Disciplina: Estrutura de dados
Aluna: Laura de Faria Maranhão Ayres
Matrícula: 20180019070

1.
a-) Estrutura de dados: Estruturas são conjuntos de tipos de dados diferentes entre si. Além disso, determinar representações para essas entidades abstratas e implementar essas operações abstratas sobre essas representações concretas.
Exemplo: vetores e listas.

Tipo abstrato de dados: É criada uma estrutura de dados, a qual possui informações e funções atreladas a ela. Nisso, quando utiliza-se desses dados ou métodos, o usuário não se preocupa com a organização interna deles, apenas nos parâmetros enviados e no resultado obtido.

Exemplo:
```
typedef struct no L_no;

L_no * create_no(int info);
int insert_sorted(L_no** lhead, int info);
int l_size(L_no* lhead);
int exists(L_no* lhead, int info);
```

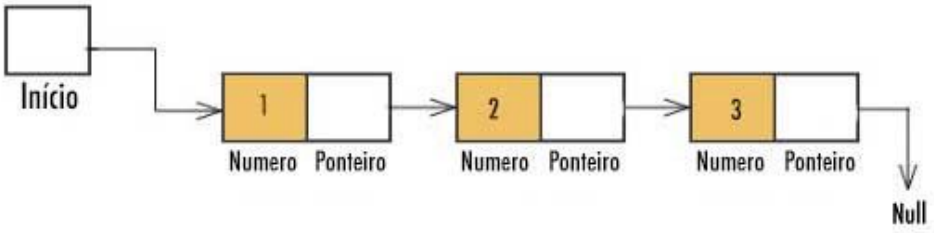
b-) Alocação estática: Os tipos de dados desse tipo de alocação já possuem um espaço reservado na memória, alocando automaticamente (armazenada durante a compilação). Como consequência desse tipo de alocação, há muitas vezes um desperdício de recursos, pois o espaço reservado previamente não é preciso, uma vez que diversas vezes não é possível determinar o espaço necessário.

Exemplo:
```
/*Espaço reservado para 10 valores tipo int(4 bytes) - 4x10 = 40 bytes*/
int array[10];
```

Alocação dinâmica: Aloca espaços (sob demanda) durante a execução do programa. Dessa forma, evita-se o desperdício de recursos, uma vez que a alocação é feita quando necessária, podendo fornecer ou reduzir o espaço de memória alocado. Para realizar esse processo, são utilizadas algumas funções, como: malloc e calloc (Servem para alocar o bloco de memória), realloc (Muda o tamanho do bloco de memória) e free(Libera um bloco alocado de memória).

Exemplo:
```
/*Espaço reservado para 40 bytes*/
array = (int*) malloc(10*sizeof(int));
```

c-) Lista encadeada: Uma lista encadeada é um tipo de estrutura de dados, sendo uma representação de uma sequência de células do mesmo tipo. Cada elemento da sequência é um nó, possuindo dentro de sua estrutura a informação dele e um ponteiro que aponta para o nó seguinte até chegar no último elemento, o qual aponta para NULL, indicando o fim da lista.



Exemplo:

```
struct no {
    int conteudo;
    struct no *prox;
};

typedef struct no Celula;
Celula *lista
```

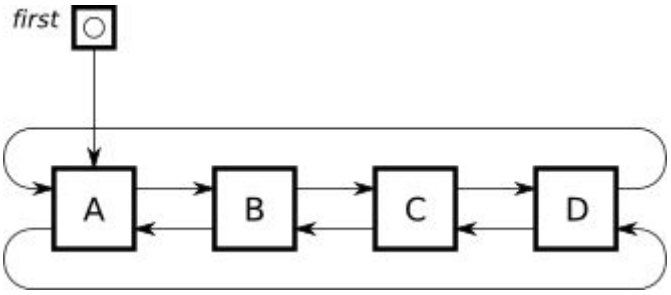
Lista duplamente encadeada: A lista duplamente encadeada segue o mesmo padrão da anterior. Contudo, além de possuir um ponteiro para o nó posterior, ela também possui um ponteiro para o nó anterior, permitindo o percurso pelas duas direções.

Exemplo:

```
struct no {
    int conteudo;
    struct no *prox;
        struct no *ant;
};

typedef struct no Celula;
Celula *lista
```

Lista Circular: Uma lista circular também pode ser construída com encadeamento duplo, porém o último nó possui o ponteiro para o início, formando um ciclo. Assim, a lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista.



Exemplo:

```
struct no {
    int conteudo;
    struct no *prox;
        struct no *ant;
};

typedef struct no Celula;
Celula *lista
```

Lista heterogênea: Na lista heterogênea trabalha da mesma forma que uma lista encadeada. Apesar disso, essa estrutura suporta tipos diferentes de dados, necessitando-se de três campos para sua manipulação: um identificador do tipo do objeto, um ponteiro para a informação e outro para o próximo nó da lista.

Exemplo:

```
struct no {
    int tipo;
    void info;;
    struct no *prox;
};

typedef struct no Celula;
Celula *lista
```

2.

a-)

```
void uniao(L_no* lhead_A, L_no* lhead_B, L_no** lhead_R)      //Realiza a união de
{                                                             //dois conjuntos
        int i;
        L_no *lista_A = lhead_A;
        L_no *lista_B = lhead_B;

        for(i=0; i < l_size(lhead_A); i++){          //Compara cada termo do conjunto (A)
                                                     //com o resultante (R)
                if(!exists(*lhead_R, lista_A->info)){   //Se o termo do conjunto não
                                                        //existir no resultante, ele insere ordenadamente
                        insert_sorted(lhead_R, lista_A->info);
```

```
            }
            lista_A = lista_A->proximo;
        }

        for(i=0; i < l_size(lhead_B); i++){       //Compara cada termo do conjunto (B)
                                                  //com o resultante (R)
            if(!exists(*lhead_R,lista_B->info)){       //Se o termo do conjunto não
                                        //existir no resultante, ele insere ordenadamente
                insert_sorted(lhead_R,lista_B->info);
            }
            lista_B = lista_B->proximo;
        }
}
```

b-) L_no* conjunto_C = NULL; //Inicializa o ponteiro na Main.c, igualando-o a NULL

c-)

```
int insert_sorted(L_no** lhead, int info) //Insere ordenadamente os elementos na lista
{

        L_no* novo = create_no(info);      //Alocação de memória
        L_no* anterior = NULL;            //ponteiro para elemento anterior
        L_no* lista = *lhead;             //ponteiro para percorrer a lista

        /* Verifica se a alocação do nó foi bem sucedida*/
        if(!novo){
            printf("Erro de alocação de memória/n");
            return 1;
        }

         /* Percorre a lista a procura da posição de inserção */
         while (lista != NULL && lista->info < info) {
            anterior = lista;
            lista = lista->proximo;
         }

         /* insere elemento */
         if (anterior == NULL) {              //insere elemento no início
             novo->proximo = *lhead;
             *lhead = novo;
         }
         else{                                  //insere elemento no meio da lista
            novo->proximo = anterior->proximo;
            anterior->proximo = novo;
```

```
        }

        return 0;
}
```

d-)

```
int remove_item(L_no** lhead, int info){       //Remove elemento da lista
        L_no* anterior = NULL;                  //Ponteiro para elemento anterior
        L_no* lista = *lhead;                   //Ponteiro para percorrer a lista

   /* Procura elemento na lista, guardando anterior */
        while (lista != NULL && lista->info != info) {
                anterior = lista;
                lista = lista->proximo;
        }

   /* Verifica se achou elemento */
        if(empty(lista)){
                return 0;
        }

   /* Retira elemento */
        if (anterior == NULL) {                 //Retira o elemento do início
                *lhead = lista->proximo;
        }
        else {                                  //Retira o elemento do meio da lista
                anterior->proximo = lista->proximo;
        }

        free(lista);
        return 0;
}
```

e-)

```
void interseccao(L_no* lhead_A, L_no* lhead_B, L_no** lhead_R)  //Realiza a
{                                                               //intersecção de dois conjuntos
        int i;
        L_no *lista_A = lhead_A;
        L_no *lista_B = lhead_B;

        for(i=0; i < l_size(lhead_B); i++){          //Compara cada termo do conjunto (B)
                                                        //com o resultante (R)
                if(exists(lista_A,lista_B->info)){      //Se o termo do conjunto existir no
                                                      //resultante, ele insere ordenadamente
```

```
                        insert_sorted(lhead_R,lista_B->info);
                }
                lista_B = lista_B->proximo;
        }
}
```

f-)

```
void diferenca(L_no* lhead_A, L_no* lhead_B, L_no** lhead_R)   //Realiza a
{                                                               //diferença entre os dois conjuntos
        L_no* lista_A = lhead_A;
        L_no* lista_B = lhead_B;
        int  i;

        for(i=0; i < l_size(lhead_B); i++){       //Insere no conjunto resultante a
                                                  //diferença A-B
                        insert_sorted(lhead_R,((lista_A->info) - (lista_B->info)));

                lista_B = lista_B->proximo;
                lista_A = lista_A->proximo;
        }
}
```

g-)

```
int exists(L_no* lhead, int info)            //Procura um determinado elemento na lista
{
        L_no* lista = lhead;

        while(lista != NULL){
                if(lista->info == info)
                        return 1;

                lista = lista->proximo;
        }

        return 0;
}
```

h-) conjunto_B->info //Apenas pega o primeiro elemento da lista, uma vez que a
//inserção dos elementos foi feita de forma ordenada (crescente)

i-)

```
int maior(L_no* lhead)                    //Encontra o maior elemento na lista
{
        L_no * lista = lhead;
        int i, valor;

        for(i=0; i < l_size(lhead); i++){         //Percorre a lista até o último elemento e
                                                  //armazena o seu valor
                valor = lista->info;
                lista = lista->proximo;
        }

        return valor;                             //Retorna o último elemento da lista, já que ela
                                                  //está ordenada de forma crescente
}
```

j-)

```
int iguais(L_no* lhead_A, L_no* lhead_B)          //Checa se os conjuntos são iguais
{
        L_no* lista_A = lhead_A;
        L_no* lista_B = lhead_B;
        int tamanho_A = l_size(lhead_A);
        int tamanho_B = l_size(lhead_B);
        int cont = 0, i;

        if(tamanho_A == tamanho_B){

                for(i=0; i < tamanho_A; i++){       //Compara cada termo do conjunto (A)
                                                    //com o outro (B)
                        if(exists(lista_B, lista_A->info)){    //Se o termo do conjunto
                                                               //existir no resultante, ele incrementa o contador
                                cont++;
                        }
                        lista_A = lista_A->proximo;
                }

                if(cont == tamanho_B){       //se o contador for do tamanho do conjunto,
                                             //significa que são iguais
                        return 1;
                }
                else{
                        return 0;
                }
        }
```

```
        else{                   //Se não forem do mesmo tamanho, retorna zero(diferentes)
                return 0;
        }
}
```

k-)

```
int l_size(L_no* lhead)                 //Verifica o tamanho da lista
{
        int cont = 0;

        while(lhead != NULL){       //Percorre a lista, incrementando o contador a
                                    //cada elemento passado
                cont++;
                lhead = lhead->proximo;
        }

        return cont;
}
```

l-)

```
int empty(L_no* lhead)                  //Verifica se a lista está vazia
{
        if(lhead == NULL){              //Se o primeiro termo for nulo, a lista está vazia
                return 1;
        }
        return 0;
}
```