DATA STRUCTURES AND ALGORITHMS

Lecture 10

Learning outcome 3

Logarithmic complexity

- Statement: if a binary tree has n nodes, then the depth of the tree $h >= log_2 n$
- Proof:
  - How can we place 15 nodes in a binary tree so that the depth is as small as possible?
    - If we make a perfect binary tree of depth 3
    - Equivalently, if we have a perfect tree of depth 3, it can hold 15 nodes
- We are interested in the following: if we have a tree of depth h, how many nodes can fit in it?

Proof of logarithmic complexity (1/2)

- A tree of depth 0 can fit: $2^1 - 1$ (1 node)
- A tree of depth 1 can fit: $2^2 - 1$ (3 nodes)
- A tree of depth 2 can fit: $2^3 - 1$ (7 nodes)
- A tree of depth 3 can fit: $2^4 - 1$ (15 nodes)
- A tree of depth 4 can fit: $2^5 - 1$ (31 nodes)
- …
- A tree of depth h can fit: $2^{h+1} - 1$

Strana ▪ 3

Proof of logarithmic complexity (2/2)

- So we calculate:

  $n = 2^{h+1} - 1$

  $n + 1 = 2^{h+1}$

  $log_2(n+1) = h + 1$

  $h = log_2(n+1) - 1 \approx log_2 n$
- So the depth of the perfect binary tree is log*n*
  - The depth of all other trees is greater
- Algorithms that process one node at each depth can achieve speed $\Omega(\log n)$
  - Worst case is $O(n)$ if we have a diagonal tree
  - Interesting: the average case is always closer to the best, i.e. $\Theta(\log n)$

Strana ▪

HEAP

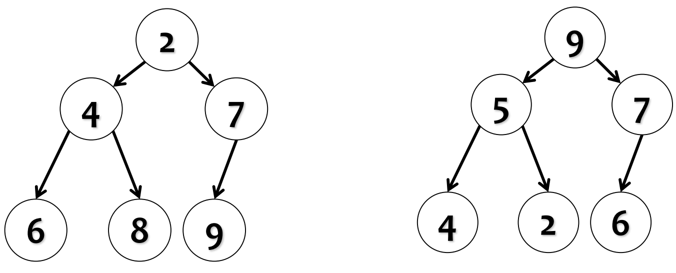
Introduction

- A heap is a data structure that satisfies the conditions:
  - It is a complete binary tree
    - It can be any complete tree, but we will only observe binary trees
  - The value in the parent node is:
    - Always greater than or equal to all children's values (max-heap), or
    - Always less than or equal to all children's values (min-heap)
    - Note: nothing is said about the values of the siblings
- The heap is of great importance and frequent application in computing:
  - To implement the priority queue
  - To implement the HEAPSORT sorting algorithm

Max-heap and min-heap

- In the min-heap, the smallest element is placed at the root of the tree
- In the max-heap, the largest element is placed at the root of the tree

Min-heap *Max-heap*



- In the rest of the lecture we will observe max-heap
  - The min-heap variant is equivalent in everything

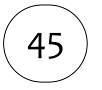
ALGEBRA

BUILDING A COMPLETE BINARY TREE

ALGEBRA

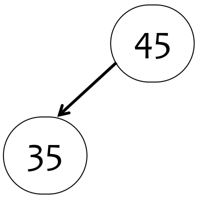
Building a complete binary tree (1/8)

- We want to place values in a complete binary tree: 45, 35, 23, 27, 21, 22, 4, 19



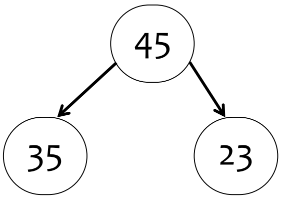
Building a complete binary tree (2/8)

- The second node is always the left root child



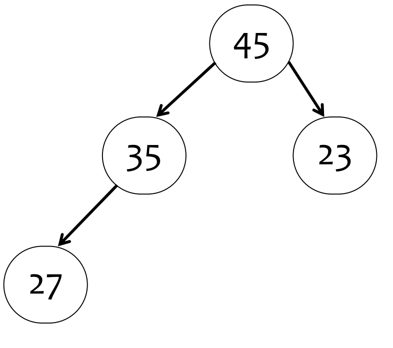
Building a complete binary tree (3/8)

- The third node is always the right root child



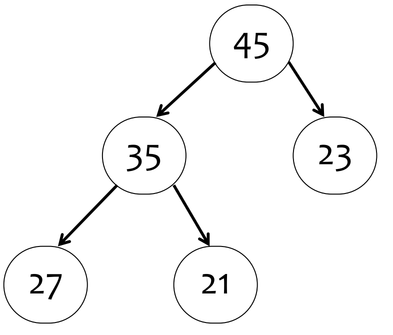
Building a complete binary tree (4/8)

- Nodes always fill the next level from left to right



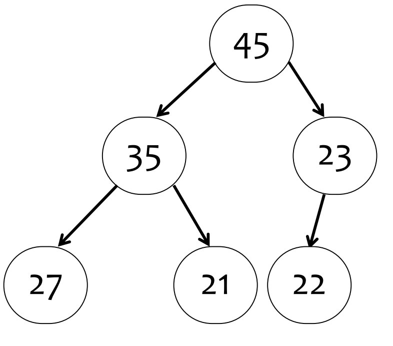
Building a complete binary tree (5/8)

- Nodes always fill the next level from left to right



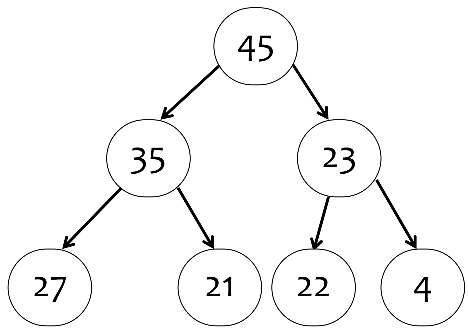
Building a complete binary tree (6/8)

- Nodes always fill the next level from left to right



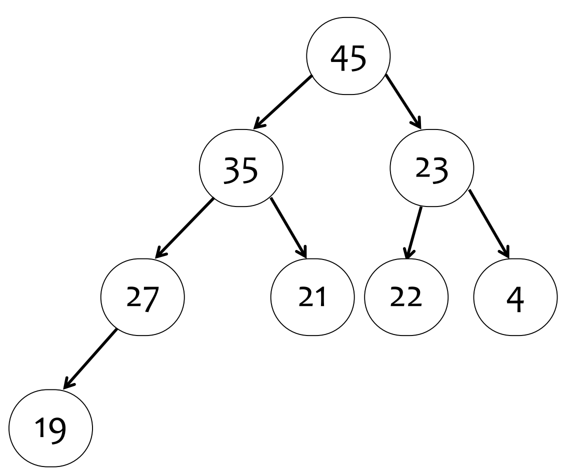
Building a complete binary tree (7/8)

- Nodes always fill the next level from left to right



Building a complete binary tree (8/8)

- We got a complete binary tree with 8 nodes
- Respecting the rule that the value of the parents must be >= the values of the children, we got a max-heap
- The data were thus prepared in advance
- Building has complexity $O(n)$



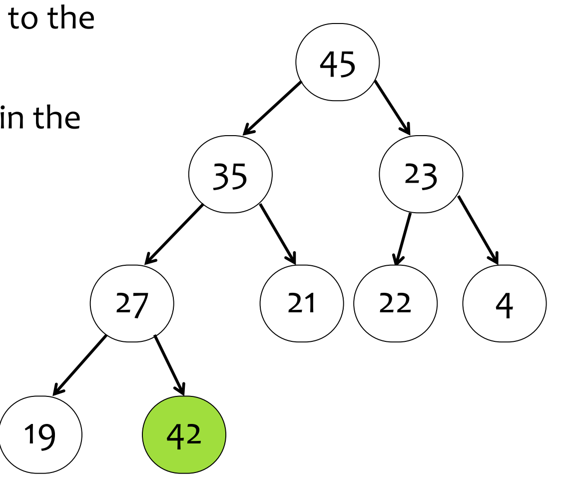
ADDING A NODE TO THE HEAP

The procedure

- We use the following procedure to add nodes to the heap:
  1. We initially add the node to the first free space according to the previous rules
  2. We move the node towards the root by swapping it with the parent until it comes to the right place

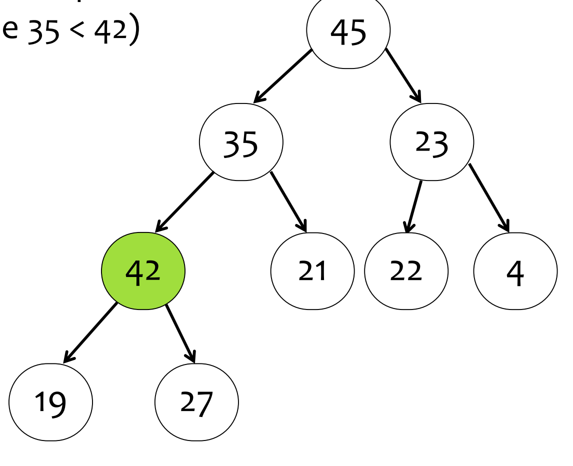
Adding a node to the heap (1/3)

- Let's add a value of 42 to the previous heap
- We put the new node in the first available place
- Furthermore, we maintain the property of the heap by lifting the new node up, swapping it with its parent, until it reaches the correct location



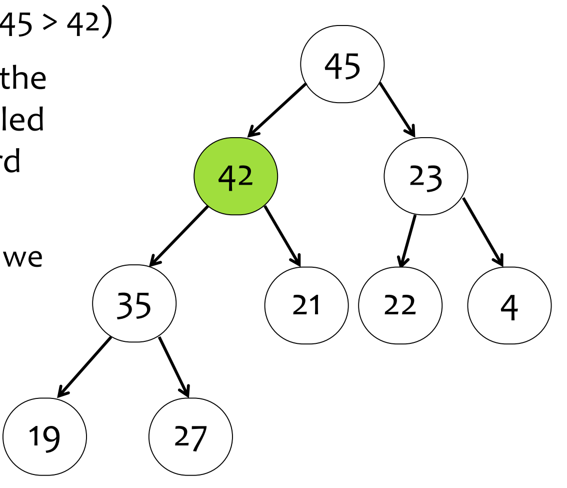
Adding a node to the heap (2/3)

- After the swap with node 42 it is still not good (because 35 < 42)



Adding a node to the heap (3/3)

- It's OK now (because 45 > 42)
- This process of lifting the node to the root is called reheapification upward
  - Complexity is $O(\log n)$ because at each level we process 1 node



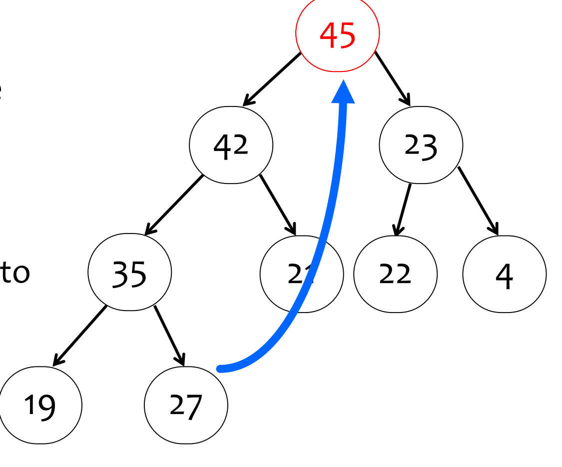
REMOVING FROM THE TOP OF THE HEAP

The procedure

- With a heap, we always process the element at the top
  - In the max-heap, it's the biggest element on the heap
- Removing the element from the top of the heap disrupts the heap structure
  - Some action needs to be taken to keep it a heap
- The procedure:
  1. We take the last node and swap it to the root
  2. We lower the node towards the leaves until it comes to the right place
     - We always swap it with an bigger child (because it is max-heap)

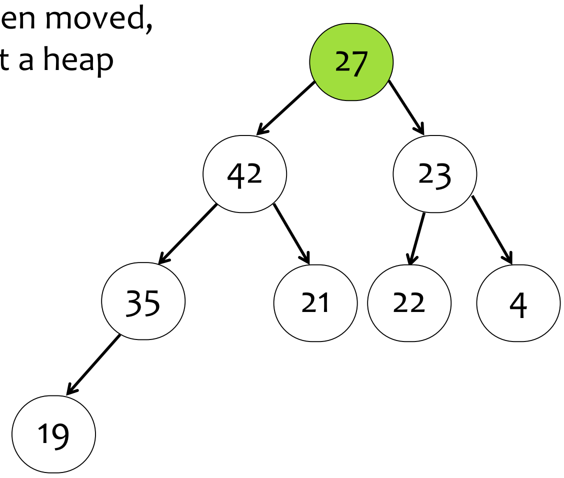
Removing from the top of the heap (1/4)

- We take and process the root
- How do you rearrange the tree to still be a heap?
- The first step is to transfer the last node to the root of the heap



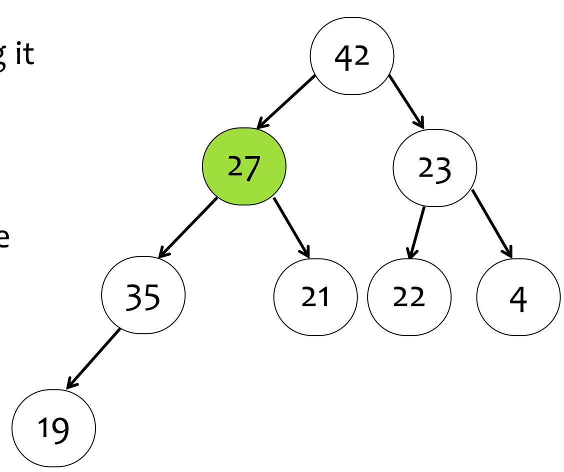
Removing from the top of the heap (2/4)

- Now the node has been moved, but the tree is still not a heap (because 27 < 42)



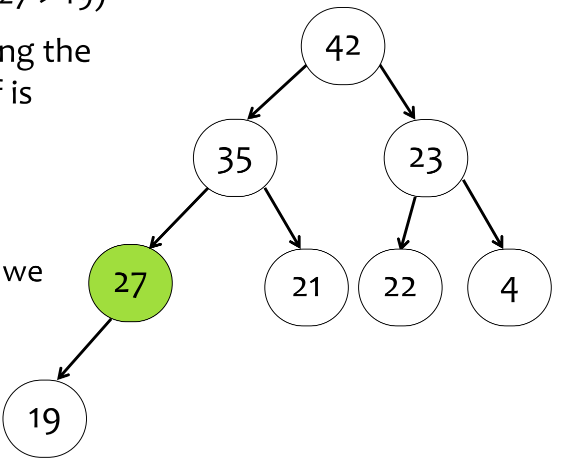
Removing from the top of the heap (3/4)

- We lower the moved node down, replacing it with an bigger child, until it comes to the correct location
- Not good yet because 27 < 35



Removing from the top of the heap (4/4)

- It's OK now (because 27 > 19)
- This process of lowering the node towards the leaf is called reheapification downward
  - Complexity is $O(\log n)$ because at each level we process 1 node



Result of removal from the heap

- Removal of the elements destroys the heap
  - Just as with the stack and the queue
- The result is descending elements
  - For min-heap they would be ascending
- The speed is great because the heap is always optimally organized

DEMO

- http://www.cs.usfca.edu/~galles/visualization/Heap.html
- Create a min-heap heap with values: 2, 4, 6, 8, 10, 12, 14
- Add values: 5, 3, 1
- Remove the root by clicking the button "Remove Smallest"

PRIORITY QUEUE

Introduction

- In practice, it is often the case that certain priorities need to be defined in one queue
  - For example, at doctor’s office patients enter according to the FIFO principle
  - The exception are emergency patients because they have an advantage
    - We say that these are patients of higher priority
  - If there are several patients of the same priority, the doctor treats them again according to the FIFO principle
    - This is not a prerequisite for the priority queues; we can also have priority queues that process elements of the same priority in a different order

Priority queue

- The priority queue is the FIFO structure in which each element has a defined priority
  - First, the elements of the highest priority are processed according to the FIFO principle
  - After that, the elements of lower priority are processed in the same way, and so on until the lowest priority

Usage

- The main usages of the priority queue are:
  - Shortest path search algorithms (computer games):
    - Dijkstra algorithm
    - A* algorithm
  - Data compression
  - Sorting (heap sort)
  - Task scheduler in operating systems

HEAP IN STL

Introduction

- STL offers two ways of working with the heap and the priority queue:
  - Direct construction and usage of the heap by using functions from `<algorithm>`
    - A bit more complex
  - Using the container `priority_queue<…>` as a wrapper around functions from `<algorithm>`
    - A bit simpler

ALGEBRA

Direct construction and usage of the heap (1/3)

- `<algorithm>` offers three functions to work with the heap:
  - `make_heap(begin, end)`
    - Reorders elements in [begin, end) so they become max-heap
    - Once the function is complete, the largest element is guaranteed to be in place begin
    - Elements can be placed in the array or in the container of types `array<T,N>`, `vector<T>` or `deque<T>`
    - Complexity is $O(n)$

ALGEBRA

Example

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

for (unsigned i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
}
cout << endl;
```

- Check to make sure it's really a heap

Direct construction and usage of the heap (2/3)

- `push_heap(begin, end)`
  - The function considers that elements [begin, end – 1) form a heap and that the newly added element is found at position end
  - Function takes element from end and moves it to a proper position (reheapification upward)
  - After the function completes, the heap is formed in the entire region [begin, end)
  - Complexity is $O(\log n)$
  - How can we understand this function: „take the last element and move it where it is needed so that we get a heap"

Example

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

v.push_back(99);
for (unsigned i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
}
cout << endl;

push_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
}
cout << endl;
```


ALGEBRA

Direct construction and usage of the heap (3/3)

- `pop_heap(begin, end)`
  - The function considers that [begin, end) form the heap
  - Function moves element from position begin to a position end – 1 (reheapification downward)
  - After the function completes, [begin, end – 1) form the heap
  - How can we understand this function: „ remove the element from the top of the heap and rearrange all other elements so that this is still a heap, just with only one element less”

ALGEBRA

Example

```
vector<int> v = { 33, 11, 22, 88, 77, 55, 44, 33, 22, 66 };
make_heap(v.begin(), v.end());

v.push_back(99);
for (unsigned i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
}
cout << endl;

push_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
}
cout << endl;

pop_heap(v.begin(), v.end());
for (unsigned i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
}
```


ALGEBRA

PRIORITY QUEUE IN STL

ALGEBRA

`priority_queue<…>`

- `priority_queue<…>` class is a container adapter
  - A wrapper around contained container that is a heap
  - Contained container can be:
    - Vector (default)
    - Deque
    - Any our class with methods:
      - empty()
      - size()
      - front()
      - push_back()
      - pop_back()
      - Class must allow a direct access to *i*-th element (so list and forward_list cannot be used)

Basic ways of creating a priority queue

- Basic ways of creating a priority queue are:
  - `priority_queue<int> one;`
    - Creates an empty priority queue supported by a vector
  - `priority_queue<int, deque<int>> two;`
    - Creates an empty priority queue supported by a deque
  - `vector<int> v({ 11, 22, 33 });`
    `priority_queue<int> three(v.begin(), v.end());`
    - Creates a priority queue supported by a vector and fills it with elements from the range [begin, end)
    - „The constructor … calls … make_heap on the range that includes all its elements …”

Using priority queue (1/2)

- `pq.push(val)` add a copy of `val` to a proper position in the priority queue
  - „This member function effectively calls … push_back of the underlying container object, and then reorders it to its location in the heap by calling the push_heap algorithm …"
- `pq.pop()` removes the element with the highest priority (the one at the top of the heap)
  - „This member function effectively calls … pop_heap algorithm to keep the heap property of priority_queues and then calls the member function pop_back of the underlying container object to remove the element"

Using priority queue (2/2)

- `pq.top()` returns a reference to an element with the highest priority (the one on the top of the heap)
- `pq.size()` returns the number of element in the priority queue
- `pq.empty()` returns if the priority queue is empty or not

MORE COMPLEX USAGES OF THE PRIORITY QUEUE

But where are the priorities?

- The previous examples assume that the integer value stored in the queue is equal to the priority
  - Can we, for example, keep emails in the priority queue, each of which has some priority?
- In order to be able to store any type of data in priority queue, we need to answer the following question:
  - If we have two objects of some type, which one is smaller?
- The easiest way is to define a comparator:

```
struct YoungerHavePriority {
  bool operator() (Person& o1, Person& o2) {
    return o1.year_of_birth < o2.year_of_birth;
  }
};
```

Function call operator

More complex usage of the priority queue

- Once we have a comparator defined, we can create a priority queue as follows:

```
priority_queue<
    Person,
    vector<Person>,
    YoungerHavePriority> pqpeople;
```

  - Creates an empty priority queue supported by a vector; priorities are defined by using `YoungerHavePriority`

ALGEBRA

Problems

1. We've been given a vector of 10 elements. Using the priority queue, we must display all elements in sorted order, from larger to smaller.
2. We've been given a vector of 10 elements. Using the priority queue, we must display all elements in sorted order, from smaller to larger.
3. Write a program that uses a priority queue to process received messages based on priorities (1 = minimal, 2 = normal, 3 = high priority). Receive some messages and display them on the console.

ALGEBRA

Solution to a problem 1

```
vector<int> numbers({ 17, 6, 99, 52, 11, 1, 8, 15, 7, 23 });

priority_queue<int> pq(numbers.begin(), numbers.end());

while (!pq.empty()) {
    cout << pq.top() << endl;
    pq.pop();
}
```


Solution to a problem 2

```
#include <functional>
…

vector<int> numbers({ 17, 6, 99, 52, 11, 1, 8, 15, 7, 23 });

priority_queue<int, vector<int>, greater<int>>
                                    pq(numbers.begin(), numbers.end());

while (!pq.empty()) {
    cout << pq.top() << endl;
    pq.pop();
}
```

Solution to a problem 3

```
struct Message {
    string subject;
    string body;
    int priority;

    Message(string subject, string body, int priority) {
        this->subject = subject;
        this->body = body;
        this->priority = priority;
    }
};

struct HigherToLowerPriorityComparator {
    bool operator() (Message& m1, Message& m2) {
        return m1.priority < m2.priority;
    }
};
```


Solution to a problem 3

```
priority_queue<Message, vector<Message>,
HigherToLowerPriorityComparator> pq;
pq.push(Message("Cat chases a ball",
    "Watch this funny video :)", 1));
pq.push(Message("I am out",
    "I am taking a day off tomorrow", 2));
pq.push(Message("Emergency meeting",
    "In 30 minutes, room 204, mandatory! ", 3));

while (!pq.empty()) {
    cout << pq.top().subject << " " << pq.top().priority <<
endl;
    pq.pop();
}
```