Estruturas de Dados

Aula 10 – Árvores Binárias

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia – Campus Feira de Santana

2026

Agenda

Árvores Binárias

Ordem 2 — no máximo dois filhos por nó

O que é uma Árvore Binária?

Definição

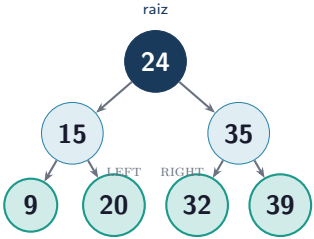
Árvore em que cada nó pode ter **zero, um ou dois filhos** (filho esquerdo e filho direito). Cada nó armazena um **ID** e dois ponteiros: LEFT e RIGHT. [Cormen et al. 2009]

Estrutura recursiva

Uma árvore binária é:

- **Vazia** (sem elementos); ou
- Um **nó raiz** com dois ponteiros para **subárvore esquerda** e **subárvore direita** (cada uma também é uma árvore binária)

Vantagem




```
typedef struct noArvore{
  int chave;
  struct noArvore *filhoDir;
  struct noArvore *filhoEsq;
}NO;
```

Altura da Árvore Binária

Tempo máximo de busca

Altura da Árvore Binária

▪ Definição

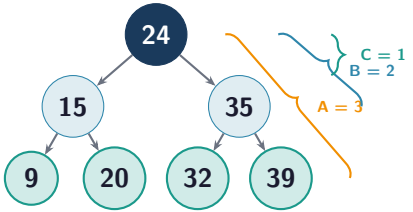
O **maior caminho** da raiz até um nó-folha. Define o **tempo máximo de busca** de um elemento. Por isso árvores binárias são ideais para Bancos de Dados.

▪ Altura $\neq$ Nível

Nível: posição de um nó na hierarquia (raiz = nível 0).

Altura: número de *nós* no caminho entre dois nós.

Exemplo: entre 24 e 39 há 3 nós (323539), logo **altura = 3**.



Árvore Binária de Busca

Ordenação e Busca Eficiente

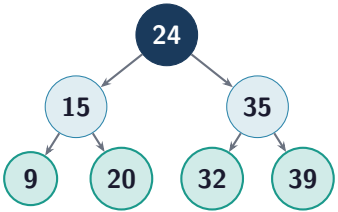
Árvore Binária de Busca (ABB)

▪ Propriedade fundamental

Para todo nó N:

- Todos os nós da **subárvore esquerda** têm valor **menor** que N
- Todos os nós da **subárvore direita** têm valor **maior** que N

esquerda $<$ pai $<$ direita



▪ Por que buscar é mais eficiente?

A cada comparação, descartamos **metade** da árvore — decidimos se vamos para a subárvore à esquerda ou à direita. Complexidade de busca: $O(\log N)$.

Inserção Ordenada

Mantendo a propriedade da ABB

Inserção na ABB

Regra

A inserção percorre a árvore comparando o novo valor com cada nó:

- Menor → vai para a **esquerda**
- Maior → vai para a **direita**
- Repete até encontrar uma posição vazia (folha)

```c
// Função para criar novo nó
No* criarNo(int valor) {
    No* novo = (No*) malloc(sizeof(No));
    if (novo == NULL) {
        printf("Erro de memória!\n");
        exit(1);
    }
    novo->valor = valor;
    novo->esq = NULL;
    novo->dir = NULL;
    return novo;
}

// Inserção ordenada
No* inserir(No* raiz, int valor) {
    if (raiz == NULL) {
        return criarNo(valor);
    }

    if (valor < raiz->valor) {
        raiz->esq = inserir(raiz->esq, valor);
    } else if (valor > raiz->valor) {
        raiz->dir = inserir(raiz->dir, valor);
    } else {
        printf("Valor já existe na árvore!\n");
    }

    return raiz;
}
```

Remoção

Três casos possíveis

Remoção — Os 3 Casos

▪ Caso 1: Nó folha

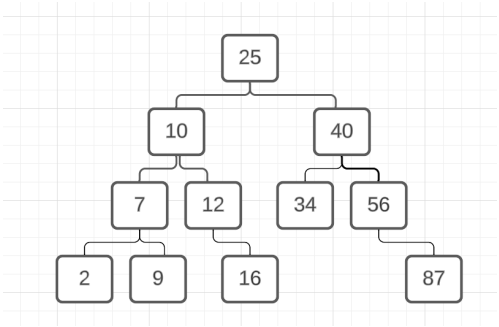
Sem filhos: simplesmente remove o nó e ajusta o ponteiro do pai para `NULL`.

▪ Caso 2: Um filho

O pai do nó removido passa a apontar diretamente para o filho do próprio nó.

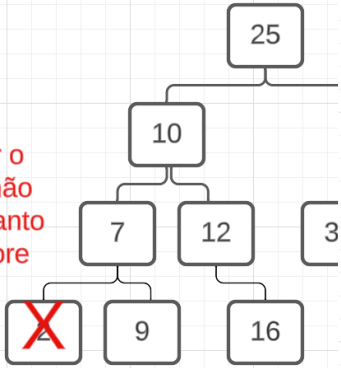
▪ Caso 3: Dois filhos

Substitui por **maior da subárvore esquerda** ou por **menor da subárvore direita**.

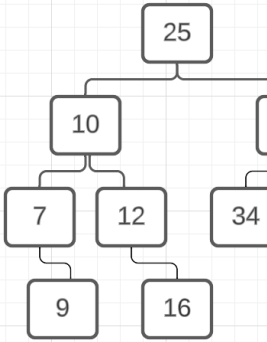


Remoção — Caso 1: Nó Folha

Deseja-se excluir o elemento 2, que não possui filhos, portanto é nó folha da árvore

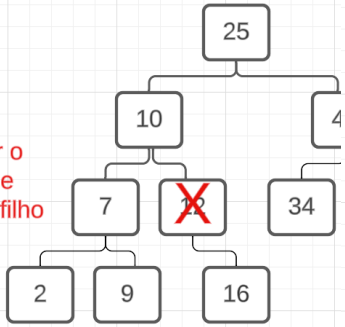


Basta apagar o nó e setar o filho esquerdo do pai, nesse caso o nó 7, para nulo

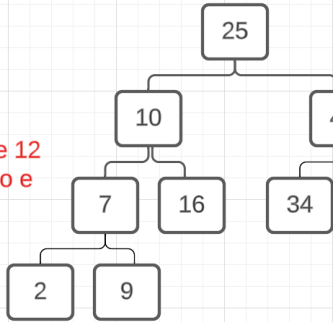


Remoção — Caso 2: Um Filho

Deseja-se excluir o elemento 12, que possui apenas um filho



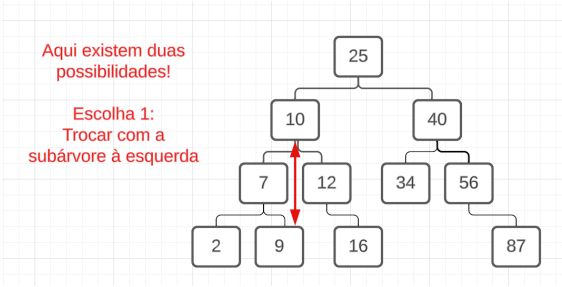
Transfere o filho de 12 para a sua posição e apaga o nó 12.

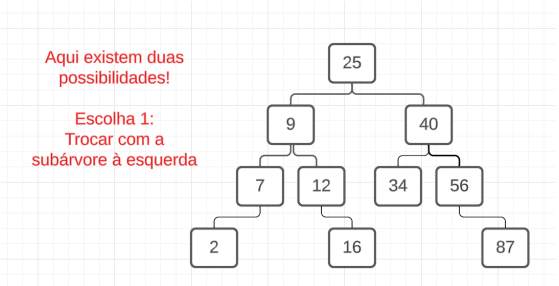


Remoção — Caso 3: Dois Filhos (parte 1)

▪ Substituir pela subárvore esquerda

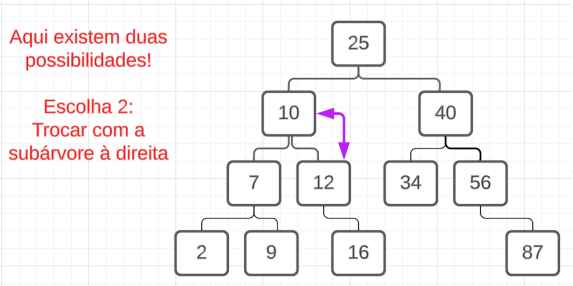
Pega o **maior elemento da subárvore esquerda** (o mais à direita dela) como substituto.

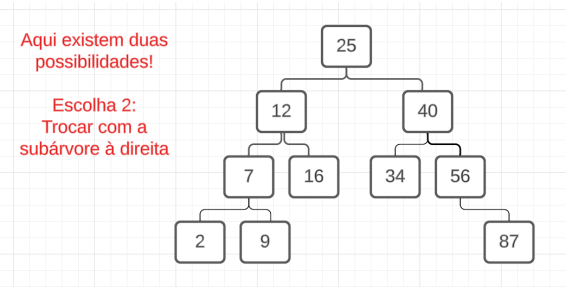




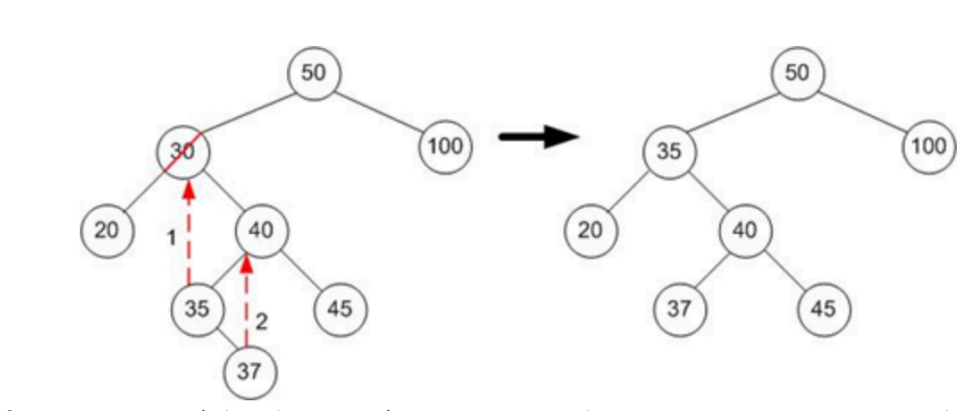
- **Substituir pela subárvore direita**

Pega o **menor elemento da subárvore direita** (o mais à esquerda dela) como substituto.





Remoção — Realocar filhos do substituto



▪ Atenção

O substituto pode ter filhos próprios — eles precisam ser realocados mantendo a propriedade da ABB.

Remoção — Código

```c
// Remoção de nó
No* remover(No* raiz, int valor) {
    if (raiz == NULL) {
        printf("Valor não encontrado!\n");
        return NULL;
    }

    if (valor < raiz->valor) {
        raiz->esq = remover(raiz->esq, valor);
    } else if (valor > raiz->valor) {
        raiz->dir = remover(raiz->dir, valor);
    } else {
        // Caso 1: sem filhos
        if (raiz->esq == NULL && raiz->dir == NULL) {
            free(raiz);
            return NULL;
        }
        // Caso 2: um filho
        else if (raiz->esq == NULL) {
            No* temp = raiz->dir;
            free(raiz);
            return temp;
        }
        else if (raiz->dir == NULL) {
            No* temp = raiz->esq;
            free(raiz);
            return temp;
        }
        // Caso 3: dois filhos
        else {
            No* temp = maisEsqDaDireita(raiz->dir);
            raiz->valor = temp->valor;
            raiz->dir = remover(raiz->dir, temp->valor);
        }
    }

    return raiz;
}
```

```c
// Encontrar menor valor da subárvore da direita (usado na remoção)
No* maisEsqDaDireita(No* raiz) {
    while (raiz && raiz->esq != NULL) {
        raiz = raiz->esq;
    }
    return raiz;
}
```

Percursos

Pré-Ordem, Em-Ordem e Pós-Ordem

Os 3 Percursos em Árvore Binária

▪ Pré-Ordem (RED)

Raiz **E**squerda **D**ireita

Visita a raiz **antes** das subárvores.

Útil para copiar/serializar a árvore.

▪ Em-Ordem (ERD)

Esquerda **R**aiz **D**ireita

Visita a raiz **entre** as subárvores.

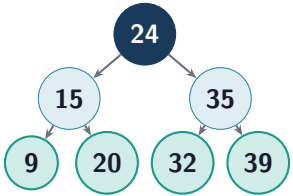
Resultado: elementos em **ordem crescente**.

▪ Pós-Ordem (EDR)

Esquerda **D**ireita **R**aiz

Visita a raiz **depois** das subárvores.

Útil para deletar a árvore.



▪ Pré-Ordem

```
void preOrdem(NO *nodo){
  if(nodo){
    printf("%d\n",nodo->chave);
    preOrdem(nodo->filhoEsq);
    preOrdem(nodo->filhoDir);
  }
}
```

▪ Em-Ordem

```
void emOrdem(NO *nodo){
  if(nodo){
    emOrdem(nodo->filhoEsq);
    printf("%d\n",nodo->chave);
    emOrdem(nodo->filhoDir);
  }
}
```

▪ Pós-Ordem

```
void posOrdem(NO *nodo){
  if(nodo){
    posOrdem(nodo->filhoEsq);
    posOrdem(nodo->filhoDir);
    printf("%d\n",nodo->chave);
  }
}
```

Imprimir e Menu

```c
void imprimir(NO *nodo, int nivel){
  int i;
  if(nodo != NULL) {
    imprimir(nodo->filhoDir, nivel + 1);
    for(i = 0; i <= nivel-1; i++) printf("   ");
    printf("+--");
    printf("%d\n",nodo->chave);
    imprimir(nodo->filhoEsq, nivel + 1);
  }
  return;
}
```

```c
int menu(void){
  int opcao;
  printf("Opções:\n");
  printf("-------\n");
  printf("0. Insere elementos na Árvore Binária Ordenada\n");
  printf("1. Mostra Árvore Binária Ordenada em Pré-ordem\n");
  printf("2. Mostra Árvore Binária Ordenada em Pós-ordem\n");
  printf("3. Mostra Árvore Binária Ordenada Em Ordem\n");
  printf("4. Desenha a Árvore Binária Ordenada da raiz às folhas\n");
  printf("5. Remove elementos da Árvore Binária Ordenada\n");
  printf("6. Busca elementos na Árvore Binária Ordenada\n");
  printf("7. Fechar\n\n");

  do{
    printf("Escolha [0,1,2,3,4,5,6 ou 7]: ");
    scanf("%d",&opcao);
    printf("\n\n");
  }
  while((opcao < 0) && (opcao > 7));
  return opcao;
}
```

Main — parte 1 e 2

```c
void main() {
  NO *raiz = NULL, *nodo;
  int escolha, chave;
  do{
    escolha = menu();
    switch(escolha){
        case 0:
            printf("Insere elementos na Árvore Binária Ordenada \n\r");
            while(escolha == 0){
                printf("Digite o elemento que deseja inserir ou -1 para sair: ");
                scanf("%d",&chave);
                if (chave != -1)
                    raiz = insereNaArvore(raiz,chave);
                else break;
            }
            break;
        case 1:
            printf("Percorre a Árvore Binária Ordenada e mostra no modo Pré-ordem\n\r");
            preOrdem(raiz);
            break;
        case 2:
            printf("Percorre a Árvore Binária Ordenada e mostra no modo Pós-ordem\n\r");
            posOrdem(raiz);
            break;
        case 3:
            printf("Percorre a Árvore Binária Ordenada e mostra no modo Em Ordem\n\r");
            emOrdem(raiz);
            break;
        case 4:
            printf("Mostra Árvore Binária Ordenada (raiz à esquerda e folhas à direita)\n\r");
            imprimir(raiz, 0);
            break;
        case 5:
            printf("Remove elementos da Árvore Binária Ordenada \n\r");
            while(escolha == 5){
                printf("Digite o elemento que deseja remover ou -1 para sair: ");
                scanf("%d",&chave);
                if (chave != -1)
                    raiz = removeDaArvore(raiz,chave);
                else break;
            }
            break;
        case 6:
            printf("Busca elemento na Árvore Binária Ordenada \n\r");
            while(escolha == 6){
                printf("Digite o elemento que deseja buscar ou -1 para sair: ");
                scanf("%d",&chave);
                if (chave != -1)
                    nodo = buscaNaArvore(raiz,chave);
                else break;
            }
            break;
    }
    printf("\n\nDigite qualquer tecla...");
    getchar();
  }
  while(escolha != 7);
  destroi(raiz);
```

Referências I

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262032937.