13.3. Реализация класса Heap

Здесь мы подробно обсудим операции вставки и удаления для пирамид, а также методы FilterUp и FilterDown. Эти вспомогательные методы отвечают за реорганизацию пирамиды при ее создании или изменении.

Операция включения элемента в пирамиду. Вначале элемент добавляется в конец списка. Однако при этом может нарушиться условие пирамидальности. Если новый элемент имеет значение меньшее, чем у его родителя, узлы меняются местами. Возможные ситуации представлены на следующем рисунке.

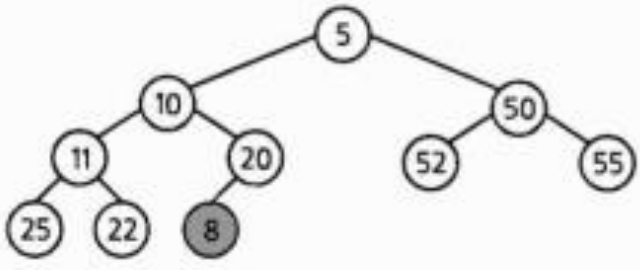
Новый элемент является левым сыном меньшим, чем родительский узел

Новый элемент является правым сыном меньшим, чем родительский узел

Этот обмен восстанавливает условие пирамидальности для данного родительского узла, однако может нарушить условие пирамидальности для высших уровней дерева. Теперь мы должны рассмотреть нового родителя как сына и проверить условие пирамидальности для более старшего родителя. Если новый элемент меньше, следует переместить его выше. Таким образом новый элемент поднимается вверх по дереву вдоль пути, проходящего через его предков. Рассмотрим следующий пример для 9-элементной пирамиды H:

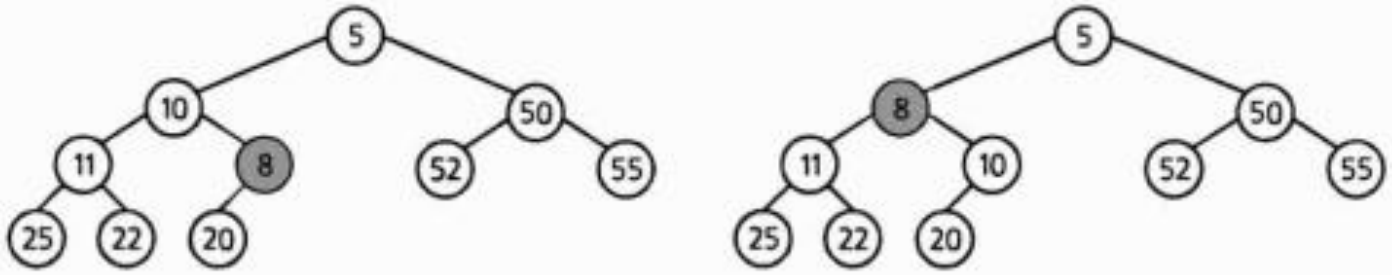
```
H.Insert(8);        // вставить элемент 8 в пирамиду
```

Вставить 8 в A[9]. Вставить новый элемент в конец пирамиды. Эта позиция определяется индексом heapsize, хранящим текущее число элементов в пирамиде.



H[9]

Отправить значение 8 по пути предков. Сравнить 8 с родителем 20. Поскольку сын меньше своего родителя, поменять их значения местами (A). Продолжить движение по пути предков. Теперь элемент 8 меньше своего родителя H[1]=10 и поэтому меняется с ним местами. Процесс завершается, так как следующий родитель удовлетворяет условию пирамидальности.



(A) (B)

Процесс включения элементов сканирует путь предков и завершается, встретив "маленького" (меньше чем новый элемент) родителя или достигнув корневого узла. Так как у корневого узла нет родителя, новое значение помещается в корень.

Чтобы поместить узел в правильную позицию, операция вставки использует метод FilterUp.

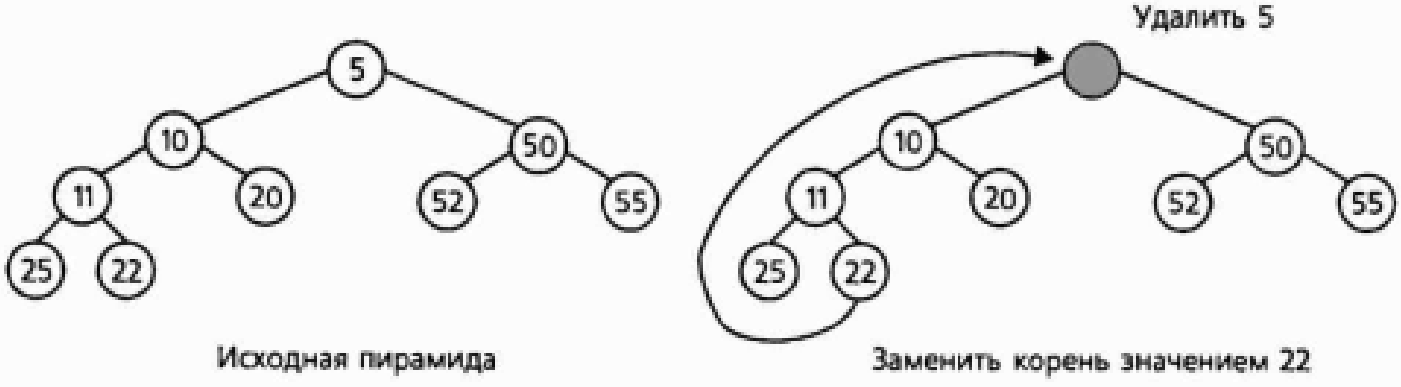
```
// утилита для восстановления пирамиды. начиная с индекса i,
// подниматься вверх по дереву, переходя от предка к предку.
// менять элементы местами, если сын меньше родителя
template <class T>
void Heap<T>::FilterUp (int i)
{
  int currentpos, parentpos;
  T target;

  // currentpos — индекс текущей позиции на пути предков.
  // target — вставляемое значение, для которого выбирается
  // правильная позиция в пирамиде
  currentpos = i;
  parentpos = (i-1)/2;
  target = hlist[i];

  // подниматься к корню по пути родителей
  while (currentpos != 0)
  {
    // если родитель <= target, то все в порядке.
    if (hlist[parentpos] <= target)
      break;
    else
    // поменять местами родителя с сыном и обновить индексы
    // для проверки следующего родителя
    {
      // переместить данные из родительской позиции в текущую.
      // назначить родительскую позицию текущей.
      // проверить следующего родителя
      hlist[currentpos] = hlist[parentpos];
      currentpos = parentpos;
      parentpos = (currentpos-1)/2;
    }
  }
  // правильная позиция найдена. поместить туда target
  hlist[currentpos] = target;
}
```

Открытый метод Insert проверяет сначала заполненность пирамиды, а затем начинает операцию включения. После записи элемента в конец пирамиды вызывается FilterUp для ее реорганизации.

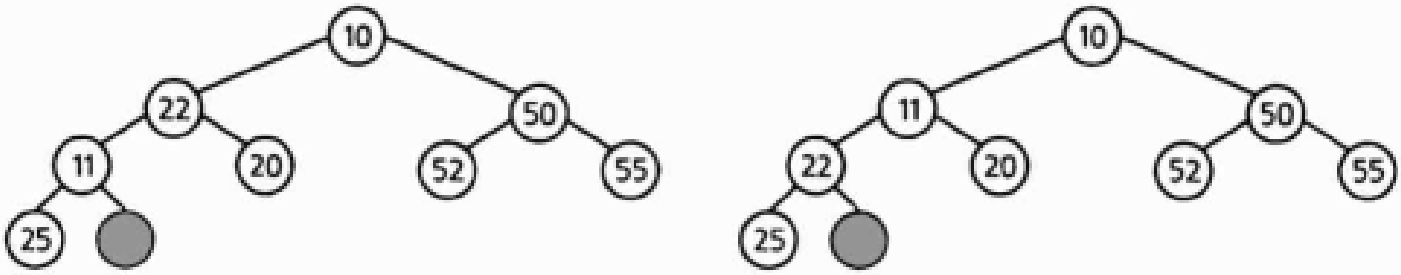
```
// вставить в пирамиду новый элемент и восстановить ее структуру
template <class T>
void Heap<T>::Insert(const T& item)
{
  // проверить, заполнена ли пирамида и выйти, если да
  if (heapsize == maxheapsize)
    error ("Пирамида заполнена");
  // записать элемент в конец пирамиды и увеличить heapsize.
  // вызвать FilterUp для восстановления пирамидального упорядочения
  hlist[heapsize] = item;
  FilterUp(heapsize);
  heapsize++;
}
```

Удаление из пирамиды. Данные удаляются всегда из корня дерева. После такого удаления корень остается ничем не занятым и сначала заполняется последним элементом пирамиды. Однако такая замена может нарушить условие пирамидальности. Поэтому требуется пробежать по всем меньшим потомкам и найти подходящее место для только что помещенного в корень элемента. Если он больше любого своего сына, мы должны поменять местами этот элемент с его наименьшим сыном. Движение по пути меньших сыновей продолжается до тех пор, пока элемент не займет правильную позицию в качестве родителя или пока не будет достигнут конец списка. В последнем случае элемент помещается в листовой узел. Например, в приведенной ниже пирамиде удаляется корневой узел 5.

Удалить корневой узел 5 и заменить его последним узлом 22. Последний элемент пирамиды копируется в корень. Новый корень может не удовлетворять условию пирамидальности, и требуется отправиться по пути, проходящему через меньших сыновей, чтобы подыскать для нового корня правильную позицию.



Передвигать число 22 от корня вниз по пути, проходящему через меньших сыновей. Сравнить корень 22 с его сыновьями. Наименьший из двух сын H[1] меньше, чем 22, поэтому следует поменять их местами (A). Находясь теперь на первом уровне, новый родитель сравнивается со своими сыновьями H[3] и H[4]. Наименьший из них имеет значение 11 и поэтому должен поменяться местами со своим родителем (B). Теперь дерево удовлетворяет условию пирамидальности.



(A) (B)

Метод Delete. Чтобы поместить узел в правильную позицию, операция удаления использует метод FilterDown. Эта функция получает в качестве параметра индекс i, с которого начинается сканирование. При удалении метод FilterDown вызывается с параметром 0, так как замещающее значение копируется из последнего элемента пирамиды в ее корень. Метод FilterDown используется также конструктором для построения пирамиды.

```
// утилита для восстановления пирамиды. начиная с индекса i,
// менять местами родителя и сына так, чтобы поддерево,
// начинающееся в узле i, было пирамидой
template <class T>
void Heap<T>::FilterDown (int i)
{
  int currentpos, childpos;
  T target;

  // начать с узла i и присвоить его значение переменной target
  currentpos = i;
  target = hlist[i];

  // вычислить индекс левого сына и начать движение вниз по пути,
  // проходящему через меньших сыновей до конца списка
  childpos = 2 * i + 1;
  while (childpos < heapsize)  // пока не конец списка
  {
    // индекс правого сына равен childpos+1. присвоить переменной
    // childpos индекс наименьшего из двух сыновей
    if ((childpos+1 < heapsize) &&
        (hlist[childpos+1] <= hlist[childpos]))
      childpos = childpos + 1;

    // если родитель меньше сына, пирамида в порядке. выход
    if (target <= hlist[childpos])
      break;
    else
    {
      // переместить значение меньшего сына в родительский узел.
      // теперь позиция меньшего сына не занята
      hlist[currentpos] = hlist[childpos];

      // обновить индексы и продолжить сканирование
      currentpos = childpos;
      childpos = 2 * currentpos + 1;
    }
  }
  // поместить target в только что ставшую незанятой позицию
  hlist[currentpos] = target;
}
```

Открытый метод Delete копирует значение из корневого узла во временную переменную, а затем замещает корень последним элементом пирамиды. После

этого heapsize уменьшается на единицу. FilterDown реорганизует пирамиду. Значение, сохраненное во временной переменной, возвращается клиенту.

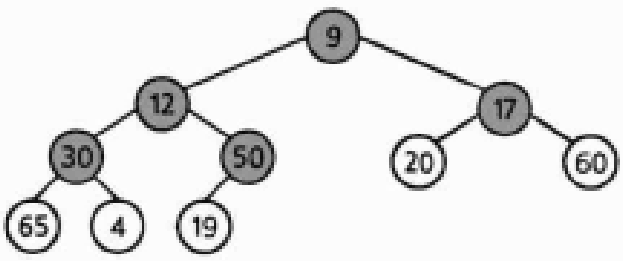
```
// возвратить значение корневого элемента и обновить пирамиду.
// попытка удаления элемента из пустой пирамиды влечет за собой
// выдачу сообщения об ошибке и прекращение программы
template <class T>
T Heap<T>::Delete(void)
   {
       T tempitem;

       //  проверить, пуста ли пирамида
       if (heapsize == 0)
       error ("Пирамида    пуста");

       // копировать корень в tempitem. заменить корень  последним элементом
       // пирамиды и произвести декремент переменной  heapsize
       tempitem = hlist[0];
       hlist[0] = hlist[heapsize-1];
       heapsize--;

       // вызвать FilterDown для установки нового значения корня
       FilterDown(0);

       // возвратить исходное значение корня
       return tempitem;
   }
```

Преобразование массива в пирамиду. Один из конструкторов класса Heap использует существующий массив в качестве входного списка и преобразует его в пирамиду. Ко всем нелистовым узлам применяется метод FilterDown. Индекс последнего элемента пирамиды равен n-1. Индекс его родителя равен

$$\text{currentpos} = \frac{(n-1)-1}{2} = \frac{n-2}{2}$$

и определяет последний нелистовой узел пирамиды. Этот индекс является начальным для преобразования массива. Если применить метод FilterDown ко всем индексам от currentpos до 0, то можно гарантировать, что каждый родительский узел будет удовлетворять условию пирамидальности. В качестве примера рассмотрим целочисленный массив

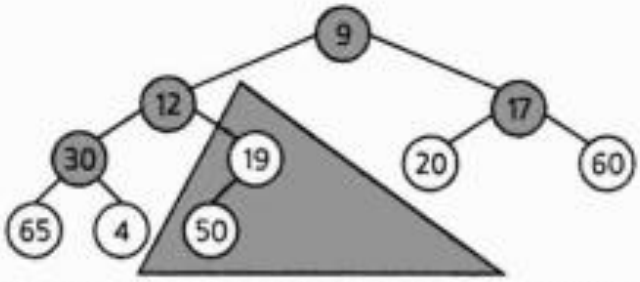
int A[10] = {9, 12, 17, 30, 50, 20, 60, 65, 4, 19}
Индексы листьев: 5, 6, ..., 9
Индексы родительских узлов: 4, 3, ..., 0



Исходный список

Приведенные ниже рисунки иллюстрируют процесс преобразования пирамиды. Для всех вызовов FilterDown соответствующее поддерево выделено на рисунках треугольником.

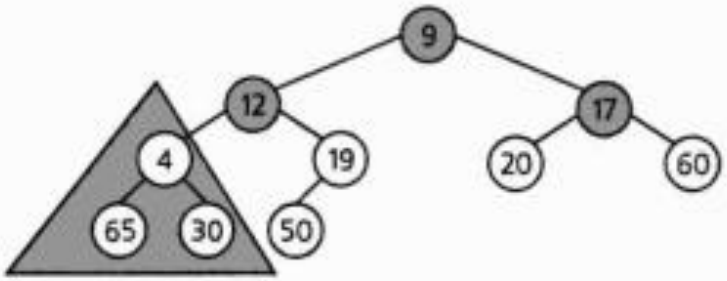
FilterDown(4). Родитель H[4] = 50 больше своего сына H[9] = 19 и поэтому должен поменяться с ним местами (A).



Поставить на место число 50 с помощью FilterDown(4)

(A)

FilterDown(3). Родитель H[3] = 30 больше своего сына H[8] = 19 и поэтому должен поменяться с ним местами (B).



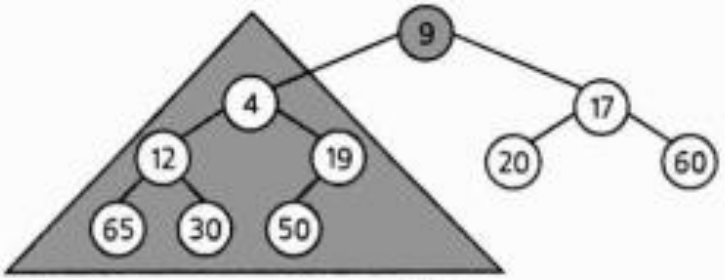
Поставить на место число 30 с помощью FilterDown(3)

(B)

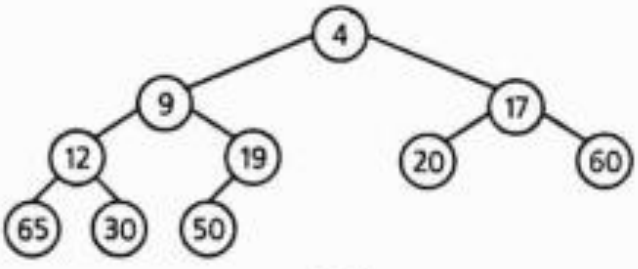
На уровне 2 родитель H[2] = 17 уже удовлетворяет условию пирамидальности, поэтому вызов FilterDown(2) не производит никаких перестановок.

FilterDown(1). Родитель H[1] = 12 больше своего сына H[3] = 19 и поэтому должен поменяться с ним местами (C).

FilterDown(0). Процесс прекращается в корневом узле. Родитель H[0] = 9 должен поменяться местами со своим сыном H[1]. Результирующее дерево является пирамидой.



(C)



(D)

КОНСТРУКТОР

```
// конструктор преобразует исходный массив в пирамиду.
// этот массив и его размер передаются в качестве параметров
template <class T>
Heap<T>::Heap(T arr[], int n)
{
  int j, currentpos;

  // n <= 0 является недопустимым размером массива
  if (n <= 0)
    error ("Неправильная размерность массива");

  // использовать n для установки размера пирамиды и максимального размера пирамиды.
  // копировать массив arr в список пирамиды
  maxheapsize = n;
  heapsize = n;
  hlist = arr;

  // присвоить переменной currentpos индекс последнего родителя.
  // вызывать FilterDown в цикле с индексами currentpos..0
  currentpos = (heapsize-2)/2;
  while (currentpos >= 0)
  {
    // выполнить условие пирамидальности для поддерева
    // с корнем hlist[currentpos]
    FilterDown(currentpos);
    currentpos--;
  }
  // присвоить флажку inArray значение True
  inArray = 1;
}
```

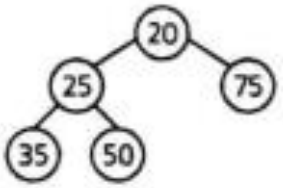
Приложение: пирамидальная сортировка

Пирамидальная сортировка имеет эффективность $O(n \log_2 n)$. Алгоритм использует тот факт, что наименьший элемент находится в корне (индекс 0) и что метод Delete возвращает это значение.

Для осуществления пирамидальной сортировки массива A объявите объект типа Heap с массивом A в качестве параметра. Конструктор преобразует A в пирамиду. Сортировка осуществляется последовательным исключением A[0] и включением его в A[N-1], A[N-2], ..., A[1]. Вспомните, что после исключения элемента из пирамиды элемент, бывший до этого хвостовым, замещает корневой и с этого момента больше не является частью пирамиды. Мы имеем возможность скопировать удаленный элемент в эту позицию. В процессе пирамидальной сортировки очередные наименьшие элементы удаляются и последовательно запоминаются в хвостовой части массива. Таким образом, массив A сортируется по убыванию. В качестве упражнения читателю предлагается построить класс максимальных пирамид, с помощью которого массив сортируется по возрастанию.

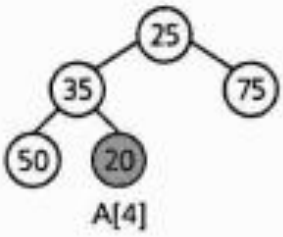
Пирамидальная сортировка пятиэлементного массива A осуществляется посредством следующих действий:

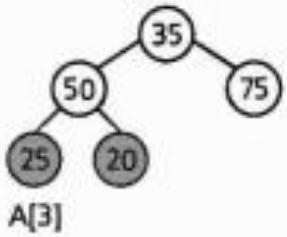
int A[] = {50, 20,75, 35, 25}



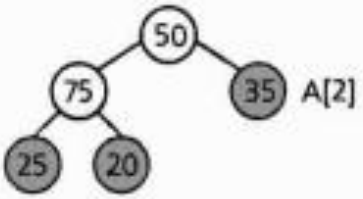
Исходная пирамида

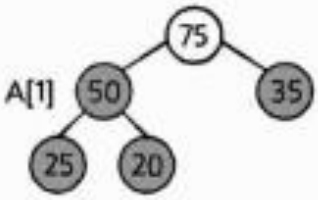
Удалить 20 и запомнить в A[4] Удалить 25 и запомнить в A[3]





Удалить 35 и запомнить в A[2] Удалить 50 и запомнить в A[1]





Поскольку единственный оставшийся элемент 75 является корнем, массив отсортирован: A = 75 50 35 25 20.

Ниже приводится реализация алгоритма пирамидальной сортировки. Функция HeapSort находится в файле heapsort.h.

```
Функция HeapSort
#include "heap.h"       // класс Heap

// отсортировать массив A по убыванию
template <class T>
void HeapSort (T A[], int n)
{
  // конструктор, преобразующий A в пирамиду
  Heap<T> H(A, n);
  T elt;

  // цикл заполнения элементов A[n-1] ... A[1]
  for (int i=n-1; i>=1; i--)
  {
    // исключить наименьший элемент из пирамиды и запомнить его в A[i]
    elt = H.Delete();
    A[i] = elt;
  }
}
```

Вычислительная эффективность пирамидальной сортировки. Массив, содержащий n элементов соответствует законченному бинарному дереву глубиной $k = \log_2 n$. Начальная фаза преобразования массива в пирамиду требует n/2 операций FilterDown. Каждой из них требуется не более k сравнений. На второй фазе сортировки операция FilterDown выполняется n-1 раз. В худшем случае она требует k сравнений. Объединив обе фазы, получим худший случай сложности пирамидальной сортировки:

$$k * \frac{n}{2} + k * (n - 1) = k * \left(\frac{3n}{2} - 1\right) = \log_2 n \times \left(\frac{3n}{2} - 1\right)$$

Таким образом, сложность алгоритма имеет порядок $O(n \log_2 n)$.

Пирамидальная сортировка не требует никакой дополнительной памяти, поскольку производится на месте. Турнирная сортировка является алгоритмом порядка $O(n \log_2 n)$, но требует создания последовательно представляемого мас-

сивом дерева из $2^{(k+1)}$ узлов, где k — наименьшее целое, при котором $n \leq 2^k$. Некоторые $O(n \log_2 n)$ сложные сортировки дают $O(n^2)$ в худшем случае. Примером может служить сортировка, рассмотренная в разделе 13.7. Пирамидальная сортировка всегда имеет сложность $O(n \log_2 n)$ независимо от исходного распределения данных.

Программа 13.2. Сравнение методов сортировки

Массив A, содержащий 2000 случайных целых чисел, сортируется с помощью функции HeapSort (пирамидальная сортировка). В целях сравнения массивы B и C заполняются теми же элементами и сортируются с помощью функций TournamentSort (турнирная сортировка) и ExchangeSort (обменая сортировка). Функция ExchangeSort находится в файле arrsort.h. Сортировки хронометрируются функцией TickCount, которая возвращает число 1/60 долей секунды, прошедших с момента старта системы. Сортировка обменом, имеющая сложность $O(n^2)$, позволит четко представить быстродействие турнирного и пирамидального методов, имеющих сложность $O(n \log_2 n)$. Функция PrintFirst_Last распечатывает первые и последние пять элементов массива. Код этой функции не включен в листинг программы. Его можно найти в программном приложении в файле *prg13_2.cpp*.

```
#include <iostream.h>

#include "random.h"
#include "arrsort.h"
#include "toursort.h"
#include "heapsort.h"
#include "ticks.h"

enum SortType {heap, tournament, exchange};
void TimeSort (int *A, int n, char *sortName, SortType sort)
{
  long tcount;
  // TickCount - системная функция.
  // возвращает число 1/60 долей
  // секунды с момента старта системы

  cout << "Испытывается " << sortName << ":" << endl;
  // засечь время. отсортировать массив A. подсчитать затраченное
  // время в 1/60 долях секунды
  tcount = TickCount();
  switch(sort)
  {
    case heap:        HeapSort(A,n);
                      break;
    case tournament:  TournamentSort(A, n);
                      break;
    case exchange:    ExchangeSort(A, n);
                      break;
  }
  tcount = TickCount() - tcount;
  // распечатать 5 первых и 5 последних элементов
  // отсортированного массива
  for (int i=0; i<5; i++)
    cout << A[i] << " ";
  cout << "...";
```

```
  for (i=n-5; i<n; i++)
    cout << A[i] << " ";
  cout << endl;

  cout << "Продолжительность " << tcount << "\n\n";
}
void main(void)
{
  // указатели массивов A, B и C
  int *A, *B, *C;
  RandomNumber rnd;

  // динамическое выделение памяти и загрузка массивов
  A = new int [2000];
  B = new int [2000];
  C = new int [2000];

  // загрузить в массивы одни и те же 2000 случайных чисел
  for (int i=0; i<2000; i++)
    A[i] = B[i] = C[i] = rnd.Random(10000);
    TimeSort(A, 2000, "пирамидальная сортировка ", heap);
    delete [] A;
    TimeSort(B, 2000, "турнирная сортировка ", heap);
    delete [] B;
    TimeSort(C, 2000, "сортировка обменом ", heap);
    delete [] C;
}
/*
<Прогон программы 13.2>
Испытывается пирамидальная сортировка :
9999 9996 9996 9995 9990 ... 11 10 9 6 3
Продолжительность 16

Испытывается турнирная сортировка :
3 6 9 10 11 ... 9990 9995 9996 9996 9999
Продолжительность 36

Испытывается сортировка обменом :
3 6 9 10 11 ... 9990 9995 9996 9996 9999
Продолжительность 818
*/
```

13.4. Очереди приоритетов

Очереди приоритетов рассматривались в гл. 5 и использовались в задаче моделирования событий. Клиенту был предоставлен доступ к оператору вставки и оператору удаления, который удалял из списка элемент с наивысшим приоритетом. В главе 5 для реализации списка, лежащего в основе объекта PQueue, использовался массив.

В этом разделе очередь приоритетов реализуется с помощью пирамиды. Поскольку мы используем минимальную пирамиду, предполагается, что элементы имеют возрастающие приоритеты. Операция удаления из пирамиды возвращает наименьший (с наивысшим приоритетом) элемент очереди приоритетов. Пирамидальная реализация обеспечивает высокую эффективность метода PQDelete, так как требует только $O(\log_2 n)$ сравнений. Это соизмеримо с $O(n)$ сравнениями в реализации с помощью массива.

Данный раздел завершается рассмотрением фильтра, преобразующего массив элементов в длинные последовательности[1]. Такой фильтр, используя очередь приоритетов, существенно повышает эффективность сортировки слиянием при упорядочении больших наборов данных файла. Эта тема обсуждается в гл. 14.

Спецификация класса PQueue (пирамидальная версия)

ОБЪЯВЛЕНИЕ

```
#include "heap.h"

template <class T>
class PQueue
{
  private:
    // пирамида, в которой хранится очередь
    Heap<T> *ptrHeap;

  public:
    // конструктор
    PQueue (int sz);

    // операции модификации очереди приоритетов
    void PQInsert(const T& item);
    T PQDelete(void);
    void ClearPQ(void);

    // методы опроса состояния очереди приоритетов
    int PQEmpty(void) const;
    int PQFull(void) const;
    int PQLength(void) const;
};
```

ОПИСАНИЕ

В конструктор передается параметр sz, который используется для динамического размещения структуры, адресуемой указателем ptrHeap. Методы реализуются простым вызовом соответствующего метода в классе Heap. Например, PQDelete использует метод исключения элемента из пирамиды.

```
// удалить первый элемент очереди посредством удаления корня
// соответствующей пирамиды. возвратить удаленное значение
template <class T>
T PQueue<T>::PQDelete(void)
{
  return ptrHeap->Delete();
}
```

Реализация PQeue находится в файле pqueue.h.

Приложение: длинные последовательности

Сортировка слиянием является основным алгоритмом упорядочения больших файлов. Его эффективность возрастает, если данные фильтруются, т.е. предварительно преобразуются в длинные последовательности. В гл. 12 мы уже видели один такой фильтр, который вводит сразу k элементов данных и сортирует их. В этом случае минимальная длина последовательностей равна

1 Другие названия: серии, отрезки, цепочки. — *Прим. пер.*

k. В данном приложении используется k-элементная очередь приоритетов и создаются последовательности, длины которых часто существенно превышают k. Алгоритм читает элементы из исходного списка A и пропускает их через фильтр очереди приоритетов. Элементы возвращаются в исходный список в форме длинных последовательностей.

Проиллюстрируем алгоритм на примере. Пусть массив A имеет 12 целых чисел, а приоритетная очередь PQ1 является фильтром с k=4 элементами. PQ1 хранит элементы, которые в конечном счете попадут в текущую последовательность. Вторая приоритетная очередь, PQ2, содержит элементы для следующей последовательности. Для сканирования массива используются два индекса. Переменная loadIndex указывает элемент, который вводится в данный момент. Переменная currIndex указывает последний элемент, покинувший очередь PQ1 и вернувшийся в исходный массив. В нашем примере массив A изначально разбит на шесть последовательностей, самая длинная из которых содержит три элемента:

```
A = [13] [6 61 96] [26] [1 72 91] [37] [25 97] [21]
```

После фильтрации получатся три последовательности, самая длинная из которых будет содержать семь элементов.

Вначале в PQ1 загружаются элементы A[0]...A[3]. Так как очередь приоритетов удаляет элементы в возрастающем порядке, у нас уже есть средство для сортировки, по крайней мере, четырех элементов последовательности. Но мы поступим даже лучше. Исключим из PQ1 первый элемент (с минимальным значением) и присвоим его A[currIndex] = A[0] = 6. Это число начинает первую последовательность, а в PQ1 остается незаполненное место. Поскольку первые четыре элемента массива A были скопированы в PQ1, продолжим с четвертого элемента (loadIndex = 4). На каждом шаге сравниваются A[currIndex] и A[loadIndex]. Если первый из них больше, он в конце концов попадет в текущую последовательность и поэтому запоминается в PQ1. В противном случае он попадет в следующую последовательность и поэтому запоминается в PQ2. Опишем это действие для каждого элемента в нашем примере. После обработки некоторого элемента мы используем следующий формат для перечисления загруженных элементов в A, элементов которые еще должны считываться, и содержимое обеих очередей приоритетов:

A: <элементы, загружаемые в последовательности> A[loadIndex]: <оставшиеся элементы>

PQ1: <содержимое текущей последовательности> PQ2: <содержимое следующей последовательности>

Выполнить по шагам: Элемент A[4]=26 > A[currIndex]=6. Запомнить 26 в PQ1; и удалить 13 из PQ1; поместить 13 в A[1].

A: 6 13 A[5]...A[11]: 1 72 91 37 25 97 21
PQ1: 61 96 26 PQ2: <пусто>

Элемент A[5]=1 < A[currIndex]=13. Поэтому 1 принадлежит следующей последовательности. Запомнить 1 в PQ2; исключить 26 из PQ1; поместить 26 в A[2].

A: 6 13 26 A[6]...A[11]: 72 91 37 25 97 21
PQ1: 61 96 PQ2: 1

Элемент A[6]=72 больше, чем элемент 26 в текущей последовательности. Запомнить A[6] в PQ1; исключить 61 из PQ1. Аналогично, следующий

элемент 91 попадает в PQ1 прежде, чем произойдет исключение элемента 72 и запись его в текущую последовательность по индексу currIndex=4.

A: 6 13 26 61 72 A[8]...A[11]: 37 25 97 21
PQ1: 91 96 PQ2: 1

Элементы A[8]=37 и A[9]=25 больше, чем 72 и попадут в следующую последовательность. Они запоминаются в PQ2. Одновременно из PQ1 исключаются два элемента и помещаются в массив, а очередь PQ1 остается пустой.

A: 6 13 26 61 72 91 96 A[10]...A[11]: 97 21
PQ1: <пусто> PQ2: 1 37 25

Мы сформировали текущую последовательность и можем начинать следующую. Скопируем PQ2 в PQ1 и исключим наименьший элемент из вновь заполненной очереди PQ1. В нашем примере удаляем 1 из PQ1 и начнем следующую последовательность.

A: 6 13 26 61 72 91 96 1 A[10]...A[11]: 97 21
PQ1: 25 37 PQ2: <пусто>

Элемент A[10]=97 > 1 и запоминается в PQ1. Затем минимальное значение 25 исключается из PQ1.

A: 6 13 26 61 72 91 96 1 25 A[11]: 21
PQ1: 37 97 PQ2: <пусто>

Элемент A[11]=21 < 25 и должен ждать следующую последовательность. Он запоминается в PQ2, а 37 исключается из PQ1.

A: 6 13 26 61 72 91 96 1 25 37 <весь список пройден>
PQ1: 97 PQ2: 21

Сканирование исходного списка завершено. Исключить все элементы из PQ1 и поместить их в текущую последовательность. Затем все элементы из PQ2 поместить в следующую последовательность.

Последовательность 1: 6 13 26 61 72 91 96

Последовательность 2: 1 25 37 97

Последовательность 3: 21

Алгоритм Runs. Алгоритм порождения длинных последовательностей реализуется с помощью класса LongRunFilter. Его закрытые данные-члены включают массив и две приоритетные очереди, содержащие текущую и следующую последовательности. Конструктор связывает объект данного класса с массивом и создает соответствующие очереди приоритетов. Алгоритм поддерживается закрытыми методами LoadPQ, который включает элементы массива в очередь PQ1, и CopyPQ, который копирует элементы из PQ2 в PQ1.

ОБЪЯВЛЕНИЕ

```
template <class T>
class LongRunFilter
{
  private:
    // указатели, определяющие ключевые параметры в фильтре
    // список A и две очереди приоритетов — PQ1 и PQ2    T *A;
    PQueue<T> *PQ1, *PQ2;
```

```
    int loadIndex;
    // размер массива и очередей приоритетов
    int arraySize;
    int filterSize;

    // копирование PQ2 в PQ1
    void CopyPQ (void);
    // загрузка массива A в очередь приоритетов PQ1
    void LoadPQ (void);

  public:
    // конструктор и деструктор
    LongRunFilter(T arr[], int n, int sz);
    ~LongRunFilter(void);

    // создание длинных последовательностей
    void LoadRuns(void);

    // оценка последовательностей
    void PrintRuns(void) const;
    int CountRuns(void) const;
};
```

ОПИСАНИЕ

Конструктор инициализирует данные-члены и загружает элементы из массива в PQ1, формируя таким образом элементы первой последовательности. Метод LoadRuns является главным алгоритмом, преобразующим элементы массива в длинные последовательности.

Методы PrintRuns и CountRuns служат для иллюстрации алгоритма. Они используются для сравнения последовательностей до и после вызова LoadRuns.

Полная реализация класса LongRunFilter находится в файле longrun.h.

```
// сканировать массив A и создать длинные последовательности,
// пропуская элементы через фильтр
template <class T>
void LongRunFilter<T>::LoadRuns(void)
{
  T   value;
  int currIndex; = 0;

  if (filterSize == 0)
    return;

  // начать с загрузки наименьшего элемента из PQ1 в A
  A[currIndex] = PQ1->PQDelete();

  // заполнить PQ1 элементами из A
  // теперь просмотреть элементы, оставшиеся в A
  while (loadIndex < arraySize)
  {
    // рассмотреть очередной элемент списка
    value = A[loadIndex++];
    // если элемент больше или равен A[currIndex],
    // он принадлежит текущей последовательности
    // и попадает в PQ1. в противном случае он копируется
    // в PQ2 и в конечном счете попадает в следующую последовательность
    if (A[currIndex] <= value)
      PQ1->PQInsert(value);
    else
      PQ1->PQInsert(value);
```

```
    // если PQ1 пуста, текущая последовательность сформирована.
    // скопировать PQ2 в PQ1 и начать следующую последовательность
    if (PQ1->PQEmpty())
      CopyPQ();

    // взять элемент из PQ1 и включить его в последовательность
    if (!PQ1->PQEmpty())
      A[++currIndex] = PQ1->PQDelete;
  }
  // удалить элементы из текущей последовательности,
  // а затем из следующей
  while (!PQ1->PQEmpty())
    A[++currIndex] = PQ1->PQDelete;
  while (!PQ2->PQEmpty())
    A[++currIndex] = PQ2->PQDelete;
}
```

Программа 13.3. Длинные последовательности

Эта программа иллюстрирует применение фильтра. В первом примере берется небольшой массив из 15 элементов и фильтруется с помощью 4-элементных очередей приоритетов. На выход выдается перечень последовательностей до и после вызова фильтра. В более практическом примере обрабатывается 10000-элементный массив, который фильтруется с помощью 5-, 50- и 500-элементных очередей приоритетов. В каждом случае распечатывается число итоговых последовательностей.

```
#include <iostream.h>

#include "random.h"
#include "longrun.h"

// копирование массива A в массив B
void CopyArray(int A[], int B[], int n)
{
  for (int i=0; i<n; i++)
    B[i] = A[i];
}

void main()
{
  // исходный 15-элементный массив для иллюстрации фильтра
  int demoArray[15];

  // большие 10000-элементные массивы для подсчета последовательностей
  int *A = new int[10000], *B = new int[10000];
  RandomNumber rnd;

  // создать 15 случайных чисел; сформировать фильтр
  for (i=0; i<15; i++)
    demoArray[i] = rnd.Random(100);
  LongRunFilter<int> F(demoArray, 15, 4);

  // распечатать список до и после создания длинных последовательностей
  // с помощью 4-элементного фильтра
  cout << "Исходные последовательности" << endl;
  F.PrintRuns();
  cout << endl;
  F.LoadRuns();
```

```
  cout << "Отфильтрованные последовательности" << endl;
  F.PrintRuns();
  cout << endl;

  // сформировать массив из 10000 случайных чисел
  for (i=0; i<10000; i++)
    A[i] = rnd.Random(25000);

  cout << "Последовательности, полученные с помощью 3-х фильтров"
       << endl;
  LongRunFilter<int> LR(A, 10000, 0);
  cout << "Число последовательностей в исходном массиве: "
       << LR.CountRuns() << endl;

  // тестирование 5- , 50- и 500-элементных фильтров
  for (i=0; i<3; i++)
  {
    CopyArray(A, B, 10000);
    LongRunFilter<int> LR(B, 10000, filterSize);

    // создать длинные последовательности
    LR.LoadRuns();
    cout << "  Число последовательностей после фильтра "
         << filterSize << " = " << LR.CountRuns() << endl;

    // 10-кратное увеличение размера фильтра
    filtersize *= 10;
  }
}
/*
<Прогон программы 13.3>

Исходные последовательности
36
22      79
26      84
44      88
44      66      81
19      86
40
2       47

Отфильтрованные последовательности 22  26  36  44  44  66  79  81  84  86  88
2   19  40  47

Последовательности, полученные с помощью 3-х фильтров
Количество последовательностей в исходном массиве: 5077
  Число последовательностей после фильтра 5 = 991
  Число последовательностей после фильтра 50 = 101
  Число последовательностей после фильтра 500 = 11

*/
```

Источники:

1. У. Топп, У. Форд – Структуры данных в c++
2. https://habr.com/ru/company/otus/blog/460087/