C Manual

Amanda Araujo

Resumo

Compilado de comandos básicos e estruturas da linguagem de programação C. O guia *4dummies* que eu precisava para relembrar conceitos, sintaxe e implementações em C, mantendo minha sanidade. Devemos sempre ser o nosso melhor colaborador e acho que a futura eu me agradeceria. É um processo em andamento e contínuo*.

Conteúdo

Básico da linguagem de programação C

*Códigos disponíveis no GitHub nos repositórios: lab-icc-I e estrutura-dados-I.

Parte I

Básico da linguagem de programação C

1 Linguagem C

A linguagem de programação C foi criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs. Características gerais de C:

- Compilada (*vs. interpretada*);
- Estruturada;
- Imperativa;
- Procedural;
- Nível baixo/médio;
- *Case sensitive*: interpreta como diferentes letras maiúsculas de minúsculas.

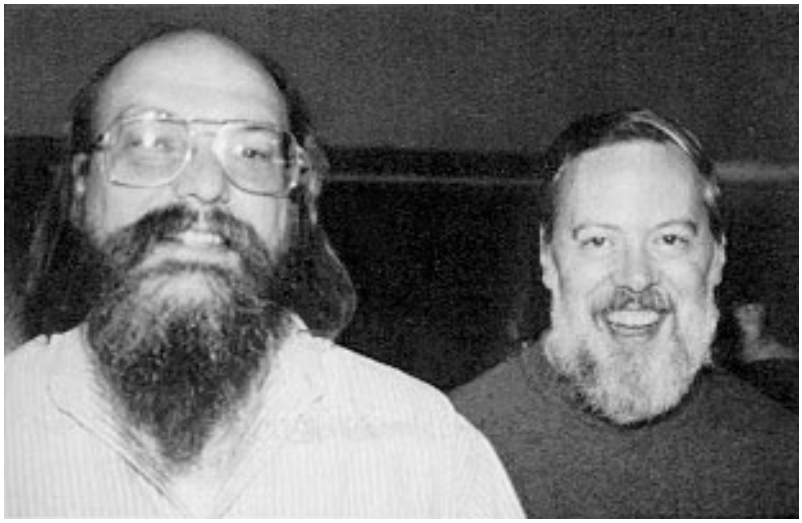


Figura 1: Ken Thompson e Dennis Ritchie (da esquerda para direita), os criadores das linguagens B e C, respectivamente.

Vantagens e desvantagens

A linguagem permite acesso direto à memória e aos recursos do sistema, além de ser capaz de gerar programas extremamente rápidos em tempo de execução.

Uma das principais características da linguagem C é que ela é fortemente tipada, o que significa que o tipo de dado que está sendo manipulado deve ser especificado de forma explícita.

2 Ambiente de trabalho

First things first: *set up* do ambiente de trabalho. Diferentemente de linguagens interpretáveis que rodam diretamente, a linguagem C precisa ser compilada. Ou seja,

mais uma etapa é adicionada, e para rodar códigos em C na sua máquina é preciso baixar e instalar um compilador. O código em si pode ser escrito em qualquer editor de texto, até mesmo no bloco de notas (não recomendável), desde que possua a extensão *.c*. O ideal é usar algum ambiente de desenvolvimento integrado (IDE), como o Visual Studio Code (VS Code), uma aplicação de software que ajuda os programadores a desenvolver código de software de maneira eficiente

Extra: *debugger*. Ao programar coisas mais complicadas em C, especialmente ao mexer com ponteiros e lidar com estrutura de dados, podem surgir erros mais difíceis de serem identificados. Nessas horas, um programa que te ajude a indicar e localizar os erros é essencial, e para isso temos os depuradores (*debugger*).

3 Códigos simples: aspectos da linguagem

Começando pelo clássico *Hello World!*, temos o primeiro exemplo de um código em C:

```c
#include <stdio.h>

/* Primeiro codigo em C */

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Nele é possível notar alguns aspectos gerais da linguagem C, presentes em todos os seus códigos:

- `#include`: todo código em C começa tendo como cabeçalho a importação de bibliotecas que contém funções úteis, por meio do comando *include*. As bibliotecas a serem carregadas devem ser enunciadas dentro de <>. O padrão é a importação da biblioteca *stdio.h*, que contém as funções básicas de C, como *printf()* e *getchar()*.

- Tipo da variável: em C, toda variável e função deve ser declarada explicitamente com seu tipo: *int*, *double*, *char*, etc. Nesse caso, temos o tipo `int`, que indica um inteiro.

- `int main()`: indica ao compilador que o programa tem uma função chamada `main` - Todo programa em C tem uma função `main`. Ela é o local de início (*entry point*) da execução de um programa em C.

- Colchetes { }: indica que tudo dentro deles é parte da função, no caso, da função `main`.

- Ponto e vírgula: toda linha de comando em C deve terminar com uma indicação do seu fim, por meio da adição de ponto e vírgula: `;`.

- `return`: toda função em C necessariamente retorna algo e o comando `return` indica o fim da função. Nesse caso, como a `main` é declarada como uma função `int`, ela vai precisar de um inteiro para retornar quando o programa acabar. Um "0" indica que o programa terminou sem erros; qualquer outro número representa problemas.

- Comentários: parte não executada do código, essencial para anotar ou comentar o que foi feito. Tudo o que se encontra entre `/* */` é um comentário, lembrar de sempre abrir e fechá-lo (barras duplas `//` também é um indicativo de comentário, no entanto, apenas da linha em questão).

4 Variáveis

Em C, uma variável é um nome personalizado definido pelo usuário ou legível pelo usuário atribuído a um local de memória. Todas as variáveis em C são declaradas ou definidas antes de serem utilizadas. Uma *declaração* indica o tipo de uma variável. Se a declaração produz também armazenamento (se inicia), então é uma *definição*.

Declarando e definindo variáveis em C. Sintaxe:

```
type variableName = value;
```

Tipos de dados/variáveis (e sua respectiva máscara de formatação):
- `char` (%c): caracteres (1 byte)
- `int` (%d): números inteiros (4 bytes/2 bytes*)
- `float` (%f): número real de precisão simples (4 bytes)
- `double` (%lf): número real de precisão dupla (8 bytes)
- `void`: valor vazio, usado principalmente ao definir funções (0 bytes)

```
1  // Create variables
2  int myNum = 15;             // Integer (whole number)
3  float myFloatNum = 5.99;    // Floating point number
4  char myLetter = 'D';        // Character
5
6  // Print variables
7  printf("%d\n", myNum);
8  printf("%f\n", myFloatNum);
9  printf("%c\n", myLetter);
```

OBS: Em C não há um tipo de variável especial hexadecimal, no entanto, para *printar* valores no formato hexadecimal é necessário utilizar a máscara %x como um especificador de formato no comando `printf()`. O mesmo vale para a leitura por meio de `scanf()` (%x: para alfabeto em letra minúscula a-f; %X: para alfabeto em letra maiúscula A-F).

*Depende do ambiente de execução do programa; ambiente 16-bit: 2 bytes, ambiente 32-bit: 4 bytes.

OBS2: A linguagem C não tem um tipo de dado específico para armazenar valores booleanos (V/F). Em C, o valor booleano é representado por um valor inteiro: 0 significa falso e qualquer `int` $\neq 0$ significa verdadeiro. Em geral, utiliza-se 1 para representar o valor verdadeiro.

```
1    //To make life easier, C Programmers typically define the terms "
     true" and "false" to have values 1 and 0 respectively.
2
3    //In the old days, this was done using #define:
4    #define true 1
5    #define false 0
6
7    //Today it is better to use const int instead of #define:
8    const int true = 1;
9    const int false = 0;
10
11   typedef int bool;
```

Estendendo um pouco mais... Diferença entre *variáveis* vs. *nomes*

De maneira contrastante, as variáveis em C são fundamentalmente diferentes das variáveis em Python. Python não tem variáveis: *Python has names, not variables.*

Ao declarar uma variável em C (`int x = 2337;`), diferentes passos são executados:

1. Alocação de memória suficiente para o tipo de variável, *p.ex.* um inteiro;
2. Atribuição do valor aquele local de memória;
3. Indicação de que o nome da variável aponta para o valor.

É possível alterar o valor de `x` (variável mutável), sobrescrevendo seu valor anterior no mesmo local de memória, *i.e.* `x` é o local de memória, não somente um nome para ele. Isso pode ser pensado como posse, em certo sentido, `x` possui o local de memória.

Em Python, por outro lado, não há variáveis, apenas nomes. O nome `x` em Python não possui diretamente nenhum endereço de memória - é criado um objeto cujo valor é definido e um nome `x` que lhe é dado -, enquanto a variável `x` em C possui um *slot* estático na memória. Ao se alterar o valor de `x`, o que ocorre em Python é a criação de uma referência a um novo objeto, que não possui o espaço de memória como antes. Além disso, o objeto anterior agora passa a possuir um *ref count* de 0, e será limpo, liberando memória, pelo *garbage collector*. Em resumo, em Python não se atribui valores a variáveis, ao invés disso, nomes são vinculados a referências.

Tudo isso pode parecer um pouco confuso e até mesmo abstrato, no entanto, essas ideias ficam mais claras ao se deparar com conceitos como o de ponteiros, presentes explicitamente em C, mas ausentes em Python. Essas diferenças surgem das propostas de cada linguagem de programação e com qual premissa elas foram criadas. C é tida como linguagem de "baixo nível" - visando tornar manipuláveis aspectos mais ligados ao hardware (memória, por exemplo) e tirar vantagem direta de sua arquitetura, permitindo um controle fino sobre os recursos do sistema -, enquanto Python foi projetada

para ser de “alto nível” - alta abstração, facilitando seu uso por programadores, que passam a estar mais livres para focar na resolução de problemas, ao invés de nos detalhes de implementação de baixo nível.

5 Operadores

Para realizar operações entre as variáveis, existem operadores de diferentes naturezas: aritméticos (1), relacionais (2) e lógicos (3), que representam as operações aritméticas básicas, estabelecem relações/comparações e representam operações baseadas na lógica matemática, respectivamente.

Operação	Operador
Adição	+
Subtração	−
Multiplicação	*
Divisão	/
Resto	%
Incremento	++
Decremento	−−

Tabela 1: Operadores aritméticos

Operação	Operador
Igualdade	==
Diferença	!=
Maior	>
Maior ou igual	>=
Menor	<
Menor ou igual	<=

Tabela 2: Operadores relacionais

Operação	Operador
Negação	!
Conjunção (e)	&&
Disjunção (ou)	||

Tabela 3: Operadores lógicos

- Operador de derreferência (*): usado para acessar o conteúdo de uma posição de memória, cujo endereço está armazenado em um ponteiro.

- Operador de endereço (&): usado para se referir ao endereço da variável valor e não ao conteúdo da mesma ao se atribuir um endereço a um ponteiro.
- Operador ponto (.): acessa um campo (ou membro) de uma *struct*.
- Operador seta (->): acessa um campo (ou membro) de uma `struct` por meio de ponteiros. É uma desreferência do ponteiro contido no membro, indica que deve pegar o valor apontado pelo ponteiro no membro; um exemplo de *syntactic sugar*[*].
- Operador sizeof: indica a quantidade de armazenamento, em *bytes*, necessária para armazenar um determinado tipo de variável `sizeof(tipo)`. É muito usado na alocação dinâmica de memória, permite evitar a especificação tamanhos de dados dependentes do compilador e máquina em uso.
- Operador cast (): operador de conversão explícita de tipos. Sintaxe: `(tipo) expressão`.

Outros tipos de operadores: operadores de manipulação de bits (*bitwise*), operadores de atribuição (= e combinação de = com outros para abreviar expressões), operador série (,), *etc.*

Syntactic sugar

Um açúcar sintático (*syntactic sugar*), termo criado por Peter J. Landin em 1964, é uma sintaxe dentro da linguagem de programação cuja finalidade é tornar suas construções mais fáceis de serem lidas e expressas. Ela faz com que o uso da linguagem se torne "mais doce"para o uso humano, permitindo que suas funcionalidades sejam expressas mais claramente, mais concisamente ou, ainda, como um estilo alternativo preferido por alguns.

---

6 Estruturas de controle

Estruturas de controle são mecanismos que permitem controlar o fluxo de execução de um programa. Elas determinam a ordem em que as instruções são executadas com base em condições ou *loops*. Elas podem ser simples comandos (Figura 2) ou funções (Seção 9).

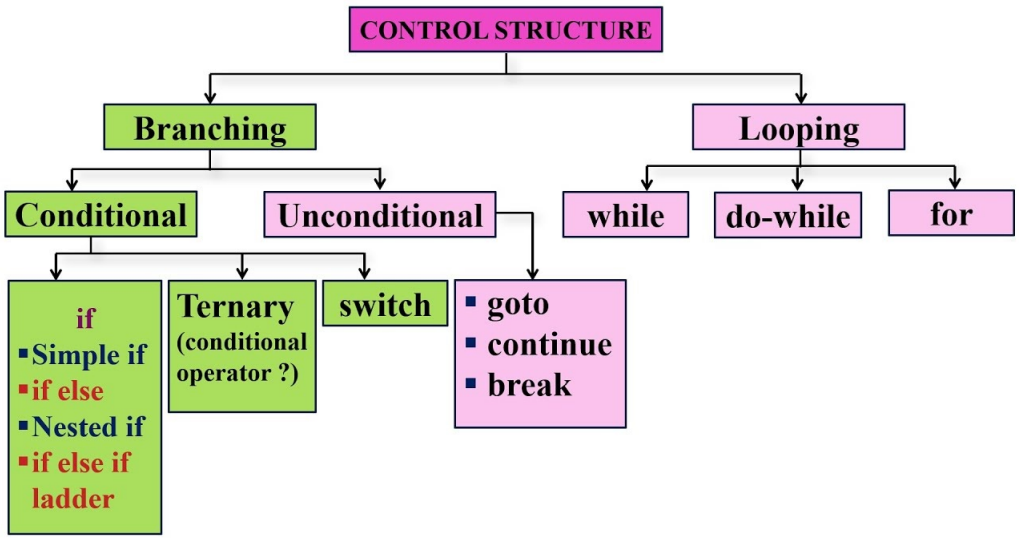


Figura 2: Tipos de estruturas de controle em C.

6.1 Estruturas condicionais

6.1.1 if else

Sintaxe básica:

```
if (condição) {
    // Código a ser executado se a condição for verdadeira;
} else {
    // Código a ser executado se a condição não for verdadeira;
}
```

OBS: Leitura e legibilidade do código. Caso o bloco de comando a ser executado na condição consista em apenas uma única linha, esse pode ser disposto ao lado do condicional, sem a necessidade de chaves, muitas vezes enxugando o código e facilitando sua leitura. Exemplo:

```
1    if (n >= 0){
2        m = m + n;
3    }
4    else{
5        prinf("Valor negativo\n");
6    }
```

```
1    if (n >= 0) m = m + n;
2    else prinf("Valor negativo\n");
```

6.1.2 Switch case

O comando `switch case` é uma forma de reduzir a complexidade de vários `if` & `else` encadeados. É muito utilizado, principalmente para uso em estruturas de menu.

O conteúdo de uma variável é comparado com um valor constante, e caso a comparação seja verdadeira, um determinado comando é executado.

```
switch (expression)
{
case /* constant-expression */:
    /* code */
     break;

    default:
    break;
}
```

Atenção: Não é possível declarar uma variável dentro de um caso do switch.

Um engano muito comum é as pessoas acharem que o `case` é um bloco de comandos e gera um novo escopo. Na verdade, o `case` é apenas um *label*. Então é apenas um nome para um endereço do código usado para provocar um desvio. Na verdade, um `switch` é apenas um `goto` baseado em um valor. Vale para o `switch`, assim como também não se pode declarar uma variável dentro de um `if`.

6.2 Estruturas de repetição: *Loops*

As estruturas de comando de repetição em C (`for`, `while` e `do-while`) são usadas para criar *loops* que executam um bloco de código repetidamente enquanto uma condição específica for verdadeira.

6.2.1 For

Sintaxe básica:

```
for (inicialização; condição; atualização) {
    // Código a ser executado repetidamente;
}
```

Exemplo de código *loop* `for`:

```
1   int n = 10;
2   for (int i = 0; i < n; i++){
3       printf("Contagem: %d", i);
4   }
```

6.2.2 While

Sintaxe geral:

```
while (condição) {
    // Código a ser executado repetidamente;
}
```

7 Ponteiros

Um ponteiro (ou apontador) é uma variável capaz de armazenar um **endereço de memória** ou o endereço de outra variável. Ponteiros podem apontar para qualquer tipo de variável, portanto temos ponteiros para `int`, `float`, `char`, etc.

Utilidade dos ponteiros: úteis quando uma variável precisa ser acessada em diferentes partes do programa. Situações exemplo:

- Alocação dinâmica de memória;
- Manipulação de arrays;
- Para retornar mais de um valor em uma função;
- Referência para listas, pilhas, árvores e grafos.

O uso de ponteiros permite uma maior eficiência no código.
Sintaxe de declaração de um ponteiro:

```
tipo *nome_Ponteiro;
```

Note que: importantíssimo, é o asterisco `*` que indica que se trata de um ponteiro.

Após a criação de um ponteiro, é necessário dizer para quem ele aponta, *i.e.* realizar a atribuição do endereço da variável de interesse, de mesmo tipo, ao ponteiro:

```
nome_Ponteiro = &valor;
```

Note que: importantíssimo, é o operador de endereço `&` que indica que se trata do endereço da variável, e não seu valor.

```c
#include <stdio.h>
void main()
{
  int a;
  int b;
  int c;
  int *ptr;  // declara um ponteiro para um inteiro
             // um ponteiro para uma variavel do tipo inteiro
  a = 90;
  b = 2;
  c = 3;
  ptr = &a;
  printf("Valor de ptr: %p, Conteudo de ptr: %d\n", ptr, *ptr);
  printf("B: %d, C: %d", b, c);
}
```

Ponteiros permitem uma manipulação a um nível mais profundo, lidando diretamente com a memória, o que torna a linguagem C altamente flexível e poderosa, mas também pode abrir a porta para erros mais críticos. Embora os ponteiros sejam poderosos e versáteis, eles também podem ser fontes de erros difíceis de depurar, como vazamentos de memória e falhas de segmentação, os infames e rechaçáveis "*memory leak*" & "*segmentation fault*".

8 Vetores e matrizes

Vetores e matrizes em C são coleções de **variáveis de um mesmo tipo**, acessíveis com um único nome e armazenados contiguamente[†] na memória. Um vetor é uma estrutura de dados indexada, *i.e.* as variáveis armazenadas (itens) são acessadas por meio de um índice próprio. Vetores são matrizes de uma dimensão.

Atenção: Em C o index começa no 0.
OBS: O último elemento de um vetor pode sempre ser acessado pelo índice -1.

8.1 Implementação estática

Sintaxe da declaração de um vetor/matriz:

```
tipo nomeVetor[TAM];
tipo nomeMatriz[NLIN][NCOL];
```

Ao se declarar um vetor/matriz em C, é preciso explicitar, além do tipo de variável a ser armazenada (`int`, `float`, etc.), a quantidade de memória a ser reservada para tal, indicada entre colchetes [ ].

Preenchimento e acesso aos elementos: realizado com o uso de índices, indicado entre colchetes [ ] após seu nome:

```
1   int Vetor[5];   // declara um vetor de 5 posicoes
2   int Matriz[5][3]; // declara uma matriz de 5 linhas e 3 colunas
3
4   Vetor[0] = 9; // coloca 9 na primeira posicao do vetor
5   Vetor[-1] = 30 // coloca 30 na ultima posicao do vetor
6   Matriz[0][1] = 15; // coloca 15 na 1a linha e 2a coluna da matriz
7
8   //Colocar os numeros de 1 a 5 em Vetor
9    for(int i = 0; i < 5; i++){
10     Vetor[i] = i + 1;
11  }
```

Uso de constantes para definir o tamanho de um vetor ou matriz (`#define`):

```
1   #define NLIN 10
2   #define NCOL 5
3   int Matriz[NLIN][NCOL];
4
5   //Preencher uma matriz com um dado
6  for(int i = 0; i < NLIN; i++) {
7      for(int j = 0; j < NCOL; j++) {
8           Matriz[i][j] = 30;
9       }
10  }
```

[†]Se aplica no caso da implementação estática: tamanho conhecido *a priori*.

É possível já no momento de criação do vetor/matriz passar sua lista de valores, entre colchetes { } (declaração e definição em conjunto). Nesse caso, os colchetes retos [ ] aparecem não preenchidos:

```
int myNumbers[] = {25, 50, 75, 100};
```

Atenção: Cuidado com os limites de vetores e matrizes! A linguagem C não faz nenhum teste de verificação dos índices usados para acessar os elementos de um vetor. Isto significa que, se estes limites não forem respeitados, resultados indesejados serão obtidos, serão cuspidos valores espúrios sem aviso de erro. Exemplo (vetor de tamanho 5 e *print* de 10 elementos):

```c
#include <stdio.h>

int main(){

    int Vetor[5];

    for(int i = 0; i < 5; i++) Vetor[i] = i + 1;
    for(int i = 0; i < 10; i++) printf("%d ", Vetor[i]);

    return 0;
}
```

```
1 2 3 4 5 32766 50827776 1247028395 0 0
```

8.2 Implementação dinâmica

Caso não se tenha as dimensões do vetor/matriz em tempo de compilação, *i.e.* não se saiba o tamanho a ser utilizado de antemão, por exemplo, quando o tamanho será fornecido como entrada `scanf`, pode-se empregar a **alocação dinâmica** (uso de ponteiros, Seção 7) de vetores e matrizes.

Alocação dinâmica de memória

Os recursos para alocação dinâmica em C são incluídos através do arquivo-cabeçalho `<malloc.h>`. A função básica para alocar memória é `malloc` (Fig. 3):

- *Parâmetro*: recebe como parâmetro o número de *bytes* que se deseja alocar;
- *Returno*: endereço inicial da área de memória alocada.

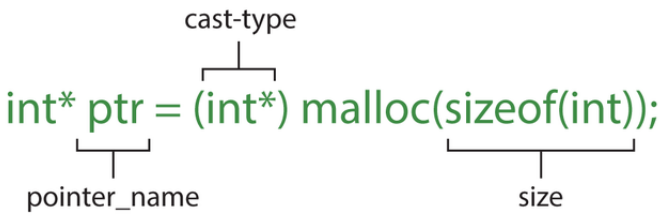


Figura 3: Sintaxe e elementos da função `malloc`

Em sua utilização mais simples, a função `malloc` exige apenas a especificação da área de memória requisitada a ser reservada pelo sistema, ou seja, a quantidade de bytes N que se deseja alocar:

```
malloc(N)
```

Note que dizer explicitamente o número de *bytes* correspondente ao espaço de memória desejado não é muito prático. De modo geral, desejamos lidar com um tipo específico de variável, como no caso dos vetores e matrizes, cada uma com seu tamanho correspondente (Seção 4). É preferível, ao invés de passar o número explícito de *bytes*, informar a quantidade de elementos de um certo tipo a ser guardado n, de maneira independente de compiladores e máquinas, por meio do operador `sizeof( )`:

```
malloc(n * sizeof(tipo))
```

Mais um ponto importante, cabe considerar que a função `malloc` lida com diversos tipos de variáveis. Para gerir essa diversidade, ela retorna um ponteiro genérico do tipo `void*`, que na sequência pode ser convertido automaticamente pela linguagem para o tipo dado na atribuição da variável para qual se aloca a memória. Para evitar erros na conversão automática de um tipo de ponteiro para o outro, é comum utilizar o operador de conversão explicita de tipos, operador cast:

```
(tipo *) malloc(n * sizeof(tipo))
```

8.2.1 Alocação dinâmica de vetores

Exemplo 1: alocação dinâmica de um vetor de 10 inteiros

```
// Declaracao ponteiro
int *v; // recebe endereco inicial do espaco alocado

// Alocacao area contigua de memoria
v = malloc(10 * 4);                     // numero de bytes
v = malloc(10 * sizeof(int));           // tamanho tipo da variavel
v  = (int *) malloc(10 * sizeof(int)); // cast explicito
```

Exemplo 2: alocação dinâmica de um vetor de tamanho dado como *input*

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

int main(){

    int n;           // tamanho vetor
    scanf("%d", &n); // leitura input

    float *v         // declaracao ponteiro
    v = (float *) malloc(n * sizeof(float)); // alocacao memoria

    free(v); // liberacao memoria alocada

    return 0;
}
```

8.2.2 Alocação dinâmica de matrizes: alocação única

Uma possibilidade para alocar dinamicamente uma matriz é através de um único vetor, onde os elementos da matriz são armazenados linearmente. Os elementos da matriz são acessados explicitamente via aritmética de ponteiros.

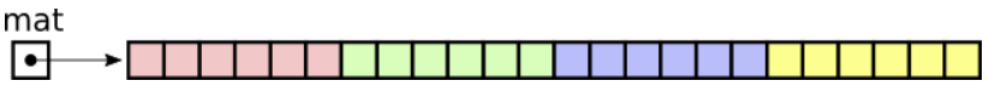


Figura 4: Alocação dinâmica de matrizes via vetor único

Vantagem: alocação da matriz igual à alocação dinâmica de um vetor, simples;
Desvantagem: acesso não intuitivo dos elementos.

Exemplo de implementação alocação dinâmica única de uma matriz:

```
1   //#define LIN 4
2   //#define COL 6
3
4   // Leitura dimensoes da matriz
5   int LIN, COL;
6   scanf("%d %d", &LIN, &COL);
7
8   int *mat;
9   int i, j;
10
11  // Aloca um vetor com todos os elementos da matriz
12  mat = malloc (LIN * COL * sizeof (int));
13
14  // Percorre a matriz
15  for (i=0; i < LIN; i++)
16     for (j=0; j < COL; j++)
17        mat[(i*COL) + j] = 0; // calcula a posicao de cada elemento
18
19  // Libera a memoria alocada para a matriz
20  free (mat);
21
```

8.2.3 Alocação dinâmica de matrizes: vetor de vetores

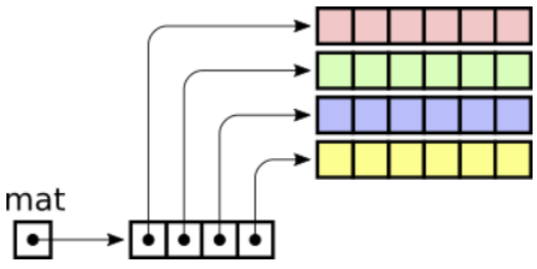


Figura 5: Alocação dinâmica de matrizes via vetor de vetores

Grande vantagem: acesso direto, fácil e intuitivo dos elementos da matriz, assim como é feito para matrizes estáticas.

Exemplo de implementação alocação dinâmica única de uma matriz:

```
#define LIN 4
#define COL 6

int **mat;
int i, j;

// Aloca um vetor de LIN ponteiros para linhas
mat = (int**) malloc (LIN * sizeof (int*));

// Aloca cada uma das linhas (vetores de COL inteiros)
for (i=0; i < LIN; i++){
   mat[i] = (int*) malloc (COL * sizeof (int));
}

// Percorre a matriz
for (i=0; i < LIN; i++){
   for (j=0; j < COL; j++){
      mat[i][j] = 0;        // acesso com sintaxe mais simples
    }
}

// Libera a memoria da matriz
for (i=0; i < LIN; i++) free (mat[i]);
free (mat);
```

Note que: dessa forma, uma matriz é declarada como um ponteiro de ponteiros (**).

9 Funções

Uma função em um código de programação é uma sub-rotina de um programa. Funções são blocos de código que realizam tarefas específicas e podem ser definidas e chamadas pelo programa principal (`main()`) ou por outras funções. O seu uso permite a separação do programa em partes (blocos) que possam ser logicamente compreendidos de forma isolada, facilitando sua leitura. Além disso, funções permitem o reaproveitamento de códigos e contribuem para evitar repetições de um mesmo trecho de código repetidamente para execução de uma mesma tarefa.

Sintaxe básica:

```
tipo nome_função(parâmetros) {
    //Corpo da função;
    return valorRetorno;
}
```

Exemplo de uma função simples:

```c
int soma(int a, int b) {
    int resultado;
    resultado = a + b;
    return resultado;
}
```

Elementos de uma função:

- Tipo (`tipo`): uma função sempre apresenta um tipo, referente ao tipo esperado do seu valor de retorno (`int`, `float`, `void` *etc.*).
- Nome da função: nomeia a função e indica como ela será chamada para ser executada futuramente.
- Declaração de parâmetros: entre parêntesis, uma listagem explicita de todos os parâmetros necessários para a execução da função, com seu respectivo tipo. É opcional.
- Colchetes { }: delimitam a função, indica que tudo dentro deles é parte da função.
- Corpo da função: contém as instruções e comandos a serem realizados como tarefa específica da função.
- `return`: toda função em C necessariamente retorna algo (a menos que se trate de uma função do tipo `void`) e o comando `return` indica o fim da função, juntamente com o valor a ser retornado. Uma função em C retorna apenas **um único valor**, porém isso pode ser contornado de maneiras indiretas (Seção 9.4).

Importante: escopo de variáveis, *i.e.* bloco de código onde as variáveis são válidas. Note que é possível declarar variáveis dentro das funções. Nesse caso, elas recebem o nome de **variáveis locais** (em contraposição às variáveis globais). Variáveis locais definidas dentro de uma função *não* são acessíveis em outras funções, mesmo que as variáveis apresentem nomes idênticos.

9.1 Localização das funções no código-fonte

Localização das funções no código-fonte

Via de regra geral:

Toda função deve ser *declarada* antes de ser usada.

Lembrando, declaração é diferente de definição. A declaração envolve anunciar qual o *tipo* da variável ou função, enquanto na definição ocorre a atribuição de um valor. No caso das funções, no momento em que definimos uma função, está implícita sua declaração (`tipo nome_função()`). Como funções são blocos de código independentes, úteis para encurtar e deixar o código principal simples e enxuto, sua definição pode (e deve) ficar localizada fora da `main()`.

Como o lema diz, a linguagem C exige que toda função seja *declarada* antes do seu

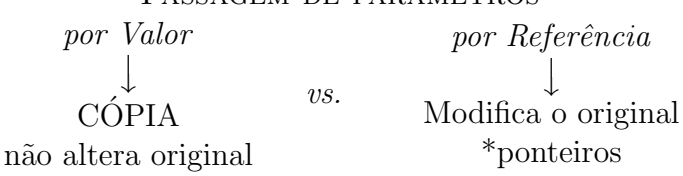
uso, de modo que é possível apresentar um código-fonte claro com o bloco correspondente à função `main()` primeiro, seguida da *definição* de todas as funções auxiliares empregadas. Para tal, basta anunciar acima da `main()` as funções a serem usadas, como um pequeno *spoiler* do que virá. Esta declaração é feita através do **protótipo** da função, que nada mais é do que um trecho de código (uma linha) que especifica o nome e os parâmetros da função.

```
#include <stdio.h>

int SOMA(float a, int b);  // Prototipo da funcao SOMA

int main()
{
    SOMA(16.7,15);  // Chamada da funcao SOMA antes de sua
                    // definicao, mas apos sua prototipacao
    return 0;
}

int  SOMA(float a, int b)  // Definicao da funcao SOMA
{
   float result;     // a declaracao de variaveis igual ao que
                     // se faz na funcao main
   result = a+b;
   printf("Soma de %f com %d: %6.3f\n", a, b, result);
   return 0;
}
```

9.2 Passagem de parâmetros: por valor *vs.* por referência

Passagem de parâmetros para funções

A finalidade dos parâmetros é fazer a comunicação entre as funções e a função principal. Chamamos de passagem de parâmetros a passagem de valores entre as funções. Existem duas formas de passar parâmetros para uma função:



A passagem de parâmetros *por valor* é o que tem sido feito até então nos exemplos, onde os parâmetros declarados são variáveis. Ao serem usados na função, é feita uma *cópia* do valor da variável, que é usada e modificada, enquanto a variável original se mantém inalterada. Já a passagem de parâmetros *por referência* é feita utilizando *ponteiros*, que contém o endereço de memória da variável e permite a alteração do conteúdo da variável original usando-se esta referência. Ou seja, o valor da variável passada como argumento é modificado não somente no bloco da função, mas afetando todo o código.

Passar por referência é recomendado no caso de tipos complexos, como é o caso de

uma `struct`. Embora a passagem da cópia gaste mais memória, esta permite um acesso mais rápido dos valores, atualmente priorizado, dado que normalmente memória não é mais um problema.

Exemplo de passagem de parâmetros por valor e por referência:

```c
#include <stdio.h>

void soma(int a);     // Prototipo Passagem por valor
void somaref(int *a); // Prototipo Passagem por referencia

int main(void)
{
  int numero;
  printf("Digite um numero: ");
  scanf("%d", &numero);

  printf("O numero digitado: %d \n", numero);

  soma(numero);  //chamada da funcao
  printf("Agora o numero vale (por valor): %d \n", numero);

  somaref(&numero);  //chamada da funcao com ponteiro como parametro
  printf("Agora o numero vale (por referencia): %d \n", numero);
  return 0;
}

// Passagem por valor
void soma(int a)
{
    a = a + 10;
    printf("Soma por valor: %d\n", a);
}

// Passagem por referencia
void somaref(int *a)
{
    *a = *a + 10;
    printf("Soma por referencia: %d\n", *a);
}
```

```
Digite um numero: 10
O numero digitado: 10
Soma por valor: 20
Agora o numero vale (por valor): 10
Soma por referencia: 20
Agora o numero vale (por referencia): 20
```

Note que: a função `soma` que recebe o parâmetro por valor realiza sua operação e ao final o valor do número original se mantém o mesmo. Em contraste, a função `somaref` recebe o argumento por referência e ao final sua operação altera o valor da variável original, que é perdido.

Providências para se passar o endereço de uma variável para uma função:

1. Na chamada da função deve-se usar o operador de endereço & antes do nome da variável;
2. No cabeçalho da função, declarar o parâmetro como um ponteiro;
3. Dentro da função, deve-se usar o operador de derreferência * para alterar o conteúdo da variável.

9.3 Passagem de vetor e de matrizes como parâmetro

Para passar uma matriz ou vetor como parâmetro, basta declarar o parâmetro da mesma forma que a matriz/vetor foi declarado.

Por definição da linguagem C, **um vetor é sempre passado por referência**, logo, qualquer alteração em seus elementos, altera a variável usada como parâmetro na chamada da rotina.

9.4 Retorno de múltiplos valores em uma função

A linguagem C permite o retorno direto de apenas **um único valor** para uma dada função. No entanto, há maneiras de se contornar essa limitação, por meio de métodos indiretos:

- Uso de ponteiros;
- Uso de estruturas;
- Uso de arrays.

Retorno de múltiplos valores usando ponteiros

É simples: basta passar o argumento para função *por referência* (endereço de memória) e fazer mudanças utilizando ponteiros (Seção 9.2). Dessa forma, os valores são alterados no argumento original, e são mantidas mesmo fora do ambiente da função.

10 Estruturas (Struct)

Até então, a linguagem de programação C já vem equipada com diversos tipos de dados: `char`, `int`, `float`, *etc.* Contudo, muitas vezes desejamos poder usar construtos que misturem tipos de dados distintos, e que agreguem informações sobre um mesmo ente. Nesses casos, é possível criar um tipo complexo de dado (dado derivado) ou uma (nova) estrutura de dados em C. Uma delas[‡] são as **estruturas**.

As estruturas (ou registros) são coleções de dados, normalmente de tipos diferentes, que atuam como um todo, permitindo o armazenamento de uma só vez dos dados de

[‡]Outras estruturas de dados, além das estruturas, seriam as arrays, uniões, cadeias e campos de bits.

uma mesma entidade. Podem conter tipos de dados simples (caractere, float, array e enumerado) ou compostos (estruturas, array ou uniões). As variáveis internas contidas no registro são denominadas *membros* da `struct`. Sintaxe geral:

```
struct NomeDaEstrutura {
   tipoDeDado membro1;
   tipoDeDado membro2;
   // Pode ter mais membros...
};
```

Um exemplo clássico:

```
struct Aluno {
   char nome[50];
   int idade;
   float nota;
};
```

10.1 Criando structs: typedef

O comando `typedef` permite ao programador definir um novo nome para um determinado tipo. Sua forma geral é:

```
typedef antigo_nome novo_nome;
```

Como exemplo vamos dar o nome de inteiro para o tipo `int`:

```
typedef int inteiro;
```

Agora podemos declarar o tipo inteiro. O comando typedef também pode ser utilizado para dar nome a tipos complexos, como as estruturas. As estruturas criadas poderiam ser definidas como tipos através do comando typedef. O exemplo ficaria:

```
#include <stdio.h>

typedef struct tipo_endereco {
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
    } TEndereco;

typedef struct ficha_pessoal {
    char nome [50];
    long int telefone;
    TEndereco endereco;
    }TFicha;

void main(void){
    TFicha *ex;
    //...
}
```

A sentença `typedef` é utilizada para atribuir um nome novo a um tipo de dado derivado ou básico. `typedef` não define um novo tipo, mas simplesmente um nome novo para um tipo existente:

```
typedef struct {
   float x;
   float y;
} PONTO;
```

A declaração de uma variável de tipo `PONTO`: Ponto de origem = {0.0, 0.0}.

OBS: Os exemplos de `struct` lembram um pouco a funcionalidade dos dicionários em Python... Na linguagem de programação C, não existe uma estrutura de dados nativa chamada "dicionário" como em Python. Em Python, um dicionário é uma coleção associativa que mapeia chaves para valores, permitindo que você armazene e recupere valores com base em suas chaves.

Em C, você não tem um tipo de dado nativo chamado "dicionário" com a mesma facilidade de uso e funcionalidade integrada. No entanto, você pode implementar um dicionário ou uma estrutura de mapeamento semelhante usando outras estruturas de dados disponíveis na linguagem, como *arrays*, *structs* e até mesmo ponteiros para funções. Isso requer mais código e trabalho manual em comparação com a simplicidade de um dicionário em Python.

10.2 Acessando e alterando campos de structs

Os campos (ou membros) de uma `struct` podem ser acessados e alterados por meio do uso do operador ponto (.), caso se trate de uma variável do tipo da `struct` em questão, ou operador seta (->), caso se trate de um ponteiro:

```
typedef struct ficha_pessoal
        {
        char nome [50];
        long int telefone;
        TEndereco endereco;
        }TFicha;

TFicha ficha1;  //Variavel do tipo TFicha
TFicha *ficha2; //Ponteiro do tipo TFicha

ficha1.telefone = 35353035;    //Operador ponto
ficha2 -> telefone = 35353035; //Operador seta (ponteiro)
(*ficha2).telefone = 35353035; //Operador ponto + derreferencia
                               //equivalente ao comando de cima
```

Parte II

Estrutura de Dados I

Programa

Funções Recursivas. Introdução a análise de algoritmos: notação assintótica, análise do pior caso, melhor caso. Tipos abstratos de dados. Listas lineares: sequenciais, simplesmente e duplamente encadeadas, estáticas e dinâmicas. Pilhas, filas, filas de prioridade. Aplicações: matrizes esparsas e grandes números. Listas generalizadas: polinômios. Listas não-lineares: árvores, árvores binárias, operações básicas sobre árvores. Árvores binárias de busca, árvores binárias de busca balanceadas (AVL).

11 Funções recursivas

11.1 Recursão

Recursividade é o ato de chamar a si mesmo; propriedade de função, programa ou afim que se pode invocar a si próprio; do latim *recursio*, -onis, corrida de volta, retorno revolução* (Seção 11.1).

Figura 6: Ilustração de recursividade.

*Dicionário Priberam da Língua Portuguesa

Na lógica matemática e em ciência da computação, uma *definição recursiva* é usada para definir um objeto em termos de si próprio. Em computação, recursão pode ser empregada como um método de resolução de problemas, que consiste em quebrá-lo em casos menores do mesmo problema, ou, em outras palavras, em subproblemas (instâncias) do mesmo tipo do problema original (Fig 7): *dividir para conquistar*. No contexto de funções recursivas e algoritmos, pode-se separar as recursões em duas partes igualmente importantes:

- ***Casos base***: *inputs* para o qual a resposta da função é trivial;
- ***Casos recursivos***: *inputs* para os quais a função chama a si própria.

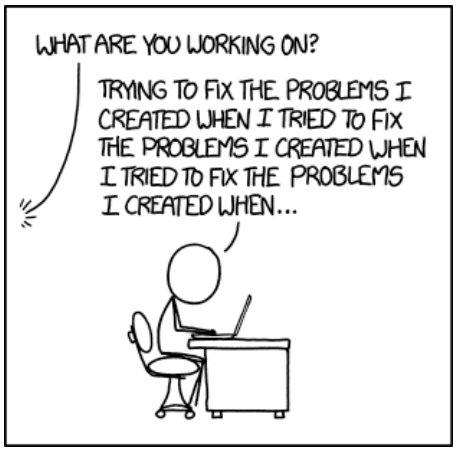


Figura 7: Resolvendo problemas por recursão.

Para garantir que não se caia em uma circularidade perigosa, desprovidos de esperança de sair de um *loop* eterno - que consumiria toda a memória disponível, é fundamental, além da lei da função de recursão, se definir um (ou mais) *casos base*. São eles os responsáveis por garantir que tenhamos uma solução para um problema pequeno o suficiente. Como os casos base quebram a cadeia de recursão, são também designados por "terminating case" (casos de encerremento).

Um dos exemplos matemáticos mais famosos e práticos, envolvem o fatorial de um número, a sequência de Fibonacci e a torre de Hanói.

Alguns fatos importantes para se ter em mente quando se fala de recursão e iteração em programação...

- Todo problema com solução recursiva pode ser resolvido iterativamente, no entanto, nem sempre é o caminho intuitivo dada a natureza do problema;
- Toda função recursiva é passível de ser reformulada como uma função iterativa;
- Há linguagens de programação que não suportam recursão, *i.e.* uma função chamando a ela própria;
- A solução recursiva geralmente é mais cara em termos computacionais de tempo e espaço de memória do que uma solução iterativa;

11.2 Iteração *vs.* Recursão

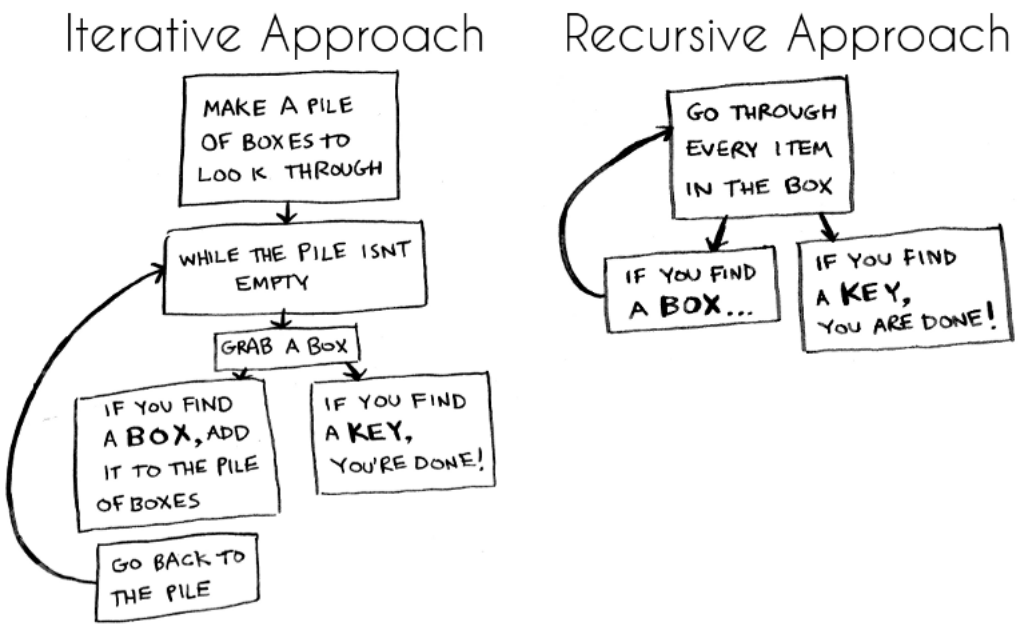


Figura 8: Programa iterativo *vs.* recursivo

Exercício. Determine o que a seguinte função recursiva em C calcula. Escreva uma função iterativa para atingir o mesmo objetivo.

```
1    func (int n)
2    {
3        if (n == 0)
4            return(0);
5        return(n + func(n-1));
6    }
```

Identificação das partes principais da função recursiva dada:

- Caso base: $f(0) = 0$
- Caso recursivo: $f(n) = n + f(n-1); \quad n > 0$

Testando para alguns valores de n:

$f(0) = 0$

$f(1) = 1 + f(0) = 1 + 0 = 1$

$f(2) = 2 + f(1) = 2 + 1 + f(0) = 2 + 1 + 0 = 3$

$f(3) = 3 + f(2) = 3 + 2 + f(1) = 3 + 2 + 1 + f(0) = 3 + 2 + 1 + 0 = 6$

Procurando por um padrão, vemos que a recursão tem por objetivo retornar a soma dos naturais (considerando $0 \in N$) até n: $\sum_{i=0}^{n} i$.

```
1    // funcao iterativa
2    int soma (int n)
3    {
4        return((n + 1)*n/2); // soma de Gauss
5    }
```

12 Análise de algoritmos

Algoritmos são conjuntos de instruções para se realizar certa tarefa e obter uma resposta. Para se solucionar um problema computacional, algoritmos são *implementados* em uma certa linguagem de programação, em um ambiente de trabalho, sujeitos a compiladores e ao processador empregados para tal, *etc.* Invocando Platão, os algoritmos habitam o mundo das ideias, consistem na essência, enquanto os **programas** vivem no mundo sensível, a realidade imperfeita com a qual nos defrontamos no dia-a-dia, que influenciam o desempenho prático do algoritmo. Portanto, deve-se prestar atenção para a utilização dos termos de maneira intercambiável: algoritmo $\neq$ programa. Isso posto, a comunidade de computação começou a pesquisar formas de comparar *algoritmos* de forma independente de hardware, linguagem de programação, habilidade do programador, entre outros fatores externos. Portanto, quer-se comparar algoritmos e não programas.

A **análise de complexidade** de algoritmos é uma etapa fundamental de boas práticas na Ciência da Computação e consiste em estimar uma previsão para o desempenho do código implementado em termos do **número de computações** efetuadas. De modo geral, o melhor algoritmo é aquele capaz de resolver o problema com o menor número de operações sobre a entrada, pois é o mais rápido. É importante ressaltar que esse tipo de análise encontra sentido e utilidade no caso assintótico, para o qual o tamanho da entrada assume valores grandes ($n \to \infty$), que de fato vão exigir poder computacional e tempo para serem resolvidos.

A notação *Big Oh* é, entre outras, a mais usada quando se fala em complexidade e permite comparar a taxa de crescimento das funções correspondentes aos algoritmos, conforme o tamanho do *input* aumenta (Fig 9).

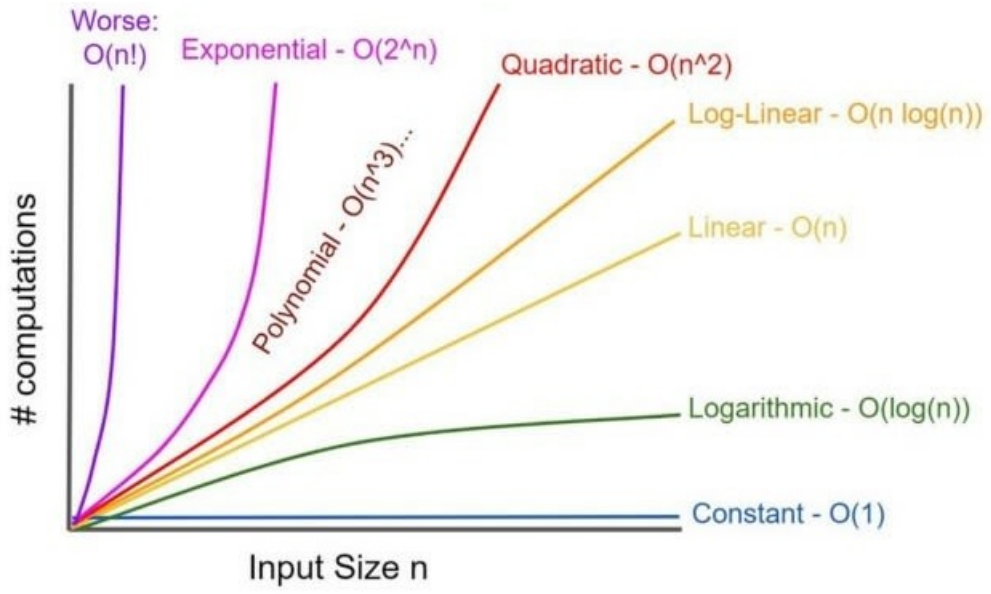


Figura 9: Complexidade de algoritmos - Big Oh

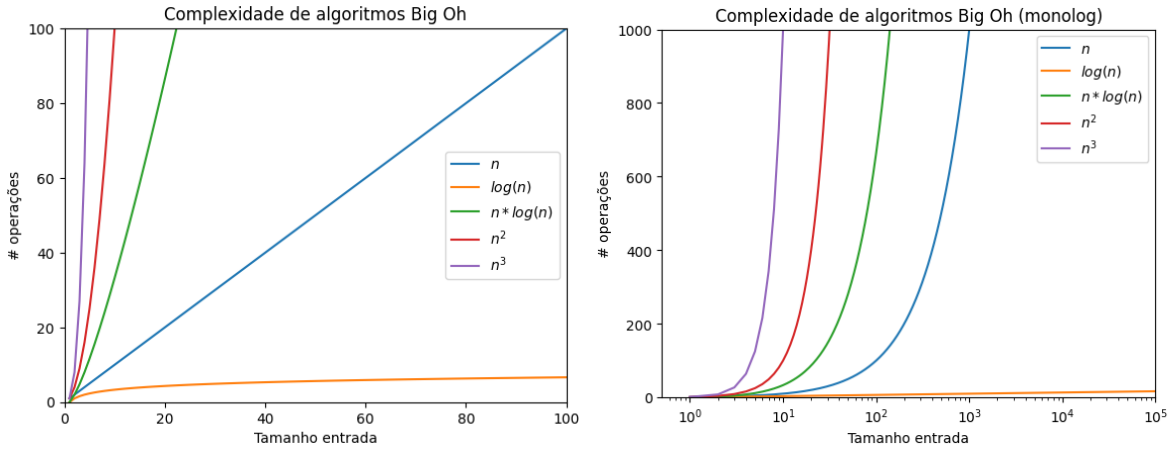


Figura 10: Complexidade de algoritmos - Big Oh, representação do comportamento das curvas em escala linear (*à esquerda*) e mono-log *à direita*.

Para a análise de *Big Oh*, o importante é o **termo de maior ordem** da contagem de operações, de modo que um algoritmo de $3n^2 + n$ operações é da ordem de $\mathcal{O}(n^2)$. Além disso, de modo geral, cada *loop* contribui com um termo n; *for* encadeados m-vezes elevam o número de operações a m-ésima potência de n, devendo ser evitados quando possível, devido ao alto custo computacional.

13 Listas lineares

Lista linear é uma estrutura de dados na qual elementos de um mesmo tipo de dado estão organizados de maneira sequencial. Não necessariamente, estes elementos estão fisicamente em sequência, mas a ideia é que exista uma ordem lógica entre eles. Cada elemento, exceto o primeiro, possui um e apenas um antecessor, e cada elemento, exceto o último, possui um e apenas um sucessor.

Alocação de uma lista na memória

1. Sequencial (ou contígua): ordem lógica = ordem física
2. Encadeada: ordem lógica ≠ ordem física

Operações com listas

- Criação de uma lista
- Remoção de uma lista
- Inserção de um elemento da lista
- Remoção de um elemento da lista
- Acesso de um elemento da lista
- Alteração de um elemento da lista
- Cópia da lista

- Localizar nodo[§] através de info

As operações com a lista são implementadas separadamente, por meio de funções de gerenciamento (modularização do código).

Implementação de listas lineares

A implementação de listas lineares é feita por meio da criação e definição de estruturas: uso de `struct`'s. Existem diversos modelos de listas lineares, cada uma apresentando características próprias que devem ser respeitadas e incluídas ao se definir a estrutura em questão.

13.1 Listas lineares sequenciais

Lista cuja alocação na memória ocorre de maneira contígua, assim como a ordem lógica dos elementos. Ou seja, elementos adjacentes da lista ocupam posições vizinhas na memória física: ordem lógica = ordem física.

Vantagens	*Desvantagens*
✓Acesso direto pelo índice	× Exige grandes movimentações para inclusão e exclusão de elementos
✓Tempo é constante	

A forma mais comum de implementação: uso de um vetor de elementos estático do tipo `REGISTRO` de tamanho `MAX`:

```
1   // Definicao da estrutura: lista linear sequencial
2   #define MAX 50 // Tamanho vetor
3
4   typedef struct{
5       int chave;
6       // outros campos...
7   } REGISTRO;
8
9   typedef struct {
10      REGISTRO A[MAX]; // Vetor de elementos
11      int nElem;       // Numero de posicoes efetivamente ocupadas
12  } LISTA_LINEAR_SEQ;
```

- A estrutura `REGISTRO` é a que define e guarda informação de cada nó. Nesse caso, um nó contém apenas sua `chave`, que é o valor a ser armazenado;
- A lista sequencial é definida de fato em uma outra `struct`: `LISTA_LINEAR_SEQ`. Nessa estrutura, temos no caso duas informações:
    - Um vetor `A[MAX]` do tipo `REGISTRO`, que armazenará os múltiplos nós que comporão a lista;
    - Um inteiro `nElem` para indicar quantos nós a lista tem de fato, *i.e.* estão preenchidos.

[§]Cada elemento da lista é chamado de nó, ou nodo.

13.2 Listas ligadas (ou encadeadas)

Lista linear na qual a ordem lógica dos elementos da lista ("nós") não necessariamente coincide com sua posição física na memória: ordem lógica $\neq$ ordem física.

O diferencial das listas ligadas (ou encadeadas) é que elas surgem para lidar melhor com a tarefa de inserção e exclusão de elementos em listas. Se em uma lista sequencial a adição ou retirada de um elemento em uma posição n implica no reposicionamento de todos os outros elementos seguintes para a casa vizinha, ação que exige muitas movimentações e operações, a lista encadeada contorna o problema com uma simples solução: permite que os dados ocupem qualquer posição em termos do endereço de memória, para tal criando um sistema de gerenciamento dos nós que guarda a informação da sua ordem e quais nós estão livres e ocupados.

Note que esse tipo de estrutura só é possível em linguagens que permitam a manipulação de memória, como C.

13.2.1 Implementação estática

A implementação estática de uma lista ligada (ou encadeada) se baseia no uso de vetores e inteiros. Consiste em uma `struct` para o nó e outra para lista em si. Elementos essenciais:

→ `Lista ligada`:
- Vetor de `REGISTRO`: guarda os nós presentes na lista;
- Campo `inicio`: indica o início da estrutura;
- Campo `dispo`: indica o início das posições livres da lista.

→ `REGISTRO`:
- Campo `chave` (e outros);
- Campo `prox`: indica o sucessor do elemento.

Na prática, `inicio` e `dispo` são as entradas de duas listas que compartilham o mesmo vetor, sendo uma para os elementos efetivos da lista, e a outra para armazenar as posições livres.

```
1   // Definicao da estrutura: lista ligada estatica
2   #define MAX 50
3
4   typedef struct{ // NO
5       int chave;  // Valor do no
6       int prox;   // Sucessor do no
7   } REGISTRO;
8
9   typedef struct{ // LISTA
10      REGISTRO A[MAX]; // Vetor de nos
11      int inicio;      // Indica primeiro nO efetivo (ocupado)
12      int dispo;       // Inicio posicoes livres
13  } LISTA_LIGADA_ESTATICA;
```

Inicialização da lista

Função `void`: tarefa que exige a passagem da lista por referência, alteração da lista original. Para inicializar uma lista, seja ela qual for, é preciso, após declará-la, preencher todos os seus campos. No caso da lista ligada estática, isso implica em associar um valor para os campos `inicio`, `dispo`, e, como a lista será apenas criada (mas não preenchida), encadear os nós, preenchendo o campo `prox` do vetor de `REGISTRO`.

Campo	Valor inicial
`inicio`	-1
`dispo`	0
`A[i].prox`	i + 1
`A[MAX - 1].prox`	-1

Tabela 4: Valores de inicialização dos campos da lista ligada estática

Na criação da lista, não há nós preenchidos, portanto, `inicio` = -1 (índice fora do vetor), indicando que a lista está vazia. Por outro lado, a lista estar vazia implica que todos os nós a partir do primeiro (índice 0) estão disponíveis, logo `dispo` = 0. É preciso encadear os nós, ligá-los entre si, sendo isso feito através da associação do campo `prox` de cada elemento (nó) do vetor de `REGISTRO` ao índice do elemento sucessor. Como estamos implementando com o uso de vetor, `A[i].prox` é simplesmente a posição do índice seguinte do elemento: i + 1, para todos, exceto para o último, que não apresenta sucessor: `A[MAX - 1].prox` = -1 (fim da lista). E *voilà*: *fiat* lista ligada estática!

Exibição da lista

É importantíssimo ter em mente, que o importante é a todo custo manter a *ordem lógica* entre as informações e valores armazenados na lista. Ao se exibir a lista ligada, deseja-se os seus elementos em ordem lógica (não a ordem conforme foram adicionados à memória).

```
1    // * Exibicao da lista
2    void exibirListaLigEstat(LISTA_LIGADA_ESTATICA lista){
3        int i = lista.inicio;
4        while (i > -1){
5            printf("%d ", lista.A[i].chave);
6            i = lista.A[i].prox;
7        }
8        printf("\n");
9    }
```

Função `void` com passagem por valor (cópia): a lista é percorrida partindo do elemento na posição do vetor dada pelo campo `inicio` da lista, e seguindo o índice apontado pelo campo `prox` de cada nó, até que se chegue ao nó do último elemento preenchido da lista, que não apresenta sucessor, portanto `prox` = -1.

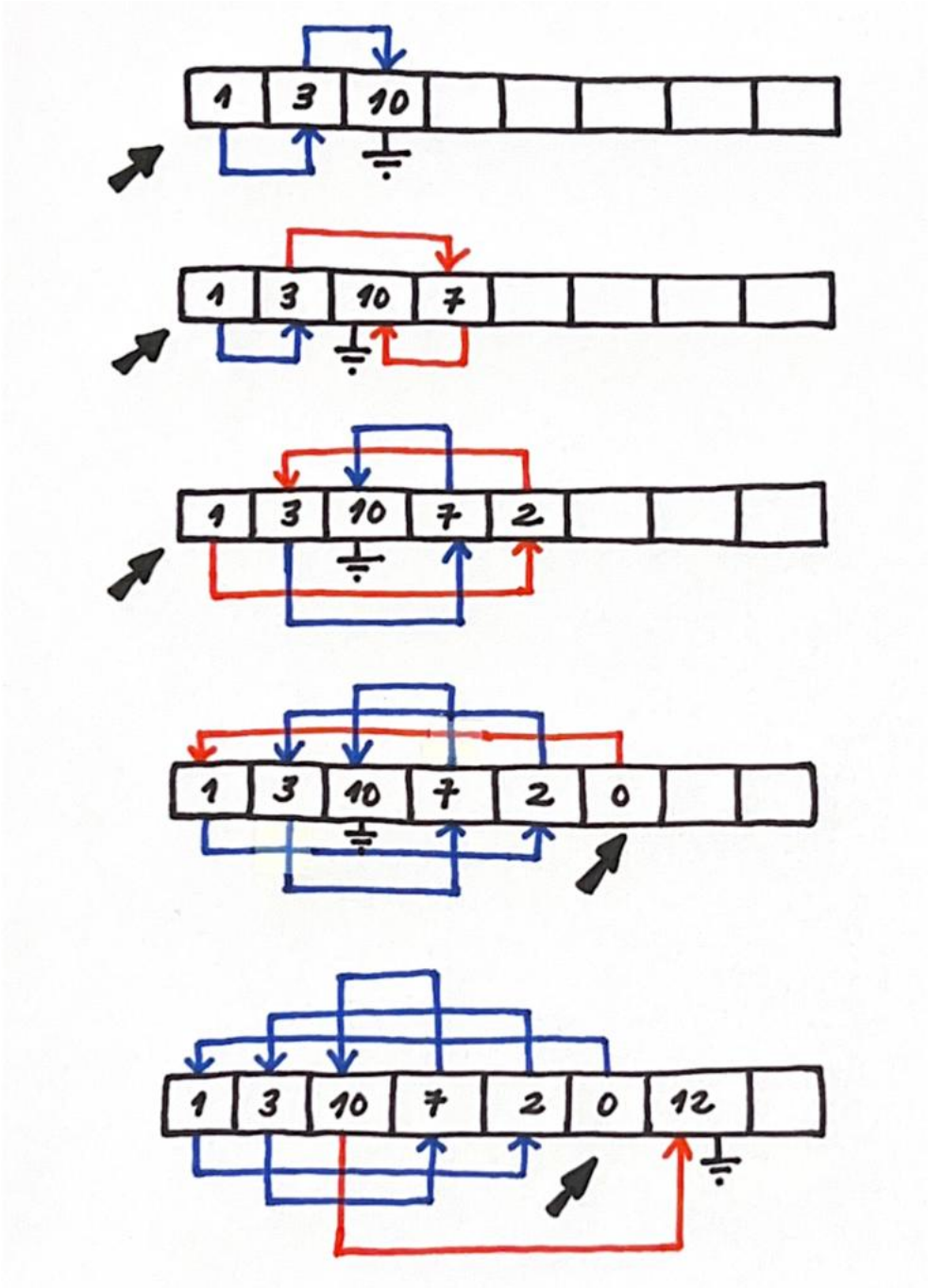


Figura 11: Esquema visual implementação lista ligada estática, múltiplas inserções de elementos. A seta (→) indica o início da lista, enquanto o símbolo de aterramento indica o final dos elementos da lista. Ao se adicionar mais um elemento à lista, uma série de ações deve ser feita: obtenção de um nó livre para a nova chave, atribuição do campo ponteiro `prox` para o novo nó, além da reconexão do campo `prox` do nó antecessor, de modo a manter a ordem lógica da lista encadeada. Note que, ao se inserir um elemento na lista que ocupe a primeira posição na ordem lógica, o campo `inicio` da estrutura da lista deve ser alterado, de modo a indicar a posição do elemento inicial da lista (não necessariamente o índice 0 do vetor).

Inserção ordenada de elementos sem duplicações

Checkouts necessários (`return` false):

✓Se há espaço na lista, *i.e.* `dispo` ! = -1;

✓Busca para checar se o elemento já não está presente.

Inserção de elemento (`return` true):

1 - Associar um nó para o novo valor (pegar um nó na lista dos disponíveis);

2 - Realizar as devidas ligações dos ponteiros `prox` para garantir a ordem lógica.

Exclusão de elemento da lista

Checkout necessário (`return` false):

✓Busca para checar se o elemento está presente.

Exclusão de elemento (`return` true):

1 - Devolver o nó para a lista dos nós disponíveis (alteração `dispo`);

2 - Realizar as devidas ligações dos ponteiros `prox` para garantir a ordem lógica.

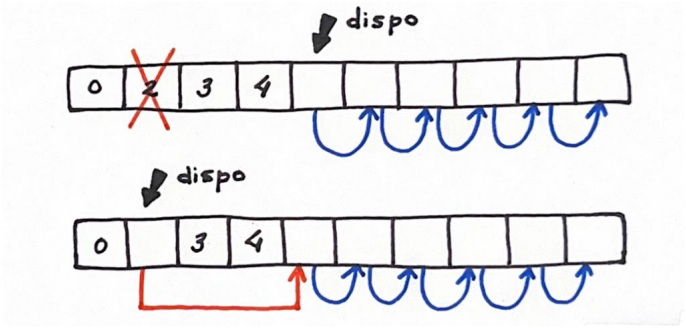


Figura 12: Esquema visual implementação lista ligada estática, exclusão de um elemento. A seta (→) indica o início da lista de *nós disponíveis*: campo `dispo` da estrutura de lista ligada.

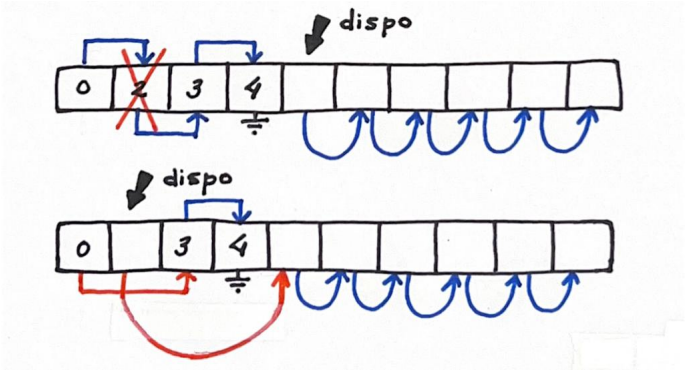


Figura 13: Esquema visual implementação lista ligada estática, exclusão de um elemento, mostrando conexões dos ponteiros `prox` tanto entre nós preenchidos da lista, quanto entre os nós disponíveis. A tarefa de exclusão de um elemento deixa claro como, de fato, há duas listas sendo mantidas em uma: nós ocupados (começo indicado por `inicio`) e nós disponíveis (indicada por `dispo`).

13.2.2 Implementação dinâmica

A implementação dinâmica de uma lista ligada (ou encadeada) se baseia no uso dos recursos de alocação dinâmica (`<malloc.h>`), *i.e.* ponteiros. Evita a necessidade da definição antecipada do tamanho máximo da lista, como acontece na implementação estática, sendo sua grande vantagem o efeito sanfona: usa tanta memória quanto necessário. No entanto, o controle do ponteiro é essencial; a perda do ponteiro implica na perda do acesso à lista.

Consiste em uma `struct` para os nós e outra para lista em si. Elementos essenciais:
→ `NÓ`:
- Campo `chave` (e outros);
- Campo `*prox`: indica o sucessor do elemento, note que temos um ponteiro de fato, autorreferente, pois aponta para outros nós.

→ `Lista ligada dinâmica`:
- Campo `*inicio`: apenas um ponteiro do tipo `NO`, indica o início da estrutura.

Na estrutura da lista ligada dinâmica, não há mais um vetor de nós (como na estática). A lista em si consta apenas do ponteiro para o nó inicial, a partir daí a lista é continuada apenas com nós apontando para nós por meio de autorreferência. Não é necessário gerir lista de nós disponíveis, a memória é alocada para cada elemento, e cada elemento tem o *endereço* de memória do próximo nó (não mais a posição no arranjo).

Sintaxe básica lista ligada dinâmica:

```
typedef struct estrutura {
    TIPOCHAVE chave;
    struct estrutura *prox;
} NO;

typedef struct {
    NO* inicio;
} LISTA;
```

Implementação estrutura de lista ligada dinâmica:

```
1   // Definicao da estrutura
2   typedef struct estrutura{
3       int chave;              // valor do no
4       // int info;            // outros campos
5       struct estrutura *prox; // sucessor do no (autorreferencia)
6   } NO;
7
8   typedef struct{
9       NO *inicio;             // ponteiro para o primeiro no
10  } LISTA_LIGADA_DINAMICA;
```

13.3 Listas dinâmicas com nó cabeça e circularidade

As listas dinâmicas com nó cabeça e circularidade mantém a estrutura de lista ligada dinâmica, com características adicionais:

- Circular: último elemento aponta para o primeiro;
- Nó cabeça: encabeça a lista, que passa a nunca estar vazia. A presença do nó cabeça (não conhecido pelo usuário) implica na existência sempre de um antecessor, facilitando o gerenciamento da lista nas tarefas de inserção e exclusão de elementos.

A definição da estrutura segue o mesmo padrão da lista ligada dinâmica, no entanto, para explicitar o uso e presença do nó cabeça, o campo do ponteiro da `LISTA` (antes intitulado `inicio`) passa a se chamar `cabeca`. A presença do **nó cabeça facilita muito a inserção e exclusão** de elementos, além do que, este pode ser usado como sentinela na tarefa de busca ordenada (importante como função auxiliar para se obter o antecessor do elemento a ser inserido).

Inicialização da lista

Função `void`, passagem da lista por referência. Diferentemente da criação da lista ligada dinâmica simples, cuja inicialização bastava apenas apontar o ponteiro da lista `inicio` para `NULL`, a lista com nó cabeça nunca está vazia, logo esse nó auxiliar fantasma deve ser criado juntamente com o nascimento da lista, alocando-se memória para tal e ligando corretamente os ponteiros. A inicialização da lista consiste em 3 etapas:

1 - Criar nó cabeça (alocar memória);
2 - Apontar campo `cabeca` da `LISTA` para nó cabeça;
3 - Apontar `prox` nó cabeça para ele mesmo (circularidade).

Tamanho da lista

Função `int`, deve percorrer a estrutura. Pequena diferença: apenas os elementos válidos devem ser contabilizados (nó cabeça não conta). O primeiro elemento válido é aquele cujo nó cabeça aponta:

```
NO *end = lista->cabeca->prox;
```

Além disso, diferentemente da implementação básica da lista ligada dinâmica, nunca passaremos por um ponteiro `NULL`, sempre há um endereço válido, devido à circularidade. Isso altera o modo como a lista é percorrida e o critério de parada no *loop*:

```
while (end != lista->cabeca) end = end->prox;
```

(⋆) Essa mesma lógica é usada em todas as tarefas que exigem que a lista seja percorrida: **exibição da lista**, **reinicialização da lista**, **último elemento da lista**, *etc.*

13.4 Listas dinâmicas duplamente encadeadas com nó cabeça e circularidade

Lista ligada dinâmica com as características adicionais de circularidade e nó cabeça e mais uma nova *feature*: encadeamento duplo. Ou seja, cada nó apresenta um campo não só para o seu sucessor (`prox`), como também um campo para o seu antecessor (`ant`), de modo que a lista passa a poder ser percorrida nos dois sentidos:

```
typedef struct estrutura {
    TIPOCHAVE chave;
    struct estrutura *prox;
    struct estrutura *ant;
} NO;

typedef struct {
    NO* cabeca;
} LISTA;
```

Devido à adição de mais um ponteiro, é preciso ter cuidado em dobro ao se ajustar os ponteiros após inserir e excluir elementos. Se antes a lista ligada dinâmica exigia o ajuste de dois ponteiros (`i->prox` e `ant->prox`), com o encadeamento duplo são 4 ponteiros.

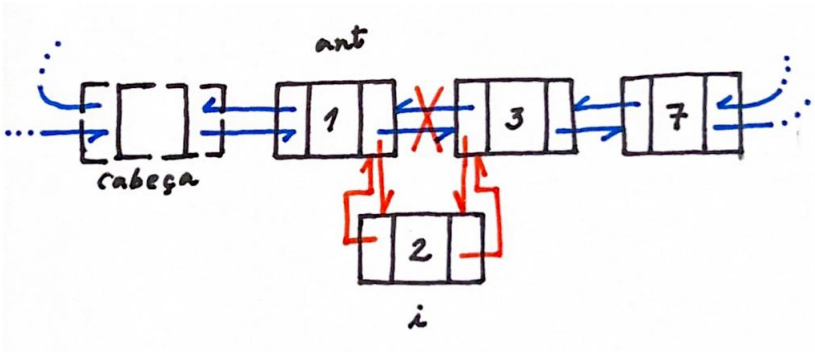


Figura 14: Esquema visual lista dinâmica duplamente ligada com nó cabeça e circularidade. A inserção de um novo elemento exige o ajuste de 4 ponteiros: `prox` e `ant` do novo nó, além dos ponteiros dos vizinhos `prox` do antecessor e `ant` do sucessor.

13.5 Filas

Filas são estruturas de listas lineares que seguem a disciplina de acesso: "first-in, first-out" (FIFO). As operações de inserção e exclusão só ocorrem nas extremidades da estrutura:

- Inserção: final da fila;
- Exclusão: início da fila.

A regra de acesso torna mais fácil lidar com filas do que com uma lista, devido à manipulação mais simples, restrita às extremidades.

Figura 15: Representação de filas estática e dinâmica, respectivamente.

13.5.1 Implementação estática

A implementação estática de uma fila consiste no uso de um vetor circular.

```
#define MAX 50

// Definicao estrutura
// Implementacao estatica - arranjo tam pre-defenido
typedef struct{
    int chave;
} REGISTRO;

typedef struct {
    REGISTRO A[MAX];
    int inicio; // indice do comeco da fila
    int nElem;  // numero de elementos
} FILA_ESTATICA;
```

13.5.2 Implementação dinâmica

A implementação dinâmica de uma fila se baseia no uso de ponteiros.

```
typedef struct estrutura {
    int chave;
    struct estrutura *prox;
} NO;

typedef struct {
    NO *inicio; // util para exclusao
    NO *fim;    // util para insercao
} FILA_DINAMICA;
```

13.5.3 Filas com prioridade

Por ironia do destino, as filas com prioridade não tiveram prioridade alguma dentre as estruturas de dados de listas lineares...

13.6 Deques

Deques são filas de duas pontas: permitem a retirada e inserção de elementos em ambas extremidades: `inicio` e `fim` passam a ser vistos como de mesma natureza, `inicio1` e `inicio2`. É possível implementar de forma estática (uso de vetor circular) ou dinâmica, eis uma implementação da dinâmica:

```
typedef struct estrutura {
    TIPOCHAVE chave;
    struct estrutura *prox;
    struct estrutura *ant;
} NO;

typedef struct {
    NO* inicio1;
    NO* inicio2;
} DEQUE;
```

A estrutura do `NO` é idêntica ao da **lista dinâmica duplamente encadeada** com nó cabeça e circularidade, no entanto, a estrutura da lista em si `LISTA` é igual ao da **fila**, com a pequena alteração do campo ponteiro `fim` para `inicio2`.

13.7 Pilhas

Pilhas são listas lineares com disciplina de acesso "last-in, first-out" (LIFO). As operações de manipulação inserção e exclusão de elementos são trabalhadas apenas no topo da pilha. Ou seja, embora seja, estruturas de dados distintas, que servem a diferentes propósitos, uma pilha poderia ser vista com uma fila ainda mais restrita: operações apenas em uma das extremidades.

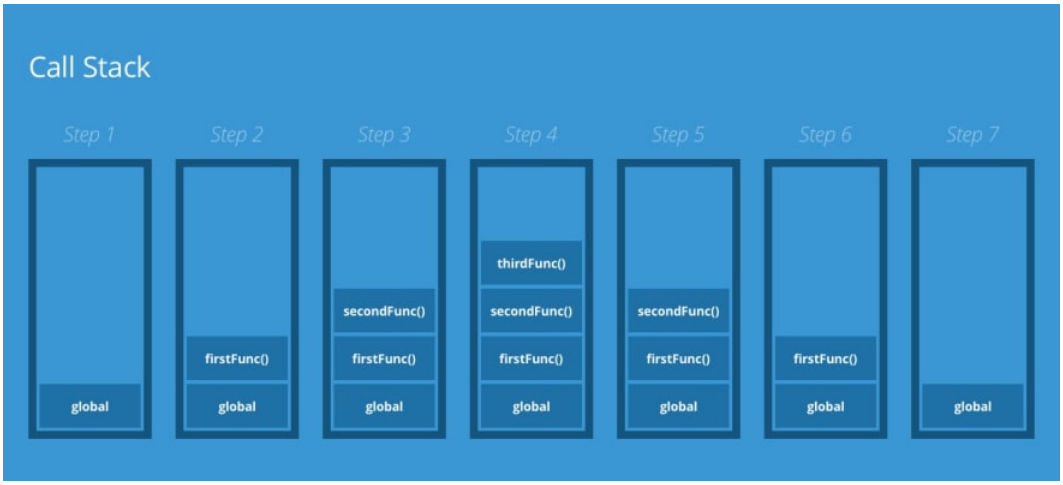


Figura 16: A estrutura de dados pilha (ou *stack* em inglês) é muito utilizada em sistemas operacionais, especialmente no retorno de funções: *main* A → B → C; ordem de `return` C → B → A → *main*.

Operações em pilhas, realizadas no seu **topo**:

- Inserção: *push*;
- Exclusão: *pop*.

13.7.1 Implementação estática

A implementação pode ser estática (usando um vetor simples) ou dinâmica (com ponteiros) sem diferenças significativas em termos de eficiência, uma vez que a estrutura só admite estas operações em seu topo.

13.7.2 Implementação dinâmica

Implementação dinâmica da pilha:

```
1   // Definicao estrutura
2   typedef struct estrutura{
3       int chave;
4       struct estrutura prox*
5   } NO;
6
7   typedef struct {
8       NO *topo;
9   } PILHA_DINAMICA;
```

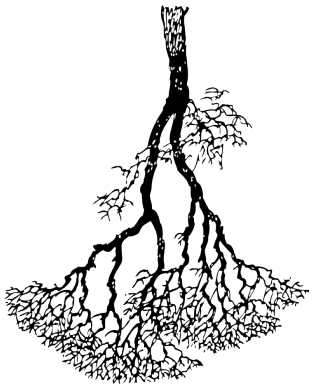
A definição da estrutura de pilha na implementação dinâmica é idêntica a da **lista ligada dinâmica**, com leve alteração no nome do campo do ponteiro da estrutura em si (na lista temos `inicio` marcando o primeiro nó, enquanto na pilha temos `topo`).

13.7.3 Representação de duas pilhas em um único vetor

Trata-se de um caso extra para um pilha estática, onde duas pilhas compartilham o mesmo espaço (pré-definido) do vetor que representa o esqueleto da estrutura da pilha. Duas pilhas de implementação estática que não necessitam de toda sua capacidade simultaneamente podem ser representadas economicamente em um único vetor compartilhado. Vantagens: uma pilha pode crescer mais rápido do que a outra, compartilhando há economia do espaço de memória que precisa ser reservado para a tarefa. Cada uma das pilhas é posicionada em uma das extremidades do vetor, e crescem em direção ao centro, conforme elementos lhes são adicionados.

```
typedef struct {
    int topo1;
    int topo2;
    int A[MAX];
} PILHADUPLA;
```

14 Listas não-lineares: Árvores



Estruturas de dados não-lineares são estruturas nas quais é possível percorrer diferentes caminhos, a exemplo de árvores e grafos. **Árvores** apresentam estrutura hierárquica, nas quais os elementos encontram-se "acima" ou "abaixo" uns dos outros, diferentemente das listas, nas quais os dados estão dispostos em uma estrutura sequencial. É uma estrutura hierárquica cuja representação facilita a organização e busca dos dados, de tratamento computacional eficiente e simples.

Elementos e terminologia de uma árvore

Uma árvore consiste em uma coleção finita de nós conectados por arestas, que apresentam uma relação hierárquica entre os nós.

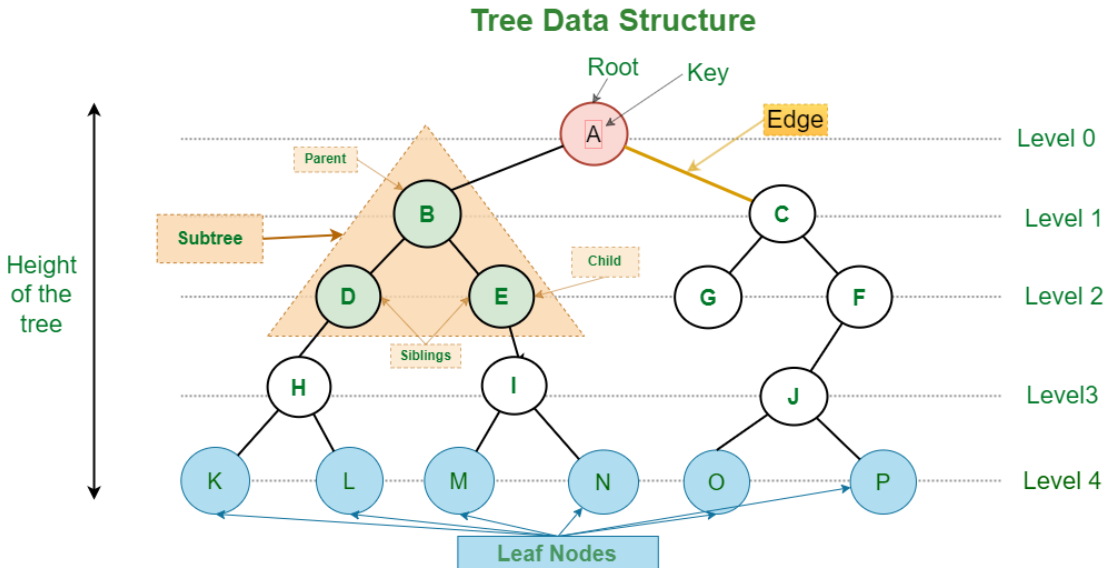


Figura 17: Elementos e terminologia da estrutura de dados árvore.

- Raiz (*root*): a mais alta hierarquia, demarca o começo a árvore, a partir do qual todos os outros nós emanam;
- Sub-árvore (*subtree*): conjuntos disjuntos de nós, subordinados ao nó raiz, que são por si próprios, árvores (sub-árvores);
- Grau de um nó: número de sub-árvores subordinadas a esse nó, ou a quantidade de nós-filhos;
- Grau da árvore: o maior grau apresentado por um nó da árvore;

- Nó-folha (*leaf nodes*): nós de grau zero, nós presentes no último nível da árvore (nós terminais);
- Nó-pai (*parent*): nó antecessor imediato de um nó, nó imediatamente um nível acima;
- Nó-filho (*child*): nó sucessor imediato de um nó, ou "filhos de x": raízes das sub-árvores de x (nó-pai);
- Nó-irmão (*siblings*): nós filhos de um mesmo nó-pai;
- Descendentes: todos os nós abaixo de um determinado nó, qualquer nó sucessor no caminho do nó folha até aquele determinado nó;
- Nível de um nó: quantidade de arestas que o separam do nó-raiz (Fig 17), ou tendo como referencial nó raiz no nível 1, o número de arestas no caminho da raiz até o nó + 1. Se um nó está no nível n, seus nós filhos estão no nível $n + 1$;
- Altura de um nó: altura de um nó-folha é sempre zero, altura de um nó não folha é a altura máxima dentre todas suas sub-árvores + 1;
- Altura da árvore: nível máximo de qualquer nó na árvore, altura de sua raiz.

O grau da árvore, ou o número máximo de nós filhos, pode ser utilizado como nomenclatura para a árvore: uma árvore de grau n é dita árvore n-ária (Fig 18).

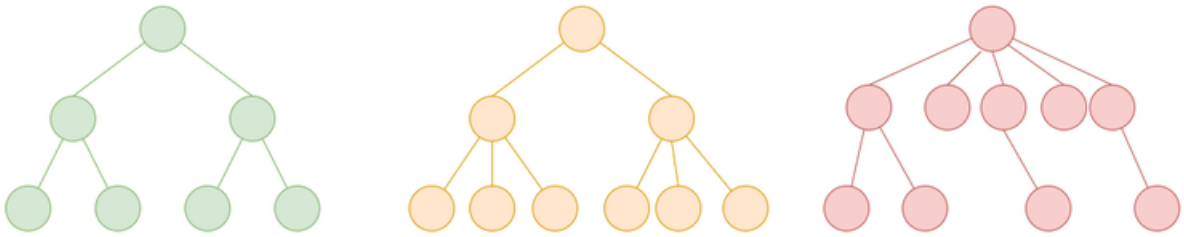


Figura 18: Exemplos de árvores n-árias (*esquerda para direita*): binária (2-ária), ternária (3-ária) e n-ária ($n = 5$), com número máximo de nós filhos 2, 3 e 5, respectivamente.

14.1 Árvores binárias

Árvores binárias são um tipo específico de estrutura de árvore, que se diferencia da árvore comum por apresentar as seguintes propriedades:

- ✓ Grau máximo de cada nó: 2;
- ✓ Ordem (esquerda e direita) nas 2 sub-árvores.

Note que: Não basta uma árvore ser n-ária com $n = 2$, para ser considerada dentro da categoria árvore binária. É introduzido o conceito de **ordem**, que não aparece na

definição de árvore comum.

Propriedades árvore binária

- Número máximo de nós no nível i: 2^{i-1}

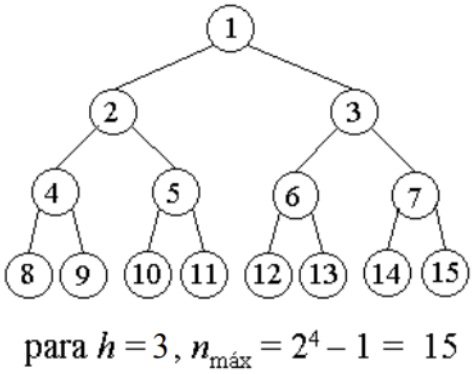
  Nível 1: $i = 1 \rightarrow 2^{1-1} = 2^0 = 1$ nó (raiz)

  Nível 2: $i = 2 \rightarrow 2^{2-1} = 2^1 = 2$ nós

  Nível 3: $i = 3 \rightarrow 2^{3-1} = 2^2 = 4$ nós

  Nível 4: $i = 4 \rightarrow 2^3 = 8$ nós

- Número máximo de nós em uma árvore de altura h: $2^{h+1} - 1$



- Árvore binária é dita *cheia* caso apresente o número máximo de nós para a altura h;
- Árvore binária é dita *completa* caso apresente altura mínima, *i.e.* seus nós não possam ser redistribuídos de modo a formar uma árvore binária de altura menor.

14.1.1 Implementação dinâmica

A árvore binária pode ser representada por meio da união de nós, sendo estes por sua vez representados por uma `struct` com os campos: chave e 2 ponteiros apontando para as sub-árvores esquerda e direita.

```
1    // Definicao da estrutura
2    typedef struct estrutura {
3        int chave;  // chave
4        struct estrutura *esq; // ponteiro filho esq (autorreferencia)
5        struct estrutura *dir; // ponteiro filho dir (autorreferencia)
6    } NO;
7
8    typedef struct {
9        *NO raiz;  // ponteiro para raiz da arvore
10   } ARVORE_BINARIA;
```

14.2 Árvores de busca binária

Árvores de busca binária são estruturas de dados não-lineares altamente otimizadas para tarefa de **busca**:

- ✓ Árvore binária: máximo de 2 filhos (sub-árvores por nó);
- ✓ Ordenação nas chaves: nó à esquerda → menor, nó à direita → maior.

A definição da estrutura segue a mesma `struct` da árvore binária. Seu diferencial, a ordenação dos nós pelo valor da chave, é garantido por meio da implementação da função de gerenciamento de inserção dos nós, sendo adicionados na árvore já obedecendo à ordem.

Diferencial das árvores de busca binária

Grande motivação por de trás da estrutura de árvore binária é a criação de uma estrutura de dados dinâmica capaz de ajudar na tarefa de busca binária (mais eficiente computacionalmente) com alocação de dinâmica de memória. Note que, a busca binária depende de um arranjo ordenado (lista sequencial estática); se tivéssemos uma lista ligada (ordem lógica dos elementos $\neq$ ordem física), não haveria como implementar a busca binária, afinal, como saber quem é o elemento do meio (qual o seu ponteiro)? E o elemento do meio do meio?... Eis que surge a árvore de busca binária.

14.2.1 Funções de gerenciamento: inserção, busca e remoção

Em uma árvore de busca binária, três tarefas são essenciais:

- Inserção de um nó
- Busca por uma chave
- Remoção de um nó

Inserção

```
1   // Funcao de gerenciamento: Insercao arvore de busca binaria
2   NO* inserir2(NO* p, int ch)
3   {
4       if(!p){ //NULL (caso base)
5       p = (NO *) malloc(sizeof(NO));
6       p->esq = NULL;
7       p->dir = NULL;
8       p->chave = ch;
9       }
10      else // recursao
11          if(ch < p->chave)
12              p->esq = inserir2(p->esq, ch);
13          else
14              p->dir = inserir2(p->dir, ch);
15      return (p);
16  }
```

Note que: a função de inserção é uma função *recursiva*. A ordem das chaves é mantida por meio da estrutura condicional `if`, que compara o valor da chave a ser inserida

com o nó atual, direcionando a inserção do novo nó para a respectiva sub-árvore.

OBS: A árvore de busca binária final **depende da ordem de inserção** dos nós. Isso pode levar a árvores de busca binária *degeneradas*, que, na prática, se comportam como listas encadeadas (Fig 19)[¶].

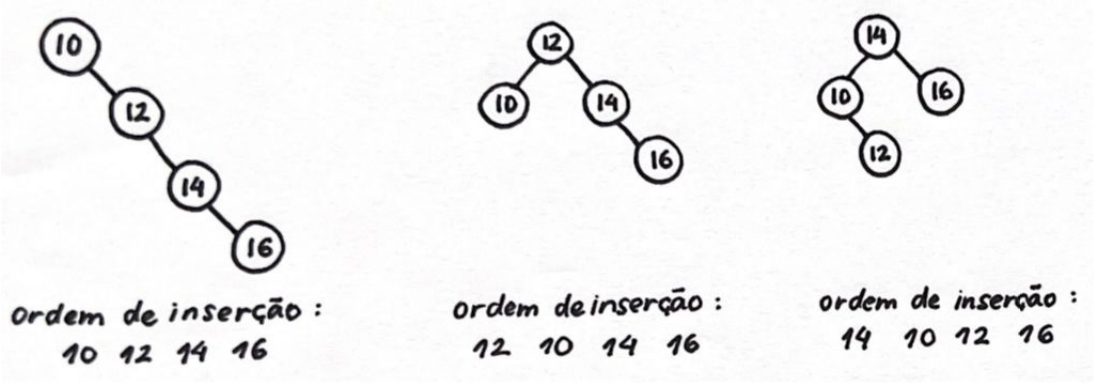


Figura 19: Diferentes ordens de inserção e suas configurações finais. A árvore binária assume diferentes configurações dependendo da ordem de inserção dos nós. As mesmas informações estão armazenadas nas três árvores, porém são acessadas por caminhos distintos, de nível mais ou menos otimizado para a tarefa de busca; no caso da árvore degenerada (*à esquerda*) a busca, em seu pior caso, é da ordem de n.

Remoção

Ao se remover um nó de uma árvore de busca binária, é preciso garantir que a árvore final se mantenha conectada e ordenada. Existem três casos a serem considerados:

1. Nó-folha: remoção direta;
2. Nó com 1 filho: nó-filho assume o lugar do nó-pai;
3. Nó com 2 filhos: * antecessor central & sucessor central

[¶]Esse possível problema foi contornado por meio do uso de árvores AVL.

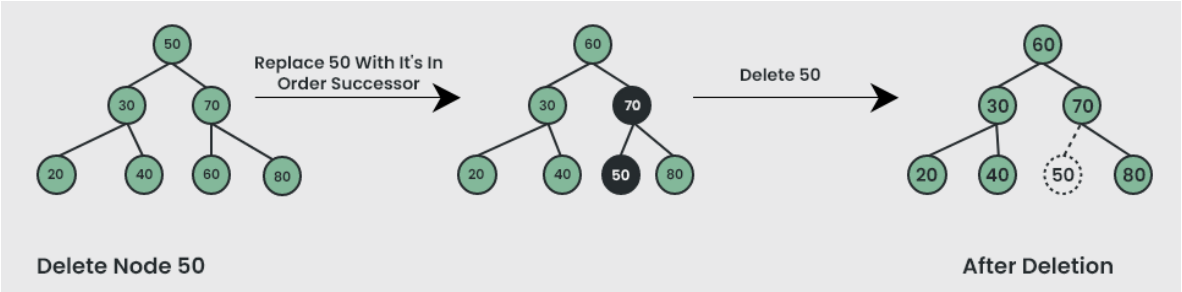


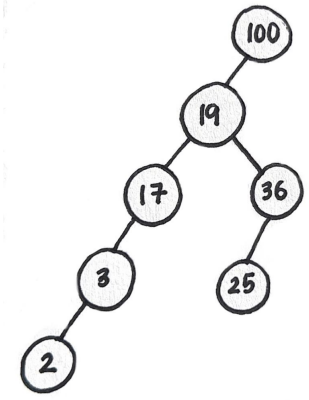
Figura 20: Remoção de um nó: caso nó com 2 filhos. Deleção do nó de chave 50. Nesse caso, optou-se pela substituição com o sucessor central (40 seria o antecessor central). Ao final do processo, o nó desejado foi removido e a árvore mantém-se de busca binária com a manutenção da ordenação.

14.2.2 Percursos em árvores

Tratando-se de uma estrutura de dados não-linear, uma das maiores implicações é a existência de múltiplos percursos possíveis. Travessias mais utilizadas:

- Pré-ordem: raiz + sub-árvore esquerda + sub-árvore direita
- (Em-)ordem: sub-árvore esquerda + raiz + sub-árvore direita
- Pós-ordem: sub-árvore esquerda + sub-árvore direita + raiz

Note que: Caso deseje-se obter o valor do nó raiz, o percurso em pré-ordem fornece como primeiro valor o nó raiz, sendo uma resposta rápida à pergunta.



Exemplo: percursos

- Pré-ordem: 100 19 17 3 2 36 25
- Em-ordem: 2 3 17 19 25 36 100
- Pós-ordem: 2 3 17 25 36 19 100

As funções usadas para percorrer árvores se aproveitam de sua natureza recursiva (uma sub-árvore é uma árvore), e, portanto, podem ser expressas de maneira clara e concisa por meio de funções recursivas[‖].

[‖] "É um excelente exercício escrever uma versão iterativa dessa função. A versão abaixo usa uma pilha de nós. O elemento no topo da pilha pode ser um NULL, mas os demais elementos são diferentes de NULL." https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html#balanced

Pré-ordem

Percorre sistematicamente a árvore, começando pelo nó-raiz, seguindo para sub-árvore esquerda inteira, finalizando com a sub-árvore direita.

```
1    void preOrdem(NO* p)
2    {
3        if(p) //true: != NULL
4        {
5            printf("%d ", p->chave); // raiz
6            preOrdem(p->esq);        // sub-arvove esquerda
7            preOrdem(p->dir);        // sub-arvove direita
8        }
9    }
```

(Em-)ordem

Exibe os valores em ordem crescente. Percorre toda a sub-árvore esquerda, passa pela raiz, seguida da sub-árvore direita.

```
1    void emOrdem(NO* p)
2    {
3        if(p) // true: != NULL
4        {
5            emOrdem(p->esq);         // sub-arvove esquerda
6            printf("%d ", p->chave); // raiz
7            emOrdem(p->dir);         // sub-arvove direita
8        }
9    }
```

Pós-ordem

Percorre sistematicamente toda a sub-árvore esquerda, seguida da direita, e, por fim, a raiz.

```
1    void preOrdem(NO* p)
2    {
3        if(p) //true: != NULL
4        {
5            preOrdem(p->esq);        // sub-arvove esquerda
6            preOrdem(p->dir);        // sub-arvove direita
7            printf("%d ", p->chave); // raiz
8        }
9    }
```

Note que: o corpo das três funções de percurso contém os mesmo comandos, uma chamada recursiva da própria função, diferindo entre si somente quanto a ordem na qual a sub-árvore esquerda, sub-árvore direita e a raiz são dispostas.

Nomenclatura percursos

Uma maneira fácil de lembrar o que significa cada possível travessia, é pensar que seus nomes seguem o seguinte padrão: *pré-* e *pós-* se referem a posição de exibição da raiz. E quanto às sub-árvores direita e esquerda, temos o termo *ordem*; sendo a árvore de busca binária, os elementos à esquerda são sempre menores do que a raiz, que por

sua vez é menor do que todos os nós à direita, portanto temos sempre o percorrimento da esquerda antes da direita.

Travessia	1º	2º	3º
Pré-ordem	raiz	esquerda	direita
Ordem	esquerda	raiz	direita
Pós-ordem	esquerda	direita	raiz

Tabela 5: Resumo ideia geral dos percursos em árvores binárias

[POSCOMP 2016 – FUNDATEC][a] A operação de destruição de uma árvore requer um tipo de percurso em que a liberação de um nó é realizada apenas após todos os seus descendentes terem sido também liberados. Segundo essa descrição, a operação de destruição de uma árvore deve ser implementada utilizando o percurso... **R: pós-ordem**

Considerando que a raiz das árvores contém o ponteiro para os dois nós associados (sub-árvore da esquerda e sub-árvore da direita) então é necessário visitar todas subárvores e liberando os nós-folha e posteriormente os nós-raiz. Ou seja, esse percurso deve deixar a raiz por último, caracterizando o percurso pós-ordem.

[a]Essa e outras perguntas do tema árvores podem ser encontradas em https://www.computersciencemaster.com.br/exercicio-arvores/

14.3 Árvores genéricas

14.3.1 Representação por meio de listas

Árvores genéricas podem ser representadas por meio do uso da notação de parêntesis e vírgulas, na qual nós-filho são indicados entre parêntesis e nós-irmão são separados por vírgula (Fig 21).

(A(B(E(K, L), F), C(G), D(H(M), I, J)))
(A(B(E(K, L), F), C(G), D(H(M), I, J)))

14.3.2 Representação como árvore binária

Na prática, é possível representar árvores genéricas, que podem conter nós de graus distintos com inúmeros nós-filho, dificultando seu manejo, como uma árvore binária. Cada nó passa a conter informação (o endereço) para apenas até dois outros nós, chamados 1º filho (um nível abaixo hierarquicamente, filho à esquerda) e um irmão (no mesmo nível da árvore). Vantagens: permite percorrer os filhos de um nó de forma sistemática, de maneira análoga ao percurso dos nós de uma lista.

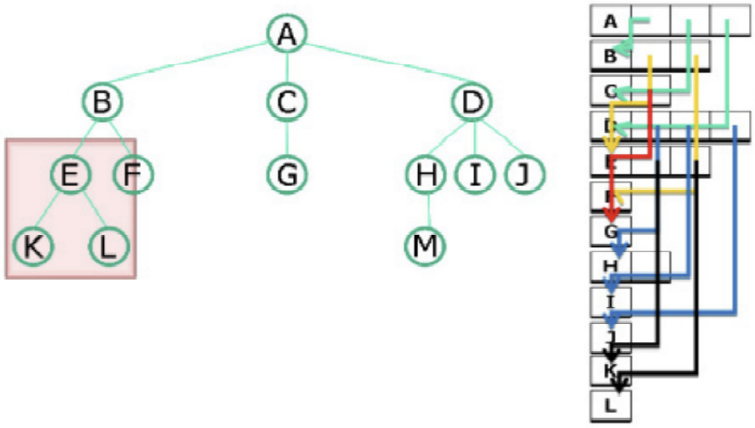


Figura 21: Visualização conceitual da implementação computacional de uma árvore genérica por meio de listas contendo ponteiros para seus nós-filho (também representados por listas).

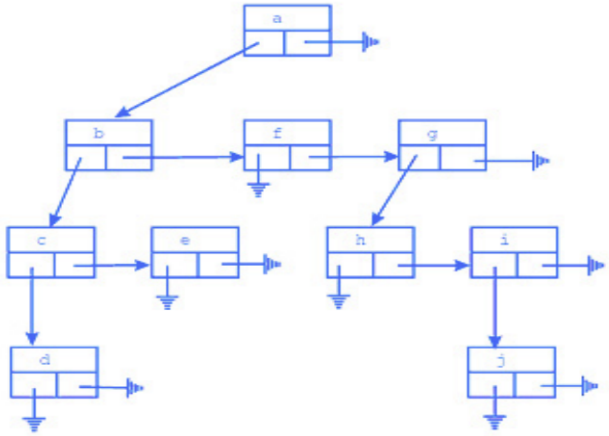


Figura 22: Representação de uma árvore genérica com múltiplos filhos por meio de uma implementação de árvore binária.

Definição da estrutura de uma árvore genérica com essas especificações que a tornam uma árvore binária:

```
// Definicao da estrutura
typedef struct estrutura {
    int chave;  // chave
    struct estrutura *prim; // ponteiro primeiro filho
    struct estrutura *prox; // ponteiro proximo irmao
} NO;

typedef struct {
    *NO raiz;  // ponteiro para raiz da arvore
} ARVORE_GENERICA;
```

Note que: a estrutura é idêntica ao nó da árvore binária, a diferença está somente nos nomes dados aos 2 ponteiros e o que eles significam.

14.4 Árvores balanceadas em altura (AVL)

Árvores AVL, nomeada em homenagem aos seus inventores, Adelson, Velskii e Landis (1962), são **árvores de busca binária balanceadas**:

- ✓Máximo de 2 sub-árvores por nó;
- ✓Ordenação nas chaves: nó à esquerda → menor, nó à direita → maior;
- ✓Balanceada: diferença de altura entre duas sub-árvores máximo 1.

Sua característica de balanceamento contorna a possibilidade de obter-se uma árvore de busca binária degenerada, garantindo que a tarefa de busca seja sempre efetuada com complexidade $log(n)$. Em outras palavras, árvores AVL são projetadas para garantir que a **altura da árvore** seja mantida em um nível logarítmico em relação ao número de nós. Isso proporciona uma eficiência significativa em operações de busca, inserção e remoção.

Vantagens

- Balanceamento automático (etapa incluída nas funções de gerenciamento após cada inserção ou remoção, realizado no momento da operação): garante a altura da árvore mantida em $O(log(n))$;
- Desempenho consistente: desempenho das operações básicas (busca, inserção, remoção) em árvores AVL é previsível e consistente, independentemente da ordem de inserção dos elementos;
- Eficiência em operações de busca: caminho até qualquer nó relativamente curto, devido à altura limitada.

Desvantagens

- *Overhead** adicional: custo adicional associado às operações de inserção e remoção, devido ao processo contínuo de manter o balanceamento;
- Uso de memória adicional: exigem o armazenamento de informações adicionais em cada nó para rastrear fatores de balanceamento, em comparação com árvores não balanceadas.

Overhead

Termo utilizado na computação para descrever a quantidade de recursos adicionais (como tempo, espaço de memória, largura de banda, etc.) consumidos por uma determinada operação, processo ou estrutura de dados, mas que não contribuem diretamente para a realização da tarefa principal ou para o resultado final desejado. Em outras palavras, *overhead* refere-se a recursos que são usados além do mínimo necessário para realizar uma operação.

Implementação da estrutura para árvore AVL:

```c
// Definicao estrutura no arvore AVL
typedef struct estrutura{
    int chave;
    struct estrutura *esq;
    struct estrutura *dir;
    int bal;    // fator de balanceamento: {-1, 0, 1}
} NO_AVL;
```

Note que: a única diferença para a estrutura de um nó de árvore binária é a adição de um novo campo `bal`, responsável por guardar o balanceamento de cada nó da árvore.

14.4.1 Fator de balanceamento

Novo conceito, responsável por caracterizar as árvores AVL. O fator de balanceamento (`bal`) é calculado para todo e cada nó da árvore, sendo atualizado após cada operação de inserção e remoção. Ele é dado pela diferença de altura entre a sub-árvore esquerda e direita do nó em questão:

$$\texttt{bal} := \Delta h = h_{esq} - h_{dir} \ ^{**}$$

```c
// Funcao auxiliar: altura do no
int altura(NOAVL* raiz)
{
    if (!raiz)
        return 0;
    else
    {
        int altEsq = altura(raiz->esq);
        int altDir = altura(raiz->dir);
        return max(altEsq,altDir) + 1;
    }
}
```

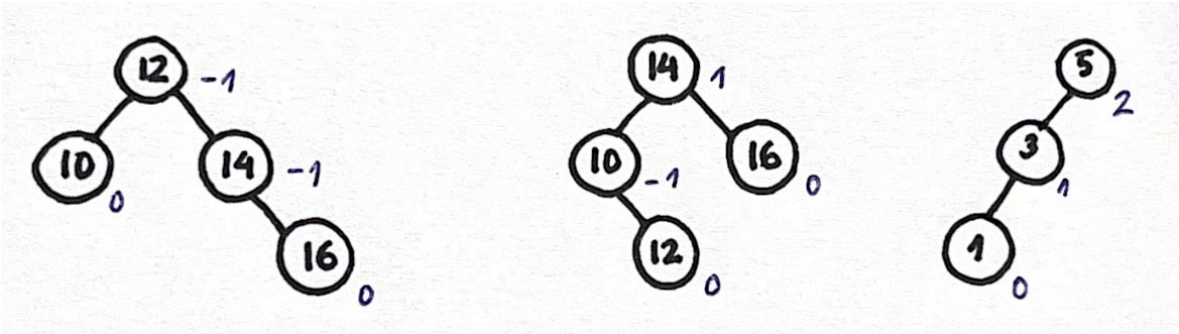


Figura 23: Fator de balanceamento para cada nó das respectivas árvores de busca binária. As duas primeiras árvores (*da esquerda para direita*) apresentam fatores de balanceamento +1 e -1, logo balanceadas, portanto, AVL. A última árvore, por sua vez, apresenta nó com balanço +2, implicando seu desbalanceamento, logo é uma árvore de busca binária, mas não AVL.

**De maneira equivalente, é possível definir `bal` como $h_{dir} - h_{esq}$, o importante é $|\Delta h| \leq 1$.

14.4.2 Funções de gerenciamento

As operações básicas com as árvores AVL de inserção, busca e remoção são as mesmas que para a árvore de busca binária, no entanto, é preciso atenção extra para garantir que ao final do processo a árvore mantenha seu balanceamento. Isso é feito por uma série de passos extras, conhecidos como operações de rotação.

14.4.3 Operações de rotação

As operações de rotação para reajustar o balanço de uma árvore AVL após inserção ou remoção de um nó são classificadas em 4 rotações:

Rotação	Tipo	Descrição
RR	Simples	$h_{dir} > h_{esq}$, $\mathbf{p} \leq \mathbf{u} \leq \mathbf{x}$
LL	Simples	$h_{esq} > h_{dir}$, $\mathbf{x} \leq \mathbf{u} \leq \mathbf{p}$
RL	Dupla	$h_{dir} > h_{esq}$, $\mathbf{p} \leq \mathbf{x} \leq \mathbf{u}$
LR	Dupla	$h_{esq} > h_{dir}$, $\mathbf{u} \leq \mathbf{x} \leq \mathbf{p}$

Tabela 6: Operações de rotação árvore AVL

Nomenclatura da rotação XY: combinação esquerda (*Left*) ou direita (*Right*) que leva *do nó desbalanceado até o novo nó*. Indica o caminho: XY → primeiro toma a sub-árvore X, depois a sub-árvore Y.

O que a rotação XY faz é **compensar** o desbalanço causado em **p** pela adição do nó na posição XY, recuperando o caráter de árvore de busca binária balanceada.

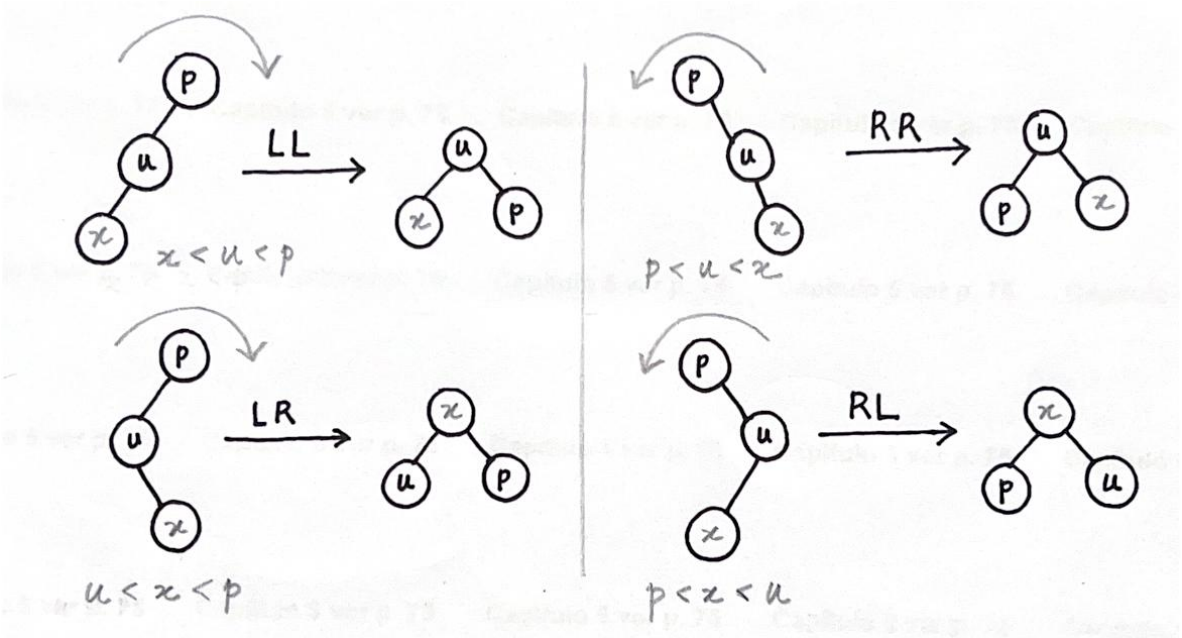


Figura 24: Representação das 4 operações de rotação na árvore AVL.

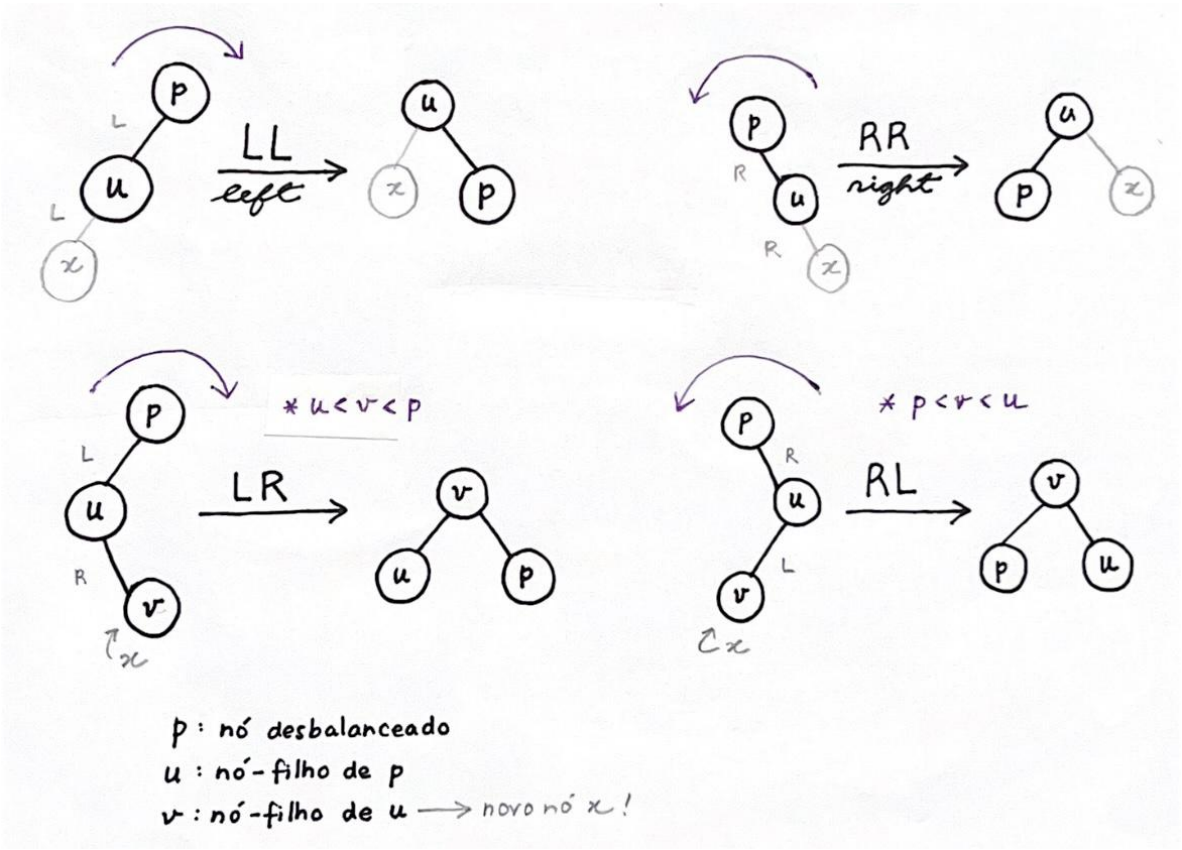


Figura 25: Representação das 4 operações de rotação na árvore AVL, empregando a ideia de rotação simples (RR, LL) e rotação dupla (RL, LR).

- Rotação simples: existe um nó **u** que é nó-filho de **p** (nó desbalanceado) na direção de **x**
- Rotação dupla: o nó **u**, filho de **p** (nó desbalanceado), possui ainda um filho esquerdo ou direito **v**, também localizado na direção de **x** → **v** passa a ser a nova raiz da sub-árvore em questão

Acho que seria o caso de **v** como um intermediário entre o nó-filho **u** do nó desbalanceado **p** e o novo nó em si, que seria **x**.

Conhecendo-se a posição **x** em que um nó foi inserido, é preciso verificar seus **ancestrais** em busca de um possível nó **p** que tenha sido desbalanceado após a operação. Feita por meio de uma função recursiva, essa busca ocorre da folha em direção à raiz, de baixo para cima. Caso seja identificado um nó desbalanceado **p**, este deve sofrer imediatamente uma **operação de rotação** para restaurar o equilíbrio. *Modus operandi*: achou problema, já conserta.

x: novo nó
p: nó desregulado mais próximo da folha **x**
u: nó-filho do nó desregulado

Desse modo, a operação de rotação gira em torno de **p**, e **somente a sub-árvore do nó desbalanceado é afetada**.

Implementação computacional: no código em C, as rotações nada mais são do que ajuste de ponteiros.

Até então foi discutido como transformar a árvore resultante da inserção de um novo nó em uma árvore final balanceada AVL. No entanto, assim como após uma inserção a árvore pode se desbalancear, o mesmo é possível após uma **remoção**. Deve-se realizar a rotação conveniente para transformá-la de volta em AVL[††].

15 Árvores não binárias

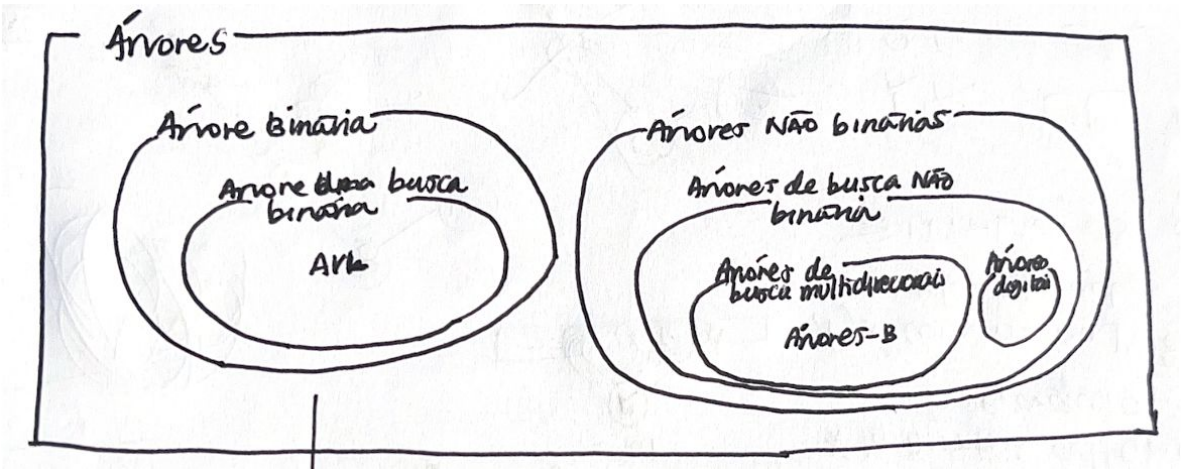


Figura 26: Relações entre árvores no universo das árvores

[††]Não entraremos aqui nas minúcias da remoção. Imagino que não seja trivial e algo me diz que não é um simples oposto dos casos analisados na inserção, pois enquanto a adição de um nó só pode ser feito em um nó folha, a remoção pode acometer qualquer nó da árvore, inclusive o raiz. Mais, deve muito provavelmente remover como na árvore de busca binária, porém com o passo extra de ajustar o balanceamento.

15.1 Árvores multidirecionais

15.2 Árvores-B

Árvores-B são **árvores de busca multidirecional de ordem n balanceada**. É o equivalente a uma versão das árvores AVL para as árvores multidirecionais.

✓Atende a todos os critérios das árvores multidirecionais;
✓Garante todos os nós no mesmo nível.

Estrutura: Nó com j chaves + $(j+1)$ ponteiros.

Implementação árvore-B:

"Aqui está um exemplo simples de implementação de uma árvore-B de ordem 2 (t=2) em C. Neste exemplo, cada nó contém até 2 chaves e até 3 ponteiros para filhos"

```c
#define T 2 // ordem da arvore

// Definicao da estrutura no da arvore-B
typedef struct NO{
    int numChaves;
    int chaves[2 * T - 1];
    struct NO *filhos[2 * T];
    int folha;  // 1: folha, 0: no interno
} NO;
```

"Aqui está um exemplo de como você pode definir uma struct de árvore-B usando `union` para representar tanto nós folha quanto nós internos"

```c
    #define T 2

    // Definicao da estrutura no da arvore-B
    typedef struct TreeBNode {
        int numChaves;
        int chaves[2 * T - 1];
        union {
            struct TreeBNode* filhos[2 * T];
            int dados[2 * T - 1];  // Poderia ser um campo de dados se for uma folha
        };
        int folha;  // 1:folha, 0: no interno
    } TreeBNode;
```

Vantagens

- Permite pesquisas, acesso sequencial, inserções e remoções em tempo logarítmico
- Muito empregada em aplicações que necessitam manipular grandes quantidades de informação tais como um banco de dados ou um sistemas de arquivos.

15.3 Árvores digitais

Árvores formadas por nós cujo conteúdo são a representação da própria chave. Também conhecidas como árvores trie, são estruturas de dados especializadas projetadas para armazenar e recuperar conjuntos de dados associados a chaves, geralmente strings ou sequências de caracteres.

Principais utilidades

- Especialmente valiosas em cenários onde a busca por prefixo é uma operação comum e eficiente;
- Úteis em aplicações como sistemas de autocompletar, corretores ortográficos e busca de palavras-chave em motores de busca;
- Armazenamento eficiente de dicionários: a estrutura compacta permite um acesso rápido e eficiente aos elementos armazenados.

16 Matrizes esparsas

Matrizes com muitos zeros, e pouquíssimos valores não-nulos. Para matrizes muito grandes, para economizar memória, é possível reter a informação completa da matriz armazenando somente os dados válidos (não nulos).

Abordagem comum: listas encadeadas. Cada nó guarda linha, coluna, chave e um ponteiro para o próximo nó (ordenados por linha para facilitar a organização).

```c
// Estrutura de um no da lista (matriz esparsa)
typedef struct Node {
    int linha;
    int coluna;
    int valor;
    struct Node* proximo;
} Node;
```

Vantagens

- Economia de espaço desta representação é bastante significativa

Para uma matriz de $MAXLIN$ x $MAXCOL$ elementos, dos quais n são não-nulos, seu uso será vantajoso (do ponto de vista da complexidade de espaço) sempre que:

$$(MAXCOL * MAXLIN * sizeof(TIPOINFO)) > (n * (sizeof(NO))$$

17 Listas generalizadas

Lista generalizada é uma estrutura de dados que pode **armazenar elementos de tipos diferentes** em uma única lista. Cada elemento da lista pode ser um átomo (um

item simples) ou uma sub-lista (outra lista generalizada). Isso oferece **maior flexibilidade** em comparação com listas simples, onde todos os elementos são do mesmo tipo.

Para decidir se um nó armazena uma chave ou um ponteiro de sublista, usamos um campo `tag` cuja manutenção é responsabilidade do programador. Dependendo do valor de `tag`, armazenamos em um campo de tipo variável (um `union` em C) o dado correspondente.

Implementação da estrutura:

```
typedef struct estrutura {
    int tipo; // 1=elemento e 2=sublista // tag
    union {
        int chave;                    // atomo
        struct estrutura *sublista; // ponteiro para sub-lista
    };
    struct estrutura *prox;
} NO;
```

A principal vantagem das listas generalizadas é sua **capacidade de representar dados heterogêneos** e **hierarquias complexas** de forma flexível. Isso permite uma modelagem mais natural de estruturas de dados que não se encaixam bem em estruturas de dados lineares convencionais, como listas lineares ou árvores binárias.

Parte III

Estrutura de Dados II

Programa

Conceitos avançados de análise de algoritmos: método da árvore de recorrência e teorema mestre. Paradigmas de projetos de algoritmos. Métodos de ordenação diretos e avançados: inserção, seleção, bubblesort, quicksort, heapsort. Métodos de busca em memória interna: sequencial, binária e árvores, comparação entre métodos. Espalhamento (hashing). Grafos: noções básicas, representação, percurso e algoritmos clássicos sobre grafos (caminhos mínimos, ordenação topológica, componentes fortemente conectados, árvores geradoras mínimas).

18 Conceitos avançados de análise de algoritmos

18.1 Método da árvore de recorrência

18.2 Teorema Mestre

$$T(n) = aT(n/b) + f(n)$$

Figura 27: Equação de recorrência

$$\begin{aligned}
&f(n) < n^{\log_b a} \Rightarrow T(n) = \Theta(n^{\log_b a}) \\
&f(n) = n^{\log_b a} \Rightarrow T(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(f(n).\log_b n) \\
&f(n) > n^{\log_b a} \Rightarrow T(n) = \Theta(f(n))
\end{aligned}$$

Figura 28: Três casos do método mestre

19 Métodos de ordenação

Tema do Trabalho 2: *Insertion Sort* e *Heapsort*.

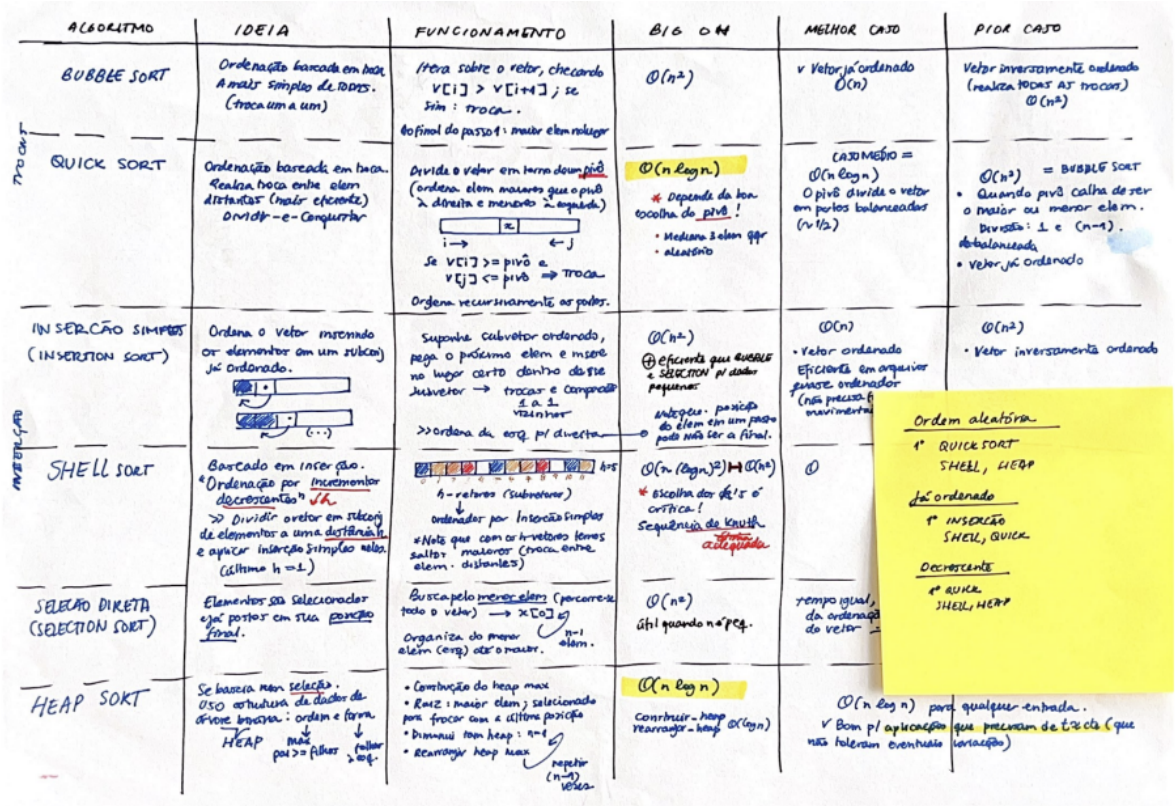


Figura 29: Resumo algoritmos de ordenação

20 Métodos de busca

Serão exploradas técnicas de busca em memória interna, cujo objetivo geral consiste em encontrar um dado registro com o **menor custo**.

Terminologia

- **Algoritmo de busca**: algoritmo que aceita um argumento A e tenta encontrar registro cuja chave seja A;
- **Tabela**: nome geral para uma estrutura de dados usada para armazenamento;
- **Registro**: elementos que compõem a tabela;
- **Chave**: item de diferenciação entre os registros. Existe uma chave associada a cada registro.

20.1 Busca sequencial

A busca mais simples que há: percorre todos os elementos da tabela, registro por registro em busca da chave.

Complexidade (*Big Oh*) $\longrightarrow \mathcal{O}(n)$

- ✓*Melhor caso*: primeiro elemento → 1 comparação
- ✓*Pior caso*: último elemento → N comparações
- ✓*Caso médio*: N/2 comparações
- ✓*Busca mal sucedida*: percorre toda a tabela → N comparações

Ao se considerar uma **tabela ordenada**, a eficiência da busca sequencial melhora (no entanto, deve-se considerar o custo de manter a tabela ordenada no caso de múltiplas inserções e deleções). Há uma nova condição de parada: a busca é encerrada caso se encontre registro com chave maior do que a procurada.

- ✓*Pior caso (busca mal sucedida)*: último elemento → N comparações
- ✓*Caso médio*: N/2 comparações

20.2 Busca binária

O famoso "dividir-e-conquistar", muito útil caso os dados estejam ordenados.

Procedimento recursivo (`T(n/2)`) ou iterativo (`while (fim <= inicio)`) de busca:

O elemento buscado é comparado ao elemento do *meio* do arranjo.

1. Se *igual* → busca bem sucedida
2. Se *menor* → busca na metade inferior do arranjo
3. Se *maior* → busca na metade superior do arranjo

Complexidade (*Big Oh*) $\longrightarrow \mathcal{O}(logn)$, pois a cada comparação reduz o tamanho do arranjo com possíveis candidatos pela metade (mesma complexidade tanto para implementação iterativa quanto recursiva).

Vantagens	*Desvantagens*
✓Eficiência de busca	× Nem todo arranjo está ordenado
✓Simplicidade implementação	× Exige lista sequencial (índices inteiros)
	× Inserção e remoção ineficientes

20.3 Busca por interpolação

Técnica de busca restrita a **arranjos ordenados**.

Caso as chaves estejam *uniformemente distribuídas*, a busca por interpolação *Interpolation Search*) consegue superar até mesmo a busca binária em termos de eficiência.

Busca binária + interpolação → redução do nº de comparações

Complexidade (*Big Oh*) $\longrightarrow \mathcal{O}(log(logn))$; distribuição uniforme arranjo ordenado.

✓Caso as chaves *não* estiverem uniformemente distribuídas, a busca por interpolação pode ser tão ruim quanto busca sequencial.

Vantagens	*Desvantagens*
✓Eficiência de busca caso chaves uniformemente distribuídas	× Na prática, chaves tendem a se aglomerar; não há distribuição uniforme

20.4 Busca em árvores

Importante saber: para uma árvore AVL temos complexidade $\mathcal{O}(logn)$.

20.5 Resumo: Métodos de busca

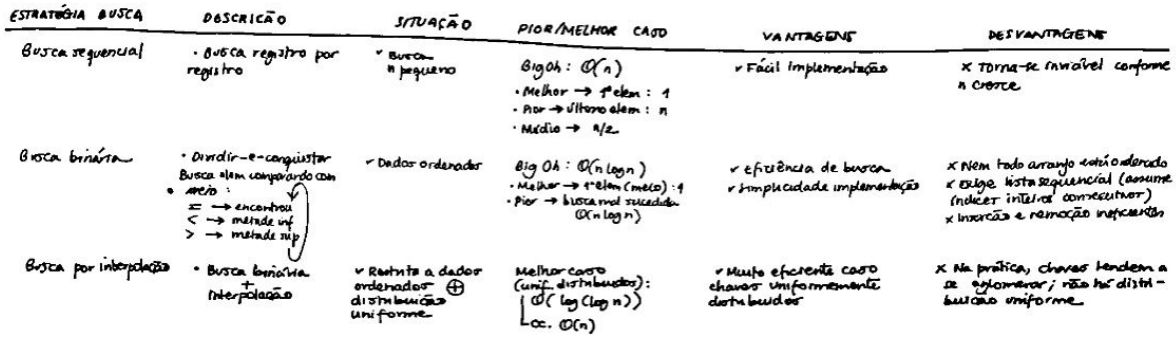


Figura 30: Comparação entre métodos de busca

Até então, os métodos de busca vistos se baseiam em *comparações entre as chaves*, as quais se tenta ao máximo reduzir, armazenando os elementos ordenados e tirando proveito dessa ordenação. Custo de ordenação no melhor caso: $\mathcal{O}(nlogn)$. Portanto, os algoritmos de busca **mais eficientes** demandam esforço computacional, custo de busca $\mathcal{O}(logn)$ (busca binária), e para uma situação com condições muito específicas e propícias de distribuição uniforme de dados em uma *array*, $\mathcal{O}(log(logn))$ (busca por interpolação).

21 *Hashing* (espalhamento)

Técnica de busca[‡‡] que promete tempo de execução constante $\mathcal{O}(c)$, idealmente $\mathcal{O}(1)$, não importando o tamanho da tabela. Ou seja, o elemento procurado estaria a apenas uma operação de leitura de distância. O método de busca *hashing* se baseia não

[‡‡]Tema do Trabalho 3.

na comparação entre chaves (como os métodos de busca vistos anteriormente (Seção 20)), mas no **cálculo de endereço**.

Endereçamento direto	*Hashing*
✓Acesso direto: arranjo indexado ✓Acesso rápido × Uso ineficiente do espaço	✓Acesso direto, endereçamento indireto ✓$\mathcal{O}(c)$ em média, independe do tamanho do arranjo

Elementos *hashing*:

- Chave: objeto a ser armazenado;
- Função de *hash*: retorna endereço para posição de armazenamento;
- Tabela *hash*: estrutura de dados para armazenamento da informação.

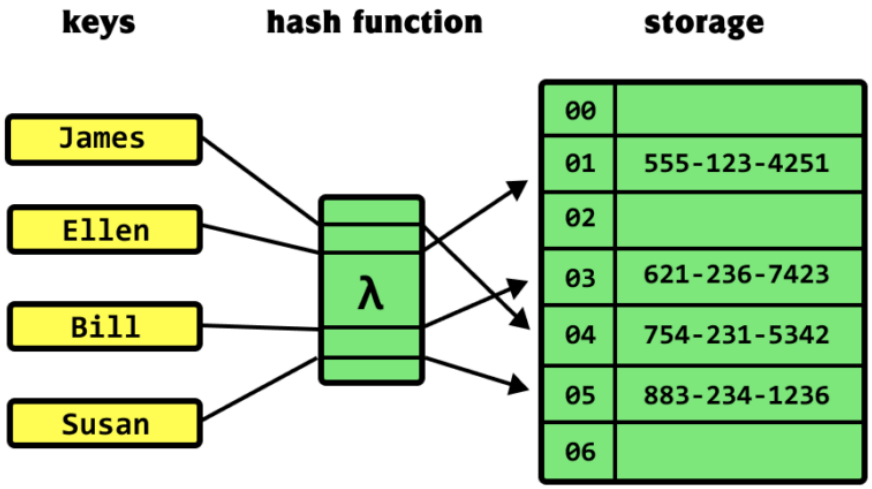


Figura 31: *Hashing*

Hashing é uma técnica que utiliza uma função h (função de *hash*) para transformar uma chave k em um endereço, usado para armazenar e recuperar registros.

Ideia: particionar um conjunto possivelmente infinito de elementos em um número finito de classes (*buckets*): B classes, de 0 a $B-1$.

Pontos importantes para um bom *hashing*:

- Escolha de uma **boa função *hash***
- Estabelecer uma boa estratégia para **tratamento de colisões**

21.1 Função de *hash*

A função h é denominada função de *hash*: $h(k)$ retorna o valor *hash* de k. Nada mais é do que uma função de mapeamento com, em geral, $h(k) \neq k$. Resolve o uso ineficiente do espaço observado para o endereçamento direto ($h(k) = k$). A função de *hash* é usada para localizar uma chave, logo deve ser determinística, sendo utilizada para inserir, remover ou buscar um elemento.

Características de uma boa função de *hash*:

- Distribui uniformemente os dados (*hash* uniforme);
- Cada chave tem igual probabilidade de ser mapeada para qualquer umas das B posições da tabela;
- O mapeamento de uma chave é independente* do mapeamento de outra chave qualquer;
- Evita colisões;
- É fácil/rápida de computar

(*) Em geral, não é possível verificar essa condição; não temos acesso à função de distribuição de probabilidades que gerou as chaves e muitas vezes, elas não são obtidas de forma independente.

Importante salientar que uma função de *hash* deve espelhar o que se deseja. No caso de compiladores, por exemplo, muitas vezes é desejável que chaves próximas sejam armazenadas em locais próximos. Por outro lado, normalmente, procura-se uma função h de aparente aleatoriedade, ligada com a ideia de *hash* uniforme, com menor chance de colisões.

Exemplo de função de *hash* para chaves tipo `string`:

```
1   // Funcao de espalhamento (funcao de hash)
2   int hash(char *palavra, int S){ // palavra, tamanho tabela
3       int v = 0;
4       for (int i = 0; palavra[i] != '\0'; i++){
5           v += v * 3 + (int)palavra[i]; // valor ASCII
6           v = v % S;                    // evita segfault
7       }
8       v = v % S;
9       return v;
10  }
```

21.2 Tratamento de colisões

Por melhor que seja a função de *hash* escolhida, é praticamente impossível que eventualmente dois elementos não sejam mapeados em um mesmo endereço (h não necessariamente é injetiva). Nesse caso, dizemos que ocorre uma **colisão**: a função de

hash produz o *mesmo endereço* para *chaves diferentes*.

OBS: Na prática, funções de *hashing* "**perfeitas**" ou "**quase perfeitas**" são encontradas apenas em aplicações onde colisões são **não** toleráveis, por exemplo, no contexto da criptografia ou quando é conhecido de antemão o conteúdo a ser armazenado na tabela. Nas tabelas **comuns**, colisões são apenas indesejáveis, na medida em que diminuem o desempenho do sistema, *i.e.* o custo de se obter o elemento buscado.

Principais causas de colisões:

* Em geral, número de chaves possíveis N é bem maior do que o tamanho da tabela (número de *buckets* B);

* Não se pode garantir que as funções de *hashing* possuam de fato um bom potencial de distribuição (espalhamento).

Para um bom *hashing*, precisamos estabelecer uma boa estratégia para lidar de maneira sistemática com as colisões inerentes do processo prático de se armazenar registros em uma tabela *hash*. Antes de adentrar nas estratégias de tratamento de colisões é necessário considerar qual tipo de *hashing* estamos lidando com, em termos do espaço de endereçamento da tabela *hash* e do tamanho do conjunto de informações possíveis de serem armazenadas (Fig. 32).

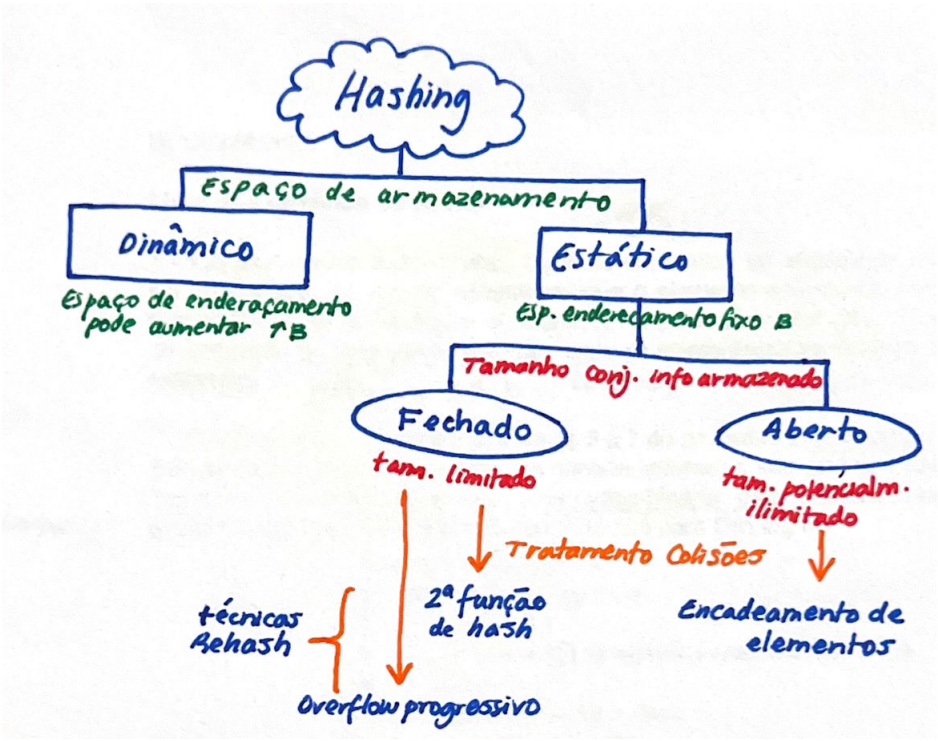


Figura 32: Tipos de *hashing*

Características *hashing* estático fechado:

× Com a tabela cheia, a busca fica lenta;
× Dificulta inserções e remoções.

21.2.1 *Hashing* estático fechado: *rehash*

No caso de *hashing* estático fechado, caso ocorra uma colisão, é aplicada uma técnica de *rehash*:

posição $h(k)$ ocupada → técnica de *rehash* → próximo *bucket* livre

Características de uma boa técnica de *rehash*:

- Cobre ao máximo os índices de 0 a $B - 1$;
- Evita aglomerações de dados.

Overflow progressivo

Definição: O *overflow* progressivo é uma técnica em que, quando ocorre uma colisão, a chave que causou a colisão é armazenada no próximo espaço livre disponível na tabela, obtida via sondagem.

→ **Sondagem linear**: `h(k, i) = (k + i) % B`

Ideia: "pula para casa vizinha"; o próximo espaço livre é obtido percorrendo a tabela sequencialmente de forma linear. No pior caso (tabela cheia), todas as posições da tabela são checadas. ✓Cobre todos os índices × não evita *clustering*.

→ **Sondagem quadrática**: `h(k, i) = (k +` c_i `* i +` c_2 `*` i^2`) % B`

Ideia: A próxima posição livre é encontrada não mais pulando para a casa vizinha, mas para uma posição mais distante dada por uma equação quadrática. É considerada uma técnica de *rehash* melhor do que a sondagem linear, pois evita "mais" o agrupamento de elementos.

Exemplo *rehash* - tratamento de colisões com sondagem quadrática:

```
1    // Funcao rehash: tratamento de colisoes
2    int rehash(char *palavra, int S, int iter, int C1, int C2){
3        // Tratamento de colisao: Sondagem quadratica
4        // input: palavra, tamanho tabela hash, iteracao, cte 1, cte 2
     (sondagem quadratica)
5        return ( (hash(palavra, S) + C1*iter + C2*iter*iter) % S );
6    }
```

2ª função de *hash* (*hash* duplo)

Definição: Técnica de tratamento de colisões em tabelas *hash*, onde duas funções *hash* distintas são usadas para calcular os índices. Se uma colisão ocorrer, a segunda função *hash* é usada para calcular um incremento (ou salto) para a próxima posição a ser sondada.

Uso de 2 funções: $h(k)$, função *hash* primária, e $h_{aux}(k)$, função *hash* secundária.

✓Evita aglomerações de dados, no geral;
× Se uma já era difícil, encontrar duas funções de *hash* que satisfaçam, ao mesmo tempo, os critérios de cobrir o máximo de índices da tabela e evitar aglomerações de dados é uma dificuldade a ser enfrentada;
× Operações de busca, inserções e remoções são difíceis.

21.2.2 *Hashing* estático aberto: encadeamento de elementos

No caso de *hashing* estático aberto, os *buckets* da tabela de *hash* contém apenas **ponteiros** para uma lista de elementos. A característica principal do tratamento de colisões por encadeamento em *hashing* aberto é o fato da informação principal ser armazenada em **estruturas encadeadas fora da tabela *hash***.

Aberto é sinônimo de que sempre cabe mais um elemento na tabela; uso de alocação dinâmica para inserção de novos nós na lista encadeada. Colisões não configuram uma dificuldade no momento de inserção, mas podem tornar a busca de uma chave mais custosa, na medida em que deve percorrer sequencialmente a lista encadeada nó a nó. Note que, caso as listas estiverem ordenadas, o tempo de busca é reduzido (no entanto, deve-se considerar o custo de manter a lista ordenada).

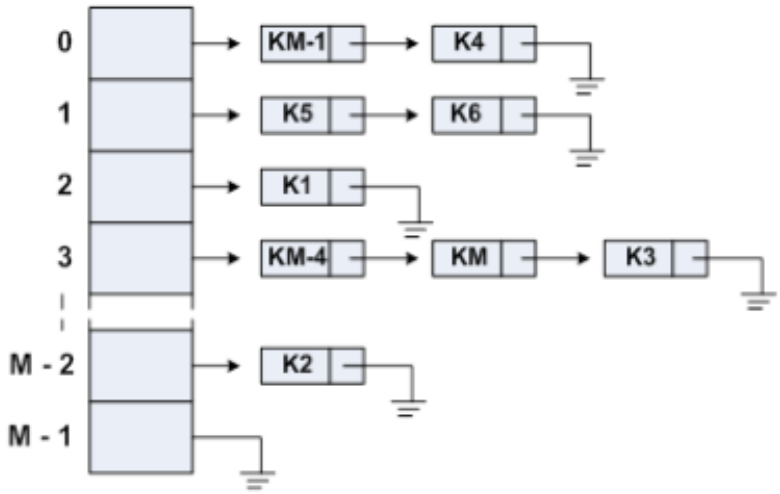


Figura 33: *Hashing* estático aberto - tratamento de colisões por encadeamento.

✓A tabela pode armazenar mais itens, mesmo quando um *bucket* já foi ocupado;
× Listas longas implicam em muito tempo gasto na busca (*possível solução*: trocar listas por árvores binárias de busca balanceadas (AVL)).

22 Grafos

22.1 Conceitos de teoria de grafos

Um **grafo** $G(V, A)$ é um objeto matemático formado por um conjunto de elementos v_i chamados **vértices** V e por um conjunto de pares de vértices (v_i, v_j) chamados **arestas** A. É possível associar um valor w_{ij} às conexões entre os vértices, chamado peso, e, caso os pesos sejam não nulos/unitários, o grafo passa a ser considerado um *grafo ponderado* $G(V, A, W)$. O tamanho ou dimensão ou ordem de um grafo é dado por $n = |V|$, ou seja, por seu número de vértices.

Grafos são uma estrutura onipresente em computação.

Grafos são estruturas não-lineares que representam conexões entre objetos. Caso essas relações sejam simétricas, *i.e.* $\exists(v_i, v_j) \in A \implies \exists(v_j, v_i) \in A$ com mesmo peso $w_{ij} = w_{ji}$, o grafo é dito não-direcionado (ou não-dirigido, ou não-orientado). Em um grafo não-dirigido, a relação de adjacência é simétrica: um vértice u é adjacente a um vértice v, se e somente se, v é adjacente a u. Em contra-partida, em um *grafo orientado* (ou dirigido, ou direcionado), o par de vértice (v_i, v_j) é de fato um par ordenado, de modo que a existência de uma aresta $v_i \to v_j$ não implica a existência da aresta de "volta" $v_j \to v_i$.

Em um contexto em que não são permitidas múltiplas arestas ligando um mesmo par de vértices (v_i, v_j) e nem auto-*loops* (v_i, v_i), o número máximo de arestas que um grafo pode apresentar é dado por $max|A| = n(n-1) = n^2 - n$. Quando um grafo possui todas as arestas possíveis, ou seja, todos os vértices conectados a todos os outros vértices (todos os vértices adjacentes), este é chamado *grafo completo*.

Um grafo é dito *simples* (ou *regular*) se ele não possui laços (auto-*loops*) nem mais de uma aresta ligando dois vértices. Um grafo que não é simples é chamado *multigrafo*.

• Vértices adjacentes (ou *vizinhos*): os vértices v_i e v_j são ditos adjacentes se são os extremos de uma mesma aresta $a_{ij} = (v_i, v_j)$, ou seja, se existe aresta entre eles;
• Arestas adjacentes (ou *vizinhas*): duas arestas são ditas adjacentes se possuem um mesmo vértice extremo: a_{ij} e a_{jk}

Em relação ao *grau* de uma vértice (em grafos direcionados $d \to d_{in}$ e d_{out}):
• Grafo *regular*: todos os vértices possuem o mesmo grau;
• *Sumidouro*: um vértice com grau de saída d_{out} nulo. Ao chegar nesse vértice é impossível sair (a menos de teletransporte);
• *Fonte*: um vértice com grau de entrada d_{in} nulo, há somente saídas, não é possível chegar no vértice.

Um *caminho* em um grafo entre dois vértices v_i e v_j é uma sequência de arestas que conectam v_i e v_j. O *comprimento* de um caminho é dado pela quantidade de arestas que o compõe, ou seja, um caminho de k-vértices apresenta comprimento de $k - 1$.

- Caminho *simples*: se todos os vértices que compõem o caminho são distintos;
- *Ciclo*: caminho no qual o vértice de partida e de chegada são os mesmos;
  → Um grafo é *cíclico* se apresenta pelo menos um ciclo.

Há dois tipos de caminhos muito famosos dentro da teoria de grafos, os caminhos *eulerianos* (em homenagem a Euler, o fundador do que se tem hoje em dia como teoria dos grafos) e os caminhos *hamiltonianos*, que se diferenciam ao considerar a unicidade das arestas ou dos vértices no caminho, respectivamente:

• **Caminho *euleriano***: caminho que contém cada e toda aresta do grafo uma única vez (problema de Königsberg);

→ Um caminho euleriano pode formar um *ciclo euleriano*: caso especial do caminho euleriano, no qual o caminho começa em um vértice qualquer, passa por todas as arestas exatamente uma vez, e termina no vértice inicial;

→ Um grafo é *euleriano* se possui um ciclo euleriano.

• **Caminho *hamiltoniano***: caminho que contém cada e todo vértice do grafo uma única vez (problema do caixeiro-viajante);

→ Um caminho hamiltoniano pode formar um *ciclo hamiltoniano*: caso especial do caminho hamiltoniano, no qual o caminho começa em um vértice qualquer, passa por todos os vértices exatamente uma vez, e termina no vértice inicial;

→ Um grafo é *hamiltoniano* se possui um ciclo hamiltoniano.

Em termos da conectividade do grafo, um grafo é dito *conexo* se existe um caminho entre cada par de vértices do grafo, ou seja, se todos os vértices estão conectados a todos os vértices. Caso contrário, o grafo é dito *desconexo*.

Um grafo *conexo* é *euleriano* $\iff$ Todos os seus vértices tem grau par

Para grafos orientados, este é *fortemente conexo* se existe um caminho de ida e volta entre cada par de vértices v_i ev_j.

22.2 Representação de grafos

A escolha da *estrutura de dados adequada* para representação de um grafo tem um enorme impacto no desempenho de um algoritmo. Há duas representações usuais mais comumente empregadas: matriz de adjacências e listas de adjacências.

Representação	Complexidade de espaço
Matriz de adjacências	$O(\|V\|^2)$
Listas de adjacências	$O(\|V\| + \|A\|)$

Tabela 7: Comparação entre as complexidades das representações de grafos

22.2.1 Matriz de adjacências

Na representação de um grafo por sua matriz de adjacência A, o objeto é armazenado em uma matriz quadrada $n \times n$ de tamanho correspondente ao número de vértices do grafo. Para um grafo sem pesos $G(V, A)$:

$$a_{ij} = \begin{cases} 1, & (v_i, v_j) \in A \\ 0, & c.c. \end{cases}$$

No caso de um grafo ponderado $G(V, A, W)$, os elementos da matriz de adjacência indicam não somente a conexão, como guardam o valor do peso correspondente da aresta, dados por:

$$a_{ij} = \begin{cases} w_{ij}, & (v_i, v_j) \in A \\ 0, & c.c. \end{cases}$$

Note que: dada uma matriz de adjacência, temos uma representação *única* de um grafo. No entanto, do outro lado da associação, temos que dado um grafo, pode-se chegar a diferentes matrizes de adjacência (permutações de linhas e colunas uma das outras; depende de como rotulamos os vértices do grafo).

Implementação da representação do grafo $G(V, A, W)$ via matriz de adjacências:

```
// Estrutura grafo: matriz de adjacencias
// alocacao vetor ponteiros para linhas
int **G = (int **) malloc(N * sizeof(int *));

// alocacao cada uma das linhas
for (int i = 0; i < N; i++){
    G[i] = (int *) malloc(N * sizeof(int));
}

// Inicializacao grafo: conjunto de arestas vazio
for (int i = 0; i < N; i++){
    for (int j = 0; j < N; j++){
        G[i][j] = 0;
    }
}

// Leitura grafo: matriz de adjacencias (arestas (v_i, v_j, w))
int v_i, v_j, w; // vertice origem, vertice destino, custo
for (int i = 0; i < M; i++){
    scanf("%d %d %d", &v_i, &v_j, &w);
    G[v_i][v_j] = w;    // adicionar no grafo
}
```

Custo de armazenamento: $O(|V|^2)$; matriz $n \times n$ $O(n^2)$.

Vantagens:

- Permite acesso direto ao elemento da matriz: custo constante $O(1)$ para dados os vértices v_i, v_j conferir se a aresta (v_i, v_j) está no grafo;

- Representação útil para grafos densos (muita arestas: $|A| > 60\%/70\%$ max$|A|$.

22.2.2 Listas de adjacência

Na representação por listas de adjacência, um grafo é representado por meio do armazenamento explícito apenas das arestas presentes dentre todo o conjunto de arestas possíveis. Evita gasto de espaço para armazenar zeros.

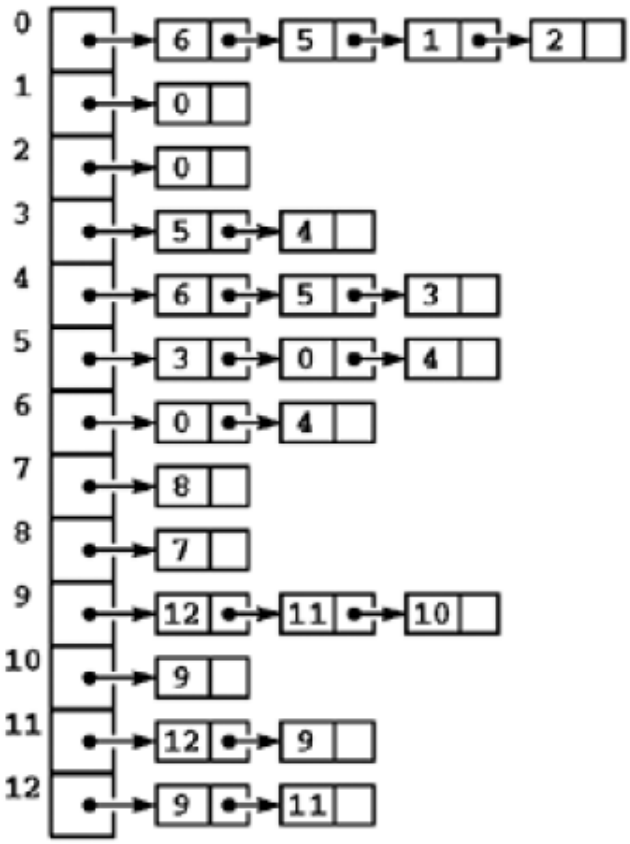


Figura 34: Representação de um grafo por listas de adjacência

Custo de armazenamento: $O(|V| + |A|) = O(n + m)$.

Vantagens:

- Representação compacta, usada de modo geral na maioria das aplicações (é mais comum termos grafos/matrizes esparsas);
- Representação vantajosa para grafos esparsos (poucas arestas: $|V| + |A| << |V|^2$, *i.e.* $|A| < 20\%$ max$|A|$), nos quais o custo do ponteiro é compensado.

22.3 Busca em grafos

Percorrendo um grafo... **algoritmos de travessia**. Precisamos de estratégias sistemáticas para percorrer todos os vértices de um grafo, lembrando de que se trata de

uma estrutura de dados não-linear.

OBS: o processo é o mesmo, é indiferente caso o grafo seja orientado ou não.

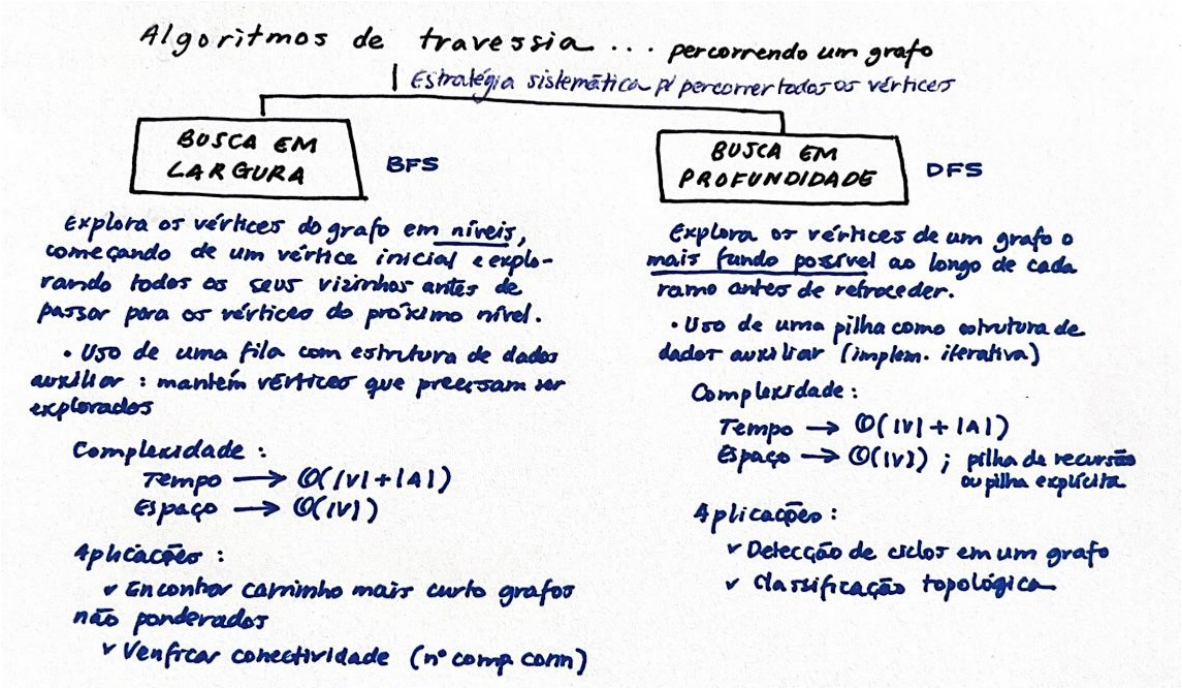


Figura 35: **Busca em Largura (BSF)** *vs.* **Busca em Profundidade (DFS)**

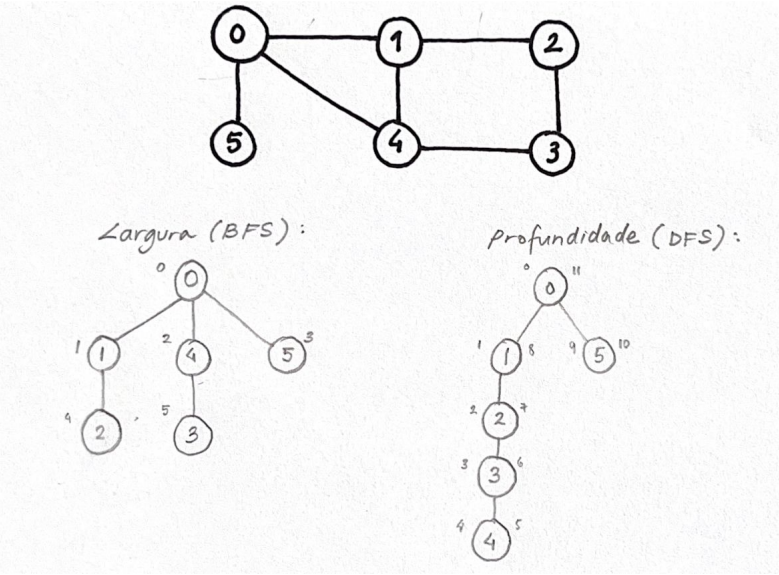


Figura 36: Exemplo algoritmo de travessia para Busca em Largura *vs.* Busca em Profundidade

22.4 Algoritmos de caminhos mínimos

Diferentemente dos algoritmos de busca em grafos (busca em largura), que levam em conta apenas a existência de conexão entre os vértices, os algoritmos de caminhos mínimos contabilizam o cálculo do menor caminho em termos do **peso total das arestas** (soma dos pesos das arestas que compõem o caminho)[§§].

22.4.1 Algoritmo de Dijkstra

→ Menor caminho de uma origem para todos os vértices.

Características:

✓ Algoritmo guloso (*greedy*); ✓ Lida com pesos (somente não negativos);

Elementos do algoritmo de Dijkstra:

- Vetor de distâncias `d[v]`: estimativa de caminho mais curto do nó inicial s até todos os vértices v;
- Vetor solução `S`
- Fila de prioridade: estrutura de dados regida pela regra FIFO (*First-In-First-Out*) com prioridade dada pelo menor custo do caminho.

Ideia/procedimento:

1. Inicialização vetor de distâncias: faça a distância do vértice inicial s a todos os outros ser infinita (na prática, igual a `INT_MAX` usando `#include <limits.h>`), exceto para ele mesmo `d[s] = 0`;
2. Adicionar todos os vértices na fila de prioridade (vértice em aberto). No primeiro momento, todos apresentam o mesmo custo ∞, exceto pelo vértice s de custo zero, logo com prioridade;
3. Enquanto houver vértice na fila, ou da mesma forma, enquanto nem todos os vértices não tiverem sido incluídos no vetor solução:

   1 - Escolher o vértice u cuja estimativa seja a menor dentre os demais (em primeiro lugar na fila de prioridade);

   2 - Adicionar u no conjunto solução S (fechar u);

   3 - Relaxamento de arestas (u, v): para todo nó aberto v na adjacência de u, atualizar sua estimativa no vetor de distâncias caso seja melhor;

22.4.2 Algoritmo de Floyd-Warshall

→ Menor caminho de todos para todos os vértices.

Características:

✓ Paradigma de programação dinâmica ✓ Pesos negativos.

[§§]Tema do Trabalho 4: caminho de menor custo em um grafo ponderado.

22.5 Árvore geradora mínima de um grafo

Árvore geradora mínima de um grafo é uma **subárvore que conecta todos os vértices do grafo com o menor peso total possível**. Isso é aplicável a grafos ponderados, onde cada aresta tem um peso associado.

Propriedades de uma árvore geradora mínima:

- Conectividade: deve conectar todos os vértices do grafo original;
- Arestas: deve conter exatamente $|V| - 1$ arestas
- Peso mínimo: o peso total das arestas na árvore geradora mínima é o menor possível entre todas as árvores geradoras possíveis do grafo.

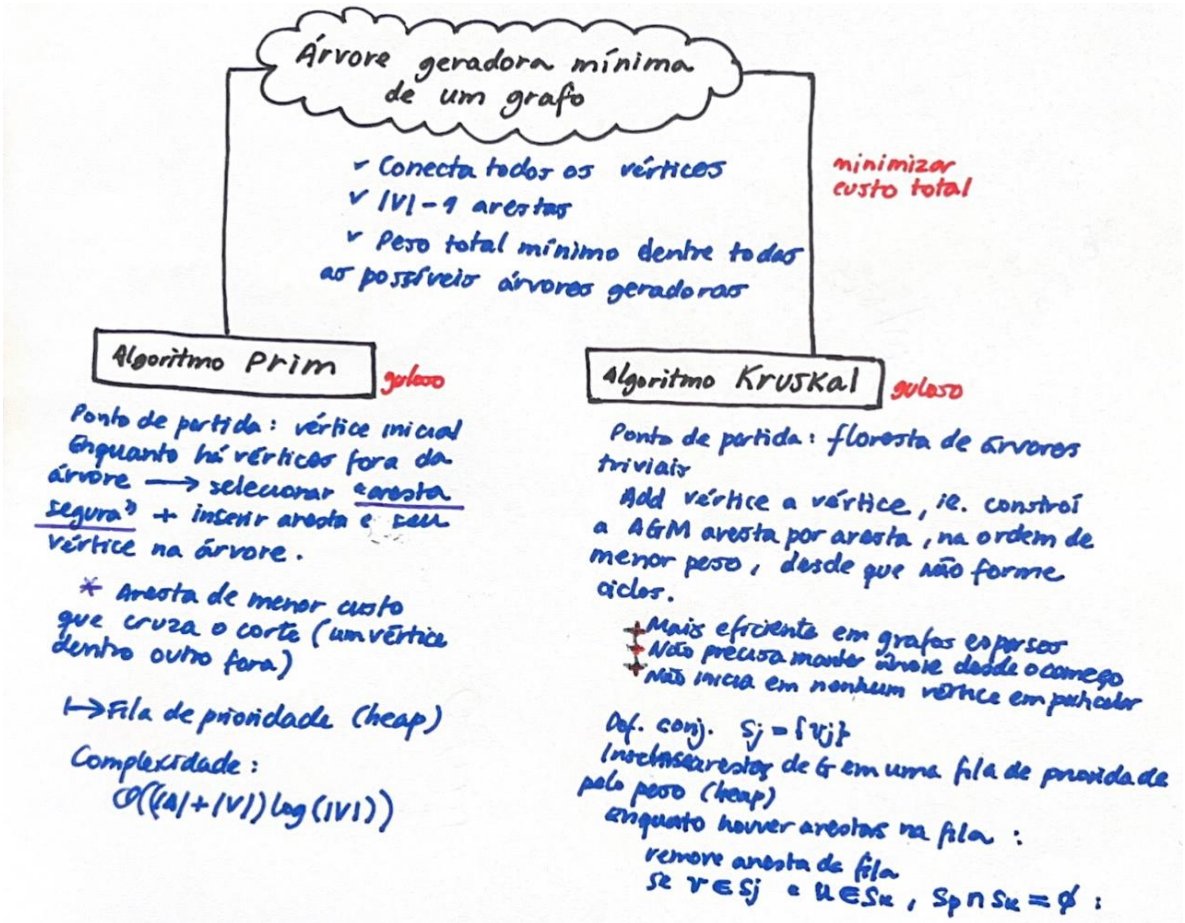


Figura 37: **Algoritmo de Prim** *vs.* **Algoritmo de Kruskal**

A Ambiente de trabalho: minha máquina

Na minha máquina...

- **Ambiente de trabalho**: Visual Studio Code (VS Code) - IDE;
- **Compilador**: gcc - instalado no terminal Ubuntu, mas com *set up* já pronto para rodar no terminal do próprio VS Code;
- **Depurador**: gdb.

Como programar: No VS Code, abrir pasta de interesse - salva na seção do Linux (Ubuntu) no meu computador, já conectada via WSL. Abrir ou criar arquivo com extensão *.c* para começar a programar em C.

B Comandos terminal Linux

B.1 Diretório de trabalho

Ao usar o terminal, é preciso ter em mente que você trabalha de dentro de uma pasta específica em seu computador. É sempre possível acessar itens de outras pastas em seu computador, mas o terminal controlará a pasta na qual você está atualmente. Isso é conhecido como seu **diretório de trabalho**.

Comando	Ação
`pwd`	Informa pasta do diretório de trabalho atual (*print working directory*)
`ls`	Lista todos os arquivos e pastas em seu diretório de trabalho (*list*)
`cd`	Altera diretório de trabalho, desce um nível (*change directory*)
`cd ..`	Altera diretório de trabalho, sobe para a pasta anterior

Tabela 8: Escrever os comandos no terminal e pressionar Enter

Para compilar e rodar um programa, é preciso estar no seu respectivo diretório de trabalho. Estando no lugar correto, em seguida é possível usar com sucesso as linhas de comando para compilar, rodar e debugar o arquivo de interesse.

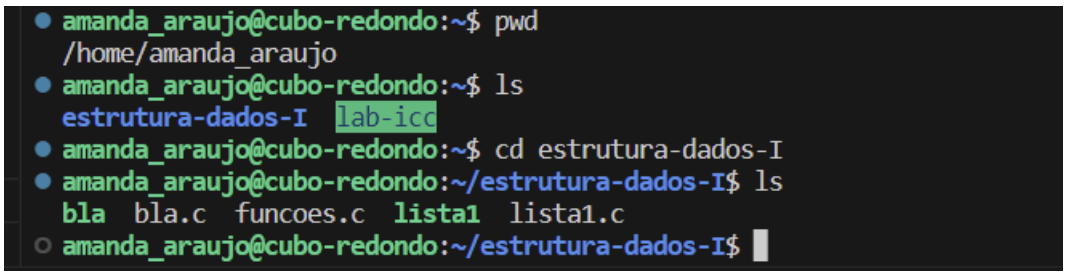


Figura 38: Acessando diretório de trabalho de interesse, onde os códigos em C estão salvos.

Dica: Seta para cima no teclado (↑) permite navegar pelos comandos anteriormente escritos no terminal. Evita o trabalho de reescrever o mesmo comando novamente.

B.2 Compilar, rodar e debugar

Linha de comando (terminal dentro do VS Code)

Compilar	`gcc nomedoarquivo.extensão -o nomedoarquivo`
Rodar	`./nomedoarquivo`
Debug	`gdb ./nomedoarquivo`

Atalho de teclado (shortcut)

Compilar + rodar	*Ctrl* + *Shift* + *B*
Rodar + Debug	*F5*
Comentário linha (//)	*Ctrl* + *;*
Comentário bloco (/**/)	*Shift* + *Alt* + *A*

Referências

1. "Pointers in Python: What's the Point?", *Real Python*, https://realpython.com/pointers-in-python/
2. "Ponteiros em C", https://linguagemc.com.br/ponteiros-em-c/
3. "Manual de Sintaxe da Linguagem C", https://www.feg.unesp.br/
4. "Estrutura de Dados e Algoritmos - COS-121", Ricardo Farias, https://www.cos.ufrj.br/~rfarias/cos121/aula_10.html
5. "ACH2023 - Algoritmos e Estruturas de Dados I", Willian Yukio Honda & Ivandré Paraboni, EACH-USP, http://each.uspnet.usp.br/digiampietri/ACH2023/ACH2023.pdf
6. "Como navegar em arquivos e pastas em um terminal", *Terminal Cheat Sheet*, https://terminalcheatsheet.com/pt-BR/guides
7. "Recursion (Computer Science)", *Wikipedia*, https://en.wikipedia.org/wiki/Recursion_(computer_science)
8. "Estruturas de dados para grafos", *IME-USP*, https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html
9. "Vetores e alocação dinâmica", W. Celes e J. L. Rangel, http://profs.ic.uff.br/~kisch/Apostila-C/ed05-vetores.pdf
10. "Matrizes e Vetores - Programação C/C++", Márcio Sarroglia Pinho, Escola Politécnica - PUCRS, https://www.inf.pucrs.br/~pinho/LaproI/Vetores
11. "Alocação dinâmica de matrizes", https://www.inf.ufpr.br/roberto/ci067/14_alocmat.html