

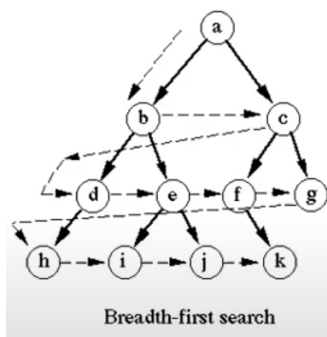
Objective :

To scrape all relative links that are stored within each and every page respectfully within the queue whilst aiming for $O(n)$ time and space complexity.

Afterwhich, an api shall be set and a client ui meant for server request and response display.

Theory:

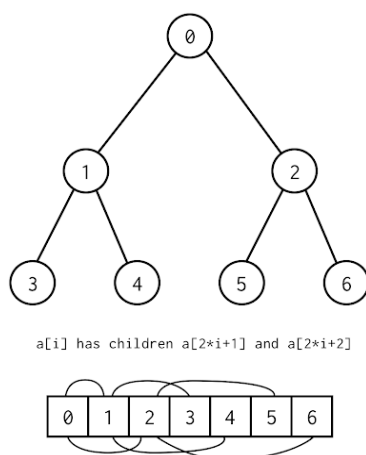
BFS(Breadth-First-Search), in breadth-first search data is accessed by order of queueing (FIFO).



As you can see in bfs the initial node will always be a and nodes are accessed only by order of insertion to the queue.

Implementation :

In JS BFS is done like so:



The best way to implement it is using a queue array and a usedNodes(array or hashmap), first execution of the function adds the inputted node to usedNodes. On every execution a node is pushed to the end of the array, and an element which is stored in the first index of the array gets executed and is removed from the queue array. Using this search algorithm rewards us with many performance benefits

Results:

https://en.wikipedia.org/wik

100000

600

CRAWL

Got 100000 links in 346.0599 seconds

```
"root" : 100000 items
  > [ 0 - 100 ]
  > [ 100 - 200 ]
  > [ 200 - 300 ]
  > [ 300 - 400 ]
  > [ 400 - 500 ]
  > [ 500 - 600 ]
  > [ 600 - 700 ]
  > [ 700 - 800 ]
  > [ 800 - 900 ]
  > [ 900 - 1000 ]
  > [ 1000 - 1100 ]
  > [ 1100 - 1200 ]
  > [ 1200 - 1300 ]
  > [ 1300 - 1400 ]
  > [ 1400 - 1500 ]
  > [ 1500 - 1600 ]
  > [ 1600 - 1700 ]
  > [ 1700 - 1800 ]
  > [ 1800 - 1900 ]
  > [ 1900 - 2000 ]
  > [ 2000 - 2100 ]
  > [ 2100 - 2200 ]
  > [ 2200 - 2300 ]
  > [ 2300 - 2400 ]
  > [ 2400 - 2500 ]
  > [ 2500 - 2600 ]
  > [ 2600 - 2700 ]
  > [ 2700 - 2800 ]
  > [ 2800 - 2900 ]
  > [ 2900 - 3000 ]
  > [ 3000 - 3100 ]
  > [ 3100 - 3200 ]
  > [ 3200 - 3300 ]
  > [ 3300 - 3400 ]
  > [ 3400 - 3500 ]
  > [ 3500 - 3600 ]
  > [ 3600 - 3700 ]
  > [ 3700 - 3800 ]
  > [ 3800 - 3900 ]
  > [ 3900 - 4000 ]
  > [ 4000 - 4100 ]
  > [ 4100 - 4200 ]
  > [ 4200 - 4300 ]
  > [ 4300 - 4400 ]
  > [ 4400 - 4500 ]
  > [ 4500 - 4600 ]
  > [ 4600 - 4700 ]
  > [ 4700 - 4800 ]
  > [ 4800 - 4900 ]
  > [ 4900 - 5000 ]
  > [ 5000 - 5100 ]
  > [ 5100 - 5200 ]
  > [ 5200 - 5300 ]
  > [ 5300 - 5400 ]
  > [ 5400 - 5500 ]
  > [ 5500 - 5600 ]
  > [ 5600 - 5700 ]
  > [ 5700 - 5800 ]
  > [ 5800 - 5900 ]
  > [ 5900 - 6000 ]
  > [ 6000 - 6100 ]
  > [ 6100 - 6200 ]
  > [ 6200 - 6300 ]
  > [ 6300 - 6400 ]
  > [ 6400 - 6500 ]
  > [ 6500 - 6600 ]
  > [ 6600 - 6700 ]
  > [ 6700 - 6800 ]
  > [ 6800 - 6900 ]
  > [ 6900 - 7000 ]
  > [ 7000 - 7100 ]
```

Performance Update:

Tried implementing hash maps in order to avoid this $O(N)$ iteration :

```
for (let queuedLink of queue) {
  if (queuedLink === relaLink) {
    isQueued = true;
    break;
  }
}
```

But, because the [0]index of the queue map still needs to be sent.

There is no hiding from Object.Keys which is as complex as iterating on the arrays.

I'd tried it for testing anyway, to my expectations I was correct

```
let queueMap = {};
let seenSitesMap = {};
const crawl = async (url, linksLen, depthLen) => {
  try {
    seenSitesMap[url] = true;

    const seenSites = Object.keys(seenSitesMap);
    if (seenSites.length > depthLen) return;

    const htmlData = await rp(url);
    const soup = new JSSoup(htmlData);
    const relaLinks = soup
      .findAll('a')
      .map((anchor) => {
        anchor.attrs.href &&
        anchor.attrs.href[0] === '/' &&
        !anchor.attrs.href.includes('.')
        ? url.match(/^.+?[\w\/:](?=[\w\/]|$)/)[0] + anchor.attrs.href
        : anchor.attrs.href
      })
      .filter(
        (link) =>
          link &&
          link.includes(url.match(/^.+?[\w\/:](?=[\w\/]|$)/)[0])
      );

    for (let relaLink of relaLinks) {
      let isQueued = queueMap[url];

      if (!isQueued) {
        const queue = Object.keys(queueMap);
        if (queue.length >= linksLen) return;
        if (!seenSites[relaLink]) queueMap[relaLink] = true;
        if (queue.length % 500 === 0)
          console.log('Relative Links Scraped:', queue.length);
      }
    }

    const queuedLink = Object.keys({ ...queueMap })[0];
    delete queueMap[queuedLink];

    return crawl(queuedLink, linksLen, depthLen);
  } catch (err) {
    console.log(err);
  }

  const queuedLink = Object.keys({ ...queueMap })[0];
  delete queueMap[queuedLink];

  return crawl(queuedLink, linksLen, depthLen);
};
```

Much slower using this solution reaches in comparison to iterating

-Second Update:

I was still bugged from this iterative loop

```
for (let queuedLink of queue) {  
  if (queuedLink === relaLink) {  
    isQueued = true;  
    break;  
  }  
}
```

Mainly because I want the queue size to be over 1M, and when it reaches high numbers iterating over it will take forever. Last time I'd used hashmaps but trying to obtain the keys is still as complex as iterating on the queue.

But I had tried a different approach, hashMaps are terrific for validation but seenSites array relatively is tiny so I thought of this:

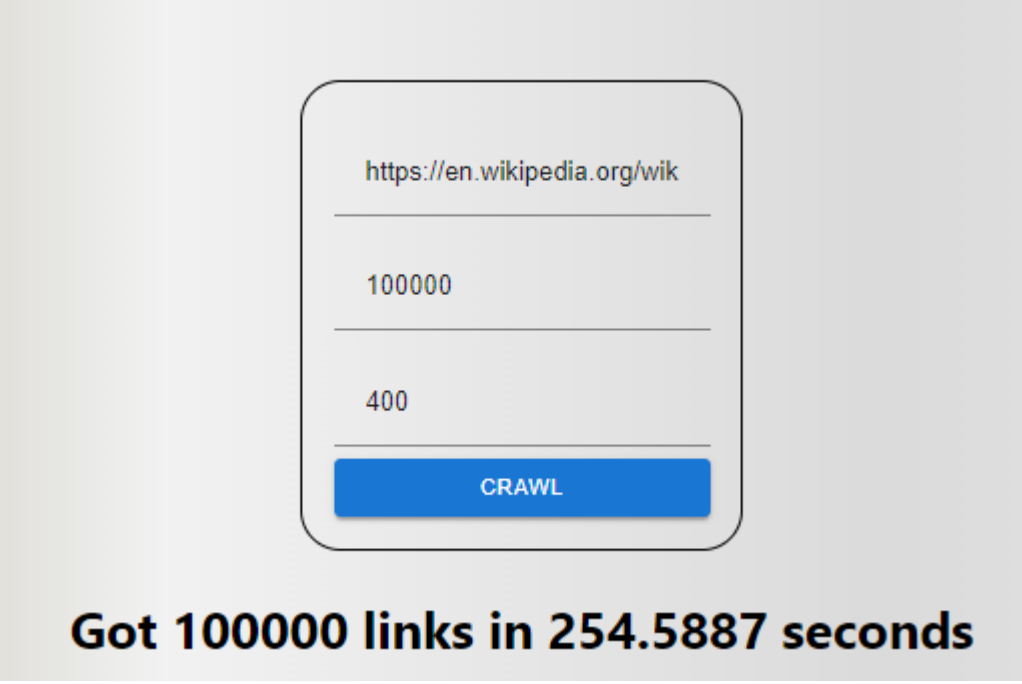
```
let queue = [];  
let queueMap = {};  
let seenSites = [];
```

The same queue is still used but a queue hash is used as well now each push to the queue although seen sites will be still stored inside the map the queue pushing is still conditional to seenSites duplicates ergo, making all conditions valid thus, the aforementioned $O(n)$ for loop is now obsolete and it's replaced with $O(1)$ hashMap.

Below, the new performant hash validated for loop :

```
for (let relaLink of relaLinks) {  
  if (!queueMap[relaLink]) {  
    if (queue.length ≥ linksLen) return;  
  
    if (!seenSites.includes(relaLink)) {  
      queue.push(relaLink);  
      queueMap[relaLink] = true;  
    }  
  
    if (queue.length % 500 === 0)  
      console.log('Relative Links Scraped:', queue.length);  
  }  
}
```

A test had been conducted using the same parameters at the fetch in page 2



https://en.wikipedia.org/wik

100000

400

CRAWL

Got 100000 links in 254.5887 seconds

That's 25% less time than the same query used to take with the solution which was implemented earlier .