تاریخچه Skip list :

Skip List در سال ۱۹۸۹ توسط William Pughمعرفی شد تا جایگزینی ساده‌تر برای درخت‌های متوازن باشد. این ساختار داده با استفاده از تصادفی‌سازی و لایه‌های چندگانه، بدون نیاز به الگوریتم‌های پیچیده‌ی متوازن‌سازی، عملیات جستجو، درج و حذف را به‌طور امید ریاضی با پیچیدگی O(log n) انجام می‌دهد Skip List .به دلیل سادگی و کارایی، در سیستم‌های واقعی مانند پایگاه‌های داده و سیستم‌های همزمان استفاده می‌شود.

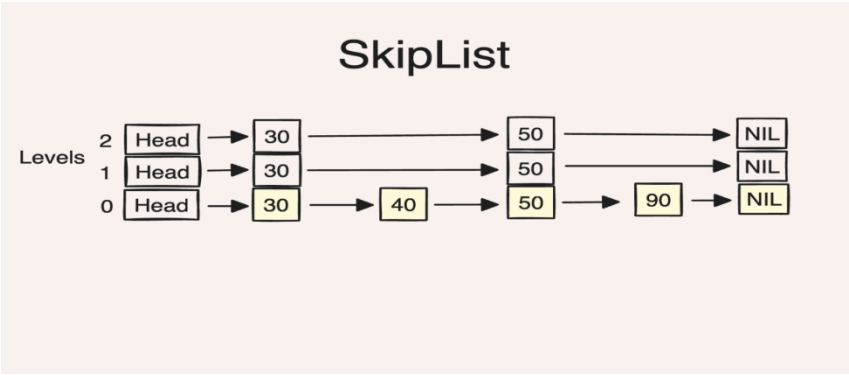
ساختار کلی Skip List :

یک Skip List شبیه لیست پیوندی مرتب است، اما با چندین لایه که روی هم قرار گرفته اند.

-لایه صفر : همان لیست پیوندی مرتب ساده است و شامل تمام عناصر می شود.

-لایه های بالا تر : شامل زیر مجموعه ای از گره های لایه پایین تر هستند که که به صورت تصادفی انتخاب می شوند.

این لایه ها نقش میان بر را برای عبور سریع از عناصر را دارند.



گره :

```java
class SkipListNode{
    int value;
    SkipListNode[] forward; // be node badi eshare mi konad
    SkipListNode(int value , int level){
        this.value = value;
        this.forward = new SkipListNode[level + 1];
    }
}
```

فیلد های کلاس :

```java
public class SkipList {
    private static final int MAX_LEVEL = 16;
    private static final double p = 0.5; // ehtemal raftan be laye bala tar
    private final SkipListNode header; // (sentinel) shoro va payan ra moshakhas mikonad
    private int level;
    private final Random random;

    public SkipList(){
        this.level = 0;
        this.header = new SkipListNode(Integer.MIN_VALUE , MAX_LEVEL);
        this.random = new Random();
    }
```

جستجو در Skip List :

```java
    public boolean search(int value){  no usages  new *
        SkipListNode curr = header;
        for(int i = level ; i >= 0 ; i--){
            while(curr.forward[i] != null && curr.forward[i].value < value){
                curr = curr.forward[i];
            }
        }
        curr = curr.forward[0];
        return curr != null && curr.value == value;
    }
```

از Header بالا ترین لایه شروع می کنیم و در لایه فعلی تا جایی که گره بعدی وجود داشته باشد و مقدار آن از value کوچکتر باشد جلو می رویم. وقتی دیگر نمی توانیم به جلو برویم به لایه پایین تر می رویم و مرحله قبل را دوباره تکرار می کنیم. در نهایت به لایه صفر می رسیم یا گره مورد نظر را پیدا می کنیم یا به انتهای لیست می رسیم و مقدار وجود ندارد.

درج یک عنصر جدید در Skip List :

```java
public boolean insert(int value){  4 usages  new *
    SkipListNode[] update = new SkipListNode[MAX_LEVEL + 1];
    SkipListNode curr = header;

    for(int i = level ; i >= 0 ; i--){
        while(curr.forward[i] != null && curr.forward[i].value < value){
            curr = curr.forward[i];
        }
        update[i] = curr;

    }
    curr = curr.forward[0];
    if(curr != null && curr.value == value)
        return false;

    int lvl = randomLevel();
    if(lvl > level){
        for(int i = level + 1 ; i <= lvl ; i++){
            update[i] = header;
        }
        level = lvl;
    }

    SkipListNode newNode = new SkipListNode(value , lvl);
    for(int i = 0 ; i <= lvl ; i++){
        newNode.forward[i] = update[i].forward[i];
        update[i].forward[i] = newNode;
    }

    return true;
}
```

مشابه جستجو از بالا ترین لایه شروع کرده و به جلو و پایین می رویم تا محل مناسب در لایه صفر را پیدا کنیم. با احتمال p تابع(random level) مشخص می کنیم که این گره چند لایه خواهد داشت. در هر لایه گره جدید را بین گره قبلی و بعدی قرار می دهیم و اشاره گر forward را در تمام لایه ها بروز می کنیم.

```java
private int randomLevel(){  1 usage  new *
    int lvl = 0;
    while(random.nextDouble() < p && lvl < MAX_LEVEL){
        lvl++;
    }

    return lvl;
}
```

حذف یک عنصر در Skip List :

```
public boolean delete(int value){  no usages  new *
    SkipListNode[] update = new SkipListNode[MAX_LEVEL + 1];
    SkipListNode curr = header;

    for(int i = level ; i >= 0 ; i--){
        while(curr.forward[i] != null && curr.forward[i].value < value){
            curr = curr.forward[i];
        }
        update[i] = curr;
    }
    curr = curr.forward[0];

    if(curr == null || curr.value != value)
        return false;

    for(int i = 0 ; i <= level ; i++){
        if(update[i].forward[i] != curr)
            break;

        update[i].forward[i] = curr.forward[i];
    }

    while(level > 0 && header.forward[level] == null){
        level--;
    }

    return true;
}
```

گره هایی که باید اشاره گر آنها به گره بعدی تغییر کند را در آرایه ای به نام update ذخیره می کنیم. در هر لایه اشاره گر گره پیشین را به گره بعدی گره حذف شده وصل می کنیم. در نهایت اگر بالاترین لایه خالی شد، ارتفاع لیست را کاهش می دهیم.

ارتفاع هر گره با احتمال تصادفی تعیین می شود. این روش احتمالاتی به جای الگوریتم های متوازن سازی درخت ها عمل می کند. اکثر گره ها در لایه های پایین قرار دارند و تعداد کمی از آنها به لایه های بالا می روند. این توزیع احتمال باعث می شود که جستجو سریع باشد ، درج و حذف ساده و با بروزرسانی محدود انجام شود و پیچیدگی زمانی(O(logn و فضایی (O(n است.

مشکلاتی که Skip List حل می کند :

1- در یک لیست پیوندی معمولی جستجو نیازمند پیمایش خطی است و پیچیدگی زمانی آن (O(n است اما در Skip List با اضافه کردن لایه های بالاتر به عنوان میان بر ، جستجو با پیچیدگی (O(logn انجام می شود.

2- درخت هایی مانند AVL , Red-Black الگوریتم های متوازن سازی پیچیده دارند اما Skip List بدون چرخش و تنها با احتمال تعادل خود را حفظ می کندو پیاده سازی ساده تری دارد.

3-عملیات های اصلی با پیچیدگی (O(logn انجام می شوند در حالی که ساختار کد ساده است.

نکات منفی و محدودیت ها :

1- عملکرد آن امیدی است و نه قطعی. در بدترین حالت ممکن است تمام عناصر در لایه صفر باشند و در این حالت پیچیدگی عملیات ها (O(n خواهد بود هر چند که وقوع این حالت کم است.

2- حافظه ای که مصرف می کنند از لیست پیوندی و حتی برخی درخت ها بیشتر است در نتیجه در سیستم هایی با حافظه کم مشکل ساز هستند.

3-اگر مولد تصادفی خوب نباشد کیفیت کاهش می یابد.

4-برای سیستم های Real-Time که باید زمان اجرای عملیات ها قابل پیش بینی باشند انتخاب مناسبی نیستند.

پیچیدگی زمان و فضا :

فرض می کنیم که :

1- تعداد کل عناصر = n

2- احتمال ارتقا هر گره به لایه بالا تر = p (p معمولا 0.5 است)

تحلیل ارتفاع :

احتمال اینکه یک گره حداقل تا لایه k بالا برود

```
P(height >= k) = p ^ k
```

امید ریاضی تعداد گره ها در لایه

```
n x (p ^ k)
```

پس بالا ترین لایه زمانی وجود دارد که

```
n x (p ^ k) >= 1
```

با حل نامساوی نتیجه می گیریم که

```
p ^ k >= 1 / n

(1 / p) ^ k <= n

k x log(1 / p) <= log(n)

k <= log(n) "base = 1 / p"

=> Expected hight of Skip List = O(log n)
```

پیچیدگی زمانی جستجو :

امید ریاضی حرکات افقی در هر لایه

```
1 / p
```

(چون احتمال وجود گره بعد در همان لایه p است)

پس خواهیم داشت

```
for search we have : O((1 / p) x log n)

if p = 1/2 => O(2 x logn) = O(log n)
```

در نتیجه پیچیدگی زمانی جستجو از مرتبه O(logn) است.

پیچیدگی زمانی درج :

1- جستجو برای محل درج = O(logn)

2- تولید ارتفاع تصادفی = O(1)

3- به روز رسانی اشاره گر ها

امید ریاضی ارتفاع یک گره

```
(p ^ 0) + (p ^ 1) + ... + (p ^ infinity)

which is equal to 1 / ( / p)

if p = 1/2 => 2
```

پس پیچیدگی زمانی درج از مرتبه O(logn) خواهد بود.

پیچیدگی زمانی حذف :

1- جستجو برای محل گره = O(logn)

2- باید آن را از تمام لایه هایی که در آن حضور دارد حذف کنیم که تعداد این لایه ها برابر با ارتفاع گره است پس پیچیدگی زمانی آن = O(1)

پس در نتیجه پیچیدگی زمانی درج از مرتبه O(logn) خواهد بود.

نکته1 : با حذف گره ممکن است بالاترین لایه خالی شود در این حالت باید ارتفاع را یک واحد کاهش دهیم که پیچیدگی آن O(n) است.

نکته2 : اگر چه بدترین حالت درج و حذف O(n) است اما احتمال وقوع آن ((p ^ (n-1)) است که برای n بزرگ عملا صفر است.

پیچیدگی فضا :

در Skip List هر گره در چند لایه حضور دارد و برای هر لایه یک اشاره گر نگه داری می شود پس فضای مصرفی به تعداد اشاره گر ها بستگی دارد و نه فقط تعداد گره ها.

فرض کنیم تعداد عناصر برابر n و احتمال رفتن یک گره به لایه بالاتر برابر p (p = 0.5) است.

احتمال اینکه یک گره حداقل ارتفاع k را داشته باشد

```
P(height >= k) = p ^ k
```

امید ریاضی تعداد اشاره گر های یک گره

```
1 + p + (p ^ 2) + (p ^ 3) + .... = 1 / (1 - p)

if p = 2 => 1 / (0.5) = 2
```

یعنی به طور میانگین هر گره فقط دو اشاره گر دارد.

در نتیجه پیچیدگی فضا از مرتبه O(n) خواهد بود.

مقایسه با داده ساختار های دیگر:

ساختار داده	جستجو(متوسط)	درج(متوسط)	حذف(متوسط)	بدترین حالت
لیست پیوندی	O(n)	O(1)	O(1)	O(n)
آرایه مرتب	O(logn)	O(n)	O(n)	O(n)
درخت متوازن	O(logn)	O(logn)	O(logn)	O(logn)
Hash Table	O(1)	O(1)	O(1)	O(n)
Skip List	O(logn)	O(logn)	O(logn)	O(n)

تست :

```java
public class Main {
    public static void main(String[] args) {
        SkipList skipList = new SkipList();
        Random random = new Random();

        int n = 1_000_000;
        int[] testValues = new int[n];


        for (int i = 0; i < n; i++) {
            testValues[i] = random.nextInt(Integer.MAX_VALUE);
        }

        long startInsert = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            skipList.insert(testValues[i]);
        }
        long endInsert = System.currentTimeMillis();
        System.out.println("Insert time for " + n + " elements: " + (endInsert - startInsert) + " ms");


        long startSearch = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            skipList.search(testValues[random.nextInt(n)]);
        }
        long endSearch = System.currentTimeMillis();
        System.out.println("Search time for " + n + " elements: " + (endSearch - startSearch) + " ms");
```

```java
        long startDelete = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            skipList.delete(testValues[i]);
            long endDelete = System.currentTimeMillis();
            System.out.println("Delete time for " + n + " elements: " + (endDelete - startDelete) + " ms");
        }
    }
}
```

```
Insert time for 1000000 elements: 2459 ms
Search time for 1000000 elements: 2330 ms
Delete time for 1000000 elements: 1961 ms
```

لینک گیت هاب : https://github.com/aryafbz/Skip-List.git