16. Dynamic Data Structures

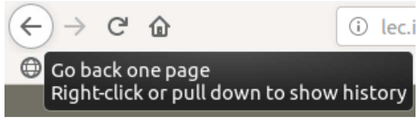
Linked lists, Abstract data types stack, queue

Data Structures

A data structure is a particular way of *organizing data* in a computer so that it can be *used efficiently*

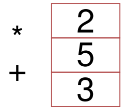
Motivation: Stack

Examples using a Stack



Go back one page
Right-click or pull down to show history

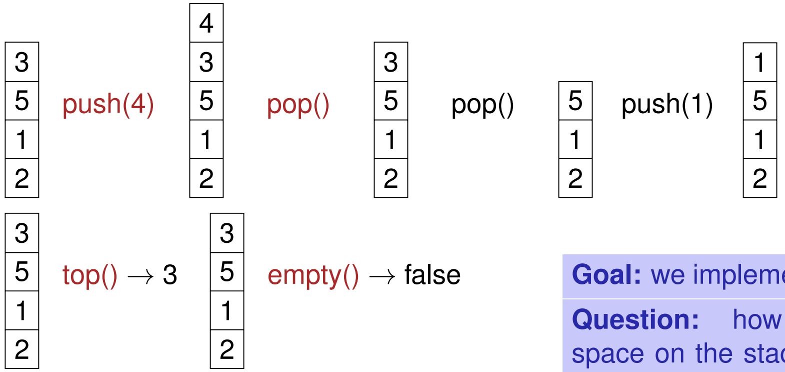
- Browsing Websites (back button)
- Undo function in a text-editor
- Calculator (using Suffix-notation)



```
3 5 2 * + = 3 + (5 * 2) = 13
```

- Suitable for introduction in a lecture like this ☺

Stack Operations (`push`, `pop`, `top`, `empty`)



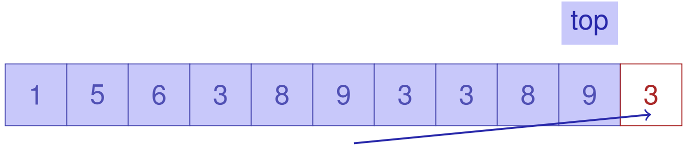
Goal: we implement a stack class

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[]`)

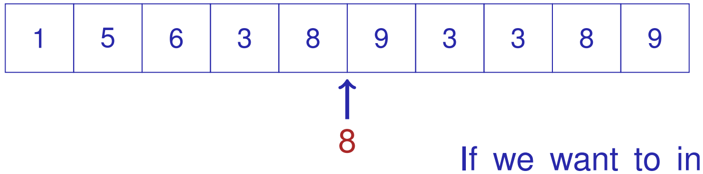
- Contiguous area of memory, random access (to ith element)
- Simulation of a stack with an array?
- No, at some time the array will become “full”.



not possible to execute `push(3)` here!

Arrays are no All-Rounders...

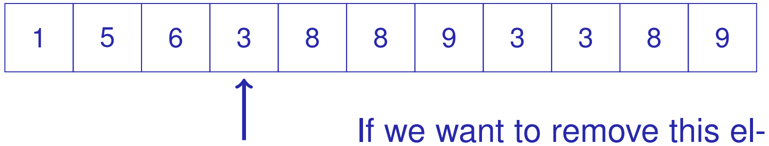
- It is expensive to insert or delete elements “in the middle ”.



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...

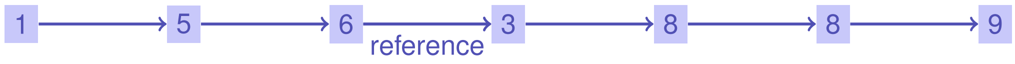
- It is expensive to insert or delete elements “in the middle ”.



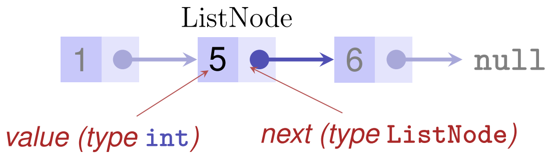
If we want to remove this element, we have to move everything to the right.

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



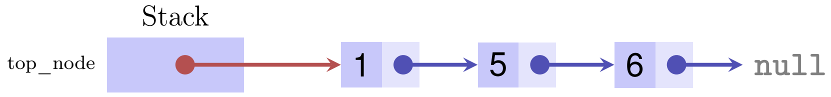
```
class ListNode {
  int value;
  ListNode next;

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

Abstract Data Types

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

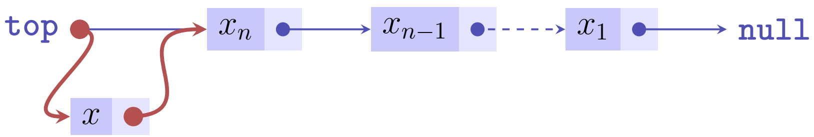
- `s.push(x)`: Puts element `x` on the stack `s`.
- `s.pop()`: Removes and returns top most element of `s` or `null` (or error message)
- `s.top()`: Returns top most element of `s` or `null` (or error message).
- `s.empty()`: Returns `true` if stack is empty, `false` otherwise.
- `new Stack()`: Returns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

Stack = Reference to Top Element



```
public class Stack {
    private ListNode top_node;
    public void push (int value) {...}
    public int pop() {...}
    public int top() {...}
    public boolean empty {...}
};
```

Implementation push



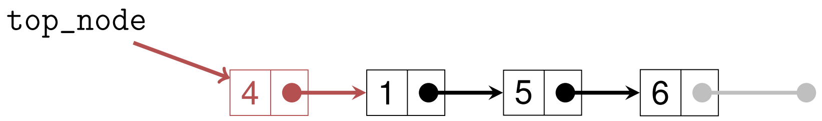
push(x):

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

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

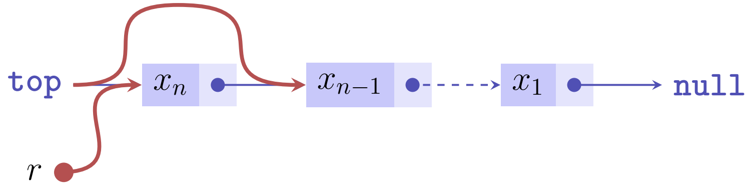
push(4);



Implementation empty in Java

```
public class Stack{
  private ListNode top_node;
  ...

  public boolean empty(){
    return top_node == null;
  }
}
```

Implementation pop

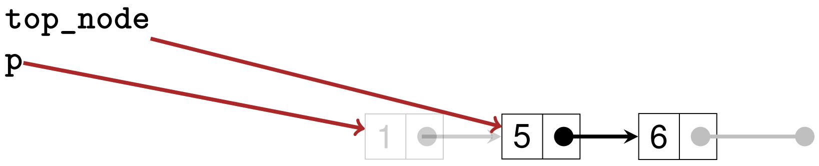


s.pop():

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

Implementation pop in Java

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


top_node

p

1

5

6

Another Example: Sorted Linked List

Required Functionality:

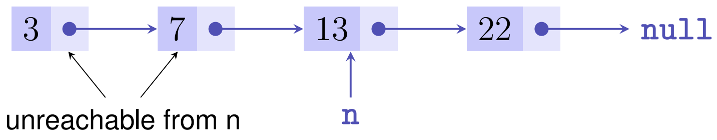
- (Sorted) Output
- Add a value
- (Search for a value)
- Remove a value

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(){ ... }
}
```

ListNode

```
class ListNode{
  int value;
  ListNode next;

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


3

7

13

22

null

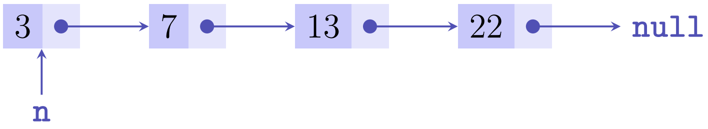
unreachable from n

n

output

```
public class SortedList{
  ListNode head = null;
  ...
  // output list values element by element, starting from head
  public void output(){
    ListNode n = head;
    while (n != null){
      Out.print(n.value + " -> ");
      n = n.next;
    }
    Out.println("NIL");
  }
}
```

Invariants



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

- either $\texttt{n} = \texttt{null}$,
- or $\texttt{n.next} = \texttt{null}$,
- or $\texttt{n.next} \neq \texttt{null}$ and $\texttt{n.value} \leq \texttt{n.next.value}$.

Invariants: Insertion of x

(a) List is empty or

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

(c) $\texttt{x} > \texttt{n.value}$ for all nodes $\texttt{n}$

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

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.value){ // (a) or (b)
    head = new ListNode(value, head);
  }
  else { // (c), (d)
    ListNode n = head;
    ListNode prev = null;
    while (n != null && value > n.value){
      prev = n;
      n = n.next;
    }
    prev.next = 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.value){
    prev = n;
    n = n.next;
  }
  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.value) {
    prev = n; n = n.next;
  }
  if (n == null) { // (a)
    return false;
  } else if (prev == null){ // (b)
    head = head.next;
  } else { // (c)
    prev.setNext(n.next);
  }
  return true;
}
```

Queue (FIFO)

A queue is an ADT with the following operations

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

First **I**n **F**irst **O**ut: Elements inserted first will be extracted first. (implementation in the exercises)