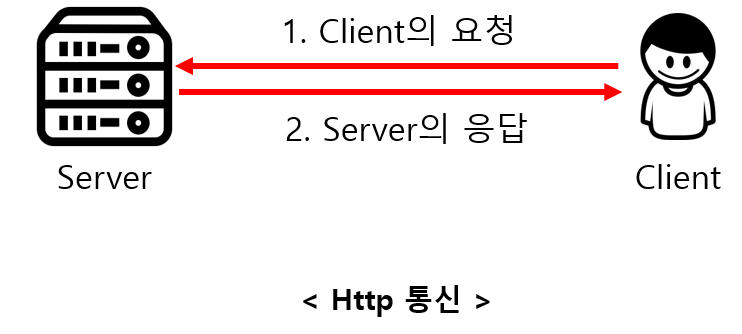
**1. http통신과 소켓통신은 어떻게 틀린가.**

**Http 통신**

Client의 요청(Request)이 있을 때만 서버가 응답(Response)하여 해당 정보를 전송하고 곧바로 연결을 종료하는 방식

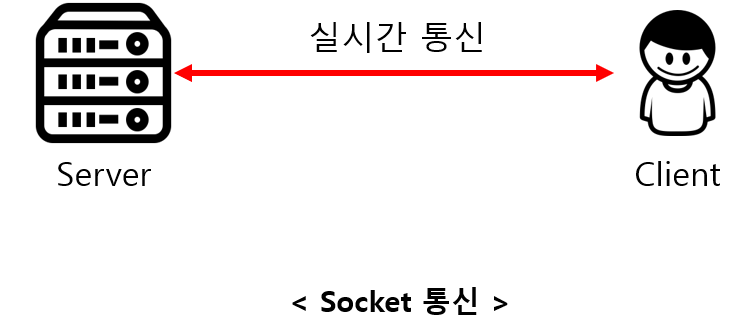
Http통신은 Client의 요청이 있을 때만 Server가 응답하여 처리를 한 후에 연결을 끊는 방식입니다. 이러한 연결 방식은 Client가 요청을 보내는 경우에만 Server가 응답하는 단방향적 통신으로, Server가 Client로 요청을 보낼수는 없습니다. 이러한 Http 통신은 실시간 연결이 아닌, 필요한 경우에만 Server로 접근하는 콘텐츠 위주의 데이터를 사용할 때 용이합니다. 만약 게시물에 대한 내용을 요청하기 위해 실시간으로 연결을 유지하는 Socket통신을 사용하게 되면, 게시물을 받은 후에도 계속 통신을 위한 연결이 성립되어 있어 부하가 걸리게 됩니다. 일반적으로 모바일 어플리케이션은 필요한 경우에만 Server로 정보를 요청하는 경우가 많은데, 이러한 Web Server로 Http 통신을 주로 사용하며 비용 및 유지보수 등 대부분의 방면에서 좋습니다.



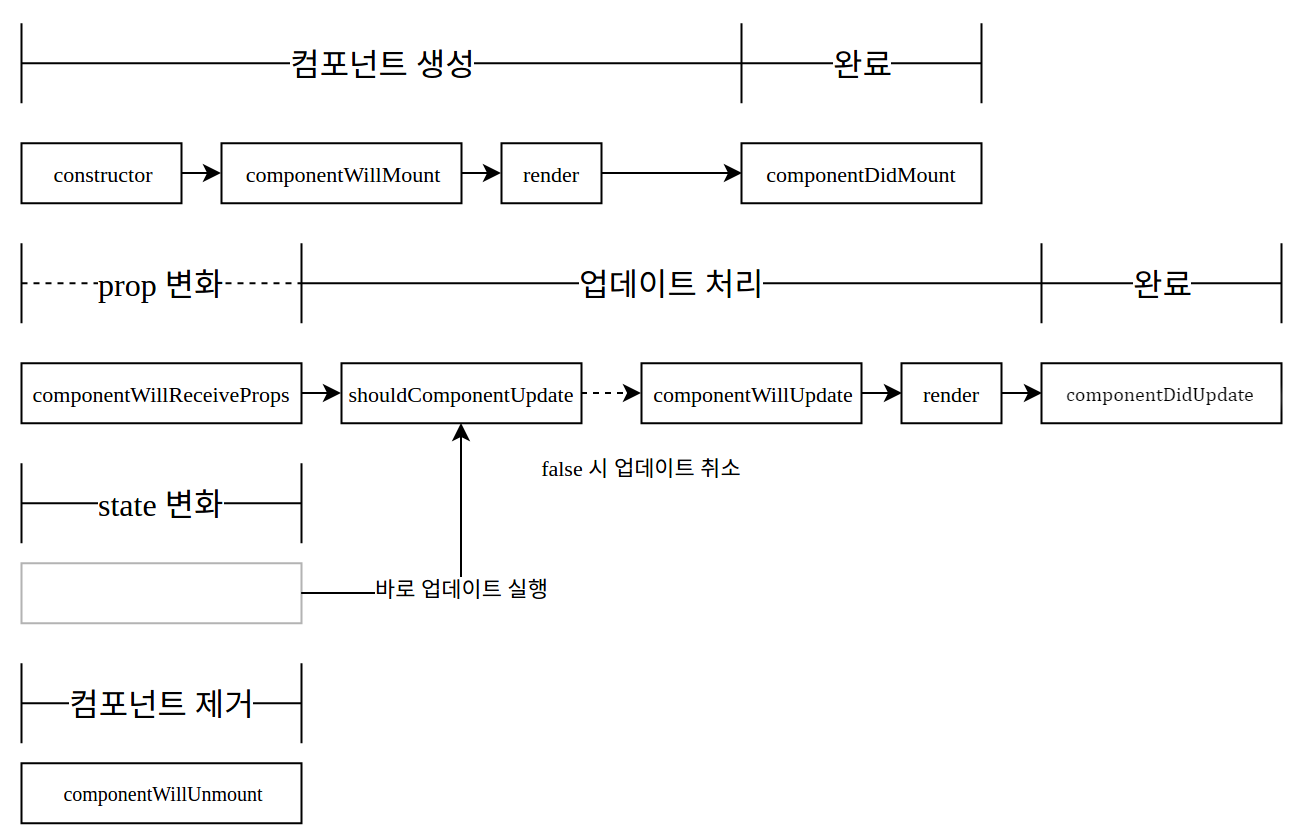
**Socket 통신**

Server와 Client가 특정 Port를 통해 실시간으로 양방향 통신을 하는 방식

Socket통신은 Http 통신과 달리 Server와 Client가 특정 Port를 통해 연결을 성립하고 있어 실시간으로 양방향 통신을 하는 방식입니다. Client만 필요한 경우에 요청을 보내는 Http 통신과 달리 Socket 통신은 Server 역시 Client로 요청을 보낼 수 있으며, 계속 연결을 유지하는 연결지향형 통신이기 때문에 실시간 통신이 필요한 경우에 자주 사용됩니다. 예를 들면, 실시간 Streaming 중계나 실시간 채팅과 같이 즉각적으로 정보를 주고받는 경우에 사용합니다. 예를 들어 실시간 동영상 Streaming 서비스를 Http 통신으로 구현하였다고 가정하겠습니다. 이러한 경우에 사용자가 서버로 동영상을 요청하기 위해서는 동영상이 종료되는 순간까지 계속해서 Http 통신을 보내야 하고 이러한 구조는 계속 연결을 요청하기 때문에 부하가 걸리게 됩니다. 그러므로 이러한 경우에는 Socket을 통해 구현하는 것이 적합합니다.



2. 컴포넌트의 각 생명주기는 어떻게 되고 각 시점은 언제 호출되는가?



React 컴포넌트의 생명 주기를 부분적으로 나누어 간단히 살펴보면 다음과 같습니다.

**Mounting**

React 컴포넌트의 인스턴스가 생성되고 DOM에 삽입되는 과정의 life cycle 입니다.  
constructor  
componentWillMount  
render  
componentDidMount

**Updating**

prop 또는 state의 변경으로 해당 컴포넌트가 re-render 되는 경우의 life cycle 입니다.  
Props Change  
 componentWillReceiveProps  
 shouldComponentUpdate  
 componentWillUpdate  
 render  
 componentDidUpdate

State Change  
shouldComponentUpdate  
componentWillUpdate  
render  
componentDidUpdate

**Unmounting**

componentWillUnmount

이제 각각에 대해서 살펴봅시다.

**componentWillMount**

개발자가 정의한 컴포넌트는 눈 깜짝할 사이에 화면에 그려집니다. 이 순간 무엇을 할 수 있을까요? 그에 대한 대답은 사실 할 수 있는 것이 별로 없습니다. componentWillMount 메소드는 딱히 쓸모가 없습니다. componentWillMount에서는 아직 생성된 컴포넌트가 없으므로 DOM 관련 작업을 수행 할 수 없습니다. 또한 컴포넌트의 기본 구성(props or state)을 설정하는 생성자(constructor)가 호출된 이후에는 아무 것도 변경되지 않았습니다.

export default class Sidebar extends Component {

  tooltipsEnabled = true;

  constructor(props) {

    super(props);

    this.state = {

      analyticsOpen: false,

      requirementsOpen: false,

      brandInfoOpen: false,

    }

  }

}

생성자가 호출되고 나면 그 상태가 바로 컴포넌트의 초기 상태가 됩니다. 추가적인 라이프 사이클 메소드를 복잡하게 사용하지 않고도 거의 모든 것이 컴포넌트 코드의 나머지 부분에서 처리되어야 합니다. 그 중에서도 예외가 있다면 초기에만 수행할 수있는 설정 즉, 외부 API에 연결하는 작업이 있을 것입니다. 예를 들어, 앱에 Firebase를 사용한다면, 앱이 처음 설치 될 때 설정을 가져와야합니다. 그러나 여기서 핵심은 이러한 작업은 앱의 최상위 컴포넌트 (루트 컴포넌트)에서 수행되어야 한다는 것입니다. 이는 컴포넌트의 99 %가 componentWillMount를 사용하지 않아야 함을 의미합니다.

루트 컴포넌트에서 App과 관련된 외부 API를 설정할 떄 사용됩니다.

**componentDidMount**

이제 컴포넌트가 mount되고 사용할 준비가 되었습니다. componentDidMount는 가지고 놀 컴포넌트가 없을 때 할 수 없었던 것들을 모두 할 수 있는 메소드입니다. 기본적으로 여기에서는 DOM에 대한 접근이 필요한 모든 설정을 수행하고 필요한 데이터를 가져오기 시작합니다.

Ajax 호출을 시작하여 컴포넌트에서 사용해야 하는 데이터를 로드합니다.  
setState를 호출할 수 있습니다.

**componentWillReceiveProps**

컴포넌트가 정상적으로 잘 동작하고 있었는데 새로운 props가 전달되었습니다. 아마 상위 컴포넌트에서 componentDidMount에 의해 로드된 데이터 중 일부가 내려왔을 것입니다. 새로운 props로 어떠한 작업을 수행하기 전 이 새로운 props를 인자로 하여 componentWillReceiveProps가 호출됩니다.

componentWillReceiveProps(nextProps) {

  if (parseInt(nextProps.id, 10) !== parseInt(this.props.id, 10)) {

    this.setState({ postsLoaded: false });

    this.contentLoaded = 0;

  }

}

우리는 이 메소드 안에서 nextProps를 통해 다음의 props(새로운 props)에 접근할 수 있고 this.props를 통해 현재의 props에 접근할 수 있습니다. 이 메소드에서 우리 해야하는 것은 다음과 같습니다.

props가 바뀌었는지 확인합니다. 가끔 props가 변경되지 않은 경우에도 호출됩니다.  
만약 props가 변경되었다면 그에 따라 로직을 구성해야 한다.  
새로운 props를 받았을 때, 다음과 같이 처리합니다.

componentWillReceiveProps(nextProps) {

  if (this.props.percent !== nextProps.percent) {

    this.setUpPercent(nextProps.percent);

  }

}

하나 더 주의해야 할 것은 componentWillReceiveProps는 초기 렌더링시 호출되지 않습니다. 코드 상으로는 컴포넌트가 props를 받는다는 것을 의미하지만 비교할 기존의 props가 없으므로 로직에 포함되지 않습니다.

해당 컴포넌트의 상태 변경에 영향을 끼지는 props의 변경에 따라 로직을 구성합니다.  
setState를 호출할 수 있습니다.

**shouldComponentUpdate**

컴포넌트가 불안정한(?) 상태입니다. 이 메소드가 호출되었다는 것은 현재 컴포넌트의 상태와 view에 그려진 내용, 상태가 다르다는 것을 의미하기 때문입니다. 우리에게는 새로운 props가 있습니다. 전형적인 React 신조에 따르면 컴포넌트가 새로운 props나 새로운 state를 받으면 업데이트해야 한다고 말합니다. shouldComponentUpdate 메소드는 nextProps를 첫 번째 인수로 사용하고 nextState는 두 번째 인수로 사용합니다.

shouldComponentUpdate(nextProps, nextState) {

  return this.props.engagement !== nextProps.engagement

    || nextState.input !== this.state.input;

}

shouldComponentUpdate는 항상 boolean을 반환해야 합니다. 이것은 “해당 컴포넌트를 다시 렌더링(re-render)해야 하나요?”라는 질문에 대한 답이 됩니다. 해당 컴포넌트를 특별히 override하지 않는 이상 이 메소드는 기본적으로 true를 반환합니다.

하지만 불필요한 재렌더링(re-render)를 방지하기 위해서는 shouldComponentUpdate를 override 할 수 있습니다. 자세한 내용은 다음 링크를 참고해주세요. (이 링크에서는 많은 필드가 있는 테이블에서 테이블이 다시 렌더링 될 때 각 필드가 다시 렌더링되어 작업속도가 느려지게 된다는 것을 보여주고 있습니다.)

shouldComponentupdate를 사용하면 변화에 신경써야 하는 props의 변경에 대해서만 update를 진행할 수 있습니다. 하지만 이에 대한 설정을 잊으면 React 컴포넌트가 제대로 동작하지 않기 때문에 명심해야 합니다.

해당 컴포넌트의 re-render를 제어할 때 사용합니다.  
setState를 호출할 수 없습니다.

**componentWillUpdate**

방금 전까지 컴포넌트를 update하는데 집중을 했는데요, 이 메소드에서는 update가 발생하기 전 어떠한 작업이 필요한 경우 해당 메소드 안에서 해결할 수 있습니다. 이 메소드는 메소드 내에서 this.setState를 호출할 수 없다는 점을 제외하면 기본적으로 componentWillReceiveProps와 동일합니다.

이미 shouldComponentUpdate를 사용 중인데 또 props가 변경될 때 작업이 필요한 경우 이 메소드가 적절하겠지만 추가적인 유용성은 많이 주지는 못할 것입니다.

shouldComponentUpdate가 이미 사용되고 있는 컴포넌트에서 componentWillReceiveProps 대신 사용됩니다. 단 해당 메소드 내에서는 이전 props에 접근할 수 없습니다.  
setState를 호출할 수 없습니다.

**componentDidUpdate**

이 메소드에서는 componentDidMount에서 했던 것과 동일한 작업을 수행할 수 있습니다. (예를 들면, 레이아웃을 다시 설정한다던가 canvas를 그린다던가) shouldComponentUpdate가 호출된 다음, render가 호출된 다음 호출되는 메소드입니다. 하지만 아쉽게도 이 메소드 내에서는 어떠한 이유로 update가 이루어졌는지 알 수 없습니다. 그렇기 때문에 변경된 데이터에 대해 DOM조작을 다시 해줘야하는 경우가 있다면 이 메소드에서 수행해줄 수 있습니다. componentWillReceiveProps에서도 충분히 해줄 수 있는 작업들이지만 불필요한 re-render를 방지하기 위해 DOM과 관련된 작업들을 주로 수행합니다.

componentDidUpdate() {

  this.createGrid();

}

prop 또는 state에 따라서 DOM을 업데이트 합니다.  
setState를 호출할 수 있습니다.

**componentWillUnmount**

컴포넌트가 unmount 되기 전 호출되는 메소드로 마지막으로 수행해야하는 작업이 있다면 이 메소드에서 수행해줄 수 있습니다. 여기서는 네트워크 요청을 취소하거나 컴포넌트와 관련된 모든 이벤트 리스터를 제거할 수 있습니다. 기본적으로 해당 컴포넌트와 관련있는 작업들만 수행합니다.

컴포넌트와 관련된 것들을 제거하는 메소드입니다.  
setState를 호출할 수 없습니다.

3. 자바스크립트에서 사용하는 자료구조에는 어떤것들이 있는가?

큰 틀로, 기본적인 3가지 유형의 자료구조가 있다.

**Array :** Stack과 Queue는 아이템을 어떻게 삽입/삭제 하는가를 다루는 배열 같은 자료구조들이다.

**LinkedList :** 링크드 리스트, 트리 그리고 그래프는 다른 노드에 대한 참조를 유지하는 노드로 이루어진 구조체이다.

**Hash :** 해쉬테이블은 데이터를 저장하고 찾기 위해 해쉬 함수에 의존한다.

 복합성 측면에서, Stack과 Queue는 가장 간단하며 Linked List로 구성할 수 있다. Tree 와 Graph 는 링크드 리스트의 개념을 확장시킨 것이기 때문에 가장 복잡하다. Hash Table 은 이런 자료구조들을 활용하여 안정적으로 돌아가야한다.

 효율성 측면에서, Linked List는 데이터의 기록 및 저장에 가장 최적화되어있으며 Hash Table은 데이터의 탐색과 검색에 가장 효과적이다.

**Stack**

 의심의 여지없이 자바스크립트에서 가장 중요한 Stack은 우리가 실행하는 펑션의 스코프를 푸쉬하는 call stack 일 것이다. 프로그래밍적으로 스택은 단순히 push와 pop의 두 가지 정책을 가진 배열이다. Push는 배열의 탑에 원소를 추가한다. 반면 Pop은 같은 위치에서 원소를 제거한다. 즉, Stacks 은 “Last In, First Out” 정책을 따른다(LIFO).   
아래는 Stack 코드의 예다..

class Stack {

    constructor(...*items*) {

        this.reverse = false;

        this.stack = [...*items*];

    }

    push(...*items*) {

        return this.reverse ? this.stack.unshift(...*items*) : this.stack.push(...*items*);

    }

    pop(...*items*) {

        return this.reverse ? this.stack.shift() : this.stack.pop();

    }

}

 아이템의 숫자가 증가함에 따라, push/pop은 unshift/shift보다 더 성능이 좋아지게 된다. 그 이유는 후자의 경우 모든 아이템이 다시 인덱싱을 해야하기 때문이다.(push/pop 은 최상단 index 만 변하기 때문) 우리는 스택의 순서를 뒤집을 수 있다는 점에 유의해야한다. 바텀이 탑으로가고 탑이 바텀에 간다는 뜻이다

**Queue**

 자바스크립트는 non-blocking을 지원하는 event-driven 방식의 프로그래밍 언어이다. 내부적으로, 브라우저는 하나의 쓰레드만 관리하기 때문에 브라우저는 자바스크립트 코드 전체를 실행하기 위해 오직 하나의 쓰레드만 관리하며, event queue를 사용하여 listener를 등록(enqueue) 하고 event loop로 등록된 event들을 listen 한다. 싱글 쓰레드 환경에서 비동기를 지원하기 위해(CPU 자원과 웹 경험의 향상을 위해서), listener function들은 call stack이 비어있을 때만 제거(dequeue) 되며 실행된다. Promise 들은 이 event-driven architecture에 의존하여 다른 작업을 차단하지 않는 비동기 코드의 “동기식” 실행을 허락한다.

 프로그래밍적으로, Queue는 두개의 단순한 작업(unshift and pop)이 있는 배열에 불과하다. Unshift는 아이템들을 배열의 끝 에 등록(enqueue) 하는 반면, Pop은 배열의 시작 에서 제거(dequeue)한다. 즉, Queue는 “First In, First Out”(FIFO) 정책을 따른다. 만약 방향이 뒤집어지면, 우리는 unshift와 pop을 각각 push와 shift로 대치할 수 있다.

아래는 Queue 코드의 예이다.

    constructor(...items) {

        this.reverse = false;

        this.queue = [...items];

    }

    enqueue(...items) {

        return this.reverse ? this.queue.push(...items) : this.queue.unshift(...items);

    }

    dequeue() {

        return this.reverse ? this.queue.shift() : this.queue.pop();

    }

**Linked List**

 Linked List 는 배열같이 데이터들을 순차적으로 저장한다. 링크드 리스트는 인덱스를 유지하는 대신 다른 원소를 가리키는 포인터 를 가지고 있다. 첫 번째 노드는 head라고 불리는 반면 마지막 노드는 tail이라 불린다. singly-linked list 에서 각 노드는 다음노드를 가리키는 오직 하나의 포인터만 가지고 있다. 여기서 head는 우리가 리스트의 나머지들로 걸어가기 위한 시작점이다. doubly-linked list의 경우 이전(previous) 노드로 가는 포인트도 가지고 있다. 그러므로 우리는 tail에서 시작해 “뒤에서” 앞으로 갈 수 있다.

 링크드 리스트는 삽입과 삭제시 상수시간 O(1)을 가진다 왜냐하면 단순히 포인터들만 바꾸면 되기 때문이다. 배열에서 같은 작동을 하기 위해선 선형 시간 O(N)이 필요하다 왜냐면 이후의 아이템들이 한 칸씩 밀려나야 하기 때문이다. 또한, 링크드리스트는 공간(메모리)이 있는 한 넓힐 수 있다. 그러나, 자동으로 리사이징을 하는 “동적” 배열도 생각외로 비싼 값을 치뤄야 할 수 있다. 물론, 이것들은 항상 등가교환이다. 링크드 리스트의 원소를 변경하거나 찾아보자. 우리는 아마도 선형 시간으로 전체 길이를 탐색해야 할 수도 있다. 반면 인덱스 배열에서는 바로 찾아갈 수 있다.

 배열처럼, 링크드 리스트도 stack 같이 조작할 수 있다. head 를 삽입, 삭제하는 공간이 하나이므로 쉽게 구현할 수 있다. 또한 링크드리스트는 queue와 같이 작동 할 수 있다. 이는 doubly-linked list 로 이루어 질 수 있는데, tail에서 삽입이 일어나거나 head에서 삭제가 일어날 때 혹은 반대일 때 가능하다. 많은 숫자들이 존재하는 원소에서 이 구현 방법은 배열을 사용해 큐를 구현하는 것보다 더 좋은 성능을 보인다. 이유는 배열에서 shift 와 unshift 작동이 일어날 때 인덱스를 재배열 하기 위해 선형 시간이 필요하기 때문이다.

 링크드리스트는 클라이언트와 서버에서 모두 쓸모있다. 클라이언트에서 Redux 같은 상태관리 라이브러리들은 미들웨어 로직을 링크드 리스트 방식으로 구현한다. action이 디스패치 될 때, 모두 reducer에 도달할 때까지 하나의 미들웨어에서 다음 미들웨어로 연결된다. 서버의 경우 Express 같은 웹 프레임워크도 비슷한 미들웨어 로직을 가지고 있다. request 를 받았을 때, 이것들은 reponse 받을 때 까지 미들웨어를 타고다닌다.

아래는 Doubly-Linked List 의 코드 예이다.

class Node {

    constructor(*value*, *next*, *prev*) {

        this.value = *value*;

        this.next = *next*;

        this.prev = *prev*;

    }

}

class LinkedList {

    constructor() {

        this.head = null;

        this.tail = null;

    }

    addToHead(*value*) {

        const node = new Node(*value*, null, this.head);

        if (this.head) this.head.next = node;

        else this.tail = node;

        this.head = node;

    }

    addToTail(*value*) {

        const node = new Node(*value*, this.tail, null);

        if (this.tail) this.tail.prev = node;

        else this.head = node;

        this.tail = node;

    }

    removeHead() {

        if (!this.head) return null;

        const value = this.head.value;

        this.head = this.head.prev;

        if (this.head) this.head.next = null;

        else this.tail = null;

        return value;

    }

    removeTail() {

        if (!this.tail) return null;

        const value = this.tail.value;

        this.tail = this.tail.next;

        if (this.tail) this.tail.prev = null;

        else this.head = null;

        return value;

    }

    search(*value*) {

        let current = this.head;

        while (current) {

            if (current.value === *value*) return *value*;

            current = current.prev;

        }

        return null;

    }

    indexOf(*value*) {

        const indexes = [];

        let current = this.tail;

        let index = 0;

        while (current) {

            if (current.value === *value*) indexes.push(index);

            current = current.next;

            index++;

        }

        return indexes;

    }

}

**Tree**

 Tree는 많은 child nodes을 유지하고 있는 계층적 구조라는 것을 제외하곤 linked list같다. 즉, 각 노드는 한 명 이상의 부모를 가질 수 없다. Document Object Model(DOM)이 이 구조이다. root 는 html 노드이며 자식들로 head와 body노드가 있다. 이들은 친숙한 모든 html 태그 로 더욱 세분화된다. 실제로 리엑트 컴포넌트 또한 prototype 상속과 구성 의 트리 구조를 생성한다. 물론 DOM의 인메모리 표현인 리엑트의 가상DOM 또한 트리 구조이다.

 Binary Search Tree는 각 노드가 두 개의 자식 이상을 가질 수 없기 때문에 특별하다. left child의 값은 반드시 부모의 값보다 작거나 같아야한다 반면 right child는 반드시 부모의 값보다 커야한다. 이와같이 구조화되고 균형이 맞춰졌다면, 우리는 어떠한 값이든 search 시 로그 시간만에 찾을 수 있다. 이유는 각 브랜치를 순회시 작업의 반을 무시할 수 있기 때문이다. 삽입과 삭제 또한 로그시간에 가능하다. 게다가, 가장 왼쪽, 가장 오른쪽 노드가 각각 최소 혹은 최대 값이므로 값을 쉽게 찾을 수 있다.

 트리의 순회는 수직 또는 수평의 절차로 발생할 수 있다. 세로 방향의 Depth-First Traversal(DFT: DFS) 에서 재귀 알고리즘은 반복문보다 더 우아하다. 노드들은 pre-order, in-order, post-order 으로 순회된다. 만약 리프노드를 검사하기 전에 루트노드를 탐색해야 한다면, 우리는 pre-order 를 선택할 것이다. 그러나 만약 루트전에 리프노드부터 탐색해야 한다면 post-order를 선택해야 한다. in-order 는 이름이 암시하듯, 노드들을 순차적으로 순회할 수 있다. 이 속성들은 이진탐색트리를 정렬에 최적화 되게 만든다.

 수평 방향의 Breadth-First Traversal(BFT: BFS)에서 반복문은 재귀보다 더 우아하다. 이것은 각 순회에 모든 자식 노드들을 트래킹하기 위해서 queue의 사용을 요구한다. 이 큐에 필요한 메모리는 사소하지 않을 수 있다. 하지만 트리의 모양이 깊이보다 넓다면, BFT가 DFT 보다 더 나은 선택이다. 또한, BFT 가 두 노드 사이를 이동하는 경로는 가능한 가장 짧은 거리이다.

아래는 Binary Search Tree의 코드 예이다.

class Tree {

    constructor(*value*) {

        this.value = *value*;

        this.left = null;

        this.right = null;

    }

    insert(*value*) {

        if (*value* <= this.value) {

            if (!this.left) this.left = new Tree(*value*);

            else this.left.insert(*value*);

        } else {

            if (!this.right) this.right = new Tree(*value*);

            else this.right.insert(*value*);

        }

    }

    contains(*value*) {

        if (*value* === this.value) return true;

        if (*value* < this.value) {

            if (!this.left) return false;

            else return this.left.contains(*value*);

        } else {

            if (!this.right) return false;

            else return this.right.contains(*value*);

        }

    }

    depthFirstTraverse(*order*, *callback*) {

*order* === "pre" && *callback*(this.value);

        this.left && this.left.depthFirstTraverse(*order*, *callback*);

*order* === "in" && *callback*(this.value);

        this.right && this.right.depthFirstTraverse(*order*, *callback*);

*order* === "post" && *callback*(this.value);

    }

    breadthFirstTraverse(*callback*) {

        const queue = [this];

        while (queue.length) {

            const root = queue.shift();

*callback*(root.value);

            root.left && queue.push(root.left);

            root.right && queue.push(root.right);

        }

    }

    getMinValue() {

        if (this.left) return this.left.getMinValue();

        return this.value;

    }

    getMaxValue() {

        if (this.right) return this.right.getMaxValue();

        return this.value;

    }

}

mocha.setup("bdd");

const { assert } = chai;

const tree = new Tree(5);

for (const value of [3, 6, 1, 7, 8, 4, 10, 2, 9]) tree.insert(value);

*/\**

*5*

*3 6*

*1 4 7*

*2   8*

*10*

*9*

*\*/*

describe("Binary Search Tree", () => {

    it("Should implement insert", () => {

        assert.equal(tree.value, 5);

        assert.equal(tree.left.value, 3);

        assert.equal(tree.right.value, 6);

        assert.equal(tree.left.left.value, 1);

        assert.equal(tree.right.right.value, 7);

        assert.equal(tree.right.right.right.value, 8);

        assert.equal(tree.left.right.value, 4);

        assert.equal(tree.right.right.right.right.value, 10);

        assert.equal(tree.left.left.right.value, 2);

        assert.equal(tree.right.right.right.right.left.value, 9);

    });

    it("Should implement contains", () => {

        assert.equal(tree.contains(2), true);

        assert.equal(tree.contains(9), true);

        assert.equal(tree.contains(0), false);

        assert.equal(tree.contains(11), false);

    });

    it("Should implement depthFirstTraverse", () => {

        const \_pre = [];

        const \_in = [];

        const \_post = [];

        tree.depthFirstTraverse("pre", *value* => \_pre.push(*value*));

        tree.depthFirstTraverse("in", *value* => \_in.push(*value*));

        tree.depthFirstTraverse("post", *value* => \_post.push(*value*));

        assert.deepEqual(\_pre, [5, 3, 1, 2, 4, 6, 7, 8, 10, 9]);

        assert.deepEqual(\_in, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

        assert.deepEqual(\_post, [2, 1, 4, 3, 9, 10, 8, 7, 6, 5]);

    });

    it("Should implement breadthDepthTraverse", () => {

        const result = [];

        tree.breadthFirstTraverse(*value* => result.push(*value*));

        assert.deepEqual(result, [5, 3, 6, 1, 4, 7, 2, 8, 10, 9]);

    });

    it("Should implement getMinValue", () => {

        assert.equal(tree.getMinValue(), 1);

    });

    it("Should implement getMaxValue", () => {

        assert.equal(tree.getMaxValue(), 10);

    });

});

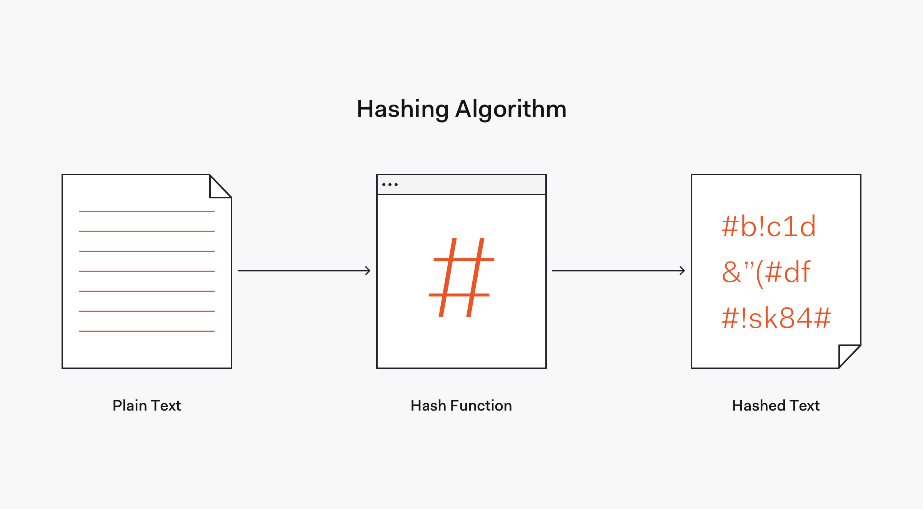
**Graph**

그래프는 상하위의 개념이 없이 각각의 node와 그 node들 간의 간선(edge)을 하나로 모아 놓은 자료구조이다.  만약 트리의 부모가 하나보다 더 많이 가질 수 있다면, 이것이 바로 Graph 이다. 그래프에서 노드들을 이어주는 Edge는 노드들이 방향, 무방향, 가중치, 비가중치 를 가지게 한다. Edge 들은 vector와 유사한 방향과 가중치를 가진다. 그래프는 꼭 모든 node들이 서로 관계를 갖고 있어야만 하지 않다. 따라서 그래프는 node간 연결이 없는 고립된 부분이 있을 수도 있고, 순환할 수도 있고, 안할 수도 있기 때문에 가장 형식에 얽매이지 않은 자료구조라고 볼 수 있다.

새로운 Graph {  
 그래프 생성자{ 비어있는 노드 리스트 }  
   
 노드 생성자 {  
 노드 이름;  
 노드와 연결된 노드 리스트[]; //edge  
 } 노드 입력 {  
 data = new 노드;  
 while(edge수){  
 연결되는 노드에도 edge 추가  
 }  
 return graph;  
 }  
}

**Hash**

해시함수를 이용하여 data를 저장하는 자료구조이고,해시함수(hash function)란 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수입니다. 이 때 매핑 전 원래 데이터의 값을 키(key), 매핑 후 데이터의 값을 해시값(hash value), 매핑하는 과정 자체를 해싱(hashing)라고 합니다.



해시함수는 해쉬값의 개수보다 대개 많은 키값을 해쉬값으로 변환(many-to-one 대응)하기 때문에 해시함수가 서로 다른 두 개의 키에 대해 동일한 해시값을 내는 해시충돌(collision)이 발생하게 됩니다. 해시충돌이 발생할 가능성이 있음에도 해시테이블을 쓰는 이유는 적은 리소스로 많은 데이터를 효율적으로 관리하기 위해서입니다. 예컨대 해시함수로 하드디스크나 클라우드에 존재하는 무한에 가까운 데이터(키)들을 유한한 개수의 해시값으로 매핑함으로써 작은 크기의 캐쉬 메모리로도 프로세스를 관리할 수 있게 됩니다.

새로운 hash table {  
 table 생성자 {} hashing 함수 (data) {  
 암호화, 변환 // 매우 다양  
 } data 입력 (data) {  
 table = hashing(data);  
 }  
}