ADATSZERKEZETEK ÉS ALGORITMUSOK

LIFO, FIFO Gyakorlati megvalósítás

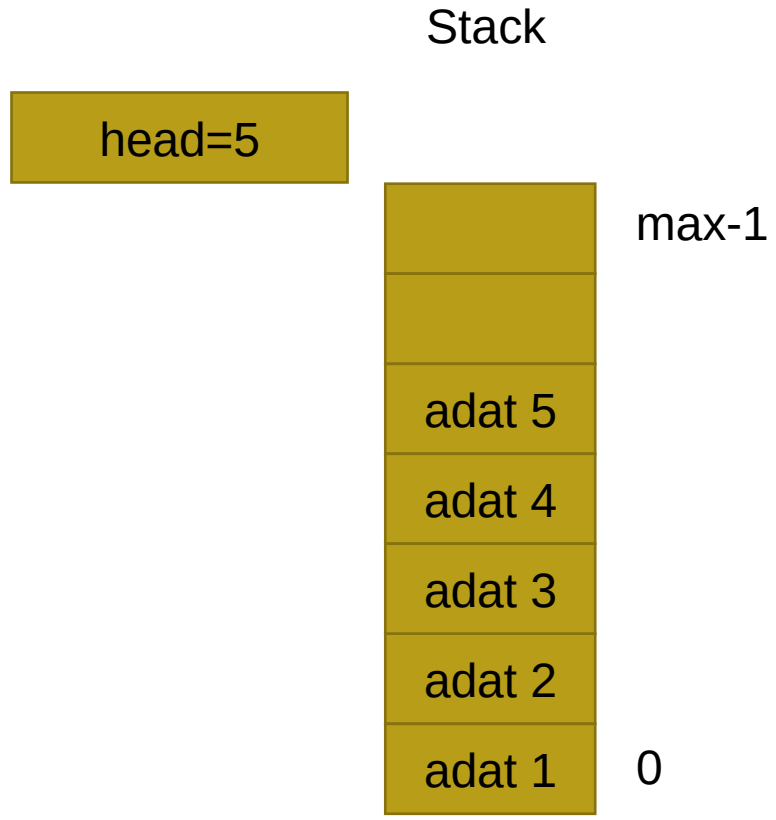
Verem – fix méretű megvalósítás

A verem egy `max` hosszú tömb és a `head` (`int`) direkt szorzata

A tömb elemei `[0…max-1]` között indexeltek)

A `head` az első szabad pozíciót jelzi a tömbben, ahova beszúrhatunk értéket

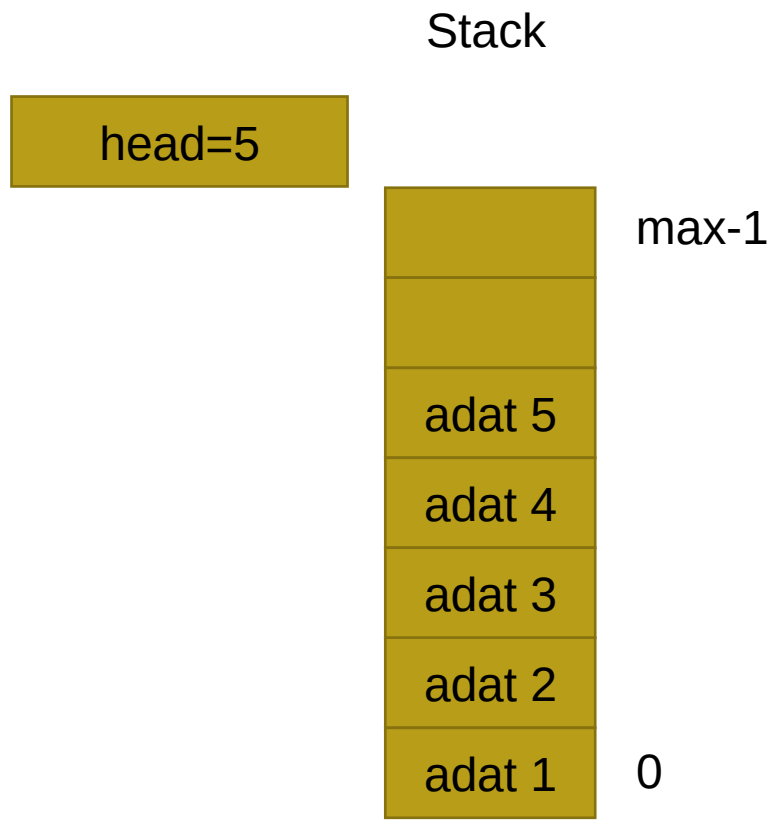
`0 ≤ head ≤ max`



Verem – fix méretű megvalósítás

Megvalósítás osztály segítségével:

```
class Stack{
  static const int max = 7;
 private:
  int tomb[max];
  int head;
 public:
  Stack();
  ~Stack();
  void push(int new_item);
  int pop();
  int top() const;
  bool isEmpty() const;
};
```



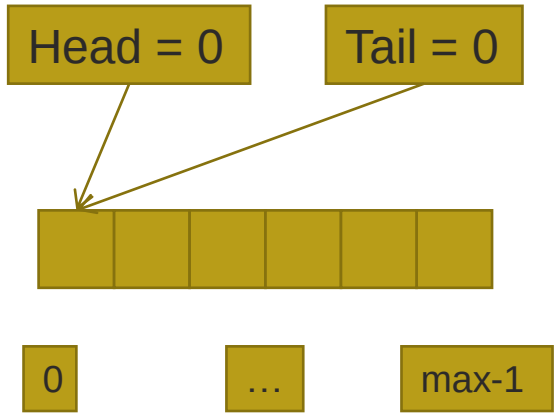
Verem – változtatható méretű megvalósítás

- A fix méretű megvalósítás korlátai:
  - Ha betelik a verem, nem tudjuk további elemek tárolására használni.
  - Ha túl nagy vermet hozunk létre, feleslegesen foglaljuk a memóriát.
- A megoldás természetesen a változtatható méretű (dinamikus) ábrázolás.
  - A dinamikus megvalósítást láncolással oldjuk meg.
  - Ez az jelenti, hogy létrehozunk egy osztályt a veremhez, amely tartalmazza az értéket, és egy pointerrel mutat a következő elemre.

```
class Node{
   public:
        int value;
        Node* pNext;
};
```

Sor – fix méretű megvalósítás

- A sor elemeit egy statikusan létrehozott `max` méretű tömbbel, a `head` és `tail` mutatókkal, `empty` paraméterrel reprezentáljuk.
  - elemei: `array[0...max-1]`
- A veremmel ellentétben a sornál a tömbnek mind a két végére szükségünk van, ezért azok helyét két változó, a `head`, és a `tail` fogják megadni.
- A megvalósításhoz ciklikus ábrázolást használunk

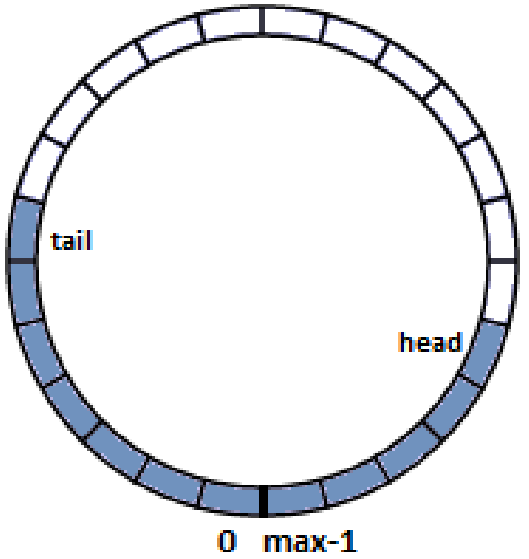


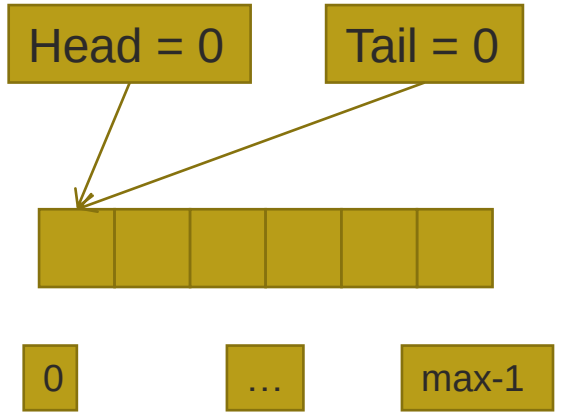
Sor – fix méretű megvalósítás

- Kezdetben a `head` és a `tail` a tömb ugyanazon elemének indexei.
  - `head`: a tömb első elemének indexe
  - `tail`: a tömb első szabad helyének indexe
- Mikor üres a sor?
  - Ha a `head` és a `tail` ugyanoda mutat, akkor a sor vagy üres, vagy tele van.
  - Ha üres, akkor az utolsó művelet szükségszerűen kivétel volt.
    - Ha az utolsó művelet betétel volt, akkor a sor most tele van.
  - Tartsunk karban egy változót, amellyel ezt követni tudjuk!

Sor – fix méretű megvalósítás

```
class FixedQueue {
 public:
  FixedQueue();
  ~FixedQueue();
  void in(int new_item);
  int out();
  int first() const;
  bool isEmpty() const;
  bool isFull() const;
 private:
  static const int CAPACITY = 10;
  int array[CAPACITY];
  int head, tail;
  bool empty;
};
```





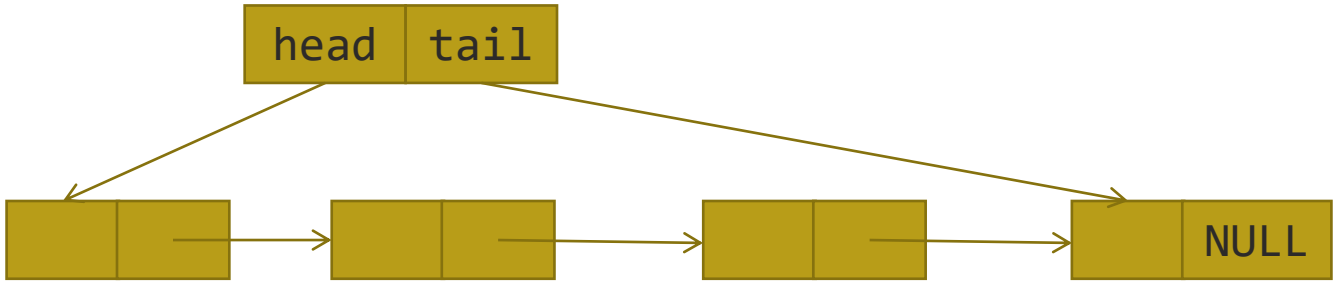
Sor – változó méretű megvalósítás

- A fix méretű megvalósítás korlátai:
  - Ha betelik a sor, nem tudjuk további elemek tárolására használni.
  - Ha túl nagy sort hozunk létre, feleslegesen foglaljuk a memóriát.
- Egy lehetséges megoldás a láncolásos implementáció.
  - Létrehozunk `Node` típusú elemeket, amely tartalmazzák az értéket, és egy pointerrel mutatnak a rákövetkező elemekre.

```
struct Node {
  int value;
  Node *pNext;
};
```

Sor – változó méretű megvalósítás

- A `head` és `tail` változók ebben az esetben nem indexek lesznek, hanem mutatók, mégpedig az első ill. az utolsó elemre mutatók. (`Node*`)



STL

- **Deque**
  - `template <class T, class Alloc = allocator<T>> class deque;`
- Double-ended sor, mely dinamikusan változó méretű tömbbel van implementálva.

- **Verem, sor**
  - `template <class T, class Container = deque<T>> class stack;`
  - `template <class T, class Container = deque<T>> class queue;`
- Az STL a deque adatszerkezetet használja a hagyományos **verem** és **sor** megvalósítására.

Verem gyakorlatban

Következő téma