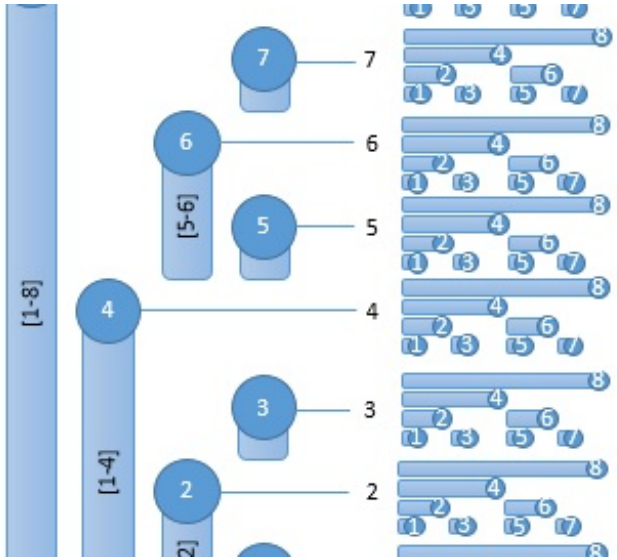
Archivo de la categoría: Estructuras de Datos



Árbol de Fenwick 2D

febrero 24, 2013 Estructuras de Datos arbol de fenwick, BIT

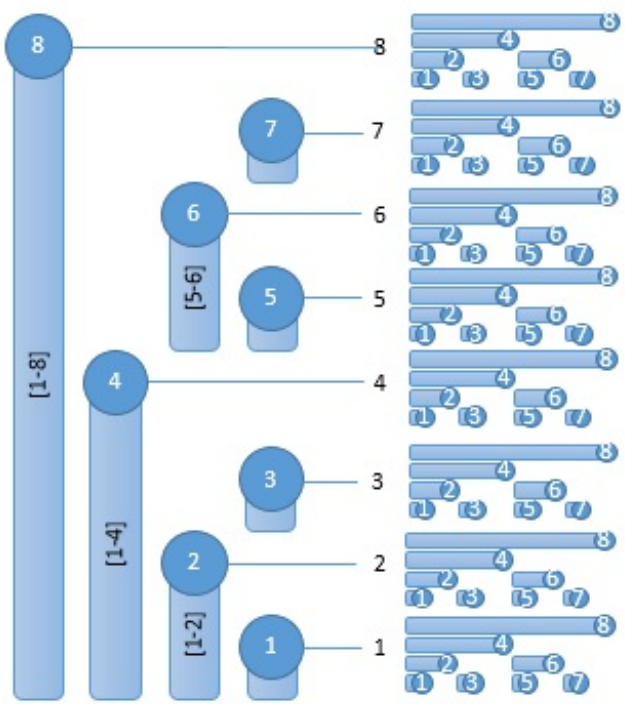
El **Árbol de Fenwick** (conocido también como **BIT**) es una estructura muy eficiente para calcular sumas en intervalos. En este post vamos a ver como podemos usarlo en un problema de dos dimensiones. El problema es el siguiente:

Supongamos que tenemos un terreno rectangular que representa un sembrado de tamaño $n \times m$. Podemos sembrar plantas en distintos puntos de este terreno, y estamos interesados en saber cuantas plantas hay en una porción dada del terreno.

Para resolver este problema, necesitamos definir 2 operaciones:

1. Consulta. Esta operación calcula, dada una porción del terreno, cuantas plantas hay dentro de esa porción.
2. Actualización. Esta operación agrega una o varias plantas en una posición dada del terreno.

Estas son operaciones similares a las del **BIT**, pero en este caso las vamos a realizar sobre un arreglo bidimensional. Esto lo podemos hacer creando un **BIT** donde cada elemento del árbol, sera a su vez un **BIT** convencional.



BIT 2D

Al crear un **BIT 2D**, podemos realizar las operaciones que necesitamos para resolver el problema. En este caso, cada elemento del **BIT** sera responsable por un intervalo de filas en el arreglo bidimensional, y contendrá otro **BIT** que sera responsable por un intervalo de columnas. Al crear un **BIT 2D** a partir del arreglo bidimensional, cada elemento (i, j) del arreglo almacenara la suma del rectángulo desde (0, 0) hasta (i, j).

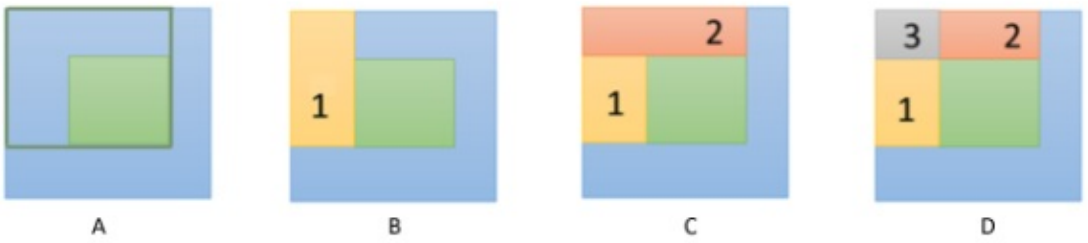
Consultando sumas acumuladas

Con este diseño podemos responder fácilmente la suma acumulada

en el rectángulo desde (0, 0) hasta (*i*, *j*). Similar al **BIT** convencional, vamos a recorrer los nodos responsables de las filas, y de cada nodo vamos a recorrer los responsables en el **BIT** que este contiene. Suponiendo que la información sea almacenada en el arreglo bidimensional **T**[i, j], la operación de consulta quedaría así:

```
1  public int sum(int i, int j)
2  {
3      int result = 0;
4      for(int r = i; r > 0; r-= lowbit(r))
5          for(int c = j; c > 0; c-= lowbit(c))
6              result+= T[r][c];
7      return result;
8  }
```

Esta operación es O(log n × log m), ya que recorremos los responsables en las filas, y para cada uno de ellos, los responsables en el **BIT** interno. Si queremos consultar la suma en un rectángulo dentro dentro del arreglo, tenemos que realizar otros cálculos.



Si deseamos calcular el valor dentro
del rectángulo verde, tendríamos que calcular la suma desde la posicion (0, 0) hasta la esquina inferior derecha de este rectángulo (A) y restarle los valores acumulados en los rectángulos 1 y 2 (B y

C); y sumarle el valor del rectángulo 3, ya que este esta comprendido dentro de 1 y 2 y lo restamos dos veces al quitar los valores de ellos (D).

```
public int sum(int i, int j, int i2, int j2)
{
    int a = sum(i2, j2) + sum(i-1, j-1);
    int b = sum(i2, j-1) + sum(i-1, j2);
    return a-b;
}
```

Cambiar los valores

Para esta operación tenemos que cambiar el valor en la posición que queremos, y luego recorrer todos los nodos responsables de la fila en que se encuentra la posición, y en cada BIT contenido en la fila, actualizar los valores en los elementos responsables.

```
public void update(int i, int j, int v)
{
    for(int r = i; r < n; r += lowbit(r))
        for(int c = j; c < m; c += lowbit(c))
            T[i][j] += v;
}
```

Esta operacion es O(log n × log m) y su analisis es similar a la operacion de consulta.

Con estas operaciones podemos resolver el problema planteado. De manera similar podemos extender el BIT a mas dimensiones, y asi resolver problemas mas complejos.

Felices codigos!

Referencias

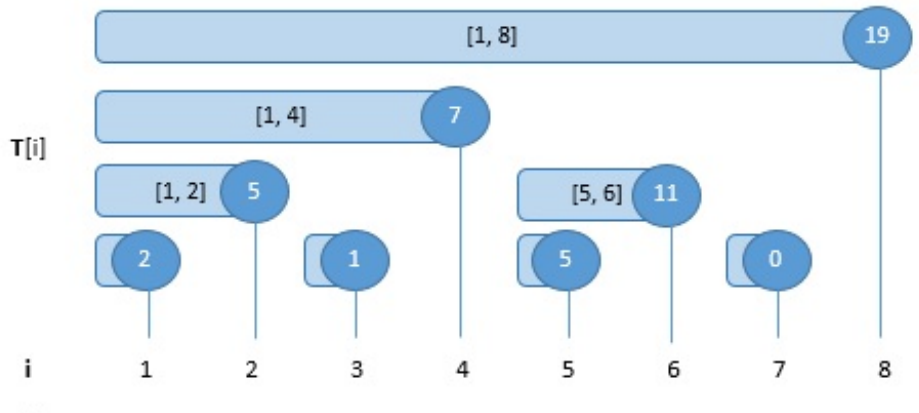
community.topcoder.com/tc?
module=Static&d1=tutorials&d2=binaryIndexedTrees

Problemas

http://coj.uci.cu/24h/problem.xhtml?abb=1904

http://www.spoj.com/problems/MATSUM/

Deja un comentario



A[i]	2	3	1	1	5	6	0	1

Árbol de Fenwick

febrero 17, 2013 Estructuras de Datos arbol de fenwick, BIT

El **Árbol de Fenwick** (también conocido como **Árbol Binario Indexado** o **BIT** por sus siglas en ingles) fue propuesto por *Peter M Fenwick* en 1994 para resolver problemas de compresión de datos. Es una estructura de datos muy eficiente para calcular sumas acumulativas. Para ilustrar el concepto de esta estructura, vamos a definir el siguiente problema:

Disponemos de *n* alcancias y varias monedas de distintos valores. Podemos insertar una moneda con cualquier valor en cualquiera de las alcancias, y estamos interesados en calcular la cantidad total de dinero que tenemos acumulada desde la alcancia 1 hasta la alcancia *i*.

Una forma de resolver este problema seria recorriendo cada alcancia desde 1 hasta i y sumar todos los valores para obtener la suma total, pero esta solucion es muy lenta cuando tenemos

muchas alcancias. El **BIT** permite solucionar este problema de forma mas eficiente. Para solucionarlo, definiremos un arbol donde habran nodos **responsables**, y nodos **dependientes**.

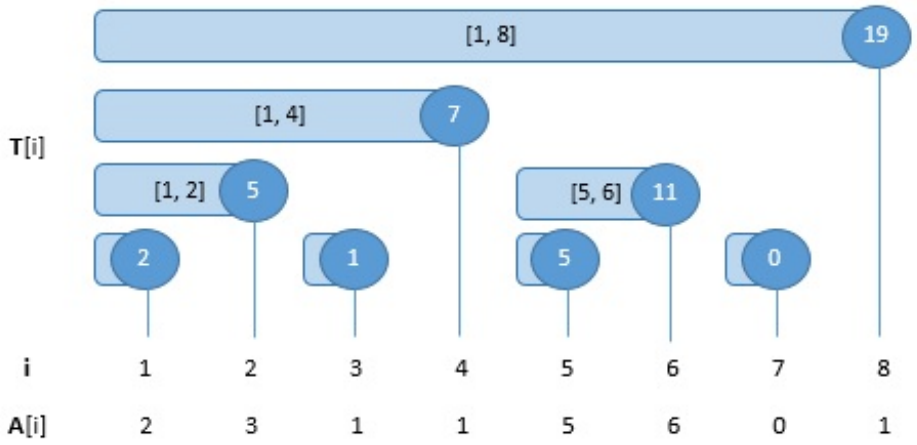
Cada nodo responsable se encarga de mantener la suma de los valores de todos sus nodos dependientes. De esta manera, para calcular una suma en cualquier intervalo solo hay que tomar los valores en los nodos responsables que estén en el intervalo, y sumarlos, ya que estos comprimen en ellos la información de sus nodos dependientes.

Como determinar si un nodo es responsable o dependiente? Para esto definimos la función lowbit(i), la cual nos dice dado un numero *i* cual es el menor bit que tiene valor 1 en la representación binaria de este numero. Por ejemplo, el numero 6 es 110 en binario (1×2^2 + 1×2^1 + 0×2^0), y su menor bit es 1×2^1 = 2. Esto se puede calcular de la siguiente forma:

```
1  public int lowbit(int i) { return (i & -i); }
```

Este árbol puede ser representado implícitamente asumiendo cada elemento original como un nodo en el árbol. Cada elemento *i* sera responsable por los elementos entre las posiciones *i*-lowbit(*i*)+1 hasta *i*, y almacenara el valor de la suma de los elementos en ese intervalo. Por ejemplo, lowbit(6)=2, por tanto, el elemento en la posición 6 es responsable por el elemento 5 y 6, y en el problema original, el valor seria la suma de la alcancia 5 + la suma de la alcancia 6. En el caso de 8, lowbit(8) = 8, y el elemento 8 seria responsable por los valores de las alcancias desde 1 hasta la 8.

Supongamos que los valores de las alcancías de la 1 hasta la 8 son [2, 3, 1, 1, 5, 6, 0, 1], respectivamente. En la siguiente figura podemos ver los elementos responsables y dependientes, y sus valores:



Árbol de Fenwick construido a partir de los valores

Obtener la suma acumulada

Dada una posición *i*, podemos usar el **árbol de Fewick** para obtener la suma de los valores acumulados desde la alcancía 1 hasta la alcancía *i*-esima. Esto es A[1] + A[2] + … + A[i]. La forma de hacer esto es comenzar en la posición *i*, y tomar los valores de los elementos responsables del árbol hasta llegar a 1.

NOTA: El **árbol de Fenwick** siempre comienza en la posición 1.

```java
public int sum(int i)
{
    int value = 0;
    for(; i > 0; i-= lowbit(i)) value+= T[i];
    return value;
}
```

De forma similar, es posible que necesitemos la suma de los valores acumulados de la alcancia i hasta la j (i < j). Esto es A[i] + A[i+1] + … + A[j]. Esto lo podemos calcular haciendo sum(j) – sum(i-1), ya que el resultado de esto seria:

A[1] + A[2] + … + A[i-1] + A[i] + A[i+1] + … + A[j] – (A[1] + A[2] + … + A[i-1]) = A[i] + A[i+1] + … + A[j].

```java
public int sum(int i, int j)
{
    return i > 1 ? sum(j) - sum(i-1) : sum(j);
}
```

Igualmente podríamos estar interesados en el valor actual en la alcancía *i*, o sea, no en la suma de los valores acumulados desde la alcancía 1 hasta la *i*-esima, sino solamente el valor en esa alcancía. Esto lo podemos calcular fácilmente como sum(i) – sum(i-1).

Actualizar las cantidades

En este caso queremos agregar una moneda con valor *v* a la *i*-esima alcancía. También podemos sacar monedas de la *i*-esima alcancía.

Para esto, cambiamos el valor en el *i*-esimo elemento del árbol y debemos notificar a todos los elementos que son responsables de el que ha ocurrido un cambio en el valor. Por ejemplo, si agregáramos una moneda en la alcancía 2, tendríamos que notificar a la 4 y a la 8 de este cambio ya que ambas son responsables por la 2. Esto mantiene el árbol sincronizado. La forma de hacerlo es actualizar el valor en la posición *i*, y moverse hacia adelante en el árbol a todos los responsables haciéndoles saber que el elemento fue actualizado.

```
1  public void update(int i, int value)
2  {
3      for(; i < T.length; i+= lowbit(i)) T[i] += valu
4  }
```

En caso de que estemos agregando una moneda con valor *v*, el valor de *value* sera positivo. Si estamos sacando una cantidad *v*, el valor sera negativo.

Construir el árbol

Ya hemos visto como hacer las operaciones fundamentales con el **árbol de Fenwick**, pero no hemos visto como construirlo! Precisamente el árbol se construye a partir de la operación **update**.

```
1  public void build(int[] A)
2  {
3      int[] T = new int[A.length + 1];
4      for(int i=0; i < A.length; i++) update(i+1, A[i
```

```
5   }
```

Y esto es todo, amigos. El **árbol de Fenwick** es una estructura muy eficiente, sin embargo sencilla. El concepto no es difícil de entender, y aun sin comprenderlo totalmente, con solo saber como utilizarla, podemos resolver muchísimos problemas. Ademas, se puede extender con facilidad a problemas de varias dimensiones, pero vamos a dedicar otro post completamente a ese tema.

Felices códigos!

Referencias

es.wikipedia.org/wiki/%C3%81rbol_binario_indexado

olds.blogcn.com/wp-content/uploads/67/6754/2009/01/bit.pdf

community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

comeoncodeon.wordpress.com/2009/09/17/binary-indexed-tree-bit/

www.algorithmist.com/index.php/Fenwick_tree

www.swageroo.com/wordpress/little-known-awesome-algorithms-fenwick-range-trees-rapidly-find-cumulative-frequency-sums/

gborah.wordpress.com/2011/09/24/bit-indexed-tree-fenwick-tree/

Problemas

spoj.pl/problems/INVCNT/

www.spoj.com/problems/YODANESS/

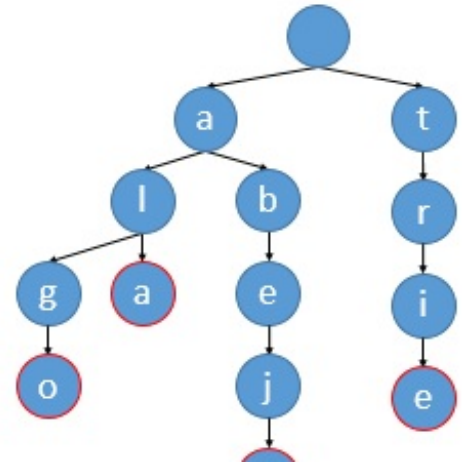
uva.onlinejudge.org/external/119/11926.pdf

uva.onlinejudge.org/external/115/11525.pdf

coj.uci.cu/24h/problem.xhtml?abb=1850

coj.uci.cu/24h/problem.xhtml?abb=2125

Deja un comentario



Trie – arbol de prefijos

febrero 13, 2013 Estructuras de Datos prefijos, string, trie

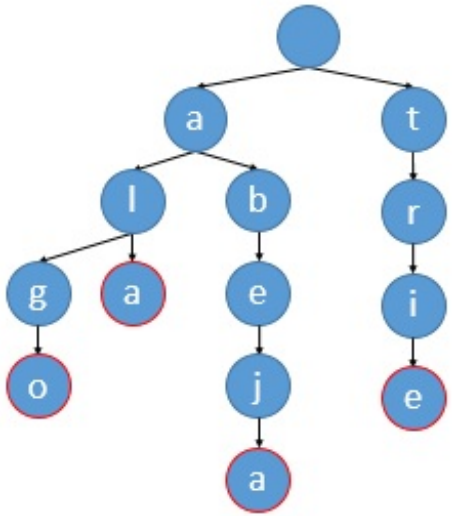
El **trie** (también conocido como **árbol de prefijos**) es una estructura de datos muy útil para resolver problemas asociados con **strings**. El problema mas conocido es dado una colección de palabras (algo como un diccionario) y una palabra en particular, saber si la palabra se encuentra en la colección de palabras.

Una primera solución podría ser comparar la palabra buscada a cada palabra, pero esta solucion es O(*n*) ya que se necesitarían al menos *n* comparaciones donde *n* es la cantidad de palabras en la colección. Cuando tenemos demasiadas palabras, esta solución es extremadamente lenta. Otra posible solución podría ser ordenar todas las palabras alfabéticamente y luego usar búsqueda binaria para encontrar la palabra. Esta solución es mejor, pero tiene la desventaja de que cada vez que introducimos una nueva palabra hay que reordenar la colección completa, lo que es O(*n* log *n*).

El **trie** resuelve este problema eficientemente, ya que tanto agregar como buscar una palabra es O(|s|) donde |s| es la cantidad de caracteres de la palabra s. Esta estructura fue introducida por por *Rene de la Briandais* y *Edward Fredking* en 1959. El nombre proviene de la palabra RE**TRIE**VAL, que significa recuperación. Internamente, funciona como

un árbol ordenado, donde cada nodo representa un carácter de una palabra insertada en el **trie**, y los nodos hijos de este son caracteres que aparecen después de el en la palabra.
La raíz del árbol es el caracter vacío, ya que se puede considerar que este aparece al principio de cualquier palabra.

El **trie** también es conocido como **árbol de prefijos** porque al insertar una palabra en el árbol se busca primero el mayor prefijo de esa palabra que ya este presente en el árbol, y a partir de ahí se continua insertando hasta donde termine la palabra. La figura a continuación muestra un **trie** construido con las palabras *algo*, *ala*, *abeja*, y *trie*.



Trie construido con las palabras. Los nodos terminales están resaltados.

Construyendo el trie

Para construir el **trie**, necesitamos almacenar en cada nodo lo siguiente:

- Letra que representa
- Si es o no la ultima letra de alguna palabra
- Nodo padre
- Nodos hijos

Adicionalmente, cada nodo implementa un método para saber si un nodo dado es hijo de ese nodo, y para agregar un nuevo hijo. Usando estos métodos se encuentra el mayor prefijo que ya este en el árbol hasta llegar a un nodo hoja, y se comienza a insertar a partir de ahí.

```
1  class Node
2  {
3      char content;
4      boolean ends;
5      Node parent;
6      Node[] child = new Node[26];
7
8      public Node(char c, Node pi)
9      {
10         parent = pi; content = c;
11     }
12
13     public Node getChild(char c) { return child[c-
14
15     public Node addChild(Node node) { child[node.c
16 }
```

En la linea 6 inicializamos el arreglo en 26 ya que ese es el tamaño del alfabeto ingles. Para un alfabeto mayor, hay que incrementar el tamaño del arreglo. El método **getChild** retornara el nodo hijo que contiene el caracter **c**, o **null** si no hay ninguno. Con esto podemos crear un **trie** cuya raíz sera el caracter vacío.

```
1  class Trie
2  {
3      Node root = new Node('\0', null);
4
5      public void insert(String word) { }
6
7      public boolean search(String word) { }
8  }
```

Insertar en el trie

El algoritmo para insertar una palabra en el **trie** es comenzar en la raíz y moverse por el árbol buscando el caracter siguiente en la palabra. Mientras se encuentre el caracter, actualizamos la posición en el árbol Cuando no se encuentre, insertamos a partir de esa posición Al llegar al final de la palabra, marcamos la posición en que estamos como terminal.

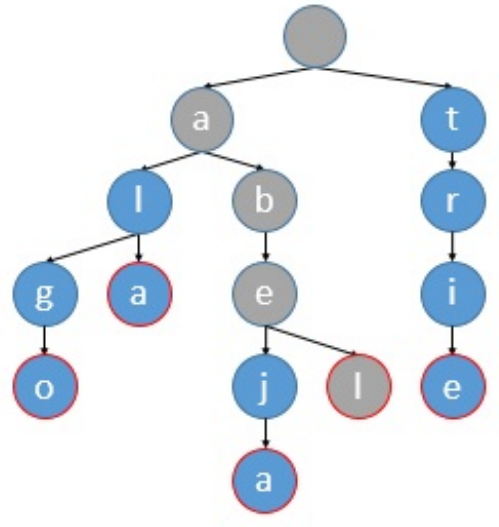
```
1  class Trie
2  {
3      Node root = new Node('\0', null);
4
5      public void insert(String word)
6      {
7          Node current = root;
8          for(int i=0; i < word.length(); i++)
9          {
```

```
10                  char c = word.charAt(i);
11                  Node sub = current.getChild(c);
12                  if (sub != null) current = sub;
13                  else
14                  {
15                      current.addChild(new Node(c, curre
16                      current = current.getChild(c);
17                  }
18                  if (i == word.length()-1) current.ends
19              }
20          }
21
22          public boolean search(String word) { }
23      }
```

Si insertáramos en el **trie** original la palabra *abel*, la secuencia de operaciones seria la siguiente:

1. Comenzamos en la raíz.
2. Buscamos el caracter 'a'. Aparece, por tanto nos movemos en el árbol hasta la 'a'.
3. Buscamos 'b'. Aparece, nos movemos a 'b'.
4. Buscamos 'e'. Aparece, nos movemos a 'e'.
5. Buscamos 'l'. No aparece, insertamos 'l' como hijo de 'e'. Nos movemos a 'l'.
6. Fin de la palabra, marcamos 'l' como terminal.

La imagen a continuación muestra el **trie** después de insertar la palabra y la ruta recorrida para insertarla.



Despues de insertar la palabra. Los nodos sombreados representan el camino.

Buscar en el trie

Buscar en el **trie** es similar a la operación de insertar. Comenzamos en la raíz y vamos buscando cada caracter de la palabra y actualizando la posición en el árbol Si en algún momento no encontramos un caracter, es porque la palabra no fue insertada. Similarmente, si el ultimo caracter no esta marcado como terminal, es porque esta palabra no aparece en el **trie**, sino aparece una palabra mayor que contiene a esta palabra como prefijo.

```
1  class Trie
2  {
3      Node root = new Node('\0', null);
4
```

```
5       public void insert(String word)
6       {
7           Node current = root;
8           for(int i=0; i < word.length(); i++)
9           {
10              char c = word.charAt(i);
11              Node sub = current.getChild(c);
12              if (sub != null) current = sub;
13              else
14              {
15                  current.addChild(new Node(c, curre
16                  current = current.getChild(c);
17              }
18              if (i == word.length()-1) current.ends
19          }
20      }
21
22      public boolean search(String word)
23      {
24          Node current = root;
25          for(int i=0; i < word.length(); i++)
26          {
27              char c = word.charAt(i);
28              Node sub = current.getChild(c);
29              if (sub == null) return false;
30          }
31          return current.ends;
32      }
33  }
```

El **trie** también puede ser usado para ordenar todas las palabras. Luego de insertarlas en el árbol un recorrido en pre-orden muestra cada palabra en orden ascendente, y en post-orden, en orden descendente.

Felices códigos!

Referencias

es.wikipedia.org/wiki/Trie

community.topcoder.com/tc?
module=Static&d1=tutorials&d2=usingTries

Problemas

codeforces.com/problemset/problem/271/D

www.spoj.com/problems/PRHYME/

Deja un comentario

Segment Trees III: Generalizacion

febrero 10, 2013 Estructuras de Datos segment tree

El **Segment Tree** es una estructura tan versátil que puede ser adaptado para resolver una gran variedad de problemas.
A continuación vamos a ver como puede extenderse para problemas mas complejos e incluso de varias dimensiones. Para comprender las descripciones de como resolver estos problemas es necesario tener un solido conocimiento de como funciona esta estructura, así que si aun no lo haz hecho, te recomiendo leer

primero los posts anteriores sobre el tema.

Mínimo/Máximo elemento

Estas funciones son similares, y en posts anteriores vimos como se pueden implementar.

Mínimo/Máximo y cantidad de veces que aparece.

En este caso tenemos que agregar en cada nodo una variable para almacenar la cantidad de veces que aparece el mínimo/ máximo valor. Y en las operaciones **merge** y **split** tener en cuenta esta variable.

Máximo Común Divisor (GCD) / Mínimo Común Múltiplo (LCM)

Esta es una generalización interesante y se obtiene de la misma forma vista hasta ahora, solo hay que llevar en cada nodo el GCD o LCM de los números en el intervalo del arreglo que corresponde.

Suma y producto

En este caso, cada nodo almacenara la suma de los valores en sus

hijos, el producto puede ser implementado de forma similar.

Máxima/Mínima suma prefija/sufija

Un prefijo consiste de los primeros *k* elementos de un intervalo, de forma análoga un sufijo son los últimos *k* elementos. La suma máxima prefija de un intervalo es el prefijo con la máxima suma. Por ejemplo, en el arreglo [1, -2, 3, -4] la suma máxima prefija es 2, que es la suma del prefijo [1, -2, 3], ya que ningún otro prefijo tiene mayor suma. Con el Segment Tree podemos encontrar la suma máxima prefija de un intervalo en el arreglo. Para esto en cada nodo colocaremos una variable para almacenar la máxima suma prefija del intervalo que representa, y otra con la suma total de ese intervalo (la suma de la suma de sus hijos). Para encontrar la máxima suma prefija en un nodo que no sea hoja, notamos que el prefijo de suma máxima termina dentro del intervalo del nodo izquierdo, o dentro del intervalo del nodo derecho. En el primer caso, tomamos el valor de la máxima suma prefija del nodo izquierdo. En el segundo caso, sumamos el valor de la suma total del nodo izquierdo con el valor de la máxima suma prefija del nodo derecho. El máximo de estos dos, es el que asumimos como la máxima suma prefija en el intervalo. El mismo concepto puede ser aplicado para calcular sumas máximas/mínimas de sufijos.

Consulta vertical

Este problema consiste en dado un punto x y una serie de intervalos $[l, r]$, determinar cuantos intervalos contienen al punto x. En este caso, formamos el Segment Tree con el tamaño del máximo r. Cada intervalo lo agregamos al árbol sumando 1 en el intervalo que representa. Del mismo modo podemos eliminar intervalos restando 1 al intervalo que representa. Finalmente, para responder las consultas, buscamos el nodo hoja que representa la coordenada x, y luego sumamos los valores en los nodos en el camino de ese nodo hasta la raíz.

Extensión a dos dimensiones

Los Segment Trees pueden ser extendidos a dos o mas dimensiones en una forma bastante natural. Dado una matriz, podríamos calcular el menor elemento en el rectángulo especificado. Para resolver este problema, podemos comenzar con un Segment Tree de una dimensión en el cual cada nodo hoja representa una fila de la matriz, y los demás nodos los intervalos contiguos de filas. Ahora en cada nodo en vez de almacenar variables sobre la información de los elementos, almacenaremos un Segment Tree de una dimensión en el cual cada nodo hoja representa la intersección de una columna con el conjunto de filas que representa el nodo que lo contiene. En este caso las operaciones serian de $O(\log n \times \log m)$

Y esto no es todo!

El Segment Tree puede ser generalizado a cualquier operación binaria asociativa. Esto significa que, tomando como ejemplo la multiplicación $(a \times b) \times c = a \times (b \times c)$. De esta manera podemos extender los Segment Tree para resolver muchísimos tipos de problemas en varias dimensiones.

Referencias

en.wikipedia.org/wiki/Segment_tree

wcipeg.com/wiki/Segment_tree

comeoncodeon.wordpress.com/2009/09/15/segment-trees/

letuskode.blogspot.com/2013/01/segtrees.html

p–np.blogspot.com/2011/07/segment-tree.html

www.algorithmist.com/index.php/Segmented_Trees

Problemas

www.spoj.com/problems/KGSS/

www.spoj.com/problems/ORDERSET/

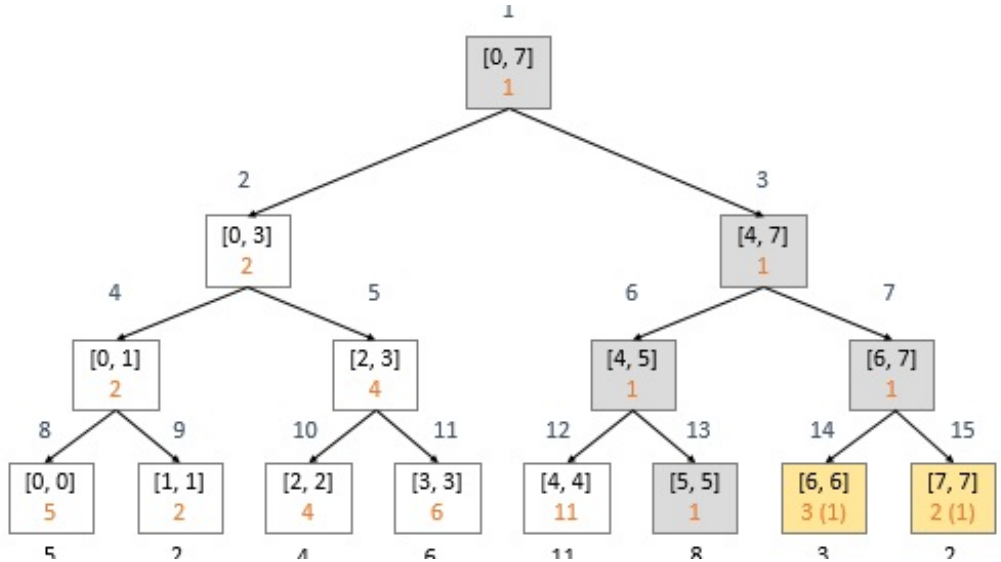
www.spoj.com/problems/GSS1/

www.spoj.com/problems/GSS2/

www.spoj.com/problems/GSS3/

www.spoj.com/problems/IOPC1207/

Deja un comentario



Segment Trees II: lazy updates

febrero 9, 2013 Estructuras de Datos lazy updates, segment tree

Cambiar todos los elementos en

un intervalo

Esta operación se puede realizar en O(log *n*) usando una técnica conocida como *lazy updates* (actualizaciones perezosas). Esta técnica básicamente actualiza solo los nodos que necesitamos en ese momento, y pospone las demás actualizaciones para el momento en que sean necesarias. Para esto utilizaremos la operación **split**.

Para llevar a cabo esta operación el intervalo que queremos actualizar lo marcaremos como pendiente, y en las operaciones de consulta es donde realmente se actualizara el intervalo, propagando las marcas de actualización de un nodo a sus hijos, y actualizando solamente el valor del nodo padre. De esta forma, solo actualizaremos en realidad los nodos que sean consultados, y los que no sean consultados, se quedarán marcados como pendientes. Para esto necesitaremos cambiar un poco la clase Node, y las operaciones de **update** y **query**.

```
1  class Node
2  {
3      boolean flag;
4      int left, right, value, updated;
5      public Node (int l, int r)
6      {
7          left = l; right = r;
8      }
9      public void merge (Node leftChild, Node rightC
10     {
11         if (leftChild == null) value = rightChild.
12         else if (rightChild == null) value = leftC
13         else value = Math.min(leftChild.value, rig
```

```
    }
    public void split(int nL, int nR)
    {
        if (left != right) // marcar los hijos
        {
            Node leftChild = tree[nL];
            leftChild.flag = true;
            leftChild.updated = updated;
            Node rightChild = tree[nR];
            rightChild.flag = true;
            rightChild.updated = updated;
        }
        flag = false;
        value = updated;
    }
}
```

Utilizaremos la variable **flag** para marcar los nodos como pendientes, y el método **split** para propagar las actualizaciones a los hijos de los nodos que han sido marcados. El método para actualizar el intervalo se encarga de marcar los nodos.

```
private void update(int node, int l, int r, int i,
{
    int leftChild = 2*node, rightChild = leftChild
    if(i <= l && j>= r)
    {
        tree[node].updated = v;
        tree[node].flag = true;
        tree[node].split(leftChild, rightChild);
    }
    else if (j < l || i > r) return;
    else
    {
        update(leftChild, l, mid, i, j, v);
        update(rightChild, mid+1, r, i, j, v);
        tree[node].merge(tree[leftChild], tree[rig
    }
```

```
17  }
18  public void update(int i, int j, int newValue)
19  {
20      update(1, 0, N-1, i, j, newValue);
21  }
```

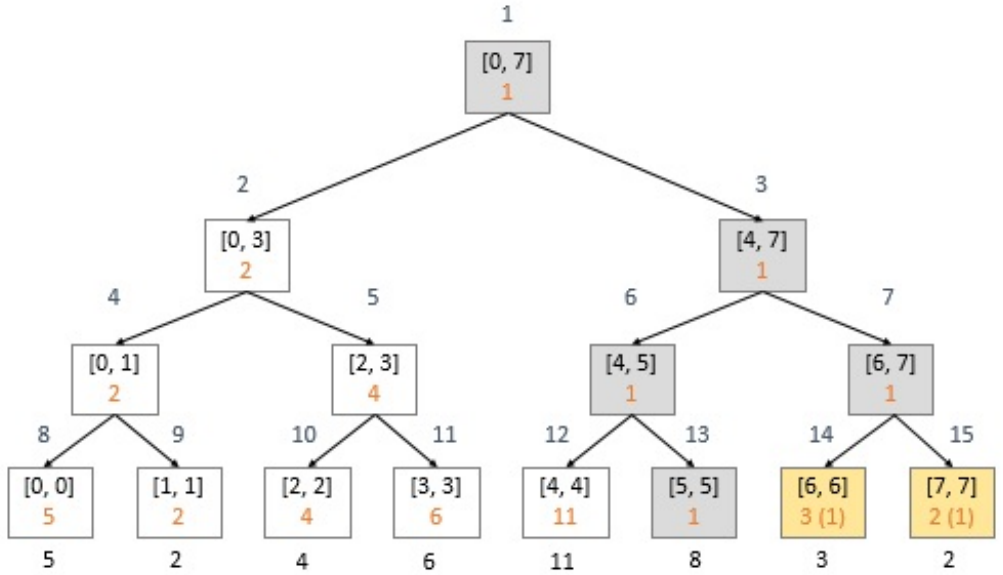
El método **query** es el que se encarga de propagar las actualizaciones por el árbol sencillamente usando la operación de **split** en cada nodo que sea visitado al ejecutar la consulta.

```
1   private Node query(int node, int l, int r, int i,
2   {
3       int leftChild = 2*node, rightChild = leftChild
4       if (l >= i && r <= j)
5       {
6           if (tree[node].flag) tree[node].split(left
7           return tree[node];
8       }
9       if (j < l || i > r) return null;
10      if (tree[node].flag) tree[node].split(leftChil
11      Node a = query(leftChild, l, mid, i, j);
12      Node b = query(rightChild, mid+1, r, i, j);
13      Node temp = new Node(N, N);
14      temp.merge(a, b);
15      return temp;
16  }
17  public int query(int i, int j)
18  {
19      Node result = query(1, 0, N-1, i, j);
20      return result.value;
21  }
```

De esta forma las operaciones de actualización son propagadas por el árbol Por ejemplo, si actualizáramos el intervalo [5, 7] a 1 en el arreglo original [5, 2, 4, 6, 11, 8, 3, 2] la secuencia de operaciones

seria la siguiente:

1. comenzamos en la raíz [0-7]
    1. hijo izquierdo [0-3] esta fuera del intervalo, retornar.
    2. hijo derecho [4-7] esta parcialmente dentro, visitar sus hijos.
        1. hijo izquierdo [4-5] esta parcialmente dentro, visitar sus hijos.
            1. hijo izquierdo [4-4] esta fuera, retornar.
            2. hijo derecho [5-5] esta completamente dentro. Marcar como pendiente y hacer **split**. Se actualiza su valor, y no se propaga porque no hay hijos.
            3. actualizar el valor de [4-5] usando **merge**. El valor ahora es 1.
        2. hijo derecho [6-7] esta completamente dentro. Marcar como pendiente y hacer **split**. Se marcan sus hijos como pendientes, y se actualiza el valor a 1.
        3. actualizar el valor de [4-7] usando **merge**. El valor ahora es 1.
    3. actualizar el valor de [0-7] usando **merge**. El valor ahora es 1.



Nodos modificados en la operación de actualización

En la figura se puede ver el resultado de la operación anterior. Los nodos marcados en gris fueron actualizados en la operación. Los nodos marcados en amarillo fueron marcados como pendientes y el valor entre paréntesis es el nuevo valor que tendrán después de actualizarlos. Dado que en la operación se recorre el camino de la raíz a un nodo, y luego de vuelta, la operación es O(log n). De esta forma podemos realizar de forma eficiente actualizaciones sobre un intervalo. En próximos post vamos a ver como puede ser generalizado el Segment Tree para resolver otros problemas mas complicados.

Referencias

en.wikipedia.org/wiki/Segment_tree

wcipeg.com/wiki/Segment_tree

comeoncodeon.wordpress.com/2009/09/15/segment-trees/

letuskode.blogspot.com/2013/01/segtrees.html

p–np.blogspot.com/2011/07/segment-tree.html

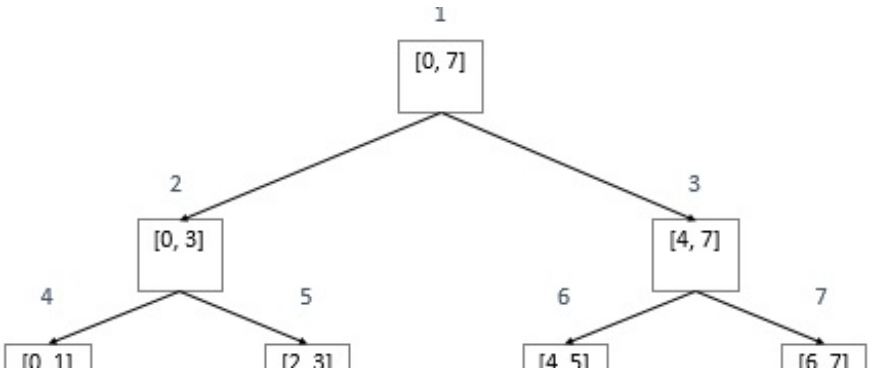
www.algorithmist.com/index.php/Segmented_Trees

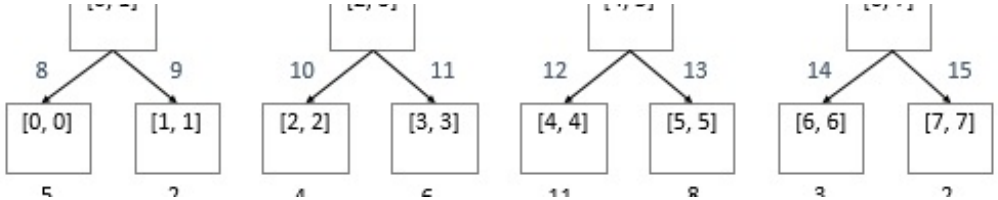
Problemas

www.spoj.com/problems/HORRIBLE/

www.spoj.com/problems/SEGSQRSS/

Deja un comentario





Segment Trees

febrero 8, 2013 Estructuras de Datos rmq, segment tree

El Segment Tree (o árbol de intervalos) es una estructura de datos que nos permite almacenar información en forma de intervalos, o segmentos. Fue descubierto por *J. L. Bentley* en 1977, y permite realizar las siguientes operaciones sobre la información de los intervalos:

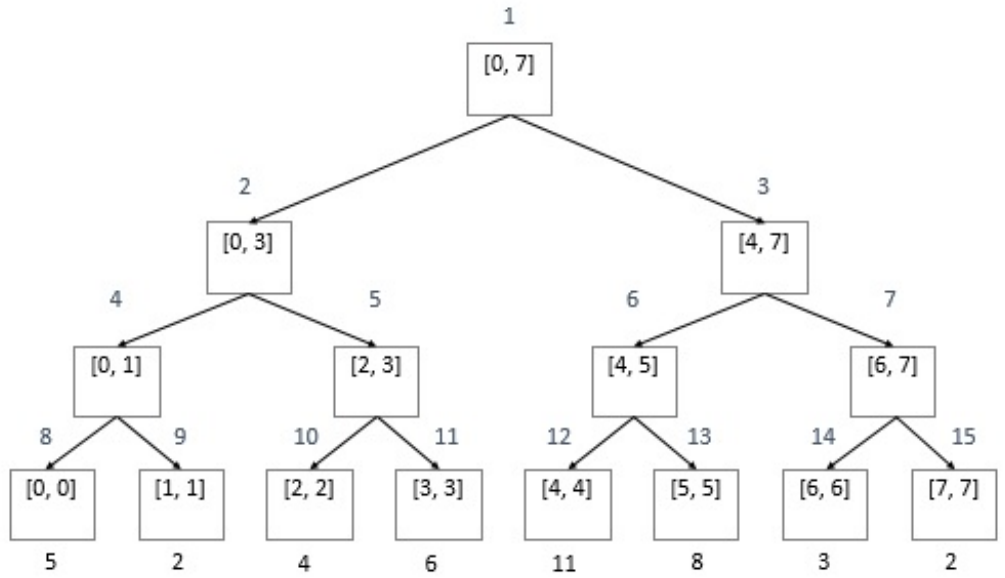
- Consultar
- Modificar un elemento
- Modificar todos los elementos en un intervalo

Para ver su uso, vamos a ver un problema que puede ser resuelto usando esta estructura.
Supongamos que tenemos el arreglo [5, 2, 4, 6, 11, 8, 3, 2], y tenemos varias operaciones de la forma:

- Determinar el menor elemento en el intervalo [i, j]
- Cambiar el elemento en la posición k

Este problema se puede resolver usando Segment Tree en O(log n) para cada operación Para esto vamos a definir una estructura en forma de árbol binario, donde en cada nodo vamos a llevar la información correspondiente al intervalo [l, r]. En el hijo izquierdo

de este nodo almacenaremos la información del intervalo [l, (l + r) / 2] y en el hijo derecho [(l + r) / 2 +1, r]. La raiz del arbol representa el intervalo completo.



Segment Tree correspondiente al arreglo

Construcción del árbol

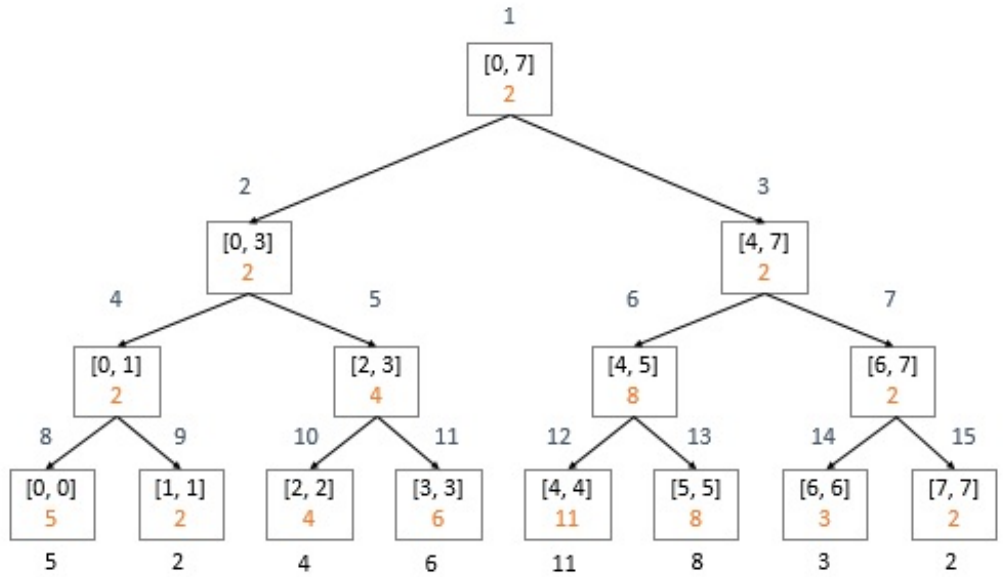
```
class Node
{
    int left, right, value;
    public Node(int l, int r)
    {
        left = l; right = r;
    }
    public void merge(Node leftChild, Node rightCh
    {
        if (leftChild == null) value = rightChild.
        else if (rightChild == null) value = leftC
```

```
12            else value = Math.min(leftChild.value, rig
13        }
14    }
```

El nodo contiene información sobre el intervalo que representa y el valor que tenemos en ese intervalo
como mínimo. La operación **merge** calcula el valor del nodo a partir de los valores de los hijos. En este caso, se almacena en cada nodo el menor valor que aparece en el intervalo que representa ese nodo. El árbol puede ser representado como un arreglo de Node donde cada posición *i* indica un nodo en el árbol y la posición 2*i* representa el hijo izquierdo de este nodo, y la posición 2*i*+1, el hijo derecho. Este arreglo tiene un tamaño proporcional al tamaño del arreglo original. No es difícil calcular que el mayor tamaño que necesitamos para guardar el árbol es 2^(log2(*n*)+1), donde *n* es la cantidad de elementos en el arreglo original. Luego vamos a construir el árbol de manera recursiva. La idea general es llegar primero hasta las hojas y luego regresar el camino hasta
la raíz calculando el valor de los nodos padres.

```
1   class SegmentTree
2   {
3       int N;
4       Node[] tree;
5       public SegmentTree(int[] A)
6       {
7           N = A.length;
8           int power = (int) Math.floor(Math.log(A.le
9           int len = (int) 2 * Math.pow(2, power);
10          tree = new Node[len];
11          build(1, 0, A.length-1, A);
12      }
13      private void build(int node, int l, int r, int
```

```
14      {
15          if (l == r) // nodo hoja
16          {
17              tree[node] = new Node(l, r);
18              tree[node].value = A[l];
19              return;
20          }
21          int leftChild = node*2, rightChild = leftC
22          build(leftChild, l, mid, A);       // calcu
23          build(rightChild, mid + 1, r, A); // calcu
24          tree[node] = new Node(l, r);
25          tree[node].merge(tree[leftChild], tree[rig
26      }
27  }
```



Segment Tree construido a partir del arreglo

De esta manera, el árbol queda construido en O(n) y podemos comenzar a responder las consultas sobre el menor elemento en un intervalo.

Consultando los intervalos

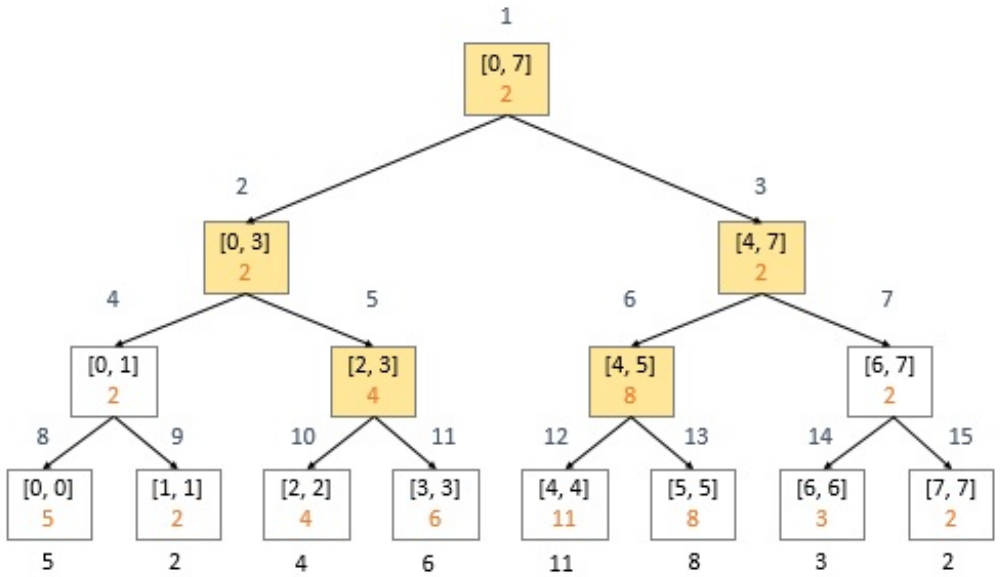
La operación **query** la podemos responder con recorrer el camino de la raíz hacia las hojas. Si nos preguntan por el menor elemento en el intervalo [2, 5] visitaremos en el árbol solamente los nodos cuyos intervalos contengan total o parcialmente este intervalo. La secuencia de operaciones es la siguiente:

1. Comenzamos en la raíz [0, 7]. Como contiene al intervalo, vamos a visitar sus hijos recursivamente.
    1. Visitamos [0, 3]. Como contiene parte del intervalo, visitamos sus hijos recursivamente.
        1. [0,1] no esta en el intervalo, retornamos.
        2. [2, 3] esta incluido completamente en el intervalo de consulta, retornamos su valor.
    2. [4, 7]. Contiene parte del intervalo, visitamos sus hijos.
        1. [4, 5] esta contenido completamente en el intervalo de consulta, retornamos su valor.
        2. [6, 7] no esta dentro del intervalo, retornamos.

Combinando la información de todos los nodos visitados tendremos el resultado. Siguiendo esta estrategia,
el método **query** quedaría de la siguiente manera:

```
1  private Node query(int node, int l, int r, int i, 
2  {
3      if (l >= i && r <= j) return tree[node]; // de
4      else if (j < l || i > r) return null;  // fuer
5      else // parcialmente dentro
6      {
```

```
7           int leftChild = 2*node, rightChild = leftC
8           Node a = query(leftChild, l, mid, i, j);
9           Node b = query(rightChild, mid+1, r, i, j)
10          Node temp = new Node(-1, -1); // combinamo
11          temp.merge(a, b);
12          return temp;
13      }
14  }
15  public int query(int i, int j)
16  {
17      Node result = query(1, 0, N-1, i, j);
18      return result.value;
19  }
```



Nodos visitados en la operacion de consulta

Cambiar un elemento

La operación de cambiar un elemento en el arreglo la podemos

realizar recorriendo el camino desde la raíz hasta el nodo hoja que corresponde al elemento que queremos cambiar, y luego el camino de regreso hacia la raíz actualizando los valores de todos los padres usando la operación **merge**.

```
1  private void update(int node, int l, int r, int i,
2  {
3      if (l == i && l == r) tree[node].value = v; //
4      else if (i < l || i > r) return;          // fue
5      else // parcialmente dentro
6      {
7          int leftChild = 2*node, rightChild = leftC
8          update(leftChild, l, mid, i, v);     // vis
9          update(rightChild, mid+1, r, i, v); // vis
10         tree[node].merge(tree[leftChild], tree[rig
11     }
12 }
13 public void update(int i, int newValue)
14 {
15     update(1, 0, N-1, i, newValue);
16 }
```

El Segment Tree es una estructura muy versátil y se puede generalizar fácilmente para soportar distintas operaciones.
En próximos posts veremos como modificarlo para soportar operaciones de actualización en intervalos y para resolver otros problemas mas complejos.

Referencias

en.wikipedia.org/wiki/Segment_tree

wcipeg.com/wiki/Segment_tree

comeoncodeon.wordpress.com/2009/09/15/segment-trees/

letuskode.blogspot.com/2013/01/segtrees.html

p-np.blogspot.com/2011/07/segment-tree.html

www.algorithmist.com/index.php/Segmented_Trees

Problemas

www.spoj.com/problems/BRCKTS/

www.spoj.com/problems/DQUERY/

www.spoj.com/problems/FREQUENT/

Deja un comentario

Buscar …

arbol de

fenwick biseccion BIT boyer-moore búsqueda binaria codeforces combinatoria criba lazy updates metodos numericos numeros primos prefijos rmq segment tree SPOJ string string matching trie UVa