Tabelas de Dispersão

Estruturas de Dados

Tabelas de Dispersão

- Até o momento estudamos algoritmos de busca com esforço computacional $O(\log n)$
- Estudaremos agora estruturas de dados conhecidas como tabelas de dispersão (*hash tables*), que, se bem projetadas, podem ser usadas para buscar um elemento em ordem constante: $O(1)$.
- O preço pago por essa eficiência é um maior uso da memória.

Um exemplo

- Armazenar os dados referentes aos alunos de uma disciplina.
- Cada aluno é identificado pelo seu número de matrícula, que contém 7 dígitos. Ex: 9711234.
- Para permitir o acesso a qualquer aluno em ordem constante, podemos usar o número de matrícula do aluno como índice de um vetor – vet. Assim, acessamos os dados do aluno pela indexação do vetor – vet[mat].

Um exemplo

- O problema é que, nesse caso, o preço pago para ter esse acesso rápido é muito grande.
- Vamos considerar que a informação associada a cada aluno seja representada pela estrutura abaixo:

```
struct aluno{
   int mat;
   char nome[81];
   char email[41];
   char turma;
};
typedef struct aluno Aluno;
```

Um exemplo

- Como a matrícula é composta por sete dígitos, o vetor tem 10.000.000 elementos!
- Se cada estrutura ocupar 127 bytes, teremos gasto 1.270.000.000 bytes, ou seja, acima de 1 GB.
- Como na prática, cada turma tem 50 alunos, precisamos de apenas 6.350 (127*50) bytes.
- Podemos amenizar o problema com um vetor de ponteiros em vez de um vetor de estruturas.
- Apesar de menor, esse gasto de memória ainda é proibitivo.

Ideia Central

- A ideia central por trás de uma tabela de dispersão é identificar, na chave de busca, quais são as partes significativas.
  - Ex: 97 1 1234
- Dessa maneira, podemos usar um número de matrícula parcial, de acordo com a dimensão que queremos dar a nossa tabela (ou nosso vetor).
  - Ex: `Aluno* tab[100];`
- Para acessar o nome de um aluno, podemos usar como índice da tabela apenas os dois últimos digitos da matrícula.
  - `Ex. vet[mat%100]->nome.`

Ideia Central

- Dessa forma, o uso de memória excedente é pequeno, e o acesso a um determinado aluno, a partir do número de matrícula, continua imediato.
- O problema é que provavelmente existirão dois ou mais alunos da turma que apresentarão os mesmos últimos dois dígitos no número de matrícula.
- Dizemos que há uma **colisão**, pois alunos diferentes são mapeados para o mesmo índice da tabela.
- É preciso tratar o problema da **colisão.**

Ideia Central

- Existem diferentes métodos para tratar as colisões em tabelas de dispersão.
- Vale salientar que não há como eliminar a ocorrência de colisões.
- A meta é minimizar as colisões e usar um método com o qual, mesmo com colisões, saibamos identificar cada elemento da tabela individualmente.

Função de dispersão

- A função de dispersão (função de *hash*) mapeia uma chave de busca em um índice da tabela.
- No caso apresentado, adotamos como função de *hash* a utilização dos dois últimos dígitos do número de matrícula.

```
int hash (int mat){
  return (mat%100);
}
```

- Podemos generalizar essa função para tabelas de dispersão de dimensão N.

```
int hash (int mat){
  return (mat%N);
}
```

Função de dispersão

- Uma função de *hash* deve, sempre que possível, apresentar as seguintes propriedades:
  - Ser eficientemente avaliada.
  - Espalhar bem as chaves de busca.
- Além disso, para minimizar o número de colisões, a dimensão da tabela deve guardar uma folga em relação ao número de elementos efetivamente armazenados.
  - Como regra empírica, não devemos permitir uma taxa de ocupação maior que 75%
  - Uma taxa de 50% costuma trazer bons resultados
  - Taxa menor que 25% pode representar gasto excessivo de memória.

Tratamento de Colisão

- Existem diversas estratégias para tratar eventuais colisões que surgem quando duas ou mais chaves de busca são mapeadas para um mesmo índice da tabela de *hash*.
- Em todas as estratégias, a tabela de dispersão em si é representada por um vetor de ponteiros para a estrutura que representa a informação a ser armazenada, no caso Aluno. Podemos definir um tipo que representa a tabela por:

```
#define N 127
Typedef Aluno* Hash[N]
```

Uso da posição consecutiva livre

- Nesse estratégia, se a função de dispersão mapeia a chave de busca para um índice já ocupado, procuramos o próximo (usando incremento circular) índice livre da tabela para armazenar o novo elemento.
- Vale lembrar que uma tabela de dispersão nunca terá todos os elementos preenchidos (já mencionamos que uma ocupação acima de 75% eleva o número de colisões, o que descaracteriza a ideia central da estrutura).
- Portanto, podemos garantir que sempre existirá uma posição livre na tabela.

Uso da posição consecutiva livre

```
Aluno * hsh_busca(Hash tab, int mat)
{
  int h=hash(mat);
   while (tab[h] !=NULL){
    if (tab[h]->mat==mat)
      return tab[h];
    h=(h+1)%N;
   }
 return NULL;
 }
```

Uso da posição consecutiva livre

```
Aluno* hsh_insere(Hash tab, int mat, char *n, char *e, char t)
{
    int h=hash(mat);
    while (tab[h] !=NULL){
     if (tab[h]->mat==mat)
       break;
     h=(h+1)%N;
    }
    if (tab[h]==NULL){
      tab[h]=(Aluno*)malloc(sizeof(Aluno));
      tab[h]->mat=mat;
    }
    strcpy(tab[h]->nome, n);
    strcpy(tab[h]->email,e);
    tab[h]->turma=t;
    return tab[h];
  }
```

Uso de uma segunda função de dispersão

- Para evitar a concentração de posições ocupadas na tabela, essa segunda estratégia faz uma variação na forma de procurar uma posição livre a fim de armazenar o elemento que colidiu.

$$h'(x) = N - 2 - x\%(N-2)$$

em que *x* representa a chave de busca e *N* é a dimensão da tabela.

- Dois cuidados devem ser tomados na escolha dessa segunda função de dispersão
  - Ela nunca pode retornar zero.
  - Segundo, de preferência, ela não deve retornar um número divisor da dimensão da tabela. (N = n° primo)

Uso de uma segunda função de dispersão

```
int hash2(int mat){
   return N-2-mat%(N-2);
 }

 Aluno* hsh_busca(Hash tab, int mat){
   int h=hash(mat);
   int h2=hash2(mat);
   while (tab[h] != NULL){
   if (tab[h]->mat==mat)
     return tab[h];
   h=(h+h2)%N;
   }
   return NULL;
  }
```

Uso de listas encadeadas

- Uma estratégia diferente, mas ainda simples, consiste em fazer com que cada elemento da tabela *hash* represente um ponteiro para uma lista encadeada.
- Todos os elementos mapeados para um mesmo índice seriam armazenados na lista encadeada.

```
struct aluno{
   int mat;
   char nome[81];
   char turma;
   char email[41];
   struct aluno* prox;
}
typedef struct aluno Aluno;
```

Uso de listas encadeadas

```
Aluno* hsh_busca(Hash tab, int mat)
{
  int h=hash(mat);
  Aluno* a=tab[h];
  while (a!=NULL){
    if (a->mat==mat)
      return a;
    a=a->prox;
  }
  return NULL;
}
```

Uso de listas encadeadas

```
Aluno* hsh_insere(Hash tab, int mat, char *n, char *e, char t) {
    int h=hash(mat);
    Aluno* a = tab[h];
    while (a !=NULL){
      if (a->mat==mat)
        break;
      a=a->prox; }
    if (a==NULL){
      a=(Aluno*)malloc(sizeof(Aluno));
      a->mat=mat;
      a->prox=tab[h];
      tab[h]=a;
    }
    strcpy(a->nome, n);
    strcpy(a->email,e);
    a->turma=t;
    return a;
  }
```

- Slides baseados no livro **Introdução a Estruturas de Dados**, Waldemar Celes, Renato Cerqueira e José Lucas Rangel, Editora Campus, 2004.