Homework 2: Skiplists and Dynamic List ADT

L5 - Group 3

Habib University
Spring 2023

Part I
Dynamic List ADT

In this assignment, you will implement a dynamic list ADT in Python that allows for inserting and deleting elements at a specified index and retrieving elements at a specified index. The list should be implemented as both a dynamic array and a linked list.

1 Implementation Details and Tasks

- Implement the `DynamicArrayList` and `LinkedList` classes in the accompanying file, `listadt.py`, in the accompanying `src/` sub-folder.
- In the `DynamicArrayList` and `LinkedList` classes, implement the following methods:
  - `insert(index, value)`: inserts value at the specified index
  - `delete(index)`: deletes the element at the specified index
  - `get(index)`: returns the element at the specified index
  - `size()`: returns the number of elements in the list
  - `display()`: returns the string representation of the list
- The dynamic array should be implemented using Python `arrays`. Upon creation, the dynamic array will have a size of one, and all empty items will be initialised as -1.
- The dynamic array should dynamically resize itself when necessary to ensure efficient use of memory i.e., **the array should double it's size when it is full and reduce by half once it is quarter the size of array.**
- The cost of resizing the array should be linear in the size of the array.
- The linked list should be implemented as a singly linked list.
- The program should take as an input a file that specifies which data structure should be used to implement the dynamic list ADT and which operations should be performed on the list.
- The program should read the input file, create the appropriate data structure, and perform the specified operations. The program should then output the result of each operation.
- The function `load` which parses the input file and saves the output of the program to a given file, has already been implented in the accompanying file, `listadt.py`, in the accompanying `src/` sub-folder.

1.1 Input and Output

The input file will specify which data structure should be used to implement the Dynamic List ADT and which operations should be performed on the list. The input file will have the following format:

```
<data structure>
<operation> <arg1> <arg2>
```

- ⟨**data structure**⟩ is either array or linkedlist, indicating which data structure should be used to implement the Dynamic List ADT
- ⟨ **operation**⟩ is one of the following:
  - **insert**: Insert an item into the list. The ⟨arg1⟩ is the index at which to insert the item and ⟨arg2⟩ is the item to be inserted.
  - **delete**: Delete an item from the list. The ⟨arg1⟩ is the index of the item to be deleted.
  - **get**: Return the item at the specified index in the list. The ⟨arg1⟩ is the index of the item to be returned.
  - **size**: Return the number of items in the list.
  - **display**: Return the string representation of the list.

The output of the program will be the result of each operation, one per line.

1.2 Example

In the following examples, the input file specifies which data structure should be used to implement the Dynamic List ADT. The program then performs the specified operations on the list and outputs the result of each operation.

1.2.1 Linked List

```
linkedlist
insert 0 10
insert 0 20
insert 1 30
get 0
get 1
get 2
delete 1
size
display
```

Code Listing 1: Sample input file for creating a LinkedList

```
20
30
10
2
[20, 10]
```

Code Listing 2: Sample output generated by the program

1.2.2 Dynamic Array List

```
array
insert 0 10
display
insert 0 15
display
insert 1 5
display
size
insert 2 30
display
delete 1
display
delete 1
display
size
delete 1
display
```

Code Listing 3: Sample input file for creating a DynamicArrayList

```
[10]
[15, 10]
[15, 5, 10, -1]
3
[15, 5, 30, 10]
[15, 30, 10, -1]
[15, 10, -1, -1]
2
[15, -1]
```

Code Listing 4: Sample output generated by the program

1.3 Testing

Once you have successfully implemented the `DynamicArrayList` and `LinkedList` classes, you can test your code by reading from the accompanying data file, `data/input.txt`, and performing operations on the list adt. For grading purposes, your submission will be tested automatically by GitHub using the accompanying `pytest` file, `test_listadt.py`.

Part II
Skiplists

The solutions to the problems in this part are to be entered inline below. Remove all the other parts and sections from this document. Enter your team name as the author in the document's title.

1. A Ranked Set [10 points]

[1] Design a version of a skiplist that implements the `SSet` interface, but also allows fast access to elements by *rank*. That is, it also supports the function `get(i)`, which returns the element whose rank is `i` in $O(\log n)$ expected time. (The rank of an element `x` in an `SSet` is the number of elements in the `SSet` that are less than `x`.) Describe how your version differs from a regular skiplist and provide pseudocode of `find(x)` and `get(i)` for this version.

Solution:

To allow for faster access to elements by rank we can alter the regular skiplist such that it keeps track of the size of the sublist rooted at each node. This will allow us to minimize the time complexity of `get()` operation, which returns the element at a given index in the sorted set.

In this altered version we would include two additional pieces of information for each node:

- **size:** the number of elements in each sublist starting at the node
- **rank:** the rank of the node in the SSet, computed using the size

In a regular skiplist, to find the element at a given index, one would need to traverse the list from the beginning until reaching the desired index. This would take $O(n)$ time, where n is the number of elements in the set. However, in this altered version of the skiplist, the size of each subtree is stored at each node, so we can use this information to efficiently locate the element at the given index.

To do this, we start at the sentinel node and traverse the skiplist, keeping track of the rank of each node (the total number of elements visited so far, including the subtree sizes). When we reach a node whose rank equals the desired index, we return its element. This takes $O\log(n)$ time on average, where n is the number of elements in the set.

While this version of skiplist is similar to a regular skiplist in that it uses a linked list with multiple levels of nodes to provide efficient search, insertion, and deletion operations, it differs in that it maintains subtree sizes and uses them to efficiently support the `get()` operation.

The pseudocode of `find(x)` for this version is given below:

```
def find_pred_node(x):
u = sentinel
r = h
while r >= 0:
  while u.next[r] is not None and u.next[r].x < x:
    u = u.next[r]  # go right in list r
  r -= 1  # go down into list r-1
return u

def find(x):
  u = find_pred_node(x)
  if u.next[0] is None:
    return None
  return u.next[0].x
```

[1] Adapted from Exercise 4.9 in the textbook.

The pseudocode of `get(i)` for this version is given below:

```
      def get(i):
        if i < 0 or i >= n:
          raise IndexError()
        u = sentinel
        r = h
        rank = 0
        while True:
          while u.next[r] is not None and rank + u.next[r].size <= i:
            rank += u.next[r].size
            u = u.next[r]
          if rank == i:
            return u.x
          r -= 1
```

In both the functions for the modified SSetSkipList, the average time complexity for both `find(x)` and `get(i)` is $O\log(n)$.

2. Finger Search [10 points]

[2] A *finger* in a skiplist is an array that stores the sequence of nodes on a search path at which the search path goes down. (The variable `stack` in the `add(x)` code on page 87 is a finger; the shaded nodes in Figure 4.3 show the contents of the finger.) One can think of a finger as pointing out the path to a node in the lowest list, L_0.

A *finger search* implements the find(x) operation using a finger, walking up the list using the finger until reaching a node `u` such that `u.x < x` and `u.next = nil` or `u.next.x > x` and then performing a normal search for `x` starting from `u`. It is possible to prove that the expected number of steps required for a finger search is $O(1 + \log r)$, where r is the number values in L_0 between `x` and the value pointed to by the finger.

Design, i.e. provide the necessary pseudo code for, a version of a skiplist that implements `find(x)` operations using an internal finger. This subclass stores a finger, which is then used so that every `find(x)` operation is implemented as a finger search. During each `find(x)` operation the finger is updated so that each `find(x)` operation uses, as a starting point, a finger that points to the result of the previous `find(x)` operation.

Solution:

To implement a skiplist that uses an internal finger for `find(x)` operations, we can store the finger as an attribute of the skiplist. Whenever a `find(x)` operation is performed, we first use the finger to perform a finger search and then update the finger to point to the node found by the search.

The `find(x)` method traverses the skip list from the highest level to the lowest level, moving the current node pointer down to the next node in the list until it finds a node with a value greater than or equal to the search value x. If it finds a node with a value equal to x, it returns `True` and resets the finger to the head. Otherwise, it updates the finger to point to the last node before x and returns `False`. This is done so we are able to use the finger for subsequent `find(x)` operations and the only way we can guarantee that the finger points to a valid node is if it points to the node just before x.

The pseudocode for this version of `find(x)` is given below:

```
class FingerSkipList:
  initialize():
    head = fsl.Node(None)
    finger = [head] * max_level

  def find(x):
    i = finger[-1]  # start at the node pointed to by the finger
    for level in reversed(range(len(i.next))):
      while i.next[level] and i.next[level].val < x:
        i = i.next[level]
    if i.next[0] and i.next[0].val == x:
      finger = [head] * max_level  # reset the finger to the head
      return True
    finger = [None] * len(i.next) + [i]
    # update the finger to point to the last node before x
    return False
```

[2] Adapted from Exercise 4.10 in the textbook.

Part III
Implementing a Database Index using a Skiplist

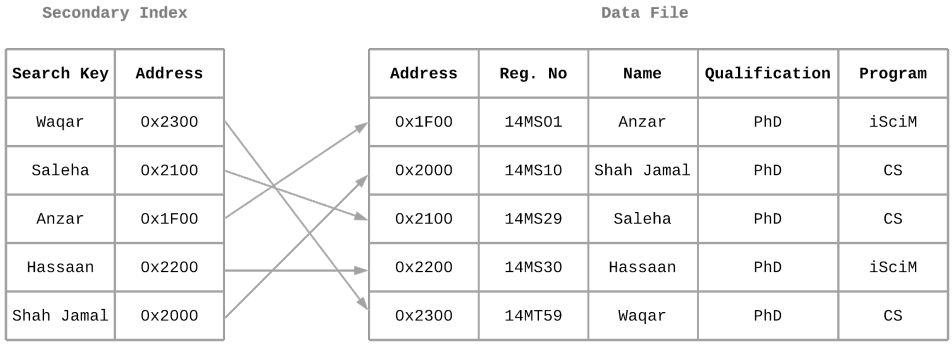

Figure 1: An index over the Name attribute of a table. The index stores values of the Name attribute as keys with the associated value being the address of the corresponding record in the table.

You have learned about database indexes in CS 355 Database Systems. Indexes in databases are similar to indexes in books. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a book is a sorted list of words with the page numbers that contain each word. An index in a database is a sorted list of key values with the storage locations of rows in the table that contain the key value. Each key in the index is associated with a particular pointer to a record in the data file. Figure 1 shows an index on the *Name* attribute of an *Instructor* table stored in a data file. Note that key values in the figure are not sorted.

The most popular data structure used for indexing in relational databases is Btree (or its variant, such as B+tree). Btrees rose to popularity because they do fewer disk I/O operations to run a lookup compared to other balanced trees. SingleStore is the first commercial relational database in production today to use a skiplist, not a Btree, as its primary index data structure for in-memory data [1].

You are given a data file, `data/books.csv`, that contains records of books. Each record contains the following attributes:

- Book Code
- Title
- Category
- Price
- Number of Pages

For some of the attributes, all the values are unique. Such attributes can be used to index the table. Others have repeating values and are thus unsuitable to be used for indexing.

2 Implementation Details and Tasks

You are required to provide the missing implementations for the methods in the files, `skiplist.py` and `db.py`, in the accompanying `src/` sub-folder. `skiplist.py` contains classes to implement a skiplist. `db.py`

contains a `Table` class to store records which can be indexed using a skiplist. Every individual field in the `Table` is stored as strings. Further details are provided in the doc strings in the files. The methods to be implemented are identified with a `pass` in their body.

2.1 Requirement

You will need to install the `typing` module to support certain type hints used in the code.

2.2 Tips

Below are some tips to avoid the errors that have previously caused tests to fail. Following these may save you many frustrating hours of debugging!

- Store each field that is read from file as a `str`.
- Store a record as a list of field values.
- Store a table as a collection of records.
- Make sure that the `Table` can be re-indexed during run-time.
- Take care that the keys for the index are the values of the specified attribute. And that the value associated with each key is not the corresponding record, but a means to locate the record in the table.
- For the `select_range` query, think of an approach better than submitting multiple `select` queries.

2.3 Testing

Once you have successfully implemented the methods, you can test your code by reading from the accompanying data file, `data/books.csv`, and performing queries on it. For grading purposes, your submission will be tested automatically by GitHub using the accompanying `pytest` file, `test_index.py`.

References

[1] The Story Behind SingleStore's Skiplist Indexes, `https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/`, last accessed on 16 Feb 2022.