Министерство образования и науки Российской Федерации

Новосибирский государственный технический университет

Кафедра вычислительной техники

Курсовая работа

по дисциплине «Программирование»

Тема: «Шаблон иерархической структуры данных в памяти»

Факультет: АВТФ
Группа: ВТБ-41
Выполнил: Клопов Д. М.
Проверил: Романов Е. Л.

Бердск, 2006г.

ОГЛОВЛЕНИЕ

I. Задание и комментарии разработчика к реализации.

Общая формулировка задачи

Для заданной двухуровневой структуры данных, содержащей указатели на объекты (или сами объекты) - параметры шаблона, разработать полный набор операций (добавление, включение и извлечение по логическому номеру, сортировка, включение с сохранением порядка, загрузка и сохранение строк в текстовом файле, балансировка выравнивание размерностей структур данных нижнего уровня) (см. bk57. rtf). Предполагается, что операции сравнения хранимых объектов переопределены стандартным образом (в виде операций <,> и т.д..). Программа должна использовать шаблонный класс с объектами-строками и реализовывать указанные выше действия над текстом любого объема, загружаемого из файла.

Выбранная тема

Шаблон структуры данных двусвязный циклический список, содержащий статический массив указателей на объекты. Последовательность указателей в каждом массиве ограничена NULL. При переполнении текущего массива указателей создается новый элемент списка, в который переписывается половина указателей из текущего.

Комментарии.

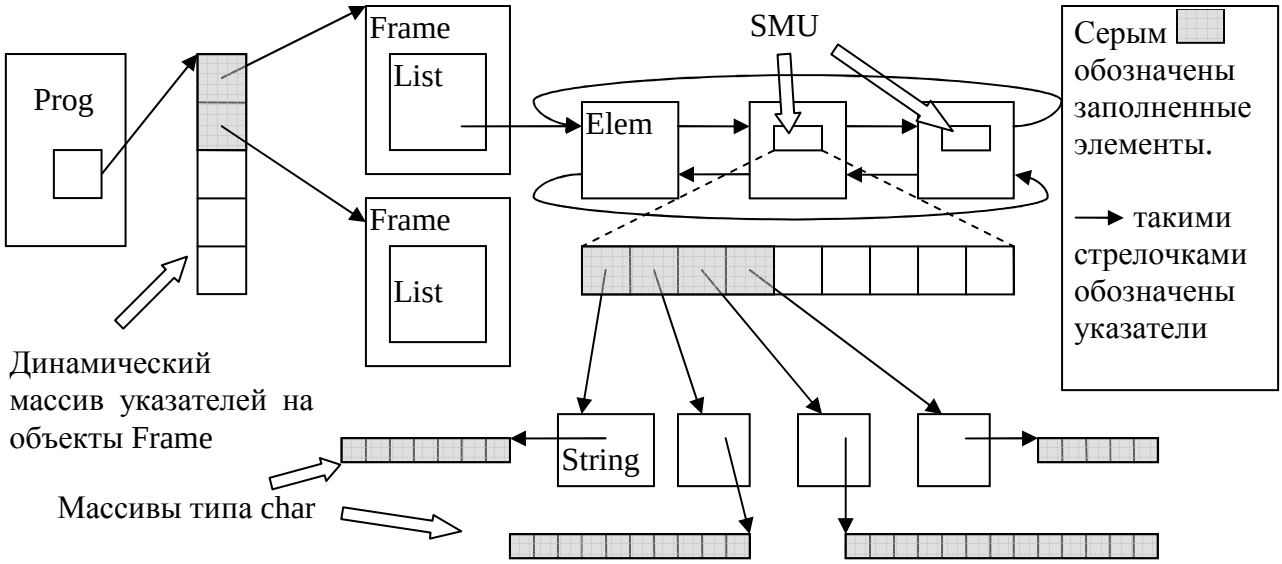
1. В массиве наряду с NULL-ограничителем будет использоваться счетчик указателей. Это упрощает реализации некоторых алгоритмов и в целом ускоряет выполнение отдельных операций. NULL-ограничитель будет присутствовать в естественной форме, так как, во избежание некорректных обращений, все элементы в массиве будут заполнены или указателями или NULL. А также, как показывает практика, при отладке сложно отличить рабочий указатель от того, который просто указывает на произвольную ячейку памяти. Применение NULL хоть и порицается Б. Страуструпом (в целях совместимости версий и еще чего-то), но несколько облегчает анализ программы.
2. При переполнении будет переписываться не половина, а какой-то процент указателей из текущего в новый элемент. Эта инициатива не влияет на суть задания, но позволяет исследовать время выполнения операций при разном критическом проценте. Более подробно об этом написано в пункте «Работа с файлом», там же приведены сравнения скорости выполнения сортировки от этого процента и сделаны соответствующие выводы.
3. Двусвязный циклический список будет представлять собой объект соответствующего класса, в котором будет содержаться указатель на первый элемент списка (указатель на заголовок). То есть список полностью – внешняя динамическая структура. Первый элемент не несет в себе данных. СМУ будет являться частью структурированной переменной – элемента списка. Параметры для шаблона СМУ будут браться из параметров шаблона списка. Иными словами, параметры для шаблона списка – это на самом деле параметры для СМУ.

II. Структурное описание разработки

В программе используются пять классов:

1. String класс строк (на нем держится все, в том числе и оформления меню)
2. SMU<class T, int N> – класс статического массива указателей.
3. List<class T,int N> – класс двусвязного циклического списка. Его элементы базируются на SMU.
4. Frame – класс документа, с которым работает пользователь. Использует LIST в качестве СД и String в качестве элемента данных.
5. Prog – класс программы вообще. Обеспечивает интерфейс между пользователем и Frame. Содержит динамический массив объектов Frame и позволяет работать с несколькими документами одновременно (простое переключение между документами и взаимодействие через буфер обмена).

Схема структуры данных в программе

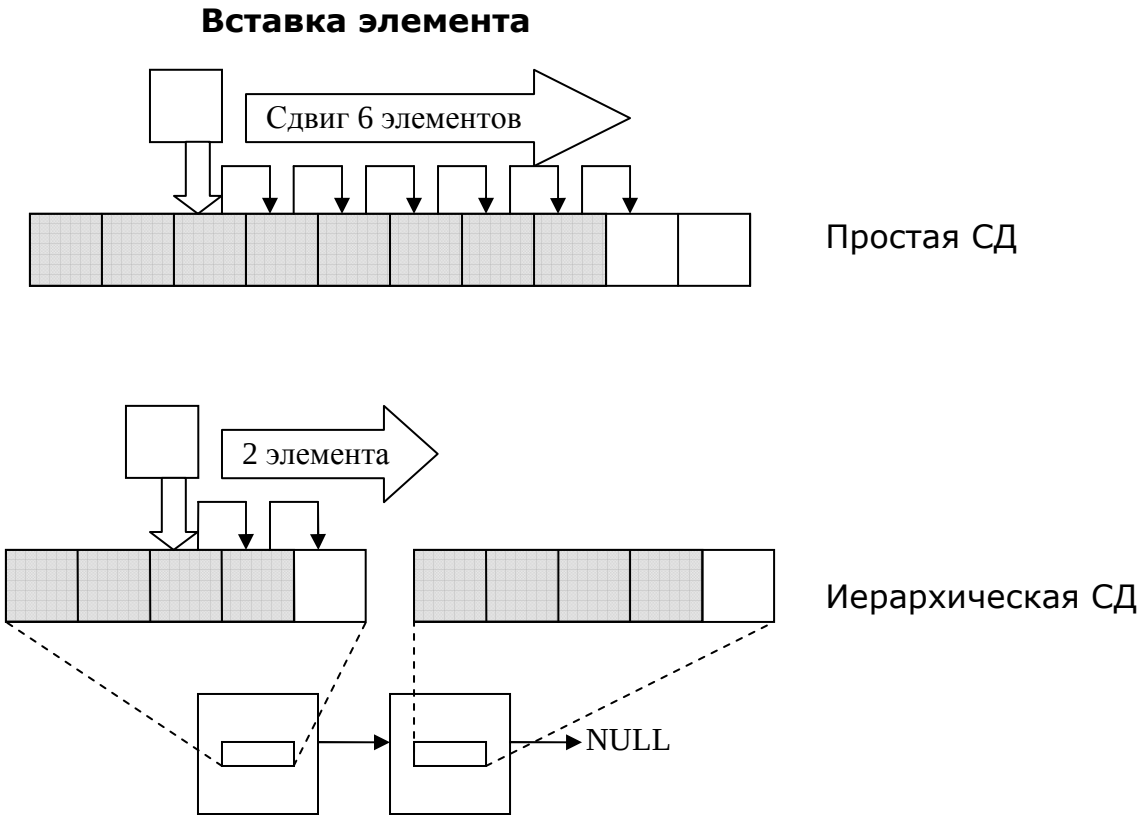


Структура программы при определенной доработке интерфейса позволяет сделать простой текстовый редактор. Особенно интересна возможность работы с несколькими открытыми документами. Буфер позволяет помещать только один объект класса строк, но и операции, выведенные в меню, требуют только одной строки. Если использовать в качестве буфера динамический массив объектов класса строк и разработать интерфейс, позволяющий произвольно выделять текст, то мы получим сносную программу для редактирования файлов.

Немного о структуре данных в программе.

Иерархия структуры данных позволяет при правильном соотношении параметров и типа управляющей структуры и структуры нижнего уровня (структуры с данными) обойти недостатки простых структур, например, массива или списка. Допустим, у нас есть N объектов. И нам нужно как-то структурировано хранить их и обрабатывать. Допустим, мы выбрали простую СД типа массив. Для вставки или исключения необходимо перераспределять данные в массиве, что приводит к большим затратам. Особенно если N большое. Зато есть преимущество прямой адресации при поиске, сортировке или извлечению по логическому номеру. Чтобы избежать больших затрат на перераспределения, можно этот массив разбить на несколько и каждую часть (то есть

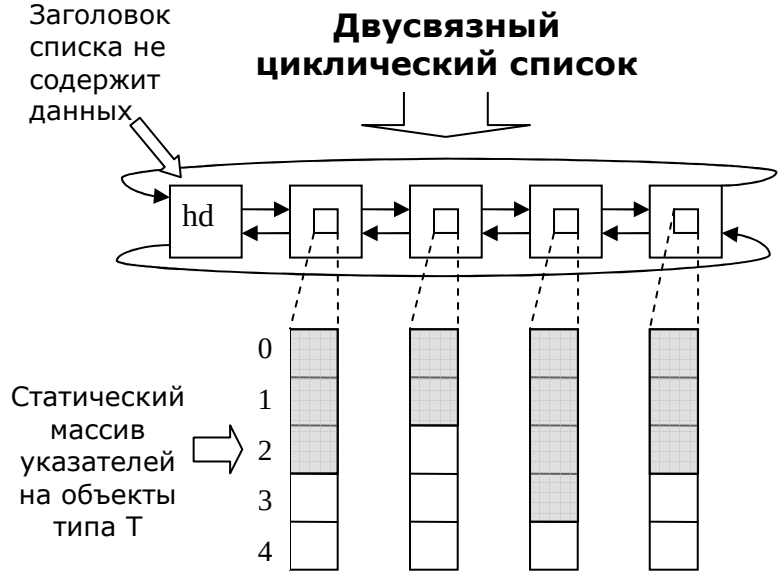
массив) объединить в качестве элементов СД верхнего уровня. Например, в массив или список как на рисунке. Конечно, при использовании списка как управляющей СД увеличится время поиска элемента, но не настолько, как при использовании просто списка. Теперь можно смещать акцент на быстродействия в одних операциях по отношению к другим, изменяя параметры СД. То есть, мы получаем возможность «настроить» СД на те операции, которые будут выполняться чаще всего.



Итак, перейдем от иерархических структур вообще к иерархической структуре в частности. Двусвязный циклический список, элементы которого содержат статический массив указателей. Для реализации такой СД нужно рассмотреть две сущности:

1. Статический массив указателей
2. Двусвязный циклический список

Графически такая иерархия выглядит следующим образом:



Что должен уметь наш статический массив.

В технологии ООП данные не живут отдельно от алгоритма, поэтому массив – это сущность с некоторыми навыками – методологическим набором самообработки. Я попробую раскрыть основные философско-алгоритмические моменты.

Что во мне?

Этот вопрос не волнует, или, по-хорошему, не должен волновать шаблон СМУ. Есть какие-то объекты какого-то типа, на них есть какие-то указатели, какие не важно. Так говорит механизм абстракции. Мы это понимаем, но работать будем только с «прилежными» объектами, для которых переопределены операции сравнения, присваивания, добавления и работы со стандартными потоками.

Сколько этого во мне?

Вопрос не праздный, в задании сказано, что в массиве нуль-ограничитель, но для ускорения работы введем еще счетчик указателей. Зачем по сто раз задавать этот вопрос и искать ответ тотальным пересчетом содержимого? Лучше сделать это один раз и при любых изменениях, касающихся количества хранимых указателей, изменять счетчик. В любом случае «этого» будет не больше размерности массива, которая является параметром шаблона.

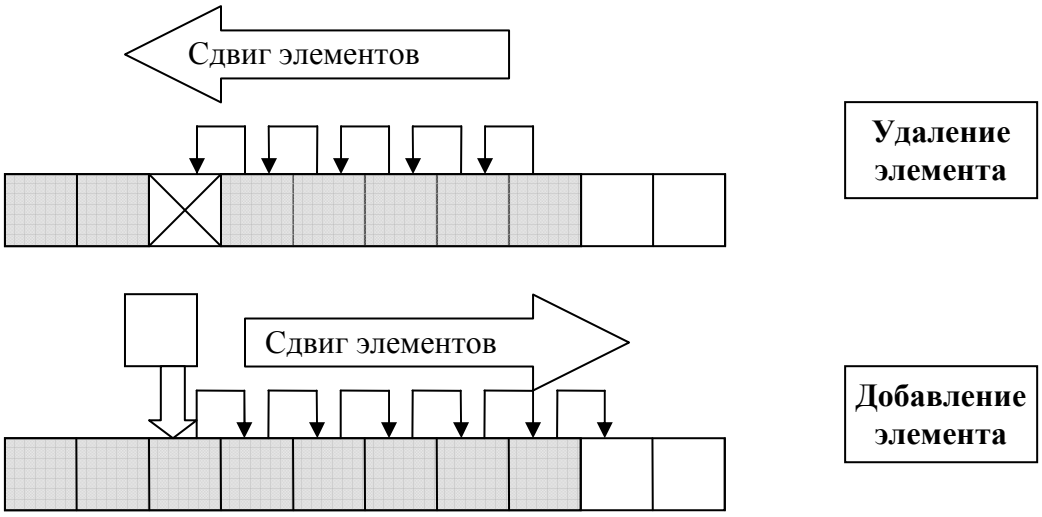
Объект напрокат или все-таки копия.

Поскольку в массиве хранится только указатель на объект, то при добавлении мы получаем только указатель. Но ведь указуемый объект может быть разрушен «вражеским» алгоритмом, внешней функцией, что тогда? А тогда выйдет парадокс: есть указатель в никуда, где соответственно есть ничто. Иными словами, может произойти завал программы. Поэтому логично обзавестись динамической копией объекта, полученного по указателю и поместить в структуру уже указатель на «свой» объект, а что будет с тем – массив не волнует.

По логическому номеру это куда?

Весь мир делится на два типа людей: те, кто начинают считать с единицы, и те, кто начинают считать с нуля. Так повелось, что в языках программирования массивы обычно «зеро бэйзед», то есть, первый элемент имеет индекс 0. Я не хочу пудрить себе и массиву голову и лишний раз отмерять метр от столба, поэтому пусть счет будет начинаться с нуля. Соответственно все остальные элементы, начиная с нашего логического номера при вставке, приобретут индекс на единицу больше, чем был до этого, а при удалении – на единицу меньше, чем был, то есть сдвинутся.
С логическим номером связаны следующие методы:

1. Метод извлечения просто возвращает указатель по логическому номеру.
2. Метод удаления освобождает память по указателю, индекс которого указан во входном параметре и сдвигает все элементы массива.
3. Метод вставки сначала сдвигает элементы в массиве, освобождая нужное место, и затем вставляет на это место полученный из входного параметра указатель.



А я упорядоченный?

Нормальный СМУ должен уметь наводить порядок в себе сам, плюс должен знать, сколько операций (вставки/удаления/замещения) было произведено с момента последнего упорядочивания (величина вроде энтропии). Но в нашем случае все это не требуется, поскольку порядок будет наводить управляющая СД, исходя из всех данных, а не только в нашем массиве. Поэтому достаточно операций извлечения и вставки по логическому номеру, которые будут использовать алгоритмы сверху. Тем более по затратам они не сильно отличаются от таких же операций внутри массива. Так что наш массив не знает и не будет знать упорядочен он или нет.

Меня переполняют…

Допустим, какой-то кривой алгоритм пытается обмануть массив и вставить в него больше, чем входит. Массив не должен на это реагировать, ведь данная ситуация является ошибкой разработчика управляющей СД, поэтому, если действительно вставлять некуда, массив просто не вставляет. Можно выкинуть наверх злобную -1 или устроить интеллектуальную систему обработки ошибок. Но лучше просто игнорировать данное злодеяние, ибо на отчетах об ошибках держится логика неправильной программы. Однако должен быть метод, который сообщает наверх: полный массив или нет, дабы переложить вопросы перераспределения указателей на того, кто выше по иерархии.

Дайте мне свободу

Освободить массив может понадобиться по разным причинам, например в целях оптимизации при копировании. Гораздо дольше удалить текущий объект, потом создать новый такой же и в него что–то поместить, чем очистить текущий и записать то, что нужно. Очистка – это простая операция: освобождаем память по каждому указателю, и вместо них записываем NULL. Массив как новый, ибо при создании все элементы равны NULL. Возможно, это и не часто востребованная операция, но пусть будет.

«Как две капли воды…» © Конструктор Копирования.

Необходимость в конструкторе копирования в данной программе несколько призрачная, но поскольку это все-таки шаблон, и он может быть применен в другом месте, то СМУ должен уметь корректно копировать себе подобного. Для этого нужно очистить содержимое (вот как раз о чем и говорилось в предыдущем пункте) и скопировать все указатели из входного массива. Правда, указатели будут уже другие, и на совсем другую

память, но это уже не важно[1], главное, что по одному и тому же логическому номеру в этих массивах мы получим указатель на одни и те же данные, но в разных областях памяти.

В поисках максимума

При различных сортировках может понадобиться найти максимальный элемент массива. Под максимальным элементом понимается указатель на тот объект, который больше либо равен любому объекту доступному по указателю из массива. Короче, для этого указателя всегда верно:
(*УКАЗАТЕЛЬ)>= (*МАССИВ[индекс любого элемента])
Уф, смысл, думаю, ясен. Алгоритм простой: берем первый элемент и сравниваем со следующим, если следующий больше текущего, делаем его текущим и сравниваем со следующим относительно нового текущего. И так пока элементы не кончатся. В итоге у нас окажется максимальный элемент. Аналогично можно найти и минимальный.
В шаблоне СМУ предусмотрено два типа функций:

1. «Информирующие». Возвращают индекс максимального или минимального элемента.
2. «Действующие». Возвращают указатель на максимальный или минимальный элемент.

Мы же друзья, я солью в твой поток… пару объектов.

Для работы со стандартными потоками существует четыре переопределенные операции (<< , >>) дружественные потокам ofstream, ifstream, istream, ostream. Для работы с файловыми потоками есть два метода. Первый пытается (более-менее корректно) загрузить из потока объекты в память и поместить их указатели на свободные места в массиве. Второй пытается сохранить все объекты, что доступны по указателям из массива. Ввод и вывод осуществляется аналогично.

Доктор, а можно без операции?

Без переопределения операций можно обойтись, но хочется, чтобы шаблон был логически завершенным, поэтому такие операторы как “+”, ”=”, ”<”, ”>”, “<=”, “>=”, “>>”, “<<” все-таки были переопределены. Под “+” понимается добавление в конец, под операциями сравнения – разница в числе элементов сравниваемых массивов.

Его величество, Двусвязный Циклический Список или что может верховный иерарх структуры данных.

Местоположение установлено.

Очень важно знать, где в иерархической СД содержатся нужные данные. Учитывая, что в каждом элементе списка может содержаться разное число указателей, то даже логический номер элемента определить не просто. Для этого нужно просматривать почти всю структуру и подсчитывать число элементов. К сожалению, совсем от таких операций отказаться нельзя. Давайте рассмотрим возможные применения функции поиска элемента по логическому номеру. Безусловно, это функции вставки, извлечения, удаления. Какие же данные нужны этим функциям для выполнения своих прямых обязанностей? Нужен указатель на элемент списка, нужен логический номер в массиве этого элемента списка, и, может быть, сам указатель по этому номеру. Получается, что на вход метода поиска будет поступать логический номер, а на выходе должны быть данные трех разных типов. В такой ситуации логично передавать в метод переменные по ссылке, которые после выполнения поиска окажутся указателем на элемент списка и логическим номером в нем.

[1] См. п. «Объект напрокат или все-таки копия»

В качестве возвращаемого значения будет указатель на объект или NULL, если логический номер некорректный.

Где выскочка?

Поиск максимального элемента – задача достаточно тривиальная. К тому же, класс СМУ умеет находить в себе максимальный и минимальный элемент. Все что требуется – это, переходя от элемента списка к элементу, сравнивать максимальные элементы в их массивах. Таким образом, максимальный элемент будет найден за один линейный просмотр. Поиск наибольшего или наименьшего элемента – это одна из немногих задач, что имеют линейную сложность и не поддаются алгоритмической оптимизации. В то же время это одна из часто используемых функций. Все, что можно сделать – это оптимизировать (точнее не усугублять) на уровне реализации.

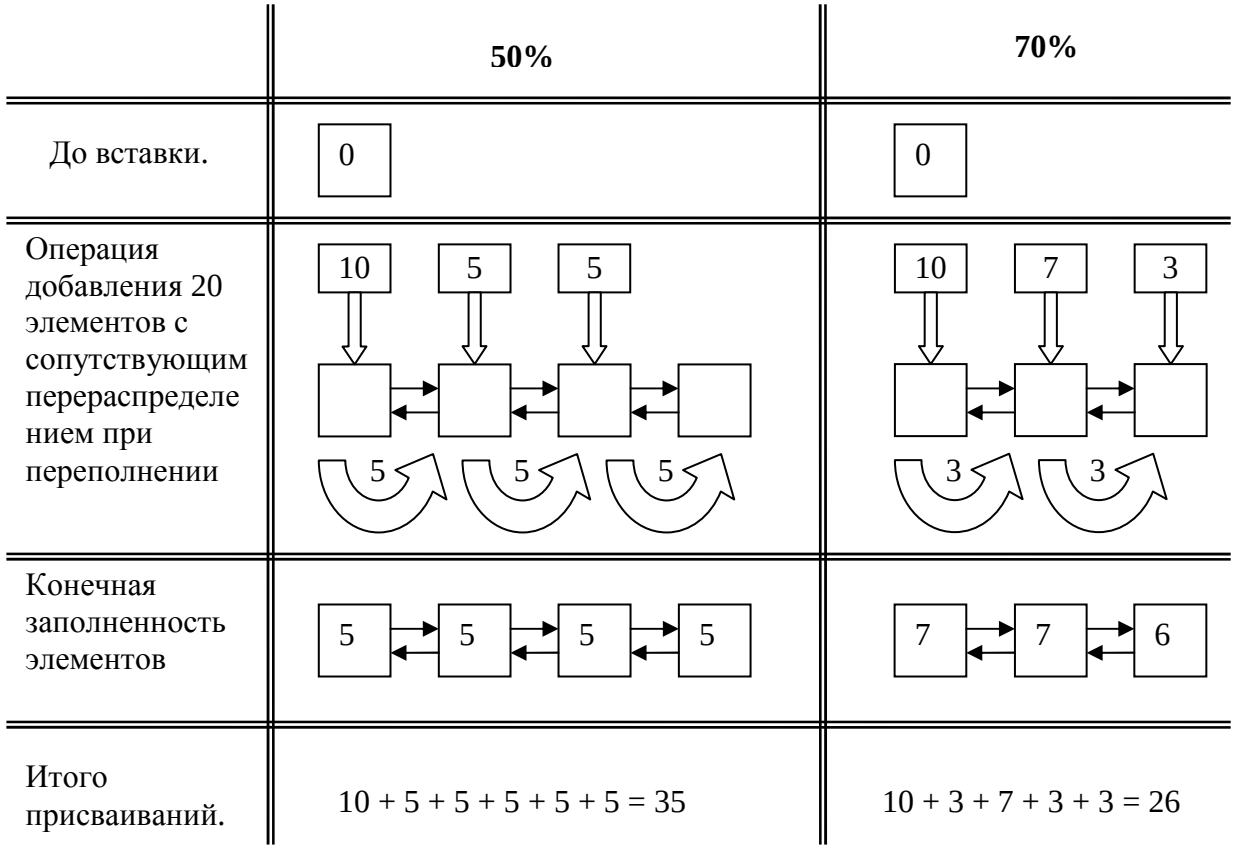
Порядок превыше всего

Сортировка. Одна из тех областей программирования, где все уже давно изучено и придумывать что-то для достаточно простой иерархической СД, как наша, просто не имеет смысла. Но можно придумать альтернативное применение самой сортировки. Мне пришла идея из сортировки сделать тестовый полигон для программы! Судите сами. Проверять производительность на глаз не спортивно. Делать сложные последовательности команд, мучить генератор случайных чисел и т.д. – на все это просто нет времени. А вот написать заведомо трудную сортировку относительно легко. Трудную не в том смысле, что «пузырьком», а в том, что все обращения к данным выполняются высокоуровневыми методами! Такими же, какие будет применять пользователь. За основу возьмем сортировку выбором. Создадим новый список. Будем искать указатель на самый «большой» объект во всем текущем списке, выкусывать и добавлять его в конец нового списка, пока текущий не станет пустым. Затем из нового перепишем все, как есть, в текущий и удалим новый.
Просто? Так нам и не надо сложно. Трудоемко? Так это и хорошо. Требует кучу памяти? Как раз и посмотрим, какого размера нужна куча. Подробнее о полевых испытаниях см. пункт «Работа с Файлами».

Расположинем!

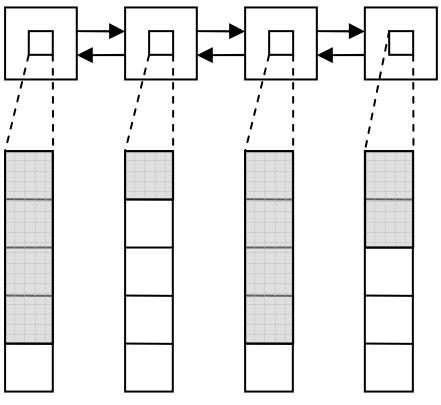
Иерархическая структура данных имеет преимущества только в том случае, если сохраняется локальность изменений. Для этого СД нижнего уровня должна иметь свободные места для включений элементов. Другими словами, массивы нижнего уровня не должны быть полностью заполненными. Поэтому после каждой операции добавления просто обязана быть проверка на переполнение. Для этого в СМУ предусмотрен информирующий метод. Но вот вопрос: каким образом перераспределять данные при переполнении. В задании говорится, что нужно переписать половину содержимого из одного элемента в новый, включенный следом за текущим. Но это приведет к тому, что СД в среднем будет заполнена на половину. Это не эффективно, так как чем меньше действительное заполнение СД второго уровня, тем больше управляющая СД. Так как управляющая СД у нас – список, то такое смещение приоритетов вызовет существенное снижение производительности. Поэтому нужно по максимуму использовать все резервы из СД нижнего уровня. Иными словами, искать такой процент заполнения, при котором будет соблюдаться разумный баланс числа вставок и числа перераспределений в уже заполненной СД. Исходя из выше сказанного, разумно ввести константу, отвечающую за критический процент заполнения, используемый при перераспределении. Значения подберем опытным путем. Допустим, оно будет ровна 0.7, тогда при заполнении массива будет создан новый элемент списка и в него перепишется SZ*0.3 указателей, а в текущем останется SZ*0.7 указателей. Где SZ – размерность массива.

Рассмотрим пример. Размерность массивов 10, нужно добавить 20 элементов. Давайте посчитаем суммарное число присвоений в случаи с критическим процентом заполнения 50% и 70%



Даже при линейном заполнении второй вариант заметно предпочтительней как по компактности итоговой СД, так и по числу присваиваний. Надо учитывать, что наряду с присваиваниями при перераспределении выполняется еще несколько операций. А значит влияние «критического процента» еще возрастет. В любом случае второй вариант минимум на 25 процентов экономичнее, чем первый. Более подробно этот вопрос рассмотрен в пункте «Работа с Файлами».

Чувство баланса



Одна из самых важных возможностей Верховного Иерарха – это возможность сохранять баланс СД. Иными словами, выравнивать размерность структуры данных второго уровня. Как уже было сказано – иерархическая структура данных имеет преимущества только в том случае, если сохраняется локальность изменений. То есть имеется свободное место в любом элементе. Но если это свободное место распределено неоднородно, как на рисунке справа, то при определенных операциях над СД возможны частые перераспределения и возможна сильная разреженность данных. Это приведет к замедлению линейного поиска и всех операций, на нем основанных. К тому же вырастает требование к памяти. Такая ситуация не желательна. К сожалению, моя реализация балансировки очень проста и не делает СД компактнее, но выравнивает заполненность массивов. Это обусловлено тем, что ко мне не сразу пришло понимание важности этого вопроса. Алгоритм очень медленный:

1. Просматриваем элементы списка и находим массив с наибольшим числом указателей и с наименьшим.

2. Если разница в их размерах больше 1, то переписываем указатель из большего в меньший, если меньше или равна 1, структура выравнена, выход.
3. Повторяем 1,2.

КАК ЭТО ДОЛЖНО БЫТЬ

1. Считаем число указателей в СД вообще.
2. Определяем число элементов списка, которое должно быть, исходя из установленного процента заполнения.
3. Если элементов списка меньше, чем должно быть, то добавляем в конец столько, сколько нужно пустых элементов. Если элементов списка больше, то переписываем из последних элементов в незаполненные первые и удаляем пустые
4. Просматриваем элементы списка и находим массив с наибольшим числом указателей и с наименьшим.
5. Если разница в их размерах больше 1,то переписываем указатель из большего в меньший, если меньше или равна 1, структура выравнена, выход.
6. Повторяем 4,5.

Данный алгоритм можно усовершенствовать, но это уже не так важно.

Добавить, извлечь, удалить.

Эти три функции являются основой редактирования, и любая СД,[2] предназначенная не только для хранения, просто обязана уметь это делать. Данные методы используют логический номер в качестве местоположения для выполнения операции. Этот логический номер поступает на вход функции поиска, описанной выше[3], и мы получаем координаты в структуре данных: указатель на элемент списка и индекс нужного нам элемента в массиве. Дальше просто выполняются соответствующие методы класса СМУ.

Функция просмотра файла.

Просмотр файла можно сделать несколькими способами. Во-первых, определимся: нас интересует постраничный просмотр с логической нумерацией строк. Для этого нужно иметь указатели на строчки, видимые на данной конкретной странице. Ввиду того, что строки могут быть разной длины, число этих строк на страницах будет не одинаковым. Отсюда следует вывод: нужно знать, с какой строки начинается страница, и сколько строк на ней поместится. Представим еще одну ситуацию: строка длиннее, чем одна страница. Значит нужно еще знать, с какого символа выводить и до какого. А если еще прикинуть, что не все символы на экране занимают только один символ, то необходимо выстраивать отдельную структуру данных с соответствующими алгоритмами. Это уже выходит за рамки данного проекта, поэтому последние две ситуации рассматривать не будем. Посчитать, сколько строчек помещается на странице, не сложно, и запомнить, с какой выводить эту страницу – тоже. Вот тут нужно рассмотреть три подхода.

1. Подход, противоречащий заповеди «Не укради времени процессора»
2. Памяти много не бывает».
3. Указатели – ключ к экономии…

[2] Имеется в виду класс структуры данных.
[3] См. пункт «Местоположение установлено» в данном разделе.

Не укради времени процессора

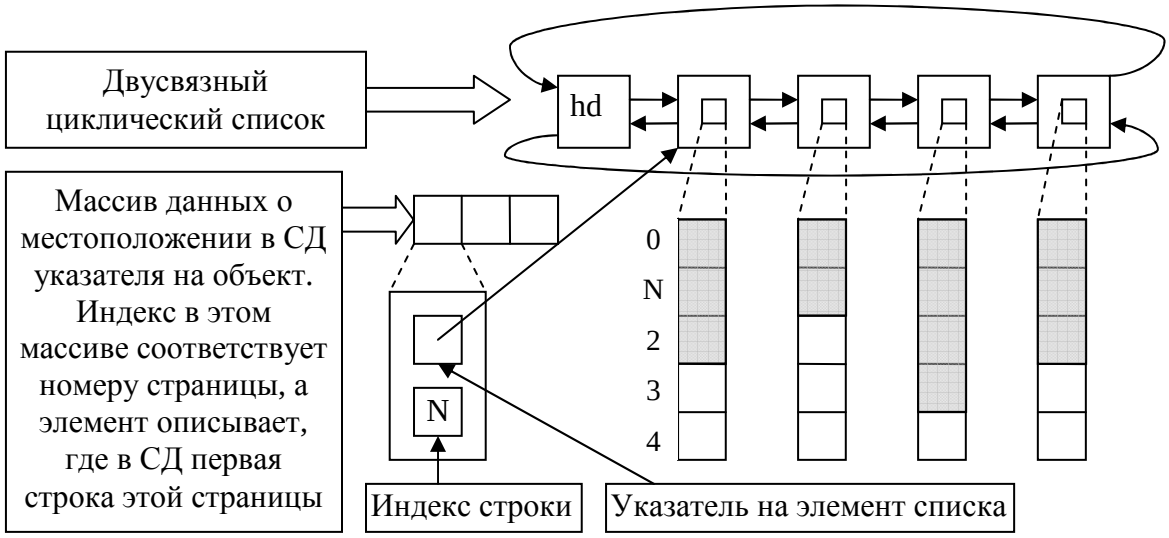
Самый простой способ своровать процессорное время – работать с данными на высоком уровне через функции, предназначенные для «разового использования». В нашем распоряжении есть переопределенная операция над списком [ ]. То есть, отбросив все предрассудки можно вообразить, что у нас и не список вовсе, а простой массив указателей. Итак, мы заводим массив целых чисел – индексов страниц. Заполняем его, и все просмотр готов. На экран будем выводить строчки по логическому номеру с текущего индекса до следующего. Все просто, только вот операция [ ] при каждом вызове будет почти весь список пролистывать. Что не есть хорошо.

Памяти много не бывает

Конечно, заголовок этого способа несколько утрирован, но все же. Что если из списка сделать массив указателей? То есть взять, посчитать, сколько объектов в СД и сделать массив указателей на эти объекты. Опять же используется [ ], но всего по одному разу на индекс. Дальше опять заводим массив индексов и вот готовый постраничный просмотр. Адресация прямая, выигрыш в скорости реальный, только вот памяти это займет как СД со списком! Способ, на мой взгляд, абсурдный, но имеющий место быть.

Низкоуровневая экономия…

В отличие от первых двух методов здесь придется лезть в шаблон списка и писать отдельный метод, чтобы не нарушался принцип ограничения доступа к данным. Один раз просмотреть каждый элемент придется все равно, но только один раз. Заведем небольшую структурированную переменную, состоящую из указателя на элемент списка и индекса строчки в массиве этого элемента. Придется тоже создать массив этих структурированных переменных и заполнить указателями на строчки начала страниц. Можно еще ввести число строк на странице или убрать указатели на элемент списка, но сути это не изменит. Мы работаем только с массивом указателей на первые строчки страниц, что уменьшает объем необходимой памяти, и имеем возможность прямого доступа через указатель, что экономит время великого процессора. То есть, приходя на уровень ниже, мы заметно выигрываем как в скорости, так и в памяти.



ПРИМЕЧАНИЕ: Я, было, начал писать этот метод, но пришёл к выводу, что в ШАБЛОНЕ он ЛИШНИЙ. Да, так можно сделать для строк, но данные в списке могут быть различных типов и сделать их грамотный просмотр сложно. Поэтому, я отказался от такого просмотра в своей программе и ограничился выводом в стандартный поток.

III. Функциональное описание

Довольно неестественно разделять функции и данные, учитывая специфику ООП, но все же. Философию и алгоритмы я изложил в разделе, посвященном структурному описанию разработки, а здесь я просто расскажу обо всех методах всех классов, использованных в программе. Конечно, коротко (все-таки 113 методов на пять классов): что передается, что возвращается, какую задачу выполняет.

template<class T,int N>class SMU{ Статический массив указателей

```
public:
        T *MU[N];   //массив указателей
         int n;     //число элементов
//…
```

SMU(); Конструктор по умолчанию
Создает массив и заполняет его NULL указателями.

~SMU()Деструктор

SMU(SMU &R);Конструктор Копирования

int Free(); Очищает массив
Освобождает память по указателям из массива, после этого заполняет массив NULL указателями.

int Remove(int); Удаляет по логическому номеру
Возвращает -1, если что-то пошло не так и 0, если операция завершена успешно.

T* Extr(int); Извлечение по лог номеру
Возвращает указатель на объект из массива указателей по индексу. Если индекс некорректный, то возвращает NULL.

int Ins(int,T*); Вставка по логическому номеру
Получает логический номер и указатель на объект.
Если вставка прошла успешно, то возвращает 0, иначе -1.

int DePut(int,T*); Замещение по логическому номеру
Получает логический номер и указатель на объект. Безусловно, замещает существующий объект вставляемым. Если по логическому номеру объекта нет (указан некорректный номер) – выходит, не выполняя вставку, возвращает -1. Если вставлено удачно, то возвращает 0.

int Max(); Возвращает индекс наибольшего элемента

int Min(); Возвращает индекс минимального элемента

T* MaxElem(); Возвращает указатель на максимальный элемент
Имеется в виду указатель на максимальный объект, а не на ячейку массива, где хранится указатель на максимальный объект.

T* MinElem(); Возвращает указатель на минимальный элемент
Аналогично предыдущему

int Add(T *R); Добавляет в конец, если есть место
Возвращает -1, если что-то пошло не так и 0, если операция завершена успешно.

void Show(); Вывод на экран содержимого
Выполняет для каждого объекта доступного по указателю из массива операцию cout<<(*указатель)

int Full(); Проверка на переполнение
Возвращает 1 если полный, и 0 если нет.

T* Ncopy(T *R); Динамическая копия объекта
Создает динамическую копию объекта типа T и возвращает указатель на созданный объект. Работает корректно, только если для объекта переопределена (или устаревает разработчика класса T) операция присваивания.

void Save(ofstream &F); Сохраняет в файловый поток F
Поэлементно выкладывает в файловый поток все объекты доступные по указателям из массива. Подразумевается, что объекты знают, как себя сохранить в файл.

void Load(ifstream &F); Загрузка из файлового потока F
Загружает в память объекты из файла пока не заполнится массив или не закончится файл. Подразумевается, что объекты знают, как себя загрузить из файла.

T* operator[](int p);
Возвращает указатель на объект из массива указателей по индексу p

SMU<T,N> &operator = (SMU<T,N> &R);
Копирует в текущий объект, объект R

SMU<T,N> &operator+(T *R);
Выполняет метод Add() для указателя R. То есть добавляет в конец, если массив не полн.

int operator<(SMU<T,N> &R);
int operator>(SMU<T,N> &R);
int operator<=(SMU<T,N> &R);
int operator>=(SMU<T,N> &R);
Сравнивается число элементов в массиве R с числом элементов в текущем.

friend ofstream &operator<<(ofstream &O,SMU<T,N> &R);
friend ifstream &operator>>(ifstream &I,SMU<T,N> &R);
friend ostream &operator<<(ostream &O, SMU<T,N> &R);
friend istream &operator>>(istream &I, SMU<T,N> &R);
Переопределенные операции с потоками.

template <class T,int N> class List{ СПИСОК

```
public:
int SZ;                 // коэффициент нормального заполнения (от 0.0 до 1.0)
                        //если, например, 0.7 ,то после заполнения элемента в нем
                        //останется N*0.7, а в следующий перепишется N*0.3
                        // указателей.
struct elem{
        SMU<T,N> MO;    //Объект класса СМУ
        elem *next;     //указатель на следующий элемент списка
        elem *prev;     //указатель на предыдущий элемент списка
        };

elem *hd;               //заголовок списка
//…
```

List(); Конструктор по умолчанию
Создает пустой список: один заголовочный элемент, который не содержит и не будет содержать данных, так как список циклический.

~List();

elem *newelem();
Создает новый элемент списка.

int Size(); Возвращет количество элементов типа T в СД вообще.

int HalfToNext(elem *&p);
Переписывает половину (или какой-то процент) указателей из текущего элемента в следующий.

void Add(T *R); Добавление в конец

void Free(int flag); Очищает содержимое списка
Если флаг=1, то удаляет элементы списка
Если флаг=0, то только очищает массивы в элементах списка.

void Show(); Вывод всех элементов на экран
Передает в стандартный поток вывода объекты СМУ содержащиеся в элементах списка.

void Copy(List<T,N>&R);
Копирует себе подобного

elem *Search(int Numb,int &sn);
Возвращает "координаты" разыскиваемого элемента

T *operator[](int n); Извлечение по логическому номеру

void operator()(T*,int); Включение по логическому номеру

void Remove(int N); Удаление по логическому номеру

void Norm(); Выравнивание структуры данных нижнего уровня
Производит балансировку СД, то есть выравнивает количество элементов в массивах.

void Ins(T*,int); Вставка по логическому номеру

void InSort(T* R); Вставка с сохранением порядка
Выполнится некорректно, если данные не были предварительно упорядочены.

T* MaxElem(elem*&,int&);Возвращает указатель на максимальный элемент

T* MinElem(); Возвращает указатель на минимальный элемент

void Sort(); Сортировка

void Save(ofstream&);

void Load(ifstream&);

friend ofstream &operator<<(ofstream &O,List<T,N> &R);
friend ifstream &operator>>(ifstream &I,List<T,N> &R);
friend ostream &operator<<(ostream &O,List<T,N> &R);

class String{

```
        int len;              //длина строки
        int numb;             //порядковый номер объекта (каким по счету был создан)
        int color;            //цвет строки
        int x;                //положение по X (столбец на экране)
        int y;                //положение по У (строка на экране)
        static int n ;// = 0;  //число объектов этого класса
        char *str;            //указатель на строку
```

public:

String(); Конструктор по умолчанию
Создает строку без символов, длина 0

String(char *s);
Загружает строку s в объект.

~String(); Деструктор

String(String &R); Конструктор Копирования

void Nstr(char *s);
Делает динамическую копию строки s, подсчитывает длину и помешает эти данные в объект.

int SkokaStrings();
Возвращает число объектов класса String

int GetLen();
Возвращает длину строки

long double StrToLong();
Преобразует строку в вещественное число типа double

int StrInStr(char *s);
Возвращает индекс символа, первое вхождения строки s в хранимой строке

void Add(char *s);
Добавляет к строке строку. Конкатенация.

void Show();
Выводит на экран строчку.

void ShowColor();
Выводить строку на экран, с цветом и координатами, указанными в параметре объекта.

void ShowC(int m);
Выводить символ по индексу.

void ShowL();
Выводит на экран строчку, преобразованную в вещественное число типа double.

void Save(ofstream &F);
Сохраняет строку в файловый поток F

void Load(ifstream &F);
Загружает строку из файлового потока F

void ShowAll();
Выводит на экран строчку, её длину, адрес объект в памяти, адрес строки в памяти, число объектов этого класса, порядковый номер текущего объекта.

void Rus();

void SetColor(int n);
Устанавливает цвет строки по номеру.

void SetX(int newx);
void SetY(int newy);
Устанавливают координаты начала строки.

void SetToCenter();
Пытается разместить строку по центру, если она не больше 80 символов.

void SetToLeft(int a);
void SetToRight(int a);
Устанавливают координаты начала строки в предопределенные положения.

void Mark();
Выделяет строку цветом, если она не была выделена и снимает выделение, если была выделена.

String &operator+(String &R);
Добавляет в текущей строке строку из R. Конкатенация

String &operator+(char *R);
Добавляет в текущей строке строку R. Конкатенация

String &operator=(String &R);
Копирует объект R в текущий.

String &operator=(char *s);
Загружает строку s в текущий объект, предварительно очистив содержимое.

int operator==(String &R);
Возвращает 1, если строки одинаковые, и 0, если разные.

char operator[](int m);
Возвращает символ из строки по индексу m. -1, если индекс не корректный.

int operator<(String &R);
int operator>(String &R);
int operator<=(String &R);
int operator>=(String &R);
Сравнивают длину строки в объекте R с длиной текущей строки.

friend ostream &operator<<(ostream &O,String &R);
friend istream &operator>>(istream &I,String &R);
friend ofstream &operator<<(ofstream &O,String &R);
friend ifstream &operator>>(ifstream &I,String &R);
Переопределенные операции для работы со стандартными потоками.

```
#define N 10 //максимальное число файлов
```

class Frame{

```
  int ID;
  ifstream From;          //поток с открытым файлом
  List<String,20> OBJ;    //список

public:
  int open;               //1, если файл открыт
  String *buf;            //буфер для операций со строками
  int operation;          //1 – работать с буфером, 0 – игнорировать буфер
  char *name;             //имя файла
  long size;              //размер файла в байтах
  long ptrs;              //число указателей на объекты в списке

//…
```

Frame();
Создает окно с новым файлом (в терминологии Windows – новый документ), имя по умолчанию: new file1 цифра выбирается в зависимости от последовательности создания. С этим файлом можно работать: вставлять, извлекать, сортировать и т.д. Но все операции только в памяти, так как на диске его нет. Для сохранения нужно выбрать соответствующий пункт меню.

void Name();
Выводить на экран имя файла.

void Open();
Открывает файл в текущее окно.

void Close();
Закрывает файл без сохранения

void Save();
Сохраняет файл
void Show();
Выводит на экран содержимое списка. Это не просмотр. Просто выполняется операция: cout<< имя объекта класса список.

void Sort();
Сортирует строчки в списке по длине

void Norm();
Балансировка.

Все методы ниже обеспечивают интерфейс с пользователем: запрашивают С клавиатуры строчки, логические номера, выводят на экран результат или работают с буфером.

void InSort();
Вставка с сохранением порядка.

void Remove();

void Ins();

void Add();

void Extr();

class Prog{

public:

```
  int frnumb;          //число открытых окон (файлов)
  Frame **MF;          //динамический массив указателей на окна
  String *buf;         //буфер для операций со строками (сохраняется при
                       //переключении между окнами и позволяет обмениваться
                       //строчками между файлами)
  int operation;       //1 – работать с буфером, 0 – игнорировать буфер
  String *parag;       //массив строк /пунктов меню
  int numpar;          //число пунктов в меню
  int set;             //текущий пункт меню
  int cur;             //текущее окно
//…
```

Prog();

~Prog();

void Menu();

void Select();

void Close();

void About();

IV. Работа с файлами.

Итак, перейдем к тестированию программы и выбору оптимального значения размерности массива указателей и процента его заполнения. Для этого откроем текстовый файл размером почти один мегабайт, содержащий 20000 строк. Файл небольшой, но зато представляет собой совершенно обычный (не сгенерированный) научно-популярный текст историко-математической направленности.

Для начала проверим зависимость времени сортировки от размерности статического массива указателей. Сортировка в данной программе нарочно имеет неоптимальный алгоритм и основана на высокоуровневых методах. Вот алгоритм вкратце:

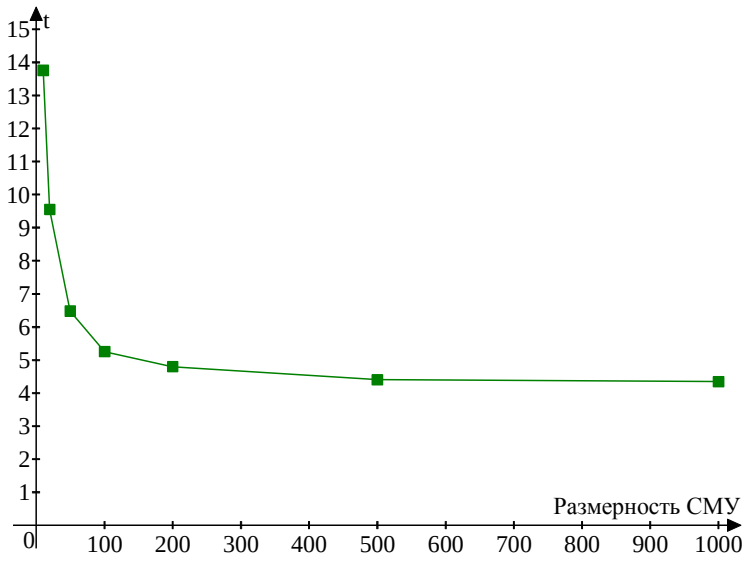
1. Создаем еще один список.
2. Находим максимальный элемент (во всей СД)
3. Переписываем его в новый список (добавляем в конец)
4. Удаляем в текущем списке максимальный элемент.
5. Повторяем 2,3,4, пока список не окажется пустым.
6. Копируем в текущий список новый.

Пункт 2 требует просмотра всех элементов (n*n)/2 раз.

Пункт 4 требует перераспределения указателей в массиве, из которого удален элемент. И т.д.

Все это делает сортировку богатой на операции вставки-удаления, поиска-просмотра – тех операций, которые наиболее часто используются пользователем при работе с данными. Таким образом, временные характеристики сортировки будут достоверным ориентиром общей производительности программы при работе с конкретной СД.

Размерность массива	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000
Время сортировки (с)	13,766	9,547	6,485	5,25	4,797	4,407	4,344	4,312	4,391	4,5	5,453	7,703



Значения размерности 20000 и 50000 позволяют поместить весь файл в один элемент списка. Но при этом весьма не эффективны, т.к. убивают преимущество иерархической СД. Справа график зависимости на участке до 1000 элементов массива. Очевидно, что значения меньше ста не оптимальны, а больше тысячи будут затратны при локальных изменениях.

Эта зависимость была получена при среднем заполнении элементов списка на 50%, но это неоптимальное значение. Исследуем влияния процента заполнения на время сортировки.

Размерность массива	Процент заполнения и время в секундах					
	50	60	70	80	90	100
100	5,531	5,203	4,969	4,781	4,703	4,344
500	4,188	4,094	3,984	3,907	3,89	3,766
1000	4,14	4,015	3,907	3,922	3,859	3,797

Если не брать во внимание 100% заполнение, которое в реальности не эффективно, а ограничиться лишь 90%, то при размерности 100 увеличение процента заполнения с 50 до 90 дает 15% прирост производительности! При размерности 500 и 1000 приблизительно 7%! Наиболее выгодным в данной ситуации является сочетание 500 и 90%, на которых я и остановился. Большая размерность СМУ проявит себя хорошо при работе с большими файлами, в то же время 10%, оставленные для добавления пользователем – это целых 50 объектов на один массив. Цифра относительно большая.

Проверим этот выбор на большом файле и на маленьком. Большой файл размером 5мб содержал 104124 строк и отсортировался за 117,140 секунд. Маленький имел размер 200кб 4241строчку и сортировался 0,079 секунды. При размерности 100 маленький отсортировался на одну тысячную быстрее, что не критично, а большой за 142,485 с, что на 25,3 секунды медленнее и только подтверждает правильность первоначального выбора.

Таким образом, мы определили оптимальные значения параметров для нашей структуры данных. Эти значения все-таки не самые лучшие, так как наиболее выгодное сочетание параметров в одной ситуации зачастую в других условиях совсем не оптимально. Поэтому более точную «заточку» нужно делать только при наличии уверенности, что при тестах моделируется именно та ситуация, в которой предстоит работать программе. Идеальным был бы вариант с авторегулировкой, то есть интеллектуальным распределением затрат при обработке данных. Но подобная система связана с мониторингом действий пользователя и неким сложным алгоритмом оценки и прогнозирования. Что в совокупности с возможной неудачной реализацией может на корню загубить преимущества. Есть еще один достаточно распространенный вариант переноса ответственности с интеллектуального алгоритма на пользователя: предоставление некоторых предустановок и возможности выбора между ними. То есть пользователь получит возможность изменять приоритеты скорости обработки данных в зависимости от решаемых задач. Поскольку ни тот, ни другой вариант не является целью данного проекта, параметры выбраны исходя из средних условий работы.

Немного о шаблонах и ООП .

Необходимость что-то менять в подходе к программированию ко мне пришла еще при написании прошлых работ в основном из-за сложности работы с динамической памятью. Постоянные её утечки и неожиданные завалы программы, причем в разных местах на одних и тех же данных привели к тому, что я перестал доверять компилятору и себе. Я знал, что ООП сильно облегчает жизнь в этом вопросе. Знакомство с этой технологией для меня проходило в два этапа. Вначале я понял суть, но не понял, как это делается. Потом я понял, как это делается и осознал, что понял суть лишь в очень узкой области…