МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент математического и компьютерного моделирования

ДОКЛАД
о практическом задание по дисциплине АИСД
«Дерево ван Эмде Боаса»

направление подготовки 09.03.03 «Прикладная информатика»
профиль «Прикладная информатика в компьютерном дизайне»

Выполнил студент
гр. Б9121-09.03.03пикд
Курышев Виктор Иванович _____
(подпись)
Руководитель практики
Доцент ИМКТ А.С Кленин
(должность, уч. звание)
_____
(подпись)
«_____» _____ 2022г.

Доклад защищен:
С оценкой _____

Рег. № _____

«____» _____ 2022 г.

г. Владивосток
2022

Оглавление

Список литературы

1. https://neerc.ifmo.ru/wiki/index.php?title=Дерево_ван_Эмде_Боаса
2. https://habr.com/ru/post/125499/
3. https://ru.frwiki.wiki/wiki/Arbre_de_Van_Emde_Boas
4. https://wiki5.ru/wiki/Van_Emde_Boas_tree
5. https://vk.com/video300356125_456239500?list=6ea77d78a2e4fe8e22
6. https://vk.com/video300356125_456239501?list=7ab36723b3063445c1
7. https://vk.com/video300356125_456239502?list=de626e26e90a6553d6
8. https://en.wikipedia.org/wiki/Van_Emde_Boas_tree
9. https://www.geeksforgeeks.org/van-emde-boas-tree-set-1-basics-and-construction/
10. https://away.vk.com/away.php
11. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/lecture-4-divide-conquer-van-emde-boas-trees/
12. https://natsugiri.hatenablog.com/entry/2016/10/12/021502
13. http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf
14. https://kopricky.github.io/code/Academic/van_emde_boas_tree.html
15. https://iq.opengenus.org/van-emde-boas-tree/
16. https://sandbigbox.com/wiki/ru/Van_Emde_Boas_tree
17. https://www.youtube.com/watch?v=ZrV7GiuMNo4
18. https://www.youtube.com/watch?v=7LTEdwJs1ao
19. https://www.youtube.com/watch?v=7LTEdwJs1ao
20. https://www.youtube.com/watch?v=r9EIAUh_V0s
21. https://bjpcjp.github.io/pdfs/math/van-emde-boas-trees-ITA.pdf
22. https://fileadmin.cs.lth.se/cs/Personal/Rolf_Karlsson/lect12.pdf
23. https://www-di.inf.puc-rio.br/~laber/vanEmdeBoas.pdf
24. https://ru.wikibrief.org/wiki/Van_Emde_Boas_tree
25. https://github.com/TISparta/Van-Emde-Boas-tree
26. https://www.programmersought.com/article/96406063495/
27. https://www.geeksforgeeks.org/van-emde-boas-tree-set-1-basics-and-construction/

Формальная постановка задачи

Исследовать и реализовать **vEB дерево** (Дерево ван Эмде Боаса)

1. Изучить алгоритм Дерево ван Эмде Боаса и описать его.
2. Реализовать алгоритм Дерево ван Эмде Боаса со всеми его операциями.
3. Выполнить исследование

Вступление

Дерево ван Эмде Боаса — также известное как **дерево vEB** или **приоритетная очередь ван Эмде Боаса**, структура данных, представляющая собой дерево поиска, позволяющее хранить целые неотрицательные числа в интервале [0;2k) и осуществлять над ними все соответствующие дереву поиска операции.
Проще говоря, данная структура позволяет хранить k-битные числа и производить над ними операции find, insert, remove, next, prev, min, max и некоторые другие операции, которые присущи всем деревьям поиска. Особенностью этой структуры является то, что все операции выполняются за O(logk), что асимптотически лучше, чем O(logn) в большинстве других деревьев поиска, где n — количество элементов в дереве. Он был изобретен командой во главе с голландским ученым Питером ван **Эмде Боасом в 1975 году.**

Применение алгоритма

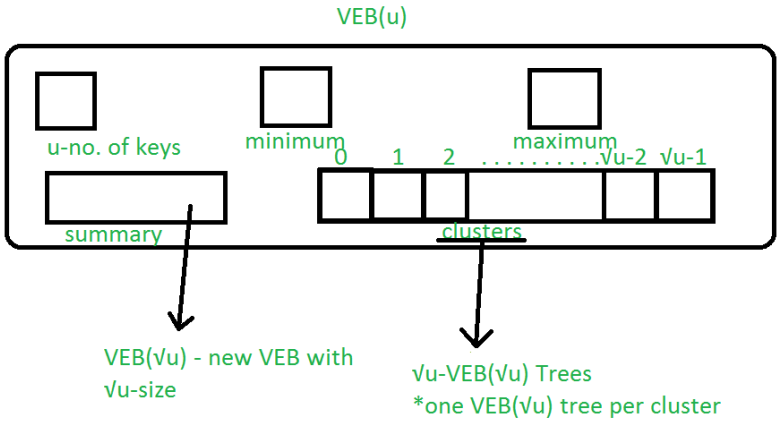
Деревья ван Эмде Боаса можно использовать где угодно вместо обычного бинарного дерева поиска, если ключи в дереве поиска являются целыми числами в некотором фиксированном диапазоне. Таким образом, для приложений, где вам нужно найти целое число в наборе, наиболее близком к некоторому другому целому, использование vEB-дерева потенциально может быть быстрее, чем использование простого сбалансированного двоичного дерева поиска.

Операции

- **Find(x)** – поиск числа x в дереве.
- **RemoveVEB(x)** – удаление числа x.
- **minimumVEB(), maximumVEB()** – найти минимум или максимум в дереве.
- **Insert(x)** – вставка числа x в дерево.

- **SuccessorVEB(), PredecessorVEB** – поиск следующего или предшествующего числа после\перед x, которое содержится в дереве.

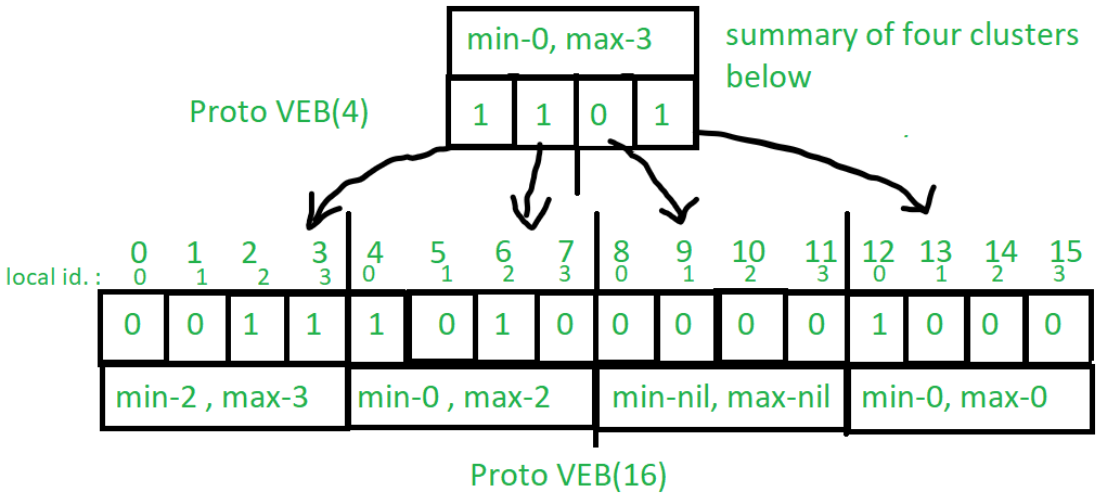
Структура



Дерево Ван Эмде Боаса – это рекурсивно определенная структура.

1. **u**: количество ключей, присутствующих в дереве VEB.
2. **Минимум:** содержит минимальный ключ, присутствующий в дереве VEB.
3. **Максимум:** содержит максимальное количество ключей, присутствующих в дереве VEB.
4. **Резюме:** указывает на новый ВЭБ ($\sqrt{u}$) Дерево, которое содержит обзор ключей, присутствующих в массиве кластеров.
5. **Кластеры:** массив размера $\sqrt{u}$ каждое место в массиве указывает на новый ВЭБ ($\sqrt{u}$) Дерево.

См. Изображение ниже, чтобы понять основы Дерева Ван Эмде Боаса, хотя оно не представляет фактическую структуру Дерева Ван Эмде Боаса:



Поддерживаемые операции

Minimum, maximum

vEB дерево хранит минимум и максимум в качестве своих атрибутов, поэтому мы сможем его вернуть, если оно есть или же 0 в противном случае.

int minimumVEB{

return minimum;

}

int maximumVEB{

return maximum;

}

empty

Чтобы определить, пусто ли дерево, будем изначально инициализировать поле **min** числом, которое не лежит в интервале [0;2k). Назовем это число **none**. Например, это может быть −1, если мы храним в числа в знаковом целочисленном типе, или 2k, если в беззнаковом. Тогда проверка на пустоту дерева будет заключаться лишь в сравнении поля **min** с этим числом.

```
boolean empty(t: Tree):
 if t.min == none
   return true
 else
  return false
```

Find(x)

Алгоритм поиска сам напрашивается из вышеописанной структуры:

- если дерево пусто, то число не содержится в нашей структуре.
- если число равно полю min или max, то число в дереве есть.
- иначе ищем число low(x)low(x) в поддереве children[high(x)].

```
boolean find(t: Tree, x: int):
 if empty(t)
   return false
 if t.min == x or t.max == x
   return true
 return find(t.children[high(x)], low(x))
```

Заметим, что, выполняя операцию find, мы либо спускаемся по дереву на один уровень ниже, либо, если нашли нужный нам элемент, выходим из нее. В худшем случае мы спустимся от корня до какого-нибудь 1-дерева, то есть выполним операцию find столько раз, какова высота нашего дерева. На каждом уровне мы совершаем O(1) операций. Следовательно время работы O(logk).
Операция вставки элемента x состоит из нескольких частей:

- если дерево пусто или в нем содержится единственный элемент (min=max), то присвоим полям min и max соответствующие значения. Делать что-то еще бессмысленно, так как информация записанная в minmin и maxmax полностью описывает состояние текущего дерева и удовлетворяет структуре нашего дерева.
- иначе:
    - если элемент x больше max или меньше min текущего дерева, то обновим соответствующее значение минимума или максимума, а старый минимум или максимум добавим в дерево.

- вставим во вспомогательное дерево aux число high(x), если соответствующее поддерево children[high(x)] до этого было пусто.
- вставим число low(x) в поддерево children[high(x)], за исключением ситуации, когда текущее дерево — это 1-дерево, и дальнейшая вставка не требуется.

```
function insert(t: Tree, x: int):
 if empty(t)                                    // проверка на пустоту текущего дерева
   t.min = x
   t.max = x
 else
   if t.min == t.max                            // проверка, что в дереве один элемент
     if T.min < x
       t.max = x
     else
       t.min = x
  else
     if t.min > x
       swap(t.min, x)                           // релаксация минимума
     if t.max < x
       swap(t.max, x)                           // релаксация максимума
     if t.k != 1
       if empty(t.children[high(x)])
         insert(t.aux, high(x))                 // вставка high(x) во вспомогательно дерево aux
       insert(t.children[high(x)], low(x))      // вставка low(x) в поддерево children[high(x)]
```

remove

Удаление из дерева также делится на несколько подзадач:

- если min = max = x, значит в дереве один элемент, удалим его и отметим, что дерево пусто.
- если x = min, то мы должны найти следующий минимальный элемент в этом дереве, присвоить min значение второго минимального элемента и удалить его из того места, где он хранится. Второй минимум — это либо max, либо children[aux.min] (для случая x=max действуем аналогично).
- если же x≠min и x≠max, то мы должны удалить low(x)l из поддерева children[high(x)].

Так как в поддеревьях хранятся не все биты исходных элементов, а только часть их, то для восстановления исходного числа, по имеющимся старшим и младшим битам, будем использовать функцию merge. Также нельзя забывать, что если мы удаляем последнее вхождение x, то мы должны удалить high(x) из вспомогательного дерева.

```
function remove(t: Tree, x: int):
 if t.min == x and t.max == x              // случай, когда в дереве один элемент
   t.min = none
   return
 if t.min == x
   if empty(t.aux)
     t.min = t.max
     return
   x = merge(t.aux.min, t.children[t.aux.min].min)
   t.min = x
 if t.max == x
   if empty(t.aux)
     t.max = t.min
     return
 else
     x = merge(t.aux.max, t.children[t.aux.max].max)
     t.max = x
 if empty(t.aux)                           // случай, когда элемента x нет в дереве
   return
 remove(t.children[high(x)], low(x))
 if empty(t.children[high(x)])             // если мы удалили из поддерева последний элемент
   remove(t.aux, high(x))                  // то удаляем информацию, что это поддерево не пусто
```

successor и predeccessor

Алгоритм нахождения следующего элемента, как и два предыдущих, сводится к рассмотрению случая, когда дерево содержит не более одного элемента, либо к поиску в одном из его поддеревьев:

- если дерево пусто, или максимум этого дерева не превосходит x, то следующего элемента в этом дереве не существует.
- если x меньше поля min, то искомый элемент и есть min.
- если дерево содержит не более двух элементов, и x <max, то искомый элемент max.
- если же в дереве более двух элементов, то:
    - если в дереве есть еще числа, большие xx, и чьи старшие биты равны high(x), то продолжим поиск в поддереве children[high(x)], где будем искать число, следующее после low(x).
    - иначе искомым элементом является либо минимум следующего непустого поддерева, если такое есть, либо максимум текущего дерева в противном случае.

```
int next(t: Tree, x: int)
 if empty(t) or t.max <= x
   return none;                                                    // следующего элемента нет
 if t.min > x
   return t.min;
 if empty(t.aux)
   return t.max;                                                   // в дереве не более двух элементов
 else
   if not empty(t.children[high(x)]) and t.childen[high(x)].max > low(x)
     return merge(high(x), next(t.children[high(x)], low(x)));     // случай, когда следующее число начинается с high(x)
   else                                                            // иначе найдем следующее непустое поддерево
     int nextHigh = next(t.aux, high(x));
     if nextHigh == none
       return t.max;                                               // если такого нет, вернем максимум
    else
       return merge(nextHigh, t.children[nextHigh].min);           // если есть, вернем минимум найденного поддерева
```

Преимущества и недостатки

Преимущества

Главным преимуществом данной структуры является ее быстродействие. Асимптотически время работы операций дерева ван Эмде Боаса лучше, чем, например, у АВЛ, красно-черных, 2-3, splay и декартовых деревьев уже при небольшом количестве элементов. Конечно, из-за довольно непростой реализации возникают немалые постоянные множители, которые снижают практическую эффективность данной структуры. Но все же, при большом количестве элементов, эффективность дерева ван Эмде Боаса проявляется и на практике, что позволяет нам использовать данную структуру не только как эффективное дерево поиска, но и в других задачах. Например:

- сортировка последовательности из n чисел. Вставим элементы в дерево, найдем минимум и $n-1n-1$ раз вызовем функцию nextnext. Так как все операции занимают не более O(logk) времени, то итоговая асимптотика алгоритма $O(n \cdot logk)$, что даже лучше, чем цифровая сортировка, асимптотика которой $O(n \cdot k)$.
- Алгоритм Дейкстры. Данный алгоритм с использованием двоичной кучи для поиска минимума работает за $O(E \cdot logV)$, где V — количество вершин в графе, а E — количество ребер между ними. Если же вместо кучи использовать дерево ван Эмде Боаса, то релаксация и поиск минимума будут занимать уже не logV, а logk, и итоговая асимптотика этого алгоритма снизится до $O(E \cdot logk)$.

Недостатки

- существенным недостатком данной структуры является то, что она позволяет хранить лишь целые неотрицательные числа, что существенно сужает область ее применения, по сравнению с

другими деревьями поиска, которые не используют внутреннюю структуру элементов, хранящихся в них.

- другим серьезным недостатком является количество занимаемой памяти. Дерево, хранящее k-битные числа, занимает $\Theta(2k)$ памяти, что несложно доказывается индукцией, учитывая, что $S(2k) = (2k/2+1) \cdot S(2k/2) + O(2k/2)$, где $S(2i)$ — количество памяти, занимаемое деревом, в котором хранятся i-битные числа. Впрочем, можно попытаться частично избежать огромного расхода памяти, создавая необходимые поддеревья «лениво», то есть только тогда, когда они нам потребуются.

Тестирование

Входные данные