Chapter 11.
탐색(Search) 1

5주차

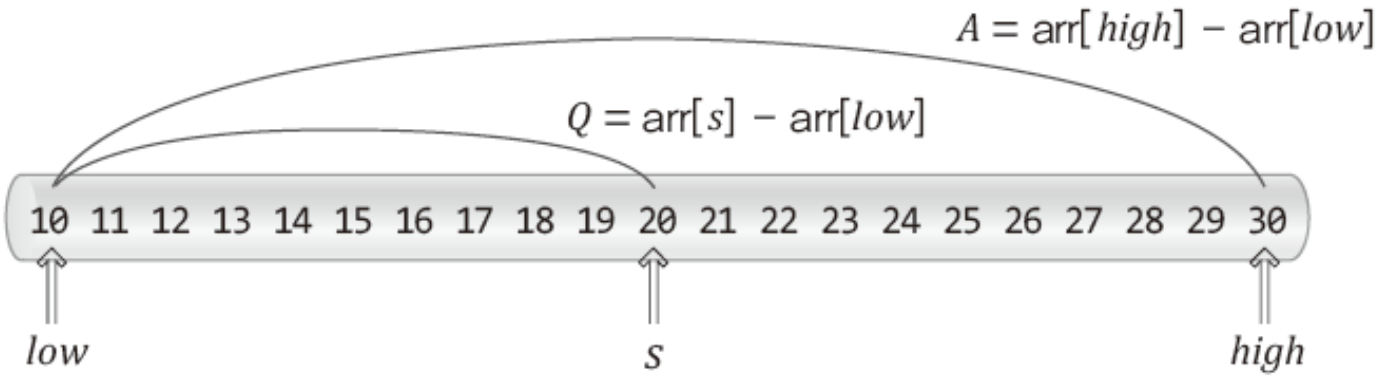
탐색의 이해

- 효율적인 탐색을 위해서는
  - '어떻게 찾을까' 보다는
  - **'어떻게 저장할까'** 를 고민해야 함.
  - 효율적인 탐색이 가능한 대표적인 자료구조는 **트리**

보간 탐색

- 이진 탐색 — 무조건 중간에 위치한 데이터를 탐색
- 보간 탐색 — 대상에 비례하여 탐색의 위치를 결정

보간 탐색
(비례식 구성)



s : 탐색 대상이 저장된 인덱스 값
low, high : 시작과 끝 인덱스 값

$$A : Q = (high-low) : (s-low)$$

비례식 구성

$$s = \frac{Q}{A}(high-low)+low$$

$$s = \frac{x-arr[low]}{arr[high]-arr[low]}(high-low)+low$$

탐색 위치의 인덱스 값 계산식

탐색 키와 탐색 데이터의 구분

- 실제 프로그램에 있어서 탐색의 대상은 '데이터' 가 아닌 '키(key)'
- 학습의 편의를 위해서, 데이터가 곧 키 인 형태로 간단히 예제를 작성하고 있을 뿐!
- 실제 활용 시에는 아래와 같은 형태일 것.

```
typedef Key int;
typedef Data double;

typedef struct item {
    Key key;          // 탐색에 사용.
    Data data;        // 탐색 후에 얻게 됨.
}
```

탐색 main 함수


```
#include <stdio.h>
int BSearch(int ar[], int first, int last, int target);
int ISearch(int ar[], int first, int last, int target);
int main(void) {
	int arr[] = { 1, 3, 5, 7, 9 };
	int idx;
	idx = BSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 7);
	if (idx == -1)
		printf("7의 탐색 실패 ₩n");
	else
		printf("7의 저장 인덱스: %d ₩n", idx);

	idx = BSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 2);
	if (idx == -1)
		printf("2의 탐색 실패 ₩n");
	else
		printf("2의 저장 인덱스: %d ₩n", idx);
	return 0;
}
```

428p

이진 탐색 함수 (복습)

```
int BSearch(int ar[], int first, int last, int target) {
    int mid;
    printf("first=%d last=%d target=%d₩n", first, last, target);

    if (first > last)
        return -1;    // -1의 반환은 탐색의 실패를 의미

    mid = (first+last) / 2;

    if (ar[mid] == target)
        return mid;    // 탐색된 타겟의 인덱스 값 반환
    else if (target < ar[mid])
        return ISearch(ar, first, mid - 1, target);
    else
        return ISearch(ar, mid + 1, last, target);
}
```

실행횟수와 인덱스값 확인.

인덱스에 의해서 탐색 중지 결정.

인덱스에 의해서만 탐색 위치 정함.

탐색 main 함수 수정

```
int main(void) {
        int arr[] = { 1, 3, 5, 7, 9 };
        int idx;
        idx = BSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 7);
        if (idx == -1)
                printf("7의 탐색 실패 ₩n");
        else
                printf("7의 저장 인덱스: %d ₩n", idx);

        idx = BSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 2);
        if (idx == -1)
                printf("2의 탐색 실패 ₩n");
        else
                printf("2의 저장 인덱스: %d ₩n", idx);
        idx = lSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 7);
        if (idx == -1)
                printf("7의 탐색 실패 ₩n");
        else
                printf("7의 저장 인덱스: %d ₩n", idx);
        idx = lSearch(arr, 0, sizeof(arr) / sizeof(int) - 1, 2);
        if (idx == -1)
                printf("2의 탐색 실패 ₩n");
        else
                printf("2의 저장 인덱스: %d ₩n", idx);
        return 0;
}
```

보간 탐색의 구현

429-432p

- 실행횟수와 인덱스값 확인.
- 탈출조건을 이진탐색과 같이 하면 안되는 이유 확인.

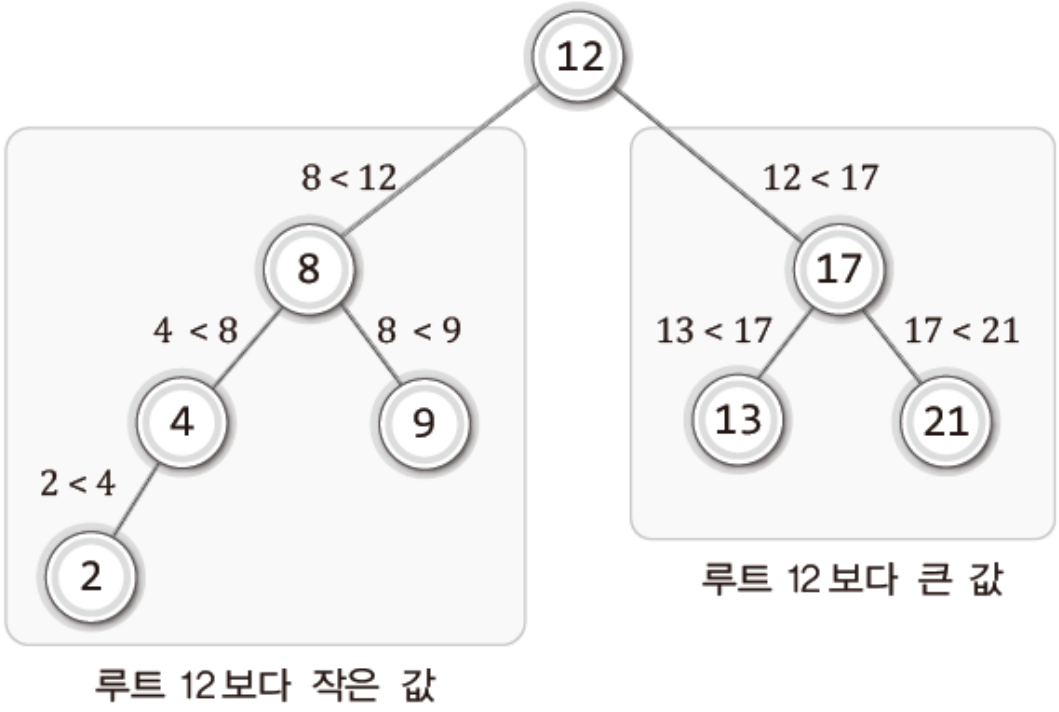
```
int ISearch(int ar[], int first, int last, int target) {
    int mid;
    printf("first=%d last=%d target=%d₩n", first, last, target);

    if (ar[first] > target || ar[last] < target)
        return -1;

    mid = ((double)(target - ar[first]) /
        (ar[last] - ar[first]) * (last - first)) + first;

    if (ar[mid] == target)
        return mid;  // 탐색된 타겟의 인덱스 값 반환
    else if (target < ar[mid])
        return ISearch(ar, first, mid - 1, target);
    else
        return ISearch(ar, mid + 1, last, target);
}
```

인덱스와 값에 의해서 탐색 중지 결정.

값과 인덱스를 고려하여 탐색.

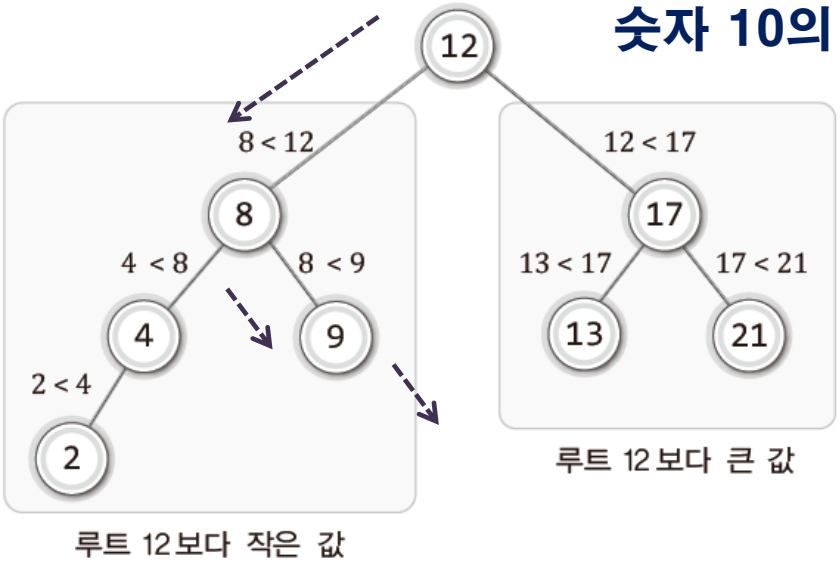
이진 탐색 트리
(이해)

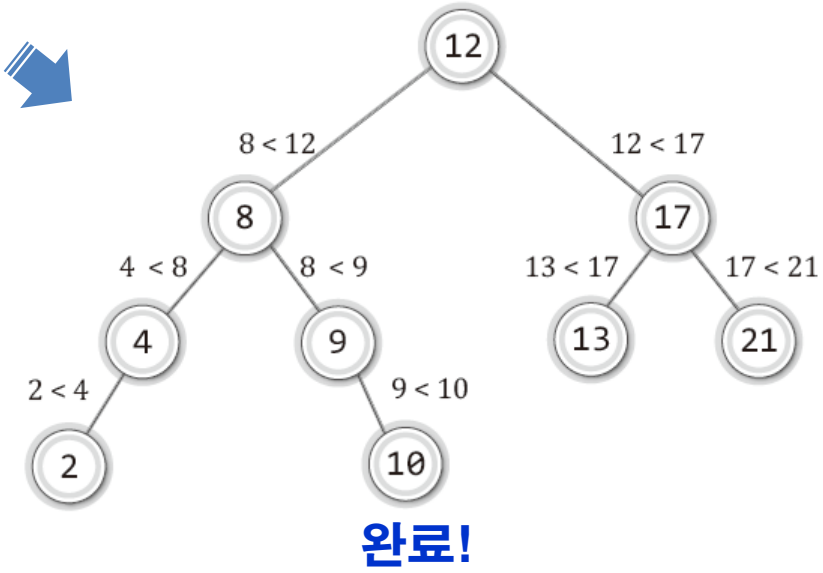
◆ 이진 탐색 트리 : 특별한 저장 규칙을 가지는 이진 트리
- 이진 탐색 트리의 노드에 저장된 키(key)는 유일!
- 루트 노드의 키 > 왼쪽 서브 트리를 구성하는 키
- 루트 노드의 키 < 오른쪽 서브 트리를 구성하는 키
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리!



이진 탐색 트리
(노드 추가 과정)

숫자 10의 저장을 진행!





완료!

(1) 10의 위치를 탐색 :

현재 노드보다 작으면 왼쪽 자식으로,

크면 오른쪽 자식으로

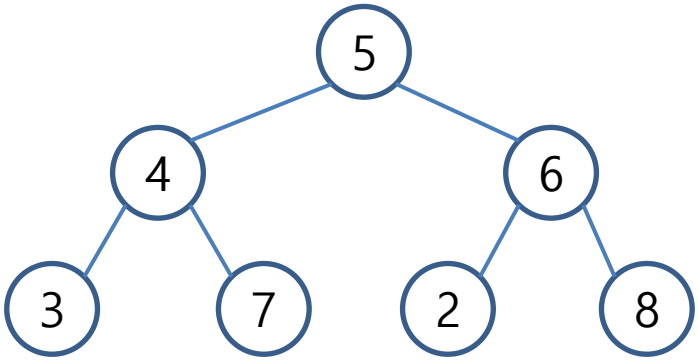
현재 위치를 옮김.

(2) 탐색이 끝난 자리에 삽입

이진 탐색 트리
(잘못된 조건)

- 맞는 조건
  - 루트 노드의 키 > 왼쪽 서브 트리를 구성하는 키
  - 루트 노드의 키 < 오른쪽 서브 트리를 구성하는 키
- 틀린 조건
  - 부모 노드의 키 > 왼쪽 자식의 키
  - 부모 노드의 키 < 오른쪽 자식의 키

- "틀린 조건" 을 만족하는 예 (이진 탐색 트리가 아님!)



436-444p

이진탐색 트리의 구현
(이진 트리와 이진 탐색 트리를 위한 선언)

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

typedef int BTData;
typedef BTData BSTData;

typedef struct _bTreeNode
{
	BTData data;
	struct _bTreeNode* left;
	struct _bTreeNode* right;
} BTreeNode;
```

이진탐색 트리의 구현
(이진 트리의 구현)

```
BTreeNode* MakeBTreeNode(void)
{
	BTreeNode* nd = (BTreeNode*)malloc(sizeof(BTreeNode));

	nd->left = NULL;
	nd->right = NULL;
	return nd;
}

BTData GetData(BTreeNode* bt)
{
	return bt->data;
}

void SetData(BTreeNode* bt, BTData data)
{
	bt->data = data;
}
```

436-444p

이진탐색 트리의 구현
(이진 트리의 구현)

```
BTreeNode* GetLeftSubTree(BTreeNode* bt) {
	return bt->left;
}
BTreeNode* GetRightSubTree(BTreeNode* bt) {
	return bt->right;
}

void MakeLeftSubTree(BTreeNode* main, BTreeNode* sub) {
	if (main->left != NULL)
		free(main->left);
	main->left = sub;
}

void MakeRightSubTree(BTreeNode* main, BTreeNode* sub) {
	if (main->right != NULL)
		free(main->right);
	main->right = sub;
}
```

이진탐색 트리의 구현

```
void BSTMakeAndInit(BTreeNode** pRoot)
{
	*pRoot = NULL;
}

BSTData BSTGetNodeData(BTreeNode* bst)
{
	return GetData(bst);
}
```

436-444p

이진탐색 트리의 구현

```
void BSTInsert(BTreeNode** pRoot, BSTData data) {
        BTreeNode* pNode = NULL;    // parent node
        BTreeNode* cNode = *pRoot;    // current node
        BTreeNode* nNode = NULL;    // new node

        while (cNode != NULL)        {
                if (data == GetData(cNode))
                        return;
                pNode = cNode;
                if (GetData(cNode) > data)
                        cNode = GetLeftSubTree(cNode);
                else
                        cNode = GetRightSubTree(cNode);
        }

        .........
```

새로운 노드가 추가될 위치 찾기.

키의 중복을 허용하지 않음

436-444p

이진탐색 트리의 구현

```
void BSTInsert(BTreeNode** pRoot, BSTData data) {
        .........
        nNode = MakeBTreeNode();
        SetData(nNode, data);

    if (pNode != NULL) {
            if (data < GetData(pNode))
                    MakeLeftSubTree(pNode, nNode);
            else
                    MakeRightSubTree(pNode, nNode);
    }
    else
    {
            *pRoot = nNode;
    }
}
```

pNode의 자식으로 추가할 새 노드 생성

pNode의 서브 노드에 새 노드를 추가

빈 트리가 아닐 때.

빈 트리일 때.
새 노드가 루트가 됨.

이진탐색 트리의 구현

```
BTreeNode* BSTSearch(BTreeNode* bst, BSTData target) {
	BTreeNode* cNode = bst;	// current node
	BSTData cd;	// current data

	while (cNode != NULL)		{
		cd = GetData(cNode);

		if (target == cd)
			return cNode;
		else if (target < cd)
			cNode = GetLeftSubTree(cNode);
		else
			cNode = GetRightSubTree(cNode);
	}

	return NULL;
}
```

436-444p

이진탐색 트리의 구현

```
int main(void) {
    BTreeNode* bstRoot;
    BTreeNode* sNode; // search node
    BSTMakeAndInit(&bstRoot);
    BSTInsert(&bstRoot, 9); BSTInsert(&bstRoot, 1);
    BSTInsert(&bstRoot, 6); BSTInsert(&bstRoot, 2);
    BSTInsert(&bstRoot, 8); //BSTInsert(&bstRoot, 4);
    BSTInsert(&bstRoot, 3); BSTInsert(&bstRoot, 5);
    //BSTInsert(&bstRoot, 7);
    sNode = BSTSearch(bstRoot, 1);
    printf("1 탐색: %s %d₩n", sNode ? "성공" : "실패",
        sNode ? BSTGetNodeData(sNode) : -1);
    sNode = BSTSearch(bstRoot, 4);
    printf("4 탐색: %s %d₩n", sNode ? "성공" : "실패",
        sNode ? BSTGetNodeData(sNode) : -1);
    sNode = BSTSearch(bstRoot, 6);
    printf("6 탐색: %s %d₩n", sNode ? "성공" : "실패",
        sNode ? BSTGetNodeData(sNode) : -1);
    sNode = BSTSearch(bstRoot, 7);
    printf("7 탐색: %s %d₩n", sNode ? "성공" : "실패",
        sNode ? BSTGetNodeData(sNode) : -1);
    return 0;
```