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

1. nakládka a vykládka zboží z dodávkového automobilu
2. obsluha stisknutých kláves na PC
3. zpracování tiskových úloh na tiskárně
4. vyhodnocení volání rekurzivních procedur
5. evidence účastníků sportovního utkání
6. obsluha požadavků při načítání www stránek
7. převrácení pořadí znaků v řetězci
8. vyřizování požadavků přetíženým zaměstnancem finančního úřadu

▲ Mějme zásobník definovaný nad polem S následujícím způsobem:

```
Stack = record
   prvek: array[1..n] of Integer      {jednotlive prvky zasobniku}
   topindex: Integer                  {ukazatel na vrchol zasobniku}
end
```

```
function Init(S)
begin
  S.topindex := 0
end
```

```
function Push(S,x)
begin
   S.topindex := S.topindex + 1
   S.prvek[S.topindex] := x
end
```

```
function StackIsEmpty(S)
begin
  if (S.topindex = 0) then
     return true
  else
     return false
end
```

```
function Pop(S)
begin
   if (StackIsEmpty(S))
        then call error "underflow"
   else
        x := S.prvek[S.topindex]
        S.topindex := S.topindex - 1
        return x
end
```

Znázorněte jednotlivé stavy zásobníku během volání následujících funkcí:
`Push(S,4)`, `Push(S,7)`, `Push(S,11)`, `Pop(S)`, `Push(S,21)`, `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.

```
Queue = record                          |   function Dequeue(Q)
   prvek: array[1..n] of Integer        |   begin
   head: Integer                        |      x := Q.prvek[Q.head]
   tail: Integer                        |      if Q.head = n then
end                                     |         Q.head := 1
----------------------------------------|      else
function Enqueue(Q,x)                   |         Q.head := Q.head + 1
begin                                   |      return x
   Q.prvek[Q.tail] := x                 |   end
   if Q.tail = n then                   |
      Q.tail := 1                       |
   else                                 |
      Q.tail := Q.tail + 1              |
end                                     |
```

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)`.

▲ 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.

```
Element = record          |    function Init(L)
   key : Integer          |    begin
   prev,next : ^Element   |       L.head = nil
end                       |    end
--------------------------------------------------------------
                          |    function Search(L,k)
List = record             |    begin
   head : ^Element        |      x := L.head
end                       |      while (x <> nil) AND (x.key <> k) do
                          |         x := x.next
                          |      return x
                          |    end
```

```
function Insert(L,x)          |   function Delete(L,x)
begin                         |   begin
   x.next := L.head           |      if (x.prev <> nil) then
   if head[L] <> nil then     |         (x.prev).next := x.next
      (L.head).prev := x      |      else
   L.head := x                |         head[L] := x.next
   x.prev := nil              |
end                           |      if x.next <> nil then
                              |         (x.next).prev := x.prev
                              |   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 třídy časových složitostí 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.

▲ Nakreslete binární strom, který je zadaný tabulkou:

index	klíč	L	P
1	D	nil	nil
2	CH	nil	nil
3	A	9	7
4	G	nil	nil
5	I	nil	8
6	W	10	5
7	F	6	2
8	E	nil	nil
9	K	1	4
10	O	nil	nil

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

▲ Napište nerekurzivní proceduru, která vypíše hodnoty všech uzlů zadaného binárního stromu s n uzly v čase $O(n)$ 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` vratí délku minimální (`min_length`) a maximální (`max_length`) větve ze zadaného uzlu.