c++ & qt day3 hw_2 알고리즘 보고서

로빛 20기 인턴 현창석

목차

1. 정렬 - 정렬 알고리즘 설명

정렬 알고리즘이란?

● 데이터 요소들을 일정한 순서(숫자 크기 또는 사전식 등)로 재배치하는 과정.

●목적 :

- 알고리즘을 사용한 프로그램의 효율성 증가
- 사람이 읽기 편하도록 설계 및 표현
- 탐색과 처리를 편리하게 하기 위함

●알고리즘 선택 기준 :

- 데이터의 양
- 데이터 일부가 정렬이 이미 되어 있는 상태인가
- 안정성 측면에서 뛰어난가

정렬 알고리즘은 저장된 요소들의 목록을 순서대로 재배치하는 알고리즘을 말한다. 흔히 숫자나 사전 순서를 정렬하는데에 많이 사용되며 좀 더 효율적으로 알고리즘을 설계하거나 사람이 읽기 편하게 하도록 하기 위함 등등의 이유로 정렬 알고리즘을 사용한다. 지금까지 개발되어온 정렬 알고리즘들 중에서 아쉽지만 아직 모든 방면에서 최고의 성능을 내는 것은 존재하지 않는다. 따라서 프로그램 설계 시 어떤 정렬 알고리즘을 사용할지 선택을 하는데 이때 고려해야할 사항이 있다.

정렬 알고리즘 선택 시 주의사항

주의해야할 사항 첫 번째는 정렬할 데이터의 양이 얼마나 되는지를 고려해야한다. 데이터 한꺼번에 많이 몰려왔을 때 오류가 갑자기 발생하 수도 있기 때문이다. 또한 이미 정렬이 되어있는 경우의 데

이터를 이용한다면 효율성이 떨어지기 때문에 효율성을 높이기 위해 다른 정렬 알고리즘 방법을 사용해야한다. 마지막으로 안정성이다 동일한 값을 지닌 상대적인 순서를 보존해야할 경우, 예를 들어 같은 점수를 가진 학생을 처리해야 할 때 안정적인 알고리즘을 선택하는 것이 필요하다.

정렬 알고리즘
선택 정렬 알고리즘에 대해서 먼저 설명하겠다. 선택 정렬은 선택한 값과 나머지 데이터를 비교하여 알맞은 자리를 찾아내는 알고리즘이다.

```
void Select(int arr[],int n)
{
        int i,j,min;

        for(i = 0;i <n - 1;i++)
        {
                min = i;
                for(j=i+1;j<n;j++)
                {
                        if(arr[j] < arr[min])
                        {
                                min=j;
                        }
                }
        //스왑은 대충 바꿔주는 함수...
        swap(&arr[min], &arr[i]);
        }
}//End of Select
```

이 사진은 선택 정렬 알고리즘을 코드로 나타낸 사진이다. 이중반복문에서 가장 바깥 반복문의 I를 기준으로 이외에 나머지 데이터와 비교하여 정렬한다.

그 다음 두 번째로 버블 정렬이다. 버블 정렬은 인접한 두 수를 비교해 오름차순 또는 내림차순으로 정렬한다.

```
void BubbleSort(int* _arr, int _length)
{
	int i = 0;
	int j = 0;
	int temp = 0;

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

이 사진은 버블 정렬을 코드로 나타낸 사진이다. 이 또한 선택 정렬 알고리즘과 같이 이중반복문에서 가장 바깥 반복문의 I를 기준으로 이외에 나머지 데이터와 비교하여 오름차순 또는 내림차순으로 정렬한다. 다만 이때 선택 정렬 알고리즘과 다르게 j 의 근방에서 정렬을 수행한다.

세 번째로 삽입 정렬이다. 삽입 정렬 알고리즘은 데이터 집합을 순회하면서 필요한 요소만 골라내 이를 다시 정렬하여 삽입하는 알고리즘이다. 성능적인 부분에서 버블정렬 알고리즘 보다 우월하다고 한다.

```
void InsertionSort(int arr[], int length)
{
        int i = 0;
        int j = 0;
        int key = 0;

        for (i = 1; i < length; i++)
        {
                key = arr[i];

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

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

가장 바깥쪽 반복문은 정렬할 원소를 차례대로 선택하는 것이고 그

안쪽 반복문은 정렬할 원소보다 아래에 있는 요소와 비교하여 정렬을 수행한다.

네 번째로 설명할 정렬 알고리즘은 퀵 정렬 알고리즘이다. 이 정렬 알고리즘에는 피봇이라는 개념이 사용된다. 피봇이란 정렬 될 기존 원소를 말한다. 보통 배열의 첫 번째 값 또는 중앙값으로 피봇을 선택한다. 비봇은 계속 설정되어 부분집합을 만들고 배열의 길이가 1이 되면 정렬이 완료되었다고 판단한다.

```
void QuickSort(int left, int right)
{
        int pivot = arr[(left+right)/2]; //피봇 중심 선정
        int startIndex = left;
        int endIndex = right;

        while (startIndex <= endIndex) //startIndex가 endIndex보다 높아질때까지 while
        {
                while (arr[startIndex] < pivot) //피벗보다 왼쪽에서 피벗보다 큰값 찾기
                {
                        ++startIndex;
                }
                while (arr[endIndex] > pivot) //피벗보다 오른쪽에서 피벗보다 작은값 찾기
                {
                        --endIndex;
                }

                if (startIndex <= endIndex) //그렇게 찾아진 왼쪽 오른쪽 값을 서로 swap
                {
                        Swap(arr[startIndex], arr[endIndex]);
                        ++startIndex;
                        --endIndex;
                }
        }

        if (left < endIndex) //피벗기준 왼쪽 smaller들 정렬
        {
                QuickSort(left, endIndex);
        }
        if (startIndex < right)//피벗기준 오른쪽 bigger들 정렬
        {
                QuickSort(startIndex, right);
        }
}
```

이 사진이 바로 퀵 정렬 알고리즘을 코드로 나타낸 것이다.

다섯 번째로 설명할 알고리즘은 병합 정렬 알고리즘이다. 퀵 정렬 알고리즘과 비슷하지만 퀵 정렬은 피봇 선택 이후에 피봇 기준으로 대소를 비교하는 반면에 병합 정렬은 원소가 하나만 남을 때까지 계속 이분할 한다. 그 후 대소관계를 비교하면서 점차 하나씩 배열을 합쳐나간다. 최종적으로 하나의 배열로 병합되면 알고리즘은 종료된다.

```
void Merge(int left, int right)
{
	//분할된 arr을 tempArr에 복사한다.
	for (int i = left; i <= right; i++)
	{
		tempArr[i] = arr[i];
	}

	int mid = (left + right) / 2;

	int tempLeft = left;
	int tempRight = mid+1;
	int curIndex = left;

	//temparr배열 수정중. 왼쪽 분할과 오른쪽 분할 비교하여
	//더 작은 값을 원래 배열에 복사
	while (tempLeft <= mid && tempRight <= right)
	{
		if (tempArr[tempLeft] <= tempArr[tempRight])
		{
			arr[curIndex++] = tempArr[tempLeft++];
		}
		else
		{
			arr[curIndex++] = tempArr[tempRight++];
		}
	}
	//왼쪽 분할에 남은 원소들을 원래 배열에 복사
	int remain = mid - tempLeft;
	for (int i = 0; i <= remain; i++)
	{
		arr[curIndex + i] = tempArr[tempLeft + i];
	}
}
void Partition( int left, int right)
{
	if (left < right)
	{
		int mid = (left + right) / 2;
		Partition(left, mid);
```

이 사진이 합병 알고리즘을 코드로 나타낸 것과 같다.

다음 여섯 번째로 설명할 알고리즘은 힙 정렬이다. 힙 정렬이란 트리 기반의 자료구조로 불리우고 두 개의 노드를 가진 이진 트리를 의한다. 힙은 크게 최대 힙과 최소 힙으로 나뉘고 각각 내림차순 정렬과 오름차순 정렬에 사용된다. 최대힙은 부모 노드가 항상 자식 도브보다 작으며 반대로 죄소힙은 부모노드가 항상 자식노드보

다 작다.

마지막으로 설명할 알고리즘은 쉘 정렬 알고리즘이다. 쉘 정렬은 interval이라는 개념을 사용한다. interval은 비교할 원소 간 간격을 의미한다. 쉘 정렬 알고리즘에서는 비교 횟수를 최소화하기 위해 interval은 큰 값에서 작은 값으로 내려간다. 배열에서 interval 만큼 떨어진 원소들을 부분집합으로 구성하고 삽입 정렬을 진행하여 정렬을 수행한다.

각 알고리즘의 특징과 장단점

선택 정렬은 배열에서 가장 큰 원소 또는 작은 원소를 찾아서 차례데로 앞쪽으로 가져온다는 특징을 가진다. 다만 시간 복잡도가 다른 알고리즘에 비해 크다는 단점이 있다. 하지만 구현이 매우 간단하고 메모리 사용이 적기 때문에 교육용 또는 원소의 개수가 적을 때 효율적으로 사용이 가능하다,(ex: 로빛 인턴 명단 이름순으로 정렬하기)
다음으로 버블 정렬은 인접해 있는 두 개의 원소를 비교하고 조건에 맞다면 교환하면서 값을 밀어낸다는 특징이 있다 버블 정렬 또한 시간 복잡도가 크며 구현이 단순하고 안정적이기 때문에 교육용 또는 자료가 복잡하게 정렬되지 않은 경우에 주로 사용된다.(거의 정렬된 상태에서)
삽입 정렬은 배열을 나누어 삽입하면서 정렬하는 특징을 가진다. 거의 정렬된 상태의 데이터에서 효율성을 드러낸다. 다만 데이터의 개수가 많아질 경우 비교하는 횟수와 데이터의 교환이 많아져 비효율성을 드러낸다. 저점이 매우 낮다고 볼 수 있다.
병합 정렬은 배열을 나누어 각각 정렬한 후 합쳐지는 특징을 가진다. 시간 복잡도가 항상 크기 않기 때문에 대용량 데이터 및 안정성이 필요한 부분에서 주로 사용된다. 다만 추가 메모리가 필요하며 구현이 복잡하다는 특징이 있다. (ex: 쇼핑몰 사이트에서 수백만

의 상품 데이터를 정렬)
퀵 정렬은 피봇을 기준으로 작은 값, 큰 값을 분할하여 정렬한다.
시간 복잡도가 평균적으로는 크지 않지만 정렬할 데이터가 매우 많다면 시간 복잡도가 매우 커져 안정적이지 않다는 특징이 있다.
퀵 정렬은 일반적인 대규모 데이터를 정렬할 때 주로 사용된다.
(ex:컴퓨터에서 수백만의 로그 데이터를 정렬)
쉘 정렬은 삽입 정렬을 보완한 형태로 일정한 간격으로 떨어진 원소들끼리 부분적으로 정렬한 후 간격을 계속 좁혀나가며 정렬한다.
간격의 선택에 따라 시간 복잡도가 천차만별이라는 특징이 있다.
구현이 쉽고 삽입 정렬보다 훨씬 빠르다는 특징이 있지만 안정적이지 않고 최적의 성능을 계속해서 보장하기 힘들다는 단점이 있기 때문에 작지도 크지도 않은 중간 크기의 데이터를 정렬할 때나 또는 메모리가 제한된 환경에서 주로 사용된다. (ex: 몇 만개 정도의 센서 데이터)
마지막으로 힙 정렬은 힙 구조를 이용해 원소를 하나씩 정렬한다.
시간 복잡도가 항상 일정하게 작아 추가 메모리가 거의 필요없으며 최악의 성능 또한 일정하다는 장점이 있다. 다만 안정적이지 않고 실제적인 속도가 퀵 정렬 보다 느리기 때문에 메모리 제한이 있거나 최악의 성능이 일정해야하는 환경에서 주로 사용된다.ㄲ
(ex: 가장 다리가 긴 사람 찾기)

● 선택 정렬
- 특징 : 배열에서 최소(최대) 원소를 찾아서 차례대로 앞으로 이동함
- 장점 : 구현이 간단, 메모리 사용 적음
- 단점 : 시간 복잡도가 매우 큼
- 활용 부분 : 교육용으로 사용 또는 소규모 데이터를 활용,
(ex: 로빛 인턴 명단 이름순 정렬)

●버블 정렬
- 특징 : 인접한 두 원소를 비교하고 교환하면서 정렬
- 장점 : 구현이 단순함 , 안정성이 높음
- 단점 : 선택 정렬과 비슷하게 시간 복잡도가 매우 큼
- 활용 부분 : 정렬이 어느정도 되어 있는 데이터, 또는 교육용으로 사용

●삽입 정렬
- 특징 : 데이터를 부분집합으로 나누고 삽입 및 정렬
- 장점 : 구현이 간단하고 작은 크기의 데이터에서 빠름, 안정성이 높음
- 단점 : 대규모 데이터를 정렬할 때에는 매우 비효율적임
- 활용 부분 : 정렬이 어느정도 되어 있는 데이터, 또는 교육용으로 사용

●퀵 정렬
- 특징 : 피봇을 기준으로 하여 분할하고 정렬함
- 장점 : 평균적으로 성능이 뛰어남
- 단점 : 저점이 매우낮음, 안정성이 떨어짐
- 활용 부분 : 대규모 데이터 활용, 수백만의 로그 데이터를 정렬

●병합 정렬
- 특징 : 데이터를 분할하고 병합하며 정렬
- 장점 : 안정성이 높음, 시간 복잡도가 거의 일정함
- 단점 : 추가 메모리가 필요함. 구현이 복잡함

활용 부분: 대용량 데이터, 컴퓨터에서 수백만의 로그 데이터를 정렬

●힙 정렬
-특징 : 힙 트리 구조를 이용하여 원소를 하나씩 정렬
- 장점 : 메모리를 효율적으로 사용 가능, 시간 복잡도가 거의 일정, 저점이 높다.
- 단점 : 안정성이 떨어짐, 실제 속도는 퀵 정렬보다 느림
활용 부분: ex: 가장 다리가 긴 사람 찾기

●쉘 정렬
- 특징 : 일정한 간격으로 떨어진 원소끼리 부분적으로 정렬, 간격을 서서히 줄여감
- 장점 ; 삽입 정렬을 보완한 형태임, 구현이 간단하고 중간 규모의 데이터에서 높은 효율성을 보임.
- 단점 : 안정성이 떨어짐, 간격 선택에 따라 시간 복잡도가 달라짐
- 활용 부분 : 수만 개 수준의 센서 데이터 정렬, 메모리 제한 환경

2. 탐색 - 탐색 알고리즘 설명

탐색 알고리즘이란?

탐색 알고리즘이란 수백만개의 방대한 데이터에서 목적에 맞는 데이터를 찾아내기 위한 알고리즘을 말한다. 일반적으로 탐색 알고리즘이라고 하면 트리 검색 알고리즘을 떠올리는 사람들이 많지만 그 이상 다양한 범위의 탐색 알고리즘이 존재한다.

먼저 행렬과 배열 기반의 탐색 알고리즘을 설명하겠다. 먼저 선형 탐색은 가장 원초적인 탐색 알고리즘이라고 할 수 있다. 모든 자료를 비교해가며 원하는 값을 가진 자료를 찾는다.
두 번째는 행렬 패턴 매칭 기반 탐색 알고리즘이다. 이는 특정 패턴을 가진 행렬을 큰 데이터 내에서 찾는 알고리즘이다.

각 알고리즘의 특징과 장단점

선형 탐색 알고리즘은 모든 원소를 처음부터 끝까지 계속 확인한다는 특징이 있다. 구현이 매우매우 간단하며 정렬이 요구되지 않는다는 장점이 있다. 하지만 매우 비효율적이고 대규모 데이터에는 적합하지 않다는 단점이 있다. 매우 간단한 탐색에만 사용한다.
행렬 패턴 매칭 기반 알고리즘은 큰 행렬 데이터 속에서 특정 패턴을 찾는 탐색 알고리즘이라는 특징이 있다. 이미지 인식이나 특정 도형, 패턴 검색 등등 다양하게 응용이 가능하기 때문에 일반적인 선형 탐색보다 복잡한 패턴 탐색에 사용할 수 있다는 특징이 있다. 다만 단순 구현할 시에 시간 복잡도가 매우 커지고 구현이 매우 복잡해 데이터가 크거나 패턴이 여러개 있을시 성능이 저하된다는 단점이 있다.

●선형 탐색
- 특징 : 모든 원소를 처음부터 끝까지 순차적으로 정렬
- 장점: 구현이 간단함, 정렬이 필요하지 않음
- 단점 : 매우 비효율적인 방식임, 원초적, 대규모 데이터를 처리하기에 부적합
- 활용 부분 : 작은 크기의 데이터 , 단순 탐색

●행렬 패턴 기반 매칭 기반 탐색

-특징 : 큰 크기의 행렬 데이터에서 특정 패턴을 탐색함
- 장점 : 이미지 인식이나 패턴탐색 등 다양하게 사용가능, 복잡한 패턴 탐색에도 적합함.
- 단점 : 단순 구현 시 시간 복잡도가 매우 높음, 데이터의 크기가 매우 크거나 패턴이 많아지면 성능이 저하됨
- 활용 부분 : 이미지 처리, 패턴 탐색

3. 트리 - 트리 알고리즘 설명

트리 알고리즘이란?

트리 구조란 노드들이 나뭇가지처럼 서로 연결된 비선형 계층적 자료 구조이다. 트리 내에 하위 트리 -> 하위 트리 이렇게 계속해서 이어지는 자료구조 이다. 대표적 예시로 컴퓨터 directory 구조가 있다.

트리 구조 용어

노드는 트리를 구성하고 있는 기본적인 요소이다. 노드에는 키, 값, 하위 노드에 대한 포인터를 가진다.
간선은 노드와 노드간 연결하는 선이다. 루트 노드는 트리 구조에서 부모 노드를 가지지 않는 최상위 노드를 뜻한다.
부모 노드는 자식 노드를 가진 노드를 말한다. 자식 노드는 부모 노드의 하위노드를 말한다. 형제 노드는 같은 부모를 공유하는 노드를 말한다. 리프 노드는 자식 노드를 가지지 않은 최하위 노드를 말한다. 가지 노드는 자식 노드를 하나 이상 가지고 있는 노드를 말한다,
깊이는 루트 노드에서부터 어떠한 노드까지의 간선 수를 말한다.

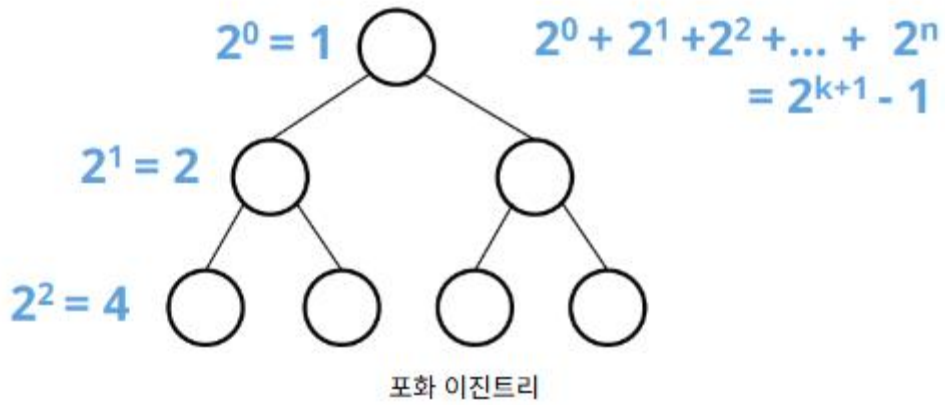
이를 level이라고도 한다. 루트 노드의 깊이는 0이라고 여긴다. 높이는 어떤 노드에서 리프 노드까지의 가장 긴 경로의 간선 수를 말한다. 따라서 리프 노드의 높이는 0이다.
차수는 한 노드의 자식 노드 수를 말한다. path는 한 노드에서 다른 노드까지 이르는 길 사이에 놓여있는 노드들의 순서를 말한다. path length는 해당 경로에 있는 총 노드의 수를 말한다. size는 자신을 포함한 자식 노드의 총 개수를 말한다.
width는 해당 레벨에 있는 노드의 수를 말한다. Breadth는 리프 노드의 수를 말한다. order는 부모 노드가 가질 수 있는 자식의 최대 개수를 말한다,

●트리란 ?
- 노드들이 나뭇가지처럼 서로 연결된 비선형 계층적 자료 구조
- 하위 트리 → 하위 트리 … 계속 이어지는 구조
- 예시) 컴퓨터 디렉토리

●트리 구조용어
- 노드(Node): 트리를 구성하는 기본 요소, 키(Key), 값(Value), 하위 노드 포인터 포함되어 있음.
- 간선(Edge): 노드와 노드를 연결하는 선
- 루트 노드(Root): 부모 노드가 없는 최상위 노드
- 부모 노드(Parent): 자식 노드를 가진 노드
- 자식 노드(Child): 부모 노드의 하위 노드
- 형제 노드(Sibling): 같은 부모를 공유하는 노드
- 리프 노드(Leaf): 자식 노드가 없는 최하위 노드
- 가지 노드(Internal Node): 자식 노드를 하나 이상 가진 노드
- 깊이(Depth / Level): 루트 노드에서 특정 노드까지의 간선 수이다. 루트 노드 깊이 = 0
- 높이(Height): 특정 노드에서 리프 노드까지의 가장 긴 경로 간

선 수이다, 리프 노드 높이 = 0.
- 차수(Degree): 한 노드가 가진 자식 노드 수
- 경로(Path): 한 노드에서 다른 노드까지 연결된 노드들의 순서
- 경로 길이(Path Length): 해당 경로에 있는 총 노드 수
- 크기(Size): 자신을 포함한 자식 노드의 총 개수
- 폭(Width): 특정 레벨에 있는 노드 수
- Breadth: 리프 노드의 수
- 순서(Order): 부모 노드가 가질 수 있는 자식 노드의 최대 개수

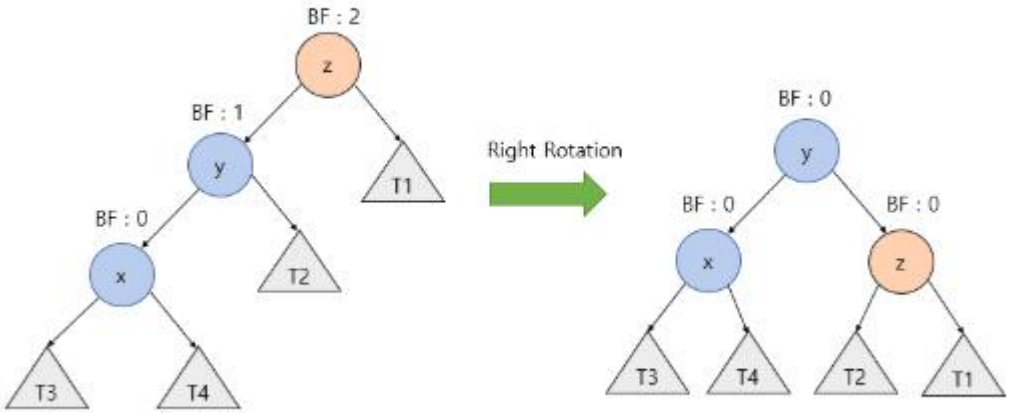
트리 알고리즘 종류



포화 이진트리

이진트리는 각 노드가 최대 2개의 자식 노드를 가지는 트리 구조를 말한다. 최대 2개이므로 자식의 개수는 0 또는 1 또는 2이다. 왼쪽 자식 노드와 오른쪽 자식 노드로 표현한다. 이때 자식이 한 개인 경우가 없는 트리 구조를 정 이진 노드라고 표현한다.
또한 모든 노드가 2개의 자식 노드를 가지고 있으며 leaf 노드가 모두 같은 레벨인 구조를 가지는 트리를 포화 이진트리라고 한다.

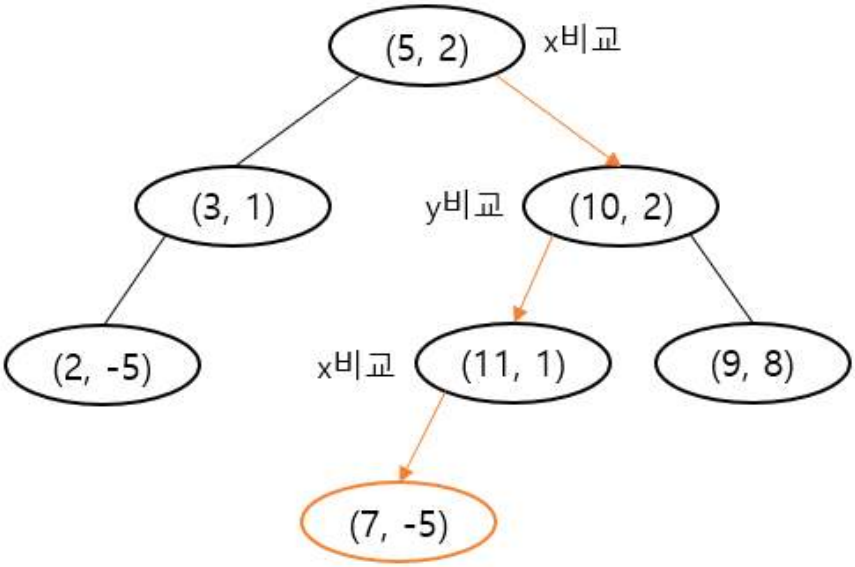
이진탐색트리는 한쪽으로 노드가 쏠릴 수 있다는 단점이 존재하는데 이 단점을 보완한 자료 구조가 AVL 트리구조이다. 이 트리구조는 어떤 노드를 탐색해도 일정한 시간 복잡도를 가진다.
기본적으로 이진 탐색 트리 구조를 따르지만 왼쪽과 오른쪽 서브트리의 높이 차이가 1을 넘어가면 회전을 통해 균형을 맞추어 높이 차이를 줄인다. 따라서 어떤 노드를 탐색해도 일정한 시간 복잡도를 가지는 것이다.
AVL트리는 균형이 무너졌는지에 대해 판단할 때 Balance Factor라는 것을 활용한다. AVL트리는 모든 노드의 BF가 -1,0,1 중 하나여야하며 이를 벗어나면 회전을 한다.



[그림 4] 우회전(Right Rotation)

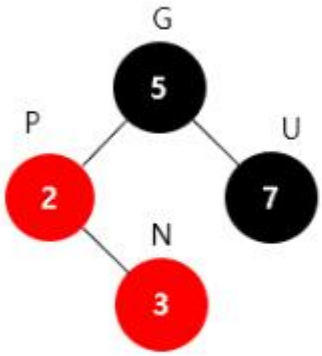
회전의 기준이 z노드라고 가정했을 때 y노드의 오른쪽 자식 노드를 z노드로 변경한 후 z노드의 왼쪽 자식 노드를 y노드의 오른쪽 t2로 변경한다.

KD 트리 구조는 다차원의 좌표 데이터를 효율적으로 탐색하기 위한 이진탐색트리를 확장한 버전이다. 각 노드가 k차원의 좌표를 가지고 있고, 이는 트리구조를 이룬다.

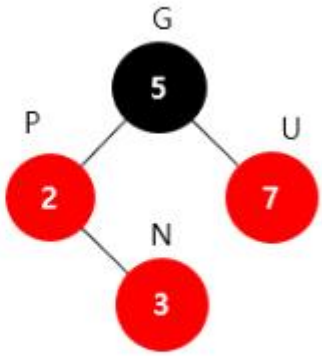


루트노드부터 시작하여 트리의 레벨에 따라 분할할 차원을 선택한다, 2차원 (x,y) 좌표가 있다면 루트 노드의 x 기준으로 분할하고 그 다음 레벨에서는 분할된 노드의 y를 기준으로 분할한다. 이렇게 서브트리로 x -> y 순으로 번갈아가며 분할된다.

레드 블랙 트리는 자가 균형 이진 탐색 트리라고 불린다. 이 트리 구조는 다음과 같은 조건을 만족해야한다.
1.모든 노드는 빨간색 혹은 검은색이다. 2.루트 노드는 검은색이다. 3.모든 리프 노드(NIL)들은 검은색이다. 여기서 NIL은 null leaf의 약어로 자료를 갖지 않고 트리의 끝을 나타내는 노드를 뜻한다. 4. 빨간색 노드의 자식은 검은색이다. 즉 빨간색 노드가 연속으로 나올 수 없다 5. 모든 리프 노드에서 Black Depth는 같다. 즉 리프 노드에서 루트 노드까지 가는 경로에서 만나는 검은색 노드의 개수가 같다.



U가 검은색 -> Restructuring



U가 빨간색 -> Recoloring

빨간색 색칠된 부분이 연속으로 2개 발생 했을 때 삼촌 노드가 검은색이면 restructing을 수행한다. 이는 다음 규칙이 있다.
1. 새로운 노드(N), 부모 노드(P), 조상 노드(G)를 오름차순으로 정렬, 2. 셋 중 중간값을 부모 노드로 만들고 나머지 둘을 자식 노드로 만든다, 3. 새로 부모가 된 노드를 검은색으로 만들고 나머지 자식들을 빨간색으로 칠한다.
빨간색 색칠된 부분이 연속으로 2개 발생 했을 때 삼촌 노드가 빨간색이면 recoloring을 실행한다. 이는 다음과 같은 규칙을 거친다. 1. 새로운 노드(N)의 부모(P)와 삼촌(U)을 검은색으로 바꾸고 조상(G)을 빨간색으로 바꾼다. 여기서 조상(G)이 루트 노드라면 검은색으로 바꾼다. 만약에 조상(G)을 빨간색으로 바꿨을 때 또다시 빨간색 색칠된 부분이 연속으로 2개 발생하면
Double Red 문제가 발생하지 않을 때까지 Restructuring 혹은 Recoloring을 진행하여 반복한다.

●이진 트리
- 각 노드가 최대 2개의 자식 노드를 가진다. 자식 수: 0, 1, 2
왼쪽/오른쪽 자식으로 표현
- 자식이 1개만 있는 트리를 정이진노드라고 부른다
- 모든 노드가 2개의 자식을 가지며 리프 노드가 같은 레벨을 가진 트리구조를 포화 이진 트리라고 부른다.

●AVL 트리
- 자가 균형 이진 탐색 트리구조이다.
- 왼쪽 또는 오른쪽 서브트리 높이 차이가 1초과 시 회전하여 균형 유지
-모든 노드의 BF는 -1, 0, 1를 벗어나지 않는다
- 회전 방식: 기준 노드 Z를 설정 -> 오른쪽 자식 Y → Z , Z의 왼쪽 자식 → Y의 오른쪽 T2 회전

●KD 트리
- 다차원 데이터 탐색용 이진 탐색 트리를 확장한 버전이다.
- 각 노드는 k차원 좌표로 구성
- 분할 방식
    레벨에 따라 분할 기준 차원 변경
ex) 2차원 → 루트: x 기준, 다음 레벨: y 기준, 계속해서 반복함

●레드 블랙 트리
- 자가 균형 이진 탐색 트리이다.
- 조건:
    모든 노드는 빨강 또는 검정
    루트 노드는 검정색이다.
    모든 리프(NIL) 노드는 검정색이다.
    빨강 노드의 자식은 검정 , 빨강 연속이 연속되면 안된다.

루트에서 리프 경로의 검정 노드 수는 동일해야한다.

- 불균형 발생 시 :

Restructuring: 빨강 연속 2개 + 삼촌 검정

N, P, G 오름차순 정렬

중간값 → 부모, 나머지 → 자식

부모: 검정, 자식: 빨강

Recoloring: 빨강 연속 2개 + 삼촌 빨강

부모(P), 삼촌(U) → 검정

조상(G) → 빨강 (루트면 검정)

Double Red 문제 해결될 때까지 반복한다.

- 각 알고리즘의 특징과 장단점

이진트리는 1차원 배열로 표현할 수 있다는 특징을 가진다.
루트 노드부터 시작하여 왼쪽 노드에서 오른쪽 노드까지 순서를 매겨 1차원 배열로 표현할 수 있다. 여기서 i번째 인덱스에 들어있는 노드의 부모는 I/2을 하여 인덱스 위치를 찾을 수 있고 노드 I의 왼쪽 자식은 I*2를 한 인덱스에 있다. 오른쪽 자식은 여기에 + 1을 하여 I*2+1을 연산한 인덱스 위치에 있다. 다만 이진 트리 알고리즘의 주된 단점이 있다. 특정 방향으로 노드가 편향적으로 삽입되면 트리가 한쪽으로만 치우친다는 점이다. 이는 효율성을 크게 저하시키고 이를 배열로 표현하면 트리 구조에서 공간 낭비가 발생한다,
AVL트리는 모든 노드에서의 서브트리 높이 차이가 1 이하로 유지되는 트리구조라는 특징을 가진다. 이러한 특징이 트리의 균형을 유지해 최악의 경우에도 일정한 시간 복잡도를 보장한다는 장점이

있다. 다만 이러한 특징 때문에 데이터의 변경 횟수가 많은 환경에서 삽입, 삭제가 많으면 성능이 크게 저하된다.
KD트리는 k 차원 공간을 확장해 트리 각 레벨에서 다른 차원을 기준으로 반복하여 분할한다는 특징이 있다. 범위 탐색이나 최근접 이웃탐색과 같이 다차원 데이터 탐색에서 큰 효율성을 보이고 이진 트리 구조를 기본적으로 따르기 때문에 구현이 상대적으로 쉽다는 특징이 있다. 다만 차원이 늘어날 효율성이 급격히 떨어진다는 치명적인 단점이 있다.

레드 블랙 트리는 위에서 설명한 AVL 트리와 반대되는 면이 있다. 레드블랙트리에서 삽입 또는 삭제 작업 시 균형을 맞추기 위한 작업 횟수가 적고, 각 노드 당 색깔을 표현하는 데 단 1bit의 저장공간만 사용한다.

●이진트리
- 특징 :
    - 1차원 배열로 표현 가능
    - 루트부터 왼쪽→오른쪽 노드 순서대로 인덱스 부여
    - i번째 노드의 부모: i/2, 왼쪽 자식: i2, 오른쪽 자식: i2+1 하여 계산 가능하다.
- 장점 : 구조가 단순하며 구현이 쉽다.
- 단점 :
    - 특정 방향으로 편향되면 탐색 효율성이 낮아짐.
    - 1차원 배열로 표현 시 공간 낭비 발생할 수 있음.

● AVL 트리
- 특징 :
    - 모든 노드의 서브트리 높이 차 1 이하
    - 균형 계속해서 유지함

- 장점 :
    - 최악의 경우에도 일정한 시간 복잡도 보장함.
- 단점 :
    - 삽입 또는 삭제 시 균형 유지 작업 필요함. 따라서 데이터 변경이 많으면 성능이 크게 저하.

● KD 트리
- 특징 :
    - K차원으로 공간을 확장함
    - 각 레벨에서 다른 차원 기준으로 반복해서 분할함
    - 기본적으로 이진 트리 구조를 기반으로 함.
- 장점 :
    - 다차원 데이터 탐색에 효율적임
    - 범위 탐색 또는 최근접 이웃 탐색에 최적인 트리 구조
    - 구현이 상대적으로 쉬움.
- 단점 ;
    - 차원이 많아지면 효율성이 급격히 감소함.

● 레드 블랙 트리
-특징 :
    - 삽입 또는 삭제 시 균형을 계속해서 유지함.
    - 각 노드 색상을 가지며 이에 1bit를 사용함.
- 장점 :
    - 균형 유지 작업이 AVL트리보다 최소화되어 있음.
    - 저장 공간 효율적으로 사용
- 단점 :
    - AVL트리보다 탐색 성능이 느림
    - 구현이 복잡함.

감사합니다.