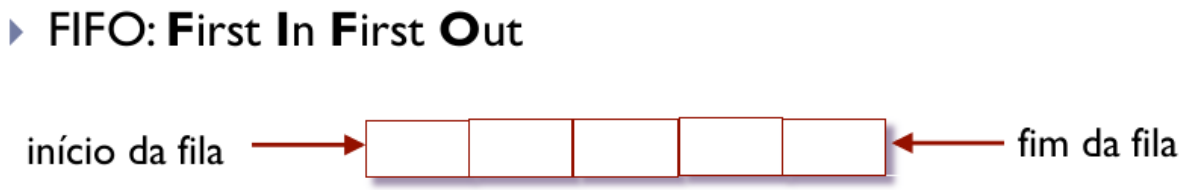
UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

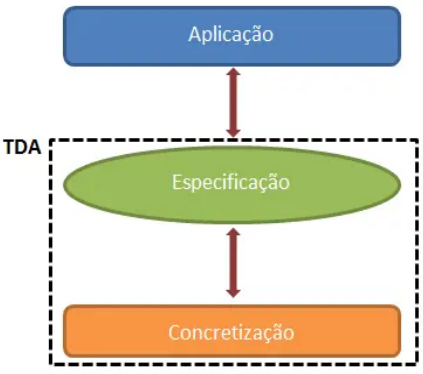
Aluno: **Rosivaldo Lucas da Silva**
Matrícula: **20190028170**
Disciplina: **Estruturas de Dados**
Semestre: **2021.1**

QUESTÃO 1

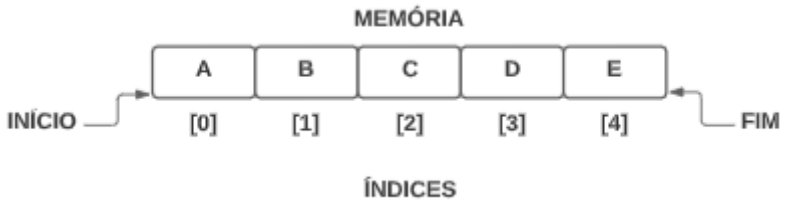
Estruturas de Dados: No ramo da computação, estruturas de dados é definido como o estudo das mais variadas formas e maneiras de se armazenar de forma agrupa e lógica dados que compartilham entre si características semelhantes. No mundo real existem diversos exemplos de estruturas de dados como uma fila de um banco que tem como característica que o primeiro que entra (chega) na fila é o primeiro que sai (atendido), esse conceito nas estruturas de dados é conhecido como **FIFO** (**F**irst **I**n **F**irst **O**ut).



Tipo Abstrato de Dados: O tipo abstrato de dados é um mecanismo utilizado pelas linguagens de programação para poder abstrair as funcionalidades que uma determinada estrutura possa realizar. Assim focando apenas no que cada operação faz, e não como faz. Um exemplo é a criação de um **TAD** para a abstração da representação de um ponto no R2, onde pode ser adicionado uma funcionalidade para realizar o cálculo da distância entre dois pontos. Assim o **TAD** irá apenas disponibilizar essa operação para o usuário, mas não irá se preocupar como vai ser realizado o cálculo da distância, apenas irá retorná-lo para o usuário.



Arrays Estáticos: O tipo de dados Array é a forma mais simples para organizar os dados na memória do computador. O array é um tipo de dado homogêneo onde só pode armazenar dados de um mesmo tipo, uma característica dos arrays é que ele armazena os dados de forma sequencial, assim tornando seu acesso rápido e fácil. Na linguagem C quando faz se necessário o uso do tipo array devemos indicar a sua capacidade máxima e também o tipo de dados que ele irá armazenar.



Arrays Dinâmicos: Na linguagem de programação C é possível utilizar o recurso de alocação dinâmica de memória, onde a memória para uma variável é alocada em tempo de execução. As funções para realizar esse tipo de alocação de memória está presente na biblioteca padrão **stdlib.h**. A alocação dinâmica é muito utilizada para alocar arrays que de antemão não se sabe o seu tamanho máximo de elementos. Para implementar a alocação dinâmica na linguagem C é preciso utilizar o conceito de ponteiros. Onde um ponteiro é uma variável especial que pode aguardar um endereço de memória para um determinado tipo de dado. Segue abaixo um exemplo de como é realizada a alocação dinâmica de memória para uma variável de tipo ponteiro de inteiro que vai guardar a referência para o primeiro endereço de memória alocado.

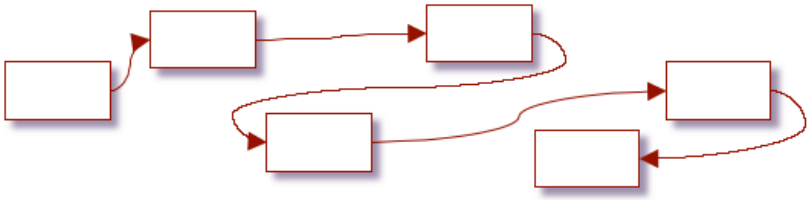
int *array = (int *) malloc(sizeof(int) * 10)

Onde o código acima chama a função malloc que recebe a quantidade de espaço que será armazenado, e o operador sizeof(int) irá retornar o número de bytes que um inteiro ocupa e depois é realizado a multiplicação desse valor pela quantidade que se deseja armazenar, nesse caso irá criar um array com 10 elementos alocados dinamicamente.

Lista Simplesmente Encadeada: As listas são estruturas de dados muito utilizadas para armazenar conjuntos de elementos relacionados. Um exemplo pode ser uma lista de clientes de uma agência bancária, onde cada elemento da lista faz referência a um cliente distinto. Na linguagem C as listas podem ser implementadas utilizando o conceito de alocação dinâmica de memória, assim cada vez que se fizer necessário inserir um novo elemento na lista é primeiro alocado um novo espaço na memória para ele. Essa abordagem traz diversos benefícios como um melhor aproveitamento da memória e há mais rapidez nos processos de inserção/remoção da lista, mas também existem algumas desvantagens ao se utilizar o conceito de listas encadeadas, que é o considerável aumento na complexidade dos algoritmos necessários para sua implementação, visto que na linguagem C essas listas são

implementadas utilizando ponteiros para realizar a ligação dos elementos porque como os elementos são alocados dinamicamente, então eles não estão organizados na memória de forma sequencial, assim dificultando o seu acesso.

- **Alocação Encadeada**
  - Elementos dispostos aleatoriamente na memória, encadeados por ponteiros



QUESTÃO 2

a) **TAD Array Dinâmico Realocável:**

```c
struct vetor {
    Elemento *vetor;
    int tam;
    int qtd_max;
};
```

```c
VetorDinamico *vd_criar() {
    VetorDinamico *vetorDinamico = (VetorDinamico *) malloc(sizeof(VetorDinamico));
    vetorDinamico->vetor = (Elemento *) malloc(sizeof(Elemento) * 10);

    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL) {
        return NULL;
    }

    vetorDinamico->qtd_max = 10;
    vetorDinamico->tam = 0;

    return vetorDinamico;
}
```

```c
int vd_inserir(VetorDinamico *vetorDinamico, Elemento elemento) {
    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL) {
        return 0;
    }

    if (vetorDinamico->tam == vetorDinamico->qtd_max) {
        vetorDinamico->vetor = (Elemento *) realloc(vetorDinamico->vetor, sizeof(Elemento) * 2);

        if (vetorDinamico->vetor == NULL) {
            return 0;
        }

        vetorDinamico->qtd_max *= 2;
    }

    vetorDinamico->vetor[vetorDinamico->tam++] = elemento;

    return 1;
}
```

```c
int vd_acessar_elemento(VetorDinamico *vetorDinamico, int i, Elemento *elemento) {
    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL || i < 0 || i >= vetorDinamico->tam) {
        return 0;
    }

    (*elemento) = vetorDinamico->vetor[i];

    return 1;
}
```

```c
int vd_tamanho(VetorDinamico *vetorDinamico) {
    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL) {
        return -1;
    }

    return vetorDinamico->tam;
}
```

```c
int vd_liberar(VetorDinamico *vetorDinamico) {
    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL) {
        return 0;
    }

    free(vetorDinamico->vetor);
    free(vetorDinamico);

    return 1;
}
```

```c
void vd_imprimir(VetorDinamico *vetorDinamico) {
    if (vetorDinamico == NULL || vetorDinamico->vetor == NULL) {
        return;
    }

    int i;
    for (i = 0; i < vetorDinamico->tam; i++) {
        printf("[%d] = %d ", i, vetorDinamico->vetor[i]);
    }
    printf("\n");
}
```

b) TAD Lista Simplesmente Encadeada:

```c
#include <stdio.h>
#include <stdlib.h>
#include "lista.h"

typedef struct no No;

struct no {
    Elem elem;
    No *prox;
};

struct lista {
    No *ini, *fim;
};
```

```c
void ll_imprimir(Lista *lista) {
    No *p = lista->ini;

    while(p != NULL) {
        printf("%.2f ", p->elem);
        p = p->prox;
    }

    printf("\n");
}
```

```c
Lista *ll_criar(Elem elem) {
    Lista *lista = (Lista *) malloc(sizeof(Lista));
    No *no = (No *) malloc(sizeof(No));

    if (lista == NULL || no == NULL) {
        return NULL;
    }

    no->elem = elem;
    no->prox = NULL;

    lista->ini = no;
    lista->fim = no;
    lista->fim->prox = NULL;

    return lista;
}
```

```c
int ll_liberar(Lista *lista) {
    if (lista == NULL || lista->ini == NULL) {
        return 0;
    }

    No *p = lista->ini;
    while (p != NULL) {
        lista->ini = lista->ini->prox;
        free(p);
        p = lista->ini;
    }

    free(lista);

    return 1;
}
```

```c
int ll_inserir_inicio(Lista *lista, Elem elem) {
    No *novoNo = (No *) malloc(sizeof(No));
    if (lista == NULL || novoNo == NULL) {
        return 0;
    }
    novoNo->elem = elem;
    novoNo->prox = lista->ini;
    lista->ini = novoNo;
    return 1;
}
```

```c
int ll_tamanho(Lista *lista) {
    if (lista == NULL) {
        return -1;
    }

    No *p = lista->ini;
    int cont = 0;
    while (p != NULL) {
        cont++;
        p = p->prox;
    }

    return cont;
}
```

```c
int ll_inserir_fim(Lista *lista, Elem elem) {
    if (lista == NULL) {
        return 0;
    }

    No *novoNo = (No *) malloc(sizeof(No));

    if (novoNo == NULL) {
        return 0;
    }

    novoNo->elem = elem;
    novoNo->prox = NULL;

    lista->fim->prox = novoNo;
    lista->fim = novoNo;

    return 1;
}
```

```c
int ll_pertence(Lista *lista, Elem elem) {
    if (lista == NULL) {
        return 0;
    }
    No *p = lista->ini;
    while (p != NULL) {
        if (p->elem == elem) {
            return 1;
        }
        p = p->prox;
    }
    return 0;
}
```

```c
int ll_eh_vazia(Lista *lista) {
    if (lista == NULL) {
        return 0;
    }
    if (ll_tamanho(lista) != 0) {
        return 0;
    }
    return 1;
}
```

```c
int ll_inserir_ordenado(Lista *lista, Elem elem) {
    if (lista == NULL) {
        return 0;
    }
    if (ll_eh_vazia(lista)) {
        ll_inserir_inicio(lista, elem);
        return 1;
    }
    No *novo = (No *) malloc(sizeof(No));
    if (novo == NULL) {
        return 0;
    }
    novo->elem = elem;
    No *ant = NULL;
    No *p = lista->ini;
    while (p != NULL && p->elem < elem) {
        ant = p;
        p = p->prox;
    }
    if (p == lista->ini) {
        novo->prox = lista->ini;
        lista->ini = novo;
        return 1;
    } else {
        novo->prox = ant->prox;
        ant->prox = novo;
    }
    return 1;
}
```

```c
int ll_ordenar(Lista *lista, TipoOrdenacao tipo) {
    if (lista == NULL) {
        return 0;
    }
    int aux;
    if (tipo == ll_asc) {
        for (No *i = lista->ini; i->prox != NULL; i = i->prox) {
            for (No *j = i->prox; j != NULL; j = j->prox) {
                if (i->elem > j->elem) {
                    aux = j->elem;
                    j->elem = i->elem;
                    i->elem = aux;
                }
            }
        }
    } else if (tipo == ll_desc) {
        for (No *i = lista->ini; i->prox != NULL; i = i->prox) {
            for (No *j = i->prox; j != NULL; j = j->prox) {
                if (i->elem < j->elem) {
                    aux = j->elem;
                    j->elem = i->elem;
                    i->elem = aux;
                }
            }
        }
    }
}
```

QUESTÃO 3

a) **Inserção em Listas:**

i) **Inserção no Início:** Para realizar a inserção no início da lista aloca um novo **Nó** para a lista e insere no **Nó** o elemento a ser inserido e faz o seu prox receber a lista, e posteriormente faz o inicio da lista receber o novo **Nó**. Com essa lógica é possível inserir numa lista que esteja vazia ou que já tenha elementos.

ii) **Inserção no Meio:** A inserção no meio da lista é usada quando se deseja inserir o elemento de forma ordenada na lista. Assim é necessário realizar a verificação se a lista está vazia, se estiver é usado a inserção de inserir no início da lista se não é procurar onde inserir o elemento, então é criado os **Nós** auxiliares que aponta para o início da lista e outro que irá apontar para o elemento anterior ao **Nó** inicial. Agora é necessário iterar sobre a lista até o ponteiro auxiliar for

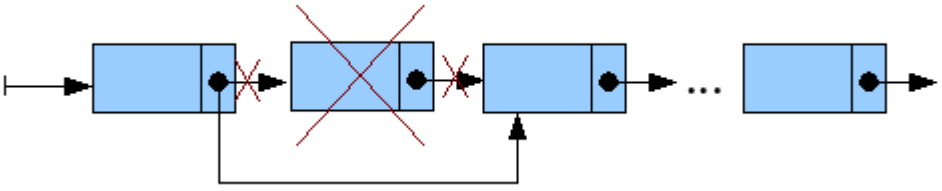
diferente de NULL e verificar se o elemento é menor que o elemento passado para inserção. Logo em seguida é feito uma verificação de ifs verificando se o ponteiro aux é igual a lista, se for significa que o elemento será inserido no início, se não o elemento será inserido no meio da lista ou no final da lista.

iii) **Inserção no Fim:** Para a inserção do elemento no fim da lista primeiro é verificado se a lista contém algum elemento, se não o elemento será inserido no início da lista, mas se a lista não estiver vazia é necessário iterar a lista e procurar o último elemento da lista e fazer ele apontar para o **Nó** criado que já foi inicializado com o seu elemento e o seu prox com NULL, assim ele passará a ser o último elemento da lista.

b) **Remoção em Listas:** A funcionalidade de remoção em uma lista tem que levar em consideração onde se encontra o elemento que se deseja remover, o mesmo pode está no início, meio e fim da lista, assim sendo necessário realizar uma lógica para cada situação. Para a implementação dessas possíveis situações é necessário a criação de dois **Nós** auxiliares que guardam a posição inicial da lista e a posição anterior à inicial, e conforme o **Nó que** guarda a posição inicial for sendo incrementado para a próxima posição o outro **No** auxiliar irá apontar sempre para o antecessor do **Nó inicial,** assim podendo sempre contar com as informações do **Nó anterior.**

i) **Remoção no Início:** Na remoção de um elemento no início da lista é necessário realizar a verificação se o **Nó auxiliar** ainda está apontando para o início da lista, isso indica que o elemento está no início. Então basta apenas fazer a **lista->ini** receber o ponteiro para o próximo elemento e por fim realizar a desalocação da memória do ponteiro auxiliar que está apontando para o antigo início da lista.

ii) **Remoção no Meio:** Após a identificação do elemento que será removido na lista e identificando que o mesmo não está no fim nem no início da lista é necessário fazer que o ponteiro auxiliar ant aponte para o sucessor do outro ponteiro auxiliar, assim mantendo a ligação dos elementos da lista.



iii) **Remoção no Fim:** Para a funcionalidade de remoção no fim da lista basta apenas identificar se o ponteiro auxiliar é igual ao fim da lista e depois fazer o fim da lista receber o ponteiro auxiliar que contém o elemento anterior e depois fazer o fim da lista próximo apontar para o

ponteiro NULL e por fim realizar a desalocação da memória do ponteiro auxiliar. Assim removendo o elemento no fim da lista.

Exemplo de código de remoção em uma lista.

```c
No *p = lista->ini;
No *ant = NULL;
while(p != NULL) {

    if (p->elem == elem) { // verifica se o elemento passado eh igual ao elemento da lista

        if (p == lista->ini) { // verifica se o elemento encontrado esta no inicio da lista

            lista->ini = lista->ini->prox; // o inicio da lista eh atualizado com o proximo elemento
            free(p); // o elemento eh liberado da memoria

        } else if (p == lista->fim) { // verifica se o elemento encontrado esta no fim da lista

            lista->fim = ant; // atualiza o fim da lista para o elemento ant
            lista->fim->prox = NULL; // atualiza o fim da lista prox para NULL
            free(p); // libera o elemento que esta no fim da memoria

        } else { // caso o elemento nao esteja nem no fim e nem no inicio o mesmo se encontra em alguma posicao no meio da lista

            ant->prox = p->prox; // atualiza o ant proximo para receber o proximo elemento da lista, nao quebrando a continuidade da lista
            free(p); // libera o elemento no meio da lista

        }

        lista->tam -= 1; // decrementa o tamanho da lista

        return 1; // retorna 1 indicando sucesso na remocao do elemento
    } else { // caso o elemento nao seja igual sao atualizados os ponteiros auxiliares ant e p
        ant = p;
        p = p->prox;
    }
}

return 0; // retorna 0 indicando falha na remocao do elemento
```

QUESTÃO 4

```c
typedef struct no No;
struct no {
    float elemento;
    No *prox;
};

struct conjunto {
    No *ini, *fim;
};
```

```c
Conjunto *cr_criar() {
    Conjunto *conjunto = (Conjunto *) malloc(sizeof(Conjunto));

    if (conjunto == NULL) {
        return NULL;
    }

    conjunto->ini = NULL;
    conjunto->fim = NULL;

    return conjunto;
}
```

```c
int cr_liberar(Conjunto *conjunto) {
    if (conjunto == NULL) {
        return 0;
    }

    No *p = conjunto->ini;
    while (p != NULL) {
        conjunto->ini = p->prox;
        free(p);
        p = conjunto->ini;
    }

    free(conjunto);
    return 1;
}
```

```c
int cr_inserir(Conjunto *conjunto, float elem) {
    if (conjunto == NULL) {
        return 0;
    }

    if (cr_pertence(conjunto, elem)) {
        return 0;
    }

    No *novoNo = (No *) malloc(sizeof(No));

    if (novoNo == NULL) {
        return 0;
    }

    novoNo->elemento = elem;
    novoNo->prox = NULL;

    if (conjunto->ini == NULL) {
        conjunto->ini = novoNo;
    } else {
        conjunto->fim->prox = novoNo;
    }

    conjunto->fim = novoNo;

    return 1;
}
```

```c
int cr_remover(Conjunto *conjunto, float elem) {
    if (conjunto == NULL) {
        return 0;
    }

    No *p = conjunto->ini;
    No *ant = NULL;
    while(p != NULL) {
        if (p->elemento == elem) {
            if (p == conjunto->ini) {
                conjunto->ini = conjunto->ini->prox;
                free(p);
            } else if (p == conjunto->fim) {
                conjunto->fim = ant;
                conjunto->fim->prox = NULL;
                free(p);
            } else {
                ant->prox = p->prox;
                free(p);
            }

            return 0;
        } else {
            ant = p;
            p = p->prox;
        }
    }

    return 0;
}
```

```c
int cr_imprime(Conjunto *conjunto) {
    if (conjunto == NULL) {
        return 0;
    }

    No *p = conjunto->ini;
    while (p != NULL) {
        printf("%.2f ", p->elemento);
        p = p->prox;
    }

    printf("\n");

    return 1;
}
```

```c
Conjunto *cr_uniao(Conjunto *A, Conjunto *B) {
    if (A == NULL || B == NULL) {
        return NULL;
    }

    Conjunto *uniao = cr_criar();

    if (uniao == NULL) {
        return NULL;
    }

    No *p = NULL;

    p = A->ini;
    while (p != NULL) {
        cr_inserir(uniao, p->elemento);
        p = p->prox;
    }

    p = B->ini;
    while (p != NULL) {
        cr_inserir(uniao, p->elemento);
        p = p->prox;
    }

    return uniao;
}
```

```c
Conjunto *cr_interseccao(Conjunto *A, Conjunto *B) {
    if (A == NULL || B == NULL) {
        return NULL;
    }

    Conjunto *interceccao = cr_criar();

    if (interceccao == NULL) {
        return NULL;
    }

    for (No *i = A->ini; i != NULL; i = i->prox) {
        for (No *j = B->ini; j != NULL; j = j->prox) {
            if (i->elemento == j->elemento) {
                cr_inserir(interceccao, i->elemento);
            }
        }
    }

    return interceccao;
}
```

```c
int cr_pertence(Conjunto *conjunto, float elem) {
    if (conjunto == NULL) {
        return 0;
    }

    No *p = conjunto->ini;
    while (p != NULL) {
        if (p->elemento == elem) {
            return 1;
        }

        p = p->prox;
    }

    return 0;
}
```

```c
int cr_iguais(Conjunto *A, Conjunto *B) {
    if (A == NULL || B == NULL) {
        return -1;
    }

    for (No *i = A->ini; i != NULL; i = i->prox) {
        for (No *j = B->ini; j != NULL; j = j->prox) {
            if (i->elemento != j->elemento) {
                return 0;
            }
        }
    }

    return 1;
}
```

```c
int cr_tamanho(Conjunto *conjunto) {
    if (conjunto == NULL) {
        return -1;
    }

    int cont = 0;
    No *p = conjunto->ini;
    while (p != NULL) {
        cont++;
        p = p->prox;
    }

    return cont;
}
```

```c
int cr_is_vazio(Conjunto *conjunto) {
    if (conjunto == NULL) {
        return 1;
    }

    if (cr_tamanho(conjunto) != 0) {
        return 0;
    }

    return 1;
}
```