Class 25 Notes

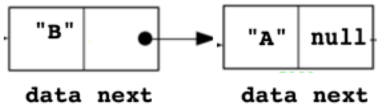
Weimplemented a stack using an array and an ArrayList.
Here is another way to implement a stack. But first, we need a new concept:
a **Node** and a **chain of Nodes**.

This is new and a little tricky
You may have to read this stuff several times**.** **It is a very important CS concept.**

Let's start with a definition of a **Node**
Definition:
A Node is an **object** with two fields

1. A data field called *data*
2. A reference field called *next*

The reference field holds the address of another Node (or null)…what?????
Here is a picture. Suppose the data is a String. Here are two nodes:



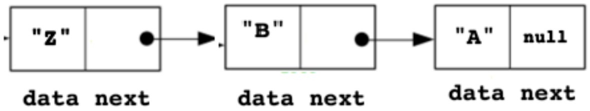
In the first node

- the data is "B" and
- the reference field (*next*) holds the address of the second node. That is , *next* "points to" or references another node.

The second node has

- data "A" and
- reference field, *next*, has the value null – it does not point to anything.

Here are three nodes "linked" together:



The *next* field of each holds the address of another node, i.e. the "next" node on the list
Notice

- the next field of first node holds the address of the second
- and the next field of the second node holds the address of the third.
- And the next field of the third node is null

So, we link or chain nodes together as in the picture above

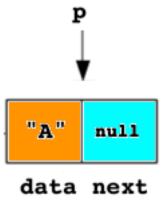
Here is a Node class. **For now, I will make the fields public .**
Assume that the data is a String.

There are only constructors, no other methods.

```
public class Node
{
    public String data;
    public Node next;  // next is the address of another node (or null)

     public Node()  // default constructor
      {
         data = “”;
         next = null;
       }
       public Node (String s)  // one argument constructor, sets data to s
       {
         data = s;
         next = null;
       }
    }
```

Now let’s look at some code segments that manipulate nodes

```
Node p = new Node (“A”);    // Creates a Node and stores its address in p
```



So the “new” operator created a new Node with data “A” The address of that Node is stored in p. Note—and this is important—

- **p is the address of the “whole” node**
  (It is a hexadecimal memory address such as 7e9f5cc and not important to us)
- **p.data is the data field (so p.data is “A”) – the orange field (above)**
- **p.next is the reference field. Her p.next is null – (The blue field (above)**

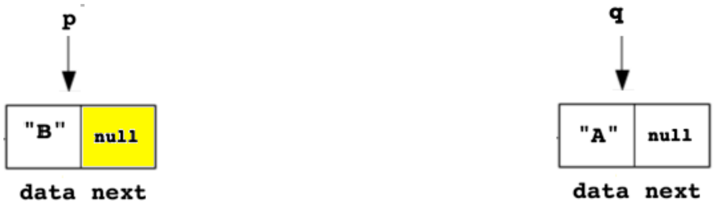
Now let’s look at the following statements:

1. Node p = new Node(“B”);
2. Node q = new Node (“A”);
3. p.next = q;

Look at the first two statements:

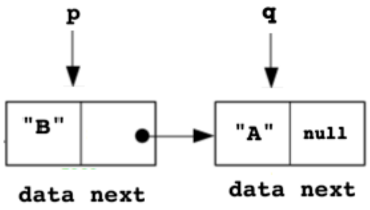
1. Node p = new Node(“B”);
2. Node q = new Node (“A”);

These two statements just create two separate nodes. p holds the address of one and q the other. (p and q are hexadecimal numbers…memory locations)



Now look at the third statement:
p.next = q;

p.next is the reference field in the first node (yellow)
q is the address of the second node
p.next = q puts the address of the second node (q) into the *next* field of the first
Wow! Look at a picture:



The next field of the first node, p.next, holds the address of the second q
The nodes are linked together.
I wrote a small program with the above 3 lines and some print statements.
Here is the output

p is Node@7e9f5cc   -- address of the “whole” node, p
q is **Node@7e9b59a2**   **--** address of the “whole” node, q

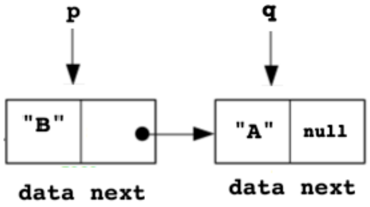
p.data is B
q.data is A

p.next is **Node@7e9b59a2** **-- this is the same as q , since p.next = q**
q.next is null

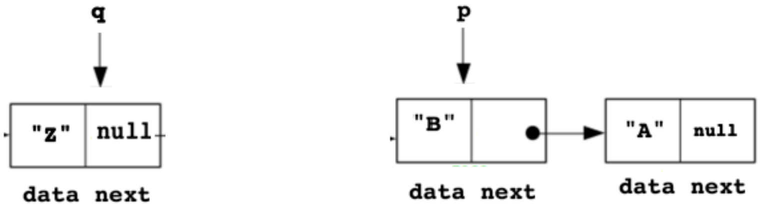
Now suppose I add these (4 and 5) two lines to the three previous lines:

1. Node p = new Node(“B”);
2. Node q = new Node (“A”);
3. p.next = q;
4. **q = new Node(“Z”);**
5. **q.next = p;**

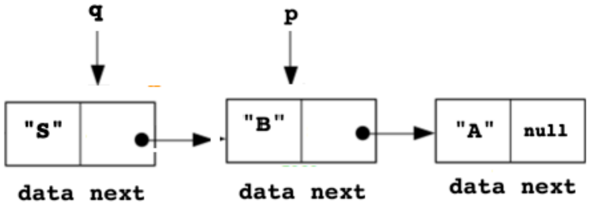
The first three lines give us the picture we had above:



Line 4 makes a new node and referenced by q (notice q now points to the new node)

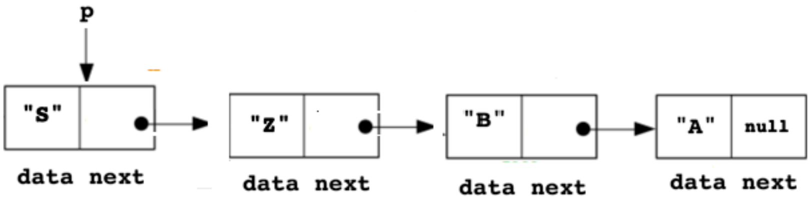


And line 5 (q.next = p) creates this chain or linked l:



Notice how we link the nodes together.

Now suppose we have a chain of nodes and p holds the address of the first node in the chain.



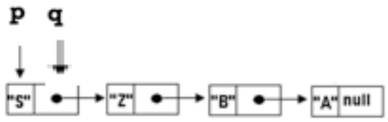
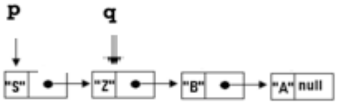
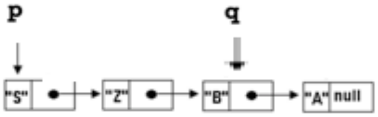
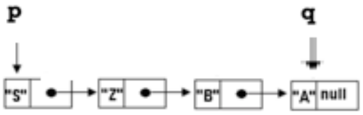
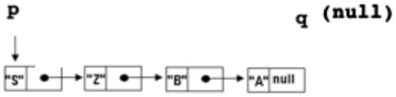
How can we access (or just print) all the data in each node of the chain?

Try to follow this short code segment:

```
Node q = p;                    //  q like p, points to the first node in the list
                               // p still holds the address of the first node

While (q != null)          // move q down the list, node by node, until it is null;
{
        System.out.println(q.data);          // print the data in Node(q)
        q = q.next;                          // move q to the next node
}
```

The important line is
q = q.next
This moves q to the next node on the list.

Here is what is happening in pictures

Node q = p;	
System.out .println(q.data)	Prints "S"
q = q.next;	 (moves q down one node)
System.out .println(q.data)	Prints "Z"
q = q.next;	
System.out .println(q.data)	Prints "B"
q = q.next;	
System.out .println(q.data)	Prints "A"
q = q.next; (q is null) and the loop stops	

Remember **q.data is the data field**; **q.next is the reference field**; q is the address of the node

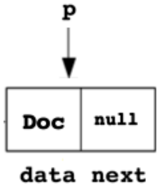
You will see this line many times**:**

q = q.next

this moves q to the next node on a list

Recap:

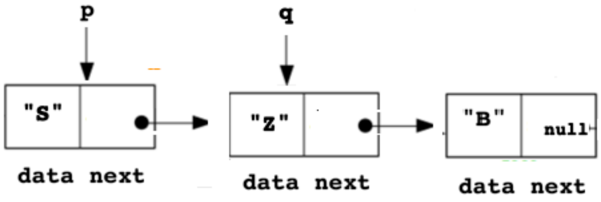
- Node p = new Node(“Doc”) creates a node and stores its address in p.



- The statement System.out.print(p) will **NOT** print “Doc” but some hexadecimal memory address such as **Node@7e9b59a2**

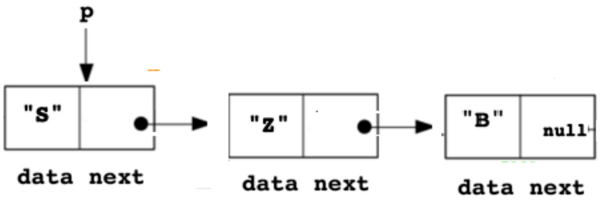
- p.data stores “Doc” → so System.out.print(p.data) prints “Doc”

- p.next stores the address of another node or null

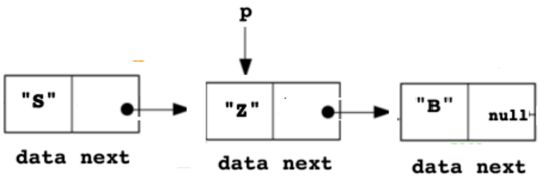


Here p.next and q have the same value. Both hold the memory address of the second node in the list

- the statement p = p.next moves pointer p to the next node in the chain
  Before



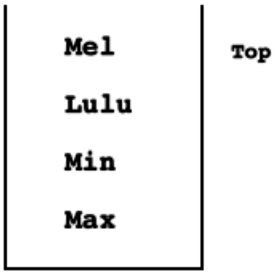
After executing p = p.next → p moves down to the next node



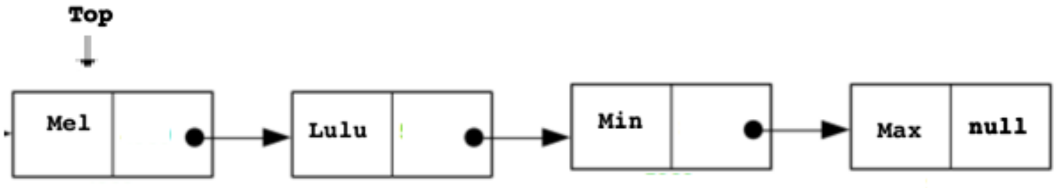
Back to Stacks

We can use a chain of Nodes to implement a stack. That is, we can store stack data in a chain of nodes (as opposed to an array or ArrayList).

For example, suppose that we have a Stack of Strings



We can implement the stack as a chain of Nodes. Notice that the variable top is a reference to the top of the stack. Top holds an address.



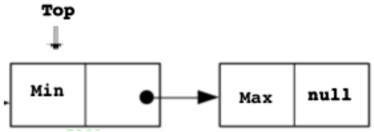
So we store the stack data in Nodes and link the nodes together.

Here is the implementation of a Stack using Nodes. The Node class (in blue) will be an inner class similar to the Listener classes.

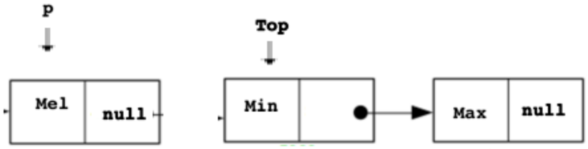
```
import java.util.*;
class Stack<E>
{
    private class Node     // this is exactly as above , just private inner class
    {
      private E data;
       private Node next;

       public Node()  // default constructor
       {
         data = null;
         next = null;
        }
       public Node(E x) // One argument constructor
       {
         data = x;
         next = null;
       }
     }  // end Node class

     private Node top;                  // a reference to the top node
     private int stackSize;             // keep track of the size of the stack


     public Stack()
                                   // default constructor; creates an empty stack
     {
         top = null;                  // initially top points to nothing
        stackSize = 0;
     }
```

I will illustrate the code for the stack operations on the right. The code is in bold.

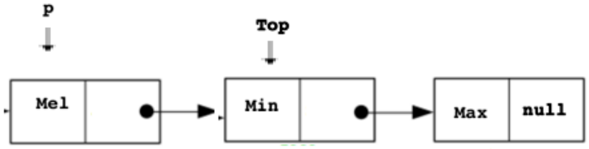
```
public void push(E x)
{
   // typical stack before the push     →
```



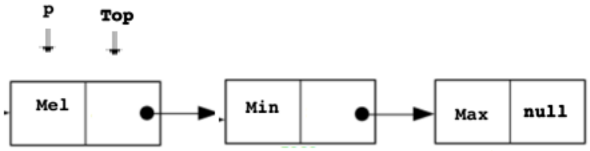
```
 // get a new node, referenced by p
   Node p = new Node(x);            //→
```



```
// connect the new node to the chain
    p.next = top;                   //→
```

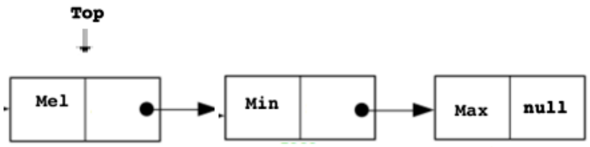


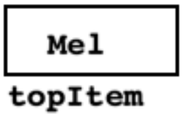
```
/ / adjust top…it is now p
   top = p;                         // →

   stackSize++;
  }
```



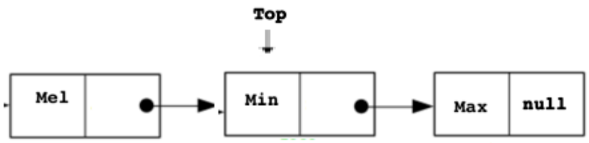
```
public E pop()
{
      //  make sure stack  is not empty
       if (stackSize == 0)
      {
         System.out.println(
                  "Stack Underflow");
         System.out.exit(0);
       }

// store  data in a temp
       E topItem = top.data;        // →
```

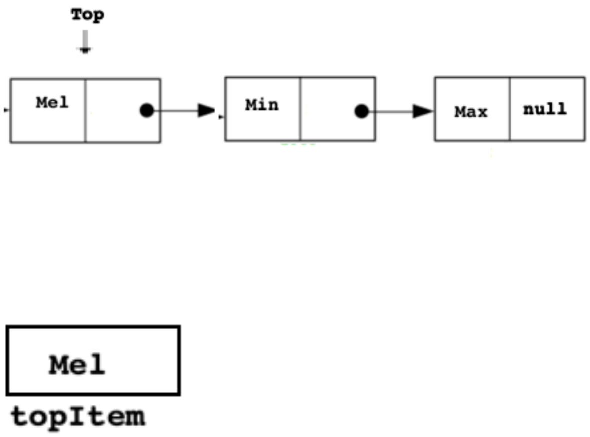
Before the pop()





```
// move top down to the next node
        top = top.next;             //  →
        stackSize--;
        return topItem;
 }
```



public E peek()
{
if (stackSize == 0)
{
System.out.println(
"Stack Underflow");
System.exit(0);
}
// get data from top node
**E topItem = top.data**;   // →
 	
return topItem;
}
	

public boolean empty()
{
return stackSize == 0;
}	

public int size()
{
return stackSize;
}
} // end class	

Very often you have to look at special cases. Will push(x) work if the original stack is empty?
In that case top is null. If you trace through the push you will see that it works in this case too.

BTW after the pop operation the garbage collector will collect the original top node.

So we have shown three ways to implement a Stack:

- Array Implementation → very efficient but the stack size is fixed. Cannot expand

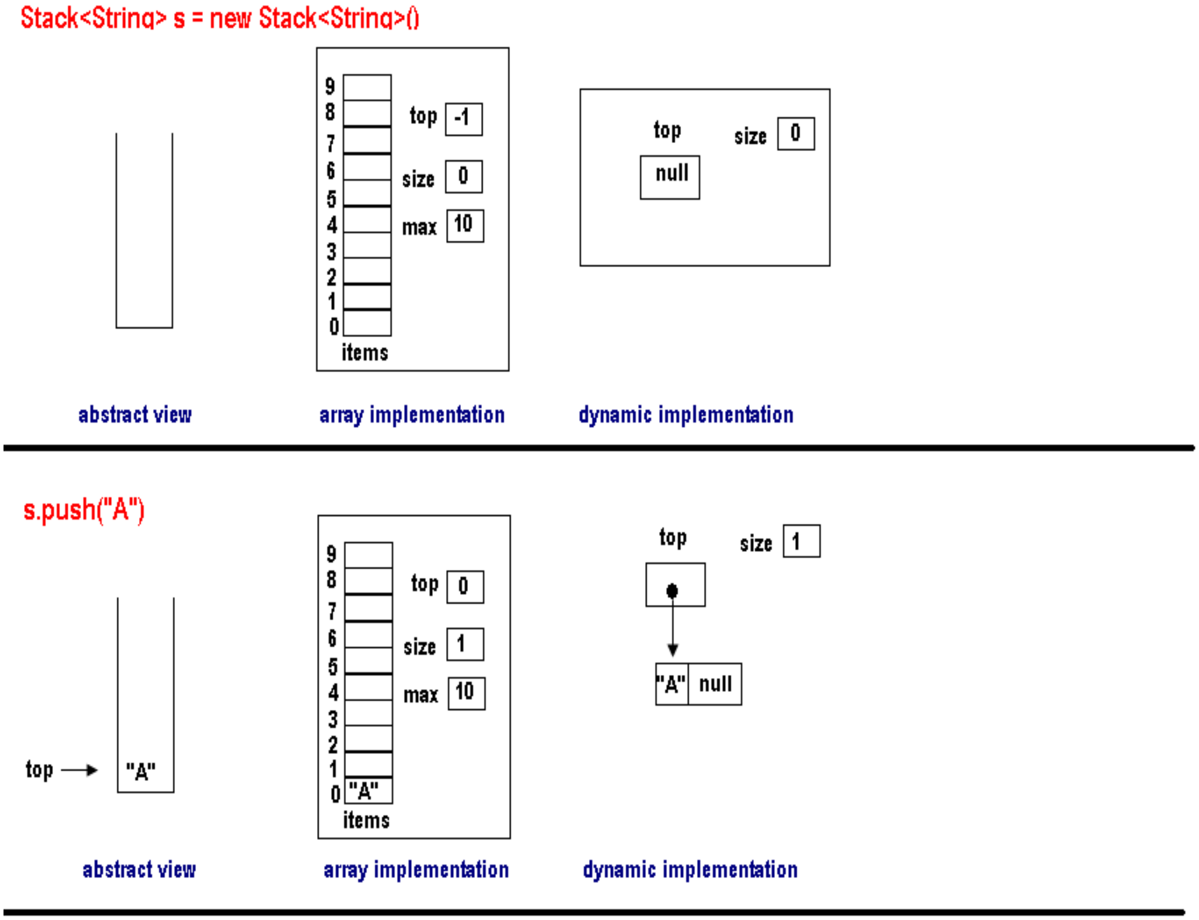
- ArrayList implementation → stack can expand as needed but that means copying over the whole ArrayList each time. So there is some time expense here

- Linked implementation → Like an ArrayList, no limit on the size. Just adds nodes as needed . No copying the whole stack when expanding. However each Node has an extra field (the reference field). So this implementation uses extra space.

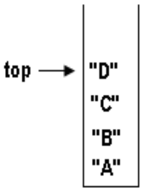
The linked implementation is also called the **dynamic implementation** because nodes are added as needed during the run of the program.

Here is a visual comparison of the array implementation and the linked implementation The code being executed is in red.

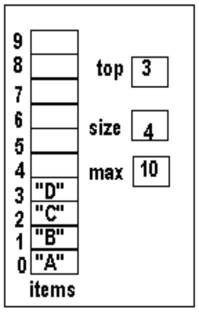
Two Stack Implementations



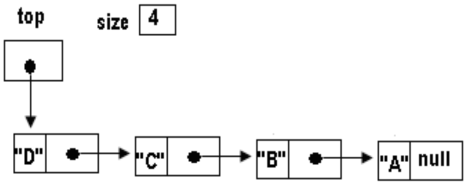
```
s.push("B"); s.push("C"); s.push("D");
```



abstract view

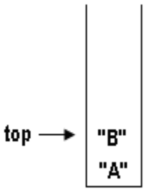


array implementation

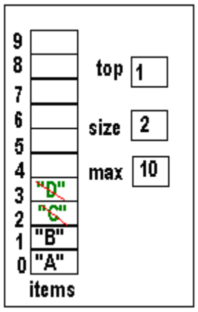


dynamic implementation

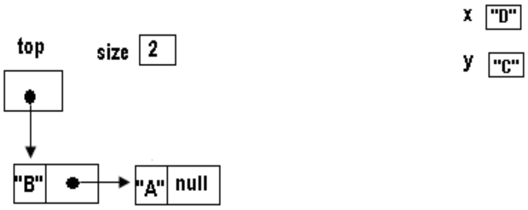
```
x = s.pop(); y = s.pop();
```



abstract view

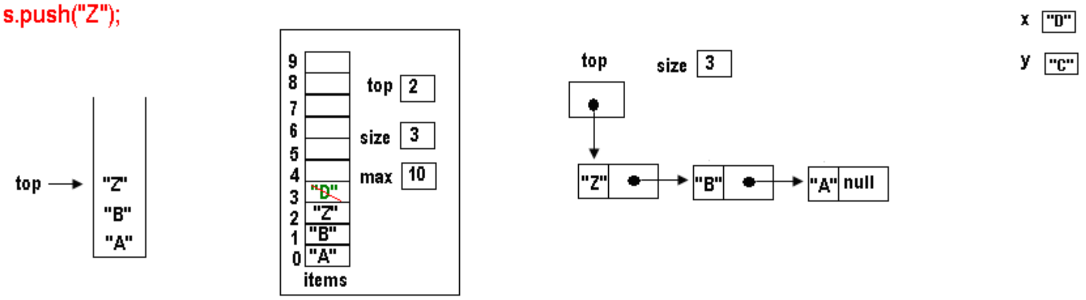


array implementation



dynamic implementation

s.push("Z");

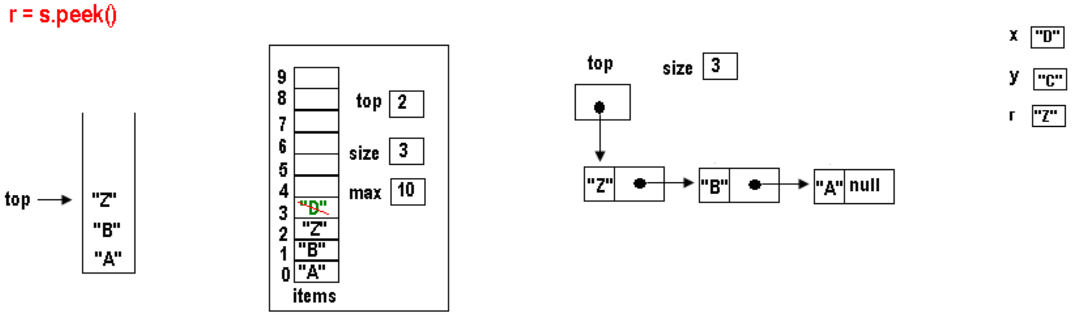


abstract view

array implementation

dynamic implementation

r = s.peek()

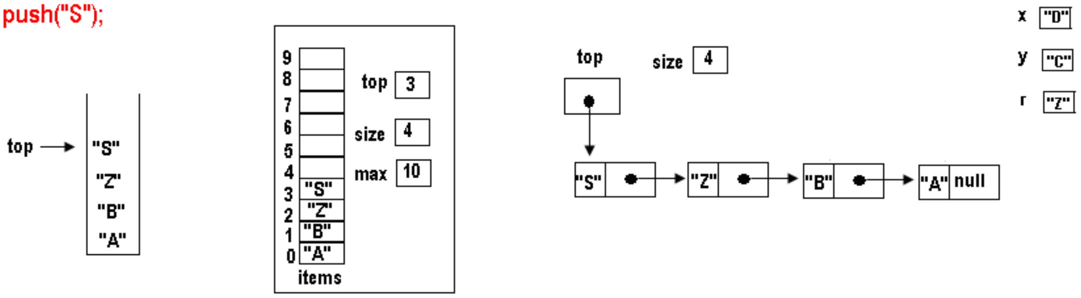


abstract view

array implementation

dynamic implementation

push("S");

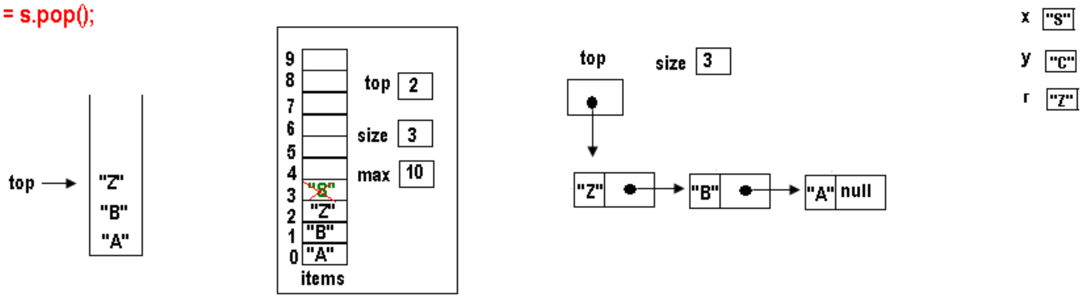


abstract view

array implementation

dynamic implementation

x = s.pop();



abstract view

array implementation

dynamic implementation