Estructuras de Datos Lineales y Arborescentes

Josefina Sierra Santibáñez

14 de abril de 2018

Estructuras de Datos Lineales y Arborescentes

En este tema se utilizan estructuras de datos genéricas ya implementadas (e.g. en STL de C++), pero no se implementan.

Un ejemplo de estructura de datos **genérica** es la clase `vector`: el tipo de los elementos almacenados en un vector es un parámetro.

```
vector<bool> subconjunto;
vector<Estudiant> grupo40;
```

- Un vector permite consultar cualquier elemento sin necesidad de modificar dicho vector. En algunas estructuras de datos consultar todos los elementos implica vaciar la estructura.
- Una consecuencia de trabajar con estructuras de datos genéricas es que ningún método que dependa de la clase de los elementos de la estructura puede ser genérico.
- Por este motivo especificamos e implementamos fuera de la clase `vector` operaciones de lectura y escritura para cada clase de elementos (e.g. `vectorIOint.hh` y `vectorIOint.cc`, o `vectorIOestudiant.hh` y `vectorIOestudiant.cc`).

Estructura de Datos Lineal

Una estructura de datos es **lineal** si sus elementos forman una secuencia $e_1, e_2, \ldots, e_n$, donde $n \geq 0$.

- Si $n = 0$, la estructura es vacía.
- Si $n = 1$, la estructura tiene un elemento, que es al mismo tiempo el primero y el último.
- Si $n = 2$, tiene dos elementos: el primero, sin antecesor y que tiene como sucesor al segundo; y el segundo, sin sucesor y que tiene como antecesor al primero.
- Si $n > 2$, todos los elementos a partir del segundo hasta el penúltimo tienen antecesor y sucesor.

Las distintas estructuras de datos lineales que veremos se diferencian por la forma en que se puede acceder a sus elementos.

Especificación de la Clase Genérica Stack (Pila)

Es una estructura lineal de tipo LIFO (*Last In, First Out*), i.e. permite consultar y eliminar solamente el último elemento añadido.
En la primera línea de la especificación de **stack** (`pila.hh`) aparece la palabra **template**. El parámetro **T** representa el **tipo de elementos** que contiene la pila. Para usar la clase **stack** es preciso indicar qué tipo de elementos contiene, y en los lugares donde aparezca un parámetro **T** sustituir éste por una expresión del tipo elegido.

```
stack();
stack(const stack& original);
void push(const T& x);
void pop();
T& top();
const T& top() const;
bool empty() const;
int size() const;
```

Para instanciar pilas de un tipo concreto lo haremos como en el caso de los vectores (e.g. `stack<int> p; stack<Estudiant> q;`).

Especificación de la Clase Genérica Queue (Cola)

Es una estructura de tipo FIFO (*First In, First Out*), i.e. permite añadir elementos sólo al final, y consultar y eliminar únicamente el primer elemento añadido.

El parámetro **T** representa el **tipo de elementos** que contiene la cola. Para usar la clase **queue** es preciso indicar qué tipo de elementos contiene, y en los lugares donde aparezca un parámetro **T** sustituir éste por una expresión del tipo elegido.

```
queue();
queue(const queue& original);
void push(const T& x);
void pop();
T& front();
const T& front() const;
bool empty() const;
int size() const;
```

Para instanciar colas de un tipo concreto lo haremos como en el caso de los vectores (e.g. `queue<int> p; queue<Estudiant> q;`).

Listas e Iteradores

Una **lista** es una estructura de datos lineal que permite acceder a sus elementos de forma **secuencial**. Se puede comenzar a consultar por los extremos, y desde cada elemento se puede acceder al anterior y al siguiente. Además, en una lista se pueden insertar, modificar y eliminar elementos en cualquier posición a la que apunte un iterador.

Un **iterador** nos permite desplazarnos por una lista, consultar sus elementos y, dependiendo del tipo de iterador, modificarlos.

Una lista de enteros y un iterador se declaran de la forma siguiente:

```
list<int> t;
list<int>::iterator it;
```

Un iterador no **se asocia** a una lista concreta sino **a la clase** (e.g. `list<int>`). Esto implica que se puede utilizar el mismo iterador `it` para recorrer distintas listas de enteros.

Listas e Iteradores

Las listas poseen un método **begin()** que devuelve un iterador que apunta al primer elemento de una lista no vacía. Existen **iteradores constantes** que impiden modificar el objeto apuntado por el iterador. Los iteradores constantes deben usarse con **parámetros const**.

```
it = t.begin();
list<Estudiant> c;
list<Estudiant>::const_iterator it2=c.begin();
```

Con el iterador `it2` nos podemos mover por la lista `c` y consultar sus elementos, pero no modificarlos.

Podemos hacer que un iterador se mueva por una lista con los operadores **++it** y **--it**, que permiten moverlo de manera que apunte al siguiente elemento y al anterior, respectivamente.

Podemos desreferenciar un iterador **para obtener el objeto al que apunta** utilizando el **operador *** . Por ejemplo, `*it2` es el objeto al que apunta el iterador `it2`.

Listas e Iteradores

Si deseamos consultar el DNI del objeto de tipo estudiant al que apunta it2, utilizaremos la siguiente expresión **(*it2).consultar_DNI()**. Esta última expresión es equivalente a **it2->consultar_DNI()**.

Las listas también poseen un método **end()** que devuelve un iterador que apunta a una posición posterior al último elemento de la lista. Intentar desreferenciar **end()** genera un error de ejecución.

Se pueden realizar asignaciones de iteradors con el operador = (e.g. `it1 = it2;`), y comparar dos iteradores con == y **!=**. Para recorrer un contenedor **t**, se suele comparar el iterador **it** con el que nos movemos con **t.end()**, que marca el final del contenedor.

```
list<int>::iterator it = t.begin();
while (it != t.end()) {
    cout << *it;
    ++it;
}
```

Si **t** es una lista vacía, entonces **t.begin() == t.end()**.

Especificación de la Clase Genérica List (Lista)

Una lista es una estructura de datos lineal que permite acceder a sus elementos de forma **secuencial**.

- Se puede comenzar a consultar por los extremos.
- Desde cada elemento se puede acceder al anterior y al posterior.
- Permite: 1) insertar elementos delante de cualquier lugar de la lista al que apunte un iterador; 2) consultar, modificar o borrar cualquier elemento al que apunte un iterador.

Especificación de **list** (`lista.hh`). **T** es el **tipo de los elementos**.

```
list();
list(const list& original);
void clear();
void insert(iterator it, const T& x);
iterator erase(iterator it);
bool empty() const;
int size() const;
```

Para instanciar listas de un tipo concreto lo haremos como en el caso de los vectores (e.g. `list<int> p; list<Estudiant> q;`).

Ejemplo: Suma de los Elementos de una Lista

```
int suma(const list<int>& p) {
/* Pre: cierto  */
/* Post: el resultado es la suma de los elementos
de p */
  int s = 0;
  list<int>::const_iterator it = p.begin();
/* Inv: it apunta a un elemento de p o a p.end();
s es la suma de los elementos de la sublista de
p comprendida entre p.begin() y el elemento
anterior al que apunta it. */
  while (it != p.end()) {
      s += *it;
      ++it;
  }
  return s;
}
```

Arboles

Un árbol es una estructura de datos **no lineal**, porque sus elementos pueden tener más de un sucesor (ver figura 5.1 de **estr_arb.pdf**).

- Los elementos de un árbol se denominan **nodos**.
- Un árbol no vacío tiene un elemento distinguido denominado **raíz**, que es el único nodo que se puede consultar directamente.
- El nodo raíz de un árbol está conectado con cero o varios **subárboles** del árbol que se denominan **hijos**. Los hijos de un árbol se pueden consultar directamente.
- Un nodo que no está conectado con ningún subárbol no vacío, es decir, que no tiene ningún sucesor, se denomina **hoja**.
- Un **camino** es una sucesión de nodos que van desde la raíz de un árbol hasta una de sus hojas.
- La **altura** de un árbol es la longitud de su camino más largo.

Arboles

Existen distintos tipos de árboles:

- **Arbol General**
  - Los hijos de un árbol general no vacío pueden ser árboles con cero o varios hijos, pero no pueden ser árboles vacíos (ver figura 10.4 de **implrec.pdf**).
  - Los hijos de un árbol general no vacío puede tener diferente número de hijos, e.g. un hijo del árbol puede tener dos hijos, otro hijo del mismo árbol puede tener tres hijos, y otro hijo del mismo árbol ningún hijo.
  - La raíz de un árbol general no vacío que no tiene ningún hijo es una hoja.

- **Arbol n-ario**
  - Los hijos de un árbol n-ario no vacío pueden ser árboles vacíos o árboles no vacíos (ver figura 10.3 de **implrec.pdf**).
  - Los hijos de un árbol n-ario no vacío tiene todos el mismo número de hijos (**n**). El valor **n** es la aridad del árbol
  - La raíz de un árbol n-ario cuyos hijos son todos árboles vacios es una hoja.

Especificación de la Clase Genérica Arbol Binario (BinTree)

Un **árbol binario** es un árbol n-ario cuya aridad es dos, i.e. es un árbol vacío o un árbol que tiene exactamente dos hijos.

Especificación de `BinTree` para usar en clase de teoría. Es preciso escribir `#include "BinTree.hh"` en la cabecera del programa.

```
BinTree();
BinTree(const T& x);
BinTree(const T& x, const BinTree& left,
        const BinTree& right);
BinTree left() const;
BinTree right() const;
const T& value() const;
bool empty() const;
```

Para instanciar árboles binarios de un tipo concreto lo haremos como en el resto de estructuras de datos genéricas (e.g. `BinTree<int> a; BinTree<Estudiant> b;`).

Ejemplo: Búsqueda en un Arbol

```
bool cerca(const BinTree<int>& a, int x) {
/* Pre: cierto */
/* Post: El resultado indica si x esta en a. */
  bool t;
  if (a.empty()) t = false;
  else if (a.value() == x) t = true;
  else {
    t = cerca(a.left(),x);
    /* HI: t indica si x esta en el hijo
       izquierdo de a. */
    if (not t) t = cerca(a.right(),x);
    /* HI: t indica si x esta en el hijo
       derecho de a. */
  }
  return t;
}
```

Recorridos de un árbol

- **Profundidad**
  - **Pre-orden**: se visita en primer lugar la raíz, después el hijo izquierdo en pre-orden, y a continuación el hijo derecho en pre-orden.
  - **In-orden**: : se visita en primer lugar el hijo izquierdo en in-orden, después la raíz, y a continuación el hijo derecho en in-orden.
  - **Post-orden**: se visita en primer lugar el hijo izquierdo en pot-orden, después el hijo derecho en post-orden, y a continuación la raíz.

  Podéis ver ejemplos de estos recorridos en las páginas 6 y 10 de **est_arb.pdf**.
- **Anchura**: la profundidad o nivel de un nodo en un árbol es su distancia al nodo raíz. En el recorrido en anchura (o **por niveles**) se visitan todos los nodos de nivel k antes de visitar los nodos de nivel $k+1$. Dentro de cada nivel los nodos se visitan de izquierda a derecha.

Ejemplo: recorrido en pre-orden

```
void pre_orden(const BinTree<int>& a,
               list<int>& r) {
/* Pre: r = R */
/* Post: r es igual a R seguida de los valores
de los nodos de a recorridos en pre-orden. */
  if (not a.empty()) {
     r.insert(r.end(), a.value());
     pre_orden(a.left(), r);
     /* HI: r es igual a R seguida del valor de
     la raiz de a y a continuacion de los nodos
     del hijo izquierdo de a en pre-orden. */
     pre_orden(a.right(), r);
     /* HI: r es igual a R seguida la raiz de a,
     de los nodos del hijo izquierdo de a en
     pre-orden, y a continuacion de los nodos
     del hijo derecho de a en pre-orden. */
  }
}
```

Ejemplo: recorrido en in-orden

```
void in_orden(const BinTree<int>& a,
              list<int>& r) {
/* Pre: r = R */
/* Post: r es igual a R seguida de los valores
de los nodos de a recorridos en in-orden. */
  if (not a.empty()) {
     in_orden(a.left(), r);
     /* HI: r es igual a R seguida de los nodos
     del hijo izquierdo de a en in-orden. */
     r.insert(r.end(), a.value());
     in_orden(a.right(), r);
     /* HI: r es igual a R seguida de los nodos
     del hijo izquierdo de a en in-orden, del
     valor de la raiz de a, y a continuacion de
     los nodos del hijo derecho de a en in-orden.
  }
}
```

Ejemplo: recorrido en anchura

```
void anchura(const BinTree<int>& a, list<int>& r)
/* Pre: r = R */
/* Post: r es igual a R seguida de los valores
de los nodos de a recorridos en anchura. */
 if (not a.empty()) {
    queue<BinTree<int> > c;
    c.push(a);
    while (not c.empty()) {
       BinTree<int> p = c.front();
       c.pop();
       r.insert(r.end(), p.value());
       if (not p.left().empty()) c.push(p.left());
       if (not p.right().empty()) c.push(p.right())
    }
 }
}
```

Arbol de sumas

```
BinTree<int> a_sumas(const BinTree<int>& a) {
/* Pre: cierto */
/* Post: El resultado es el arbol de sumas de a.
  BinTree<int> as;
  if (not a.empty()) {
      BinTree<int> al = a_sumas(a.left());
/* HI: al arbol de sumas del hijo izquierdo de a.
      BinTree<int> ar = a_sumas(a.right());
/* HI: ar arbol de sumas del hijo derecho de a. *
      int raiz = a.value();
      if (not al.empty()) raiz += al.value();
      if (not ar.empty()) raiz += ar.value();
      as = BinTree<int>(raiz, al, ar);
  }
  return as;
}
```

Suma x a los nodos de un árbol

```
void suma_k(int x, BinTree<int>& a) {
/* Pre: a = A */
/* Post: Cada nodo de a es igual que su
   correspondiente en A mas x. */
  if (not a.empty()) {
     BinTree<int> a_left = a.left();
     BinTree<int> a_right = a.right();
     suma_k(x, a_left);
/* HI: a_left es el hijo izquierdo de A mas x. */
     suma_k(x, a_right);
/* HI: a_right es el hijo derecho de A mas x. */
a = BinTree<int>(a.value() + x, a_left, a_right);
  }
}
}
```

Corrección de Programas Iterativos

Un programa iterativo suele tener la siguiente forma

```
/* Pre */
inicializaciones
while (B) {
    S
}
/* Post */
```

Para demostrar su **corrección** es preciso probar que "si al inicio de la ejecución las variables cumplen la precondición, al finalizar la ejecución del bucle las variables cumplirán la postcondición".

Para conseguirlo es esencial encontrar la **invariante**: una propiedad de las variables que se ha de mantener en cada iteración.

La invariante permite transmitir información de la precondición a la postcondición.

Corrección de Programas Iterativos

Concretamente, debemos demostrar que

- $\{Pre\}$ *Inicializaciones* $\{Invariante\}$

  Si las variables cumplen la precondición, después de realizar las instrucciones de inicialización anteriores al bucle, las variables cumplirán la invariante.

- $\{Invariante \wedge B\}\ S\ \{Invariante\}$

  Si las variables cumplen la invariante y la condición de entrada al bucle, después de ejecutar las instrucciones del cuerpo del bucle, las variables seguirán cumpliendo la invariante.

- $\{Invariante \wedge \neg B\} \Rightarrow \{Post\}$

  Si las variables cumplen la invariante pero no la condición de entrada al bucle, entonces las variables deben cumplir la post-condición.

Corrección de Programas Iterativos

Para demostrar la corrección de un algoritmo debemos comprobar también que el bucle **termina en un número finito de pasos**.

Concretamente, hemos de encontrar una **función de cota** $t(\vec{x})$, definida en términos de una o más variables $\vec{x}$ del bucle, que cumpla lo siguiente

- $\mathit{Invariante} \Rightarrow t(\vec{x}) \in N$
  $t(\vec{x})$ devuelve un valor natural
- $\{\mathit{Invariante} \wedge B \wedge t(\vec{x}) = T\}\ \mathrm{S}\ \{t(\vec{x}) < T\}$
  $t(\vec{x})$ decrece en cada iteración

Esquema de Justificación de un Algoritmo Iterativo

Precondición y **postcondición** se escriben en la especificación de la función o acción.

La **invariante** se escribe justo encima de la instrucción **while**. Si hay varios `while` anidados, se escribe una invariante distinta para cada `while` y una justificación separada para cada uno de ellos.

Además de estos tres elementos, es imprescindible describir

- Inicializaciones
- Condición de Salida del Bucle ($\neg B$)
- Cuerpo del Bucle
- Terminación (función de cota t y decrecimiento)

En el archivo **it_rec_corr.pdf** del material docente de la página web de PRO2 podéis ver ejemplos de justificación de algoritmos iterativos que operan con vectores y matrices.

Consideraciones generales

En las operaciones recursivas pasamos por referencia pilas y colas, aunque sean parámetros de entrada, para evitar que se haga una copia del argumento en cada llamada recursiva. Listas y árboles pueden pasarse por referencia constante.

Para referirnos al **estado inicial** de un parámetro utilizaremos la expresión `nom_estructura = NOM_ESTRUCTURA` en la precondición de una operación (e.g. `p=P, q=Q`). El estado del parámetro en cualquier otro momento de la ejecución de la operación se representará mediante `nom_estructura` en minúsculas.

Si un programa realiza llamadas incorrectas a operaciones, como consultar o extraer un elemento de una pila vacía, es posible que el programa continue ejecutándose, pero comportándose de forma imprevisible. Para evitar esto se recomienda compilar con la opción **-D_GLIBCXX_DEBUG** cuando trabajemos con objetos de las clases `stack`, `queue`, `list`, `vector` o `BinTree`.

Restricciones para Facilitar la Justificación

Para facilitar la justificación de los programas que diseñemos, impondremos las siguientes restricciones en su codificación

- Sólo podemos hacer iteraciones con la instrucción **while**.
- Las operaciones booleanas no se evalúan con prioridad (i.e. de forma corto-circuitada).
- En las funciones sólo se puede usar una instrucción de tipo **return**, que debe estar al final del cuerpo de la función.

Cualquier código que no cumpla alguna de estas restricciones se puede transformar fácilmente en uno equivalente que sí las cumpla.

Ejemplo: Suma de los Elementos de una Lista

```
int suma(const list<int>& p) {
/* Pre: cierto  */
/* Post: el resultado es la suma de los elementos
de p */
  int s = 0;
  list<int>::const_iterator it = p.begin();
/* Inv: it apunta a un elemento de p o a p.end();
s es la suma de los elementos de la sublista de
p comprendida entre p.begin() y el elemento
anterior al que apunta it. */
  while (it != p.end()) {
      s += *it;
      ++it;
  }
  return s;
}
```

Ejemplo: Suma de los Elementos de una Lista

Precondición, postcondición e **invariante** escritas en la función.

- **Inicializaciones**: Inicialmente no se ha tratado ningún elemento de `p`. Si asignamos `it=p.begin()` y `s=0` se satisface la invariante.
- **Condición de salida del bucle**: La postcondición requiere que `s` sea la suma de todos los elementos de `p`. Por tanto, saldremos del bucle cuando `it == p.end()`, porque esta condición junto con la invariante implican la postcondición.
- **Cuerpo del bucle**: La forma más sencilla de avanzar es hacer que `it` apunte al siguiente elemento. Pero antes de avanzar, hemos de satisfacer la invariante sumando a `s` el elemento al que apunta `it`. La condición de entrada al bucle garantiza que `++it` satisface la primera parte de la invariante.
- **Terminación**: En cada iteración decrece la longitud de la sublista de `p` comprendida entre `it` y `p.end()`.

Ejemplo: Búsqueda en una Lista

```
bool cerca_i(const list<int>& c, int x) {
  /* Pre: cierto */
  /* Post: El resultado indica si x es un
           elemento de c o no */
  bool r = false;
  list<int>::const_iterator it = c.begin();
/* Inv: it apunta a un elemento de c o a c.end();
r <-> x esta en la sublista de c comprendida entr
c.begin() y el elemento anterior al elemento al
que apunta it. */
  while (not r and it != c.end()) {
    r = (*it == x);
    ++it
  }
  return r;
}
```

Ejemplo: Búsqueda en una Lista

Precondición, postcondición e **invariante** escritas en la función.

- **Inicializaciones**: Inicialmente no se ha comprobado ningún elemento de c. Por tanto, inicializamos `it` a `c.begin()`, cumpliendo la primera parte de la invariante. Para satisfacer la segunda parte de la invariante, `r` debe ser falso.
- **Condición de salida del bucle**: Si `it == c.end()`, por la invariante habremos tratado toda la lista y r indicará si x está en c. Por otra parte, si r pasa a ser cierta, por la invariante sabremos que x está en la parte tratada de c. La condición de salida es por tanto $(r \vee it == c.end())$.
- **Cuerpo del bucle**: La forma más natural de avanzar es hacer que el iterador apunte al siguiente elemento. Antes de avanzar, hemos de satisfacer la invariante: r indica si x está en la parte tratada de c antes de avanzar. Por tanto, debemos comprobar si el elemento al que apunta it es igual a x, y actualizar el valor de r, antes de hacer `++it`.
- **Terminación**: En cada iteración decrece la longitud de la sublista comprendida entre `it` y `c.end()`.

Ejemplo: Búsqueda en una Lista

```
bool cerca_i(const list<int>& c, int x,
             list<int>::const_iterator& it) {
/* Pre: cierto */
/* Post: El resultado indica si x es un elemento
de c o no. Si el resultado es cierto, it apunta
al primer elemento de c igual a x. */
  bool r = false;
  it = c.begin();
/* Inv: it apunta a un elemento de c o a c.end();
x no esta en la sublista de c comprendida entre
c.begin() y el elemento anterior al que apunta it
Si r es true, el elemento al que apunta it es x.
  while (not r and it != c.end()) {
    if (*it == x) r = true;
    else ++it
  }
  return r;
}
```

Ejemplo: Búsqueda en una Lista

Precondición, postcondición e **invariante** escritas en la función.

- **Inicializaciones**: Inicialmente no se ha tratado ningún elemento de c. Inicializando `it` a `c.begin()` satisfacemos la primera y la segunda parte de la invariante. Para satisfacer la tercera parte, basta asignar a `r` el valor `false`.
- **Condición de salida del bucle**: Si `it == c.end()`, por la invariante sabemos que x no está en c. Por otra parte, si `r=true`, la invariante implica que el elemento al que apunta `it` es `x`. Luego la condición de salida debe ser $r \vee it == c.end()$.
- **Cuerpo del bucle**: Por la invariante sabemos que `x` no está en la sublista comprendida entre `c.begin()` y el elemento anterior al que apunta `it`. Por la condición de entrada al bucle `it != c.end()` y `r` es falso. Necesitamos comprobar por tanto si el elemento al que apunta `it` es `x`. Si no lo es, basta avanzar `it` para que se cumpla la invariante. En otro caso, debemos asignar `r=true` para poder salir del bucle y cumplir la invariante.
- **Terminación**: En cada iteración decrece la longitud de la sublista comprendida entre `it` y `c.end()`, o se asigna `r=true`.

Ejemplo: Sumar k a cada Elemento de una Lista

```
void suma_k(list<int>& p, int k) {
/* Pre: p = P  */
/* Post: Cada elemento de p es la suma de k y
el elemento que en P ocupa la misma posicion */
  list<int>::iterator it = p.begin();
/* Inv: it apunta a un elemento de p o a p.end();
Cada elemento de la sublista de p comprendida
entre p.begin() y el elemento anterior al que
apunta it es igual a k mas el elemento correspon-
diente de P. Cada elemento de la sublista de p
comprendida entre it y p.end() es igual a su
correspondiente en P. */
  while (it != p.end()) {
      *it += k;
      ++it;
  }
}
```

Ejemplo de justificación de corrección

Precondición, postcondición e **invariante** escritas en la función.

- **Inicializaciones**: Si asignamos `it = p.begin()`, la invariante dice que `p` y `P` deben ser iguales. Lo cual es cierto por la precondición.
- **Condición de salida del bucle**: La postcondición requiere cada elemento de `p` sea la suma de `k` y el correspondiente elemento de `P`. La invariante implica que esto se cumple cuando `it == p.end()`.
- **Cuerpo del bucle**: La forma más natural de avanzar es que `it` apunte al siguiente elemento. Pero antes de avanzar debemos satisfacer la invariante sumando `k` al elemento al que apunta `it`. La condición de entrada al bucle garantiza que cuando ejecutamos `++it`, se cumple la primera parte de la invariante.
- **Terminación**: En cada iteración decrece la longitud de la sublista comprendida entre `it` y `p.end()`.

Especificación ⇒ Diseño Inductivo de Algoritmos Iterativos

1. Caracterizar la **invariante** a partir de la postcondición. Dado que la postcondición describe el objetivo del bucle, se puede considerar la invariante como una forma débil de la postcondición: es **la postcondición con respecto a la parte tratada** de los datos en un punto intermedio de la ejecución.
2. Deducir a partir de la invariante el estado en el que termina la ejecución, i.e. la **condición de terminación del bucle**. La negación de esta condición es la expresión `B` en `while (B)`.
3. Ver qué instrucciones necesita el **cuerpo del bucle** para que, si a la entrada se cumple la invariante, se cumpla también al final de cada iteración. Dado que cada iteración nos debe acercar al final de la ejecución del bucle, hemos de encontrar una **función de cota** que decrezca en cada iteración.
4. Deducir las instrucciones de **inicialización** que hacen que se cumpla la invariante antes de entrar en el bucle.

Ver ejemplo *prefijo de suma máxima de un vector* en **iteracio.pdf**.

Justificación de la Corrección de Algoritmos Recursivos

La forma más sencilla de función recursiva es la siguiente.

- Si se cumple la condición $c(x)$, ejecutamos las instrucciones $d(x)$.
- Si no, ejecutamos otras instrucciones entre las cuales hay una llamada a la misma función f cuyos argumentos son el resultado de aplicar una función de simplifiación **simp** a los datos originales.

```
Tipus2 f(Tipus1 x) {
/* Pre: Pre(x) */
/* Post: Post(x,s) */
    Tipus2 r, s;
    if (c(x)) s = d(x); // caso directo
    else {
        r = f(simp(x));      // caso recursivo
        s = h(x,r);
    }
    return s;
}
```

Justificación de la Corrección de Algoritmos Recursivos

En general, puede haber más de un caso directo y más de un caso recursivo.

- Cada caso directo constará de su condición de entrada c_i y de sus instrucciones directas d_i.
- Cada caso recursivo constará de su condición de entrada c_j, y de sus funciones $simp_j$ y h_j.

Justificación de la Corrección de Algoritmos Recursivos

Para demostrar la **corrección de f**, i.e. "si al inicio de la ejecución de f las variables cumplen la precondición, al final de la ejecución de f cumplirán la postcondición", es preciso probar lo siguiente:

- **Casos sencillos**: Demostrar que, si las variables cumplen la precondición y la condición $c(x)$, después de ejecutar las instrucciones $s = d(x)$, el resultado s y x cumplen la postcondición.

$$Pre(x) \wedge c(x) \Rightarrow Post(x, s)$$

- **Casos recursivos**: En estos casos se realiza un llamada a la misma función f con un argumento *"menor"* que el dato original.

Justificación de la Corrección de Algoritmos Recursivos

En cada caso recursivo se deben realizar los siguientes pasos

1. Demostrar que, si $Pre(x) \wedge \neg c(x)$, entonces $simp(x)$ está bien definida y cumple la precondición de f, i.e $Pre(simp(x))$.
2. Asumir la **hipótesis de inducción** (**HI**):

   $$\{Pre(simp(x))\} \quad r = f(simp(x)) \quad \{Post(simp(x), r)\}$$

   Si $simp(x)$ cumple la precondición, después de la llamada recursiva $r = f(simp(x))$, el resultado r y $simp(x)$ cumplen la postcondición $Post(simp(x), r)$.
3. Demostrar que, si
   - x cumple la precondición $Pre(x)$
   - x cumple la condición de entrada al caso recursivo, y
   - asumimos la conclusión de la **HI**, i.e. que después de la llamada recursiva $simp(x)$ y r cumplen la postcondición,

   entonces, después de ejecutar las instrucciones $s = h(x, r)$, el resultado s y x cumplen la postcondición $Post(x, s)$.

$$Pre(x) \wedge \neg c(x) \wedge Post(simp(x), r) \Rightarrow Post(x, s)$$

Justificación de la Corrección de Algoritmos Recursivos

- **Terminación**: Encontrar una **función de cota** $t(x)$, definida en términos de los parámetros de la función, que decrezca en cada llamada recursiva, para garantizar que no se genera un número infinito de llamadas recursivas.

  La función de cota $t(x)$ debe cumplir que

  - $Pre(x) \Rightarrow t(x) \in N$, devuelve un natural; y
  - $Pre(x) \wedge \neg c(x) \Rightarrow t(simp(x)) < t(x)$, su valor decrece en cada llamada recursiva.

Esquema de Justificación de un Algoritmo Recursivo

Precondición y **postcondición** se escriben en la especificación de la función o acción.

La **hipótesis de inducción** se escribe en el cuerpo de la función inmediatamente después de cada llamada recursiva.

Además es imprescindible justificar

- Casos Sencillos
- Casos Recursivos
- Terminación

Ejemplo: Búsqueda en un Arbol

```
bool cerca(const BinTree<int>& a, int x) {
/* Pre: cierto */
/* Post: El resultado indica si x esta en a. */
  bool t;
  if (a.empty()) t = false;
  else if (a.value() == x) t = true;
  else {
    t = cerca(a.left(),x);
    /* HI: t indica si x esta en el hijo
       izquierdo de a. */
    if (not t) t = cerca(a.right(),x);
    /* HI: t indica si x esta en el hijo
       derecho de a. */
  }
  return t;
}
```

Ejemplo: Búsqueda en un Arbol

Precondición y **postcondición** se especifican en la cabecera de la función, **hipótesis de inducción** después de cada llamada recursiva.

Casos Sencillos:

- Si a es vacío, sabemos que x no puede estar en a.
- Si x es igual a la raíz de a, sabemos que x está en a.

Caso Recursivo: Si x es diferente de la raíz de a, podemos consultar sus hijos izquierdo y derecho mediante `left` y `right`.

- Por hipótesis de inducción, podemos determinar si x está en el hijo izquierdo con la llamada recursiva `cerca(a.left(),x)`.
- Si el resultado de esta llamada recursiva es `true`, x está en a; en otro caso, por hipótesis de inducción, podemos determinar si x está en el hijo derecho con `cerca(a.right(),x)`.
- Si el resultado de esta llamada recursiva es `true`, x está en a; en otro caso, podemos concluir que x no está en a, porque no es igual a la raíz y no está en ninguno de sus hijos.

Terminación: En cada llamada recursiva decrece el tamaño (número de nodos) del primer argumento.

Ejemplo: Búsqueda en una Lista (diseño recursivo)

```
bool cerca(const list<int>& s, int x) {
/* Pre: cierto.
   Post: El resultado indica si x esta en s. */
  list<int>::const_iterator it = s.begin();
  return cerca_r(s,x,it);
}

int main() {
  list<int> s;
  lee(s);
  int x;
  cin >> x;
  list<int>::const_iterator it = s.begin();
  if (cerca_r(s,x)) {
    cout << "encontrado" << endl;
  }
  else cout << "no encontrado" << endl;
}
```

```
bool cerca_r(const list<int> &s, int x,
     list<int>::const_iterator& it) {
/* Pre: IT apunta a un elemento de s o a s.end().
/* Post: El resultado indica si x es un elemento
de la sublista de s que comienza en el elemento
al que apunta IT y termina en s.end(). Si el
resultado es cierto, it apunta a x. */
  bool r;
  if (it == s.end()) r = false;
  else if (*it == x) r = true;
  else {
     it++;
     r = cerca_r(s,x,it);
/* HI: r indica si x es un elemento de sublista
comprendida entre el siguiente elemento al que
apunta IT y s.end(). Si el r es cierto, it apunta
a un elemento de dicha sublista igual a x.*/
  }
  return r;
}
```

Ejemplo: Búsqueda en una Lista

Casos Sencillos:

1. Si `IT == s.end()`, el resultado es falso, porque la sublista es vacía.
2. Si el elemento al que apunta `IT` es igual a `x`, el resultado es cierto.

Caso Recursivo: Si `IT != s.end()` y el elemento al que apunta `IT` no es `x`, el resultado depende de si `x` es un elemento de la sublista comprendida entre el elemento siguiente al que apunta `IT` y `s.end()`. Si avanzamos it, se cumple la precondición de la llamada recursiva porque IT != s.end(), it apunta al siguiente elemento al que apunta IT, y por hipótesis de inducción podemos suponer que el resultado de la llamada recursiva `cerca_r(s,x,it)` determina si `x` es un elemento de la sublista comprendida entre dicho elemento y `s.end()`.

Terminación: En cada llamada recursiva la longitud de la sublista comprendida entre `IT` y `s.end()` decrece.