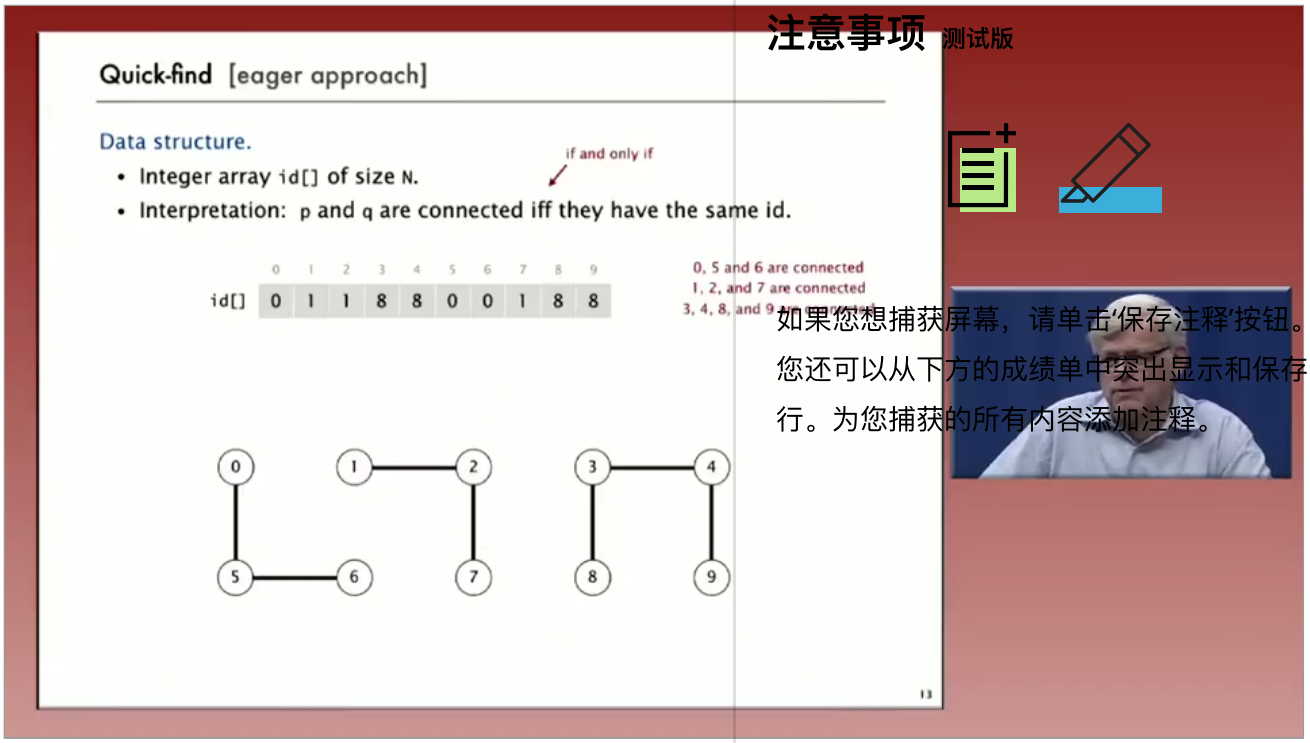
Quick Find



英语（English）

帮助我们翻译

0:01 Now we'll look at our first implementation of an algorithm for solving the dynamic connectivity problem, called Quick-find. This is a so called eager algorithm, for solving kind activity problem. The data structure that we're going to use to support the algorithm is simply an integer array indexed by object. The interpretation is the two objects, P and Q are connected if and only if, their entries in the array are the same. So for example in this example with our ten objects the idea array that describes the situation after seven connections is illustrated in the middle of the slide. So that, after the, at this point zero, five, and six are all in the same connected component, because they have the same array entry, zero. One, two, and seven all have entry one. And three, four, eight, and nine all have entry eight. So that representation is, shows that they're connected. And clearly, that's going to support a quick implementation of the find operation. We just check the array entries to see if they're equal. Check if P and Q have the same ID. So, six and one have different IDs. One has ID one, six has ID zero. They're not in the same connected component. Union is more difficult in order to merge the components, containing two given objects. We have to change all the entries, whose ID is equal to one of them to the other one. And arbitrarily we choose to change the ones that are the same as P to the ones that are same as Q. So if we're going to union six and one, then we have to change entries zero, five, and six. Everybody in the same connected component as six. From zero

to one. And this is, as we'll see, this is a bit of a problem when we have a huge number of objects, because there's a lot of values that can change. But still, it's easy to implement, so that'll be our starting point. So we'll start with a, a demo of how this works. So, initially, we set up the ID array, with each entry, equal to its index. And so all that says is that all the objects are independent. They're in their own connected component. Now, when we get a union operation. So, say, four is supposed to be u[illegible] then we're going to change, all entries, whose ID is equal to the first ID to the second one. So in this case, we'll change the, connect three and four means that we need to change the four to a three. And we'll continue to do a few more so you'll get an idea of how it works. So three and eight now so to connect three and eight now three and four have to be connected to eight. So both of those entries have to change to eight. Okay? So now, what about six and five? So again, we change the first one to match the second one. So to connect six and five, we change the six to a five. What about nine and four? So, now we have to change the, to connect nine and four, we have to change, 9's entry to be the same as 4's. So now we have three, four, eight, and nine. All have entries eight. They're all on the same connected component. Two and one means that we connect two and one by changing the 2201. Eight and nine are already connected. They have the same, entries in the idea array. So, that connected query, that find says, true, they're already connected. And five and zero have different entries. They're not connected, so we'd return false, in that case, not connected. And then, if we want to connect five and zero. Then, as usual we'll connect, the entry corresponding to both five and six to zero. Seven and two, union seven and two. That's an easy one. And union, six and one so there is three entries that have to get changed. All those zeros have to get changed to ones. So, that's a quick demo of Quick-find. Now next we'll look at the code for implementating that. Okay, with this concrete demo in mind then moving to coding up this algorithim is pretty straight forward. Although it's an interesting programming exercise that a lot of us would get wrong the first time. So let's start with the constructor, well we have a, a private integer array. That's our ID array. That's the data structure that's going to support this implementation. The constructor has to create the array and then go through and set the value corresponding to each index I to I. That's straight forward. The find operation, or connected operation. That's the easy one . This is the Quick-find algorithm. So it simply takes its two arguments, P and Q, and checks whether their ID entries are equal, and returns that value. If they're equal, it returns true. If they're not equal, it returns false. The more complicated operation implement is a union. And there, we find first the ID corresponding with the first argument, and then the ID corresponding to the second argument. And then we go through the whole array, and looking for the entries whose IDs are equal to the ID of the first argument, and set those to the ID of the second argument. That's a pretty straightforward implementation. And I mentioned that a lot of us would get us wrong. The mistake we might make is to put ID of P here rather than first picking out, that value. And you can think about the implications of that. That's an insidious bug. So, that's a fine

implementation of QuickFind so the next thing to decide is how effective or efficient that algorithm is going to be and we'll talk in some detail about how to do that but for this it's sufficient to just think about the number of times the code has to access the array. As we saw when doing the implementation, both the initialized and union operations involved the for-loop that go through the entire array. So they have to touch in a constant proportional to n times after touching array entry in a constant time, it's just to a constant number of times check array entries. And this is problematic because the union operation is too expensive. In particular if you just have N union commands on N objects which is not unreasonable. They're either connected or not then it will take quadratic time in squared time. And one of the themes that we'll go through over and over in this course is that quadratic time is much to slow. And we can't accept quadratic time algorithms for large problems. The reason is they don't scale. As computers get faster and bigger, quadratic algorithms actually get slower. Now, let's just talk roughly about what I mean by that. A very rough standard, say for now, is that people have computers that can run billions of operations per second, and they have billions of entries in main memory. So, that means that you could touch everything in the main memory in about a second. That's kind of an amazing fact that this rough standard is really held for 50 or 60 years. The computers get bigger but they get faster so to touch everything in the memory is going to take a few seconds. Now it's true when computers only have a few thousand words of memory and it's true now that they have billions or more. So let's accept that as what computers are like. Now, that means is that, with that huge memory, we can address huge problems. So we could have, billions of objects, and hope to do billions of union commands on them. And, but the problem with that quick find algorithm is that, that would take ten^18th operations, or, say array axises or touching memory. And if you do the math, that works out to 30 some years of computer time. Obviously, not practical to address such a problem on today's computer. And, and the reason is, and the problem is that quadratic algorithms don't scale with technology. You might have a new computer that's ten times as fast but you could address a problem that's ten times as big. And with a quadratic algorithm when you do that. It's going to be ten times as slow. That's the kind of situation we're going to try to avoid by developing more efficient algorithms for solving problems like this.