Project: Stage 1
The Index Structures

Due: Monday, October 25, midnight.

Outline

The first stage of the project involves implementing the structures described in the "XML Indexing Schema" handout using NEUStore for pagination and buffer management.

On this stage you are implementing individual structures from the indexing schema, but not the overall insertion/deletion/retrieval operations. Your goal is to ensure that each index structure defined in the "XML Indexing Schema" handout is properly stored on disk, and implement atomic operations inserting and deleting index records/entrees, and updating the information stored in individual records.

During Stage 2 of the project, you will be using the Stage 1 functionality to develop methods for inserting information about XML elements/tree nodes into the index, and retrieving them.

Below is the list of classes and methods for each class that you need to develop. The methods listed present **officially mandated API** which we will be using for testing your code. In addition to the classes/methods defined above, you are welcome to build any classes/methods that you deem necessary/desirable for successful completion of this (and, also, further) stage(s) of the project.

API

Addresses

In order to be able to test various implementations, you are asked to implement a container class for the disk addresses, `DiskAddress`. Please refer to the "XML Indexing Schema" handout for information about disk addresses. The following method should be provided and should return a nicely formatted string

- `toString()`: output string representation of the adddress.

In addition, `DiskAddress` class shall have two **static** methods:

- `public static DiskAddress createAddressByRN(int PageId, int RecordNumber)`.
- `public static DiskAddress createAddressByOffset DiskAddress(int PageId, int Offset)`.

These methods will be used by test programs to generate `DiskAddress` instances.

ElementIndex

The **ElementIndex** structure is to be represented by the `ElementIndex` class. The following methods must be implemented for `ElementIndex`:

- `ElementIndex(DBBuffer buffer, String filename, int isCreate)`: class constructor, specifies the file name under which the index is to be stored on disk, also provides the buffer instance for the class. Finally, `isCreate`=1 means the index file is created from scratch, `isCreate`=0 means, an existing index file needs to be opened.
- `int insertElementName(String Name)`: insert information about a new element name into the index. Returns `ElementId`, the unique Id assigned to the new element.
- `int getElementId(String Name)`: return the `ElementId` of the given element.
- `String getElementName(int Id)`: return the name of the XML element with a given id.
- `DiskAddress getAddress(String Name)`: Return the address from which the records for this element will start (this can be a full address for a dense index, or just a pageId for the case when buckets are used for each element).
- `int insertElementEntry(String name, DiskAddress address)`: index element entry with given `name` and `address`. Returns exit status.
- `int deleteElementEntry(String name, DiskAddress address)`: delete specified entry from the index. Returns exit status.
- `int insertElementEntryById(int Id, DiskAddress address)`: index element entry given its ElementId and `address`. Returns exit status.
- `int deleteElementEntryById(int Id, DiskAddress address)`: delete specified entry given its ElementId and `address`. Returns exit status.

- `int getNumberEntries(int Id)`: given an element id, return the number of entries for this element in the index.
- `DiskAddress getElementEntry(int Id, int i)`: retrieve the address of the ith entry for the XML element with given element id.

StructureIndex

The **StructureIndex** structure is to be represented by the `StructureIndex` class. The record of the **StructureIndex** is to be represented by the `StructureIndexRecord` class. The following methods must be implemented in the `StructureIndex` class.

- `StructureIndex(DBBuffer buffer, String filename, int isCreate)`: class constructor, specifies the file name under which the index is to be stored on disk, also provides the buffer instance for the class. Finally, `isCreate`=1 means the index file is created from scratch, `isCreate`=0 means an existing index file needs to be opened.
- `DiskAddress insertEntry(StructureIndexRecord record)`: insert a new entry into the structure given the information specified. You can assume insertion always happens to the end of current file. Returns the address of the new record.
- `DiskAddress getAddress(int nodeId);` return address of the record with a given `nodeId`
- `int deleteEntry(int NodeId)`: delete specified entry from the file. Since this file is sequential, you must slide other records on the page (but sliding across pages is not necessary). Returns exit status
- `int updatePostOrder(int nodeId, int EndPosition)`: update PostOrder value for a given node. Returns exit status.
- `int setContent(int nodeId, DiskAddress contentAddress)`: update the pointer to the content of the XML node. Return exit status.
- `StructureIndexRecord getRecordById(int nodeId)`: given a node id, return its record.
- `StructureIndexRecord getRecordByAddress(DiskAddress address)`: given a disk address, retrieve the record.

For the `StructureIndexRecord` class, you must have a following constructor:

```
StructureIndexRecord(int nodeId, int elementId,
int PreOrder, int PostOrder, int Ordinal, int Layer,
int Parent, int isLeaf, DiskAddress ptrContent);
```

Here, all parameters are as in the description of the **StructureIndex** record, with the `ptrContent` replacing the last two parameters from that list.

Content

Content data must be represented in your implementation by `XMLContent` class. The following methods are to be implemented.

- `XMLContent(DBBuffer buffer, String filename, int isCreate)`: class constructor, specifies the file name under which the index is to be stored on disk, also provides the buffer instance for the class. Finally, `isCreate`=1 means the index file is created from scratch, `isCreate`=0 means an existing index file needs to be opened.
- `DiskAddress insertContent(String record)`: insert a new content record. Returns the address of the record. Note, that here, depending on your implementation, the address may wind up being of the form `(PageId, Offset)`
- `String getContent(DiskAddress address)`: return the content string stored at the given address.
- `int deleteContent(DiskAddress address)`: delete specified content record. Return exit status.
- `int getSize(DiskAddress address)`: return the length of the content record stored at the given address.

AttributeIndex

AttributeIndex must be represented in your implementation by `AttributeIndex` class. The following methods are to be implemented:

- `AttributeName(DBBuffer buffer, String filename, int isCreate)`: class constructor, specifies the file name under which the index is to be stored on disk, also provides the buffer instance for the class. Finally, `isCreate`=1 means the index file is created from scratch, `isCreate`=0 means an existing index file needs to be opened.
- `int insertEntry(int nodeId, String attName, String attValue)`: insert a new entry into the table. Returns exit status.
- `String getValue(int nodeId, String attName)`: return the value of the specified attribute of a given node.
- `int deleteEntry(int nodeId, String attName)`: delete specified entry. Return exit status.

Notes

The assignment is based on the use of NEUStore package as the underlying backend. In particular, all four main classes described above, `ElementIndex, StructureIndex,` XMLContent and AttributeIndex should extend the base DBIndex abstract class and, possibly be a modification of the HeapFile class implementation.

It is up to you to decide how many files you want to use to store the entire index. There are four parts to the indexing scheme, so, one can use one file per index structure. It is also possible to combine all indexes into a single file, and to choose any midpoint between these two possibilities. The key is in correctly defining class constructors for each case.

As mentioned above, the list of classes/methods given here merely reflects the API your implementation must provide. You will need to define other classes (e.g., for each disk page type, you need to extend DBPage with an appropriate structure; additionally, you may need to define a separate class for each record type, each block header type and each file header page type). You may also need to define/implement methods for mandated classes that are not specified here.

Grading

A grading script will be Java main program, which will perform the following activities:

1. Create all index structures.
2. Insert a few items in each index structure.
3. Retrieve inserted items.
4. Delete items, verify that deletion took place.
5. Test any additional API functionality.

One or two tests will consist of the list of operations that precisely index a specific (not very large) XML document. Some other tests will insert and delete records into the index structures independently, without requiring the combined stored data to correctly index any XML documents. Some tests of both types will be hidden.

We will describe expected results for each publically released test.

Each test item will recieve a weight/percentage towards total grade for the stage. The grade for the stage is the sum of all test items correctly executed. (a test item may be a portion of the test, not the full test).

Coding and Submission

Since NEUStore encourages use of Eclipse as a programming environment, so do I, although, it is not necessary. All grading will take place on CSL machines, so, please, make sure you test your code on one of the machines there.

While code is not graded, we reserve the right to examine it. Please comment adequately and make effort to make your code readable (your teammates will be the first to benefit from this). Each file must contain a comment header, specifying the team name, names of the team members and the purpose of the file.

If your submission requires any explanations (e.g., you have changed the indexing structure), put them in a `README` file, which you are to submit separately from the code archive. Put the publically available test files in the root of your submission. I should be able to compile your submission by running `javac test.java` command for each public test file. If your submission requires putting test files in a different location, and compiling with one or more flags set, please include a complete description of the running procedure in the README file.

You should have one submission per team. Zip and gzipped tar are the allowed submission formats. Name your archive `project-stage1-<team>.ext`, where `<team>` is the name of your team and `<ext>` is either `zip` or `tar.gz`.

Submit your work using `handin`:

```
handin dekhtyar project01 <archive> README
```