September 30, 1998 at 07:16

1. Introducao. Este programa `freq` ilustra um uso de tabelas de espalhamentos. O usuário fornece como entrada uma lista de palavras, uma palavra por linha, e a saída devolvida por `freq` é uma lista das palavras que ocorrem na entrada, sem repetição, uma por linha, com cada uma delas seguida do número de vezes que ela ocorre na entrada.

Por simplicidade, assumimos que a entrada vem no *stdin* e que a saída é devolvida no *stdout*. Mensagens de erro são enviadas para *stderr* e se encontramos algum erro terminamos a execução e devolvemos um código de erro negativo.

Executaremos a nossa tarefa da seguinte forma. Manteremos uma tabela de espalhamento para o conjunto de palavras da entrada. Em um dado instante, já lemos um certo número de palavras, e elas estão armazenadas em uma tabela de espalhamento, com as respectivas freqüências até o momento registradas. Ao lermos uma nova palavra, nós a procuramos em nossa tabela. Se a encontramos, incrementamos o contador dela, e caso contrário nós a inserimos em nossa tabela.

Pretendo adicionar algumas coisas a este programa, para aumentar a flexibilidade de uso e eficiência. Espero que eu possa fazer isto em breve. Esta versão, entretanto, tem a vantagem de ser "minimal" e portanto possivelmente mais facil de entender

2. A organização de `freq` é típica de programas em C.

```
#include <stdio.h>
#include <string.h>
  ⟨ Typedefs 3 ⟩
  ⟨ Funções 6 ⟩
  int main( )
  { ⟨ Variáveis locais a main 4 ⟩;
    ⟨ Inicialize a tabela de espalhamento 7 ⟩;
    ⟨ Leia a entrada palavra por palavra e as processe uma por vez 8 ⟩;
    ⟨ Imprima a lista de palavras e seu número de ocorrências 11 ⟩;
    ⟨ Imprima o número de palavras lidas e o número de comparações feitas 14 ⟩;
    return 0;
  }
```

3. Usaremos resolução de colisões por encadeamento em nosso esquema de *hashing*. Os elementos das listas ligadas em nossa tabela de espalhamento vão ser as *celulas*, que são as estruturas definidas abaixo.

```
⟨ Typedefs 3 ⟩ ≡
  typedef struct celula_struct {
    char *palavra;   /* a palavra */
    int n_ocorrencias;   /* número de ocorrências da palavra */
    struct celula_struct *prox;   /* para a lista ligada */
  } celula;   /* células das listas ligadas */
```

This code is used in section 2.

4. Assumiremos que as palavras da entrada têm no máximo `COMP_MAX` caracteres.

```
#define COMP_MAX 50
⟨ Variáveis locais a main 4 ⟩ ≡
  char s[COMP_MAX];   /* a palavra corrente */
  long n_pals = 0;   /* número de palavras lidas */
  long n_pals_dist = 0;   /* número de palavras lidas distintas */
```

See also sections 5 and 13.

This code is used in section 2.

5. Tabelas de espalhamento. Usaremos uma das funções de *hashing* vistas em sala, e usaremos resolução de colisões por encadeamento.

#define `HASH_PRIME` 1009

⟨ Variáveis locais a *main* 4 ⟩ +≡
  celula $*tab$[`HASH_PRIME`];   /* a tabela de *hashing* */

6. ⟨ Funções 6 ⟩ ≡
  int *hash_div*(**char** $*s$)
  { **int** h;
    for $(h = 0;\ *s \neq$ `'\0'`$;\ s{+}{+})\ h = (258 * h +$ (**unsigned char**) $*s)$ % `HASH_PRIME`;
    return h;
  }

This code is used in section 2.

7. ⟨ Inicialize a tabela de espalhamento 7 ⟩ ≡
  { **int** i;
    for $(i = 0;\ i <$ `HASH_PRIME`$;\ i{+}{+})\ tab[i] = \Lambda$;
  }

This code is used in section 2.

8. ⟨ Leia a entrada palavra por palavra e as processe uma por vez 8 ⟩ ≡
  while (*gets*(s)) {
    int *h_valor*;
    n_pals++;
    h_valor = *hash_div*(s);
    ⟨ Procure s na lista *tab*[*h_valor*]; insira ou incremente contador 9 ⟩;
  }

This code is used in section 2.

9. ⟨ Procure s na lista *tab*[*h_valor*]; insira ou incremente contador 9 ⟩ ≡
  { **int** *achou* = 0;
    celula $*ptr$ = *tab*[*h_valor*];
    while (($\neg$*achou*) $\wedge$ *ptr*)
      if (o, *strcmp*(s, *ptr*→*palavra*) $\equiv 0$) *achou* = 1;
      else *ptr* = *ptr*→*prox*;
    if (*achou*) *ptr*→*n_ocorrencias*++;
    else ⟨ Insira nova célula na lista *tab*[*h_valor*] 10 ⟩;
  }

This code is used in section 8.

10. ⟨ Insira nova célula na lista *tab*[*h_valor*] 10 ⟩ ≡

```
{ celula *cel_nova;
  char *t;
  cel_nova = (celula *) malloc(sizeof(celula));
  if (cel_nova ≡ Λ) {
    fprintf(stderr, "freq:␣faltou␣memoria␣para␣celulas!\n");
    return −1;
  }
  t = (char *) malloc(strlen(s) + 1);      /* +1 para o '\0' */
  if (t ≡ Λ) {
    fprintf(stderr, "freq:␣faltou␣memoria␣para␣caracteres!\n");
    return −2;
  }
  strcpy(t, s);
  cel_nova→palavra = t;
  cel_nova→n_ocorrencias = 1;
  n_pals_dist++;
  cel_nova→prox = tab[h_valor];
  tab[h_valor] = cel_nova;
}
```

This code is used in section 9.

11. ⟨ Imprima a lista de palavras e seu número de ocorrências 11 ⟩ ≡

```
{ int i;
  for (i = 0; i < HASH_PRIME; i++) ⟨ Imprima as palavras na lista tab[i] 12 ⟩;
}
```

This code is used in section 2.

12. ⟨ Imprima as palavras na lista *tab*[*i*] 12 ⟩ ≡

```
{ celula *ptr = tab[i];
  for ( ; ptr; ptr = ptr→prox) printf("%s␣%d\n", ptr→palavra, ptr→n_ocorrencias);
}
```

This code is used in section 11.

13. Faremos uma contagem do número de comparações que são feitas entre palavras. Usaremos um contador *n_comp* para isto.

```
#define o n_comp++
```

⟨ Variáveis locais a *main* 4 ⟩ +≡

```
  long n_comp = 0;
```

14. ⟨ Imprima o número de palavras lidas e o número de comparações feitas 14 ⟩ ≡

```
  printf("Numero␣de␣palavras␣lidas␣=␣%ld\n", n_pals);
  printf("Numero␣de␣comparacoes␣feitas␣=␣%ld\n", n_comp);
  printf("Numero␣medio␣de␣comparacoes␣feitas␣por␣palavra␣lida␣=␣%f\n", (double)
      n_comp/n_pals);
  printf("Numero␣de␣palavras␣distintas␣=␣%ld\n", n_pals_dist);
  printf("Numero␣esperado␣de␣palavras␣por␣lista␣ligada␣=␣%f\n",
      (double) n_pals_dist/HASH_PRIME);
```

This code is used in section 2.

15. Indice. Aqui está uma lista de identificadores que são usados, com indicação de onde eles aparecem. As entradas sublinhadas indicam onde eles aparecem.