Stacks and Queues

ADTs and implementations

Array resizing

Abstract data types

- Abstract data type (ADT) – a mathematical description of an object and a set of operations on the object
  - Alternatively, a collection of data and the operations for accessing the data

- Example: Dictionary ADT
  - Stores pairs of strings: (word, definition)
  - Operations:
    - Insert(word, definition)
    - Remove(word)
    - Lookup(word)

Insert

- Feet
  - Useful for something, presumably

Find(Z125 Pro)

- Z125 Pro
  - Fun in the sun!

- Super 9 LC
  - Smell like a lawnmower
- Z125 Pro
  - Fun in the sun!
- CB300F
  - For the mild-mannered commuter

data storage implemented with a data structure

Implementing ADTs

Using data structures

- Theoretically (in programming languages that support OOP)
  - abstract base class describes ADT (operations etc.)
  - inherited implementations apply data structures
  - data structure can be changed transparently (to client code)
- Practice
  - performance of a data structure may influence form of client code
    - time vs space, one operation vs another

Reverse Polish Notation (RPN)

- Reverse Polish Notation (RPN)
  - Also known as postfix notation
  - A mathematical notation
    - Where every operator follows its operands
- Example
  - Infix: 5 + ((1 + 2) * 4) − 3
  - RPN: 5 1 2 + 4 * + 3 –

$$\overbrace{\overbrace{5 + ((\overbrace{\overbrace{1 + 2}^{3}) * 4}^{12}}^{17}) - 3}$$

Infix: 5 + ((1 + 2) * 4) – 3 = 14

RPN example

- 5 1 2 + 4 * + 3 –

To evaluate a postfix expression, read it from left to right

Apply '–' to last two operands

4

12

14

Retrieve result

Note: the postfix string contains integers and characters, but the data collection contains only integers

Calculating a Postfix Expression

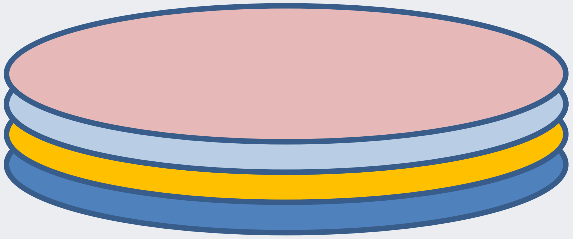
- for each input symbol
  - if symbol is operand
    - store(operand)
  - if symbol is operator
    - RHS = remove()
    - LHS = remove()
    - result = LHS operator RHS
    - store(result)
- result = remove()

store & remove
will affect the contents of
our data container in
some conceptual way

(add / remove from top
of a linear structure)

Describing an ADT

- What are the storage properties of the data type that was used?
  - Specifically how are items stored and removed?
- Note that items are never inserted between existing items
  - The last item to be entered is the first item to be removed
  - Known as LIFO (Last In First Out)
- This ADT is referred to as a *stack*



The Stack ADT

- A stack only allows items to be inserted and removed at *one end*
  - We call this end the *top* of the stack
  - The other end is called the bottom
- Access to other items in the stack is not allowed
- A stack can be used to naturally store data for postfix notation
  - Operands are stored at the top of the stack
  - And removed from the top of the stack
- Notice that we have not (yet) discussed how a stack should be implemented
  - Just *what* it does
- An example of an *Abstract Data Type*

Stack behaviour

- A stack ADT should support at least the first two of these operations:
  - **push** – insert an item at the top of the stack
  - **pop** – remove and return the top item
  - **peek** – return the top item
  - **isEmpty** – does the stack contain any items
- ADT operations should be performed efficiently
  - The definition of efficiency varies from ADT to ADT
  - The order of the items in a stack is based solely on the order in which they arrive

use a concrete data structure to implement

iClicker 07.1

LIFO behaviour

- Suppose we perform the stack operations listed at the left side. In order, what items are removed?

output/removed items: o t u s

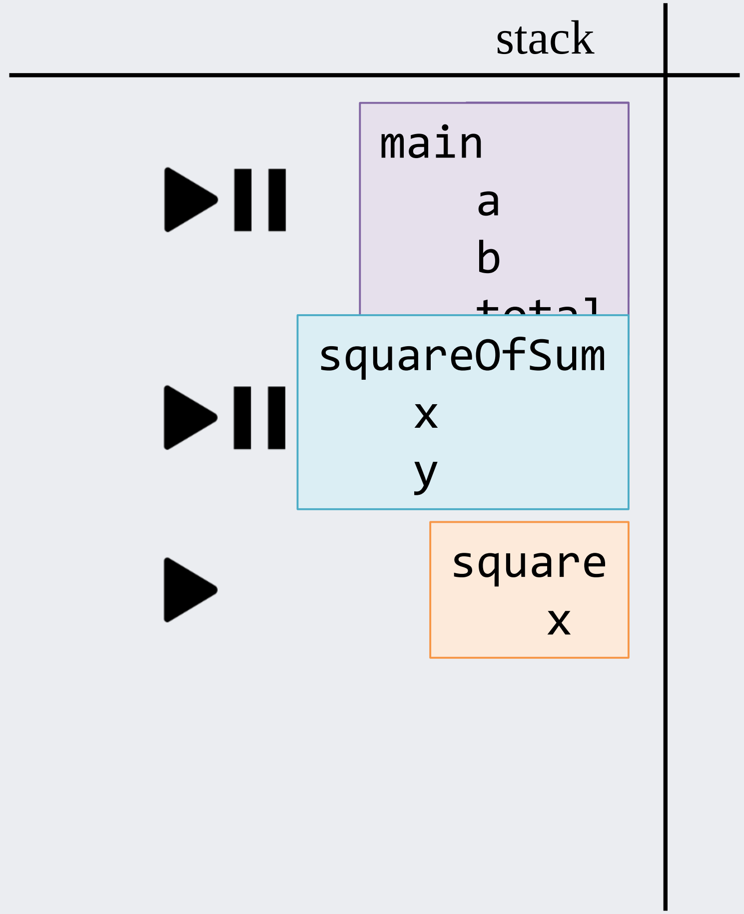
```
push(d)
push(o)
pop()
push(n)
push(u)
push(t)
pop()
pop()
push(s)
pop()
```

A. duns
B. otus
C. dtus
D. None of these, but it **can** be determined from just the ADT
E. None of these, and it **cannot** be determined from just the ADT

Stacks in the wild

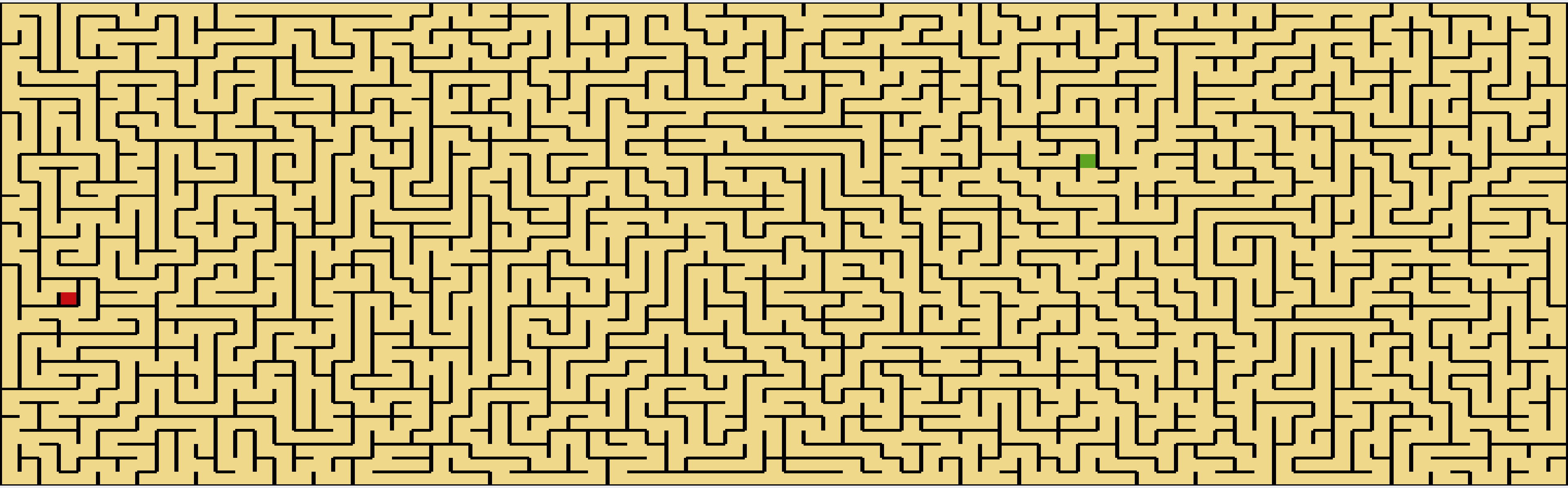
Call Stack

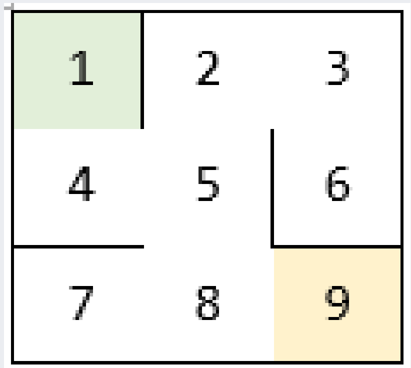
```c
int square(int x) {
  return x*x;
}

int squareOfSum(int x, int y) {
  return square(x+y);
}

int main() {
  int a = 4;
  int b = 8;
  int total = squareOfSum(a, b);
  printf("Total: %d\n", total);
  return 0;
}
```

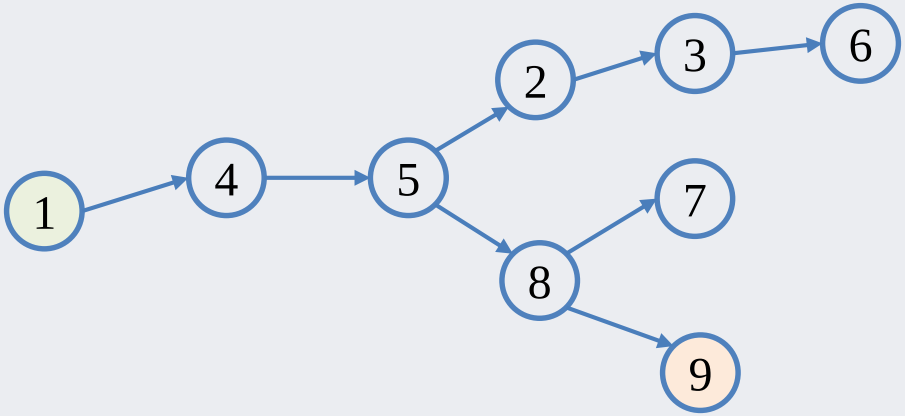


Stacks in the wild

Backtracking







Stack implementation

- The stack ADT can be implemented using a variety of data structures, e.g.
  - Arrays
  - Linked Lists
- Both implementations must implement all the stack operations
  - In constant time (time that is independent of the number of items in the stack)

Stack implementation

Using an array

- Suppose we use an array which does not contain any gaps between elements
  - We can add and remove elements at the right side, as long as we know which element is the last item
    - Treat the last element of the array as the "top" of the stack
  - Information to track:
    - index of top item
    - maximum size of the array (capacity)

Stack implementation

Using an array

```
typedef struct {
  int top; // index of last occupied space
  int capacity; // maximum size of array
  int* arr; // pointer to array (in dynamic memory)
} Stack;
```

or pointer to array of whatever type your stack holds

```
#define TRUE 1
#define FALSE 0
```

in main:

Stack mystack;

initialize(&mystack);

// ready to use

```
void initialize(Stack* st) {
  st->top = -1;
  st->capacity = INITIALCAPACITY; // or some other value
  st->arr = (int*) malloc(capacity * sizeof(int));
}
```

(also useful to have a destroy function)

```
int isEmpty(Stack* st) {
  if (st->top == -1)
    return TRUE;
  else
    return FALSE;
}
```

```
int isFull(Stack* st) {
  if (st->top == st->capacity - 1)
    return TRUE;
  else
    return FALSE;
}
```

Stack implementation

Using an array

- push – the item is inserted and becomes the new stack top
  - increment `top` before access
- However, we must ensure that the stack has space available before pushing

ADT doesn't provide any understanding of a "full" stack

```
int push(Stack* st, int value) {
  if (!isFull(st))
    st->top++;
    st->arr[st->top] = value;
    return TRUE;
  else
    return FALSE;
}
```

Stack implementation

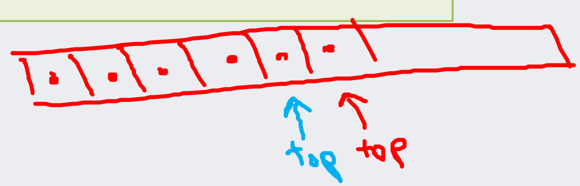
Using an array

- pop – the top item is removed and stack shrinks
  - decrement `top` after access
- However, we must ensure that the stack is not already empty

not necessary

```
int pop(Stack* st) {
  if (!isEmpty(st))
    st->arr[st->top] = -1;
    st->top--;
    return TRUE;
  else
    return FALSE;
}
```

```
int peek(Stack* st) {
  if (!isEmpty(st))
    return st->arr[st->top];
  else
    return FALSE;
}
```



Stack implementation

Using an array

- In this implementation, array is created with an initial size, and push returns false if the stack is full
  - What if we really need to push additional items?
  - Expand/reallocate a larger array
  - This should happen transparently to client code

Stack implementation

- Array resizing

```c
int push(Stack* st, int val) {
  int i; // used for reallocation
  int* newarr;
  if (st->top == st->capacity – 1) {
    // reallocate a larger array
    st->capacity = 2 * st->capacity;
    newarr = (int*) malloc(st->capacity * sizeof(int));
    for (i = 0; i <= st->top; i++)
      newarr[i] = st->arr[i];
    free(st->arr);
    st->arr = newarr;
  }
  // continue with push
  st->top++;
  st->arr[st->top] = val;
  return TRUE;
}
```

```c
#define TRUE 1
#define FALSE 0
```

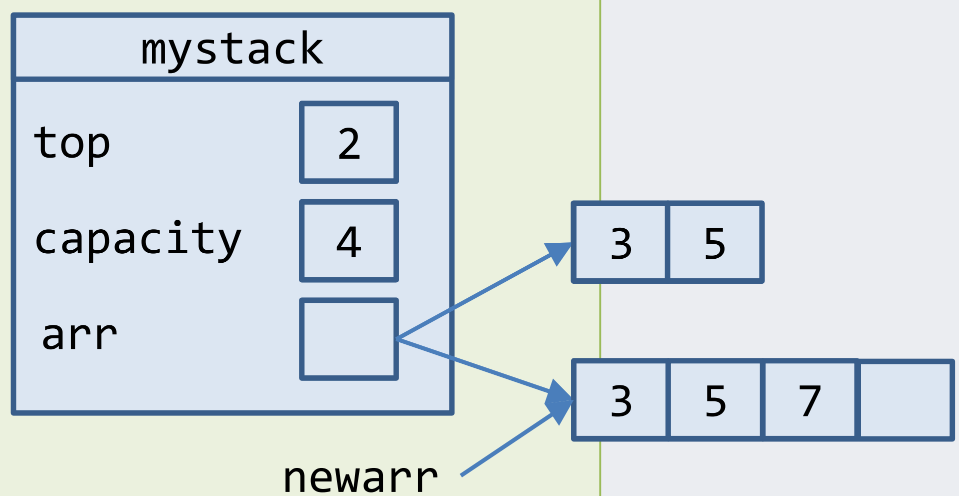
```c
typedef struct {
  int top;
  int capacity;
  int* arr;
} Stack;
```

Stack implementation

- Array resizing

```
int push(Stack* st, int val) {
  int i; // used for reallocation
  int* newarr;
  if (st->top == st->capacity - 1) {
    // reallocate a larger array
    st->capacity = 2 * st->capacity;
    newarr = (int*) malloc(st->capacity * sizeof(int));
    for (i = 0; i <= st->top; i++)
      newarr[i] = st->arr[i];
    free(st->arr);
    st->arr = newarr;
  }
  // continue with push
  st->top++;
  st->arr[st->top] = val;
  return TRUE;
}
```

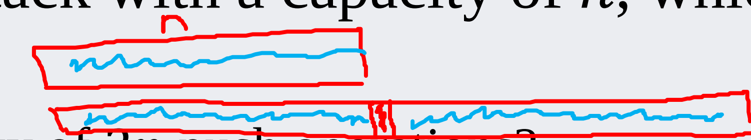
```
push(&mystack, 3);
push(&mystack, 5);
push(&mystack, 7);
```



Complexity of array resizing

- Suppose we have a stack with a capacity of n, which is completely full
  - What is the complexity of one push operation? $O(n)$

- Suppose we have a stack with a capacity of n, which is completely empty



  - What is the complexity of $2n$ push operations? $2n$

$O(1)$ each

$1 \times O(n)$

$(2n-1) \times O(1)$

$n + (2n-1) = 3n-1$

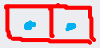
- What then is the average complexity of a single push operation? $(3n-1)/2n \in O(1)$

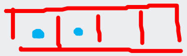
resize: capacity *= 2 ← $\text{capacity} = \lceil \text{capacity} * c \rceil$, $c > 1$

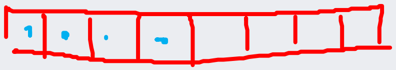
initial capacity: 1

average push $O(1)$

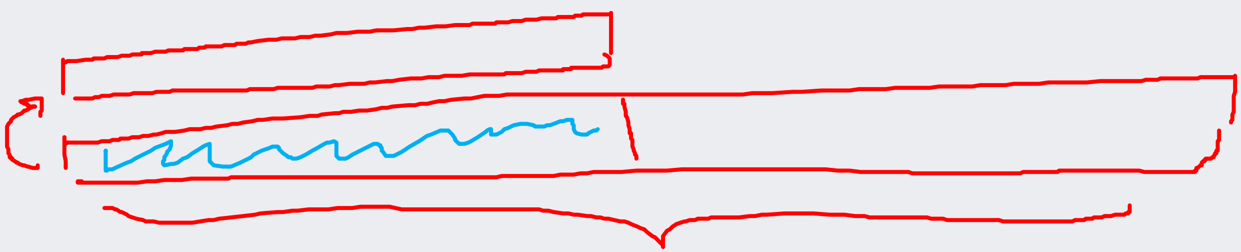
 1 element pushes at $O(1)$

 1 element copied, 1 element pushes at $O(1)$

 copy 2 elems, 2 elems pushed at $O(1)$

 copy 4 elems, push 4 elems at $O(1)$

⋮



copy $n/2$ elems, push $n/2$ more elems at $O(1)$ each

elements copied: $1 + 2 + 4 + 8 + n \ldots + \frac{n}{2} = n$

total pushes: n

Total cost of n pushes $= 2n$

average : O(n)

what is the average complexity of one push operation if:

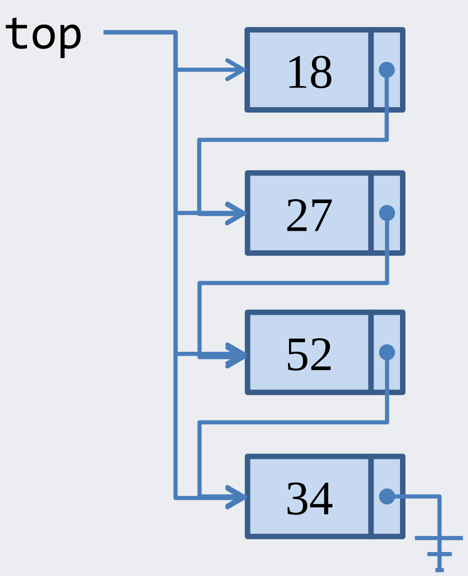
$$capacity = capacity + c$$

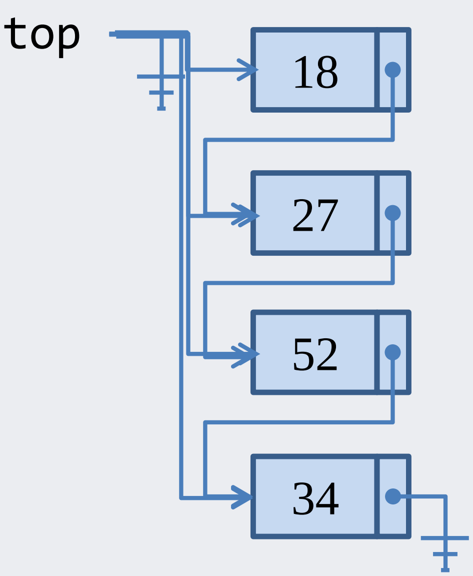
c is an integer > 0

Stack implementation

Using a linked list

- From a client perspective, usage of the stack involves only calling stack functions (e.g. push, pop, peek, etc.)
  - The data storage can be implemented with other data structures
- With a singly-linked list, the front of the list is accessed easily
  - The stack inserts and removes from the top, so let's insert and remove from the front of the list!





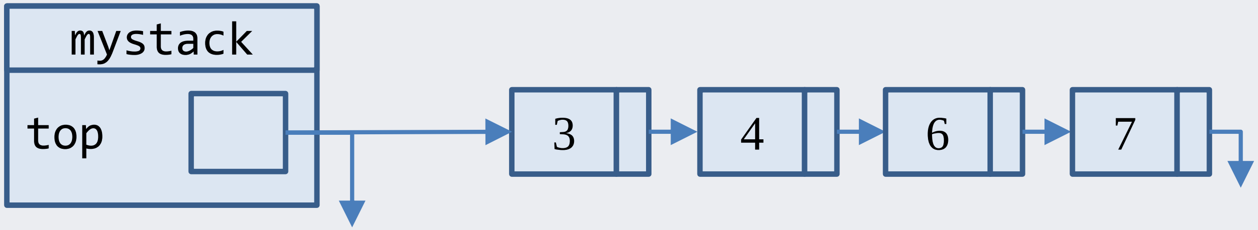
Stack implementation

```
struct Node {
  int data;
  struct Node* next;
};
```

```
typedef struct {
  struct Node* top;
} Stack;
```

```
void initStack(Stack* st) {
  st->top = NULL;
}
```

```
int isEmpty(Stack* st) {
  if (st->top == NULL)
    return TRUE;
  else
    return FALSE;
}
```

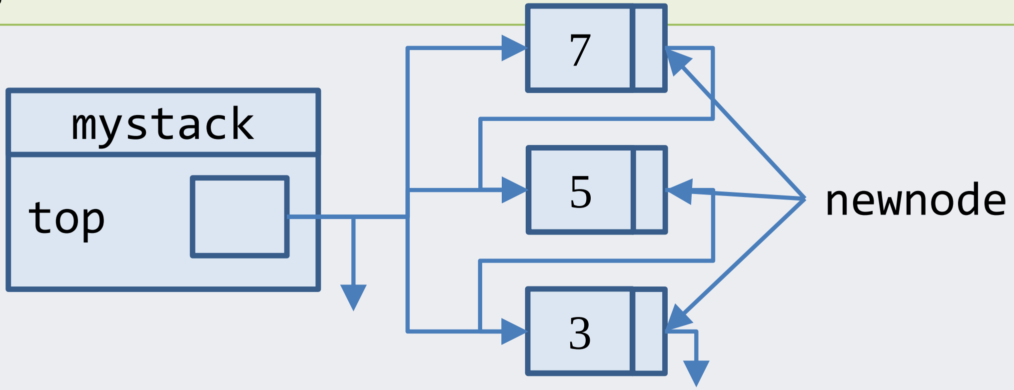


Stack implementation

- Using a linked list

```
int push(Stack* st, int val) {
  struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
  if (newnode == NULL)
    return FALSE;

  newnode->data = val;
  newnode->next = st->top;
  st->top = newnode;
  return TRUE;
}
```

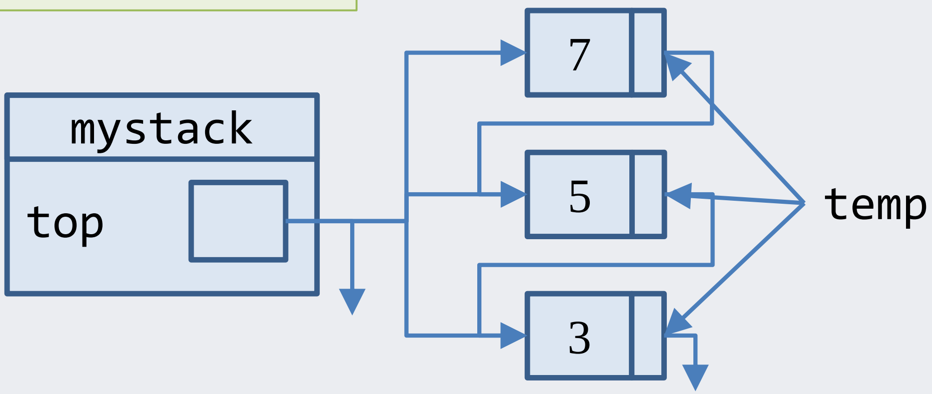
```
push(&mystack, 3);
push(&mystack, 5);
push(&mystack, 7);
```



Stack implementation

```
int pop(Stack* st) {
  if (!isEmpty(st)) {
    struct Node* temp = st->top;
    st->top = st->top->next;
    free(temp);
    temp = NULL;
    return TRUE;
  }
  return FALSE;
}
```

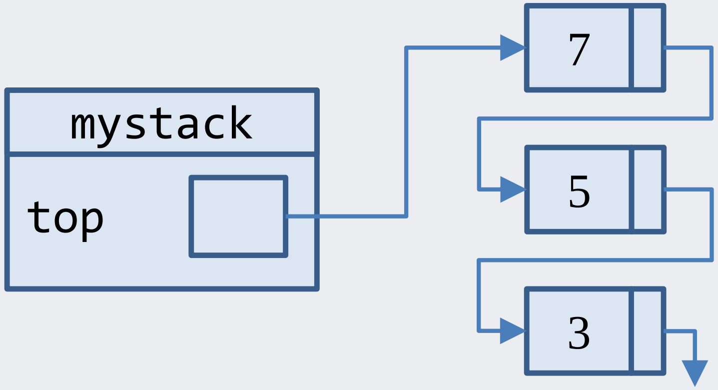
```
pop(&mystack);
pop(&mystack);
pop(&mystack);
```

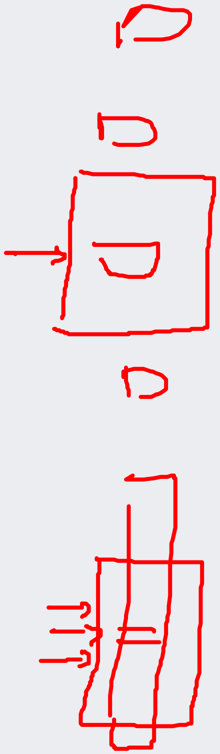


Stack implementation

Using a linked list

```
int peek(Stack* st) {
  if (!isEmpty(st))
    return st->top->data;
  else
    return FALSE;
}
```





- Notice that the function signatures for the array-based implementation and the linked list-based implementation are identical
  - The client interacts with the stack in the same way, regardless of implementation
  - We can replace the data structure and implementation, and the client will not notice at all (in most cases)

LL: O(1) avg, worst, no wasted space

array: O(1) avg, O(n) worst, possible unused array space, better cache performance

Queue ADT

Queues

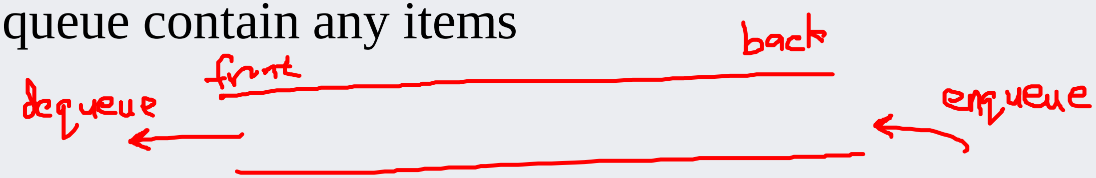
- Suppose we want to devise a system for students to ask Geoff questions after class
  - need to keep track of some information, e.g.
    - student name
    - question details
    - timestamp of request
  - need a fair system, i.e. nobody cuts in line

- The Queue ADT satisfies the required FIFO behaviour
  - first in, first out
  - if x is inserted into the collection before y, x will be removed before y

Queue applications

- Holding printer jobs
- CPU job scheduling
- Database requests
- Packet routing for a messaging server
- Course waitlists
- Network searching (breadth-first search)
- ...

- Queue ADT should support at least the first two operations:
  - **enqueue** – insert an item to the back of the queue
  - **dequeue** – remove an item from the front of the queue
  - **peek** – return the element at the front of the queue
  - **isEmpty** – does the queue contain any items
  - initialization, etc.



- The Queue ADT is completely described by the above behaviour
  - a client using the queue only needs to understand how to call the functions
  - like the stack, queue ADT can be implemented using different data structures, transparently to client code

iClicker 07.1

FIFO behaviour

- Suppose we perform the queue operations listed at the left side. In order, what items are removed?

front back

t s

removed order d o n u

FIFO

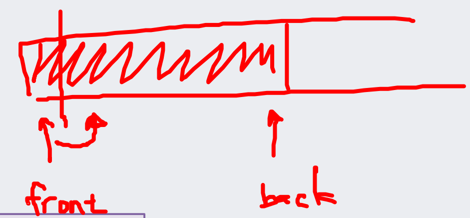
```
enqueue(d)
enqueue(o)
dequeue()
enqueue(n)
enqueue(u)
enqueue(t)
dequeue()
dequeue()
enqueue(s)
dequeue()
```

A. outs
B. dnus
C. dtus
D. None of these, but it **can** be determined from just the ADT
E. None of these, and it **cannot** be determined from just the ADT

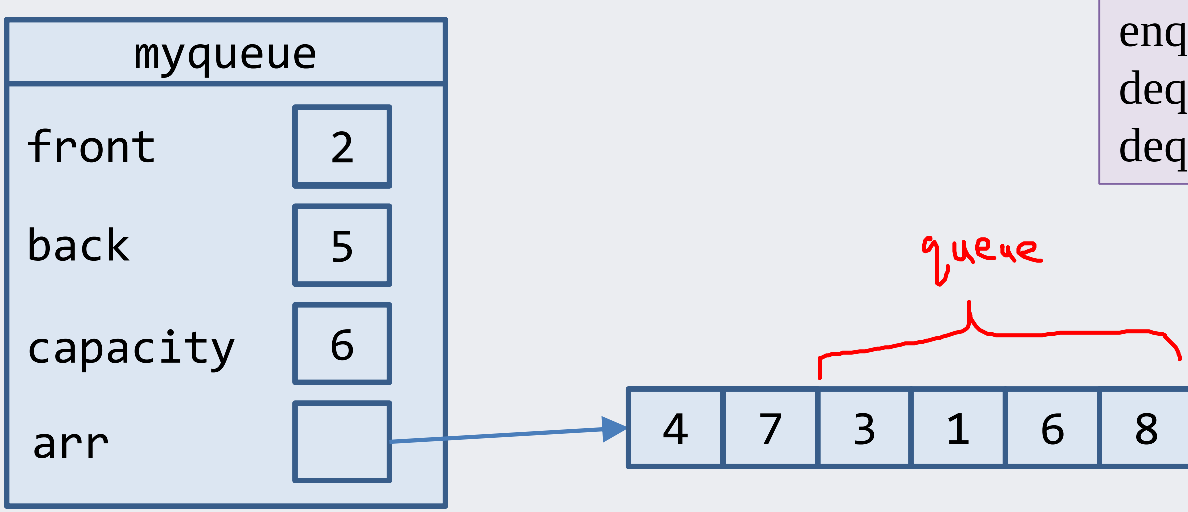
Queue implementation

Using an array

- Insertions happen at the back of the queue
  - Let's try making the back of the array the back of the queue
  - and the front of the array the front of the queue
- If the front is always index 0, we need to shuffle with every dequeue – $O(n)$
  - So insertions will increment the back index, and
  - Removals will increment the front index



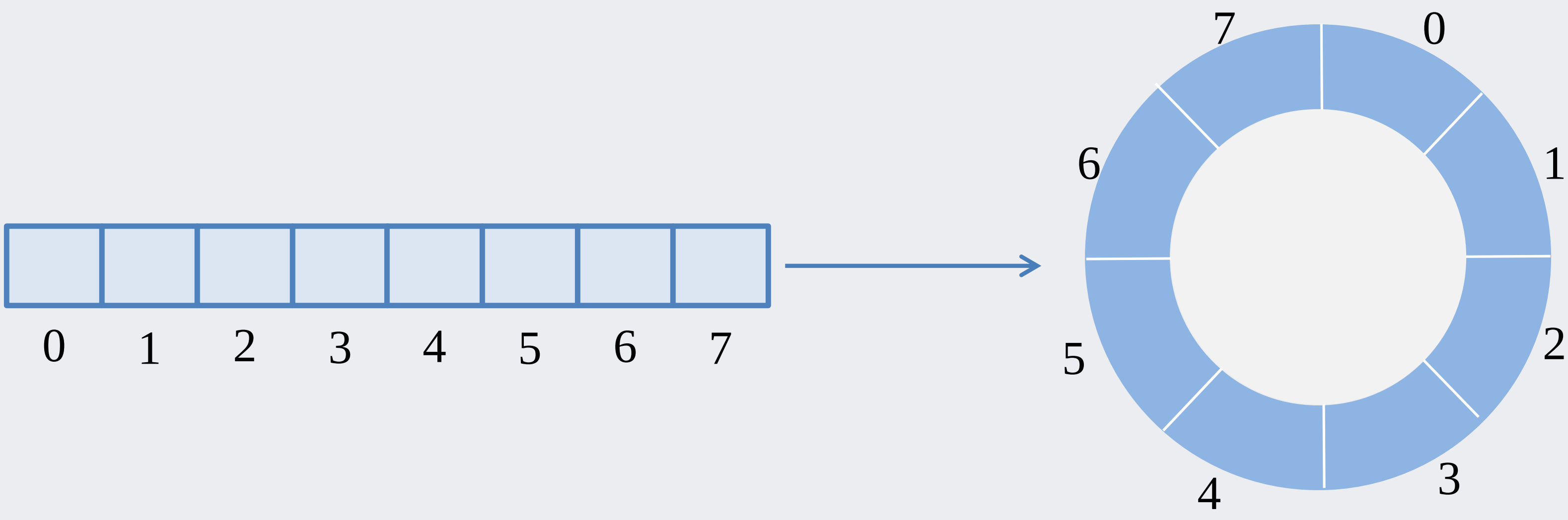
enqueue 8
dequeue
dequeue



Problem?

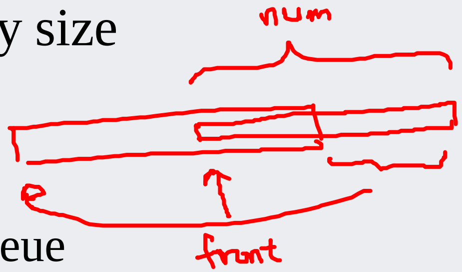
Circular arrays

- **Trick**: use a *circular array* to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array "wraps around" to the start of the array



The modulo operator

- The mod operator (%) calculates remainders:
  - `1%5 = 1, 2%5 = 2, 5%5 = 0, 8%5 = 3`
- The mod operator can be used to calculate the front and back positions in a circular array
  - Thereby avoiding comparisons to the array size
  - The back of the queue is:
    - `(front + num) % capacity`
    - where num is the number of items in the queue
  - After removing an item, the front of the queue is:
    - `(front + 1) % capacity`




```c
typedef struct {
  int front;
  int num;
  int capacity;
  int* arr;
} Queue;
```

Queue implementation

Using an array

```c
typedef struct {
  int front;
  int num;
  int capacity;
  int* arr;
} Queue;
```

```c
void initialize(Queue* q) {
  q->front = 0;
  q->num = 0;
  q->capacity = 6; // or some other value
  q->arr = (int*) malloc(q->capacity * sizeof(int));
}
```

```c
int isEmpty(Queue* q) {
  if (q->num == 0)
    return TRUE;
  else
    return FALSE;
}
```

```c
int isFull(Queue* q) {
  if (q->num == q->capacity)
    return TRUE;
  else
    return FALSE;
}
```

Queue implementation

Using an array

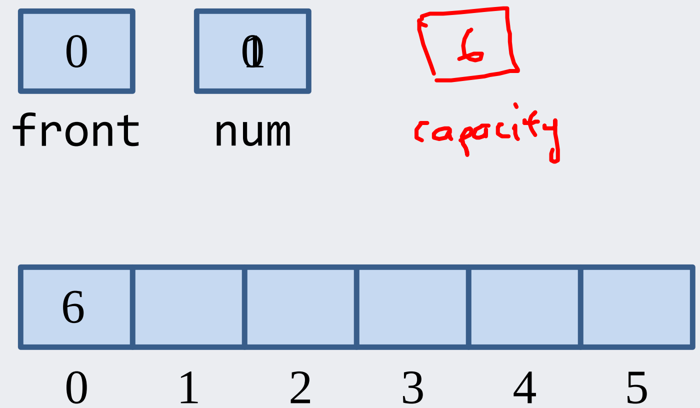
```
int enqueue(Queue* q, int val) {
  if (isFull(q))
    return FALSE;
                          index of back of queue
  else {
    q->arr[(q->front + q->num) % q->capacity] = val;
    q->num++;
    return TRUE;
  }
}
```

```
int dequeue(Queue* q) {
  if (isEmpty(q))
    return FALSE;
  else {
    q->arr[q->front] = -1; optional
    q->front = (q->front + 1) % q->capacity;
    q->num--;
    return TRUE;
  }
}
```

Array queue example

```
Queue myq;
initialize(&myq);
```

```
enqueue(&myq, 6);
```



Insert item at (front + num) % capacity, then increment num

Array queue example

```
Queue myq;
initialize(&myq);
```

```
enqueue(&myq, 6);
```

```
enqueue(&myq, 4);
enqueue(&myq, 7);
enqueue(&myq, 3);
enqueue(&myq, 8);
```

```
dequeue(&myq);
```

```
dequeue(&myq);
```

front	num
[illegible]	[illegible]

6	4	7	3	8	
0	1	2	3	4	5

Insert item at (front + num) % capacity, then increment num

Remove item at front, then decrement num and make front = (front + 1) % capacity

Array queue example

```
Queue myq;
initialize(&myq);
```

```
enqueue(&myq, 6);
```

```
enqueue(&myq, 4);
enqueue(&myq, 7);
enqueue(&myq, 3);
enqueue(&myq, 8);
```

```
dequeue(&myq);
```

```
dequeue(&myq);
```

```
enqueue(&myq, 9);
```

```
enqueue(&myq, 5);
```

front	num
2	4

0	1	2	3	4	5
5		7	3	8	9

Insert item at (front + num) % capacity, then increment num

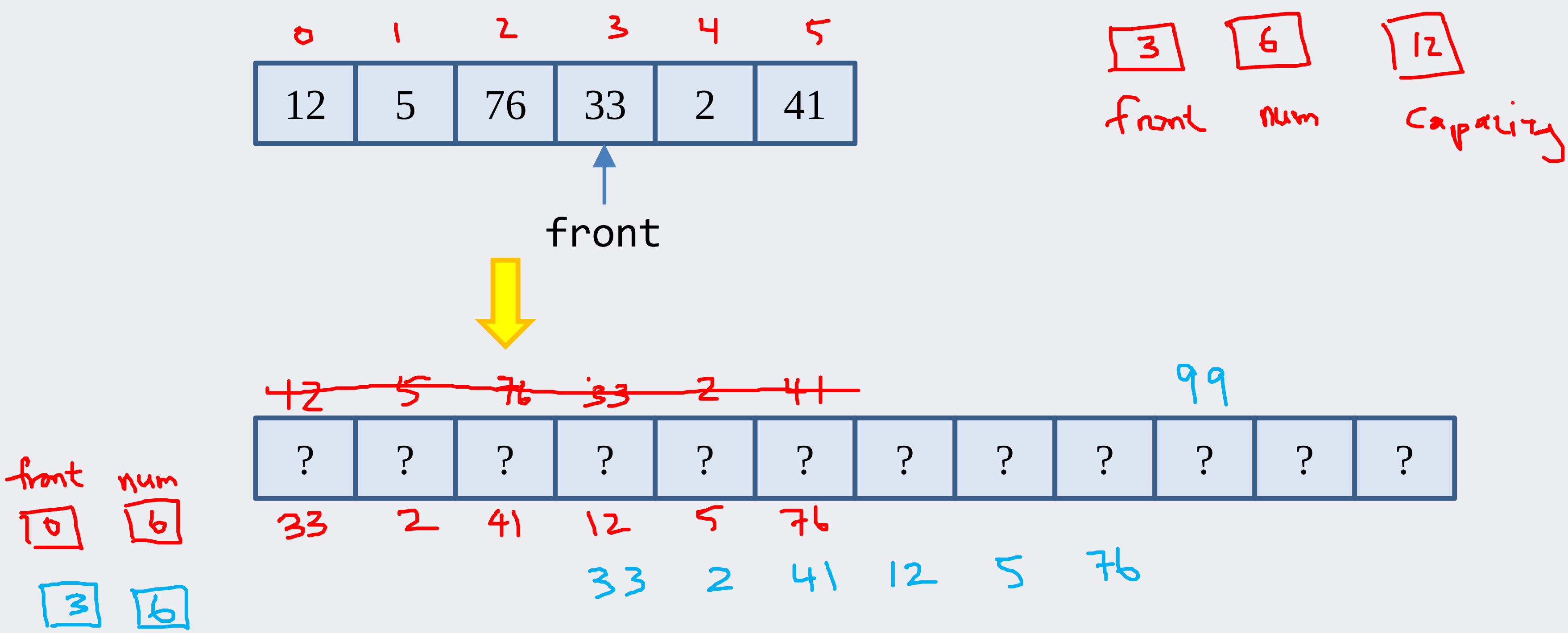
Remove item at front, then decrement num and make front = (front + 1) % capacity

enqueue is possible as long as the array is not full

Array queue resizing

- Suppose we have an array-based queue with (theoretically) unlimited enqueueing and we have performed some enqueue and dequeue operations
  - Then we perform more enqueues to fill the array
  - How should we resize the array to allow for more enqueue operations?

33 2 41 12 5 76 99



Queue implementation

Using a linked list

- Removing items from the front of the queue is straightforward
- Items should be inserted at the back of the queue in constant time
  - So we must avoid traversing through the list
  - Use a back pointer!

```c
typedef struct {
  struct Node* front;
  struct Node* back;
  int num;
} Queue;
```

```c
struct Node {
  int data;
  struct Node* next;
};
```

```c
void initialize(Queue* q) {
  q->front = NULL;
  q->back = NULL;
  q->num = 0;
}
```

```c
int isEmpty(Queue* q) {
  if (q->front == NULL)
    return TRUE;
  else
    return FALSE;
}
```

Queue implementation

```
int enqueue(Queue* q, int val) {
  struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
  if (q->front == NULL) { // special case: empty list
    q->front = newnode;
    q->back = newnode;
  }
  else { // general case
    q->back->next = newnode;
    q->back = newnode;
  }
  q->num++;
  return TRUE;
}
```

newnode->data = val;

```
int peek(Queue* q) {
  if (isEmpty(q))
    return FALSE;
  else
    return q->front->data;
}
```

```
int dequeue(Queue* q) {
  struct Node* temp = front;
  if (isEmpty(q))
    return FALSE;
  else {
    q->front = q->front->next;
    free(temp);
    temp = NULL;
    q->num--;
    if (isEmpty(q))
      q->back = NULL;
    return TRUE;
  }
}
```

List queue example

```
Queue myq;
initQueue(&myq);
```

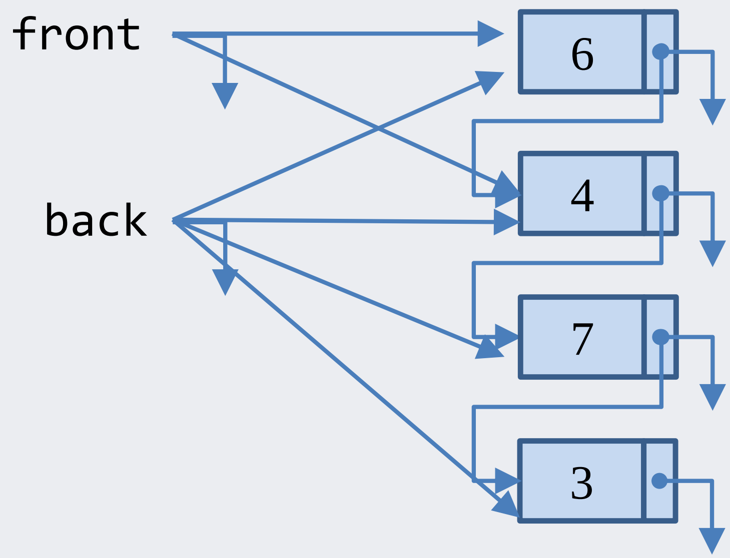
```
enqueue(&myq, 6);
```

```
enqueue(&myq, 4);
```

```
enqueue(&myq, 7);
```

```
enqueue(&myq, 3);
```

```
dequeue(&myq);
```



Exercise

- For the array-based queue, write the enqueue function to support array resizing!
  - See the array resizing stack example for reference, but be aware of the issues mentioned in slide #20