Informatics II

Tutorial Session 8

Wednesday, 13th of April 2022

Discussion of Exercises 6 and 7,
Pointers in C, Linked Lists, Stacks, Queues

14.00 – 15.45

BIN 0.B.06

Who Am I?

Agenda

- Repetition and More on Pointers
- Review of Exercise 6 (continued)
  - Linked Lists
- Review of Exercise 7
  - Stacks
  - Queues

Building a Heap: Comprehension Question

Why do we iterate from ⌊n/2⌋ to 1 and not from 1 to ⌊n/2⌋ in the BuildHeap algorithm?

(Does this even matter?)

Why do we iterate only from ⌊n/2⌋ and not until n?

Algo: Heapify(A,i,s)

```
m = i;
l = Left(i);
r = Right(i);
if l≤s ∧ A[l]>A[m] then m = l;
if r≤s ∧ A[r]>A[m] then m = r;
if i≠m then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

Algo: BuildHeap(A)

```
for i = ⌊n/2⌋ to 1 do Heapify(A,i,n);
```

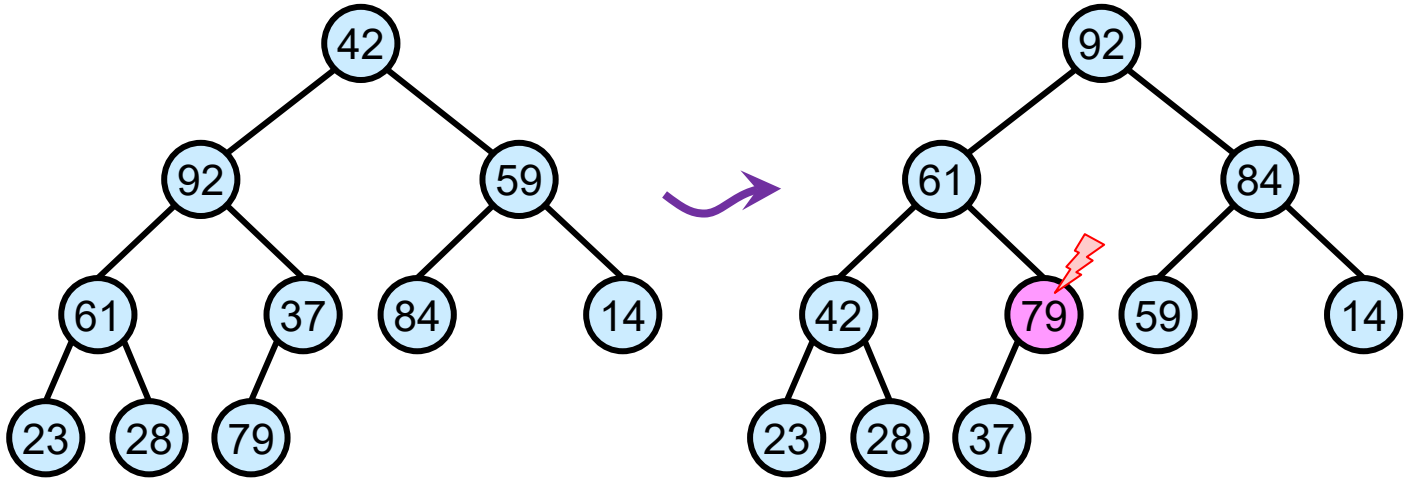
Building a Heap: Comprehension Question

Why do we iterate from ⌊n/2⌋ to 1 and not from 1 to ⌊n/2⌋ in the BuildHeap algorithm?

(Does this even matter?)

The main idea of the BuildHeap algorithm (and a loop invariant of its loop) is that

Counter example: When one tries to max-heapify the adjacent binary tree, an incorrect result will emerge. In the second step, when the number 61 at the second position in the array representation shall be trickled down, this will not happen because both of its children are smaller. Although a grandchild would have been bigger but is not "seen" and never reached



Universität Zürich[UZH]

Institut für Informatik

C Coding Reloaded: Structs, Pointers, Dynamic Memory

- Structs
- Pointers
- Dynamic Memory, Memory Management

Structs: General Remarks

- «Precursors of classes» known from object-oriented programing
  - only fields/attributes, no methods
  - no access control
  - no inheritance, polymorphism etc.

- Keeps things (data) together which belong together.
- Allows to ship all these things together to a function or store in an array together.
- Used for implementing more complicated data structures (than arrays, strings).

Structs: Declaration Syntax

Example: We want to store information about people.

Each person has a name, a date of birth and a height:

```
struct Person {
        char name[100];
        int yearOfBirth;
        float heightInMeters;
};
```

- Note that the declaration has to end with `};`.
- It is not allowed to initialize values within the declaration of a struct.

Structs: Initialization and Access Syntax

- Initialization can be done with curly braces.
- Access to the elements of the struct is achieved using the dot operator.

Example: I want to store the data about my only friend who is named «Hans», was born in 1996 and is 1.76 m tall:

```
struct Person myOnlyFriend = {"Hans", 1997, 1.76};

int currentYear = 2020;
int age = currentYear - myOnlyFriend.yearOfBirth;
printf("My friend only friend %s turns %d this year.\n", myOnlyFriend.name, age);
```

Structs: Using the `typedef` Keyword

The usage of structs can be made a bit more comfortable by using the **`typedef`** keyword. This prevents us from having to type «struct» every time and allows to treat the struct's name like a built-in type:

Without **`typedef`** keyword:

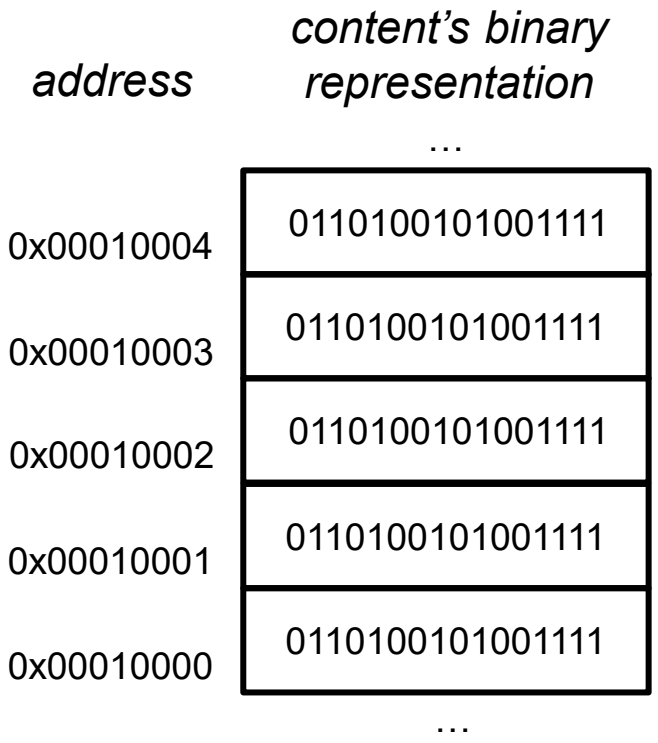
```
struct Point2D {
        int x;
        int y;
};
```

With **`typedef`** keyword:

```
typedef struct {
        int x;
        int y;
} Point2D;
```

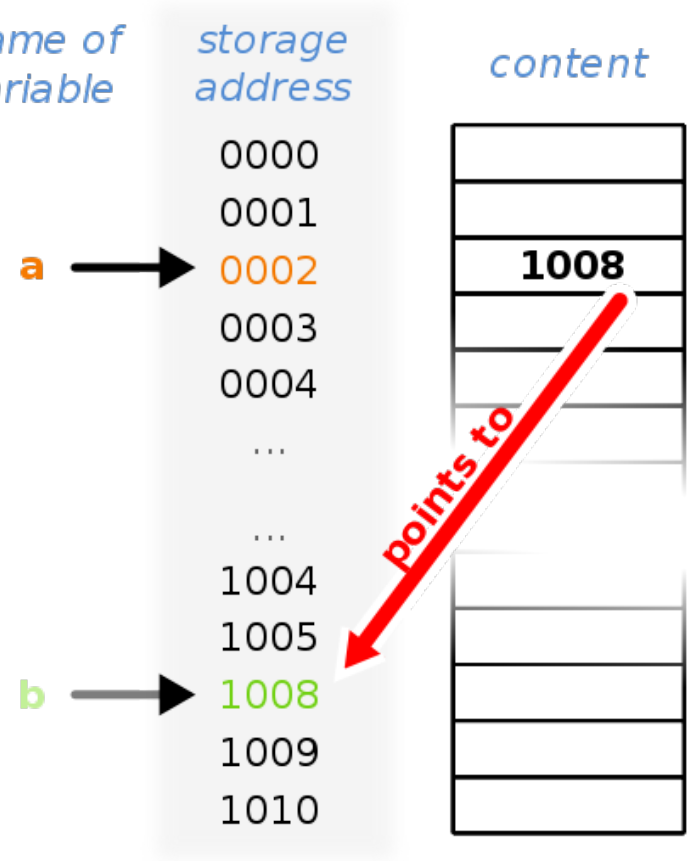
Pointers: Introduction

- Until now, we didn't give much thought to the question at which memory address these contents were stored. We were just referencing the content of memory cells through the name of the respective variable. This will change now. The concept of working with memory addresses is realized in C through pointers.
- You can think of a pointer as a data type which contains a memory address (instead of a particular values). In this sense it points to another location in memory, where something of interest is stored. Accordingly, pointers are often visualized as arrows.
- What kind of variable is (expected to be) actually stored at the location pointed to, is defined in the declaration of the pointer. (Therefore there will be a pointer type to any other data type, e.g. pointer to a integer, pointer to a float etc.)



Pointers: Graphical Perspective / Pointers as Memory Addresses

Example: The variable with name a is a pointer. It contains the address of another memory cell where something of interest is stored.



Pointers: Syntax

- Declare a pointer variable with name **`p`** which is pointing at a memory location where the bit pattern is to interpreted as of type **`int`**:

```
int *a;  or  int* a;
```

- Syntactically, `int * a;` and `int*a;` would also be valid, but are uncommon.
- All kinds of types can be used for pointers, in particular structs. (There is even the special case of `void*` which has special meaning.)
- Beware: `int* x, y, z;` is equivalent to `int *x; int y; int z;`.

Pointers: a Conceptual Perspective

Let's define the following:

- **address(x)** shall denote the memory address where the variable x is stored; takes a variable name as input.
- **content(p)** shall denote the content which is stored at the memory address p; takes a memory address as input.

Let's also use the symbol ← as the assignment operator. Thus when we write x ← 42, we mean that we assign value 42 to the variable x. In programming languages this operator is usually represented by a single equals sign (=) which is distinguished from the sign == which is used to make comparisons.

Pointers: a Conceptual Perspective

Using this definition, how would one have to express the following C code fragments?

```
a = 42;
```

content(address(a)) ← 42

«Assign the value 42 as the content of the memory address where the variable a is stored.»

```
a = b;
```

content(address(a)) ← content(address(b))

Of course, in such statements, there should be the same type on both sides: either a content or a address.

Pointers: Address-Of, Dereferencing

The functions address() and content() also exist in C language:

address(x)	`&x`	address-of operator
content(p)	`*p`	dereference operator

Remarks:

- Note that the same symbol which is used to access the content at a given memory address (i.e. *), is also used to define a pointer.
- Applying the dereference operator on a pointer p and thus accessing the content at the respective memory address is called dereferencing the pointer p.
- **Make sure that you are absolutely confident using those operators and really understand their meaning. Otherwise you will constantly run into trouble when using pointers!**

Pointers: Examples

What will be the output of the following code examples?

Example A:

```
int main(void) {
        int a = 0;
        int* b = a;
        printf("%d", *b);
        return 0;
}
```

→ Segmentation fault: Program tries to access memory address 0 (which means: none, normally: NULL) at line 4.

Example B:

```
int main(void) {
        int a = 0;
        int* b = a;
        printf("%d", &b);
        return 0;
}
```

→ The address at which variable b is stored, interpreted as integer value, will be put to the console.

Exercise: Pointer Declaration and Assignments

Explain what the output of the following C program is:

```c
#include <stdio.h>

int main() {
        int* a;
        int* b;
        int c = 5;
        a = &c;
        c++;
        *a = -10;
        b = &c;
        c = 15;
        *b = *a / 2;
        printf("%d -- %d", *a, *b);
        return 0;
}
```

Pointers and Structs

Oftentimes, we want to dereference a pointer to a struct and afterwards access one of its elements.

Example using the struct **`Person`** from before:

```
struct Person myOnlyFriend = {"Hans", 1996, 1.76};
struct Person* personPtr = &myOnlyFriend;
printf((*personPtr).name);
```

This kind of dereferencing followed by accessing an element of a struct is quite common and therefore a simplified syntax (syntactic sugar) exists for this.

Given a pointer p to a struct with element a, instead of `(*p).a`

we can write the following: `p->a`

Pointers and Arrays

Array variables are actually pointers to the first element of the array.

This explains a lot of things that might have seemed a bit odd until now when with arrays, e.g. the syntax for passing an array to a function and the respective behaviour (pass by reference).

This also means, that for arrays `&(a[0])` is equivalent to `a` and `&(a[i])` is equivalent to `a+i`.

Therefore, we can also write

```
*(a + i)
```

instead of

```
a[i]
```

This leads to the concept of pointer arithmetics.

Call By Value and Call By Reference (Pass By Value and Pass By Reference)

Consider the following code snippet: What will be the output?

```c
#include <stdio.h>

int foo1(int a) {
    a = a % 3 + 5;
    return a;
}

int main() {
    int x = 32;
    int y = foo1(x);
    printf("x = %d\n", x);
    printf("y = %d", y);
    return 0;
}
```

Call By Value and Call By Reference

Consider the following code snippet: What will be the output?

```c
#include <stdio.h>

int foo2(int a[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        a[i] = a[i] % 3 + 5;
        sum += a[i];
    }
    return sum;
}
```

```c
int main() {
    int x[] = {32, 33, 34};
    int y = foo2(x, 3);
    printf("x = %d, %d, %d\n", x[0], x[1], x[2]);
    printf("y = %d", y);
    return 0;
}
```

Call By Value and Call By Reference

(from https://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php)

Another Example: Pass by Value and Pass by Reference

The implementation of a swap function shown below is *wrong* and does not perform what one might expect:

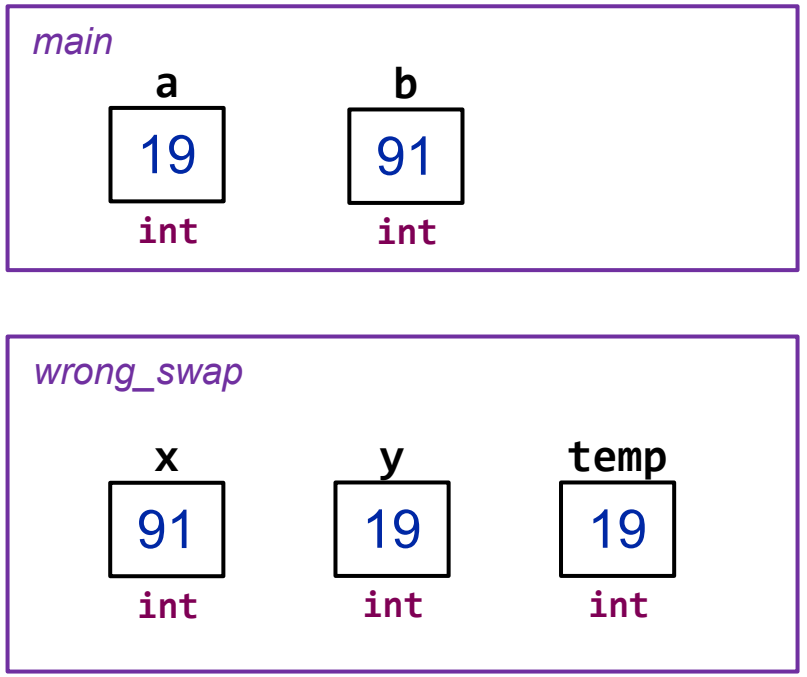
```
1  #include <stdio.h>
2
3  void wrong_swap (int x, int y) {
4      int temp = x;
5      x = y;
6      y = temp;
7  }
8
9  int main() {
10     int a = 19;
11     int b = 91;
12     wrong_swap(a, b);
13     printf("%d", a); /* will print 19 */
14 }
```

Why does this not work correctly and how could this be implemented such that it works?

Another Example: Pass by Value and Pass by Reference

The following implementation of a swap function is *wrong* and does not perform what one might expect:

```
1  #include <stdio.h>
2
3  void wrong_swap (int x, int y) {
4      int temp = x;
5      x = y;
6      y = temp;
7  }
8
9  int main() {
10     int a = 19;
11     int b = 91;
12     wrong_swap(a, b);
13     printf("%d", a); /* will print 19 */
14 }
```



Dynamical Memory Allocation

- Memory can be allocated dynamically using the **`malloc()`** function. The function **`free()`** is used the release the reserved memory for further usage.
- It is important to never forget to release dynamically allocated memory. This is of particular importance in case memory is dynamically allocated within a loop, since if it is not released, memory will shrink at every iteration (memory leak).

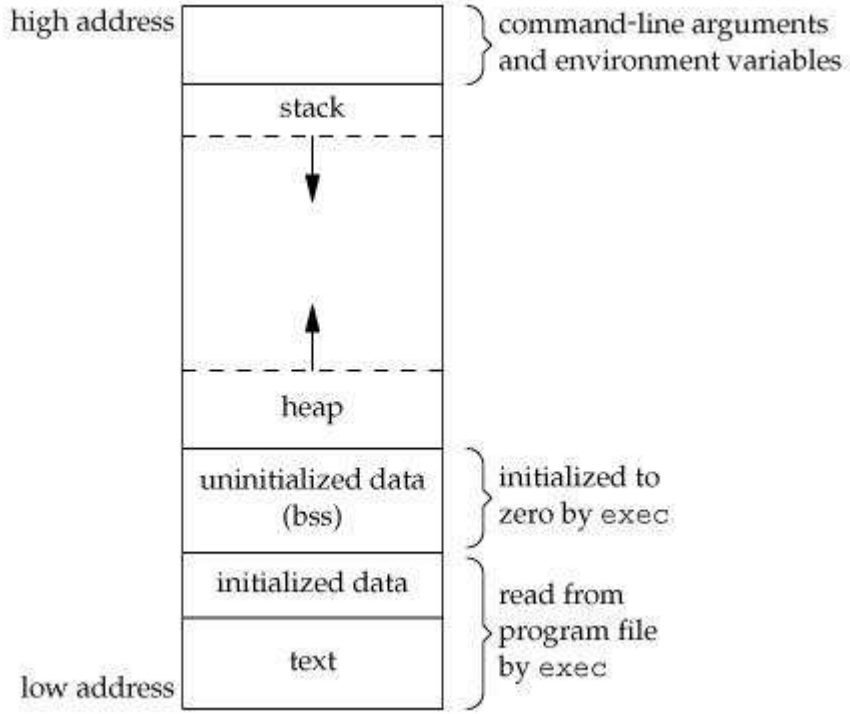
Syntax example:

```
int* pointer;
pointer = malloc(sizeof(int));
*pointer = 11;

printf("%d\n", *pointer);

free(pointer);
```

Memory Layout

Dynamical memory is allocated in a special part in memory called the heap. When a program is in execution in the RAM it has a certain amount of memory available which belongs to different segments:

- **Stack**: statically allocated memory (e.g. recursion)
- **Heap**: dynamically allocated memory
- **Data**: statically allocated data, e.g. arrays, strings declared in the code
  - initialized / uninitialized (BSS)
- **Text**: executable machine instructions (compiled code); read-only



Memory Layout: Example

What problem is lurking in the following C code function which was intended to initialize a struct?

```
struct Point2D {
        int x;
        int y;
};

struct Point2D* initNewPoint() {
        struct Point2D p;
        p.x = 4;
        p.y = 2;
        return &p;
}
```

→ The memory for the struct named `p` is allocated on the stack segment of memory. After the function `initNewPoint` has returned, anything may happen to this memory space since it is no longer regarded as needed afterwards.

Never try to use a variable which you have declared in a function outside of this function! If you need this behaviour, either allocate the respective memory dynamically (i.e. using `malloc`).

Exercise 6: Task 1a: Pointers

State if the following statements about pointers are true or false:

a) "Pointers in C language can only point to primitive data types (e.g., int, double), not non-primitive data types (e.g., struct, array)."

✗ false

b) "The expressions `*ptr++` and `++*ptr` produce the same results."

✗ false `*ptr++` is parsed as `*(ptr++)` while `++*ptr` is parsed as `++(*ptr)`

c) "If we use a primitive data type variable as the argument to a function, and modify that variable inside the function, the original data will be changed."

✗ false

d) "If we use a pointer as the argument to a function, and change the pointer inside the function, the original data will be changed."

✓ true ✗ false

depends on what "original data" refers to…

Reminder: Operator Precedence in C

The operator precedence in C is more or less identical to operator precedence in Python.

C Operator	Type	Associativity
`()`	parentheses (function call operator)	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postincrement	
`--`	unary postdecrement	
`++`	unary preincrement	right to left
`--`	unary predecrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`~`	unary bitwise complement	
`( type )`	C-style unary cast	
`*`	dereference	
`&`	address	
`sizeof`	determine size in bytes	
`*`	multiplication	left to right
`/`	division	
`%`	modulus	
`+`	addition	left to right
`-`	subtraction	
`==`	equals	left to right
`!=`	does not equal	
`===`	strict equals (no type conversions allowed)	
`!==`	strict does not equal (no type conversions allowed)	
`&`	bitwise AND	left to right
`^`	bitwise XOR	left to right
`|`	bitwise OR	left to right
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	conditional	right to left
`=`	assignment	right to left
`+=`	addition assignment	
`-=`	subtraction assignment	
`*=`	multiplication assignment	
`/=`	division assignment	
`%=`	modulus assignment	
`&=`	bitwise AND assignment	
`^=`	bitwise exclusive OR assignment	
`|=`	bitwise inclusive OR assignment	
`<<=`	bitwise left shift assignment	
`>>=`	bitwise right shift with sign extension assignment	
`>>>=`	bitwise right shift with zero extension assignment	

Reminder: Pre-Increment and Post-Increment (and Decrement)

If the value of an integer variable `x` needs to be incremented in C, this can be done by just reassigning using `x = x + 1` or `x += 1`. There are also two convenient operators for this purpose:

- **pre-increment operator**: **`x++`**
- **post-increment operator**: **`++x`**

These operators have (apart from being shorter) the advantage that they can be applied in places where an assignment like `x = x + 1` could not be used, for example when printing a value: `printf("%d", x++)`. In this case, it would be necessary to perform the increment on a separate line.

These two operators are not working in exactly the same manner. As their names indicate, the

- pre-increment operator will increment the value of the variable first, before it is used within the larger code context, and the
- post-increment operator will use the current, unchanged value of the variable first within the code and only then increment it.

Thus, you can think of pre-incrementation and post-incrementation as two-step processes.

Reminder: Pre-Increment and Post-Increment (and Decrement)

pre-increment

++x

increment – use

```
x = ++i;   ≙   i = i + 1;
               x = i;
```

```
int x = 42;
printf("%d, ", ++x);
printf("%d", x);
```

} will print 43, 43

post-increment

x++

use – increment

```
x = i++;   ≙   x = i;
               i = i + 1;
```

```
int x = 42;
printf("%d, ", x++);
printf("%d", x);
```

} will print 42, 43

Exercise 6 – Task 1b: Understanding Pointers: Solution

```c
#include <stdio.h>

int main() {
	char ch = 'a';
	int i = 1;
	double d = 1.2;
	char* p_ch = &ch;
	int* p_i = &i;
	double* p_d = &d;

	printf("The address of ch is %p \n", &ch);
	printf("The address of i is %p \n", &i);
	printf("The address of d is %p \n", &d);
	printf("The address of p_ch is %p \n", &p_ch);
	printf("The address of p_i is %p \n", &p_i);
	printf("The address of p_d is %p \n", &p_d);

	printf("The value of ch is %c \n", ch);
	printf("The value of i is %d \n", i);
	printf("The value of d is %f \n", d);
	printf("The value of p_ch is %p \n", p_ch);
	printf("The value of p_i is %p \n", p_i);
	printf("The value of p_d is %p \n", p_d);

	printf("The size of ch is %lu bytes \n", sizeof(ch));
	printf("The size of i is %lu bytes \n", sizeof(i));
	printf("The size of d is %lu bytes \n", sizeof(d));
	printf("The size of p_ch is %lu bytes \n", sizeof(p_ch));
	printf("The size of p_i is %lu bytes \n", sizeof(p_i));
	printf("The size of p_d is %lu bytes \n", sizeof(p_d));

	return 0;
}
```

Exercise 6 – Task 1a: Understanding Pointers: Solution

Sample output:

```
The address of ch is 0xffffcc3f
The address of i is 0xffffcc38
The address of d is 0xffffcc30
The address of p_ch is 0xffffcc28
The address of p_i is 0xffffcc20
The address of p_d is 0xffffcc18
The value of ch is a
The value of i is 1
The value of d is 1.200000
The value of p_ch is 0xffffcc3f
The value of p_i is 0xffffcc38
The value of p_d is 0xffffcc30
The size of ch is 1 bytes
The size of i is 4 bytes
The size of d is 8 bytes
The size of p_ch is 8 bytes
The size of p_i is 8 bytes
The size of p_d is 8 bytes
```

Additional Exercise: Changing Pointers

What will be the output of the adjacent C program?

A: Does not compile

B: Run time error

C: `3 -- 7`

D: `21 -- 3`

E: `21 -- 7`

F: `21 -- 21`

Answer **E** is correct. Note that the function `foo` only gets a *copy* of the address of variable `b` and thus can only change the *value* of `b` but not its address (i.e. the *value* of variable `b` is passed by reference but not its address).

```
1  #include <stdio.h>
2  int my_global = 3;
3
4  void foo(int* input_ptr) {
5          my_global = *input_ptr * my_global;
6          input_ptr = &my_global;
7          printf("%d -- ", *input_ptr);
8  }
9
10 int main() {
11         int a = 7;
12         int* b = &a;
13         foo(b);
14         printf("%d", *b);
15         return 0;
16 }
```

Additional Exercise: Pointer to Pointer to Pointer to…

Explain what the output of the following C program is:

```c
#include <stdio.h>

int main() {
        int a = 42;
        int* b = &a;
        int** c = &b;
        *(*c) = *(*c) * 2;
        int*** d = &c;
        int x = ***&*d;
        printf("%d", x);
        return 0;
}
```

→ The output will be 84.

Additional Exercise: Pointer as a Return Type

Explain what the output of the following C program is:

```c
#include <stdio.h>

double* foo(double* a, int* b) {
        *a = *a * *b;
        return a;
}

int main() {
        double x = 2.5;
        int y = 8;
        double* z = foo(&x, &y);
        x = *z;
        printf("%f", x);
        return 0;
}
```

→ The output will be 20.000000.

Additional Exercise: Pointers and Arrays, Pointer Arithmetic

Explain what the output of the following C program is:

```c
#include <stdio.h>
int main() {
      double x[10];
      double* pt;
      double* qt;
      pt = x;
      *pt = 0;
      *(pt + 2) = 1.61803;
      pt[5] = 2.5;
      pt = x + 2;
      qt = pt;
      *qt = 0.57721;
      qt[4] = 3.5;
      *(x + 8) = 6.7;
      pt = x;
      qt = x + 10;
      printf("%d\n", qt - pt);
      return 0;
}
```

Additional Exercise: Structs and Pointers

Explain what the output of the following C program is:

```c
#include <stdio.h>

typedef struct my_struct {
      int* a;
      char b;
      double c[4];
      struct my_struct* d;
} my_struct;

int main() {
      int pos1 = 24;
      int pos2 = pos1;

      my_struct x = {
            NULL,
            '0',
            {1.1, 2.2, 3.3, 4.4},
            NULL
      };
      x.a = &pos1;
      x.b = 'X';

      my_struct* y;
      y->a = &pos2;
      *(y->a) = (*(x.a))++;
      y->b = 'Y';
      y->d = &x;

      my_struct z[15];
      z[7] = x;
      z[9] = *y;

      my_struct* w[15];
      w[7] = &x;
      w[9] = y;

      printf("%c\n", z[9].d->b);
      printf("%c\n", w[9]->d->b);

      return 0;
}
```

Additional Exercise: In The Beginning…

Explain what the following code does and why it is problematic.

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
     int* counter = malloc(sizeof(int));
     for (int i = 0; i < 100; i++) {
          if (i % 2 == 0) {
               *counter = *counter + 1;
          }
     }
     printf("%d", *counter);
     return 0;
}
```

Additional Exercise: Freeing Memory

What will be the result when trying to compile and run the adjacent C program?

A: Does not compile because of line 12

B: Does not compile because of line 14

C: Does not compile because of lines 12 and 14

D: Runtime error after printing 2

E: Runtime error after printing 3

F: Runtime error before printing 5

G: Runtime error after printing 5 and before printing 9

H: No errors, runs and terminates normally

Answer **F** is correct.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5        int a = 1;
6        int* b = &a;
7        int* c = malloc(sizeof(int));
8        *c = 3;
9        printf("%d\n", a + *b);
10       free(c);
11       printf("%d\n", *c);
12       free(b);
13       printf("5\n");
14       free(&a);
15       printf("7\n");
16       free(c);
17       printf("9\n");
18       return 0;
19 }
```

Additional Exercise: Don't Lose It!

The adjacent C program contains two (and a half) severe coding errors. Find them.

→ There is a memory leak both on lines 9 and 10.

And furthermore, memory allocated for variable c is never freed.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5       double* a = malloc(sizeof(double));
6       double b = 100.0;
7       double* c = malloc(sizeof(double));
8       *a = b * 2.0;
9       a = NULL;
10      c = &b;
11      printf("%f\n", *c);
12      free(a);
13      double d = 50.0;
14      printf("%f\n", d);
15      return 0;
16 }
```

Additional Exercise: The Sword of Damocles

The adjacent C program might lead to problems which are difficult to debug, hard to understand outputs and even crashes at runtime. Explain why this is the case.

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int* a = malloc(sizeof(int));
    int b = 17;
    *a = b;
    printf("%d\n", *a);
    free(a);
    int counter = 0;
    for (int i = 0; i < 100; i++) {
        int* c = malloc(sizeof(int));
        *c = i * *a;
        if (*c % b != 0) {
            counter++;
        }
        free(c);
    }
    printf("%d", counter);
    return 0;
}
```

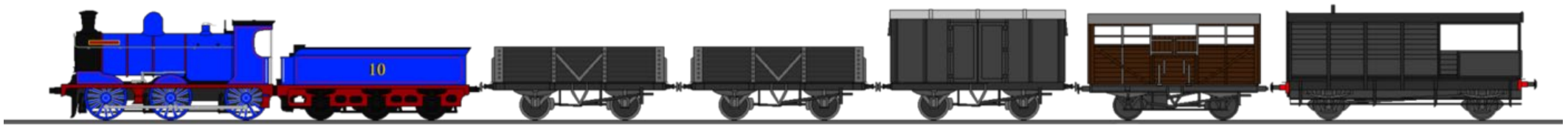
Universität Zürich[UZH]

Institut für Informatik

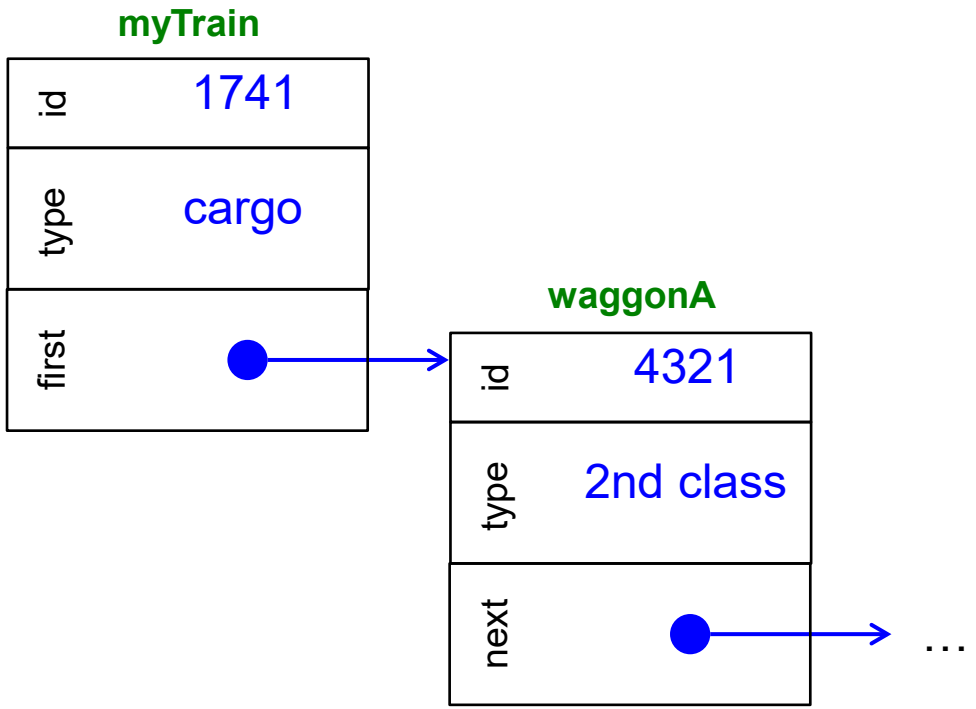
Linked Lists

Linked Lists: Review

– Operations on linked lists like inserting or deleting elements etc. are best visualized using graphical depictions. The can be thought of like assembling different trains…



Linked Lists: Review

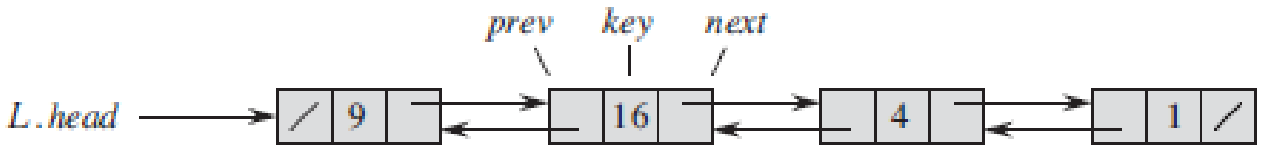



```
struct train {
  struct waggon* first;
};

struct waggon {
  int id;
  int type;
  struct waggon* next;
};
```

Types of Linked Lists

There is a whole bunch of different flavours of linked lists:

- Singly linked list with tail pointer
- Doubly linked list
- Circular linked list (singly, doubly, with/without tail pointer)
- …



Exercise: Asymptotic Complexities of ADT Operations

What is the smallest reachable worst-case asymptotic time complexity for each combination of operation and data structure? (Assume canonical properties of all involved data structures.)

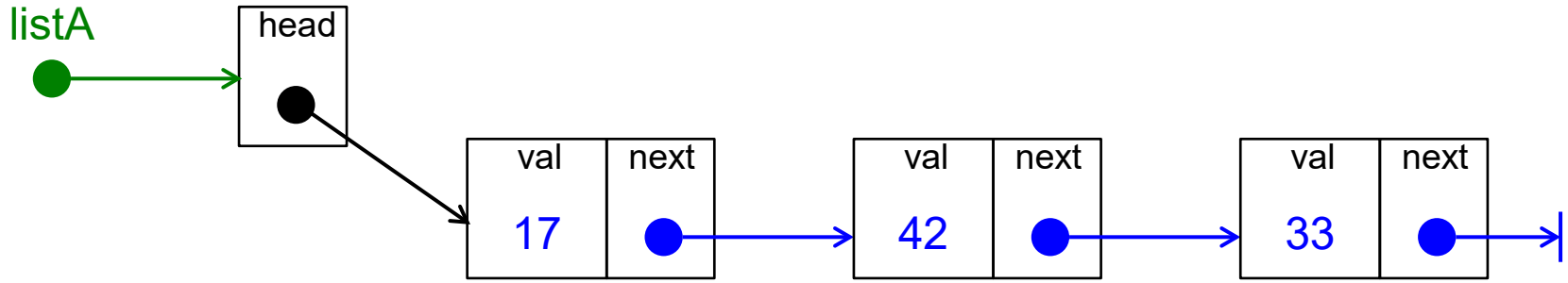
Operation \ *Data structure*	*Ascendingly sorted singly-linked list*	*Ascendingly sorted doubly-linked list*	*Ascendingly sorted array*
finding the smallest element	O(1)	O(1)	O(1)
finding the largest element	O(n)	O(1) (using tail pointer)	O(1)
searching for a given element	O(n)	O(n)	O(log(n)) (using binary search)
deleting a found element (i.e. not including previous search)	O(1) (assuming a reference to the previous node is available; otherwise O(n))	O(1)	O(n)
finding the median	O(n)	O(n)	O(1)

Linked Lists: Conceptual Questions

- Why should we even bother to use linked lists, why not just work with arrays?
- What is the difference between singly linked lists and doubly linked lists? When should we use which?
- What is a tail pointer and why / when is it useful?
- What are linked lists good for / what are real-world applications of linked lists?
- What are advantages and disadvantages of using linked lists when compared to arrays?

Linked List: Implementation in C

```
struct list {
  struct node* head;
};

struct node {
  int val;
  struct node* next;
};
```



Why do we need the additional struct „list“? Isn‘t this unnecessary? After all, we could just set a pointer to the first element to reference a list…

Toolkit for Building Algorithms Working on Linked Lists

- Iterating over a linked list
- Using a second pointer («walker and runner» / «rabbit and turtle»)

Blueprint for Iterating over a Linked List

In almost every problem, it will sooner or later be necessary to visit all nodes (or a subset of the nodes) in the linked list. This is usually done using a while loop which constantly checks whether a `NULL` pointer is encountered. The current pointer is advanced by reassigning it the next pointer of the current node.

```
while (current != NULL) {
        //do stuff
        current = current->next;
}
```

Remarks:

– The iteration can also be done using a for loop:

```
for (current = head; current != NULL; current = current->next) { … }
```

– If there is a tail pointer, it is also possible to look for that instead.

Additional Exercise for Linked Lists Implementation

a) Given a Linked List, write a function which takes the Linked List as a parameter and returns a struct with the number of even and odd elements in it.

b) Given a Linked List which might or might not contain a cycle (i.e. a node which points to itself or one of ist predecessors), write a function which returns true if there is a cycle and returns false if there isn't.

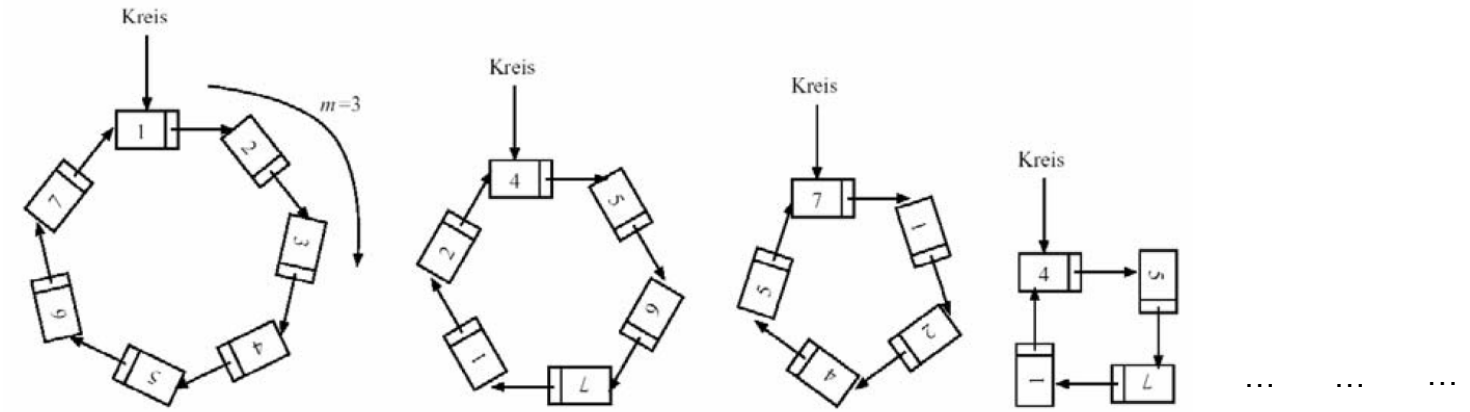
c) Josephus problem (see next slide).

Additional Exercise for Linked Lists Implementation: Josephus Problem

Der Geschichtsschreiber Josephus Flavius berichtet, dass er im Jahre 67 n. Chr. zur Zeit der jüdischen Rebellion gegen die Römer zusammen mit 40 Aufständischen in eine Höhle geflüchtet sei während des Kampfes um die Stadt Yodefat in Galiläa. Um dem Feind nicht in die Hände zu fallen, wollten sich die Rebellen gegenseitig selbst umbringen. Josephus war dagegen, konnte die anderen jedoch nicht umstimmen. Um dem Tod zu entgehen, schlug er vor, man solle sich im Kreis aufstellen und abzählen. Jeder Dritte solle auf der Stelle umgebracht werden. Sein Vorschlag wurde akzeptiert. Josephus stellte sich so in den Kreis, dass er als Letzter übrig und damit am Leben blieb.

Das Problem des Josephus lässt sich mit einfach verketteten Ringlisten simulieren. Entwerfen Sie eine entsprechende Datenstruktur und schreiben Sie ein Programm, das die Reihenfolge ausgibt, in der die Personen gewählt werden. Das Programm soll als Eingabe die Anzahl n der Personen erhalten, die sich am Anfang im Kreis befinden, sowie die Zahl m, um die weitergezählt wird.

Die untenstehende Grafik zeigt am Beispiel n = 7, m = 3 wie sich die Datenstruktur entwickeln soll:



Ausgabe: 3 6 2 7 5 1 4 = Josephus

Exercise 6, Task 2: Grouping Linked Lists

There is a linked list of N nodes. Each node contains an integer value. Given the head to the linked list, create a new linked list that contains the odd values of the original linked list, followed by the even values.

Example:

a) Implement the insertList function in C, which inserts a new element at the end of a list.

b) Implement the displayList function in C, which prints a given linked list.

c) Give a pseudocode of a function groupingLinkedList, which groups a linked list such that it contains the odd values of the original linked list, followed by the even values as described above.

d) Implement your pseudocode for grouping a linked list in C.

e) State if the following statements are true or false, justify why:

(i) “The space complexity of the function groupingLinkedList is O(1).”

(ii) “The time complexity of the function groupingLinkedList is O(n2).”

Exercise 6, Task 3: Anagrams Validation

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, using all the original letters exactly once. For example, below and elbow are anagrams of each other, dusty and study are anagrams of each other.

Consider each word is represented by a linked list, then the task is to validate if both linked lists are anagram to each other. This task involves three sub-tasks:

a) Write a pseudocode for this task.

b) Implement a linked list in C with displayList and insertList functions, such that each node in the list represents a letter in a word.

c) Implement the anagram's validation function in C.

Exercise 6, Tasks 2 and 3: Remark

In the sample solution dynamically allocated memory is never (explicitly) deallocated (freed). While the memory is indeed freed when the process halts (i.e. main returns) it is recommended that you actually do this explicitly. Whenever you see a program which contains `malloc` but does not contain `free`, this should raise a red flag.

Universität Zürich[UZH]

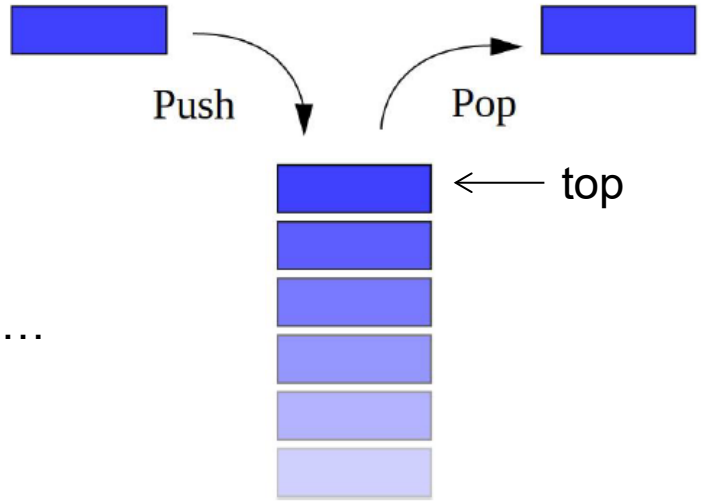
Institut für Informatik

Stacks and Queues

Overview: Stacks and Queues

Stack	Queue
	
LIFO	FIFO
– push(x) – pop()	– enqueue(x) – dequeue()

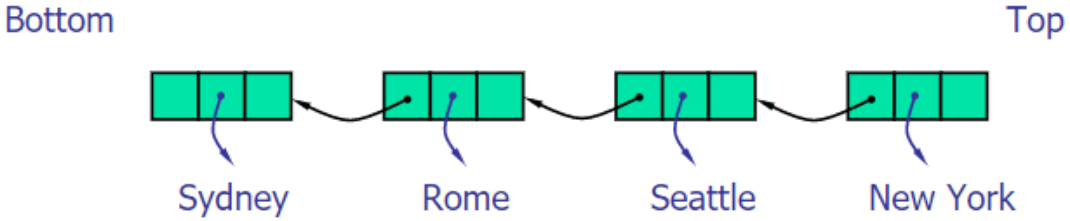
Stacks



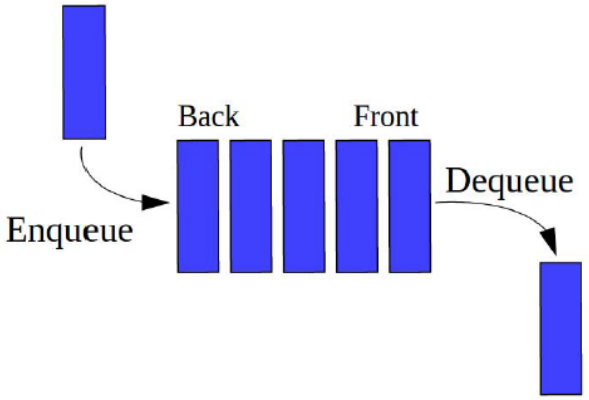
- Some use cases:
  - page-visited history in a web browser, undo sequence in a text editor, parentheses checking, iterative implementation of recursive algorithms, …
  - auxiliary data structure for many algorithms, e.g. depth-first search, …
- Typical operations: size(), isEmpty(), push(object), top(), pop(), peek()

Array-based implementation:

Implementation using linked list:



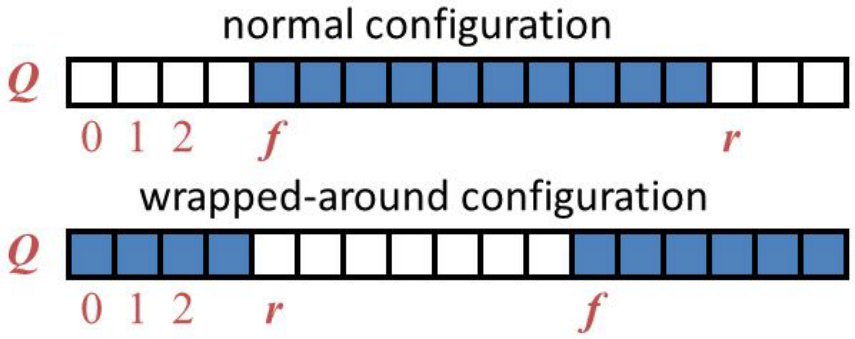
Queues



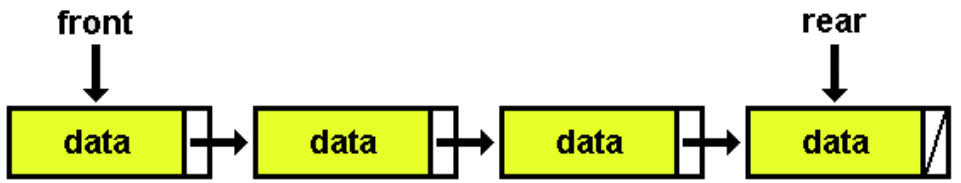
- Some use cases:
  - waiting lists, access to shared resources (e.g. round robin schedulers), …
  - auxiliary data structure for many algorithms, e.g. breath-first search, …

- Typical operations: size(), isEmpty(), enqueue(object), dequeue(), front(), back()

Array-based implementation:



Implementation using linked list:



Stack: Implementation in C using Dynamic Array

- In the `main` function, dynamically allocate memory for a struct of type `stack` on the heap segment of memory. (Don't forget to free memory at the end of the `main` function.)
- Use a preprocessor directive to define the initial size of the stack:

```
#define INITIAL_STACK_SIZE 5
```

- In the `initialize` function, dynamically allocate memory for an integer array of size `INITIAL_STACK_SIZE`. Assign the pointer to the location in dynamic memory to the field `elements`.

```
typedef struct stackADT {
    int *elements;
    int size;
    int count;
} stack;
```

(address of first element of an) integer array containing the contents of the stack

current size of the dynamical array `elements`

number of values currently stored in the stack

Code Example Stacks

Consider the following (partial) C code showing some stack operations. What will be the output?

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct stackADT {
    int* elements;
    int size;
    int count;
} stack;

void initialize(stack* s);
int pop(stack* s);
int push(stack* s, int value);
int peek(stack* s);
int size(stack* s);

/* implementation of stack operations
   not shown here... */
```

```c
int main() {
    stack* myStack = malloc(sizeof(stack));
    initialize(myStack);
    push(myStack, 7);
    push(myStack, 10);
    printf("%d ", peek(myStack));
    printf("%d ", pop(myStack));
    push(myStack, 3);
    push(myStack, 5);
    printf("%d ", pop(myStack));
    printf("%d ", size(myStack));
    printf("%d ", peek(myStack));
    push(myStack, 8);
    printf("%d ", pop(myStack));
    printf("%d ", pop(myStack));
    free(myStack);
    return 0;
}
```

Solution: `10 10 5 2 3 8 3`

Code Example Queues

Consider the following (partial) C code showing some queue operations. What will be the output?

```c
#include <stdlib.h>
#include <stdio.h>
#define QUEUE_SIZE 16

typedef struct queueADT {
    int elements[QUEUE_SIZE];
    int head;
    int count;
} queue;

void initialize(queue* q);
int enqueue(queue* q, int value);
int dequeue(queue* q);
int size(queue* q);

/* implementation of queue operations
   not shown here... */
```

```c
int main() {
    queue* myQueue = malloc(sizeof(queue));
    initialize(myQueue);

    for (int i = 1; i <= 6; i++) {
        enqueue(myQueue, i);
    }
    for (int i = 0; i < size(myQueue); i++) {
        printf("%d ", dequeue(myQueue));
    }
    printf("%d ", size(myQueue));

    free(myQueue);
    return 0;
}
```

Solution: 1 2 3 3

Beware: One might get lured into answering «1 2 3 4 5 6 0» which is **wrong**. Note that the upper bound of the second `for` loop (`size(myQueue)`) is not a constant but continuously changes throughout the iterations of the `for` loop because items are removed from the queue within the loop.

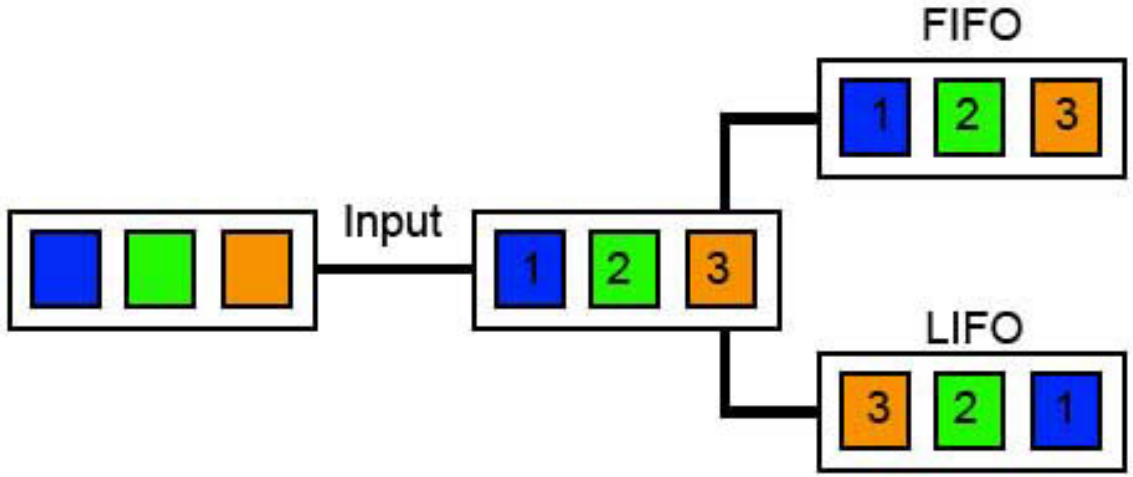
Conceptual Questions ADTs

– Why is it *not* a good idea to implement heapsort using a linked list?

– Which abstract data type is best suited to manage the current standings on the display in the finish area of a sports contest?

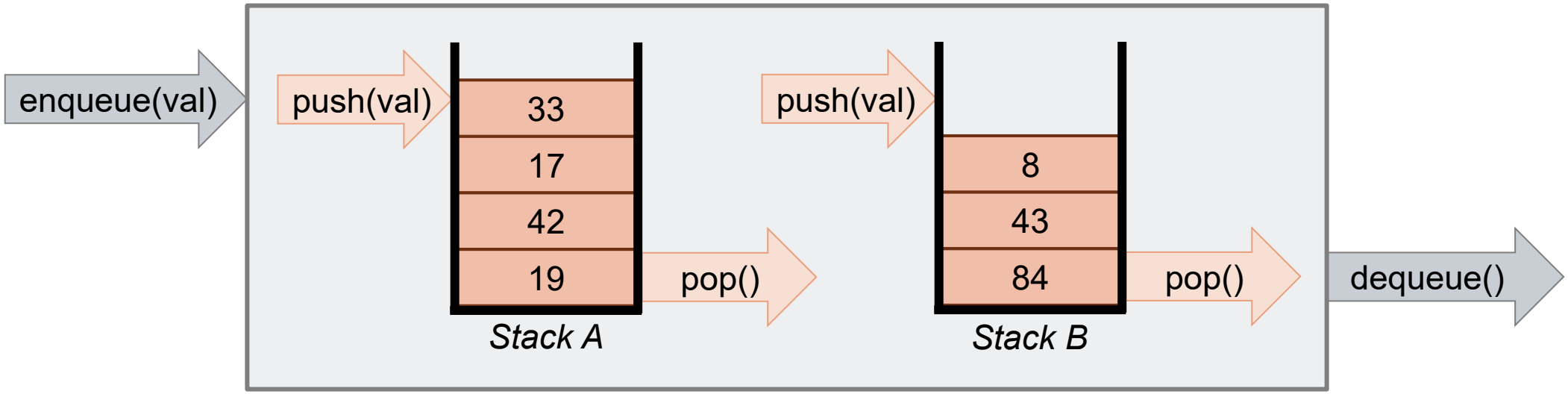
LIFO vs. FIFO

- *LIFO:* The output sequence is reverse of input sequence.
- *FIFO:* The output sequence is the same as the input sequence.



Additional Exercise: Implementing a Queue Using Two Stacks

Task: Create a queue which is based on two stacks (i.e. wrap a queue interface around a pair of stack interfaces).



Additional Exercise: Implementing a Stack Using Other ADTs

How would you implement a stack using…

a) …two queues?

b) …a (single) queue?

How would you implement a stack using a linked list?

How would you implement a stack using a binary heap?

Stacks and Queues: Additional Practice

Additional exercises for practice can be found here:

https://h5p.org/node/471454

Exercise 7, Task 1a

$S_1 = [3, 5]$ (where item 3 is at the top)
$S_2 = [7]$
$Q_1 = [4]$
$Q_2 = [\ ]$ (empty)

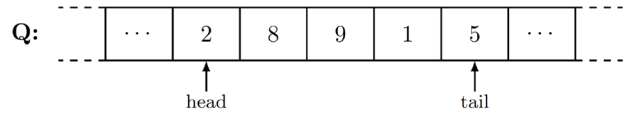
Consider the operations $push(S, x)$ which inserts item x into the stack S and the operation $pop(S)$ which removes the item at the top of the stack S and returns it. Further consider the operations $enqueue(Q, x)$ which inserts item x at the head of queue Q and the operation $dequeue(Q)$ which removes the item at the tail of Q and returns it.
The following sequence of operations are performed on two initially empty stacks S_1 and S_2 and two initially empty queues Q_1 and Q_2:

(1) $push(S_1, 5)$
(2) $push(S_2, 7)$
(3) $push(S_1, 3)$
(4) $enqueue(Q_1, 2)$
(5) $enqueue(Q_1, 4)$
(6) $enqueue(Q_2, 1)$
(7) $push(S_1, dequeue(Q_1))$
(8) $enqueue(Q_2, pop(S_2))$
(9) $push(S_2, 6)$
(10) $push(S_1, pop(S_1) + pop(S_2))$
(11) $enqueue(Q_2, pop(S_1) + dequeue(Q_2))$
(12) $push(S_2, dequeue(Q_2))$
(13) $dequeue(Q_2)$

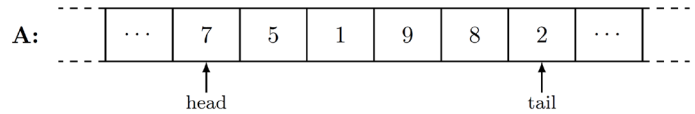
→ 9

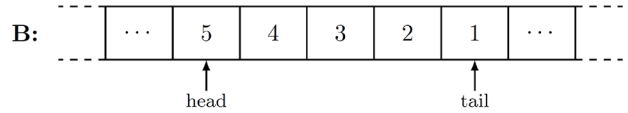
Exercise 7, Task 1b

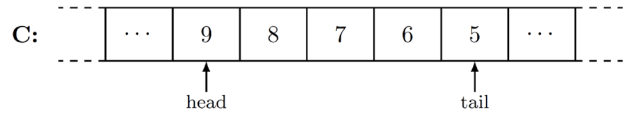
Consider the following queue Q:

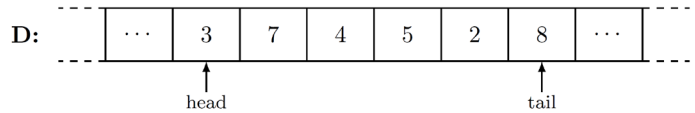


Which of the following queues could be transformed into queue Q *after less than ten* enqueue and / or dequeue operations? Any values are allowed in the enqueue operations.









→ **A, D**

Exercise 7, Task 1c

Consider the following state of a stack S at some point in time (where the leftmost item is the top of the stack): $S = [7, 3, 5, 1]$
Which of the following are possible sequences of operations which were performed *immediately before* the situation shown above has arisen?

A: $pop(), pop(), push(2), pop(), push(7)$

B: $pop(), pop(), push(4), push(1), pop(), push(3), push(7)$

C: $push(7), pop(), push(5), pop(), push(3), pop()$

D: $push(5), push(7), push(3), pop(), pop()$

E: $push(pop()), push(pop()), push(pop()), push(pop())$

→ **A, C, E**

Exercise 7, Task 1d

Consider a queue Q which currently contains n distinct integers as items. Assume you want to remove the all items from Q which are divisible by 3. The other elements in the queue shall be in the exact same order as before after the removal of these elements and you may not use another storage for the items other than Q and a single helper variable.
How many enqueue and dequeue operations are minimally required in the best case and worst case to achieve this task?

Best case: n operations (if all items are divisible by 3)
Worst case: $2n$ operations (if no item is divisible by 3)

Algorithm: remove_divisible_three($queue, n$)

```
1 for i = 1 to n do
2     temp = dequeue(queue)
3     if temp mod 3 ≠ 0 then
4         enqueue(queue, temp)
```

Exercise 7, Task 1e

Consider a sequence of $m > 0$ stacks $(S_1, \ldots, S_m)$ and a sequence of $n > 0$ queues $(Q_1, \ldots, Q_n)$. All of the stacks and all of the queues have an identical size (capacity) of c. An input sequence of k mutually distinct integers is processed as items by these sets of m stacks and n queues in the following way:

(i) At first, any item from the input sequence is pushed to stack S_1.

(ii) Whenever a stack S_i is full, all its items are popped and then immediately pushed to the next stack S_{i+1} until the stack S_i is empty.

(iii) When the last stack in the sequence S_m is full, all its items are popped and then immediately enqueued to the first queue in the sequence Q_1 until the stack S_m is empty.

(iv) Whenever a queue Q_i is full, all its items are dequeued and then immediately enqueued to the next queue Q_{i+1} until the queue Q_i is empty.

(v) When the last queue in the sequence Q_n is full, all its items are dequeued and constitute, in the order that they were dequeued, the output sequence.

For which values of m, n, k and c will the output sequence be identical to the input sequence?

$m \geq 2$, $m \mod 2 = 0$, $n > 0$, $k = d \cdot c$ where d is a positive integer

Exercise 7, Task 2

Consider the given code skeleton `task2_skeloton.c` which contains a starting point for the implementation of a stack in C. In particular, the skeleton contains a `struct` which shall be used for the implementation and is also shown below.

```
typedef struct Stack {
    unsigned int capacity;
    int* items;
  int top;
} Stack;
```

Expand the given skeleton and implement in C the following functions. Test your implementation with appropriate calls in the `main` function.

Exercise 7, Task 3

Consider an array $A[0..n-1]$ of n integers. The span $s(A,i)$ of array element $A[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and such that $A[j] \leq A[i]$. For example, array $A = [6,3,4,5,2]$ has the spans $s(A,0) = 1$, $s(A,1) = 1$, $s(A,2) = 2$, $s(A,3) = 3$ and $s(A,4) = 1$ which can be written as an array as follows: $s(A) = [1,1,2,3,1]$.

Exercise 7, Task 4

Consider an initially empty stack S for which a number of n *push* operations and n *pop* operations is performed. Every time when an item is pushed to S this item is stored at the same time at the left-most (towards to the begining) of array In and every time when an item is popped from S this item is stored at the same time at the left-most of array Out.
The goal of this task is to devise and analyse an algorithm which takes two arrays $In[0..n-1]$ and $Out[0..n-1]$ as paramters and decides whether they could potentially be the result of such *push* and *pop* operations. For example, consider $In = [1, 2, 3, 4, 5]$ and $Out = [4, 5, 3, 2, 1]$. These two arrays can be the result of *push* and *pop* operations as follows:

(1) $push(1)$
(2) $push(2)$
(3) $push(3)$
(4) $push(4)$
(5) $pop() \rightarrow 4$
(6) $push(5)$
(7) $pop() \rightarrow 5$
(8) $pop() \rightarrow 3$
(9) $pop() \rightarrow 2$
(10) $pop() \rightarrow 1$

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

– Summary

Outlook on Next Thursday's Lab Session

Coming up next: spring break

Next tutorial: Wednesday, 27.04.2022, 14.00 h, BIN 0.B.06

Topics:

- Review of Exercise 8
- Binary Search Trees
- (Preview on Exercise 9)
- …
- … (your wishes)

Questions?

Thank you for your attention.