Programming Project 2024-25

Version 4 - December 5, 2024

This project requires you to implement a Priority Queue using a skip list, a probabilistic data structure introduced in 1989 by William Pugh for the efficient implementation of the Map ADT [1].

1 Skip Lists

Let M be a collection of (key, value) pairs with distinct keys (in the project you will have to remove this assumption). A skip list S for M consists in a series of lists $\{S_0, S_1, \ldots, S_h\}$, with $h \geq 1$, which contain elements of M and obey the following constraints:

- Each S_i is sorted in increasing order, and it is delimited by two sentinels with keys $-\infty$ and $+\infty$, respectively.
- S_0 contains all elements of M, in addition to the two sentinels.
- For every $0 \leq i < h$, the entries in S_{i+1} are a subset of those in S_i. In particular, if $e \in S_i$ then $e \in S_{i+1}$ with probability α, independently of the other entries (in typical implementations $\alpha = 0.5$).
- S_h contains only the two sentinels.
- For every entry e, including the sentinels, the positions containing its occurrences in $S_0, S_1, \ldots$ are connected to form a list.

The data structure can be viewed as a two-dimensional collection of positions threaded both vertically and horizontally, and can be accessed through its top-left position s, referred to as *start position*. An example is shown Figure 1.

The two-dimensional list structure implies that from each position $p \in S_i$, with $0 \leq i \leq h$, the following positions can be directly accessed in $O(1)$ time:

- `next`(p): the successor of p in S_i, which is `null` if p is the right sentinel;
- `prev`(p): the predecessor of p in S_i, which is `null` if p is the left sentinel;
- `above`(p): the position above p in S_{i+1} containing the same entry, which is `null` if such a position does not exist.
- `below`(p): the position below p in S_{i-1} containing the same entry, which is `null` if $i = 0$.

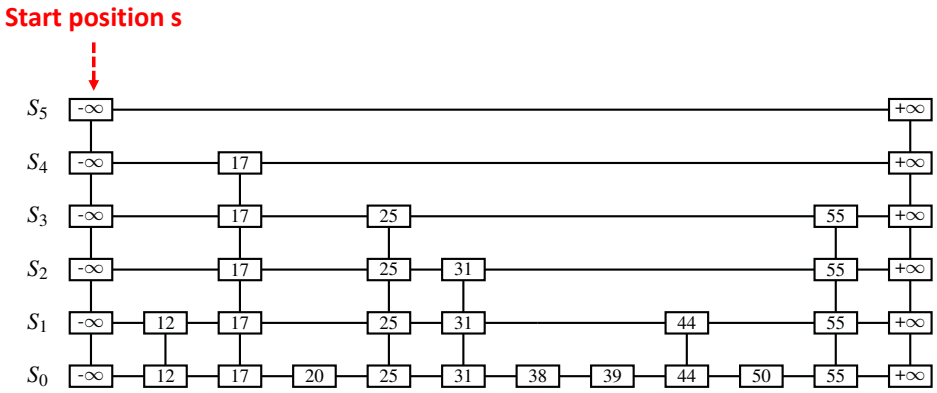

Figure 1: Example of skip list from [1].

1.1 Search for a key k

Searching for a given key k aims to locate the position $p \in S_0$ that contains the entry with the largest key $\leq k$. The search algorithm, called `SkipSearch`(k), begins at s and, for each list S_i, with $i = h, h-1, \ldots, 0$, proceeds to the position p in S_i that contains the largest key $\leq k$. Then, if $i > 0$, it moves to position `below`(p) in S_{i-1} and continues the search in S_{i-1}; if $i = 0$, it stops and returns p. (See [1, Code Fragment 10.14] for the pseudocode of `SkipSearch`(k).) Examples are given in Figures 2 and 3.

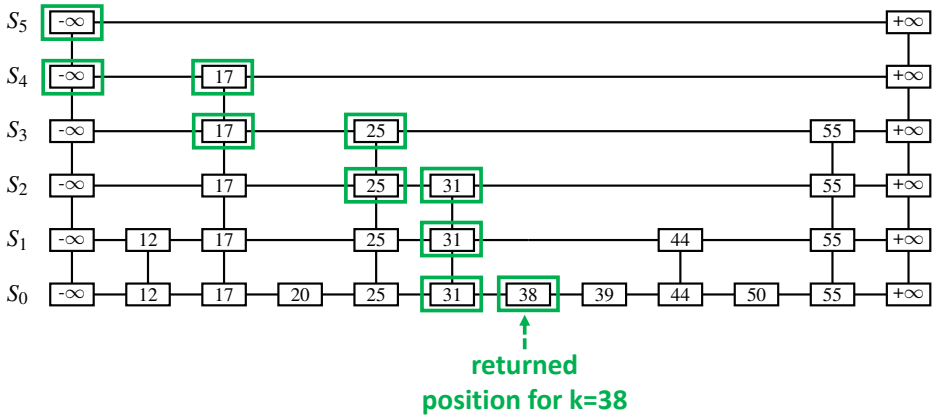

Figure 2: Search of $k = 38$: green nodes are those traversed by the search algorithm.

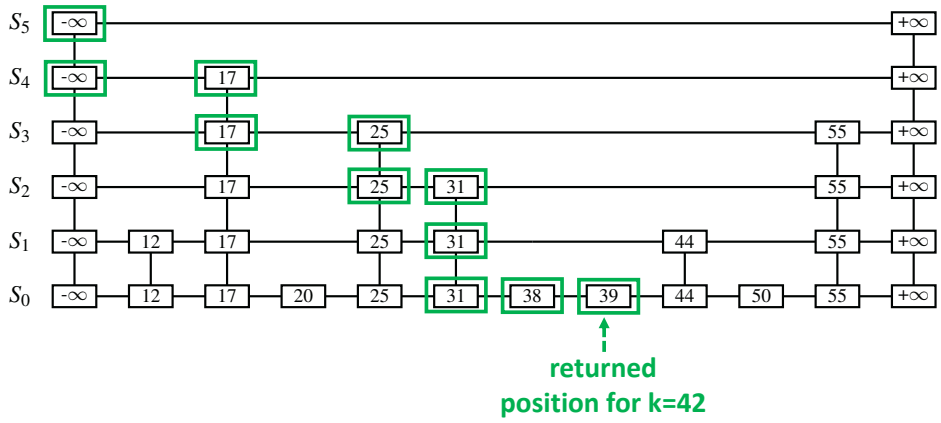

Figure 3: Search of $k = 42$: green nodes are those traversed by the search algorithm.

1.2 Insert an entry with key k

When an entry e with key k is inserted into a skip list S, first `SkipSearch`(k) is run. For $h \geq i \geq 0$, let p_i denote the rightmost position of S_i reached by `SkipSearch`(k). If p_0 contains an entry e' with key k and the skip list requires entries with distinct keys (as in a Map), then occurrences of e' in p_0 and nodes above it are replaced with the new entry e. In all other cases, a new position q_i containing e is inserted between p_i and `next`(p_i), for $0 \leq i \leq \ell$, where ℓ is set equal to the number of consecutive *heads* in independent coin tosses until a *tail* is obtained.[1] Each coin toss yields a *head* with probability α. The new positions q_i are connected vertically, with q_i serving as the predecessor of q_{i+1} for $0 \leq i < \ell$. An example is shown in Figure 4 (adapted from [1]).

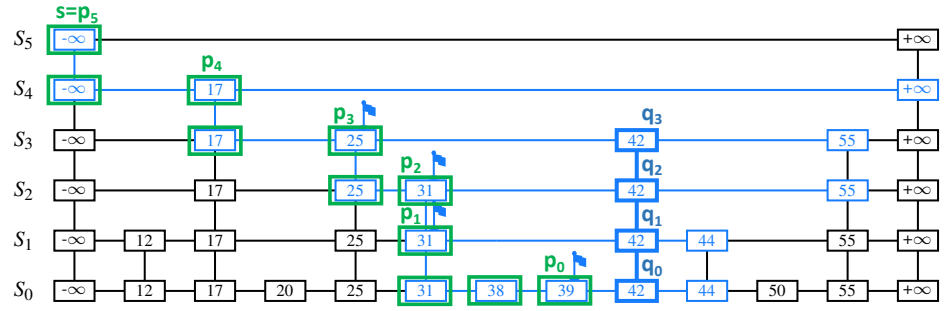

Figure 4: Insertion of $k = 42$: blue flags indicate positions p_i's preceding the newly added q_i's, with $0 \leq i \leq \ell$ (in this case $\ell = 3$).

In case $\ell \geq h$, new lists S_j, with $h + 1 \leq j \leq \ell + 1$ are added to the structure, where $S_{\ell+1}$

[1] For example, if the coin tosses result in *head*, *head*, *head*, *tail*, then $\ell = 3$.

contains only the two sentinels, and S_j, with $h+1 \leq j \leq \ell$, contains the two sentinels and q_j. Also, the starting position s is updated to the first sentinel of $S_{\ell+1}$. (See [1, Code Fragment 10.15] for the pseudocode of the insert algorithm.)

1.3 Efficiency of the skip list methods

It is known that when $\alpha \leq 1/2$ is a constant independent of n, then, with high probability, the complexity of the search, insertion, and deletion methods is $O(\log n)$ and that the total number of nodes among all lists is $O(n)$. Moreover, the probability α regulates a space-time trade-off. For $\alpha \in (0, 1/2)$, the space increases with α while the running time of the methods decreases with α. For details on the Skip Lists and their analysis refer to [1, Section 10.4] (there, the complexity of the methods is analyzed only in expectation).

2 Assignment for 2 points

You must implement a Priority Queue using a skip list. For simplicity, we assume that the keys of the entries are of type `Integer` and the values are of type `String`. Specifically, you must design the following 3 classes.

1. A class `MyEntry`, whose instances are the entries of the Priority Queue, namely (`Integer`, `String`) pairs.
2. A class `SkipListPQ`, whose instances are skip lists containing entries of class `MyEntry`. The class is instantiated with the α parameter, with $\alpha \in [0, 1)$, and must define the following Priority Queue methods. For an instance S of the class:
   - `size`(): returns the number of entries in S.
   - `min`(): returns an entry of S with minimum key and prints the key and the value of the entry separated by space.
   - `insert`(key, string): inserts a new entry e = (key, string) in S, based on the algorithm described before. Since S implements a Priority Queue, the new entry e must always be added to S, even if S contains other entries with the same key. In order to generate the maximum index ℓ of a list where e be added, you can invoke method `generateEll`(α, key), where α is the head probability of a coin toss. The code of `generateEll` will be available in the Moodle Exam page of the course. If $\alpha = -1$, the method computes ℓ as a (deterministic) function of the key; we will use this setting for checking correctness.
   - `removeMin`(): removes and returns an entry of S with minimum key. After removing the entry, the method must also do the following. Let $j \geq 0$ be the largest

index such that S_j contains some entry, besides the sentinels; if the skip list becomes empty, let $j = 0$. Hence $S_{j+1}, S_{j+2}, \ldots, S_h$ contain only the two sentinels. In this case, the lists of index $j + 2$ or larger are removed, and h is set equal to $j + 1$. In this fashion, we keep the lists $S_0, \ldots, S_{j+1}$, where S_{j+1} only contains the two sentinels.

- `print`(): prints all entries in S in the order in which they appear in the bottom list S_0 (i.e., by increasing key). Entries are printed on the same line, separated by comma (","). Each entry is printed together with the size of the vertical list corresponding to that entry. For example, for an entry (15,car) that occurs in lists $S_0, S_1, \ldots, S_7$, the method will print

15 car 8

3. A class `TestProgram` that receives, as a command-line argument, the path to a text file that specifies the operations that the program must execute. The file is structured as follows:

   - Line 1 contains N (the number of operations to be performed) and α (the head probability for coin tosses);
   - Line $1 + r$, for $1 \leq r \leq N$, specifies the r-th operation to be performed, using an integer code: 0 for `min`, 1 for `removeMin`, 2 for `insert`, and 3 for `print`. If the code is 2, the same line contains the integer and the string that must be given in input to the method, separated by space.

   After acquiring the path to the file and opening the file, the program reads parameters N and α from the first line, stores them into suitable variables, and prints their values in output. Then, it defines an empty instance S of `SkipListPQ` and, for $r = 1, 2, \ldots, N$, executes on S the operation specified by Line $1 + r$ of the file.

3 Assignment for 3 points

In order to get full score (3 points) you must integrate the assignment described in the previous section as follows.

- The method `insert`(key,string) must return the number of nodes traversed in the skip list along the path from the starting node s to the node p_0 in the list S_0. For the example depicted in Figure 4 the number to be returned is 11.
- At the end of execution, the program prints the following values in a single line: the value of α, the number of entries in the skip list, the total number of executed inserts, and the average number of nodes traversed per insert (as returned by the `insert` method).

Example:

0.5 1085 67153 12.05

- Run the program `TestProgram` over the 6 input files `alphaEfficiencyTest_m_i.txt`, with m = 10K,100K and i = 1,2,3, which represent inputs with m insert to be executed with different values of α ($\alpha \in \{0, \frac{1}{4}, \frac{1}{2}\}$). The files are available on the Moodle Exam website (inside the `alphaEfficiencyTest.zip` archive). For each input file, write down the statistics printed at the end of the execution of the program (from the previous point), in the text box of the Moodle Exam submission form, one line for each value of the input file.

4 Deliverables and material available in Moodle Exam

You must submit a `TestProgram.java` file containing all classes, using the dedicated link in the Moodle Exam site. Your code cannot call library functions other than those in `java.lang`, `java.io`, and `java.util` packages.

In Moodle Exam you will find the following material:

- Template to be used for the `TestProgram.java` file, which also includes the definition of the method `generateEll` mentioned before.
- Some input files, each paired with a corresponding output file that shows the expected output for that input. You can use these files to test the correctness of your code and to make sure it complies with the specified output format.
- Zip archive `alphaEfficiencyTest.zip` (see Section 3).

IMPORTANT: do not change the name of the java file, and make sure that your program compiles and runs (with the output formatted exactly as in the given examples) using the command `javac TestProgram.java`, otherwise, we may not be able to evaluate your submission.

References

[1] M.T. Goodrich, R. Tamassia, M.H. Goldwasser. *Data Structures and Algorithms in Java*, Sixth Edition. John Wiley and Sons, 2014.