[Linux Term Project Presentation]

[1]

Hello, this is team 7.

Our term project title is squared root list, fast random access linked list with a square root decomposition technique.

[2]

First, we will start with a brief introduction and explain our project goal.

Then we will explain and show the implementation of our project and performance analysis of test results.

[3]

First, a linked list is a popular data structure not only in the Linux kernel but any other programming language.

A linked list is widely used throughout the Linux kernel code, in process management, file system, networking, and many other parts.

So, its performance can highly affect the overall system performance.

[4]

An array is another basic data structure similar to a linked list.

Both store elements sequentially.

In different situations, different data structures are preferred.

If random access occurs mostly, using an array is better than a linked list.
And if insertion and deletion occur mostly, using a linked list is better than a linked list.
Why? Below is the time complexity table of each operation of two data structures.
As you see, sequential access has constant time complexity in both data structures.
But the linked list has O(N) time complexity when operating random access, and the array has O(N) time complexity when operating insertion and deletion.
In this situation, which data structure should we choose when random access, insertion, and deletion all occur at a similar number of times?

[5]
In such situations, we need a data structure in which all operations operate at a moderately high speed.

[6]
So, our project goal is to improve the performance of the original linked list.
We enhanced the random access speed, reading the i-th element of the list from linear time complexity O(N) to O(squared root of N).
We kept the sequential access speed as constant time.
But, we sacrificed a little bit of insertion and deletion speed from constant time to O(squared root of N), which is the same as random access speed.
So, how do we achieve this?

[7]

Our team solved this problem with the square root decomposition technique, and we named this new data structure SQRT List.

[8]

Now, we will explain our implementation of the squared root list.

First, we need the concept of a multi-level list.

We divided the elements into something called buckets.

We use buckets to increase random access speed.

[9]

Using the concept of buckets, we divide elements into buckets, in sequence.

When dividing elements, we use the square root decomposition technique.

This is not a complex concept.

When there are N elements, we just simply divide elements into the squared root of N and place each divided group into buckets.

Then, in each bucket, there will be a squared root of N elements.

So, the bucket count, which is the number of buckets, and the bucket size, which is the number of elements in a single bucket, will be the same as the squared root of N.

This squared root of N is an optimal number of bucket count for a random-access operation.

And after dividing elements into buckets, each bucket has a direct link to the previous and next buckets.

Then, the buckets will form a linked list.

The below picture shows the concept of a bucket when there are 9 elements.

[10]

Then, in a random-access situation, how do we find the i-th element?

First, find the correct bucket, which contains the i-th element.

There are only squared roots of N buckets to traverse, so the time complexity of finding the bucket is O(squared root of N).

[11]

Next, in the selected bucket, find the element.

Also, there are only squared roots of N elements to traverse in a single bucket, so the time complexity of finding the element is O(squared root of N).

So the overall time complexity will be O(squared root of N).

[12]

But in the case of insertion operation, there are several problems.

The first problem is, if we keep inserting elements into only one bucket, the size between buckets can become unbalanced.

These unbalanced buckets degrade the random-access performance.

So we need a self-balancing bucket.

[13]

To implement a self-balancing bucket, we need just 2 steps.

First, right after the insertion, find the minimum size bucket, which takes O(squared root of N).

Second, From the bucket of an inserted element to the minimum size bucket, move one element of each bucket in the direction towards the minimum size bucket.
This process also takes O(squared root of N).
After that, the size between buckets can differ by at most 1, always.
Through this procedure, the buckets will be always balanced, which will make a self-balancing bucket.

[14]
The second problem is, if we keep inserting elements, the size of a bucket gets bigger and bigger.
Although the buckets are balanced, this problem also degrades random access performance.
So, we need to add a new bucket to reduce the size of each bucket.
To solve this problem, we can simply rebuild the buckets.
Remove all old buckets and create a new squared root of N buckets from scratch (from the initial state).
This procedure costs O(N) time.
But, we don't need to rebuild all buckets every time.
We can rebuild buckets when the number of elements N becomes about 1, 4, 9, and so on, which are square numbers because the integer part value of the squared root of N changes only at those moments.
So, O(N) cost operation occurs once per squared root of N times, and the amortized time complexity can be calculated as O(squared root of N).
This is similar to the technique used in C++'s vector and Java's ArrayList.

[15]

Next, the deletion operation is the inverse of insertion.

The self-balancing algorithm is the same except for the order, moving one element of each bucket from the maximum size bucket to the bucket with a deleted element.

The rebuilding algorithm is similar, with some slightly different conditions, but these two algorithms keep showing O(squared root of N) time complexities.

So, the time complexity of each operation can be written as below, all showing O(squared root of N).

[16]

Finally, we had a performance test on our implemented squared root list.

For comparison, we conducted the same test for the Linux data structures, XArray and List.

Each operation, random access, sequential access, insertion, and deletion was repeated 20 thousand times.

[17]

And this is the test result.

For the random-access operation, the squared root list had 57 times faster execution time than the Linux list.

But as you can see, the increased time complexity of the insertion and deletion operation is also shown in the result.

Additionally, the XArray had terrible execution time in insertion and deletion operations, which has a time complexity of O(N).

So, in total, when each operation occur a similar number of times, our implemented squared root list had 36 times faster execution time than the Linux list.
In such a situation, we can conclude that we can get better performance in systems or applications using this squared root list.