Project 2

Due date: April 15, 2019, 11:55pm IST

1 Introduction

In this project, you will implement three different functions on top of the buffer manager provided to you.

1. **Binary search**: Given an integer, find whether the integer is present in the file using binary search.
2. **Insertion**: Given an integer, insert the integer into the file, while maintaining sorted order of integers in the file.
3. **Merge sort**: Given an unsorted file of integers, perform merge sort and create a new file that contains the integers in sorted order.

While all of you would have studied and implemented these algorithms in your data structures course, the key differences between the implementation your are being asked to do now are the following:

- You can no longer assume that your data is in memory.
- Access to the data is only through a very specific API, provided to you by our implementation of a buffer manager.

1.1 Configuration

1. `g++ compiler version`: 8.2.1. Requires C++ 11 standard.
2. We have tested on: Ubuntu 16.04 LTS
3. Your code should correctly compile and run on: Ubuntu 16.04 LTS
4. Only the standard C++ libraries (including STL) are allowed.

2 File Manager

The buffer manager implementation is available at `https://git.iitd.ac.in/cs5150292/COL362632_Project2`. It consists of two main files: `buffer_manager.h/cpp` and `file_manager.h/cpp` along with a bunch of supporting functions in various other files (look at the documentation for details). In this Section, we will briefly describe the functionality provided in the `file_manager`.
Important note: Your access to the data file is *solely* through the functions provided by the `file_manager`. *Do not* directly use any functions from `buffer_manager`. Some useful constants are defined the file `constants.h`.

2.1 Structure of the data file

Since we have not implemented a separate record manager, it is up to you to implement one if you like. Note that the only record type we have is integers. The way in which integers are stored are shown in Figure 2.1. The contents of a page are defined by the following parameters:

1. **Size of the page**: The constant `PAGE_CONTENT_SIZE` defined in `constants.h`.

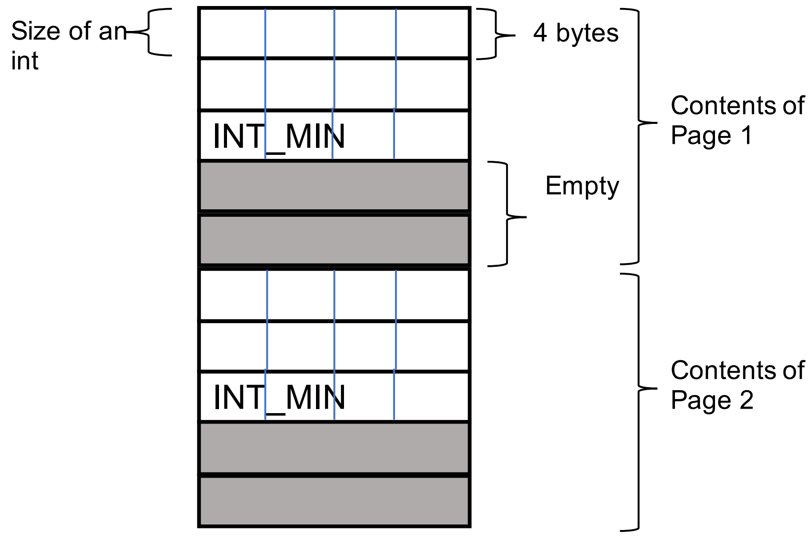

Figure 1: Two pages of size 20B, containing 3 integers (including INT_MIN), with occupancy fraction 0.6

2. **Occupancy fraction**: Occupancy fraction decides how much of a page is occupied and how much is left empty[1]. By default, this is 1, which means that all available space is packed with integers.

Further, any file that is supplied as a test case or that you create as the output of your code has to strictly adhere to the following format.

1. Integers occupy `sizeof(int)` space. This is typically 4 bytes in most systems.
2. Integers are always packed from the beginning of the page. Therefore, the empty space is contiguous and starts after the last integer in the page until the end of the page.
3. There is no header in the page that tells you where to find the free space. Instead, the last integer in the page is indicated by the constant `INT_MIN` (the minimum integer expressible in the system). That is, the integer that occurs just before `INT_MIN` is the last integer in the data that is stored in this page.
4. Only the last page in the file may have less integers than what the occupancy fraction specifies.
5. The number of integers per page *includes* `INT_MIN`.
6. Your algorithms *will not* need to maintain any occupancy requirements.

3 File manager API

A typical usage of the file manager API can be found in `sample_run.cpp`. Please pay attention to how integers are stored and retrieved from a page, since the same procedure has to be followed in order to generate output files from your programs. Alternate ways of storing integers will result in erroneous reads from our testing software.

1. Go through the File Manager API in detail and understand the use of functions such as `MarkDirty()` and `UnpinPage()`. These functions are essential to ensure that the buffer manager is able to evict pages to make way for new ones. Note that a page that is read is automatically pinned.
2. Go through the `errors.h` file to understand what kind of errors may be thrown. *Do not make changes to this file.* But you may need to make use of these errors in your own `try-catch` blocks.
3. Go through the `constants.h` file. The two main constants that are of interest here are `BUFFER_SIZE` which denotes the number of buffers available in memory and `PAGE_SIZE` which in turn determines the `PAGE_CONTENT_SIZE`. All these constants may be change to test your code.

[1] Recall that a B+-tree typically leaves a part of its nodes empty. We are trying to emulate a similar behaviour.

4 Binary Search

The binary search algorithm proceeds in the well-known way. However, recall that in database systems, we do not access *elements*, we access *pages*. Therefore, binary search will access the *middle page*, then either access a page to the left or a page to the right. The process repeats until the element is found or not found.

Once the page is accessed, the computation is now in-memory. *No restrictions are placed on in-memory computations.* Therefore, you are free to choose your own data structures or functions to perform the in-memory operations.

Note: It is *not acceptable* to simply read all the pages into memory and perform a binary search on it. You are required to perform binary search *on disk*. Our testing software will keep track of your access patterns, and a sequential access to all blocks is easily detected.

Algorithm 1: Binary Search on disk

Input : A file accessible through the File Manager API containing the sorted integers, *num*, the integer to search for

Output: (Page number, offset) containing the integer if present, else, (-1,-1)

```
mid = floor((lastPageNumber + firstPageNumber)/2)
Read page mid into the buffer
if num is present in mid then
    output (mid, offset at which num is present)
end
else
    determine if num is present in pages to the left of mid or the right of mid
    repeat the procedure by updating either the lastPageNumber or the firstPageNumber
end
```

5 Insertion

The insertion algorithm is provided a file of sorted integers and a new integer or set of integers to insert. The insertion should be a *sorted* insertion. That is, the file remains sorted after the insertion.

Algorithm 2 shows how the insertion should proceed. Your code will be tested with both single insertions as well as multiple insertions at a time. Therefore, you have opportunities to optimize your code for the latter[2].

Note: It is not acceptable to use linear search to find the relevant page. Further, no occupancy restrictions are enforced. That is, while the initial input file may have an occupancy factor < 1, the file resulting from multiple insertions will not need to adhere to any such constraints. *However*, note that the final file cannot

[2] Recall that "optimize" in this scenario means reducing the number of disk reads and writes

have a page with occupancy *less than* the initial file.

Algorithm 2: Insertion

Input : A file accessible through the File Manager API containing sorted integers into which integers are to be inserted, a file accessible through the API that contains the integers to be inserted
Output: None

```
// Note : below steps are only for inserting single integer into file
```
Use a modification of Algorithm 1 to locate the page p where *num* should be inserted
repeat
  if *there is space in the* p **then**
    `// Note: Ensure that the file correctly follows the structure explained in Section 2.1`
    insert *num* in the correct offset in page p (use in memory operations)
    break
  end
  else
    `// since` *num* `is to be inserted in page` p `only`
    `// we will shift the last integer integer in` p `to next page(if exists)`
    $last\_num$ = last integer in page p
    erase $last\_num$
    insert *num* in the correct offset in page p (use in memory operations)
    `// shifting` $last\_num$ `to page` $p+1$ `is same as inserting it in next page`
    if *page* $p+1$ *exists* **then**
      repeat loop with $p = p + 1$ and num=$last\_num$
    end
    else
      create page $p+1$ at end of file
      write $last\_num$ in page $p+1$ (add INT_MIN at end)
      break
    end
  end
until *num is inserted*;

6 Merge Sort

For merge sort, we will follow the algorithm provided in Section 12.4 of the textbook by Silberschatz et. al. The following is a reproduction of the algorithm with a few modifications. Algorithms 3 and 4 together show how the external merge sort algorithm should be implemented.

Algorithm 3: Creating sorted runs

Input : A file accessible through the File Manager API, containing the integers
Output: A set of run files R_i, each of which are sorted and accessible through the File Manager API

```
// Note: The output file can have any occupancy
// Assumptions: No. of buffers available = B
```
$i = 0$;
repeat
  `// Reserve 1 frame of the buffer for the output file`
  read $B - 1$ blocks of integers (or the remaining set of integers) into the buffer;
  sort the integers;
  write the sorted data to file R_i, one block at a time;
  $i = i + 1$
until *until no more blocks are available in the input file*;

Algorithm 4: Merging the sorted runs

```
Input  : The list of run files
Output: A file accessible through the File Manager API, containing the sorted integers
// Note:  The output file must have occupancy 1
if No. of buffers available = B, no. of run files N < B then
    read one block of each of the N files R_i into the buffer
    repeat
        // Reserve at least 1 frame of the buffer for the output file
        choose the first integer (in sort order) among all frames
        write the integer to the output buffer, delete it from the frame
        if the frame is empty and not end-of-file (R_i) then
            read the next block of R_i into the frame
        end
    until until all buffer blocks are empty;
end
else
    // You will have to work out the case N >= B yourself
end
```

7 Submission details

7.1 Submission format

You will need to submit a zip file containing your code named `entrynumber1_entrynumber2_entrynumber3.zip` (e.g. 2017MCS0001_2017MCS0002_2017MCS0003.zip).

1. The zip file shall create a folder with same name.
2. The folder shall only contain your code(and not the buffer manager files). We will add buffer manager code to it during evaluation.
3. The folder shall contain a `Makefile`. The Makefile will be used to compile your code.

The Makefile will be used to compile your code with the following fixed targets. The executable for each program should have the same name as the targets given below -

1. Binary Search - `make binary_search`
2. Insertion - `make insertion`
3. Merge sort - `make merge_sort`

Your Makefile should also contain "clean" target for cleaning up compiled binaries. Our format checker will check whether your submission follows the above format or not at submission time.

7.2 Program execution

1. Binary Search

   Your compiled submission file will be run as -

   ```
   ./binary_search file_name integer_to_search
   ```

   As specified in Algorithm 1, your program shall print (Page number,offset) to standard output if `integer_to_search` is present in `file_name` file, otherwise (-1,-1) . Note both page number and offset start from 1.

   E.g., if `integer_to_search` is present as 16th integer in 4th page of `file_name`, you have to print

```
4,16
```

If integer is not found, print

```
-1,-1
```

2. Insertion

   Your compiled submission file will be run as -

   ```
   ./insertion sorted_input_file input_integer_file
   ```

   For each integer in `input_integer_file`, your program shall insert it in correct place in `sorted_input_file`. Note you do not need to print anything to standard output for this program.

3. Merge sort

   Your compiled submission file will be run as -

   ```
   ./merge_sort input_unsorted_file output_sorted_file
   ```

   Your program shall sort the integers present in `input_unsorted_file` file and write them to `output_sorted_file`. Note you don't need to output anything to the standard output for this program.

A submission VPL activity will be available on Moodle (details will be updated on Piazza later). You will receive a feedback regarding whether the submission conforms to the above submission format. Note - A successful submission shall not mean it is correct - your submission will be evaluated after the submission deadline is over.