Лабораторная работа №7

Сортировка

Цель работы: изучить быстрые и медленные алгоритмы сортировки.

Теория

В мире компьютеров сортировка и поиск принадлежат к числу наиболее распространенных и хорошо изученных задач. Процедуры сортировки и поиска используются почти во всех программах управления базами данных, (а также в компиляторах, интерпретаторах и операционных системах. В настоящей главе представлены основные алгоритмы сортировки и поиска. Как вы сможете убедиться, они также иллюстрируют некоторые важные приемы программирования на языке С. Вообще говоря, поскольку целью сортировки, является облегчение и ускорение поиска данных, алгоритмы сортировки рассматриваются в первую очередь.

Сортировка

Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию. Сортировка является одной из наиболее приятных для умственного анализа категорией алгоритмов, поскольку процесс сортировки очень хорошо определен. Алгоритмы сортировки были подвергнуты обширному анализу, и способ их работы хорошо понятен. К сожалению, вследствие этой изученности сортировка часто воспринимается как нечто само собой разумеющееся. При необходимости отсортировать данные многие программисты просто вызывают стандартную функцию `qsort()`, входящую в стандартную библиотеку `<cstdlib>`. Однако различные подходы к сортировке обладают разными характеристиками. Несмотря на то, что некоторые способы сортировки могут быть в среднем лучше, чем другие, ни один алгоритм не является идеальным для всех случаев. Поэтому широкий набор алгоритмов сортировки – полезное добавление в инструментарий любого программиста.

Будет полезно кратко остановиться на том, почему вызов `qsort()` не является универсальным решением всех задач сортировки. Во-первых, функцию общего назначения вроде `qsort()` невозможно применить во всех ситуациях. Например, `qsort()` сортирует только массивы в памяти. Она не может сортировать данные, хранящиеся в связанных списках. Во-вторых, `qsort()` – параметризованная функция, благодаря чему она может обрабатывать широкий набор типов данных, но вместе с тем вследствие этого она работает медленнее, чем эквивалентная функция, рассчитанная на какой-то один тип данных. Наконец, как вы увидите, хотя алгоритм быстрой сортировки, примененный в функции `qsort()`, очень эффективен в общем случае, он может оказаться не самым лучшим алгоритмом в некоторых конкретных ситуациях.

Существует две общие категории алгоритмов сортировки: алгоритмы, сортирующие объекты с произвольным доступом (например, массивы или дисковые файлы произвольного доступа), и алгоритмы, сортирующие последовательные объекты (например, файлы на дисках и лентах или связанные списки). В данной лабораторной работе рассматриваются только алгоритмы первой категории, поскольку они наиболее полезны для среднестатистического программиста.

Чаще всего при сортировке данных лишь часть их используется в качестве ключа сортировки. Ключ – это часть информации, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. Для простоты в нижеследующих примерах будет производиться сортировка массивов символов, в которых ключ и данные совпадают. Далее вы увидите, как адаптировать эти методы для сортировки структур данных любого типа.

Классы алгоритмов сортировки

Существует три общих метода сортировки массивов:

- Обмен
- Выбор (выборка)
- Вставка

Чтобы понять, как работают эти методы, представьте себе колоду игральных карт. Чтобы отсортировать карты методом обмена, разложите их на столе лицом вверх и меняйте местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе выбора разложите карты на столе, выберите карту наименьшей значимости и положите ее в руку. Затем из оставшихся карт снова выберите карту наименьшей значимости и положите ее на ту, которая уже находится у вас в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом вставки, возьмите все карты в руку. Выкладывайте их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.

Медленные алгоритмы сортировки

Среди медленных алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

- алгоритм сортировки вставками;
- алгоритм сортировки пузырьком;
- минимаксные алгоритмы;
- алгоритм сортировки перечислениями.

Алгоритм сортировки вставками заключается в прохождении по всем элементам массива, начиная со второго, при котором каждый элемент массива вставляется на свое место. Для этого, начальный элемент каждой итерации цикла помещается во временную переменную и его значение сравнивается со всеми элементами, стоящими до этого элемента. Если их значение больше текущего, то они сдвигаются на один элемент вправо. После достижения элемента меньшего текущего, или начала массива, текущий элемент вставляется на новое место. Так продолжается до последнего элемента массива.

```
void sortInsert(TYPE array[], int num) {

	// цикл по всему массиву начиная со второго элемента
	for (int j = 1; j < num; j++) {

		// сохраняем индекс предыдущего элемента в массиве
		int i = j - 1;		// перед текущим элементом

		// сохраняем текущий элемент в локальной переменной
		TYPE value = array[j];

		// цикл по всем элементам от текущего до первого элемента,
		// меньшего его по значению, или до начала массива
		while ((value < array[i]) && (i >= 0)) {

			// сдвиг всех ранее идущих элементов на одну позицию "вправо"
			array[i + 1] = array[i];

			// движение в сторону начала массива
			i--;
		}

		// вставка текущего элемента на новую позицию в массиве
		array[i + 1] = value;
	}
}
```

ПРИМЕЧАНИЕ! Здесь и далее во всех функциях сортировки:

1. первый параметр `array` – массив, который необходимо упорядочить;
2. второй параметр `num` – количество элементов в массиве.

Алгоритм пузырьковой сортировки является наиболее распространенным среди медленных алгоритмов сортировки. Он заключается в сравнении двух элементов попарно и их перестановкой согласно направлению сортировки. Существует несколько реализаций.

```
void sortFullBubble(TYPE array[], int num) {

	// внешний и внутренний циклы от первого до последнего элемента в массиве
	for (int i = 0; i < num - 1; i++) {
		for (int j = 0; j < num - 1; j++) {

			// если предыдущий элемент "больше" последующего элемента,
			// то обмен значений элементов через дополнительную переменную
			if (array[j] > array[j + 1]) {
				TYPE tmp = array[j];
				array[j] = array[j + 1];
				array[j + 1] = tmp;
			}
		}
	}
}
```

Приведенный алгоритм сортировки полным пузырьком не учитывает «эффекта пузырька»: самый легкий (тяжелый) элемент за одну итерацию по массиву поднимается (опускается) до самого конца массива, поэтому его следует исключить из обработки на последующих итерациях по массиву.

```
void sortSimpleBubble(TYPE array[], int num) {

	// в переменной last на каждой итерации по массиву содержится
	// позиция последнего элемента, который необходимо обрабатывать
	int last = num;

	// пока есть элементы для обрботки
	while (last > 0) {

		// в переменной pos содержится индекс последнего элемента массива,
		// значение которого было обменено с предыдущим элементом
		int pos = 0;

		// цикл от первого элемента массива, до того элемента, который
		// исключён из обработки на предыдущей итерации цикла while
		for (int i = 0; i < last - 1; i++) {

			// если предыдущий элемент "больше" последующего элемента,
			// то обмен значений элементов через дополнительную переменную
			if (array[i] > array[i + 1]) {
				TYPE tmp = array[i];
				array[i] = array[i + 1];
				array[i + 1] = tmp;

				// запоминаем позицию последнего переставленного элемента
				pos = i + 1;
			}
		}

		// исключаем упорядоченные элементы из последующей обработки
		last = pos;
	}
}
```

Суть минимаксных алгоритмов сортировки заключается в поиске максимального и/или минимального элементов в массиве и обмен значений найденных элементов с последним и/или первым элементами в массиве. После каждой итерации границы массива сужаются.

```
void sortMaximum(TYPE array[], int num) {

	// цикл по количеству элементов в массиве
	while (num > 0) {

		// записываем в переменную maxpos позицию начального максимума
		// (первый элемент массива)
		int maxpos = 0;

		// цикл поиска максимума
		for (int i = 1; i < num; i++) {
			if (array[i] > array[maxpos])	// если новый максимум найден,
				maxpos = i;		// то запоминаем его позицию
		}

		// если найденный максимум находится не на последней позиции,
		// то обмен значений элементов через дополнительную переменную
		if (maxpos != num - 1) {
			TYPE tmp = array[maxpos];
			array[maxpos] = array[num - 1];
			array[num - 1] = tmp;
		}

		// исключение нейденного и переставленного
		// максимума из дальнейшей обработки
		num--;
	}
}
```

Далее приведен алгоритм, реализованный в виде функции, осуществляющий сортировку минимаксным способом: ищутся и максимальный и минимальный элементы и помещаются в конец и начало рассматриваемого массива соответственно.

```
void sortMinMax(TYPE array[], int num) {

	// в переменных lo и hi содержатся индексы первого
	// и последнего рассматриваемых элемнетов массива
	int lo = 0, hi = num - 1;

	// пока не сойдёмся на середине массива
	while (hi > lo) {

		// позиции максимума и минимума в текущей итерации
		int imax = lo, imin = lo;

		// цикл поиска максимума и минимума массива
		for (int i = lo; i <= hi; i++) {

			// если найден новый максимум,
			if (array[i] > array[imax])
				imax = i;	// то запоминаем его позицию

			// если найден новый минимум,
			if (array[i] < array[imin])
				imin = i;	// то запоминаем его позицию
		}

		// если позиция максмума и минимума совпадают,
		// то массив состоит из одинаковых элементов,
		if (imax == imin)
			return;		// поэтому выход из сортировки
```

```
            // если максимум расположен не в последнем рассматриваемом элементе
            // массива, то обмен значениями через дополнительную переменную
            if (imax != hi) {
                  TYPE tmp = array[imax];
                  array[imax] = array[hi];
                  array[hi] = tmp;
            }

            // если минимум расположен не в первом рассматриваемом элементе
            if (imin != lo) {

                  // если минимум находился на позиции последнего элемента,
                  // который был переставлен с максимальным элемнетом местами,
                  // то устанавливаем новую позицию максимума
                  if (imin == hi)
                        imin = imax;

                  // обмен значениями через дополнительную переменную
                  TYPE tmp = array[imin];
                  array[imin] = array[lo];
                  array[lo] = tmp;
            }

            // исключение найденных максимума и минимума из дальнейшей обработки
            lo++;
            hi--;
      }
}
```

Идея сортировки перечислением состоит в том, чтобы сравнить попарно все элементы и подсчитать, сколько из них меньше каждого отдельного элемента. Для подсчета используется вспомогательный массив, в котором, после завершения подсчета, находится окончательное положение элементов исходного массива в упорядоченной последовательности.

```
void sortEnumeration(TYPE array[], int num) {

      // создание целочисленного массива
      int *intarray = (int *)calloc(num, sizeof(int));

      // если массив не создан,
      if (!intarray)
            return;      // то завершение сортировки

      // создание дополнительного массива для сортировки
      TYPE *newarray = (TYPE *)calloc(num, sizeof(TYPE));

      // если массив не создан,
      if (!newarray) {
            free(intarray);     // то удаление целочисленного массива
            return;             // и завершение сортировки
      }

      // заполнение целочисленного массива нулями
      memset(intarray, 0, num * sizeof(int));

      // два вложенных цикла перебора всех комбинаций из
      // двух элементов исходного массива для их сравнения
      for (int i = 0; i < num - 1; i++) {
            for (int j = i + 1; j < num; j++) {

                  // если i-ый элемент "больше" j-го,
                  if (array[i] > array[j])
                        intarray[i]++;      // то инкрементируем i-ую ячейку
                  else intarray[j]++; // в противном случае j-ую ячейку
            }
      }
```

```
        // заполнение дополнительного массива элементами
        // из исходного массива в упорядоченном виде
        for (int i = 0; i < num; i++)
              newarray[intarray[i]] = array[i];

        // копирование дополнительного массива в исходный массив
        memcpy(array, newarray, num * sizeof(int));

        free(intarray);     // удаление целочисленного массива
        free(newarray);     // удаление дополнительного массива
}
```

Быстрые алгоритмы сортировки

Среди быстрых алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

- алгоритм Шелла;
- алгоритм Хоара;
- алгоритм Флойда.

Идея алгоритма Шелла состоит в том, что в исходном наборе элементов (массиве) сначала упорядочиваются элементы, расположенные на расстоянии друг от друга, т.е. итерации по массиву осуществляются через несколько элементов. Затем шаг уменьшается и производится сортировка с новым размером шага. Так продолжается до тех пор, пока шаг не станет равным единице. Таким образом, алгоритм Шелла представляет собой обычный медленный алгоритм сортировки, в котором ускорение процесса осуществляется за счет того, что на первых этапах (с большим шагом) обрабатывается меньше элементов, и они упорядочиваются быстрее, перемещаясь по массиву с большими шагами. На практике скорость данного алгоритма редко достигает указанной ранее зависимости $(n \times \log(2 \cdot n))$, поэтому данный алгоритм относят к классу быстрых алгоритмов скорее условно, чем в действительности. Сама сортировка в алгоритме Шелла может осуществляться любым из рассмотренных ранее медленных алгоритмов.

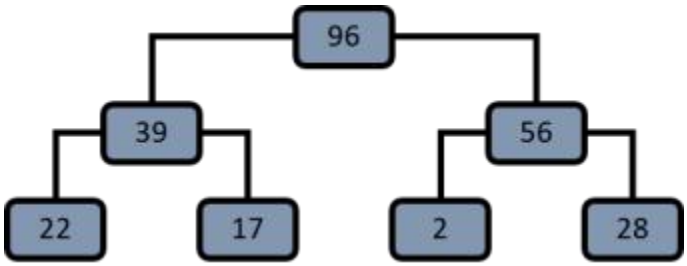
```
void ShellSort(TYPE array[], int num) {

        // массив, содержащий размеры шагов обработки массива
        // данных (должен представлять собой убывающую
        // последовательность с последним элементом равным 1)
        int steps[] = { 9, 5, 3, 1 };

        // цикл сортировки Шелла (по всему массиву шагов)
        for (int n = 0; n < 4; n++) {
              int first = 0;      // установка первой позиции в массиве

              // пока не дойдём до конца массива
              while (first < num) {

                    // устанавливаем в переменную pos текущую позицию,
                    // в переменную min - позицию текущего минимума
                    int pos = first, min = first;

                    // цикл минимального элемента от позиции first до конца массива
                    while (pos < num) {

                          // если найден новый минимум,
                          if (array[pos] < array[min])
                                min = pos;   // то запоминаем его позицию

                          // переходим к следующему элементу
                          // (с заданным на текущей итерации шагом)
                          pos += steps[n];
                    }
```

```
                        // если позиция найденного минимума не совпадает с первым,
                        // то обмен значений элементов через дополнительную переменную
                        if (min > first) {
                                TYPE tmp = array[min];
                                array[min] = array[first];
                                array[first] = tmp;
                        }

                        // переход к новой итерации по массиву
                        first += steps[n];
                }
        }
}
```

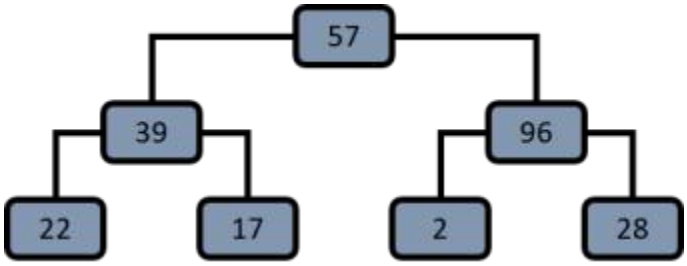
Алгоритм быстрой сортировки Хоара заключается в следующем: берется массив, выбирается один элемент, а все остальные элементы делятся на два подмножества - элементы, большие выбранного или равные ему, и элементы, меньшие выбранного. Затем процедура применяется рекурсивно к каждому из двух подмножеств. Если в подмножестве меньше двух элементов, сортировка ему не требуется, и рекурсия на этом прекращается.

```
void HoareSort(TYPE array[], int num) {

        // вызов сортировки для всего массива
        quickSort(array, 0, num - 1);
}

void quickSort(TYPE *array, int iLo, int iHi) {

        // заносим границы подмножества в лакальные переменные
        int Lo = iLo, Hi = iHi;

        // выбираем один элемент, с которым будем сравнивать все остальные
        // элементы модмножества массива (центральный элемент)
        TYPE mid = array[(Lo + Hi) / 2];

        do {

                // поиск элемента "больше" выбранного элемента слева от него
                while (array[Lo] < mid) Lo++;

                // поиск элемента "меньше" выбранного элемента справа от него
                while (array[Hi] > mid) Hi--;

                // если элементы найдены,
                if (Lo <= Hi) {

                        // то обмен значений через дополнительную переменную
                        TYPE tmp = array[Lo];
                        array[Lo] = array[Hi];
                        array[Hi] = tmp;

                        // модификация индексов для продолжения поиска
                        Lo++;
                        Hi--;
                }

        // выход из цикла, когда всё подможество будет рассмотрено
        } while (Lo <= Hi);

        // рекурсивный вызов для левой части подмножества
        if (Hi > iLo) quickSort(array, iLo, Hi);

        // рекурсивный вызов для правой части подмножества
        if (Lo < iHi) quickSort(array, Lo, iHi);
}
```

Алгоритм Флойда является самым оптимальным из алгоритмов сортировки. В нём активно используется упорядоченное двоичное дерево:

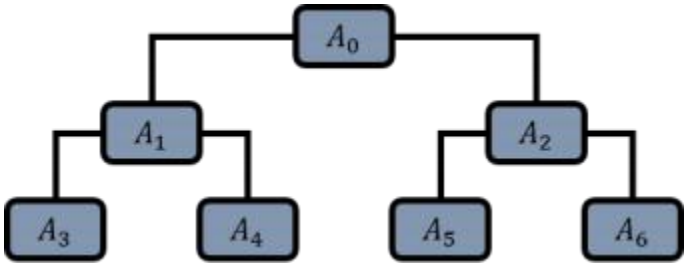


Значение в каждой из его вершин не меньше, чем значения в его дочерних вершинах. Двоичное дерево называется частично упорядоченным, если свойство упорядоченности выполняется для каждой из его вершин, однако для корня это свойство нарушается. Пример частично упорядоченного дерева:

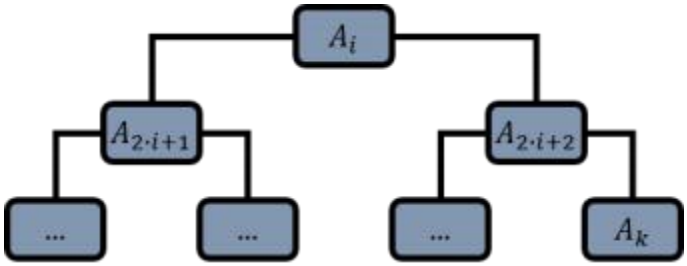


Структура дерева позволяет сохранить состояние процесса сортировки последовательности на каждом его шаге, с целью использования этого состояния в дальнейших расчетах и уменьшении числа операций сравнения при поиске наибольшего (наименьшего) из оставшихся элементов.

В алгоритме сортировки Флойда исходная последовательность данных представляется в виде дерева на смежной памяти. В таком дереве ребра присутствуют неявно и вычисляются с помощью арифметических операций над индексами элементов массива. Корень дерева `a[0]`, за каждой вершиной `a[k]` следуют вершины `a[2*k+1]` и `a[2*k+2]`. На рисунке ниже приведен пример нумерации вершин двоичного дерева на смежной памяти.

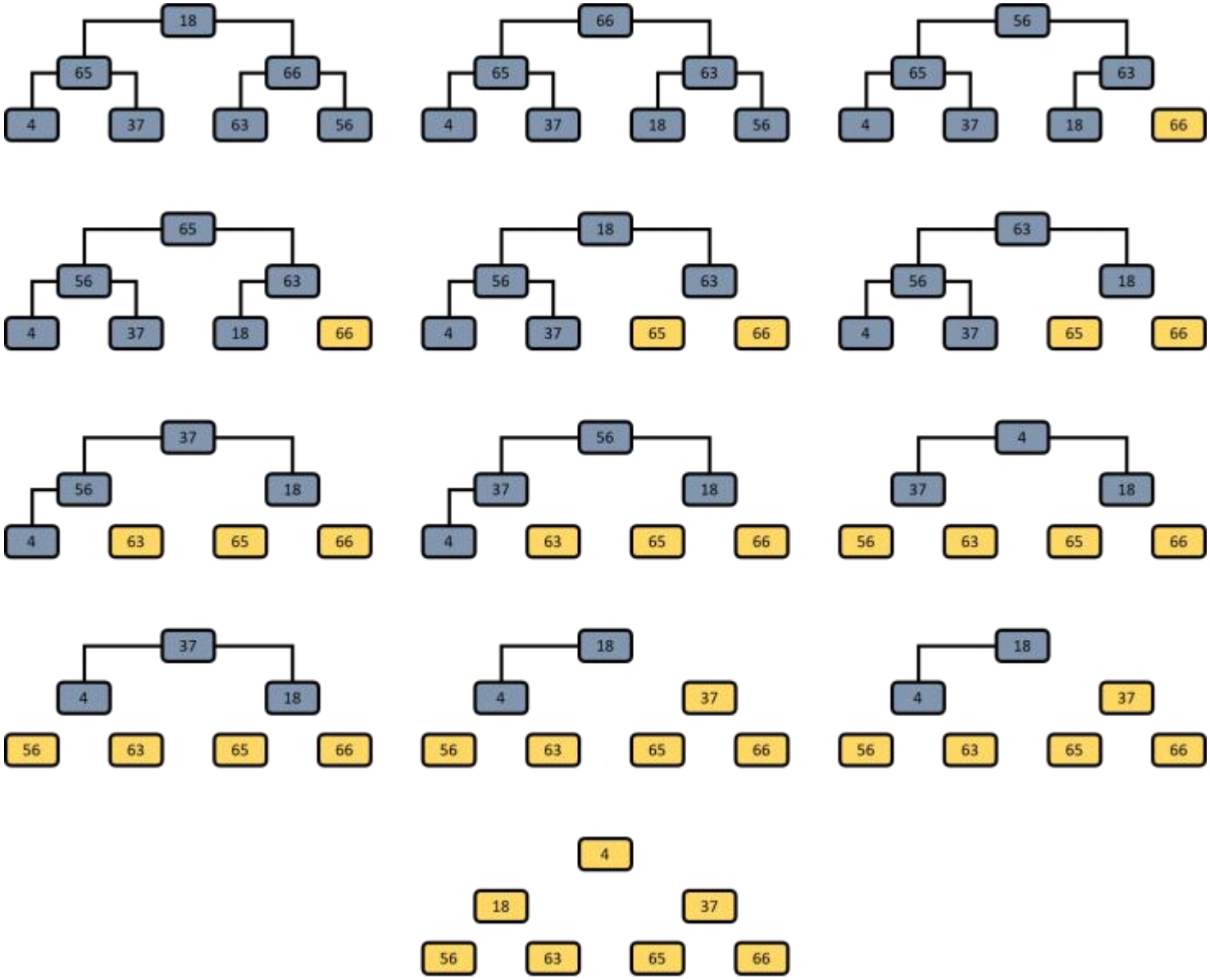


Основу алгоритма составляет функция `surface (a[i..k])` всплытия Флойда, которая за $Q(n) \sim (\log(2 \cdot n))$ сравнений преобразует почти упорядоченное поддерево в упорядоченное. Поддерево представляется на одномерно массиве `a[i..k]`, где `a[i]` – корень дерева, `a[k]` – максимальный элемент массива, который еще может принадлежать поддереву:



Функция заключается в том, что значение корня (здесь может нарушаться условие упорядоченности) всплывает по направлению к листьям (последний уровень вершин в дереве) до тех пор, пока дерево не преобразуется в упорядоченное. Функция `surface` всплытия Флойда позволяет в почти упорядоченном дереве найти наибольший (наименьший) элемент за число сравнений $Q(\log(2 \cdot n))$, преобразуя дерево к упорядоченному виду. В результате найденный элемент будет располагаться в вершине дерева. Для сортировки множества элементов, из них сначала организуется почти упорядоченное двоичное дерево при помощи повторного применения функции `surface` сначала к самым мелким его поддеревьям от листьев и затем ко все более крупным. Листья тривиально упорядочены, поэтому можно начинать с минимальных поддеревьев, содержащих несколько вершин и укрупнять их, каждый раз полностью, применяя алгоритм всплытия до тех пор, пока не будет достигнут корень дерева. Заметим, что каждое из поддеревьев, к которому применяется алгоритм всплытия, удовлетворяет условию почти упорядоченности, поскольку упорядочивание происходит от листьев к корню.

После того, как дерево упорядочено, наибольший (наименьший) элемент оказывается в его корне. Найденный элемент меняют местами с самым последним листом в дереве (последний элемент рассматриваемого массива), дерево уменьшается на одну вершину и все готово для определения нового наибольшего (наименьшего) элемента множества. На рисунке ниже показана полная последовательность перестановок и всплытий, которые происходят после формирования из исходного множества почти упорядоченного дерева и вплоть до того, как в этом дереве останется всего одна вершина, а исходное множество окажется отсортированным.



Листинг функции сортировки Флойда:

```
void FloidSort(TYPE array[], int num) {

	// цикл для построения почти упорядоченных деревьев
	for (int i = num / 2; i > 0; i--)
		surface(array, i - 1, num);

	// цикл сортировки
	for (int k = num - 1; k > 0; k--) {

		// построение упорядоченного дерева
		surface(array, 0, k);

		// обмен значениями первого элемента в массиве (корня дерева)
		// и текущего последнего элемента в массиве
		TYPE tmp = array[k];
		array[k] = array[0];
		array[0] = tmp;
	}
}

void surface(TYPE array[], int i, int k) {

	// сохраняем корень в локальной переменной
	TYPE copy = array[i];

	// вычисляем индекс элемента дерева "слева" от корня
	int m = 2 * i + 1, j = 0;

	while (m < k) {
		// если это последний элемент в дереве,
		if (m == (k - 1))
			j = m;		// то запоминаем его позицию
		// если элемент слева "больше" элемента справа,
		else if (array[m] > array[m + 1])
			j = m;		// то запоминаем позицию элемента справа
		// иначе
		else
			j = m + 1;	// запоминаем позицию элемента справа

		// если найденный элемент больше расположенного в корне дерева,
		if (array[j] > copy) {
			array[i] = array[j];  // то записываем его в корень
			i = j;                 // корень устанавливаем на его позицию
			m = 2 * i + 1;         // вычисляем индекс следующего элемента
		}
		// иначе выходим из цикла
		else break;
	}

	// итоговой вершине дерева присваиваем
	// сохранённое значение исходной вершины
	array[i] = copy;
}
```

При выборе того или иного алгоритма сортировки следует учитывать то, что быстрые алгоритмы выгодны для сортировки достаточно больших массивов (размер которых исчисляется сотнями и даже тысячами элементов). Также необходимо обращать внимание на то, что сортируемые данные могут иметь большой объем. Поэтому алгоритмы сортировки следует реализовывать так, чтобы минимизировать число перестановок элементов в процессе сортировки.

Практика

Реализуйте сортировки по двум полям динамической структуры курсовой работы (Лабораторные работы 5-6), алгоритмами согласно варианту. При реализации алгоритмов сортировок НЕ преобразовывать динамические структуры в массив.

Вариант	**Алгоритмы**
1	1. Вставками 2. Шелла
2	1. Вставками 2. Пузырьком
3	1. Пузырьком 2. Шелла
4	1. Вставками 2. Флойда
5	1. Пузырьком 2. Хоара
6	1. Вставками 2. Хоара
7	1. Шелла 2. Флойда
8	1. Пузырьком 2. Флойда
9	1. Флойда 2. Хоара

Дополнение

Для проверки корректности работы описанных выше функций необходимо в файл исходного кода добавить следующее содержимое:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

typedef int TYPE;

void sortInsert(TYPE array[], int num);
void sortFullBubble(TYPE array[], int num);
void sortSimpleBubble(TYPE array[], int num);
void sortMaximum(TYPE array[], int num);
void sortMinMax(TYPE array[], int num);
void sortEnumeration(TYPE array[], int num);

void ShellSort(TYPE array[], int num);

void HoareSort(TYPE array[], int num);
void quickSort(TYPE *array, int iLo, int iHi);

void FloidSort(TYPE array[], int num);
void surface(TYPE array[], int i, int k);

void printArray(char *str, TYPE array[], int num);
void randomInt(TYPE array[], int num);

int main(void) {

	int n = 10;

	TYPE *arr = new TYPE[n];

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortInsert(arr, n);
	printArray((char *) "SortInsert", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortFullBubble(arr, n);
	printArray((char *) "SortFullBubble", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortSimpleBubble(arr, n);
	printArray((char *) "SortSimpleBubble", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortMaximum(arr, n);
	printArray((char *) "SortMaximum", arr, n);
	cout << endl;
```

```
	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortMinMax(arr, n);
	printArray((char *) "SortMinMax", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	sortEnumeration(arr, n);
	printArray((char *) "SortEnumeration", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	ShellSort(arr, n);
	printArray((char *) "ShellSort", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	HoareSort(arr, n);
	printArray((char *) "HoareSort", arr, n);
	cout << endl;

	randomInt(arr, n);
	printArray((char *) "Array", arr, n);
	FloidSort(arr, n);
	printArray((char *) "FloidSort", arr, n);
	cout << endl;


	for (int i = 0; i < col; i++)
		cout << "arr[" << ind[i] << "] = " << arr[ind[i]] << endl;

	return 0;
}

void printArray(char *str, TYPE array[], int num) {

cout << str << ": ";

	for (int i = 0; i < num; i++)
		cout << array[i] << " ";

	cout << endl;
}
void randomInt(TYPE array[], int num) {

	srand((unsigned int) time(NULL));

	for (int i = 0; i < num; i++)
		array[i] = rand() % 50;
}
```

Результат работы программы:

```
Array: 5 33 45 8 42 30 40 20 12 49
SortInsert: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
SortFullBubble: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
SortSimpleBubble: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
SortMaximum: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
SortMinMax: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
SortEnumeration: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
ShellSort: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
HoareSort: 5 8 12 20 30 33 40 42 45 49

Array: 5 33 45 8 42 30 40 20 12 49
FloidSort: 5 8 12 20 30 33 40 42 45 49

Enter value to search (linear): 42
arr[4] = 42

Enter value to search (linear modified): 8
arr[3] = 8

Enter value to search (multi linear after sort): 42
arr[7] = 42
```