CSC230

Intro to C++ Lecture 18

Outline

- Stack
- Queue
- Lab 8 discussion

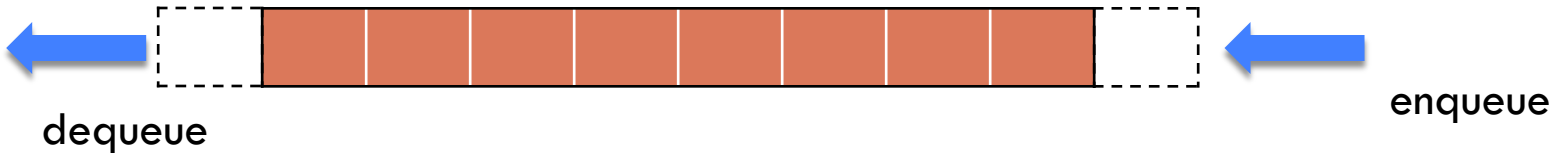
Data types

Stack:

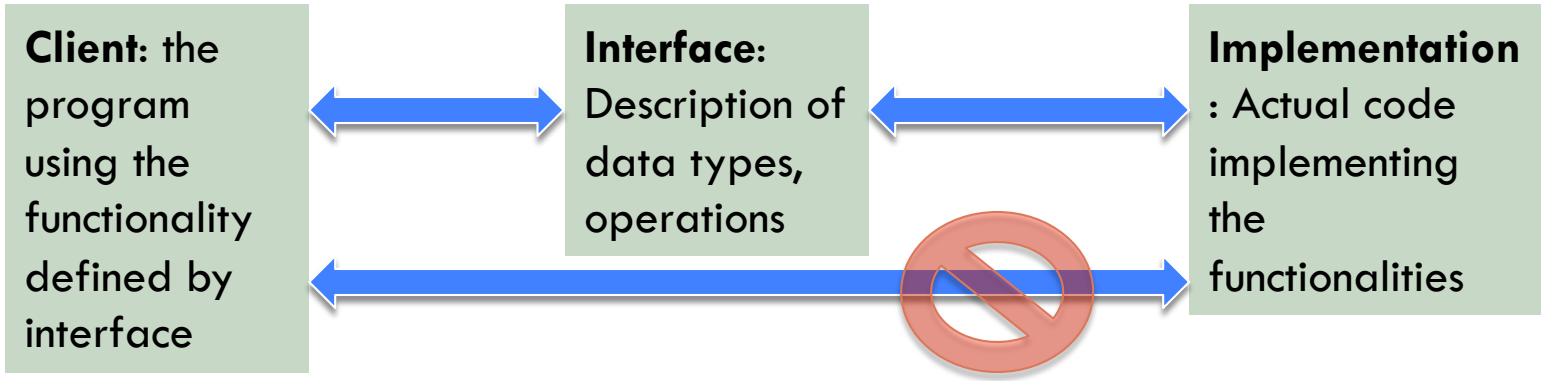
- A group of elements
- The elements follow the rule of LIFO (last in, first out)

Queue:

- A group of elements
- The elements follow the rule of FIFO (First in, first out)



Client, implementation, interface



- Client just need to know interface, which specify how to use the data type
- Client does not have to know the implementation details, which can be changed
- Implementation does not have to know who is the client, which may change

Data Structure Building Blocks

- These are *implementation* "building blocks" that are often used to build more-complicated data structures
  - Arrays
  - Linked Lists
    - Singly linked
    - Doubly linked
  - Binary Trees
  - Graphs
    - Adjacency matrix
    - Adjacency list

From interface to implementation

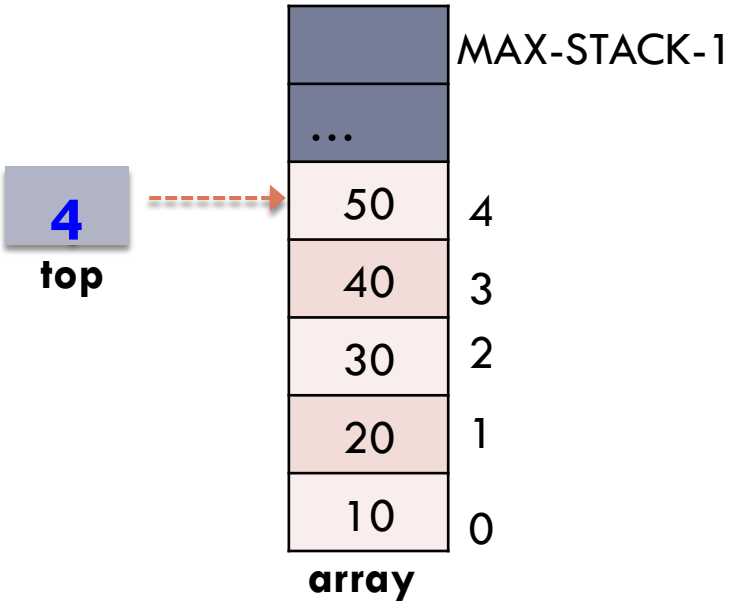
- Given that we want to support some interface, the designer still faces a choice
  - What will be the best way to implement this interface for my expected type of use?
  - Choice of implementation can reflect many considerations

- Major factors we think about
  - **Speed** for typical use case
  - **Storage space** required

Stack ADT

Operation	Description	Input(s)
push	Adds one element to the stack	item
pop	Removes (also returns) the last element that was added	
top	Returns (without removal) the last element that was added	
size	Returns the size of the stack	
empty	Return whether the stack is empty or not	

Array-based stack



- Array size is fixed
- Variable top indicates the top position of the stack

stackInterface.h


```
template<class ItemType>
class StackInterface
{
public:
   virtual bool empty() const = 0;

   virtual bool push(const ItemType& newEntry) = 0;

   virtual bool pop() = 0;

   virtual ItemType peek() const = 0;
};
```

empty(), push(), pop(), peek() are **pure virtual functions,** which must be implemented by the derived classes.

empty() const = 0

- means empty() will not change the object.

ArrayStack.h


```
const int MAX_STACK = 1000;

template<class ItemType>
class ArrayStack : public StackInterface<ItemType>
{
private:
        ItemType items[MAX_STACK]; // Array of stack items
        int      top;              // Index to top of stack

public:
        ArrayStack();             // Default constructor
        bool empty() const;
        bool push(const ItemType& newEntry);
        bool pop();
        ItemType peek() const;
};
```

ArrayStack::push()


```
template<class ItemType>
bool ArrayStack<ItemType>::push(const ItemType& newEntry)
{
        bool result = false;
        if (top < MAX_STACK - 1)  // Enough room?
        {
                top++;
                items[top] = newEntry;
                result = true;
        }
        return result;
}
```

isEmpty() and pop()


```
template<class ItemType>
bool ArrayStack<ItemType>::empty() const
{
        return top < 0;
}
```

Does not change the object

```
template<class ItemType>
bool ArrayStack<ItemType>::pop()
{
        bool result = false;
        if (!empty())
        {
                top--;
                result = true;
        }
        return result;
}
```

Array-based stack considerations

Underflow: **Peek/pop** an **empty** stack. The function should throw an exception.
Overflow: Push data when the array is full. The function should resize the array.

```
template<class ItemType>
ItemType ArrayStack<ItemType>::peek() const
{
  if (empty())
     throw PrecondViolatedExcep("peek() called with empty stack");

  return items[top];
}
```

Array-based stack: resizing

Q: How to grow and shrink array?

A: Create new array, copy the data from the existing array to the new array

First try:

- push(): increase array size by 1
- pop(): decrease array size by 1

Too expensive

- Whenever a new array is created, all items must be copied to the new array

If we push N items to the stack

- Need to push new items N times
- Need to read exiting items 1+2+3+4…+ (N-1) times
- Need to write the existing items to the new array 1+2+3+4…+ (N-1) times
- Total work = N + (1+2+3+4…+ (N-1)) + (1+2+3+4…+ (N-1)) ≈ N^2

Array-based stack: resizing

Q. What array size should we use when we grow it?

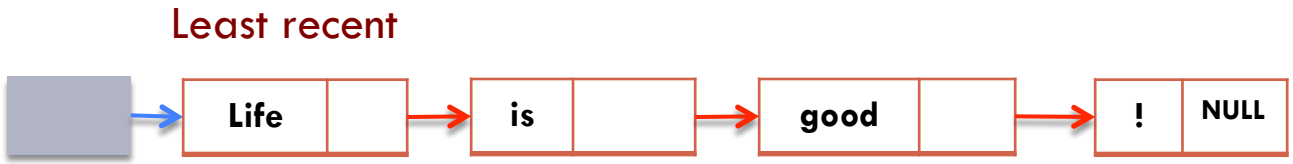
A. Double the current array size.

The array sizes are $1, 2^1, 2^2, 2^3 \ldots (2^i = N)$

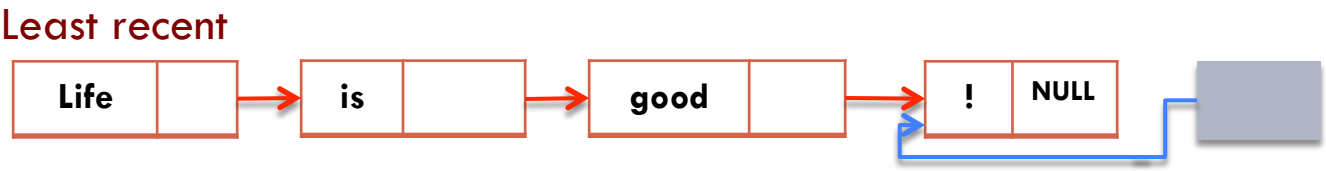
- Need to push the new items N times
- Need to read the existing items $1+ 2^1+ 2^2+ 2^3 \ldots$ times
- Need to write the existing items $1+ 2^1+ 2^2+ 2^3 \ldots$ times
- Total work = $N + (1+ 2^1+ 2^2+ 2^3 \ldots) + (1+ 2^1+ 2^2+ 2^3 \ldots \text{ times}) \approx 3N$

Stack implementation with linked list

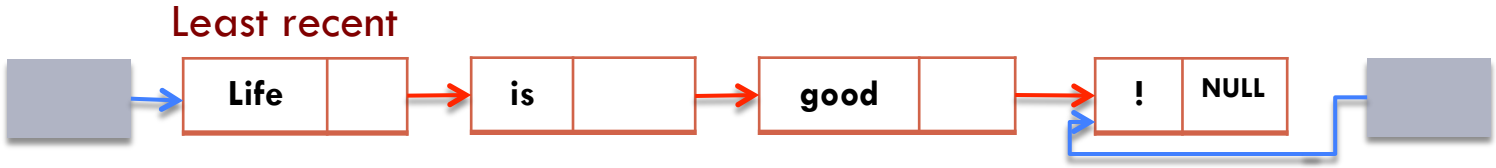
A.



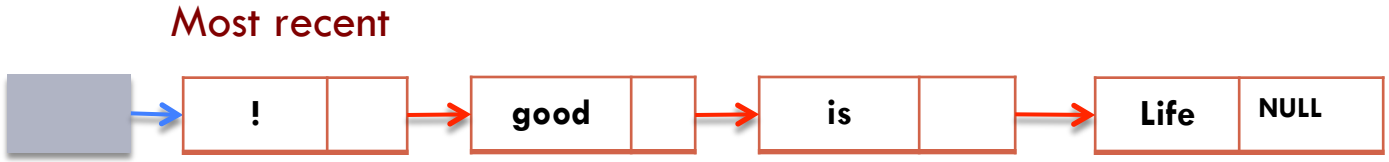
B.



C.



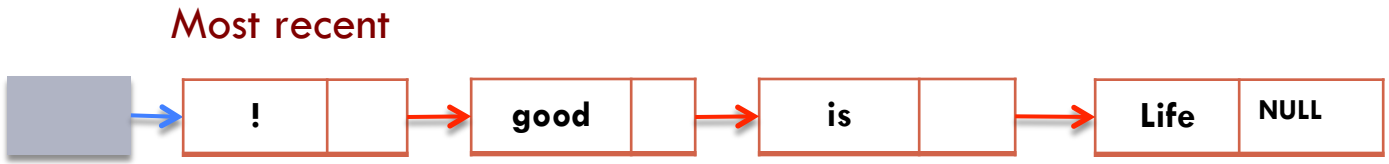
D.



Stack: singly-linked list

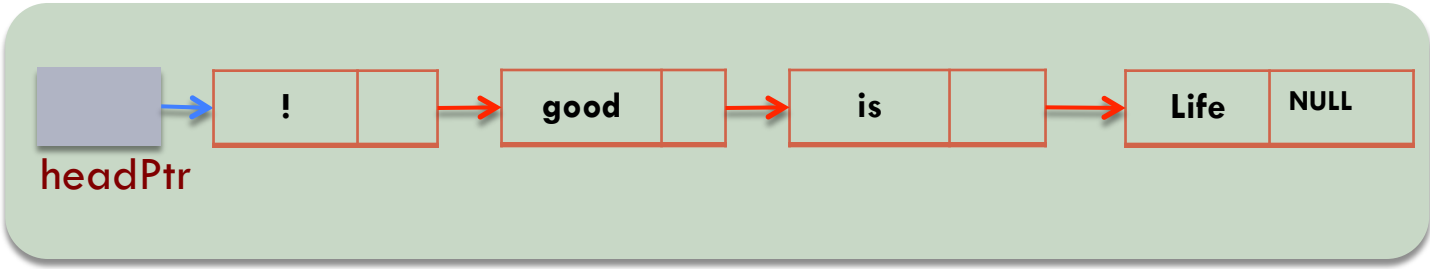
- Head pointer points to the most recently added node
- Push new node before the first node
- Pop the first node from the list

D.

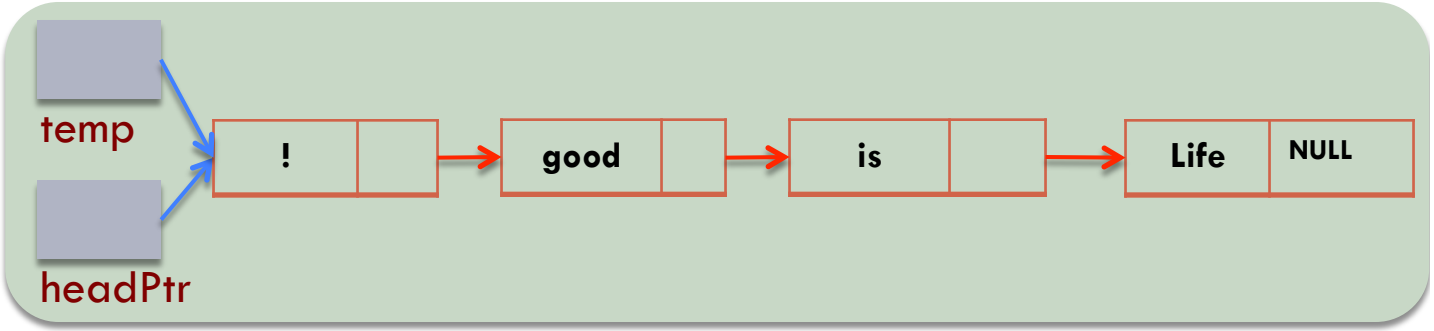


Stack pop: Linked-list implementation

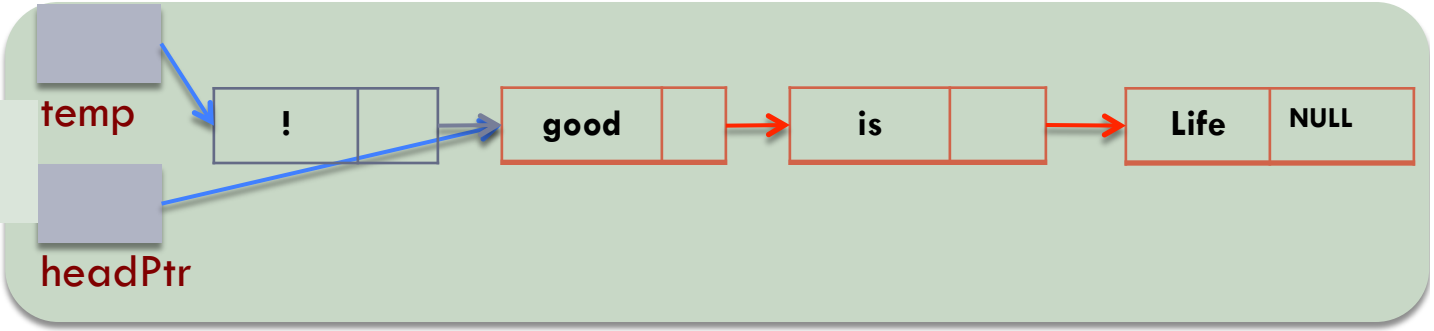
Before pop()



Node *temp = headPtr;



headPtr = headPtr->next;
return temp->item;



Stack pop: Linked-list implementation

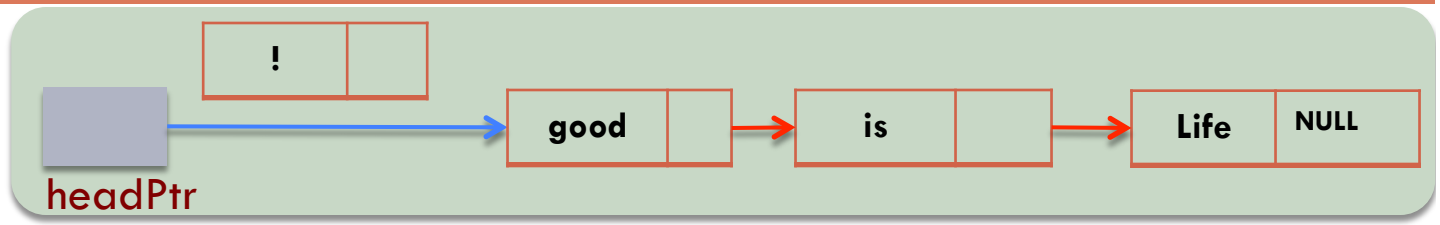

```
template<class ItemType>
bool LinkedStack<ItemType>::pop()
{
  bool result = false;
  if (!empty())
  {
     // Stack is not empty; delete top
     Node<ItemType>* temp = headPtr;
     headPtr = headPtr->getNext();
     // Return deleted node to system
     temp->setNext(nullptr);
     delete temp;
     temp = nullptr;

     result = true;
   }  // end if

   return result;
}
```

Stack peek: linked-list implementation

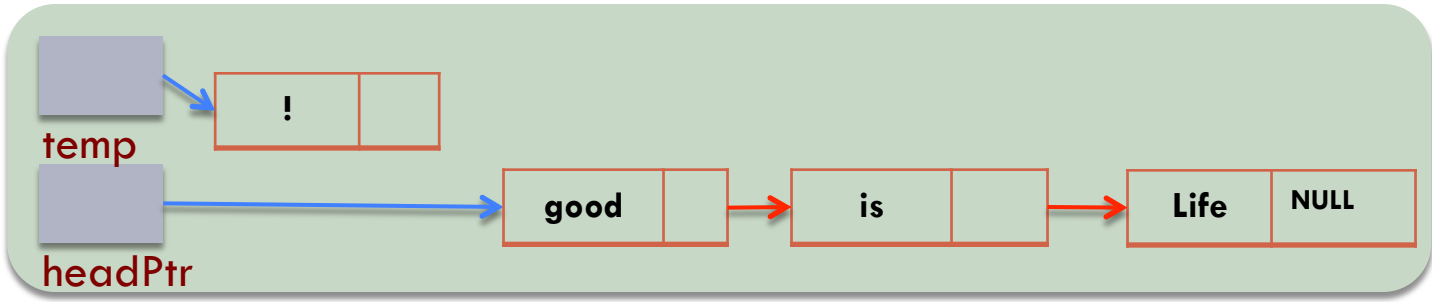

```
template<class ItemType>
ItemType LinkedStack<ItemType>::peek() const
{
        assert(!empty());  // Enforce precondition

        // Stack is not empty; return top
        return headPtr->getItem();
}
```

Stack push: Linked-list implementation

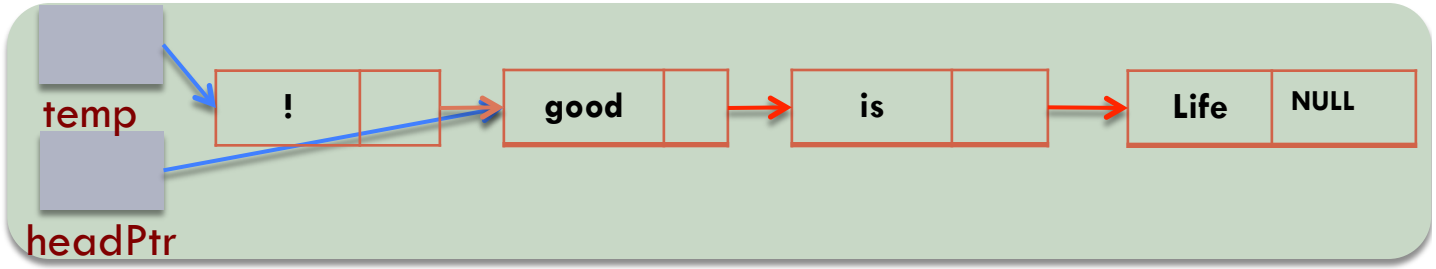
Before push()



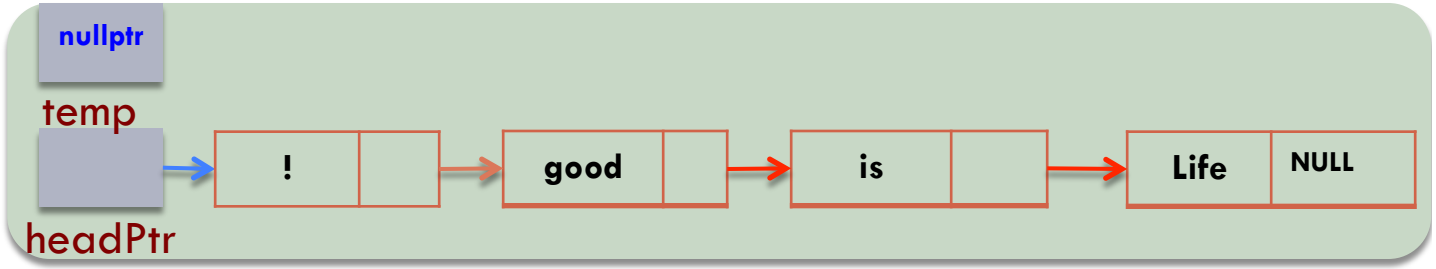
A.



B.



C.



Stack push: Linked-list implementation


```
template<class ItemType>
bool LinkedStack<ItemType>::push(const ItemType& newItem)
{
  Node<ItemType>* temp = new Node<ItemType>(newItem, headPtr);
  headPtr = temp;
  temp = nullptr;

  return true;
}
```

A. B.

C.

Create a Node object, newItem is the item value of the object. The next pointer of the object points to the node pointed by headPtr

Stack implementation: Array vs. Linked-list

Which one is better?

- Both resizing array and linked-list can implement stack. The user can use them interchangeably.

Linked-list implementation

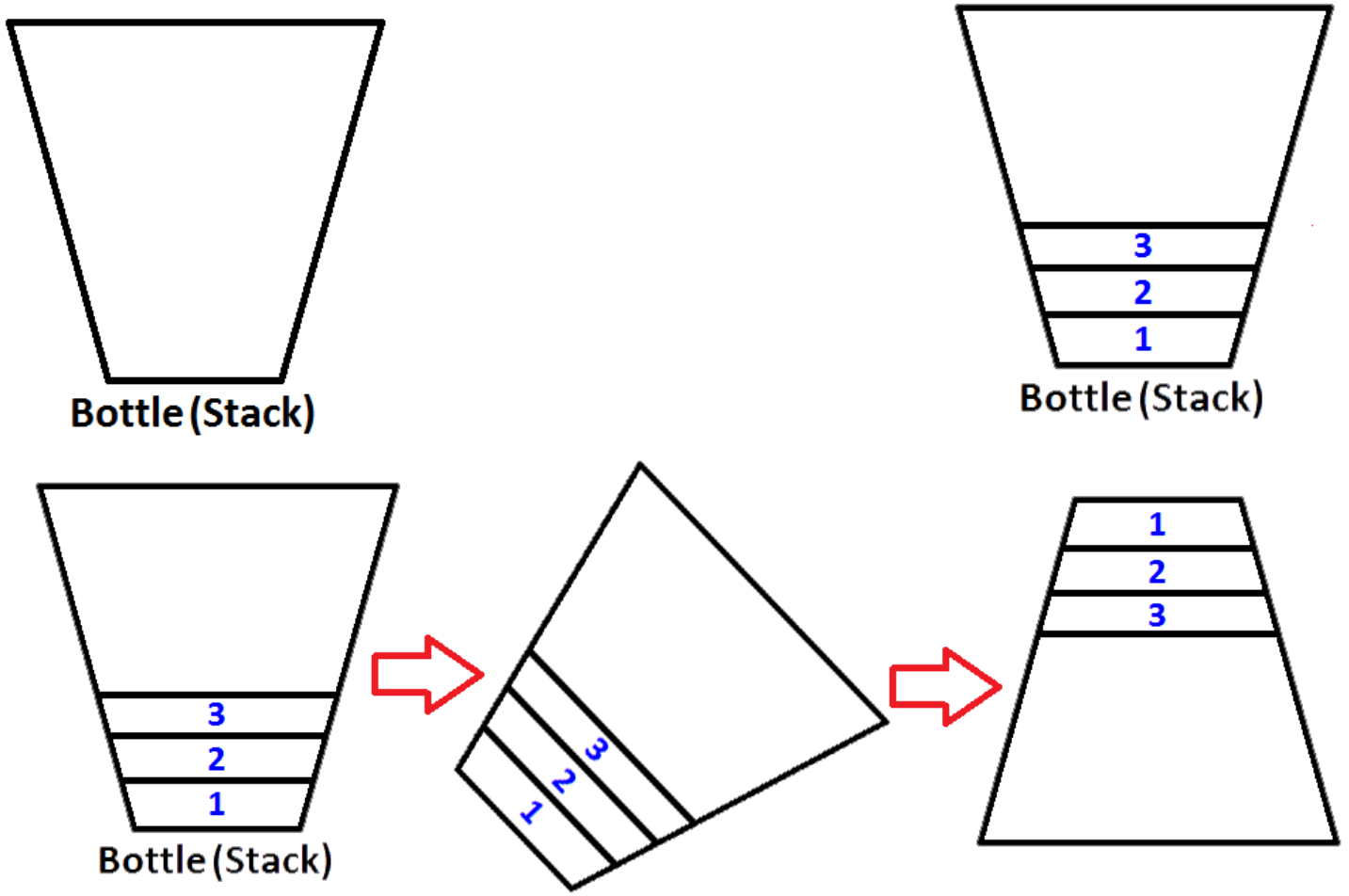
- Every operation takes constant time to finish
- Need extra time and space to handle pointers

Resizing-array implementation

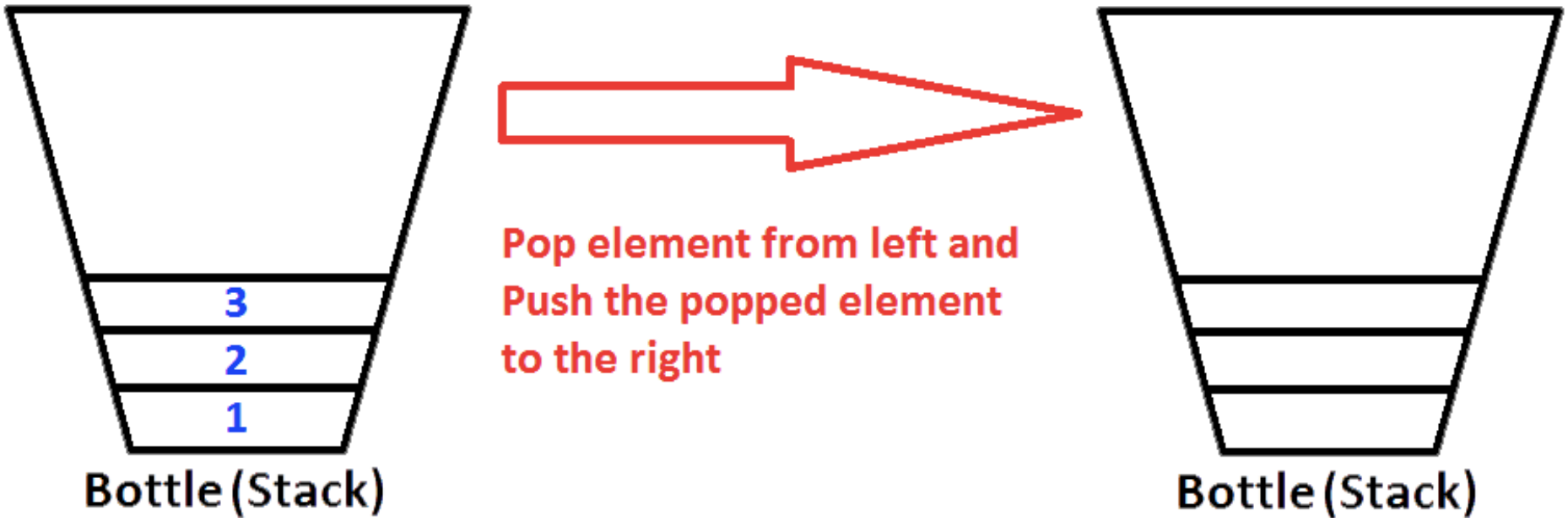
- Need extra time to copy data from existing array to the new array
- Cannot guarantee constant time to finish the operations
- On average, performance is good
- Take less space

- Example: stack.cpp

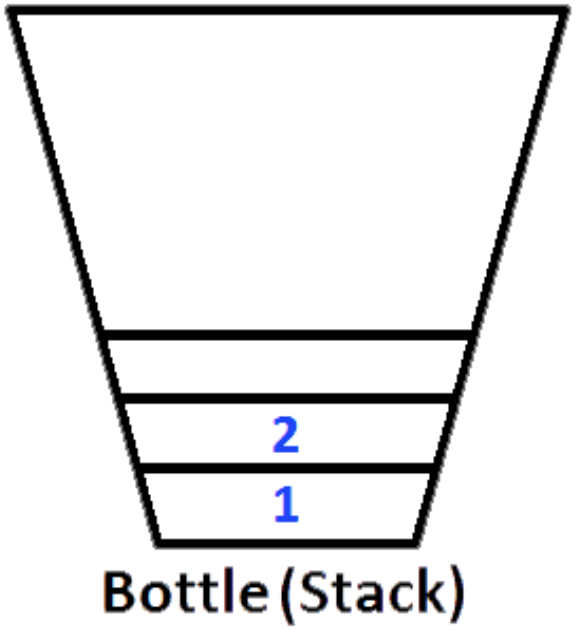
Challenge: How to reserve a stack



Challenge: How to reserve a stack

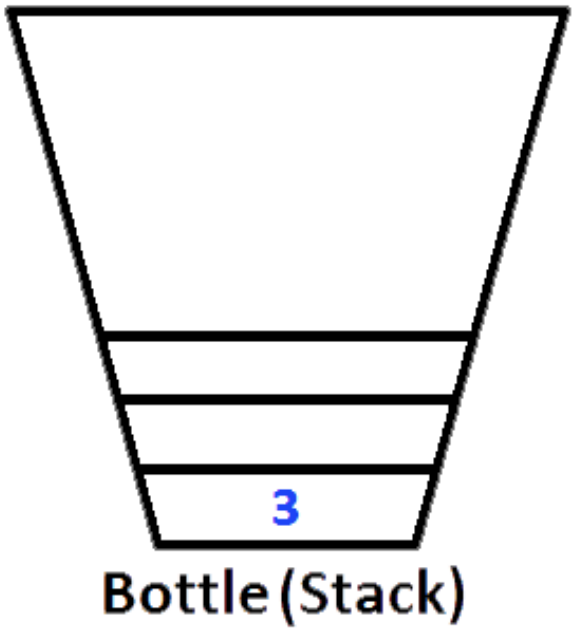


Challenge: How to reserve a stack

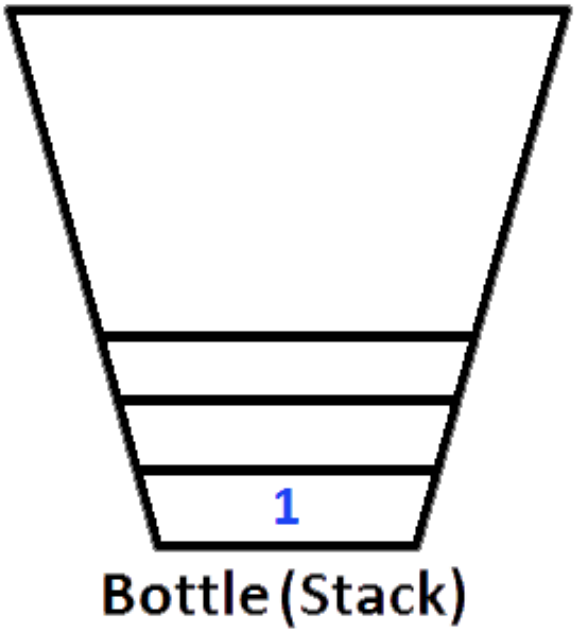


STEP #1

Pop 3 from left stack
Push 3 to the right stack

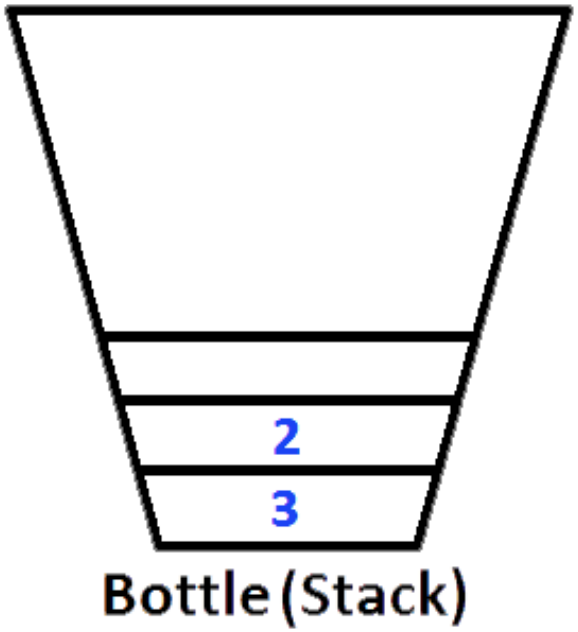


Challenge: How to reserve a stack

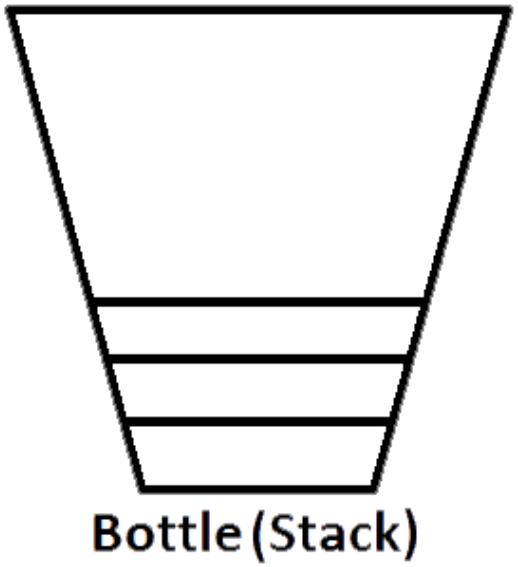


STEP #2

Pop 2 from left stack
Push 2 to the right stack

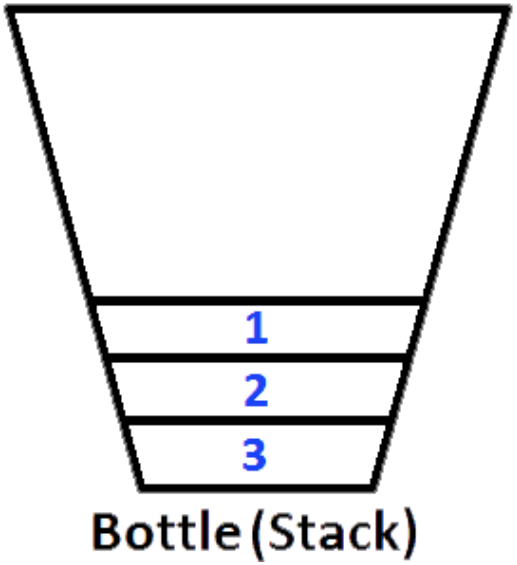


Challenge: How to reserve a stack



STEP #3

Pop 1 from left stack
Push 1 to the right stack



ADT Example: Queue

Queue:

- A group of elements
- The elements follow the rule of FIFO (First in, first out)

Where used:

- Simple job scheduler (e.g., print queue)
- Wide use within other algorithms

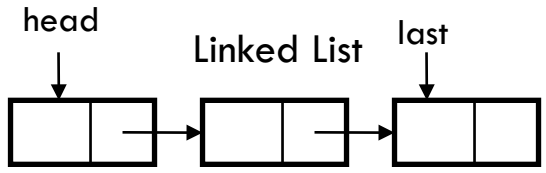
Operation	Description	Input(s)
enQueue	Adds one element to the queue	item
deQueue	Removes (also returns) the first element that was added	
front	Returns (without removal) the first element that was added	
size	Returns the size of the queue	
empty	Return whether the queue is empty or not	

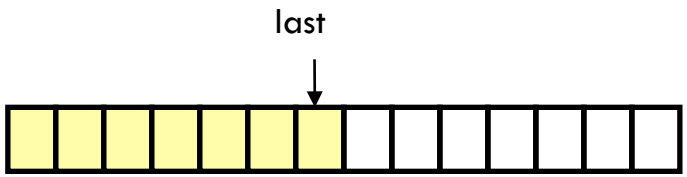
dequeue

enqueue

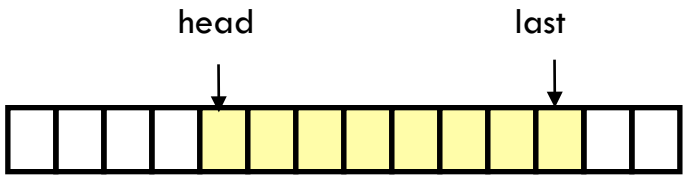
Queue Implementations

- Possible implementations





Array with head always at A[0]
(poll( ) becomes expensive)
(can overflow)



Array with wraparound
(can overflow)

- Recall: operations are **enQueue, deQueue, peek,…**

- For linked-list
  - All operations are O(1)

- For array with head at A[0]
  - deQueue takes time O(n)
  - Other ops are O(1)
  - Can overflow

- For array with wraparound
  - All operations are O(1)
  - Can overflow

- Example: queue.cpp

A Queue From 2 Stacks

- Add pushes onto stack A
- Poll pops from stack B
- If B is empty, move all elements from stack A to stack B
- Some individual operations are costly, but still O(1) time per operations over the long run

Dealing with Overflow

- For array implementations of **stacks** and **queues**, use *table doubling*
- Check for overflow with each insert op
- If table will overflow,
  - Allocate a new table twice the size
  - Copy everything over
- The operations that cause overflow are expensive, but still constant time per operation over the long run

Lab 8 discussion

- Implement a queue with two stack (first / second)