Динамические структуры данных

А. Г. Фенстер, http://info.fenster.name

10 апреля 2010 г.

Описывая алгоритмы, бывает удобно абстрагироваться от реализации конкретных структур данных, используемых для хранения информации. Оказывается, внутренняя структура любого хранилища данных не так уж важна, если есть чётко определённый интерфейс: строгое описание всех операций, которые можно сделать с данными из этого хранилища. Можно привести пример из жизни: чтобы пользоваться телевизором, необходимо лишь уметь пользоваться его «интерфейсом» (пульт дистанционного управления), но вовсе не обязательно знать, как именно происходит приём и отображение сигнала.

Описание интерфейсов

Мы определим несколько структур данных в виде «чёрных ящиков»: опишем интерфейсные функции, но не будем пока говорить, как они устроены «изнутри».

Стек

Стеком (stack) называется структура данных, работающая по принципу «последним пришёл — первым вышел» («last in, first out»). Мы определим стек как «чёрный ящик», имеющий три интерфейсные функции:

1. `void push(int a); /* добавить в стек число a */`
2. `int pop(void); /* удалить число с верхушки стека */`
3. `int empty(void); /* проверить стек на пустоту */`

В этом примере функция `pop` удаляет верхний элемент стека и возвращает его значение. Иногда бывает удобно реализовать четвёртую функцию `int top()`, которая будет возвращать значение с верхушки стека, не удаляя его.

Гарантируется, что функция `pop` извлекает элементы в порядке, обратном тому, в котором они были помещены в стек функцией `push`. Здесь мы сделали стек для хранения целых чисел, в реальности тип данных может быть любым.

Как пользоваться таким «чёрным ящиком»? Рассмотрим пример:

```
int a;
while (scanf("%d", &a) == 1)
{
    push(a);
}
while (!empty())
{
    printf("%d\n", pop());
}
```

Приведённый фрагмент кода «разворачивает» введённую последовательность: числа, введённые с клавиатуры, сначала помещаются в стек, а затем удаляются из него в обратном порядке (по определению функций `push` и `pop`). Важно, что независимо от способа реализации функций работы со стеком (пусть это будет массив, список или что угодно) фрагмент кода будет работать корректно (если, конечно, функции реализованы правильно и удовлетворяют указанным выше условиям).

В качестве примера задачи, решаемой при помощи стека, можно привести задачу о проверке корректности скобочной строки. Пусть строка состоит из трёх видов открывающих и закрывающих скобок:

```
( ) { } [ ]
```

Корректной назовём строку, удовлетворяющую следующему определению:

1. пустая строка корректна;
2. если некоторая строка s корректна, то строки $[s]$, (s) и $\{s\}$ также корректны;

3. если некоторые строки s_1 и s_2 корректны, то строка $s_1 s_2$ корректна;

4. других корректных строк нет.

Например, строка `({})[]` является корректной, а строки `([)]` и `(()` — нет.

Чтобы проверить, является ли данная строка корректной, можно перебирать её символы и помещать каждый символ в стек, если он является открывающей скобкой. При встрече закрывающей скобки нужно извлечь один символ из стека и сравнить тип скобок: если открывающая и закрывающая скобка разных типов, строка не является корректной. При достижении конца строки необходимо убедиться, что стек пуст (в противном случае в строке есть открывающие скобки, которым не соответствует ни одна закрывающая, и строка не является корректной).

Очередь

Очередью (queue, читается [kju:]) называется структура данных, работающая по принципу «первым пришёл — первым вышел» («first in, first out»). В терминах «чёрного ящика» интерфейсными функциями очереди являются:

1. `void enqueue(int a); /* добавить число в очередь */`

2. `int dequeue(void); /* извлечь число из очереди */`

3. `int empty(void); /* проверить очередь на пустоту */`

Гарантируется, что функция `dequeue` извлекает элементы в том же порядке, в котором функция `enqueue` их помещает в очередь (в этом главное отличие очереди от стека). Пример использования очередей — реализация обхода дерева или графа в ширину.

Дек

Если представить, что в стеке и очереди элементы расположены «в одну линию», будет ясно, что при работе со стеком элементы добавляются и удаляются с одного конца этой линии, а при работе с очередью — добавляются с одного конца, а удаляются с другого (как реальная очередь в магазине). *Деком* (deque, double-ended queue — двусторонняя очередь) называется такая структура данных, в которую элементы можно

как добавить, так и удалить с обоих «концов линии». Мы не будем разбирать подробно интерфейсные функции деков, т.к. ничего нового там мы не увидим (две функции вставки элемента, две функции удаления и функция проверки на пустоту).

Куча

Куча — структура данных, предназначенная для хранения набора чисел и быстрого поиска максимального (минимального) элемента в этом наборе. Для кучи мы определим две операции:

1. вставка элемента в кучу;
2. удаление максимального элемента из кучи.

Куча может быть использована при реализации алгоритма пирамидальной сортировки (слово heap на русский язык в различных источниках переводится как «куча» или как «пирамида», отсюда некоторое несоответствие терминов).

Реализация

При реализации описанных выше структур данных нужно стараться писать такой код, чтобы те, кто его используют, могли писать программу, не задумываясь о том, как на самом деле реализована та или иная функция. Рассмотрим один-два различных способа реализации стека, очереди и дека.

Реализация стека

Стек на массивах

В самом простом случае вы можете заранее ограничить размер стека (максимальное число хранимых элементов) некоторым числом N. В этом случае функции `push`, `pop` и `empty` можно реализовать буквально в одну строчку каждую, используя массив заранее заданной длины:

```
int stack[N];
```

```
int size = 0;

void push(int a)
{
    stack[size++] = a;
}

int pop(void)
{
    return (stack[--size]);
}

int empty(void)
{
    return (size == 0);
}
```

Обратите внимание, что в функции `pop` не производится никакой проверки на то, что в стеке есть хотя бы один элемент. В реальном коде отсутствие обработки ошибок — источник проблем, но обсуждение обработки ошибок — отдельная тема, которая выходит за рамки этого текста.

Вполне реально реализовать стек на массиве, даже если ограничить количество элементов не представляется возможным. В этом случае вместо массива необходимо использовать указатель и вызывать `realloc`, если выделенного объёма памяти недостаточно для сохранения нового элемента. Однако, при хранении очень большого числа элементов целесообразнее реализовать стек иначе, а именно при помощи списков.

Стек на списках

Для односвязного списка две наиболее простые операции, выполняемые за константное время — это вставка элемента в начало списка и удаление первого элемента. Используя их, можно представить стек как список, в который элементы добавляются в начало и удаляются от начала, тем самым условие «последним пришёл — первым вышел» будет выполняться.

Обе реализации стека (на массиве и на списке) выполняют основные операции (добавление, удаление, проверка на пустоту) за константное время ($O(1)$).

Реализация очереди

Очередь на массивах

Очередь часто используется для реализации обхода в ширину в теории графов, и в этом случае часто можно ограничить максимально возможный размер очереди и даже максимально возможное количество записей, когда-либо помещённых в очередь. В этом случае удобно реализовать очередь при помощи массива и хранить два числа: номер позиции, в которую будет произведена следующая запись, и номер позиции, из которой будет произведено следующее чтение. Если эти два числа равны, очередь пуста.

```
int queue[N];
int r = 0, w = 0;

void enqueue(int a)
{
    queue[w++] = a;
}

int dequeue(void)
{
    return (queue[r++]);
}

int empty(void)
{
    return (r == w);
}
```

Плюсами такого подхода является то, что хранится порядок добавления элементов в очередь, а также простота реализации, минусом — то, что памяти может потребоваться гораздо больше, чем для хранения только элементов, которые не были удалены.

Очередь на списках

При реализации очереди при помощи односвязного списка нужно помнить, что за константное время выполняются три операции: вставка

в начало, удаление первого элемента и вставка в конец (в случае, когда хранятся указатели и на первый, и на последний элементы списка). В связи с этим логично будет добавлять новый элемент очереди в конец списка, а удалять элементы из начала списка (а не наоборот).

В обеих описанных реализациях очереди основные операции выполняются за константное время ($O(1)$).

Реализация дека

Деки не так часто встречаются при реализации простых алгоритмов (в частности, они ни разу не встретятся в нашем курсе методов программирования). Отметим лишь, что реализовывать их удобно при помощи двусвязных списков.

Реализация кучи

Кучу обычно реализуют при помощи массива, для элементов которого выполняются условия пирамиды:

$$A[i] \geqslant A[2i+1] \ \forall i \text{ при условии, что } 2i+1 < N,$$

$$A[i] \geqslant A[2i+2] \ \forall i \text{ при условии, что } 2i+2 < N,$$

где N — размер массива. При добавлении элемента в кучу его добавляют в конец и, двигаясь к началу, проверяют, не нарушено ли условие пирамиды для какого-либо элемента:

1. Добавить в конец массива (в N-ю позицию) новый элемент, увеличить N на единицу.
2. Присвоить $i = N - 1$.
3. Присвоить $k = [\frac{i-1}{2}]$.
4. Если $A_k < A_i$, поменять местами A_k и A_i, в противном случае завершить выполнение алгоритма.
5. Присвоить $i = k$.
6. Если $i = 0$, завершить выполнение алгоритма, в противном случае перейти к шагу 3.

Чтобы удалить из кучи максимальный элемент, необходимо поставить на его место последний элемент, уменьшить N и выполнить просеивание 0-го элемента массива (см. основной цикл пирамидальной сортировки).

Обе операции с кучей требуют для работы $O(\log_2 N)$ времени.