Overview
Phase 1: During Phase 1 the program reads records in XML from standard input. When reading each line the program splits the line by each XML tag to retrieve the appropriate values in each of the following sections: ad ID, category, location, title, description and a price. The program handles the following conditions: when retrieving records for terms the program Ignore all special characters coded as &#number; such as &#29987; which represents 産 as well as &apos;, &quot; and &amp; which respectively represent ', " and & and ignore terms of length 2 or less, and retrieves non empty dates & non empty price records. As the values are retrieved they are stored in a temporary list to be appropriately written to their newly created files: terms.txt, pdates.txt, prices.txt, ads.txt.

Phase 2: Using the Linux sort command, the system takes the output files from Phase 1 and sorts them, while eliminating duplicates. The system takes the sorted files and performs string and line operations to split the files into the required information and writes it to new files. The functions include the removal of extra whitespace and backslashes and then splits the lines at the colon into indexable objects. The new split files are entered as arguments into the Linux Berkely database function db_load to create B+ tree indexes for pdates, prices and terms (da.idx, pr.idx, te.idx respectively) and a hash index for ads (ad.idx). These index files are used for operations in Phase 3.

Phase 3: During phase 3, the program simply prompts the user to enter a query (or multiple queries). Upon receiving the query (or queries) from the user, the terminal displays all query results. The user can change the type of output by typing 'output=full' where all results display the title and description. Or the user can type 'output=brief' to have the results display the title and entire ad record (in XML format) to the terminal. The default setting is the brief display. If using multiple queries, they must be separated by at least one space (the number of spaces internal to the query itself can have 0 or more spaces between operators and terms i.e. 'price <=   100' is a valid query)

Query Algorithm
The program begins by taking the entire user input and parsing it using regular expressions to determine what type of queries the user wants to conduct. There are five main types of queries: location query, category query, term query, price query, and date query. For each type of query, there is a python list that contains strings of that query type. These lists are populated as the user input is parsed. For instance, take the following query,

Price <= 200 date > 1997/06/15

The main program uses regular expressions to determine that price <= 200 is a price query and date > 1997/06/15 is a date query and populates the corresponding price list and date list. The lists would look as follows

priceList = ['price<=200'], dateList = ['date<=1997/06/15']

For every type of query we also have a function for that query (priceQuery(), dateQuery(), etc.) and for every item in each list, we execute the corresponding query. Each query function returns a list of tuples containing the results of the query in form [(Ad Title, Full Ad)...]. Upon returning the list of results, we then add it to a ‘master’ results list that contains all the results of all the queries executed. Before adding the returned list to the master results list, however, we take the intersection of the returned results and all results currently in the master list. This guarantees that the remaining results in the master list have met the conditions of all queries.

Searching for query results in regards to terms, price, and dates is fairly simple using the Berkeley DB Cursor commands since they are key values within the indices we have created. For terms we find the first record that matches the term key and iterate down the index, returning all records that match the term key until we hit a key that does not match. Partial matching was implemented using range search. Since the records were sorted, we first found the key that was a closest match to the term that was entered, and iterated through the index as long as the key we were currently on started with the term that was entered by the user. We then grab the ad Ids of all the records and search for the ad using the ad index we created. Price and date queries operate similarly to the term index (we find the first key that obeys the query conditions and iterate forwards or backwards depending on the query conditions until we reach a key that does not match). Ranged search was used for prices and dates. After finding the corresponding ads from the ad index, we then return the list of tuples that matched the query conditions back to the main function.

Category and location queries were a little more difficult to handle. Neither category nor location are keys in any of the indexes. We still have a query function for both, but they are only executed in the worst case scenario. The worst case scenario is if a location or category query is executed by themselves. When this happens, we have to iterate through the entirety of either the price or dates index and find the records where the location or category is a match, and then finding the full records using the ad index like the other queries. This is incredibly slow, and to mitigate this, whenever possible we make sure to pair the location and category queries with either the date or price queries. When executing the price or date query, the program checks to see if there are also category, location queries. If there are, we search for the price or date query results like normal, except when we go to the ad index to find the full ads we only search for the prices (or dates) that also match the location (or category) conditions. By doing this we no longer have to execute the (worst case scenario) category or location queries as they are handled by the price and date queries. If and only if there was not price or date queries, do we execute the location or category queries.

Testing Strategy (Everyone)

We used the small files (of only 10 records) that were provided with the project spec on eClass while implementing our python program. Once everything was implemented, we conducted testing the larger files (1000 records) to determine if the integrity of our program still held.

Group Work Breakdown (Everyone)
Jeffrey Baglinit
CCID: baglinit
- Completed Phase 1 and date query and phase 3. Total Work Time: 12hrs

Warren Thomas
CCID: wlt
- Expanded upon the main program created by Aaron and developed the interactive main program that called upon the phases. The main program can be given a file argument in the command line when called upon in the format “>>python3 miniproject.py filename.txt” and the phases will be implemented on the given file. If a file is not supplied when the program is called, a filename is requested. The program gives feedback on the progress of the program, including the creation of all items and the optional removal of items upon exit. The program works in such a way that it progresses through phases 1 and 2, and then continually loops on phase 3 until explicitly exited. The main program allowed for modularity of the program, so that all phases could be improved upon independently and the system could easily be tested. Also worked on the category queries used by Phase 3 based on work and research completed by Aaron. Total work time: 12 hours.

Aaron Espiritu-
CCID: adespiri
For phase 3 completed 3 of the 5 main queries (price query, term query, location query). Set up the command line interface and formatting of printed results on the command line. For phase 2, studied up on how to sort files and build indexes using these sorted files. Drafted a rough program that would format the sorted files and build the indexes for us. This program was drastically improved by Warren. Total work time: 14 hours.