1. **HW4 Report**
2. **-AVL implementation-**

**2013104352 한진희**

**File Structure and Compilation Procedure**

|  |  |
| --- | --- |
| **AVL\_Tree** | **ADT\_Tree** |
|  | **EMB0000061062ec** |

기존 HW3에서 사용하였던 코드에서 ADT\_tree의 코드를 AVL\_tree로 바꾸어 앞서 구했던 ADT\_tree와의 실행결과를 비교한다.

**목표 결과**

**ADT\_TREE와 AVL\_TREE의 검색 시간의 차이를 구한다.**

|  |
| --- |
| RUN ‘main\_tree’ and open ‘profile\_tree.txt’ under ADT folder |
|  |
| RUN ‘main\_tree’ and open ‘profile\_tree.txt’ under AVL folder |
|  |

* 1. **1.ADT\_tree.h(튜터시간에 작성+search,balance\_right문 작성)**

|  |
| --- |
| 1. AVL\_tree.h |
| 1. #include <stdio.h> 2. #include <stdlib.h> 3. #define EH 0 4. #define RH -1 5. #define LH 1 6. typedef struct node1{ 7. int data; 8. struct node1\* left; 9. struct node1\* right; 10. int balance; 11. }T\_NODE; 12. typedef struct{ 13. int count; 14. T\_NODE\* root; 15. }AVL\_TREE; 16. AVL\_TREE\* create\_avl\_tree(); 17. T\_NODE\* rotate\_left(T\_NODE\* root); 18. T\_NODE\* rotate\_right(T\_NODE\* root); 19. bool AVL\_insert(AVL\_TREE\* tree, int data); 20. T\_NODE\* insert\_rotate(T\_NODE\* root, T\_NODE\* new\_node, bool\* taller); 21. T\_NODE\* balance\_left(T\_NODE\* root, bool\* taller); 22. T\_NODE\* balance\_right(T\_NODE\* root, bool\* taller); 23. void traverse\_postorder(T\_NODE\* root); 24. void AVL\_print(AVL\_TREE\* tree); 25. T\_NODE\* search\_avl(T\_NODE\* root, int key); |

**2.AVL\_tree.c (튜터시간에 작성+search,balance\_right문 작성)**

|  |
| --- |
| **ADT\_tree.c** |
| #include "AVL\_tree.h"  AVL\_TREE\* create\_avl\_tree(){  AVL\_TREE\* tree = (AVL\_TREE\*)malloc(sizeof(AVL\_TREE));  tree->count = 0;  tree->root = NULL;  return tree;  }  T\_NODE\* rotate\_left(T\_NODE\* root){  T\_NODE\* new\_root;  new\_root = root->right;  root->right = new\_root->left;  new\_root->left = root;  return new\_root;  }  T\_NODE\* rotate\_right(T\_NODE\* root){  T\_NODE\* new\_root;  new\_root = root->left;  root->left = new\_root->right;  new\_root->right = root;  return new\_root;  }  bool AVL\_insert(AVL\_TREE\* tree, int data){  T\_NODE\* new\_root;  bool taller;  new\_root = (T\_NODE\*)malloc(sizeof(T\_NODE));  if(!new\_root) return false;  new\_root->balance = EH;  new\_root->right = NULL;  new\_root->left = NULL;  new\_root->data = data;  tree->root = insert\_rotate(tree->root, new\_root, &taller);  (tree->count)++;  return true;  }  T\_NODE\* insert\_rotate(T\_NODE\* root, T\_NODE\* new\_root, bool\* taller){  if(root == NULL){  root = new\_root;  \*taller = true;  return root;  }  if((new\_root->data)<(root->data)){  root->left = insert\_rotate(root->left, new\_root, taller);  if(\*taller){  switch(root->balance){  case LH: root = balance\_left(root, taller);  break;  case EH: root->balance= LH;  break;  case RH: root->balance = EH;  \*taller = false;  break;  }  }  return root;  }  else{  root->right = insert\_rotate(root->right, new\_root, taller);  if(\*taller){  switch(root->balance){  case LH: root->balance = EH;  \*taller = false;  break;  case EH: root->balance= RH;  break;  case RH: root = balance\_right(root, taller);  break;  }  }  return root;  }  return root;  }  T\_NODE\* balance\_left(T\_NODE\* root, bool\* taller){  T\_NODE\* right\_tree;  T\_NODE\* left\_tree;  left\_tree = root->left;  switch(left\_tree->balance){  case LH:  root->balance = EH;  left\_tree->balance = EH;  root = rotate\_right(root);  \*taller = false;  break;  case EH:  exit(0);  case RH:  right\_tree = left\_tree->right;  switch(right\_tree->balance){  case LH:  root->balance = RH;  left\_tree->balance = EH;  break;  case EH:  root->balance = EH;  left\_tree->balance = EH;  break;  case RH:  root->balance = EH;  left\_tree->balance = LH;  break;  }  right\_tree->balance = EH;  root->left = rotate\_left(left\_tree);  root = rotate\_right(root);  \*taller = false;  }  return root;  }  T\_NODE\* balance\_right(T\_NODE\* root, bool\* taller){  T\_NODE\* right\_tree;  T\_NODE\* left\_tree;  right\_tree = root->right;  switch(right\_tree->balance){  case RH:  root->balance = EH;  right\_tree->balance = EH;  root = rotate\_left(root);  \*taller = false;  break;  case EH:  exit(0);  case LH:  left\_tree = right\_tree->left;  switch(left\_tree->balance){  case RH:  root->balance = LH;  right\_tree->balance = EH;  break;  case EH:  root->balance = EH;  right\_tree->balance = EH;  break;  case LH:  root->balance = EH;  right\_tree->balance = RH;  break;  }  left\_tree->balance = EH;  root->right = rotate\_right(right\_tree);  root = rotate\_left(root);  \*taller = false;  }  return root;  }  void traverse\_preorder(T\_NODE\* root){  if(root != NULL){  printf("%d ",root->data);  traverse\_preorder(root->left);  traverse\_preorder(root->right);  }  }  void AVL\_print(AVL\_TREE\* tree){  printf("AVL\_TREE\n");  printf("size : %d\n",tree->count);  printf("data : ");  traverse\_preorder(tree->root);  printf("\n");  }  T\_NODE\* search\_avl(T\_NODE\* root, int key){  if(root == NULL)  return NULL;  if(key < (root->data))  return search\_avl(root->left,key);  else if(key > (root->data))  return search\_avl(root->right,key);  else  return root;  } |

1. +기존에 ADT\_tree의 c,h를 반영했던 코드를 AVL\_tree코드로 수정(main\_tree.c, makefile)

**3. main\_tree.c (과제 공지 파일 참조)**

|  |
| --- |
| **main\_tree.c** |
| #include "ADT\_llist.h"  #include "ADT\_tree.h"  #include <stdlib.h>  #include <stdio.h>  #define sample\_NUM 10000000  int compare1(void\* x, void\* y)  { return \*((int\*)x) - \*((int\*)y);  }void print1(void\* x)  { int \* xp = (int\*) x;  printf(" - int data %d\n", \*xp);  }  int main()  { FILE\* fin = fopen("sample.txt","r");  int\* N = (int\*)malloc(sizeof(int)\*sample\_NUM);  int\* M = (int\*)malloc(sizeof(int)\*sample\_NUM);  int i, iter, cmp\_result;  int search\_num = sample\_NUM-1;  T\_NODE\* search;  LLIST\* new\_llist = create\_list();  BST\_TREE\* new\_avl = create\_avl\_tree();  for(i=0;i<sample\_NUM;i++)  {  \*M = sample\_NUM -i;  fscanf(fin,"%d",N);  AVL\_insert(new\_avl,\*N);  N++;  M++;  }  search = search\_avl(new\_avl->root, search\_num);  printf("iter num = %d\n", search->data);  fclose(fin);  return 0;  } |
| **해석** |
| sample\_NUM을 10000000으로 정의하고 “sample.txt”파일을 읽기모드로 fopen한 것을 FILE 구조체 포인터로 반환한다. 앞서 정의한 sample\_NUM의 주소의 크기만큼 정수 포인터 N, M에 동적 할당을 한다. I, iter, cmp\_result를 정수형으로 선언하고 9999999의 데이터를 가진 search\_num을 정의하여 찾는 값으로 정의한다.  Tree의 Node 포인터형 search를 선언하고 새로운 tree인 new\_avl를 생성한다. 10000000부터 1까지를 포인터 M으로 정의하면서 sample.txt의 정수 데이터를 scan하여 N에 저장한다. 이 N을 전에 정의한 new\_avl에 insert하고 N과 M을 증가시키는 과정을 반복한다.  반복문이 끝나면 new\_avl에서 9999999의 값(sample.txt의 마지막 데이터)을 가진 root를 찾아 search에 저장한다. search의 데이터를 printf하고 fclose 함수를 호출하여 파일 스트림을 닫는다. |

**4. 실행 결과**(profile\_tree.txt)

|  |
| --- |
| ADT\_Tree |
|  |
| AVL\_Tree |
|  |
| **해석** |
| ADT, AVL방식의 tree를 각각 실행하여 나온 profile.txt를 보니 10000000개의 데이터 중 9999999의 데이터를 찾기까지의 시간은 ADT\_tree는 20.05ms, AVL\_tree는 0.00ns가 나왔다.  AVL\_tree의 검색속도가 너무 빨라 저런 0초로 표시되는 것 같다.  예상대로 AVL\_tree가 ADT\_tree보다 검색속도가 빨랐다.  그러나 프로그램을 실행시키면서 AVL\_tree에선 데이터를 넣으면서 실행되는 함수(insert\_rotate, AVL\_insert, balance\_left, rotate\_left 등등)에 대해서는 한 번 실행할 때마다 트리의 노드들의 관계를 최신화해주는 작업이 필요하여 시간이 걸리는 것 같다. |

1. **5. 고찰**

HW3의ADT\_Tree 코드에서 사용했던 부분에서 AVL\_Tree 코드로 수정해서 동작시키는데 크게 어려움은 없었습니다. 기말고사를 공부하면서 AVL\_Tree부분의 insert문을 공부하면서 헷갈리는 부분에 대해서 이번 HW4를 하면서 바로 잡았습니다.

기말고사 시험공부를 하면서 스택부터 그래프까지 복습을 하면서 참 많이 배운 것 같아 뿌듯했습니다.

한 학기동안 수고하셨습니다.