a.What is the main functionality of the browser?

Browser is software made of a structured code that acts as an interface to locate, retrieve as well as display multimedia information from a particular web-server

b.High Level Components of a browser.

There can be six major High-Level components of a browser

1. User Interface
2. Browser Engine
3. Rendering Engine
4. TCP/IP Networking Protocol Handler
5. JS Interpreter
6. User Interface Backend
7. Storage Support

c. Rendering engine and its use. e.Script Processors;

Rendering Engine is mainly responsible for parsing the HTML/XML content from the network layer and convert it to Content or DOM Tree where each entity is referred to as a Node, it also makes a Render tree which is a graphical representation of the whole document structure. The renderer doesn’t have any specific place or size on screen so they implement the “layout()” or “reflow()” method to achieve this. The next step is called painting where the render tree is read and page is displayed using the UI backend layer by implementing the paint() method.

d.Parsers (HTML, CSS, etc);

Parsers can be understood as readers who analyze the document; They do it in two steps, first is called Lexical analysis aka tokenization which can be simply understood as breaking the words of a page, and the second is called parser which makes meaningful sentences of the previously broken words as per their Grammer=Syntax. The compiler first parses it into a parse tree and then finally translates the tree into a machine code document.

When it comes to HTML, unlike conventional parsers, Here they use an enhanced version called as Document-Type-Definition format which has the capability of parsing any HTML code as it omits lots of tags and other data and that is what makes designing this parser more difficult. Also, in HTML, dynamic code using scripts & containing document.write() calls constantly adds extra tokens, so the parsing process actually modifies the input, so the process of tokenization is followed by tree construction. Let’s understand it by simple code:

<html>

<body>

Hello world

</body>

</html>

The initial state is the "Data state". When the < character is encountered, the state is changed to "Tag open state". Consuming an a-z character causes creation of a "Start tag token", the state is changed to "Tag name state". We stay in this state until the > character is consumed. Each character is appended to the new token name. In our case the created token is an html token.

When the > tag is reached, the current token is emitted and the state changes back to the "Data state". The <body> tag will be treated by the same steps. So far the html and body tags were emitted. We are now back at the "Data state". Consuming the H character of Hello world will cause creation and emitting of a character token, this goes on until the < of </body> is reached. We will emit a character token for each character of Hello world.

We are now back at the "Tag open state". Consuming the next input / will cause creation of an end tag token and a move to the "Tag name state". Again we stay in this state until we reach >.Then the new tag token will be emitted and we go back to the "Data state". The </html> input will be treated like the previous case.

f. Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The first mode is the **"initial mode"**. Receiving the "html" token will cause a move to the **"before html"** mode and a reprocessing of the token in that mode. This will cause creation of the HTMLHtmlElement element, which will be appended to the root Document object.

The state will be changed to **"before head"**. The "body" token is then received. An HTMLHeadElement will be created implicitly although we don't have a "head" token and it will be added to the tree.

We now move to the **"in head"** mode and then to **"after head"**. The body token is reprocessed, an HTMLBodyElement is created and inserted and the mode is transferred to **"in body"**.

The character tokens of the "Hello world" string are now received. The first one will cause creation and insertion of a "Text" node and the other characters will be appended to that node.

The receiving of the body end token will cause a transfer to **"after body"** mode. We will now receive the html end tag which will move us to **"after after body"** mode. Receiving the end of file token will end the parsing.

At this stage the browser will mark the document as interactive and start parsing scripts that are in "deferred" mode: those that should be executed after the document is parsed. The document state will be then set to "complete" and a "load" event will be fired.

g.Order of script processing

If we aren't dynamically loading scripts or marking them as defer or async, then scripts are loaded in the order encountered in the page. It doesn't matter whether it's an external script or an inline script - they are executed in the order they are encountered in the page. Inline scripts that come after external scripts are held until all external scripts that came before them have loaded and run. Async scripts (regardless of how they are specified as async) load and run in an unpredictable order. For classic scripts, if the async attribute is present, then the classic script will be fetched in parallel to parsing and evaluated as soon as it is available.

For module scripts, if the async attribute is present then the scripts and all their dependencies will be executed in the defer queue, therefore they will get fetched in parallel to parsing and evaluated as soon as they are available. This attribute allows the elimination of **parser-blocking JavaScript** where the browser would have to load and evaluate scripts before continuing to parse. defer has a similar effect in this case.

If one needed a predictable order, then it would have to be coded in by registering for load notifications from the async scripts and manually sequencing javascript calls when the appropriate things are loaded.

A script tag with defer waits until the entire parser is done and then runs all scripts marked with defer in the order they were encountered. This allows you to mark several scripts that depend upon one another as defer. They will all get postponed until after the document parser is done, but they will execute in the order they were encountered preserving their dependencies. I think of defer like the scripts are dropped into a queue that will be processed after the parser is done. Technically, the browser may be downloading the scripts in the background at any time, but they won't execute or block the parser until after the parser is done parsing the page and parsing and running any inline scripts that are not marked defer or async.

h.Layout and Painting

The layout usually has the following pattern:

1. Parent renderer determines its own width.
2. Parent goes over children and:
   * Place the child renderer (sets its x and y).
   * Calls child layout if needed–they are dirty or we are in a global layout, or for some other reason–which calculates the child's height.
3. Parent uses children's accumulative heights and the heights of margins and padding to set its own height–this will be used by the parent renderer's parent.
4. Sets its dirty bit to false.

In the painting stage, the render tree is traversed and the renderer's "paint()" method is called to display content on the screen. Painting uses the UI infrastructure component.

This is actually the order in which the elements are stacked in the stacking contexts. This order affects painting since the stacks are painted from back to front. The stacking order of a block renderer is:

1. background color
2. background image
3. border
4. children
5. outline