November 14, 2013 at 07:48

1. Introducao. Este programa resolve um problema proposto pelos professores Imre Simon e Paulo Feofiloff em MAC122. Resolvi escrever este programa porque gostei do problema e achei que poderia me divertir em meu tempo livre escrevendo este programa. (Tempo livre? Que tempo livre?) A solução que implementei está além do escopo de MAC122; acredito entretanto que pode ser estimulante para os alunos desta disciplina ver uma solução usando uma estrutura de dados muito interessante, chamada *treaps*. Com um pouco de sorte, esta estrutura será discutida na disciplina MAC323, Estruturas de Dados.

Este programa, na verdade, deve ser fácil para eu escrever, pois estou basicamente apenas modificando um programa já pronto, devido a Knuth, a saber, o `wordtest`, escrito em `CWEB`. O `CWEB` é um sistema de programação baseado na filosofia de *Programação Literária* (*Literate Programming*).

2. O problema proposto é o seguinte. É dado um arquivo de palavras, com uma palavra por linha. Vamos admitir que todas as palavras deste arquivo são distintas. (Usando `sort` e `uniq`, podemos sempre eliminar palavras repetidas.) Dizemos que duas palavras são *equivalentes* se uma for anagrama da outra. Deseja-se encontrar um subconjunto das palavras do arquivo dado que sejam todas equivalentes entre si; ademais, queremos um tal subconjunto de maior *peso* possível, onde o *peso* de um conjunto de palavras é o número total de letras neste conjunto. Note que a relação de ser anagrama é uma relação de equivalência, de forma que estamos à procura de uma classe de equivalência de peso maior possível; em outras palavras, queremos maximizar o produto lm, onde l é o comprimento comum das palavras na classe de equivalência e m é o número de elementos nesta classe de equivalência.

Este programa recebe no *stdin* o arquivo dado e devolve em *stdout* um subconjunto de palavras equivalentes de peso máximo. Assumimos que todas as palavras de *stdin* terminam com `'\n'`.

3. A idéia básica que usaremos é a seguinte. Associamos a cada palavra s uma *forma canônica*, a saber, a palavra t que têm exatamente as mesmas letras de s, mas em ordem alfabética (de acordo com o código ASCII dos caracteres). Por exemplo, se s é a palavra "abóbora", a forma canônica de s é a palavra "aabboró" (bem, não sei se chamar esta palavra de "palavra" faz sentido). Então, duas palavras são equivalentes se e só se elas têm a mesma forma canônica.

4. Supomos que as palavras tem no máximo `MAX_LENGTH_DEFAULT` caracteres.

```
#define MAX_LENGTH_DEFAULT 50
```

5. A organização deste programa é a típica de programas em C. Se erros são detectados, uma mensagem é enviada para *stderr* e um valor não nulo é devolvido. O usuário pode escolher entre duas opções. Ele pode pedir que o programa envie para *stdout* as palavras de uma classe de equivalência de maior peso (este é o comportamento default), ou ele pode especificar a opção `-nN`, onde `N` é um inteiro. Com esta opção, este programa determina os `n` maiores inteiros w para os quais existem classes de equivalência de peso w e imprime todas estas classes de equivalência. Desta forma, o comportamento deste programa quando ele é chamado sem opção nenhuma é basicamente o mesmo daquele quando ele é chamado com a opção `-n1`.

```
#include <stdio.h>
#include <stdlib.h>
  ⟨ Typedefs 7 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ Variáveis locais 8 ⟩;
    ⟨ Interprete a linha de comando 6 ⟩;
    ⟨ Processe a lista de palavras 18 ⟩;
    if (n_pesos) ⟨ Imprima as classes cujos pesos são os n_pesos maiores pesos 27 ⟩
    else ⟨ Encontre uma classe de equivalência de peso máximo e imprima 24 ⟩
    return 0;
  }
```

6. ⟨ Interprete a linha de comando 6 ⟩ ≡

```
  while (--argc) {
    if (sscanf(argv[argc], "-n%ld", &n_pesos) ≡ 1) ;
    else {
      fprintf(stderr, "Uso: %s [-nN]\n", argv[0]);
      return -6;
    }
  }
```

This code is used in section 5.

7. Usaremos o tipo *byte* para os caracteres. As palavras de uma classe de equivalência serão colocadas em uma lista ligada, cujas células serão do tipo *cell*.

⟨ Typedefs 7 ⟩ ≡

```
  typedef unsigned char byte;      /* um byte varia de 0 a 255 */
  typedef struct list_node *link;
  typedef struct list_node {
    byte *w;
    link next;
  } cell;      /* célula das listas ligadas de palavras */
```

See also section 9.

This code is used in section 5.

8. ⟨ Variáveis locais 8 ⟩ ≡

```
  unsigned max_length = MAX_LENGTH_DEFAULT;      /* comprimento máximo de uma palavra */
  register long c;      /* índice genérico */
  register byte *u, *v;      /* ponteiros para caracteres de strings sendo processados no momento */
  long n_pesos = 0;      /* imprimimos as classes cujos pesos estão dentre os n_pesos maiores pesos */
  long N_classes = 0;      /* número de classes encontradas */
```

See also sections 11, 16, and 17.

This code is used in section 5.

9. Treaps. A parte mais interessante deste programa é a estrutura de dados que é utilizada para se manter uma tabela de símbolos. Usaremos a estrutura de "treap" de Aragon e Seidel [*30th IEEE Symposium on Foundations of Computer Science* (1989), 540–546]. Um treap é uma árvore binária cujos nós têm duas chaves. Os nós da árvore vão corresponder às classes de equivalências das palavras da lista de palavras dadas. A chave primária de um nó, que em nossa aplicação será a forma canônica naturalmente associada à classe de equivalência deste nó, obedece a condição de árvore de busca binária: o filho esquerdo de um nó p e todos os seus descendentes têm chave primária menor que a chave primária de p, e o filho direito de p e todos os seus descendentes têm chave primária maior que a chave primária de p. A chave secundária, que em nossa aplicação será um número pseudoaleatório associado ao nó em questão, obedece a *propriedade de heap*: a chave secundária de todos os descendentes de um nó p têm chave secundária maior que a chave secundária de p.

Um dado conjunto de nós com chaves primárias e secundárais todas distintas pode ser organizado em um treap de forma única. Este treap pode ser obtido, por exemplo, usando-se o procedimento padrão de inserção em árvore binária: basta se inserir os nós em ordem crescente da chave secundária. Daí segue que se as chaves secundárias forem aleatórias, a árvore será balanceada com alta probabilidade.

As chaves secundárias serão **unsigned long**, e o valor atribuído ao n-ésimo nó será $(cn) \bmod 2^{32}$, onde c é um número ímpar. Isto vai garantir que as chaves secundárias sejam todas distintas. Escolhendo c próximo a $(cn) \bmod 2^{32}$, onde ϕ é a razão áurea $(1+\sqrt{5}\,)/2$, teremos, muito provavelmente, uma distribuição aleatória o suficiente em relação aos dados de entrada. Para uma discussão deste fato, veja o volume 3 do TAOCP de Knuth [*The Art of Computer Programming*, *Sorting and Searching* (seção sobre hashing)].

Em nosso treap, cada classe de equivalência corresponde a um nó. Este nó contém um campo m que registra o número corrente de elementos nesta classe de equivalência; temos também o campo l que contém o comprimento comum das palavras desta classe. Ademais, um campo h aponta para o primeiro elemento de uma lista ligada que contém as palavras desta classe de equivalência.

```
#define PHICLONE 2654435769 /* ≈ 2^32/φ */
```

⟨ Typedefs 7 ⟩ +≡

```
typedef struct node_struct {
  struct node_struct *left, *right; /* filhos */
  byte *keyword; /* chave primária */
  unsigned long rank; /* chave secundária */
  long m; /* número de palavras na classe de equivalência */
  long l; /* comprimento das palavras desta classe de equivalência */
  link h; /* início da lista de palavras da classe de equivalência */
} node; /* nó do treap */
```

10. Nesta implementação, quando descobrimos que queremos inserir um novo nó no treap, refazemos a busca (sem sucesso) para determinar o local da árvore onde devemos fazer a inserção. Este processo é um pouco mais demorado que o processo de busca simples, no qual queremos apenas determinar a presença ou não de um elemento no treap. Naturalmente, ao invés de adotarmos a estratégia que escolhemos (busca simples e repetição de trabalho quando precisamos executar uma inserção), poderíamos executar este processo de busca um pouco mais complicado sempre, para evitar esta repetição de trabalho no caso em que inserção é necessária. Não é claro qual das duas estratégias é melhor em geral. Isto depende, naturalmente, da proporção de buscas sem sucesso que esperamos com as nossas entradas.

No problema que queremos resolver, muito provavelmente o número de buscas sem sucesso é muito grande, e portanto a estratégia alternativa (aquela não utilizada neste programa) deve ser a mais eficiente! Fica como exercício para o leitor alterar este programa e verificar experimentalmente o ganho de eficiência. Um argumento heurístico sugere que o ganho deve ser substancial.

O processo de inserção de uma nova classe de equivalência (isto é, um novo nó) em nosso treap é parecido com o processo de busca. Percorremos o treap até encontrarmos um nó cuja chave secundária é maior que a do nó a ser inserido. Inserimos então o novo nó nesta posição, e partimos o nó antigo e seus descendentes em duas subárvores que se tornarão as subárvores esquerda e direita do nó recém-inserido.

```
⟨ Insira o nó correspondente a buffer no treap 10 ⟩ ≡
  { register node *p, **q, **qq, *r;
    current_rank += PHICLONE;      /* soma (unsigned) mod 2^32 */
    p = root; q = &root;
    while (p) {
      if (p→rank > current_rank) break;      /* fim da primeira fase */
      for (u = buffer, v = p→keyword; *u ≡ *v; u++, v++) ;
      if (*u < *v) q = &(p→left), p = *q;
      else q = &(p→right), p = *q;
    }
    ⟨ Faça r apontar para um novo nó e ponha buffer neste nó 13 ⟩;
    r→rank = current_rank;
    *q = r;      /* adicionamos o novo nó na árvore */
    N_classes++;
    ⟨ Quebre a subárvore p e pendure as duas árvores resultantes em r 12 ⟩;
  }
```

This code is used in section 18.

11. ⟨ Variáveis locais 8 ⟩ +≡

```
  unsigned long current_rank = 0;      /* número pseudoaleatório */
```

12. Neste ponto a árvore p pode já estar vazia. Caso contrário, executamos um procedimento de partição nesta árvore e penduramos as duas árvores resultantes em r. O fato de que este processo de partição resulta em um treap é não trivial, e fica como exercício para o leitor.

```
⟨ Quebre a subárvore p e pendure as duas árvores resultantes em r 12 ⟩ ≡
  q = &(r→left); qq = &(r→right);      /* ponteiros a serem determinados durante o processo */
  while (p) {
    for (u = buffer, v = p→keyword; *u ≡ *v; u++, v++) ;
    if (*u < *v) {
      *qq = p;
      qq = &(p→left);
      p = *qq;
    }
    else {
      *q = p;
      q = &(p→right);
      p = *q;
    }
  }
  *q = *qq = Λ;
```

This code is used in section 10.

13. Alocamos memória para os nós de nossa árvore dinamicamente, 500 nós por vez, para evitar chamadas frequentes ao sistema. Analogamente, alocamos memória dinamicamente para as células de nossas listas ligadas, 500 células por vez. Também alocamos memória para os nossos strings dinamicamente, 5000 caracteres por vez (além do necessário para uso imediato). Usaremos a variável l para guardar o comprimento da palavra em *buffer*.

```
#define NODES_PER_BLOCK 500
#define CELLS_PER_BLOCK 500
#define CHARS_PER_BLOCK 5000
#define out_of_mem(x)
        { fprintf(stderr, "%s: Acabou memória!\n", *argv);
          return x; }
⟨ Faça r apontar para um novo nó e ponha buffer neste nó 13 ⟩ ≡
  if (next_node ≡ bad_node) {
    next_node = (node *) calloc(NODES_PER_BLOCK, sizeof(node));
    if (next_node ≡ Λ) out_of_mem(−2);
    bad_node = next_node + NODES_PER_BLOCK;
  }
  r = next_node++;
  ⟨ Copie buffer na memória alocada para strings e faça r→keyword apontar para esta cópia 14 ⟩;
  ⟨ Insira palavra_corrente na lista de r 15 ⟩;
```

This code is used in section 10.

14. ⟨ Copie *buffer* na memória alocada para strings e faça *r*¬*keyword* apontar para esta cópia 14 ⟩ ≡

```
if (next_string + l + 1 ≥ bad_string) { int block_size = CHARS_PER_BLOCK + l + 1;
  next_string = (byte *) malloc(block_size);
  if (next_string ≡ Λ) out_of_mem(−3);
  bad_string = next_string + block_size;
}
r¬keyword = next_string;
for (u = buffer, v = next_string; *u ≠ '\n'; u++, v++) *v = *u;
*v = '\0';
next_string = v + 1;
```

This code is used in section 13.

15. ⟨ Insira *palavra_corrente* na lista de *r* 15 ⟩ ≡

```
{ register link p;
  if (next_cell ≡ bad_cell) {
    next_cell = (cell *) calloc(CELLS_PER_BLOCK, sizeof(cell));
    if (next_cell ≡ Λ) out_of_mem(−4);
    bad_cell = next_cell + CELLS_PER_BLOCK;
  }
  p = next_cell++;
  p¬w = palavra_corrente;
  p¬next = Λ;
  r¬h = p;
  r¬m = 1;
  r¬l = l;
}
```

This code is used in section 13.

16. Declaramos aqui as variáveis que temos usado nas rotinas de alocação de memória.

⟨ Variáveis locais 8 ⟩ +≡

```
node *next_node = Λ, *bad_node = Λ;
cell *next_cell = Λ, *bad_cell = Λ;
byte *next_string = Λ, *bad_string = Λ;
node *root = Λ;
byte *buffer;
byte *palavra_corrente;
long l;      /* comprimento da palavra em buffer */
```

17. Ao terminarmos uma busca com sucesso em nosso treap, teremos *anagrama* ≡ 1 e *loco_encontrado* será o nó do treap onde encontramos a palavra procurada.

⟨ Variáveis locais 8 ⟩ +≡

```
node *loco_encontrado = Λ;
long anagrama = 0;
```

18. Os mecanismos de busca e inserção em nosso treap já estão quase todos prontos; devemos apenas executá-los nos momentos apropriados.

```
⟨ Processe a lista de palavras 18 ⟩ ≡
  buffer = (byte *) malloc(max_length + 1);
  if (buffer ≡ Λ) out_of_mem(−5);
  while (1) {
    ⟨ Leia em buffer a próxima palavra em stdin; goto done caso o arquivo terminou 23 ⟩;
    ⟨ Faça uma cópia de buffer na memória alocada para strings e canonize buffer "in loco" 21 ⟩;
    if (l) {
      ⟨ Procure buffer no treap 19 ⟩;
      if (anagrama) ⟨ Insira palavra_corrente na lista do loco_encontrado 20 ⟩
      else ⟨ Insira o nó correspondente a buffer no treap 10 ⟩;
    }
  }
done: ;
```

This code is used in section 5.

19. Implementamos abaixo a nossa própria versão de *strcmp*( ), que é usada para a comparação da forma canônica da palavra lida de *stdin* e da chave primária de um nó do treap. Supomos que a forma canônica está em *buffer*, terminada por `'\n'`, e que as chaves primárias terminam com `'\0'`. Fazemos acesso ao treap através de *root*, um ponteiro para a sua raiz.

```
⟨ Procure buffer no treap 19 ⟩ ≡
  { register node *p = root;
    while (p) {
      for (u = buffer, v = p→keyword; *u ≡ *v; u++, v++) ;
      if (*v ≡ '\0' ∧ *u ≡ '\n') {
        anagrama = 1;
        loco_encontrado = p;
        break;
      }
      if (*u < *v) p = p→left;
      else p = p→right;
    }
  }
```

This code is used in section 18.

20.

```
⟨ Insira palavra_corrente na lista do loco_encontrado 20 ⟩ ≡
  { register link p;
    if (next_cell ≡ bad_cell) {
      next_cell = (cell *) calloc(CELLS_PER_BLOCK, sizeof(cell));
      if (next_cell ≡ Λ) out_of_mem(−4);
      bad_cell = next_cell + CELLS_PER_BLOCK;
    }
    p = next_cell++;
    p→w = palavra_corrente;
    p→next = loco_encontrado→h;
    loco_encontrado→h = p;
    loco_encontrado→m++;
  }
```

This code is used in section 18.

21. ⟨ Faça uma cópia de *buffer* na memória alocada para strings e canonize *buffer* "in loco" 21 ⟩ ≡

```
{
  if (next_string + l + 1 ≥ bad_string) { int block_size = CHARS_PER_BLOCK + l + 1;
    next_string = (byte *) malloc(block_size);
    if (next_string ≡ Λ) out_of_mem(−3);
    bad_string = next_string + block_size;
  }
  palavra_corrente = next_string;
  for (u = buffer, v = next_string; *u ≠ '\n'; u++, v++) *v = *u;
  *v = '\0';
  next_string = v + 1;
  ⟨ Canonize buffer 22 ⟩;
}
```

This code is used in section 18.

22. Usamos aqui o *insertion sort*.

⟨ Canonize *buffer* 22 ⟩ ≡

```
{ register byte *p, *q;
  register byte b;
  for (p = buffer + 1; p < buffer + l; p++)
    for (q = p; q > buffer; q−−)
      if (*(q − 1) < *q) break;
      else {
        b = *(q − 1);
        *(q − 1) = *q;
        *q = b;
      }
}
```

This code is used in section 21.

23. ⟨ Leia em *buffer* a próxima palavra em *stdin*; **goto** *done* caso o arquivo terminou 23 ⟩ ≡

```
u = buffer; l = 0; anagrama = 0;
while (l < max_length) {
  c = getchar( );
  if (c ≡ EOF) {
    if (ferror(stdin)) {
      fprintf(stderr, "%s: Erro na leitura de stdin!\n", *argv);
      return −6;
    }
    goto done;      /* fim de arquivo; a palavra corrente, se houver uma, é ignorada */
  }
  if (c ≡ '\n') break;
  else {
    *u++ = (byte) c;
    l++;
  }
}
*u = '\n';
```

This code is used in section 18.

24. Após termos processado todas as palavras da entrada, percorreremos o nosso treap e montaremos uma lista ligada com os nós desta árvore. Ao fazermos isto, destruiremos o nosso treap. Nest fase, *root* vai apontar para uma pilha de nós que ainda devem ser visitados (cada visita destas seguida da visita a toda a subárvore direita de cada um destes nós).

```
⟨ Encontre uma classe de equivalência de peso máximo e imprima 24 ⟩ ≡
  { register node *p_max = Λ;
    register long pp = 0;
    if (root ≠ Λ) { register node *p, *q;
      p = root;
      root = Λ;
      while (1) {
        while (p→left ≠ Λ) {
          q = p→left;
          p→left = root;      /* usamos left para implementarmos a pilha */
          root = p;
          p = q;
        }
      visit: ⟨ Verifique o peso de p e atualize o máximo caso necessário 25 ⟩
        if (p→right ≡ Λ) {
          if (root ≡ Λ) break;      /* a pilha está vazia; terminamos */
          p = root;
          root = root→left;      /* desempilha! */
          goto visit;
        }
        else p = p→right;
      }
    }
    ⟨ Imprima a classe de p_max 26 ⟩;
  }
```

This code is used in section 5.

25.

```
#define peso(A) ((A→l) * (A→m))
⟨ Verifique o peso de p e atualize o máximo caso necessário 25 ⟩ ≡
  { register long w = peso(p);
    if (w > pp) { p_max = p; pp = w; }
  }
```

This code is used in section 24.

26.

```
⟨ Imprima a classe de p_max 26 ⟩ ≡
  if (p_max) { register link q = p_max→h;
    printf("[Uma␣classe␣de␣maior␣peso:␣%ld␣palavras␣de␣comprimento␣%ld␣(peso␣%ld)]\n",
        p_max→m, p_max→l, peso(p_max));
    while (q) {
      printf("␣␣%s\n", q→w);
      q = q→next;
    }
    printf("\nNumero␣total␣de␣classes␣que␣encontrei:␣%ld.\n", N_classes);
  }
```

This code is used in section 24.

27. Precisamos determinar os maiores pesos que obtivemos. Para tanto, usamos um *counting sort*, que nada mais é que a classificação de todas as classes de equivalência em "baldes" indexados pelos respectivos pesos. Supomos que o número máximo de classes que teremos é `MAX_NO_BALDES`.

```
#define MAX_NO_BALDES 500
```

```
⟨ Imprima as classes cujos pesos são os n_pesos maiores pesos 27 ⟩ ≡
  { node *bucket[MAX_NO_BALDES + 1];
    register node **p;
    printf("[Montei␣o␣treap;␣agora␣executarei␣o␣counting␣sort.]\n");
    fflush(stdout);      /* progress report */
    for (p = bucket; p ≤ bucket + MAX_NO_BALDES; p++) *p = Λ;
    ⟨ Percorra o treap e monte bucket 28 ⟩;
    ⟨ Imprima as classes nos maiores n_pesos baldes não-vazios 30 ⟩;
  }
```

This code is used in section 5.

28.

```
⟨ Percorra o treap e monte bucket 28 ⟩ ≡
  { if (root ≠ Λ) { register node *p, *q;
      p = root;
      root = Λ;
      while (1) {
        while (p→left ≠ Λ) {
          q = p→left;
          p→left = root;      /* usamos left para implementarmos a pilha */
          root = p;
          p = q;
        }
        visit2: ⟨ Coloque a classe de p no balde correspondente 29 ⟩;
        if (p→right ≡ Λ) {
          if (root ≡ Λ) break;      /* a pilha está vazia; terminamos */
          p = root;
          root = root→left;      /* desempilha! */
          goto visit2;
        }
        else p = p→right;
      }
    }
  }
```

This code is used in section 27.

29. Manteremos os objetos em cada balde em listas ligadas. Usamos o campo *left* para a lista ligada.

```
⟨ Coloque a classe de p no balde correspondente 29 ⟩ ≡
  { register long w = peso(p);
    if (w > MAX_NO_BALDES) {
      fprintf(stderr, "%s:␣Encontrei␣uma␣classe␣com␣peso␣excessivo!\n", *argv);
      return −7;
    }
    p→left = bucket[w];
    bucket[w] = p;
  }
```

This code is used in section 28.

30. ⟨ Imprima as classes nos maiores *n_pesos* baldes não-vazios 30 ⟩ ≡

```
{ register long i;
  register node **p = bucket + MAX_NO_BALDES;
  for (i = 0; i < n_pesos; i++) {
    while (¬*p ∧ --p > bucket) ;
    if (p > bucket) ⟨ Imprima as classes em *p e avance p 31 ⟩
    else break;
  }
}
```

This code is used in section 27.

31. ⟨ Imprima as classes em *p e avance p 31 ⟩ ≡

```
{ register node *pp = *p;
  while (pp) {
    register link q = pp→h;
    printf("[Peso␣%ld]␣%ld␣palavras␣de␣comprimento␣%ld␣(peso␣%ld)\n", i + 1, pp→m, pp→l,
        peso(pp));
    while (q) {
      printf("␣␣%s\n", q→w);
      q = q→next;
    }
    pp = pp→left;
  }
  p--;
}
```

This code is used in section 30.

32. Terminamos este programa com o seguinte comentário. Eu havia planejado escrever também um programa baseado na interface `ST.h` de Sedgewick [*Algorithms in C*, Third Edition, Parts 1–4]. Este é a interface para uma tabela de símbolos abstrata. Desta forma, poderíamos facilmente experimentar as várias implementações de tabelas de símbolos dadas naquele livro. Seria bastante interessante fazer um estudo empírico das várias possiblidades. Infelizmente, preciso voltar a outras tarefas e tal programa fica para outra ocasião.

Lembre que observamos acima que este programa ainda pode ser melhorado em termos de eficiência: poderíamos tirar proveito do fato de que, em geral, as buscas que fazemos em nosso treap é sem sucesso e que portanto queremos inserir elementos muito frequentemente.

33. Índice. Aqui está uma lista dos identificadores usados por `anagramas`; as seções onde eles aparecem são dadas (as entradas sublinhadas indicam os pontos de definição).

⟨ Canonize *buffer* 22 ⟩ Used in section 21.
⟨ Coloque a classe de p no balde correspondente 29 ⟩ Used in section 28.
⟨ Copie *buffer* na memória alocada para strings e faça $r\rightarrow$*keyword* apontar para esta cópia 14 ⟩ Used in section 13.
⟨ Encontre uma classe de equivalência de peso máximo e imprima 24 ⟩ Used in section 5.
⟨ Faça uma cópia de *buffer* na memória alocada para strings e canonize *buffer* "in loco" 21 ⟩ Used in section 18.
⟨ Faça r apontar para um novo nó e ponha *buffer* neste nó 13 ⟩ Used in section 10.
⟨ Imprima a classe de *p_max* 26 ⟩ Used in section 24.
⟨ Imprima as classes cujos pesos são os *n_pesos* maiores pesos 27 ⟩ Used in section 5.
⟨ Imprima as classes em $*p$ e avance p 31 ⟩ Used in section 30.
⟨ Imprima as classes nos maiores *n_pesos* baldes não-vazios 30 ⟩ Used in section 27.
⟨ Insira o nó correspondente a *buffer* no treap 10 ⟩ Used in section 18.
⟨ Insira *palavra_corrente* na lista de r 15 ⟩ Used in section 13.
⟨ Insira *palavra_corrente* na lista do *loco_encontrado* 20 ⟩ Used in section 18.
⟨ Interprete a linha de comando 6 ⟩ Used in section 5.
⟨ Leia em *buffer* a próxima palavra em *stdin*; **goto** *done* caso o arquivo terminou 23 ⟩ Used in section 18.
⟨ Percorra o treap e monte *bucket* 28 ⟩ Used in section 27.
⟨ Processe a lista de palavras 18 ⟩ Used in section 5.
⟨ Procure *buffer* no treap 19 ⟩ Used in section 18.
⟨ Quebre a subárvore p e pendure as duas árvores resultantes em r 12 ⟩ Used in section 10.
⟨ Typedefs 7, 9 ⟩ Used in section 5.
⟨ Variáveis locais 8, 11, 16, 17 ⟩ Used in section 5.
⟨ Verifique o peso de p e atualize o máximo caso necessário 25 ⟩ Used in section 24.

ANAGRAMAS