UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Algoritmos e Estrutura de Dados
Bacharelado em Ciência da Computação
2017.1

<DEINFO - UFRPE>

Lista de Exercícios para VA 2

Aluna:
Mayara Simões de Oliveira Castro
Professor:
Rodrigo Nonamor Pereira Mariano de Souza

4 de setembro de 2017

Exercícios

1º)

```
// Tira um elemento da fila fi e devolve
// o conteudo do elemento removido.
// Supõe que a fila não está vazia.
int tiradafila (void) {
   int x = fila[p++];
   if (p == N) p = 0;
   return x;
}

// Coloca um novo elemento com conteudo y
// na fila fi. Devolve o endereço da
// cabeça da fila resultante.
void colocanafila (int y) {
   fila[u++] = y;
   if (u == N) u = 0;
}

// Retorna 1 se a fila estiver vazia
// e 0 caso contrario.
int filavazia () {
   return (u == p);
}
// Retorna 1 se a fila estiver cheia
// e 0 caso contrario.
int filacheia () {
   return ((u+1) % N == p);
}

// Retorna o comprimento da fila.
int filacomprimento () {
    int tamanho = p - u;
    if(tamanho <0 ) tamanho * = -1;
    return (tamanho);
}
```

2º)

3º)

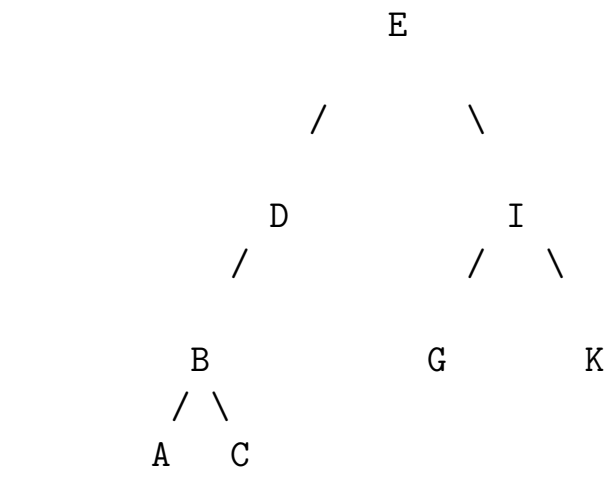
```
TERMINAR
void insere_inicio (Fila* f, float v){

    int prec;
    if (f->n == N) {
    /* fila cheia: capacidade esgotada */
     printf("Capacidade da fila estourou.\n");
     exit(1); /* aborta programa */
     }
/* insere elemento na posição precedente ao início */
    prec = (f->ini - 1 + N) % N; /* decremento circular */
    f->vet[prec] = v;
    f->ini = prec; /* atualiza índice para início */
    f->n++;
}

float retira_fim (Fila* f){

    float v;
    if (fila_vazia(f)) {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    /* retira elemento do início */
    v = f->vet[f->ini];
    f->ini = (f->ini + 1) % N;
    f->n--;
    return v;
}
```

4º)

5º)

6º)

7º)
Não, pois se tomarmos a seguinte árvore binária como exemplo de altura 3, e somarmos a profundidade + altura do nó I, o resultado seria 2, logo, por contra-exemplo não é verdade.



8º)

```
// Recebe uma árvore binária não vazia r
// e devolve o primeiro nó da árvore
// na ordem e-r-d.
noh *primeiro(arvore r){
    if(r->esq == NULL){
        return r;
    }
    return primeiro(r->esq);
}
```

9º)

```
// Recebe uma árvore binária não vazia r
// e devolve o ultimo nó da árvore
// na ordem e-r-d.
noh *ultimo (arvore r) {
    while (r->dir != NULL)
       r = r->dir;
    return r;
}
```

10º)

```
// Recebe o endereço de um nó x. Devolve o endereço
// do nó anterior na ordem e-r-d.
// A função supõe que x != NULL.
noh *anterior (noh *x) {
    if (x->esq != NULL) {
       noh *y = x->esq;
       while (y->dir != NULL) y = y->dir;
       return y;                                  // A
    }
    while (x->pai != NULL && x->pai->esq == x) // B
       x = x->pai;                                // B
    return x->pai;
}
```

11º)

12º)

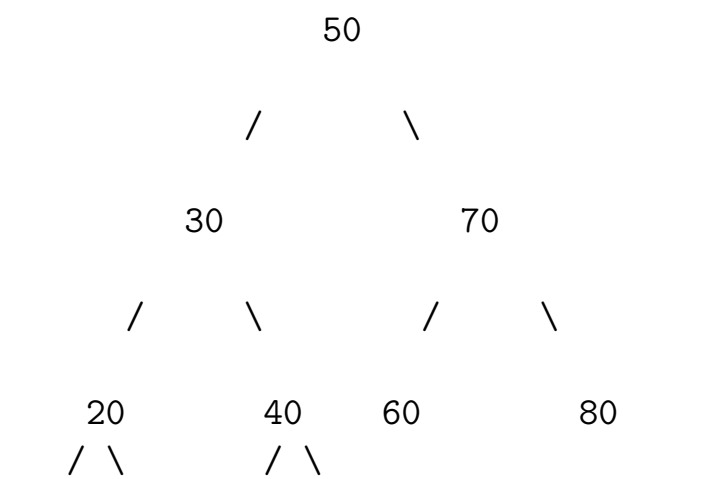
```
// Recebe a raiz r de uma árvore binária
// e retorna true caso a árvore seja de busca.
bool ehBusca (arvore r) {
   if (r != NULL) {
     bool e =  ehBusca (r->esq);

     bool d = ehBusca (r->dir);

     if(r->dir->chave >= r->chave && r->esq->chave <= r->chave && e && d)
        return true;
     }
```

```
     else{
        return false;
     }
   }
   return true;
}
```

13º)

```
//Transforma um vetor crescente em uma árvore
//binária de busca balanceada.
void balancearArvoreBinaria (arvore r, int vetor[], int inicio, int fim)
{
    if (inicio <= fim)
    {
        int meio = (inicio+fim)/2;

        noh *novo;
        novo = malloc (sizeof (noh));
        novo->chave = vetor[meio];
        novo->esq = novo->dir = NULL;

        inserir( r, novo);
        balancearArvoreBinaria ( r, vetor, inicio, meio-1);
        balancearArvoreBinaria ( r, vetor, meio+1, fim);
    }
}
```

14º)



```
 15   25    35 45
              \
               36
```

15º)

16º)

```
// Devolve a altura da árvore binária
// cuja raiz é r.
int altura (arvore r) {
   if (r == NULL)
      return -1; // altura da árvore vazia
   else {
      int he = altura (r->esq);
      int hd = altura (r->dir);
      if (he < hd) return hd + 1;
      else return he + 1;
   }
}

//Preenche o campo fb de cada nó da arvore
void preencherFB(arvore r){
   if (r != NULL) {
      preencherFB (r->esq);
      r->fb = altura(r->esq) - altura(r->dir);
      preencherFB (r->dir);
   }
}
```

17º)

```
//Transforma um vetor crescente em uma árvore
//binária de busca balanceada.
void balancearArvoreBinaria (arvore r, int vetor[], int inicio, int fim)
{
    if (inicio <= fim)
    {
        int meio = (inicio+fim)/2;

        noh *novo;
        novo = malloc (sizeof (noh));
```

```
        novo->chave = vetor[meio];
        novo->esq = novo->dir = NULL;

        inserir( r, novo);
        balancearArvoreBinaria ( r, vetor, inicio, meio-1);
        balancearArvoreBinaria ( r, vetor, meio+1, fim);
    }
}

// Recebe a raiz r de uma árvore binária de busca
// e transforma em uma arvore de busca balanceada
arvore transformarArvore(arvore r){

    int *vetor = malloc(sizeof(int));
    arvoreVetor( r, vetor);

    arvore nova;
    balancearArvoreBinaria ( nova, vetor, 0, sizeof(vetor)/sizeof(int));
}

//Recebe a raiz de uma arvore binaria
//e a transforma em um vetor de ordem crescente
void arvoreVetor (arvore r, int* vetor) {
   if (r != NULL) {
      arvoreVetor (r->esq);

      vetor = realloc(vetor,((sizeof(vetor)/sizeof(int))+ 1) *sizeof(int)
      vetor[(sizeof(vetor)/sizeof(int))-1] = r->chave;

      arvoreVetor (r->dir);
   }
}
```

18º)

19º)

20º)