# Лабораторная работа №5

Выполнил: Лосев Данил ЕПИ-4-23

**Код**

|  |
| --- |
| #include <chrono>  #include <conio.h>  #include <cstddef>  #include <cstdio>  #include <cstdlib>  #include <ctime>  #include <iomanip>  #include <iostream>  #include <synchapi.h>  #include <windows.h>  int PRINT\_MAX\_SIZE = 10000;  int TEST\_STEP\_SIZE = 50;  struct Node  {  int value;  Node \*next;  };  Node \*ListSort(Node \*(\*sort)(Node \*), Node \*fHead, long long &fTime);  void FillListBestCase(Node \*&fHead, int fSize);  void FillListMiddleCase(Node \*&fHead, int fSize);  void FillListBadCase(Node \*&fHead, int fSize);  Node \*BubbleSortList(Node \*fHead);  Node \*SelectionSortList(Node \*fHead);  Node \*InsertionSortList(Node \*fHead);  Node \*DeleteList(Node \*&fHead);  void PrintList(Node \*fHead);  int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime);  void FillArrayBestCase(int \*&fArray, int fSize);  void FillArrayMiddleCase(int \*&fArray, int fSize);  void FillArrayBadCase(int \*&fArray, int fSize);  int \*BubbleSortArray(int \*&fArray, int fSize);  int \*SelectionSortArray(int \*&fArray, int fSize);  int \*InsertionSortArray(int \*&fArray, int fSize);  void PrintArray(int \*fArray, int fSize);  void listAutoTest(int fMaxSize);  void arrayAutoTest(int fMaxSize);  long long autoSortTestForList(Node \*(\*sort)(Node \*), void (\*fill)(Node \*&, int), int fSize);  long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize);  int main()  {  int programm = 1;  int choose = 1;  int size = 1;  long long sortTime;  while (programm != 0)  {  std::system("cls");  std::cout << "Enter the size of list / array: ";  std::cin >> size;  Sleep(500);  std::system("cls");  Node \*head;  int \*array = new int[size]{};  std::system("cls");  std::cout << "Enter the type of filling list / array\n";  std::cout << " 1) Best case\n";  std::cout << " 2) Middle case\n";  std::cout << " 3) Bad case\n";  std::cout << "\n> ";  std::cin >> choose;  switch (choose)  {  case 1:  FillListBestCase(head, size);  FillArrayBestCase(array, size);  break;  case 2:  FillListMiddleCase(head, size);  FillArrayMiddleCase(array, size);  break;  case 3:  FillListBadCase(head, size);  FillArrayBadCase(array, size);  break;  }  if (size <= PRINT\_MAX\_SIZE)  {  std::cout << "\nList: ";  PrintList(head);  std::cout << "Array: ";  PrintArray(array, size);  }  getch();  std::system("cls");  std::cout << "Enter the sort for list:\n";  std::cout << " 1) Bubble Sort\n";  std::cout << " 2) Selection Sort\n";  std::cout << " 3) Insertion Sort\n> ";  std::cin >> choose;  switch (choose)  {  case 1:  head = ListSort(BubbleSortList, head, sortTime);  break;  case 2:  head = ListSort(SelectionSortList, head, sortTime);  break;  case 3:  head = ListSort(InsertionSortList, head, sortTime);  break;  }  if (size <= PRINT\_MAX\_SIZE)  {  std::cout << "Sorted list: ";  PrintList(head);  }  std::cout << "Sort time: " << sortTime << std::endl;  getch();  std::system("cls");  std::cout << "Enter the sort for array:\n";  std::cout << " 1) Bubble Sort\n";  std::cout << " 2) Selection Sort\n";  std::cout << " 3) Insertion Sort\n> ";  std::cin >> choose;  switch (choose)  {  case 1:  array = ArraySort(BubbleSortArray, array, size, sortTime);  break;  case 2:  array = ArraySort(SelectionSortArray, array, size, sortTime);  break;  case 3:  array = ArraySort(InsertionSortArray, array, size, sortTime);  break;  }  if (size <= PRINT\_MAX\_SIZE)  {  std::cout << "Sorted array: ";  PrintArray(array, size);  }  std::cout << "Sort time: " << sortTime << std::endl;  getch();  std::system("cls");  delete[] array;  array = NULL;  head = DeleteList(head);  head = NULL;  int choose1 = 0;  std::cout << "Do you want to do auto test? (1=yes 0=no) : ";  std::cin >> choose1;  if (choose1 == 1)  {  std::system("cls");  int sizeOfTest;  std::cout << "Enter the size of test: ";  std::cin >> sizeOfTest;  listAutoTest(sizeOfTest);  std::cout << std::endl;  arrayAutoTest(sizeOfTest);  getch();  }  std::system("cls");  std::cout << "Go to start? (1=yes 0=no)\n> ";  std::cin >> programm;  }  return 0;  }  Node \*ListSort(Node \*(\*sort)(Node \*), Node \*fHead, long long &fTime)  {  auto start = std::chrono::high\_resolution\_clock::now();  Node \*Temp = sort(fHead);  auto end = std::chrono::high\_resolution\_clock::now();  fTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  return Temp;  }  void FillListBestCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = i + 1;  (\*curNode).next = NULL;  }  }  void FillListMiddleCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = (rand() % fSize) + 1;  (\*curNode).next = NULL;  }  }  void FillListBadCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = fSize - i;  (\*curNode).next = NULL;  }  }  Node \*BubbleSortList(Node \*fHead)  {  Node \*curNode;  Node \*lastSort = NULL;  int temp;  while (lastSort != (\*fHead).next)  {  curNode = (\*fHead).next;  while ((\*curNode).next != lastSort)  {  if ((\*curNode).value > (\*(\*curNode).next).value)  {  temp = (\*curNode).value;  (\*curNode).value = (\*(\*curNode).next).value;  (\*(\*curNode).next).value = temp;  }  curNode = (\*curNode).next;  }  lastSort = curNode;  }  return fHead;  }  Node \*SelectionSortList(Node \*fHead)  {  Node \*curNode = (\*fHead).next;  Node \*searchNode, \*minNode;  int temp;  while (curNode != NULL)  {  minNode = curNode;  searchNode = (\*curNode).next;  while (searchNode != NULL)  {  if ((\*searchNode).value < (\*minNode).value)  {  minNode = searchNode;  }  searchNode = (\*searchNode).next;  }  if (minNode != curNode)  {  temp = (\*minNode).value;  (\*minNode).value = (\*curNode).value;  (\*curNode).value = temp;  }  curNode = (\*curNode).next;  }  return fHead;  }  Node \*InsertionSortList(Node \*fHead)  {  Node \*curNode, \*searchNode;  Node \*sortedList = (\*fHead).next;  (\*fHead).next = (\*(\*fHead).next).next;  (\*sortedList).next = NULL;  while (fHead != NULL)  {  curNode = fHead;  fHead = (\*fHead).next;  if ((\*curNode).value < (\*sortedList).value)  {  (\*curNode).next = sortedList;  sortedList = curNode;  }  else  {  searchNode = sortedList;  while ((\*searchNode).next != NULL)  {  if ((\*(\*searchNode).next).value > (\*curNode).value)  {  break;  }  searchNode = (\*searchNode).next;  }  (\*curNode).next = (\*searchNode).next;  (\*searchNode).next = curNode;  }  }  return sortedList;  }  Node \*DeleteList(Node \*&fHead)  {  Node \*curNode = (\*fHead).next;  while ((\*curNode).next != NULL)  {  Node \*temp = curNode;  curNode = (\*curNode).next;  delete temp;  }  delete fHead;  delete curNode;  curNode = fHead = NULL;  return fHead;  }  void PrintList(Node \*fHead)  {  if (fHead == 0)  {  return;  }  Node \*curNode = fHead;  while ((\*curNode).next != NULL)  {  curNode = (\*curNode).next;  std::cout << (\*curNode).value << ' ';  }  std::cout << std::endl;  }  int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime)  {  auto start = std::chrono::high\_resolution\_clock::now();  int \*Temp = sort(fArray, fSize);  auto end = std::chrono::high\_resolution\_clock::now();  fTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  return Temp;  }  void FillArrayBestCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = i + 1;  }  }  void FillArrayMiddleCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = (rand() % fSize) + 1;  }  }  void FillArrayBadCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = fSize - i;  }  }  int \*BubbleSortArray(int \*&fArray, int fSize)  {  int temp;  for (int i = 0; i < fSize - 1; i++)  {  for (int j = 0; j < fSize - i - 1; j++)  {  if (fArray[j] > fArray[j + 1])  {  temp = fArray[j];  fArray[j] = fArray[j + 1];  fArray[j + 1] = temp;  }  }  }  return fArray;  }  int \*SelectionSortArray(int \*&fArray, int fSize)  {  int minIndex, temp;  for (int i = 0; i < fSize; i++)  {  minIndex = i;  for (int j = i + 1; j < fSize; j++)  {  if (fArray[j] < fArray[minIndex])  {  minIndex = j;  }  }  temp = fArray[i];  fArray[i] = fArray[minIndex];  fArray[minIndex] = temp;  }  return fArray;  }  int \*InsertionSortArray(int \*&fArray, int fSize)  {  for (int i = 1; i < fSize; i++)  {  int curInt = fArray[i];  int j;  for (j = i - 1; j >= 0 && fArray[j] > curInt; j--)  {  fArray[j + 1] = fArray[j];  }  fArray[j + 1] = curInt;  }  return fArray;  }  void PrintArray(int \*fArray, int fSize)  {  if (fArray == 0)  {  return;  }  for (int i = 0; i < fSize; i++)  {  std::cout << fArray[i] << ' ';  }  std::cout << std::endl;  }  void listAutoTest(int fMaxSize)  {  std::cout << "\n List Time Auto Test\n";  const int width = 12;  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  std::cout << std::setw(width / 2) << ' ' << " | " << std::setw(width) << ' ' << " " << std::setw(width)  << "Best case" << " " << std::setw(width) << ' ' << " | " << std::setw(width) << ' ' << " "  << std::setw(width) << "Average case" << " " << std::setw(width) << ' ' << " | " << std::setw(width)  << ' ' << " " << std::setw(width) << "Worst case" << " " << std::setw(width) << ' ' << " | "  << std::endl;  // Sort type headers  std::cout << std::setw(width / 2) << "Count" << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width)  << "Selection" << " | " << std::setw(width) << "Insertion" << " | " << std::setw(width) << "Bubble"  << " | " << std::setw(width) << "Selection" << " | " << std::setw(width) << "Insertion"  << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width) << "Selection" << " | "  << std::setw(width) << "Insertion" << " | " << std::endl;  // Отделитель  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE)  {  int size = i;  std::cout << std::setw(width / 2) << i << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListBadCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListBadCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListBadCase, size) << " | " << std::endl;  }  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  }  void arrayAutoTest(int fMaxSize)  {  std::cout << "\n Array Time Auto Test\n";  const int width = 12;  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  std::cout << std::setw(width / 2) << ' ' << " | " << std::setw(width) << ' ' << " " << std::setw(width)  << "Best case" << " " << std::setw(width) << ' ' << " | " << std::setw(width) << ' ' << " "  << std::setw(width) << "Average case" << " " << std::setw(width) << ' ' << " | " << std::setw(width)  << ' ' << " " << std::setw(width) << "Worst case" << " " << std::setw(width) << ' ' << " | "  << std::endl;  // Sort type headers  std::cout << std::setw(width / 2) << "Count" << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width)  << "Selection" << " | " << std::setw(width) << "Insertion" << " | " << std::setw(width) << "Bubble"  << " | " << std::setw(width) << "Selection" << " | " << std::setw(width) << "Insertion"  << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width) << "Selection" << " | "  << std::setw(width) << "Insertion" << " | " << std::endl;  // Отделитель  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE)  {  int size = i;  std::cout << std::setw(width / 2) << i << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayBadCase, size) << " | " << std::endl;  }  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  }  long long autoSortTestForList(Node \*(\*sort)(Node \*), void (\*fill)(Node \*&, int), int fSize)  {  Node \*testHead = NULL;  fill(testHead, fSize);  long long sortTime = 0;  testHead = ListSort(sort, testHead, sortTime);  DeleteList(testHead);  return sortTime;  }  long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize)  {  int \*testArray = new int[fSize]{};  fill(testArray, fSize);  long long sortTime = 0;  testArray = ArraySort(sort, testArray, fSize, sortTime);  delete[] testArray;  testArray = NULL;  return sortTime;  } |

**Блок схемы**

|  |  |
| --- | --- |
| Node \*ListSort(Node \*(\*sort)(Node \*), Node \*fHead, long long &fTime)  {  auto start = std::chrono::high\_resolution\_clock::now();  Node \*Temp = sort(fHead);  auto end = std::chrono::high\_resolution\_clock::now();  fTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  return Temp;  } |  |
| void FillListBestCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = i + 1;  (\*curNode).next = NULL;  }  } |  |
| void FillListMiddleCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = (rand() % fSize) + 1;  (\*curNode).next = NULL;  }  } |  |
| void FillListBadCase(Node \*&fHead, int fSize)  {  fHead = new (Node);  Node \*curNode = fHead;  (\*curNode).next = NULL;  (\*curNode).value = 0;  for (int i = 0; i < fSize; i++)  {  (\*curNode).next = new (Node);  curNode = (\*curNode).next;  (\*curNode).value = fSize - i;  (\*curNode).next = NULL;  }  } |  |
| Node \*BubbleSortList(Node \*fHead)  {  Node \*curNode;  Node \*lastSort = NULL;  int temp;  while (lastSort != (\*fHead).next)  {  curNode = (\*fHead).next;  while ((\*curNode).next != lastSort)  {  if ((\*curNode).value > (\*(\*curNode).next).value)  {  temp = (\*curNode).value;  (\*curNode).value = (\*(\*curNode).next).value;  (\*(\*curNode).next).value = temp;  }  curNode = (\*curNode).next;  }  lastSort = curNode;  }  return fHead;  } |  |
| Node \*SelectionSortList(Node \*fHead)  {  Node \*curNode = (\*fHead).next;  Node \*searchNode, \*minNode;  int temp;  while (curNode != NULL)  {  minNode = curNode;  searchNode = (\*curNode).next;  while (searchNode != NULL)  {  if ((\*searchNode).value < (\*minNode).value)  {  minNode = searchNode;  }  searchNode = (\*searchNode).next;  }  if (minNode != curNode)  {  temp = (\*minNode).value;  (\*minNode).value = (\*curNode).value;  (\*curNode).value = temp;  }  curNode = (\*curNode).next;  }  return fHead;  } |  |
| Node \*InsertionSortList(Node \*fHead)  {  Node \*curNode, \*searchNode;  Node \*sortedList = (\*fHead).next;  (\*fHead).next = (\*(\*fHead).next).next;  (\*sortedList).next = NULL;  while (fHead != NULL)  {  curNode = fHead;  fHead = (\*fHead).next;  if ((\*curNode).value < (\*sortedList).value)  {  (\*curNode).next = sortedList;  sortedList = curNode;  }  else  {  searchNode = sortedList;  while ((\*searchNode).next != NULL)  {  if ((\*(\*searchNode).next).value > (\*curNode).value)  {  break;  }  searchNode = (\*searchNode).next;  }  (\*curNode).next = (\*searchNode).next;  (\*searchNode).next = curNode;  }  }  return sortedList;  } |  |
| Node \*DeleteList(Node \*&fHead)  {  Node \*curNode = (\*fHead).next;  while ((\*curNode).next != NULL)  {  Node \*temp = curNode;  curNode = (\*curNode).next;  delete temp;  }  delete fHead;  delete curNode;  curNode = fHead = NULL;  return fHead;  } |  |
| void PrintList(Node \*fHead)  {  if (fHead == 0)  {  return;  }  Node \*curNode = fHead;  while ((\*curNode).next != NULL)  {  curNode = (\*curNode).next;  std::cout << (\*curNode).value << ' ';  }  std::cout << std::endl;  } |  |
| int \*ArraySort(int \*(\*sort)(int \*&, int), int \*&fArray, int fSize, long long &fTime)  {  auto start = std::chrono::high\_resolution\_clock::now();  int \*Temp = sort(fArray, fSize);  auto end = std::chrono::high\_resolution\_clock::now();  fTime = std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  return Temp;  } |  |
| void FillArrayBestCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = i + 1;  }  } |  |
| void FillArrayMiddleCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = (rand() % fSize) + 1;  }  } |  |
| void FillArrayBadCase(int \*&fArray, int fSize)  {  for (int i = 0; i < fSize; i++)  {  fArray[i] = fSize - i;  }  } |  |
| int \*BubbleSortArray(int \*&fArray, int fSize)  {  int temp;  for (int i = 0; i < fSize - 1; i++)  {  for (int j = 0; j < fSize - i - 1; j++)  {  if (fArray[j] > fArray[j + 1])  {  temp = fArray[j];  fArray[j] = fArray[j + 1];  fArray[j + 1] = temp;  }  }  }  return fArray;  } |  |
| int \*SelectionSortArray(int \*&fArray, int fSize)  {  int minIndex, temp;  for (int i = 0; i < fSize; i++)  {  minIndex = i;  for (int j = i + 1; j < fSize; j++)  {  if (fArray[j] < fArray[minIndex])  {  minIndex = j;  }  }  temp = fArray[i];  fArray[i] = fArray[minIndex];  fArray[minIndex] = temp;  }  return fArray;  } |  |
| int \*InsertionSortArray(int \*&fArray, int fSize)  {  for (int i = 1; i < fSize; i++)  {  int curInt = fArray[i];  int j;  for (j = i - 1; j >= 0 && fArray[j] > curInt; j--)  {  fArray[j + 1] = fArray[j];  }  fArray[j + 1] = curInt;  }  return fArray;  } |  |
| void PrintArray(int \*fArray, int fSize)  {  if (fArray == 0)  {  return;  }  for (int i = 0; i < fSize; i++)  {  std::cout << fArray[i] << ' ';  }  std::cout << std::endl;  } |  |
| void listAutoTest(int fMaxSize)  {  std::cout << "\n List Time Auto Test\n";  const int width = 12;  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  std::cout << std::setw(width / 2) << ' ' << " | " << std::setw(width) << ' ' << " " << std::setw(width)  << "Best case" << " " << std::setw(width) << ' ' << " | " << std::setw(width) << ' ' << " "  << std::setw(width) << "Average case" << " " << std::setw(width) << ' ' << " | " << std::setw(width)  << ' ' << " " << std::setw(width) << "Worst case" << " " << std::setw(width) << ' ' << " | "  << std::endl;    std::cout << std::setw(width / 2) << "Count" << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width)  << "Selection" << " | " << std::setw(width) << "Insertion" << " | " << std::setw(width) << "Bubble"  << " | " << std::setw(width) << "Selection" << " | " << std::setw(width) << "Insertion"  << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width) << "Selection" << " | "  << std::setw(width) << "Insertion" << " | " << std::endl;    std::cout << std::string(width \* 11 + 11, '-') << std::endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE)  {  int size = i;  std::cout << std::setw(width / 2) << i << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListBestCase, size) << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForList(BubbleSortList, FillListBadCase, size) << " | " << std::setw(width)  << autoSortTestForList(SelectionSortList, FillListBadCase, size) << " | " << std::setw(width)  << autoSortTestForList(InsertionSortList, FillListBadCase, size) << " | " << std::endl;  }  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  } |  |
| void arrayAutoTest(int fMaxSize)  {  std::cout << "\n Array Time Auto Test\n";  const int width = 12;  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  std::cout << std::setw(width / 2) << ' ' << " | " << std::setw(width) << ' ' << " " << std::setw(width)  << "Best case" << " " << std::setw(width) << ' ' << " | " << std::setw(width) << ' ' << " "  << std::setw(width) << "Average case" << " " << std::setw(width) << ' ' << " | " << std::setw(width)  << ' ' << " " << std::setw(width) << "Worst case" << " " << std::setw(width) << ' ' << " | "  << std::endl;  std::cout << std::setw(width / 2) << "Count" << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width)  << "Selection" << " | " << std::setw(width) << "Insertion" << " | " << std::setw(width) << "Bubble"  << " | " << std::setw(width) << "Selection" << " | " << std::setw(width) << "Insertion"  << " | " << std::setw(width) << "Bubble" << " | " << std::setw(width) << "Selection" << " | "  << std::setw(width) << "Insertion" << " | " << std::endl;  std::cout << std::string(width \* 11 + 11, '-') << std::endl;  for (int i = 100; i <= fMaxSize; i += TEST\_STEP\_SIZE) {  int size = i;  std::cout << std::setw(width / 2) << i << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayBestCase, size) << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayMiddleCase, size) << " | " << std::setw(width)  << autoSortTestForArray(BubbleSortArray, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(SelectionSortArray, FillArrayBadCase, size) << " | " << std::setw(width)  << autoSortTestForArray(InsertionSortArray, FillArrayBadCase, size) << " | " << std::endl; }  std::cout << std::string(width \* 11 + 11, '-') << std::endl; } |  |
| long long autoSortTestForList(Node \*(\*sort)(Node \*), void (\*fill)(Node \*&, int), int fSize)  {  Node \*testHead = NULL;  fill(testHead, fSize);  long long sortTime = 0;  testHead = ListSort(sort, testHead, sortTime);  DeleteList(testHead);  return sortTime;  } |  |
| long long autoSortTestForArray(int \*(\*sort)(int \*&, int), void (\*fill)(int \*&, int), int fSize)  {  int \*testArray = new int[fSize]{};  fill(testArray, fSize);  long long sortTime = 0;  testArray = ArraySort(sort, testArray, fSize, sortTime);  delete[] testArray;  testArray = NULL;  return sortTime;  } |  |

**Графики и таблицы**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Список | | | | | | | | | |
| кол-во | Лучший случай | | | Средний случай | | | Худший случай | | |
| пузырьковая | выбором | вставкой | пузырьковая | выбором | вставкой | пузырьковая | выбором | вставкой |
| 100 | 17 | 17 | 14 | 31 | 20 | 8 | 23 | 19 | 0 |
| 200 | 76 | 67 | 55 | 134 | 75 | 28 | 91 | 71 | 1 |
| 300 | 150 | 148 | 123 | 281 | 163 | 101 | 203 | 156 | 1 |
| 400 | 265 | 335 | 218 | 597 | 277 | 117 | 360 | 278 | 2 |
| 500 | 410 | 407 | 337 | 793 | 430 | 172 | 563 | 427 | 2 |
| 600 | 592 | 582 | 593 | 1033 | 642 | 246 | 808 | 614 | 2 |
| 700 | 844 | 793 | 682 | 1598 | 834 | 347 | 1081 | 820 | 3 |
| 800 | 1050 | 1246 | 1091 | 1515 | 1074 | 436 | 1409 | 1094 | 4 |
| 900 | 1332 | 1309 | 1319 | 2089 | 1654 | 690 | 1853 | 1415 | 5 |
| 1000 | 1701 | 1669 | 1408 | 2317 | 2030 | 975 | 2263 | 1728 | 5 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Массив | | | | | | | | | |
| кол-во | Лучший случай | | | Средний случай | | | Худший случай | | |
| пузырьковая | выбором | вставкой | пузырьковая | выбором | вставкой | пузырьковая | выбором | вставкой |
| 100 | 13 | 14 | 0 | 41 | 23 | 13 | 44 | 22 | 22 |
| 200 | 71 | 72 | 1 | 104 | 170 | 29 | 104 | 52 | 55 |
| 300 | 130 | 115 | 1 | 292 | 128 | 66 | 233 | 114 | 122 |
| 400 | 220 | 203 | 1 | 516 | 221 | 112 | 413 | 214 | 244 |
| 500 | 347 | 316 | 2 | 598 | 339 | 178 | 648 | 311 | 337 |
| 600 | 504 | 453 | 2 | 980 | 486 | 265 | 955 | 618 | 921 |
| 700 | 759 | 615 | 3 | 1218 | 651 | 360 | 1314 | 604 | 690 |
| 800 | 907 | 803 | 3 | 1546 | 842 | 467 | 2191 | 808 | 904 |
| 900 | 1142 | 1014 | 3 | 1972 | 1062 | 646 | 2716 | 994 | 1383 |
| 1000 | 1873 | 1250 | 4 | 2437 | 1330 | 743 | 2764 | 1226 | 1413 |

|  |  |  |
| --- | --- | --- |
| пузырьковая сортировка | | |
| колличество | время | |
| список | массив |
| 100 | 23 | 26 |
| 150 | 51 | 61 |
| 200 | 91 | 107 |
| 250 | 142 | 162 |
| 300 | 204 | 233 |
| 350 | 368 | 317 |
| 400 | 399 | 413 |
| 450 | 469 | 574 |
| 500 | 574 | 644 |
| 550 | 884 | 928 |
| 600 | 948 | 1021 |
| 650 | 1010 | 1267 |
| 700 | 1092 | 1259 |
| 750 | 1238 | 1471 |
| 800 | 1456 | 1729 |
| 850 | 1588 | 1872 |
| 900 | 1794 | 2099 |
| 950 | 1995 | 2226 |
| 1000 | 2221 | 2356 |

**Ответы на вопросы**

1. **Назовите основные алгоритмы простых сортировок?**

Основные алгоритмы простых сортировок:

* Сортировка выбором (Selection Sort)
* Сортировка вставками (Insertion Sort)
* Сортировка обменом (Пузырьковая сортировка) (Bubble Sort)

1. **Опишите алгоритмы их работы.**

 **Сортировка выбором**:

* Алгоритм проходит по массиву, выбирает минимальный элемент и меняет его с первым элементом.
* Затем повторяет процесс для следующей позиции, исключая отсортированные элементы.

 **Сортировка вставками**:

* Начинает с первого элемента (предполагая его отсортированным) и добавляет каждый последующий элемент в подходящее место внутри отсортированной части массива.

 **Пузырьковая сортировка**:

* Проходит по массиву, «всплывая» максимальный элемент в конец массива путем попарного обмена соседних элементов, пока весь массив не окажется отсортированным.

1. **Преимущества и недостатки**

|  |  |  |
| --- | --- | --- |
| **Алгоритм** | **Преимущества** | **Недостатки** |
| **Сортировка выбором** | Простая реализация | Неэффективна на больших массивах, постоянная временная сложность O(n2) независимо от упорядоченности |
| **Сортировка вставками** | Эффективна на малых и частично отсортированных массивах | Неэффективна на больших массивах |
| **Пузырьковая сортировка** | Легко реализовать, подходит для обучения | Медленная на больших массивах; часто работает медленнее других простых сортировок |

1. **Сравните скорости работы этих алгоритмов в зависимости от количества элементов в массиве.**

 Все три алгоритма имеют временную сложность O(n2) что означает, что их производительность резко снижается при увеличении размера массива.

 Однако **сортировка вставками** может быть быстрее в случаях с небольшим числом элементов или частично отсортированными данными.

1. **Сравните скорости работы этих алгоритмов в массиве и в односвязном списке в зависимости от количества элементов.**

Сортировка вставками более эффективна для односвязных списков, поскольку можно добавлять элементы на нужные позиции без затрат на обмен соседних элементов.

Для сортировки выбором и пузырьковой сортировки работа с односвязными списками усложняется из-за необходимости многократного обращения к элементам, что снижает эффективность.

1. **Сравните скорости работы этих алгоритмов в зависимости от первоначальной упорядоченности элементов в этих массивах?**

 **Сортировка вставками**: работает быстрее при частично отсортированных массивах, приближаясь к линейной временной сложности O(n).

 **Сортировка выбором** и **пузырьковая сортировка** не адаптируются к упорядоченности данных и сохраняют O(n2) сложность независимо от исходной упорядоченности.

1. **Что такое временная сложность алгоритмов для математиков и для программистов? От чего она зависит?**
2. **Для математиков**:
   * Временная сложность — это функция, показывающая, как растет количество элементарных операций алгоритма при увеличении объема входных данных.
   * Временная сложность записывается с использованием **Big-O нотации** (например, O(n)O(n)O(n), O(n2)O(n^2)O(n2), O(log⁡n)O(\log n)O(logn)), которая описывает асимптотическое поведение алгоритма.
   * Анализ временной сложности помогает строить точные математические модели, прогнозирующие поведение алгоритма на больших наборах данных.
3. **Для программистов**:
   * Временная сложность — это практический способ оценки производительности алгоритма, указывающий, как алгоритм будет себя вести при разных объемах данных.
   * Она помогает выбирать наиболее эффективные алгоритмы для задач с большими или критически важными данными и учитывает зависимости от размера данных, что важно для оптимизации кода.
   * Программисты часто оценивают, сколько операций потребуется алгоритму для завершения и как это скажется на времени выполнения в реальных условиях.

**От чего зависит временная сложность?**

Временная сложность алгоритма зависит от следующих факторов:

* **Структуры алгоритма**: Алгоритмы с различными подходами (поиск, сортировка, рекурсия) имеют разные временные сложности.
* **Размер входных данных**: С увеличением количества элементов растет количество операций, необходимых для выполнения алгоритма.
* **Тип структуры данных**: Работа с массивами, списками, деревьями и графами требует различных операций и, соответственно, имеет разную сложность.
* **Упорядоченность входных данных**: Многие алгоритмы, особенно сортировки, выполняются быстрее на частично упорядоченных или отсортированных данных, что уменьшает количество необходимых операций.
* **Дополнительные особенности алгоритма**: Некоторые алгоритмы имеют лучшие показатели в среднем, но худшие в худшем случае, и наоборот; это тоже отражается на сложности.

1. **Как используется память при работе простых сортировок?**

Простые сортировки, такие как **сортировка вставками**, **выбором** и **пузырьковая сортировка**, используют O(1) дополнительной памяти, так как выполняются на месте, требуя лишь несколько переменных для обмена и отслеживания индексов, что делает их экономичными с точки зрения памяти.