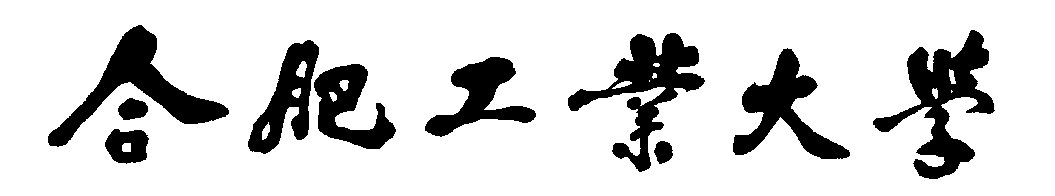
****

计算机与信息学院

数据结构实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 物联网一班 |
| 学生姓名及学号 | 敬成超 2023212388 |
| 课程教学班号 |  |
| 任 课 教 师 | 胡学钢 |
| 实验指导教师 |  |
| 实验地点 | C栋304 |
| 2023 ~2024 学年第二学期 | |

实验序号及名称：实验 七 **查找与排序的综合应用实验**

实验时间∶ 2024年 6 月15 日

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  **（1）掌握不同存储方式的查找与排序算法原理；**  **（2）掌握查找与排序算法在大数据上的应用；**  **（3）掌握算法的时间和空间性能的分析方法。** |
| 二、实验任务∶ |
| 三、实验准备方案，包括以下内容：  （硬件类实验：实验原理、实验线路、设计方案等）  （软件类实验：所采用的核心方法、框架或流程图及程序清单）  **排序类设计**  **核心方法**   1. **基顺序表的排序算法**：选择1~2种排序算法，如冒泡排序、选择排序等。 2. **堆排序算法**：基于二叉堆实现堆排序。 3. **链式基数排序算法**：实现基于链表的基数排序。 4. **性能测试**：使用随机数据对排序算法进行性能测试和时间消耗对比。   **框架**   * **Sort类**：包含排序算法的实现。 * **基顺序表排序方法**：实现1~2种基于数组的排序算法。 * **堆排序方法**：实现基于二叉堆的堆排序。 * **链式基数排序方法**：实现基于链表的基数排序。 * **性能测试方法**：生成随机数据，测试排序算法，并记录时间。   **流程图** |

|  |
| --- |
| **查找类设计**  **核心方法**   1. **顺序表查找算法**：实现1~2种基于数组的查找算法，如线性查找、二分查找等。 2. **树表查找算法**：实现1~2种基于树的查找算法，如二叉搜索树查找。 3. **散列表查找算法**：实现2种基于散列表的查找算法，考虑不同的哈希函数和冲突解决策略。 4. **性能测试**：使用随机数据对查找算法进行性能测试和时间消耗对比。   **框架**   * **Search类**：包含查找算法的实现。 * **顺序表查找方法**：实现基于数组的查找算法。 * **树表查找方法**：实现基于树的查找算法。 * **散列表查找方法**：实现基于散列表的查找算法。 * **性能测试方法**：生成随机数据，测试查找算法，并记录时间。   **流程图** |

|  |
| --- |
| 实验内容 |
| 一、实验用仪器、设备：  电脑 |
| 二、实验内容与步骤（过程及数据记录）  **实验任务一：查找类设计**   1. **设计查找类**：   创建一个查找类，包含数据存储和查找算法的实现。   1. **顺序表查找算法**：   实现至少1种顺序表查找算法，如线性查找。   1. **树表查找算法**：   实现至少1种树表查找算法，如二叉搜索树查找。   1. **散列表查找算法**：   实现至少2种基于散列表的查找算法，如开放寻址法和链地址法。   1. **数据生成与验证**：   使用随机函数生成长度不少于N（N=10000）的数据。  对每种查找算法进行验证，记录并比较查找的时间消耗。   1. **性能对比**：   分析不同查找算法的时间复杂度。  给出不同算法的时间消耗对比结果  **实验任务二：排序类设计**   1. **设计排序类**：   创建一个排序类，包含不同排序算法的实现。   1. **顺序表排序算法**：   实现至少1种顺序表排序算法，如冒泡排序或选择排序。   1. **树表堆排序算法**：   实现基于树表的堆排序算法。   1. **链式基数排序算法**：   实现基于链式表的基数排序算法。   1. **数据生成与验证**：   使用随机函数生成长度不少于N（N=10000）的数据。  对每种排序算法进行验证，记录并比较排序的时间消耗。   1. **性能对比**：   分析不同排序算法的时间复杂度。  给出不同算法的时间消耗对比结果。   1. **测试案例**：   准备测试数据，验证排序算法的正确性和性能。  ：  #include <iostream>  #include <vector>  #include <unordered\_map>  #include <algorithm>  #include <ctime>  #include <cstdlib>  #include <chrono>  class SearchAlgorithms {  public:  // 基于顺序表的查找算法  static int linearSearch(const std::vector<int>& arr, int key) {  for (int i = 0; i < arr.size(); ++i) {  if (arr[i] == key) {  return i;  }  }  return -1;  }  static int binarySearch(const std::vector<int>& arr, int key) {  int left = 0, right = arr.size() - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  if (arr[mid] == key) {  return mid;  }  else if (arr[mid] < key) {  left = mid + 1;  }  else {  right = mid - 1;  }  }  return -1;  }  // 基于树表的查找算法  struct TreeNode {  int val;  TreeNode\* left;  TreeNode\* right;  TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  };  static TreeNode\* insertBST(TreeNode\* root, int key) {  if (root == nullptr) {  return new TreeNode(key);  }  if (key < root->val) {  root->left = insertBST(root->left, key);  }  else {  root->right = insertBST(root->right, key);  }  return root;  }  static TreeNode\* searchBST(TreeNode\* root, int key) {  if (root == nullptr || root->val == key) {  return root;  }  if (key < root->val) {  return searchBST(root->left, key);  }  else {  return searchBST(root->right, key);  }  }  // 基于散列表的查找算法  static int hashTableSearch(const std::unordered\_map<int, int>& hashTable, int key) {  auto it = hashTable.find(key);  if (it != hashTable.end()) {  return it->second;  }  return -1;  }  static int hashTableSearch2(const std::vector<int>& arr, int key) {  std::unordered\_map<int, int> hashTable;  for (int i = 0; i < arr.size(); ++i) {  hashTable[arr[i]] = i;  }  return hashTableSearch(hashTable, key);  }  static std::vector<int> generateRandomData(int N) {  std::vector<int> data(N);  std::srand(std::time(0));  for (int i = 0; i < N; ++i) {  data[i] = std::rand() % (N \* 10);  }  return data;  }  static void timeComparison(int N) {  std::vector<int> data = generateRandomData(N);  int key = data[std::rand() % N];  auto start = std::chrono::high\_resolution\_clock::now();  linearSearch(data, key);  auto end = std::chrono::high\_resolution\_clock::now();  std::cout << "Linear Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";  std::vector<int> sortedData = data; // 创建一个副本进行排序  std::sort(sortedData.begin(), sortedData.end());  start = std::chrono::high\_resolution\_clock::now();  binarySearch(sortedData, key);  end = std::chrono::high\_resolution\_clock::now();  std::cout << "Binary Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";  TreeNode\* root = nullptr;  for (int num : data) {  root = insertBST(root, num);  }  start = std::chrono::high\_resolution\_clock::now();  searchBST(root, key);  end = std::chrono::high\_resolution\_clock::now();  std::cout << "Binary Search Tree Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";  std::unordered\_map<int, int> hashTable;  for (int i = 0; i < data.size(); ++i) {  hashTable[data[i]] = i;  }  start = std::chrono::high\_resolution\_clock::now();  hashTableSearch(hashTable, key);  end = std::chrono::high\_resolution\_clock::now();  std::cout << "Hash Table Search Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";  start = std::chrono::high\_resolution\_clock::now();  hashTableSearch2(data, key);  end = std::chrono::high\_resolution\_clock::now();  std::cout << "Hash Table Search 2 Time: " << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count() << " microseconds\n";  }  };  int main() {  int N = 100000; // 可以根据需要调整数据规模  SearchAlgorithms::timeComparison(N);  return 0;  }    #include <iostream>  #include <vector>  #include <algorithm>  #include <ctime>  #include <cstdlib>  #include <queue>  #include <list>  class Sorting {  public:  // 顺序表排序：快速排序  void quickSort(std::vector<int>& arr, int left, int right) {  if (left >= right) return;  int i = left, j = right;  int pivot = arr[left + (right - left) / 2];  while (i <= j) {  while (arr[i] < pivot) i++;  while (arr[j] > pivot) j--;  if (i <= j) {  std::swap(arr[i], arr[j]);  i++;  j--;  }  }  quickSort(arr, left, j);  quickSort(arr, i, right);  }  // 树表排序：堆排序  void heapSort(std::vector<int>& arr) {  std::priority\_queue<int, std::vector<int>, std::greater<int>> minHeap;  for (int num : arr) {  minHeap.push(num);  }  for (int i = 0; i < arr.size(); ++i) {  arr[i] = minHeap.top();  minHeap.pop();  }  }  // 链式基数排序  void radixSort(std::vector<int>& arr) {  const int base = 10;  std::vector<std::list<int>> buckets(base);  int maxVal = \*std::max\_element(arr.begin(), arr.end());  for (int exp = 1; maxVal / exp > 0; exp \*= base) {  for (int num : arr) {  buckets[(num / exp) % base].push\_back(num);  }  int i = 0;  for (auto& bucket : buckets) {  for (int num : bucket) {  arr[i++] = num;  }  bucket.clear();  }  }  }  // 生成随机数据  std::vector<int> generateRandomData(int size) {  std::vector<int> data(size);  std::srand(std::time(0));  for (int i = 0; i < size; ++i) {  data[i] = std::rand() % 100000;  }  return data;  }  // 测试排序算法性能  void testSortingAlgorithms(int N) {  std::vector<int> data = generateRandomData(N);  std::vector<int> dataCopy = data;  // 快速排序  clock\_t start = clock();  quickSort(dataCopy, 0, dataCopy.size() - 1);  clock\_t end = clock();  std::cout << "Quick Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";  dataCopy = data;  // 堆排序  start = clock();  heapSort(dataCopy);  end = clock();  std::cout << "Heap Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";  dataCopy = data;  // 基数排序  start = clock();  radixSort(dataCopy);  end = clock();  std::cout << "Radix Sort Time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " seconds\n";  }  };  int main() {  Sorting sorting;  sorting.testSortingAlgorithms(10000);  return 0;  } |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶   1. **快速排序**：快速排序是一种高效的排序算法，平均时间复杂度为O(n log n)。在实际应用中，它通常比其他O(n log n)的排序算法（如归并排序和堆排序）更快，因为它在处理大数据集时具有良好的局部性，且不需要额外的存储空间。 2. **堆排序**：堆排序也是一种时间复杂度为O(n log n)的排序算法。它的主要优点是空间复杂度为O(1)，因为它只需要一个固定大小的额外空间来构建堆。然而，堆排序的缓存性能较差，因为它不具有良好的局部性，这可能导致在实际应用中比快速排序慢。 3. **基数排序**：基数排序是一种非比较排序算法，其时间复杂度为O(nk)，其中k是数字的最大位数。基数排序在处理具有固定范围的整数时非常有效。它的主要缺点是需要额外的存储空间来存储桶，并且如果k较大，性能可能会下降。   在实际测试中，快速排序通常会比堆排序和基数排序更快，尤其是在数据集较大的情况下。堆排序和基数排序的相对性能取决于数据的特性，例如数据的分布和范围。基数排序在处理具有固定范围的整数时表现良好，而堆排序在空间效率方面具有优势 |
| 四、感想、体会、建议∶  **感想与体会**   1. **理论与实践的结合**：通过实验，深刻理解了查找和排序算法背后的理论，并将其应用于实际问题的解决中。 2. **算法多样性**：体会到了不同查找和排序算法的特点，以及它们在不同场景下的适用性。 3. **性能重要性**：意识到了算法性能的重要性，特别是在处理大量数据时，算法的效率对应用的性能有直接影响。 4. **代码实现的挑战**：在将算法转化为代码的过程中，遇到了一些实现上的挑战，如算法逻辑的准确表达、边界条件的处理等。 5. **优化的必要性**：学习了对现有算法进行优化的方法，以适应特定的数据特性或提高性能。   **建议**   1. **更多实践**：建议增加更多的实践机会，通过解决实际问题来加深对算法的理解和应用能力。 2. **算法比较**：建议对不同算法在不同数据集上的表现进行比较，以更好地理解它们的优缺点和适用场景。 3. **性能分析工具**：建议学习使用性能分析工具来评估算法的效率，包括时间复杂度和空间复杂度的考量。 4. **算法可视化**：建议使用算法可视化工具来帮助理解算法的执行过程，特别是在排序和查找过程中元素的变化。 5. **并行计算**：鉴于现代计算硬件的多核特性，建议探索并行版本的查找和排序算法，以提高处理速度。 6. **内存管理**：建议关注算法实现中的内存使用情况，优化内存管理，尤其是在处理大规模数据集时。 7. **算法选择**：建议根据实际应用场景的需求，选择合适的算法，考虑因素包括数据规模、数据特性、时间限制等。 8. **持续学习**：建议持续关注新的算法研究成果，不断更新和扩展算法知识库 |
| 实验成绩∶  指导教师签名：  年 月 日 |