ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

СПИСОК С ПРОПУСКАМИ (SKIP LIST)

Выполнил студент Рябов Кирилл Александрович
Ф.И.О.

Группы ИС-042

Работу принял _____ ст. преп. Кафедры ВС Д. М. Берлизов
подпись

Защищена _____ Оценка _____

Новосибирск – 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Список с пропусками (skip list) — это структура данных для реализации словаря, основанная на нескольких параллельных отсортированных связных списках. Такие списки балансируются с использованием генератора случайных чисел. Несмотря на то, что у списков с пропусками плохая производительность в худшем случае, не существует такой последовательности операций, при которой бы это происходило постоянно. Очень маловероятно, что эта структура данных значительно разбалансируется (например, для словаря размером более 250 элементов вероятность того, что поиск займёт в три раза больше ожидаемого времени, меньше одной миллионной).

Балансировать структуру данных вероятностно проще, чем явно обеспечивать баланс. Для многих задач списки пропуска это более естественное представление данных по сравнению с деревьями. Алгоритмы получаются более простыми для реализации и, на практике, более быстрыми по сравнению со сбалансированными деревьями. Кроме того, списки с пропусками очень эффективно используют память. Они могут быть реализованы так, чтобы на один элемент приходился в среднем примерно 1.33 указатель (или даже меньше) и не требуют хранения для каждого элемента дополнительной информации о балансе или приоритете.

ОПИСАНИЕ СТРУКТУРЫ

Структура списка с пропусками имеет следующие свойства:

1) Каждый элемент в списке с пропусками представлен узлом.
2) У каждого узла есть уровень, который соответствует количеству указателей на следующие уровни.
3) Уровнем списка называется максимальный уровень узла в этом списке (если список пуст, то уровень равен 1).
4) Количество уровней ограничено заранее выбранной константой *MaxLevel*.
5) *i*-ый указатель узла указывает на следующий узел, находящийся на уровне *i* или выше.
6) При вставке нового элемента в список, узел вставляется на уровень со случайным номером.
7) Уровни со случайными номерами генерируются по шаблону. Например: 50% — уровень 1, 25% — уровень 2, 12.5% — уровень 3 и т.д.
8) Заголовок списка (header) содержит указатели на уровни с 1 по *MaxLevel*. Если элементов такого уровня ещё нет, то значение указателя — специальный элемент *NIL*, ключ которого больше любого ключа, который может когда-либо появиться в списке.

Список с пропусками является связным списоком, в котором каждый узел содержит различное количество связей, причём *i*-ые связи в узлах реализуют односвязные списки, пропускающие узлы, содержащие менее чем *i* связей.

Такая структура данных позволяет реализовать бинарный поиск для связных списков (т.е. поиск выполняется быстрее, чем тривиальный проход по списку за линейное время).

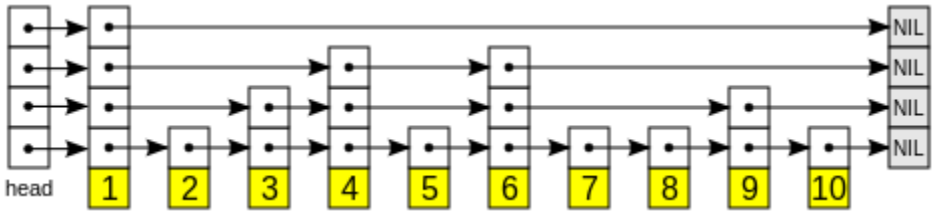

Рисунок 1 – Пример списка с пропусками

Инициализация списка с пропусками проходит следующим образом: создается элемент *NIL*, ключ которого больше любого ключа, который может когда-либо появиться в списке. Элемент *NIL* будет завершать все списки с пропусками. Уровень списка становится равен 1, а все указатели из заголовка ссылаются на *NIL*.

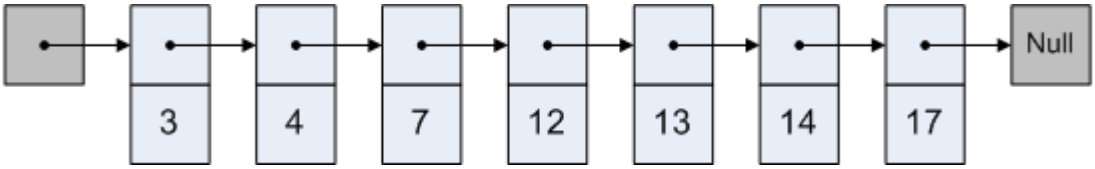

Рисунок 2 – Односвязный отсортированный список

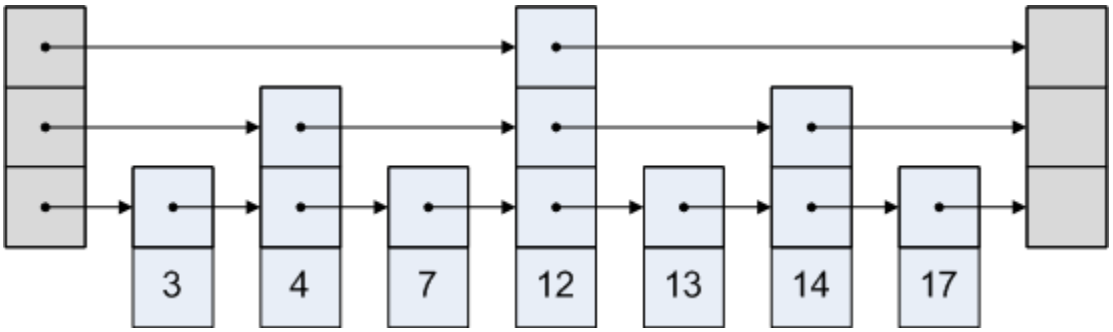

Рисунок 3 – Получившийся список с пропусками

ОСНОВНЫЕ ОПЕРАЦИИ НАД СТРУКТУРОЙ

Опишем алгоритмы для поиска, вставки и удаления элементов в словаре, реализованном на списках с пропусками:

1) Операция ***поиска*** (*search*) возвращает значение для заданного ключа или сигнализирует о том, что ключ не найден.

Начиная с указателя наивысшего уровня, двигаемся вперед по указателям до тех пор, пока они ссылаются на элемент, не превосходящий искомый. Затем спускаемся на один уровень ниже и снова двигаемся по тому же правилу. Если мы достигли уровня 1 и не можем идти дальше, то мы находимся как раз перед элементом, который ищем (если он там есть).

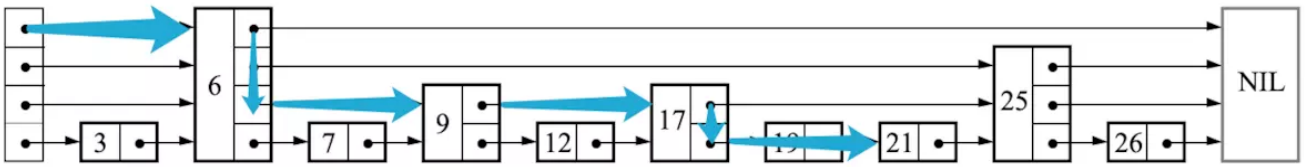

Рисунок 4 – Поиск элемента 21 в списке с пропусками

```
1    Search(list, searchKey)
2        x := list->header
3        # инвариант цикла: x->key < searchKey
4        for i := list->level downto 1 do
5            while x->forward[i]->key < searchKey do
6                x := x->forward[i]
7        # x->key < searchKey <= x->forward[1]->key
8        x := x->forward[1]
9        if x->key = searchKey then return x->value
10       else return failure
```

2) Для операций ***вставки*** (*insert*) или ***удаления*** (*delete*) узла применяем алгоритм ***поиска*** для нахождения всех элементов перед вставляемым (или удаляемым), затем обновляем соответствующие указатели.

Для запоминания элементов перед вставляемым (или удаляемым) используется массив *update*. Элемент *update*[*i*] — это указатель на самый правый узел, уровня *i* или выше, из числа находящихся слева от места обновления.

Если случайно выбранный уровень вставляемого узла оказался больше, чем уровень всего списка (т.е. если узлов с таким уровнем ещё не было), увеличиваем уровень списка и инициализируем соответствующие элементы массива *update* указателями на заголовок. После каждого удаления проверяем, удалили ли мы узел с максимальным уровнем и, если это так, уменьшаем уровень списка.

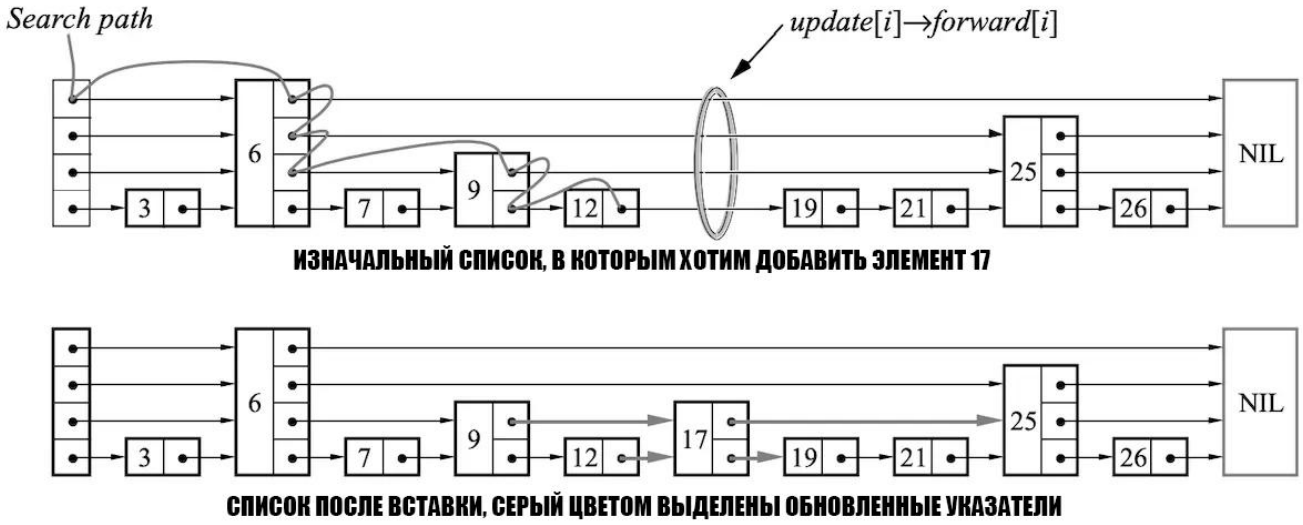

Рисунок 5 – Добавление элемента 17 в список с пропусками

```
1    Insert(list, searchKey, newValue)
2        local update[1..MaxLevel]
3        x := list->header
4        for i := list->level downto 1 do
5            while x->forward[i]->key < searchKey do
6                x := x->forward[i]
7            # x->key < searchKey <= x->forward[i]->key
8            update[i] := x
9        x := x->forward[1]
10       if x->key = searchKey then x->value := newValue
11       else
12           lvl := randomLevel()
13           if lvl > list->level then
14               for i := list->level + 1 to lvl do
15                   update[i] := list->header
16               list->level := lvl
17           x := makeNode(lvl, searchKey, value)
18           for i := 1 to level do
19               x->forward[i] := update[i]->forward[i]
20               update[i]->forward[i] := x
21
22   Delete(list, searchKey)
23       local update[1..MaxLevel]
24       x := list->header
25           for i := list->level downto 1 do
26               while x->forward[i]->key < searchKey do
27                   x := x->forward[i]
28           update[i] := x
29       x := x->forward[1]
30       if x->key = searchKey then
31           for i := 1 to list->level do
32               if update[i]->forward[i] != x then break
33               update[i]->forward[i] := x->forward[i]
34           free(x)
35           while list->level > 1 and list->header->forward[list->level] =
36   NIL do
37               list->level := list->level - 1
```

3) Номер уровня для новой вершины генерируется случайно. Заметим, что количество элементов в списке не участвует в генерации.

```
1    randomLevel()
2        lvl := 1
3        # random() возвращает случайное число в полуинтервале [0...1)
4        while random() < p and lvl < MaxLevel do
5            lvl := lvl + 1
6        return lvl
```

ЭФФЕКТИВНОСТЬ АЛГОРИТМОВ

Перед тем как перейти к разбору эффективности основных операций списка с пропусками, введем функцию от n, которая будет отображать ожидаемое число уровней. Пусть L — некий уровень, на котором мы ожидаем $1 / p$ узлов, где p — доля узлов уровня i, содержащих указатели на узлы уровня $i + 1$. Это случится при $L(n) = log_{1/p} n$.

Так как ожидаемое число уровней равно $L(n) = log_{1/p} n$, то эффективней всего выбрать MaxLevel = $L(N)$, где N — максимальное число элементов в списке с пропусками. Например, если $p = 0.5$, то MaxLevel = 16 подойдет для списков, содержащих не менее 2^{16} элементов.

В операциях поиска, вставки и удаления больше всего времени уходит на поиск подходящего элемента. Для вставки и удаления дополнительно нужно время, пропорциональное уровню вставляемого или удаляемого узла. Время поиска элемента пропорционально количеству пройденных в процессе поиска узлов, которое, в свою очередь, зависит от распределения их уровней.

Структура списка с пропусками определяется только количеством элементов в этом списке и значениями генератора случайных чисел. Последовательность операций, с помощью которых получен список, не важна. Мы предполагаем, что у пользователя нет доступа к уровням узлов, иначе он может сделать так, чтобы алгоритм работал за наихудшее время, удалив все узлы, уровень которых не равен 1.

Для последовательных операций на одной структуре данных времена их выполнения не являются независимыми случайными величинами. Две

последовательные операции поиска одного и того же элемента займут одно и то же время.

Рассмотрим пройденный при поиске путь с конца, т.е. будем двигаться вверх и влево. Хотя уровни узлов в списке известны и зафиксированы на момент поиска, мы предположим, что уровень узла определяется только когда мы встретили его при движении с конца.

В любой заданной точке пути мы находимся в следующей ситуации:

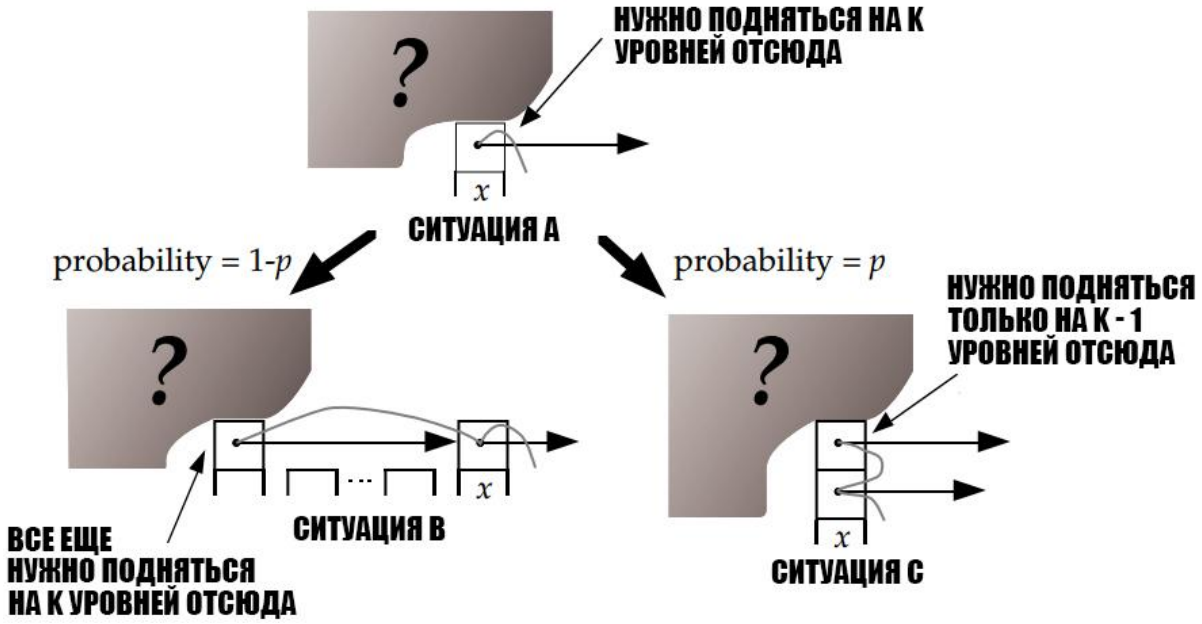

Рисунок 6 – Возможные ситуации при обратном обходе пути поиска

Мы смотрим на i-ый указатель узла x и не знаем об уровнях узлов слева от x. Также мы не знаем точного уровня x, но он должен быть как минимум i. Предположим, что x — это не заголовок списка (это эквивалентно предположению, что список расширяется влево бесконечно). Если уровень x равен i, то мы находимся в ситуации B. Если уровень x больше i, то мы в ситуации C. Вероятность того, что мы находимся в ситуации C, равна p. Каждый раз, когда это

происходит, мы поднимаемся вверх на один уровень. Пусть $C(k)$ — ожидаемая длина обратного пути поиска, при котором мы двигались вверх k раз:

$C(0) = 0$

$C(k) = (1 - p)(\text{длина пути в ситуации } B) + p(\text{длина пути в ситуации } C)$

Упрощаем:

$C(k) = (1 - p)(1 + C(k)) + p(1 + C(k - 1))$

$C(k) = 1 + C(k) - pC(k) + pC(k - 1)$

$C(k) = 1 / p + C(k - 1) \Rightarrow C(k) = k / p$

Наше предположение о том, что список бесконечный — пессимистично. Когда мы доходим до самого левого элемента, мы просто двигаемся все время вверх, не двигаясь влево. Это дает нам верхнюю границу $(L(n) - 1) / p$ ожидаемой длины пути от узла с уровнем 1 до узла с уровнем $L(n)$ в списке из n элементов.

Мы используем эти рассуждения, чтобы добраться до узла уровня $L(n)$, но для остальной части пути используются другие рассуждения. Количество оставшихся ходов влево ограничено числом узлов, имеющих уровень $L(n)$, или выше во всем списке. Наиболее вероятное число таких узлов $1 / p$.

Также двигаемся вверх от уровня $L(n)$ до максимального уровня в списке. Вероятность того, что максимальный уровень списка больше k, равна $1 - (1 - p^k)^n$, что не больше, чем np^k. Мы можем вычислить, что ожидаемый максимальный уровень не более $L(n) + 1 / (1 - p)$. Собирая все вместе, получим, что ожидаемая длина пути поиска для списка из n элементов не превышает $L(n) / p + 1 / (1 - p)$, что составляет $O(\log n)$.

Другими словами, средняя временная сложность как операции поиска, так и операций добавления и удаления составляют $O(\log\ n)$, что говорит нам о возможности списка с пропусками стать более простой с точки зрения реализации альтернативой сбалансированным деревьям. Также стоит заметить, что операция поиска по временной эффективности сравнима с бинарным поиском. Фактически, благодаря данной структуре реализуется «бинарный поиск» на основе односвязных списков.

Операция	Средний случай (average case)	Худший случай (worst case)
Add(*key*, *value*)	*O*(log*n*)	*O*(*n*)
Lookup(*key*)	*O*(log*n*)	*O*(*n*)
Remove(*key*)	*O*(log*n*)	*O*(*n*)

Рисунок 7 – Временная сложность основных операций списка с пропусками

СРАВНЕНИЕ СТРУКТУР SKIP LIST И AVL TREE

Список с пропусками предлагает использовать балансировку вероятностно с помощью генератора случайных чисел, что на практике реализуется намного проще, чем обеспечивание баланса явно как в сбалансированных деревьях. Например, для того, чтобы сбалансировать структуру данных АВЛ-дерева, необходимо реализовывать четыре операции поворота для восстановления соответственно свойств и баланса данной структуры.

Другими словами, такие операции как вставка и удаление в АВЛ-дереве могут привести к корректировке поддерева, что является сложным по логике. В это же время, список с пропусками в операциях вставки и удаления предлагает только изменить указатели соседних узлов, а это достаточно быстрая и к тому же простая операция.

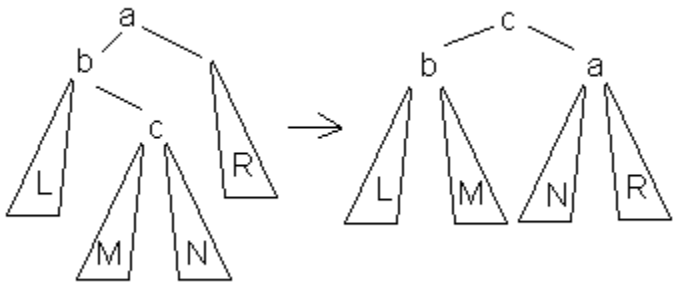

Рисунок 8 – Пример балансировки АВЛ-дерева. Большое правое вращение

С точки зрения использования памяти список с пропусками более гибок, чем АВЛ-дерево. Каждый узел сбалансированного дерева содержит два указателя, указывающих на левое и правое поддеревья соотвественно, в то время как среднее

количество указателей, которое содержется в каждом узле списка с пропусками, равно $1 / (1 - p)$, в зависимости от доли узлов уровня i, содержащих указатели на узлы уровня $i + 1$ (т.е. параметра p). Например, если взять $p = 0.25$, то каждый узел будет содержать в среднем 1.33 указателей, что более выгодно, чем в том же АВЛ-дереве.

Тем не менее, АВЛ-дерево оправдывает свою сложность в реализации отличной временной эффективностью операций как в худшем, так и в среднем случаях. В то же время, список с пропусками предлагает более простую реализацию и отличную временную эффективность операций, но лишь только в среднем случае.

Попробуем на практике сравнить скорость выполнения операций поиска и удаления элементов из списка с пропусками и АВЛ-дерева.

ЭКСПЕРИМЕНТАЛЬНОЕ СРАВНЕНИЕ СЛОЖНОСТИ ОПЕРАЦИЙ SKIP LIST И AVL TREE

10 тысяч элементов:

Операция	Список с пропусками	АВЛ-дерево
Lookup, с·10^6	130.89	118.058
Delete, с·10^6	134.91	194.362

50 тысяч элементов:

Операция	Список с пропусками	АВЛ-дерево
Lookup, с·10^6	360.93	325.545
Delete, с·10^6	536.213	621.511

100 тысяч элементов:

Операция	Список с пропусками	АВЛ-дерево
Lookup, с·10^6	457.12	412.304
Delete, с·10^6	621.1	684.268

Из полученных данных можно сказать, что операция поиска элемента у АВЛ-дерева все же выполняется быстрее, чем у списка с пропусками, хотя разница

между их скоростями достаточно мала. Что касается операции удаления, то тут ситуация обратная. Удаление в списке с пропусками на практике проходит быстрее, чем в АВЛ-дереве. Если учесть еще тот факт, что реализовать структуру списка с пропусками в разы легче, чем структуру АВЛ-дерева, то можно было бы сказать, что на практике лучше всего отдавать предпочтение именно спискам с пропусками.

Тем не менее, не стоит забывать, что сложности обеих структур данных одинаковы лишь в ***среднем случае***. Если рассматривать ***худший случай***, то сложность операций над списком с пропусками станет линейной, в то время как у АВЛ-дерева она все еще будет логарифмическая. Но опять же, не существует такой последовательности операций, при которой бы производительность списка с пропусками была равна ***худшему случаю*** на постоянной основе, что позволяет данной структуре быть на равне с тем же АВЛ-деревом. Поэтому список с пропусками является отличной альтернативой сбалансированных деревьев, а не полной их заменой.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработана и исследована структура данных список с пропусками, было проведено сравнение со структурой сбалансированного дерева, а именно с АВЛ-деревом.

Экспериментально подтвердили, что основные операции над список с пропусками в среднем занимают такое же время, что и операции над АВЛ-деревом. Таким образом, подтвердилось то, что список с пропусками может выступать отличной альтернативой сбалансированным деревьям в большинстве случаев за счет своей простоты в реализации. А базовые операции над структурой списка с пропусками в среднем случае примерно такие же быстрые как и операции над сбалансированными деревьями вроде АВЛ-дерева и выполняются за $O(\log n)$.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.:ДиаСофт, 2001. — 688с. (С. 555)
2. *Pugh W.* A Skip List Cookbook // cg.scs.carleton.ca/teaching/5408/refs/p90b.pdf
3. Списки с пропусками: вероятностная альтернатива сбалансированным деревьям — habr.com/ru/post/230413/
4. Список с пропусками — neerc.ifmo.ru/wiki/index.php?title=skip_list
5. Skip list — en.wikipedia.org/wiki/Skip_list

ПРИЛОЖЕНИЕ

Исходный код с комментариями;

skiplist.c

```
1    #include "skiplist.h"
2    #include <stdlib.h>
3    #include <limits.h>
4    #include <time.h>
5
6    #define SKIPLIST_MAX_LEVEL 6
7
8    SkipList *skiplist_init(SkipList *list)
9    {
10       int i;
11       Node *header = malloc(sizeof(Node));
12       list->header = header;
13       header->key = INT_MAX;
14       header->forward = malloc(sizeof(Node *) * (SKIPLIST_MAX_LEVEL +
15   1));
16       for (i = 0; i <= SKIPLIST_MAX_LEVEL; i++)
17       {
18           header->forward[i] = list->header;
19       }
20
21       list->level = 1;
22       list->size = 0;
23
24       return list;
25   }
26
27   static int rand_level()
28   {
29       srand(time(0));
30       int level = 1;
31       float r = rand() / RAND_MAX;
32       while ((r < 0.5) && (level < SKIPLIST_MAX_LEVEL))
33       {
34           ++level;
35       }
36       return level;
37   }
38
39   int skiplist_insert(SkipList *list, int key, int value)
40   {
41       Node *update[SKIPLIST_MAX_LEVEL + 1];
42       Node *x = list->header;
43       int i, level;
44       for (i = list->level; i >= 1; i--)
```

```
45          {
46              while (x->forward[i]->key < key)
47              {
48                  x = x->forward[i];
49              }
50              update[i] = x;
51          }
52          x = x->forward[1];
53
54          if (key == x->key)
55          {
56              x->value = value;
57              return 0;
58          }
59          else
60          {
61              level = rand_level();
62              if (level > list->level)
63              {
64                  for (i = list->level + 1; i <= level; i++)
65                  {
66                      update[i] = list->header;
67                  }
68                  list->level = level;
69              }
70
71              x = malloc(sizeof(Node));
72              x->key = key;
73              x->value = value;
74              x->forward = malloc(sizeof(Node *) * (level + 1));
75              for (i = 1; i <= level; i++)
76              {
77                  x->forward[i] = update[i]->forward[i];
78                  update[i]->forward[i] = x;
79              }
80          }
81          return 0;
82      }
83
84      Node *skiplist_search(SkipList *list, int key)
85      {
86          Node *x = list->header;
87          int i;
88          for (i = list->level; i >= 1; i--)
89          {
90              while (x->forward[i]->key < key)
91              {
92                  x = x->forward[i];
93              }
94          }
95          if (x->forward[1]->key == key)
96          {
97              return x->forward[1];
```

```
98          }
99          else
100         {
101             return NULL;
102         }
103         return NULL;
104     }
105
106     static void skiplist_node_free(Node *x)
107     {
108         if (x)
109         {
110             free(x->forward);
111             free(x);
112         }
113     }
114
115     int skiplist_delete(SkipList *list, int key)
116     {
117         int i;
118         Node *update[SKIPLIST_MAX_LEVEL + 1];
119         Node *x = list->header;
120         for (i = list->level; i >= 1; i--)
121         {
122             while (x->forward[i]->key < key)
123             {
124                 x = x->forward[i];
125             }
126             update[i] = x;
127         }
128
129         x = x->forward[1];
130         if (x->key == key)
131         {
132             for (i = 1; i <= list->level; i++)
133             {
134                 if (update[i]->forward[i] != x)
135                 {
136                     break;
137                 }
138                 update[i]->forward[1] = x->forward[i];
139             }
140             skiplist_node_free(x);
141
142             while (list->level > 1 && list->header->forward[list->level] ==
143     list->header)
144             {
145                 list->level--;
146             }
147             return 0;
148         }
149         return 1;
150     }
```

Исходный код с комментариями;

skiplist.h

```
1    #pragma once
2
3    typedef struct Node
4    {
5        int key;
6        int value;
7        struct Node **forward;
8    } Node;
9
10   typedef struct SkipList
11   {
12       int level;
13       struct Node *header;
14   } SkipList;
15
16   SkipList *skiplist_init(SkipList *list);
17   int skiplist_insert(SkipList *list, int key, int value);
18   Node *skiplist_search(SkipList *list, int key);
19   int skiplist_delete(SkipList *list, int key);
```

Исходный код с комментариями;

main.c

```
1    #include "skiplist.h"
2    #include <stdio.h>
3
4    const int NUM = 10000;
5
6    int main()
7    {
8        srand(time(0));
9        SkipList* list = skiplist_init();
10
11       for (int i = 0; i < NUM; ++i)
12       {
13           skiplist_insert(list, i, i);
14       }
15       for (int i = 0; i < 10; ++i) {
16           Node* founder = skiplist_search(list, rand() % NUM);
17           printf("key: %d\n", founder->key);
18       }
19       skiplist_delete(list, rand() % NUM);
20
21       return 0;
22   }
```