Stacks
COL 106

Slides by Amit Kumar, Shweta Agrawal

How should data be stored?

Depends on your requirement

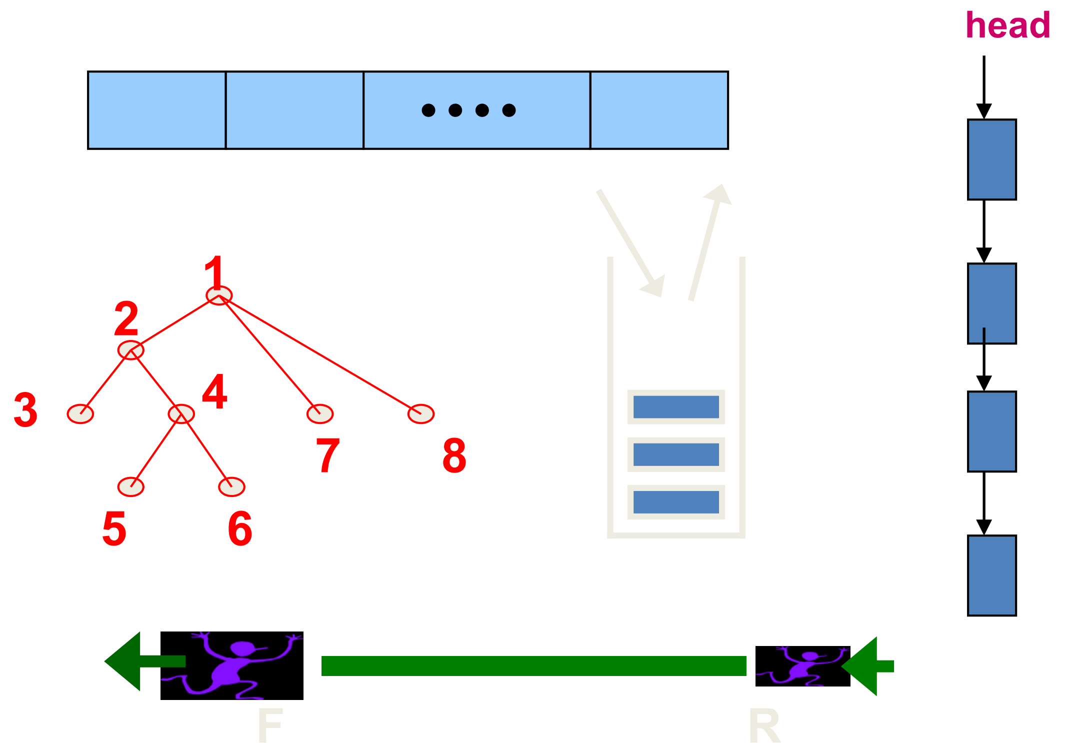
“We back up our data on sticky notes because sticky notes never crash.”

Data is diverse ..
But we have some building blocks

To store our big data

Elementary Data "Structures"

- **`Arrays`**
- **`Lists`**
- **`Stacks`**
- **`Queues`**
- **`Trees`**



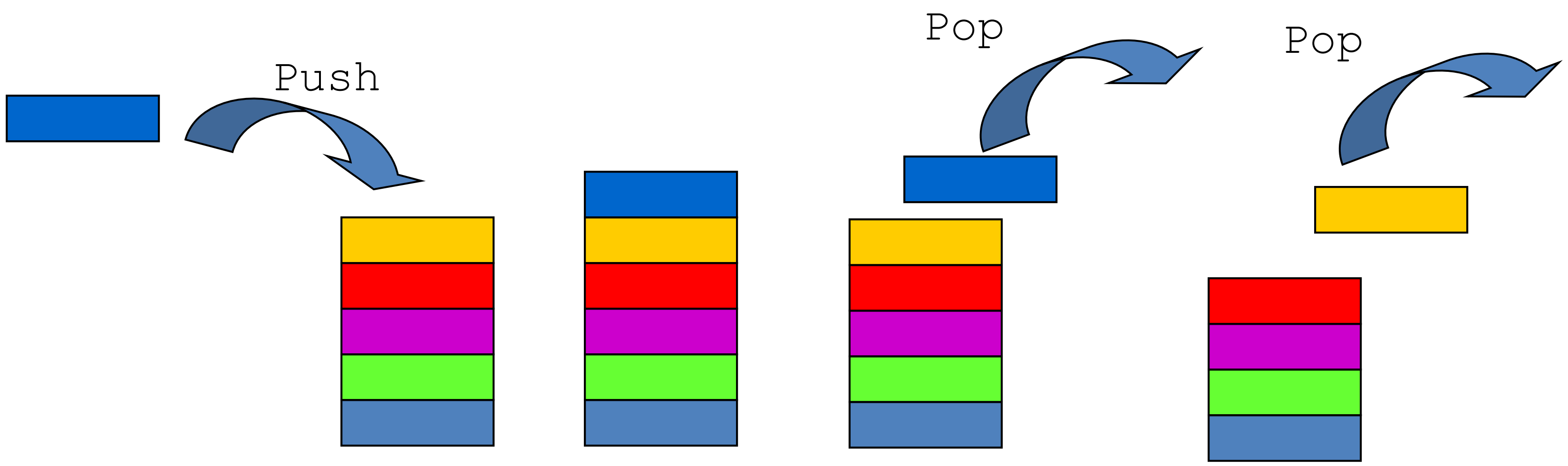
In some languages these are basic data types – in others they need to be implemented

Stacks

Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

– LIFO – Last in, First out



What is this good for ?

- Page-visited history in a Web browser

What is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor

What is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another

How should we represent it ?

- Write code in python ?

How should we represent it ?

- Write code in python ?
- Write code in C ?

How should we represent it ?

- Write code in python ?
- Write code in C ?
- Write code in Java ?

Aren't we essentially doing the same thing?

Abstract Data Type

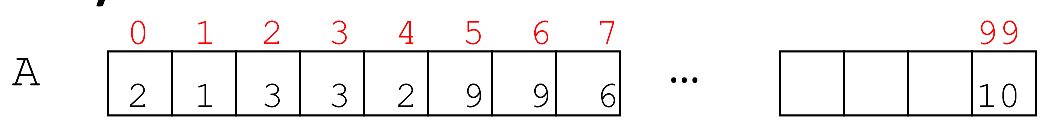
A mathematical definition of objects, with operations defined on them

Three operations

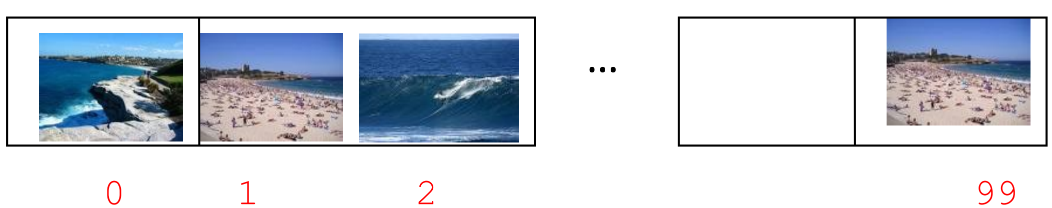
- constructors
- access functions
- manipulation procedures

Examples

- Basic Types
  - integer, real (floating point), boolean (0,1), character
- Arrays
  - A[0..99] : integer array



  - A[0..99] : array of images



ADT: Array

A mapping from an index set, such as **{0,1,2,…,n}**, into a cell type

Objects: set of cells

Operations:

- **create(A,n)**
- **put(A,v,i)** or **A[i] = v**
- **value(A,i)**

Abstraction

The notion of abstraction is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language.

Abstract Data Type

An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations

Abstract Data Types (ADTs)

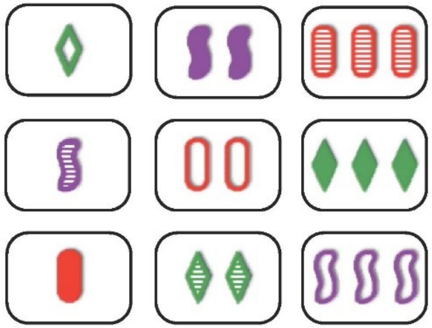
- An abstract data type (ADT) is an abstraction of a data structure

- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

ADT for stock trade

- The data stored are buy/sell orders
- The operations supported are
  - order buy (stock, shares)
  - order sell(stock, shares)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

Set ADT



Objects:

A bag of nodes

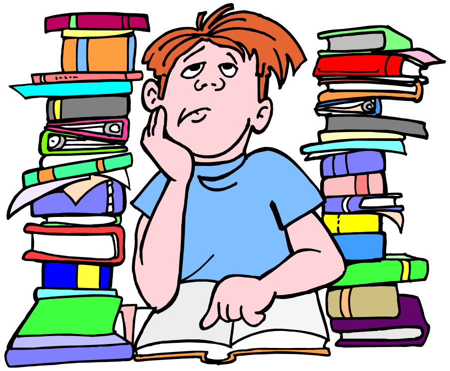
Operations:

- New():Set
- Insert(S:Set, v:element):Set
- Delete(S:Set, v:element):Set
- IsIn(S:Set, v:element):Boolean

Axioms

- IsIn(New(), v) = false
- IsIn(Insert(S,v), v) = true
- IsIn(Insert(S,u), v) = IsIn(S, v) if $v \neq u$
- IsIn(Delete(S,v), v) = false
- IsIn(Delete(S,u), v) = IsIn(S, v) if $v \neq u$

Stack ADT

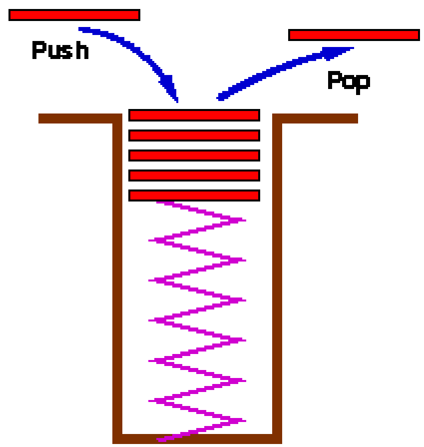


Objects:

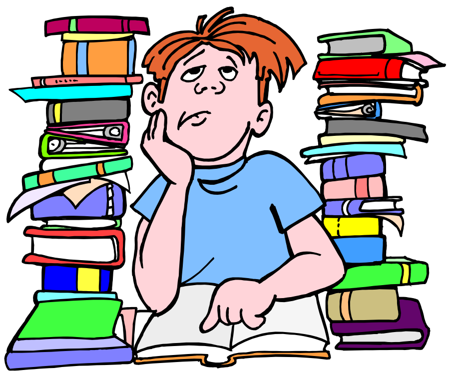
A finite sequence of nodes

Operations:

- New
- Push: Insert element at top
- Top: Return top element
- Pop: Remove top element
- IsEmpty: test for emptiness
- Size: number of elements in stack



Stack ADT

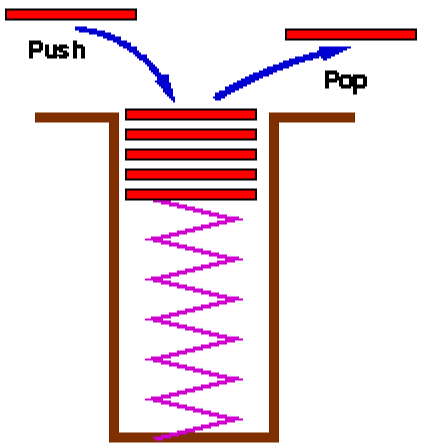


Objects:

A finite sequence of nodes

Operations:

- New():Stack
- Push(S:Stack, v:element):Stack
- Top(S:Stack):element
- Pop(S:Stack):Stack
- IsEmpty(S:Stack):Boolean
- Size(S:Stack):integer



Axioms

- Pop(Push(S,v)) = S
- Top(Push(S,v)) = v
- IsSize(New()) = 0
- IsSize(Push(S,v)) = IsSize(S)+1

Exceptions

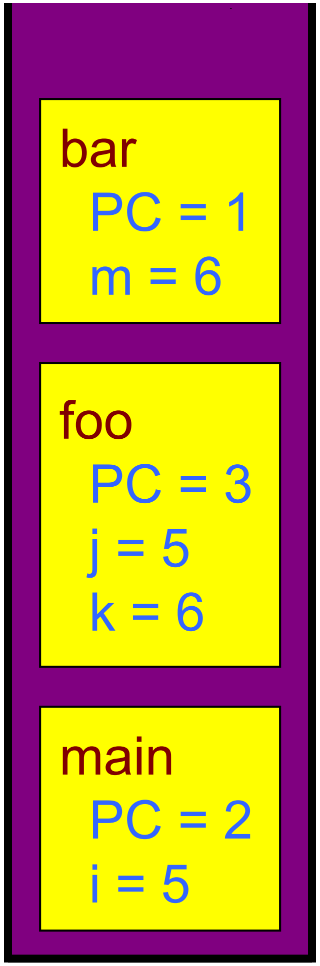
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Exercise: Stacks

- Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)

Java Run-time Stack

- The Java run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack



Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "["
  - ( )(( )){([( )])} – correct
  - ((( )(( )){([( )])})) – correct
  -)(( )){([( )])} – incorrect
  - ({[ ])} – incorrect
  - (– incorrect

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.isEmpty() then
            return false {nothing to match with}
        if S.pop() does not match the type of X[i] then
            return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else
    return false {some symbols were never matched}
```

Postfix Evaluator

- 5 3 6 * + 7 - = ?

Stack Interface in Java

- Interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException

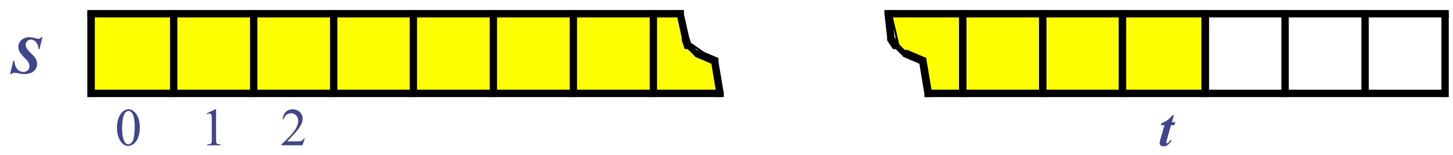
```
public interface Stack {

    public int size()
    public bool isEmpty()
    public Object top()
            throw(EmptyStackException)

    public void push(Object o)
    public Object pop()
            throw(EmptyStackException);
};
```

functionality of a data structure is expressed through the public interface of the associated class or classes that define the data structure.

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

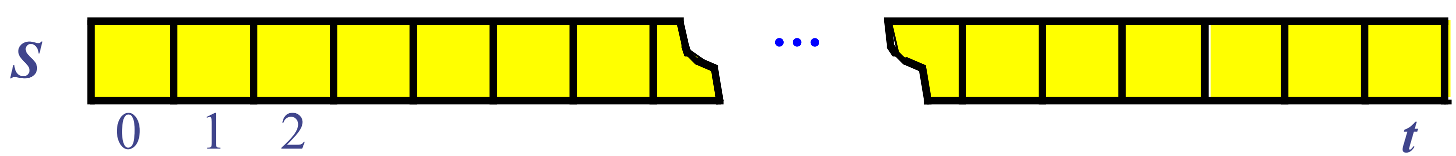
```
Algorithm size()
  return t + 1

Algorithm pop()
  if empty() then
    throw EmptyStackException
   else
    t = t - 1
    return S[t + 1]
```



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
  if t = S.length - 1 then
    throw FullStackException
   else
    t = t + 1
    S[t] = o
```



Performance and Limitations

of array-based implementation of stack ADT

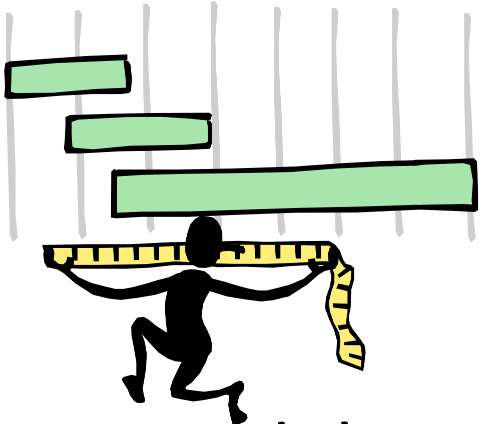
- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

```
Algorithm push(o)
  if t = S.length - 1
then
     A = new array of
          size …
     for i = 0 to t do
        A[i] = S[i]
        S = A
  t = t + 1
  S[t] = o
```

Comparison of the Strategies



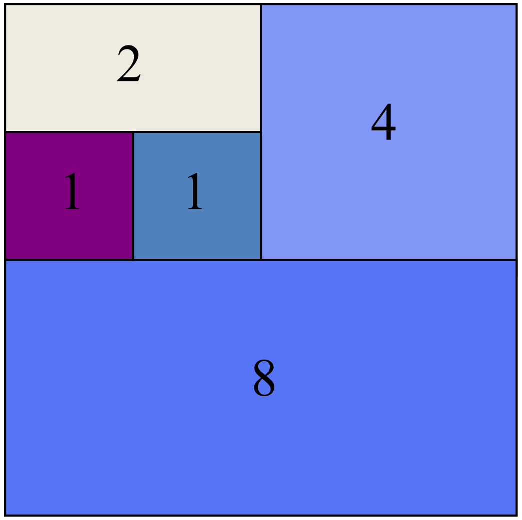
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $\boldsymbol{k} = \boldsymbol{n}/\boldsymbol{c}$ times
- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ push operations is proportional to
  - $\boldsymbol{n} + \boldsymbol{c} + 2\boldsymbol{c} + 3\boldsymbol{c} + 4\boldsymbol{c} + \ldots + \boldsymbol{kc} =$
    - $\boldsymbol{n} + \boldsymbol{c}(1 + 2 + 3 + \ldots + \boldsymbol{k}) =$
      - $\boldsymbol{n} + \boldsymbol{ck}(\boldsymbol{k} + 1)/2$
- Since $\boldsymbol{c}$ is a constant, $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{k}^2)$, i.e., $\boldsymbol{O}(\boldsymbol{n}^2)$
- The amortized time of a push operation is $\boldsymbol{O}(\boldsymbol{n})$

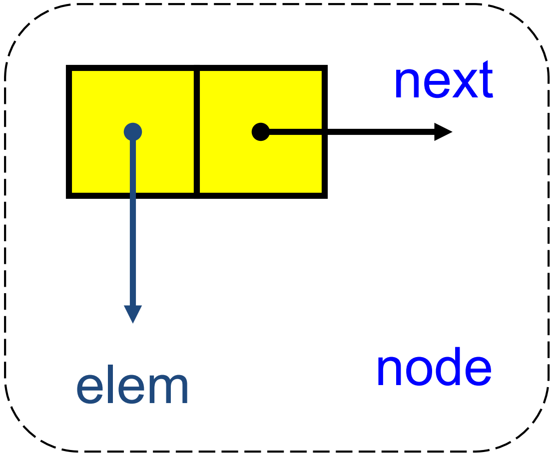
Doubling Strategy Analysis

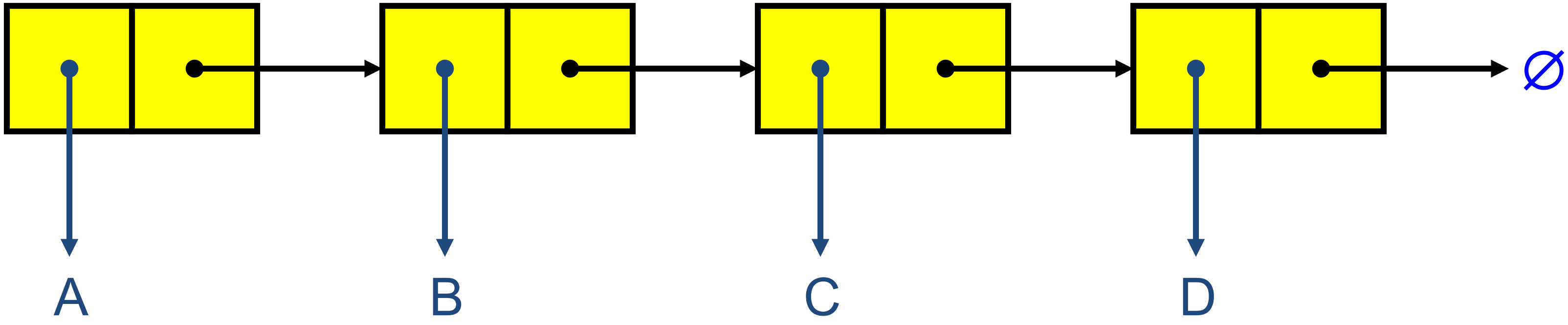
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to
  - $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$
    - $n + 2^{k+1} - 1 = 3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$



Singly Linked List

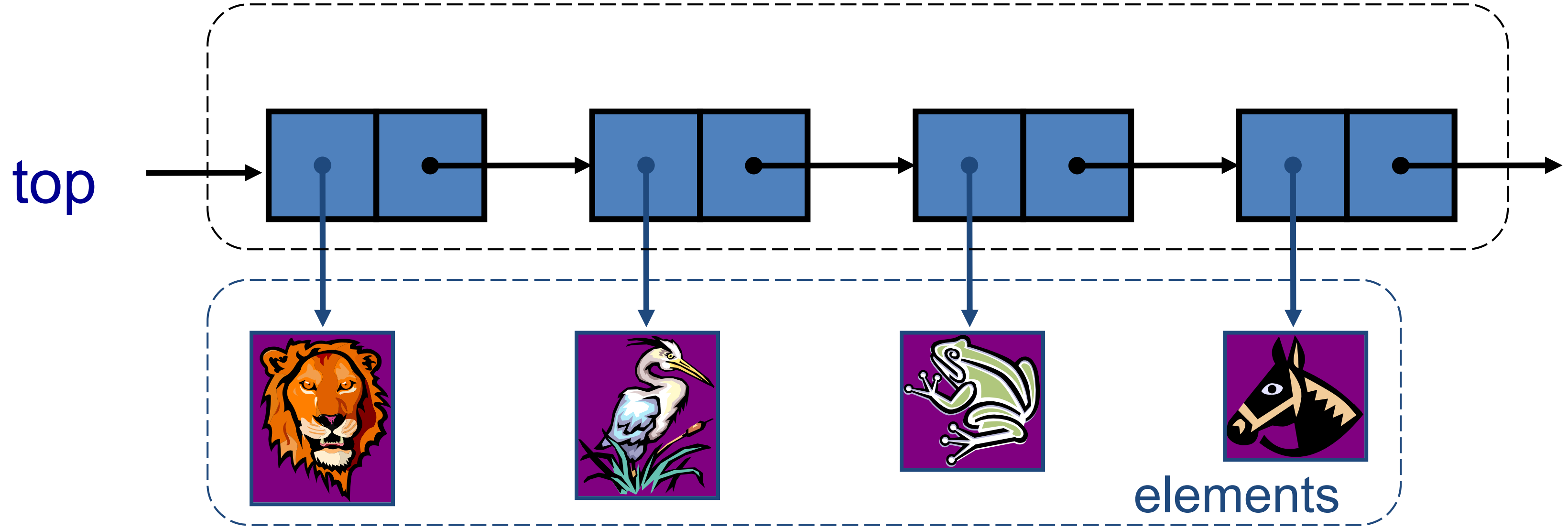
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Exercise

- Describe how to implement a stack using a singly-linked list
  - Stack operations: push(x), pop(), size(), isEmpty()
  - For each operation, give the running time

Stack Summary

- Stack Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
Pop()	O(1)	O(1)	O(1)
Push(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Amortized	O(1)
Top()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)