Assignment 3
Deadline: 22 Apr 2021

1 Introduction

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

1. **Linear search**: Given an integer, find the position of all occurrences of the integer in the file using linear search.
2. **Binary search**: Given an integer, find the position of all occurrences of the integer in the file using binary search.
3. **Deletion**: Given an integer, delete **all occurrences** of the integer from the file while keeping the file compact.
4. **Join 1**: Given two unsorted files of integers, perform Join 1 and create a new file that contains the join of the integers.
5. **Join 2**: Given two files, first unsorted and the second sorted, of integers, perform Join 2 and create a new file that contains the join of the integers.

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 18.04 LTS
3. Your code should correctly compile and run on: Ubuntu 18.04 LTS
4. Only the standard C++ libraries (including STL) are allowed.

2 File Manager

The buffer manager implementation is available at `https://github.com/chauhankomal/COL362_632_Assignment3`. 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 contents of a page are defined by the following parameters:

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

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. A page can contain $n = floor$(`PAGE_CONTENT_SIZE / sizeof(int)`) integers, so all pages must have n integers except the last one.
3. Integers are always packed from the beginning of the page. Therefore, the empty space is contiguous and starts after the last integer in the last page until the end of the page. Empty space is indicated with INT_MIN.

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.

4 Linear Search [10 marks]

The linear search algorithm proceeds in the well-known way. However, recall that in database systems, we do not access *elements*, we access *pages*. Therefore, linear search will access pages from the first page till last page until the element is found. If the element is not found in any page then return (-1, -1).

Once the page is accessed, the computation is now in-memory. *You may use a small of amount of extra memory for any additional computation, such as defining and assigning variables.*

Restrictions: Only 2 buffers are available in memory. You may use one buffer to read blocks one at a time and one buffer to for writing your output to the file (one block at a time).

```
Algorithm 1: Linear Search on disk to find the first occurrence of num
Input  : A file accessible through the File Manager API containing the sorted integers, and num,
         the integer to search
Output: An output file accessible through the File Manager API in which result is to be stored.
         Store (Page number, offset) containing the integer if present, else (-1, -1) in the output file
$curr = firstPageNumber$
while $curr \leq lastPageNumber$ do
    Read page curr into the buffer
    if num is present in curr then
        output (curr, offset at which num is present)
        return
    end
    else
        increment curr by 1
    end
end
output(-1, -1)
```

You need to write a modified version of the above algorithm to output all the occurrences of a number in the output file. *You need to do this for all integers in the query file and the output sequence for each query integer will end with (-1, -1).*

Note: The order of the output sequence of occurrences doesn't matter. See sample test cases. The size of $PAGE\_CONTENT\_SIZE$ will be an even multiple of $sizeof(INT)$ for search test cases.

5 Binary Search [15 marks]

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. *You may use a small of amount of extra memory for any additional computation, such as defining and assigning variables.*

Restrictions: Only 2 buffers are available in memory. You may use one buffer to read blocks one at a time and one buffer to for writing your output to the file (one block at a time).

```
Algorithm 2: Binary Search on disk to find out any occurrence of num
Input  : A file accessible through the File Manager API containing the sorted integers, num, the
         integer to search for
Output: An output file accessible through the File Manager API in which result is to be stored.
         Store (Page number, offset) containing the integer if present, else (-1, -1) in the output file
while $lastPageNumber \geq firstPageNumber$ do
    $mid = floor((lastPageNumber + firstPageNumber)/2)$
    Read page mid into the buffer
    if num is present in mid then
        output(mid, offset at which num is present in mid)
        return
    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
end
output(-1, -1)
```

You need to write a modified version of the above algorithm to output all the occurrences of a number in the output file. *You need to do this for all integers in the query file and the output sequence*

for each query integer will end with (-1, -1).
Note: The order of the output sequence of occurrences doesn't matter. See sample test cases. The size of $PAGE\_CONTENT\_SIZE$ will be an even multiple of $sizeof(INT)$ for search test cases.

6 Deletion [25 marks]

The given deletion algorithm deletes the first occurrence of the integer num in the sorted input file and keeps the file compact.

Restrictions: Only 2 buffers are available in memory so you can load at most 2 pages of the file at a time. You need to modify the same file, don't create new output file.

Algorithm 3: Deletion of first occurrence of num

```
Input  : A file accessible through the File Manager API containing sorted integers from which
         integer is to be deleted, and the integer to be deleted
Output: None
1  Use a modification of Algorithm 1 or 2 to locate the page $p$ from where $num$ should be deleted
2  $offset\_p$ = offset at which $num$ is present in $p$
3  $(q, offset\_q)$ = next position of $(p, offset\_p)$
4  while $q <= lastPageNumber$ do
5      copy integer at $(q, offset\_q)$ to $(p, offset\_p)$
6      if $offset\_p ==$ last offset at page $p$ then
7          $p = p + 1$
8          $offset\_p = 0$
9      end
10     else
11         $offset\_p = offset\_p + 1$
12     end
13     if $offset\_q ==$ last offset at page $q$ then
14         $q = q + 1$
15         $offset\_q = 0$
16     end
17     else
18         $offset\_q = offset\_q + 1$
19     end
20 end
21 delete last page if empty
```

You need to write a modified version of the above algorithm to delete all occurrences of the number in the input file. *You need to do this for all integers in the query file such that the file stays compact.*
Note: You may need to think carefully about all the corner-cases. See sample test cases for input-output format.

7 Join 1 [25 marks]

In this join algorithm the relations $R1$ and $R2$ contain integers which may *not be sorted.*
Restrictions: Assume you have n memory blocks available then you will use 1 block for output, 1 for $R1$ and $n-2$ blocks for $R2$. Update the Algorithm 4 for the above assumptions. *You may use a small of amount of extra memory for any additional computation, such as defining and assigning variables.*

Algorithm 4: Join 1

Input : Two files accessible through the File Manager API containing integers from which integers are to be read.

Output: An output file accessible through the File Manager API in which result is to be stored.

```
for each page p in R1 do
    for each page q in R2 do
        for each integer num1 in p do
            for each integer num2 in q do
                if $num1 == num2$ then
                    add num1 in the output file
                end
            end
        end
    end
end
```

8 Join 2 [25 marks]

Here relation $R1$ contains integers which are not sorted and integers in relation $R2$ are sorted.

Assume you have n memory blocks available then you will use 1 block for output, 1 for $R2$ and $n-2$ blocks for $R1$. Update the Algorithm 5 for the above assumptions. *You may use a small of amount of extra* *memory for any additional computation, such as defining and assigning variables.*

Algorithm 5: Join 2

```
Input  : Two files accessible through the File Manager API containing integers from which integers
         are to be read.
Output: An output file accessible through the File Manager API in which result is to be stored.
for each page $p$ in $R1$ do
    for each integer $num$ in $p$ do
        $(q, offset)$ = Use Algorithm 1 or 2 to locate the page $q$ where $num$ is present   if $q == -1$
         then
            continue
        end
        else
            while $q \leq$ last page number of file and integer at $(q, offset) == num$ do
                add $num$ in the outfile file
                if $offset ==$ last offset of page $q$ then
                    $offset = 0$
                    $q = q + 1$
                end
                else
                    $offset = offset + 1$
                end
            end
        end
    end
end
```

9 Submission details

9.1 Submission format

You will need to submit a zip file containing your code named `entrynumber1_entrynumber2_entrynumber3.zip` (e.g. 2020MCS0001_2020MCS0002_2020MCS0003.zip). **Only one member of the team should submit.**

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 with the following fixed targets.

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. Linear Search - `make linearsearch`
2. Binary Search - `make binarysearch`
3. Deletion - `make deletion`
4. Join1 - `make join1`
5. Join2 - `make join2`

Your Makefile should also contain "clean" target for cleaning up compiled binaries.

9.2 Program execution

Note: Input file may contain duplicates. Optimal computation might fetch you some extra marks.

1. Linear Search

   Your compiled submission file will be run as -

   ```
   ./linearsearch <input_filename> <query_filename>.txt <output_filename>
   ```

   For each integer in `<query_filename>.txt`, your program shall search for it in the `<input_filename>` and output the page and offset of all the occurrences of the element followed by (-1, -1). The input file need **not** be sorted for linear search.
   Note you do not need to print anything to standard output for this program.

2. Binary Search

   Your compiled submission file will be run as -

   ```
   ./binarysearch <sorted_input_filename> <query_filename>.txt <output_filename>
   ```

   For each integer in `<query_filename>.txt`, your program shall search for it in the `<sorted_input_filename>` and output the page and offset of all the occurrences of the element followed by (-1, -1).
   Note you do not need to print anything to standard output for this program.

3. Deletion

   Your compiled submission file will be run as -

   ```
   ./deletion <sorted_input_filename> <query_filename>.txt
   ```

   For each integer in `<query_filename>.txt`, your program shall delete all entries of the integer in `<sorted_input_filename>`.
   Note you do not need to print anything to standard output for this program.

4. Join 1

   Your compiled submission file will be run as -

   ```
   ./join1 <unsorted_input_file1name> <unsorted_input_file2name> <output_filename>
   ```

   Your program shall join the integers present in the `<unsorted_input_file1name>` and `<unsorted_input_file2name>` and write them to the file named `<output_filename>`.
   Note you do not need to print anything to standard output for this program.

5. Join 2

   Your compiled submission file will be run as -

   ```
   ./join2 <unsorted_input_file1name> <sorted_input_file2name> <output_filename>
   ```

   Your program shall join the integers present in the `<unsorted_input_file1name>` and `<sorted_input_file2name>` and write them to the file named `<output_filename>`.
   Note you do not need to print anything to standard output for this program.