Estructura de Datos y Algoritmos

Implementación de tipos de datos recurrentes lineales

1. El TAD Lista

Las listas son una estructura particularmente flexibles porque pueden crecer o achicarse a demanda y los elementos pueden ser accedidos, insertados o borrados en cualquier posición dentro de la lista.

Introduciremos un número de operaciones básicas en listas y veremos que estructuras de datos utilizar para representar listas que soporten esas operaciones.

Trabajaremos con listas generales de la forma $A_1, A_2, \ldots, A_n$. Diremos el tamaño de la lista es n. LLamaremos a la lista de tamaño cero *lista vacia*.

Decimos que A_{i+1} sigue a A_i $(0 \leq i < n)$ y que A_{i-1} precede a A_i $(1 < i \leq n)$. El primer elemento de la lista es A_1 y el último A_n. El predecesor de A_1 y el sucesor de A_n no están definidos. La posición del elemento A_i es la i.

Para formar un tipo abstracto de datos a partir de la noción matemática de lista debemos definir un conjunto de operaciones en objetos del tipo lista. Como con muchos otros TAD no hay un conjunto de operaciones que sea adecuado para todas las aplicaciones. Veremos un conjunto de operaciones representativo. Luego veremos representaciones de las listas y las funciones adecuadas para cada representación.

Para simplificar asumiremos los elementos de la lista son enteros pero en general se admiten estructuras más complejas.

A continuación presentamos un conjunto de operaciones representativas en listas. En lo que sigue, l es una lista de objetos de tipo int, x es un objeto de este tipo y p es un entero que representa una posición.

Operaciones:

1. creo_vacia(l): esta función devuelve la lista vacia.
2. es_vacia(l): devuelve true si la lista es vacia y false si no lo es.
3. primero(l): devuelve el valor del elemento en la primer posición de la lista. Precondición: se puede utilizar esta operación solo si l no es vacia.
4. siguiente(p,l): devuelve la lista sin sus primeros p elementos. Si la lista tiene menos de p elementos se retorna la lista vacia.
5. anterior(p,l): devuelve la lista formada por los elementos anteriores e incluyendo el de posición p. Si p es 0 retorna NULL. Si la lista tiene menos de p elementos se retorna l.

6. inserto(l,x,p): inserta x en la posición p de la lista, es decir siguiendo al elemento p-1 y antes del p. Si la lista tiene menos de p elementos se coloca al final. Si $p \leq 0$ no se inserta.

7. busqueda(l,x): retorna la posición del elemento x en la lista l. Si x aparece más de una vez se retorna la posición de su primera ocurrencia. Si x no aparece se devuelve -1.

8. devolver(l,p,x): retorna en x el elemento en la posición p en l. Se devuelve falso si l no tiene posición p.

9. borrar(l,p): borra el elemento en la posición p de la lista l. El resultado es falso si no hay posición p.

10. imprimo(l): imprime los elementos de l en orden de ocurrencia.

Veremos dos representaciones: con **arreglos** y con **listas encadenadas**. Se presentan las operaciones y sus tiempos (para esto último asumimos la lista tiene largo n).

1.1. Representación de listas con arreglos

```
#define largomax 100;

struct lista{
        int elem[largomax];
        int ultimo;
        };
```

En una representación de listas con arreglos, los elementos se almacenan en las posiciones contiguas del arreglo.

Con esa representación, una lista se recorre facilmente y elementos nuevos se pueden agregar al final. Insertar un elemento en el medio del arreglo requiere trasladar todos los elementos que le siguen una posición a la derecha para hacer lugar para el nuevo elemento. De igual modo, borrar un elemento del final es sencillo mientras que borrar del medio requiere mover todos los elementos que siguen al borrado una posición hacia la izquierda.

En la representación, el campo ultimo indica la cantidad de elementos que forman parte del arreglo. Luego, ultimo=0 indica el array es vacio y la última posición ocupada del array (cuando no es vacio) es último - 1.

Se presenta a continuación la implementación de las operaciones.

```
lista creo_lista()
{       lista l;
        l.ultimo=0;
        return l; }
```

es O(1) y $\Omega(1)$.

```
bool es_vacia(lista l)
{     return l.ultimo==0; }
```

es O(1) y $\Omega(1)$.

```
bool esta_llena(lista l)
{
return l.ultimo==largomax; }
```

es O(1) y $\Omega(1)$.

```
bool primero(lista l, int &a)
// precondicion: l es no vacia
{     if (es_vacia(l)) return false;
      else {a=l.elem[0]; return true;}
}
```

es O(1) y $\Omega(1)$.

```
void inserto(lista &l,int x,int p)
// precondicion: l no esta llena
{
      for(int i=l.ultimo;i>=p+1;i - -)
         l.elem[i]=l.elem[i-1];
      l.elem[p]=x;
      l.ultimo++;
}
```

el tiempo es $\Sigma_{i=p+1}^{n} 1 = n - p$ el peor caso es cuando $p = 1$.
Es O(n) $(n - 1 \leq n)$ y $\Omega(n)(n - 1 \geq n/2)$ para todo $n \geq 2$.

```
void siguiente(lista & l,int p)
{
      if (l.ultimo<p) l.ultimo=0;
      else
          { for(int i=0; i<l.ultimo-p; i++)
                l.elem[i]=l.elem[i+p];
            l.ultimo-=p;
          }
}
```

el tiempo es $\Sigma_{i=0}^{n-p} 1 = n - p + 1$ el peor caso es cuando $p = 0$.
Es O(n) $(n + 1 \leq 2n)$ para todo $n \geq 1$ y $\Omega(n)(n + 1 \geq n)$
para todo $n \geq 0$.

```
void anterior(lista & l, int p)
{
      if (p==0) l.ultimo=0;
      else if (l.ultimo<p) return;
           else l.ultimo=p;
}
```

es O(1) y $\Omega(1)$.

```
int busqueda(lista l,int x)
{   int i=0;
    while (i<l.ultimo && l.elem[i]! =x) i++;
    if (i==l.ultimo) return -1;
    else return i;
}
```

en el peor caso el tiempo es $\Sigma_{i=0}^{n-1} 1 = n$
el peor caso es cuando no se encuentra el elemento.
Es O(n) y $\Omega(n)$.

```
bool devolver(lista l,int p,int &x)
{     if (l.ultimo < p-1) return false;
      else {x=l.elem[p-1]; return true;}
}
```

es O(1) y $\Omega(1)$.

```
void borrar(lista &l,int p)
{
      if (l.ultimo < p) return false;
      else if (l.ultimo == p) {l.ultimo - - ;return true;}
           else
                for (int i=p;i<l.ultimo;i++)
                    l.elem[i]=l.elem[i+1];
      l.ultimo - - ;
}
```

en el peor caso el tiempo es $\Sigma_{i=p}^{n} 1 = n - p + 1$
el peor caso es cuando p=0. Es O(n) y $\Omega(n)$.

```
void imprimo_lista(lista q)
{
      cout << "\ n"<< "Lista:"<< "\ n";
      for (int i=0;i<q.ultimo;i++)
          cout << "elemento "<< i+1 << " "<< q.elem[i] << "\ n";
}
```

el tiempo es $\Sigma_{i=0}^{n-1} 1 = n$. Es O(n) y $\Omega(n)$.

1.2. Listas encadenadas

Es una estructura flexible porque puede crecer o achicarse a demanda. La definición de la estructura es la siguiente:

```
struct nodo{
        int valor;
        nodo *sig;};

typedef nodo *lista;
```

Creamos la lista creando nodos y agregandolos a partir de la lista vacia. Creamos nodos y los agregamos con la operación inserto.

Para crear nodos utilizamos la operación **new**. Si q es de tipo lista (o sea puntero a nodo), la operación q=new nodo; crea espacio en memoria para un nodo y asocia su dirección de memoria a q o sea q será un puntero a nodo.

Utilizamos q para cargar la información en el nodo. $q->valor$ hace referencia al dato valor del nodo apuntado por q y $q->sig$ hace referencia a un puntero a nodo (el siguiente a q).

En una lista encadenada utilizamos el dato "sig"para referirnos al siguiente nodo en la lista. Cuando un nodo es el último en una lista su campo "sig"tendrá valor NULL. Cuando no es el último tendrá la dirección de memoria del siguiente nodo.

Cuando borramos un nodo de una lista utilizamos la operación **delete** para retornar memoria que no utilizamos más.

delete q; retorna a la memoria el espacio ocupado por el nodo al cual q apunta.

Veamos a continuación la implementación de las operaciones.

```
lista creo_vacia()
{     return NULL; }
```

es O(1) y Ω(1).

```
bool es_vacia(lista l)
{     return (l==NULL); }
```

es O(1) y Ω(1).

```
int primero(lista l)
/* precondicion l no vacia */
{   return l->x; }
```

es O(1) y Ω(1).

```
lista siguiente(lista l, int p)
/* precondicion l no vacia */
{     lista q;

      if (es_vacia(l)) return NULL;
      else
          if (p==0) return l;
          else {
               q=siguiente(l− >sig,p-1);
               delete l;
               return q;}
}
```

es O(1) y $\Omega(1)$.

```
lista inserto(lista l,int a,int p)
{
      lista q,l1,l2;
      l1=l;
      if (p<=0) return l;
      else if ((p==1) || (l==NULL))
           { q=new nodo;
             q− >x=a;
             q− >sig=l;
             return q;}
           else {
                while (p>1 && l! =NULL)
                { l2=l;
                  l=l− >sig;
                  p - - ;}
                q=new nodo;
                q− >x=a;
                q− >sig=l;
                l2− >sig=q;
                return l1;
                }
}
```

en el peor caso es cuando $p \geq n$. El while interior tiene O(n).
El resto de las instrucciones es O(1), luego es O(n) y $\Omega(n)$.

```
void eliminar (lista l)
{
      lista q;
      q=l;
      while (l!=NULL)
      { l=l− >sig;
        delete q;
        q=l;}
}
```

el while se ejecuta n veces con n el largo de l,
es O(n) y $\Omega(n)$.

```
lista anterior(lista l,int p)
{     lista l1=l,l2;

      if (p==0) return NULL;
      while (p>=1 && l! =NULL)
      { l2=l;
        l=l− >sig;
        p - - ;
      }
      if (l==NULL) return l1;
      else {eliminar(l2− >sig); l2− >sig=NULL; return l1;}
}
```

el tiempo es el maximo entre p y n. El peor caso es cuando $p = n$.
El while es O(p) y eliminar es O(n-p), luego la funcion es O(n) y $\Omega(n)$.

```
int busqueda(lista l,int a)
{   int b;

    if (es_vacia(l)) return -1;
    else if ((l− >x)==a) return 1;
         else if ((b=busqueda(l− >sig,a))>0) return b+1;
              else return -1;
}
```

la recurrencia correspondiente es:

T(0)=1
T(1)=1
⋮
T(n)=T(n-1)+1

luego T(n)=n+1. Se sigue T(n) = O(n) y T(n) = $\Omega(n)$.

otra implementacion de busqueda:

```
int busqueda(lista l,int a)
{   int i;
    for(i=1;l! =NULL && (l->x)! =a;i++,l=l->sig) ;
    if (l==NULL) return -1;
    else return i;
}
```

el for se ejecuta n veces en el peor caso. Es O(n) y $\Omega(n)$.

```
bool devolver(lista l,int p,int *a)
{     if (p<=0) {*a=0;return false;}
      else if (p==1)
             if (es_vacia(l)) {*a=0;return false;}
             else {(*a)=l− >x; return true;}
           else if (es_vacia(l)) *a=0; return false;
                else return devolver(l− >sig,p-1,a);
}
```

la recurrencia correspondiente es:

T(0)=1
T(1)=1
⋮
T(n)=T(n-1)+1

luego T(n)=n+1. Se sigue T(n) = O(n) y T(n) = $\Omega(n)$.

otra implementacion de devolver:

```
bool devolver(lista l,int p,int *a)
{     int i;
      for(i=1;l! =NULL && i<p;i++,l=l->sig) ;
      if (l==NULL) return false;
      else {*a=l->x; return true;}
}
```

el for se ejecuta n veces en el peor caso. Es O(n) y $\Omega(n)$.

```
bool borrar(lista *l, int p)
{     lista l1,l2,q;
      l1=*l;

      if (p==0) return false;
      else if (p==1) {q=*l;*l=(*l)− >sig; delete q; return true;}
           else {while(p>1 && l1! =NULL)
                        {l2=l1;
                         l1=l1− >sig;
                         p - - ;}
                 if (l1==NULL) return false;
                 else {q=l1;l2− >sig=l1− >sig; delete q; return true;}
                 }
}
```

el tiempo es el maximo entre p y n. El peor caso es cuando $p = n$
luego la funcion es O(n) y $\Omega(n)$.

```
void imprimo(lista q)
{
    if (es_vacia(q)) cout << "Lista vacia \ n";
    else
        {cout << "\ n"<< "Lista:"<< "\ n";
        while (q!=NULL)
        {
         cout << q− >x << " ";
         q=q− >sig;
         }
        cout << "\ n";}
}
```

es O(n) y $\Omega(n)$ (se recorre toda la lista).