Лабораторная работа №7
Динамические объекты сложной структуры
Разработка, отладка и выполнение программы с использованием динамических объектов сложной структуры

Задание 1.

Создать класс **(не использовать шаблоны STL, boost)**, реализующий методы работы с очередью. Принципы организации объекта взять из лабораторной работы №2. Написать программу, иллюстрирующую работу всех методов работы с очередью. Результат формирования и преобразования очереди отображать в компонентах ListBox или его аналогах. Написать обработчик события, реализующий вызов метода решения своего варианта.

Индивидуальные задания

1. Создать очередь из случайных целых чисел. Найти минимальный элемент и сделать его первым.
2. Создать две очереди из случайных целых чисел. В первой найти максимальный элемент и за ним вставить элементы второй очереди.
3. Создать двухсвязную очередь из случайных целых чисел. Удалить из очереди все элементы, находящиеся между максимальным и минимальным.
4. Упорядочить элементы двухсвязной очереди случайных целых чисел в порядке возрастания методом «пузырька», когда можно переставлять местами только два соседних элемента.
5. Представить текст программы в виде двухсвязной очереди. Задать номера начальной и конечной строк. Этот блок строк следует переместить в заданное место очереди.
6. Создать двухсвязную очередь из случайных целых чисел. Удалить все отрицательные элементы очереди.
7. Создать двухсвязную очередь из случайных целых чисел. Из элементов, расположенных между максимальным и минимальным, создать первое кольцо. Остальные элементы должны составить второе кольцо.
8. Создать двухсвязную очередь из случайных целых, положительных и отрицательных чисел. Из этой очереди образовать две очереди, первая из которых должен содержать отрицательные числа, а вторая – положительные. Элементы очередей не должны перемещаться в памяти.
9. Создать двухсвязную очередь из строк программы. Преобразовать очередь в кольцо. Организовать видимую в компоненте Memo или его аналоге циклическую прокрутку текста программы.

10. Создать две двухсвязные очереди из случайных целых чисел. Вместо элементов первой очереди, заключенных между максимальным и минимальным, вставить вторую очередь.
11. Создать двухсвязную очередь из случайных целых чисел. Удалить из очереди элементы с повторяющимися более одного раза значениями.
12. Создать двухсвязную очередь и поменять в ней элементы с максимальным и минимальным значениями, при этом элементы не должны перемещаться в памяти.
13. Создать двухсвязную очередь из нарисованных вами картинок. Преобразовать её в кольцо и организовать его циклический просмотр в компоненте Image или его аналоге.
14. Создать двухсвязную очередь из случайных чисел. Преобразовать её в кольцо. Предусмотреть возможность движения по кольцу в обе стороны с отображением места положения текущего элемента с помощью компоненты Gauge или их аналогах и числового значения – с помощью Label или его аналога.
15. Создать двухсвязную очередь из текста вашей программы и отобразить его в ListBox или его аналоге. Выделить в ListBox или его аналоге часть строк и обеспечить запоминание этих строк. Далее выделить любую строку и нажать кнопку, которая должна обеспечивать перемещения выделенных ранее строк перед текущей строкой. При этом в ListBox или его аналоге должны отображаться строки из двухсвязной очереди.

Задание 2.

Исходная информация в виде массива находится в компоненте StringGrid или его аналоге. Каждый элемент массива содержит строку текста и целочисленный ключ (например, Ф.И.О. и номер паспорта).

Разработать класс для работы с деревом поиска, содержащий следующие методы:

внести информацию из массива в дерево поиска;

1. сбалансировать дерево поиска;
2. добавить в дерево поиска новую запись;
3. по заданному ключу найти информацию в дереве поиска и отобразить ее;
4. удалить из дерева поиска информацию с заданным ключом;
5. распечатать информацию прямым, обратным обходом и в порядке возрастания ключа.

На основе родительского класса создать производный класс для решения задачи выбранного варианта.

Написать программу, иллюстрирующую все методы работы с деревом поиска. Результат формирования и преобразования дерева отображать в компонентах TreeView, Memo или аналогах. Написать обработчик события, реализующий работу с методом решения своего варианта.

Индивидуальные задания

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.
2. Подсчитать число листьев в дереве. (Лист – это узел, из которого нет ссылок на другие узлы дерева.)
3. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.
4. Определить максимальную глубину дерева, т.е. число узлов в самом длинном пути от корня дерева до листьев.
5. Определить число узлов на каждом уровне дерева.
6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.
7. Определить количество символов во всех строках, находящихся в узлах дерева.
8. Определить число листьев на каждом уровне дерева.
9. Определить число узлов в дереве, в которых есть указатель только на одну ветвь.
10. Определить число узлов в дереве, у которых есть две дочери.
11. Определить количество записей в дереве, начинающихся с определенной буквы (например а).
12. Найти среднее значение всех ключей дерева и найти узел, имеющий ближайший к этому значению ключ.
13. Найти запись с ключом, ближайшим к среднему значению между максимальным и минимальным значениями ключей.
14. Определить количество узлов в левой ветви дерева.
15. Определить количество узлов в правой ветви дерева.

Задание 3.

Разработать приложение, в котором содержатся следующие классы:
Родительский класс, реализующий методы работы с хеш-таблицей на основе массива стеков **(не использовать шаблоны STL и boost)**.
Производный класс, созданный на базе родительского и реализующий метод решения своего варианта.
В приложении продемонстрировать работу всех методов работы с хеш-таблицей. Результат формирования и преобразования хеш-таблицы показывать в компоненте Memo методом Print(Memo) или его аналогах в виде строк, отображающих стеки.
Написать обработчик события, реализующий вызов метода решения своего варианта.

Индивидуальные задания

1. Создать хеш-таблицу со случайными целыми ключами в диапазоне от–50 до +50 и преобразовать ее в две таблицы. Первая должна содержать только положительные ключи, а вторая – отрицательные.
2. Создать хеш-таблицу со случайными целыми ключами и удалить из него записи с четными ключами.
3. Создать хеш-таблицу со случайными целыми ключами в диапазоне от –10 до 10 и удалить из него записи с отрицательными ключами.
4. Создать хеш-таблицу со случайными целыми ключами и найти запись с минимальным ключом.
5. Создать хеш-таблицу со случайными целыми ключами и найти запись с максимальным ключом.
6. Подсчитать, сколько элементов хеш-таблицы со случайными ключами превышает среднее значение от всех ключей.
7. Создать хеш-таблицу из случайных целых чисел и найти в ней номер стека, содержащего минимальное значение ключа.
8. Создать хеш-таблицу из случайных целых чисел и найти в ней номер стека, содержащего максимальное значение ключа.

Данных задач достаточно, чтобы защитить лабораторную на минимальную оценку.

Задание 4.

В соответствии со своим вариантом построить ***хеш-таблицы*** с ***открытой*** адресацией размерами 16, 64, 128, 2048 и заполнить с коллизиями. В таблице **h'(key)** – значение хеш-функции, приведшее к коллизии.
Исследовать время поиска в хеш-таблицах. Реализовать оконное приложение, в котором строятся соответствующие графики.

Индивидуальные варианты:

1. Изменить функцию вычисления хеш для решения коллизии на квадратичную функцию, которая строится на основе формулы: $\mathbf{h(key, i) = (h'(key) + c_1*i + c_2*i^2) \bmod hashTableSize}$.
2. Использовать в проекте функцию универсального хеширования.
3. Изменить функцию вычисления хеш на мультипликативную функцию, которая строится на основе формулы: $\mathbf{H(key) = [hashTableSize(key \cdot A \bmod 1)]}$, где $\mathbf{A = (sqrt(5) - 1) / 2 = 0{,}6180339887499}$.

4. Использовать в проекте функции универсального хеширования и модульного. Сравнить количество коллизий при введении одинаковых ключей.
5. Использовать в проекте линейный алгоритм вычисления последовательности испробованных мест.
6. Использовать в проекте функции мультипликативного и модульного хеширования. Сравнить время поиска информации.
7. Изменить функцию вычисления хеш на универсальную.
8. Изменить проект с тем, чтобы использовался аддитивный метод хеширования (ключи должны быть строковыми данными).
9. Изменить хеш-функцию в проекте на модульную.
10. Изменить функцию вычисления хеш для решения коллизии на линейную функцию, которая строится на основе формулы: **h(key, i) = (h'(key) + i)·mod hashTableSize**.
11. Реализовать хеш-таблицу с открытой адресацией для хранения строк. Таблица должна увеличивать свой размер вдвое при достижении 80% заполнения.
12. Реализовать динамическую хеш-таблицу с открытой адресацией для хранения строк. Таблица должна увеличивать свой размер вдвое при достижении 50% заполнения.
13. Использовать в проекте функции мультипликативного и модульного хеширования. Сравнить количество коллизий при введении одинаковых ключей.
14. Использовать в проекте функции универсального хеширования и модульного. Сравнить время поиска информации.
15. Изменить функцию вычисления хеш для решения коллизии на функцию, которая строится на основе формулы: **h(key, i) = (h1(key) + i·h2(key))·mod hashTableSize**, где **h1 (key) = key·mod hashTableSize** , **h2(key) = 1 + (key·mod hashTableSize)**.
16. Реализовать хеш-таблицу с открытой адресацией для хранения строк. Таблица должна увеличивать свой размер втрое при достижении 70% заполнения.

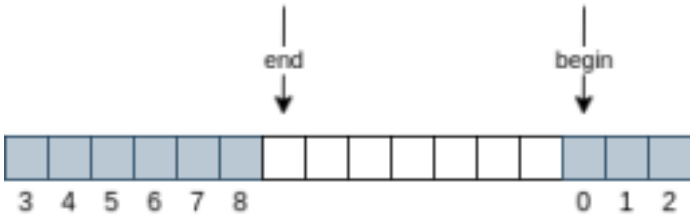
Задание 5. Deque

Deque (double-ended queue) - индексируемая двусвязная очередь, поддерживающее следующие операции, каждое из которых работает за константу:
• *push_back(x)* - добавляет *x* в конец очереди.
• *push_front(x)* - добавляет *x* в начало очереди.
• *pop_back()* - удаляет последний элемент из очереди.
•*pop_front()* - удаляет первый элемент из очереди.
• *random access* индексирование.
Простейший *deque<T>* (здесь и далее за *T* будем считать тип данных, с которым позволяет работать контейнер deque) представляет из себя некоторый массив типа *T* размера *capacity*, из которых задействовано лишь *size* элементов (размер *deque*).
При добавлении нового элемента в *deque* мы обращаемся к зарезервированным и еще не используемым элементам массива и, при отсутствии таковых, создаем новый массив размера $capacity * k$, после чего можно переместить значения старого массива в новый и, наконец, добавить новый элемент. Такой подход используется и в *vector*, что позволяет обходиться без больших расходов на память.
Итераторы могут инвалидироваться, храня указатели на элементы старого массива. Предложенный далее алгоритм позволит *resize deque* без инвалидации итераторов. Немного о процессе добавления/удаления элементов из обычного *deque*.
В любой момент времени необходимо поддерживать два, скажем так, указателя:

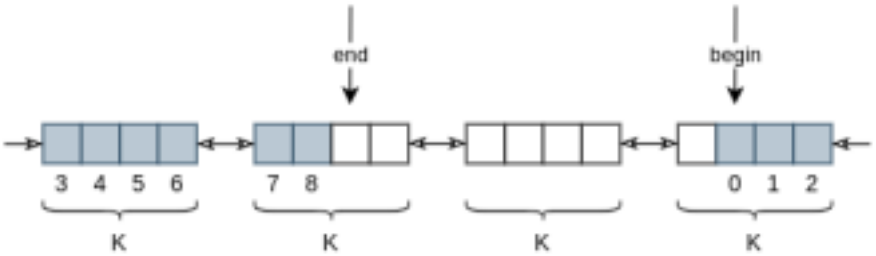
- первый (левый) указывает на начало очереди;
- второй (правый) указывает на следующий элемент после последнего в очереди;

Для того чтоб добавить элемент в конец очереди достаточно положить его в ячейку массива, на которую указывает второй указатель, после чего инкрементировать его.
Для добавления элемента в начало очереди необходимо декрементировать левый указатель и положить туда добавляемый элемент. Делаем каждый раз *resize* при добавлении, когда *capacity == size* и получаем амортизированную сложность O(1).
Удаление - действия, обратные добавлению.

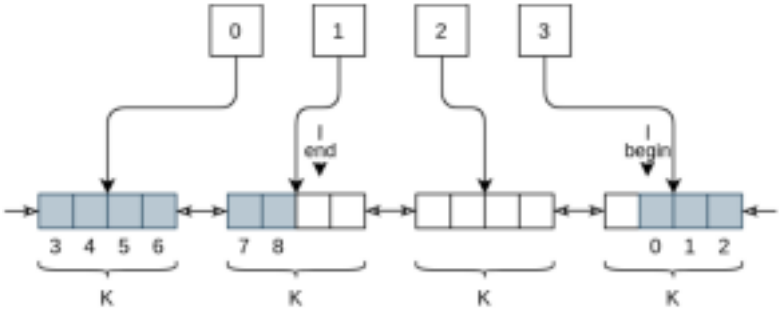
При инкрементировании / декрементировании указателей следует отметить, что первый и последний элементы выделенного массива связаны (т.е. после инкрементирования указателя на последний элемент массива он должен указывать на первый).



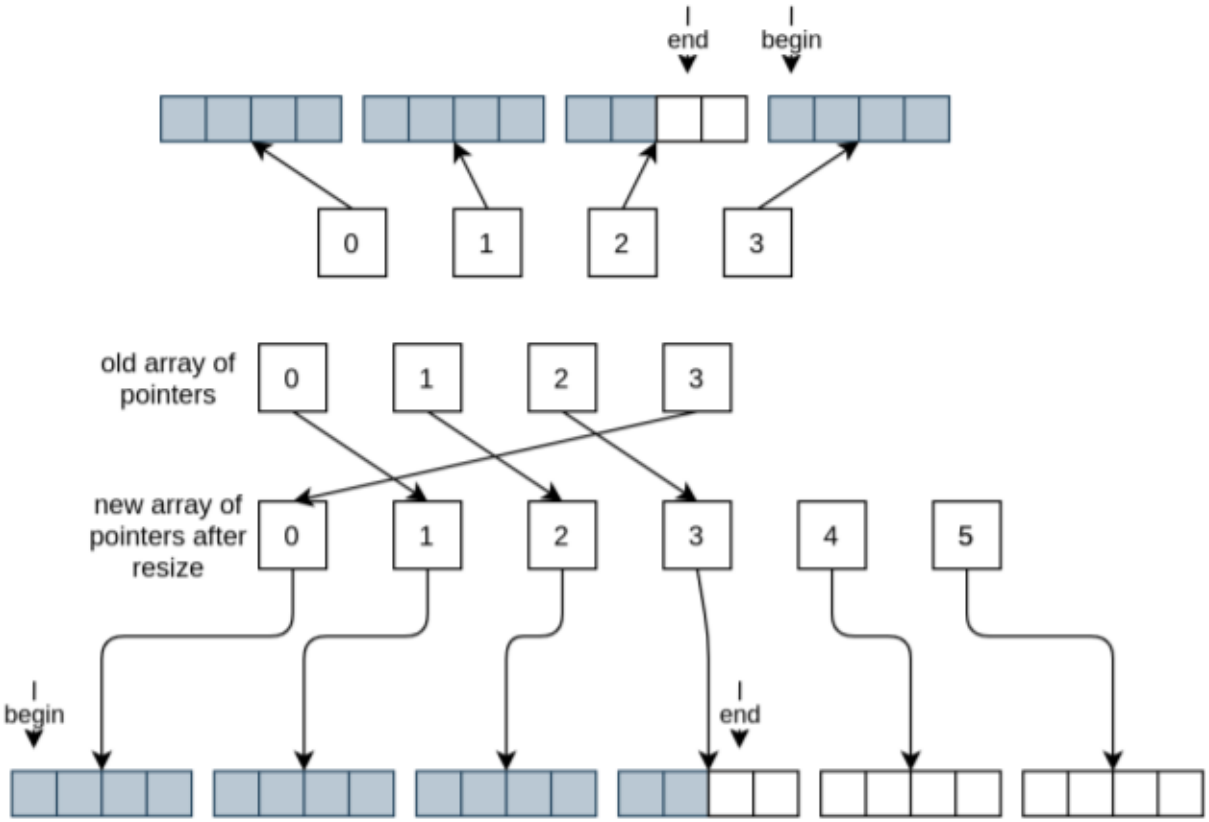
Чтоб получить структуру, где итераторы не инвалидируются, разобьём выделенный массив на блоки фиксированного размера *K*.



Для возможности *random access* индексирования создадим массив размера cap/k (*capacity* делится на *K*), который будет хранить указатели на блоки.



При попытке инкрементировать указатель (на объект), указывающий на последний элемент блока, то указатель в кольце из блоков перемещается на первый элемент следующего блока. Аналогично и при декременте.
resize теперь будет проводиться над массивом указателей.
Может быть ситуация, когда при добавлении необходимо сдвинуть указатель в соседний блок со свободными элементами, однако писать туда будет нельзя, так как при *resize* тот блок, в котором находится первый элемент очереди должен быть под первым указателем нового массива. Иными словами, начальный блок должен оказаться в начале, а конечный в конце. Но если в одном блоке получится, что *end* левее *begin*, то этот блок придется разбить, что тоже приведет к инвалидации итераторов. Потому следует наложить запрет на то, чтоб в одном блоке *end* был левее *begin*.



Необходимо реализовать описанную структуру данных со следующими свойствами и продемонстрировать её работу при помощи визуальных компонентов:

• *push_back* - O(1)
• *push_front* - O(1)
• *pop_back* - O(1)
• *pop_front* - O(1)
• *clear*
• *size*
• *empty*
• *random access iterator* без инвалидации при *resize*, который дополнительно необходимо реализовать.