UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

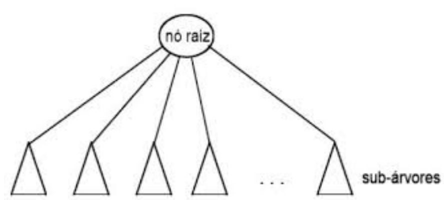
Disciplina: Estrutura de Dados
Semestre: 2020.2
Professor: Leandro Carlos de Souza
Nome: Francisco Siqueira Carneiro da Cunha Neto e Felipe Honorato de Sousa
Matrícula: 20190029015 e 20190031130

Exercícios de Fixação e Aprendizagem III

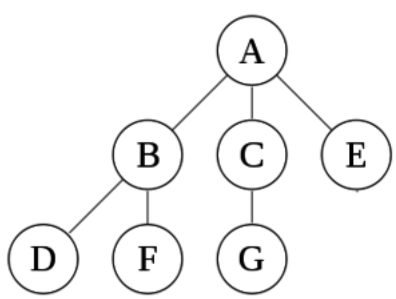
Questão 1.

(a) **Explique, com suas palavras, as seguintes estruturas: árvores, árvores binárias, árvores binárias de busca, árvores de busca balanceadas.**

Árvores são estruturas com modelagem hierárquica, onde partindo de um nó pai, conseguimos alcançar os nós filhos.

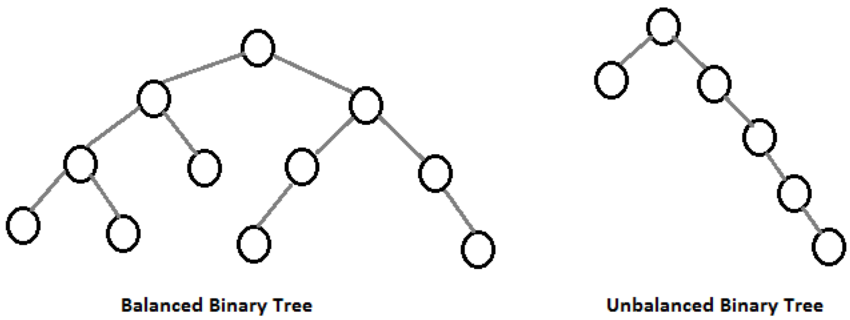


Árvores binárias são aquelas em que o número de filhos de cada nó não excede duas unidades.

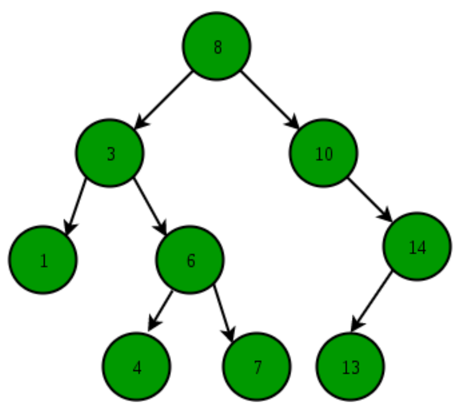


As árvores binárias de busca seguem basicamente um único princípio: todos os nós à esquerda do nó analisado terão valor inferior ao nó de referência, enquanto todos os nós à direita do nó terão valores superiores ao nó de referência.

Para uma árvore ser considerada balanceada, o nó pai da subárvore à esquerda precisa ter uma altura que não destoe em mais de uma unidade em relação à altura do nó pai da subárvore à direita. O balanceamento é de extrema importância para garantir eficiência nas operações de busca exercidas na árvore.



(b) Explique sobre as formas de se percorrer uma árvore binária. Dê um exemplo para cada uma das formas citadas.



Podemos percorrer em pré-ordem, ou seja, em cada nó visitaremos primeiro a subárvore à esquerda, depois visitaremos o nó de referência, e então a subárvore à direita. Assim teríamos:
1 - 3 - 4 - 6 - 7 - 8 - 10 - 13 - 14.
Em pós-ordem visitaremos primeiramente o nó à direita, posteriormente nó de referência, e então a subárvore à esquerda. Assim teríamos:
14 - 13 - 10 - 8 - 7 - 6 - 4 - 3 - 1
Por fim, podemos visitar todos os nós de maneira simétrica, ou seja, visitamos primeiramente o nó referência, posteriormente a subárvore à esquerda/ direita e então visitamos a subárvore restante. Assim teríamos:
8 - 3 - 1 - 6 - 4 - 7 - 10 - 14 - 13

(c) Implemente um TAD para árvores binárias de busca para armazenar chaves inteiras. Esse TAD deve ter as seguintes operações: criação da árvore, inserção de um nó, remoção de um nó, busca de um nó, mostrar árvore e retornar o número de folhas da árvore. Faça um programa para testar o seu TAD.

Código 1: avl.h

```c
#ifndef _AVL_H_
#define _AVL_H_

typedef struct bintree Avl;

Avl *avl_create_node(int data);
Avl *avl_create(int data, Avl *l, Avl *r);
void avl_print_bars(Avl *tree);
void avl_print_parenthesis(Avl *tree);
void avl_free(Avl *tree);
int avl_exists(Avl *tree, int data);
Avl *avl_find(Avl *tree, int data);
int avl_height(Avl *tree);
Avl *avl_insert(Avl **tree, int data);
int avl_remove(Avl **tree, int data);
int avl_num_leaves(Avl *tree);

#endif
```

Código 2: avl.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "../include/avl.h"

#ifndef _AVL_C_
#define _AVL_C_

// Struct da arvore binaria
struct bintree
{
    int data;
    Avl *l;
    Avl *r;
};

// Cria uma nova (sub)arvore, sem especificar seus filhos
Avl *avl_create_node(int data)
{
    return avl_create(data, NULL, NULL);
}

// Cria uma nova (sub)arvore
```

```c
Avl *avl_create(int data, Avl *l, Avl *r)
{
    Avl *tree = (Avl *)malloc(sizeof(Avl));

    if (tree == NULL)
    {
        return NULL;
    }

    tree->data = data;
    tree->l = l;
    tree->r = r;

    return tree;
}

// Imprime uma subarvore por um diagrama de barras
void subtree_print(Avl *tree, int parent_height)
{
    if (tree == NULL)
        return;

    int h = parent_height - 1;
    for (int i = 0; i < h; i++)
    {
        printf("-");
    }
    printf(" %d\n", tree->data);
    subtree_print(tree->l, h);
    subtree_print(tree->r, h);
}

// Imprime a arvore por um diagrama de barras
void avl_print_bars(Avl *tree)
{
    if (tree == NULL)
        return;

    int h = avl_height(tree);
    for (int i = 0; i < h; i++)
    {
        printf("-");
    }
    printf(" %d\n", tree->data);
    subtree_print(tree->l, h);
    subtree_print(tree->r, h);
}

void avl_print_parenthesis(Avl *tree)
{
    if (tree == NULL)
```

```c
        return;

    printf(" (%d", tree->data);
    avl_print_parenthesis(tree->l);
    avl_print_parenthesis(tree->r);
    printf(")");
}

// Libera a memoria alocada para a arvore
void avl_free(Avl *tree)
{
    if (tree == NULL)
        return;

    avl_free(tree->l);
    avl_free(tree->r);

    free(tree);
}

// Checa se a arvore contem o dado informado
int avl_exists(Avl *tree, int data)
{
    if (tree == NULL)
        return 0;

    if (tree->data == data)
        return 1;

    return avl_exists(tree->l, data) || avl_exists(tree->r, data);
}

// Retorna a subarvore cuja raiz tem o dado
Avl *avl_find(Avl *tree, int data)
{
    if (tree == NULL)
        return NULL;

    if (tree->data == data)
        return tree;

    Avl *l = avl_find(tree->l, data);
    Avl *r = avl_find(tree->r, data);

    if (l)
        return l;
    if (r)
        return r;

    return NULL;
}
```

```c

// Retorna o maior valor entre dois inteiros
int max2(int a, int b)
{
    if (a >= b)
        return a;

    return b;
}

// Retorna a altura da arvore
int avl_height(Avl *tree)
{
    if (tree == NULL)
        return -1;

    return 1 + max2(avl_height(tree->l), avl_height(tree->r));
}

// Retorna a diferenca entre a altura da subarvore a direita e a esquerda
// Valores positivos indicam subavore a esquerda maior, negativos indicam
    subarvore a esquerda maior
int avl_balance_factor(Avl *tree)
{
    return avl_height(tree->l) - avl_height(tree->r);
}

// Operacao para restabelecer regulagem: Rotacao a direita
Avl *avl_rotate_right(Avl *tree)
{
    Avl *p = tree;
    Avl *u = p->l;
    Avl *t2 = u->r;

    u->r = p;
    p->l = t2;

    return u;
}

// Operacao para restabelecer regulagem: Rotacao a esquerda
Avl *avl_rotate_left(Avl *tree)
{
    Avl *p = tree;
    Avl *z = p->r;
    Avl *t2 = z->l;

    z->l = p;
    p->r = t2;

    return z;
```

```c
}

// Operacao para restabelecer regulagem: Rotacao dupla a direita
Avl *avl_rotate_double_right(Avl *tree)
{
    tree->l = avl_rotate_left(tree->l);
    return avl_rotate_right(tree);
}

// Operacao para restabelecer regulagem: Rotacao dupla a esquerda
Avl *avl_rotate_double_left(Avl *tree)
{
    tree->r = avl_rotate_right(tree->r);
    return avl_rotate_left(tree);
}

// Se estiver desbalanceada, faz operacoes para restabelecer a regulagem
void avl_balance(Avl **tree)
{
    // Apos insercao, confere balanceamento da arvore
    int balance = avl_balance_factor(*tree);

    if (abs(balance) > 1)
    {
        // Se o filho a esquerda e maior do que o a direita
        if (balance > 0)
        {
            // Se o neto esquerda-direita e maior do que o esquerda-esquerda
            if (avl_balance_factor((*tree)->l) < 0)
                // Rotacao dupla direita
                *tree = avl_rotate_double_right(*tree);
            else
                // Rotacao direita
                *tree = avl_rotate_right(*tree);
        }

        // Se o filho a direita e maior do que o a esquerda
        if (balance < 0)
        {
            // Se o neto direita-esquerda e maior do que o direita-direita
            if (avl_balance_factor((*tree)->r) > 0)
                // Rotacao dupla esquerda
                *tree = avl_rotate_double_left(*tree);
            else
                // Rotacao direita
                *tree = avl_rotate_left(*tree);
        }
    }
}
```

```
// Insere um novo dado na arvore
Avl *avl_insert(Avl **tree, int data)
{
    // Nao permite dados duplicados na arvore
    if (avl_exists(*tree, data))
        return NULL;

    // Se a arvore nao tem raiz, torna ela a raiz
    if (*tree == NULL)
    {
        *tree = avl_create_node(data);
        return *tree;
    }

    Avl *inserted_node = NULL;

    // Se o dado e menor do que a raiz da arvore, insere ele na subarvore
         da esquerda
    if (data < (*tree)->data)
        inserted_node = avl_insert(&((*tree)->l), data);

    // Se o dado e maior do que a raiz da arvore, insere ele na subarvore
         da direita
    if (data > (*tree)->data)
        inserted_node = avl_insert(&((*tree)->r), data);

    // Se necessario, rebalanca a arvore
    avl_balance(tree);

    return inserted_node;
}

int avl_find_parent(Avl *tree, int data, Avl **parent)
{
    if (tree == NULL || tree->data == data)
        return 0;

    if (tree->l != NULL && tree->l->data == data)
    {
        *parent = tree;
        return -1;
    }
    if (tree->r != NULL && tree->r->data == data)
    {
        *parent = tree;
        return 1;
    }

    int l = avl_find_parent(tree->l, data, parent);
    int r = avl_find_parent(tree->r, data, parent);
```

```

    if (l != 0)
        return l;
    if (r != 0)
        return r;

    return 0;
}

// Remove um no de acordo com o dado que ele armazena
int avl_remove(Avl **tree, int data)
{
    if (tree == NULL || !avl_exists(*tree, data))
        return 0;

    Avl *node = avl_find(*tree, data);

    // Se no tem filho a direita e a esquerda
    if (node->r != NULL && node->l != NULL)
    {
        // Encontrar o sucessor
        Avl *sucessor = node->r;
        while (sucessor->l != NULL)
            sucessor = sucessor->l;

        int sucessor_data = sucessor->data;

        if (!avl_remove(tree, sucessor_data))
            return 0;

        node->data = sucessor_data;
    }
    else
    {
        Avl *subtitute;

        // Se no e uma folha
        if (node->r == NULL && node->l == NULL)
            subtitute = NULL;
        // Se no so tem filho a direita
        else if (node->r != NULL && node->l == NULL)
            subtitute = node->r;
        // Se no so tem filho a esquerda
        else if (node->r == NULL && node->l != NULL)
            subtitute = node->l;

        Avl *parent = NULL;
        int dir = 0;
        if ((*tree)->data != data)
            dir = avl_find_parent(*tree, data, &parent);

```

```c
        if (dir == -1)
        {
            free(parent->l);
            parent->l = subtitute;
        }
        else if (dir == 1)
        {
            free(parent->r);
            parent->r = subtitute;
        }
        else if (dir == 0)
        {
            free(tree);
            *tree = subtitute;
        }
    }

    avl_balance(tree);
    return 1;
}

// Retorna o numero de folhas da arvore
int avl_num_leaves(Avl *tree)
{
    if (tree == NULL)
        return 0;

    if (tree->l == NULL && tree->r == NULL)
        return 1;

    return avl_num_leaves(tree->r) + avl_num_leaves(tree->l);
}

#endif
```

Código 3: main.c

```c
#include <stdio.h>
#include "../include/avl.h"

int main(void)
{
    int data = 0;
    printf("Digite o primeiro valor para sua arvore: ");
    scanf("%d", &data);
    Avl *t = avl_create(data, NULL, NULL);

    while (1)
    {
        printf("\nArvore binaria de busca\n");
        printf("Representacao de parentese: ");
```

```c
        avl_print_parenthesis(t);
        printf("\nRepresentacao de barras:\n");
        avl_print_bars(t);

        int op = 0;

        printf("\nQue operacao voce quer fazer? (0 para sair, 1 para inserir novo no, 2 para remover um no, 3 para buscar um no, 4 para ver o numero de folhas da arvore): ");
        scanf("%d", &op);

        if (op == 0 || op > 4)
        {
            break;
        }

        if (op == 4)
        {
            int num_leaves = avl_num_leaves(t);
            printf("A arvore tem %d folha(s).\n", num_leaves);
            continue;
        }

        printf("\nDigite um valor: ");
        scanf("%d", &data);

        if (op == 1)
        {
            if (!avl_insert(&t, data))
            {
                printf("Erro ao inserir %d\n", data);
            }
        }
        if (op == 2)
        {
            if (!avl_remove(&t, data))
            {
                printf("Erro ao remover %d\n", data);
            }
        }
        if (op == 3)
        {
            Avl *node = avl_find(t, data);
            if (!node)
            {
                printf("No %d nao foi encontrado\n", data);
            }
            else
            {
                printf("No %d encontrado!\nSubarvore que tem no %d como raiz: ", data, data);
```

```
                avl_print_parenthesis(node);
                printf("\n");
            }
        }
    }

    avl_free(t);

    return 0;
}
```

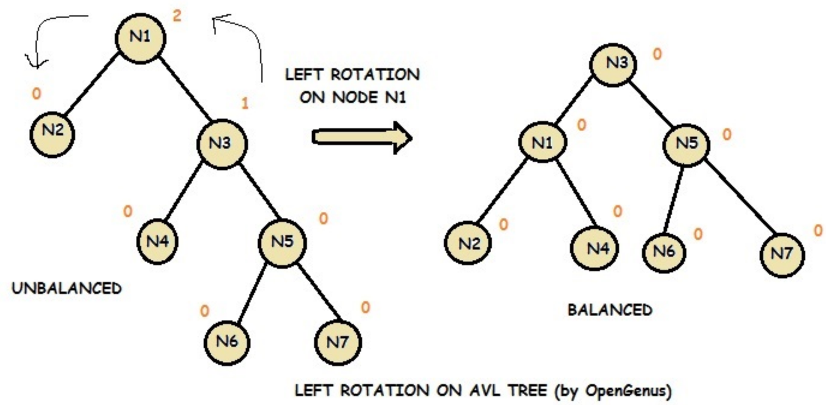
(d) Explique e dê exemplos sobre as operações aplicadas em árvores binárias de busca para que estas se tornem árvores AVL

Para que árvores binárias de busca se tornem AVL, elas precisam conter constante verificações de balanceamento em suas operações, verificando se onde ocorreram as modificações foi promovido algum desbalanceamento.

Caso seja identificado um desbalanceamento, o algoritmo deve entender onde esse foi gerado e efetuar movimentos denominados rotações para consertar o balanceamento da árvore. Só é necessário balancear o nó em que ocorreu o desbalanceamento, para que a árvore inteira volte a ser balanceada.

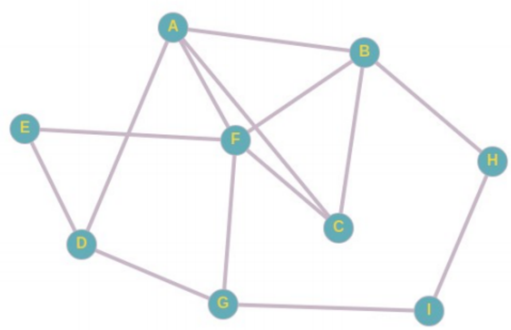
O movimento de rotação pode ser feito para a esquerda ou para a direita. Fazendo a rotação para a esquerda de um nó N1 que tem como filho a esquerda N3 e pai N0: N1 passa a ser filho a esquerda de N3, o filho a esquerda de N3 passa a ser filho a direita de N1, por fim, N0 deixa de ser pai de N1 e passa a ser pai de N3. A rotação para a direita é feita de forma análoga, mas na outra direção.

Para determinar qual movimento de rotação é realizado, é analisado a altura das suas subárvores filhas. Caso a altura da subárvore S1 filha a esquerda seja maior do que a direita, temos duas possibilidades: Caso a altura da subárvore filha a esquerda da raiz de S1 seja maior do que a direita, é realizada apenas uma rotação a direita; se a altura da subárvore filha a direita da raiz de S1 for a maior, é realizada uma rotação "dupla a direita", que consiste em rotacionar a esquerda a filha a esquerda e depois rotacionar a direita. Analogamente, caso a altura da subárvore S2 filha a direita seja maior do que a esquerda, também temos duas possibilidades: Caso a altura da subárvore filha a direita da raiz de S2 seja maior do que a esquerda, é realizada apenas uma rotação a esquerda; se a altura da subárvore filha a esquerda da raiz de S2 for a maior, é realizada uma rotação "dupla a esquerda", que consiste em rotacionar a direita a filha a direita e depois rotacionar a esquerda.



Questão 2.

Considere o Grafo não direcionado e sem pesos em que a lista de adjacências de cada nó é percorrida em ordem alfabética.



(a) Construa a matriz de adjacências para este grafo.

$$\begin{vmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{vmatrix}$$

(b) Começando do nó A, explique como o grafo é percorrido em profundidade.

Percorrer um grafo em profundidade é utilizar da pilha (LIFO) para visitar os nós para assim evitar que eles sejam visitados mais de uma vez. O algoritmo consiste em ir o mais "fundo"possível em cada ramo. É especialmente utilizado para verificar se o grafo possui ciclos, problemas do tipo labirinto com apenas uma solução e buscar se determinado nó possui ligação com outro.

Percorrendo o grafo em profundidade teremos: *A -> B -> C -> F -> E -> D -> G -> I -> H*

(c) Começando do nó A, explique como o grafo é percorrido em largura.

Percorrer um grafo em largura é utilizar da fila para percorrer "camadas". Assim, dado o nó inicial, o marcamos como visitado e adicionamos à fila. Enquanto a fila não estiver vazia, faremos a remoção de um nó da fila (FIFO) e a adição dos seus adjacentes não visitados à ela. Percorrendo em largura teremos: *A -> B -> C -> D -> F -> H -> E -> G -> I*

(d) Construa o pseudo-código para o algoritmo de Bellman-Ford e explique o seu funcionamento.

Algoritmo 4: Bellman-Ford

```
Entrada:
    Grafo direcionado G: Conjunto de vertices V e arestas A;
    Custo de cada aresta a ∈ A;
    Vertice ponto de partida s ∈ V;

Saida:
    Para cada vertice v ∈ V, o custo minimo associado a percorrer o grafo
       indo de s ate v;

Procedimento:

Para cada vertice v ∈ V, faca:
    Custo de v ← INFINITO;
    Vertice anterior a v ← NENHUM;

Custo de s ← 0;

Repita |V| − 1 vezes:
    Para cada aresta a ∈ A, faca:
        v ← Vertice origem da aresta a;
        u ← Vertice destino da aresta a;
        c ← Custo da aresta a;

        Se custo de u > Custo de v + c:
            Custo de u ← Custo de v + c;
            Vertice anterior a u ← v;

    Se nao houve melhoria:
        Interromper laco;
```

A base do algoritmo de Bellman-Ford é a relaxação de arestas do grafo. Esse procedimento, descrito nas linhas 19 a 25 do pseudo-código [4], busca o caminho mínimo até certo vértice u analisando apenas as arestas que levam diretamente a ele e os vértices de origem dessas arestas. Caso já sejam conhecidos os caminhos mínimos até todos os vértices cujas arestas levam a u, o caminho mínimo até u será aquele cuja a soma do custo da aresta que vai de v a u com o custo até v é menor. Contudo, mesmo que todos os caminhos mínimos até os vértices v que levam a u não sejam conhecidos, o procedimento ainda encontra o menor caminho possível de ser encontrado sem alterar os caminhos até os vértices v.

Partindo desse principio, a cada iteração, o algoritmo realiza a relaxação de arestas para todas as arestas do grafo. Como inicialmente já se é conhecido o custo do caminho mínimo até o ponto de partida s (custo 0, já que não há deslocamento), na primeira iteração são revelados caminhos até os vértices vizinhos a s. Então, gradualmente são descobertos os menores caminhos possíveis, com melhorias a cada iteração, até que os caminhos mínimos até todos os vértices sejam descobertos.

Em seu pior caso, o algoritmo passa por $|V|-1$ iterações, onde $|V|$ é a quantidade de vértices do grafo. Contudo, caso nenhuma melhoria de caminho seja encontrada numa iteração, o caminho mínimo até todos os vértices já foi descoberto, e o algoritmo pode ser interrompido.

(e) Construa o pseudo-código para o algoritmo de Dijkstra e explique o seu funcionamento.

Algoritmo 5: Dijkstra

```
Entrada:
    Grafo direcionado G: Conjunto de vertices V e arestas A;
    Custo de cada aresta a ∈ A;
    Vertice ponto de partida s ∈ V;
    Vertice destino t ∈ V;

Saida:
    O custo minimo associado a percorrer o grafo indo de s ate t;

Procedimento:

Para cada vertice v ∈ V, faca:
    Custo de v ← INFINITO;
    Vertice anterior a v ← NENHUM;
    Estado de v ← NAO VISITADO;

Custo de s ← 0;
Estado de s ← FRENTE DE AVANCO;

Enquanto houver vertices com estado igual a FRENTE DE AVANCO, faca:
    v ← Vertice com estado igual a FRENTE DE AVANCO de menor custo;

    Se v e igual a t:
        Interromper laco;

```

```
    Estado de v ← VISITADO;

    Para cada aresta a ∈ A que possui v como vertice de origem, faca:
        u ← Vertice destino da aresta a;
        c ← Custo da aresta a;

        Se estado de u e diferente de VISITADO:
            Estado de u ← FRENTE DE AVANCO;

            Se custo de u > custo de v + c:
                Custo de u ← Custo de v + c;
                Vertice anterior a u ← v;
```

O algoritmo de Dijkstra também se utiliza da relaxação de arestas, descrita na resposta para **(d)**. Contudo, ao invés de realizar a relaxação em todas as arestas por até $|V| - 1$ iterações, o algoritmo de Dijkstra utiliza um conjunto de vértices chamado "frente de avanço" para determinar a ordem das arestas a serem relaxadas.

A frente de avanço é iniciada contendo apenas o vértice de ponto de partida s, e o algoritmo chega ao seu fim quando não restarem vértices nesse conjunto. A cada iteração é encontrado o vértice v da frente de avanço que possui o menor custo de caminho, e então são relaxadas todas as arestas a que possuem v como vértice de origem e cujo vértice u de destino ainda não foi visitado, possivelmente revelando caminhos menores até os vértices u do que os conhecidos. Esses vértices u são então adicionados à frente de avanço e o vértice v é marcado como "visitado", para que arestas que o tem como destino não sejam mais relaxadas.

Ao seguir essa ordem, se garante que o caminho mínimo até o vértice v da frente de avanço com menor custo foi encontrado, já que os únicos caminhos até v inexplorados teriam que obrigatoriamente passar pelos outros vértices da frente de avanço e, como os outros vértices possuem custo maior do que o custo de v, seriam caminhos mais custosos. Consequentemente, se não houverem mais vértices na frente de avanço, o caminho mínimo até cada um dos vértices foi encontrado.

Caso um vértice destino t tenha sido determinado, é possível interromper o algoritmo no momento que for percebido que t é o vértice de menor custo da frente de avanço.