Simulação de Manipulação da Heap

Disciplina: Linguagem de Programação

Docente: Carlos Bazílio Martins

Discente: Júlia Miranda Rodrigues

Algoritmos de Manipulação do Heap

- ✓ First Fit
- ✓ Best Fit
- ✓ Worst Fit

Programa

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAM_HEAP 20

enum algo_heap{First_Fit = 1, Best_Fit, Worst_Fit};

enum tipo_instrucao{set_heap = 1, new, del, exibe, atribui};

enum status_var{ sofreuDel = -1 };

enum status_heap{ livre, alocado, lixo = -1 };

int MARCADOR_HEAP;
int linha = 0;
```

Programa

- ✓ Vetor de instruções

```
typedef struct Instrucao{
    char str[10];
}t_instrucao;
```

Programa

- ✓ Lista de variáveis

```c
typedef struct Variavel{
    char nome[20];
    int qtdMem;
    int blocoInicial;
}t_variavel;


typedef struct ListaDeVariaveis{
    t_variavel info;
    struct ListaDeVariais *prox;
}l_variavel;
```

Programa

- ✓ Lista de áreas livres

```
typedef struct AreaLivre{
    int inicio;
    int qtdBlocosContiguos;
}t_areaLivre;


typedef struct ListaDeAreasLivres{
    t_areaLivre info;
    struct ListaDeAreasLivres *prox;
}l_areaLivre;
```

Programa

- ✓ main()

```c
int main()
{
    int continuar;
    char nomeDoArquivo[20];

    while(1){
        printf("Nome do programa: ");
        fflush(stdin);
        gets(nomeDoArquivo);

        ExecutarPrograma(nomeDoArquivo);

        printf("Executar novo programa?\n<0> N\n<1> S\n>>>");
        scanf("%d", &continuar);

        if(!continuar)
            exit(0);

        linha = 0;
    }
    return (0);
}
```

Programa

- ✓ Execução do programa(ExecutarPrograma())
  - main()

```
void ExecutarPrograma(char *nomeDoPrograma)
{
    FILE *arq = fopen(nomeDoPrograma, "r");
    int heap[TAM_HEAP] = {livre};

    if (arq != NULL)
    {
        l_variavel *listaVar = InicializarListaVariavel();
        l_areaLivre *listaAre = InicializarListaAreaLivre();

        char *palavra;
        char instrucao[30];

        int tipoDeInstrucao;
        int indice;

        t_instrucao vet_instrucao[3];
```

Programa

- ✓ Execução do programa(ExecutarPrograma())
  - main()

```
while (fscanf(arq, " %[^\n]s ", instrucao) != EOF)
{
    linha++;
    indice = 0;

    printf("\n%d| %s\n", linha, instrucao);

    for (palavra = strtok(instrucao, " "); palavra != NULL; palavra = strtok(NULL, " "))
    {
        strcpy(vet_instrucao[indice].str, palavra);
        indice++;
    }

    tipoDeInstrucao = VerificarInstrucao(vet_instrucao);
```

Programa

- ✓ Execução do programa(ExecutarPrograma())
  - main()

```
switch (tipoDeInstrucao)
{
case set_heap:
    ConfigurarHeap(vet_instrucao);

    break;
case new:
    Declarar(heap, &listaVar, &listaAre, vet_instrucao);

    break;
case del:
    Deletar(heap, &listaVar, &listaAre, vet_instrucao[1].str);

    break;
```

Programa

- ✓ Execução do programa(ExecutarPrograma())
  - main()

```
            case exibe:
                Exibe(listaVar, listaAre, heap);
                break;
            case atribui:
                Atribuir(&listaVar, &listaAre, vet_instrucao, heap);
                break;
            default:
                break;
            }
            system("pause");
            system("cls");
        }
        listaAre = DestruirListaDeAreasLivres(listaAre);
        listaVar = DestruirListaDeVariaveis(listaVar);
    }
    fclose(arq);
}
```

Programa

- ✓ Inicialização de lista de variáveis
  - Execução do programa (ExecutarPrograma())

```
l_variavel *InicializarListaVariavel()
{
    return NULL;
}
```

Programa

- ✓ Inicialização de lista de áreas livres
  - Execução do programa(ExecutarPrograma())

```
l_areaLivre *InicializarListaAreaLivre()
{
    l_areaLivre *areaLivreInicial = NovoElementoAreaLivre(0, TAM_HEAP);

    return areaLivreInicial;
}
```

Programa

- ✓ Criação de novo elemento para lista de áreas livres
  - Inicialização da lista (InicializarListaArealivre())

```
l_areaLivre *NovoElementoAreaLivre(int inicioDeAreaLivre, int qtdBlocosContiguos)
{
    l_areaLivre *novoElemento = (l_areaLivre *)malloc(sizeof(l_areaLivre));

    novoElemento->info.inicio = inicioDeAreaLivre;

    novoElemento->info.qtdBlocosContiguos = qtdBlocosContiguos;

    novoElemento->prox = NULL;

    return novoElemento;
}
```

Programa

- ✓ Verificação de instrução
  - Execução do programa(ExecutarPrograma())

```
int VerificarInstrucao(t_instrucao *vet_instrucao)
{
    if (strcmp(vet_instrucao[0].str, "exibe") && !strcmp(vet_instrucao[1].str, "="))
        return atribui;
    else
    {
        if (!strcmp(vet_instrucao[0].str, "new"))
            return new;

        else if (!strcmp(vet_instrucao[0].str, "heap"))
            return set_heap;

        else if (!strcmp(vet_instrucao[0].str, "del"))
            return del;

        else if (!strcmp(vet_instrucao[0].str, "exibe"))
            return exibe;
        else
            return -1;
    }
}
```

Programa

- ✓ Configurar marcador do algoritmo de manipulação do heap
  - Execução do programa(ExecutarPrograma())

```
void ConfigurarHeap(t_instrucao *instrucao)
{

    if (!strcmp(instrucao[1].str, "first"))
        MARCADOR_HEAP = First_Fit;

    else if (!strcmp(instrucao[1].str, "best"))
        MARCADOR_HEAP = Best_Fit;

    else if (!strcmp(instrucao[1].str, "worst"))
        MARCADOR_HEAP = Worst_Fit;
}
```

Programa

- ✓ Declaração de variável
  - Execução do programa(ExecutarPrograma())

```
void Declarar(int *heap, l_variavel **listaVar, l_areaLivre **listaAreaLivre, t_instrucao *instrucao)
{
    int n_inicioAlocado;
    int resp = 0;
    int mem = atoi(instrucao[2].str);

    if ((*listaAreaLivre) == NULL)
    {

        printf("\n<AVISO> Heap cheia\n");
    }
```

Programa

- ✓ Declaração de variável
  - Execução do programa(ExecutarPrograma())

```
else
{
    switch (MARCADOR_HEAP)
    {
        case First_Fit:
            FirstFit(&resp, listaAreaLivre, &n_inicioAlocado, mem);
            break;

        case Best_Fit:
            BestFit(&resp, listaAreaLivre, &n_inicioAlocado, mem);
            break;

        case Worst_Fit:
            WorstFit(&resp, listaAreaLivre, &n_inicioAlocado, mem);
            break;
    }
```

Programa

- ✓ Declaração de variável
  - Execução do programa(ExecutarPrograma())

```
        if (!resp)
        {
            InserirListaDeVar(listaVar, n_inicioAlocado, mem, instrucao[1].str);

            AtualizarHeap(heap, n_inicioAlocado, mem, alocado);
        }
    }
}
```

Programa

- ✓ Criação de novo elemento para lista de variáveis
  - Inserção (InserirListaDeVar())

```
l_variavel *NovoElementoVariavel(char *nome, int blocoInicial, int qtdBlocosContiguos)
{
    l_variavel *novaVar = (l_variavel *)malloc(sizeof(l_variavel));

    strcpy(novaVar->info.nome, nome);

    novaVar->info.blocoInicial = blocoInicial;
    novaVar->info.qtdMem = qtdBlocosContiguos;
    novaVar->prox = NULL;

    return novaVar;
}
```

Programa

- ✓ Inserção de elemento na lista de variáveis
  - Atribuição (Atribuir())
  - Declaração(Declarar())

```
void InserirListaDeVar(l_variavel **listaVar,int  n_inicioAlocado,int  qtdBlocosContiguos, char *nome){
    int resp = Existe(listaVar, nome, n_inicioAlocado, qtdBlocosContiguos);

    if(resp == 1)
    {
        l_variavel* novaVar = NovoElementoVariavel(nome, n_inicioAlocado, qtdBlocosContiguos);
        l_variavel* aux = (*listaVar);
        l_variavel* ant = NULL;

        if((*listaVar) == NULL)
            (*listaVar) = novaVar;
        else{
            while(aux != NULL)
            {
                ant = aux;
                aux = aux->prox;
            }
            ant->prox = novaVar;
        }
    }
}
```

Programa

- ✓ Verificação se variável existe
  - Inserção na lista de variáveis (InserirListaDeVar())

```
int Existe(l_variavel **listaVar, char *nome, int n_inicioAlocado, int qtdBlocosContiguos)
{
    l_variavel *aux = (*listaVar);

    if ((*listaVar) == NULL)
        return 1;
    else
    {
        while (aux != NULL)
        {
            if (!strcmp(aux->info.nome, nome))
            {
                aux->info.blocoInicial = n_inicioAlocado;
                aux->info.qtdMem = qtdBlocosContiguos;
                return 0;
            }
            aux = aux->prox;
        }
        return 1;
    }
}
```

Programa

- ✓ Atualização do estado do heap
  - Atribuição (Atribuir())
  - Declaração (Declarar())
  - Desalocar área (Deletar())

```
void AtualizarHeap(int *heap, int indiceInicial, int indiceFinal, int status)
{
    int max = indiceInicial + indiceFinal;

    for (int i = indiceInicial; i < max; i++)
    {
        heap[i] = status;
    }
}
```

Programa

- ✓ Algoritmo do First Fit
  - Declaração de variável (Declarar())

```
void FirstFit(int *resp, l_areaLivre **lista, int *n_inicioAlocado, int qtdTotal)
{
    l_areaLivre *aux = (*lista);
    l_areaLivre *ant = NULL;

    int sobra;
    int cpy_inicio;

    while (aux != NULL)
    {
        sobra = aux->info.qtdBlocosContiguos - qtdTotal;

        if (sobra == 0)
        {
            (*n_inicioAlocado) = aux->info.inicio;

            (*resp) = 0;
```

Programa

- ✓ Algoritmo do First Fit
  - Declaração de variável (Declarar())

```
        if ((*lista)->prox == NULL)
        {
            free(*lista);
            (*lista) = NULL;
        }
        else
        {
            if (ant != NULL)
                ant->prox = aux->prox;
            else
                (*lista) = aux->prox;
            free(aux);
        }
        break;
    }
```

Programa

- ✓ Algoritmo do First Fit
  - Declaração de variável (Declarar())

```
        else if (sobra > 0)
        {
            (*n_inicioAlocado) = aux->info.inicio;
            (*resp) = 0;

            ParticionarNo(&aux, qtdTotal);

            break;
        }
        ant = aux;
        aux = aux->prox;
    }

    if (aux == NULL)
    {
        printf("\n<AVISO> Sem espaco para alocar essa quantidade de memo'ria\n");

        (*resp) = 1;
    }
}
```

Programa

- ✓ Algoritmo do Best Fit
  - Declaração de variável (Declarar())

```
void BestFit(int *resp, l_areaLivre **lista, int *n_inicioAlocado, int qtdTotal)
{
    int tmp_sobra;
    l_areaLivre *aux = (*lista);
    l_areaLivre *ant = NULL;
    l_areaLivre *p_menorSobra = NULL;
    int menorSobra = TAM_HEAP;

    while (aux != NULL)
    {
        tmp_sobra = aux->info.qtdBlocosContiguos - qtdTotal;

        if (tmp_sobra == 0)
        {
            p_menorSobra = aux;

            (*n_inicioAlocado) = aux->info.inicio;
```

Programa

- ✓ Algoritmo do Best Fit
  - Declaração de variável (Declarar())

```
        if ((*lista)->prox == NULL)
        {
            free(*lista);
            (*lista) = NULL;
        }
        else
        {
            if (ant != NULL)
                ant->prox = aux->prox;
            else
                (*lista) = aux->prox;
        }
        break;
    }
```

Programa

- ✓ Algoritmo do Best Fit
  - Declaração de variável (Declarar())

```
    else if (tmp_sobra > 0 && tmp_sobra < menorSobra)
    {
        p_menorSobra = aux;

        if ((*lista)->prox == NULL)
        {
            (*n_inicioAlocado) = (*lista)->info.inicio;
            (*resp) = 0;
            break;
        }

        menorSobra = tmp_sobra;
    }
    ant = aux;
    aux = aux->prox;
}
```

Programa

- ✓ Algoritmo do Best Fit
  - Declaração de variável (Declarar())

```
    if (p_menorSobra == NULL || tmp_sobra < 0)
    {
        printf("\n<AVISO> Sem espaco para alocar essa quantidade de memo'ria\n");
        (*resp) = 1;
    }
    else if (tmp_sobra != 0)
    {
        (*n_inicioAlocado) = p_menorSobra->info.inicio;

        ParticionarNo(&p_menorSobra, qtdTotal);

        (*resp) = 0;
    }
}
```

Programa

- ✓ Algoritmo do Worst Fit
  - Declaração de variável (Declarar())

```
void WorstFit(int *resp, l_areaLivre **lista, int *n_inicioAlocado, int qtdTotal)
{
    int tmp_sobra;
    l_areaLivre *aux = (*lista);
    l_areaLivre *ant = NULL;
    l_areaLivre *p_maiorSobra = NULL;
    int maiorSobra = 0;

    while (aux != NULL)
    {
        tmp_sobra = aux->info.qtdBlocosContiguos - qtdTotal;

        if (tmp_sobra == 0)
        {
            p_maiorSobra = aux;

            (*n_inicioAlocado) = aux->info.inicio;
```

Programa

- ✓ Algoritmo do Worst Fit
  - Declaração de variável (Declarar())

```
            if ((*lista)->prox == NULL)
            {
                free(*lista);
                (*lista) = NULL;
            }
            else
            {
                if (ant != NULL)
                    ant->prox = aux->prox;
                else
                    (*lista) = aux->prox;
            }
            break;
        }
```

Programa

- ✓ Algoritmo do Worst Fit
  - Declaração de variável (Declarar())

```
        else if (tmp_sobra > 0 && tmp_sobra > maiorSobra)
        {
            p_maiorSobra = aux;

            if ((*lista)->prox == NULL)
            {
                (*n_inicioAlocado) = (*lista)->info.inicio;
                (*resp) = 0;

                break;
            }

            maiorSobra = tmp_sobra;
        }
        ant = aux;
        aux = aux->prox;
    }
```

Programa

- ✓ Algoritmo do Worst Fit
  - Declaração de variável (Declarar())

```
    if (p_maiorSobra == NULL || tmp_sobra < 0)
    {
        printf("\n<AVISO> Sem espaco para alocar essa quantidade de memo'ria\n");
        (*resp) = 1;
    }
    else if (tmp_sobra != 0)
    {
        (*n_inicioAlocado) = p_maiorSobra->info.inicio;

        ParticionarNo(&p_maiorSobra, qtdTotal);

        (*resp) = 0;
    }
}
```

Programa

- ✓ Alocar parte de área livre
  - Alocações com algoritmos First, Best e Worst Fit (First(), BestFit(), WorstFit())

```
void ParticionarNo(l_areaLivre **area, int qtdTotal)
{
    int inicioAntigo = (*area)->info.inicio;
    int qtdTotalAntiga = (*area)->info.qtdBlocosContiguos;

    (*area)->info.inicio = inicioAntigo + qtdTotal;
    (*area)->info.qtdBlocosContiguos = qtdTotalAntiga - qtdTotal;
}
```

Programa

- ✓ Desalocar área
  - Execução do programa(ExecutarPrograma())

```
void Deletar(int *heap, l_variavel **listaVar, l_areaLivre **listaArea, char *variavel)
{

    int n_blocoLivre, n_blocosContiguosLivres;

    l_variavel *auxVar = (*listaVar);

    while (auxVar != NULL)
    {
        if (!strcmp(auxVar->info.nome, variavel))
            break;

        auxVar = auxVar->prox;
    }
    if (auxVar == NULL)
        printf("Varia'vel inexistente\n");
```

Programa

- ✓ Desalocar área
  - Execução do programa(ExecutarPrograma())

```
    else
    {

        n_blocoLivre = auxVar->info.blocoInicial;
        n_blocosContiguosLivres = auxVar->info.qtdMem;

        auxVar->info.blocoInicial = sofreuDel;

        VerificarOutrasVariaveis(listaVar, n_blocoLivre);

        LiberarListaDeAreasLivres(listaArea, n_blocoLivre, n_blocosContiguosLivres);

        AtualizarHeap(heap, n_blocoLivre, n_blocosContiguosLivres, livre);
    }
}
```

Programa

- ✓ Verificação de variáveis que referenciam a mesma área
  - Desalocar área (Deletar())

```
void VerificarOutrasVariaveis(l_variavel **lista, int inicioBlocoLivre)
{
    l_variavel *auxVar = (*lista);

    while (auxVar != NULL)
    {
        if (auxVar->info.blocoInicial == inicioBlocoLivre)
        {

            auxVar->info.blocoInicial = sofreuDel;
            MostrarOutrasVariaveis(auxVar->info.nome, inicioBlocoLivre);
        }
        auxVar = auxVar->prox;
    }
}
```

Programa

- ✓ Mostrar outras variáveis que referenciam área
  - Verificação de variáveis que referenciam a mesma área (VerificarOutrasVariaveis())

```
void MostrarOutrasVariaveis(char *nome, int inicioBlocoLivre)
{
    printf("\n<AVISO> %s tambe'm referenciava o bloco %d\n", nome, inicioBlocoLivre);
}
```

Programa

- ✓ Liberação de áreas livres
  - Desalocar área (Deletar())

```
void LiberarListaDeAreasLivres(l_areaLivre **listaArea, int n_blocoLivre, int n_blocosContiguosLivres)
{
    l_areaLivre *aux = (*listaArea);

    int indiceAnteriorAoPrimeiroBloco = n_blocoLivre - 1;
    int indicePosteriorAoUltimoBloco = (n_blocoLivre + (n_blocosContiguosLivres - 1)) + 1;
    int tmp_inicio, tmp_final;

    while (aux != NULL)
    {
        tmp_final = aux->info.inicio + (aux->info.qtdBlocosContiguos - 1);
        tmp_inicio = aux->info.inicio;

        if (tmp_final == indiceAnteriorAoPrimeiroBloco)
        {

            aux->info.qtdBlocosContiguos = aux->info.qtdBlocosContiguos + n_blocosContiguosLivres;
```

Programa

- ✓ Liberação de áreas livres
  - Desalocar área (Deletar())

```
            if (aux->prox->info.inicio == indicePosteriorAoUltimoBloco)
            {
                aux->info.qtdBlocosContiguos = aux->info.qtdBlocosContiguos + aux->prox->info.qtdBlocosContiguos;
                break;
            }
        }
        else if (tmp_inicio == indicePosteriorAoUltimoBloco)
        {
            aux->info.inicio = n_blocoLivre;
            aux->info.qtdBlocosContiguos = aux->info.qtdBlocosContiguos + n_blocosContiguosLivres;
            break;
        }

        aux = aux->prox;
    }
    if (aux == NULL)
    {
        InserirAreaLivre(listaArea, n_blocoLivre, n_blocosContiguosLivres);
    }
}
```

Programa

- ✓ Inserção de área livre
  - Liberação da área livre (LiberarListaDeAreasLivres())

```
void InserirAreaLivre(l_areaLivre **listaAre, int inicio, int qtdBlocos)
{
    l_areaLivre *nova = NovoElementoAreaLivre(inicio, qtdBlocos);
    l_areaLivre *aux = (*listaAre);
    l_areaLivre *ant = NULL;

    if ((*listaAre) != NULL)
    {
        while (aux != NULL)
        {
            if (aux->info.inicio > inicio)
                break;
            ant = aux;
            aux = aux->prox;
        }
```

Programa

- ✓ Inserção de área livre
  - Liberação da área livre (LiberarListaDeAreasLivres())

```
        if (ant == NULL)
        {
            nova->prox = (*listaAre);

            (*listaAre) = nova;
        }
        else
        {
            ant->prox = nova;
            nova->prox = aux;
        }
    }

    (*listaAre) = nova;
}
```

Programa

- ✓ Atribuição
  - Execução de programa (ExecutarPrograma())

```
void Atribuir(l_variavel **listaVar, l_areaLivre **listaAreaLivre, t_instrucao *instrucao, int *heap)
{
    int e_varInicioDeBloco, d_varInicioDeBloco, e_varBlocosContiguos, d_varBlocosContiguos;
    int resp;

    resp = ProcuraVariavel(listaVar, instrucao[0].str, &e_varInicioDeBloco, &e_varBlocosContiguos);

    ProcuraVariavel(listaVar, instrucao[2].str, &d_varInicioDeBloco, &d_varBlocosContiguos);

    if (!resp)
    {
        if (e_varInicioDeBloco != d_varInicioDeBloco)
        {
            Atribuir_aux(listaVar, instrucao[0].str, d_varInicioDeBloco, d_varBlocosContiguos);

            if (e_varInicioDeBloco != sofreuDel)
                AtualizarHeap(heap, e_varInicioDeBloco, e_varBlocosContiguos, lixo);
        }
    }
    else
        InserirListaDeVar(listaVar, d_varInicioDeBloco, d_varBlocosContiguos, instrucao[0].str);
}
```

Programa

- ✓ Procura por variável
  - Atribuição (Atribuir()) - 2 vezes

```
int ProcuraVariavel(l_variavel **lista, char *var, int *inicioDeBloco, int *blocosContiguos)
{
    l_variavel *aux = (*lista);

    while (aux != NULL)
    {
        if (!strcmp(aux->info.nome, var))
        {
            (*inicioDeBloco) = aux->info.blocoInicial;
            (*blocosContiguos) = aux->info.qtdMem;
            return 0;
        }
        aux = aux->prox;
    }

    return 1;
}
```

Programa

- ✓ Atribuição (auxiliar)
  - Atribuição (Atribuir())

```
void Atribuir_aux(l_variavel **lista, char *e_var, int inicioDeBloco, int blocosContiguos)
{
    l_variavel *aux = (*lista);

    while (aux != NULL)
    {
        if (!strcmp(aux->info.nome, e_var))
        {
            aux->info.blocoInicial = inicioDeBloco;
            aux->info.qtdMem = blocosContiguos;
        }

        aux = aux->prox;
    }
}
```

Programa

- ✓ Exibição do heap, lista de áreas livres e de variáveis
  - Execução do programa(ExecutarPrograma())

```
void Exibe(l_variavel *listaVar, l_areaLivre *listaArea, int *heap)
{
    printf("\n\n> HEAP\n\n");
    MostrarHeap(heap);
    printf("\n\n> VARIA'VEIS \n\n");
    MostrarVariaveis(listaVar);

    printf("\n\n> A'REAS LIVRES \n\n");
    MostrarAreasLivres(listaArea);
}
```

Programa

- ✓ Mostrar estado do heap
  - Exibição do heap, áreas livres e de variáveis(Exibir())

```
void MostrarHeap(int *heap)
{
    for (int i = 0; i < TAM_HEAP; i++)
    {
        switch (heap[i])
        {
            case livre:
                if (i == 0)
                    printf("|   |");
                else
                    printf("   |");
                break;

            case alocado:
                if (i == 0)
                    printf("| * |");
                else
                    printf(" * |");
                break;
```

Programa

- ✓ Mostrar estado do heap
  - Exibição do heap, áreas livres e de variáveis(Exibir())

```
            case alocado:
                if (i == 0)
                    printf("| * |");
                else
                    printf(" * |");
                break;

            case lixo:
                if (i == 0)
                    printf("|lixo|");
                else
                    printf("lixo|");
                break;

            default:
                break;
        }
    }
    printf("\n");
}
```

Programa

✓ Mostrar variáveis do programa

- Exibição do heap, áreas livres e de variáveis(Exibir())

```
void MostrarVariaveis(l_variavel *lista)
{
    if (lista == NULL)
    {
        printf("O programa ainda nao possui variaveis.\n");
    }
    else
    {
        l_variavel *aux = lista;

        while (aux != NULL)
        {
            if (aux->info.blocoInicial == -1)
                printf("%s -  nao referencia a'rea no heap\n", aux->info.nome);
            else
                printf("%s -  Blocos alocados: %d   --  Inicio: %d\n", aux->info.nome, aux->info.qtdMem, aux->info.blocoInicial);
            printf("_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  _ _ \n");

            aux = aux->prox;
        }
    }
}
```

Programa

- ✓ Mostrar áreas livres no heap
  - Exibição do heap, áreas livres e de variáveis(Exibir())

```
void MostrarAreasLivres(l_areaLivre *lista)
{
    l_areaLivre *aux = lista;

    while (aux != NULL)
    {
        printf("Inicio: %d   --  Blocos livres contiguos: %d\n", aux->info.inicio, aux->info.qtdBlocosContiguos);
        printf("_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ \n");

        aux = aux->prox;
    }
}
```