Дървета на ван Емде Боас. Разширение и допълнителен анализ.

1. Как да модифицираме vEB дърво, за да поддържа дублиращи се ключове?

   За да поддържа дублиращи се ключове, за всяко базово vEB дърво с $u = 2$, вместо да съхраняваме само бит във всеки от записите в неговия масив/член данни, то трябва да съхранява цяло исло, представящо колко елемента от тази стойност съдържа vEB. В случая трябва във всяко базово vEB дърво с $u = 2$, освен $\min$ и $\max$ елементите, да имаме и $cntMin$ и $cntMax$ атрибути, за да оказват броя на всеки един от тях.

2. Как да модифицираме vEB дърво, за да поддържаме ключове, които имат/поддържат асоциирани данни от външни/сателитни обекти?

   За всеки ключ, който е минимален за някое vEB дърво, ще трябва да съхраняваме сателитните му данни с минималната стойност, тъй като ключът не се появява в поддървото. Останалата част от сателитните данни ще се съхранява заедно с ключовете на vEB дърветата с размер 2 (базовите vEB структури). Експлицитно за всяко vEB дърво, което не е от $summary$ тип, съхраняваме указател в допълнение към $\min$. Ако $\min$ е равно на NIL, указателят също трябва да сочи към NIL. В противен случай указателя трябва да сочи към сателитните данни, свързани с този минимум. Във базовото vEB дърво с размер 2 ще имаме два допълнителни указателя, които ще сочат към сателитните данни на минимума и максимума, или към NIL, ако те не съществуват. В случая, когато $\min = \max$, указателите ще сочат към един и същи блок от данни.

3. Напишете псевдокод за процедура, която създава празно дърво на ван Емде Боас.

   Определяме процедурата за всяко u, което е степен на 2. Ако $u = 2$, тогава просто подкрепете този факт с масив с дължина 2, който съдържа нули и в двата си записа. Ако $u = 2k > 2$, тогава просто създаваме vEB дърво, наречено $summary$ с $u = 2^{\lceil k/2 \rceil}$. Също така правим масив, наречен $cluster$ с дължина $2^{\lceil k/2 \rceil}$ с всеки запис, инициализиран в празно дърво на vEB с $2^{\lfloor k/2 \rfloor}$. И накрая създаваме $\min$ и $\max$ елементи инициализирани с NIL.

```
struct Node{
    ll u, size; // 2^u capacity (Universe size), real size
    ll min, max;
    Node *summary;
    vector<Node*> cluster;
    Node(int u=1) { // Constructor for Base Node with capacity=2
        min = max = -1;
        this->u=u; // size of Universe 2^u
        size = 0; // No elements at the initialization
        summary = nullptr;
        cluster.assign(1LL << ((u + 1)>>1), nullptr);
    }
};
void build(Node *&V, int u){
    V = new Node(u);
    if(u == 1) return; // Base case vEB
    build(V->summary, (u+1)>>1);
    ll SIZE = (ll)V->cluster.size(); // cluster size can be sqrt(ll_max) which is ll
    for(ll i = 0; i < SIZE; ++i) build(V->cluster[i], u>>1);
}
```

4. Какво се случва, ако извикаме $vEB–TREE–INSERT$ с елемент, който вече е във vEB дървото?
   Какво се случва, ако извикаме $vEB–TREE–DELETE$ с елемент, който не е във vEB дървото?
   Обяснете поведението което процедурите демонстрират. Покажете как да модифицирате vEB дърво и неговите операции, така че да може да проверяваме в константно време дали даден елемент присъства в множеството описано от дървото.

   Да предположим, че x вече е във V и извикваме $INSERT$. Процедурата по добавяне имаше следния алгоритъм:

```
vEB–TREE–INSERT(V, x)
1.  if V.min == NIL
2.          vEB–EMPTTY–TREE–INSERT(V, x)
3.  else if x < V.min
4.          exchange x with V.min
5.  if V.u > 2
6.          if vEB–TREE–MINIMUM(V.cluster[high(x)]) == NIL
7.                  vEB–TREE–INSERT(V.summary, high(x))
8.                  vEB–EMPTY–TREE–INSERT(V.cluster[high(x)], low(x))
9.          else vEB–TREE–INSERT(V.cluster[high(x)], low(x))
10.  if x > V.max
11.         V.max = x
```

   Очевидно няма да може да удовлетворим редове 1, 3, 6 или 10, така че ще влизаме в else случая на ред 9 всеки път докато не стигнем до базовия случай. Ако x е вече е в базовото дърво, тогава няма да променим нищо. Ако x се съхранява в атрибута $\min$, на дърво, което не е базово, то тогава ще вмъкнем дубликат от него в някакво дърво което е от базов тип. Сега да предположим, че извикаме $DELETE$ когато x не е във V. Процедурата по изтриване на елемент имаше следния алгоритъм:

```
vEB–TREE–DELETE(V, x)
1.  if V.min == V.max
2.          V.min = NIL
3.          V.max = NIL
4.  elseif V.u == 2
5.          if x == 0
6.                  V.min = 1
7.          else V.min = 0
8.          V.max = V.min
9.  else if x == V.min
10.                 first–cluster = vEB–TREE–MINIMUM(V.summary)
11.                 x = index(first–cluster,
                            vEB–TREE–MINIMUM(V.cluster[first–cluster]))
12.                 V.min = x
13.         vEB–TREE–DELETE(V.cluster[high(x)], low(x))
14.         if vEB–TREE–MINIMUM(V.cluster[high(x)]) == NIL
15.                 vEB–TREE–DELETE(V.summary, high(x))
```

```
16.          if x == V.max
17.              summary-max = vEB-TREE-MAXIMUM(V.summary)
18.              if summary-max == NIL
19.                  V.max = V.min
20.              else V.max = index(summary-max,
                 vEB-TREE-MAXIMUM(V.cluster[summary-max]))
21.      elseif x == V.max
22.          V.max = index(high(x),
                 vEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

Ако във V има само един елемент, редовете от 1 до 3 ще го изтрият, независимо за кой елемент става дума. За да влезем в elseif условието от ред 4, x не трябва да е равно на 0 или 1 и vEB дървото трябва да е с размер 2. В този случай ние изтриваме елемента $\max$, независимо от това какъв е. Тъй като рекурсивното извикване винаги ни поставя в този случай, ние винаги изтриваме елемент, който не трябва. За да избегнем тези проблеми, дръжте и поддържайте спомагателен масив A с u елемента. Съхранете на $A[i] = 0$, ако i не е в дървото или 1, ако е. Тъй като може да извършваме константни актуализации по време на този масив, това няма да повлияе на времето за изпълнение на никоя от нашите операции. Когато вмъкваме x, първо проверяваме дали $A[x] = 0$. Ако не е, то просто ще излезем от процедурата (return). В противен случай ще презапишем $A[x] = 1$ и ще продължим с обичайната процедура по вмъкване. Когато изтриваме елемент x, първо ще проверяваме дали $A[x] = 1$. Ако това условие не е изпълнено, тогава просто ще излезем от процедурата (return). Ако обаче е изпълнено, ще презапишем $A[x] = 0$ и ще продължим обичайно с процедурата по изтриване.

5. Да предположим, че вместо $\sqrt[\uparrow]{u}$ на брой клъстери, всеки с размер $\sqrt[\downarrow]{u}$, построяваме vEB дървото да има $u^{1/k}$ клъстери, всеки с размер на Вселената $u^{1-1/k}$, където $k > 1$ е константа. Ако трябва да модифицираме операциите по подходящ начин, какво би било времето им на работа? За целите на анализа приемете че $u^{1/k}$ и $u^{1-1/k}$ са винаги цели числа.

Аналогично на анализа на рекурентната зависимост, която получихме при коефициент на свиване най-много $\sqrt[\uparrow]{u}$: $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1)$, анализираме следната рекурентна зависимост:

$$T(u) \leq T(u^{1-1/k}) + T(u^{1/k}) + O(1).$$

Това е добър избор за анализ, тъй като за много операции първо проверяваме $summary$ vEB дървото, което ще има размер $u^{1/k}$ (втория израз)
и после е възможно да проверим vEB дървото някъде в някой клъстер, който ще има размер $u^{1-1/k}$ (първия израз). Полагаме $T(2^m) = S(m)$ и неравенството добива вида:

$$S(m) \leq S(m(1 - 1/k)) + S(m/k) + O(1).$$

Ако $k > 2$, първия израз доминира, и следователно от $master\ method$ теоремата, ще имаме, че $S(m)$ ще работи в порядъка на $O(\lg m)$. Това означава, че T ще работи в порядъка на $O(\lg \lg u)$, което е асимптотично същото като оригиналната идея, в която взимаме просто корен квадратен за броя на клъстерите.

6. Създаването на vEB дърво с размер на вселената u изисква $O(u)$ време. Да предположим, че искаме експлицитно да отчитаме това време. Какъв е най-малкия брой операции n, за които амортизираното време на всяка операция във vEB дърво е $O(\lg \lg u)$? (Грубо казано искаме да знаем, в какви ситуации/случай спрямо броя на заявките и размерността на колекцията, която ще разглеждаме, ще е разумно да използваме дърво на ван Емде Боас.)

   Нека $n = u / \lg \lg u$. Тогава извършването на n операции отнема $c(u + n \lg \lg u)$ време за някаква константа c. Използвайки съвкупния амортизиран анализ, раделяме на n, за да видим, че амортизираната цена на всяка операция е с $c(\lg \lg u + \lg \lg u) = O(\lg \lg u)$ за операция. Следователно, ще имаме че $n \geq u / \lg \lg u$.