Mais tabela de símbolos
Uma **tabela de símbolos** (= symbol table = dictionary) é um conjunto de objetos (itens), cada um dotado de uma chave (= key) e de um valor (= value).
As chaves podem ser números inteiros ou strings ou outro tipo de dados.
Uma tabela de símbolos está sujeito a dois tipos de operações:
- **inserção**: consiste em introduzir um objeto na tabela;
- **busca**: consiste em encontrar um elemento que tenha uma dada chave.

Problema
Problema: Organizar uma tabela de símbolos de maneira que as operações de inserção e busca sejam razoavelmente eficientes.
Em geral, uma organização que permite inserções rápidas impede buscas rápidas e vice-versa.
Já vimos como organizar tabelas de símbolos através de vetores, lista encadeadas e hash.
Hoje: mais uma maneira de organizar uma tabela de símbolos.

Árvore binárias
Uma **árvore binária** (= binary tree) é um conjunto de nós/células que satisfaz certas condições.
Cada nó terá três campos:
```c
typedef struct celula Celula;
struct celula {
    int conteudo; /* tipo devia ser Item*/
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
No x, y;
```
Os campos `esq` e `dir` dão estrutura à árvore.
Se `x.esq == y`, `y` é o **filho esquerdo** de `x`.
Se `x.dir == y`, `y` é o **filho direito** de `x`.
Assim, `x` é o **pai** de `y` se `x.esq == y` ou `x.dir == y`.
Um **folha** é um nó sem filhos.
Ou seja, se `x.esq == NULL` e `x.dir == NULL` então `x` é um folha.
Pais e filhos
Ilustração de uma árvore binária

Árvores e subárvores

Suponha que r e p são (endereços de/ponteiros para) nós. p é descendente de r se p pode ser alcançada pela iteração dos comandos

\[ p = p->esq; \quad p = p->dir; \]

em qualquer ordem.

Um nó r juntamente com todos os seus descendentes é uma árvore binária e r é dito a raiz (=root) da árvore.


Endereço de uma árvore

O endereço de uma árvore binária é o endereço de sua raiz.

```c
typedef No *Arvore;
Arvore r;
```

Um objeto r é uma árvore binária se

- r == NULL ou
- r->esq e r->dir são árvores binárias.

Endereço de uma árvore

Maneiras de varrer uma árvore

Existem várias maneiras de percorrermos uma árvore binária. Talvez as mais tradicionais sejam:

- \textit{inorder traversal}: esquerda-raiz-direita (e-r-d);
- \textit{preorder traversal}: raiz-esquerda-direita (e-d-r);
- \textit{posorder traversal}: esquerda-direita-raiz (e-d-r);

esquerda-raiz-direita

Visítamos

1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. a subárvore direita da raiz, em ordem e-r-d;

```c
void inOrdem(Arvore r) {
    stackInit();
    while (r != NULL || !stackEmpty()) {
        if (r != NULL) {
            stackPush(r);
            r = r->esq;
        } else {
            r = stackPop();
            printf("%d \n", r->conteudo);
            r = r->dir;
        }
    }
}
```

esquerda-raiz-direita versão iterativa

```c
void inOrdem(Arvore r) {
    stackInit();
    while (r != NULL || !stackEmpty()) {
        if (r != NULL) {
            stackPush(r);
            r = r->esq;
        } else {
            r = stackPop();
            printf("%d \n", r->conteudo);
            r = r->dir;
        }
    }
}
```
Ilustração de percursos em árvores binárias

pré-ordem (r-e-d): 12 -1 31 7 3 -6 10 11 25 42
pós-ordem (e-d-r): 31 7 -1 10 11 -6 42 25 3 12
in-ordem (e-r-d): 31 27 * 10 11 - True not / +

Primeiro nó esquerda-raiz-direita

Recebe a raiz r de uma árvore binária não vazia e retorna o primeiro nó na ordem e-r-d

```c
No *primeiro(Arvore r)
{
    while (r->esq != NULL)
    { r = r->esq; 
    return r; 
}
```

Árvores balanceadas

A altura de uma árvore com n nós é um número entre $\lg(n)$ e n.

Uma árvore binária é balanceada (ou equilibrada) se, em cada um de seus nós, as sub-árvore esquerda e direita tiverem aproximadamente a mesma altura.

Árvores balanceadas têm altura próxima de $\lg(n)$.

O consumo de tempo dos algoritmos que manipulam árvores binárias dependem frequentemente da altura da árvore.

Ilustração de percursos em árvores binárias

pré-ordem (r-e-d): 12 -1 31 7 3 -6 10 11 25 42
pós-ordem (e-d-r): 31 7 -1 10 11 -6 42 25 3 12
in-ordem (e-r-d): 31 27 * 10 11 - True not / +

Altura

A altura de p é o número de passos do mais longo caminho que leva de p até uma folha.

A altura de uma árvore é a altura da sua raiz. Altura de árvore vazia é -1.

```c
#define MAX(a,b) ((a) > (b)? (a): (b))
int altura(Arvore r) {
    if (r == NULL) return-1;
    else {
        int he = altura(r->esq);
        int hd = altura(r->dir);
        return MAX(he,hd) + 1;
    }
    }
```

Nós com campo pai

Em algumas aplicações é conveniente ter acesso imediato ao pai de qualquer nó.

```c
typedef struct celula Celula;
struct celula {
    int conteudo; /* tipo devia ser Item*/
    Celula *pai;
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
typedef No *Arvore;
```
Sucessor e predecessor

Recebe o endereço \( p \) de um nó de uma árvore binária não vazia e retorna o seu sucessor na ordem e-r-d.

No *sucessor(No *p) {
    if (p->dir != NULL) {
        No *q= p->dir;
        while (q->esq != NULL) q = q->esq;
        return q;
    }
    while (p->pai!=NULL && p->pai->dir==p)
        p = p->pai;
    return p->pai;
}
Exercício: função que retorna o predecessor.

Árvore binárias de busca

Considera uma árvore binária cujos nós têm um campo chave (como int ou String, por exemplo).

typedef struct celula Celula;
struct celula {
    int conteudo; /* tipo devia ser Item*/
    int chave; /* tipo devia ser Chave*/
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
typedef No *Arvore;
No x, *p, *q, *r, *t;

Árvore binárias de busca

Uma árvore binária deste tipo é de busca (em relação ao campo chave) se para cada nó \( x \) \( x.chave \) é

1. maior ou igual à chave de qualquer nó na subárvore esquerda de \( x \) e
2. menor ou igual à chave de qualquer nó na subárvore direita de \( x \).

Assim, se \( p \) é um nó qualquer então vale que

\[
q->chave \leq p->chave \quad \text{e} \quad p->chave \leq t->chave
\]

para todo nó \( q \) na subárvore esquerda de \( p \) e todo nó \( t \) na subárvore direita de \( p \).

Ilustração de uma árvore binária de busca

in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

Sucessor e predecessor

Recebe o endereço \( p \) de um nó de uma árvore binária não vazia e retorna o seu sucessor na ordem e-r-d.

No *sucessor(No *p) {
    if (p->dir != NULL) {
        No *q= p->dir;
        while (q->esq != NULL) q = q->esq;
        return q;
    }
    while (p->pai!=NULL && p->pai->dir==p)
        p = p->pai;
    return p->pai;
}
Exercício: função que retorna o predecessor.

Árvore binárias de busca

Considera uma árvore binária cujos nós têm um campo chave (como int ou String, por exemplo).

typedef struct celula Celula;
struct celula {
    int conteudo; /* tipo devia ser Item*/
    int chave; /* tipo devia ser Chave*/
    Celula *esq;
    Celula *dir;
};
typedef Celula No;
typedef No *Arvore;
No x, *p, *q, *r, *t;

Árvore binárias de busca

Uma árvore binária deste tipo é de busca (em relação ao campo chave) se para cada nó \( x \) \( x.chave \) é

1. maior ou igual à chave de qualquer nó na subárvore esquerda de \( x \) e
2. menor ou igual à chave de qualquer nó na subárvore direita de \( x \).

Assim, se \( p \) é um nó qualquer então vale que

\[
q->chave \leq p->chave \quad \text{e} \quad p->chave \leq t->chave
\]

para todo nó \( q \) na subárvore esquerda de \( p \) e todo nó \( t \) na subárvore direita de \( p \).

Ilustração de uma árvore binária de busca

in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42
**Busca**

Recebe um inteiro \( k \) e uma árvore de busca \( r \) e retorna um nó cuja chave é \( k \); se tal nó não existe, retorna \( NULL \).

```c
No *busca(Arvore r, int k) {
    if (r == NULL || r->chave == k)
        return r;
    if (r->chave > k)
        return busca(r->esq, k);
    return busca(r->dir, k);
}
```

**Busca versão iterativa**

Recebe um inteiro \( k \) e uma árvore de busca \( r \) e retorna um nó cuja chave é \( k \); se tal nó não existe, retorna \( NULL \).

```c
No *busca(Arvore r, int k) {
    while (r != NULL && r->chave != k)
        if (r->chave > k)
            r = r->esq;
        else
            r = r->dir;
    return r;
}
```

**Inserção**

Recebe uma árvore de busca \( r \) e um nó novo. Insere o nó no lugar correto da árvore de modo que a árvore continue sendo de busca e retorna o endereço da nova árvore.

```c
No *new(int chave,int conteudo,No *esq,No*dir) {
    No *novo = mallocSafe(sizeof *novo);
    novo->chave= chave;
    novo->conteudo = conteudo;
    novo->esq = esq;
    novo->dir = dir;
    return novo;
}
```

**Inserção**

```c
Arvore *insere(Arvore r, No *novo) {
    No *f, /* filho de p */
    No *p; /* pai de f */
    if (r == NULL) return novo;
    f = r;
    while (f != NULL) {
        p = f;
        if (f->chave > novo->chave)
            f = f->esq;
        else
            f = f->dir;
    }
    /* novo sera uma folha
    novo sera filho de p */
    if (p->chave > novo->chave)
        p->esq = novo;
    else
        p->dir = novo;
    return r;
}
```

**Remoção**

Recebe uma árvore de busca não vazia \( r \). Remove a sua raiz e rearranja a árvore de modo que continue sendo de busca e retorna o endereço da nova árvore.

```c
Arvore *remove(Arvore r) {
    No *p; /* pai de f */
    if (r == NULL) return NULL;
    No *folha; /* no filho de p */
    while (r != NULL) {
        No *f = r;
        if (f->chave > novo->chave)
            f = f->esq;
        else
            f = f->dir;
    }
    /* novo sera uma folha
    novo sera filho de p */
    if (p->chave > novo->chave)
        p->esq = novo;
    else
        p->dir = novo;
    return r;
}
```
Remoção

Arvore *removeRaiz(Arvore r) {
  No *p, *q;
  if (r->esq == NULL) {
    q = r->dir; free(r); return q;
  }
  /* encontre na subárvore r->esq o nó q com maior valor */
  p = r; q = r->esq;
  while (q->dir != NULL) {
    p = q;
    q = q->dir;
  }
  /* q é o nó anterior a r na ordem e-r-d, p é o pai de q */
  if (p != r) {
    p->dir = q->esq;
    q->esq = r->esq;
  }
  q->dir = r->dir; free(r); return q;
}

Consumo de tempo

O consumo de tempo das funções busca, insere e removeRaiz é, no pior caso, proporcional à altura da árvore.

Conclusão: interessa trabalhar com árvores balanceadas: árvores AVL, árvores rubro-negras, árvores ...

Comentários finais

Fonte: http://www.quickmeme.com/

MAC0122 – Edição 2014

Livros

Nossa referência básica foi o livro
*PF = Paulo Feofiloff, Algoritmos em linguagem C,*

Este livro é baseado no material do sítio *Projeto de Algoritmos em C.*

Outro livro foi
*S = Robert Sedgewick, Algorithms in C, vol. 1*

MAC0122 foi uma disciplina introdutória em:

- **recursão:** torres de Hanoi, ... (EP2, EP5...)
- **divisão-e-conquista:** Mergesort, Quicksort, EP5
- **pré-processamento:** Heapsort, Boyer-Moore
- **heurísticas:** Boyer-Moore, EP5
- **algoritmos de enumeração:** EP1, nrainhas
- **programação dinâmica** ("recursão com tabela"): números binomiais.
MAC0122 foi uma disciplina introdutória em:

Correção de algoritmos:
- relações invariantes: vários problemas nas aulas

Eficiência de algoritmos:
- consumo de tempo: vários problemas nas aulas
- notação assintótica $O$: vários problemas nas aulas
- análise experimental: vários problemas nas aulas
- consumo de espaço: Mergesort usa espaço extra $O(n)$, Quicksort usa espaço extra $O(lg n)$

MAC0122 foi uma disciplina introdutória em:

Estruturas de dados:
- filas: distâncias, EP2, EP3
- heaps

MAC0122 combinou conceitos e recursos de programação:
- strings: todos os EPs?
que nasceram de aplicações cotidianas em ciência da computação: bsearch (stdlib), qsort (stdlib), strstr (string), hsearch (search), lsearch (search), tsearch (search),...

Principais tópicos
Alguns dos tópicos de MAC0122 foram:
- recursão;
- busca em um vetor; busca (binária) em vetor ordenado;
- listas encadeadas;
- listas lineares: filas e pilhas;
- algoritmos de enumeração; divisão e conquista;
- busca de palavras em um texto;
- algoritmos de ordenação: bubblesort, heapsort, mergesort, ...

Tudo isso regado a muita análise de eficiência de algoritmos e invariantes.

PAUSA PARA Nossos comerciais