Глава 2. Структуры данных.

Введение.

Выбор верного представления данных является неотъемлемой частью успешного и легкого решения задачи программирования. При выборе неверной структуры данных можно потратить много времени и кода на исправление первоначального решения.

В данной главе рассмотрим основные структуры данных, которые помогут программисту в удачном решении задачи. Для лучшего понимания будем использовать программе, основанные на детских карточных играх.

2.1. Элементарные структуры данных.

В данном разделе представлены операции с наиболее важными структурами данных: стеками, очередями, словарями, очередями приоритетов и наборами. Рассмотрим простейшие способы реализации этих структур с нуля.

Если Вы работаете с такими объектно-ориентированными языка программирования как C++ и Java, помните, что в них есть стандартные библиотеки, и вместо того, чтобы придумывать велосипед, подробно ознакомьтесь с ними. После этого вернитесь в этот раздел, чтобы понять как хороша каждая структура данных.

2.1.1. Стеки.

Стеки и очереди – это контейнеры, в которых элементы извлекаются в соответствии с порядком вставки, независимо от содержимого. Стеки поддерживают порядок поступления первыми. Абстрактные операции над стеком включают:

*Push (x, s)* — вставить элемент x в начало стека s.

*Pop (s)* — возвращает (и удаляет) верхний элемент стеков.

*Initialize (s)* — создать пустой стек.

*Full (s), Empty (s)* — проверьте, может ли стек принимать больше нажатий или всплывающих окон, соответственно.

В стандартных стеках и очередях не определена операция поиска элементов.

Определение этих абстрактных операций позволяет нам создавать модуль стека для использования повторно без знания деталей реализации. Самая простая реализация использует массив с индексной переменной для представления вершины стека. Стек – подходящая структура данных для этой задачи. Таким образом, одно из важных применений стеков — это когда порядок не имеет значения, потому что стеки — это особенно простые в реализации контейнеры.

2.1.2. Очереди.

В отличие от стеков очереди поддерживают порядок поступления первыми. Это является более справедливым, ведь в магазине очередь устроена по такому же принципу. На примере колоды карт можно представить, что они тоже смоделированы очередями, мы сдаем колоду карт сверху и складываем обратно вниз. Очереди FIFO будут использоваться при реализации поиска в ширину в графиках в главе 9.

Абстрактные операции над очередью включают:

*Enqueue (x, q)* — вставить элемент x в конец очереди q.

*Dequeue (q)* — возвращает (и удаляет) первый элемент из очереди q.

*Initialize(q), Full(q), Empty(q)* — аналогично этим операциям со стеками.

Из-за того, что действие происходит на обоих концах, очереди сложнее реализовать чем стеки. Простейшая реализация использует массив, вставляя новые элементы с одного конца и перемещая все оставшиеся элементы, чтобы заполнить пустое пространство, создаваемое при каждом удалении из очереди.

Т. к. перемещать каждый элемент достаточно сложно, то мы можем поддерживать индексы к первому (head) и последнему (tail) элементам в очереди, выполняю все операции локально, чтобы не очищать ранее используемые ячейки.

Циклические очереди позволяют нам повторно использовать пустое пространство. Обратите внимание, что указатель на начало списка всегда находится за указателем на конец! Когда очередь заполнена, два индекса будут указывать на соседние или идентичные элементы. Существует несколько возможных способов настройки индексов для циклических очередей. Все они сложны! Самое простое решение отличать заполненную очередь от пустой, подсчитывая, сколько элементов существует в очереди:

*typedef struct {*

*int q[QUEUESIZE +1]; /\* тело очереди \*/*

*int first; /\* положение первого элемента \*/*

*int last; /\* положение последнего элемента \*/*

*int count; /\* количество элементов очереди \*/*

*} queue;*

*init\_queue(queue \* q)*

*{*

*q->first = 0;*

*q->last = QUEUESIZE-1;*

*q->count = 0;*

*}*

*enqueue(queue \*q, int x)*

*{*

*if (q->count >= QUEUESIZE)*

*printf("Warning: queue overflow enqueue x=%d\n",x);*

*else {*

*q->last = (q->last+1) % QUEUESIZE;*

*q->q[ q->last ] = x;*

*q->count = q->count + 1;*

*}*

*}*

*int dequeue(queue \*q)*

*{*

*int x;*

*if (q->count <= 0) printf("Warning: empty queue dequeue.\n");*

*else {*

*x = q->q[ q->first ];*

*q->first = (q->first+1) % QUEUESIZE;*

*q->count = q->count - 1;*

*}*

*return(x);*

*}*

*int empty(queue \*q)*

*{*

*if (q->count <= 0) return (TRUE);*

*else return (FALSE);*

*}*

Очереди — это одна из немногих структур данных, которые легче программировать с использованием связанных списков, чем массивов, поскольку они устраняют необходимость проверки состояния переноса.

2.1.3. Словари.

В отличие от стеков и очередей, которые выполняют поиск на основе местоположения, словари выполняют поиск на основе содержимого. Они поддерживают три основные операции:

Insert(x,d) – вставить элемент x в словарь d.

Delete(x,d) – удалить элемент x (или элемент, на который указывает x) из словаря d.

Search(k,d) – возвращает элемент с ключом k, если таковой существует в словаре d.

В таком направлении как «структура данных» представлены различные способы реализации словарей. Они могут быть отсортированными и неотсортированными связанными спискам, отсортированными и неотсортированными массивами и лесами. Леса в свою очередь могут включать случайные деревья, splay, AVL, красно-черные деревья и т.д.

Основной проблемой при реализации алгоритмов является производительность. На практике нам нужно выполнить работу как можно скорее. Оценить ответ поможет то насколько сильно измениться содержимое словаря в ходе выполнения задачи. Рассмотрим плюсы и минусы словарей разных видов:

1. Статические словари – это структуры, которые создаются один раз без изменений. Они поддерживают поиск, но не вставку или удаление.

Правильным ответом для статических словарей является массив. Вопрос в том, нужно ли сохранять его отсортированным, чтобы использовать двоичный поиск. Если нет жестких ограничений по времени, можно не использовать двоичный поиск до тех пор, пока n>100.

1. Полудинамические словари – эти структуры поддерживает поиск и вставку, но не поддерживают удаление.

При условии, что мы знаем верхнюю границу количества вставляемых элементов, то можем использовать массив, если нет, то используем связанную структуру.

Хэш-таблицы являются превосходными словарными структурами данных, особенно если удаление не требуется поддерживать. Идея состоит в том, чтобы применить функцию к клавише поиска, чтобы мы могли определить, где элемент будет отображаться в массиве, не глядя на другие элементы. Чтобы создать таблицу разумного размера, мы должны допускать коллизии, два разных ключа, сопоставленных одному и тому же местоположению.

1. Полностью динамические словари — хэш-таблицы отлично подходят для полностью динамического описания, при условии, что мы используем цепочку в качестве механизма разрешения коллизий.

Здесь мы связываем связанный список с каждым местоположением таблицы, поэтому вставка, удаление и запрос сводятся к одной и той же проблеме. Если хэш-функция выполняет хорошую работу, m ключей будут равномерно распределены в таблице размером n, поэтому каждый список будет достаточно коротким для быстрого поиска.

2.1.4. Приоритетные очереди.

Приоритетные очереди – это структуры данных для наборов элементов, поддерживающих три операции:

Insert(x,p) — вставить элемент x в очередь приоритетов p.

Maximum(p) — возвращает элемент с наибольшим ключом в приоритетной очереди p.

ExtractMax(p) — возвращает и удаляет элемент с наибольшим ключом в p.

Приоритетные очереди используются для моделирования очередей, определения кто будет следующий, ведения расписаний и календарей, для планирования определенных событий.

2.1.5. Множества.

Множества (подмножества) – это неупорядоченные наборы элементов, взятые из заданного универсального множества U. Структуры таких данных отличаются от словарей, потому что существует, необходимость кодировать, какие элементы из U не входят в данное подмножество.

Основными операциями над подмножествами являются:

Member (x, S) — является ли элемент x элементом подмножества S?

Union (A,B) — построение подмножества A ∪ B всех элементов в подмножестве A или в подмножестве B.

Intersection (A, B) — построить подмножество A ∩ B всех элементов в подмножестве A и в подмножестве B.

Insert (x, S), Delete (x,S) — Вставить/удалить элемент x, в/из подмножества S.

В множествах используются отсортированным словари для упрощения операции объединения и пересечения.

2.2. Объектные библиотеки.

В таких языках программирования как C++ и Java есть реализации таких базовых структур данных в стандартных библиотеках.

2.2.1. Стандартная библиотека шаблонов C++.

Функции в C не могут определять тип своих аргументов, поэтому библиотеки, состоящие из структур данных общего назначения (стеки, очереди), не могут существовать.

Стандартная библиотека шаблонов C ++ (STL) предоставляет реализации всех структур данных, определенных выше. Тип элементов каждого объекта данных должен быть фиксированным (т. е. шаблонным) во время компиляции, поэтому объявляются два стека с разными типами элементов. Описание рекомендуемых структур данных:

* Стековые методы - S.push(), S.top(), S.pop() и S.empty().
* Очередь - Q.front(), Q.back(), Q.push(), Q.pop() и Q.empty()
* Словари - Методы включают H.erase(), H.find() и H.insert().
* Очереди приоритетов – объявлено priority queue<int> Q, методы включают в себя Q.top(), Q.push(), Q.pop() и Q.empty().
* Множества – наборы представлены в виде отсортированных ассоциативных контейнеров d set<key, comparison> S, Установленные алгоритмы включают set\_union и set\_intersection, а также другие стандартные операторы набора.

2.2.2. Java пакет *java.util*.

Полезные стандартные объекты Java появляются в пакете java.util. Коллекция всех классов Java определяет иерархию наследования, что означает, что подклассы создаются на основе суперклассов путем добавления методов и переменных. По мере продвижения вверх по иерархии наследования классы становятся более общими и абстрактными. Единственная цель абстрактного класса – предоставить соответствующий суперкласс, от которого другие классы могут наследовать его интерфейс и/или реализацию. Абстрактные классы могут только объявлять объекты, но не создавать их экземпляры. Классы, из которых могут быть созданы экземпляры объектов, называются конкретными классами.

2.3. Пример разработки программы: Отправляясь на войну.

В детской карточной игре "Война" двум игрокам раздается стандартная колода из 52 карт таким образом, чтобы у каждого игрока было по 26 карт. Игроки не смотрят на свои карты, а хранят их в пачке рубашкой вверх. Цель игры – выиграть все карты. Оба игрока играют, переворачивая свои верхние карты рубашкой вверх и кладя их на стол. Тот, кто перевернул старшую карту, берет обе карты и добавляет их (рубашкой вниз) вниз своей колоды.

Игра продолжается до тех пор, пока один игрок не выиграет, забрав все карты. Когда открытые карты имеют одинаковый ранг, начинается война. Эти карты остаются на столе, пока оба игрока разыгрывают следующую карту из своей стопки рубашкой вверх, а затем еще одну карту рубашкой вверх. Тот, у кого старшая из новых открытых карт выигрывает войну и добавляет все шесть карт в нижнюю часть своей колоды.

2.4. Попадание в колоду.

Какая структура данных является наилучшей для представления колоды карт? Основное действие, которое нам нужно от нашей колоды – это раздача карт сверху и добавление их в заднюю часть нашей колоды. Таким образом представляем руку каждого игрока с использованием очереди FIFO, которую мы определили ранее.

Но есть еще более фундаментальная проблема. Как мы представляем каждую карту? Помните, что карты бывают разных мастей (трефы, бубны, червы и пики) и достоинств (туз, 2-10, валет, дама, король). У нас есть несколько возможных вариантов. Мы можем представить каждую карту парой символов или цифр, определяющих масть и номинал. Альтернативный подход мог бы представлять каждую карточку отдельным целым числом, скажем, от 0 до 51. Основной операцией в war является сравнение карт по их номинальной стоимости.

Мы упорядочиваем значения карт от младшего к старшему и отмечаем, что есть четыре разные карты каждого значения. Умножение и деление являются ключом к отображению их от 0 до 51.

Функции ранжирования и снятия рейтинга легко разработать для перестановок, подмножеств и большинства комбинаторных объектов. Это общая методика программирования, которая может упростить операции со многими различными типами данных.

2.5. Ввод/вывод строк.

Для нашей программы входные данные состоят из пар строк для каждой входной колоды, первая из которых соответствует картам игрока 1, вторая – картам игрока 2. В нашей реализации мы выбрали вариант последовательно считывать символы и обрабатывать их один за другим. Для этого представим две колоды как массив очередей, а не как две отдельные переменные очереди. Таким образом, мы устраняем необходимость дублировать весь кол обработки ввода для каждой колоды.

2.6. Победа в войне.

При надлежащем построении, предоставляемом нашим дизайном структуры данных, основная процедура становится довольно простой. Обратите внимание, что порядок, в котором выигранные карты возвращаются в выигрышную колоду, также может быть смоделирован как очередь, поэтому мы снова можем использовать наш абстрактный тип.

2.7. Тестирование и откладка.

Из-за того, что мы не сможем увидеть тестовый пример, уточнить, где произошла ошибка программы, отладка может быть сложной задачей. Поэтому очень важно систематически тестировать свою программу перед отправкой. Своевременное устранение ошибок сэкономит время в долгосрочной перспективе.

Можно прибегнуть к следующим идеям при разработке программы:

* Протестируйте заданные входные данные.
* Проверьте неверный ввод.
* Проверка граничных условий.
* Тестируйте экземпляры, в которых вы знаете правильный ответ.
* Тестируйте большие примеры, где вы не знаете правильного ответа.

Тестирование – это искусство выявления ошибок. Отладка – это искусство их устранения.

Но как избежать попадания в такие ловушки?

1. Познакомьтесь со своим отладчиком.
2. Отображение ваших структур данных. Напишите специальные процедуры отображения для всех нетривиальных структур данных, поскольку отладчикам часто бывает трудно разобраться в них.
3. Тщательно проверяйте инварианты. Инвариант – это свойство программы, которое истинно независимо от входных данных.
4. Проверка вашего кода. Ошибки, как правило, заражают код, который слишком уродлив для чтения или слишком умен для понимания.
5. Сделайте так, чтобы ваши инструкции Print что—то значили. Добавьте отладочные инструкции печати, выведите на экран имена переменных и их положения в программе, откомментируйте свои отладочные инструкции.
6. Сделайте ваши массивы немного больше, чем необходимо. Будет полезно определить массивы на один или два элемента больше, чтобы свести к минимуму последствия любой случайной ошибки.
7. Убедитесь, что ваши ошибки действительно являются ошибками.

Заключение.

В главе рассмотрели различные структуры данных от элементарных структур данных до объектных библиотек. Разобрали положительные и отрицательные стороны таких элементарных структур данных, как стеки, очереди, словари, приоритетные очереди и наборы. Изучили стандартные библиотеки шаблонов C++, а также пакеты Java. Привели пример разработки программы на основе карточной игры.