Министерство образования и науки Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет фотоники и оптоинформатики

Кафедра Компьютерной фотоники и видеоинформатики

Отчет по научно-исследовательской работе

Тема:

Исследование левосторонней и косой сливаемых куч, их реализация и ускорение для последующего анализа эффективности относительно базовых реализаций и стандартных решений

Студент _____ _Краюшкин О.Н._ Группа 5357
(подпись) (Ф.И.О)

Руководитель _____
(Ф.И.О.,должность, уч. степень)

_____ _____
(подпись) (дата)

Работа защищена "___"_____ 20__ г.

с оценкой _____

_____
(подпись)

Санкт-Петербург
2015 г.

Введение

Куча (англ., Heap) – специализированная структура данных, которая является максимально эффективной реализацией абстрактного типа данных «очередь с приоритетом».

Наиболее известная куча – двоичная – обладает хорошей асимптотической оценкой сложности для всех операций, но если стоит задача периодического объединения (слияния) большого количества куч, то эффективность её использования резко падает. Сложность операции объединения для двоичной кучи – $\Theta(n)$. Здесь к месту приходятся т.н. левосторонняя (англ., leftist heap) и косая кучи (англ., skew heap), которые идеально подходят для этих целей.

Данная работа рассматривает применение техники ускорения к обеим сливаемым кучам и последующее сравнение эффективности их работы с работой стандартных реализаций структур упорядоченного мультимножества и очереди с приоритетом на основе массива, взятых из STL (Standard Template Library).

Цель работы

Исследовать сливаемые кучи (Leftist и Skew) и реализовать их ускоренные версии для последующего анализа эффективности относительно базовых реализаций и стандартных решений.

Задачи

1. Реализация левосторонней и косой куч.
2. Ускорение операций добавления элемента и слияния для каждой кучи.
3. Тестирование корректности реализации.
4. Перекрестный анализ эффективности обеих куч (с ускорениями и без) в сравнении со стандартными (STL) реализациями структур множества и очереди с приоритетом.

Реализация

Левосторонняя куча (базовая)

Левосторонней является такая куча, для каждого из узлов которой сохраняется следующее требование: расстояние до ближайшего листа в левом поддереве этого узла больше или равно аналогичному расстоянию в его правом поддереве.

Основной целью сливаемых куч является их быстрое объединение. Операция слияния, реализованная во внутреннем методе `merge`, является основной для левосторонней кучи, на ней строятся операции добавления нового элемента и извлечения минимума. Добавление элемента – слияние с кучей из одного элемента; извлечение минимума – слияние поддеревьев корня исходной кучи.

В `leftist_heap.cpp` реализован шаблон класса LHeap со следующим основным интерфейсом:

```
void build(InputIterator first, InputIterator last) .... O(n)
void add(T val) ..................................... O(log n)
void mergeWithHeap(LHeap<T>& h) ..................... O(log n)
bool empty() ........................................ O(1)
T min() ............................................. O(1)
void pop() .......................................... O(log n)
void clear() ........................................ O(n)
void print() ........................................ O(n)
bool checkMin() ..................................... O(n)
```

и несколькими внутренними методами:

```
void clear(LNode<T> * n) ............................ O(n)
LNode<T> * merge(LNode<T> * a, LNode<T> * b) ....... O(log n)
void printSubtree(LNode<T>* n, int level) ........... O(n)
bool checkMin(LNode<T>* n) .......................... O(n)
```

Косая куча (базовая)

Реализация косой кучи не столь сильно отличается от реализации кучи левосторонней. По сути, вместо того, чтобы пытаться сохранить более нагруженным левого потомка каждого узла, косая куча решает менять местами всех потомков во время обратного хода рекурсии при операции слияния. Может показаться, что такой подход ничем не оправдан, но амортизационный анализ сложности показывает, что асимптотическая сложность для операций добавления или удаления элемента, а также для слияния куч – *O(log n)*.[1] В файле `skew_heap.cpp` реализован шаблон класса `SHeap` с аналогичным классу `LHeap` интерфейсом, но амортизированными оценками сложности для вышеупомянутых операций.

Отложенное добавление элемента

В `leftist_heap_boost.cpp` и `skew_heap_boost.cpp` была улучшена асимптотика операции добавления нового элемента – `add()`. Здесь так же реализован аналогичный интерфейс, с той разницей, что `add()` выполняется здесь за *O(1)*, а оценка *O(k + log n)* для операции изъятия минимума теперь амортизировано учитывает *k* добавлений в кучу с числом элементов *n*.

Это достигается благодаря заведению дополнительного контейнера для вновь добавляемых объектов и дополнительной переменной, в которой всегда хранится обновляемый минимум. Как только требуется выполнить операцию слияния или извлечения минимума (т.е. одну, из работающих за *O(log n)*), происходит построение двоичной кучи за *O(k)* из объектов дополнительного контейнера, после чего происходит слияние куч за *O(log n)* и последующее извлечение минимума за *O(log n)*.

Таким образом, от оценки *O(k log n)* для *k* добавлений удалось перейти к оценке *O(k + log n)*.

Отложенное слияние

В `leftist_heap_merge_boost.cpp` и `skew_heap_merge_boost.cpp` реализованы надстройки над описанными выше кучами, улучшающие оценку на операцию слияния до амортизированного *O(1)*.

Это гораздо менее тривиальное улучшение асимптотической оценки операции. Предположим, мы хотим хранить в куче объекты типа `T`. Для этого заведем шаблон класса `B<T>`, экземпляр которого представляет собой одно из поддеревьев кучи и состоит из объекта типа `T`, являющегося минимумом данного поддерева, и очереди `Q<B<T>>`, содержащей все остальное поддерево.

Другими словами, мы разрешаем хранить в куче другую кучу. Изначальная идея, называемая авторами Data-structural bootstrapping [2], принадлежит Роберту Тарьяну и Адаму Буксбауму.

При применении обоих описанных улучшений первое фактическое построение кучи происходит лишь при первом запросе на извлечение минимума. Это приводит к интересному эффекту. Если, например, в две такие кучи было добавлено по n элементов, после чего кучи были слиты, произойдет простое слияние двух массивов элементов, ожидающих добавления в кучу, после чего за $O(n)$ будет впервые построена куча.

Эти модификации куч реализованы не в виде шаблонов классов, т.к. иначе пришлось бы применять специализированное шаблонирование для любого используемого контейнера `Q`, что выходит за рамки этой работы. Тем не менее, приведенная специализация для типа данных `int` алгоритмически неотличима.

Тестирование

Тестирование всех классов выполнялось по единой схеме. Первый тест являлся проверкой корректности работы, второй ставил своей целью сравнение времени выполнения ключевых операций разных модификаций куч, а так же стандартных решений.

Тест на корректность построения (`test_wa.cpp`) 10 000 раз строит и разбирает кучи различных размеров на случайно сгенерированных данных, поверяя на корректность очередной извлеченный минимум.

Тест на сравнение времени работы (`test_tl_compare.cpp`) на основе $n \cdot k$ случайно сгенерированных элементов, путем их последовательного добавления, строит k куч по n элементов, выполняет их слияние в одну и разбирает получившуюся кучу, последовательно извлекая минимумы.

Вышеописанное сопровождается измерением затраченного времени в ключевых точках:

- после построения всех куч данного вида
- после слияния всех куч вида
- после первого извлечения минимума из объединенной кучи (это время показательно для куч с амортизированной оценкой на сложность операции добавления элемента)
- после полного разбора кучи

Тест выполнялся в 10 проходов. От прохода к проходу менялись только наборы данных, на которых работали структуры. Это производилось для последующего усреднения времени выполнения операций для снижения влияния на него конкретных наборов данных, а также для последующей оценки этого влияния.

Каждый из проходов состоял из 2 частей, во время которых один из параметров n и k оставался постоянным, в то время как второй менялся. При каждой итерации смены значения одного из параметров генерировался новый набор данных, на нем поочередно отрабатывали все операции все модификации куч, а также реализации структур множества и очереди с приоритетом на массиве, взятые из STL.

Постоянное значение параметров k и n, равное двум, было задано исходя из того, что одной из анализируемых операций является слияние, для которого необходимо, по меньшей мере, 2 кучи. На вычислительной машине, используемой для расчетов, была возможность выделить объем мощностей достаточный, для обработки кучами порядка 10^7 элементов типа `int`, исходя из чего, была определена верхняя граница для изменяющегося параметра – $5 \cdot 10^6$. Нижняя – 10^4 – была определена экспериментально – при меньшем значении переменного параметра операции выполняются слишком быстро для корректного измерения времени их выполнения. От нижней границы до

верхней переменный параметр изменяется в геометрической прогрессии с базой равной двум.

Тесты проводились на вычислительной машине со следующими параметрами: процессор Intel® Core™ i5-2450 CPU @ 2.50GHz 2.50GHz, 6,00Гб ОЗУ, ОС Windows 7 Максимальная SP1 x64.

Из полученных данных за 10 проходов было рассчитано среднее арифметическое и СКО (среднеквадратическое отклонение) времени выполнения каждой операции конкретной структурой данных для различного количества куч (k) и элементов в них (n). На их основании были построены графики, приведенные на рисунках 1-8.

Для пар графиков, отражающих зависимость времени выполнения одной операции от разных параметров, для большей наглядности был сохранен одинаковый диапазон значений оси ординат. Порядок перечисления результатов работы различных структур данных в легенде соответствует их расположению на графике. Линии ошибок на графиках отражают СКО.

Результаты тестирования

Операция добавления элемента

На рисунке 1 приведен график зависимости времени выполнения операции добавления в кучи по два элемента от k – количества куч. Этот график не столь интересен, т.к. с ростом числа куч нагрузка возрастает линейно, ведь они никак не связаны между собой в ходе данной операции.

Тем не менее, можно заметить что модификации левосторонней кучи показывают себя чуть хуже, чем аналогичные модификации косой, что связано с дополнительными затратами на расчет расстояния до ближайшего листа.

Лучше всего с добавлением нескольких элементов в большое количество куч справились простейшие реализации левосторонней и кривой

куч. Чуть хуже себя показали модификации, которые как раз улучшают асимптотическую оценку на эту операцию, что связано с заведением для каждой кучи дополнительного массива для хранения вновь добавленных элементов. С бóльшим отрывом идут модификации куч с оптимизированной операцией слияния, расплачиваясь еще и за более сложную структуру. Последнее так же относится к очереди с приоритетом и мультимножеству, которому нужно поддерживать раскраску внутреннего красно-черного дерева.

На рисунке 2 приведен график зависимости времени выполнения операции добавления элементов в две кучи от числа добавляемых элементов. Этот график отражает преимущества куч с операцией добавления за амортизированное время *O(1)* – и левосторонняя, и косая куча с этим улучшением показывают себя лучше других, но стоит помнить, что они на данном этапе лишь собирают элементы в один массив.

Нагрузка сложной структуры сказывается на кучах с оптимизированной оценкой слияния и в условиях малого количества куч большого размера, и если бы не ускорение операции добавления, они отработали бы еще хуже.

Неожиданно плохо в этом тесте себя показала базовая косая куча, хотя и ожидалось, что она будет работать на таких данных чуть хуже, так как её оценка сложности на добавление – O(log n) – изначально амортизированная, тогда как аналогичная оценка для левосторонней кучи – в худшем случае. Видно, что куча плохо справляется с поочередным добавлением большого числа элементов в одну кучу.

Закономерное место здесь занимает очередь с приоритетом – первое, среди структур, осуществляющих добавление сразу же. Это связано с тем, её куча является бинарной.

Рисунок 1. Добавление в *k* куч по 2 элемента

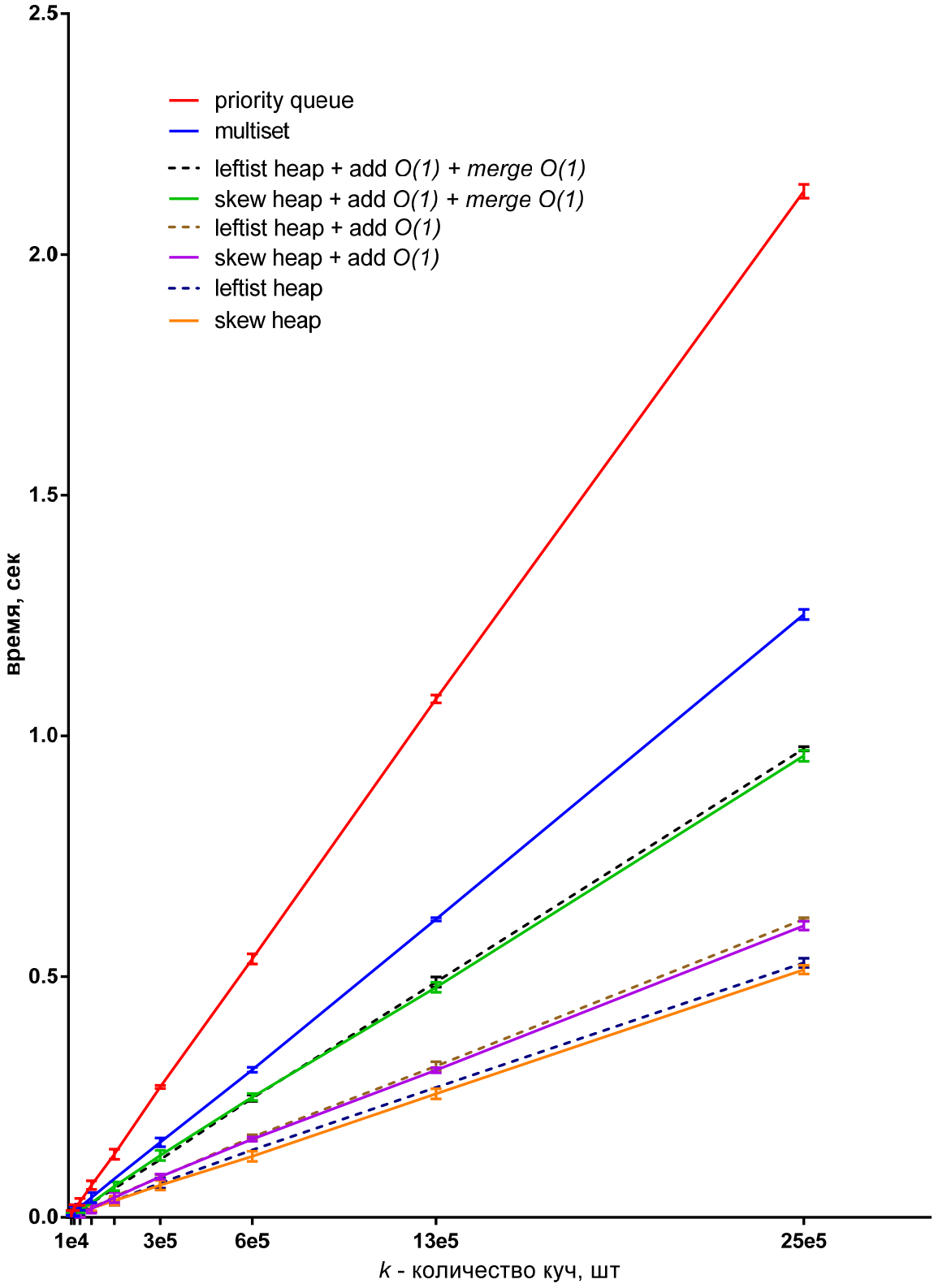
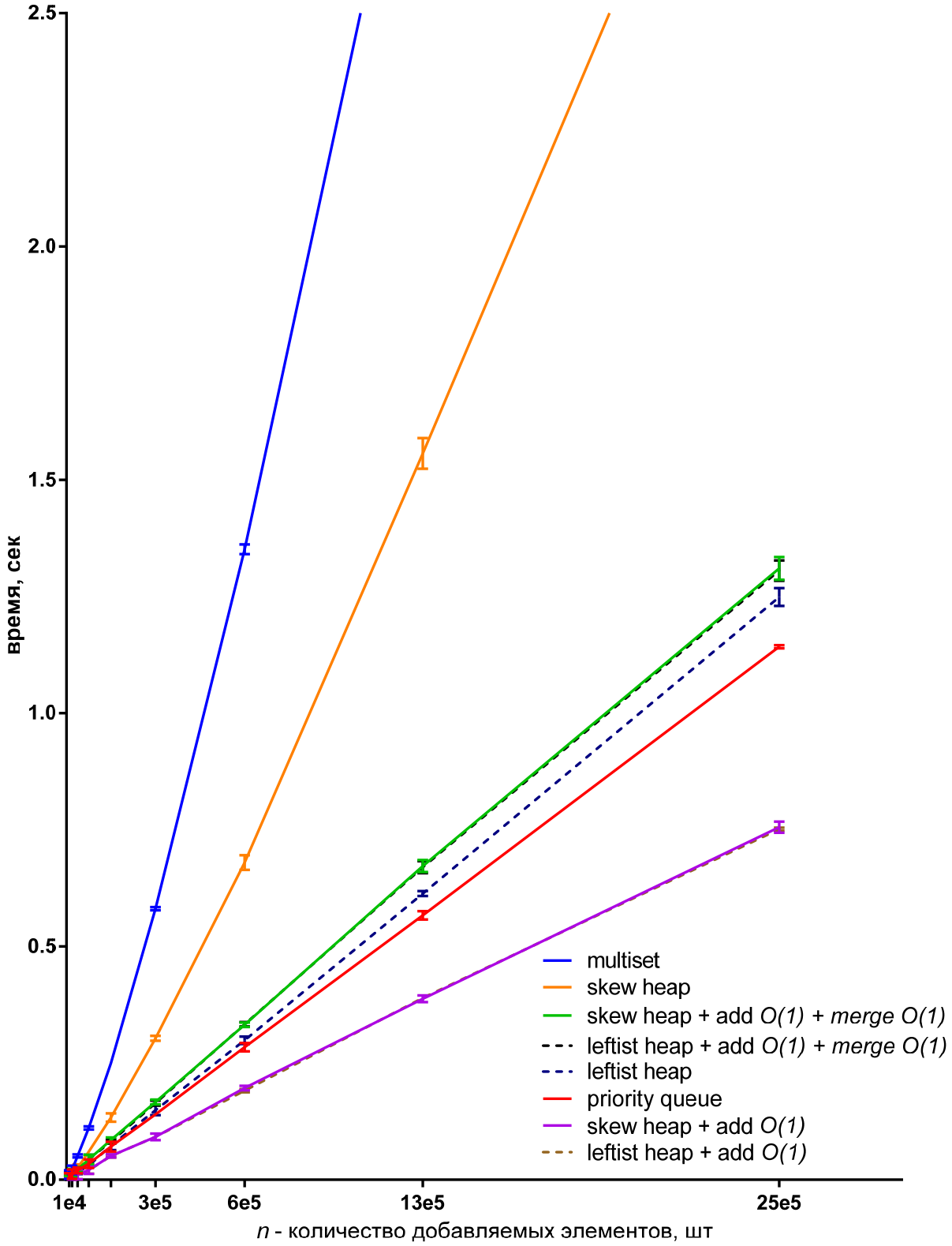


Рисунок 2. Добавление в 2 кучи по *n* элементов



Операция слияния куч

На рисунке 3 приведен график зависимости времени выполнения операции слияния куч по два элемента от числа таких куч. Здесь на первый план выходят модификации левосторонней и косой кучи с улучшенной оценкой слияния, которые выполняют множественное отложенное слияние. Причем, поскольку ни к одной из куч еще не было запроса на извлечение элемента, происходит обычное слияние k массивов, правда с попутным уточнением действительного минимума, т.к. запрос на получение минимума необходимо выполнять за $O(1)$.

Примечательно, что, не теряя времени на работе со сложной внутренней структурой и не откладывая построение кучи, базовая левосторонняя куча отрабатывает наравне со своими модифицированными версиями.

Если не считать тех вышеописанных куч, которые лишь сливают массивы, версии косой кучи показывает себя точно так же как и на графике на рисунке 2. С точки зрения косой кучи ситуация мало чем изменилась – на рисунке 2 в кучу много раз добавлялось по одному элементу, что реализуется через слияние с одноэлементной кучей, здесь же происходит многократное слияние одной кучи с другими двухэлементными кучами.

Аналогичное сходство присуще и графикам зависимостей для очереди с приоритетом и мультимножества с той разницей, что ситуацию для всех сливаемых куч можно сильно улучшить, не сливая одну кучу со всеми остальными подряд, а сделать это так, как реализовано построение новой сливаемой кучи из n элементов за $O(n)$ – добавить все кучи в очередь, вытаскивать по две, сливать их и класть получившуюся в конец очереди – и так, пока не получится одна куча. Однако в данном тесте подразумевалось, что к нам поступают новые кучи, которые надо сразу сливать в одну и быть готовыми быстро отвечать на запросы.

Рисунок 3. Объединение *k* куч, по 2 элемента каждая

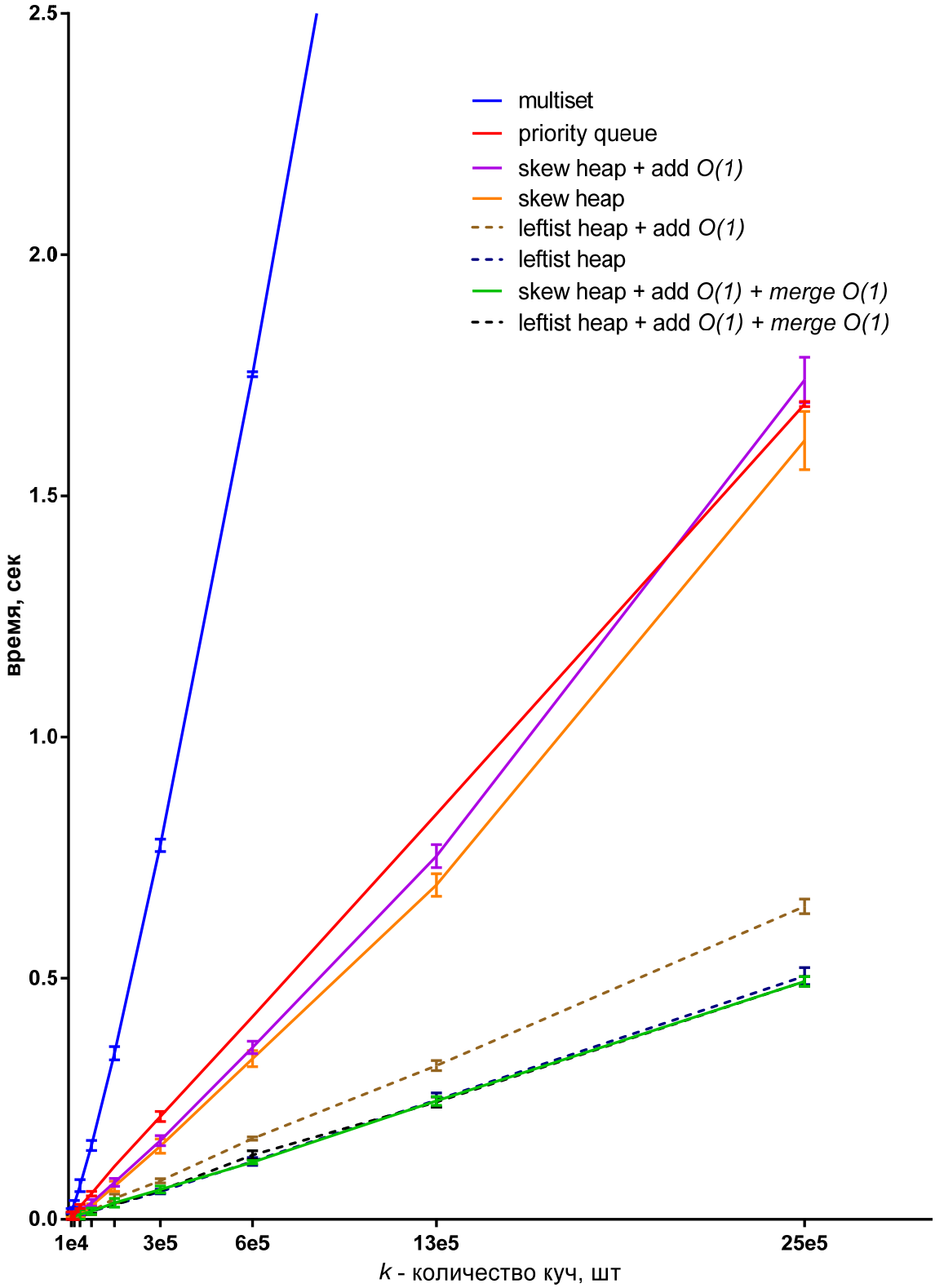
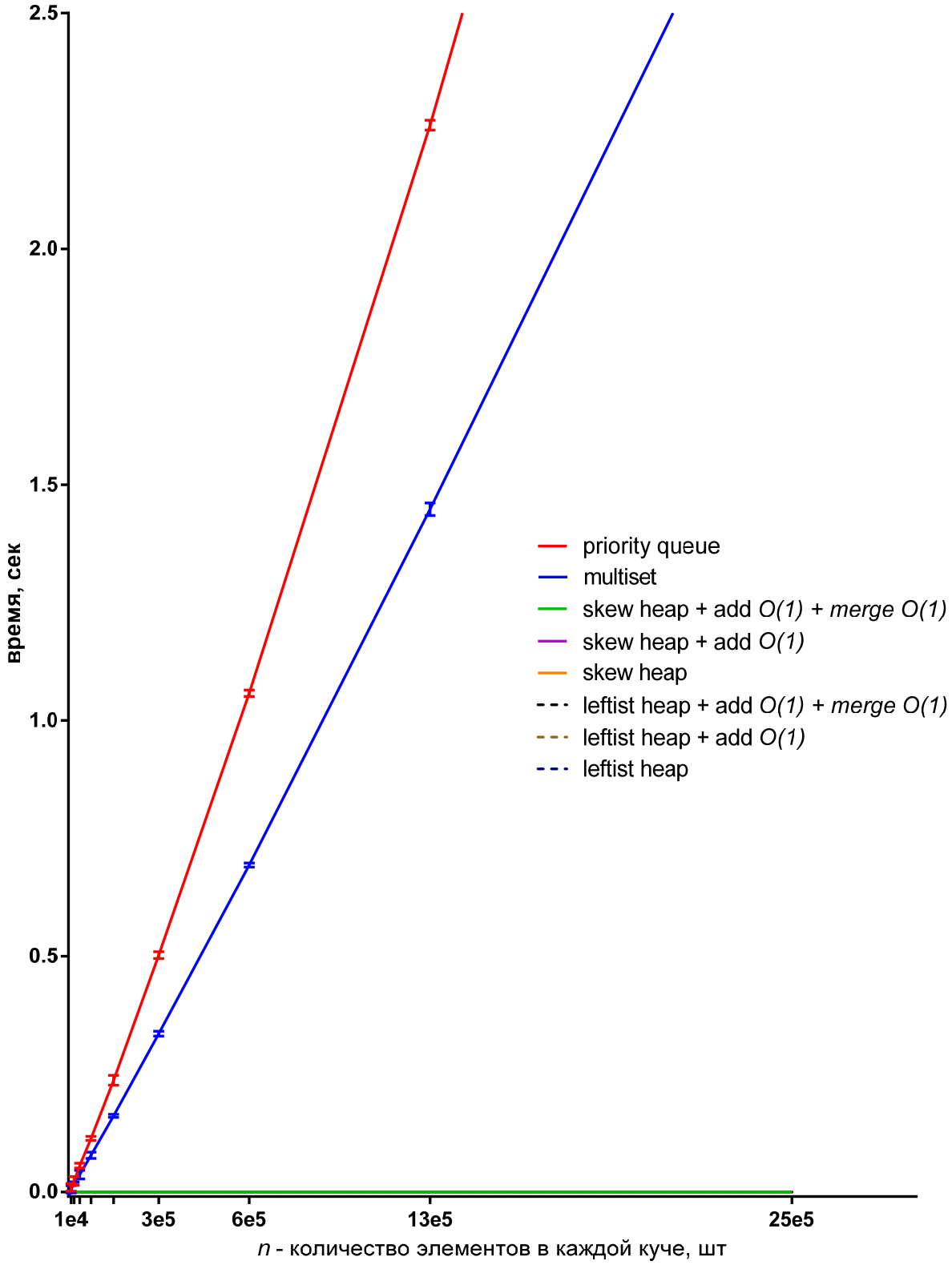


Рисунок 4. Объединение 2-х куч, по *n* элементов каждая



На рисунке 4 приведен график зависимости времени выполнения операции слияния двух куч от числа элементов в них. Именно на схожих задачах проявляются все преимущества сливаемых куч в целом – объединение больших куч даже на базовых левосторонней и косой выполняется менее, чем за миллисекунду, т.к. это всего одна операция со сложностью *O(log n)*.

Операция извлечения минимума

На рисунке 5 приведен график зависимости времени выполнения первой операции извлечения минимума после слияния куч из двух элементов от числа таких куч, на рисунке 6 – аналогичный график, но после слияния двух куч, каждая размером в *n* элементов.

Эта пара графиков хорошо отражает, что представляет амортизированная оценка для модифицированных версий куч. Примечательно, что графики для куч с улучшенной оценкой на операцию слияния обладают большой погрешностью, так как количество выполняемых слияний и построений элементов отложенного добавления во время первого извлечения минимума принципиально зависит от конкретного набора элементов.

Кучи же лишь с модификацией операции добавления ведут себя так по-разному на этой паре графиков, так как в первом случае (рисунок 5) до слияния имелось *k* куч с одним элементом в корне и одним элементом в массиве, ожидающим добавления. После слияния мы получили одну кучу с k элементов в самой куче и еще k – в массиве на добавление, которое и происходит при первом обращении на изъятие минимума. Во втором случае (рисунок 6) до слияния имелось две кучи с одним элементом в корне и *n-1* элементом в массиве, после слияния куча стала содержать два элемента в корне и *2n-2* – в массиве. Естественно, во втором случае необходимо больше времени для построения новой кучи и слияния её с основной.

Рисунок 5. Первое извлечение минимума после слияния *k* куч из 2-х элементов

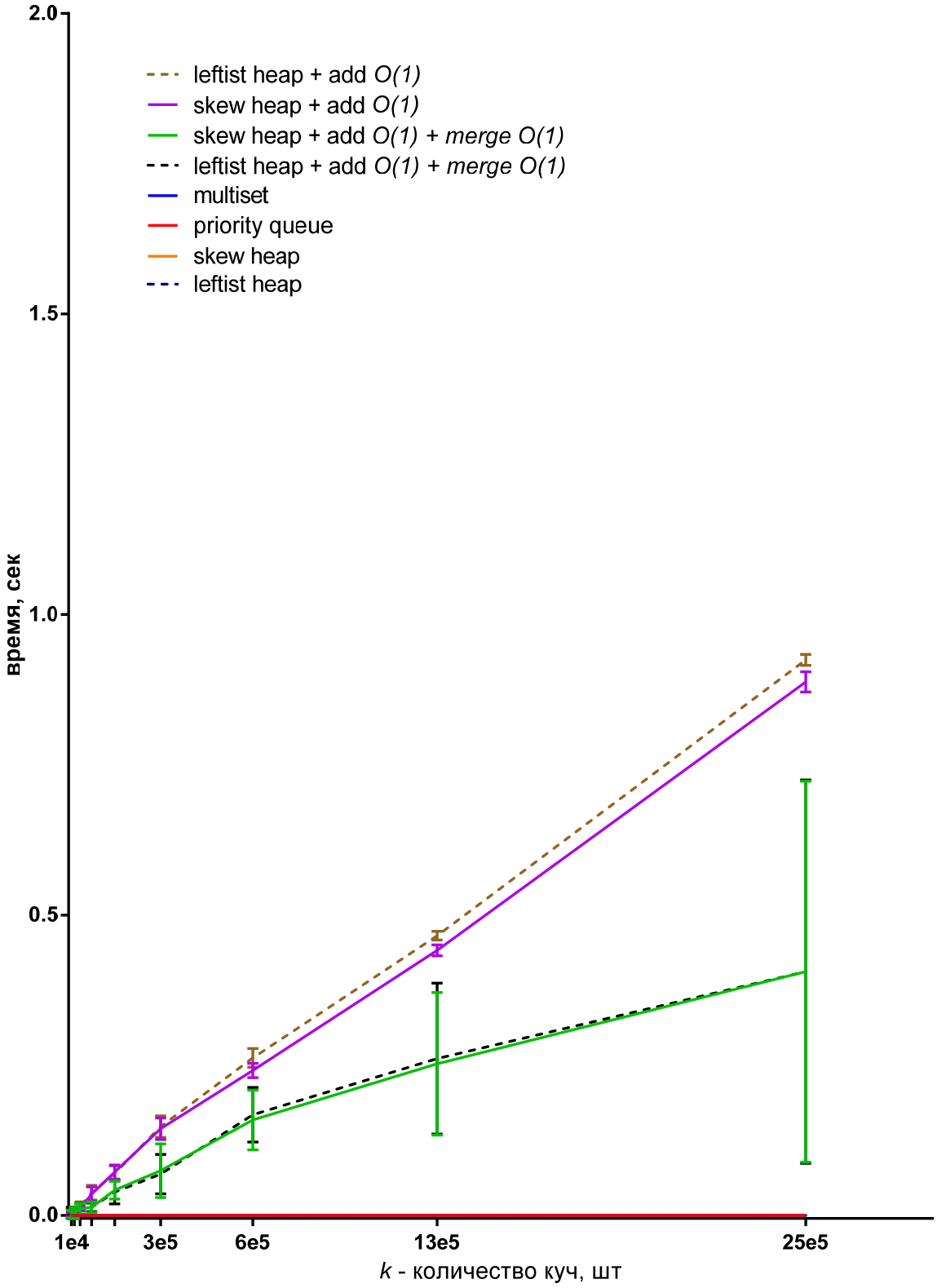
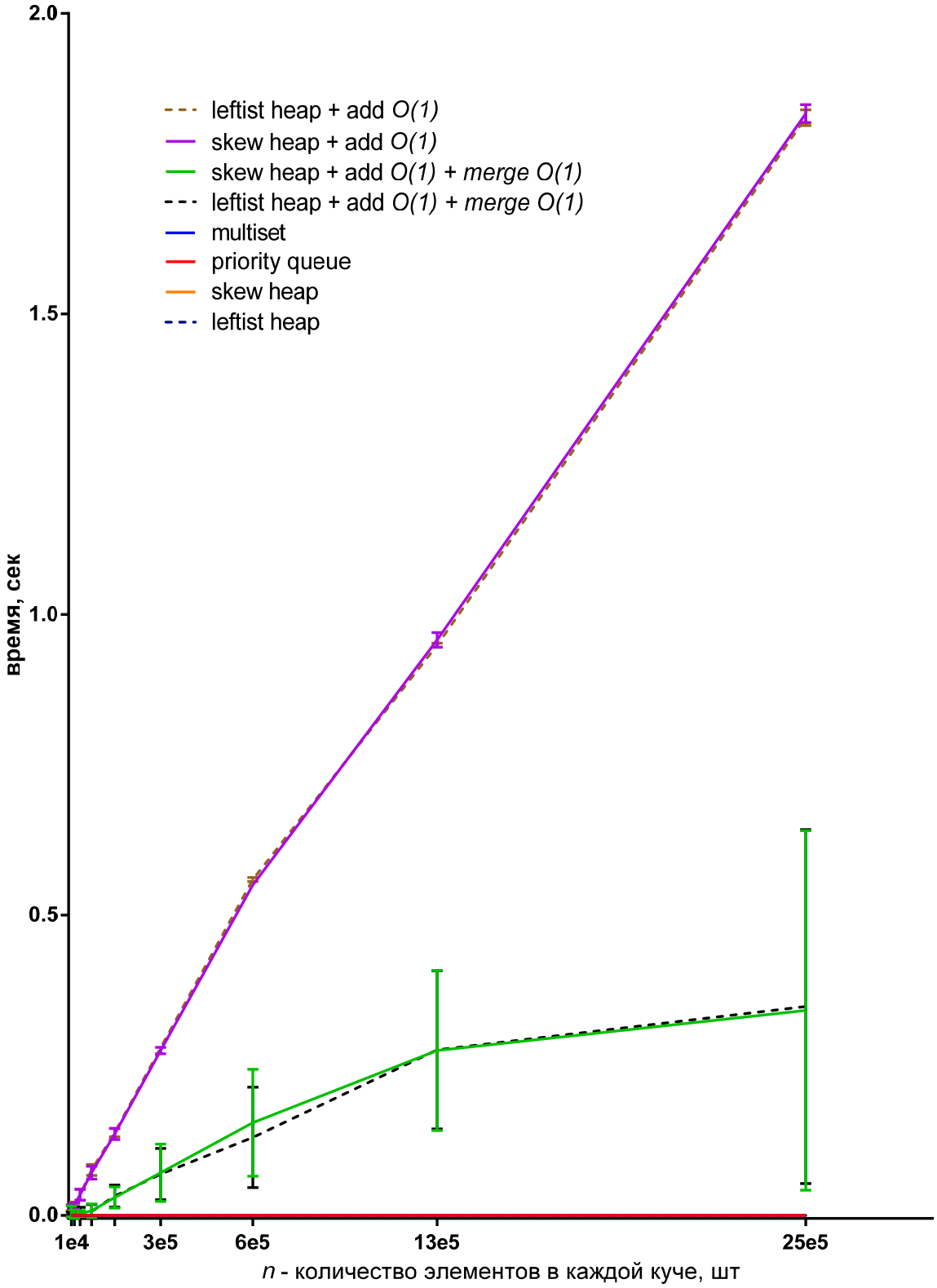


Рисунок 6. Первое извлечение минимума после слияния 2-х куч из *n* элементов



На рисунках 7 и 8 приведены графики зависимости времени последовательного извлечения всех элементов из получившейся кучи. На рисунке 7 – из кучи, получившейся слиянием k куч по два элемента, на рисунке 8 – слиянием двух куч, по n элементов в каждой.

Очень хорошую скорость извлечения минимума показало мультимножество. Не смотря на то, что красно-черное дерево, на котором реализована эта структура в STL, также имеет оценку *O(log n)* на эту операцию, реальное поведение выводит мультимножество в лидеры по суммарному времени, требующемуся на такой набор манипуляций с данными.

Разница на двух графиках не столь велика и для большинства структур находится в пределах погрешностей, однако её нельзя не заметить для модификаций куч с улучшенной оценкой на операцию слияния – им требуется заметно больше времени в задаче, где итоговая куча получилась из множества мелких (рисунок 7). Это связано с тем, что часть слияний все еще находилась в отложенном состоянии. А так как то, насколько большое количество слияний еще не произведено, отчасти зависит от конкретного набора данных, то погрешности для этих модификаций куч относительно прочих структур так же увеличиваются на обоих графиках.

Так же заметно отставание всех модификаций левосторонней кучи от аналогичных версий кучи косой, что объясняется дополнительными затратами на перерасчет расстояний от каждой вершины до ближайшего листа.

Рисунок 7. Последовательное извлечение всех элементов после слияния *k* куч из 2-х элементов

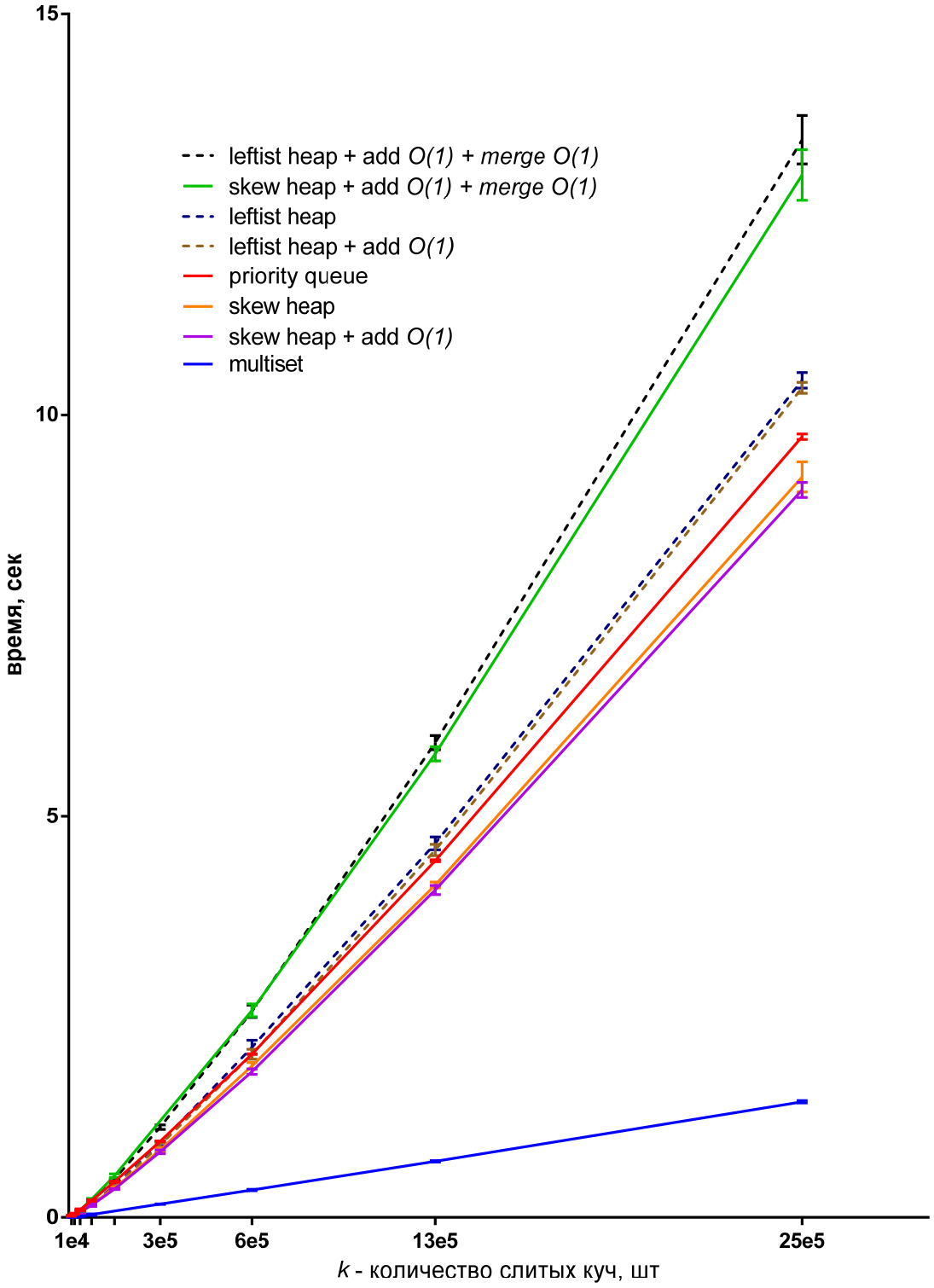
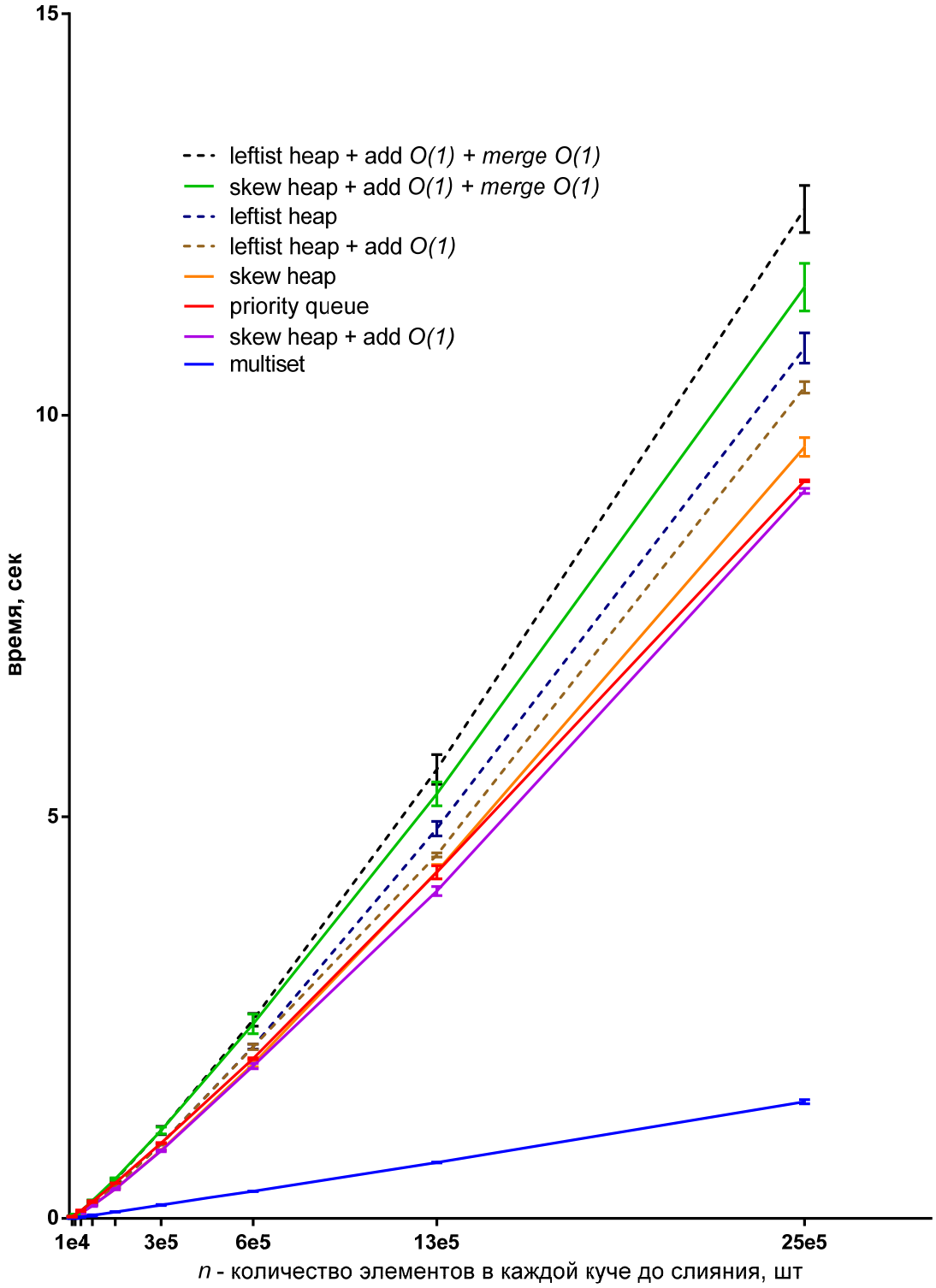


Рисунок 8. Последовательное извлечение всех элементов после слияния 2-х куч из *n* элементов



Заключение

Метод постоянного чередования потомков во время операции слияния плохо сказывается на косых кучах в задачах с частыми слияниями с мелкими кучами (см. рисунки 2 и 3). Если не учитывать этот факт, то в остальном базовые левостороння и косая кучи работают практически с одинаковой скоростью.

Накопление вновь добавляемых элементов особенно хорошо сказывается на косой куче, решая вышеупомянутую проблему даже с учетом дополнительной нагрузки при первом извлечении минимума (см. рисунки 2, 5 и 6).

Из привносимых недостатков можно отметить незначительное снижение скорости работы, проявляющееся при создании куч из небольшого числа элементов (см. рисунок 1).

Возможно, это улучшение стоит расширить на накопление мелких куч для последующего слияния методом, который используется при построении таких куч за *O(n)*. Такое решение лишь частично являлось бы аналогом использованной модификации метода слияния: не внося усложнений во внутреннюю структуру кучи, слияние просто откладывалось бы на некоторое время, позже производясь согласно возрастанию размеров куч. Такое улучшение могло бы устранить или уменьшить проигрыш косой кучи левосторонней куче на задаче множественных слияний с мелкими кучами (см. рисунок 3).

Базовая левосторонняя куча, в отличие от косой, хорошо справляется с многократным поэлементным добавлением (см. рисунки 1 и 2). С учетом добавляемых ею дополнительных вычислений при первом извлечении (см.

рисунки 5 и 6) можно утверждать, что модификация метода добавления не является необходимой.

Модификация сливаемых куч с оценкой на слияние, равной амортизированному *O(1)*, практически одинаково работает как на базе левой, так и на базе косой кучи. Тем не менее, вкупе с накоплением добавляемых элементов, данная модификация, пусть и незначительно, все же показала себя лучше на базе косой кучи.

Ответ на вопрос о резонности этой модификации можно дать, зная, что будет чаще всего требоваться от структуры данных в конкретной задаче, т.к. если речь не идет о полном разборе, то этот метод позволяет избавиться от некоторых ненужных слияний. В тоже время, применение метода влечет за собой дополнительные вычислительные расходы на поддержание сложной структуры.

Очередь с приоритетом и мультимножество ожидаемо проигрывают на задачах слияния, для которых они плохо приспособлены. Тем не менее, эти структуры помогли в сравнении увидеть преимущества и недостатки сливаемых куч, так как обладают аналогичными логарифмическими асимптотическими оценками сложности на все операции, которые, естественно, не означают идентичного поведения на различных задачах, особенности которого и были выяснены.

Список литературы

1. CSE 4101 lecture notes, York University. SKEW HEAPS: Self-Adjusting Heaps // Интернет ресурс: cse.yorku.ca.

2. Buchsbaum A.L., Tarjan R.E. “Confluently persistent deques via data structural bootstrapping”, J. Algorithms 18 (1995), 513-547 cc.

Приложение

Несмотря на весомые алгоритмические различия, реализации левосторонней и косой куч отличаются между собой несколькими строчками кода, связанными с хранением в узлах информации о расстоянии до ближайшего листа и наличием условия для смены левого и правого потомков между собой в теле метода `merge`. Исходя из этого, приводится только базовая реализация косой кучи.

После кода реализаций куч приведен код тестов. Первый тест на проверку корректности приведен на примере базовой левосторонней кучи – для других структур менялись только названия классов. Второй тест работает со всеми структурами и приведен полностью.

Все реализации, результаты тестирования и текст этой работы доступен в репозитории по следующему адресу:
https://github.com/Allight7/leftist_and_skew_heaps_bootstrapped

`leftist_heap.cpp` – базовая версия левосторонней кучи

```
#include <queue>
#include <algorithm>
#include <iostream>

#ifndef _LNODE_
#define _LNODE_
template <class T>
struct LNode {
   int d;
   T val;
   LNode <T> * L;
   LNode <T> * R;

   LNode(T _val) {
      d = 1;
      val = _val;
      L = R = 0;
   }
};
#endif

#ifndef _LHEAP_
#define _LHEAP_
```

```
template <class T>
class LHeap {
  friend class LeftistHeapMergeBootInt;
  LNode<T> * root;

public:

  LHeap() {
    root = 0;
  }

  template <class InputIterator>
  LHeap(InputIterator first, InputIterator last) {
    root = 0;
    build(first, last);
  }

  ~LHeap() {
    clear();
  }

  template <class InputIterator>
  void build(InputIterator first, InputIterator last) { // O(n)
    clear();
    std::queue<LNode<T> *> ptrs;

    while(first != last)
      ptrs.push(new LNode<T>(*first++));

    LNode<T> * a;
    LNode<T> * b;
    while(ptrs.size() > 1){
      a = ptrs.front();
      ptrs.pop();
      b = ptrs.front();
      ptrs.pop();
      ptrs.push(merge(a,b));
    }

    root = ptrs.front();
  }

  void mergeWithHeap(LHeap<T>& h){
    root = merge(root, h.root);
    h.root = 0;
  }

  void print() {
    printSubtree(root,1);
  }

  bool checkMin(){
    return checkMin(root);
  }

  void clear() {
    if(root)
      clear(root);
    root = nullptr;
  }

  void add (T val) {
    if(!root)
```

```
      root = new LNode<T>(val);
    else
      root = merge(root, new LNode<T>(val));
  }

  T min (){
    return root->val;
  }

  void pop(){
    LNode<T> * oldRoot = root;
    root = merge(oldRoot->L, oldRoot->R);
    delete oldRoot;
  }

  bool empty() {
    return root == nullptr;
  }


private:

  void clear(LNode<T> * n) {
    if(n->L) clear(n->L);
    if(n->R) clear(n->R);
    delete n;
  }

  LNode<T> * merge (LNode<T> * a, LNode<T> * b) {
// returns pointer to the head of merged heap;
// if only one of nodes NOTNULL, returns it
    if(!a || !b || a == b){
      if(a && !b) return a;
      if(a == b && a != 0) throw "attempt of self merge";
      return b;
    }
    if(a->val > b->val)
      std::swap(a, b);
    if(!a->R)
      a->R = b;
    else
      a->R = merge (a->R, b);
    if(!a->L || a->R->d > a->L->d)
      std::swap(a->R, a->L);
    a->d = a->R ? a->R->d + 1 : 1;
    return a;
  }

  void printSubtree(LNode<T>* n, int level) {
    if(!n)
      std::cout << std::string(level, ' ') << "-:" << (n? n->d : 0) <<
std::endl;
    else{
      printSubtree(n->R, level+1);
      std::cout << std::string(level, ' ') << (n? n->val : 0) << ':' << (n?
n->d : 0) << std::endl;
      printSubtree(n->L, level+1);
    }
  }

  bool checkMin(LNode<T>* n){
    if(!n) throw "NullPointerException";
    return (n->L ? (n->val <= n->L->val && checkMin(n->L)) : true) &&
```

```
          (n->R ? (n->val <= n->R->val && checkMin(n->R)) : true);
   }
};

#endif
```

skew_heap.cpp – базовая версия косой кучи

```
#include <queue>
#include <algorithm>
#include <iostream>

#ifndef _SNODE_
#define _SNODE_
template <class T>
struct SNode {
   T val;
   SNode <T> * L;
   SNode <T> * R;

   SNode(T _val) {
      val = _val;
      L = R = 0;
   }
};
#endif


#ifndef _SHEAP_
#define _SHEAP_
template <class T>
class SHeap {
   friend class SkewHeapMergeBootInt;
   SNode<T> * root;

public:

   SHeap() {
      root = 0;
   }

   template <class InputIterator>
   SHeap(InputIterator first, InputIterator last) {
      root = 0;
      build(first, last);
   }

   ~SHeap() {
      clear();
   }

   template <class InputIterator>
   void build(InputIterator first, InputIterator last) { // O(n)
      clear();
      std::queue<SNode<T> *> ptrs;

      while(first != last)
         ptrs.push(new SNode<T>(*first++));

      SNode<T> * a;
      SNode<T> * b;
```

```
      while(ptrs.size() > 1){
        a = ptrs.front();
        ptrs.pop();
        b = ptrs.front();
        ptrs.pop();
        ptrs.push(merge(a,b));
      }

      root = ptrs.front();
    }

    void mergeWithHeap(SHeap<T>& h){
      root = merge(root, h.root);
      h.root = 0;
    }

    void print() {
      printSubtree(root,1);
    }

    bool checkMin(){
      return checkMin(root);
    }

    void clear() {
      if(root)
        clear(root);
      root = nullptr;
    }

    void add (T val) {
      if(!root)
        root = new SNode<T>(val);
      else
        root = merge(root, new SNode<T>(val));
    }

    T min (){
      return root->val;
    }

    void pop(){
      SNode<T> * oldRoot = root;
      root = merge(oldRoot->L, oldRoot->R);
      delete oldRoot;
    }

    bool empty() {
      return root == nullptr;
    }


  private:


    void clear(SNode<T> * n) {
      if(n->L) clear(n->L);
      if(n->R) clear(n->R);
      delete n;
    }

    SNode<T> * merge (SNode<T> * a, SNode<T> * b) {  // returns pointer to the
  head of merged heap
```

```
                                        // if only one of nodes NOTNULL,
returns it
     if(!a || !b || a == b){
       if(a && !b) return a;
       if(a == b && a != 0) throw "attempt of self merge";
       return b;
     }
     if(a->val > b->val)
       std::swap(a, b);
     if(!a->R)
       a->R = b;
     else
       a->R = merge (a->R, b);
     std::swap(a->R, a->L);

     return a;
   }

     void printSubtree(SNode<T>* n, int level) {
     std::cout << std::string(level, ' ') << (n? n->val : 0) << ':' << (n? n-
>d : 0) << std::endl;
     if(!n)
       std::cout << std::string(level, ' ') << "-" << std::endl;
     else{
       printSubtree(n->R, level+1);
       std::cout << std::string(level, ' ') << (n? n->val : 0) << std::endl;
       printSubtree(n->L, level+1);
     }
   }

   bool checkMin(SNode<T>* n){
     if(!n) throw "NullPointerException";
     return (n->L ? (n->val <= n->L->val && checkMin(n->L)) : true) &&
          (n->R ? (n->val <= n->R->val && checkMin(n->R)) : true);
   }
};
#endif
```

leftist_heap_boost.cpp – левосторонняя куча с модификацией операции add()

```
#include <queue>
#include <algorithm>
#include <iostream>
#include <list>

#ifndef _LNODE_
#define _LNODE_
template <class T>
struct LNode {
   int d;
   T val;
   LNode <T> * L;
   LNode <T> * R;

   LNode(T _val) {
     d = 1;
     val = _val;
```

```
    L = R = 0;
  }
};
#endif

#ifndef _LHEAP_BOOT_
#define _LHEAP_BOOT_

template <class T>
class LHeapBoot {
  friend class LeftistHeapMergeBootInt;
  LNode<T> * root;
  std::list<T> addition;

public:
  LHeapBoot() {
    root = 0;
  }

  template <class InputIterator>
  LHeapBoot(InputIterator first, InputIterator last) {
    root = 0;
    build(first, last);
  }

  ~LHeapBoot() {
    clear();
  }

  template <class InputIterator>
  void build(InputIterator first, InputIterator last) { // O(n)
    clear();
    std::queue<LNode<T> *> ptrs;

    while(first != last)
      ptrs.push(new LNode<T>(*first++));

    LNode<T> * a;
    LNode<T> * b;
    while(ptrs.size() > 1){
      a = ptrs.front();
      ptrs.pop();
      b = ptrs.front();
      ptrs.pop();
      ptrs.push(merge(a,b));
    }

    root = ptrs.front();
  }

  void mergeWithHeap(LHeapBoot<T>& h){
    addition.splice(addition.end(), h.addition);
    root = merge(root, h.root);
    h.root = 0;
  }

  void print() {
    mergeAddition();
    printSubtree(root,1);
  }

  bool checkMin(){
    mergeAddition();
```

```
    return checkMin(root);
  }

  void clear() {
    addition.clear();
    if(root)
      clear(root);
    root = nullptr;
  }

  void add (T val) {
    if(!root)
      root = new LNode<T>(val);
    else
      if(val < root->val){
        addition.push_back(root->val);
        root->val = val;
      }
      else
        addition.push_back(val);
  }

  T min (){
    return root->val;
  }

  void pop(){
    if(!addition.empty())
      mergeAddition();
    LNode<T> * oldRoot = root;
    root = merge(oldRoot->L, oldRoot->R);
    delete oldRoot;
  }

  bool empty() {
    return root == nullptr;
  }


private:

  void clear(LNode<T> * n) {
    if(n->L) clear(n->L);
    if(n->R) clear(n->R);
    delete n;
  }

  LNode<T> * merge (LNode<T> * a, LNode<T> * b) {  // returns pointer to the
head of merged heap
                                                   // if only one of nodes NOTNULL,
returns it
//   if(!addition.empty())
//     mergeAddition();
    if(!a || !b || a == b){
      if(a && !b) return a;
      if(a == b && a != 0) throw "attempt of self merge";
      return b;
    }
    if(a->val > b->val)
      std::swap(a, b);
    if(!a->R)
      a->R = b;
    else
```

```
      a->R = merge (a->R, b);
    if(!a->L || a->R->d > a->L->d)
      std::swap(a->R, a->L);
    a->d = a->R ? a->R->d + 1 : 1;
    return a;
  }

  void mergeAddition(){
    LHeapBoot<T> h(addition.begin(), addition.end());
    addition.clear();
    root = merge(root, h.root);
    h.root = 0;
  }

  void printSubtree(LNode<T>* n, int level) {
    if(!n)
      std::cout << std::string(level, ' ') << "-:" << (n? n->d : 0) <<
std::endl;
    else{
      printSubtree(n->R, level+1);
      std::cout << std::string(level, ' ') << (n? n->val : 0) << ':' << (n?
n->d : 0) << std::endl;
      printSubtree(n->L, level+1);
    }
  }

  bool checkMin(LNode<T>* n){
    if(!n) throw "NullPointerException";
    return (n->L ? (n->val <= n->L->val && checkMin(n->L)) : true) &&
        (n->R ? (n->val <= n->R->val && checkMin(n->R)) : true);
  }


};

#endif
```

leftist_heap_merge_boost_int.cpp – левосторонняя куча с модификацией операций add() и merge()

```
#include "..\\leftist_heap_bootstrapped\\leftist_heap_bootstrapped.h"
#include <algorithm>
#include <climits>

#ifndef _LHEAP_MERGE_BOOT_
#define _LHEAP_MERGE_BOOT_

struct LeftistHeapMergeBootIntNode {
  int min;
  LHeapBoot <LeftistHeapMergeBootIntNode> * subheap;

  LeftistHeapMergeBootIntNode(){
    subheap = nullptr;
  }
```

```
  LeftistHeapMergeBootIntNode(int _min, LHeapBoot
<LeftistHeapMergeBootIntNode> * _subheap = nullptr){
    min = _min;
    subheap = _subheap;
    _subheap = nullptr;
  };
};

inline bool operator <(const LeftistHeapMergeBootIntNode &a, const
LeftistHeapMergeBootIntNode &b){
  return a.min < b.min;
};

inline bool operator >(const LeftistHeapMergeBootIntNode &a, const
LeftistHeapMergeBootIntNode &b){
  return a.min > b.min;
};


class LeftistHeapMergeBootInt {
  LeftistHeapMergeBootIntNode * root;

public:
  LeftistHeapMergeBootInt(){
    root = nullptr;
  }

  ~LeftistHeapMergeBootInt(){
    clear();
  }

  void add(int _val){
    root = merge(root, new LeftistHeapMergeBootIntNode (_val, nullptr));
  }

  void mergeWithHeap(LeftistHeapMergeBootInt & h){
    root = merge(root, h.root); //ok if arg would be null
    h.root = nullptr;
  }

  bool empty(){
    return root == nullptr;
  }

  int min (){   //undefined behavior if !root
    return root->min;
  }

  void pop(){   //undefined behavior if !root
    if(!root->subheap || root->subheap->empty()){
      if(root->subheap){
        delete root->subheap;
        root->subheap = nullptr;
      }
      delete root;
      root = nullptr;
      return;
    }
    LeftistHeapMergeBootIntNode new_min = root->subheap->min();
    root->min = new_min.min;
    root->subheap->pop();
    if(new_min.subheap) root->subheap->mergeWithHeap(*(new_min.subheap));
  }
```

```
  void clear() {
    if(root)
      clear(root, true);
    root = nullptr;
  }

private:
  LeftistHeapMergeBootIntNode * merge(LeftistHeapMergeBootIntNode * a,
LeftistHeapMergeBootIntNode * b){
    if(!a || !b || a == b){
      if(a && !b) return a;
      if(a == b && a != 0) throw "attempt of self merge";
      return b;
    }
    if(a->min > b->min)
      std::swap(a,b);
    if(a->subheap)
      a->subheap->add(*b);   //copied
    else{
      a->subheap = new LHeapBoot<LeftistHeapMergeBootIntNode>;
      a->subheap->add(*b);   //copied
    }
    if(b) delete b;
    return a;
  }

  void clear(LeftistHeapMergeBootIntNode * n, bool to_del){
    if(n->subheap) {
      if(!n->subheap->addition.empty()){
        for (auto nn : n->subheap->addition)
          clear(&nn, false);
        n->subheap->addition.clear();
      }
      if(n->subheap->root){
        clear(n->subheap->root);
        n->subheap->root = nullptr; //?
      }

      delete n->subheap;
      n->subheap = nullptr;
    }
    if(to_del)delete n;
  }

  void clear(LNode<LeftistHeapMergeBootIntNode>* n){
    clear(&n->val, false);
    if(n->L) clear(n->L);
    if(n->R) clear(n->R);
    delete n;
  }
};

#endif
```

`test_wa.cpp` – тест на корректность

на примере базовой левосторонней кучи

```
#include "leftist_heap.h"

 #include <vector>
 #include <cassert>
 #include <climits>


#define forn(i, n) for (int i = 0; i < (int)(n); i++)

void gen( int n, std::vector <int> &a ) {
  a.resize(n);
  forn(i, n)
    a[i] = rand() % 100;
}

LHeap<int> h;

int main() {
  std::vector <int> x, ns(4);
  ns[0]=3;  ns[1]=5;  ns[2]=10; ns[3]=20;
  for (auto n : ns) // size of test
    forn(t, 10000) { // number of tests

      // test build + min
      gen(n, x);
      h.build(x.begin(),x.end());
      //if(!t) h.print();
      forn(i, n) {
        auto it = min_element(x.begin(), x.end());
        assert(h.min() == *it);
        h.pop();
        x.erase(it);
      }

     // test add + min
      gen(n, x);
      forn(i, n)
        h.add(x[i]);
      forn(i, n) {
        auto it = min_element(x.begin(), x.end());
        assert(h.min() == *it);
        h.pop();
        x.erase(it);
      }
    }
  fprintf(stderr, "Test for WA: OK.\n");
}
```

test_tl_compare.cpp – тест на сравнение времени работы

```
#include "../leftist_heap/leftist_heap.h"
#include "../leftist_heap_bootstrapped/leftist_heap_bootstrapped.h"
#include
"../leftist_heap_merge_bootstrapped_int/leftist_heap_merge_bootstrapped_int.h
"
#include "../skew_heap/skew_heap.h"
#include "../skew_heap_bootstrapped/skew_heap_bootstrapped.h"
#include
"../skew_heap_merge_bootstrapped_int/skew_heap_merge_bootstrapped_int.h"
#include "../test_param.h"
#include <climits>
#include <ctime>
#include <iostream>
#include <functional>
#include <queue>
#include <set>

#define forn(i, n) for (int i = 0; i < (int)(n); i++)

unsigned R() { return (rand() << 15) | rand(); } // mingw g++ has 15-bit
rand()

void gen( int n, std::vector <int> &a ) {
  a.resize(n);
  forn(i, n)
    a[i] = R();
}

#define timeStamp(...) fprintf(stderr, __VA_ARGS__, (clock() - start) /
CLOCKS_PER_SEC), start = clock()
// #define timeStamp(...) fprintf(stderr, __VA_ARGS__, clock() - start),
start = clock()

int main() {
  // int n = HEAP_SIZE;
  // int k = HEAPS_NUM;

  const int rounds          = 10;
  const int memoryInit      = (int)1e4;
  const int memoryCapacity  = (int)5e6;
  const int memoryMult      = 2;

  double start = 0;

  forn(r,rounds){
    fprintf(stderr, "\nr = %2d\t",r+1);
    fprintf(stdout, "\nr = %2d\t",r+1);

    forn(b,2){
      int n = 2; //heap size;
      int k = 2; //heaps num;
      int * p = b ? &n : &k;

      for(*p = memoryInit; *p < memoryCapacity+1; *p *= memoryMult){
        // fprintf(stderr,
"============================================================\n");
        fprintf(stderr, "k = %8d\tn = %8d\t",k,n);
        fprintf(stdout, "\n\tk = %8d\tn = %8d\t",k,n);
```

```
        std::vector <std::vector <int>> x(k);
        std::vector <LHeap<unsigned>>          hl(k);
        std::vector <LHeapBoot<unsigned>>      hla(k);
        std::vector <LeftistHeapMergeBootInt> hlam(k);
        std::vector <SHeap<unsigned>>          hs(k);
        std::vector <SHeapBoot<unsigned>>      hsa(k);
        std::vector <SkewHeapMergeBootInt>     hsam(k);
        std::vector< std::priority_queue<int, std::vector<int>,
std::greater<int>>> q(k);
        std::vector<std::multiset<int>>        s(k);

        forn(i,k)
          gen(n,x[i]);
        start = clock();

        // ---------------------------------------- //
        // fprintf(stderr, "\n\nleftist\n\n");
        fprintf(stderr, "|\tlh\t");

        forn(i,k)
          forn(j, n)
            hl[i].add(x[i][j]);
        // timeStamp("k heaps builded: \t%.2f\n");
        timeStamp("%.2f\t");

        for(int i = 1; i < k; ++i)
          hl[0].mergeWithHeap(hl[i]);
        // timeStamp("k heaps merged: \t%.2f\n");
        timeStamp("%.2f\t");

        hl[0].pop();
        // timeStamp("heap firts acc: \t%.2f\n");
        timeStamp("%.2f\t");

        while (!hl[0].empty())
          hl[0].pop();
        // timeStamp("heap extracted: \t%.2f\n");
        timeStamp("%.2f\t");

        forn(i,k)
          hl[i].clear();
        start = clock();

      // ---------------------------------------- //
        // fprintf(stderr, "\n\nleftist add\n\n");
        fprintf(stderr, "|\tlah\t");

        forn(i,k)
          forn(j, n)
            hla[i].add(x[i][j]);
        // timeStamp("k heaps builded: \t%.2f\n");
        timeStamp("%.2f\t");

        for(int i = 1; i < k; ++i)
          hla[0].mergeWithHeap(hla[i]);
        // timeStamp("k heaps merged: \t%.2f\n");
        timeStamp("%.2f\t");

        hla[0].pop();
        // timeStamp("heap firts acc: \t%.2f\n");
        timeStamp("%.2f\t");

        while (!hla[0].empty())
```

```
    hla[0].pop();
  // timeStamp("heap extracted: \t%.2f\n");
  timeStamp("%.2f\t");

  forn(i,k)
    hla[i].clear();
  start = clock();

// ---------------------------------------- //
  // fprintf(stderr, "\n\nleftist add merge\n\n");
  fprintf(stderr, "|\tlamh\t");

  forn(i,k)
    forn(j, n)
      hlam[i].add(x[i][j]);
  // timeStamp("k heaps builded: \t%.2f\n");
  timeStamp("%.2f\t");

  for(int i = 1; i < k; ++i)
    hlam[0].mergeWithHeap(hlam[i]);
  // timeStamp("k heaps merged: \t%.2f\n");
  timeStamp("%.2f\t");

  hlam[0].pop();
  // timeStamp("heap firts acc: \t%.2f\n");
  timeStamp("%.2f\t");

  while (!hlam[0].empty())
    hlam[0].pop();
  // timeStamp("heap extracted: \t%.2f\n");
  timeStamp("%.2f\t");

  forn(i,k)
    hlam[i].clear();
  start = clock();

// ---------------------------------------- //
  // fprintf(stderr, "\n\nskew\n\n");
  fprintf(stderr, "|\tsh\t");

  forn(i,k)
    forn(j, n)
      hs[i].add(x[i][j]);
  // timeStamp("k heaps builded: \t%.2f\n");
  timeStamp("%.2f\t");

  for(int i = 1; i < k; ++i)
    hs[0].mergeWithHeap(hs[i]);
  // timeStamp("k heaps merged: \t%.2f\n");
  timeStamp("%.2f\t");

  hs[0].pop();
  // timeStamp("heap firts acc: \t%.2f\n");
  timeStamp("%.2f\t");

  while (!hs[0].empty())
    hs[0].pop();
  // timeStamp("heap extracted: \t%.2f\n");
  timeStamp("%.2f\t");

  forn(i,k)
    hs[i].clear();
  start = clock();
```

```
// ---------------------------------------- //
  // fprintf(stderr, "\n\nskew add\n\n");
  fprintf(stderr, "|\tsah\t");

  forn(i,k)
    forn(j, n)
      hsa[i].add(x[i][j]);
  // timeStamp("k heaps builded: \t%.2f\n");
  timeStamp("%.2f\t");

  for(int i = 1; i < k; ++i)
    hsa[0].mergeWithHeap(hsa[i]);
  // timeStamp("k heaps merged: \t%.2f\n");
  timeStamp("%.2f\t");

  hsa[0].pop();
  // timeStamp("heap firts acc: \t%.2f\n");
  timeStamp("%.2f\t");

  while (!hsa[0].empty())
    hsa[0].pop();
  timeStamp("%.2f\t");
  // timeStamp("heap extracted: \t%.2f\n");

  forn(i,k)
    hsa[i].clear();
  start = clock();

// ---------------------------------------- //
  // fprintf(stderr, "\n\nskew add merge\n\n");
  fprintf(stderr, "|\tsamh\t");

  forn(i,k)
    forn(j, n)
      hsam[i].add(x[i][j]);
  // timeStamp("k heaps builded: \t%.2f\n");
  timeStamp("%.2f\t");

  for(int i = 1; i < k; ++i)
    hsam[0].mergeWithHeap(hsam[i]);
  // timeStamp("k heaps merged: \t%.2f\n");
  timeStamp("%.2f\t");

  hsam[0].pop();
  // timeStamp("heap firts acc: \t%.2f\n");
  timeStamp("%.2f\t");

  while (!hsam[0].empty())
    hsam[0].pop();
  // timeStamp("heap extracted: \t%.2f\n");
  timeStamp("%.2f\t");

  forn(i,k)
    hsam[i].clear();
  start = clock();

// ---------------------------------------- //
  // fprintf(stderr, "\n\npriority queue\n\n");
  fprintf(stderr, "|\tqu\t");

  forn(i, k)
    forn(j, n)
```

```
            q[i].push(x[i][j]);
          // timeStamp("k queues builded: \t%.2f\n");
          timeStamp("%.2f\t");

          for(int i = 1; i < k; ++i)
            forn(j, n){
              q[0].push(q[i].top());
              q[i].pop();
            }
          // timeStamp("k queues merged: \t%.2f\n");
          timeStamp("%.2f\t");

          q[0].pop();
          // timeStamp("queue firts acc: \t%.2f\n");
          timeStamp("%.2f\t");

          while (!q[0].empty())
            q[0].pop();
          // timeStamp("queue extracted: \t%.2f\n");
          timeStamp("%.2f\t");

        // ---------------------------------------- //
          // fprintf(stderr, "\n\nset\n\n");
          fprintf(stderr, "|\tset\t");

          forn(i, k)
            forn(j, n)
              s[i].insert(x[i][j]);
          // timeStamp("k sets builded: \t%.2f\n");
          timeStamp("%.2f\t");

          for(int i = 1; i < k; ++i)
            forn(j, n){
              s[0].insert(*(s[i].begin()));
              s[i].erase(s[i].begin());
            }
          // timeStamp("k sets merged: \t\t%.2f\n");
          timeStamp("%.2f\t");

          s[0].erase(s[0].begin());
          // timeStamp("set firts acc: \t\t%.2f\n");
          timeStamp("%.2f\t");

          while (!s[0].empty())
          s[0].erase(s[0].begin());
          // timeStamp("set extracted: \t\t%.2f\n");
          timeStamp("%.2f\t");


          fprintf(stderr, "|\t");
        }
      }
    }
  }
```