Fundamentos de Análise e Deseño de Algoritmos e Estruturas de Dados

---

TEMA 2: ESTRUTURAS DE DADOS.

A **programación modular** é o estilo que se impón no traballo en grupo, basea-se na ideia de dividir o programa nunha série de módulos lòxicamente homoxéneos, cada un cumprindo unha parte definida do traballo que conduce ao resultado final. Esta técnica permite a reutilización de módulos e simplifica as labores de depuración de erros. Ademais, a programación modular minimiza o número de variábeis globais para evitar ao máximo os efeitos colaterais próprios das linguaxens imperativas.

TDA (Tipo de Dado Abstracto) é unha abstracción matemática, un conxunto de operacións asociadas a unha estrutura dada que o usuário non ha coñecer. Na definición do tipo de dado na sección type (morfoloxia do TDA) non se fai mención de como son esas operacións que definen o TDA, senón que estas van despois en forma de procedimentos e funcións (sintaxe do TDA). O usuário só coñece o conxunto de operacións que manexan o TDA e nengunha parte do código accede à implementación do TDA, só acceden internamente estas operacións predeterminadas.

As vantaxens de un TDA son:

-As operacións escreben-se unha soa vez e son empregadas polos restantes módulos do programa.

-As correccións nos módulos das operacións que definen ou TDA son transparentes ao usuário e aos programas que empreguen o TDA.

2.1.- Listas, pillas e colas.

Pillas e colas: conxuntos dinámicos que se diferencian pola eliminación de un elemento. Mentres que nunha pilla se elimina o mais recentemente insertado (LIFO), nunha cola elimina-se o elemento que leva mais tempo insertado (FIFO).

Operacións con pillas:

-apillar ou push (elem , pilla) mete elem en pilla.

-desapillar ou pop (elem , pilla) saca un elemento da pilla e mete-o en elem.

-cima (elem , pilla) devolve unha cópia do elemento mais recentemente insertado na pilla, pero sen sacá-lo da mesma, é dicer devolve unha cópia do elemento que está na cima da pilla, na variábel elem.

Implementación estática de pillas.

Define-se un vector S con un atributo cima_pilla(S) que aponta en todo momento ao último elemento insertado na pilla. Se cima_pilla(S) = 0, a pilla está valeira. Entón as operacións desapillar ou cima de unha pilla valeira producen un erro. Outro erro ven dado por apillar nunha pilla chea (cando cima_pilla(S) aponta ao maior índice do vector estático S). Hai que implementar sempre a verificación de erros, ainda que sobrecarga o tempo de execución de unha primitiva de un TDA.

2.1.1.- Pillas.

Implementación a base de vectores.

Inconveniente: saber o tamaño máximo da pilla a priori.

```
{-----------------------------------------------------------------}
tipopilla = tupla
                  cimadepilla : inteiro ; { 0 se pilla valeira }
                  vectordepilla : tábua [ 1 .. tammaxpilla ]
                                        de tipoelemento
             fintupla
{-----------------------------------------------------------------}
procedimento criarpilla ( p )
   p . cimadepilla ← 0
finprocedimento
{-----------------------------------------------------------------}
función pillavaleira ( p )
   devolver ( p . cimadepilla = 0 )
finfunción
{-----------------------------------------------------------------}
función cima ( p )
   se pillavaleira ( p )
   entón erro ( “pilla valeira” )
   senón devolver ( p . vectordepilla [ p . cimadepilla ] )
   finse
finfunción
{-----------------------------------------------------------------}
procedimento apillar ( x , p )
   se p . cimadepilla = tammaxpilla
   entón erro ( “pilla chea” )
   senón p . cimadepilla ← p . cimadepilla + 1 ;
        p . vector [ p . cimadepilla ] ← x
   finse
finprocedimento
{-----------------------------------------------------------------}
procedimento desapillar ( p )
   se pillavaleira ( p )
   entón erro ( “pilla valeira” )
   senón p . cimadepilla ← p . cimadepilla - 1
   finse
finprocedimento
{-----------------------------------------------------------------}
```

Cal é a diferenza entre procedimentos e funcións? En princípio, emprega-se unha función cando se van devolver os parámetros sen modificar, mentres que un procedimento si modifica os parámetros. Por iso pomos desapillar como procedimento, porque modificamos o parámetro pilla. Pola outra banda, o subprograma cima pomo-lo como función porque non modificamos o parámetro pilla e devolvemos o valor mais recentemente insertado na pilla -sen modificala-.

O tempo de execución teria que ser a priori constante, O(1), pero poderia ser O(n) cando non se pasa por referência (por referência é "con var") a pilla. Hai que pasar por referência a pilla sempre, ainda cando non se modifica (por exemplo en pillavaleira, que o usamos en muitas outras primitivas). Hai que indicar que se pasa por referência para mellorar o tempo de execución.

APLICACIÓNS DAS PILLAS

♦ **1ª aplicación das pillas**: Para verificar o equilíbrio de símbolos (corchetes, parénteses, chaves, pares begin-end, etc). Vexamos un esquema de este algoritmo (esquema é unha representación ainda mais abstracta que o pseudódigo):

```
-criar a pilla
-para cada carácter
     se é de apertura, apillá-lo
     se é de clausura, mirar se pilla é valeira
          se pilla é valeira, entón erro por peche sen apertura
          se pilla non valeira
               desapillar
               corresponde símbolo de peche co de apertura?
                    non corresponde, erro "non corresponde"
                    corresponde, está ben
-se pilla non é valeira, erro "quedou símbolo sen pechar"
```

Non é necesário pasar este esquema a pseudocódigo para calcular o seu tempo. "Para cada carácter" implica O(n). Usan-se as primitivas criarpilla, apillar, desapillar, pillavaleira, todas de O(1), daquela a orden total é O(n).

Trata-se de un algoritmo en liña (percorre a secuência de entrada de esqueda a direita sen necesidade de percorrê-lo de novo e en todo momento o algoritmo pode dar unha solución parcial en función do que xá leva lido).

♦ **2ª aplicación das pillas**: conversións de expresións post/prefixa a infixa e viceversa, infixa a post/prefixa.

(as vantaxens das notacións prefixa e postfixa son que non se necesita coñecer a prioridade dos operadores e que non se necesita usar parénteses)

O tempo de execución do algoritmo post $\rightarrow$ in é de O(n) porque os pasos do proceso son un para cada carácter, grazas a que as operacións elementais son O(1); de non ser asi este algoritmo seria como mínimo $O(n^2)$

POSTFIXA A INFIXA

A B C * D E F ^ / G * - H * +

A + (B*C - D/E^F*G)* H

---

esquema infixa → postfixa

```
criar pilla
percorrer de esquerda a direita, para cada elemento:
    se
    |    é operando
    entón
         directamente a saída
    se
    |    é operador
    entón
         se
         |    a prioridade é maior que a do operador na cima da pilla
         ou se
         |    hai un paréntese na cima
         entón
              apillar
         se
         |    a prioridade non é maior
         entón
              desapillar os operadores de prioridade superior ou igual
              e despois apillar o operador
         se
         |    é )
         entón
              desapillar todo até atopar (
desapillar todo o que quede na pilla
```

Usa unha pilla de operadores. Orden de prioridades de maior a menor:

( ) ^ * / + -

pero tratamos os parénteses como caso aparte.

exemplo:

a+b*c+(d*e+f)*g → a b c * + d e * f + g * +

anaco da cola percorrido	**pilla** (arriba= cima)	**saída**	comentário
a		a	operando sai directamente
a+	+	a	primeiro operador apilla-se
a+b	+	a b	operando sai directamente
a+b*	* +	a b	prioridade (*) > prioridade (+) apillar *
a+b*c	* +	a b c	operando sai directamente
a+b*c**+**	 **+**	a b c * +	prioridade (**+**) non > prior. (*) desapillar operadores de ≥ prio.

			apillar **+**
a+b*c+(	(+	a b c * +	prioridade (() > prioridade (+) apillar (
a+b*c+(d	(+	a b c * + d	operando sai directamente
a+b*c+(d*	* (+	a b c * + d	"(" non se sacan excepto cando se procesa un ")" apillar *
a+b*c+(d*e	* (+	a b c * + d e	operando sai directamente
a+b*c+(d*e**+**	**+** (+	a b c * + d e *	prioridade (+) non > prior. (*) desapillar operadores de ≥ prio. apillar **+**
a+b*c+(d*e+f	+ (+	abc*+de*f	operando sai directamente
a+b*c+(d*e+f)	+	abc*+de*f+	")"→desapillar até atopar "("
a+b*c+(d*e+f)*	* +	abc*+de*f+	prioridade (*) > prioridade (+) apillar *
a+b*c+(d*e+f)*g	* +	abc*+de*f+g	operando sai directamente
final		abc*+de*f+g*+	desapillar todo o que haxa apillado

♦ **3ª aplicación das pillas**: pilla de execución do ordenador, para chamadas a procedimentos e funcións: a pilla garda en cada chamada todas as variábeis locais e o contador de programa (dentro dos rexistos físicos do sistema).

En calquer linguaxen de programación que permita a recursión existe unha pilla para gardar a información asociada a cada chamada. A información gardada chama-se ***rexisto de activación*** ou ***marco da pilla***. De cote a pilla de execución medra desde memória alta cara abaixo; en muitos sistemas non se controla o erro fatal derivado do rebasamento da pilla.

Exemplo de recursión inapropriada: "recursión pola cola" (porque a chamada recursiva atopa-se na derradeira liña do subprograma):

```
tipolista = ^ nodo ;
nodo = tupla
              elemento : tipoelemento ;
              seguinte : tipolista ;
fintupla ;

procedimento visualizar_lista ( L ) ;
     se non lista_valeira (L)
     entón
              escreber ( L ^ . elemento ) ;
              visualizar ( L ^ . seguinte ) ;
     finse ;
finprocedimento ;
```

Note-se o que fai para unha lista [a b c d e f g h]

```
escrebe a
cópia [b c d e f g h]
escrebe b
```

```
cópia [c d e f g h]
escrebe c
cópia [d e f g h]
escrebe d
cópia [e f g h]
...
```

Obviamente aqui este uso da recursión non está xustificado (¡imaxine-se o proceso de unha lista de 30.000 elementos!), seria muito mais razoábel facer un código non recursivo:

```
procedimento visualizar_lista ( L ) ;
     mentres non lista_valeira (L)
     entón
               escreber ( L ^ . elemento ) ;
               L ← L ^ . seguinte ;
     finse ;
finprocedimento ;
```

A eliminación da recursión por cola é mui sinxela, pero non todos os compiladores a realizan.

Os códigos recursivos soen ser mais lentos que os non recursivos pero muito mais autoexplicativos e sinxelos de entender.

Toda recursión se pode eliminar co uso de unha pilla.

2.1.2.- Colas.

Ao igual que as pillas, as colas son listas, só que nas colas a inserción e a eliminación levan-se a cabo por extremos opostos e non polo mesmo.

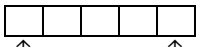
insertar_cola (x , C) → [cola C] → eliminar_cola (C)

Como as pillas, calquer implementación de listas é legal para as colas. Como as pillas, tanto as implementacións con listas enlazadas como con arranxos dan rápidos tempos de execución O(1) para toda operación elemental. A implementación con listas enlazadas é mui sinxela. Estudamos a implantación con vectores.

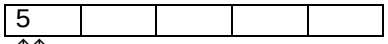
A estrutura de dados cola compón-se de catro partes (tupla de catro campos):

```
vector_de_cola : arranxo [ 1 .. tam_max_de_cola ]
                    { vector circular }
                    { tam_max_de_cola constante }
tamaño_de_cola : contador de posicións ocupadas
                    { 0 .. tam_max_de_cola }
frente_de_cola : índice de por onde se extrai elemento
                    { 1 .. tam_max_de_cola }
final_de_cola : índice de por onde se inserta elemento
                    { 1 .. tam_max_de_cola }
```

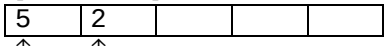
```
{-----------------------------------------------------------------}
procedimento criar_cola ( C ) ;
   C . tamaño_de_cola ← 0 ;
   C . frente_de_cola ← 1 ;
   C . final_de_cola ← tam_max_de_cola ;
finprocedimento ;
```



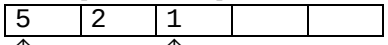
```
↑         ↑
frente  final
[extr.] [insertar]
{-----------------------------------------------------------------}
procedimento incrementar_indice ( x ) ; { ¡privado! }
   se x = tam_max_de_cola
   entón x ← 1
   senón x ← x + 1
   finse ;
finprocedimento ;
{-----------------------------------------------------------------}
función cola_valeira ( C ) ;
   devolver ( C . tamaño_de_cola = 0 ) ;
finfunción ;
{-----------------------------------------------------------------}
procedimento insertar_cola ( x , C ) ;
   se C . tamaño_de_cola = tam_max_de_cola
   entón erro “cola chea”
   senón
        C . tamaño_de_cola ← C . tamaño_de_cola + 1 ;
        incrementar_indice ( C . final_de_cola ) ;
        C . vector_de_cola [ C . final_de_cola ] ← x ;
   finse ;
finprocedimento ;
```




```
↑↑
frente
[extrair]
final
[insertar]
```

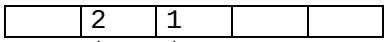



```
↑   ↑
frente
[extrair]
    final
    [insertar]
```




```
↑         ↑
frente  final
[extr.] [insertar]
{-----------------------------------------------------------------}
función extrair_cola ( C ) : tipoelemento ;
   se cola_valeira ( C )
   entón erro “cola valeira”
   senón
        C . tamaño_de_cola ← C . tamaño_de_cola - 1 ;
        devolver ( C . vector_de_cola [ C . frente_de_cola ] ) ;
```

```
        incrementar_indice ( C . frente_de_cola ) ;
    finse ;
finfunción ;
```




```
frente  final
[extr.] [insertar]
{-------------------------------------------------------------}
```

Do mesmo xeito que nas pillas, aqui nas colas temos que apontar duas cousas: primeiro, que se pasa sempre a cola por referência (con var) a todos os subprogramas por motivos de eficiência (evitar duplicados) incluso cando non se modifica a mesma. Segundo, os tempos das operacións elementais ao igual que nas pillas son O(1).

2.1.3.- Listas.

A lista é unha estrutura de dados lineal definida por un vector $[a_1|a_2|...|a_n]$ e un parámetro n, tamaño (que debemos manter obrigatoriamente ao menos cando se implementan listas con arranxos e non mediante enlaces de ponteiros). Tamén por definición temos que posición de un elemento a_i é o índice i. O sucesor do elemento a_i é o elemento a_{i+1} $\forall$ $1 \leq i < n$ e o antecesor do elemento a_i é o elemento a_{i-1} $\forall$ $1 < i \leq n$

As operacións elementais son:

```
visualizar_lista ( L )
anular ( L )
      { inicializar }
buscar ( x , L )
      { devolve a posición da primeira ocorrência de x en L }
insertar ( x , L , p )
      { decidir se vai antes da posición p ou na posición p }
eliminar ( x , L )
      { elimina a primeira ocorrência de x en L }
buscar_k_esimo ( L , p )
      { devolve elemento que se atopa na posición p }
```

Os tempos para a implementación con vectores son:

visualizar: O(n) tempo lineal. É o mellor que se pode lograr.
buscar: O(n) tempo lineal. É o mellor que se pode lograr.
buscar_k_esimo: O(1) tempo constante.
insertar 1ª posición (pior caso): correr todos elementos unha posición. Pior caso é O(n). Caso promédio: necesário correr metade da lista: segue-se requerindo un tempo lineal O(n).
eliminar 1ª posición (pior caso): correr todos elementos unha posición. Pior caso é O(n). Caso promédio: necesário correr metade da lista: segue-se requerindo un tempo lineal O(n).
construción de lista con n insercións sucesivas: tempo cuadrático $O(n^2)$

Debido ao custo lineal da inserción e eliminación chegamos à necesidade do non armacenamento contiguo dos elementos que conforman a lista: precisamos empregar unha lista enlazada.

Nodo de lista enlazada: [dado|ponteiro a seguinte nodo→]

Cando non hai nodo seguinte isto indica-se asignando-lle o valor “null” (nulo) ao ponteiro: “non aponta a nengunha parte”.

A inserción de un nodo requere a reserva de memória prévia (con “new” ou “alloc”) e anàlogamente a eliminación de un nodo debe culminar-se coa liberación da memória antes retida (mediante “dispose” ou “release”).

Os tempos para a implementación con ponteiros son:

visualizar: O(n) tempo lineal. É o mellor que se pode lograr. Superior ao tempo do visualizar de arranxos estáticos nunha constante.
buscar: O(n) tempo lineal. É o mellor que se pode lograr. Superior ao tempo do buscar de arranxos estáticos nunha constante.
buscar_k_esimo: non eficiente como en vectores, aqui é O(i), é dicer O(n) no pior caso.
eliminar: é O(i) sendo i a posición do elemento a eliminar. No pior caso é O(n).

Ao implementar a inserción e a eliminación de elementos en listas implementadas mediante enlaces de vectores observará-se que en ambas operacións se tên que tratar como casos especiais o proceso (inserción/eliminación) do primeiro elemento da lista. Para evitar isto e ter un código homoxéneo que se encarregue igualmente de todos os elementos aporta-se a posibilidade de traballar con listas con cabeceira. Son listas iguais que as xá definidas coa única diferenza de que o primeiro nodo non garda un dado válido (consideramos só como posicións válidas para gardar dados da segunda à derradeira). A morfoloxia do TDA lista (a declaración type) é idéntica, o que câmbia é a implementación interna da sintaxe do TDA (o código de cada procedimento e función que realiza unha operación elemental).

Exemplo: a lista [1|2|3] representaria-se nunha lista con cabeceira como [?|ptr]→[1|ptr]→[2|ptr]→[3|nil] onde ? é un dado que non nos importa nen sabemos.

```
{-----------------------------------------------------------------}
type
      t_ptr = ^ t_nodo_cabeceira ;
      t_nodo_cabeceira =      record
                                        dado : elemento ;
                                        seguinte : t_ptr ;
                              end ;
      t_lista_cabeceira = t_ptr ;
      t_posicion = t_ptr ;
{-----------------------------------------------------------------}
procedure inicializa_lista_cabeceira ( var L : t_lista_cabeceira ) ;
begin
      new ( L ) ;
      L ^ . seguinte := nil ;
end ;
{-----------------------------------------------------------------}
function lista_cabeceira_valeira ( var L : t_lista_cabeceira ) : boolean ;
begin
      lista_cabeceira_valeira := ( L ^ . seguinte = nil ) ;
end ;
{-----------------------------------------------------------------}
function e_derradeiro ( posicion : t_posicion ) : boolean ;
begin
```

```
      e_derradeiro := ( posicion ^ . seguinte = nil ) ;
end ;
{-----------------------------------------------------------------}
function buscar ( x : elemento ; var L : t_lista_cabeceira ) : posicion ; {
devolve ponteiro à primeira ocorrência ou nil se non existe }
var
      posicion : t_posicion ;
begin
      posicion := L ^ . seguinte ;
      while posicion <> nil
      and then posicion ^ . dado <> x
      do
            posicion := posicion ^ . seguinte ;
      buscar := posicion ;
end ; { so funciona en linguaxens que avalien por cortocircuíto (avaliación
preguiceira) }
{-----------------------------------------------------------------}
function buscar ( x : elemento ; var L : t_lista_cabeceira ) : posicion ; {
devolve ponteiro à primeira ocorrência ou nil se non existe }
var
      atopado : boolean ;
      posicion : t_posicion ;
begin
      atopado := false ;
      posicion := L ^ . seguinte ;
      while p <> nil
      and not atopado
      do
            if posicion ^ . dado = x
            then atopado := true
            else posicion := posicion ^ . seguinte ;
      buscar := posicion ;
end ;
{-----------------------------------------------------------------}
function buscar_anterior ( x : elemento ; var L : t_lista_cabeceira ) :
t_posición ;
{ privada ; devolve a posición anterior do nodo que contén a x, ou o
derradeiro nodo da lista se x non está en L }
var
      atopado : boolean ;
      posición : t_posición ;
begin
      atopado := false ;
      posición := L ;
      while posicion ^ . seguinte <> nil
      and not atopado
      do
            if posicion ^ . seguinte ^ . elemento = x
            then atopado := true
            else posicion := posicion ^ . seguinte ;
      busca_anterior := posicion ;
end ;
{-----------------------------------------------------------------}
procedure eliminar ( x : elemento ; var L : t_lista_cabeceira ) ;
var
      posición , tmp : t_posición ;
begin
      posicion := busca_anterior ( x , L ) ;
      if e_derradeiro ( posicion )
      then erro “non atopado”
```

```
      else begin
            tmp := posicion ^ . seguinte ;
            posicion ^ . seguinte := tmp ^ . seguinte ;
            dispose ( tmp ) ;
      end ;
end ;
{-----------------------------------------------------------------}
procedure insertar ( x : elemento ; var L : t_lista_cabeceira ; p : posicion {
insertar despois de posicion } ) ;
var
      tmp : t_posicion ;
begin
      new ( tmp ) ;
      if tmp = nil
      then erro “erro fatal: non hai espazo”
      else begin
                  tmp ^ . dado := x ;
                  tmp ^ . seguinte := posicion ^ . seguinte ;
                  posicion ^ . seguinte := tmp ;
      end ;
end ;
{-----------------------------------------------------------------}
```

Todas as primitivas son O(1) salvo busca e busca_anterior debido a que *en todos os casos se realiza un número fixo de instruccións, independentemente do grande que sexa a lista*. As operacións busca e busca_anterior son no pior caso O(n) polo percorrido da lista (que o elemento non estexa na lista ou que sexa o derradeiro). En promédio o tempo de execución é O(n) xá que hai que percorrer a metade da lista. Como eliminar chama a busca_anterior, eliminar tamén é O(n).

Se non se quer “pagar” o O(n) de eliminar, emprégan-se listas dobremente enlazadas. Os inconvenientes neste caso son que se duplica o custo de manexo de ponteiros e que se precisa mais memória. A vantaxen é precisamente o tempo, que pasa a ser O(1).

Outro tipo de listas que nos quedan por comentar son as listas circulares. Segundo a aplicación poden mellorar os tempos de acceso.

Tamén se poden implementar as listas enlazadas mediante arranxos:

```
tipo
      Pnodo = inteiro ;
      Nodo =      tupla
                        elemento : tipoelemento ;
                        seguinte : Pnodo ;
                  fintupla ;
      Lista = Pnodo ;
      Posicion = Pnodo ;
variábel
      EspazoMemoria = tabua [ 0 .. TamEspazo ] de Nodo ;
```

As cabeceiras das primitivas son as mesmas e o 0 funciona como nil. Hai que ter en conta que o dado contido na posición 0 non é válido. Tentar acceder a EspazoMemoria [0] debemo-lo ter controlado como nas listas enlazadas mediante ponteiros temos sempre cuidado de non tentar acceder a nil ^. Hai que manter unha lista de posicións ocupadas e unha lista de posicións libres.

lista de máximo cinco posicións inicialmente valeira

ocupadas = 0
libres = 1

	elemento	seguinte
0	2019820982829	0
1	0393993093034	2
2	8948899304949	3
3	0392093893839	4
4	83939009	5
5		0

insertar (66)

	elemento	seguinte
0	2019820982	0
1	66	0
2	0930348948899	3
3	3049490392093	4
4	8938398393900	5
5	9	0

ocupadas = 1
libres = 2

insertar 23, 91, 55,

	elemento	seguinte
0	2019820982	0
1	66	2
2	23	3
3	91	4
4	55	0
5	3983939009	0

ocupadas = 1
libres = 5

eliminar 23

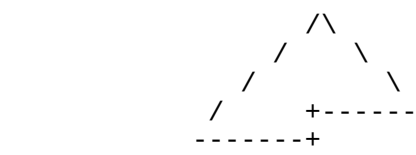
	elemento	seguinte
0	2019820982	0
1	66	3
2	23	0
3	91	4
4	55	0
5	3983939009	2

ocupadas = 1
libres = 5

2.2.- Montículos.

Un montículo é unha árbore binária que satisfai duas propriedades: unha concerniente à sua figura e outra à orden dos seus elementos:

-é unha árbore binária que está completa ou case completa, coas follas do derradeiro nível tan à esquerda como sexa posíbel



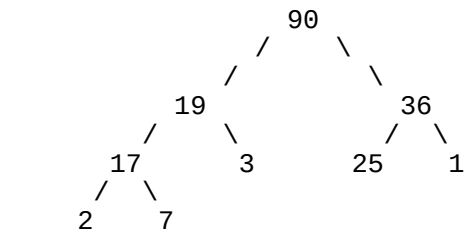
-para todo nodo, cumpre-se que o seu valor é maior ou igual que o valor de calquer dos seus fillos.

Empregan-se para:
-operacións de ordenación -melloran a ordenación por mistura (Mergesort), complexidade n $\log_2$ n-.
-às veces para o manexo de colas de prioridade.

Representación de un montículo nun vector lineal

Hai que lembrar que:
-o nodo k armacena-se na posición k correspondente do arranxo
-o fillo esquerdo do nodo k armacena-se na posición 2^k
-o fillo direito do nodo k armacena-se na posición 2^{k+1}

exemplo: dado o montículo...



...a representación nun vector lineal é:

90 19 36 17 3 25 1 2 7

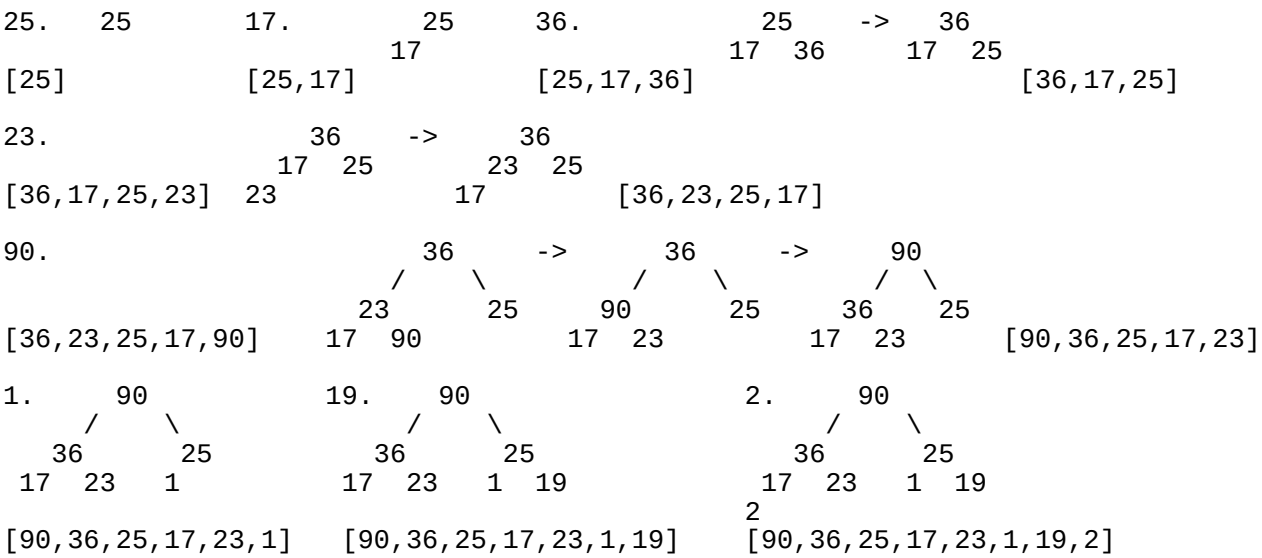
(é un percorrido en anchura do montículo)

```
anchura ( a )
     limpacola ( cola )
     if a <> nil
     then metecola ( a , cola )
     while not colavaleira ( cola ) do begin
          sacacola ( a )
          procesar ( a )
          if a ^ . esq <> nil
          then metecola ( a ^ . esq , cola )
          if a ^ . dir <> nil
          then metecola ( a ^ . dir , cola ) ;
```

Inserción en montículo

A operación de inserción consta de dous pasos:
-inserta-se o elemento na primeira posición disponíbelno vector.
-verifica-se se o seu valor é maior que o do seu pai. Se é asi intercâmbian-se entre si estes dous valores. Se non é asi, o algoritmo chegou à sua fin. Este paso aplica-se de forma recursiva de abaixo cara arriba.

exemplo: criar un montículo dada a secuência 25 17 36 23 90 1 19 2



Ordenación por montículos ou heapsort

É o mais eficiente dos métodos de ordenación que usan árbores.
Consta de dous pasos: construir un montículo; e sacar os elementos do mesmo, pola raiz e segundo un algoritmo que veremos.

Algoritmo de transformación de un array desordenado nun montículo

Supomos un arranxo con elementos de 1 a n, dados desordenados de tipo inteiro. Pasando-lle o arranxo, o procedimento troca os valores de sítio de modo que o array resultante represente un montículo.

```
monticulo ( a : vector ; n : inteiros )
variábeis
         i , k , aux : inteiros
         b : booleana
comezo
          i inicializa-se a 1, o comezo do vector
          mentres i <= n facer
               k <- i
               b <- certo
               mentres k > 1 e b = certo facer
                    b <- falso
                    se a [ k ] > a [ inteiro k/2 ] entón
                         trocar ( a [ k ] , a [ inteiro k/2 ] )
                         k <- inteiro k/2
                         b <- certo
                    finse
               finmentres
```

```
            i <- i+1

          finmentres
fin
```

Implementación dinámica

Para engadir un elemento fai-se un percorrido en anchura e inserta-se no primeiro nodo que teña un fillo a nil. Poderá-se engadir un campo ponteiro ao pai do nodo e poderá-se usar recursividade.

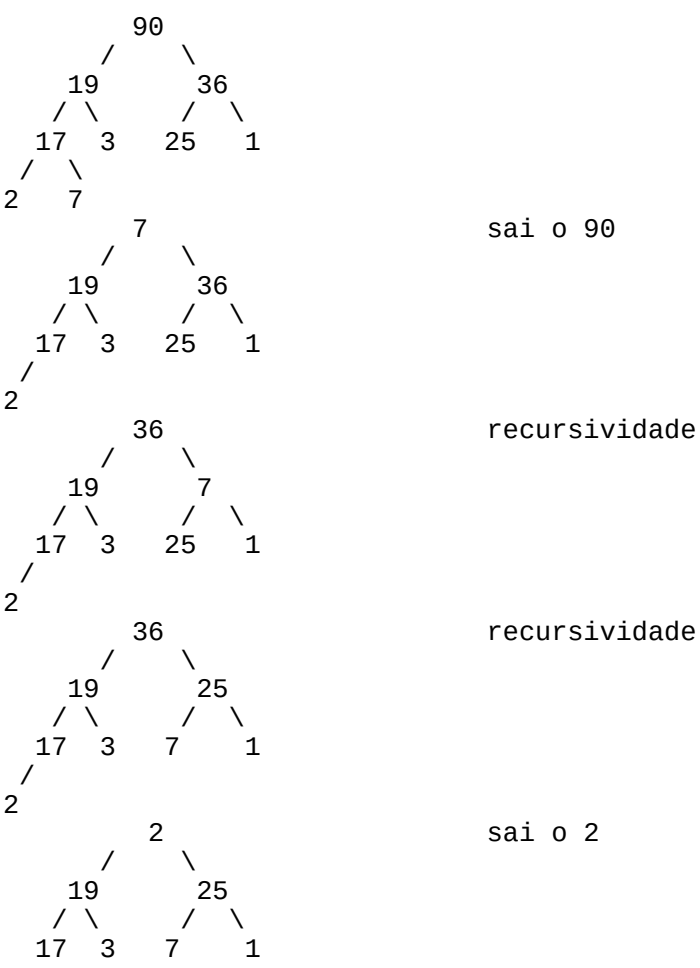
Eliminación de un montículo

O proceso para obtener os elementos ordenados efectuará-se da seguinte maneira:

1º) reempráza-se a raiz co elemento que ocupa a derradeira posición do montículo.

2º) verifica-se se o valor da raiz é maior que os seus dous fillos (maior que o maior dos seus fillos). Se é asi, remata o algoritmo. Se non é asi, troca-se a raiz co maior. Aplica-se de forma recursiva desde arriba cara abaixo.

exemplo: [90,19,36,17,3,25,1,2,7]



```
        25                    recursividade
       /    \
     19      2
    / \     /  \
   17  3   7    1
```

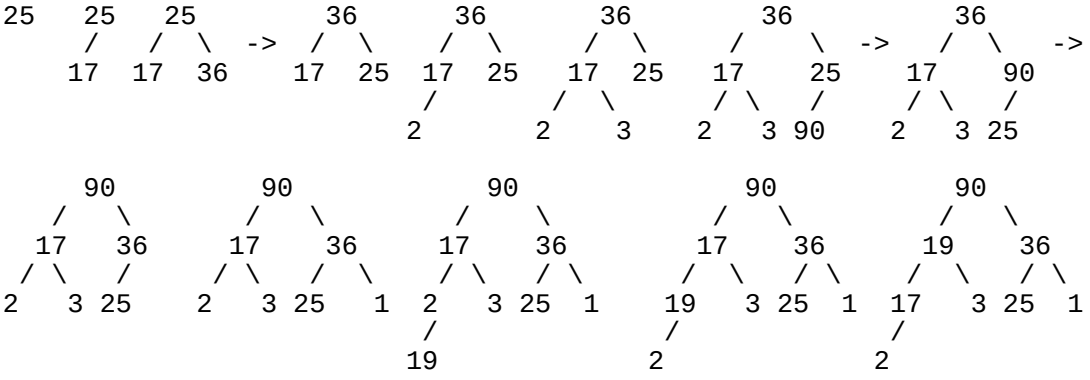
...etc

Os montículos son estruturas que serven para representar colas de prioridade.

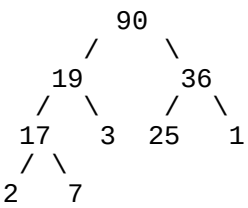
```
eliminar ( a : vector ; n : inteiro ) ;
variábeis
          i , aux , z , d , k , ad , maior : inteiros
comezo
          i <- n
          mentres i > 1 facer
               aux <- a[i]
               a[i] <- a[1]
               z <- 2
               d <- 3
               k <- 1
               mentres z < i facer
                    maior <- a[z]
                    ad <- z
                    se maior < a[d] e d <> i entón
                         maior <- a[d]
                         ad <- d
                    se aux < maior entón
                         a[k] <- a[ad]
                    k <- ad
                    z <- k¨2
                    d <- z+1
               finmentres
               a[k] <- aux
               i <- i-1
          finmentres
fin
```

EXERCICIOS RESOLTOS.-

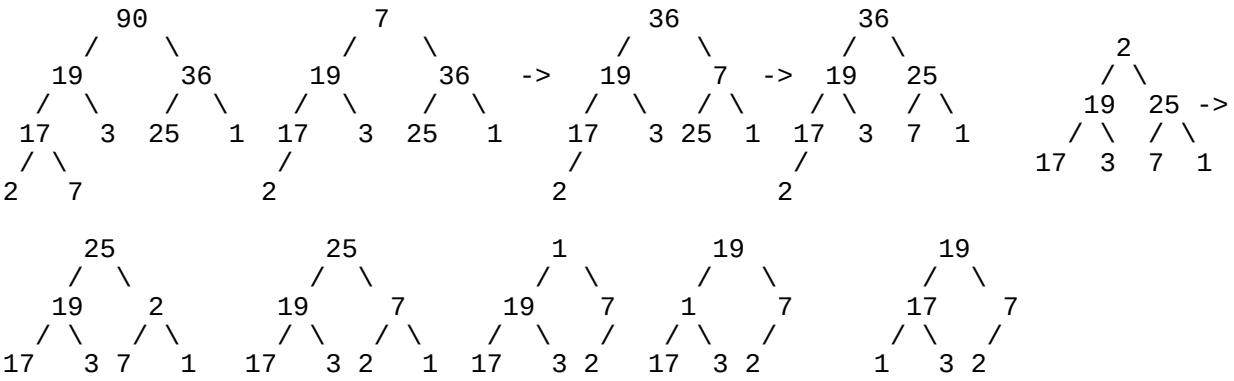
exercicio: cria montículo 25, 17, 36, 2, 3, 90, 1, 19, 7



Tema 2 Páxina 16



exercicio: quitar três elementos do seguinte montículo.



2.3.- Árbores.

A busca nas listas leva un tempo O(n) xá que se percorren de xeito secuencial os n elementos da lista no pior dos casos.

Veremos como as árbores binárias de busca [ABB] son un TDA que ten como promédio de tempo de execución O(log n) para a maioria das operacións.

Unha árbore en xeral define-se recursivamente como un conxunto de nodos que pode estar valeiro ou non. Se non o está, está formado por un nodo distinguido chamado raiz, que pode ter un ou mais sucesores (a diferenza das listas, nas que o máximo para cada elemento é un sucesor) chamados “fillos” e sendo cada un de estes eventuais sucesores à sua vez unha árbore.

A lonxitude de un camiño no seio de un grafo é o número de aristas do mesmo.

Para calquer nodo n, a sua **profundidade** é a lonxitude do camiño único entre a raiz e o nodo n. A raiz ten profundidade cero.

Para calquer nodo n, a sua altura é a lonxitude do camiño mais longo de n a unha folla. A altura das follas é cero. A altura da árbore é a altura da raiz.

Chamamos árbore binária [AB] a unha árbore na que cada nodo pode ter cero, un ou dous fillos.

Profundidade média AB = O(√n)

Profundidade média ABB = O(log n)

```
tipo
      TPtrÁrboreBin = ^ Tnodo ;
      Tnodo =     tupla
                        elemento : Telemento ;
                        esq, dir : TPtrÁrboreBin ;
                  fintupla ;
      AB = TPtrÁrboreBin ;
```

Hai vários xeitos de definir as árbores binárias de busca, pero nós imolo facer asi:

Def.- ABB = AB (árbore onde todo nodo ten 0, 1 ou 2 fillos)
∧ "as chaves son inteiros"
∧ "non hai chaves duplicadas"
∧ "para todo nodo os valores da sua subárbore esquerda son inferiores à chave do nodo; e os valores da sua subárbore direita son superiores à chave do nodo"

Note-se como a morfoloxia do TDA ABB é exactamente igual à morfoloxia do TDA AB (o que cambiará será a implementación da sintaxe, o código interno das operacións):

```
{-------------------------------------------------------------}
type
      TPtrArboreBinBusq = ^ Tnodo ;
      Tnodo =     record
                        elemento : integer ;
                        esq, dir : TPtrArboreBinBusq ;
                  end ;
      ABB = TPtrArboreBinBusq ;
{-------------------------------------------------------------}
procedure criar_ABB ( var A : ABB ) ;
begin
      A := nil ;
end ;
{-------------------------------------------------------------}
function buscar ( x : integer ; A : ABB ) : TPtrArboreBinBusq ;
begin
      if A = nil
      then buscar := nil
      else
            if x = A ^ . elemento
            then buscar := A
            else
                  if x < A ^ . elemento
                  then buscar := buscar ( x , A ^ . esq )
                  else buscar := buscar ( x , A ^ . dir ) ;
end ;
{ Con este procedimento recursivo por cola precisa-se unha pilla de tamaño
O(log n) o cal non é excesivo }
{-------------------------------------------------------------}
procedure buscar_min ( A : ABB ) : TPtrArboreBinBusq ;
{ recursivo }
begin
      if A = nil
      then buscar_min := nil
      else
            if A ^ . esq = nil
            then buscar_min := A
```

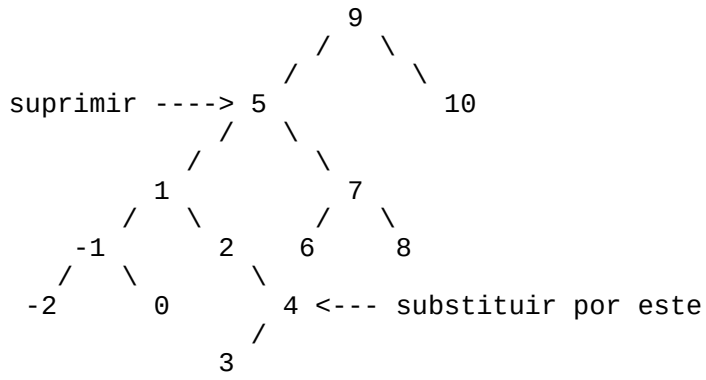
```
            else buscar_min := buscar_min ( A ^ . esq ) ;
end ;
{-----------------------------------------------------------------}
procedure buscar_max ( A : ABB ) : TPtrArboreBinBusq ;
{ non recursivo }
begin
      if A <> nil
      then
            while A ^ . dir <> nil do
                  A := A ^ . dir ;
      buscar_max := A ;
end ;
{-----------------------------------------------------------------}
procedure insertar_ABB ( x : integer ; A : ABB ) ;
begin
      if A = nil
      then begin
                  new ( A ) ;
                                    { if A = nil
                                    then erro ( “memória esgotada” )
                                    else begin }
                  A ^ . elemento := x ;
                  A ^ . esq := nil ;
                  A ^ . dir := nil ;
                                    { end ; }
      end
      else
            if x < A ^ . elemento
            then insertar_ABB ( x , A ^ . esq )
            else
                  if x > A ^ . elemento
                  then insertar_ABB ( x , A ^ . dir ) ;
      { ¡notar que non se insertan duplicados! }
end ;
{-----------------------------------------------------------------}
```

Se se quer incluir a admisión de duplicados o mellor é engadir un campo frecuência en cada nodo indicando o número de ocorrências. Non se pode meter nodos distintos con chaves repetidas xá que isto penaliza a profundidade da árbore en exceso.

En canto à supresión de nodos en árbores binárias de busca: primeiro hai que atopar o nodo que queremos eliminar. Despois consiste en eliminar un nodo da árbore binária de busca sen violar os princípios que precisamente o definen. Hai os seguintes casos, en función de se o número de fillos é 0, 1 ou 2:

-se o elemento a borrar é terminal ou folla, simplesmente suprímese.
-se o elemento a borrar ten un so fillo, substitui-se por ese fillo.
-se o elemento a borrar ten dous fillos, substitui-se polo nodo que se atopa mais à direita da subárbore esquerda.

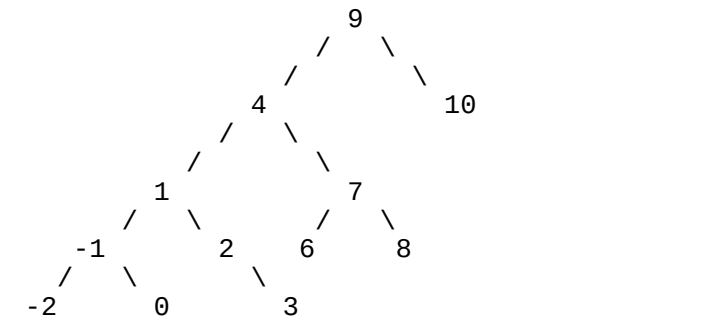
Exemplo:



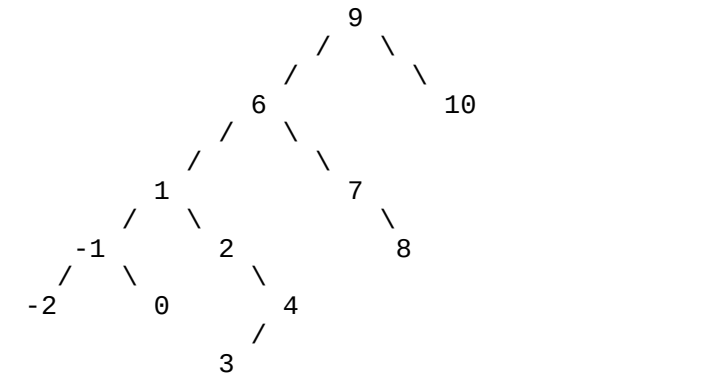
Do que se trata é de localizar o elemento maior da subárbore esquerda do que imos borrar. Tamén poderíamos localizar o elemento menor da subárbore direita.

É dicer facemos o movimento 1 à esquerda, todo à direita (ou na outra opción faríamos 1 à direita, todo à esquerda) e logo estabelece-se unha "ponte" do pai ao fillo do elemento que vimos de borrar.

Coa primeira solución quedaria:



Coa segunda solución quedaria:




```
{-------------------------------------------------------------}
procedure eliminar_en_ABB ( var A : ABB ; x : integer ) ;
var
      tmp : TPtrArboreBinBusq ;
begin
      if A = nil
      then erro ( “chave non atopada na árbore” )
```

```
    else
          if x < A ^ . elemento
          then eliminar_en_ABB ( A ^ . esq , x )
          else
               if x > A ^ . elemento
               then eliminar_en_ABB ( A ^ . dir , x )
               else { x = A ^ . elemento }
                    if A ^ . esq = nil then begin
                          tmp := A ;
                          A := A ^ . dir ; {sexa nil ou non}
                          dispose ( tmp ) ;
                    end
                    else
                          if A ^ . dir = nil then begin
                                tmp := A ;
                                A := A ^ . esq ; {sexa nil ou non}
                          dispose ( tmp )
                          end
                          else begin {ten dous fillos}
                             tmp := buscar_min ( A ^ . dir ) ;
                             A ^ . elemento := tmp ^ . elemento ;
                             eliminar_en_ABB
                                   ( A ^ . elemento , A ^ . dir ) ;
                          end ;
end ;
{-------------------------------------------------------------}
```

Unha alternativa constitui-a a eliminación preguiceira que consiste en manter o campo frecuência en cada nodo e ao eliminar, dar-lle a este campo o valor cero. Isto supón ter na árbore nodos fictícios ou falsos. Nas ABB impón unha penalización mui pequena en tempo porque ainda nunha relación 50%-50% de nodos “reais” e “falsos”, a profundidade da árbore só aumenta en 1. A vantaxen fica na “reentrada” ou “reactivación” de chaves na árbore, xá que se evita a sobrecarga de asignar unha nova celda.

Intuitivamente esperamos que todas as operacións da sección anterior, excepto criar_ABB, **tarden un tempo O(log n), porque en O(1) -tempo constante- descende-se un nível na árbore, operando asi sobre unha árbore que ten aproximadamente a metade de lonxitude.** Efectivamente, o tempo de execución de estas operacións é O(p) sendo p a profundidade do nodo que contén a chave buscada.

Un dos problemas fundamentais cos que se enfrenta o TDA ABB é o do desequilíbrio derivado da inserción ordenada de chaves na estrutura.

Se se insertan chaves ascendentemente, a ABB derivará nunha lista descedente da raiz cara à direita (todo nodo agás o derradeiro ten só o fillo direito). Se se insertan chaves descendentemente, a ABB derivará nunha lista descendente desde a raiz cara a esquerda (todo nodo agás o derradeiro ten so o fillo esquerdo).

Asi que estamos nun proceso puramente secuencial de unha lista e polo tanto o tempo de acceso no pior caso é de O(n).

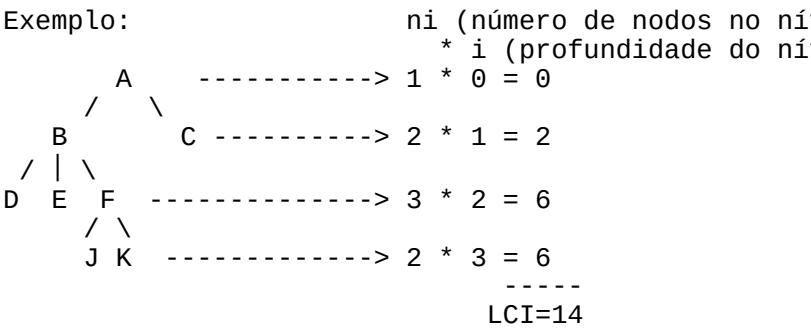
Existen estruturas alternativas que permiten que a busca sexa no pior caso o desexado O(log n):

-as árbores AVL (balanceadas) mantên o equilíbrio
-as árbores despregadas (splay trees) proceden à reestruturación

Análise do caso promédio

Demostración de que a profundidade média de todo nodo nunha ABB é O(log n), supondo que todas as árbores son equiprobábeis.

Def.- LCI Lonxitude de Camiño Interno = Σ profundidade nodos



profundidade média = LCI/n = 14/8 = 1.75

Calcularemos a LCI média de todas as ABB posíbeis.

Sexa D(n) a lonxitude média de camiño interno de unha árbore de n nodos.

Como unha árbore de un so nodo non ten aristas, LCI média = D(1)=0.

Unha árbore de n nodos consta de:

1 raiz (profundidade 0 → non intervén no cálculo)
subárbore esquerda de i nodos
subárbore direita de n-i-1 nodos

D(i) é a LCI média da subárbore esquerda respeito à sua raiz. Na ABB principal, todos estes nodos da esquerda son 1 unidade mais profundos.

D(n-i-1) é a LCI média da subárbore direita respeito à sua raiz. Na ABB principal, todos estes nodos da direita son 1 unidade mais profundos.

Conclusión:

$$D(n) = D(i) + i\cdot 1 + D(n-i-1) + (n-i-1)\cdot 1$$
$$D(n) = D(i) + D(n-i-1) + n-1$$

Se todos os tamaños de subárbores son equiprobábeis (o cal é certo para ABBs pero non para ABs) entón o valor promédio de D(i) e D(n-i-1) é

$$(1/n)\cdot \sum_{j=0}^{n-1} D(j)$$

é dicer que:

$$D(n)=(2/n)\cdot \sum_{j=0}^{n-1} D(j) + n-1 \qquad [*]$$

[*] multiplicando x n

$$n\cdot D(n)=2\cdot\sum_{j=0}^{n-1} D(j) + n^2-n \qquad [1]$$

se a expresión [*] é válida para n tamén o é para n-1:

$$D(n-1)=\left(\frac{2}{n-1}\right)\cdot\sum_{j=0}^{n-2} D(j) + (n-2)$$

multiplicamos esta anterior por n-1

$$(n-1)\cdot D(n-1)=2\cdot\sum_{j=0}^{n-2} D(j) + (n-1-1)(n-1)$$

$$(n-1)\cdot D(n-1)=2\cdot\sum_{j=0}^{n-2} D(j) + (n-1)^2 -(n-1) \qquad [2]$$

agora restamos [1]-[2]

$$n\cdot D(n)=2\cdot\sum_{j=0}^{n-1} D(j) + n^2-n \qquad [1]$$

$$(n-1)\cdot D(n-1)=2\cdot\sum_{j=0}^{n-2} D(j) + (n-1)^2 -(n-1) \qquad [2]$$

$$n\cdot D(n)-(n-1)\cdot D(n-1)=2\cdot D(n-1)+(n-1)\cdot n-(n-1)^2 +(n-1)$$
$$n\cdot D(n)-(n-1)\cdot D(n-1)=2\cdot D(n-1)+(n-1)(n+1)-(n-1)^2$$
$$n\cdot D(n)-(n-1)\cdot D(n-1)=2\cdot D(n-1)+n^2-1-n^2+2n-1$$
$$n\cdot D(n)-(n-1)\cdot D(n-1)=2\cdot D(n-1)+2n-2$$
$$n\cdot D(n)=(n-1)\cdot D(n-1)+2\cdot D(n-1)+2n-2$$
$$n\cdot D(n)=(n+1)\cdot D(n-1)+2n$$

dividimos entre n(n+1)

$$D(n)/(n+1) = D(n-1)/n + 2/(n+1)$$

desenrolamos e tachamos o subliñado na suma das ecuacións:

$$D(n)/(n+1) = \underline{D(n-1)/n} + 2/(n+1)$$

$$\underline{D(n-1)/n} = \underline{D(n-2)/(n-1)} + 2/n$$

$$\underline{D(n-2)/(n-1)} = D(n-3)/(n-2) + 2/(n-1)$$

...

$$\underline{D(2)/3} = D(1)/2 + 2/3$$

$$D(n)/(n+1) = D(1)/2 + 2\cdot\sum_{i=3}^{n+1} 1/i$$

como D(1)=0

entón:

$$D(n)/(n+1) = 2 \cdot \sum_{i=3}^{n+1} 1/i$$

como $\sum_{i=3}^{n+1} 1/i \approx \log_e (n+1) + \gamma - 3/2$

onde γ = cte Euler $\approx 0'577$
entón:

$D(n)/(n+1) = O(\log n)$

entón

$D(n) = O(n \cdot \log n)$ c.q.d.

"lonxitude média de camiño interno"

"profundidade esperada para un nodo é O(log n)"

***O tempo** de execución de **calquer operación** é **O(log n)**?*

Podemos afirmá-lo se todos os ABB son equiprobábeis e o único que desfavorece esta hipóte-se é a primitiva de eliminación. Unha solución é a eliminación preguiceira. Poderian usar-se árbores equilibradas AVL; outra solución tamén son as árbores despregadas, mediante o recurso da reestruturación.

PERCORRIDOS EN ÁRBORES

```
{-----------------------------------------------------------------}
procedimento visualizar_árbore ( A ) { inorde }
{é O(n) porque se percorre cada elemento unha vez}
      se A ≠ nil
      entón
            visualizar_árbore ( A ^ . esq )
            escreber ( A ^ . elemento )
            visualizar_árbore ( A ^ . dir )
      finse
finprocedimento
{-----------------------------------------------------------------}
procedimento altura ( A ) { postorde }
{é O(n) porque se percorre cada elemento unha vez}
      se A = nil
      entón altura ← -1 { 1 nodo → altura = 0 }
      senón altura ← 1 + máximo(altura(A^.esq),altura(A^.dir))
      finse
finprocedimento
{-----------------------------------------------------------------}
procedimento anchura ( A )
{é O(n) porque se percorre cada elemento unha vez}
{ usa unha cola de ponteiros }
      se A ≠ nil
      entón
            limpa_cola ( C )
            mete_cola ( C , A )
```

```
        mentres non cola_valeira ( C )
        facer
              saca_cola ( C , x )
              escrebe ( x )
              se A ^ . esq <> nil
              entón mete_cola ( C , A ^ . esq )
              se A ^ . dir <> nil
              entón mete_cola ( C , A ^ . dir )
{-------------------------------------------------------------}
```

2.4.- O TDA tábua de dispersión (hash table)

À implantación de TDDs de cote se lle chama simplesmente dispersión (hashing). O obxectivo da TDD é realizar insercións, eliminacións e buscas nun tempo promédio constante (tempo O(1)). Con unha TDD só se poden efectuar un subconxunto das operacións permitidas nas árbores binárias de busca anteriormente vistas (por exemplo a TDD non ten os percorridos en orden). Non se conta con "buscar mínimo", "buscar máximo" nen "visualizar tábua ordenada" en tempo lineal.

A estrutura ideal para unha TDD é un vector de tamaño TamD: vector [0 .. TamD -1] de chaves coas que operar (insertar, eliminar, buscar). Considera-se o tamaño da tábua TamD como parte da TDA, e non algunha variábel global flotante. Acompañando a este vector temos unha función de dispersión, que é a que permite calcular a partir da chave a posición onde se atopa esa chave na estrutura, podendo acceder asi à información asociada a esa chave:

F(): chave $\longrightarrow$ índice $\in$(0..TamD-1)

Para isto define-se o tipo tipo_índice = 0 .. TamD -1

Pretendemos que a función de dispersión sexa sinxela, pero principalmente e por en riba de todo queremos que dadas duas chaves distintas caisquera, obteñamos da función de dispersión posicións distintas. En outras palabras, é precisa unha distribución homoxénea das chaves, co obxectivo de minimizar o número de colisións (circunstância na que a duas ou mais chaves distintas lles corresponde unha mesma posición de acordo coa función de dispersión).

Problemas que se plantexan: Cal función de dispersión se vai usar? Cal é a resposta às colisións? Que podemos dicer acerca do tamaño TamD?

Dado que as chaves que manexamos son inteiras, a primeira función que se pode usar é chave MOD TamD agás no caso de que a chave teña algunhas propriedades indesexábeis. Por exemplo, se o tamaño da tábua é dez e todas as chaves acaban en cero, é obvio que a función de dispersión estándar é unha mala opción.

A solución -por isto e por outras cousas que veremos mais adiante- é coller sempre TamD un número primo. Se TamD é primo e as chaves son aleatórias e equiprobábeis, terá-se con MOD unha distribución homoxénea.

PRIMEIRA FUNCIÓN DE DISPERSIÓN

En xeral as chaves serán cadeias de caracteres, polo cal a función de dispersión de cote fai a operación "MOD TamD" sobre o sumatório dos códigos ASCII dos caracteres da chave:

F() = **(**Σascii(chave[i])**)** MOD TamD

O que en pseudocódigo é:

```
función dispersión ( chave , tamaño_chave ) : tipo_índice
     valor ← ascii ( chave [ 1 ] ) ;
     para j ← 1 até tamaño_chave facer
          valor ← valor + ascii ( chave [ j ] )
     finpara ;
     devolver ( valor mod TamD )
finfunción
```

Sen embargo esta función non distribui ben as chaves se a tábua é grande. Por exemplo:

```
constante
   TamD = 10.007 { número primo }
tipo
   TipoChave = arranxo [ 1 .. 8 ] caracteres
```

Como o código ascii devolve un valor ao sumo 127 ⇒ como máximo a función de dispersión devolve posicións entre cero e 127·8 = 1.016 asi que a distribución é malísima. A groso modo estamos situando todas as chaves na porción inicial do 10% quedando o restante 90% da tábua totalmente valeiro.

SEGUNDA FUNCIÓN DE DISPERSIÓN

Vexamos agora outra función de dispersión (non mui boa). Supomos que o tamaño da chave é como mínimo 3. Se é menor que 3, preenche-se con espazos en branco. Note-se que só examina os três primeiros caracteres da chave, pero para un tamaño 10.007 como o do exemplo anterior e se os caracteres son aleatórios ten-se unha distribución razoàbelmente homoxénea.

```
función dispersión ( chave , tamaño_libre ) : tipo_índice
{ tamaño da chave ≥ 3 }
   devolver (
                  (
                      ascii ( chave [ 1 ] )
                    + ascii ( chave [ 2 ] ) * k
                    + ascii ( chave [ 3 ] ) * k²
                  )
                       mod TamD
   )
finfunción
{ k = número de letras do alfabeto + 1 polo espazo en branco }
```

Desafortunadamente as linguaxens naturais non son aleatórias, é mais probábel en galego dar coa série "abe" que coa "wxz". Se tomamos como exemplo o inglês, ainda que hai 26^3 = 17.576 combinacións posíbeis de três caracteres (ignorando brancos), mirando un dicionário chega-se a conclusión de que o

número de combinacións diferentes reais é 2.851. Isto implica que só o 28% da tábua se aproveitaria para a dispersión.

Como conclusión: esta función é sinxela de calcular pero non é axeitada para tábuas relativamente grandes.

TERCEIRA FUNCIÓN DE DISPERSIÓN

```
función dispersión ( chave , tamaño_chave ) : tipo_índice
   valor ← ascii ( chave [ 1 ] ) ;
   para j ← 2 até tamaño_chave facer
        valor ← (
                        valor * 32 + ascii ( chave[ j ] )
                  )
                        mod TamD
   finpara ;
   devolver ( valor )
finfunción
```

Esta é unha función de dispersión correcta. Intervên todos os caracteres da chave e en xeral pode-se agardar unha boa distribución. Na función calcula-se unha función polinómica de 32 basada na regra de Horner. A posición da chave é:

$$chave[n] + 32 \cdot chave[n-1] + 32^2 \cdot chave[n-2] + \ldots$$

Usamos 32 porque "·32" non é en realidade un produto senón mais ben un desprazamento de cinco bits. O malo é que para evitar o desbordamento en cada iteración é preciso empregar a operación MOD, co cal estamos penalizando o tempo de execución (e se TamD ademais fose potência de dous, a función de dispersión seria claramente ineficiente). Se a linguaxen de programación permite o desbordamento, este é un caso no que está ben empregado ——a operación MOD non se poria, estaria implícita como "valor MOD (tamaño do tipo de dado de valor)"——.

Esta función non é necesariamente a mellor, pero é extremadamente sinxela e rápida se se permite desbordamento. Se as chaves son mui longas, a función de dispersión tardará muito no cálculo, asi que a solución típica é usar só uns cantos caracteres de entre todos os que ten a chave.

Estratéxias para a resolución de colisións

- DISPERSIÓN ABERTA ou ENCADEIAMENTO SEPARADO
- DISPERSIÓN PECHADA ou ENDEREZAMENTO ABERTO
  - exploración lineal
  - exploración cuadrática
  - dispersión dobre

• DISPERSIÓN ABERTA ou ENCADEIAMENTO SEPARADO

Consiste en que a TDD é un arranxo de listas con cabeceira. Cada lista contén todas as chaves que a través da función de dispersión obtiveron a mesma posición:

```
const
        TamD = 10007 ;
type
        ptr_lista = ^ nodo ;
        nodo_lista = record
                                        chave : tipo_chave ;
                                        seg : ptr_lista ;
                        end ;
        tipo_índice = 0 .. TamD -1 ;
        tábua_de_dispersión = arranxo [ tipo_índice ]
                                        de ptr_lista ;

exemplo:
constantes
        TamD = 11 ;
tipos
        igual que arriba ;
        as chaves son letras ;
función de dispersión ( chave )
        devolve ascii ( chave ) mod TamD ;

alfabeto inglês e dispersión correspondente a cada chave posíbel:
A B C D E F G H I J  K L M N O P Q R S T U  V W X Y Z
1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4

chaves: E X E M P L O D E B U  S Q U  E D A R A P I D A
        5 2 5 2 5 1 4 4 5 2 10 8 6 10 5 4 1 7 1 5 9 4 1
   __
 0|__→ ? →
 1|__→ ? → L → A → A → A →
 2|__→ ? → X → M → B →
 3|__→ ? →
 4|__→ ? → O → D → D → D →
 5|__→ ? → E → E → P → E → E → P →
 6|__→ ? → Q →
 7|__→ ? → R →
 8|__→ ? → S →
 9|__→ ? → I →
```

```
10|__→ ? → U → U →
```

? indica que o contido do nodo cabeceira é indiferente
indica nil

Outra solución é insertar ordenadamente por orden alfabética en cada lista e non repetir elementos:

```
   __
 0|__→ ? →
 1|__→ ? → A → L →
 2|__→ ? → B → M → X →
 3|__→ ? →
 4|__→ ? → D → O →
 5|__→ ? → E → P →
 6|__→ ? → Q →
 7|__→ ? → R →
 8|__→ ? → S →
 9|__→ ? → I →
10|__→ ? → U →
```

e tamén se pode insertar ao primeiro ou último elemento en chegar.

A inicialización da tábua consiste en converter...

```
    __
  0|__→ ? → ?
  1|__→ ? → ?
  2|__→ ? → ?
... __
n-1|__→ ? → ?
```

o primeiro ? indica que o contido do nodo cabeceira é indiferente.
o segundo ? indica que o valor do campo “seguinte” do nodo cabeceira ten un valor indeterminado.

...no arranxo de listas con cabeceiras, inicializadas:

```
    __
  0|__→ ? →
  1|__→ ? →
  2|__→ ? →
... __
n-1|__→ ? →
```

? indica que o contido do nodo cabeceira é indiferente
indica nil

```
{-------------------------------------------------------------}
procedure inicializa_tábua ( var D : tábua_de_dispersión ) ;
var
   i : integer
begin
   for i := TamD -1 downto 0 do begin
        new ( D [ i ] ) ;
        if D [ i ] = nil
        then erro_fatal ( ‘memória esgotada’ )
        else D [ i ] ^ . seg := nil
```

```
   end { for }
end ;
```

A chamada buscar (chave , tábua) devolverá un apontador à celda que contén chave. Note-se que se pasa a tábua por referência (con "var") ainda que non se vai modificar. Isto é por se acaso o compilador non é *listo* abondo para dar-se conta de que non se vai modificar a tábua, xá que faria unha cópia da mesma.

```
{-------------------------------------------------------------}
function buscar
( chave : tipo_chave ; var D : tábua_de_dispersión ) : ptrlista ;
var
   ptr , seg : ptr_lista ;
   atopado : boolean ;
begin
   atopado := false ;
   ptr := D [ dispersión ( chave ) ] ;
   seg := ptr ^ . seg ;
   while ( seg <> nil )
   and then ( not atopado )
   do begin
        if seg ^ . elemento = chave
        then atopado := true
        else seg := seg ^ . seg
   end ;
   buscar := seg ;
end ;
```

En canto à inserción, non admitimos duplicados asi que se o elemento a insertar xá existe na lista, non facemos nada. Se non existe, colocamo-lo à frente da lista (o elemento último en chegar queda xusto "à direita" do nodo cabeceira na representación que vimos antes).

```
{-----------------------------------------------------}
procedure insertar
( chave : tipo_chave ; var D : tábua_de_dispersión ) ;
var
   pos , lista , celda_nova : ptr_lista ;
begin
   pos := buscar ( chave , D ) ;
   if pos = nil then { chave non atopada }
   begin
        new ( celda_nova ) ;
        if celda_nova = nil
        then erro_fatal ( 'memória esgotada' )
        else begin
             lista := D [ dispersión ( chave ) ] ;
             celda_nova ^ . seg := lista ^ . seg ;
             celda_nova ^ . elemento := chave ;
             lista ^ . seg := celda_nova ;
        end
   end
end ;
```

Aspectos pendentes de discusión:

-non usar cabeceira? Se a TDD non inclui eliminacións, probàbelmente é mellor non usar cabeceiras, xá que o seu uso non ofereceria nengunha simplificación e malgastaria unha cantidade considerábel de espazo.

-resolución de colisións: mediante ABB ou outra TDD. Xustifica-se unicamente se o número de colisións se converte nun problema realmente grave, pero en princípio isto é impensábel porque se supón que con unha tábua grande e unha función de dispersión axeitada as listas se manterán mui curtas.

-eliminación de chave:

```
{-----------------------------------------}
procedure eliminación
( chave : tipo_chave ; var D : tábua_de_dispersión ) ;
var
   pos , seg : ptr_lista ;
   atopado : boolean ;
begin
   pos := D [ dispersión ( chave ) ] ;
   seg := pos ^ . seg ;
   atopado := seg ^ . elemento = chave ;
   while not atopado
   and then seg ^ . seg <> nil
   do begin
        pos := pos ^ . nil ;
        seg := seg ^ . nil ;
        atopado := seg ^ . elemento = chave
   end ;
   if atopado then begin
        pos ^ . seg := pos ^ . seg ^ . seg ;
        dispose ( seg ) ;
   end ;
end ;
```

Definimos λ factor de carga de unha tábua de dispersión como o cociente:

$$\lambda = \frac{\text{número de chaves na tábua}}{\text{tamaño da tábua*}}$$

*o que denotamos até agora como TamD

exemplo: no exemplo EXEMPLODEBUSQ... λ = 14/11 (coa segunda solución, sen chaves duplicadas en cada lista)

A lonxitude média de cada unha das listas da TDD é λ

O esforzo necesário para efectuar unha busca é o tempo constante que fai falla para avaliar a función de dispersión mais o tempo necesário para percorrer a lista:

$$T_{\text{búsqueda}} = \underset{\uparrow}{O(1)} + \underset{\uparrow}{O(\lambda)}$$

$T_{\text{f.dispersión}}$ $T_{\text{percorrido lista}}$

No pior caso (busca infructuosa porque o elemento non existe ou está na derradeira posición), o número promédio de enlaces por percorrer é λ (excluíndo o enlace final a nil):

$$T_{\text{búsqueda}} = \lambda \quad \text{(pior caso)}$$

Unha busca exitosa requere que se visiten case 1+(λ/2) enlaces, posto que se garante que un enlace sexa percorrido (pois a busca é exitosa) e tamén agardamos ir até a metade da lista para atopar o elemento correspondente.

$T_{búsqueda} = 1 + \lambda/2$ (busca exitosa)

Esta análise demostra que o tamaño da tábua non é realmente importante, pero o factor de carga si o é.

A regra xeral de unha dispersión aberta é facer o tamaño da tábua case tan grande como o número de elementos agardados (en outras palabras $\lambda=1$). Tamén é boa ideia conservar primo o tamaño da tábua para asegurar unha boa distribución.

Unha solución que se pode aportar coa ideia de mellorar a tábua é insertar as chaves ordenadamente en cada lista (asi o tempo promédio para percorrer a lista divide-se entre dous); o que pasa é que isto supón bastante esforzo e bastante pouco resultado.

Propriedade da dispersión aberta: a dispersión aberta divide o número de comparacións da busca secuencial por un factor de tamaño TamD* utilizando un extra de TamD* nodos (estamos usando mais espazo pero reducindo o número de comparacións de unha busca secuencial).

*o tamaño da tábua de dispersión

• DISPERSIÓN PECHADA ou ENDEREZAMENTO ABERTO

A dispersión aberta ten a desvantaxen de que requere apontadores. Isto tende a facer un pouco lento o algoritmo, debido ao tempo necesário para asignar celdas novas, e tamén requere en esência a implantación de unha segunda estrutura de dados.

A dispersión pechada é unha alternativa para a resolución das colisións con listas enlazadas. Nun sistema de dispersión pechada, ante o caso de colisión, buscan-se celdas alternativas até atopar unha celda valeira. Isto é:

busca-se nas celdas $d_0(x)$, $d_1(x)$, $d_2(x)$,...

onde $d_i(x)$ = dispersión(x) MOD TamD + f(i) MOD TamD

↑ ↑

función de dispersión estratéxia de resolución de colisións

f(0)=0

Como todos os dados se meten na tábua, precisa-se unha tábua mais grande para a dispersión pechada que para a aberta.

Na dispersión pechada, en xeral, o factor de carga debe estar por debaixo de $\lambda=0.5$ ($\lambda<0.5$ ou $\lambda<2/3$ de acordo con distintos autores)

dispersión pechada { exploración lineal
exploración cuadrática
dispersión dobre

EXPLORACIÓN LINEAL

f (estratéxia de resolución de colisións) é unha función lineal de i, polo regular f(i)=i

Isto equival a percorrer as celdas en secuência circular na procura de unha celda valeira.

Exemplo:

TamD = 19

estratéxia de resolución de colisións f(i)=i

alfabeto inglês e dispersión correspondente a cada chave posíbel:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 0 1 2 3 4 5 6 7

chaves: E X E M P L O D E B U S Q U E D A
5 5 5 13 16 12 15 4 5 2 2 0 17 2 5 4 1

O paso de cada coluna à seguinte no seguinte gráfico corresponde à resposta a unha colisión:

posición
| estado1
| | estado2 ...
↓ ↓ ↓

0						S	S	S	S
1									A
2					B	B	B	B	B
3						U	U	U	U
4				D	D	D	D	D	D
5	E	E	E	E	E	E	E	E	E
6		X	X	X	X	X	X	X	X
7			E	E	E	E	E	E	E
8					E	E	E	E	E
9							U	U	U
10								E	E
11									D
12				L	L	L	L	L	L
13				M	M	M	M	M	M
14									
15				O	O	O	O	O	O
16				P	P	P	P	P	P
17						Q	Q	Q	Q
18									

estado final, filas 0 a 13, 15 a 17: efeito de formación de bloco (“agrupamento imán”, “clustering”, ou “agrupamento primário”)

$\lambda = 17/19 = 0.89 >>> 0.5$

número de intentos (buscas sen éxito e para inserción)

$$\approx \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

número médio de intentos (buscas con éxito)

$$\approx \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

respeitivamente número de intentos (sen, con éxito):

$\lambda=2/3$ $\left\lceil \frac{5}{2} \right\rfloor$

$\lambda=0.5$ $\left\lceil \frac{2.5}{1.5} \right\rfloor$

$\lambda=0.75$ $\left\lceil \frac{8.5}{2.5} \right\rfloor$

$\lambda=0.9$ $\left\lceil \frac{50}{5.5} \right\rfloor$

asi que ben vemos a importância de manter o factor de carga baixo 0.5

```
const
        TamD = { un número primo } ;
type
        tipo_índice = 0 .. TamD -1 ;
        tábua_de_dispersión = array [ tipo_índice ]
                                    of tipo_chave ;

{-------------------------------------------------------------}
procedure inicializa_tábua ( var D : tábua_de_dispersión ) ;
var
   i : integer
begin
   for i := 0 to TamD -1 do
        D [ i ] := max ;
end ;
{ max é intuitivamente ∞, un valor especial demasiado grande
que indica que a posición está valeira }
```

solución alternativa:

```
type
   clase_entrada = ( válido , valeiro , eliminado ) ;
   chave = record
                dado : tipo_chave ;
```

```
                controlo : clase_entrada ;
            end ;
   tipo_índice = 0 .. TamD -1 ;
   tábua_de_dispersión = array [ tipo_índice ]
                               of tipo_chave ;

{-----------------------------------------------------------}
procedure inicializa_tábua ( var D : tábua_de_dispersión ) ;
var
   i : integer
begin
   for i := 0 to TamD -1 do
        D [ i ] . controlo := valeiro ;
end ;

{-----------------------------------------------------------}
function insertar
( chave : tipo_chave ; var D : tábua_de_dispersión ) : integer ;
begin
   x := dispersion ( chave ) ;
   while D [ x ] <> max
   { mentres non estexa valeira cada posición }
   do x := ( x + 1 ) mod TamD ;
   { aqui incluir test: }
   D [ x ] := chave ;
   insertar := x ;
end ;
```

a dispersión pechada requere eliminación preguiceira (pondo a posición a "eliminada")

EXPLORACIÓN CUADRÁTICA

A exploración cuadrática é un método de resolución de colisións que elimina o problema do agrupamento primário (ou clustering ou formación de blocos) que padece a exploración lineal. A función de colisións é cuadrática. A elección común é $f(i)=i^2$

Isto implica que se a posición obtida para a chave está ocupada, procurará-se insertar en posición+1, se tamén está ocupada tentará-se en posición+4, se tamén está ocupada en posición+9... asi sucesivamente sumando os cuadrados perfeitos inferiores a TamD

TEOREMA

Se se usa a exploración cuadrática e o tamaño da tábua TamD é primo, entón sempre se pode insertar unha nova chave se a tábua está, ao menos, médio valeira.

/! se $\lambda > 0.5$ a inserción pode fallar

/! TamD = 16 $\rightarrow$ 1, 4, 9 (cuadrados perfeitos) posicións alternativas únicas.

Aqui tamén se necesita a eliminación preguiceira (en xeral para calquer estratéxia de dispersión pechada)

Exemplo:

estratéxia de resolución de colisións $f(i)=i^2$

TamD = 19 → posicións alternativas: 1, 4, 9, 16

alfabeto inglês e dispersión correspondente a cada chave posíbel:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	0	1	2	3	4	5	6	7

chaves:

E	X	E	M	P	L	O	D	E	B	U	S	Q	U	E	D	A
5	5	5	13	16	12	15	4	5	2	2	0	17	2	5	4	1

O paso de cada coluna à seguinte no seguinte gráfico corresponde à resposta a unha colisión:

posición
↓ estado1
↓ ↓ estado2 ...
↓ ↓ ↓

0					S	S	S
1							A
2				B	B	B	B
3					U	U	U
4			D	D	D	D	D
5	E	E	E	E	E	E	E
6		X	X	X	X	X	X
7							
8							D
9			E	E	E	E	E
10							
11						U	U
12			L	L	L	L	L
13			M	M	M	M	M
14				E	E	E	E
15			O	O	O	O	O
16			P	P	P	P	P
17					Q	Q	Q
18							E

```
Exemplodebusqueda 5 valeira
eXemplodebusqueda 5 ocupada
                  5+1 = 6 valeira
exEmplodebusqueda 5 ocupada
                  5+1 = 6 ocupada
                  5+4 = 9 valeira
...
exemplodEbusqueda 5 ocupada
                  5+1 = 6 ocupada
                  5+4 = 9 ocupada
                  5+9 = 14 valeira
...
exemplodebUsqueda 2 ocupada
                  2+1 = 3 valeira
```

```
...
exemplodebusqUeda 2 ocupada
                  2+1 = 3 ocupada
                  2+4 = 6 ocupada
                  2+9 = 11 valeira
exemplodebusquEda 2 ocupada
                  2+1 = 3 ocupada
                  2+4 = 6 ocupada
                  2+9 = 11 ocupada
                  2+16 = 18 valeira
exemplodebusqueDa 4 ocupada
                  4+1 = 5 ocupada
                  4+4 = 8 valeira
```

DISPERSIÓN DOBRE

A elección común de resposta a colisións é $f(i)=h_2(x)$ é dicer unha segunda función de dispersión.

/! hai que decatar-se de que $h_2(x)$ nunca pode dar cero porque se entraria nun bucle infinito (probaria-se indefinidamente na mesma posición).

$h_2(x) = R - (x \bmod R)$ con R primo menor que TamD

Exemplo:

TamD = 19

$h_2(x) = 7 - (x \bmod 7)$

alfabeto inglês e dispersión correspondente a cada chave posíbel:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	0	1	2	3	4	5	6	7

chaves:	E	X	E	M	P	L	O	D	E	B	U	S	Q	U	E	D	A
h1 →	5	5	5	13	16	12	15	4	5	2	2	0	17	2	5	4	1
h2 →	2	2	2	1	5	2	6	3	2	5	5	7	4	5	2	3	6

Sen embargo na prática o tipo de estratéxia que se utiliza é a exponencial cuadrática.

2.5.- Colas de prioridade.

```
Insertar(x,M)                        Eliminar(M,x): prioridade máx.
──────────────→    cola de prioridade M ──────────────→
```

Emprega-se en SO, simulación de sistema, algoritmos ávidos.

Implementación con listas enlazadas:
- -inserción en frente: O(1)
- -eliminación implica percorrido: O(n)

Implementación con listas ordenadas:
- -inserción en frente implica percorrido: O(n)
- -eliminación: O(1)

Cal das duas escoller? Tendo en conta que nºeliminacións ≤ nºinsercións -- non se poden sacar mais dos que se meteron--, escollemos a primeira solución.

Unha terceira solución é mediante árbores binárias de busca, O(log n) en promédio.

Elimina-se sempre o maior, polo que se xeran árbores con certo desequilíbrio, mais pesados pola rama esquerda. Ainda asi o promédio para insercións e eliminacións é de O(log n)

Unha cuarta solución consiste en usar montículos, o cal permite operacións en O(log n) para o pior caso.

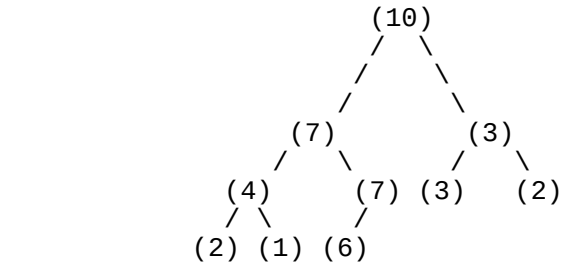
Simplificamos as ABB xá que non son necesárias todas as operacións que nos permite facer a ABB para implementar as colas de prioridade.

Un montículo ten duas características:
- -sobre a estrutura: é unha AB case completa
- -sobre a orden: a maior prioridade está na raiz, todo nodo é maior que o(s) seu(s) fillo(s), se o(s) ten.

Nunha AB case completa, os nodos de altura h atópan-se nas posicións 2^h e 2^{h+1}-1 (en implementación con vectores).

h = ⌊log n⌋ = O(log n) (altura da árbore)

$$v\left[\ \underbrace{\ \ }_{2^h}\ \underbrace{\ \ }_{2^{h+1}-1}\ \right]$$



T = (10,7,3,4,7,3,2,2,1,6)

∀ i, fillo esquerdo en 2i, direito en 2i+1 e pai en i div 2 (coas limitacións normais de princípio e fin do arranxo).

BuscarMáximo: acceso à raiz, O(1)

Insertar: filtrado ascendente (“aboiar”)
- -pior caso: O(log n)
- -caso médio: O(1)

Inserta-se na primeira posición libre (despois da última) e reaxusta-se seguindo a condición de que todo nodo é maior que o(s) seu(s) fillo(s).

Eliminar: filtrado descendente (“afundir”)
- -pior caso: O(log n)
- -caso médio: O(log n)

Saca-se sempre a raiz. Pon-se o último elemento na raiz e reaxusta-se seguindo a condición de que todo nodo é maior que os seu(s) fillo(s).

AumentarPrioridade(x, A, M): filtrado ascendente.

DiminuirPrioridade(x, A, M): filtrado descendente.

Eliminar_C(x, M): eliminar un elemento calquer. Trata-se de aumentar-lle muito a prioridade para que pase à posición da raiz e despois eliminar normalmente.

ConstruirMontículo(M) (con n insercións)
-pior caso O(n log n)
-caso promédio O(n)

Aplicacións.-

Xá vimos que hai duas solucións ao problema da selección do k-ésimo maior:

1.- ler n valores ordenados e acceder ao k-ésimo. A ordenación realiza-se en $O(n^2)$
2.- ler k valores, ordená-los -ordenar k é $O(k^2)$-; insertar -insertar é $O(k\cdot(n-k))$-, procédese, até acabar de procesar os n elementos. Resulta $O(k^2+k(n-k))=O(n-k)$. Se $k=\lceil n/2 \rceil$ entón é $O(n^2)$, o pior caso.

Agora vemos unha terceira solución mediante montículos: ler n elementos, criar un montículo, eliminar k elementos, devolver o elemento da raiz:

construir O(n)
eliminar O(k log n)
total O(n + k log n)

se k=O(n/log n) entón total O(n+n)=O(2n)=O(n)
se k > n entón total O(k log n)

pior caso $k=\lceil n/2 \rceil$ entón $\theta(n \log n)$

Se n=k e se rexistan as saídas nun vector, está-se ordenando nun tempo O(n log n). Esta é a ordenación por montículos ou heapsort.

Consideremos C o conxunto dos k maiores (C un montículo onde a raiz é o menor elemento)
construir un montículo: O(k)
para n-k veces: O(1) se hai que incluir o elemento no montículo C
+O(log k) para eliminar o menor se inserta o novo elemento (modificación do montículo)

o total será O(k+(n-k)log k) = O(n log k)

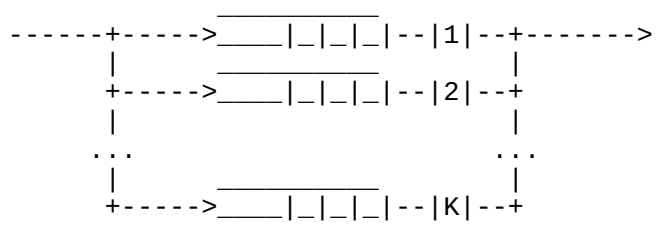
$k=\lceil n/2 \rceil \rightarrow \theta(n \log n)$

Simulación de eventos.-

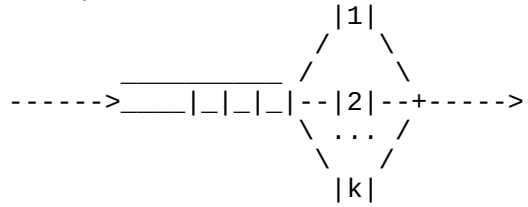
Sistema: un banco con clientes e k caixeiros no que se necesita saber o tempo de espera promédio e a lonxitude das colas que se forman. Mediante estes dados o banco tomará decisións relativas ao seu funcionamiento.

As solucións analíticas (mediante fórmulas) só son válidas en algúns casos.

O modelo:



equival a:



Un cliente está determinado por <hora de chegada, tempo de servicio>

(1) h=0 inicialmente
h <- h+1
flujo de entrada
até fluxo terminado e colas valeiras

(2) avanzar relóxio até evento seguinte
*saída: recollen estimacións
outro cliente? -> novo cliente -> cola de prioridade(h)
*chegada: caixeiro?
non -> cola
si -> novo cliente -> cola de prioridade(h)
+próxima chegada -> cola de prioridade(h)

Cola de prioridade(h)
até k saídas
+1: próxima chegada

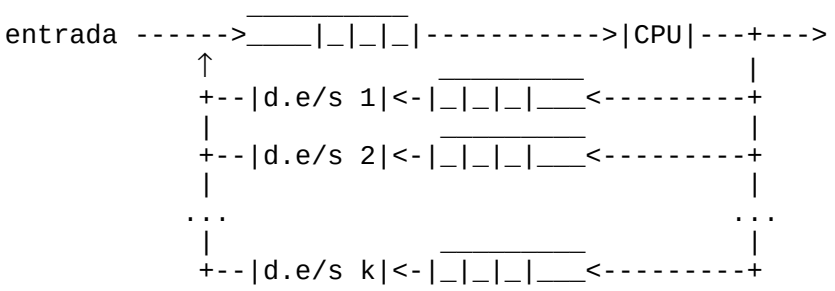
Se temos c clientes (2c eventos) e k caixeiros, o tempo da execución é $O(c \cdot \log(k+1))$

Proceso de cada evento $O(\log(k+1))$

tamaño do montículo k+1 (?)

Sistemas informativos (protocolo de comunicacións).

Imos ver un exemplo con procesos. Ciclo de vida dos procesos nun sistema operativo. Este estudo é necesário para que nun sistema de tempo real se poda ter a garantia de que unha operación se vai levar a cabo dentro de unha determinada marxe de tempos.



<fin do tema 2>