Universos Discretos y Finitos

Para algunos tipos de datos en particular (como enteros en un rango o strings) es posible diseñar otros algoritmos y estructuras de datos más eficientes".

Ordenamiento

→ **Counting Sort**: queremos ordenar $A[1 \dots n]$, $A[i] \in [1 \dots u]$

1) Creamos $C[1 \dots u]$ y recorremos A para ir actualizando C:

for $i \in [1 \dots n]$:
  $C[A[i]] \leftarrow C[A[i]] + 1$

★ Vamos contando cuántos elementos $j \in [1, u]$ hay en A.

2) Luego, reescribimos A poniendo $C[j]$ copias del valor j.

$$\boxed{\begin{array}{l} \text{tiempo: } O(n+u) \\ \text{memoria: } O(n+u) \end{array}}$$

Conveniente cuando el universo es pequeño → $u = O(n)$

→ **Bucket Sort**: cuando queremos diferenciar los números (claves) que son iguales (en caso de que los elementos tengan información satélite asociada). // Ordenar $A = [5, 2, 3, 3, 2, 3, 4]$, $u = 9$

5	2	3	3	2	3	4
1	2	3	4	5	6	7

1) Hacemos Counting Sort // C=

0	2	3	1	1	0	0	0	0
1	2	3	4	5	6	7	8	9

2) Modificamos arreglo $C[1, \dots u]$ que copia la posición donde empieza a escribirse las copias del valor j en A.

$C[0] = 1$
for $j \in [2, n]$:
  $C[j] \leftarrow C[j-1] + C[j]$

// C=

	1	2	3	4	5	6	7	8	9	→ elemento
1	1	3	6	7	8	8	8	8	8	→ posición en B.
0	1	2	3	4	5	6	7	8		

3) Creamos arreglo $B[1, n]$ con los valores de A ordenados.

for $i \in [1, n]$:
  $B[C[A[i] - 1]] \leftarrow A[i]$
  $C[A[i] - 1]$ ++

// B =

2	2	3	3	3	4	5
1	2	3	4	5	6	7

$$\boxed{\begin{array}{l} \text{tiempo: } O(n+u) \\ \text{memoria: } O(2n+u) \end{array}}$$

→ **Radix Sort** : hace varias veces Bucket Sort, ordenando del bit menos a más significativo. Universo = {0, 1}

Cada pasada del Bucket Sort toma tiempo $O(n+2) = O(n)$

Los números de A tienen $\lceil \log_2 u \rceil$ bits ⟹ $\boxed{\text{tiempo: } O(n \cdot \log_2 u)}$

Si ordenamos de a k bits, ⟹ tiempo: $O\left((n+2^k) \cdot \frac{\log_2 u}{k}\right)$ $n \geq 2^k$, $k \leq \log(n)$

$$\boxed{\begin{array}{l} \text{tiempo: } O\left(n \frac{\log(u)}{\log(n)}\right) = O(n \log_n u) \\ \text{memoria: } O(n) \end{array}}$$

★ Si $u = O(n^c)$, el tiempo es constante!

// $A = [5, 14, 6]$

5 = 0101	14 = 1110	5 = 0101	5	$A' = [5, 6, 14]$ ✓
14 = 1110	6 = 0110	14 = 1110	6	
6 = 0110	5 = 0101	6 = 0110	14	

Van Emde Boas Tree espacio $O(u)$, tiempo $O(\log \log u)$

Objetivo: encontrar el predecesor, insertar, buscar y borrar un elemento

n enteros $\in [0, u-1]$

El vEB es una estructura recursiva que particiona el universo recursivamente en subuniversos

vEB(u):
- int min, max, size : mínimo, máximo y # elementos del árbol
- array bottom $[0, \ldots, \sqrt{u} - 1]$ de vEB($\sqrt{u}$)
  // bottom(i) contiene los elementos que van de $i \cdot \sqrt{u}$ hasta $(i+1) \cdot \sqrt{u} - 1$
- subárbol vEB($\sqrt{u}$) top que nos indica qué subuniversos no son vacíos.

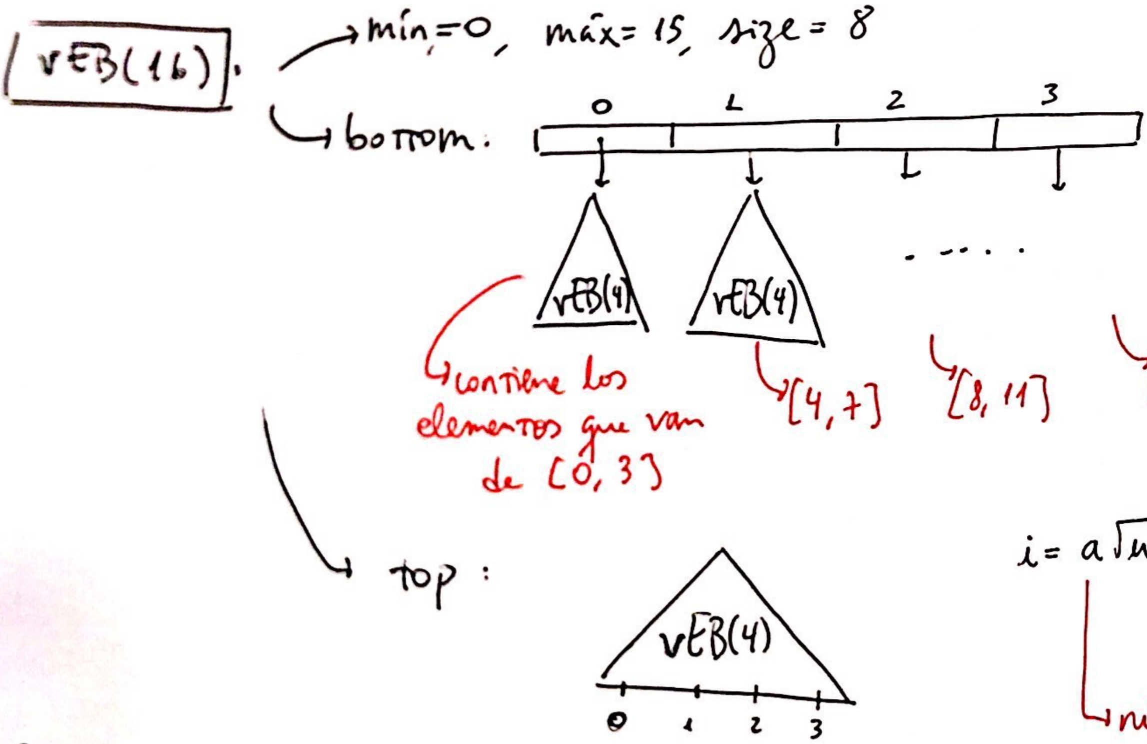
★ Detenemos la recursión cuando el tamaño del subuniverso sea $O(\log^c u)$, c constante.

Cada elemento podemos escribirlo en bits de la siguiente forma: $i = a\sqrt{u} + b$ $\qquad i = \underbrace{\boxed{a \mid b}}_{\log u}$

↳ funciona como un shift

// Ej: 0, 3, 5, 7, 8, 10, 13, 15

$u = 16$



$i = a\sqrt{u} + b$

↳ b: valor relativo dentro del subuniverso

↳ a: número del subuniverso.

Operaciones

→ Búsqueda predecesor: pred(i) $\quad i = a\sqrt{u} + b$

```
pred(i):
    # ver si el predecesor también está en el subuniverso a
    if bottom[a].size > 0  AND  bottom[a].min ≤ b:
            return  bottom[a] → pred(b)  # busco el pred. de "b" en este vEB
  # no se encuentra en el subuniverso a, hay que buscarlo en el
  # subuniverso no vacío + cercano a la izquierda de a.
  # Con el TOP podemos saber esto
  else:
      a' ← TOP → pred(a - 1)
      if a' != null :  return  a'√u + bottom[a'].max
```

$5 = 01\,01$

En el ejemplo: pred(5): $a = 1$, $b = 1$

En el vEB que está en bottom[1] están los elementos [4, 7]

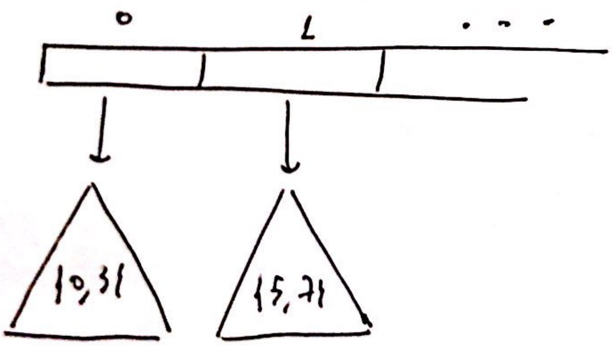
que, en nuestro caso, serían los elementos {5, 7}

Luego, bottom[1].size = 2 > 0 y $a_5 = 01 = 1$ $a_7 = 01 = 1$

bottom[1].min = 1 ≤ 1 $b_5 = 01 = 1$, $b_7 = 10 = 2$

Entonces, hacemos bottom[1] → pred(1)

Como sabemos, en el vEB que está en bottom[1] solo encontraremos los elementos {5, 7}.



Por lo que, en la siguiente llamada buscaremos el vEB más cercano de la izquierda (que es el de bottom[0]) y retornaremos el máximo de su árbol.

a' ← TOP → pred(0)

a' = 0

bottom[0].max = 3

Luego, pred(5) = 3

Las otras operaciones están en el apunte, pero siguen la misma idea.