AED1 - Aula 14
Pilha implementada em vetor, aplicação com parênteses e colchetes, pilha de execução, relação de pilha com recursão

Pilha

Uma pilha (no inglês stack) é uma lista dinâmica,
- ou seja, uma sequência em que elementos podem ser removidos e inseridos,
- mas que possui regras bem específicas de funcionamento.

Em particular, as seguintes regras devem ser obedecidas:
- uma operação de remoção sempre remove o elemento do fim da sequência,
- uma operação de inserção sempre insere o elemento no fim da sequência.

Chamamos a última posição de uma pilha de topo, assim
- as operações de inserção, remoção e consulta
    - sempre são feitas no topo da pilha.

Costumamos resumir o comportamento de uma pilha na frase
- o último a entrar é o primeiro a sair.
- Por isso, pilhas também são conhecidas por LIFO,
    - acrônimo do inglês Last-In-First-Out.

Implementação de pilha usando vetor

Uma pilha s é armazenada em um vetor de tamanho n
- alocado estática ou dinamicamente.

Um inteiro t indica o topo da pilha
- que é a posição do próximo elemento

Note que t corresponde ao número de elementos presentes na pilha.
- Em particular,
    - se t = 0 a pilha está vazia
    - se t = n a pilha está cheia.

Para empilhar um elemento x fazemos
- s[t++] = x;
- que corresponde a
    - s[t] = x; t = t + 1;
- note que esta operação não é segura se a pilha estiver cheia,

    - i.e., se t = n.

Para desempilhar um elemento e armazená-lo em x fazemos

- x = s[--t];
- que corresponde a
    - t = t - 1; x = s[t];
- note que esta operação não é segura se a pilha estiver vazia,
    - i.e., se t = 0.

Para consultar o valor do elemento no topo da pilha fazemos

- s[t - 1];

Note que todas as operações de manipulação da pilha

- levam tempo constante, i.e., O(1).

Se o número de elementos crescer muito, a pilha pode ficar cheia.

- Neste caso, uma alternativa é redimensionar a pilha, por exemplo,
    - alocando um vetor com o dobro do tamanho do anterior
    - e copiando todos os elementos do vetor anterior para esse novo,
        - preservando a ordem dos elementos.

Aplicação de pilha para verificação de parênteses e colchetes

Sequência bem formada:

- (( ) [( )])

Sequência mal formada:

- ([ )]

Definição geral recursiva:

- sendo S uma sequência de parênteses e colchetes bem formada, temos
- S = { sequência vazia,
  (S) S,
  [S] S }

Códigos:

```
#define N 100
char pilha[N];
int t;

void criapilha(void)
{
   t = 0;
```

```
}

void empilha(char y)
{
   pilha[t++] = y;
}

char desempilha(void)
{
   return pilha[--t];
}

int pilhavazia(void)
{
   return t <= 0;
}

// Esta função devolve 1 se a string ASCII s
// contém uma sequência bem-formada de
// parênteses e colchetes e devolve 0 se
// a sequência é malformada.
int bemFormada(char s[])
{
   criapilha();
   for (int i = 0; s[i] != '\0'; ++i)
   {
       char c;
       switch (s[i])
       {
       case ')':
           if (pilhavazia())
               return 0;
           c = desempilha();
           if (c != '(')
               return 0;
           break;
       case ']':
           if (pilhavazia())
               return 0;
           c = desempilha();
           if (c != '[')
               return 0;
           break;
       default:
           empilha(s[i]);
       }
   }
   return pilhavazia();
```

```
}
```

Note que nesta aplicação uma alternativa para a pilha nunca estourar seu tamanho

- seria alocar para ela um vetor do tamanho da string de entrada.

Quizzes:

- Como modificar o algoritmo/código anterior para que ele passe a verificar sequências bem formadas envolvendo { }, além de ( ) e [ ]?
- Como modificar as operações empilha e desempilha anteriores para que indiquem erro caso a pilha esteja cheia ou vazia?
- Como modificar a operação empilha anterior para realocar a pilha num vetor maior caso a pilha esteja cheia?

Pilha de execução de um programa

A pilha de execução de um programa é usada para armazenar:

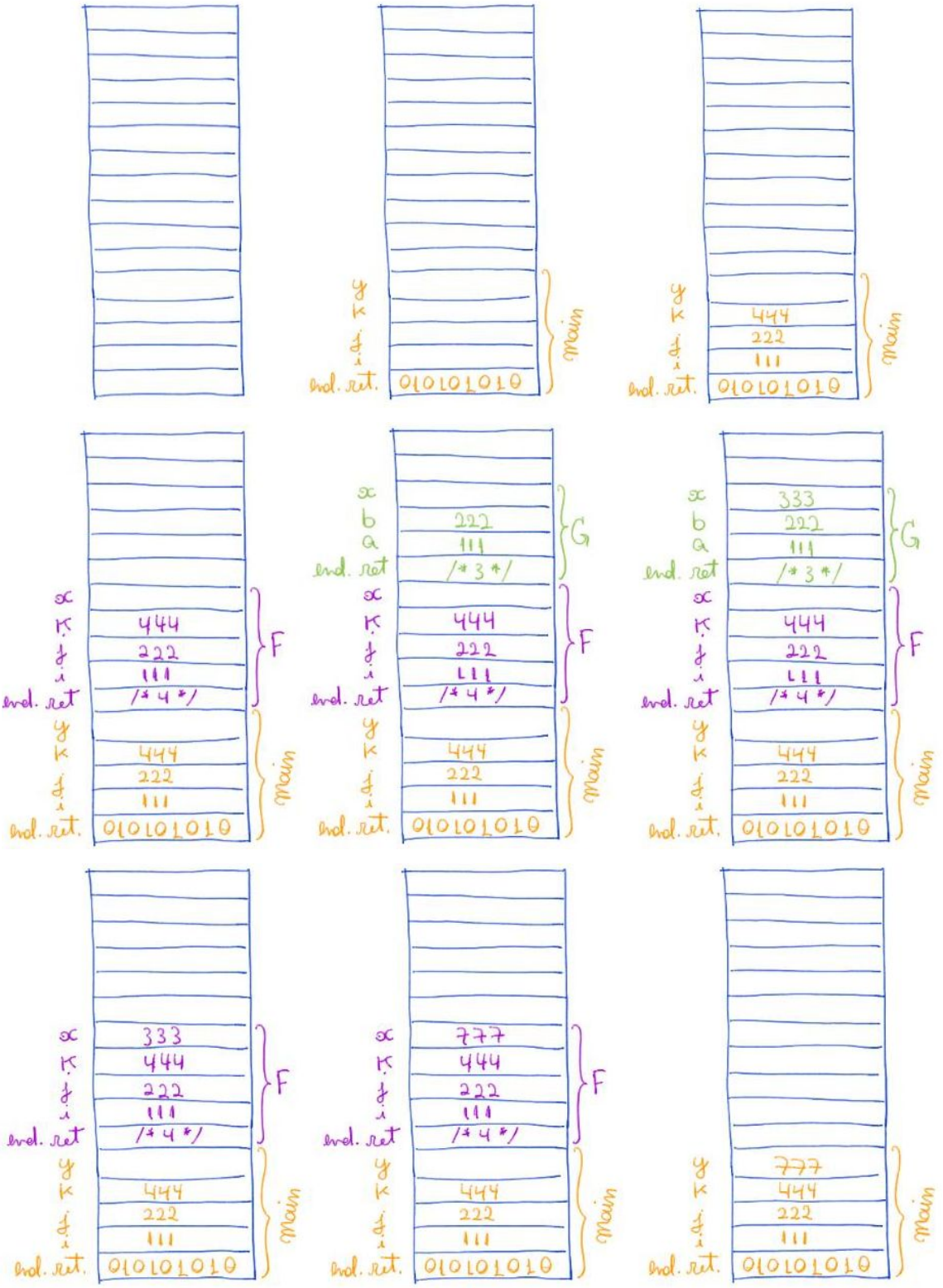
- variáveis locais das funções ativas
- parâmetros das funções ativas
- endereço de retorno para o ponto do código em que a função foi chamada
- cálculo de expressões

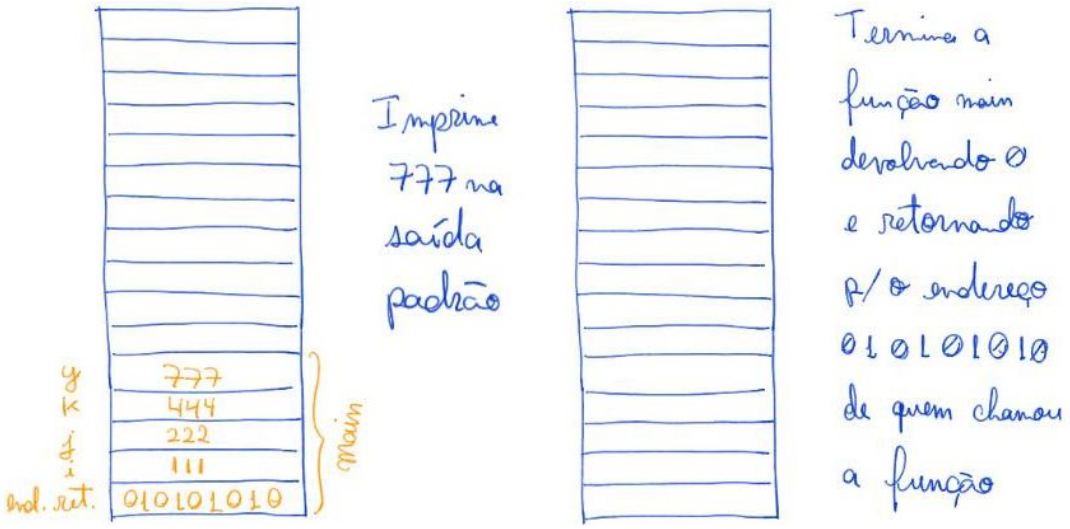
Códigos:

```c
int G(int a, int b)
{
   int x;
   x = a + b;
   return x;
}

int F(int i, int j, int k)
{
   int x;
   x = /*2*/ G(i, j) /*3*/;
   x = x + k;
   return x;
}

int main(void)
{
   int i, j, k, y;
   i = 111;
   j = 222;
   k = 444;
   y = /*1*/ F(i, j, k) /*4*/;
   printf("%d\n", y);
```

```
    return EXIT_SUCCESS;
}
```

Ilustração da pilha de execução deste código:





Quizz:

- onde são armazenadas as variáveis alocadas dinamicamente?
    - na outra ponta do espaço de memória.

Relação entre pilhas e recursão

Existe uma relação muito íntima entre pilhas e recursão.

- De fato, sempre é possível converter sistematicamente
    - um algoritmo recursivo em um algoritmo iterativo
        - usando uma pilha explícita.
- Olhando por outro ângulo, um algoritmo recursivo
    - se vale da pilha de execução para atacar problemas
        - de maneiras que não seriam viáveis sem uma pilha.

Como exemplo,

- considere o problema de verificar se uma sequência é bem formada.
- Podemos construir um algoritmo recursivo para este problema
    - utilizando a definição recursiva do mesmo

Código:

```
int bemFormadaR(char s[], int *i)
{
   int sol;
   if (s[*i] == '(')
   {
       *i = *i + 1;
       sol = bemFormadaR(s, i) && s[*i] == ')';
       *i = *i + 1;
       return sol && bemFormadaR(s, i);
   }
```

```
    if (s[*i] == '[')
    {
        *i = *i + 1;
        sol = bemFormadaR(s, i) && s[*i] == ']';
        *i = *i + 1;
        return sol && bemFormadaR(s, i);
    }
    return 1;
}

int bemFormada2(char s[])
{
    int i = 0;
    return bemFormadaR(s, &i) && s[i] == '\0';
}
```

Bônus

Como converter um algoritmo recursivo para um iterativo?

- Primeiro um exemplo do caso simples, quando tratasse de recursão caudal.
- Depois um exemplo de recursão geral, usando pilha.

Recursão caudal:

- caso em que a chamada recursiva é a última coisa a acontecer antes do final da função.

Algoritmo recursivo para encontrar o máximo:

```
int buscaR(int x, int *v, int n)
{
    if (n == 0)
        return -1;
    if (x == v[n - 1])
        return n - 1;
    return buscaR(x, v, n - 1);
}
```

Conversão para iterativo:

```
int buscaI(int x, int *v, int n)
{
    while (1)
    {
        if (n == 0)
            return -1;
        if (x == v[n - 1])
            return n - 1;
        n = n - 1; /* atualiza o valor dos parâmetros que mudam na chamada recursiva */
```

```
    }
}
```

Note que, recursão caudal é facilmente convertida para algoritmo iterativo,

- sem uso de pilha,
- pois quando a chamada recursiva termina,
    - não há mais nada que fazer na função que a chamou.
- É exatamente para tratar o retorno da recursão que a pilha é essencial.

Recursão geral:

Algoritmo recursivo para somar os elementos positivos de um vetor.

```
int somaPositivosR(int *v, int n)
{
   int res; /* variável supérflua que ajuda a entender a conversão */
   if (n == 0)
   { /* caso base */
       res = 0;
       return res;
   }
   /* 111 - marcador do inicio da função (após caso base) */
   if (v[n - 1] > 0)
   {
       res = somaPositivosR(v, n - 1); /* 222 - marcador da volta da primeira recursão */
       res += v[n - 1];
       return res;
   }
   else
   {
       res = somaPositivosR(v, n - 1); /* 333 - marcador da volta da segunda recursão */
       return res;
   }
}
```

Conversão para iterativo com pilha:

```
int somaPositivosI(int *v, int n)
{
   int res;
   int addr = 1; /* variável auxiliar para saber em que ponto voltar na função */
   int *s;
   int t; /* variáveis para pilha e topo */
   s = malloc(n * sizeof(int));
   t = 0;
   s[t++] = 0;
   s[t++] = n; /* colocando endereço inicial arbitrário e valor original de n na pilha */
   while (t > 0)
   {
```

```
        if (n == 0)
        { /* caso base */
            res = 0;
            n = s[--t];
            addr = s[--t]; /* corresponde ao return */
        }
        else
        {
            switch (addr)
            {
            case 111: /* inicio da função (após caso base) */
                if (v[n - 1] > 0)
                {
                    s[t++] = 222;
                    s[t++] = n; /* armazena variáveis na pilha para voltar */
                    addr = 111;
                    n = n - 1; /* atualiza variáveis para chamada recursiva */
                }
                else
                {
                    s[t++] = 333;
                    s[t++] = n; /* armazena variáveis na pilha para voltar */
                    addr = 111;
                    n = n - 1; /* atualiza variáveis para chamada recursiva */
                }
                break;
            case 222: /* volta da primeira recursão */
                res += v[n - 1];
                n = s[--t];
                addr = s[--t]; /* corresponde ao return */
                break;
            case 333:      /* volta da segunda recursão */
                res = res; /* supérfluo para manter o padrão na conversão */
                n = s[--t];
                addr = s[--t]; /* corresponde ao return */
                break;
            }
        }
    }
    free(s);
    return res;
}
```

Notem que, se o valor de res não fosse apenas acumulado ao longo das chamadas/iterações,

- ele também teria que ser salvo na pilha e restaurado desta,
- da mesma forma que fazemos com o endereço de retorno addr
    - e com o valor de n.