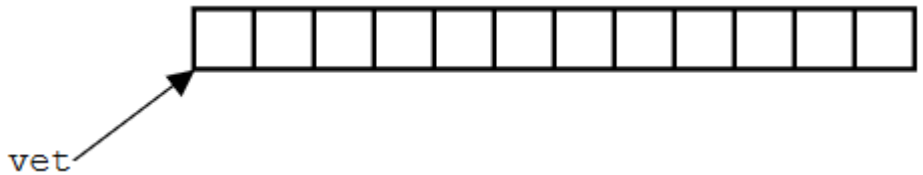
Listas Encadeadas Simples

Prof. Leandro Colevati

Introdução

Para representarmos um grupo de dados, podemos usar um vetor.

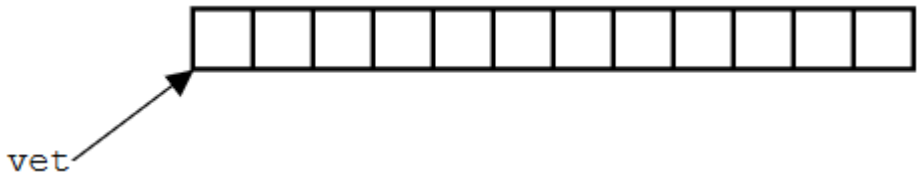
O vetor é a forma mais primitiva de representar diversos elementos agrupados. Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.



Introdução

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo vet, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de vet é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação vet[i].

Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.



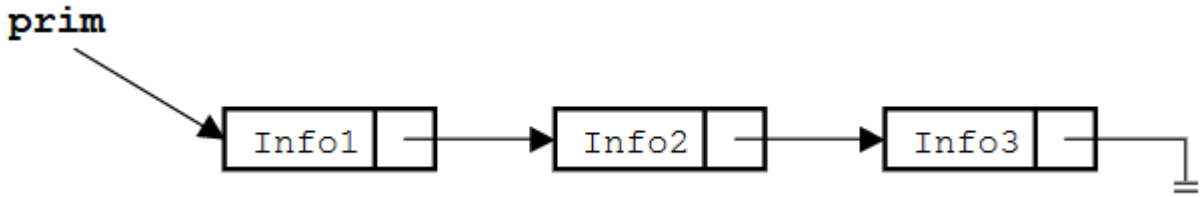
Introdução

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos sub-utilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas dinâmicas e armazenam cada um dos seus elementos usando alocação dinâmica.

Definição

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista.



Definição

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista.

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó).

Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

Simular operações

addFirst(1)
addLast(2)
add(10, 1)
addFirst(0)
get(1)
add(20, 2)
get(3)
removeFirst()
removeLast()
remove(1)
get(1)
size()

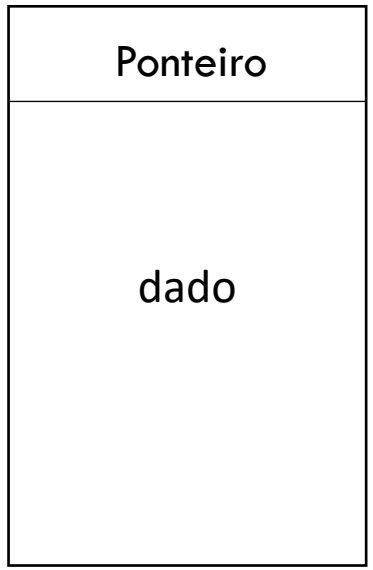
Introdução

- Operações Básicas:
  - Teste de lista vazia;
  - Criação da lista;
  - Adicionar um elemento no início da lista;
  - Adicionar um elemento no fim da lista;
  - Adicionar um elemento em qualquer lugar da lista;
  - Remover um elemento do início da lista;
  - Remover um elemento no fim da lista;
  - Remover um elemento em qualquer lugar da lista;
  - Acesso aos elementos da lista.
    - Verificar elemento de uma posição da lista
    - Tamanho

Alocação Dinâmica

- Considere a definição do tipo Lista abaixo:

```
class No {
   tipo    dado;
   No      próximo; //Ponteiro
}
```



Alocação Dinâmica

- Considere a definição do tipo Lista abaixo:
  - Ponteiro Primeiro → NULL

Alocação Dinâmica

- Teste de lista vazia:

```
No primeiro;
booleano listaVazia() {
    se (primeiro == nulo) {
        retorne verdadeiro;
    } senão {
        retorne falso;
    }
}
```

Alocação Dinâmica

- Tamanho da lista:

```
No primeiro;
int size() {
        int cont = 0;
        se (listaVazia() == falso) {
                No auxiliar = primeiro;
                enquanto (auxiliar != nulo) {
                   cont = cont + 1;
                   auxiliar = auxiliar.proximo;
                }
        }
        retorne cont;
}
```

Alocação Dinâmica

- ## Retornar um Nó:

```
No primeiro;
No getNo(int pos) {
        se (listaVazia == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        int tamanho = size();
        se (pos < 0 || pos > tamanho - 1) {
            exceção(“Posição inválida”);
        }
        No auxiliar = primeiro;
        int cont = 0;
        enquanto (cont < pos) {
             auxiliar = auxiliar.proximo;
             cont = cont + 1;
        }
        retorne auxiliar;
}
```

Alocação Dinâmica

- Adicionar elemento no início:

```
No primeiro;
void addFirst(tipo valor) {
        No elemento = new No();
        elemento.dado = valor;
        elemento.proximo = primeiro;
        primeiro = elemento;
}
```

Alocação Dinâmica

- Adicionar elemento no fim:

```
No primeiro;
void addLast(tipo valor) {
        int tamanho = size();
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        No elemento = new No();
        elemento.dado = valor;
        elemento.proximo = nulo;
        No ultimo = getNo(tamanho - 1);
        ultimo.proximo = elemento;
}
```

Alocação Dinâmica

- Adicionar elemento em qualquer posição válida:

No primeiro;

```
void add (tipo valor, int pos) {
        Int tamanho = size();
        se (pos < 0 || pos > tamanho) {
            exceção(“posição inválida”);
        }
        se (pos == 0) {
            addFirst(valor);
        } senao se (pos == tamanho) {
            addLast(valor)
        } senao {
            No elemento = new No();
            elemento.dado = valor;
            No anterior = getNo(pos - 1);
            elemento.proximo = anterior.proximo;
            anterior.proximo = elemento;
        }
}
```

Alocação Dinâmica

- Remover elemento do início:

```
No primeiro;
void removeFirst() {
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        primeiro = primeiro.proximo;
}
```

Alocação Dinâmica

- Remover elemento do fim:

```
No primeiro;
void removeLast() {
        se (listaVazia() == verdadeiro) {
             exceção(“Lista Vazia”);
        }
        int tamanho = size();
        se (tamanho == 1) {
             removeFirst();
        } else {
             No penultimo = getNo(tamanho – 2);
             penultimo.proximo = null;
        }
}
```

Alocação Dinâmica

- Remover elemento do qualquer posição válida:

```
No primeiro;
void remove (int pos) {
        Int tamanho = size();
        se (pos < 0 || pos > tamanho - 1) {
            exceção(“posição inválida”);
        }
        se (listaVazia() == verdadeiro) {
            exceção(“Lista Vazia”);
        }
        se (pos == 0) {
            removeFirst();
        } senao se (pos == tamanho – 1) {
            removeLast();
        } senao {
            No anterior = getNo(pos – 1);
            No atual = getNo(pos);
            anterior.proximo = atual.proximo;
        }
}
```

Alocação Dinâmica

- Acessando elementos da Lista
  - Como estamos usando uma lista simplesmente encadeada podemos acessar todos os elementos da fila, a partir do primeiro.

Alocação Dinâmica

- Verificando o elemento de uma posição:

```
No primeiro;
tipo get(int pos) {
    se (listaVazia() == verdadeiro) {
        exceção(“Lista Vazia");
    }
    int tamanho = size();
    se (pos < 0 || pos > tamanho - 1) {
        exceção(“Posição inválida");
    }
    int cont = 0;
    No auxiliar = primeiro;
    enquanto (cont < pos) {
        auxiliar = auxiliar.proximo;
        cont++;
    }
    return auxiliar.dado;
}
```

Alocação Dinâmica

- Exemplo(Lista de Strings):

```
class exemplo {
        void main(String[] args) {
            Lista l = new Lista()
            booleano vazia = l.listaVazia();
            escreva(vazia);
            int tamanho = l.size();
            escreva("Tamanho:"+tamanho);
            String valor = l.get(0);
            l.addFirst("C");
            l.addFirst("B");
            l.addFirst("A");
            l.addLast("D");
            l.add ("Y", 0);
            l.add ("X", 2);
            tamanho = l.size();
            escreva("Tamanho:"+tamanho);
            dado = l.get(0);
            escreva(dado);
            l.add ("K", 10);
            l.add ("Z", 6);
            l.removeFirst();
            ...
```

...continuação

```
class exemplo {
        void main(String[] args) {
            ...
            dado = l.get(0);
            escreva(dado);
            l.removeLast();
            tamanho = l.size();
            dado = l.get(tamanho – 1);
            escreva(dado);
            l.remove (1);
            tamanho = l.size();

            escreva(l.toString());
        }
}
```

Teste de Mesa

Considere o vetor:

36	28	146	14	-65	117	-40	24	138	116

Faça o teste de mesa conforme o algoritmo:

```
Lista l = new Lista();
Para (int valor : vetor) {
        Se (listaVazia()) {
                l.addFirst(valor + 5);
        } Senao Se (l.size() == 1) {
                l.addFirst(valor * 5);
        } Senao Se (valor < 0) {
                l.addLast(valor * (-1));
        } Senao Se (valor % 2 == 0) {
                l.add(valor * 2, 1);
        } Senao {
                l.add(valor * 3, 2);
        }
}
```

```
int tamanho = l.size();
escreva(tamanho);
enquanto(! l.isEmpty()) {
        Se (tamanho == 1) {
                escreva(l.get(0));
                l.removeFirst();
        } Senao Se (tamanho > 4) {
                escreva(l.get(2));
                l.removeFirst();
        } Senao {
                escreva(l.get(tamanho – 1));
                l.removeLast();
        }
        tamanho = l.size();
}
```

Exercício 1

Simular o comportamento de pilhas dinâmicas para os algoritmos abaixo (A simulação deve deixar evidente a Lista que sobrou na memória):

a)

```
Para (i = 0 ; i < 10 ; i++) {
        Se (i % 2 == 0) {
             lista.addFirst(i * i);
        } Senão Se (i <= 6) {
             lista.addFirst(i);
        } Senão {
             escreva(lista.get(size() – 1);
             lista.removeLast();
        }
}
Escreva(Size());
```

b)

```
Para (i = 100 ; i < 115 ; i++) {
        Se (lista.isEmpty()) {
                 lista.addFirst(i + 100);
        } Senão Se (Size() <= 4) {
                 lista.addLast(i + 50);
        } Senão {
                 Escreva(lista.get(0));
                 lista.removeFirst();
        }
}
Escreva(Size());
```

Exercício 2

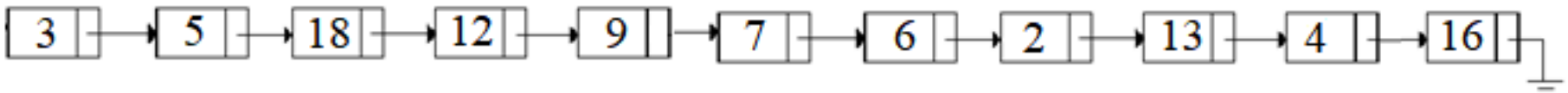
a) Ajustar o projeto de aula para criar uma biblioteca de Lista de Inteiros, gerando o JAR ListaInt

b) Transformar o projeto de em uma biblioteca de Lista de Strings, gerando o JAR ListaStrings.

c) Transformar o projeto de em uma biblioteca de Lista de Objetos, gerando o JAR ListaObject.

Exercício 3

Dada a Lista L abaixo, fazer:



a) Determine a sequência de passos para fazer a exibição dos elementos em ordem invertida

b) Fazer, em Java, um novo projeto que insira a Lista L como acima e implemente operações que permita a exibição dos elementos em ordem invertida