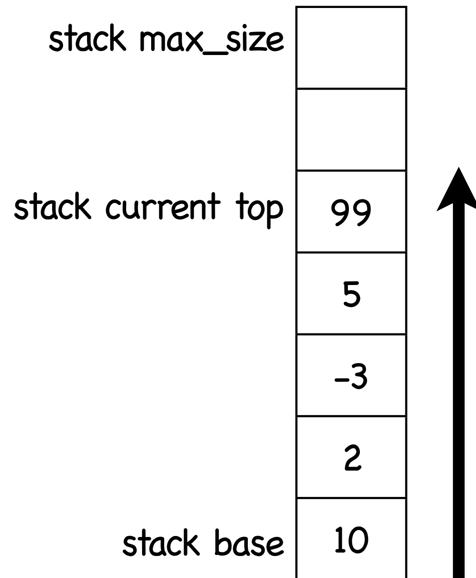
Stacks

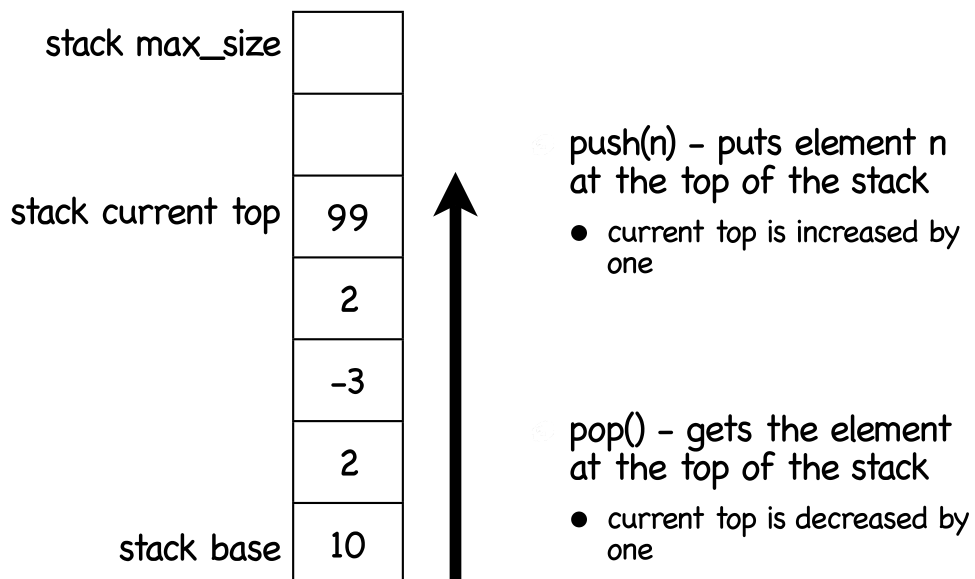
Stacks

- Stacks are first-in-last-out structures
  - or last-in-first-out
- Useful "auxiliary" structures
  - might use several stacks to solve a problem

Stacks - abstract



Stack Methods



Stacks implemented as arrays

```
class stack{
  • int* dataarray;
  • int max_size;
  • int current_top;
  • void push(int);
  • int pop();
}
```

- need to be allocated properly (constructor)
  - and freed properly (destructor)
- write up the two methods pop() and push()
  - push() might need to resize the stack

Stacks implemented as lists

```
class stack{
  • listobject* base;
  • listobject* top;
  • void push(int);
  • int pop();
}
```

```
struct listobject{
  • int value;
  • listobject next;
  • listobject prev;
}
```

- allocation easier, still need a constructor
  - and freed properly (destructor)
- two methods pop() and push() create or delete objects as they need
  - no resizing necessary

Stack Applications

- reverse order of a list/array
- function calls in a computer program
- physical towering problems (see Towers of Hanoi)
- rearranging railroad cars
- convert decimals to binary (endian rule may force reverse order)
- evaluations of non-parenthesized expressions
- memory management

Queues

Queues

- Stacks are first-in-first-out structures
  - or last-in-last-out
- Useful "auxiliary" structures
  - might use several queues to solve a problem

Queues

consumer builder max_size

10	2	-3	2	99		

→

Queue Methods

consumer builder max_size

10	2	-3	2	99		

- enqueue(n) - add/build/put_in value n to the queue
  - builder advances by one
- dequeue() - retrieves the next "unprocessed"/available/ take_out element
  - consumer advances by one

Queue implemented as arrays

```
class queue{
  int* dataarray;
  int max_size;
  int builder_index;
  int consumer_index;
  void enqueue(int n);
  int dequeue();
}
```

- need to be allocated properly (constructor)
  - and freed properly (destructor)
- write up the two methods enqueue() and dequeue()
  - enqueue() might need to resize the queue

Queue implemented as lists

```
class queue{
  • listobject* consumer;
  • listobject* builder;
  • void enqueue(int);
  • int dequeue();
}
```

```
struct listobject{
  • int value;
  • listobject next;
  • listobject prev;
}
```

- allocation easier, still need a constructor
  - and freed properly (destructor)
- two methods enqueue() and dequeue() create or delete objects as they need
  - no resizing necessary

Queue Applications

- tree traversals: nodes are stored for future processing
- process management
- printer jobs
- waiting lists
  - vehicles on tolls
  - phone answering systems
  - luggage checking
  - patients order