Técnicas de Busca e Ordenação
Roteiro de Laboratório 3 – Recursão e Árvores

1 Introdução

O conceito de *recursão* é fundamental na matemática e na ciência da computação. Uma definição simples de recursão (para ambas as áreas) usa o conceito de função recursiva, isto é, uma função que chama ela mesma (ou é definida em termos dela mesma). Também é necessário um *caso base* ou *condição de terminação* para que a recursão termine. Apesar de recursão ser um conceito simples, todo o paradigma de programação funcional se baseia nele. De fato, conforme mostrado por Alonso Church na década de 1930 com o formalismo de λ-*calculus*, qualquer computação pode ser resumida a definições de funções e suas aplicações (possivelmente recursivas).

O estudo de recursão aqui é interlaçado com o estudo de estruturas recursivamente definidas conhecidas como *árvores*. Vamos usar árvores tanto como estruturas de dados explícitas, bem como construções auxiliares para a compreensão e análise de programas recursivos. Recursão nos ajuda a desenvolver estruturas de dados e algoritmos elegantes e eficientes para inúmeras aplicações.

O objetivo deste laboratório é relembrar os conceitos de árvores e de recursão vistos na disciplina de Estrutura de Dados I, expandindo o uso desses conceitos para algoritmos não-recursivos.

2 *Binary Search Tree (BST)*

Uma **árvore binária** (*binary tree*) é formada por um *nó externo (folha)* ou por um *nó interno* conectado a um par de árvores binárias, chamadas de sub-árvores da esquerda e da direita do nó, respectivamente. Uma **árvore binária de busca** (*binary search tree – BST*) é uma árvore binária aonde cada nó possui uma *chave*, e satisfaz a seguinte propriedade adicional: para qualquer nó n da árvore, a chave de n é (i) maior do que as chaves de todos os nós da sub-árvore da esquerda de n; e (ii) menor do que as chaves de todos os nós da sub-árvore da direita de n. Por essa definição é imediato notar que uma BST não admite chaves repetidas.

Exercício 1 – TAD BST. Crie um TAD para uma BST assumindo que as chaves são valores inteiros (tipo `int`). Note que foi pedido um TAD, o que implica que o nó da BST deve ser uma estrutura opaca (isto é, com a implementação "escondida" no `.c`). A seguir, implemente as seguintes operações:

1. Criação de uma BST vazia.
2. Inserção de uma chave na BST.
3. Destruição (liberação da memória) de uma BST.

As operações 2 e 3 podem ser implementadas com funções recursivas ou não, como você preferir. A inserção de uma chave que já existe na BST termina silenciosamente sem modificar a estrutura.

Exercício 2 – Cliente do TAD BST. Uma BST é uma estrutura não balanceada. Uma sequência de inserção de chaves infeliz (ordenada) leva a uma BST *degenerada*, isto é, uma árvore aonde todas as sub-árvores da esquerda (ou da direita) são vazias. Neste caso a árvore "degenera" para uma lista encadeada.

Uma forma de se tentar evitar a construção de uma BST desbalanceada é causar alguma "pertubação" na sequência de inserções na tentativa de evitar que todas as chaves fiquem de um único lado. Uma "pertubação" normalmente é gerada por algum efeito aleatório, seja através de trocas aleatórias das posições de chaves na sequência de inserção, seja através da simples geração de chaves aleatórias.

Implemente um programa cliente para a implementação da BST que:

- Recebe como argumento um número N de chaves a serem geradas.
- Cria uma BST vazia.
- Gera N chaves (`int`s) aleatórias e insere na BST.
- Determina a altura final da BST criada e exibe essa informação em `stdout`. (Veja, por exemplo, os exercícios de caminhamento em árvore a seguir.) Lembrando que a altura de uma árvore é o comprimento do caminho mais longo da raiz até alguma folha. A altura de uma árvore com um único nó raiz é *zero* e de uma árvore vazia é -1.

Para esses exercícios é suficiente utilizarmos números pseudo-aleatórios. A forma mais simples de fazer isso em C é com o código abaixo.

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

srand(time(NULL));   // should only be called once
int r = rand();      // returns a pseudo-random integer between 0 and RAND_MAX
```

Veja `https://en.wikipedia.org/wiki/Pseudorandomness` para mais detalhes sobre o surgimento e uso de geradores de números pseudo-aleatórios.

Concluído o seu programa cliente, realize o seguinte experimento: para $N = 10^6$, execute o cliente 10 vezes e anote a altura da BST para cada execução. Calcule a média e desvio padrão da altura. Perguntas:

1. Qual conclusão você pode obter pelo desvio padrão?
2. Qual é a altura máxima e mínima de uma BST em função de N?
3. Considerando os limites da questão anterior, a média experimental da altura da BST é boa ou ruim? Por quê?

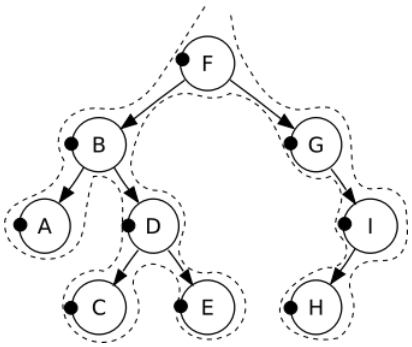
3 Caminhamento em árvore (*tree traversal*)

Um caminhamento em profundidade em uma árvore binária pode ser realizado de três formas distintas. Em todos os casos, todos os nós da árvore são *visitados*, representando uma abstração da operação que se quer realizar sobre cada nó durante o caminhamento. Um exemplo trivial de uma operação de visitação é exibir o conteúdo do nó. Os três caminhamentos em profundidade diferem somente do momento quando um nó é visitado. Todas as três formas podem ser facilmente descritas através de algoritmos recursivos.

Recursive pre-order traversal. Dado um nó t, o caminhamento em pré-ordem de t:

1. Visita t.
2. Caminha recursivamente em pré-ordem na sub-árvore da esquerda.
3. Caminha recursivamente em pré-ordem na sub-árvore da direita.

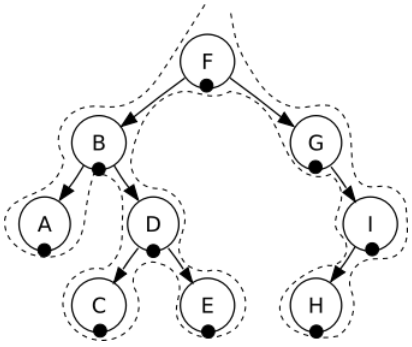
A figura ao lado exibe um caminhamento em pré-ordem em uma BST. Os nós são visitados na ordem: F, B, A, D, C, E, G, I, H.



Recursive in-order traversal. Dado um nó t, o caminhamento em ordem de t:

1. Caminha recursivamente em ordem na sub-árvore da esquerda.
2. Visita t.
3. Caminha recursivamente em ordem na sub-árvore da direita.

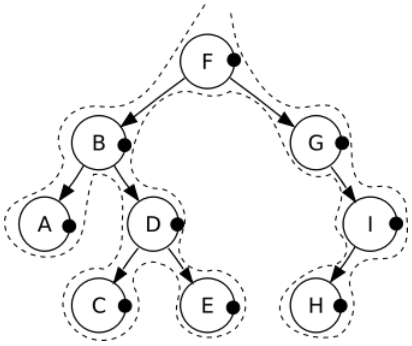
A figura ao lado exibe um caminhamento em ordem em uma BST. Os nós são visitados na ordem: A, B, C, D, E, F, G, H, I. Como o nome já diz, um caminhamento em ordem em uma BST recupera as chaves de forma ordenada.



Recursive post-order traversal. Dado um nó t, o caminhamento em pós-ordem de t:

1. Caminha recursivamente em pós-ordem na sub-árvore da esquerda.
2. Caminha recursivamente em pós-ordem na sub-árvore da direita.
3. Visita t.

A figura ao lado exibe um caminhamento em pós-ordem em uma BST. Os nós são visitados na ordem: A, C, E, D, B, H, I, G, F.



Exercício 3 – *Recursive BST traversal.* Implemente as três formas de caminhamento em profundidade descritas acima através de funções recursivas. Faça funções genéricas, isto é, a função de visitação deve ser passada como parâmetro para as funções de caminhamento. Um exemplo da cabeçalho da função de caminhamento em pré-ordem:

```
void rec_preorder(BST *t, void (*visit)(BST*));
```

4 Caminhamento não-recursivo

É importante considerarmos também implementações não-recursivas dos algoritmos de caminhamento. As três variantes dos caminhamentos em profundidade podem ser implementadas utilizando uma pilha de nós da BST.

Exercício 4 – TAD Pilha. Implemente um TAD pilha que permite o empilhamento (`push`) e remoção (`pop`) de nós da BST. A sua implementação pode ser bastante simples. É aceitável uma implementação com um limite máximo para a pilha. O foco aqui é o uso das operações de pilha no caminhamento, não uma implementação avançada dessa estrutura.

Exercício 5 – *Non-recursive BST traversal.* Utilizando a pilha desenvolvida no exercício anterior, implemente versões não-recursivas dos três algoritmos de caminhamento em profundidade: *pre-order*, *in-order* e *post-order*. Comece com o *pre-order* que é mais simples. Os outros dois são um pouco mais elaborados, mas não muito. Se estiver em dúvida, veja um exemplo de pseudo-código em `https://en.wikipedia.org/wiki/Tree_traversal`.

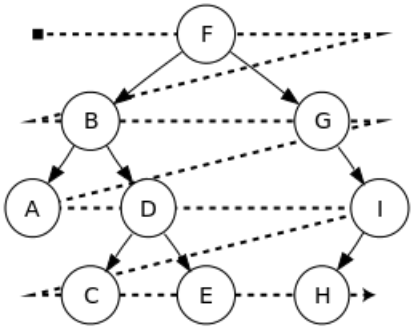
Exercício 6 – Análise empírica. Modifique o seu programa cliente para comparar o tempo de execução das variantes recursivas e não-recursivas dos algoritmos de caminhamento. Utilize a biblioteca `time.h` para medir o tempo de cada função. (Veja um exemplo no código abaixo .) Considere $N = 10^7$. Qual versão ficou mais rápida? Qual conclusão você consegue tirar dessa análise?

```c
#include<stdio.h>
#include<time.h>

int main() {
    clock_t start = clock();
    /* do something here... */
    clock_t end = clock();

    double seconds = ((double) end - start) / CLOCKS_PER_SEC;
    printf("%lf\n", seconds);
    return 0;
}
```

5 *Level-order traversal*

Árvores também podem ser exploradas em níveis (*level-order*), aonde todos os nós de um mesmo nível são visitados antes de se passar para o nível seguinte. Esse método também é chamado de caminhamento em *largura*.

A figura ao lado exibe um caminhamento em níveis em uma BST. Os nós são visitados na ordem: F, B, G, A, D, I, C, E, H.



Curiosamente, é razoavelmente difícil construir um algoritmo recursivo de caminhamento por níveis sem usar uma estrutura auxiliar. No caso de um algoritmo iterativo, basta utilizar uma fila simples de nós.

Exercício 7 – TAD Fila. Implemente um TAD fila que permite o enfileiramento (`enqueue`) e remoção (`dequeue`) de nós da BST. Valem os mesmos comentários do exercício 4.

Exercício 8 – *Non-recursive level-order traversal.* Utilizando a fila desenvolvida no exercício anterior, implemente o algoritmo não-recursivo de caminhamento em níveis.