Часть III

Структуры данных

Введение

Множество — это фундаментальное понятие как в математике, так и в теории вычислительных машин. В то время как математические множества остаются неизменными, множества, которые обрабатываются в ходе выполнения алгоритмов, могут с течением времени разрастаться, уменьшаться или подвергаться другим изменениям. Назовем такие множества ***динамическими*** (dynamic). В пяти последующих главах описываются некоторые основные методы представления конечных динамических множеств и манипуляции с ними на компьютере.

В некоторых алгоритмах, предназначенных для обработки множеств, требуется выполнять операции нескольких различных видов. Например, набор операций, используемых во многих алгоритмах, ограничивается возможностью вставлять элементы в множество, удалять их, а также проверять, принадлежит ли множеству тот или иной элемент. Динамическое множество, поддерживающее перечисленные операции, называется ***словарем*** (dictionary). В других множествах могут потребоваться более сложные операции. Например, в неубывающих очередях с приоритетами, с которыми мы ознакомились в главе 6 в контексте пирамидальной структуры данных, поддерживаются операции вставки элемента и извлечение минимального элемента. Оптимальный способ

реализации динамического множества зависит от того, какие операции должны им поддерживаться.

Элементы динамического множества

В типичных реализациях динамического множества каждый его элемент представлен некоторым объектом; если в нашем распоряжении имеется указатель на объект, то можно проверять и изменять значения его полей. (В разделе 10.3 обсуждается реализация объектов и указателей в средах, где они не являются базовыми типами данных.) В динамических множествах некоторых типов предполагается, что одно из полей объекта идентифицируется как ***ключевое*** (key field). Если все ключи различны, то динамическое множество представимо в виде набора ключевых значений. Иногда объекты содержат ***сопутствующие данные*** (satellite data), которые находятся в других его полях, но не используются реализацией множества. Кроме того, объект может содержать поля, доступные для манипуляции во время выполнения операций над множеством; иногда в этих полях хранятся данные или указатели на другие объекты множества.

В некоторых динамических множествах предполагается, что их ключи являются членами полностью упорядоченного множества, например, множества действительных чисел или множества всех слов, которые могут быть расположены в алфавитном порядке. (Полностью упорядоченные множества удовлетворяют свойству трихотомии, определение которого дано в разделе 3.1.) Полное упорядочение, например, позволяет определить минимальный элемент множества или говорить о ближайшем элементе множества, превышающем заданный.

Операции в динамических множествах

Операции динамического множества можно разбить на две категории: ***запросы*** (queries), которые просто возвращают информацию о множестве, и ***модифицирующие операции*** (modifying operations), изменяющие множество. Ниже приведен список типичных операций. В каждом конкретном приложении требуется, чтобы были реализованы лишь некоторые из них.

SEARCH(S,k)

Запрос, который возвращает указатель на элемент x заданного множества S, для которого $key[x] = k$, или значение NIL, если в множестве S такой элемент отсутствует.

INSERT(S,x)

Модифицирующая операция, которая пополняет заданное множество S одним элементом, на который указывает x. Обычно предполагается, что выполнена предварительная инициализация всех полей элемента x, необходимых для реализации множества.

DELETE(S,x)

Модифицирующая операция, удаляющая из заданного множества S элемент, на который указывает x. (Обратите внимание, что в этой операции используется указатель на элемент, а не его ключевое значение.)

MINIMUM(S)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наименьшим ключом.

MAXIMUM(S)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент этого множества с наибольшим ключом.

SUCCESSOR(S,x)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим соседом ключа элемента x и превышает его. Если же x — максимальный элемент множества S, то возвращается значение NIL.

PREDECESSOR(S,x)

Запрос к полностью упорядоченному множеству S, который возвращает указатель на элемент множества S, ключ которого является ближайшим меньшим по значению соседом ключа элемента x. Если же x — минимальный элемент множества S, то возвращается значение NIL.

Запросы SUCCESSOR и PREDECESSOR часто обобщаются на множества, в которых не все ключи различаются. Для множества, состоящего из n элементов, обычно принимается допущение, что вызов операции MINIMUM, после которой $n-1$ раз вызывается операция SUCCESSOR, позволяет пронумеровать элементы множества в порядке сортировки.

Время, необходимое для выполнения операций множества, обычно измеряется в единицах, связанных с размером множества, который указывается в качестве одного из аргументов. Например, в главе 13 описывается структура данных, способная поддерживать все перечисленные выше операции, причем время их выполнения на множестве размера n выражается как $O(\lg n)$.

Обзор третьей части

В главах 10–14 описываются несколько структур данных, с помощью которых можно реализовать динамические множества. Многие из этих структур

данных будут использоваться впоследствии при разработке эффективных алгоритмов, позволяющих решать разнообразные задачи. Еще одна важная структура данных, пирамида, уже была рассмотрена в главе 6.

В главе 10 представлены основные приемы работы с такими простыми структурами данных, как стеки, очереди, связанные списки и корневые деревья. В ней также показано, как можно реализовать в средах программирования объекты и указатели, в которых они не поддерживаются в качестве примитивов. Б*о*льшая часть материала этой главы должна быть знакома всем, кто освоил вводный курс программирования.

Глава 11 знакомит читателя с хеш-таблицами, поддерживающими такие присущие словарям операции, как INSERT, DELETE и SEARCH. В наихудшем случае операция поиска в хеш-таблицах выполняется в течение времени $\Theta(n)$, однако математическое ожидание времени выполнения подобных операций равно $O(1)$. Анализ хеширования основывается на теории вероятности, однако для понимания большей части материала этой главы не требуются предварительные знания в этой области.

Описанные в главе 12 бинарные деревья поиска поддерживают все перечисленные выше операции динамических множеств. В наихудшем случае для выполнения каждой такой операции на *n*-элементном множестве требуется время $\Theta(n)$, однако при случайном построении бинарных деревьев поиска математическое ожидание времени работы каждой операции равно $O(\lg n)$. Бинарные деревья поиска служат основой для многих других структур данных.

С красно-черными деревьями, представляющими собой разновидность бинарных деревьев поиска, нас знакомит глава 13. В отличие от обычных бинарных деревьев поиска, красно-черные деревья всегда работают хорошо: в наихудшем случае операции над ними выполняются в течение времени $O(\lg n)$.

Красно-черное дерево представляет собой сбалансированное дерево поиска; в главе 18 представлено сбалансированное дерево поиска другого вида, получившее название B-дерево. Несмотря на то, что механизмы работы красно-черных деревьев несколько запутаны, из этой главы можно получить подробное представление об их свойствах без подробного изучения этих механизмов. Тем не менее, будет довольно поучительно, если вы внимательно ознакомитесь со всем представленным в главе материалом.

В главе 14 показано, как расширить красно-черные деревья для поддержки операций, отличных от перечисленных выше базовых. Сначала мы рассмотрим расширение красно-черных деревьев, обеспечивающее динамическую поддержку порядковых статистик для множества ключей, а затем — для поддержки интервалов действительных чисел.

Глава 10

Элементарные структуры данных

В этой главе рассматривается представление динамических множеств простыми структурами данных, в которых используются указатели. Несмотря на то, что с помощью указателей можно сформировать многие сложные структуры данных, здесь будут представлены лишь простейшие из них: стеки, очереди, связанные списки и деревья. Мы также рассмотрим метод, позволяющий синтезировать из массивов объекты и указатели.

10.1. Стеки и очереди

Стеки и очереди — это динамические множества, элементы из которых удаляются с помощью предварительно заданной операции DELETE. Первым из ***стека*** (stack) удаляется элемент, который был помещен туда последним: в стеке реализуется стратегия "***последним вошел — первым вышел***" (last-in, first-out — LIFO). Аналогично, в ***очереди*** (queue) всегда удаляется элемент, который содержится в множестве дольше других: в очереди реализуется стратегия "***первым вошел — первым вышел***" (first-in, first-out — FIFO). Существует несколько эффективных способов реализации стеков и очередей в компьютере. В данном разделе будет показано, как реализовать обе эти структуры данных с помощью обычного массива.

Стеки

Операция вставки применительно к стекам часто называется PUSH (запись в стек), а операция удаления — POP (снятие со стека).

Как видно из рис. 10.1, стек, способный вместить не более n элементов, можно реализовать с помощью массива $S[1..n]$. Этот массив обладает атрибутом $top[S]$, представляющим собой индекс последнего помещенного в стек элемента. Стек состоит из элементов $S[1..top[S]]$, где $S[1]$ — элемент на дне стека, а $S[top[S]]$ — элемент на его вершине.

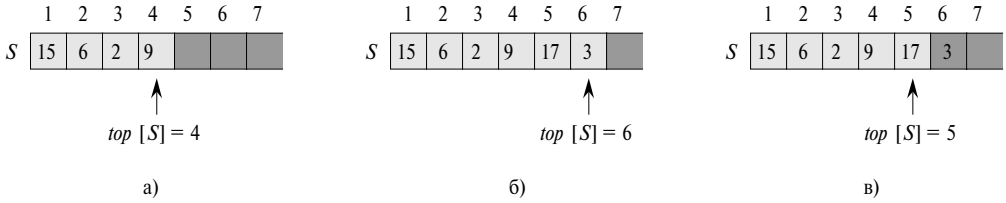

Рис. 10.1. Реализация стека S в виде массива

Если $top[S] = 0$, то стек не содержит ни одного элемента и является ***пустым*** (empty). Протестировать стек на наличие в нем элементов можно с помощью операции-запроса STACK_EMPTY. Если элемент снимается с пустого стека, говорят, что он ***опустошается*** (underflow), что обычно приводит к ошибке. Если значение $top[S]$ больше n, то стек ***переполняется*** (overflow). (В представленном ниже псевдокоде возможное переполнение во внимание не принимается.)

Каждую операцию над стеком можно легко реализовать несколькими строками кода:

```
STACK_EMPTY(S)
1  if top[S] = 0
2       then return TRUE
3       else return FALSE

PUSH(S,x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK_EMPTY(S)
2       then error "Underflow"
3       else top[S] ← top[S] - 1
4            return S[top[S] + 1]
```

Из рис. 10.1 видно, какое воздействие на стек оказывают модифицирующие операции PUSH и POP. Элементы стека находятся только в тех позициях массива, которые отмечены светло-серым цветом. В части *а* рисунка изображен стек S, состоящий из 4 элементов. На вершине стека находится элемент 9. В части *б* представлен этот же стек после вызова процедур PUSH(S,17) и PUSH(S,3), а в части *в* — после вызова процедуры POP(S), которая возвращает помещенное в стек последним значение 3. Несмотря на то, что элемент 3 все еще показан в массиве, он больше не принадлежит стеку; теперь на вершине стека — 17. Любая из трех описанных операций со стеком выполняется в течение времени $O(1)$.

Очереди

Применительно к очередям операция вставки называется ENQUEUE (поместить в очередь), а операция удаления — DEQUEUE (вывести из очереди). Подобно стековой операции POP, операция DEQUEUE не требует передачи элемента массива в виде аргумента. Благодаря свойству FIFO очередь подобна, например, живой очереди к врачу в поликлинике. У нее имеется ***голова*** (head) и ***хвост*** (tail). Когда элемент ставится в очередь, он занимает место в ее хвосте, точно так же, как человек занимает очередь последним, чтобы попасть на прием к врачу. Из очереди всегда выводится элемент, который находится в ее головной части аналогично тому, как в кабинет врача всегда заходит больной, который ждал дольше всех.

На рис. 10.2 показан один из способов, который позволяет с помощью массива $Q[1..n]$ реализовать очередь, состоящую не более чем из $n-1$

элементов. Эта очередь обладает атрибутом $head[Q]$, который является индексом головного элемента или указателем на него; атрибут $tail[Q]$ индексирует местоположение, куда будет добавлен новый элемент. Элементы очереди расположены в ячейках $head[Q]$, $head[Q]+1$, ..., $tail[Q]-1$, которые циклически замкнуты в том смысле, что ячейка 1 следует сразу же после ячейки n в циклическом порядке. При выполнении условия $head[Q]=tail[Q]$ очередь пуста. Изначально выполняется соотношение $head[Q]=tail[Q]=1$. Если очередь пустая, то при попытке удалить из нее элемент происходит ошибка опустошения. Если $head[Q]=tail[Q]+1$, то очередь заполнена, и попытка добавить в нее элемент приводит к ее переполнению.

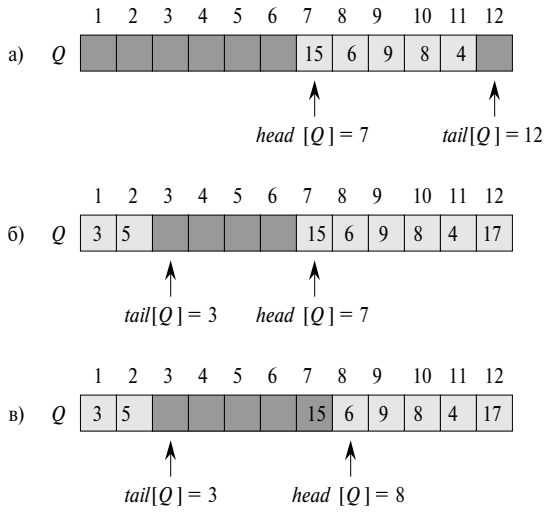

Рис. 10.2. Очередь, реализованная с помощью массива $Q[1..12]$

В наших процедурах ENQUEUE и DEQUEUE проверка ошибок опустошения и переполнения не производится. (В упражнении 10.1-4 предлагается добавить в процедуры соответствующий код.)

```
ENQUEUE(Q,x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3       then tail[Q] ← 1
4       else tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
```

```
3       then head[Q] ← 1
4       else head[Q] ← head[Q] + 1
5  return x
```

На рис. 10.2 показана работа процедур ENQUEUE и DEQUEUE. Элементы очереди содержатся только в светло-серых ячейках. В части *а* рисунка изображена очередь, состоящая из пяти элементов, расположенных в ячейках $Q[7..11]$. В части *б* показана эта же очередь после вызова процедур ENQUEUE(Q,17), ENQUEUE(Q,3) и ENQUEUE(Q,5), а в части *в* — конфигурация очереди после вызова процедуры DEQUEUE(Q), возвращающей значение ключа 15, которое до этого находилось в голове очереди. Значение ключа новой головы очереди равно 6. Каждая операция выполняется в течение времени $O(1)$.

Упражнения

10.1-1. Используя в качестве модели рис. 10.1, проиллюстрируйте результат воздействия на изначально пустой стек S, хранящийся в массиве $S[1..6]$, операций PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8) и POP(S).

10.1-2. Объясните, как с помощью одного массива $A[1..n]$ можно реализовать два стека таким образом, чтобы ни один из них не переполнялся, пока суммарное количество элементов в обоих стеках не достигнет n. Операции PUSH и POP должны выполняться в течение времени $O(1)$.

10.1-3. Используя в качестве модели рис. 10.2, проиллюстрируйте результат воздействия на изначально пустую очередь Q, хранящуюся в массиве $Q[1..6]$, операций ENQUEUE(Q,4), ENQUEUE(Q,1), ENQUEUE(Q,3), DEQUEUE(Q), ENQUEUE(Q,8) и DEQUEUE(Q).

10.1-4. Перепишите код процедур ENQUEUE и DEQUEUE таким образом, чтобы они обнаруживали ошибки опустошения и переполнения.

10.1-5. При работе со стеком элементы можно добавлять в него и извлекать из него только с одного конца. Очередь позволяет добавлять элементы с одного конца, а извлекать — с другого. ***Очередь с двусторонним доступом***, или ***дек*** (deque), предоставляет возможность производить вставку и удаление с обоих концов. Напишите четыре процедуры, выполняющиеся в течение времени $O(1)$ и позволяющие вставлять и удалять элементы с обоих концов дека, реализованного с помощью массива.

10.1-6. Покажите, как реализовать очередь с помощью двух стеков. Проанализируйте время работы операций, которые выполняются с ее элементами.

10.1-7. Покажите, как реализовать стек с помощью двух очередей. Проанализируйте время работы операций, которые выполняются с его элементами.

10.2. Связанные списки

Связанный список (linked list) — это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. Связанные списки обеспечивают простое и гибкое представление динамических множеств и поддерживают все операции (возможно, недостаточно эффективно), перечисленные на стр. 3.

Как показано на рис. 10.3, каждый элемент ***дважды связанного списка*** (doubly linked list) L — это объект с одним полем ключа *key* и двумя полями-

указателями: *next* (следующий) и *prev* (предыдущий). Этот объект может также содержать другие сопутствующие данные. Для заданного элемента списка x указатель $next[x]$ указывает на следующий элемент связанного списка, а указатель $prev[x]$ — на предыдущий. Если $prev[x] = \text{NIL}$, у элемента x нет предшественника, и, следовательно, он является первым, т.е. ***головным*** в списке. Если $next[x] = \text{NIL}$, то у элемента x нет последующего, поэтому он является последним, т.е. ***хвостовым*** в списке. Атрибут $head[L]$ указывает на первый элемент списка. Если $head[L] = \text{NIL}$, то список пуст.

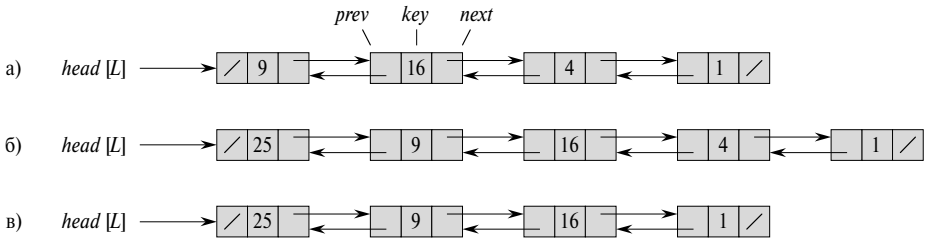

Рис. 10.3. Вставка в дважды связанный список и удаление из него

Списки могут быть разных видов. Список может быть однократно или дважды связанным, отсортированным или неотсортированным, кольцевым или некольцевым. Если список ***однократно связанный*** (***однонаправленный***) (singly linked), то указатель *prev* в его элементах отсутствует. Если список ***отсортирован*** (sorted), то его линейный порядок соответствует линейному порядку его ключей; в этом случае минимальный элемент находится в голове списка, а максимальный — в его хвосте. Если же список не отсортирован, то его элементы могут располагаться в произвольном порядке. Если ***список кольцевой*** (circular list), то указатель *prev* его головного элемента указывает на его хвост, а указатель *next* хвостового элемента — на головной элемент. Такой список можно рассматривать как замкнутый в виде кольца набор элементов. В оставшейся части раздела предполагается, что списки, с которыми нам придется работать, — неотсортированные и дважды связанные.

Поиск в связанном списке

Процедура LIST_SEARCH(L,k) позволяет найти в списке L первый элемент с ключом k путем линейного поиска, и возвращает указатель на найденный элемент. Если элемент с ключом k в списке отсутствует, возвращается значение NIL. Процедура LIST_SEARCH(L,4), вызванная для связанного списка, изображенного на рис. 10.3*а*, возвращает указатель на третий элемент, а процедура LIST_SEARCH(L,7) — значение NIL:

```
LIST_SEARCH(L,k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3       do x ← next[x]
4  return x
```

Поиск с помощью процедуры LIST_SEARCH в списке, состоящем из n элементов, в наихудшем случае выполняется в течение времени $\Theta(n)$, поскольку может понадобиться просмотреть весь список.

Вставка в связанный список

Если имеется элемент x, полю *key* которого предварительно присвоено значение, то процедура LIST_INSERT вставляет элемент x в переднюю часть списка (рис. 10.3*б*):

```
LIST_INSERT(L,x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3       then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

Время работы процедуры LIST_INSERT равно $O(1)$.

Удаление из связанного списка

Представленная ниже процедура LIST_DELETE удаляет элемент x из связанного списка L. В процедуру необходимо передать указатель на элемент x,

после чего она удаляет x из списка путем обновления указателей. Чтобы удалить элемент с заданным ключом, необходимо сначала вызвать процедуру LIST_SEARCH для получения указателя на элемент:

```
LIST_DELETE(L,x)
1  if prev[x] ≠ NIL
2       then next[prev[x]] ← next[x]
3       else head[L] ← next[x]
4  if next[x] ≠ NIL
5       then prev[next[x]] ← prev[x]
```

Удаление элемента из связанного списка проиллюстрировано на рис. 10.3*в*. Время работы процедуры LIST_DELETE равно $O(1)$, но если нужно удалить элемент с заданным ключом, то в наихудшем случае понадобится время $\Theta(n)$, поскольку сначала необходимо вызвать процедуру LIST_SEARCH.

Ограничители

Код процедуры LIST_DELETE мог бы быть проще, если бы можно было игнорировать граничные условия в голове и хвосте списка:

```
LIST_DELETE′(L,x)
1  next[prev[x]] ← next[x]
2  prev[next[x]] ← prev[x]
```

Ограничитель (sentinel) — это фиктивный объект, упрощающий учет граничных условий. Например, предположим, что в списке L предусмотрен объект $nil[L]$, представляющий значение NIL, но при этом содержащий все поля, которые имеются у других элементов. Когда в коде происходит обращение к значению NIL, оно заменяется обращением к ограничителю $nil[L]$. Из рис. 10.4 видно, что наличие ограничителя преобразует обычный дважды связанный список в ***циклический дважды связанный список с ограничителем*** (circular, doubly linked list with a sentinel). В таком списке ограничитель $nil[L]$ расположен между головой и хвостом. Поле $next[nil[L]]$ указывает на голову списка, а поле $prev[nil[L]]$ — на его хвост. Аналогично, поле *next* хвостового

элемента и поле *prev* головного элемента указывают на элемент $nil[L]$. Поскольку поле $next[nil[L]]$ указывает на голову списка, можно упразднить атрибут $head[L]$, заменив ссылки на него ссылками на поле $next[nil[L]]$. Пустой список содержит только ограничитель, поскольку и полю $next[nil[L]]$, и полю $prev[nil[L]]$ можно присвоить значение $nil[L]$.

Код процедуры LIST_SEARCH остается тем же, что и раньше, однако ссылки на значения NIL и $head[L]$ заменяются так, как описано выше:

```
LIST_SEARCH′(L,k)
1  x ← next[nil[L]]
2  while x ≠ nil[L] and key[x] ≠ k
3      do x ← next[x]
4  return x
```

Удаление элемента из списка производится с помощью уже описанной процедуры LIST_DELETE′, состоящей всего из двух строк. Для вставки элемента в список используется приведенная ниже процедура:

```
LIST_INSERT′(L,x)
1  next[x] ← next[nil[L]]
2  prev[next[nil[L]]] ← x
3  next[nil[L]] ← x
4  prev[x] ← nil[L]
```

Действие процедур LIST_INSERT′ и LIST_DELETE′ проиллюстрировано на рис. 10.4. В части *а* этого рисунка приведен пустой список. В части *б* изображен связанный список, уже знакомый нам по рис. 10.3*а*, в голове которого находится ключ 9, а в хвосте — ключ 1. В части *в* изображен этот же список после выполнения процедуры LIST_INSERT′(*L*,*x*), где $key[x] = 25$. Новый объект становится в голову списка. В части *г* рассматриваемый список представлен после удаления в нем объекта с ключом 1. Новым хвостовым объектом становится объект с ключом 4.

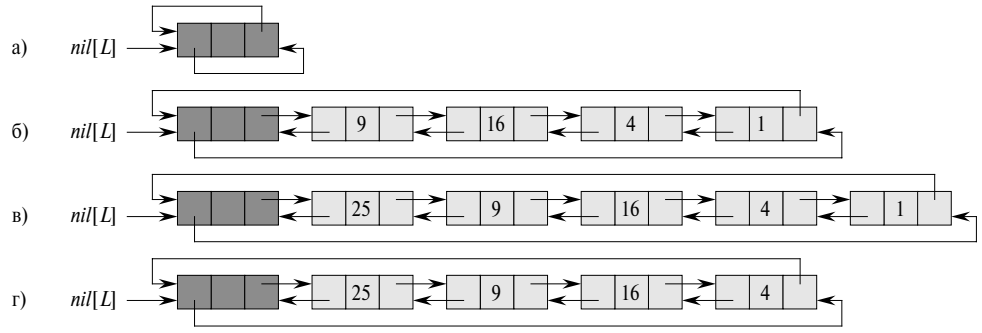

Рис. 10.4. Циклический дважды связанный список с ограничителями

Применение ограничителей редко приводит к улучшению асимптотической зависимости времени обработки структур данных, однако оно может уменьшить величину постоянных множителей. Благодаря использованию ограничителей в циклах обычно не столько увеличивается скорость работы кода, сколько повышается его ясность. Например, представленный выше код, предназначенный для обработки связанного списка, упрощается в результате применения ограничителей, но сэкономленное в процедурах LIST_INSERT′ и LIST_DELETE′ время равно всего лишь $O(1)$. Однако в других ситуациях применение ограничителей позволяет сделать код в циклах компактнее, а иногда даже уменьшить время работы в n или n^2 раз.

Ограничители не стоит применять необдуманно. Если в программе обрабатывается большое количество маленьких списков, то дополнительное пространство, занимаемое ограничителями, может стать причиной значительного перерасхода памяти. В этой книге ограничители применяются лишь тогда, когда они действительно упрощают код.

Упражнения

10.2-1. Можно ли реализовать операцию INSERT над динамическим множеством в однократно связанном списке так, чтобы время ее работы было равно $O(1)$? А операцию DELETE?

10.2-2. Реализуйте стек с помощью однократно связанного списка L. Операции PUSH и POP должны по-прежнему выполняться в течение времени $O(1)$.

10.2-3. Реализуйте очередь с помощью однократно связанного списка L. Операции ENQUEUE и DEQUEUE должны выполняться в течение времени $O(1)$.

10.2-4. В каждой итерации цикла, входящего в состав процедуры LIST_SEARCH′, необходимо выполнить две проверки: $x \neq nil[L]$ и $key[x] \neq k$. Покажите, как избежать проверки $x \neq nil[L]$ в каждой итерации.

10.2-5. Реализуйте базовые словарные операции INSERT, DELETE и SEARCH с помощью однократно связанного циклического списка. Определите время работы этих процедур.

10.2-6. Операция UNION (объединение) над динамическим множеством принимает в качестве входных данных два отдельных множества S_1 и S_2 и возвращает множество $S = S_1 \cup S_2$, состоящее из всех элементов множеств S_1 и S_2. В результате выполнения этой операции множества S_1 и S_2 обычно разрушаются. Покажите, как организовать поддержку операции UNION со временем работы $O(1)$ с помощью подходящей списочной структуры данных.

10.2-7. Разработайте нерекурсивную процедуру со временем работы $\Theta(n)$, обращающую порядок расположения элементов в однократно связанном списке. Процедура должна использовать некоторый постоянный объем памяти помимо памяти, необходимой для хранения самого списка.

10.2-8.* Объясните, как можно реализовать дважды связанный список, используя при этом всего лишь один указатель $np[x]$ в каждом элементе вместо обычных двух (*next* и *prev*). Предполагается, что значения всех указателей могут рассматриваться как *k*-битовые целые чисел, а величина $np[x]$ определяется как $np[x] = next[x]$ XOR $prev[x]$, т.е. как *k*-битовое "исключающее или" значений $next[x]$ и $prev[x]$. (Значение NIL представляется нулем.) Не забудьте указать, какая информация нужна для доступа к голове списка. Покажите, как реализовать операции SEARCH, INSERT и DELETE в таком списке. Покажите также, как можно обратить порядок элементов в таком списке за время $O(1)$.

10.3. Реализация указателей и объектов

Как реализовать указатели и объекты в таких языках, как Fortran, где их просто нет? В данном разделе мы ознакомимся с двумя путями реализации связанных структур данных, в которых такой тип данных, как указатель, в явном виде не используется. Объекты и указатели будут созданы на основе массивов и индексов.

Представление объектов с помощью нескольких массивов

Набор объектов с одинаковыми полями можно представить с помощью массивов. Каждому полю в таком представлении будет соответствовать свой массив. В качестве примера рассмотрим рис. 10.5, где проиллюстрирована реализация с помощью трех массивов связанного списка, представленного на рис. 10.3*a*. В каждом столбце элементов на рисунке представлен отдельный

объект. В массиве *key* содержатся значения ключей элементов, входящих в данный момент в динамическое множество, а указатели хранятся в массивах *next* и *prev*. Для заданного индекса массива x элементы $key[x]$, $next[x]$ и $prev[x]$ представляют объект в связанном списке. В такой интерпретации указатель x — это просто общий индекс в массивах *key*, *next* и *prev*.

Указатели при таком способе хранения представляют собой просто индексы объектов, на которые они указывают. На рис. 10.3*a* объект с ключом 4 в связанном списке следует после объекта с ключом 16. Соответственно, на рис. 10.5 ключ 4 является значением элемента $key[2]$, а ключ 16 — значением элемента $key[5]$, поэтому $next[5] = 2$ и $prev[2] = 5$. В качестве значения NIL обычно используется целое число (например, 0 или -1), которое не может быть индексом массива. В переменной L содержится индекс головного элемента списка.

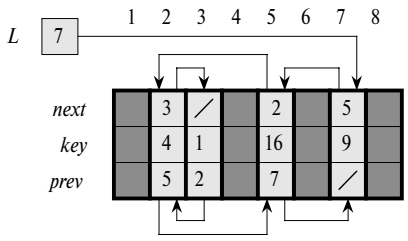

Рис. 10.5. Связанный список, представленный массивами *key*, *next* и *prev*

В нашем псевдокоде квадратные скобки используются как для обозначения индекса массива, так и для выбора поля (атрибута) объекта. В любом случае значение выражений $key[x]$, $next[x]$ и $prev[x]$ согласуется с практической реализацией.

Представление объектов с помощью одного массива

Обычно слова в памяти компьютера адресуются с помощью целых чисел от 0 до $M-1$, где M — это достаточно большое целое число. Во многих языках программирования объект занимает непрерывную область памяти компьютера. Указатель — это просто адрес первой ячейки памяти, где находится начало объекта. Другие ячейки памяти, занимаемые объектом, можно индексировать путем добавления к указателю величины соответствующего смещения.

Этой же стратегией можно воспользоваться при реализации объектов в средах программирования, в которых отсутствуют указатели. Например, на рис. 10.6 показано, как можно реализовать связанный список, знакомый нам из рис. 10.3*a* и 10.5, с помощью одного массива A. Объект занимает подмассив смежных элементов $A[j..k]$. Каждое поле объекта соответствует смещению, величина которого принимает значения от 0 до $k-j$, а указателем на объект является индекс j. Каждый элемент списка — это объект, занимающий по три расположенных рядом элемента массива. Указатель на объект — это индекс его первого элемента. Объекты, в которых содержатся элементы списка, на рисунке отмечены светло-серым цветом. На рис. 10.6 сдвиги, отвечающие полям *key*, *next* и *prev*, равны 0, 1 и 2 соответственно. Чтобы считать значение поля $prev[i]$ для данного указателя i, к значению указателя добавляется величина сдвига 2, в результате чего получается $A[i+2]$.

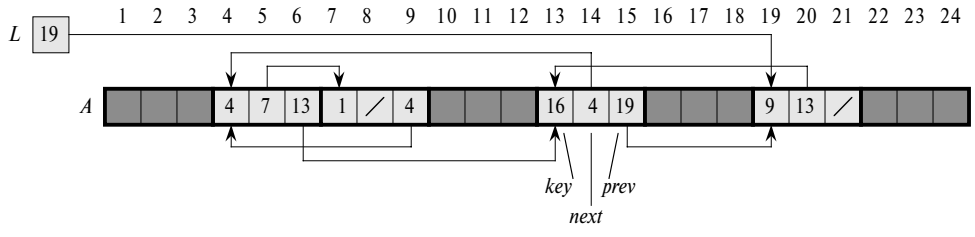

Рис. 10.6. Связанный список, представленный единственным массивом

Представление в виде единственного массива можно считать более гибким в том плане, что с его помощью в одном и том же массиве можно хранить объекты различной длины. Задача управления такими неоднородными наборами объектов сложнее, чем аналогичная задача для однородного набора объектов, где все объекты состоят из одинаковых полей. Поскольку большинство структур данных, которое нам предстоит рассмотреть, состоит из однородных элементов, для наших целей достаточно использовать представление объектов с помощью нескольких массивов.

Выделение и освобождение памяти

При добавлении нового элемента в список надо выделить для него место в памяти, что влечет за собой необходимость учета использования адресов. В некоторых системах функцию определения того, какой объект больше не используется, выполняет "***сборщик мусора***" (garbage collector). Однако многие приложения достаточно просты и вполне способны самостоятельно возвращать неиспользованное объектами пространство модулю управления памятью. Давайте рассмотрим задачу выделения и освобождения памяти для однородных объектов на примере дважды связанного списка, представленного с помощью нескольких массивов.

Предположим, что в таком представлении используются массивы длиной m, и что в какой-то момент динамическое множество содержит $n \le m$ элементов. В этом случае n объектов представляют элементы, которые находятся в данный момент времени в динамическом множестве, а $m - n$ элементов ***свободны***. Их можно использовать для представления элементов, которые будут вставляться в динамическое множество в будущем.

Свободные объекты хранятся в однократно связанном списке, который мы назовем ***списком свободных позиций*** (free list). Список свободных позиций

использует только массив *next*, в котором хранятся указатели *next* списка. Головной элемент списка свободных позиций находится в глобальной переменной *free*. Когда динамическое множество, представленное связанным списком L, не является пустым, список свободных позиций используется вместе со списком L, как показано на рис. 10.7. Заметим, что каждый объект в таком представлении находится либо в списке L, либо в списке свободных позиций, но не в обоих списках одновременно.

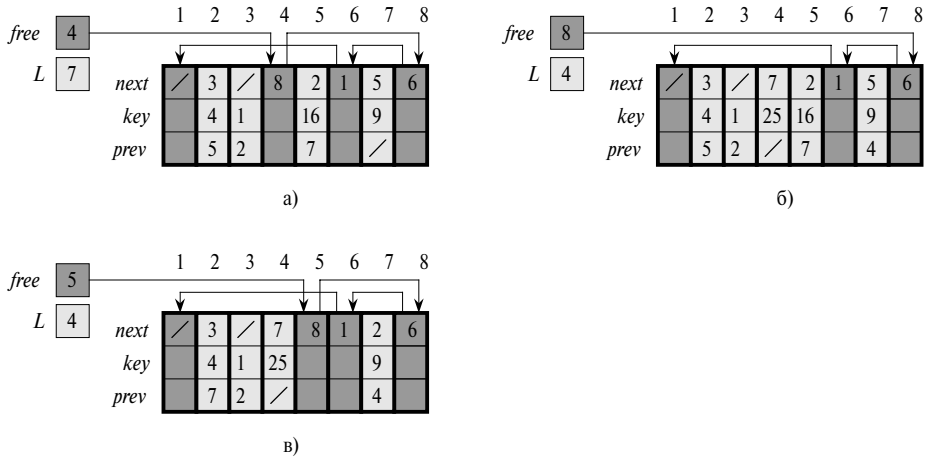

Рис. 10.7. Процедуры выделения и освобождения объекта

Список свободных позиций — это стек: очередной выделяемый объект является последним освобожденным. Таким образом, реализовать процедуры выделения и освобождения памяти для объектов можно с помощью стековых операций PUSH и POP. Глобальная переменная *free*, использующаяся в приведенных ниже процедурах, указывает на первый элемент списка свободных позиций:

```
ALLOCATE_OBJECT ()
1  if free = NIL
2       then error "Нехватка памяти"
3       else x ← free
4            free ← next[x]
5            return x

FREE_OBJECT(x)
1  next[x] ← free
2  free ← x
```

В начальный момент список свободных позиций содержит все n объектов, для которых не выделено место. Когда в списке свободных позиций больше не остается элементов, процедура ALLOCATE_OBJECT выдает сообщение об ошибке.

На рис. 10.7 показано, как изменяется исходный список свободных позиций (рис. 10.7*а*) при вызове процедуры ALLOCATE_OBJECT(), которая возвращает индекс 4, и вставке в эту позицию нового элемента (рис. 10.7*б*), а также после вызова LIST_DELETE(L,5) с последующим вызовом FREE_OBJECT(5) (рис. 10.7*в*).

Часто один список свободных позиций обслуживает несколько связанных списков. На рис. 10.8 изображены два связанных списка и обслуживающий их список свободных позиций.

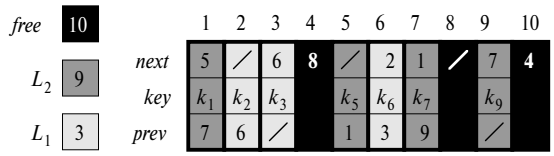

Рис. 10.8. Связанные списки L_1 (светло-серый) и L_2 (темно-серый), и обслуживающий их список свободных позиций (черный)

Время выполнения обеих процедур равно $O(1)$, что делает их весьма практичными. Эти процедуры можно модифицировать так, чтобы они работали с любым набором однородных объектов, лишь бы одно из полей объекта работало в качестве поля *next* списка свободных позиций.

Упражнения

10.3-1. Изобразите последовательность $\langle 13,4,8,19,5,11 \rangle$, хранящуюся в дважды связанном списке, представленном с помощью нескольких массивов. Выполните это же задание для представления с помощью одного массива.

10.3-2. Разработайте процедуры ALLOCATE_OBJECT и FREE_OBJECT для однородного набора объектов, реализованных с помощью одного массива.

10.3-3. Почему нет необходимости присваивать или вновь устанавливать значения полей *prev* при реализации процедур ALLOCATE_OBJECT и FREE_OBJECT?

10.3-4. Зачастую (например, при страничной организации виртуальной памяти) все элементы списка желательно располагать компактно, в непрерывном участке памяти. Разработайте такую реализацию процедур ALLOCATE_OBJECT и FREE_OBJECT, чтобы элементы списка занимали позиции $1..m$, где m — количество элементов списка. (*Указание:* воспользуйтесь реализацией стека с помощью массива.)

10.3-5. Пусть L — дважды связанный список длины m, который хранится в массивах *key*, *prev* и *next* длины n. Предположим, что управление этими массивами осуществляется с помощью процедур ALLOCATE_OBJECT и FREE_OBJECT, использующих дважды связанный список свободных позиций F. Предположим также, что ровно m из n элементов находятся в списке L, а остальные $n-m$ элементов — в списке свободных позиций. Разработайте процедуру COMPACTIFY_LIST(L,F), которая в качестве параметров получает список L и список свободных позиций F, и перемещает элементы списка L таким образом, чтобы они занимали ячейки массива с индексами $1,2,\ldots,m$, а также преобразует список свободных позиций F так, что он остается корректным и содержит ячейки массива с индексами $m+1,m+2,\ldots,n$. Время работы этой процедуры должно быть равным $\Theta(n)$, а объем использующейся ею дополнительной памяти не должен превышать некоторую фиксированную величину. Приведите тщательное обоснование корректности представленной процедуры.

10.4. Представление корневых деревьев

Приведенные в предыдущем разделе методы представления списков подходят для любых однородных структур данных. Данный раздел посвящен задаче представления корневых деревьев с помощью связанных структур данных. Мы начнем с рассмотрения бинарных деревьев, а затем рассмотрим представление деревьев с произвольным количеством дочерних элементов.

Каждый узел дерева представляет собой отдельный объект. Как и при изучении связанных списков, предполагается, что в каждом узле содержится поле *key*. Остальные интересующие нас поля — это указатели на другие узлы, и их вид зависит от типа дерева.

Бинарные деревья

Как показано на рис. 10.9, для хранения указателей на родительский, дочерний левый и дочерний правый узлы бинарного дерева T, используются поля p, *left* и *right*. Если $p[x] = \text{NIL}$, то x — корень дерева. Если у узла x нет дочерних узлов, то $left[x] = right[x] = \text{NIL}$. Атрибут $root[T]$ указывает на корневой узел дерева T. Если $root[T] = \text{NIL}$, то дерево T пустое.

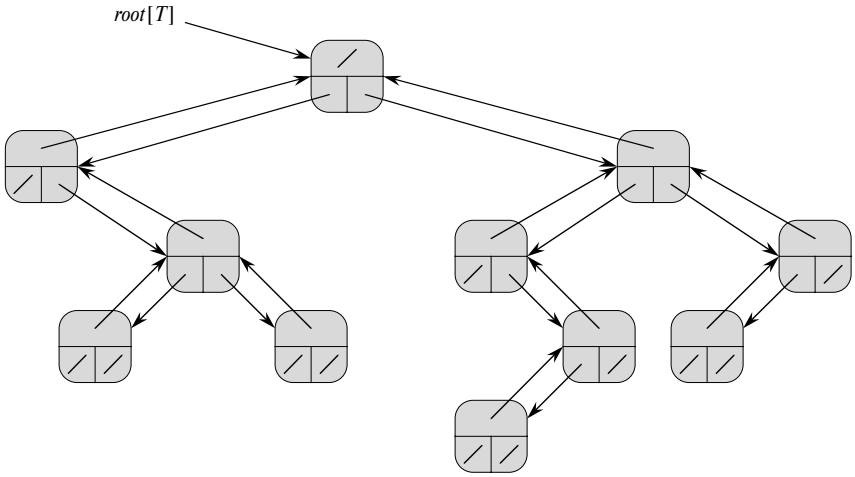

Рис. 10.9. Представление бинарного дерева T (поле *key* не показано)

Корневые деревья с произвольным ветвлением

Схему представления бинарных деревьев можно обобщить для деревьев любого класса, в которых количество дочерних узлов не превышает некоторой константы k. При этом поля правый и левый заменяются полями $child_1$, $child_2$, …, $child_k$. Если количество дочерних элементов узла не ограничено, то эта схема не работает, поскольку заранее не известно, место для какого количества полей (или массивов, при использовании представления с помощью нескольких массивов) нужно выделить. Кроме того, если количество дочерних элементов k ограничено большой константой, но на самом деле у многих узлов потомков намного меньше, то значительный объем памяти расходуется напрасно.

К счастью, существует остроумная схема представления деревьев с произвольным количеством дочерних узлов с помощью бинарных деревьев. Преимущество этой схемы в том, что для любого корневого дерева с n дочерними узлами требуется объем памяти $O(n)$. На рис. 10.10 проиллюстрировано представление ***с левым дочерним и правым сестринским узлами*** (left-child, right-sibling representation). Как и в предыдущем представлении, в каждом узле этого представления содержится указатель p, а

атрибут $root[T]$ указывает на корень дерева T. Однако вместо указателей на дочерние узлы каждый узел x содержит всего два указателя:

1. в поле $left\_child[x]$ хранится указатель на крайний левый дочерний узел узла x;
2. в поле $right\_sibling[x]$ хранится указатель на узел, расположенный на одном уровне с узлом x справа от него.

Если узел x не имеет потомков, то $left\_child[x] = \text{NIL}$, а если узел x — крайний правый дочерний элемент какого-то родительского элемента, то $right\_sibling[x] = \text{NIL}$.

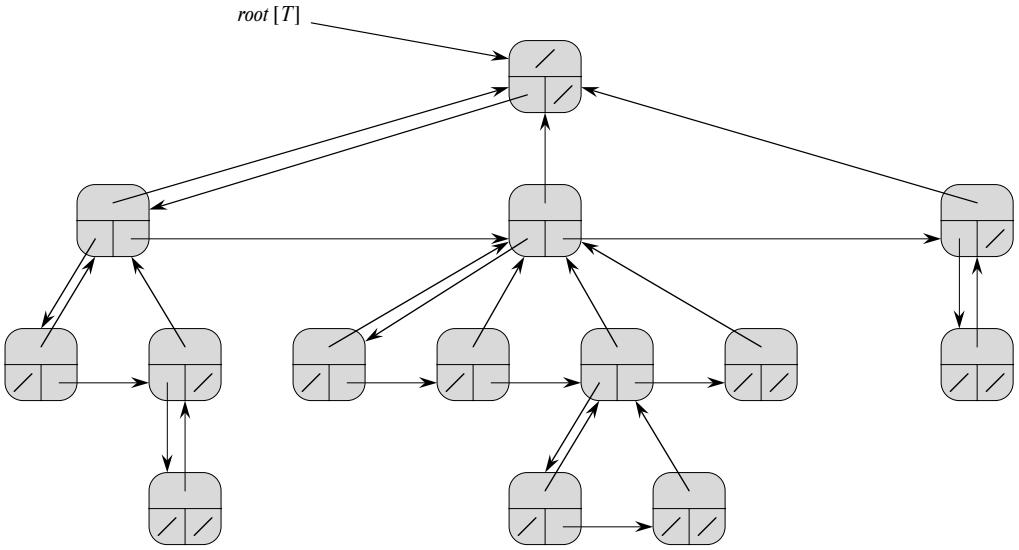

Рис. 10.10. Представление дерева с левым дочерним и правым сестринским узлами

Другие представления деревьев

Иногда встречаются и другие способы представления корневых деревьев. Например, в главе 6 описано представление пирамиды, основанной на полном бинарном дереве, в виде индексированного массива. Деревья, о которых идет речь в главе 21, восходят только к корню, поэтому в них содержатся лишь указатели на родительские элементы; указатели на дочерние элементы

отсутствуют. Здесь возможны самые различные схемы. Наилучший выбор схемы зависит от конкретного приложения.

Упражнения

10.4-1. Начертите бинарное дерево, корень которого имеет индекс 6, и которое представлено приведенными ниже полями.

Индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

10.4-2. Разработайте рекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева.

10.4-3. Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. В качестве вспомогательной структуры данных воспользуйтесь стеком.

10.4-4. Разработайте процедуру, которая за время $O(n)$ выводит ключи всех n узлов произвольного корневого дерева. Дерево реализовано в представлении с левым дочерним и правым сестринским элементами.

10.4-5.* Разработайте нерекурсивную процедуру, которая за время $O(n)$ выводит ключи всех n узлов бинарного дерева. Объем дополнительной памяти (кроме той, что отводится под само дерево) не должен превышать

некоторую константу. Кроме того, в процессе выполнения процедуры дерево (даже временно) не должно модифицироваться.

10.4-6.* В представлении произвольного корневого дерева с левым дочерним и правым сестринским узлами в каждом узле есть по три указателя: *left_child*, *right_sibling* и *parent*. Если задан какой-то узел дерева, то определить его родительский узел и получить к нему доступ можно в течение фиксированного времени. Определить же все дочерние узлы и получить к ним доступ можно в течение времени, линейно зависящего от количества дочерних узлов. Разработайте способ хранения дерева с произвольным ветвлением, в каждом узле которого используется только два (а не три) указателя и одна логическая переменная, при котором родительский узел или все дочерние узлы определяются в течение времени, линейно зависящего от количества дочерних узлов.

Задачи

10-1. **Сравнения списков**

Определите асимптотическое время выполнения перечисленных в приведенной ниже таблице операций над элементами динамических множеств в наихудшем случае, если эти операции выполняются со списками перечисленных ниже типов.

	Неотсортированный однократно связанный список	Отсортированный однократно связанный список	Неотсортированный дважды связанный список	Отсортированный дважды связанный список
SEARCH(*L*,*k*)				
INSERT(*L*,*x*)				
DELETE(*L*,*x*)				
SUCCESSOR(*L*,*x*)				
PREDECESSOR(*L*,*x*)				
MINIMUM(*L*)				

MAXIMUM(*L*)				

10-2. **Реализация объединяемых пирамид с помощью связанных списков**

В ***объединяемой пирамиде*** (mergable heap) поддерживаются следующие операции: MAKE_HEAP (создание пустой пирамиды), INSERT, MINIMUM, EXTRACT_MINIMUM и UNION[1]. Покажите, как в каждом из перечисленных ниже случаев реализовать с помощью связанных списков объединяемые пирамиды. Постарайтесь, чтобы каждая операция выполнялась с максимальной эффективностью. Проанализируйте время работы каждой операции по отношению к размеру обрабатываемого динамического множества.

а) Списки отсортированы.

б) Списки не отсортированы.

в) Списки не отсортированы, и объединяемые динамические множества не перекрываются.

10-3. **Поиск в отсортированном компактном списке**

В упражнении 10.3-4 предлагается разработать компактную поддержку n-элементного списка в первых n ячейках массива. Предполагается, что все ключи различны, и что компактный список отсортирован, т.е. для всех $i = 1,2,\ldots,n$, таких что $next[i] \neq \text{NIL}$, выполняется соотношение $key[i] < key[next[i]]$. Покажите, что при данных предположениях

[1] *Поскольку в пирамиде поддерживаются операции* MINIMUM *и* EXTRACT_MINIMUM, *такую пирамиду можно назвать* **объединяемой неубывающей пирамидой** *(mergable min-heap). Аналогично, если бы в пирамиде поддерживались операции* MAXIMUM *и* EXTRACT_MAXIMUM, *ее можно было бы назвать* **объединяемой невозрастающей пирамидой** *(mergable max-heap).*

математическое ожидание времени поиска с помощью приведенного ниже рандомизированного алгоритма равно $O\left(\sqrt{n}\right)$:

```
COMPACT_LIST_SEARCH(L,n,k)
 1  i ← head[L]
 2  while i ≠ NIL and key[i] < k
 3      do j ← RANDOM(1,n)
 4        if key[i] < key[j] and key[j] ≤ k
 5          then i ← j
 6              if key[i] = k
 7                then return i
 8        i ← next[i]
 9  if i = NIL or key[i] > k
10      then return NIL
11      else return i
```

Если в представленной выше процедуре опустить строки 3–7, получится обычный алгоритм, предназначенный для поиска в отсортированном связанном списке, в котором индекс *i* пробегает по всем элементам в списке. Поиск прекращается в тот момент, когда происходит “обрыв” индекса *i* в конце списка или когда $key\left[i\right] \geq k$. В последнем случае, если выполняется соотношение $key\left[i\right] = k$, то понятно, что ключ *k* найден. Если же $key\left[i\right] > k$, то ключ *k* в списке отсутствует и поэтому следует прекратить поиск.

В строках 3–7 предпринимается попытка перейти к случайно выбранной ячейке *j*. Такой переход дает преимущества, если величина $key\left[j\right]$ больше величины $key\left[i\right]$, но не превышает значения *k*. В этом случае индекс *j* соответствует элементу списка, к которому рано или поздно был бы осуществлен доступ при обычном поиске. Поскольку список компактен, любой индекс *j* в интервале от 1 до *n* отвечает некоторому объекту списка и не может быть пустым местом из списка свободных позиций.

Вместо анализа производительности процедуры COMPACT_LIST_SEARCH мы проанализируем связанный с ним алгоритм COMPACT_LIST_SEARCH′, в котором содержатся два отдельных цикла. В этом алгоритме

используется дополнительный параметр t, определяющий верхнюю границу количества итераций в первом цикле:

```
COMPACT_LIST_SEARCH′(L,n,k,t)
 1  i ← head[L]
 2  for q ← 1 to t
 3      do j ← RANDOM(1,n)
 4        if key[i] < key[j] and key[j] ≤ k
 5          then i ← j
 6              if key[i] = k
 7                then return i
 8  while i ≠ NIL and key[i] < k
 8      do i ← next[i]
 9  if i = NIL or key[i] > k
10      then return NIL
11      else return i
```

Для простоты сравнения алгоритмов COMPACT_LIST_SEARCH(L,n,k) и COMPACT_LIST_SEARCH′(L,n,k,t) будем считать, что последовательность случайных чисел, которая возвращается процедурой RANDOM$(1,n)$, одна и та же для обоих алгоритмов.

а) Предположим, что в ходе работы цикла **while** в строках 2–8 процедуры COMPACT_LIST_SEARCH(L,n,k), выполняется t итераций. Докажите, что процедура COMPACT_LIST_SEARCH′(L,n,k,t) даст тот же ответ, и что общее количество итераций в циклах **for** и **while** этой процедуры не меньше t.

Пусть при работе процедуры COMPACT_LIST_SEARCH′(L,n,k,t) X_t — случайная величина, описывающая расстояние в связанном списке (т.е. длину цепочки из указателей *next*) от элемента с индексом i до искомого ключа k после t итераций цикла **for** в строках 2–7.

б) Докажите, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH′(L,n,k,t) равно $O\left(t + \mathrm{E}\left[X_t\right]\right)$.

в) Покажите, что $\mathrm{E}\left[X_t\right] \le \sum_{r=1}^{n} \left(1 - r/n\right)^t$. (*Указание:* воспользуйтесь уравнением (B.24)).

г) Покажите, что $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$.

д) Докажите, что $\mathrm{E}[X_t] \le n/(t+1)$.

е) Покажите, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH′(L,n,k,t) равно $O(t+n/t)$.

ж) Сделайте вывод о том, что математическое ожидание времени работы процедуры COMPACT_LIST_SEARCH равно $O(\sqrt{n})$.

з) Объясните, зачем при анализе процедуры COMPACT_LIST_SEARCH понадобилось предположение о том, что все ключи различны. Покажите, что случайные переходы не обязательно приведут к сокращению асимптотического времени работы, если в списке содержатся ключи с одинаковыми значениями.

Заключительные замечания

Прекрасными справочными пособиями по элементарным структурам данных являются книги Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [6] и Кнута (Knuth) [182]. Описание базовых структур данных и их реализации в конкретных языках программирования можно также найти во многих других учебниках. В качестве примера можно привести книги Гудрича (Goodrich) и Тамазии (Tamassia) [128], Мейна (Main) [209], Шаффера (Shaffer) [273] и Вейса (Weiss) [310, 312, 313]. В книге Гоннета (Gonnet) [126] приведены экспериментальные данные, описывающие производительность операций над многими структурами данных.

Начало использования стеков и очередей в информатике в качестве структур данных по сей день остается недостаточно ясным. Вопрос усложняется тем, что

соответствующие понятия существовали в математике и применялись на практике при выполнении деловых расчетов на бумаге еще до появления первых цифровых вычислительных машин. В своей книге [182] Кнут приводит относящиеся к 1947 году цитаты А.Тьюринга (A.M.Turing), в которых идет речь о разработке стеков для компоновки подпрограмм.

По-видимому, структуры данных, основанные на применении указателей, также являются “народным” изобретением. Согласно Кнуту, указатели, скорее всего, использовались еще на первых компьютерах с памятью барабанного типа. В языке программирования A-1, разработанном Хоппером (G.M. Hopper) в 1951 году, алгебраические формулы представляются в виде бинарных деревьев. Кнут считает, что указатели получили признание и широкое распространение благодаря языку программирования IPL-II, разработанному в 1956 году Ньюэлом (A. Newell), Шоу (J.C. Shaw) и Симоном (H.A. Simon). В язык IPL-III, разработанный в 1957 году теми же авторами, операции со стеком включены явным образом.