Segundo teste de Algoritmos e Estruturas de Dados

18 de Novembro de 2019 **14h10m – 15h00m**

Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.
O teste é composto por 5 grupos de perguntas.

Nome: _____

N. Mec.: _____

4.0 **1:** Pretende-se que a seguinte função implemente uma pesquisa binária. Complete-a (isto é, preencha as caixas).

```
int binary_search(int n,int a[n],int v)
{
  int low = [ 0 ];
  int high = [ n - 1 ];
  while(high [ >= ] low)
  {
    int middle = [ (low + high) /2 ];
    if(a[middle] == v)
      return middle;
    if(a[middle] [ > ] v)
      [ high ] = middle - 1;
    else
      [ low ] = middle + 1;
  }
  return [ -1 ];
}
```

? → depende do que forem os args de entrada

Indique (não é preciso justificar) qual é a complexidade computacional desta função.

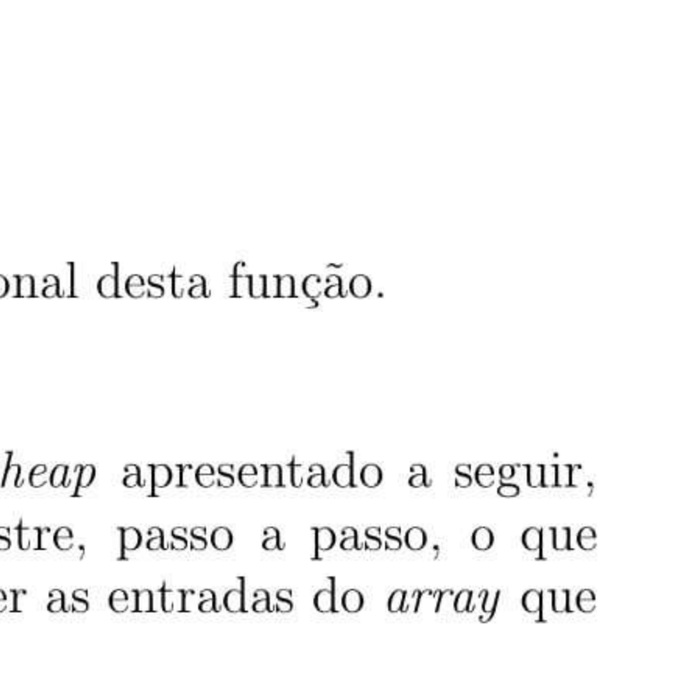
Resposta: O(log(n))

4.0 **2:** Explique como está organizado um *max-heap*. Para o *max-heap* apresentado a seguir, insira o número 8. Não apresente apenas o resultado final; mostre, passo a passo, o que acontece ao *array* durante a inserção. Em cada linha, basta escrever as entradas do *array* que foram alteradas.

Respostas: Um max-heap é uma árvore binária completa tal que o valor de cada nó interno é maior ou igual a qualquer um dos valores dos nós que descendem de si mesmo.

9	6	7	3	1	4	5	2	8
9	6	7	8	1	4	5	2	3
9	8	7	6	1	4	5	2	3
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

final binary tree:



Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.
O teste é composto por 5 grupos de perguntas.

3.0 **3:** Pretende-se implementar uma fila (*queue*) usando um *array* circular. O código seguinte define a estrutura de dados a usar (para simplificar, vamos usar variáveis globais).

```
#define array_size  1024

int array[array_size];
int read_pos = 1;  // incremented after reading
int write_pos = 0; // incremented before writing
int count = 0;     // equal to (read_pos - write_pos - 1) % array_size
```

(rear) (front)

A expressão

read_pos - write_pos -1

dá-nos o número de casas livres no array entre as posição para as quais read_pos e write_pos apontam (em que read_pos é o rear e write_pos é a front). O resto da divisão deste resultado por array_size permite-nos detetar situações de *undeflow* (em caso de estarmos a retirar elementos do *array*) ou *overflow* (em caso de estarmos a colocar elementos no *array*)

Responda às seguintes perguntas:

1.5 **a)** Por que é que neste caso é vantajoso usar um *array* circular?

Resposta:

Uma *queue* é uma estrutura de dados que segue uma ordem particular para executar instruções. A ordem é FIFO (*First In First Out*), o que significa que, se quisermos retirar um elemento da *queue*, temos sempre que retirar o primeiro que lá foi colocado (ou seja, o elemento que já foi colocado há mais tempo, de entre os disponíveis). O ponteiro read_pos vai apontar sempre para o fundo da *queue* (ou seja, para o primeiro elemento colocado). O ponteiro write_pos aponta sempre para o topo da *queue*, ou seja, para o último elemento que foi colocado. Na operação de remoção de um elemento com um *array* circular, podemos simplesmente aceder ao índice desse elemento com o ponteiro de leitura e incrementar esse mesmo ponteiro, efetivamente removendo o elemento em questão; utilizando, por exemplo, uma lista ligada, teríamos que percorrer a lista inteira de cada vez que quiséssemos efetuar uma remoção só para conseguir aceder ao elemento a retirar, que seria sempre o último. Um *array* também poupa tempo aquando da alocação de memória.

1.5 **b)** Use algumas das seguintes linhas de código para implementar a função `enqueue` (que coloca um item de informação na fila). Risque as linhas que estão a mais.

```
int enqueue(int v)
void enqueue(int v)
{
  if(count == 0) exit(1); // underflow
  if(count == array_size) exit(1); // overflow
  array[write_pos] = v;
  v = array[write_pos];
  write_pos = (write_pos + 1 == array_size) ? 0 : write_pos + 1;
  write_pos = (write_pos > 0) ? write_pos - 1 : array_size - 1;
  array[write_pos] = v;
  v = array[write_pos];
  count--;
  count++;
  return v;
}
```

a função *enqueue* escreve no *array* circular representativo da *queue*, logo não tem valor de retorno; a função que retorna v é a função *int dequeue(int v)*, que retira um elemento da *queue* e, portanto, tem que o ler e retornar (*)

na operação *enqueue* (colocar um elemento na *queue*), nunca vai ocorrer *underflow*, tal como na operação *dequeue* (retirar um elemento da *queue*) nunca vai ocorrer *overflow* (NOTA: *underflow* ocorre quando tentamos retirar um elemento de uma *queue* que já está vazia; *overflow* ocorre quando tentamos inserir um elemento numa *queue* que já está cheia)

(1)

(2)

(3)

(4)

(5)

(6)

não precisamos de incrementar ou decrementar count porque esta variável é calculada segundo a fórmula count = (read_pos - write_pos - 1) % array_size

(*)

(1) nesta linha, estaríamos a tentar inserir o elemento na *queue* antes de incrementar o ponteiro de escrita; como é indicado por um dos comentários no código da alínea anterior, o ponteiro é incrementado antes de se escrever no *array*
(2) não precisamos de guardar o que está no espaço de memória para o qual o ponteiro está a apontar (isso é na função *dequeue*, em que retornamos esse mesmo valor)
(3) incrementação do ponteiro de escrita de acordo com o funcionamento de um *array* circular
(4) o ponteiro de escrita é incrementado, não decrementado (queremos escrever no fundo da *queue* e ir escrevendo para cima, e não ao contrário; o fundo da *queue* corresponde ao início do *array* circular e o topo da *queue* corresponde ao fim do *array* circular)
(5) inserção do novo elemento
(6) ver (2); para além disso, mesmo que estivéssemos na função *dequeue*, esta instrução não faria sentido aqui porque o ponteiro de leitura é incrementado depois da leitura ser efetuada, ou seja, o valor de *array[pointer]* teria que ser recolhido antes da incrementação do ponteiro

Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.
O teste é composto por 5 grupos de perguntas.

5.0 **4:** Um programador pretende utilizar uma *hash table* (tabela de dispersão, dicionário) para contar o número de ocorrências de palavras num ficheiro de texto. O programador está à espera que o ficheiro tenha cerca de 6000 palavras distintas, pelo que usou uma *hash table* do tipo *separate chaining* com 10007 entradas, e usou a seguinte *hash function*:

```
unsigned int hash_function(unsigned char *s,unsigned int hash_table_size)
{
  unsigned int sum;
  for(sum = 0;*s != '\0';s++)
    sum += (unsigned int)(*s);
  return sum % hash_table_size;
}
```

Infelizmente, as expetativas do programador estavam erradas, e o ficheiro de texto era muito maior que o esperado, tendo cerca de 1000000 palavras distintas. Responda às seguintes perguntas:

2.0 **a)** A *hash function* apresentada acima é muito má. Porquê? Sugira uma outra que seja bem melhor.

3.0 **b)** Uma implementação do tipo *separate chaining* usa habitualmente uma lista ligada para armazenar todas as chaves (neste caso, as palavras) para as quais a *hash function* tem o mesmo valor. Que vantajens/desvantagens teria uma implementação que usa uma árvore binária ordenada em vez da lista ligada? E se for uma árvore binária ordenada e balanceada?

Respostas:

(a) A *hash function* apresentada é muito má porque utiliza uma simples cifra de substituição, ou seja, não tem em conta a posição de cada caracter na *string* - apenas o seu valor. Isto significa que palavras com a mesma combinação de letras em posições diferentes (anagramas) vão ter o mesmo *hash code*. Para além disso, é possível adicionar um valor aleatório a uma letra (por exemplo, A+1=B) e subtrair o mesmo valor noutra letra (por exemplo, N-1=M) e, mudando a ordem dos caracteres, criar novas *strings*, todas com o mesmo *hash code*. Fazendo isto para todas as letras de uma palavra, podemos criar uma palavra com um conjunto de letras inteiramente diferente que, mesmo assim, tem o mesmo *hash code* da palavra inicial. Com este mesmo truque, podemos mesmo variar o comprimento das novas palavras geradas e obter os mesmos *hash codes*. Uma maneira simples de resolver este problema seria modificar a linha dentro do ciclo *for* para:

sum += (157u * sum) + (unsigned int)(*s) ,

sendo o fator multiplicativo 157 escolhido quase arbitrariamente (é um número primo).

(b) Uma árvore binária ordenada é muito mais rápida de pesquisar que uma lista ligada porque a inserção de dados na mesma garante a sua ordenação (obviamente), enquanto que a inserção de dados numa lista ligada é sequencial, basicamente impossibilitando pesquisa binária, etc. A desvantagem da árvore binária é que, no pior caso (por exemplo, quando inserimos informação já ordenada), a sua eficiência é igual à de uma lista ligada. Para além disso, uma árvore binária requer codificação consideravelmente mais complexa que uma lista ligada. Listas ligadas são mais simples de implementar. Se a árvore binária for ordenada e balanceada, isto resolve o problema do pior caso (em termos de eficiência) da árvore binária ordenada (mas não balanceada). Contudo, a árvore balanceada agrava o problema da complexidade de codificação.

4.0 **5:** Apresentam-se a seguir várias funções que visitam todos os nós de uma árvore binária, e mostram-se várias ordens pelas quais a função `visit` foi chamada para cada um dos nós (1 significa que o nó correspondente foi o primeiro a chamar a função `visit`, 2 que foi o segundo, e assim por diante). Para cada uma das ordens apresentadas, indique que função, ou funções, deram origem a essa ordem.

```
void f1(tree_node *link)
{
  queue *q = new_queue();
  enqueue(q,link);
  while(is_empty(q) == 0)
  {
    link = dequeue(q);
    if(link != NULL)
    {
      visit(link);
      enqueue(q,link->left);
      enqueue(q,link->right);
    }
  }
  free_queue(q);
}
```

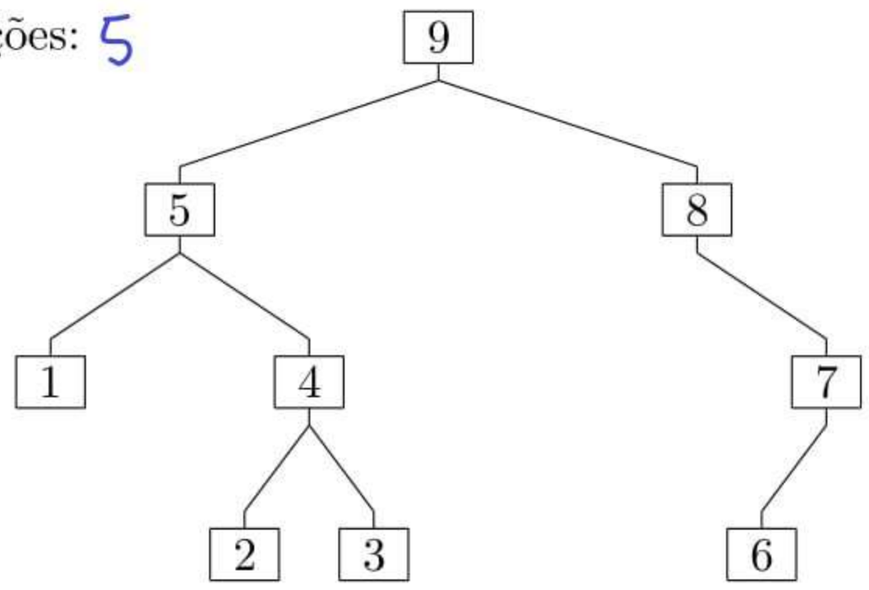
```
void f2(tree_node *link)
{
  stack *s = new_stack();
  push(s,link);
  while(is_empty(s) == 0)
  {
    link = pop(s);
    if(link != NULL)
    {
      visit(link);
      push(s,link->right);
      push(s,link->left);
    }
  }
  free_stack(s);
}
```

```
void f3(tree_node *link)
{
  if(link != NULL)
  {
    visit(link);
    f3(link->left);
    f3(link->right);
  }
}
```

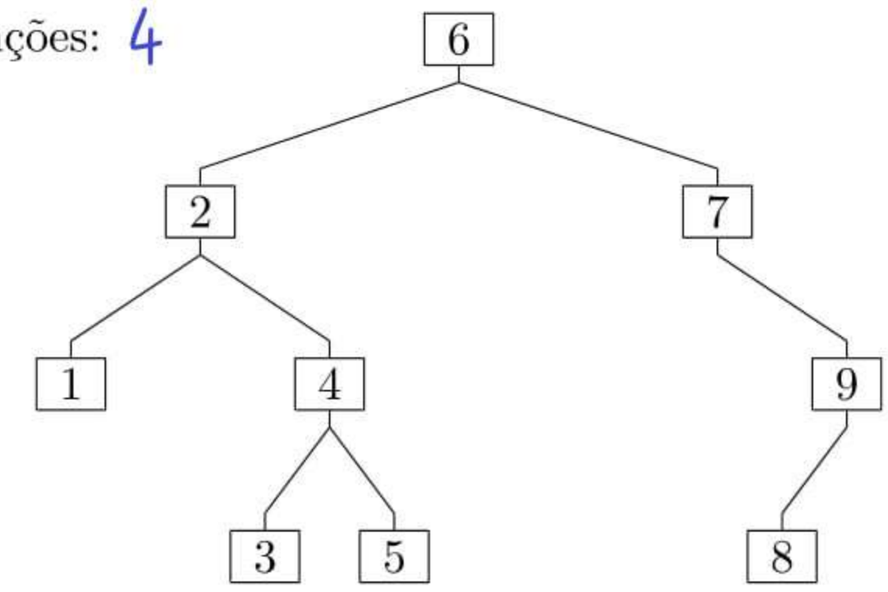
```
void f4(tree_node *link)
{
  if(link != NULL)
  {
    f4(link->left);
    visit(link);
    f4(link->right);
  }
}
```

```
void f5(tree_node *link)
{
  if(link != NULL)
  {
    f5(link->left);
    f5(link->right);
    visit(link);
  }
}
```

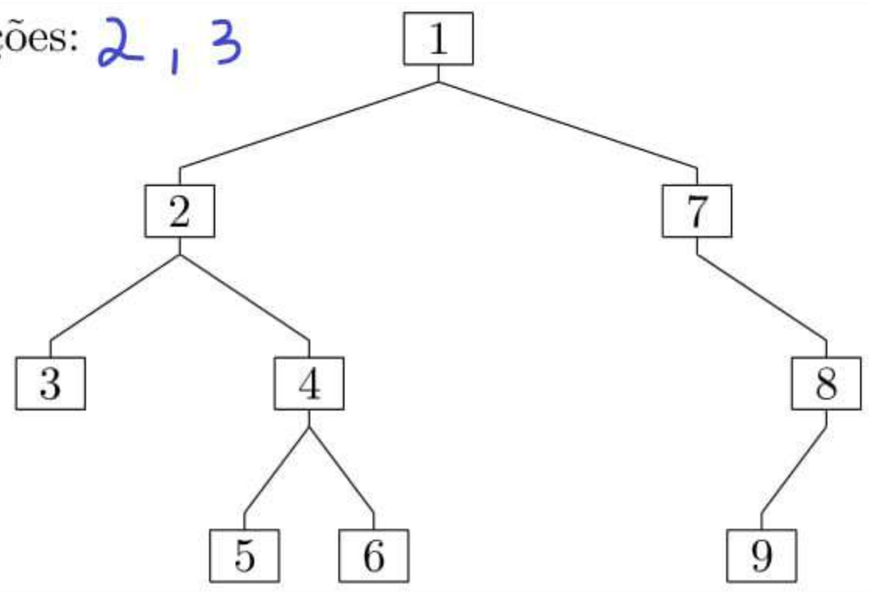
Funções: 5



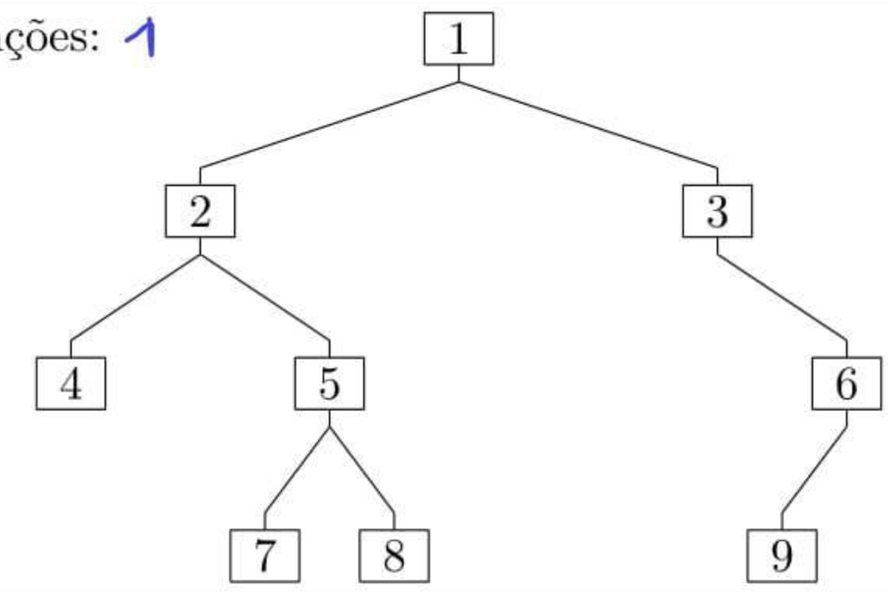
Funções: 4



Funções: 2, 3

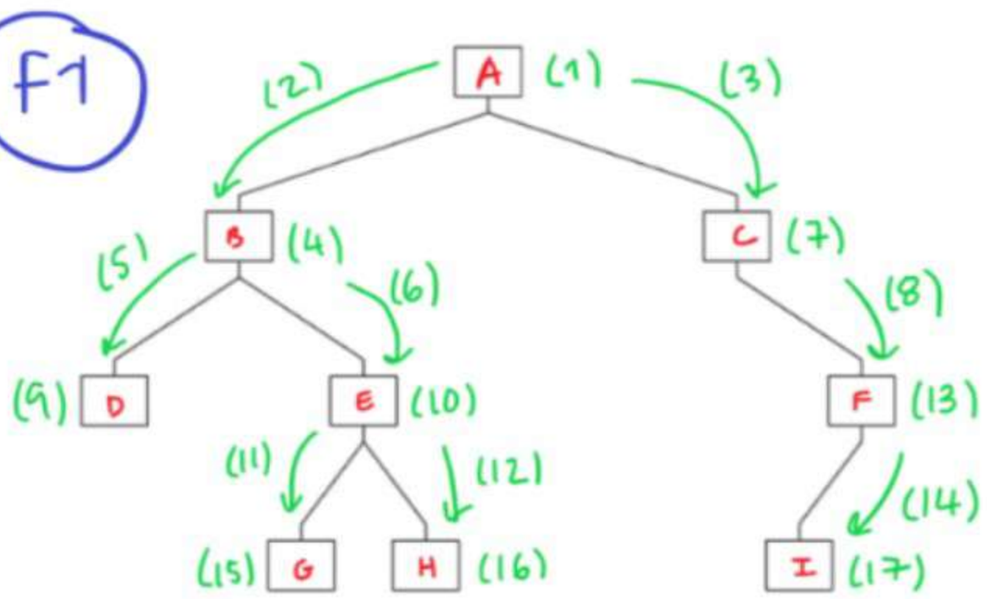


Funções: 1

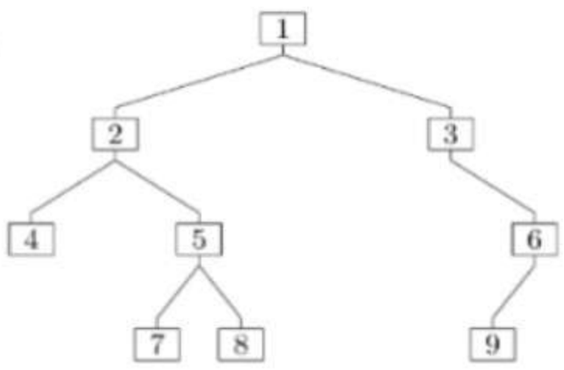


Explicação de cada função

F1



Solução:

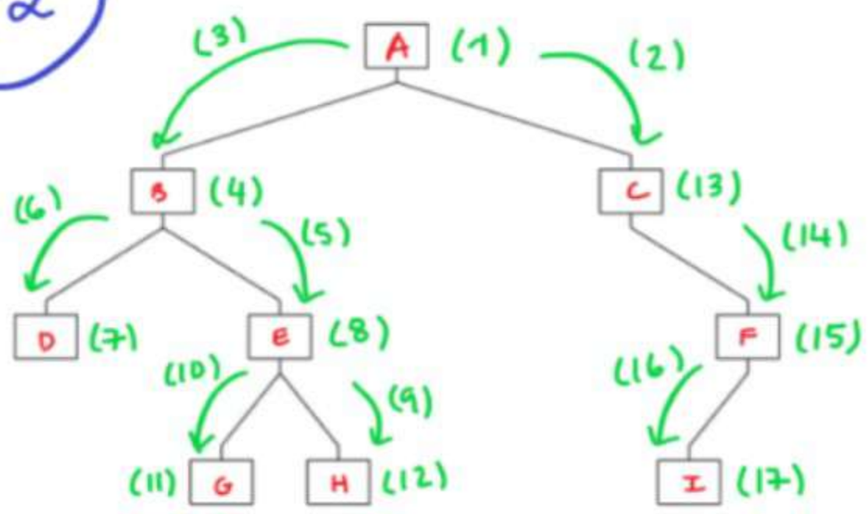


Passos que são feitos na binary tree:
(1) visit(A);
(2) enqueue(q, B); // B = A->left
(3) enqueue(q, C); // C = A->right
(4) visit(B);
(5) enqueue(q, D); // D = B->left
(6) enqueue(q, E); // E = B->right
(7) visit(C);
(C não tem C->left, então...)
(8) enqueue(q, F); // F = C->right
(9) visit(D);
(D não tem D->left nem D->right, então...)
(10) visit(E);
(11) enqueue(q, G); // G = E->left
(12) enqueue(q, H); // H = E->right
(13) visit(F);
(14) enqueue(q, I); // I = F->left
(F não tem F->right, então...)
(15) visit(G);
(16) visit(H);
(17) visit(I);

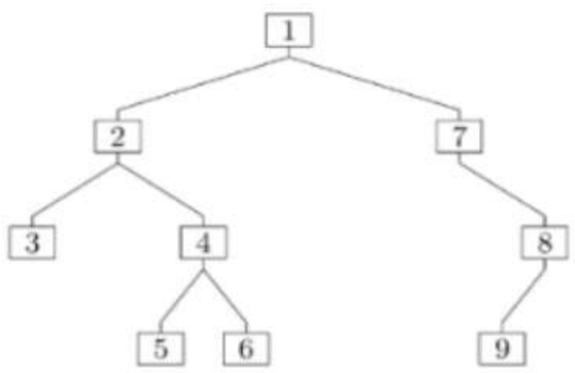
Estado da queue (FIFO)
(0) A
(1)
(2) B
(3) C, B
(4) C
(5) D, C
(6) E, D, C
(7) E, D
(8) F, E, D
(9) F, E
(10) F
(11) G, F
(12) H, G, F
(13) H, G
(14) I, H, G
(15) I, H
(16) I
(17)

Nota: como se pode observar, devido à *queue* ser uma estrutura FIFO (*First In First Out*), quando é preciso colocar um novo elemento, coloca-se no ínicio da *queue* e quando é preciso tirar um, tira-se sempre do final da *queue*.

F2



Solução:



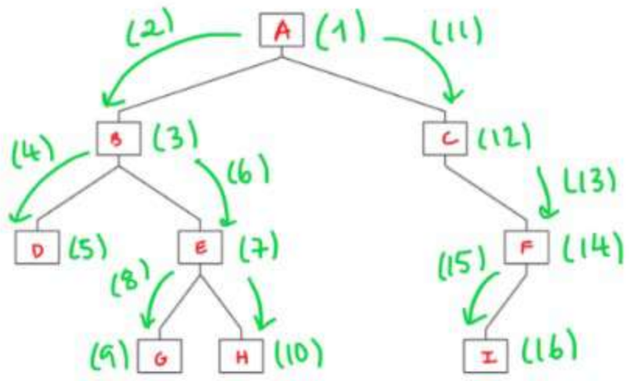
Passos que são feitos na binary tree:
(1) visit(A);
(2) push(s,C); // C = A->right
(3) push(s,B); // B = A->left
(4) visit(B);
(5) push(s,E); // E = B->right
(6) push(s,D); // D = B->left
(7) visit(D);
(D não tem D->right nem D->left, então...)
(8) visit (E);
(9) push(s,H); // H = E->right
(10) push(s,G); // G = E->left
(11) visit(G);
(G não tem G->right nem G->left, então...)
(12) visit(H);
(H não tem H->right nem H->left, então...)
(13) visit(C);
(14) push(s,F); // F = C->right
(C não tem C->left, então...)
(15) visit(F);
(F não tem F->right, então...)
(16) push(s,I); // I = F->left
(17) visit(I);

Estado da stack (LIFO)
(0) A
(1)
(2) C
(3) B, C
(4) C
(5) E, C
(6) D, E, C
(7) E, C
(8) C
(9) H, C
(10) G, H, C
(11) H, C
(12) C
(13)
(14) F
(15)
(16) I
(17)

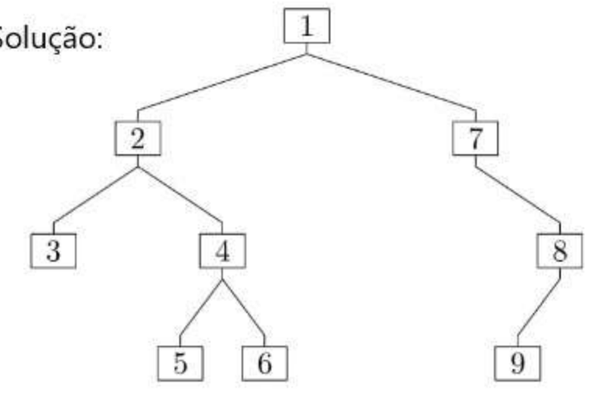
Nota: como se pode observar, devido à *stack* ser uma estrutura LIFO (*Last In First Out*), quando é preciso colocar um novo elemento, coloca-se no ínicio da *stack* e quando é preciso tirar um, tira-se sempre do início da *stack*.

F3

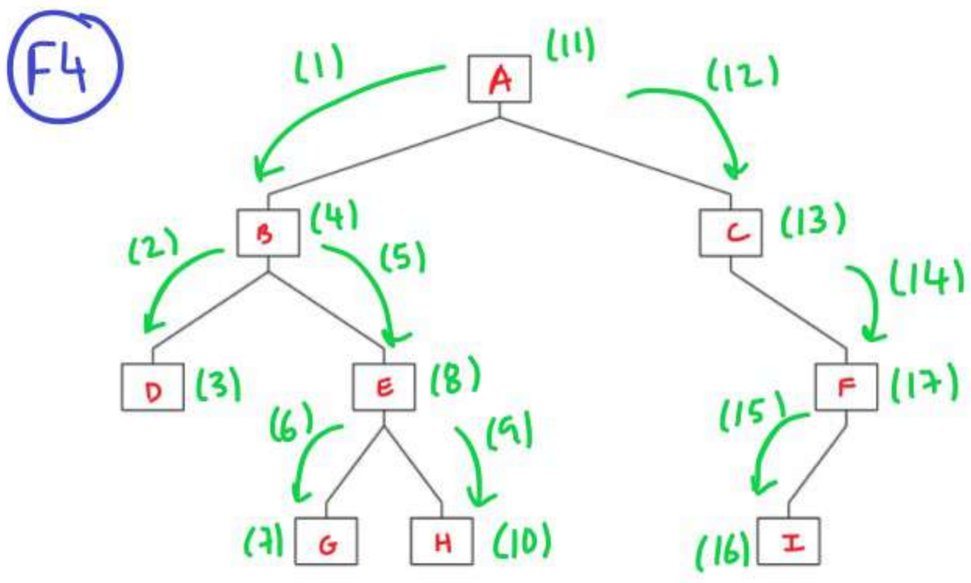
Algoritmo recursivo:
(1) visit(A);
(2) f3(B); // B = A->left
(função começa do ínicio para B devido à chamada recursiva da função...)
(3) visit(B);
(4) f3(D); // D = B->left
(função começa do ínicio para D devido à chamada recursiva da função...)
(5) visit(D);
(D não tem D->left nem D->right, logo a chamada recursiva de D retorna, voltando à chamada recursiva de B...)
(6) f3(E); // E = B->right
(função começa do ínicio para E devido à chamada recursiva da função...)
(7) visit(E);
(8) f3(G); // G = E->left
(9) visit(G);
(G não tem G->left nem G->right, logo a chamada recursiva de G retorna, voltando à chamada recursiva de E...)
(10) f3(H); // H = E->right
(11) visit(H);
(H não tem H->left nem H->right, logo a chamada recursiva de H retorna, voltando à chamada recursiva de E...)
(a chamada recursiva de E retorna, voltando à chamada recursiva de B...)
(a chamada recursiva de B retorna, voltando à chamada original de A...)
(12) f3(C); // C = A->right
(13) visit(C);
(C não tem H->left, então...)
(14) f3(F); // F = C->right
(15) visit(F);
(16) f3(I); // I = F->left
(I não tem I->left nem I->right, logo a chamada recursiva de I retorna, voltando à chamada recursiva de F...)
(F não tem F->right, logo a chamada recursiva de F retorna, voltando à chamada recursiva de C...)
(a chamada recursiva de C retorna, voltando à chamada original de A...)
(a chamada de A retorna)



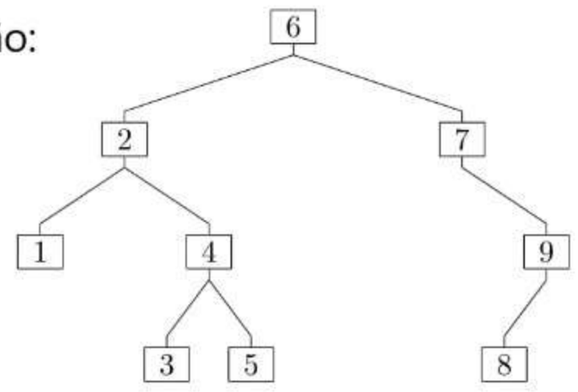
Solução:

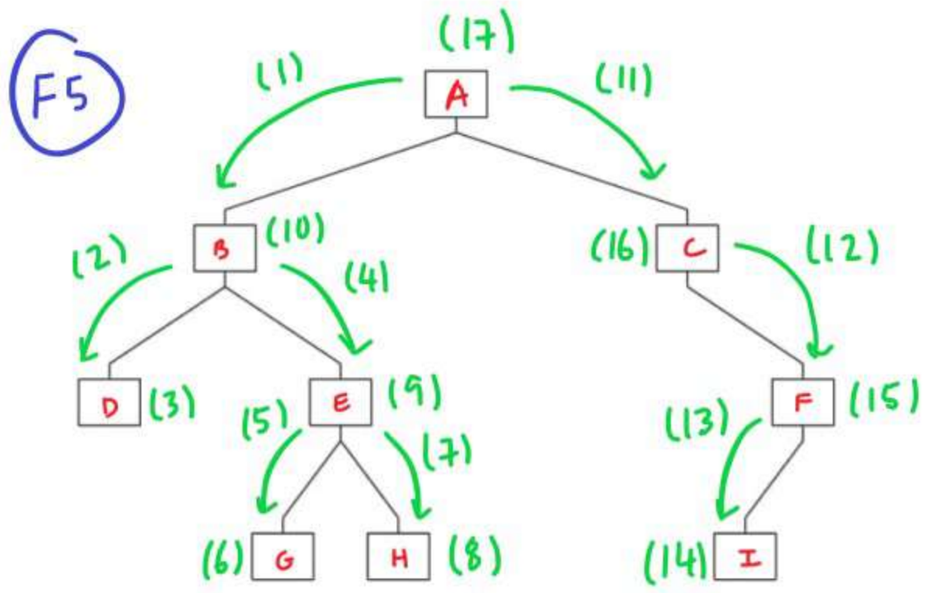


Para f4 e f5, o raciocínio do algoritmo recursivo é análogo ao de f3...



Solução:





Solução: