|  |  |
| --- | --- |
|  | **Министерство образования и науки Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение высшего образования**  **«Московский государственный технический университет имени Н.Э. Баумана**  **(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

**ЛАБОРАТОРНАЯ РАБОТА №6.**

Студент Алахов Андрей Германович  *фамилия, имя, отчество*

Группа ИУ7-32Б

Студент Алахов А. Г.

*подпись, дата фамилия, и.о.*

Проверяющий

*подпись, дата фамилия, и.о.*

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 1. Описание условия задачи

Вариант 2:

Построить ДДП, в вершинах которого находятся слова из текстового файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из слов текстового файла. Использовать метод цепочек для устранения коллизий. Осуществить поиск введенного слова в ДДП, в сбалансированном дереве, в хеш-таблице и в файле. Сравнить время поиска, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного (вводить), то произвести реструктуризацию таблицы, выбрав другую функцию.

# 2. Техническое задание

1. Описание исходных данных

Все входные данные являются читаются в виде строк и переводятся в числа. В цикле вводятся с клавиатуру числа от 0 до 5, которые определяют дальнейшие поведение программы.

Формат ввода:

Возможные пункты меню, которые можно выбрать, введя число с клавиатуры:

1 – работа с бинарным деревом

2 – работа с АВЛ деревом

3 – работа с файлом

4 – работа с хеш-таблицей

5 – вывод информации об эффективности алгоритмов

Возможные способы обработки структур данных, описанных выше:

1 – считывание дынных из файла

2 – добавление элемента

3 – удаление элемента поиск элемента

4 – поиск элемента

5 – печать

При работе с файлом не доступен 1-ый пункт, а при работе с хеш-таблицей есть возможность посмотреть среднее кол-во сравнений при поиске

1. Описание результата программы

Результатом работы программы могут являться (в зависимости от введённого действия):

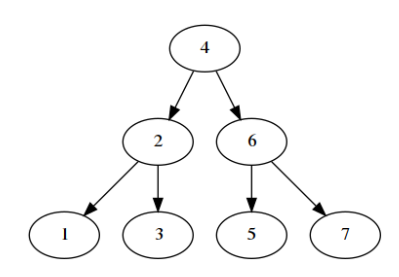
1) Двоичное дерево (его графическая интерпретация)

2) Сбалансированное двоичное дерево (его графическая интерпретация)

3) Хеш-таблица

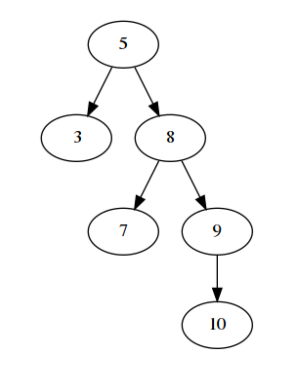
4) Содержимое файла

5) Статистика по времени выполнения операции поиска числа в разных структурах данных с разным содержимым.



Формат вывода:

1) Вывод сбалансированного дерева

2) Вывод сбалансированного дерева

3) Хеш-таблица

| index | data

| 0 | <data> <data> ...

…………………………

4) Статистика выводится в виде таблиц.

| struct type | time add | time search | avg comparisons | memory |

1. Описание задачи, реализуемой программой

Программа выполняет операции добавления, поиска, удаления элемента в структуру данных и печать этой структуры. Также возможна оценка эффективности алгоритма поиска элемента в этих структурах.

1. Способ обращения к программе

Обращение к программе происходит с помощью стандартных потоков ввода и вывода. Вызов программы app.exe

1. Описание возможных аварийных ситуаций и ошибок пользователя

Ошибки операционной системы при выделении динамической памяти, удаление элемента из пустой очереди, добавление элемента в очередь, хранящую максимальное допустимое число элементов.

# 3. Описание внутренних структур данных

Для хранения стека в виде вектора была выбрана данная структура:

struct vect\_fifo

{

void \*q1, \*qm, \*pin, \*pout;

size\_t len, size;

};

q1 — адрес нижней границы очереди. qm – адрес верхней границы очереди. pin – указатель на «хвост» очереди. pout – указатель на «голову» очереди. len — кол-во элементов в очереди. size – размер элемента стека.

Для хранения стека в виде списка были выбраны следующие структуры данных:

struct list

{

void \*value;

struct list \*next;

};

list — узел списка с элементами стека. value — указатель на значение элемента. next — указатель на следующий узел списка.

struct list\_fifo

{

struct list \*pin, \*pout;

size\_t len, size;

};

pin — указатель на «хвост» очереди. pout – указатель на «голову» очереди. len — кол-во элементов в очереди. size – размер элемента стека.

# 4. Алгоритм

Алгоритм моделирования процесса обслуживания:

1. Генерируем время для добавления заявок обоих типов.
2. Проверяем, какая заявка добавится быстрее и помещаем её в обслуживающий аппарат. Вычитаем прошедшее время из другой заявки и генерируем время для добавления новой.
3. В зависимости от того, какое действие произойдёт быстрее всего выполняем один из следующих пунктов:
   1. Добавляем в очередь заявку 1-го типа, если аппарат не занят, помещаем заявку на обработку. Если аппарат обслуживает заявку 2-го типа, помещаем заявку 1-го типа на обработку и помещаем заявку 2-го типа в конец очереди. Если аппарат обслуживает заявку 1-го типа, вычитаем из её времени прошедшее. Вычитаем прошедшее время из другой заявки и генерируем время для добавления новой.
   2. Добавляем в очередь заявку 2-го типа, если аппарат не занят, помещаем заявку на обработку. Вычитаем прошедшее время из другой заявки, а также времени на обработку и генерируем время для добавления новой.
   3. Удаляем элемент, прошедший обработку из соответствующей очереди. Вычитаем прошедшее время из обеих заявок. При наличии элемента в первой очереди помещаем его на обработку. Иначе, при наличии элемента во второй очереди помещаем его на обработку.
4. Проводится статистический подсчет: количество вошедших и вышедших, время работы, кол-во вошедших и вышедших элементов и т. д.
5. Пункты 3-4 повторяются, пока из ОА не выйдет 1000 заявок 1 типа.

**5. Тесты**

Позитивные тесты:

|  |  |  |
| --- | --- | --- |
| Входные данные | Что проверяем | Ожидаемый выходной результат |
| Выбор: смоделировать процесс обслуживания очередью в виде вектора 7 | Проверка моделирования процесса обслуживания | В соответствии с форматом вывода |
| Выбор: смоделировать процесс обслуживания очередью в виде списка 8  Выбор: изменение временных интервалов 1  1 3 2 5 0 2 0,5 | Проверка изменения временных интервалов | В соответствии с форматом вывода, при этом ожидаемое время моделирования должно быть равно 2000, а погрешность не более 3% |
| Начальный стек: 1 2 3 Выбор: добавление элемента в очередь в виде вектора 1  -6  Выбор: вывод состояния очереди в виде вектора 5 | Проверка на добавление элемента в очередь | Очередь: -6 1 2 3 |
| Начальный стек: 1 2 3 Выбор: удаление элемента из очереди в виде вектора 3  Выбор: вывод состояния очереди в виде вектора 5 | Проверка на удаление из очереди | Элемент: 1  Очередь: 2 3 |
| Начальный стек: 1 2 3 Выбор: добавление элемента в очередь в виде списка 2  -6  Выбор: вывод состояния очереди в виде списка 6 | Проверка на добавление элемента в очередь | Очередь: -6 1 2 3 |
| Начальный стек: 1 2 3 Выбор: удаление элемента из очереди в виде списка 4  Выбор: вывод состояния очереди в виде списка 6 | Проверка на удаление из очереди | Элемент: 1  Очередь: 2 3 |

Негативные тесты:

|  |  |  |
| --- | --- | --- |
| Входные данные | Что проверяем | Ожидаемый выходной результат |
| Выбор: удаление элемента из очереди в виде списка | Проверка на взятие элемента из пустой очереди | Deletion from empty list queue |
| Начальный стек: 1 2 3 4 5 Максимальный размер очереди: 5 Выбор: добавление элемента в очередь в виде вектора 1 | Проверка на переполнение очереди | Vect queue overflow |
| Выбор: смоделировать процесс обслуживания очередью в виде списка 8  Выбор: изменение временных интервалов 1  1 4 0 0 1 3 0 1 | Проверка на нулевой максимум | max can't be bigger than min |
| Выбор: смоделировать процесс обслуживания очередью в виде вектора 8  Выбор: изменение временных интервалов 1  4 1 0 3 1 2 0 1 | Проверка на минимум больше максимума | max can't be zero |
| Выбор: смоделировать процесс обслуживания очередью в виде списка 8  Выбор: изменение временных интервалов 1  1 4 0 3 -1 2 0 1 | Проверка на отрицательный минимум | min can't be less than zero |

# 6. Функции

Печать очереди в виде списка

int list\_fifo\_print(struct list\_fifo list\_queue, struct list \*used\_ptr, int (\*f)(struct list\_fifo \*temp));

Добавление элемента в очередь в виде списка

int list\_fifo\_add(struct list\_fifo \*list\_queue, void \*value, struct list \*\*used\_ptr);

Удаление элемента из очереди в виде списка

int list\_fifo\_delete(struct list\_fifo \*list\_queue, void \*value, struct list \*\*used\_ptr);

Освобождение очереди в виде списка

int list\_fifo\_free(struct list\_fifo \*list\_queue);

Аналогичные функции для очереди в виде вектора

int vect\_fifo\_print(struct vect\_fifo vect\_queue, int (\*f)(struct vect\_fifo \*temp));

int vect\_fifo\_add(struct vect\_fifo \*vect\_queue, void \*value);

int vect\_fifo\_get(struct vect\_fifo \*vect\_queue, void \*value);

int vect\_fifo\_free(struct vect\_fifo \*vect\_queue);

Функции для моделирования процесса обслуживания

void list\_operation(double t1\_min, double t1\_max, double t2\_min, double t2\_max, double t3\_min, double t3\_max, double t4\_min, double t4\_max);

void vect\_operation(double t1\_min, double t1\_max, double t2\_min, double t2\_max, double t3\_min, double t3\_max, double t4\_min, double t4\_max);

**7. Оценка эффективности программы.**

Произведем замер по времени и памяти для разного количество элементов в очереди и разного максимального размера очереди в виде вектора.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во элементов в очереди | Максимальное кол-во эл. в очереди в виде вектора | Очередь в виде вектора | | Очередь в виде списка | |
| Время на выполнение операции (в мс) | Память на хранение элементов (в байтах) | Время на выполнение операции (в мс) | Память на хранение элементов (в байтах) |
| 10 | 10 | 0,0003 | 88 | 0,0011 | 232 |
| 20 | 0,0002 | 128 | 0,0010 | 232 |
| 40 | 0,0001 | 208 | 0,0010 | 232 |
| 50 | 0,0001 | 248 | 0,0010 | 232 |
| 100 | 100 | 0,0020 | 448 | 0,0120 | 2032 |
| 200 | 0,0024 | 848 | 0,0125 | 2032 |
| 400 | 0,0020 | 1648 | 0,0120 | 2032 |
| 500 | 0,0020 | 2048 | 0,0120 | 2032 |
| 1000 | 1000 | 0,0195 | 4048 | 0,1318 | 20032 |
| 2000 | 0,0187 | 8048 | 0,1327 | 20032 |
| 4000 | 0,0190 | 16048 | 0,1309 | 20032 |
| 5000 | 0,0186 | 20048 | 0,1316 | 20032 |

Очередь в виде списка всегда проигрывает очереди в виде вектора по времени, так как на каждую операцию добавления и удаления в списке происходит выделение или очистка динамической памяти, в то время как в очереди на основе вектора происходит просто изменение адреса и запись элемента.

Очередь на основе списка для типа занимает 32 байта и каждый новый элемент занимает ещё 16 + size байт, где size – размер элемента. В это время очередь на основе вектора занимает 48 + size \* max байт, где max – максимальное кол-во элементов.  
Таким образом очередь на основе списка будет эффективнее по памяти при выполнении следующего условия: el < (size \* max + 16) / (16 + size), где el – это кол-во элементов в очереди.

Для типа char очередь на основе списка будет гарантированно эффективнее по памяти, если el меньше max в 17 раз.

Для типа int очередь на основе списка будет гарантированно эффективнее по памяти, если el меньше max в 5 раз.

Для типа double очередь на основе списка будет гарантированно эффективнее по памяти, если el меньше max в 3 раза.

При этом эффективность очереди на основе списка можно увеличить, если использовать заранее известный тип данных, а не указатель, как в данной реализации.

# 8. Выводы

Если при решении задачи основная цель — улучшение времени работы программы, то стоит использовать очередь на основе вектора. Если точное количество элементов в очереди заранее не известно, а его размер большой, то в целях экономии памяти может быть целесообразнее использовать очередь на основе списка. Также стоит отметить, что количество элементов в очереди в виде списка ограничено только количеством динамической памяти, которую может выделить операционная система под нашу программу, а не каким-то конкретным числом, как в случае очереди-вектора.  
В ходе тестирования программы была обнаружена фрагментация памяти, что свидетельствует о чередовании свободных и занятых участков памяти.

Ответы на контрольные вопросы:

1. Что такое очередь?

Очередь — это последовательный список с переменной длинной, добавление элементов в который происходит с одной стороны - в хвост, а исключение элементов из которого происходит с другой стороны — с головы. Очередь действует по принципу first in first out.

2. Каким образом и сколько памяти выделяется под хранение очереди при различной его реализации?

При реализации очереди на основе вектора память под нее выделяется сплошным куском при ее создании. При реализации очереди на основе списка память выделяется в процессе ее обработки, то есть при каждом вызове push создается элемент узла списка, а при каждом pop он удаляется. Размер памяти пропорционален количеству элементов в очереди.

3. Каким образом освобождается память при удалении элемента очереди при различной его реализации?

В очереди на основе вектора просто перемещается указатель в векторе, а память очищается, когда очередь уже не нужна. В очереди на основе списка память под элементом каждый раз очищается при снятии этого элемента с очереди.

4. Что происходит с элементами очереди при ее просмотре?

Для того чтобы получить элемент, его надо снять с очереди, то есть после просмотра очереди, она становится пустой

5. Каким образом эффективнее реализовывать очередь. От чего это зависит?

См. «Вывод»

6. В каком случае лучше реализовать очередь посредством указателей, а в каком –массивом?

См. «Вывод»

7. Каковы достоинства и недостатки различных реализаций очереди в зависимости от выполняемых над ней операций?

Недостатки:

• при реализации на основе обычного массива часто приходится сдвигать элементы, что занимает время

• при реализации на основе кольцевого массива количество элементов в очереди ограничено заранее определенным числом

• при реализации на основе списка затрачивается много времени на выделение и очистку памяти при добавлении элемента в очередь и снятии с нее

Достоинства:

• Операции добавления и удаления в очереди на основе вектора работают относительно быстро

• Память для очереди на основе списка теоретически ограничена только размером памяти, которую может выделить операционная система

8. Что такое фрагментация памяти?

Фрагментация — процесс появления незанятых участков в памяти. Вызвана наличием в каждом виде памяти деления на мелкие единицы фиксированного размера, в то время как объем информации необязательно кратен этому делению.

9. На что необходимо обратить внимание при тестировании программы?

На переполнение очереди на основе вектора и на процент расхождения расчетного времени и общего времени моделирования (он должен быть в переделах 2-3 процентов)

10.Каким образом физически выделяется и освобождается память при динамических запросах?

Выделение памяти происходит блоками — непрерывными фрагментами оперативной памяти. В какой-то момент в куче может не оказаться блока подходящего размера и, даже если свободная память достаточна для размещения объекта, операция выделения памяти окончится неудачей