

IDT, Přednáška 6

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

12. 3. 2024

Abstraktní datové typy

Abstraktní datové typy

- **definují** možné operace nad daty
- **nedefinují** způsob uložení dat
- **nedefinují** způsob provedení operací (implementaci)

Abstraktní datové typy

- **definují** možné operace nad daty
- **nedefinují** způsob uložení dat
- **nedefinují** způsob provedení operací (implementaci)

Příklad

ADT: celé číslo

- lze sčítat dvě celá čísla, výsledkem je celé číslo
- lze násobit ...
- int, long, uint, ... jsou implementace tohoto ADT

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

rozhraní je způsob implementace konceptu ADT v C#

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

rozhraní je způsob implementace konceptu ADT v C#

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

rozhraní je způsob implementace konceptu ADT v C#

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

Rozdíl:

ADT

součástí je sémantika, **význam!**

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

rozhraní je způsob implementace konceptu ADT v C#

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

Rozdíl:

ADT

součástí je sémantika, **význam!**

rozhraní

- jen říká jaká data jdou dovnitř a jaká ven
- co se s daty má stát musíme vědět
- tuto informaci nám dodá právě znalost ADT

- budeme řešit především struktury uchovávající sady prvků, tzv. **kolekce**
- většina moderních jazyků poskytuje nějakou implementaci kolekcí
- musíme ale vědět, co se děje **uvnitř**!

Složitost práce s kolekcemi

```
bool ContainsDuplicates2(int[] a){  
    MyCollection c = new MyCollection();  
    c.Add(a[0]);  
    for (int i = 1;i<a.Length;i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

Složitost práce s kolekcemi

```
bool ContainsDuplicates2(int[] a){  
    MyCollection c = new MyCollection();  
    c.Add(a[0]);  
    for (int i = 1;i<a.Length;i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

Otázka

Lineární nebo kvadratický algoritmus?

Složitost práce s kolekcemi

```
bool ContainsDuplicates2(int[] a){  
    MyCollection c = new MyCollection();  
    c.Add(a[0]);  
    for (int i = 1; i < a.Length; i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

Otázka

Lineární nebo kvadratický algoritmus?

Odpověď

Záleží na **složitosti** Add() a Contains()!

Operace nad kolekcí

Otázka

Co bychom mohli od kolekce chtít?

Operace nad kolekcí

Otázka

Co bychom mohli od kolekce chtít?

- **přidat** prvek (add)
 - na začátek
 - na konec
 - za/před nějaký prvek
 - s nějakým klíčem

Operace nad kolekcí

Otázka

Co bychom mohli od kolekce chtít?

- **přidat** prvek (add)
 - na začátek
 - na konec
 - za/před nějaký prvek
 - s nějakým klíčem

Pozorování

Kolekce se může chovat jako seznam prvků s číselným indexem (řada), nebo jako množina (nemá indexy, nemá pořadí)

Otázka

Co bychom mohli od kolekce chtít?

- **vybrat** prvek (get)
- po vybrání prvek v kolekci zůstává
 - na začátku
 - na konci
 - na nějakém indexu
 - na nějakém místě (máme nějaké "ukazovátko")
 - s nějakým klíčem
 - s extrémním (nejvyšším, nejnižším) klíčem
- zjistit, zda kolekce obsahuje nějaký prvek

Otázka

Co bychom mohli od kolekce chtít?

- **odstranit** prvek (remove)
- po odstranění je prvek pryč, někdy ho můžeme dostat jako návratovou hodnotu
 - na začátku
 - na konci
 - na nějakém indexu
 - na nějakém místě
 - s nějakým klíčem
 - s extrémním (nejvyšším, nejnižším) klíčem
 - všechny

Ideální kolekce

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\mathcal{O}(1)$
 - čas nezávisí na počtu prvků, které už v kolekci jsou

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\mathcal{O}(1)$
 - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\mathcal{O}(1)$
 - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje
- v praxi ale většinou všechny možnosti nepotřebujeme

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\mathcal{O}(1)$
 - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje
- v praxi ale většinou všechny možnosti nepotřebujeme

Abstraktní datový typ (ADT)

- určuje podmnožinu operací, které je možno provádět
- neurčuje implementaci
- složitost operací není určená

Implementace ADT

- určuje složitost operací
- obvykle reprezentována třídou
 - metody třídy = operace nad ADT

Implementace ADT

- určuje složitost operací
- obvykle reprezentována třídou
 - metody třídy = operace nad ADT

Úkol programátora

- vybrat vhodný ADT
- vybrat vhodnou implementaci ADT
- **vědět co dělá** (jaká je složitost operací) když používá nějakou implementaci ADT

ADT Zásobník

ADT zásobník (Stack)

Operace

- přidej prvek na konec
- vyber prvek na konci
- odstraň prvek z konce
- zjisti, jestli je zásobník prázdný

ADT zásobník (Stack)

Operace

- přidej prvek na konec
- vyber prvek na konci
- odstraň prvek z konce
- zjisti, jestli je zásobník prázdný

Použití:

- seznamy věcí, které je třeba vyřídit
 - když nezáleží na pořadí
 - když chceme nové úkoly řešit jako první, třeba proto, že jejich výsledky jsou potřeba k vyřešení starších úkolů

ADT zásobník (Stack)

Operace

- přidej prvek na konec
- vyber prvek na konci
- odstraň prvek z konce
- zjisti, jestli je zásobník prázdný

Použití:

- seznamy věcí, které je třeba vyřídit
 - když nezáleží na pořadí
 - když chceme nové úkoly řešit jako první, třeba proto, že jejich výsledky jsou potřeba k vyřešení starších úkolů
- pro uložení hodnot, které potřebujeme dočasně změnit, ale ke kterým se budeme chtít vrátit

ADT zásobník (Stack)

Operace

- přidej prvek na konec
- vyber prvek na konci
- odstraň prvek z konce
- zjisti, jestli je zásobník prázdný

Použití:

- seznamy věcí, které je třeba vyřídit
 - když nezáleží na pořadí
 - když chceme nové úkoly řešit jako první, třeba proto, že jejich výsledky jsou potřeba k vyřešení starších úkolů
- pro uložení hodnot, které potřebujeme dočasně změnit, ale ke kterým se budeme chtít vrátit
- LIFO: Last In, First Out

Zásobník



wikiHow to Fill a Pez Dispenser

Prvky v zásobníku

Rozhraní

```
interface IStack<T> {  
    void Add(T element);  
    T Get();  
    void RemoveLast();  
    bool IsEmpty();  
}
```

Implementace ADT zásobník

- polem (array)

Implementace ADT zásobník

- polem (array)
- spojovou strukturou

Implementace zásobníku polem

- třída
- obsahuje pole prvků
- obsahuje index prvního neobsazeného prvku

```
class StackArray<T> : IStack<T>{
    T[] array;
    int freeIndex;
    ...
}
```

Implementace zásobníku polem

- konstruktor založí pole o nějaké velikosti

```
public StackArray(int initialCapacity) {  
    array = new T[initialCapacity];  
    freeIndex = 0;  
}
```

Implementace zásobníku polem

Přidání prvku na konec musí ošetřit situaci, kdy se prvek do pole nevejde

```
void Add(T e) {
    if (freeIndex == array.Length)
        ExpandArray();
    array[freeIndex++] = e;
}
```

Implementace zásobníku polem

Přidání prvku na konec musí ošetřit situaci, kdy se prvek do pole nevejde

```
void Add(T e) {
    if (freeIndex == array.Length)
        ExpandArray();
    array[freeIndex++] = e;
}

void ExpandArray() {
    <typ>[] newArray = new <typ>[array.Length * 2];
    for (int i = 0; i < array.Length; i++)
        newArray[i] = array[i];
    array = newArray;
}
```

operace přidání se také často označuje Push()

Implementace zásobníku polem

Vybrání prvku musí testovat zda v zásobníku vůbec něco je

```
T Get () {  
    if (freeIndex == 0)  
        throw new Exception ();  
    return array[freeIndex-1];  
}
```

Implementace zásobníku polem

Vybrání prvku musí testovat zda v zásobníku vůbec něco je

```
T Get () {  
    if (freeIndex == 0)  
        throw new Exception ();  
    return array[freeIndex-1];  
}
```

Tuto funkcionality je dobré nabídnout i uživateli zásobníku

```
bool IsEmpty () {  
    return (freeIndex == 0);  
}
```

Implementace zásobníku polem

Odstranění prvku nemusí prvek smazat, stačí jeho místo označit jako volné

```
void RemoveLast () {  
    if (freeIndex == 0)  
        throw new Exception();
```

Implementace zásobníku polem

Odstranění prvku nemusí prvek smazat, stačí jeho místo označit jako volné

```
void RemoveLast () {
    if (freeIndex == 0)
        throw new Exception();
    freeIndex--;
}
```

- odstranění by také mohlo uvolňovat paměť, pokud už není potřeba
 - překopírovat prvky do menšího pole

Implementace zásobníku polem

Vybrání a odstranění posledního prvku se často kombinuje do jedné metody `Pop()`

```
T Pop() {  
    if (freeIndex == 0)  
        throw new Exception();  
    freeIndex--;  
    return array[freeIndex];  
}
```

Složitost operací

- vybrání prvku zřejmě $\Theta(1)$

Složitost operací

- vybrání prvku zřejmě $\Theta(1)$
- odstranění prvku $\Theta(1)$, pokud se pole nezmenšuje

Složitost operací

- vybrání prvku zřejmě $\Theta(1)$
- odstranění prvku $\Theta(1)$, pokud se pole nezmenšuje
- přidání prvku?
 - pokud se vejde do pole, pak $\Theta(1)$
 - pokud se pole musí zvětšit, pak $\Theta(n)$
 - v průměru?

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1$:

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0 \times$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 :$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0 \times$
 - $n \leq 2 : 1 \times$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 :$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$
 - $n \leq 8 : 3\times$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$
 - $n \leq 8 : 3\times$
 - $n \leq 2^k : k\times$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$
 - $n \leq 8 : 3\times$
 - $n \leq 2^k : k\times$
- počet zvětšování: $k = \lceil \log_2(n) \rceil$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$
 - $n \leq 8 : 3\times$
 - $n \leq 2^k : k\times$
- počet zvětšování: $k = \lceil \log_2(n) \rceil$
- počet kopírovaných prvků pro i-té zvětšení: $2^{(i-1)}$

Amortizovaná složitost

- třída složitosti v průměrném případě
- zkoumáme přidání n prvků, pro jednoduchost začneme s polem délky 1
- kolikrát se pole zvětšuje?
 - záleží na n
 - $n \leq 1 : 0\times$
 - $n \leq 2 : 1\times$
 - $n \leq 4 : 2\times$
 - $n \leq 8 : 3\times$
 - $n \leq 2^k : k\times$
- počet zvětšování: $k = \lceil \log_2(n) \rceil$
- počet kopírovaných prvků pro i-té zvětšení: $2^{(i-1)}$
- celkový počet operací při zvětšování: $\sum_{i=1}^{\lceil \log_2(n) \rceil} 2^{(i-1)}$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

$$n = 2^k + 1, o = 2^{k+1} - 1$$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

$$n = 2^k + 1, o = 2^{k+1} - 1$$

$$o = 2(2^k) - 1$$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

$$n = 2^k + 1, o = 2^{k+1} - 1$$

$$o = 2(2^k) - 1 = 2(2^k + 1) - 3$$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

$$n = 2^k + 1, o = 2^{k+1} - 1$$

$$o = 2(2^k) - 1 = 2(2^k + 1) - 3 = 2n - 3 < 2n$$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	$0 \times$	0
$n \leq 2$	$1 \times$	1
$n \leq 4$	$2 \times$	$1 + 2 = 3$
$n \leq 8$	$3 \times$	$1 + 2 + 4 = 7$
$n \leq 16$	$4 \times$	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

$$n = 2^k + 1, o = 2^{k+1} - 1$$

$$o = 2(2^k) - 1 = 2(2^k + 1) - 3 = 2n - 3 < 2n$$

Závěr

- složitost operace "přidání n prvků" je $\mathcal{O}(n)$

Celkový počet operací při zvětšení

počet prvků n	počet zvětšení	počet kopírovaných prvků o
$n \leq 1$	0×	0
$n \leq 2$	1×	1
$n \leq 4$	2×	$1 + 2 = 3$
$n \leq 8$	3×	$1 + 2 + 4 = 7$
$n \leq 16$	4×	$1 + 2 + 4 + 8 = 15$
$n \leq 2^k$	$k \times$	$2^k - 1$

Nejhorší případ: hned po zvětšení pole:

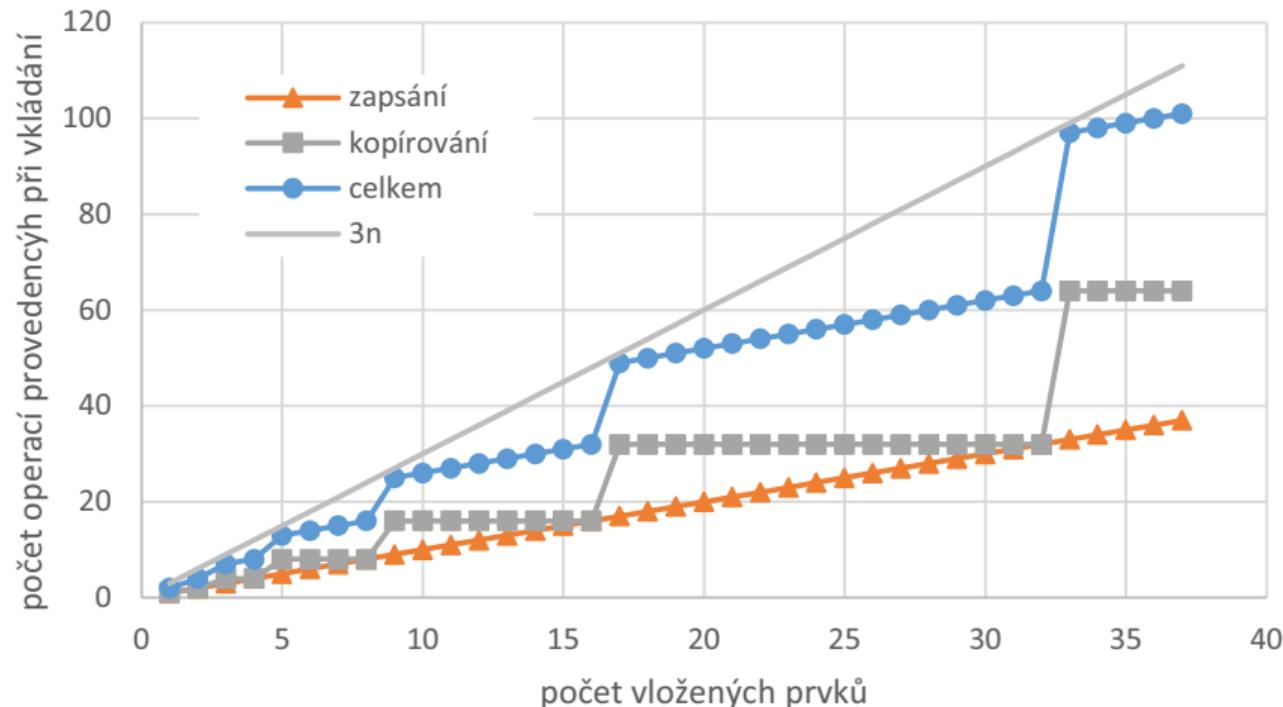
$$n = 2^k + 1, o = 2^{k+1} - 1$$

$$o = 2(2^k) - 1 = 2(2^k + 1) - 3 = 2n - 3 < 2n$$

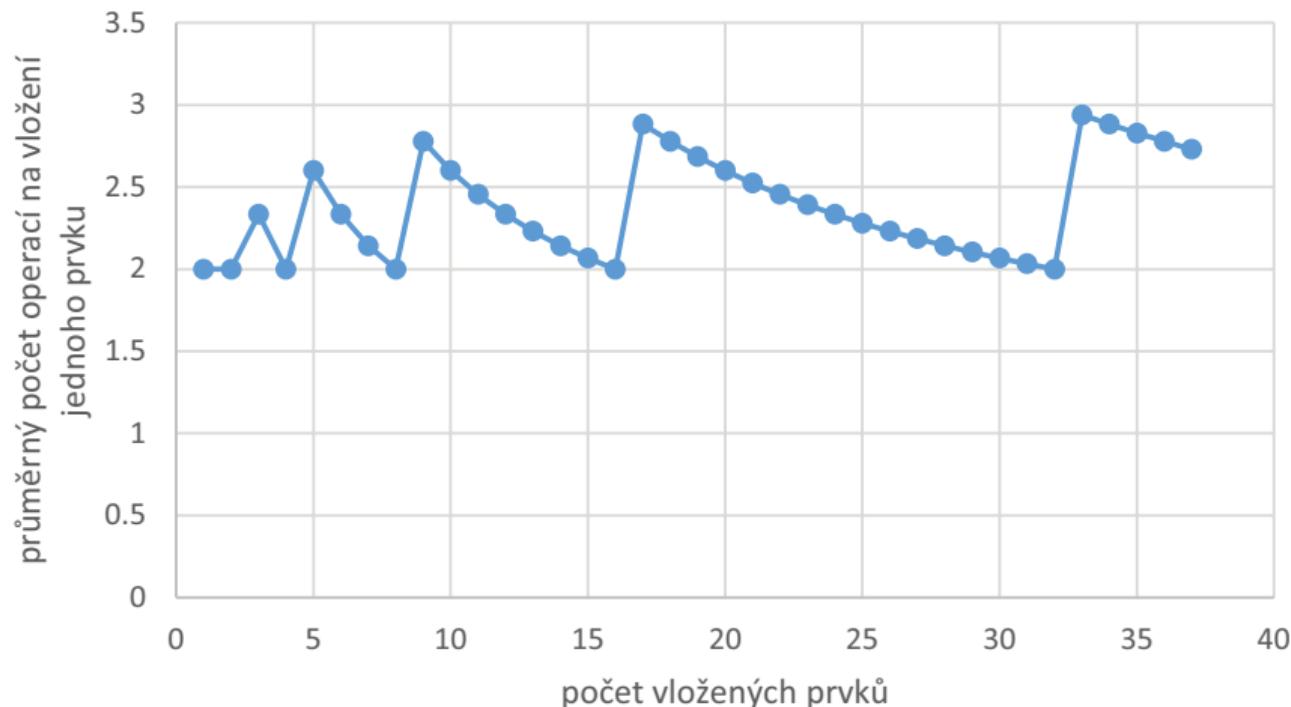
Závěr

- složitost operace "přidání n prvků" je $\mathcal{O}(n)$
- průměrná (amortizovaná) složitost operace "přidání jednoho prvku" je $\Theta(1)$

Celkové počty operací při přidávání



Průměrný počet operací při přidávání



Důležitá poznámka

- tato analýza platí tehdy, když při zvětšování alokujeme dvojnásobné pole
- platila by i při použití jiného násobku (1.5x namísto 2x)

Důležitá poznámka

- tato analýza platí tehdy, když při zvětšování alokujeme dvojnásobné pole
- platila by i při použití jiného násobku (1.5x namísto 2x)
- přizvětšování o konstantu analýza **neplatí!**
 - přidání n prvků je v takovém případě $\Omega(n^2)$
 - přidání jednoho prvku je pak v průměru $\Omega(n)$
 - pro zvětšování o jeden prvek je to zjevné
 - zvětšování o větší počty (o 1000...) nepomůže!

Problém

- přidání prvku je někdy rychlé a někdy pomalé
- občas potřebujeme **záruku**, že operace proběhne v nějakém rozumném čase
- pole někdy zabírá místo v paměti, které vlastně není potřeba

Problém

- přidání prvku je někdy rychlé a někdy pomalé
- občas potřebujeme **záruku**, že operace proběhne v nějakém rozumném čase
- pole někdy zabírá místo v paměti, které vlastně není potřeba

Řešení

Spojový seznam

Spojový seznam

- chceme alokovat velké množství malých kousků paměti
- nemůžeme mít pro každý kousek vlastní identifikátor v programu

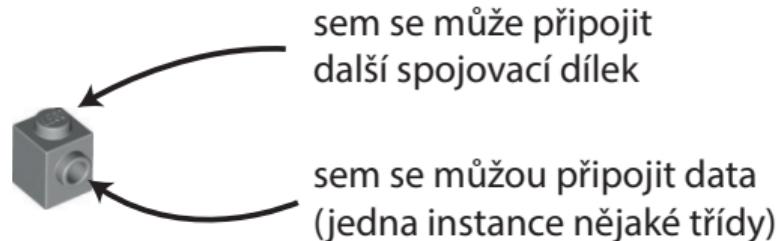
Spojový seznam

- chceme alokovat velké množství malých kousků paměti
- nemůžeme mít pro každý kousek vlastní identifikátor v programu

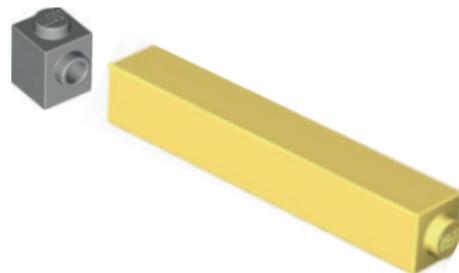
Řešení

- vytvoříme si **spojovací prvek**
- funguje trochu jako lego
- připojuje se na něj **jeden kousek dat**
- může se na něj připojit **další spojovací prvek**

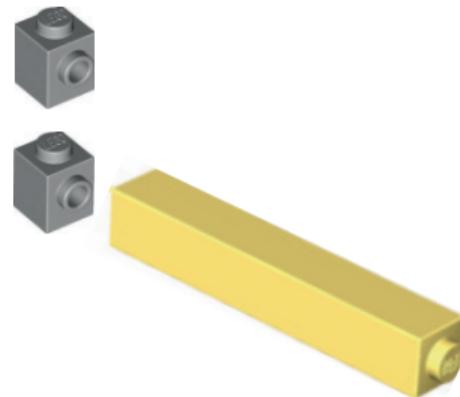
Spojový seznam



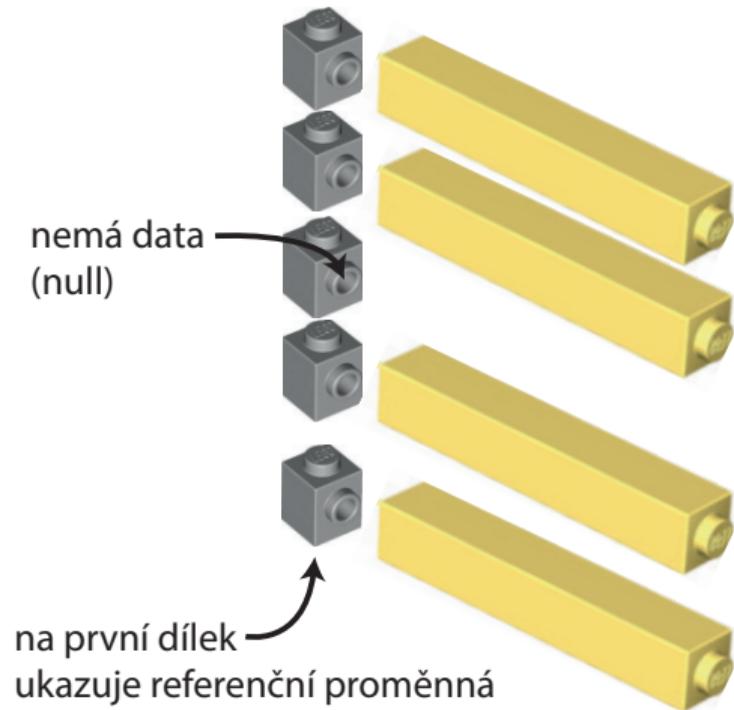
Spojový seznam



Spojový seznam



Spojový seznam



Spojový seznam

- umožňuje alokovat libovolné množství paměti na haldě (heap)
- alokaci je možné dělat po malých částech – pro každý přidávaný prvek
- cena: budeme navíc potřebovat paměť pro spojovací kousky

Třída

Spojovací kousek

```
class Link<T>{
    public T data;
    public Link<T> next;
}
```

- data: vlastní informace (libovolný typ)
- next: referenční proměnná na další kousek dat (pokud nějaký je)
- tato třída je v jistém smyslu rekurzivní (definovaná pomocí sebe sama)

Zásobník implementovaný spojovým seznamem

```
class StackLinkedList<T> : IStack<T>{
    Link<T> top;
    ...
}
```

referenční proměnnou inicializuje C# na null

Přidání prvku

```
void Add(T e) {  
    Link<T> nl = new Link<T>();  
    nl.data = e;  
    nl.next = top;  
    top = nl;  
}
```

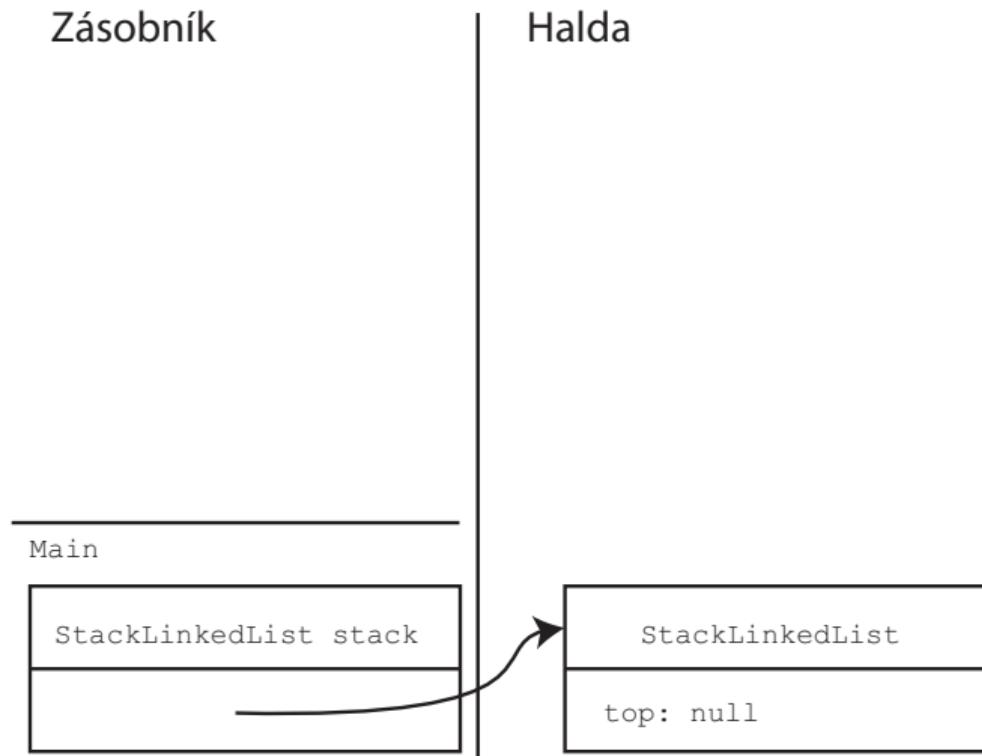
Přidání prvku

```
void Add(T e) {  
    Link<T> nl = new Link<T>();  
    nl.data = e;  
    nl.next = top;  
    top = nl;  
}
```

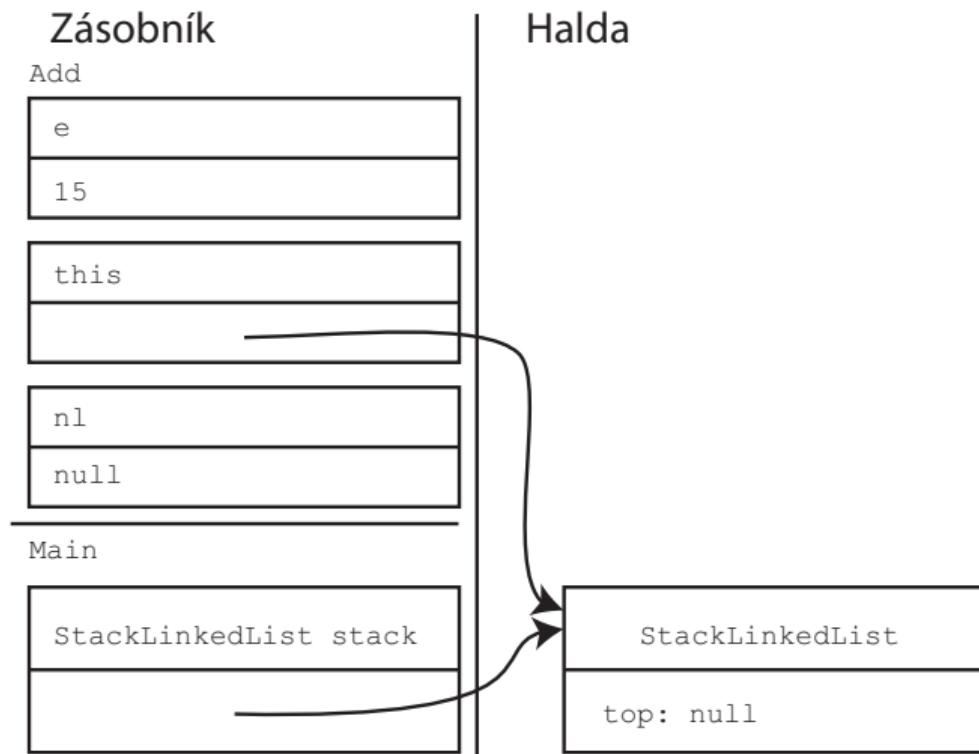
Použití

```
StackLinkedList<int> stack = new StackLinkedList<int>();  
stack.Add(15);  
stack.Add(26);  
stack.Add(57);
```

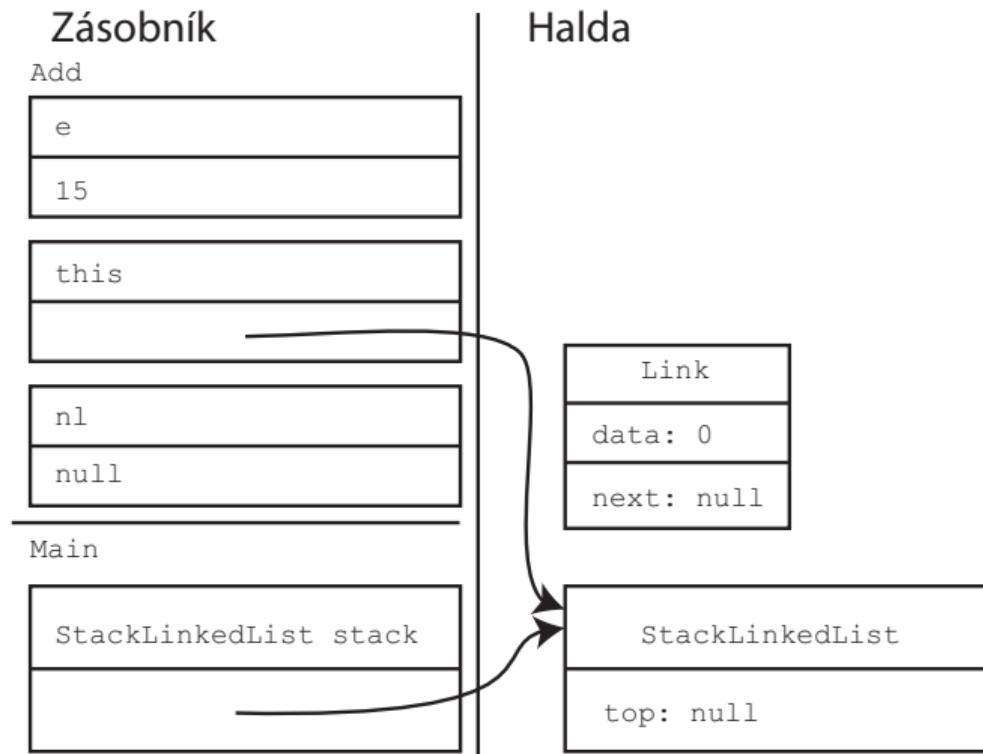
Přidávání prvků do zásobníku



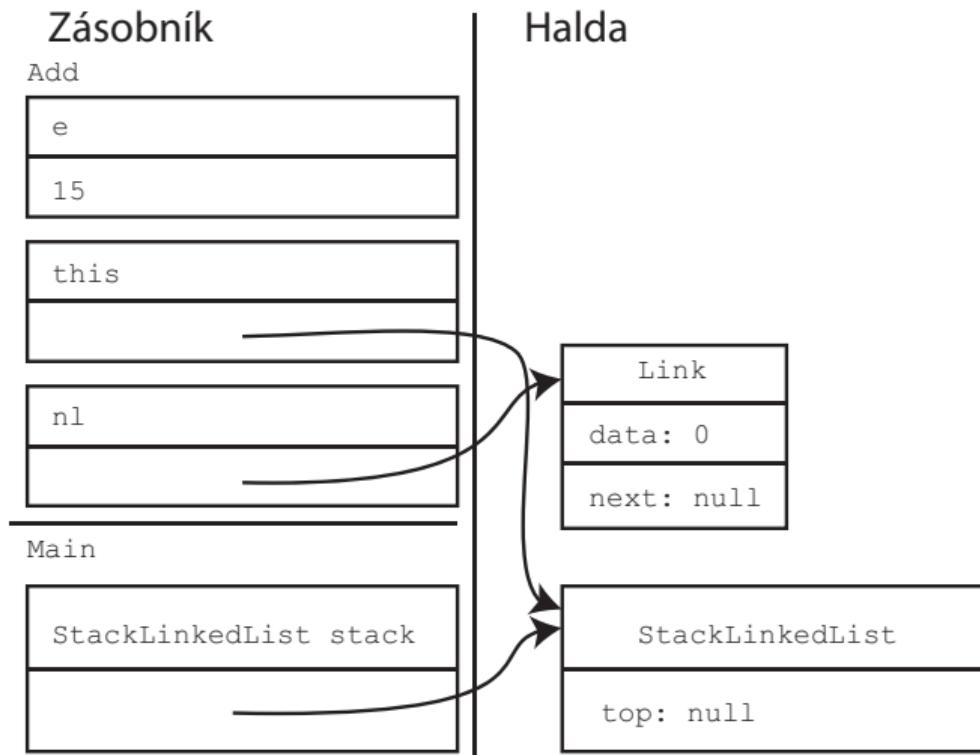
Přidávání prvků do zásobníku



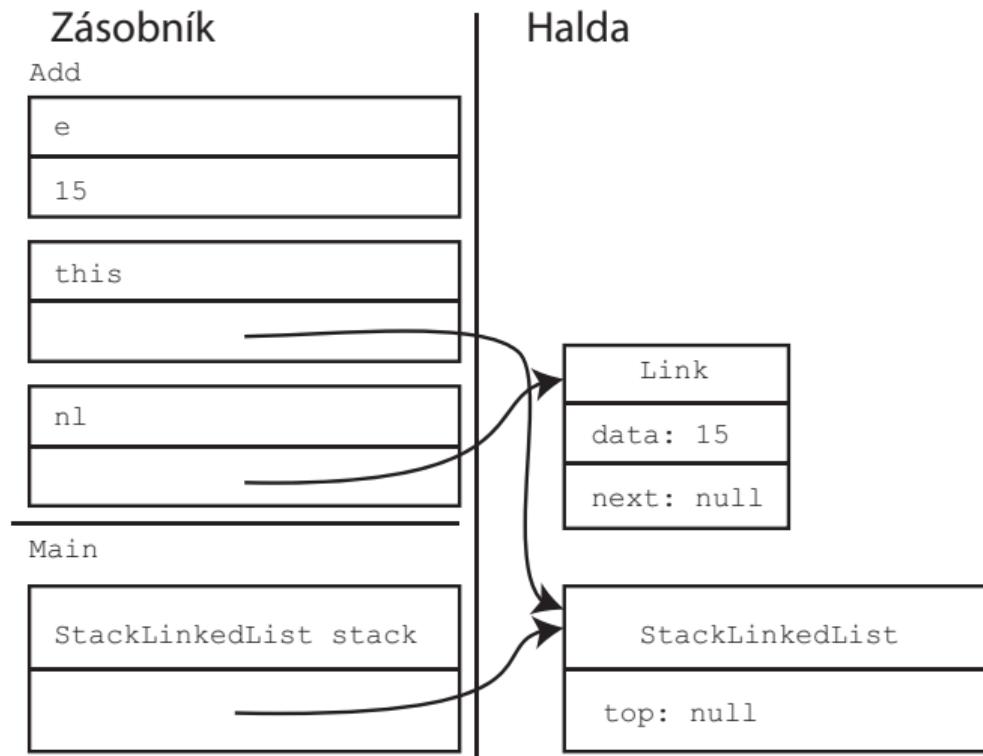
Přidávání prvků do zásobníku



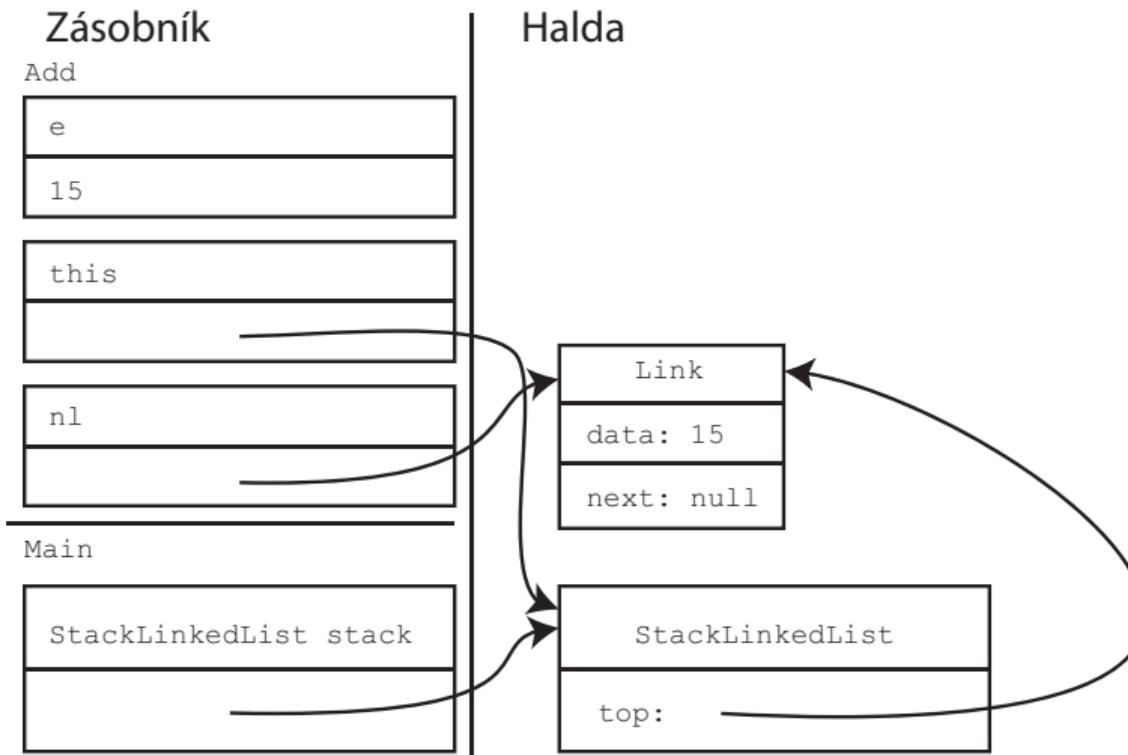
Přidávání prvků do zásobníku



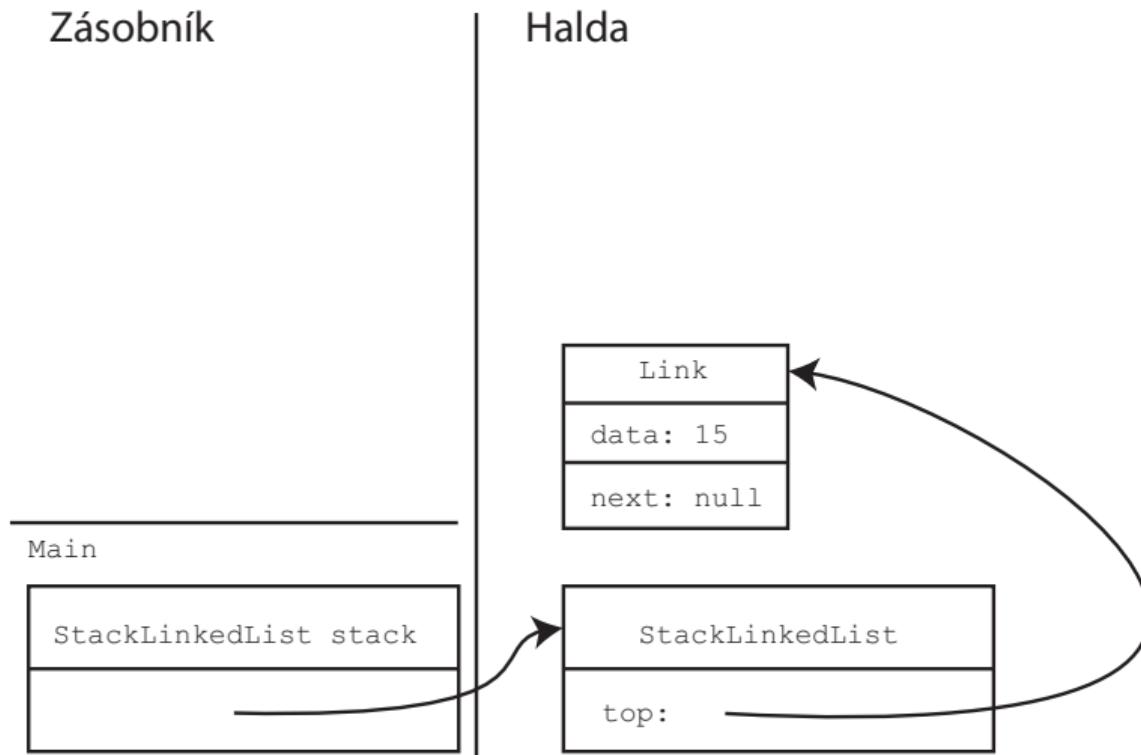
Přidávání prvků do zásobníku



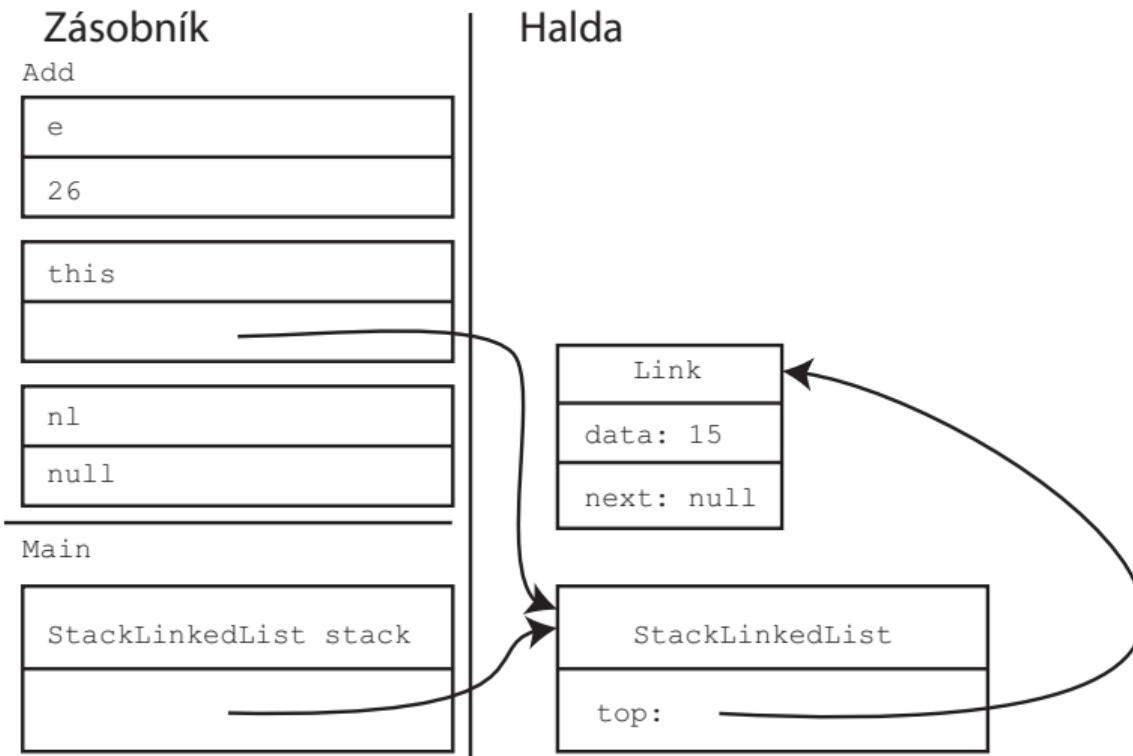
Přidávání prvků do zásobníku



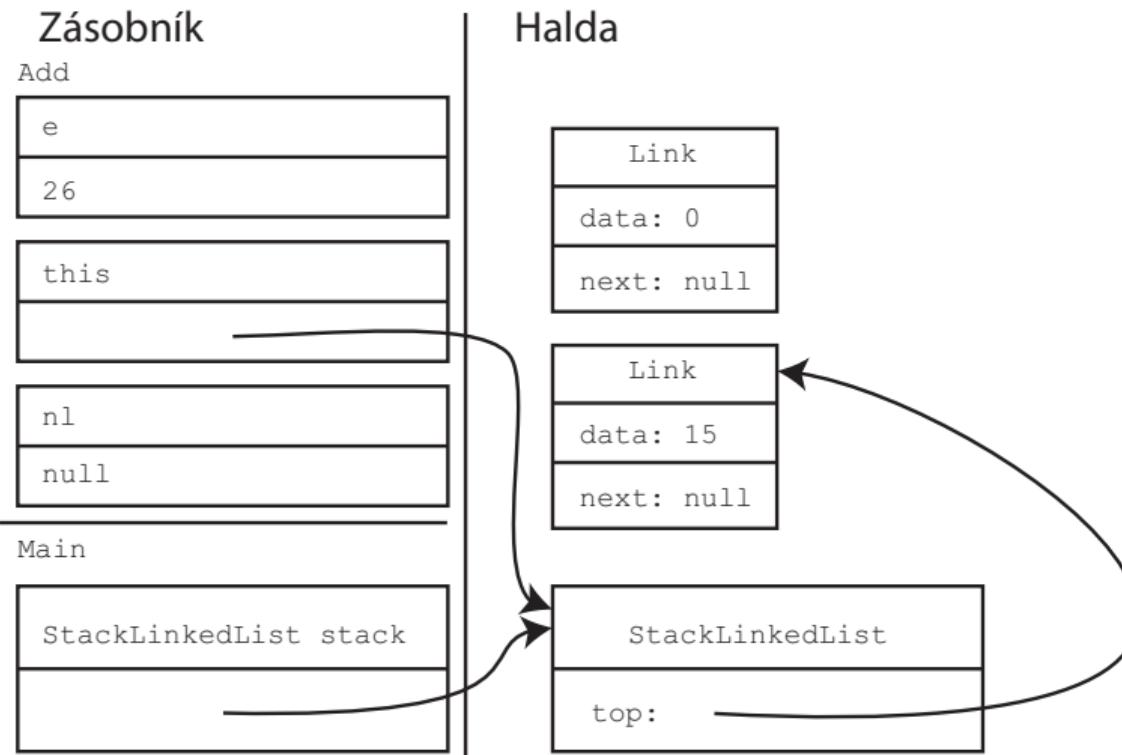
Přidávání prvků do zásobníku



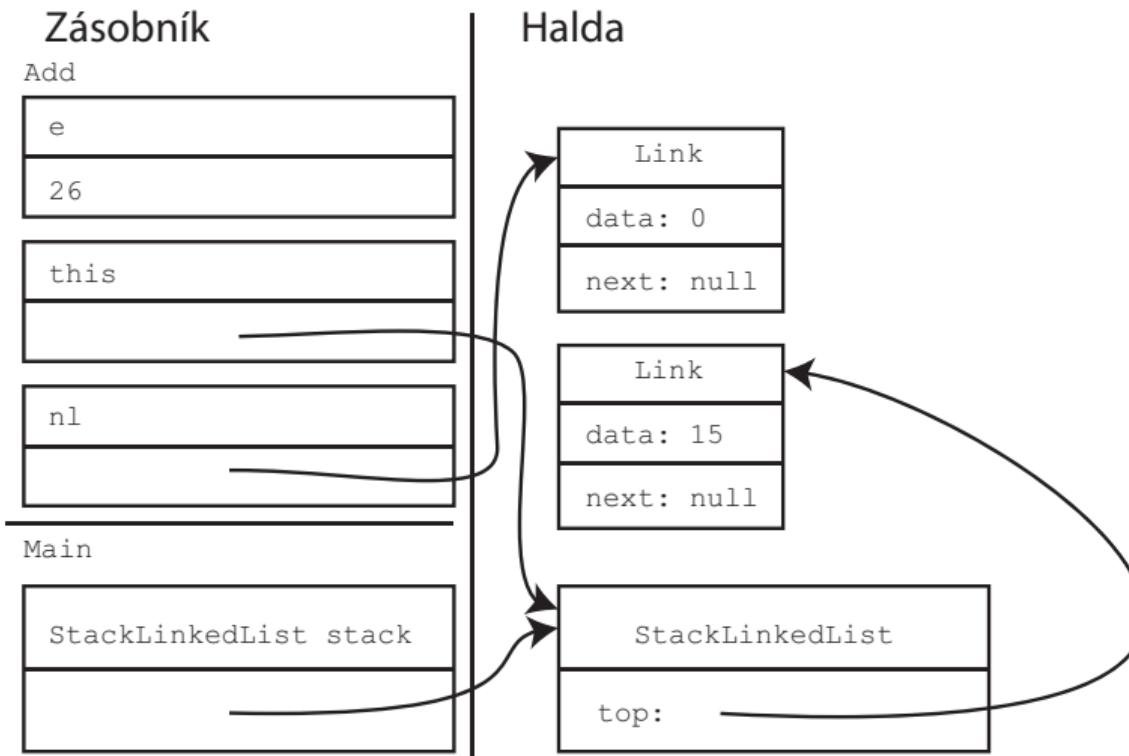
Přidávání prvků do zásobníku



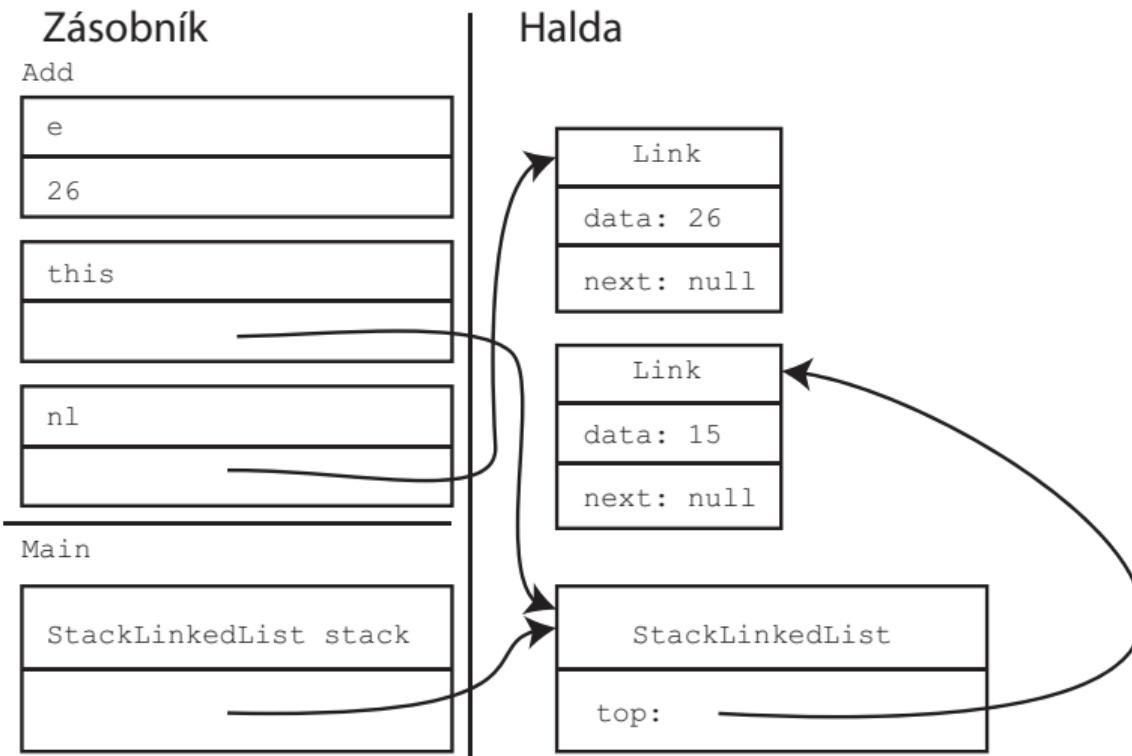
Přidávání prvků do zásobníku



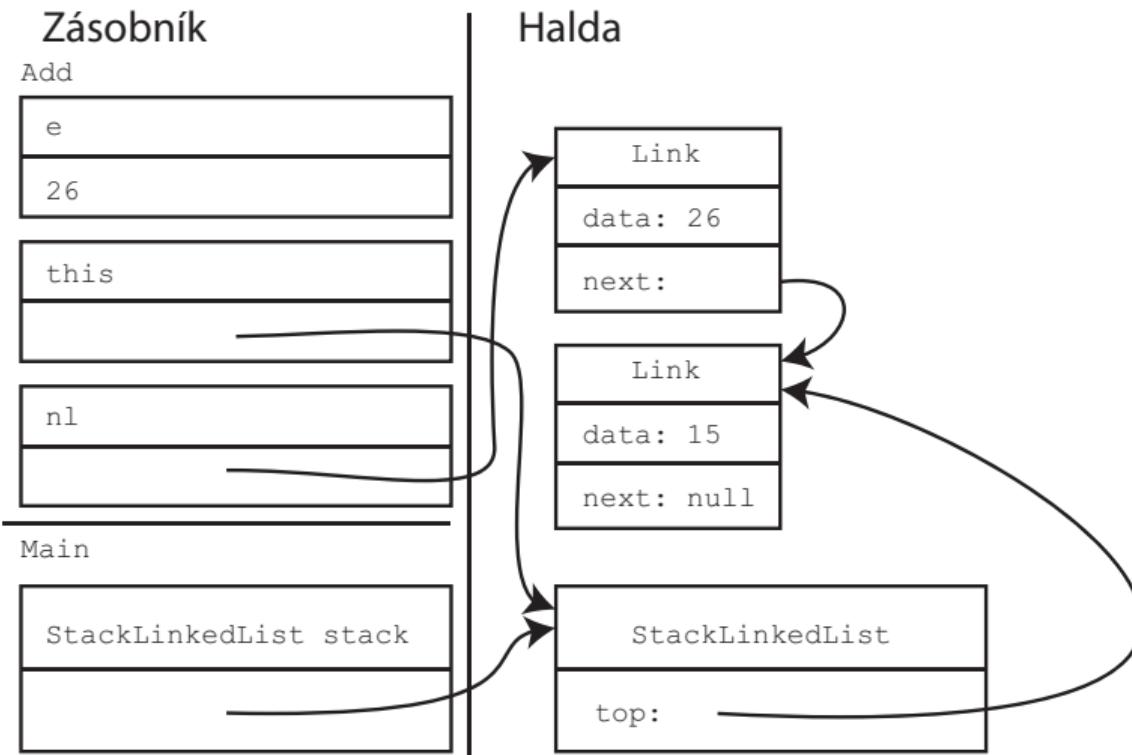
Přidávání prvků do zásobníku



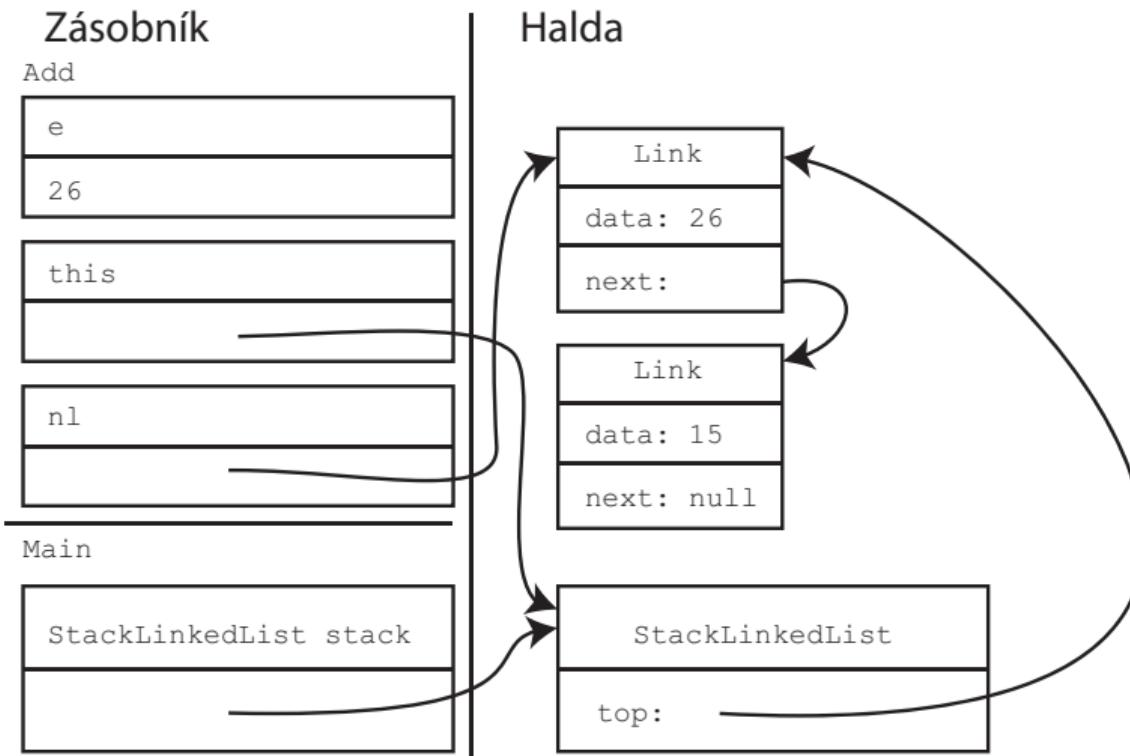
Přidávání prvků do zásobníku



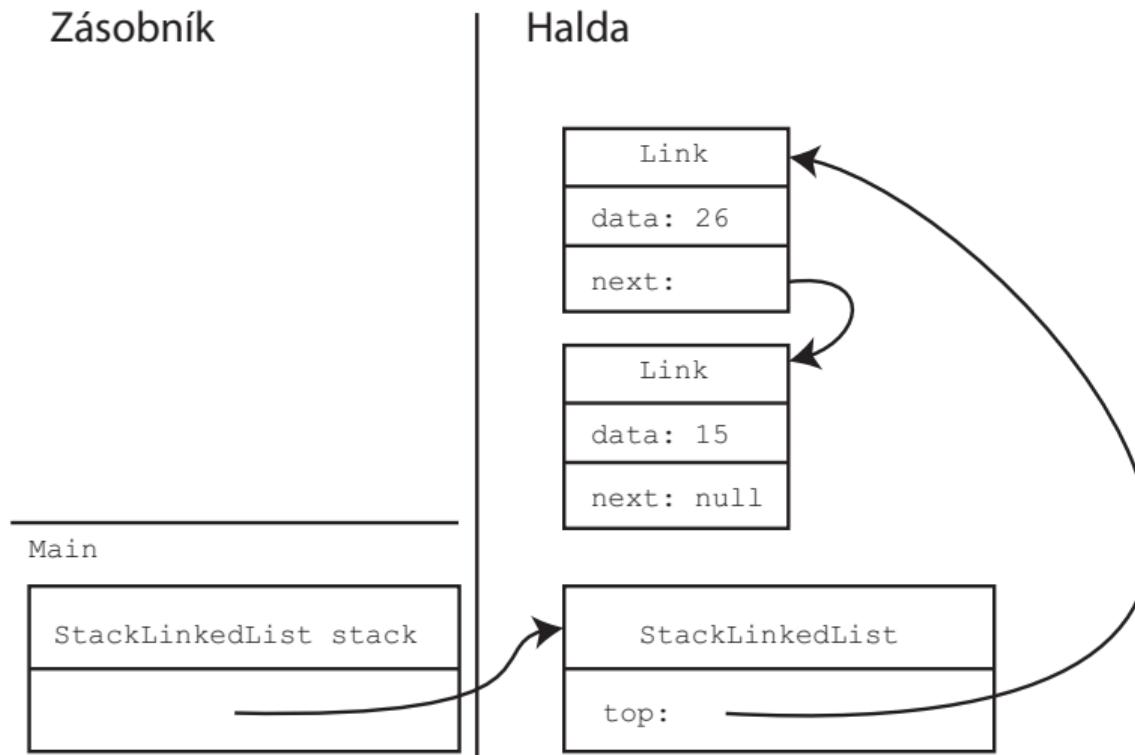
Přidávání prvků do zásobníku



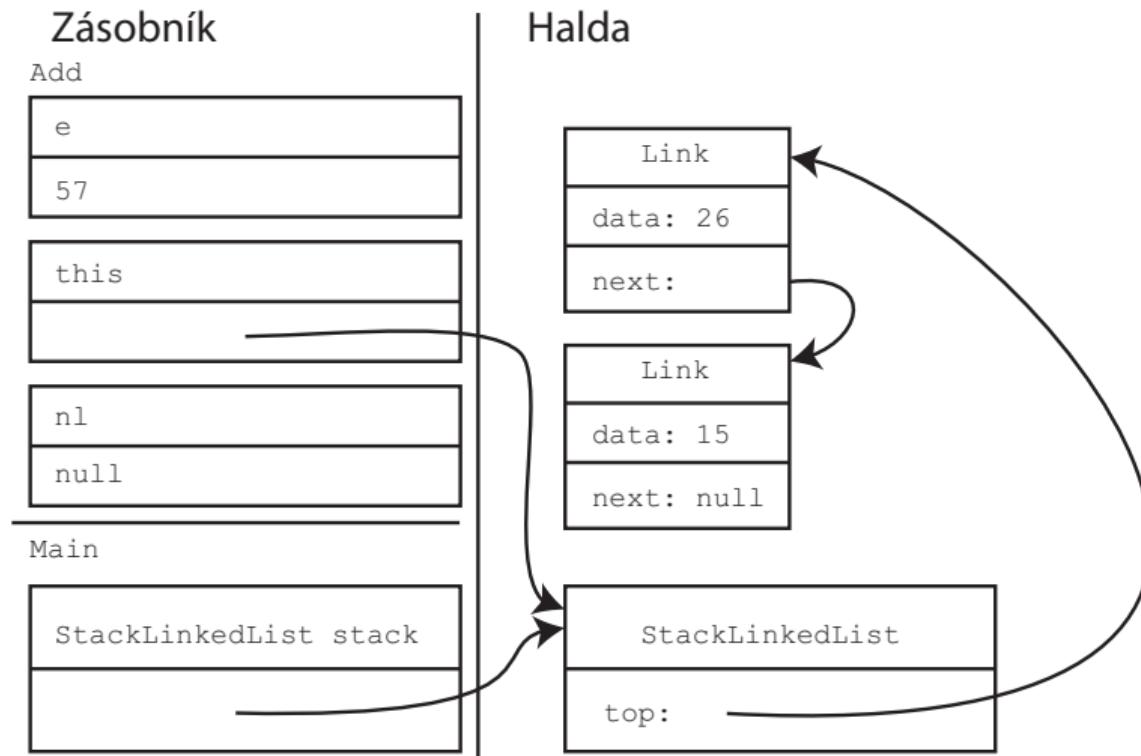
Přidávání prvků do zásobníku



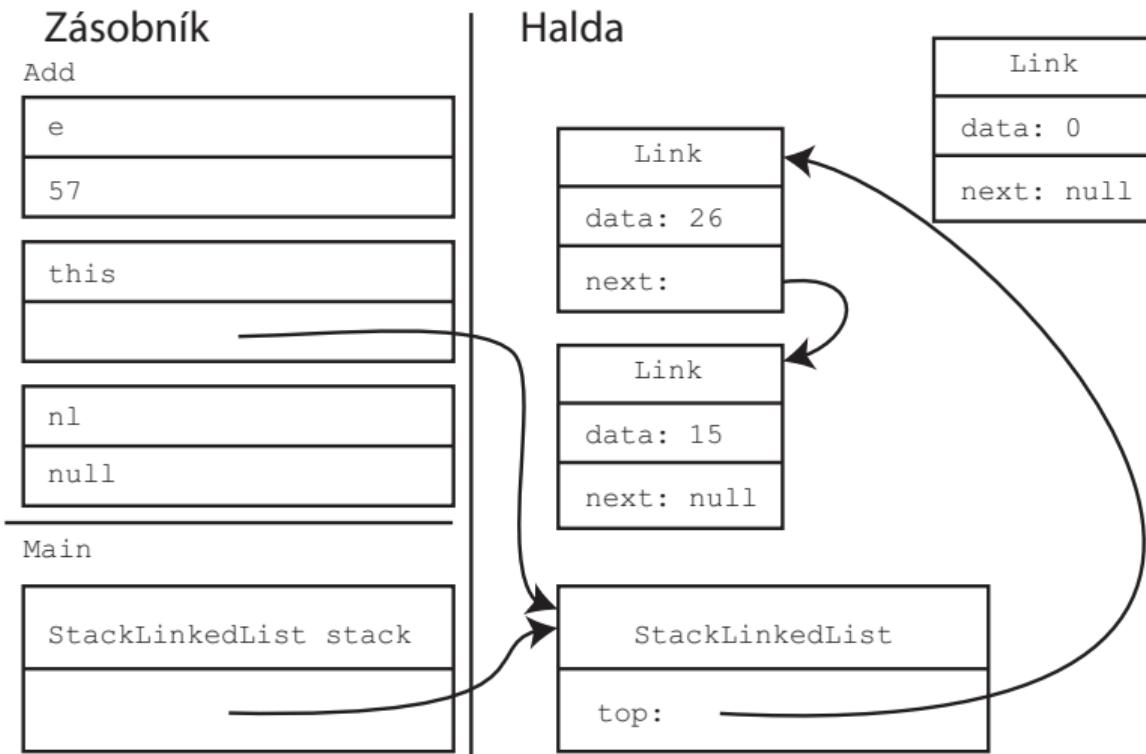
Přidávání prvků do zásobníku



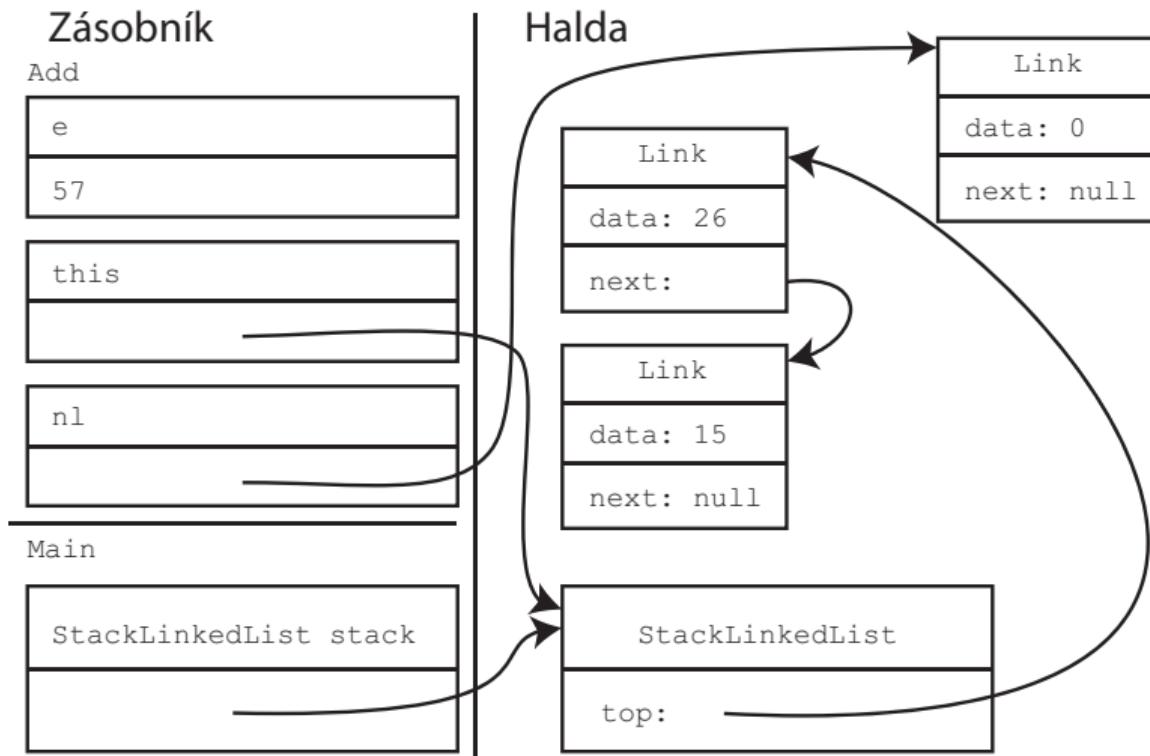
Přidávání prvků do zásobníku



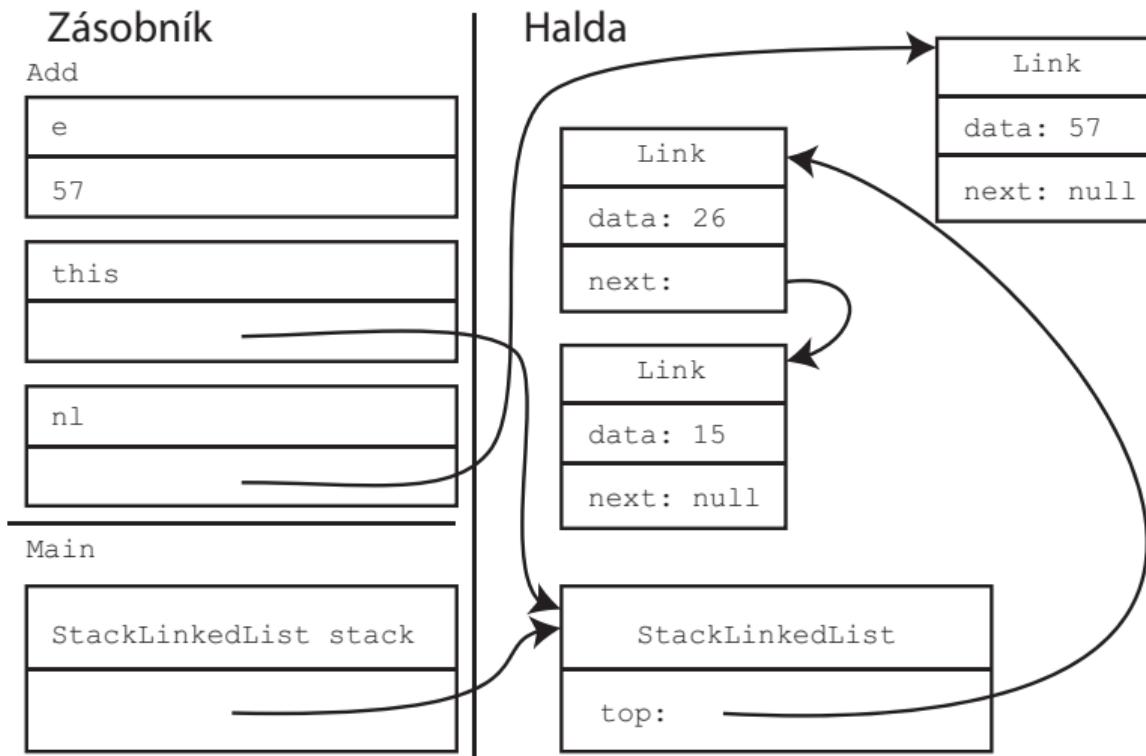
Přidávání prvků do zásobníku



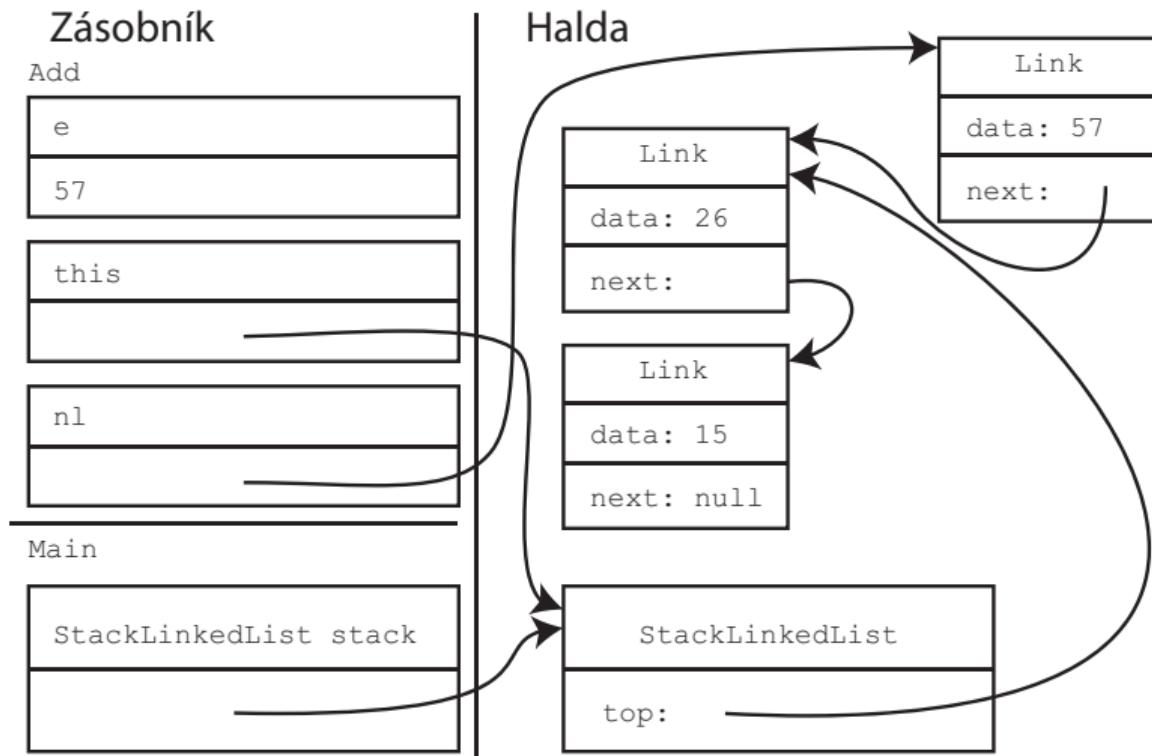
Přidávání prvků do zásobníku



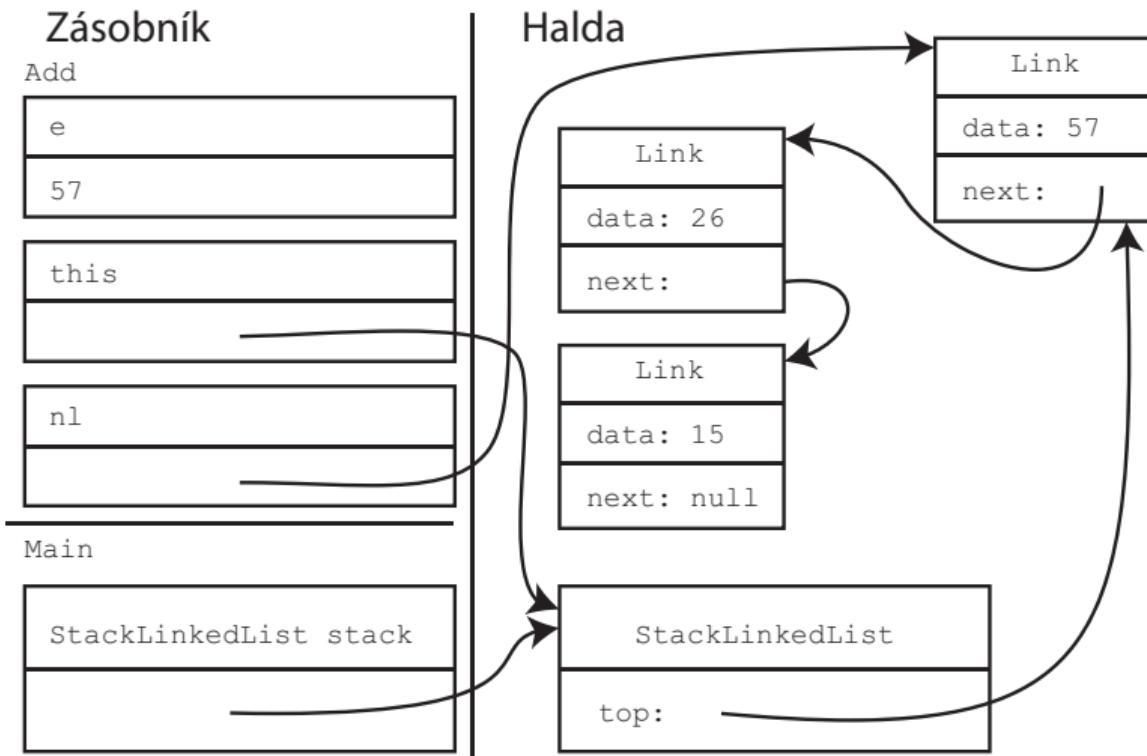
Přidávání prvků do zásobníku



Přidávání prvků do zásobníku



Přidávání prvků do zásobníku



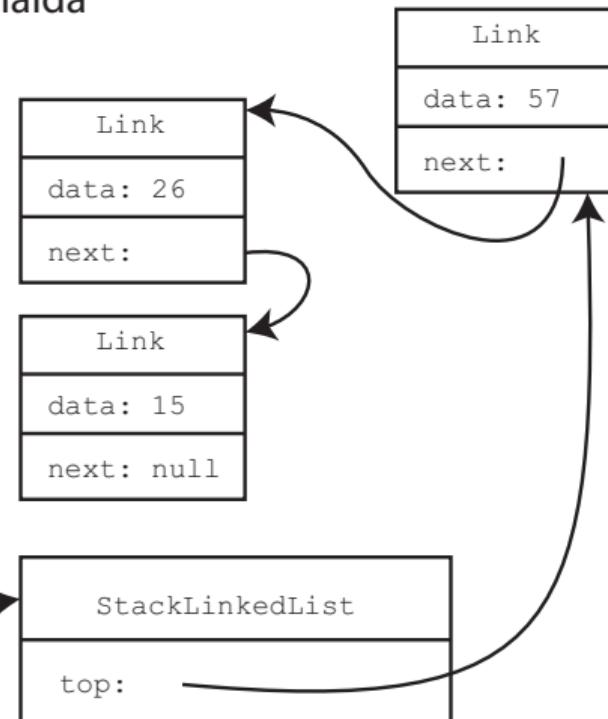
Přidávání prvků do zásobníku

Zásobník

Main



Halda



Vypsání celého zásobníku

```
void Print() {
    Link<T> link = top;
    while (link != null)
    {
        Console.WriteLine(link.data);
        link = link.next;
    }
}
```

(metoda není součástí rozhraní, ale zásobník ji může implementovat)

Vybrání posledního prvku, zjištění, zda je zásobník prázdný

```
T Get () {  
    if (top!=null)  
        return top.data;  
    else throw new Exception();  
}
```

Vybrání posledního prvku, zjištění, zda je zásobník prázdný

```
T Get () {  
    if (top!=null)  
        return top.data;  
    else throw new Exception();  
}  
  
bool IsEmpty () {  
    return (top == null)  
}
```

Odstranění posledního prvku

lze provést v konstantním čase

```
void RemoveLast () {  
    if (top == null)  
        throw new Exception ();  
    top = top.next;  
}
```

Odstranění posledního prvku

lze provést v konstantním čase

```
void RemoveLast () {  
    if (top == null)  
        throw new Exception ();  
    top = top.next;  
}
```

často se provádí společně s vybráním posledního prvku

```
T Pop () {  
    if (top == null)  
        throw new Exception ();  
    T result = top.data;  
    top = top.next;  
    return result;  
}
```

Srovnání implementací

Polem

- přidání prvku může trvat déle
- může zabrat (až 2x) více paměti než je třeba
- trochu rychlejší operace
- umožňuje vybrat prvek na libovolném indexu
 - v konstantním čase
 - nikoli odstranit!

Srovnání implementací

Polem

- přidání prvku může trvat déle
- může zabrat (až 2x) více paměti než je třeba
- trochu rychlejší operace
- umožňuje vybrat prvek na libovolném indexu
 - v konstantním čase
 - nikoli odstranit!

Spojovým seznamem

- přidání prvku vždy v konstantním čase
- paměť navíc pro spojovací struktury
- trochu pomalejší operace (práce s referenčními proměnnými, alokace paměti při každém přidání prvku)
 - může být poměrně velká, pokud jsou data malá (např. int, byte, ...)
- neumožňuje v konstantním čase vybrat prvek na libovolném indexu

Otázky

- jak velké jsou položky v seznamu?
 - jsou-li malé, pak bude lepší implementace polem, kvůli šetření paměti
 - je to ale praktický problém?
- potřebuji zaručenou rychlosť pro přidávání prvku?
 - přidání při implementaci polem může zabrat nějaký čas

Otázky

- jak velké jsou položky v seznamu?
 - jsou-li malé, pak bude lepší implementace polem, kvůli šetření paměti
 - je to ale praktický problém?
- potřebuji zaručenou rychlosť pro přidávání prvku?
 - přidání při implementaci polem může zabrat nějaký čas

Důležité pozorování

Tyto aspekty není třeba se učit! Vyplývají ze způsobu implementace!

Použití zásobníku

V různých implementacích je možné doplnit další funkce, např. v implementaci polem:

```
T ElementAt (int index) {  
    if ((index<freeIndex) && (index>=0))  
        return array[freeIndex - index - 1];  
    else throw new Exception();  
}
```

Použití zásobníku

nebo v implementaci spojovým seznamem:

```
T ElementAt (int index) {  
    Link<T> lnk = top;  
    for (int i = 0; i < index; i++) {  
        if (lnk == null)  
            throw new Exception();  
        else lnk = lnk.next;  
    }  
    if (lnk != null)  
        return lnk.data;  
    else throw new Exception();  
}
```

Odstranění rekurze uživatelským zásobníkem

Proč odstraňovat rekurzi?

- rekurze je vhodná pro návrh algoritmů
- rekurzivní kód bývá dobře srozumitelný

Proč odstraňovat rekurzi?

- rekurze je vhodná pro návrh algoritmů
- rekurzivní kód bývá dobře srozumitelný (...? ☺)

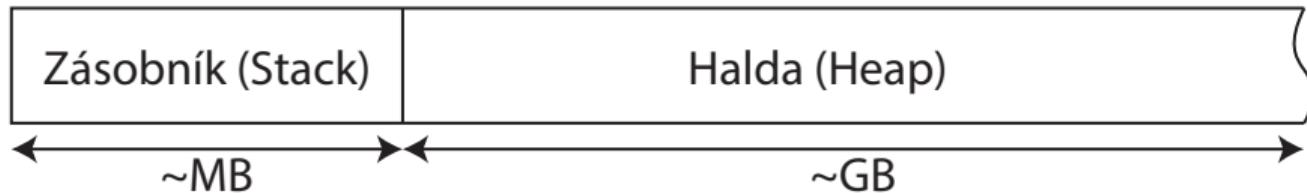
Proč odstraňovat rekurzi?

- rekurze je vhodná pro návrh algoritmů
- rekurzivní kód bývá dobře srozumitelný (...? ☺)

ALE

- problémy s hloubkou zanoření
- při volání metod se kopírují hodnoty skutečných parametrů do zásobníku (neefektivní!)

Organizace paměti



- zásobník uložen na začátku paměti
- velikost zásobníku je omezená
- halda za ním
- halda je podstatně větší než zásobník
- nedostatek paměti v haldě vesměs značí hardwarový nedostatek paměti
- OS řeší nedostatek paměti swapováním, ale pouze pro haldu

Lze rekurzi odstranit?

Někdy je to snadné

rekurzivní pseudokód:

```
void RecursiveFunction(int x) {  
    if (Trivial(x))  
        return;  
    else {  
        int y = SomeFunction(x);  
        RecursiveFunction(y);  
    }  
}
```

můžeme přepsat jako

```
void NonRecursiveFunction(int x) {  
    while (!Trivial(x))  
        x = SomeFunction(x);  
}
```

Koncová rekurze

- kód končí jediným rekurzivním voláním
- nejjednodušší případ
- přepíšeme na smyčku

Koncová rekurze

- kód končí jediným rekurzivním voláním
- nejjednodušší případ
- přepíšeme na smyčku

Co ale dělat, když

- rekurze není na konci
- rekurzivních volání je v kódu více

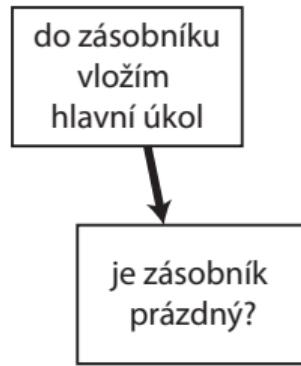
Eliminace vlastním zásobníkem

- použijeme zásobník jako seznam úkolů
- úkol může být v nějaké fázi rozpracovanosti (segment)
- položka v zásobníku musí obsahovat:
 - co se má udělat (parametry)
 - jak jsme s úkolem daleko (segment)
 - data nutná pro pokračování výpočtu (mezivýsledky) - stavové proměnné

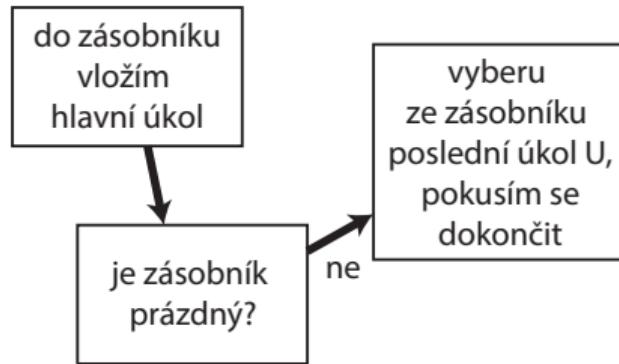
Postup

do zásobníku
vložím
hlavní úkol

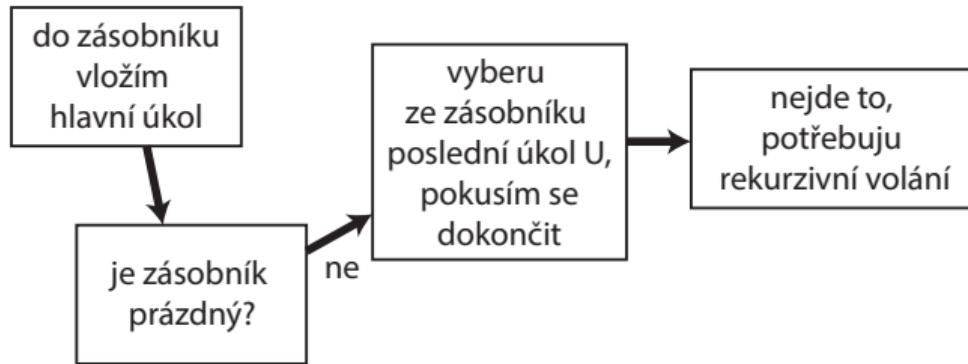
Postup



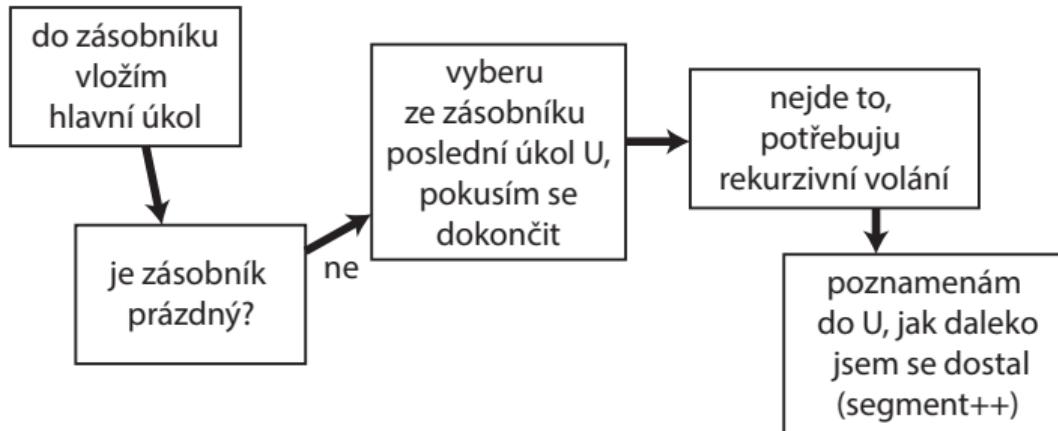
Postup



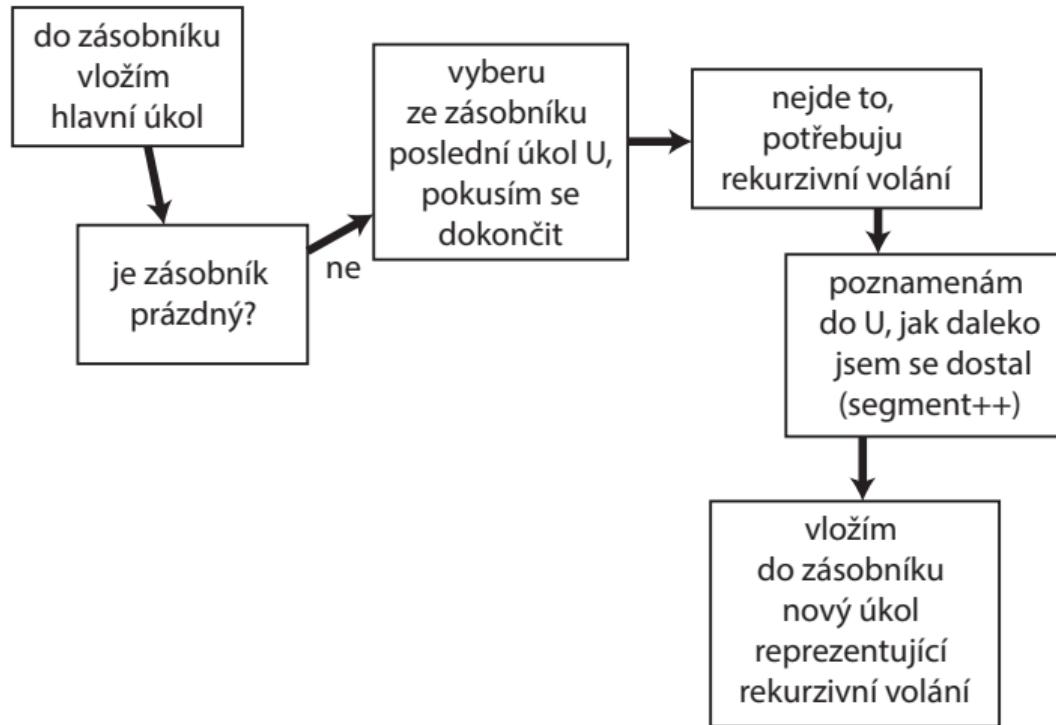
Postup



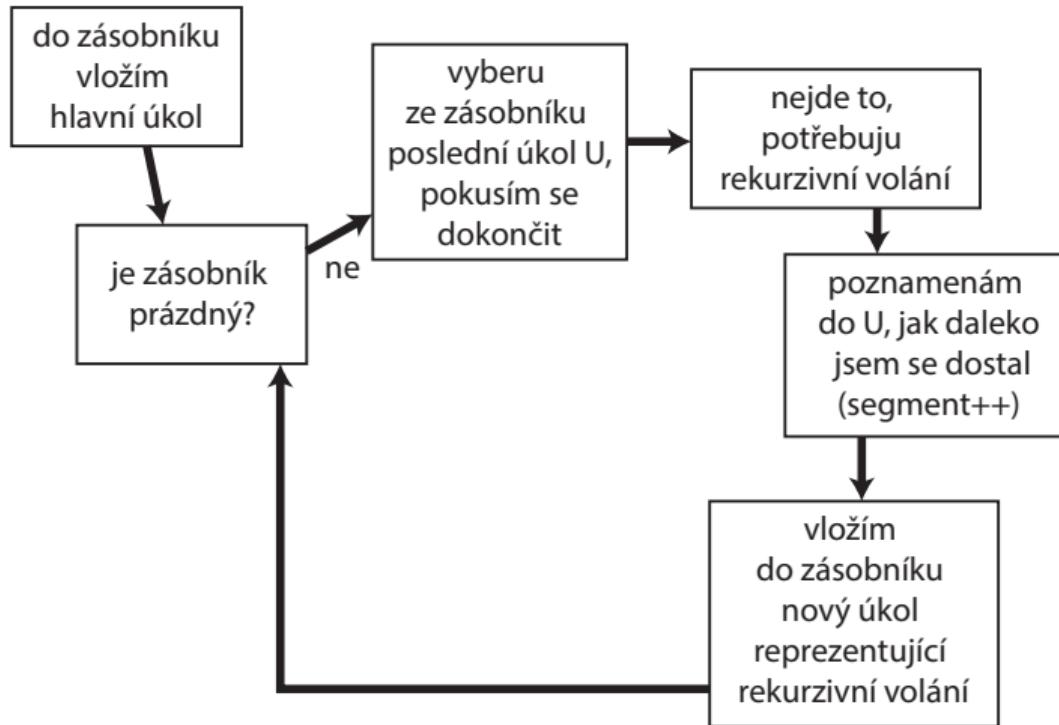
Postup



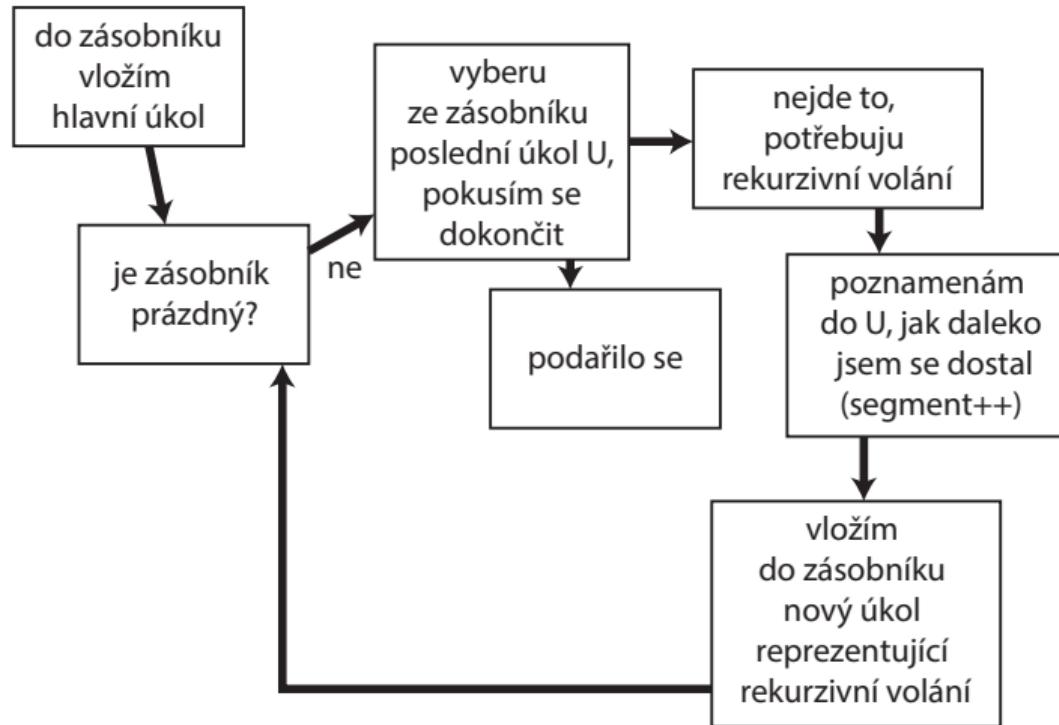
Postup



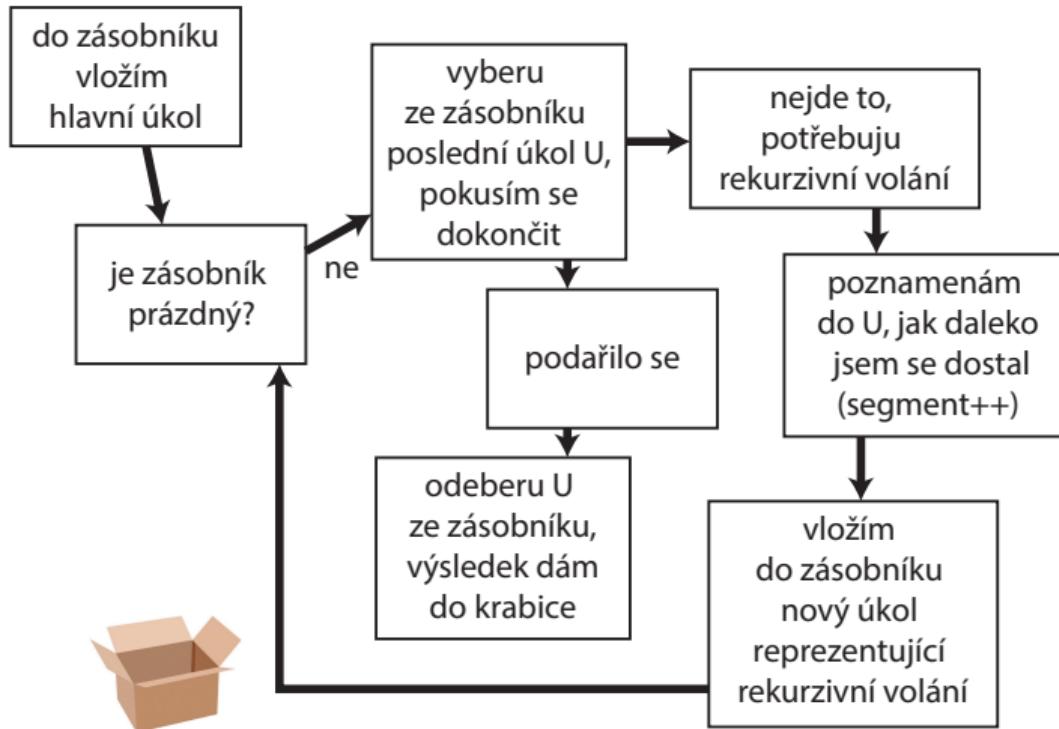
Postup



Postup



Postup



Postup

je zásobník
prázdný?

do zásobníku
vložím
hlavní úkol

vyberu
ze zásobníku
poslední úkol U,
pokusím se
dokončit

nejde to,
potřebuju
rekurzivní volání

podařilo se

poznamenám
do U, jak daleko
jsem se dostal
(segment++)

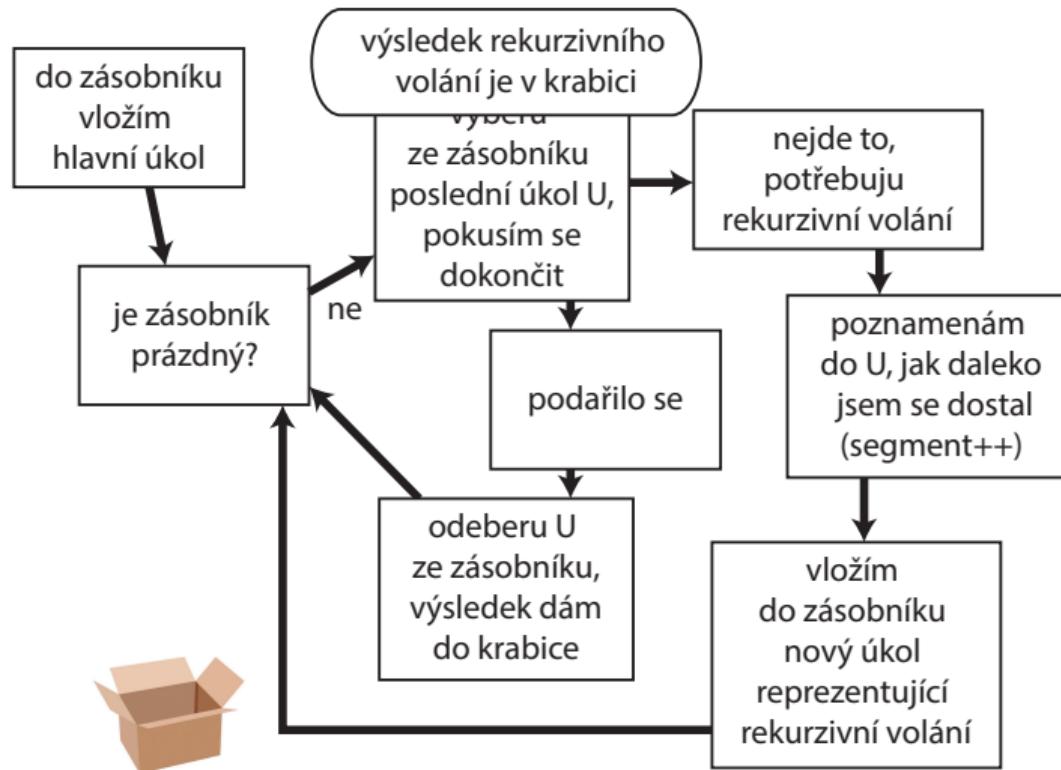
odeberu U
ze zásobníku,
výsledek dám
do krabice

vložím
do zásobníku
nový úkol
reprezentující
rekurzivní volání

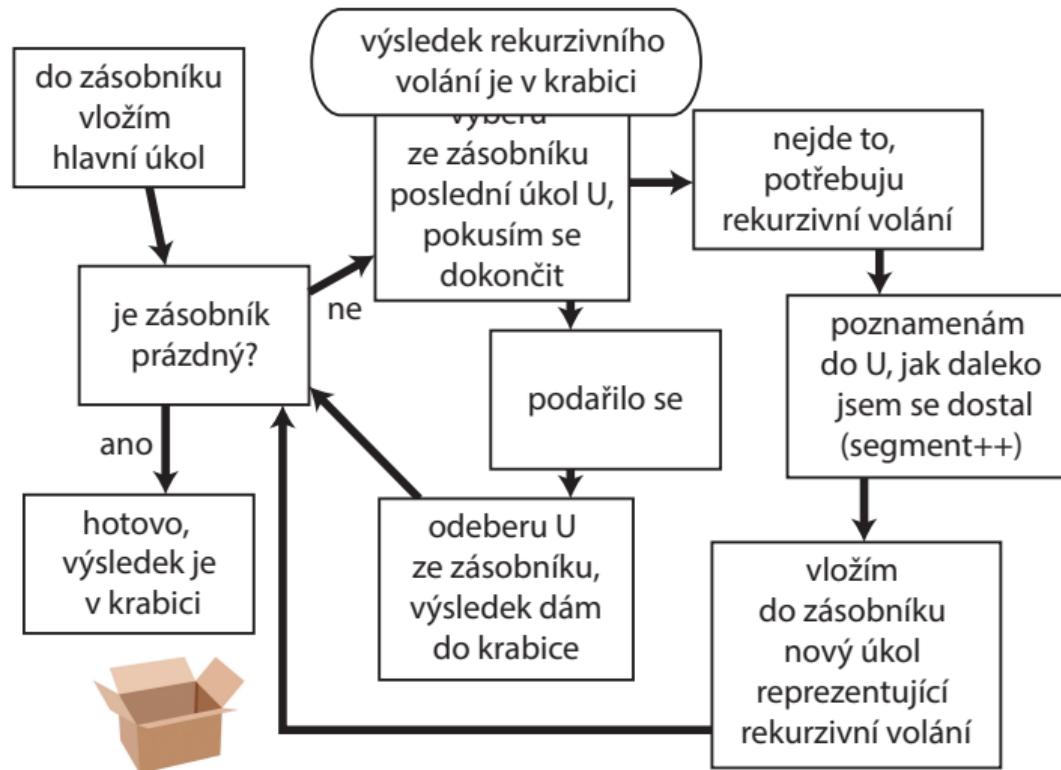


ne

Postup



Postup



Příklad

Rekurzivní kód

```
int Compute(int p) {
    if (Trivial(p))
        return trivialValue;
    int t1 = SomeFunction(p);
    int t2 = Compute(t1);
    int t3 = OtherFunction(t2);
    int t4 = Compute(t3);
    int result = FinalFunction(t4);
    return result;
}
```

Příklad

Rekurzivní kód

```
int Compute(int p){  
    if (Trivial(p))  
        return (trivialValue);  
    int t1 = SomeFunction(p);  
    int t2 = Compute(t1);  
    int t3 = OtherFunction(t2);  
    int t4 = Compute(t3);  
    int result = FinalFunction(t4);  
    return result;  
}
```

segment 0

Příklad

Rekurzivní kód

```
int Compute(int p){  
    if (Trivial(p))  
        return (trivialValue);  
    int t1 = SomeFunction(p);  
    int t2 = Compute(t1);  
    int t3 = OtherFunction(t2);  
    int t4 = Compute(t3);  
    int result = FinalFunction(t4);  
    return result;  
}
```

segment 0

segment 1

Příklad

Rekurzivní kód

```
int Compute(int p){  
    if (Trivial(p))  
        return (trivialValue);  
    int t1 = SomeFunction(p);  
    int t2 = Compute(t1);  
    int t3 = OtherFunction(t2);  
    int t4 = Compute(t3);  
    int result = FinalFunction(t4);  
    return result;  
}
```

segment 0

segment 1

segment 2

Záznam v zásobníku

```
class Task{  
    public int parameter;  
    public int segment;  
  
    public Task(int p) {  
        this.parameter = p;  
    }  
}
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(p));
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {
    IStack<Task> stack = new StackArray<Task>();
    stack.Add(new Task(p));
    int result = 0;
```

Nerekurzivní verze

```
int ComputeNonRecursive (p) {  
    IStack<Task> stack = new StackArray<Task> ();  
    stack.Add (new Task (p));  
    int result = 0;  
    while (stack.IsEmpty () == false) {  
        Task t = stack.Get ();
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {
    IStack<Task> stack = new StackArray<Task>();
    stack.Add(new Task(p));
    int result = 0;
    while(stack.IsEmpty() == false) {
        Task t = stack.Get();
        switch (t.segment) {
            0: if (Trivial(t.parameter)) {
                result = trivialValue;
                stack.RemoveLast();
                break;
            }
            int t1 = SomeFunction(t.parameter)
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {
    IStack<Task> stack = new StackArray<Task>();
    stack.Add(new Task(p));
    int result = 0;
    while(stack.IsEmpty() == false) {
        Task t = stack.Get();
        switch (t.segment) {
            0: if (Trivial(t.parameter)) {
                result = trivialValue;
                stack.RemoveLast();
                break;
            }
            int t1 = SomeFunction(t.parameter)
            t.segment = 1;
            stack.Add(new Task(t1));
            break;
        }
    }
}
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {
    IStack<Task> stack = new StackArray<Task>();
    stack.Add(new Task(p));
    int result = 0;
    while (stack.IsEmpty() == false) {
        Task t = stack.Get();
        switch (t.segment) {
            0: if (Trivial(t.parameter)) {
                result = trivialValue;
                stack.RemoveLast();
                break;
            }
            int t1 = SomeFunction(t.parameter)
            t.segment = 1;
            stack.Add(new Task(t1));
            break;
        }
    }
}
```

```
1: int t3 = OtherFunction(result);
    t.segment = 2
    stack.Add(new Task(t3));
    break;
```

Nerekurzivní verze

```
int ComputeNonRecursive(p) {
    IStack<Task> stack = new StackArray<Task>();
    stack.Add(new Task(p));
    int result = 0;
    while(stack.IsEmpty() == false) {
        Task t = stack.Get();
        switch (t.segment) {
            0: if (Trivial(t.parameter)) {
                result = trivialValue;
                stack.RemoveLast();
                break;
            }
            int t1 = SomeFunction(t.parameter)
            t.segment = 1;
            stack.Add(new Task(t1));
            break;
        }
    }
}
```

```
1: int t3 = OtherFunction(result);
   t.segment = 2
   stack.Add(new Task(t3));
   break;
2: result = FinalFunction(result);
   stack.RemoveLast();
   break;
}
}
return result;
```

Pravidla

- rozdělíme kód na segmenty mezi rekurzivními voláními

- rozdělíme kód na segmenty mezi rekurzivními voláními
- na začátku vložíme do zásobníku „hlavní“ úkol

- rozdělíme kód na segmenty mezi rekurzivními voláními
- na začátku vložíme do zásobníku „hlavní“ úkol
- zpracováváme položky zásobníku, dokud není prázdný

- rozdělíme kód na segmenty mezi rekurzivními voláními
- na začátku vložíme do zásobníku „hlavní“ úkol
- zpracováváme položky zásobníku, dokud není prázdný
- namísto rekurzivního volání:
 - uložíme stav výpočtu do položky na vrcholu zásobníku (stavové proměnné, jsou-li)
 - inkrementujeme segment na vrcholu zásobníku (s úkolem jsme teď o krok dál)
 - vložíme do zásobníku nový úkol reprezentující rekurzivní volání
 - ukončíme zpracovávání aktuálního úkolu (smyčka se k němu vrátí později)

- namísto vrácení hodnoty:
 - uložíme výsledek do proměnné `result`
 - odstraníme úkol z vrcholu zásobníku (je hotový)

- namísto vrácení hodnoty:
 - uložíme výsledek do proměnné `result`
 - odstraníme úkol z vrcholu zásobníku (je hotový)
- při pokračování dalším segmentem:
 - obnovíme stavové proměnné z položky na vrcholu zásobníku
 - výsledek posledního rekurzivního volání je uložen v proměnné `result`

- namísto vrácení hodnoty:
 - uložíme výsledek do proměnné `result`
 - odstraníme úkol z vrcholu zásobníku (je hotový)
- při pokračování dalším segmentem:
 - obnovíme stavové proměnné z položky na vrcholu zásobníku
 - výsledek posledního rekurzivního volání je uložen v proměnné `result`
- na konci vrátíme obsah proměnné `result`

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = 2$$

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = 2$$

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = 2$$

Rekurzivní program:

```
int Fibonacci(int n) {  
    if (n<2)  
        return (n);
```

Příklad

Výpočet Fibonacciho čísla

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = 2$$

Rekurzivní program:

```
int Fibonacci(int n) {
    if (n<2)
        return (n);
    int f1 = Fibonacci(n-1);
    int f2 = Fibonacci(n-2);
    return (f1+f2);
}
```

Nerekurzivní verze

Třída pro záznam v zásobníku

```
class Task{  
    public int parameter;  
    public int segment;  
    public int f1;  
  
    public Task(int p) {  
        this.parameter = p;  
    }  
}
```

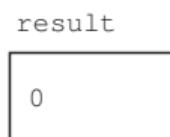
obsahuje stavovou proměnnou f1

Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment) {  
            case 0: if (t.parameter < 2) {  
                result = t.parameter;  
                stack.RemoveLast();  
                break;  
            }  
            t.segment = 1;  
            stack.Add(new Task(t.parameter - 1));  
            break;  
            case 1: t.f1 = result;  
            t.segment = 2;  
            stack.Add(new Task(t.parameter - 2));  
            break;  
            case 2: result = t.f1 + result;  
            stack.RemoveLast();  
            break;  
        }  
    }  
    return result;  
}
```

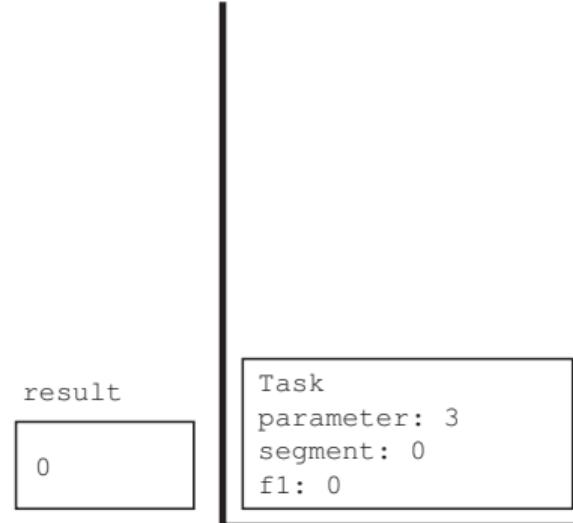
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment) {  
            case 0: if (t.parameter < 2) {  
                result = t.parameter;  
                stack.RemoveLast();  
                break;  
            }  
            t.segment = 1;  
            stack.Add(new Task(t.parameter - 1));  
            break;  
            case 1: t.f1 = result;  
            t.segment = 2;  
            stack.Add(new Task(t.parameter - 2));  
            break;  
            case 2: result = t.f1 + result;  
            stack.RemoveLast();  
            break;  
        }  
    }  
    return result;  
}
```



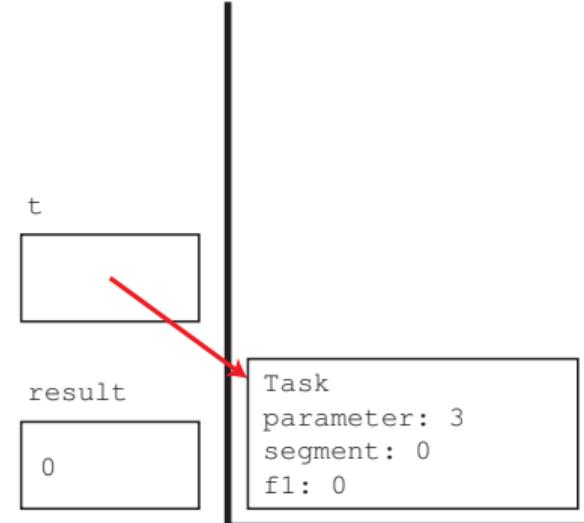
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                result = t.parameter;  
                stack.RemoveLast();  
                break;  
            }  
            t.segment = 1;  
            stack.Add(new Task(t.parameter-1));  
            break;  
            case 1: t.f1 = result;  
                t.segment = 2;  
                stack.Add(new Task(t.parameter-2));  
                break;  
            case 2: result = t.f1 + result;  
                stack.RemoveLast();  
                break;  
        }  
    }  
    return result;  
}
```



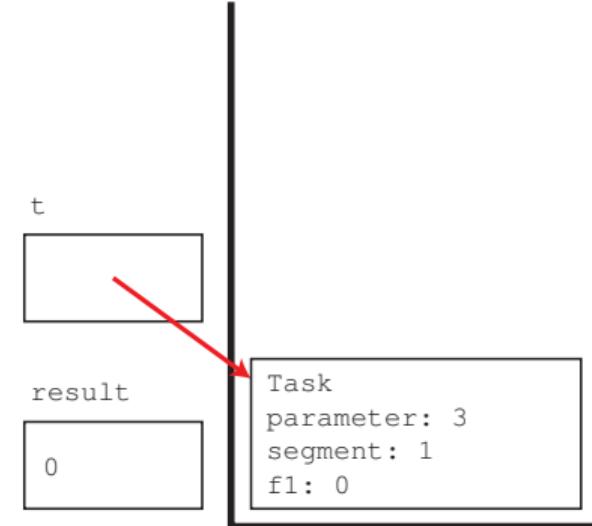
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



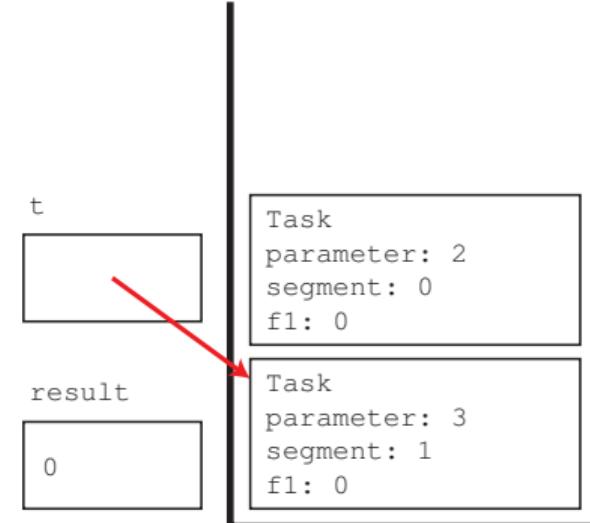
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



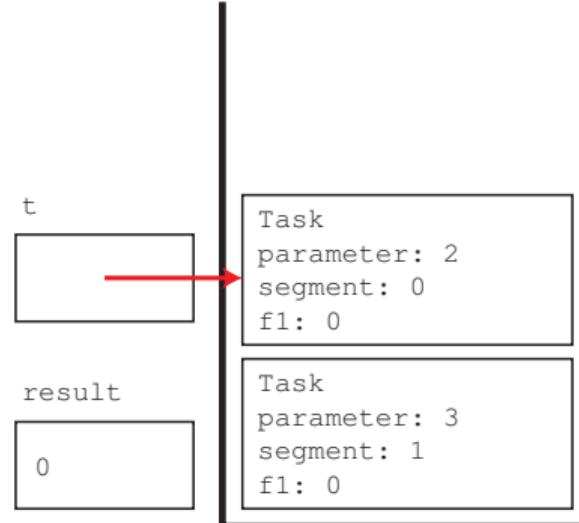
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



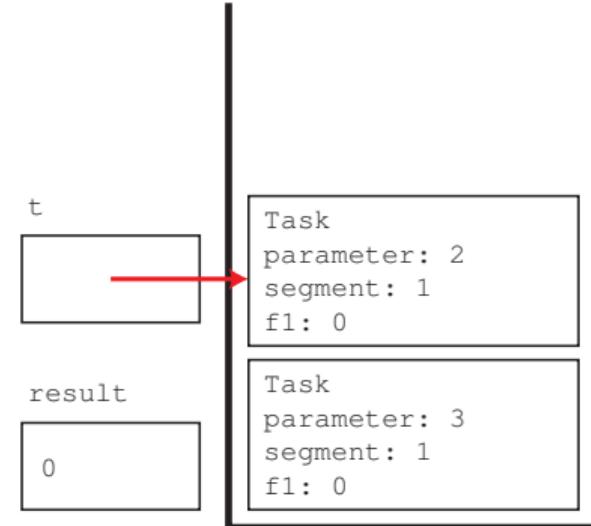
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



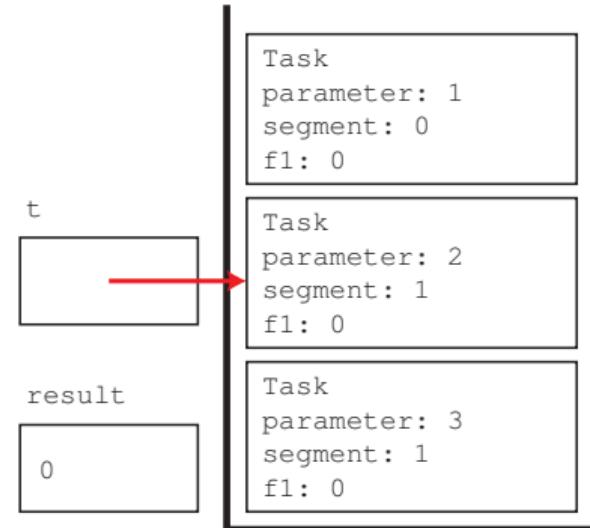
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



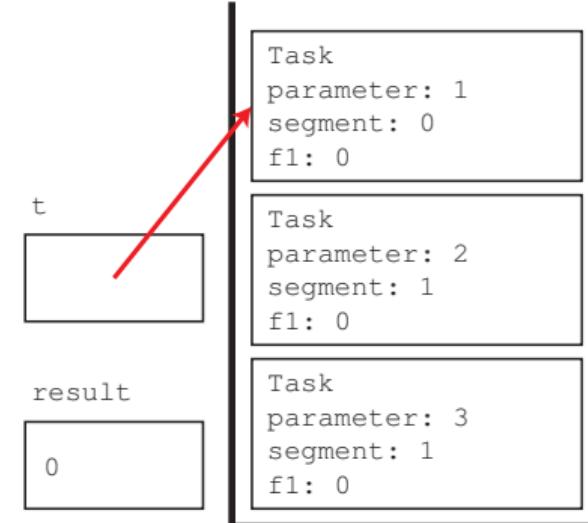
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



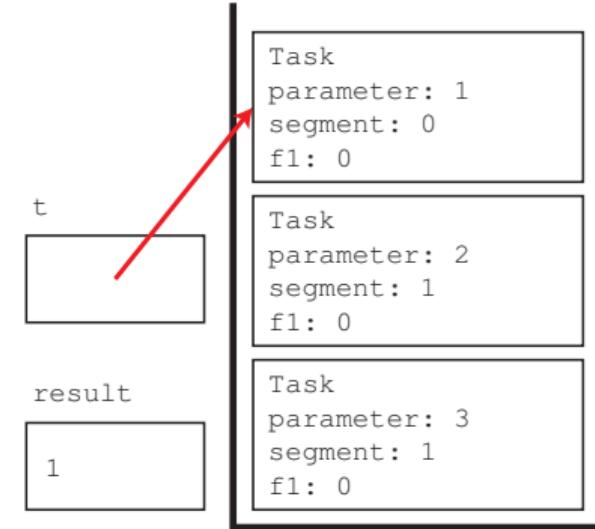
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



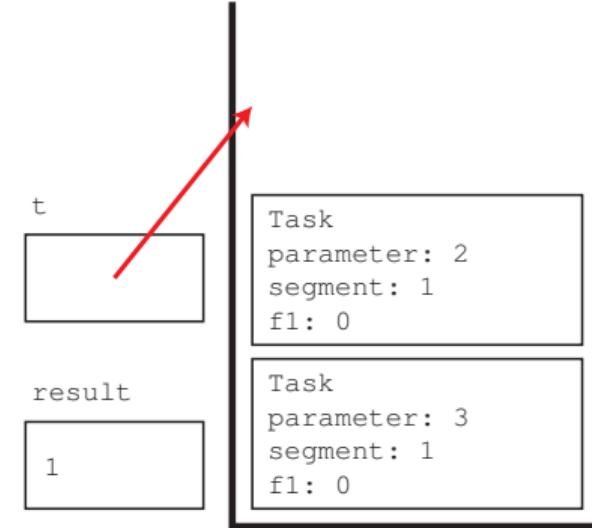
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



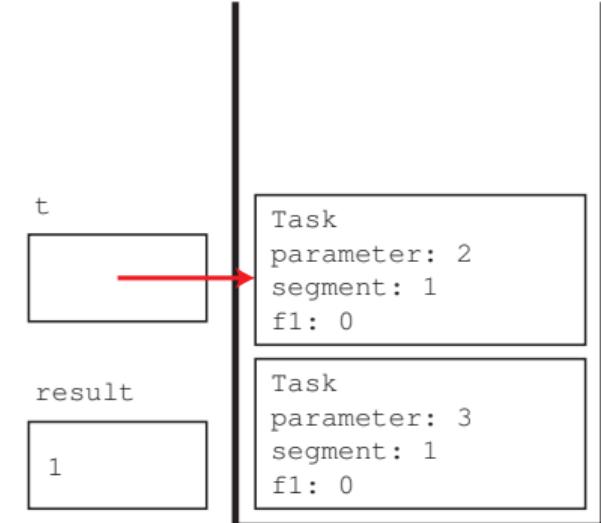
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



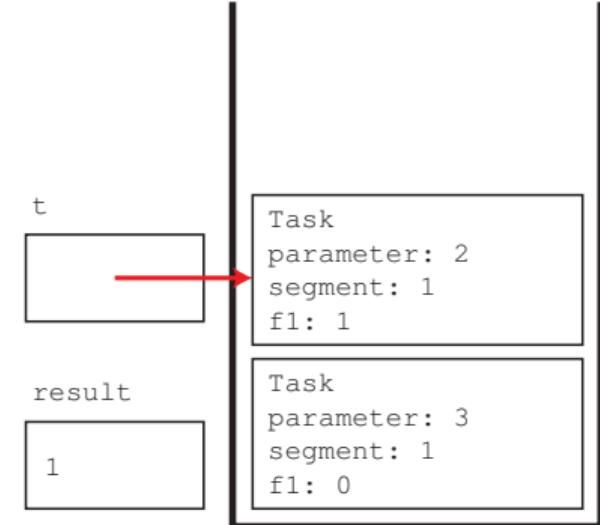
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



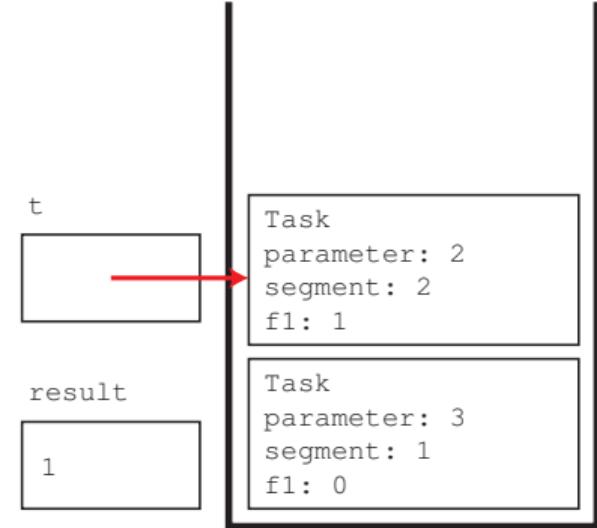
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



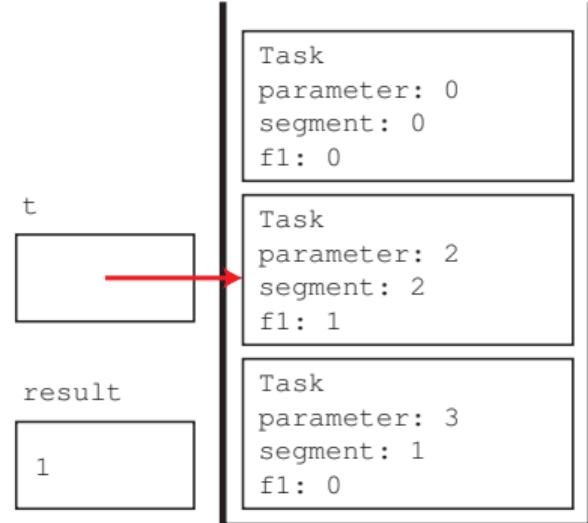
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



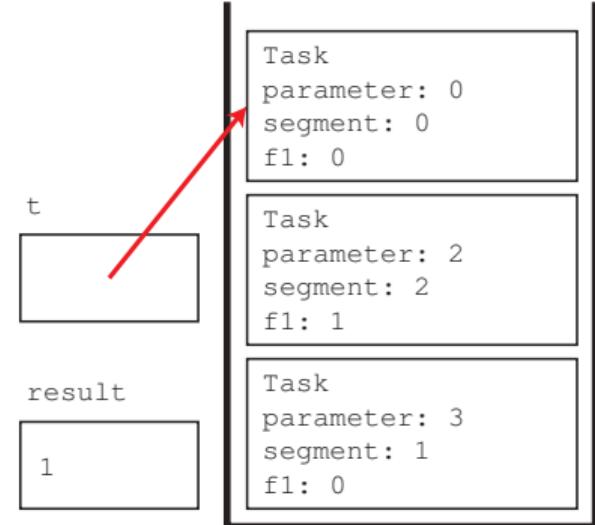
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



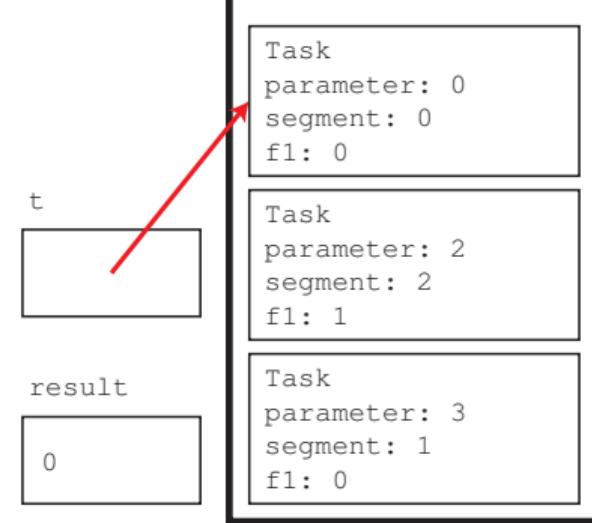
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



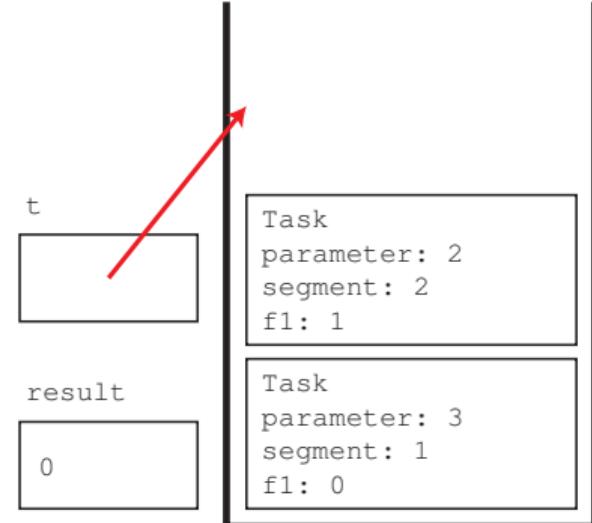
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



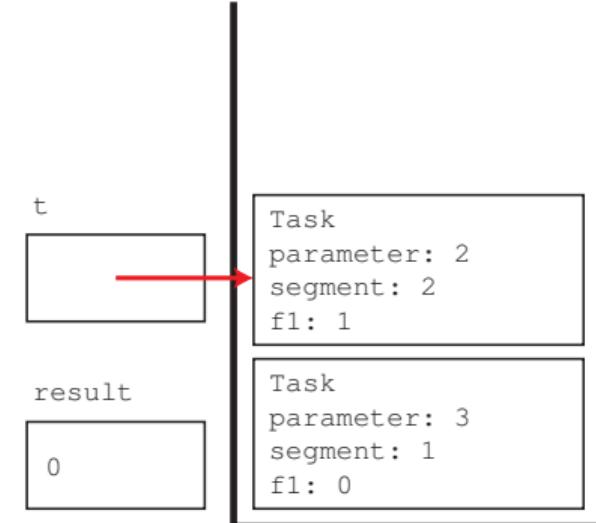
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



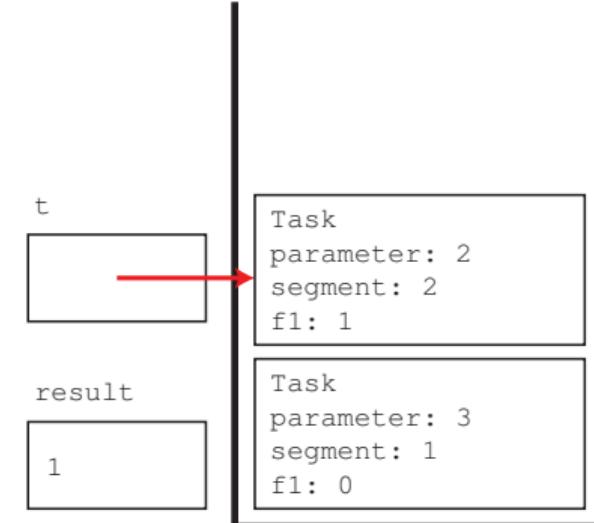
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                result = t.parameter;  
                stack.RemoveLast();  
                break;  
            }  
            t.segment = 1;  
            stack.Add(new Task(t.parameter-1));  
            break;  
            case 1: t.f1 = result;  
            t.segment = 2;  
            stack.Add(new Task(t.parameter-2));  
            break;  
            case 2: result = t.f1 + result;  
            stack.RemoveLast();  
            break;  
        }  
    }  
    return result;  
}
```



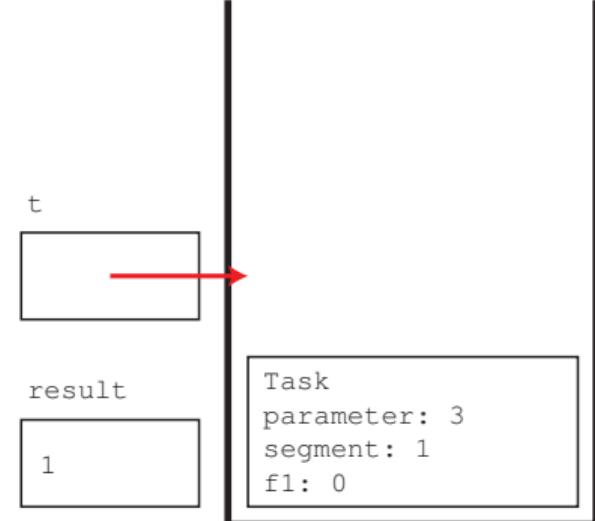
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



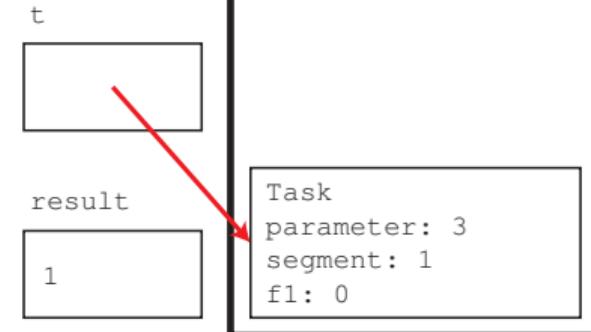
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



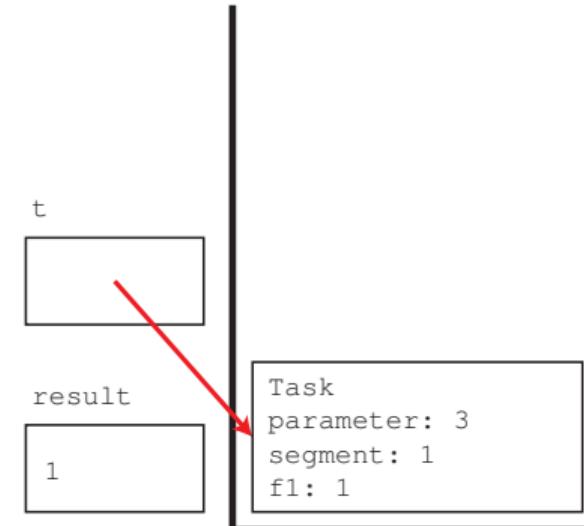
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



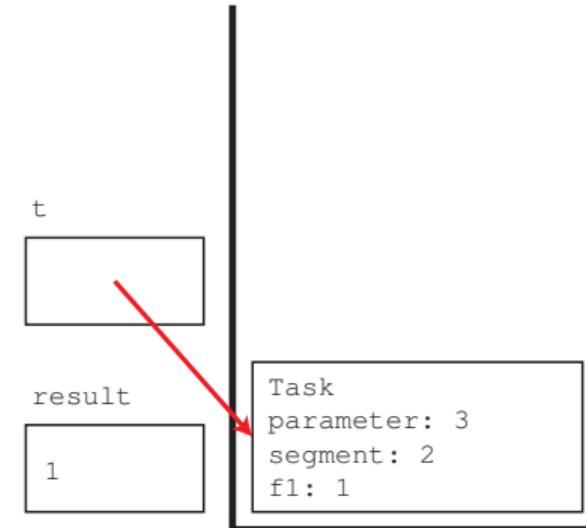
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



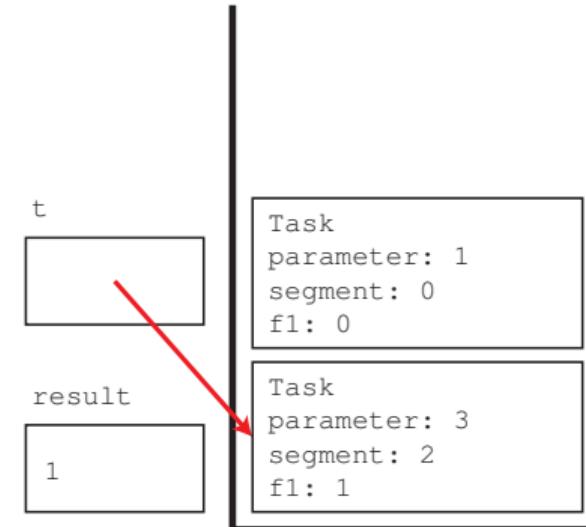
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



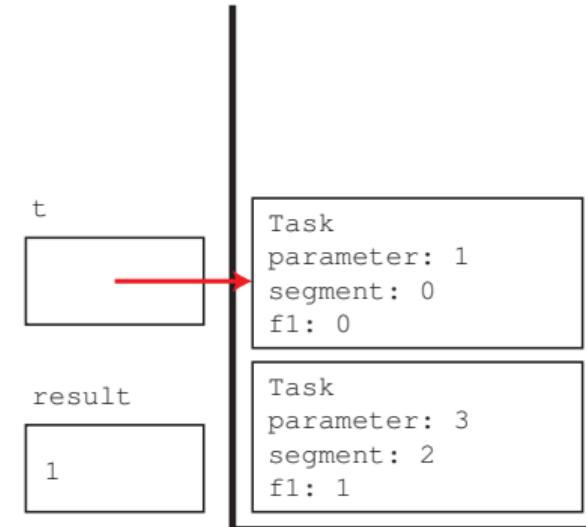
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



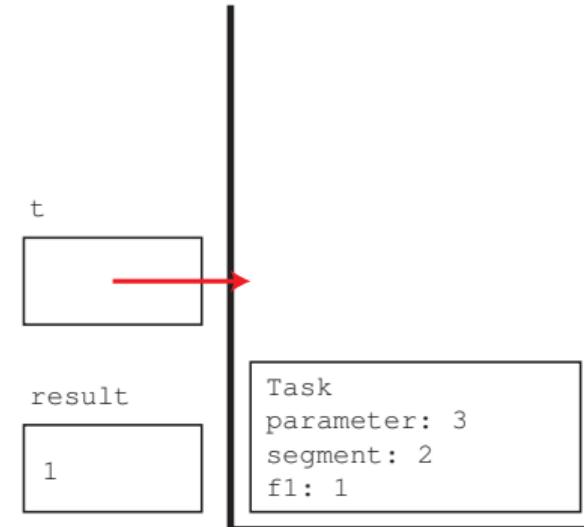
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



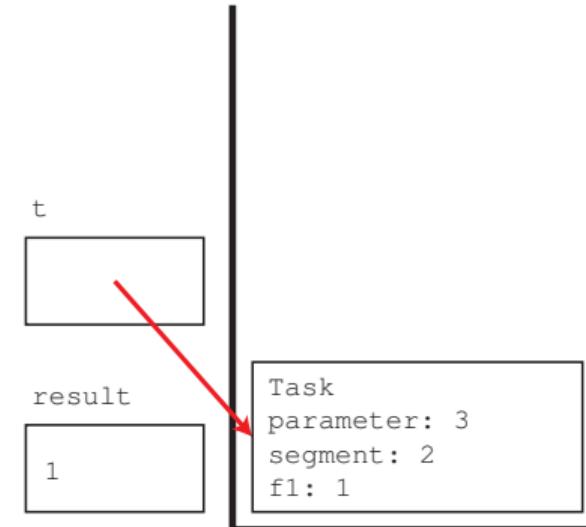
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



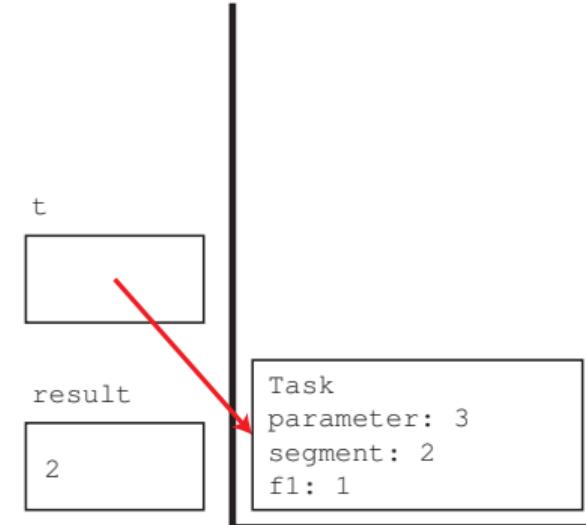
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



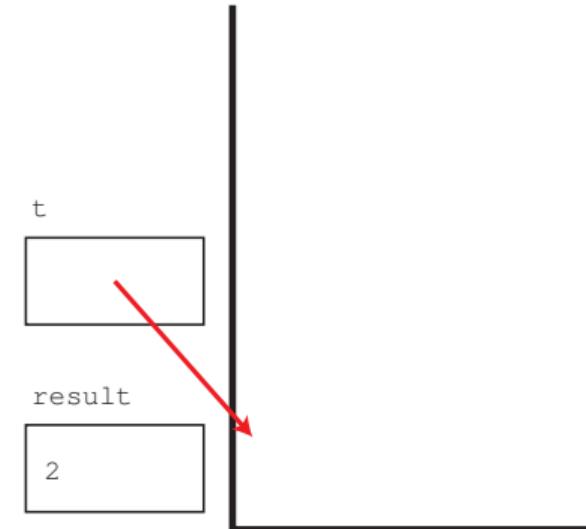
Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment){  
            case 0: if (t.parameter < 2){  
                    result = t.parameter;  
                    stack.RemoveLast();  
                    break;  
                }  
                t.segment = 1;  
                stack.Add(new Task(t.parameter-1));  
                break;  
            case 1: t.f1 = result;  
                    t.segment = 2;  
                    stack.Add(new Task(t.parameter-2));  
                    break;  
            case 2: result = t.f1 + result;  
                    stack.RemoveLast();  
                    break;  
        }  
    }  
    return result;  
}
```



Nerekurzivní verze

```
int FibonacciNonRecursive(int n){  
    IStack<Task> stack = new StackArray<Task>();  
    stack.Add(new Task(n));  
    int result = 0;  
    while(stack.IsEmpty() == false) {  
        Task t = stack.Get();  
        switch (t.segment) {  
            case 0: if (t.parameter < 2) {  
                result = t.parameter;  
                stack.RemoveLast();  
                break;  
            }  
            t.segment = 1;  
            stack.Add(new Task(t.parameter - 1));  
            break;  
            case 1: t.f1 = result;  
            t.segment = 2;  
            stack.Add(new Task(t.parameter - 2));  
            break;  
            case 2: result = t.f1 + result;  
            stack.RemoveLast();  
            break;  
        }  
    }  
    return result;  
}
```



Výpočetní složitost

Kolikrát se bude volat funkce fibonacci?

$n = 2 : 3 \times (n = 2, n = 1, n = 0)$

Výpočetní složitost

Kolikrát se bude volat funkce fibonacci?

$$n = 2 : 3 \times (n = 2, n = 1, n = 0)$$

$$n = 3 : 5 \times (n = 1, n = 2(3 \times), n = 3)$$

Výpočetní složitost

Kolikrát se bude volat funkce fibonacci?

$n = 2 : 3 \times (n = 2, n = 1, n = 0)$

$n = 3 : 5 \times (n = 1, n = 2(3 \times), n = 3)$

$n = 4 : 9 \times (n = 3(5 \times), n = 2(3 \times), n = 4)$

Výpočetní složitost

Kolikrát se bude volat funkce fibonacci?

$n = 2 : 3 \times (n = 2, n = 1, n = 0)$

$n = 3 : 5 \times (n = 1, n = 2(3 \times), n = 3)$

$n = 4 : 9 \times (n = 3(5 \times), n = 2(3 \times), n = 4)$

$n = 5 : 15 \times (n = 4(9 \times), n = 3(5 \times), n = 5)$

$n = 6 : 25 \times (n = 5(15 \times), n = 4(9 \times), n = 6)$

$n = 7 : 41 \times (n = 6(25 \times), n = 5(15 \times), n = 7)$

- volání tvoří stromovou strukturu
- počet volání se dá sám vyjádřit Fibonacciho číslů +1, tj. $F_n = F_{n-1} + F_{n-2} + 1$

Výpočetní složitost

Kolikrát se bude volat funkce fibonacci?

$n = 2 : 3 \times (n = 2, n = 1, n = 0)$

$n = 3 : 5 \times (n = 1, n = 2(3 \times), n = 3)$

$n = 4 : 9 \times (n = 3(5 \times), n = 2(3 \times), n = 4)$

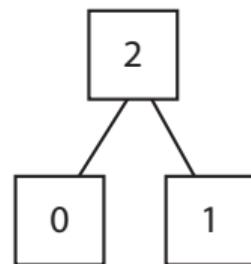
$n = 5 : 15 \times (n = 4(9 \times), n = 3(5 \times), n = 5)$

$n = 6 : 25 \times (n = 5(15 \times), n = 4(9 \times), n = 6)$

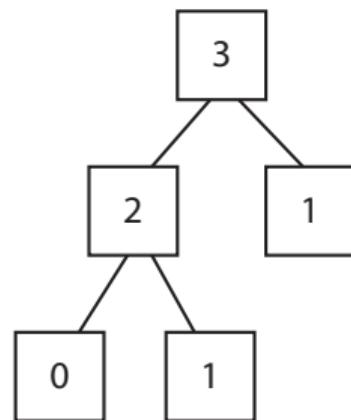
$n = 7 : 41 \times (n = 6(25 \times), n = 5(15 \times), n = 7)$

- volání tvoří stromovou strukturu
- počet volání se dá sám vyjádřit Fibonacciho čísla +1, tj. $F_n = F_{n-1} + F_{n-2} + 1$
- počet volání roste exponenciálně ($\Omega(c^n)$, $c > 1$)

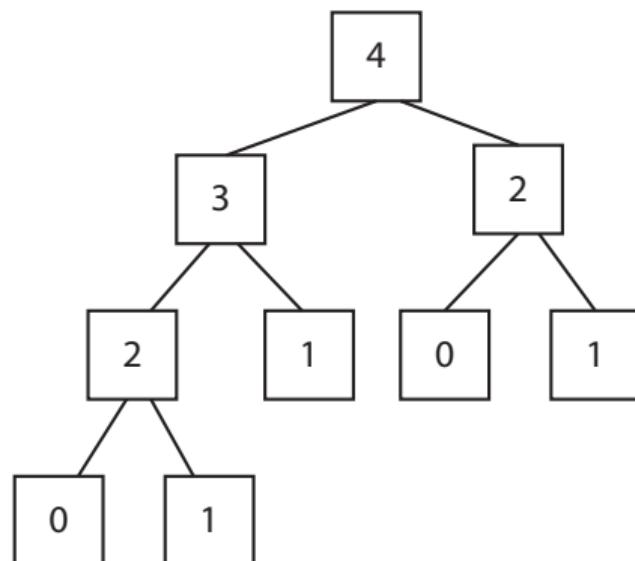
Volání funkce



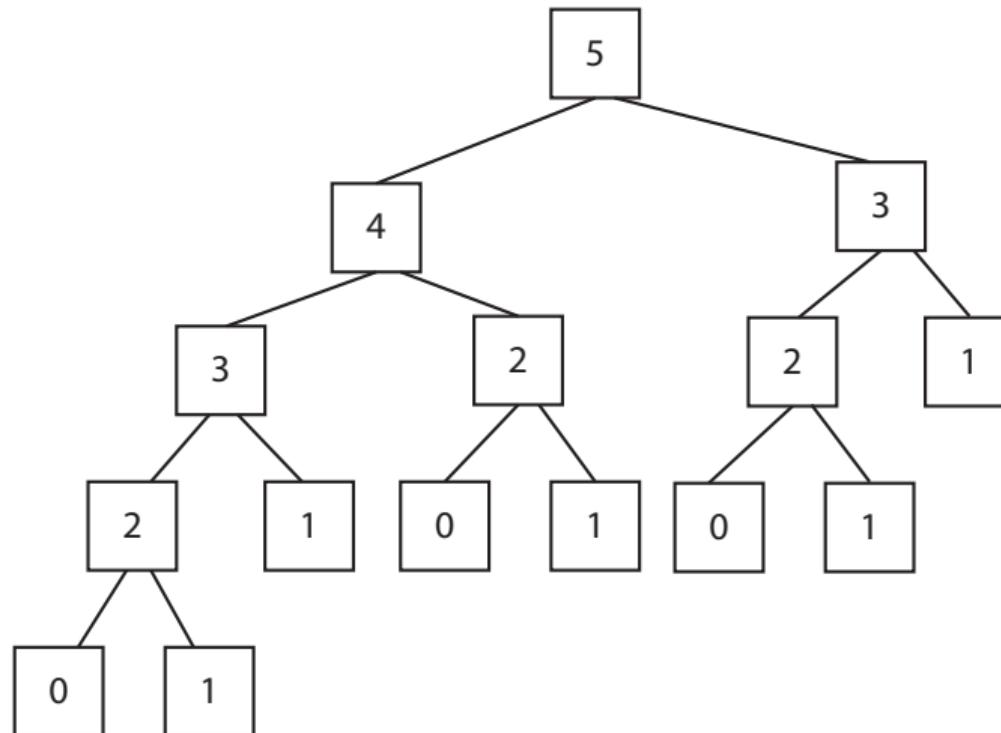
Volání funkce



Volání funkce



Volání funkce



Výpočetní složitost

Srovnejte s následující implementací:

```
int Fibonacci(int n) {  
    if (n<2)  
        return n;  
    int fnm2 = 0;  
    int fnm1 = 1;  
    int fn = 1;  
    for (int i = 2;i<n;i++) {  
        fnm2=fnm1;  
        fnm1=fn;  
        fn=fnm1+fnm2;  
    }  
    return fn;  
}
```

Výpočetní složitost

Srovnejte s následující implementací:

```
int Fibonacci(int n) {
    if (n<2)
        return n;
    int fnm2 = 0;
    int fnm1 = 1;
    int fn = 1;
    for (int i = 2;i<n;i++) {
        fnm2=fnm1;
        fnm1=fn;
        fn=fnm1+fnm2;
    }
    return fn;
}
```

smyčka se bude opakovat $(n - 2) \times$, algoritmus tedy patří do $\mathcal{O}(n)$, tj. nesrovnatelně efektivnější

- každý rekurzivní program je možné přepsat bez použití rekurze
 - mechanický postup
 - je to pracné
 - výsledný kód není moc přehledný
 - potřebujeme implementaci ADT Zásobník

- každý rekurzivní program je možné přepsat bez použití rekurze
 - mechanický postup
 - je to pracné
 - výsledný kód není moc přehledný
 - potřebujeme implementaci ADT Zásobník
- přepsání bez rekurze ale ještě **nezaručuje efektivitu!**

- každý rekurzivní program je možné přepsat bez použití rekurze
 - mechanický postup
 - je to pracné
 - výsledný kód není moc přehledný
 - potřebujeme implementaci ADT Zásobník
- přepsání bez rekurze ale ještě **nezaručuje efektivitu!**
- přepsání **zachovává třídu složitosti**

- každý rekurzivní program je možné přepsat bez použití rekurze
 - mechanický postup
 - je to pracné
 - výsledný kód není moc přehledný
 - potřebujeme implementaci ADT Zásobník
- přepsání bez rekurze ale ještě **nezaručuje efektivitu!**
- přepsání **zachovává třídu složitosti**
- pro nalezení efektivního algoritmu (v lepší třídě složitosti) neexistuje mechanický postup