Villar Pedro

Algoritmos 2 - TP

Apunte

CAPÍTULO

ÍNDICE GENERAL

CAPÍTULO 1

INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN DE LA MATERIA

El lenguaje a utilizar está inspirado en el lenguaje imperativo *Pascal*, al cual se le han ido realizando modificaciones de acuerdo a las necesidades didácticas de la materia. Se utilizará principalmente para la incorporación de conceptos tales como análisis de algoritmos, definición de tipos abstractos de datos, o la comprensión de distintas técnicas de programación.

1.1 TIPOS DE DATOS

En este lenguaje, existen varios tipos de datos, incluyendo:

- `int`: para valores enteros,
- `real`: para valores decimales,
- `bool`: para valores de verdad (verdadero o falso),
- `char`: para caracteres.

Además, se pueden crear estructuras de datos más complejas como arreglos para agrupar elementos del mismo tipo.

1.1.1 Declaración de variables

Las variables se declaran utilizando la palabra clave `var`, seguida de una lista de variables separadas por comas. Cada variable debe tener un tipo asociado. Por ejemplo:

Declaración de variables

```
{- Variable entera -}
var i : int
i := 10
{- Variable real -}
var x : real
x := 3.14
{- Variable booleana -}
var b : bool
b := true
{- Variable caracter -}
var c : char
c := 'a'
```

Código 1

1.2 OPERADORES

Los operadores aritméticos básicos son +, -, * y /. Además, se pueden utilizar los operadores de comparación <, >, <=, >=, == y !=.

Uso de operadores

```
{- Operadores aritmeticos -}
var a : int
a := 10 + 5
a := 10 - 5
a := 10 * 5
a := 10 / 5
{- Operadores de comparacion -}
var b : bool
b := 10 < 5
b := 10 > 5
b := 10 <= 5
b := 10 >= 5
b := 10 == 5
b := 10 != 5
```

Código 2

Junto con las constantes lógicas `true` y `false`, se pueden utilizar los operadores lógicos `&&`, `||` y `!`.

Uso de operadores lógicos

```
{- Operadores logicos -}
var b : bool
b := true && false
b := true || false
b := !true
```

Código 3

No hay operadores definidos para el tipo `char`.

1.3 TIPOS DE DATOS ESTRUCTURADOS

Un tipo estructurado permite representar colecciones de otros tipos de datos. De manera similar a los tipos básicos, se definen operaciones específicas para acceder a los elementos que conforman al tipo. En el lenguaje solo tenemos definidos de forma nativa a los arreglos.
Para definir un arreglo es necesario detallar el tipo de sus componentes y los tamaños para cada una de sus dimensiones, los cuales deberán ser mayores a cero.

Declaración de arreglos

```
{- Arreglo de enteros -}
var a : array [1..10] of int
a[1] := 10
{- Arreglo de reales -}
var b : array [1..10] of real
b[1] := 3.14
{- Arreglo de booleanos -}
var c : array [1..10] of bool
c[1] := true
{- Arreglo de caracteres -}
var d : array [1..10] of char
d[1] := 'a'
```

Código 4

En este ejemplo, se declaran arreglos de 10 elementos de distintos tipos. Luego, se asigna un valor a la primera posición de cada arreglo.

1.4 DEFINICIÓN DE TIPOS

El lenguaje permite definir tipos de datos personalizados. Esto es útil para abstraer la representación de un concepto y facilitar la comprensión del código. Por ejemplo, se puede definir un tipo `punto` para representar un punto en el plano cartesiano.

Definición de tipos

```
type punto = tuple
                x : real
                y : real
                end tuple
```

Código 5

En este ejemplo, se define un tipo `punto` que contiene dos campos `x` e `y` de tipo `real`. Luego, se pueden declarar variables de tipo `punto` y asignarles valores.

1.5 TIPOS ENUMERADOS

Un tipo enumerado representa un conjunto finito de valores. Cada valor está definido mediante un identificador único. Para declarar un tipo enumerado se emplean las palabras claves **enumerate** y **end enumerate**. Por ejemplo, definamos un tipo enumerado para los días de la semana.

Definición de tipos enumerados

```
type day = enumerate
            Lunes
            Martes
            Miercoles
            Jueves
            Viernes
            Sabado
            Domingo
            end enumerate
```

Código 6

Ya una vez definido el tipo enumerado, se pueden declarar variables de este tipo y asignarles valores.

Uso de tipos enumerados

```
var d : day
d := Lunes
```

Código 7

1.6 SINÓNIMOS DE TIPOS

Un sinónimo de tipo es una forma de referirse a un tipo de datos con un nombre diferente. Por ejemplo, se puede definir un sinónimo de tipo `real` para representar la temperatura en grados Celsius.

Definición de sinónimos de tipos

```
type celsius = real
```

Código 8

No necesariamente los sinónimos de tipo deben ser de tipos básicos, también se pueden definir sinónimos de tipos estructurados.

Definición de sinónimos de tipos estructurados

```
type matrizdereales = array [1..10] of real
```

Código 9

Una expresión de este tipo, es útil cuando se utiliza una función donde se espera un valor de uno de los sinónimos de su tipo. En el siguiente ejemplo se declara una variable `mR` del tipo `matrizdereales`, y se opera de manera transparente como si fuese un arreglo tradicional.

Uso de sinónimos de tipos

```
var mR : matrizdereales
for i := 0 to 9 do
    for j := 0 to 9 do
        mR[i, j] := 0.0
    od
od
```

Código 10

1.7 TUPLAS

Una tupla es un tipo de dato estructurado que permite agrupar un número finito de elementos de distintos tipos. Para definir una tupla se emplean las palabras claves **tuple** y **end tuple**. Por ejemplo, definamos una tupla para representar los datos de una persona.

Definición de tuplas

```
type persona = tuple
                nombre : string
                edad : int
                altura : real
                end tuple
```

Código 11

Y para darle valor a una variable de tipo `persona` se hace de la siguiente manera.

Uso de tuplas

```
var p : persona
p.nombre := "Juan"
p.edad := 20
p.altura := 1.80
```

Código 12

1.8 FUNCIONES Y PROCEDIMIENTOS

Las funciones y procedimientos son bloques de código que pueden ser invocados desde otros bloques de código. La diferencia entre ambos radica en que las funciones devuelven un valor, mientras que los procedimientos no. La sintaxis para definir funciones y procedimientos es la siguiente:

Definición de funciones

```
fun nombre (p1 : T1, p2 : T2, ... , pn : Tn) ret r : T
    {- Cuerpo de la funcion -}
end fun
```

Código 13

Esta función toma los parametros p1, p2, ..., pn de tipos T1, T2, ..., Tn respectivamente, y devuelve un valor de tipo T. Por ejemplo, definamos una función que sume dos números enteros.

Función suma

```
fun suma (a : int, b : int) ret r : int
    ret := a + b
end fun
```

Código 14

En el siguiente ejemplo se muestra la implementación de la función `factorial`, que calcula el factorial de un número entero positivo `n`. La variable de retorno fact almacena la productoria de números, y al finalizar la ejecución de la función, se retorna su valor al contexto donde se efectuó la llamada. El comentario simplemente indica la precondición ha satisfacer para garantizar el comportamiento esperado de la función.

Función factorial

```
{- PRE : n >= 0 -}
fun factorial (n : int) ret fact : int
    fact := 1
    for i := 2 to n do
        fact := fact * i
    od
end fun
```

Código 15

Un procedimiento realiza una computación de acuerdo a un conjunto de parámetros de lectura, para modificar un conjunto de parámetros de escritura. Su comportamiento es determinado solamente por los parámetros que recibe donde cada uno lleva un decorado que indica si es de lectura `in`, de escritura `out`, o ambas `in/out`. Un procedimiento no modifica el estado de los parámetros de lectura, y tampoco consulta el estado de los parámetros de escritura.
En el siguiente ejemplo se implementa el procedimiento initialize, el cual se encarga de inicializar un arreglo de enteros según un valor determinado. Lo interesante a destacar en este ejemplo es la forma en que se manejan los parámetros de lectura y escritura. El parámetro de lectura solo ocurre del lado derecho de la asignación, mientras que el parámetro de escritura solo ocurre del lado izquierdo. Esto significa que al llamar a la función initialize, se pasarán los parámetros como argumentos de lectura (para ser utilizados dentro de la función) y como argumentos de escritura (para almacenar el resultado de la función).

Procedimiento initialize

```
proc initialize ( in e : int , out a : array [ 10 ] of int )
    for i := 9 downto 0 do
        a [ i ] := e
    od
end proc
```

Código 16

1.9 RECURSIÓN

El lenguaje soporta la recursión, es decir, una función o procedimiento puede llamarse a sí mismo. Por ejemplo, definamos una función que calcule el factorial de un número de manera recursiva. Por ejemplo definamos la función factorial de la siguiente manera.

Función factorial recursiva

```
{- PRE : n >= 0 -}
fun factorial (n : int) ret fact : int
    if n >= 2 then
        fact := n * factorial(n - 1)
    else
        fact := 1
    fi
end fun
```

Código 17

1.10 POLIMORFISMO PARAMÉTRICO

El lenguaje soporta el polimorfismo paramétrico, es decir, la capacidad de definir funciones y procedimientos que operan sobre un rango de tipos. Por ejemplo, definamos una función que intercambie los valores de dos variables de cualquier tipo.

Función swap

```
proc swap ( in / out a : array [ n ] of T , in i , j : int )
    var temp : T
    temp := a[i]
    a[i] := a[j]
    a[j] := temp
end fun
```

Código 18

1.11 POLIMORFISMO *AD HOC*

El lenguaje soporta el polimorfismo *Ad Hoc*, es decir, la capacidad de definir funciones y procedimientos que podrían ser implementados de manera genérica pero no para cualquier tipo de datos, sino para ciertos tipos que comparten alguna característica, en consecuencia no es posible utilizar polimorfismo paramétrico. Consideremos las siguientes implementaciones de `belongs` y `selectionSort`. La primera decide si un valor determinado pertenece a un arreglo y la segunda permite ordenar un arreglo de menor a mayor.

Función belongs

```
fun belongs ( e : int , a : array [ n ] of int ) ret b : bool
var i : int
i := 0
b := false
    while ! b && i < n do
        b := a [ i ] == e
        i := i + 1
    od
end fun
```

Código 19

Procedimiento selectionSort

```
proc selectionSort ( in / out a : array [ n ] of int )
var minPos : int
    for i := 0 to n - 1 do
        minPos := i
        for j := i + 1 to n - 1 do
            if a [ j ] < a [ minPos ] then minPos := j fi
        od
        swap ( a , i , minPos )
    od
end proc
```

Código 20

En ambas implementaciones los elementos son de tipo entero. Sin embargo podríamos definir las mismas operaciones para otros tipos de datos, como por ejemplo, caracteres. Se tendrían que redefinir las anteriores con los nombres `belongsInt` y `selectionSortInt`, y declarar de manera idéntica las operaciones `belongsChar` y `selectionSortChar` donde solo cambiaríamos int por char en los tipos de los parámetros. Con un trabajo tediosamente repetitivo se podrían dar declaraciones para todos los tipos que tengan definidas las operaciones de comparación; aunque no sería posible para aquellos que no las tengan definidas. El *polimorfismo ad hoc* nos permite escribir de manera genérica una función o un procedimiento donde la tarea que realiza sólo está bien definida para algunos tipos. Además esta tarea puede no ser la misma dependiendo del tipo.
En el lenguaje se definen una serie de clases las cuales pueden ser pensadas como una especie de interfaz que caracteriza algún comportamiento. Un tipo es una instancia de una clase, cuando implementa el comportamiento que la clase describe. El lenguaje sólo incorpora de forma nativa las clases **Eq** y **Ord**, y no existe posibilidad de declarar nuevas clases. La primera representa a los tipos que tienen alguna noción de igualdad, y sus operaciones definidas comprenden al == y !=. La segunda representa a los tipos que poseen alguna relación de orden, y sus operaciones definidas comprenden al <, <=, >=, >.

1.12 MEMORIA DINÁMICA

El lenguaje permite manipular explícitamente la memoria dinámica mediante un tipo de datos especial, que llamaremos puntero. Supongamos que deseamos definir el tipo correspondiente a las listas. Una lista permite representar una colección ordenada de elementos de algún tipo de datos, cuyo tamaño es variable; lo cual significa que su tamaño crece tanto como sea necesario, de acuerdo a la cantidad de elementos almacenados. Todos los tipos presentados hasta el momento utilizan una cantidad fija de memoria, la cual no puede ser modificada en tiempo de ejecución. Recordemos una vez más que los arreglos implementados en el lenguaje tienen un tamaño fijo al momento de la ejecución. En este aspecto el uso de punteros resulta fundamental, ya que permiten reservar y liberar memoria en la medida que sea necesario durante la ejecución del programa.

1.12.1 Punteros

Un puntero es una variable que almacena la dirección de memoria de otra variable. En el lenguaje, los punteros se representan con el tipo **pointer**, indica el lugar de memoria donde se encuentra almacenado el valor de la variable. La sintaxis para declarar un puntero es la siguiente:

Declaración de punteros

```
var p : pointer of int
```

Código 21

1.12.2 Ejemplo

Ejemplo de uso de punteros

```
type node of ( T ) = tuple
                        elem : T ,
                        next : pointer of node of ( T )
                        end tuple

type list of ( T ) = pointer of node of ( T )
```

Código 22

En el ejemplo anterior se declara una lista enlazada denominada list, la cual se compone de una sucesión de nodos node que se integran por los campos elem de cierto tipo paramétrico T, y next el cual referencia al siguiente nodo en la lista.

1.12.3 Operaciones con punteros

En el lenguaje se definen tres operaciones para manipular punteros. El procedimiento nativo `alloc` toma una variable de tipo puntero, y le asigna la dirección de un nuevo bloque de memoria, cuyo tamaño estará determinado por el tipo de la variable. El operador `#` permite acceder al bloque de memoria apuntado por el puntero. El procedimiento nativo **free** toma una variable de tipo puntero, y libera el respectivo bloque de memoria referenciado.
Retomando el ejemplo de la lista enlazada, los procedimientos `empty` y `addL` son utilizados para construir valores del tipo en cuestión.

Procedimiento empty

```
proc empty ( out l : list of ( T ) )
    l := null
end proc
```

Código 23

El procedimiento empty construye una lista vacía. La constante null representa un puntero que no referencia un lugar de memoria válido. En el ejemplo, la constante representa una lista que no posee ningín nodo.

Procedimiento addL

```
proc addL ( in e : T , in / out l : list of ( T ) )
    var p : pointer of node of ( T )
    alloc ( p )
    p -> elem := e
    p -> next := l
    l := p
end proc
```

Código 24

CAPÍTULO

2 ORDENACIÓN ELEMENTAL

Generalmente, se considera que la ordenación de un conjunto de elementos es la disposición de los mismos en un orden determinado. En el caso de los algoritmos de ordenación, se busca que los elementos de un conjunto se encuentren en un orden específico, ya sea ascendente o descendente. En este capítulo se presentan los algoritmos de ordenación más simples, los cuales son útiles para ordenar conjuntos de elementos pequeños.

Figura 2.1: Ordenación de arreglos

La principal condición a imponer a los métodos de ordenación de arreglos es la utilización económica de la memoria disponible. Esto implica que las permutaciones de items, con vistas a su ordenación, deben realizarse utilizando el espacio ocupado por el arreglo y que los métodos que transportan artículos de un array a hacia un array resultado b son intrínsecamente de menor interés.

2.1 ORDENACIÓN POR SELECCIÓN

Este método se basa en los siguientes principios:

1. Seleccionar el menor elemento.
2. Intercambiarlo con el primer elemento.

Y luego se repite el proceso con el resto del arreglo, es decir, se selecciona el menor elemento del subarreglo restante y se intercambia con el segundo elemento, y así sucesivamente.

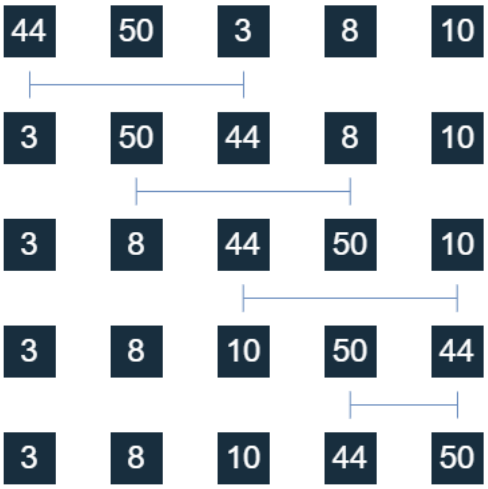

Figura 2.2: Ejemplo de proceso de ordenación por selección

Para el algoritmo de ordenación por selección, se ha desarrollado un procedimiento `selection_sort` que recibe un arreglo de tamaño n y lo ordena. El procedimiento `selection_sort` utiliza dos funciones auxiliares: `min_pos_from` y `swap`. La función `min_pos_from` recibe un arreglo y una posición, y retorna la posición del menor elemento a partir de la posición dada. La función `swap` recibe un arreglo y dos posiciones, e intercambia los elementos en dichas posiciones. Se pueden marcar las siguientes observaciones:

- `selection_sort` utiliza un ciclo `for` que recorre el arreglo desde la primera posición hasta la última. En cada iteración, se busca el menor elemento a partir de la posición actual y se intercambia con el elemento en la posición actual,
- se encuentra una llamada a la función `min_pos_from` que recibe el arreglo y la posición actual, y retorna la posición del menor elemento a partir de la posición actual,
- y se recibe tambien una llamada al procedimiento `swap` que recibe el arreglo y las posiciones actual y la posición del menor elemento, e intercambia los elementos en dichas posiciones,
- encontramos una **comparación** entre elementos de un arreglo, y una **asignación** de elementos de un arreglo,
- la operación que mas se repite es la comparación de elementos de un arreglo, y toda operación se repite a lo sumo de manera proporcional a esa,
- como la celda de un arreglo es constante, su costo no depende de cual es la celda o del tamaño del arreglo, por lo que el costo de la operación es constante.

2.1.1 Código

Ordenación por selección

```
proc selection_sort (in/out a: array[1..n] of T)
    var minp: nat
    for i := 1 to n do
        minp := min_pos_from(a, i)
        swap(a, i, minp)
    od
end proc

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    minp := i
    for j:= i+1 to n do
        if a[j] < a[minp] then
            minp:= j
        fi
    od
end fun

proc swap (in/out a: array[1..n] of T, in i, j: nat)
    var temp: T
    temp := a[i]
    a[i] := a[j]
    a[j] := temp
end proc
```

Código 25

2.1.2 Análisis de complejidad

- Al llamar a la función `min_pos_from` se realiza una cantidad de operaciones proporcional a $n-i$,
- `selection_sort` llama a `min_pos_from(a,i)` para $i = 1,2,\ldots,n-1$,
- por lo tanto, en total son $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$ comparaciones.

2.2 ORDENACIÓN POR INSERCIÓN

Este método se basa en los siguientes principios:

1. Seleccionar el primer elemento del arreglo.
2. Insertarlo en la posición correcta.

Y luego se repite el proceso con el resto del arreglo, es decir, se selecciona el siguiente elemento y se inserta en la posición correcta.

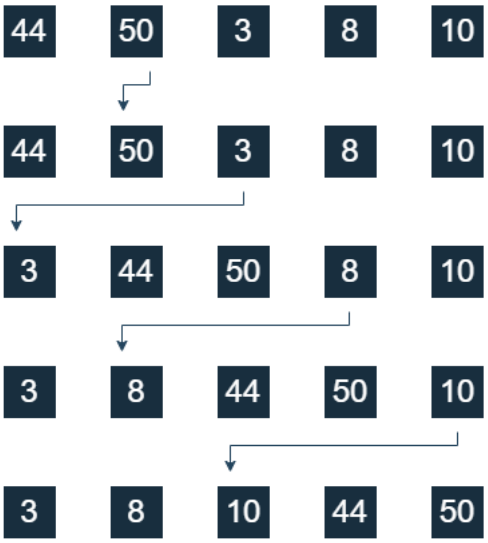

Figura 2.3: Ejemplo de proceso de ordenación por inserción

Para el algoritmo de ordenación por inserción, se ha desarrollado un procedimiento `insertion_sort` que recibe un arreglo de tamaño n y lo ordena. El procedimiento `insertion_sort` utiliza un procedimiento auxiliar `insert`. El procedimiento `insert` recibe un arreglo y una posición, y mueve el elemento en la posición dada a la posición correcta.

- `insertion_sort` utiliza un ciclo `for` que recorre el arreglo desde la segunda posición hasta la última. En cada iteración, se llama al procedimiento `insert` que recibe el arreglo y la posición actual, y mueve el elemento en la posición actual a la posición correcta,
- se encuentra una llamada al procedimiento `insert` que recibe el arreglo y la posición actual, y mueve el elemento en la posición actual a la posición correcta.

2.2.1 Código

Ordenación por inserción

```
proc insertion_sort (in/out a: array[1..n] of T)
    for i:= 2 to n do
        insert(a,i)
    od
end proc
proc insert (in/out a: array[1..n] of T, in i: nat)
    var j: nat
    j:= i
    do j > 1 && a[j] < a[j - 1] ->
        swap(a,j-1,j)
        j:= j-1
    od
end proc
```

Código 26

2.2.2 Análisis de complejidad

- En el peor caso, el arreglo está ordenado en forma inversa, por lo que se deben realizar $i-1$ comparaciones en la posición i,
- `insertion_sort` llama a `insert(a,i)` para $i = 2,3,\ldots,n$,
- por lo tanto, en total son $1+2+\ldots+n-1 = \frac{n(n-1)}{2}$ comparaciones.

CAPÍTULO

3 ORDENACIÓN AVANZADA

En este caso se presentan algoritmos de ordenación más eficientes que los vistos en el capítulo anterior. Estos algoritmos son más eficientes en términos de tiempo de ejecución y se basan en la técnica de dividir y conquistar. Podría verse mas como una ordenación de ficheros.

Figura 3.1: Ordenación de ficheros

3.1 ORDENACIÓN POR MEZCLA (MERGE SORT)

El algoritmo de ordenación por mezcla es un algoritmo recursivo que se basa en la técnica de dividir y conquistar. Este algoritmo consiste en dividir el arreglo a ordenar en varias partes, estas partes se ordenan y, posteriormente, se mezclan entre ellas de forma ordenada. Dado que el proceso de mezcla es menos costoso que el proceso de ordenación obtenemos un algoritmo más eficiente.
El algoritmo de ordenación por mezcla se basa en los siguientes principios:

1. Dividir recursivamente el arreglo en 2 partes.
2. Si el arreglo tiene un solo elemento, entonces está ordenado por definición.
3. Si la lista tiene más de un ítem, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades.
4. Una vez que ambas mitades estén ordenadas, la operación de mezcla se encarga de unir ambas mitades en un solo arreglo ordenado.

El proceso de división puede verse de la siguiente manera:

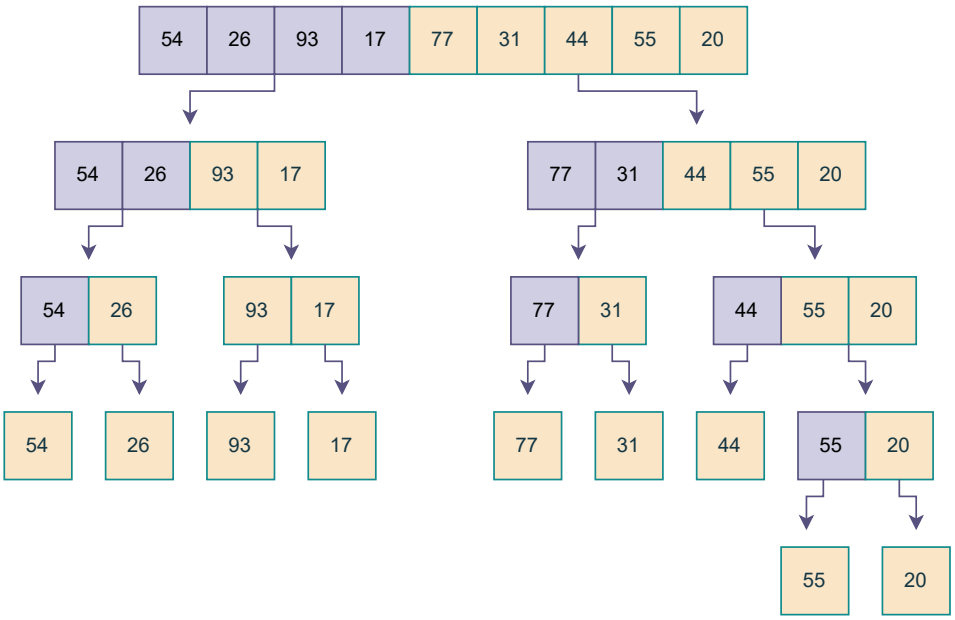

Figura 3.2: Ejemplo de proceso de división

El proceso de mezcla puede verse de la siguiente manera:

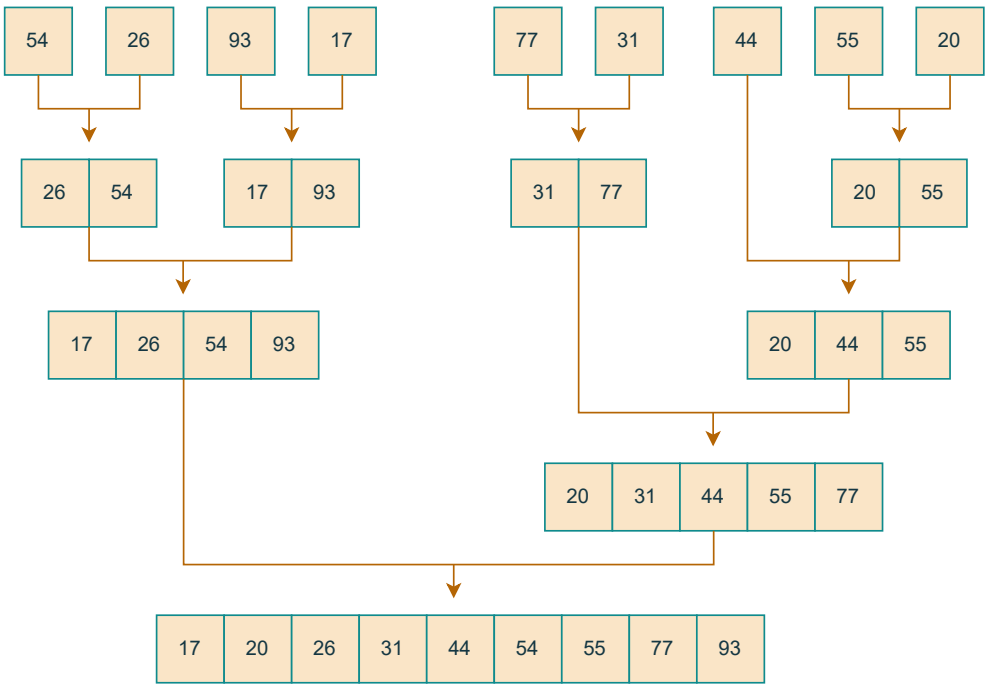

Figura 3.3: Ejemplo de proceso de mezcla

3.1.1 Idea de crear el código

La idea es definir un procedimiento al que le pasamos en qué parte del arreglo queremos hacer lo fundamental del mergesort: dividirlo en dos, ordenar cada mitad y luego intercalar las dos mitades. Este procedimiento es `merge_sort_rec`, que toma el arreglo a y las posiciones inicial y final del pedazo de arreglo que vamos a ordenar. El procedimiento principal llama al recursivo con los índices 1 y n (el arreglo completo).

Estructura de la función merge_sort

```
proc merge_sort(in/out a: array[1..n] of T)
    merge_sort_rec(a,1,n)
end proc
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
...
end proc
```

Código 26

El procedimiento `merge_sort_rec` toma el arreglo a, y los índices **lft** y **rgt**, que corresponden con el comienzo y el final del pedazo de arreglo que queremos ordenar. Recordando la idea del algoritmo, el caso más simple es cuando el pedazo de arreglo tiene un solo elemento. En nuestra implementación eso corresponde a que lft sea igual a rgt. En ese caso el procedimiento no debe hacer nada, ya que el pedazo está trivialmente ordenado. En caso que no se dé esa situación, debemos:

1. Dividir el pedazo de arreglo en dos,
2. ordenar cada una de esas mitades utilizando el mismo algoritmo, e
3. intercalar cada mitad ordenada.

Para dividir el pedazo de arreglo, se define una variable **mid** de tipo nat a la que se le asigna el índice correspondiente a la posición del medio.

División del arreglo

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    ...
end proc
```

Código 27

Ahora entonces hay que llamar recursivamente al procedmiento dos veces: una para la primera mitad que irá desde la posición **lft** hasta **mid**, y otra para la segunda mitad, que irá desde la posición **mid+1** hasta **rgt**.

Llamada recursiva

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    merge_sort_rec(a,lft,mid)
    merge_sort_rec(a,mid+1,rgt)
    ...
end proc
```

Código 28

y por último, hay que intercalar. Esta tarea la implementaremos con un procedimiento llamado `merge`, que se define mas adelante.

Definición completa de la función merge_sort_rec

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    merge_sort_rec(a,lft,mid)
    merge_sort_rec(a,mid+1,rgt)
    merge(a,lft,mid,rgt)
end proc
```

Código 29

Ahora para implementar el procedimiento de **intercalación**, se necesita un arreglo auxiliar, en donde se van a guardar los valores de la primera mitad a intercalar. Se define entonces una variable de tipo array, dos variables en las que luego almacenaremos índices `j` y `k`, y se copia la primera mitad del arreglo en el arreglo auxiliar:

Inicialización de variables

```
proc merge(in/out a: array[1..n] of T, in lft,mid,rgt: nat)
    var tmp: array[1..n] of T
    var j,k: nat
    for i:=lft to mid do ->
        tmp[i] := a[i]
    od
    ...
end proc
```

Código 30

Los índices `j` y `k` indicarán respectivamente el elemento de la primera mitad que estoy analizando para insertar en el pedazo de arreglo que quedará ordenado, y el índice de la segunda mitad que estoy analizando. Inicialmente observo el primero de cada mitad, es decir **lft** y **mid+1**.

Inicialización de índices

```
proc merge(in/out a: array[1..n] of T, in lft,mid,rgt: nat)
    var tmp: array[1..n] of T
    var j,k: nat
    for i:=lft to mid do ->
        tmp[i] := a[i]
    od
    j := lft
    k := mid+1
    ...
end proc
```

Código 31

Ahora hay que rellenar el pedazo completo de arreglo que contendrá las dos mitades intercaladas ordenadamente. Lo recorro con un for desde lft hasta rgt. Y se puede ver que en cada paso si el elemento que estoy observando de la primera mitad es menor o igual que el de la segunda mitad, de acuerdo a esa comparación

sabré qué elemento va a ubicarse en el arreglo ordenado.

Intercalación

```
proc merge(in/out a: array[1..n] of T, in lft,mid,rgt: nat)
    var tmp: array[1..n] of T
    var j,k: nat
    for i:=lft to mid do ->
        tmp[i] := a[i]
    od
    j := lft
    k := mid+1
    for i:=lft to rgt do ->
        if j <= mid && (k > rgt || tmp[j] <= a[k])
            then a[i] := tmp[j]
                 j:= j+1
            else a[i] := a[k]
                 k := k+1
        fi
    od
end proc
```

Código 32

En la guarda del if hay que considerar también el caso en que ya haya agotado todos los elementos de la segunda mitad, lo que sucederá cuando `k >rgt`, y entonces en ese caso también completo con los elementos de la primera mitad (es decir los que están en el arreglo auxiliar).

3.1.2 Análisis de complejidad

Para analizar la función, debemos considerar los dos procesos distintos que conforman su implementación. En primer lugar, la lista se divide en mitades. Podemos dividir una lista por la mitad en un tiempo $\log n$ donde n es la longitud de la lista. El segundo proceso es la mezcla. Cada ítem de la lista se procesará y se colocará en la lista ordenada. Así que la operación de mezcla que da lugar a una lista de tamaño n requiere n operaciones. El resultado de este análisis es que se hacen $\log n$ divisiones, cada una de las cuales cuesta n para un total de $n \log n$ operaciones. Un ordenamiento por mezcla es un algoritmo $O(n \log n)$.

3.1.3 Código

Ordenación por mezcla (Merge Sort)

```
proc merge_sort(in/out a: array[1..n] of T)
    merge_sort_rec(a,1,n)
end proc

proc merge_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
        merge_sort_rec(a,lft,mid)
        merge_sort_rec(a,mid+1,rgt)
        merge(a,lft,mid,rgt)
    fi
end proc

proc merge(in/out a: array[1..n] of T, in lft,mid,rgt: nat)
    var tmp: array[1..n] of T
    var j,k: nat
    for i:=lft to mid do ->
        tmp[i] := a[i]
    od
    j := lft
    k := mid+1
    for i:=lft to rgt do ->
        if j <= mid && (k > rgt || tmp[j] <= a[k])
            then a[i] := tmp[j]
                 j:= j+1
            else a[i] := a[k]
                 k := k+1
        fi
    od
end proc
```

Código 33

3.2 ORDENACIÓN RÁPIDA (QUICK SORT)

El ordenamiento rápido usa dividir y conquistar para obtener las mismas ventajas que el ordenamiento por mezcla, pero sin utilizar almacenamiento adicional. Sin embargo, es posible que la lista no se divida por la mitad. Cuando esto sucede, el desempeño disminuye.
El algoritmo de ordenación rápida se basa en los siguientes principios:

1. Primero selecciona un valor, que se denomina el **pivot** (El papel del valor pivote es ayudar a dividir la lista).

2. La posición real a la que pertenece el valor pivote en la lista final ordenada, comúnmente denominado punto de división, se utilizará para dividir la lista para las llamadas posteriores a la función de ordenamiento rápido.

3. El proceso de partición sucederá a continuación. Encontrará el punto de división y al mismo tiempo

moverá otros ítems al lado apropiado de la lista, según sean menores o mayores que el valor pivote.

A modo ilustrativo, se puede ver el proceso de partición en la siguiente figura:

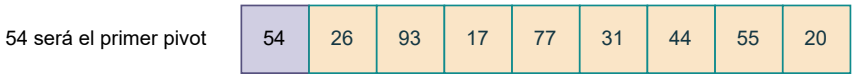

Figura 3.4: Ejemplo de proceso de partición

El particionamiento comienza localizando dos marcadores de posición `lft` y `rgt` al principio y al final de los ítems restantes de la lista. El objetivo del proceso de partición es mover ítems que están en el lado equivocado con respecto al valor pivote mientras que también se converge en el punto de división.

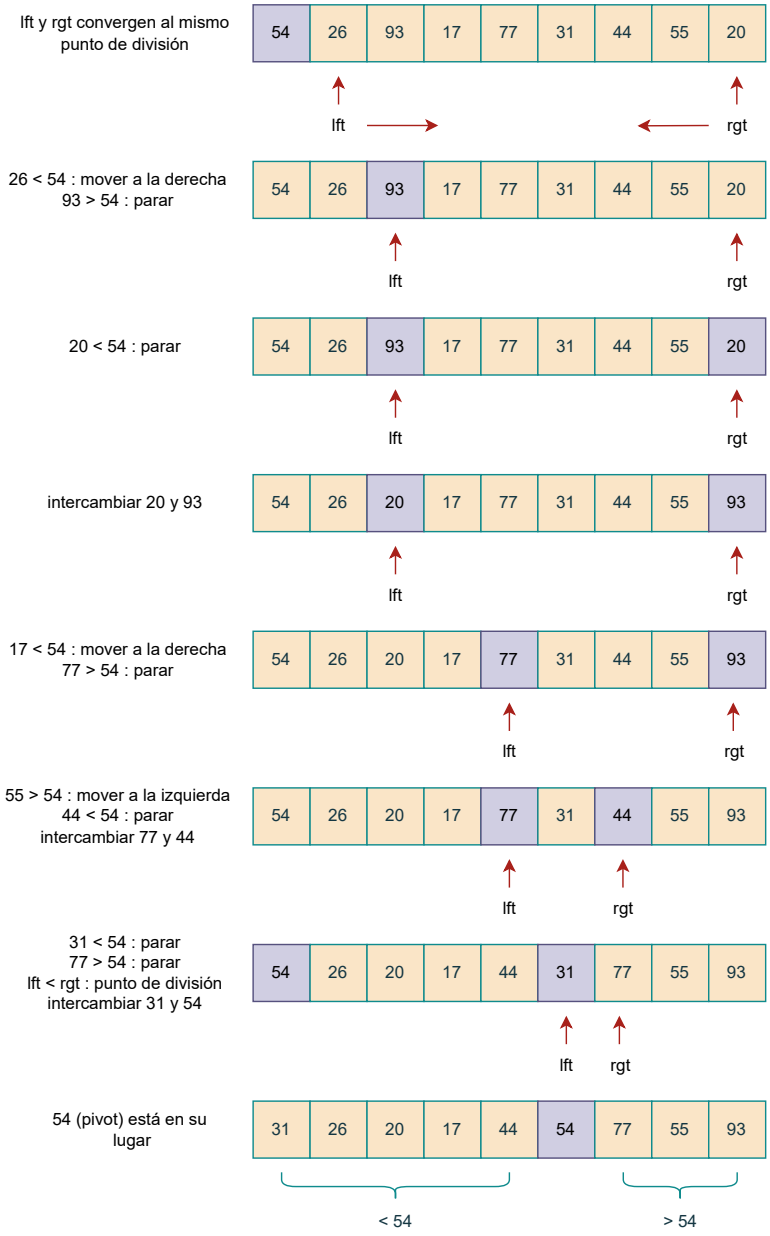

Figura 3.5: Ejemplo de proceso

- Comienza incrementando `lft` hasta que localicemos un valor que sea mayor que el valor pivote. Luego decrementamos `rgt` hasta que encontremos un valor que sea menor que el valor pivote. En tal punto habremos descubierto dos ítems que están fuera de lugar con respecto al eventual punto de división.

- Para este ejemplo, esto ocurre en 93 y 20. Ahora se deben intercambiar estos dos ítems y luego repetir el proceso de nuevo.
- Se detiene en el punto donde `rgt` se vuelva menor que `lft`. La posición de `lft` es ahora el punto de división. El valor pivote se puede intercambiar con el contenido del punto de división y el valor pivote está ahora en su lugar.

La lista ahora se puede dividir en el punto de división y el ordenamiento rápido se puede invocar recursivamente para las dos mitades.

3.2.1 Idea de crear el código

De manera similar al algoritmo de ordenación `merge_sort`, se define un procedimiento recursivo que tomará el arreglo de elementos y dos índices correspondientes al pedazo de arreglo que se ordenará. El algoritmo principal llama a este procedimiento con los índices `1` y `n`, correspondiendo con la ordenación del arreglo completo.

Estructura de la función quick_sort

```
proc quick_sort(in/out a: array[1..n] of T)
    quick_sort_rec(a,1,n)
end proc
proc quick_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
...
end proc
```

Código 34

Este procedimiento recursivo tiene su caso más simple cuando **lft** y **rgt** son iguales, lo que significa que estoy ordenando un arreglo de un solo elemento. En el caso interesante, al procedimiento se lo llama partition que será el encargado de acomodar los elementos del pedazo de arreglo utilizando el elemento de más a la izquierda como **pivot**.

Llamada a la función partition

```
proc quick_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var ppiv: nat
    if rgt > lft ⟶
        partition(a,lft,rgt,ppiv)
    ...
...
end proc
```

Código 35

El procedimiento `partition` modifica el arreglo desde `lft` hasta `rgt` dejando al comienzo todos los elementos que son menores o iguales al que se encontraba originalmente en la posición `lft`, y al final a todos los que son mayores o iguales. También modifica la variable `ppiv` asignándole el índice correspondiente al lugar donde queda ubicado definitivamente el elemento que se usó como **pivot**. Luego lo único que queda por hacer es llamar recursivamente al procedimiento, una vez para los elementos que quedaron acomodados a la izquierda del pivot, y otra vez para los elementos que quedaron acomodados a la derecha.

Llamada recursiva

```
proc quick_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var ppiv: nat
    if rgt > lft —>
        partition(a,lft,rgt,ppiv)
        quick_sort_rec(a,lft,ppv-1)
        quick_sort_rec(a,ppiv+1,rgt)
    fi
end proc
```

Código 36

Queda por ver el procedimiento partition. Toma el arreglo, los dos índices que indican qué fragmento estamos ordenando, y una variable de solo escritura, en la cual indicaremos el índice en donde queda el pivot una vez que finalice el procedimiento.

Estructura de la función partition

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
...
end proc
```

Código 37

Lo que hay que hacer en este procedimiento es ir mirando con un índice los elementos que están a la izquierda y con otro los que están a la derecha. El índice `i` indicará el elemento que estoy mirando desde la izquierda, y respectivamente `j` indicará el de la derecha. El elemento tomado como **pivot** será el que está más a la izquierda.

Inicialización de variables

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    ppiv:= lft
    i:= lft+1
    j:= rgt
    ...
end proc
```

Código 38

Luego hay que ir viendo si el elemento indicado con el índice `i` y el indicado con `j` están bien ubicados, es decir, si `a[i]` es menor o igual al **pivot**, y si `a[j]` es mayor o igual. En caso que sea así, `.ªvanzo"` el índice. Este avance corresponde a sumar uno para `i`, y a restar uno para `j`. En caso que el elemento de la izquierda esté mal ubicado, debemos encontrar un elemento de la derecha que también esté mal ubicado, y los intercambiamos.

Particionamiento

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    ppiv:= lft
    i:= lft+1
    j:= rgt
    do i <= j -->
        if a[i] <= a[ppiv] --> i:= i+1
           a[j] >= a[ppiv] --> j:= j-1
           a[i] > a[ppiv] && a[j] < a[ppiv] --> swap(a,i,j)
        fi
    od
...
end proc
```

Código 39

Esto se repite hasta que los índices `i` y `j` se hayan cruzado. En ese momento se termina el ciclo y solo queda ubicar correctamente al elemento **pivot**, y asignar la variable `ppiv` para que indique la posición final en donde queda el mismo.

Finalización del procedimiento

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    ppiv:= lft
    i:= lft+1
    j:= rgt
    do i <= j --> if a[i] <= a[ppiv] --> i:= i+1
                     a[j] >= a[ppiv] --> j:= j-1
                     a[i] > a[ppiv] && a[j] < a[ppiv] --> swap(a,i,j)
                  fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```

Código 40

3.2.2 Análisis de complejidad

Para analizar la función, hay que tener en cuenta que para una lista de longitud n, si la partición siempre ocurre en el centro de la lista, habrá de nuevo $\log n$ divisiones. Con el fin de encontrar el punto de división, cada uno de los n ítems debe ser comparado contra el valor pivote. El resultado es $n \log n$. Además, no hay necesidad de memoria adicional como en el proceso de ordenamiento por mezcla. En el peor de los casos, los puntos de división pueden no estar en el centro y podrían estar muy sesgados a la izquierda o a la derecha, dejando una división muy desigual. En este caso, ordenar una lista de n ítems se divide en ordenar una lista de 0 ítems y una lista de $n-1$ ítems. Similarmente, ordenar una lista de tamaño $n-1$ se divide en una lista de tamaño 0 y una lista de tamaño $n-2$ y así sucesivamente. El resultado es un ordenamiento $O(n^2)$ con toda la sobrecarga que requiere la recursión.

3.2.3 Código

Ordenación rápida (Quick Sort)

```
proc quick_sort(in/out a: array[1..n] of T)
    quick_sort_rec(a,1,n)
end proc

proc quick_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var ppiv: nat
    if rgt > lft ->
        partition(a,lft,rgt,ppiv)
        quick_sort_rec(a,lft,ppv-1)
        quick_sort_rec(a,ppiv+1,rgt)
    fi
end proc

proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    ppiv:= lft
    i:= lft+1
    j:= rgt
    do i <= j -> if a[i] <= a[ppiv] -> i:= i+1
                    a[j] >= a[ppiv] -> j:= j-1
                    a[i] > a[ppiv] && a[j] < a[ppiv] -> swap(a,i,j)
                fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```

Código 41

CAPÍTULO

RECURRENCIAS DIVIDE Y VENCERÁS

4

Un algoritmo Divide y Vencerás típico resuelve un problema siguiendo estos 3 pasos:

1. **Dividir:** Descomponer el problema en sub-problemas del mismo tipo. Este paso involucra descomponer el problema original en pequeños sub-problemas. Cada sub-problema debe representar una parte del problema original. Por lo general, este paso emplea un enfoque recursivo para dividir el problema hasta que no es posible crear un sub-problema más.

2. **Vencer:** Resolver los sub-problemas recursivamente. Este paso recibe un gran conjunto de sub-problemas a ser resueltos. Generalmente a este nivel, los problemas se resuelven por sí solos.

3. **Combinar:** Combinar las respuestas apropiadamente. Cuando los sub-problemas son resueltos, esta fase los combina recursivamente hasta que estos formulan la solución al problema original. Este enfoque algorítmico trabaja recursivamente y los pasos de conquista y fusión trabajan tan a la par que parece un sólo paso.

4.1 ESTRUCTURA DE UN ALGORITMO DIVIDE Y VENCERÁS

La estrucutra de un algoritmo utilizado esta técnica es la siguiente:

Divide y Vencerás

```
fun DyV(x) ret y
    if x suficientemente pequeño o simple then y:= ad_hoc(x)
    else descomponer x en x1, x2, . . . , xa
        for i:= 1 to a do
            yi := DyV(xi)
        od
        combinar y1, y2, . . . , ya para obtener la solución y de x
    fi
end fun
```

Código 42

Donde normalmente x_i es una fracción de x ($|x_i| = \frac{|x|}{b}$), para algun b constante mayor a 1).
Normalmente se ve lo siguiente:

- **Solución trivial**: Si el problema es suficientemente pequeño, se resuelve de manera directa.
- **Descomposisión**: Se divide el problema en subproblemas más pequeños.
- **Combinación**: Se combinan las soluciones de los subproblemas para obtener la solución del problema original.

4.2 EJEMPLOS

1. Ordenación por intercalación
   - **Solución trivial**: Si el arreglo tiene un solo elemento, ya está ordenado.
   - **Descomposición**: Se divide el arreglo en dos mitades ($b = 2$).
   - **Combinación**: Se intnercalan los dos arreglos ordenados.
2. Ordenación rápida
   - **Solución trivial**: Si el arreglo tiene un solo elemento, ya está ordenado.
   - **Descomposición**: Se elige un pivote y se divide el arreglo en dos partes, una con los elementos menores al pivote y otra con los elementos mayores ($b = 2$).
   - **Combinación**: Se ordenan las dos partes recursivamente.

4.3 ANÁLISIS DE ALGORITMOS DIVIDE Y VENCERÁS

Si queremos contar el costo computacional (número de operaciones) $t(n)$ de la función DyV obtenemos:

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a \cdot t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

Donde:

- c es una constante que representa el costo computacional de la función `ad_hoc` en el caso base.
- $a \cdot t(n/b)$, el valor a que representa la cantidad de llamadas recursivas por nivel y n/b es el tamaño de la entrada en cada llamada recursiva es decir, el tamaño de los subproblemas.
- $g(n)$ es el costo computacional de los procesos de descomposición y de combinación.

Si $t(n)$ no es decreciente, y $g(n)$ es polinomial, entonces:

$$t(n) = \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

4.4 EJEMPLO COMPLETO DE ANÁLISIS

Se busca calcular el orden de complejidad de:

Ejemplo completo

```
proc f1(in n : nat)
    if n ≤ 1 then skip
    else
        for i := 1 to 8 do f1(n div 2) od
        for i := 1 to n 3 do t := 1 od
    fi
end proc
```

Código 43

En este caso notar que:

- Tamaño de la entrada: n,
- Operación a contar: $t := 1$.

Entonces, se puede definir una funcion $r(n)$, que representará la cantidad de asignaciones a la variable t que ocurren al llamar a la función $f1$ con el dato de entrada n.
Podemos observar que la función $f1$ esta dividida en dos casos, si $n \leq 1$ entonces no se realiza ninguna asignación a la variable t, por lo que $r(n) = 0$.

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \dots & \text{en caso contrario} \end{cases}$$

Como hay dos ciclos for en una secuencia, se puede analizar cada uno por separado y sumarlos, por ahora los puedo expresar como sumatoria:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} \dots + \sum_{i=1}^{n^3} \dots & \text{en caso contrario} \end{cases}$$

En el primer for, queremos contar la cantidad de asignaciones que se realizan al llamar a la función $f1$ con el dato de entrada $n/2$, por lo que se puede expresar como:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} r(n/2) + \sum_{i=1}^{n^3} \dots & \text{en caso contrario} \end{cases}$$

En el segundo for, se realiza una asignación a la variable t por cada iteración, por lo que se puede expresar como:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} r(n/2) + \sum_{i=1}^{n^3} 1 & \text{en caso contrario} \end{cases}$$

Resolviendo las sumatorias, se obtiene:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ 8 \cdot r(n/2) + n^3 & \text{en caso contrario} \end{cases}$$

Por lo que se puede observar que:

- $a = 8$,
- $b = 2$,
- $g(n) = n^3$.

Como $a = 8 = 2^3 = b^3$, se puede decir que el orden de complejidad es $O(n^3 \log n)$.

CAPÍTULO 5

TIPOS CONCRETOS

Se van a presentar los tipos enumerados, tuplas, arreglos y punteros.

5.1 TIPOS ENUMERADOS

Los tipos enumerados son aquellos que permiten definir un conjunto finito de valores, los cuales son llamados elementos del tipo. En Pascal, se definen de la siguiente manera:

Definición de un tipo enumerado

```
type E = enumerate
            elem1
            elem2
            ...
            elemk;
         end enumerate;
```

Código 44

En esta definición el tipo E tiene k elementos, los cuales son $elem1, elem2, ..., elemk$, si se quiere utilizar este tipo, en un programa se puede declarar una variable de tipo E y asignarle uno de los elementos del tipo.

Declaración de una variable de tipo enumerado

```
var v : E;
v := elem1;
```

Código 45

Por defecto en el lenguaje de la materia, vamos a poder utilizar los tipos enumerados con orden, es decir, podemos con un ciclo for recorrer todos los elementos del tipo enumerado:

Recorrido de un tipo enumerado

```
for v := elem1 to elemk do ... od
```

Código 46

5.2 TUPLAS

Las tuplas son un tipo de dato que permite agrupar varios valores en una sola variable. En el lenguaje de la materia, las tuplas se definen de la siguiente manera:

Definición de una tupla

```
type Tupla = tuple
                campo1 : tipo1;
                campo2 : tipo2;
                ...
                campok : tipok;
             end tuple;
```

Código 47

En esta definición, la tupla Tupla tiene k campos, los cuales son campo1, campo2, ..., campok, cada campo tiene un tipo asociado, el cual puede ser cualquier tipo de dato. Para acceder a los campos de una tupla se utiliza el operador punto.
Por ejemplo, se quiere definir un tipo para persona y luego acceder a los campos de la persona:

Definición de una tupla y acceso a los campos

```
type Persona = tuple
                name: string
                age: nat
                weight: real
             end tuple;
var p: Persona;
p.name := "Juan";
p.age := 20;
p.weight := 70.5;
```

Código 48

5.3 ARREGLOS

Los arreglos son un tipo de dato que permite almacenar varios valores del mismo tipo en una sola variable de tamaño fijo. En el lenguaje de la materia, los arreglos se definen de la siguiente manera:

Definición de un arreglo

```
var arr = array[M..N] of T;
```

Código 49

En esta definición, arr es un arreglo de tamaño $N-M$ de elementos de tipo T, los índices del arreglo van desde M hasta N. Para acceder a los elementos de un arreglo se utiliza el operador corchetes.
Tambien se puede definir un arreglo cuyos índices sean de tipo enumerado:

Definición de un arreglo con índices de tipo enumerado

```
type E = enumerate
            elem1
            elem2
            ...
            elemk;
         end enumerate;
var arr = array[elem1..elemk] of T;
```

Código 50

Si se busca crear arreglos de mas de una dimension, simplemente se separan por comas los tamaños de las dimensiones:

Definición de un arreglo de dos dimensiones

```
var arr = array[M1..N1, M2..N2] of T;
```

Código 51

Y el acceso a los elementos de un arreglo de dos dimensiones se hace de la siguiente manera:

Acceso a los elementos de un arreglo de dos dimensiones

```
arr[i, j] := 5;
```

Código 52

donde i es el índice de la primera dimensión y j es el índice de la segunda dimensión.

5.4 PUNTEROS

Dado un tipo T, un puntero a T es un tipo de datos que representa el lugar en la memoria en donde está alojado un elemento de tipo T. Por ejemplo, se puede definir un puntero a un natural de la siguiente manera:

Definición de un puntero

```
var p: pointer to nat;
```

Código 53

5.4.1 Operaciones con punteros

1. **Reservar:** Para reservar un nuevo bloque de memoria donde pueda almacenar un elemento del tipo al que apunta se utiliza la operación `alloc`.

2. **Acceder al valor:** Puedo acceder al valor en el bloque de memoria apuntado por p mediante la operación $\star$. Por ejemplo $\star p := 10$ decimos que el valor al que apunta p es 10.

3. **Liberar:** Para liberar el bloque de memoria reservado se utiliza la operación `free`. Por ejemplo `free(p)`, decimos que p ya no apunta a ningún bloque de memoria.

Para representar punteros que no apunten a nada se utiliza el valor `null`. Por ejemplo, si se quiere inicializar un puntero a nat que no apunte a nada se puede hacer de la siguiente manera:

Inicialización de un puntero a nat

```
var p: pointer to nat;
p := null;
```

Código 54

5.4.2 Ejemplo de uso de punteros

Se tiene el tipo persona y se quiere crear un puntero a persona:

Definición de un puntero a persona

```
type Persona = tuple
                 name: string
                 age: nat
                 weight: real
               end tuple;
var p: pointer to Persona;
```

Código 55

Ahora se analizan los casos posibles para p:

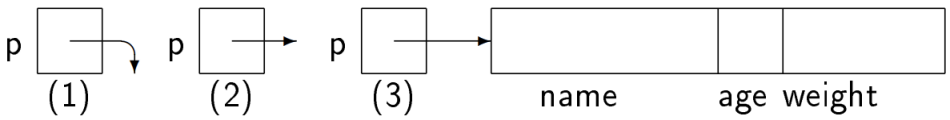

Figura 5.1: Casos posibles para un puntero

1. El valor de p es `null`, lo cual indica que no apunta a ningún bloque de memoria.
2. La dirección de memoria a la que apunta p no está reservada, por lo que no se puede acceder a ella.
3. El valor de p es una dirección de memoria reservada, por lo que se puede acceder a ella. En este caso, $\star p$ denota a la persona a la que apunta p y por lo tanto se podria acceder a los campos de la persona ($\star$p.age, $\star$p.name, $\star$p.weight). Y tambien se podrian modificar los campos de la persona ($\star$p.age := 20).

Una notación alternativa para acceder a los campos de una persona a la que apunta p es $\rightarrow$, así por ejemplo en vez de $\star$p.age se puede escribir $p \rightarrow age$, tanto para leerlo como para modificarlo ($p \rightarrow age := 20$).

CAPÍTULO

TIPOS ABSTRACTOS DE DATOS (TADS O ADTS EN INGLÉS) 6

Un tipo abstracto de datos (TAD) es una colección de valores y operaciones que se definen mediante una especificación que es independiente de cualquier representación.
Un TAD es una abstracción:

- Se destacan los detalles de la especificación (el qué),
- Se ocultan los detalles de la implementación (el cómo).

6.1 ESPECIFICACIÓN DE UN TAD

Para especificar un TAD se deben definir:

1. Su **nombre**.
2. Especificar **constructores**: procedimientos o funciones mediante los cuales puedo crear elementos del tipo que estoy especificando.
3. Especificar **operaciones**: todos los procedimientos o funciones que permitirán manipular los elementos del tipo de datos que estoy especificando.
4. Indicamos los **tipos** de cada constructor y operación (el encabezado de los procedimientos o funciones), y mediante lenguaje natural explicamos qué hacen.
5. Algunas operaciones pueden tener restricciones que las indicamos mediante **precondiciones**.
6. Debemos especificar también una operación de destrucción que libera la memoria utilizada por los elementos del tipo, en caso que sea necesario.

6.1.1 Ejemplo

Suponga que se va a desarrollar un programa para administrar la información de una biblioteca.
Puesto que allí hay elementos como ficheros, usuarios, libros, etc., que participan en el problema, en el software existirá un TAD que represente y simule la operación de cada uno de ellos: el TAD Fichero, el TAD Usuario y el TAD Libro. Estos TAD estarán relacionados dentro del programa, de la misma manera como los elementos que modelan están relacionados en la biblioteca: un elemento del TAD Usuario puede tener en préstamo un elemento del TAD Libro, los elementos del TAD Fichero tienen elementos del TAD Ficha, que representan libros de la biblioteca, etc. No existirá un TAD Pared, puesto que no participa en el problema, así haga parte de la biblioteca (a menos, claro está, que se trate de un sistema de diseño arquitectónico, en el cual las paredes sean los elementos de base.

6.2 IMPLEMENTACIÓN DE UN TAD

- Definir un nuevo tipo con el nombre del TAD especificado. Para ello utilizamos tipos concretos y otros tipos definidos previamente.

- Implementar cada constructor respetando los tipos tal como fueron especificados.
- Implementar cada operación respetando los tipos tal como fueron especificados.
- Implementar operación de destrucción liberando memoria si es que se ha reservado al construir los elementos.
- Pueden surgir nuevas restricciones que dependen de cómo implementamos el tipo.
- Puedo necesitar operaciones auxiliares que no están especificadas en el tipo.

6.3 TAD LISTA

Las listas son colecciones 0 o mas elementos de un mismo tipo, de tamaño variable.

6.3.1 Especificación

- **Nombre**: Lista
- **Constructores**:
  - `empty()`: crea una lista vacía.

    empty()

    ```
    fun empty() ret l : List of T
    {— crea una lista vacia. —}
    ```

    Constructores

  - `addl()`: agrega un elemento al comienzo de la lista.

    addl()

    ```
    proc addl (in e : T, in/out l : List of T)
    {— agrega el elemento e al comienzo de la lista l. —}
    ```

    Constructores

- **Operaciones**:
  - decidir si una lista es vacía,
  - tomar el primer elemento,
  - tirar el primer elemento,
  - agregar un elemento al final,
  - obtener la cantidad de elementos,
  - concatenar dos listas,
  - obtener el elemento en una posición específica,
  - tomar una cantidad arbitraria de elementos,
  - tirar una cantidad arbitraria de elementos,
  - copiar una lista en una nueva.

```
spec List of T where

constructors
    fun empty() ret l : List of T
    {— crea una lista vacia. —}

    proc addl (in e : T, in/out l : List of T)
    {— agrega el elemento e al comienzo de la lista l. —}

destroy
    proc destroy (in/out l : List of T)
    {— Libera memoria en caso que sea necesario. —}

operations
    fun is_empty(l : List of T) ret b : bool
    {— Devuelve True si l es vacia. —}

    fun head(l : List of T) ret e : T
    {— Devuelve el primer elemento de la lista l —}

    {— PRE: not is_empty(l) —}
    proc tail(in/out l : List of T)
    {— Elimina el primer elemento de la lista l —}

    {— PRE: not is_empty(l) —}
    proc addr (in/out l : List of T,in e : T)
    {— agrega el elemento e al final de la lista l. —}

    fun length(l : List of T) ret n : nat
    {— Devuelve la cantidad de elementos de la lista l —}

    proc concat(in/out l : List of T,in l0 : List of T)
    {— Agrega al final de l todos los elementos de l0
    en el mismo orden.—}

    fun index(l : List of T,n : nat) ret e : T
    {— Devuelve el n—esimo elemento de la lista l —}

    {— PRE: length(l) > n —}
    proc take(in/out l : List of T,in n : nat)
    {— Deja en l so lo los primeros n
    elementos, eliminando el resto —}

    proc drop(in/out l : List of T,in n : nat)
    {— Elimina los primeros n elementos de l —}

    fun copy_list(l1 : List of T) ret l2 : List of T
    {— Copia todos los elementos de l1 en la nueva lista l2 —}
```

TAD Lista

6.3.2 Ejemplo de uso

Uso de operaciones de listas

```
fun promedio (l : List of float) ret r : float
    var largo : nat
    var elem : float
    var laux : List of float

    laux := copy(l) {– copio la lista para no modificar la original –}
    r := 0.0 {– inicializo el promedio en 0 –}
    largo := length(laux) {– obtengo la cantidad de elementos –}
    do (not is_empty(laux)) –> {– mientras la lista no sea vacia –}
        elem := head(laux) {– tomo el primer elemento –}
        r := r + elem {– sumo el elemento al promedio –}
        tail(laux) {– elimino el primer elemento –}
    od
    destroy(laux) {– libero la memoria –}
    r := r / largo {– calculo el promedio –}
end proc
```

Promedio

6.4 IMPLEMENTACIÓN DE UN TAD LISTA MEDIANTE PUNTEROS

- Implementaremos el TAD lista utilizando punteros, implementación conocida como `lista enlazada`.
- Cada elemento de la lista estará alojado en un nodo conteniendo además un puntero hacia el siguiente.

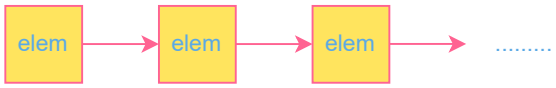

Figura 6.1: Lista enlazada

- Una lista será un puntero a un nodo.
- La lista vacía se implementa con el puntero null.
- Esta implementación permite tener la lista de elementos almacenada en lugares de la memoria no necesariamente contiguos.
- No existe límite teórico para almacenar elementos. En la práctica dicho límite será la cantidad de memoria.

En resumen, la implementación de un TAD lista mediante punteros consiste en:

- Definir un tipo nodo que contenga un elemento y un puntero al siguiente nodo.
- Definir un tipo lista que sea un puntero a un nodo.
- Implementar cada constructor y operación respetando los tipos especificados.
- Implementar la operación de destrucción liberando la memoria utilizada.

La implementación completa del TAD lista mediante punteros se encuentra en el solucionario.

6.5 TAD CONTADOR

Un problema interesante (y no del todo trivial) es el de controlar que una cierta expresión tiene balanceados sus paréntesis. Se quiere dar un algoritmo que tome una expresión (dada, por ejemplo, por un arreglo de caracteres) y devuelva verdadero si la expresión tiene sus paréntesis correctamente balanceados, y falso en caso contrario. Este problema puede solucionarse con un algoritmo que recorre la expresión de izquierda a derecha y que utiliza un entero, que se inicializa en 0 y se incrementa cada vez que se encuentra un paréntesis que abre y se decrementa (chequeando previamente que dicho número no sea nulo) cada vez que se encuentra un paréntesis que cierra. Al analizar, sólo resta comprobar que dicho entero sea cero. Esta descripción debería alcanzar para darse uno cuenta de que no es en realidad un entero lo que hace falta, sino mucho menos. Un entero es un objeto que admite todas las operaciones aritméticas, acá sólo se necesita inicializar en 0, incrementar, decrementar y controlar si su valor es o no cero.

6.5.1 Especificación

Los **constructores** (en este caso inicial e incrementar) deben ser capaces de generar todos los valores posibles del TAD. En lo posible cada valor debe poder generarse de manera única. Intuitivamente, esto se cumple para inicial e incrementar: partiendo del valor inicial y tras sucesivos incrementos se puede alcanzar cualquier valor posible; y hay una única forma de alcanzar cada valor posible de esa manera.
Las demás **operaciones** quedan declaradas simplemente como tales. Las operaciones se definen con ecuaciones que deben cubrir todos los casos posibles, salvo los declarados que no pueden ocurrir, como en el caso de decrementar que no puede aplicarse a un contador que sea inicial (a pesar de que se podría defnir de manera obvia para ese caso).

- comprobar si su valor es el inicial
- decrementar si no lo es

```
spec Counter where

constructors
    fun init() ret c : Counter
    {— crea un contador con valor inicial —}

    proc incr(in/out c : Counter)
    {— incrementa el valor del contador c —}

destroy
    proc destroy(in/out c : Counter)
    {— Libera memoria en caso que sea necesario. —}

operations
    fun is_init(c : Counter) ret b : bool
    {— Devuelve True si el contador c tiene el valor inicial —}

    {— PRE: not is_init(c) —}
    proc decr(in/out c : Counter)
    {— decrementa el valor del contador c —}
```

TAD Contador

6.5.2 Implementación

```
implement Counter where
type Counter = nat

proc init (out c: Counter)
    c:= 0
end proc

proc inc (in/out c: Counter)
    c:= c+1
end proc

fun is_init (c: Counter) ret b: bool
    b:= (c = 0)
end fun

{- PRE: not is_init(c) -}
proc dec (in/out c: Counter)
    c:= c-1
end proc

proc destroy (in/out c: Counter)
    skip
end proc
```

Implementación del TAD Contador

6.5.3 Algoritmo de balanceo de paréntesis

```
fun matching_parenthesis (a: array[1..n] of char) ret b: bool
    var i: nat
    var c: Counter
    b:= true
    init(c)
    i:= 1
    do i ≤ n ∧ b → if a[i] = '(' → inc(c)
                                   a[i] = ')' ∧ is_init(c) → b:= false
                                   a[i] = ')' ∧ ¬is_init(c) → dec(c)
                                   otherwise → skip
                    fi
                    i:= i+1
    od
    b:= b ∧ is_init(c)
    destroy(c)
end fun
```

Algoritmo de balanceo de paréntesis

6.6 TAD PILA