MC202 - Estruturas de Dados

Guilherme P. Telles

IC

28 de abril de 2023

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides usam português anterior à reforma ortográfica de 2009.

Parte I

Estruturas de dados

Estruturas de dados

- Uma **estrutura de dados** é uma forma de organizar registros $R_1, R_2, \ldots, R_n$ na memória para permitir operações eficientes sobre esses dados.
- Cada registro tem um identificador único (chave) e outros dados de interesse da aplicação (dados satélites).
- A chave tipicamente é um número inteiro. Se não for, isso não muda as estruturas de dados mas pode mudar o desempenho delas.

- Exemplos são
  - registros de pessoas com chave CPF,
  - registros de veículos com chave Renavan,
  - registros de estudantes da U com chave RA,
  - registros de produtos em uma loja com chave código-de-barras,
  - etc.

- Quando considerarmos as estruturas de dados vamos considerar:
  - A composição dos dados e a forma como mudam.
  - O tipo e freqüência das operações realizadas sobre os dados.
  - A eficiência em tempo e memória.

Visões

- Quando estudamos as estruturas de dados alternamos entre duas visões: a de quem usa (especificação) e a de quem constrói (implementação).
- Para a especificação usa-se o conceito de *tipo abstrato de dados*:
  - Um TAD especifica um tipo de dados em termos dos valores que ele pode assumir e em termos da semântica das operações que podem modificá-lo, independentemente de implementação.
  - Nossa especificação de TADs vai ser pouco rigorosa.
- Para a implementação levaremos em conta as particularidades da linguagem de programação e alguns detalhes do hardware onde a estrutura de dados vai ser processada.

Situações triviais

- Se o volume de dados é muito pequeno então colocar os dados em um array e fazer uma busca seqüencial provavelmente é uma boa solução.
  - Por exemplo, registrar 20 pessoas por CPF.
  - As operações não levarão tempo constante, mas são tão poucos dados que qualquer solução mais sofisticada não será mais eficiente ou terá um ganho de eficiência irrisório.
- Se a freqüência de acesso aos dados é muito pequena (processamentos anuais, bienais etc.) então provavelmente não será vantajoso usar alguma estrutura de dados sofisticada.

Roteiro

- Formas lineares de organizar memória: arrays e listas encadeadas
- Estruturas de dados básicas
- Recursão
- Formas hierárquicas de organiza memória: árvores
- Filas de prioridades
- O problema da busca
- O problema da ordenação
- Grafos

Parte II

"Modelos" de memória

Dois "modelos" de organização de memória

- acesso aleatório: a memória consiste de posições consecutivas em que cada posição armazena um registro e seus campos. Cada registro pode ser modificado ou recuperado acessando diretamente a posição que ele ocupa na memória, em tempo constante.
  - Array
- encadeado: a memória consiste de nós. Cada nó contém um registro. Campos do registro no nó podem ser apontadores. Para modificar ou recuperar um registro, o endereço dele deve ser conhecido.
  - Listas encadeadas, árvores
- Podem ser combinados em estruturas de dados.

Arrays

Array

- Um array é formado por elementos consecutivos.
- Cada elemento do array pode ser lido ou escrito fazendo apenas um acesso à memória ocupada pelo array.
- O número de elementos de um array e o número de bytes que cada elemento ocupa são definidos quando ele é criado e não mudam.
- Um array com mais de uma dimensão é apenas uma visão diferente de um array com uma única dimensão.
  - P.ex., uma matriz na memória é um vetor onde as linhas são colocadas sucessivamente.

- Nomenclatura:
  - array, arranjo
  - array unidimensional: vetor
  - array bidimensional: matriz
- Algumas linguagens de programação não têm arrays de verdade.
- C tem e já sabemos usar.

Vetores dinâmicos

Vetor

- Um vetor permite acesso direto a cada elemento. Isso é uma vantagem.
- O tamanho de um vetor não muda. Isso é uma limitação.

Vetor dinâmico

- Um vetor dinâmico é um vetor associado com um método para aumentá-lo e reduzi-lo de acordo com a ocupação.
- Útil quando o número de registros que tem que ser armazenado não é conhecido antecipadamente.

Redimensionamento

- Uma forma bem estabelecida é:
  - Quando não há mais posições vazias e acontece uma inserção então o tamanho dobra.
  - Quando 3/4 das posições estão vazias e acontece uma remoção então o tamanho é reduzido à metade.
- O tamanho mínimo pode ser 1 (todos os tamanhos serão potências de 2) ou outra constante.

Inserção

- Antes de inserir um elemento testamos o número de posições ocupadas.
- Se o vetor estiver totalmente ocupado, criamos um vetor maior, copiamos os dados e liberamos o vetor antigo.

INSERT(A, x)

```
 1  if A.k == 0
 2        allocate A.array with 1 slot
 3        A.k = 1
 4        A.ℓ = 0
 5  elseif A.ℓ == A.k
 6        allocate new-array with 2 × A.k slots
 7        insert all items in A.array into new-array
 8        free A.array
 9        A.array = new-array
10        A.k = 2 × A.k
11  insert x into A.array
12  A.ℓ = A.ℓ + 1
```

Remoção

- Depois de remover um elemento testamos o número de posições ocupadas.
- Se o vetor ficou pouco ocupado, criamos um vetor menor, copiamos os dados e liberamos o vetor antigo.

DELETE(A, x)

```
 1  delete x from A.array
 2  A.ℓ = A.ℓ − 1
 3  if A.ℓ == 0
 4        free A.array
 5        A.k = 0
 6  elseif A.ℓ == A.k/4
 7        allocate new-array with A.k/2 slots
 8        insert all items in A.array into new-array
 9        free A.array
10        A.array = new-array
11        A.k = A.k/2
```

- Os detalhes das operações que foram chamadas de inserção e remoção não são importantes no que diz respeito ao vetor dinâmico.
- O que é importante é que o vetor pode ficar mais ou menos cheio.

Desempenho

- Os dados podem ter que ser movidos quando o vetor é redimensionado, o que tem custo:
  - de tempo para copiar os dados.
  - de memória, já que ao redimensionar deve haver espaço para as duas cópias.

- Esse processo de ficar copiando os dados sempre que o vetor é redimensionado é eficiente em tempo?
- Para essa política de redimensionamento é eficiente: para cada dado inserido ou removido são realizadas no máximo 2 operações de memória adicionais para realizar as cópias durante **todos** os redimensionamentos.

- Para ver que isso é verdade suponha que o vetor tem tamanho $2m$ e acabou de ser redimensionado.
- Suponha que cada nova inserção custa uma operação de memória e "deposita" 2 operações de memória em uma poupança.
- Quando o vetor estiver cheio teremos $2 \times 2m$ operações de memória na poupança e usamos essa poupança para copiar os dados no novo vetor.

- Para a remoção, suponha que o vetor tem tamanho m e que uma remoção acabou de ocorrer deixando a ocupação do vetor igual a $m/4$.
- Houve pelo menos $m/4$ remoções depois do redimensionamento que aumentou o tamanho de $m/2$ para m. Suponha que cada uma delas custou uma operação de memória e "depositou" 1 operação de memória na poupança.
- Então a poupança acumulou pelo menos $m/4$ operações de memória e usamos essa poupança para copiar os dados no novo vetor.

- Logo a sobrecarga para verificar o tamanho do vetor e redimensionar causa apenas um aumento constante (pequeno) no número de operações por inserção e remoção no vetor, o que é bastante eficiente.
- Essas contas valem para uma política de redimensionamento como PG de razão 2 realizado apenas quando o vetor está cheio ou $3/4$ vazio.

- Para outras políticas de redimensionamento, as contas e a conclusão sobre o desempenho podem ser diferentes.
- Em particular,
  - Quanto maior a razão, menor o custo por operação e mais "espaço não-ocupado".
  - Quanto menor a razão, maior o custo por operação e menos "espaço não-ocupado".

- Em algumas aplicações pode não ser necessário diminuir o vetor.

Arrays dinâmicos

- Essas idéias são generalizáveis para arrays com mais dimensões.

Vetores de bits

Vetor de bits

- Há dados que têm apenas dois estados: aprovado/reprovado, marcado/não-marcado, vendido/não vendido etc.
- Para armazenar dados binários precisamos no máximo de 1 bit para cada item.
- Não é comum encontrar um tipo primitivo de apenas um bit nas linguagens de programação.

- Em C, o menor inteiro tem 8 bits: unsigned char (quase sempre) ou uint8_t.
- Em um vetor binário desses tipos, vamos usar de fato apenas $\frac{1}{8}$ do espaço ocupado.

- Podemos implementar um vetor de bits usando operadores bit-a-bit e máscaras.
- Uma máscara é uma palavra com padrão de bits bem definido.
- Para construir máscaras para selecionar um único bit usamos deslocamento e negação bit-a-bit.

- As operações básicas são set, reset e test para cada bit do vetor.
  - set atribui o valor 1 a um bit.
  - reset atribui o valor 0 a um bit.
  - test determina se um bit é 0 ou 1.

set

```
   01011000        01011000
 | 00000010      | 00001000
-----------     -----------
   01011010        01011000
```

```
          01011000        01011000
set     | 00000010      | 00001000
        ----------      ----------
          01011010        01011000

          01011010        01011000
reset   & 11111101      & 11111101
        ----------      ----------
          01011000        01011000
```

	01011000	01011000
set	| 00000010	| 00001000
	01011010	01011000
	01011010	01011000
reset	& 11111101	& 11111101
	01011000	01011000
	01011010	01011000
test	& 00000010	& 00000010
	00000010	00000000

Em C

```
bitarray-char.h
bitarray-char.c
bitarray-char-main.c
```

- A forma considerada "clássica" é por macros para manipular um vetor de `unsigned char`.

```
#define nslots(n) (((n)>>3)+1)
#define set(A,i) ((A)[(i)>>3] |= 0x80 >> ((i) & 7))
#define reset(A,i) ((A)[(i)>>3] &= ~(0x80 >> ((i) & 7)))
#define test(A,i) (((A)[(i)>>3] & (0x80 >> ((i) & 7))) ? 1 : 0)
```

```
unsigned char *B = calloc(nslots(51),sizeof(unsigned char));

set(B,13)
reset(B,50)
if (test(B,0)) {
```

Listas encadeadas

Lista encadeada

- É formada por nós encadeados em seqüência.
- Cada nó guarda um registro de dados e um dos campos é o endereço do próximo nó.
- Precisamos saber o endereço do primeiro nó.
- O último nó aponta para um endereço especial, o endereço nulo.
- O primeiro nó é chamado de cabeça e o último é chamado de cauda (ou rabo).
- Também é chamada de *lista ligada*.

- Listas encadeadas podem aumentar e diminuir de tamanho.
- Permitem inserções e remoções de nós em qualquer posição.
- Não permitem acesso direto a um nó.

Exemplos de nó

```
struct player {
  char* nick;
  char* name;
  int age;
  int id;
  struct player* next;
};

typedef struct player player;
```

Exemplos de nó

```
struct player {
  char* nick;
  char* name;
  int age;
  int id;
  struct player* next;
};

typedef struct player player;
```

```
struct node {
  int data;
  struct node* next;
};

typedef struct node node;
```

Em C

- Encontramos duas formas principalmente:
  - usando apenas um struct: nó.
  - usando dois structs: nó e lista-encadeada.

Um struct

- Nessa forma definimos um struct para nó de lista encadeada.
- A lista encadeada é representada pou um apontador para a cabeça.
- Uma lista encadeada vazia é um apontador nulo.
- O apontador para a cabeça da lista encadeada é passado (por referência) para funções que manipulam a lista.

Dois structs

- Nessa forma definimos dois structs, um para nó de lista e outro para lista-encadeada.
- O struct do tipo lista-encadeada guarda pelo menos um apontador para a cabeça. Ele também pode guardar outras informações como um apontador para o rabo, o tamanho etc.
- Uma lista encadeada vazia é uma lista com apontador para a cabeça nulo.

Vamos fazer

- Inserir
- Procurar
- Remover
- Verificar se duas listas encadeadas são iguais

Nó sentinela

- Um nó sentinela (ou dummy) é um nó adicionado à lista encadeada para marcar uma posição e sinalizar alguma condição.

Sentinela no início

- Nesse caso qualquer lista encadeada contém pelo menos um nó.
- Permite escrever programas mais homogêneos, que não precisam tratar o caso de lista encadeada vazia.

Lista encadeada circular

- Na lista encadeada circular o rabo aponta para a cabeça e o apontador para a lista aponta para o rabo.

Lista encadeada circular

- Na lista encadeada circular o rabo aponta para a cabeça e o apontador para a lista aponta para o rabo.
- Dessa forma, mantendo um único apontador é possível ter acesso à cabeça e ao rabo e é possível inserir no início e no fim.

Lista duplamente encadeada

- Na lista duplamente encadeada cada nó aponta para seu sucessor e para seu predecessor.

Lista duplamente encadeada

- Na lista duplamente encadeada cada nó aponta para seu sucessor e para seu predecessor.
- Normalmente mantemos um apontador para a cabeça um apontador para o rabo da lista.
- Usa mais memória por nó.
- Há mais flexibilidade para navegar na lista e para modificá-la.

Combinações e modificações

- Combinações e modificações dos tipos de listas anteriores são possíveis.
- Por exemplo, lista duplamente encadeada com sentinela, lista duplamente encadeada circular etc.

Lista exógena

- As listas que consideramos até aqui são endógenas: cada nó armazena os dados e o(s) apontador(es) que encadeiam.
- Na lista exógena, cada nó armazena o(s) apontador(es) que encadeiam e um apontador para os dados. Isto é, os dados estão fora da lista.
- Pode ser mais econômica em memória quando há muitas repetições dos elementos da lista.
- Pode ser mais econômica em tempo quando há movimentações de nós entre listas e os dados são grandes, p.ex.
  dividir uma lista em k outras listas por intervalo de algum valor, calcular a interseção de listas etc.

Estrutura recursiva

- Uma lista encadeada é uma estrutura recursiva: cada nó tem uma referência para uma estrutura menor e do mesmo tipo.
- Logo, funções recursivas podem ser usadas com listas encadeadas.

Fila, pilha, deque, lista

Dados em seqüência

- Vamos dizer que um conjunto de dados que pode ter repetições está em *seqüência* se os dados são consecutivos e se a ordem relativa entre eles é importante.
- Estar em seqüência não quer dizer que seja ordenado.
- Quer dizer apenas que existe o primeiro, o segundo, o terceiro etc. mas não quer dizer que o primeiro seja menor que o segundo, que o segundo seja menor que o terceiro etc.

Fila (queue)

- Uma fila armazena dados em seqüência de tal forma que o primeiro a entrar é o primeiro a sair (FIFO).
- As operações típicas são:
  - $\textsc{push}(Q, x)$: adicionar x ao fim de Q.
  - $y = \textsc{eject}(Q)$: remover o primeiro elemento de Q.
  - $y = \textsc{first}(Q)$: recuperar o primeiro elemento de Q.
  - $y = \textsc{last}(Q)$: recuperar o último elemento de Q.

Pilha (stack)

- Uma pilha armazena dados em seqüência de tal forma que o último a entrar é o primeiro a sair (LIFO).
- As operações típicas são:
  - $\textsc{push}(S, x)$: adicionar x ao fim de S.
  - $y = \textsc{pop}(S)$: remover o último elemento de S.
  - $y = \textsc{last}(S)$: recuperar o último elemento de S.

Deque (double ended queue)

- Uma deque armazena dados em seqüência e permite inserções e remoções nas duas extremidades.
- As operações típicas são:
  - $\textsc{push}(D, x)$: adicionar x ao fim de D.
  - $y = \textsc{pop}(D)$: extrair o último elemento de D.
  - $\textsc{inject}(D, x)$: adicionar x ao início de D.
  - $y = \textsc{eject}(D)$: extrair o primeiro elemento de D.
  - $y = \textsc{first}(D)$: recuperar o primeiro elemento de D.
  - $y = \textsc{last}(D)$: recuperar o último elemento de D.

Lista

- Uma lista[1] armazena dados em seqüência e permite inserções, remoções e recuperação em qualquer posição.
- As operações típicas são:
  - $\text{INSERT}(L, x, k)$: adicionar x tornando-o o k-ésimo elemento de L, sendo que os elementos $k \ldots n$ antes da inserção se tornam o elementos $k+1, \ldots, n+1$.
  - $y = \text{REMOVE}(L, k)$: remover o k-ésimo elemento de L, sendo que os elementos $k+1 \ldots n$ antes da inserção se tornam o elementos $k, \ldots, n-1$.
  - $y = \text{GET}(L, k)$: recuperar k-ésimo elemento de L.

[1] Não há consenso sobre qual estrutura-de-dados é chamada de lista.

Implementação

- Um vetor, vetor dinâmico ou lista encadeada podem ser usados para implementar essas estruturas de dados.
- As operações nas pontas levam tempo constante.
- Usando um vetor ou vetor dinâmico recuperação na posição k leva tempo constante (ou amortizado constante) mas inserção e remoção na posição k levam tempo $O(k)$.
- Usando uma lista encadeada recuperação, inserção e remoção na posição k levam tempo $O(k)$.

- O vetor pode ficar com uma parte de seus elementos não utilizados pela estrutura de dados. A lista encadeada usa memória adicional nos apontadores.
- O vetor dinâmico é muito usado como solução geral.
- Em situações específicas, a composição das operações e localidade de acessos podem favorecer a lista encadeada.