Вопрос 41.

Структуры данных и алгоритмы работы с ними. Массивы, списки, деревья, методы хэширования. Алгоритмы построения, модификации, балансировки деревьев (бинарное дерево, B-дерево, AVL-дерево, красно-черное дерево). Типовые алгоритмы сортировки и поиска. Сортировка пузырьком, сортировка вставками, сортировка выбором, быстрая сортировка, сортировка слиянием, внешняя сортировка.

Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс. Структура данных часто является реализацией какого-либо абстрактного типа данных.

Абстрактный тип данных — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций. Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор, независимых от конкретной реализации типа, функций для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

Наиболее популярная структура данных – **массив**, где обращение к элементу происходит через его номер. Слово «массив» употребляется в различных контекстах:

- как абстрактный тип данных, т. е. множество с операциями:
  - получить элемент с номером N,
  - записать элемент с номером N,
- как физическая структура, реализованная в виде непрерывной области памяти.

Положительные черты массива:

- доступ за константное время к любому элементу,
- память тратится только на данные.

Следующая структура данных, которую рассмотрим – это **список**.

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется линейным. Длина списка равна числу элементов, содержащихся в списке, список нулевой длины называется пустым списком. Линейные связные списки являются простейшими динамическими структурами данных.

На рис. 1 приведена структура односвязного списка. На нем поле DATA – информационное поле, данные, NEXT – указатель на следующий элемент списка. Каждый список должен иметь особый элемент, называемый указателем начала списка (BEGIN), который обычно по формату отличен от

остальных элементов. В поле указателя последнего элемента списка находится специальный признак NIL, свидетельствующий о конце списка.

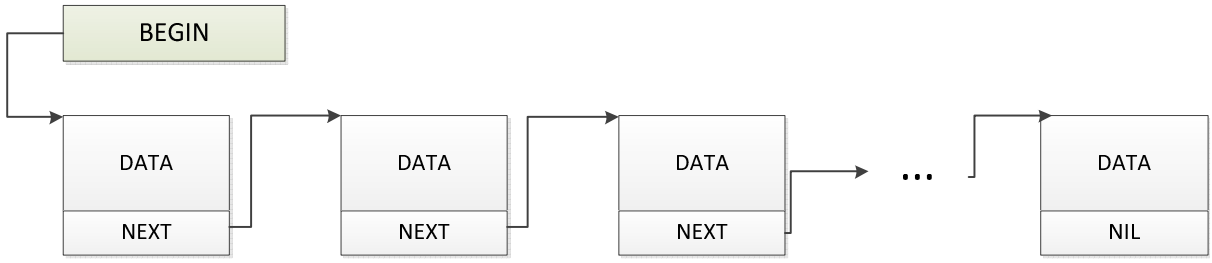


Рис. 1: Представление односвязного списка в памяти

Двусвязный список характеризуется наличием пары указателей в каждом элементе: на предыдущий элемент (PREV) и на следующий (NEXT) (см. рис.2).

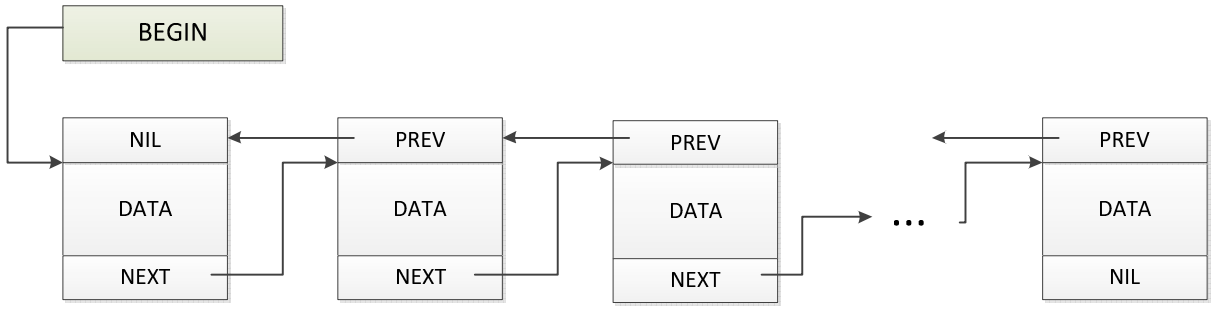


Рис. 2: Представление двусвязного списка в памяти

Положительные черты списков:

- добавление элементов в список за время $O(1)$;
- удаление элементов из списка $O(1)$.

Отрицательные черты:

- поиск только последовательный, выполняется путем полного перебора элементов списка и даже в отсортированном списке выполняется за $O(n)$;
- дополнительная память для хранения указателя на предыдущий (и следующий) элемент списка.

Приведенные структуры данных (массив и стек) не позволяют реализовать одновременно быстрые операции проверки принадлежности, добавления и удаления элемента. Если массив позволяет проверить принадлежность элемента массиву за время $O(\log_2 n)$ (в случае, если он отсортирован), то добавление и удаление элементов осуществляется за время пропорциональное числу элементов. Список же позволяет быстро добавлять и удалять элементы, но проверка принадлежности элемента списку (поиск) осуществляется за время пропорциональное числу элементов.

Существуют способы, позволяющие получить для всех трех упомянутых операций оценку $C \log_2 n$ – это хеширование и деревья.

Хеш-таблица

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существует два основных варианта хеш-таблиц: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив H, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками).

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение i = hash(key) играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией. Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

В некоторых специальных случаях удаётся избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую совершенную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий. Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются хеш-таблицами с прямой адресацией.

Двоичное дерево поиска

Двоичное дерево поиска (binary search tree, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

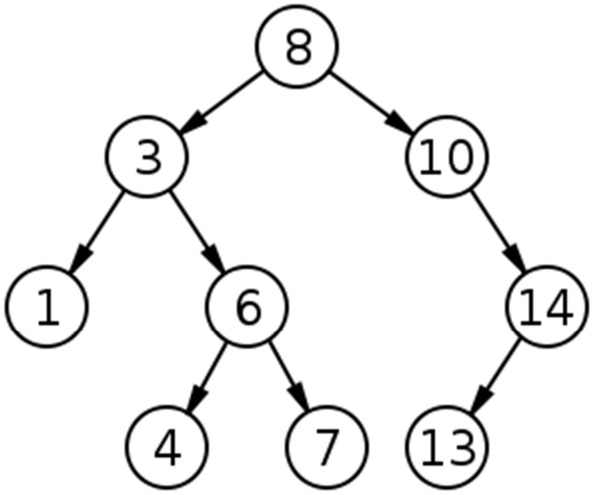
- оба поддерева — левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных больше, нежели значение ключа данных узла X.

Очевидно, данные в каждом узле должны обладать ключами на которых определена операция сравнения меньше.

Двоичное дерево поиска можно определить так:

- двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла – левый и правый сыновья соответственно (для оптимизации алгоритмов предполагают также определения в каждом узле ссылки на родительский элемент);
- данные (data) обладают ключом (key) на котором определена операция сравнения «меньше» (в конкретных реализациях это может быть пара (key, value), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё);

- для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], т. е. ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.



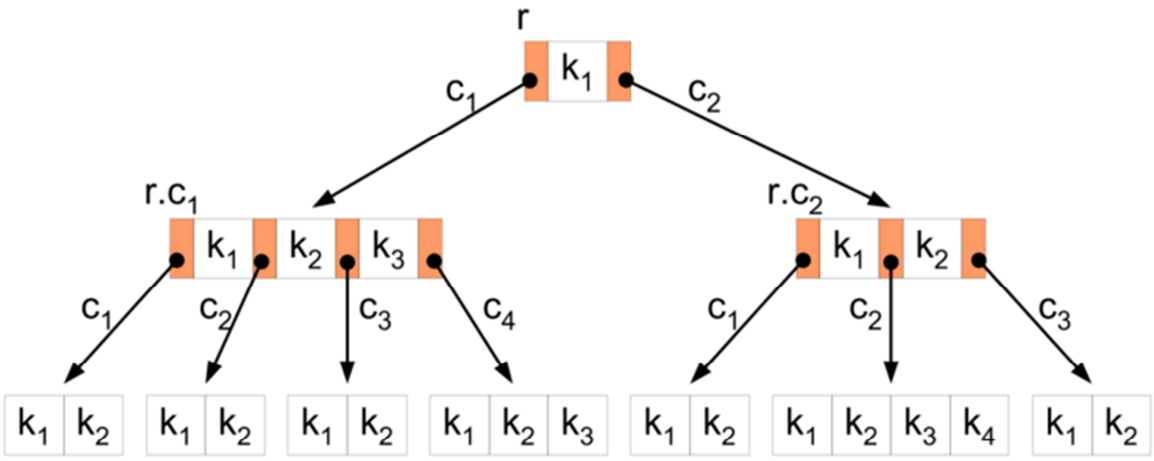
Балансировка дерева

Всегда желательно, чтобы все пути в дереве от корня до листьев имели примерно одинаковую длину, т.е. чтобы глубина и левого, и правого поддеревьев была примерно одинакова в любом узле. В противном случае теряется производительность. В вырожденном случае может оказаться, что все левое дерево пусто на каждом уровне, есть только правые деревья, и в таком случае дерево вырождается в список (идущий вправо). Поиск (а значит, и удаление и добавление) в таком дереве по скорости равен поиску в списке и намного медленнее поиска в сбалансированном дереве.

B-дерево

B-дерево — сбалансированное дерево поиска. Использование B-деревьев впервые было предложено Р. Бэйером (R. Bayer) и Е. МакКрейтом (E. McCreight) в 1970 году.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, то есть каждому узлу дерева соответствует блок внешней памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.



B-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Каждый узел содержит хотя бы один ключ. Ключи в каждом узле упорядочены. Корень содержит от 1 до $2t-1$ ключей. Любой другой узел содержит от $t-1$ до $2t-1$ ключей. Листья не являются исключением из этого правила. Здесь t – параметр дерева, не меньший 2.
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1,...,K_n$, содержит $n+1$ сыновей. При этом
    1. Первый сын и все его потомки содержат ключи из интервала $(-\infty, K_1)$.
    2. Для $2 \le i \le n$, i-й сын и все его потомки содержат ключи из интервала (K_{i-1}, K_i).
    3. $(n+1)$-й сын и все его потомки содержат ключи из интервала (K_n, ∞).
3. Глубина всех листьев одинакова.

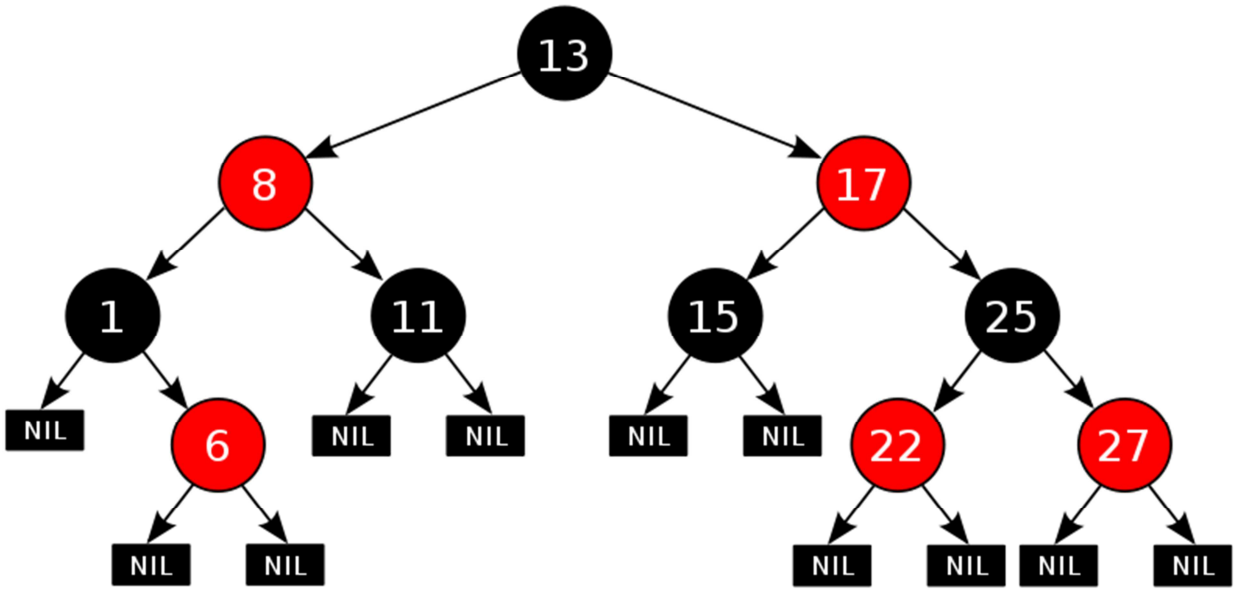
Свойство 2 можно сформулировать иначе: каждый узел B-дерева, кроме листьев, можно рассматривать как упорядоченный список, в котором чередуются ключи и указатели на сыновей.

Красно-чёрное дерево

Красно-чёрное дерево (Red-Black-Tree, RB-Tree) — это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счет введения дополнительного атрибута узла дерева — «цвет». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный». Красно-чёрное дерево обладает следующими свойствами:

- все листья черные;
- все потомки красных узлов черные (т. е. запрещена ситуация с двумя красными узлами подряд);
- на всех ветвях дерева, ведущих от его корня к листьям, число чёрных узлов одинаково. Это число называется чёрной высотой дерева.

При этом для удобства листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных.



AVL дерево

AVL-дерево (названные в честь их двух изобретателей Г.М. Адельсона-Вельского и Е.М. Ландиса) — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. В каждой вершине дополнительно хранится разность между высотами левого и правого поддеревьев, которая в сбалансированном дереве может принимать только три значения: -1, 0, 1. Строго говоря, AVL-деревья не являются сбалансированными в смысле приведенного выше определения. Требуется только, чтобы для любой вершины AVL-дерева разность высот ее левого и правого поддеревьев была по абсолютной величине не больше единицы. При этом длины путей от корня к внешним вершинам могут различаться больше, чем на единицу.

Поиск

Если массив является неупорядоченным, то единственный алгоритм поиска, применимый в этом случае, является *последовательный поиск*, при котором последовательно перебираются все элементы массива, и если на каком-то шаге цикла обнаруживается, что массив закончился или обнаруживается искомый элемент, то цикл заканчивается.

Классическим алгоритмом поиска в отсортированном массиве является *двоичный* (*бинарный*) *поиск* (также известен как метод деления пополам и дихотомия). Суть метода состоит в следующем. Необходимо запомнить левую (first) и правую (last) границы отрезка массива, где находится искомый элемент (первоначально это первый и последний элементы массива). Далее исследуется средний элемент отрезка (mid=(first+last)/2). Если искомое значение меньше среднего элемента, то искомый элемент находится в левой половине массива и изменяется правая граница (last=mid-1). Иначе, искомый элемент находится в правой половине массива, следовательно, изменяем левую границу (first=mid). На следующей итерации работаем с половиной массива. Таким образом, двоичный поиск требует $\log_2(N)$ сравнений (здесь N – количество элементов в массиве).

Пример кода на языке программирования Си для поиска элемента x в массиве A из n элементов, отсортированного в возрастающем порядке:

```
int first = 0; // Первый элемент в массиве
int last = n - 1; // Последний элемент в массиве
if ( x < A[first] || A[last] < x ) {
    // x лежит вне заданного массива
}else{
  while ( first < last ) {
    int mid = ( first + last ) / 2;
    if ( x < A[mid] ) {
      last = mid - 1;
    } else {
      first = mid;
    }
  }
  if ( A[last] == x ) {
    // Искомый элемент найден. last - искомый индекс
  } else {
    // Искомый элемент не найден.
  }
}
```

Если для элементов массива введено правило, по которому вычисляется расстояние между ними, т. е. введена операция вычитания, то можно реализовать *интерполяционный поиск*.

Чтобы понять принцип работы алгоритма интерполяционного поиска, представим себе задачу поиска слова в словаре (или записи в телефонной книге). Маловероятно, что сначала заглянем в середину словаря, затем отступим от начала на 1/4 или 3/4 и т. д, как в двоичном поиске. Если нужное слово начинается с буквы 'A', то, наверное, начнем поиск где-то в начале словаря. Если при этом заметим, что искомое слово должно находиться гораздо дальше открытой страницы, то пропустим порядочное их количество, прежде чем сделать новую попытку. Это в корне отличается от алгоритма двоичного поиска, не делающего разницы между «много больше» и «чуть больше». Если бинарный поиск учитывал только знак разности между искомым элементом и текущим значением, то интерполирующий поиск учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки left+(right-left)(K-Kleft)/(Kright-Kleft), здесь искомый элемент K лежит между Kleft и Kright.

В среднем, интерполирующий поиск производит $\log_2(\log_2(N))$ операций, где N есть число элементов, среди которых производится поиск. Число необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(N) операций.

Пример кода на языке программирования Си для поиска элемента X в массиве A из n элементов, отсортированного в возрастающем порядке:

```
    int index=-1;
    int left = 0; // Первый элемент в массиве
    int right = N - 1; // Последний элемент в массиве
    if ( toFind < A[left] || A[right] < toFind ) {
        // toFind  лежит вне заданного массива
    }else{
      while(left<=right) {
          index=left+(right-left)*(toFind-A[left])/(A[right]-A[left]);
          if(toFind<A[index]]
              right=index-1;
```

```
            else if(A[index]<toFind)
               left=index+1;
            else
             break;
      }

      if ( index == -1 ) {
            // Искомый элемент не найден
      } else {
            // Искомый элемент найден. index - искомый индекс
      }
    }
```

Типовые алгоритмы сортировки

Алгоритм сортировки — это алгоритм для упорядочения элементов в массиве (списке). Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти.

Сортировка пузырьком

Сортировка простыми обменами, сортировка пузырьком (англ. bubble sort) — простой алгоритм сортировки, эффективен лишь для небольших массивов. Сложность алгоритма: $O\left(n^2\right)$.

Алгоритм состоит в повторяющихся проходах по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При проходе алгоритма, элемент, стоящий не на своём месте, «всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма.

Пример кода на языке программирования Си для сортировки массива A из n элементов:

```
bool swapped=true;
for(int j=0; j<n-1 && swapped; ++j){
  swapped=false;
  for(int i=1; i<n-j; ++i){
    if (A[i]<A[i-1]){
       swap(A[i-1], A[i]);
       swapped=true;
    }
  }
}
```

Булева переменная `swapped` используется для того, чтобы определить, был ли произведён хотя бы один обмен на очередной итерации внешнего цикла. Алгоритм останавливается, когда таких обменов не было.

Сортировка выбором

Сортировка выбором (англ. selection sort) — алгоритм сортировки, относящийся к неустойчивым алгоритмам сортировки. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $O\left(n^2\right)$, предполагая, что сравнения делаются за постоянное время. При этом данный алгоритм гарантирует, что максимальное количество перестановок будет n, количество сравнений $\frac{n(n-1)}{2}$.

Алгоритм состоит в поиске минимального значения в текущем списке, обмене этого значения со значением на первой неотсортированной позиции и дальнейшем переходе к сортировке хвоста списка, исключив из рассмотрения уже отсортированные элементы.

Пример кода на языке программирования Си для сортировки массива A из n элементов:

```
for(int j=0; j<n-1; ++j){
  int i=j;
  for(i=j+1; i<n; ++i){
    if (A[i]<A[j]) i=j;
  }
  if(i!=j) swap(A[i],A[j]);
}
```

Сортировка вставками

Сортировка вставками (англ. insertion sort) — простой алгоритм сортировки, эффективный на наборах данных, которые уже частично отсортированы. Данный алгоритм является устойчивым алгоритмом сортировки (не меняет порядок элементов, которые уже отсортированы), может сортировать список по мере его получения. Сложность алгоритма: $O\left(n^2\right)$.

На каждом шаге алгоритма выбирается один из элементов входных данных и вставляется на нужную позицию в уже отсортированном списке, до тех пор пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по порядку их появления во входном массиве.

Пример кода на языке программирования Си для сортировки массива A из n элементов:

```
for(int j=1; j<n-1; ++j){
  for(int i=j; i>0 && A[i]<A[i-1]; --i){
     swap(A[i],A[i-1]);
}
```

Сократить количество операций присваивания (в среднем) можно путем запоминания выбранного элемента из входных данных в отдельной переменной (TMP). Пример кода:

```
for(int j=1; j<n-1; ++j){
  int i=j;
  TMP=A[j];
  while(i>0 && A[j]<A[i-1]){
     A[i]=A[i-1];
     --i;
  }
  if(i>0 && i!=j) A[i]=TMP;
}
```

На больших массивах более эффективным является использование бинарного поиска для определения местоположения очередного элемента при вставке его в отсортированную последовательность.

Быстрая сортировка

Быстрая сортировка (англ. quick sort) — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром в 1960 году. Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем $O(n \log_2 n)$ обменов при упорядочении n элементов), хотя и имеющий ряд недостатков.

Быстрая сортировка использует стратегию «разделяй и властвуй». Алгоритм состоит в следующем. Выбираем в массиве некоторый элемент, который будем называть опорным элементом. Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него. Обычный алгоритм операции:

- два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива соответственно;
- вычисляется индекс опорного элемента m;
- индекс l последовательно увеличивается до m до тех пор, пока l-й элемент не превысит опорный;
- индекс r последовательно уменьшается до m до тех пор, пока r-й элемент не окажется меньше либо равен опорному;
- если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент;
- если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно.

Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.

Пример кода на языке программирования C++:

```
template<class T>
void quickSort(T* A, long N) {
  long left = 0, right = N-1;
  T p;
  p = A[ (left+right)/2 ];  // центральный элемент

  // процедура разделения
  do {
    while ( A[left] < p ) ++left;
    while ( p < A[right] ) --right;

    if (left <= right) {
      swap(A[left],A[right]);
      ++left; --right;
    }
  } while ( left<=right );

  // рекурсивные вызовы, если есть, что сортировать
  if ( left > 0 ) quickSort(A, j);
  if ( right<N-1 ) quickSort(A+right, N-right-1);
}
```

Сортировка слиянием

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Для решения задачи сортировки эти три этапа выглядят так:

- сортируемый массив разбивается на две части примерно одинакового размера;
- каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
- два упорядоченных массива половинного размера соединяются в один.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

Нетривиальным этапом является соединение двух упорядоченных массивов в один. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем две стопки карт, лежащих рубашками вниз так, что в любой момент мы видим верхнюю карту в каждой из этих стопок. Пусть также, карты в каждой из этих стопок идут сверху вниз в неубывающем порядке. Как сделать из этих стопок одну? На каждом шаге мы берём меньшую из двух верхних карт и кладём её (рубашкой вверх) в результирующую стопку. Когда одна из оставшихся стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке.

Время работы алгоритма порядка $O(n \log_2 n)$ при отсутствии деградации на неудачных случаях, которая есть больное место быстрой сортировки (тоже алгоритм порядка $O(n \log_2 n)$, но только для лучшего случая). Для сортировки массива данных, расход памяти выше, чем для быстрой сортировки. Требуется дополнительно памяти столько же, сколько занимает исходный массив. Для сортировки списков указанным методом, дополнительная память не требуется.

Внешняя сортировка

Рассмотренные выше алгоритмы являются алгоритмами внутренней сортировки. Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат.

Внешняя сортировка оперирует с запоминающими устройствами большого объёма, но с доступом не произвольным, а последовательным, т. е. в данный момент мы «видим» только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

Идея большинства методов заключается в расчленении данных на ряд последовательностей помещающихся в оперативную память. Далее применяется один из методов внутренней сортировки, после чего последовательности сливаются. Чем больше объём оперативной памяти, тем длиннее будут последовательности и, следовательно, тем меньшим окажется их количество, что увеличит скорость сортировки. Если же объём оперативной памяти мал, то можно разделить исходные данные на несколько последовательностей, после чего непосредственно использовать процедуру слияния.

Литература

1. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. Структуры данных и алгоритмы. — М.: Вильямс, 2000. — С. 384.
2. Майкл Мейн, Уолтер Савитч. Структуры данных и другие объекты в C++. — 2-е изд. — М.: Вильямс, 2002. — С. 832.
3. Роберт Седжвик. Фундаментальные алгоритмы на C. Анализ / Структуры данных / Сортировка / Поиск. — СПб.: ДиаСофтЮП, 2003. — С. 672.