Trabalho 1 - Estrutura de Dados

Gustavo Carvalho, Tomás Rosemberg

Maio 2017

1 Introdução

Propõe-se o desenvolvimento de um sistemas de alocação estudantil para a Diretoria de Desenvolvimento Social (DDS). Atualmente, a distribuição de alunos no programa Moradia Estudantil da Graduação é realizada à mão pelos servidores da DDS. Tal trabalho mostra-se enfadonho e demorado, causando desconforto tanto para os servidores, como para os alunos. Para otimizar esse programa, que na Universidade de Brasília oferece condições de moradia a novos estudantes, desenvolveu-se o sistema a ser descrito neste trabalho.

2 Módulos

O programa desenvolvido é dividido em três partes. Um módulo para leitura dos dados de entrada, um contendo o Tipo Abstrato de Dados desenvolvido para listar alunos e quartos e um programa principal que engloba todas as implementações. A seguir, descreve-se em detalhes cada uma das partes do projeto.

2.1 Tipo Abstrato de Dados

Para organizar os estudantes alocados e não alocados, bem como os quartos disponíveis e ocupados, desenvolveu-se dois tipos de lista. A primeira, é destinada aos estudantes, a segunda, aos quartos. As listas foram implementadas com alocação dinâmica de memória, sendo listas encadeadas. O conteúdo da célula de cada lista é descrito na Tabela 1 e a discriminação destes é dada na Tabela 2.

Table 1: Conteúdo de cada tipo de Célula

Celula Aluno	Chave	Semestre	Exigência	Oferecido
Celula Quarto	Chave	Nota	Ocupado	

Inicialmente, os estudantes serão todos organizados em uma única lista. A medida que são alocados em um quarto, são transferidos à uma segunda lista

Table 2: Discriminação do conteúdo das células

Chave	Identificação do Aluno/Quarto
Semestre	Número de semestres a serem cursados
Nota	Avaliação da qualidade do quarto
Exigência	Nota mínima exigida pelo estudante
Oferecido	*Flag* que indica se um quarto já foi oferecido ao estudante no presente semestre
Ocupado	Aponta para célula do estudante que o ocupa ou para NULL, caso esteja vazio

de estudantes com dormitório. O mesmo é valido para os quartos. Inicialmente, todos encontram-se na lista de quartos vazios. Ao serem destinados a um aluno, são movidos à lista de quartos ocupados. Deste modo, tem-se quatro listas, duas do tipo Aluno e duas do tipo Quarto.

Para trabalhar com essas listas, desenvolveu-se diversas funções. Cada uma delas é descrita a seguir com sua respectiva análise de ordem.

2.1.1 FLAluno e FLQuarto

Funções que criam listas vazias do tipo Aluno e Quarto, respectivamente. Cria-se a célula cabeça, tornando-a a primeira e ultima célula da lista.

```
void FLAluno(TipoListaAluno *Lista){
  Lista->Primeiro = (ApontadorCelulaAluno) malloc(sizeof(CelulaAluno));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}

void FLQuarto(TipoListaQuarto *Lista){
  Lista->Primeiro = (ApontadorCelulaQuarto) malloc(sizeof(CelulaQuarto));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}
```

Trivialmente, tem-se complexidade $\mathcal{O}(1)$.

2.1.2 InsereAluno e InsereQuarto

Funções que inserem novos alunos nas listas do tipo Aluno e Quarto, respectivamente. Aloca-se dinamicamente um novo espaço de memória, e incluí-o na última posição da lista. Atualiza-se a informação da célula Aluno/Quarto com os dados correspondentes.

```
void InsereAluno(TipoAluno Aluno, TipoListaAluno *Lista){
  Lista->Ultimo->Prox = (ApontadorCelulaAluno) malloc(sizeof(CelulaAluno));
  Lista->Ultimo = Lista->Ultimo->Prox;
  Lista->Ultimo->Aluno = Aluno;
  Lista->Ultimo->Prox = NULL;
}

void InsereQuarto(TipoQuarto Quarto, TipoListaQuarto *Lista){
  Lista->Ultimo->Prox = (ApontadorCelulaQuarto) malloc(sizeof(CelulaQuarto));
  Lista->Ultimo = Lista->Ultimo->Prox;
  Lista->Ultimo->Quarto = Quarto;
  Lista->Ultimo->Prox = NULL;
}
```

Trivialmente, tem-se complexidade $\mathcal{O}(1)$.

2.1.3 VaziaAluno e VaziaQuarto

Funções que verificam se a lista encontra-se vazia. Retorna 1 caso esteja vazia e 0 caso contrário. O faz comparando se o último elemento é igual ao primeiro, ou seja, verifica se há apenas a célula cabeça.

```
int VaziaAluno(TipoListaAluno Lista){
  return (Lista.Primeiro == Lista.Ultimo);
}

int VaziaQuarto(TipoListaQuarto Lista){
  return (Lista.Primeiro == Lista.Ultimo);
}
```

Trivialmente, tem-se complexidade $\mathcal{O}(1)$.

2.1.4 ImprimeAluno e ImprimeQuarto

Funções que imprimem todos os alunos da lista e estado do quarto, respectivamente. A primeira é essencial para apresentar a ordem que os alunos foram alocados. A segunda apresenta o número do quarto, juntamente com sua nota e ocupação.

```
void ImprimeAluno (TipoListaAluno Lista){
  ApontadorCelulaAluno aux;
  aux = Lista.Primeiro->Prox;
  if(aux==NULL){
    return;
  }
  while(aux!=NULL){
    printf("%ld\n",aux->Aluno.Chave);
    aux = aux->Prox;
  }
}

void ImprimeQuarto (TipoListaQuarto Lista){
  ApontadorCelulaQuarto aux;
  aux = Lista.Primeiro->Prox;
    if(aux==NULL){
    printf("Lista Vazia\n");
  return;
    }
  while(aux!=NULL){
    printf("O Quarto numero:%ld e estado:%d\n",aux->Quarto.Chave,aux->Quarto.Nota);
    if((aux->Quarto.Ocupado)==NULL){
      printf("Esta desocupado\n");
    }
    else{
      printf("Esta ocupado\n");
    }
    aux = aux->Prox;
  }
}
```

Tem-se complexidade $\mathcal{O}(M)$ para ImprimeAluno e $\mathcal{O}(N)$ para ImprimeQuarto, no pior dos casos. De fato, utilizou-se apenas a função ImprimeAluno para identificar a ordem que os alunos foram alocados. Como esta chamada é efetuada quando a lista de alunos alocados está cheia, tem-se que a complexidade efetiva é $\mathcal{O}(M)$, onde M é o número de alunos.

2.1.5 AlocaAlunoQuarto

Uma das mais importantes funções implementadas. É responsável pela alocação dos alunos nos apartamentos. Conforme as regras do DDS, primeiro um quarto é oferecido a todos os alunos para depois ir ao próximo quarto da lista. Caso um aluno aceite o quarto, este aluno é transferido de lista. O quarto ocupado é atualizado com as informações deste aluno e transferido de lista.

```
void AlocaAlunoQuarto(TipoListaAluno *AlunoNAloc,TipoListaQuarto *ListaQuartoVazio, TipoListaAluno *ListaAlunoAloc, TipoListaQuarto *ListaQuartoOcupado){
  ApontadorCelulaAluno AuxAluno1, AuxAluno2;    // Inicializa ponteiro para alunos
  ApontadorCelulaQuarto AuxQuarto1, AuxQuarto2; // Inicializa ponteiro para quartos
  AuxAluno1 = AlunoNAloc->Primeiro;             // Celula cabeca da lista de alunos nao alocados
  AuxAluno2 = AuxAluno1->Prox;                  // Primeiro aluno da lista de alunos nao alocados
  AuxQuarto1 = ListaQuartoVazio->Primeiro;      // Celula cabeca da lista de quartos vazios
  AuxQuarto2 = AuxQuarto1->Prox;                // Primeiro quarto da lista de quartos vazios
  if(AuxAluno2==NULL){ // Caso em que a lista de alunos nao alocados ficou vazia:
    return;           // Finalizar
  }
  while((AuxQuarto2)!=NULL){    // Enquanto nao chegar ao ultimo quarto da lista, continuar
    if(AuxAluno2!=NULL){              // Se ha um quarto valido e um aluno valido:
      AuxAluno2->Aluno.Oferecido = 1;   // Flag que indica que um quarto foi oferecido ao aluno
    }
    if((AuxQuarto2->Quarto.Nota)>=AuxAluno2->Aluno.Exigencia){ // Se aluno tiver exigencia menor q o estado do quarto, aloque-o
      // O quarto a ser ocupado recebe as informacoes do aluno que o ocupara
      AuxQuarto2->Quarto.Ocupado = &AuxAluno2->Aluno;
      // Transfere o Aluno da lista de alunos nao alocados para a lista de alunos alocados
      AuxAluno1->Prox = AuxAluno2->Prox;                    // Elimina o aluno alocado da lista de alunos nao alocados
      ListaAlunoAloc->Ultimo->Prox = AuxAluno2;             // Transfere o aluno alocado para a lista de alunos alocados
      ListaAlunoAloc->Ultimo = ListaAlunoAloc->Ultimo->Prox;  // Atualiza o ultimo aluno da lista de alunos alocados
      ListaAlunoAloc->Ultimo->Prox = NULL;                  // "Aterra" o final da lista de alunos alocados
      if (AuxAluno1->Prox == NULL) {      // Caso seja o ultimo aluno:
        AlunoNAloc->Ultimo = AuxAluno1;   // Atualiza o ultimo aluno da lista de alunos nao alocados
      }
      if (VaziaAluno(*AlunoNAloc)){       // Se nao ha mais alunos para alocar, finalizar
        return;                           // Finalizar
      }
      // Transfere o Quarto da lista de quartos vazios para a lista de quartos ocupados
      AuxQuarto1->Prox = AuxQuarto2->Prox;                  // Elimina o quarto ocupado da lista de quartos vazios
      ListaQuartoOcupado->Ultimo->Prox = AuxQuarto2;        // Transfere o quarto ocupado para a lista de quartos ocupados
      ListaQuartoOcupado->Ultimo = ListaQuartoOcupado->Ultimo->Prox;  // Atualiza o ultimo quarto da lista de ocupados
      ListaQuartoOcupado->Ultimo->Prox = NULL;              // "Aterra" o final da lista de quartos ocupados
      if (AuxQuarto1->Prox == NULL) {     // Caso seja o ultimo quarto:
        ListaQuartoVazio->Ultimo = AuxQuarto1;  // Atualiza o ultimo quarto da lista de desocupados
      }
      // Move os ponteiros para continuar a tentativa de alocar alunos
      AuxAluno1 = AlunoNAloc->Primeiro;         // Retorna a primeira celula da lista de nao alocados
      AuxAluno2 = AuxAluno1->Prox;              // Retorna ao primeiro aluno da lista de nao alocados
      AuxQuarto2 = AuxQuarto1->Prox;            // Avanca para o proximo quarto disponivel da lista
    } else if((AuxAluno2->Prox)==NULL){         // Se chegar ao ultimo aluno da lista, volta ao primeiro e avanca um quarto
      AuxAluno1 = AlunoNAloc->Primeiro;         // Celula cabeca da lista de alunos nao alocados
      AuxAluno2 = AuxAluno1->Prox;              // Primeiro aluno da lista de alunos nao alocados
      AuxQuarto1 = AuxQuarto2;                  // Avanca um quarto
      AuxQuarto2 = AuxQuarto2->Prox;            // Anda um quarto, pois como auxquarto2!=NULL ainda existe auxquarto2->prox
    }
    else{                                       // Caso seja uma varredura falha padrao, avanca para o proximo aluno
      AuxAluno1 = AuxAluno2;                    // Avanca um aluno
      AuxAluno2 = AuxAluno2->Prox;              // Avanca um aluno, pois como auxaluno2!=NULL ainda existe auxaluno2->prox
    }
  }
}
```

Tem-se complexidade, no pior dos casos, $\mathcal{O}(M \times N)$. Tem-se este caso ao oferecer todos os quartos a todos os alunos.

2.1.6 FimDoSemestre

Função responsável por decrementar o Semestre dos Alunos alocados e decrementar em 5 a Exigência dos alunos da lista de alunos não alocados (caso ao menos um quarto o tenha sido oferecido). Essa chamada é realizada a cada final de semestre.

```
void FimDoSemestre(TipoListaAluno *AlunoNAloc,TipoListaAluno *AlunoAloc,TipoListaQuarto *ListaQuartoVazio){
  ApontadorCelulaAluno FimSemestreANA,FimSemestreAA; // Fim do semestre para os alunos nao alocados e para alunos alocado, respectivamente.
  FimSemestreANA = AlunoNAloc->Primeiro->Prox;  // Primeiro aluno nao alocado
  FimSemestreAA  = AlunoAloc->Primeiro->Prox;   // Primeiro aluno alocado
  while(FimSemestreANA != NULL){                // Enquanto nao chegar ao ultimo aluno nao alocado
    if(FimSemestreANA->Aluno.Oferecido == 1){   // Caso se tenha oferecido ao menos um quarto
      FimSemestreANA->Aluno.Exigencia -=5;      // Decrementar em 5 a exigencia do aluno (desespero)
      FimSemestreANA->Aluno.Oferecido = 0;      // Reseta flags setadas
    }
    FimSemestreANA = FimSemestreANA->Prox;      // Avancar aluno
  }
  while(FimSemestreAA != NULL){                 // Enquanto nao chegar ao ultimo aluno alocado
   FimSemestreAA->Aluno.Semestre -=1;           // Decrementar numero de semestres restantes
   FimSemestreAA->Aluno.Oferecido = 0;          // Reseta flags setadas
   FimSemestreAA = FimSemestreAA->Prox;         // Avancar aluno
  }
}
```

Tem-se complexidade $\mathcal{O}(M)$. Sempre se percorrerá os M alunos, já que o faz nas listas de alunos não alocados e alocados.

2.1.7 MoveListaQuartoDesocupou

Função responsável por desocupar os quartos liberados no fim do semestre. Quartos desocupados são colocados na lista de quartos vazios.

```
void MoveListaQuartoDesocupou(TipoListaQuarto *ListaQuartoOcupado, TipoListaQuarto *ListaQuartoVazio){
  ApontadorCelulaQuarto p,q;        // Inicializa ponteiro para quarto
  p = ListaQuartoOcupado->Primeiro; // Aponta para celula cabeca
  q = p->Prox;                      // Aponta para o primeiro quarto Ocupado
  if(q == NULL){                    // Caso a lista esteja vazia
    return;                         // Finalizar
  }
  while(q != NULL){                 // Enquanto nao chegar ao final da lista de quartos ocupados
    if( ((q->Quarto.Ocupado)->Semestre) == 0 ){ // Averiguar se aluno finalizou o curso (deve-se desocupar quarto)
        p->Prox = q->Prox;                      // Elimina quarto da lista de quartos ocupados
        ListaQuartoVazio->Ultimo->Prox = q;     // Transfere o quarto para a lista de quartos desocupados
        ListaQuartoVazio->Ultimo = ListaQuartoVazio->Ultimo->Prox;  // Atualiza o ultimo quarto da lista de vazios
        ListaQuartoVazio->Ultimo->Prox = NULL;  // "Aterra" o final da lista de quartos vazios
        if (p->Prox == NULL) {            // Caso seja o ultimo quarto:
          ListaQuartoOcupado->Ultimo = p; // Atualiza o ultimo quarto da lista de quartos ocupados
        }
        q = p->Prox;       // Avanca um quarto
    } else {               // Se nao desocupou o quarto, averiguar o prozimo
      p = q;               // Avanca apontador
      q = q->Prox;         // Avanca quarto
    }
  }
}
```

Tem-se complexidade $\mathcal{O}(N)$. No pior caso, a lista de quartos ocupados contém todos os N quartos.

2.1.8 TrabalhaDados

Função do programa responsável pelas manipulações das listas e parte lógica do programa, a partir dos dados obtidos pelo usuário. Basicamente reúne diversas outras funções para implementar de maneira simples o tratamento dos dados em uma única função.

```
void TrabalhaDados(TipoListaQuarto *QuartoVazio, TipoListaQuarto *QuartoOcupado, TipoListaAluno *AlunoNAloc, TipoListaAluno *AlunoAloc, int *SemestreTotal){
  do{                                                                  // Roda ao menos uma vez
    *SemestreTotal +=1;                                                // Incrementa a quantidade de semestres transcorridos
    AlocaAlunoQuarto(AlunoNAloc,QuartoVazio,AlunoAloc,QuartoOcupado);  // Aloca os alunos sem quarto nos quartos possíveis
    FimDoSemestre(AlunoNAloc,AlunoAloc,QuartoVazio);                   // Decrementa semestre nos alocados e padrão nos não alocado caso tenha quarto vazio
    MoveListaQuartoDesocupou(QuartoOcupado,QuartoVazio);               // Quartos desocupados são transferidos para a lista correta
  }while(!VaziaAluno(*AlunoNAloc));                                    // Tenta alocar alunos ate que todos tenham um quarto
}
```

Nesta função, o cálculo da ordem de complexidade passa a ser mais complexo. Será assumido o pior caso possível, isto é, considerando todos os quartos com o estado de conservação 0, padrão mínimo de apartamento de cada aluno 100 e quantidade de semestres que os mesmos irão residir no apartamento é máxima e igual a S. Extrapolando, assim, qualquer caso realista. O *loop* da função TrabalhaDados será executado, no máximo, de acordo com a Equação 1:

$$\mathcal{O}(\left\lceil \frac{M}{N} \right\rceil \times S + \frac{100}{5}) \times (\mathcal{O}(N) + \mathcal{O}(M) + \mathcal{O}(M \times N)) = \mathcal{O}(M^2 \times S) \qquad (1)$$

Visto que a quantidade de semestres S que cada aluno irá residir em um apartamento é nada mais que uma constante multiplicativa, a ordem é, de fato, $\mathcal{O}(M^2)$.

2.1.9 FreeAll

Função responsável por liberar toda a memoria dinâmica utilizada, evitando *leaks* de memoria.

```
void FreeAll(TipoListaQuarto *ListaQuartoOcupado,TipoListaQuarto *ListaQuartoVazio,TipoListaAluno *ListaAlunoAloc, TipoListaAluno *ListaAlunoNAloc){
//COMEÇA A DAR FREE EM TUDO QUE FOI USADO NO TRABALHO
  ApontadorCelulaAluno AuxAluno;
  ApontadorCelulaQuarto AuxQuarto;
//DAR FREE EM TUDO QUE RESTA NA LISTA DE ALUNOS NAO ALOCADOS
  free(ListaAlunoNAloc->Primeiro);
  ListaAlunoNAloc->Primeiro = NULL;
  ListaAlunoNAloc->Ultimo = NULL;

//TERMINA DE DAR FREE NA LISTA DE ALUNOS NAO ALOCADOS

//COMEÇA A DAR FREE NA LISTA DE ALUNOS ALOCADOS
  while(ListaAlunoAloc->Primeiro->Prox != NULL){
    AuxAluno = ListaAlunoAloc->Primeiro;
    while(AuxAluno->Prox != ListaAlunoAloc->Ultimo){
      AuxAluno = AuxAluno->Prox;
    }
    free(AuxAluno->Prox);
    ListaAlunoAloc->Ultimo = AuxAluno;
    AuxAluno->Prox = NULL;
  }
  free(ListaAlunoAloc->Primeiro);
  ListaAlunoAloc->Primeiro = NULL;
  ListaAlunoAloc->Ultimo = NULL;
  AuxAluno = NULL;
//TERMINA DE DAR FREE NA LISTA DE ALUNOS ALOCADOS

//COMEÇA A DAR FREE NA DE QUARTOS ALOCADOS
  while(ListaQuartoOcupado->Primeiro->Prox != NULL){
    AuxQuarto = ListaQuartoOcupado->Primeiro;
    while(AuxQuarto->Prox != ListaQuartoOcupado->Ultimo){
      AuxQuarto = AuxQuarto->Prox;
    }
    free(AuxQuarto->Prox);
    ListaQuartoOcupado->Ultimo = AuxQuarto;
    AuxQuarto->Prox = NULL;
  }
  free(ListaQuartoOcupado->Primeiro);
  ListaQuartoOcupado->Primeiro = NULL;
  ListaQuartoOcupado->Ultimo = NULL;
  AuxQuarto = NULL;
//TERMINA DE DAR FREE NA LISTA DE QUARTOS ALOCADOS

//COMEÇA A DAR FREE NA DE QUARTOS LIVRES
  while(ListaQuartoVazio->Primeiro->Prox != NULL){
    AuxQuarto = ListaQuartoVazio->Primeiro;
    while(AuxQuarto->Prox != ListaQuartoVazio->Ultimo){
      AuxQuarto = AuxQuarto->Prox;
    }
    free(AuxQuarto->Prox);
    ListaQuartoVazio->Ultimo = AuxQuarto;
    AuxQuarto->Prox = NULL;
  }
  free(ListaQuartoVazio->Primeiro);
  ListaQuartoVazio->Primeiro = NULL;
  ListaQuartoVazio->Ultimo = NULL;
  AuxQuarto = NULL;
//TERMINA DE DAR FREE NA LISTA DE QUARTOS LIVRES

}
```

Tem-se complexidade $\mathcal{O}(M^2 + N^2)$.

2.2 Leitura

Nesta seção, desenvolveu-se as funções responsáveis pela interpretação dos dados de entrada digitados pelo servidor do DDS. Vê-se a seguir a descrição da funcionalidade e ordem de complexidade de cada uma delas.

2.2.1 ComparaDados

Função responsável por tratar as entradas de texto escritas pelo usuário. Seta uma flag para cada um dos comandos digitados, de modo que se possa imprimir

ao final da execução do programa somente as informações solicitadas. Também é tratado um possível erro de digitação, em que um comando invalido é inserido.

```
int ComparaDados(char *token, TipoFlags *Flags){
  if (!strcmp(token,"semestres")) {     // Se inseriu semestres, seta flag de impressao de semestres
    Flags->Semestres = 1;
  } else if (!strcmp(token,"lista")) {  // Se inseriu lista, seta flag de impressao de lista
    Flags->Lista = 1;
  } else if (!strcmp(token,"tempo")) {  // Se inseriu tempo, seta flag de impressao de tempo
    Flags->Tempo = 1;
  } else if (!strcmp(token,"tudo")) {   // Seta todas as flags para 1. Nao ha mais possibilidades de setar flags. Retorno 1.
    Flags->Semestres = 1;
    Flags->Lista = 1;
    Flags->Tempo = 1;
    return 1;
  } else {// Esse caso somente será acionado se for digitado algo que nao seja palavra chave, fazendo o retorno da função ser 2
    Flags->Semestres = 0;
    Flags->Lista = 0;
    Flags->Tempo = 0;
    return 2;
  }
  return 0; // Retorna 0 em caso padrao.
}
```

Tem-se trivialmente complexidade $\mathcal{O}(1)$.

2.2.2 LeituraDeEntrada

Para ser possível a manipulação das entradas, criamos o arquivo leitura o qual é responsável pela recepção dos dados e manipulação dos mesmos. Conforme consta nas definições de entradas do projeto, receberemos strings as quais selecionarão as saídas desejadas. Tais strings serão "semestres" para que a saída seja a quantidade total de semestres gastos para alocar todos os alunos, "lista" para imprimir a lista na ordem em que os alunos foram alocados, "tempo" para imprimir o tempo de execução do programa e "tudo" para que seja impresso todas as opções anteriores. Para a manipulação das entradas em strings, utilizamos a função *strtok* para separar as palavras recebidas e chamamos a função "ComparaDados" para cada palavra.

```
void LeituraDeEntrada(TipoFlags *Flags,TipoListaQuarto *QuartoVazio,TipoListaAluno *AlunoNAloc){
    volatile int i;
    int  numApt, numEst;
    char *palavra;
    char input[30];
    TipoQuarto Quarto;
    TipoAluno Aluno;

    /*O do while a seguir funciona da seguinte forma, ele escaneara o input todo ate o /n desprezando o mesmo. Depois checará a presença das palavras
    chaves, se elas nao ocorrerem ou ocorrer alguma palavra que nao é chave, apitara msg de erro e pedira para entrar com os parametros de entrada novamente.
    Caso identifique em algum momento a palavra chave tudo, ele automaticamente sai do do while e se identificar palavras chaves(diferentes de tudo) ele irá
    setar as flags comparando com a chave e assim que o strtok varrer todas as palavras chaves, ele sairá da função do while
    */
    do{
    scanf("%[^\n]%*c",input);
    palavra = strtok(input," ");
      while(palavra != NULL)
      {
        if(ComparaDados(palavra, Flags)==1){// Se o retorno da função comparadados for 1, logo a palavra tudo foi digitada nao precisado mais ler nada
           break;
        }else if ((ComparaDados(palavra, Flags))==2){// Foi digitado algo q nao é palavra chave, msg de erro e sai do loop de analise de token
              printf("Uma das palavras nao consta no registro de palavras-chaves\nFavor redigitar os parametros de entrada\n");
              break;
        }
       palavra = strtok (NULL, " ");
      }
      if(palavra == NULL)break;
    }while((ComparaDados(palavra, Flags))==2);// Faz toda a analise novamente com novo scanf caso tenha sido digitado algo q nao seja palavra chave

     // Leitura de Numero de Apartamentos e Numero de Estudantes
     do {
      scanf("%d %d", &numApt, &numEst);
     } while (!(numApt>=0 && numEst>=0));

     /* Leitura de Dados dos Apartamentos, criamos o TipoQuarto Quarto, o qual receberá a o valor do estado de conservação e salvará em Quarto.Nota,
     salvará em Quarto.Chave(que será o numero do quarto) o valor de i+1 pois i começa em zero e o ponteiro Quarto.Ocupado, que se ocupado aponta para o Aluno
```

```
    que o ocupa, aponta para NULL*/
    for (i = 0; i < numApt; i++){
       scanf("%d", &Quarto.Nota);
       Quarto.Chave= i+1;
       Quarto.Ocupado = NULL;
       InsereQuarto(Quarto,QuartoVazio);
    }

    for (i = 0; i < numEst; i++){
       scanf("%d %d",&Aluno.Exigencia, &Aluno.Semestre);
       Aluno.Chave = i+1;
       Aluno.Oferecido = 0;
       InsereAluno(Aluno,AlunoNAloc);
    }
}
```

Tem-se complexidade $\mathcal{O}(M + N)$.

2.3 Main

Módulo principal do programa. Responsável por unir todas as funções desenvolvidas para implementar a funcionalidade final desejada.

```
int main(int argc, char const *argv[]) {
  // Inicializa variaveis do programa
  // AlunosNAloc é a lista de alunos sem alocação, AlunosAloc é a lista de alunos alocados
  // QuartoVazio é a lista de quartos vazios e QuartoOcupado é a lista de quartos ocupados
  TipoFlags Flag;
  TipoListaAluno AlunoNAloc,AlunoAloc;
  TipoListaQuarto QuartoVazio,QuartoOcupado;
  struct timeval beginTime, endTime, beginT, endT;
  unsigned long time1, time2;
  // Procedimento de inicializacao das listas e das flags de impressao
  // Le dados de entrada inseridos pelo usuario
  // Obtem tempo entrada e alocacao de estruturas
  gettimeofday(&beginTime, NULL);
  int SemestreTotal = 0 ;
  Flag.Semestres = 0;
  Flag.Lista = 0;
  Flag.Tempo = 0;
  FLAluno(&AlunoNAloc);
  FLAluno(&AlunoAloc);
  FLQuarto(&QuartoVazio);
  FLQuarto(&QuartoOcupado);
  LeituraDeEntrada(&Flag,&QuartoVazio,&AlunoNAloc);
  gettimeofday(&endTime, NULL);
  // Processa dados obtidos no procedimento anterior
  // Obtem tempo de computacao
  gettimeofday(&beginT, NULL);
  TrabalhaDados(&QuartoVazio, &QuartoOcupado, &AlunoNAloc, &AlunoAloc, &SemestreTotal);
  gettimeofday(&endT, NULL);
  // Imprime resultados solicitados pelo usuario de acordo com as flags obtidas previamente
  if(Flag.Semestres){
    printf("%d\n",SemestreTotal); // Quantidade de semestres gastos para alocar todos os alunos
  }
  if(Flag.Lista){
    ImprimeAluno(AlunoAloc);      // Ordem de alocacao dos estudantes
  }
  if(Flag.Tempo){
    time1 = ((endT.tv_sec * 1000000 + endT.tv_usec) - (beginT.tv_sec * 1000000 + beginT.tv_usec));
    time2 = ((endTime.tv_sec * 1000000 + endTime.tv_usec) - (beginTime.tv_sec * 1000000 + beginTime.tv_usec));
    printf("%ld\n",time1);   // Tempo de computacao
    printf("%ld\n",time2);   // Tempo de entrada e alocacao de estruturas
  }

  // Libera memoria utilizada no programa
  FreeAll(&QuartoOcupado, &QuartoVazio, &AlunoAloc, &AlunoNAloc);

  return 0;                // Fim de programa
}
```

Tem-se complexidade $\mathcal{O}(M^2)$, já que é a função TrabalhaDados que define a ordem do programa a ser impressa. Sendo preciosista, a ordem é $\mathcal{O}(M^2 + N^2)$, definida pela função FreeAll. De fato, mantém-se a mesma ordem para entradas realistas $M > N$.

3 Resultados

Com o programa completo, simulou-o em diversas situações. Para averiguar seu funcionamento, estipulou-se uma situação hipotética feita à mão, averiguando se o comportamento do sistema era condizente com o esperado. Passado o teste, procurou-se investigar a performance do programa sob uma carga elevada de dados, preocupando-se com o tempo de execução sem averiguar efetivamente a saída do programa. Com ambos os testes, espera-se confirmar a confiabilidade do sistema. Para gerar os dados necessários ao teste de carga do sistema, desenvolveu-se o seguinte *script* em MATLAB:

```
clear
clc

stud = 1e5;
apto = 1e2;

conservacao = randi([0 100],apto,1);
duplas(1,:) = randi([0 100],stud,1);
duplas(2,:) = randi([6 20],stud,1);

fileID = fopen('data.txt','w');
fprintf(fileID,'tempo\n');
fprintf(fileID,'%d %d\n',apto,stud);
fprintf(fileID,'%d\n',conservacao);
fprintf(fileID,'%d %d\n',duplas);
fclose(fileID);
```

Observe que se supôs o número máximo de semestres em 100 e o mínimo em 0, esperando-se uma entrada de dados que extrapola a realidade. Além disso, a qualidade dos quartos e exigência dos alunos tem distribuição uniforme com valores entre 0 e 100. Tem-se os parâmetros *stud* que indica o número de M estudantes e *apto* que indica o número de N apartamentos. As entradas variam de 1 a 100.000, valores realistas para a quantidade de alunos em uma universidade. Os resultados obtidos são resumidos na Tabela 3.

Table 3: Tempo de Computação (µs) para Diferentes Entradas (Conservação, Exigência e Semestres em [0,100])

N/M	1E+00	1E+01	1E+02	1E+03	1E+04	1E+05
1E+00	0	29	1556	85821	10243194	932440264
1E+01	1	2	166	12357	1146803	205365388
1E+02	0	2	18	1348	117504	15543431
1E+03	1	2	16	261	21610	1826323
1E+04	1	1	11	269	11493	439059
1E+05	1	2	10	232	10954	708717

Pode-se comparar os valores obtidos na Tabela 3 com a análise da ordem de complexidade do sistema desenvolvida. Como visto, a ordem do programa é $\mathcal{O}(M^2)$. Vê-se pela Tabela 3 que a ordem prática para as entradas especificadas é $\mathcal{O}(\frac{M^2}{N})$. De fato, a ordem teórica é calculada para o pior caso possível. Assim, a ordem prática está de acordo com a teórica. De fato, pode-se comprovar a ordem com uma nova simulação.

Para realizá-la, setou-se os parâmetros: conservação dos quartos 0, exigência dos alunos 100, semestres de cada aluno 100. Deste modo, obteve-se a Tabela 4:

Table 4: Tempo de Execução (µs) para Entrada Limite

N/M	1E+05
1E+00	3592316775
1E+01	334572037
1E+02	37555320
1E+03	16212749

É evidente que os resultados foram análogos. Pode-se explicar o termo $\frac{1}{N}$ na ordem prática pela função AlocaAlunoQuarto. De fato, na maioria dos casos, a função será executada de acordo com $\mathcal{O}(M \times 1)$, já que no caso limite analisado os alunos estarão todos alocados. Deste modo, a ordem calculada passa a ser $\mathcal{O}(\frac{M^2}{N})$, precisamente igual a ordem observada.