Pilha

Estruturas de Dados I

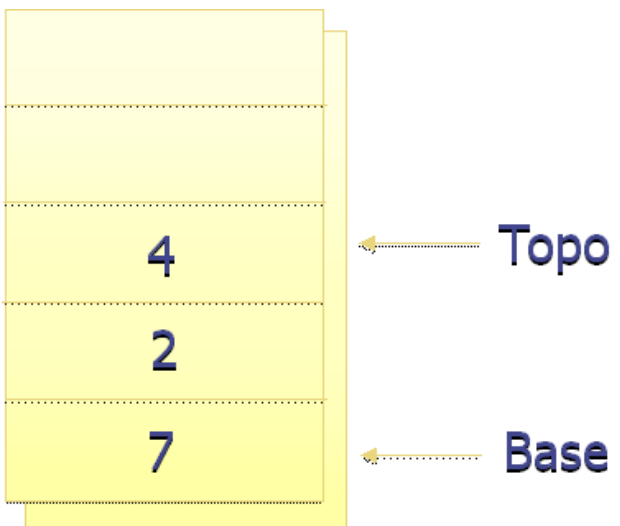
Departamento de Computação

Universidade Federal de São Carlos (UFSCar)

Sumário

Pilhas

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



Pilhas

- Nas pilhas as operações de inserção/remoção são realizadas na ordem last-in/first-out, por isso são chamadas de 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
  - *topo(P)*: retorna o elemento do topo de **P**, sem remover
  - *contar(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).

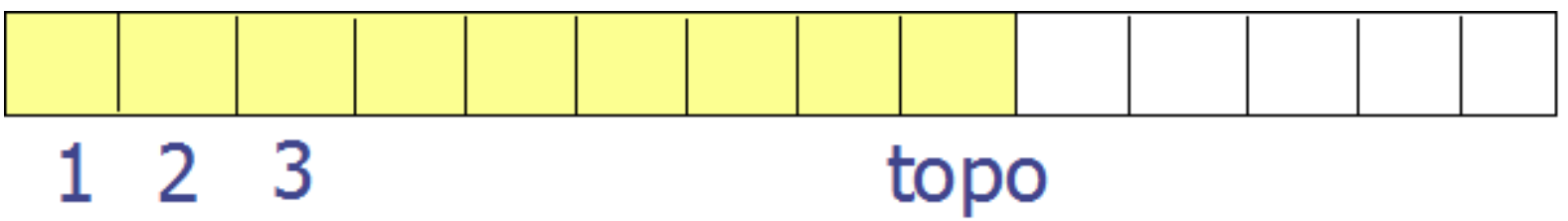
Aplicações

- Exemplos de aplicações de pilhas
  - O botão "back" de um navegado 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

Implementação Estática

- Implementação simples
- Inserções e remoções realizadas no fim da estrutura (posição topo)
- 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  #define TAM 100
2
3  typedef struct {
4    int valor;
5  } ITEM;
6
7  typedef struct {
8    ITEM itens[TAM];
9    int topo;
10 } PILHA_ESTATICA;
```

Implementação Estática

```
1  void criar(PILHA_ESTATICA *pilha) {
2    pilha->topo = -1;
3  }
4
5  int vazia(PILHA_ESTATICA *pilha) {
6    return (pilha->topo == -1);
7  }
8
9  int cheia(PILHA_ESTATICA *pilha) {
10   return (pilha->topo == TAM-1);
11 }
12
13 int contar(PILHA_ESTATICA *pilha) {
14   return (pilha->topo+1);
15 }
```

Implementação Estática

```
1  int empilhar(PILHA_ESTATICA *pilha, ITEM *item) {
2    if (!cheia(pilha)) {
3      pilha->topo++; //incremento o topo
4      pilha->itens[pilha->topo] = *item; //insiro o item
5      return 1;
6    }
7
8    return 0;
9  }
10
11 int desempilhar(PILHA_ESTATICA *pilha, ITEM *item) {
12   if (!vazia(pilha)) {
13     *item = pilha->itens[pilha->topo]; //recupero o item no topo
14     pilha->topo--; //decremento o topo
15     return 1;
16   }
17
18   return 0;
19 }
```

Sumário

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 {
2    int valor;
3  } ITEM;
4
5  typedef struct NO {
6    ITEM item;
7    struct NO *anterior;
8  } NO;
9
10 typedef struct {
11   int contador;
12   NO *topo;
13 } PILHA_DINAMICA;
```

Implementação Dinâmica

```
1  void criar(PILHA_DINAMICA *pilha) {
2    pilha->contador = 0;
3    pilha->topo = NULL;
4  }
5
6  int vazia(PILHA_DINAMICA *pilha) {
7    return (pilha->topo == NULL);
8  }
9
10 int contar(PILHA_DINAMICA *pilha) {
11   return pilha->contador;
12 }
13
14 void limpar(PILHA_DINAMICA *pilha) {
15   NO *paux = pilha->topo;
16
17   while (paux != NULL) {
18     NO *prem = paux;
19     paux = paux->anterior;
20     free(prem);
21   }
22 }
```

Implementação Dinâmica

```
1  int empilhar(PILHA_DINAMICA *pilha, ITEM *item) {
2    NO *pnovo = (NO *)malloc(sizeof(NO));
3
4    if (pnovo != NULL){
5      pnovo->item = *item; //armazena o novo item
6      pnovo->anterior = pilha->topo; //aponta o nó anterior
7      pilha->topo = pnovo; //atualiza o topo
8      pilha->contador++; //incrementa o contador
9      return 1;
10   }
11
12   return 0;
13 }
14
15 int desempilhar(PILHA_DINAMICA *pilha, ITEM *item) {
16   if (!vazia(pilha)) {
17     *item = pilha->topo->item; //recupera o item no topo
18     NO *paux = pilha->topo;
19     pilha->topo = pilha->topo->anterior; //topo recebe o anterior do topo
20     pilha->contador--; //decrementa o contador
21     free(paux); //remove da memória
22     return 1;
23   }
24
25   return 0;
26 }
```

Estática versus Dinâmica

Operação	Estática	Dinâmica
Cria	$O(1)$	$O(1)$
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$
Topo	$O(1)$	$O(1)$
Vazia	$O(1)$	$O(1)$
Conta	$O(1)$	$O(1)$ (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

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	$P[12]$

Sumário

Algoritmo de Conversão

1. Realize uma varredura na expressão infixa
   1. Ao encontrar um operando, copie na expressão de saída
   2. Ao encontrar o operador
      1. Enquanto a pilha não estiver vazia e houver no seu topo um operador com prioridade maior ou igual ao encontrado, desempilhe o operador e copie-o na saída
      2. Empilhe o operador encontrado
   3. Ao encontrar um parêntese de abertura, empilhe-o
   4. 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
2. 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/