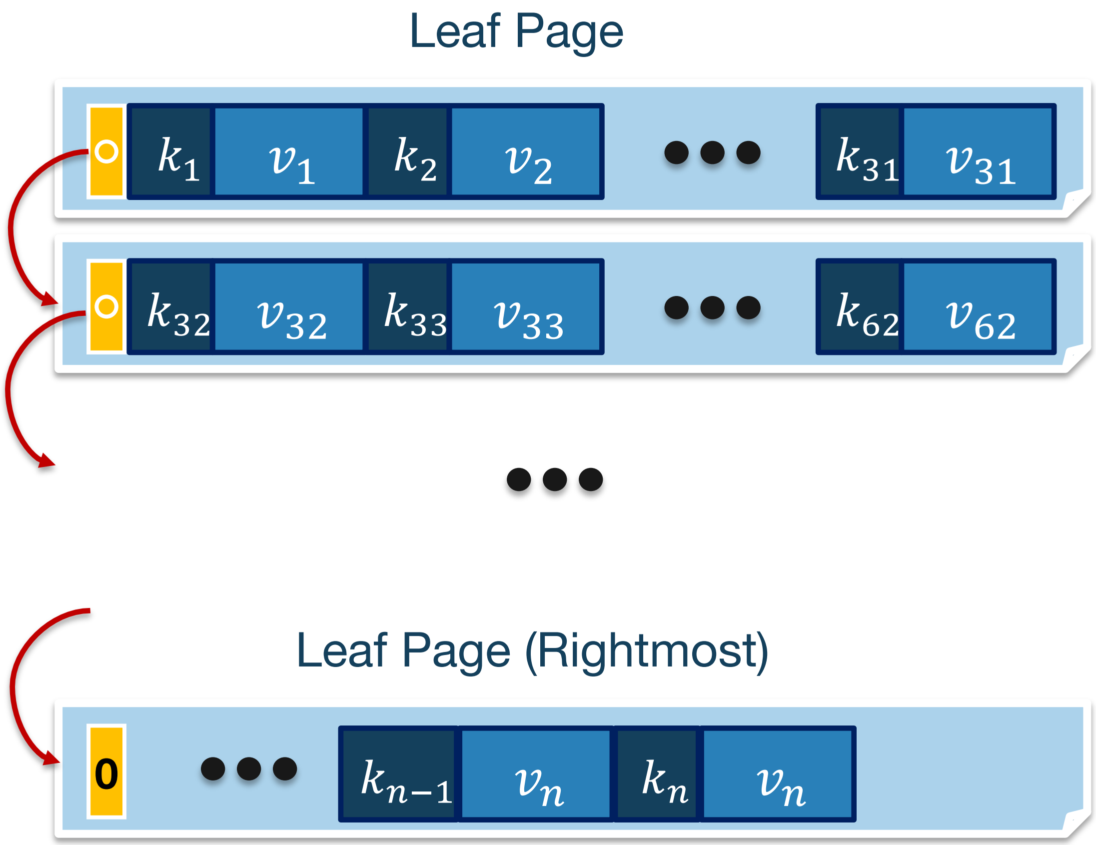
Leaf Page (Cont.)

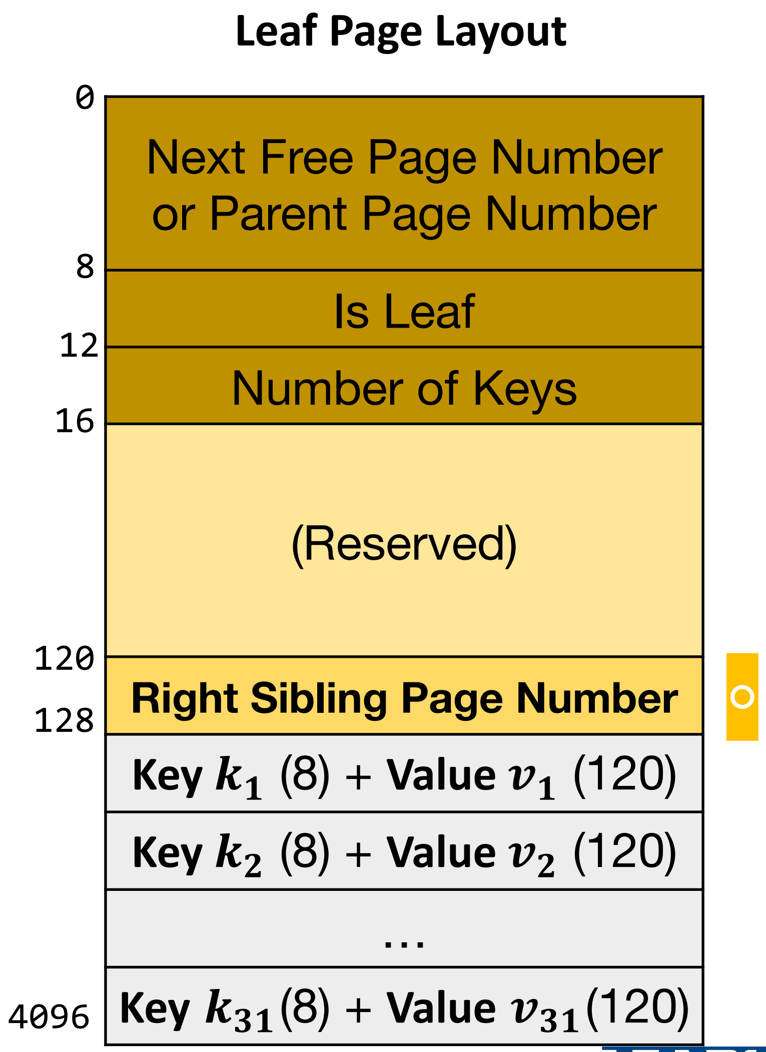
- We can say that the order of leaf page in disk-based b+tree is 32, but there is a minor problem.
- There should be one more page number added to store right sibling page number for leaf page. (see the comments of *node* structure in include/bpt.h)
- So we define one special page number at the end of page header.
- If rightmost leaf page, right sibling page number field is 0.

Leaf Page Layout



Leaf Page (Cont.)

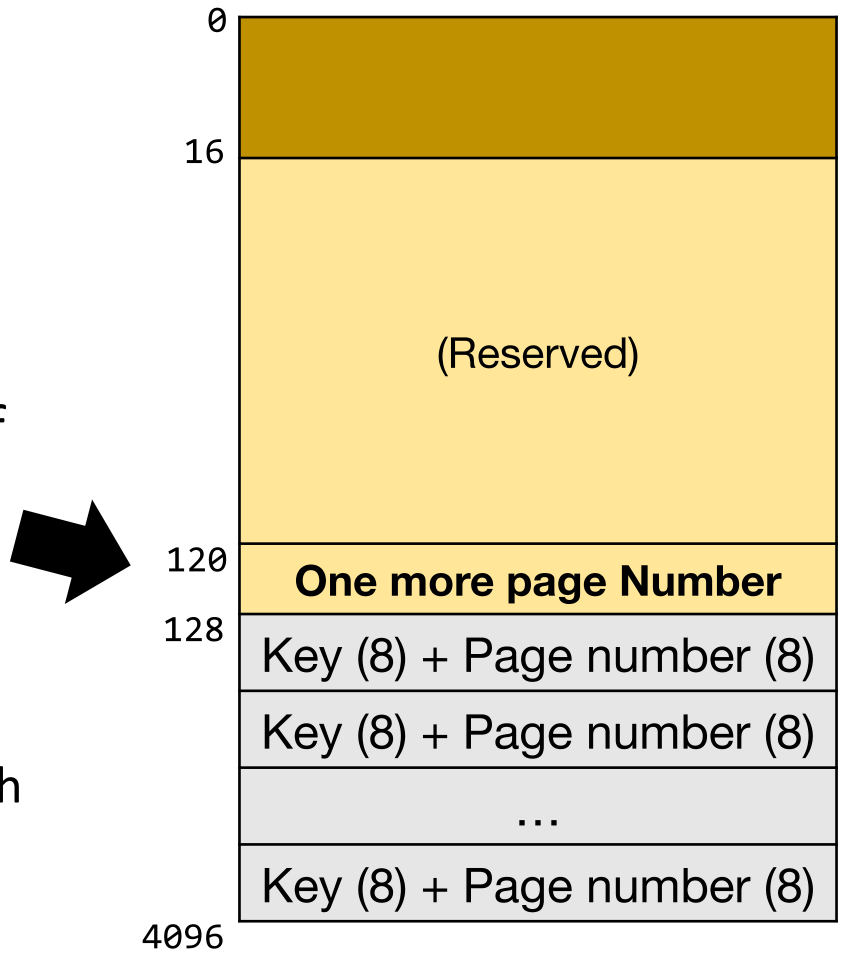




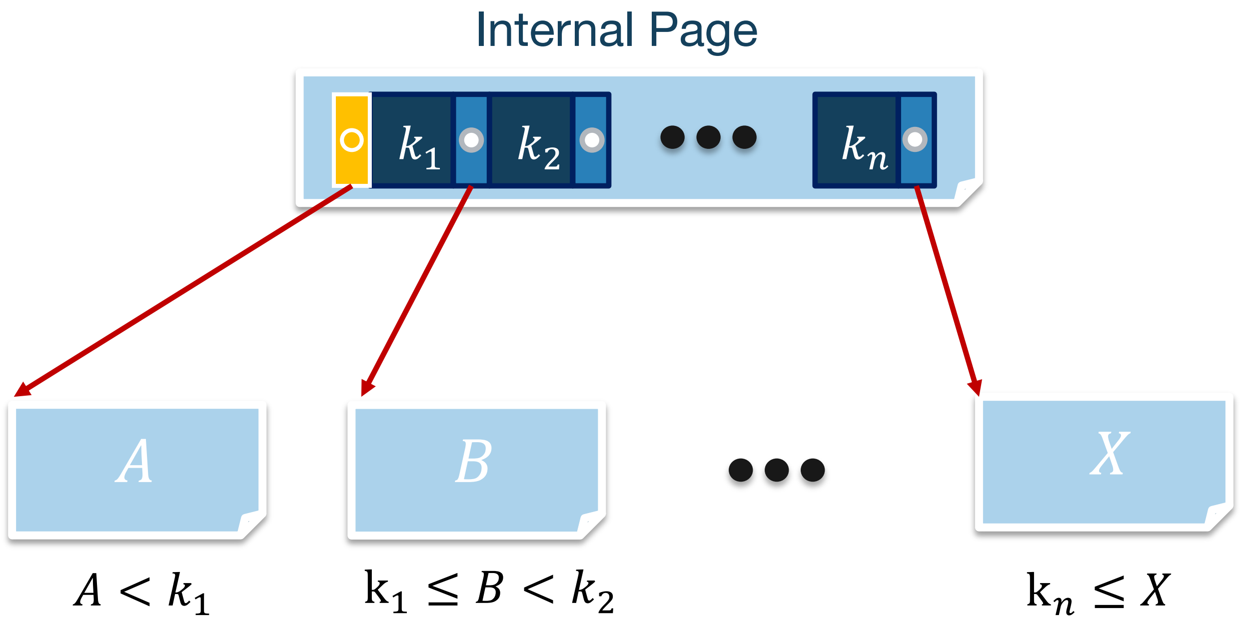
Internal Page

- 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.
- Branch factor (order) = 249
  - Internal page can have <u>maximum 248 entries</u>, because ‘key + page number’ (8+8 bytes) can cover up to whole page (except page header) with the number of 248.
  - (4096 – 128) / (8+8) = 248

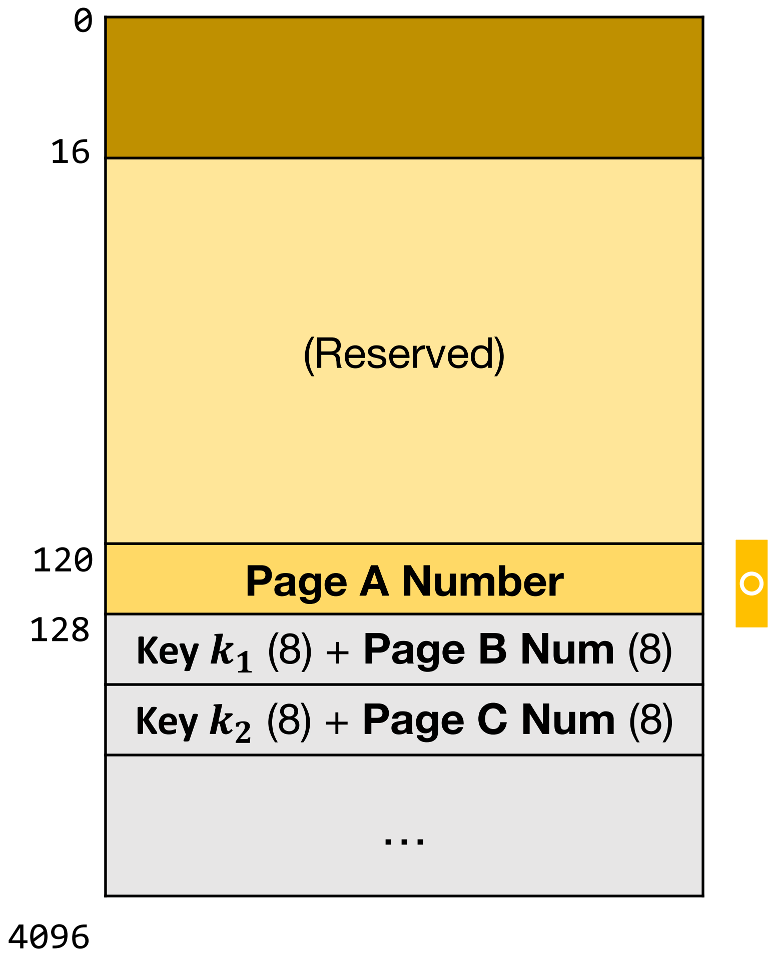
Internal Page Layout



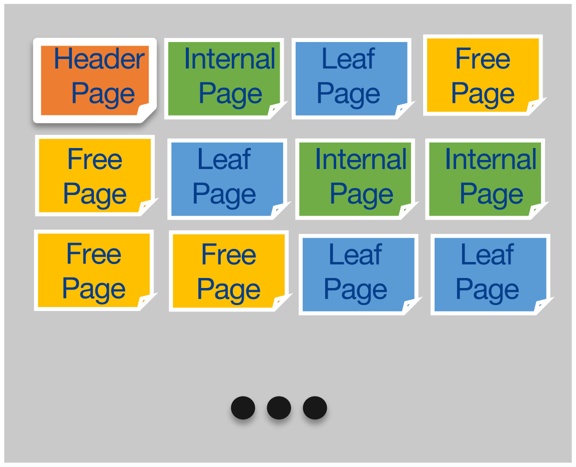
Internal Page



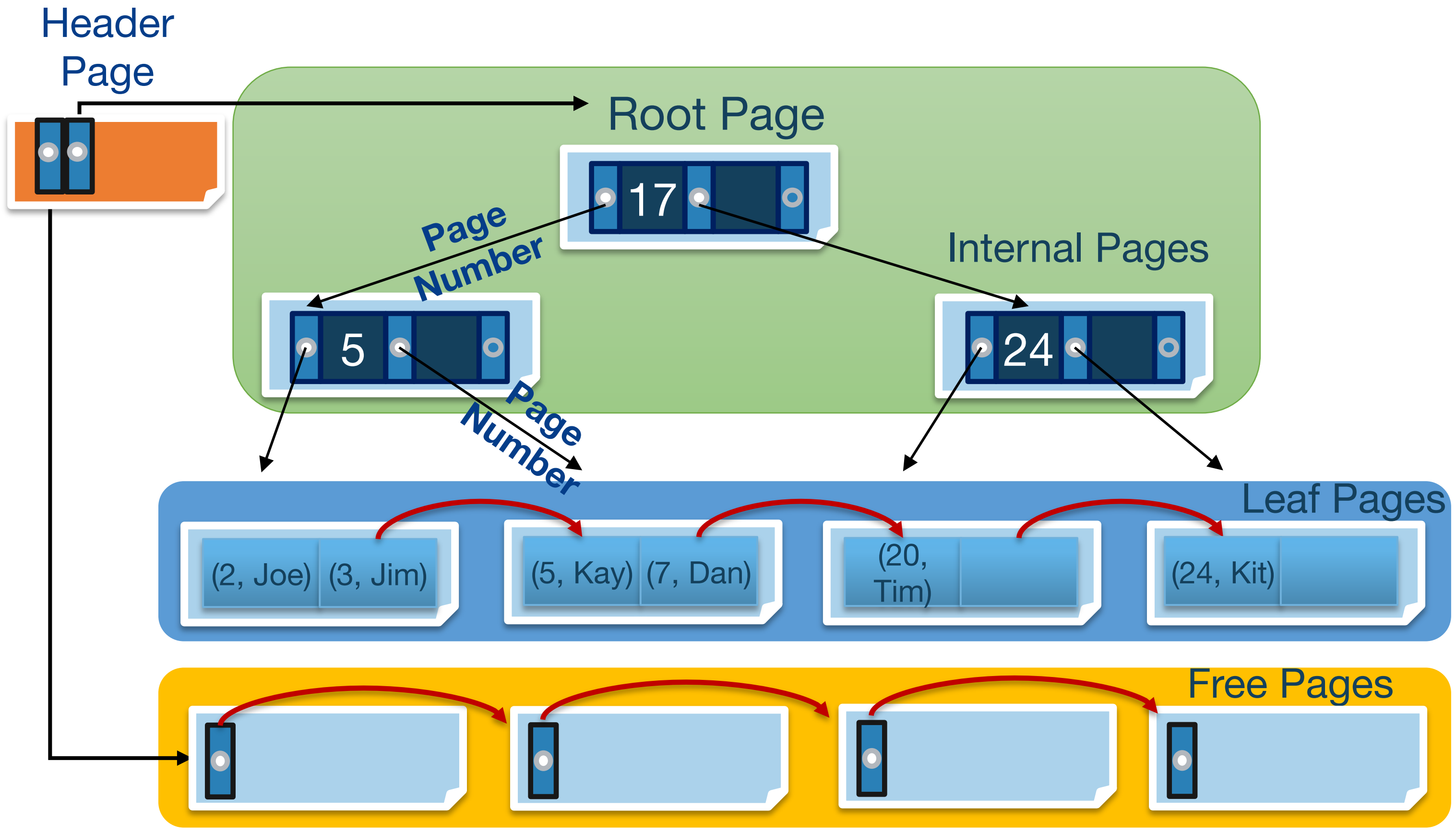
Internal Page Layout



Disk-based B+tree Example



Your Data File (Example)



File manager API

- We **strongly recommand** you to implement those APIs for managing a database file and synchronizing between in-memory pages and on-disk pages. **If you violate this layered architecture you will get zero score in this project.**

```
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);
```

Delayed Merge

- Structure modification(Split, Merge) incurs heavy disk I/O, resulting in performance degradation.
- One way to reduce it (and you should implement): **Delayed Merge**
- Delayed Merge
  - Do not merge a page(leaf, internal) until all keys in the page have deleted, regardless of the branching factor.

Milestone & DEADLINE

- Milestone 1
  - Analyze the given b+ tree code and submit a report to the hconnect Wiki.
  - Your report should includes
    1. Possible call path of the insert/delete operation
    2. Detail flow of the structure modification (split, merge)
    3. (Naïve) designs or required changes for building on-disk b+ tree
  - Deadline: Sep 30 11:59pm
- Milestone 2
  - Implement on-disk b+ tree and submit a report(Wiki) including your design.
  - Deadline: Oct 13 11:59pm
- We'll only score your commit before the deadline and your submission after that deadlines will not accepted.