```
void inverte_fila (FILA_SEQ *f)
{
   PILHA_SEQ p;
   cria_pilha(&p);
   while (!eh_vazia(f))
      push(&p,cons_ret(f));
   while (!eh_vazia_pilha(&p))
      ins(f, top_pop(&p));
}
```

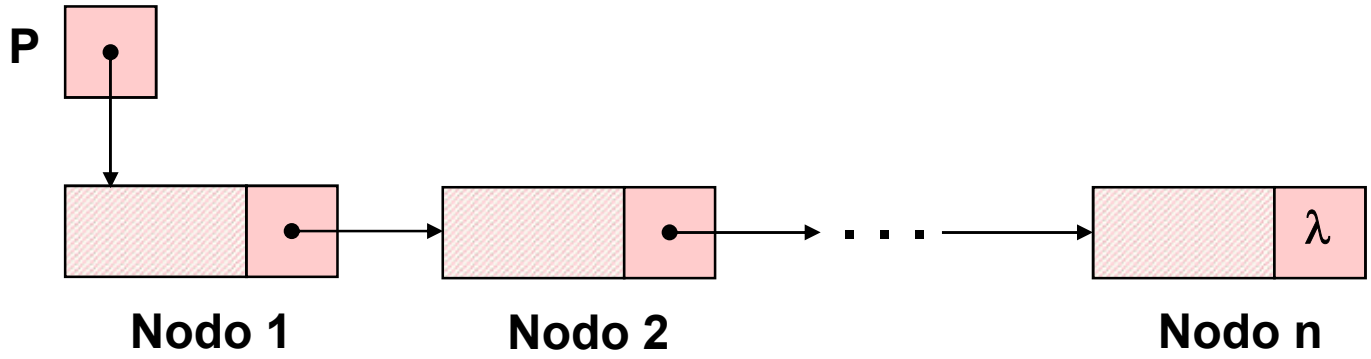
Pilha - Alocação Encadeada

Como já discutimos, a alocação sequencial apresenta algumas desvantagens. Em virtude disso, podemos nós utilizar de uma lista encadeada para armazenarmos uma pilha, assim como fizemos com as filas.

Como todas as operações ocorrem numa das extremidades da lista, a representação da pilha se reduz a um único ponteiro para o primeiro nodo da lista.

UNIVASF

Alocação Encadeada



A implementação das operações é trivial. Para fazer uma inserção, basta alocar um nodo para o novo valor, ligá-lo ao primeiro nodo da lista e fazer o ponteiro apontar para o novo nodo. Uma retirada exige apenas que o ponteiro passe a apontar para o segundo nodo da lista (ou ser anulado, se houver

Alocação Encadeada

apenas um nodo). Uma consulta exige apenas a recuperação do valor do primeiro nodo. OBS. : em uma retirada o espaço de memória ocupado pelo primeiro nodo deve ser liberado.

Desta forma, definiremos e implementaremos, agora, o TAD PILHA_ENC (de valores inteiros).

UNIVASF

```
/*Definição do TAD PILHA_ENC*/
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação cria_pilha() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
void cria_pilha (PILHA_ENC *pp)
{
   *pp=NULL;
}
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação eh_vazia() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
int eh_vazia (PILHA_ENC p)
{
   return (!p);
}
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação push() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
void push (PILHA_ENC *pp, int v)
{
   NODO *novo;
   novo = (NODO *) malloc (sizeof(NODO));
   if (!novo)
   {
      printf ("\nERRO! Memoria insuficiente!\n");
      exit (1);
   }
   novo->inf = v;
   novo->next = *pp;
   *pp=novo;
}
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação top() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
int top (PILHA_ENC p)
{
   if (eh_vazia(p))
   {
      printf ("\nERRO! Consulta em pilha vazia!\n");
      exit (2);
   }
   else
      return (p->inf);
}
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação pop() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
void pop (PILHA_ENC *pp)
{
   if (eh_vazia(*pp))
   {
      printf ("\nERRO! Retirada em pilha vazia!\n");
      exit (3);
   }
   else
   {
      NODO *aux=*pp;
      *pp=(*pp)->next;
      free (aux);
   }
}
```

Pilha - Alocação Encadeada

Com base no que foi visto implemente a operação top_pop() que compõem o TAD PILHA_ENC.

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
int top_pop (PILHA_ENC *pp) {
   if (eh_vazia(*pp))
   {
      printf ("\nERRO! Consulta e retirada em pilha vazia!\n");
      exit (4);
   }
   else
   {
      int v=(*pp)->inf;
      NODO *aux=*pp;
      *pp=(*pp)->next;
      free (aux);
      return (v);
   }
}
```

Notações: in, pré e posfixada

Examinaremos agora uma importante aplicação que ilustra a utilização da estrutura de dados pilha e das operações definidas a partir dela.

O exemplo é, em si mesmo, um relevante tópico da computação.

Considerando a soma de A mais B. Imaginamos a aplicação do **operador** "+" sobre os **operandos** A e B e escrevemos a soma como A + B.

Essa representação particular é chamada infixada. Existem duas notações alternativas para expressar a soma de A e B usando os símbolos A, B e +.

UNIVASF

Notações: in, pré e posfixada

São elas:

+ A B prefixada

A B + posfixada

Analisando expressões infixadas um pouco mais complexas, como, por exemplo, A + B * C.

Notamos a necessidade da definição de precedência entre os operadores (em casos em que é preciso alterar a ordem de precedência pré estabelecida se utilizam parênteses) visando eliminar a ambiguidade, tornando a tarefa computacional menos simples.

Notações: in, pré e posfixada

A representação pré e posfixada para expressões aritméticas é mais conveniente do ponto de vista computacional.

Para ilustrarmos os diferentes tipos de representações, utilizaremos em nosso exemplos cinco operações binárias: adição, subtração, multiplicação, divisão e exponenciação.

Vamos fazer algumas conversões da forma infixada para a prefixada e posfixada.

UNIVASF

Notações: in, pré e posfixada

Forma Infixada	Forma Prefixada
A + B - C	- + ABC
(A + B) * (C - D)	* + AB - CD
A ^ B * C - D + E / F / (G + H)	+ - * ^ ABCD / / EF + GH
((A + B) * C - (D - E)) ^ (F + G)	^ -* + ABC - DE + FG
A - B / (C * D ^ E)	- A / B * C ^ DE

Notações: in, pré e posfixada

Forma Infixada	Forma Posfixada
A + B - C	AB + C -
(A + B) * (C - D)	AB + CD - *
A ^ B * C - D + E / F / (G + H)	AB ^ C * D - EF / GH + / +
((A + B) * C - (D - E)) ^ (F + G)	AB + C * DE - - FG + ^
A - B / (C * D ^ E)	ABCDE ^ * / -

Notações: in, pré e posfixada

Como avaliar uma expressão posfixada?

A resposta a esta pergunta, vem de uma análise. Estamos tratando de operações binárias.

Portanto, devemos avaliar a expressão da esquerda para direita, em busca de dois operandos consecutivos seguidos de um operador, neste momento aplicamos a operação sobre os respectivos operandos, substituindo-os, na expressão, pelo resultado e continuando em seguida com a análise.

UNIVASF

Notações: in, pré e posfixada – Exercício

Vamos agora escrever um algoritmo para avaliar uma expressão aritmética posfixada.

Devemos inicialmente definir a entrada, ou seja, de que forma representaremos nossa expressão.

Sendo assim, vamos imaginar uma string representando a expressão posfixada. Para não tornarmos o algoritmo muito complexo e não nos desviarmos do nosso foco principal. Desta forma, nossos operandos serão positivos e compostos por apenas um dígito.

UNIVASF