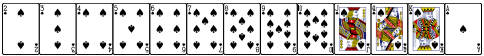
COMP 2011: Data Structures

Lecture 3. Arrays and Stacks

Dr. CAO Yixin

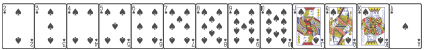
yixin.cao@polyu.edu.hk

September, 2021



Review of Lecture 2

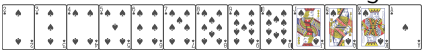
- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.



Review of Lecture 2

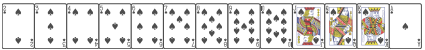
- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.


After Mangkhut (🌀): Hong Kong was safe in the worst typhoon (2018).



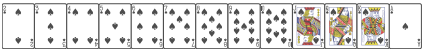
Review of Lecture 2

- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.



Review of Lecture 2

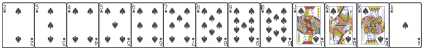
- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.



Review of Lecture 2

- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.

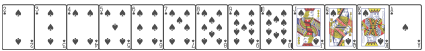
by Felix Reidl

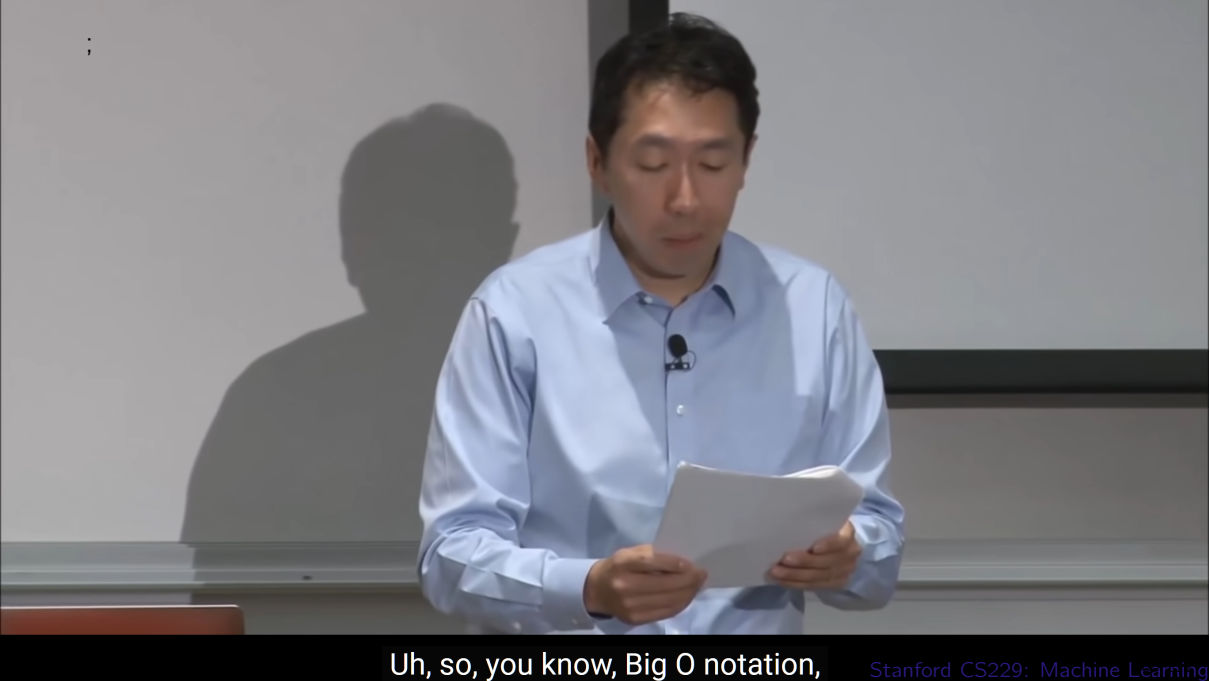


Review of Lecture 2

- Why worst-case analysis?
- The running time of an algorithm is *not* the physical time, but #steps.
- Assumptions.
- We only care about the trend of running time: when n is large.
- So we use the Big-Oh notation.

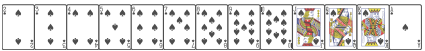
the pot calling the kettle black



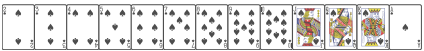

;
Uh, so, you know, Big O notation,
Stanford CS229: Machine Learning

Learning objectives

- Arrays and adjustable arrays.
- The fundamentals of design of data structures.
- The concept of stacks, and its implementation.
- The use of stacks.



Searching

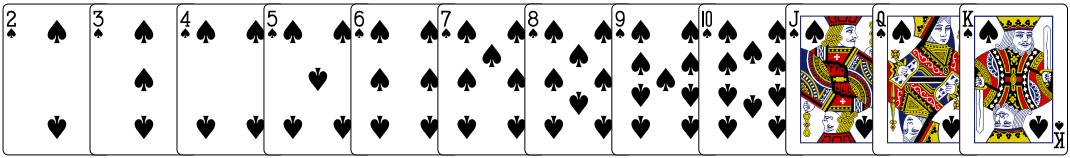


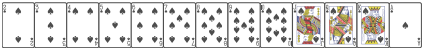
Raise your hand if you've this experience

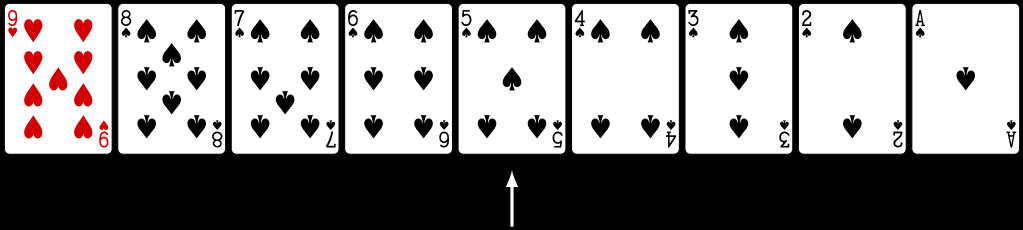
A guessing game

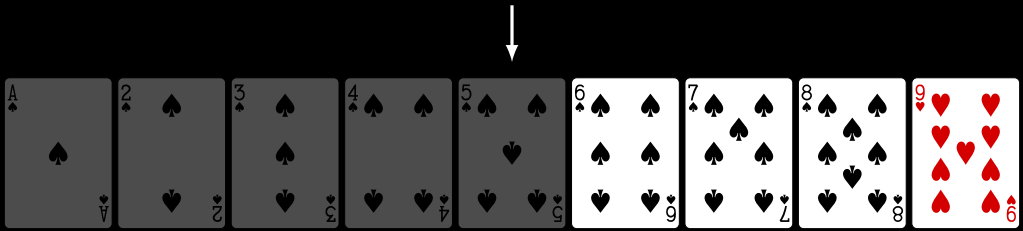
Guess which card I'm holding.
Each student have only one chance.

I'll tell you whether my card is larger/smaller than your guess.









$T(n)$: # guesses in a range of n numbers.

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3, \\ T(\frac{n}{2}) + 1 & \text{otherwise.} \end{cases}$$

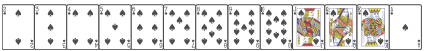
Searching

```
int linearSearch(int[] a, int k) {
   int n = a.length;
   for (int i = 0; i < n; i++)
      if (a[i] == k) return i;
   return -1;
}
```

linear search

```
int binarySearch(int[] a, int key) {
  int n = a.length;
    int low = 0, high = n - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (a[mid] == key) return mid;
      else if (a[mid] > key) high = mid - 1;
      else if (a[mid] < key) low = mid + 1;
    }
  return -1;
}
```

binary search

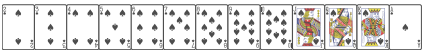


Demonstration of binary search

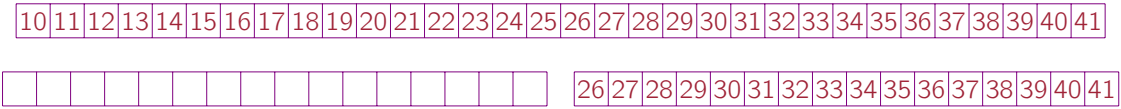
10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41

```
mid = (low + high) / 2;
```

low	high	mid	a[mid]
0	31	15	25

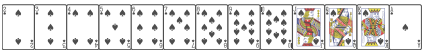


Demonstration of binary search

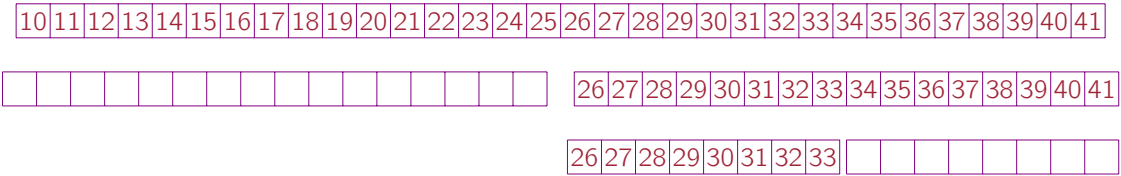



```
mid = (low + high) / 2;
```

low	high	mid	a[mid]
0	31	15	25
16	31	23	33

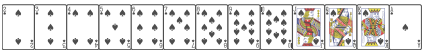


Demonstration of binary search

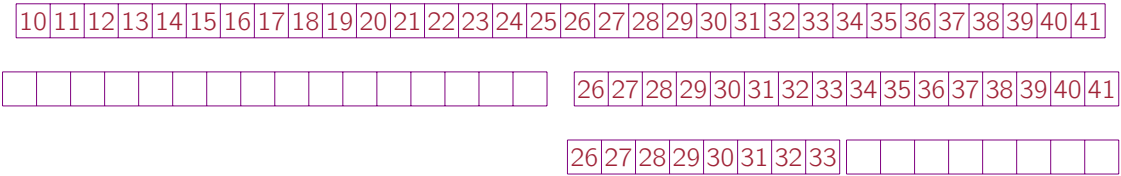



```
mid = (low + high) / 2;
```

low	high	mid	a[mid]
0	31	15	25
16	31	23	33
16	22	19	29

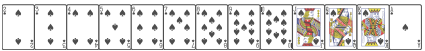


Demonstration of binary search

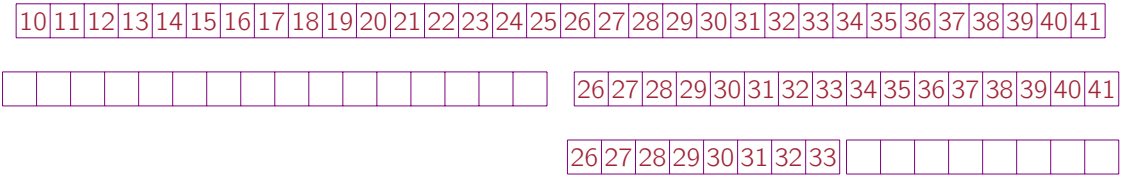



```
mid = (low + high) / 2;
```

low	high	mid	a[mid]
0	31	15	25
16	31	23	33
16	22	19	29
20	22	21	31



Demonstration of binary search



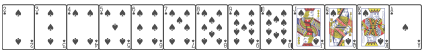

```
mid = (low + high) / 2;
```

low	high	mid	a[mid]
0	31	15	25
16	31	23	33
16	22	19	29
20	22	21	31
22	22	22	32

It takes only one iteration to find 25,
and two iterations to find 33.

Which element can also be found with two iterations?

What's the maximum number of iterations?

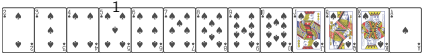


Search in the array [1, 2, 4, 8].

Summary of searching

- $\log n$ is (roughly) the number of times you can divide n by 2 before the result is less than 1.
  $2011 \to 1005. \to 502. \to 251. \to 125. \to 62. \to 31. \to 15. \to 7. \to 3. \to 1. \to 0.$

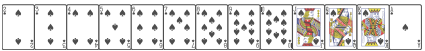
- A linear search takes $O(n)$ time (exactly n steps).
- A binary search takes $O(\log n)$ time ($\lfloor \log n \rfloor + 1 = \lceil \log(n+1) \rceil$ steps[1]).
- A binary search can only be applied to an ordered array.
- Unordered arrays offer fast insertion but slow searching.
- Arrays are always slow in deletion, sorted or not.



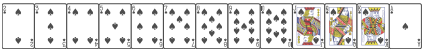
[1] You don't need to memorize these exact numbers.

Trade-offs

- time vs. space
  - HK is small, so lots of efforts are used to save space.
  - similar for computing, e.g., compressed files.
- time of one operation vs. time of another
  - e.g., insertion and finding the least element.
  - an array is maintained sorted or unsorted?



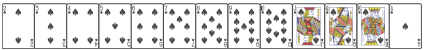
Arrays



Arrays

Array: A collection of elements of the same type stored consecutively in the memory.

- the simplest and most used data structure;
- built into all programming languages;
- operations an array should provide?
  1. add an element,
  2. delete an element,
  3. search for an element, and
  4. find the smallest/largest elements.
- Big difference between sorted and unsorted arrays.

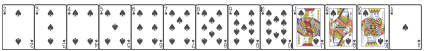


Arrays

Array: A collection of elements of the same type stored consecutively in the memory.

All strengths and weaknesses of arrays come from the consecutive storage.

- the simplest and most used data structure;
- built into all programming languages;
- operations an array should provide?
  1. add an element,
  2. delete an element,
  3. search for an element, and
  4. find the smallest/largest elements.
- Big difference between sorted and unsorted arrays.



Insertion and deletion

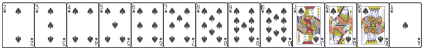
259 988 160 · · · · · · · · ·

RosterUnsorted

122	712	608	857					

RosterSorted

122	608	712	857					



Insertion and deletion

To insert into an unsorted array, simply put it at the end.

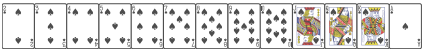
259 988 160 .....

RosterUnsorted

122	712	608	857	259				

RosterSorted

122	608	712	857					



Insertion and deletion

To insert into an unsorted array, simply put it at the end.

259 988 160 .....

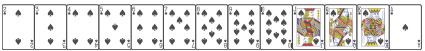
RosterUnsorted

122	712	608	857	259				

RosterSorted

122	608	712	857					

To insert into a sorted array, we need to find the correct location for the new element.



Insertion and deletion

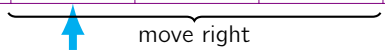
259 988 160 .....

RosterUnsorted

122	712	608	857	259				

RosterSorted

122	608	712	857					



Insertion and deletion

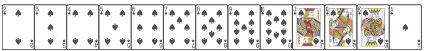
259 988 160 .....

RosterUnsorted

122	712	608	857	259				

RosterSorted

122		608	712	857				



Insertion and deletion

259 988 160 .....

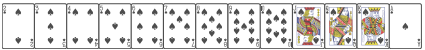
RosterUnsorted

122	712	608	857	259				

RosterSorted

122	259	608	712	857				

This is exactly the insertion sort.



Insertion and deletion

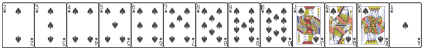
988 160 609 · · · · · · · · ·

RosterUnsorted

122	712	608	857	259				

RosterSorted

122	259	608	712	857				



Insertion and deletion

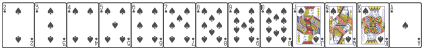
988 160 609 · · · · · · · · ·

RosterUnsorted

122	712	608	857	259	988			

RosterSorted

122	259	608	712	857				



Insertion and deletion

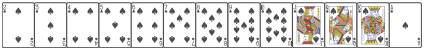
988 160 609 · · · · · · · · ·

RosterUnsorted

122	712	608	857	259	988			

RosterSorted

122	259	608	712	857	988			



Insertion and deletion

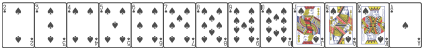
160 609 458 .....

RosterUnsorted

122	712	608	857	259	988			

RosterSorted

122	259	608	712	857	988			



Insertion and deletion

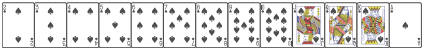
RosterUnsorted

122	712	608	857	259	988			

RosterSorted

122	259	608	712	857	988			

What if 857 decides to drop?...



Insertion and deletion

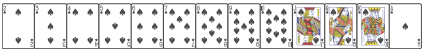
RosterUnsorted

122	712	608	857	259	988			

RosterSorted

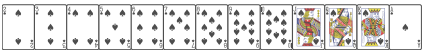
122	259	608	712	857	988			

What if 857 decides to drop?...

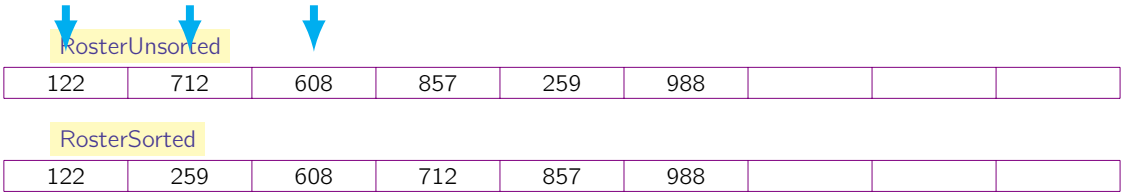


Insertion and deletion

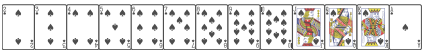
What if 857 decides to drop?...



Insertion and deletion

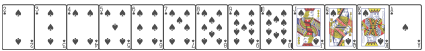


What if 857 decides to drop?...

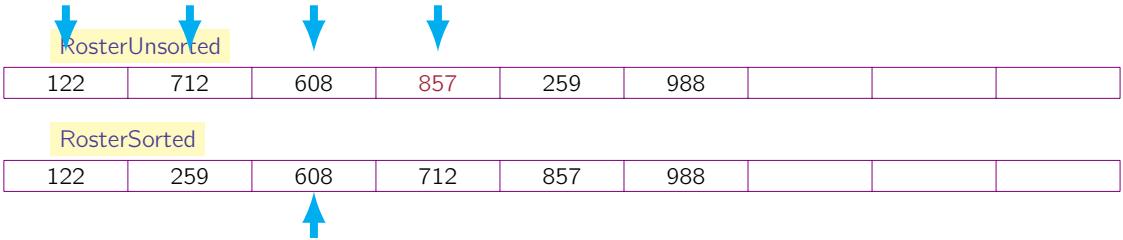


Insertion and deletion

What if 857 decides to drop?...

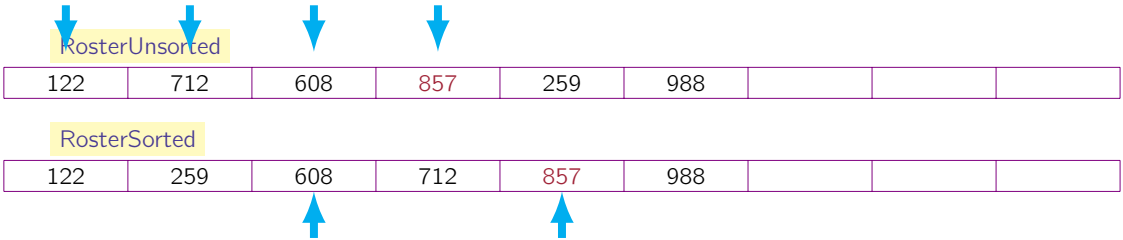


Insertion and deletion



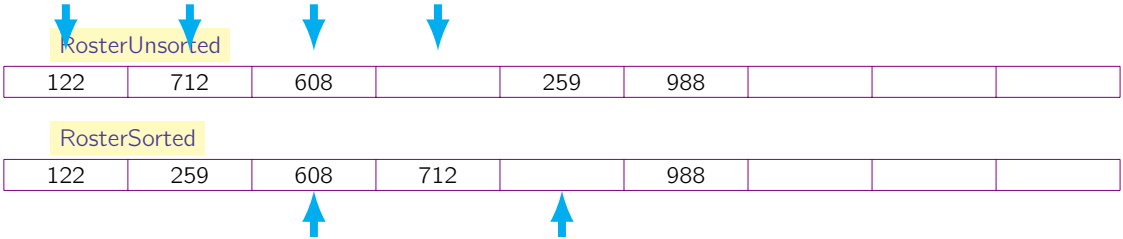
What if 857 decides to drop?...

Insertion and deletion

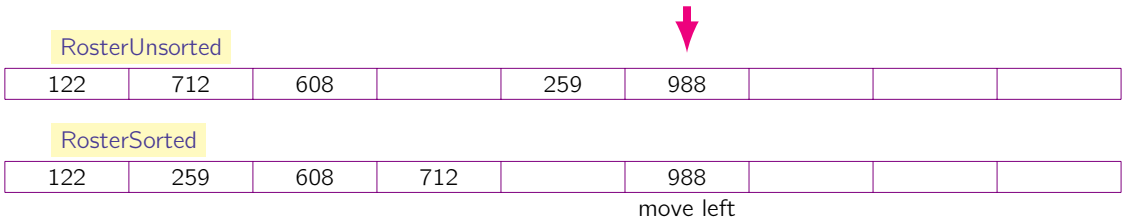


What if 857 decides to drop?...

Insertion and deletion



Insertion and deletion



Insertion and deletion

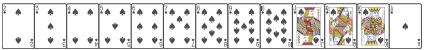
RosterUnsorted

122	712	608	988	259				

RosterSorted

122	259	608	712	988				

To insert to the sorted array, why don't we use `binary search` to find the position?



Insertion and deletion

RosterUnsorted

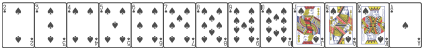
122	712	608	988	259				

RosterSorted

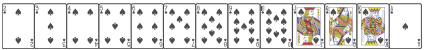
122	259	608	712	988				

Complexity (Section 3.1)?

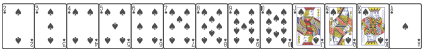
All strengths and weaknesses of arrays come from the consecutive storage.



References in Java

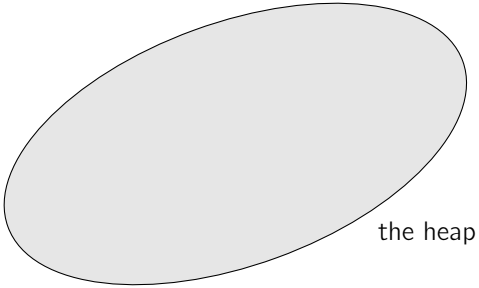


What's the difference between `int count;` and `int[] students;`?

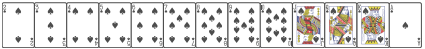


What's the difference between `int count;` and `int[] students;`?

the stack

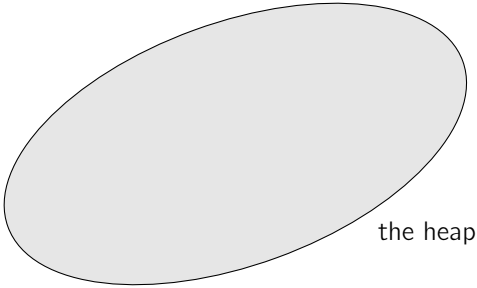


the heap

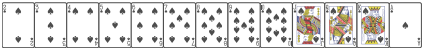


What's the difference between `int count;` and `int[] students;`?

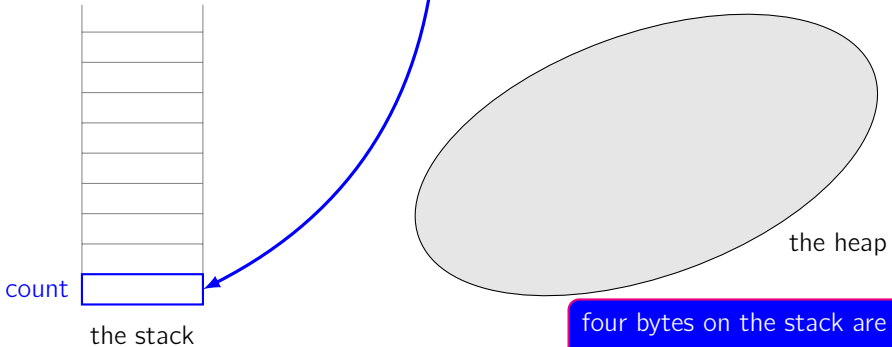
the stack



Objects and variables of primitive types are declared differently.

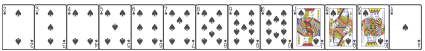


What's the difference between `int count;` and `int[] students;`?

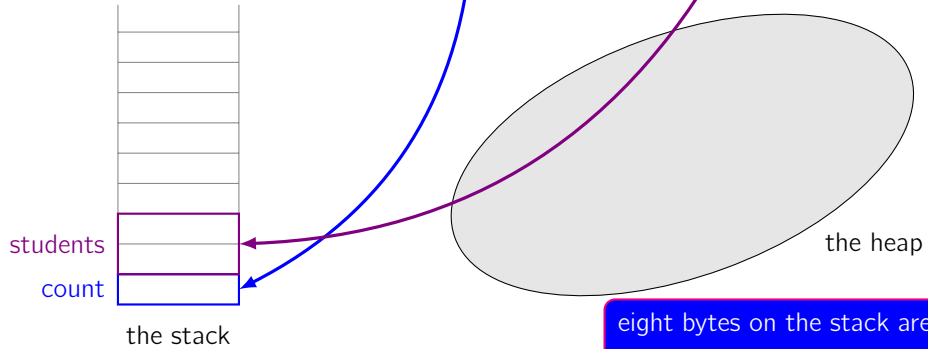


four bytes on the stack are allocated

the name `count` associated with the address of this space.

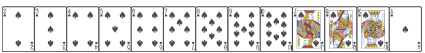


What's the difference between `int count;` and `int[] students;`?

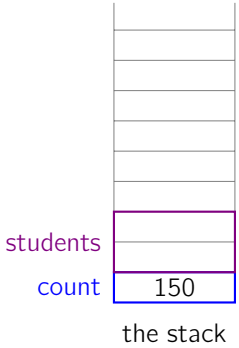


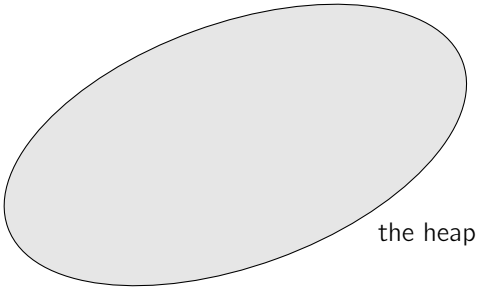
eight bytes on the stack are allocated

the name `students` associated with the address of this space.

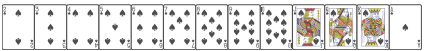


What's the difference between `int count;` and `int[] students;`?

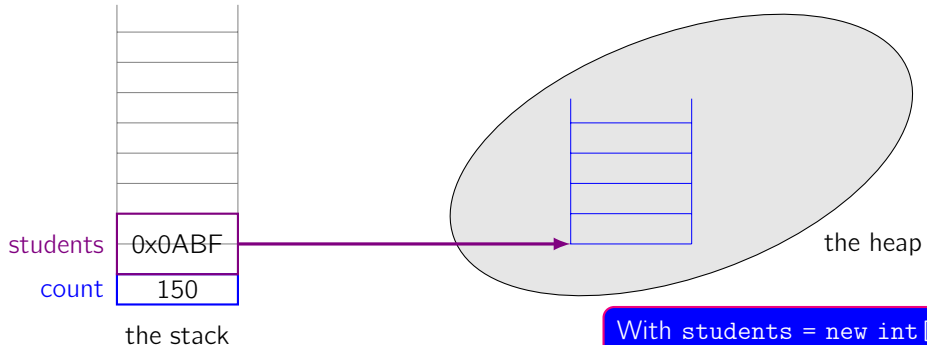




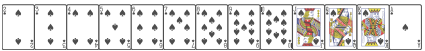
With `count = 150`
the value 150 is stored in the stack.



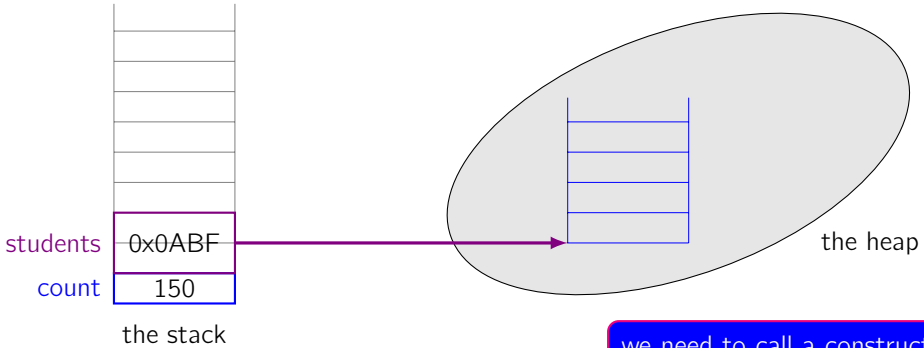
What's the difference between `int count;` and `int[] students;`?



With `students = new int[150]` 600 bytes are allocated on the heap to hold the array.

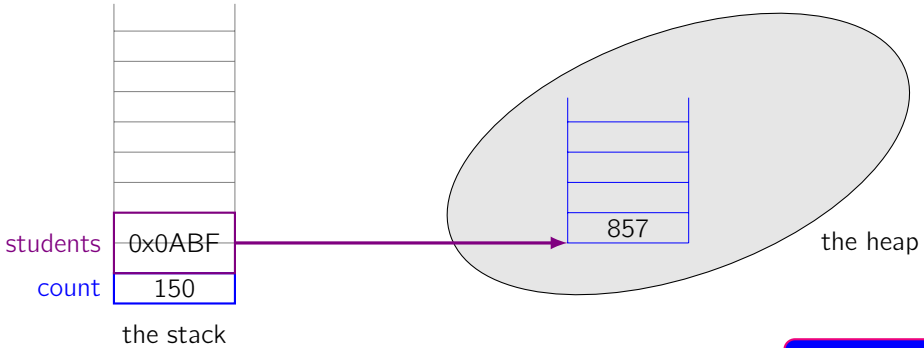


What's the difference between `int count;` and `int[] students;`?



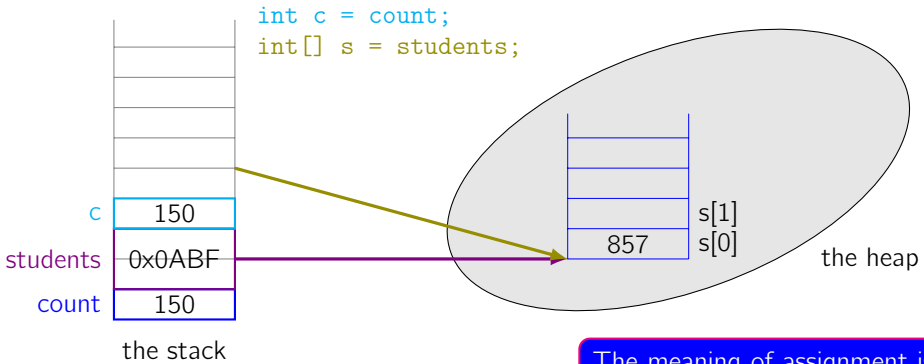
we need to call a constructor (a special method) to create an object.

What's the difference between `int count;` and `int[] students;`?



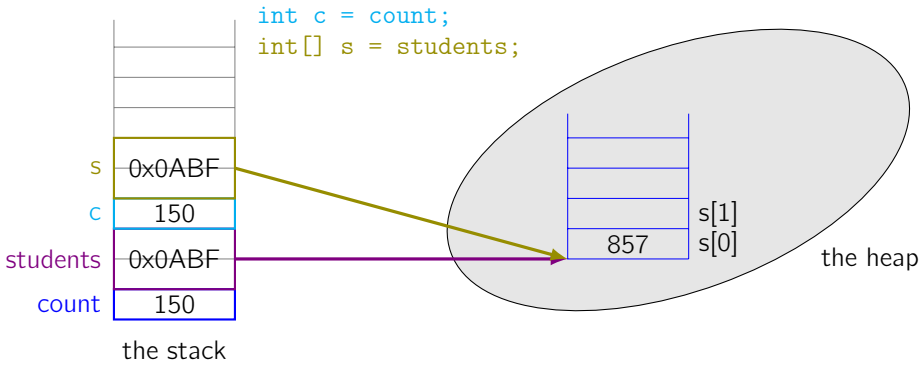
use the keyword `new`.

What's the difference between `int count;` and `int[] students;`?

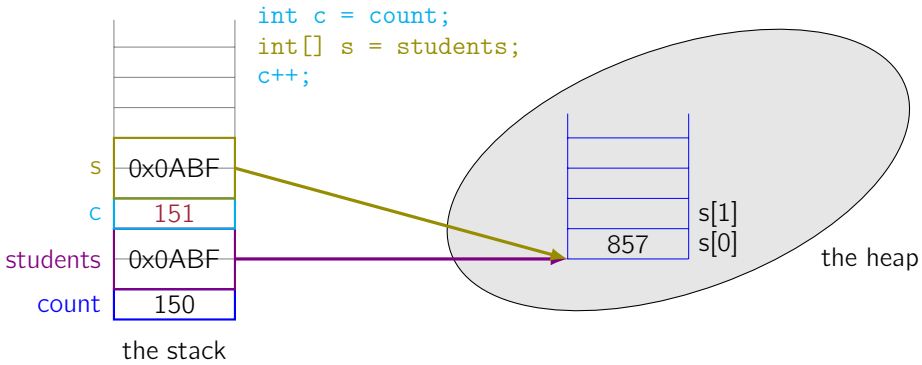


The meaning of assignment is different for objects than it is for primitive types.

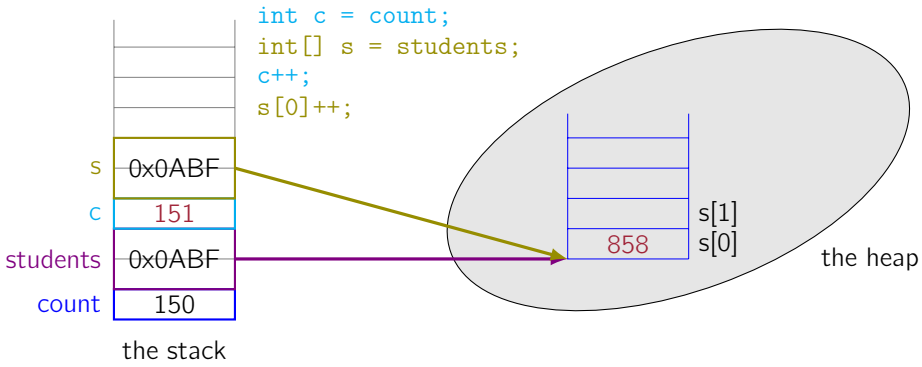
What's the difference between `int count;` and `int[] students;`?



What's the difference between `int count;` and `int[] students;`?



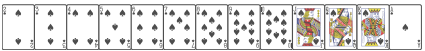
What's the difference between `int count;` and `int[] students;`?



Test

After `int[] a = new int[10];` which is wrong:

- `a[10]++;`
- `a = new int[20];`
- `a = new float[10];`

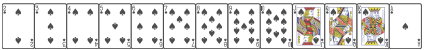


Adjustable arrays

An array has a fixed-size: it needs to be specified when initialization.

Can you build an adjustable array in Java?

The `ArrayList` in Java is exactly the array data structure we're discussing.

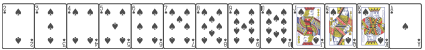


Adjustable arrays

An array has a fixed-size: it needs to be specified when initialization.

Can you build an adjustable array in Java?

The `ArrayList` in Java is exactly the array data structure we're discussing.

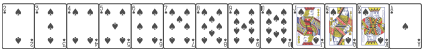


Adjustable arrays

An array has a fixed-size: it needs to be specified when initialization.

Can you build an adjustable array in Java?

The `ArrayList` in Java is exactly the array data structure we're discussing.



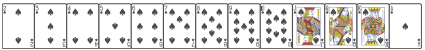
Adjustable arrays

An array has a fixed-size: it needs to be specified when initialization.

Can you build an adjustable array in Java?

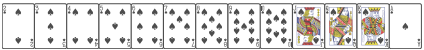
The `ArrayList` in Java is exactly the array data structure we're discussing.

All data structures are built on arrays, because ...

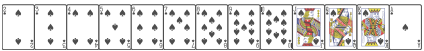


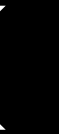
Arrays are simple and easy to use,

why do we still need other structures?



Parentheses



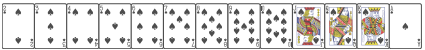


An unmatched left parenthesis
creates an unresolved tension
that will stay with you all day.

Balanced parentheses

Which are balanced:

- ( )
- ( )) (
- (( ) (( )))
- (( ))) ( )



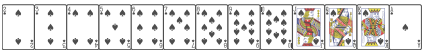
Balanced parentheses

Which are balanced:

- ( )
- ( )) (
- (( ) (( )))
- (( ))) ( )

The empty sequence is balanced.

If S and T balanced, then so are ST and (S).



Balanced parentheses

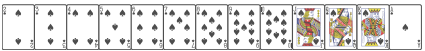
Which are balanced:

- ( )
- ( )) (
- (( ) (( )))
- (( ))) ( )

The empty sequence is balanced.

recursion is evil If S and T balanced, then so are ST and (S).

algorithmic It can be made empty by removing adjacent ().



Balanced parentheses

Which are balanced:

- ( )
- ( )) (
- (( ) (( )))
- (( ))) ( )

The empty sequence is balanced.

If S and T balanced, then so are ST and (S).

recursion is evil

It can be made empty by removing adjacent ().

algorithmic

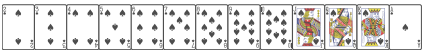
(1) There must be the same total number of (as).
(2) In any prefix the number of) cannot exceed the number of (.

```
boolean isBalanced(String s) {
  int count = 0;
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '(') count++;
    if (s.charAt(i) == ')') count--;
    if (count < 0) return false;
  }
  return count==0;
}
```

Multiple kinds of parentheses/brackets (🐦)

$$(1 \div 2 + 3) \times \big[4 + 5 \times (6 \times 7 + 8 \times 9)\big] + 10 = 2019.$$

- { [( )] }
- { ( )]
- { [ ] ([ ]) [ ] }
- { (})



Multiple kinds of parentheses/brackets (🧉)

$$(1 \div 2 + 3) \times \big[4 + 5 \times (6 \times 7 + 8 \times 9)\big] + 10 = 2019.$$

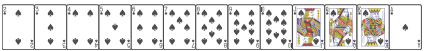
- { [( )] }
- { ( )]
- { [ ] ([ ]) [ ] }
- { (})

The empty sequence is balanced.

If S and T balanced, then so are ST and (S) and $[S]$ and $\{S\}$.

It can be made empty by removing adjacent () or [] or {}.

Can we adapt the previous algorithm to check multiple kinds of parentheses?



Multiple kinds of parentheses/brackets (🐿)

$$(1 \div 2 + 3) \times \big[4 + 5 \times (6 \times 7 + 8 \times 9)\big] + 10 = 2019.$$

- { [( )] }
- { ( )]
- { [ ] ([ ]) [ ] }
- { (})

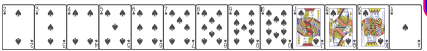
The empty sequence is balanced.

If S and T balanced, then so are ST and (S) and $[S]$ and $\{S\}$.

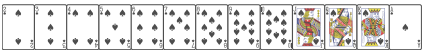
It can be made empty by removing adjacent () or [] or {}.

Can we adapt the previous algorithm to check multiple kinds of parentheses?

After seeing { and (we are expecting) and then }.



Stacks



Recursion and pancakes

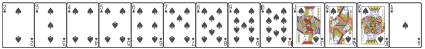
Recursion

a method that calls itself.

```
int f (int a) {
  if (a <= 1) return 1;
  else return a * f(a - 1);
}
```

Step	incomplete calls	complete calls
1	`f(4)`	
2	`f(4) f(3)`	
3	`f(4) f(3) f(2)`	
4	`f(4) f(3) f(2) f(1)`	$f(1) = 1$
5	`f(4) f(3) f(2)`	$f(2) = 2 * 1 = 2$
6	`f(4) f(3)`	$f(3) = 3 * 2 = 6$
7	`f(4)`	$f(4) = 4 * 6 = 24$

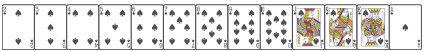
Examples of stacks



Examples of stacks

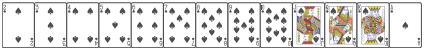
- The “back” button of a web browser
- The “undo” button of a text editor
- The most recent pending method/function call is the current one to execute.
- Evaluation of arithmetic expression (including checking the the balancedness of parentheses)
- Validation of XML documents (very similar as parentheses)

Then what operations a stack should provide?



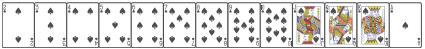
Specification of the stack

A stack's state is modeled as a sequence of elements, initially empty.



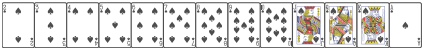
Specification of the stack

`push(Hung Hom)`
add Hung Hom to the top of the stack, and returns nothing.



Specification of the stack

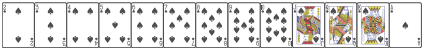

```
push(Mong Kok East)
push(Kowloon Tong)
push(Tai Wai)
```



Specification of the stack

`pop()`
remove the last element (on the top) from the stack, and returns it. Tai Wai here

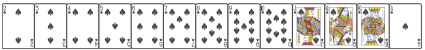
What do we get by a second `pop()`?



Specification of the stack

Last-In First-Out (LIFO)

- A `push(`x`)` operation appends x to the end of the sequence.
- A `pop` operation deletes the last element of the sequence, and returns the element that was deleted.
- An `isEmpty` operation tests if this stack is empty.
- A (nonstandard) `peek` operation looks at the object at the top.



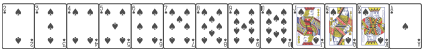
Exercises

Write the output of the following operations:

- `push(2021); push(9); push(13); push(2011); pop; pop`
- `push(2021); pop; push(9); pop; push(13); push(2011)`
- `push(2021); pop; push(9); pop; push(13); push(2011)`
- `push(2021); push(9); pop; push(13); pop; push(2011)`
- `push(2021); push(9); push(13); pop; pop; push(2011)`
- `pop; push(2021); push(9); push(13); push(2011); pop`

e.g., the first: the output is 2011, 13, and the final stack is [2021, 9

Note: 2011 is printed before 13, and [denotes the bottom of the stack.



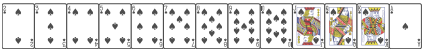
Exceptions (🔥)

Popping from an empty stack cannot be done. An error/exception is reported.

```
class CharStack {
    private char[] data;

    char pop(){
        if(isEmpty()) {
            System.out.println("Oops...");
            return '␣';
        }
        /*

        */
    }
}
```



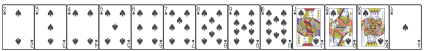

Coca-Cola
Unleash the Beast
COLA
A 1 2
B 3 4
C 5 6
D 7 8
E 9 0
F * #

Demonstration

{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.



Demonstration

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

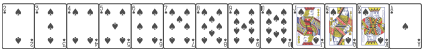
{ [ ] ([ ]) [ ] }

{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{



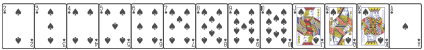
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ [



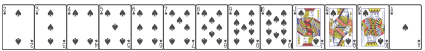
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ [



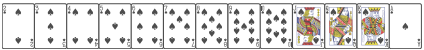
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ (



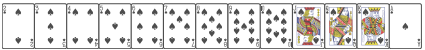
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ ([



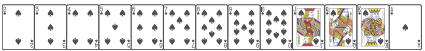
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ ([



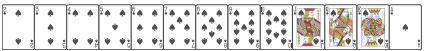
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ (



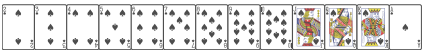
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ [



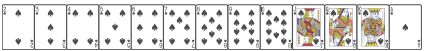
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ [



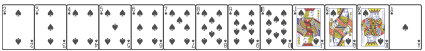
Demonstration

{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

{ so balanced



Demonstration

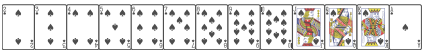
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }
{ [ ] ([ ]) [ ] }

A sequence is balanced if
it can be made empty by removing adjacent () [] {}.

- when seeing an opening bracket, push it onto the stack;
- when seeing a closing bracket, check whether it matches the top of the stack;
- at the end, check whether the stack is empty.

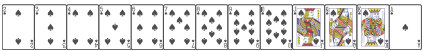
{ so balanced

try { ( )] and { (})

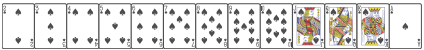


```
push(1) push(2) push(3) push(4) push(5) push(6)
pop() pop() pop() pop() pop() pop()
```

How many different outputs we can get?



Expressions



How computers handle arithmetic expressions?

$3 - 2 - 1$ and $3 - (2 - 1)$

Postfix notation, also known as Reverse Polish notation ()
The two operands come first, and then he operator.

$$3 + 4 \Rightarrow 34+$$

$$3 - 2 - 1 \Rightarrow 32 - 1-$$
$$3 - (2 - 1) \Rightarrow 321 - -$$

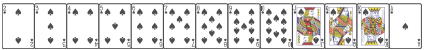
$$1 - 2 - 5 + 6 \div 5 \Rightarrow \_\_\_\_\_$$
$$(22 \div 7 + 4) \times (6 - 2) \Rightarrow \_\_\_\_\_$$
$$7 - (2 \times 3 + 5) \times (8 - 4 \div 2) \Rightarrow \_\_\_\_\_$$

Evaluate postfix expressions:

- Create an empty stack, go through the postfix expression,
  operand: push it onto the stack;
  operator: pop two operands off, do calculation, and push the result;
- return the number remaining in the stack.

The stack states after seeing each part in $32-1-$: 3; 3 2; 1; 1 1; 0.
$321--$: 3; 3 2; 3 2 1; 3 1; 2.

Generics and Abstraction



Add stability to your code by making more bugs detectable at compile time.

```
Stack stack = new Stack();
stack.add("hello");
String s = (String) stack.pop();
```

```
Stack<String> stack =
    new Stack<String>();
stack.add("hello");
String s = stack.pop();
```

a tutorial of generics

Objects

An object is an entity that has state — variables
behavior — methods

A class is the model, or pattern, from which objects are created.

Many objects can be created from the same class

The class concept supports data abstraction.

- group data that belongs together (class in Java/C++ and struct in C)
- group data together with accompanying behavior
- separate the issue of how the behavior is implemented from the issue of how it is used.

Objects

An object is an entity that has
- state — variables
- behavior — methods

A class is the model, or pattern, from which objects are created.

Many objects can be created from the same class

The class concept supports data abstraction.

- group data that belongs together (class in Java/C++ and struct in C)
- group data together with accompanying behavior
- separate the issue of how the behavior is implemented from the issue of how it is used.

Specification vs. Implementation

Week 4: Queues and Linked Lists