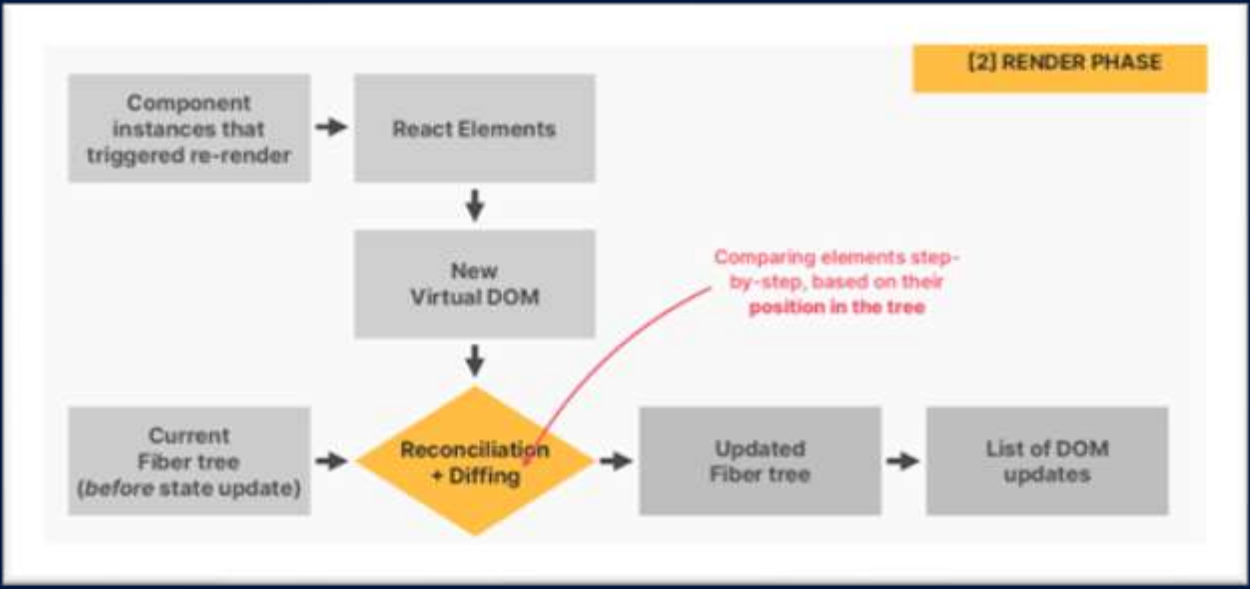
React

DIFFING



[2] RENDER PHASE

Component instances that triggered re-render

React Elements

New Virtual DOM

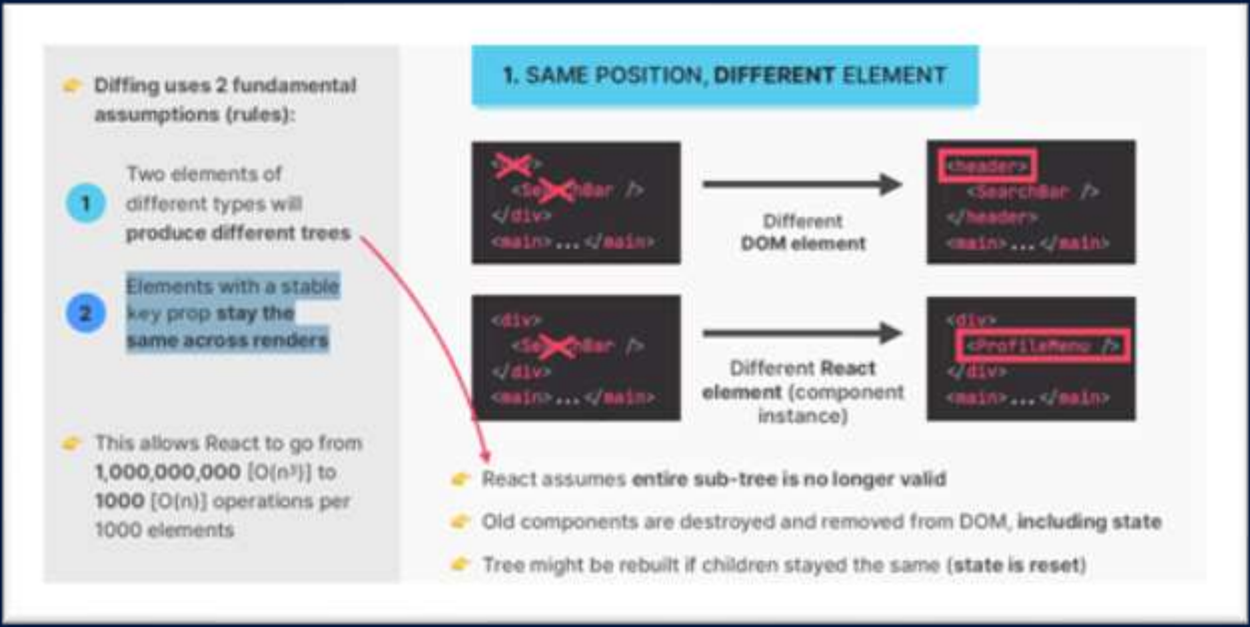
Current Fiber tree (*before* state update)

Reconciliation + Diffing

Updated Fiber tree

List of DOM updates

Comparing elements step-by-step, based on their **position in the tree**



- **Diffing uses 2 fundamental assumptions (rules):**

1. Two elements of different types will **produce different trees**
2. Elements with a stable key prop **stay the same across renders**

- This allows React to go from **1,000,000,000** [O(n³)] to **1000** [O(n)] operations per 1000 elements

1. SAME POSITION, **DIFFERENT** ELEMENT

```
<div>
  <SearchBar />
</div>
<main>...</main>
```

Different **DOM element**

```
<header>
  <SearchBar />
</header>
<main>...</main>
```

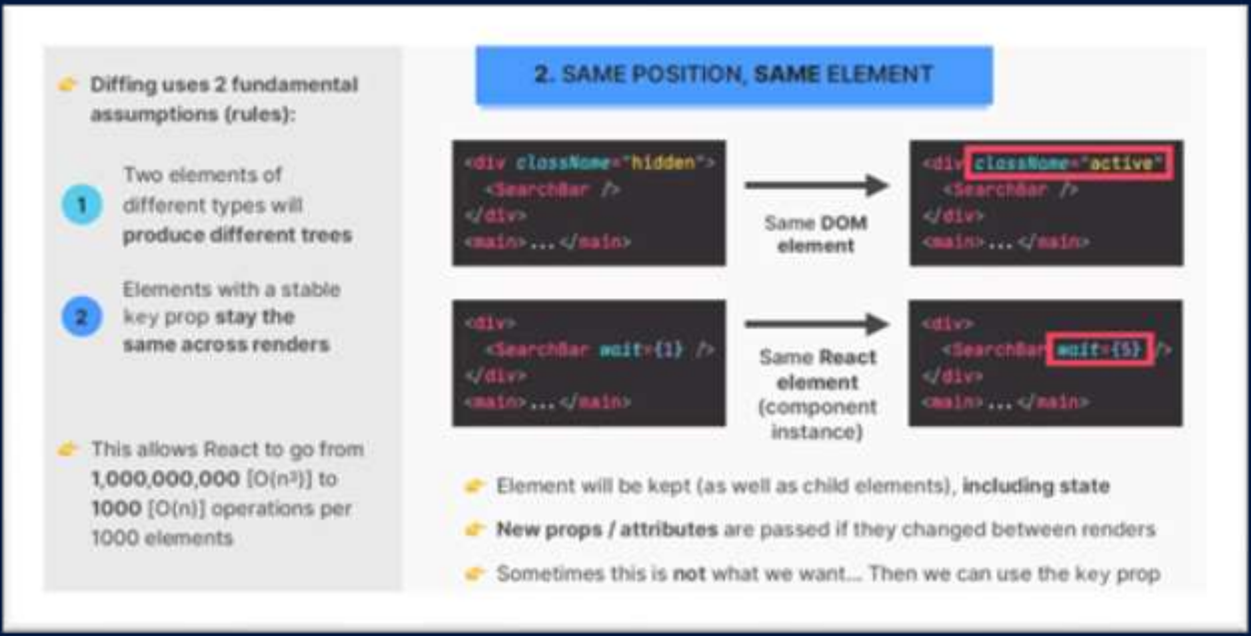
```
<div>
  <SearchBar />
</div>
<main>...</main>
```

Different **React element** (component instance)

```
<div>
  <ProfileMenu />
</div>
<main>...</main>
```

- React assumes **entire sub-tree is no longer valid**
- Old components are destroyed and removed from DOM, **including state**
- Tree might be rebuilt if children stayed the same (**state is reset**)

React



Diffing uses 2 fundamental assumptions (rules):

1. Two elements of different types will **produce different trees**
2. Elements with a stable key prop **stay the same across renders**

This allows React to go from **1,000,000,000** [O(n³)] to **1000** [O(n)] operations per 1000 elements

2. SAME POSITION, SAME ELEMENT

```
<div className="hidden">
  <SearchBar />
</div>
<main>...</main>
```

Same **DOM element**

```
<div className="active">
  <SearchBar />
</div>
<main>...</main>
```

```
<div>
  <SearchBar wait={1} />
</div>
<main>...</main>
```

Same **React element** (component instance)

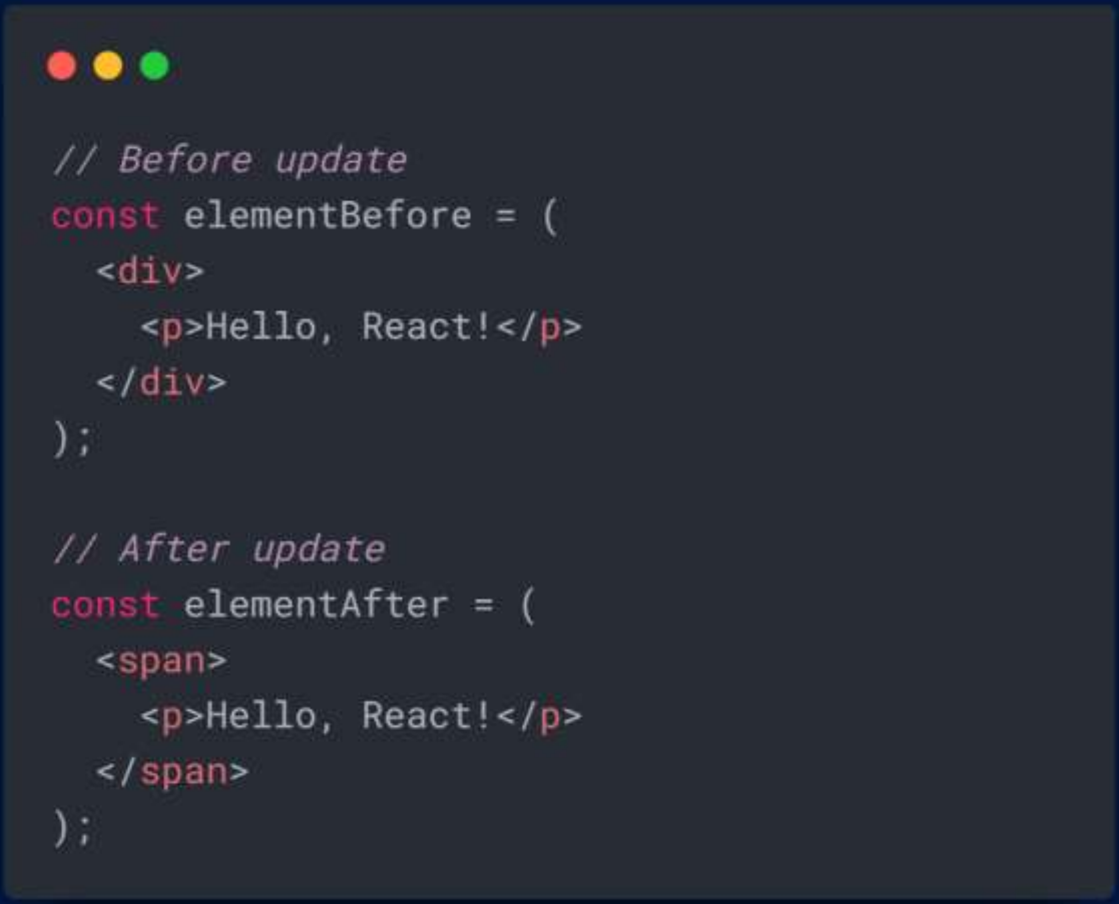
```
<div>
  <SearchBar wait={5} />
</div>
<main>...</main>
```

- Element will be kept (as well as child elements), **including state**
- **New props / attributes** are passed if they changed between renders
- Sometimes this is **not** what we want... Then we can use the key prop

React

TWO ELEMENTS OF DIFFERENT TYPES WILL PRODUCE DIFFERENT TREES

In React, when comparing two elements of different types during the reconciliation process, React assumes that the content of the elements has completely changed, and it treats them as completely different subtrees. This is a fundamental principle in React's diffing algorithm and is often referred to as the "reconciliation assumption."



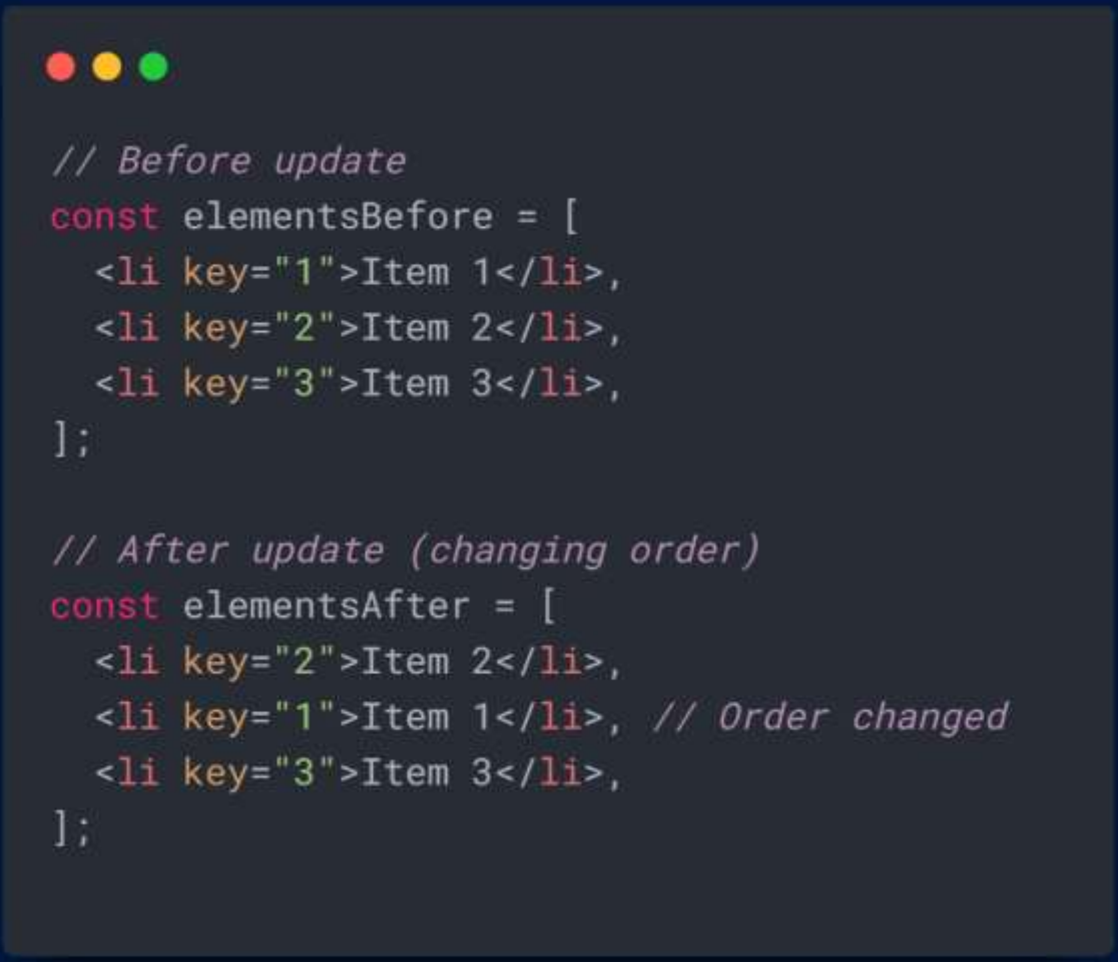
```
// Before update
const elementBefore = (
  <div>
    <p>Hello, React!</p>
  </div>
);

// After update
const elementAfter = (
  <span>
    <p>Hello, React!</p>
  </span>
);
```

In this example, elementBefore and elementAfter have different root types (div vs. span). When React reconciles these elements, it will consider the entire subtree to be different, even though the only change is the root element type. React will unmount the old div and mount the new span, resulting in the DOM being completely replaced.

This approach helps React ensure that it doesn't make assumptions about how different types of elements should be updated. Treating them as completely different subtrees provides a clear and predictable behavior, even if it may seem less efficient for small changes in the structure of the elements.

ELEMENTS WITH A STABLE KEY PROP STAY THE SAME ACROSS THE RENDERS



```
// Before update
const elementsBefore = [
  <li key="1">Item 1</li>,
  <li key="2">Item 2</li>,
  <li key="3">Item 3</li>,
];

// After update (changing order)
const elementsAfter = [
  <li key="2">Item 2</li>,
  <li key="1">Item 1</li>, // Order changed
  <li key="3">Item 3</li>,
];
```

React

When elements have a stable and unique key prop, React uses the key to optimize the update process during reconciliation. The key prop helps React identify which elements in the new set correspond to which elements in the previous set, even when the order of elements changes. When React finds elements with matching keys, it assumes that the elements represent the same underlying data and attempts to update them rather than unmounting and remounting.

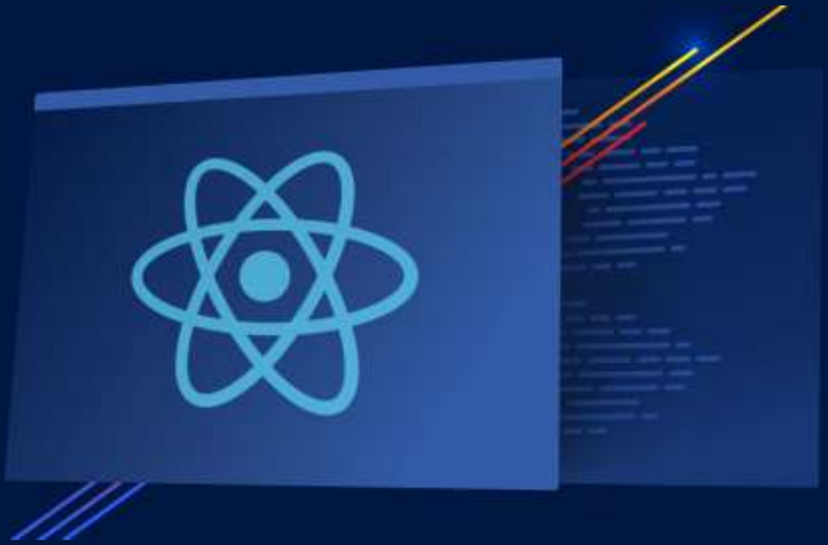
In this example, each li element has a unique and stable key prop. When React reconciles the before and after sets of elements, it uses the key prop to determine that the first and second elements have swapped places. Instead of unmounting and remounting the elements, React updates their positions efficiently.

Using stable keys is important when dealing with dynamic lists of elements. It helps React correctly identify and update elements as the order changes, leading to more efficient rendering and a better user experience.

React

Do you find it helpful?

Let me know down in the comments



Click To Follow For More On LinkedIN