# 게임 자료구조와 알고리즘
## -CHAPTER10-

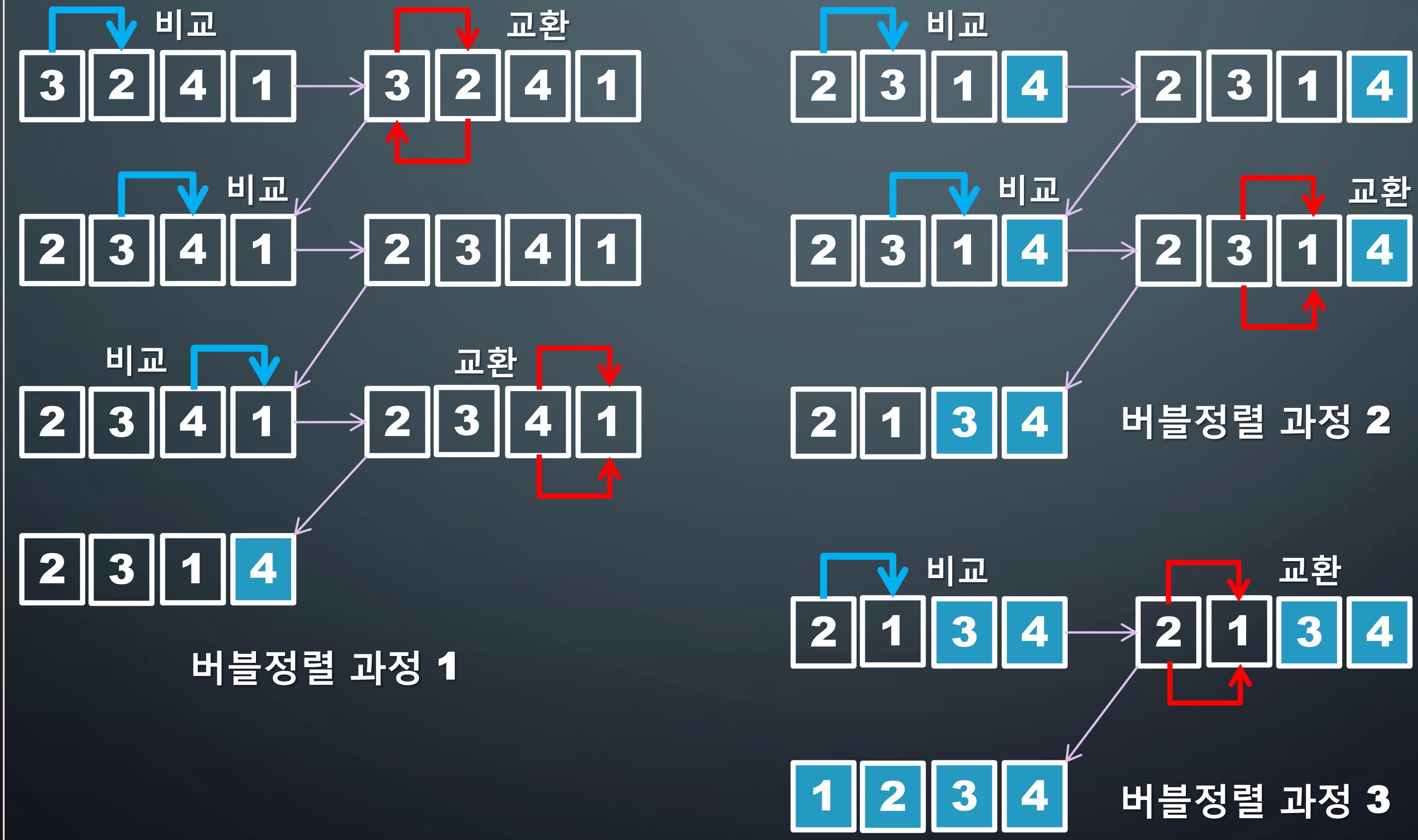
SOULSEEK

# 목차

# 버블정렬(BUBBLE SORT)

# 1. 버블정렬(BUBBLE SORT)

- 인접한 두 개의 데이터를 비교해가면서 정렬을 진행하는 방식
- 정렬 순서상 위치가 바뀌어야 하는 경우에 두 데이터의 위치를 바꿔 나간다.
- 비교를 반복하면서 자리를 이동하는 모양이 거품 같다고 해서 버블정렬




버블정렬 과정 1

버블정렬 과정 2

버블정렬 과정 3

# 1. 버블정렬(BUBBLE SORT)

```
#include <stdio.h>

void BubbleSort(int arr[], int n)
{
	int temp;

	for (int i = 0; i < n - 1; i++)
	{
		for (int j = 0; j < (n - 1) - i; j++)
		{
			if (arr[j] > arr[j + 1])
			{
				temp = arr[j];
				arr[j] = arr[j + 1];
				arr[j + 1] = temp;
			}
		}
	}
}
```

```
int main(void)
{
	int arr[4] = { 3, 2, 4, 1 };

	BubbleSort(arr, sizeof(arr) / sizeof(int));

	for (int i = 0; i < 4; i++)
		printf("%d ", arr[i]);

	printf("\n");

	return 0;
}
```

# 1. 버블정렬(BUBBLE SORT)

**성능평가**

- 비교연산과 대입연산을 체크한다.
- 비교 횟수 : 두 데이터간의 비교연산 횟수
- 이동의 횟수 : 위치의 변경을 위한 데이터의 이동 횟수

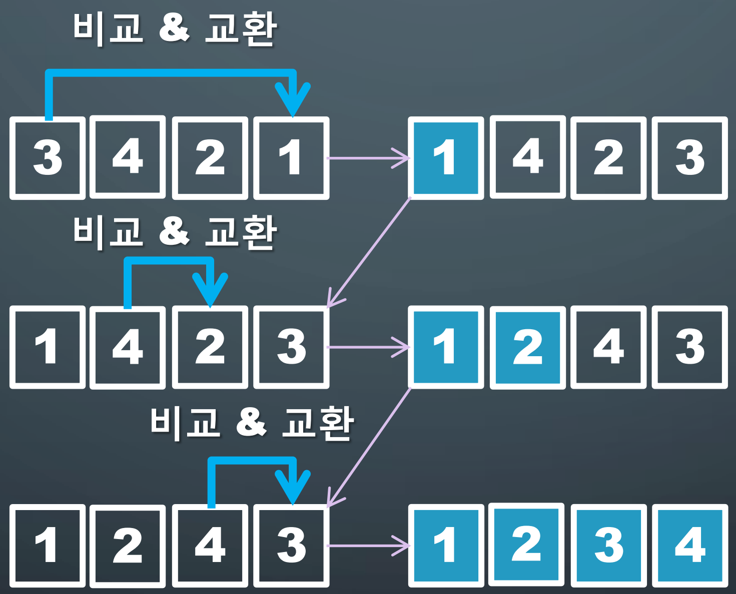
```
for(int I = 0; I < n – 1; i++)
{
	for(int j = 0; j < (n - 1) – 1; j++)
	{
		if(arr[j] > arr[j + 1])
		{
			// 비교연산이 발생하는 장소
			// 이동연산이 발생하는 장소
		}
	}
}
```

- 반복 할 수록 수가 점점 작아진다.
- **(n - 1) + (n - 2) + … + 2 + 1 ➔** 등차수열의 합에 해당한다. **➔** $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$
- **O(**$n^2$**) –** 비교 횟수**,** 이동 횟수 모두 최악의 경우에는 이보다 **3**배 가량 많은 시간을 소모하지만 빅 **-** 오에서는 제외

# 선택정렬(SELECTION SORT)

# 2. 선택정렬(SELECTION SORT)

- 정렬 순서상 가장 앞서는 것을 선택해서 가장 왼쪽으로 이동시키고, 원래 그 자리에 있던 데이터는 빈 자리에 가져다 놓는다.
- 임의의 빈자리를 마련해 놓아야 한다.



# 2. 선택정렬(SELECTION SORT)

```c
#include <stdio.h>

void SelSort(int arr[], int n)
{
	int maxIdx;
	int temp;

	for (int i = 0; i < n - 1; i++)
	{
		maxIdx = i;

		for (int j = i + 1; j < n; j++)
		{
			if (arr[j] < arr[maxIdx])
				maxIdx = j;
		}

		temp = arr[i];
		arr[i] = arr[maxIdx];
		arr[maxIdx] = temp;
	}
}
```

```c
int main(void)
{
	int arr[4] = { 3, 4, 2, 1 };

	SelSort(arr, sizeof(arr) / sizeof(int));

	for (int i = 0; i < 4; i++)
		printf("%d ", arr[i]);

	printf("\n");
	return 0;
}
```

# 2. 선택정렬(SELECTION SORT)

**성능평가**

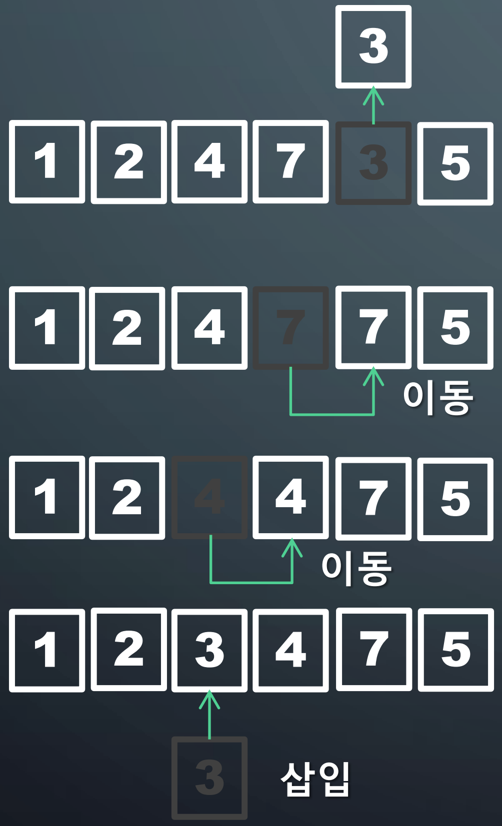
```
for (int i = 0; i < n - 1; i++)
{
	maxIdx = i;

	for (int j = i + 1; j < n; j++)
	{
		if (arr[j] < arr[maxIdx])
		{
			// 비교연산이 발생하는 장소
		}

	}

	// 이동연산이 발생하는 장소
}
```

- 비교횟수는 버블 정렬과 똑같다. – **O(**$n^2$**)**
- 바깥쪽 **for**문 안에 포함 되어 있어서 최대배열 크기보다 **1**개 적은 **n – 1**회를 하게 된다. – **O(n)**

# 삽입정렬(INSERTION SORT)

# 3. 삽입정렬(INSERTION SORT)

- 첫 번째와 두 번째를 비교한 후 그 다음 데이터의 위치가 속할 곳을 찾아서 데이터를 밀어내고 삽입되는 정렬 방식이다.
- 정렬이 완료된 영역의 다음에 위치한 데이터가 그 다음 정렬 대상이다.
- 삽입할 위치를 발견하고 데이터를 한 칸씩 밀수도 있지만, 데이터를 한 칸씩 밀면서 삽입할 위치를 찾을 수도 있다.




3의 자리를 찾자!

3과 7 비교 후, 7을 한 칸 뒤로 이동

3과 4 비교 후, 4를 한 칸 뒤로 이동

3과 2 비교 후, 3을 삽입!

# 3. 삽입정렬(INSERTION SORT)

```
#include <stdio.h>

void InserSort(int arr[], int n)
{
	int i, j;
	int insData;

	for (i = 1; i < n; i++)
	{
		insData = arr[i];

		for (j = i - 1; j >= 0; j--)
		{
			if (arr[j] > insData)
				arr[j + 1] = arr[j];
			else
				break;
		}

		arr[j + 1] = insData;
	}
}
```

```
int main(void)
{
	int arr[5] = { 5, 3, 2, 4, 1 };
	int i;

	InserSort(arr, sizeof(arr) / sizeof(int));

	for (i = 0; i < 5; i++)
		printf("%d ", arr[i]);

	printf("\n");
	return 0;
}
```

# 3. 삽입정렬(INSERTION SORT)

**성능평가**

```
for (i = 1; i < n; i++)
{
    insData = arr[i];   // 정렬 대상을 insData에 저장

    for (j = i - 1; j >= 0; j--)
    {
        if (arr[j] > insData)         // 비교 연산
            arr[j + 1] = arr[j];      // 이동 연산
        else
            break;
    }
    ……
}
```

- 비교 연산과 이동 연산 모두 이중 반복문 안쪽에 있다 – **O(**$n^2$**)**

# 힙 정렬(HEAP SORT)

# 4. 힙 정렬(HEAP SORT)

- 힙의 특성을 살린 알고리즘
  - ex) 힙의 루트 노드에 저장된 값이 가장 커야 한다. (최대 힙)

  == 힙의 루트 노드에 저장된 값이 정렬순서상 가장 앞선다.
  - 이를 토대로 UsefulHeep를 활용한 정렬을 만들어 보자.

```
int PriComp(int n1, int n2)
{
	return n2 - n1; // 오름차순의 정렬
	//return n1 - n2;
}

void HeapSort(int arr[], int n, PriorityComp pc)
{
	Heap heap;

	HeapInit(&heap, pc);

	for (int i = 0; i < n; i++)
		HInsert(&heap, arr[i]);

	for (int i = 0; i < n; i++)
		arr[i] = HDelete(&heap);
}
```

```
int main()
{
	int arr[4] = {5, 8, 3, 1};

	HeapSort(arr, sizeof(arr) / sizeof(int), PriComp);

	for (int i = 0; i < 4; i++)
		printf("%d ", arr[i]);

	printf("\n");
	return 0;
}
```

# 4. 힙 정렬(HEAP SORT)

```
void HeapSort(int arr[], int n, PriorityComp pc)
{
	Heap heap;

	HeapInit(&heap, pc);

	//힙 정렬 단계1: 데이터를 모두 힙에 넣는다.
	for (int i = 0; i < n; i++)
		HInsert(&heap, arr[i]);

	//힙 정렬 단계2: 힙에서 다시 데이터를 꺼낸다.
	for (int i = 0; i < n; i++)
		arr[i] = HDelete(&heap);
}
```

- 데이터를 넣었다가 꺼내는 행동을 할 뿐이지만 힙의 특성상 우선순위에 의해서 오름차순, 내림차순 같은 순서 정렬이 가능한 것이다.

# 4. 힙 정렬(HEAP SORT)

## 성능평가

- 삽입과 삭제에 대한 시간 복잡도는 모두 **O**($\log_2 n$)이다 모두 합해서 **O**($2\log_2 n$)이지만 빅 – 오에서는 고려 사항이 아니므로 **O**($\log_2 n$)이 된다.
- **O**($\log_2 n$)이 정렬과정에서 정렬대상의 데이터 개수 만큼 삽입 삭제를 해야 하므로 **O**($\text{n}\log_2 n$)이 된다.

**O($nlog_2 n$)과 O($n^2$)은 차이가 많이 난다.**

| n | 10 | 100 | 1,000 | 3,000 | 5,000 |
|---|---|---|---|---|---|
| $n^2$ | 100 | 10,000 | 1,000,000 | 9,000,000 | 25,000,000 |
| $\text{n}\log_2 n$ | 66 | 664 | 19,931 | 34,652 | 61,438 |

# 병합 정렬(MERGE SORT)

# 5. 병합 정렬(MERGE SORT)

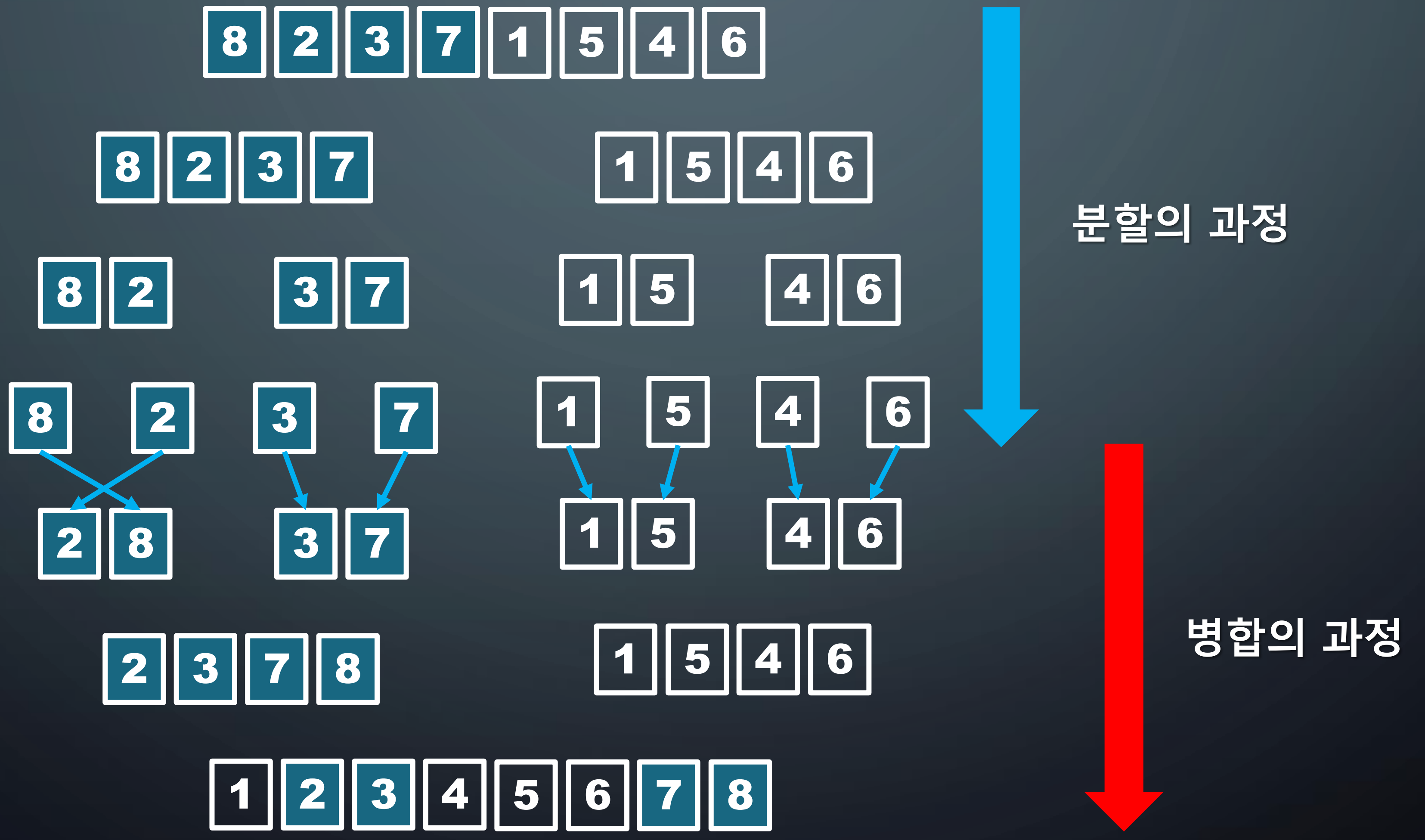
- 분할 정복**(divide and conquer)**이라는 알고리즘 디자인 기법에 근거하여 만들어진 정렬 방법이다.

  - 분할 정복 **-** 분할하여 정렬을 한 후 결합과정을 거친다**, 3**단계의 과정을 거쳐서 완성된다**.**
    - **1**단계 **:** 분할**(Divide) –** 해결이 용이한 단계까지 문제를 분할해 나간다**.**
    - **2**단계 **:** 정복**(Conquer) –** 해결이 용이한 수준까지 분할된 문제를 해결한다**.**
    - **3**단계 **:** 결합**(Combine) –** 분할해서 해결한 결과를 결합하여 마무리한다**.**

- **ex) 8**개의 데이터를 동시에 정렬하는 것보다**,** 이를 둘로 나눠서 **4**개의 데이터를 정렬하는 것이 쉽고**,** 또 이들 각각을 다시 한번 둘로 나눠서 **2**개의 데이터를 정렬하는 것이 더 쉽다**.**

# 5. 병합 정렬(MERGE SORT)

- 둘로 나누는 것보다 1개의 데이터가 남을 때까지 나눠서 정렬을 하고 **다시 합치는 과정이 더 중요**하다.
- 나눠질 수 없을 만큼 나눴다가 **정렬을 한 후 병합을 진행**한다.
- 나누는 과정의 순서대로 다시 병합을 하기 때문에 **나누는 과정과 합치는 과정의 횟수는 같다**.
- **재귀적 구현을 위한 방법**이다.



# 5. 병합 정렬(MERGE SORT)

```c
void MergeTwoArea(int arr[], int left, int mid, int right)
{
	int fIdx = left;
	int rIdx = mid + 1;
	int i;

	int * sortArr = (int*)malloc(sizeof(int)*(right + 1));
	int sIdx = left;

	while (fIdx <= mid && rIdx <= right)
	{
		if (arr[fIdx] <= arr[rIdx])
		sortArr[sIdx] = arr[fIdx++];
		else
		sortArr[sIdx] = arr[rIdx++];

		sIdx++;
	}

	if (fIdx > mid)
	{
		for (i = rIdx; i <= right; i++, sIdx++)
			sortArr[sIdx] = arr[i];
	}
	else
	{
		for (i = fIdx; i <= mid; i++, sIdx++)
			sortArr[sIdx] = arr[i];
	}

	for (i = left; i <= right; i++)
		arr[i] = sortArr[i];

	free(sortArr);
}
```

```c
void MergeSort(int arr[], int left, int right)
{
	int mid;

	if (left < right)
	{
		// 중간 지점을 계산한다.
		mid = (left + right) / 2;

		// 둘로 나눠서 각각을 정렬한다.
		MergeSort(arr, left, mid);
		MergeSort(arr, mid + 1, right);

		// 정렬된 두 배열을 병합한다.
		MergeTwoArea(arr, left, mid, right);
	}
}

int main(void)
{
	int arr[7] = { 3, 2, 4, 1, 7, 6, 5 };
	int i;

	// 배열 arr의 전체 영역 정렬
	MergeSort(arr, 0, sizeof(arr) / sizeof(int) - 1);

	for (i = 0; i < 7; i++)
		printf("%d ", arr[i]);

	printf("\n");
	return 0;
}
```

# 5. 병합 정렬(MERGE SORT)

- 병합 정렬을 진행하는 함수

**void MergeSort(int arr[], int left, int right);**

- 첫 번째 인자로 정렬대상이 담긴 배열의 주소 값을 전달하고**,** 두 번째 인자와 세 번째 인자로 정렬대상의 범위정보를 인덱스 값의 형태로 전달한다**.**
- **ex)** 정렬대상이 배열 전체라면 배열의 첫 번째 와 마지막의 인덱스 값을 두 번째 인자와 세 번째 인자로 각각 전달한다**.**

```
void MergeSort(int arr[], int left, int right)
{
	int mid;

	if (left < right)
	{
		// 중간 지점을 계산한다.
		mid = (left + right) / 2;

		// 둘로 나눠서 각각을 정렬한다.
		MergeSort(arr, left, mid);
		MergeSort(arr, mid + 1, right);

		// 정렬된 두 배열을 병합한다.
		MergeTwoArea(arr, left, mid, right);
	}
}
```

# 5. 병합 정렬(MERGE SORT)

**void MergeTwoArea(arr, left, mid, right);**

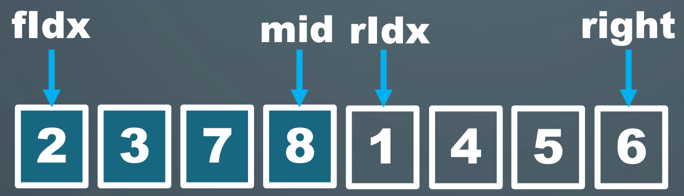
- 배열 **arr**의 **left ~ mid**까지, 그리고 **mid + 1 ~ right**까지 각각 정렬이 되어 있으니, 이를 하나의 정렬된 상태로 묶어서 배열 **arr**에 저장해라!

```
void MergeTwoArea(int arr[], int left, int mid, int right)
{
        // 병합 한 결과를 담을 배열 sortArr의 동적 할당!
        int * sortArr = (int*)malloc(sizeof(int)*(right+1));

        while(fIdx<=mid && rIdx<=right)
        {
                // 병합 할 두 영역의 데이터들을 비교하여, 정렬 순서대로 sortArr에 하나씩 옮겨 담는다.
        }

        if(fIdx > mid)      // 배열의 앞부분이 모두 sortArr에 옮겨졌다면,
        {
                // 배열의 뒷부분에 남은 데이터들을 sortArr에 그대로 옮긴다.
        }
        else  // 배열의 뒷부분이 모두 sortArr에 옮겨졌다면,
        {
                // 배열의 앞부분에 남은 데이터들을 sortArr에 그대로 옮긴다.
        }

        for(i=left; i<=right; i++)
        arr[i] = sortArr[i];

        free(sortArr);
}
```

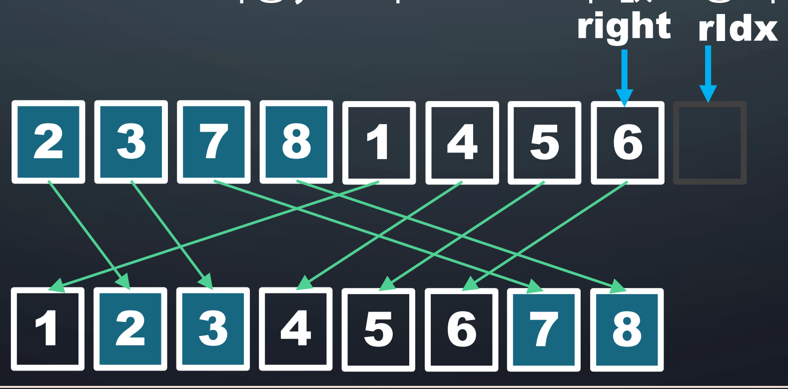
# 5. 병합 정렬(MERGE SORT)

```
while(fIdx <= mid && rIdx <= right)
{
}
```

- **fIdx**와 **rIdx**에는 각각 병합할 두 영역의 첫 번째 위치정보(인덱스 값)가 담긴다.
- **fIdx**와 **rIdx**의 값을 증가시키면서 두 영역의 데이터를 비교해 나가게 된다.
- **mid**의 값을 중심으로 좌우를 나눌 값이 담겨져 있다.




- **2**와 **1** 비교연산 후 **1**을 **sortArr**로 이동, 그리고 **rIdx**의 값 **1**증가
- **2**와 **4** 비교연산 후 **2**를 **sortArr**로 이동, 그리고 **fIdx**의 값 **1**증가
- **3**과 **4** 비교연산 후 **3**을 **sortArr**로 이동, 그리고 **fIdx**의 값 **1**증가
- **7**과 **4** 비교연산 후 **4**를 **sortArr**로 이동, 그리고 **rIdx**의 값 **1**증가
- **7**과 **5** 비교연산 후 **5**를 **sortArr**로 이동, 그리고 **rIdx**의 값 **1**증가
- **7**과 **6** 비교연산 후 **6**을 **srotArr**로 이동, 그리고 **rIdx**의 값 **1**증가 후 **right**를 넘어서 **while** 탈출



# 5. 병합 정렬(MERGE SORT)

- **While** 탈출 후 아직 이동 하지 못한 데이터를 처리한다**.**

**If(fIdx > mid)** // 배열의 앞 부분이 모두 **sortArr**에 옮겨졌다면**,**
**{**
  // 배열의 뒷부분에 남은 데이터들을 **sortArr**에 그대로 옮긴다**.**
**}**
**else** // 배열의 뒷부분이 모두 **sortArr**에 옮겨졌다면**,**
**{**
  // 배열의 앞부분에 남은 데이터들을 **sortArr**에 그대로 옮긴다**.**
**}**

# 5. 병합 정렬(MERGE SORT)

## 성능평가

- 실제 정렬을 진행하는 **MergeTwoArea** 함수를 중심으로 진행되기 때문에 비교, 이동연산을 계산하는 주체가 된다.

## 비교연산

```
while (fIdx <= mid && rIdx <= right)
{
	if (arr[fIdx] <= arr[rIdx])		// 핵심이 되는 비교연산
		sortArr[sIdx] = arr[fIdx++];
	else
		sortArr[sIdx] = arr[rIdx++];

	sIdx++;
}
```

- 8개의 원소를 한 개가 남을 때까지 나눈 뒤 다시 합치는 과정에서 비교연산은 최대 2회 진행한다.
- 두 개씩 모인 원소가 다시 4개가 될 때 최대 비교연산은 4회가 된다.
- 8개일 때 병합 3회, 16회 일 때 4회… $K = \log_2 n$ ➔ $O(n\log_2 n)$
- 임시메모리가 필요하다는 단점이 있지만 배열을 연결 리스트로 바꾸면 임시메모리의 단점이 사라진다.

# 5. 병합 정렬(MERGE SORT)

**이동연산**

- **while(fIdx <= mid & rIdx <= right) –** 배열 **sortArr**에 데이터를 정렬하며 이동**!**
- **if(fIdx > mid){} ~ else{} –** 각각 배열 **sortArr**에 나머지 데이터를 이동**!**
- **for(i = left; i <= right; i++) arr[i] = sortArr[i] –** 임시 배열에 저장된 데이터 전부를 이동**!**

즉**,** 임시배열에 데이터를 병합하는 과정과 저장된 데이터 전부를 원위치로 옮기는 과정에서 각각 한번씩 일어 나는 것이다**.**

결론은 이동연산 역시 **O($nlog_2 n$)** 이 된다**.**

# 퀵 정렬(QUICK SORT)

# 6. 퀵 정렬(QUICK SORT)

- **Pivot**이라는 중심 값을 기준으로 두 자료의 키 값을 비교하여 위치를 교환하는 정렬 방식
- **Pivot**의 위치 교환이 끝난 다음, 기존 자료 집합을 **Pivot**을 기준으로 **2**개의 부분 집합으로 나누고, 분할된 부분 집합에 대해 다시 퀵 정렬을 실행하는 방식으로 진행된다.

**{80, 50, 70, 10, 60, 20, 40, 30}**의 **8**개의 정수 자료가 주어졌다고 했을 때 오름차순으로 정렬을 한다고 가정하면..

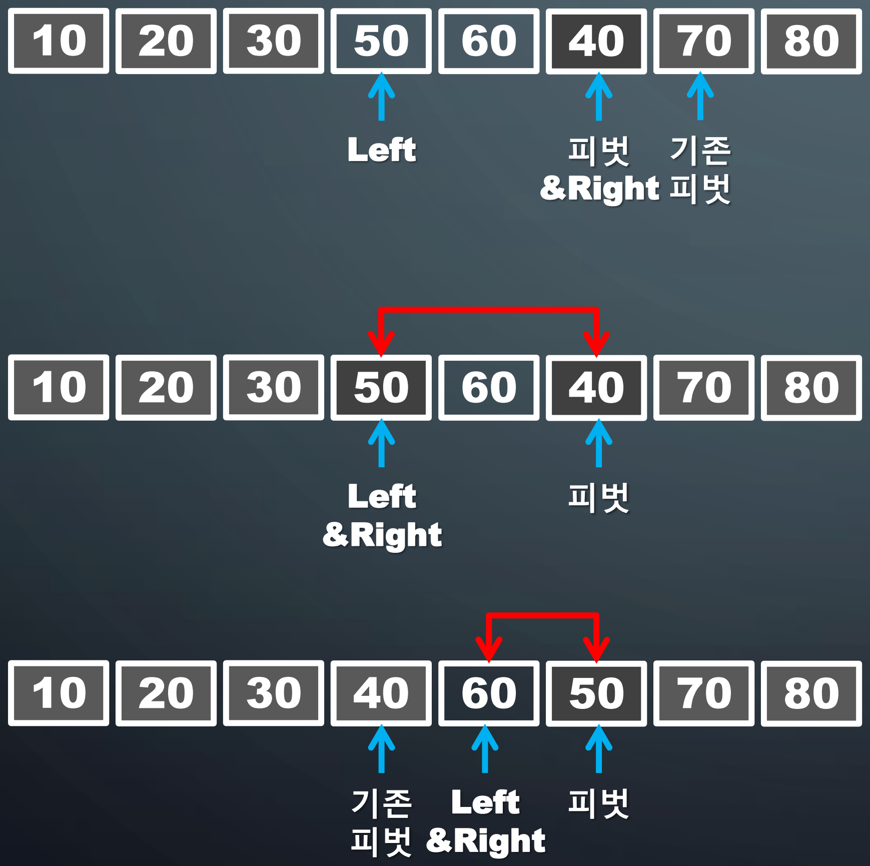
**단계1.** 오른쪽에 **30**을 **Pivot**으로 설정하고 **Pivot**과 양쪽에서 이동하면서 비교해줄 **Right, Left**를 준비한다. **Left**는 왼쪽 끝에서 이동하면서 **30**보다 큰 값을 찾으면 그곳에서 멈추고 **Right**는 오른쪽 끝에서 이동하면서 **30**보다 작은 값을 찾는다.

# 6. 퀵 정렬(QUICK SORT)

**단계2.** 교환된 **30**을 기준으로 왼쪽과 오른쪽으로 나누어 정렬 여부를 판단하고 정렬을 진행한다. 이때 각 나누어진 부분에 새롭게 **Pivot**을 설정해서 진행한다. 왼쪽에 있는 **20**과 **10** 그리고 오른쪽에 있는 **50, 60, 80, 40, 70**을 정렬을 진행한다.

# 6. 퀵 정렬(QUICK SORT)

**단계3.** 남은 정렬이 되지 않은 **3**개 자료 **50, 60, 70**을 대상으로 퀵 정렬을 수행한다.





피봇을 기준으로 두 개 부분 집합으로 나누어 자료의 위치를 교환하기 때문에 **n**개의 자료를 평균 **O(**$n\log_2 n$**)**번 만에 정렬하는 효율성을 가진다**.** 즉**, n**개의 자료가 균형되게 분포된 경우면 정렬 횟수가 모두 **n**번의 비교가 필요하기 때문에 평균 비교횟수는 $n\log_2 n$이 된다**.**

# 탐색의 이해와 보간 탐색

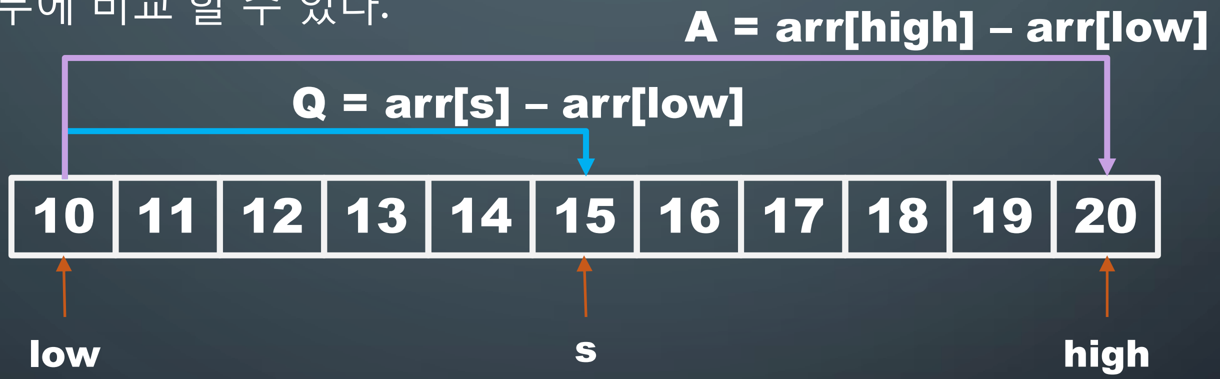
# 7. 탐색의 이해와 보간 탐색

## 탐색의 이해

- 효율적인 탐색을 위해서 검색방법을 고민하는 것보다 효율적으로 저장을 하면 탐색도 효율적이게 된다.
- 자료구조에서 탐색은 중요한 위치를 차지하고 있다.

## 보간탐색

- 이진 탐색과 비슷하지만 탐색 시작위치에서 차이가 있다.
- 탐색하고자 하는 인덱스의 위치에 최대한 가까운 곳부터 검색을 해서 탐색대상을 줄이는 방법이다.
- 지정된 대상과 찾을 인덱스의 위치가 가까울 수록 시간이 줄어들고 최악의 경우라도 일정 이상의 인덱스 탐색대상이 줄어들게 된다.
- 사전과 전화번호부에 비교 할 수 있다.




- 데이터 값과 그 데이터가 저장된 위치의 인덱스 값이 비례한다고 가정하기 때문에
  - $A : Q = (high - low) : (s - low)$
- **S**를 찾는 식으로 표현하면 - $s = \frac{Q}{A}(high - low) + low$
- **Arr[s]**(찾으려는 위치)를 **x**라 하면 - $s = \frac{x - arr[low]}{arr[high] - arr[low]}(high - low) + low$
- 오차율을 줄이기위한 실수형 **나눗셈**을 진행한다는 사실이 보간 탐색의 단점이다.

# 7. 탐색의 이해와 보간 탐색

## 탐색 키와 탐색 데이터

- 탐색을 할 때 대상의 인덱스가 중요하지 해당 데이터를 직접적으로 탐색을 할 때 사용하지 않는다. – 탐색을 진행하는 동안 탐색위치를 이동하는 것에 대한 이야기.
- 탐색키는 고유해야 한다. – 찾아야 할 대상이 무엇인지 정확 판단해야 한다.

```
typedef int Key;              // 탐색 키에 대한 typedef 선언
typedef double Data;          // 탐색 데이터에 대한 typedef 선언

typedef struct item
{
      Key searchKey;          // 탐색 키
      Data searchData;        // 탐색 데이터
}
```

# 7. 탐색의 이해와 보간 탐색

**보간탐색의 구현**

- 이진탐색에서 탐색대상의 선정에 있기 때문에 그 부분만 수정해서 구현 할 수 있다

```
int BSearchRecur(int ar[], int first, int last, int target)
{
	int mid;
	if (first > last)
		return -1;			// -1의 반환은 탐색의 실패를 의미

	mid = (first + last) / 2;	// 탐색대상을 찾기 위한 장소 찾는다.

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

# 7. 탐색의 이해와 보간 탐색

- 탐색대상을 지정하는 방법과 공식이 다르기 때문에 대상을 지정하는 부분은 바꿔 준다.

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

**if (first > last) return -1;** 의 조건으로 탈출 조건을 했을 경우에는 탐색 실패 시,

```
if (ar[mid] == target)
    return mid;
else if (target < ar[mid])
    return BSearchRecur(ar, first, mid - 1, target);
else
    return BSearchRecur(ar, mid + 1, last, target);
```

부분이 다시 한번 실행한다.

- 탈출해야 할 상황에서 탈출하지 못하고 동일한 인자 값을 다시 호출하게 되기때문에 수정해줘야 한다.

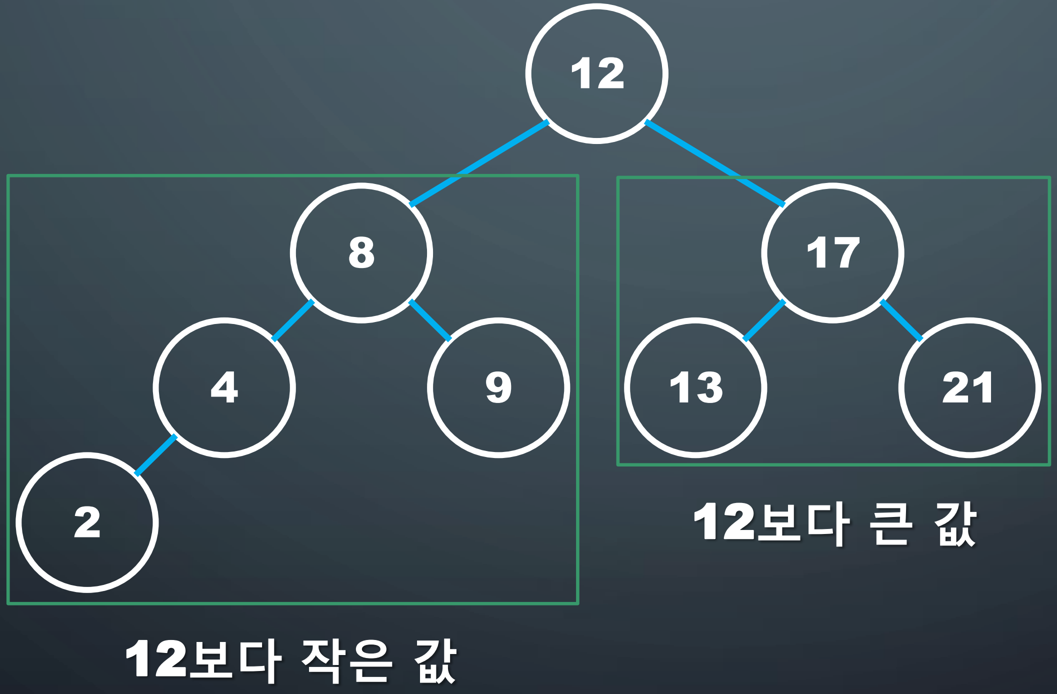
```
if(ar[first] > target || ar[last] < target) return -1;
```

# 이진 탐색 트리

# 8. 이진 탐색 트리

**이진 탐색 트리의 특성** – 이진 트리의 특성도 포함되어 있다.

- 노드에 저장된 키는 유일하다
- 루트 노드의 키가 왼쪽 서브 트리를 구성하는 어떠한 노드의 키보다 크다
- 루트 노드의 키가 오른쪽 서브 트리를 구성하는 어떠한 노드의 키보다 작다.
- 왼쪽과 오른쪽 서버 트리도 이진 탐색 트리이다.
- 서브 노드 안에서 부모 자식관계에서도 규칙이 적용 되어야 한다.




**왼쪽 자식 노드의 키 < 부모 노드의 키 < 오른쪽 자식 노드의 키**

# 8. 이진 탐색 트리

**이진 탐색 트리의 구현**

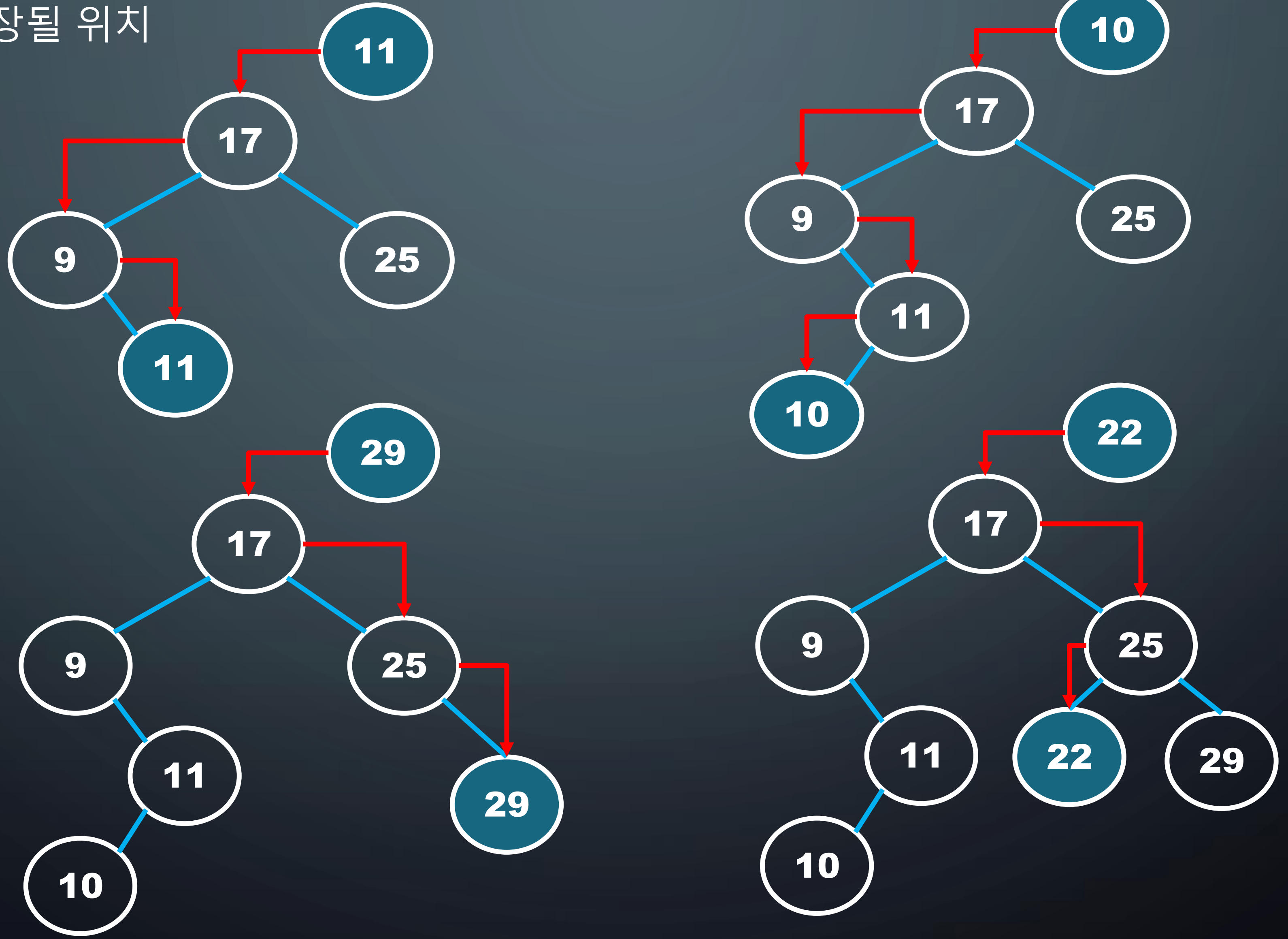
- 이진 트리의 확장을 이용한 이진 탐색 트리
- 이진 탐색 트리를 구현 하면서 이진 트리를 확장함으로써 리팩토링을 해 볼 수 있다.

```
#include "BinaryTree.h"

typedef BTDataBSTData;

// BST의 생성 및 초기화
void BSTMakeAndInit(BTreeNode ** pRoot);

// 노드에 저장된 데이터 반환
BSTData BSTGetNodeData(BTreeNode * bst);

// BST를 대상으로 데이터 저장(노드의 생성과정 포함)
void BSTInsert(BTreeNode ** pRoot, BSTData data);

// BST를 대상으로 데이터 탐색
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target);
```

# 8. 이진 탐색 트리

**삽입과 탐색**

- 비교대상이 없을 때까지 내려간다. 그리고 비교 대상이 없는 그 위치가 새 데이터가 저장될 위치



# 8. 이진 탐색 트리

## 삽입함수를 구현해 보면..

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

        // 새로운 노드가 추가될 위치를 찾는다. - 저장위치 찾기
        while (cNode != NULL)
        {
                if (data == GetData(cNode))
                        return;    // 키의 중복을 허용하지 않음

                pNode = cNode;

                if (GetData(cNode) > data)
                        cNode = GetLeftSubTree(cNode);
                else
                        cNode = GetRightSubTree(cNode);
        }

        // pNode의 서브 노드에 추가할 새 노드의 생성
        nNode = MakeBTreeNode();    // 새 노드의 생성
        SetData(nNode, data);    // 새 노드에 데이터 저장

        // pNode의 서브 노드에 새 노드를 추가
        if (pNode != NULL)    // 새 노드가 루트 노드가 아니라면,
        {
                if (data < GetData(pNode))
                        MakeLeftSubTree(pNode, nNode);
                else
                        MakeRightSubTree(pNode, nNode);
        }
        else    // 새 노드가 루트 노드라면,
        {
                *pRoot = nNode;
        }
}
```

While 문을 빠져나오면 cNode에 새 노드가 저장될 위치정보가 담긴다. 하지만 이를 위해서는 **저장 위치를 자식으로 하는 부모 노드의 주소 값이 필요**하다. 그래서 이어지는 if ~ else 구문에서 부모 노드의 주소 값이 담긴 pNode를 기반으로 자식 노드를 추가하고 있는 것이다.

# 8. 이진 탐색 트리

**탐색 함수를 구현해보면..**

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

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

		if (target == cd)
			return cNode;
		else if (target < cd)		// 비교대상의 노드보다 값이 작으면 왼쪽 자식 노드
			cNode = GetLeftSubTree(cNode);
		else				// 비교대상의 노드보다 값이 크면 오른쪽 자식 노드
			cNode = GetRightSubTree(cNode);
	}

	return NULL; // 탐색대상이 저장되어 있지 않음. – while 탈출 조건
}
```

# 8. 이진 탐색 트리

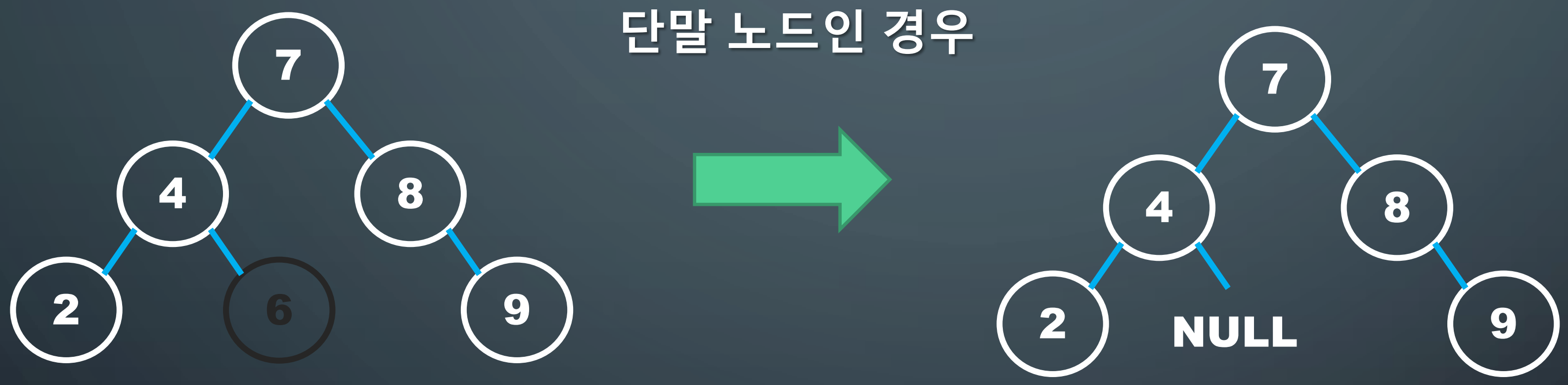
**학습과제**

**제공된 BinarySearchTree.h, BinarySearchTree.cpp를 이용해 수 입력 받고 탐색 결과를 보여주는 BinarySearchTreeMain.cpp를 만들어보자.**

# 8. 이진 탐색 트리

## 삭제

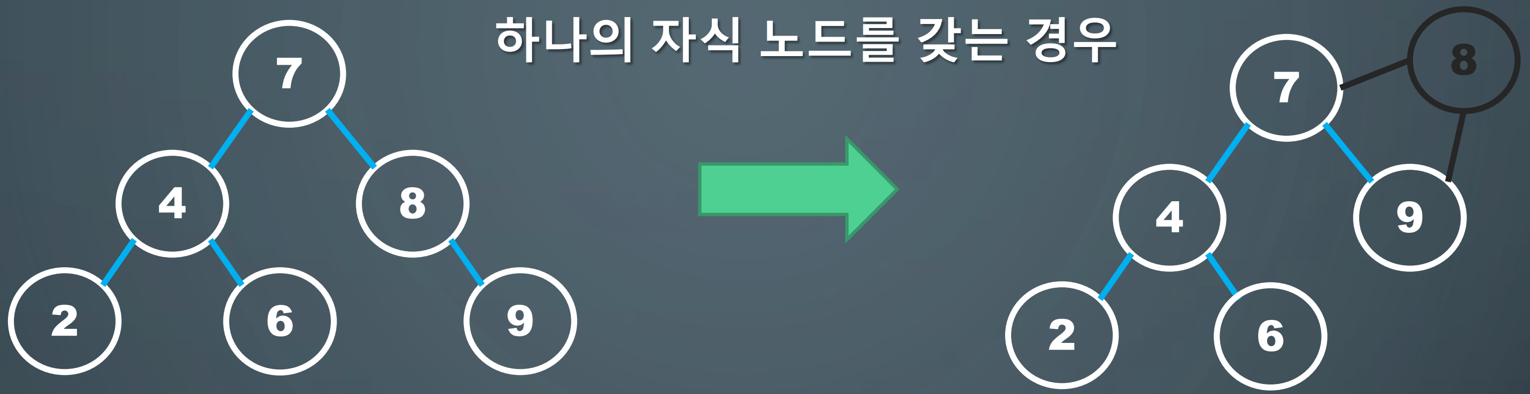
1. 삭제할 노드가 단말 노드인 경우
2. 삭제할 노드가 하나의 자식 노드를 갖는 경우
3. **삭제할 노드가 두 개의 자식 노드를 갖는 경우**
    - 임의의 노드를 삭제할 경우, 삭제 후 이진 탐색 트리가 유지되도록 채워 줘야 한다.




```
if(삭제할 노드가 단말노드이다!)
{
    if(GetLeftSubTree(pNode) == dNode) // 삭제할 노드가 왼쪽 자식이라면,
        RemoveLeftSubTree(pNode);      // pNode가 가리키는 왼쪽 자식 노드를 트리에서 제거
    else                               // 삭제할 노드가 오른쪽 자식이라면,
        RemoveRightSubTree(pNode);     // pNode가 가리키는 오른쪽 자식 노드를 트리에서 제거
}
```

# 8. 이진 탐색 트리

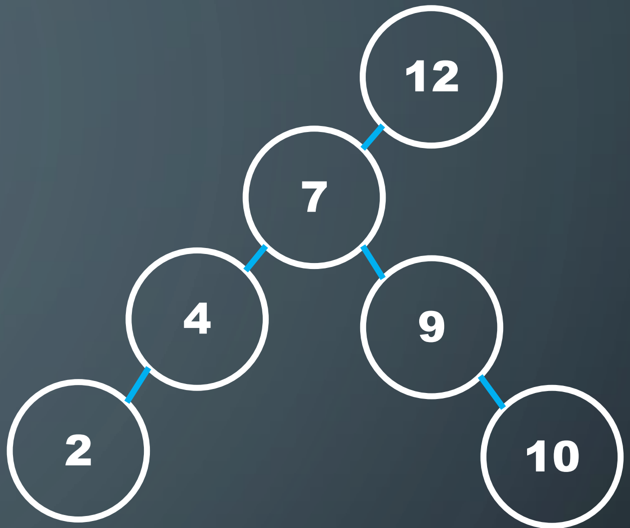
하나의 자식 노드를 갖는 경우



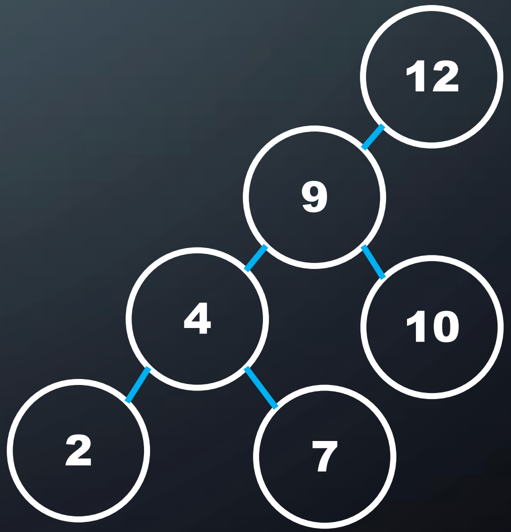

```
if(삭제할 노드가 하나의 자식 노드를 지닌다!)
{
    BTreeNode* dcNode;                         // 삭제 대상의 자식 노드를 가리키는 포인터 변수

    if(GetLeftSubTree(dNode) != NULL)          // 자식 노드가 왼쪽에 있다면,
        dcNode = GetLeftSubTree(dNode);
    else                                       // 자식 노드가 오른쪽에 있다면,
        dcNode = GetRightSubTree(dNode);

    //삭제 대상의 부모 노드와 자식 노드를 연결한다.
    if(GetLeftSubTree(pNode) == dNode)         //삭제 대상이 왼쪽 자식 노드이면,
        ChangeLeftSubTree(pNode, dcNode);      // 왼쪽으로 연결
    else                                       //삭제 대상이 오른쪽 자식 노드이면,
        ChangeRightSubTree(pNode, dcNode);     // 오른쪽으로 연결
}
```

# 8. 이진 탐색 트리

두 개의 자식 노드를 갖는 경우


8이 저장된 노드의 왼쪽 서브 트리에서 가장 큰 값인 7을 저장한 노드




8이 저장된 노드의 오른쪽 서브 트리에서 가장 작은 값인 9를 저장한 노드

# 8. 이진 탐색 트리

- 삭제할 노드의 오른쪽 서브 트리에서 가장 작은 값을 지니는 노드를 찾아서 이것으로 삭제할 노드를 대체한다.

단계1. 삭제할 노드를 대체할 노드를 찾는다.
단계2. 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
단계3. 대체할 노드의 부모 노드와 자식 노드를 연결한다.

```
If(삭제할 노드가 두개의 자식 노드를 지닌다.)
{
	//mNode는 대체 노드를 가리킴
	BTreeNode* mNode = GetRightSubTree(dNode);
	//mpNode는 대체 노드의 부모 노드를 가리킴
	BTreeNode* mpNode = dNode

	//단계1.
	while(GetLeftSubTree(mNode) != NULL)
	{
		mpNode = mNode;
		mNode = GetLeftSubTree(mNode);
	}

	//단계2.
	setData(dNode, GetData(mNode));
```

```
//단계3.
	//대체할 노드가 왼쪽 자식이라면
	if(GetLeftSubTree(mpNode) == mNode)
	{
		//대체할 노드의 자식 노드를 부모 노드의 왼쪽에..
		ChangeLeftSubTree(mpNode,
					GetRightSubTree(mNode));
	}
	else // 대체할 노드가 오른쪽 자식 노드라면
	{
		// 대체할 노드의 자식 노드를 부모 노드의 오른쪽에..
		ChangeRightSubTree(mpNode,
					GetRightSubTree(mNode));
	}
}
```

삭제할 노드의 오른쪽 서브 트리에서 가장 작은 값을 지니는 노드를 찾아서 이것으로 삭제할 노드를 대체한다 그러므로 자장 작은 값을 지니는 노드를 찾이려면 NULL을 만날 때까지 왼쪽 자식 노드를 계속해서 이동해야 한다. 그러니 자식 노드가 있다면 오른쪽 자식 노드가 존재하기 때문이다.

# **8.** 이진 탐색 트리

**BinaryTree 와 BinarySearchTree의 구현 부분의 확장이 많이 이루어 졌을 것이다.**
**BinaryTreeAddDelete의 코드들을 비교해서 알아보자.**