7. Rekurze
(a „hrubá síla“)

BI-EP1
Efektivní programování 1

ZS 2025/2026

Ing. Martin Kačer, Ph.D.

Katedra teoretické informatiky
Fakulta informačních technologií
České vysoké učení technické v Praze

Rekurze – co je to?

- Funkce (metoda) během svého vykonávání způsobí své nové vyvolání
  - Přímo
  - Nepřímo (přes jinou funkci/metodu)

Rekurze – proč?

- Programovací technika
  - V některých jazycích v podstatě jediná možnost pro vyjádření opakování atp.
  - Je stejně „silná“ jako iterace (cykly)

- Způsob uvažování o problému
  - Může zjednodušit řešení

Rekurze – kdy?

- Rekurzivní povaha problému
  - Hierarchické datové struktury
  - Rekurentní definice

- Rekurzivní přístup k řešení
  - Převod na jednodušší instanci
  - Rozdělení na více menších podproblémů

Rekurzivní pohled

- Instance problému (konkrétní zadání)
- K řešení použijeme řešení menší instance
  - Jedné (lineární rekurze)
  - Více (stromová rekurze)
- Nejjednodušší instanci vyřešíme

Příklad – faktoriál

- N! = 1 . 2 . 3 . 4 . … . (N-1) . N

- „Menší“ instance
  - (N-1)!
- Krok řešení
  - N! = (N-1)! . N
- Základní případ (ukončující podmínka)
  - 0! = 1

Faktoriál – kód

- Ukončující podmínka
- Rekurzivní krok

```
int factorial(int n) {
    if (n < 1) return 1;
    return n * factorial(n-1);
}
```

Příklad – Fibonacciho čísla

- 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- F(1) = F(2) = 1
- pro N>2: F(N) = F(N-2) + F(N-1)

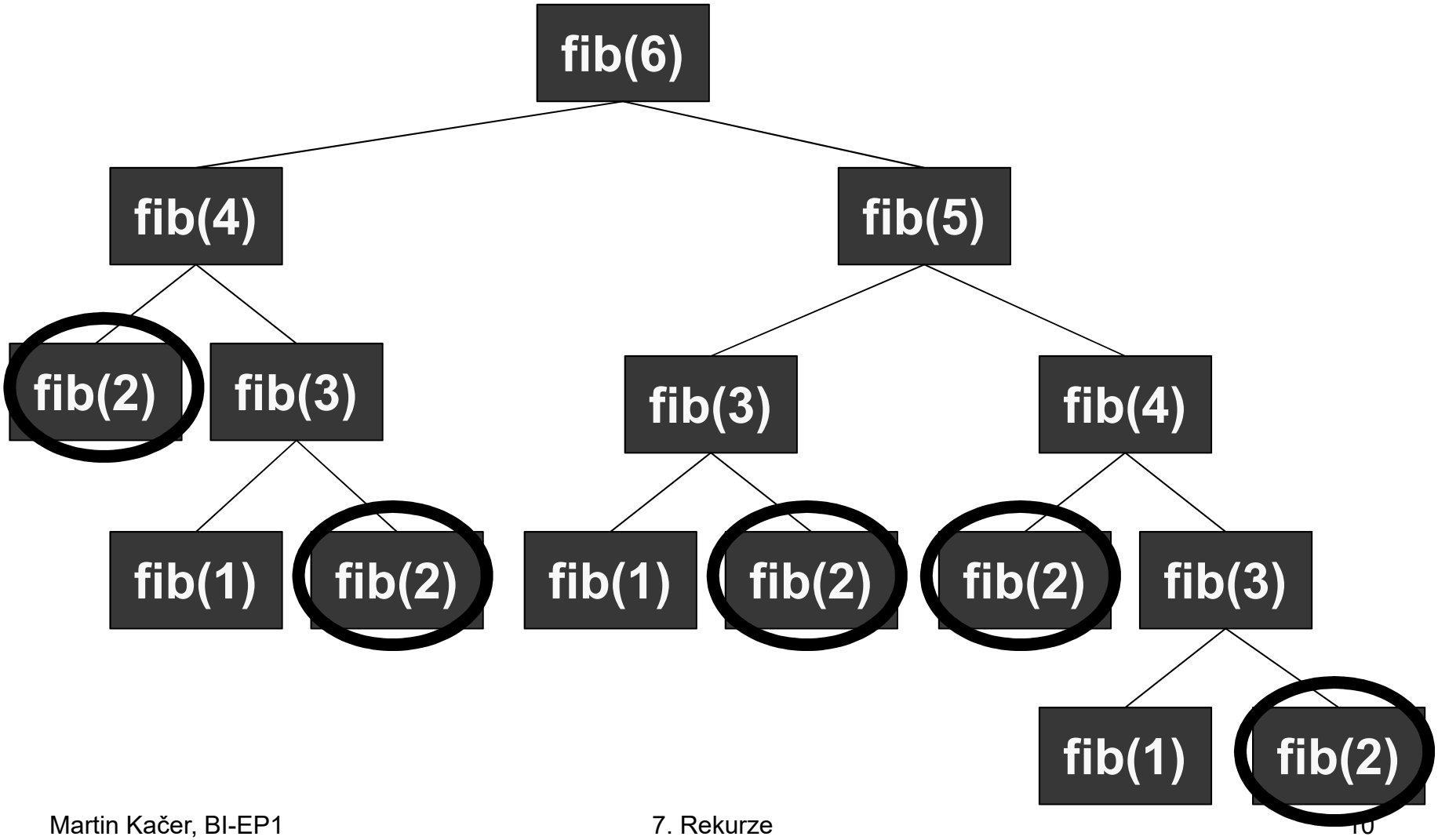
- Rekurzivní řešení se nabízí

Fibonacciho čísla – kód

- F(1) = F(2) = 1
- pro N>2: F(N) = F(N-2) + F(N-1)

```
int fib(int n) {
    if (n <= 2) return 1;
    return fib(n-2) + fib(n-1);
}
```

Strom rekurzivního volání

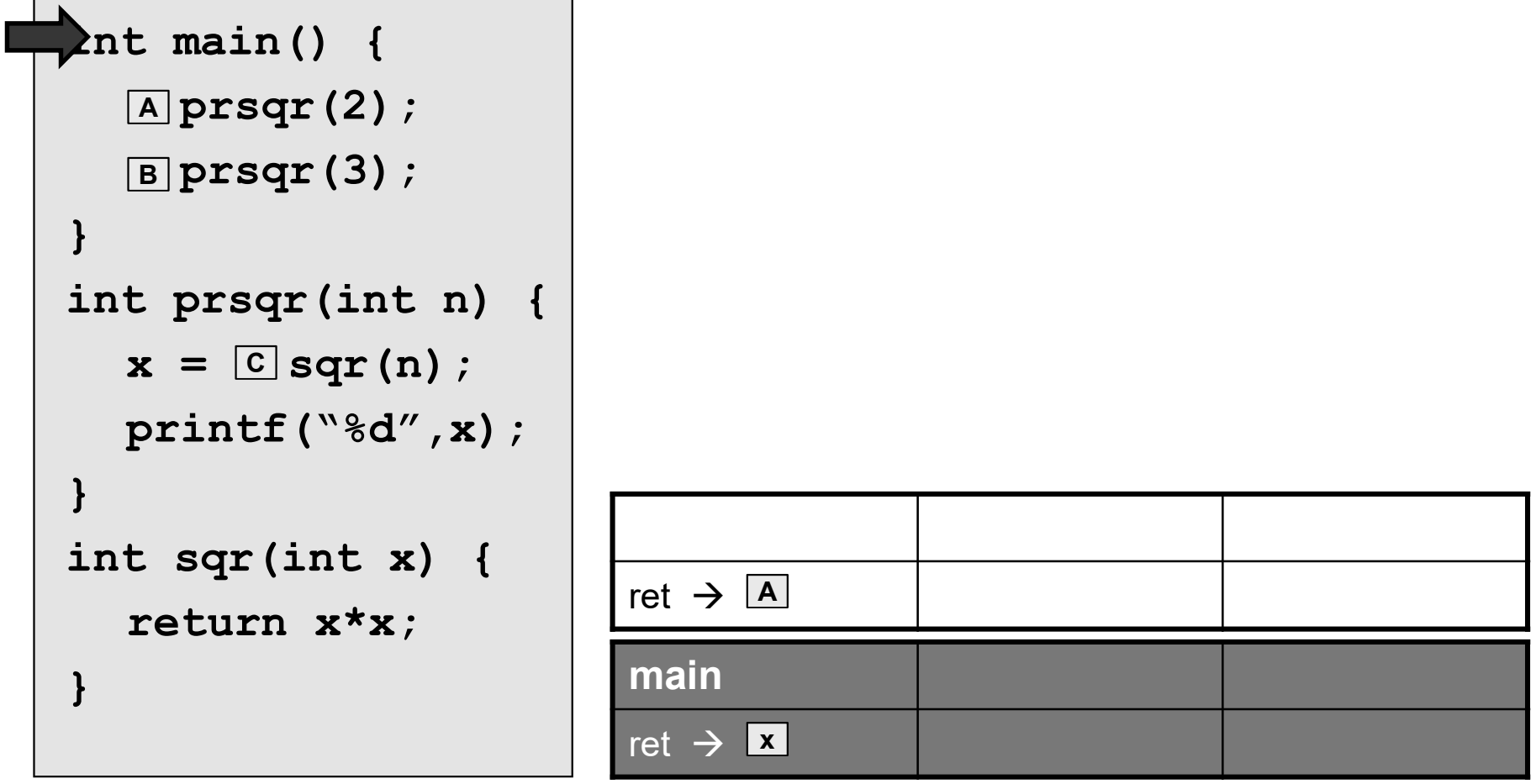


Princip volání funkcí

- Lokální proměnné – systémový zásobník
- Při volání funkce uložení na zásobník
  - Registry
  - Návratová adresa
  - Parametry funkce
- Při návratu
  - Odstranění ze zásobníku
  - Předání výsledku

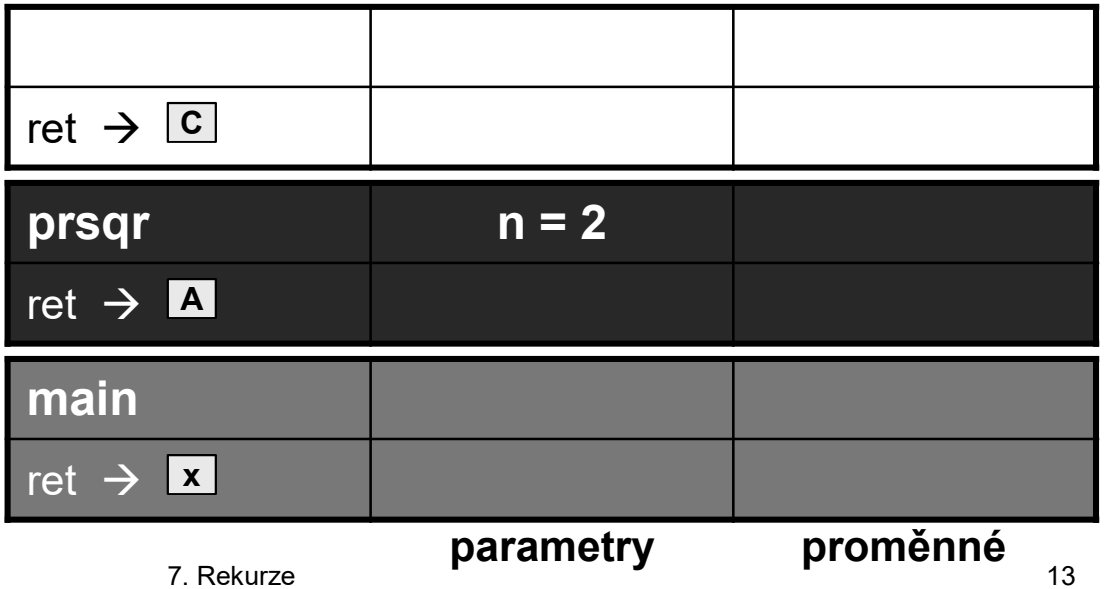
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



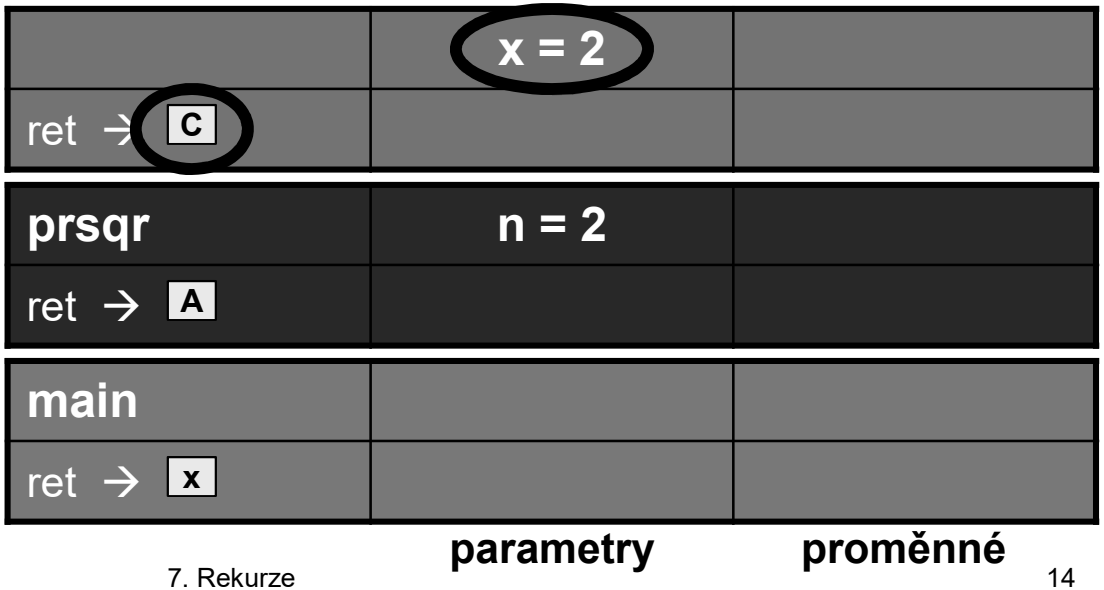
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



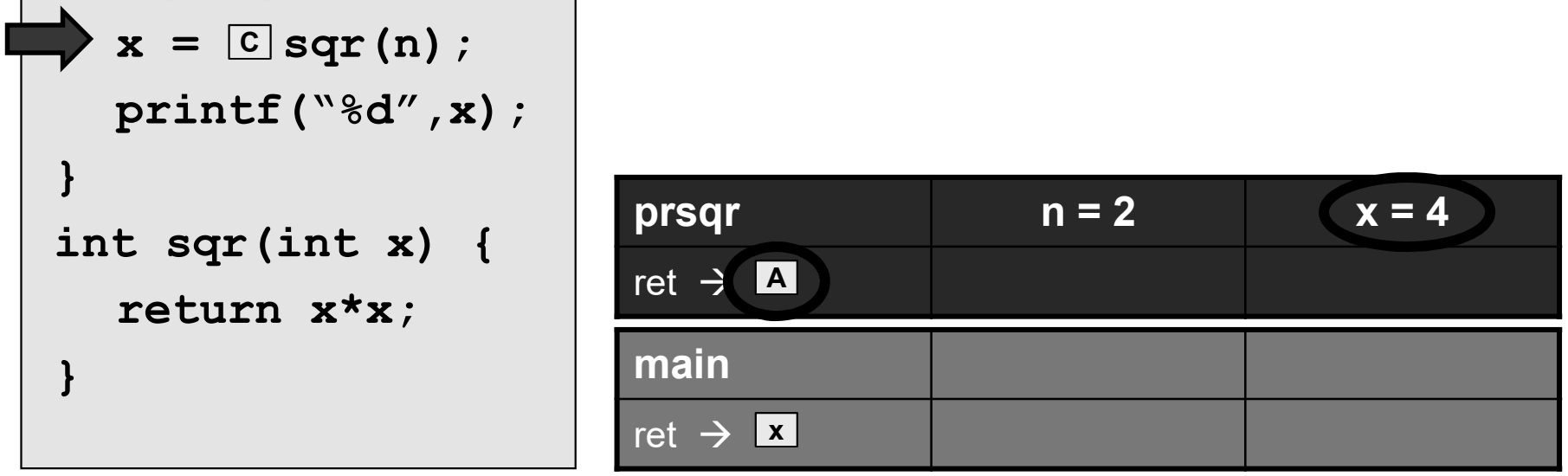
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



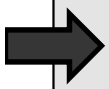
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



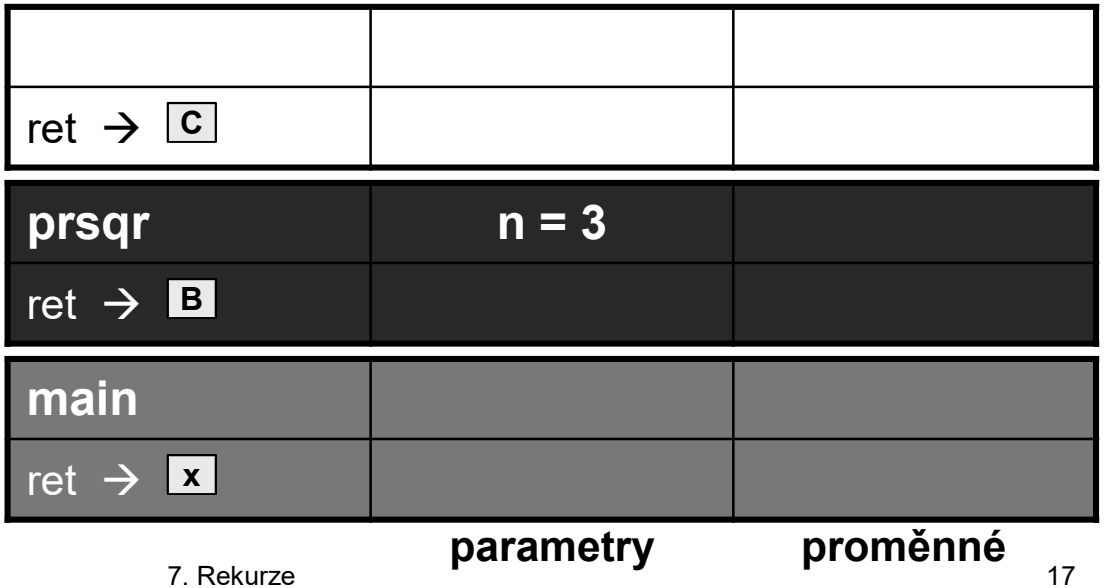
Volání funkcí – demo

```
int main() {
  A prsqr(2);
  B prsqr(3);
}
int prsqr(int n) {
  x = C sqr(n);
  printf("%d",x);
}
int sqr(int x) {
  return x*x;
}
```



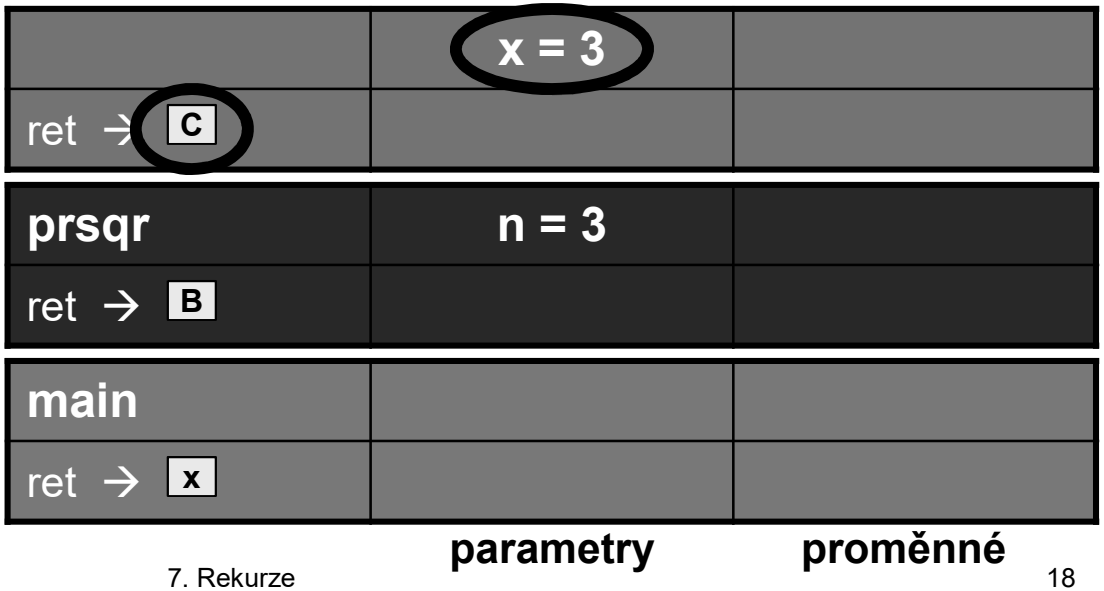
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



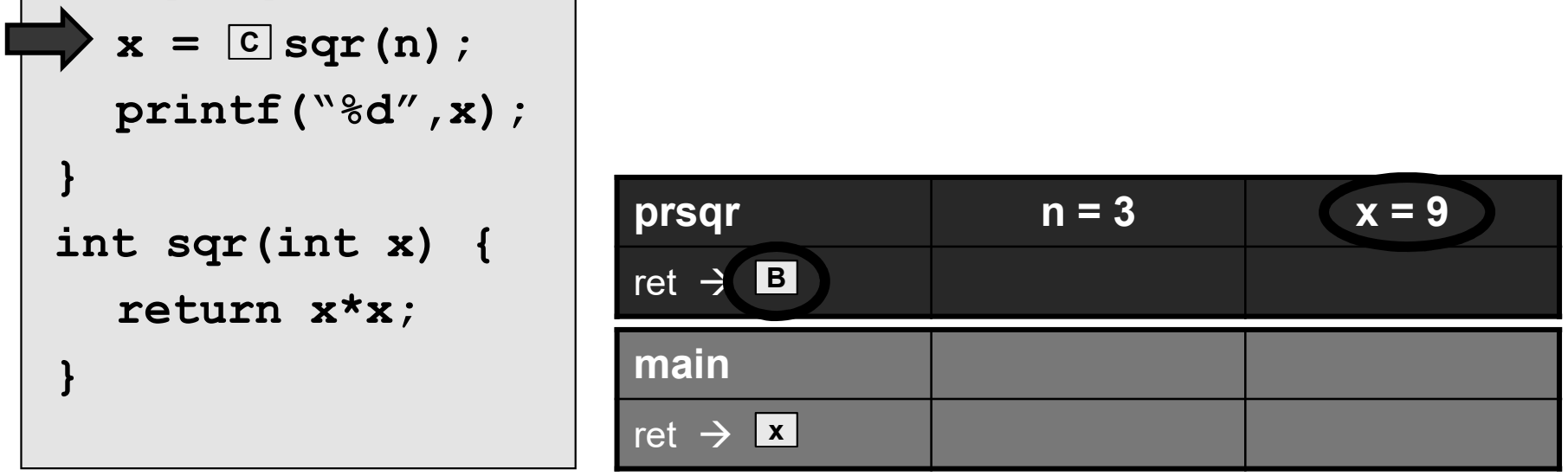
Volání funkcí – demo

```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



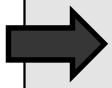
Volání funkcí – demo

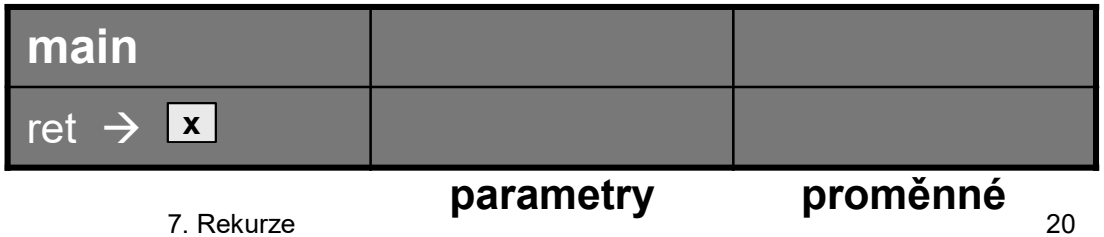
```
int main() {
   A prsqr(2);
   B prsqr(3);
}
int prsqr(int n) {
   x = C sqr(n);
   printf("%d",x);
}
int sqr(int x) {
   return x*x;
}
```



Volání funkcí – demo

```
int main() {
  A prsqr(2);
  B prsqr(3);
}
int prsqr(int n) {
  x = C sqr(n);
  printf("%d",x);
}
int sqr(int x) {
  return x*x;
}
```





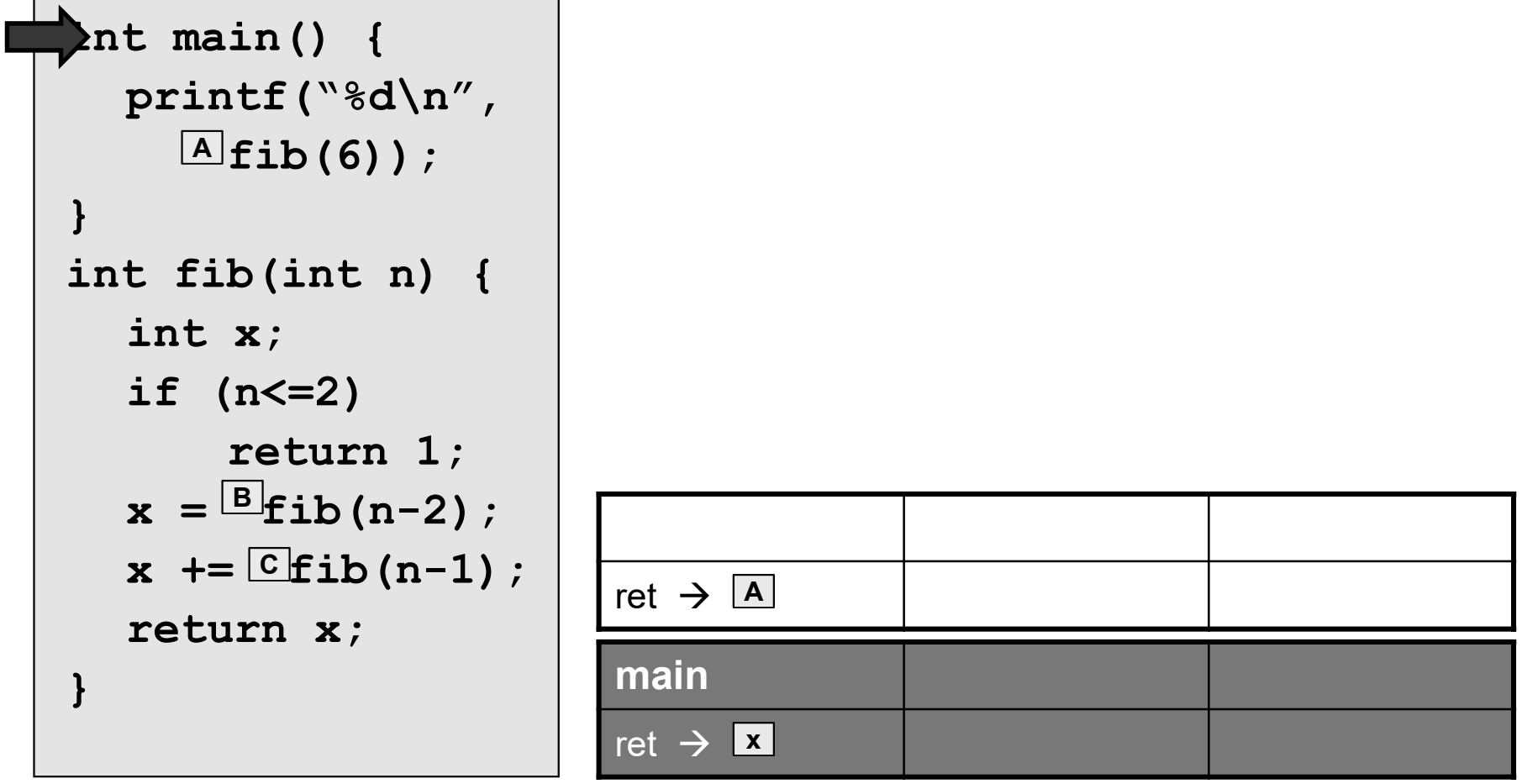
Rekurze – funguje stejně

- Rekurze – při vykonávání nic nového

- Vykonává se znovu stejná funkce
- Stav uložen na zásobníku:
  - Parametry
  - Proměnné
  - Návratové adresy

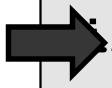
Rekurze – demo

```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



Rekurze – demo

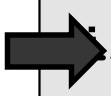
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

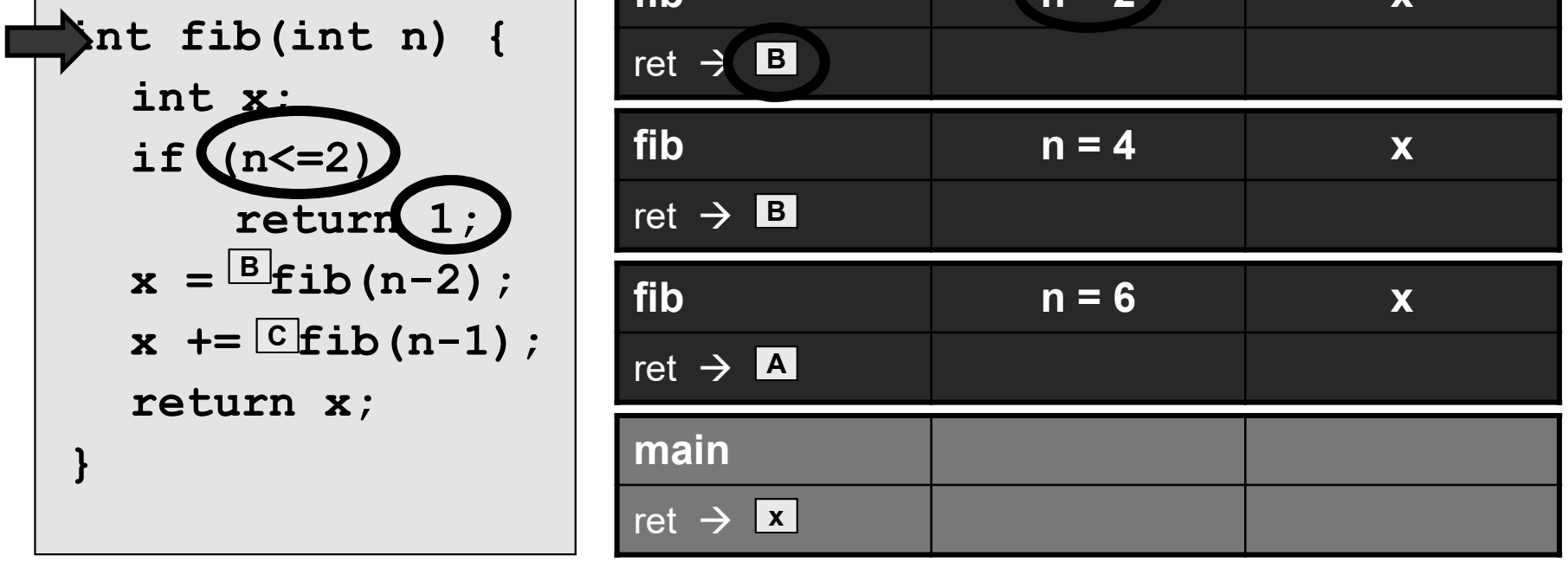
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



ret → B		
fib	**n = 4**	**x**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

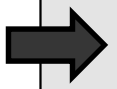
Rekurze – demo

```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
      return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



Rekurze – demo

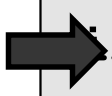
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



ret → C		
fib	**n = 4**	**x = 1**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

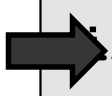
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



ret → B		
fib	**n = 3**	**x**
ret → C		
fib	**n = 4**	**x = 1**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

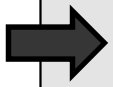
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
      return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



fib	**n = 1**	**x**
ret → B		
fib	**n = 3**	**x**
ret → C		
fib	**n = 4**	**x = 1**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

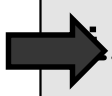
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



ret → C		
fib	**n = 3**	**x = 1**
ret → C		
fib	**n = 4**	**x = 1**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
      return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```

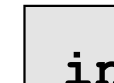


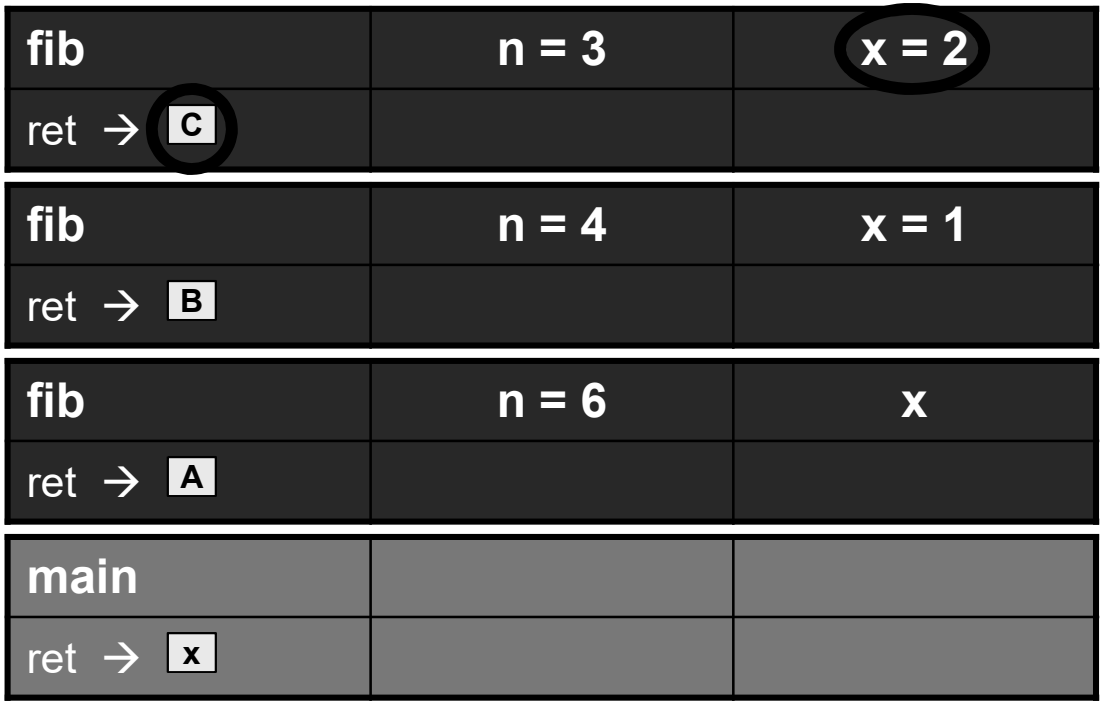
fib	n = 2	x
ret → C		
fib	**n = 3**	**x = 1**
ret → C		
fib	**n = 4**	**x = 1**
ret → B		
fib	**n = 6**	**x**
ret → A		
main		
ret → x		

parametry **proměnné**

Rekurze – demo

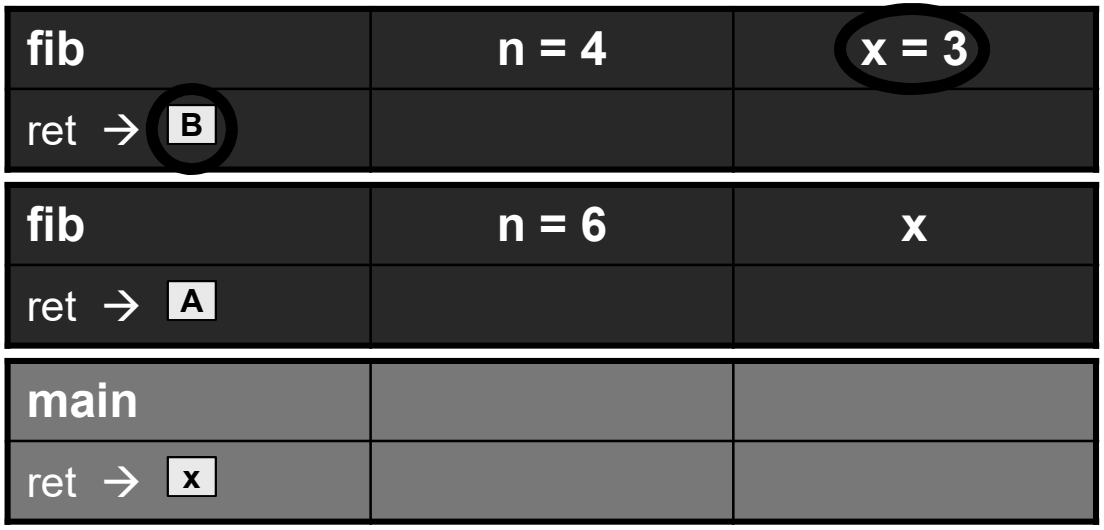
```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```





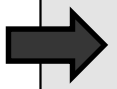
Rekurze – demo

```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



Rekurze – demo

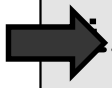
```
int main() {
  printf("%d\n",
    A fib(6));
}
int fib(int n) {
  int x;
  if (n<=2)
      return 1;
  x = B fib(n-2);
  x += C fib(n-1);
  return x;
}
```



ret → C		
fib	**n = 6**	**x = 3**
ret → A		
main		
ret → x		
	parametry	**proměnné**

Rekurze – demo

```
int main() {
   printf("%d\n",
      A fib(6));
}
int fib(int n) {
   int x;
   if (n<=2)
       return 1;
   x = B fib(n-2);
   x += C fib(n-1);
   return x;
}
```



fib	n = 5	x
ret → C		

fib	n = 6	x = 3
ret → A		

main		
ret → x		

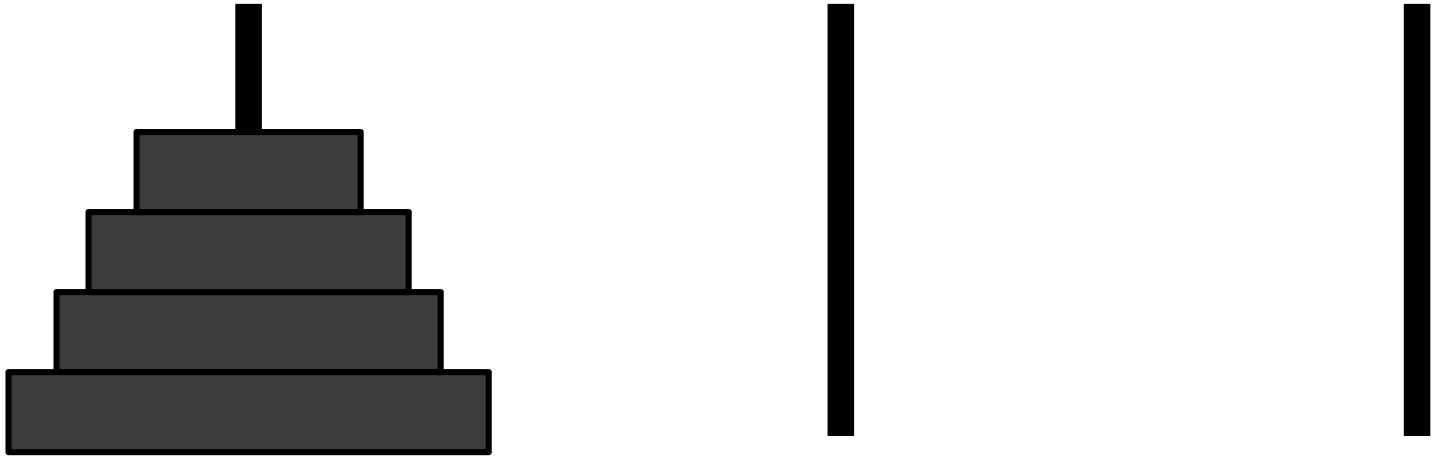
parametry proměnné

Rekurze – demo

- … a tak dále

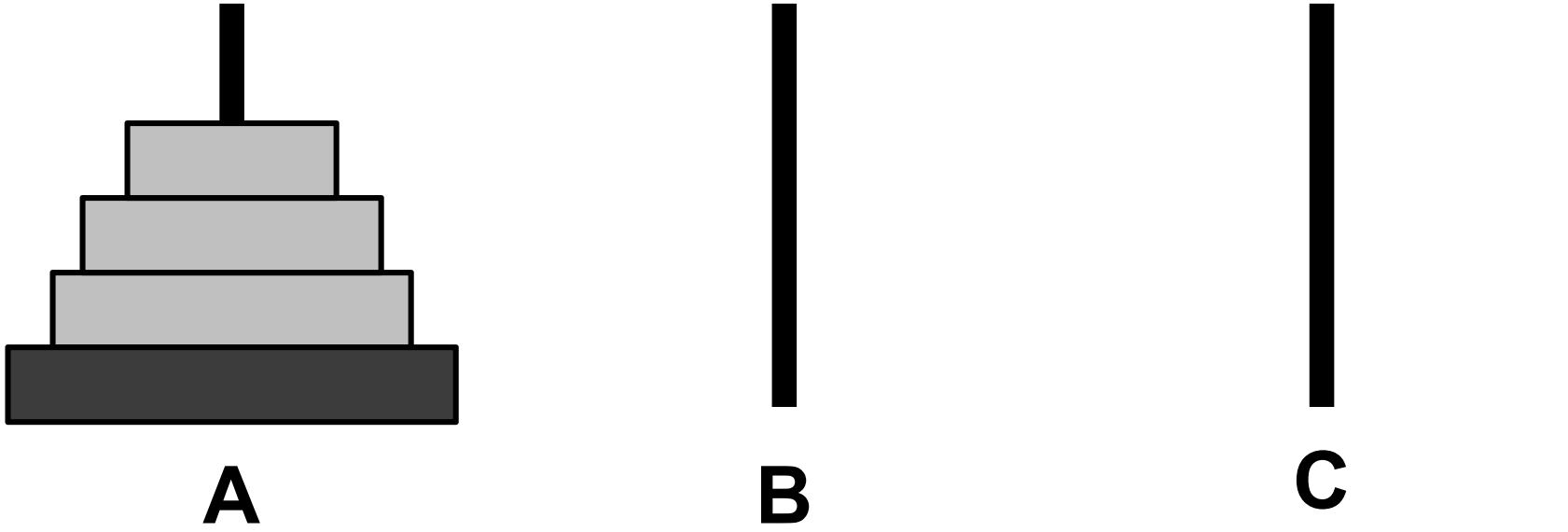
Příklad – hanojské věže

- Přesouvání disků mezi třemi trny
- Po jednom a pouze menší na větší
- Úkol: přenést z A na B



Hanojské věže – jak na to?

- Rekurzivně
  - N-1 disků z A na C (největší disk nepřekáží)
  - největší z A na B
  - N-1 disků z C na B



Hanojské věže – kód

- Ukončovací podmínka?
  - Co nejjednodušší => žádný disk!

```
void hanoi(int n,
  int from, int to, int use) {
     if (!n) return;
     hanoi(n-1, from, use, to);
     move_one(n, from, to);
     hanoi(n-1, use, to, from);
}
```

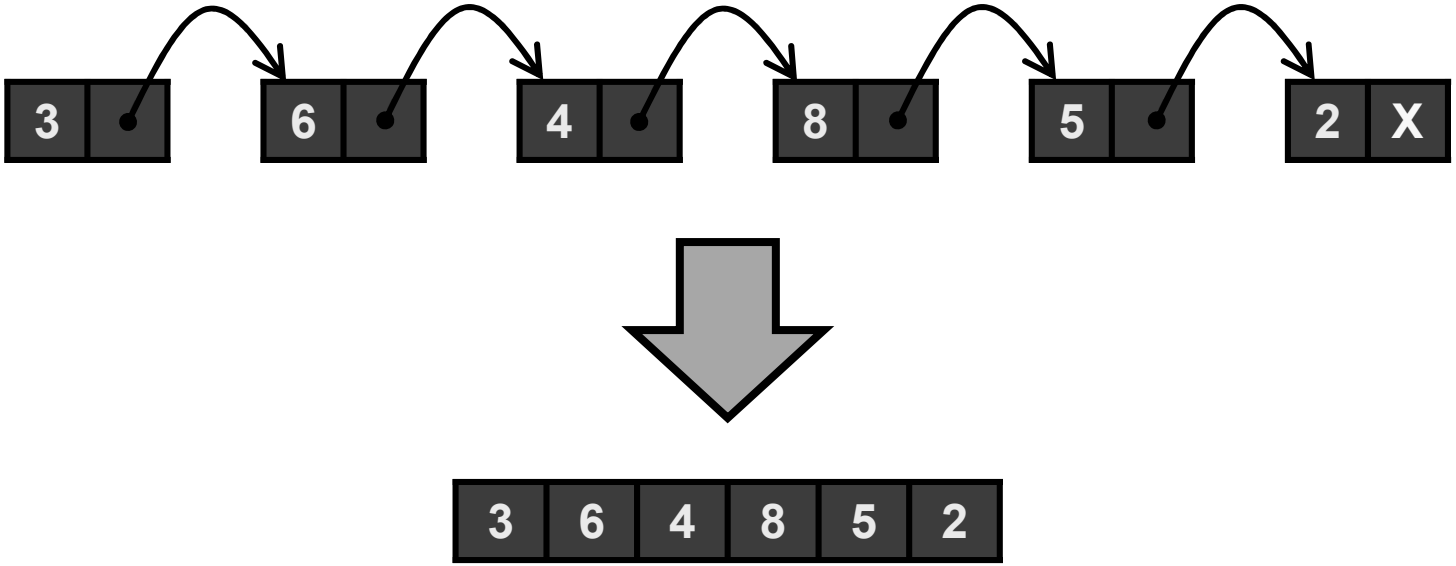
Jak vidět rekurzi?

- „Kdyby byla instance problému o maličko menší, uměl bych řešení rozšířit“
- Voláme funkci, aby menší instanci vyřešila (jako když voláme standardní funkci)

- Je lepší si příliš nepředstavovat úrovně zanoření a zásobník
  - (ale měli bychom znát)

Kopírování seznamu do pole

- Na vstupu spojový seznam
- Výsledkem pole se stejnými prvky



Kopírování seznamu do pole

- Standardní řešení má 2 průchody:
  1. Určení počtu prvků
  2. Kopírování

- Rekurze:
  - Při zanořování počítá prvky
  - Při návratu kopíruje

Seznam – rekurzivní krok

- Zkopírujeme zbytek seznamu do dostatečně velkého pole
- Na první pozici přidáme první prvek

- Potřebujeme přidat parametr
  - „Kolik prvků na začátku vynechat“

Seznam – kód

- Ukončovací podmínka = prázdný seznam
- Šlo by to bez rekurze napsat kratší?

```
int[] copyList(Node head) {
      return copyRest(0, head);
}
int[] copyRest(int n, Node head) {
      if (head == null)
            return new int[n];
      int[] res = copyRest(n+1, head.next);
      res[n] = head.number;
      return res;
}
```

Anagramy – zadání

- Anagram = přesmyčka
  - Permutace písmen ve slově
- **abcd:**

abcd	bacd
abdc	badc
acbd	bcad
acdb	bcda
adbc	bdac
adcb	….

Anagramy – pomocí rekurze

- Vybereme první znak (postupně všechny)
- Rekurzivně přidáme permutaci zbytku

```
char word[100];
int used[100], wlen;
char anagram[100];
. . .
wlen = strlen(word);
for (i = 0; i < wlen; ++i) used[i] = 0;
generate(0);
```

Anagramy – pomocí rekurze

```
void generate(int alen) {
   int i;
   if (alen == wlen) {
       printf("%s\n", anagram);
       return;
   }
   for (i = 0; i < wlen; ++i) {
       if (used[i]) continue;
       used[i] = 1;
       anagram[alen] = word[i];
       generate(alen+1);
       used[i] = 0;
} }
```

Anagramy – rozšíření

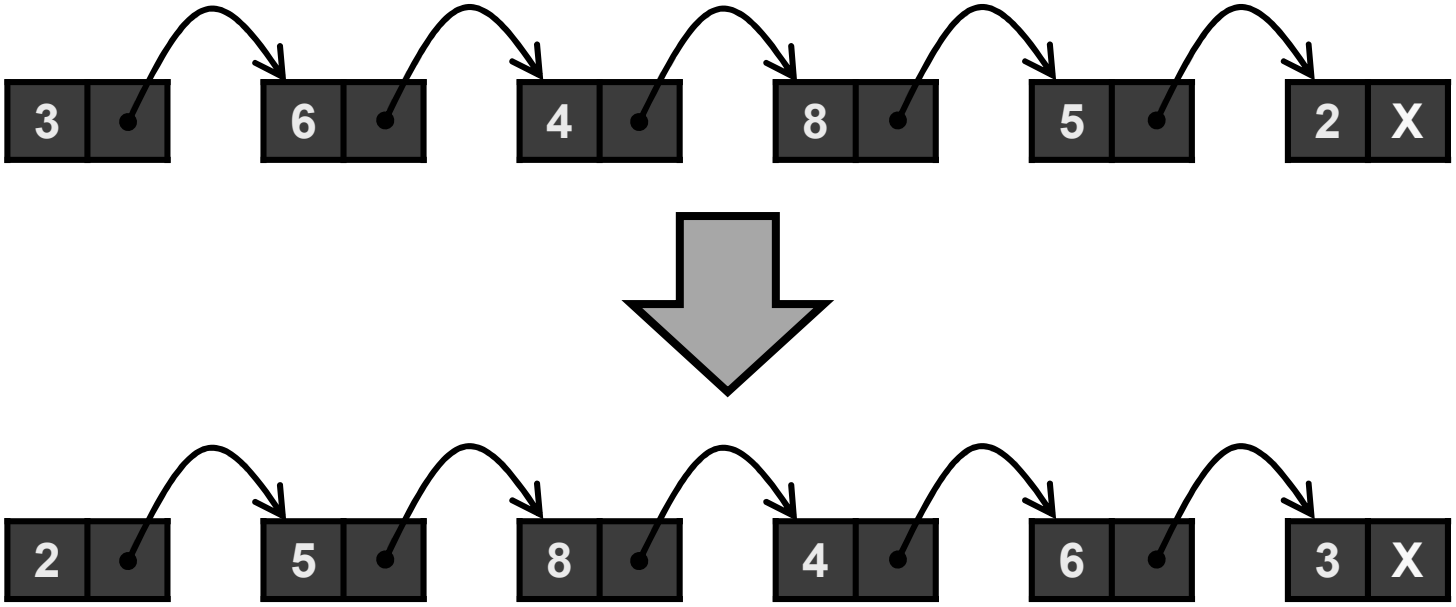
- Co když se budou písmena opakovat?
  - Chceme odstranit duplicitní výsledky
- **abba:**

aabb	baab
abab	baba
abba	bbaa

- Změnit rekurzivní krok (bez opakování)

Obrácení pořadí prvků

- Na vstupu spojový seznam
- Výsledkem seznam v opačném pořadí



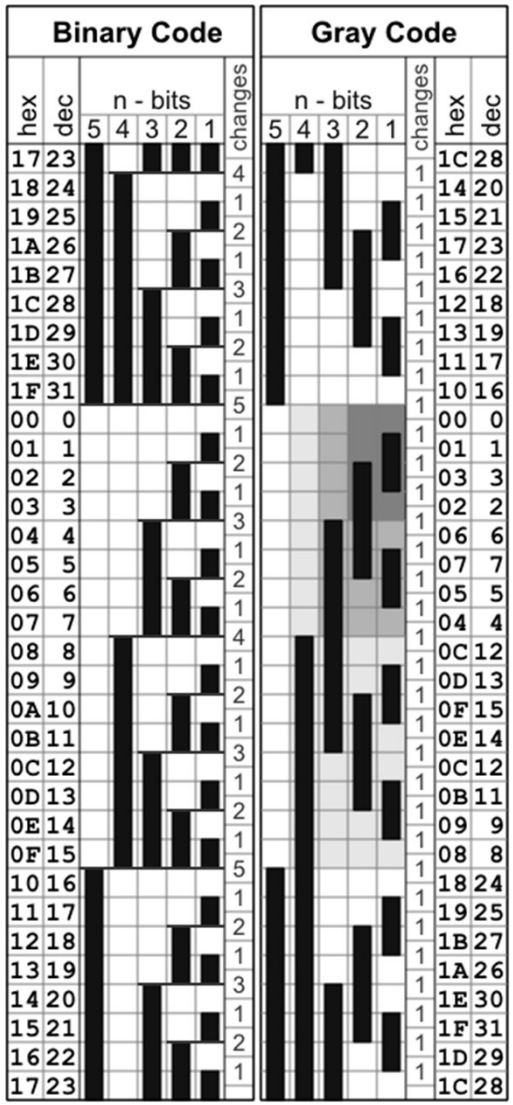
Grayův kód

- Všechna binární čísla, tj. 2^N prvků
- Po sobě jdoucí kódy se liší jedinou číslicí

- Kód délky N:
  - 0 na začátku + kód délky N-1
  - 1 na začátku + kód délky N-1 *pozpátku*

Grayův kód – příklad

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0



Grayův kód – zakódování

- 0 na začátku + kód délky N-1
- 1 na začátku + kód délky N-1 pozpátku

```
String encode(int idx, int len) {
  if (len == 0) return "";
  int cnt = 1 << len;
  return (idx < cnt/2)
     ? "0" + encode(idx, len-1)
     : "1" + encode(cnt-idx-1, len-1);
}
```

Grayův kód – dekódování

- 0 na začátku + kód délky N-1
- 1 na začátku + kód délky N-1 pozpátku

```
int decode(String code) {
  if (code.length() == 0) return 0;
  int rest = decode(code.substring(1));
  return (code.charAt(0) == '0')
     ? rest
     : (1 << code.length()) - 1 - rest;
}
```

Grayův kód – nerekurzivně

```
unsigned binaryToGray(unsigned num) {
   return num ^ (num >> 1);
}

unsigned grayToBinary(unsigned num) {
   unsigned mask;
   for (mask = num>>1; mask; mask = mask>>1)
       num = num ^ mask;
   return num;
}

unsigned grayToBinary32(unsigned num) {
    num = num ^ (num >> 16);
    num = num ^ (num >> 8);
    num = num ^ (num >> 4);
    num = num ^ (num >> 2);
    return num ^ (num >> 1);
}
```

Rekurzivní struktury I

- Kořenové stromy
- XML soubory
- Adresáře na disku

```
void traverse(File f) {
  System.out.println(f); // or do anything else
  if (f.isDirectory()) {
      for (String name : f.list())
           traverse(new File(f, name));
  }
}
```

Rekurzivní struktury II

- Vyhodnocování výrazů (priority, závorky)

```
char next() {
  while (pos < str.length &&
          Character.isWhitespace(str[pos])) ++pos;
  return (pos < str.length) ? str[pos] : 0;
}

char move() {
  char ch = next();
  ++pos;
  return ch;
}
```

Rekurzivní struktury II

```
int expression() {
   int x = term();
   while (next() == '+' || next() == '-')
        if (move() == '+') x += term(); else x -= term();
   return x;
}

int term() {
   int x = factor();
   while (next() == '*' || next() == '/')
        if (move() == '*') x *= factor(); else x /= factor();
   return x;
}

int factor() {
   int x = 0;
   if (next() == '(') { move(); x = expression(); move(); return x; }
   while (Character.isDigit(next())) { x = x * 10 + (move() - '0'); }
   return x;
}
```

Rekurze – na co dát pozor

- Ukončovací podmínka
  - MUSÍ být vždy přítomna
- Místo na systémovém zásobníku
  - Je omezené, musíme s tím počítat
- Časová složitost!
  - Někdy nesnadné určit

Rekurze – operační složitost

- Lineární
  - O(h)
- Stromová (k volání)
  - $O(k^h)$
- Podrobněji viz BI-MA2

Back-tracking („hrubá síla“)

- Zkoušení všech možností
- Obvykle rekurzivně
  - Zkus začátek jedné možnosti
  - Pokračuj rekurzivně všemi možnými způsoby
  - Vyřeš triviální případy
- Příklad: *tahové hry*, *anagramy*

Koncová rekurze

- Co je koncová rekurze? *(tail recursion)*
  - Rekurzivní volání je úplně poslední
    (pak už se jen vrací výsledek)

- Umožňuje efektivnější vykonávání
  - Nevytváří nový rámec na zásobníku

Koncová rekurze – příklad

- Poslední prvek spojového seznamu

```
Item getTail(Item head) {
  if (head.next == null)
     return head;
  return getTail(head.next);
}
```

Koncová rekurze – faktoriál

- Je rekurze u faktoriálu koncová?
  - NENÍ (pak se ještě násobí) → lze upravit?

```
int factorial(int n) {
    if (n < 1) return 1;
    return n * factorial(n-1);
}
```

Koncová rekurze – faktoriál

- Příklad převodu na koncovou rekurzi
  - Často je třeba nový parametr („akumulátor“)

```
int factorial(int a, int n) {
    if (n < 1) return a;
    return factorial(a*n, n-1);
}
```

Koncová rekurze – význam

- Jak lze využít koncovou rekurzi?

- Při volání se nemusí vytvořit nový rámec na zásobníku
  - Nahradí se současný

Koncová rekurze – demo

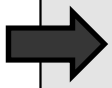
```
int factorial(int a, int n) {
    if (n < 1) return a;
    return factorial(a*n, n-1);
}
```

A

factorial		
ret → x		
	parametry	proměnné

Koncová rekurze – demo

```
int factorial(int a, int n) {
    if (n < 1) return a;
    return factorial(a*n, n-1);
}
```

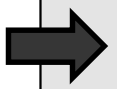


A

factorial	a = 1	n = 5
ret → x		
	parametry	proměnné

Koncová rekurze – demo

```
int factorial(int a, int n) {
    if (n < 1) return a;
    return factorial(a*n, n-1);
}
```



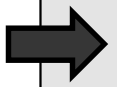
factorial	a = 5	n = 4
ret → A		

factorial	a = 1	n = 5
ret → x		

parametry proměnné

Koncová rekurze – demo

```
int factorial(int a, int n) {
    if (n < 1) return a;
    return A factorial(a*n, n-1);
}
```



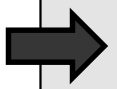
factorial	a = 5	n = 4
ret → A		

factorial	a = 1	n = 5
ret → x		

parametry proměnné

Koncová rekurze – demo

```
int factorial(int a, int n) {
     if (n < 1) return a;
     return factorial(a*n, n-1);
}
```



factorial	a = 5	n = 4
ret → x		
	parametry	proměnné

Koncové volání

- Ve skutečnosti nejde jen o rekurzi
- Jakékoli poslední volání funkce/metody jde takto nahradit

- Některé překladače optimalizují
  - Zejména u funkcionálních jazyků

Dotazy?