Projeto de Tutoria do DECSI

Apostila de CEA488 - Algoritmos e Estruturas de Dados I

João Monlevade, MG

Março, 2015

Projeto de Tutoria do DECSI

Apostila de CEA488 - Algoritmos e Estruturas de Dados I

Apostila de exercícios para a Tutoria da Disciplina CEA488 - Algoritmos e Estruturas de Dados I

Universidade Federal de Ouro Preto (UFOP)

João Monlevade, MG

Março, 2015

Agradecimentos

Agradecer à pró-reitoria da UFOP (PROPLAD? PROGRAD?) pelas bolsas do projeto.

Agradescer aos professores e alunos que ministraram as disciplinas e tutorias, respectivamente.

Agradecer aos alunos que prepararam a apostila.

Resumo

Criar um resumo padrão para as apostilas.

Segundo a o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e o tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão .

Palavras-chaves: latex. abntex. editoração de texto.

Sumário

1 Noções de análise de complexidade.

1.1 Conceitos básicos

1. Dê o conceito de:

   a) Algoritmo: Sequência de passos bem definidos para resolver um problema.

   b) Tipo de dados: O tipo de dados de uma variável, constante ou função define o conjunto de valores que a variável, constante ou função pode assumir.

   c) Tipo abstrato de dados: Tipo abstrato de dados pode ser definido como um conjunto de valores com operações bem definidas que podem ser aplicadas à este conjunto.

2. O que significa dizer que uma função de $g(n)$ é $O(f(n))$? Significa dizer que $f(n)$ domina assintoticamente $g(n)$, ou seja, $f(n)$ é o limite assintótico superior para $g(n)$.

1.2 Análise de Complexabilidade

3. Explique a diferença entre $O(1)$ e $O(2)$.

   Resposta:
   Na realidade não há diferenças entre $O(1)$ e $O(2)$, pois:

$$O(2) = O(2 * 1) \tag{1.1}$$

   e pela propriedade

$$O(k * g) = k * O(g) = O(g) \tag{1.2}$$

   temos que:

$$O(2) = O(2 * 1) = 2 * O(1) = O(1) \tag{1.3}$$

   Em fato, é difícil de visualizar tal propriedade pois todo algoritmo em que sua complexidade é $O(k)$, onde k é uma constante positiva maior que zero, são ditos de complexidade constante e por consequência, a execução desses algoritmos não depende do crescimento de n, pois suas instruções para resolver tal problema, seja qual for nossa entrada, são executas com uma ẗelocidadec̈onstate. Basicamente não haverá crescimento do tempo de execução quando n crescer.

4. Comente sobre os tipos de análise em algoritmos.

   Resposta:

   -Análise de um algoritmo particular: Custo (número de vezes que cada parte do algoritmo deve ser executada, seguida do estudo da quantidade de memória necessária).

   -Análise de uma classe de algoritmos: Qual é o melhor algoritmo de menor custo possível para resolver um problema particular? Toda família de algoritmos é investigada. Com família, queremos dizer, todos os algoritmos que resolvem o mesmo problema mas de maneira diferente.

5. Comente sobre os três cenários distintos relacionados ao tempo de execução de um algoritmo.

   Resposta:

   Melhor caso: menor tempo de execução sobre todas as entradas de tamanho n.
   Pior caso: maior tempo de execução sobre todas as entradas de tamanho n.
   Caso médio: média dos tempos de execução de todas as entradas de tamanho n.

6. Qual algoritmo você prefere: um algoritmo que requer n^5 passos ou um que requer 2^n passos?

   Resposta:

   Depende totalmente to tamanho da entrada e do problema que pretendemos resolver.

7. Indique se as afirmativas a seguir são verdadeiras ou falsas e justifique a sua resposta:

   (a) $2^{n+1} = O(2^n)$

   Resposta:

Pela definição essa informação é verdadeira se pudermos achar duas constantes positivas c e n_0 onde $c > 0$ tal que

$$g(n) \leq cf(n)$$

com $n \geq n_0$ substituindo:

$$2^{n+1} \leq c2^n$$

Sabemos que $2^{n+1} = 2^n.2^1$ logo:

$$c \geq \frac{2^n.2^1}{2^n} \Rightarrow c \geq 2$$

Se tomarmos $c = 2$, então:

$$2^{n+1} \leq 2.2^n \qquad para \quad n \geq 0$$

O valor de n foi escolhido substituindo valores na equação encontrada, até achar o valor mínimo que ela se tornava verdadeira!

(b) $2^{2n} = O(2^n)$

Resposta:

Pela definição essa informação é verdadeira se pudermos achar duas constantes positivas c e n_0 onde $c > 0$ tal que

$$g(n) \leq cf(n)$$

com $n \geq n_0$ substituindo:

$$2^{2n} \leq c2^n$$

Sabemos que $2^{2n} = (2^2)^n = 4^n$ logo:

$$4^n \leq c2^n$$

Ao olhar para a expressão encontrada, podemos fazer a seguinte pergunta: existe valor de uma constante que faça com que $2^n \geq 4^n$? Facilmente percebemos que isto é um absurdo, pois nunca uma constante fará que uma função de base 2 cresça mais rapidamente que uma função de base 4, ambas com mesmo comportamento (elevadas a n). Portanto, a afirmação $2^{2n} = O(2^n)$ não é verdadeira!

(d) $f(n) = O(u(n)) \quad e \quad g(n) = O(v(n)) \Rightarrow f(n) + g(n) = O(u(n) + v(n))$

Resposta:

Utilizando as propriedades de complexidade, temos que na primeira parte, para que a preposição seja verdadeira, pela definição, existem constantes positivas c e n_0 tais que $f(n) \leq c_1 u(n)$ para $n \geq n_a$ e $g(n) \leq c_2 v(n)$ para $n \geq n_b$
Portanto:

$$f(n) + g(n) \leq c_1 u(n) + c_2 v(n)$$

para

$$c = max(c_1, c_2) \quad e \quad n_0 = max(n_a, n_b)$$

Logo:

$$f(n) + g(n) \leq O(u(n) + v(n))$$

então, provamos que a afirmação é verdadeira.

(e) $f(n) = O(u(n)) \quad e \quad g(n) = O(v(n)) \Rightarrow f(n) - g(n) = O(u(n) - v(n))$

Resposta:
Se $f(n) - g(n) = O(u(n)) - O(v(n)) = O(u(n)) + (-1)O(v(n))$
Como -1 é uma constante, ela pode ser desconsiderada de acordo com as propriedades da notação O. Portanto:

$$f(n) - g(n) = O(u(n)) + O(v(n))$$

$f(n) - g(n) = O(max\{u(n), v(n)\}) \quad pelas \quad propriedades \quad de \quad O$

E portanto, a afirmação:

$$f(n) - g(n) = O(u(n) - v(n))$$

não é verdadeira!

8. O que caracteriza programas considerados resolvidos e os não resolvidos?

Resposta:

Programas resolvidos são aqueles em que existe algoritmo de tempo polinomial que o resolve.
Programas não resolvidos são aqueles em que não existe algoritmo de tempo polinomial que o resolve, que é o mesmo que dizer, que são programas de ordem exponencial ou superior.

9. Suponha um algoritmo A e um algoritmo B, com funções de complexidade de tempo $a(n) = n^2 - n + 549$ e $b(n) = 49n + 49$ respectivamente. Determine quais valores de n pertencentes ao conjunto dos números naturais para os quais A leva menos tempo para executar do que B.

Resposta:

A ideia é determinar quando:

$$n^2 - n + 549 < 49n + 49 \Rightarrow n^2 - 50n + 500 < 0$$

As raízes para equação são:

$$n_1 = 13.8 \quad e \quad n_2 = 36.1$$

Logo $a(n) < b(n)$ para $13 < n < 37$, com n pertencente aos Naturais.

10. Resolva as seguintes equações de recorrência:

a) $T(n) = T(n-1) + c \quad c \quad constante, \quad n > 1$
$T(1) = 0$

Resposta:

Vamos primeiro encontrar alguns valores de n para $n = n - 1$, $n = n - 2$...

$$T(n-1) = T(n-2) + c$$
$$T(n-2) = T(n-3) + c$$

Vamos agora substituir os valores encontrados em $T(n)$, ou seja, vamos expandir nossa relação de recorrência:
Substituindo $T(n-1)$ em $T(n)$

$$T(n) = [T(n-2) + c] + c$$

Substituindo $T(n-2)$ no $T(n)$ encontrado a partir de $T(n-1)$

$$T(n) = [T(n-3) + c] + 2c = T(n-3) + 3c$$

Se continuássemos a substituir $T(n-3)$, $T(n-4)$ e assim por diante, podemos perceber que, em geral teríamos a seguinte expressão a cada expansão:

$$T(n) = T(n-k) + kc$$

E podemos pensar então em: Quando a recorrência para? A recorrência para no momento em que atingirmos $T(1)$, pois ele é o nosso passo base e então ao invés de uma nova recorrência teríamos um valor. Analisando a expressão geral que encontramos, podemos encontrar facilmente quando $T(n-k) = T(1)$, pois isso evidentemente irá acontecer quando $n - k = 1$, logo

$$n - k = 1 \Rightarrow k = n - 1$$

Agora já sabemos o valor que k tem que assumir para que possamos atingir nosso passo base. Substituindo na expressão geral:

$$T(n) = T(n - (n-1)) + (n-1)c$$
$$T(n) = T(1) + (n-1)c$$

Como já sabemos o valor de $T(1)$ pela definição da recorrência, podemos então substituir em $T(n)$:

$$T(n) = 0 + (n-1)c \Rightarrow T(n) = (n-1)c$$

onde $c > 1$.
E então pelas propriedades da ordem O, podemos concluir que $T(n) = O(n)$.

b) $T(n) = T(n-1) + 2^n \quad n \geq 1$
$T(0) = 1$

Resposta:

Assumindo que calculamos $T(n-1) e T(n-2)$ (como fizemos na letra a), expandimos então $T(n)$

$$T(n) = T(n-1) + 2^n$$
$$T(n) = T(n-2) + 2^{n-1} + 2^n$$
$$T(n) = T(n-3) + 2^{n-2} + 2^{n-1} + 2^n$$

O problema aqui é que não podemos somar $2^{n-1} + 2^n$... porque não temos valores inteiros e sim funções da qual não sabemos o valor de n, correto? Então, este é um caso em que achar o termo geral não é tão simples quanto quando temos constantes somadas a $T(n-k)$. Apesar de ser mais complexo, é visível que $...2^{n-2} + 2^{n-1} + 2^n$ é uma sequência que pode ser representada pelo seguinte somatório:

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 + ... + 2^{n-1} + 2^n$$

Quando temos funções e não constantes presentes na expansão de $T(n)$, o termo geral sempre será uma somatória, e nosso trabalho é identificá-la. Lembre-se que se ficássemos expandindo $T(n)$, uma hora ela tem que chegar no caso base, que para o nosso caso é em $T(0) = 1$, que é exatamente quando 2^0. Uma maneira mais fácil de enxergar este padrão, é somente calcular alguns termos de $T(n)$ e pensar bem lá na frente quanto a recorrência irá parar:

$$T(n) = T(n-1) + 2^n$$
$$T(n-1) = T(n-2) + 2^{n-1}$$
$$T(2) = T(1) + 2^2$$
$$T(1) = T(0) + 2^1$$
$$T(0) = 1$$

Desta maneira, fica mais fácil enxergar a somatória. É importante deixar claro que quando estamos tentando encontrar o termo geral desta maneira, temos que olhar somente para o termo que estamos somando as parcelas de $T(n)$, porque essas parcelas sempre vão ser substituídas por algum valor a partir do momento em que encontramos $T(0) = 1$ e então podemos voltar substituindo todos os valores acima dele, como na recursão. Neste momento teremos vários 2 elevados a algum expoente se somando e está é a somatória que queremos encontrar. A solução para o somatório que encontramos acima é

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

A maneira como encontramos o termo geral da somatória é utilizando propriedades de somatório e também fazendo somas parciais para tentar identificar algum padrão em cada uma destas somas:

1.2.1 Análise de recorrência

1. Apresente a complexidade de espaço para a função recursiva para calcular a sequência de Fibonacci (algoritmo recursivo). **Resposta:**

   Para calcular a complexidade de espaço não podemos utilizar a mesma técnica que utilizamos para calcular a complexidade de tempo, pois estamos querendo saber agora, quanta memória é necessária para manter todas as chamadas recursivas abertas até que elas comecem a ter valor de retorno. A melhor maneira de entendermos isso é encontrando a árvore de recursão para uma chamada qualquer e verificarmos qual a maior profundidade que existe lá, esta profundidade irá nos dizer a complexidade de espaço do algoritmo. Vamos analisar a árvore de recursão para uma chamada de $F(5)$ por exemplo, onde $F(n)$ estamos considerando a nossa função recursiva:
   Podemos ver que a maior profundidade nesta árvore está em L4. Isto nos indica que temos no máximo, 5 nós na memória ao mesmo tempo. Quando estamos falando de nós, estamos querendo nos referir a chamadas abertas na memória ao mesmo tempo esperando por um retorno de valor e não outra chamada recursiva. Desta maneira, podemos concluir que a complexidade de espaço para este algoritmo sempre será $O(n)$, pois é o número máximo de chamadas que teremos armazenadas na memória simultaneamente à espera de um retorno.

2. Prove por indução que $T(2k+1) = T(2k) = 2^{k+1} - 1$ na qual k é um inteiro qualquer e:

$$T(n) = 2T(n-2) + 1 \quad p/n \geq 1$$

$$T(n) = 1 \quad p/n < 1$$

Resposta:

Vamos provar por indução. Primeiro provamos que o caso base é verdadeiro:

$$T(0) = 2^{0+1} - 1 = 1 \quad \checkmark \tag{1.4}$$

Supomos então que $T(2k+1) = T(2k) = 2^{k+1} - 1$ é verdadeiro e queremos provar que $T(2(k+1)+1) = T(2k+3) = T(2(k+1)) = T(2k+2) = 2^{(k+1)+1} - 1 = 2^{k+2} - 1$ é verdadeiro. Provando a primeira parte:

$T(2(k+1)+1) = T(2k+3) =$
$T(2k+3) = 2T(2k+3-2) + 1$ *[Hipótese de indução]*
$T(2k+3) = 2T(2k+1) + 1$

Utilizando o valor de $T(2k+1)$ que supomos acima, temos:

$T(2k+3) = 2(2^{k+1} - 1) + 1$
$T(2k+3) = 2.2^{k+1} - 2 + 1$
$T(2k+3) = 2^{k+2} - 1 \quad \checkmark$

Portanto provamos a primeira parte utilizando a hipótese de indução. Vamos agora, provar a segunda parte:
$T(2(k+1)) = T(2k+2) =$
$T(2k+2) = 2T(2k+2-2) + 1$ *[Hipótese de indução]*
$T(2k+2) = 2T(2k) + 1$

Utilizando o valor de $T(2k)$ que supomos acima, temos:

$T(2k+2) = 2(2^{k+1} - 1) + 1$
$T(2k+2) = 2.2^{k+1} - 2 + 1$
$T(2k+2) = 2^{k+2} - 1 \quad \checkmark$

Logo, provamos por indução que a afirmação $T(2k+1) = T(2k) = 2^{k+1} - 1$ é verdadeira.

2 Recursão

2.1 Recursão

1. Escreva uma função recursiva para calcular o fatorial de n.

```c
#include <stdio.h>

int fatorial(int n){
  if(n == 1)
    return 1;

  return n*fatorial(n-1);
}


int main(){

  int fat;
  int n;

  printf("Entre com o valor n!: ");
  scanf("%d", &n);

  fat = fatorial(n);

  printf("\nO fatorial de %d e: %d",n, fat );

  return 0;
}
```

Código 2.1 – codigos_CEA488/fatorial.c

3 Tipos abstratos de dados

3.1 Lista

1. Crie um programa em C e insira no mesmo a estrutura e as operações necessárias para uso de lista com arranjos.

```c
#include <stdio.h>

#define INICIOARRANJO 1
#define MAXTAM 1000

typedef int TipoApontador;
typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;


typedef struct {
  TipoItem item[MAXTAM];
  TipoApontador primeiro, ultimo;
} TipoLista;

void FLVazia(TipoLista *lista) {

  lista->primeiro = INICIOARRANJO;
  lista->ultimo = lista->primeiro;
}

int Vazia(TipoLista lista){
  return (lista.primeiro == lista.ultimo);
}

void Insere(TipoItem x, TipoLista *lista)
{
  if (lista->ultimo > MAXTAM)
    printf("lista esta cheia\n");
  else {
    lista->item[lista->ultimo-1] = x;
    lista->ultimo++;
  }
}
```

```
void Retira(TipoApontador p, TipoLista *lista, TipoItem *item){
  int aux;
  if(Vazia(*lista) || p >= lista->ultimo)
  {
    printf("Erro : Posicao nao existe \n");
    return;
  }

  *item = lista->item[p-1];
  lista->ultimo--;

  for(aux = p; aux < lista->ultimo; aux++)
    lista->item[aux-1] = lista->item[aux];
}

void Imprime(TipoLista lista){
  int aux;

  for(aux = lista.primeiro-1; aux <= (lista.ultimo-2); aux++)
    printf("%d\n", lista.item[aux].chave);
}


int main(){


return 0;
}
```

Código 3.1 – codigos_CEA488/lista_arranjo.c

2. Crie um programa em C e insira no mesmo a estrutura e as operações necessárias para uso de lista com apontadores.

```
#include <stdio.h>


typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;



typedef struct TipoCelula *TipoApontador;
```

```
typedef struct TipoCelula{
  TipoItem item;
  TipoApontador prox;
} TipoCelula;

typedef struct{
  TipoApontador primeiro, ultimo;
}TipoLista;


void FLVazia(TipoLista *lista) {

  lista->primeiro = (TipoApontador) malloc(sizeof(↩
     TipoCelula));
  lista->ultimo = lista->primeiro;
  lista->primeiro->prox = NULL;
}

int Vazia(TipoLista lista){
  return ( lista.primeiro == lista.ultimo);
}

void Insere(TipoItem x, TipoLista *lista)
{
  lista->ultimo->prox = (TipoApontador) malloc(sizeof(↩
     TipoCelula));

  lista->ultimo = lista->ultimo->prox;
  lista->ultimo->item = x;
  lista->ultimo->prox = NULL;
}

void Retira(TipoApontador p, TipoLista *lista, TipoItem *↩
   item){
  TipoApontador q;
  if(Vazia(*lista) || p == NULL || p->prox == NULL){
    printf("Erro: Lista Vazia ou posicao nao existe.\n");
    return;
  }
  q = q->prox;
  *item = q->item;
  p->prox = q->prox;
  if(p->prox == NULL)
    lista->ultimo = p;
  free(q);

}

void Imprime(TipoLista lista){
```

```
  TipoApontador aux;
  aux = lista.primeiro->prox;

  while(aux != NULL){
    printf("%d\n", aux->item.chave);
    aux = aux->prox;
  }
}


int EstaNalista(TipoChave c, TipoLista *lista){

  TipoApontador aux;
  aux = lista->primeiro->prox;

  while(aux != NULL){
    if(c == aux->item.chave){
      return 1; //1 significa verdadeiro
    }
    aux = aux->prox;
  }

  return 0; //nao achou o elemento 0 significa falso
}

void TrocaDois(TipoItem item1, TipoItem item2, TipoLista ↩
   *lista){

  TipoApontador aux1;
  TipoApontador aux2;

  aux1 = lista->primeiro->prox;

  //vamos verificar se o primeiro elemento esta na lista
  //poderiamos ter chamado a funcao EstaNaLista, mas ↩
     vamos
  //assumir que ela nao existe
  while(aux1 != NULL){
    if(item1.chave == aux1->item.chave){
      break; //para o loop para que aux1 fique apontando
           //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux1 == NULL){
    printf("Item 1 nao esta na lista!\n");
    return;
```

```
    }

    //vamos verificar se o segundo elemento esta na lista
    while(aux2 != NULL){
      if(item2.chave == aux2->item.chave){
        break; //para o loop para que aux2 fique apontando
               //para o item procurado
      }
    }

    //o item nao esta na lista, entao avisamos o usuario
    //e saimos da funcao
    if(aux2 == NULL){
      printf("Item 2 nao esta na lista!\n");
      return;
    }

    //se achou os dois item, vamos troca-los de lugar
    //vamos usar o item1 como auxiliar pois nao
    //precisamos mais dele ou do item 2 agora que ja
    //sabemos a localizacao deles na lista
    //vamos guardar o item para o qual aux1 esta apontado
    item1 = aux1->item;

    //o item para o qual aux1 esta apontando recebe o item
    //que aux2 aponta
    aux1->item = aux2->item;

    //agora so precisamos colocar o item de aux1 que ↩
       guardamos
    //provisoriamente em item1 no item que aux2 aponta
    aux2->item = item1;


}

int main(){


return 0;
}
```

Código 3.2 – codigos_CEA488/lista_dinamica.c

3. Considere a implementação de listas lineares utilizando apontadores e com célula cabeça. Escreva uma função em C ***function*** EstaNaLista(TipoChave c, TipoLista *L) que retorne true se a chave estiver na lista e retorne false se a chave não estiver

na lista. Considere que não há ocorrências de chaves repetidas na lista. Determine a complexidade do seu algoritmo.

```
int EstaNalista(TipoChave c, TipoLista *lista){

  TipoApontador aux;
  aux = lista->primeiro->prox;

  while(aux != NULL){
    if(c == aux->item.chave){
      return 1; //1 significa verdadeiro
    }
    aux = aux->prox;
  }

  return 0; //nao achou o elemento 0 significa falso
}
```

Código 3.3 – codigos_CEA488/esta_na_lista.c

O pior caso é quando não achamos o elemento e então teremos percorrido todos os elementos da lista, ou seja, percorrido n elementos. Logo a função é $O(n)$.

4. Considere a implementação de listas lineares utilizando apontadores e célula cabeça. Escreva uma função para trocar de lugar dois elementos da lista.

```
void TrocaDois(TipoItem item1, TipoItem item2, TipoLista ↩
   *lista){

  TipoApontador aux1;
  TipoApontador aux2;

  aux1 = lista->primeiro->prox;

  //vamos verificar se o primeiro elemento esta na lista
  //poderiamos ter chamado a funcao EstaNaLista, mas ↩
     vamos
  //assumir que ela nao existe
  while(aux1 != NULL){
    if(item1.chave == aux1->item.chave){
      break; //para o loop para que aux1 fique apontando
          //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux1 == NULL){
    printf("Item 1 nao esta na lista!\n");
    return;
```

```
    }

    //vamos verificar se o segundo elemento esta na lista
    while(aux2 != NULL){
      if(item2.chave == aux2->item.chave){
        break; //para o loop para que aux2 fique apontando
            //para o item procurado
      }
    }

    //o item nao esta na lista, entao avisamos o usuario
    //e saimos da funcao
    if(aux2 == NULL){
      printf("Item 2 nao esta na lista!\n");
      return;
    }

    //se achou os dois item, vamos troca-los de lugar
    //vamos usar o item1 como auxiliar pois nao
    //precisamos mais dele ou do item 2 agora que ja
    //sabemos a localizacao deles na lista
    //vamos guardar o item para o qual aux1 esta apontado
    item1 = aux1->item;

    //o item para o qual aux1 esta apontando recebe o item
    //que aux2 aponta
    aux1->item = aux2->item;

    //agora so precisamos colocar o item de aux1 que ↩
       guardamos
    //provisoriamente em item1 no item que aux2 aponta
    aux2->item = item1;

}
```

Código 3.4 – codigos_CEA488/troca_dois.c

5. Um problema que pode surgir na manipulação de lista lineares simples é o de voltar atrás na lista, ou seja, percorrê-la no sentido inverso ao dos apontadores. A solução geralmente adotada é a incorporação à célula de um apontador para o seu antecessor. Lista deste tipo são chamadas duplamente encadeadas.

   a) Declare os tipos necessários para a manipulação da lista;

   Resposta:
   Vamos ver de modo geral o que é preciso mudar em relação as estruturas da lista somente encadeada:

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

```
typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;



typedef struct TipoCelula *TipoApontador;
typedef struct TipoCelula{
  TipoItem item;
  TipoApontador prox, anterior;
} TipoCelula;

typedef struct{
  TipoApontador primeiro, ultimo;
}TipoLista;


void FLVazia(TipoLista *lista) {

  lista->primeiro = (TipoApontador) malloc(sizeof(↩
     TipoCelula));
  lista->ultimo = lista->primeiro;
  lista->primeiro->prox = NULL;
  lista->primeiro->anterior = NULL;
}

int Vazia(TipoLista lista){
  return ( lista.primeiro == lista.ultimo);
}

void Insere(TipoItem x, TipoLista *lista)
{
  lista->ultimo->prox = (TipoApontador) malloc(sizeof↩
     (TipoCelula));
  lista->ultimo->prox->anterior = lista->ultimo;
  lista->ultimo = lista->ultimo->prox;
  lista->ultimo->item = x;
  lista->ultimo->prox = NULL;
}

void Retira(TipoApontador p, TipoLista *lista, ↩
   TipoItem *item){

  if(Vazia(*lista) || p == NULL || p->prox == NULL){
```

```
    printf("Erro: Lista Vazia ou posicao nao existe.\n");
    return;
  }

  *item = p->item;
  if(p->prox == NULL){
    lista->ultimo = p->anterior;
    p->anterior->prox = NULL;

  }else{

    p->prox->anterior = p->anterior;
    p->anterior->prox = p->prox;
  }

  free(p);
}

void Imprime(TipoLista lista){
  TipoApontador aux;
  aux = lista.primeiro->prox;

  while(aux != NULL){
    printf("%d\n", aux->item.chave);
    aux = aux->prox;
  }
}



int main(){


return 0;
}
```

Código 3.5 – codigos_CEA488/lista_duplamente_encadeada.c

b) Escreva um procedimento para retirar da lista a célula apontada por p. Não deixe de considerar eventuais casos especiais.

```
void Retira(TipoApontador p, TipoLista *lista, TipoItem *item){

  if(Vazia(*lista) || p == NULL || p->prox == NULL){
    printf("Erro: Lista Vazia ou posicao nao existe.\n");
    return;
```

```
    }

    *item = p->item;
    if(p->prox == NULL){
      lista->ultimo = p->anterior;
      p->anterior->prox = NULL;
    }else{
      p->prox->anterior = p->anterior;
      p->anterior->prox = p->prox;
    }

    free(p);
}
```

Código 3.6 – codigos_CEA488/retira_lista_duplamente_encadeada.c

6. Crie um programa em C e insira no mesmo a estrutura e as operações necessárias para uso de lista com arranjos.

```
#include <stdio.h>

#define INICIOARRANJO 1
#define MAXTAM 1000

typedef int TipoApontador;
typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;


typedef struct {
  TipoItem item[MAXTAM];
  TipoApontador primeiro, ultimo;
} TipoLista;

void FLVazia(TipoLista *lista) {

  lista->primeiro = INICIOARRANJO;
  lista->ultimo = lista->primeiro;
}

int Vazia(TipoLista lista){
  return (lista.primeiro == lista.ultimo);
}
```

```
void Insere(TipoItem x, TipoLista *lista)
{
  if (lista->ultimo > MAXTAM)
    printf ("lista esta cheia\n");
  else {
    lista->item[lista->ultimo-1] = x;
    lista->ultimo++;
  }
}

void Retira(TipoApontador p, TipoLista *lista, TipoItem *item){
  int aux;
  if(Vazia(*lista) || p >= lista->ultimo)
  {
    printf ("Erro : Posicao nao existe \n");
    return;
  }

  *item = lista->item[p-1];
  lista->ultimo--;

  for(aux = p; aux < lista->ultimo; aux++)
    lista->item[aux-1] = lista->item[aux];
}

void Imprime(TipoLista lista){
  int aux;

  for(aux = lista.primeiro-1; aux <= (lista.ultimo-2); aux++)
    printf ("%d\n", lista.item[aux].chave);
}


int main(){


return 0;
}
```

Código 3.7 – codigos_CEA488/lista_arranjo.c

7. Crie um programa em C e insira no mesmo a estrutura e as operações necessárias para uso de lista com apontadores.

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

```
typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;



typedef struct TipoCelula *TipoApontador;
typedef struct TipoCelula{
  TipoItem item;
  TipoApontador prox;
} TipoCelula;

typedef struct{
  TipoApontador primeiro, ultimo;
}TipoLista;


void FLVazia(TipoLista *lista) {

  lista->primeiro = (TipoApontador) malloc(sizeof(↩
     TipoCelula));
  lista->ultimo = lista->primeiro;
  lista->primeiro->prox = NULL;
}

int Vazia(TipoLista lista){
  return ( lista.primeiro == lista.ultimo);
}

void Insere(TipoItem x, TipoLista *lista)
{
  lista->ultimo->prox = (TipoApontador) malloc(sizeof(↩
     TipoCelula));

  lista->ultimo = lista->ultimo->prox;
  lista->ultimo->item = x;
  lista->ultimo->prox = NULL;
}

void Retira(TipoApontador p, TipoLista *lista, TipoItem *↩
   item){
  TipoApontador q;
  if(Vazia(*lista) || p == NULL || p->prox == NULL){
    printf("Erro: Lista Vazia ou posicao nao existe.\n");
    return;
  }
```

```
  q = q->prox;
  *item = q->item;
  p->prox = q->prox;
  if(p->prox == NULL)
    lista->ultimo = p;
  free(q);

}

void Imprime(TipoLista lista){
  TipoApontador aux;
  aux = lista.primeiro->prox;

  while(aux != NULL){
    printf("%d\n", aux->item.chave);
    aux = aux->prox;
  }
}


int EstaNalista(TipoChave c, TipoLista *lista){

  TipoApontador aux;
  aux = lista->primeiro->prox;

  while(aux != NULL){
    if(c == aux->item.chave){
      return 1; //1 significa verdadeiro
    }
    aux = aux->prox;
  }

  return 0; //nao achou o elemento 0 significa falso
}

void TrocaDois(TipoItem item1, TipoItem item2, TipoLista ↩
   *lista){

  TipoApontador aux1;
  TipoApontador aux2;

  aux1 = lista->primeiro->prox;

  //vamos verificar se o primeiro elemento esta na lista
  //poderiamos ter chamado a funcao EstaNaLista, mas ↩
     vamos
  //assumir que ela nao existe
  while(aux1 != NULL){
    if(item1.chave == aux1->item.chave){
```

```
      break; //para o loop para que aux1 fique apontando
            //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux1 == NULL){
    printf("Item 1 nao esta na lista!\n");
    return;
  }

  //vamos verificar se o segundo elemento esta na lista
  while(aux2 != NULL){
    if(item2.chave == aux2->item.chave){
      break; //para o loop para que aux2 fique apontando
            //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux2 == NULL){
    printf("Item 2 nao esta na lista!\n");
    return;
  }

  //se achou os dois item, vamos troca-los de lugar
  //vamos usar o item1 como auxiliar pois nao
  //precisamos mais dele ou do item 2 agora que ja
  //sabemos a localizacao deles na lista
  //vamos guardar o item para o qual aux1 esta apontado
  item1 = aux1->item;

  //o item para o qual aux1 esta apontando recebe o item
  //que aux2 aponta
  aux1->item = aux2->item;

  //agora so precisamos colocar o item de aux1 que ↩
     guardamos
  //provisoriamente em item1 no item que aux2 aponta
  aux2->item = item1;


}

int main(){
```

```
return 0;
}
```

Código 3.8 – codigos_CEA488/lista_dinamica.c

8. Considere a implementação de listas lineares utilizando apontadores e com célula cabeça. Escreva uma função em C ***function*** EstaNaLista(TipoChave c, TipoLista *L) que retorne true se a chave estiver na lista e retorne false se a chave não estiver na lista. Considere que não há ocorrências de chaves repetidas na lista. Determine a complexidade do seu algoritmo.

```
int EstaNalista(TipoChave c, TipoLista *lista){

  TipoApontador aux;
  aux = lista->primeiro->prox;

  while(aux != NULL){
    if(c == aux->item.chave){
      return 1; //1 significa verdadeiro
    }
    aux = aux->prox;
  }

  return 0; //nao achou o elemento 0 significa falso
}
```

Código 3.9 – codigos_CEA488/esta_na_lista.c

Resposta:
O pior caso é quando não achamos o elemento e então teremos percorrido todos os elementos da lista, ou seja, percorrido n elementos.
Logo a função é $O(n)$.

9. Considere a implementação de listas lineares utilizando apontadores e célula cabeça. Escreva uma função para trocar de lugar dois elementos da lista.

```
void TrocaDois(TipoItem item1, TipoItem item2, TipoLista ↩
   *lista){

  TipoApontador aux1;
  TipoApontador aux2;

  aux1 = lista->primeiro->prox;

  //vamos verificar se o primeiro elemento esta na lista
  //poderiamos ter chamado a funcao EstaNaLista, mas ↩
     vamos
  //assumir que ela nao existe
```

```
  while(aux1 != NULL){
    if(item1.chave == aux1->item.chave){
      break; //para o loop para que aux1 fique apontando
          //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux1 == NULL){
    printf("Item 1 nao esta na lista!\n");
    return;
  }

  //vamos verificar se o segundo elemento esta na lista
  while(aux2 != NULL){
    if(item2.chave == aux2->item.chave){
      break; //para o loop para que aux2 fique apontando
          //para o item procurado
    }
  }

  //o item nao esta na lista, entao avisamos o usuario
  //e saimos da funcao
  if(aux2 == NULL){
    printf("Item 2 nao esta na lista!\n");
    return;
  }

  //se achou os dois item, vamos troca-los de lugar
  //vamos usar o item1 como auxiliar pois nao
  //precisamos mais dele ou do item 2 agora que ja
  //sabemos a localizacao deles na lista
  //vamos guardar o item para o qual aux1 esta apontado
  item1 = aux1->item;

  //o item para o qual aux1 esta apontando recebe o item
  //que aux2 aponta
  aux1->item = aux2->item;

  //agora so precisamos colocar o item de aux1 que ↩
     guardamos
  //provisoriamente em item1 no item que aux2 aponta
  aux2->item = item1;

}
```

Código 3.10 – codigos_CEA488/troca_dois.c

10. Um problema que pode surgir na manipulação de lista lineares simples é o de voltar atrás na lista, ou seja, percorrê-la no sentido inverso ao dos apontadores. A solução geralmente adotada é a incorporação à célula de um apontador para o seu antecessor. Lista deste tipo são chamadas duplamente encadeadas.

    a) Declare os tipos necessários para a manipulação da lista;

    Resposta:
    Vamos ver de modo geral o que é preciso mudar em relação as estruturas da lista somente encadeada:

```
#include <stdio.h>


typedef int TipoChave;

typedef struct {
  TipoChave chave;

} TipoItem;



typedef struct TipoCelula *TipoApontador;
typedef struct TipoCelula{
  TipoItem item;
  TipoApontador prox, anterior;
} TipoCelula;

typedef struct{
  TipoApontador primeiro, ultimo;
}TipoLista;


void FLVazia(TipoLista *lista) {

  lista->primeiro = (TipoApontador) malloc(sizeof(↩
     TipoCelula));
  lista->ultimo = lista->primeiro;
  lista->primeiro->prox = NULL;
  lista->primeiro->anterior = NULL;
}

int Vazia(TipoLista lista){
  return ( lista.primeiro == lista.ultimo);
}

void Insere(TipoItem x, TipoLista *lista)
{
  lista->ultimo->prox = (TipoApontador) malloc(sizeof↩
     (TipoCelula));
```

```
  lista->ultimo->prox->anterior = lista->ultimo;
  lista->ultimo = lista->ultimo->prox;
  lista->ultimo->item = x;
  lista->ultimo->prox = NULL;
}

void Retira(TipoApontador p, TipoLista *lista, ↩
   TipoItem *item){

  if(Vazia(*lista) || p == NULL || p->prox == NULL){
    printf("Erro: Lista Vazia ou posicao nao existe.\↩
       n");
    return;
  }

  *item = p->item;
  if(p->prox == NULL){
    lista->ultimo = p->anterior;
    p->anterior->prox = NULL;

  }else{

    p->prox->anterior = p->anterior;
    p->anterior->prox = p->prox;
  }

  free(p);
}

void Imprime(TipoLista lista){
  TipoApontador aux;
  aux = lista.primeiro->prox;

  while(aux != NULL){
    printf("%d\n", aux->item.chave);
    aux = aux->prox;
  }
}



int main(){


return 0;
}
```

Código 3.11 – codigos_CEA488/lista_duplamente_encadeada.c

b) Escreva um procedimento para retirar da lista a célula apontada por p. Não deixe de considerar eventuais casos especiais.

```
void Retira(TipoApontador p, TipoLista *lista, ↩
    TipoItem *item){

  if(Vazia(*lista) || p == NULL || p->prox == NULL){
    printf("Erro: Lista Vazia ou posicao nao existe.\↩
       n");
    return;
  }

  *item = p->item;
  if(p->prox == NULL){
    lista->ultimo = p->anterior;
    p->anterior->prox = NULL;
  }else{
    p->prox->anterior = p->anterior;
    p->anterior->prox = p->prox;
  }

  free(p);
}
```

Código 3.12 – codigos_CEA488/retira_lista_duplamente_encadeada.c

11. Crie uma função que gere a lista representada abaixo: [2,4,6,7]

a) Construa uma aplicação que cadastre os seguintes dados de um aluno: Nome, nota, idade. Assuma que o TAD Lista para o Cadastro de um aluno já esta pronto, e que o seu TipoItem possua os campo char[20] nome, int nota, int idade. Faça o que se pede:

b) Crie uma função que some a nota de todos os alunos.

c) Crie uma função que retorne a media da nota dos alunos.

d) Crie uma função que percorra a lista e diga se o aluno foi aprovado, ou seja, sua nota é >= 60, ou reprovado, sua nota é <60.

e) Crie uma função que some a idade de todos os alunos.

f) Crie uma função que retorne a media da idade dos alunos.

g) Crie uma função que percorra a lista e diga se o aluno já é pode votar, ou seja, sua idade é >= 16, ou não pode votar, sua idade é <16.
Obs.: Lembre-se de usar as funções de lista, desenhe para entender o problema.

```
void PreencheLista()
{//Exercicio 1
```

```
    //Exercicio 1
    lista l;
    FLVazia(&l);
    TipoItem it;

    it.codigo = 2;
    Inserir(&l,it);

    it.codigo = 4;
    Inserir(&l,it);

    it.codigo = 6;
    Inserir(&l,it);

    it.codigo = 7;
    Inserir(&l,it);
}
int SomaNota(Lista l){//Exercicio 2, letra a
    int soma;
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        soma+=aux->item.nota;
    }
    return soma;
}
int SomaIdade(Lista l){//Exercicio 2, letra d
    int soma;
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        soma+=aux->item.idade;
    }
    return soma;
}
int MediaNota(Lista l){//Exercicio 2, letra b
    int soma=0;
    int cont=0;
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        soma+=aux->item.nota;
        cont++
    }
    return soma/cont;
}
```

```
int MediaIdade(Lista l){//Exercicio 2, letra e
    int soma=0;
    int cont=0;
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        soma+=aux->item.idade;
        cont++
    }
    return soma/cont;
}
void VerificaAprovacao(Lista l){//Exercicio 2, letra c
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        if(aux->item.nota>=60)
            printf("Aluno Aprovado!");
        else
            printf("Aluno Reprovado!");
    }
}
void VerificaEleitor(Lista l){//Exercicio 2, letra f
    Celula *aux;
    aux = lista->primeiro;
    while(aux->prox!=NULL){
        aux=aux->prox;
        if(aux->item.idade>=16)
            printf("Este aluno j possui idade para votar↩
               !");
        else
            printf("Este aluno no possui idade para ↩
               votar!");
    }
}
```

Código 3.13 – codigos_CEA488/lista.c

3.2 Pilha

1. Crie todo o código para pilha representada abaixo, incluindo alocações de memória, as atribuições podem ser feitas manualmente: [2,4,6,7]

```
void PreenchePilha()
{//Exercicio 1
    pilha p;
```

```
    FPVazia(&p);
    TipoItem it;

    it.codigo = 7;
    Empilha(&p,it);

    it.codigo = 6;
    Empilha(&p,it);

    it.codigo = 4;
    Empilha(&p,it);

    it.codigo = 2;
    Empilha(&p,it);

}


void inserenumerosemordemcrescente(Pilha *p,TipoItem it)
{//Exercicio 2
    TipoItem b;
    Pilha aux;
    FPVazia(&aux);
    if(!Vazia(*p))
    {
        Empilha(p,it);
    }
    else
    {
        do
        {
            Desempilha(p,&b);
            Empilha(&aux,b);
        }
        while(b.codigo>it.codigo);

        while(!Vazia(aux))
        {
            Desempilha(p,&b);
            Empilha(&aux,b);
        }
    }
    LiberaMemoria(&aux);//Funcao LiberaMemoria do TAD ↩
        Pilha
}

void ImprimeDecrescente(Pilha p)
{//Exercicio 2, letra B
    /*Como os codigos ja foram inseridos em ordem ↩
```

```
        crescente, basta imprimir a pilha usando a funcao ↩
        Imprimir que a impresso ocorrera automaticamente ↩
        em ordem decrescente*/
    Imprime(p);//Funcao Imprime do TAD Pilha
}
void ImprimeCrescente(Pilha p)
{//Exercicio 2, letra A
    TipoItem b;
    fila f;
    FFVazia(&f)
    while(!Vazia(&p)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    Imprime(f);//Funcao Imprime do TAD Fila
    while(!Vazia(f)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    /*↩
        ------------------------------------------------------------------------

    Voltando os elementos da Fila a posicao inicial*/
    while(!Vazia(p)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    while(!Vazia(&f)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    LiberaMemoria(&f);//Funcao LiberaMemoria do TAD Fila
}
int SomaPilha(Pilha p)
{//Exercicio 2, letra C
    int soma = 0;
    TipoItem b;
    Pilha aux;
    FPVazia(&aux);
    while(!Vazia(p)){
        Desempilha(p,&b);
```

```
            soma+=b.codigo;
            Empilha(&aux,b);
        }
        while(!Vazia(aux)){
            Desempilha(aux,&b);
            Empilha(&p,b);
        }
        return soma;
}
int MediaPilha(Pilha p)
{//Exercicio 2, letra D
        int soma = 0;
        int cont = 0;
        TipoItem b;
        Pilha aux;
        FPVazia(&aux);
        while(!Vazia(p)){
            Desempilha(p,&b);
            soma+=b.codigo;
            cont++;
            Empilha(&aux,b);
        }
        while(!Vazia(aux)){
            Desempilha(aux,&b);
            Empilha(&p,b);
        }
        return soma/cont;
}
```

Código 3.14 – codigos_CEA488/pilha.c

2. Crie uma função que insira números em ordem, lembrando que desta forma o elemento do topo sempre será o maior ou o menor elemento.

   a) Imprima em ordem crescente os elementos;

   b) Imprima em ordem decrescente os elementos;

   c) Crie uma função que retorne a soma de todos os elementos;

   d) Crie uma função que retorne a média dos elementos;
   Modifique o TAD Pilha se julgar necessário, RESPEITE A FORMA DE ACESSO DA PILHA – LIFO!!!

```
void PreenchePilha()
{//Exercicio 1
        pilha p;
        FPVazia(&p);
        TipoItem it;
```

```
    it.codigo = 7;
    Empilha(&p,it);

    it.codigo = 6;
    Empilha(&p,it);

    it.codigo = 4;
    Empilha(&p,it);

    it.codigo = 2;
    Empilha(&p,it);

}


void inserenumerosemordemcrescente(Pilha *p,TipoItem it)
{//Exercicio 2
    TipoItem b;
    Pilha aux;
    FPVazia(&aux);
    if(!Vazia(*p))
    {
        Empilha(p,it);
    }
    else
    {
        do
        {
            Desempilha(p,&b);
            Empilha(&aux,b);
        }
        while(b.codigo>it.codigo);

        while(!Vazia(aux))
        {
            Desempilha(p,&b);
            Empilha(&aux,b);
        }
    }
    LiberaMemoria(&aux);//Funcao LiberaMemoria do TAD ↩
       Pilha
}

void ImprimeDecrescente(Pilha p)
{//Exercicio 2, letra B
    /*Como os codigos ja foram inseridos em ordem ↩
       crescente, basta imprimir a pilha usando a funcao ↩
       Imprimir que a impresso ocorrera automaticamente ↩
```

```
        em ordem decrescente*/
    Imprime(p);//Funcao Imprime do TAD Pilha
}
void ImprimeCrescente(Pilha p)
{//Exercicio 2, letra A
    TipoItem b;
    fila f;
    FFVazia(&f)
    while(!Vazia(&p)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    Imprime(f);//Funcao Imprime do TAD Fila
    while(!Vazia(f)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    /*↩
        ------------------------------------------------------------------------

    Voltando os elementos da Fila a posicao inicial*/
    while(!Vazia(p)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    while(!Vazia(&f)) //note que nesta linha usamos a ↩
        funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    LiberaMemoria(&f);//Funcao LiberaMemoria do TAD Fila
}
int SomaPilha(Pilha p)
{//Exercicio 2, letra C
    int soma = 0;
    TipoItem b;
    Pilha aux;
    FPVazia(&aux);
    while(!Vazia(p)){
        Desempilha(p,&b);
        soma+=b.codigo;
        Empilha(&aux,b);
```

```
    }
    while(!Vazia(aux)){
        Desempilha(aux,&b);
        Empilha(&p,b);
    }
    return soma;
}
int MediaPilha(Pilha p)
{//Exercicio 2, letra D
    int soma = 0;
    int cont = 0;
    TipoItem b;
    Pilha aux;
    FPVazia(&aux);
    while(!Vazia(p)){
        Desempilha(p,&b);
        soma+=b.codigo;
        cont++;
        Empilha(&aux,b);
    }
    while(!Vazia(aux)){
        Desempilha(aux,&b);
        Empilha(&p,b);
    }
    return soma/cont;
}
```

Código 3.15 – codigos_CEA488/pilha.c

3.3 Fila

1. Considere F uma fila não vazia e P uma pilha vazia. Usando apensa a variável temporária x, as quatro operações x<=P, P<=x, x<=F, F<=x e os dois testes P = vazio e F = vazio, escreva um algoritmo para reverter a ordem dos elementos F.

```
algoritmo inverte
  enquanto nao F=vazio
    x<=F
    P<=x
  fim enquanto

  enquanto nao P=vazio
    x<=P
    F<=x
  fim enquanto
fim algoritmo
```

Código 3.16 – codigos_CEA488/troca.c

2. Considere F uma fila não vazia e P uma pilha vazia. Usando apensa a variável temporária x, as quatro operações x<=P, P<=x, x<=F, F<=x e os dois testes P = vazio e F = vazio, escreva um algoritmo para reverter a ordem dos elementos F.

```
algoritmo inverte
  enquanto nao F=vazio
    x<=F
    P<=x
  fim enquanto

  enquanto nao P=vazio
    x<=P
    F<=x
  fim enquanto
fim algoritmo
```

Código 3.17 – codigos_CEA488/troca.c

3. Considerando uma Fila, que guarde o numero do bilhete de cinema dos clientes, dentro do seu TAD TipoItem, assuma que os bilhetes são vendido em ordem crescente, começando do menor até o maior, faça o que se pede:

   a) Imprima em ordem crescente os bilhetes;

   b) Imprima em ordem decrescente os bilhetes;

   c) Crie uma função que diga se a pessoa esta sentada na cadeira de código par ou impar;

   d) Crie uma função que diga o código do ultimo bilhete comprado;
   Modifique o TAD fila se julgar necessário, RESPEITE A FORMA DE ACESSO DA fila – LIFO!!!

```
void ImprimeCrescente(Fila f)
{//Exercicio 1, letra A
    /*Como os codigos ja foram inseridos em ordem ↩
       crescente, basta imprimir a fila usando a funcao ↩
       Imprimir que a impresso ocorrera automaticamente ↩
       em ordem decrescente*/
    Imprime(p);//Funcao Imprime do TAD Fila
}

void ImprimeDecrescente(Fila f)
{//Exercicio 1, letra B
```

```
    TipoItem b;
    Pilha P;
    FPVazia(&p)
    while(!Vazia(&f)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    while(!Vazia(p)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    Imprime(f);//Funcao Imprime do TAD Fila
    /*↩
       ---------------------------------------------------------------------

    Voltando os elementos da Fila a posicao inicial*/
    while(!Vazia(&f)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    while(!Vazia(p)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    LiberaMemoria(&p);//Funcao LiberaMemoria do TAD Pilha
}
void ParOuImpar(Fila f)
{//Exercicio 1, letra C
    TipoItem b;
    Fila aux;
    FFVazia(&aux);
    while(!Vazia(f)){
        Desenfileira(&f,&b);
        if(b.codigo%2==0)
            printf("Cadeira numero %d   par",b.codigo);
        else
            printf("Cadeira numero %d   mpar",b.codigo);
        Enfileira(&aux,b);
    }
    while(!Vazia(aux)){
        Desenfileira(&aux,&b);
```

```
            Enfileira(&f,b);
        }
    }

void InformaCodigoUltimoBilhete(Fila f)
{//Exercicio 1, letra D
    TipoItem b;
    Pilha P;
    FPVazia(&p)
    while(!Vazia(f)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);
    }
    if(!Vazia(f)){
    Desempilha(&p,&b);
    printf("O cdigo do ultimo bilhete vendido : %d",b.↩
       codigo);

    }
    else
    {
      printf("No foram vendidos bilhetes!");
    }
    while(!Vazia(p)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    /*↩
       ------------------------------------------------------------------------

    Voltando os elementos da Fila a posicao inicial*/
    while(!Vazia(&f)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Fila
    {
        Desenfileira(&f,&b);
        Empilha(&p,b);

    }
    while(!Vazia(p)) //note que nesta linha usamos a ↩
       funcao Vazia do TAD Pilha
    {
        Desempilha(&p,&b);
        Enfileira(&f,b);
    }
    LiberaMemoria(&p);//Funcao LiberaMemoria do TAD Pilha
```

```
}
```

Código 3.18 – codigos_CEA488/fila.c

3.4 Árvores

1. Crie uma árvore e insira nela os números [5,4,6,3,8,2,1,9,7];

   a) Crie uma função que imprima toda a árvore;

   b) Crie uma função que imprima toda a árvore em ordem crescente;

   c) Crie uma função que imprima toda a árvore em ordem decrescente;

   d) Crie uma função que percorre toda a árvore e diz se o número é par ou ímpar;

```
void PreencheArvore(Arvore * t)
{ //Exercicio 1
TipoItem item;

item.chave = 5;
insereArvore(t,item);

item.chave = 4;
insereArvore(t,item);

item.chave = 6;
insereArvore(t,item);

item.chave = 3;
insereArvore(t,item);

item.chave = 8;
insereArvore(t,item);

item.chave = 2;
insereArvore(t,item);

item.chave = 1;
insereArvore(t,item);

item.chave = 9;
insereArvore(t,item);

item.chave = 7;
insereArvore(t,item);
}
```

```
void ImprimeArvore(Arvore *x)
{ //Exercicio 2
     if(x!= NULL)
     {
          ImprimeArvore(x->esq);
          printf("\n%d",x->item.chave);
          ImprimeArvore(x->dir);
     }
     else
     {
          return;
     }
}

void EmOrdemCrescente(Arvore *x)
{ //Exercicio 3
     if(x!= NULL)
     {
          EmOrdemCrescente(x->esq);
          printf("\n%d",x->item.chave);
          EmOrdemCrescente(x->dir);
     }
     else
     {
          return;
     }
}

void EmOrdemDecrescente(No *x)
{ //Exercicio 4
     if(x!= NULL)
     {
          EmOrdemDecrescente(x->dir);
          printf("\n%d",x->item.chave);
          EmOrdemDecrescente(x->esq);
     }
     else
     {
          return;
     }
}

void ParOuImpar(Arvore *x)
{ //Exercicio 5
     if(x!= NULL)
     {
          ParOuImpar(x->esq);
          if(x->item.chave%2==0)
               printf("\n%d    par",x->item.chave);
```

```
            else
                printf("\n%d     mpar",x->item.chave);
            ParOuImpar(x->dir);
        }
        else
        {
            return;
        }
}
```

Código 3.19 – codigos_CEA488/arvore.c

4 Métodos de ordenação