주영석 / 나종우 / 오혜성 / 한신

SQRT List

Fast random access linked list with
square root decomposition technique

Index

- Introduction
- Project Goal
- Implementation
- Performance Test

1. Introduction

❖ Linked List

- It is widely used throughout the Linux kernel code

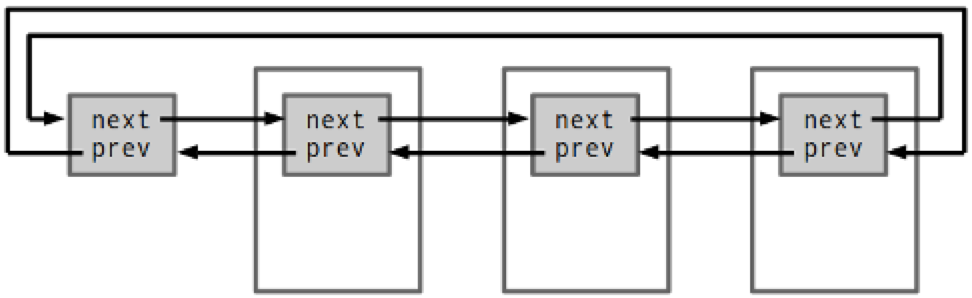
  Process Management (task_struct)

  File System (super_block, inode)

  Networking (tcp_sock)

  And many things…

- Its performance can highly affect the overall system performance



1. Introduction

Array vs Linked List

- Use an array if **random access** occurs mostly
- Use a linked list if **insertion** and **deletion** occur mostly
- But when **random access**, **insertion**, and **deletion** all occur at a similar number of times, which data structure should we choose?

Operation	Array	Linked List
Sequential Access	O(1)	O(1)
Random Access	O(1)	O(N)
Insertion	O(N)	O(1)
Deletion	O(N)	O(1)

1. Introduction

❖ Array vs Linked List

- Use an array if **random access** occurs mostly
- Use a linked list if **insertion** and **deletion** occur mostly
- But when **random access**, **insertion**, and **deletion** all occur at a similar number of times, which data structure should we choose?

We need a data structure in which all operations operate at a **moderately high speed**.

2. Project Goal

- **Faster random access speed of linked list**

  a.k.a. **Read *i*th element of the list**

  Linked List: **O(N)**

  → SQRT List: **O($\sqrt{N}$)**

- **Keep sequential access speed as is**

  Linked List: **O(1)**

  → SQRT List: **O(1)**

- **Sacrifice a little bit of insertion & deletion speed**

  Linked List: **O(1)**

  → SQRT List: **O($\sqrt{N}$)**

2. Project Goal

- **Faster random access speed of linked list**

  a.k.a. **Read *i*th element of the list**

  Linked List: **O(N)**

  → SQRT List: **O($\sqrt{N}$)**

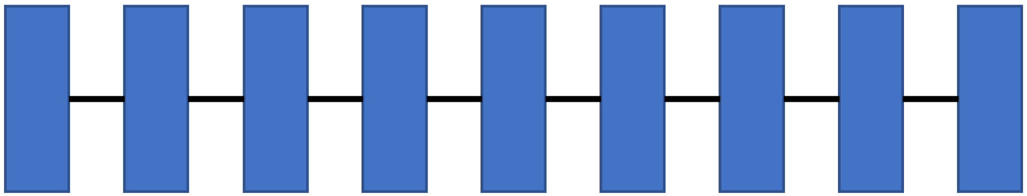
Our team solved this problem with
<u>square root decomposition technique</u>.

So we named this new data structure **SQRT List**.

3. Implementation

From the Linked List

- We need the concept of a multi-level list
- We will divide the elements into something called buckets
- We will use buckets to increase random access speed



Linked List (*N*=9)

Square Root Decomposition

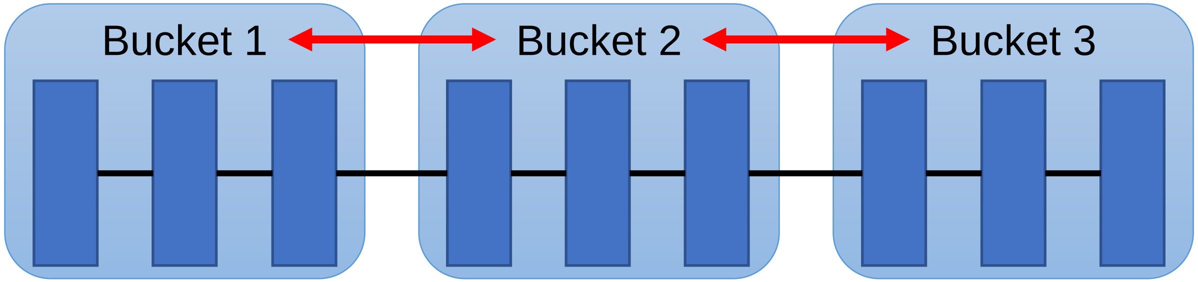
Building Buckets

- Divide elements into buckets, in sequence → **Square Root Decomposition**

  Bucket Count (the number of buckets): $\sqrt{N}$ (a near integer)

  Bucket Size (the number of elements in a bucket): ***N* ÷ (*Bucket Count*) =** $\sqrt{N}$

  The bucket count $\sqrt{N}$ is optimal for random access
- Each bucket has a direct link to the prev & next bucket



SQRT List (N=9)

O($\sqrt{N}$) Random Access

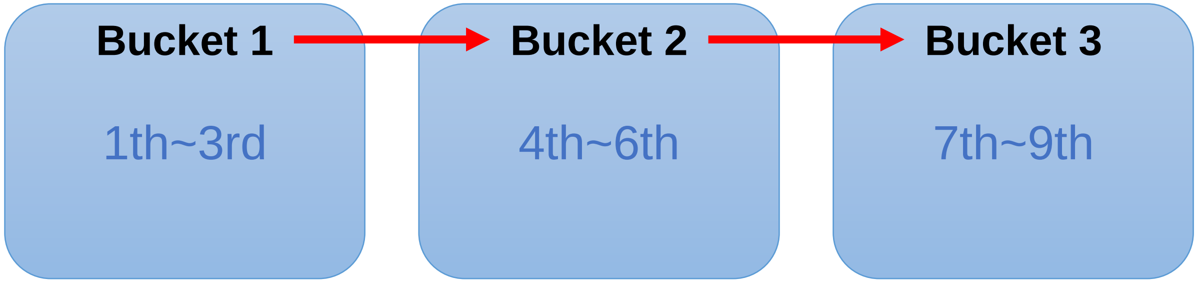
❖ How to find *i*th element

- Step 1: Find the bucket

  Find the bucket that contains *i*th element

  There are only $\sqrt{N}$ buckets to traverse

  Time complexity: **O($\sqrt{N}$)**



SQRT List (*N*=9)

O($\sqrt{N}$) Random Access

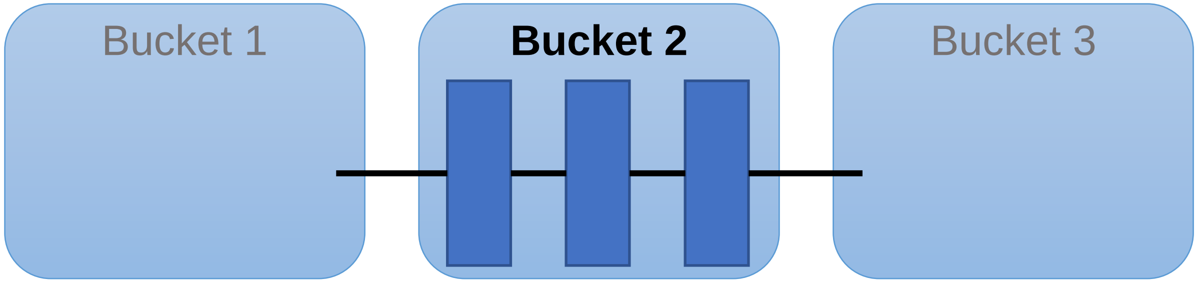
How to find *i*th element

- Step 2: Find the element

  Find the element in the selected bucket

  There are only $\sqrt{N}$ elements to traverse in one bucket

  Time complexity: **O($\sqrt{N}$)**

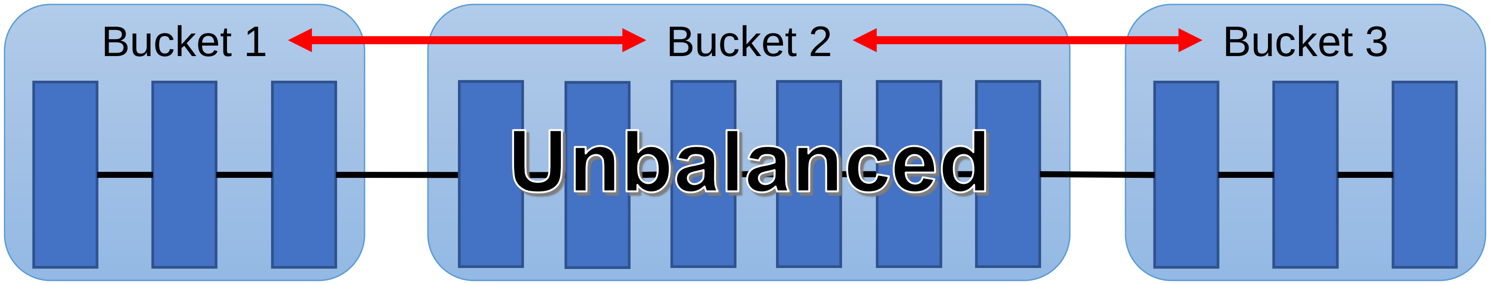


SQRT List (N=9)

Insertion

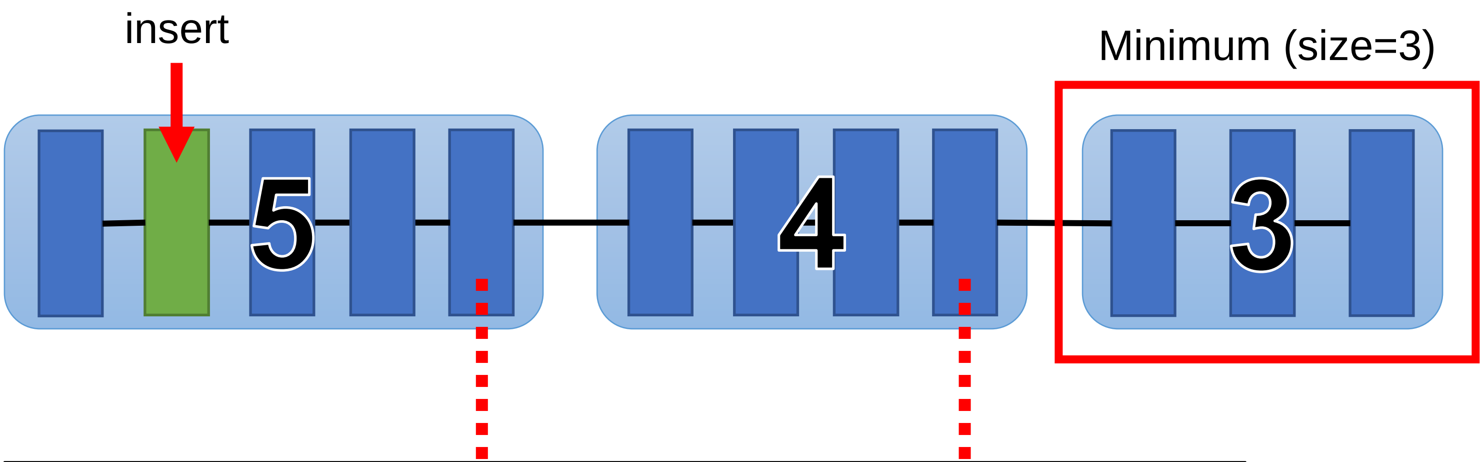
❖ Problem 1

- If we keep inserting elements into only one bucket, the size between buckets can become unbalanced
- Unbalanced buckets degrade random access performance
- We need a self-balancing bucket

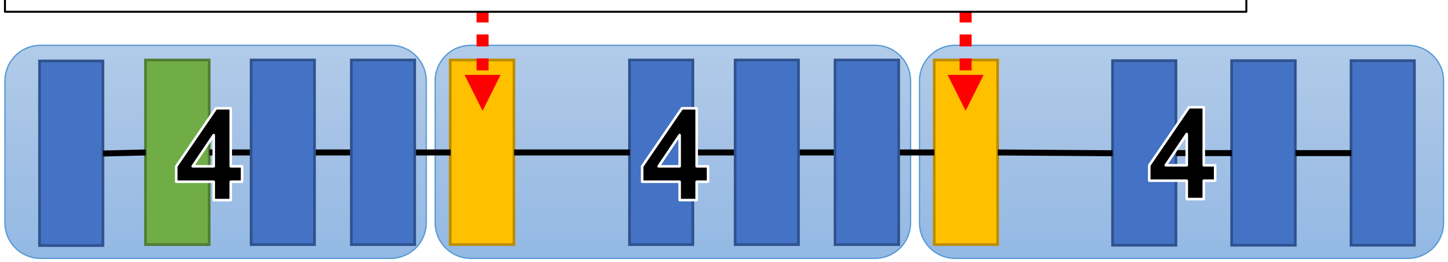


O($\sqrt{N}$) Self-Balancing Bucket

Step 1. Right after insertion, find minimum size bucket : **O($\sqrt{N}$)**



Step 2. From the bucket of an inserted element to the minimum bucket, move one element of each bucket in the direction of minimum bucket : **O($\sqrt{N}$)**



After that, The size between buckets can differ by at most 1, always! → **Balanced**

Insertion

Problem 2

- If we keep inserting elements, the size of a bucket is getting bigger
- Even if they are balanced, it degrades random access performance
- We need to add a new bucket to reduce the size of each bucket

The simple solution: Rebuilding Buckets

- **Remove all old buckets and create new $\sqrt{N}$ buckets from scratch**

  Unfortunately, it costs **O(N)** to do that
- However, you don't need to rebuild buckets every time

  Rebuild when the number of elements N becomes about 1, 4, 9, 16, 25...

  Because the integer value of $\sqrt{N}$ is changed only at those moments
- So the **amortized time complexity** is low

  O(N) cost, once per $\sqrt{N}$ times = $\mathbf{O}(\frac{N}{\sqrt{N}})$ = $\mathbf{O}(\sqrt{N})$

Similar to the technique used in C++'s vector and Java's ArrayList

Deletion

Deletion is the inverse of insertion

- Deletion is almost the same as insertion, except that it is the inverse
- Self-Balancing Algorithm: Do it from the maximum size bucket to the bucket of a deleted element, **O($\sqrt{N}$)**
- Rebuilding Algorithm: Only some conditions are slightly different, **O($\sqrt{N}$)**

Total Time Complexity

- **Random Access**: (Find the bucket) + (Find the element) = **O($\sqrt{N}$)**
- **Insertion**: (Self-Balancing) + (Rebuilding) = **O($\sqrt{N}$)**
- **Deletion**: (Self-Balancing) + (Rebuilding) = **O($\sqrt{N}$)**

4. Performance Test

Linux Data Structures

- XArray, List, SQRT List

Operations

- Random Access (x20000) : Read *i*th element

  i: random number between 1 and 20000

- Sequential Access (x20000) : Read the next element consecutively

  From 1st element to 20000th element

- Insert (x20000) : Insert an element at a random position

  An array must have consecutive indices from 0 to *N*-1 after this operation

- Delete (x20000) : Delete a random element

  An array must have consecutive indices from 0 to *N*-1 after this operation

4. Performance Test

❖ Result

- **Random Access:** SQRT List is 57x faster than List (17ms < 1014ms)
- **Total:** SQRT List is 36x faster than List (28ms < 1016ms)

```
[ 2602.245114] ====== SQRT List Module Init ======
[ 2609.338761] xarray_test    (random access)     :    0.62 ms (20000 times)    O(1)
[ 2609.338783] xarray_test    (sequential access):    0.63 ms (20000 times)    O(1)
[ 2609.338784] xarray_test    (insert)            : 3531.08 ms (20000 times)    O(N)
[ 2609.338784] xarray_test    (delete)            : 3558.24 ms (20000 times)    O(N)
[ 2609.338784] xarray_test    (total)             : 7090.59 ms

[ 2611.187750] list_test      (random access)     : 1014.69 ms (20000 times)    O(N)
[ 2611.187751] list_test      (sequential access):    0.44 ms (20000 times)    O(1)
[ 2611.187752] list_test      (insert)            :    0.46 ms (20000 times)    O(1)
[ 2611.187752] list_test      (delete)            :    0.47 ms (20000 times)    O(1)
[ 2611.187752] list_test      (total)             : 1016.09 ms

[ 2611.236182] sqrt_list_test (random access)     :   17.85 ms (20000 times)    O(√N)
[ 2611.236184] sqrt_list_test (sequential access):    0.49 ms (20000 times)    O(1)
[ 2611.236184] sqrt_list_test (insert)            :    4.74 ms (20000 times)    O(√N)
[ 2611.236184] sqrt_list_test (delete)            :    5.15 ms (20000 times)    O(√N)
[ 2611.236184] sqrt_list_test (total)             :   28.24 ms
[ 2611.238229] ====== SQRT List Module Exit ======
ubuntu@ubuntu:~/source/cau-linux/sqrt-list$
```

Q&A