

Pilha

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich

**Baseado no material do Prof. Gustavo Batista*

<http://www.icmc.usp.br/~paulovic>
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

2 de outubro de 2013



Sumário

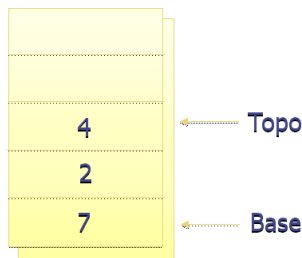
- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infix para posfixa

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infix para posfixa

Pilhas

- Pilhas são listas nas quais as inserções e remoções são feitas na mesma extremidade, chamada **topo**



Pilhas

- Nas pilhas elementos são adicionados no topo e removidos do topo
 - Política Last-In/First-Out (LIFO)
- Para lembrar o conceito de pilha, pode-se utilizar a associação com uma pilha de pratos ou xícaras

TAD Pilhas

- Operações principais
 - $empilhar(P, x)$: insere o elemento x no topo de P
 - $desempilhar(P)$: remove o elemento do topo de P , e retorna esse elemento

TAD Pilhas

- Operações auxiliares
 - *criar*(P): cria uma pilha P vazia
 - *apagar*(P): apaga a pilha P da memória
 - *topo*(P): retorna o elemento do topo de P , sem remover
 - *tamanho*(P): retorna o número de elementos em P
 - *vazia*(P): indica se a pilha P está vazia
 - *cheia*(P): indica se a pilha P está cheia (útil para implementações estáticas).

Aplicações

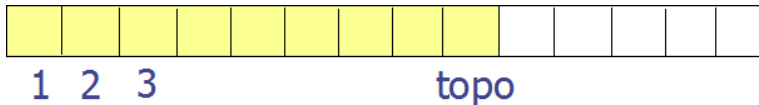
- Exemplos de aplicações de pilhas
 - O botão “back” de um navegador web ou a opção “undo” de um editor de textos
 - Controle de chamada de procedimentos
 - Estrutura de dados auxiliar em alguns algoritmos como a busca em profundidade

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infix para posfixa

Implementação Estática

- Implementação simples
- Uma variável mantém o controle da posição do topo, e pode ser utilizada também para informar o número de elementos na pilha



Definição de Tipos

```
1 typedef struct pilha_estatica PILHA_ESTATICA;  
2  
3 #define TAM 100  
4  
5 struct pilha_estatica {  
6     ITEM * vetor[TAM];  
7     int topo;  
8 };
```

Implementação Estática

```
1 PILHA_ESTATICA *criar_pilha() {  
2     PILHA_ESTATICA *pilha = (PILHA_ESTATICA *) malloc(sizeof (↵  
        PILHA_ESTATICA));  
3  
4     if (pilha != NULL) {  
5         pilha->topo = -1;  
6     }  
7     return pilha;  
8 }  
9  
10 int vazia(PILHA_ESTATICA *pilha) {  
11     return (pilha->topo == -1);  
12 }  
13  
14 int cheia(PILHA_ESTATICA *pilha) {  
15     return (pilha->topo == TAM-1);  
16 }  
17  
18 int tamanho(PILHA_ESTATICA *pilha) {  
19     return (pilha->topo+1);  
20 }
```

Implementação Estática

```
1  int empilhar(PILHA_ESTATICA *pilha, ITEM *item) {
2      if (!cheia(pilha)) {
3          pilha->topo++;
4          pilha->itens[pilha->topo] = item;
5          return 1;
6      }
7      return 0;
8  }
9
10 int desempilhar(PILHA_ESTATICA *pilha) {
11     if (!vazia(pilha)) {
12         limpar_item(pilha->itens[pilha->topo]);
13         pilha->topo--;
14         return 1;
15     }
16     return 0;
17 }
```

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica**
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infix para posfixa

Implementação Dinâmica

- O topo pode ser o início da lista
 - A implementação se torna mais eficiente, pois não há necessidade de percorrer a lista
 - Não há necessidade de utilizar listas duplamente ligadas
- Pode-se implementar as operações utilizando a abordagem de lista ligada simples

Definição de Tipos

```
1 typedef struct pilha_dinamica PILHA_DINAMICA;
2
3 typedef struct NO {
4     ITEM *item;
5     struct NO *anterior;
6 } NO;
7
8 struct pilha_dinamica {
9     NO *topo;
10    int tamanho;
11 };
```


Implementação Dinâmica

```
1 PILHA_DINAMICA *criar_pilha() {
2     PILHA_DINAMICA *pilha = (PILHA_DINAMICA *) malloc(sizeof (←
3     PILHA_DINAMICA));
4     if (pilha != NULL) {
5         pilha->topo = NULL;
6         pilha->tamanho = 0;
7     }
8     return pilha;
9 }
10 void apagar_pilha(PILHA_DINAMICA **pilha) {
11     if (!vazia(*pilha)) {
12         NO *paux = (*pilha)->topo;
13
14         while (paux != NULL) {
15             NO *prem = paux;
16             paux = paux->anterior;
17             apagar_no(prem);
18         }
19     }
20
21     free(*pilha);
22     *pilha = NULL;
23 }
```

Implementação Dinâmica

```
1  int vazia(PILHA_DINAMICA *pilha) {  
2      return (pilha->topo == NULL);  
3  }  
4  
5  int tamanho(PILHA_DINAMICA *pilha) {  
6      return pilha->tamanho;  
7  }  
8  
9  ITEM *topo(PILHA_DINAMICA *pilha) {  
10     if (!vazia(pilha)) {  
11         return pilha->topo->item;  
12     }  
13     return NULL;  
14 }
```

Implementação Dinâmica

```
1  int empilhar(PILHA_DINAMICA *pilha, ITEM *item) {
2      NO *pnovo = (NO *) malloc(sizeof (NO));
3      if (pnovo != NULL) {
4          pnovo->item = item;
5          pnovo->anterior = pilha->topo;
6          pilha->topo = pnovo;
7          pilha->tamanho++;
8          return 1;
9      }
10     return 0;
11 }
12
13 ITEM *desempilhar(PILHA_DINAMICA *pilha) {
14     if (!vazia(pilha)) {
15         NO *pno = pilha->topo;
16         pilha->topo = pilha->topo->anterior;
17         pno->anterior=NULL;
18         pilha->tamanho--;
19         return pno->item;
20     }
21     return NULL;
22 }
```

Estática versus Dinâmica

Operação	Estática	Dinâmica
Criar	$O(1)$	$O(1)$
Apagar	$O(n)$	$O(n)$
Empilhar	$O(1)$	$O(1)$
Desempilhar	$O(1)$	$O(1)$
Topo	$O(1)$	$O(1)$
Vazia	$O(1)$	$O(1)$
Tamanho	$O(1)$	$O(1)$ (com contador)

Estática versus Dinâmica

- Estática
 - Implementação simples
 - Tamanho da pilha definido a priori
- Dinâmica
 - Alocação dinâmica permite gerenciar melhor estruturas cujo tamanho não é conhecido a priori ou que variam muito de tamanho

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha**
 - Avaliação de expressões aritméticas
 - Conversão de infixa para posfixa

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infixa para posfixa

Aplicação de Pilhas

- Avaliação de expressões aritméticas
 - Notação infixa é ambígua
 - $A + B * C = ?$
 - Necessidade de precedência de operadores ou utilização de parênteses
 - Entretanto existem outras notações...

Aplicação de Pilhas

Notação polonesa (prefixa)

- Operadores precedem os operandos
- Dispensa o uso de parênteses
- $- * AB / CD = (A * B) - (C / D)$

Notação polonesa reversa (posfixa)

- Operandos sucedem os operadores
- Dispensa o uso de parênteses
- $AB * CD / - = (A * B) - (C / D)$

Aplicação de Pilhas

- Expressões na notação posfixa podem ser avaliadas utilizando uma pilha
 - A expressão é avaliada de esquerda para a direita
 - Os operandos são empilhados
 - Os operadores fazem com que dois operandos sejam desempilhados, o cálculo seja realizado e o resultado empilhado

Aplicação de Pilhas

- Por exemplo: $6\ 2\ /\ 3\ 4\ *\ +\ 3\ -\ =\ 6\ /\ 2\ +\ 3\ *\ 4\ -\ 3$

Símbolo	Ação	Pilha
6	empilhar	$P[6]$
2	empilhar	$P[2, 6]$
/	desempilhar, aplicar operador e empilhar	$P[(6/2)] = P[3]$
3	empilhar	$P[3, 3]$
4	empilhar	$P[4, 3, 3]$
*	desempilhar, aplicar operador e empilhar	$P[(3 * 4), 3] = P[12, 3]$
+	desempilhar, aplicar operador e empilhar	$P[(3 + 12)] = P[15]$
3	empilhar	$P[3, 15]$
-	desempilhar, aplicar operador e empilhar	$P[(15 - 3)] = P[12]$
	final, resultado no topo da pilha	$P[12]$

Implementação

- Código para ler uma string e extrair os elementos da mesma (operandos e operadores) – os elementos separados por espaço

```
1  #include <string.h>
2
3  int main () {
4      char str[] = "( 30 * ( 2 + 3 ) / 2 ) * 5";
5
6      char *token = strtok (str, " ");
7      while (token != NULL) {
8          //... processa 'token'
9
10         token = strtok (NULL, " ");
11     }
12
13     return 0;
14 }
```

Implementação

- Definição da estrutura Pilha que pode armazenar **int** ou **char**

```
1  #define TIPO_CHAR 0
2  #define TIPO_INT 1
3
4  typedef struct {
5      int tipo_union;
6      union {
7          int vint;
8          char vchar;
9      };
10 } ITEM;
11
12 ITEM *criar_item(int tipo_union, int valor);
13 void apagar_item(ITEM **item);
14 void imprimir_item(ITEM *item);
```

Implementação

```
1  ITEM *criar_item(int tipo_union, int valor) {
2      ITEM *item = (ITEM *) malloc(sizeof (ITEM));
3      if (tipo_union == TIPO_CHAR) {
4          item->vchar = valor;
5      } else {
6          item->vint = valor;
7      }
8      item->tipo_union = tipo_union;
9      return item;
10 }
11
12 void imprimir_item(ITEM *item) {
13     if (item != NULL) {
14         if (item->tipo_union == TIPO_CHAR) {
15             printf("%c\n", item->vchar);
16         } else {
17             printf("%d\n", item->vint);
18         }
19     }
20 }
```

Implementação

```
1 int avalia_posfixa(char *posfix_expr) {
2     ITEM *op1, *op2, *res;
3     PILHA_DINAMICA *pilha = criar_pilha();
4
5     char *token = strtok(posfix_expr, " ");
6     while (token != NULL) {
7         if (isdigit(token[0])) {
8             empilhar(pilha, criar_item(TIPO_INT, atoi(token)));
9         } else {
10             op2 = desempilhar(pilha);
11             op1 = desempilhar(pilha);
12
13             res = criar_item(TIPO_INT, 0);
14             switch (token[0]) {
15                 case '+': res->vint = op1->vint + op2->vint; break;
16                 case '-': res->vint = op1->vint - op2->vint; break;
17                 case '*': res->vint = op1->vint * op2->vint; break;
18                 case '/': res->vint = op1->vint / op2->vint; break;
19                 default: printf("Operador nao suportado <%c>!\n", token[0]);
20             }
21
22             apagar_item(&op1);
23             apagar_item(&op2);
24             empilhar(pilha, res);
25         }
26         token = strtok(NULL, " ");
27     }
28
29     res = desempilhar(pilha);
30     int resultado = res->vint;
31     apagar_item(&res);
32     apagar_pilha(&pilha);
33
34     return resultado;
35 }
```

Sumário

- 1 Conceitos Introdutórios
- 2 Implementação Estática
- 3 Implementação Dinâmica
- 4 Aplicações com Pilha
 - Avaliação de expressões aritméticas
 - Conversão de infixa para posfixa

Algoritmo para Conversão de Infixa para Posfixa

- Para chegar a esse algoritmo, observamos que a ordem dos operandos não é alterada quando a expressão é convertida: eles são copiados para a saída logo que encontrados

Algoritmo para Conversão de Infixa para Posfixa

- Para chegar a esse algoritmo, observamos que a ordem dos operandos não é alterada quando a expressão é convertida: eles são copiados para a saída logo que encontrados
- Por outro lado os operadores devem mudar de ordem, já que na posfixa eles aparecem logo depois dos seus operandos

Algoritmo para Conversão de Infixa para Posfixa

- Para chegar a esse algoritmo, observamos que a ordem dos operandos não é alterada quando a expressão é convertida: eles são copiados para a saída logo que encontrados
- Por outro lado os operadores devem mudar de ordem, já que na posfixa eles aparecem logo depois dos seus operandos
- Numa expressão posfixa, as operações são efetuadas na ordem em que aparecem. Logo, o que determina a posição de um operador é a prioridade que ele tem na forma infixa

Algoritmo para Conversão de Infixa para Posfixa

- Para chegar a esse algoritmo, observamos que a ordem dos operandos não é alterada quando a expressão é convertida: eles são copiados para a saída logo que encontrados
- Por outro lado os operadores devem mudar de ordem, já que na posfixa eles aparecem logo depois dos seus operandos
- Numa expressão posfixa, as operações são efetuadas na ordem em que aparecem. Logo, o que determina a posição de um operador é a prioridade que ele tem na forma infix
- Aqueles operadores de maior prioridade aparecem primeiro na expressão de saída

Algoritmo de Conversão

- ❶ Realize uma varredura na expressão infixa
 - ❶ Ao encontrar um operando, copie na expressão de saída
 - ❷ Ao encontrar o operador
 - ❶ Enquanto a pilha não estiver vazia e houver no seu topo um operador com prioridade maior ou igual ao encontrado e no topo não houver um parêntese de abertura, desempilhe o operador e copie-o na saída
 - ❷ Empilhe o operador encontrado
 - ❸ Ao encontrar um parêntese de abertura, empilhe-o
 - ❹ Ao encontrar uma parêntese de fechamento, remova os símbolos da pilha e copie-os na saída até que seja desempilhado o parêntese de abertura correspondente
- ❷ Ao final da varredura, esvazie a pilha, copiando os símbolos desempilhados para a saída

Conversão de Infixa para Posfixa

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

Símbolo	Ação	Pilha	Saída
A	copia para a saída	P:[]	A
*	pilha vazia, empilha	P:[*]	A
(sempre deve ser empilhado	P:[(, *]	A
B	copia para a saída	P:[(, *]	AB
+	prioridade maior, empilha	P:[+, (, *]	AB
C	copia para a saída	P:[+, (, *]	ABC
)	desempilha até achar '('	P:[*]	ABC+
/	prioridade igual, desempilha	P:[/]	ABC+*
D	copia para a saída	P:[/]	ABC+*D
	final, esvazia a pilha	P:[]	ABC+*D/

Algoritmo para Definir Prioridades dos Operadores

```
1  int prioridade(char operador) {  
2      switch (operador) {  
3          case '-':  
4          case '+': return 1;  
5          case '/':  
6          case '*': return 2;  
7          default:  
8              printf("Operador nao suportado <%c>!\n", operador);  
9              return -1;  
10     }  
11 }
```

Implementação

```
1 void strcat_str(char *dst, char *org) {  
2     strcat(dst, org);  
3     strcat(dst, " ");  
4 }  
5  
6 void strcat_char(char *dst, char c) {  
7     char aux[3] = {c, ' ', '\0'};  
8     strcat(dst, aux);  
9 }
```


Implementação

```
1 char *infixa_para_posfixa(char *infix_expr) {
2     ITEM *op;
3     PILHA_DINAMICA *pilha = criar_pilha();
4     static char posfix_expr[512];
5
6     char *token = strtok(infix_expr, " ");
7     while (token != NULL) {
8         if (isdigit(token[0])) {
9             strcat_str(posfix_expr, token);
10        } else if (token[0] == '(') {
11            empilhar(pilha, criar_item(TIPO_CHAR, token[0]));
12        } else if (token[0] == ')') {
13            while (!vazia(pilha) && topo(pilha)->vchar != '(') {
14                op = desempilhar(pilha);
15                strcat_char(posfix_expr, op->vchar);
16                apagar_item(&op);
17            }
18            op = desempilhar(pilha);
19            apagar_item(&op);
20        } else {
21            while (!vazia(pilha) && topo(pilha)->vchar != '(' &&
22                prioridade(topo(pilha)->vchar) >= prioridade(token[0])) {
23                op = desempilhar(pilha);
24                strcat_char(posfix_expr, op->vchar);
25                apagar_item(&op);
26            }
27            empilhar(pilha, criar_item(TIPO_CHAR, token[0]));
28        }
29        token = strtok(NULL, " ");
30    }
31
32    while (!vazia(pilha)) {
33        op = desempilhar(pilha);
34        strcat_char(posfix_expr, op->vchar);
35        apagar_item(&op);
36    }
37    apagar_pilha(&pilha);
38
39    return posfix_expr;
40 }
```