Estructura de Datos

Especificaciones

Pablo Riutort

7 de agosto de 2014

Índice general

Capítulo 1

Estructuras Lineales

1.1. Pila

1.1.1. Especificación

```
generic
  type item is private;

  bad_use: exception;
  space_overflow: exception;

procedure empty(p:out pila);
procedure push (p:in out pila; x:in item);
procedure pop(p:in out pila);
procedure top(p:in pila; x:out item);
function is_empty(p:in pila) return boolean;
```

1.1.2. Compacta

Esta implementación requiere de un índice y un array.

```
  max: natural:=100; --Debajo del item en el generic

private

  type index is integer range (0..max);
  type mem_space is new array (range 1...index'last) of item;

  type pila is
    record
      a: mem_space;
      top: index;
    end record;
```

1.1.3. Punteros

Para esta implementación hacemos uso de la estructura del nodo. El nodo será nuestra cima de la pila.

```
type node;
type pnode is access;

type pila is
  record
    top: pnode;
  end record;
```

1.1.4. Cursores

La implementación con cursores nos permite tener control de la memoria, controlando los bloques de memoria libres. Para esto, necesitamos definir la estructura de un bloque.

```
--La lista de bloques ira en un array
type index is integer range 0..max;

 type pila is
  record
    top: index;
  end record;

end dstack;
```

Ahora especificamos en el cuerpo el resto de estructuras necesarias.

```
package body dstack is

  type block is
    record
      x: item;
      n: index;
    end record;

  type mem_space is array (index range 1..index'last) of block;

  ms: mem_space;
  free: index;


--Funciones auxiliares para los bloques
procedure get_block(); --Coge un bloque de la lista de bloques vacios
    .
procedure release_block() --Vuelve a dejar un bloque en la lista de
    bloques vacios
```

Cursores VS Others

Así como en la implementación compacta definimos el tamaño del array al principio, este no crece ni decrece en función de los elementos, por tanto, no podemos solicitar memoria ni liberarla.

En el caso de los punteros, confiamos en el garbage colector para que haga las gestiones de memoria.

1.2. Cola

1.2.1. Especificación

```
generic
  type item is private;

bad_use: exception;
space_overflow: exception;

procedure empty(c: out cola);
procedure put(c: in out cola; x:in item);
procedure rem_first(c: in out cola);
function get_first(c: in cola, q:out index) return item;
function is_empty(c:in cola) return boolean;
```

1.2.2. Compacta

```
max: natural:=100;

type index is integer range (0..max);
type mem_space is new array range (1.. index'last) of item;

type cola is
  record
    p,q: index;
    a: mem_space;
    n: natural;
  end record;
```

1.2.3. Punteros

```
type node;
type pnode is acces node;

type node is
  record
    x: item;
    next: pnode;
  end record;

type cola is
  record
    p,q: pnode;
  end record;
```

1.2.4. Cursores

```
max: natural:=100;
type index is new integer range 0..max;

type cola is
  record
    p,q: index:=0;
  end record;
end dqueue;
```

Ahora especificamos en el cuerpo el resto de estructuras necesarias.

```
package body dqueue is

  type block is
    record
      x: item;
      n: index;
    end record;

  type mem_space is array (index range 1..index'last) of block;

  ms: mem_space;
  free: index;

--Funciones auxiliares para los bloques
procedure get_block(); --Coge un bloque de la lista de bloques vacios
    .
procedure release_block() --Vuelve a dejar un bloque en la lista de
    bloques vacios
```

1.3. Listas

```
generic
        MAX: natural:= 100;
        type item is private;
        with function "<" (X,Y: in element) return boolean;
package d_list is
        type list is private;
        procedure empty(t: out list);
        procedure put(l:in out list; x:in item);
        procedure remove(l:in out list; x:in item);
        procedure get(l:in list; x:out item);
        procedure iterate(l:in list);
```

1.3.1. Punteros

```
private
        type node;
        type pnode is acces node;
        type node is
                record
                        x: item;
                        next: pnode;
                end record;

        type list is
                first: pnode;
```

1.3.2. Compacta

```
private
        type container is array (0..MAX) of item;
        type list is
                record
                        num_elements: natural:=0;
                        elements: container;
                end record
```

Capítulo 2

Conjuntos

2.1. Especificación

Tenemos dos posibles, especificaciones: Set y Mapping

Set

```
generic
 type item is private;

space_overflow: exception;

procedure empty (s: out set);
procedure put (s: in out set; x:in item);
function is_in (s: in set; x:in item) return boolean;
procedure remove (s: in set; x: in set) ;
function is_empty (s:in set) return boolean;
```

Mapping

```
generic
  type key is private;
  type item is private;

space_overflow: exception;
alredy_exist: expcetion;
does_not_exist: exception;

procedure empty (s: out set);
procedure put (s: in out set; ;k:in key; x:in item);
procedure remove (s: in set; k:in key) ;
procedure get (s: in set; k:in key; x:out item);
procedure update (s:in out set; k:in ket; x:in set);
```

2.2. Árboles binarios

Requiere la especificación del mapping. Los siguientes árboles se basan en la especificación de este.

```
type node;
type pnode is access node;

type node is
 record
   k: key;
   x: item;
   fd,fe: pnode; --hijos derecho e izquierdo respectivamente
 end record;

 type set is
   record
     root: pnode;
   end record;

 end d_set;
```

2.2.1. AVL

Estos árboles se caracterizan porque tienen diferencias de alturas y necesitan estar balaceados.

```
type node;
type pnode is access node;

type node is
 record
   k: key;
   x: item;
   fd,fe: pnode;
   bal: -1...1 --factor de balaceo
 end record;

 type set is
   record
     root: pnode;
   end record;
```

2.2.2. Red-Black

Estos árboles, al igual que los AVL, necesitan ser balanceados. Siguen un criterio de colores para efectuar el balanceo.

```
type node;
type pnode is acces node;

type color is (red,black);

type node is
 record
   k: key;
   x: item;
   fd,fe: pnode;
   c: color;
```

```
  end record;

  type set is
    record
      root: pnode;
    end record;
```

2.2.3. B+

2.2.4. Tries

2.3. Hashing

2.3.1. Abierto

2.3.2. Cerrado

2.3.3. Extendible

2.4. Iteradores

Capítulo 3

Conjuntos especiales

3.1. Colas de prioridad

3.2. Relaciones de equivalencia

Capítulo 4

Grafos

4.1. Especificación

```
generic
        size_vertices:positive;
package d_graph is

        type vertex is new positive 1..size_vertices;
        type graph is limited private;
        type iterator is private;

        nv:constant:positive:=size_vertices;

        --operaciones de grafos
        procedure empty (g:out graph);
        procedure put_edge (g:in out, x:in vertex);
        procedure remove_edge (g:in out, x:in vertex);

        --operaciones de iteradores
        procedure first (g:in graph, x: out vertex, it: out iterator)
        procedure next (g: in  graph, it: in out iterator, x:out vertex
            );
        function is_valid (it: in itertator) return boolean;
        procedure get (g:in graph, it: in out iterator, x: out vertex);

end d_graph
```

Hace falta añadir un paquete de excepciones

4.2. Dijsktra

4.3. Warshal

4.4. Prim

4.5. Edmonds-Karp