МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В.Н.Каразіна
Факультет математики і інформатики
Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

магістр

на тему Доведення коректності алгоритмів на чисто функціональних структурах даних за допомогою The Coq Proof Assistant

Виконав: студент 6 курсу, групи МФ-61
спеціальність 122 «Комп'ютерні науки»
освітньо-наукова програма «Інформатика»

Єна А. В.
(прізвище та ініціали)
Керівник Жолткевич Г. М.
(прізвище та ініціали)
Рецензент
(прізвище та ініціали)

Харків – 2024 року

АНОТАЦІЯ

У цій науковій роботі проводиться детальне дослідження та формальне доведення властивостей пріоритетних черг на основі структури даних Кучі Бродала-Окасакі. Основна увага приділяється розгляду й аналізу методів, що забезпечують ефективну та надійну реалізацію чисто функціональних структур даних, зокрема, у сферах, де критично важливі безпека та надійність. Результати дослідження демонструють, що успішне формальне доведення властивостей Кучі Бродала-Окасакі відкриває нові перспективи для їх застосування у комерційних та промислових системах.

Метою даної роботи є підтвердження потенціалу чисто функціональних структур даних через систематичне формальне доведення їх властивостей, що може сприяти оптимізації та надійності програмного забезпечення, яке використовує пріоритетні черги. Висновки з цього дослідження не тільки підтверджують значимість Кучі Бродала-Окасакі як ефективної чисто функціональної структури даних, але й наголошують на важливості подальших розробок та досліджень у цьому напрямку.

Загальна характеристика роботи: робота містить вступ, основну частину, висновок, список використаної літератури та додаток. Кількість сторінок - 64, кількість ілюстрацій - 4, кількість використаних джерел - 10, кількість лістингів - 25.

Ключові слова: Куча Бродала-Окасакі, чисто функціональні структури даних, пріоритетні черги, формальна верифікація, надійність програмного забезпечення.

ANNOTATION

In this scientific paper, a detailed study and formal proof of the properties of priority queues based on the Brodahl-Okasaki Kucha data structure is carried out. The focus is on the consideration and analysis of methods that ensure efficient and reliable implementation of purely functional data structures, in particular, in areas where security and reliability are critical. The results of the study demonstrate that the successful formal proof of the properties of the Brodal-Okasaki Kucha opens up new perspectives for their application in commercial and industrial systems.

The purpose of this work is to confirm the potential of purely functional data structures through a systematic formal proof of their properties, which can contribute to the optimization and reliability of software that uses priority queues. The findings from this study not only confirm the significance of the Brodahl-Okasaki Heap as an efficient purely functional data structure, but also emphasize the importance of further development and research in this direction.

General characteristics of the work: the work contains an introduction, the main part, a conclusion, a list of used literature and an appendix. Number of pages - 64, number of illustrations - 4, number of used sources - 10, number of listings - 25.

Keywords: Brodal-Okasaki heap, purely functional data structures, priority queues, formal verification, software reliability.

ЗМІСТ

ВСТУП

Для формулювання мети роботи треба дати визначення понять “Чисто функціональна структура даних” та “коректність алгоритмів” з теми кваліфікаційної роботи, а також термінів “Сертифіковане програмування” і “Сертифікат корректності програми”.

Чисто функціональна структура даних — це така структура даних, яка повністю відповідає принципам функціонального програмування. Основною її особливістю є немутабельність (незмінність) даних. Це означає, що після створення екземпляра структури даних його стан не може бути змінений. Якщо потрібно модифікувати таку структуру, функції, які це роблять, повертають новий екземпляр структури з необхідними змінами, залишаючи оригінальний екземпляр незмінним.

Коректність алгоритмів — це властивість алгоритму, яка вказує на його здатність вирішувати поставлену задачу згідно зі специфікацією, тобто на здатність алгоритму виконувати визначені функції або вирішувати задачі, для яких він був розроблений. Оцінка коректності алгоритму заснована на двох основних критеріях: відповідності специфікації та завершеності.

1. Відповідність специфікації означає, що алгоритм вирішує задачу правильно на всіх вхідних даних, що відповідають специфікації. Іншими словами, для кожного вхідного набору даних, який відповідає вимогам задачі, алгоритм повинен забезпечувати очікуваний результат.
2. Завершеність означає, що алгоритм завжди закінчує свою роботу за скінченну кількість кроків для будь-якого вхідного набору даних, який відповідає специфікації.

Сертифіковане програмування — це підхід до розробки програмного забезпечення, при якому використовуються формальні методи для доведення коректності програми відповідно до її специфікації. Цей процес включає математичне доведення того, що програмний код веде себе так, як очікується, на всіх можливих вхідних даних і в усіх можливих станах виконання.

Сертифікат коректності — це документ або формальне свідоцтво, що підтверджує, що певний алгоритм, програмний код або програмне забезпечення відповідає заздалегідь визначеним специфікаціям і вимогам коректності. Це означає, що доведено, за допомогою формальних методів, що програма або алгоритм працює правильно у всіх можливих умовах використання згідно зі своєю специфікацією, виконуючи всі задані функції без помилок і непередбачених поведінок.

Метою цієї роботи є доведення аксіоматики структури даних “пріоритетна черга”, формалізована в [1], при реалізації з використанням Купи Бродала-Окасакі [8], за допомогою методів сертифікованого програмування задля отримання сертифікату коректності реалізації та подальшого його використання.

Виходячи з мети були сформульовані завдання, які описані в самому дослідженні, так як для них треба ввести багато визначень з попередніх досліджень.

В ході дослідження було запропоновано використовувати The Coq Proof assistant [4] в якості системи автоматичних доведень, яка дає змогу отримати сертифікат коректності реалізації, яка розглядатиметься в роботі.

Актуальність роботи не може бути переоцінена в контексті сучасних вимог до надійності та коректності програмного забезпечення. У світлі зростаючого застосування комп'ютерних систем у критично важливих сферах, таких як фінансові послуги, медицина та авіація, де помилка у програмному забезпеченні може мати фатальні наслідки, важливість розробки та впровадження формально верифікованих структур даних є

очевидною. Реалізація та сертифікація коректності пріоритетної черги, однієї з фундаментальних структур даних у комп'ютерних науках, забезпечує не лише теоретичний внесок у розвиток області формальних методів і сертифікованого програмування, але й практичну цінність у підвищенні надійності програмних систем, що використовують цю структуру.

Поставлені задачі були частково виконані, а результати - проаналізовані. Результуючий сертифікат покриває більшість вимог, які від нього вимагаються, де доведення припущених теорем залишене для подальших досліджень.

Робота містить вступ, основну частину, висновок, список використаної літератури та додаток. Кількість сторінок - 64, кількість ілюстрацій - 4, кількість використаних джерел - 10, кількість лістингів - 25.

ЧАСТИНА 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Обґрунтування необхідності дослідження

У контексті безперервного прогресу інформаційних технологій та збільшення складності програмних систем, питання надійності, безпеки та ефективності програмного забезпечення стає дедалі актуальнішим. Зокрема, це стосується розробки та впровадження критично важливих систем, які використовуються в таких сферах, як медицина, авіація, автомобільна промисловість та фінансові технології. У цих галузях, навіть найменша помилка у програмному забезпеченні може призвести до серйозних фінансових втрат, порушення робочих процесів, а в деяких випадках - до загрози людським життям. Тому, глибоке розуміння та верифікація основних компонентів програмних систем, таких як алгоритми та структури даних, є ключовим аспектом забезпечення високого рівня надійності та безпеки програмного забезпечення.

Пріоритетні черги є однією з таких фундаментальних структур даних, яка знайшла широке застосування в різноманітних алгоритмах та програмних системах. Від ефективності їх реалізації значною мірою залежать швидкість обробки даних і загальна продуктивність програм. Купа Бродала-Окасакі, як передова структура для реалізації пріоритетних черг, особливо цінується в контексті функціонального програмування за свою високу ефективність та надійність. Враховуючи це, детальне дослідження та верифікація цієї структури в контексті її застосування для пріоритетних черг набуває особливого значення.

Використання методів сертифікованого програмування для аналізу та доведення коректності реалізації пріоритетної черги на основі Купи Бродала-Окасакі є стратегічно важливим підходом. Це дозволяє не тільки підтвердити відповідність реалізації визначеним специфікаціям, але й

забезпечити високий рівень довіри з боку розробників та користувачів програмного забезпечення. Застосування формальних методів дозволяє отримати математичні докази коректності алгоритмів, що є набагато надійнішим підходом, ніж традиційні методи тестування, які не можуть гарантувати виявлення всіх потенційних помилок.

Використання методів сертифікованого програмування для аналізу та доведення коректності реалізації пріоритетної черги на основі Купи Бродала-Окасакі є стратегічно важливим підходом. Це дозволяє не тільки підтвердити відповідність реалізації визначеним специфікаціям, але й забезпечити високий рівень довіри з боку розробників та користувачів програмного забезпечення. Застосування формальних методів дозволяє отримати математичні докази коректності алгоритмів, що є набагато надійнішим підходом, ніж традиційні методи тестування, які не можуть гарантувати виявлення всіх потенційних помилок.

Окрім того, це дослідження відкриває широкі перспективи для подальшого вдосконалення технік верифікації програмного забезпечення, розробки нових інструментів для автоматизації процесів верифікації, а також створення надійніших та безпечніших програмних продуктів. У довгостроковій перспективі, результати такого дослідження можуть сприяти стандартизації методів сертифікованого програмування та їх ширшому застосуванню в індустрії програмного забезпечення, особливо в галузях з високими вимогами до надійності та безпеки.

Таким чином, актуальність даного дослідження полягає не лише в його практичному значенні для підвищення якості та надійності програмних систем, але й у внеску в теоретичні основи комп'ютерних наук та розвиток методів формальної верифікації. Це дослідження сприятиме подальшому прогресу в області сертифікованого програмування, відкриваючи нові можливості для забезпечення високого рівня безпеки та надійності в критично важливих програмних системах.

1.2 Огляд сучасного стану справ в області дослідження

Дана робота використовує декілька результатів, які були отримані в декількох різних дослідженнях.

Точне визначення того, хто перший запропонував поняття пріоритетної черги, може бути складним завданням, оскільки концепція пріоритетних черг виникла як природне розширення ідей в області обчислювальної техніки та алгоритмічних структур даних. Це поняття розвивалося протягом кількох десятиліть, і багато ранніх робіт внесли вклад у його формування та уточнення.

Пріоритетна черга, в основному, це абстрактна структура даних, що підтримує операції вставки та, зазвичай, вилучення мінімального або максимального елементу. Визначення пріоритетної черги в академічній літературі часто акцентує на її властивості та операціях, які можна здійснювати, але рідше вказує на конкретного автора, як винахідника цієї концепції.

Однак, якщо говорити про формальне визначення пріоритетної черги та її популяризацію, то варто згадати роботи з області алгоритмів та структур даних, такі як роботи Едсгера Дейкстри та Роберта Тар'яна, які зробили значний внесок у розвиток та застосування пріоритетних черг у комп'ютерних науках. Їхні роботи допомогли закласти основи для розуміння та використання пріоритетних черг в різноманітних алгоритмах, таких як алгоритми пошуку найкоротших шляхів.

Визначення пріоритетної черги з однієї з класичних робіт може звучати приблизно так: "Пріоритетна черга - це структура даних, яка підтримує дві основні операції: вставку елемента з довільним пріоритетом і вилучення елемента з найвищим пріоритетом. Ця структура дозволяє швидко доступатися до елемента з найвищим пріоритетом за час, що є функцією від кількості елементів у черзі, незалежно від їх порядку вставки."

Таке визначення відображає загальне розуміння пріоритетних черг як важливої структури даних, що забезпечує ефективне управління даними з урахуванням їх пріоритетності, що є ключовим для багатьох алгоритмічних задач.

Одним з ранніх описів бінарних куч і їх застосування для реалізації пріоритетних черг було наведено в [1].

Так як поняття пріоритетної черги є ключовим для подальшого дослідження, нижче приведено перекладені цитати з цієї публікації.

Черга з пріоритетами - це АТД (абстрактний тип даних), заснований на моделі множин з операторами INSERT і DELETEMIN, а також з оператором MAKENULL для ініціалізації структури даних перед визначенням нового оператора DELETEMIN спочатку пояснимо, що таке "черга з пріоритетами". Цей термін передбачає, що у багатьох елементів задана функція пріоритету (priority), тобто. для кожного елемента множини можна обчислити функцію p(a), пріоритет елемента a, яка зазвичай приймає значення з множини дійсних чисел, або, у більш загальному випадку, з деякої лінійно впорядкованої множини. Оператор INSERT для черг з пріоритетами розуміється у звичайному сенсі, тоді як DELETEMIN є функцією, яка повертає елемент з найменшим пріоритетом і як побічний ефект видаляє його з безлічі. Таким чином, виправдовуючи свою назву, DELETEMIN є комбінацією операторів DELETE та MIN.

Назва "черга з пріоритетами" походить від того виду упорядкування (сортування), якому піддаються дані цього АТД. Слово "черга" припускає, що люди (або вхідні елементи) чекають деякого обслуговування, а слова "з пріоритетом" позначають, що обслуговування буде здійснюватися не за принципу "перший прийшов - першим отримав обслуговування", як відбувається з АТД QUEUE (Черга), а на основі пріоритетів усіх персон, що стоять у черзі. Наприклад, у приймальному відділенні лікарні спочатку приймають пацієнтів з потенційно фатальними діагнозами, незалежно від того, як довго вони чи інші пацієнти перебували у черзі.

Реалізація пріоритетних черг за допомогою бінарних куп має наступні асимптотичні оцінки:

Таблиця 1.2.1. Асимптотичні оцінки операцій бінарної купи.

Операція	Асимптотична оцінка
INSERT	O(log n), де n - довжина черги
DELETEMIN	O(log n), де n - довжина черги

Ця дослідницька робота не зосереджується на аналізі асимптотичних характеристик структур даних; таким чином, формальні докази їх асимптотичної поведінки не будуть представлені. Відповідні докази та аналітичні оцінки детально розглянуті в наукових публікаціях, які безпосередньо досліджують зазначені структури даних. У контексті цього дослідження, згадки про асимптотичні оцінки мають на меті лише ілюструвати специфічні переваги, які одна структура даних може мати у порівнянні з іншими, забезпечуючи тим самим контекстуальний огляд їх потенційного застосування.

У рамках цієї дослідження, структура даних "Біноміальна куча" відіграє важливу роль, оскільки Куча Бродала-Окасакі розроблена на основі цієї самої структури. Концепція біноміальної кучі була вперше представлена Жаном Вюйменом у 1978 році, як зазначено у публікації [10]. Біноміальна куча є структурою даних, яка забезпечує ефективну реалізацію операцій, типових для пріоритетних черг, включаючи вставку, пошук та видалення мінімального елементу, а також об'єднання куч. Однією з визначальних характеристик біноміальної кучі є її чисто-функціональна природа.

Особливо релевантним для даного дослідження є визначення, надане Крісом Окасакі, яке виокремлює чисто-функціональні аспекти біноміальної кучі, відображене в джерелі [8]. Це визначення подається з акцентом на чисто-функціональність, що є критично важливим для аналізу

та розуміння структури даних у контексті цього дослідження. Далі наведено цитату з даного джерела.

Біноміальні черги, які ми, щоб уникнути плутанини з чергами FIFO, називатимемо біноміальними купами (binomial heaps) – ще одна поширена реалізація куп. Біноміальні купи влаштовані складніше, ніж ліво орієнтовані, і, на перший погляд, не відшкодовують цю складність жодними перевагами. Однак у наступних розділах ми побачимо, як у різних варіантах біноміальних куп можна змусити insert і merge виконуватися за час O(1).

Біноміальні купи будуються з найпростіших об'єктів, які називаються біноміальними деревами. Біноміальні дерева індуктивно визначаються так:

- Біноміальне дерево рангу 1 є одиночним вузолом.
- Біноміальне дерево рангу r + 1 виходить шляхом зв'язування (linking) двох біноміальних дерев рангу r, так що одне з них стає найлівішим нащадком іншого.

З цього визначення видно, що біноміальне дерево рангу містить рівно 2^r елементів. Існує друге, еквівалентне першому, визначення біноміальних дерев, яким іноді зручніше користуватися: біноміальне дерево рангу r являє собою вузол з r нащадками $t_1 \ldots t_r$, де кожне t_i є біноміальним деревом рангу r - i. На рисунку 1.2.1 показано біномні дерева рангів від 0 до 3.

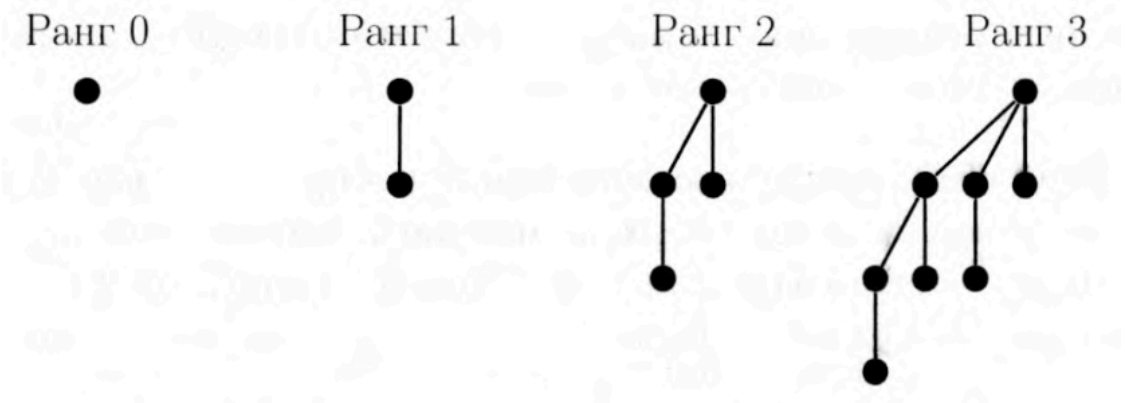

Рисунок 1.2.1. Біноміальні дерева рангів 0-3.

Ми представляємо вершину біноміального дерева у вигляді елемента та списку його нащадків. Для зручності ми також помічаємо кожний вузол його рангом. Кожен список нащадків зберігається в порядку спадання рангів, а елементи зберігаються з порядком купи. Щоб зберігати цей порядок, ми завжди прив'язуємо дерево з більшим коренем до дерева з меншим коренем.

```
datatype Tree = Node of int x Elem.T x Tree list

fun link (t1 as Node(r, x1, c1), t2 as Node(_, x2, c2)) =
     if Elem.leq(x1, x2) then Node(r + 1, x1, t2 :: c1)
     else Node(r + 1, x2, t1 :: c2)

type Heap = Tree list
```

Лістинг 1.2.1: Визначення Типу даних Біноміальне дерево, та операції зв’язування двох дерев. Визначення Типу Біноміальна купа.

Ми завжди пов'язуємо дерева одного рангу.

Тепер визначаємо біномну купу як колекцію біномних дерев, кожне р яких має порядок купи, і жодні два дерева не збігаються на ранг. Ми представляємо цю колекцію у вигляді списку дерев у зростаючому порядку рангу.

Оскільки кожне біноміальне дерево містить 2^r елементів, і ніякі два дерева по рангу не збігаються, дерева розміру n точно відповідають одиницям у двійковому поданні n. Наприклад, число 21 у двійковому вигляді виглядає як 10101, і тому біноміальна купа розміру 21 містить одне дерево рангу 0, одне рангу 2 і одне рангу 4 (розмірами, відповідно, 1, 4 і 16). Зауважимо, що як і, як двійкове уявлення n містить не більше $\lfloor \log(n + 1) \rfloor$ одиниць, біноміальна купа розміру n містить не більше $\lfloor \log(n-\- 1) \rfloor$ дерев.

Тепер ми готові описати функції, що діють на біноміальних деревах. Починаємо ми з insert і merge, які визначаються приблизно аналогічно до складання двійкових чисел. Щоб внести елемент у купу, ми спочатку створюємо одноелементне дерево (тобто біноміальне дерево рангу 0), потім піднімаємося за списком існуючих дерев у порядку зростання рангів, пов'язуючи при цьому однорангові дерева. Кожне зв'язування відповідає перенесення у двійковій арифметиці.

```
fun rank (Node (r, x, c)) = r

fun insTree (t, []) = [t]
  | insTree (t, ts as t'::ts')
     if rank t < rank t' then t::ts
                         else insTree (link (t, t'), ts')

fun insert (x, ts) = insTree (Node (0, x, []), ts)
```

Лістинг 1.2.2: Визначення операцій insTree та insert для біноміальної купи.

У гіршому випадку, при вставці в купу розміру $n = 2^k - 1$, потрібно k зв'язувань і $O(k) = O(\log n)$ часу.

При злитті двох куп ми проходимо через обидва списки дерев у порядку зростання рангу і пов'язуємо по дорозі дерева рівного рангу. Як і раніше, кожне зв'язування відповідає переносу в двійковій арифметиці.

```
fun merge (ts1, []) = ts1
  | merge ([], ts2) = ts2
  | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
     if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
     else if rank t2 < rank t1 then t1 :: merge (ts1, ts2')
     else insTree (link (t1, t2), merge (ts1', ts2'))
```

Лістинг 1.2.3: Визначення операції merge для біноміальної купи.

Функції findMin і deleteMin викликають допоміжну функцію removeMinTree, яка знаходить дерево з мінімальним коренем, виключає його зі списку і повертає як це дерево, так і список дерев, що залишилися.

Функція findMin просто повертає корінь знайденого дерева.

```
fun removeMinTree [t] = (t, [])
  | removeMinTree (t :: ts) =
     let val (t', ts') = removeMinTree ts
     in if Elem.leq (root t, root t') then (t, ts)
        else (t', t :: ts') end

fun findMin ts = let val (t, _) = removeMinTree ts in
     root t end
```

Лістинг 1.2.4: Визначення операцій findMin та removeMinTree.

Функція deleteMin влаштована трохи більш хитро. Відкинувши корінь знайденого дерева, ми ще повинні повернути його нащадків до списку інших дерев. Зауважимо, що список нащадків майже відповідає визначенню біноміальної купи. Це колекція біноміальних дерев з неповторними рангами, але тільки відсортована вона не за зростанням, а за зменшенням рангу. Таким чином, звернувши список нащадків, ми перетворимо його в біноміальну купу, а потім зливаємо з деревами, що залишилися.

```
fun deleteMin ts = let val (Node(_, x, ts1), ts2) =
     removeMinTree ts in merge(rev ts1, ts2) end
```

Лістинг 1.2.5: Визначення операції deleteMin.

Повна реалізація біноміальних куп приведена на рисунку 1.2.2. Усі чотири основні операції в гіршому випадку вимагають O(log n) часу.

Ця конкретна реалізація становитиме основу для проведення нашого дослідження, оскільки вона не лише задовольняє всі вимоги до чистої функціональності, а й надає коректно реалізовану імплементацію, що є критично важливим для точності та ефективності вивчення обраної теми.

```
functor BinomialHeap (Element: ORDERED): HEAP =
struct
  structure Elem = Element
  datatype Tree = Node of int × Elem.T × Tree list
  type Heap = Tree list

  val empty = []
  val isEmpty ts = null ts

  fun rank (Node (r, x, c)) = r
  fun root (Node (r, x, c)) = x

  fun link (t1 as Node (r1, x1, c1), t2 as Node (r2, x2, c2)) =
    if Elem.leq (x1, x2) then Node(r+1, x1, t2 :: c1)
    else Node(r+1, x2, t1 :: c2)

  fun insTree (t, []) = [t]
    | insTree (t, ts as t' :: ts') =
        if rank t < rank t' then t :: ts else insTree (link(t, t'), ts')
  fun insert (x, ts) = insTree (Node(0, x, []), ts)

  fun merge (ts1, []) = ts1
    | merge ([], ts2) = ts2
    | merge (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
        if rank t1 < rank t2 then t1 :: merge (ts1', ts2)
        else if rank t2 < rank t1 then t2 :: merge (ts1, ts2')
        else insTree (link (t1, t2), merge (ts1', ts2'))

  fun removeMinTree []= raise EMPTY
    | removeMinTree [t] = (t, [])
    | removeMinTree (t :: ts) =
        let val (t', ts') = removeMinTree ts
        in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end

  fun findMin ts = let val (t, _) = removeMinTree ts in root t end

  fun deleteMin ts =
    let val (Node(_, x, ts1), ts2) = removeMinTree ts
    in merge (rev ts1, ts2) end
end
```

Рисунок 1.2.2. Повна реалізація біноміальних куп.

У даному дослідженні, на відміну від попередніх робіт, особлива увага приділяється аналізу операції злиття (merge) двох незалежних куп в єдину структуру. Операція злиття є ключовою для багатьох алгоритмів, що використовують купи, оскільки дозволяє ефективно об'єднувати дві структури даних з збереженням властивостей порядку.

Для бінарних куп часова складність операції злиття становить O(n + m), де n і m - розміри злитих куп. Ця асимптотика може бути не оптимальною для задач, де часто вимагається об'єднання великих куп. На противагу цьому, у біноміальних кучах операція злиття реалізована значно ефективніше та забезпечує час виконання в межах O(log n), що є важливим покращенням для застосувань, які потребують високої ефективності обробки даних.

Операція злиття у біноміальних кучах особливо цінна у контексті алгоритмів, що використовують часті об'єднання куп, таких як покращені алгоритми пошуку найкоротшого шляху та інші задачі оптимізації. Ефективне злиття дозволяє значно знизити загальну витрату часу на обробку куп, сприяючи підвищенню продуктивності системи в цілому. Тому, розгляд цієї операції у контексті біноміальних куп є важливим аспектом даного дослідження.

Реалізація пріоритетних черг за допомогою біноміальних куп має наступні асимптотичні оцінки

Таблиця 1.2.2. Асимптотичні оцінки оперицій біноміальної купи.

Операція	Асимптотична оцінка
INSERT	O(log n), де n - довжина черги
DELETEMIN	O(log n), де n - довжина черги
MERGE	O(log n), де n - довжина черги

Структура даних скошена біноміальна куча була вперше запропонована в 1987 році Майклом Л. Фредманом і Робертом Е. Тар'яном в [6]. Ця стаття представила не тільки фібоначчієві кучі, але й згадувала про skew binomial heaps як варіацію біноміальних куч, що покращують деякі аспекти їх виконання, зокрема зменшення часу на операції вставки. Далі наведено цитату з [8].

Скошена біноміальна купа являє собою список скошених біноміальних дерев, упорядкованих у порядку купок, відсортованих за зростанням рангу, і тільки два перші дерева можуть мати однаковий ранг. Оскільки скошені біноміальні дерева одного рангу можуть мати різний розмір, тут уже немає прямої відповідності між деревами в купі і цифрами скошеного двійкового числа, що становить розмір купи. Наприклад, хоча скошене двійкове уявлення числа 4 дорівнює 11, скошена биномиальная купа розміру 4 може містити або одне дерево рангу 2 розміру 4, або два дерева рангу 1 розміром 2, або дерево рангу 1 розміром 3 п дерево рангу 0, або дерево рангу 1 ром 2 і два дерева рангу 0. Однак максимальна кількість дерев у купі, як і раніше, дорівнює O(log n).

Велика перевага скошених биномиальных куп полягає в тому, що новий елемент вставляється за час O(1). Спочатку ми порівнюємо ранги двох найменших дерев. Якщо вони збігаються, ми робимо скошене зв'язування нового елемента з цими деревами. В іншому випадку ми створюємо нове дерево одноелементне і додаємо його до початку списку.

На рисунку 1.2.3 наведена реалізація скошеної біноміальної черги взята з [8].

Реалізація пріоритетних черг за допомогою скошених біноміальних куп має наступні асимптотичні оцінки:

```
functor SkewBinomialHeap (Element: ORDERED): HEAP = struct
  structure Elem = Element
  datatype Tree = NODE of int × Elem.T × Elem.T list × Tree list
  type Heap = Tree list
  val empty = []
  fun isEmpty ts = null ts
  fun rank (NODE (r, x, xs, c)) = r
  fun root (NODE (r, x, xs, c)) = x
  fun link (t1 as NODE (r, x1, xs1, c1), t2 as NODE (_, x2, xs2, c2)) =
        if Elem.leq (x1, x2) then NODE (r+1, x1, xs1, t2 :: c1)
        else NODE (r+1, x2, xs2, t1 :: c2)
  fun skewLink (x, t1, t2) =
        let val NODE (r, y, ys, c) = link (t1, t2)
        in  if Elem.leq (x, y) then NODE (r, x, y :: ys, c)
            else NODE (r, y, x :: ys, c)
        end
  fun insTree (t, []) = [t]
    | insTree (t1, t2 :: ts) = if rank t1 <rank t2 then t1 :: t2 :: ts
                               else insTree (link (t1, t2), ts)
  fun mergeTrees (ts1, []) = ts1 | mergeTrees ([], ts2) = ts2
    | mergeTrees (ts1 as t1 :: ts1', ts2 as t2 :: ts2') =
        if rank t1 <rank t2 then t1 :: mergeTrees (ts1', ts2)
        else if rank t2 <rank t1 then t2 :: mergeTrees (ts1, ts2')
        else insTree (link (t1, t2), mergeTrees (ts1', ts2'))
  fun normalize [] = [] | normalize (t :: ts) = insTree (t, ts)
  fun insert (x, ts as t1 :: t2 :: rest) =
        if rank t1 = rank t2 then skewLink (x, t1, t2) :: rest
        else NODE (0, x, [], []) :: ts
    | insert (x, ts) = NODE (0, x, [], []) :: ts
  fun merge (ts1, ts2) = mergeTrees (normalize ts1, normalize ts2)
  fun removeMinTree []= raise EMPTY
    | removeMinTree [t] = (t, [])
    | removeMinTree (t :: ts) =
        let val (t', ts') = removeMinTree ts
        in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end
  fun findMin ts = let val (t, _) = removeMinTree ts in root t end
  fun deleteMin ts =
        let val (NODE (_, x, xs, ts1), ts2) = removeMinTree ts
            fun insertAll ([], ts) = ts
              | insertAll (x :: xs, ts) = insertAll (xs, insert (x, ts))
        in insertAll (xs, merge (rev ts1, ts2)) end
end
```

Рисунок 1.2.3. Повна реалізація скошених біноміальних куп.

Таблиця 1.2.3. Асимптотичні оцінки операцій скошеної біноміальної купи.

Операція	Асимптотична оцінка
INSERT	O(1)
DELETEMIN	O(log n), де n - довжина черги
MERGE	O(log n), де n - довжина черги

Перше визначення та детальний опис Кучі Бродала-Окасакі було надано у науковій роботі, опублікованій Гертрудом Бродалом і Крісом Окасакі. Структура даних, відома як Куча Бродала-Окасакі, є прикладом чисто функціональної кучі та була представлена в [3].

Ця робота являє собою значний внесок у розвиток чисто функціональних структур даних, оскільки вона вперше ввела концепцію кучі, яка повністю відповідає принципам функціонального програмування, забезпечуючи ефективність операцій, яка раніше була доступна лише у мутабельних структурах.

Також, Куча Бродала-Окасакі була детально описана Крісом Окасакі у [8]. У даному джерелі вона відома під назвою "Bootstrapped Heap" або у деяких випадках просто як "Functional Heaps".

"Bootstrapped Heap" як частина Кучі Бродала-Окасакі використовує інноваційну структуру, яка дозволяє підтримувати операції, необхідні для пріоритетних черг, на високому рівні ефективності з точки зору часу виконання, при цьому залишаючись в межах функціональних мов програмування, де імутабельність є ключовою характеристикою. Ця структура дозволяє виконувати оперативні операції вставки, пошуку та видалення мінімального елемента з амортизованими гарантіями часу, що є значним досягненням для чисто функціональних структур даних. Далі наведено цитату з даного джерела.

У цьому розділі ми використовуємо структурну абстракцію для куп і отримуємо ефективну операцію злиття.

Припустимо, у нас є реалізація куп, що підтримує insert за час O(1) у гіршому випадку, а merge, findMin та deleteMin за час O(log n) у гіршому випадку. Одна така реалізація — скошені біномні купи з розділу 9.3.2; ще одна біноміальні купи з розкладом із розділу 7.3. За допомогою структурної абстракції ми збираємося покращити час роботи операцій merge та findMin до O(1) у гіршому випадку.

Припустимо поки, що тип куп поліморфний щодо типу елементів, що для будь-якого типу елементів ми магічним чином знаємо, яку функцію порівняння використовувати. Пізніше ми врахуємо, що як тип елементів, так і функція порівняння на цих елементах задаються в момент застосування функтора.

З урахуванням перерахованих припущень тип розгорнутих куп можна задати як:

```
datatype α Heap = E | H of α ╳ (α Heap) PrimH.Heap
```

Лістинг 1.2.6: Визначення типу елемента купи Бродала Окасакі.

де PrimH - реалізація елементарних куп. Елемент, що зберігається у кожному вузлі H, буде мінімальним елементом піддерева з коренем у цьому вузлі. Елементами елементарних куп будуть служити розгорнуті купи. Усередині елементарних куп розгорнуті купи впорядковані за своїми мінімальними елементами (тобто корінням). Можна думати про цей тип як про тип дерев із змінним ступенем розгалуження, причому діти кожного вузла власними силами зберігаються в елементарних купах.

Оскільки мінімальний елемент зберігається в корені, функція findMin є доволі простою. Щоб купу злити дві розгорнуті купи, ми поміщаємо купу з більшим коренем з меншим коренем як елемент.

```
fun findMin (H (x, _)) = x
fun merge (E, h) = h
  | merge (h, E) = h
  | merge (h₁ as H (x, p₁), H₂ as H (y, p₂)) =
```

```
    if x < y then H (x, PrimH.insert (h2, p1))
    else H (y, H.insert (h1, p2))
```

Лістинг 1.2.7: операцій findMin і merge для купи Бродала Окасакі.

(У виразі x < y ми припускаємо, що функція < — правильна функція порівняння для цих елементів.) Функція insert визначається через merge. Також, одразу розглянемо deleteMin.

```
fun insert (x, h) = merge (H (x, PrimH.empty), h)
fun deleteMin (H (x, p)) = h
    if PrimH.isEmpty p then E
    else let val (H (t, p1)) = PrimH.findMin p
             val p2 = PrimH.deleteMin p
         in H (y, PrimH.merge (p1, p2)) end
```

Лістинг 1.2.8: операцій insert і deleteMin для купи Бродала Окасакі.

Відкинувши корінь, спочатку ми дивимося, чи є елементарна купа p порожньою. Якщо так, то нова купа також порожня. В іншому випадку ми знаходимо і виймаємо мінімальний елемент p, що є розгорнутою купою з мінімальним з усіх елементом; цей елемент стає новим коренем. Нарешті, ми зливаємо p_1 і p_2 і отримуємо нову елементарну купу.

Аналіз цих куп не становить складності. Очевидно, що findMin працює за час O(1) у гіршому випадку незалежно від нижче лежачої реалізації елементарних куп. Функції insert і merge залежить тільки від PrimH.insert. Оскільки ми припускаємо, що час роботи PrimH.insert дорівнює O(1) у гіршому випадку, так само і час роботи insert і merge. Нарешті, deleteMin викликає PrimH.findMin, PrimH.deleteMin та PrimH.merge. Оскільки всі вони працюють за O(log n) у гіршому випадку, така ж і характеристика deleteMin.

У подальшому розділі, автор проводить детальний аналіз теорії типів та функціонального програмування, з метою визначення механізму інтеграції однієї кучі в іншу. Однак, ця тема виходить за рамки основних

питань даного дослідження, тому відповідні описи та аналізи не включаються в цей документ.

Таблиця 1.2.4. Асимптотичні оцінки купи Болдала Окасакі.

Операція	Асимптотична оцінка
INSERT	O(1)
FINDMIN	O(1)
DELETEMIN	O(log n), де n - довжина черги
MERGE	O(1)

Coq Модуль Пріорітетної черги з необхідним інтерфейсом операцій і набором аксіом, що повинні виконуватися, був сформований Ендрю Еппелєм у [VFA]. Ця реалізація є надзвичайно цінною для даного дослідження, так як саме цей модуль реалізовуватиметься за допомогою Купи Бродала Окасакі.

У даному випадку, необхідно зазначити, що в лістингу 1.2.4 використовується модуль `Perm`, розроблений автором у попередніх розділах роботи. Модуль Perm містить визначення функції `Permutation al bl`, яка є булевою функцією. Ця функція приймає два параметри: `al` і `bl`, обидва типу `list nat`. Функція повертає значення `true`, якщо список `al` є перестановкою списку `bl`, та `false` у протилежному випадку. Таке визначення дозволяє перевіряти, чи можна отримати один список з другого шляхом перестановки елементів, що є критично важливим для алгоритмів, які залежать від властивостей перестановок списків.

```
functor Bootstrap (functor MakeH (structure E : ORDERED) : sig
                                                             include HEAP
                                                             sharing Elem = E
                                                           end)
                  (structure E : ORDERED) : HEAP =
struct
  structure Elem = E
  (* recursive structures not supported in SML! *)
  structure rec BootstrappedH =
    struct
      datatype T = Empty | Heap of Elem.T × H.Heap
      fun leq (Heap (x, _), Heap (y, _)) = Elem.leq (x, y)
    end
  and H = MakeH (structure E = BootstrappedH)
  open BootstrappedH   (* expose Empty and Heap constructors *)
  type Heap = BootstrappedH.T
  exception EMPTY
  val empty = Empty
  fun isEmpty Empty = true
    | isEmpty (Heap _) = false
  fun merge (Empty, h) = h
    | merge (h, Empty) = h
    | merge (h1 as Heap (x, p1), h2 as Heap (y, p2)) =
        if Elem.leq (x, y) then Heap (x, H.insert (h2, p1)) else Heap (y, H.insert (h1, p2))
  fun insert (x, h) = merge (Heap (x, H.empty), h)
  fun findMin Empty = raise EMPTY
    | findMin (Heap (x, _)) = x
  fun deleteMin Empty = raise EMPTY
    | deleteMin (Heap (x, p)) =
        if H.isEmpty p then Empty
        else let val (Heap (y, p1)) = H.findMin p
                 val p2 = H.deleteMin p
             in Heap (y, H.merge (p1, p2)) end
end
```

Рисунок 1.2.4. Повна реалізація скошених купи Бродала Окасакі.

```
From VFA Require Import Perm.

Module Type PRIQUEUE.
  Parameter priqueue: Type.
  Definition key := nat.
  Parameter empty: priqueue.
  Parameter insert: key → priqueue → priqueue.
  Parameter delete_max: priqueue → option (key × priqueue).
  Parameter merge: priqueue → priqueue → priqueue.
  Parameter priq: priqueue → Prop.
  Parameter Abs: priqueue → list key → Prop.
  Axiom can_relate: ∀ p, priq p → ∃ al, Abs p al.
  Axiom abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
```

```
  Axiom empty_priq: priq empty.
  Axiom empty_relate: Abs empty nil.
  Axiom insert_priq: ∀ k p, priq p → priq (insert k p).
  Axiom insert_relate:
        ∀ p al k, priq p → Abs p al → Abs (insert k p)
(k::al).
  Axiom delete_max_None_relate:
        ∀ p, priq p → (Abs p nil ↔ delete_max p = None).
  Axiom delete_max_Some_priq:
      ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
  Axiom delete_max_Some_relate:
  ∀ (p q: priqueue) k (pl ql: list key), priq p →
   Abs p pl →
   delete_max p = Some (k,q) →
   Abs q ql →
   Permutation pl (k::ql) ∧ Forall (ge k) ql.
  Axiom merge_priq: ∀ p q, priq p → priq q → priq (merge p q).
  Axiom merge_relate:
    ∀ p q pl ql al,
       priq p → priq q →
       Abs p pl → Abs q ql → Abs (merge p q) al →
       Permutation al (pl++ql).
End PRIQUEUE.
```

Лістинг 1.2.9: Визначення модуля PRIQUE.

Важливо відзначити, що у розглянутому випадку куча реалізована для визначення максимальних елементів, тоді як у попередніх прикладах всі операції були адаптовані для виявлення мінімумів. Демонстрація того, що кучу, орієнтовану на мінімуми, можна ефективно адаптувати для роботи з максимумами шляхом інвертування знаку кожного з її елементів, є досить простою. У подальшому аналізі цього дослідження основна увага буде приділена саме інтерфейсу, що використовує максимальні значення. Це дозволить детальніше оцінити поведінку кучі в контексті операцій на максимумах та її застосування в різних алгоритмічних задачах.

1.3 Постановка задачі

Головною метою цього дослідження є доведення коректності реалізації інтерфейса PRIQUE з використанням купи Бродала Окасакі. Для цього потрібно виконати наступні кроки:

1. Створити інтерфейс PRIQUEWITHINFO, як модифікацію інтерфейса PRIQUE, який дозволить в якості key мати не просто тип nat, а пару nat і елемент довільного типу, яким key був типізований.
2. Реалізувати PRIQUEWITHINFO в якості скошеної біноміальної купи.
3. Довести теореми `can_relate`, `abs_perm`, `empty_priq`, `empty_relate`, `insert_priq`, `insert_relate`, `delete_max_None_relate`, `delete_max_Some_priq`, `delete_max_Some_relate`, `merge_priq`, `merge_relate` для реалізації PRIQUEWITHINFO з пункту 2.
4. Реалізувати PRIQUE в якості купи Бродала Окасакі, яка використовує реалізацію PRIQUEWITHINFO з пункту 2.
5. Довести теореми `can_relate`, `abs_perm`, `empty_priq`, `empty_relate`, `insert_priq`, `insert_relate`, `delete_max_None_relate`, `delete_max_Some_priq`, `delete_max_Some_relate`, `merge_priq`, `merge_relate` для реалізації PRIQUE з пункту 4.

ЧАСТИНА 2. ДОСЛІДЖЕННЯ І АНАЛІЗ РЕЗУЛЬТАТІВ

2.1 Виведення допоміжних результатів

Дослідження, представлене у даній роботі, фокусується на впровадженні та верифікації реалізації пріоритетної черги з використанням структури даних, відомої як куча Бродала-Окасакі. Одним із ключових аспектів успішної реалізації цієї структури даних є створення поліморфного інтерфейсу для пріоритетної черги, який підтримує використання ключів, складених з пари типу `nat` та довільного типу `T`. Такий підхід дозволяє адаптувати інтерфейс до широкого спектру застосувань, де ключі можуть включати додаткову інформацію поряд із числовим ідентифікатором.

Поліморфність інтерфейсу пріоритетної черги є критично важливою для гнучкості і широкої адаптивності структури. Визначення типу ключа як пари `nat` та `T` має значні переваги, оскільки nat забезпечує природний порядок для основних операцій порівняння, необхідних для управління чергою, тоді як `T` може містити додаткові дані, що належать до конкретного використання структури. Це дозволяє відмовитися від необхідності реалізації додаткових інтерфейсів порівняння для типу `T`, спрощуючи розробку та інтеграцію структури.

У контексті кучі Бродала-Окасакі, яка сама по собі є складною структурою, додатковий виклик полягає в тому, що вона розглядає кожен елемент кучі як пару `nat` з іншою кучею, що не має вбудованих операцій порівняння. Рішення полягає у використанні натурального порядку чисел nat для організації основної логіки черги, тоді як вторинні дані в `T` мають лише допоміжне значення. Це унікально вирішує задачу без необхідності зміни основних принципів роботи кучі.

При розробці модульного типу `PRIQWITHINFO` з'явилася необхідність параметризувати цей модуль для роботи з даними різних

типів. Ця потреба виникає з метою забезпечення гнучкості та ширшого застосування модулю, що може ефективно функціонувати в різноманітних програмних контекстах. Рішенням, яке дозволяє ефективно запровадити таку параметризацію, стало введення модульного типу `ElemType`.

```
Module Type ElemType.
  Parameter t : Type.
End ElemType.
```

Лістинг 2.1.1: Визначення модульного типу `ElemType`.

Модульний тип `ElemType` визначає абстракцію для типу даних, що використовується як елементи в кучі. Завдяки цій абстракції, `PRIQWITHINFO` може бути адаптовано для використання з будь-яким типом даних, що значно розширює можливості його застосування.

За визначенням модлуь `PRIQWITHINFO` не відрізняється від `PRIQUE` За винятком параметризації `(E : ElemType)` і визначення ключа як тип `(nat * E.t) % type`. В Coq, вираз `%type` використовується для вказівки конкретного універсуму в контексті поліморфізму типів. Coq має систему з кількома рівнями універсумів (відомих як Set, Prop, і Type), кожен з яких використовується для різних цілей і має різні обмеження.

Коли ви вказуєте `%type` після визначення типу у Coq, ви явно вказуєте, що цей тип повинен розглядатися у контексті універсуму `Type`. Це може бути корисним для вирішення неоднозначностей, особливо в більш складних системах типів, де типи можуть інтерпретуватися по-різному в залежності від контексту.

```
Module Type PRIQWITHINFO (E : ElemType).
   Definition key := (nat * E.t) % type. (* Визначення
ключа як пари *)
  (* Основні типи і функції *)
  Parameter priqueue: Type.
  Parameter empty : priqueue.
  Parameter insert : key -> priqueue -> priqueue.
```

```
    Parameter delete_min : priqueue -> option (key *
priqueue).
  Parameter find_min : priqueue -> option key.
  Parameter merge : priqueue -> priqueue -> priqueue.
  Parameter priq : priqueue -> Prop.
  Parameter Abs : priqueue -> list key -> Prop.

  (* Аксіоми, що визначають поведінку структури *)
  Axiom can_relate : forall p, priq p -> exists al, Abs p
al.
  Axiom abs_perm : forall p al bl,
    priq p -> Abs p al -> Abs p bl -> Permutation al bl.
  Axiom empty_priq : priq empty.
  Axiom empty_relate : Abs empty [].
   Axiom insert_priq : forall k p, priq p -> priq (insert
k p).
  Axiom insert_relate :
       forall p al k, priq p -> Abs p al -> Abs (insert k
p) (k::al).
  Axiom delete_min_None_relate :
        forall p, priq p -> (Abs p [] <-> delete_min p =
None).
  Axiom delete_min_Some_priq :
       forall p q k, priq p -> delete_min p = Some (k,q)
-> priq q.
  Axiom merge_priq : forall p q, priq p -> priq q -> priq
(merge p q).
  Axiom merge_relate :
      forall p q pl ql al,
        priq p -> priq q -> Abs p pl -> Abs q ql -> Abs
(merge p q) al ->
      Permutation al (pl ++ ql).
End PRIQWITHINFO.
```

Лістинг 2.1.2: Визначення модульного типу `PRIQWITHINFO`.

Далі необхідно надати визначення модулю `SkewBinomialHeap`, який є реалізацією модульного типу `PRIQWITHINFO`, як це було запропоновано в пункті 2 розділу 1.3.

За своєю сутністю - ця куча є списком скошених біноміальних дерев, з декількома обмеженнями, переліченими в 1.2. Тому першим необхідним визначенням є індуктивний тип `Tree`, з одним конструктором `Node`, що міститеме ранг вузла типу `nat`, елемент кучі типу `key`, список скошених елементів `list key` і список вузлів нащадків типу `list key`.

```
Module SkewBinomialHeap (E : ElemType) <: PRIQWITHINFO E.

Inductive Tree : Type :=
  | Node : nat -> key -> list key -> list Tree -> Tree.
```

Лістинг 2.1.2: Визначення індуктивного типу `Tree`.

Індуктивні типи у мові Coq відіграють фундаментальну роль в моделюванні математичних структур та алгоритмів, забезпечуючи механізм для визначення складних даних та їх поведінки через самовизначення. Ці типи дозволяють виразити не тільки прості списки або послідовності, а й складніші ієрархічні структури, такі як дерева, графи, та багато інших форм рекурсивних структур. Основна особливість індуктивних типів полягає в тому, що вони визначаються через власні конструктори, які дозволяють побудову значень цього типу з інших значень того ж або інших типів, включаючи самі себе.

У контексті формальної верифікації програмного забезпечення, індуктивні типи дозволяють розробникам сформулювати властивості та гарантії про структури даних та алгоритми, що обробляють ці дані, через строгі математичні докази. Це важливо для гарантування коректності, безпеки, та ефективності програмних систем. Так, наприклад, за допомогою індуктивних типів можна формалізувати концепції, такі як натуральні числа (через конструктори для нуля та наступника числа),

булеві вирази, або навіть кроки виконання складного алгоритму, таким чином влаштовуючи глибокий аналіз та перевірку програмних компонентів на рівні типів даних.

Подальщі визначення операцій для скошеної біноміальної кучі будуть наведені у додатку. Для визначення операції здебільшого використовувались ключові слова `Definition` і `Fixpoint`.

`Definition` використовується для створення статичних визначень або присвоєнь. Це можуть бути константи, визначення типів даних або навіть цілісні структури. Ідея полягає в тому, що `Definition` забезпечує спосіб надання імені певному значенню або виразу. Визначення, створені за допомогою `Definition`, не містять рекурсивних виразів та зазвичай використовуються для зазначення точних, незмінних значень або властивостей. Наприклад, визначення конкретного числа або конфігурації, що не змінюється під час виконання програми. Вони є засобом абстрагування та узагальнення даних у Coq.

`Fixpoint`, з іншого боку, використовується для визначення рекурсивних функцій, що є фундаментальними в функціональних мовах програмування, таких як Coq. Рекурсивні функції потребують механізму для виходу з рекурсії, який зазвичай реалізується через базові випадки. Fixpoint дозволяє визначати такі функції, що можуть викликати самі себе з модифікованими аргументами. Цей механізм є ключовим для вирішення задач, які природньо структуровані через рекурсію, наприклад, обчислення факторіалу або перебір елементів у списку. Використання `Fixpoint` вимагає, щоб рекурсивний виклик був структурно рекурсивним, тобто кожний рекурсивний виклик повинен працювати з "меншим" або "простішим" аргументом, що гарантує зупинку рекурсії.

Проте, є 1 функція, яка не виходить визначити тільки з використанням `Fixpoint`. Це фукнція `merge_trees`. При використанні `Fixpoint` в цьому випадку видається помилка про те, що не вдається визначити аргумент рекурсії, що зменшується, тобто є підозра на те, що це

може бути нескінченний цикл. Запобігти цього можно за допомогою ключового слова `Program Fixpoint` і вказанням як визначається глибина рекурсивного виклику.

```
From Coq Require Import Program.

Program Fixpoint merge_trees (ts1 ts2: priqueue) {measure
(length ts1 + length ts2)}: priqueue :=
  match ts1, ts2 with
  | _, nil           => ts1
  | nil, _           => ts2
  | t1::tl1, t2::tl2 => if rank t1 <? rank t2
    then t1::(merge_trees tl1 ts2)
    else if rank t2 <? rank t1
         then t2::(merge_trees ts1 tl2)
         else ins_tree (link t1 t2) (merge_trees tl1 tl2)
  end.
Next Obligation.
  simpl. lia.
Defined.
Next Obligation.
  simpl. lia.
Defined.
```

Лістинг 2.1.3: Визначення функцій `merge_trees` і `remove_min_tree`.

У мові програмування Coq, `Program Fixpoint` розширює стандартний `Fixpoint` засіб, надаючи можливість визначати рекурсивні функції з неповними специфікаціями і автоматично генерувати обов'язки для доведення коректності таких визначень. Це полегшує розробку, дозволяючи розробникам відкласти складні докази термінування та інші інваріанти, які можуть бути доведені пізніше, зосереджуючись спочатку на основній логіці програми. Цей інструмент також автоматизує управління доказами та знижує обсяг ручної роботи, необхідної для забезпечення правильності рекурсивних визначень, що робить процес розробки більш гнучким і ефективним.

Також були запропоновані визначення функцій `Abs` і `priq` з модульного типу `PRIQWITHINFO`. Функція `Abs` є предикатом, що приймає пріоритетну чергу і список елементів типу `list key`. Цей предикат використовується для перевірки, чи може черга бути коректно побудована з поданого списку елементів. Логічно, якщо кількість елементів у списку не відповідає кількості елементів у черзі, визначено, що дана куча не може бути правильно сформована з цього списку, що допомагає в обробці винятків та помилок у структурі даних

Для реалізації предиката `Abs`, використовується методика рекурсивного порівняння, яка передбачає видалення мінімальних елементів з черги та списку відповідно, щоб забезпечити їх однаковість. Видалення мінімального елемента в черзі вже підтримується через існуючу функцію `delete_min`, в той час як для списку потрібно було визначити додаткову функцію `delete_min_list`. Ця функція в функціональному програмуванні працює у два етапи: спершу відбувається пошук мінімального елемента в списку за допомогою функції `list_min`, а потім відбувається видалення першого входження цього елемента за допомогою функції `remove_first`. Ці функції разом формують механізм, що дозволяє підтримувати інтегритет структури даних та її відповідність заданому списку, забезпечуючи гнучке управління даними у складних програмних системах, що вимагають високої точності та надійності обробки.

```
Fixpoint list_min (l : list key) : option key :=
  match l with
  | [] => None
  | [x] => Some x
  | x :: xs =>
    match list_min xs with
    | Some m => Some (min_key x m)
    | None => Some x
    end
```

```
  end.

Fixpoint remove_first (x : key) (l : list key) : list key
:=
  match l with
  | [] => []
  | h :: t => if fst h =? fst x then t else h ::
remove_first x t
  end.

Definition delete_min_list (l : list key) : option (key *
list key) :=
  match list_min l with
  | Some m => Some (m, remove_first m l)
  | None => None
  end.
```

Лістинг 2.1.4: Визначення функції `delete_min_lsit`.

Функція `Abs` є рекурсивною функцією, з визначеною базою рекурсії, а також основним визначенням, в якому якщо мінімальний елемент кучі дорівнює мінімальному елементу списку, а також якщо даний предикат є вірним для залишкового списку і кучі, то твердження є дійсним. У інших випадках твердження є хибним. Така рекурсивна структура дозволяє методично перевіряти відповідність між кучею та списком, забезпечуючи точну верифікацію їх елементів. Основний виклик при реалізації такої функції полягає у забезпеченні, що рекурсія завершується належним чином, з гарантією, що кожен рекурсивний крок зменшує розмір проблеми, що запобігає можливості зациклення або переповнення стеку.

Необхідно зазначити, що для `Abs` також треба використовувати `Program Fixpoint`. Глибина рекурсії буде не глибшою за кількість елементів у списку. Технічно ця глибина не більша за мінімум між довжиною списку і кількістю елементів в черзі, бо на кожен рекурсивний виклик передаються обидві структури з видаленим елементом, і зупинка

відбувається у випадку, коли хоча б одна зі структур стала порожньою. Використання `Program Fixpoint` допомагає формалізувати цей процес, забезпечуючи необхідні докази для гарантії коректності та завершення рекурсії, що особливо важливо в контексті формальної верифікації програмного забезпечення. Проте доведення такого факту виявляється доволі складним для даного дослідження, тому воно залишається для подальших робіт, що вимагатиме додаткових ресурсів і спеціалізованих методик.

```
Program Fixpoint Abs (q : priqueue) (lk : list key)
{measure (length lk)} : Prop :=
  let pq := delete_min q in
  let pl := delete_min_list lk in
    match pq, pl with
    | None, None => True
    | None, _ => False
    | _, None => False
     | Some (mq, pqs), Some (ml, pls) => fst mq = fst ml
/\ Abs pqs pls
    end.
Next Obligation.
Admitted.
Next Obligation.
Admitted.
```

Лістинг 2.1.5: Визначення функції `delete_min_lsit`.

Предикат `priq` відіграє критичну роль у верифікації структури пріоритетних черг у контексті кваліфікаційної роботи. Він приймає пріоритетну чергу як вхід і перевіряє її на відповідність ряду заздалегідь визначених правил, що стосуються структури черги. "Правильність" структури в даному контексті означає, що кожне дерево в списку дерев черги повинне задовольняти критерії, які забезпечують ефективне функціонування черги:

1. Ранг кожного наступного дерева в списці дерев повинен бути більшим за попереднє.
2. Виключенням до правила 1 можуть бути перші 2 дерева. Вони можуть мати однакові ранги.
3. В кожному дереві в кожному вузлі усі ранги нащадків менші за ранг вузла.
4. В кожному дереві в кожному вузлі усі значення ключа нащадків менші за значення ключа вузла. Порівняння відбувається за першим елементом пари.
5. Усі значення залишкового списку в вузлі є меншими або дорівнюють значенню ключа в вузлі.

Для визначення пунктів 3-5 були декларовані функції `tree_size`, `is_heap_tree'` та `is_heap_tree`. Пункт 1 був визначений за допомогою функції `is_heap_tail`. І пункт 2 був визначений безпосередньо в функції `priq`.

```
Fixpoint tree_size (t : Tree) : nat :=
  match t with
  | Node _ _ _ children =>
      1 + fold_right (fun child acc => tree_size child + acc) 0 children
  end.

Program Fixpoint is_heap_tree' (t : Tree) (pr : nat) {measure (tree_size t)}: Prop :=
  match t with
  | Node r (x, _) xs children =>
      (r < pr) /\
      (Forall (fun child =>
        match child with
          | Node _ (y, _) _ _ => is_heap_tree' child r /\ (x <= y)
        end) children) /\
      (Forall (fun z => x <= fst z) xs)
```

```
      end.
    Next Obligation.
    Admitted.

    (* Root function to initiate the recursive check *)
    Definition is_heap_tree (t : Tree) : Prop :=
      match t with
      | Node r (x, _) xs children =>
          (Forall (fun child =>
            match child with
             | Node _ (y, _) _ _ => (x <= y) /\ is_heap_tree'
child r
            end) children) /\
          (Forall (fun z => x <= fst_key z) xs)
      end.


    (* Additional check for rank decrease in heap *)
    Fixpoint is_heap_tail (ts: list Tree) (pr: nat): Prop :=
      match ts with
        | nil    => True
          | t::ts1 => is_heap_tree t /\ (pr < (rank t)) /\
is_heap_tail ts1 (rank t)
      end.


    Definition priq (q: priqueue) : Prop :=
      match q with
      | nil        => True
      | t::nil     => is_heap_tree t
           |  t1::t2::ts  =>  (rank  t1)  <=  (rank  t2)  /\
is_heap_tree(t1) /\ is_heap_tree(t2) /\ is_heap_tail ts (rank
t2)
      end.
```

Лістинг 2.1.6: Визначення функції `priq`.

Таким чином були надані визначення структури, усіх операції, а також предикатів з модульного типу PRIQWITHINFO. Далі необхідно вивести доведення теорем з PRIQWITHINFO, визначити модуль кучі Бродала-Окасакі, а також довести теореми для неї. Цей етап включатиме формалізацію доказів для кожної теореми, що гарантує, що реалізовані структури даних та алгоритми відповідають усім вимогам і працюють згідно з визначеннями.

2.2 Доведення коректності основного предмету дослідження

Для коректного доведення кучі Бродала Окасакі необхідно довести теореми з модуля PRIQWITHINFO при реалізації зі скошеною Біноміальною кучею, написати реалізацію самовкладенної кучі, а також доведення для неї.

Зі списку теорем на доведення є різні за складністю на доведення. Зазвичай - ті, що націлені на базові випадки з порожніми чергами є найлегшими для доведення, так як не мають рекурсивних визначень. Тому теореми `empty_priq`, `empty_relate`, `delete_min_None_relate` були доведені першими.

Для доведення `empty_priq` були використані 2 тактики системи Coq - це `simpl` і `trivial`. Тактика `simpl` дозволяє спростити вираз, коли в базі знань є достатньо гіпотез, або коли сам вираз автоматично спрощується. Так як priq визначена як співставлення по шаблону списка, а empty - це порожній список, можна спростити цей вираз до терму, що написаний для випадку `nil` в `priq`, тобто `True`. Це, в свою чергу, вже є саме по собі завжди правильне твердження, тому застосовується тактика `trivial`, аби показати що доказ закінчений.

```
Theorem empty_priq : priq empty.
Proof.
```

```
  simpl.
  trivial.
Qed.
```

Лістинг 2.2.1: Доведення `empty_priq` для скошеної біноміальної кучі.

Для доведення `empty_relate` тактика simpl не працює одразу, для цього необхідно спочатку явно зазначити як виглядає визначення `Abs`, тому для цього використовується тактика `unfold Abs`. Так як `delete_min` і `delete_min_list` повертають `None` в випадку порожніх структур - вираз також спрощується до `True` в цьому випадку.

```
Theorem empty_relate : Abs empty [].
Proof.
  unfold Abs.
  simpl.
  trivial.
Qed.
```

Лістинг 2.2.2: Доведення `empty_relate` для скошеної біноміальної кучі.

Доведення `delete_min_None_relate` вже є складнішим завданням, так як рівносильність повинна доводитись спочатку в одну, а потім в іншу сторону як імплікація. Для цього використовується тактика split. У випадку `Abs p [] -> delete_min p = None` треба разбирати 2 випадки значення `remove_min_tree p` з визначення `delete_min` за допомогою тактики `destruct`. У випадку `None` - показуємо рефлексивність визначення `None=None`. У випадку повернення `Some` - треба показати, що в наведених гіпотезах є невірні припущення. Тоді буде працювати логіка висказувань, де `False -> False = True`, бо, очевидно `Some` і `None` це різні конструктори, які не можуть бути рівними. Для цього треба спростити припущення з `Abs` до випадку, коли або `p=[]` і це `True`, або `False` в іншому випадку. В першому випадку треба

показати, що в припущеннях `Some=None`, що є хибним твердженням. Або треба показати, що в припущеннях є `False`.

В зворотному напрямку відбувається все те саме, але тепер з `delete_min` перетворення відбуваються в гіпотезах, а в `Abs` треба робити більше розкриття визначень, аби дійти до базових випадків, де можна показати або протиріччя, або тривіальність.

```
Theorem delete_min_None_relate :
  forall p, priq p -> (Abs p [] <-> delete_min p = None).
Proof.
  intros p Hp.
  split.
  - intro H.
    unfold delete_min.
    destruct (remove_min_tree p) eqn:Hr.
      + unfold Abs in H. unfold Abs' in H. simpl in H.
destruct p.
        * simpl in Hr. discriminate Hr.
        * exfalso. exact H.
      + reflexivity.
  - intro H.
    unfold delete_min in H.
    destruct (remove_min_tree p) eqn:Hr.
    + destruct p0. destruct t. discriminate H.
    + unfold Abs. unfold Abs'. simpl. destruct p eqn:Hep.
      * trivial.
      * simpl in Hr. destruct p0.
        -- discriminate Hr.
              -- destruct (remove_min_tree (t0 :: p0))
eqn:Hrmt.
          ++ destruct p1; destruct t; destruct t1;
             destruct (root (Node n k l l0));
             destruct (root (Node n0 k0 l1 l2));
             destruct (n2 <? n1); discriminate Hr.
          ++ discriminate Hr.
```

```
Qed.
```

Лістинг 2.2.3: Доведення `delete_min_None_relate` для скошеної біноміальної кучі.

Двома фундаментальними теоремами в випадку пріоритетної черги є теореми `insert_priq` і `delete_min_Some_priq`. Ці теореми показують чи зберігає структура всі обмеження після додавання, або видалення елементів з неї.

Для доведення `insert_priq` необхідно довести 3 допоміжні лемми, що дуже допоможуть в подальшому доведенні. Першою Леммою є `le_S_leq`, яка визначена як `forall a b : nat, a < b -> S a <= b`. Тобто якщо a менше за `b`, тоді successor a менший, або дорівнює за `b`. Ця лемма доводиться дуже просто з використанням `Nat.succ_lt_mono` (Дод. А).

Другою Леммою є `skew_link_rank_increment`, яка визначена як `forall k t1 t2, is_heap_tree t1 -> is_heap_tree t2 -> rank (skew_link k t1 t2) = S (rank t1)`. Тобто операція лемма `skew_link` показує, що ранг отриманого дерева на 1 більший за ранг першого переданого дерева. Тут варто зазначити, що операція `skew_link` використовується тільки у випадках, коли ранги дерев є рівними, у інших випадках її використання є недоцільним. Очевидно, що операція `skew_link` використовує, в свою чергу, операцію `link`, яка в будь-якому випадку повертає дерево рангом `S (rank t1)`. Тому доведення спрямоване на простий розбір випадків, де в кінцевому результаті показується ця рівність.

Лема `is_heap_tree_skew_link` стверджує, що якщо обидва дерева `t1` та `t2` є кучами, тобто задовольняють властивість `is_heap_tree`, тоді дерево, яке утворюється в результаті виконання функції `skew_link` з ключем `k` для `t1` та `t2`, також буде задовольняти цю властивість. Доведення цієї леми передбачає показ того, що

властивості, забезпечувані `is_heap_tree`, зокрема порядок ключів та структурні особливості дерев, зберігаються після виконання операцій в `skew_link`. Ключовим є переконання у тому, що мінімальний елемент завжди залишається у корені результуючого дерева після виконання `skew_link`, що забезпечується завдяки умовному розміщенню `k` або оригінального кореня в залежності від їх порівняння. Основними кроками в доведенні буде аналіз структури та порядку ключів перед та після виконання `skew_link`, використання індукції за структурою `t1` та `t2` і демонстрація того, що зміни, внесені у `skew_link` (додавання ключа до списку ключів дітей та потенційне перегрупування дитячих вузлів на основі порівняння ключів), не порушують умов кучі.

Доведення теореми є доволі складним і громіздким, в першу чергу через визначення `priq`. Кожен раз, коли робиться розгортка функції `priq` - необхідно доводити вираз, який в свою чергу є кон'юнкцією з чотирьох інших виразів, кожен з яких треба довести окремо. Основне доведення ділиться на 2 основні гілки - порожня черга і не порожня. Для порожньої черги випадок є тривіальним, а для не порожньої йде розбір за випадками чи перші 2 дерева є рівними за рангами, чи ні. У випадку з рівністю в процесі розбору застосовуються в ті три леми що були визначені попередньо.

Теорема `delete_min_Some_priq` стверджує, що якщо деяка пріоритетна черга p задовольняє властивість кучі, і в результаті виклику функції `delete_min` на p отримуємо пару `(k, q)`, де `k` - це мінімальний ключ, а `q` - нова пріоритетна черга після видалення мінімального ключа, тоді `q` також повинна задовольняти властивість кучі. Для доведення цієї теореми слід показати, що операція `delete_min` коректно модифікує структуру дерева, зберігаючи необхідний порядок серед елементів. Основні кроки доведення включають аналіз того, як `delete_min` обробляє різні випадки структури дерева, зокрема

збереження глобального порядку ключів після видалення мінімуму. Важливим є показ того, що всі субдерева і залишені елементи після видалення мінімуму досі задовольняють умови `priq`, зокрема, підтримка властивостей кучі для кожного субдерева та відповідних рангів серед них, що вимагає індуктивного аналізу структури дерев та застосування властивостей операцій insert та merge, використаних у процесі видалення.

Для доведення теореми `delete_min_Some_priq` можна використати індуктивний аналіз структури `p`, продемонструвати, що видалення мінімального елемента через функцію `delete_min` коректно перерозподіляє елементи та піддерева, зберігаючи умови кучі для всіх субдерев та забезпечуючи, що отримана черга `q` відповідає умовам властивості `priq`, а також включаючи аналіз поведінки `insert_all` та `merge` функцій, які застосовуються під час видалення мінімального елемента, для гарантування, що зміни у структурі не порушують інваріанти кучі. Нажаль, на даний момент довести цю теорему не вдалося через специфіку індуктивних доведень в Coq, але план доведення є визначений і може бути реалізованим в подальших дослідженнях.

За тією самою причиною і не вдалося довести теореми `merge_priq`, `insert_relate` та `can_relate`, так як ці теореми є більш складними в плані індуктивних доведень. Проте, були складені загальні плани доведення цих теорем, які описують шлях їх доведення.

Для доведення теореми merge_priq потрібно показати, що об'єднання двох черг, кожна з яких задовольняє властивість кучі, через функцію `merge` зберігає інваріанти кучі у результуючій черзі. Спершу, потрібно перевірити коректність операцій `normalize` та `merge_trees`, що використовуються у `merge`, та впевнитися, що вони зберігають впорядкованість та структурні властивості, такі як ранги дерев. Далі, необхідно демонструвати, що будь-яка комбінація дерев з `p` і `q` утворює правильну кучу через індукцію по структурі дерев.

В основі доведення теореми `insert_relate` лежить показ зв'язку між операцією вставки та абстракцією черги в список. При виконанні `insert`, необхідно продемонструвати, що додавання елемента `k` до черги `p`, яка задовольняє `priq` та відповідає списку `al` через властивість `Abs`, результує в черзі, що відповідає додаванню `k` на початок `al`. Цей процес включає аналіз модифікацій, які `insert` вносить у структуру черги, та відстеження цих змін в абстракції списку.

Теорема `can_relate` вимагає показати, що для будь-якої черги `p`, яка задовольняє властивість `priq`, існує список `al`, такий що `Abs p al` відповідає правді. План доведення містить конструкцію такого `al` шляхом ітерації через процес видалення мінімальних елементів з `p` та їхнього накопичення у список, перевіряючи при цьому, що кожен крок відповідає абстракції. Цей підхід демонструє, як фізична структура черги може бути відтворена або представлена як список елементів, що дає змогу утвердити відповідність між структурною реалізацією та її абстрактним представленням.

Теореми `delete_min_Some_relate`, `abs_perm`, `merge_relate` використовують визначення `Permutation` з [2], тому їх доведення потребує глибшого розуміння цієї роботи. Доведення цих теорем залишається для подальших досліджень.

Провівши дослідження в області імплементації та доведення коректності пріоритетних черг на базі скошеної біноміальної кучі тепер можна приступити до імлементації та доведення коректності модуля `PRIQUE` на базі кучі Бродала Окасакі.

Одним з найбільших випробувань цієї роботи була саме імплементація типу BPQ, який використовує модуль `SkewBinomialHeap`, типізований самим собою. Нажаль вивести правильну імплементацію цього типу не вдалося, бо Coq має свої обмеження в цьому контексті. Тому наразі буде використовуватись

імплементація з 2.2.4, як схематичне описання як такий індуктивний тип може виглядати.

```
Inductive BPQ : Type :=
| bpq : SkewBinomialHeap(BPQ).prique -> BPQ.
```

Лістинг 2.2.4: Схематичне визначення індуктивного типу `BPQ`.

Імплементація усіх операцій пріоритетної черги, визначених в [8] відрізняється тільки за синтаксисом, тому їх імплементація не придає інтересу в контексті дослідження. Імплементація всіх необхідних операцій буде наведена у додатку. Так само і `Abs`, визначений абсолютно ідентично до визначення з `PRIQWITHINFO`, тому в даному розділі увага йому приділятись не буде.

Визначення `priq` доволі простим як з точки зору семантичного визначення, так і з точки зору коду. Необхідно показати, що отримана структура типу `priq` з цього модуля задовольняє таким критеріям:

- Перший елемент в парі - це мінімальний елемент в кучі
- Другий елемент в парі - задовольняє критерію `priq` з модуля `PRIQWITHINFO`

Перевірити першу умову можна застосувавши `delete_min` до переданої черги, і перевіривши, що значення співпадають. Друга умова перевіряється застосуванням `priq` з `PRIQWITHINFO` до другого елемента пари.

```
Definition priq: (p: priqueue): Prop :=
  match p with
  | None => True
  | Some (x, bpq q) => let pr := delete_min p in (
      match pr with
        | None => True
        |  Some  (x1,  p2)  =>  x1  =  x  /\
SkewBinomialHeap(BPQ).priq q
      end
    )
```

```
end.
```

Лістинг 2.2.5: Визначення `priq` для Кучі Бродала-Окасакі.

Теорема can_relate з модуля `BootstrapQueue` заснована на основній ідеї, що кожна приоритетна черга, яка задовольняє властивість priq, може бути асоційована з деяким списком ключів таким чином, що цей список відповідає структурі черги згідно з властивістю `Abs`. Доведення цієї теореми вимагає показати, що для будь-якого стану пріоритетної черги, яка задовольняє `priq`, можна знайти список ключів, який відповідає структурі цієї черги згідно з визначенням `Abs`. Це включає використання властивостей функцій `delete_min` та інших операцій, що дозволяють індуктивно побудувати або підтвердити цей список ключів.

```
Theorem can_relate: forall p, priq p -> exists al, Abs p
al.
Proof.
  intros p H.
  induction p using priqueue_ind.
  - exists []. simpl. auto.
  - destruct p as [n bpq].
    unfold priq in H. unfold delete_min in H.
         destruct  (SkewBinomialHeap(BPQ).delete_min  bpq)
eqn:Hmin.
    + destruct p as [k q]. simpl in H.
       assert (Hq: SkewBinomialHeap(BPQ).priq q) by (apply
SkewBinomialHeap(BPQ).delete_min_Some_priq  with  (k:=k);
assumption).
      specialize (IHp q Hq).
      destruct IHp as [al Hal].
      exists (n::al).
      simpl. unfold Abs. simpl.
            rewrite  Hmin.  simpl.  split;  [reflexivity|
assumption].
```

```
      + simpl in H. destruct bpq; simpl in Hmin; try
discriminate.
      exists [n]. simpl. unfold Abs. rewrite Hmin. simpl.
auto.
Qed.
```

Лістинг 2.2.6: Потенційне доведення `can_relate` для модуля `BootstrapQueue`

У цьому доведенні використовується індукція по структурі `priqueue` та розглядаються випадки, коли черга порожня та коли вона містить елементи. У випадку порожньої черги, показується, що асоційований список також порожній. Для непорожньої черги, розглядається результат операції `delete_min` і використовується індукційне припущення для решти черги після видалення мінімуму, що дозволяє нам рекурсивно побудувати відповідний список ключів.

Для доведення `empty_priq` і `empty_relate`, використовуються визначення `priq` і `Abs` для порожньої черги, що зазначено в модулі `SkewBinomialHeap` як `empty` та порожній список `[]` відповідно. Виклик `unfold` в Coq розгортає ці визначення, а `trivial` використовується для автоматичного доведення простих випадків, де порожня структура безпосередньо задовольняє визначені властивості за допомогою початкового визначення цих властивостей в Coq.

```
Theorem empty_priq : priq empty.
Proof.
  unfold priq, empty.
  simpl. trivial.
Qed.

Theorem empty_relate : Abs empty [].
Proof.
  unfold Abs, empty.
  simpl. trivial.
Qed.
```

Лістинг 2.2.7: Потенційні доведення `empty_priq`, `empty_relate` для модуля `BootstrapQueue`

Для доведення теореми `insert_priq` необхідно розкласти `insert` з подальшим розкладом `merge`. Довівши тривіальні випадки - необхідно довести випадок, в якому вставляється мінімальний елемент, а також випадок, в якому вставляється не мінімальний елемент. В обох випадках можна застосувати `insert_priq` з `PRIQWITHINFO`. Саме доведення є доволі складним для написання, тому це дослідження обмежується лише описанням як воно може виглядати.

Для доведення теореми `insert_relate`, спочатку необхідно розглянути дію вставки елемента `k` в пріоритетну чергу `p`, яка вже асоційована зі списком `al` через властивість `Abs`. Основна мета полягає у підтвердженні, що після вставки ключа `k` в чергу, результуючу чергу можна асоціювати зі списком `k::al`, зберігаючи абстрактну відповідність між структурою даних та її абстрактним представленням. Теорема вимагає аналізу внутрішньої структури функції `insert`, переконуючись, що новий ключ `k` правильно інтегрується на початку списку ключів, що відповідає інтуїтивному розумінню черг з пріоритетами, де новий елемент додається до існуючої структури.

Теорема `merge_priq`, з іншого боку, зосереджується на доведенні того, що злиття двох приоритетних черг `p` та `q`, кожна з яких відповідає властивості `priq`, результує чергою, яка також відповідає цій властивості. Злиття, як операція, повинна забезпечити, що всі елементи обох черг впорядковано інтегровані в одну результуючу структуру без порушення інваріантів черги. Аналіз цієї операції включає розгляд способу, яким внутрішні структури `p` та `q` об'єднуються, гарантуючи, що жоден інваріант пріоритетної черги не порушується. Для цього може бути використано індуктивний підхід для аналізу властивостей priq на кожному кроці об'єднання.

Для теореми `delete_min_None_relate`, важливо продемонструвати, що відношення відсутності мінімального елемента в приоритетній черзі `p` (тобто, коли функція `delete_min` повертає `None`) еквівалентно тому, що черга асоційована з порожнім списком ключів. Ця теорема забезпечує взаємний зв'язок між абстрактним представленням черги як порожнього списку та конкретною реалізацією, що підтверджує порожнечу черги. Доведення цієї теореми передбачає розгляд умови, за якої функція `delete_min` на фактично порожній черзі повертає `None`, і зворотне, перевіряючи, що не існує жодного елемента в черзі, що може бути видалений.

Теорема `delete_min_Some_priq` зосереджується на верифікації властивостей пріоритетної черги після видалення мінімального елемента. Це передбачає, що якщо з пріоритетної черги `p`, яка задовольняє властивість priq, видаляється мінімальний елемент, результатом є нова черга `q` і мінімальний ключ `k`, то `q` також повинна відповідати властивості `priq`. Доведення цієї теореми включає аналіз механізму функції `delete_min`, який гарантує, що всі властивості впорядкованості та коректності структури даних зберігаються навіть після видалення елемента, що є критично для забезпечення надійності черги в динамічних операціях.

Інші теореми не доводяться за такою самою причиною, за якою вони не доводились для Скошеної біноміальної кучі.

2.3 Аналіз результатів

В якості оцінки результатів будуть використовуватись таблиці зі звітом щодо результатів, яких вдалося досягти в ході цього дослідження. Для оцінки ступеня досягнення з приводу того, чи іншого елементу використовуватимуться такі оцінки:

1. Реалізовано - Стосується функцій, реалізацію яких треба було перевести на мову Coq. Відноситься до пунктів 2 та 4 з постановки задачі. Означає, що реалізація повністю виконана.
2. Схема - Стосується функцій, реалізацію яких треба було перевести на мову Coq. Відноситься до пунктів 2 та 4 з постановки задачі. Означає, що реалізація не виконана, але схематично описана.
3. Доведено - Стосується теорем, які необхідно було довести з пунктів 3 та 5. Означає, що теорема повністю була доведена.
4. Частково - Стосується теорем, які необхідно було довести з пунктів 3 та 5. Означає, що теорема була доведена, але має не доведені лемми, які використовуються в доведенні.
5. План - Стосується теорем, які необхідно було довести з пунктів 3 та 5. Означає, що теорема не була доведена, але має план доведення, який може потім бути переведений в саме доведення
6. Не доведено - Стосується теорем, які необхідно було довести з пунктів 3 та 5. Означає, що теорема повністю була не доведена.

Пункт 1 з постановки задачі був повністю виконаним, і сам не підлягає під звіт у вигляді таблиці, тому тут будуть викладені таблиці тільки за пунктами 2-5.

Параметр	**SkewBinomialHeap**	**BootstrapQueue**
prique	Реалізовано	Схема
empty	Реалізовано	Реалізовано
insert	Реалізовано	Реалізовано
delete_min	Реалізовано	Реалізовано
find_min	Реалізовано	Реалізовано
merge	Реалізовано	Реалізовано
priq	Реалізовано	Реалізовано

Abs	Реалізовано	Реалізовано

Таблиця 2.3.1: Звіт по результатам пунктів 2 та 4.

З цього звіту видно, що Пункт 2 був повністю виконаний. Пункт 4 майже повністю був реалізований за виключенням prique, що може бути зроблено в подальших дослідженнях.

Теорема	**SkewBinomialHeap**	**BootstrapQueue**
can_relate	Схема	Схема
abs_perm	Не доведено	Не доведено
empty_priq	Доведено	Схема
empty_relate	Доведено	Схема
insert_priq	Частково	Схема
insert_relate	Схема	Схема
delete_min_None_relate	Доведено	Схема
delete_min_Some_priq	Схема	Схема
merge_priq	Схема	Схема
merge_relate	Не доведено	Не доведено
delete_min_Some_relate	Не доведено	Не доведено

Таблиця 2.3.2: Звіт по результатам пунктів 3 та 5.

В пункті 2 є різноманіття за результатами, де є як доведені теореми, так і ті, які тільки описані з точки зору ідеї доведення. Але в пункті 5 немає жодної доведеної теореми, через те що в пункті 4 реалізація не була повністю закінчена. Тому дуже важливо мати повну реалізацію перед приступанням до доведень безпосередньо.

Проте, запропоновані плани (або схеми) доведень можуть бути використані в майбутніх роботах, коли реалізація з пункту 4 буде повністю закінчена.

ВИСНОВОК

Це дослідження зосереджене на формальному доведенні властивостей пріоритетної черги з використанням структури даних Кучі Бродала-Окасакі принесло важливі результати, які можуть мати значний вплив на подальші розробки в області чисто функціональних структур даних та їх застосування.

Оскільки Куча Бродала-Окасакі є чисто функціональною структурою, її успішне формальне доведення демонструє потенціал функціональних методів в областях, де важлива надійність та безпека. Це доведення відкриває двері для ширшого застосування чисто функціональних структур у комерційних та промислових застосуваннях.

Докази, отримані в ході дослідження, можуть слугувати основою для розробки нових версій програмного забезпечення, що використовують пріоритетні черги, з гарантіями щодо їх продуктивності та надійності. Розробники можуть використовувати ці результати для оптимізації алгоритмів, що залежать від швидких і надійних операцій з даними.

Це дослідження відкриває можливості для подальших досліджень в області оптимізації чисто функціональних структур даних. Можливо розглядати варіації Кучі Бродала-Окасакі або розробляти нові структури, які можуть пропонувати ще кращі характеристики за певних умов використання.

Підсумовуючи, результати цього дослідження не тільки підтверджують важливість Кучі Бродала-Окасакі як чисто функціональної структури даних у реалізації пріоритетних черг, але й підкреслюють значення формальної верифікації у розробці надійного програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aho A. V. The design and analysis of computer algorithms. Reading, Mass : Addison-Wesley Pub. Co., 1974. 470 p.
2. Appel A. W. Verified functional algorithms. *Software Foundations*. URL: https://softwarefoundations.cis.upenn.edu/vfa-current/index.html (date of access: 08.04.2024).
3. Brodal G. S., Okasaki C. Optimal purely functional priority queues. *BRICS report series*. 1996. Vol. 3, no. 37. URL: https://doi.org/10.7146/brics.v3i37.20019 (date of access: 19.04.2024).
4. Coquand T. Coq Proof Assistant official site. *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (date of access: 08.04.2024).
5. Dijkstra E. W. A note on two problems in connection with graphs. *Numerische mathematik*. 1959. Vol. 1, no. 1. P. 269–271. URL: https://doi.org/10.1007/bf01386390 (date of access: 19.04.2024).
6. Fredman M. L., Tarjan R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*. 1987. Vol. 34, no. 3. P. 596–615. URL: https://doi.org/10.1145/28869.28874 (date of access: 18.04.2024).
7. Introduction to algorithms / V. J. Rayward-Smith et al. *The journal of the operational research society*. 1991. Vol. 42, no. 9. P. 816. URL: https://doi.org/10.2307/2583667 (date of access: 19.04.2024).
8. Okasaki C. Purely functional data structures. Cambridge, U.K : Cambridge University Press, 1998. 220 p.
9. Sedgewick R. Algorithms. Reading, Mass : Addison-Wesley, 1984. 551 p.
10. Vuillemin J. A data structure for manipulating priority queues. *Communications of the ACM*. 1978. Vol. 21, no. 4. P. 309–315. URL: https://doi.org/10.1145/359460.359478 (date of access: 18.04.2024).

ДОДАТОК: ЛІСТИНГ ПРОГРАМНОГО КОДУ

```
Require Import Coq.Lists.List.
Import ListNotations.
Load "Prique".

Module Type ElemType.
  Parameter t : Type.
End ElemType.

Module Type PRIQWITHINFO (E : ElemType).
  Definition key := (nat * E.t) % type. (* Визначення ключа як
пари *)
  (* Основні типи і функції *)
  Parameter priqueue: Type.
  Parameter empty : priqueue.
  Parameter insert : key -> priqueue -> priqueue.
  Parameter delete_min : priqueue -> option (key * priqueue).
  Parameter find_min : priqueue -> option key.
  Parameter merge : priqueue -> priqueue -> priqueue.
  Parameter priq : priqueue -> Prop.
  Parameter Abs : priqueue -> list key -> Prop.

  (* Аксіоми, що визначають поведінку структури *)
  Axiom can_relate : forall p, priq p -> exists al, Abs p al.
  Axiom abs_perm : forall p al bl,
    priq p -> Abs p al -> Abs p bl -> Permutation al bl.
  Axiom empty_priq : priq empty.
  Axiom empty_relate : Abs empty [].
  Axiom insert_priq : forall k p, priq p -> priq (insert k p).
  Axiom insert_relate :
      forall p al k, priq p -> Abs p al -> Abs (insert k p)
(k::al).
  Axiom delete_min_None_relate :
      forall p, priq p -> (Abs p [] <-> delete_min p = None).
  Axiom delete_min_Some_priq :
      forall p q k, priq p -> delete_min p = Some (k,q) ->
priq q.
  Axiom merge_priq : forall p q, priq p -> priq q -> priq
(merge p q).
  Axiom merge_relate :
      forall p q pl ql al,
      priq p -> priq q -> Abs p pl -> Abs q ql -> Abs (merge p
q) al ->
      Permutation al (pl ++ ql).
End PRIQWITHINFO.
```

```
Module SkewBinomialHeap (E : ElemType) <: PRIQWITHINFO E.

Definition key := (nat * E.t) % type.

Inductive Tree : Type :=
  | Node : nat -> key -> list key -> list Tree -> Tree.

Definition priqueue := list Tree.

Definition empty : priqueue := nil.

Definition isEmpty (h : priqueue) : bool :=
  match h with
  | [] => true
  | _ => false
  end.

Definition rank (t : Tree) : nat :=
  match t with
  | Node r _ _ _ => r
  end.

Definition root (t : Tree) : key :=
  match t with
  | Node _ x _ _ => x
  end.

Definition link (t1 t2: Tree) : Tree :=
  match t1, t2 with
  | Node r (x1, e1) xs1 c1, Node _ (x2, e2) xs2 c2 =>
      if x1 >? x2
      then Node (S r) (x2, e2) xs2 (t1::c2)
      else Node (S r) (x1, e1) xs1 (t2::c1)
  end.

Definition skew_link (x: key) (t1 t2: Tree): Tree :=
  let t := link t1 t2
  in (
    match x, t with
    | (n, k), Node r (y, e) ys c   =>
        if n >? y
        then Node r (y, e) ((n, k)::ys) c
        else Node r (n, k) ((y, e)::ys) c
    end
  ).

Fixpoint ins_tree (t: Tree) (q: priqueue): priqueue :=
```

```
  match q with
  | nil    => [t]
  | th::ts => if rank t <? rank th
    then t::q
    else ins_tree (link t th) ts
  end.

From Coq Require Import Program.

Program Fixpoint merge_trees (ts1 ts2: priqueue) {measure
(length ts1 + length ts2)}: priqueue :=
  match ts1, ts2 with
  | _, nil           => ts1
  | nil, _           => ts2
  | t1::tl1, t2::tl2 => if rank t1 <? rank t2
    then t1::(merge_trees tl1 ts2)
    else if rank t2 <? rank t1
         then t2::(merge_trees ts1 tl2)
         else ins_tree (link t1 t2) (merge_trees tl1 tl2)
  end.
Next Obligation.
  simpl. lia.
Defined.
Next Obligation.
  simpl. lia.
Defined.


Definition normalize (q: priqueue): priqueue :=
  match q with
  | nil   => nil
  | t::ts => ins_tree t ts
  end.

Definition insert (k: key) (q: priqueue): priqueue :=
  match q with
  | t1::t2::ts  => if (rank t1) =? (rank t2)
    then (skew_link k t1 t2)::ts
    else (Node 0 k nil nil)::q
  | _           => (Node 0 k nil nil)::q
  end.

Definition merge (ts1 ts2: priqueue): priqueue := merge_trees
(normalize ts1) (normalize ts2).

Fixpoint remove_min_tree (q: priqueue): option (Tree *
priqueue) :=
```

```
  match q with
  | nil    => None
  | t::ts  => let p := remove_min_tree ts in (
    match p with
    | Some (t1, ts1) => (
      match root t, root t1 with
      | (x, e), (x1, e1)  => if x >? x1 then Some (t1, t::ts1)
else Some (t, ts)
      end
    )
    | _ => Some (t, nil)
    end
  )
  end.

Definition find_min (q: priqueue): option key :=
  match q with
  | nil => None
  | _   => let p := remove_min_tree q in (
    match p with
    | Some (t, ts) => Some (root t)
    | _            => None
    end
  )
  end.

Fixpoint insert_all (l: list key) (q: priqueue): priqueue :=
  match l with
  | nil   => q
  | x::xs => insert_all xs (insert x q)
  end.


Definition delete_min (q: priqueue): option (key * priqueue)
:=
  let p := remove_min_tree q in (
    match p with
    | Some (Node _ x xs ts1, ts2) => Some (x, (insert_all xs
(merge (rev ts1) ts2)))
    | None                        => None
    end
  ).

Definition fst_key (p : key) : nat := match p with
                                      | (x, _) => x
                                      end.
```

```
(*
Fixpoint is_heap_tree' (t: Tree) (pr: nat) : bool :=
  match t with
  | Node r (x, _) xs children => (r <? pr) &&
    ( forallb (fun child => match child with
                            | Node _ (y, _) _ _ => (Nat.leb x
y) && is_heap_tree' child r
                            end
              ) children
    ) && forallb (Nat.leb x) (map fst_key xs)
  end.
*)
(* Function to recursively check heap property returning Prop
*)
Fixpoint tree_size (t : Tree) : nat :=
  match t with
  | Node _ _ _ children =>
      1 + fold_right (fun child acc => tree_size child + acc)
0 children
  end.

Program Fixpoint is_heap_tree' (t : Tree) (pr : nat) {measure
(tree_size t)}: Prop :=
  match t with
  | Node r (x, _) xs children =>
      (r < pr) /\
      (Forall (fun child =>
        match child with
        | Node _ (y, _) _ _ => is_heap_tree' child r /\ (x <=
y)
        end) children) /\
      (Forall (fun z => x <= fst z) xs)
  end.
Next Obligation.
Admitted.

(* Root function to initiate the recursive check *)
Definition is_heap_tree (t : Tree) : Prop :=
  match t with
  | Node r (x, _) xs children =>
      (Forall (fun child =>
        match child with
        | Node _ (y, _) _ _ => (x <= y) /\ is_heap_tree' child
r
        end) children) /\
      (Forall (fun z => x <= fst_key z) xs)
  end.
```

```
(* Additional check for rank decrease in heap *)
Fixpoint is_heap_tail (ts: list Tree) (pr: nat): Prop :=
  match ts with
    | nil    => True
    | t::ts1 => is_heap_tree t /\ (pr < (rank t)) /\
is_heap_tail ts1 (rank t)
  end.


Definition priq (q: priqueue) : Prop :=
  match q with
  | nil        => True
  | t::nil     => is_heap_tree t
  | t1::t2::ts => (rank t1) <= (rank t2) /\ is_heap_tree(t1)
/\ is_heap_tree(t2) /\ is_heap_tail ts (rank t2)
  end.

Definition min_key (a b : key) : key :=
  if fst a <=? fst b then a else b.

(* Function to find the minimum element in a list of naturals
*)
Fixpoint list_min (l : list key) : option key :=
  match l with
  | [] => None
  | [x] => Some x
  | x :: xs =>
    match list_min xs with
    | Some m => Some (min_key x m)
    | None => Some x
    end
  end.

(* Function to remove the first occurrence of a given element
in a list *)
Fixpoint remove_first (x : key) (l : list key) : list key :=
  match l with
  | [] => []
  | h :: t => if fst h =? fst x then t else h :: remove_first
x t
  end.

(* Function to delete the minimum element and return it with
the rest of the list *)
```

```
Definition delete_min_list (l : list key) : option (key * list
key) :=
  match list_min l with
  | Some m => Some (m, remove_first m l)
  | None => None
  end.

Fixpoint Abs' (q : priqueue) (lk : list key) (n : nat) {struct
n} : Prop :=
  match n with
  | 0 => match q, lk with
    | nil, nil => True
    | _, _ => False
    end
  | S h =>
    let pq := delete_min q in
    let pl := delete_min_list lk in
    match pq, pl with
    | None, None => True
    | None, _ => False
    | _, None => False
    | Some (mq, pqs), Some (ml, pls) =>
        fst mq = fst ml /\ Abs' pqs pls h
    end
  end.

Definition Abs (q : priqueue) (lk : list key) : Prop := Abs' q
lk (length lk).

Theorem empty_priq : priq empty.
Proof.
  simpl.
  trivial.
Qed.

Theorem empty_relate : Abs empty [].
Proof.
  unfold Abs.
  simpl.
  trivial.
Qed.

Theorem delete_min_None_relate :
  forall p, priq p -> (Abs p [] <-> delete_min p = None).
Proof.
  intros p Hp.
  split.
```

```
  - intro H.
    unfold delete_min.
    destruct (remove_min_tree p) eqn:Hr.
    + unfold Abs in H. unfold Abs' in H. simpl in H. destruct
p.
      * simpl in Hr. discriminate Hr.
      * exfalso. exact H.
    + reflexivity.
  - intro H.
    unfold delete_min in H.
    destruct (remove_min_tree p) eqn:Hr.
    + destruct p0. destruct t. discriminate H.
    + unfold Abs. unfold Abs'. simpl. destruct p eqn:Hep.
      * trivial.
      * simpl in Hr. destruct p0.
        -- discriminate Hr.
        -- destruct (remove_min_tree (t0 :: p0)) eqn:Hrmt.
          ++ destruct p1; destruct t; destruct t1;
             destruct (root (Node n k l l0));
             destruct (root (Node n0 k0 l1 l2));
             destruct (n2 <? n1); discriminate Hr.
          ++ discriminate Hr.
Qed.

Lemma is_heap_tree_skew_link :
  forall k t1 t2,
    is_heap_tree t1 ->
    is_heap_tree t2 ->
    is_heap_tree (skew_link k t1 t2).
Proof.
Admitted.

Lemma skew_link_rank_increment :
  forall k t1 t2,
    is_heap_tree t1 ->
    is_heap_tree t2 ->
    rank (skew_link k t1 t2) = S (rank t1).
Proof.
intros. unfold skew_link. destruct k. unfold link.
destruct t1; destruct t2; destruct k; destruct k0. destruct
(n3 <? n2).
  - destruct (n3 <? n); simpl; reflexivity.
  - destruct (n2 <? n); simpl; reflexivity.
Qed.

Lemma le_S_leq :
  forall a b : nat,
```

```
    a < b -> S a <= b.
Proof.
  intros a b H.
  apply Nat.succ_lt_mono in H.
  apply le_S_n.
  apply H.
Qed.

Theorem insert_priq : forall k p, priq p -> priq (insert k p).
Proof.
  intros k p Hp.
  unfold insert.
  destruct p as [|t1 p].
  - simpl. destruct k as [x _]. split; apply Forall_nil.
  - destruct p as [|t2 p].
    + simpl. split. (* p = [t1], single Node insertion *)
      * apply Nat.le_0_l.
      * destruct k as [x _]. split.
      -- split; apply Forall_nil.
      -- split.
        ++ simpl in Hp. exact Hp.
        ++ trivial.
    + destruct (rank t1 =? rank t2) eqn:Eq.
      * simpl. (* p = t1 :: t2 :: p, rank t1 = rank t2 *)
        unfold priq in *.
        destruct Hp as [Hp1 [Hp2 Hp3]].
        apply Nat.eqb_eq in Eq. destruct p eqn:Hpeq.
        -- simpl. apply is_heap_tree_skew_link.
          ++ exact Hp2.
          ++ destruct Hp3 as [Hp31 Hp32]. exact Hp31.
        -- destruct Hp3 as [Hp31 Hp32]. unfold is_heap_tail in
Hp32.
           destruct Hp32 as [Hp321 [Hp322 H_tail]]. split.
          ++ rewrite skew_link_rank_increment.
            ** symmetry in Eq. rewrite Eq in Hp322. apply
le_S_leq. exact Hp322.
            ** exact Hp2.
            ** exact Hp31.
          ++ split.
            ** apply is_heap_tree_skew_link.
              --- exact Hp2.
              --- exact Hp31.
            ** split.
              --- exact Hp321.
              --- exact H_tail.
      * simpl. split.
        -- apply Nat.le_0_l.
```

```
        -- destruct k as [x _]. split.
          ++ split; apply Forall_nil.
          ++ split.
            ** simpl in Hp. destruct Hp as [Hp1 [Hp2 Hp3]].
exact Hp2.
            ** split.
              --- simpl in Hp. destruct Hp as [Hp1 [Hp2 [Hp3
Hp4]]]. exact Hp3.
              --- split.
                +++ simpl in Hp. destruct Hp as [Hp1 _]. apply
Nat.eqb_neq in Eq.
                    apply Nat.le_neq. split.
                  *** exact Hp1.
                  *** exact Eq.
                +++ simpl in Hp. destruct Hp as [Hp1 [Hp2 [Hp3
Hp4]]]. exact Hp4.
Qed.

Axiom delete_min_Some_priq :
    forall p q k, priq p -> delete_min p = Some (k,q) -> priq
q.

Axiom merge_priq : forall p q, priq p -> priq q -> priq (merge
p q).
Axiom insert_relate :
    forall p al k, priq p -> Abs p al -> Abs (insert k p)
(k::al).
Axiom can_relate : forall p, priq p -> exists al, Abs p al.

Axiom delete_min_Some_relate :
    forall (p q : priqueue) k (pl ql : list key),
    priq p -> Abs p pl -> delete_min p = Some (k,q) -> Abs q
ql ->
    Permutation pl (k::ql) /\ Forall (fun x => fst x >= fst k)
ql.
Axiom abs_perm : forall p al bl,
  priq p -> Abs p al -> Abs p bl -> Permutation al bl.
Axiom merge_relate :
    forall p q pl ql al,
    priq p -> priq q -> Abs p pl -> Abs q ql -> Abs (merge p
q) al ->
    Permutation al (pl ++ ql).

End SkewBinomialHeap.


Module BootstrapQueue <: PRIQUEUE.
```

```
Inductive BPQ : Type :=
| bpq : SkewBinomialHeap(BPQ).prique -> BPQ.

Definition key := nat.

Definition priqueue := option (nat * BPQ) % type.

Definition empty: priqueue := None.

Definition merge: (p q: priqueue): priqueue :=
  match p q with
  | _, None => p
  | None, _ => q
  | Some (n1, bpq q1), Some (n2, bpq q2) => if n1 <? n2
    then Some (n1, bpq (SkewBinomialHeap(BPQ).insert q1 q))
    else Some (n2, bpq (SkewBinomialHeap(BPQ).insert q2 p))
  end.


Definition insert: (k: key) (p: priqueue): priqueue := merge
(k, bpq SkewBinomialHeap(BPQ).empty) p.

Definition find_min: (p: priqueue): option key :=
  match p with
  | None => None
  | Some (x, _) => Some x
  end.

Definition delete_min: (p: priqueue): option (key * priqueue)
:=
  match p with
  | None => None
  | Some (x, bpq q) => let pr :=
SkewBinomialHeap(BPQ).delete_min q in (
      match pr with
        | None => Some (x, None)
        | Some ((x1, bpq q2), q1) => Some (x, Some (x1, bpq
(SkewBinomialHeap(BPQ).merge q1 q2)))
      end
    )
  end.


Definition priq: (p: priqueue): Prop :=
  match p with
  | None => True
```

```
  | Some (x, bpq q) => let pr := delete_min p in (
      match pr with
        | None => True
        | Some (x1, p2) => x1 = x /\
SkewBinomialHeap(BPQ).priq q
      end
    )
  end.

Definition min_key (a b : key) : key :=
  if a <=? b then a else b.

(* Function to find the minimum element in a list of naturals
*)
Fixpoint list_min (l : list key) : option key :=
  match l with
  | [] => None
  | [x] => Some x
  | x :: xs =>
    match list_min xs with
    | Some m => Some (min_key x m)
    | None => Some x
    end
  end.

(* Function to remove the first occurrence of a given element
in a list *)
Fixpoint remove_first (x : key) (l : list key) : list key :=
  match l with
  | [] => []
  | h :: t => if h =? x then t else h :: remove_first x t
  end.

(* Function to delete the minimum element and return it with
the rest of the list *)
Definition delete_min_list (l : list key) : option (key * list
key) :=
  match list_min l with
  | Some m => Some (m, remove_first m l)
  | None => None
  end.


Program Fixpoint Abs (q : priqueue) (lk : list key) {measure
(length lk)} : Prop :=
  let pq := delete_min q in
  let pl := delete_min_list lk in
```

```
    match pq, pl with
    | None, None => True
    | None, _ => False
    | _, None => False
    | Some (mq, pqs), Some (ml, pls) => mq = ml /\ Abs pqs pls
    end.
Next Obligation.
Admitted.
Next Obligation.
Admitted.

Axiom can_relate:  forall p, priq p -> exists al, Abs p al.
Axiom empty_priq: priq empty.
Axiom empty_relate:  Abs empty nil.

Axiom insert_priq: forall k p, priq p -> priq (insert k p).
Axiom delete_min_None_relate:
      forall p, priq p -> (Abs p nil <-> delete_min p = None).
Axiom delete_min_Some_priq:
    forall p q k, priq p -> delete_min p = Some(k,q) -> priq
q.

Axiom insert_relate:
      forall p al k, priq p -> Abs p al -> Abs (insert k p)
(k::al).
Axiom merge_priq: forall p q, priq p -> priq q -> priq (merge
p q).


Axiom delete_max_Some_relate:
forall (p q: priqueue) k (pl ql: list key), priq p ->
 Abs p pl ->
 delete_max p = Some (k,q) ->
 Abs q ql ->
 Permutation pl (k::ql) /\ Forall (ge k) ql.
Axiom merge_relate:
  forall p q pl ql al,
     priq p -> priq q ->
     Abs p pl -> Abs q ql -> Abs (merge p q) al ->
     Permutation al (pl++ql).
Axiom abs_perm: forall p al bl,
 priq p -> Abs p al -> Abs p bl -> Permutation al bl.

Definition priqueue := list key.
(* Implementations of insert, delete_min, etc. *)
End BootstrappedQueue.
```