# 1. REALDOM VS VIRTUALDOM 각각 설명하시오

* DOM이란 객체로 문서 구조를 표현하는 방법으로 XML이나 HTML로 작성한다. 웹 브라우저는 DOM을 활용하여 객체에 자바스크립트와 CSS를 적용하며 DOM은 트리 형태여서 특정 node를 찾아서 원하는 곳에 삽입하거나 수정, 제거할 수 있다.
* Virtual DOM이란 실제 DOM의 가벼운 사본 개념으로 데이터를 업데이트 했을 때 실제 DOM에 접근하지 않고 DOM을 추상화한 자바스크립트 객체를 구성하여 사용한다.

출처: 리액트를 다루는 기술(김민준 지음)

출처: <https://pgg-dev.tistory.com/14>

# 2. 버츄얼돔은 왜 좋은가. 어떻게 동작하는가. JQUERY의 돔 직접참조에 비해서 무엇이 개선되었는가.

* Virtual DOM은 업데이트 처리를 더욱 간결하게 해준다. Virtual DOM을 활용해 실제 DOM을 업데이트할 때는 크게 세 절차를 밟는데 첫번째, 데이터를 업데이트 하면 전체 UI를 Virtual DOM에 리렌더링 하고 두번째, 이전 Virtual DOM에 있던 내용과 현재 내용을 비교한 뒤 세번째, 바뀐 부분만 실제 DOM에 적용한다. 이런 작업 방식을 통해 DOM 조작 횟수를 최소화하여 브라우저가 페이지를 새로 그리는 작업을 최소화 함으로써 브라우저 성능이 개선된다.
* Virtual DOM을 이용할 때 얻을 수 있는 가장 큰 장점은 더욱 간결하게 업데이트 처리를 할 수 있다는 것이다. 구현된 UI가 많을 수록 데이터를 업데이트 했을 때 새로 그리지 않아도 되는 부분까지 새로 렌더링되면서 브라우저 성능이 눈에 띄게 저하되는데 Virtual DOM은 이런 복잡성을 해소하고 데이터 업데이트를 더욱 쉽게 한다.

출처: 리액트를 다루는 기술(김민준 지음)

# 3. 버츄얼돔이 동작하는 DIFF알고리즘에 대해 설명하시오.

React는 방금 만들어진 트리에 맞게 가장 효과적으로 UI를 갱신하는 방법을 알아내기 위해 아래의 두 가지 가정에 기반하여 휴리스틱 알고리즘을 구현했습니다.

1. 서로 다른 타입의 두 엘리먼트는 서로 다른 트리를 만들어낸다.

2. 개발자가 key prop을 통해, 여러 렌더링 사이에서 어떤 자식 엘리먼트가 변경되지 않아야 할지 표시해 줄 수 있다.

두 개의 트리를 비교할 때, React는 두 엘리먼트의 루트(root) 엘리먼트부터 비교한다. 이후의 동작은 루트 엘리먼트의 타입에 따라 달라집니다.

1. 비교 알고리즘

1) 엘리먼트의 타입이 다른 경우

두 루트 엘리먼트의 타입이 다르면, **React는 이전 트리를 버리고 완전히 새로운 트리를 구축합니다.** <a>에서 <img>로, <Article>에서 <Comment>로, 혹은 <Button> to <div>로 바뀌는 것 모두 트리 전체를 재구축하는 경우입니다. 트리를 버릴 때 이전 DOM 노드들은 모두 파괴됩니다. 컴포넌트 인스턴스는 componentWillUnmount()가 실행됩니다. 새로운 트리가 만들어질 때, 새로운 DOM 노드들이 DOM에 삽입됩니다. 그에 따라 컴포넌트 인스턴스는 componentWillMount()가 실행되고 componentDidMount()가 이어서 실행됩니다. 이전 트리와 연관된 모든 state는 사라집니다.

2) DOM 엘리먼트의 타입이 같은 경우

같은 타입의 두 React DOM 엘리먼트를 비교할 때, **React는 두 엘리먼트의 속성을 확인하여, 동일한 내역은 유지하고 변경된 속성들만 갱신**합니다. 예를 들어,

<div className="before" title="stuff" />

<div className="after" title="stuff" />

이 두 엘리먼트를 비교하면, React는 현재 DOM 노드 상에 className만 수정합니다. style이 갱신될 때, React는 또한 변경된 속성만을 갱신합니다. 예를 들어,

<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />

위 두 엘리먼트 사이에서 변경될 때, React는 fontWeight는 수정하지 않고 color 속성 만을 수정합니다. DOM 노드의 처리가 끝나면, React는 이어서 해당 노드의 자식들을 재귀적으로 처리합니다.

3) 같은 타입의 컴포넌트 엘리먼트

컴포넌트가 갱신되면 인스턴스는 동일하게 유지되어 렌더링 간 state가 유지됩니다. React는 새로운 엘리먼트의 내용을 반영하기 위해 현재 컴포넌트 인스턴스의 props를 갱신합니다. 이때 해당 인스턴스의 componentWillReceiveProps()와 componentWillUpdate()를 호출합니다.

다음으로 render() 메소드가 호출되고 비교 알고리즘이 이전 결과와 새로운 결과를 재귀적으로 처리합니다.

2. 자식에 대한 재귀적 처리

DOM 노드의 자식들을 재귀적으로 처리할 때, React는 기본적으로 동시에 두 리스트를 순회하고 차이점이 있으면 변경을 생성합니다. 예를 들어, 자식의 끝에 엘리먼트를 추가하면, 두 트리 사이의 변경은 잘 작동할 것입니다.

ex1) <ul>

<li>first</li>

<li>second</li>

</ul>

ex2) <ul>

<li>first</li>

<li>second</li>

<li>third</li>

</ul>

React는 두 트리에서 <li>first</li>가 일치하는 것을 확인하고, <li>second</li>가 일치하는 것을 확인합니다. 그리고 마지막으로 <li>third</li>를 트리에 추가합니다.

하지만 위와 같이 단순하게 구현하면, 리스트의 맨 앞에 엘리먼트를 추가하는 경우 성능이 좋지 않습니다. 예를 들어, 아래의 두 트리 변환은 형편없이 작동합니다.

ex1) <ul>

<li>Duke</li>

<li>Villanova</li>

</ul>

ex2) <ul>

<li>Connecticut</li>

<li>Duke</li>

<li>Villanova</li>

</ul>

React는 <li>Duke</li>와 <li>Villanova</li> 종속 트리를 그대로 유지하는 대신 모든 자식을 변경합니다. 이러한 비효율은 문제가 될 수 있습니다.

1) Keys

이러한 문제를 해결하기 위해, React는 key 속성을 지원합니다. **자식들이 key를 가지고 있다면, React는 key를 통해 기존 트리와 이후 트리의 자식들이 일치하는지 확인합니다.** 예를 들어, 위 비효율적인 예시에 key를 추가하여 트리의 변환 작업이 효율적으로 수행되도록 수정할 수 있습니다.

ex 1) <ul>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

ex 2) <ul>

<li key="2014">Connecticut</li>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

이제 React는 '2014' key를 가진 엘리먼트가 새로 추가되었고, '2015'와 '2016' key를 가진 엘리먼트는 그저 이동만 하면 되는 것을 알 수 있습니다.

실제로, key로 사용할 값을 정하는 것은 어렵지 않습니다. 그리려고 하는 엘리먼트는 일반적으로 식별자를 가지고 있을 것이고, 그대로 해당 데이터를 key로 사용할 수 있습니다.

<li key={item.id}>{item.name}</li>

만약 이러한 상황에 해당하지 않는다면, 여러분의 **데이터 구조에 ID라는 속성을 추가**해주거나 **데이터 일부에 해시를 적용해서 key를 생성**할 수 있습니다. 해당 **key는 오로지 형제 사이에서만 유일하면 되고, 전역에서 유일할 필요는 없습니다.**

최후의 수단으로 **배열의 인덱스를 key로 사용**할 수 있습니다. 만약 항목들이 **재배열되지 않는다면 이 방법도 잘 동작할 것이지만, 재배열되는 경우 비효율적으로 동작**할 것입니다. 인덱스를 key로 사용 중 배열이 재배열되면 컴포넌트의 state와 관련된 문제가 발생할 수 있습니다. 컴포넌트 인스턴스는 key를 기반으로 갱신되고 재사용됩니다. 인덱스를 key로 사용하면, 항목의 순서가 바뀌었을 때 key 또한 바뀔 것입니다. 그 결과로, 컴포넌트의 state가 엉망이 되거나 의도하지 않은 방식으로 바뀔 수도 있습니다.

출처: <https://ko.reactjs.org/docs/reconciliation.html>