DCA0204, Módulo 4
Pilhas e recursão

Daniel Aloise

baseado em slides do prof. Leo Liberti, École Polytechnique, França

DCA, UFRN

Sumário

- Chamadas de função
- Pilhas e aplicações
- Recursão

Conhecimento mínimo

- Uma função f pode chamar outra função g: toda vez que isto acontece, o endereço A_g da instrução de f logo depois da instrução `call` g é armazenada na memória; quando g termina, o controle é transferido para A_g.
- Uma pilha é uma estrutura de dados onde você pode ler (e deletar) apenas o último elemento que você adicionou.
- Recursão é quando uma função f chama a si própria. Uma vez que isto permite essencialmente a execução repetida do código de f, a recursão é similar a um loop. Algumas vezes é mais conveniente escrever código usando recursão do que loops.

Chamadas de função

O que é uma chamada de função?

Uma receita é um programa, você é a CPU, sua cozinha é a memória

Receita de salada de nozes

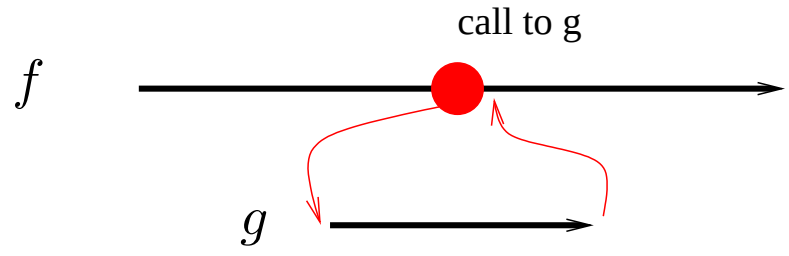
1. adicione a salada
2. adicione as nozes
3. adicione o vinagrete
4. misture e sirva

- Parece simples, mas quando você chega no Step 3 você percebe que a fim de se adicionar o vinagrete você precisa *prepará-lo antes!*
- Então você deixa tudo como está, mistura óleo e vinagre, adiciona sal, e recomeça a receita de onde você a deixou.
- Você apenas chamou uma função.

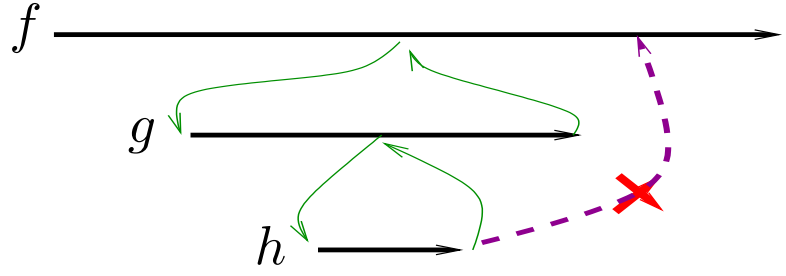
O essencial sobre funções

- Uma chamada de função é um desvio da ordem sequencial de instruções.
  - você precisa saber para onde ir depois
  - você precisa armazenar o endereço da instrução para a qual retornar depois que a função terminar.

f chama g:



- Assuma que f chama g, g chama h, e h está em execução
- A fim de que f retome o controle, g precisa ter terminado primeiro



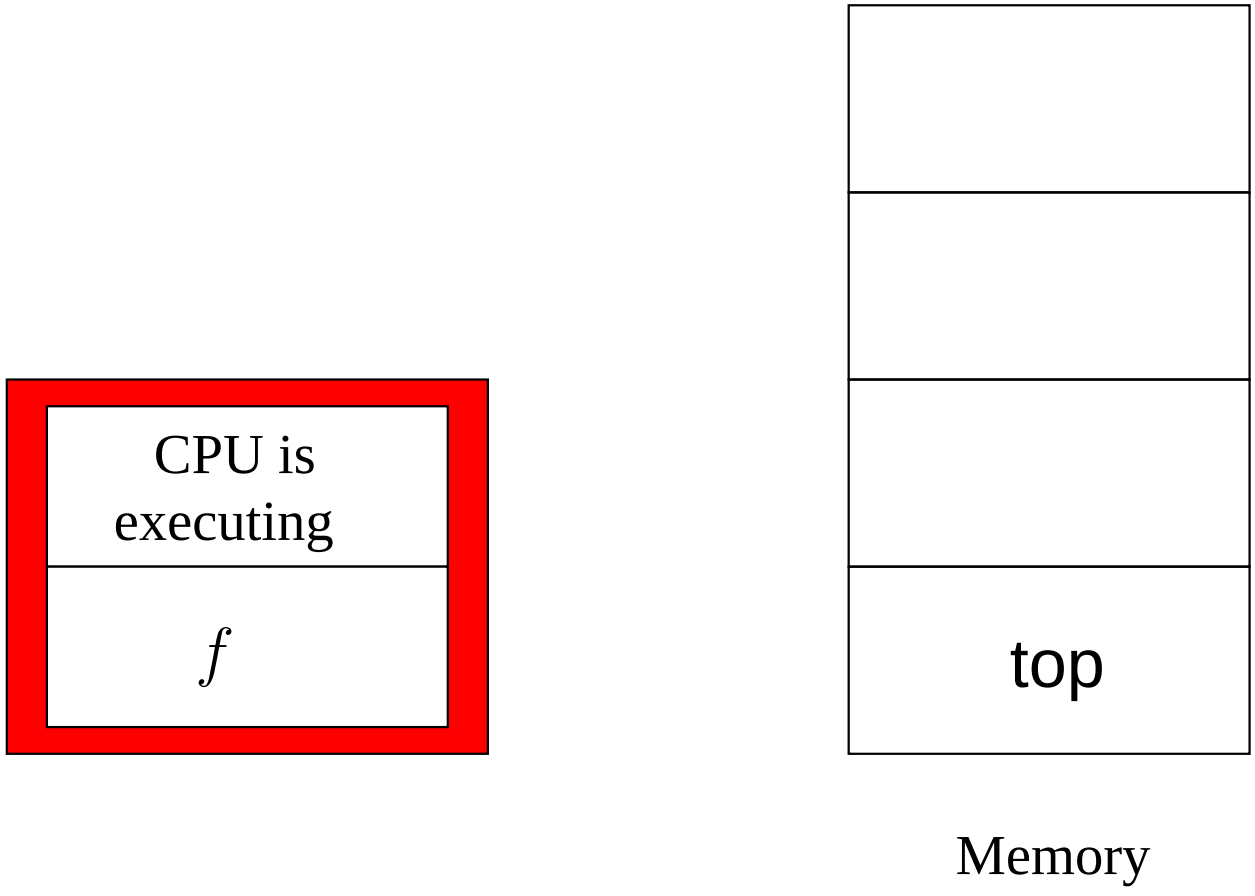
h não pode passar o controle para f diretamente

Salvando o estado

- Toda função define um “escopo” (denote uma entidade x definida dentro de uma função f por $f::x$)
- Se f chama g, ambos podem definir uma variável local x, mas $f::x$ e $g::x$ se referem a diferentes células de memória.
- Antes de chamar g, f precisa salvar o seu *estado atual*:
  - o nome e endereço de cada variável local em f
  - o endereço da instrução logo depois de “call g” em f
- Quando g termina, o estado atual de f é recuperado, e f retoma o controle
- Necessita de uma ED para salvar os estados atuais
- Como chamadas de funções são muito comuns, esta estrutura precisa ser tão simples e eficiente quanto possível

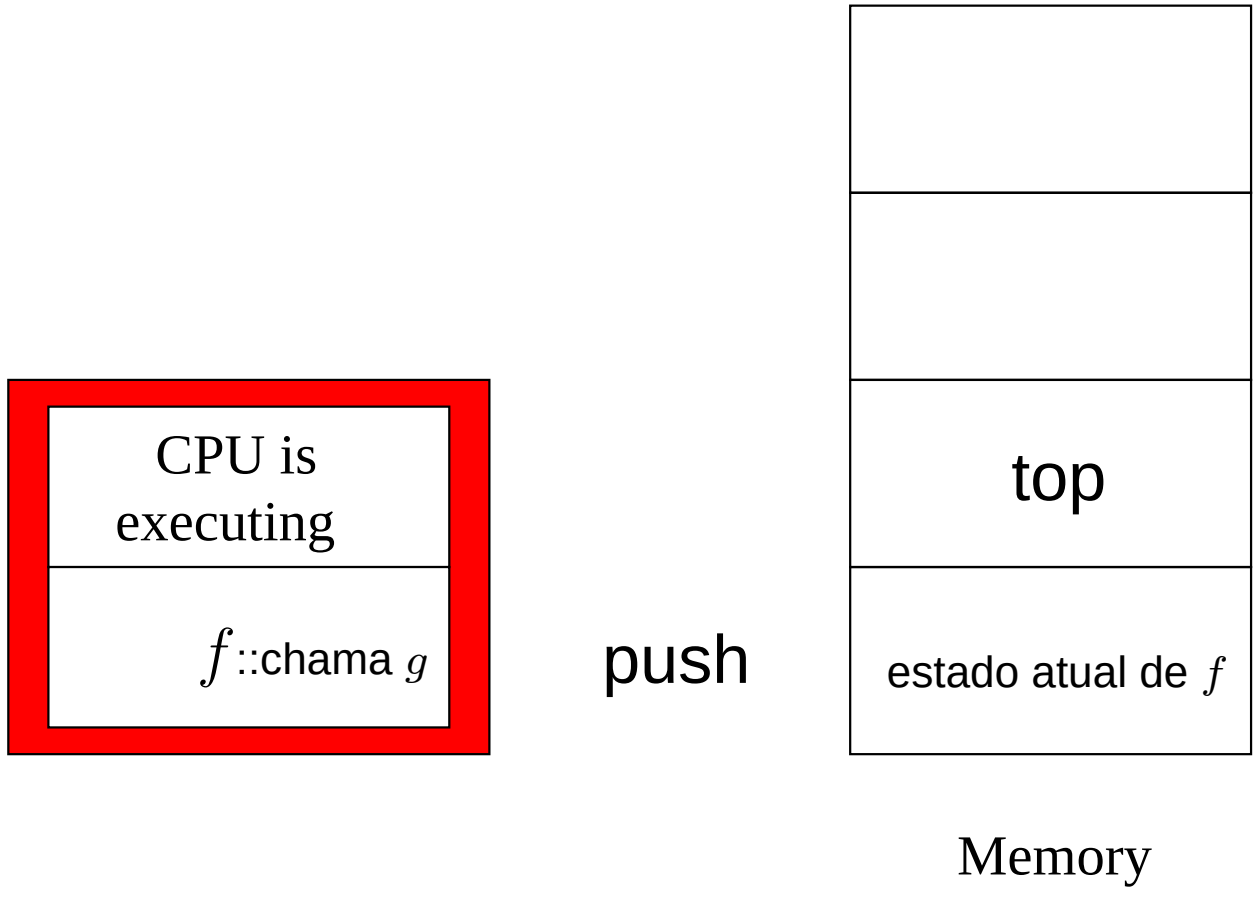
Estados atuais são salvos em uma pilha

f chama g chama h



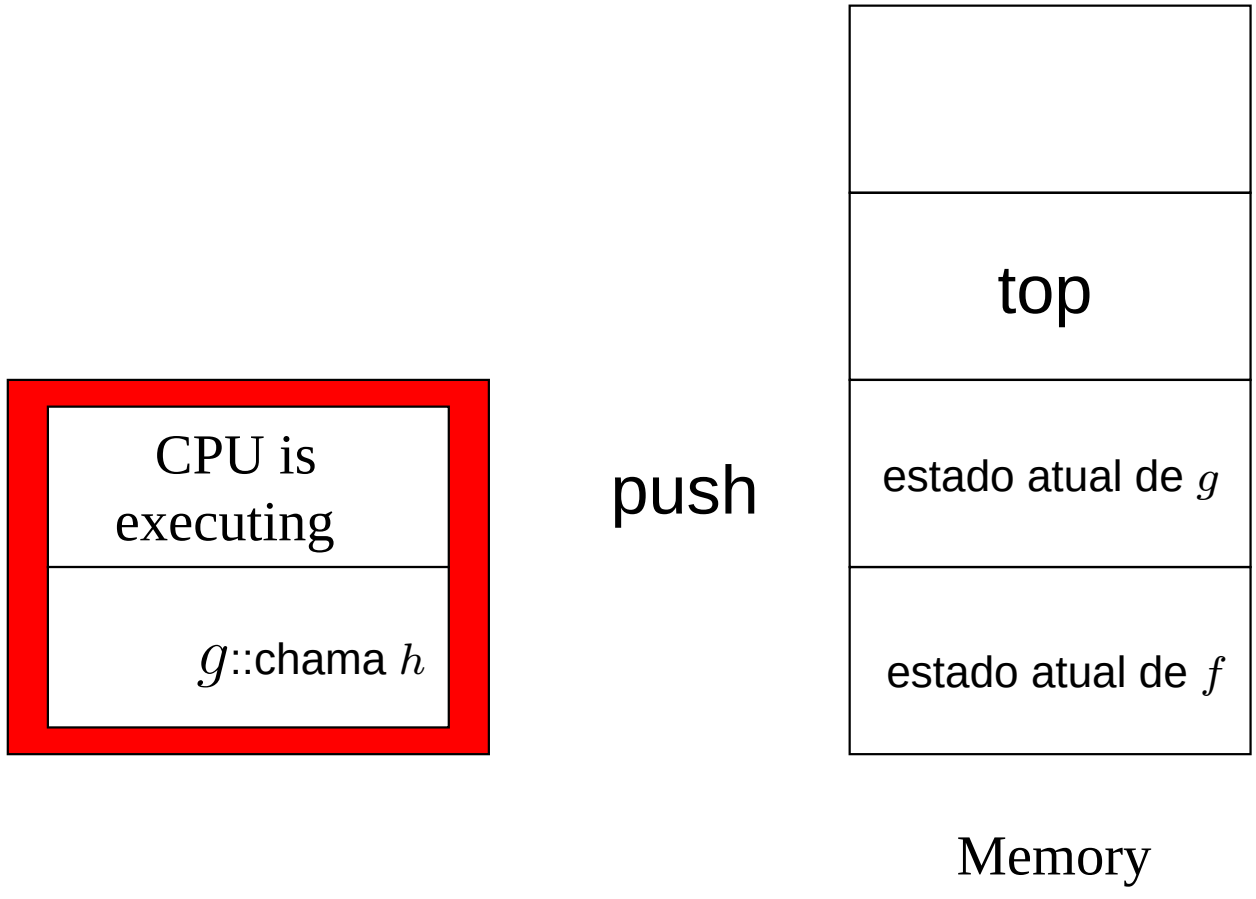
Estados atuais são salvos em uma pilha

f chama g chama h



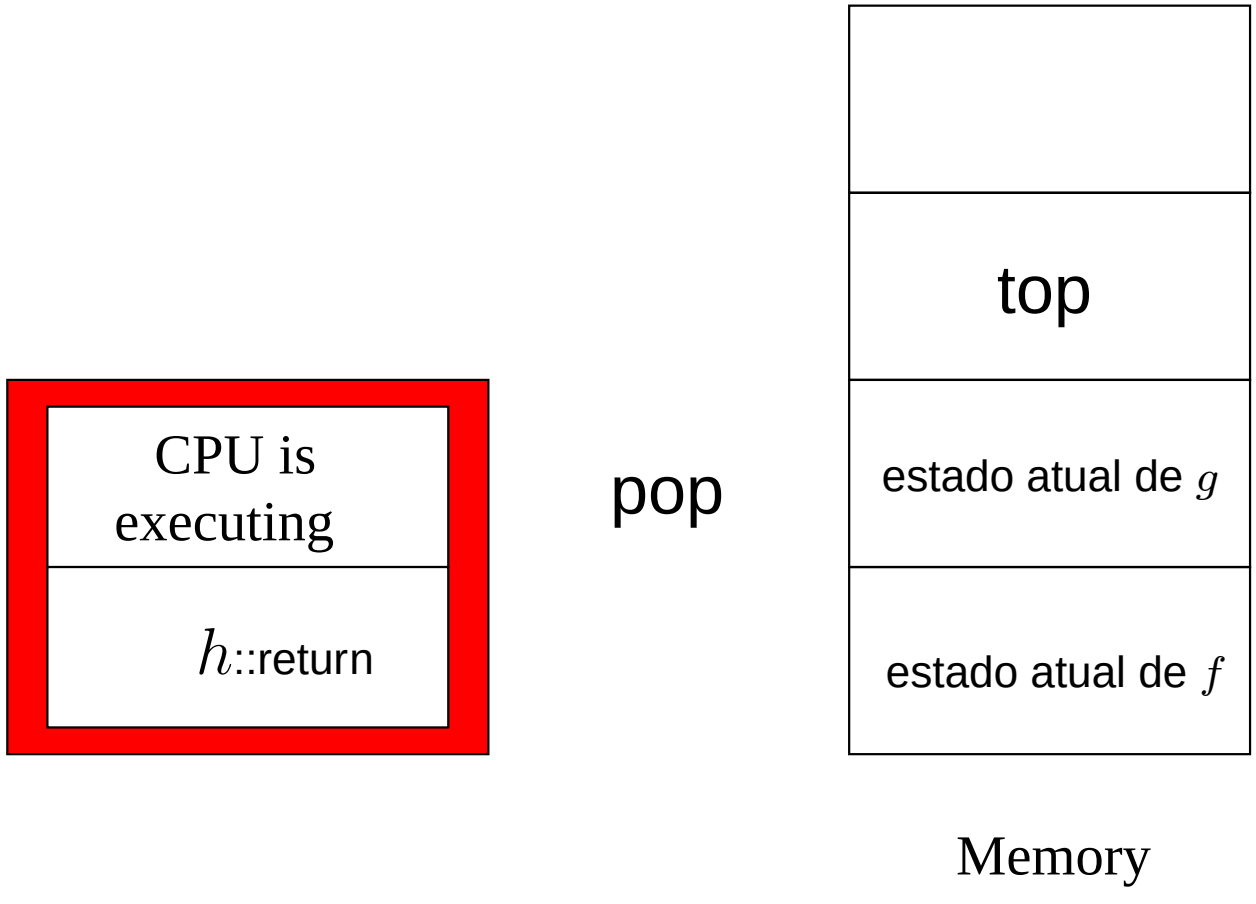
Estados atuais são salvos em uma pilha

f chama g chama h



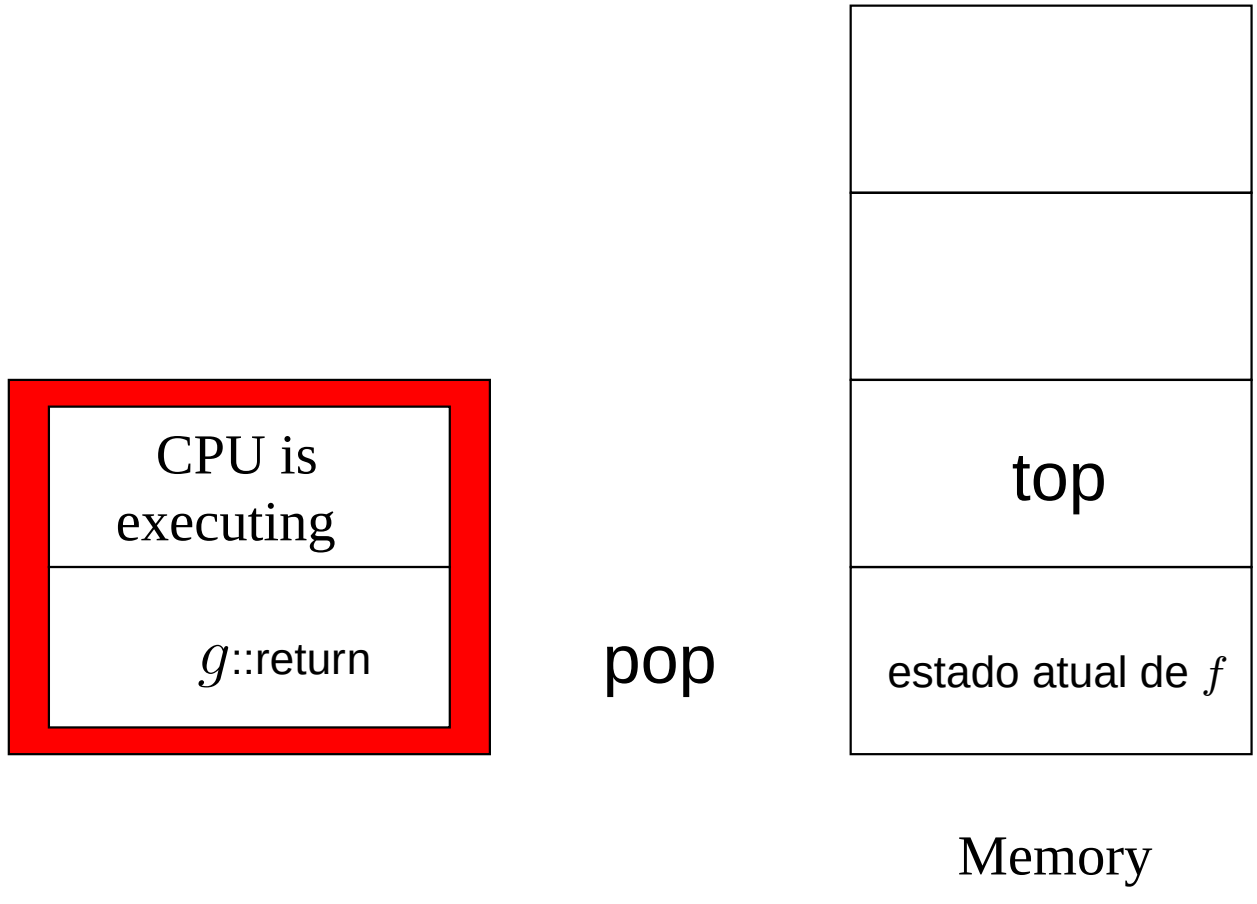
Estados atuais são salvos em uma pilha

f chama g chama h



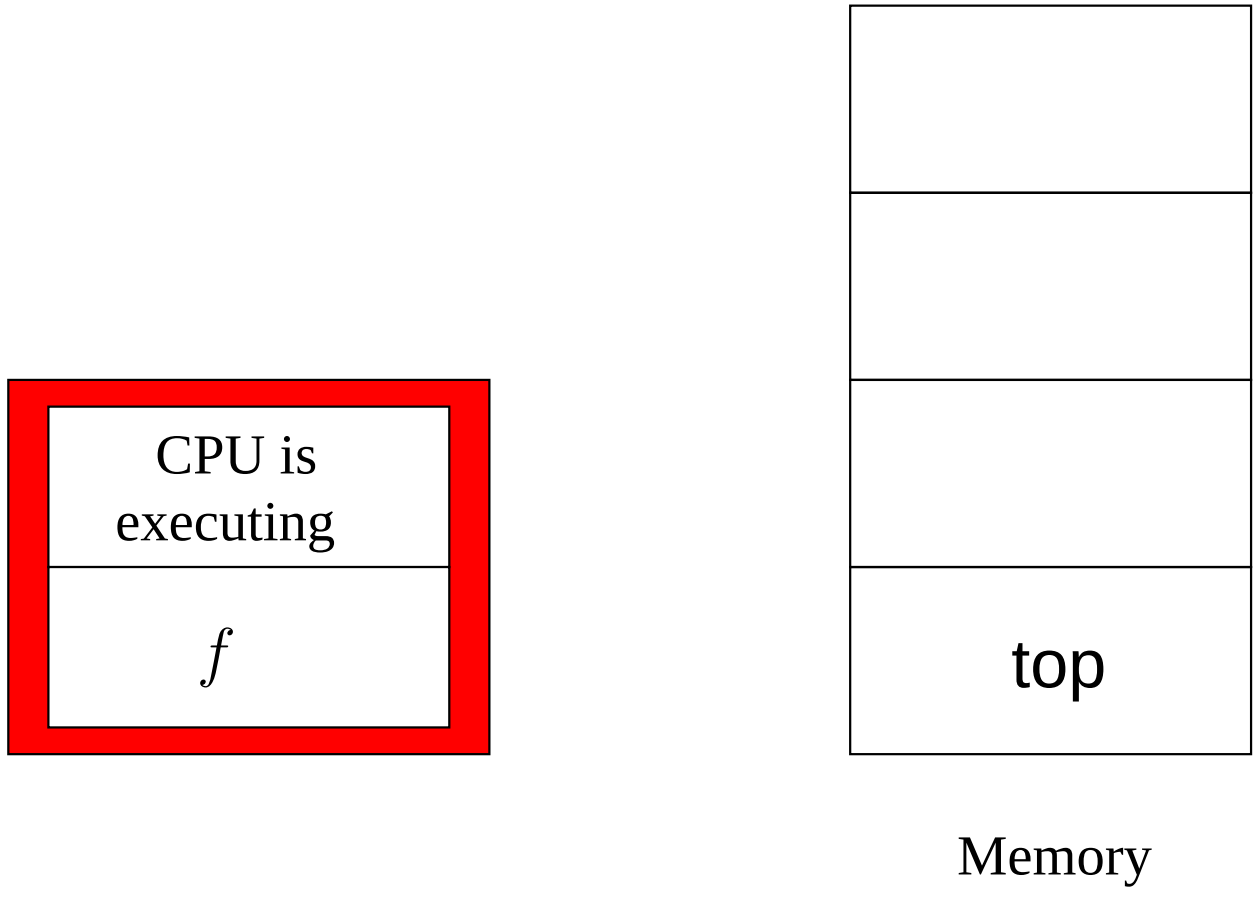
Estados atuais são salvos em uma pilha

f chama g chama h



Estados atuais são salvos em uma pilha

f chama g chama h



Pilhas e aplicações

Pilha

- Estrutura de dados linear
- Acessível a partir de apenas uma extremidade (topo)
- Operações:
  - adicionar um dado no topo (*push data*)
  - remover um dado do topo (*pop data*)
  - testar quando a pilha está vazia
- Toda operação precisa ser $O(1)$
- Não precisa de inserção/remoção do meio da pilha: pode ser implementada usando arrays

Hack the stack

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

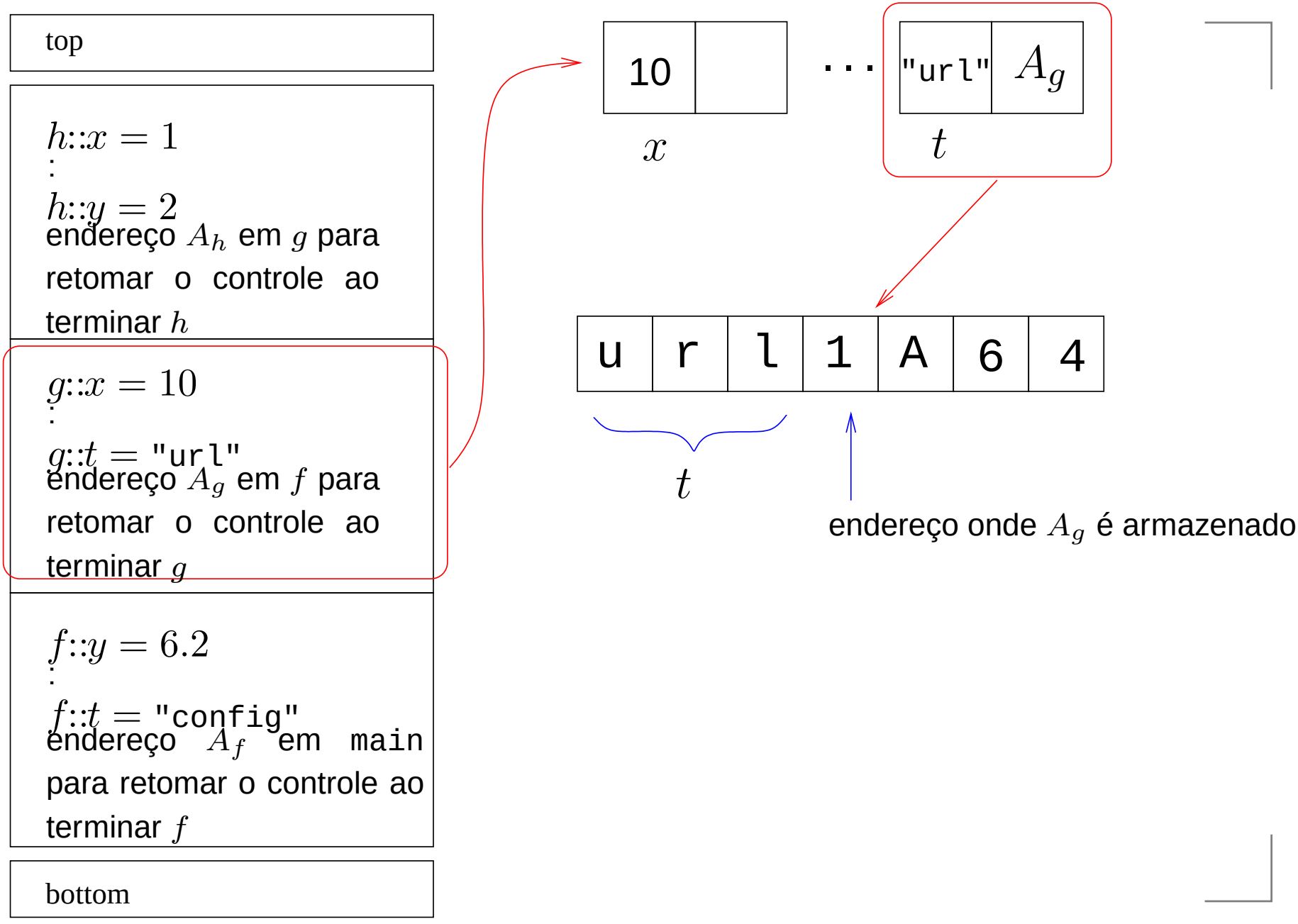
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine. Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address. This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.

Lá atrás em 1996, hackers se infiltravam em sistemas escrevendo códigos disfarçado na pilha de ex

Como funciona?



Como funciona?

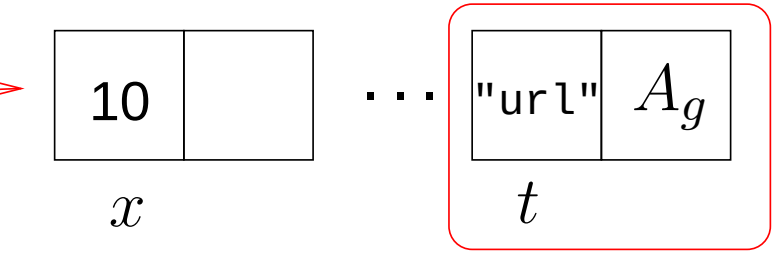
top

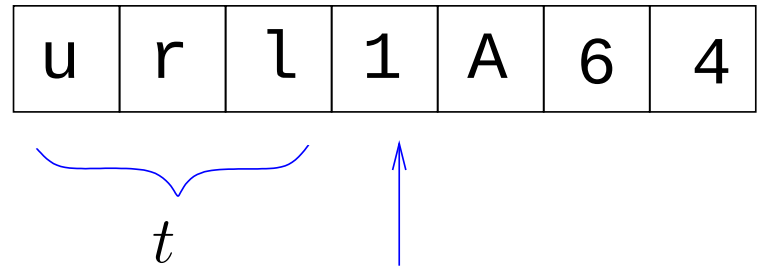
$h::x = 1$
⋮
$h::y = 2$
endereço A_h em g para retomar o controle ao terminar h

$g::x = 10$
⋮
$g::t =$ `"url"`
endereço A_g em f para retomar o controle ao terminar g

$f::y = 6.2$
⋮
$f::t =$ `"config"`
endereço A_f em `main` para retomar o controle ao terminar f

bottom





$g::t$: entrada do usuário (e.g. URL from browser)

Código de g não checa o tamanho da entrada

Usuário pode entrar com strings maiores do que 3 chars

Por exemplo, input `"leo5B"`

Como funciona?

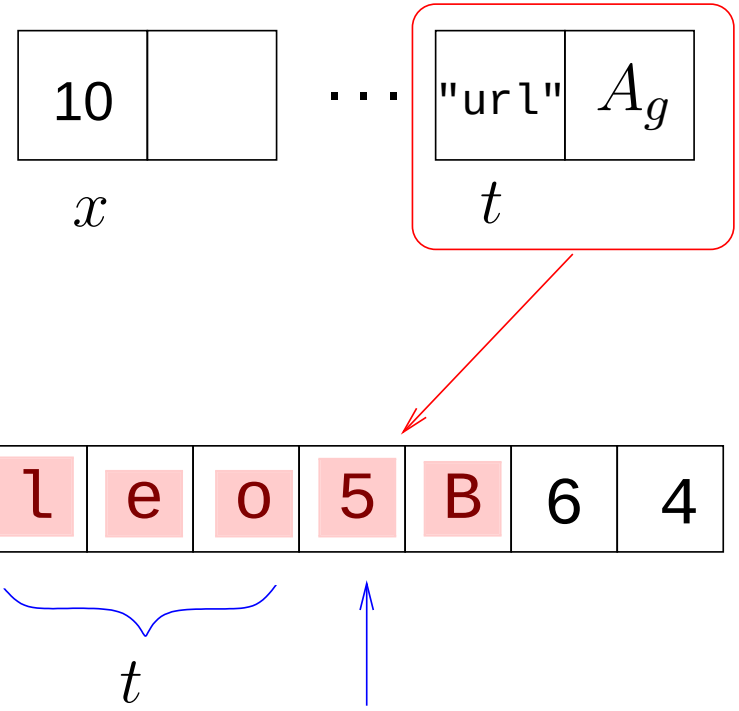
top

$h::x = 1$
$\vdots$
$h::y = 2$
endereço A_h em g para retomar o controle ao terminar h

$g::x = 10$
$\vdots$
$g::t =$ "url"
endereço A_g em f para retomar o controle ao terminar g

$f::y = 6.2$
$\vdots$
$f::t =$ "config"
endereço A_f em main para retomar o controle ao terminar f

bottom



Entrada $t =$ "leo5B" modifica o endereço de retorno

A_g =0x1A64 se torna A' =0x5B64

Quando g termina, a CPU salta para o endereço $A' \neq A_g$

Basta fazer com que o código em A' abra um terminal no modo root

Máquina hackeada

Verificando delimitadores

Dada uma sentença matemática com dois tipos de delimitadores “( )” e “[ ]”, escreva um programa que checa quando a sentença está corretamente delimitada

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar
- Você está escrevendo um interpretador ou um compilador

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar
- Você está escrevendo um interpretador ou um compilador
- Você está escrevendo um sistema operacional

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar
- Você está escrevendo um interpretador ou um compilador
- Você está escrevendo um sistema operacional
- Você está escrevendo código gráfico que precisa executar super rápido e as bibliotecas existentes são muito lentas.

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar
- Você está escrevendo um interpretador ou um compilador
- Você está escrevendo um sistema operacional
- Você está escrevendo código gráfico que precisa executar super rápido e as bibliotecas existentes são muito lentas.
- Você é um expert de segurança que deseja codificar uma pilha impossível de ser corrompida.

Utilidade

Hoje em dia, pilhas são fornecidas por bibliotecas Java/C++, elas são implementadas como um subconjunto de operações de listas ou vetores. Aqui estão algumas razões para você aprender a codificar a sua própria pilha.

- Você é um estudante aprendendo a programar
- Você está escrevendo um interpretador ou um compilador
- Você está escrevendo um sistema operacional
- Você está escrevendo código gráfico que precisa executar super rápido e as bibliotecas existentes são muito lentas.
- Você é um expert de segurança que deseja codificar uma pilha impossível de ser corrompida.
- Você sou eu tentando ensinar pilhas a vocês

Recursão

Compare recursão com iteração

```
while (true) do
  print "hello";
end while
```

```
function f() {
   print "hello";
   f();
}
f();
```

os dois programas resultam no mesmo loop infinito

Quais são as diferenças?

Por que deveríamos nos preocupar com isso?

Differença? Esqueça as atribuições

```
input n;
r = 1
for (i = 1 to n) do
  r = r × i
end for
output r
```

```
function f(n) {
  if (n = 0) then
    return 1
  end if
  return n × f(n − 1)
}
f(n);
```

- Os dois programas computam n!
- A versão iterativa tem atribuições, a versão recursiva não.
- Toda função pode ser computada por meio de {testes, atribuições, iterações} *ou* {testes, recursão}.
- “recursão = atribuição + iteração”

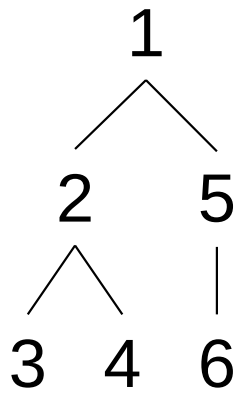
Não esqueça que a função chamadora implica **salvar o estado corrente na pilha**

Terminação

- Tenha certeza de que sua recursão **termina**
- Por exemplo: se $f(n)$ é recursiva,
  - a receorrência recai sobre inteiros menores, e.g. $f(n-1)$ ou $f(n/2)$
  - forneça "casos base", e.g. $f(0)$ or $f(1)$
- Compare com *indução*: prove uma sentença para $n = 0$ e prove que se ela vale para todo $i < n$ então ela vale para n também; conclua que ela vale para todo n
- Tipicamente, uma função recursiva $f(n)$ é assim:

if n é o "caso base" **then**
  compute $f(n)$ diretamente, não continue a recursão
else
  faça a recursão em $f(i)$ com algum $i < n$
end if

Explore esta árvore



Tente instruir o computador a explorar "em profundidade" esta estrutura em árvore (i.e. de modo a imprimir $1, 2, 3, 4, 5, 6$)

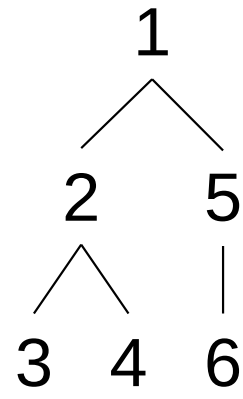
Códificação:
use um array denteado A

A_1: $A_{11} = 2$, $A_{12} = 5$
A_2: $A_{21} = 3$, $A_{22} = 4$
A_3: $\varnothing$
A_4: $\varnothing$
A_5: $A_{51} = 6$
A_6: $\varnothing$

$A_{ij} =$ label do j-ésimo filho do nó i

The iterative failure

```
int a = 1;
print a;
for (int z = 1 to |A_a|) do
  int b = A_az;
  print b;
  for (int y = 1 to |A_b|) do
    int c = A_by;
    print c;
    ...
  end for
end for
```



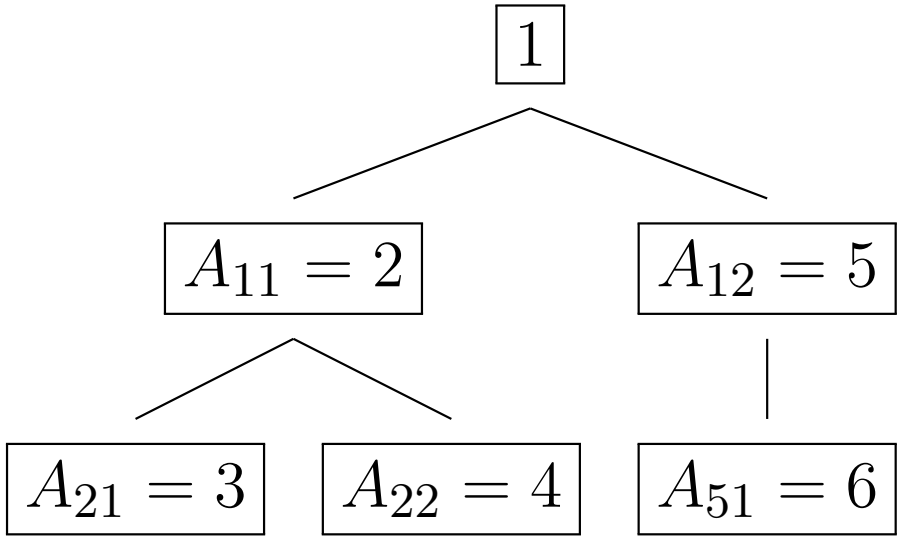
O código precisa mudar de acordo com a árvore???

Nós queremos um código que funcione para **todas** as árvores!

Salvo pela recursão

```
function f(int ℓ) {
    print ℓ;
    for (int i = 1 to |A_ℓ|) do
      f(A_ℓi);
    end for
}
```

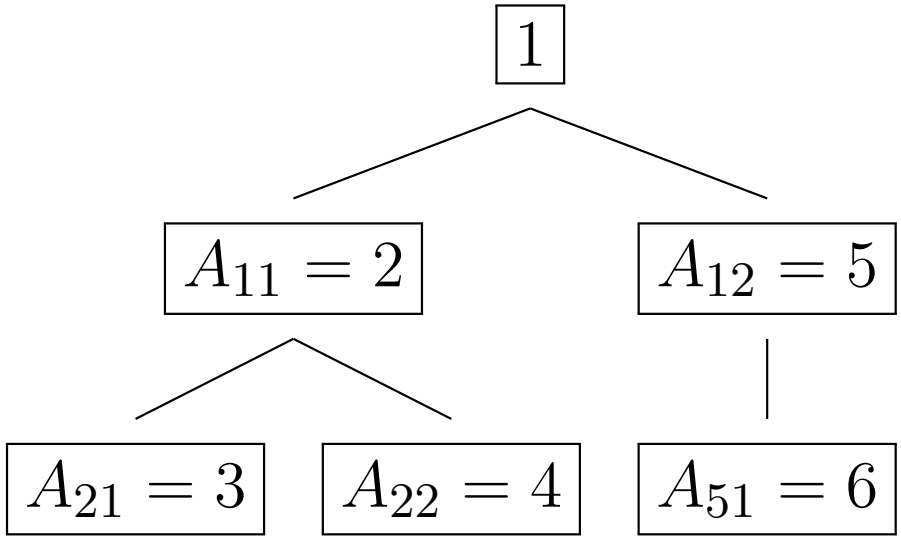
```
main() { f(1); }
```



Salvo pela recursão

```
function f(int ℓ) {
    print ℓ;
    for (int i = 1 to |A_ℓ|) do
      f(A_ℓi);
    end for
}
```

```
main() { f(1); }
```



1. $\ell = 1$; print 1
2. $|A_1| = 2$; $i = 1$
3. call $f(A_{11} = 2)$ [push $\ell = 2$]
4. $\ell = 2$; print 2
5. $|A_2| = 2$; $i = 1$
6. call $f(A_{21} = 3)$ [push $\ell = 3$]
7. $\ell = 3$; print 3
8. $A_3 = \varnothing$
9. return [pop $\ell = 3$]
10. $|A_2| = 2$; $i = 2$
11. call $f(A_{22} = 4)$ [push $\ell = 4$]
12. $\ell = 4$; print 4
13. $A_4 = \varnothing$
14. return [pop $\ell = 4$]
15. return [pop $\ell = 2$]
16. $|A_1| = 2$; $i = 2$
17. call $f(A_{12} = 5)$ [push $\ell = 5$]
18. $\ell = 5$; print 5
19. $|A_5| = 1$; $i = 1$
20. call $f(A_{51} = 6)$ [push $\ell = 6$]
21. $\ell = 6$; print 6
22. $A_6 = \varnothing$
23. return [pop $\ell = 6$]
24. return [pop $\ell = 5$]
25. return; end

Poder da recursão

- À primeira vista, a recursão pode expressar programas que um procedimento iterativo não pode
- Como dito, o “poder de expressão” da recursão é igual ao dos procedimentos iterativos

  você pode escrever programas de qualquer uma das formas

- Entretanto, certos programas são mais facilmente escritos com iterações, e alguns outros com recursão
- **Atenção**: sempre tenha certeza de que sua recursão termina! *Devem existir alguns “casos base”*

Poder da recursão

- À primeira vista, a recursão pode expressar programas que um procedimento iterativo não pode
- Como dito, o “poder de expressão” da recursão é igual ao dos procedimentos iterativos

  você pode escrever programas de qualquer uma das formas
- Entretanto, certos programas são mais facilmente escritos com iterações, e alguns outros com recursão
- **Atenção**: sempre tenha certeza de que sua recursão termina! *Devem existir alguns “casos base”*

Escreva um programa que lista todas as permutações de n elementos

Listando permutações

- Dado um inteiro $n > 1$, liste todas as permutações $\{1, \ldots, n\}$
- Exemplo, $n = 4$
- Suponha que você já tem todas as permutações listadas de $\{1, 2, 3\}$:

$$(1,2,3), (1,3,2), (3,1,2), (3,2,1), (2,3,1), (2,1,3)$$

- Escreva cada uma 4 vezes, e escreva o número 4 em todas as posições:

1	2	3	**4**
1	2	**4**	3
1	**4**	2	3
4	1	2	3
1	3	2	**4**
1	3	**4**	2
1	**4**	3	2
4	1	3	2
3	1	2	**4**
3	1	**4**	2
3	**4**	1	2
4	3	1	2
3	2	1	**4**
3	2	**4**	1
3	**4**	2	1
4	3	2	1
2	3	1	**4**
2	3	**4**	1
2	**4**	3	1
4	2	3	1
2	1	3	**4**
2	1	**4**	3
2	**4**	1	3
4	2	1	3

The algorithm

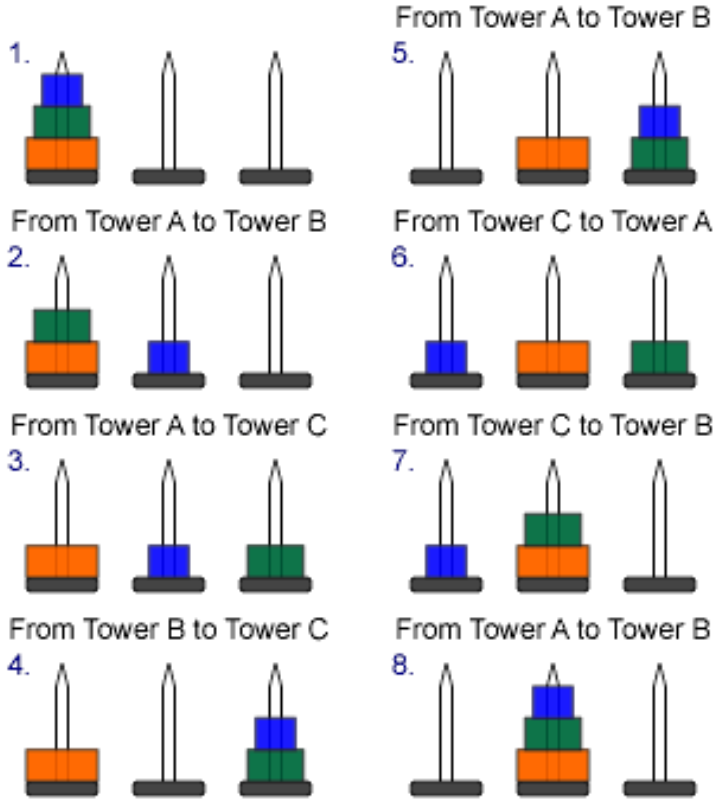
- Se você pode listar permutações para $n-1$, você pode fazer para n
- **Caso base**: $n = 1$ resulta na permutação (1) (nenhuma recursão)

```
function permutations(n) {
 1: if (n = 1) then
 2:    L = {(1)};
 3: else
 4:    L' = permutations(n − 1);
 5:    L = ∅;
 6:    for ((π_1, ..., π_{n−1}) ∈ L') do
 7:       for (i ∈ {1, ..., n}) do
 8:          L ← L ∪ {(π_1, ..., π_{i−1}, n, π_i, ..., π_{n−1})};
 9:       end for
10:    end for
11: end if
12: return L;
}
```

Detalhes de implementação

- L, L' são conjuntos (mathemáticos) : como nós implementamos isto?
- Dada uma lista $(\pi_1, \ldots, \pi_{n-1})$, precisa produzir lista $(\pi_1, \ldots, \pi_{i-1}, n, \pi_i, \ldots, \pi_{n-1})$: como implementamos estas listas?
- **Operações necessárias**:
  - O tamanho de L é conhecido a priori: $|L| = n!$
  - Passar por todos os elementos do conjunto L' usando alguma ordem (**for** no Step 6)
  - Insirir um nó numa posição arbitrária da lista $(\pi_1, \ldots, \pi_{n-1})$ no Step 8
  - Adicionar um elemento ao conjunto L
  - L', L precisam ser do mesmo tipo segundo os Steps 4, 12
- L', L podem ser arrays
- $(\pi_1, \ldots, \pi_{n-1})$ pode ser uma lista

A torre de Hanoi



Mova pilha de discos para diferentes hastes, uma por vez, nunca a maior sobr

Torre de Hanoi

Abordagem recursiva

Para mover k discos da haste 1 para a haste 3:

1. mova os $k-1$ discos mais ao topo da haste 1 para a haste 2
2. mova o disco maior da haste 1 para a haste 3
3. mova os $k-1$ discos da haste 2 para a haste 3

Torre de Hanoi

Abordagem recursiva

Para mover k discos da haste 1 para a haste 3:

1. mova os $k-1$ discos mais ao topo da haste 1 para a haste 2
2. mova o disco maior da haste 1 para a haste 3
3. mova os $k-1$ discos da haste 2 para a haste 3

Reduza o problema ao subproblema com $k-1$ discos

Hipótese: subproblemas para $k-1$ nos Steps 1 e 3 são do *mesmo tipo de problema* para com k

A hipótese é válida porque o disco que é movido no Step 2 é o maior: um jogo da Torre de Hanoi "funciona da mesma forma" se você adiciona discos maiores na base das hastes

Torre de Hanoi

Abordagem recursiva

Para mover k discos da haste 1 para a haste 3:

1. mova os $k-1$ discos mais ao topo da haste 1 para a haste 2
2. mova o disco maior da haste 1 para a haste 3
3. mova os $k-1$ discos da haste 2 para a haste 3

Reduza o problema ao subproblema com $k-1$ discos

Hipótese: subproblemas para $k-1$ nos Steps 1 e 3 são do *mesmo tipo de problema* para com k

A hipótese é válida porque o disco que é movido no Step 2 é o maior: um jogo da Torre de Hanoi "funciona da mesma forma" se você adiciona discos maiores na base das hastes

Você precisa de pilhas para implementar este algoritmo?

Fim do módulo 3