SCC-0223 - Capítulo 3
Estruturas de Dados Elementares

João Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - São Carlos
joaoluis@icmc.usp.br

2023

Sumário

Agradecimento

Ao Prof. Ivandré Paraboni, da EACH-USP, pela permissão de uso de sua apostila de Estruturas e Dados I [1] para preparação deste material.

Sumário

Lista linear sequencial

- Lista linear: série de elementos ordenados na qual cada elemento exceto o primeiro possui um e apenas um antecessor, e cada elemento exceto o último possui um e apenas um sucessor.
- **Lista Linear Sequencial**: lista linear na qual a ordem (lógica) dos elementos da lista coincide com sua posição física (em memória); elementos adjacentes da lista ocupam posições contíguas de memória.
- A forma mais comum de implementação de uma lista sequencial é através de um vetor de elementos (`A`) do tipo `REGISTRO` de tamanho `MAX`.
- Há um contador de número de posições efetivamente ocupadas (`nroElem`): ocupação do vetor se dá sempre em posições contíguas; `nroElem-1` indica o último elemento existente na estrutura.

Lista linear sequencial

```
// tamanho máximo do vetor estático
#define MAX 50
typedef struct {
    int chave;
    // outros campos...
} REGISTRO;
typedef struct {
    REGISTRO A[MAX];
    int nroElem;
} LISTA;
```

Lista linear sequencial

- **Vantagens**:
  - Acesso direto a qualquer elemento com base no seu índice. O tempo é constante $\mathcal{O}(1)$. Em muitas aplicações o índice do dado procurado não é conhecido: vantagem apenas relativa.
  - Se a lista estiver ordenada pela chave em questão, a busca por uma chave pode ser efetuada através de busca binária $\mathcal{O}(log\ n)$.
- **Desvantagens**:
  - Mesmo em uma estrutura ordenada, o pior caso de inserção e exclusão (na frente da lista) exige movimentação de todos os n elementos da lista, ou seja, $\mathcal{O}(n)$: inadequado para aplicações em que ocorrem muitas atualizações.
  - Implementação estática, exigindo que o tamanho do vetor seja previamente estabelecido.

Sumário

Listas Lineares Ligadas (ou Encadeadas)

- Se a inserção e exclusão em listas sequenciais podem acarretar grande movimentação de dados, uma solução óbvia é permitir que os dados ocupem qualquer posição disponível (e não necessariamente a posição física correta), e então criar um esquema para preservar a ordem dos elementos e gerenciamento de nós livres/ocupados.
- Uma lista linear ligada (ou simplesmente **lista ligada**) é uma lista linear na qual a ordem (lógica) dos elementos da lista (chamados "nós") não necessariamente coincide com sua posição física (em memória).
- Pode ser implementada de forma estática (usando-se um vetor) ou dinâmica, com uso de ponteiros.

Listas Ligadas de Implementação Estática

- A lista é formada pelo vetor de registros (`A`), um indicador de início da estrutura (`inicio`) e um indicador de início da lista de nós disponíveis (`dispo`).
- Cada registro contém, além dos campos exigidos pela aplicação, um campo `prox` que contém um índice para o próximo elemento na série.
- Um campo `prox` com valor `-1` será usado para designar que o elemento em questão não possui sucessor.

Listas Ligadas de Implementação Estática

```
typedef struct {
    REGISTRO A[MAX];
    int inicio;
    int dispo;
} LISTA;
typedef struct {
    int chave;
    int prox;
} REGISTRO;
```

Listas Ligadas de Implementação Dinâmica

- Para evitar a necessidade de definição antecipada do tamanho máximo da estrutura de implementação estática (i.e. o vetor), podemos tirar proveito dos recursos de alocação dinâmica de memória, deixando o gerenciamento de nós livres/ocupados a cargo do ambiente de programação.
- Esta técnica constitui à implementação dinâmica de listas ligadas, e requer o uso das funções disponibilizadas em `malloc.h`.

Listas Ligadas de Implementação Dinâmica

- Em uma lista ligada de implementação dinâmica, não há mais uso de vetores.
- Cada elemento da lista é uma estrutura do tipo `NO`, que contém os dados de cada elemento (inclusive a chave) e um ponteiro `prox` para o próximo nó da lista. Um nome auxiliar (`estrutura`) é usado para permitir a auto-referência ao tipo `NO` que está sendo definido.

```
typedef struct estrutura {
    int chave;
    int info;
    struct estrutura *prox;
} NO;
typedef struct {
    NO* inicio;
} LISTA;
```

Listas Ligadas de Implementação Dinâmica

- O tipo `LISTA` é simplesmente um ponteiro `inicio` apontando para o primeiro nó da estrutura (ou para `NULL` no caso da lista vazia).
- O último elemento da lista possui seu ponteiro `prox` também apontando para `NULL`.

Listas Ligadas de Implementação Dinâmica

- A alocação e desalocação de nós é feita dinamicamente pelo próprio compilador C através das primitivas `malloc` e `free`, respectivamente.

```
NO* p = (NO*) malloc(sizeof(NO)); // cria um novo nó em memória,
                                  // apontado por p
free(p); // a área de memória apontada por p é liberada;
```

Listas dinâmicas com nó cabeça e circularidade

- O nó cabeça, criado no início de uma lista, é usado para simplificar o projeto dos algoritmos de inserção e exclusão. Pode também armazenar a chave de busca quando a lista for circular.
- A circularidade é em geral exigência da aplicação (e.g. que precisa percorrer continuamente a estrutura) mas pode também facilitar inserções e exclusões quando combinada com uso de um nó cabeça.

```
// Inicialização da lista circular e com nó cabeça
void inicializarLista(LISTA *l) {
      l->cabeca = (NO*) malloc(sizeof(NO));
      l->cabeca->prox = l->cabeca;
}
```

Listas dinâmicas duplamente encadeadas com nó cabeça e circularidade

- Quando necessitamos percorrer a lista indistintamente em ambas as direções, usamos encadeamento duplo (ligando cada nó ao seu antecessor e ao seu sucessor).

```
typedef struct estrutura {
      int chave;
      int info;
      struct estrutura *prox;
      struct estrutura *ant;
} NO;
typedef struct {
      NO* cabeca;
} LISTA;
```

Sumário

Filas

- **Filas** são listas lineares com disciplina de acesso FIFO (*first-in, first-out*, ou, primeiro a entrar é o primeiro a sair).
- Sua principal aplicação é o armazenamento de dados em que é importante preservar a ordem FIFO de entradas e saídas.
- O comportamento de fila é obtido armazenando-se a posição das extremidades da estrutura (chamadas aqui de `fim` e `inicio`), e permitindo entradas apenas na extremidade `fim` e retiradas apenas na extremidade `inicio`.
- A implementação pode ser estática (usando um vetor circular) ou dinâmica (com ponteiros) sem diferenças significativas em termos de eficiência, uma vez que estas operações só podem ocorrer nas extremidades da estrutura.

Filas

```
typedef struct estrutura {
    int chave;
    struct estrutura *prox;
} NO;
typedef struct {
    NO* inicio;
    NO* fim;
} FDINAM; // implementação de fila dinâmica
```

Sumário

Deques (Filas de duas pontas - *double-ended queues*)

- **Deques** são filas que permitem tanto entrada quanto retirada em ambas extremidades.
- Neste caso não faz mais sentido falar em início e fim de fila, mas simplesmente `inicio1` e `inicio2`.
- Implementações estáticas e dinâmicas são possíveis, mas a dinâmica é mais comum, tirando proveito do encadeamento duplo para permitir acesso a ambas extremidades em tempo $\mathcal{O}(1)$.

Deques

```
typedef struct estrutura {
    int chave;
    struct estrutura *prox;
    struct estrutura *ant;
} NO;
typedef struct {
    NO* inicio1;
    NO* inicio2;
} DEQUE;
// Inicialização do deque
void inicializarDeque(DEQUE* d) {
    d->inicio1 = NULL;
    d->inicio2 = NULL;
}
```

Sumário

Pilhas

- **Pilhas** são listas lineares com disciplina de acesso LIFO (*last-in, first-out*, ou, o último a entrar é o primeiro a sair).
- A pilha armazena apenas a posição de uma de suas extremidades (chamada *topo*), que é o único local onde são realizadas todas as operações de entrada e saída. A operação de entrada de dados (sempre no topo da pilha) é chamada *push* e a retirada (também sempre do topo) é chamada *pop*.
- A implementação pode ser estática (usando um vetor simples) ou dinâmica (com ponteiros) sem diferenças significativas em termos de eficiência, uma vez que a estrutura só admite estas operações em seu topo.

Pilhas

```
typedef struct estrutura {
    int chave;
    struct estrutura *prox;
} NO;
typedef struct {
    NO* topo;
} PDINAM;
// Inicialização da pilha dinâmica
void inicializarPdinam(PDINAM* p) {
    p->topo = NULL;
}
```

Sumário

Fila de prioridade

- **Fila de prioridade** é uma estrutura de dados que armazena elementos com base em suas prioridades. Cada elemento possui uma prioridade associada e a fila garante que os elementos com maior prioridade sejam atendidos primeiro [5].
- As filas de prioridade podem ser implementadas de várias maneiras, como por meio de uma heap binária, uma árvore de busca binária balanceada ou uma lista ordenada.
- Essa estrutura de dados é amplamente utilizada em algoritmos de otimização e em sistemas em que é necessário processar tarefas em ordem de prioridade.

Fila de prioridade

- Uma fila de prioridade é uma estrutura de dados muito útil para lidar com tarefas em que uma ordem de prioridade deve ser respeitada.
- **Aplicações** [5]:
  - *Algoritmos de busca*: em muitos algoritmos de busca, como A*, Dijkstra e Algoritmo de Bellman-Ford, é importante manter uma fila de prioridade para explorar os nós mais promissores primeiro.
  - *Sistemas de atendimento*: em sistemas de atendimento, como filas de espera em hospitais, bancos ou serviços de suporte, é comum usar uma fila de prioridade para lidar com pacientes ou clientes de acordo com sua gravidade ou urgência.
  - *Escalonamento de processos*: em sistemas operacionais, uma fila de prioridade é frequentemente usada para decidir qual processo deve ser executado em seguida, levando em consideração fatores como prioridade, tempo de espera e recursos disponíveis.

Fila de prioridade

```
struct cel{
    struct cel *ant;
    struct cel *prox;
    int dado;
    int prioridade;
};
typedef struct cel celula;
```

Sumário

Introdução

- Uma **matriz esparsa** é uma matriz extensa na qual poucos elementos são não-nulos (ou de valor diferente de zero). O problema de representação destas estruturas consiste em economizar memória armazenando apenas os dados válidos (i.e. não nulos) sem perda das propriedades matriciais (i.e. a noção de posição em linha e coluna de cada elemento).
- As implementações mais comuns são a representação por linhas ou por listas cruzadas.

Representação por Linhas

- A forma mais simples (porém não necessariamente mais eficiente) de armazenar uma matriz esparsa é na forma de uma tabela (na verdade, uma lista ligada) de nós contendo a linha e coluna de cada elemento e suas demais informações.
- A lista é ordenada por linhas para facilitar o percurso neste sentido.
- A matriz será acessível a partir do ponteiro de início da lista ligada que a representa.

Representação por Linhas

	1	2	3	4	5	6	7	8
1								
2			A			B		
3			C					
4			D					
5			E					
6					F	G	H	
7								
8								

	lin	col	info
0	8	8	
1	2	3	A
2	2	6	B
3	3	3	C
4	4	3	D
5	5	3	E
6	6	5	F
7	6	6	G
8	6	7	H

Representação por Linhas

- A economia de espaço desta representação é bastante significativa.
- Para uma matriz de `MAXLIN x MAXCOL` elementos, dos quais n são não-nulos, seu uso será vantajoso (do ponto de vista da complexidade de espaço) sempre que:

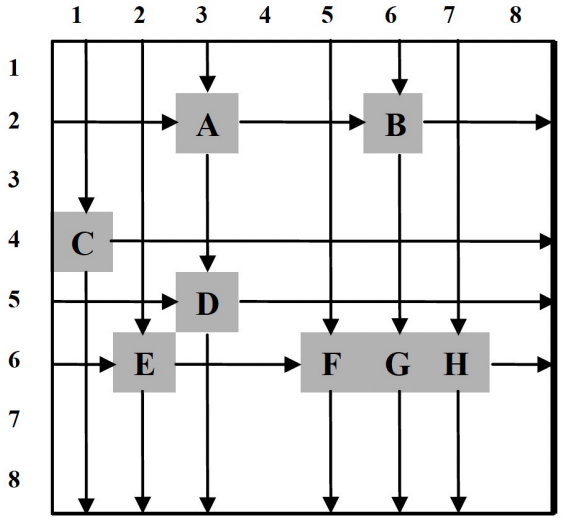
```
(MAXCOL*MAXLIN*sizeof(TIPOINFO)) > (n*(sizeof(NO))
```

- Lembrando que um nó é composto de dois inteiros, um `TIPOINFO` e um `NO*`.
- Por outro lado, a representação por linhas não apresenta bom tempo de resposta para certas operações matriciais. Em especial, percorrer uma linha da matriz exige que todas as linhas acima dela sejam percorridas.
- Pior do que isso, percorrer uma coluna da matriz exige que a matriz inteira (ou melhor, todos os seus elementos não-nulos) seja percorrida. Considerando-se que matrizes esparsas tendem a ser estruturas de grande porte, em muitas aplicações estes tempos de execução podem ser inaceitáveis.

Representação por Listas Cruzadas

- Com um gasto adicional de espaço de armazenamento, podemos representar uma matriz esparsa com tempo de acesso proporcional ao volume de dados de cada linha ou coluna. Nesta representação, usamos uma lista ligada para cada linha e outra para cada coluna da matriz.
- As listas se cruzam, isto é, compartilham os mesmos nós em cada intersecção, e por este motivo cada nó armazena um ponteiro para a próxima linha (abaixo) e próxima coluna (à direita).
- O acesso a cada linha ou coluna é indexado por meio de um vetor de ponteiros de linhas e outro de colunas. Cada ponteiro destes vetores indica o início de uma das listas da matriz.

Representação por Listas Cruzadas



Sumário

Big Numbers

- Representações via concatenação de dígitos:
- Arranjos de Dígitos:
  - Elemento inicial representa o dígito menos significativo.
  - Mantém-se um contador com o número de dígitos significativos.
  - Contador está relacionado ao índice da célula do último dígito.
  - Vantagens:
    - Simples.
    - Eficiente: 100.000 dígitos ? Arranjo de 100.000 chars ? 100Kb !!!
  - Requerimento:
    - Limitante superior não conservativo para o no. máximo de dígitos.
- Lista Encadeada de Dígitos (Estrutura Dinâmica):
  - Mais complexa, mas necessária quando não se dispõe de um limitante superior não conservativo para o número de dígitos.

Big Numbers - Representação via concatenação de dígitos

```
#include <stdio.h>

#define MAXDIGITS   100      /* maximum length bignum */
#define PLUS        1        /* positive sign bit */
#define MINUS       -1       /* negative sign bit */

typedef struct {
        char digits[MAXDIGITS]; /* represent the number */
        int signbit;            /* 1 if positive, -1 if negative */
        int lastdigit;          /* index of high-order digit */
} bignum;
```

Big Numbers - Representação via concatenação de dígitos

```
print_bignum(bignum *n)
{
    int i;

    if (n->signbit == MINUS) printf("- ");
    for (i=n->lastdigit; i>=0; i--)
        printf("%c",'0'+ n->digits[i]);

    printf("\n");
}
```

Nota:

- Os dígitos não correspondem aos caracteres 0 a 9 (ASCII 48 a 57), mas aos caracteres cujos códigos ASCII são 0 a 9.
- Isso permite operar com os dígitos (`char`) como se fossem inteiros (`int`).

Sumário

Introdução

- **Listas Generalizadas** são listas contendo dois tipos de elementos: elementos ditos "normais" (e.g. que armazenam chaves) e elementos que representam entradas para sublistas.
- Para decidir se um nó armazena uma chave ou um ponteiro de sublista, usamos um campo tag cuja manutenção é responsabilidade do programador.
- Dependendo do valor de tag, armazenamos em um campo de tipo variável (um `union` em C) o dado correspondente.
- Por exemplo, podemos convencionar que `tipo=1` representará elementos regulares da lista (i.e. contendo chaves), e que `tipo=2` representará uma sublista (i.e. contendo um ponteiro para esta).

Listas generalizadas

```
typedef struct estrutura {
    int tipo; // 1=elemento e 2=sublista
    union {
        int chave;
        struct estrutura *sublista;
    };
    struct estrutura *prox;
} NO;
// Inicialização
void inicializarLista(NO* *p) {
    *p = NULL;
}
```

Referências I

[1] Honda, W. Y., Paraboni, I.
ACH2023 - Algoritmos e Estruturas de Dados I
Universidade de São Paulo, Escola de Artes, Ciências e Humanidades, Sistemas de Informação. Apostila atualizada em 26/09/2022

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Introduction to Algorithms.
Third edition, The MIT Press, 2009.

[3] Rosa, J. L. G.
Análise de Algoritmos - parte 2 e Divisão e Conquista. SCC-201 Introdução à Ciência da Computação II (capítulo 3).
Slides. Ciência de Computação. ICMC/USP, 2009.

Referências II

[4] Rosa, J. L. G.
SCC-210 Algoritmos Avançados - Capítulo 5 - Aritmética e Álgebra
Slides. Ciência de Computação. ICMC/USP, 2010.

[5] Wikipedia
Fila de prioridade
https://pt.wikipedia.org/wiki/Fila_de_prioridade