Ejemplos

Implementación del TDA pila

A continuación se muestran dos maneras de implementar una pila: utilizando un arreglo y utilizando una lista enlazada. En ambos casos el costo de las operaciones es de O(1).

Implementación utilizando arreglos

Para implementar una pila utilizando un arreglo, basta con definir el arreglo del tipo de dato que se almacenará en la pila. Una variable de instancia indicará la posición del tope de la pila, lo cual permitirá realizar las operaciones de inserción y borrado, y también permitirá saber si la pila esta vacía, definiendo que dicha variable vale -1 cuando no hay elementos en el arreglo.

```
class PilaArreglo
{
 private Object[] arreglo;
 private int tope;
 private int MAX_ELEM=100; // maximo numero de elementos en la pila

 public PilaArreglo()
 {
  arreglo=new Object[MAX_ELEM];
  tope=-1; // inicialmente la pila esta vacía
 }

 public void apilar(Object x)
 {
  if (tope+1<MAX_ELEM) // si esta llena se produce OVERFLOW
  {
   tope++;
   arreglo[tope]=x;
  }
```

```
 }

 public Object desapilar()
 {
  if (!estaVacia()) // si esta vacia se produce UNDERFLOW
  {
    Object x=arreglo[tope];
    tope--;
    return x;
  }
 }

 public Object tope()
 {
  if (!estaVacia()) // si esta vacia es un error
  {
    Object x=arreglo[tope];
    return x;
  }
 }

 public boolean estaVacia()
 {
  if (tope==-1)
  {
    return true;
  }
  else
  {
    return false;
  }
 }
}
```

El inconveniente de esta implementación es que es necesario fijar de antemano el número máximo de elementos que puede contener la pila, MAX_ELEM, y por lo tanto al apilar un elemento es necesario controlar que no se inserte un elemento si la pila esta llena. Sin embargo, en Java es posible solucionar este problema creando un nuevo arreglo más grande que el anterior, el doble por ejemplo, y copiando los elementos de un arreglo a otro:

```
public void apilar(Object x)
{
 if (tope+1<MAX_ELEM) // si esta llena se produce OVERFLOW
 {
  tope++;
  arreglo[tope]=x;
 }
 else
 {
  MAX_ELEM=MAX_ELEM*2;
  Object[] nuevo_arreglo=new Object[MAX_ELEM];
  for (int i=0; i<arreglo.length; i++)
  {
   nuevo_arreglo[i]=arreglo[i];
  }
  tope++;
  nuevo_arreglo[tope]=x;
  arreglo=nuevo_arreglo;
 }
}
```

Implementación utilizando listas enlazadas

En este caso no existe el problema de tener que fijar el tamaño máximo de la pila (aunque siempre se está acotado por la cantidad de memoria disponible!). La implementación es bastante simple: los elementos siempre se insertan al principio de la lista (apilar) y siempre se extrae el primer elemento de la lista (desapilar y tope), por lo que basta con tener una referencia al principio de la lista enlazada. Si dicha

referencia es null, entonces la pila esta vacía.

```
class PilaLista
{
 private NodoLista lista;

 public PilaLista()
 {
  lista=null;
 }

 public void apilar(Object x)
 {
  lista=new NodoLista(x, lista);
 }

 public Object desapilar() // si esta vacia se produce UNDERFLOW
 {
  if (!estaVacia())
  {
   Object x=lista.elemento;
   lista=lista.siguiente;
   return x;
  }
 }

 public Object tope()
 {
  if (!estaVacia()) // si esta vacia es un error
  {
   Object x=lista.elemento;
   return x;
  }
```

```
 }

 public boolean estaVacia()
 {
  return lista==null;
 }
}
```

Dependiendo de la aplicación que se le de a la pila es necesario definir que acción realizar en caso de que ocurra overflow (rebalse de la pila) o underflow (intentar desapilar cuando la pila esta vacía). Java posee un mecanismo denominado excepciones, que permite realizar acciones cuando se producen ciertos eventos específicos (como por ejemplo overflow o underflow en una pila).

En ambas implementaciones el costo de las operaciones que provee el TDA tienen costo O(1).

Ejemplo de uso: eliminación de recursividad
Suponga que una función F realiza un llamado recursivo dentro de su código, lo que se ilustra en la siguiente figura:

Si la llamada recursiva es lo último que hace la función F, entonces dicha llamada se puede substituir por un ciclo while. Este caso es conocido como tail recursion y en lo posible hay que evitarlo en la programación, ya que cada llamada recursiva ocupa espacio en la memoria del computador, y en el caso del tail recursion es muy simple eliminarla. Por ejemplo:

```
void imprimir(int[] a, int j) // versión recursiva
{
 if (j<a.length)
 {
  System.out.println(a[j]);
  imprimir(a, j+1); // tail recursion
```

```
 }
}

void imprimir(int[] a, int j) // versión iterativa
{
 while (j<a.length)
 {
  System.out.println(a[j]);
  j=j+1;
 }
}
```

En el caso general, cuando el llamado recursivo se realiza en medio de la función F, la recursión se puede eliminar utilizando una pila.

Por ejemplo: recorrido en preorden de un arbol binario.

```
// "raiz" es la referencia a la raiz del arbol
// llamado inicial: preorden(raiz)

// version recursiva

void preorden(Nodo nodo)
{
 if (nodo!=null)
 {
  System.out.print(nodo.elemento);
  preorden(nodo.izq);
  preorden(nodo.der);
 }
}

// primera version iterativa
```

```
void preorden(Nodo nodo)
{
  Nodo aux;
  Pila pila=new Pila(); // pila de nodos
  pila.apilar(nodo);
  while(!pila.estaVacia()) // mientras la pila no este vacia
  {
    aux=pila.desapilar();
    if (aux!=null)
    {
      System.out.print(aux.elemento);
      // primero se apila el nodo derecho y luego el izquierdo
      // para mantener el orden correcto del recorrido
      // al desapilar los nodos
      pila.apilar(aux.der);
      pila.apilar(aux.izq);
    }
  }
}

// segunda version iterativa
// dado que siempre el ultimo nodo apilado dentro del bloque if es
// aux.izq podemos asignarlo directamente a aux hasta que éste sea
// null, es decir, el bloque if se convierte en un bloque while
// y se cambia el segundo apilar por una asignacion de la referencia

void preorden(Nodo nodo)
{
  Nodo aux;
  Pila pila=new Pila(); // pila de nodos
  pila.apilar(nodo);
  while(!pila.estaVacia()) // mientras la pila no este vacia
  {
```

```
   aux=pila.desapilar();
   while (aux!=null)
   {
     System.out.print(aux.elemento);
     pila.apilar(aux.der);
     aux=aux.izq;
   }
 }
}
```

Si bien los programas no recursivos son más eficientes que los recursivos, la eliminación de recursividad (excepto en el caso de tail recursion) le quita claridad al código del programa. Por lo tanto:

A menudo es conveniente eliminar el tail recursion.
Un método recursivo es menos eficiente que uno no recursivo, pero sólo en pocas oportunidades vale la pena eliminar la recursión.

Bibliografias

Seymour Lipschutz, Ph.D.Estructura de datos.Editorial Revolucionaria.2002

Fundamentals of Data Structures in C++. E. Horowitz, S. Sanhi, D. Mchta. Computer Science Press, 1995.

Data structures and algorithms. Aho .et al.