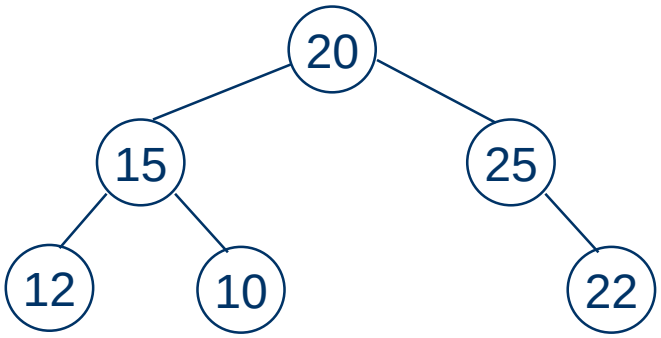
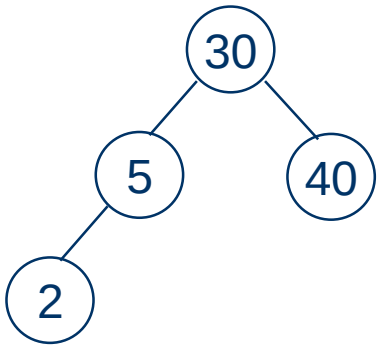
이원 탐색 트리(binary search tree)

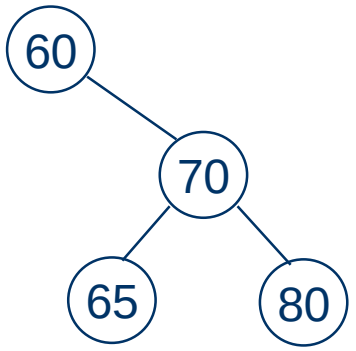
- 정의
  - 이진트리로서 공백가능하고, 만약 공백이 아니라면
  - (1) 모든 원소는 서로 상이한 키를 갖는다.
  - (2) 왼쪽 서브트리의 키들은 그 루트의 키보다 작다.
  - (3) 오른쪽 서브트리의 키들은 그 루트의 키보다 크다
  - (4) 왼쪽과 오른쪽 서브트리도 이원 탐색 트리이다.
    - 즉 이 조건으로 인해 전체 트리의 모든 노드에 대해 (2),(3) 조건이 만족해야 한다.
- 이원 탐색 트리



(a) X



(b) O



(c) O

일반적인 트리 (general tree)

- 임의의 일반 트리
  - 각 노드는 자식의 갯수에 제한이 없는 트리임
  - 이러한 트리는 일반적인 트리로서 실제 많이 나타남

- 다음과 같은 일반 트리를 만들어 본다.
  - 입력: 텍스트 화일

일반 트리의 데이터 구조

```
#define MAX 10
typedef struct ano * nodeptr ;
typedef struct ano {
        char name[30];
        float gpa ;
        char place[30] ;
        nodeptr links[MAX] ; // 맨 마지막 자식 다음에 NULL 을 넣어야 함!!
} nodetype ;                 // int child_cnt ; 라는 필드를 준비하여 자식의 수를 넣는 방식도
                                가능!
```

일반 트리 만들기 실험

- 아래 트리는 다음과 같은 데이타 파일의 읽어 구축한 것이다.

```
인영    상현    진선    미라
상현    서희    주진    은진
진선    주리
미라    명숙    한빛    도영    전민
---
인영  3.13  통영시
상현  4.12  제주시
진선  3.93  원주시
미라  3.34  여주시
서희  3.45  정선시
주진  2.67  상주시
은진  3.14  진주시
주리  4.02  파주시
명숙  2.34  철원시
한빛  3.89  남원시
도영  3.90  수원시
전민  2.67  천안시
```

Project： Handling a general tree

- 목적: 일반트리를 다루는 프로그래밍 능력 배양
  - 제공된 모든 타입 및 변수선언, 함수들 (메인 및 보조)을 사용하여도 좋다.
- 작업:
  - 1) 트리에 넣을 데이타의 텍스트화일 “tree_data.txt“ 을 준비한다.
    - 데이타 화일의 상단부: 트리의 노드 생성 및 링크 연결에 사용한다.
      한 줄의 내용: 사람이 먼저 오고 그 다음에 이 사람의 자식들의 이름이 온다.
    - 데이타 화일의 하단부: 각 사람의 개인 데이타(학점, 지역)를 저장한다.
    - 아래는 단순히 예이므로 변화시켜 사용
    - 데이타 화일의 위치는 프로그램 화일과 동일한 디렉토리이어야 함

```
인영    상현    진선    미라
상현    서희    주진    은진
진선    주리
미라    명숙    한빛    도영    전민
명숙    해미
해미    정아
----
인영 3.13 통영시
상현 4.12 제주시
진선 3.93 원주시
미라 3.34 여주시
서희 3.45 정선시
주진 2.67 상주시
은진 3.14 진주시
주리 4.02 파주시
명숙 2.34 철원시
한빛 3.89 남원시
도영 3.90 수원시
전민 2.67 천안시
해미 3.05 서울시
정아 2.98 시흥시
```

  - 2) 데이타화일을 읽어서 트리를 구축한다.
    - 트리의 루트 노드에 대한 포인터를 전역변수 ROOT 에 저장한다.(root 가 아님. 선언시에 NULL 로 초기화해야 함!.)

- 3) 명령 루프:
  - 아래와 같은 명령들을 실행하여 주는 루프를 작성한다.

    - 명령의 종류:

    > se 진선
    >
    > 진선이 저장된 노드를 탐색하여 이 노드 안의 데이타를 한 줄에 출력한다. 없으면 없다고 출력한다.
    >
    > ht
    >
    > 전체 트리의 높이 (height) 를 출력한다.
    >
    > dp 은진
    >
    > 은진이 존재하는 노드의 깊이(depth) 를 출력한다.
    >
    > ac 명숙
    >
    > 명숙의 조상을 모두 출력한다. (조상이란: 루트 노드에서 이 노드로의 경로 상에 모든 사람들)
    >
    > nm 상현
    >
    > 상현 및 그의 모든 자손들의 수를 출력한다.
    >
    > br 미라
    >
    > 미라의 형제들의 이름을 출력한다.
    >
    > ex
    >
    > 루프를 종료한다.

- 제출 방법:
  - 프로그램 화일 1 개만을 압축하지 않고 업로드한다.
    - 주의: 여러 화일이면 1 개로 만들어 수행 가능하게 하여 제출한다.
    - 절대로 개발환경 전체를 zip 하여 제출하지 말 것.

타입 및 변수 선언

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 10 // maximum number of children of a node

typedef struct ano * nodeptr ;
typedef struct ano {
     char name[30];
     float gpa ;
     char place[100] ;
     nodeptr links[MAX] ;
} nodetype ;

nodeptr ROOT =NULL ;  // Pointer to the root node of the tree
nodeptr stack[100]; // stack used for ancestors list
int top = -1; // stack top
```

DFS 기법을 사용하는 search 함수

```
20  //  cur : 트리의 노드에 대한 포인터.
21  //  sname : 찾을 이름.
22  //  반환값:  찾은 노드의 포인터 ( 못찾은 경우 NULL )
23  nodeptr search_node (nodeptr cur,  char sname [ ] ) {
24        int i ;
25        nodeptr rptr ;
26        if (!cur )
27           return NULL ;
28
29        if ( strcmp(cur->name , sname) == 0 ) {
30              return cur ; // 성공.
31        }
32        else
33        {
34            for (i=0; cur->links[i] != NULL ; i++ )
35            {
36                rptr = search_node ( cur->links[i], sname ) ;
37                if ( rptr )
38                  return rptr ;
39            }
40            return NULL ; // 자식들 모두에서도 못찾았으므로 실패.
41        }
42  }// search_node
```

데이타 화일의 상반부의 한 줄 읽는 함수

```
44 // 한줄을 읽어서 2 차원 배열 buf 에 넣고, 사람수를 반환한다.
45 // ( 0 은  ---  줄을 읽은 경우에 반환된다.)
46 int read_upper ( FILE *fp , char buf[20][50] ) {
47     // read "parent - children" lines to create the tree.
48     int i, j, k;
49     char line[400] ; char *cp ;
50     cp = fgets ( line, 400, fp ) ;
51     if (!cp) {
52         // fail in reading .
53         printf ( "Impossible for the control to reach here.₩n" ) ;
54         return 0 ;
55     }
56     else { //
57         if (line[0] == '-') // 첫 영역에 대한 읽기 종료.(수정한 것임)
58             return 0 ;
59 
60         i=0; // number of names read so far.
61         j=0 ; // index to line.
62         while ( 1 ) { // i loop
63             k=0 ; // index to a character of the i-th name.(수정한 것임)
64             while ( line[j] != ' ' && line[j] != '₩n' ) { // j loop.
65                 buf[i][k] = line[j] ;
66                 j++; k++ ;
67             }
68             buf[i][k] = '₩0' ; // finish one name.
69             i++ ; // increase the name count.
70             if ( line[j] == '₩n' )
71                 break ; // exit the i loop.
72             else {
73                 // advance j till it reach a  non-blank character.
74                 do j++; while ( line[j] == ' ' ) ;
75             }
76         }
77         return i ;
78     }
79 }// read_upper
```

데이타 화일의 하단부을 읽어 정보를 저장하는 함수들

```
void read_store_lower_data ( FILE *fp )
{
    char rname[30], address[100] ;
    float rgpa ;
    nodeptr np ;
    int read_num ;

    do  {
        read_num = fscanf (fp, "%s%f%s", rname, &rgpa, address ) ;
        if ( read_num != 3 )
            break ;
        else        {   // 읽은 이름을 가진 노드를 트리에서 찾는다.
            np = search_node ( ROOT, rname ) ;
            if (!np)
                printf ( "이런 이름을 가진 노드가 없습니다.₩n"  ) ;
            else
                // 읽은 데이타를 저장한다.
                strcpy ( np->name, rname ) ; np->gpa = rgpa ; strcpy(np->place, address) ;
        }
    }   while ( 1 ) ;
}
```

DFS 방식으로 모든 노드의 내용을 출력하는 함수

```
115 void print_general_tree ( nodeptr curr )
116 {   int i ;
117     if (!curr)
118         return ;
119     // 이 노드의 내용을 먼저 출력한다.
120     printf ( "이름:%s  학점:%5.2f  주소:%s ₩n", curr->name, curr->gpa, curr->place ) ;
121     for ( i=0; curr->links[i] ; i++ )
122         print_general_tree ( curr->links[i] ) ;
123
124 }
```

main 함수 : 데이타 화일을 읽어 트리를 구축.

```
void main ( ) {
    char buf [20][50] ;
    int num_persons, k ;
    nodeptr np ;

    FILE *fp ;
    fp = fopen ("data_general_tree.txt", "r") ;
    if (!fp) {
        printf ("file open error.₩n"); return ;
    }

    do {
        num_persons = read_upper ( fp, buf ) ; // read one line of several names
        if ( num_persons == 0 ) // '-'  in first letter of the first person's name
            break; // a line of ------ separating the file
        if ( !root ) {// no node in the tree.
            np = (nodeptr) malloc ( sizeof(nodetype) ) ;
            strcpy( np-> name, buf[0]) ; //부모 이름을 넣음.
            root = np ;
            np->links[0] = NULL ;
            for ( k = 1; k < num_persons ; k++ )  // 자식들을 만들어 매달음.
                make_node_and_attach_to_parent ( np, buf[k], k-1 ) ;
            // put NULL in links field after the last child.
            np->links[k-1] = NULL ; // k-1 인지 확인요.
        }
        else { // ordinary case of a tree with one or more nodes
            np = search_node ( root, buf[0] ) ;
            if (!np)
            {
                printf ( "Error: 2째 줄 이하의 첫 이름의 노드가 없음.₩n"  ) ;
                getchar() ;
            }
            for ( k = 1; k < num_persons ; k++ )
                make_node_and_attach_to_parent ( np, buf[k], k-1 ) ;
            // put NULL in links field after the last child.
            np->links[k-1] = NULL ; // k-1 인지 확인요.
        }
    } while ( 1 ) ;

    // 각 노드에 대한 데이타를 읽어서 저장한다.
    read_store_lower_data ( fp ) ;

    // 트리 안의 모든 노드의 데이타를 preorder 방식으로 출력한다.
    print_general_tree ( root ) ;
```

```
// parent 의 loc 위치에 자식노드를 만들어 매달고 이름은 tname 을 넣어 준다.
void make_node_and_attach_to_parent ( nodeptr parent, char *tname, int loc )
{   nodeptr np1 ;
    np1 = (nodeptr) malloc ( sizeof(nodetype) ) ;
    strcpy( np1-> name , tname ) ;
    np1->links[0] =NULL ; // 자식이 없는 걸로 함.
    parent->links[loc] = np1 ; // 부모에 매단다.
}
```

ROOT

main 함수 (계속): 명령 수행 루프

```
    // This is the command execution loop.
    do {
        printf("Type a command> ");
        gets(line);

        i = 0;
        while (line[i] == ' ' || line[i] == '₩t') i++; // move to 1st letter of the command.
        k = 0;
        while ( !(line[i] == ' ' || line[i] == '₩t' || line[i] == '₩0')  ) {
            command[k] = line[i]; i++; k++;  // Bring letters of the command
        }
        command[k] = '₩0'; // finish the command

        if (strcmp(command, "ex") == 0)
            break;
        else if (strcmp(command, "ht") == 0){
            int tree_height = 0;
            dfs_height (ROOT, 0, &tree_height );
            printf("Height of the tree : %d ₩n", tree_height);
            continue;
        }
        else;

        // Read a name given after the command.
        k = 0;
        while (line[i] == ' ' || line[i] == '₩t') i++; // move to 1st letter of the command.
        while (!(line[i] == line[i] == ' ' || line[i] == '₩t' || line[i] == '₩0')) {
            name[k] = line[i]; i++; k++;  // Bring letters of the command
        }
        name [k] = '₩0'; // finish the command

        if (strcmp(command, "se") == 0){
            np = search_node(ROOT, name);
            if (np)
                printf("Seach success:  %s %5.2f  %s₩n", np->name, np->gpa, np->place);
            else printf("Such a person does not exist.₩n");
        }
        else if (strcmp(command, "dp") == 0) {
            res = dfs_depth(ROOT, 0, name);
            if (!res)
                printf("Such a name does not exist in the tree.₩n");
        }
        else if (strcmp(command, "ac") == 0) {
            top = -1; // stack is empties at first.
            res = dfs_ancestors(ROOT, name);
            if (!res)
                printf("Such a name does not exist in the tree.₩n");
        }
        else {
            printf("₩rong command.₩n");
        }
    } while (1);

}// main.
```

```c
// Depth first search for finding the height of the whole tree.
void dfs_height (nodeptr cur , int par_height, int *tree_height) {
    int my_height = par_height + 1; // height of this node
    int i;
    if (*tree_height < my_height)
        *tree_height = my_height; // change the tree height
    for (i = 0; cur->links[i]; i++)
        dfs_height(cur->links[i], my_height, tree_height);
    return;
}

// Depth first search for finding the depth of a specific person.
//   return value: 1 if the name has been found and its depth was printed.
//                 0 if the name was not found in the subtree with root of cur.
int dfs_depth (nodeptr cur, int par_height, char *sname ) {
    int my_height = par_height + 1; // height of this node
    int i, res ;
    if (strcmp(cur->name, sname) == 0) {
        printf("Height of the node of %s : %d\n", sname, my_height);
        return 1;
    }
    for (i = 0; cur->links[i]; i++) {
        res = dfs_depth (cur->links[i], my_height, sname );
        if (res)
            return 1;  // stop dfs search and return to its caller.
    }
    return 0;  // report 0 since the name was not found in this subtree.
}
```

```c
void push_stack(nodeptr nptr){
    top++;
    stack[top] = nptr;
}

void pop_stack( ) { // this pop does not return the element.
    top--;
}

int dfs_ancestors  (nodeptr cur , char *sname) {
    int i, res ;
    if ( strcmp(cur->name, sname) == 0 ) {
        printf("This person's ancestors: ");
        for (i = 0; i <= top; i++)
            printf("%s  ", stack[i]->name) ;
        printf("₩n");
        return 1;
    }

    push_stack  (cur);
    for (i = 0; cur->links[i]; i++) {
        res = dfs_ancestors  (cur->links[i], sname) ;
        if (res)
            return 1;  // stop dfs search and return to its caller.
    }

    pop_stack( ) ; // cur was pushed before. We remove it before returning to its parent.
    return 0;  // report 0 since the name was not found in this subtree.
}
```