사전평가문제.

1. (아스키코드를 배우지 않았음)

2. Tree 는 stack 이 쌓일 때 마다 2 의 배수로 쌓일 수 있는 이점이 있다. 그리하여 자료를 찾음에 있어 Stack 과 Queue 보다 매우 빠른 속도를 지닌다.

4. int p[7] : 7 개의 변수를 가진 배열을 선언
   (*p)[7] : 7 개의 주소에 접근

7. main 함수
9. 메모리구조 : 레지스트 → 캐시 → 메모리 → 디스크 (오른쪽으로 갈 수록 용량증가, 왼쪽으로 갈 수록 속도증가)

10. User 가 사용하는 메모리 공간에는 위에서부터 stack, heap, data, text 가 존재. Stack : 지역변수존재, Heap : 동적할당, Data : 전역변수, static 존재, Text : 기계어존재

11. 파이프라인은 새로운 함수가 들어올 경우 한 칸씩 밀리는 현상에 의하여 깨짐.

19. 자료첨부(스택)
20. while
21. 자료첨부(이진트리)
22. 검색속도 자체만 보았을 경우에는, Rb tree 가 월등히 빠르나 입력과 삭제시에는 회전을 하는 경우가 많아 자료가 들어오고 삭제될 경우, 더 오래걸린다.
23. 자료첨부(AVL 트리)

88. 빛은 파동의 성질과 입자의 성질을 동시에 가지고 있는데, 파동의 성질이 전자기파와 같다.

첨부.
19
```
#include <stdio.h>
#include <malloc.h>

#define EMPTY 0

typedef struct node
{
   int data;
   struct node *link;
}Stack;

Stack *get_node()
{
   Stack *tmp;
   tmp = (Stack *)malloc(sizeof(Stack));
   tmp -> link = EMPTY;
   return tmp;
}

void push(Stack **top, int data)
```

```
{
   Stack *tmp;
   tmp = *top;
   *top = get_node();
   (*top) -> data = data;
   (*top) -> link = tmp;
}

int pop(Stack **top)
{
   Stack *tmp;
   tmp = *top;
   int num;
   if(*top == EMPTY)
   {
      printf("Stack is EMPTY!!!\n");
      return 0;
   }
   num = tmp -> data;
   *top = (*top) -> link;
   free(tmp);
   return num;
}

int main(void)
{
   Stack *top = EMPTY;
   push(&top, 10);
   push(&top, 20);
   push(&top, 30);
   printf("%d\n", pop(&top));
   printf("%d\n", pop(&top));
   printf("%d\n", pop(&top));
   printf("%d\n", pop(&top));

   return 0;
}
```

21.

```
#include <stdio.h>
#include <malloc.h>

#define EMPTY 0

typedef struct node
{
   int data;
   struct node *left;
   struct node *right;
}tree;

tree *get_node()
```

```
{
  tree *tmp;
  tmp = (tree *)malloc(sizeof(tree));
  tmp -> left = EMPTY;
  tmp -> right = EMPTY;
  return tmp;
}

void tree_ins(tree **root, int data)
{
  if(*root == NULL)
  {
     *root = get_node();
     (*root) -> data = data;
  }
  else if(data < (*root) -> data)
    tree_ins(&(*root) -> left, data);
  else if(data > (*root) -> data)
    tree_ins(&(*root) -> right, data);
}

int print_tree(tree *root)
{
  if(root)
  {
     printf("%d\n", root -> data);
     print_tree(root -> left);
     print_tree(root -> right);
  }
  return 0;
}

tree *chg_node(tree *root)
{
   tree *tmp;
   tmp = root;
   if(root -> left)
   root = root -> left;
   else if(!root -> right)
   root = root -> right;

   free(tmp);

   return root;
}

tree *find_max(tree *root, int *data)
{
   if(root -> right)
     root -> right = find_max(root -> right, data);
   else
   {
```

```
      *data = root -> data;
      root = chg_node(root);
    }
    return root;
}

tree *detree(tree *root, int data)
{
    int num;
    if(root == NULL)
    {
        printf("NOT FOUND\n");
        return NULL;
    }
    else if(data < root -> data)
     root -> left = detree(root -> left, data);
    else if(data > root -> data)
     root -> right = detree(root -> right, data);
    else if(root -> left && root -> right)
    {
     root -> left = find_max(root -> left, &num);
     root -> data = num;
    }
    else
      root = chg_node(root);

    return root;
}


int main(void)
{
    tree *root = EMPTY;
    int i;
    int arr[13] = {50, 45, 73, 32, 48, 46, 16, 37, 120, 47, 130, 127, 124};
    for(i = 0; i < 13; i++)
    {
        tree_ins(&root, arr[i]);
    }
    print_tree(root);

    root = detree(root, 48);

    print_tree(root);

    return 0;
}
```

23.

```
#include <stdio.h>
#include <malloc.h>
```

```
#define EMPTY 0

typedef struct __avl
{
   int data;
   int level;
   struct __avl *left;
   struct __avl *right;
}avl;

avl* get_node()
{
   avl *tmp;
   tmp = (avl *)malloc(sizeof(avl));
   tmp -> left = EMPTY;
   tmp -> right = EMPTY;
   tmp -> level = 1;
   return tmp;
}

int update_level(tree *root)
{
   int left = root -> left ? root -> left -> level:0;
   int right = root -> right ? root -> right -> level:0;

   if(left > right)
   {
      return left + 1;
   }
   return right + 1;
}

int rotation_check(tree *root)
{
   int left = root -> left ? root -> left -> level :0;
   int right = root -> right ? root -> right -> level :0;

   return right - left;
}

void avl_ins(avl **root, int data)
{
   if(*root == EMPTY)
   {
      *root = get_node();
      (*root) -> data = data;
   }
   else if(data < (*root) -> data)
    avl_ins(&(*root) -> left, data);
   else if(data > (*root) -> data)
    avl_ins(&(*root) -> right, data);
```

```
    (*root) -> level = update_level(*root);

    if(abs(rotation_check(*root)) > 1)

}
```