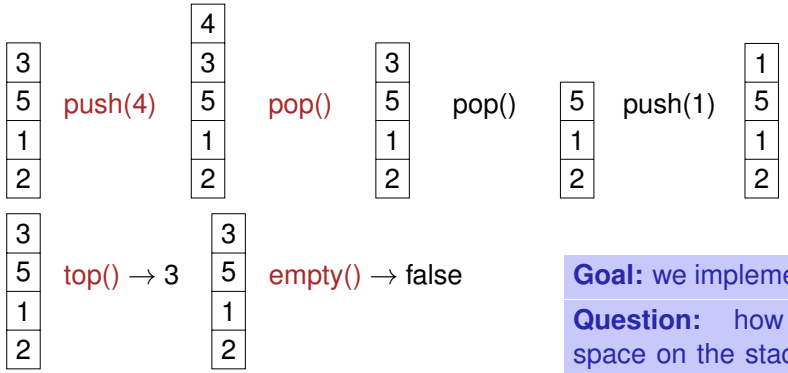
13. Dynamic Data Structures

Linked lists, Abstract data types stack, queue , Sorted List

Motivation: Stack

Motivation: Stack (`push, pop, top, empty`)



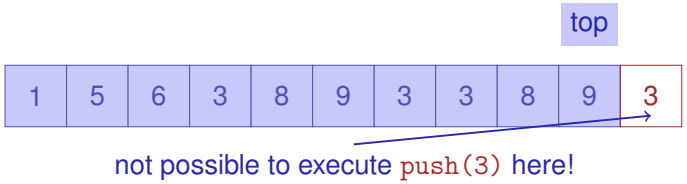
Goal: we implement a stack class

Question: how do we create space on the stack when push is called?

We Need a new Kind of Container

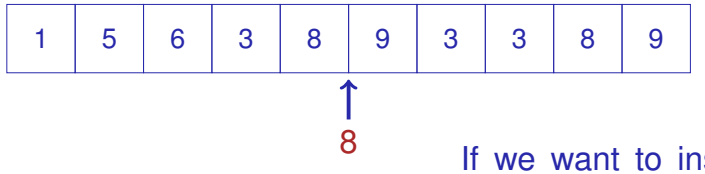
Up to this point: container = Array (`T[]`)

- Contiguous area of memory, random access (to ith element)
- Simulation of a stack with an array?
- No, at some time the array will become “full”.



Arrays are no All-Rounders...

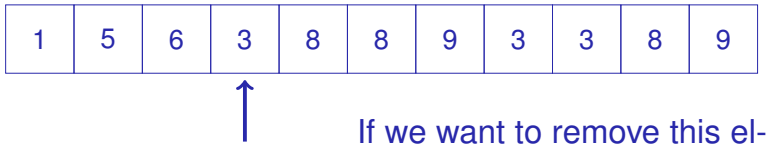
- It is expensive to insert or delete elements “in the middle ”.



If we want to insert, we have to move everything to the right (if at all there is enough space!)

Arrays are no All-Rounders...

- It is expensive to insert or delete elements “in the middle ”.

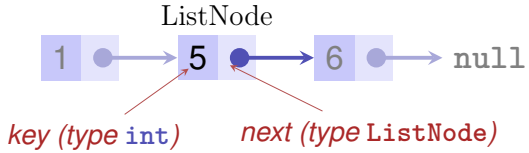


If we want to remove this element, we have to move everything to the right.

The new Container: Linked List

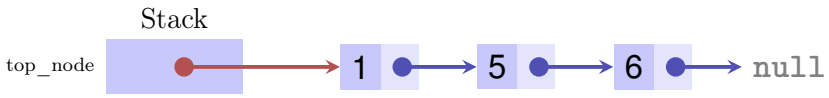
- *No* contiguous area of memory and *no* random access
- Each element “knows” its successor
- Insertion and deletion of arbitrary elements is simple, *even at the beginning of the list*
- ⇒ A stack can be implemented as linked list

Linked List: Zoom




```
class ListNode {
  int key;
  ListNode next;

  ListNode (int key, ListNode next){
    this.key = key;
    this.next = next;
  }
}
```

Stack = Reference to Top Element



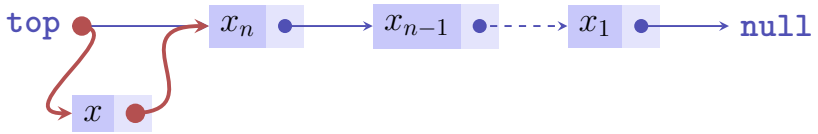

```
public class Stack {
    private ListNode top_node;

    public void push (int value) {...}
};
```


Abstract Data Types

A *stack* is an abstract data type (ADT) with operations

- `push`(x, S): Puts element x on the stack S.
- `pop`(S): Removes and returns top most element of S or `null` (or error message)
- `top`(S): Returns top most element of S or `null` (or error message).
- `empty`(S): Returns `true` if stack is empty, `false` otherwise.
- `emptyStack`$()$: Returns an empty stack.

Implementation Push



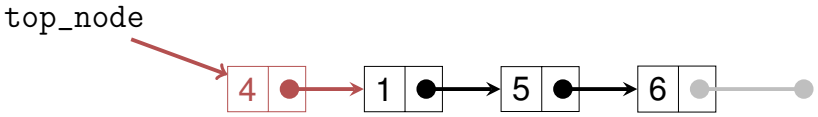
`push`(x, S):

1. Create new list element with x and pointer to the value of `top`.
2. Assign the node with x to `top`.

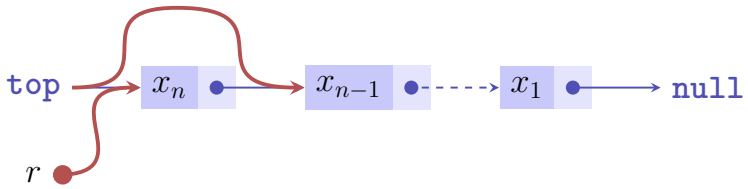
Implementation Push in Java

```
class Stack{
ListNode top_node;
  void push (int value){
    top_node = new ListNode (value, top_node);
  }
}
```

```
push(4);
```



Implementation Pop

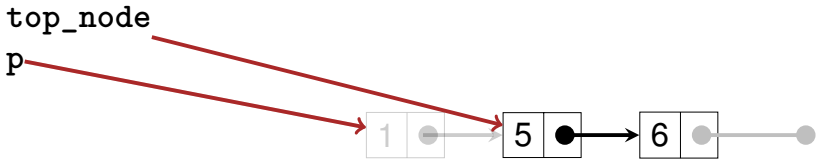


pop(S):

1. If top=null, then return null, or emit error message
2. otherwise memorize pointer p of top in r.
3. Set top to $p.next$ and return r

Implementation Pop in Java

```
int pop()
{
    assert (!empty());
    ListNode p = top_node;
    top_node = top_node.next;
    return p.key;
}
```



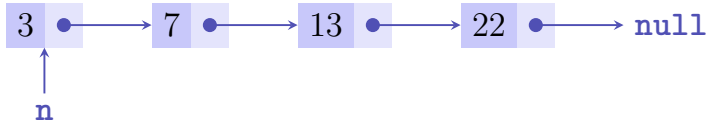
Sorted Linked List

Required Functionality:

- (Sorted) Output
- Add a value
- (Search for a value)
- Remove a value

ListNode

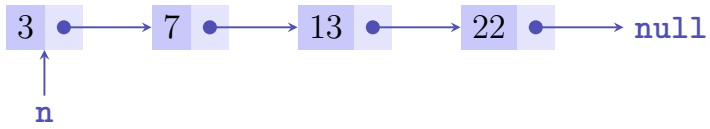
```
class ListNode{
  int key;
  ListNode next;

  ListNode (int k, ListNode n){
    key = k;
    next = n;
  }
}
```



ListNode with Getter/Setter Methods

```
public class ListNode{
    private ListNode next;
    private int key;

    ListNode (int k, ListNode nxt){
        key = k; next = nxt;
    }

    public void setNext(ListNode next){
        this.next = next;
    }

    public ListNode getNext(){
        return next;
    }

    public int getKey(){
        return key;
    }
}
```


Invariants



For a reference `n` to a node in a sorted list it holds that

- either `n` $=$ `null`,
- or `n.next` $=$ `null`
- or `n.next` $\neq$ `null` and `n.key` $\leq$ `n.next.key`.

Goal

```
public class SortedList{
  ListNode head = null;

  // insert value in a sorted way
  public void insert(int value){ ...
  }

  // remove value if in list, return if value was found in list
  public boolean remove(int value){ ...
  }

  // output list values element by element
  public void output(){ ...
  }
}
```


Output

```
// output list values element by element, starting from head
public void output(){
  ListNode n = head;
  while (n != null){
    Out.print(n.getKey() + " ");
    n = n.getNext();
  }
  Out.println();
}
```

Invariants: Insertion of x

(a) List is empty or

(b) $x \leq n.value$ for all nodes n

(c) $x > n.value$ for all nodes n

(d) There is a node n with successor m such that $x > n.value$ and $x \leq m.value$

Development of the following code live in the lecture

Insertion

```
// insert value in a sorted way (sorted increasingly by value)
public void insert(int value){
  if (head == null || value <= head.getKey()){ // (a) or (b)
    head = new ListNode(value, head);
  }
  else { // (c), (d)
    ListNode n = head;
    ListNode prev = null;
    while (n != null && value > n.getKey()){
      prev = n;
      n = n.getNext();
    }
    prev.setNext(new ListNode(value, n));
  }
}
```


Combine

```
// insert value in a sorted way (sorted increasingly by value)
public void insert(int value){
  ListNode n = head;
  ListNode prev = null;
  while (n != null && value > n.getKey()){
    prev = n;
    n = n.getNext();
  }
  if (prev == null){
    head = new ListNode(value, n);
  } else {
    prev.setNext(new ListNode(value,n));
  }
}
```


Invariants: Deletion of x

(a) x is not contained

(b) x is the first element (head)

(c) x has a predecessor

Removal

```
public boolean remove(int value){
  ListNode n = head;
  ListNode prev = null;
  while (n != null && value != n.getKey()) {
    prev = n; n = n.getNext();
  }
  if (n == null) { // (a)
    return false; }
  if (prev == null){ // (b)
    head = head.getNext();
  } else { // (c)
    prev.setNext(n.getNext());
  }
  return true;
}
```


Queue (FIFO)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x, `null` (or error message) otherwise
- `empty`(Q): return `true` if the queue is empty, otherwise `false`