

# 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

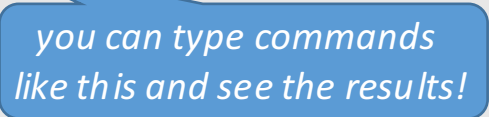
- B+Tree source code (originated <http://www.amittai.com/prose/bpt.c>)
  - Compile the source file using 'Makefile'
    - If you don't have make util, type 'sudo apt-get install make'
    - Don't change the makefile unless you add another source file.
  - You will see the executable file (main) like below.

```
[hyeongwon@dev project]$ cd bpt
[hyeongwon@dev bpt]$ ls
include lib Makefile src
[hyeongwon@dev bpt]$ make
gcc -g -fPIC -I include/ -c -o src/main.o src/main.c
gcc -g -fPIC -I include/ -o src/bpt.o -c src/bpt.c
make static_library
make[1]: Entering directory `/home/hyeongwon/workspace/db.class/project/bpt'
ar cr lib/libbpt.a src/bpt.o
make[1]: Leaving directory `/home/hyeongwon/workspace/db.class/project/bpt'
gcc -g -fPIC -I include/ -o main src/main.o -L lib/ -lbpt
[hyeongwon@dev bpt]$ ls
include lib main 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
```

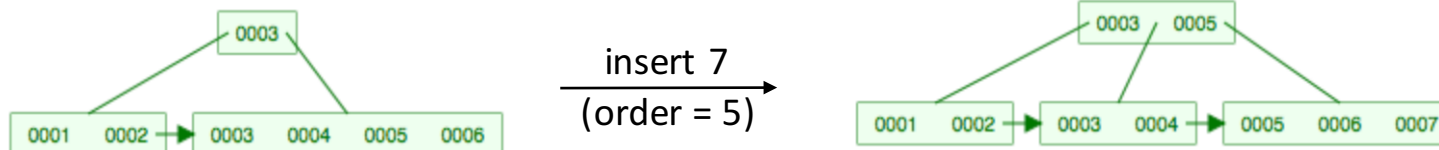
```
3 5 |
```

```
1 2 | 3 4 | 5 6 7 |
```

```
>
```

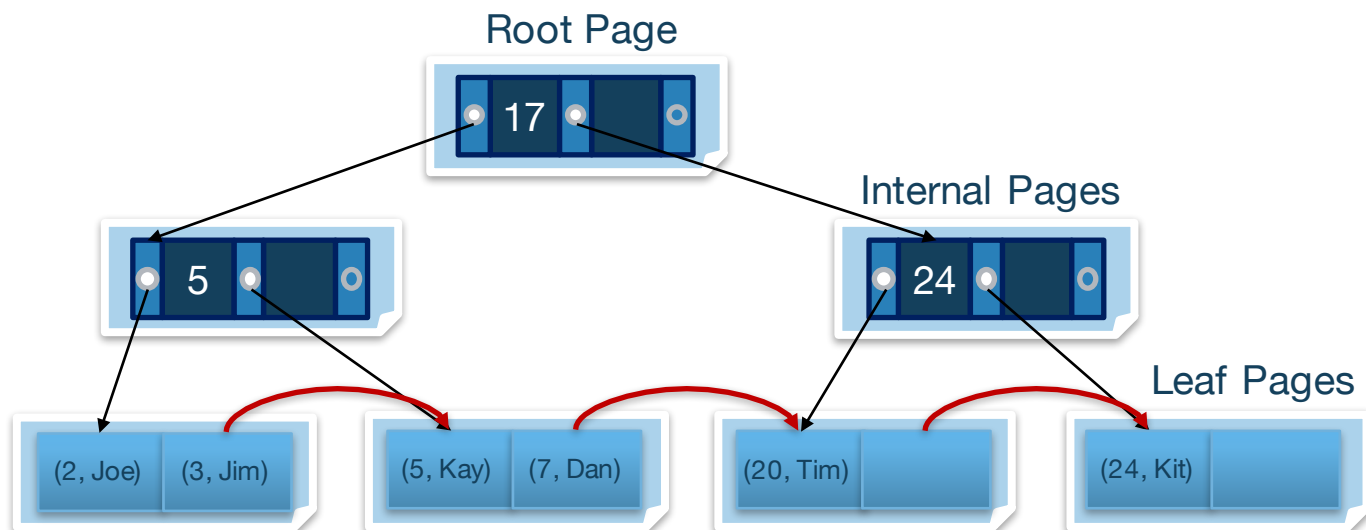
*insert key 7 after inserting  
key 1 to 6 (sequentially)*

- 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 is to implement **disk-based** b+ tree. (like below example)



# 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' or 'null' if not found.
  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 (char \*pathname);**

- Open existing data file using 'pathname' or create one if not existed.
- If success, return 0. Otherwise, return non-zero value.

**2. int 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 \* find (int64\_t key);**

- Find the record containing input 'key'.
- If found matching 'key', return matched 'value' string. Otherwise, return NULL.

**4. int 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.

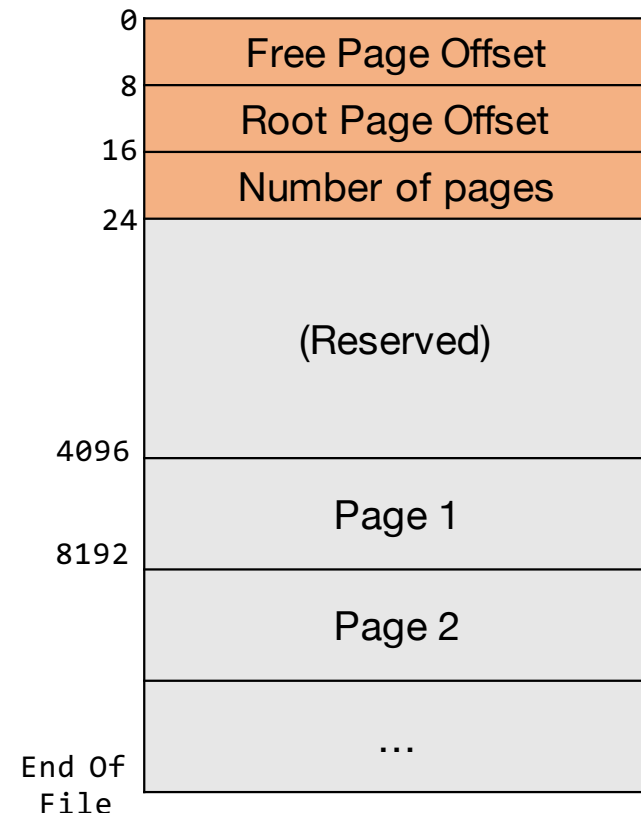
# Project Specification

---

- 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)

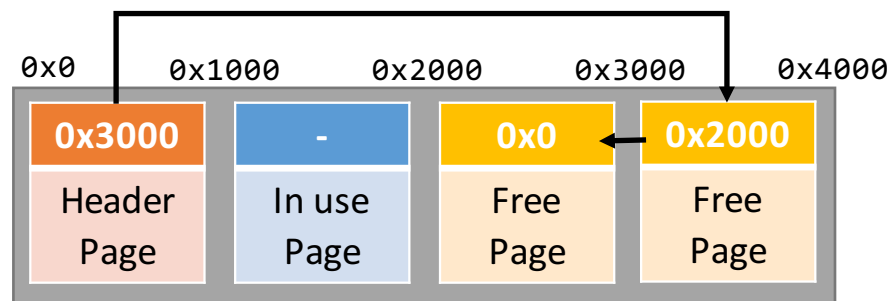
# Header Page (Special)

- 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 offset: [0-7]
  - points the first free page (head of free page list)
  - 0, if there is no free page left.
- Root page offset: [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

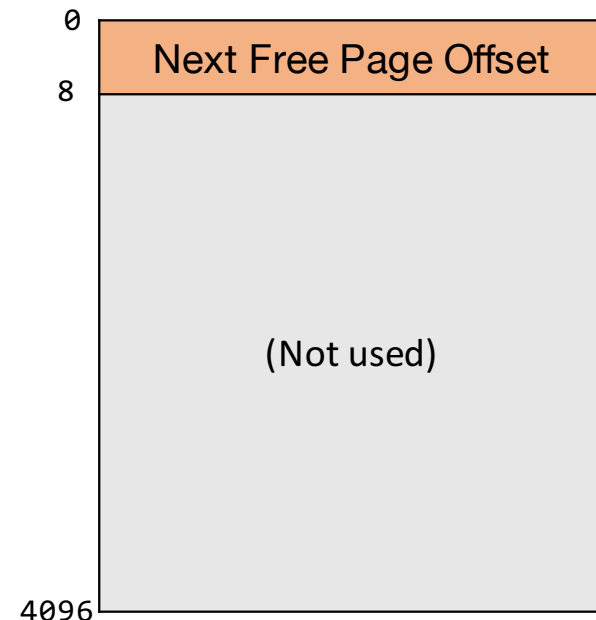
- In previous slide, header page contains the position of the first free page.
- Free pages are linked and allocation is managed by the free page list.
- Next free page offset: [0-7]
  - points the next free page.
  - 0, if end of the free page list.



Data file example

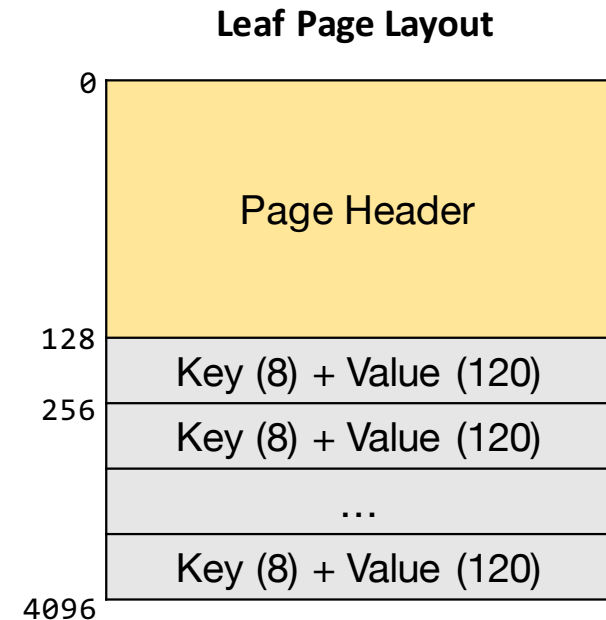
Scalable Computing Systems Laboratory  
Hanyang University

Free Page Layout



# Leaf Page

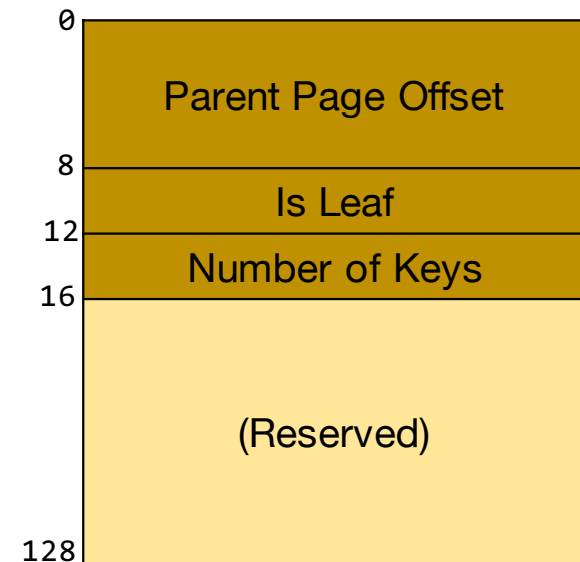
- Leaf page contains the **key/value records**.
- Keys are sorted in the page.
- One record size is 128 bytes and we contain maximum 31 records per one data page.
- First 128 bytes will be used as a page header for other types of pages. (see next slides)
- Branch factor (order) = 32



# Page Header

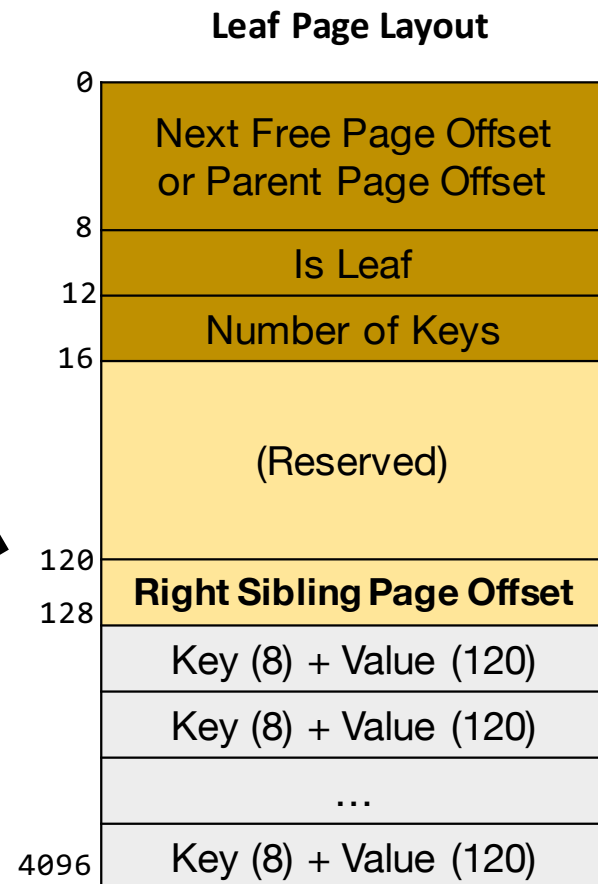
- Internal/Leaf page have **first 128 bytes** as a page header.
- Leaf/Internal page should contain those data (see the *node* structure in include/bpt.h)
  - Parent page offset [0-7]: If internal/leaf page, this field points the position of parent page.
  - Is Leaf [8-11] : 0 is internal page, 1 is leaf page.
  - Number of keys [12-15] : the number of keys within this page.

Page Header Layout



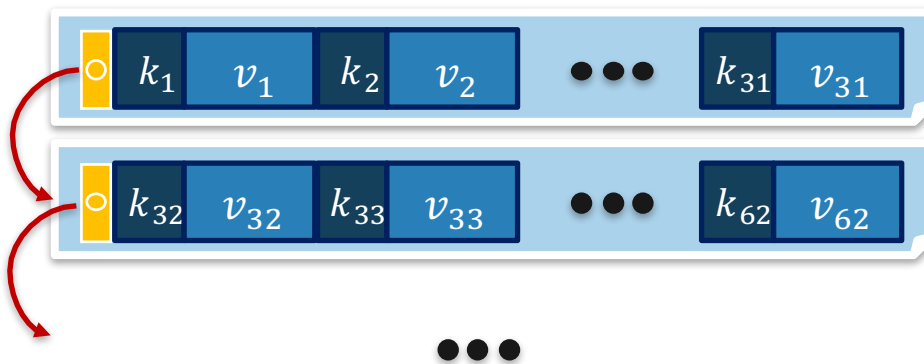
# Leaf Page (Cont.)

- 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 offset added to store right sibling page offset for leaf page. (see the comments of *node* structure in include/bpt.h)
- So we define one special offset at the end of page header.
- If rightmost leaf page, right sibling page offset field is 0.



# Leaf Page (Cont.)

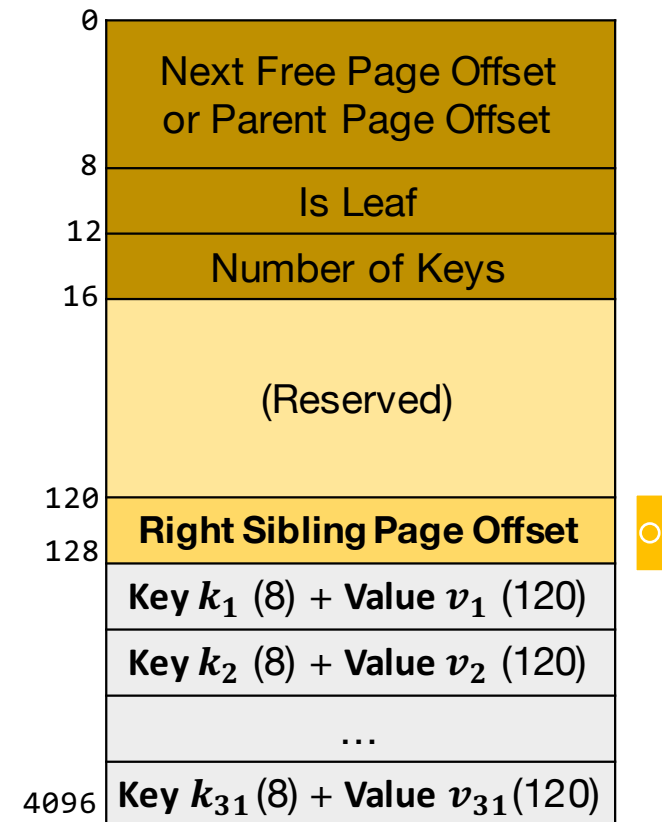
Leaf Page



Leaf Page (Rightmost)



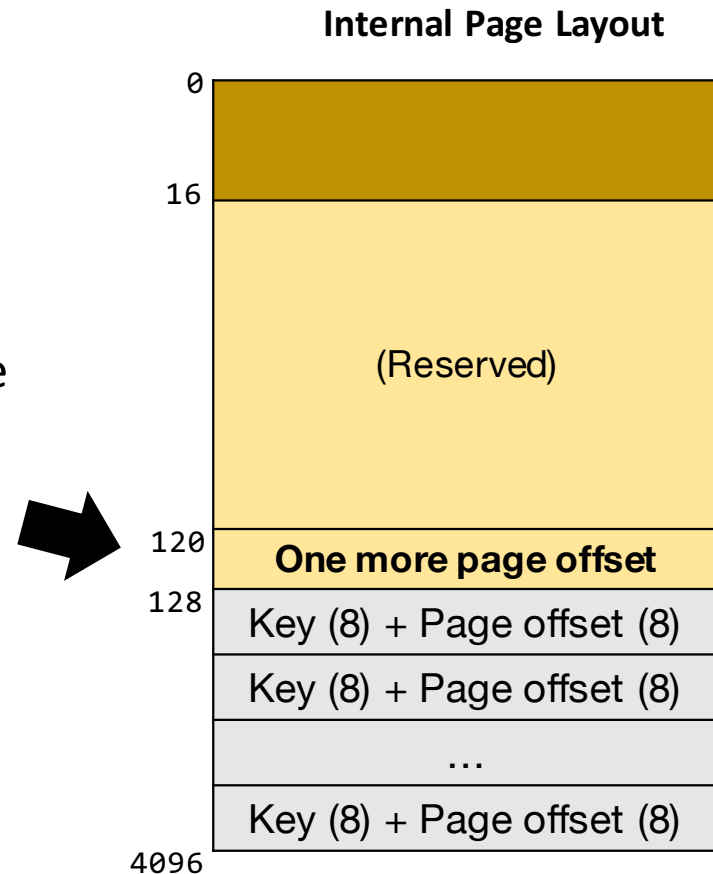
Leaf Page Layout



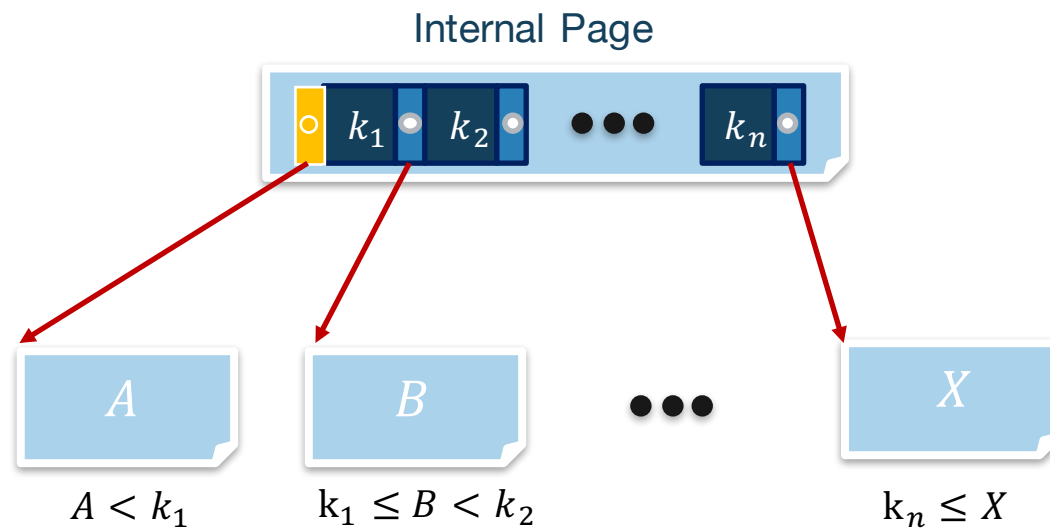


# Internal Page

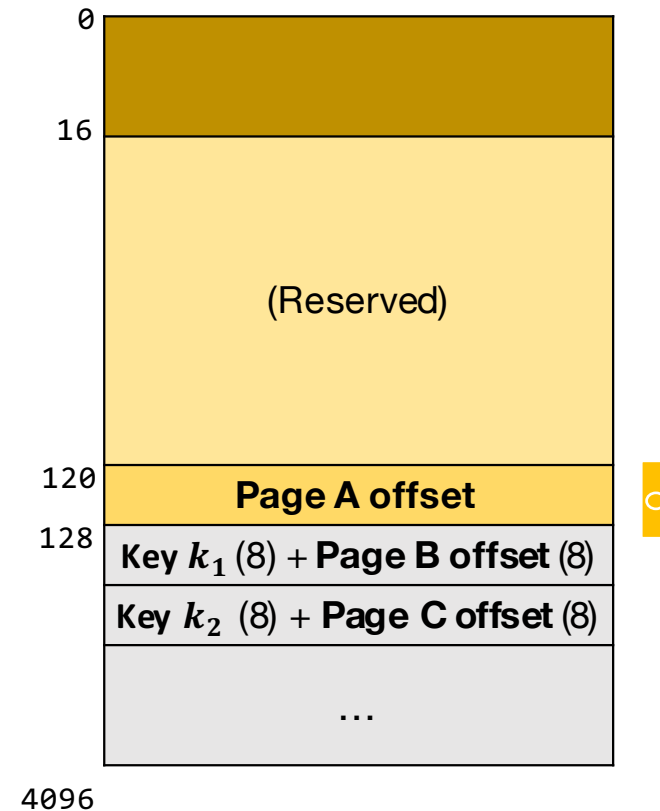
- Internal page is similar to leaf page, but instead of containing 120 bytes of values, it contains 8 bytes of another page (internal or leaf) offset.
- Internal page also needs one more page offset to interpret key ranges and we use the field which is specially defined in the leaf page for indicating right sibling.
- Branch factor (order) = 249
  - Internal page can have maximum 248 entries, because 'key + page offset' (8+8 bytes) can cover up to whole page (except page header) with the number of 248.
  - $(4096 - 128) / (8 + 8) = 248$



# Internal Page



Internal Page Layout



# Disk-based B+tree Example

