山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202300130207 | 姓名：李臣思 | | 班级：23.5 |
| 实验题目：1.跳表实现与分析 | | | |
| 实验学时：6 | | 实验日期：25.3.26 | |
| 实验目的：实现并分析跳表结构。 | | | |
| 软件开发工具：  Visual Studio 2022 | | | |
| 1. 实验内容 2. 构造并实现**跳表ADT**，跳表ADT 中应包括**初始化、查找、插入、删除指**   **定关键字的元素、删除关键字最小的元素、删除关键字最大的元素**等基本操作。   1. **分析各基本操作的时间复杂性**。 2. 能对跳表维护动态数据集合的效率进行实验验证，获得一定量的实验数   据，如**给定随机产生1000 个数据并将其初始化为严格跳表**，在此基础上进行一系列插入、删除、查找操作（操作序列也可以随机生成），**获得各种操作的平均时间（或统计其基本操作**  **个数）；获得各操作执行时间的变化情况**。应该是越来越大，当大到一定程度后应该进行适当的整理，需设计相应的整理算法，并从数量上确定何时较为合适；能和其他简单线性数据结构，如排序数组上的折半查找进行各类操作效率上的数量对比。   1. 数据结构与算法描述（整体思路描述，所需要的数据结构与算法） 2. **整体思路描述：**   实现了一个跳表（skiplist）和一个基于二分查找的有序数组，实现跳表的各项功能以及动态数据集合的高效维护，通过菜单交互比较二者之间的操作与性能差别。通过测量操作时间，验证跳表在动态操作中的优势，并支持重建跳表以优化性能。   1. 跳表（skiplist）：   数据结构：跳表节点包括同层指向下一个节点的节点指针**next**，指向下层节点的指针**down**，以及存储的关键字的值**val**。在跳表类中包含私有成员一个**头节点head**指向最高层的起始节点。**最大层数maxlevel预设为32**，限制跳表的高度。  核心算法：**查找函数（bool serach(int target)）:**从头节点开始向下寻找，如果同层的下一个节点的值小于要寻找的值或者下一个节点为空时，则寻找指针p从该节点指向下一层，在下一层中继续向后遍历，循环往复。直到找到某个节点的值等于target或者某节点的同层下一个节点值等于target，则循环结束，返回true；当遍历完所有的层级，仍未找到target，则返false。平均**O(log n)**，最坏**O(n)**。  **插入函数（void insert(int target)）：**从头节点开始向下查找插入的位置，当该节点的next为空或者同层的下一个节点的值大于该值时，则将该节点记录到栈中，直到找到最后一层该插入的位置，首先将该元素插入到最后一层中。最后栈中记录的是在寻找插入位置时，每次需要向下转移一层时的所指的节点。  然后以当前的时间为种子生成随机数，与上1，得到50%的概率选择是否将该节点向上继续增加节点。当栈不为空时，如果增加，则从栈中得到上一层压入的节点，在该节点后插入target；如果选择不增加，则结束循环，循环往复。直到栈为空或者停止向上继续进一层增加target的节点。时间复杂度平均为**O(log n)**,最坏情况下是**O(n)**。  **删除函数（void erase(int target）):**首先寻找出应该删除的节点的每层前一个节点的位置，与函数search函数思路一致，如果下一个节点不是应该删除的元素的节点，则向下继续寻找；如果找到某个节点的下一个节点的关键字的值等于target，那么删除这个‘下一个节点’，继续向下一层寻找出应该删除的节点的位置。时间平均 **O(log n)**，最坏 **O(n)**。  **初始化函数(void initialize(int\* a,int n)):**删除原本的跳表，重新建立各层头节点，再根据数组a多次调用insert函数。时间复杂度为**O(n log n)**.  **删除关键字最小的元素(void erasemin()):**从头节点一直向下直到最后一层，此时指针指向最底层的第一个元素，即关键字最小的元素，然后调用erase函数删除该元素。定位最小元素O(1),删除操作O(log n),总时间复杂度为O(log n)。  **删除关键字最大的元素(void erasemax()):**从头节点先向下找到最底层链表然后同级向右寻找到最后一个元素，即为关键字最大的元素，调用erase函数删除该元素即可。定位最大元素O(n)，删除操作O(log n),所以总时间复杂度为O(n).  **重建函数(void rebuild())**：收集所有元素到栈中，清空原始跳表后重新依次插入元素，优化层数分布。  **输出函数(void print()):**输出跳表各层。便于调试。   1. 有序数组(erfen类)：   **数据结构：**动态数组(a)，存储有序元素，通过冒泡排序初始化。数组长度(n)记录当前元素数量。  **核心算法：插入函数(void insert(int target)):**使用二分查找确定插入的位置，创建新的数组并复制元素以及插入的元素，时间复杂度为**O(n)**.  **删除函数(voiderase(int target)):**二分查找找到目标位置，复制剩余元素到新的数组中。时间复杂度为**O(n)**.  **查找函数(void serach(int target)):**标准二分查找。时间复杂度为O(log n).   1. **实验设计与对比：**   操作时间测量：   * 使用 high\_resolution\_clock 记录每个操作的耗时（纳秒级）。 * 统计总时间和平均时间，支持性能分析。   性能对比项：   * 跳表 vs 有序数组：对比插入、删除、查找操作的时间差异。 * 跳表重建：验证重建对性能优化的效果。   用户交互：   * 初始页面可以选择各项操作，选择初始化方式。 * 每次操作均会输出各项操作的时间消耗。  1. 测试结果（测试输入，测试输出）   用户交互界面：    图 1  选择初始化方式，这里我直接采用随机生成1000个随机数进行初始化：    图 2  然后为了方便展示删除以及插入等各项操作后的结果重新初始化输入15个数字：    图 3  插入元素-9：    图 4  查询元素6：    图 5  删除元素7：    图 6  删除最小的元素以及最大的元素：    图 7    图 8  查看与二分查找的查找元素时间对比：    图 9  在后续的多次尝试中发现，跳表查询的时间消耗相对于有序数组的折半查找更加不稳定，在很多时候是大于折半查找的消耗的时间。  跳表重建优化层级：     1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径） 2. **跳表的层级随机生成：**   当前使用gen()&1随机决定是否继续向上层插入，可能会导致层数分布不均。改进：可以采用更标准的概率控制或者某些算法控制（如p=0.5的几何分布）。   1. **有序数组的排序效率**：   冒泡排序进行初始化时间复杂度为O(n^2)，性能较差，可以改用快速排序或者归并排序O(n log n)。   1. **跳表初始化重复插入：**   skiplist::initialize通过循环调用insert初始化数据，导致多次随机层数决策和指针调整，时间复杂度为 O(n log n)。可以直接批量构建底层链表后逐层提升节点，降低初始化开销。   1. **erasemax()优化**：   当前定位最大元素需要遍历底层链表(O(n))。可以在跳表中维护一个尾指针  或缓存最大值，将时间复杂度优化至O(log n).   1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）   class skiplist {  private:  node\* head; // 跳表的头节点  const int maxlevel = 32; // 跳表的最大层数  //stack<node\*> stack1; // 如果需要，可以使用栈来保存节点  public:  skiplist() {  node\* below = nullptr;  for (int i = 0; i < maxlevel; i++) {  // 创建跳表的多层头节点，每一层的头节点指向下层的头节点  head = new node(nullptr, below, -1); // 创建多层头节点，初始值为 -1  below = head; // 设置下层节点为当前层的头节点  }  }  ~skiplist() {  node\* curr = head;  while (curr) {  node\* temp = curr;  // 删除当前层的所有节点  while (temp->next) {  node\* del = temp->next;  temp->next = del->next;  delete del;  }  node\* down = curr->down; // 获取下一层  delete curr; // 删除当前层头节点  curr = down; // 进入下一层  }  }  // 查找元素，返回是否找到 (O(log n))  bool search(int target);  // 初始化跳表，用一个数组填充跳表 (O(n log n))  void initialize(int\* x, int size);  // 向跳表中插入一个元素 (O(log n))  void insert(int target);  // 从跳表中删除一个元素 (O(log n))  void erase(int target);  // 删除跳表中最小的元素 (O(log n))  void erasemin();  // 删除跳表中最大的元素 (O(log n))  void erasemax();  // 打印跳表中的每一层 (O(n))  void print();  // 重建跳表，以优化层数 (O(n log n))  void rebuild();  };  // 查找跳表中是否存在目标值 (O(log n))  bool skiplist::search(int target) {  node\* p = head;  while (p) {  // 从当前层向右查找，直到找到目标值或者越过目标值  while (p->next && p->next->val < target) {  p = p->next;  }  if (!p->next || p->next->val > target) {  // 如果未找到目标值，进入下层继续查找  p = p->down;  } else {  // 找到目标值，返回true  return true;  }  }  // 未找到目标值，返回false  return false;  }  // 向跳表插入一个目标值 (O(log n))  void skiplist::insert(int target) {  stack<node\*> stack1; // 用于存储每一层插入的位置  node\* p = head;  // 从最上层开始向下查找合适的插入位置  while (p) {  while (p->next && p->next->val < target) {  p = p->next;  }  stack1.push(p); // 将当前层的节点保存到栈中  p = p->down; // 进入下一层  }  bool insertup = true;  node\* newp = NULL;  // 在栈中保存的每一层中插入新节点  while (insertup && !stack1.empty()) {  node\* insert = stack1.top();  stack1.pop();  insert->next = new node(insert->next, newp, target);  newp = insert->next;  // 随机决定是否继续插入到更高一层  if ((gen() & 1) == 0) {  insertup = false; // 停止向上插入  }  }  // 如果插入到最顶层，增加一层  if (insertup) {  head = new node(new node(NULL, newp, NULL), head, -1);  }  }  // 删除跳表中的指定值 (O(log n))  void skiplist::erase(int target) {  node\* p = head;  bool found = false;  while (p) {  while (p->next && p->next->val < target) {  p = p->next;  }  if (!p->next || p->next->val > target) {  p = p->down; // 进入下一层继续查找  } else {  // 找到目标值，进行删除  found = true;  node\* todelete = p->next;  p->next = p->next->next;  p = p->down; // 进入下层  delete todelete;  }  }  if (!found) {  cout << -1 << endl; // 如果未找到目标值，输出 -1  }  }  // 初始化跳表，从数组 x 中读取数据 (O(n log n))  void skiplist::initialize(int\* x, int size) {  // 清除旧数据  node\* cur = head;  while (cur) {  node\* tmp = cur->down;  while (cur->next) {  node\* del = cur->next;  cur->next = del->next;  delete del;  }  delete cur;  cur = tmp;  }  // 重新构建跳表  node\* below = nullptr;  for (int i = 0; i < maxlevel; i++) {  below = new node(nullptr, below, -1);  }  head = below; // 设置新构建的头节点  // 使用 insert 函数向跳表插入元素  for (int i = 0; i < size; i++) {  insert(x[i]);  }  }  // 删除跳表中最小的元素 (O(log n))  void skiplist::erasemin() {  node\* p = head;  while (p->down) {  p = p->down; // 进入最底层  }  if (!p->next) {  cout << "链表为空,无法删除!" << endl;  return;  }  int minval = p->next->val; // 获取最小值  erase(minval); // 删除最小值  cout << "成功删除关键词最小的元素:" << minval << endl;  }  // 删除跳表中最大的元素 (O(log n))  void skiplist::erasemax() {  node\* p = head;  while (p->down) {  p = p->down; // 进入最底层  }  if (p->next == NULL) {  cout << "跳表为空" << endl;  }  while (p->next) {  p = p->next; // 进入最右端  }  int maxval = p->val; // 获取最大值  erase(maxval); // 删除最大值  cout << "成功删除关键词最大的元素:" << maxval << endl;  }  // 打印跳表的所有层  void skiplist::print() {  node\* layer = head;  int level = maxlevel;  while (layer) {  if (layer->next) {  cout << "Lv" << level << ":\t";  node\* curr = layer->next; // 跳过头节点  while (curr) {  cout << curr->val;  if (curr->next) cout << " -> "; // 格式化输出  curr = curr->next;  }  cout << endl;  }  layer = layer->down; // 进入下一层  level--;  }  }  // 重建跳表，通过排序和重新插入优化层级结构  void skiplist::rebuild() {  stack<int> elements;  // 收集元素（正序）  node\* p = head;  while (p->down) {  p = p->down; // 进入最底层  }  p = p->next;  if (p == NULL) {  cout << "链表为空" << endl;  return;  }  while (p) {  elements.push(p->val); // 将元素保存到栈中  p = p->next;  }  stack<int> temp; // 用于重建跳表  while (!elements.empty()) {  temp.push(elements.top()); // 将元素倒入临时栈中  elements.pop();  }  // 清除旧数据  initialize(nullptr, 0);  // 插入新数据  while (!temp.empty()) {  insert(temp.top());  temp.pop();  }  cout << "跳表重建完成，层级已优化" << endl;  }  int randint(int l, int r) {  return gen() % (r - l + 1) + l; // 返回 [l, r] 范围内的随机数  } | | | |