Tutorial Session 6

Friday, 27th of March 2020

Discussion of Exercise 5, Preview on Exercise 6

Trees, Heaps & Heapsort, Quicksort, Pointers

14.15 – 15.45

Y35-F-32

Agenda

- Recap: Trees
- Recap: Heaps
- Recap: Quicksort
- Discussion of Exercise 5 (interleaved)
- Pointers and Structs
- Preview on Exercise 6

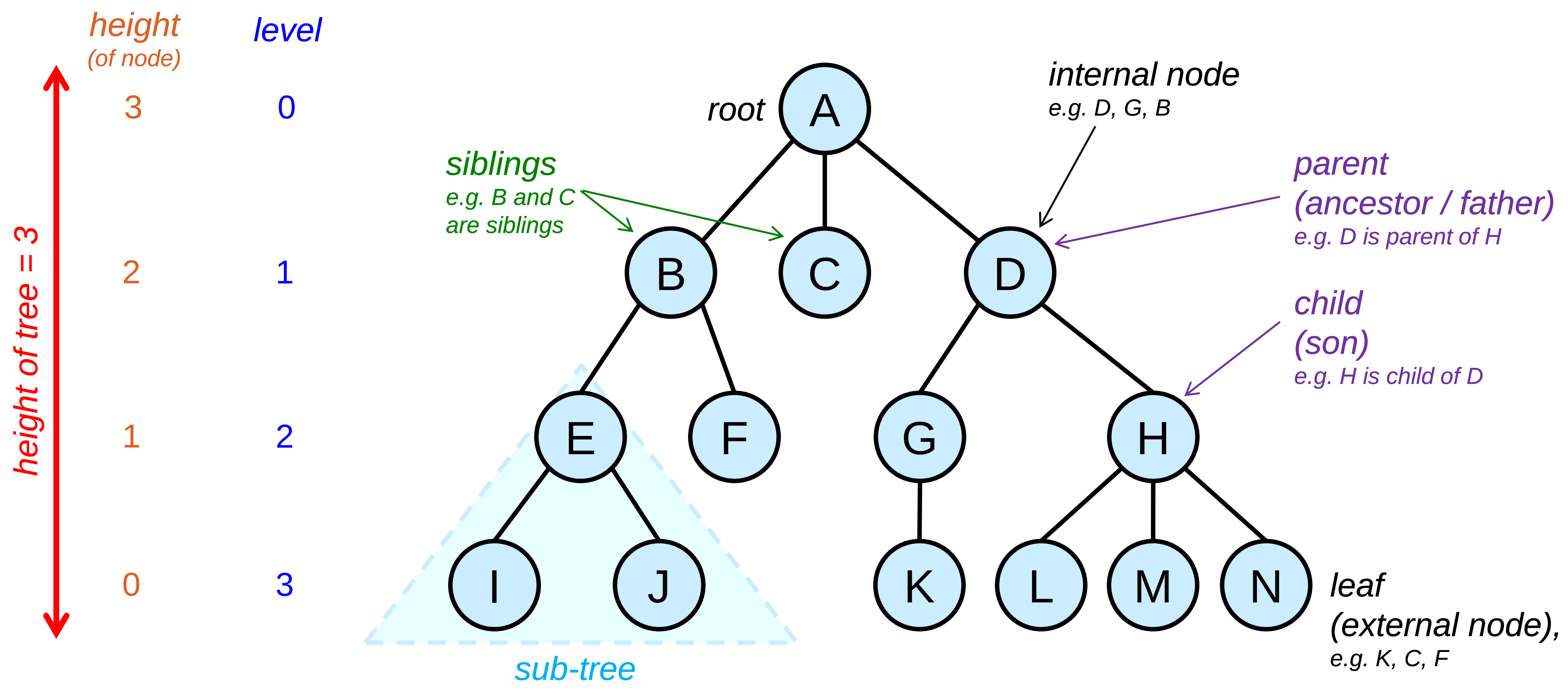
Universität Zürich^UZH

Institut für Informatik

Trees

– Nomenclature

Trees: Nomenclature



Universität Zürich^UZH

Institut für Informatik

Heaps

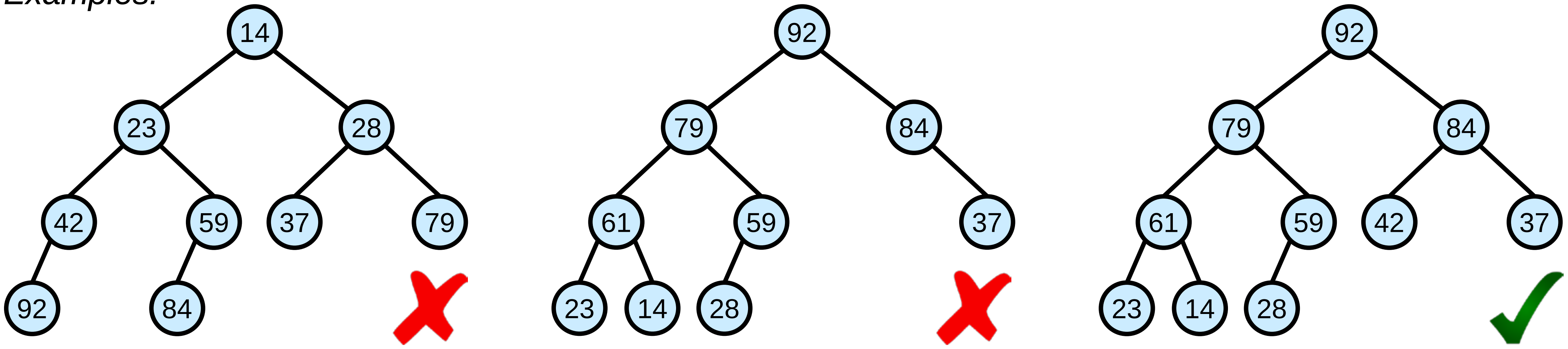
- Heap Conditions
- Representation of a Heap as an Array
- Heap Operations
- Heapsort

Heap Conditions

– A (binary) heap is a binary tree.

– All levels except the lowest are fully filled and the lowest level is occupied starting from left without gaps (= «nearly complete tree»).

Examples:

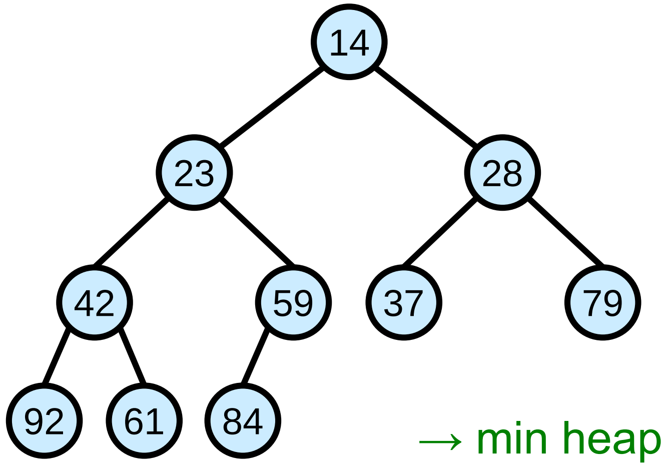


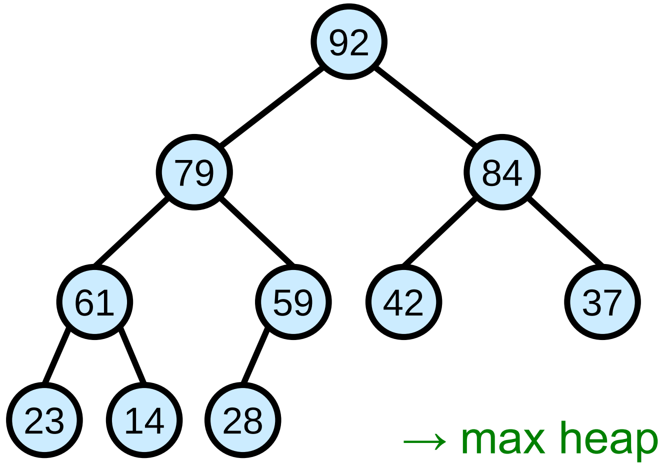
Heap Conditions

- The keys stored in the nodes of the heap are ordered:
  - in a max-heap: for all nodes, the key is bigger than or equal to all its children

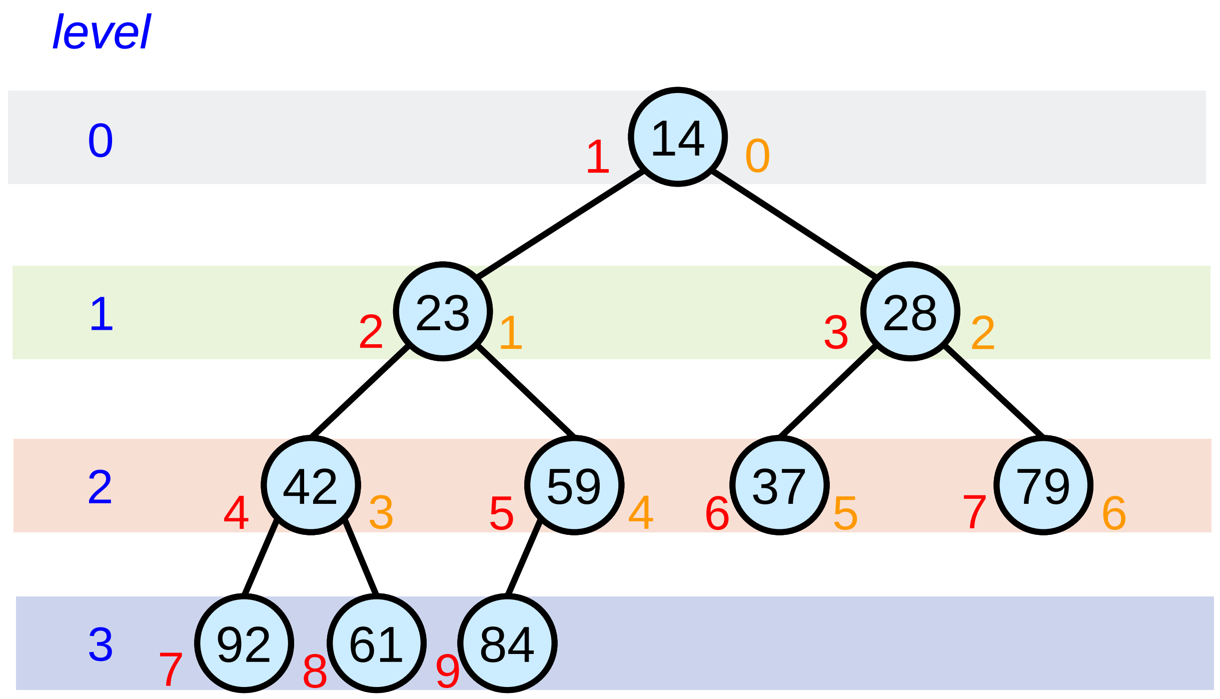
    → largest element at root
  - in a min-heap: for all nodes, the father node is always smaller than or equal to his sons

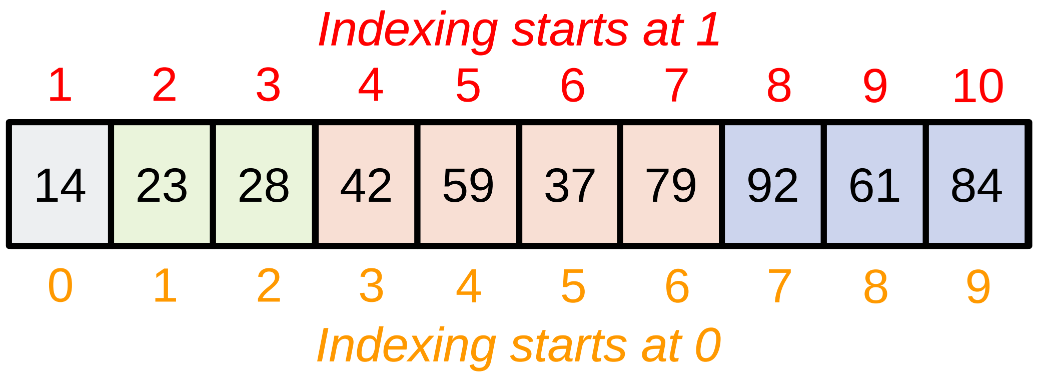
    → smallest element at root





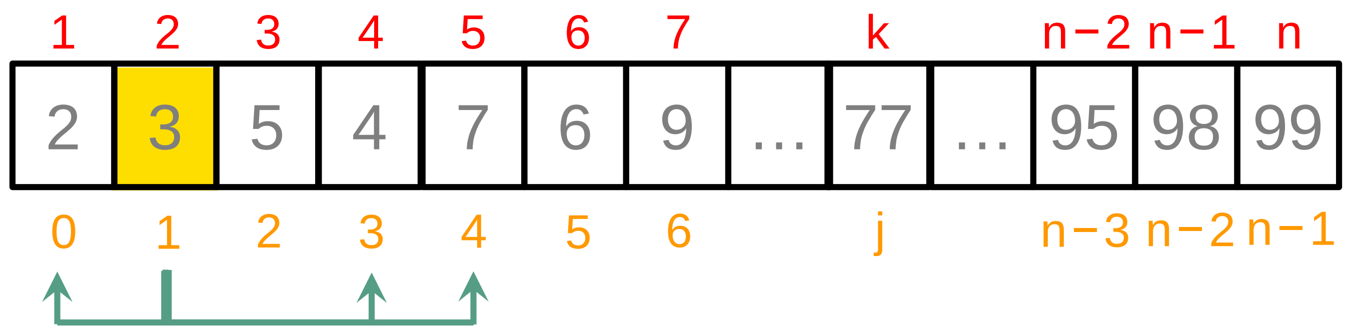
Representation of a Heap as an Array

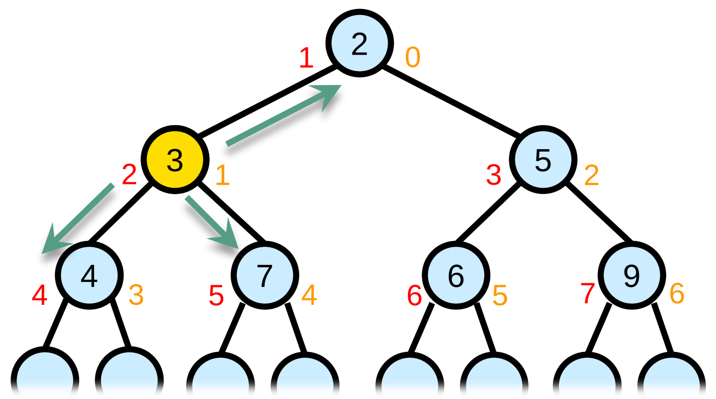




Note: This kind of tree traversal is called level order.

Addressing Parent and Child Nodes





	Indexing starts at 1	*Indexing starts at 0*
Getting to parent of node with index k	$\lfloor k / 2 \rfloor$	$\lfloor (j - 1) / 2 \rfloor$
Getting to left child of node with index k	$2 \cdot k$	$2 \cdot j + 1$
Getting to right child of node with index k	$2 \cdot k + 1$	$2 \cdot j + 2$

Heap Conditions: Exercise

The following arrays shall each represent a binary heap. Do they fulfill the conditions of a max-heap?

A:

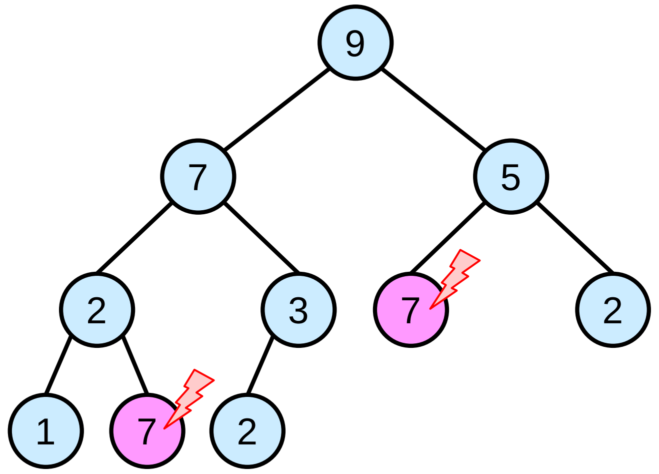
7	1	1	2	4	9	3	3	6	5

→ Not a heap, because the first element is neither the maximum nor the minimum of the elements

B:

9	7	5	2	3	7	2	1	7	2

→ By drawing the corresponding heap, we can see that elements at indices 5 and 8 (starting from zero) violate the heap condition. Therefore it is not a heap.

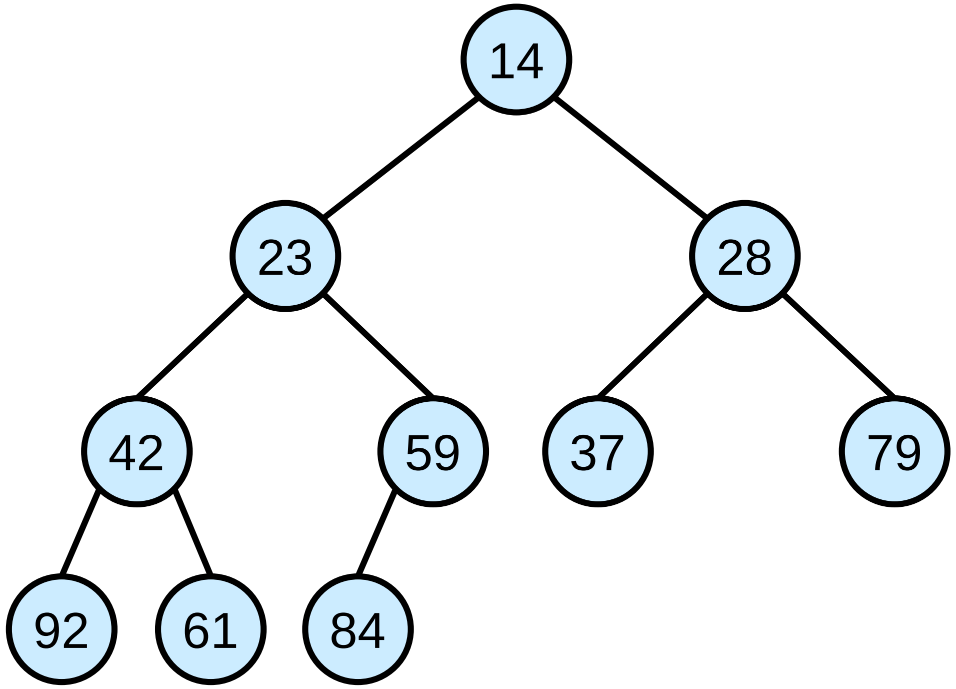


Heaps: Removing the Root

1) Remove the key from the root node.
2) Take the value in the last node and (temporarily) store it in the root node.
3) Let the value at the root node «trickle down» in the tree by swapping it with the value from the smaller (in case of min-heap) or bigger (max-heap) child node until both children are bigger (min-heap) or smaller (max-heap) than it. This process is also called to «heapify» the array / tree.

Remark: When implementing this as part of heap sort, we will just store the removed value in the now empty last node and no longer consider it as part of the heap.

Heaps: Demonstration: Removing the Root



14	23	28	42	59	37	79	92	61	84
23	42	28	61	59	37	79	92	84	14
28	42	37	61	59	84	79	92	23	14
37	42	79	61	59	84	92	28	23	14
42	59	79	61	92	84	37	28	23	14
59	61	79	84	92	42	37	28	23	14
61	84	79	92	59	42	37	28	23	14
79	84	92	61	59	42	37	28	23	14
84	92	79	61	59	42	37	28	23	14
92	84	79	61	59	42	37	28	23	14
92	84	79	61	59	42	37	28	23	14

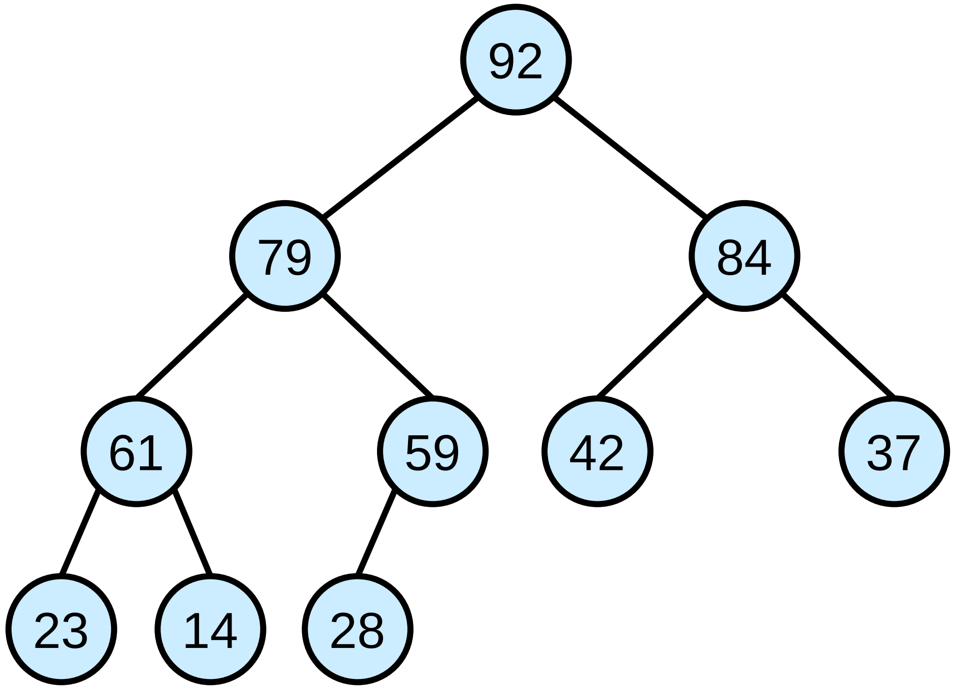
Heaps: Building a Heap

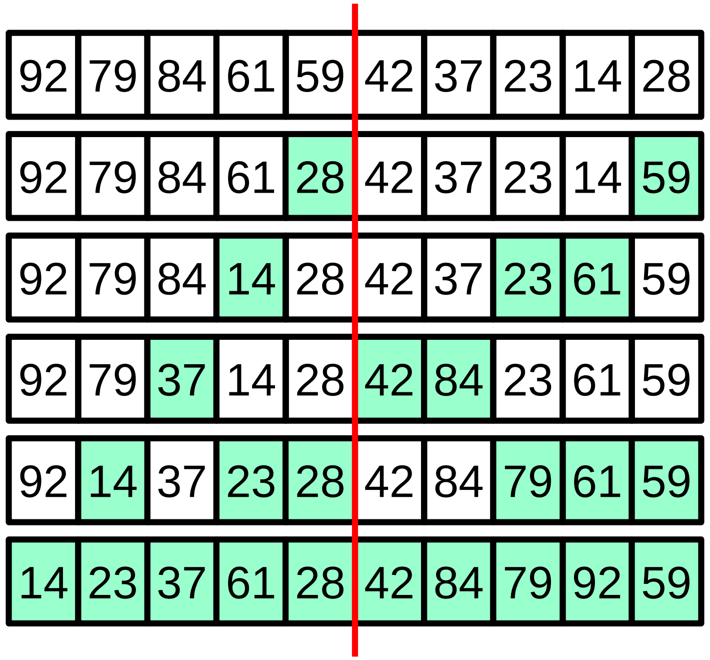
- How to get to a heap from an unordered array of values?

- Let’s assume that for a certain node, we knew that both its subtrees are already heaps: In this case, we can apply the same procedure which we used before to trickle down the nodes.
- Thus, we just start at the last node which has at least one child node and then work our way up to the root while making sure for each node, that everything below (its subtrees) are nice heaps.

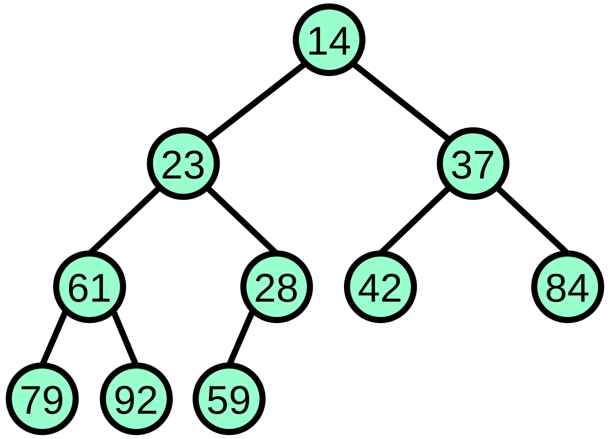
Institut für Informatik

Heaps: Demonstration: Building a Heap





92	79	84	61	59	42	37	23	14	28
92	79	84	61	28	42	37	23	14	59
92	79	84	14	28	42	37	23	61	59
92	79	37	14	28	42	84	23	61	59
92	14	37	23	28	42	84	79	61	59
14	23	37	61	28	42	84	79	92	59



Heaps: Sorting Algorithm

1) Transform input data into heap.
2) Successively remove the root element and reorganize the remaining heap.

Heapifying a Tree / Array Into a Min Heap in C Code

```
void heapify(int A[], int current_node, int length) {
    int left_child = get_left_child(current_node);
    int right_child = get_right_child(current_node);

    int smallest_node = current_node;
    if (left_child < length && A[left_child] < A[smallest_node]) {
        smallest_node = left_child;
    }
    if (right_child < length && A[right_child] < A[smallest_node]) {
        smallest_node = right_child;
    }

    if (smallest_node != current_node) {
        swap(A, current_node, smallest_node);
        heapify(A, smallest_node, length);
    }
}
```

get the array position of the left and right child of the current node

detect which of the three nodes (parent, left child, right child) is the smallest

if necessary, swap current node with smallest node and continue recursively upwards the tree structure

Comparison of Heapifying Code for Min Heap vs. Max Heap

```
void min_heapify(int A[], int i, int n) {
    int min = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] < A[min]) {
        min = l;
    }
    if (r < n && A[r] < A[min]) {
        min = r;
    }

    if (min != i) {
        swap(A, i, min);
        heapify(A, min, n);
    }
}
```

```
void max_heapify(int A[], int i, int n) {
    int max = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] > A[max]) {
        max = l;
    }
    if (r < n && A[r] > A[max]) {
        max = r;
    }

    if (max != i) {
        swap(A, i, max);
        heapify(A, max, n);
    }
}
```

Building a Heap in C Code

```
void buildHeap(int A[], int length) {
    int i;
    for (i = length/2; i >= 0; i--) {
        heapify(A, i, length);
    }
}
```

Heapsort in C Code

```c
void heapSort(int A[], int length) {
    buildHeap(A, length);

    int last_node_still_in_heap = length;
    int i;
    for (i = length-1; i > 0; i--) {
        swap(A, i, 0);
        last_node_still_in_heap--;
        heapify(A, 0, last_node_still_in_heap);
    }
}
```

transform the input array into a valid heap

successively take out the root and place it to the position which got free by removing the node (i.e. node next to the last node still considered a part of the heap).

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 $l \leq s \wedge A[l] > A[m]$ then m = l;
if $r \leq s \wedge A[r] > A[m]$ then m = r;
if $i \neq m$ then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

```
Algo: BuildHeap(A)
for $i = \lfloor n/2 \rfloor$ to $1$ do Heapify(A,i,n);
```

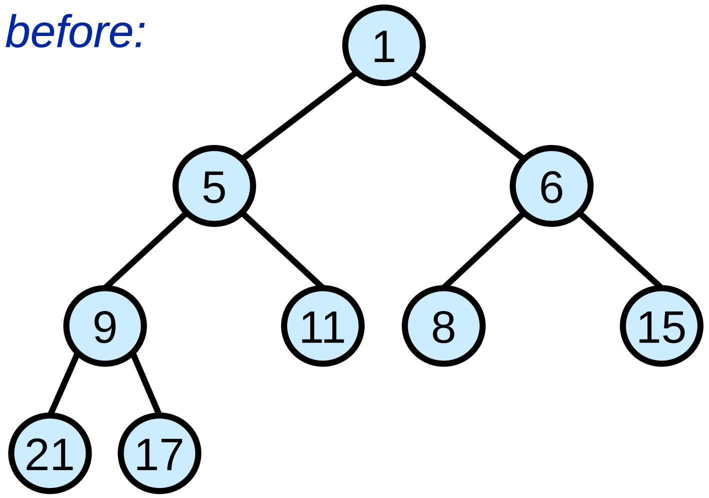
Heaps: Inserting a Node Into a Min Heap

A new node can be inserted into a min heap by performing the following steps:

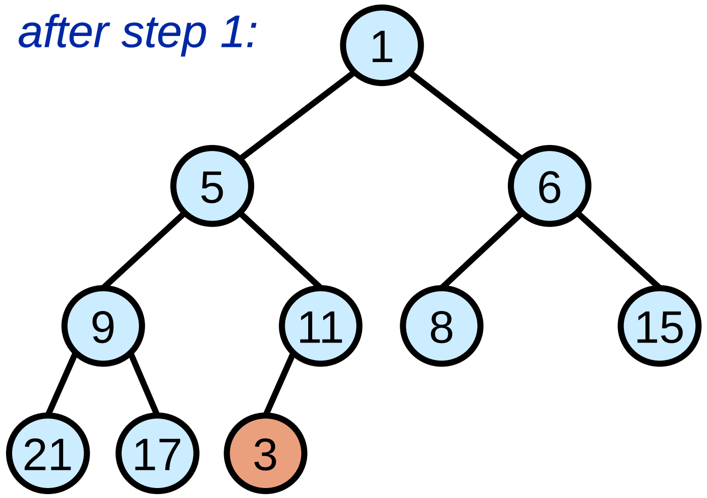
1) Insert the new node/element to the end of the heap/array (which will break the heap property in general)
2) To restore the heap property, sift *up* the new node by swapping it with its parent node as long as the parent node is bigger than the new node.

Example:

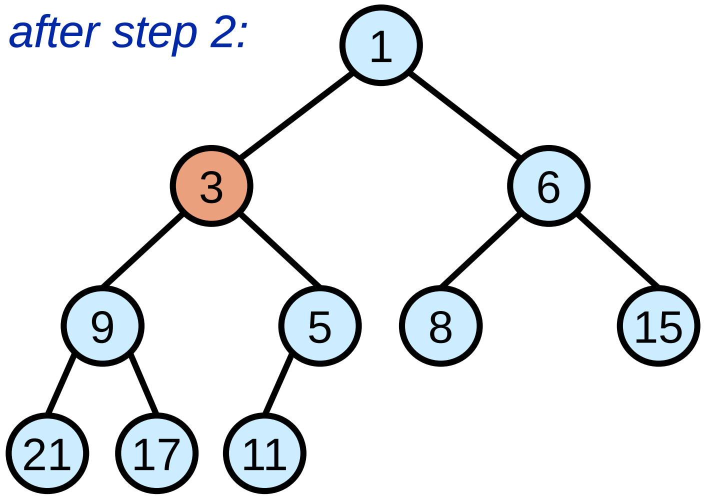
before:



after step 1:



after step 2:



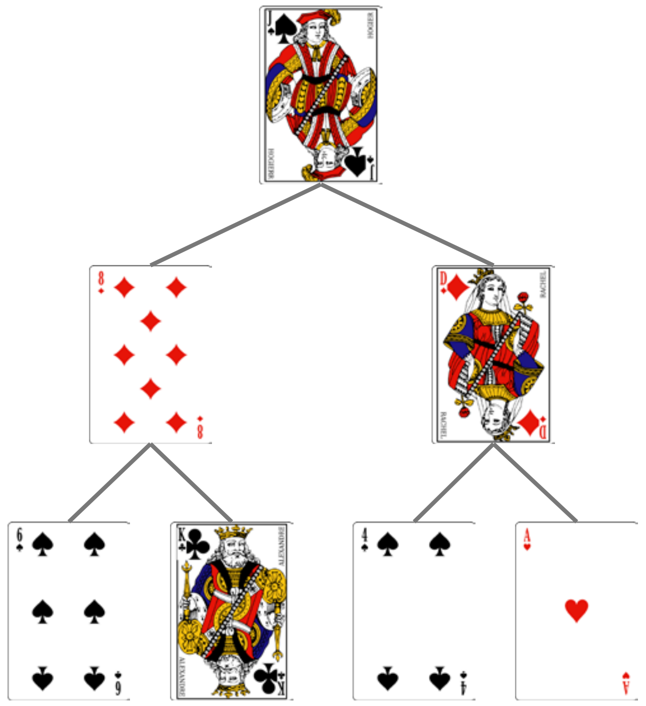
Heaps: Deleting any Node From a Heap

Exercise: How can an arbitrary node, specified by its node value, be deleted from a heap?

Asymptotic Running Times of Heap Operations

- Remove root: $O(\log(n))$
- Insert: $O(\log(n))$
- Build heap: $O(n \cdot \log(n))$
- Heapify: $O(n \cdot \log(n))$
- Heapsort: $O(n \cdot \log(n))$

Heaps, Heapsort: DIY / Game



1) Form groups of two to three people.
2) Get a deck of playing cards.
3) (If necessary: Familiarize yourself with the ordering of the cards.)
4) Shuffle the cards.
5) Lay out 12 cards in form of a binary three before you on the desk.

   (If you got the Joker, replace it through another card.)
6) Transform the card into a max- or min-heap.
7) Get a sorted set of cards by successively removing the root element.

8) If you feel comfortable with heapsort or if you get bored: Take 8 random cards of your deck and perform bubble sort, selection sort, insertion sort, quicksort, mergesort, … on it.

Heaps: Additional Practice

Additional exercises for practice can be found here:

https://www.physik.uzh.ch/~vogelch/algodat/practice/heaps.html

Institut für Informatik

Review of Exercise 5

- Task 1: Implementation of d-ary Heaps in C
- Task 2: Step-by-step Application of Heapsort
- Task 3: Dual-Pivot Quicksort

Exercise 5 – Task 1: Heapify Function

```
void heapify(int A[], int i, int n, int d) {
    int max = i;
    int k, child;
    for (k = 1; k<=d; k++) {
        child = (d*i) + k;
        if (child < n && A[child]> A[max]) {
            max = child;
        }
    }
    if (max != i) {
        swap(A, i, max);
        heapify(A, max, n, d);
    }
}
```

Exercise 5 – Task 1: Building the Heap and Heapsort Function

```
void buildMaxHeap(int A[], int n, int d) {
    int i;
    for (i = (n-1)/d; i >= 0; i--) {
        heapify(A, i, n, d);
    }
}
```

```
void heapSort(int A[], int n, int d) {
    int i, s = n;

    buildMaxHeap(A, n, d);
    for (i = n-1; i > 0; i--) {
        swap(A, i, 0);
        s--;
        heapify(A, 0, s, d);
    }
}
```

Exercise 5 – Task 1: Printing the Array

```
void printArray(int A[], int n) {
    int i;
    printf("[ ");
    for (i = 0; i < n; i++) {
        printf("%d", A[i]);
        if (i < n-1) {
            printf(", ");
        }
    }
    printf(" ]\n");
}
```

Exercise 5 – Task 1: Printing the Heap

```
void printHeap(int A[], int n, int d) {
    int i, l, r, k;

    printf("graph g {\n");
    for (i = 0; i < n; i++) {
        for (k = 1; k <= d; k++){
            if ((d*i) + k < n) {
                printf("  %d -- %d\n", A[i], A[(d*i) + k]);
            }
        }
    }
    printf("}");
}
```

Exercise 5 – Task 2: Step-by-step Application of Heapsort

Task description:

«Given the array [37, 24, 45, 9, 21, 17, 5, 14], sort it in ascending order using Heapsort algorithm. Precisely explain every step performed. You can start from making max-heap from initialy binary tree shown below. Show every node exchange that occur.»

Exercise 5 – Task 3: Heapsort by Hand

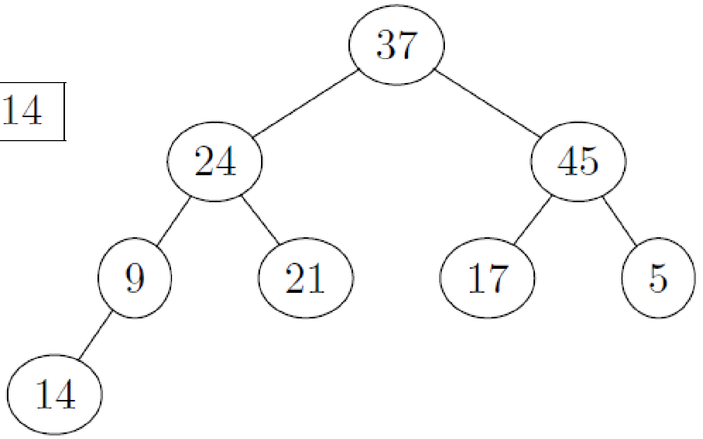
Task description:

«Given the array [57, 42, 69, 11, 35, 28, 7, 19], sort it in ascending order using the Heapsort algorithm. Precisely explain every step performed. You can start from making a max-heap from the initial binary tree shown below. Show every node exchange that occurs.»

Input array

37	24	45	9	21	17	5	14

Initial binary tree



Universität Zürich^UZH

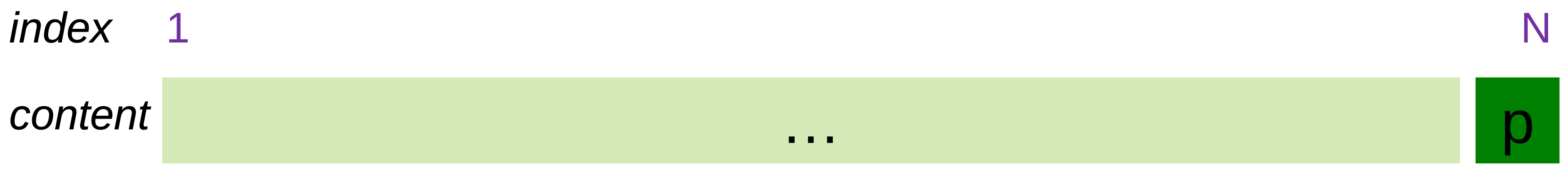
Institut für Informatik

Quicksort

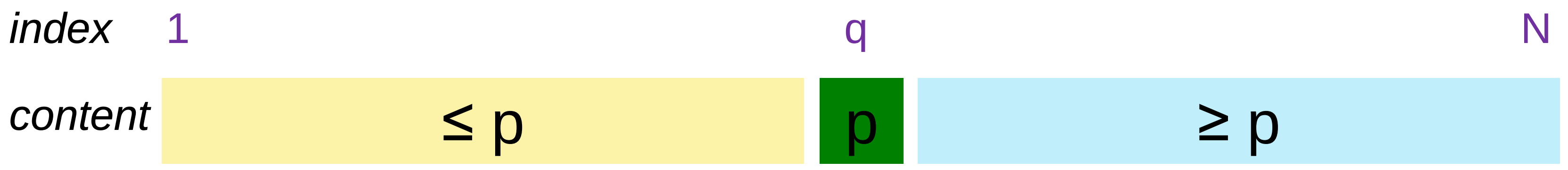
- Principles
- Partitioning
- Quicksort vs. Binary Tree Sort

Quicksort: Principle

1) **Pivot selection:** Choose an element from the input array which should be used for dividing it into two subarrays; this element is called the pivot element. (In the example below, the last element of the array is choosen as pivot.)



2) **Divide:** partition the array into two subarrays such that elements in the left part are smaller or equal to the elements in the right part:

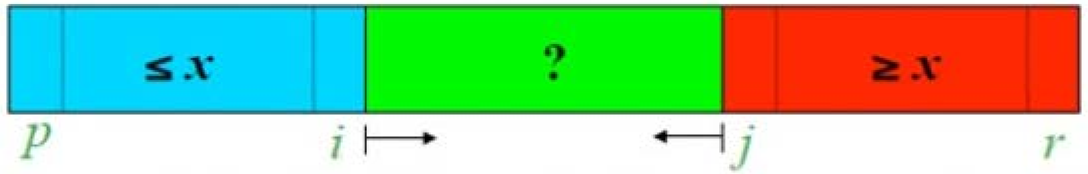


3) **Conquer:** recursively apply the steps 1 and 2 (pivot selection and partitioning) on the two subarrays.

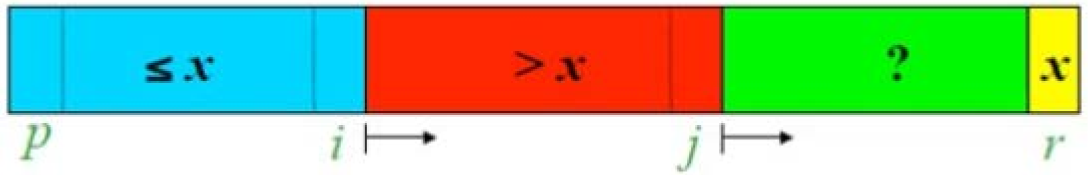
Partitioning Algorithms: Lomuto / Hoare

There are two main ways how the partitioning of the array in quicksort can be done:

- **Hoare's Algorithm**:
  - Grows two regions: A[p..i] from left to right and A[j..r] from right to left



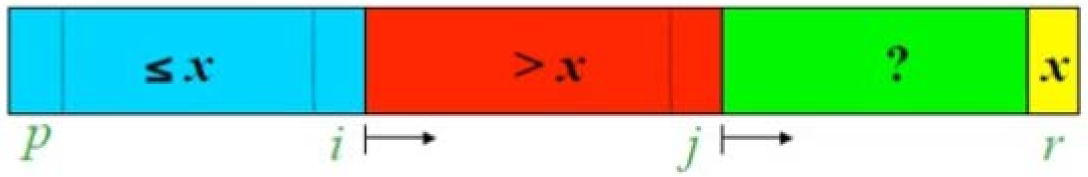
- **Lomuto's Algorithm**:
  - Works with a pivot; grows two regions from left: A[p..i] and A[i..j]; inferior in performance to Hoare's algorithm



Lomuto Partitioning Algorithm

The algorithm returns the position / index of the pivot in partitioned array.

It's complexity is Θ(n).



Algo: LomutoPartition(A,l,r)

```
x = A[r];
i = l-1;
for $j = l$ to $r$-1 do
    if $A[j] \leq x$ then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

- Comprehension question: What happens when the input array is sorted ascendingly?
- Additional exercise: What is the loop invariant of Lomuto partitioning?

Quicksort: Comprehension Question

Quicksort has an asymptotic running time of $O(n \cdot \log(n))$ in the average case and thus is faster than algorithms like bubblesort, insertionsort or selectionsort.

This is achieved by the quicksort algorithm because it has to do considerably less comparisons of elements than the aforementioned algorithms (in the average case).

Question: How / where does quicksort save comparisons?

Quicksort needs less comparisons because of the **partitioning** it applies. All elements on the left (lower) side of a partitioning will never have to be compared with an element on the right (upper) side of the partitioning. This is not ensured in algorithms like bubblesort.

This also reveals that there is a very close relationship between binary search trees and quicksort. Actually, bottom line, they both apply exactly the same principle.

Review of Exercise 5

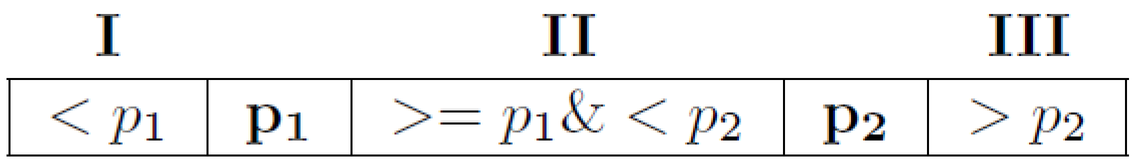
- Task 1: Implementation of d-ary Heaps in C
- Task 2: Step-by-step Application of Heapsort
- Task 3: Dual-Pivot Quicksort

Exercise 5 – Task 3: Dual-Pivot Quicksort

Task:

In a dual-pivot quicksort implementation, an array A[low...high] is sorted based on two pivots p1 = A[low] and p2 = A[high]. The pivot p1 must be less than p2, otherwise they have to be swapped. Here low is the left end of the array A and high is the right end of the array A.

The partitioning process is the core of the dual-pivot quicksort. We begin partitioning the array in three proper partition I, II or III.



Exercise 5 – Task 3: Dual-Pivot Quicksort

```c
void partitioning(int A[], int low, int high, int *p1, int *p2) {
    int case1 = 0, case2 =0, case3 = 0;
    int l = low+1;
    int k = low+1;
    int g = high;
    while(k < g) {
        case1 = 0; case2 = 0; case3 = 0;
        if (A[k] < A[low]) { swap(A, l++, k++); case1 = 1; }
        else if (A[k] >= A[high]) { swap(A, k, --g); case2 = 1; }
        else { k++; case3 = 1; }
        printf("%d - %d - %d === ",case1,case2,case3);
        printArray(A, ARRAY_SIZE);
    }
    swap(A, low, l-1);
    swap(A, high, k);
    *p1 = l-1;
    *p2 = k;
}
```

Exercise 5 – Task 3: Dual-Pivot Quicksort

```
void quicksort(int A[], int low, int high) {
    if (high - low <= 0) { return; }
    if (A[low] > A[high]) { swap(A, low, high); }
    int p1, p2;
    partitioning(A, low, high, &p1, &p2);
    quicksort(A, low, p1-1 );
    quicksort(A, p1+1, p2-1 );
    quicksort(A, p2+1 , high);
}
```

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:

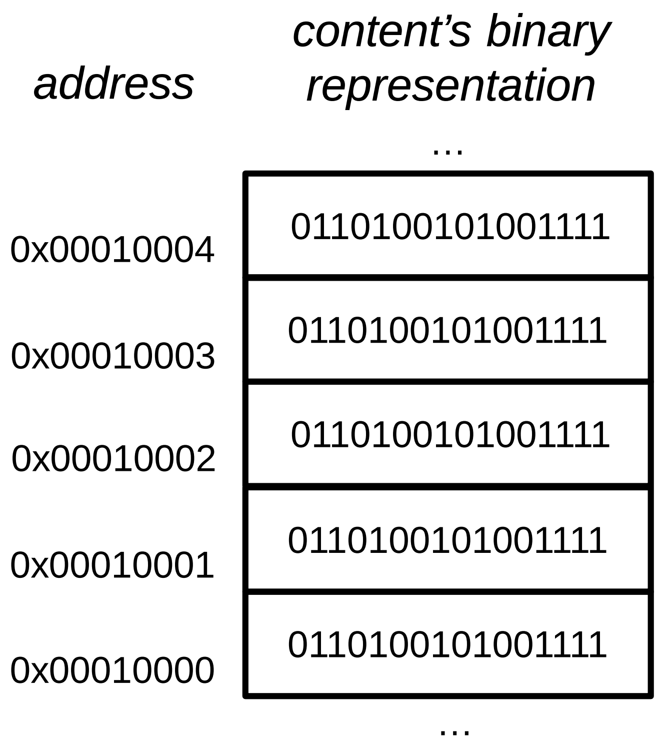
```c
struct Point2D {
        int x;
        int y;
};
```

With **`typedef`** keyword:

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

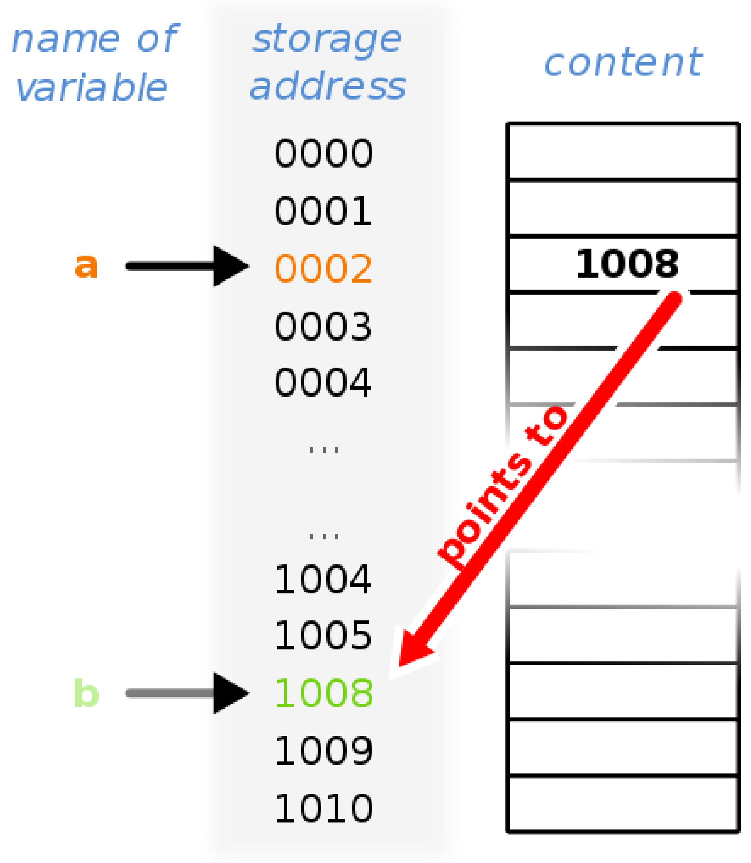
Pointers: Introduction

- 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 $\leftarrow$ as the assignment operator. Thus when we write x $\leftarrow$ 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)) $\leftarrow$ 42

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

```
a = b;
```

content(address(a)) $\leftarrow$ 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.

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 – theoretically – also could 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?

```
#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?

```
#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;
}
```

```
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

pass by reference
cup =
fillCup( )
pass by value
cup =
fillCup( )

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

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).

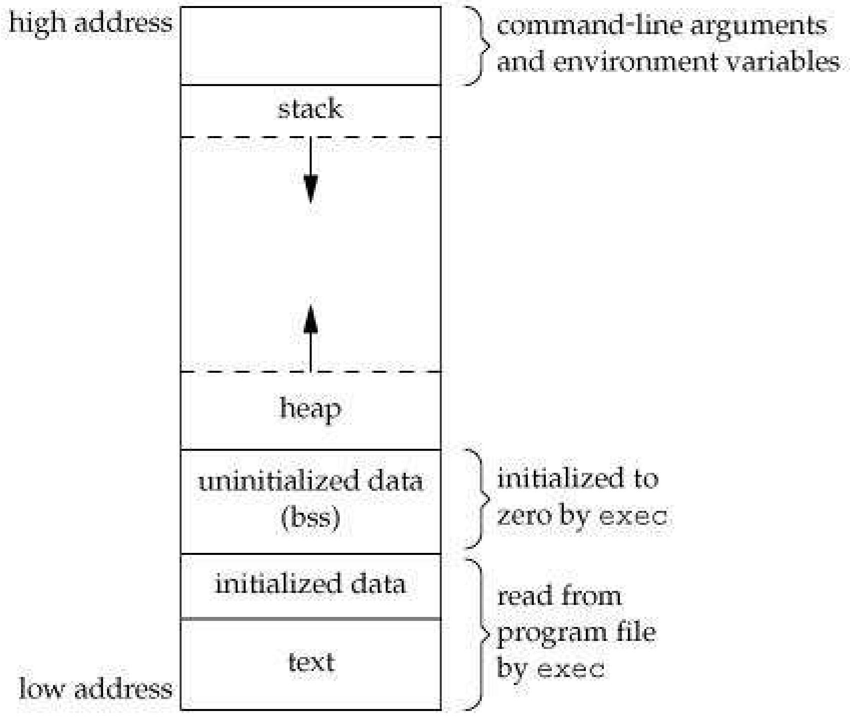
Syntax example:

```
int* pointer;
pointer = (int*) 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


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

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`).

Preview on Exercise 6

- Task 1: Bubblesort with Pointers
- Task 2: Copying a String
- Task 3: Singly Linked List Implementation
- Task 4: Representing Numbers with Linked Lists

Exercise 6 – Task 1: Bubblesort with Pointers

Apply what you’ve learnt in the lecture and the tutorials.

Exercise 6 – Task 2: Copying a String

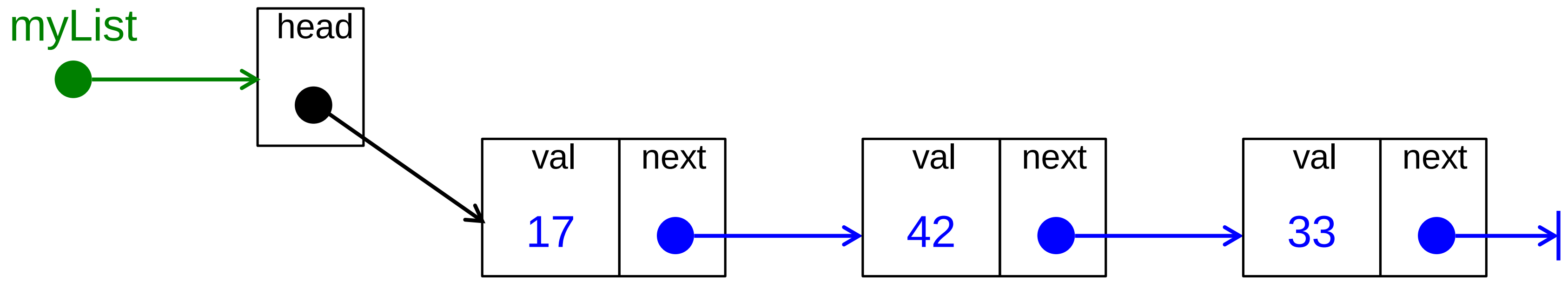
It may help to draw a sketch of the array and the pointers.

Exercise 6 – Task 3: Implementation of a Singly Linked List

Look at the respective slides from the lecture: SL05, pp. 16 – 26.

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

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



Exercise 6 – Task 4: Representing Numbers with a Linked List

Apply what you’ve learnt in the lecture and the tutorials.

Universität Zürich^UZH

Institut für Informatik

Wrap-Up

- Summary
- Outlook

Thank you for your attention.