Algoritmos

*Vetor dinâmico (*Python List*)*

Atividade avaliativa

1 Introdução

1.1 Informações práticas

Disciplina	:	Algoritmos
Data de entrega	:	09/01/2023
Formato de entrega	:	Relatório **INDIVIDUAL**

1.2 Objetivo

Implementar biblioteca de funções para manipulação de um vetor dinâmico de números inteiros. São duas bibliotecas a serem desenvolvidas, uma implementada com alocação dinâmica de *arrays* e outra implementada com lista ligada.

1. Implementação com vetores de tamanho fixo alocados dinamicamente, com **realocação** a medida que houver necessidade de mais espaço.
2. Implementação com listas **duplamente ligadas**.

1.3 Descrição

Este trabalho consiste na prática de programação, em Linguagem C, relativas a gerenciamento de memória. Para a realização deste trabalho o aluno deve assistir aos vídeos de ponteiros, alocação dinâmica e listas ligadas do curso de Linguagem C disponível em `https://youtube.com/playlist?list=PLJoRO664gMYRn3aiMxb40AKD4QFBh-yV7`. O conteúdo de ponteiros e alocação dinâmica está abordado a partir do vídeo 17 e termina no vídeo 32.

Apresenta-se, por meio deste documento, as instruções referentes à execução deste trabalho. Leia atentamente todo o conteúdo deste documento e, caso surja quaisquer dúvida, entrem em contato com o professor através de mensagem eletrônica (*email*).

2 Vetor dinâmico

Um vetor dinâmico é uma estrutura que possui capacidade de armazenamento modificada dinamicamente, conforme a quantidade de elementos inseridos/removidos do vetor.

A principal forma de acesso aos elementos é através de um valor inteiro associado à localização do elemento no vetor, chamado índice. A quantidade de elementos determina os índices válidos em um vetor. Exemplo: Se o vetor possuir 1234 elementos os índices válidos são de 0 a 1023.

Inserções podem ser feitas para qualquer índice válidos e o próximo índice disponível, que é igual a quantidade de elementos. A remoção pode ser feita para qualquer índice válido do vetor, o causa uma diminuição da quantidade de elementos armazenados. A busca deve retornar o índice do elemento encontrado ou -1 caso o elemento buscado não esteja presente no vetor. A lista de operações (funções) a serem implementadas está descrita na seção 5.

3 Implementação

Para este trabalho deverão ser implementadas as funções para cada uma das implementações descritas na seção 4. Para cada estrutura você deve implementar os arquivos de cabeçalho (`.h`) e código (`.c`), de acordo com exemplo de código mostrado na seção 5.

Casos de testes devem ser desenvolvidos para realizar testes de corretude e de desempenho. Uma análise comparativa entre as formas de implementação deve ser feita, com indicações de quando usar cada uma. Os casos de testes a seguir devem ser feitos e colocados na tabela.

- Inserção consecutiva de elementos.
- Conjunto de buscas de elementos aleatório. Considere o tempo apenas da busca.
- Conjunto de remoções de elementos. Considere apenas o tempo de remoção.
- Conjunto de busca+remoção. Busca elementos aleatório e remove caso encontre.
- Conjunto de inserções+busca+remoção. Gere, de forma aleatória, as operações a serem realizadas e o elemento a ser inserido/buscado/removido.

IMPORTANTE: Faz parte do trabalho a elaboração de casos de teste consistentes. A definição e explicação de casos de testes são fundamentais para uma boa avaliação do relatório.

A quantidade de elementos usados em cada teste deve ser definida de acordo com o desempenho do seu computador. Use valores que tenha significância para a geração de tabelas e gráficos. Inclua casos de testes para testar outras funções, como `size()`, `destroy()`, `percent_occuped()`, etc.

4 Estruturas

Duas formas diferentes de implementação deverão ser feitas:

1. *Array*/Vetor de tamanho fixo, com alocação dinâmica. Este conteúdo esta abordado a partir do vídeo 20 da lista até o vídeo 28.
2. Listas duplamente encadeadas. Este conteúdo é abordado entre os vídeos 29 e 32.

4.1 Alocação dinâmica

Nesta implementação você deve usar um ponteiro uma um *array* estático, pois a capacidade do *array* deve ser aumentada quando o *array* está completo e um novo elemento deve ser inserido. 3 (três) formas distintas de aumento do *array* deve ser implementada:

1. Começa com capacidade 10 (dez) e aumenta de 10 cada vez que precisar de mais.
2. Começa com capacidade 10000 (dez mil) e aumenta de 10000 cada vez que precisar de mais.
3. Começa com capacidade 8 (dois) e duplica cada vez que precisar de mais.

A estrutura a ser usada para o controle do vetor é definida a seguir:

```
struct array_list_int{
  int *a;
  unsigned int size;
  unsigned int capacity;
};
```

Obs: *Campos extras podem ser adicionados e devem ser explicados no relatório.*

4.2 Lista ligada

Nesta implementação você deve usar uma lista duplamente ligada para armazenar os elementos. As estruturas a seguir definem o `list_node` e o `list` a serem usadas na implementação.

```
struct linked_list_int_node{
    int value;
    struct linked_list_int_node *prev;
    struct linked_list_int_node *next;
  };

struct linked_list_int{
  struct linked_list_int_node *first;
  struct linked_list_int_node *last;
  unsigned int size;
};
```

As duas estruturas que representam o vetor dinâmico devem ter uso e comportamento equivalente ao `Python List`.

5 Funções

As funções a seguir devem ser implementadas para se realizar todos os testes. Em anexo há arquivos fonte com uma base de implementação feita para o vetor implementado com alocação dinâmica. Para a implementação com listas ligadas você deve criar os arquivos e substituir `array_list` por `linked_list`.

Também, em anexo, há um arquivo de teste de inserção, chamado `teste01.c`. A explicação de como compilar e executar está no arquivo `README.txt`.

`array_list_int * array_list_create()` Cria uma nova lista em memória. Retorna um ponteiro para a lista recém criada.

`int array_list_get(array_list_int * list, int index, int *error)` Retorna um elemento localizado no índice `index`.

`unsigned int array_list_push_back(array_list_int * list, int i)` Adiciona um novo elemento ao final da lista.

`unsigned int array_list_pop_back(array_list_int * list)` Remove um elemento do final da lista, caso exista. O final da lista é o índice `tamanho-1`.

`unsigned int array_list_size(array_list_int * list)` Retorna a quantidade de elementos na lista.

`int array_list_find(array_list_int * list, int element)` Busca um elemento na lista. Retorna o índice onde ele se encontra ou -1 se ele não estiver na lista.

`unsigned int array_list_insert_at(array_list_int * list, int index, int value)` Insere um novo elemento na lista, aumentando a quantidade de elementos. O elemento inserido deve se localizar no índice `index`.

`int array_list_remove_from(array_list_int * list, int index)` Remove elemento localizado no índice `index`.

`unsigned int array_list_capacity(array_list_int * list)` Retorna o espaço efetivamente reservado para a lista.

`double array_list_percent_occupied(array_list_int * list)` Retorna o percentual do espaço reservado efetivamente ocupado por elementos da lista. O percentual é um valor entre $0,0$ e $1,0$.

`void array_list_destroy(array_list_int * list)` Libera memória usado pela lista `list`.

6 Entrega

O trabalho deve ser entregue em um arquivo **pdf** com tabelas, gráficos e explicações das implementações e dos testes. Um exemplo de organização do relatório é mostrado a seguir:

```
01 - Introdução
02 - Vetores dinâmicos
03 - Implementação
  03.01 - Organização dos arquivos fontes
  03.02 - Arrays com alocação dinâmica
  03.03 - Lista ligada
  03.04 - Testes
04 - Resultados
05 - Conclusão
```

Os arquivos contendo o código fonte pode ser colocado como anexo no relatório ou em um arquivo `.zip` a parte. A avaliação do trabalho é feita pelo relatório, os códigos fonte são usados para gerar os valores de tempo de execução e uso de memória dos teste e para auxiliar a construção do relatório.

Esta organização é apenas uma sugestão, fique a vontade para modificar.