Estruturas de Dados — Pilhas, Filas, Listas

Fabio Gagliardi Cozman

PMR2300
Escola Politécnica da Universidade de São Paulo

Introdução

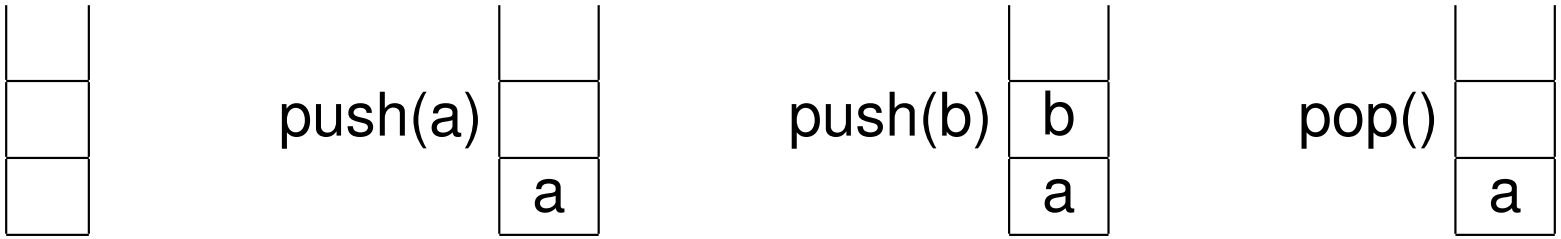
- Estruturas de dados são objetos que armazenam dados de forma eficiente, oferecendo certos “serviços” para o usuário (ordenação eficiente dos dados, busca por meio de palavras chave, etc).
- Técnicas de programação orientada a objetos são úteis quando temos que codificar estruturas de dados.
- As estruturas básicas abordadas neste curso são:
  - Pilhas, filas, listas ligadas.
  - Árvores e árvores de busca.
  - Hashtables (tabelas de dispersão).
  - Grafos.

Abstração

- Uma estrutura de dados abstrai as características principais de uma atividade que envolve armazenamento de informações.
- Por exemplo, a estrutura de *fila* armazena dados de forma que o dado há mais tempo na estrutura é o primeiro a ser retirado.

Pilhas

- Uma *pilha* é uma estrutura de dados em que o acesso é restrito ao elemento mais recente na pilha.



Pilhas: operações básicas

- As operações básicas realizadas com uma pilha são:
  - push: inserir no topo;
  - pop: retirar do topo;
  - top: observar o topo.
- Em uma pilha “ideal”, operações básicas devem ocorrer em $O(1)$, independentemente do tamanho N da pilha (ou seja, em tempo constante).

Implementação de Pilha

- Temos a seguir uma implementação de Pilha que usa um arranjo para armazenar dados.
- Note que esta implementação de Pilha usa *amortização*: quando o arranjo é inteiramente preenchido, seu tamanho é duplicado.

Implementação de Pilha (1)

```
public class PilhaAr {
    private Object arranjo[];
    private int topo;

    static private final int DEFAULT = 10;

    public PilhaAr() {
        arranjo = new Object[DEFAULT];
        topo = -1;
    }

    public void esvazie() { topo = -1; }

    public int tamanho() { return(topo + 1); }
```

Implementação de Pilha (2)

```
public void push(Object x) {
    topo++;
    if (topo == arranjo.length)
        dupliqueArranjo();
    arranjo[topo] = x;
}

private void dupliqueArranjo()  {
    Object novoArranjo[] =
            Object[2 * arranjo.length];
    for (int i=0; i<arranjo.length; i++)
        novoArranjo[i] = arranjo[i];
    arranjo = novoArranjo;
}
```

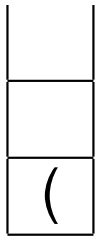
Implementação de Pilha (3)

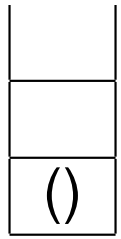
```
    public Object top() {
        if (topo >= 0)
            return(arranjo[topo]);
        else return(null);
    }

    public Object pop() {
        if (topo >= 0)
            return(arranjo[topo--]);
        else return(null);
    }
}
```

Consistência de parênteses

- Considere o problema de verificar se existe um fechamento de parênteses para cada abertura em uma expressão algébrica com letras e símbolos $+, -, *, /$.
- Pode-se utilizar uma pilha:

$$A + B * (C/D + E)$$





Verificação de consistência de parênteses (1)

Note: pilha que manipula caracteres; retorna '0' se vazia.

```
public class Verifica {
   static public void main(String args[]) {
      if (args.length < 1) {
         System.out.println("Insira entrada!");
         System.exit(-1);
      }
      String entrada = args[0];
      PilhaAr pilha = new PilhaAr();
      for (int i=0; i<entrada.length(); i++) {
         char c = entrada.charAt(i);
         if (c=='(')
                  pilha.push(c);
         if (c==')') {
                  char o = pilha.pop();
```

Verificação de consistência de parênteses (2)

```
            if (o=='0') {
                System.out.println("Falta abertura!");
                System.exit(-2);
            }
        }
    }
    if (pilha.tamanho()>0) {
        System.out.println("Falta fechamento!");
        System.exit(-3);
    }
    else System.out.println("Entrada consistente!");
  }
}
```

Avaliação de expressões

- Pilhas são muito usadas no processamento de linguagens, por exemplo em compiladores.
- Uma aplicação importante é a conversão e avaliação de expressões numéricas.
- Existem três tipos de notações para expressões numéricas:
  1. infixa, onde operador entre operandos: $(A + B)$;
  2. pós-fixa, onde operador segue operandos: $(AB+)$ (notação polonesa reversa);
  3. pré-fixa, onde operador precede operandos: $(+AB)$ (notação polonesa).

Notação pós-fixa

A vantagem da notação pós-fixa é que ela dispensa parênteses.

Infixa	Pós-fixa
$A - B * C$	$ABC * -$
$A * (B - C)$	$ABC - *$
$A * B - C$	$AB * C-$
$(A - B) * C$	$AB - C*$
$A + D/(C * D\hat{}E)$	$ADCDE\hat{} * /+$
$(A + B)/(C - D)$	$AB + CD - /$

Avaliação de expressões

- Suponha que tenhamos uma expressão pós-fixa e desejemos obter o valor da expressão (“avaliar a expressão”).
- Fazemos isso passando pelos elementos da expressão,
  1. empilhando cada operando;
  2. processando cada operador:
     1. retiramos dois operandos da pilha;
     2. executamos a operação;
     3. empilhamos o resultado.
- No final o resultado está no topo da pilha.

Exemplo: expressão $1; 2; 3; \hat{}; -; 4; 5; 6; *; +; 7; *; -$

Operação Parcial	Conteúdo da Pilha
Insere 1	1
Insere 2	1; 2
Insere 3	1; 2; 3
Operador: 2^3	1; 8
Operador: 1-8	-7
Insere 4	-7; 4
Insere 5	-7; 4; 5
Insere 6	-7; 4; 5; 6
Operador: 5*6	-7; 4; 30
Operador: 4+30	-7; 34
Insere 7	-7; 34; 7
Operador: 34*7	-7; 238
Operador: -7-238	-245 (**Resultado final**)

Avaliação de expressões - Implementação (1)

```
import java.util.StringTokenizer;

public class Avalia {
   static public void main(String args[]) {
      if (args.length<1) {
         System.out.println("Insira entrada!");
         System.exit(-1);
      }

      String entrada = args[0];
      PilhaAr pilha = new PilhaAr();

      StringTokenizer st = new StringTokenizer(entrada);
```

Avaliação de expressões - Implementação (2)

```
while (st.hasMoreElements()) {
        String nextToken = st.nextToken();
        if (nextToken.compareTo("+") == 0) {
            opera(pilha, 1);
        } else if (nextToken.compareTo("-")==0) {
            opera(pilha, 2);
        } else if (nextToken.compareTo("*")==0) {
            opera(pilha, 3);
        } else if (nextToken.compareTo("/")==0) {
            opera(pilha, 4);
        } else if (nextToken.compareTo("^")==0) {
            opera(pilha, 5);
        } else
            pilha.push(nextToken);
    }
    System.out.println("Resultado = " + pilha.pop());
}
```

Avaliação de expressões - Implementação (3)

```
    static void opera(PilhaAr pilha, int tipo) {
        int int1 = Integer.parseInt((String)(pilha.pop()));
        int int2 = Integer.parseInt((String)(pilha.pop()));
        int resultado = 0;
        if (tipo == 1)
            resultado = int2 + int1;
        else if (tipo == 2)
            resultado = int2 - int1;
        else if (tipo == 3)
            resultado = int2 * int1;
        else if (tipo == 4)
            resultado = int2 / int1;
        else if (tipo == 5) {
            resultado = 1;
            for (int i=0; i<int1; i++)
                resultado = resultado * int2;
        }
        pilha.push((new Integer(resultado)).toString());
    }
}
```

Conversão para notação pós-fixa

- Considere que temos uma expressão em notação infixa.
- Para convertê-la a notação pós-fixa, usamos uma pilha.
- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando...

Exemplo

- $A - B * C + D$:

Entrada	Pilha	Saída
A		A
$-$	$-$	A
B	$-$	$A\ B$
$*$	$-*$	$A\ B$
C	$-*$	$A\ B\ C$
$+$	$+$	$A\ B\ C * -$
D		$A\ B\ C * -\ D\ +$

Conversão para notação pós-fixo: algoritmo

- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando e colocando na saída os operadores na pilha até encontrarmos um operador com precedência menor...
    - Precedência: $+$ e $-$, seguida por $*$ e $/$, seguida por $\hat{}$.
- Ao final, desempilhamos e colocamos na saída os operadores que restarem na pilha.

Exemplo

- $A * B - C + D$:

Entrada	Pilha	Saída
A		A
$*$	$*$	A
B	$*$	$A\ B$
$-$	$-$	$A\ B\ *$
C	$-$	$A\ B\ *\ C$
$+$	$+$	$A\ B\ *\ C\ -$
D		$A\ B\ *\ C\ -\ D\ +$

Parênteses

- Para lidar com parênteses, podemos criar uma nova pilha a cada abertura de parênteses, e operar nessa nova pilha até encontrar o fechamento correspondente (quando então envaziamos a pilha e retornamos à pilha anterior).
- Podemos fazer isso usando uma única pilha, “simulando” a abertura e fechamento de outras pilhas no seu interior...

Conversão para notação pós-fixo: algoritmo completo

- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando e colocando na saída os operadores na pilha até encontrarmos um operador com precedência menor ou uma abertura de parênteses;
    - Precedência: $+$ e $-$, seguida por $*$ e $/$, seguida por $\hat{}$.
  - se encontramos uma abertura de parênteses, colocamos na pilha;
  - se encontramos um fechamento de parênteses, desempilhamos e copiamos na saída os operadores na pilha, até a abertura de parênteses correspondente (que é desempilhada e descartada).
- Ao final, desempilhamos e colocamos na saída os operadores que restarem na pilha.

Exemplos

- $A/(B+C)*D$
- $((A-(B*C))+D)$
- $1-2\hat{}\,3-(4+5*6)*7$

Avaliação de expressões em notação infixa

- Combinando os dois algoritmos anteriores, podemos fazer a avaliação de uma expressão em notação infixa usando duas pilhas (uma para operadores, outra para operandos).
- Exemplo: $1 - 2 \hat{} 3 - (4 + 5 * 6) * 7$

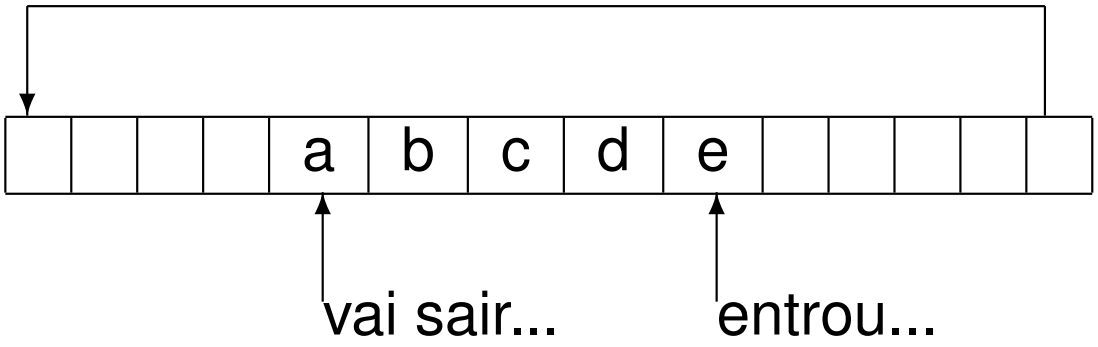
Entrada	Pilha Operadores	Pilha Operandos
1		1
–	–	1
2	–	1, 2
^	–, ^	1, 2
3	–, ^	1, 2, 3
–	–	-7
...	...	...

Filas

- Uma fila é uma estrutura em que o acesso é restrito ao elemento mais antigo.
- Operações básicas:
  - enqueue: inserir na fila;
  - dequeue: retirar da fila.

Arranjos circulares

A implementação mais comum de uma fila é por “arranjo circular”.



Implementação de Fila (1)

Fila baseada em arranjo (com amortização).

```
public class FilaAr {
    private Object arranjo[];
    private int tamanho;
    private int indiceVaiSair;
    private int indiceEntrou;
    static private final int DEFAULT = 10;

    public FilaAr() {
        arranjo = new Object[DEFAULT];
        esvazie();
    }

    public int tamanho() {
        return(tamanho);
    }
```

Implementação de Fila (2)

```
public void esvazie() {
    tamanho = 0;
    indiceVaiSair = 0;
    indiceEntrou=arranjo.length-1;
}

private int incremente(int indice) {
    indice++;
    if (indice == arranjo.length)
        indice = 0;
    return(indice);
}
```

Implementação de Fila (3)

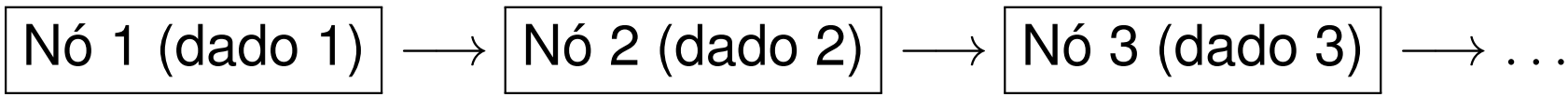
```
public void enqueue(Object x) {
      if (tamanho == arranjo.length)
         dupliqueArranjo();
      indiceEntrou = incremente(indiceEntrou);
      arranjo[indiceEntrou] = x;
      tamanho++;
}
private void dupliqueArranjo() {
   Object novo[] = new Object[2*arranjo.length];
   for(int i=0; i<tamanho; i++) {
      novo[i] = arranjo[indiceVaiSair];
      indiceVaiSair = incremente(indiceVaiSair);
   }
   arranjo = novo;
   indiceVaiSair = 0;
   indiceEntrou = tamanho - 1;
}
```

Implementação de Fila (4)

```
    public Object dequeue() {
        if (tamanho == 0)
            return(null);
        tamanho--;
        Object x = arranjo[indiceVaiSair];
        indiceVaiSair = incremente(indiceVaiSair);
        return(x);
    }
}
```

Lista Ligada

Uma alternativa a arranjos é a estrutura de lista ligada, na qual armazenamos dados em células interligadas.



Lista Ligada

- Esse tipo de estrutura é muito flexível e pode acomodar inserção e retirada de dados de locais arbitrários.
  - A vantagem desse tipo de estrutura é a flexibilidade permitida no uso da memória.
  - A desvantagem é que alocar memória é uma tarefa demorada (mais lenta que acesso a arranjos).

Implementação de Lista

Para definir uma lista ligada, precisamos primeiro definir o elemento armazenador (nó):

```
public class No {
    Object dado;
    No proximo;

    public No(Object x, No p) {
        dado = x;
        proximo = p;
    }
}
```

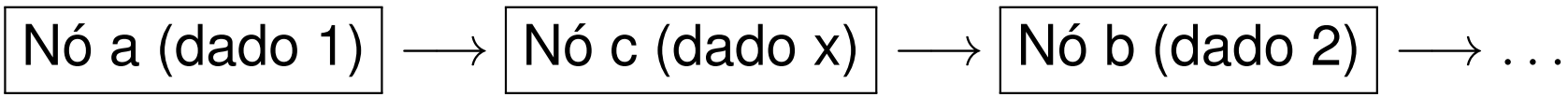
Inserção de nó

Considere inserir dado *x* após Nó a.


```
No c = new No(x, a.proximo);
a.proximo = c;
```

Alternativamente,

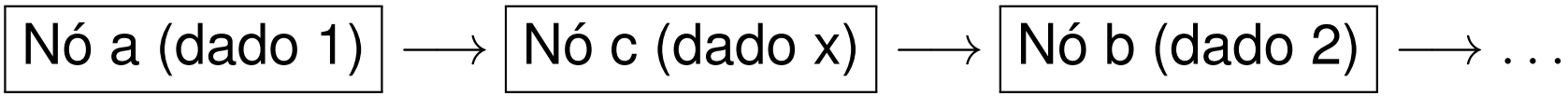
```
a.proximo = new No(x, a.proximo);
```



Remoção de nó, visita a sequência de nós

Remoção:

```
if (a.proximo != null)
    a.proximo = corrente.proximo.proximo;
```

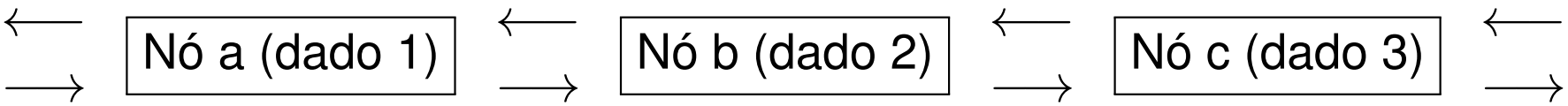


Para visitar todos os elementos de uma lista, de forma similar a um laço que percorre um arranjo:

```
for(No p = lista.primeiro; p!= null; p = p.proximo)
...
```

Lista Duplamente Encadeada

Uma estrutura interessante é o deque, composto por nós que apontam em duas direções:



Com essa estrutura é possível percorrer os dados em ambos os sentidos.

Usos de listas

A partir daí podemos implementar várias funcionalidades:

1. Pilhas;
2. Filas;
3. Vector: estrutura genérica de inserção/remoção em local arbitrário.

Note: as bibliotecas padrão da linguagem Java oferecem uma classe Vector, mas com implementação por arranjo (com amortização)!

Implementação de Pilha usando Lista

```
public class PilhaLi {
    private No topo;

    public PilhaLi() {
        topo = null;
    }

    public void push(Object x) {
        No n = new No(x);
        n.proximo = topo;
        topo = n;
    }

    public Object pop() {
        if(topo == null) return(null);
        Object t = topo.dado;
        topo = topo.proximo;
        return(t);
    }
}
```

Implementação de Fila usando Lista

```
public class FilaLi {
    private No vaiSair;
    private No entrou;

    public FilaLi() {
        vaiSair = null;
        entrou = null;
    }
    public void enqueue(Object x) {
        if(vaiSair == null) {
            entrou = new No(x);
            vaiSair = entrou;
        } else {
                entrou.proximo = new No(x);
                entrou = entrou.proximo;
            }
    }
    public Object dequeue() {
        if(vaiSair == null) return(null);
        Object t = vaiSair.dado;
        vaiSair = vaiSair.proximo;
        return(t);
    }
}
```