ARMANDO BARBOT CAMPOS MATOS

ALGORITMOS E ESTRUTURAS DE INFORMAÇÃO PARA REPRESENTAÇÃO DE CONJUNTOS

ALGORITMOS E ESTRUTURAS DE INFORMAÇÃO PARA REPRESENTAÇÃO DE CONJUNTOS

Trabalho apresentado nos termos da alínea b) do número 3 do artigo 8º do decreto-lei nº 388/70

ARMANDO BARBOT CAMPOS MATOS

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

1980

ÍNDICE

1. CONSIDERAÇÕES GERAIS

1.1. Introdução

Em cada fase de desenvolvimento de um algoritmo há um conjunto de operações e de estruturas de informação que são consideradas básicas, isto é executáveis directamente por uma máquina imaginária correspondente a esse nível. Este modo de encarar a estrutura e o desenvolvimento dos programas ([20], [24]), embora não único ([2], [13]) corresponde ao binómio instruções-objectos em que é baseada a grande maioria das linguagens de programação. Neste trabalho consideraremos alguns esquemas de representação com vista à realização de forma eficiente de operações com conjuntos. A questão de definir se a estrutura de conjunto deve aparecer ao nível do desenvolvimento do algoritmo ou ao nível da linguagem de programação (como em [18]) é secundária relativamente aos objectivos deste trabalho.

Dado o seu caracter fundamental pode parecer surpreendente que o conjunto não esteja incluido como estrutura primitiva da grande maioria das linguagens de programação; as causas deste facto podem talvez dividir-se em duas partes: (1) é relativamente dificil escolher um método eficiente de representação dos conjuntos que esteja razoavelmente próximo da linguagem máquina; (2) as linguagens de programação tradicionais foram concebidas de "baixo para cima" isto é pretendiam ser uma evolução no sentido linguagem máquina — facilidade de utilização, refletindo muitas vezes as particularidades do "hardware" subjacente (um exemplo desta situação é a linguagem FORTRAN).

É difícil dar uma ideia das aplicações possíveis da formulação dos algoritmos em termos de conjuntos; na realidade, a utilização de conjntos como es-

truturas de informação depende muitas vezes mais da metodologia do programador do que das características próprias dos algoritmos. Referimos apenas algumas aplicações típicas:

— tabelas de identificadores dos compiladores de linguagems de programação ([6]);

— representações dos grafos (dirigidos ou não);

— operações com bases de dados, nomeadamente no modelo relacional ([4]); por exemplo, a resposta a uma pergunta "quais os livros de Análise Numérica escritos em Inglês ou Francês e que usam o PASCAL como linguagem de programação?" envolve operações de reunião e intersecção de conjuntos.

1.2. Operações com conjuntos

Distinguimos dois tipos de operações com conjuntos:

— operações básicas que envolvem um conjunto S e um elemento x

— operações derivadas que envolvem dois conjuntos S1 e S2.

Consideramos as seguintes operações básicas:

pesquisa (x,S): pesquisa de um elemento x num conjunto S, isto é "$x \in S$?";

inserção (x,S): inserção de um elemento x num conjunto S, isto é "$S \leftarrow S \cup \{x\}$";

eliminação (x,S): o elemento x é retirado do conjunto S, isto é "$S \leftarrow S-\{x\}$";

processamento sequencial: a operação $x \leftarrow$ FIRST(S) fornece o primeiro elemento de S e a operação $x \leftarrow$ NEXT(S) fornece o elemento seguinte de S (numa

ordem arbitrária, sem repetições ou omissões); quando não houver mais elementos em S, x toma o valor especial Λ .

A uma estrutura que suporte as operações de pesquisa, inserção e eliminação chama-se dicionário ([1]); estas operações são muitas vezes combinadas no esquema seguinte:

```
FOUND ← pesquisa (x,S);
CASE operação OF
    pesquisa:;
    inserção: IF NOT FOUND THEN insira x em S;
    eliminação: IF FOUND THEN elimine x de S;
    END
```

A operação de processamento sequencial é importante em certas aplicações e pode ser utilizada na implementação das operações derivadas.

Consideramos os seguintes operações derivadas:

reunião de dois conjuntos: $S3 \leftarrow S1 \cup S2$;

intersecção de dois conjuntos: $S3 \leftarrow S1 \cap S2$;

diferença de dois conjuntos: $S3 \leftarrow S1 - S2$;

Várias outras operações poderiam ser consideradas, em particular quando se introduz uma relação de ordem total nos elementos do conjunto.

1.3. Caracterização da eficiência dos algoritmos

Designaremos por U o universo em consideração e por $q=|U|$ o número de

elementos do universo; a dimensão de um conjunto particular, S será representada por $N = |S|$.

A análise de um algoritmo para a implementação de uma operação op envolve dois aspectos importantes:

(1) a escolha de um modelo estatístico apropriado que traduza na medida do possível a situação nas aplicações concretas; por exemplo, ao inserir N elementos numa estrutura S é habitual supor que qualquer das N! permutações possíveis é igualmente provável; como um exemplo da importância do modelo estatístico utilizado consideremos a intersecção de S1 e S2 (subconjuntos de U) com, respectivamente N1 e N2 elementos; como em geral $N1 << q$, $N2 << q$ a intersecção $S1 \cap S2$ é, com grande probabilidade, vazia se S1 e S2 forem considerados subconjuntos aleatórios de U; como se observa, este modelo pode não reflectir a situação real ([16]);

(2) a escolha de parâmetros que traduzam a eficiência do algoritmo em termos de tempo de execução e de memória ocupada; seguiremos a prática comum de indicar a complexidade temporal (tempo de execução) em termos de ordens de grandeza ([10]); este método fornece uma informação incompleta que tem a vantagem de ser largamente independente do computador particular utilizado (máquina de acesso aleatário, RAM) e dos detalhes da programação.

Os parâmetros que utiliaaremos com mais frequência para descrever a complexidade temporal da operação op são: o tempo médio de execução e o tempo máximo de execução.

Como exemplo, se dissermos que o tempo máximo da pesquisa é $O(N^2)$ isto significa que a pesquisa de um elemento x no conjunto S com N elementos demora no máximo $O(N^2)$ tempo (isto é, é majorada por kN^2 para um certo k fixo).

1.4. Ocupação da memória

Quando realizamos uma operação com conjuntos, S3 ← S1 ⊗ S2 pode acontecer que os operandos S1 e S2 não sejam mais necessários após a operação; neste caso pode ser possível construir S3 a partir das estruturas S1 e S2 por forma a que tanto a eficiência temporal como espacial do algoritmo sejam melhoradas (um caso particular são os algoritmos de "UNION-MEMBER" para conjuntos disjuntos ([1]) ou não ([15])). Poderemos indicar da seguinte forma os conjuntos que não são mais necessários na continuação:

S3 ← S1 ⊗ S2, FREE(S1,S2).

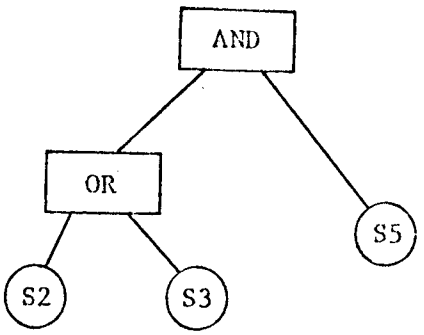
1.5. Operações simbólicas

Algumas vezes é mais eficiente não executar as operações entre conjuntos, deixando-as no conjunto resultado numa forma simbólica; consideremos por exemplo o seguinte programa:

```
1: S1 ← S2 ∪ S3;
2: S4 ← S1 ∩ S5;
3: B ← pesquisa (x,S4);
4: S7 ← S4 ∪ S6;
5: L1 ← pesquisa (x',S7);
```

se os conjuntos S2, S3, S5, S6 não forem muito pequenos é preferível, na operação de pesquisa na linha 3, representar S4 na forma simbólica (árvore AND-OR) da figura 1.a,

1.a



1.b

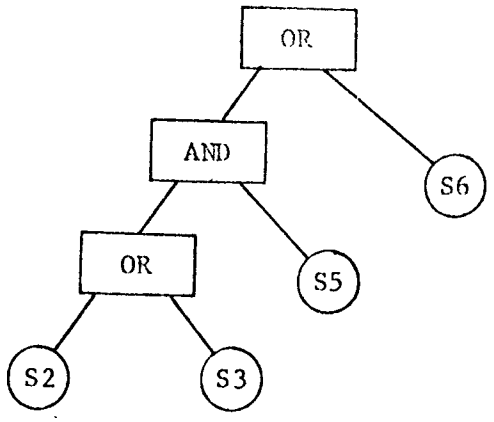


Fig. 1 Representações simbólicas de conjuntos resultantes de reuniões e intersecções.

e, na linha 5 representar S7 na forma da figura 1.b.

A reunião e a intersecção de conjuntos representados em árvores AND-OR é trivial e a pesquisa de um elemento pode implementar-se usando algoritmos recursivos apropriados.

É um problema interessante determinar numa sequência de reuniões, intersecções e pesquisas dadas à priori (OFF-LINE) quais as reuniões e intersecções que devem ser executadas e quais as que devem ser representadas em forma simbó-

lica.

A representação simbólica das operações entre conjuntos não será mais considerada na sequência deste trabalho.

1.6. Quando o universo é pequeno

Algumas vezes o universo U é relativamente pequeno (por exemplo se U for o conjunto das idades em anos de pessoas ou o conjunto dos países europeus); nestes casos é muitas vezes mais simples e eficiente representar cada conjunto $S \subseteq U$ por uma sequência de $q = |U|$ bits indicando a presença ou ausência dos elementos correspondentes de S; este método tem as seguintes vantagens principais:

— a eficiência dos algoritmos não depende da esparsidade do conjunto S mas apenas de q;

— é possível aproveitar a capacidade de processamento paralelo dos vários bits de uma palavra ou byte usando as instruções lógicas (e, ou, not, ou exclusivo)

— a implementação é muito simples estando já incorporada na linguagem de programação PASCAL (por exemplo na implementação da U.C.S.D. do PASCAL é possível usar universos com $q \leqslant 4096$ o que é suficiente para muitas aplicações).

A implementação das operações de reunião, intersecção, complementação e diferença de conjuntos é trivial a partir das operações lógicas de, respectivamente, ou, e, negação, negação do segundo operando seguida de e entre os operandos (diferença lógica).

As operações de pesquisa, inserção e eliminação podem ser implementadas determinando primeiro a palavra (ou byte) contendo o bit associado ao elemento

x em questão, e, em seguida, operando-a com a "máscara" m_x representativa do bit associado a x nessa palavra (utilizando as operações de e e comparação com zero para a pesquisa, ou para a inserção, e com a negação de m_x para a eliminação).

Para representar subconjuntos de um universo com q bits necessitamos de $\lceil q/n_b \rceil$ palavras ou bytes em que n_b é o número de bits por palavra ou byte; cada operação de reunião, intersecção, complementação ou diferença traduz-se em $\lceil q/n_b \rceil$ operações lógicas; desta forma tanto a complexidade temporal como espacial são O(q) (nota: $\lfloor x \rfloor$ representa o maior inteiro não superior a x e $\lceil x \rceil$ o menor inteiro não inferior a x).

1.7. Quando o universo é grande

A maior parte das vezes q é demasiado grande para que se possa usar o método que acabamos de descrever; por exemplo, se o universo for o conjunto dos identificadores de não mais de 8 símbolos (letras ou dígitos) em que o primeiro é uma letra temos que $q = 26\ (1 + 36 + 36^2 + \ldots + 36^7) \approx 209.6$ E+10 bits necessários para representar um conjunto o que excede a memória central de qualquer computador actual; mesmo que haja memória suficiente, o método descrito torna-se muito ineficiente quando q é elevado.

Todos os métodos que referiremos na sequência deste trabalho pressupõem que o número N de elementos de cada conjunto representado é relativamente pequeno por forma a caberem na memória central e que a dimensão do universo é muito maior que N.

Descreveremos sumariamente os seguintes tipos de estruturas para representação de conjuntos:

— estruturas elementares

— árvores
  - árvores binárias
  - árvores equilibradas
  - árvores de pesquisa digital

— esquemas de "hashing"

As estruturas para representação de conjuntos que consideraremos são geralmente derivadas de estruturas utilizadas em algoritmos de pesquisa, inserção e eliminação; a referência [11] contém uma análise bastante completa e cuidada deste tipo de estruturas.

Descreveremos dois métodos particulares de implementação da estrutura de conjunto baseados respectivamente nas árvores 3-2 e num esquema de hashing.

2. ESTRUTURAS ELEMENTARES

A maior parte dos programadores perante um problema de reunião ou intersecção utilizaria provavelmente uns dois esquemas gerais descritos em seguida.

2.1. Sequência arbitrária

Cada conjunto é representado por uma tabela em que os seus elementos aparecem sequencialmente sem repetições; por exemplo os conjuntos:

S1 = {Rui, Francisco, Teresa, João, José}

S2 = {Teresa, Ana, João}

podem ser representados na forma indicada na figura 2.

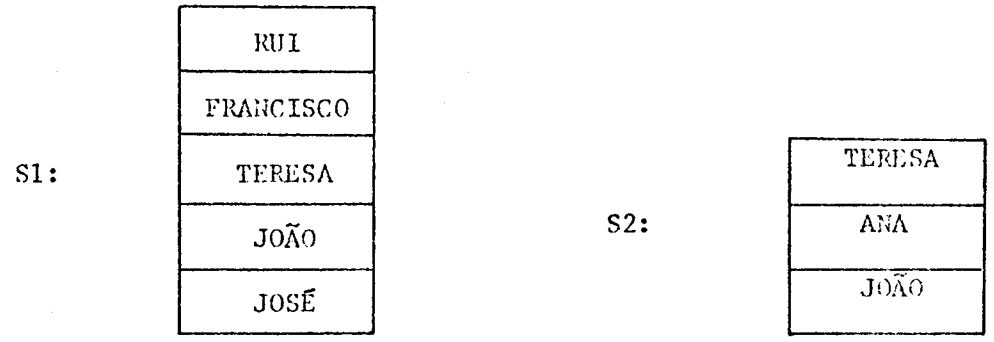


Fig. 2 Representação dos conjuntos S1 e S2 como sequências arbitrárias.

Os seguintes factos são de verificação elementar:

— a pesquisa (sequencial) é de complexidade O(N)

— a inserção (após a pesquisa) é de complexidade O(1)

— a eliminação (após a pesquisa) é de complexidade O(1) (colocando o último elemento no espaço deixado livre)

— a reunião e intersecção são de complexidade $O(N^2)$ pois podem ser obtidas (por exemplo) com, respectivamente, |S1| + |S2| pesquisas no conjunto em formação S1 U S2, e |S1| pesquisas em S2.

2.2. Sequência ordenada

Cada conjunto é representado pela sequência ordenada dos seus elementos usando uma relação de ordem total apropriada; por exemplo, com a ordem alfabé-

tica, a representação dos conjuntos S1 e S2 está indicada na figura 3.

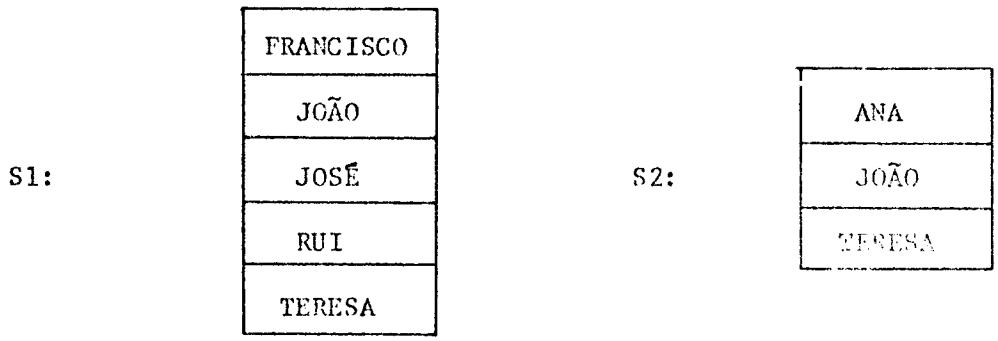


Fig. 3 Representação ordenada dos conjuntos S1 e S2 como sequências ordenadas.

Quanto à eficiência desta representação é fácil estabelecer o seguinte:

— a pesquisa é de complexidade O(ln N) se utilizarmos o algoritmo de pesquisa binária

— a inserção e a eliminação são de complexidade O(N) pois exigem a translação de O(N) elementos na tabela

— a reunião e intersecção são muito eficientes (complexidade O(N)) usando um algoritmo que faz o varrimento sequencial de ambos os conjuntos; por exemplo, para a intersecção (quais os nomes comuns às listas telefónicas do Porto e de Lisboa?) podemos usar o seguinte algoritmo:

```
S3 ← { };
X1 ← FIRST(S1);
X2 ← FIRST(S2);
WHILE (X1 ≠ Λ ) AND (X2 ≠ Λ ) DO
   BEGIN
      IF X1 < X2 THEN X1 ← NEXT(S1) ELSE
      IF X2 < X1 THEN X2 ← NEXT(S2) ELSE
        BEGIN
           S3 ← S3 U {X1};
           X1 ← NEXT(S1); X2← NEXT(S2)
        END
   END;
```

observe-se que a inserção S3← S3 U {X1} é de complexidade O(1) dado que a sequência dos elementos inseridos é crescente (admitindo que o processamento sequencial é correspondente à relação de ordem).

Os dois métodos referidos (sequência arbitrária e sequência ordenada) estão comparados na tabela 1 (os tempos médios e mais desfavoráveis são da mesma ordem de grandeza neste caso).

A sequência ordenada é preferível ([21]) quando a operação predominante é a pesquisa; um método muito usado consiste em combinar uma pequena sequência arbitrária S1 onde se fazem as inserções com uma sequência ordenada S2; de tempos a tempos S1 é ordenada (se o algoritmo for eficiente em tempo O(N1 ln N1), N1 = |S1|) e reunida a S2.

É possível operar com representações mistas; por exemplo para intersectar S1 (com N1 elementos) representado como sequência arbitrária e S2 (com N2 elementos) representado como sequência ordenada podemos fazer N1 pesquisas em S2,

resultando uma complexidade temporal de O(N1 ln N2).

TABELA 1

Comparação das ordens de grandeza dos tempos de execução para as estruturas elementares

	Sequência arbitrária	Sequência ordenada
pesquisa	O(N)	O(ln N)
pesquisa + inserção	O(N)	O(N)
inserção	O(1)	O(N)
pesquisa + eliminação	O(N)	O(N)
eliminação	O(1)	O(N)
reunião e intersecção	$O(N^2)$	O(N)

3. MÉTODOS BASEADOS EM ÁRVORES

3.1. Introdução

Os dois métodos elementares referidos são ineficientes no sentido em que requerem tempo O(N) para a realização de pelo menos uma operação dos dicionários; consideraremos agora a utilização de árvores para representar conjuntos de uma forma mais eficiente. As árvores binárias simples permitem a realização das operações da estrutura de dicionário em tempo médio O(ln N) mas no caso mais desfavorável esse tempo pode ser O(N); nas árvores equilibradas os tempos são

O(ln N) mesmo no caso mais desfavorável; nas árvores de pesquisa digital cada elemento é representado como uma sequência de símbolos elementares (letras ou dígitos, por exemplo) o que permite um grau de ramificação maior e consequentemente uma maior eficiência.

Dado que existem algumas variantes nas definições dadas na literatura relacionadas com árvores, indicamos em seguida o significado de alguns conceitos que utilizaremos:

— uma árvore ou é nula ou consiste num nó r (raíz) e k ≥ 0 árvores (também chamadas de sub-árvores ou árvores descendentes); r é o antecessor imediato (ou pai) dos nós que são raízes das k' ≤ k sub-árvores não nulas, os quais se dizem filhos de r; a k' (número de sub-árvores não nulas) chama-se ramificação de r; os nós com ramificação zero chamam-se folhas ou nós terminais; cada árvore nula pode ser representada por Λ ou por um nó ficticio dito externo.

— o nível da raíz é 1 e o nível da qualquer outro nó é 1 + o nível do seu antecessor imediato; a altura de uma árvore é o máximo nível dos seus nós (o comprimento do maior caminho de um nó terminal da árvore até à raíz + 1).

3.2. Árvores binárias

São árvores em que cada nó tem duas sub-árvores (e portanto cuja ramificação pode ser 0, 1 ou 2). Podemos usar a seguinte estrutura para a representação das árvores binárias:

```
TYPE ABINARIA = RECORD
                    CONT:OBJECTO;
                    AESQ, ADIR: ↑ABINARIA
                    END
```

Na representação habitual dos conjuntos como árvores binárias utiliza-se uma relação de ordem total no universo U por forma que, sendo o conjunto S representado numa árvore binária, temos:

— cada $x \in S$ está representado (directamente ou através de um apontador) no campo CONT de um nó

— se x está num nó da AESQ (sub-árvore esquerda) de uma sub-árvore cuja raíz é y e se z está num nó da ADIR dessa árvore então $x < y < z$.

A definição recursiva de árvore binária presta-se à implementação de algoritmos recursivos para diversas operações; por outro lado em certas linguagens é difícil ou ineficiente utilizar os mecanismos de recursividade pelo que é necessário modificar os algoritmos para que não haja chamadas recursivas.

Não descreveremos as versões recursivas e não recursivas dos algoritmos de pesquisa, inserção e eliminação; mencionamos apenas o seguinte resultado:

— em árvores binárias construídas por inserção de uma sequência aleatória de N elementos (todas as N! sequências igualmente prováveis) as operações de pesquisa, inserção e eliminação requerem um tempo médio de ordem O(ln N) e, no caso mais desfavorável, de O(N).

O caso mais desfavorável ocorre, por exemplo, quando os elementos são inseridos por ordem crescente, resultando uma árvore de altura N em que todas os nós (excepto o terminal) têm apenas a sub-árvore direita não nula.

As operações com conjuntos (reunião, intersecção) requerem o processamento sequencial para o qual consideraremos três métodos distintos, todos eles respeitando a relação de ordem definida entre os elementos do conjunto.

1) Visita em ordem simétrica por um algoritmo recursivo

O processamento sequencial crescente corresponde à vista da árvore por ordem simétrica, isto é, utilizando o seguinte algoritmo recursivo a que chamamos VISITA:

a) visita da sub-árvore esquerda (se existir)

b) visita da raíz

c) visita da sub-árvore direita (se existir)

Ao visitar um nó particular x são executadas as ações correspondentes; com esta estrutura o algoritmo de processamento sequencial é externo ao algoritmo "principal".

Outra alternativa será definir as funções FIRST e NEXT (por forma a que o processamento sequencial seja interno ao algoritmo principal) com base em chamadas a um "procedure" do tipo visita que permitisse, logo após uma visita a um nó x, uma suspensão de execução e retorno ao algoritmo principal; quando a função NEXT fosse novamente invocada a visita prosseguia a partir do estado (definido pelo mecanismo de recursividade) na altura da suspensão (tal "procedure" podia ser designado por co-rotina recursiva).

É fácil verificar que o número total de entradas e saídas recursivas de VISITA é o dobro do número de ramos na árvore, ou seja 2(N-1) sendo N o número de elementos do conjunto representado. Assim o tempo médio de acesso ao nó seguinte no processamento sequencial é para qualquer árvore de O(1), podendo ser O(N) para certos nós em árvores degeneradas.

Para árvores binárias equilibradas (ver 3.3) o caso mais desfavorável é de complexidade O(ln N).

2) Em [10] descreve-se um método que permite o processamento sequencial

sem utilização de recursividade ou de "stacks" auxiliares; basicamente em cada nó x da árvore:

(i) se a árvore esquerda é nula, então utiliza-se o apontador respectivo (AESQ) para indicar o nó predecessor de x em ordem simétrica

(ii) se a árvore direita é nula, então o apontador respectivo (ADIR) indica o nó sucessor de x em ordem simétrica

É necessário um bit adicional de informação em cada apontador para indicar se ele define uma subárvore (+) ou um nó sucessor ou predecessor (-).

Na figura 4 os apontadores que definem subárvores são indicados a cheio e os outros a tracejado.

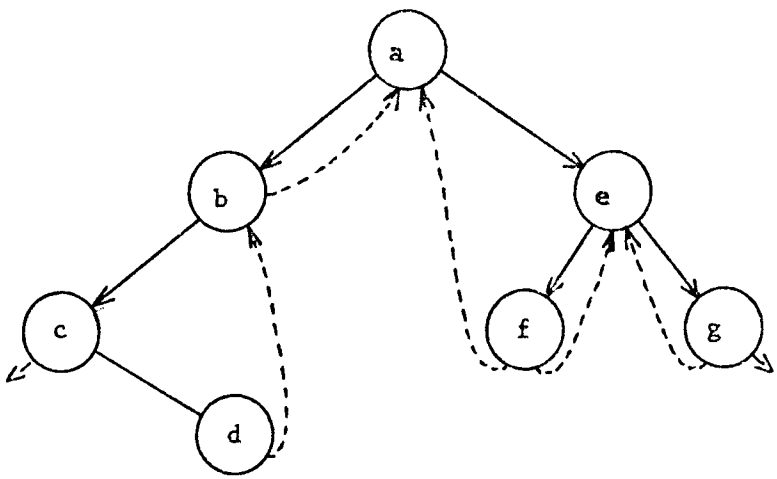


Fig. 4 Um método de implementação do processamento sequencial

A eficiência deste método é semelhante à do anterior mas a memória necessária é menor.

Nas operações de inserção e eliminação não é difícil manipular este tipo de apontadores.

3) Com a utilização de dois apontadores adicionais por nó é possível definir uma lista duplamente ligada dos elementos do conjunto com a qual é elementar realizar o processamento sequencial em tempo máximo O(1), sem degradar os tempos das operações de inserção e eliminação.

Este é o método mais eficiente e geral (aplicável a outros métodos de representação) tendo a desvantagem de requerer alguma memória adicional.

3.3. Árvores equilibradas

Existem vários métodos de evitar que as árvores construídas por inserções e eliminações de elementos degenerem (isto é que tenham uma altura de ordem O(N)); na maior parte desses métodos garante-se que a altura da árvore é sempre de ordem O(ln N) (mesmo no caso mais desfavorável); os principais esquemas utilizados para manter as árvores equilibradas quando há operações arbitrárias de inserção e eliminação baseiam-se nos seguintes tipos de árvores ([9],[15],[22]):

— árvores AVL (denominadas com base nos seus autores, Adel'son Vel'skii e Landis): são árvores binárias em que, para nenhum nó, a altura da sub-árvore esquerda (h_e) difere da altura da sub-árvore direita de mais de uma unidade, isto é

$$|h_e - h_d| \leq 1$$

— árvores HB [k] : trata-se de uma generalização das árvores AVL para as quais a diferença de alturas das duas sub-árvores de qualquer nó não excede k:

$$|h_e - h_d| \leq k,\ k \geq 1$$

— árvores de equilíbrio unilateral designadas por OSHB: são árvores AVL em que a sub-árvore esquerda nunca tem altura superior à sub-árvore direita (para qualquer nó):

$$0 \leq h_d - h_e \leq 1$$

— árvores de irmãos, designadas por B_rT: são árvores binárias em que todos as folhas estão ao mesmo nível e em que todo o nó só com um filho tem um irmão com dois filhos

— árvores de irmãos à direita (RB_rT): são árvores de irmãos em que todo o nó só com um filho tem um irmão à direita com dois filhos

— árvores 3-2 (3-2 T): são árvores em que todas as folhas estão ao mesmo nível e, em que cada nó que não é uma folha pode conter ou

- um valor k e duas sub-árvores não nulas (com elementos menores ou iguais a k e superiores a k) ou

- dois valores $k1 < k2$ e três sub-árvores não nulas (com elementos respectivamente menores ou iguais a k1, maiores que k1 e menores ou iguais a k2, e superiores a k2)

— árvores do tipo B de ordem m (BT_m): são árvores que satisfazem às seguintes condições:

- todos os nós têm $\leq m$ sub-árvores não nulas

- todos os nós excepto as raízes e as folhas têm $\geq m/2$ sub-árvores não nulas

- todas as folhas estão ao mesmo nível

- a raíz tem pelo menos duas sub-árvores

- todo o nó que não é folha e tem k sub-árvores contem k-1 valores $V_1, V_2, \ldots, V_{k-1}$ que definem a partição correspondente às sub-árvores(definindo

$V_0 = -\infty$, $V_k = +\infty$, a sub-árvore i contém valores x satisfazendo $V_{i-1} < x \leq V_i$, para i = 1, 2, ..., k). Na figura 5 estão representados exemplos de árvores equilibradas de vários tipos.

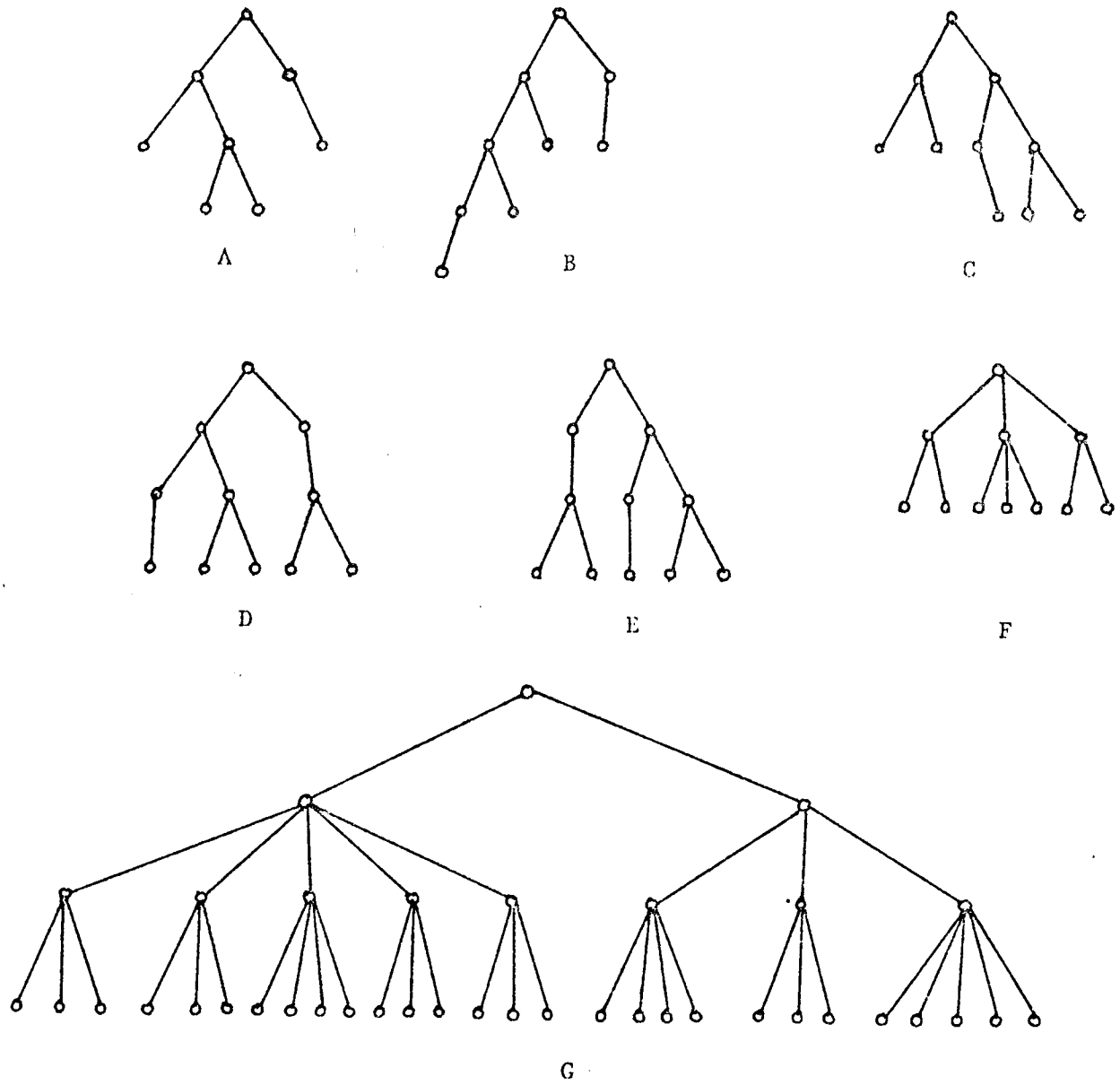


Fig. 5 Exemplos de árvores equilibradas: A: AVL, B: EB[2], C: OSMB, D: $B_r T$, E: $RB_r T$, F: 3-2 T, G: ET_5

É claro que estes tipos de árvores não são disjuntos; é muito fácil estabelecer as seguintes relações a partir das definições:

$$C(OSHB) \subseteq C(AVL) \subseteq C(HB[k])$$

$$C(RB_r T) \subseteq C(B_r T)$$

$$C(3\text{-}2\ T) = C(BT_3)$$

onde C(T) representa o conjunto de todas as árvores do tipo T.

O interesse destes tipos de árvores resulta das seguintes propriedades fundamentais:

1) Qualquer dos tipos referidos tem uma altura de ordem O(ln N) (em que N é o número de nós na árvore) o que permite que a operação de pesquisa seja, mesmo no caso mais desfavorável de complexidade O(ln N);

2) As operações de inserção e eliminação podem ser efectuadas em tempo O(ln N) mantendo a invariância do tipo de árvores; estes algoritmos podem ser esquematizados segundo a figura 6; a operação op (de inserção ou eliminação) pode alterar a classe T a que pertencia a árvore; se isso acontecer são necessários uma ou mais operações de correcção c para manter o tipo de árvore original.

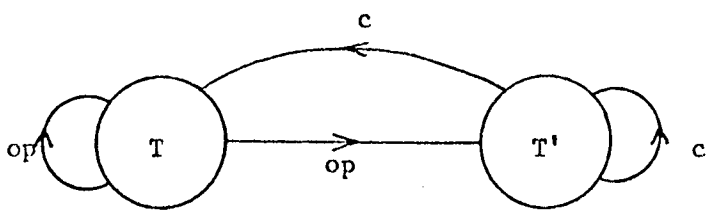


Fig. 6 Esquema geral das operações de inserção e eliminação em árvores equilibradas

Todos os tipos de árvores apresentados permitem algoritmos de complexidade O(ln N) para as três operações básicas: pesquisa, inserção e eliminação; só recentemente ([25]) foi descoberto um algoritmo para a inserção de nós em árvores de equilibrio unilateral com eficiência O(ln N). As operações de reunião e intersecção implementadas a partir das operações básicas são de complexidade O(N ln N); neste trabalho apresentaremos dois sistemas de manipulação de conjuntos em que as operações derivadas são realizadas de forma mais eficiente (O(N)).

3.4. Árvores de pesquisa digital

Quando os elementos do universo podem ser codificados por uma sequência de símbolos definidora da relação de ordem respectiva, é possível utilizar os métodos de pesquisa digital para implementar as operações básicas e derivadas; como exemplo de sequências de símbolos temos:

(1) representação alfabética dos elementos

(2) representação, numa certa base, dos valores associados aos objectos.

As árvores de pesquisa digital têm um grau de ramificação k igual ao número de símbolos do alfabeto; mudando a base de representação é possível ajustar o grau de ramificação.

Devido às limitações de espaço não nos é possível desenvolver estes métodos, remetendo o leitor para [11], capítulo 6.

4. MÉTODOS DE "HASHING"

Os métodos de "hashing" são muito utilizados na implementação eficiente de estruturas suportando as operações de pesquisa e inserção de elementos, nomeadamente em tabelas de identificadores de "assemblers" e compiladores, em esquemas de acesso a ficheiros, etc.. Começaremos por resumir os princípios destes métodos e depois faremos alguns comentários à sua aplicação aos conjuntos.

A cada elemento x do universo U a função de "hashing" h(x) associa um endereço entre, digamos, 0 e d-1 do espaço de hashing H:

$$U \longrightarrow H$$

$$x \longrightarrow h(x)$$

A função h deve procurar distribuir as imagens dos elementos $x \in U$ de uma forma uniforme no espaço H mesmo quando (tal como acontece com frequência) os objectos x estão distribuídos de uma forma altamente não uniforme em U; as funções de "hashing" mais utilizadas são baseadas em métodos de divisão, multiplicação e (para identificadores) de soma binária sem transporte (ou exclusivo) dos códigos dos caracteres ([11],[14]). O valor $d = |H|$ deve ser escolhido por forma a que o espaço H seja representável na memória central (ou na memória total nos métodos externos) e tenha dimensão suficiente para conter o maior número de elementos de U previstos.

Dado um elemento $x \in U$ a inserir na tabela H, determina-se a sua imagem h(x) e examina-se essa posição de H; se está livre então x não pertence ao subconjunto representado podendo ser inserido nessa posição; se a posição está ocupada por x, então x já está na tabela; se a posição já está ocupada por $y \neq x$ a pesquisa de x na tabela prossegue segundo o método adoptado de resolução de colisões.

Uma colisão é um par de elementos x_1, x_2 com a mesma imagem, $h(x_1)=h(x_2)$; se a imagem do elemento x a inserir na tabela, h(x) está ocupada pelo elemento y há vários processos possíveis de resolver a dificuldade:

(1) formação de d listas externas a H contendo todos os elementos com a correspondente imagem de H;

(2) após percorrer a lista interna a partir de h(x) e verificar que x não está na tabela, procura-se uma posição livre de H onde se insere x que passa a ser apontada pelo último elemento de cadeia pesquisado

(3) endereçamento aberto: procura-se uma posição livre segundo uma sequência determinada que abranja todo o espaço H: h(x), $h_1(x)$, $h_2(x)$, ... onde se insere x; nos casos mais simples $h_i(x) = h(x) + i \pmod d$ ou $h_i(x) = h(x)+p.i \pmod d$ onde p e d são primos entre si.

Nos casos (1) e (2) cada nó de H deve incluir um apontador para definir o elemento seguinte da lista. Existem variações em torno destes métodos ([3],[7]).

As principais características dos métodos de "hashing" são:

(1) o tempo médio de pesquisa ou inserção embora dependente do método de resolução de colisões adoptado é de ordem O(1); trata-se pois de métodos extremamente eficientes em termos médios

(2) os tempos máximos de pesquisa ou inserção são muito maus (O(N)), ocorrendo quando todos os elementos inseridos têm a mesma imagem

(3) as tabelas de "hashing" não são fácilmente expansíveis; quando há overflow é necessário aumentar d e reestruturar toda a tabela (ver contudo [5])

(4) a eliminação de um elemento e o processamento sequencial dos elementos representados (em particular segundo uma relação de ordem) causam problemas

relativamente delicados nos métodos de "hashing".

Quando se aplicam os métodos de hashing à representação de conjuntos, além dos problemas referidos em (4) é necessário fazer coexistir todos os conjuntos representados no mesmo espaço de "hashing" (a fixação de um espaço H para cada conjunto é muito ineficiente em espaço devido ao facto referido em (3)).

Neste trabalho (capítulo 7) descrevemos um sistema particular de manipulação de conjuntos baseado num método de "hashing" onde se explica o modo como foram resolvidos os vários problemas referidos.

5. OUTROS PROBLEMAS

Neste trabalho além de apenas abordarmos os vários métodos de representação dos conjuntos de uma forma resumida, somos forçados a não incluir o desenvolvimento de várias questões importantes, em especial:

1) representação de conjuntos associados a relações entre os seus elementos, em particular relações de ordem parcial;

2) representação de multiconjuntos ([23]) e de conjuntos estruturados

3) aplicação dos métodos à representação dos conjuntos em memórias externas (discos); neste caso os critérios de opção entre algoritmos têm mais a ver com o número de acessos à memória externa do que com o número de operações elementares

4) estudos de optimização da sequência de operações entre conjuntos numa expressão ou programa.

6. SISTEMA DE MANIPULAÇÃO DE CONJUNTOS BASEADO EM ÁRVORES 3-2

6.1. Introdução

Foi implementado um conjunto de algoritmos para manipulação eficiente de conjuntos com bases nas árvores 3-2. Basicamente, para as operações de pesquisa, inserção e eliminação a complexidade temporal é O(ln N) e para as operações de reunião, intersecção e diferença a eficiência é O(N), onde N representa a dimensão dos conjuntos. Estas ordens de grandeza são válidas mesmo no caso mais desfavorável, situação em que o método apresentado é vantajoso relativamente a outros métodos (árvores binárias e métodos de "hashing").

6.2. Descrição das estruturas de informação

A base para a construção das árvores 3-2 é o NO32 que podemos esquematizar da seguinte forma:

```
NO32 = RECORD
          TIPO: (NO3, NO2, TERMINAL, BASE);
          K1, K2: CONTEUDO;
          P1, P2, P3: ↑NO32
       END
```

Nos nós dos tipos NO3 e TERMINAL apenas utilizaremos um conteúdo (K2) e dois apontadores (P2 e P3) e nos nós do tipo BASE apenas utilizaremos os apontadores (P1, P2 e P3).

Os tipos de nós utilizados são:

TERMINAL - são os nós da árvore 3-2 sem descendentes onde estão representados (em K2) os elementos do conjunto respectivo; P2 e P3 são usados para formar uma lista de nós terminais duplamente ligada onde os elementos do conjunto aparecem por ordem crescente no sentido definido por P2; o nó BASE faz também parte desta lista.

NO2 - são nós da árvore 3-2 não terminais com dois descendentes apontados por P2 (à esquerda) e P3 (à direita); K2 é o maior elemento terminal da sub-árvore apontada por P2.

NO3 - são nós da árvore 3-2 não terminais com três descendentes apontados por P1 (à esquerda), P2 (ao centro) e P3 (à direita); K1 é o maior elemento terminal da sub-árvore apontada por P1 (esquerda) e K2 é o maior elemento terminal da sub-árvore apontada por P2 (ao centro).

BASE - há um nó deste tipo por conjunto; P2 aponta para o primeiro e P3 para o último nó da cadeia de nós terminais; P1 aponta para a raíz da árvore; todas as referências ao conjunto são feitas por apontadores para o nó de base respectivo ; K2 do nó de base poderá conter o nome do conjunto representado.

Na figura 7 está representada uma árvore 3-2 (que não é única) correspondente ao conjunto:

$$S1 = \{ \text{JOSE, ANA, JOAO, RUI, SUSANA} \}$$

Observe-se que todos os nós terminais estão ao mesmo nível, ordenados por ordem crescente.

É claro que os conteúdos dos nós (K1 e K2) podem ser apontadores para descrições dos elementos o que permite em muitos casos (principalmente quando os el

mentos são comuns a mais que um conjunto) poupar memória; por outro lado a estrutura de NO32 podia ser representada num "RECORD" com variantes de modo a só reservar espaço para os componentes usados por cada tipo de nó.

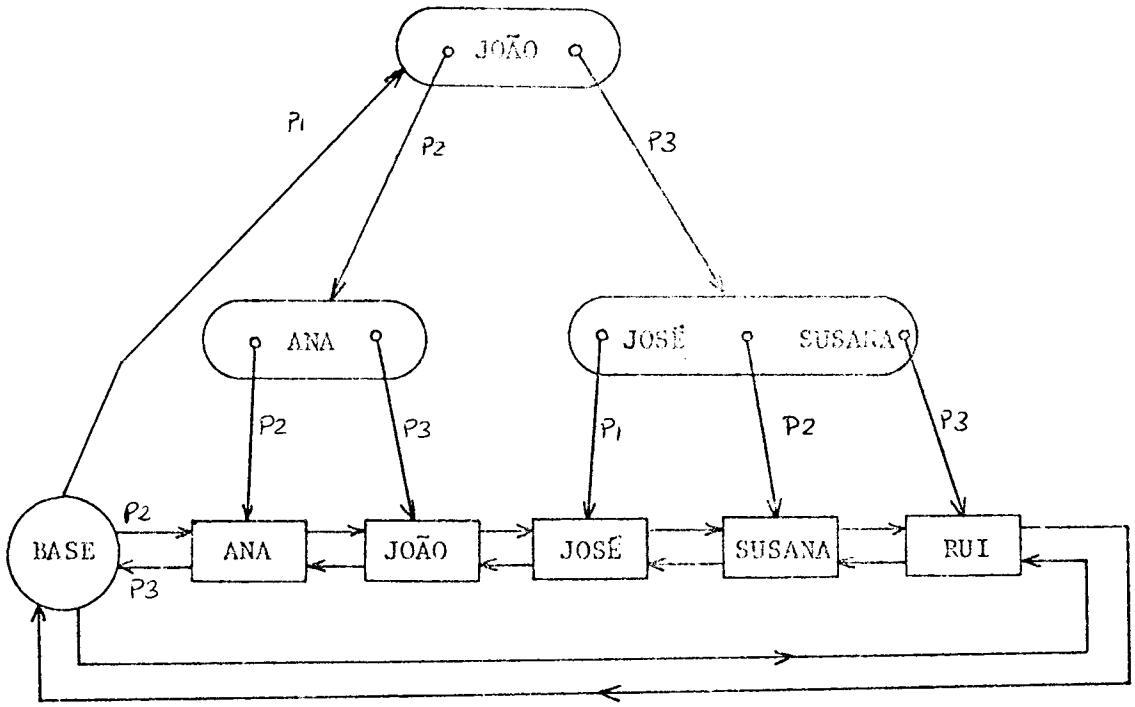


Fig. 7 Representação do conjunto S1 em árvore 3-2.

6.3. Descrição das operações básicas

A operação de pesquisa é simples; seja x o elemento a pesquisar no conjunto representado pelo nó base S; podemos usar o seguinte algoritmo:

```
T ← S.P1;
IF T = Λ THEN FOUND ← FALSE ELSE BEGIN
  WHILE T↑. TIPO≠TERMINAL DO CASE T↑.TIPO OF:
    NO2: IF X <= T↑.K2 THEN T ← T↑ .P2 ELSE T ← T↑.P3;
    NO3: IF X <= T↑.K1 THEN T ← T↑ .P1 ELSE
         IF X <= T↑.K2 THEN T ← T↑ .P2 ELSE T←T↑ .P3
    END
  FOUND←X=T↑ .K2
END;
```

É conveniente durante a pesquisa ir formando uma lista dos nós visitados da árvore (caminho da raíz a um nó terminal) que representaremos por A[1],...; estes nós serão utilizados nas operações de inserção e eliminação. Se X não pertence a S o nó terminal encontrado a que chamamos P é o menor nó imediatamente a seguir a X (excepção: quando X é maior que qualquer elemento de S).

Consideramos agora o problema da inserção (que é um pouco mais complexo); inicialmente o nó X é inserido na posição apropriada de dupla cadeia de nós terminais; em seguida o objectivo é substituir o nó P da árvore (encontrado na fase de pesquisa) pelos nós P e X; seja N o antecessor de P; se N é do tipo NO2 podemos ter os casos das figuras 8.a e 8.b e o caso simétrico de 8.b (X à direita); o nó N passa a ser do tipo 3 e o algoritmo termina. Se o nó N é do tipo NO3 então construímos dois nós do tipo NO2 (figura 8.c); estes novos nós que designamos por P' e X' estão na mesma situação para o antecessor N' de N em que estavam P e X para N; assim o algoritmo prossegue até que N seja do tipo NO2 ou até que N seja a raíz; neste último caso N é desdobrado em dois nós do tipo NO2 e a altura da árvore aumenta de uma unidade. Por exemplo se no conjunto S1 representado atrás inserirmos o nome "LUIS" obtemos a árvore representada na figura 9.

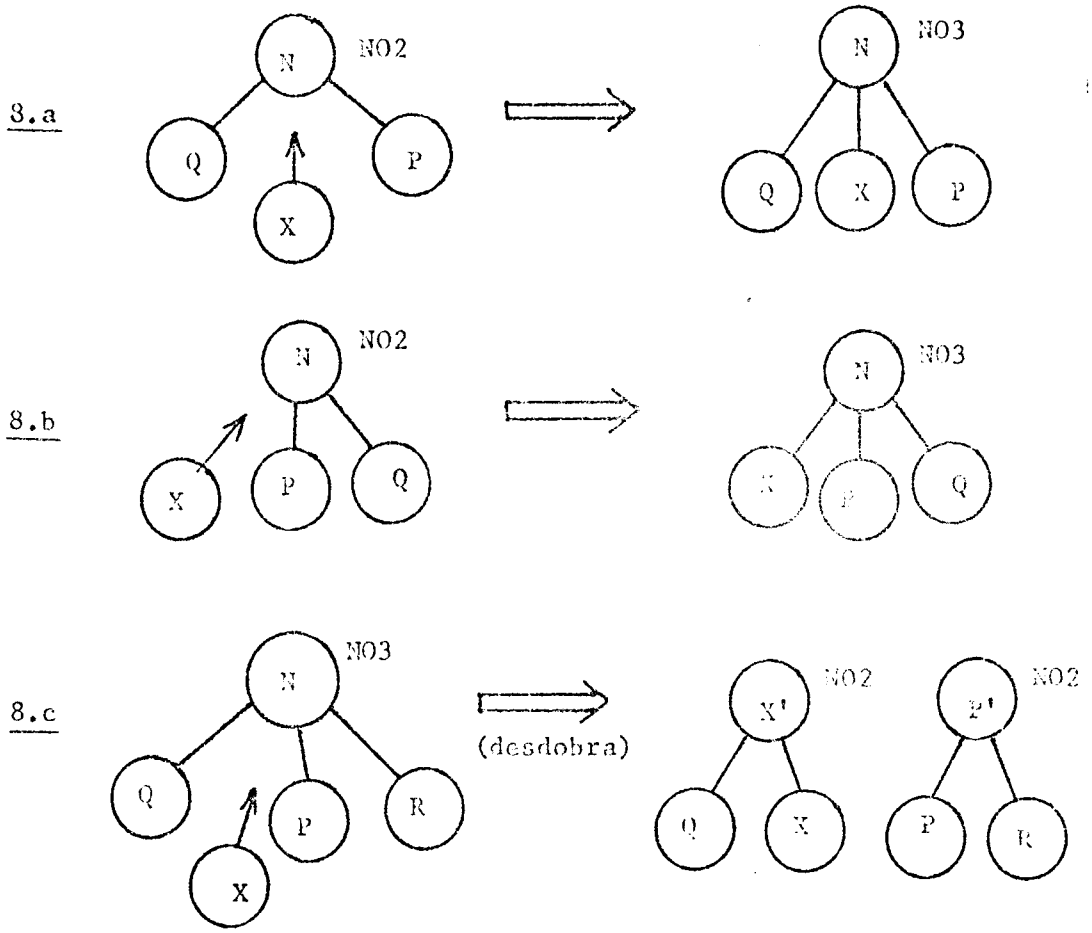


Fig. 8 Casos na inserção do nó x.

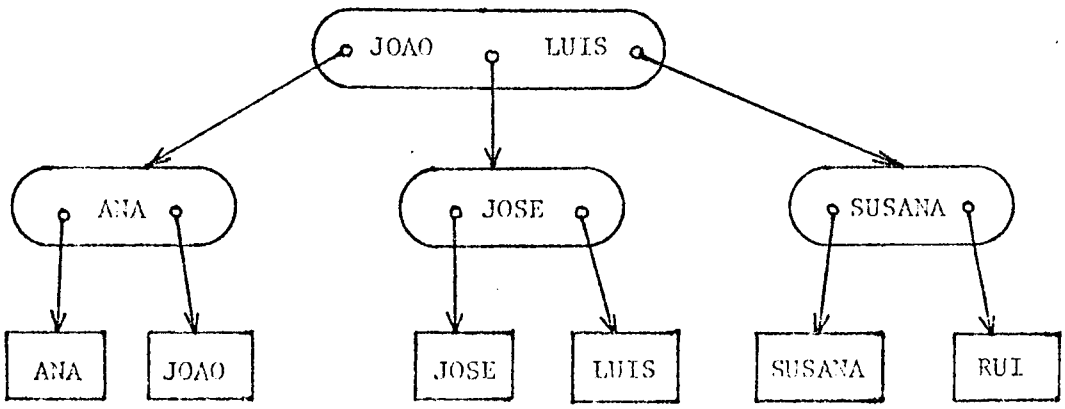


Fig. 9 Árvore 3-2 após a inserção do nome "LUIS".

Consideremos agora o problema da eliminação de um nó P de uma árvore 3-2; após retirar o nó P da dupla cadeia de nós terminais, consideremos o nó N, antecessor de P na árvore 3-2; se N é do tipo NO3 então passa a ser do tipo NO2, como no caso representado na figura 10.a, e o algoritmo termina depois de actualizar, se necessário (se X era o nó direito de N) o conteúdo de um antecessor de N; se N é do tipo NO2, retiramos P de N e consideramos N e o seu único descendente na aplicação do resto do algoritmo (figura 10.b)

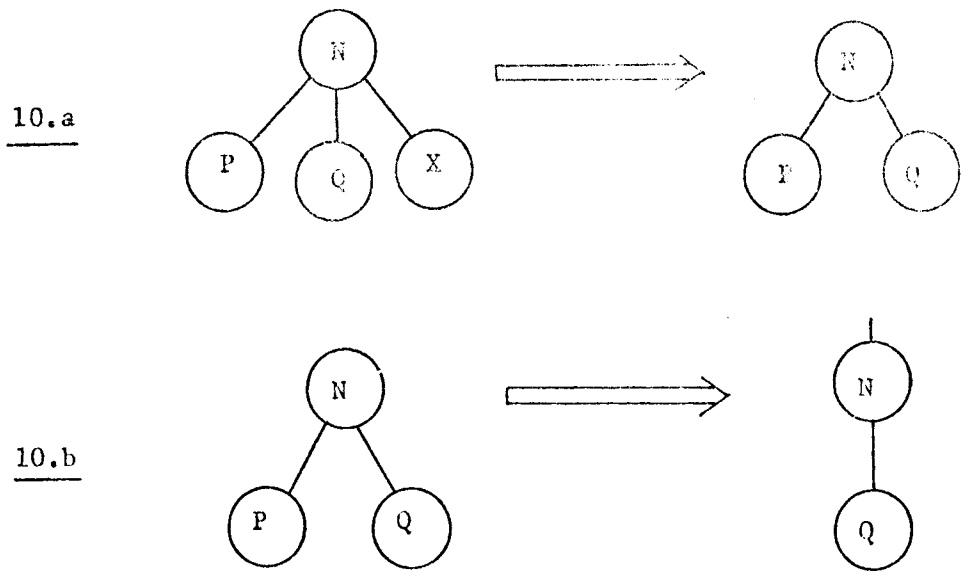


Fig. 10 Casos iniciais de aplicação do algoritmo de eliminação do nó P.

Seja N' o antecessor de N na árvore 3-2; temos a distinguir os seguintes casos:

Caso 1: N' é do tipo NO2 ou NO3 e tem um filho F contíguo a N do tipo NO3; a transformação é do tipo indicado na figura 11.a; o algoritmo termina (actualizando se necessário o conteúdo de um antecessor de N').

Caso 2: N' é do tipo NO3 e tem um filho F (irmão de N) do tipo NO2; a transformação é do tipo representado na figura 11.b; o algoritmo termina (actualizando se necessário o conteúdo de um antecessor de N'); o nó N' passa a ser do tipo NO2 e o nó N é libertado.

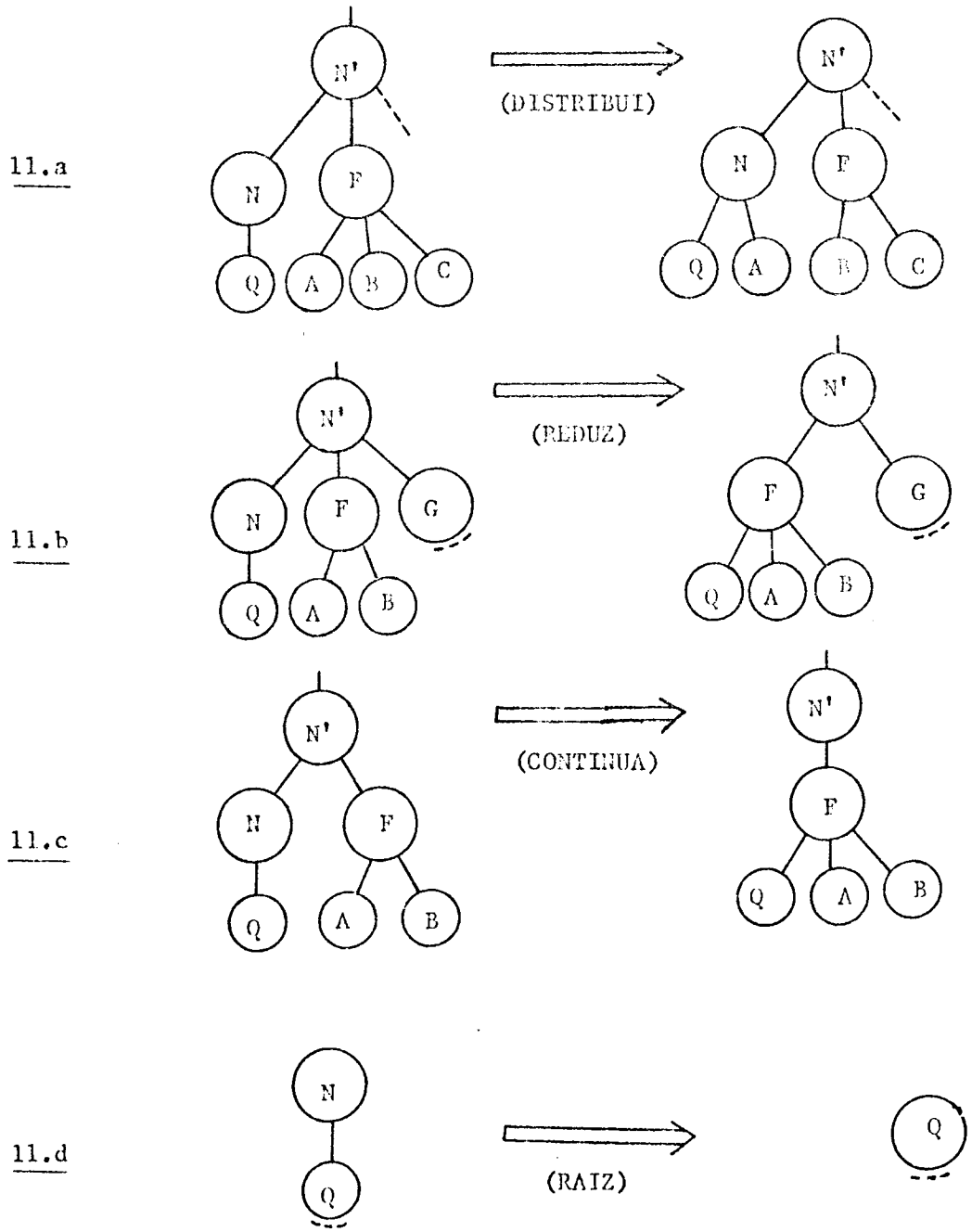


Fig. 11 Casos gerais na eliminação de um nó numa árvore 3-2.

to em 2.2); considerando as operações de reunião, intersecção e diferença e, sendo x_1 e x_2 os nós correntes de respectivamente S1 e S2, temos:

reunião: é criado um novo nó $x_3 = \min \{x_1, x_2\}$

intersecção: é criado um novo nó $x_3 = x_1$ se $x_1 = x_2$

diferença: é criado um novo nó $x_3 = x_1$ se $x_1 \neq x_2$

As condições de terminação para estas três operações são:

reunião: não há mais nós de S1 nem de S2

intersecção: não há mais nós de S1 ou de S2

diferença: não há mais nós de S1.

(2) Construção da árvore 3-2

A árvore 3-2 é construída por níveis a partir da cadeia de terminais, associando grupos de 2 ou 3 nós do nível inferior até se atingir a raíz. Escolhemos um método aleatório de optar entre nós do tipo NO2 ou NO3 uma vez que se houver caminhos da raíz para os nós terminais em que todos os nós são do tipo NO2 (NO3) a operação de eliminação (inserção) de um nó pode ser relativamente ineficiente.

Como exemplo consideremos a reunião do conjunto S1 referido atrás com o conjunto S2 = {CARLOS, JOAO, MARIA, SUSANA, RITA, RUI} ; obtemos inicialmente a cadeia representada na figura 12.

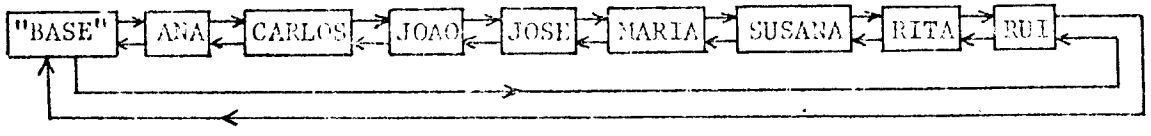


Fig. 12 Cadeia de nós terminais resultante da reunião de S1 e S2.

Constroi-se então uma árvore 3-2 com estes nós terminais, obtendo-se, por exemplo (sem especificar os apontadores nos nós terminais) a árvore representada na figura 13.

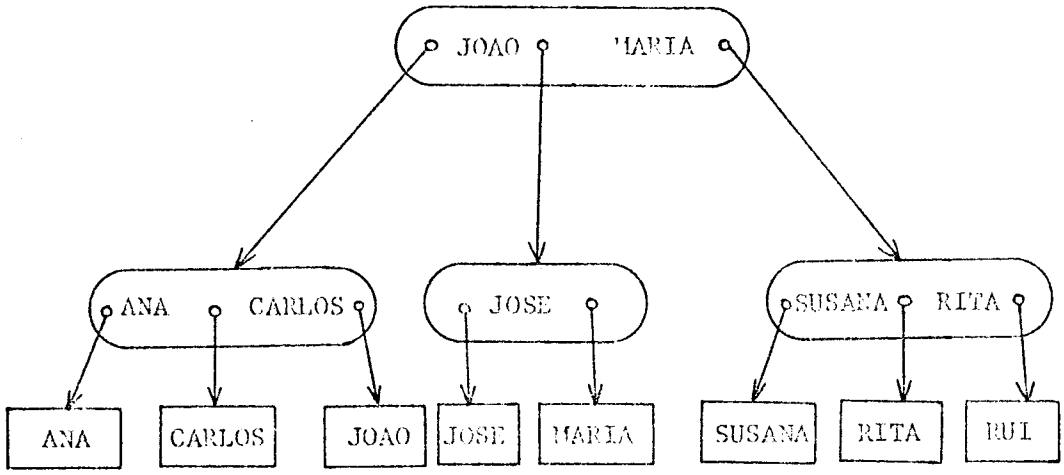


Fig. 13 Árvore 3-2 resultante da reunião de S1 e S2.

É possível utilizar algoritmos directamente derivados das operações básicas para implementar as operações derivadas; contudo a sua eficiência é menor do que a dos algoritmos apresentados (O(N ln N) em vez de O(N)).

6.5. Análise da eficiência do método

Se atendermos aos seguintes factos fáceis de estabelecer:

— a altura de uma árvore 3-2 com N nós terminais está sempre compreendida entre $1+\ln_3 N$ (todos os nós do tipo 3, $N=3^D$) e $1+\ln_2 N$ (todos os nós do tipo

2, $N=2^p$)

— a inserção ou eliminação de um nó na cadeia duplamente ligada de nós terminais é realizada em tempo O(1),

concluímos que as operações básicas são realizadas, mesmo no caso mais desfavorável em tempo O(ln N)

Quanto às operações derivadas observemos que (sendo N1 = |S1|, N2 = |S2|, N3 = |S3|):

(1) a cadeia de nós terminais é construída em tempo O(N1+N2)

(2) o número de nós não terminais está compreendido entre $N3/3 + N3/3^2 + \ldots$ e $N3/2 + N3/2^2 + \ldots$ sendo portanto de ordem O(N3)

Como $N3 \leq N1 + N2$ as operações derivadas são realizadas em tempo O(N1+N2); a eficiência é pois muito grande no caso geral, podendo ainda ser melhorada em casos especiais; como exemplo consideremos a operação:

$$S3 \leftarrow S1 \cup S2,\ FREE(S1)$$

sendo $|S1| \gg |S2|$; construímos S3 inserindo os elementos de S2 em S1, donde resulta um algoritmo com uma complexidade temporal O(N2 ln N1).

Quanto à memória ocupada observamos que o número NT de nós não terminais numa árvore representativa de S, com |S| = N está compreendido entre $N/3+N/3^2+\ldots$ e $N/2+N/2^2+\ldots$; logo $N_T \leq N$; cada nó não terminal pode conter dois valores e 3 apontadores ou um valor e dois apontadores.

Em resumo, o sistema que descrevemos garante (mesmo no caso mais desfavorável) tempo O(ln N) para as operações básicas e O(N) para as operações derivadas; a ocupação de memória é reduzida embora as estruturas representativas das árvores 3-2 não tenham sido concebidas no sentido de minimizar a memória ocupada.

Este sistema foi implementado em linguagem PASCAL, encontrando-se a listagem do programa à disposição dos interessados; não foram feitos quaisquer testes significativos de tempos de execução.

7. SISTEMA DE MANIPULAÇÃO DE CONJUNTOS BASEADO NUM MÉTODO DE "HASHING"

7.1. Introdução

Iremos descrever um sistema particular de manipulação de conjuntos baseado na utilização de um método de "hashing" cujo processo de resolução de colisões combina certas características dos métodos designados por "listas externas" e "listas internas". O tempo de execução das operações de pesquisa, inserção e eliminação é O(1) em média e O(N) no caso mais desfavorável, sendo a dimensão dos conjuntos O(N). Dado que o sistema permite o processamento sequencial em O(1) (por elemento) as operações derivadas podem ser realizadas em tempo médio O(N) e em tempo máximo $O(N^2)$. A análise da eficiência do método mostra que ele é bastante vantajoso para a manipulação de conjuntos.

7.2. Descrição das estruturas de informação

É conveniente dividir a estrutura do sistema em duas partes (uma divisão deste tipo poderia ter sido elaborada para o método das árvores 3-2 estudado anteriormente:

— o espaço dos conjuntos S onde, por cada conjunto representado, existe um nó que contém as suas principais características, entre as quais um apontador

para a cadeia de nós do espaço H (descrito seguidamente) correspondente aos elementos desse conjunto; podemos representar o espaço S da seguinte forma:

S: ARRAY [0 .. SMAX] OF SNO; em que SNO é a estrutura:

```
SNO = RECORD
        NOME: PALAVRA;
        PRIM: ↑HNO;
        DIM: INTEGER
      END;
```

— o espaço de "hashing" H contendo nós indexados de 0 a HMAX, sendo cada elemento X de um conjunto S1 representado por um desses nós; utilizamos a seguinte representação para o espaço H:

H: ARRAY [0 .. HMAX] OF HNO; em que HNO tem a seguinte estrutura:

```
HNO = RECORD
        K: CONTEUDO;
        CONJ: ↑SNO;
        SU, PR: ↑HNO;
        L: ↑HNO
      END;
```

onde

K - representa o elemento correspondente ao nó (pode ser um nome, um valor real, uma posição de memória, etc.)

CONJ - aponta para o conjunto a que pertence o elemento

SU, PR - são dois apontadores que definem respectivamente o nó sucessor e o nó predecessor da dupla cadeia de nós representativa dos elementos do conjunto

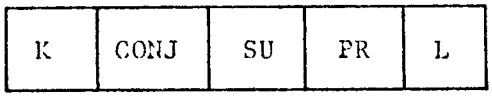
a que pertence o elemento em consideração

L - é um apontador que define o nó seguinte na lista de "hashing" em que o nó corrente está inserido.

Na figura 14 está esquematizada uma representação possível dos conjuntos

$A = \{a, b, c\}$, $B = \{d\}$, $C = \{d, b\}$,

sendo a estrutura de cada nó de H a seguinte:



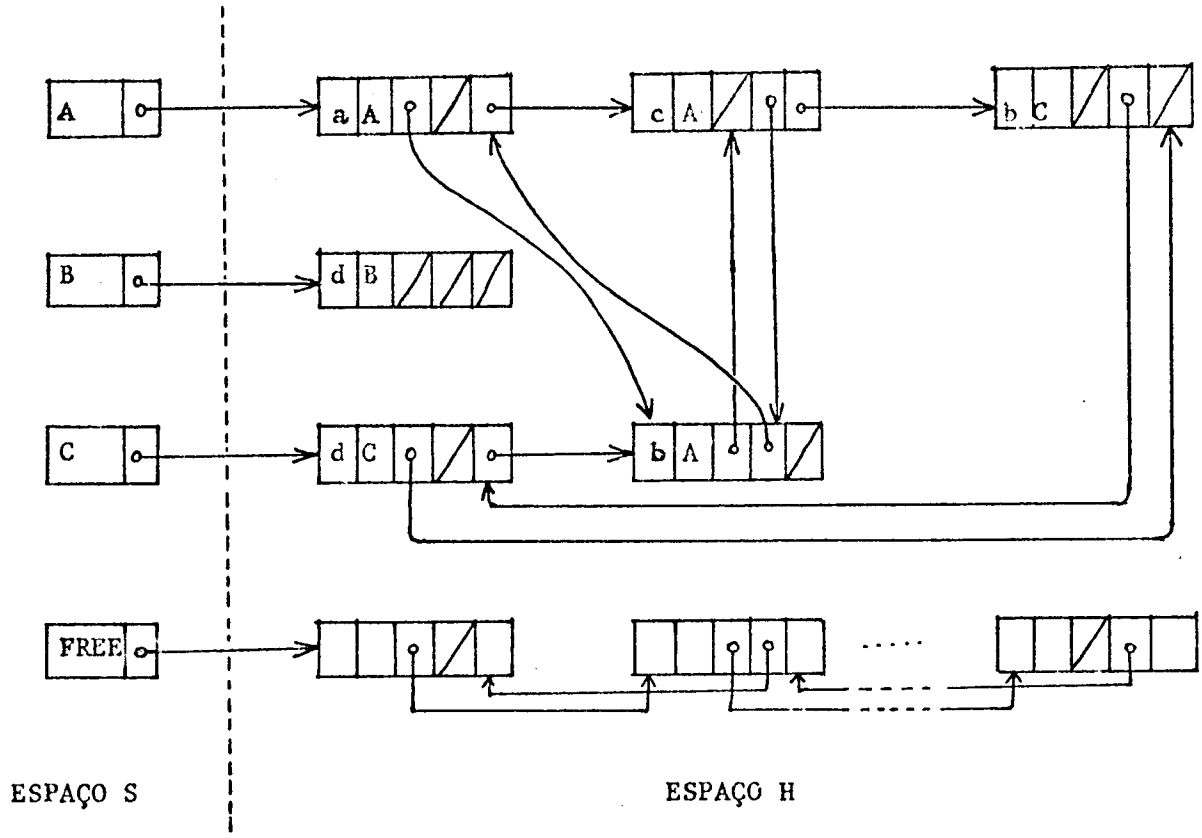


Fig. 14 Possível representação dos conjuntos A, B, e C; os pares (A,a), (A, c) e (C, b) bem como os pares (C, d) e (A, b) têm o mesmo valor de "hashing".

O conjunto "FREE" existe sempre e contém uma dupla cadeia de todos os nós não utilizados.

7.3. Descrição dos algoritmos

A função de "hashing", h, associa a cada par (conjunto, elemento) o índice de um nó de H (entre 0 e HMAX); no sistema que foi implementado os conjuntos e os elementos são designados por palavras de não mais de 10 letras e a função h dá como resultado o valor módulo HMAX+1 de um número obtido pelos códigos das letras dessas palavras.

As operações de pesquisa, inserção e eliminação efectuam-se de forma que os seguintes factos se mantêm invariantes:

(1) cada conjunto (incluindo FREE) existe na forma de uma cadeia de nós de H duplamente ligada cujo primeiro elemento é apontado pelo nó de S correspondente a esse conjunto

(2) cada nó I, $0 \leq I \leq$ HMAX, de H pertence a um e a um só conjunto (incluindo FREE)

(3) cada par (S1,x) com $x \in$ S1 e S1 $\neq$ FREE encontra-se numa lista de "hashing" formada através dos apontadores L e cujo primeiro nó é h_1=h(S1,x); assim, se o nó h_1 não estiver ocupado ou não for o primeiro da sua lista de "hashing", não existe nenhum par (S1, x) tal que h_1 = h (S1,x).

Seja h_1 = h(S1,x) e, se H [h_1] estiver ocupado, h_2 = h(S2, x') onde (S1,x) é o par em consideração e (S2,x') o par contido em H [h_1]; temos a considerar os seguintes casos:

a) se $H[h_1]$ está vazio conclui-se que $x \notin S1$ e (S1,x) poderá ser colocado em $H[h_1]$ depois de retirar este nó de FREE;

b) se $h_2 \neq h_1$ e se $H[h_1]$ estiver ocupado por (S2,x') $\neq$ (S1,x) conclui-se que $x \notin S1$ e a inserção de (S1,x) efectua-se em duas fases:

(1) o par (S2,x') é colocado num novo nó (retirado de FREE) e inserido na lista de "hashing" original;

(2) (S1,x) é colocado em $H[h_1]$, ficando a ser o único elemento da sua lista de hashing.

c) se $h_2 = h_1$ e se $H[h_1]$ estiver ocupado por (S2,x') $\neq$ (S1,x), (S1,x) é pesquisado na lista de "hashing" que começa em h_1; se não for encontrado pode ser inserido num novo nó (retirado de FREE) como último elemento dessa lista; se for encontrado pode ser retirado da lista, sendo o nó respectivo inserido em FREE;

d) finalmente se (S2,x') = (S1,x) (e $h_1 = h_2$) então é claro que $x \in S1$; se a lista de "hashing" que se inicia em h_1 só contém este nó, x pode ser retirado colocando $H[h_1]$ em FREE; se não, retira-se x colocando em $H[h_1]$ o nó que se lhe segue na lista e coloca-se este em FREE.

Há vários detalhes dos algoritmos que não foram referidos; em particular é importante modificar correctamente a dupla cadeia dos conjuntos (apontadores SU e PR) e ajustar, se necessário, qualquer referência externa aos pares que mudem de posição.

O processamento sequencial é trivial utilizando o apontador SU e os algoritmos para as operações derivadas resultam imediatamente da combinação do processamento sequencial com os algoritmos descritos (tendo em atenção que o indice do nó corrente do processamento sequencial pode ser modificado por uma operação de inserção ou de eliminação).

O esquema descrito mantém as propriedades essenciais das listas de "hashing" externas usando um espaço de "hashing" interno.

7.4. Análise da eficiência do método

O esquema que descrevemos permite manter as listas de "hashing" bastante curtas evitando fenómenos de acumulação local de nós ("clustering") quando a tabela está bastante cheia. A análise do tempo de execução dos algoritmos pode ser baseada no número médio de nós examinados durante uma pesquisa.

Designemos por:

$m \equiv$ HMAX + 1, o número de nós do espaço H

$n \equiv \Sigma |Si|$, $Si \neq$ FREE, o número de nós ocupados

$C_n \equiv$ o número médio de nós examinados numa pesquisa de um par existente em H

$C'_n \equiv$ o número médio de nós examinados numa pesquisa de um par não existente na tabela.

Após uma inserção aleatória de n pares é possível estabelecer os seguintes resultados:

$$C'_n = 1 + n/m = 1 + \alpha$$

$$C_n = 1 + \frac{n - 1}{2m} \approx 1 + \alpha/2$$

onde $\alpha \equiv n/m$ é o factor de enchimento da tabela; estes resultados foram obtidos pelo estudo do número médio e do comprimento médio das listas de "hashing".

A estrutura dos algoritmos é tal que, se numa sequência de instruções de inserção e eliminação juntarmos em qualquer posição I_k (inserção do par k) e, depois, (mas não necessariamente a seguir) D_k (eliminação do mesmo par), a configuração final das listas de "hashing" não se altera; assim o método descrito é insensível a efeitos de destruição da aleatoriedade por instruções de eliminação ([12]).

Dado que as operações básicas podem ser realizadas em tempo médio O(1) é fácil definir algoritmos de complexidade temporal média O(N) para as operações de reunião, intersecção e diferença.

O sistema descrito foi implementado nas linguagens BASIC (numa versão bastante poderosa) e PASCAL.

8. COMPARAÇÃO DOS DOIS SISTEMAS DE MANIPULAÇÃO DE CONJUNTOS

Os dois sistemas de manipulação de conjuntos que foram descritos podem ser comparados segundo diversos aspectos; consideremos inicialmente os tempos de execução, cujas ordens de grandeza são resumidas na tabela 2.

Em valores médios o método de "hashing" é vantajoso para as operações básicas; no caso mais desfavorável o método das árvores 3-2 é mais eficiente tanto para as operações básicas como para as derivadas, sendo portanto preferível para sistemas de "tempo real" em que o tempo de resposta deve ser garantido. É interessante comparar estes resultados com os métodos elementares (tabela 1).

Quanto à memória ocupada pode-se afirmar que não há grande diferença no

TABELA 2

Comparação das ordens de grandeza dos tempos de execução dos dois sistemas implementados

Método	Operações básicas		Operações derivadas	
	Tempo médio	Tempo máximo	Tempo médio	Tempo máximo
árvores 3-2	O(ln N)	O(ln N)	O(N)	O(N)
"hashing"	O(1)	O(N)	O(N)	$O(N^2)$

espaço ocupado por cada elemento de um conjunto representado (com alguma vantagem para o método de "hashing"); todavia o método das árvores 3-2 é muito mais flexível em termos de memória pois em cada instante ocupa apenas os nós que estão utilizados; deve contudo dizer-se que o espaço fixado antecipadamente para o método de "hashing" pode ser compartilhado com outras estruturas ligadas (árvores binárias, por exemplo).

Foi efectuado um estudo um pouco mais detalhado da ocupação de memória pelos dois métodos referidos; nas árvores 3-2 a percentagem de nós do tipo 2 depende da história de formação das árvores; admitindo por exemplo que há 2/3 de nós do tipo 2 (independentemente do nível), que os nós terminais são representados por um "conteúdo" e dois apontadores e que os nós não terminais utilizam dois "conteúdos" e três apontadores obtemos para o número de bits ocupados por cada par armazenado:

árvores 3-2: 2.5k + 4.25p

"hashing": 2k + 3p

em que k é o número de bits usado por cada "conteúdo" (conjunto ou elemento) e p o número de bits usado por cada apontador.

Finalmente deve referir-se que a qualidade de implementação pode influir bastante na eficiência final; pela experiência adquirida parece-nos que o método das árvores 3-2 é relativamente difícil de implementar com eficiência.

BIBLIOGRAFIA

[1] AHO, HOPCROFT, ULLMAN - The Design and Analysis of Computer Algorithms; Addison-Wesley, 1974

[2] BACKUS - Can Programming Be Liberated from the Von - Neumann Style? A Functional Style and its Algebra of Programs; CACM, Agosto 1978, pág. 613-641

[3] BRENT - Reducing the Retrieval time of Scatter Storage Techniques; CACM, Fevereiro 1973, pág. 105-109

[4] CODD - A Relational Model of Data for Large Shared Data Banks; CACM, Junho 1970, pág. 377-387

[5] FAGIN, NIEVERGELT, PIPPENGER, STRONG - Extendible Hashing - A Fast Access Method for Dynamic Files; ACM Transactions on Database Systems, Setembro 1979, pág. 315-344

[6] GRIES - Compiler Construction for Digital Computers; Wiley, 1970

[7] HALATSIS, PHILOKYPROU - Pseudochaining in Hash Tables; CACM, Julho 1978, pág. 554-557

[8] HARARY - Graph Theory; Addison-Wesley, 1969

[9] HIRSCHBERG - An Insertion Technique for One-Sided Height Balanced-Trees; CACM, Agosto 1976, pág. 471-473

[10] KNUTH - The Art of Computer Programming, Vol. 1: Fundamental Algorithms; Addison-Wesley, 1968

[11] KNUTH - The Art of Computer Programming, Vol. 3: Sorting and Searching; Addison-Wesley, 1973

[12] KNUTH - Deletions That Preserve Randomness; IEEE Transactions on Software Engineering; Setembro 1977, pág. 351-359

[13] KOWALSKY - Algorithms = Logic + Control; CACM, Julho 1979, pág. 424-436

[14] MAURER, LEWIS - Hash Table Methods; Computing Surveys, Março 1975, pág. 5-19

[15] SHILOACH - Union Members Algorithms for non-disjoint Sets; Stanford Computer Science Department Report STAN-CS-79-728

[16] PARDO - Set Representation and Set Intersection; Stanford Computer Science Department Report STAN-CS-78-681

[17] TARJAN - Storing a Sparse Table; Stanford Computer Science Department Report STAN-CS-78-683

[18] WARREN - SETL Implementation: Data Structures, Primitives and Storages Menagement; I.J.C.M., 1974, pág. 77-94

[19] OTTMANN, SIX - Right Brother Trees; CACM, Setembro 1978, pág. 769-776

[20] WIRTH - Algorithms + Data Structures = Programs; Prentice-Hall, 1976

[21] YAO - Should Tables be Sorted?; 19th Symposium on Foundation of Computer Science, IEEE, 1978

[22] ZWEBEN, McDONALD - An Optimal Method for Deletion of One-Sided Height Balance Trees, CACM, Junho 1978, pág. 441-445

[23] DERSHOWITZ AND MANNA - Proving Termination with Multiset Ordering; CACM, Agosto 1979, pág 465- 475

[24] GRIES - An Illustration of Current Ideas on the Derivation of Correcteness Proofs and Correct Programs - IEEE TSE, Dezembro 1976, pág. 238-243

[25] RÄIHÄ e ZWEBEN - An Optimal Insertion Algorithm for One-Sided Height-Balanced Binary Search Trees - CACM, Setembro 1979, pág. 508-512