Faculdade Anhanguera – Marte

Algoritmo e Estrutura de Dados

Aiman de Carvalho Gonçalves

Camile Pessutti Figueiredo

Daniel Freitas da silva

Henry Gabriel Marinho Souza

Pesquisa linguagem em tipo C

São Paulo

2025

Sumário

Arrayas, funções, alocações, dinâmica, ponteiro e struct.....3

Lista encadeada, pilhaa e fila.....9

Árvore binária de busca (BST); algoritmos de ordenação (Bubble Sort, insertions Sort).....22

Tabela hast com encadeamento, busca linear e busca binária.....34

Referências.....37

Arrayas, funções, alocações, dinâmica, ponteiro e struct

Arrays

Um array é uma estrutura de dados que consiste em uma coleção de elementos do mesmo tipo, armazenados de forma sequencial na memória. Cada elemento é identificado por um índice, cuja contagem inicia-se em zero.

Arrays são utilizados para armazenar e manipular conjuntos de dados do mesmo tipo, como números ou textos. Por exemplo, em uma lista de nomes, cada nome ocupa uma posição no array. Isso facilita o acesso e a organização dos dados.

Exemplo de Array

```
#include <stdio.h>

int main() {
    char alunos[][20] = {
        "Laura",
        "Maria",
        "Thiago",
        "José",
        "Vitória"
    };

    printf("%s\n", alunos[2]);  // Saída: Thiago
    return 0;
}
```

Funções

Uma função é um bloco de código que realiza uma tarefa específica, podendo ou não retornar um valor. As funções permitem modularizar o código, tornando-o mais organizado e reutilizável.

- Sem retorno: apenas executa uma ação.
- Com retorno: processa dados e devolve um resultado.

Exemplo Função sem retorno

```
#include <stdio.h>

void exibir_mensagem() {
    printf("Bem-vindo!\n");
}

int main() {
    exibir_mensagem();
    return 0;
}
```

Exemplo Função com retorno

```
#include <stdio.h>

int quadrado(int numero) {
    return numero * numero;
}

int main() {
    int resultado = quadrado(5);
    printf("O quadrado de 5 é %d\n", resultado);
    return 0;
}
```

Exemplo Soma

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Digite a: ");
    scanf("%d", &a);

    printf("Digite b: ");
    scanf("%d", &b);

    printf("Soma igual: %d\n", a + b);

    return 0;
}
```

Alocação Dinâmica

A alocação dinâmica permite reservar espaço na memória em tempo de execução, tornando o programa mais flexível em relação à quantidade de dados.

As funções usadas para alocação dinâmica são:
- malloc() → aloca memória.
- calloc() → aloca e inicializa memória.
- realloc() → redimensiona uma área de memória.
- free() → libera a memória.

Exemplo de Alocação Dinâmica

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *vetor;
    vetor = (int *)malloc(5 * sizeof(int));

    if (vetor == NULL) {
        printf("Erro de alocação.\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        vetor[i] = i + 1;
        printf("%d ", vetor[i]);
    }

    free(vetor);
    return 0;
}
```

Ponteiros

Um ponteiro é uma variável que armazena o endereço de memória de outra variável. Ele permite acessar e modificar diretamente valores armazenados na memória.

- Declaração: int *p; → um ponteiro para inteiro.
- Operador &: obtém o endereço da variável.
- Operador *: acessa o valor no endereço.

Exemplo de Ponteiro

```
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;

    printf("Valor de x: %d\n", *p); // Saída: 10
    return 0;
}
```

Struct

Uma struct (estrutura) permite agrupar diferentes tipos de dados sob um mesmo nome, representando um objeto do mundo real, como uma pessoa, produto ou aluno.

Características:
- Agrupa variáveis de tipos diferentes.
- Facilita organização e manipulação de dados complexos.
- Pode ser usada com ponteiros e arrays.
- Pode ser aninhada (struct dentro de struct).
- Permite alocação dinâmica.

Exemplo de Struct

```
#include <stdio.h>
#include <string.h>

struct Pessoa {
   char nome[50];
   int idade;
};

int main() {
   struct Pessoa lista[3];

   for (int i = 0; i < 3; i++) {
      printf("Digite o nome da pessoa %d: ", i + 1);
      scanf("%s", lista[i].nome);
      printf("Digite a idade: ");
      scanf("%d", &lista[i].idade);
   }

   printf("\nLista de pessoas:\n");
   for (int i = 0; i < 3; i++) {
      printf("%s tem %d anos\n", lista[i].nome, lista[i].idade);
   }

   return 0;
}
```

Lista encadeada, pilhaa e fila

Lista encadeada

Uma lista encadeada é uma estrutura de dados dinâmica que consiste em nós (ou elementos), onde cada nó armazena um valor e um ponteiro para o próximo nó da lista.

Diferente de Arrays, as listas encadeadas não precisam de um tamanho fixo e são eficientes em inserções e remoções de elementos em posições intermediárias.

Vantagens	Desvantagens
Alocação dinâmica de memória	Acesso sequencial (não é possível acessar diretamente um elemento como num Array)
Inserções e remoções eficientes (especialmente no início da lista)	Uso extra de memória (devido aos ponteiros)
Tamanho da lista pode crescer conforme necessário	Operações mais lentas em busca de elementos (comparando a Arrays)

Tipo	Característica
Lista simples	Cada nó aponta para o próximo
Lista duplamente encadeada	Cada nó aponta para o próximo e o anterior
Lista Circular	O último nó aponta para o primeiro

Pilha

Uma pilha é uma estrutura de dados do tipo LIFO (Last In, First Out)**,** ou seja, o último elemento inserido é o primeiro a ser removido.

Imagine uma pilha de pratos: você sempre coloca (empilha) e retira (desempilha) o prato do topo da pilha.

Operação	Descrição
Push	Insere (empilha) um novo elemento no top
POP	Remove (desempilha) o elemento no topo
TOP ou PEEK	Retorna o elemento do topo (sem remover)
Isempty	Verificar se a pilha está vazia
IsFull	Verificar se a pilha está cheia (no caso da pilha estática)

Aplicação da pilha

- Controle de chamadas de função (pilha de execução)
- Desfazer/refazer em editores de texto
- Verificação de expressões balanceadas (parênteses)
- Algoritmos de conversão de notações (infixa → pós-fixa)
- Backtracking em jogos ou soluções de labirintos

Pilha Estática (exemplo):

```
#include <stdio.h>
#define TAM 5
typedef struct {
    int dados[TAM];
    int topo;
} Pilha;
void inicializar(Pilha *p) {
    p->topo = -1;
}
int pilhaVazia(Pilha *p) {
    return p->topo == -1;
}
int pilhaCheia(Pilha *p) {
    return p->topo == TAM - 1;
}
void push(Pilha *p, int valor) {
    if (!pilhaCheia(p)) {
```

```
        p->dados[++p->topo] = valor;
    } else {
        printf("Pilha cheia!\n");
    }
}
int pop(Pilha *p) {
    if (!pilhaVazia(p)) {
        return p->dados[p->topo--];
    } else {
        printf("Pilha vazia!\n");
        return -1;
    }
}
void imprimirPilha(Pilha *p) {
    for (int i = p->topo; i >= 0; i--) {
        printf("%d\n", p->dados[i]);
    }
}

int main() {
    Pilha p;
    inicializar(&p);
    push(&p, 10);
    push(&p, 20);
    push(&p, 30);
```

```
    printf("Conteúdo da pilha:\n");
    imprimirPilha(&p);
    pop(&p);
    printf("\nApós pop:\n");
    imprimirPilha(&p);
    return 0;
}
```

Pilha Dinâmica (com ponteiros e alocação dinâmica)

```
#include <stdio.h>
#include <stdlib.h>
typedef struct No {
    int valor;
    struct No *prox;
} No;
typedef struct {
    No *topo;
} Pilha;
void inicializar(Pilha *p) {
    p->topo = NULL;
}

int pilhaVazia(Pilha *p) {
    return p->topo == NULL;
}
void push(Pilha *p, int valor) {
```

```
    No *novo = (No *)malloc(sizeof(No));
    novo->valor = valor;
    novo->prox = p->topo;
    p->topo = novo;
}
int pop(Pilha *p) {
    if (pilhaVazia(p)) {
        printf("Pilha vazia!\n");
        return -1;
    }
    No *remover = p->topo;
    int valor = remover->valor;
    p->topo = remover->prox;
    free(remover);
    return valor;
}
void imprimirPilha(Pilha *p) {
    No *aux = p->topo;
    while (aux != NULL) {
        printf("%d\n", aux->valor);
        aux = aux->prox;
    }
}
int main() {
    Pilha p;
```

```
    inicializar(&p);

    push(&p, 5);

    push(&p, 15);

    push(&p, 25);

    printf("Conteúdo da pilha:\n");

    imprimirPilha(&p);

    pop(&p);

    printf("\nApós pop:\n");

    imprimirPilha(&p);

    return 0;

}
```

Fila

A estrutura de dados "fila" é amplamente utilizada em computação para armazenar dados de maneira ordenada, seguindo o princípio FIFO (First In, First Out). Neste trabalho, exploraremos o conceito de filas, suas aplicações, e implementações básicas usando a linguagem de programação C.

Aplicações de FILA

- **Sistemas operacionais**: gerenciamento de processos.
- **Redes de computadores**: transmissão de pacotes.
- **Atendimento ao cliente**: chamadas em call centers.
- **Simulações**: filas em bancos, supermercados, etc.

Exemplo de fila estática

```
#include <stdio.h>

#define TAM 5



typedef struct {

    int dados[TAM];
```

```
    int inicio;
    int fim;
} Fila;

void inicializarFila(Fila *f) {
    f->inicio = 0;
    f->fim = 0;
}

int filaVazia(Fila *f) {
    return f->inicio == f->fim;
}

int filaCheia(Fila *f) {
    return f->fim == TAM;
}

void enfileirar(Fila *f, int valor) {
    if (!filaCheia(f)) {
        f->dados[f->fim++] = valor;
    } else {
        printf("Fila cheia!\n");
    }
}
```

```
int desenfileirar(Fila *f) {
    if (!filaVazia(f)) {
        return f->dados[f->inicio++];
    } else {
        printf("Fila vazia!\n");
        return -1;
    }
}

void imprimirFila(Fila *f) {
    for (int i = f->inicio; i < f->fim; i++) {
        printf("%d ", f->dados[i]);
    }
    printf("\n");
}

int main() {
    Fila fila;
    inicializarFila(&fila);

    enfileirar(&fila, 10);
    enfileirar(&fila, 20);
    enfileirar(&fila, 30);
```

```
    imprimirFila(&fila);

    desenfileirar(&fila);
    imprimirFila(&fila);

    return 0;
}
```

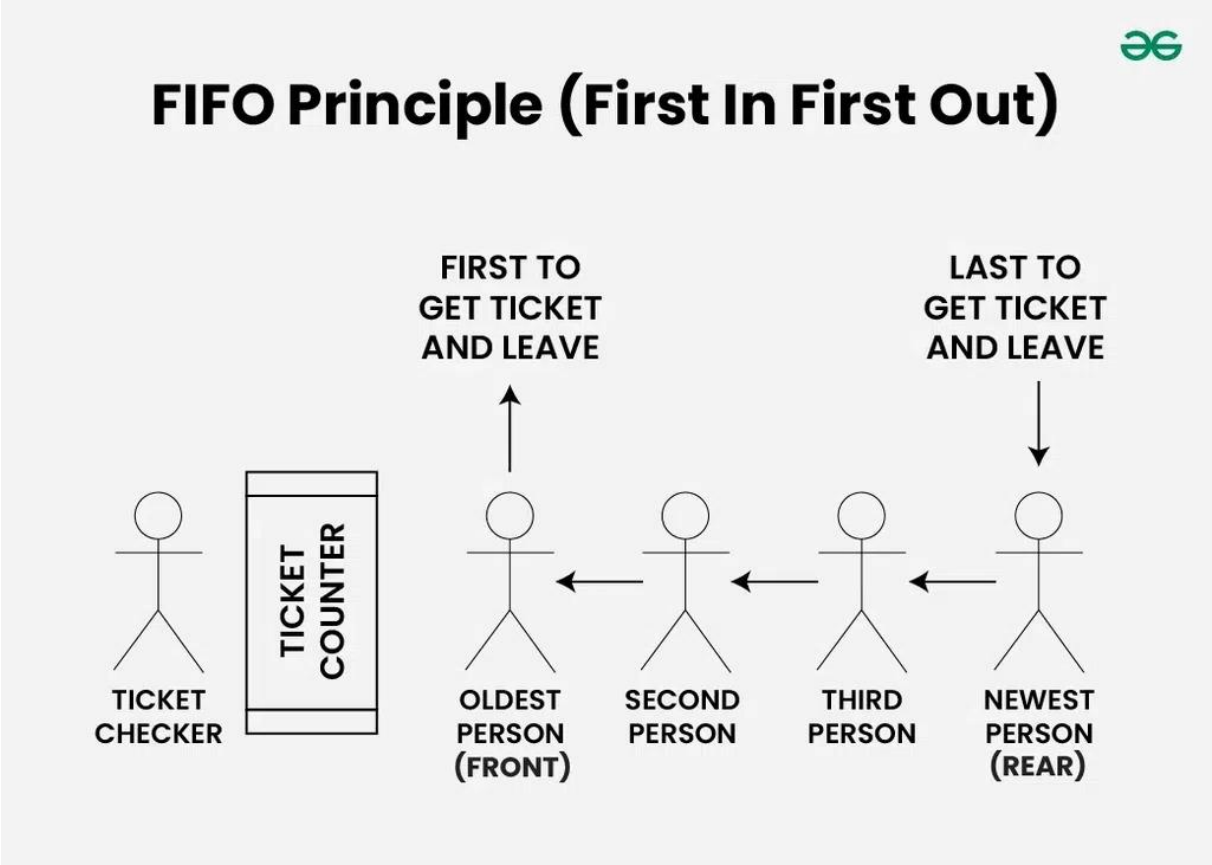
Fila dinâmica (usando ponteiros e alocação dinâmica)

```
#include <stdio.h>
#include <stdlib.h>
typedef struct No {
    int valor;
    struct No *prox;
} No;
typedef struct {
    No *inicio;
    No *fim;
} Fila;
void inicializar(Fila *f) {
    f->inicio = NULL;
```

```
    f->fim = NULL;
}
int filaVazia(Fila *f) {
    return f->inicio == NULL;
}
void enfileirar(Fila *f, int valor) {
    No *novo = (No *)malloc(sizeof(No));
    novo->valor = valor;
    novo->prox = NULL;
    if (filaVazia(f)) {
        f->inicio = novo;
    } else {
        f->fim->prox = novo;
    }

    f->fim = novo;
}
int desenfileirar(Fila *f) {
    if (filaVazia(f)) {
        printf("Fila vazia!\n");
        return -1;
    }
    No *remover = f->inicio;
    int valor = remover->valor;
```

```
    f->inicio = remover->prox;

    if (f->inicio == NULL)
        f->fim = NULL;
    free(remover);
    return valor;
}
void imprimirFila(Fila *f) {
    No *aux = f->inicio;
    while (aux != NULL) {
        printf("%d ", aux->valor);
        aux = aux->prox;
    }
    printf("\n");
}
int main() {
    Fila f;
    inicializar(&f);
    enfileirar(&f, 5);
    enfileirar(&f, 15);
    enfileirar(&f, 25);
    imprimirFila(&f);

    desenfileirar(&f);
```

```
    imprimirFila(&f);
    return 0;
}
```



https://www.geeksforgeeks.org/introduction-to-queue-data-structure-and-algorithm-tutorials/

Vantagens	Desvantagens
Organização lógica	Limitação de tamanho (fila estática)
Processamento justo	Complexidade de manutenção (fila circular)
Fácil implementação	Baixa flexibilidade para acesso aleatório
Utilização prática ampla	Custo de movimentação (fila estática simples)
Facilidade para simulação	Gerenciamento de memória (fila dinâmica)

Aplicação das filas:

1. No cotidiano
    - ✓ Supermercado, bancos, serviços públicos;
    - ✓ Trânsitos e controle de fluxo de veículo;
    - ✓ Atendimento telefônico e suporte técnico.
2. Na computação
    - ✓ Gerenciamento de tarefas em sistemas operacionais;
    - ✓ Impressão de documentos (spooling);
    - ✓ Algoritmo de busca de grafos (como BFS);
    - ✓ Transmissão de dados em rede (buffers de pacotes).
3. Na indústria
    - ✓ Linhas de produção;
    - ✓ Logística e distribuição de mercadorias.

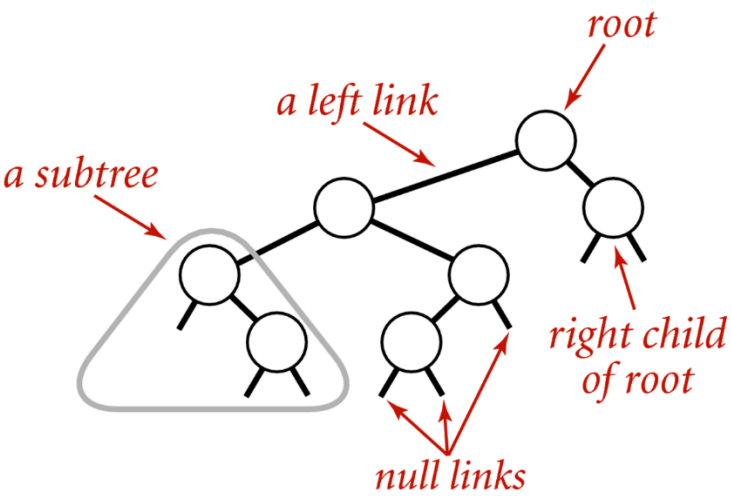
Operação de fila

1. **Enfileirar**: Adiciona um elemento ao final (final) da fila. Se a fila estiver cheia, ocorrerá um erro de estouro.
2. **Dequeue**: Remove o elemento do início da fila. Se a fila estiver vazia, ocorre um erro de estouro negativo.
3. **Peek/Front**: Retorna o elemento na frente sem removê-lo.
4. **Tamanho**: Retorna o número de elementos na fila.
5. **isEmpty**: Retorna truese a fila estiver vazia, caso contrário false.
6. **isFull**: Retorna truese a fila estiver cheia, caso contrário false.

Árvore binária de busca (BST); algoritmos de ordenação (Bubble Sort, insertions Sort)

O que é uma árvore binária?

- Antes das BSTs, precisamos discutir algo mais geral: as BTs
- BT = binary tree = árvore binária.
- Cada nó tem no máximo dois filhos: um *esquerdo* e um *direito*.



Anatomy of a binary tree

- Exemplo: uma BT com 256 nós.
- Uma árvore binária na natureza:

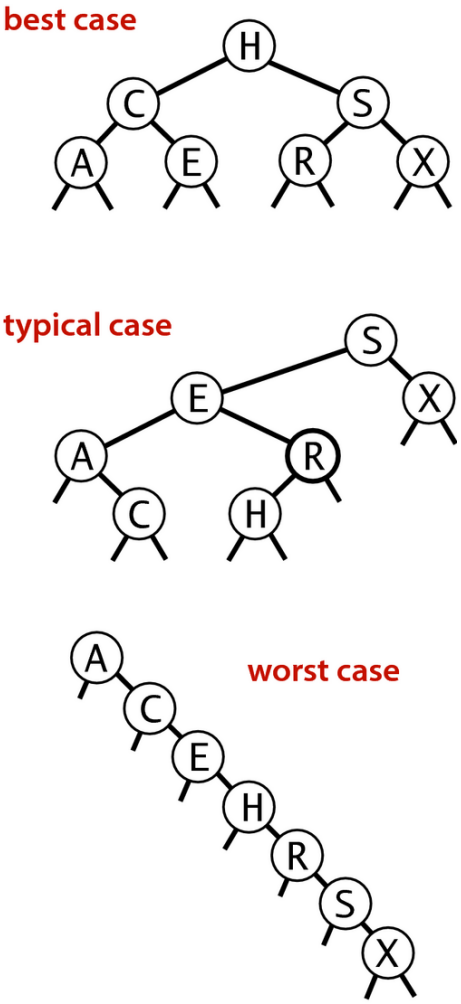
- Definição de um nó de uma árvore binária:

```
private class Node {
  private Node left, right;
}
```

- A *raiz* (root) é o único nó que não é filho de outro. A árvore é *vazia* se root == null.

```
private Node root;
```

- BTs são estruturas *recursivas*: cada nó da BT é raiz de uma sub-BT.
- A *profundidade* (depth) de um nó de uma BT é o número de links no caminho que vai da raiz até o nó.
- A *altura* (height) de uma BT é o máximo das profundidades dos nós, ou seja, a profundidade do nó mais profundo.
- Uma BT com N nós, tem altura no máximo $N-1$ e no mínimo $\lfloor \lg N \rfloor$. Se a altura estiver perto de $\lg N$, a BT é *balanceada*.
- Exemplos:

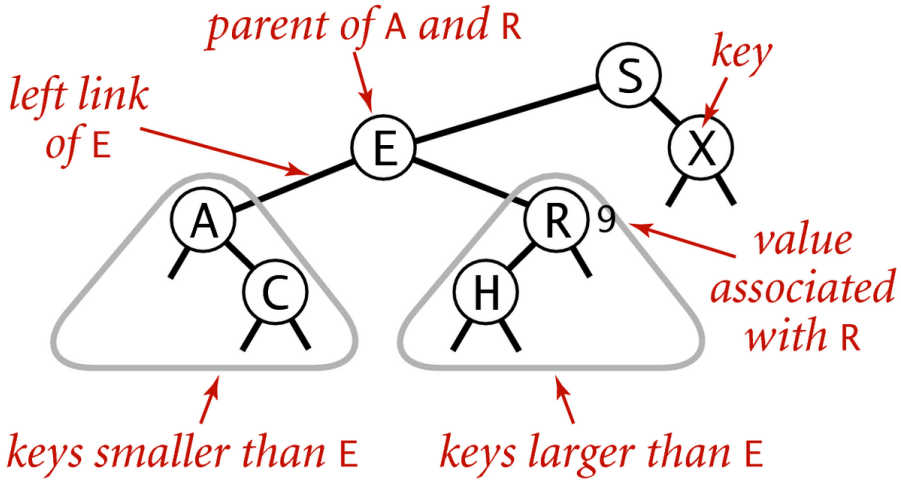


BST possibilities

- O *comprimento interno* (internal path length) de uma BT é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó. (Esse conceito é usado para estimar o desempenho esperado de TSs implementadas com BSTs).

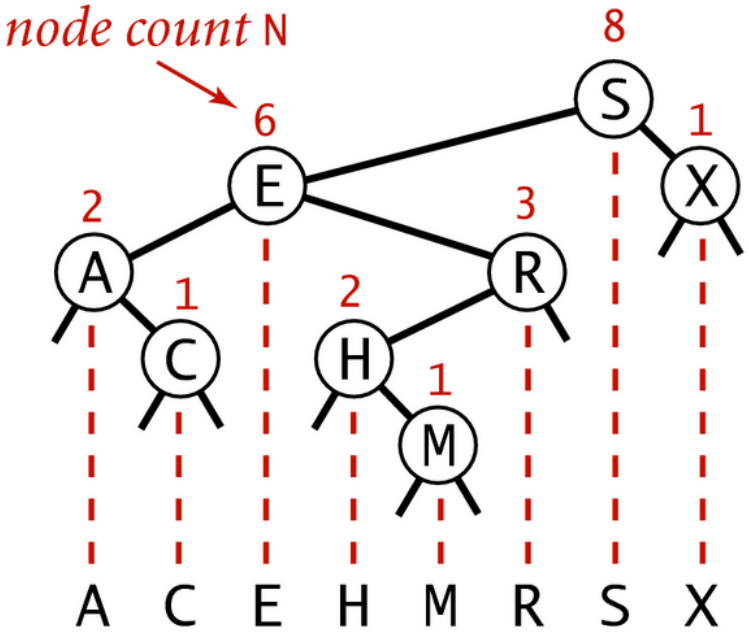
O que é uma árvore binária de busca?

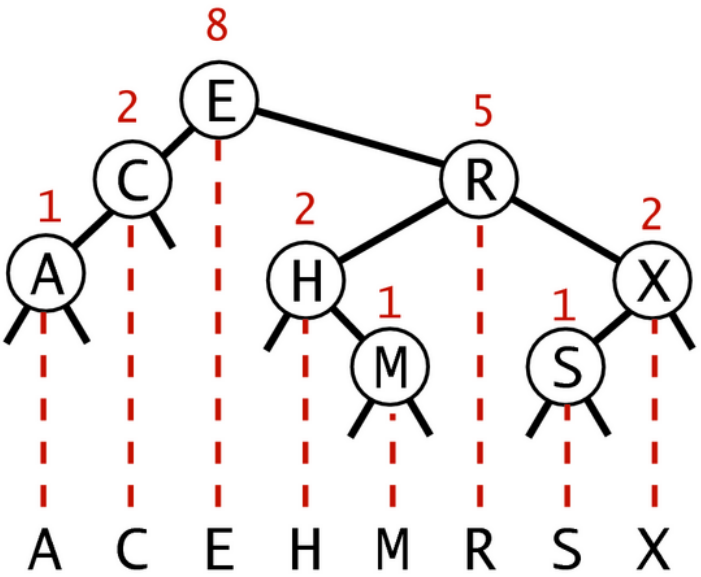
- Uma árvore binária *de busca* (binary search tree) é um tipo especial de BT: para cada nó x, todos os nós na subárvore esquerda de x têm chave menor que x.key e todos os nós na subárvore direita de x têm chave maior que x.key.
- BST = binary search tree = árvore binária de busca.
- As chaves de uma BST precisam ser *comparáveis*.
- Nosso exemplo padrão:



Anatomy of a binary search tree

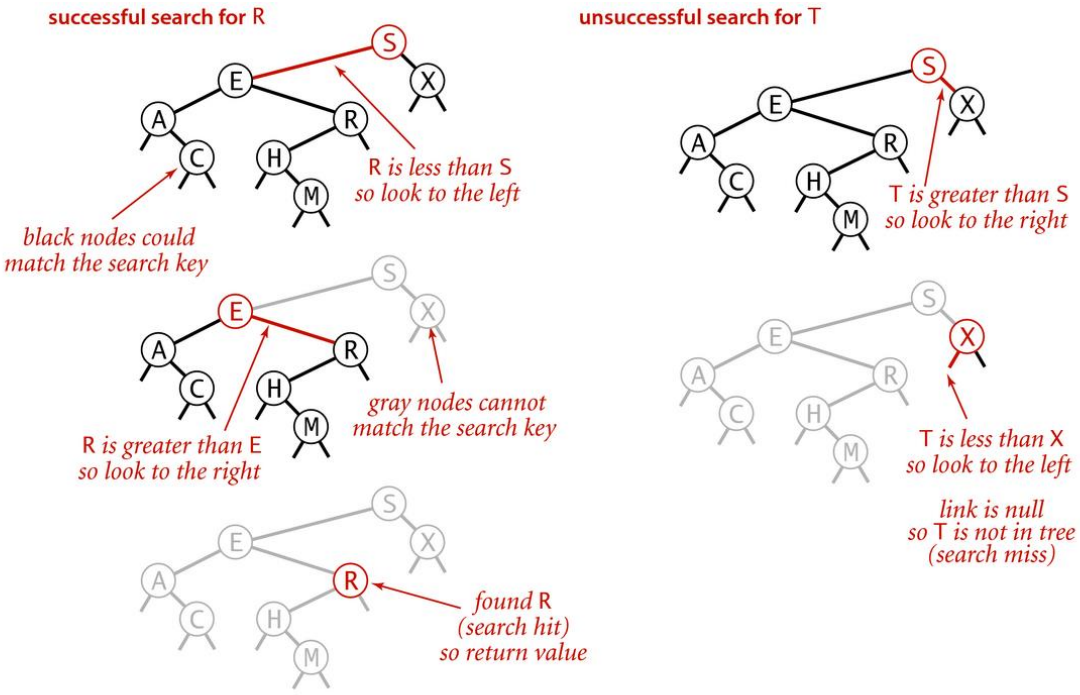
- Exemplo: Duas BSTs que representam o mesmo conjunto de chaves:





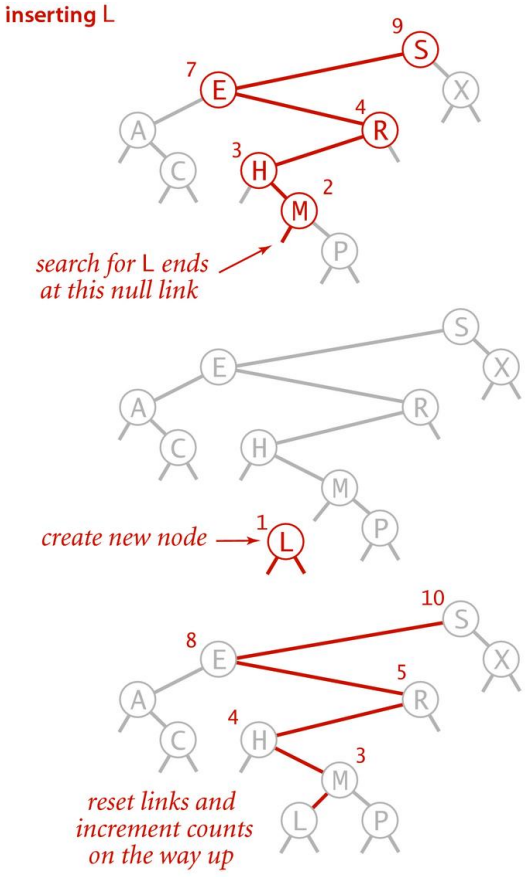
Two BSTs that represent the same set of keys

- Busca (get) numa BST: o processo é *muito* parecido com a busca binária em um vetor ordenado:



Search hit (left) and search miss (right) in a BST

- Inserção (put) em uma BST: o processo é muito mais barato que a inserção em um vetor ordenado, pois não envolve movimentação de dados.



Insertion into a BST

Algoritmos

Algoritmos descrevem passo a passo os procedimentos para chegar a uma solução de um problema e podem ser representados de três formas:

- A forma de descrição narrativa, na qual se usa a linguagem nativa de quem escreve. Essa forma não segue um padrão definido e pode sofrer várias interpretações por quem lê;
- Outra forma de representar um algoritmo é o fluxograma, uma representação visual que utiliza símbolos que são figuras geométricas, cada uma com sua função específica. Essa representação, como o próprio nome diz, mostra o fluxo do algoritmo e também elimina as várias interpretações que a descrição narrativa permitia sobre um algoritmo;
- Por último, existe a linguagem algoritma (Pseudocódigo ou Portugol) que é a que mais se aproxima da estrutura de uma **linguagem estruturada**.

Um tipo de algoritmo muito usado na resolução de problemas computacionais são os **algoritmos de ordenação**, que servem para ordenar/organizar uma lista de números ou palavras de acordo com a sua necessidade. As linguagens de programação já possuem métodos de ordenação, mas é bom saber como funcionam os algoritmos, pois há casos de problemas em que o algoritmo de ordenação genérico não resolve, às vezes é necessário modificá-lo.

Definição de Algoritmos

O **Algoritmo** é um esquema de resolução de um problema. Pode ser implementado com qualquer sequência de valores ou objetos que tenham uma lógica infinita (por exemplo, a língua portuguesa, a linguagem Pascal, a linguagem C, uma sequência numérica, um conjunto de objetos tais como lápis e borracha), ou seja, qualquer coisa que possa fornecer uma sequência lógica.

Podemos ilustrar um algoritmo pelo exemplo de uma receita culinária, embora muitos algoritmos sejam mais complexos. Um Algoritmo mostra passo a passo os procedimentos necessários para resolução de um problema.

Descrição Narrativa

A descrição narrativa é o uso da sua língua nativa para descrição dos passos para se resolver um problema.

A vantagem dessa forma de representação é que qualquer um pode fazê-la sem ter conhecimentos avançados.

A desvantagem é que não há um padrão, cada pessoa pode escrever como quiser. Outra desvantagem é a imprecisão, ou seja, a descrição pode não ficar clara e pode-se tirar várias interpretações diferentes de um mesmo algoritmo.

Abaixo temos um exemplo de algoritmo usando a descrição narrativa:

```
1  Início
2          Passo 1: Obter os valores de n1, n2, n3;
3          Passo 2: Somar os valores do passo 1;
4          Passo 3: Dividir o resultado obtido no Passo 2 por 3;
5          Passo 4: Se o resultado do Passo 3 for maior ou igual a 6 então escreva
6          "Você foi aprovado", senão, escreva "Infelizmente você ficou de exame"
7           e vá para o fim do programa
8  Fim
```

Listagem 1. Algoritmo "Calculando a média" em descrição narrativa.

Fluxograma

O fluxograma passou a ser usado para eliminar ambiguidades dos algoritmos. São símbolos gráficos padronizados, cada um representado por uma forma geométrica que implica em uma ação, instrução ou um comando distinto.

Esta forma é intermediária a descrição narrativa e ao pseudocódigo, pois é mais precisa do que a primeira, porém, não se preocupa com detalhes de implementação do programa, como os tipos das variáveis usadas.

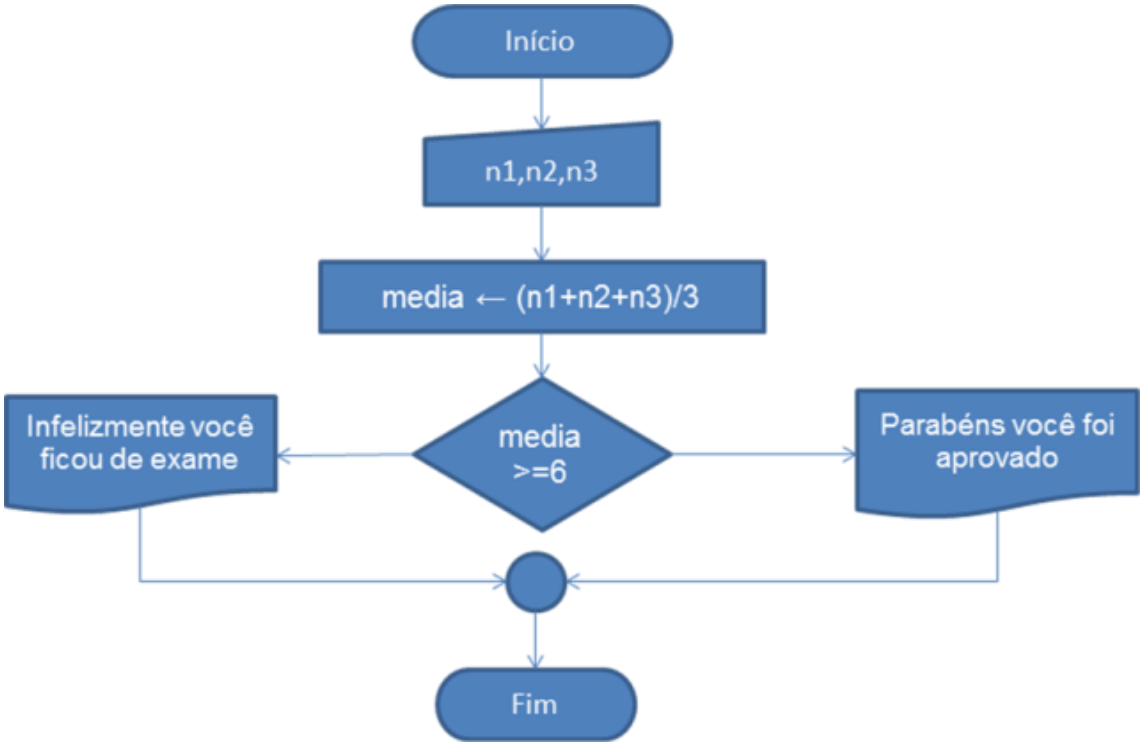


Figura 1. Algorítimo “Calcular Média” em representação de fluxograma.

Linguagem Algoritma (Pseudocódigo ou Portugol)

Essa forma de representação surgiu para tentar suprir as deficiências das outras representações. Consiste na definição de uma pseudolinguagem de programação, cujos comandos são em português, mas que já lembram um pouco a estrutura de uma linguagem de programação estruturada, ou seja, a pseudolinguagem se assemelha muito ao modo como os programas são escritos. Isso vai permitir que os algoritmos sejam traduzidos, quase que diretamente, para uma linguagem de programação.

```
1    algoritmo “CalcularMedia”
2    var
3        n1, n2, n3, media :real;
4
5    inicio
6        leia(n1,n2,n3);
7            media← (n1+n2+n3)/3;
8
9        se  media>=6 entao
10               escreva(“Parabéns você foi aprovado”);
11       senão
12                               escreva(“Infelizmente você ficou de exame”);
13       Fimse
14   fimalgoritmo
```

Listagem 2. Algoritmo “Calcular Média” em linguagem algoritma.

Algoritmos de Ordenação

Algoritmo de ordenação, em ciência da computação, é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem. Em outras palavras efetua sua ordenação completa ou parcial. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista.

Para este artigo foram escolhidos alguns algoritmos de ordenação para serem estudados que são: Bubble Sort, Selection Sort, Quick Sort e Insertion Sort.

Bubble Sort

Bubble sort é o algoritmo mais simples, mas o menos eficientes. Neste algoritmo cada elemento da posição i será comparado com o elemento da posição i + 1, ou seja, um elemento da posição 2 será comparado com o elemento da posição 3. Caso o elemento da posição 2 for maior que o da posição 3, eles trocam de lugar e assim sucessivamente. Por causa dessa forma de execução, o vetor terá que ser percorrido quantas vezes que for necessária, tornando o algoritmo ineficiente para listas muito grandes.

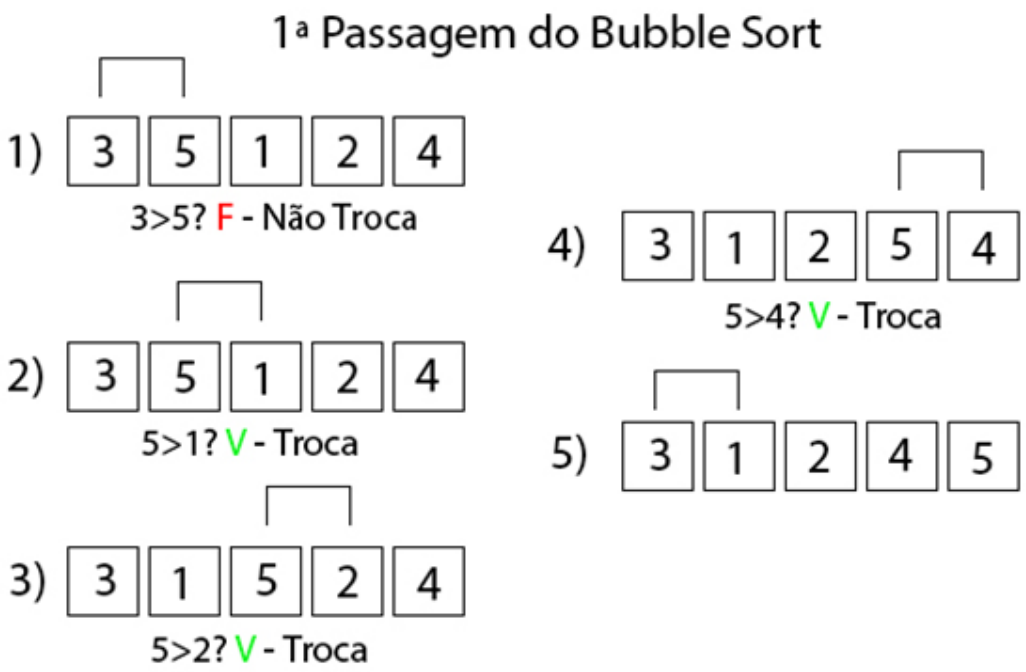


Figura 2. Esquema de funcionamento do Buble Sort.

- É verificado se o 3 é maior que 5, por essa condição ser falsa, não há troca.
- É verificado se o 5 é maior que 1, por essa condição ser verdadeira, há uma troca.
- É verificado se o 5 é maior que 2, por essa condição ser verdadeira, há uma troca.
- É verificado se o 5 é maior que 4, por essa condição ser verdadeira, há uma troca.
- O método retorna ao início do vetor realizando os mesmos processos de comparações, isso é feito até que o vetor esteja ordenado.

Insertion sort

O Insertion sort é um algoritmo simples e eficiente quando aplicado em pequenas listas. Neste algoritmo a lista é percorrida da esquerda para a direita, à medida que avança vai deixando os elementos mais à esquerda ordenados.

O algoritmo funciona da mesma forma que as pessoas usam para ordenar cartas em um jogo de baralho como o pôquer.

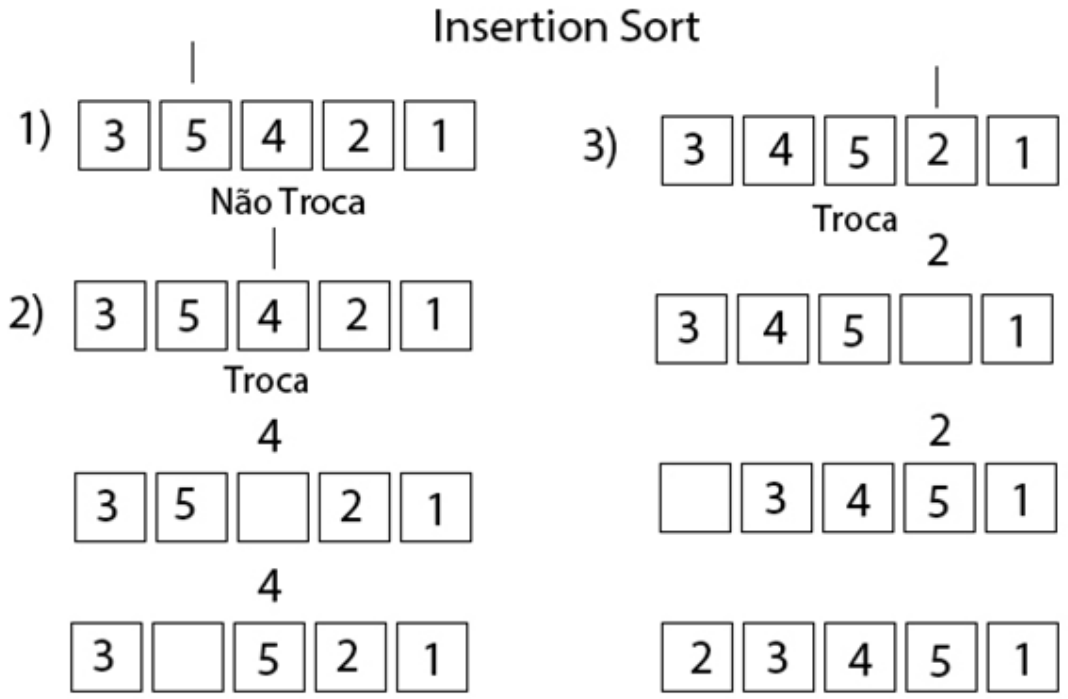


Figura 4. Esquema de funcionamento do Insertion Sort.

- Neste passo é verificado se o 5 é menor que o 3, como essa condição é falsa, então não há troca.
- É verificado se o quatro é menor que o 5 e o 3, ele só é menor que o 5, então os dois trocam de posição.
- É verificado se o 2 é menor que o 5, 4 e o 3, como ele é menor que 3, então o 5 passa a ocupar a posição do 2, o 4 ocupa a posição do 5 e o 3 ocupa a posição do 4, assim a posição do 3 fica vazia e o 2 passa para essa posição.

O mesmo processo de comparação acontece com o número 1, após esse processo o vetor fica ordenado.

Tabela hast com encadeamento, busca linear e busca binária

Introdução

Algoritmos de busca são fundamentais na ciência da computação, sendo amplamente utilizados para localizar informações em estruturas de dados. Entre os métodos mais comuns destacam-se a Tabela Hash com Encadeamento, a Busca Linear e a Busca Binária. Cada um possui características específicas que influenciam sua eficiência e aplicabilidade.

Tabela Hash com Encadeamento

A Tabela Hash é uma estrutura de dados que associa chaves a valores por meio de uma função hash, que calcula o índice de armazenamento no array. Quando duas chaves diferentes resultam no mesmo índice (colisão), o encadeamento é uma técnica eficaz para resolver esse problema.

No encadeamento, cada posição da tabela (bucket) contém uma lista ligada que armazena todos os elementos que compartilham o mesmo índice. Assim, mesmo com colisões, é possível armazenar múltiplos elementos na mesma posição.

Exemplo de Implementação

```
class HashTable:
    def __init__(self, size):
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return hash(key) % len(self.table)

    def insert(self, key, value):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])
```

```
    def search(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None
```

Vantagens e Desvantagens

- Vantagens:
- Inserção, exclusão e busca eficientes na média.
- Boa distribuição de dados com uma função hash adequada.

- Desvantagens:
- Desempenho dependente da qualidade da função hash.
- Possibilidade de listas longas em casos de muitas colisões.

Busca Linear

A Busca Linear, também conhecida como busca sequencial, percorre cada elemento da estrutura de dados até encontrar o valor desejado ou até o final da estrutura.

Exemplo de Implementação

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1
```

Complexidade

- Melhor caso: O(1), quando o elemento está na primeira posição.
- Pior caso: O(n), quando o elemento está na última posição ou não está presente.

Aplicações

Adequada para estruturas de dados pequenas ou não ordenadas, onde a sobrecarga de ordenação não compensa os benefícios de algoritmos mais complexos.

Busca Binária

A Busca Binária é um algoritmo eficiente para encontrar um elemento em uma estrutura de dados ordenada. Ela divide repetidamente o intervalo de busca pela metade até localizar o elemento ou determinar sua ausência.

Exemplo de Implementação

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Complexidade

- Melhor caso: O(1), quando o elemento está no meio.
- Pior caso: O(log n), devido à divisão contínua do intervalo de busca.

Aplicações

Ideal para grandes volumes de dados ordenados, como em bancos de dados e sistemas de arquivos, onde a eficiência é crucial.

Referências

https://linguagemc.com.br/struct-em-c/

https://blog.betrybe.com/linguagem-de-programacao/o-que-e-array/.

https://programae.org.br/cursoprogramacao/glossario/o-que-e-ponteiro-em-programacao/

https://www.geeksforgeeks.org/introduction-to-queue-data-structure-and-algorithm-tutorials/

https://www.geeksforgeeks.org/advantages-and-disadvantages-of-linked-list/

https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-stack/

https://igormcoelho.github.io/curso-estruturas-de-dados-i/slides/3-pilhas/3-pilhas.html