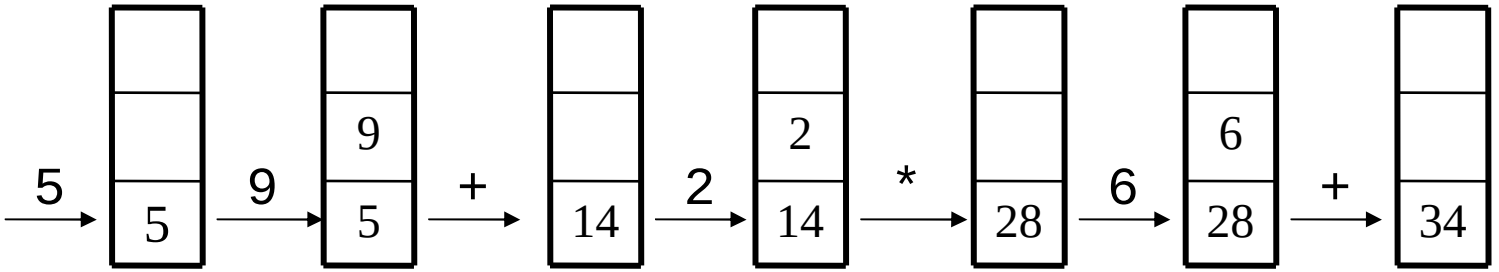
ADT zásobník - příklad

- Prakticky aplikovatelný příklad využití ADT zásobník je vyhodnocování aritmetických výrazů v Postfixové notaci:
- Pro zápis aritmetických výrazů se používají tři druhy zápisu:
  - infix: (5+9)*2+6
  - prefix: +*+5926
  - postfix: 59+2*6+
- Prefixová notace se používá od roku 1920. Jejím průkopníkem byl polský matematik Jan Lukasiewicz. Postfixu se říká také „obrácená polská notace“.
- Výhoda postfixu i prefixu spočívá v tom, že nevyžadují použití závorek. Postfix lze pak vyhodnocovat velmi jednoduše pomocí zásobníkového algoritmu. Např. výše uvedený příklad se vyhodnotí takto:

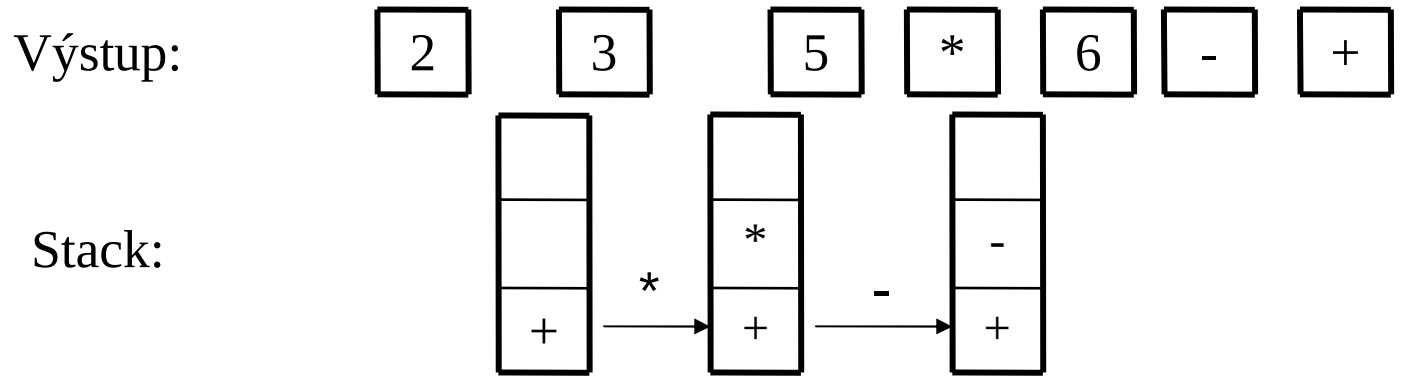


ADT zásobník - příklad

- Postfixový zápis je ale pro většinu uživatelů nestravitelný, proto algoritmus INFIX vyhodnocování výrazů musí převést infix do Postfixu a teprve ten vyhodnotit.
- Základní algoritmus pro výrazy bez závorek:

```
While not eof(vstup)
    read(znak)
        If znak je operand then write(znak)
        If znak je operator then
            while Precedence(znak) <= Top(Stack)
                    write(Pop(Stack))
            end loop
            Push(znak, operatorStack)
end while
popAndWrite(operatorStack)
```

- Příklad: 2+3*5-6



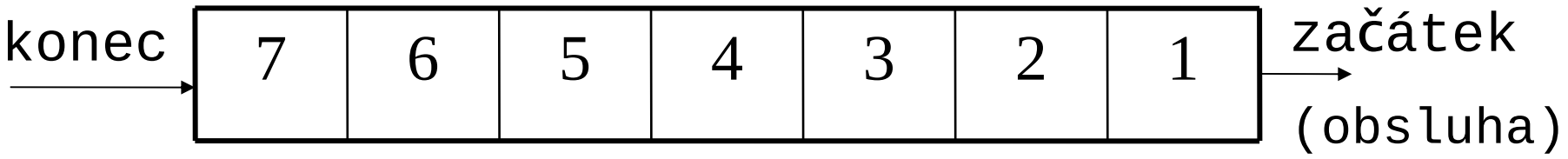
Infix2Postfix

Ošetření závorek v infix2postfix:

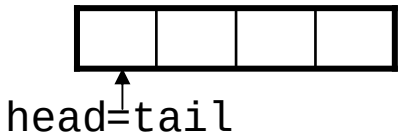
- Operandy považuj za operátor s nejvyšší prioritou. Závorky považuje za operátor s nejnižší prioritou
- Operátory si ukládejte na zásobník takto:
  - Pokud je na vstupu levá závorka, vždy ji vložte na zásobník
  - pokud je vstup = pravá závorka, vyprázdni a vypiš všechny operátory na zásobníku až po levou závorku
  - pokud je na vstupu operátor a jeho priorita < priorita znaku na zásobníku, vyprázdni (a vypiš) zásobník až po levou závorku. Pokud je priorita operátoru > znak na zásobníku, ulož operátor na zásobník.
  - Na začátku si na zásobník vložte startovací levou závorku

ADT Fronta

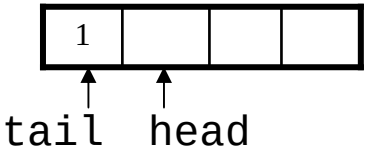
- Abstrakce fronty je lidskému životu nejbližší – v nějaké frontě stojíme každý den alespoň jednou
- Fronta se v programech používá všude tam, kde je rychlost příchodu požadavků je nárazově nebo i trvale větší než rychlost jejich zpracování
- Fronta má většinou konstantní velikost – nejčastější implementace = dynamicky alokované pole.
- Pokud potřebujeme dynamicky rostoucí frontu, lze ji implementovat lineárním seznamem
- Fronta může implementovat různé politiky průchodu prvků – nejčastější je FIFO, ale často nacházíme také FIFO s prioritami atd..
- U fronty stačí 2 operace:
  - enque(…) – zařadí na konec fronty nový prvek
  - Deque(…) – vyřadí prvek ze začátku fronty
  - Doplňující operace: size(…) – vrací počet prvků ve frontě, empty(..) – true pokud je fronta prázdná…



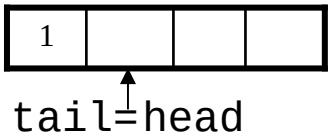
Implementace fronty statickým polem =kruhový buffer



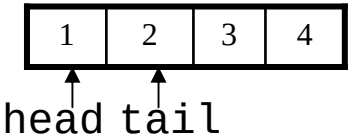
Fronta je prázdná



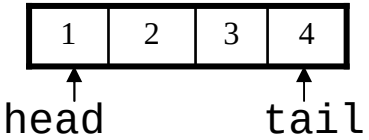
Fronta po vložení 1. prvku



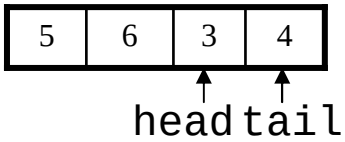
Prázdná fronta po vyzvednutí 1. prvku



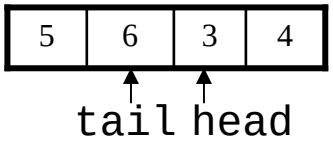
Plná fronta po vložení dalších 3 prvků



Fronta po vyzvednutí 2 prvků



Plná fronta po vložení 2 prvků



Fronta po vyzvednutí 2 prvků

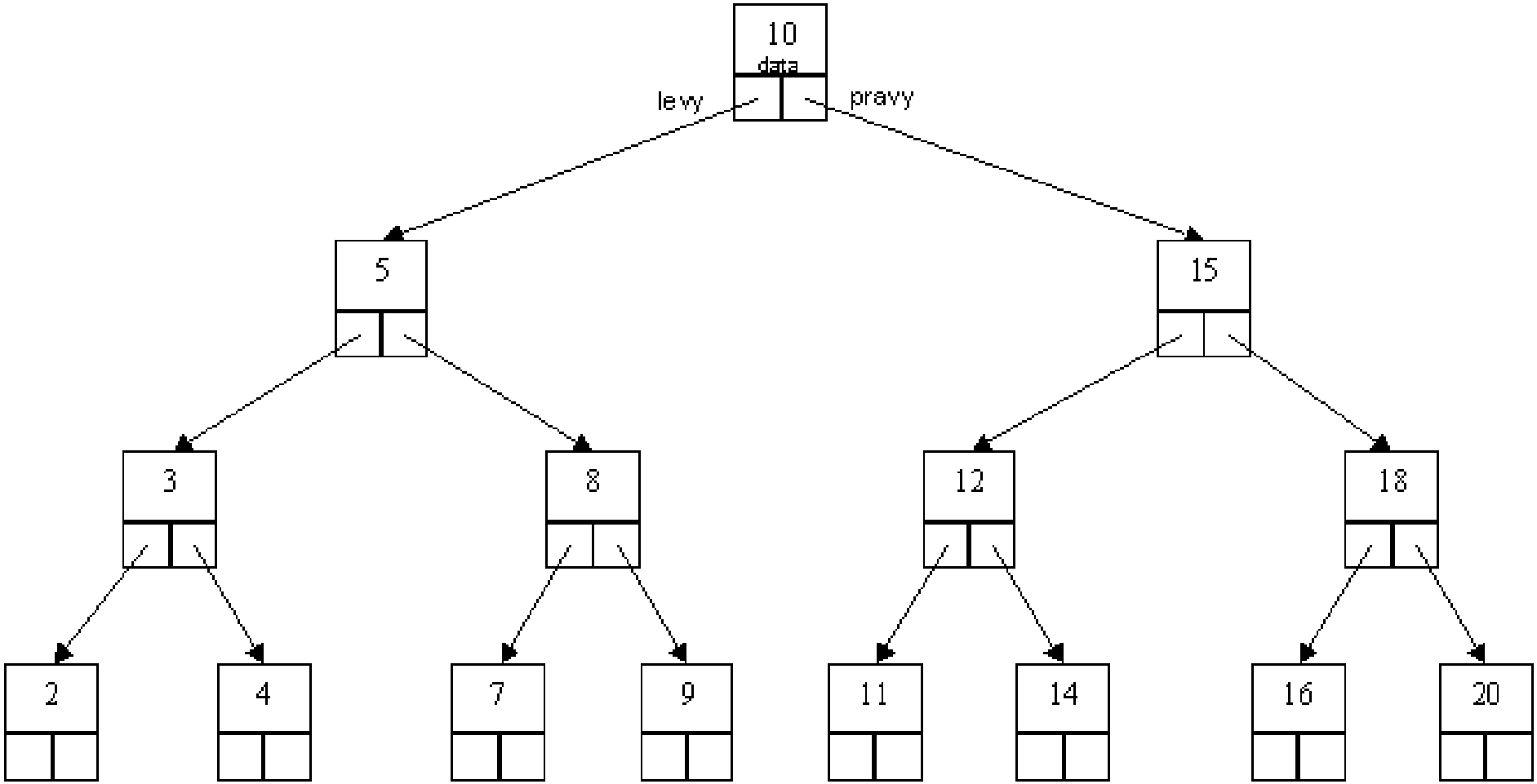
Implementace fronty lineárním seznam

- Enqueue ~ PostInsert(..); Succ();
- Dequeue ~ CopyFirst(..); DeleteFirst(..);

ADT Binární strom

- Binární strom je dynamická datová struktura, která umožňuje ukládat data zároveň se jejich setříděním tak, že operace vložení i vyhledávání trvá (v ideálním případě) log2(N) kroků, kde N je počet položek uložených ve stromu. Tzn. maximálně efektivně!
- jedná se o poměrně jednoduchou strukturu, jejíž každý prvek obsahuje data a 2 ukazatele - ukazatel na levého následníka, což je opět strom ve kterém jsou uloženy prvky s menší hodnotou, než má aktuální prvek, a ukazatel na pravého následníka, což je strom, ve kterém jsou uloženy prvky s větší hodnotou než má aktuální prvek
- Protože je tato definice rekurzivní (podstrom prvku je opět strom), při práci se stromem najdou uplatnění rekurzivní techniky (viz dále).

ADT Binární strom



ADT Binární strom v Pascalu

```
Type tUkUzel=^tUzelStromu;
tUzelStromu  = record
    ukLevy, ukPravy: tUkUzel;
    Data:tData;
End;
procedure BSTInsert (var RootPtr:tUkUzel; d:tData);
begin
  if RootPtr<>Nil then
  with RootPtr^ do
    if d.key<data.key then BSTInsert(ukLevy,d)
      else if d.key>data.key then BSTInsert(ukPravy,d)
        else data:=d
  else begin
   New(RootPtr);                    {vytvoření nového uzlu}
    with RootPtr^ do
    begin
      ukLevy:=Nil;          {a jeho naplnění správnými hodnotami}
      ukPravy:=Nil;
      data:=d;
    end;
  end;
end;
```

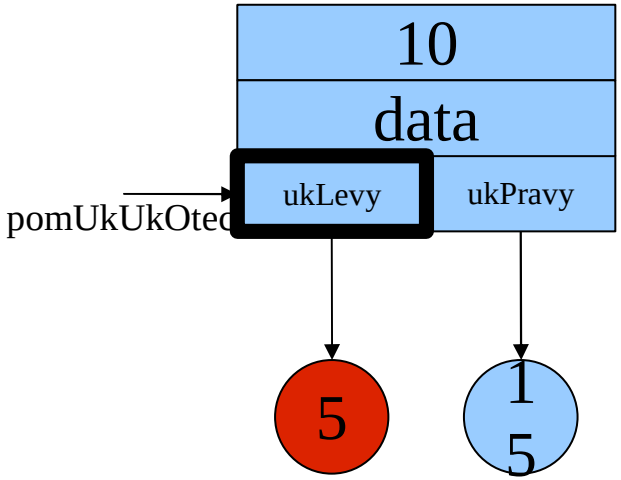
Prohledávání stromu

```
function BSTSearch (RootPtr:tukUzel; d:tdata) : tUkUzel;
begin
  if RootPtr<>nil then
  with RootPtr^ do
  begin
    if d.Key<data.Key then BSTSearch:=BSTSearch(ukLevy, d)
    else if d.Key>data.Key then BSTSearch:=BSTSearch(ukPravy,d)
         else
              BSTSearch:=RootPtr;
         end
  end
  else BSTSearch:=False;
end;
```

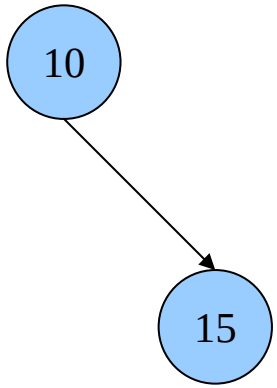
Operace delete

- Mazaný uzel nemá podstrom – můžeme jej přímo smazat
- Nezapomeňte ukazatel na něj v otcovském uzlu nastavit na nil!

Před smazáním



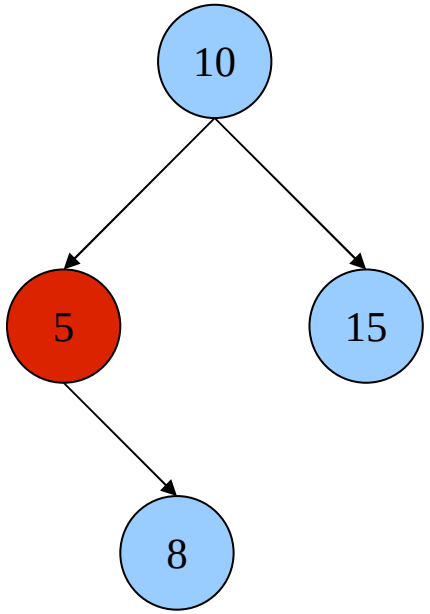
Po smazání



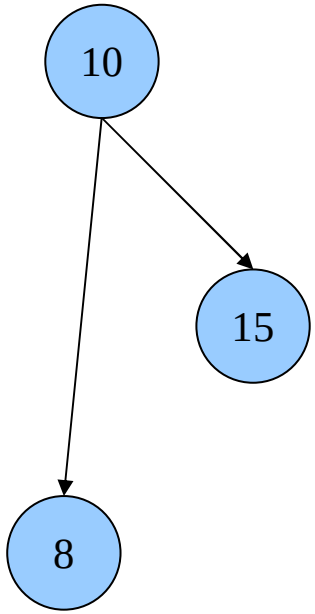
Operace delete

- Mazaný uzel má pouze 1 poduzel – „dejte vnouče dědovi“

Před smazáním

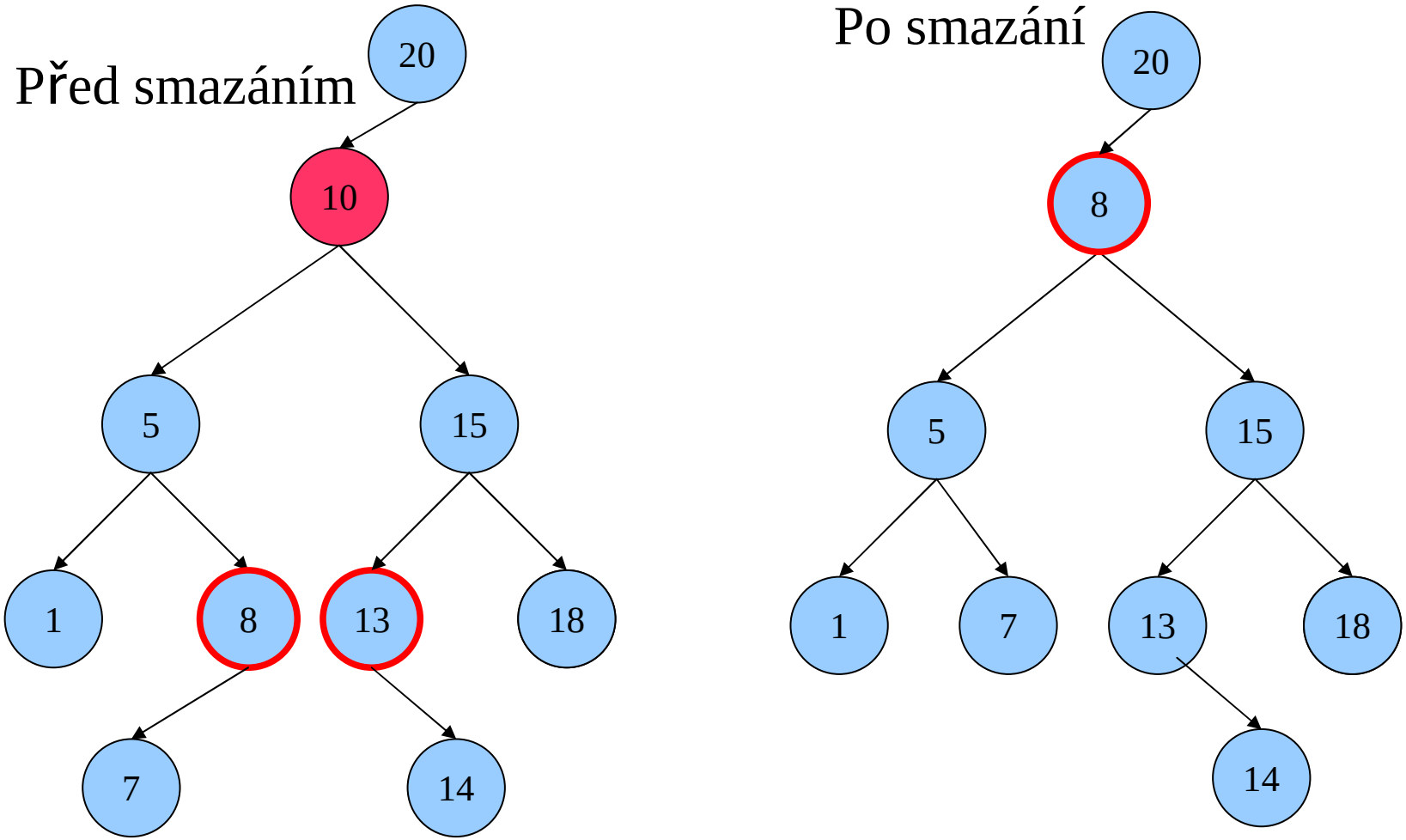


Po smazání



Operace delete

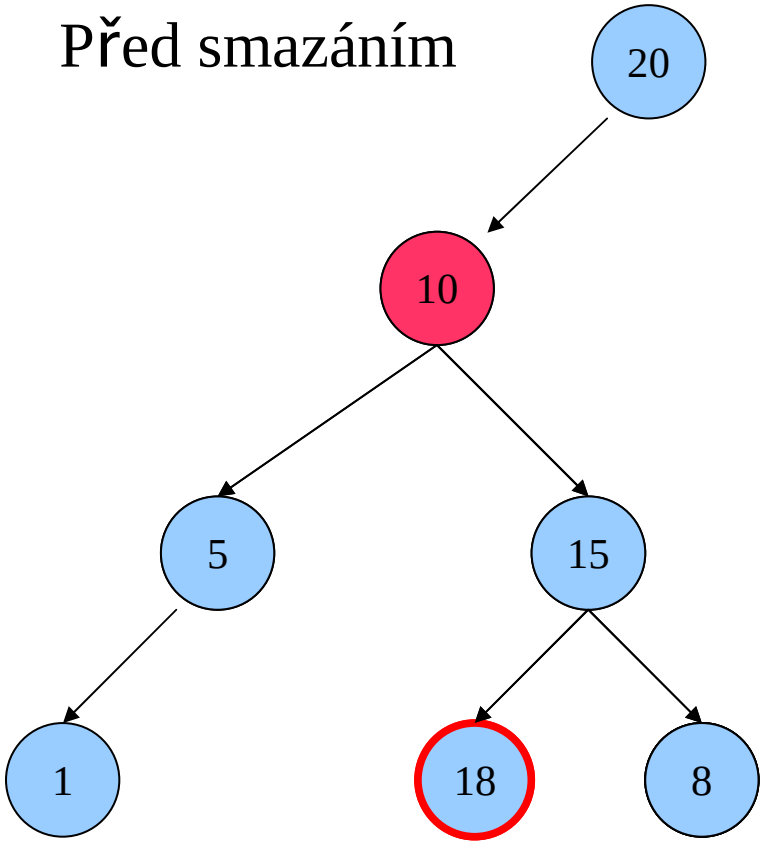
- Mazaný uzel má oba podstromy:
- Lze jej nahradit
  - nejpravějším **listem** levého podstromu
  - nejlevějším **listem** pravého podstromu



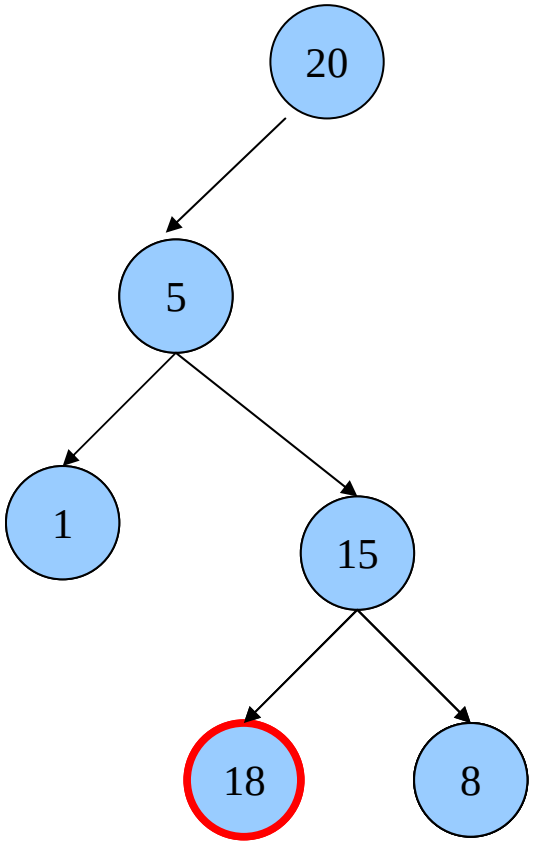
Operace delete

- Mazaný uzel má oba podstromy, ale kořen levého podstromu nemá pravého syna:
- Nahraď jej levým uzlem

Před smazáním



Po smazání



Operace Delete

```
function delete(var t:tTabulka; key:string):string;
var pomUk:tUkUzel;
  pomUkLevy, pomUkLP, otecNejpravejsiho, nejpravejsi:tUkUzel;
  pomUkUkOtec:^tUkUzel;
  nalezeno:boolean;
begin
  pomUkUkOtec:=nil;
  nejpravejsi:=nil;
  otecNejpravejsiho:=nil;
  pomUk:=t;
  nalezeno:=false;
  while (pomUk<>nil) and not nalezeno do begin
    if key=pomUk^.key then nalezeno:=true
    else if key>pomUk^.key then begin
      pomUkUkOtec:=@(pomUk^.ukPravy);
      pomUk:=pomUk^.ukPravy;
    end else begin
      pomUkUkOtec:=@(pomUk^.ukLevy);
      pomUk:=pomUk^.ukLevy;
    end;
  end;
```

```
  if nalezeno then begin
    if (pomUk^.ukLevy=nil) and  (pomUk^.ukPravy=nil)then begin
      dispose(pomUk);
      if pomUkUkOtec=nil then t:=nil
      else pomUkUkOtec^:=nil;
    end else if (pomUk^.ukLevy=nil) or (pomUk^.ukPravy=nil) then begin
        if pomUk^.ukLevy<>nil then pomUkUkOtec^:=pomUk^.ukLevy
        else pomUkUkOtec^:=pomUk^.ukPravy;
        dispose(pomUk);
      end else begin {mazany uzel ma oba podstromy}
        pomUkLevy:=pomUk^.ukLevy;
        if (pomUkLevy^.ukPravy=nil) then begin
          pomUkUkOtec^:=pomUkLevy;
        end else begin{ukPravy<>nil - musime najit nejpravejsiho}
          pomUkLP:=pomUkLevy^.ukPravy;
          while pomUkLP<>nil do begin
            otecNejpravejsiho:=nejpravejsi;
            nejpravejsi:=pomUkLP;
            pomUkLP:=pomUkLP^.ukPravy;
          end;
          pomUk^.key:=nejpravejsi^.key;
          pomUk^.data:=nejpravejsi^.data;
          otecNejpravejsiho^.ukPravy:=nejpravejsi^.ukLevy;
        end;
        dispose(pomUk);
      end;
  end;
end;
```

Průchody stromem

```
Procedure preorder(uzel: tUkUzel);
begin
  if uzel <> nil then with uzel^ do
  begin
      writedata(data);
      preorder( ukLevy );
      preorder( ukPravy );
  end;
End;
```

Průchody stromem

```
Procedure inorder(uzel: tUkUzel);
begin
  if uzel <> nil then with uzel^ do
  begin
      inorder( ukLevy );
      writedata(data);
      inorder( ukPravy );
  end
end
```

Průchody stromem

```
Procedure postorder(uzel: tUkUzel);
begin
  if uzel <> nil then with uzel^ do
  begin
      postorder( ukLevy );
      postorder( ukPravy );
      writedata(data);
  end
end
```

Degradace stromu

- Pokud budeme do stromu vkládat již seřazené hodnoty – např. 10, 5, 3, 2 , vznikne strom degradovaný na lineární seznam
- Aby se to nestalo, je potřeba algoritmus vkládání obohatit o algoritmus AVL nebo Red-Black