TAVE 서기

|  |  |  |  |
| --- | --- | --- | --- |
| **서기 내용** | | | |
| **서기 일자** | 20.08.25 | **서기** | 차영훈 |
| **주제** | 정렬 | | |
| **시간** | 11:00~13:00 | **장소** | 영통역 스터디 플로우 |
| **스터디**  **인원** | 방수영 송나은 최인아 양효빈 차영훈 | | |
| **내용** | | | |
| **배운 내용** | 1. 버블 정렬, 선택 정렬  2. 삽입 정렬  3. 병합 정렬  4. 힙 정렬  5. 퀵 정렬 | | |
| 1. 버블정렬  1-1. 방법     * 인접한 2개의 key를 비교하여 * 순서대로 되어있지 않으면 * 서로 교환하는 * 과정을 반복   1-2. cpp 코드 구현  void bubble\_sort(int list[], int n){      int temp;      for(int i=n-1; i>0; i--) {          // 0 ~ (i-1)까지 반복          for(int j=0; j<i; j++) {              // j번째와 j+1번째의 요소가 크기 순이 아니면 교환              if(list[j] > list[j+1]){                  temp = list[j];                  list[j] = list[j+1];                  list[j+1] = temp;              }          }      }  }  1-3. 특징   * 구현이 매우 간단 * But 하나의 요소가 가장 왼쪽에서 오른쪽으로 이동하려면 모든 다른 요소와 교환 * 매우 비효율적!! * 시간 복잡도 O(n^2) * 너무 느려서 잘 안씀   1-4. 백준 문제 2750 코드  int main() {      int n;      cin >> n;      int\* list = new int[n];      for(int i=0; i<n; i++) cin >> list[i];      bubble\_sort(list, n);      for(int i=0; i<n; i++) cout << list[i] << endl;      delete list;      return 0;  }     * 버블 정렬로 해결 가능   2. 선택 정렬  2-1. 정렬방법    2-2. cpp코드 구현  void selection\_sort(int list[], int n){      int least, temp;      // 마지막 숫자는 자동으로 정렬되기 때문에 (숫자 개수-1) 만큼 반복한다.      for(int i=0; i<n-1; i++){ // i 가 맨앞          least = i;          // 최솟값을 탐색한다.          for(int j=i+1; j<n; j++){              if(list[j] < list[least]) least = j;          }          // 최솟값이 자기 자신이면 자료 이동을 하지 않는다.          if(i != least){              temp = list[i];              list[i] = list[least];              list[least] = temp;          }      }  }  2-3. 특징   * 구현이 매우 간단 * 버블 정렬보다는 비교적 빠름 * 시간 복잡도 O(n^2) * 데이터 크기가 클수록 효율이 많이 떨어짐 * 이것도 너무 느려서 잘 안씀   2-4. 백준 문제 2750 코드  int main() {      int n;      cin >> n;      int\* list = new int[n];      for(int i=0; i<n; i++) cin >> list[i];      selection\_sort(list, n);      for(int i=0; i<n; i++) cout << list[i] << endl;      delete list;      return 0;  }  1-7. C++ 코드 구현  - 더미헤드가 있는 단일 연결 리스트  - 간단한 스택을 구현한다.    #include <iostream>  using namespace std;  template <typename T>  class Node {  public:      T data;      Node<T>\* link;      Node() : link(nullptr) { }      Node(T \_data, Node<T>\* \_link) : data(\_data), link(\_link) { }      ~Node() { delete link; }  };     * 선택 정렬로 해결 가능!   3. 병합 정렬  3-1. 분할 정복 법  - 분할: 해결하고자 하는 문제를 작은 크기의 동일한 문제들로 분할  - 정복: 각각의 작은 문제를 순환적으로 해결  - 합병: 작은 문제의 해를 합하여(merge) 원래문제에 대한 해를 구함(전체적인 흐름)  - 데이터가 저장된 배열을 절반으로 나눔  - 각각을 순환적으로 정렬  - 정렬된 두 개의 배열을 합쳐 전체를 정렬  3-2. 의사표현 코드  mergeSort(A[], p, r)  if (p < r) then  q = (p + r) /2;  mergeSort(A, p, q);  mergeSort(A, q + 1, r);  merge(A, p, q, r);  merge(A[], p, q, r)  정렬되어 있는 두 배열 A[p...q]와 A[q + 1...r]을 합하여  정렬된 하나의 배열 A[p...r]을 만든다.  3-3. 시간 복잡도  순환 호출의 깊이 만큼의 합병 단계 \* 각 합병 단계의 비교 연산 = O(N\*logN)  3-4. 장점  - 안정적인 정렬 방법 : 데이터의 분포에 영향을 덜 받아 입력 데이터가 무엇이든 간에 정렬되는 시간은 동일(O(nlongn))  - 레코드를 연결 리스트로 구성하면, 링크 인덱스만 변경되므로 데이터의 이동은 무시할 수 있을 정도로 작아짐 : 제자리 정렬로 구현 가능  - 크기가 큰 레코드를 정렬할 경우에 연결 리스트를 사용한다면, 합병 정렬은 퀵 정렬을 포함한 다른 어떤 정렬 방법보다 효율적  3-4. 단점  - 레코드를 배열로 구성하면, 임시 배열이 필요함 : 제자리 정렬이 아님  - 레코드들의 크기가 큰 경우에는 이동 횟수가 많으므로 매우 큰 시간적 낭비를 초래  3-5. 코드  #include <iostream>  using namespace std;  void merge(int\* arr, int p, int q, int r)  {  int i, j, k;  int temp[1000];  i = p;  k = 0;  j = q + 1;  while (i <= q && j <= r) {  if (arr[i] < arr[j]) {  temp[k++] = arr[i++];  }  else {  temp[k++] = arr[j++];  }  }  while (i <= q) {  temp[k++] = arr[i++];  }  while (j <= r) {  temp[k++] = arr[j++];  }  for (i = p; i <= r; i++) {  arr[i] = temp[i - p];  }  }  void mergeSort(int\* arr, int p, int r)  {  int q;  if (p < r) {  q = (p + r) / 2;  mergeSort(arr, p, q);  mergeSort(arr, q + 1, r);  merge(arr, p, q, r);  }  }  int main()  {  int arr[1000];  int n = 0;  cin >> n;  for(int i = 0; i < n; i++) {  cin >> arr[i];  }    mergeSort(arr, 0, n - 1);  cout << "-------merge sort-------" << endl;  for (int i = 0; i < n; i++) {  cout << arr[i] << endl;  }  }  4. 힙 정렬  4-1. 자료 구조 힙  - 완전 이진 트리의 형태로, 우선 순위를 중심으로 정렬할 필요가 있을 때 유용하게 활용되는 자료구조이다.  - 부모 노드의 값이 자식 노드의 값보다 항상 크거나 작은 이진트리.  4-2. 힙 정렬  힙 정렬(Heap Sort)는 이러한 Heap을 활용하는 정렬이다.    이를 HeapSort하게 되면    4-3. 백준 문제 2751 코드  #include <iostream>  using namespace std;  const int MAX = 1000000;  int heap[MAX];  void swap(int a, int b)  {  int temp = heap[a];  heap[a] = heap[b];  heap[b] = temp;  }  void max\_heap(int root, int bottom) {    int max\_child;  int left\_child = root \* 2 + 1;  int right\_child = root \* 2 + 2;  if (left\_child <= bottom)  //bottom 값을 통해 leaf node를 판단한다. 무한재귀호출방지.  //자식노드가 0개가 아닌경우(1개 또는 2개).  {  if (left\_child == bottom)  //right\_child가 없는 경우. == 자식노드가 1개인 경우.  max\_child = left\_child;  else {  // 자식 노드가 2개인 경우.  if (heap[left\_child] <= heap[right\_child])  max\_child = right\_child;  else max\_child = left\_child;  }  if (heap[root] < heap[max\_child]) {  swap(root, max\_child);  max\_heap(max\_child, bottom);  //swap을 하고 나서 하위 서브트리에서 heap 조건 만족하지 않을 수 있기 때문에  // swap한 max\_child가 root인 서브트리를 모두 재귀적으로 재검사, heap화.  }  }  }  void heap\_sort(int size)  {  int index;  for (index = size / 2 - 1; index >= 0; index--)  // 비정렬된 배열을 heap 형태로 만들어줌.  //모든 부모-자식간 관계(1세대 차이)에 적용.  // 완전 이진 트리이기 때문에 사용할 수 있는 수식.  max\_heap(index, size - 1);  for (index = size - 1; index >= 0; index--) // 이진탐색트리x.'이진트리', left, right 정렬x.  {  swap(0, index); //0(root)가 최대값.  max\_heap(0, index - 1); //다시 heap화.  }  }  int main()  {  int n;  scanf("%d", &n);  for (int i = 0; i < n; i++)  scanf("%d", &heap[i]);  heap\_sort(n);  //오름차순  for (int i = 0; i < n; i++)  printf("%d\n", heap[i]);  //내림차순  /\*  for (int i = n - 1; i >= 0; i--)  printf("%d\n", heap[i]);  \*/  return 0;  }  4-4. Heap vs Binary Search Tree(이진 탐색 트리).  힙과 이진 탐색 트리 모두 이진트리라는 점에서 공통점을 가지만 노드값이 다소 다르게 구성되어있다. 힙은 각 노드의 값이 자식노드보다 큰 반면, 이진 탐색 트리는 왼쪽 자식노드가 제일 작고 부모노드가 그 다음 크며 오른쪽 자식노드가 가장 큰 값을 가진다. 힙은 우선순위(키)에 따른 정렬에, 이진 탐색 트리는 탐색에 강점을 가진다.  5. 퀵 정렬  5-1. 정렬 방법    여기서 기준이되는 수(PIVOT)을 하나 잡는다. 배열의 중앙을 PIVOT으로 잡겠다. ( 맨앞을 잡고 하는 방법도 있음)  중앙을 PIVOT으로 잡았으면 왼쪽끝과 오른쪽끝을 LEFT와 RIGHT 로 잡는다.    LEFT와 RIGHT는 PIVOT과 비교하여 LEFT는 PIVOT보다 큰수를 만나면,  RIGHT는 PIVOT보다 작은수를 만나면 멈춰 서로 교환하게 되어있다.  이렇게 조건이 맞으면 멈춘 뒤 서로 교환을 한다.        LEFT가 RIGHT보다 커질때까지 반복을한다. (단, PIVOT을 만나면 더이상 가지않음)  LEFT는 6에서 멈추고 RIGHT는 3(PIVOT)에서 멈추었다. 이상태로 교환.    교환을 하게되면 LEFT와 RIGHT는 한칸씩 움직인다.    이때, LEFT가 RIGHT보다 커졌기때문에 비교를 멈추고 왼쪽부분, 오른쪽부분을 다시 퀵정렬 하게된다.  왼쪽부분을 퀵정렬, 오른쪽부분을 퀵정렬 을 하다보면 보든 배열이 정렬이 되어진다    5-2. cpp 코드 구현  #include <iostream>  using namespace std;  //퀵정렬  int n,cnt, quick[10000001];  void quickSort(int i, int j)  {  if(i>=j) return;  int pivot = quick[(i+j)/2];  int left = i;  int right = j;    while(left<=right)  {  while(quick[left]<pivot) left++;  while(quick[right]>pivot) right--;  if(left<=right)  {  swap(quick[left],quick[right]);  left++; right--;  }  }  quickSort(i,right);  quickSort(left,j);  }  int main()  {  scanf("%d",&n);  for(int i = 0; i < n; i++)  scanf("%d",&quick[i]);  quickSort(0,n-1);  for(int j = 0; j < n; j++) // 출력  printf("%d\n",quick[j]);  } | | |
| **과제할당** | 백준 문제 <https://www.acmicpc.net/problem/11399> 풀기  각자 문제 하나씩 정해서 풀기 | | |
| **특이사항** |  | | |
| **비고** |  | | |