Algoritmos e Estrutura de Dados

Árvores Binárias de Busca

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Faculdade de Engenharia

Agenda

Exercício

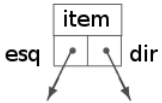
Vamos implementar a estrutura de dados Árvore Binária de Busca (ABB).

Árvore Binária

Implementação

Uma ABB é formada pela estrutura **nodo**, que contém três campos:

- Um ponteiro **esq**, que indica o filho da esquerda daquele nodo.
- Um ponteiro **dir**, que indica o filho da direita daquele nodo.
- Um campo **item** do tipo **int**, que é o tipo de dado a ser armazenado no nodo da árvore.



Árvore Binária

Estrutura Nodo

Segue o pseudo-código referente à estrutura nodo:

Algoritmo 1: Nodo

```
1 início
2     registro {
3         Inteiro: item;
4         Ponteiro Nodo: esq;
5         Ponteiro Nodo: dir;
6     } Nodo;
```

Árvore Binária

Estrutura Nodo

Na linguagem C, a implementação fica conforme o código a seguir:

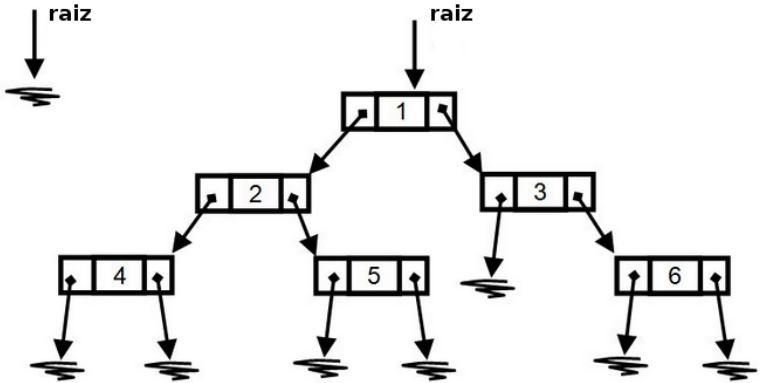
```
1 typedef struct nodo_t
2 {
3     int item;
4     struct nodo_t *esq;
5     struct nodo_t *dir;
6 } nodo;
```

Árvore Binária

Estrutura Árvore

A estrutura **árvore** trata-se de um ponteiro do tipo **nodo**:

- O ponteiro **raiz** aponta para o nodo raiz da árvore.
- Se a árvore está vazia, o ponteiro **raiz** aponta para NULL.



Algoritmo 2: Arvore

```
1 início
2     Tipo Nodo* Arvore;
```

Na linguagem C, a implementação fica conforme o código a seguir:

```c
typedef nodo* Arvore;
```

Árvore Binária de Busca

Inserção

Agora, vamos implementar dois algoritmos básicos:

- **CriaÁrvore**: cria uma árvore vazia.
- **ArvoreVazia**: verifica se uma árvore está vazia.

Algoritmo 3: CriaÁrvore

Saída: Ponteiro r para raiz.

```
1 início
2     r ← NULL
```

Algoritmo 4: ÁrvoreVazia

Entrada: Ponteiro r para raiz.
Saída: V ou F

```
1 início
2     retorna (r = NULL)
```

Árvore Binária

Estrutura Árvore

Na linguagem C, a implementação dessas funções fica conforme o código a seguir:

```
1  Arvore* cria_arvore() {
2      Arvore* raiz = (Arvore*) malloc(sizeof(Arvore));
3      if(raiz != NULL)
4          *raiz = NULL;
5      return raiz;
6  }
7
8  int arvore_vazia(nodo *r) {
9      return r == NULL;
10 }
```

Inserção

Introdução

Agora, vamos implementar um algoritmo para **inserir** dados na nossa árvore. Podemos implementar duas versões:

- **Recursiva**: utiliza chamadas recursivas.
- **Iterativa**: utiliza laços de repetição.

Inserção

Pseudo-Código Versão Recursiva

Algoritmo 5: InserirÁrvore

Entrada: Ponteiro *r* para raiz, item *x* a ser inserido na árvore.

```
1  início
2      se (r=NULL) então
3          novo ← ALOCA_NODO()
4          novo.item ← x
5          novo.esq ← NULL
6          novo.dir ← NULL
7          r ← novo
8      se (x < r.item) então
9          InserirÁrvore(r.esq, x)
10     senão
11         InserirÁrvore(r.dir, x)
```

Inserção

Código-Fonte Versão Recursiva

Na linguagem C, a implementação dessa função fica conforme o código a seguir:

```
1  int inserir_arvore_recursivo(Arvore* raiz, int valor){
2      if(raiz == NULL)
3          return 0;
4      nodo* novo;
5      novo = (nodo*) malloc(sizeof(nodo));
6      if(novo == NULL)
7          return 0;
8      novo->item = valor;
9      novo->dir = NULL;
10     novo->esq = NULL;
11
12     if(*raiz == NULL)
13         *raiz = novo;
14     else{
15         if(valor < (*raiz)->item){
16             return inserir_arvore_recursivo(&(*raiz)->esq, valor);
17         }
18         else
19             return inserir_arvore_recursivo(&(*raiz)->dir, valor);
20     }
21     return 1;
22 }
```

Inserção

Pseudo-Código Versão Iterativa

Algoritmo 6: InserirÁrvore

Entrada: Ponteiro *r* para raiz, item *x* a ser inserido na árvore.

```
1  início
2      novo ← ALOCA_NODO()
3      novo.item ← x
4      novo.esq ← NULL
5      novo.dir ← NULL
6      se (r=NULL) então
7          r ← novo
8      senão
9          atual ← r
10         anterior ← NULL
11         enquanto (atual ≠ NULL) faça
12             anterior ← atual
13             se (novo.item < atual.item) então
14                 atual ← atual.esq
15             senão
16                 atual ← atual.dir
17         se (novo.item < anterior.item) então
18             anterior.esq ← novo
19         senão
20             anterior.dir ← novo
```

Inserção

Código-Fonte Versão Iterativa - Parte 1

Na linguagem C, a implementação dessa função fica conforme o código a seguir:

```
1  int inserir_arvore_iterativo(Arvore* raiz, int valor){
2      if(raiz == NULL)
3          return 0;
4      nodo* novo;
5      novo = (nodo*) malloc(sizeof(nodo));
6      if(novo == NULL)
7          return 0;
8      novo->item = valor;
9      novo->dir = NULL;
10     novo->esq = NULL;
11
12     if(*raiz == NULL)
13         *raiz = novo;
14         /** continua no proximo slide **/
```

Inserção

Código-Fonte Versão Iterativa - Parte 2

```
12      if(*raiz == NULL)
13          *raiz = novo;
14      /** slide anterior **/
15      else {
16          nodo* atual = *raiz;
17          nodo* ant = NULL;
18          while (atual != NULL) {
19              ant = atual;
20              if(valor == atual->item) {
21                  free(novo);
22                  return 0;
23              }
24
25              if(valor > atual->item)
26                  atual = atual->dir;
27              else
28                  atual = atual->esq;
29          }
30          if(valor > ant->item)
31              ant->dir = novo;
32          else
33              ant->esq = novo;
34      }
35      return 1;
36  }
```

Quantidade de Nodos

Introdução

Vamos implementar um algoritmo que calcula a **quantidade de nodos** na nossa árvore.

Quantidade de Nodos

Introdução

A quantidade de nodos em uma árvore com raiz em r é calculada da seguinte forma:

- Verifica se r é vazio.
  - Caso sim, a quantidade de nodos é zero.
  - Caso contrário, o total de nodos da árvore com raiz em r será a soma da quantidade de nodos em suas subárvores mais um (referente ao nodo raiz r).
- A seguir, o pseudocódigo.

Quantidade de Nodos

Pseudo-Código

Algoritmo 7: QuantidadeNodos

Entrada: Ponteiro r para raiz.
Saída: Quantidade de nodos em r.

```
1 início
2     se (r = NULL) então
3         retorna 0
4     senão
5         total_esq ← QuantidadeNodos(r.esq)
6         total_dir ← QuantidadeNodos(r.dir)
7         retorna total_esq + total_dir + 1
```

Quantidade de Nodos

Código-Fonte

```
1 int quantidade_nodos(nodo *r) {
2     if (r == NULL)
3         return 0;
4     else {
5         int t_esq = quantidade_nodos(r->esq);
6         int t_dir = quantidade_nodos(r->dir);
7         return t_esq + t_dir + 1;
8     }
9 }
```

Altura

Introdução

Vamos implementar um algoritmo que calcula a **altura** da nossa árvore.

Altura

Introdução

A altura é calculada da seguinte forma:

- Verifica se o nodo é um nodo folha.
  - Caso sim, sua altura é zero.
  - Caso contrário, obtém-se a maior altura entre suas subárvores, e incrementa em um.
- A seguir, o pseudocódigo.

Altura

Pseudo-Código

Algoritmo 8: AlturaÁrvore

Entrada: Ponteiro r para raiz.
Saída: Altura da árvore com raiz em r.

```
1  início
2      se (r = NULL) então
3          retorna -1
4      senão
5          alt_esq ← AlturaÁrvore(r.esq)
6          alt_dir ← AlturaÁrvore(r.dir)
7          se (alt_esq > alt_dir) então
8              retorna alt_esq + 1
9          senão
10             retorna alt_dir + 1
```

Altura

Código-Fonte

```
1  int altura_arvore(nodo *r) {
2      if (r==NULL)
3          return -1;
4      else {
5          int alt_esq = altura_arvore(r->esq);
6          int alt_dir = altura_arvore(r->dir);
7          if (alt_esq > alt_dir)
8              return alt_esq + 1;
9          else
10             return alt_dir + 1;
11     }
12 }
```

Percurso

Introdução

Vamos implementar os algoritmos que percorrem uma árvore. São eles:

- Percurso **pré-ordem**: visita a raiz, o filho da esquerda e o filho da direita.
- Percurso **em-ordem**: visita o filho da esquerda, a raiz e o filho da direita.
- Percurso **pós-ordem**: visita o filho da esquerda, o filho da direita e a raiz.

Percurso PreOrdem

Pseudo-Código

Algoritmo 9: PercursoPréOrdem

Entrada: Ponteiro r para raiz.

```
1 início
2     se (r ≠ NULL) então
3         Imprima (r.item)
4         PercursoPréOrdem(r.esq)
5         PercursoPréOrdem(r.dir)
```

Percurso PreOrdem

Código-Fonte

```
1 void pre_ordem(Arvore *raiz){
2     if(raiz == NULL)
3         return;
4     if(*raiz != NULL){
5         printf("%d ",(*raiz)->item);
6         pre_ordem(&((*raiz)->esq));
7         pre_ordem(&((*raiz)->dir));
8     }
9 }
```

Percurso EmOrdem

Pseudo-Código

Algoritmo 10: PercursoEmOrdem

Entrada: Ponteiro r para raiz.

```
1 início
2     se (r ≠ NULL) então
3         PercursoEmOrdem(r.esq)
4         Imprima (r.item)
5         PercursoEmOrdem(r.dir)
```

Percurso EmOrdem

Código-Fonte

```
1 void em_ordem(Arvore *raiz){
2     if(raiz == NULL)
3         return;
4     if(*raiz != NULL){
5         em_ordem(&((*raiz)->esq));
6         printf("%d ",(*raiz)->item);
7         em_ordem(&((*raiz)->dir));
8     }
9 }
```

Percurso PósOrdem

Pseudo-Código

Algoritmo 11: PercursoPósOrdem

Entrada: Ponteiro r para raiz.

```
1 início
2     se (r ≠ NULL) então
3         PercursoPósOrdem(r.esq)
4         PercursoPósOrdem(r.dir)
5         Imprima (r.item)
```

Percurso EmOrdem

Código-Fonte

```
1 void pos_ordem(Arvore *raiz){
2     if(raiz == NULL)
3         return;
4     if(*raiz != NULL){
5         pos_ordem(&((*raiz)->esq));
6         pos_ordem(&((*raiz)->dir));
7         printf("%d ",(*raiz)->item);
8     }
9 }
```

Busca

Introdução

Vamos implementar duas versões do algoritmo de **busca**:

1. Uma versão **recursiva**
2. Uma versão **iterativa**

Busca

Pseudo-Código Versão Recursiva

Algoritmo 12: Busca Recursiva

Entrada: Ponteiro para a raiz r e o item x a ser procurado.
Saída: Retorna o nodo que contém o item x ou NULL caso não encontrado.

```
1 início
2     se (r = NULL) ou (x = r.item) então
3         retorna r
4     se (x > r.item) então
5         retorna Buscar(r.dir)
6     se (x < r.item) então
7         retorna Buscar(r.esq)
```

Busca

Código-Fonte

```
1  nodo* buscar_arvore_recursiva(Arvore *raiz, int valor){
2      if(raiz == NULL)
3          return NULL;
4      if (*raiz == NULL || valor == (*raiz)->item)
5        return (*raiz);
6      else
7          if (valor > (*raiz)->item)
8              return buscar_arvore_recursiva(&((*raiz)->dir), valor);
9          else
10             return buscar_arvore_recursiva(&((*raiz)->esq), valor);
11     return NULL;
12 }
```

Busca

Pseudo-Còdigo Versão Iterativa

Algoritmo 13: Busca Iterativa

Entrada: Ponteiro para a raiz r e o item x a ser procurado.
Saída: Retorna o nodo que contém o item x ou NULL caso não encontrado.

```
1  início
2      se (r = NULL) então
3          retorna NULL
4      senão
5          atual ← r
6          enquanto (atual ≠ NULL) faça
7              se (x = atual.item) então
8                  retorna atual
9              se (x > atual.item) então
10                 atual ← atual.dir
11             se (x < atual.item) então
12                 atual ← atual.esq
13         retorna atual
```

Busca

Código-Fonte Versão Iterativa

```
1  nodo* buscar_arvore_iterativa(Arvore *raiz, int valor){
2      if(raiz == NULL)
3          return NULL;
4      nodo *atual = *raiz;
5      while(atual != NULL){
6          if(valor == atual->item)
7              return atual;
8          if(valor > atual->item)
9              atual = atual->dir;
10         else
11             atual = atual->esq;
12     }
13     return NULL;
14 }
```

Remoção

Introdução

Vamos implementar o algoritmo **Remover**, que retira um elemento da nossa árvore. São necessárias duas funções:

1. Algoritmo **RemoverAtual**: recebe como parâmetro o endereço de um nodo da árvore (**atual**) a ser removido e retorna qual deverá ser o seu nodo substituto na árvore.
2. Algoritmo **RemoverValor**: recebe um ponteiro para a raiz da árvore e o valor do item a ser removido.

Remoção

Pseudo-Código RemoverAtual

Algoritmo 14: RemoverAtual

Entrada: Ponteiro **atual** para o nodo a ser removido.
Saída: Retorna o nodo substituto do nodo **atual**.

```
1  início
2      se (atual.esq = NULL) então
3          nodo2 ← atual.dir
4          DESALOCA_NODO(atual)
5          retorna nodo2
6      nodo1 ← atual
7      nodo2 ← atual.esq
8      enquanto (nodo2.dir ≠ NULL) faça
9          nodo1 ← nodo2
10         nodo2 ← atual.esq
11     se (nodo1 ≠ atual) então
12         nodo1.dir ← nodo2.esq
13         nodo2.esq ← atual.esq
14     nodo2.dir ← atual.dir
15     DESALOCA_NODO(atual)
16     retorna nodo2
```

Remoção

Código-Fonte RemoverAtual

```
1  nodo* remove_atual(nodo* atual) {
2      nodo *no1, *no2;
3      if(atual->esq == NULL){
4          no2 = atual->dir;
5          free(atual);
6          return no2;
7      }
8      no1 = atual;
9      no2 = atual->esq;
10     while(no2->dir != NULL){
11         no1 = no2;
12         no2 = no2->dir;
13     }
14     if(no1 != atual){
15         no1->dir = no2->esq;
16         no2->esq = atual->esq;
17     }
18     no2->dir = atual->dir;
19     free(atual);
20     return no2;
21 }
```

Remoção

Pseudo-Código Remover

Algoritmo 15: RemoverValor

Entrada: Ponteiro r para raiz, item x a ser removido na árvore.
Saída: Retorna V ou F.

```
1  início
2      se (r=NULL) então
3          retorna Falso
4      anterior ← NULL
5      atual ← r
6      enquanto (atual ≠ NULL) faça
7          se (valor = atual.item) então
8              se (atual = r) então
9                  r ← RemoveAtual(atual)
10             senão
11                 se (anterior.dir = atual) então
12                     anterior.dir ← RemoverAtual(atual)
13                 senão
14                     anterior.esq ← RemoverAtual(atual)
15             retorna Verdadeiro
16         senão
17             anterior ← atual
18             se (x > atual.item) então
19                 atual ← atual.dir
20             senão
21                 atual ← atual.esq
22     retorna Falso
```

Remoção

Código-Fonte Remover

```
1  int remover_valor(Arvore *raiz, int valor){
2      if(raiz == NULL)
3          return 0;
4      nodo* ant = NULL;
5      nodo* atual = *raiz;
6      while(atual != NULL){
7          if(valor == atual->item){
8              if(atual == *raiz)
9                  *raiz = remove_atual(atual);
10             else{
11                 if(ant->dir == atual)
12                     ant->dir = remove_atual(atual);
13                 else
14                     ant->esq = remove_atual(atual);
15             }
16             return 1;
17         }
18         ant = atual;
19         if(valor > atual->item)
20             atual = atual->dir;
21         else
22             atual = atual->esq;
23     }
24     return 0;
25 }
```

Apagar Árvore

Introdução

Vamos implementar o algoritmo **Apagar Árvore**, que exclui uma árvore, apagando todos os seus nodos.

Apagar Árvore

Apagar Árvore

Algoritmo 16: ApagarÁrvore

Entrada: Ponteiro *r* para raiz da árvore a ser deletada.

```
1 início
2     se (!ÁrvoreVazia(r)) então
3         ApagarÁrvore(r.esq)
4         ApagarÁrvore(r.dir)
5         DESALOCA_NODO(r)
```

Apagar Árvore

Código-Fonte Apagar Árvore

```
1  void apaga_nodos_rec(nodo* raiz){
2
3      if (raiz != NULL) {
4          printf("%d", raiz->item);
5          apaga_nodos_rec(raiz->dir);
6          apaga_nodos_rec(raiz->esq);
7          apaga_nodo_aux(raiz);
8      }
9  }
10
11 void apaga_arvore(Arvore *raiz) {
12     if ((*raiz) != NULL) {
13         apaga_nodos_rec(*raiz);
14         free(raiz);
15     }
16 }
```

Função Main

Introdução

Por fim, falta apenas criar uma **função Main** para manipular nossa estrutura de dados Árvore.

Função Main

Código-Fonte Apagar Árvore

```
int main() {
    int rand_max = 100; int vaux = 0;
    srand(42);
    Arvore *raiz;
    nodo *aux = NULL;
    raiz = cria_arvore();
    for(int i = 0; i<5; i++) {
        inserir_arvore_recursivo(raiz, rand() % rand_max);
        inserir_arvore_iterativo(raiz, rand() % rand_max);
    }
    printf("Percurso Pre-Ordem\n");
    pre_ordem(raiz);
    printf("\nPercurso Em-Ordem\n");
    em_ordem(raiz);
    printf("\nPercurso Pos-Ordem\n");
    pos_ordem(raiz);
    vaux = rand() % rand_max;
    aux = buscar_arvore_recursiva(raiz, vaux);
    if (aux != NULL)
        printf("\nBusca Recursiva %d : encontrou %d\n", vaux, aux->item);
    vaux = rand() % rand_max;
    aux = buscar_arvore_iterativa(raiz, vaux);
    if (aux != NULL)
        printf("\nBusca Recursiva %d : encontrou %d\n", vaux, aux->item);
    vaux = rand() % rand_max;
    remover_valor(raiz, vaux);
    printf("Total nodos: %d, Altura Arvore: %d", quantidade_nodos(*raiz), altura_arvore(*raiz));
    apaga_arvore(raiz);
}
```

Bibliografia

- Estrutura de dados descomplicada, André Backes, 1 ed. Editora Elsevier Brasil, 2017
- Material complementar disponível em `https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados`

Algoritmos e Estrutura de Dados

Árvores Binárias de Busca

prof. Frederico Santos de Oliveira

Universidade Federal de Mato Grosso
Faculdade de Engenharia