Random Algorithms (RA-MIRI)
Assignment 3

Daniel Benedí

December 2022

During this assignment we are going to carry out an emperical study of the performance of skip lists. Skip lists are a probablistic data structe proposed as an alternative to balanced trees. [2]

1 Implementation

In this section I will explain my implementation with pseudo-code, but the whole implementation using `C++` can be found in the appendix A.

A skip list can be seen as a linked list with different layers where the height of each element depends on a geometric distribution. An example of this can be found in figure 1; first of all, we observe that although the input is not ordered, the elements in the skip list will end up ordered. Also, we see that the higher we go in the skip list, the less amount number of elements there are. This height is a random variable with a geometric distribution.

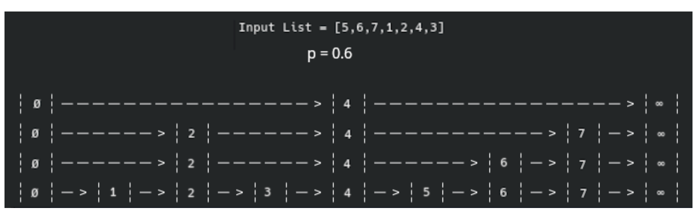

Figure 1: An example of skip list with 7 elements and $p = 0.6$.

First of all, we need to define what a node is (data structure 1). It will contain the data to be stored in the set (its data type is represented by **T**). It also has two lists to represent the elements before and after in each level, this means that $|\text{forward}| = |\text{backward}| = \text{height}$. The list `backward` wouldn't be needed if the operation `remove` was necessary. Also, we need to have a list, called `head`, to represent the first element of each level.

Data Structure 1 Node of skiplist

struct Node
  list forward, backward
  T data
endStruct

Once the data structure is defined, we can start describing the operations that are going to be implemented: `insert`, `remove` and `contains`. Firstly, we will implement an auxiliary operation called `search`. This operation will return the node iff it is in the set or the node before in case it wasn't in the set.

Algorithm 1 Search for a node or the place where it should be

1: **procedure** SEARCH(A)
2: $h \leftarrow |\text{head}| - 1$
3: **while** $h > 0 \wedge \text{head}[h].\text{data} > A$ **do**
4: $h \leftarrow h - 1$
5: **end while**
6: **if** $h < 0$ **then**
7: **return** null
8: **end if**
9: $r \leftarrow \text{head}[h]$
10: **while** $h > 0$ **do**
11: **if** $r.\text{forward}[h] \wedge r.\text{forward}[h].data \leq A$ **then**
12: $r \leftarrow r.\text{forward}[h]$
13: **else**
14: $h \leftarrow h - 1$
15: **end if**
16: **end while**
17: **return** r
18: **end procedure**

In this algorithm, first of all we check if the searched item is bigger than any node in the head (lines 2 to 5), otherwise it means that all the items are bigger than the searched item and therefore it isn't in the set and there is no node lower than it (lines 6 to 8). In there rest of the pseudocode (lines 10 to 16) it will search if in that level there following node exists and if it does also checks whether it is lower or equal than the searched item, in that case the next result node will be the node stored in the forward list at that level. Otherwise, we decrease the level, because we cannot look for anymore in that level.

With this implementation in mind we can easily define the `contains` operation, that would be: This function is trivial because it calls the `search` procedure and it will return `true|` if and only if the returned node is not `null|` and it is equal to the searched element.

Algorithm 2 Returns whether the set contains a element

```
procedure CONTAINS(A)
    r ← SEARCH(A)
    return r ≠ null ∧ r.data = A
end procedure
```

The `remove` operation is also quiet trivial as well, it will call the `search` procedure and if the node contains the searched item, it will assign to the nodes in the backward list the nodes in the forward list and the nodes in the forward list the nodes in the backward list.

Algorithm 3 Removes the element from the set

```
procedure CONTAINS(A)
    r ← SEARCH(A)
    if r ≠ null ∧ r.data = A then
        h ← |r.forward|
        for l = h − 1 to 0 do
            if r.backward[l] ≠ null then
                r.backward[l].forward[l] ← r.forward[l]
                if r.forward[l] ≠ null then
                    r.forward[l].backward[l] ← r.backward[l]
                end if
            else                                        ▷ This means it is in the head
                head[l] ← r.forward[l]
                if head[l] = null then
                    remove top of head
                end if
            end if
        end for
    end if
end procedure
```

Although the `insert` operation is really similar to the `search` operation or the remove one, we cannot use it because we need to insert the new node in the levels as we are decreasing them. This is the reason because we haven not reused it. For the simplicity of the pseudo-code we are going to assume that the element is not it the set already, but in the real pseudo-code we take it into consideration.

As we explained, first of all we insert the element to the new levels of the head if there is new levels. Then we update the levels of the head that are bigger to the new elem and finally we start going down the skip list and we need to decrease a level we insert the node in that point.

We have opted to implement our own geometric distribution generator. We generate u.a.r a number between 0 and 1 (both included) and will increase a

Algorithm 4 Insertion in skip list

Pre: The set doesn't contain the element A

Post: The set contains the element A

```
procedure INSERT(A)
    h ← Geom(p)
    node ← Node(A, h)
    if |head| < h then
        insert the node to the head until it has the size of h
        h ← |head| − |{x ∈ head | x = A}|
    end if
    h ← h − 1
    if h < 0 then
        return
    end if
    r ← head[h]
    while h ≥ 0 ∧ r.data > A do
        Update head element at height h
        h ← h − 1
        r ← head[h]
    end while
    while h ≥ 0 do
        while r.data[h] ∧ r.data[h] < A do
            r ← r.forward[h]
        end while
        node.backward[h] = r
        if r.forward[h] then
            node.forward[h] = r.forward[h]
            r.forward[h].backward[h] = node
        end if
        node.forward[h] = node
        h ← h − 1
    end while
end procedure
```

counter while the generated number is bigger than q. We are aware that in the `random` library there is a geometric distribution generator of integers (adding 1 to the generated number because it starts at 0 and we need to them to start at 1), but we prefer to implement our own version without any technical reason.

2 Experimentation

In order to to the experimentation we have tried different sizes of the amount of input elements, n, all of them with different probability, q, in the Geometric distribution in the construction. In order to get statistically significant result, we have run several times, M, each case shuffling the input every time.

The values for n have been taken from the assignment's statement where it is proposed to do the experiment from $n = 2000$ to $n = 20000$ in steps of 100, so we will try $n = 2000, 2100, 2200, 2300, ...19700, 19800, 19900$ and 20000. For every case we tried all possible $q = 1-p$ with p taking values from 0.1 to 0.9, avoiding the extreme probabilities because they would have a strange behaviour; also taken from the statement. Every case of n and q have been run 100 times so the results are significant and they would have low deviation from the theoretical values.

2.1 Results

Once we have done the experiments, we can plot the experimental total search cost to compare with the theoretical one. In figures 2 and 3 we observe different means of the experiment total search cost compared with the theory. We observe that they are really similar to the theoretical cost. To obtain this results, the total search cost of each experiment has been gathered and then ploted the mean of the 100 experiments for each n and q. The theoretical cost $C_{n,q}$ has been explained in the lectures and in the assignment's statement and it is:

$$Cn, q = \frac{1}{q} n \log_{1/q} n + n \left(\frac{1/q}{\ln 1/q}(\gamma - 1) + \frac{1}{\ln 1/q} - \frac{1/q}{2} \right), \gamma = 0.5772156649$$

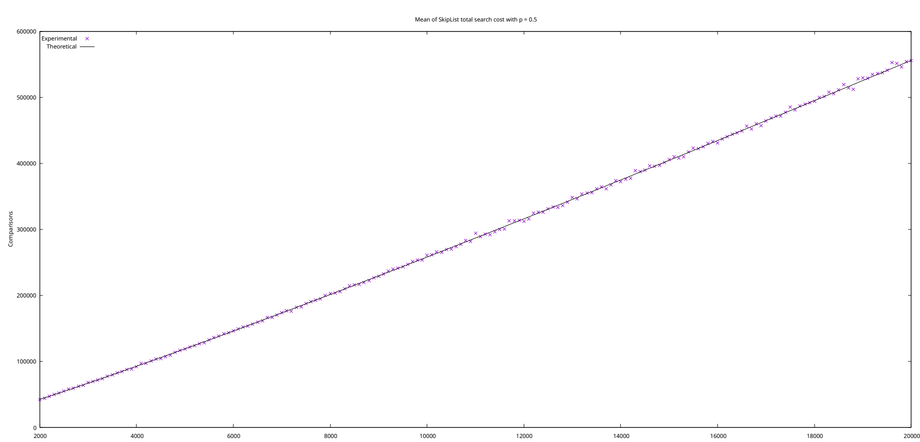

Figure 2: Theoretical and experimental total search cost with $q = 0.5$

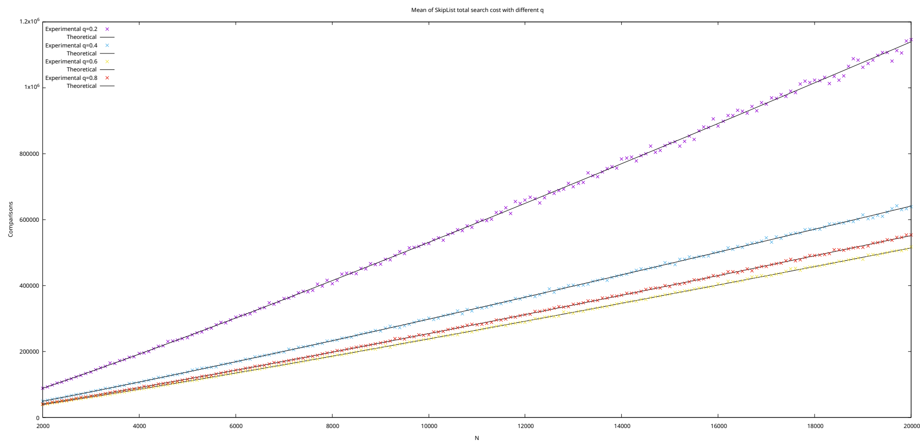

Figure 3: Theoretical and experimental total search cost with $q = 0.2, 0.4, 0.6$ and 0.8

Apparently, the implementation seems to fit the theoretical cost. We propose the R^2 metric, coefficient of determination, to measure how well it fits the theoretical cost. We will assume that our observed data y is the mean cost of our experiments for each input size (we will fix the q parameter) and $\hat{y}$ will be the theoretical values. This metric consist on the proportion of the variation in the dependent variable that is predictable from the independent variable:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \frac{1}{n}\sum_j y_j)^2}$$

It is used to know how well it approximates to the theoretical value compared with the worst possible least-squares predictor (the mean). The higher is the metric the better it approximates. In table 1, we observe that all the values are really close to 1 which means that our experiments seems to corroborate the theoretical expectation of the total search cost.

q	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
R^2	0.99676	0.99761	0.99837	0.99926	0.99968	0.99983	0.99960	0.99911	0.99774

Table 1: $R^2 of the experiments with respect to the theoretical values$

Similarly, to what we have done with the expected total search cost, we can do an analysis of its variance. Peter Kirschenhofer and Conrado Martinez did a detailed analysis of the total search cost (succeded and unsucceded) of the skiplist including its expectation and **variance** [1]. In their article, they give a complex formula for the variance, but also some factors K to approximate the variance assintotically with:

$$Var(C_{n,q}) = K * n^2$$

This values can be found in their original work [1]. In our experiments we have obtained results that are really close to this theoretical analysis. In figures 4 and 5 we observe that effectively they follow this theoretical analysis as expected, but it is important to notice that as n grows, the standard deviation is more disperse around the expected value, this probably indicates that we should have done more experiments at least as n was getting bigger.

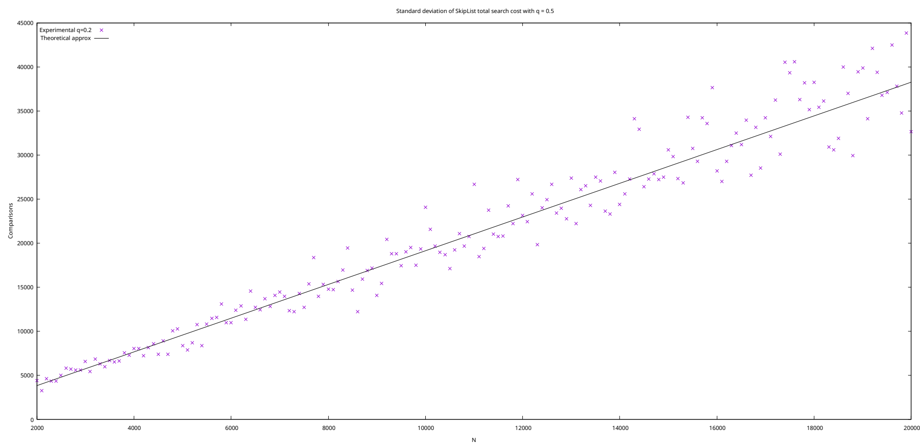

Figure 4: Theoretical and experimental total search cost standard deviation with $q = 0.5$

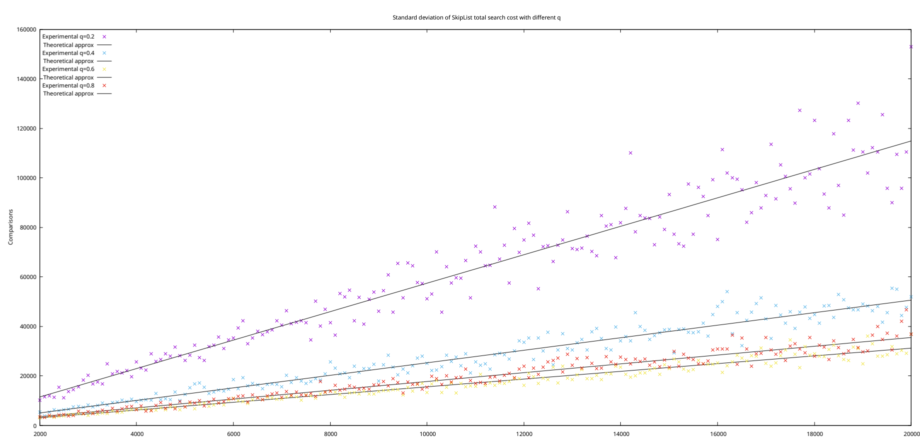

Figure 5: Theoretical and experimental total search cost with $q = 0.2, 0.4, 0.6$ and 0.8

When analyzing data structures not only their efficiency should be taken into account, but also their memory usage, because usually it is a trade off. From the assignment's statements we have that:

$$S_{n,q} = \frac{n}{1-q} + \log_{1/q}(n) = E[\text{sum of the height of all nodes}] + E[\text{size of head}]$$

Note that the size of the head will be equal to the maximum of the heights of all nodes. We could do a theoretical analysis of the variance of the memory usage, taking into account that the nodes' heights are independent and identically distributed geometric variables, X, and we will define the size of the head as Y. Although we know that $Var(X) = n^2 \frac{1-p}{p^2}$ and could use an approximation to find the $Var(Y)$ using an article by Szpankowski and Rego [3], we are not going to do this analysis because it is beyond the analysis of this work.

In figure 6, we observe in the first two rows the mean memory use of the skip list with respect to the parameter q and, as we expected, the mean memory usage follows the theoretical analysis. We know that the standard deviation should be lineal because $Var(X) = O(n^2)$ and $Var(Y)$ seems to be $O(n^2)$ ass well according to the previous article [3], and of course our standard deviation is lineal in the experiments as we expected, but as before it is quite disperse probably because we should have had to increase the number of experiments as n was growing.

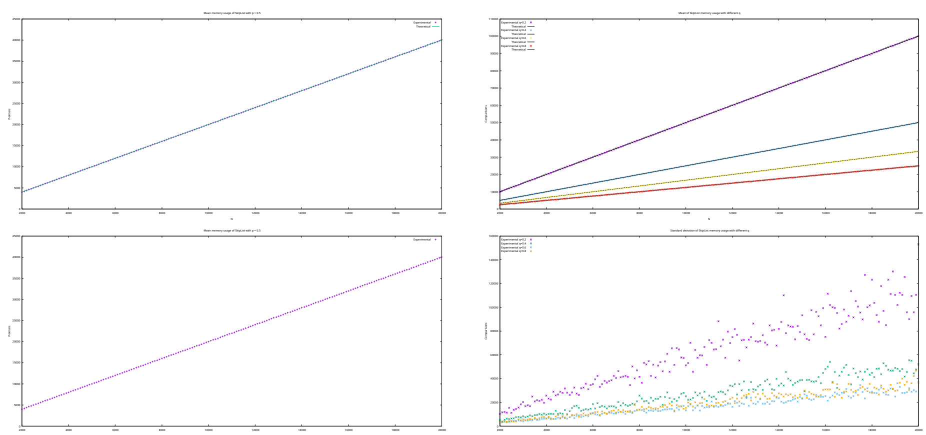

Figure 6: Mean and standard deviation of memory usage of Skip List.

3 Conclusion

Overall, we can conclude that the Skip List performs really nice with respect their average case and with a good performance. The total search cost seems to be linear in the number of items which is a really good performance for a data structure. The implementation was really simple, compared with self-balancing binary search tree. It is remarkable that as the number of elements in the skip list increased, the memory usage and the performance got more disperse around the theoretical values and therefore, one could think that when using it with a big amount of elements its good performance might not be guaranteed for single use cases.

Appendices

A C++ Implementation

```cpp
//**********************************************
// File:    Set.h
// Author: Daniel Benedi Garcia
// Date:    8th December 2022
// Coms:    Set class that implements a skip list
//          and a lot of associated utilities
//**********************************************
#pragma once
#include <vector>

template <typename T>
class Set {
  public:
    /**
     * Constructor.
     *
     * Creates an empty Set, sets the skip list parameter q to the
     * given value
     *
     * @param q    double. Represents the probability of an element
     *         not having
     */
    Set(double q) : q(q) {}

    /**
     * Destructor.
     *
     * Clear all the extra structures used.
     */
    ~Set(){
      Node* elem = head[0];
      while(elem != nullptr){
        Node* next = elem->forward[0];
        delete elem;
        elem = next;
```

```
    }
  }

  /**
   * Insert element x in the set.
   *
   * @param x   Type depends on template. The element will be added
   *            to the Set.
   */
  void insert(const T& x){
    if(contains(x)) return;

    int height = randomHeight();
    Node* new_elem = new Node(x, height);

    // If it is the first element in that height, add it to the head
    while(head.size() < new_elem->height){
      head.push_back(new_elem);
      height--;
    }
    height--;

    // It might happen that it is the fisrt one added
    if(height < 0) return;

    Node* tmp = head[height];
    while(height >= 0 && tmp->data > x){
      new_elem->forward[height] = tmp;
      tmp->backward[height] = new_elem;
      head[height] = new_elem;
      height--;
      if(height >= 0)
        tmp = head[height];
    }
    while(height >= 0){
      // While for that level there is a smaller element
      // and exist and following node, go forward
      while(tmp->forward[height] && tmp->forward[height]->data < x)
        tmp = tmp->forward[height];

      new_elem->backward[height] = tmp;
```

```
        if(tmp->forward[height]) {
          new_elem->forward[height] = tmp->forward[height];
          tmp->forward[height]->backward[height] = new_elem;
        }
        tmp->forward[height] = new_elem;
        height--;
      }
    }

    /**
     * Removes the element x from the set if present.
     *
     * @param x    Type depends on template. The element will be removed
     *         from the set.
     */
    void remove(const T& x){
      Node* elem = search(x);

      if(elem && elem->data == x){
        for(int lvl = elem->height-1; lvl >= 0; lvl--){
          if(elem->backward[lvl]){
            elem->backward[lvl]->forward[lvl] = elem->forward[lvl];
            if(elem->forward[lvl]){
              elem->forward[lvl]->backward[lvl] = elem->backward[lvl];
            }
          }else{
            //If there is no previous element, that means it is in the head
            head[lvl] = elem->forward[lvl];
            if(!head[lvl]){
              while(lvl + 1 > head.size())
                head.pop_back();
            }
          }
        }
        delete elem;
      }
    }

    /**
```

```
     * It will return wheter the element is in the set.
     *
     * @param x    Type depends on template. The element will be searched
     *         in the set.
     * @return    It will return true if the element is in the set, false
     *         otherwise.
     */
    bool contains(const T& x) const{
      Node* res = search(x);
      return res && res->data == x;
    }

    /**
     * Returns the total search cost.
     *
     * @return    It will return the toal search cost.
     */
    int total_search_cost() const{
      int cost = 0;
      Node* next = head[0];
      while(next != nullptr){
        int height = head.size() - 1;
        while(height > 0 && head[height]->data > next->data){
          cost++;
          height--;
        }

        Node* res = head[height];
        while(height >= 0){
          cost++;
          if(res->forward[height] && res->forward[height]->data <= next->data){
            res = res->forward[height];
          }else{
            height--;
          }
        }

        next = next->forward[0];
      }

      return cost;
```

```
    }

    /**
     * Returns the number of pointers used, that is
         the sum of theheight
     * of all nodes in the skip list.
     *
     * @return    Total number of pointers.
     */
    int number_pointers() const{
      Node* next = head[0];
      int size = head.size();
      while(next != nullptr){
        size += next->height;
        next = next->forward[0];
      }

      return size;
    }

    double get_q() const {
      return q;
    }

    int nodes_lvl(int lvl) const {
      if(lvl >= head.size()) return 0;

      int nodes = 0;
      Node* node = head[lvl];
      while(node != nullptr){
        nodes++;
        node = node->forward[lvl];
      }

      return nodes;
    }

    friend std::ostream& operator<< (std::ostream&
       stream, const Set& set) {
      // Print HEAD
      for(int i = 0; i < set.head.size(); i++)
        stream << "—— "; stream << std::endl;
      for(int i = 0; i < set.head.size(); i++)
        stream << "    "; stream << std::endl;
      for(int i = 0; i < set.head.size(); i++)
        stream << "—— "; stream << std::endl;
```

```
    // Print BODY
    Node* node = set.head[0];
    while(node != nullptr){
      for(int i = 0; i < set.head.size(); i++)
        stream << " |  "; stream << std::endl;
      for(int i = 0; i < node->height; i++)
        stream << " v  ";
      for(int i = node->height; i < set.head.size(); i++)
        stream << " |  "; stream << std::endl;
      for(int i = 0; i < node->height; i++) stream << "---- ";
      for(int i = node->height; i < set.head.size(); i++)
        stream << " |  "; stream << std::endl;
      for(int i = 0; i < node->height; i++)
        stream << " " << node->data << "  ";
      for(int i = node->height; i < set.head.size(); i++)
        stream << " |  "; stream << std::endl;
      for(int i = 0; i < node->height; i++)
        stream << "---- ";
      for(int i = node->height; i < set.head.size(); i++)
        stream << " |  "; stream << std::endl;

      node = node->forward[0];
    }

    // Print END
    for(int i = 0; i < set.head.size(); i++)
      stream << " |  "; stream << std::endl;
    for(int i = 0; i < set.head.size(); i++)
      stream << " v  "; stream << std::endl;
    for(int i = 0; i < set.head.size(); i++)
      stream << "---- "; stream << std::endl;
    for(int i = 0; i < set.head.size(); i++)
      stream << "      "; stream << std::endl;
    for(int i = 0; i < set.head.size(); i++)
      stream << "---- "; stream << std::endl;

    return stream;
  }
private:
  struct Node{
```

```
    const T& data;
    Node** forward;
    Node** backward;
    int height;
    Node(const T& data, int height) : data(data), height(height) {
      forward = new Node*[height];
      backward = new Node*[height];

      for(int i = 0; i < height; i++){
        forward[i] = nullptr;
        backward[i] = nullptr;
      }
    }

    ~Node(){
      delete[] forward;
      delete[] backward;
    }
  };


  // data members
  std::vector<Node*> head; // First element for every height
  double q; // Probability of not having another level

  // Internal procedures
  Node* search(const T& x) const{
    int height = head.size()-1;
    while(height > 0 && head[height]->data > x) height--;
    if(height < 0) return nullptr;

    Node* res = head[height];
    while(height >= 0){
      if(res->forward[height] && res->forward[height]->data <= x){
        res = res->forward[height];
      }else{
        height--;
      }
    }

    return res;
```

```
    }

    int randomHeight(){
        static std::random_device rd;
        static std::mt19937 gen(rd());
        static std::uniform_real_distribution<> dis
            (0.0,1.0);

      int height = 1;
      while(dis(gen) > q) height++;

      return height;
    }
};
```

```
//*********************************************
// File:    assignment3.cpp
// Author: Daniel Benedi Garcia
// Date:    8th December 2022
// Coms:    Main file of assignment 3 with functions
//          to do the experiments and debug.
//*********************************************

#include <iostream>
#include <sstream>
#include <vector>
#include <fstream>
#include <random>
#include <algorithm>
#include "Set.h"

#define M 100

void debug(){
  std::random_device rd;
  std::mt19937_64 g(rd());
  int n = 7, i = 60;

  // Initialize vector with increasing numbers
  // and then shuffle it
  std::vector<int> elems(n, 0);
  std::iota(elems.begin(), elems.end(), 1);
  std::shuffle(elems.begin(), elems.end(), g);


```

```
  std::cout << "Input = [" << elems[0];
  for(int i = 1; i < elems.size();i++)
    std::cout << "," << elems[i];
  std::cout << "]" << std::endl;

  double q = i/100.;
  Set<int> set(q);

  // Insert the numbers
  for(int i = 0; i < elems.size(); i++)
    set.insert(elems[i]);

  int nodes;
  std::cout << "levels = ";
  for(int lvl = 0; nodes=set.nodes_lvl(lvl); lvl++)
    std::cout << nodes << ", ";
  std::cout << std::endl;

  std::cout << "Skiplist" << std::endl;
  std::cout << set << std::endl;

  std::cout << "Total search cost = " << set.total_search_cost() << std::endl;
}

void experiment(){
  std::random_device rd;
  std::mt19937_64 g(rd());

  std::ofstream cost("total_search_cost.csv");
  std::ofstream mem("total_memory_use.csv");

  #pragma omp parallel for collapse(2)
  for(int n = 2000; n <= 20000; n += 100){
    for(int i = 10; i < 100; i += 10){
      double q = i/100.;

      std::stringstream buf1, buf2;
      buf1 << n << "," << q;
      buf2 << n << "," << q;

      std::vector<int> elems(n, 0);
      std::iota(elems.begin(), elems.end(), 1);
      for(int m = 0; m < M; m++){
        Set<int> set(q);

```

```
        std::shuffle(elems.begin(), elems.end(), g);

        for(int i = 0; i < elems.size(); i++)
          set.insert(elems[i]);

        buf1 << "," << set.total_search_cost();
        buf2 << "," << set.number_pointers();
      }

      buf1 << std::endl;
      buf2 << std::endl;

      #pragma omp critical
      cost << buf1.str();

      #pragma omp critical
      mem << buf2.str();
    }
  }
}

int main(int argc, char* argv[]){
#ifdef DEBUG
  debug();
#else
  experiment();
#endif
}
```

References

[1] Peter Kirschenhofer, Conrado Martínez, and Helmut Prodinger. "Analysis of an optimized search algorithm for skip lists". In: *Theoretical Computer Science* 144.1 (1995), pp. 199–220. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)00296-U. URL: https://www.sciencedirect.com/science/article/pii/030439759400296U.

[2] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

[3] W. Szpankowski and V. Rego. "Yet another application of a binomial recurrence order statistics". In: *Computing* 43.4 (1990), pp. 401–410. DOI: 10.1007/bf02241658.