NPRG041 – C++

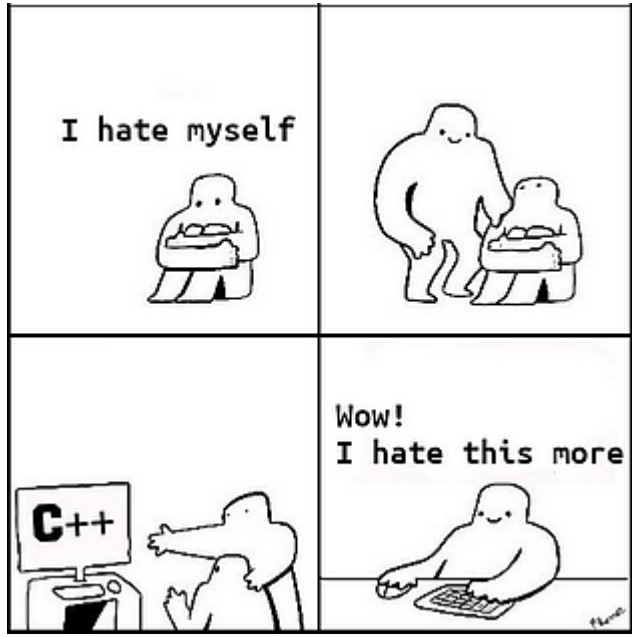
cvičení – Jiří Klepl

mattermost: ulita/2526ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

klepl@d3s.mff.cuni.cz

Agenda

- Modularita, objekty
- Vlastnictní a RAII
- Pointery a reference
  - Raw pointery
  - `std::unique_ptr`
  - `std::shared_ptr`
- `std::move`
- `std::swap`



Správa paměti: RAII

- Život C++ objektu začíná konstruktorem a končí destruktorem
  - Konstruktor: může alokovat paměť, otevřít soubor, atd.
  - Destruktor: dealokuje paměť, zavírá soubor, atd.
- Objekt „umírá“ při
  - Konec scopu, ve kterém byl definován
  - Smrt rodičovského objektu
    - std::vector<T> zabíjí svoje objekty
      - ⚠️ std::span<T> ne (je jen reference na pole)

```
struct Object {
    std::string name_;
    Object(string name) : name_(name) {
        cout << "Object " << name_ << " created" << endl;
    }
    ~Object() {
        cout << "Object " << name_ << " destroyed" << endl;
    }
};

int main() {
    Object a("a");
    Object b("b");
    {
        Object c("c");
    } // c is destroyed here
    Object d("d");
}
// destroyed in reverse order:
//   d, b, a
```

Raw (observer) pointery

- Definice: **T* ptr = &object**
  - Operátory **&** (získání adresy) a ***** (dereference) jsou opaky

☠️ Nepíšeme **ptr = new Object**

☠️ Raw pointery by nikdy neměly vlastnit objekty, pouze observovat

⚠️ Pointerová aritmetika (spíše nepoužívat)

```
std::vector<int> vec{0, 1, 2, 3, 4};
int* beg = &vec[0];
int* end = &vec[vec.size() - 1];
for (; beg < end; ++beg, --end) {
    std::swap(*beg, *end);
}
for (int i : vec) {
    std::println("{}", i);
}
```

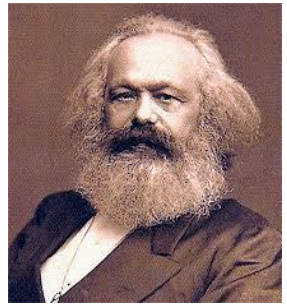
Funkce na prohození objektů

Smart pointery

`#include <memory>`

- **`unique_ptr<Object>`**
  - Reprezentuje unikátní vlastnictví objektu na haldě
    - ℹ️ nejde kopírovat
  - Automaticky dealokuje paměť
    - Pokud ten pointer umře / nastavíme ho na **nullptr**

- **`shared_ptr<Object>`**
  - Reprezentuje sdílené vlastnictví objektu na haldě
    - ℹ️ má schovaný reference-counter vlastníků a automaticky dealokuje, pokud 0
  - ⚠️ práce s ním je dražší



Smart pointery

`#include <memory>`

- **`unique_ptr<Object>`**
  - Vyrobení objektu na haldě: **pointer = std::make_unique<Object>(parametry)** ⚒
- **`shared_ptr<Object>`**
  - Vyrobení objektu na haldě: **pointer = std::make_shared<Object>(parametry)** ⚒
- Obojí se chová jako normální pointer
  - ***ptr** dereference
  - **ptr->obj_method()** volání metod
- Převedení na observer pointer **Object***: **ptr.get()**
- Předání vlastnictví objektu pomocí: **dst = std::move(src)**
  - **dst** ukazuje na objekt, **src** má hodnotu **nullptr** (nic)

std::move(objekt)

`#include <utility>`

- Tato funkce nic nedělá, jen objekt castne na **T&&** referenci
  - **T&&** typicky ukazuje na *temporary* objekty
    - Např. návratová hodnota funkce ve výrazu: **obj = func()**
  - **T&&** se jmenuje **rvalue reference** (**T&** je lvalue reference)
    - r jako right/return (návratová hodnota **func()**); l jako left (**obj**)
- Značí, že objekt má být chápán jako *temporary*
  - Funkce, co jej přijme, ho může volně „vykrást“ – převzít jeho data
- Díky **std::move** můžeme zavolat **T&&** overload funkce namísto **T&**
  - **func(T& obj)** bude obj typicky kopírovat/číst nebo nějak modifikovat
  - **func(T&& obj)** převezme data z obj a obj nechá prázdný/nepoužitelný

⚠️ move-ování typicky negarantuje vyprázdnění movovaného obj.

Úkol: bst (binární vyhledávací strom)

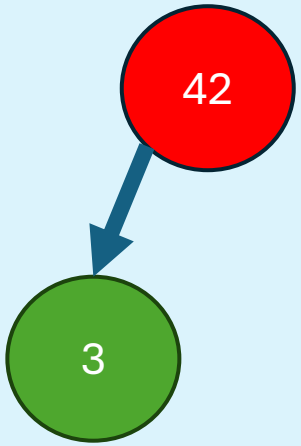
ℹ️ nejsou požadavky na vyvažování

- Zadání: dokončete implementaci zdrojového kódu bst.cpp
  1. Program čte **stdin** nebo **soubor** (pokud je zadán v cmd argumentu)
  2. Na každém NEPRÁZDNÉM řádku vstupu je příkaz
     - **insert <rest of the line after ws>** ← přidá nový node
     - **remove <rest of the line after ws>** ← odebere node (rozebráno na dalších slidech)
     - **print** ← vytiskne obsah stromu v “In-Order” pořadí indentované (hloubka × '/t') od 0
     - **clear** ← smaže obsah stromu
  3. Konec vstupu/error ukončí program
- Strom uložený pomocí **`std::unique_ptr`**
- **<rest of the line after ws>**: to, co přečte **`std::getline(input, str)`**

Remove Node – the most difficult method

- First, **find the deleted node**
  - If it already doesn't exist, we are done
- Otherwise, perform changes to the tree according to the following three cases
  1) The node has up to one child
  2) The node has two children, and its right child is also its successor
  3) The node has two children, and its right child is not its successor

  All three cases are discussed in detail on the following three slides

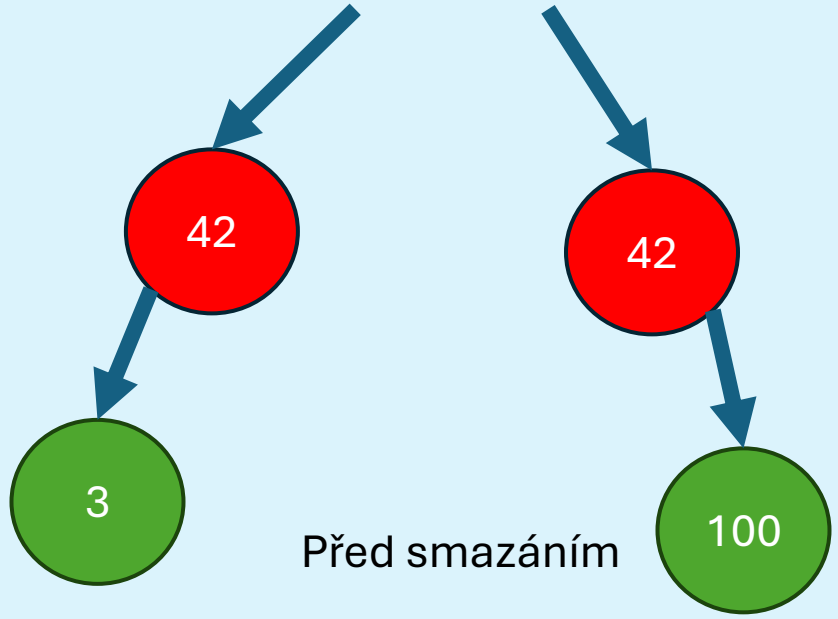


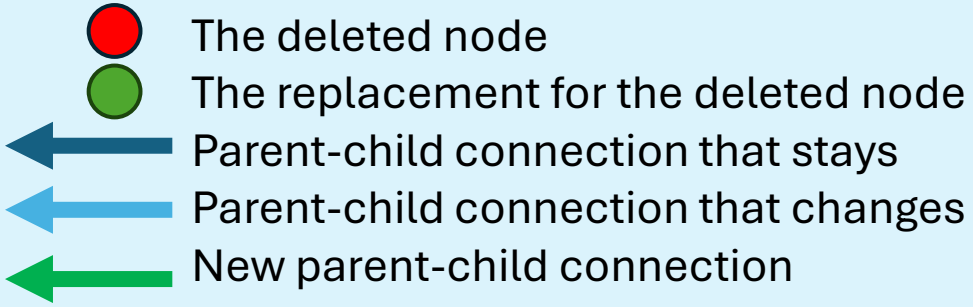
Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



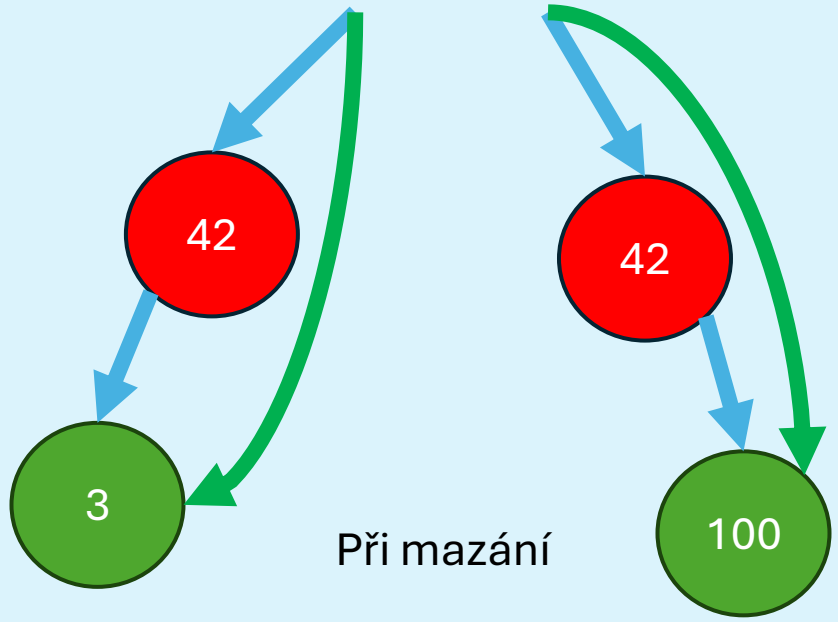


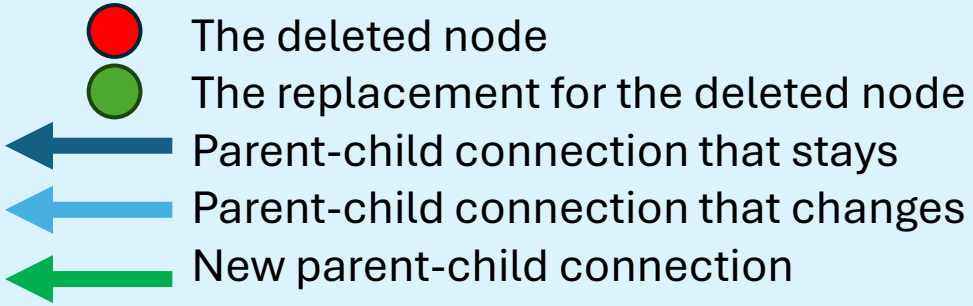
Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



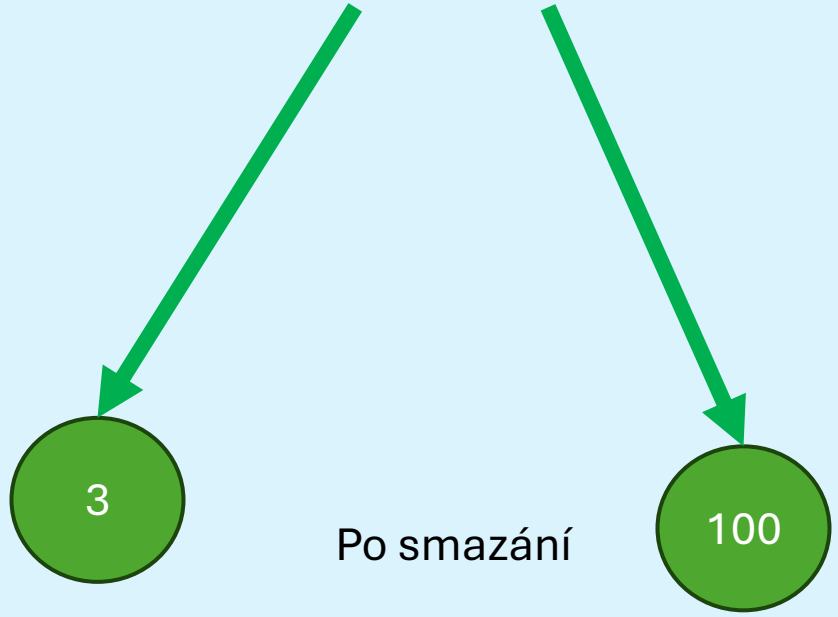


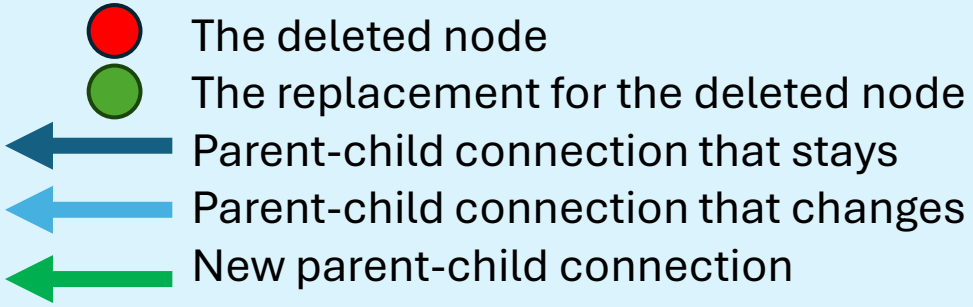
Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



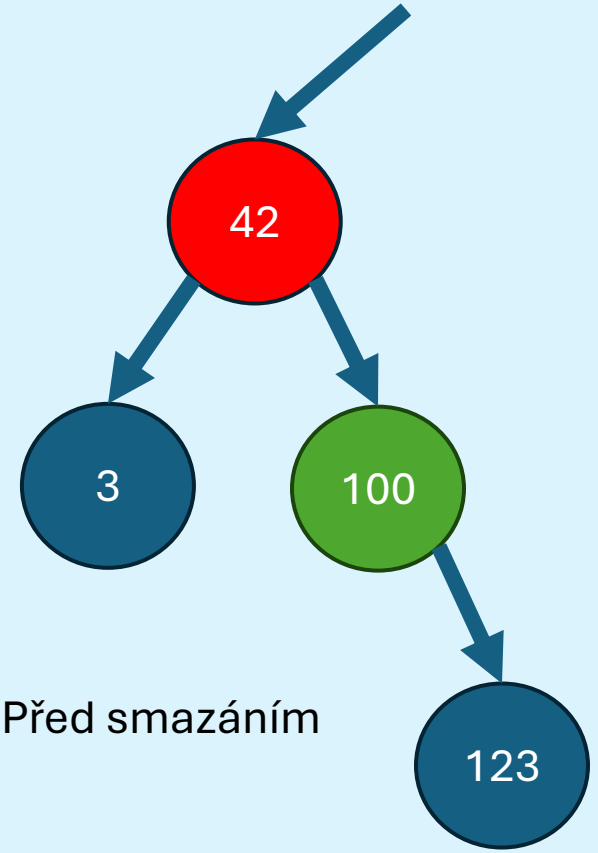


Remove Node – the second case

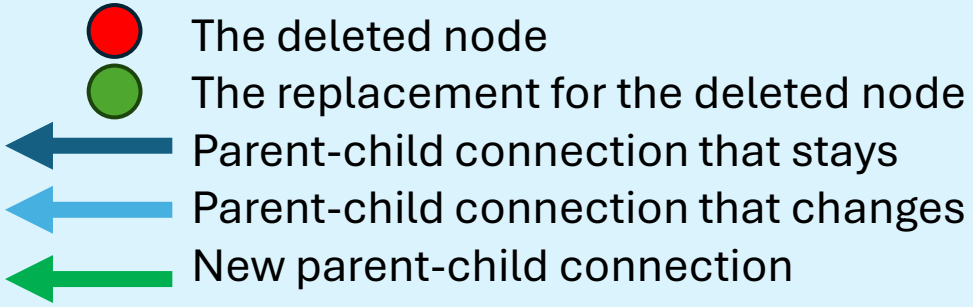
- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node's left child to the successor

➔Replace the node with the successor



Před smazáním

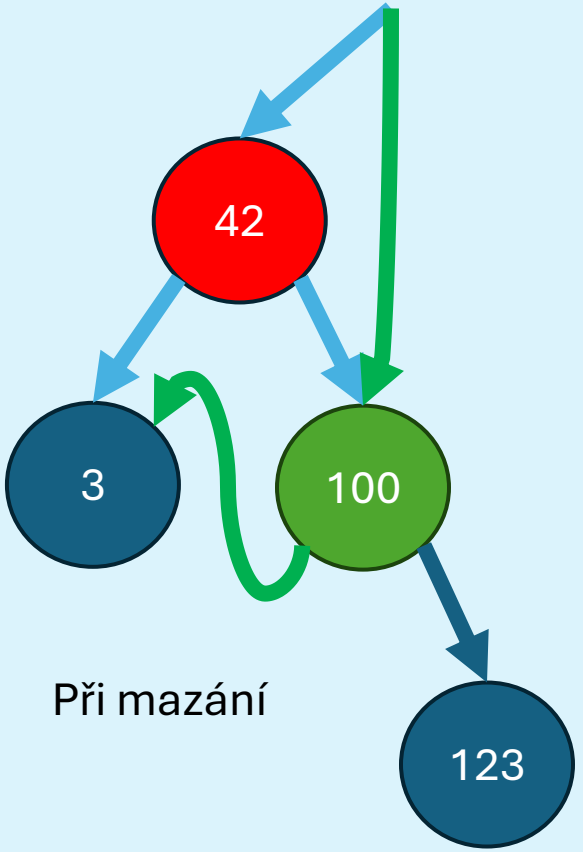


Remove Node – the second case

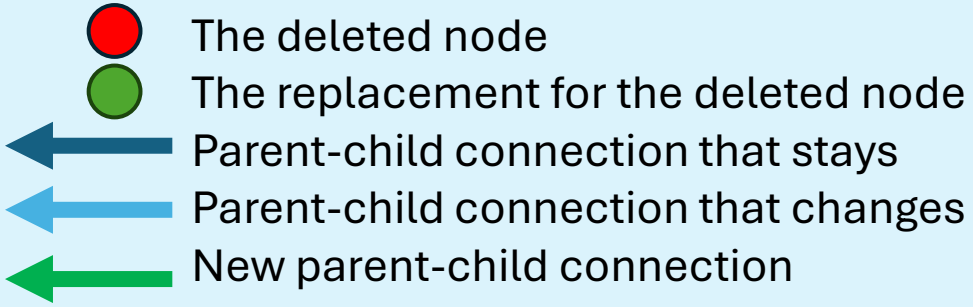
- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node's left child to the successor

➔Replace the node with the successor



Při mazání

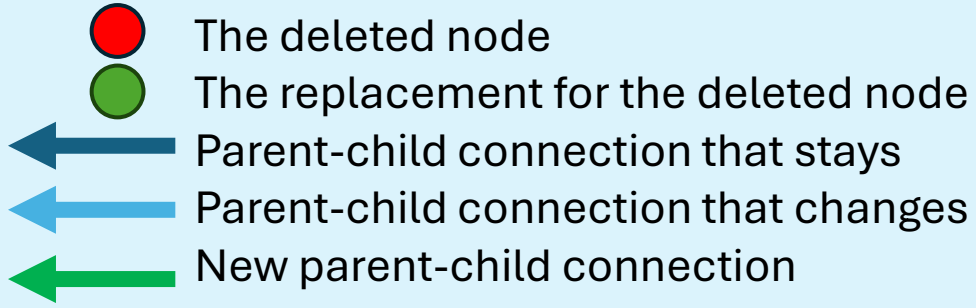


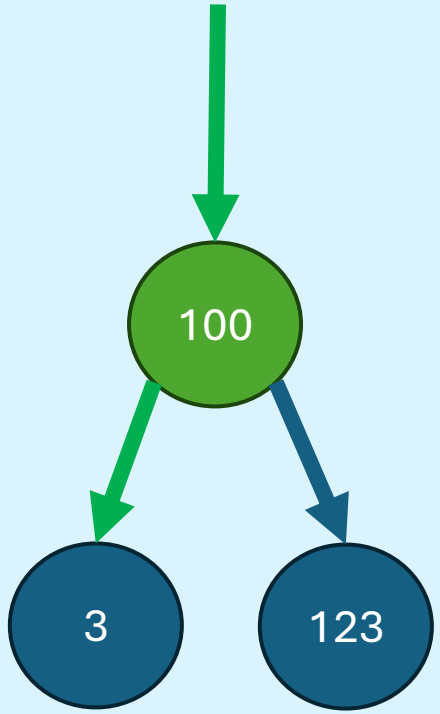
Remove Node – the second case

- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node's left child to the successor

➔Replace the node with the successor





Po smazání

Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

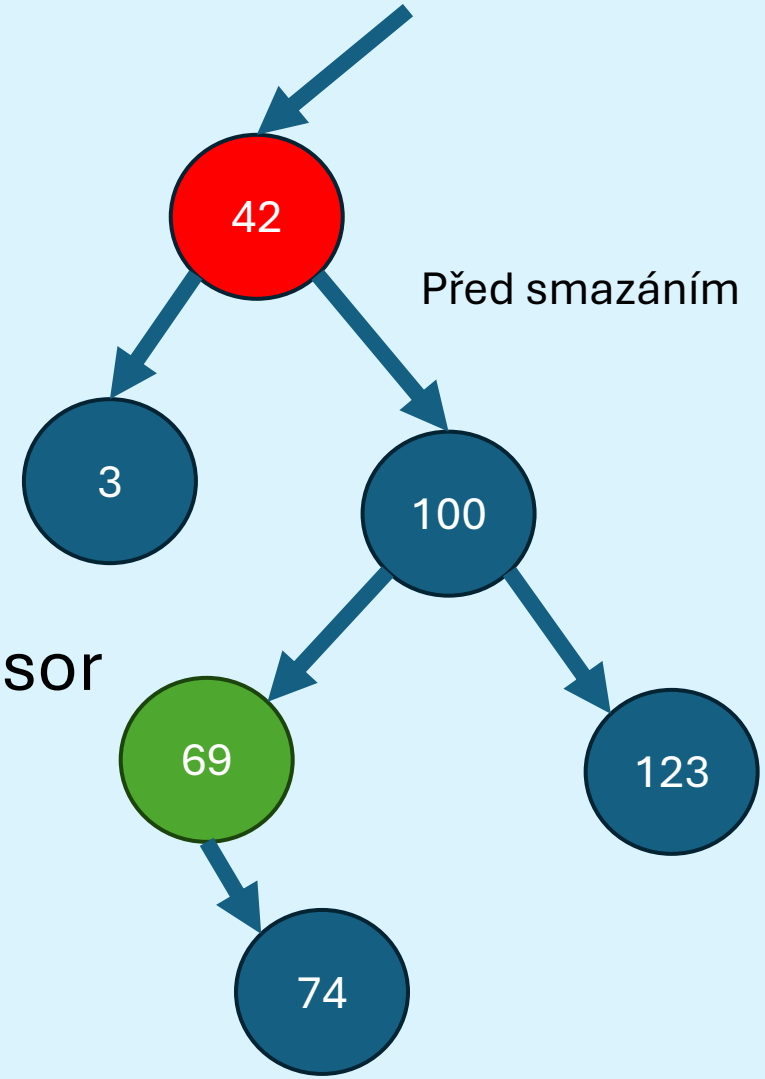
One solution is:

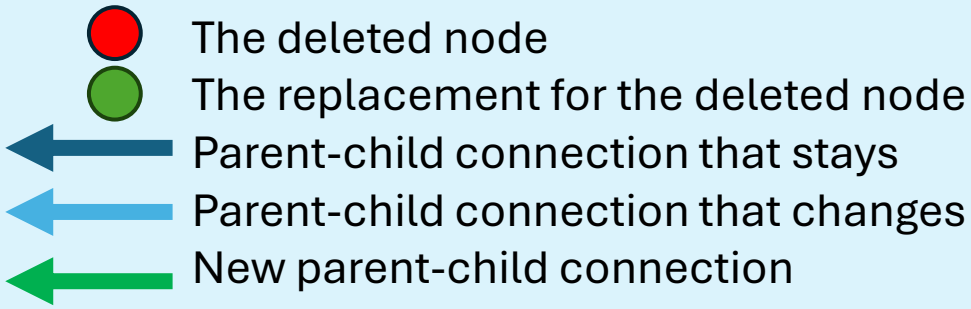
➔Move the successor to a new unique_ptr

➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children
(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor





Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

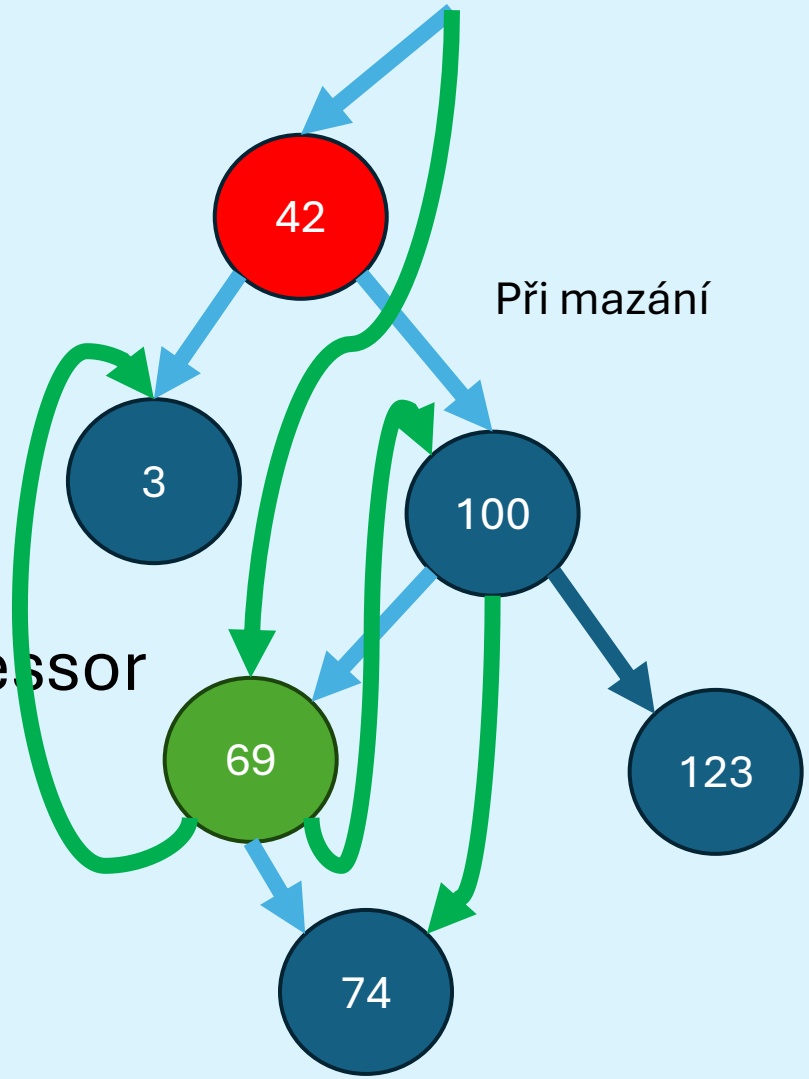
One solution is:

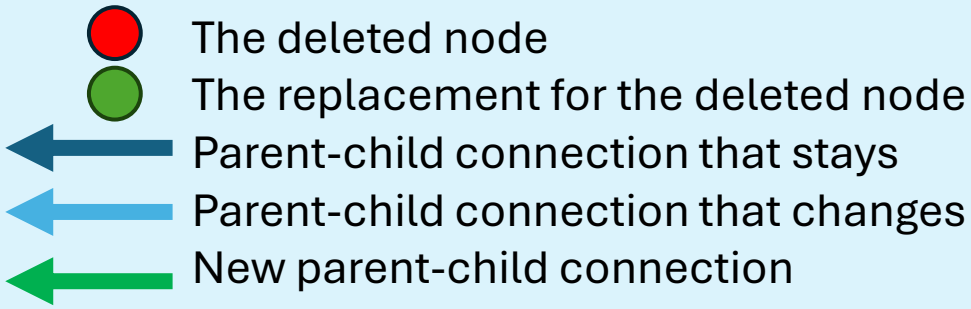
➔Move the successor to a new unique_ptr

➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children
(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor





Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

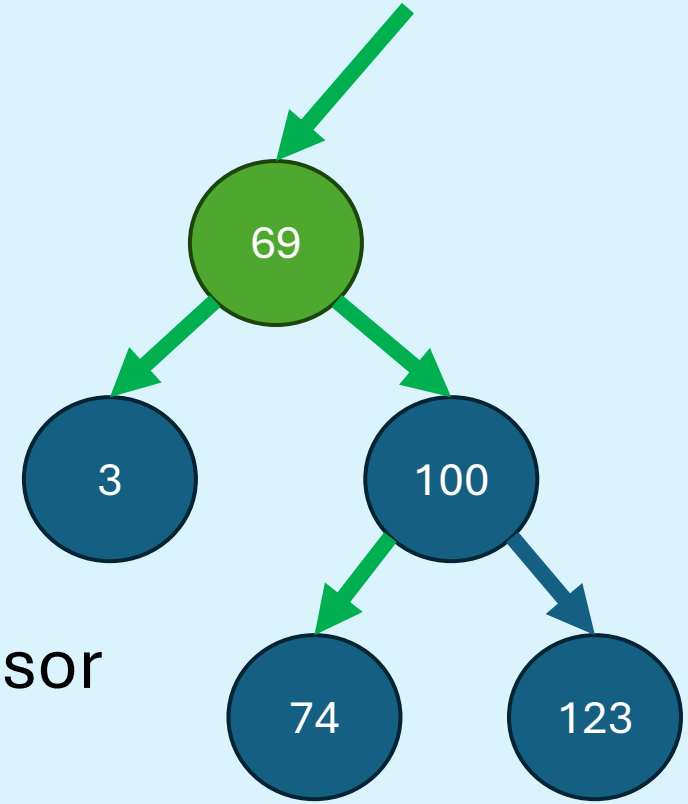
One solution is:

➔Move the successor to a new unique_ptr

➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children
(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor



Po smazání