CS69001 Computing Laboratory – I

Assignment No: B1

Date: 05–August–2019

Foobarnautic Institute of Science and Technology (FIST) stores student records in files. Each student has a unique key (the roll number). This key is used to locate a student's record. The key is associated with satellite data, like the student's personal and academic profiles, the student's photo, the QR code of the student's fingerprints, and so on. In short, there is so much information stored against each student that it is infeasible to keep all the student records in main memory.

Each file stores the information of only a few students. For example, the records of all the MTech students of the Department of Bargodics in the 2019 batch are stored in one file. As FIST follows a similar roll-number format as IIT, the different files are naturally ordered. Take two files **`F1.dat`** and **`F2.dat`**. Then, the files do not contain a common roll number. Moreover, either all the roll numbers in **`F1.dat`** are smaller (with respect to string ordering) than all the roll numbers in **`F2.dat`** or all the roll numbers in **`F1.dat`** are larger than all the roll numbers in **`F2.dat`**.

We assume that each data file has a limit L on the number of records that it can store. When one wants to add a new record to a data file already storing L records, the file is split into two parts. One part consists of the records pertaining to the smaller $L/2$ roll numbers, and the other part the larger $L/2$ roll numbers. Then, the record for the new roll number is inserted in the appropriate part.

Although it is not very efficient, we assume that each file stores the records as a min-heap in the contiguous representation. The heap ordering is with respect to the primary key (roll number). So long as L is small, this strategy is fine. Moreover, this assignment is expected to give you an exposure to programming heaps. A heap is typically implemented in an array. Here the array resides in a file. Make a convention to ensure that each student record fits in exactly the same amount of space. So whenever a particular item in the heap is to be accessed, you can straightaway jump to that location using the call **`fseek()`** declared in **`stdio.h`**.

This assignment deals with two major parts. First, you need to manage min-heaps in files. Second, you need a binary search tree (BST) for indexing the files. The BST resides in the main memory. Given a roll number, the BST will guide you to a unique file which may contain the roll number. Moreover, if you want to insert the record of a new student (roll number), the BST will identify the correct unique file where this record is to be inserted. If that insertion attempt lets the file exceed its storage capacity of L records, the file is split into two parts as explained above. The BST may be kept height-balanced (this is not a part of this assignment). If the database stores n student records, then there will be $\Theta(n/L)$ files, so the height of the BST will be $\Theta(\log(n/L))$.

For each part, you need to write a FISTful of functions which are described now.

PART 1: MIN-HEAPS IN FILES

This assignment is meant for dealing with data structures (not building real-life database applications). So to start with, we make several simplifying assumptions. First, we assume that each key (roll number) is an integer in the range $[0, 10^7 - 1]$. Such an integer fits in a seven-character space. Roll numbers having less than seven digits may be padded with leading zeros or spaces. The second assumption is that there is no satellite data, that is, only a key value is stored in each record. If you add a separator (space or new line) between two consecutive records, you need eight characters per record (or key).

A database file starts with the size $s \in \{0, 1, 2, \ldots, L\}$ of the heap. Take $L = 32$ throughout this assignment, that is, s too fits in a seven-character space. Following s, there are s keys stored in a min-heap order. The file should have enough space to store L keys. The non-empty positions may be filled with spaces or some special markers. A sample data file is given below. It is a min-heap, and is not needed to be sorted.

```
     27
  83830  128029  131764  525055  218608  149260  639212  709327  768379  758038
 359634  612350  264834 1308751 1445835 1634258  786158 1430734 1368213  961768
1153024 1017586  534067 1178746 1386051 1199079  585768       _       _       _
      _       _
```

Traditionally, heaps are allowed to store duplicate values. This assignment prohibits the storage of duplicate keys. Heaps are indeed not a natural data structure to store dictionaries. You have to implement some unusual operations on heaps. Since all heaps are restricted in size to $L = 32$, these operations would be reasonably efficient. Write the following functions on heaps stored in files. Do not read any file to the main memory, apply the operations, and write back to the file. Instead make all the changes in the file itself. Use **`fseek`** to jump to the desired location of the file. Then do **`fscanf`** or **`fprintf`** as needed.

hinit Create a new database file to store an empty heap. Store 0 as the size of the heap. Create space in the file for storing all of the L keys that may be inserted in future. In an empty heap, these L positions would initially store a placeholder (like _).

hsearch Search for a given key in a database file. Heaps do not admit efficient searching. You need to make a linear search through all the elements of the heap. The running time is $O(L)$.

hinsert Insert a given key to a heap residing in a file. Use the standard heap-insertion procedure. Instead of reading an element $H[i]$ directly from an array, you need to **`fseek`** to the place in the file from where the i-th record begins, and then use standard **`fscanf`**. We do not allow duplicate keys in our database heaps. So a search for the key may be needed in the file. But this search should better be handled in the BST functions. Likewise, a check whether the heap capacity L is exceeded or not may be handled in the BST functions. That is, *hinsert* may pretend that it is given a key not present in the heap, and its insertion will leave us with a heap of size $\leqslant L$.

hfindmin Return the minimum key in a heap residing in a file. If the heap is empty, return ∞.

hdelmin Delete the minimum from a heap residing in a file. This involves copying the last element to the first position, and running heapify at the root. This function should fail if the heap is empty.

hfindmax Return the maximum of a heap residing in a file (or $-\infty$ if the heap is empty). The standard binary-heap data structure can be easily augmented to keep track of the maximum. Here, you may just make a linear scan through the file to locate the maximum. The running time is $O(L)$. You may restrict the search only to the leaf nodes, but the asymptotic running time will anyway stay as $O(L)$. This function is needed for diagnostic purposes only.

PART 2: BST FOR INDEXING HEAP FILES

A usual binary search tree (BST) stores a key in each node. The BST to be used in this assignment is a bit different. The keys are stored in files. The BST is needed to index the files. That is, no node in our BST would store a key or a list of keys. Each node, on the contrary, would store the two endpoints of the interval of key values stored in the files in the subtree rooted at the node. Each non-leaf (that is, internal) node has two child nodes, and is not linked to any database file. Each leaf stores the information (like the name or some identifier) of a database file.

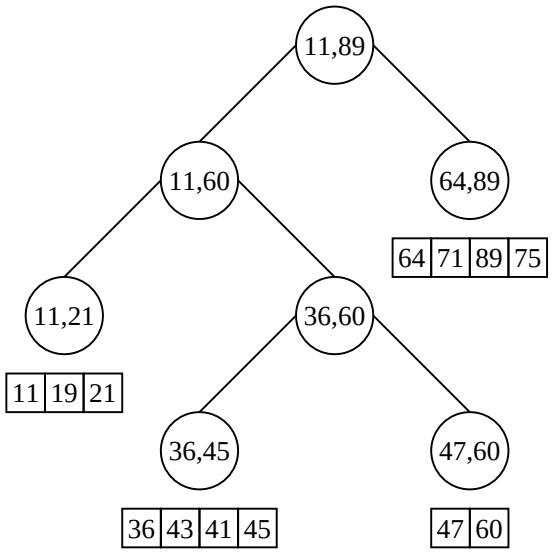
The figure on the next page provides an example. The figure assumes that the file capacity is $L = 4$. The minimum and maximum values stored in each node are shown inside the circle. The tree is a BST with respect to neither the minimum nor the maximum values. We will soon see that this information is enough to identify a leaf node uniquely by a procedure akin to search in standard BSTs.

The arrays in the figure show the contents of the files associated with the leaf nodes. Each file is organized as explained earlier. The heaps are shown as arrays (without the size) for conceptual clarity.

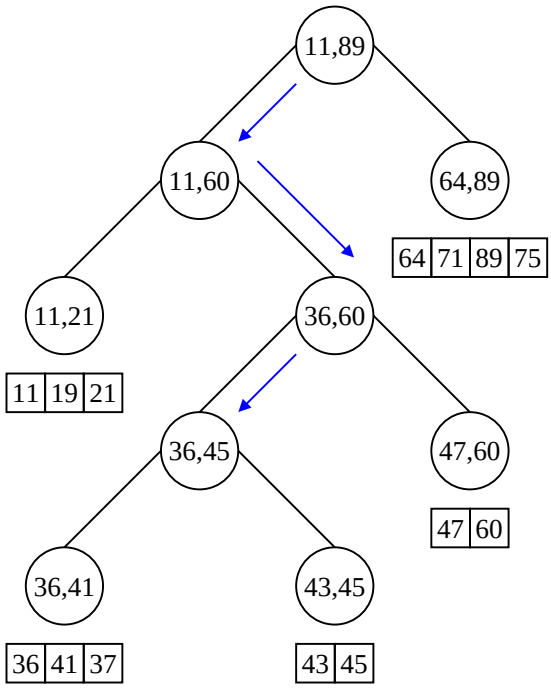
The root of the tree is an internal node having two children. The right child is a leaf node associated with a file storing the min-heap $64, 71, 89, 75$. The left child of the root is again an internal node. The minimum and maximum keys stored in the database are 11 and 75, so the root stores these two values.

The important database-related functions are explained now.

dbinit This creates an empty database of keys. First, the root node is created. This is now a leaf node, so a database file is opened and initialized to store an empty heap (use *hinit*). The function should return a pointer to the root node of the tree. In all future references to the database, this pointer is used.



(a) A database



(b) Effect of inserting 37

dbsearch Search for a key x in the database. You need to reach a unique leaf node, and find out whether x resides in the heap associated with that leaf node. Start the search from the root node. Repeat until you reach a leaf node. Suppose that p is the current node on the search path. If x is smaller than the minimum value stored at p, or larger than the maximum value stored at p, the search fails immediately. So suppose that x lies between these two values (both inclusive). We need to make a decision about the child node where the search will proceed.

Let L_{min} and L_{max} denote the minimum and maximum values stored in the left child of p, whereas R_{min} and R_{max} denote the same values for the right child of p. Recall that each internal node is required to have both the child nodes. If $L_{min} \leqslant x \leqslant L_{max}$, proceed to the left child of p. If $R_{min} \leqslant x \leqslant R_{max}$, proceed to the right child of p. If $L_{max} < x < R_{min}$, the search fails immediately.

If the search does not fail at an internal node, you eventually reach a leaf node. Call *hsearch* on the database file associated with that leaf node. Part (b) of the above figure shows the search path for 37 in the database of Part (a). You reach the leaf node marked $36,45$, and make a linear search for 37 in the file storing $36,43,41,45$.

dbinsert Insert a key x to the database. Like *dbsearch*, you make a search procedure to reach a leaf node. Unlike the search procedure mentioned above, you should not abort at any internal node. Instead, go to the left subtree if $x \leqslant L_{max}$, or to the right subtree if $x > L_{max}$.

Let v be the leaf node at which you end up. First, make a search for x in the database file associated with v (use *hsearch*). If x is found in the file, the insertion attempt stops (duplicate insertions are not allowed). Otherwise, you try to insert x in the heap stored in the database file associated with v. Call *hinsert* provided that the pre-insert size s of the heap is $< L$.

The most involved case is when you face the situation $s = L$. The new key will let the heap (the file) exceed its storage capacity. In this case, you need to split the heap in two parts. Initialize a new database file to empty using *hinit*. Then, by a sequence of *hfindmin-hdelmin-hinsert*, you transfer the $L/2$ smaller keys from the old database file to the new database file. If x is smaller than the largest key in the new file, insert x to the new database file, else insert x to the old database file. After this splitting, one of the two files contains $L/2$ keys, and the other $L/2+1$ keys.

The tree structure should also be updated to reflect this splitting of the old database file. The earlier leaf node v now is made an internal node by creating two new leaf nodes as children of v. The left child

is associated with the new database file, whereas the right child is associated with the old database file. The earlier association of v with the old database file is erased.

This case of splitting is exemplified in Part (b) of the figure on the last page. In the figure, $L = 4$. So the insertion of 37 forces the splitting of the database file already storing $36, 43, 41, 45$.

Throughout this process, you need to check for updating the minimum and maximum values at every node in the insertion path. Nodes outside this path are not affected.

This essentially covers all the database-related functions. However, for diagnostic purposes, you need to write a few more functions listed now.

- A function to print the minimum and maximum values stored in the leaf nodes. An inorder traversal of the BST will print a sorted list.
- A function to print the same listing as above. The difference is that instead of printing the minimum and the maximum values stored at a leaf node, you find (and print) these values by calling *hfindmin* and *hfindmax* on the database file stored at that leaf.
- A function to print the minimum values stored at all the nodes of the BST in the sorted order (use preorder traversal).
- A function to print the maximum values stored at all the nodes of the BST in the sorted order (use postorder traversal).
- One or more functions to print the statistics of the BST (its number of nodes, its number of leaves, and its height).
- A function to print the tree in a format specified in the sample output.

The *main*() function

- Create an empty database.
- The user first supplies the number n_{ins} of insertions to be made to the database. This is followed by n_{ins} integer key values in the range $[0, 10^7 - 1]$. If the user supplies duplicate keys, the final size n of the database will be smaller than n_{ins}. In any case, you do not need to keep track of n.
- Call the diagnostic functions described at the end of Part 2 to verify the correctness of your constructions.
- The user then supplies two keys, one present in the database, and the other not. Search for these keys, and print the search results.

Submit a single C/C++ source file. Do not use global/static variables.

Sample output

```
nins = 200
Insert keys:
 5113962 5464394 1275937 4842962 8823776 4825036 3313992  964499 7675308 3064941
  537555 2870180 9220258 7523937 1951044 8295709  347981 6592215 8032007 5067207
 5960519 4782654 4980953 2484026 3503275  395607 7934719 2573456 6887124 7812731
 2429182 2001086 5793477 3705119 9360401 4617253 1046508 5190745 8098104 1238168
 8255686 8635660 4108348 9992296 6159597 8575744  804357 9023931 5167960 8836364
 6607490 3644831 6135371 4104796 8645209 2154998 4500403 6579929 7244806 1387527
 6909012 9673988 5904965 2702490 5895460 5265366 7319743 9458320 2972463 7934200
  696488 3744501 6569860 4804836 3736797 5245809 5896933 7057506 4269740 1064893
 8410223 3393583 4709724 7061946 7498379 3354933 9216944 1998782 2451214 6461750
 5902661 9360227 6135738 1807626 2062717 4547550 7072993 1898812 4005870   45456
 9833012 4702358 3789958 8919224 2023547   43107 4165034 7920480 9616966  951126
 1501725 8027189 4344709 8727801 5089135 4359440 4599086 4306079 6358222 7050301
 3284181 2260883 8926880 1936271 4068510  989597 6483822 3657855 5404761 3006044
 6219663 7754126  224755 2525973 6673350 4764654 2569081 3354736 2685134 4702399
 4305863 6703211 2729588 1166924 7947364  335075 5526365 2546450 7157506 4400939
 2113103  441687 6661823 1039983 4894310 3246685 4545932 3894484 9420892 2467046
 6900529 5640555  221172 7125284 8166529 6894522 1889938 3251962  249259 7091424
 7954361 7071474 6310987 3200301  754750 4258351 6051728 6281115 9321153 3209234
 3198407 1434257 6167273 9860230 4990592 1061583 5623267 2052877 7472420 5044159
 4519923 4372949 3201066 4741095 4014585 3883947 4151969 8420875 7135909 6917580

+++ Inorder listing of min and max values of leaves
  43107 1387527 1434257 2573456 2685134 4068510 4104796 4825036 4842962 6135371
6135738 6703211 6887124 7523937 7675308 8645209 8727801 9992296

+++ Inorder listing of min and max values read from files
  43107 1387527 1434257 2573456 2685134 4068510 4104796 4825036 4842962 6135371
6135738 6703211 6887124 7523937 7675308 8645209 8727801 9992296

+++ Sorted listing of min values at all nodes
  43107   43107   43107   43107 1434257 2685134 2685134 4104796 4842962 4842962
4842962 6135738 6135738 6887124 7675308 7675308 8727801

+++ Sorted listing of max values at all nodes
1387527 2573456 2573456 4068510 4825036 4825036 4825036 6135371 6703211 7523937
7523937 7523937 8645209 9992296 9992296 9992296 9992296

+++ Statistics of the BST
    Number of nodes  = 17
    Number of leaves = 9
    Height           = 4

+++ The BST
    Range = [43107,9992296], File: None
    +---Range = [43107,4825036], File: None
        +---Range = [43107,2573456], File: None
            +---Range = [43107,1387527], File: B1/000006.dat
            +---Range = [1434257,2573456], File: B1/000003.dat
        +---Range = [2685134,4825036], File: None
            +---Range = [2685134,4068510], File: B1/000005.dat
            +---Range = [4104796,4825036], File: B1/000001.dat
    +---Range = [4842962,9992296], File: None
        +---Range = [4842962,7523937], File: None
            +---Range = [4842962,6135371], File: B1/000004.dat
            +---Range = [6135738,7523937], File: None
                +---Range = [6135738,6703211], File: B1/000008.dat
                +---Range = [6887124,7523937], File: B1/000002.dat
        +---Range = [7675308,9992296], File: None
            +---Range = [7675308,8645209], File: B1/000007.dat
            +---Range = [8727801,9992296], File: B1/000000.dat

+++ Search results
    search( 754750): PRESENT
    search(9878012):  ABSENT
```