Año 2025

Pertenencia sobre algunas representaciones computacionales de conjuntos

Introducción

Como ya dijimos, la operación más sencilla que podemos imaginar sobre un conjunto es la pregunta sobre la *pertenencia* de un elemento a ese conjunto. La respuesta a esta pregunta es de naturaleza booleana, un valor de verdad, un sí o un no, dependiendo de si el elemento pertenece o no al conjunto.

Ya aclaramos también que si procesamos información ésta estará representada por un conjunto definido por extensión, al cual alojaremos en un conjunto de celdas de la memoria de una computadora, el cual sí puede venir definido de diferentes maneras. En este conjunto de información, sólo podremos responder que un elemento pertenece al conjunto si lo hemos localizado en el conjunto de celdas en que está representado el conjunto. La localización aparece entonces como un subproducto no buscado, sólo queríamos saber la pertenencia y no dónde estaba el elemento, pero veremos que ésta será tan importante como saber la pertenencia.

Si no poseemos ninguna información adicional fuera de la descripción de las celdas utilizadas para guardar el conjunto, la única acción posible será examinar celda tras celda. Podríamos recorrer las celdas de acuerdo a distintos órdenes de búsqueda, pero no nos daría ninguna ventaja; tanto podemos encontrar antes lo buscado como postergar su encuentro debido a este cambio de orden. Por lo tanto, lo más razonable es recorrer el conjunto de celdas en un orden que resulte sencillo de programar.

Analizaremos a continuación como se resolvería la pertenencia sobre distintas representaciones computacionales posibles para un conjunto que contenga información, es decir cómo se resolvería la pertenencia para distintas maneras posibles de organizar las celdas de memoria que representan al conjunto.

Listas no ordenadas

Las listas son la manera más simple de organizar un conjunto de celdas de memoria para alojar un conjunto. A su vez las listas no ordenadas corresponden a la forma más sencilla de almacenar los elementos del conjunto, dado que los almacenamos en cualquier orden.

Detallaremos a continuación los tipos más comunes de listas no ordenadas o desordenadas.

Lista de alojamiento secuencial de largo conocido (sin orden)

Tenemos al conjunto almacenado en un trozo de vector del cual conocemos cuál es el primer elemento y cuál el último. Como no existe relación entre la posición de la celda que le corresponde a un dato y el dato en sí, decimos que esta representación es una *lista secuencial desordenada de tamaño conocido*.

Si por ejemplo tuviéramos la siguiente lista:

V:

25	***158***	***77***	***93***	***64***	***51***	***234***	***171***	***258***
41	***42***	***43***	***44***	***45***	***46***	***47***	***48***	***49***

El siguiente código resolvería la pregunta de pertenencia de un valor x a este conjunto.

```
function pertenece (x : integer): boolean;
  var i: byte;
  begin
1.    i := 41;
2.    while (i<50) and (V[i]<>x) do
3.      i := i + 1;
4.    return (i <50);
  end;
```

En muchos compiladores el modo de evaluar el *and* es una opción. En este caso debería ser en modalidad de evaluación condicional del segundo operando. De no ser así se podría querer examinar una celda que no pertenece al trozo de vector en el que estamos interesados, más aún podría no haber elementos adicionales en el vector y en algunas arquitecturas de máquina invadir otra partición o un lugar de memoria inexistente con el consiguiente error de ejecución.

La situación inicial es que no conocemos los contenidos de las celdas y luego de recorrer por primera vez el control del ciclo pasamos a la situación en que podemos examinar el contenido de la celda con subíndice 41. Si 25 fuese el valor buscado, podríamos concluir positivamente la pertenencia y abandonaríamos el ciclo con $i = 41$. Si 25 no fuese el valor buscado, avanzaríamos i, lo que equivale a la situación:

V:

25	***158***	***77***	***93***	***64***	***51***	***234***	***171***	***258***
~~***41***~~	***42***	***43***	***44***	***45***	***46***	***47***	***48***	***49***

El subíndice 41 ya no describe más una celda a examinar. De ahora en más la pertenencia o no al conjunto depende de lo que encontremos en las demás celdas. Esto repite la situación inicial, lo que justifica el carácter iterativo del método.

Después de algunas iteraciones y suponiendo que x valga 200 tendríamos:

V:

25	***158***	***77***	***93***	***64***	***51***	***234***	***171***	***258***
~~***41***~~	~~***42***~~	~~***43***~~	~~***44***~~	~~***45***~~	~~***46***~~	~~***47***~~	***48***	***49***

¿Este proceso finalizará? Sí, porque la finalización queda asegurada debido a que en cada paso el conjunto a examinar tiene menos elementos que el del paso anterior y en un número finito de pasos llegaremos a un conjunto vacío, si el elemento no pertenece al conjunto. En un conjunto vacío podemos asegurar la *no pertenencia*, la situación final entonces sería:

V:

25	***158***	***77***	***93***	***64***	***51***	***234***	***171***	***258***
~~***41***~~	~~***42***~~	~~***43***~~	~~***44***~~	~~***45***~~	~~***46***~~	~~***47***~~	~~***48***~~	~~***49***~~

Entonces, la decisión de pertenencia se resuelve preguntando si el ciclo se cortó porque el conjunto a examinar es vacío, en este caso $(i \geq 50)$ (aunque el único valor posible en este caso es $i = 50$).

Otra manera de razonar el programa anterior es recursivamente, lo que por ejemplo sería:

$$x \in A \Leftrightarrow \begin{cases} \text{Si} \quad A \neq \emptyset \text{ entonces} \quad x = z \ \vee \ x \in A - \{z\} \\ \text{Si} \quad A = \emptyset \text{ entonces} \quad \bot \end{cases}$$

siendo z un elemento cualquiera de A.

Lista de alojamiento secuencial con terminación dada por contenido (sin orden)

Tenemos al conjunto almacenado en un trozo de vector del cual conocemos cuál es el primer elemento, pero no cuál es el último. En cambio, sabemos que si encontramos en la celda el valor $\star$ hemos alcanzado una celda que no contiene ningún elemento del conjunto, o sea el fin de la lista. Nuevamente, como no existe relación entre la posición de la celda que le corresponde a un dato y el dato en sí, decimos que esta representación es una *lista secuencial desordenada con terminación dada por contenido*.

Si por ejemplo tuviéramos la siguiente lista:

V:

25	***158***	***77***	***93***	***64***	***51***	***234***	***171***	***258***	★		★
41	***42***	***43***	***44***	***45***	***46***	***47***	***48***	***49***			

Como sólo cambia, respecto de la lista secuencial desordenada de tamaño conocido, que no conocemos de antemano cuál es la última celda del conjunto, el código para resolver la pertenencia de un valor x al conjunto difiere sólo en cómo preguntamos por haber llegado al conjunto vacío.

El siguiente código resolvería la pregunta de pertenencia de un valor x a este conjunto.

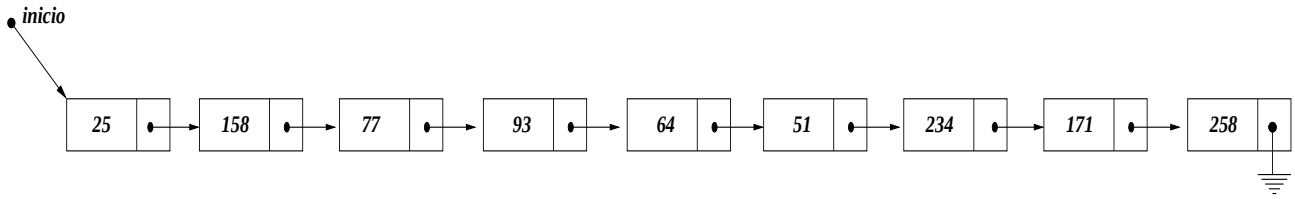
```
function pertenece (x : integer): boolean;
  var i: byte;
  begin
1.    i := 41;
2.    while (V[i]<> ⋆) and (V[i]<>x) do
3.      i := i + 1;
4.    return (V[i]<> ⋆);
  end;
```

Lista vinculada (sin orden)

Ya vimos que en este tipo de descripción del conjunto de celdas se comparten características con la descripción por recurrencia y con la descripción por extensión.

Tenemos al conjunto almacenado en un conjunto de celdas del cual en comienzo sólo conocemos la ubicación de la primera de ellas, en realidad de una desde la cual se pueden ir recuperando las posiciones de las demás. Como en este caso tampoco existe relación entre la posición de la celda que le corresponde a un dato y el dato en sí, decimos que esta representación es una *lista vinculada desordenada*.

Si por ejemplo tuviéramos la siguiente lista:



Las celdas que representan al conjunto serían de tipo *"nodo"* y tendrían un campo *"siguiente"* (de tipo puntero a nodo: ↑*nodo*) y un campo *"valor"* (del mismo tipo que x, o sea que en este caso serían enteros). El valor **nil** describe un valor inválido para la dirección de una celda. La variable *inicio* es un apuntador que contiene la dirección de la primera celda de la lista vinculada y p es un puntero a celdas de tipo *nodo*.

Cuando en una celda le colocamos en su campo *siguiente* el valor **nil**, estamos indicando que es la última celda de la lista, es decir que a esa celda no la sucede ninguna otra.

El siguiente código resolvería la pregunta de pertenencia de un valor x a este conjunto.

```
function pertenece (x : integer): boolean;
  var p: ↑nodo;
  begin
1.    p := inicio;
2.    while (p <>nil) and (p↑.valor<>x) do
3.      p := p↑.siguiente;
4.    return (p <>nil);
  end;
```

Como hemos mencionado en el apunte anterior y comparando los códigos vistos hasta ahora, podemos ver que al cambiar la forma de describir el conjunto de celdas no cambia la estructura básica del programa. Sólo cambiarán:

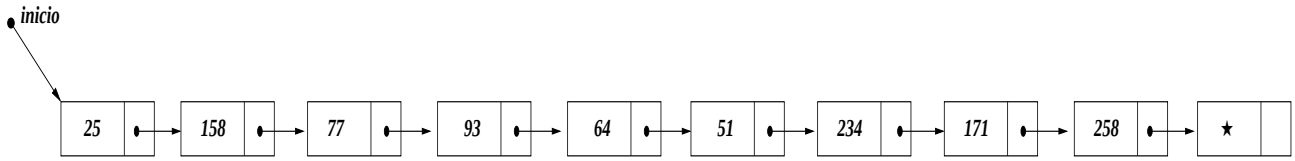
- La naturaleza (tipo) de los elementos que describen la celda a examinar.
- El modo de decir cuál es el primer valor.
- La forma de preguntar por conjunto remanente vacío.
- La forma sintáctica de referirse a los contenidos de las celdas.
- La manera de pasar de una celda a la otra.

Este hecho es muy importante, ya que da una idea de la forma general de recorrer un conjunto almacenado en un conjunto de celdas.

Lista vinculada con terminación dada por contenido (sin orden)

Como vimos, esta descripción del conjunto de celdas indica que acabó el conjunto colocando una marca de fin $\star$ en lugar de un dato del conjunto.

Por ejemplo, podríamos tener la siguiente lista:



En este caso, como ya vimos, cuando una celda contiene en su campo *valor* el valor $\star$, estamos indicando que esta celda ya no pertenece al conjunto.

El siguiente código resolvería la pregunta de pertenencia de un valor x a este conjunto.

```
function pertenece (x : integer): boolean;
  var p: ↑nodo;
  begin
1.    p := inicio;
2.    while (p↑.valor <> ⋆) and (p↑.valor<>x) do
3.      p := p↑.siguiente;
4.    return (p↑.valor <> ⋆);
  end;
```

Listas ordenadas con examinación secuencial

Si en una lista se conserva la manera de pasar de una celda a la otra, se puede decir que las celdas tienen un orden que es independiente de los contenidos de las mismas, tal cual lo veníamos haciendo. En cambio, si los contenidos también admiten un orden y los depositamos convenientemente en las celdas se puede conseguir que al recorrerlas los contenidos sean monótonos crecientes (o decrecientes). En tales circunstancias se dice que la lista está *ordenada*.

Cuando se hace una examinación secuencial de las celdas, como las programadas hasta ahora, se puede decidir que un elemento no está en el conjunto cuando se ve uno mayor que el buscado. Un sencillo razonamiento por propiedad transitiva de la desigualdad permite razonar que todos los elementos siguientes serán mayores que el buscado y por lo tanto distintos de él.

Entonces, sabiendo que existe un orden en los contenidos de las celdas lo podemos aprovechar haciendo lo siguiente: *si aún no hemos encontrado al elemento buscado y sabemos que el contenido de la celda que estamos examinando es mayor que el elemento buscado, podemos asegurar que el elemento no va a pertenecer al conjunto*. Podemos afirmar esto debido a que los elementos que podríamos encontrar si siguiéramos examinando serían aún mayores (debido al orden monótono creciente de los contenidos de las celdas).

Lista de alojamiento secuencial ordenada

Si por ejemplo tuviéramos la siguiente lista:

V:

25	***51***	***64***	***77***	***93***	***158***	***171***	***234***	***258***
41	***42***	***43***	***44***	***45***	***46***	***47***	***48***	***49***

Sigue siendo importante aquí no intentar examinar una celda que no corresponde al vector.

La pregunta de pertenencia se debe resolver ahora no sólo sabiendo que estamos dentro del vector sino que hay que preguntarse también si el elemento en el que se detuvo la búsqueda es el buscado. Porque ya observamos que podemos cortar el análisis de los contenidos del vector cuando encontramos un contenido que nos permite hacerlo y sin embargo tener que responder que el elemento buscado no pertenece al conjunto.

Para que la función responda verdadero se debe cumplir que estamos dentro del vector y que el contenido de la celda en la que se detuvo la iteración sea igual al elemento buscado.

Así, el siguiente código resolvería la pregunta de pertenencia de un valor x a este conjunto, si conocemos el tamaño del vector.

```
function pertenece (x : integer): boolean;
  var i: byte;
  begin
1.    i := 41;
2.    while (i<50) and (V[i]<x) do
3.      i := i + 1;
4.    return ((i<50) and (V[i]=x ));
  end;
```

Ya vimos que si no conocemos el tamaño del vector que almacena al conjunto y la terminación está dada por contenido, el código no debe cambiar demasiado para reflejar este hecho. En ese caso el código sería:

```
function pertenece (x : integer): boolean;
  var i: byte;
  begin
1.    i := 41;
2.    while (V[i]<>⋆) and (V[i]<x) do
3.      i := i + 1;
4.    return ((V[i]<>⋆) and (V[i]=x));
  end;
```

Lista vinculada ordenada

Si almacenamos el conjunto en una lista vinculada ordenada y manteniendo los formatos de *nodo* que mencionamos previamente, el código sería:

```
function pertenece (x : integer): boolean;
  var p: ↑nodo;
  begin
1.    p := inicio;
2.    while (p <>nil) and (p↑.valor<x) do
3.      p := p↑.siguiente;
4.    return ((p <>nil) and (p↑.valor=x));
  end;
```

Fácilmente podemos transformar este código si la lista vinculada ordenada tiene terminación dada por contenido.

Lista de alojamiento secuencial ordenada con examinación no secuencial

Podemos aprovechar aún más el orden de los contenidos de las celdas de manera tal de descartar partes más grandes de la lista. Si comparamos x con un elemento, con esa comparación podríamos decir que divide la lista en dos partes: la que contiene elementos menores que el examinado y la que contiene elementos mayores (porque no admitimos elementos iguales en un conjunto). Entonces elijo en qué parte de la lista debo seguir buscando a x.

Mientras más elementos permita descartar esa pregunta mejor se comportará el método. Se puede elegir para hacer la pregunta un elemento de una posición intermedia, pero el que permite siempre descartar una porción más grande de la lista es el elemento del medio.

Si tenemos la siguiente lista:

V:

25	***51***	***64***	***77***	***93***	***158***	***171***	***234***	***258***
41	***42***	***43***	***44***	***45***	***46***	***47***	***48***	***49***

Por ejemplo, si tomamos el elemento de la posición 43 para comparar puede ocurrir que:

(a) que el x buscado sea menor que él y en ese caso descartamos el trozo de la posición 43 a la 49 (el trozo más grande) y seguiríamos buscando entre los elementos de las posiciones 41 y 42 del vector (el trozo más chico).

(b) que el elemento x sea igual que él, en cuyo caso la búsqueda se detiene.

(c) que el x sea mayor que el elemento de la posición 43, entonces se debe seguir buscando en el trozo comprendido entre las posiciones 44 y 49 (el trozo más grande) y se descartan los elementos de las posiciones 41 y 42 (el trozo más chico).

Entonces vemos que esta elección de la posición puede ser muy buena en los casos **(a)** y **(b)** y muy mala en el caso **(c)**.

Queda claro que de esta manera hemos dividido la lista en tres partes 41 a 42, 43 y 44 a 49, por ello éste método sería una *trisección*.

¿Qué posición sería la más adecuada en general? La más adecuada sería aquella que dividiera las partes, en las que se debería continuar la búsqueda, en tamaños lo más similares posibles para que no haya un caso muy bueno y uno muy malo, sino dos casos semejantes.

Entonces, la posición que se utiliza es llamada el *medio de la lista*. Si la lista tiene cantidad *impar* de elementos, el medio es una celda, entonces esa celda será la lista del medio y quedan la otras dos listas de igual tamaño a ambos lados. Si la lista posee cantidad *par* de elementos, el medio sería una frontera entre celdas; en ese caso se elige la celda de la derecha o la de la izquierda de la frontera, para que sea la lista del medio con la cual comparar. Por lo tanto, las listas que quedan a cada lado tendrán una diferencia de un elemento, es decir que una lista contiene un elemento más que la otra.

Este tipo de estrategia se conoce como ***búsqueda binaria***, y en particular este algoritmo que divide la lista en tres se lo denomina ***trisección***.

En la lista que tenemos de ejemplo, como tiene cantidad impar de elementos elegiríamos el elemento de la posición 45 para comparar contra x la primera vez, y si ése no era el elemento buscado continuaríamos la búsqueda de la misma manera sobre el trozo de la derecha o el de la izquierda como corresponda en función del x buscado.

Como el trozo de vector sobre el que se busca puede ir variando, necesitamos mantener los límites que describen el trozo corriente.

Así, el siguiente código resolvería la pregunta de pertenencia de un valor x al conjunto almacenado en una lista secuencial ordenada de tamaño conocido, utilizando búsqueda por *trisección*.

```
function pertenece (x : integer): boolean;
  var li, ls, m: byte;
  begin
1.    li := 41;
2.    ls := 49;
3.    m := (li + ls) div 2;
4.    while (li<=ls) and (V[m]<>x) do
      begin
5.        if V[m]<x then li := m+1;
6.                  else ls := m-1;
7.        m := (li + ls) div 2;
      end;
8.    return (li<=ls);
  end;
```

La posición media se calcula en este caso como $\lfloor (l_i+l_s)/2 \rfloor$ si es división real, o haciendo $(l_i+l_s)/2$ si es división entera.

En caso que la búsqueda se detenga por la condición de $V[m] <> x$ significa que hemos encontrado el elemento. En caso que la búsqueda se detenga por $l_i > l_s$ significa que la lista quedó vacía y entonces el x no pertenece.

En las líneas 1, 2 y 3 se hace la inicialización de los límites y de la posición media. Las condiciones de la iteración de la línea 4 hablan de que la lista no es vacía y que el elemento examinado no es el buscado. Entre las líneas 5 y 6, dependiendo del elemento examinado, elegimos el trozo de lista donde continuará la búsqueda modificando el límite que corresponda. En función de los nuevos límites se recalcula la posición media en la línea 7.

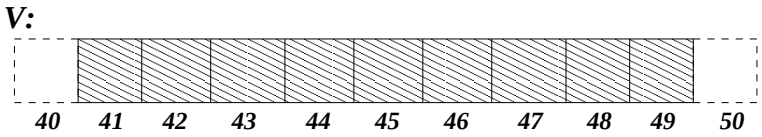
Al salir de la iteración, la pertenencia se verifica sólo chequeando si la lista considerada finalmente no fue vacía (en otro caso la respuesta será falso).

Cabe destacar que en caso de que la lista quedara vacía dentro de la iteración (tenía un elemento y al actualizar los límites en las líneas 5 y 6 queda vacía) el valor medio se calcula igual, pero claramente este valor quedará en una posición no válida para buscar (fuera o dentro) del trozo de vector inicial. Por lo tanto, el **and** necesita seguir siendo con evaluación condicional del segundo operando.

Por ejemplo: si en un paso de la iteración tenemos $l_i = l_s = 41$ entonces $m = 41$ y si el elemento no es el buscado, por ejemplo si es menor entonces $l_s = 40$. Como $l_i > l_s$ la lista está vacía, pero m será 40 que es una posición fuera del trozo del vector inicial.

Hemos descripto hasta ahora un trozo de vector dando la posición de la primera celda y de la última, pero no es la única manera de hacerlo. Por ejemplo, podríamos describir el trozo dando la posición de la primera celda del trozo de interés y de la celda siguiente a la última (la primera fuera del trozo). Si describiéramos el trozo de lista de esta manera, deberíamos mantener esta política para describir cada trozo a lo largo de la iteración del algoritmo.

Analizamos las distintas maneras (las más sencillas, porque habría más) de describir un trozo de vector, viendo como dar las fronteras. Tomemos por ejemplo la siguiente lista, en donde la zona sombreada corresponde a las celdas donde están almacenados los elementos del conjunto:



Las distintas formas de describir sus límites serían:

Límite inferior		*Límite superior*	
inclusivo	**41**	inclusivo	**49**
inclusivo	**41**	exclusivo	**50**
exclusivo	**40**	inclusivo	**49**
exclusivo	**40**	exclusivo	**50**

Notar que el algoritmo debería reflejar cómo se elija describir las fronteras. Los límites que usamos en ese caso eran ambos inclusivos. La forma de describir los límites afecta no sólo a la inicialización y actualización de límites, sino también al cálculo del medio y a la manera de preguntar por lista vacía. Por ejemplo, si ambos límites son inclusivos $l_i = l_s + 1$ indica lista vacía; pero si ambos límites son exclusivos la condición de lista vacía sería $l_i = l_s$.

Realmente lo correcto sería poder describir fronteras: el trozo empieza en la frontera entre las celdas 40 y 41 y termina en la frontera entre las celdas 49 y 50, pero ningún lenguaje de programación nos permite dar fronteras realmente. Entonces, se debe adoptar alguna convención para describirlo (tal como ocurrió cuando dividimos en dos una lista de tamaño par) y luego de elegida la convención se respeta en todos los lugares en que sea necesario utilizarla.

Así, antes de realizar el algoritmo de búsqueda como la trisección debemos elegir la convención para cada uno de los límites (inclusivo o exclusivo) y cómo dividimos la lista de tamaño par (el trozo más grande hacia dónde queda).

Con cada una de las convenciones posible se obtendrá un código distinto, aunque todavía no vamos a ver qué otras elecciones se pueden modificar en dicho código, es bueno que sepan desde ahora que las hay y que si uno no las tiene en cuenta puede traer problemas a la hora de utilizar el algoritmo.

Distribución pseudo-aleatoria de datos

Otra manera de almacenar los elementos de un conjunto $X \subseteq U$ en un conjunto de celdas es haciendo uso de una función pseudo-aleatoria $h : U \longmapsto \mathbb{I}_{M-1} \cup \{0\}$, que nos permita particionar a U y por consiguiente a X, de acuerdo a los valores devueltos por h [1]. Es decir que tendríamos un subconjunto, no necesariamente distinto de vacío, asociado a cada uno de los posibles valores de la función [2].

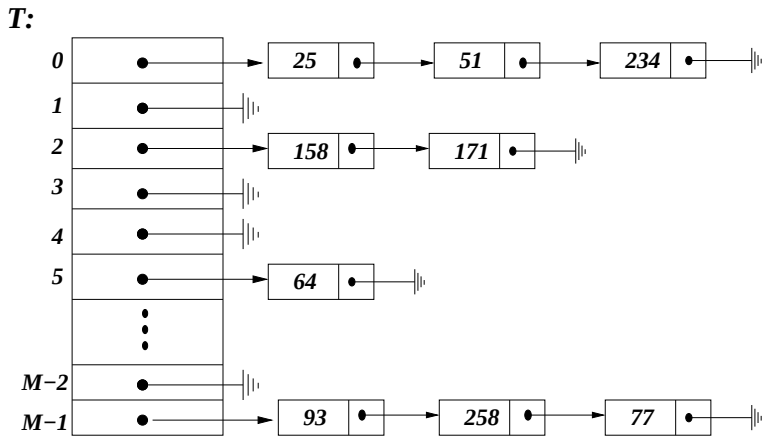
Hay distintas maneras de almacenar cada subconjunto: las dos más sencillas son la lista secuencial desordenada y la lista vinculada desordenada. De acuerdo al método elegido, se dice que usan una *técnica de rebalse abierto* (si usan listas secuenciales) o una *técnica de rebalse separado* (si usan listas vinculadas).

Más detalles sobre este tema los podrán encontrar en la bibliografía y en los apuntes de la cátedra sobre *Distribución pseudo-aleatoria de datos*.

Rebalse separado

En esta técnica cada subconjunto se almacena como una lista vinculada desordenada y tenemos una cabecera de punteros que dan los inicios de cada lista, para ello tenemos un espacio secuencial cuyos subíndices son los posibles valores de la función.

La siguiente figura muestra gráficamente una posible situación para un conjunto como el que se usa de ejemplo en las listas secuenciales, si usamos una distribución pseudo-aleatoria de datos con tratamiento de rebalse separado:



En este caso hemos considerado la función h como:

$$h(25) = h(51) = h(234) = 0$$
$$h(158) = h(171) = 2$$
$$h(64) = 5$$
$$h(93) = h(258) = h(77) = M - 1$$

[1] Recordar que con $\mathbb{I}_{M-1}$ representamos el conjunto de los $M - 1$ primeros números naturales.

[2] Por ser h una función total, se puede definir una relación de equivalencia sobre U: $R \subseteq U \times U$ tal que $R = \{(x, y)/x, y \in U \wedge h(x) = h(y)\}$, y toda relación de equivalencia permite particionar el conjunto sobre el que se define en clases de equivalencia.

Asumiendo que los inicios de las listas se mantienen en T con posiciones de 0 a $M-1$, el siguiente código resolvería la pertenencia sobre una distribución pseudo-aleatoria de datos con tratamiento de rebalse separado:

```
function pertenece (x : integer): boolean;
  var p: ↑nodo;
  begin
1.    p := T[h(x)];
2.    while (p <>nil) and (p↑.valor<>x) do
3.      p := p↑.siguiente;
4.    return (p <>nil );
  end;
```

Claramente se puede observar en el código que la búsqueda que se realiza es la misma que se realizaba para una lista vinculada desordenada, y que lo único que cambia es dónde se inicia la búsqueda (en este caso en la posición de T dada por la aplicación de $h(x)$).

Rebalse Abierto

Tal como mencionamos previamente, esta técnica trabaja con listas secuenciales desordenadas y en particular listas secuenciales desordenadas con terminación dada por contenido. Sin embargo, estas listas secuenciales difieren un poco respecto de las que vimos previamente porque se espera que se alojen en un espacio común a todas.

Como la terminación de las listas es dada por contenido, debe ser siempre M mayor que la cardinalidad del conjunto a alojar. Mientras mayor sea la proporción de celdas que quedarán vacías va a ser menor la cantidad de celdas a examinar para resolver la pertenencia (más adelante lo veremos en detalle).

Cada lista se inicia en la posición de T dada por la aplicación de $h(x)$, por lo tanto T consta también de M posiciones (desde la posición 0 hasta la posición $M-1$). Un detalle importante es que como todas las listas comparten el espacio disponible éste se considera *circular*, es decir la posición siguiente a la $M-1$ es la 0.

Anteriormente vimos que en una lista secuencial para pasar de una celda a la siguiente sólo sumábamos 1, pero hay distintas maneras de avanzar a la siguiente celda de la lista. Cada una de estas maneras da origen a una técnica de rebalse abierto particular. Mencionaremos aquí algunas de ellas, a las que creemos las más representativas.

Un aspecto importante para tener en cuenta al utilizar una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto, es que se debe siempre asegurar que cualquier lista podría contener eventualmente las M celdas posibles. Así, si intentamos realizar M pasos sobre cualquiera de las listas deberíamos pasar por las M celdas; es decir que en M pasos no podemos volver a pasar por una celda ya visitada. Por ello en muchos casos, para estar seguros, se suele colocar una condición adicional en la parte iterativa del proceso para que se hagan a lo sumo M pasos al recorrer cualquier lista.

Rebalse abierto lineal

El *rebalse abierto lineal* se caracteriza porque el avance a la siguiente celda es una cantidad constante. Por ejemplo la más simple es que la celda siguiente a la celda i sea la $i + 1$, es decir para pasar a la celda siguiente sumamos 1 a la posición actual. Otros ejemplos de avances para esta técnica podría ser siempre sumar 2 o siempre sumar 3, o siempre restar 1. Los avances a la siguiente celda siempre se hacen desplazándose una cantidad fija de celdas desde la actual.

Al considerar el espacio disponible como circular al avanzar debemos tenerlo en cuenta. Si tenemos que el avance suma 1, por ejemplo, al sumar 1 a la posición $M - 1$ nos daría la posición M que no pertenece a la estructura y en su lugar debería darnos la posición 0. Eso lo logramos fácilmente haciendo que el avance de la celda i a la siguiente se calcule como $(i + 1) \mod M$.

La siguiente figura muestra una posible situación usando una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto lineal:

T:

0	***25***
1	***51***
2	***158***
3	***234***
4	***171***
5	★
6	***258***
7	★
8	***77***
	⋮
M−3	***64***
M−2	★
M−1	***93***

En este caso hemos tomado la función h tal que:

$h(25) = h(51) = h(234) = 0$
$h(158) = h(171) = 2$
$h(258) = 6$
$h(77) = 8$
$h(64) = M - 3$
$h(93) = M - 1$

Los elementos compiten entre sí por el uso del espacio disponible (cualquier celda puede ser ocupada por un elemento de cualquier lista que lo necesite). Aunque ahora no se pretende entrar en demasiados detalles sobre esta estructura, es necesario aclarar que los elementos del conjunto necesariamente se almacenan en algún orden y en esta estructura el elemento que encuentra una marca de fin ($\star$) en una celda, es el que se puede ubicar ahí por llegar primero.

Por ejemplo, para los elementos del conjunto que se muestran en la gráfica, una posible secuencia seguida en su almacenamiento que hubiera resultado en esa estructura sería: 25, 51, 93, 258, 158, 64, 234, 77, 171. Éste no es el único orden posible que hubiera dado como resultado la misma gráfica, hay varios más, lo que todos ellos deben tener en común es que 25, 51 y 158 deben haber llegado antes de 234 y éste antes de 171. Usted podría responder ¿por qué?

Asumiendo que trabajamos con un vector T con posiciones de 0 a $M - 1$, el siguiente código

resolvería la pertenencia sobre una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto lineal:

```
function pertenece (x : integer): boolean;
  var i: integer;
  begin
1.    i := h(x);
2.    while (T[i]<> ⋆) and (T[i]<>x) do
3.      i := (i + 1)mod M;
4.    return (T[i]<> ⋆);
  end;
```

Rebalse abierto cuadrático

El *rebalse abierto cuadrático* se caracteriza porque el avance a la siguiente celda es una cantidad que varía con el número de celda de la lista que se desea acceder; es decir, primero suma 1, luego suma 2, luego suma 3 y así siguiendo. Por lo tanto si queremos alcanzar la k-ésima celda de la lista i habremos sumado a la posición i la cantidad $\sum_{j=1}^{k-1} j = \frac{(k-1)*k}{2}$, que es de naturaleza *cuadrática*.

Por ejemplo, para los elementos del conjunto que estamos considerando y dado que el almacenamiento de los elementos se realizó también en el orden: 25, 51, 93, 258, 158, 64, 234, 77, 171; la estructura resultante sería:

T:

0	***25***
1	***51***
2	***158***
3	***234***
4	★
5	***171***
6	***258***
7	★
8	***77***
	⋮
M−3	***64***
M−2	★
M−1	***93***

Considerando la función h:

$h(25) = h(51) = h(234) = 0$
$h(158) = h(171) = 2$
$h(258) = 6$
$h(77) = 8$
$h(64) = M - 3$
$h(93) = M - 1$

El siguiente código resolvería la pertenencia sobre una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto cuadrático:

```
function pertenece (x : integer): boolean;
  var i,k: integer;
  begin
1.     i := h(x);
2.     k := 1;
3.     while (T[i]<> ⋆) and (T[i]<>x) do
        begin
4.          i := (i + k)mod M;
5.          k := k+1;
        end
6.     return (T[i]<> ⋆);
  end;
```

Rebalse abierto paso realeatorizado

El *rebalse abierto de paso realeatorizado* se caracteriza porque el avance a la siguiente celda depende de una nueva función pseudo-aleatoria $p : U \longmapsto \mathbb{I}_{M-1}$, que es una cantidad que varía con el elemento que se está considerando en ese momento; pero para ese elemento siempre se usa el mismo valor (por ser p una función). Es decir que la lista secuencial que se recorre puede ser distinta para cada elemento cuyo valor de función sea i.

Por ejemplo, para el mismo conjunto que venimos utilizando, considerando también que el almacenamiento de los elementos fue en el orden: 25, 158, 51, 93, 258, 64, 234, 77, 171 y que la función p para los elementos que encontraron su celda ya ocupada toma los valores que se indican; la estructura resultante sería:

T:

0	***25***
1	***234***
2	***158***
3	★
4	***51***
5	***171***
6	***258***
7	★
8	***77***
	⋮
M−3	***64***
M−2	★
M−1	***93***

Considerando la función h:

$h(25) = h(51) = h(234) = 0$
$h(158) = h(171) = 2$
$h(258) = 6$
$h(77) = 8$
$h(64) = M - 3$
$h(93) = M - 1$

y la función p como:

$p(51) = 2$
$p(234) = 1$
$p(171) = 3$

El siguiente código resolvería la pertenencia sobre una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto paso realeatorizado:

```
function pertenece (x : integer): boolean;
  var i,k: integer;
  begin
1.     i := h(x);
2.     k := p(x);
3.     while (T[i]<> ⋆) and (T[i]<>x) do
4.       i := (i + k)mod M;
5.     return (T[i]<> ⋆);
  end;
```

Rebalse abierto realeatorizado total

El *rebalse abierto realeatorizado total* se caracteriza porque más que pensar que la posición de la siguiente celda se obtiene como un desplazamiento desde la celda actual, este método obtiene directamente la posición a considerar para la siguiente celda, de acuerdo al número de celda de la lista a alcanzar. Por lo tanto, la función de pseudo-azar en este caso se transforma en:

$$h : U \times \mathbb{I}_M \longmapsto \mathbb{I}_{M-1} \cup \{0\}$$

Es decir, la posición de inicio de la lista para x la da $h(x, 1)$, $h(x, 2)$ da la segunda posición y así siguiendo hasta la M-ésima celda. Entonces, esta secuencia de posiciones para x se puede ver como una permutación de los valores desde 0 a $M - 1$.

Para el mismo conjunto que venimos utilizando como ejemplo y considerando también que el almacenamiento de los elementos fue en el orden: 25, 51, 93, 258, 158, 64, 234, 77, 171 y que la función h ahora es como se detalla; la estructura resultante sería:

T:

0	***25***
1	***234***
2	***158***
3	★
4	***51***
5	***171***
6	***258***
7	★
8	***77***
	⋮
M−3	***64***
M−2	★
M−1	***93***

Considerando la función h:

$h(25, 1) = 0$
$h(51, 1) = 0, \quad h(51, 2) = 4$
$h(93, 1) = M - 1$
$h(258, 1) = 6$
$h(158, 1) = 2$
$h(64, 1) = M - 3$
$h(234, 1) = 0, \quad h(234, 2) = 1$
$h(77, 1) = 8$
$h(171, 1) = 2, \quad h(171, 2) = 5$

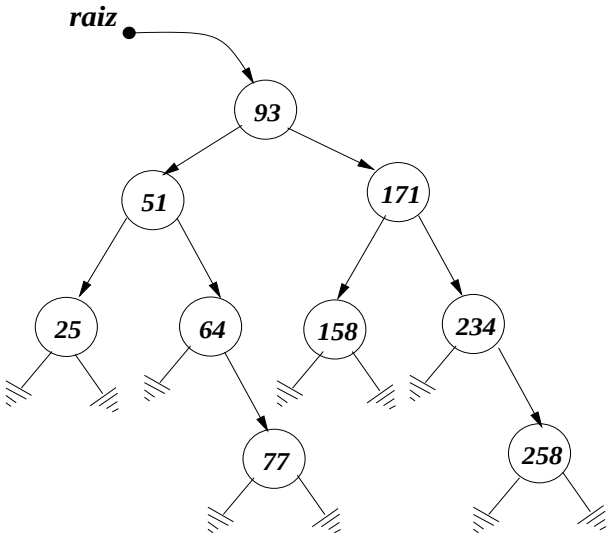
El siguiente código resolvería la pertenencia sobre una distribución pseudo-aleatoria de datos con tratamiento de rebalse abierto realeatorizado total:

```
function pertenece (x : integer): boolean;
  var i,k: integer;
  begin
1.    k := 1;
2.    i := h(x,k);
3.    while (T[i]<> ⋆) and (T[i]<>x) do
      begin
4.        k := k+1;
5.        i := h(x,k);
      end
6.    return (T[i]<> ⋆);
  end;
```

Árboles Binarios Ordenados de Búsqueda

Vimos que una manera mejor que la examinación secuencial para recorrer una lista secuencial ordenada era realizando examinación de acuerdo a una trisección. Si colocáramos en una estructura vinculada los elementos del conjunto de acuerdo a cómo se examinan en una trisección, es decir primero el que se encontraba en la posición media de la lista, a su izquierda el de la posición media del trozo de la izquierda, a la derecha el de la posición media del trozo de la derecha, y recursivamente siguiendo el mismo razonamiento hasta vincular todos los elementos de la lista; obtendríamos un caso particular de una nueva estructura a la que llamamos *árbol binario ordenado de búsqueda*.

Si representamos de esta manera el conjunto utilizado como ejemplo para la lista secuencial ordenada, tendríamos la siguiente estructura:



Por ser una estructura vinculada debemos conocer la dirección de la primera celda a fin de poder recorrerla, a esa dirección la llamamos *raíz* (por ser dónde se inicia un árbol). Cada celda debe contener un valor y dos apuntadores a celda, uno a la izquierda que apunta a una celda cuyo valor será menor (su hijo izquierdo) y uno a la derecha que apunta a una celda cuyo contenido será mayor (su hijo derecho). En caso de que no exista alguno de estos elementos el apuntador será nulo (puntero a nil).

Esta estructura es un árbol *binario* porque los elementos pueden vincular a lo sumo a otros dos, es *ordenado* porque justamente se habla de hijo izquierdo e hijo derecho y no sólo de hijos, y es de *búsqueda* porque está organizado de manera tal de facilitar la búsqueda o localización de un elemento.

Básicamente, la idea del árbol binario ordenado de búsqueda es que tenemos un elemento con el que comparamos a x y que nos permite dividir nuestro conjunto en tres partes: el subconjunto de elementos menores que x, $\{x\}$ y el subconjunto de elementos mayores. Luego de la comparación con x, si x es el elemento buscado podemos responder que el elemento pertenece al conjunto, en otro caso la comparación con x nos permite decidir si debemos continuar buscando en el subconjunto de los elementos menores que x (si lo que buscamos es menor que x) y descartar el subconjunto de los mayores que x, o a la inversa.

En este caso si la búsqueda de un x nos guía a un puntero nil significa, como en lista vinculada, que agotamos el trozo del conjunto donde se podía encontrar a x y por lo tanto debemos responder que x no pertenece.

El siguiente código resuelve la pertenencia si el conjunto se ha almacenado en un árbol binario ordenado de búsqueda, asumiendo que las celdas son de tipo *nodo* y cada celda tiene un campo *valor*, y dos campos que son puntero a nodo, a los que llamamos *hi* y *hd*, correspondiendo a los punteros a hijo izquierdo y derecho respectivamente; *raíz* es también de tipo puntero a nodo.

```
function pertenece (x : integer): boolean;
  var p:↑nodo;
  begin
1.    p := raíz;
2.    while (p<>nil) and (p↑.valor<>x) do
      begin
3.        if p↑.valor<x then p := p↑.hd;
4.                      else p := p↑.hi;
      end;
5.    return (p <>nil);
  end;
```

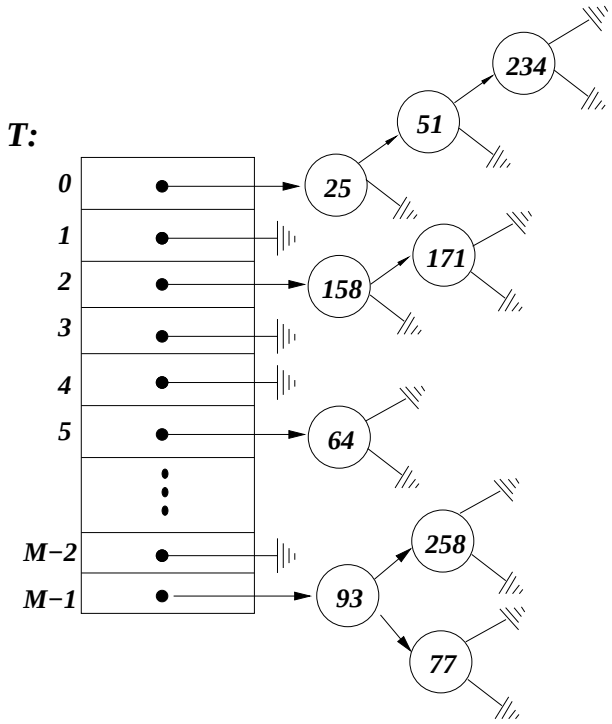
Analizando este código y el código que resuelve la pertenencia sobre una lista secuencial ordenada usando trisección podemos observar algunas similitudes en las estructuras. La línea 1 aquí realiza la inicialización de la búsqueda lo que hacían las líneas 1, 2 y 3 de la trisección. La condición para la iteración también corresponde a la pregunta por que el conjunto no sea vacío y que que no hayamos alcanzado el elemento buscado. Las líneas 3 y 4 actualizan, de acuerdo a si el elemento buscado es mayor o menor que el actualmente examinado, hacia dónde debe continuar la búsqueda (en la trisección esto lo hacían las líneas 5, 6 y 7). Finalmente la pertenencia se resuelve en ambos casos preguntando si el conjunto actual es vacío o no.

Combinando estructuras

Hemos visto algunas de las representaciones más conocidas y usadas para conjuntos y cómo se resuelve la pertenencia sobre cada una de ellas. También pudimos observar que existen aspectos comunes en todos los códigos.

Podemos ahora intentar combinar algunas de estas estructuras para obtener otras nuevas. Por ejemplo, habíamos visto que una distribución pseudo-aleatoria de datos con tratamiento de rebalse separado particionaba el conjunto, en subconjuntos, basados en una función de pseudo-azar h. Esta estructura representaba cada uno de los subconjuntos utilizando una lista vinculada desordenada, pero nada impide que a estos subconjuntos se los represente como árboles binarios ordenados de búsqueda.

Si representamos gráficamente el conjunto que veníamos utilizando tendríamos la siguiente figura:



El código necesario para resolver la pertenencia al conjunto debe combinar también aspectos referentes a ambos códigos, se inicializa la búsqueda como en un rebalse separado, pero la búsqueda procede de ahí en más como en un árbol binario ordenado de búsqueda.

```
function pertenece (x : integer): boolean;
  var p:↑nodo;
  begin
1.    p := T[h(x)];
2.    while (p<>nil) and (p↑.valor<>x) do
      begin
3.        if p↑.valor<x then p := p↑.hd;
4.                      else p := p↑.hi;
      end;
5.    return (p <>nil);
  end;
```

Reconocimientos

El presente apunte se realizó tomando como base notas de clases, de Estructuras de la Información y de Estructuras de Datos y Algoritmos, del Ing. Hugo Ryckeboer en la Universidad Nacional de San Luis.