9.16. (За этим упражнением на самом деле стоят 24 упражнения.) Докажите правильность границ, установленных для 4 элементарных реализаций, представленных в таблице 9.1, используя для доказательства реализацию операций *вставить (insert)* и *удалить наибольший (remove the maximum)* из программы 9.2, и реализации, выполненные в упражнениях 9.7—9.9, а также формальное описание методов реализации других операций. Что касается операций *удалить (remove)*, *изменить приоритет (change priority)* и *объединить (join)*, предположите, что имеется средство, обеспечивающее прямой доступ к объекту ссылки.

9.2. Пирамидальная структура данных

Основной темой настоящей главы является простая структура данных, получившая название *сортирующего дерева (heap)*, которая может эффективно поддерживать основные операции в очереди по приоритетам. В сортирующем дереве записи хранятся в виде массива таким образом, что каждый ключ обязательно принимает значение большее, чем значения двух других ключей, занимающих относительно него строго определенные положения. В свою очередь, каждый из этих ключей должен быть больше, чем два других определенных ключа и т.д. Подобное упорядочение легко обнаружить, если рассматривать эти ключи как входящие в бинарную древовидную структуру с ребрами, идущими от каждого ключа к двум другим ключам, о которых известно, что они меньше его по значению.

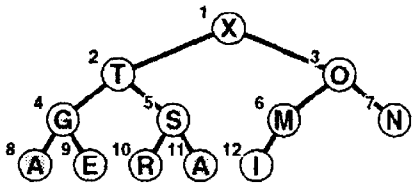

РИСУНОК 9.2. ПРЕДСТАВЛЕНИЕ ПОЛНОГО ДВОИЧНОГО ДЕРЕВА В ВИДЕ ПИРАМИДАЛЬНО УПОРЯДОЧЕННОГО МАССИВА

Если рассматривать элемент в позиции $\lfloor i/2 \rfloor$ массива как родителя элемента в позиции i, для $2 \le i \le N$ (или, что эквивалентно, считая i-й элемент родителем $2i$-го и $2i+1$-го элементов), то становится возможным удобное представление совокупности элементов массива в виде дерева. Такое соответствие эквивалентно нумерации полного двоичного дерева (элементы на нижнем уровне нумеруются, начиная с самого левого) по уровням. Дерево называется пирамидально упорядоченным, если ключ любого заданного узла больше или равен ключам узлов потомков. Сортирующее дерево есть представление в виде массива полного упорядоченного двоичного дерева. i-й элемент сортирующего дерева больше или равен по величине $2i$-го и $2i+1$-го элементов.

Определение 9.2. *Дерево называется **пирамидально упорядоченным** (heap-ordered), если ключ в каждом его узле больше или равен ключам всех потомков этого узла (если таковые имеются). Эквивалентная формулировка: ключ в каждом узле пирамидально упорядоченного дерева меньше или равен ключу узла, который является родителем данного узла.*

Лемма 9.1. *Ни один из узлов пирамидально упорядоченного дерева не может иметь ключа, большего чем ключ корня дерева.*

На любое дерево можно наложить ограничения, обусловленные пирамидальной упорядоченностью. Однако особенно удобно пользоваться *полным бинарным деревом (complete binary tree)*. Из главы 3 известно, что мы можем начертить такую структуру, поместив в верхней части страницы корневой узел, а затем спускаясь вниз по странице и перемещаясь слева направо, подсоединять к каждому конкретному узлу предыдущего уровня два узла текущего уровня до тех пор, пока не будут помещены все N узлов. Достаточ-

но просто представить полное бинарное дерево в виде массива, поместив корневой узел в позицию 1, его потомков в позицию 2 и 3, узлы следующего уровня в позиции 4, 5, 6, 7 и т.д., как показано на рис. 9.2.

Определение 9.3. *Сортирующее дерево есть совокупность узлов с ключами, образующих полное пирамидально упорядоченное бинарное дерево, представленное в виде массива.*

Можно было бы воспользоваться связным представлением пирамидально упорядоченных деревьев, но полные деревья предоставляют возможность задействовать компактное представление в виде массива, в котором легко переходить с некоторого узла к его родителю или к его предкам без необходимости поддержки явных связей. Родителя узла, находящегося в позиции i, необходимо искать в позиции $\lfloor i / 2 \rfloor$, и, соответственно, два потомка узла в позиции i находятся в позициях $2i$ и $2i + 1$. При подобной организации прохождение по такому дереву выполняется проще, чем если бы это дерево было реализовано в связном представлении, поскольку в таком случае могут понадобиться связи дерева, принадлежащие каждому ключу, чтобы иметь возможность перемещаться вверх и вниз по дереву (каждый элемент будет иметь один указатель на родителя и один указатель на каждого потомка). Полные бинарные деревья, представленные в виде массивов, являются жесткими структурами, но все же обладают достаточной гибкостью, чтобы позволить реализовать эффективные алгоритмы, манипулирующие очередями по приоритетам.

В разделе 9.3 мы убедимся в том, что можем воспользоваться сортирующими деревьями для реализации всех операций на очередях по приоритетам (за исключением операции *объединить*) таким образом, что на свое выполнение они в худшем случае потребуют логарифмическое время. Все такие реализации функционируют вдоль некоторого пути в сортирующем дереве (перемещаясь от родителя к потомкам по направлению вниз или от потомка к родителю по направлению вверх, не меняя при этом направления движения). Как уже было показано в главе 3, все пути в полном дереве, состоящем из N узлов, содержат в себе порядка $\lg N$ узлов: примерно $N/2$ узлов находятся на самом нижнем уровне, $N/4$ узлов — потомки которых занимают нижний уровень, $N/8$ узлов — "внуки" которых занимают нижний уровень и т.д. Каждое следующее поколение узлов содержит наполовину меньше узлов, чем предыдущее, всего же может быть максимум $\lg N$ поколений.

Можно также воспользоваться явными связными представлениями древовидной структуры для разработки эффективных реализаций операций над очередями по приоритетам. В качестве примеров рассматриваются полные деревья с тремя связями (см. раздел 9.5), сортировка повторной выборкой (см программу 5.19) и биномиальные очереди (см. раздел 9.7). Как и в случае простых стеков и очередей, одна из главных причин, побуждающих рассматривать связные представления, заключается в том, что они освобождают от необходимости заранее знать максимальные размеры очереди, что требуется в обязательном порядке в случае представления в виде массива. Мы также можем извлечь в некоторых ситуациях определенную пользу из гибкости, обеспечиваемой связными структурами, при разработке эффективных алгоритмов. Читателям, которые не имеют достаточного опыта использования явных древовидных структур, мы настоятельно рекомендуем прочитать главу 12, чтобы изучить базовые методы реализации даже более важных абстрактных типов данных, таких как *сим-*

вольные таблицы, прежде чем иметь дело со связными представлениями деревьев, рассматриваемых в упражнениях этой главы и в разделе 9.7. Однако внимательное изучение связных структур можно оставить до второго чтения, поскольку основной темой настоящей главы является сортирующее дерево (представление в виде массива без связей полного пирамидально упорядоченного дерева).

Упражнения

▷ **9.17.** Является ли массив, отсортированный в нисходящем порядке, сортирующим деревом?

○ **9.18.** Наибольший элемент сортирующего дерева должен появиться в позиции 1, а второй наибольший элемент должен занимать позицию 2 или 3. Представьте список позиций в сортирующем дереве из 15 элементов, в котором k-й наибольший элемент (i) может появиться и (ii) не может появиться, для $k = 2, 3, 4$ (в предположении, что все значения попарно различны).

● **9.19.** Решить задачу 9.18 для общего значения k как функции от N, представляющего собой размер сортирующего дерева.

● **9.20.** Решить задачи 9.18 и 9.19 для k-го *наименьшего* элемента.

9.3. Алгоритмы для сортирующих деревьев

Все алгоритмы очередей по приоритетам для сортирующих деревьев работают таким образом, что сначала вносят простое изменение, способное нарушить структуру пирамиды, затем выполняют проход вдоль пирамиды, внося при этом в сортирующее дерево такие изменения, которые гарантируют, что структура сортирующего дерева сохраняется везде. Этот процесс иногда называют *установлением пирамидального порядка (heapifying)*. Возможны два случая. Когда приоритет какого-либо узла увеличивается (или в нижний уровень сортирующего дерева добавляется новый узел), мы должны двигаться вверх по дереву, чтобы восстановить структуру сортирующего дерева. Когда приоритеты каких-либо узлов уменьшаются (например, при замене корневого узла новым узлом), необходимо пройти вниз по дереву, чтобы восстановить структуру сортирующего дерева. Для начала обсудим, как реализовать две эти базовые функции, а затем подумаем, как их использовать в различных операциях с очередями по приоритетам.

Если свойства сортирующего дерева нарушены из-за того, что ключ некоторого узла становится больше ключа родительского узла, можно сделать шаг в направлении исправления этого нарушения, обменяв местами этот узел с его родителем. После обмена этот узел становится больше, чем оба его потомка (один из них — это прежний родитель, другой меньше, чем старый родитель, поскольку он был потомком этого узла), но все еще может оставаться больше своего родителя. Мы можем устранить и это нарушение аналогичным способом и продвигаться далее вверх по дереву, пока не достигнем узла с наибольшим ключом, каковым является корень дерева. Пример описанного процесса показан на рис. 9.3. Программа проста и понятна, в ее основе лежит тот факт, что родитель узла, занимающего позицию k в сортирующем дереве, находится в позиции **k / 2** этого дерева. Программа 9.3 представляет собой реализацию функции, которая восстанавливает возможные нарушения, обусловленные

увеличением приоритета в заданном узле сортирующего дерева, благодаря продвижению вверх по дереву.

Программа 9.3. Восходящая установка структуры сортирующего дерева

Чтобы восстановить пирамидальную структуру после того, как приоритет одного из узлов сортирующего дерева изменился, мы продвигаемся вверх по дереву, меняя при необходимости местами узел в позиции *k* с его родителем (который находится в позиции **k/2**) и продолжаем этот процесс до тех пор, пока выполняется условие **a[k/2] < a[k]** или пока не выйдем в вершину сортирующего дерева.

```
template <class Item>
void fixUp(Item a[], int k)
 {
   while (k > 1 && a[k/2] < a[k])
     { exch (a[k], a[k/2]); k = k/2; }
 }
```

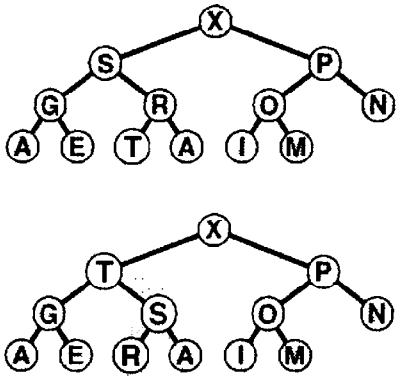

РИСУНОК 9.3. ВОСХОДЯЩАЯ УСТАНОВКА СТРУКТУРЫ СОРТИРУЮЩЕГО ДЕРЕВА

На верхнем дереве, показанном на диаграмме, установлен пирамидальный порядок, в который не вписывается узел T на нижнем уровне. Если мы поменяем T местами с его родителем, дерево остается пирамидально упорядоченным, за исключением разве что того случая, когда T оказывается больше своего нового родителя. Продолжая обмен местами узла T со своими родителями до тех пор, пока мы не выйдем на своем пути на корневой узел, либо на узел, который больше T, мы можем установить пирамидальный порядок во всем дереве. Мы можем использовать эту процедуру в качестве основы для операции ***insert*** *(вставить), выполняемой на сортирующих деревьях с целью восстановления пирамидального порядка после добавления в это дерево нового элемента (крайнюю правую позицию на нижнем уровне, вводя в дереве при необходимости новый уровень).*

Если свойство пирамидальности сортирующего дерева было нарушено в силу того, что ключ какого-либо узла становится меньше, чем один или оба ключа его потомков, выполняются шаги с целью устранения нарушения путем замены узла на больший из его двух потомков. Такая замена может вызвать нарушение свойств сортирующего дерева на узле-потомке; это нарушение устраняется аналогичным путем и выполняется продвижение вниз по дереву до тех пор, пока не будет достигнут узел, оба потомка которого меньше его самого, либо нижний уровень дерева. Пример процесса показан на рис. 9.4. Опять-таки, программный код учитывает то обстоятельство, что потомки узла сортирующего дерева в позиции **k** занимают в нем позиции **2k** и **2k+1**. Программа 9.4 содержит реализацию функции, которая восстанавливает сортирующее дерево после возможных нарушений по причине повышения приоритета заданного узла, перемещаясь вниз по этому дереву. Эта функция должна знать размер сортирующего дерева (N), чтобы иметь возможность отследить момент, когда будет достигнут нижний уровень дерева.

Обе эти операции не зависят от способа представления структуры дерева, если возможен доступ к родителям (восходящий метод) и к потомкам (нисходящий метод) любого узла. В случае восходящего метода мы перемещаемся вверх по дереву, заменяя ключ заданного узла ключом его родителя до тех пор, пока не выйдем на корневой узел или на родителя с большим (или равным) ключом. В случае нисходящего

метода мы перемещаемся вниз по дереву, заменяя ключ заданного узла большим из ключей его потомков до тех пор, пока не достигнем нижнего уровня или точки, в которой нет потомков с большими ключами. В обобщенном виде упомянутые операции применимы не только к бинарным деревьям, но и к любой древовидной структуре. Усовершенствованные алгоритмы, манипулирующие очередями по приоритетам, обычно ориентированы на древовидные структуры общего вида, но в то же время полагаются именно на эти базовые операции, обеспечивающие доступ к наибольшим ключам этой структуры, сосредоточенным в ее верхней части.

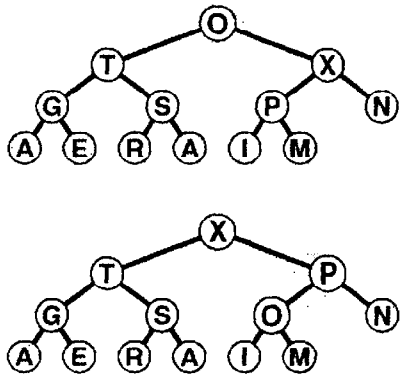

РИСУНОК 9.4. НИСХОДЯЩАЯ УСТАНОВКА СТРУКТУРЫ СОРТИРУЮЩЕГО ДЕРЕВА

Дерево, изображенное в верхней части диаграммы, почти везде пирамидально упорядочено, исключением является корень дерева. Если заменить узел ***O*** *большим из его потомков (****X****), то рассматриваемое дерево приобретает пирамидальный порядок, за исключением поддерева с корнем в узле* ***O****. Продолжая обмен местами с большим из его двух потомков до тех пор, пока не будет достигнут нижний уровня пирамиды или точка, в которой* ***O*** *больше любого из своих потомков, можно восстановить условие пирамиды на всем дереве. Этой процедурой можно воспользоваться в качестве основы для операции* ***remove the maximum*** *(удалить наибольший) в сортирующем дереве с целью восстановить пирамидальный порядок после замены ключа в корне дерева на ключ, который находится на нижнем уровне в крайней правой позиции.*

Если вообразить себе, что некоторое сортирующее дерево представляет иерархию некой корпорации, в котором каждый потомок представляет подчиненное подразделение (и каждый родитель представляет собой вышестоящее подразделение), то эти операции допускают любопытные аналогии. Восходящий метод соответствуют появлению на сцене нового перспективного управляющего, который продвигается по служебной лестнице с одного поста на другой, более высокий (обмениваясь служебными обязанностями с любым начальником, имеющим более низкую квалификацию) до тех пор, пока этот новый работник не столкнется с более квалифицированным начальником. Нисходящий метод аналогичен ситуации, когда президента некоторой компании сменяет на его посту некто, уступающий ему по квалификации. Если какой-либо из подчиненных президента, обличенный наибольшими полномочиями, превосходит по совокупности качеств нового работника, они обмениваются своими обязанностями, и мы двигаемся вниз по служебной лестнице, понижая в должности нового работника и повышая других, пока не будет достигнут уровень компетенции нового работника, когда не остается ни одного подчиненного, превосходящего нового работника по квалификации (этот идеализированный сценарий в реальной жизни встречается редко). Продолжая эту аналогию, движение по сортирующему дереву часто будет называться *продвижением*.

Программа 9.4. Нисходящая установка структуры сортирующего дерева

Чтобы восстановить пирамидальную структуру в случае, когда приоритет узла понижается, мы двигаемся вниз по сортирующему дереву, меняя при необходимости местами узел в позиции **k** с большим из его двух потомков, и останавливаемся, когда узел в позиции **k** не превышает какой-либо из двух своих потомков, или когда

достигнут нижний уровень. Обратите внимание на то обстоятельство, что если **N** есть четное число и **k** равно **N/2**, то узел в позиции **k** имеет только одного потомка — этот случай требует особого подхода!

Внутренний цикл в этой программе имеет два четко определенных выхода: один для случая, когда достигнут нижний уровень сортирующего дерева, а другой для случая, когда условия сортирующего дерева удовлетворяются где-то внутри дерева. Этот случай может служить прототипным примером необходимости существования конструкции **break**.

```
template <class Item>
void fixDown(Item a[ ], int k, int N)
 {
  while (2*k <= N)
   { int j = 2*k;
       if (j < N && a[j] < a[j+1]) j++;
       if (!(a[k] < a[j])) break;
       exch (a[k], a[j]); k = j;
   }
 }
```

Программа 9.5. Очередь по приоритетам на базе сортирующего дерева

Чтобы реализовать операцию вставки, мы увеличиваем **N** на 1, добавляем новый элемент в конец сортирующего дерева, а затем при помощи функции **fixUp** восстанавливаем пирамидальный порядок. При выполнении функции **getmax** (получить наибольший) размер сортирующего дерева должен быть уменьшен на 1, таким образом, мы выбираем в качестве возвращаемого значения величину **pq[1]**, затем уменьшаем размер сортирующего дерева на 1, перемещая значение **pq[N]** в **pq[1]**, после чего выполняем функцию **fixDown** с целью восстановить в дереве пирамидальный порядок. Реализации конструктора и функции **empty** предельно тривиальны. Первая позиция **pq[0]** массива здесь не используется, но она может быть задействована в качестве сигнальной метки в других реализациях.

```
template <class Item>
class PQ
   {
        private:
              Item *pq;
              Int N;
        Public:
              PQ(int maxN)
                    {pq = new Item[maxN+1]; N = 0; }
              int empty( ) const
                    {return N==0; }
              void insert (Item, item)
                    {pq[++N] = item; fixUp(pq, N;}
              Item getmax()
                    {
                          exch(pq[1], pq[n] );
                          fixDown(pq, 1, N-1 );
                          return pq[N--];
                    }
   };
```

Как показывает программа 9.5, эти две основные операции позволяют построить эффективную реализацию АТД очереди по приоритетам. Если очередь по приоритетам представлена как пирамидально упорядоченный массив, использование операции

вставить (insert) равносильно добавлению нового элемента в конец массива и перемещение этого элемента по массиву с целью восстановления структуры сортирующего дерева; операция *удалить наибольший (remove the maximum)* равносильна удалению наибольшего элемента из вершины дерева с последующим размещением элемента, находящегося в конце дерева, в его вершине и перемещением его вниз вдоль массива с целью восстановления структуры сортирующего дерева.

Лемма 9.2. *Операции* ***вставить*** *(insert) и* ***удалить наибольший*** *(remove the maximum) для абстрактного типа данных могут быть реализованы на пирамидально упорядоченных деревьях таким образом, что операция* ***вставить*** *требует для своего выполнения на очереди, состоящей из N элементов, не более $\lg N$ сравнений, а операция* ***удалить наибольший*** *— не более $2 \lg N$ сравнений.*

Обе операции предусматривают перемещение вдоль пути между корнем и нижним уровнем дерева, при этом ни один путь вдоль сортирующего дерева, состоящего из N элементов, не содержит более $\lg N$ элементов (см. например, лемму 5.8 и упражнение 5.77). Операция *удалить наибольший* требует двух сравнений для каждого узла: одну для того, чтобы найти потомка с большим ключом, другую — для того, чтобы принять решение, нужно ли продвигать этого потомка.

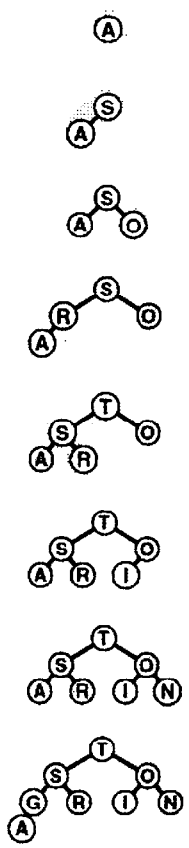

РИСУНОК 9.5. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СВЕРХУ ВНИЗ

Представленная последовательность диаграмм описывает операцию вставки ключей ***A S O R T I N G*** *в первоначально пустое сортирующее дерево. Новые узлы добавляются в сортирующее дерево на нижнем уровне в направлении слева направо. Каждая вставка затрагивает только узлы, которые находятся на пути между точкой вставки и корнем, поэтому затраты в худшем случае пропорциональны логарифму размера сортирующего дерева.*

На рис. 9.5 и 9.6 показаны примеры, в рамках которых выполняется построение сортирующего дерева путем последовательной вставки элементов в первоначально пустое сортирующее дерево. В представлении сортирующего дерева в виде массива, которое использовалось ранее, этот процесс соответствует пирамидальному упорядочению массива, при этом каждый раз, когда мы переходим к новому элементу, размер сортирующего дерева увеличивается на 1, а для восстановления пирамидального порядка используется функция **fisUp**. В худшем случае на выполнение этого процесса требуется время, пропорциональное $N \lg N$ (когда каждый новый элемент превосходит по величине все предыдущие, т.е. он проделывает весь путь к корню дерева), но в среднем на это требуется линейное время (новый элемент произвольной природы поднимается всего лишь на несколько уровней). В разделе 9.4 будет рассматриваться способ построения сортирующего дерева (установления в массиве пирамидального порядка) за линейное время в худшем случае.

Базовые процедуры **fixUp** и **fixDown** из программ 9.3 и 9.4 позволяют получить прямую реализацию операций *изменить приоритет (change priority)* и *удалить (remove)*. Для изменения приоритета элемента, находящегося где-то в середине сортирующего дерева, применяется процедура **fixUp** для перемещения вверх по дереву, если приоритет элемента увеличивается, и процедура **fixDown** для перемещения вниз по дереву, если приоритет уменьшается. Полная реализация этих процедур применительно к конкретным элементам данных имеет смысл, только если для каждого элемента имеется дескриптор, отслеживающий место этого элемента в структуре данных. Мы подробно рассмотрим реализации, которые выполняют эти действия, в разделах 9.5—9.7.

Лемма 9.3. *Операции **изменить приоритет** (change priority), **удалить** (remove) и **изменить приоритет** (change priority) для АТД очередь по приоритетам могут быть реализованы через сортирующие деревья, такие, что для любой из указанных выше операций требуется выполнение не более чем* $2 \lg N$ *операций сравнения на очереди из* N *элементов.*

Поскольку эти операции требуют наличия специальных дескрипторов, отложим рассмотрение реализаций, поддерживающих эти операции, до раздела 9.7 (см. программу 9.12 и рис. 9.14). Все они предусматривают движение по сортирующему дереву вдоль одного пути, возможно, сверху вниз или снизу вверх в худшем случае.

Специально обращаем внимание на тот факт, что в этот список не включена операция *объединить (join)*. Эффективное объединение двух очередей по приоритетам, по-видимому, потребует более сложной структуры данных, которая будет подробно рассматриваться в разделе 9.7. С другой стороны, представленный здесь простой метод, в основу которого положен пирамидальный порядок, вполне достаточен для большинства различных приложений. Он использует минимальное дополнительное пространство памяти и обеспечивает высокую эффективность выполнения операций за исключением тех случаев, когда часто и на больших объемах данных выполняются операции *объединить*.

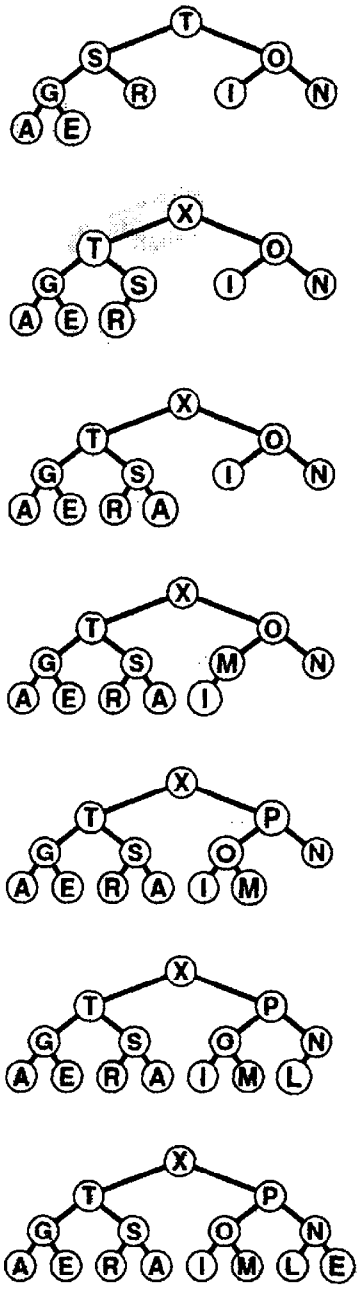

РИСУНОК 9.6. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СВЕРХУ ВНИЗ (ПРОДОЛЖЕНИЕ)

Представленная последовательность диаграмм отображает процедуру вставки ключей ***E X A M P L E*** *в сортирующее дерево, построение которого было начато на рис. 9.5. Общая стоимость построения сортирующего дерева размером* N *составляет* $\lg 1 + \lg 2 + \ldots + \lg N$, *что меньше* $N \lg N$.

Как уже отмечалось выше и как продемонстрировано в программе 9.6, любую очередь по приоритетам можно использовать для построения еще одного метода сортировки. Мы просто вставляем все подлежащие сортировке ключи в очередь по приоритетам, а затем многократно используем операцию *удалить наибольший*, чтобы удалять их в порядке убывания приоритетов. Такое использования очереди по приоритетам, представленной в виде неупорядоченного списка, соответствует выполнению сортировки выбором, а применение упорядоченного списка соответствует выполнению сортировки вставками.

Программа 9.6. Сортировка с использованием очереди по приоритетам

Чтобы отсортировать подмассив **a[1],..., a[r]**, используя для этой цели АТД *очереди по приоритетам*, следует при помощи функции **insert** (вставить) поместить элементы в очередь по приоритетам, а затем с использованием функции **getmax** (найти наибольший) удалять их в порядке убывания значений приоритетов. Подобный алгоритм сортировки выполняется за время, пропорциональное $N \lg N$, но при этом он требует дополнительного объема памяти, пропорционального количеству сортируемых элементов N (в очереди по приоритетам).

```
#include "PQ.cxx"
template <class Item>
void PQsort(Item a[], int l, int r)
 { int k;
   PQ<Item> pq(r-l+1);
   for (k = l; k <= r; k++) pq.insert(a[k]);
   for (k = r; k >= l; k--)
     a[k] = pq.getmax();
 }
```

На рис. 9.5 и 9.6. показан пример первой фазы (процесс построения), на которой используется реализация очереди по приоритетам с пирамидальным порядком; на рис. 9.7 и 9.8 представлена вторая фаза (которую будем называть *нисходящей сортировкой* — sortdown). В условиях практических применений этот метод не выглядит особенно элегантно, поскольку он без особой необходимости копирует сортируемые элементы (в очереди по приоритетам). Действительно, выполнение N последовательных вставок — не самый эффективный способ построения сортирующего дерева, состоящего из N элементов. В следующем разделе при обсуждении реализации классического алгоритма пирамидальной сортировки этим двум вопросам уделяется особое внимание.

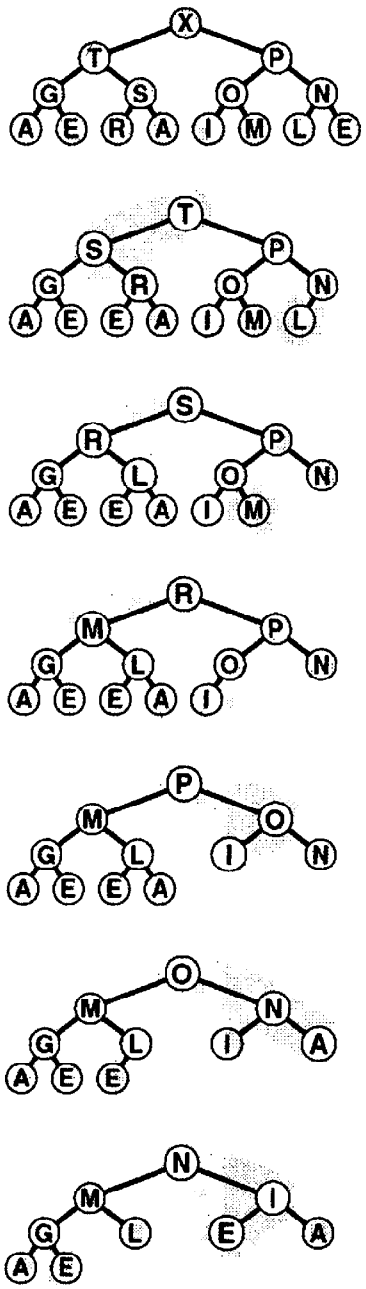

РИСУНОК 9.7. СОРТИРОВКА В СОРТИРУЮЩЕМ ДЕРЕВЕ

После замены наибольшего элемента сортирующего дерева самым правым элементом нижнего уровня можно восстановить пирамидальный порядок за счет перемещения по пути из корня на нижний уровень дерева.

Упражнения

▷ **9.21**. Построить сортирующее дерево, которое получается после того как ключи **E A S Y Q U E S T I O N** вставлены в первоначально пустое сортирующее дерево.

▷ **9.22**. Воспользовавшись соглашением, принятым в упражнении 9.1, представьте последовательность сортирующих деревьев, полученных в результате выполнения операций

PRIO*RI*T*Y***QUE***U*E**

на первоначально пустом сортирующем дереве.

9.23. Поскольку примитив **exch** используется в операциях по установке пирамидального порядка, элементы загружаются и запоминаются в два раза чаще, чем это необходимо. Предложите более эффективные реализации, которые не сталкиваются с такой проблемой, в духе сортировки вставками.

9.24. Почему мы не пользуемся сигнальной меткой, чтобы избежать проверку **j<N** в функции **fixDown**?

○ **9.25**. Добавить операцию *заменить наибольший (replace the maximum)* в реализацию очереди по приоритетам с пирамидальным порядком в программе 9.5. Рассмотреть случай, когда добавляемое значение больше всех остальных значений в очереди. *Совет*: К элегантному решению приводит использование элемента **pq[0]**.

9.26. Каким является минимальное количество перемещений ключей, которое потребуется выполнить на сортирующем дереве в операции *удалить наибольший (remove maximum)*? Приведите пример сортирующего дерева, содержащего 15 элементов, для которого достигается этот минимум.

9.27. Каким является минимальное количество ключей, которое потребуется переместить в процессе выполнения на сортирующем дереве трех последовательных операций *удалить наибольший*? Приведите пример сортирующего дерева из 15 элементов, для которого достигается этот минимум.

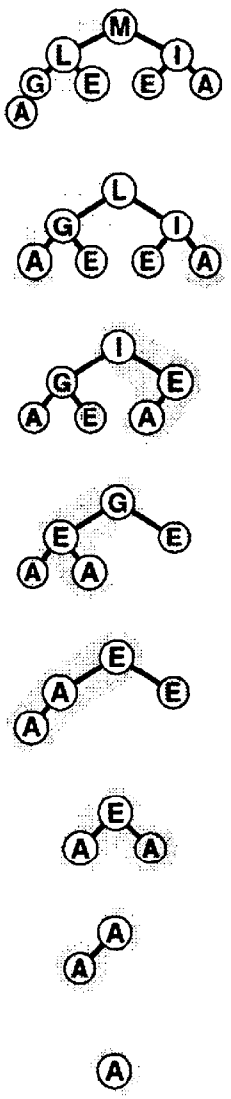

РИСУНОК 9.8. СОРТИРОВКА ИЗ СОРТИРУЮЩЕГО ДЕРЕВА

Представленная здесь последовательность диаграмм отображает удаление остальных ключей из сортирующего дерева на рис. 9.7. Даже если каждый элемент возвращается на нижний уровень, то общие затраты на выполнение этой фазы сортировки не больше $\lg N + \ldots + \lg 2 + \lg 1$, *что меньше* $N \lg N$.