Estructuras de datos útiles (AKA piolas)

Emanuel Lupi

Facultad de Matemática Astronomía Física y Computación
Universidad Nacional de Córdoba

Training Camp 2020

Contenidos

Contenidos

¿Qué es una estructura de datos útil?

Estructura: Visión del usuario.

Una estructura de datos adecuada es un tipo abstracto de datos que nos sirve para responder “queries” sobre un conjunto de datos, posiblemente sujetos a modificaciones durante el proceso de query.

- Se le llama “queries” simplemente a preguntas que nos interesa hacerle a la estructura. Notar que esta forma de ver lo que es una estructura es extremadamente general y virtualmente cualquier cosa se puede pensar como una estructura. No obstante, esta forma de pensar resulta útil.

Contenidos

Visión del usuario de una tabla aditiva

Tabla aditiva

Dado un arreglo de r dimensiones, digamos de $n_1 \times n_2 \times \cdots \times n_r$, una tabla aditiva es una estructura que permite averiguar rápidamente la suma de los elementos de cualquier subarreglo contiguo (intervalo, rectángulo, paralelepípedo, etc). En este caso **NO** nos interesará realizar modificaciones a la matriz durante el proceso de consultas.

La estructura consta de dos fases de uso:

- Primero la estructura es inicializada con los datos del arreglo.
- Y luego se realizan una serie de consultas a la misma, sin modificar el arreglo.

Complejidad pretendida

La implementación que propondremos tendrá una complejidad:

- *Lineal* para la inicialización o preproceso (es decir, proporcional a la **cantidad de elementos** del arreglo).
- *Constante* para las queries (Es decir, responderemos cada consulta en $\mathbf{O}(\mathbf{1})$).

Contenidos

Arreglo acumulado

Definición

Dado un arreglo unidimensional v de n elementos $v_0, v_1, \cdots, v_{n-1}$, definimos el **arreglo acumulado** de v como el arreglo unidimensional V de $n+1$ elementos $V_0, \cdots, V_n$ y tal que:

$$V_i = \sum_{j=0}^{i-1} v_j$$

Notar que V es muy fácil de calcular en tiempo lineal utilizando programación dinámica:

- $V_0 = 0$
- $V_{i+1} = v_i + V_i, \forall\, 0 \leq i < n$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por intervalos $[i, j)$ con $0 \leq i \leq j \leq n$.
- La respuesta al query $[i, j)$ sera $Q(i, j) = \sum_{k=i}^{j-1} v_k$
- Pero:

$$Q(i, j) = \sum_{k=i}^{j-1} v_k = \sum_{k=0}^{j-1} v_k - \sum_{k=0}^{i-1} v_k = V_j - V_i$$

- Luego podemos responder cada query en tiempo constante computando una sola resta entre dos valores del arreglo acumulado.

Contenidos

Arreglo acumulado

Definición

Dado un arreglo bidimensional v de $n \times m$ elementos $v_{i,j}$ con $0 \leq i < n, 0 \leq j < m$, definimos el **arreglo acumulado** de v como el arreglo bidimensional V de $(n+1) \times (m+1)$ elementos tal que:

$$V_{i,j} = \sum_{a=0}^{i-1} \sum_{b=0}^{j-1} v_{a,b}$$

¿Podremos calcular fácilmente V en tiempo lineal como en el caso unidimensional? **¡Sí!** Utilizando programación dinámica.

- $V_{0,j} = V_{i,0} = 0 \ \forall \ 0 \leq i \leq n, 0 \leq j \leq m$
- $V_{i+1,j+1} = v_{i,j} + V_{i,j+1} + V_{i+1,j} - V_{i,j} \ \forall \ 0 \leq i < n, 0 \leq j < m$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por rectángulos $[i_1, i_2) \times [j_1, j_2)$ con $0 \leq i_1 \leq i_2 \leq n, 0 \leq j_1 \leq j_2 \leq m$.
- La respuesta al query $[i_1, i_2) \times [j_1, j_2)$ sera $Q(i_1, i_2, j_1, j_2) = \sum_{a=i_1}^{i_2-1} \sum_{b=j_1}^{j_2-1} v_{a,b}$
- Pero de manera similar a como hicimos para calcular el arreglo acumulado, resulta que:

$$Q(i_1, i_2, j_1, j_2) = V_{i_2,j_2} - V_{i_1,j_2} - V_{i_2,j_1} + V_{i_1,j_1}$$

- Luego podemos responder cada query en tiempo constante computando sumas y restas de cuatro valores del arreglo acumulado.

Una imagen vale mas que mil palabras

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

Cuadro: Matrix

1	4	8	15
5	11	20	32
11	22	35	49

Cuadro: Matrix acumulada

Contenidos

Tarea

- http://www.spoj.pl/problems/KPMATRIX/
- http://www.spoj.pl/problems/TEM/
- http://www.spoj.pl/problems/MATRIX/

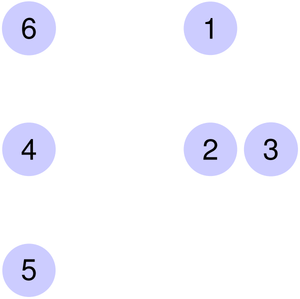
Contenidos

Visión del usuario del union find

Es una estructura que permite hacer eficientemente **Union** y **Find**
Union: Permite **unir** componentes. **Find**: permite **averiguar** la componente de un elemento.

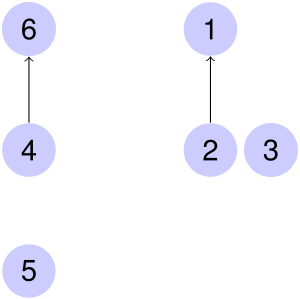
Idea base

Diremos que cada componente tiene un representante:

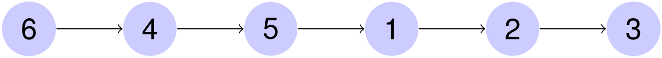


Idea base

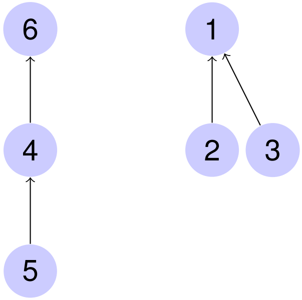
Diremos que cada componente tiene un representante:
Union entre componentes se da con la union de sus representantes



Un problema

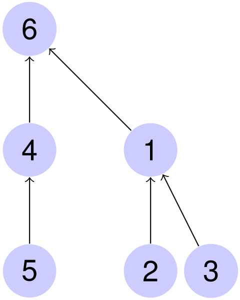


Unas soluciones



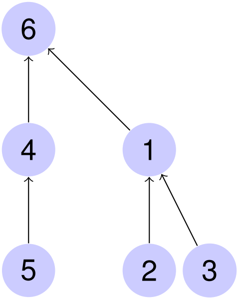
- Unir por rango. Siendo el rango el tamaño de la rama mas larga del árbol.
- Compresión de camino (Achatamiento).

Unas soluciones



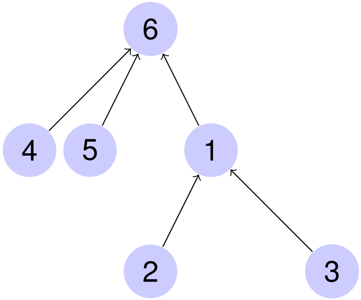
- Unir por rango. Siendo el rango el tamaño de la rama mas larga del árbol.
- Compresión de camino (Achatamiento).

Unas soluciones



- Unir por rango. Siendo el rango el tamaño de la rama mas larga del árbol.
- Compresión de camino (Achatamiento).

Unas soluciones



- Unir por rango. Siendo el rango el tamaño de la rama mas larga del árbol.
- Compresión de camino (Achatamiento).

Contenidos

Find

```
int uf[MAXN];

void uf_init(){
  for(int i=0 ; i<MAXN ; i++) uf[i] = -1;
}

// int uf_find(int x){return uf[x]<0?x:uf[x]=uf_find(uf[x]);}
int uf_find(int x){
  int rep = uf[x] < 0 ? x : uf_find(uf[x]);
  if (x < 0)
    return x;
  uf[x] = rep;
  return rep;
}
```

Union

```
bool uf_join(int x, int y){
  x=uf_find(x);
  y=uf_find(y);
  if(x==y) return false;
  if(uf[x]>uf[y]) swap(x,y);
  uf[x]+=uf[y];
  uf[y]=x;
  return true;
}
```

Contenidos

Tarea

- http://livearchive.onlinejudge.org/index.php?option=com_onlinejudge&Itemi
  ("Island", regional Polaca 2009)
- http://www.topcoder.com/stat?c=problem_statement&pm=2932
- http://www.topcoder.com/stat?c=problem_statement&pm=7921

Contenidos

Visión del usuario de RMQ

RMQ

Dado un arreglo v de n elementos $v_0, \cdots, v_{n-1}$, una estructura de RMQ permite responder rápidamente consultas por el valor

$$RMQ(i,j) = \min_{k=i}^{j-1} v_k, 0 \leq i < j \leq n$$

.

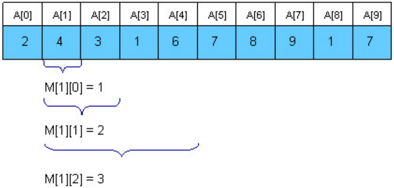
Visión del usuario (Continuada)

- Hablamos del mínimo pero según el problema puede interesar el máximo. Todo lo que hagamos para mínimo vale igual para máximo, intercambiando las nociones de máximo/mínimo.
- Opcionalmente, la estructura puede soportar modificaciones al arreglo v o no. Veremos una estructura que lo soporta y una que no.
- Hemos dicho que las queries de RMQ devuelven el mínimo valor en el intervalo, pero a veces puede ser útil devolver el **índice** de un mínimo valor. Todo lo que hagamos sirve igual para este caso, con sólo tener cuidado de guardar en la estructura índices en lugar de valores.

Contenidos

Estructura : Inicialización

Un mejor enfoque es preprocesar RMQ para sub matrices de longitud 2^k usando programación dinámica.
Mantendremos una matriz $M[0, N-1][0, log(N)]$ donde $M[i][j]$ es el índice del valor mínimo en la matriz secundaria que comienza en i con una longitud de 2^j. Aquí hay un ejemplo:



Estructura : Inicialización

Para calcular $M[i][j]$ debemos buscar el valor mínimo en la primera y segunda mitad del intervalo. Sabemos que las piezas pequeñas tienen 2^{j-1} de longitud, por lo que la recurrencia es:

$$M[i][j] = \begin{cases} M[i][j-1] \iff A[M[i][j-1]] \leq A[M[i+2^{j-1}-1][j-1]] \\ M[i+2^{j-1}-1][j-1] \textit{ caso contrario.} \end{cases}$$

Codigo fuente C++

```
void st_init(int M[MAXN][LOGMAXN], int A[MAXN], int N){
  int i, j;
  //initialize M for the intervals with length 1
  for (i = 0; i < N; i++)
    M[i][0] = i;
  //compute values from smaller to bigger intervals
  for (j = 1; 1 << j <= N; j++)
    for (i = 0; i + (1 << j) - 1 < N; i++)
      if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
        M[i][j] = M[i][j - 1];
      else
        M[i][j] = M[i + (1 << (j - 1))][j - 1];
}
```

Consulta

Una vez que tenemos estos valores preprocesados, vamos a mostrar cómo podemos usarlos para calcular $RMQ_A(i,j)$. La idea es seleccionar dos bloques que cubran completamente el intervalo $[i...j]$ y encontrar el mínimo entre ellos. Deje $k = \lfloor log(j-i+1) \rfloor$. Para calcular $RMQ_A(i,j)$ podemos usar la siguiente fórmula:

$$RMQ_A(i,j) = \begin{cases} M[i][k] \iff A[M[i][k]] \leq A[M[j-2^k+1][k]] \\ M[j-2^k+1][k] \textit{ caso contrario.} \end{cases}$$

$$M = \begin{bmatrix} 5 & 3 & 3 & 7 & 5 & 6 & 7 & 18 \end{bmatrix}$$

Consulta

Una vez que tenemos estos valores preprocesados, vamos a mostrar cómo podemos usarlos para calcular $RMQ_A(i,j)$. La idea es seleccionar dos bloques que cubran completamente el intervalo $[i...j]$ y encontrar el mínimo entre ellos. Deje $k = \lfloor log(j - i + 1) \rfloor$. Para calcular $RMQ_A(i,j)$ podemos usar la siguiente fórmula:

$$RMQ_A(i,j) = \begin{cases} M[i][k] \iff A[M[i][k]] \leq A[M[j-2^k+1][k]] \\ M[j-2^k+1][k] \; caso \; contrario. \end{cases}$$

$$M = \begin{bmatrix} 5 & 3 & 3 & 7 & 5 & 6 & 7 & 18 \end{bmatrix}$$

Contenidos

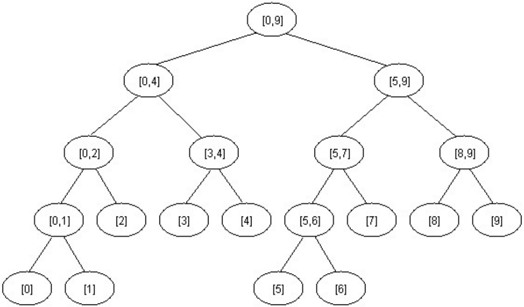
Descripción

Para resolver el problema RMQ también podemos usar segment tree (árboles de segmentos). Un segment tree es una estructura de datos tipo *heap* que se puede usar para realizar operaciones de actualización / consulta en intervalos de matriz en tiempo logarítmico. Definimos el árbol de segmentos para el intervalo [i, j] de la siguiente manera recursiva:

- El primer nodo guarda la información del intervalo $[i, j]$.
- Si $i < j$ el hijo izquierdo y derecho guardaran la información para el intervalo $[i, (i + j)/2]$ y $[(i + j)/2 + 1, j]$.

Notar que la altura de un segment tree para un intervalo con N elementos es $\lfloor log(N) \rfloor + 1$. Así es como se vería un árbol de segmento para el intervalo $[0, 9]$:

Seg tree



Implementación

Segment Tree se puede implementar con un array.
Si tenemos un nodo x que no es una hoja, el hijo izquierdo de x es $2 * x$ y el hijo derecho $2 * x + 1$.
Para resolver el problema de *RMQ* usando segment tree, debemos usar una matriz $M[1, 2 * 2\lfloor log(N) \rfloor + 1]$ donde $M[i]$ mantiene la posición de valor mínimo en el intervalo asignado al nodo i. Al principio, todos los elementos en M deben ser -1. El árbol debe inicializarse con la siguiente función (b y e son los límites del intervalo actual):

Codigo fuente C++

```
void initialize(int node, int b, int e, int M[MAXIND], int A[MAXN], int N){
  if (b == e)
    M[node] = b;
  else{
    initialize(2 * node, b, (b + e) / 2, M, A, N);
    initialize(2 * node + 1, (b + e) / 2 + 1, e, M, A, N);
    if(A[M[node * 2]] < A[M[node * 2 + 1]])
      M[node] = M[node * 2];
    else
      M[node] = M[node * 2 + 1];
  }
}
```

Implementación

Ahora podemos comenzar a hacer consultas. Si queremos encontrar la posición del valor mínimo en algún intervalo $[i, j]$ debemos usar la siguiente función:

query segment tree

```
int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int j){
  int p1, p2;
  if (i > e || j < b)
    return -1;

  if (b >= i && e <= j)
    return M[node];

  p1 = query(2 * node, b, (b + e) / 2, M, A, i, j);
  p2 = query(2 * node + 1, (b + e) / 2 + 1, e, M, A, i, j);

  if (p1 == -1)
    return p2;
  if (p2 == -1)
    return p1;
  if (A[p1] <= A[p2])
    return p1;
  return p2;
}
```

update segment tree

```
void update(int node, int b, int e, int M[MAXIND], int A[MAXN], int i){
  int p1, p2;
  if (i > e || i < b)
    return ;
  if(b == e) return;

  int m = (b + e) / 2 + 1;
  if(i < m) update(2 * node, b, (b + e) / 2, M, A, i);
  else update(2 * node + 1, (b + e) / 2 + 1, e, M, A, i);

  if(A[M[node * 2]] < A[M[node * 2 + 1]])
    M[node] = M[node * 2];
  else
    M[node] = M[node * 2 + 1];
}
```

Contenidos

Tarea

- http://www.spoj.pl/problems/KGSS/
- http://poj.org/problem?id=2374

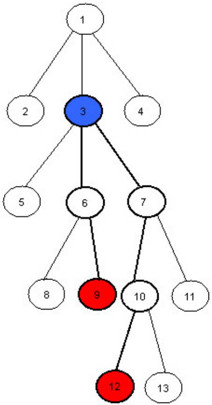
Contenidos

Visión del usuario de LCA

Dado un árbol con raíz de n nodos, una estructura de LCA permite responder rápidamente consultas por el **ancestro común más bajo** entre dos nodos (es decir, el nodo más alejado de la raíz que es ancestro de ambos).

- Sorprendentemente se puede reducir una estructura de LCA a una de RMQ!
- Luego no daremos explícitamente una estructura para LCA, sino que mostraremos como reducirla a RMQ para poder aplicar cualquiera de las técnicas ya vistas.

Visión del usuario de LCA



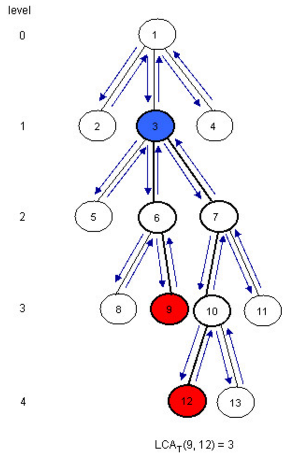
$LCA_T(9, 12) = 3$

Contenidos

Recorrido de DFS

- Utilizando un recorrido de DFS, podemos computar un arreglo E de $2n - 1$ posiciones que indica el orden en que fueron visitados los nodos por el DFS (cada nodo puede aparecer múltiples veces).
- Aprovechando este recorrido podemos computar también la **profundidad** (distancia a la raíz) de cada nodo i, que notaremos L_i.
- También guardaremos el índice de la primer aparición de cada nodo i en E, que notaremos H_i (Cualquier posición servirá, así que es razonable tomar la primera).

Recorrido de DFS



Euler Tour:

1	2	1	3	5	3	6	8	6	9	6	3	7	10	12	10	13	10	7	11	7	3	1	4	1

closest node from the root

Utilización del RMQ

- La observación clave consiste en notar que para dos nodos i, j con $i \neq j$:

$$LCA(i,j) = RMQ(\min(L_i, L_j), \max(L_i, L_j))$$

- Tomamos mínimo y máximo simplemente para asegurar que en la llamada a *RMQ* se especifica un rango válido ($i < j$).
- Notar que en la igualdad anterior, $RMQ(i,j)$ compara los elementos de v por sus valores de profundidad dados por P.
- La complejidad de la transformación del LCA al RMQ es $O(n)$, con lo cual la complejidad final de las operaciones será la del *RMQ* utilizado.

LCA RMQ

Supongamos que: $H[u] < H[v]$ (de lo contrario, debe intercambiar u y v). Podemos ver que los nodos entre la primera aparición de u y la primera aparición de v son $E[H[u]...H[v]]$.
Ahora, debemos encontrar el nodo situado en el nivel más pequeño. Para esto, podemos usar RMQ. Entonces, $LCAT(u, v) = E[RMQ_L(H[u], H[v])]$ (recuerde que RMQ devuelve el índice). Así es como E, L y H encuentra el resultado:

LCA RMQ

$\mathrm{LCA_T}(10, 15) = E[12] = 3$

E[10...15]

E:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	2	1	3	5	3	6	8	6	9	6	3	7	10	12	10	13	10	7	11	7	3	1	4	1

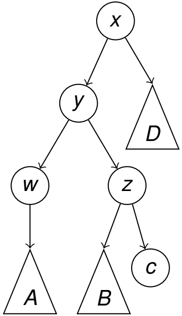
$\mathrm{RMQ_L}(10, 15) = 12$

L:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	0	1	0	1	2	1	2	3	2	3	2	1	2	3	4	3	4	3	2	3	2	1	0	1	0

H:	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2	4	24	5	7	13	8	10	14	20	15	17

R[9] = 10 R[12] = 15

Aclaración



$$L = \left[\begin{array}{ccccccccccccccc} x & y & w & \ldots & w & y & z & \ldots & z & c & z & y & \ldots & x \end{array} \right]$$

Contenidos

Distancias en árboles

- Queremos utilizar consultas de LCA para obtener una estructura capaz de computar rápidamente distancias entre nodos en un árbol.
- Notar que computar todas las distancias en un árbol utilizando un algoritmo como DFS o BFS n veces toma tiempo $O(n^2)$, que es lineal en la cantidad de distancias existentes.
- El algoritmo que propondremos por lo tanto solo constituirá una ventaja importante cuando se quieran consultar muchas menos que las n^2 distancias (pero suficiente cantidad como para que resolver cada una en forma independiente no sea práctico)

Distancias en árboles (Algoritmo)

- Tomamos un elemento cualquiera como raíz.
- Utilizamos DFS o BFS para recorrer el árbol computando las distancias desde la raíz hasta cada uno de los vértices.
- A partir de ahora, para resolver la distancia entre dos nodos cualesquiera i y j, utilizamos la identidad:

$$D(i,j) = D(r,i) + D(r,j) - 2D(r, LCA(i,j))$$

- El algoritmo propuesto responde una distancia con una complejidad de $O(1)$ más una consulta de LCA. La inicialización más allá del LCA es un único DFS o BFS, por lo que es $O(n)$.

Contenidos

Tarea

- http://acm.uva.es/p/v109/10938.html
- http://www.spoj.pl/problems/QTREE2/
- http://poj.org/problem?id=2763
- http://poj.org/problem?id=1986

Contenidos

Uso

Tenemos un arreglo $arr[0...n-1]$. Y queremos calcular

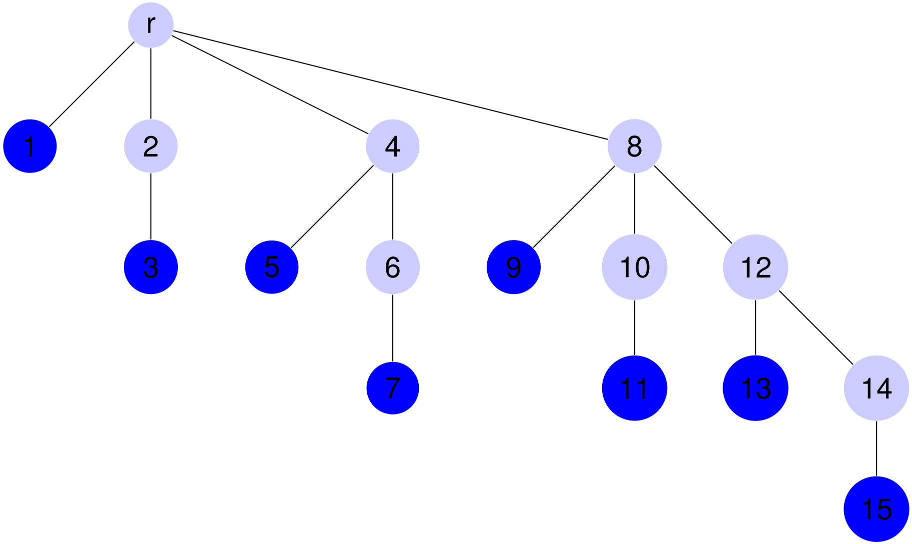
- La suma de los primeros i elementos.
- Modificar el valor de un elemento especificado del arreglo $arr[i] = x$ donde $0 <= i <= n-1$.

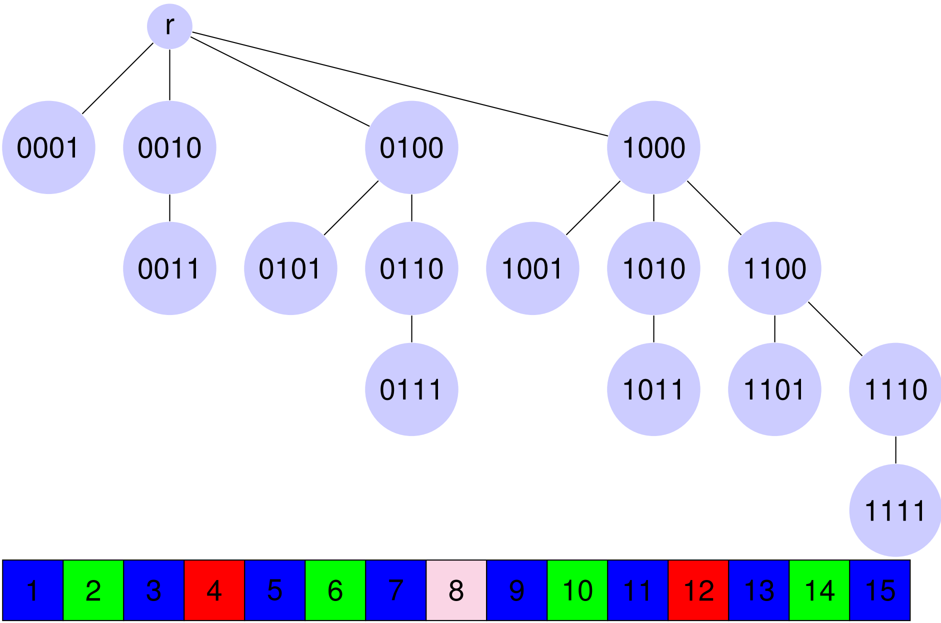
Descripción

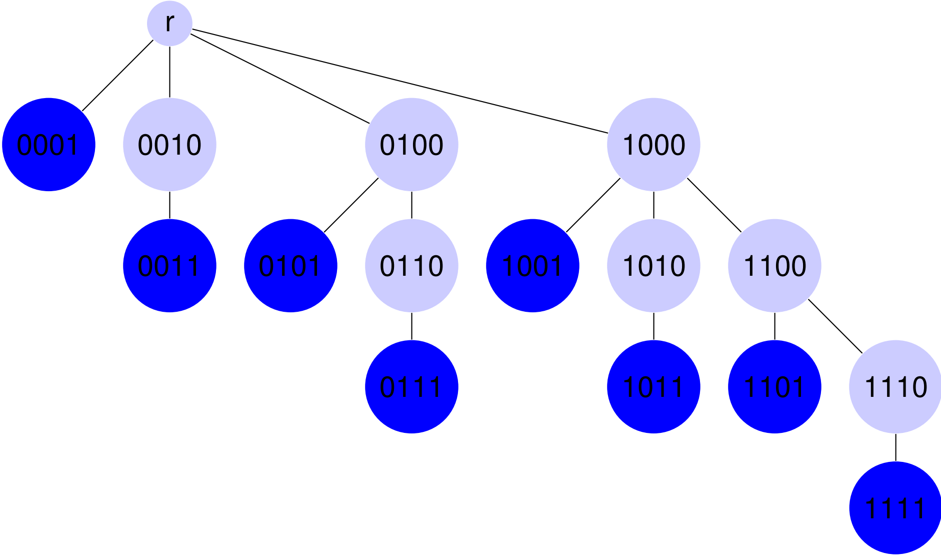
Fenwick Tree se representa como un arreglo $BITree[N]$.

- Cada nodo del árbol almacena la suma de segmentos del array.
- El tamaño de este árbol es igual al tamaño del array de entrada, denotado como N.
- Los intervalos involucrados para resolver el problema del acumulado hasta el valor i están dados por su representación binaria.

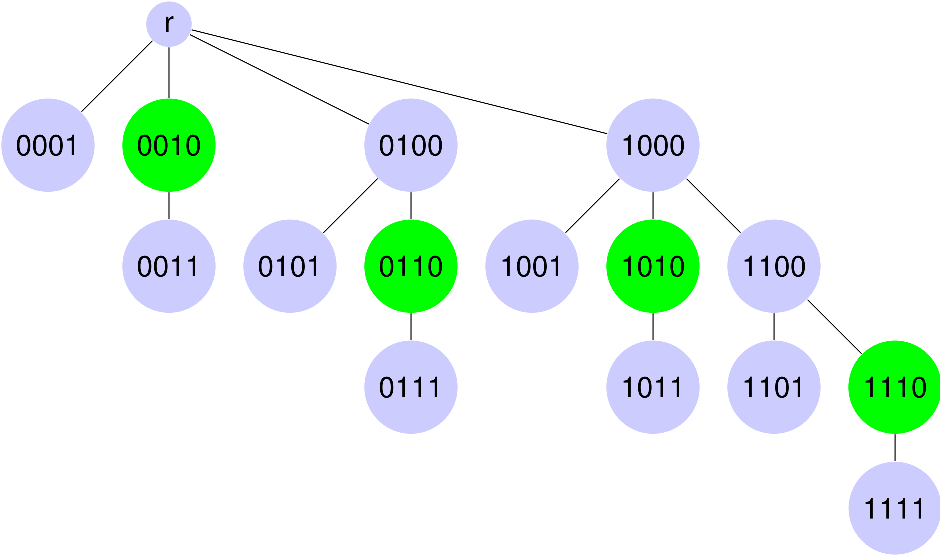
Contenidos


r
1
2
4
8
3
5
6
9
10
12
7
11
13
14
15

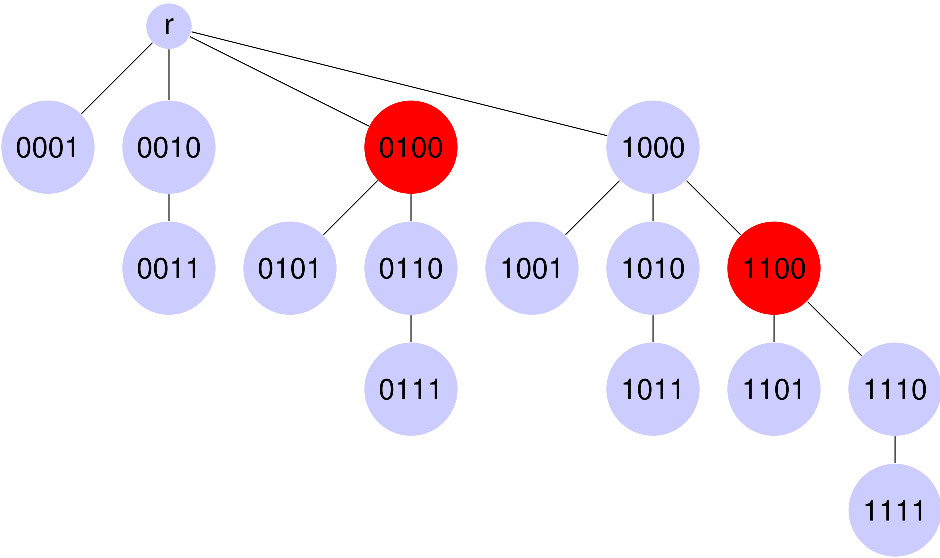

r
0001
0010
0100
1000
0011
0101
0110
1001
1010
1100
0111
1011
1101
1110
1111
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15



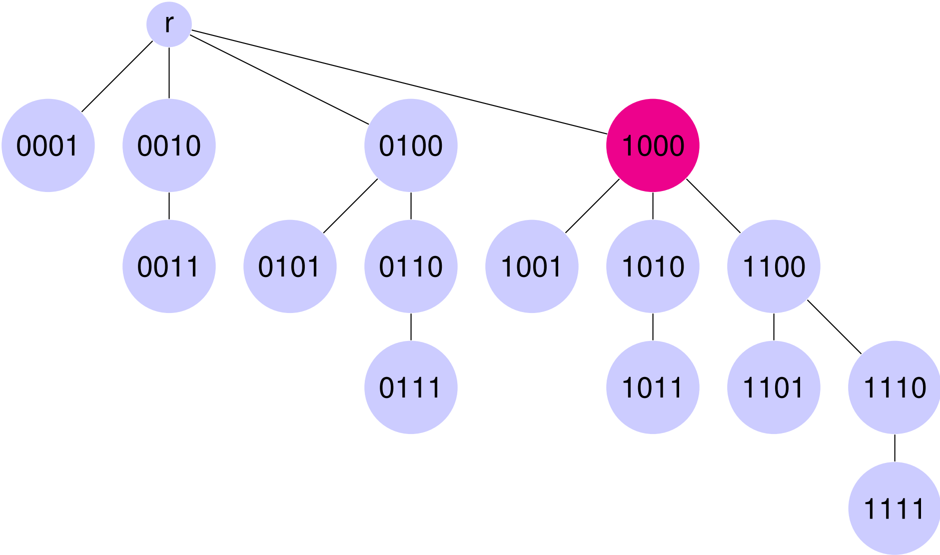
A_1	-	A_3	-	A_5	-	A_7	-	A_9	-	A_{11}	-	A_{13}	-	A_{15}



A_1	A_2	-	-	A_5	A_6	-	-	A_9	A_{10}	-	-	A_{13}	A_{14}	-

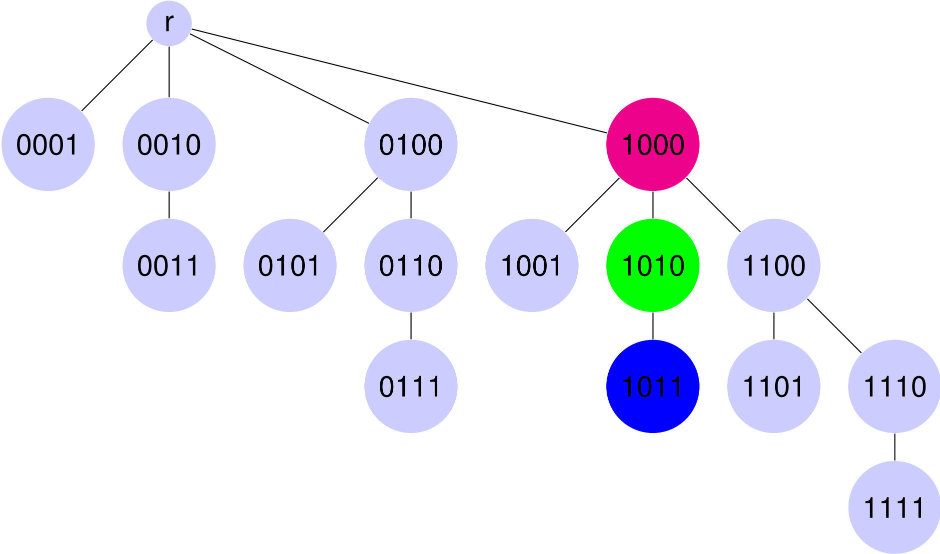


A_1	A_2	A_3	A_4	-	-	-	-	A_9	A_{10}	A_{11}	A_{12}	-	-	-



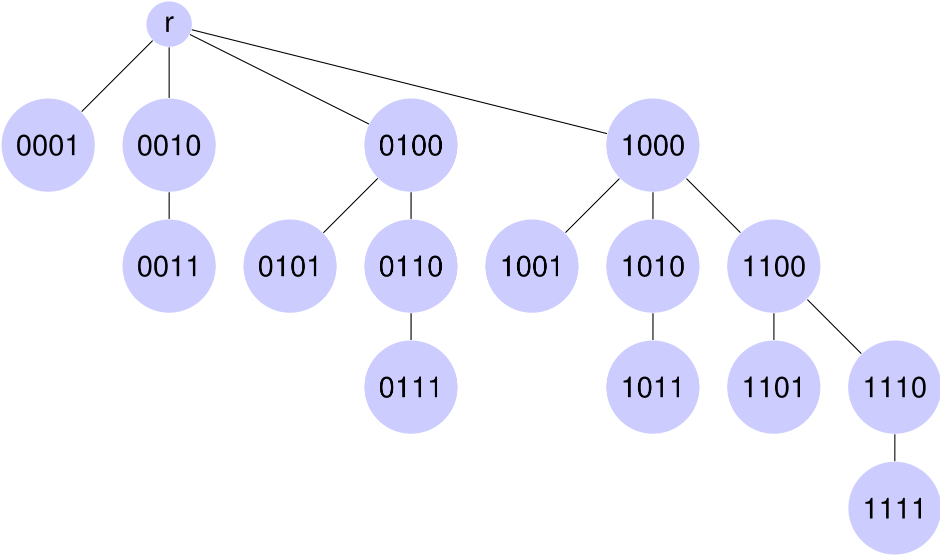
A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	-	-	-	-	-	-	-

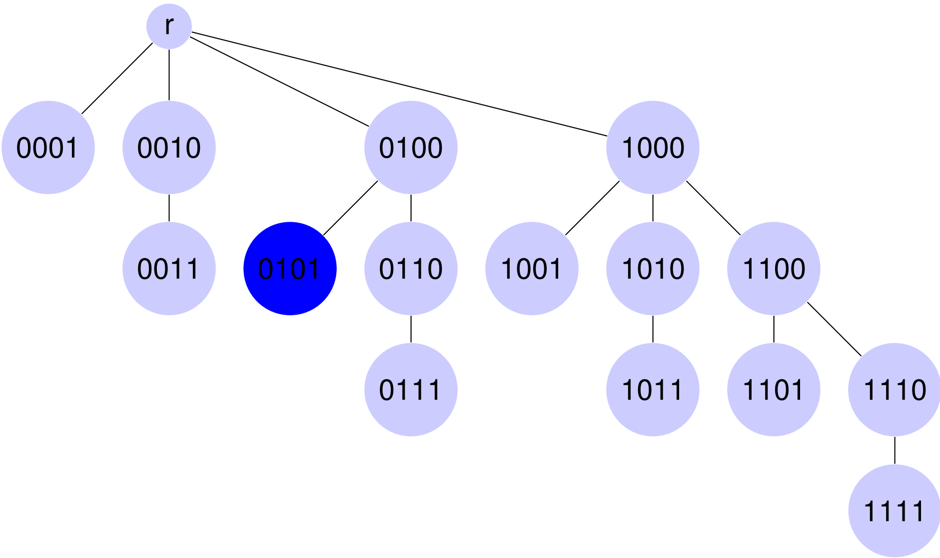
Contenidos



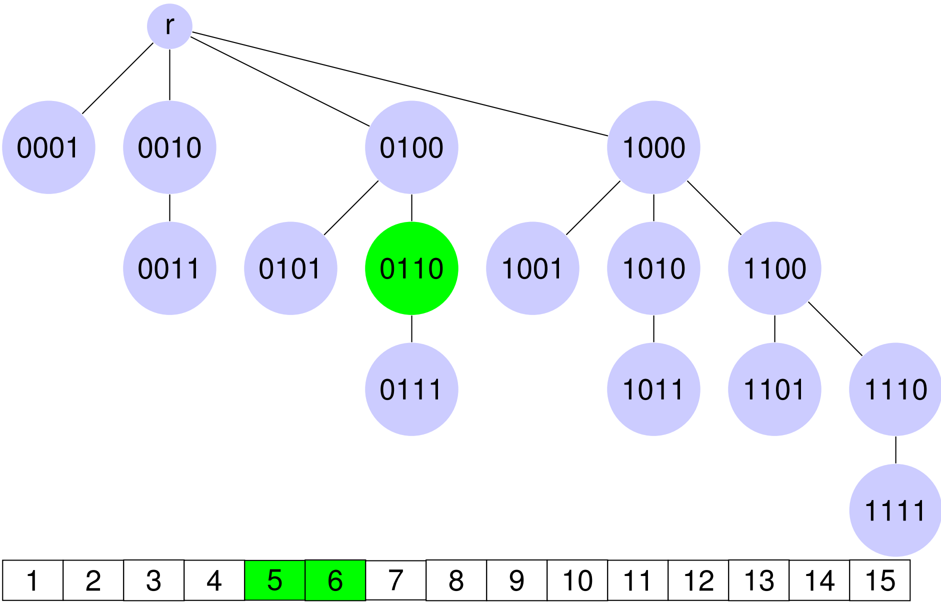
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

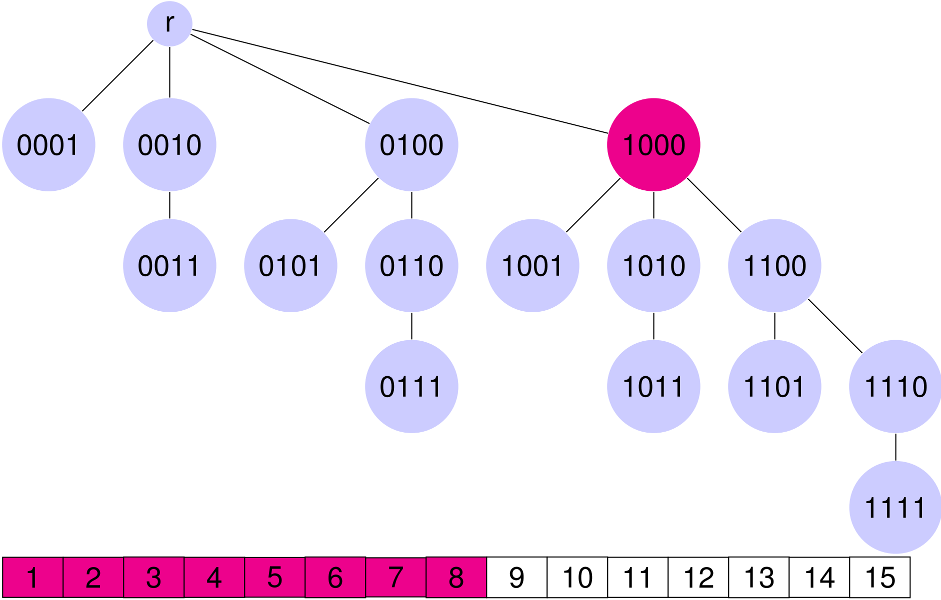
Contenidos


r
0001
0010
0100
1000
0011
0101
0110
1001
1010
1100
0111
1011
1101
1110
1111



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15


r
0001
0010
0100
1000
0011
0101
0110
1001
1010
1100
0111
1011
1101
1110
1111
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


r
0001
0010
0100
1000
0011
0101
0110
1001
1010
1100
0111
1011
1101
1110
1111
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Get value

```
int getSum(int BITree[], int index){
    int sum = 0;
    index = index + 1; // indexamos de 1
    while (index>0){
        sum += BITree[index];
        index -= index & (-index);
    }
    return sum;
}
void updateBIT(int BITree[], int n, int index, int val){
    index = index + 1; //indexamos de 1
    while (index <= n){
        BITree[index] += val;
        index += index & (-index);
    }
}
```

Initialize

```
int *constructBITree(int arr[], int n){
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    return BITree;
}
```

Magic operation

Para entender esa operación vamos a repasar como es el negativo a un número y el **bitwise and.**
Cuando se realiza un cambio de signo de un número *n* pasa lo siguiente: se cambia de valor todos sus bits por su opuesto y **se suma 1.** Ejemplo:

$$5 = 00101 \rightarrow \bar{5} = 11010 \rightarrow -5 = 11011 \rightarrow 00101 + 11011 = 0$$

$$10 = 01010 \rightarrow \bar{10} = 10101 \rightarrow -10 = 10110 \rightarrow 01010 + 10110 = 0$$

$$12 = 01100 \rightarrow \bar{12} = 10011 \rightarrow -12 = 10100 \rightarrow 01100 + 10100 = 0$$

Magic operation

Analizando un poco como funciona el negativo de un número podemos ver que el único bit que un número y su negativo coinciden es el primero de la derecha.

0	...	0	1	1	0	0
1	...	1	0	1	0	0
0	...	0	0	1	0	0

Cuadro: 12, -12 y (12 & -12)

Magic operation

Analizando un poco como funciona el negativo de un número podemos ver que el único bit que un número y su negativo coinciden es el primero de la derecha.

0	...	0	1	1	0	0
1	...	1	0	1	0	0
0	...	0	0	1	0	0

Cuadro: 12, -12 y (12 & -12)

$$x = x_0, ..., x_i = 1, 0, ..., 0$$

$$\bar{x} = \bar{x}_0, ..., \bar{x}_i = 0, 1, ..., 1$$

$$-x = \bar{x}_0, ..., x_i = 1, 0, ..., 0$$

Contenidos

Chiao

¡¡¡¡¡¡¡Éxito a todos en todo!!!!!!!!