September 30, 1998 at 09:34

1. Introducao. Este programa `freq_ABB` ilustra um uso de árvores binárias de busca. Como no caso do programa `freq`, o usuário fornece como entrada uma lista de palavras, uma palavra por linha, e a saída devolvida por `freq_ABB` 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 árvore binária de busca contendo o conjunto corrente de palavras. Isto é, em um dado instante, já lemos um certo número de palavras, e elas estão armazenadas em nossa árvore, com as respectivas freqüências até o momento registradas. Ao lermos uma nova palavra, nós a procuramos em nossa árvore. Se a encontramos, incrementamos o contador dela, e caso contrário nós a inserimos em nossa árvore.

Esta é uma versão "minimal" deste exemplo, equivalente ao programa `freq`. Vale a pena comparar e conferir os programas em si e também o desempenho de `freq` e `freq_ABB`.

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

```
#include <stdio.h>
#include <string.h>
  ⟨ Typedefs 3 ⟩
  ⟨ Funções 14 ⟩
  int main( )
  { ⟨ Variáveis locais a main 4 ⟩;
    ⟨ Inicialize a árvore binária de busca 6 ⟩;
    ⟨ Leia a entrada palavra por palavra e as processe uma por vez 7 ⟩;
    ⟨ Imprima a lista de palavras e seu número de ocorrências 12 ⟩;
    ⟨ Imprima o número de palavras lidas e o número de comparações feitas 17 ⟩;
    return 0;
  }
```

3. Usaremos as estruturas *no_struct* como os nós de nossa árvore binária de busca. Por simplicidade, definimos o tipo **No** através de um **typedef**.

```
⟨ Typedefs 3 ⟩ ≡
  typedef struct no_struct {
    char *palavra;    /* a palavra */
    int n_ocorrencias;    /* número de ocorrências da palavra */
    struct no_struct *f_esq, *f_dir;    /* para os filhos esquerdo e direito */
  } No;    /* 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, 13, and 16.

This code is used in section 2.

5. Arvores Binarias de Busca. A única rotina de manipulação de árvores de que precisamos é a de busca e inserção, visto em sala.

⟨ Variáveis locais a *main* 4 ⟩ +≡

```
No *t;    /* o ponteiro para a raiz da árvore */
```

6. ⟨ Inicialize a árvore binária de busca 6 ⟩ ≡

```
t = Λ;
```

This code is used in section 2.

7. ⟨ Leia a entrada palavra por palavra e as processe uma por vez 7 ⟩ ≡

```
while (gets(s)) {
  n_pals++;
  ⟨ Procure s na árvore t; insira um novo nó caso não a encontre 8 ⟩;
}
```

This code is used in section 2.

8. ⟨ Procure *s* na árvore *t*; insira um novo nó caso não a encontre 8 ⟩ ≡

```
if (¬t)    /* a árvore está vazia */
  ⟨ Construa uma árvore com um único nó para s 9 ⟩
else    /* a árvore não está vazia */
  ⟨ Busque s em t 10 ⟩;
```

This code is used in section 7.

9. Começamos com o mais fácil.

⟨ Construa uma árvore com um único nó para *s* 9 ⟩ ≡

```
{ char *s_aux;
  t = (No *) malloc(sizeof(No));
  if (¬t) {
    fprintf(stderr, "freq_ABB:␣faltou␣memoria␣para␣nos␣da␣arvore!\n");
    return −1;
  }
  s_aux = (char *) malloc(strlen(s) + 1);    /* +1 para o '\0' */
  if (¬s_aux) {
    fprintf(stderr, "freq_ABB:␣faltou␣memoria␣para␣caracteres!\n");
    return −2;
  }
  strcpy(s_aux, s);    /* fazemos uma cópia de s em s_aux */
  t→palavra = s_aux;
  t→n_ocorrencias = 1;
  t→f_esq = t→f_dir = Λ;
  n_pals = n_pals_dist = 1;
}
```

This code is used in section 8.

10. Agora sabemos que estamos lidando com uma árvore não vazia. Procuramos s da segunte forma: primeiro comparamos s com a palavra na raiz da árvore (i.e., *t→palavra*). Se $s <$ *t→palavra*, continuamos a busca na subárvore esquerda de t, cuja raiz é *t→f_esq*. Analogamente, se $s >$ *t→palavra*, continuamos a busca na subárvore direita de t, cuja raiz é *t→f_dir*. Se $s =$ *t→palavra*, encontramos s! Quando descemos nas subárvores esquerda ou direita, continuamos o mesmo processo. Para este procedimento, usamos um ponteiro para nós p que percorre a árvore (vai "descendo" na árvore). Temos ainda um ponteiro auxiliar *p_anterior*, que aponta para o nó visitado logo antes de visitar p.

```
⟨ Busque s em t 10 ⟩ ≡
  { int achou = 0;
    int comp_res;     /* resultado da comparação entre s e p→palavra */
    No *p = t;
    No *p_anterior;
    while (¬achou ∧ p)
      if (o, (comp_res = strcmp(s, p→palavra)) < 0) {
        p_anterior = p;
        p = p→f_esq;
      }
      else if (comp_res > 0) {
        p_anterior = p;
        p = p→f_dir;
      }
      else achou = 1;
    if (achou) p→n_ocorrencias++;
    else ⟨ Insira um novo nó para a palavra s 11 ⟩;
  }
```

This code is used in section 8.

11. ⟨ Insira um novo nó para a palavra s 11 ⟩ ≡

```
  { No *no_novo;
    char *s_aux;
    no_novo = (No *) malloc(sizeof(No));
    if (¬no_novo) {
      fprintf(stderr, "freq_ABB:␣faltou␣memoria␣para␣nos!\n");
      return −1;
    }
    s_aux = (char *) malloc(strlen(s) + 1);     /* +1 para o '\0' */
    if (¬s_aux) {
      fprintf(stderr, "freq_ABB:␣faltou␣memoria␣para␣caracteres!\n");
      return −2;
    }
    strcpy(s_aux, s);
    no_novo→palavra = s_aux;
    no_novo→n_ocorrencias = 1;
    no_novo→f_esq = no_novo→f_dir = Λ;
    n_pals_dist++;
    if (comp_res < 0) p_anterior→f_esq = no_novo;
    else p_anterior→f_dir = no_novo;
  }
```

This code is used in section 10.

12. Percorremos a árvore em in-ordem para imprimir as palavras que encontramos na entrada, sem repetição e em ordem alfabética. A função *in_ordem*( ) abaixo é trivialmente implementada recursivamente.

Na realidade, fazemos mais que simplesmente imprimir as palavras e suas respectivas ocorrências. Vamos determinar a profundidade da árvore e também a profundidade média dos nós da árvore. Recordemos que a profundidade de um nó na árvore é o comprimento do caminho da raiz até ele. A profundidade da árvore é o máximo das profundidades dos nós que ocorrem na árvore. A função *in_ordem*( ) também determina o número de nós na árvore; naturalmente, isto deve bater com o número de palavras distintas na entrada.

```
⟨ Imprima a lista de palavras e seu número de ocorrências 12 ⟩ ≡
  in_ordem(t, &n_nos, &profdd, &profdd_media);
  if (n_pals_dist ≠ n_nos) {
    fprintf(stderr, "freq_ABB:␣n_nos␣e␣n_pals␣se␣contradizem.␣␣Conserte-me␣por␣favor!\n");
    return −3;
  }
```

This code is used in section 2.

```
13. ⟨ Variáveis locais a main 4 ⟩ +≡
  long profdd;    /* a profundidade da árvore */
  double profdd_media;    /* a profundidade média dos nós da árvore */
  long n_nos;    /* o número de nós na árvore */
```

```
14. ⟨ Funções 14 ⟩ ≡
  void in_ordem(No *p, long *n_nos, long *profdd, double *profdd_media)
  { long profdd_esq, profdd_dir;
    double profdd_media_esq, profdd_media_dir;
    long n_nos_esq, n_nos_dir;
    if (p) {
      in_ordem(p→f_esq, &n_nos_esq, &profdd_esq, &profdd_media_esq);
      ⟨ Visite p 15 ⟩;
      in_ordem(p→f_dir, &n_nos_dir, &profdd_dir, &profdd_media_dir);
      *n_nos = n_nos_esq + n_nos_dir + 1;
      if (*n_nos ≡ 1) *profdd = *profdd_media = 0;
      else {
        *profdd_media = 1 + (profdd_media_esq * n_nos_esq + profdd_media_dir * n_nos_dir − 1)/(*n_nos);
        *profdd = 1 + ((profdd_esq > profdd_dir) ? profdd_esq : profdd_dir);
      }
    }
    else {    /* a árvore está vazia */
      *n_nos = 0;
      *profdd = *profdd_media = −1;    /* apenas por convenção */
    }
  }
```

This code is used in section 2.

```
15. ⟨ Visite p 15 ⟩ ≡
  printf("%s␣%d\n", p→palavra, p→n_ocorrencias);
```

This code is used in section 14.

16. 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;
```

17. ⟨ Imprima o número de palavras lidas e o número de comparações feitas 17 ⟩ ≡

```
printf("\nNumero␣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("\nNumero␣de␣palavras␣distintas␣=␣%ld\n", n_pals_dist);
printf("Profundidade␣da␣arvore␣=␣%ld\n", profdd);
printf("Profundidade␣media␣dos␣nos␣da␣arvore␣=␣%f\n", profdd_media);
```

This code is used in section 2.

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