**CENTRO DE INFORMÁTICA**
**UNIVERSIDADE FEDERAL DA PARAÍBA**

# EFA3_ESTRUTURA_DE_DADOS

**Laura de Faria Maranhão Ayres - 20180019070**

**João Pessoa, 08 de dezembro de 2020**

## Questão 1

Na linguagem C, assim como em muitas outras linguagens de programação, uma função pode chamar a si própria, gerando um loop finito. Esse tipo de função é chamada função recursiva, a qual tem como objetivo quebrar um problema em subproblemas menores, até chegar a um problema pequeno o suficiente para que ele possa ser resolvido trivialmente.

Com isso, o potencial da recursividade está na possibilidade de definição dos elementos com base em suas versões mais simples. Assim, para implementar uma função recursiva, é necessária a criação de uma condição de parada e uma mudança de estado a cada chamada (precisa haver alguma diferença entre o estado inicial e o próximo estado da função).

A partir disso, percebe-se que apesar de algumas versões recursivas consumirem um maior número de recursos (principalmente memória e processamento), a recursividade ainda pode ser considerada positiva, sendo utilizada na obtenção de códigos mais “enxutos” e mais fáceis de compreender/implementar.

Apesar de certas vantagens, tornam-se necessários alguns cuidados ao manipular funções recursivas. Com isso, não é recomendado utilizá-las quando: a recursão é de cauda, parâmetros consideravelmente grandes têm que ser passados por valor, não é possível prever o número de chamadas ou existe uma única chamada do procedimento/função recursiva no fim ou no começo da rotina.

Por fim, é aconselhada a implementação das funções da maneira mais geral possível, tornando-as fáceis de serem reutilizadas e entendidas. Além disso, sugere-se evitar funções que utilizem variáveis globais. Nesse contexto, se houver uma rotina que deve ser o mais veloz possível, seria bom implementá-la sem nenhuma (ou com o mínimo de) chamadas a funções, porque uma chamada a uma função consome tempo e memória.

# Questão 2

### a-) Resumo

Árvores são estruturas de dados não-lineares utilizadas para representar hierarquias, a qual pode ser trabalhada usando a recursividade. Seu funcionamento se dá através de nós, uma vez que existe um nó raiz '*r*' que pode ou não conter subárvores, cujas (sub)raízes são ligadas diretamente a *r*. Esses nós raízes das subárvores são ditos filhos do nó pai, *r*. Assim, nós com filhos são comumente chamados de nós internos e nós que não têm filhos são chamados de folhas ou nós externos.

Nesse contexto, as árvores possuem outras características próprias, sendo uma delas a altura. Essa propriedade é identificável através do caminho da raiz até a folha mais distante. Pode-se também numerar os níveis em que os nós aparecem na árvore, no qual a raiz está no nível 0, seus filhos diretos no nível 1 e assim por diante, até chegar no último nível (será a altura *h*).

Uma árvore é dita cheia se todos os seus nós folhas estão no mesmo nível e não se pode acrescentar nenhuma nova folha sem aumentar a altura da árvore. Além disso, é dita degenerada se todos os seus nós internos têm uma única subárvore associada. Também podemos notar que o número de nós de determinado nível de uma árvore cheia é uma unidade a mais do que a soma de todos os nós dos níveis anteriores, ou seja, $2h+1 - 1$ nós.

$$N = \sum_{i}^{n} 2^i = 2^{n+1} - 1$$

Uma árvore pode ser classificada como *vazia* (ponteiro para a raiz nulo) ou com um nó raiz tendo duas subárvores, identificadas como a subárvore da direita (*sad*) e a subárvore da esquerda (*sae*). Assim, cada nó deve armazenar três informações: a informação propriamente dita, no caso, um caractere, e dois ponteiros para as subárvores, à esquerda e à direita.

Com isso, informações armazenadas e quantidade de filhos por nó (grau) dependem do tipo de árvore trabalhada. Os tipos trabalhados são as que cada nó possui no máximo dois filhos (Binárias) e outras cujo número de filhos é variável.

A partir da altura, pode-se avaliar a eficiência com que se visita os nós dessa árvore, uma vez que a altura indica o esforço computacional necessário para alcançar qualquer nó da árvore a partir da raiz. Uma árvore binária com *n* nós tem uma altura mínima proporcional a log2 *n* (caso da árvore cheia) e uma altura máxima proporcional a *n* (caso da árvore degenerada).

Muitas operações em árvores binárias envolvem o percurso de todas as subárvores, executando alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

- Pré-ordem: trata raiz, percorre sae, percorre sad;
- Ordem simétrica: percorre sae, trata raiz, percorre sad;
- Pós-ordem: percorre sae, percorre sad, trata raiz.

Desse modo, essa estrutura de dados possui algumas operações com utilidade geral. Sendo algumas delas:

- Criar um novo nó
- Inserir um nó
- Remover nó
- Consultar os nós da árvore em ordem
- Consultar em pré-ordem
- Consultar em pós-ordem

Como vimos, o algoritmo de busca binária tem bom desempenho computacional e deve ser usado quando temos os dados ordenados em um vetor, exceto quando precisa-se inserir e remover elementos da estrutura e, ao mesmo tempo, dar suporte a funções de busca eficientes. As árvores binárias de busca possuem elementos menores que a raiz armazenados na subárvore da esquerda(*sae*) e elementos maiores que a raiz na subárvore da direita(*sad*). Essa propriedade garante que, quando a árvore é percorrida em ordem simétrica (*sae*-raiz-*sad*), os valores são encontrados em ordem crescente.

Para procurar um valor numa árvore, compara-se o valor procurado com o associado à raiz. Em caso de igualdade, o valor foi encontrado; se o valor dado for menor que o valor associado à raiz, a busca continua na *sae*; caso contrário, se o valor associado à raiz for menor, a busca continua na *sad*.

**b-)** **OBS: Coloquei esses 3 primeiros exercícios no mesmo arquivo, compartilhando a main entre eles.**

## Exercício 1

```c
int pares(Arv* a)
{
    return busca_par(a->raiz);     //Procura os numeros pares começando pela raiz
}

int busca_par(ArvNo* a)
{
    static int cont=0;          //Contador de números pares
    if(a == NULL){
        return 0;
    }

    busca_par(a->esq);          //Busca por profundidade (inicia pela esquerda)
    busca_par(a->dir);
    if(a->info%2 == 0){         //Caso o nó seja par, contabiliza
        cont++;
    }

    return cont;                //Retorna a quantidade
}
```

## Exercício 2

```c
int folhas(Arv* a)
{
    return busca_folhas(a->raiz);     //Procura as folhas da árvore começando pela raiz
}

int busca_folhas(ArvNo* a)
{
    static int cont=0;          //Contador de folhas
    if(a == NULL){
        return 0;
    }

    busca_folhas(a->esq);          //Busca por profundidade (inicia pela esquerda)
    busca_folhas(a->dir);
    if(a->esq == NULL && a->dir == NULL){   //Caso o nó não possua ramificação, contabiliza
        cont++;
    }

    return cont;                //Retorna a quantidade
}
```

## Exercício 3

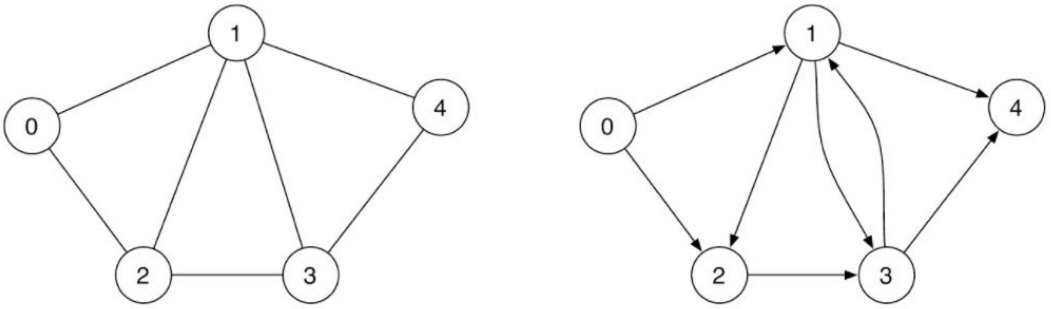
```c
int filhos(Arv* a)
{
    return busca_filhos(a->raiz);     //Procura os nós da árvore começando pela raiz
}

int busca_filhos(ArvNo* a)
{
    static int cont=0;          //Contador de nós com 1 filho
    if(a == NULL){
        return 0;
    }

    busca_filhos(a->esq);          //Busca por profundidade (inicia pela esquerda)
    busca_filhos(a->dir);

    /*Caso o nó possua apenas uma ramificação, contabiliza*/
    if(a->esq == NULL){
        if(a->dir != NULL){
            cont++;
        }
    }
    if(a->dir == NULL){
        if(a->esq != NULL){
            cont++;
        }
    }

    return cont;                //Retorna a quantidade
}
```

# Questão 3

### a-) Resumo

Grafos são estruturas de dados que possuem conectividade entre nós, semelhante a árvores. Assim, na construção do grafo, cada nó representa um vértice (*V*) e cada conexão reproduz uma aresta (*E*). Essa estrutura possui diversas utilidades, sendo para representar algo (rede de computadores, um mapa viário ou uma rede de fluxo) ou solucionar um problema (achar um caminho mínimo, ordem de execução de tarefas interdependentes ou o fluxo máximo de uma rede).

Com isso, um grafo pode ser caracterizado como não orientado, o qual cada aresta representa uma conexão entre os dois vértices ligados, nos dois sentidos, diferentemente do grafo orientado, que possui cada aresta com apenas uma orientação.




A partir disso, para a representação de grafos faz-se uso com frequência de duas estruturas: lista de adjacência e matriz de adjacência. A primeira é representada através de um vetor de vértices, o qual possui uma lista de arestas que partem desse vértice (origem). Enquanto isso, em uma matriz de adjacência há uma representação de matriz V x V, em que cada elemento representa a aresta (ou a ausência dela) que estaria conectando dois vértices.

Além das mudanças estruturais, há uma diferença com relação ao espaço de memória, uma vez que o espaço de memória requerido para representar um grafo por lista de adjacência é O(V + E), enquanto o espaço necessário para representar por matriz de adjacência é O(V 2). Desse modo, a lista de adjacência é preferível para grafos esparsos (possuem um número pequeno de arestas), enquanto a por matriz de adjacência é recomendada para grafos densos.

Para visitar os vértices de um grafo partindo da origem são feitas estratégias como: Busca em profundidade ou busca em amplitude. Na busca em profundidade, parte-se do de um vértice para os vértices adjacentes, percorrendo os seus descendentes e retrocedendo quando encontra-se um vértice já explorado, criando uma recursividade. Para evitar que um vértice seja explorado mais de uma vez, deve-se marcar os vértices já visitados, sendo comum associá-los a cores: branco (não explorados), cinza (os filhos daquele vértice ainda estão sendo explorado) e preto (já explorados completamente).

Após uma busca em profundidade, podemos classificar as arestas do grafo em três tipos: *arestas de árvore*, *arestas para trás* e *outras arestas*. As arestas de árvore representam as arestas que levam à exploração de um novo vértice do grafo (levam a um vértice de cor *branca*), enquanto as arestas para trás apontam para vértices já visitados no mesmo ramo da árvore (são as arestas que levam a um vértice de cor *cinza*). Por fim, as arestas que levam aos vértices de cor *preta* são as *outras arestas*: podem ser *arestas para frente* ou *arestas que cruzam*. Assim, além de classificar as arestas durante a busca, pode-se também registrar o “tempo” de início e fim de exploração de cada vértice.

No caso da busca em amplitude ou em largura, exploram-se todos os vértices adjacentes a um dado vértice antes de prosseguir com a exploração dos descendentes deles, diminuindo a diferença entre o tempo inicial e final de exploração de um vértice. Enquanto a busca em profundidade é implementada de forma recursiva, usando a pilha de execução para armazenar os vértices em exploração, esse método deve utilizar uma fila, adicionando nela cada vértice descoberto, prosseguindo enquanto existirem vértices na fila. Nisso, se o vértice é retirado da fila, seus adjacentes são colocados para serem explorados.

Quanto à manipulação das informações em um grafo, podemos inteirar através de diversos algoritmos, com diversas complexidades e o jeito de resolver determinada situação. Dentre os problemas mais triviais, temos o de encontrar o caminho mínimo de um vértice a outro qualquer, contido no grafo. Geralmente, se baseiam na lógica de que: havendo uma aresta que liga dois vértices("a" e "b") qualquer, se a soma do custo do caminho que chegue até "a" com o custo da aresta que conecta "a" e "b" for menor do que o caminho mínimo já alcançado, então este passa a ser o novo caminho mínimo.

Três algoritmos abordados no livro e que lidam com essa problemática de caminho mínimo são os algoritmos de Bellman-Ford, Dijkstra e A estrela(A*). O primeiro consiste em tornar o peso da origem igual a zero, já as demais arestas como infinito. Após isso, deve-se aplicar o relaxamento em todas as arestas, enquanto for possível. Desse modo, cada aresta possuirá o valor da soma do seu peso com o caminho até ele, até então, quando for possível que este valor seja inferior ao valor que ele guardava (outro caminho). Assim, guardam-se sempre o valor dos menores caminhos.

O algoritmo de Dijkstra funciona de maneira semelhante ao de Bellman-Ford, porém enquanto este realiza um relaxamento de forma a se aprofundar no grafo (relaxando os próximos vértices ao vértice recém relaxado), o Dijkstra relaxa o grafo em "largura". Com isso, relaxa-se por níveis do grafo, após relaxar todos de um mesmo nível, passa-se ao próximo nível.

Em contrapartida, o A* une as correções provindas do algoritmo de Dijkstra com a robustez de uma heurística que estima a distância até o destino. É uma melhoria significativa em relação ao Dijkstra e bastante utilizado para buscar caminho mínimo em mapas, sendo bastante utilizado em jogos e na robótica, em situações de labirintos. Seu funcionamento consiste em buscar sempre pelos vértices do caminho mínimo, a fim de avançar sempre em direção ao destino, considerando todos os possíveis caminhos e custos envolvidos para o deslocamento.

Por fim, destaca-se o problema de determinação do fluxo máximo em uma rede. Nesse contexto, o grafo atual representa uma rede de fluxo, o qual cada aresta tem associada uma capacidade máxima de fluxo. Isso tem-se por meta obter um algoritmo que informe o fluxo máximo possível da rede do vértice origem ($s$) ao vértice destino ($t$). Assim, em uma solução válida, o fluxo em cada aresta deve ser no máximo igual à sua capacidade. Além disso, o fluxo de entrada em um vértice deve ser igual ao fluxo saindo, para qualquer vértice interno (com exceção da origem e do destino), mantendo sua conservação.

**b-) Matriz de adjacência**

| | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* |
|---|---|---|---|---|---|---|---|---|---|
| ***A*** | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| ***B*** | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| ***C*** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| ***D*** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| ***E*** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| ***F*** | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| ***G*** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| ***H*** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| ***I*** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

**c-) Percorrido em profundidade**

A partir do vértice A possui-se 4 opções: B, C, F ou D (vértices adjacentes), os quais passarão para seus vértices adjacentes, como por exemplo: a próxima rota seguindo por B pode ser através de F, C ou H. Assim, quando chegar em alguma situação em que o próximo vértice da sequência já foi explorado, há um retorno para o vértice anterior. Isso pode ser representado na situação em que seguiria A-B-F-G-I-H, na qual retorna para o ponto G (último vértice que possui mais de um filho) e parte para o ponto D.

Dessa forma, a forma mais eficiente em tempo e custo (assumindo que todas as arestas possuem o mesmo custo) seria através de rotas como:

**A - B - H - I - G - D - E - F - C**

**A - C - F - E - D - G - I - H - B**

Seguindo essa ideia, as rotas mais custosas seriam aquelas que cruzam com um vértice já explorado, precisando retornar para a última bifurcação, como por exemplo:

**A - B - H - I - G - F - C - E - D**

**A - F - G - I - H - B - C - D - E**

**d-) Percorrido em largura**

Diferentemente do processo passado, parte-se do vértice A e visita-se as 4 opções: B, C, F ou D (vértices adjacentes). Nisso, após analisar todos os vértices adjacentes de A, prossegue-se para o primeiro visitado, repetindo o mesmo processo. Assim, tem-se como percurso total: