

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

1. kontrola párovosti 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. převrácení pořadí znaků v řetězci
8. vyřizování požadavků přetíženou linkou technické podpory

▲ Mějme zásobník definovaný nad polem S následujícím způsobem:

Stack = record

prvek: array[1..n] of Integer {jednotlive prvky zásobníku}
topindex: Integer {ukazatel na vrchol zásobníku}

end

function Init(S)		function Push(S,x)
begin		begin
S.topindex := 0		S.topindex := S.topindex + 1
end		S.prvek[S.topindex] := x
		end

function StackIsEmpty(S)		function Pop(S)
begin		begin
if (S.topindex = 0) then		if (StackIsEmpty(S))
return true		then call error "underflow"
else		else
return false		x := S.prvek[S.topindex]
end		S.topindex := S.topindex - 1
		return x
		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.

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

Jak vypadá inicializace takové cyklické fronty?

Znáznorněte 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 $\text{Enqueue}(\text{In}, \text{Out}, x)$ a $\text{Dequeue}(\text{In}, \text{Out})$ 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 odebrání 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 $\text{Insert}(S, x)$, $\text{Delete}(S, x)$ a $\text{Search}(S, x)$ 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.

▲ Nakreslete binární strom, který je zadán 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` vrátí délku minimální (`min_length`) a maximální (`max_length`) větve ze zadaného uzlu.