Introduction:

This is Computing Assignment 2 project from E4 group. Based on Computing Assignment 1, we add B-tree, B+ tree, index structure and some basic operations. There are also lots of improvements from Computing Assignment 1.

Besides the original files, we add *Bptree.cpp, btree.cpp, maindata.cpp, index_structure.cpp* files containing all important class functions. We also have *IndexStrucuture_test.cpp* file to show you our implementation of retrieve, insertion, deletion, query and other operations. Input dataset file is stored in *Data* directory, which contains a small collection of data, a collection file with 1000 tuples and a collection file with 20000 tuples. Reports of 1000 tuples and 20000 are already stored in directories *Report-1000* and *Report-20000* respectively. Your newly generated report are under the directory *Report*. You can try different sets of data as we also provide files to produce new datasets. But according to our assumption, the upper number is 28 * 3 * 24 * 10 = 20160. Don't try large number easily, as it may consume much longer time.

Exercise 1 Implementation: Basic data structure for the storage of relation

1.**Design of four relation classes**: we set up four relation classes person, Medical_status, Registration, and appointment(which is the same as the TREATMENT type required in the assignment instruction file). And for each relation class, we recognize a pri_key, and a secon_key.(see table1) As for the person relation, we add the three treatment type, and use different priority rule to compute the key value for heap.Besides the keys, each relation class also has one dataptr that stores its current position in the block and one dataptr_old which is the dataptr of the tuple in b tree, and is used to update the b tree's dataptr information.

	Pri_key	Secondary_key
person	id	Status (definition see CA1 Read Me file)
Medical_status	id	Medical risk
Registration	Registration hour	Registration day
Appointment(treatment)	id	Treatment day

Table1: the pri_key and secondary_key for the four relation classes.

2. **The maindata relation storage:** This is realized using the doubly linked block lists, and block class introduction will be provided in the Ex3 part.

Exercise 2 Implementation: Basic operations on data structure

1.Modify the main function in Ca1 to integrate the index structure use:

Report list: use an index structure to store all the *person* class inside, we use *insert_I()* to insert the new person into the report list, use *bp_retrieve_I()* to use the person's id to retrieve person and upate the person's status as well as other information.

Appointment list: There is an appointment list which contains an index structure with the appointment class, for each hospital. we use *insert_I()* to insert the new appointment into the report list, use *bp_retrieve_I()* to read the appointment information, and use *bp_delete_I()* to delete the person's appointment from the appointment list when he withdraws.

Registration_I: This is an index structure that stores all the Registration relation information, when the person register, we use *insert_I()* to store the registration information inside for further report use. Note that registration time may be treated as primary key so we add the restriction to avoid possible repetition of registration. But for ID we haven't put restrictions since the possibility of random number producing repeated ID is very small

Medical_status_I: This is an index structure that stores all the Medical_status relation information, we use *insert_I()* to store the medical status information inside for further report use.

2. Use index structure to output various report files:

For weekly reports:

Firstly, we use function *b_retrieve_I()* in class *index_structure* to retrieve the people we need by patients' status and return a vector pointer. Then we use *merge()* to get a vector including all people's information.

Secondly, we need to input the information including their profession category, age category, risk status and the waiting time of the three types of people: the people who have been treated, the registered people with a set appointment and the queueing people without a set appointment. So, by checking the vector formed in the previous step, we produce three files per week, and each file includes one type people, i.e., "Weekly1File1" for people without appointment(status==1) in the 1st week; "Weekly3File2" for the registered people with a set appointment(status==2) in the 3rd week; "Weekly7File3" for treated people(status==3) in the 7th week. Because our test time is 3-months, there are totally 36 files are created.

Thirdly, to order these reports by name, profession category or age group, we create a sort function, by which a user is provided an input to choose the way he need to order the report. In this function, we compare each parameter in the report list and arrange them in ascending order, and then the other 36 sorted files are produced as well. The file name is in "SortedWeekly<m>File<n>" format, i.e., "SortedWeekly1File1" means this file is the sorted files for people without appointment(status==1) in the 1st week.

In addition, to make the files well-formatted, we choose to generate ".csv" files which can be open in Excel format.

For monthly reports:

We use function *b_retrieve_I()* in class *index_structure* to retrieve the peoople we need by patients' status and return a vector pointer. Then we use *merge()* to get a vector including all people's information. We write a function to put in the following information which is got from the vector into monthly reports: how many people have registered, how many of them are waiting, how many are waiting in total, how many treatment appointments have been made, the average waiting time, and the number of people who withdrew their registration. Because the test time is 3-months, there will be three monthly reports generated, and the file names are “monthly1_report.txt”, “monthly2_report.txt”, “monthly3_report.txt”.

For report by choice:

This class is used to provide functions which can generate reports according to the users’ will. Users can choose to generate the reports according to medical_status, register_day, register_hour_range, and treatment_time. We use different types of class pointer and print the information the class includes. In the process of using this system, firstly, you are asked to choose what criteria you want to filter and generate reports. Then, you will jump to the corresponding function and you are asked to input the parameter. Every time you want to exit, you can input “-1” and then you will return to the previous interface. If you choose medical_status, you are asked to input one of 0-3, 0 means no risk, 1 means low risk, 2 means medium risk and 3 means high risk. If you choose register day, you are asked to input a value in 0-83, which means the day a patient registered. If you choose register hour range, you are asked to input two values, which means the range of hour during which a patient registered, and in this function, we used function *query* to get the information we need. And if you choose treatment time, you are asked to input a value in 0-83, and then you are asked to choose the hospital which you want to check. Then you will get the reports you want.

Exercise 3 Implementation: Index structure on relations

Block class is used for storage of maindata tuples, and it is a doubly linked node. In each block, there’s a l_sibling ptr, and a r_sibling pointer. Each block also contains a left separate key and a right separate key, which are the separate primary keys with respect to the right and left sibling. The data storage is realized in the main block with size 15 in our use, and the overflow block with size 4 in our example. It is a template class, with T as the relation type, G as the pri_key type, and H as the secon_key type.

The dataptr for the tuple in the block is defined as follows： it is a pair<block<T,G,H>*, int>, the first term block<T,G,H>* is the block ptr, int is the index that marks the location of the block, the index for block is the index for mainblock if it is smaller than mainblock size, if its bigger, (index-mianblocksize) is index in the overflow block

Sort(): This function first put all the elements in overflow block into mainblock, then use insertion sort to sort according to the primary key of the tuples. As the dataptr has changed, we update them in the b tree.

Split(): When the elements in block is higher than the highfill threshold, this function is called to create a new right sibling for the block, and put half of the elements in old block into the new block. As the dataptr has changed, we update them in the b tree. And we insert the new separate keys in b+ tree.

Merge(): When the elements in block is smaller than the lowfill threshold, this function is called. If there exists a right or left sibling block, such that the total number of this block and the sibling block is higher than the lowfill factor, we simplly redistribute the elements, if not the case, we merge the block with one of its sibling and delete the block which has bigger elements. As the dataptr has changed, we update them in the b tree. And we update the new separate key in b+tree for case 1, delete the old separate keys in b+ tree for case two.

bp_retrieve(G pri_key): Given the pri_key, binary search in mainblock, and linear search in the overflow block, and return the right tuple

b_retrieve(pair<block<T,G,H>,int> dataptr)*: Given the dataptr, use the index information in dataptr to find and return the tuple.

Insert(G pri_key, T tuple)*: First insert in the overflow block, if the overflow block is full, call sort(), if the total number of tuples exceeds the high threshold, call split() to split.

bp_delete(G pri_key): Given the pri_key, first use bp_retrieve() to find the tuple's position, if it is in the mainblock, mark it with a tombstone ptr, else if in the overflow block directly delete it. If the number of tuples goes below the lowfill threshold, call merge().

b_delete(pair<block<T,G,H>,int> dataptr)*: Given the dataptr, delete the corresponding tuple. If the number of tuples goes below the lowfill threshold, call merge().

B+ Tree is the mostly same as described in lecture slides. Keys, blocks and children of a node are stored using arrays for convenient use. Besides, all siblings (besides leaf nodes) are connected with each other with a doubly linked list. B+ Tree has an order of 5 by default and supports implementation of retrieve, insert and delete. Note that B+ tree changes only when new blocks are created so instead of inserting or deleting a tuple directly, these operations operate on blocks. And they are mainly for block class to use. It has the following four functions:

Retrieve(G pri_key): given a primary key, it returns the pointer to the block where this primary key is in. It's done recursively using auxiliary function *_retrieve(Bpnode<T,G,H>* node, pri_key).*

Insert(G pri_key, block<T,G,H> blockptr)*: given a primary key, it finds the place where *blockptr* should be inserted. Since B+ Tree contains one more block than primary keys. *Init_insert(block<T,G,H>* blockptr)* should be implemented before using other inserting operations. Insert is implemented recursively using an auxiliary function *_inert(Bpnode<T,G,H>* node, G pri_key, block<T,G,H>* blockptr).* When a node needs to be split, *split(Bpnode<T,G,H>* node)* is used to split the node. And splitting continues if parent node needs to be split as well.

Delete1(G oldpri_key, G newpri_key): This function corresponds to the case where two blocks are merged and splitted again. For B+ Tree, this means a new separate key is required. So it simply updates *oldpri_key* with *newpri_key.*

Delete2(G pri_key, block<T,G,H> blockptr)*: This function corresponds to the case where two blocks are merged and not splitted again. For B+ Tree, this means a primary key and a block are deleted, which may lead to reorganization of B+ tree. So if reorganization is required. *Merge(Bpnode<T,G,H>* node)* is used to merge two nodes. *Merge1 and merge2* are called by *merge* corresponding to different cases of merging. Merge proceeds if parent node needs to be merged as well. Similarly, both *delete1* and *delete2* are implemented recursively using auxiliary functions.

Besides, B+ Tree also contains different binary search functions and linear search functions. They are used to find the index of key/block/child in a node to proceed searching recursively. They have different arguments and correspond to different cases. One has to admit these functions may be a little confusing. There is another function *block_retrieve(block<T,G,H>* blockptr),* which is just for block merge function to find which node this block is in.

B+ Tree printing format: A node is presented with (key1, key2, key3...) where key1, key2, key3 are primary keys stored in this node. For nonleaf nodes, children are followed behind them. For leaf nodes, blocks stored in the node are followed behind them. Blocks are represented by [Left separate key, Right separate key] of the block. Nodes of different levels have different indentations for a better view of the tree structure.

B-tree is almost same as introduced in the lecture. However, this B-tree is used to store secondary keys, which means that all objects with same secondary key are stored in one same place. So we use spider class to store all objects with same secondary key. In spider class, it has two member, one is secondary key value, another is vector to store objects. So in B-tree, every node has two list, one list is spider list and another is child list. Another property is all same as normal b-tree. Below I will introduce some functions will be called in the index structure.

bp_insert_B(pair<block<T,G,H>, int> dataptr, H k)* :this function is to insert object into b-tree. We first use secondary key k to find proper spider, and then insert this dataptr into spider. If not find the corresponding spider, just create a new spider and insert into btree node.

b_retrieve_B(H k): this function is to use property of btree to quickly find the corresponding spider and return this spider.

b_delete_B(H k): this function first call retrieve to find the spider with secondary key k and then delete and return one object in the spider. If we want to delete all the objects with k, we will call this function many times until the vector in this spider is empty.

bp_delete_B(pair<block<T,G,H>, int> dataptr,H k):* This function is to delete one particular dataptr in one spider of b-tree. We first call retrieve function to find spider with key k and then find this dataptr in the spider and delete it.

b_update(H k, pair<block<T,G,H>, int> oldptr, pair<block<T,G,H>*, int> newptr):* this function is to update objects in the spider. We first call retrieve to find spider with k, and find old object in the vector of spider and update it to new one.

Index structure contains a first block (blocks are linked together with doubly linked lists), a B+ tree for primary key and a B tree for secondary key. It's the main structure than contains all of the information. Functions of Index structure simply call functions from block, B+ tree and B tree. Also, both B+ tree and B tree contain all information stored in blocks. When you operate on B+ tree, B tree is updated as well and vice versa. It supports the following operations: *bp_retrieve_I(G pri_key):* given a primary key, a pointer to where this tuple is stored is returned. Bp means it retrieves using B+ tree, I means this function belongs to index structure.

B_retrieve_I(H secon_key): given a secondary key, all tuples with this secondary key are returned. B means it retrieves using B tree. I means this function belongs to index structure.

Insert_I(T a_tuple)*: insert the tuple to the block. Update both B+ tree and B tree accordingly.

Bp_delete_I(G pri_key): given a primary key, tuple with this primary is deleted. Bp means it deletes using B+ tree. Note that B tree is synchronized as well.

B_delete_I(H second_key): given a secondary key, all tuples with this secondary key are returned. B means it deletes using B tree. These tuples are deleted from blocks. Note that B+ tree is synchronized as well.

Query: Given one primary key, it returns a vector containing the tuple with this primary. Given two primary keys, it returns a vector containing all tuples with primary keys in range between. To print the result, use *printquery()* with returned vector as argument. **Attention about *query* with a range**: since tuples in overflow block are randomly ordered, it's a little complicated to select tuples in overflow block in order. For simplicity, we sort the mainblock and overflow block (Put all tuples in overflow block into main block) before we try to search target tuples. So tuples's locations in blocks may be different from their original locations.

You can use *prettyprint()* to display all information, or use *printblock(), printBptree(), printbtree(),* to print different structures separately.

Index structure Test

This file "IndexStructure_test.cpp" contains an interactive presentation of all of the index structure implementations. You can compile and run it. Please pay attention to the instructions on the terminal. In this test, we mainly have three parts :insert, retrieve and delete. For the insert part, we first randomly creates 100 tuples with (primary key, secondary key), and store primary keys in vector v and secondary keys in vector v2. The primary keys are integers ranged from [0,2000] and secondary keys are integers ranged in[0,99]. We insert these 100 relation tuples into the index structure and display the structure for you in the terminal. Then for retrieve part, we will show two operations. The first is retrieve one relation tuple according to one primary key you choose. So terminal will ask you to enter one primary key, an integer ranged in [0,2000], and terminal will show whether it can retrieve this relation or not. The second retrieve operation is query, which means you enter two integer, terminal will show you all relation tuples with primary key between two integer. Please ensure that the first number you entered is smaller than the second one, and they are in range (0,2000). The final test part is delete operations. The first delete is delete relations according to primary key. To show delete operation intuitively, we will ask you enter two integers a and b, then we will delete tuples with keys from v[a] to v[b]. So you should enter two integers and first number you entered is smaller than the second one, and they are in range [0,99]. So terminal will show you current index structure. Another deletion operation is delete according to all the tuples with given secondary key. You will be asked to enter an interger ranged in [0,99] and not be those you just have deleted. Then terminal will show you current index structure. That's all for this test.cpp.

Instruction:

1.First type "make" in the terminal to compile our program.

2. **IMPORTANT:**

Enter 1 to test a small set of data (only two day's register information, no weekly and monthly report).

Enter any other number to test a big set of data of three months.

3. Enter 1 or 2 or 3 to generate particular type of sorted weekly report. There are 12 weekly reports in total, so you need to enter these numbers for 12 times. Please be patient (Especially if you try large datasets)😊

4. Enter integers according to the instruction appears in the terminal to generate "report by choice".

5.Type "make clear" in the terminal to clear all the output files and execution files.