Unidad 3 – Estructuras de Datos Avanzadas

Introducción

Hasta el momento se estudiaron las siguientes estructuras estáticas y dinámicas básicas para representar diferentes colecciones de elementos:

- Arreglo simple y matriz
- Arreglo con tope
- Arreglo dinámico
- Arreglo dinámico con tope
- Lista simple
- Árbol binario de búsqueda

En este capítulo se analizarán algunas variantes a dichas estructuras y se presentarán otras nuevas con el principal objetivo de escribir algoritmos más eficientes.

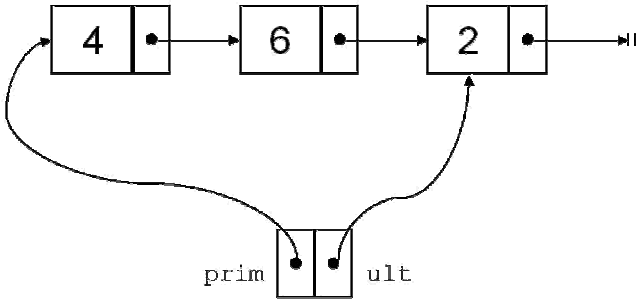
Lista Simple con Puntero al Principio y Puntero al Final

Hasta ahora se vio una estructura de lista cuyo único acceso es por su extremo inicial. En varias operaciones es necesario acceder al último nodo y en una lista simple esto es costoso. La lista PPF es una estructura que mejora el orden de dichas operaciones. Además de tener un puntero al primer nodo, se tiene un puntero al último.

Implementación:

```
typedef struct nodoL { T info;
                       nodoL *sig;
                     } Nodo;
typedef Nodo * Lista;

typedef struct { Lista prim;
                 Lista ult;
               } ListaPPF;
```

Ejemplo:



Operaciones Básicas:

```
void Crear (ListaPPF &l)
{
      l.prim = NULL;
      l.ult  = NULL;
}

boolean Vacia (ListaPPF l)
{
      return (boolean)(l.prim==NULL && l.ult==NULL);
}

//Precondición : !Vacia(l)
T Primero (ListaPPF l)
{
      return l.prim->info;
}

//Precondición : !Vacia(l)
T Ultimo (ListaPPF l)
{
      return l.ult->info;
}

//Precondición : !Vacia(l)
void Resto (ListaPPF &l)
{
      Lista aux = l.prim->sig;
      delete (l.prim);
      l.prim = aux;
      if (l.prim == NULL))
            l.ult = NULL;
}

void Insfront (ListaPPF &l, T e)
{
      Lista nuevo = new nodo;
      nuevo->info = e;
      nuevo->sig = l.prim;
      l.prim = nuevo;
      if (l.ult == NULL)
            l.ult = nuevo;
}
```

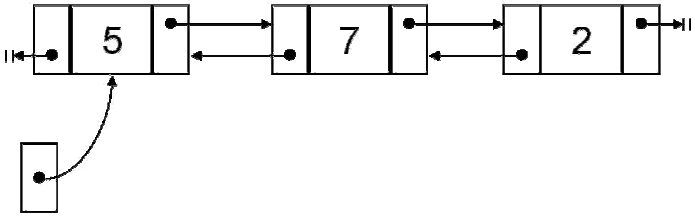
Lista Doblemente Encadenada

En algunas aplicaciones puede ser deseable determinar con rapidez el siguiente y el anterior de un determinado elemento de una lista. En una lista simple determinar el anterior es una operación costosa. Una lista DE es una nueva estructura donde cada nodo no sólo contiene un puntero al nodo siguiente sino también al anterior.

Implementación:

```
typedef struct nodoL { T info;
                       nodoL * sig;
                       nodoL * ant;
                     } Nodo;

typedef Nodo * ListaDE;
```

Ejemplo:



Operaciones Básicas:

```
void Crear (ListaDE &l)
{
      l = NULL;
}

boolean Vacia (ListaDE l)
{
      return (boolean)(l==NULL);
}

void Insfront(ListaDE &l, T e)
{
      ListaDE nuevo = new nodo;
      nuevo->info = e;
      nuevo->ant = NULL;
      if (l == NULL)
            nuevo->sig = NULL;
      else
      {
            l->ant = nuevo;
            nuevo->sig = l;
      }
      l = nuevo;
}

//Precondición : !Vacia(l)
T Primero (ListaDE l)
{
      return l->info;
}
```

```
//Precondición : !Vacia(l)
void Resto (ListaDE &l)
{
      ListaDE aux = l;
      l = l->sig;
      delete (aux);
}
```

Mapeo o Correspondencia

Un mapeo o correspondencia es una función:

$M : A \rightarrow B$
$M(a) = b \qquad a \in A \text{ y } b \in B$

Deseamos definir una estructura de datos que represente esta función, de modo tal que dado $a \in A$, podamos obtener en forma rápida el elemento M(a) del codominio.

La estructura que vamos a definir es particularmente apropiada cuando A es un subrango de los naturales. Por ahora vamos a considerar que A es un subrango de los naturales empezando en cero. Una forma clara y eficiente de representar el mapeo en este caso es definir un arreglo donde cada valor del dominio es un índice en el arreglo y es también la posición de su imagen según la función M.

Implementación:

No necesariamente en una correspondencia todos los elementos del dominio tienen una imagen en el codominio. En esos casos, para la implementación, el codominio debería contener un valor $\perp$ (indefinido) para que la función sea total, es decir $B \cup \perp = B_{\perp}$. Para contemplar esto, el tipo base del arreglo se define de la siguiente manera:

```
typedef struct {  boolean existe;
                  T info;
               } T';

typedef T' Mapeo[TAMAÑO];
```

T es el tipo de los elementos del codominio B, y T' es el tipo para $B_{\perp}$. TAMAÑO es el valor máximo del subrango A.

Operaciones Básicas:

```
void Crear (Mapeo &m)
{
      int i;
      for(i=0; i<TAMAÑO; i++)
            m[i].existe = false;
}

boolean Pertenece (Mapeo m, int pos)
{
      return (m[pos].existe);
}
```

```
void Insertar (Mapeo &m, T e, int pos)
{
      m[pos].existe = true;
      m[pos].info = e;
}

//Precondición: Pertenece(m,pos)
T Obtener (Mapeo m, int pos)
{
      return m[pos].info;
}

//Precondición: Pertenece(m,pos)
void Eliminar (Mapeo &m, int pos)
{
      m[pos].existe = false;
}
```

En el caso en que el subrango A no empiece en cero, al implementar las operaciones anteriores será necesario efectuar una traslación que, dado a∈ A, le reste a dicho número el valor mínimo del subrango A en forma previa a acceder a la celda que le corresponde en el mapeo.

En el caso en que el conjunto A no sea un subrango de los naturales, la estructura de datos que veremos a continuación será más apropiada que ésta para implementar la correspondencia M.

Hash

Una técnica muy importante y muy útil para implementar colecciones se denomina *Hashing* o *dispersión*. Es particularmente apropiada para colecciones en las cuales los elementos poseen una clave K que los identifica.

La idea fundamental de esta técnica es que el conjunto (potencialmente infinito) de elementos se divide en un número finito de clases. Si se desea tener *B* clases (numeradas de 0 a *B*-1), se usa una función llamada función de dispersión $h : K \rightarrow N$ tal que para la clave *x* de cada elemento del tipo base de la colección, *h*(*x*) sea uno de los enteros de 0 a *B*-1.

Lógicamente *h*(*x*) es la clase a la cual pertenece el elemento. A *h*(*x*) se le denomina *valor de dispersión* de *x*. A las clases se les denomina *cubetas* y se dice que el elemento con clave *x* pertenece a la cubeta *h*(*x*).

En esta estructura se espera que las cubetas tengan casi el mismo tamaño, es decir que la cantidad de elementos para cada cubeta sea pequeña. Entonces, si hay *n* elementos en el conjunto, en promedio, cada cubeta tendrá *n*/*B* miembros. Si se puede estimar *n* y elegir *B* aproximadamente igual de grande, entonces una cubeta tendrá en promedio solo uno o dos elementos, y de esa manera el acceso a un determinado elemento se puede realizar con mayor eficiencia (O(1) en el promedio de los casos). Cuando esto sucede, decimos que existe un número muy bajo de *colisiones*. Esto significa que la probabilidad de que dos elementos caigan en una misma cubeta es muy baja, garantizando así que la cantidad de elementos en cada cubeta sea pequeña.

La definición de una función de dispersión *h* que produzca un bajo número de colisiones es un problema complejo que no abordaremos en este curso. En la mayoría de los problemas, asumiremos que se cuenta con una función de dispersión ya definida, que es lo suficientemente buena como para garantizar un número bajo de colisiones.

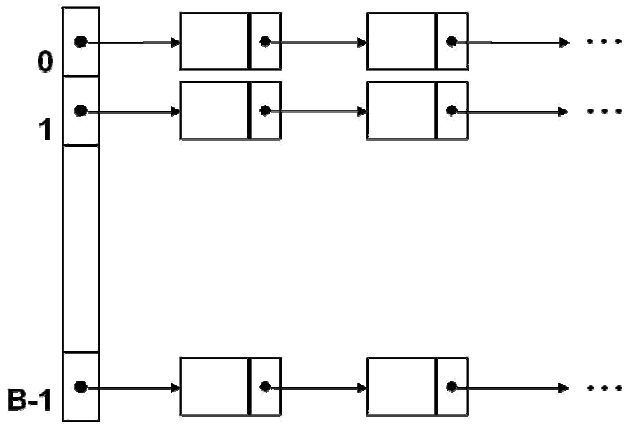
Debido a que la cantidad de cubetas es fija, se utiliza un arreglo para almacenarlas. En cambio como la cantidad de elementos en cada una es variable, se utiliza una estructura de lista simple para almacenarlos. Se opta por una lista simple debido a que la cantidad de elementos en cada cubeta es no acotada, si bien se espera que tenga pocos elementos.

Implementación:

```
const int B = /* valor adecuado */;

typedef struct nodoL { T info;
                       nodoL * sig;
                     } Nodo;

typedef Nodo * Lista;

typedef Lista Hash[B];
```

Se asume que cada elemento de tipo T cuenta con una clave de tipo K que lo identifica y que además se cuenta con la función *DarClave*: T → K que devuelve dicha clave.

Ejemplo:



Operaciones Básicas:

```
void Crear(Hash &H)
{
      int i;
      for(i=0;i<B;i++)
            CrearLista(H[i]);
}

boolean Pertenece(Hash H, K clave)
{
      int cubeta = h(clave);
      return PerteneceLista (H[cubeta],clave);
}
```

```
//Precondición: !Pertenece(H,DarClave(e))
void Insertar (Hash &H, T e)
{
      K clave = DarClave(e);
      int cubeta = h(clave);
      Insfront (H[cubeta],e);
}

//Precondición: !Pertenece(H,DarClave(e))
T Obtener (Hash H, K clave)
{
      int cubeta = h(clave);
      return ObtenerEnLista (H[cubeta],clave);
}

//Precondición: !Pertenece(H,DarClave(e))
void Eliminar (Hash &H, K clave)
{
      int cubeta = h(clave);
      BorrarEnLista (H[cubeta],clave);
}
```