Дерево ван Эмде Боаса

Голобородько Димитрий

25 декабря 2022 г.

1 Введение

1.1 Суть и назначение

Дерево ван Эмде Боаса — поисковая структура данных, представляющая собой дерево поиска, позволяющее хранить уникальные целые неотрицательные числа в интервале $[0; 2^k)$ и осуществлять над ними основные операции поисковой структуры данных за $O(logk)$ при затратах памяти $\Theta(2^k)$.

1.2 История

Структура разработана осенью 1974 года Питером ван Эмде Боасом во время его трёхмесячного пост-докторского резидентства в Корнеллском университете и представлена в 1975 году. Автор структуры отмечает, что на его ранние публикации оказывал влияние культурный климат в алгоритмике середины 1970-х годов, на фоне которого структура была разработана для машины указателей (pointer machine). Причина в том, что рекурсивный подход требует адресных вычислений. Данные операции не допускались в модели машины с произвольным доступом (RAM), которая была стандартной моделью в развивающейся области исследований проектирования и анализа алгоритмов в 1974 году. Недостатком этого подхода является то, что он приводит к довольно сложным алгоритмам, которые и сегодня трудно корректно реализовать. [5]

1.3 Применение

Данное дерево может применяться в задачах, где требуется поисковая структура данных для множества уникальных целых неотрицательных чисел, верхняя граница которого известна заранее, либо ограничена типом данных, при этом количество чисел достаточно велико, чтобы оправдать накладные расходы памяти. [8]

В сети встречаются упоминания применения дерева ван Эмде Боаса в алгоритмах на графах и вычислительной геометрии, а также в сетевых маршрутизаторах. [5]

Структура может применяться для сортировки последовательности из n чисел с асимптотикой $O(n \cdot logk)$, оптимизации алгоритма Дейкстры до асимптотики $O(E \cdot logk)$ где E - количество рёбер между вершинами графа. [10]

2 Описание

2.1 Поддерживаемые операции

- member - проверка наличия числа в структуре
- insert - вставка числа
- remove - удаление числа
- successor - поиск следующего по возрастанию числа за данным
- predecessor - поиск следующего по убыванию числа за данным
- min, max - посик минимального, максимального хранимых чисел

2.2 Вспомогательные функции

Для поддержки тех размерностей поддеревьев, из значений которых целый квадратный корень не извлекается, введены следующие вспомогательные функции:

- lower_sqrt(x) - "нижний квадратный корень", равен $2^{\lfloor \log_2(x)/2 \rfloor}$
- upper_sqrt(x) - "верхний квадратный корень", равен $2^{\lceil \log_2(x)/2 \rceil}$

При этом $x = upper\_sqrt(x) * lower\_sqrt(x)$, а когда $\sqrt{x}$ - целое, верно $\sqrt{x} = upper\_sqrt(x) = lower\_sqrt(x)$

- low(u, x) - младшие $2^{\lfloor \log_2(u)/2 \rfloor}$ бит номера элемента x, вычисляются как $\lfloor x/lower\_sqrt(u) \rfloor$ где u - размерность текущего поддерева
- high(u, x) - старшие $2^{\lceil \log_2(u)/2 \rceil}$ бит номера x, вычисляются как $x \mod lower\_sqrt()$ где u - размерность текущего поддерева
- index(u, x, y) - совмещает номер элемента из старших бит x и младших бит y, вычисляется как $x * lower\_sqrt(u) + y$, где u - размерность текущего поддерева

[2]

2.3 Структура данных

Дерево ван Эмде Боаса является рекурсивной структурой, каждый узел которого является корнем поддерева и содержит в себе:

- u - размерность поддерева, элементов
- min, max - хранят минимальный и максимальный элемент поддерева
- summary - указатель на справочную структуру, которая также является деревом ван Эмде Боаса с размерностью $upper\_sqrt(u)$ элементов, хранит номера непустых поддеревьев
- cluster - массив из $upper\_sqrt(u)$ указателей на поддеревья с размерностью $upper\_sqrt(u)$

Свойства:

- Минимум и максимум дерева хранятся в корневом элементе и извлекаются за $O(1)$.
- Значения минимума/максимума хранятся только в соответствующих полях корней поддеревьев и не упоминаются глубже по дереву, это позволяет определять за $O(1)$:
  - Отсутствие элементов в (под)дереве
  - Наличие ровно одного элемента (min==max)
  - Наличие двух и более элементов
- Вставка в пустое поддерево и удаление последнего элемента выполняются за $O(1)$

Совокупность данных свойств позволяет избежать необходимости более одного рекурсивного вызова за $O(logk)$ на операцию, выполнять часть операций за $O(1)$.

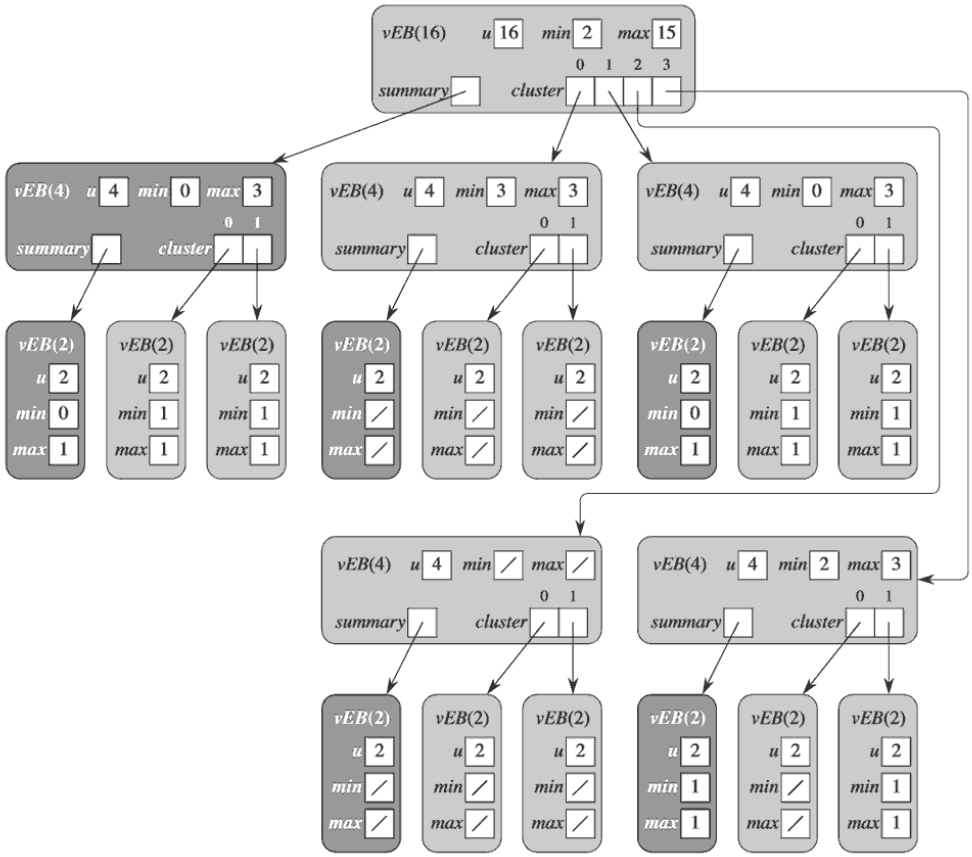

Рис. 1: Дерево ван Эмде Боаса размерностью 16, содержащее числа {2, 3, 4, 5, 7, 14, 15}

2.4 Входные данные и ограничения

Оригинальная структура, представленная Питером ван Эмде Боасом требует корректности входных данных, а именно, не гарантирует корректного удаления не существующего элемента, добавления уже существующего элемента. Размерность дерева u должна быть степенью числа 2. Все операции на дереве размерностью u подразумевают работу только с числами из интервала $[0; u)$. [2]

3 Формальная постановка задачи

Исследовать дерево ван Эмде Боаса и реализовать в виде набора функций для создания и удаления рекурсивной структуры дерева, исполнения набора поддерживаемых операций (member, insert, remove, successor, predecessor, min, max). Протестировать производительность поисковых операций (member, successor, predecessor).

4 Реализация

В реализации в рамках данного доклада для значения элемента выбран тип unsigned int, что ограничивает максимальную размерность дерева значением в $2^{32} - 1$ элементов, при этом последний элемент unsigned int $(2^{32} - 1)$ зарезервирован как флаг, обозначающий пустой элемент NIL в полях min и max.

4.1 Структура узла

```
1  typedef struct veb_node veb;
2  struct veb_node
3  {
4      unsigned int u;
5      unsigned int min;
```

```
6       unsigned int max;
7       veb *summary;
8       veb **cluster;
9   };
```

4.2 Реализация вспомогательных функций

```
1   const unsigned int NIL = -1;
2
3   unsigned int veb_upper_sqrt(unsigned int u) {
4       return pow(2, ceil(log2(u)/2));
5   }
6
7   unsigned int veb_lower_sqrt(unsigned int u) {
8       return pow(2, floor(log2(u)/2));
9   }
10
11  unsigned int veb_high(unsigned int u,  unsigned int x) {
12      return floor(x/veb_lower_sqrt(u));
13  }
14
15  unsigned int veb_low(unsigned int u, unsigned int x) {
16      return x % veb_lower_sqrt(u);
17  }
18
19  unsigned int veb_index(unsigned int u, unsigned int x, unsigned int y)
        {
20      return x*veb_lower_sqrt(u)+y;
21  }
```

4.3 Min, max

Значения минимума и максимума дерева ван Эмде Боаса, хранятся в полях структуры корня, поэтому могут быть получены за константное время.

```
1   unsigned int veb_tree_minimum(veb* v) {
2       return v->min;
3   }
4   unsigned int veb_tree_maximum(veb* v) {
5       return v->max;
6   }
```

4.4 Member

```
1   unsigned int veb_tree_member(veb* v, unsigned int x){
2       return
3           x == v->min || x == v->max ? 1 :
4           v->u == 2 ? 0 :
5           veb_tree_member(v->cluster[veb_high(v->u,x)], veb_low(v->u, x))
                ;
6   }
```

Строка 3 проверяет случай, в котором искомый элемент x является максимальным или минимальным в дереве, и возвращает истину, если это так. Строка 4 проверяет случай, в котором поддерево размерностью $u = 2$ не имеет элементов, возвращает ложь если это так. В противном случае происходит рекурсивное углубление в поддерево с индексом high(u, x) и поиск числа $x = low(u, x)$ [2]

4.5 Successor, predecessor

```
1  unsigned int veb_tree_successor(veb* v, unsigned int x){
2      if (v->u == 2) {
3          if (x == 0 && v->max == 1) {
4              return 1;
5          } else {
6              return NIL;
7          }
8      } else if (v->min != NIL && x < v->min)
9      {
10         return v->min;
11     } else {
12         unsigned int max_low = veb_tree_maximum(v->cluster[veb_high(v->
               u,x)]);
13         if (max_low != NIL && veb_low(v->u, x) < max_low) {
14             unsigned int offset = veb_tree_successor(v->cluster[
                   veb_high(v->u, x)], veb_low(v->u,x));
15             return veb_index(v->u, veb_high(v->u, x), offset);
16         }
17         else {
18             unsigned int succ_cluster = veb_tree_successor(v->summary,
                   veb_high(v->u,x));
19             if (succ_cluster == NIL)
20             {
21                 return NIL;
22             } else {
23                 unsigned int offset = veb_tree_minimum(v->cluster[
                       succ_cluster]);
24                 return veb_index(v->u, succ_cluster, offset);
25             }
26         }
27     }
28 }
```

Строки 2-7 проверяют базовый случай, в котором искомый элемент является следующим за 0 и элемент 1 при этом находится в поддереве размерностью 2. Иначе, строка 8 проверяет если x строго меньше минимума дерева, и возвращает его если это так.

Иначе если исполнение доходит до строки 12, известно что x больше или равен минимальному значению дерева. В строке 12 происходит присваивание переменной max_low максимального элемента в кластере x. Если найден элемент больше x, то известно, что последующий элемент за x находится где-то в поддереве кластера x. Эта проверка выполняется строкой 13. Если это так, строка 14 определяет где именно в кластере находится элемент и строка 15 вычисляет и возвращает найденный элемент.

Выполнение доходит до строки 17 в том случае, если x больше или равен наибольшему элементу в кластере. В этом случае, строки 18-28 производят поиск в дргуих непустых кластерах. Переменной succ_cluster присваивается номер следующего непустого кластера, найденного с помощью справочной структуры. [2]

```
1  unsigned int veb_tree_predecessor(veb* v, unsigned int x){
2      if (v->u == 2) {
3          if (x == 1 && v->min == 0) {
4              return 0;
5          } else {
6              return NIL;
7          }
8      } else if (v->max != NIL && x > v->max)
9      {
10         return v->max;
11     } else {
12         unsigned int min_low = veb_tree_minimum(v->cluster[veb_high(v->
               u, x)]);
```

```
13          if (min_low != NIL && veb_low(v->u, x) > min_low) {
14              unsigned int offset = veb_tree_predecessor(v->cluster[
                    veb_high(v->u, x)], veb_low(v->u, x));
15              return veb_index(v->u, veb_high(v->u, x), offset);
16          } else {
17              unsigned int pred_cluster = veb_tree_predecessor(v->summary
                    , veb_high(v->u, x));
18              if (pred_cluster == NIL) {
19                  if (v->min != NIL && x > v->min){
20                      return v->min;
21                  } else {
22                      return NIL;
23                  }
24              } else {
25                  unsigned int offset = veb_tree_maximum(v->cluster[
                        pred_cluster]);
26                  return veb_index(v->u, pred_cluster, offset);
27              }
28          }
29      }
30  }
```

Операция predecessor противоположена successor за исключением дополнительного случая, обрабатываемого строками 19-20. Случай возникает, когда предыдущий элемент от x, если он находится в дереве, не находится в кластере x. Если предыдущий элемент x - минимум дерева, то последующего элемента в дереве нет (он в поле min). Строка 20 возвращает минимум в данном случае. [2]

4.6 Insert

```
1  void veb_empty_tree_insert(veb* v, unsigned int x) {
2      v->min = x;
3      v->max = x;
4  }
```

Так как элементы хранятся в полях min и max, вставка элемента в пустое дерево выполняется за константное время.

```
1  void veb_exchange(unsigned int* a, unsigned int* b) {
2      unsigned int temp = *a;
3      *a = *b;
4      *b = temp;
5  }
```

Вспомогательная функция обмена значений переменных между собой

```
1  void veb_tree_insert(veb* v, unsigned int x) {
2      if (v->min == NIL) {
3          veb_empty_tree_insert(v, x);
4      } else {
5          if (x < v->min) {
6              veb_exchange(&x,&(v->min));
7          }
8          if (v->u > 2) {
9              if (veb_tree_minimum(v->cluster[veb_high(v->u,x)]) == NIL)
                   {
10                 veb_tree_insert(v->summary, veb_high(v->u, x));
11                 veb_empty_tree_insert(v->cluster[veb_high(v->u, x)],
                       veb_low(v->u, x));
12             } else {
13                 veb_tree_insert(v->cluster[veb_high(v->u, x)], veb_low(
                       v->u, x));
```

```
14              }
15          }
16          if (x > v->max)
17          {
18              v->max = x;
19          }
20      }
21  }
```

Вставка элемента в дерево ван Эмде Боаса предполагает, что дерево ещё не содержит данного элемента.

Строка 2 проверяет если поддерево пустое, и вызывает функцию veb_empty_tree_insert() вставки в пустое дерево. Иначе строки 4-21 выполняют вставку в один из кластеров непустого дерева. Строка 5 проверяет если x является новым минимумом дерева, в данном случае строка 6 меняет минимум и вставляемый элемент местами и алгоритм продолжает вставлять уже предыдущий минимум. Строки 9-15 выполняются только если размерность дерева больше двух. Строка 9 определяет, если кластер, в который необходимо вставить элемент - пуст, и в данном случае строка 10 добавляет номер кластера в справочное поддерево, а строка 11 добавляет x в пустой кластер. Иначе, если кластер не пуст, строка 13 рекурсивно добавляет x в кластер. Строки 16-19 обновляют максимум дерева, в случае если вставляемый элемент больше максимума. [2]

4.7 Delete

Удаление элемента из дерева ван Эмде Боаса предполагает, что дерево содержит данный элемент.

```
1  void veb_tree_delete(veb* v, unsigned int x) {
2      if (v->min == v->max) {
3          v->min = NIL;
4          v->max = NIL;
5      } else if (v->u == 2) {
6          if (x == 0) {
7              v->min = 1;
8          } else {
9              v->min = 0;
10         }
11         v->max = v->min;
12     } else {
13         if (x == v->min) {
14             unsigned int first_cluster = veb_tree_minimum(v->summary);
15             x = veb_index(v->u, first_cluster, veb_tree_minimum(v->
                   cluster[first_cluster]));
16             v->min = x;
17         }
18         veb_tree_delete(v->cluster[veb_high(v->u,x)], veb_low(v->u, x))
               ;
19         if (veb_tree_minimum(v->cluster[veb_high(v->u, x)]) == NIL) {
20             veb_tree_delete(v->summary, veb_high(v->u, x));
21             if (x == v->max) {
22                 unsigned int summary_max = veb_tree_maximum(v->summary)
                       ;
23                 if (summary_max == NIL) {
24                     v->max = v->min;
25                 } else {
26                     v->max = veb_index(v->u, summary_max,
                           veb_tree_maximum(v->cluster[summary_max]));
27                 }
28             }
29         } else if (x == v->max) {
```

```
30                  v->max = veb_index(v->u, veb_high(v->u, x),
                        veb_tree_maximum(v->cluster[veb_high(v->u, x)]));
31              }
32          }
33      }
```

Если дерево содержит только один элемент, удалить его можно также просто и за константное время, как и вставить элемент в пустое дерево. Строка 2 проверяет этот случай и строки 3-4 заменяют максимум и минимум дерева на NIL. Иначе дерево содержит как минимум два элемента. Строка 5 проверяет это условие, и строки 6-11 устанавливают минимум и максимум равными оставшемуся элементу.

Иначе выполняются сроки 12-33 для дерева с размерностью 4 и более, с двумя и более элементами. В этом случае нужно удалять элемент из кластера. Элемент может не быть x но, если элемент равен минимуму, то, когда удаляется x, другой элемент в одном из кластеров становится новым минимумом и также должен быть удалён из кластера. Если проходит проверка на строке 13, строка 14 задаёт переменную first_cluster номером кластера, который содержит минимальный элемент, отличный от минимума, и строка 15 задаёт x равным значению наименьшего элемента в кластере. Строка 16 задаёт элемент новым минимумом поддерева, и так как этот элемент - новый x, он будет удалён.

Со строки 18 начинается удаление элемента x, после удаления кластер может оказаться пуст, что проверяет строка 19, и, если это так - удаляет номер кластера x из справочного поддерева на строке 20. После обновления справочного поддерева, необходимо обновить поле max. Строка 21 проверяет, если удаляется максимальный элемент дерева, и, если это так, задаётся переменная summary_max с наибольшим номером непустого кластера (полученным из справочного поддерева). Это и последующие обращения работают корректно, поскольку максимум справочного поддерева уже был скорректирован после рекурсивного запуска (если это требовалось). Если все кластеры пусты, то оставшийся элемент - min, строка 23 проверяет этот случай, и строка 24 соответственно обновляет max, иначе строка 26 устанавливает max равным максимальному элементу кластера с наибольшим номером.

Наконец, необходимо обработать случай, в котором кластер x не стал пуст после удаления x. В этом случае может потребоваться обновление максимума. Этот случай проверяется строкой 29, и строка 30 выполняет обновление. [2]

4.8 Создание и разрушение дерева

```
1   veb* create_veb(unsigned int u) {
2       veb* new_veb = (veb*)malloc(sizeof(veb));
3
4       if (!new_veb) return NULL;
5       new_veb->max = NIL;
6       new_veb->min = NIL;
7       new_veb->u = u;
8       if (u > 2) {
9           unsigned int upper_sqrt = veb_upper_sqrt(u);
10          unsigned int lower_sqrt = veb_lower_sqrt(u);
11          new_veb->summary = create_veb(upper_sqrt);
12          if (!(new_veb->summary)) return NULL;
13          new_veb->cluster = (veb**)malloc(upper_sqrt * sizeof(veb*));
14          if (!(new_veb->cluster)) return NULL;
15          for (int i = 0; i < upper_sqrt; i++){
16              new_veb->cluster[i] = create_veb(lower_sqrt);
17              if (!(new_veb->cluster[i])) return NULL;
18          }
19      }
20      return new_veb;
21  }
```

```
1   void destroy_veb(veb* veb) {
2       if (veb->u > 2) {
3           unsigned int upper_sqrt = veb_upper_sqrt(veb->u);
```

```
4           destroy_veb(veb->summary);
5           for (int i = 0; i < upper_sqrt; i++) {
6               destroy_veb(veb->cluster[i]);
7           }
8           free(veb->cluster);
9       }
10      free(veb);
11  }
```

5 Тестирование производительности

В рамках доклада протестирована производительность операций поиска (member, successor, predecessor). Тестирование производительности выполнено на трёх наборах случайных данных объёмом 1 тыс., 10 тыс. и 100 тыс. элементов на деревьях размерностью от 2^{17} до 2^{24}.

Демонстрируется зависимость среднего времени, затраченного на операцию от размерности дерева и количества хранимых элементов, а также время, затраченное на создание и разрушение дерева (выделение/высвобождение памяти). ...

5.1 Стратегия тестирования производительности

...

5.2 Результаты тестирования производительности

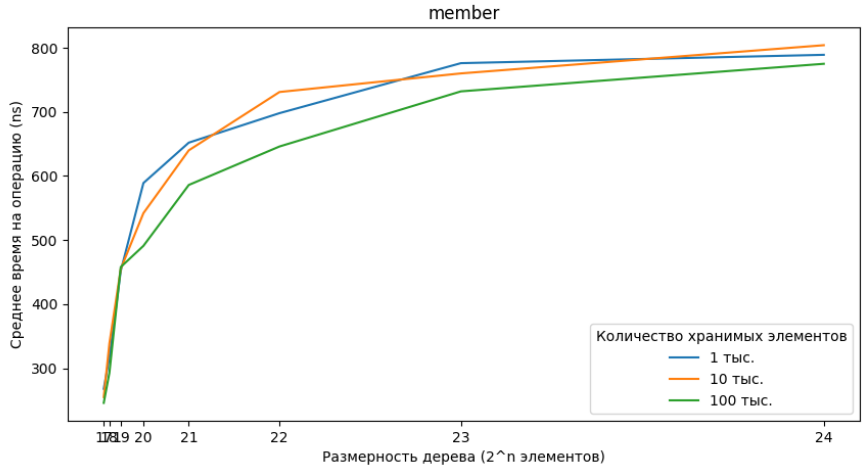

Рис. 2: Производительность операции member

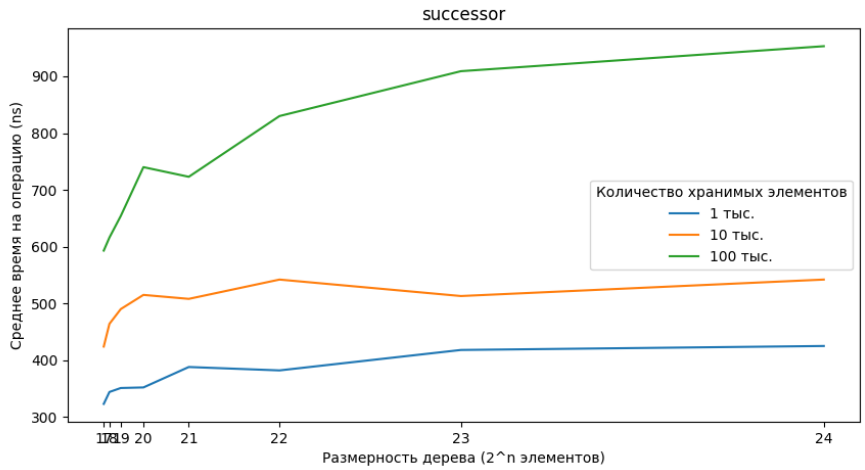

Рис. 3: Производительность операции successor

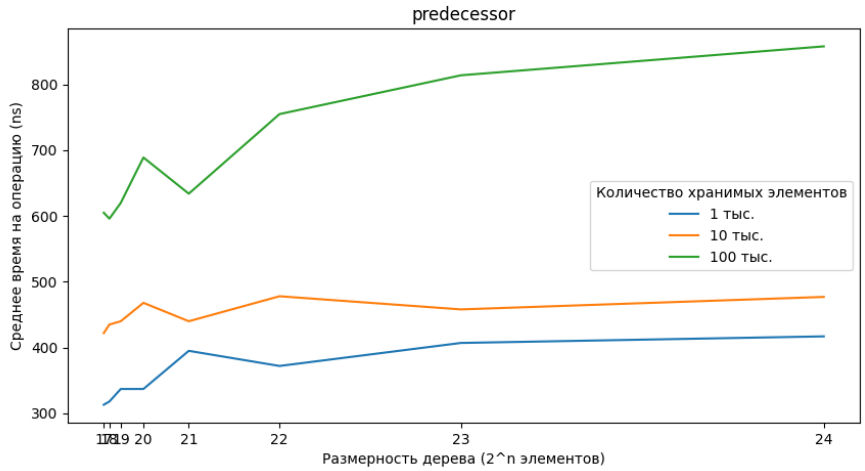

Рис. 4: Производительность операции predecessor

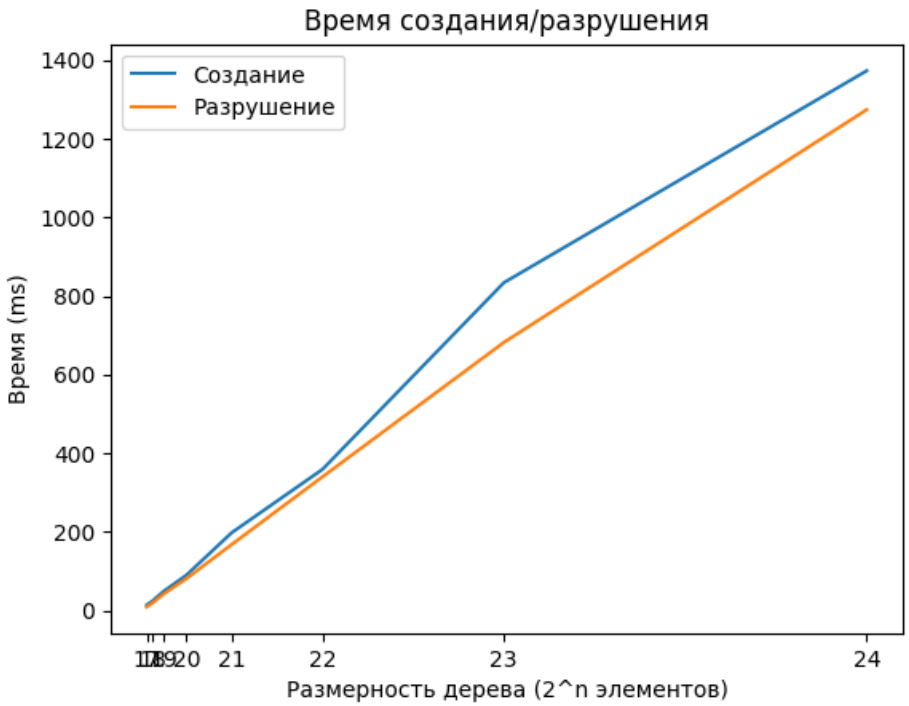

Рис. 5: Производительность конструктора/деструктора

6 Заключение

...

Список литературы

[1] Артур Хашаев "Invizory". *Дерево ван Эмде Боаса*. 3 авг. 2011. URL: https://habr.com/ru/post/125499/.

[2] Thomas H. Cormen. *Introduction to Algorithms: Third Edition*. 2009. URL: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf.

[3] *Determining the space complexity of van Emde Boas trees*. URL: https://mathoverflow.net/questions/2245/determining-the-space-complexity-of-van-emde-boas-trees.

[4] Marcel Ehrhardt. *An In-Depth Analysis of Data Structures Derived from van-Emde-Boas-Trees*. 12 окт. 2015. URL: https://www.mi.fu-berlin.de/inf/groups/ag-ti/theses/download/Ehrhardt15.pdf.

[5] Peter van Emde Boas. *History of the van Emde Boas Trees*. 18 марта 2015. URL: https://toc.csail.mit.edu/node/757.

[6] Rolf Karlsson. *van Emde Boas trees*. URL: https://fileadmin.cs.lth.se/cs/Personal/Rolf_Karlsson/lect12.pdf.

[7] Harish Rajput. *van Emde Boas ⌈ vEB ⌋ Trees*. 11 дек. 2015. URL: https://leprofesseur.org/algorithms-lecture-019-van-emde-boas-veb-trees/.

[8] *Van Emde Boas tree*. URL: https://en.wikipedia.org/wiki/Van_Emde_Boas_tree.

[9] *van Emde Boas Trees*. 17 мая 2016. URL: https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf.

[10] *Дерево ван Эмде Боаса*. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%B2%D0%B0%D0%BD_%D0%AD%D0%BC%D0%B4%D0%B5_%D0%91%D0%BE%D0%B0%D1%81%D0%B0.