CSCI 2270: Data Structures - Project Specifications

IMDB top 1000 movie database using Hashtables and Skiplists

Due Sunday, April 30, 11:59 PM

1 Introduction

The IMDB-Movie-Database project is aimed at developing data structures and algorithms to manage a database of movies from IMDB. The project provides header files, parser code, and sample data in CSV format, but not the implementation of the hash tables and skip lists. Students are encouraged to choose their own hash collision algorithm and implement the skip list data structure using the pseudocode provided.

The goal of this project is to provide students with an opportunity to practice data structures and algorithms while developing a functional database application. The project will also encourage students to extend the basic functionality and the data structures beyond what is asked, allowing them to showcase their programming skills to future employers while looking for internships.

2 Core Features

The basic functionality of the IMDB-Movie-Database project includes the ability to search for movies based on their title and director. The project should be able to display the director of a movie, the number of movies directed by a given director, the description of a movie, and a list of movies directed by a given director.

2.1 Data Structures

The IMDB-Movie-Database project uses two primary data structures: a hash table and a skip list. The hash table is used to store and retrieve movies based on their title. The skip list is used to store directors and their associated movies, allowing the project to quickly retrieve information about movies directed by a given director.

2.1.1 Hash Table

The hash table uses a hash function to compute an index for a given movie title. If two movie titles hash to the same index, a collision occurs. Students are encouraged to choose their own hash collision algorithm (chaining, linear probing, quadratic probing, or double hashing) to minimize collisions for the given IMDB-Movie database. To allow for both kind of implementations (chaining and open addressing), the `MovieNode` object provides a `next` pointer to chain the `MovieNodes`. For open addressing based implementations, this next pointer will always be assigned to `nullptr`. Students **must** use their identikey to come up with a creative hash function that minimizes collisions for the given IMDB-Movie database to ensure efficient insertion and retrieval of movie nodes.

2.1.2 Skip List

The skip list is a data structure that provides fast search and insert operations on a sorted list of data. The skip list is similar to a linked list, but it includes additional "skip" pointers that allow for faster search and insertion times. The skiplist is probabilistic data structure that allows for efficient search, insertion, and removal operations with $O(\log n)$ time complexity. This makes it a powerful tool for storing and accessing large datasets, and an important data structure for computer scientists to master.

2.2 Overview

You have been provided with starter code. The contents of the starter code are as follows:

- driver.cpp: The driver program including code to parse IMDB-1000 csv file
- MovieNode.hpp: Declaration of the MovieNode structure
- MovieHashTable.hpp: Provides the MovieHashTable class declaration
- MovieHashTable.cpp: To complete the MovieHashTable implementation
- DirectorSkipList.hpp: Provides the DirectorSkipList class declaration
- DirectorSkipList.cpp: To complete the DirectorSkipList implementation
- IMDB-Movie-Data.csv and IMDB-small.csv: sample input files

2.3 Implementation Requirements

2.3.1 Preliminary Steps

Your `main()` function should accept four command-line arguments: the program name, csv input file, the hash table size and the skiplist size. If the user doesn't execute the

program using the correct format, prompt them to run the program again and display the usage. For example:

```
Invalid number of arguments.
Usage: ./<program name> <csv file> <hashTable size> <skipList size>
```

Note: you are provided 2 csv files: IMDB-Movie-Data.csv and IMDB-small.csv.

You are provided a parsing function`MovieNode* parseMovieLine(string line)` in `driver.cpp` to read movies from the `csv` file and construct a `MovieNode` object on heap. Next, create an instance of MovieHashTable and DirectorSkipList and set their sizes to the one specified by the user. Also, you **must** report the number of hashcollisions after populating the MovieHashTable database.

2.3.2 User Interface

Once the program is executed, display a menu that looks as follows:

```
Number of collisions:276
Please select an option:
1. Find the director of a movie
2. Find the number of movies by a director
3. Find the description of a movie
4. List the movies by a director
5. Quit

Enter an Option:
```

Note that your actual menu may look different if you have added functionality. This menu should be displayed to the user after every input so that the user can continuously run menu options. Each menu item is detailed below.

2.4 Find the director of a movie

Given the title of a movie, search the MovieHashTable to return the name of the director of the movie.

```
Please select an option:
1. Find the director of a movie
2. Find the number of movies by a director
3. Find the description of a movie
4. List the movies by a director
5. Quit
```

```
Enter an Option: 1
Enter movie name: La La Land
The director of La La Land is Damien Chazelle
```

2.5 Find the number of movies by a director

Given the name of a director, search the DirectorSkipList to return the number of the movies by the director in the database.

```
Please select an option:
1. Find the director of a movie
2. Find the number of movies by a director
3. Find the description of a movie
4. List the movies by a director
5. Quit

Enter an Option: 2
Enter director name: Damien Chazelle
Damien Chazelle directed 2 movies
```

2.6 Find the description of a movie

Given the title of a movie, return a description of the movie including plot detail, list of actors, year, and genre. You can use any formatting to organize this information.

```
Please select an option:
1. Find the director of a movie
2. Find the number of movies by a director
3. Find the description of a movie
4. List the movies by a director
5. Quit

Enter an Option: 3
Enter movie name: La La Land
Summary: La La Land is a 2016 (Comedy,Drama,Music) film featuring "Ryan
   Gosling, Emma Stone, Rosemarie DeWitt, J.K. Simmons".
Plot: "A jazz pianist falls for an aspiring actress in Los Angeles."
```

2.7 List the movies by a director

Given the name of a director, search the DirectorSkipList to list all of the movies by the director in the database.

```
Please select an option:
1. Find the director of a movie
2. Find the number of movies by a director
3. Find the description of a movie
4. List the movies by a director
5. Quit

Enter an Option: 4
Enter director name: Damien Chazelle
Damien Chazelle directed the following movies:
        0: La La Land
        1: Whiplash
```

2.8 Exit

When the user decides to exit, your program will call the destructors. Your destructors should free all memory from the hash tables, the SkipList, and finally, exit the program. Your program should not have any memory leaks. We will use Valgrind to detect memory leaks. You can use the Debugging Guide provided earlier in the semester for instructions on how to do this. Note that while MovieNodes are shared by both data structures, HashTables 'own' these nodes and hence their destructor should free the memory used by MovieNodes.

2.9 Miscellaneous

Your code should be well-documented. Every file should have a commenting section at the top and every function should have a comment section above it stating the name of the function, its purpose, the input parameters, and the output parameters. Consider using the Google C++ Style Guide. A common style used for functions is as follows:

```
/**
```

```
 * hash - calculates the hash of a key
 *
 * @param title
 * @return int
 */
int HashChaining::hash(int title) {
...
}
```

3 Project Submission and Grading

3.1 Deliverables

Submit the following files to the master branch of your GitHub repo.

- driver.cpp: The driver program including code to parse IMDB-1000 csv file
- MovieNode.hpp: Declaration of the MovieNode structure
- MovieHashTable.hpp: Provides the MovieHashTable class declaration
- MovieHashTable.cpp: Completed the MovieHashTable implementation
- DirectorSkipList.hpp: Provides the DirectorSkipList class declaration
- DirectorSkipList.cpp: Completed the DirectorSkipList implementation
- IMDB-Movie-Data.csv and IMDB-small.csv: sample input files
- Edit the README.md file within your GitHub repo and explain the following:
  - Which hash collision resolution method did you use and why?
  - Explain your hash function.
  - Did you implement skiplist to search for director specific information? If not, what alternative data structure did you use, and why?
  - Explain any additional features your have implemented.

3.2 Interview Grading

There will be mandatory interview grading for this project. It is your responsibility to schedule an interview with their TA (scheduling links will be provided soon). We expect that you will be able to successfully answer all questions asked. If a student does not complete the interview grading, they will receive a 0 as their project score.

3.3 Coding Conduct

You are required to work individually on this project. All code must be written on your own. You are not allowed to use any code from the Internet or post this project to web sites such as Chegg. You are also not allowed to post this project to public GitHub repos. See the Collaboration Policies section within the Syllabus for more information.

You **must** utilize, at a minimum, everything that exists in the starter code. For example, your solution must use hash tables and skip lists and you must use all of the provided functions. The parameter types should not be modified either. Your task is to populate the functions within the starter code but you are welcome to add helper functions as needed. You can add additional functionality by adding new header and implementation files to add new data structures.

4 Potential Extensions

To further extend the functionality of the IMDB-Movie-Database project, students can consider implementing additional data structures or algorithms, such as:

- Fuzzy search: Implement a fuzzy search algorithm (Hamming Distance or Levenstein Distance) that can handle spelling mistakes and return suggestions for movie titles or director names that closely match the user's query.
- Bloom filters: Implement a bloom filter data structure to efficiently check if a movie title or director name exists in the database, without having to perform a full search.
- Graph database: Implement a graph database to compute the **Kevin Bacon** distance between actors in different movies. This could involve parsing additional data from the IMDB dataset, such as actor names.
- Recommendation system: Implement a recommendation system that suggests movies to users based on their viewing history or ratings.
- Advanced search: Implement an advanced search feature that allows users to filter movies based on criteria such as genre, year, runtime, rating, and revenue.

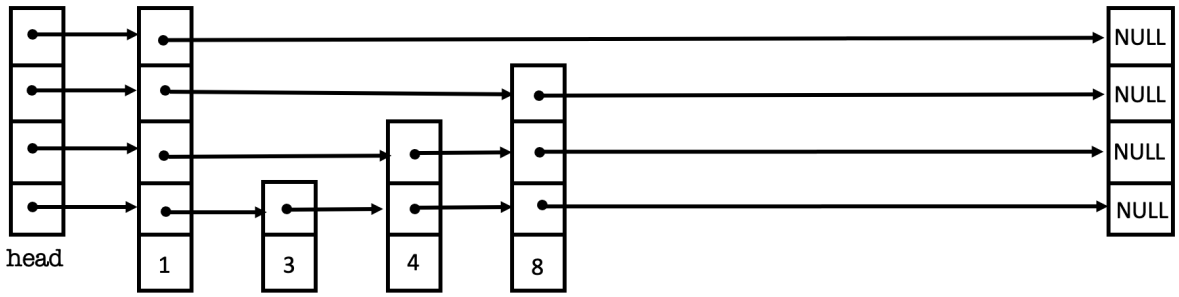

Figure 1: A skiplist with four keys $\{1, 3, 4, 8\}$.

5 Introduction to SkipLists

The skiplist is a data structure that provides $O(\log n)$ complexity (average case) for insert, search, and remove operations, making it an excellent choice for implementing lists. This performance is comparable to that of a balanced binary search tree like the red-black tree, but skiplists are easier to implement. However, it's important to note that skiplists are probabilistic data structures that use random coin tosses during the insertion process, which means the discussed complexities are expected time over these random coin tosses. In this project, you will be implementing the insert and search operations for skiplists.

5.1 What is a Skiplist?

A skiplist is a sorted linked list with multiple layers, where each layer is a subset of the nodes from the layer below it. The base layer (layer 0) contains all nodes in sorted order, and each subsequent layer includes a subset of the nodes from the previous layer chosen randomly using a biased coin with probability $p = \frac{1}{2}$ (or even $\frac{1}{4}$). A quick calculation[1] can show that each node can appear on average in $\frac{1}{1-p} = 2$ lists. Also, the expected number of layers[2] in a skiplist of size n equals $\log_2 n$.

Let's understand the search and insert operations in a skiplist. The multilane highway analogy can be used to explain the concept of traversing a skip list while searching for a node. In this analogy, a skiplist is compared to a multi-lane highway, and nodes in the skiplist are compared to vehicles on the highway. Suppose you need to search for a node

[1] with probability $(1-p)$ in only 1 list, with probability $p(1-p)$ in 2 lists, with probability $p^2(1-p)$ in 3 lists, and so on. Hence the expected number of lists a node can appear equal

$$\sum_{i=1}^{\infty} ip^{i-1}(1-p) \quad = \quad (1-p)\sum_{i=1}^{\infty} ip^{i-1} = (1-p)\frac{1}{(1-p)^2} = \frac{1}{(1-p)}.$$

[2] If there are n nodes at the base layer, there will be (on average) $n/2$ nodes at layer 1, $n/4$ nodes at layer 2, and so on. Hence the expected number of layers be equal to $\log_{\frac{1}{p}} n = \log_2 n$.

with a specific key value in the skiplist. If you start from the slowest lane (layer 0) and the node doesn't exist, you may need to traverse all the nodes in the list, which would be inefficient ($O(n)$). However, the skip list structure allows for a more efficient search strategy. Instead of starting from the slowest lane, you can start from the highest lane (fastest lane) and proceed horizontally until the key of the next node in that lane is greater than or equal to the target key. If the next node has a key equal to the target, the node has been found. If it is greater than the target, or the search reaches the end of the list at the current layer, the process is repeated from the current node but from a lower layer. This process is repeated until the last lane is searched.

This strategy is analogous to overtaking cars on a multi-lane highway: you start on the fastest lane and move to the slower lanes when you find a car with a higher speed than yours. Similarly, in a skiplist, you start from the highest level and move to the lower levels only when the node you are looking for cannot be found at the current level.

What is the expected length of *search path*? An easier way to count is to consider the reverse path from a target node to the root node at the highest layer. This path is such that it either stays at the same layer or moves up. The probability of staying in the same layer is p and hence the expected number of steps in the same layer before moving up the layer is $\frac{1}{1-p}$. Since the total number of layers are $\log_{\frac{1}{p}} n$, the expected length of the search path is $\frac{1}{1-p} \log_{\frac{1}{p}} n = 2 \log_2 n = O(\log n)$.

The insertion works in a similar way. While a node can be inserted to the right place in a sorted order in $O(n)$ by traversing the lowest layer, the search of the right place to insert the node can be reduced to $O(\log_2 n)$ by following the previous strategy. Once the node has been inserted, it can be weaved in previous layers by repeatedly tossing the coin (probability p) to decide if it should be also inserted in the previous layer.

5.2 Skiplist: Data Structure and Algorithms

Instead of having multiple linked lists as shown in Figure 1, it is common to implement skiplists as a single linked list with nodes having multiple "`next`"pointers, implemented as a vector of next pointers, depending upon how many layers that node appears in (see, Figure 2). The first node is a "dummy" sentinel node whose key is some out of range value (say -1). Various indices of the vector of node pointers characterize various layers.

Here is an example of a node of the skiplist:

```
struct SLNode {
  int key;
  vector<SLNode*> next;
};
```

A skiplist is simply a linked list of `SLNode` objects shown below.

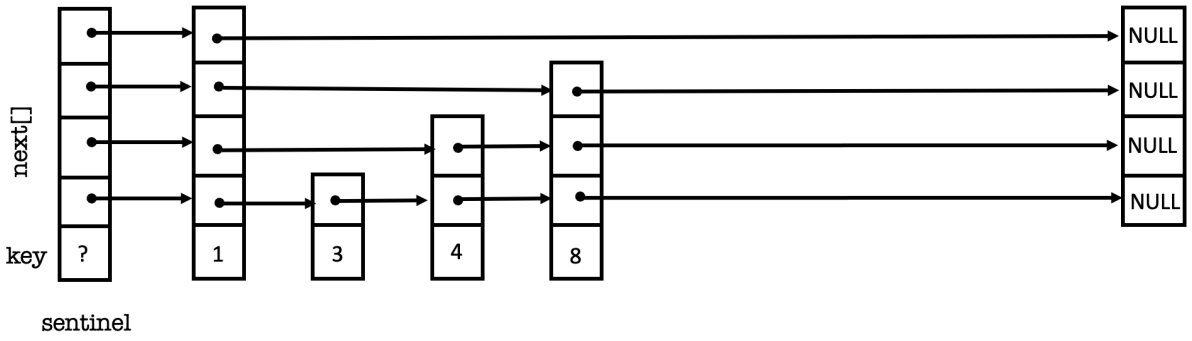

Figure 2: A list based representation of skiplist with four keys $\{1, 3, 4, 8\}$ from Figure 1. Node with key 1 appears in four layers while the node 3 appears only in one layer. The very first node is a dummy sentinel node.

```
class SkipList {
    private:
        int max_levels;  // the maximum number of levels
        SLNode* head;   // a pointer to the head node

    public:
        // Constructors should insert the sentinel node in the Skiplist.
        SkipList() {
            max_levels = DEFAULT_LEVELS;
            head = new SLNode();
            head->key = -1; // some dummy value
            head->next = vector<SLNode*>(max_levels, nullptr);
            // sentinel node is present at every level where next pointer
   vector at every level is initalized as null pointers.
        }
        SkipList(int _levels);
        ~SkipList();
        void insert(int key);
        SLNode* search(int key);
};
```

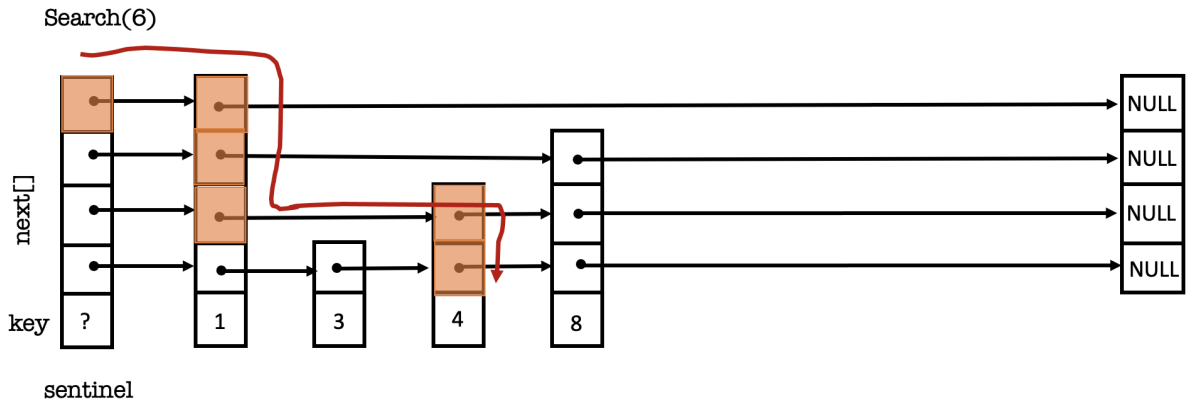

Figure 3: Search in a skiplist.

5.2.1 Search

The search for a key begins by initializing a (`SLNode * curr`) variable to the sentinel node, and iterating from the highest level. At each level as long as the next node is not null and the key of the next node is smaller than the search key, the `curr` variable simply will be updated to the next node at the current level. Here is the pseudocode.

```
1. Set curr to head
2. For i from maxlevels - 1 down to 0, do the following:
    a. While curr's next node at level i is not null and its key is less
   than search key:
        i. Set curr to curr's next node at level i
    b. Set prev[i] to curr
3. Set curr to curr's next node at level 0 // Why is this important?
4. If curr is not null and its key equals key, then return curr
5. Otherwise, return null
```

5.2.2 Insert

While inserting a new node, we need to decide how many levels this node will be appearing in. This can be done by tossing coins repeatedly until the first `tail`. In C++ this can be accomplished using the following code:

```
int n_levels = 1;
while (n_levels < max_levels && rand() % 2 == 0) {
    n_levels++;
}
```

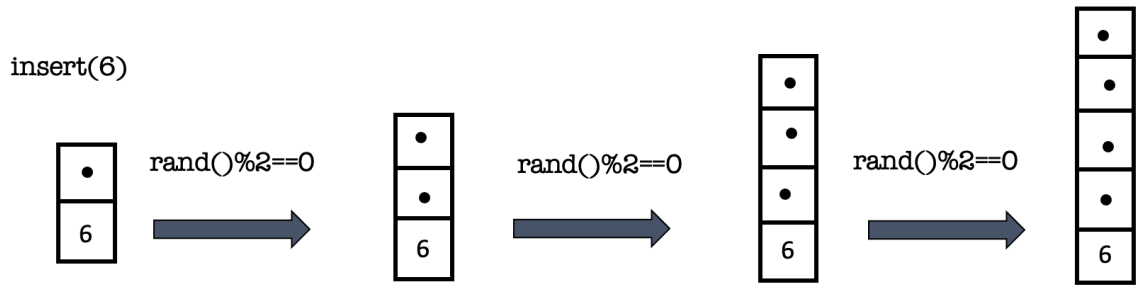

Figure 4: Repeatedly tossing a coin to decide the number of layers. In this case, three consecutive coin tosses showed head (the upper bound on the number of layers). Hence this node is to be inserted at every level.

After creating an appropriate new node, the search for the place where this node needs to be inserted follows the search algorithm. To remember previous nodes of the newly created node at various levels, one should remember the previous pointers at various levels during the search by remembering the node where level got decremented. It is shown in figure 5. The rest of the code is simply updating the previous pointers to insert the new node at appropriate levels.

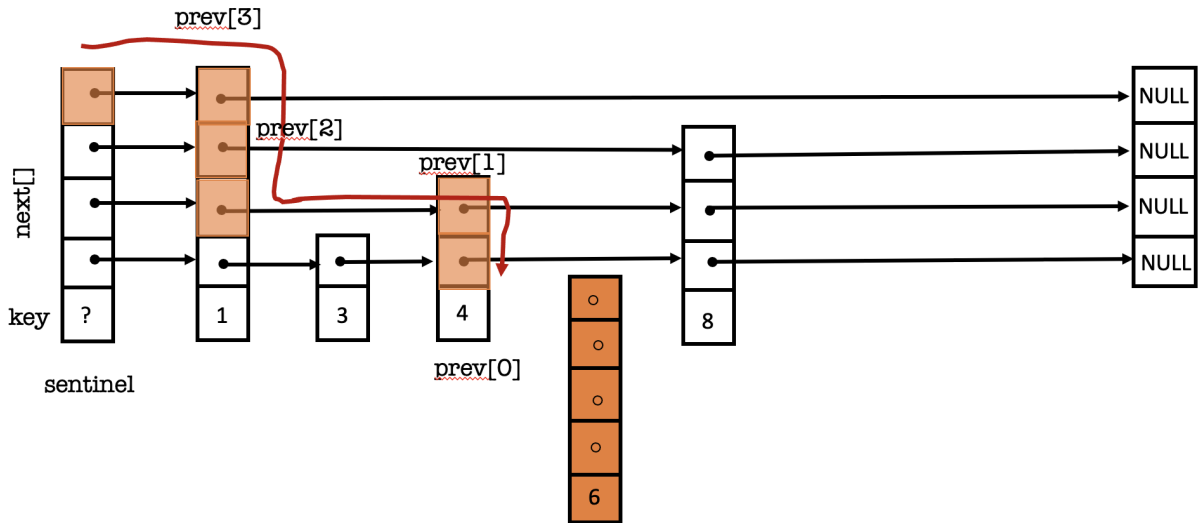

Figure 5: Inserting a new node within SkipList.