Final Exam

Algorithms and Applications

2 February 2026

You have 3 hours to answer this final exam. You are allowed 10 A4 sheets (i.e., 20 pages) with the content of your choice. No electronic devices or any other material is allowed. Do not forget to add your name on all your answer sheets and to number them.

The exam consists of a single problem divided into several parts, which are largely independent; it is graded out of 20 points.

A *priority queue* stores objects with real-valued priorities and supports the following operations:

constructor() Create an empty priority queue.

insert(object, priority) Insert an object with a given priority.

extract_max() Remove and return the object with the highest priority, along with the priority.

increase_key(object, priority) Increase the priority of an object.

The purpose of this exam is to study several ways of implementing such a priority queue data structure. Throughout the exam, we denote by n the number of objects currently stored in a priority queue; thus, for example, an operation is said to have complexity $\Theta(n)$ when its complexity is linear in the number of stored objects.

Part A: Warm-up (2 points)

Q1. (1 point) Consider the following sequence of operations:

$$\mathsf{constructor}(),\ \mathsf{insert}(a, 4),\ \mathsf{insert}(b, 7),\ \mathsf{insert}(c, 5),\ \mathsf{increase\_key}(a, 8),$$

followed by three calls to extract_max. What objects are returned, in order?

Q2. (1 point) Propose a way to implement increase_key using only the other operations. If insert has complexity $\Theta(f(n))$ and extract_max has complexity $\Theta(g(n))$, what is the complexity of such an implementation?

Part B: Unsorted Array in Python (5 points)

We implement a priority queue using a Python list storing pairs `(object, priority)` in the order they were inserted in the structure.

Q3. (2.5 points) Complete the following Python class by writing the content of the constructor and of the methods `insert` and `extract_max`. You may assume the queue is non-empty when `extract_max` is called.

```
class PriorityQueue:
    def __init__(self):
        pass

    def insert(self, obj, prio):
        pass

    def extract_max(self):
        pass
```

Q4. (1 point) Write a Python method `increase_key(self, obj, prio)`. You may assume that `obj` is present and that the new priority is larger.

Q5. (1.5 points) Give the worst-case time complexity of `insert`, `extract_max`, and `increase_key`. Briefly justify.

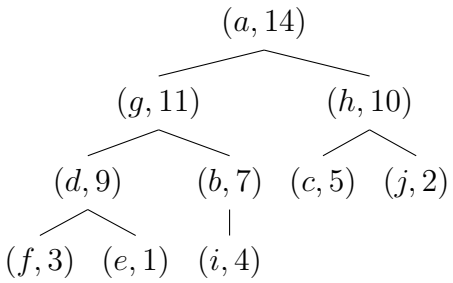
Part C: Binary Heap in C++ (7 points)

One of the data structures we used to design *sorting algorithms* is actually a priority queue: a *heap*, or *binary heap*, is an implementation of a priority queue as a binary tree storing objects together with their priorities. The binary tree is full at all levels except possibly the last one (that is, if the tree has depth p, all nodes at depth $\leqslant p-2$ have exactly two children). In addition, we impose the condition that the priority of a node is always greater than or equal to the priority of its descendants; this is called the *heap property*.

A binary tree with n nodes is stored in an array A of size n as follows: $A[0]$ stores the root node; if a node u is stored at position $A[k]$ and has children u_1 and u_2, then $A[2k+1]$ stores node u_1 and $A[2k+2]$ stores node u_2.

An important operation on binary heaps is the make_heap procedure, which is used to fix a local violation of the heap property. It takes as input the array A and an index i, such that the subtrees rooted at indices $2i+1$ and $2i+2$ in the array A are heaps that satisfy the heap property, but where the priority of node i may be smaller than that of its children. It returns an array A in which the subtree rooted at index i is a valid heap.

Q6. (1 point) The following tree represents a binary heap where priorities are in the second component of every pair:



Write the corresponding array A.

Q7. (1.5 points) Write pseudocode for the procedure make_heap(A, i) that restores the heap property at index i, assuming both subtrees are heaps. You may recall the procedure seen in class, or come up with your own.

Q8. (1 point) In order to implement a binary heap in C++, we design the following generic C++ class able to store objects of any type:

```
#include <vector>
template<class T> class BinaryHeap {
private:
  struct ObjectPriorityPair
  {
    T object;
    int priority;
  };
  std::vector<ObjectPriorityPair> array;

  void make_heap(unsigned index);
public:
  void insert(T o, int p);
  ObjectPriorityPair extract_max();
  void increase_key(T o, int newp);
};
```

Assuming this generic class is implemented, write C++ code fragment to create a heap where objects are of type `std::string`, and then perform the sequence of operations from **Q1.**

Q9. (2.5 points) Assuming `make_heap` is defined as in **Q7.**, propose an implementation in C++ of the methods `insert` and `extract_max`.

Q10. (1 point) Give the worst-case time complexity of insert and extract_max for binary heaps.

Part D: Fibonacci Heap (6 points)

A *Fibonacci heap* is a complex data structure for priority queues implemented as a collection of t trees, each of which verifying the *heap condition*: the priority of a node is always greater

than or equal to that of its descendants. In addition, we store additional information:

- The roots of the trees are stored in a *circular* doubly linked list (a circular list does not have a specific first or last element, each element points to the next and previous one, in a cyclic manner).
- In each tree, each node stores pointers to its parent (except for the root of the tree), to one of its children (if any), and to its left and right siblings (as a circular doubly linked list, so that the right sibling of the rightmost child of a node is the leftmost child).
- Each node has a Boolean *mark*, initially false. The mark is set to true if the node *has lost a child since the last time it changed parents.*
- A pointer *max* points to the root with maximum priority.
- For each node u, we store the number $\delta(u)$ of its children.

Q11. (0.5 point) Describe how to implement the constructor of an empty Fibonacci heap.

Q12. (1 point) Propose a simple implementation of insert and argue why its actual (non-amortized) cost is $O(1)$.

Q13. (1.5 points) Consider the following potential function: $\phi(H) = \gamma \cdot (t + 2m)$, where t is the number of trees in the heap, m the number of marked nodes (i.e., the number of nodes whose mark is set to true), and $\gamma > 0$ a constant.

a) (0.5 point) What is the potential of an empty Fibonacci heap?

b) (1 point) For an operation x that transforms a Fibonacci heap H into a new Fibonacci heap $x(H)$ with cost $c_x(H)$, we define $\hat{c}_x(H) \coloneqq c_x(H) + \phi(x(H)) - \phi(H)$. Show that for any sequence of operations $x_1 \dots x_k$ starting from the empty Fibonacci heap H_0, with $H_i \coloneqq x_i(H_{i-1})$,

$$\frac{1}{k}\sum_{i=1}^{k} c_{x_i}(H_{i-1}) \leqslant \frac{1}{k}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}).$$

We therefore call $\hat{c}_x(H)$ the *amortized cost* of operation x, and we will use it to characterize the complexity of operations in Fibonacci heaps.

Q14. (2 points) The operation extract_max proceeds as follows: the maximum root is removed; its children become new roots; then trees whose root has the same number of children are repeatedly merged until all roots have a different number of children. To do so, whenever two roots have the same number of children, one of them becomes a child of the other (while preserving the heap property), and this process is repeated until all roots have distinct degrees. During these operations, the mark is updated whenever necessary.

Show that, with an appropriate choice of γ, the amortized complexity of extract_max is

$$O\left(\max\left(\max_{u \text{ node of } H} \delta(u), \max_{u \text{ node of extract\_max}(H)} \delta(u)\right)\right).$$

Q15. (1 point) We will accept without proof that, in a Fibonacci heap:

- increase_key has amortized complexity $O(1)$;
- the maximum degree of any node is $O(\log n)$.

What can you conclude about the complexity of extract_max? How do Fibonacci heaps compare to other priority queue implementations?