Estructura de Datos y Algoritmos

Implementación de tipos de datos recurrentes lineales

1. Listas doblemente encadenadas

Algunas veces es conveniente recorrer listas de atrás hacia adelante. La implementeación estandar no es adecuada. La solución es simple: agregue un campo adicional a los nodos con un puntero al nodo anterior.

En las listas doblemente encadenadas, las inserciones tienen lugar por ambos extremos (el final y el frente) y los borrados tienen lugar también por ambos extremos.

El puntero al anterior es particularmente util para el borrado del nodo final.

De ahora en más diremos "lista"para referirnos a las listas doblemente encadenadas.

Este TAD incluye las siguientes operaciones:

1. creo_vacia(s): crea una lista vacia.
2. es_vacia(s): retorna true si la lista es vacia, en otro caso retorna false.
3. primero(s): retorna la información del elemento al frente de la lista.
4. ultimo(s): retorna la información del elemento al final de la lista.
5. inserto_al_frente(s,a): inserta el elemento a al frente de la lista.
6. inserto_al_final(s,a): inserta el elemento a al final de la lista.
7. borrar_al_frente(s): quita el primer elemento de la lista.
8. borrar_al_final(s): quita el último elemento de la lista.
9. imprimo_de_frente_a_final(s): imprime los elementos de la lista del frente al final.
10. imprimo_de_final_a_frente(s): imprime los elementos en orden inverso.

Podemos implementar las operaciones utilizando las operaciones de listas lineales (no doblemente encadenadas) pero esto es particularmente ineficiente en lo que respecta a inserción y borrado del último nodo (se debe recorrer toda la lista).

Veremos una sola representación: con **listas encadenadas**.

La representación con arrays es ineficiente pues quitar el primero implica correr todos los elementos del array una posición.

1.1. Representación de listas doblemente encadenadas

La representación de la información es:

```
struct nodo {
        int x;
        nodo *ant;
        nodo *sig;
        };

typedef nodo *lista;

struct cabezal {
        lista primero;
        lista ultimo;
        };
```

La implementación de las operaciones es:

```
cabezal creo_vacia()
{         cabezal c;
          c.primero=NULL;
          c.ultimo=NULL;
          return c; }
```

es O(1) y Ω(1).

```
bool es_vacia(cabezal l)
{     return (l.primero==NULL); }
```

es O(1) y Ω(1).

```
int primero(cabezal l)
{   return (l.primero)->x; }
```

es O(1) y Ω(1).

```
int ultimo(cabezal l)
{   return (l.ultimo)->x; }
```

es O(1) y Ω(1).

```
void inserto_al_frente(cabezal & l,int a)
{
      lista l1;

      l1 = new nodo;
      l1->x = a;
      l1->sig = l.primero;
      if (l.primero ! = NULL)
```

```
        (l.primero)->ant = l1;
    else l.ultimo=l1;
    l1->ant = NULL;
    l.primero = l1;
}
```

es O(1) y Ω(1).

```
void inserto_al_final(cabezal & l,int a) {
    lista l1;

    l1 = new nodo;
    l1->x = a;
    l1->ant = l.ultimo;
    if (l.ultimo ! = NULL)
      (l.ultimo)->sig = l1;
    else l.primero=l1;
    l1->sig = NULL;
    l.ultimo = l1;
}
```

es O(1) y Ω(1).

```
bool borrar_al_frente(cabezal &l)
{   lista l1;

    l1=l.primero;
    if (es_vacia(l)) return false;
    else
        { ((l.primero)->sig)->ant=NULL;
          l.primero=(l.primero)->sig;
          delete l1;
          return true;
        }
}
```

es O(1) y Ω(1).

```
bool borrar_al_final(cabezal &l)
{   lista l1;

    l1=l.ultimo;
    if (es_vacia(l)) return false;
    else
        { ((l.ultimo)->ant)->sig=NULL;
          l.ultimo=(l.ultimo)->ant;
          delete l1;
          return true;
        }
}
```

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

```
void imprimo_de_frente_a_final(cabezal q)
{     lista l;

      if (es_vacia(q)) cout << "Lista vacia\n";
      else
           { l=q.primero;
             cout << "\n"<< "Lista:"<< "\n";
             while (l! =NULL)
             {
              cout << l->x << " ";
              l=l->sig;
             }
             cout << "\n";}
}
```

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

```
void imprimo_de_frente_a_final(cabezal q)
{     lista l;

      if (es_vacia(q)) cout << "Lista vacia\n";
      else
           { l=q.ultimo;
             cout << "\n"<< "Lista:"<< "\n";
             while (l! =NULL)
             {
              cout << l->x << " ";
              l=l->ant;
             }
             cout << "\n";}
}
```

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