

Algorithms and Data Structures in C

Estefania Talavera

University of Groningen — Bernoulli Institute

Based on slides by Gerard Renardel and Malvin Gattinger



**university of
 groningen**

**faculty of science
 and engineering**

3 February 2020

Lecture 1

Introduction

Course Goals

Linear Data Structures

Stacks

Queues

Priority Queues

Linked Lists

Course Goals

Learning goals of the course:

- knowledge of several fundamental algorithms and data structures

- ability to write efficient programs in C for these algorithms

- ability to describe algorithms in pseudocode

- ability to solve problems, using these algorithms and data structures

- ability to write programming reports

Course Goals

Learning goals of the course:

- knowledge of several fundamental algorithms and data structures
 - ability to write efficient programs in C for these algorithms
 - ability to describe algorithms in pseudocode
 - ability to solve problems, using these algorithms and data structures
 - ability to write programming reports
-
- ▶ **Algorithms**: abstract version of programs
 - ▶ **Data Structures**: abstract version of files

Course Goals

Learning goals of the course:

- knowledge of several fundamental algorithms and data structures
 - ability to write efficient programs in C for these algorithms
 - ability to describe algorithms in pseudocode
 - ability to solve problems, using these algorithms and data structures
 - ability to write programming reports
- ▶ **Algorithms**: abstract version of programs
 - ▶ **Data Structures**: abstract version of files

Position in the bachelor programme:

- ▶ second course on programming, after Imperative Programming
- ▶ prepares for Artificial Intelligence 1
- ▶ prepares for Advanced Algorithms and Data Structures

Modus operandi

Lectures: 2 hours weekly (Estefania Talavera and Fadi Mohsen)

Tutorials: 2 hours weekly (teaching assistants)

Lab sessions: 2 hours weekly (teaching assistants)

Modus operandi

Lectures: 2 hours weekly (Estefania Talavera and Fadi Mohsen)

Tutorials: 2 hours weekly (teaching assistants)

Lab sessions: 2 hours weekly (teaching assistants)

If you have not done it already, please **enroll to your group on Nestor**.

Teaching assistants:

Nicu Ghidirimschi, Alexander Ivanov, Evi Xhelo, Hleb Shmak, Sneha Lodha, Marten Struijk, Joel During

Modus operandi

Lectures: 2 hours weekly (Estefania Talavera and Fadi Mohsen)

Tutorials: 2 hours weekly (teaching assistants)

Lab sessions: 2 hours weekly (teaching assistants)

If you have not done it already, please **enroll to your group on Nestor**.

Teaching assistants:

Nicu Ghidirimschi, Alexander Ivanov, Evi Xhelo, Hleb Shmak, Sneha Lodha, Marten Struijk, Joel During

If you need help outside the tutorials or lab sessions,

first read the guidelines on Nestor and then email your TA.

Help and advice via email: **adinc1920@gmail.com**

during office hours (Monday to Friday 9 – 17 h.)

See the Nestor page for guidelines for asking help.

Weekly planning (1)

- week 1** setup of the course, a bit about C, stacks & queues, lists
assignment: Digital Signals (no report)
- week 2** lists + an application: processing of arithmetical expressions
assignment: Solving equations (part 1, no report)
- week 3** binary trees, search trees
assignment: Solving equations (part 2, no report)
- week 4** heaps, tries; pseudocode
assignment: the Spellchecker problem (with programming report)

Weekly planning (2)

week 5 application: expression trees

assignment: Expressions (part 1, no report)

week 6 graphs

assignment: Expressions (part 2, no report)

week 7 Dijkstra's shortest path algorithm

assignment: Assignment 5 TBD (with programming report)

week 8 *only lab sessions*

week 9 deadline Assignment 5

week 10 exam

5 lab assignments: made by groups of 2 students

program are checked automatically by Themis (and rechecked by us).

assignments 3 and 5: also a **programming report**, written in \LaTeX , handed in via the post box (Bernoulliborg, near room 2.22)

Assignments 1, 2, 4: *may* be made and submitted by 1 student

See Nestor for all grading criteria.

Deadline: Monday 11:00 h.

5 lab assignments: made by groups of 2 students
program are checked automatically by Themis (and rechecked by us).
assignments 3 and 5: also a **programming report**, written in L^AT_EX, handed in via the
post box (Bernoulliborg, near room 2.22)
Assignments 1, 2, 4: *may* be made and submitted by 1 student
See Nestor for all grading criteria.
Deadline: Monday 11:00 h.

Exam: Thursday morning 9 April at 08:30 (digital examination)
Final grade: average of lab result and exam grade
only sufficient (≥ 6) when both are 5.0 or above

Information on the Nestor page

Announcements

Course information link to the schedule, all deadlines, tutorial and lab groups, the lab, help from the TAs, grading criteria

Literature and files the lecture notes, template for programming report, example programming report, lab assignments, lecture slides, answers to tutorial exercises, last year's exam

Weekly schedule content of the lectures, list of tutorial exercises, deadlines for the lab assignments

Teaching staff Estefania Talavera and Fadi Mohsen

Extra Material Valgrind, text editors, Git and GitHub, L^AT_EX, Make and Makefile

Contents of a programming report

problem description What is the problem?

Specification of the input-output behaviour.

problem analysis What is the essence of the problem?

How can it be solved?

Useful methods: abstraction, divide-and-conquer, analogy.

program design How to translate the problem solution into a C program.

Not too detailed, keep it general!

Indicate design choices.

program evaluation Report about the testing and the performance of the program.

Also check for memory leaks.

process description What was easy, what was difficult, what did you learn?

Who did what?

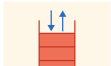
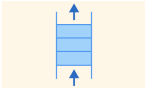
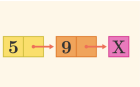
conclusions Is the problem solved? Is the solution efficient? is it optimal?

appendices program text, test sets

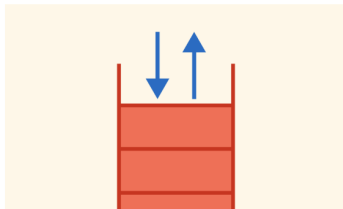
optional extensions of the program

Questions so far about the course
structure? 🤔

Linear Data Structures: Overview

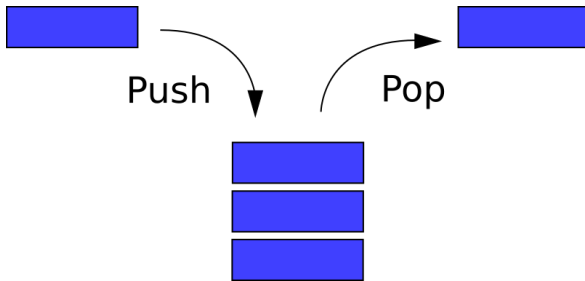
	name	methods
	Stack (LIFO)	push, pop
	Queue (FIFO)	enqueue, dequeue
	Priority Q	enqueue, removeMax
	Linked List	addItem, firstItem, removeFirstNode, visitList, itemAtPos

The Stack



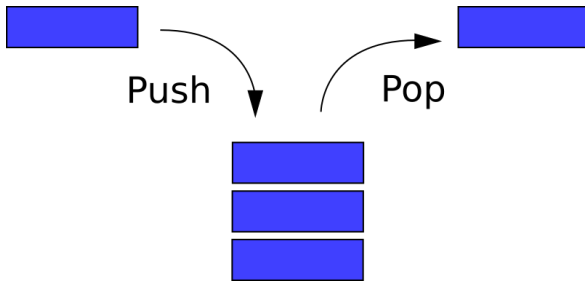
the stack (1.1)

A *stack* is a linear data structure with two operations to add and remove items:



the stack (1.1)

A *stack* is a linear data structure with two operations to add and remove items:

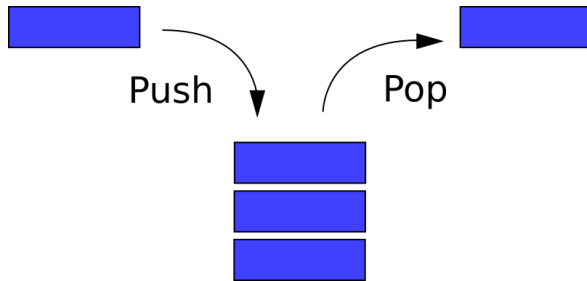


push adds an item on top of the stack

pop removes and returns the uppermost item

the stack (1.1)

A *stack* is a linear data structure with two operations to add and remove items:



push adds an item on top of the stack

pop removes and returns the uppermost item

A stack follows the **LIFO** rule: **L**ast **I**n, **F**irst **O**ut.

the stack

1	push(7)
2	push(8)
3	pop()
4	push(pop()*3)
5	push(40)
6	push(pop()/2)

the stack

1	push(7)
2	push(8)
3	pop()
4	push(pop()*3)
5	push(40)
6	push(pop()/2)

Answer = 21, 20

the stack

1	push(1)
2	push(pop()*2)
3	push(pop()+1)
4	pop()
5	pop()

the stack

1	push(1)
2	push(pop()*2)
3	push(pop()+1)
4	pop()
5	pop()

Answer = *stack underflow* \Rightarrow an item is called for from the stack, but the stack is empty.

reading from a stack

```
Value = pop()  
push(Value)
```

reading from a stack

Value = pop()

push(Value)

or \Rightarrow peek() , returns the value of the top element of the stack without changing the stack.

reading from a stack

Value = pop()

push(Value)

or \Rightarrow peek() , returns the value of the top element of the stack without changing the stack.

1	push(7)
2	push(8)
3	push(pop()*3)
4	push(9)
5	pop()
6	peek()

reading from a stack

Value = pop()

push(Value)

or \Rightarrow peek() , returns the value of the top element of the stack without changing the stack.

1	push(7)
2	push(8)
3	push(pop()*3)
4	push(9)
5	pop()
6	peek()

Answer = 24

Implementation of a stack in C

How to make a stack? For example with an array:

```
typedef struct Stack {  
    int *array;  
    int top;  
    int size;  
} Stack;
```

top indicates the first free position in the array

size is the size of the array — **not the current number of items on the stack!**

create a stack

```
Stack newStack(int s) {  
    Stack st;  
    st.array = malloc(s * sizeof(int));  
    assert(st.array != NULL);  
    st.top = 0;  
    st.size = s;  
    return st;  
}
```

the function push – naively and wrong

```
void push(int value, Stack *stp) {  
    stp->array[stp->top] = value;  
    stp->top++;  
}
```

the function push – naively and wrong

```
void push(int value, Stack *stp) {  
    stp->array[stp->top] = value;  
    stp->top++;  
}
```

This will fail if `stp->array` is too small.
We need to check `stp->size` first!

the function push – correct version

```
void push(int value, Stack *stp) {  
    if (stp->top == stp->size) {  
        doubleStackSize(stp); // see next slide  
    }  
    stp->array[stp->top] = value;  
    stp->top++;  
}
```

make a stack larger

What to do when the stack is full?

```
void doubleStackSize(Stack *stp) {  
    int newSize = 2 * stp->size;  
    stp->array = realloc(stp->array, newSize * sizeof(int));  
    assert(stp->array != NULL);  
    stp->size = newSize;  
}
```

Note `*stp` is a **reference parameter** — see Appendix A.2.

Recall that `stp->size` and `stp->array` are shorthand
for `(*stp).size` and `(*stp).array`.

how to deal with an empty stack

Before we implement `pop()`, consider what can go wrong.

how to deal with an empty stack

Before we implement `pop()`, consider what can go wrong.

```
int isEmptyStack(Stack st) {  
    return (st.top == 0);  
}  
  
void stackEmptyError () {  
    printf("stack empty\n");  
    abort();  
}
```

the functions pop and freeStack

```
int pop(Stack *stp) {  
    if ( isEmptyStack(*stp) ) {  
        stackEmptyError();  
    }  
    stp->top--;  
    return (stp->array)[stp->top];  
}
```

the functions pop and freeStack

```
int pop(Stack *stp) {  
    if ( isEmptyStack(*stp) ) {  
        stackEmptyError();  
    }  
    stp->top--;  
    return (stp->array)[stp->top];  
}
```

```
void freeStack (Stack st) {  
    free(st.array);  
}
```

the stack: code overview

```
typedef struct Stack {  
    int *array;  
    int top;  
    int size;  
} Stack;  
  
Stack newStack(int s);  
  
void doubleStackSize(Stack *stp);  
int isEmptyStack(Stack st);  
void stackEmptyError();  
void push(int value, Stack *stp);  
int pop(Stack *stp);  
void freeStack(Stack st);
```

These declarations are in LibStack.h, the definitions in LibStack.c.

the stack: a usage example

```
#include <stdio.h>
#include "LibStack.h"

int main(int argc, char *argv[]) {
    Stack stack;
    stack = newStack(2);
    push(42, &stack);
    push(17, &stack);
    push(23, &stack);
    pop(&stack);
    printf("%d\n", pop(&stack));
    return 0;
}
```



What will be the output?

time complexity of stack operations

time complexity of pop: $\mathcal{O}(1)$

time complexity of stack operations

time complexity of pop: $\mathcal{O}(1)$

time complexity of push

if the stack is full: $\mathcal{O}(\text{st.size})$ (due to doubleStackSize)

otherwise: $\mathcal{O}(1)$

Hence the **worst-case** is $\mathcal{O}(\text{st.size})$.

time complexity of stack operations

time complexity of pop: $\mathcal{O}(1)$

time complexity of push

if the stack is full: $\mathcal{O}(\text{st.size})$ (due to `doubleStackSize`)

otherwise: $\mathcal{O}(1)$

Hence the **worst-case** is $\mathcal{O}(\text{st.size})$.

What about the average?

*start with `st.size` = 1000, do push 1 000 000 times,
then `doubleStackSize` runs 10 times,
with $1000 + 2000 + \dots + 512\,000 \approx 1\,000\,000$ steps*

time complexity of stack operations

time complexity of pop: $\mathcal{O}(1)$

time complexity of push

if the stack is full: $\mathcal{O}(\text{st.size})$ (due to `doubleStackSize`)

otherwise: $\mathcal{O}(1)$

Hence the **worst-case** is $\mathcal{O}(\text{st.size})$.

What about the average?

*start with `st.size` = 1000, do push 1 000 000 times,
then `doubleStackSize` runs 10 times,
with $1000 + 2000 + \dots + 512\,000 \approx 1\,000\,000$ steps*

Hence the **average** is $\mathcal{O}(1)$.

time complexity of stack operations

time complexity of pop: $\mathcal{O}(1)$

time complexity of push

if the stack is full: $\mathcal{O}(\text{st.size})$ (due to `doubleStackSize`)

otherwise: $\mathcal{O}(1)$

Hence the **worst-case** is $\mathcal{O}(\text{st.size})$.

What about the average?

*start with `st.size` = 1000, do push 1 000 000 times,
then `doubleStackSize` runs 10 times,
with $1000 + 2000 + \dots + 512\,000 \approx 1\,000\,000$ steps*

Hence the **average** is $\mathcal{O}(1)$.

Exercise: Other strategies for enlarging a stack.

Suppose that we model a line at an amusement park as a stack. We would like the top of the stack to represent the end of the line, and the bottom of the stack to represent the front of the line.



Suppose that we model a line at an amusement park as a stack. We would like the top of the stack to represent the end of the line, and the bottom of the stack to represent the front of the line.



So, if someone new stands in line, he adds them to the top of the stack. When someone gets on the ride, he will need to somehow remove them from the bottom of the stack rather than the top.

Suppose that we model a line at an amusement park as a stack. We would like the top of the stack to represent the end of the line, and the bottom of the stack to represent the front of the line.



So, if someone new stands in line, he adds them to the top of the stack. When someone gets on the ride, he will need to somehow remove them from the bottom of the stack rather than the top.

If there are 300 people waiting in line, how many operations on the stack will be required in order to model the line's first inhabitant getting on the ride?

Suppose that we model a line at an amusement park as a stack. We would like the top of the stack to represent the end of the line, and the bottom of the stack to represent the front of the line.



So, if someone new stands in line, he adds them to the top of the stack. When someone gets on the ride, he will need to somehow remove them from the bottom of the stack rather than the top.

If there are 300 people waiting in line, how many operations on the stack will be required in order to model the line's first inhabitant getting on the ride?

Answer = 599

we would need to pop 299 people off of the stack, then pop off the person who gets on the ride, and then hed need to push the other 299 people back onto the stack. That is 599 operations in total.

Suppose that we model a line at an amusement park as a stack. We would like the top of the stack to represent the end of the line, and the bottom of the stack to represent the front of the line.



So, if someone new stands in line, he adds them to the top of the stack. When someone gets on the ride, he will need to somehow remove them from the bottom of the stack rather than the top.

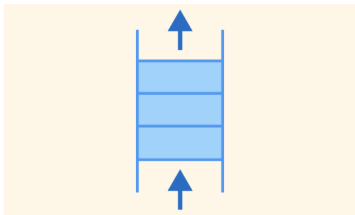
If there are 300 people waiting in line, how many operations on the stack will be required in order to model the line's first inhabitant getting on the ride?

Answer = 599

we would need to pop 299 people off of the stack, then pop off the person who gets on the ride, and then need to push the other 299 people back onto the stack. That is 599 operations in total.

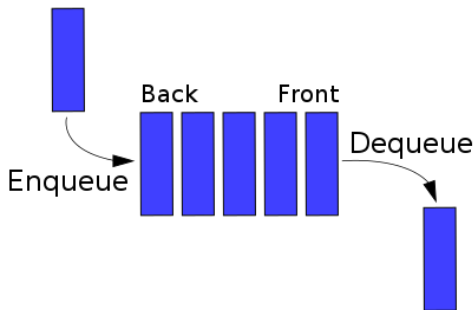
Is there a faster way to do this?

The Queue



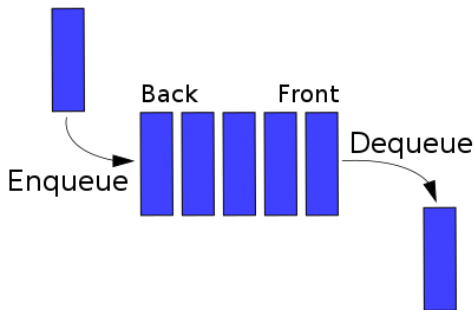
the queue (1.2)

A *queue* is a linear data structure with two operations to add and remove items:



the queue (1.2)

A *queue* is a linear data structure with two operations to add and remove items:

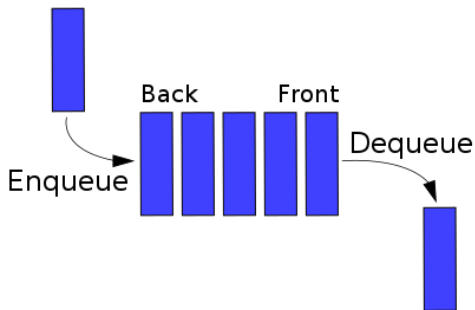


`enqueue` adds an item at the back

`dequeue` removes and returns the item at the front

the queue (1.2)

A *queue* is a linear data structure with two operations to add and remove items:



`enqueue` adds an item at the back

`dequeue` removes and returns the item at the front

A queue follows the **FIFO** rule: **F**irst **I**n, **F**irst **o**ut

The Queue: example

1	<code>enqueue(3)</code>
2	<code>enqueue(dequeue()*4)</code>
3	<code>enqueue(5)</code>
4	<code>enqueue(3*dequeue())</code>
5	<code>enqueue(503)</code>

The Queue: example

1	enqueue(3)
2	enqueue(dequeue()*4)
3	enqueue(5)
4	enqueue(3*dequeue())
5	enqueue(503)

Head **3** Tail

Head **12** Tail

Head **12** **5** Tail

Head **5** **36** Tail

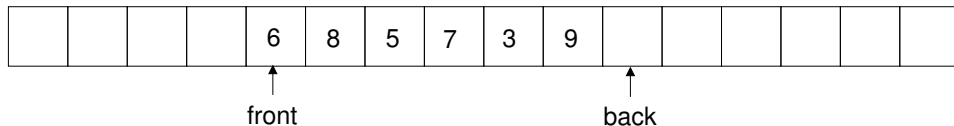
Head **5** **36** **503** Tail

implementation of the queue using an array

```
typedef struct Queue {  
    int *array;  
    int back;  
    int front;  
    int size;  
} Queue;
```

back is the first free position in the array

front is the position of the 'oldest' item (unless the queue is empty)



The queue is empty iff **q.back == q.front**.

create an empty queue

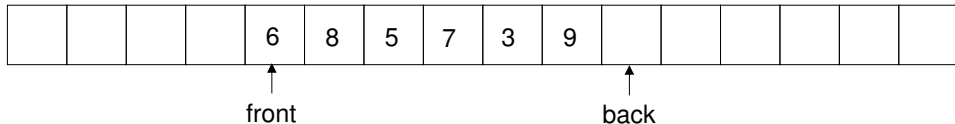
```
Queue newQueue(int s) {  
    Queue q;  
    q.array = malloc(s*sizeof(int));  
    assert(q.array != NULL);  
    q.back = 0;  
    q.front = 0;  
    q.size = s;  
    return q;  
}
```

testing if a queue is empty

```
int isEmptyQueue(Queue q) {  
    return (q.back == q.front);  
}  
  
void queueEmptyError() {  
    printf("queue empty\n");  
    abort();  
}
```

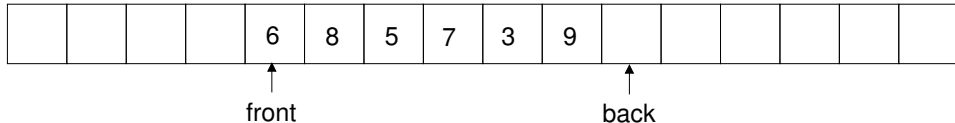
preparing the queue implementation

enqueue will increment `q.back`,
dequeue will increment `q.front`



preparing the queue implementation

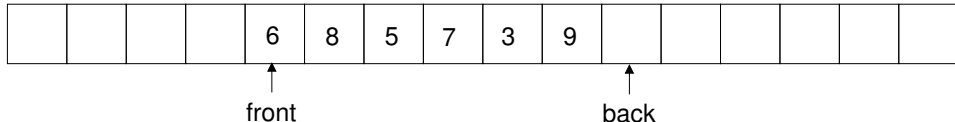
enqueue will increment q.back,
dequeue will increment q.front



When `q.back` gets the value `q.size`, the end of `q.array` has been reached. But there may be free places between 0 and `q.front`!

preparing the queue implementation

enqueue will increment `q.back`,
dequeue will increment `q.front`



When `q.back` gets the value `q.size`, the end of `q.array` has been reached.

But there may be free places between 0 and `q.front`!

We let both markers jump to 0 instead of `q.size`:

`q.back = (q.back+1) % q.size`

`q.front = (q.front+1) % q.size`

preparing the queue implementation

enqueue will increment `q.back`,

dequeue will increment `q.front`

When `q.back` gets the value `q.size`, the end of `q.array` has been reached.

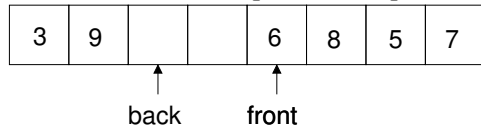
But there may be free places between 0 and `q.front`!

We let both markers jump to 0 instead of `q.size`:

`q.back = (q.back+1) % q.size`

`q.front = (q.front+1) % q.size`

This might lead to `q.back < q.front`. We call this a **split configuration**.



the functions enqueue and dequeue

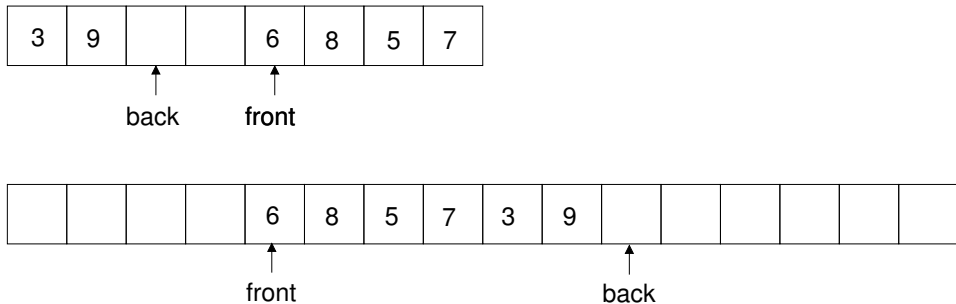
```
void enqueue(int item, Queue *qp) {
    qp->array[qp->back] = item;
    qp->back = (qp->back + 1) % qp->size;
    if ( qp->back == qp->front ) {
        doubleQueueSize(qp);
    }
}
```


the functions enqueue and dequeue

```
void enqueue(int item, Queue *qp) {
    qp->array[qp->back] = item;
    qp->back = (qp->back + 1) % qp->size;
    if ( qp->back == qp->front ) {
        doubleQueueSize(qp);
    }
}
```

```
int dequeue(Queue *qp) {
    int item;
    if (isEmptyQueue(*qp)) {
        queueEmptyError();
    }
    item = qp->array[qp->front];
    qp->front = (qp->front + 1) % qp->size;
    return item;
}
```

doubleQueueSize removes a split configuration



extending and freeing a queue

```
void doubleQueueSize(Queue *qp) {
    int i;
    int oldSize = qp->size;
    qp->size = 2 * oldSize;
    qp->array = realloc(qp->array, qp->size * sizeof(int));
    assert(qp->array != NULL);
    for (i=0; i < qp->back; i++) { /* eliminate split */
        qp->array[oldSize + i] = qp->array[i];
    }
    qp->back = qp->back + oldSize; /* update qp.back */
}

void freeQueue(Queue q) {
    free(q.array);
}
```

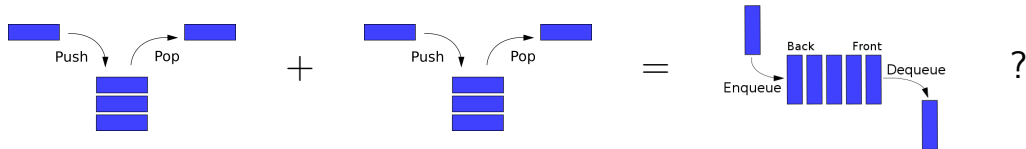
the queue: code overview

```
typedef struct Queue {  
    int *array;  
    int back;  
    int front;  
    int size;  
} Queue;  
  
Queue newQueue (int s);  
int isEmptyQueue(Queue q);  
void queueEmptyError();  
void doubleQueueSize(Queue *qp);  
void enqueue(int item, Queue *qp);  
int dequeue(Queue *qp);  
void freeQueue(Queue q);
```

These declarations are in LibQueue.h, the definitions in LibQueue.c.

make a queue from two stacks (1.2.3)

Is it possible to make a queue from two stacks `stack0` and `stack1`?



make a queue from two stacks (1.2.3)

Is it possible to make a queue from two stacks `stack0` and `stack1`?

Yes, it is. We use the type

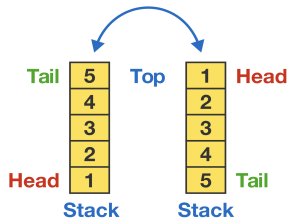
```
typedef struct Queue {  
    Stack stack0;  
    Stack stack1;  
} Queue;
```

make a queue from two stacks (1.2.3)

Is it possible to make a queue from two stacks `stack0` and `stack1`?

Yes, it is. We use the type

```
typedef struct Queue {  
    Stack stack0;  
    Stack stack1;  
} Queue;
```



Idea:

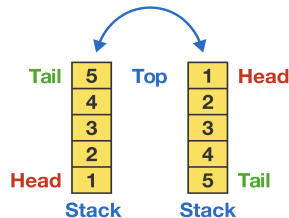
implement enqueue with a push on `q.stack0`

make a queue from two stacks (1.2.3)

Is it possible to make a queue from two stacks `stack0` and `stack1`?

Yes, it is. We use the type

```
typedef struct Queue {  
    Stack stack0;  
    Stack stack1;  
} Queue;
```



Idea:

implement enqueue with a push on `q.stack0`

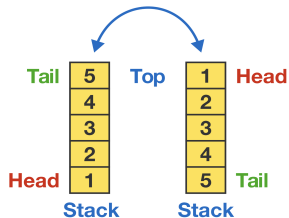
implement dequeue with a pop from `q.stack1`,

make a queue from two stacks (1.2.3)

Is it possible to make a queue from two stacks `stack0` and `stack1`?

Yes, it is. We use the type

```
typedef struct Queue {  
    Stack stack0;  
    Stack stack1;  
} Queue;
```



Idea:

implement enqueue with a push on `q.stack0`

implement dequeue with a pop from `q.stack1`,

when `q.stack1` is empty, transfer `q.stack0` to `q.stack1` using pop and push, respectively.

make a queue from two stacks: enqueue and dequeue

```
void enqueue(int value, Queue *qp) {  
    push(value, &(qp->stack0)); /* push on stack0 */  
}
```

make a queue from two stacks: enqueue and dequeue

```
void enqueue(int value, Queue *qp) {  
    push(value, &(qp->stack0)); /* push on stack0 */  
}
```

```
int dequeue(Queue *qp) {  
    if ( isEmptyStack(qp->stack1) ) {  
        while (! isEmptyStack(qp->stack0) ) {  
            /* transfer from stack0 to stack1, reversing order */  
            push(pop(&(qp->stack0)), &(qp->stack1));  
        }  
    }  
    return pop(&(qp->stack1)); /* pop from stack1 */  
}
```

make a queue from two stacks: enqueue and dequeue

```
void enqueue(int value, Queue *qp) {  
    push(value, &(qp->stack0)); /* push on stack0 */  
}
```

```
int dequeue(Queue *qp) {  
    if ( isEmptyStack(qp->stack1) ) {  
        while (! isEmptyStack(qp->stack0) ) {  
            /* transfer from stack0 to stack1, reversing order */  
            push(pop(&(qp->stack0)), &(qp->stack1));  
        }  
    }  
    return pop(&(qp->stack1)); /* pop from stack1 */  
}
```

Exercise: Vice versa, make a stack from two queues.

Priority Queues (1.2.1)

A *priority queue* is a queue where

- ▶ every item has a *priority*
- ▶ the function `dequeue` is replaced by `removeMax`:
remove and return the item with the highest priority

Priority Queues (1.2.1)

A *priority queue* is a queue where

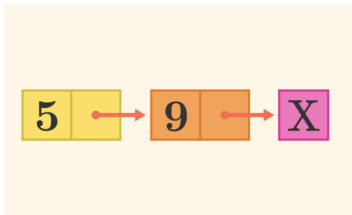
- ▶ every item has a *priority*
- ▶ the function `dequeue` is replaced by `removeMax`:
remove and return the item with the highest priority

We will see two implementations of the priority later in this course:

using an *ordered list*: `enqueue` in $\mathcal{O}(n)$, `removeMax` in $\mathcal{O}(1)$

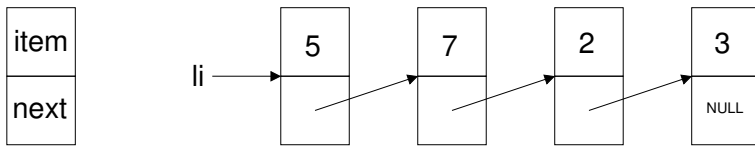
using a *heap*: both `enqueue` and `removeMax` in $\mathcal{O}(\log(n))$

Linked Lists



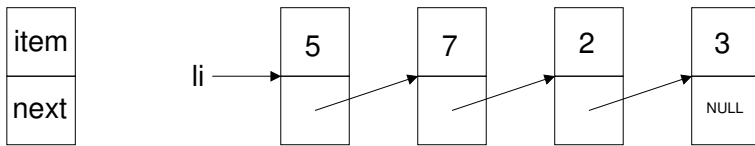
Linked Lists

A *linked list* is a linear data structure where each item has a pointer to the next element.



Linked Lists

A *linked list* is a linear data structure where each item has a pointer to the next element.



Methods: addItem, firstItem, removeFirstNode, visitList, itemAtPos

Linked Lists in C

```
typedef struct ListNode* List;  
  
struct ListNode {  
    int item;  
    List next;  
};
```

struct ListNode is a composite type, with a field **item** and a field **next**
List is the type of pointers to structures of type **ListNode**

Linked Lists: creating and adding

create a new (empty) list:

```
List newList() {  
    return NULL;  
}
```

Linked Lists: creating and adding

create a new (empty) list:

```
List newList() {  
    return NULL;  
}
```

add an item:

```
List addItem(int n, List li) {  
    List newList = malloc(sizeof(struct ListNode));  
    assert(newList != NULL);  
    newList->item = n;  
    newList->next = li;  
    return newList;  
}
```

Linked Lists: getting items

return the first item in a list:

```
void listEmptyError() {  
    printf("list empty\n");  
    abort();  
}  
  
int firstItem(List li) {  
    if ( li == NULL ) {  
        listEmptyError();  
    }  
    return li->item;  
}
```

Linked Lists: removing items

```
List removeFirstNode(List li) {  
    List returnList;  
    if ( li == NULL ) {  
        listEmptyError();  
    }  
    returnList = li->next;  
    free(li);  
    return returnList;  
}
```

Linked Lists: freeing

```
void freeList(List li) {  
    List li1;  
    while ( li != NULL ) {  
        li1 = li->next;  
        free(li);  
        li = li1;  
    }  
    return;  
}
```

traversing a list

Walk through a list and run some function `visit` on each node:

```
void visitList(list li) {  
    while (li != NULL) {  
        visit(li);  
        li = li->next;  
    }  
}
```


traversing a list

Walk through a list and run some function `visit` on each node:

```
void visitList(list li) {  
    while (li != NULL) {  
        visit(li);  
        li = li->next;  
    }  
}
```

Recursively:

```
void visitListRec(list li) {  
    if (li != NULL) {  
        visit(li);  
        visitListRec(li->next);  
    }  
}
```

give the item at position p

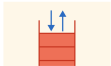
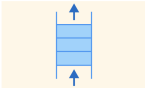
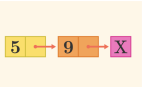
```
void listTooShort() {
    printf("List too short\n");
    abort();
}

int itemAtPos(List li, int p) {
    if (li == NULL) {
        listTooShort();
    }
    if (p == 0) {
        return firstItem(li);
    } else {
        return itemAtPos(li->next, p-1);
    }
}
```

Properties of Linked Lists

- Pros
 - ▶ No size limitation
 - ▶ Insertion and deletion operations
- Cons
 - ▶ Sequential access. To find a node at position n , you have to start the search at the first node in the linked list, following the path of references n time
 - ▶ Use more storage than the array due to their property of referencing the next node in the linked list

Linear Data Structures: Overview

	name	methods
	Stack (LIFO)	push, pop
	Queue (FIFO)	enqueue, dequeue
	Priority Q	enqueue, removeMax
	Linked List	addItem, firstItem, removeFirstNode, visitList, itemAtPos

“Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.”

— <https://valgrind.org/>

Segmentation Fault — What to do?

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      char *s = " here is a string!";
5      s[0] = 'T';
6      printf("%s\n", s);
7      return 0;
8  }
```

Segmentation Fault — What to do?

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      char *s = " here is a string!";
5      s[0] = 'T';
6      printf("%s\n", s);
7      return 0;
8  }
```

Compiling and running this leads to:

```
$ gcc badString.c -o badString
$ ./badString
Segmentation fault
```



Segmentation fault: valgrind

Compile again with `-g`, then use `valgrind ./badString`

```
$ gcc -g badString.c -o badString
$ valgrind ./badString
==28771== ...
==28771==
==28771== Process terminating with default action of signal 11 (SIGSEGV)
==28771==  Bad permissions for mapped region at address 0x108774
==28771==    at 0x1086CE: main (badString.c:5)
==28771==
==28771== ...
Segmentation fault
```


Segmentation fault: valgrind

Compile again with `-g`, then use `valgrind ./badString`

```
$ gcc -g badString.c -o badString
$ valgrind ./badString
==28771== ...
==28771==
==28771== Process terminating with default action of signal 11 (SIGSEGV)
==28771== Bad permissions for mapped region at address 0x108774
==28771==    at 0x1086CE: main (badString.c:5)
==28771==
==28771== ...
Segmentation fault
```

We may not modify or free hard-coded strings!

⇒ see Appendix A.6

fixing the segmentation fault — correctly modifying a string

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char *s = malloc(19*sizeof(char));
7      strcpy(s, " here is a string!");
8      s[0] = 'T';
9      printf("%s\n", s);
10     return 0;
11 }
```

fixing the segmentation fault — correctly modifying a string

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char *s = malloc(19*sizeof(char));
7      strcpy(s, " here is a string!");
8      s[0] = 'T';
9      printf("%s\n", s);
10     return 0;
11 }
```

```
$ gcc badStringFixed.c -o badStringFixed
$ ./badStringFixed
There is a string!
```

fixing the segmentation fault — correctly modifying a string

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char *s = malloc(19*sizeof(char));
7      strcpy(s, " here is a string!");
8      s[0] = 'T';
9      printf("%s\n", s);
10     return 0;
11 }
```

```
$ gcc badStringFixed.c -o badStringFixed
$ ./badStringFixed
There is a string!
```

So everything is fine? 🤔

checking for memory leaks with valgrind

```
$ valgrind ./badStringFixed
==30188== ...
==30188==
There is a string!
==30188==
==30188== HEAP SUMMARY:
==30188==      in use at exit: 19 bytes in 1 blocks
==30188==    total heap usage: 2 allocs, 1 frees, 1,043 bytes allocated
==30188==
==30188== LEAK SUMMARY:
==30188==    definitely lost: 19 bytes in 1 blocks
==30188==    indirectly lost: 0 bytes in 0 blocks
==30188==    possibly lost: 0 bytes in 0 blocks
==30188==    still reachable: 0 bytes in 0 blocks
==30188==    suppressed: 0 bytes in 0 blocks
==30188== Rerun with --leak-check=full to see details of leaked memory
```

checking for memory leaks with valgrind

```
$ valgrind --leak-check=full ./badStringFixed
==30209== ...
==30209==
There is a string!
==30209==
==30209== HEAP SUMMARY:
==30209==      in use at exit: 19 bytes in 1 blocks
==30209==    total heap usage: 2 allocs, 1 frees, 1,043 bytes allocated
==30209==
==30209== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==30209==    at 0x4C2BBAF: malloc (vg_replace_malloc.c:299)
==30209==    by 0x108708: main (badStringFixed.c:6)
==30209==
==30209== ...
```

checking for memory leaks with valgrind

```
$ valgrind --leak-check=full ./badStringFixed
==30209== ...
==30209==
There is a string!
==30209==
==30209== HEAP SUMMARY:
==30209==      in use at exit: 19 bytes in 1 blocks
==30209==    total heap usage: 2 allocs, 1 frees, 1,043 bytes allocated
==30209==
==30209== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==30209==    at 0x4C2BBAF: malloc (vg_replace_malloc.c:299)
==30209==    by 0x108708: main (badStringFixed.c:6)
==30209==
==30209== ...
```

line 6 contains: `char *s = malloc(19*sizeof(char));`

fixing the segmentation fault — correctly modifying *and freeing* a string

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char *s = malloc(19*sizeof(char));
7      strcpy(s, " here is a string!");
8      s[0] = 'T';
9      printf("%s\n", s);
10     free(s);
11     return 0;
12 }
```


fixing the segmentation fault — correctly modifying *and freeing* a string

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char *s = malloc(19*sizeof(char));
7      strcpy(s, " here is a string!");
8      s[0] = 'T';
9      printf("%s\n", s);
10     free(s);
11     return 0;
12 }
```

```
$ gcc badStringFree.c -o badStringFree
$ ./badStringFree
There is a string!
```

fixing the segmentation fault — correctly modifying *and freeing* a string

```
$ gcc -g badStringFree.c -o badStringFree
$ valgrind --leak-check=full ./badStringFree
==31190== ...
==31190==
There is a string!
==31190==
==31190== HEAP SUMMARY:
==31190==      in use at exit: 0 bytes in 0 blocks
==31190==    total heap usage: 2 allocs, 2 frees, 1,043 bytes allocated
==31190==
==31190== All heap blocks were freed -- no leaks are possible
==31190==
==31190== ...
```

valgrind: another (bad) example from lecture 1

```
#include <stdio.h>
#include "LibStack.h"

int main(int argc, char *argv[]) {
    Stack stack;
    stack = newStack(2);
    push(42, &stack);
    push(17, &stack);
    push(23, &stack);
    pop(&stack);
    printf("%d\n", pop(&stack));
    return 0;
}
```

valgrind: another (bad) example from lecture 1

```
#include <stdio.h>
#include "LibStack.h"

int main(int argc, char *argv[]) {
    Stack stack;
    stack = newStack(2);
    push(42, &stack);
    push(17, &stack);
    push(23, &stack);
    pop(&stack);
    printf("%d\n", pop(&stack));
    return 0;
}
```

Missing: freeStack(stack) before return 0.

valgrind: another (bad) example from lecture 1

```
#include <stdio.h>
#include "LibStack.h"

int main(int argc, char *argv[]) {
    Stack stack;
    stack = newStack(2);
    push(42, &stack);
    push(17, &stack);
    push(23, &stack);
    pop(&stack);
    printf("%d\n", pop(&stack));
    return 0;
}
```

Missing: freeStack(stack) before return 0.

More about valgrind in Appendix A.4 and A.5.

Always clean up after yourself!

Rule of thumb:

For every `malloc` you should have a corresponding `free`.

Always clean up after yourself!

Rule of thumb:

For every `malloc` you should have a corresponding `free`.

Note: count the number of **executions**, not how often you wrote the command!

```
Stack s;  
while (...) {  
    ...  
    s = newStack(2);  
    ...  
}  
freeStack(s);
```

Always clean up after yourself!

Rule of thumb:

For every malloc you should have a corresponding free.

Note: count the number of **executions**, not how often you wrote the command!

```
Stack s;  
while (...) {  
    ...  
    s = newStack(2); /* inside the loop, multiple allocations! */  
    ...  
}  
freeStack(s); /* outside the loop, only one free! */
```


Always clean up after yourself!

Rule of thumb:

For every `malloc` you should have a corresponding `free`.

Note: count the number of **executions**, not how often you wrote the command!

```
Stack s;  
while (...) {  
    ...  
    s = newStack(2);  
    ...  
    freeStack(s); /* free inside the loop! */  
}
```