



#### UNIVERSITY OF GOTHENBURG

#### **Data structures**

Priority queues, binary heaps

Dr. Alex Gerdes DIT961 - VT 2018



- Course representatives
  - Mohamad Qutbuddin Habib
  - Carl Agrell
  - Gunnar Stenlund
- Kunsulttid: jag är på kontor till 16:00 idag.



- A priority queue has three operations:
  - *insert*: add a new element
  - *find minimum*: return the smallest element
  - *delete minimum*: remove the smallest element
- Similar idea to a stack or a queue, but:
  - you get the smallest element out instead of the first-added (queue) or last-added (stack)
- (Java: PriorityQueue class)



- A natural application for priority queues is to handle access to a limited resource where the resource users can be assigned different degree of urgency
- An electronic queueing system for an emergency room where the patients' conditions varies in severity
- Processes with priority levels queueing for resources (such as computation time)

# **Applications - simulation**



- Keep a priority queue of future events
  - "At 10am a person will enter the shop"
- Simulator's job: remove earliest event and run it, then repeat
  - In the priority queue, earlier events will be counted as "smaller" than later events
- When we run that event, it can in turn add more events to the priority queue
  - When a person enters the shop, add an event "the person picked up some milk" to the priority queue at a time of 1 minute later



#### • Sorting:

- Start with an empty priority queue
- Add each element of the input list in turn
- As long as the priority queue is not empty, find and remove the smallest element
- You get all elements out in ascending order!
- *Heapsort* is an in-place version of this



- An efficient implementation of priority queues, the binary heap, with the help of:
  - Invariants
  - Trees
- We'll wander off into those two topics and then come back to priority queues
- Also, hopefully, a bit of: how to think like a data structure designer
  - I will try to explain why heaps are the way they are rather than just how they work



- Idea 1: implement a priority queue as a dynamic array
  - Insert: add new element to end of array: **O(1)**
  - Find minimum: linear search through array: **O**(*n*)
  - Delete minimum: remove minimum element: **O**(*n*)
- Finding the minimum is quite expensive though
- Another idea?



- Idea 2: use a sorted array
  - Insert: insert new element in right place: **O**(*n*)
  - Find minimum: minimum is first element: **O(1)**
  - Delete minimum: remove first element: **O**(*n*)
- Finding the minimum is cheap! Yay!
- But... insertion and deletion got expensive :(



- Idea 3: implement a priority queue as a reverse-sorted array
  - Insert: insert new element in right place: **O**(*n*)
  - Find minimum: minimum is last element: **O(1)**
  - Delete minimum: remove last element: **O(1)**
- A bit better, but O(n) insertion is not so good...



- "The array is reverse-sorted" is an example of an invariant of a data structure
  - An invariant is a property that always holds in our implementation of the data structure
  - Something the data structure designer picks that helps implementing the data structure
- Insert, find minimum and delete minimum can assume that the array is already reverse- sorted...
  - ...but they must make sure that the array remains reversesorted afterwards (they must *preserve the invariant*)

## Pre-, postconditions, and invariants



- Preconditions requirements on a function's input (not expressed by types) that must hold when it's called
- Postcondition requirements on a function's output that will hold when it returns
- Invariants requirements on data that exists in between function calls (in Java this typically means requirements on an object's instance variables)
- Invariants can be seen as pre- and postconditions that are added to all instance methods of a class



- What happens if you break the invariant?
  - e.g., insert simply adds the new element to the end
- Answer: nothing goes wrong straight away, but later operations might fail
  - A later *find minimum* might return the wrong answer!
- These kind of bugs are a nightmare to track down!
- Solution: check the invariant



• Define a method

```
bool invariant()
```

that returns true if the invariant holds

- in this case, if the array is reverse-sorted
- $\cdot$  Then, in the implementation of every operation, do

assert invariant();

- This will throw an exception if the invariant doesn't hold!
- (Note: in Java, must run program with -ea)



• Define a function

```
invariant :: Whatever -> Bool
```

• Then add an extra case to all operations:

 [Perhaps remove this case when you've finished testing your code]



- Writing down and checking invariants will help you find bugs much more easily
  - I'd say *most* data structure bugs involve breaking an invariant
  - Even if you don't think about an invariant, if your data structure is at all fancy there is probably one hiding there!
  - Almost all programming languages support assertions use them to check invariants and make your life easier

### Live coding

## **Designing data structures**



- Here is how *not* to design a data structure:
  - 1. Take the operations you have to implement
  - 2. Think very hard about how to implement them
  - 3. Bash something together that seems to work
- Because:
  - You will probably have lots of bugs
  - You will probably miss the best solution



- We implemented bounded queues by an array and a pair of indices front and back
  - The content of the queue is the elements between index front and index back
- Once we decide on this representation, there is only one way to implement the queue!
  - Here, "representation" means what datatype we use, plus what an instance of that datatype *means* as a queue (in this case, what the queue contains)



- We represented a priority queue by an array with the invariant that the array is reverse-sorted
- Once we choose this invariant, there is only one way to implement it!



- How to design a data structure:
  - Pick a representation Here: we represent the priority queue by an array
  - Pick an invariant Here: the array is reverse-sorted
- Once you have the right representation and invariant, the operations often almost "design themselves"!
  - There is often only one way to implement them
- You could say...

data structure = representation + invariant



- How do you know which representation and invariant to go for?
- Good plan: have a first guess, see if the operations work out, then tweak it
  - Queues: at first we tried a dynamic array, but there was no way to efficiently remove items, so we switched to a circular array
  - Priority queues: at first we tried a sorted array, but then remove minimum needed to delete the first element (inefficient), so we switched to a reverse- sorted array
- Takes practice!

## More on invariants



- A strong invariant like "the array is reverse-sorted":
  - Can make it easier to *get* information from the data structure (the data is more structured)
  - Can make it harder to *update* the data structure (you have to preserve the invariant)
- In our case:
  - *find minimum* becomes easier (array is sorted)
  - *insert* becomes harder (must make sure array is sorted afterwards)
- A good invariant will provide some extra structure that makes the operations you want easier
  - sorting the array makes it easier to find the minimum

#### **Trees**





- A *tree* is a hierarchical data structure
  - Each node can have several children but only has one parent
  - The root has no parents; there is only one root
- Example: directory hierarchy







 Most often we use *binary trees*, where each node has at most two children



## Terminology





- The *depth* of a node is the distance from the root
- The *height* of a tree is the number of edges from the root to the deepest leaf
- The *size* of a tree is the number of nodes in it









- Two major modes of tree traversal (visiting all nodes):
  - DFS, depth-first search; for binary trees there are three variants:
    - Pre-order
    - In-order
    - Post-order
  - BFS, breadth-first search
- See the following slides for visualisations



- First current node,
- then left sub tree,
- then right sub tree



• Quiz: sequence?

[F, B, A, D, C, E, G, I, H]



- First left sub tree,
- then current node,
- then right sub tree



• Quiz: sequence?

[A, B, C, D, E, F, G, H, I]



- First left sub tree,
- then right sub tree,
- then current node



• Quiz: sequence?

[A, C, E, D, B, H, I, G, F]



• Each level from left to right



## **Balanced trees**



• A tree can be balanced or unbalanced





- If a tree of size *n* is
  - balanced, its height is  $O(\log n)$
  - unbalanced, its height could be O(n)
- Many tree algorithms have complexity O(height of tree), so are efficient on balanced trees and less so on unbalanced trees
- Normally: balanced trees good, unbalanced bad!

### Heaps

A heap implements a priority queue as a tree. Here is a tree:



 This is not yet a heap. We need to add an *invariant* that makes it easy to find the minimum element.





• A tree satisfies the *heap property* if the value of each node is less than (or equal to) the value of its children:



• Where can we find the smallest element?



- Why did we pick this invariant? One reason:
  - It puts the smallest element at the root of the tree, so we can find it in O(1) time
- Why not just have the invariant "the root node is the smallest"? Because:
  - Trees are a *recursive* structure the children of a node are also trees
  - It's then a good rule of thumb to have a recursive invariant each node of the tree should satisfy the same sort of property
  - In this case, instead of "the root node is smaller than its descendants", we pick "each node is smaller than its descendants"
- General hint: when using a tree data structure, make each node have the same invariant



• A *binary heap* is a *complete* binary tree that satisfies the heap property:



• Complete means that all levels except the bottom one are full, and the bottom level is filled from left to right



- The height is  $O(\log n)$  since  $2^{h} \le n \le 2^{h+1} 1$
- So complete trees are balanced
- If we manage to implement operations with complexity O(h) then they will be O(log n)



- There are a couple of reasons why we choose to have a complete tree:
  - It makes sure the tree is balanced
  - When we insert a new element, it means there is only one place the element can go – this is one less design decision we have to make
- There is a third reason which trumps the first two, but that will have to wait for next time!



- The binary heap invariant:
  - The tree must be *complete*
  - It must have the *heap property* (each node is less than or equal to its children)
- Remember, all our operations must preserve this invariant
- Once we have picked this invariant, there is only one sensible way to implement the operations!











## Adding an element to a binary heap



 Step 1: insert the element at the next empty position in the tree



- This might break the heap invariant!
- $\cdot$  In this case, 12 is less than 66, its parent.



- To modify a data structure with an invariant, we have to
  - modify it,
  - while preserving the invariant
- Often it's easier to separate these:
  - first modify the data structure, possibly breaking the invariant in the process
  - then "repair" the data structure, making the invariant true again
- This is what we are going to do here

## Adding an element to a binary heap

CHALMERS

 Step 2: if the new element is less than its parent, swap it with its parent



 The invariant is still broken, since 12 is less than 29, its new parent

## Adding an element to a binary heap



 Repeat step 2 until the new element is greater than or equal to its parent.



Now 12 is in its right place, and the invariant is restored.
(Think about why this algorithm restores the invariant.)



- At every step, the heap property almost holds except that the new element might be less than its parent
- After swapping the element and its parent, still only the new element can be in the wrong place (why?)





- To remove the minimum element, we are going to follow a similar scheme as for insertion:
  - First remove the minimum (root) element from the tree somehow, breaking the invariant in the process
  - Then repair the invariant
- Replace the root with another element from the tree... which one?
- Because of completeness, we can only really remove the last (bottom-right) element from the tree
  - Solution: first *swap* the root element with the last element, then remove the last element



• Step 1: replace the root element with the *last element* in the tree, and remove the last element



The invariant is broken, because 66 is greater than its children

• Step 2: if the moved element is greater than its children, swap it with its *least child* 



• (Why the least child in particular?)

 Step 3: repeat until the moved element is less than or equal to its children







- Two useful operations we can extract from all this
- Sift up: if an element might be less than its parent, i.e. needs "moving up" (used in insert)
  - Repeatedly swap the element with its parent
- Sift down: if an element might be greater than its children, i.e. needs "moving down" (used in removing the minimum element)
  - Repeatedly swap the element with its least child
- Also called swim and sink



- Implementation of priority queues
  - Heap property means smallest value is always at root
  - *Completeness* means tree is always balanced
- Complexity:
  - find minimum **O(1)**
  - insert, delete minimum O(height of tree), O(log n) because tree is balanced



- Today
  - Main topic was binary heaps, but it was also about how to design data structures
    - The main task is not *how to implement the operations*, but choosing the right representation and invariant
    - These are the main design decisions once you choose them, lots of stuff falls into place
    - Understanding them is the best way to understand a data structure, and checking invariants is a very good way of avoiding bugs!
  - But you also need lots of existing data structures to get inspiration from!
    - Many of these in the rest of the course