



1. Introducción

1.1 Qué hacer ante un problema real

Cuando tenemos un problema real, lo primero que tenemos que preguntarnos es si se puede resolver con la ayuda de un ordenador, en caso afirmativo, debemos modelizar el problema. Para ello, diseñamos un algoritmo siguiendo alguna técnica de diseño, que se verá en cursos dedicados a Algorítmica. Unos de los factores que tendremos en cuenta al escoger una técnica de diseño será la eficiencia que se obtenga en el algoritmo siguiendo esa técnica de diseño. Nuestro objetivo es obtener el algoritmo más eficiente. Una vez planteado nuestro algoritmo veremos que datos maneja nuestro problema, y por lo tanto que tipos de datos abstractos tendremos que utilizar para contener estos datos junto con las operaciones que actuarán sobre estos datos. Al valorar las operaciones necesarias, también haremos un estudio de las operaciones más frecuentes que usarán el algoritmos sobre esos datos. Este aspecto será clave también para establecer la eficiencia, de forma que elegir una estructura de datos u otra hará que estas operaciones más frecuentes se hagan de forma más rápida en tiempo o con menos gasto de memoria.

Por lo tanto una fase fundamental será definir qué tipos de datos abstractos se usarán: dando su dominio (datos que representan), especificación e implementación. También usaremos estructuras de control (p.e sentencias condicionales, sentencias repetitivas,etc) que implementen los pasos fundamentales de mi algoritmo. Todo esto dará lugar a un programa que una vez hecho debemos ver si funciona correctamente y si da una buena solución al problema que teníamos.

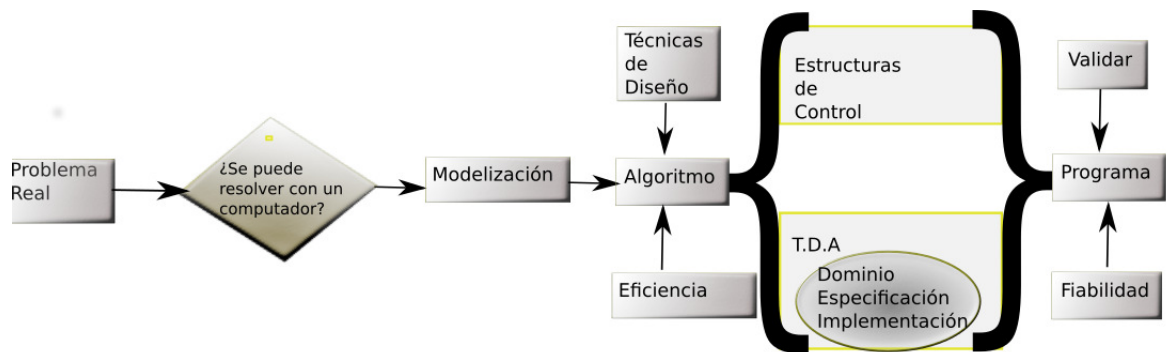



Figura 1.1: Esquema: Pasos a seguir para resolver un problema en un ordenador

1.2 Elección de la estructura de Datos

Para elegir la estructura de datos, vamos a basarnos en las operaciones más frecuentes que vamos a realizar. Por ejemplo, si tenemos un conjunto de enteros y la operación más frecuente es la inserción, utilizaremos un conjunto de celdas enlazadas, pero, si la operación más frecuente es la consulta, utilizaremos un vector dinámico. En este punto analizaremos las operaciones más frecuentes que se pueden llevar a cabo con datos almacenados en un vector y/o conjunto de celdas enlazadas.

1.2.1 Algoritmos de búsqueda o consulta

Búsqueda binaria

 Si tus datos están ordenados y se acceden a ellos de forma directa, la búsqueda binaria es lo óptimo

En el caso de que los datos que se manejan estén ordenados y se acceden a ellos de forma directa (p.e datos almacenados en un vector), la búsqueda binaria es la búsqueda mas eficiente. El elemento que estamos buscando se compara con el elemento que ocupa la mitad del vector, si coinciden se termina la búsqueda, si no, se determina la mitad del vector en la que puede estar el elemento y se repite el proceso.

Su código sería:

```

1      class MiVector{
2      private:
3          //puntero con direccion a la zona de memoria con los datos
4          char *datos;
5          //numero total de elementos almacenados
6          int total_utilizados;
7          ...
8      public:
9          ..
10         //El metodo devuelve la posicion del elemento buscado
11         int BusquedaBinaria (char buscado)
12         {
13             int izda, dcha, centro;
14             bool encontrado = false;
15
16             izda = 0;
17             //indice del elemento valido mas a la derecha
18             dcha = total_utilizados - 1;
19             //indice del elemento en la mitad del vector
20             centro = (izda + dcha) / 2;
21
22             while (izda <= dcha && !encontrado)
23             {
24                 if (datos[centro] == buscado) //lo hemos encontrado?
25                     encontrado = true;
26
27                 else
28                     //esta a la izquierda
29                     if (buscado < datos[centro])
30                         dcha = centro - 1;
31
32                 else //debe estar a la derecha
33                     izda = centro + 1;
34
35                 centro = (izda + dcha) / 2;
36             }
37
38             if (encontrado)
39                 return centro;
40
41             else
42                 return -1; //No esta en el vector
43         }

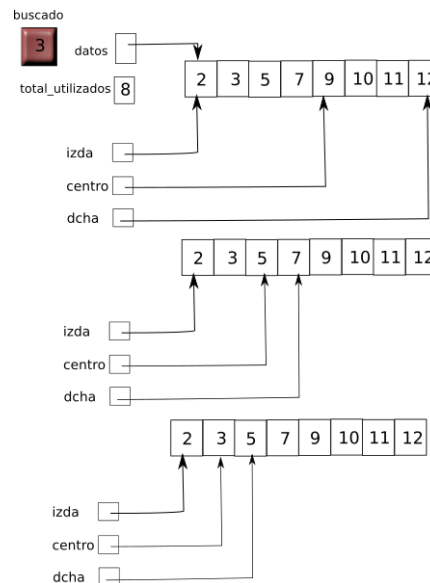
```



¿Cuál
es el máximo
número de
iteraciones para
buscar un
elemento
usando la
búsqueda
binaria?

En la figura 1.2 se puede ver un ejemplo de ejecución de la búsqueda secuencial para un vector dado cuando el objetivo es buscar el elemento 3.

Figura 1.2: **Ejemplo:** Pasos ejecutados por la búsqueda binaria sobre un vector dado y el elemento a buscar es 3



Matemáticas de fondo 1.2.1 La función logaritmo será usada en muchos de nuestros algoritmos para obtener su tiempo de ejecución. Por lo tanto es importante saber que representa la función logaritmo. Las siguientes expresiones matemáticas $a = \log_b c$ o $c = b^a$ son expresiones equivalentes. En el primer caso podemos decir de forma coloquial que a es el logaritmo en base b de c y en el segundo caso que b multiplicado por si mismo a veces es igual a c , $c = \underbrace{b \dots b}_a$. Así por ejemplo cuando

la base del logaritmo es $b = 2$ y $a = 3$ entonces $c = 8$. Situaciones en las que nos hagan falta usar la función logaritmo puede ser la siguiente: dado un numero n , que puede ser por ejemplo el numero de elementos de un vector, nos preguntemos cuantas veces puedo dividir n por 2 es decir cuantas veces puedo dividir mi vector por dos. Cuando me hago esta pregunta estoy resolviendo la ecuación $x = \log_2(n)$. O la cuestión alternativa es cuantas veces tengo que multiplicar 2 por si mismo para obtener n es decir $n = 2^x$. Así en la siguiente figura se muestra para un vector cuantas veces podría dividirlo por 2.

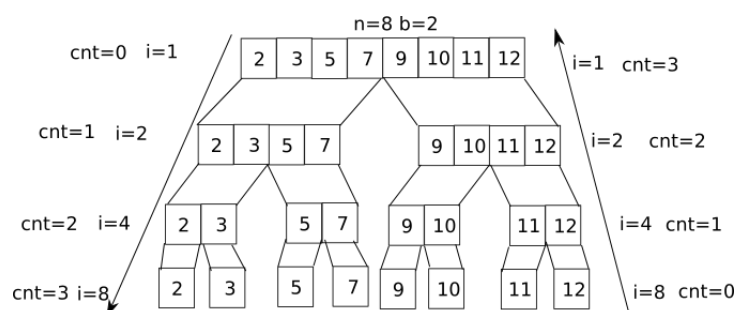
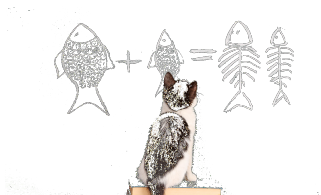
A continuación se muestran dos formas de implementar el logaritmo en base b de un número n

```

1  int Logaritmo (int n,int b)
2  {
3      int i=1;
4      int cnt=0;
5      while (i<n){
6          i*=b;
7          cnt++;
8      }
9      return cnt;
10 }
```

```

1  int Logaritmo (int n,int b)
2  {
3      int i=n;
4      int cnt=0;
5      while (i>1){
6          i/=b;
7          cnt++;
8      }
9      return cnt;
10 }
```



Búsqueda secuencial

En el caso de que tengamos una secuencia de elementos no ordenados, por ejemplo almacenados en un vector, aplicar una búsqueda sobre estos datos de forma directa es aplicar una búsqueda secuencial. Para ellos vamos recorriendo el vector hasta encontrar el elemento que buscamos o hasta llegar al final, en cuyo caso devolvemos un -1.

Su código sería:

```

1  class MiVector{
2      private:
3          //puntero con direccion a la zona de memoria con los datos
4          char *datos;
5          //numero total de elementos almacenados
6          int total_utilizados;
7      ...
8      public:
9      int BusquedaSecuencial (char buscado)
10     {
11         bool encontrado = false;
12         int pos_encontrado;
13
14         for (int i=0; i<total_utilizados && !encontrado; i++)
15         {
16             if (buscado == datos[i])
17             {
18                 encontrado = true;
19                 pos_encontrado = i;
20             }
21         }
22
23         if (encontrado)
24             return pos_encontrado;
25
26         else
27             return -1;
28     }

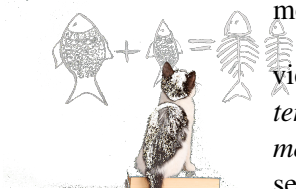
```

¿Cuántas iteraciones hacemos cuando el elemento buscado no se encuentra?

¿Cuántas iteraciones hacemos cuando el elemento buscado está en la primera posición?

Matemáticas de fondo 1.2.2 La siguiente ecuación $\sum_{i=1}^n 1$ representa que estamos realizando la siguiente suma $\underbrace{1 + \dots + 1}_n$, es decir sumamos n veces 1. A veces

viendo $\sum_{i=1}^n 1$ no sabemos exactamente cuantos unos estamos sumando. **Regla:** si tenemos $\sum_{i=1}^n 1$ el numero de veces que sumamos 1 es el valor ultimo de i que es n menos el primer valor que toma i que es 1 y siempre mas 1, así en nuestro ejemplo sería $n - 1 + 1 = n$. Un ejemplo genérico es si tenemos $\sum_{i=j}^n 1 = 1 * (n - j + 1)$ siendo el valor inicial que toma i j .



1.2.2 Algoritmo de inserción

Si tenemos una secuencia de elementos ordenada y queremos insertar sobre esta secuencia un nuevo elemento, dejando la secuencia tras la inserción también ordenada, nos podríamos plantear dos situaciones: contener la secuencia en una lista con celdas enlazadas o en un vector dinámico.

Celdas enlazadas

Para insertar un dato en una lista (celdas enlazadas), creamos una celda para el valor que vamos a insertar, recorremos la lista hasta encontrar la posición donde vamos a insertar nuestro dato (la posición tal que la celda anterior sea menor y la siguiente mayor a nuestro dato) y lo insertamos enlazando la celda anterior con la que hemos creado y la que hemos creado con la siguiente. Su código sería:

```

1      class Lista{
2      private:
3          Celda *primera;
4          ....
5          //inserta de forma ordenada el elemento
6          void Insertar ( double insertado)
7          {
8              //Creamos una celda para el valor que vamos a insertar
9              Celda *celda_nueva = new Celda;
10             celda_nueva->dato=insertado;
11             //Nuestra lista esta vacia
12             if (primera==0){
13                 primera=celda_nueva;
14                 primera->sig=0;
15             }
16             else {
17
18                 //Si el valor es menor que la de la  la primera celda,enlazamos nuestra
19                 //celda con la que empieza la lista
20                 if (primera->dato > insertado){
21                     celda_nueva->sig = primera;
22                     primera = celda_nueva;
23                 }
24
25                 else
26                 { //Recorremos la lista para buscar la posicion donde insertar nuestro valor:
27                     Celda *p=primera;
28                     while (p->sig!=0 && p->sig->dato<insertado)
29                     {
30                         p=p->sig;
31                     }
32                     //una vez encontrada la posicion donde queremos insertar
33                     celda_nueva->sig=p->sig;
34                     p->sig=celda_nueva;
35
36                 }
37             }
38         }

```


Vector Dinámico

Supongamos el mismo problema de inserción usando un vector dinámico, el algoritmo consistiría en recorrer el vector hasta encontrar la casilla en la que vamos a insertar nuestro dato (tiene que cumplir que el dato anterior sea menor y el siguiente mayor al que vamos a insertar), después reservamos memoria para un vector con una casilla más y copiamos en él todos los elementos hasta la casilla anterior a la insertada, después copiamos el valor a insertar y por último el resto del vector. Eliminamos por último el vector antiguo.

Su código sería:

```

1      class VectorDinamico{
2      private:
3          int *v;
4          int total_utilizados;
5          ....
6
7      void Insertar (int insertado)
8      {
9          int posicion_insertado, i=0;
10
11         while (v[i]<=insertado && i<total_utilizados)
12             i++;
13
14         posicion_insertado = i;
15
16         int * nuevo_v = new VectorDinamico [total_utilizados+1];
17
18         //Ahora insertamos todos los valores hasta el anterior a insertado:
19         for (i=0; i<posicion_insertado; i++)
20             nuevo_v[i] = v[i];
21
22         //Despues insertamos nuestro valor:
23         nuevo_v[posicion_insertado] = insertado;
24
25         //Y luego el resto de valores:
26         //Recorremos hasta total_utilizados+1 porque el
27         //vector ahora tiene un valor mas
28         for (i=posicion_insertado+1; i<total_utilizados+1; i++)
29             nuevo_v[i] = v[i-1];
30
31         //Borramos la memoria :
32
33         delete [] v;
34         v = nuevo_v;//asignamos nueva memoria
35         total_utilizados++;//incrementamos el numero de elementos
36     }

```

Ejercicio 1.2.0

Analizado el algoritmo de *Insertar* usando un conjunto de celdas enlazadas y un vector dinámico:

- ¿Cual de los dos algoritmos es mas eficiente?
- Suponed que en ambos algoritmos *Insertar* nos da la posición exacta donde insertar el nuevo dato. ¿Qué escogerías un vector dinámico o un conjunto de celdas enlazadas si esta es la

operación más frecuente?.

□

1.2.3 Algoritmo de borrado

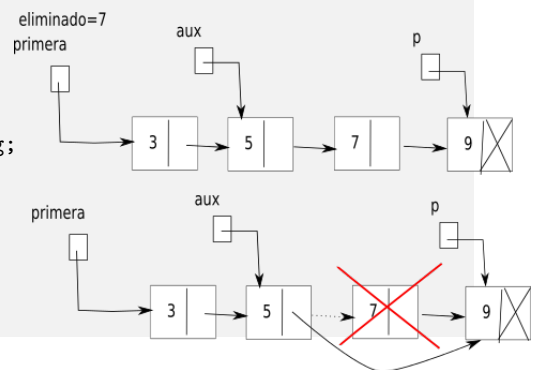
Celdas enlazadas

Para eliminar un dato en un conjunto de celdas enlazadas, debemos recorrer la lista hasta encontrar el dato que queremos eliminar. Entonces, enlazamos la celda anterior a la que contiene ese dato, con la siguiente y después liberamos la memoria que ocupa esa celda. Si el dato a eliminar está en la primera celda, hacemos que la lista empiece en la segunda celda. Su código sería:

```

1  struct Celda{
2      int dato;
3      Celda * sig;
4  }
5  class Lista{
6      private:
7          Celda *primera;
8          ....
9      void Borrar ( double eliminado)
10     {
11
12         //si el valor coincide con el primer elemento de la lista,
13
14         if (eliminado == primera->dato)
15         {
16             Celda *aux=primera;
17             primera=primera->sig;
18             delete aux;
19         }
20         else
21         {
22             //Buscamos en la lista hasta encontrar el valor
23             //que buscamos para eliminar
24             Celda *aux = primera;
25             while (aux->sig!=0 && eliminado != aux->sig->dato)
26             {
27                 aux=aux->sig;
28             }
29             if (aux->sig!=0)
30             {
31
32                 Celda* p=aux->sig->sig;
33                 delete aux->sig;
34                 aux->sig=p;
35             }
36         }
37     }
38 }

```



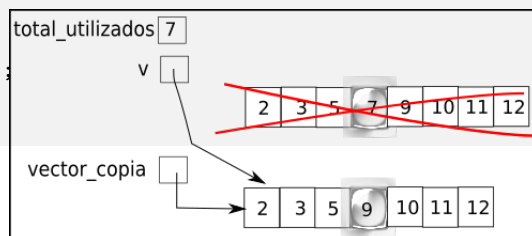
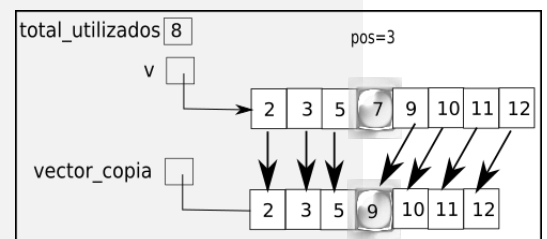
Vector Dinámico

Para un vector, tendríamos que desplazar una posición todos los elementos siguientes al elemento que queremos eliminar para así borrar el elemento. Su código sería

```

1  typedef int TipoBase;
2  class VectorDinamico{
3  private:
4      int *v;
5      int total_utilizados;
6      ....
7
8  void Borrar(int pos)
9  {
10     TipoBase* vector_copia;
11
12     int i;
13
14     vector_copia = new TipoBase[total_utilizados - 1];
15     for (i=0;i<pos;i++)
16         vector_copia[i]=v[i];
17
18     for(i = pos; i < total_utilizados-1; i++)
19     {
20         vector_copia[i] = v[i+1];
21     }
22
23     delete [] v;
24
25     v = vector_copia;
26
27     total_utilizados--;
28
29 }

```



1.2.4 Algoritmos de ordenación

Inserción

El vector se divide en dos subvectores: el de la izquierda ordenado y el de la derecha desordenado. Cogemos el primer elemento del subvector desordenado y lo insertamos de forma ordenada en el subvector de la izquierda. Para ello, vamos fijando el inicio del vector derecho con un índice izda, seleccionamos el valor de $v[\text{izda}]$ y lo insertamos en el vector izquierdo. Su código sería:

¿Cómo
debe estar el
vector
inicialmente
para realizar el
mayor número
de operaciones?

```

1      typedef int TipoBase;
2      class VectorDinamico{
3      private:
4          int *v;
5          int total_utilizados;
6          ....
7          void Ordena_Insercion ()
8          {
9              int izda, i;
10             TipoBase a_desplazar;
11
12             for (izda = 1; izda < total_utilizados; izda++)
13             {
14                 a_desplazar = v[izda];
15
16                 for (i = izda; i>0 && a_desplazar < v[i-1]; i--)
17                     v[i] = v[i-1];
18
19                 v[i] = a_desplazar;
20             }
21     }
```

Burbuja

A la izquierda se va dejando un subvector ordenado. Desde el final y hacia atrás, se van comparando los elementos dos a dos y se deja a la izquierda el más pequeño (intercambiándolos). Para ello, vamos fijando el inicio del subvector derecho con un contador, recorremos el subvector de la derecha desde el final hasta el principio con un contador *i* y si *v[i]* menor que *v[i-1]* se intercambian. Su código sería:

```

1      typedef int TipoBase;
2      class VectorDinamico{
3      private:
4          int *v;
5          int total_utilizados;
6          ....
7          void Ordena_Burbuja ()
8          {
9              int izda, i;
10             //Esta variable comprueba si se ha hecho un cambio en un bucle,
11             //en caso negativo ya tendríamos el vector ordenado.
12             bool cambio;
13
14             for (izda = 0; izda < total_utilizados && cambio; izda++)
15             {
16                 cambio = false;
17                 for (i = total_utilizados-1; i>izda; i--)
18                     if (v[i] < v[i-1])
19                     {
20                         Intercambia(i, i-1);
21                         cambio = true;
22                     }
23             }
24     }
```

Selección

Nuestro vector está dividido en dos subvectores, uno que está ordenado y otro que no. Para ello, vamos comparando los valores que hay al principio del vector con los del resto y los menores, los movemos al principio. Acabamos cuando el subvector de elementos ordenados ocupe ya todo el vector. Su código sería:

```
1  typedef int TipoBase;
2  class VectorDinamico{
3  private:
4      int *v;
5      int total_utilizados;
6      ....
7
8  void Ordena_Seleccion ()
9  {
10     int pos_min;
11
12     //Donde total_utilizados es el numero de elementos del vector
13     for (int izda=0; izda<total_utilizados; izda++)
14     {
15         pos_min = PosicionMinimoEntre (izda, total_utilizados-1);
16         Intercambia (izda, pos_min);
17     }
18 }
19
20 void Intercambia (int pos_izda, int pos_dcha)
21 {
22     char intercambia;
23
24     intercambia = v[pos_izda];
25     v[pos_izda] = v[pos_dcha];
26     v[pos_dcha] = intercambia;
27 }
28
29 int PosicionMinimoEntre (int izda, int dcha)
30 {
31     int posicion_minimo;
32     char minimo;
33
34     minimo = v[izda];
35     posicion_minimo = izda;
36
37     for (int i=izda+1; i<=dcha; i++)
38         if (v[i] < minimo)
39         {
40             minimo = v[i];
41             posicion_minimo = i;
42         }
43
44     return posicion_minimo;
45 }
```

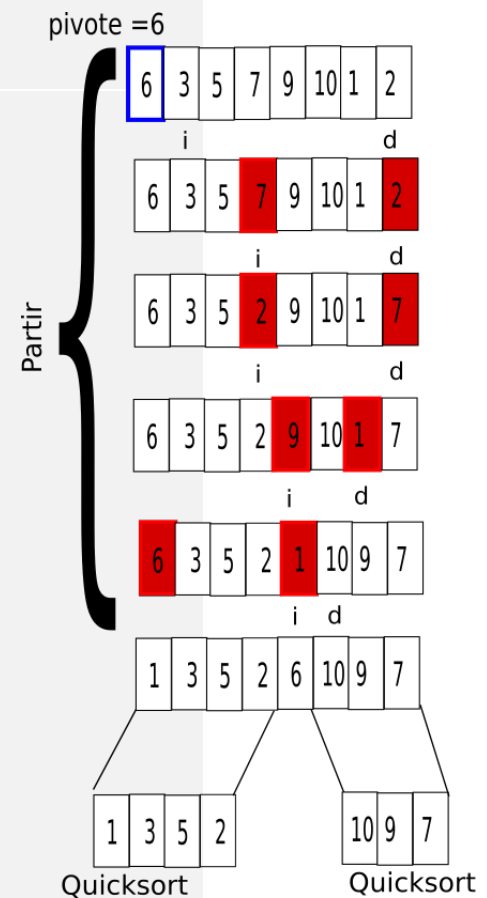
Quicksort

Este método en cada paso ordena los elementos relativos a un pivote (uno de los elemento del vector).

```

1 void QuickSort (int inicio, int final)
2 {
3     int pos_pivote;
4
5     if (inicio < final)
6     {
7         pos_pivote = Partir (inicio, final);
8         //Ordena primera mitad
9         QuickSort (inicio, pos_pivote-1);
10        //Ordena segunda mitad
11        QuickSort (pos_pivote + 1, final);
12    }
13 }
14
15 int Partir (int primero, int ultimo)
16 {
17     int intercambia, izda, cha;
18     int pivote = v[primero];
19
20     i = primero + 1; //avanza hacia delante
21     d = ultimo; //retrocede hacia atras
22
23     while (i <= d)
24     {
25         while (i <= d && v[i] <= pivote)
26             i++;
27
28         while (i <= d && v[d] > pivote)
29             d--;
30
31         if (i < d)
32         {
33             intercambia = v[i];
34             v[i] = v[d];
35             v[d] = intercambia;
36             d--;
37             i++;
38         }
39     }
40
41     intercambia = v[primero];
42     v[primero] = v[d];
43     v[d] = intercambia;
44
45     return d;
46 }

```



La ordenación relativa al pivote la realiza la función Partir. Esta ordenación consiste en dejar todos los elementos menores o iguales que el pivote a la izquierda y todos los elementos mayores a la derecha. La función de ordenación QuickSort se llama recursivamente para el subvector izquierda y subvector

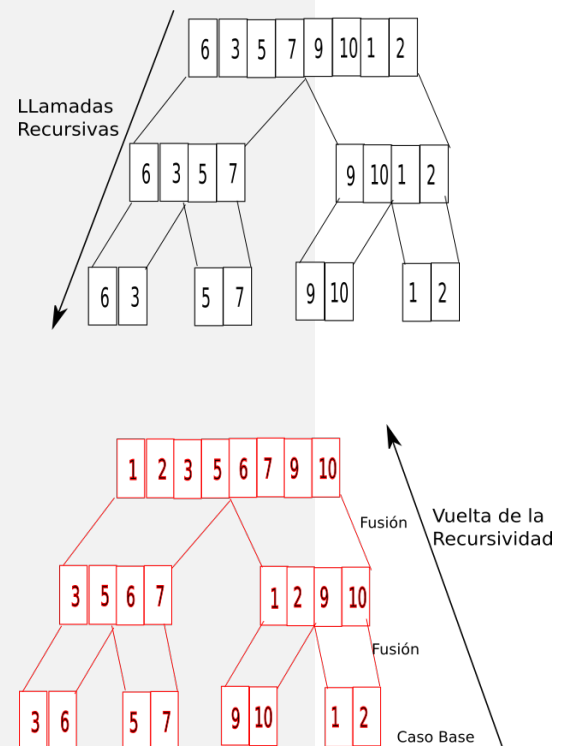
derecha. La función Partir toma un elemento arbitrario del vector (pivote), en nuestro código se toma el elemento primero. A continuación la función Partir recorre el vector de izquierda a derecha usando el índice i hasta encontrar un elemento apuntado por i tal que $v[i]$ sea mayor que pivote. Después, recorre el vector de derecha a izquierda (con el índice d) hasta encontrar otro elemento tal que $v[d]$ menor que pivote e intercambia los elementos $v[i]$ y $v[d]$. Se repite el proceso hasta que se cruzan i y d . En este caso se intercambian $v[\text{primero}]$ por $v[d]$ y se devuelve d (en $v[d]$ tenemos el pivote) para establecer las dos mitades siguientes. El elemento $v[d]$ ya está colocado y no se volverá a tratar más.

Mergesort

Si tenemos un vector de x elementos y queremos ordenarlos, dividimos el vector por la mitad hasta llegar a un número mínimo de elementos en el que el vector se ordena y luego se fusiona con el nivel superior. Es una función recursiva. Su código sería:

```

1  void Orden_MergeSort (int * v, int n)
2  {
3      if (n==1)
4          return v[0];
5      else
6          if (n == 2)
7          {
8              if (v[0] > v[1])
9                  Intercambiar (v[0],v[1]);
10         }
11     else if (n>2)
12     {
13         int ni=(n/2), nd=n-(n/2);
14         int * vi=new int [ni];
15         int * vd=new int [nd];
16
17         for(int i=0;i<ni;i++)
18             vi[i]=v[i];
19         for(int i=0;i<nd;i++)
20             vd[i]=v[i+(n/2)];
21
22         //Ordenamos a la izquierda
23         Orden_MergeSort (vi,ni);
24
25         //Ordenamos a la derecha
26         Orden_MergeSort (vd,nd);
27
28         //Fusionamos en v, vi y vd
29         Fusion(v,vi,vd,ni,nd);
30         delete [] vi;
31         delete [] vd;
32     }
33 }
```



```
1 void Fusión (int * vout, int * vi, int * vd, int ni, int nd)
2 {
3     int pi=0, pd=0, p=0;
4
5     while (pi < ni && pd < nd)
6     {
7         if (vi[pi] < vd[pd])
8         {
9             vout[p] = vi[pi];
10            pi++;
11        }
12
13        else
14        {
15            vout[p] = vd[pd];
16            pd++;
17        }
18        p++;
19    }
20    //si queda algo en vi
21    while (pi < ni)
22    {
23        vout[p] = vi[pi];
24        p++;
25        pi++;
26    }
27
28    //si queda algo en vd
29    while (pd < nd)
30    {
31        vout[p] = vd[pd];
32        p++;
33        pd++;
34    }
35 }
```

1.2.5 Estructuras de Datos

Hasta aquí hemos repasado dos estructuras de datos lineales: vector y lista con celdas enlazadas y hemos repasado las operaciones más importantes: insertar, borrar, consultar y ordenar. No obstante estas estructuras de datos, vectores y celdas enlazadas, se muestran poco eficientes dependiendo de las condiciones de nuestros datos y operaciones más frecuentes que hagamos. Es por ello que en este curso analizaremos otras estructuras. En la figura 1.3 se puede ver la estructura de datos en los que derivamos dependiendo de las condiciones que se le imponen a nuestros datos.

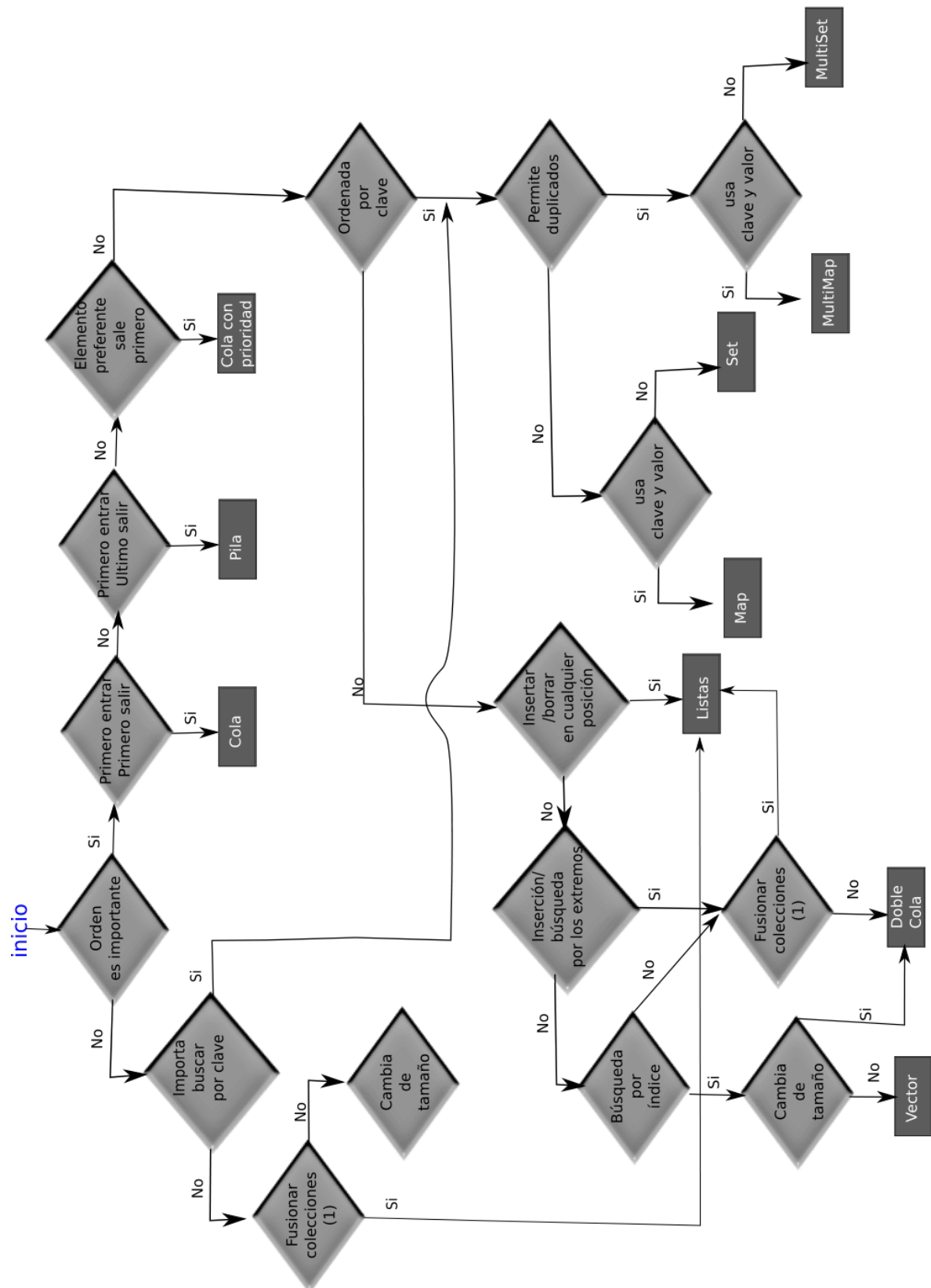


Figura 1.3: Estructuras de datos más eficientes dependiendo de las restricciones de los datos. (1) No se admite nueva reserva de memoria

