Estructura de Datos y Algoritmos

Recursividad

1. Algoritmos Recursivos

1.1. Introducción

Se dice que un objeto es recursivo si en parte está formado por si mismo o se define en función de si mismo. Una función recursiva o recurrente es una función que se llama a si misma.

La recursión se encuentra en matemáticas. Ejemplos son los números naturales, las listas y ciertas funciones como por ejemplo factorial.

1. Números naturales:
   a) 0 es un número natural
   b) El sucesor de un número natural es otro número natural
2. Listas
   a) NULL es la lista vacia
   b) Si l es una lista y a un elemento entonces el resultado de agregar a a l es una lista
3. Función factorial (para enteros no negativos)
   a) 0!=1
   b) si $n > 0$ entonces n! = n * (n-1)!

El poder de la recursión radica en la posibilidad de definir conjuntos infinitos mediante proposiciones finitas. De la misma manera un número infinito de cálculos puede describirse con un programa recursivo finito. En particular, los algoritmos recursivos son apropiados cuando el problema a resolver, la función por calcular o la estructura de datos por procesar ya están definidos en términos recursivos.

Los programas suelen estructurarse como funciones que se llaman entre si de manera disciplinada y jerarquica. El caso particular de función recursiva se llama a ella misma. Este llamado puede ser directo o indirecto a través de otra función.

Una función recursiva es invocada para solucionar un problema. Dicha función sabe como resolver los casos más sencillos llamados casos base. Si la función es llamada mediante un caso base, está simplemente devuelve un resultado. En otro caso la función llama una copia de ella misma que se encargará de resolver un problema más pequeño, a esto se le conoce como llamada recursiva o paso de recursión.

El paso de recursión se ejecuta mientras la llamada original de la función todavía no ha terminado de ejecutarse.

A semejanza de las proposiciones repetitivas, los procedimientos recursivos introducen la posibilidad de computaciones que no terminan. Para que la recursión termine, la función debe llamarse a si misma mediante una versión mas sencilla del problema original hasta desembocar en los casos base.

1.2. Ejemplos

1.2.1. Ejemplo: serie de Fibonacci

C/C++ permiten procedimientos y funciones recursivos.

La serie de Fibonacci, comienza con 0 y 1 y tiene la propiedad de que cada número subsiguiente es la suma de los dos números previos.

La serie de Fibonacci puede definirse recursivamente como sigue:

$fibonacci(0) = 0$
$fibonacci(1) = 1$
$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$

la función recursiva correspondiente es

```
long fibonacci (long n)
{
    if (n==0 || n==1) return n;
    else
        return fibonacci (n-1) + fibonacci (n-2);
}
```

Hemos elegido long como tipo para el argumento y el resultado ya que estos tienden a crecer radpidamente.

Cada vez que fibonacci es invocada prueba de inmediato el caso base (n=0 o 1). Si la prueba es verdadera se devuelve n. Resulta interesante que si n es mayor que 1 se generan dos llamadas recursivas, cada una de las cuales es para un valor menor de n (problema más sencillo).

1.2.2. Ejemplo: Impresión de números

Supongamos tenemos un entero positivo N que deseamos imprimir. Nuestra función tendra el cabezal:

```
int imprimir_numero(int)
```

Además de imprimir, la función retorna -1 si el argumento es negativo y 0 en caso contrario.

Supongamos la unica función de entrada/salida disponible es la impresión de un simple caracter (putchar).

La recursión nos da una solución clara del problema. Para imprimir 76234 primero imprimimos 7623 y luego 4. Nos falta definir cual es el paso base: este será la impresión del digito del 0 al 9 cuando ya imprimimos todas los coeficientes de las potencias de 10.

Presentamos la función abajo:

```
int imprimir_numero(int N)
{
    if (N < 0) return -1;
    else
        if (N < 10) putchar(N % 10 + '0');
        else { imprimir_numero (N/10);
               putchar(N % 10 + '0');
               }
    return 0;
}
```

1.3. Operaciones sobre el TAD Lista

Consideremos el TAD lista de enteros con las operaciones:

1. Creo_vacia : devuelve la lista vacia.
2. Es_vacia(l): devuelve true si l es la lista vacia, false en otro caso.
3. Agrego_elemento(x,l): devuelve la lista con x como primer elemento y l como resto.
4. Primero(l):devuelve el primer elemento de la lista. Precondición: la lista no es vacia.
5. Resto(l): devuelve la lista l sin su primer elemento. Precondición: la lista no es vacia.

Creo_vacia y Agrego_elemento son constructoras. Es_vacia es observadora. Primero y Resto son selectoras. Veremos otras operaciones Extensoras:

1. Miembro(l,x): devuelve true si x está en l.
2. Largo(l): devuelve la cantidad de elementos de l.
3. Cantidad(l,x): devuelve la cantidad de ocurrencias de x en l.

```
bool Miembro(lista l,int x)
{
     if (Es_vacia(l)) return false;
     else if (Primero(l)==x) return true;
          else Miembro(Resto(l),x);
}


int Largo(lista l)
{
    if (Es_vacia(l)) return 0;
    else return 1+ Largo(Resto(l));
}
```

```
int Cantidad(lista l,int x)
{
    if (Es_vacia(l)) return 0;
    else if (Primero(l)==x) return 1+Cantidad(Resto(l),x);
         else return Cantidad(Resto(l),x);
```

1.4. Recursión en comparación con iteración

Tanto la iteración como la recursión se basan en una estructura de control: la iteración se vale de una estructura de repetición la recursión emplea una estructura de selección. Tanto la iteración como la recursión se basan en la repetición: la iteración utiliza explicitamente una estructura de repetición, la recursión logra la repetición a través de llamadas repetidas a una función.

La iteración y la recursión tienen una prueba de terminación: la iteración termina cuando la condición de terminación del ciclo falla, la recursión termina cuando se reconoce un caso base.

Tanto la iteración como la recursión pueden suceder infinitamente: ocurre un ciclo infinito en la iteración si la prueba de continuación del ciclo nunca se vuelve falsa; la recursión infinita sucede si el paso de recursión no reduce el problema para que converja al paso base.

Se acostumbra asociar un conjunto de objetos locales a un procedimiento, esto es, un conjunto de variables, constantes, tipos y procedimientos que se definen localmente a este procedimiento y que carecen de existencia o significado fuera de este procedimiento.

Cada vez que el procedimiento se activa de modo recursivo se crea un nuevo conjunto de variables locales. La misma regla se aplica a los parámetros. Cuando finaliza la ejecución correspondiente a la llamada recursiva se reanuda la ejecución del procedimiento desde la instrucción siguiente a la llamada recursiva.

La recursión tiene muchos aspectos negativos. Invoca repetidamente el mecanismo y en consecuencia la sobrecarga de las llamadas de función. Esto puede ser costoso tanto en tiempo de procesador como de espacio de memoria. Cada llamada recursiva provoca la creación de otra copia de las variables de la función, esto puede consumir una cantidad apreciable de memoria. La iteración sucede dentro de la función por lo que se evita la sobrecarga de las llamadas de función repetidas y la asignación de memoria extra.

Cualquier problema que puede resolverse por recursión también puede resolverse por iteración. Normalmente se prefiere un enfoque recursivo cuando refleja de manera más natural el problema y su resultado es un programa más facil de entender y depurar. Otra razón para seleccionar la solución recursiva es que tal vez no se encuentre una solución iterativa aparente.

Debemos evitar utilizar recursión en situaciones en las que es importante la eficiencia. Las llamadas recursivas tardan y consumen memoria.