Complexidade Prova 2

Questão 1

A função create tem sua complexidade a O(1) pois quando a uma atribuição como é somente o que acontece com a função create sempre vai fazer aquilo só uma vez caso não tenha uma função de repetição.

```
1.  arvoreBI * create(){
2.  arvoreBI * arv = (arvoreBI *) malloc(sizeof(arvoreBI));
3.  if(arv != NULL){
4.  arv->raiz = NULL;
5.  arv->tam = 0;
6.  arv->nivel = 0;
7.  }
8.  return arv;
9.  }
```

A função create tem sua complexidade 1 e O(1) pois quando a uma atribuição como é somente o que acontece com a função create sempre vai fazer aquilo só uma vez caso não tenha uma função de repetição.

```
10. No * createRoot(arvoreBI *arv,int valor){
11. arv->raiz = (No*) malloc(sizeof(No));
12. if(arv->raiz != NULL){
13. arv->raiz->filhoEsquerda = NULL;
14. arv->raiz->filhoDireita = NULL;
15. arv->raiz->valor = valor;
16. arv->tam ++;
17. }
18. return arv->raiz;
19. }
```

A função add ela vai ter complexidade a O(n), pois ela só irá fazer atribuições mas nesse código ele precisa verificar se o número é maior e descer para o próximo filho pra vê se ele também é menor ou maior, então ele vai repetir essa verificação n vezes para achar o lado que vai adicionar o novo filho.

```
20. No * add(arvoreBI * arv,No *raiz,int valor){
21. No * aux = (No*) malloc(sizeof(No));
22. if(aux != NULL){
23. aux->filhoEsquerda = NULL;
24. aux->filhoDireita = NULL;
25. aux->valor = valor;
26. if(raiz->filhoEsquerda != NULL && raiz->filhoDireita!= NULL && raiz->valor >
    valor){
27. add(arv,raiz->filhoEsquerda,valor);
28. }
29. else if(raiz->filhoEsquerda != NULL && raiz->valor > valor){
30. add(arv,raiz->filhoEsquerda,valor);
31. }
```

```
32.         else if(raiz->filhoEsquerda != NULL && raiz->filhoDireita!= NULL && raiz->valor
    < valor){
33.         add(arv,raiz->filhoDireita,valor);
34.         }
35.         else if(raiz->filhoDireita != NULL && raiz->valor < valor){
36.         add(arv,raiz->filhoDireita,valor);
37.         }
38.         else if(raiz->filhoEsquerda == NULL && raiz->filhoDireita == NULL){
39.         if(raiz->valor > valor){
40.         raiz->filhoEsquerda = aux;
41.         arv->tam ++;
42.         }
43.         else if(raiz->valor < valor){
44.         raiz->filhoDireita = aux;
45.         arv->tam ++;
46.         }
47.         }
48.         else if(raiz->filhoEsquerda != NULL && raiz->filhoDireita == NULL){
49.         if(raiz->valor > valor){
50.         raiz->filhoEsquerda = aux;
51.         arv->tam ++;
52.         }
53.         else if(raiz->valor < valor){
54.         raiz->filhoDireita = aux;
55.         arv->tam ++;
56.         }}}}
```

A função clear ela irá ter complexidade igual a O(1) pois ela irá só atribuir valores as variáveis do struct, então dessa forma como não há estrutura de repetição ele irá executar isso somente uma vez dando a complexidade a O(1).

```
57.         void clear(arvoreBI * arv){
58.         arv->raiz->filhoDireita = NULL;
59.         arv->raiz->filhoDireita = NULL;
60.         arv->raiz = NULL;
61.         arv->tam = 0;
62.         printf("Limpo com sucesso!\n\n");
63.         }
```

A função isEmpty ela irá ter complexidade igual a O(1) pois ela irá só comparar valores com as variáveis do struct, então dessa forma como não há estrutura de repetição ele irá executar isso somente uma vez dando a complexidade a O(1).

```
64.         int isEmpty(arvoreBI * arv){
65.         if(arv->raiz == NULL){
66.         return 0;
67.         }
```

```
68.         else{
69.         return -1;
70.         }}
```

A função size tem complexidade igual a O(1), pois ela irá simplesmente verificar o tamanho do Struct e mandar isso para o usuário, ou seja, isso irá acontecer simplesmente uma vez por isso complexidade a O(1).

```
71.         int size(arvoreBI * arv){
72.         return arv->tam;
73.         }
```

A função find_aux tem complexidade igual a O(n), pois ela irá percorrer a arvore até encontrar o valor do usuário ou até perceber que aquele valor não está lá, então ele irá executar n vezes até achar o valor então esse é o motivo da complexidade ser n.

```
74.         int find_aux(arvoreBI * arv,No *raiz,int valor){
75.         No * aux= (No*) malloc(sizeof(No));
76.         aux->valor = valor;
77.         arv->nivel += 1;
78.         if (raiz == NULL) {
79.         return -1;
80.         }
81.         else{
82.         if(raiz->valor == aux->valor){
83.         arv->nivel --;
84.         printf("Find: %d\n",arv->nivel);
85.         }
86.         else if(raiz->filhoEsquerda != NULL && raiz->filhoDireita!= NULL && raiz->valor
    > aux->valor){
87.         find_aux(arv,raiz->filhoEsquerda,valor);
88.         }
89.         else if(raiz->filhoEsquerda != NULL && raiz->valor > aux->valor){
90.         find_aux(arv,raiz->filhoEsquerda,valor);
91.         }
92.         else if(raiz->filhoEsquerda != NULL && raiz->filhoDireita!= NULL && raiz->valor
    < aux->valor){
93.          find_aux(arv,raiz->filhoDireita,valor);
94.          }
95.          else if(raiz->filhoDireita != NULL && raiz->valor < aux->valor){
96.          find_aux(arv,raiz->filhoDireita,valor);
97.          }
98.          else if(raiz->filhoEsquerda == NULL && raiz->filhoDireita == NULL){
99.         arv->nivel == 0;
100.        find_aux(arv,arv->raiz,valor);
101.        }}}
```

A função find tem complexidade igual a O(1), pois ela irá executar tudo isso apenas umas vez chamando a função find_aux e passando a raiz dá arvore e o valor que o usuário quer achar para, procurar o valor passado pelo usuário.

```
102. void find(arvoreBI *arv,int valor) {
103. find_aux(arv,arv->raiz,valor);
104. }
```

A função printAllPre_aux tem complexidade igual a O(n), pois ele irá imprimir todos os elementos recursivamente, ou seja, ele irá executar n vezes até percorrer todos os elementos da arvore e imprimir.

```
105. void printAllPre_aux(No *raiz) {
106. if (raiz != NULL) {
107. printf("%d\n", raiz->valor);
108. printAllPre_aux(raiz->filhoEsquerda);
109. printAllPre_aux(raiz->filhoDireita);
110. }}
```

A função printAllPre tem sua complexidade igual a O(1), pois ele irá executar só uma vez para chamar a função printAllPre_aux com todos os seus parâmetros e lá ela imprimir todos os elementos da arvore.

```
111. void printAllPre(arvoreBI *arv) {
112. printAllPre_aux(arv->raiz);
113. }
```

A função printAllIn_aux tem complexidade igual a O(n), pois ele irá imprimir todos os elementos recursivamente, ou seja, ele irá executar n vezes até percorrer todos os elementos da arvore e imprimir.

```
114. void printAllIn_aux(No *raiz) {
115. if (raiz != NULL) {
116. printAllIn_aux(raiz->filhoEsquerda);
117. printf("%d\n", raiz->valor);
118. printAllIn_aux(raiz->filhoDireita);
119. }}
```

A função printAllIn tem sua complexidade igual a O(1), pois ele irá executar só uma vez para chamar a função printAllIn_aux com todos os seus parâmetros e lá ela imprimir todos os elementos da arvore.

```
120. void printAllIn(arvoreBI *arv) {
121. printAllIn_aux(arv->raiz);
122. }
```

A função printAllIn_aux tem complexidade igual a O(n), pois ele irá imprimir todos os elementos recursivamente, ou seja, ele irá executar n vezes até percorrer todos os elementos da arvore e imprimir.

```
123.    void printAllPost_aux(No *raiz) {
124.    if (raiz != NULL) {
125.    printAllPost_aux(raiz->filhoEsquerda);
126.    printAllPost_aux(raiz->filhoDireita);
127.    printf("%d\n", raiz->valor);
128.    }}
```

A função printAllIn tem sua complexidade igual a O(1), pois ele irá executar só uma vez para chamar a função printAllIn_aux com todos os seus parâmetros e lá ela imprimir todos os elementos da arvore.

```
129.    void printAllPost(arvoreBI *arv) {
130.     printAllPost_aux(arv->raiz);
131.    }
```

Complexidade Prova 2

Questão 2

A criação do Struct não tem complexidade, pois a criação de variável é como se fosse complexidade zero e por isso não contamos essa complexidade como algo que fosse requerer um tempo de execução necessário para precisar de uma complexidade.

```
1.  struct Repositorio{
2.  int *A;
3.  int *D;
4.  int ini;
5.  int fim;
6.  };
```

A função inicializaLista irá ter complexidade a O(1), pois ela é uma atribuição de valores, ou seja, ela irá executar essa função apenas uma vez para chegar no resultado desejado.

```
7.  int tam;
8.  repositorio * inicializaLista(){
9.  repositorio * a = (repositorio*) malloc(sizeof(repositorio));
10. return a;
11. }
```

A função create irá ter sua complexidade igual a O(1), pois ela irá fazer apenas atribuições e mostrar ao usuário através do printf se o repositorio foi criado e como já vimos atribuição é complexidade é O(1).

```
12. void create(repositorio *a,int n){
13. a->A[n];
14. a->D[n];
15. tam=n;
16. printf("Repositorio criada com sucesso!\n");
17. }
```

A função ini irá ter sua complexidade igual a O(1), pois ela irá fazer apenas atribuição a varáveis do Struct e atribuição tem a melhor complexidade possível que é 1 e O(1).

```
18. void ini(repositorio *a){
19. a->fim=0;
20. a->ini=0;
21. }
```

A função vazia verifica se o repositorio está vazio ou não e retorna isso após uma comparação, ou seja, isso irá ser complexidade igual a O(1), pois comparação é praticamente a mesma coisa de uma atribuição.

```
22. int vazia(repositorio *a){
23. return a->fim == a->ini;
24. }
```

A função cheia verifica se o repositorio está cheio ou não e retorna isso após uma comparação, ou seja, isso irá ser complexidade igual a O(1), pois comparação é praticamente a mesma coisa de uma atribuição.

```
25. int cheia(repositorio *a){
26. return a->fim ==tam;
27. }
```

A função estaCheia verifica se o repositorio está cheio a partir da função cheia que irá retorna se está cheio ou não, ou seja, isso irá ser complexidade igual a O(1), pois temos que essa função irá executar só uma vez fazer condições e retornar o valor.

```
28. int estaCheio(repositorio *a){
29. if(cheia(a)){
30. return 0;
31. }
32. else {
33. return -1;
34. }}
```

A função estaVazio verifica se o repositorio está cheio a partir da função vazia que irá retorna se está vazio ou não, ou seja, isso irá ser complexidade igual a O(1), pois temos que essa função irá executar só uma vez fazer condições e retornar o valor.

```
35. int estaVazio(repositorio *a){
36. if(vazia(a)){
37. return 0;
38. }
39. else{
40. return -1;
41. }}
```

A função tamanho irá fazer uma operação com variáveis contidas no Struct que nessa operação irá ter o resultado do tamanho e depois irá retornar esse valor ao usuário, ou seja, essa função tem complexidade a O(1) pois é feito essa operação apenas uma vez.

```
42. int tamanho(repositorio *a){
43. return (a->fim-a->ini);
44. }
```

A função add irá adicionar valores ao repositório, desse modo temos que ele executará essa função atribuindo valoras aos vetores contidos no Struct, além de reorganizar os elementos dos vetores em ordem crescente , ou seja, a complexidade dessa função é igual a $O(n^2)$, pois ela irá rodar uma repetição dentro de outra repetição para conseguir reorganizar os valores nos vetores.

```
45. int add(repositorio *a,int e,int c){
46. if(cheia(a)){
```

```
47.  return 1;
48.  printf("Repositorio cheio!\n");
49.  }
50.  else{
51.  a->A[a->fim] = c;
52.  a->D[a->fim] = e;
53.  if(a->fim != 0){
54.  for(int i=a->ini;i<a->fim;i++){
55.  for(int j=i;j<=a->fim;j++){
56.  if(a->A[i] > a->A[j]){
57.  int aux1,aux2;
58.  aux1 = a->A[j];
59.  aux2 = a->D[j];
60.  a->A[j] = a->A[i];
61.  a->A[i] = aux1;
62.  a->D[j] = a->D[i];
63.  a->D[i] = aux2;
64.  }}}}
65.  a->fim ++;
66.  return 0;
67.  }}
```

A função tratar irá percorrer todo o vetor e verificar qual o produto mais caro e retira-lo do repositório. Dessa forma temos que a complexidade dessa função é igual a $O(n^2)$ pois a função contém uma repetição dentro da outra.

```
68.  int tratar(repositorio *a){
69.  int cont = 0,aux;
70.  for(int i=a->ini;i<a->fim;i++){
71.  for(int j=a->ini;j<a->fim;j++){
72.  if(a->A[i] > a->A[j]){
73.  cont ++;
74.  if(cont == a->fim){
75.  aux=i;
76.  a->fim --;
77.  for(i = aux;i < a->fim; i++){
78.  a->A[i] = a->A[i+1];
79.  }
80.  return a->A[aux];
81.  }}}}}
```

A função remover irá receber um código do usuário e retirar esse código do vetor e depois manter o vetor na sequência então ele precisará de uma repetição para reorganizar o vetor desse modo temos a complexidade igual a $O(n)$.

```
82.  int remover(repositorio *a,int e){
83.  if(vazia(a)){
84.  return -1;
85.  }
```

```
86.  else{
87.  int cont;
88.  for(int i=a->ini;i<a->fim;i++){
89.  if(a->A[i] == e){
90.  cont = 1;
91.  }
92.  if(cont == 1){
93.  a->A[i]=a->A[i+1];
94.  }}
95.  if(cont == 1){
96.  a->fim --;
97.  return 0;
98.  }}}
```

A função imprime irá percorrer o vetor um uma estrutura de repetição e após irá imprimir todos os elementos para o usuário, ou seja, como tem repetição esse código irá executar n vezez, tendo assim a complexidade igual a O(n).

```
99.  void imprime(repositorio *a){
100. if(vazia(a)){
101. printf("Erro: lista vazia.");
102. }
103. else{
104. for(int i=a->ini;i<a->fim;i++){
105. printf("O codigo e:%d  e o valor e:%d\n",a->D[i],a->A[i]);
106. }}}
```

A função pesquisarCodigo irá receber um código do usuário e irá retornar a posição desse elemento no vetor ao usuário, para fazer irá precisará de uma repetição para percorrer o vetor procurando o código, desse modo temos a complexidade igual a O(n).

```
107. int pesquisarCodigo(repositorio *a,int e){
108. for(int i=a->ini;i<a->fim;i++){
109. if(a->D[i]==e){
110. return i;
111. }}
112. return -1;
113. }
```

A função pesquisarValor irá ter complexidade de O(log n), pois a forma de pesquisar dela ao vetor é uma pesquisa binária, ou seja, ela irá pegar o vetor e irá divide-lo para poder achar o valor com um tempo muito maior do que verificar todo o vetor com uma estrutura de repetição.

```
114. int pesquisarValor(repositorio *a,int e){
115. int esquerda=a->ini;
116. int direita=a->fim;
117. while(esquerda<=direita){
118. int meio = (esquerda+direita)/2;
119. if(a->A[meio]==e){
```

```
120.  return meio;
121.  }
122.  else if(a->A[meio]>e){
123.  direita=meio - 1;
124.  }
125.  else{
126.  esquerda=meio + 1;
127.  }}
128.  return -1;
129.  }
```

A função esvaziar irá ter complexidade de O(n), pois ela irá usar uma estrutura de repetição para percorrer todos os valores do vetor e apaga-los. Dessa forma essa função irá executar n vezes.

```
130.  void esvaziar(repositorio *a){
131.  for(int i=a->ini;i<a->fim;i++){
132.  a->D[i]=0;
133.  a->A[i]=0;
134.  }
135.  a->fim=a->ini;
136.  printf("Lista vazia!\n");
137.  }
```

Complexidade Prova 2

Questão 3

A criação do Struct não tem complexidade, pois a criação de variável é como se fosse complexidade zero e por isso não contamos essa complexidade como algo que fosse requerer um tempo de execução necessário para precisar de uma complexidade.

```
1.  struct Catalogo{
2.  int *A;
3.  int *B;
4.  int ini;
5.  int fim;
6.  };
7.  int tam;
```

A função inicializaLista irá ter complexidade a O(1), pois ela é uma atribuição de valores, ou seja, ela irá executar essa função apenas uma vez para chegar no resultado desejado.

```
8.  catalogo * inicializaLista(){
9.  catalogo * a = (catalogo*) malloc(sizeof(catalogo));
10. return a;
11. }
```

A função create irá ter sua complexidade igual a O(1), pois ela irá fazer apenas atribuições e mostrar ao usuário através do printf se o repositorio foi criado e como já vimos atribuição é complexidade é O(1).

```
12. void create(catalogo *a,int n){
13. a->A[n];
14. a->B[n];
15. tam=n;
16. printf("Catalogo criada com sucesso!\n");
17. }
```

A função ini irá ter sua complexidade igual a O(1), pois ela irá fazer apenas atribuição a varáveis do Struct e atribuição tem a melhor complexidade possível que é O(1).

```
18. void ini(catalogo *a){
19. a->fim=0;
20. a->ini=0;
21. }
```

A função vazia verifica se o repositorio está vazio ou não e retorna isso após uma comparação, ou seja, isso irá ser complexidade igual a O(1), pois comparação é praticamente a mesma coisa de uma atribuição.

```
22.  int vazia(catalogo *a){
23.  return a->fim == a->ini;
24.  }
```

A função cheia verifica se o repositorio está cheio ou não e retorna isso após uma comparação, ou seja, isso irá ser complexidade igual a O(1), pois comparação é praticamente a mesma coisa de uma atribuição.

```
25.  int cheia(catalogo *a){
26.  return a->fim ==tam;
27.  }
```

A função cataalogoCheia verifica se o catalogo está cheio a partir da função cheia que irá retorna se está cheio ou não, ou seja, isso irá ser complexidade igual a O(1), pois temos que essa função irá executar só uma vez fazer condições e retornar o valor.

```
28.  int catalogoCheio(catalogo *a){
29.  if(cheia(a)){
30.  return 0;
31.  }
32.  else {
33.  return -1;
34.  }}
```

A função catalogoVazio verifica se o catalogo está cheio a partir da função vazia que irá retorna se está vazio ou não, ou seja, isso irá ser complexidade igual a 1, pois temos que essa função irá executar só uma vez fazer condições e retornar o valor.

```
35.  int catalogoVazio(catalogo *a){
36.  if(vazia(a)){
37.  return 0;
38.  }
39.  else{
40.  return -1;
41.  }}
```

A função tamanho irá fazer uma operação com variáveis contidas no Struct que nessa operação irá ter o resultado do tamanho e depois irá retornar esse valor ao usuário, ou seja, essa função tem complexidade a O(1) pois é feito essa operação apenas uma vez.

```
42.  int tamanho(catalogo *a){
43.  return (a->fim-a->ini);
44.  }
```

A função adicionar tem complexidade igual a O(n), pois ela irá adicionar o valor ao vetor, porém ela irá verificar se o código já não existe no vetor, ou seja, para fazer essa verificação ela irá utilizar uma estrutura de repetição.

```
45.  int adicionar(catalogo *a,int e,int c){
46.  if(cheia(a)){
47.  return -1;
48.  printf("Catalogo cheio!\n");
49.  }
50.  else{
51.  for(int i=a->ini;i<a->fim;i++){
52.  if(a->A[i] ==  e){
53.  return -1;
54.  }}
55.  a->A[a->fim] = e;
56.  a->B[a->fim] = c;
57.  a->fim ++;
58.  return 0;
59.  }}
```

A função removerVent irá ter complexidade igual a O(n), pois ela irá procurar o código do elemento no vetor e depois retira-lo, ou seja, depois ele precisará reorganizar o vetor para mater a ordem e ela faz isso com uma estrutura de repetição que executará n vezes;

```
60.  int removerVent(catalogo *a,int e){
61.  if(vazia(a)){
62.  return -1;
63.  }
64.  else{
65.  int cont;
66.  for(int i=a->ini;i<a->fim;i++){
67.  if(a->A[i] == e){
68.  cont = 1;
69.  }
70.  if(cont == 1){
71.  a->A[i]=a->A[i+1];
72.  a->B[i]=a->B[i+1];
73.  }}
74.  if(cont == 1){
75.  a->fim --;
76.  return 0;
77.  }}}
```

A função imprime irá ter complexidade de O(n), pois ela irá percorrer um vetor com uma estrutura de repetição executando esse código n vezes para conseguir mostrar todos os elementos do vetor.

```
78.  void imprime(catalogo *a){
79.  if(vazia(a)){
```

```
80.     printf("Erro: catalogo vazio.");
81.     }
82.     else{
83.     for(int i=a->ini;i<a->fim;i++){
84.     printf("Codigo:%d",a->A[i]);
85.     printf("Velocidade em RPM e:%d\n",a->B[i]);
86.     }}}
```

A função imprime irá ter complexidade de O(n), pois ela irá percorrer um vetor com uma estrutura de repetição executando esse código n vezes para conseguir encontrar o elementos dito pelo usuário e caso não ache irá retornar -1 que equivale a falso.

```
87.     int catalogoPes(catalogo *a,int e){
88.     for(int i=a->ini;i<a->fim;i++){
89.     if(a->A[i]==e){
90.     return i;
91.     }}
92.     return -1;
93.     }
```

A função imprime irá ter complexidade de O(n), pois ela irá percorrer um vetor com uma estrutura de repetição executando esse código n vezes para conseguir passar por todos os elementos do vetor para deletar todos e deixar o vetor zerado.

```
94.     void limparCatalogo(catalogo *a){
95.     for(int i=a->ini;i<a->fim;i++){
96.     a->A[i]=0;
97.     a->B[i]=0;
98.     }
99.     a->fim=a->ini;
100.    printf("Catalogo vazia!\n");
101.    }
```

A função ordenar irá ter complexidade igual a $O(n^2)$, pois ela irá ter que executar uma estrutura de repetição dentro da outra para poder reorganizar o vetor em ordem crescente de velocidade.

```
102.    int ordenar(catalogo * a){
103.    for(int i=a->ini;i<a->fim;i++){
104.    for(int j=i;j<=a->fim;j++){
105.    if(a->B[i] > a->B[j]){
106.    int aux1,aux2;
107.    aux1 = a->A[j];
108.    aux2 = a->B[j];
109.    a->A[j] = a->A[i];
110.    a->A[i] = aux1;
111.    a->B[j] = a->B[i];
```

```
112.    a->B[i] = aux2;
113.    }}}
114.    return 0;
115.    }
```