Advanced Data Structures: Skip Lists

Constantino Gómez and Cristobal Ortega
{name.surname}@est.fib.upc.edu

June 2016

1 Implementation

Our implementation of the skip lists are based on the material provided in the Advanced Data Structures course[2], therefore our skip list class is composed of nodes with the following characteristics:

- integer key
- string value
- List of nodes corresponding to the different levels

```
1     struct Node {
2         int key;
3         std::string value;
4
5         std::vector<Node*> _next;
6         Node (int k, const std::string& v, int level);
7     };
```

Listing 1: Class Node

For simplification, we have fixed the data type our skip lists can work with. In our implementation, all the keys are Integers and the values are Strings.

Also, instead of having a lists of pointers to nodes representing the different levels we can have, we simplified this mechanism as a vector with a maximum height (this has a limitation: we cannot have more levels than MAX_LEVELS).

Even with these changes, we used the material provided in the course as a base to code the functions we had to develop for the assignment and the

additional functions we consider that are needed to the basic usage of the skip lists. Our implementation includes this following data members and functions:

- void insert (int key, string value): as in any data structure we need to be able to insert new elements
- void erase (int key): as in any data structure we need to be able to delete elements
- bool contains (int key): in order to check if inserts are done correctly, we coded this function to check if *key* is in the list
- string find (int key): we need to retrieve information from the data structure, even more, for this assignment we need to be able to measure the cost of finding
- int total_search_cost(): requirement of the assignment
- int number_pointers(): requirement of the assignment
- int randomLevel(): because of how skip lists work we need to provide a random function to get the height of new nodes created
- int getLevel(vector<Node*>& x): because the levels can differ we need an auxiliary function to evaluate the height of the given *key*
- void print(): coded to be able to debug the data structure, it prints the lowest level (the most complete list) of the data structure and prints for each *key* its level.

The different functions that go through the data structure (insert, erase, contains, find, etc.) are almost the same as the snippet shown in the material [2]:

```
1       int l = actualLevel;
2       while ( l>=0 ){
3         if (x->_next[l] == nullptr || key <= x->_next[l]->key){
4           to_update[l] = x;
5           --l;
6         }
7         else {
8           x = x->_next[l];
9         }
10      }
11      x = x->_next[0];
```

Listing 2: Snippet to iterate through the skip list

With the snippet shown above we are able to get the position where the pair *<key, value>* should be or should not be placed. After, finding the position and depending on what function was called we will do different things: return true if it exists, return the value, etc.

The code of our skip lists implementation can be obtained from the appendixes of this work and also from our Github repository[1].

2 Experimental framework

We describe the hardware characteristics of the machine where this data structure has been tested in Table 1

Processor	Intel(R) Core(TM) i7-4600U
Frequency	2.1 GHz
L1	64 KB
L2	512 KB
Memory	8 GB

Table 1: Machine specifications

We use a main program to execute several tests and measure the different metrics required for the assignment:

Total cost of searching. We need to search for every element and compute its cost, C(n,k), and for the total cost it will be the addition of all individual costs of searching: $C(n) = \sum_{k=1}^{n} C(n,k)$

Total usage of memory. We can count the number of pointers that we use for storing the data, at the end we would need to multiply each pointer to its real size on memory (key, value, etc.)

In order to obtain these metrics values, we added auxiliar functions: *total_search_cost* and *number_pointers*. With those values, we analyze and compare different skip lists configurations varying the size *(N)* and the probability to copy the node to the upper level *(p)*

Comment: we repeat each experiment 50 times and get the average to minimize the impact of outliers in our results.

3 Results

In this section we discuss the results obtained experimenting with our implementation of skip lists.

From now on, we use the term skip list size to refer to the total number of elements contained in a skip list. In figure 1 we show the total search cost in seconds ($C(n) = \sum_{k=1}^{n} C(n,k)$) varying the size of skip list and p factor; meaning that for each size of skip lists ranging from 1K to 10K with 1K increase, we try different [0 to 1; 0.1++] values of p factor. The p factor determines the probability of copying an element to an upper level.

We easily identify different trends:

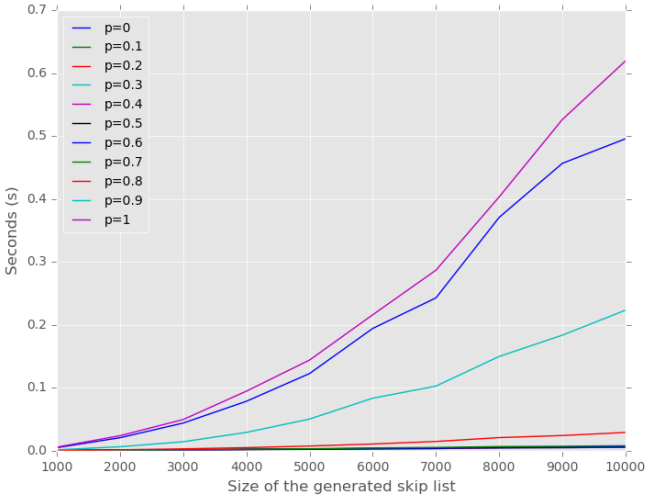

Figure 1: Total search cost for different sizes and p

- For extreme values of p searches become slower. The reason for this behavior is due to how we build the skip list, with extreme values of p we get a *flat* list:
  - At $p = 0$, elements are never copied to an upper level; resulting in a list with elements only in the 1st level.
  - At $p = 1$, elements are **always** copied to an upper level (for this reason we have a *MAX_LEVELS*, so our data structure does not grow to infinity).
- We focus on non-extreme p values on figure 2. As we see, all lines exhibit very similar behavior in terms of performance.

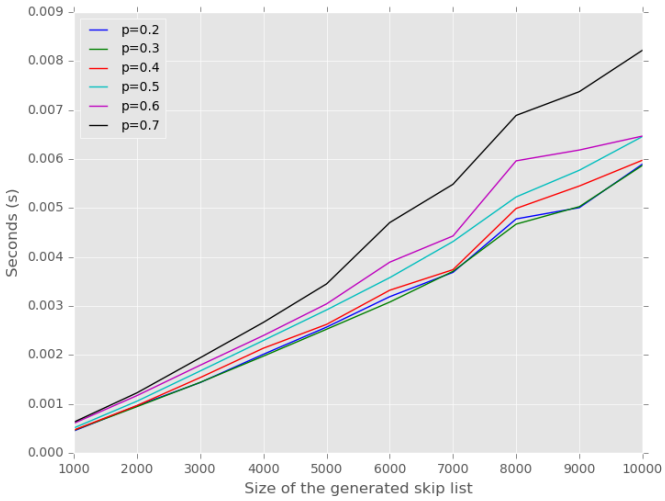

Figure 2: Total search cost for different sizes and p (without extreme values of p)

To reason about memory space consumption, for different p and skip list size values, we measured the total number of pointers allocated in each skip list configuration. Figure 3 shows our results varying p and skip list size the same way as in the previous experiment.

We observe that for high values of p we are using more pointers, therefore more memory space. This has an easy explanation: a higher value of p more the possibility to copy the element to an upper level, and in consequence we will have more copies of the element (that as seen in figure 1 can be useful for reducing the search cost).

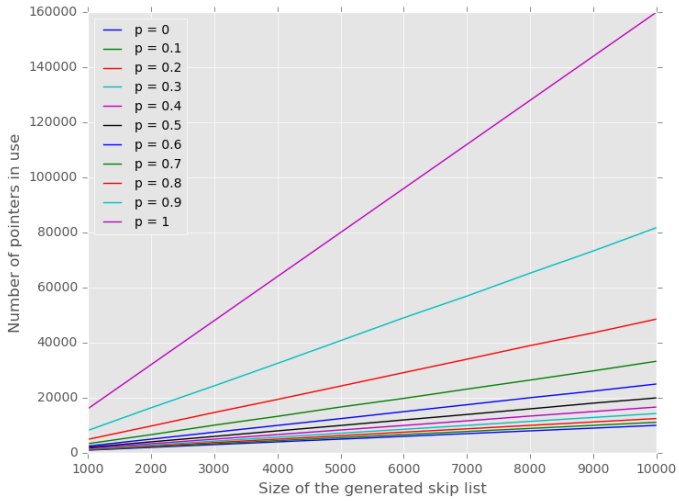

Figure 3: Total number of used pointers for different sizes and p

In this case, we filter high values of p because they are not representative (see figure 4).

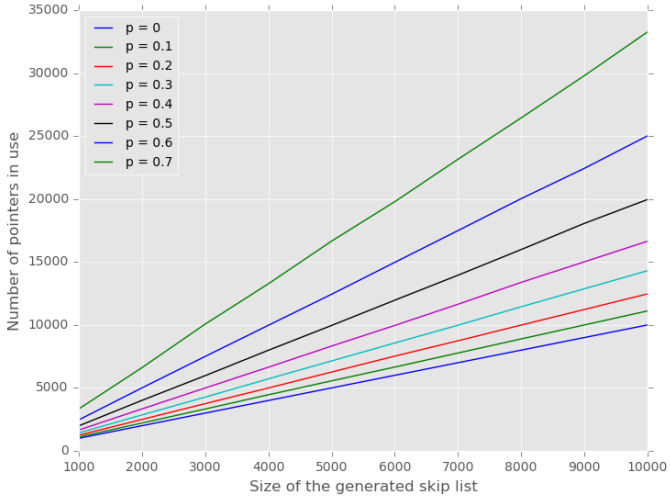

Figure 4: Total number of used pointers for different sizes and p (without high values of p)

3.1 Theoretical comparison

Following, we compare our implementation against the theorical skip list characteristics:

- Total Search cost: $n \times \log n$
- Space in memory: n

We compare search cost in figure 5 and space in memory in figure 6. Both of them are in logarithmic scale to see better how our implementation has the same growing shape. They only differ in a ratio, probably because on how we implemented our version.

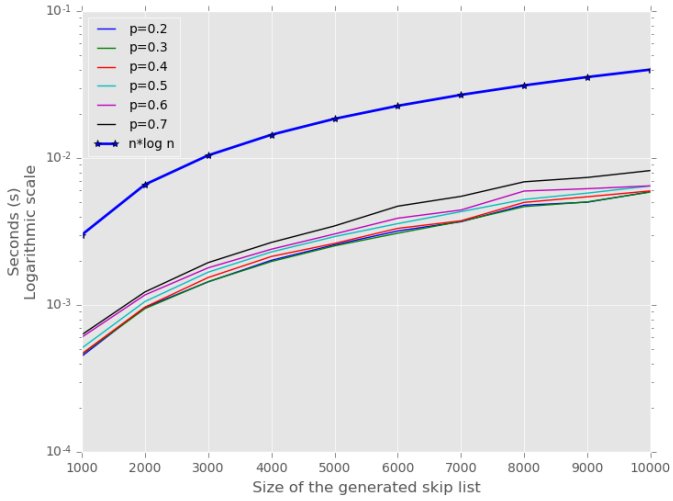

Figure 5: Total search cost for different sizes and p (without extreme values of p) and the theoretical function

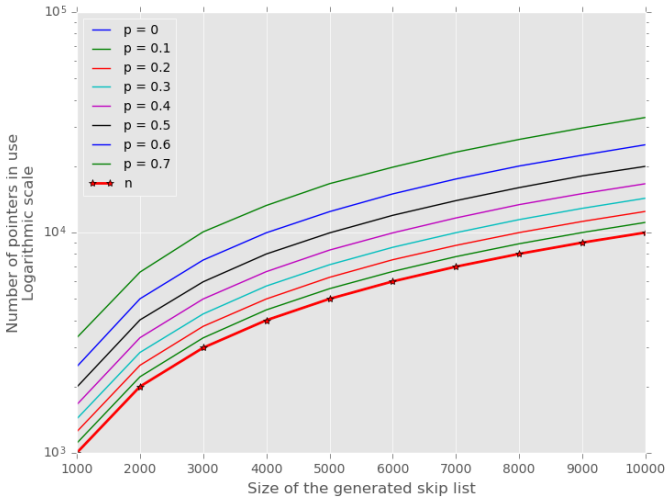

Figure 6: Total number of used pointers for different sizes and p (without high values of p) and the theorical function

Additionaly, we tested how the probability p affects our total search cost, the experiments realized are represented in the figure 7. We made 2 large skip lists (n=10000 and n=50000) and compared their total search cost against the theoretical one represented as the function: $\frac{-1}{q \ln q}$.

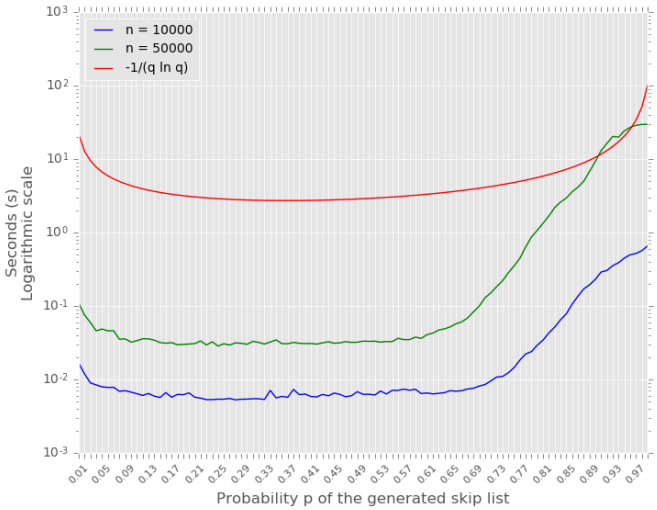

Figure 7: Total search cost for different probabilities p

As we can observe in figure 7 it seems to be a global minimum for the total search cost, in the next table 2 we wrote down the probability p that offers the most performance in terms of total search cost, in the table we can see that we are close to the minimum theoretical (even lower), this may be an artifact on the implementation or the experiments: (1) our implementation can have corner cases where we are not getting the good node, even we have debug several cases (coded in the main.cpp (appendix C)) or (2) our experiments are very short in repetitions, we had to lower the repetitions for the case when *N=50000* due to high execution time, therefore we can be seeing problems with extreme cases that lower our average time.

N	p with minimum total search cost
10000	0.33
50000	0.28
Theoretical	0.37 ($\frac{1}{e}$)

Table 2: Probability p that offers the best total search cost

4 Conclusion

After finalizing this assignment, we do have some final remarks to conclude. Skip lists seem a good alternative to that data structures that are more often used: their behavior it is similar to those alternatives (and also the idea behind them it is a quite interesting one) as we have proven in the previous section 3 Actually, we had a great time implementing this data structure because we never heard about them before and as we already said, we think this data structure is interesting. And, overall we think we are close enough to the theoretical values and that makes our implementation good to be used in the future.

References

[1] C. Gomez and C. Ortega. Skip list class in c++. `https://github.com/frankicia/skip_list/`, 2016.

[2] Conrado Martínez. *Slides from Advanced Data Structures course*. FIB, 2016.

A skipList.hpp

```
#ifndef SKIP_LIST_H
#define SKIP_LIST_H
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>
#include <random>
#include <cmath>
#include <chrono>
#include <sys/time.h>
#include <iomanip>
#include <unistd.h>

struct Node {
    int key;
    std::string value;
    std::vector<Node*> _next;

    Node (int k, const std::string& v, int level);
};

class skipList {
  public:
    skipList();
    skipList(double probability);
    ~skipList();

    void print();

    void insert (int key, std::string v);
    void remove (int key);
    std::string find(int key);
    bool contains (int key);

    //Support for ADS
    long long total_search_cost();
    int number_pointers();


    Node* head;
    Node* NIL;

  private:
    int maxLevel;
    double prob;
    int _height;
    long long _lastFind;
    long long _totalSearchCost;
```

```
53
54      int randomLevel();
55      int getLevel (std::vector<Node*>& x);
56  };
57
58  #endif
```

Listing 3: Code from file skipList.hpp

B skipList.cpp

```
1   #include "skipList.hpp"
2
3   skipList::skipList() {
4     this->prob = 0.5;
5     this->maxLevel = 16;
6     int key = std::numeric_limits<int>::min();
7     head = new Node(key, "head", maxLevel);
8
9     key = std::numeric_limits<int>::max();
10    NIL = new Node(key, "NIL", maxLevel);
11
12    for(unsigned int i = 0; i < maxLevel; i++) {
13      head->_next[i] = NIL;
14    }
15    //std::cout << "Created" << std::endl;
16    _height = 0;
17    _lastFind = 0;
18    _totalSearchCost = 0;
19  }
20
21  skipList::skipList(double probability) {
22    this->prob = probability;
23    this->maxLevel = 16;
24    int key = std::numeric_limits<int>::min();
25    head = new Node(key, "head", maxLevel);
26
27    key = std::numeric_limits<int>::max();
28    NIL = new Node(key, "NIL", maxLevel);
29
30    for(unsigned int i = 0; i < maxLevel; i++) {
31      head->_next[i] = NIL;
32    }
33    //std::cout << "Created" << std::endl;
34    _height = 0;
35    _lastFind = 0;
36    _totalSearchCost = 0;
37  }
38
39  Node::Node(int k, const std::string& v, int level) {
40    key = k;
```

```
41    value = v;
42    for(unsigned int i = 0; i< level; ++i)
43      _next.emplace_back(nullptr);
44  }
45
46  skipList::~skipList() {
47    //MEMORY IS FREE! :D
48    //But leakage problem is real :(
49  }
50
51
52  int skipList::getLevel( std::vector<Node*>& x ) {
53    int aux = 0;
54    int max = std::numeric_limits<int>::max();
55
56    //We are in NIL status
57    if (x[0]->key == max) {
58      return aux;
59    }
60
61    int result;
62    for (unsigned int i = 0; i < x.size(); ++i) {
63      if (x[i] != nullptr && x[i]->key != max){
64        ++aux;
65      }else{
66        break;
67      }
68    }
69    return aux;
70  }
71
72  void skipList::print() {
73    std::cout << "==========" << std::endl;
74    std::cout << "Printing information about skip list" << std::
          endl;
75    std::cout << "Probability: " << this->prob << " Max Level: " \
76      << this->maxLevel << std::endl;
77
78
79    Node* list = head;
80
81    std::cout << "HEAD address: " << &head << std::endl;
82    std::cout << "NIL address: " << &NIL << std::endl;
83
84    std::cout << "===PRINT AS A VECTOR===" << std::endl;
85    while (list->_next[0] != nullptr) {
86      std::cout << "Address: " << &list->_next[0] << std::endl;
87      std::cout << "Key: " << list->_next[0]->key \
88        << " value: " << list->_next[0]->value \
89        << " level: " << getLevel(list->_next) \
90        << std::endl;
91
92      list = list->_next[0];
93
94    }
```

```
95
96
97
98
99
100   std::cout << "==========" << std::endl;
101
102 }
103
104
105 void skipList::insert (int key, std::string v) {
106   //std::cout << "Inserting Key: " << key << " with value: " <<
          v << std::endl;
107   int actualLevel = getLevel(head->_next);
108   std::vector<Node*> to_update(head->_next);
109
110   Node* x = head;
111
112   int l = actualLevel;
113   while ( l>=0 ){
114     if (x->_next[l] == nullptr || key <= x->_next[l]->key){
115       to_update[l] = x;
116       --l;
117     }
118     else {
119       x = x->_next[l];
120     }
121   }
122
123   if ( x->_next[0] == nullptr || x->_next[0]->key != key) {
124    int level = randomLevel();
125    int height = getLevel(to_update);
126    if ( level > height ) {
127      for (unsigned int i = height; i < level - 1; ++i) {
128       to_update[i] = head;
129      }
130    }
131    x = new Node(key, v, level);
132
133     for (unsigned i = 0; i < level; ++i) {
134       x->_next[i] = to_update[i]->_next[i];
135       to_update[i]->_next[i] = x;
136     }
137   }
138   else {
139     x->_next[0]->value = v;
140   }
141
142   return;
143 }
144
145 void skipList::remove (int key) {
146   //std::cout << "Removing Key: " << key << std::endl;
147   int actualLevel = getLevel(head->_next);
148   std::vector<Node*> to_update(head->_next);
```

```
149
150   Node* x = head;
151
152   int l = actualLevel;
153   while ( l>=0 ){
154     if (x->_next[l] == nullptr || key <= x->_next[l]->key){
155       to_update[l] = x;
156       --l;
157     }
158     else {
159       x = x->_next[l];
160     }
161   }
162   x = x->_next[0];
163
164   if ( x == nullptr) {
165     //we did something wrong
166     std::cout << "Removing a nullptr" << std::endl;
167     return; //we like return in voids
168   }
169
170   if (x->key == key) {
171     int height = getLevel(x->_next);
172     for (int i = 0; i <= to_update.size(); i++) {
173       if( to_update[i]->_next[i] != x)
174         break;
175
176       to_update[i]->_next[i] = x->_next[i];
177     }
178     delete x;
179
180   }
181
182 }
183
184 std::string skipList::find(int key) {
185   timeval start,end;
186   gettimeofday(&start,NULL);
187
188   int actualLevel = getLevel(head->_next);
189   std::vector<Node*> to_update(head->_next);
190
191   Node* x = head;
192
193   int l = actualLevel;
194   while ( l>=0 ){
195     if (x->_next[l] == nullptr || key <= x->_next[l]->key){
196       to_update[l] = x;
197       --l;
198     }
199     else {
200       x = x->_next[l];
201     }
202   }
203   x = x->_next[0];
```

```
204
205
206
207   std::string result;
208   if ( x == nullptr) {
209     //we did something wrong
210     std::cout << "Searching in a nullptr :D" << std::endl;
211     result = "";
212   }
213
214   if (x->key == key) {
215     result = x->value;
216   } else {
217     result = "";
218   }
219
220   gettimeofday(&end,NULL);
221   this->_lastFind = (end.tv_sec - start.tv_sec)*1000000 + (end.
          tv_usec - start.tv_usec);
222   this->_totalSearchCost += this->_lastFind;
223
224   return result;
225 }
226
227 bool skipList::contains (int key) {
228   //std::cout << "Searching Key: " << key << std::endl;
229   int actualLevel = getLevel(head->_next);
230   std::vector<Node*> to_update(head->_next);
231
232   Node* x = head;
233
234   int l = actualLevel;
235   while ( l>=0 ){
236     if (x->_next[l] == nullptr || key <= x->_next[l]->key){
237       to_update[l] = x;
238       --l;
239     }
240     else {
241       x = x->_next[l];
242     }
243   }
244   x = x->_next[0];
245
246   if ( x == nullptr) {
247     //we did something wrong
248     std::cout << "Searching in a nullptr :D" << std::endl;
249     return false;
250   }
251
252   if (x->key == key) {
253     return true;
254   } else {
255     return false;
256   }
257   return true;
```

```
258 }
259
260 int skipList::number_pointers() {
261   Node* list = head;
262   //std::cout << "Counting pointers" << std::endl;
263
264   //Initialize to 1 because of the HEAD pointer needed to have
          referenced
265   // the skip list, even empty, we need it
266   int total_pointers = 1;
267   while (list->_next[0] != nullptr) {
268     total_pointers += getLevel(list->_next);
269
270     list = list->_next[0];
271   }
272   return total_pointers;
273
274 }
275
276 long long skipList::total_search_cost(){
277   return this->_totalSearchCost;
278 }
279
280 //PRIVATE
281 int skipList::randomLevel() {
282   //std::cout << "randomLevel generator" << std::endl;
283   //int aux = 1;
284   //srand ((unsigned)time(NULL));
285   //bool cond = ((double)std::rand() / RAND_MAX) < this->prob;
286   ////while (cond || (aux < maxLevel - 1) ) {
287     ////++aux;
288     ////cond = ((double) std::rand() / RAND_MAX) < this->prob;
289   ////}
290   //while ((((double)std::rand() / RAND_MAX)) < prob && std::abs
          (aux) < maxLevel)
291     //++aux;
292
293   int aux = 1;
294   std::random_device rd;
295   std::default_random_engine rng(rd());
296   std::uniform_real_distribution<float> uniform_dist(0,1);
297   float random = uniform_dist(rng);
298   bool cond = random < this->prob;
299   while ( cond && std::abs(aux) < maxLevel ) {
300     random = uniform_dist(rng);
301     cond = random < this->prob;
302     ++aux;
303   }
304   //std::cout << "RANDOM LEVEL: " << aux << std::endl;
305   return aux;
306 }
```

Listing 4: Code from file skipList.cpp

C main.cpp (launch experiments and collect data)

```
#include "skipList.hpp"
#include <stdio.h>

#define REPETIONS 50

void benchmark(int size) {

  std::cout << "0";
  for (double p = 0.1; p <= 1; p+=0.1) {
    std::cout << ":" << p;
  }
  std::cout << std::endl;

  float  n_pointers[11];
  float  n_cost[11];
  for (int i = 0; i < 11; i++){
    n_pointers[i] = 0;
    n_cost[i] = 0;
  }

  int pos = 0;
  for (int j = 0; j < REPETIONS; j++) {
    pos = 0;
    for (double p = 0; p <= 1; p+=0.1) {
      skipList sl(p);

      //fill skip list
      for (unsigned int i = 0; i < size; i++) {
        std::stringstream ss;
        ss << i;
        sl.insert(i,ss.str());
      }

      //search in find list for cost
      for (unsigned int i = size - 1; i > 0; i--) {
        std::string aux = sl.find(i);
      }
      n_cost[pos] += sl.total_search_cost();
      /*std::cout << sl.total_search_cost();*/
      /*if (p+0.1 < 1.0 )*/
        /*std::cout << ":";*/
      /*else std::cout << std::endl;*/

      /*n_pointers.push_back(sl.number_pointers());*/
      n_pointers[pos] += sl.number_pointers();
      ++pos;
    }
  }
  for(int i = 0; i < 11; ++i) {
    std::cout << n_cost[i]/REPETIONS;
    if (i+1 < 11)
      std::cout << ":";
```

```
53      else
54        std::cout << std::endl;
55
56    }
57    for(int i = 0; i < 11; ++i) {
58      std::cout << n_pointers[i]/REPETIONS;
59      if (i+1 < 11)
60        std::cout << ":";
61      else
62        std::cout << std::endl;
63
64    }
65
66
67  }
68
69  void test() {
70    int size = M;
71    double p = 0.75;
72
73    skipList sl(p);
74
75    //fill skip list
76    for (unsigned int i = 0; i < size; i++) {
77      std::stringstream ss;
78      ss << i;
79      sl.insert(i,ss.str());
80    }
81
82    //search in find list
83    for (unsigned int i = size - 1; i > 0; i--) {
84      std::string aux = sl.find(i);
85      /*std::cout << "Our string stored in key " << i << "is: " <<
            aux << std::endl;*/
86    }
87    std::cout << "========REPORT==========" << std::endl;
88    std::cout << "N: " << size << std::endl;
89    std::cout << "p: " << p << std::endl;
90    std::cout << "Total search cost: " << sl.total_search_cost()
          << std::endl;
91    std::cout << "Total pointers: " << sl.number_pointers() << std
          ::endl;
92
93  }
94
95
96  int main(int argc, char *argv[]) {
97    std::cout << "Testing Skip Lists..." << std::endl;
98
99    /*int debug = atoi(argv[1]);*/
100   int debug = 0;
101   if (!debug)
102     for (int i = M; i <= M*10; i +=M){
103       std::cout << "N: " << i << std::endl;
104       benchmark(i);
```

```
105
106       }
107     else
108       test();
109 }
```

Listing 5: Code from file main.cpp