Estructura de Datos y Algoritmos

Implementación de tipos de datos recurrentes lineales

1. El TAD Stack

Un stack es un caso particular de lista en la cual todas las inserciones y borrados tienen lugar en uno solo de los extremos que llamamos "tope".

Otro nombre para un stack es LIFO que significa last-in-first-out, o sea que el último insertado es el primero que se quita.

El TAD stack incluye las siguientes operaciones:

1. creo_vacio(s): crea un stack vacio. La operación es la misma que en el caso de listas generales.
2. es_vacio(s): retorna true si el stack es vacio, en otro caso retorna false.
3. top(s): retorna la información del elemento en el tope del stack.
4. pop(s): quita el elemento que está en el tope del stack.
5. push(s,x): inserta el elemento x en el tope del stack.
6. imprimo: lista los elementos del stack comenzando por el tope.

La implementación de listas que describimos trabaja para stacks porque un stack con sus operaciones es un caso especial de lista.

- top coincide con devolver el elemento de la posición 1.
- pop coincide con borrar el elemento de la posición 1.
- push coincide con insertar un elemento en posición 1.

Veremos dos representaciones: con **arreglos** y con **listas encadenadas**. Calcularemos los tiempos de las distintas operaciones.

1.1. Representación de stacks con arreglos

```
#define largomax 100;

struct stack{
        int datos[largomax];
        int tope;
        };
```

En la representación con arreglos necesitamos definir una función esta_lleno(s) que retorna true si el stack ocupo todo el espacio reservado. tope es la cantidad de elementos en el stack.

```
struct stack creo_vacio()
{       struct stack s;
        s.tope=0;
        return s;
}
```

es O(1) y Ω(1).

```
bool es_vacio(struct stack s)
{     return (s.tope==0); }
```

es O(1) y Ω(1).

```
bool esta_lleno(struct stack s)
{     return (s.tope==largomax); }
```

es O(1) y Ω(1).

```
bool push(struct stack *s,int x)
{
      if (esta_lleno(*s)) return false;
      else
           {(*s).datos[(*s).tope]=x;
            (*s).tope++;
            return true; }
}
```

es O(1) y Ω(1).

```
int top(struct stack s)
// precondicion : el stack es no vacio
{   return s.datos[s.tope - 1]; }
```

es O(1) y Ω(1).

```
void pop(struct stack *s)
// precondicion : el stack es no vacio
{     (*s).tope = (*s).tope - 1;
}
```

es O(1) y Ω(1).

```
struct stack imprimo (struct stack s)
{       struct stack l;
        int i,j;
```

```
        bool b;

        if (es_vacio(s)) {cout << "Stack Vacio\n";
                           return creo_vacio();
                          }
        else {
              j=top(s);
              pop(&s);
              cout << j << "\n";
              l=imprimo(s);
              b=push(&l,j);
              return l;
             }
}
```

El tiempo es O(n) y $\Omega(n)$. Se cumple la recurrencia $T(0) = 1$, $T(1) = 1$ y $T(n) = T(n-1) + 1)$

```
void imprimo (struct stack s)
{
      int i;

      for (i=s.tope-1;i ≥ 0;i - - )
          cout << s.datos[i] << endl;
}
```

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

1.2. Representación de stacks con listas encadenadas

La representación de los datos es:

```
struct nodo_stack {
        int x;
        nodo_stack *sig;};

typedef nodo_stack *stack;
```

la implementación de las operaciones:

```
stack creo_vacio()
{      return NULL; }
```

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

```
bool es_vacio(stack s)
{      return (s==NULL); }
```

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

```
int top (stack l)
// precondicion : el stack es no vacio
{   return l->x; }
```

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

```
void pop (stack &l)
// precondicion : el stack es no vacio
{     stack q;
      q=l;
      l=l− >sig;
      delete q;
}
```

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

```
void push (stack &l,int a)
{     stack q;

      q = new nodo_stack;
      q->x=a;
      q->sig=l;
      l=q; }
```

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

```
void imprimo(stack q)
{     if (es_vacio(q)) cout << "Stack vacio\n";
      else
           {cout << "\n"<< "Stack:"<< "\n";
            while (q! =NULL)
            {
              cout << q->x << endl;
              q=q->sig;
            }
            cout << "\n";}
}
```

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