1. **HW3 Report**
2. **-BST implementation-**

**2013104352 한진희**

**File Structure and Compilation Procedure**

|  |  |
| --- | --- |
| EMB0000061062eb | EMB0000061062ec |

**목표 결과**

**LLIST와 TREE의 검색 시간의 차이를 구한다.**

|  |
| --- |
| RUN ‘main\_list’ and open ‘profile\_list.txt’ under llist folder |
| EMB0000061062ed |
| RUN ‘main\_tree’ and open ‘profile\_tree.txt’ under tree folder |
| EMB0000061062ee |

* 1. **ADT\_tree.c (튜터시간에 작성)**

|  |
| --- |
| **ADT\_tree.c** |
| #include "ADT\_tree.h"  BST\_TREE\* create\_bst\_tree(){  BST\_TREE\* tree = (BST\_TREE\*)malloc(sizeof(BST\_TREE));  tree->count = 0;  tree->root = NULL;  return tree;  }T\_NODE\* find\_smallest\_node(T\_NODE\* root){  if(root->left == NULL)  return root;  else  return find\_smallest\_node(root->left);  }T\_NODE\* find\_largest\_node(T\_NODE\* root){  if(root->right == NULL)  return root;  else  return find\_largest\_node(root->right);  }T\_NODE\* search\_bst(T\_NODE\* root, int key){  if(root == NULL)  return NULL;  if(key < (root->data))  return search\_bst(root->left,key);  else if(key > (root->data))  return search\_bst(root->right,key);  else  return root;  }T\_NODE\* add\_bst(T\_NODE\* root, int data){  if(root == NULL){  T\_NODE\* new\_node = (T\_NODE\*)malloc(sizeof(T\_NODE));  if(!new\_node)  return NULL;  new\_node->left = NULL;  new\_node->right = NULL;  new\_node->data = data;  return new\_node;  }  if(data < (root->data)){  root->left = add\_bst(root->left, data);  return root;  }  if(data > (root->data)){  root->right = add\_bst(root->right, data);  return root;  }  }T\_NODE\* delete\_bst(T\_NODE\* root, int data, bool\* success){  T\_NODE\* new\_root;  T\_NODE\* deleted\_node;  T\_NODE\* search;  if(root == NULL){  \*success = false;  return NULL;  }  if(data < (root->data)){  root->left = delete\_bst(root->left,data,success);  } else if(data > (root->data)){  root->right = delete\_bst(root->right,data,success);  } else {  deleted\_node = root;  if(root->left == NULL){// 1. both left and right are NULL  new\_root = root->right;  free(deleted\_node);  \*success = true;  return new\_root;  } else {  if(root->right ==NULL){ // 2. only right is NULL  new\_root = root->left;  free(deleted\_node);  \*success = true;  return new\_root;  } else { // 3. both are not NULL  search = root->left;  while(search->right !=NULL) // most right of left ->left understand??  search = search->right;  root->data = search->data;  root->left = delete\_bst(root->left, search->data, success); // from the root->left so root is not deleted  }  }  }  return root;  }bool BST\_insert(BST\_TREE\* tree, int data){  T\_NODE\* new\_root;  new\_root = add\_bst(tree->root, data);  if(new\_root == NULL)  return false;  else{  tree->root = new\_root;  (tree->count)++;  return true;  }  }void traverse\_inorder(T\_NODE\* root){  if(root !=NULL){  traverse\_inorder(root->left);  printf("%d.",root->data);  traverse\_inorder(root->right);  }  }void traverse\_postorder(T\_NODE\* root){  if(root !=NULL){  traverse\_postorder(root->left);  traverse\_postorder(root->right);  printf("%d.",root->data);  }  }void traverse\_preorder(T\_NODE\* root){  if(root !=NULL){  printf("%d.",root->data);  traverse\_preorder(root->left);  traverse\_preorder(root->right);  }  }bool BST\_delete(BST\_TREE\* tree, int data){  bool success;  T\_NODE\* new\_root;  new\_root = delete\_bst(tree->root, data, &success);  if(success){  tree->root = new\_root;  (tree->count)--;  if(tree->count == 0)  tree->root = NULL;  }  return success;  }void BST\_print(BST\_TREE\* tree, int method){  printf("BST\_TREE\n");  printf("size : %d\n",tree->count);  printf("data: ");  if(method == 0)  traverse\_preorder(tree->root);  else if(method ==1)  traverse\_inorder(tree->root);  else if(method ==2)  traverse\_postorder(tree->root);  else  printf("type error");  printf("\n");  } |

* 1. **ADT\_list.c (앞의 챕터에서 작성)**

|  |
| --- |
| **ADT\_list.c** |
| #include "ADT\_llist.h"  LLIST\* create\_list(){  LLIST\* list = (LLIST\*)malloc(sizeof(LLIST));  if(!list)  return NULL;  list->count = 0;  list->front = NULL;  list->rear = NULL;  list->pos = NULL;  return list;  }  bool add\_node\_at(LLIST\* list,unsigned int index,void\* in){  if(index>(list->count))  return false;  NODE\* node = (NODE\*)malloc(sizeof(NODE));  if(node == NULL)  return false;  node->data\_ptr = in;  node->next = NULL;    if(list->count == 0){  list->front = node;  list->rear = node;  (list->count)++;  return true;  }    int iter\_i = 0;    if(index == 0){  node -> next = list -> front;  list -> front = node;  (list -> count) ++;  return true;  }    iter\_i++;  list -> pos = list -> front;    while(iter\_i != index){  list -> pos = list -> pos -> next;  iter\_i++;  }    if(iter\_i == list->count){  list -> pos -> next = node;  list -> rear = node;  (list -> count)++;  return true;  }  else{  node -> next = list -> pos -> next;  list -> pos -> next = node;  (list -> count)++;  return true;  }  }  bool del\_node\_at(LLIST\* list,unsigned int index){  if(list -> count == 0)  return false;  else if(index >= list -> count)  return false;  if(list -> count == 1){  free(list -> front);  list -> front =NULL;  list -> rear =NULL;  list -> count =0;  return true;  }  int iter\_i=0;  list->pos = list->front;  NODE\* pre = NULL;  while(iter\_i!=index){  pre=list->pos;  list->pos=list->pos->next;  iter\_i++;  }  if(index==0){  list->front=list->pos->next;  free(list->pos);  list->pos=NULL;  (list->count)--;  }  if(index==(list->count-1)){  list->rear=pre;  pre->next=NULL;  free(list->pos);  list->pos = NULL;  (list->count)--;  return true;  }  else{  pre->next=list->pos->next;  free(list->pos);  list->pos=NULL;  (list->count)--;  return true;  }  return false;  }  void\* get\_data\_at(LLIST\* list,unsigned int index){  if(index>=list->count)  return NULL;  list->pos=list->front;  int iter\_i=0;  while(list->pos!=NULL){  if(iter\_i==index)  return list->pos->data\_ptr;  list->pos=list->pos->next;  iter\_i++;  }  return NULL;  }  int find\_data(LLIST\* list, void\* search\_data){  list->pos = list->front;  int cmp\_result, left, right;  int iter\_i=0;  while(list->pos!=NULL){  left=\*(int\*)(list->pos->data\_ptr);  right=\*(int\*)search\_data;  cmp\_result=left-right;  if(cmp\_result==0)  return iter\_i;    list->pos=list->pos->next;  iter\_i++;  }  return -1;  } |

**2-1. main\_llist.c (과제 공지 파일 참조)**

|  |
| --- |
| **main\_llist.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\_bst = create\_bst\_tree();  for(i=0; i<sample\_NUM;i++)  {  \*M = sample\_NUM-i;  fscanf(fin, "%d", N);  add\_node\_at(new\_llist, 0, M);  N++;  M++;  }  iter = find\_data(new\_llist, &search\_num);  printf("iter num=%d\n",iter);  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. 데이터를 담을 새로운 list인 new\_llist를 생성한다. 10000000부터 1까지를 포인터 M으로 정의하면서 sample.txt의 정수 데이터를 scan하여 N에 저장한다. 이 N을 전에 정의한 new\_llist에 add하고 N과 M을 증가시키는 과정을 반복한다.  반복문이 끝나면 new\_bst에서 9999999의 값(sample.txt의 마지막 데이터)을 가진 list를 찾아 iter에 저장한다. iter의 값을 printf하고 fclose 함수를 호출하여 파일 스트림을 닫는다. |

**2-2. 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\_bst = create\_bst\_tree();  for(i=0;i<sample\_NUM;i++)  {  \*M = sample\_NUM -i;  fscanf(fin,"%d",N);  BST\_insert(new\_bst,\*N);  N++;  M++;  }  search = search\_bst(new\_bst->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\_bst를 생성한다. 10000000부터 1까지를 포인터 M으로 정의하면서 sample.txt의 정수 데이터를 scan하여 N에 저장한다. 이 N을 전에 정의한 new\_bst에 insert하고 N과 M을 증가시키는 과정을 반복한다.  반복문이 끝나면 new\_bst에서 9999999의 값(sample.txt의 마지막 데이터)을 가진 root를 찾아 search에 저장한다. search의 데이터를 printf하고 fclose 함수를 호출하여 파일 스트림을 닫는다. |

**3. 실행 결과**

|  |
| --- |
| profile\_llist.txt |
| EMB0000061062ef |
| profile\_tree.txt |
| EMB0000061062f0 |
| **해석** |
| tree와 llist 각각의 폴더에서 실행하여 나온 profile.txt를 보니 10000000개의 데이터 중 9999999의 데이터를 찾기까지의 시간은 llist는 50.11ms, tree는 20.05ms가 걸려 예상했던 대로 tree가 list보다 검색시간이 빠름을 확인했다. |

**4. 고찰**

이번 과제는 전의 과제와는 다르게 앞서 튜터시간에 헤더파일을 같이 작성해서 진행하는데 큰 어려움이 없었습니다. 다만 처음 과제 파일 코드를 다 작성한 후 디버깅을 할 때, main~.c 부분의 create\_list()함수의 인자부분을 비우고 정의했는데 main에서 인자를 사용하여 오류가 떴었습니다. 그것을 제외하곤 별다른 이상이 없었습니다.

수업시간에 배운 tree와 list들의 개념을 이용하여 실제로 검색방법으로 이용하였을 때 예상대로 tree의 search\_bst가 list의 find\_data보다 빨리 수행하는 결과를 얻어내어 뿌듯했습니다.