Sobre os vários esquemas. . .

Aplicação ao problema das matrizes.

O "dividir para conquistar" funciona do topo para a base ("top-down").

A "programação dinâmica" funciona da base para o topo ("bottom-up"). to topo para a base.

Neste caso o "dividir para conquistar" é menos eficiente uma vez que resolve sub-problemas repetidas vezes.

Mas o "dividir para conquistar" + "tabulação" têm uma eficiência semelhante à da "programação dinâmica". Diferem essencialmente na ordem da análise ("top-down" no primeiro caso, "bottom-up" no segundo).

O "Dicionário"
Um tipo de dados abstracto importante

Dicionário - definição

Definição Tipo abstracto de dados caracterizado pelas seguintes operações, sendo $\mathcal{A}$ um domínio (por exemplo, inteiros) e $\mathcal{D}$ o conjunto dos dicionários com elementos desse domínio.

insert : $\mathcal{A} \times \mathcal{D} \to \mathcal{D}$ $(x, d) \to d'$ sendo $d' = d \cup \{x\}$
delete : $\mathcal{A} \times \mathcal{D} \to \mathcal{D}$ $(x, d) \to d'$ sendo $d' = d \setminus \{x\}$
search : $\mathcal{A} \times \mathcal{D} \to \{V, F\}$ $(x, d) \to search(x, d)$ resultado: resposta a "$x \in d$?"
init : $\to \mathcal{D}$ resultado: novo dicionário (vazio)

Exercício 45 *Implemente a estrutura de dicionário em C++, definindo a classe correspondente com as funções públicas indicadas, etc. Não se preocupe com a eficiência, podendo usar um método de implementação simples (possivelmente ineficiente).* $\diamond$

Dicionário - aplicações

- Bases de dados.
- Listas de objectos com informação associada (listas de telefone, por exemplo).
- "Assembladores"

Dicionário
Breve explicação dos "assembladores"

Os nomes de um programa em linguagem "assembly" têm de ser definidos, devendo ocorrer uma e uma só vez no campo de "label", ex

```
LABEL     OPR   OPN
x         add   a1
          jz    x
y         ...
```

Os nomes e as suas propriedades (endereço, tipo de variável para as posições de memória. . .) são guardados num dicionário inicialmente vazio que designaremos por *dicionário de identificadores*.

1. Na primeira passagem o assemblador só "olha" para o campo de LABEL:
   Seja x um nome encontrado no campo de LABEL. Procura-se x no dicionário de identificadores. Se já lá está há erro "Identificador duplamente definido". Se ainda não está é inserido juntamente com as suas propriedades.

2. Na segunda passagem o assemblador "olha" para os campos de OPR e OPNN. Seja y um nome referido nesse campo. Procura-se y no dicionário de identificadores. Se não se encontra lá, há erro "Nome não definido". Caso contrário vão-se obter do dicionário as propriedades de y.

Dicionário - variantes

Existem diversas variantes do tipo de dados abstracto dicionário, em particular

- A operação de "delete" pode ou não ser incluída pois em algumas aplicações não é utilizada.

- Os objectos no dicionário podem formar um conjunto (nunca há repetições) ou um multiconjunto (pode haver repetições).

- Podem ser incluidas outras operações relativas a conjuntos como união e intersecção. Como caso particular pode acontecer que só se trate de conjuntos disjuntos.

Dicionário
Algumas implementações que estudaremos

- Vector não ordenado.
- Vector ordenado.
- Árvore binária de pesquisa.
- Árvore binária equilibrada.
- Árvore 2-3 e árvore B.
- "Tries"
- Métodos de "hashing" externo e interno.

Dicionário - Implementações elementares

Vector não ordenado Os elementos são mantidos num vector não nessessáriamente ordenado.
Vector ordenado Os elementos são mantidos num vector permanentemente ordenado.
Processamento por blocos Como "vector não ordenado" mas, de tantas em tantas inserções, o vector é ordenado.

	Vector não ordenado	Vector ordenado
Pesquisa	$O(n)$	$O(\log n)$
Insersão	$O(1)$ sem pesquisa $O(n)$ com pesquisa	$O(n)$
Eliminação	$O(n)$	$O(n)$

Nota Os tempos $O(n)$ na insersão e pesquisa no vector ordenado resultam da nessessidade de deslocar em média cerca de metade do vector para abrir ou eliminar ema posição.

Conjuntos e dicionários:
Outra implementação: Vectores de bits

Vectores de bits - "bit arrays":
Elementos do universo $\{0, 1, 2, \cdots, \mathtt{MAX} - 1\}$ com `MAX` não muito grande. Um conjunto $\{a, b, \cdots, c\}$ é representado colocando em 1 os bits números "a", "b", . . . , "c" – e em 0 os outros bits.
Por exemplo: $\{0, 1, 8, 10, 17, 18\}$

```
0  0  0  0  0  0  1  1  - a[0]=3 representa 0,1,...,7
7  6  5  4  3  2  1  0    (elementos 0 e 1)

0  0  0  0  0  1  0  1  - a[1]=5 representa 8,9,...,15
15 14 13 12 11 10 9  8    (elementos 8 e 10)

0  0  0  0  0  1  1  0  - a[2]=6 representa 16,17,...,23
23 22 21 20 19 18 17 16

......................
```


Elementos em bits ou bytes

Pode ser como indicado atrás: cada elemento num bit, ou (mais simples mas mais gastador de espaço) cada elemento num byte.
Exemplo: universo com 1 milhão de elementos. Cada conjunto gasta

1 Mbyte	se cada elemento num byte
125 kbyte	se cada elemento num bit

Implementações das operações

Pesquisa: $x \in A$?, já visto
Inserção: $A = A \cup \{x\}$, já visto
Reunião: $A \cup B$, "`a|b`"
Intersecção: $A \cap B$, "`a\&b`"
Diferença: $A \setminus B$, "`a & ~b`"

onde "`a`" e "`b`" são as representações de "`A`" e "`B`" respectivamente.

Estas representações são muito eficientes - inclusivamente correspondem em certos casos a um processamento "paralelo" - mas limitadas a conjuntos de um certo universo limitado de inteiros não negativos.

Exercício Implementar em linguagem `C++` uma classe cujos objectos sejam conjuntos e com implementação por "vector de bits".

Dicionário - Árvore binária

Definição (indutiva) Uma *árvore binária* ou é nula ou é da forma

(árvore binária, Conteúdo, árvore binária)

A primeira (segunda) árvore binária chama-se *sub-árvore esquerda* (respectivamente *sub-árvore direita*.

Definição Uma *árvore binária de pesquisa* é uma árvore binária em que há uma relação de ordem total nos conteúdos e que satisfaz:

- Para qualquer nó, todos os conteúdos da sub-árvore esquerda (se houver algum) são menores ou iguais ao conteúdo desse nó e todos os conteúdos da sub-árvore direita (se houver algum) são maiores que conteúdo desse nó.

<u>Nota</u> Em geral há muitas árvores binárias de pesquisa que podem representar o mesmo conjunto de conteúdos.

Implementação em C Com técnicas conhecidas usando "`struct`" e apontadores.

Inserção e pesquisa - ver programa seguinte.

Eliminação Consideremos o nó N que tem o valor a emiminar x.

1. Se a sub-árvore esquerda de N é nula: o nó N é substituído pela sua sub-árvore direita. O espaço ocupado por N é libertado.

2. Se a sub-árvore direita de N é nula: o nó N é substituído pela sua sub-árvore esquerda. O espaço ocupado por N é libertado.

3. Se nenhuma das sub-árvores de N é nula: Seja R o nó na sub-árvore direita de N com o valor mais pequeno. Este nó obtém-se descendo o mais possível para a esquerda no filho à direita de N. O valor que está em R passa para N e o nó R é eliminado. O espaço previamente ocupado por R é libertado.

Eficiência

Depende da estrutura da árvore a qual por sua vez depende da ordem pela qual os nós são inseridos.

- **Caso médio** Todas as operações (pesquisa, inserção e eliminação) são de ordem $O(\log n)$ onde n é o número de valores na árvore (ou um majorante desse número).
  Este caso ocorre quando a alture da árvore é $O(\log n)$.

- **Caso mais desfavorável** Todas as operações são de ordem $O(n)$.
  Este caso ocorre quando a alture da árvore é $O(n)$. Tal sucede, por exemplo, quando a árvore é criada pela inserção dos valores por ordem crescenta.

Há processos - ou tipos de árvores - de garantir que a altura é $O(\log n)$, *mesmo no pior caso*.

Um tipo de árvores que estudaremos são as *árvores-2-3*.

Dicionário, ordenação usando árvore binária

Modos de visita:

- Pré-ordem: Raiz, sub-árvore esquerda, sub-árvore direita (recursivamente).
- Em-ordem: Sub-árvore esquerda, raiz, sub-árvore direita (recursivamente).
- Pós-ordem: Sub-árvore esquerda, sub-árvore direita, raiz (recursivamente).

Modos de visita e expressões aritméticas: forma pré-fixa, em-fixa (e uso de parêntesis) e pósfixa.

Um modo de ordenar vectores:

1. Inserir os elementos a ordenar numa árvore binária.
2. Listar a árvore em-ordem.

Ordenação usando árvore binária - I

```
typedef struct no{
  int valor;
  struct no *esq;
  struct no *dir;
} NO, *ARVORE;

/* insert: Arv x N -> Arv        */
ARVORE insert(int,ARVORE);
void em_ordem(ARVORE);
/* Exemplo:                      */
int v[]={55,12,88,3,21,4};

main()
{
  ARVORE a=0;
  int i;
  for(i=0;i<6;i++)
    a=insert(v[i],a);
  em_ordem(a);
}
```

Ordenação usando árvore binária - II

```
ARVORE insert(int x,ARVORE a)
{
  if(a==0){
    ARVORE nova;
    nova=(ARVORE)malloc(sizeof(NO));
    nova->valor = x;
    nova->esq   = 0;
    nova->dir   = 0;
    return(nova);
  }
  if(x <= a->valor){
    a->esq = insert(x,a->esq);
    return(a);
  }
  a->dir = insert(x,a->dir);
  return(a);
}

void em_ordem(ARVORE a)
{
  if(a==0)
    return;
  em_ordem(a->esq);
  printf(" %d ",a->valor);
  em_ordem(a->dir);
}
```


Treino com apontadores

Exercícios

```
typedef struct no{
  char *pal;
  struct no *esq;
  struct no *dir;
} NO, *ARVORE;

char s[100];

main(){
  ARVORE a, b;
  /* 1. a aponta para novo nó, "batatas", apont. nulos */
  a=(ARVORE)malloc(sizeof(NO));
  a->pal="batatas";  /* a->pal = (*a).pal */
  a->esq=0;
  a->dir=0;
  (*a).dir=0;
  printf("%s\n",a->pal);

  /* 2. esq aponta para a, direito para o esquerdo */
  a->esq=a;
  a->dir=a->esq;
  printf("%s\n",(a->esq)->pal);
```

Treino com apontadores

```
  /* 3. b: novo nó, filho dir -> a, esquerdo nulo */
  b=(ARVORE)malloc(sizeof(NO));
  if(b==0)
    exit(1);
  scanf("%s",s);
  b->pal=(char *)malloc(1+strlen(s));
  strcpy(b->pal,s);
  b->dir=a;
  b->esq=0;
  printf("%s\n",b->pal);
  printf("%s\n",(b->dir)->pal);
  printf("%s\n",((b->dir)->esq)->pal);

  /* 4. Liberta a, liberta b */
  free(a);
  free(b);
  /* Errado: a=0; free(a); */
}

/*
  batatas
  batatas
* uvas
  uvas
  batatas
  batatas
*/
```


Dicionário - Árvores 2-3

Definição Uma *árvore-2-3* ou é nula ou tem um só nó ou satisfaz

- Todos os valores estão representados nas folhas.
- Todas as folhas estão à mesma altura.
- Cada nó interno ou tem 2 filhos (tipo 2) ou 3 filhos (tipo 3).
- Os nós do tipo 2 tem também um campo contendo o maior dos valores representados na sua subárvore esquerda.
- Os nós do tipo 3 tem também dois campos. Um deles contém o maior dos valores representados na sua subárvore esquerda. O outro contém o maior dos valores representados na sua subárvore central.

Os campos indicados são suficientes para decisão da sub-árvore a considerar durante a pesquisa.

As pequisas, inserções e eliminações são de ordem $O(\log n)$ mesmo no pior caso e *mantêm sempre uma árvore-2-3*.

Dicionário - Árvores-2-3

Uma árvore-2-3 que representa $\{3, 4, 6, 9, 11\}$

Inserção do valor 8

Dicionário - Árvores-2-3

Noção

Implementações

Dicionário - Árvores-2-3

Exercício 46 *Descreva o algoritmo de pesquisa de um valor numa árvore-2-3.* $\diamond$

Exercício 47 *Ilustre as sucessivas árvores-2-3 que resultam da inserção de 1, 2, 3, 4, 5, 6, 7, e 8 (por esta ordem).* $\diamond$

Exercício 48 *Descreva o algoritmo de inserção de um valor numa árvore-2-3.* $\diamond$

Generalização das árvores 2-3: as árvores B de grau n

- Todos os nós não terminais com possível excepção da raiz têm um número de filhos f que satisfaz: $\lfloor n/2 \rceil \leq f \leq n$.

As árvores 2-3 são árvores B de grau 3.

"Unions" em linguagem C

Por vezes há interesse em guardar no mesmo espaço de memória variáveis de diferentes tipos - uma de cada vez!

Sintaxe semelhante à do "`struct`":

```
union valor{
  int vali;
  float valf;
  char *nome;
} u;
```

O compilador reserva para a variável `u` um espaço suficientemente grande para conter qualquer dos tipos. Por exemplo, se `int`, `float` e um apontador ocupassem respectivamene 4, 8 e 4 bytes, o compilador reservaria para `u` um espaço de ≥ 8 bytes.

É da responsabilidade do programador usar este tipo de variáveis de forma consistente. Podia existir, por exemplo, uma variável `enum` que indicasse o tipo de variável na `union`.

“Unions” em linguagem C

Exemplo

```
if(tipo==INT)
  printf("%d\n",u);
else
if(tipo==FLOAT)
....
```

As `union` podem usar-se dentro das estruturas e “arrays” e vice-versa. Exemplo

```
struct{
  char * morada;
  int idade;
  int tipo_salario;
  union{
    float sal;
    BOOLEAN secreto;
    int comissão;
  } salario;
} empregado[MAX];
...
if(empreagado[i].tipo_salario==MONEY)
  printf("EMP %4d RECEBE %10f \n",
              i,empreagado[i].salario.sal);
...
```


“Unions”: aplicação às árvores-2-3 - I

```
struct n2{
  struct no * esq;
  int valor;
  struct no * dir;
};
struct n3{
  struct no * esq;
  int valor1;
  struct no * med;
  int valor2;
  struct no * dir;
};
struct folha{
  int valor;
  char *nome;
};
```

“Unions”: aplicação às árvores-2-3 - II

```
typedef struct no{
  int tipo;
  union{
    struct n2    NO2;
    struct n3    NO3;
    struct folha FOL;
  } node;
} NO, *ARV23;

main(){
  NO x;
  x.tipo=2;
  x.node.NO2.valor=99;
  x.node.NO2.esq=0;
  printf("%d\n",x.node.NO2.valor);
}
```


Dicionários - “tries”

Uma “trie” é uma árvore onde se podem representar elementos que são sequências da forma

$$(a_1, a_2, \cdots, a_k)$$

onde k pode variar de elemento para elemento e os a_i's são elementos de um alfabeto com n símbolos.

Exemplo Os elementos podem ser “strings”.
Exemplo Os elementos podem ser inteiros escritos na base 10. Cada a_i é um dígito. O alfabeto é $\{1, 2, \cdots, 9\}$.

Cada nó do “trie” ou é nulo ou tem n filhos.
Se o elemento $(a_1, a_2, \cdots, a_k)$ está presente, a raiz tem o filho de ordem a_1, Este filho tem o filho de ordem a_2, etc, até chegarmos a um filho de ordem a_k que por sua vez não tem filho nenhum (se a palavra não for prefixo próprio de outras).

“Tries” - I

Exercício 49 *Represente gráficamente um “trie” contendo os elementos* `ab`, `aba` *e* `c`. *O alfabeto é* {`a`, `b`, `c`}. ◇

Exercício 50 *Num “trie” existe um certo desperdício de espaço uma vez que o último nó correspondente a uma palavra (que não seja prefixo próprio de outras) tem os filhos todos nulos, não servindo para grande coisa... Diga como procederia para evitar este inconveniente.* ◇

Eficiência É fácil ver que os tempos de pesquisa, inserção ou eliminação de um elemento x são todos de ordem

$O(|x|)$ onde $|x|$ é o comprimento da sequência x

“Tries” - espaço

Ocupação de espaço numa visão optimista.
Hipótese: Altura: n.
Nós às alturas 1,..., $n-1$: cada tem a apontadores. Ocupam

$$(1+a+a^2+\cdots+a^{n-1})ab$$

bytes onde b é o número de bytes por apontador (4 por exemplo).
Nós à altura n: 2^n. Cada tem a bits de presença. Ocupação: $a^n a/8$. Total

$$t = ab\frac{a^n-1}{a-1}+\frac{a^{n+1}}{8}$$

Número de “strings” representados: a^{n+1}.
Número de bytes por “string” representado:

$$\approx \frac{b}{a-1}+\frac{1}{8}$$

o que para a grande é $\approx \frac{1}{8}$. Muito económico em termos de espaço!

A programação é mais complexa que a apresentada no exemplo!

Exemplo de uso de um “trie” - I

No programa seguinte inserem-se elementos num “trie”. Depois pesquisam-se elementos.

```
typedef struct no{
  struct no *filho[MAXF];
  boolean palavra;
} NO, *TRIE;

TRIE constroi(void);
void lista_trie(TRIE);

boolean procura(char *,TRIE);
main()
{
  TRIE t;
  char nome[MAXNOME];
  t=constroi();
  while(1){
    printf("Procura o que^ (0 para acabar)? ");
    scanf("%s",nome);
    if(strcmp(nome,"0")==0)
      break;
    if(procura(nome,t))
      printf("Existe\n");
    else
      printf("Nao existe\n");
  }
}
```


Exemplo de uso de um “trie” - II

```
boolean procura(char *s, TRIE t){
  if(t==0)
    return(NAO);
  if(s[0]==0)
    return(t->palavra);
  return(procura(&s[1],t->filho[s[0]-'a']));
}

TRIE constroi(void){
  TRIE t=0;
  char nome[MAXNOME];
  while(1){
    printf("Nome (0 para acabar)? ");
    scanf("%s",nome);
    if(strcmp(nome,"0")==0)
      break;
    t=insere(nome,t);
  }
  return(t);
}
```

Exemplo de uso de um "trie" - III

```
TRIE insere(char *s, TRIE t){
  TRIE tmp; int i;
  if(t!=0 && s[0]==0){   /* existe no trie */
    t->palavra=SIM;
    return(t);
  }
  if(t==0){
    tmp=(TRIE)malloc(sizeof(NO));
    tmp->palavra = s[0]==0;
    for(i=0;i<MAXF;i++)
      tmp->filho[i]=0;
    if(s[0]==0)
      return(tmp);
    tmp->filho[s[0]-'a'] =
          insere(&s[1],tmp->filho[s[0]-'a']);
    return(tmp);
  }
  /* Agora t!=0 e s!=0 */
  t->filho[s[0]-'a'] =
          insere(&s[1],t->filho[s[0]-'a']);
  return(t);
}
```


Métodos de "hashing"

Ideia geral

Uma implementação de dicionários:
Os elementos de um dicionário pertencem geralmente a um universo grande $\mathcal{D}$ ou até infinito - por exemplo, conjunto dos "strings". No método de "hashing" vão ser colocados em posições (índices) entre 0 e $n-1$ (conjunto dos índices). A função h ou *função de hashing* determina esses índices:

$$\begin{array}{rccc} h: & \mathcal{D} & \rightarrow & \{0, 1, \cdots, n-1\} \\ & x & \rightarrow & i = h(x) \end{array}$$

Como em geral $\mathcal{D}$ vai ter mais que n elementos (e podem não se conhecer antecipadamente os elementos que se pretendem inserir no dicionério), a função h não é normalmente injectiva.

Vai haver pois *colisões*, isto é pares (x, y) com $x \neq y$ tais que $h(x) = h(y)$; por outras palavras, pode haver elementos de $\mathcal{D}$ que "pretendem" ir para a mesma posição.
Há que resolvê-las! Veremos 2 métodos principais: o "hashing" interno e o "hashing" externo.

Métodos de “hashing”
Funções de “hashing”

Uma boa função de “hashing” deve

- Distribuir com igual probabilidade os elementos do domínio pelo conjunto dos índices.

- Evitar, na medida do possível, que dois elementos do domínio com grande probabilidade de ocorrerem conjuntamente tenham a mesma imagem.
  Exemplo Se num compilador representarmos por uma tabela de “hashing” o conjunto dos identificadores de um programa seria, por exemplo, má ideia usar (exclusivamente) os 2 primeiros caracteres de um identificador para calcular a função de “hashing” uma vez que são frequentes conjuntos de identificadores como “`contador1`”, “`contador2`”...

- Poder ser implementada com um algoritmo muito eficiente.
  Exemplo No caso dos identificadores pode não ser boa ideia considerar todos os seus caracteres.

Estes objectivos são de algum modo contraditórios e difíceis de atingir...

Métodos de “hashing”
Funções de “hashing”

Exemplo Domínio $\mathcal{D}$: inteiros representáveis como “`int`”.
Conjunto dos identificadores: $\{0, 1, \cdots, n-1\}$.
Função de “hashing”: $h(x) = x\%n$ (x módulo n).

Exemplo Domínio $\mathcal{D}$: Conjunto dos “strings”.
Conjunto dos identificadores: $\{0, 1, \cdots, n-1\}$.
Função de “hashing”:

```
int h(char *s){
  int m=strlen(s);
  return(s[0]+s[1]<<1+s[m-2]<<4+s[m-1]<<5)%n;
}
```

Comentário: dependência com o n...

Exercício 51 *Supondo* $n = 100$, *calcule:*
`h("contador1")` *e* `h("contador2")`. $\diamond$

“Hashing” interno

Os elementos são colocados num vector $h[]$ cujo índice vai de 0 a $n-1$.
Os algoritmos de inserção e pesquisa são simples. Para **inserir** $x \in \mathcal{D}$:

1. Calcula-se $i = h(x)$.

2. Se $h[i]$ estiver livre coloca-se lá x.
   Se não estiver, procura-se o primeiro elemento livre da sequência
   $$h[(i+1)\%n], h[(i+2)\%n], \cdots$$
   e coloca-se lá x.
   Se estiverem todos ocupados ocorre um erro (“overflow”).

Exemplo Caso $n = 5$, $h(x) = x\%n$ (x módulo n).
Inseriram-se sucessivamente os elementos 23, 34, 11, 43; obteve-se

0	1	2	3	4
43	11		23	34

“Hashing” interno
Eliminação

Para eliminar s não basta libertar a posição correspondente pois pode haver y, inserido depois de x que foi para a frente de x (ciclicamente) só porque x estava lá. No seguinte algoritmo supõe-se, por simplicidade, que a cadeia de elementos não “dá a volta” isto é, que não trabalhamos módulo n. No caso geral tem que ser adaptado. . .

```
/* ... i: posição em que está x ... */
j=i+1;
while(ocupado[j]){
  y=v[j];
  if(h(y)<=i){  // se imagem atrás de i...
    v[i]=y;     // ... y vai para i
    i=j;        // j é o novo buraco
    j=i+1;
  }
  else
    j=j+1;
}
```

Pior caso, $h(x)$ é sempre o mesmo:
Pesquisa com sucesso (x está na tabela): $n-1$ comparações.
Pesquisa sem sucesso (x não está na tabela): n comparações.

Caso médio Método muito eficiente, todas as operações são $O(1)$.
Resultado assimptótico, onde

$$\alpha = \frac{\text{Número de elementos na tabela}}{\text{Tamanho da tabela } (= n)}$$

Note-se que é sempre $0 \leq \alpha \leq 1$.
Quando α se aproxima de 1 o comportamento degrada-se. . .

Caso médio Método muito eficiente, todas as operações são $O(1)$.
Resultado assimptótico, onde

$$\alpha = \frac{\text{Número de elementos na tabela}}{\text{Tamanho da tabela } (= n)}$$

Note-se que é sempre $0 \leq \alpha \leq 1$.
Quando α se aproxima de 1 o comportamento degrada-se. . .

“Hashing” interno
Eficiência média assimptótica - I

O comportamento médio é $O(1)$ mas interessa caracterizá-lo melhor.

Contamos o número médio de acessos à tabela. Note-se que existe sempre pelo menos um acesso! Consideramos duas situações possíveis numa pesquisa de um elemento x:
C_α: Número médio de acessos de uma pesquisa com sucesso (x está na tabela).
C'_α: Número médio de acessos de uma pesquisa sem sucesso (x não está na tabela).

Valores médios para o “hashing” interno:
$C'_\alpha \approx \frac{1}{1-\alpha}$
$C_\alpha \approx \frac{1}{\alpha} \log \frac{1}{1-\alpha}$

“Hashing” interno
Eficiência média assimptótica - II

É possível reduzir a tamanho das cadeias que se formam quando α é próximo de 1.

- Uma ideia seria, em vez de usar a sequência
$$i,\ (i+1)\%n,\ (i+2)\%n, \cdots$$
usar
$$i,\ (i+p)\%n,\ (i+2p)\%n, \cdots$$
sendo n e p primos entre si. Todas os índices da tabela são percorridos.
Infelizmente o comportamento médio é igual! Tudo se passa como se a ordem dos índices fosse outra.

- A método anterior pode ser modificado por forma a reduzir de facto o tamanho médio das cadeias. Toma-se um número $p(i)$ que é primo com n mas depende de i. Por exemplo, $p(i) = i$ sendo n primo.

“Hashing” interno
Exemplo - I

“Hashing” interno
Exemplo - I - Início. . .

```
/* Teste de "hashing" interno.  Conta as colisoes */
#include <stdio.h>
#include <string.h>
#define NO   0
#define YES  1
#define MAXH 1000
#define MAXP 1000
typedef char boolean;
/* Dominio: conjunto das srings
   Contradominio: {0,1,2,...,MAXH-1}
   Tabela de hash: h[ ]                         */
struct no{
  char *word;
  boolean used;
} h[MAXH];
int n;            /* 0<n<MAXH                   */
boolean more=YES; /* O ficheiro não acabou      */
boolean got_word; /* Obteve nova palavra        */
int nd_pal=0, colisoes=0;

void get_word(char *p);
void display_table(void);
/*--------------------- funções do dicionario */
void make_empty(void);
boolean search(char *);
void insert(char *);
```

“Hashing” interno
Exemplo - II - “main”

```
main(){
  int n_pal=0;
  char p[MAXP];
  n=500;          /* Tamanho da tabels, n<MAXH */
  make_empty();
  while(more){
    get_word(p);
    if(got_word){
      n_pal++;
      insert(p);
    }
  }
  printf("Tamanho da tabela      = %d\n",n);
  printf("No. palavras distintas = %d\n",nd_pal);
  printf("No. palavras           = %d\n",n_pal);
  printf("No. colisões           = %d\n",colisoes);
  printf("Colisões por palavra   = %f\n",
                        ((double)colisoes)/n_pal);
  display_table();
}
```


“Hashing” interno
Exemplo - III - “hash”, “search”, “dict_void”

```
int hash(char *w){
  int i,sum=0;
  for(i=0;i<strlen(w);i++)
    sum += w[i]<<(i%8);
  return(sum%n);
}

void make_empty(void){
  int i;
  for(i=0;i<n;i++)
    h[i].used=NO;
}

boolean search(char *w){
  int count=0,
      k=hash(w);
  while(1){
    count++;
    if(!h[k].used || count==n)
      return(NO);
    if(strcmp(w,h[k].word)==0)
      return(YES);
    k=(k+1)%n;
  }
}
```

“Hashing” interno
Exemplo - IV - “insert”,

```
void insert(char *w){
  int count=0,
      k=hash(w);
  while(1){
    if(count==n){
      printf("** Table overflow\n");
      exit(1);
    }
    count++;
    if(!h[k].used){
      nd_pal++;
      h[k].used=YES;
      h[k].word = (char *) malloc(strlen(w)+1);
      strcpy(h[k].word,w);
      return;
    }
    if(strcmp(w,h[k].word)==0)
      return;
    colisoes++;
    k = (k+1)%n;
  }
  printf("out\n"); fflush(stdout);
}
```


“Hashing” interno
Exemplo - V - leitura...

```
void get_word(char *p){
  int c;
  got_word=NO;
  c=get_letter();
  if(!more) return;
  got_word=YES;
  *p++=c;
  while(letter(c=getchar()))
    *p++=c;
  *p++=0;
}

int get_letter(void){
  int c;
  while(1){
    c=getchar();
    if(letter(c)) return(c);
    if(c==-1)
      {more=NO; return(0);}
  }
}

boolean letter(int c){
  return(c>='a'&&c<='z' || c>='A'&&c<='Z' || c=='_');
}
```

“Hashing” interno
Exemplo - VI - fim e resultados

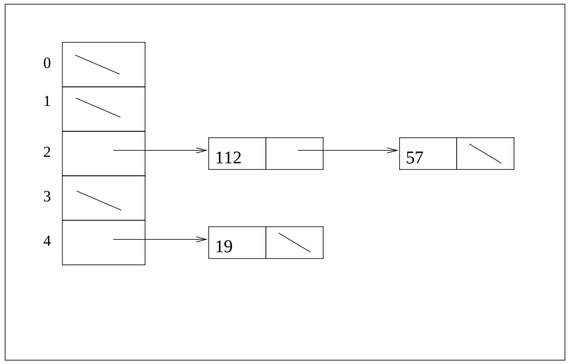
```
void display_table(void){
  int i,m=0;
  for(i=0;i<n;i++)
    if(h[i].used){
      printf("%15s",h[i].word);
      m++;
      if(m%5==0)
        printf("\n");
    }
  printf("\n");
}
```

Resultados, aplicação ao próprio programa...

```
progs$ a.out < hashi.c
Tamanho da tabela      = 500
No. palavras distintas = 115
No. palavras           = 468
No. colisões           = 96
Colisões por palavra   = 0.205128
       ficheiro           count          double
         struct          define          malloc
        interno            used            file
            YES            void          return
              A  Contradominio                0
etc.............................................
```


“Hashing” externo
Exemplo

- Domínio: conjunto dos inteiros.
- Função de hashing: n mod 5.
- Foram sucessivamente inseridos os elementos: 112, 19, 57



“Hashing” externo
Exemplo - “main”

```
#include <stdio.h>
#include <string.h>
#define MAX   100

void mostra_tabela(void);
void inicia(void);
void insere(char*);

main()
{
  char w[MAX];
  inicia();
  while(1){
    printf("Palavra ('fim' para terminar)? ");
    scanf("%s",w);
    if(strcmp(w,"end")==0)
      break;
    insere(w);
  }
  mostra_tabela();
}
```


“Hashing” externo
Exemplo - “hash”...

```
#define NO    0
#define YES   1
#define M     5
typedef char boolean;

int hash(char*);

struct no{
  char *c;
  struct no * next;
} *h[M];              /* tabela de hash */

void inicia(void){
  int i;
  for(i=0;i<M;i++)
    h[i]=NO;
}

int hash(char *w){
/* Uma funcao de hash muito simples... */
  int s=0, len=strlen(w), i;
  for(i=0;i<=len;i++)
    s^=((w[i]<<i)%M);
  return(s%M);
}
```

“Hashing” externo
Exemplo - “insere”

```
void insere(char *w){
/* Insere a palavra w na tabela de hash */
  int i=hash(w);
  struct no *p = h[i], *p_ant, *pt;
  boolean primeira_vez;
  primeira_vez=YES;
  while(p!=0){
    if(strcmp(p->c,w)==0)
      return;
    primeira_vez=NO;
    p_ant=p;
    p = p->next;
  }
  pt = (struct no*)malloc(sizeof(struct no));
  pt->next = 0;
  pt->c = (char *)malloc(strlen(w)+1);
  strcpy(pt->c,w);
  if(primeira_vez)
    h[i] = pt;
  else
    p_ant->next = pt;
}
```


“Hashing” externo
Exemplo - restos. . .

```
void mostra_tabela(void){
  int i;
  struct no *p;
  printf("*---*\n");
  for(i=0;i<M;i++){
    if(h[i]==0)
      printf("| \\ |\n ---\n");
    else{
      printf("|  --");
      p=h[i];
      while(p!=0){
        printf("--> %7s",p->c);
        p=p->next;
      }
      printf(" \\\n*---*\n");
    }
  }
}
```

“Hashing” externo
Exemplo - resultados

```
progs$ hashe
Word (end to finish)? jupiter
Word (end to finish)? urano
Word (end to finish)? saturno
Word (end to finish)? lua
Word (end to finish)? end
  *---*
  | \ |
   ---
  |  ----> saturno \
  *---*
  | \ |
   ---
  |  ----> urano   \
  *---*
  |  ----> jupiter -->   lua \
  *---*
```


Algoritmos em grafos

Alguns conceitos que devem estar presentes...

- Grafo dirigido: $G = (V, E)$, V é um conjunto (normalmente finito) de vértices, $E \subseteq V \times V$
- Grafo não dirigido.
- Subgrafo.
- Árvore.
- Sucessores (imediatos), descendentes.
- Caminhos e ciclos. Caminhos e ciclos simples.