Funções Finitas II: árvores AVL

Métodos de Programação II

2004/2005

1 Objectivos

Estender a biblioteca de árvores binárias para suportar *Árvores AVL*.

2 Motivação

> Pretende-se adaptar o programa apresentado na última aula prática para produzir o histograma das palavras que ocorrem num texto (lido do stdin). Adicionalmente, devemos prever que o referido histograma assuma dimensões muito elevadas (milhares de entradas), pelo que se deverá investir na melhoria de eficiência do programa.

A extensão proposta para o programa coloca duas questões:

- O histograma que se pretende realizar é agora das palavras (e não do tamanho das palavras). Isto quer dizer que as chaves para a função finita (e consequentemente da árvore binária) vão deixar de ser inteiros, passando a ser sequências de caracteres (*strings*).
- As preocupações de eficiência referidas devem-se traduzir na implementação de algoritmos que maximizem a eficiência das operações sobre as árvores binárias. Este assunto levar-nos-á a considerar os aspectos de balanceamento dessas árvores como o factor determinante para a eficiência das referidas operações.

3 Adaptação do tipo de chaves

São concebíveis três aproximação à alteração do tipo das chaves na biblioteca de árvores binárias:

1. Modificar a biblioteca existente para trabalhar com chaves do tipo `char*`;
2. Generalizar o tipo de chaves atribuindo ao programador a responsabilidade de definir uma função para comparação de chaves;
3. Generalizar o tipo de chaves o parametrizar todas as operações que dependem da comparação de chaves com uma função de comparação.

A primeira solução é notoriamente mais simples. Resume-se a alterar a definição do tipo do nó da árvore e substituir todas as comparações de chaves realizadas nas operações pelas invocações apropriadas de `strcmp`.

Na segunda abordagem, generaliza-se o tipo da chave para `void*`. Para a comparação das chaves, declara-se uma função como:

```
extern int compareKeys(void *, void*);
```

que produz um resultado do tipo de `strcmp` (0, -1 ou 1). O código das operações fará uso dessa função, sendo que ela só será definida por que vier a utilizar a biblioteca (instanciando aí com a comparação apropriada para o tipo de chave utilizado). Como exemplo, para os tipos *string* e *int* poderíamos definir

```
int compareKeys(void* k1, void* k2)
{
  return strcmp((char*) k1, (char*) k2);
}
```

```
int compareKeys(void *k1, void* k2)
{
  if ( *(int*)k1 == *(int*)k2 ) return 0;
  if ( *(int*)k1 < *(int*)k2 ) return -1;
  return 1;
}
```

(note a utilização dos *casts* para aceder à informação apontada por um apontador de tipo `void*`.

O problema desta solução reside unicamente se pretendermos fazer uso dois ou mais tipos de chaves num mesmo programa. Nesse caso deveríamos definir várias instâncias da função de comparação, o que não nos é permitido na linguagem C. Para ultrapassar essa dificuldade poderíamos passar a função de comparação como parâmetro das funções da biblioteca que dela necessitem. A título de exemplo, o protótipo para a função de inserção seria:

```
Tree treeInsert(Tree t, void* key, void* value, int (*compK)(void* k1,void* k2));
```

Desta forma, cada invocação da função de inserção passa como argumento a função de comparação de chaves apropriada.

4 Árvores AVL

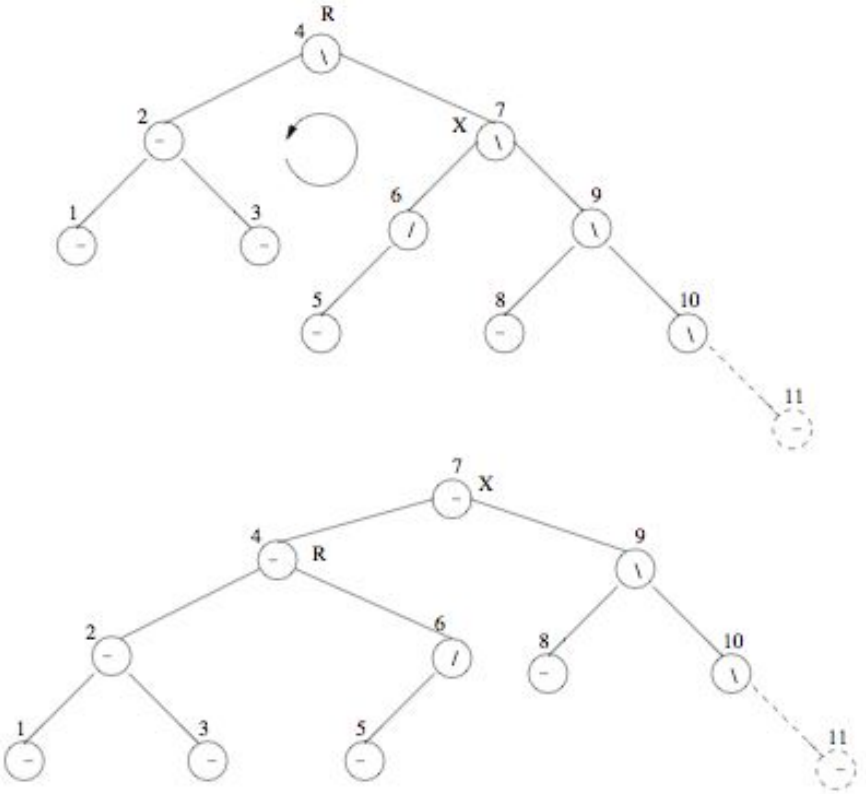
A eficiência das operações nas árvores binárias de procura dependem crucialmente de estas se manterem equilibradas (em cada nó encontramos um número aproximado de elementos em cada uma das sub-árvores). As árvores AVL impõe um invariante adicional sobre as árvores binárias de procura:

> O peso de cada uma das sub-árvores difere de, quanto muito, uma unidade e cada uma delas satisfaz recursivamente este invariante.

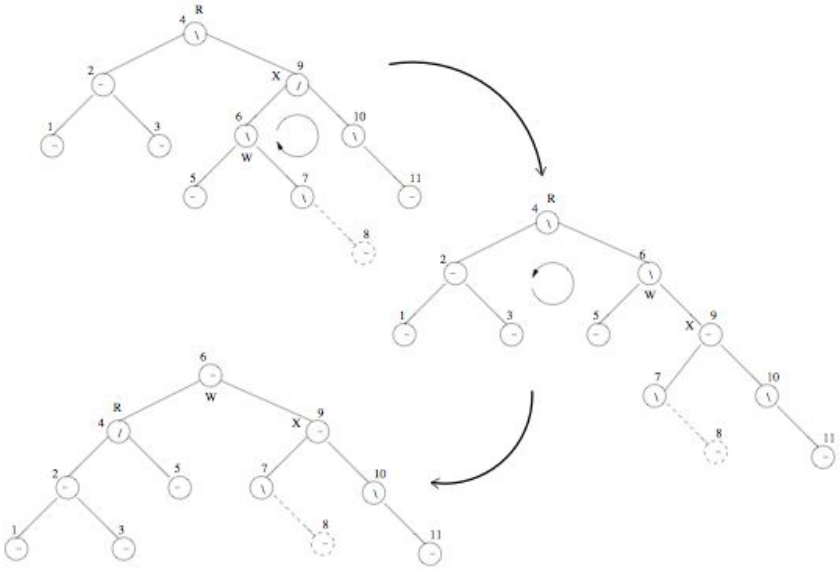
4.1 Inserção em árvores AVL

O algoritmo de inserção usual em árvores binárias de procura não é aplicável nas árvores AVL porque não é garantido que o invariante se verifique na árvore resultante. O algoritmo de inserção em árvores AVL é assim consideravelmente mais elaborado já que prevê todas as possibilidades de violação do invariante e procede a rearranjos apropriados na árvore por forma a repor a sua validade.

As figuras que se seguem exibem os casos onde a inserção de um elemento numa árvore AVL faz com que o invariante seja violado (só se exibem os casos onde a inserção ocorre na sub-árvore direita – o tratamento dos casos onde essa inserção ocorre na sub-árvore esquerda deriva-se facilmente por simetria).



Neste caso, uma rotação da árvore é suficiente para repor a validade do invariante. Já quando o desequilíbrio da sub-árvore direita é o contrário, devemos proceder a duas rotações para repor o invariante.



4.2 Implementação em C

Para a sua implementação em C, é conveniente considerar os seguintes aspectos:

1. deve-se guardar na própria árvore qual o tipo de desequilíbrio que ela exibe (o *factor de balanceamento* — esquerdo, nulo ou direito). Desta forma ser-nos-á possível decidir se a inserção do novo elemento obriga, ou não, ao rearranjo da árvore;

2. a função de inserção deve retornar, para além da árvore resultante, informação se a inserção provocou ou não um acréscimo da profundidade da árvore. Essa informação permitir-nos-á recalcular os factores de balanceamento.

Teremos oportunidade de, nas aulas práticas, aprofundar o funcionamento do algoritmo e de explicar a implementação que se segue.

```
typedef (char*) KeyType;
#define KEYCMP(k1,k2) strcmp(k1,k2)

typedef enum balancefactor { LH , EH , RH } BalanceFactor;

typedef struct treeNode{
  BalanceFactor bf;
  void *key;
  void *value;
  struct treeNode *left;
  struct treeNode *right;
} TreeNode;


typedef TreeNode *Tree;

Tree treeInsertBal(Tree t, void *key, void *value, int *cresceu) {
  if (t==NULL){
    t = (Tree) malloc(sizeof(TreeNode));
    t->key = key;
    t->value = value;
    t->bf=EH;
    t->right=NULL;
    t->left=NULL;
    *cresceu=1;
  }
  else if (KEYCMP(key,t->key)==0) {
    if ( t->value != NULL ) free(t->value);
    t->value = value;
    *cresceu = 0;
  }
  else if (KEYCMP(key,t->key)>0) {
    t->right = treeInsertBal(t->right, key, value, cresceu);
    if (*cresceu) {
      switch (t->bf) {
      case LH:
t->bf=EH;
*cresceu=0;
```

```
break;
      case EH:
t->bf=RH;
*cresceu=1;
break;
      case RH:
t=balanceRight(t);
*cresceu=0;
      }
    }
  }
  else {
// A IMPLEMENTAR...
    }
  }
  return(t);
}

Tree balanceRight(Tree t) {
  if (t->right->bf==RH) {
    // Rotacao simples a esquerda
    t=rotateLeft(t);
    t->bf=EH;
    t->left->bf=EH;
  }
  else {
    //Dupla rotacao
    t->right=rotateRight(t->right);
    t=rotateLeft(t);
    switch (t->bf) {
    case EH:
      t->left->bf=EH;
      t->right->bf=EH;
      break;
    case LH:
      t->left->bf=EH;
      t->right->bf=RH;
      break;
    case RH:
      t->left->bf=LH;
      t->right->bf=EH;
    }
    t->bf=EH;
  }
  return t;
}

Tree balanceLeft(Tree t) {
// A IMPLEMENTAR...
}

Tree rotateRight(Tree t) {
  Tree aux;
```

```
  if ((! t)||(! t->left)) {
    printf("Erro\n");
  }
  else {
    aux=t->left;
    t->left=aux->right;
    aux->right=t;
    t=aux;
  }

  return t;
}

Tree rotateLeft(Tree t) {
  Tree aux;

  if ((! t)||(! t->right)) {
    printf("Erro\n");
  }
  else {
    aux=t->right;
    t->right=aux->left;
    aux->left=t;
    t=aux;
  }

  return t;
}
```