Analysis, Design, and Justification (II): Algorithms

Ruben Acuña

Spring 2022

Learning Objectives

The final goal:

– ADJ-CO1: Understand problem solving as a learning and growth process.

– ADJ-CO2: Construct a sound and evidence-based solution to a problem.

– ADJ-CO3: Create solutions to open-ended problems without a specific correct solution.

– ADJ-CO4: Create solutions to ill-defined problems with problems statements which interfere with problem solution.

– ADJ-CO5: Construct solutions based on contextual needs of a problem, team, or customer.

Algorithmic
Analysis, Design, and Justification

ADJ for Algorithms

- For our next topic, we'll look at how to solve algorithm design problems within the ADJ framework.

- Fundamentally, the difference in using ADJ for algorithms is that the result of ADJ will include pseudocode in the design portion. This is huge!

- It's a big deal because it means we have a large "search space" of possible answers we could invent. Maybe that makes our life easier – we're not looking for a needle in a haystack, but it also makes life harder: when justifying our answer, we're competing with an infinite number of possible choices.

A few notes about the arrangement of these slides:

- Each slide is labeled in the top right with its the corresponding ADJ stage.

- Specific techniques for algorithm analysis, design, and justification will be taught JIT style. JIT stands for Just-In-Time, meaning we'll only mention something when it's important.

Justification for Algorithms

– Although we're not at the step of justifying our solution, we should start thinking about justification since it takes a long time and will have a different flavor than previous examples.

– In previous problems, we had statements like "select the data structure". This meant the result was either "list" or "array". Showing one was better only required looking at one alternative.

– In algorithms we have an infinite number of competing solutions. This is addressed by that rather than being *the best*, a solution is *at least as good as the best solution*. The view is that we are defining solution by the constraints it must meet, rather than picking a specific one.

  – For functionality, we say that a solution should have feature X, therefore any algorithm that supports X is at least as good as the best option.

  – For performance, things are trickier. An efficient algorithm is one that runs in polynomial time but often it is implied that an algorithm be as fast as possible. That requires writing a lower bound proof.

ADJ Outline

So, what's the basic plan to deal with an algorithm?

- Analysis:
    - Need to define the problem:
        - *What is meant by updating a PQ? (and heap – how is data represented?).*
        - *Are any (reasonable) assumptions needed?*
    - Need to define metrics to define what makes a good solution.
- Design:
    - Need to create a pseudocode algorithm for the problem.
- Justification:
    - Need to evaluate our metrics from analysis on the design that is produced.

Problem Statement

To get started, let's see the sample design problem:

Design an efficient algorithm to update the priority of an entry inside of a priority queue. Analyze the problem, design an algorithm for updating the priority queue, and justify the algorithm's optimality.

Thoughts...

- How might ADJ be different for an algorithm than other examples we've seen?

  - What will our result for Design look like?

- Is this problem possible? (Or does it appear to be in anyway malformed...?)

- Can this problem (or a portion of it) be solved by something we know about already?

- Are any aspects of this prompt underdefined?

- What should be our first step?

Understanding the Problem

– *Design an efficient algorithm to update the priority of an entry inside of a priority queue. Analyze the problem, design an algorithm for updating the priority queue, and justify the algorithm's optimality.*

Underspecified *aspects:*

– *Is this a max or min PQ? A: This is a max PQ.*

– *What implementation do we need to support? A: Sedgewick's.*

Doing Examples by Hand

Although it might be fun to jump immediately in to writing some code, let's not do that. Let's be sure we know exactly the problem we're trying to solve. Afterall, won't that save us from potentially wasting our time?
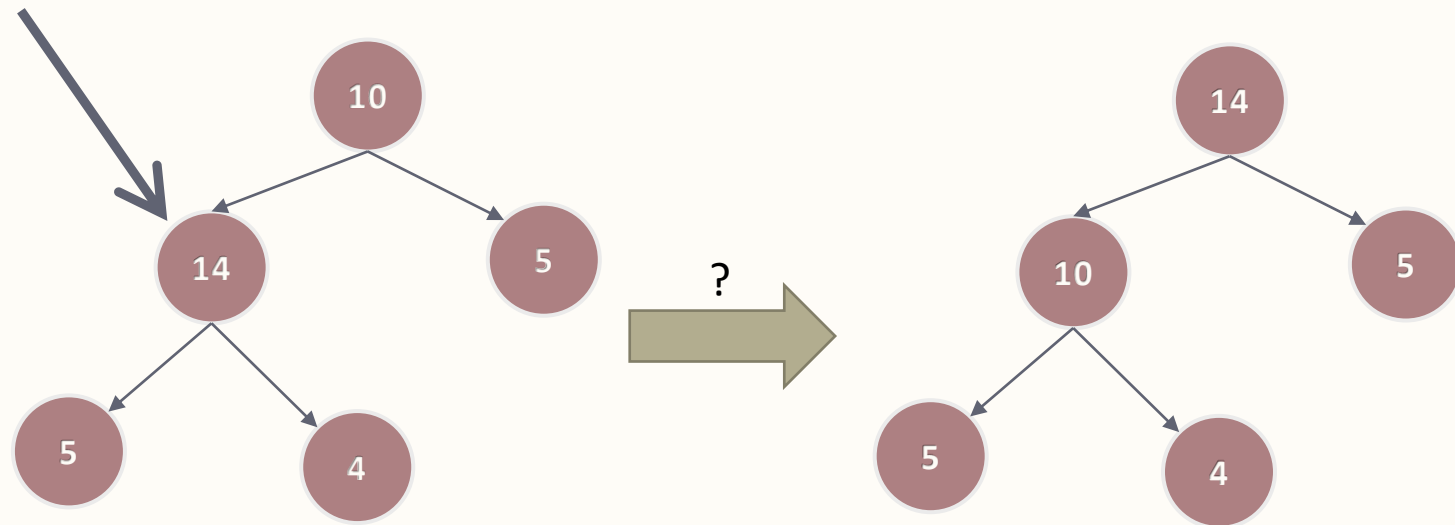
Definitions

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Any ambiguity here?

If there is, definitions are the way we fight it.

Definitions

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Assumptions:
- The priority queue (PQ) is a maximum PQ.
- The specific implementation is Sedgewick's.
- The algorithm will assume that it operates over a valid PQ and produces a valid PQ.

Hopefully, these are all reasonable...?

Definition 1. Binary Tree: A binary tree is a recursive data structure composed of elements called nodes that contain a value, and then references to at most two child nodes.

Definition 2. Complete Binary Tree: a complete binary tree is one where every level is full except the last, and the last level is filled from the left to the right.

Definition 3. Maximum Heap: "A binary tree is heap-ordered if each node is larger than or equal to the keys in that node's two children (if any)." (Sedgewick). The term *key* refers to the label (defining ordering) of an element in a heap, and the term *value* refers to some piece of data that is attached to it.

Definition 4. Heap-ordered Array: We say that an array is heap-ordered if the root element of the heap (if it exists) is stored in at index 1, and where the following formulas may be used to find a parent's (call it p) left (call it c_{left}) and right children (call it c_{right}): $p = \lfloor \frac{k}{2} \rfloor$, $c_{left} = 2k$, $c_{right} = 2k + 1$.

Definition 5. Priority Queue: a priority queue is an abstract data structure that supports adding ("insert") and removing elements ("delMax"). Data is represented as a complete binary tree, and is stored as a heap-ordered array. Per the Sedgewick implementation, both operations take O(logn) time, and both result in a complete and heap-ordered tee. These times will be taken to be optimal. See Algorithm 1.

Definitions

Let's go a step further: let's define what an answer would look like:

−   We know that it will look like pseudocode for an algorithm.

−   We know the input and the output.

Write it up:

Wait, are these not bundled into a class…?

Algorithm 1 Pseudocode for outline of potential solutions.

```
Integer N           //number of entries in PQ
Key[] keys          //contains N elements
Value[] values      //contains N elements

void update(Value val, Key new_pri):
    //Input: the value of the node is changing, and the new priority.
    //Output: updated contents in keys and values.
```

Metrics

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Need to find what is important, distill it, and define it:

– M1: The ability of the algorithm to process a heap sorted array with at most one node which violates the heap rule and produce a heap sorted array with no violations of the maximum heap rule. The array must contain a complete tree at all times.

– M2: The efficiency of the algorithm. For a cost metric, we will use the number of lines run as a measure of computational time needed for a particular design.

Complete Analysis

What we've done:

– We understand the problem.

– We have a sound technical context for the problem (definitions).

– We know what makes a good solution (metrics).

Now we can design an answer.

Remember: the analysis should not be biased in any towards a possible solution and may contain work we ultimately don't need.

Updating a PQ: Design

———

JIT: Pseudocode

- The focus of pseudocode is to **clearly** express an algorithm.

  - No ambiguity allowed!

  - The value is logic – not making the compiler happy.

- Important Note: do not use pseudocode to hide important details. For example: it's only fair to reference an undefined function when it doesn't add ambiguity like calling "distance(Integer x, Integer y)". In contrast, something like "has_deviation(Image original, Image new)" that is supposed to compare images to find a difference isn't okay since we don't know what makes two images different.

Treat as a generic programming language.

Some Rules:
- Use Python style block rules (use colon to start intent, indents define where code belongs, not bracers).
- No semicolons.
- Use words instead of &|! operators.
- For loops should have the form FOR start TO end.
- Use complete type names (e.g., Integer instead of int).
- Avoid ++ or --.

JIT: Pseudocode

Here are some examples (left is Java from Sedgewick's book):   (colors optional)

```java
public void insert(Key v) {
  pq[++N] = v;
  swim(N);
}


private void swim(int k) {
  while (k > 1 && less(k/2, k)) {
    exch(k, k/2);
    k = k/2;
  }
}
```

```
void insert(Key v):
  N = N + 1
  pq[N] = v
  swim(N)



void swim(integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

JIT: K

- K stands for Knowledge and is the symbol which we will use to refer the set of information which is known to be good and real (i.e., true).

  - In a class, K is typically provided as a working document…

- The purpose of K is to make sure that our design is *well-founded.*

- K can serve as a safe place to put generic definitions.

$$K =$$

Definition 1. Binary Tree: A binary tree is a recursive data structure composed of elements called nodes that contain a value, and then references to at most two child nodes.

Definition 2. Complete Binary Tree: a complete binary tree is one where every level is full except the last, and the last level is filled from the left to the right.

Definition 3. Maximum Heap: "A binary tree is heap-ordered if each node is larger than or equal to the keys in that node's two children (if any)." (Sedgewick). The term *key* refers to the label (defining ordering) of an element in a heap, and the term *value* refers to some piece of data that is attached to it.

Definition 4. Heap-ordered Array: We say that an array is heap-ordered if the root element of the heap (if it exists) is stored in at index 1, and where the following formulas may be used to find a parent's (call it p) left (call it c_{left}) and right children (call it c_{right}): $p = \lfloor \frac{k}{2} \rfloor$, $c_{left} = 2k$, $c_{right} = 2k + 1$.

Definition 5. Priority Queue: a priority queue is an abstract data structure that supports adding ("insert") and removing elements ("delMax"). Data is represented as a complete binary tree, and is stored as a heap-ordered array. Per the Sedgewick implementation, both operations take O(logn) time, and both result in a complete and heap-ordered tee. These times will be taken to be optimal. See Algorithm 1.

Note: please refer to the assignment ADJ ruleset for information on the use of K.

Attempt

– We already have a good understanding of our problem: can we break it into pieces?

– This is called *decomposition* and is the standard way to deal with any problem whose solution isn't immediately obvious.

    – A thought: is it always safe to decompose a problem? Can you prove your answer?
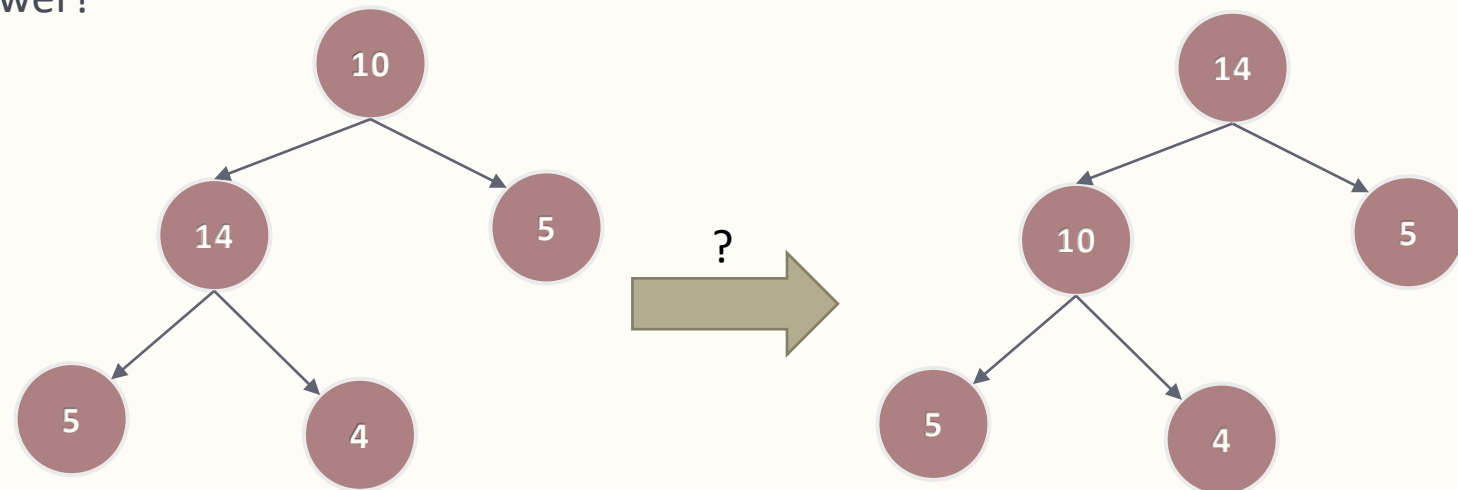
Attempt

Here we go:

– Step 1: find target key/value.

– Step 2: if needed, adjust position of target key/value.

Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N−1:
        if values[i] = val:
            idx = i

    if idx = NULL:
        return

    //Step 2: if needed, adjust position of target key/value.
    keys[idx] = new_pri;
    if less(keys[idx/2], new_pri):
        swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
        sink(idx)
```

Updating a PQ: Justification

Proof of Termination

- A *proof of termination* shows that an algorithm always terminates. It's the first step towards producing something useful – otherwise your program might just be an infinite loop.

- Typically is performed by determining some *progress* metric, and then showing that the algorithm "moves along" that metric.

Examples:

- If we had an algorithm that displayed the contents of an array, we would want to show that is a loop that uses each index only once, always increments, and checks if it is at the end.

- If we had an algorithm to find a path from some place to some other place, we might need to show that it visits every single place once, that no place is unreachable, there is a finite number of places, and it checks if there is no other place left to visit.

Proof of Termination

Example: let's show swim terminates

Proof:

Suppose that k is some number – is a valid index into the array. It must be positive. The only way for this code to not terminate is if the while-loop cannot end. However, in each iteration, k is halved, meaning it is closer to K <= 1 than it was on the previous iteration. Therefore, we monotonically converge on an condition that can break out of the loop. ∎

```
void swim(Integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

This has three simple lines, let's assume it terminates

What is the progress metric?

Proof of Termination

Note: the proof of termination won't occur in our final result for the update() problem. Instead, we will focus on correctness – why?

Proof:

This method will always terminate. Assume that that both less and sink will always terminate. Initially line 2 runs. Then, there is a loop, which goes from 0 to N-1. Since N is the size of the tree, it will be a finite number, hence the loop will run a finite number of times. The code that follows contains no repetition constructs, therefore each line may be executed at most once. Therefore, this algorithm will terminate. ∎

Not that reasonable.

Algorithm 2 Pseudocode for update algorithm.

```
1  void update(Value val, Key new_pri):
2      Integer idx = NULL
3
4      //Step 1: find target key/value
5      for i = 0 to N−1:
6          if values[i] = val:
7              idx = i
8
9      if idx = NULL:
10         return
11
12     //Step 2: if needed, adjust position of target key/value.
13     if less(keys[idx/2], new_pri):
14         swim(idx)
15     else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
16         sink(idx)
```

What is the progress metric?

Proof of Correctness

- A *proof of correctness* shows that an algorithm always performs in a certain way, or always acts according to certain constraints. If we have a proof of correctness, that asserts that our algorithm is "right". The remaining question is of course: is it fast enough?

- Typically is performed by looking at the way that the algorithm works and transforms data to argue that the output will always have certain properties.

Examples:

- If we had an algorithm that displayed the contents of an array, we would want to show that is a loop that uses each index only once, always increments, and checks if it is at the end.

- If we had an algorithm to find a path from some place to some other place, we would need to show if a path existed in the input data, then the algorithm would find (and vice versa).

Proof of Correctness

Example: let's show swim is correct. We need to show that the result of running swim is a valid and complete heap, despite one node potentially being too large.

Notation: Let *parent(k)* indicate the parent k, *left(k)* the left child, *and right(k)* the right child. Since we are using a heap: parent(k) ≥ left(k) and parent(k) ≥ right(k).

Proof: We can trivially see that the tree will complete since the only change that can be made is exchanges.

Let k be the node that will be swim'ed. If k is a node already in its proper position then less(parent(k), k)) will be false, and no code will run. Hence, the result is correct. If k=1, then we know that the heap is correctly ordered and the loop will terminate. In other cases: we swap k with parent(k). In this case, since k > parent(k), and parent(k) ≥ left(k) we see that both k > left(k) and k > parent(k) so moved k will be a valid root of a heap. It is moved. We then repeat again on parent(k), which then falls into one of these three categories repeatedly. ∎

This whole thing falls under the technique of showing that running an algorithm preserves the existing properties of a data set.

```
void swim(integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

⬇

```
void swim(integer k)
  while (k > 1 AND less(parent(k), k)):
    parent = parent(k)
    exch(k, parent)
    k = parent
```

Proof of Correctness

The metric we defined earlier is was:

–   M1: The ability of the algorithm to process a heap sorted array with at most one node which violates the heap rule and produce a heap sorted array with no violations of the maximum heap rule. The array must contain a complete tree at all times.

Here we just need to show that we fulfill (in a yes/no sense) this metric – not too bad.

Proof of Correctness: Metric 1

This can be shown by cases:

Case 1: tree does not contain node with value. In Step 1 of the algorithm, we loop over element and check its contents. No changes are made to N, keys, or value, so after that step, the data will still be heap ordered.

If there is no node to update, and the algorithm will terminate before step 2 (see if-conditional).

Case 2: tree contains node with value. Step 1 will be fine. For Step 2, we defer to the mechanisms for swim and skin which are already known to result in a heap-ordered array that stores a complete tree.

Both functions will terminate if there is no work to do (thus we neglect the case of the node needing to move or not move).

The if-statements evaluates the case of being out of order with parent or children, and follows the formula as defined.

Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N−1:
        if values[i] = val:
            idx = i

    if idx = NULL:
        return

    //Step 2: if needed, adjust position of target key/value
    keys[idx] = new_pri;
    if less(keys[idx/2], new_pri):
        swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
        sink(idx)
```

Proof of Efficiency

The metric we defined earlier was:

– M2: For a cost metric, we will use the number of lines run as a measure of computational time needed for a particular design.

Since we are working with an algorithm, we ideally want to show that our algorithm is at least as fast as any other.

Proof of Efficiency: Metric 2

- Per K, insert takes O(logn). Thus, the best we can get is O(logn) or we would be able to implement insert() faster using update().

- The code has a loop and a call to swim/sink: O(n)+O(logn)=O(n)

- We need to show there is no O(logn) performing algorithm.

- We propose that the linear time loop is a bottleneck.

Proof Sketch:

- We have to search for the index of the target value.

- Looking at one element, we have two children to consider. Moving downward means selecting a child, but since there is no way to tell which is likely to lead to the target, the choice must be arbitrary, and we probably need to backtrack to explore a different path.

- This means we cannot have a log time solution, and linear time is required. ∎

Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N-1:
        if values[i] = val:
            idx = i

    if idx = NULL:
        return

    //Step 2: if needed, adjust position of target key/value
    keys[idx] = new_pri;
    if less(keys[idx/2], new_pri):
        swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
        sink(idx)
```