▲ Vyberte vhodnou datovou strukturu, kterou byste použili k řešení/simulaci dané úlohy:

1. Kontrola párovosti různých typů závorek v textu.
2. Obsluha I/O požadavků na PC.
3. Zpracování tiskových úloh na tiskárně.
4. Vyhodnocení volání rekurzivních procedur.
5. Evidence účastníků přijímacích zkoušek.
6. Vyhodnocování aritmetických výrazů zapsaných postfixově.
7. Vyřizování požadavků přetíženou linkou technické podpory.

▲ Mějme zásobník definovaný nad polem S následujícím způsobem:

```
1   Stack = record
2      prvek: array[1..n] of Integer      {jednotlive prvky zasobniku}
3      topindex: Integer                  {ukazatel na vrchol zasobniku}
4   end
5   ------------------------------------------------------------------
6   function Init(S)            |   function Push(S,x)
7   begin                       |   begin
8     S.topindex := 0           |      S.topindex := S.topindex + 1
9   end                         |      S.prvek[S.topindex] := x
10                              |   end
11                              |
12  function StackIsEmpty(S)    |   function Pop(S)
13  begin                       |   begin
14    if (S.topindex = 0) then  |      if (StackIsEmpty(S))
15      return true             |           then call error "underflow"
16    else                      |      else
17      return false            |           x := S.prvek[S.topindex]
18  end                         |           S.topindex := S.topindex - 1
19                              |           return x
20                              |   end
```

Znázorněte jednotlivé stavy zásobníku během volání následujících funkcí:
`Init(S)`, `Push(S,7)`, `Pop(S)`, `Push(S,3)`, `Push(S,11)`,
`Pop(S)`, `Push(S,21)`, `Pop(S)`, `Pop(S)`, `Pop(S)`.

▲ Navrhněte implementaci dvou zásobníků v poli délky n tak, aby došlo k přetečení obou zásobníků pouze v případě, že počet prvků v obou zásobnících dohromady je větší než n.

▲ Jakou časovou složitost má přidání a odebrání prvku v implementaci z předchozí úlohy?

▲ Mějme cyklickou frontu definovanou nad polem `Q.prvek` pevné délky `n`. Atribut `Q.tail` ukazuje za konec fronty a `Q.head` na čelo fronty. Funkce `Enqueue` má na starosti zařazení prvku do fronty, funkce `Dequeue` jeden prvek z fronty vypustí.

```
1  Queue = record                       |   function Dequeue(Q)
2    prvek: array[1..n] of Integer     |   begin
3    head: Integer                     |     x := Q.prvek[Q.head]
4    tail: Integer                     |     if Q.head = n then
5  end                                 |       Q.head := 1
6                                      |     else
7  function Enqueue(Q,x)               |       Q.head := Q.head + 1
8  begin                               |     return x
9    Q.prvek[Q.tail] := x              |   end
10   if Q.tail = n then                |
11     Q.tail := 1                     |
12   else                              |
13     Q.tail := Q.tail + 1            |
14 end                                 |
```

Jak vypadá inicializace takové cyklické fronty?

Znázorněte jednotlivé stavy fronty během volání následujících funkcí: `Enqueue(Q,4)`, `Enqueue(Q,7)`, `Enqueue(Q,11)`, `Dequeue(Q)`, `Enqueue(Q,21)`, `Dequeue(Q)`, `Dequeue(Q)`.

Co se stane, když provedete operaci Dequeue(Q) na čerstvě inicializované frontě Q?

▲ Navrhněte, jak implementovat frontu pomocí dvou zásobníků.

▲ Určete časovou složitost operací `Enqueue(In,Out,x)` a `Dequeue(In,Out)` s frontou definovanou v předchozí úloze pro vstupní sekvenci délky n.

▲ Definice: Seznam (`List`) je dynamická datová struktura, jejíž prvky tvoří posloupnost. Na rozdíl od pole, kde je pořadí prvků určeno pevnými indexy, pořadí v seznamu je dáno vazbami mezi sousedními prvky. V následujících funkcích je zadefinován obousměrně zřetězený spojový seznam a operace nad ním, tj. seznam, kde každý prvek "vidí" (ukazuje na) svého předchůdce i následníka.

```
 1  Element = record              |   function Init(L)
 2    key : Integer               |   begin
 3    prev,next : ^Element        |      L.head = nil
 4  end                           |   end
 5                                |
 6  List = record                 |   function Search(L,k)
 7    head : ^Element             |   begin
 8  end                           |     x := L.head
 9                                |     while (x <> nil) AND (x.key <> k) do
10  function Init(L)              |       x := x.next
11  begin                         |     return x
12     L.head = nil               |   end
13  end                           |
14                                |   function Delete(L,x)
15  function Insert(L,x)          |   begin
16  begin                         |      if x.prev <> nil then
17     x.next := L.head           |        x.prev.next := x.next
18     if L.head <> nil then      |      else
19       L.head.prev := x         |        L.head := x.next
20     L.head := x                |      if x.next <> nil then
21     x.prev := nil              |        x.next.prev := x.prev
22  end                           |   end
```

Je možné realizovat operace přidávání a odebírání prvků v **jednosměrně** zřetězeném seznamu (v seznamu, kde prvek "vidí" jen napravo od sebe, ale již ne nalevo) v časové složitosti $O(1)$?

▲ Navrhněte implementaci operací `Insert(S,x)`, `Delete(S,x)` a `Search(S,x)` ve slovníku, který je realizován jednosměrně zřetězeným seznamem. Určete asymptotickou časovou složitost vámi navržených operací.

▲ Navrhněte nerekurzivní proceduru, která obrátí pořadí prvků v jednosměrně zřetězeném seznamu o n prvcích v čase $\Theta(n)$, a to tak, aby procedura využívala pouze konstantní množství paměti nad rámec paměti potřebné k uchování seznamu. Použijte operace seznamu z předcházejícího příkladu, k polím prvků nepřistupujte přímo.

▲ Nakreslete binární strom od uzlu `root`, který je reprezentovaný kódem:

```
1 root = Node (3,A) a b
2 a = Node (9,K) c d
3 b = Node (7,F) Empty e
4 c = Node (1,D) Empty Empty
5 d = Node (4,G) Empty Empty
6 e = Node (6,W) f g
7 f = Node (10,O) Empty Empty
8 g = Node (5,I) Empty Empty
```

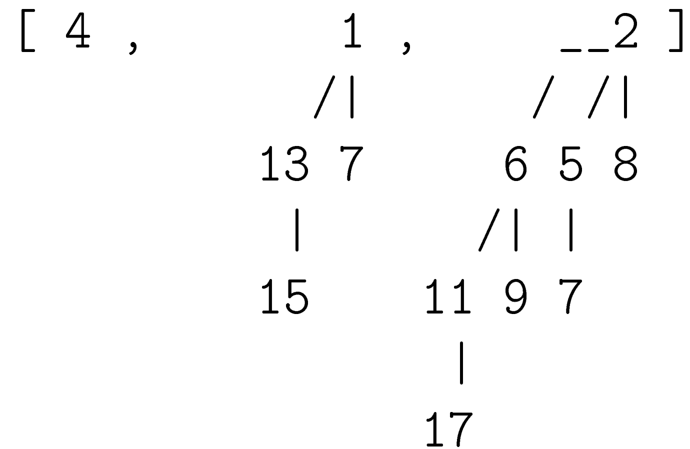
▲ Napište rekurzivní proceduru, která vypíše hodnoty všech uzlů zadaného binárního stromu s n uzly v čase $\mathcal{O}(n)$.

▲ Napište nerekurzivní proceduru, která vypíše hodnoty všech uzlů zadaného binárního stromu s n uzly v čase $\mathcal{O}(n)$ s využitím zásobníku.

▲ Napište proceduru, která ověří, zda daný binární strom je také binární haldou. Předpokládejte, že máme implementovanou funkci `get_length(node, min_length, max_length)`, která pro zadaný uzel `node` vrátí délku minimální (`min_length`) a maximální (`max_length`) větve ze zadaného uzlu.

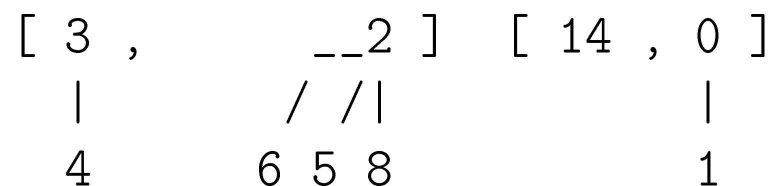
▲ Graficky znázorněte minimovou binomiální haldu vzniklou postupným přidáváním prvků [8,2,5,7,9,6,11,17,7,1,13,15,4].

▲ Vložte prvek 3 do binomiální haldy



Jaká je časová složitost vložení prvku do hlady?

▲ Slučte binomiální hlady:



```
     /|  |
   11 9 7
    |
   17
```

Jaká je časová zložitosť sloučení dvou hlad?

▲ Jak lze nalézt minimální klíč v binomiální haldě? S jakou časovou složitostí?

▲ Odeberte minimum z binomiální hlady

```
[ 14 ,   0 ,      __2 ]
        /|       / /|
       3 1      6 5 8
       |       /|  |
       4      11 9 7
               |
              17
```

Jaká je časová složitost tohoto algoritmu?