
# پروژه‌ی نهایی درس ساختمان داده‌ها

# پیاده‌سازی درخت BJR برای حل مسئله‌ی Skyline

نازنین فروتن ۴۰۲۲۴۳۰۸۴

ریحانه داداش‌پور ۴۰۲۲۴۳۰۶۰

## | مقدمه

- تعریف مسئله: در این پروژه با استفاده از درخت BJR و کشND ، هدف محاسبه مجموعه نقاط Skyline از یک مجموعه داده است. نقاط Skyline نقاطی هستند که در هیچ بعدی توسط هیچ نقطه دیگری کاملاً تسلط ندارند. این نقاط در مسائل مختلفی مانند پایگاه‌های داده، تصمیم‌گیری، و تحلیل داده‌ها کاربرد دارند.
- هدف پروژه: بررسی روش‌های بهینه‌سازی برای محاسبه Skyline در فضای چندبعدی با استفاده از BJR-tree و ND-Cache .

## | توضیح مسأله

### ○ عناصر

- Skyline: مجموعه‌ای از نقاط است که هیچ نقطه‌ای در تمام ابعاد بر آن تسلط ندارد. این نقاط، برای تحلیل‌های مختلف، معمولاً اطلاعات بیشتری نسبت به دیگر نقاط دارند.
- فضای چندبعدی: با افزایش ابعاد، مسئله‌ی Skyline پیچیده‌تر می‌شود. تعداد نقاط Skylineممکن است به‌طور غیرمنتظره‌ای زیاد شود، مخصوصاً زمانی که ابعاد فضای داده زیاد باشد.
- مفهوم تسلط(Domination) : نقطه‌ایa ، نقطه‌ای b را Dominate می‌کند اگر در تمامی ابعاد، a بهتر از b باشد.
- Continuous Skyline Computation: اگر مجموعه داده پویا باشد (یعنی نقاط به‌طور پیوسته وارد و خارج شوند)، نیاز به محاسبه‌ی پیوسته نقاط Skyline خواهیم داشت.

### ○ ساختار داده

#### BJR-Tree (a

- **ساختار درخت BJR:** در درختBJR ، هر گره نماینده یک نقطه است و یال‌ها نشان‌دهنده روابط تسلط هستند. اگر یالی از a به b وصل باشد، به این معناست که a نقطه b را تسلط می‌کند.

- **ویژگی‌های BJR-Tree :**
  - درخت **سلسله‌مراتبی** است.
  - مستقل از ابعاد عمل می‌کند.
  - نقاط Skyline به ریشه وصل می‌شوند.
  - گره‌هایی که به ریشه نزدیک‌ترند، Skyline potential بیشتری دارند.
- **Lazy Strategy:** این استراتژی در مقایسه با روش معمول که در آن به فرزندان ریشه بررسی می‌شود، به تعداد اولاد هر گره توجه می‌کند و به طور هوشمندتر گره‌ها را بررسی می‌کند تا طول درخت را کاهش دهد.

**b) ND-Cache**

- **وظیفه‌ی ND-Cache:** این کش اطلاعات non-dominance را نگه می‌دارد تا از محاسبات مجدد جلوگیری کند.
- **روش کار:** هر زمان که یک نقطه Skyline باشد، اطلاعات آن در کش ذخیره می‌شود. برای بررسی اینکه آیا نقطه‌ای در زمان‌های مختلف یکدیگر را dominate می‌کنند یا نه، می‌توان از کش استفاده کرد.
- **اطلاعات ذخیره‌شده:** به‌طور مثال، اگر نقطه‌ای در زمان T3 جزء Skyline باشد، در کش ذخیره می‌شود که در زمانT3 ، این نقطه Skyline بوده است.

**c) Continuous Skyline Computation**

- **فعال‌سازی و غیرفعال‌سازی نقاط:** نقطه‌ای می‌تواند فعال یا غیرفعال شود بسته به اینکه در بازه زمانی فعلی جزء Skyline است یا خیر.
- **Skyline Potential:** پتانسیل یک نقطه برای Skyline بودن به تعداد نقاطی که آن را تسلط می‌کنند بستگی دارد. هرچه تعداد نقاط تسلط بیشتر باشد، پتانسیل آن برای Skyline شدن کمتر است.

## پیاده‌سازی

برای پیاده‌سازی این پروژه از دو فایل استفاده شده که شامل main.cpp و modules.hpp می‌شه. فایل main شامل توابع مربوط به ران شدن و خواندن فایل‌ها و تست‌کیس است. فایل modules هم شامل کلاس‌هایی هستند که برای ساخت درخت و کار با BJR tree باهاشون کار داریم.

- **modules.hpp**
  - **ساختار point**

در فایل modules.hpp، ساختار Point به‌طور مستقیم برای نگهداری اطلاعات هر نقطه از مجموعه داده‌ها استفاده می‌شود. هر نقطه دارای شناسه (id) و مقدارهایی برای هر بعد است که در آرایه values ذخیره می‌شود.

```cpp
struct Point {
    int id;
    int dim;
    int* values;

    Point() : id(-1), dim(0), values(nullptr) {}
    Point(int id, int dim) : id(id), dim(dim) {
        values = new int[dim];
    }
};
```

  - **ساختار node**

درخت ما از گره‌ها (Node) تشکیل می‌شود. هر گره نماینده یک نقطه است و به فرزندان خود لینک دارد. اگر گره‌ای به ریشه متصل باشد، یعنی آن نقطه Skyline است.

```cpp
struct Node {
    Node* head_child;
    Node* next;
    Node* parent;
    Point point;
    bool is_root = false;

    void add_child(Node* child) { }

    int children_size(){ }

    void remove_child(Node* child){ }

};
```

- **ساختار BJR Tree**

درخت BJR برای ذخیره‌سازی روابط تسلط (Domination) بین نقاط استفاده می‌شود.

در این درخت، هر گره یک نقطه است و یال‌ها نشان‌دهنده تسلط هستند.

```cpp
struct BJR_tree{
    Node* root;
    int total_points = 0;
    bool* exists;
    int depth = 0;
    bool lazy = false;
    int* ND_cache = nullptr;
    bool ND_use = false;

    BJR_tree(int num_points) : total_points(num_points) {
        exists = new bool[total_points];
        for(int i=0 ; i<total_points ; i++){
            exists[i] = false;
        }
    }

```

```
16
17      bool does_point_exist(int id){}
18      Node* find_node_by_id(int id, Node* root) {}
19      int Depth(Node* node){}
20      int Desc(Node* node){}
21      void add_to_exists(Node* node){}
22      void remove_exists(Node* node){}
23      void inject(Node* root, Node* new_node){}
24      void lazy_inject(Node* root, Node* new_node){}
25      void eject(Node* example_node) {}
26
27  };
28
```

- **نحوه ذخیره‌سازی دیتا در درخت**

در این نحوه ذخیره‌سازی از روشی مشابه ایده صورت پروژه استفاده شده.

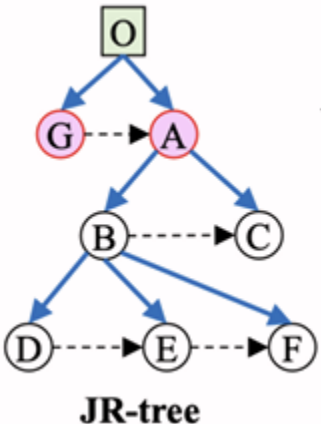
داده‌ها در ساختار درختی به‌طور خاص به روش JR-tree ذخیره می‌شوند. در اینجا سه ستون اصلی از جمله "آدرس(Address) "، "والد(Parent) "، "بعدی (Next) "و "سرچکیده (Head child) "برای هر گره در درخت مشخص شده است. این روش ذخیره‌سازی اطلاعات از سه جدول مختلف استفاده می‌کند:

۱. **آدرس (Address)**: این ستون به هر گره در درخت یک آدرس خاص اختصاص می‌دهد. در این ساختار، هر گره در درخت یک شناسه منحصر به فرد دارد که آن را از دیگر گره‌ها متمایز می‌کند.

۲. **والد(Parent)**: این ستون نشان می‌دهد که هر گره به کدام گره دیگر به‌عنوان والد تعلق دارد. مثلا، گره A والد گره‌های B و C است، و گره B والد گره‌های D ، Eو F است.

۳. **بعدی(Next)**: این ستون روابط بین گره‌ها را مشخص می‌کند. اگر گره‌ای فرزند دیگری داشته باشد، به گره بعدی اشاره می‌کند. در برخی موارد، این مقدار "Null"است که نشان‌دهنده این است که گره دیگر فرزند ندارد.

۴. **سرچکیده(Head child)**: این ستون نشان می‌دهد که هر گره فرزند اول خود را در کدام آدرس ذخیره کرده است. اگر گره‌ای فرزند نداشته باشد، مقدار آن "Null"خواهد بود.




**Contents of tree-structure memory**

| Address | Parent | Next | Head child |
|---|---|---|---|
| A | Root | Null | B |
| B | A | C | D |
| C | A | Null | Null |
| D | B | E | Null |
| E | B | F | Null |
| F | B | Null | Null |
| G | Root | A | Null |

- **معرفی متدهای اصلی درخت**

✓ **Depth()**

- توضیح: این تابع معمولاً برای محاسبه عمق درخت یا گره‌ها استفاده می‌شود.
- کاربرد: در پروژه‌های درختی مانندBJR-tree ، عمق درخت می‌تواند بر عملکرد الگوریتم تاثیر بگذارد. برای مثال، درخت‌های کم‌عمق معمولاً سریع‌تر عمل می‌کنند. این تابع معمولاً عمق گره را نسبت به ریشه درخت محاسبه می‌کند.

```
int Depth(Node* node){
    int res = 0;
    Node* current = node;
    while(current->parent != nullptr){
        res++;
        current = current->parent;
    }
    return res;
}
```

✓ **Desc()**

- توضیح: این متد برای برگرداندن تعداد نوادگان یک گره استفاده می‌شه.

- کاربرد: برای زمانی استفاده می‌شه که بخوایم از lazy evaluation استفاده کنیم.

```cpp
int Desc(Node* node){
    if(node->head_child == nullptr) return 0;

    int res = 1;
    Node* current = node->head_child;
    while(current != nullptr){
        res++;
        res += Desc(current);
        current = current->next;
    }

    return res;
}
```

### ✓ inject()

- **توضیح :**این تابع برای وارد کردن یک نقطه جدید به درخت و برقراری روابط تسلط (domination) استفاده می‌شود.
- **کاربرد :**در درختBJR ، زمانی که یک نقطه جدید به درخت افزوده می‌شود، این تابع باید روابط تسلط را بررسی کرده و نقطه جدید را در مکان مناسب در درخت قرار دهد. در صورت نیاز، باید گره‌ها و یال‌ها به‌طور داینامیک تغییر یابند.

**Algorithm 1** New node injection

```
1:  procedure inject(parent, e: new node)
2:  children ← children of parent;
3:  for all c ∈ children do
4:      if dominates(c, e) then
5:          inject(c, e);
6:          return
7:  add e to parent as a child;
8:  for all c ∈ children do
9:      if dominates(e, c) then
10:         move c to e as a child;
```

✓ **lazy_inject()**

- **توضیح:** این تابع مشابه به inject() است، اما با یک تفاوت کلیدی. در این روش، گره‌ها فقط در صورت نیاز به‌طور کامل وارد درخت می‌شوند، یعنی عملیات کمتری انجام می‌دهند و این به‌ویژه در درخت‌های بزرگ که ممکن است به عملکرد آسیب بزنند، مفید است.
- **کاربرد:** این تابع می‌تواند به‌ویژه در پروژه‌هایی که نیاز به محاسبات سریع‌تر دارند، مفید باشد. این روش باعث می‌شود که از اضافه کردن گره‌ها به‌طور پیوسته جلوگیری شود و فقط زمانی که نیاز است، تغییرات اعمال شود.

✓ **eject()**

- **توضیح**: این تابع برای حذف یک نقطه از درخت استفاده می‌شود. زمانی که یک نقطه دیگر جزء نقاط Skylineنیست، باید از درخت حذف شود.
- **کاربرد:** در درخت‌های دینامیک مانند BJR-tree، زمانی که یک نقطه دیگر شرایط لازم برای Skyline بودن را ندارد (مثلاً توسط نقطه دیگری تسلط شده است)، این تابع باید آن را از درخت حذف کند و گره‌ها و روابط تسلط را به‌روزرسانی کند.

**Algorithm 2** Node ejection

```
1: procedure eject(e: node)
2: parent ← parent of e;
3: children ← children of e;
4: remove e from parent;
5: for all c ∈ children do
6:     inject(parent, c);
```

- **main.cpp**

**۱. خواندن داده‌های پیکربندی و ورودی‌ها**

داده‌های ورودی شامل چند فایل مختلف هستند و ما سه نوع سایز small, medium و large داریم:

- **size.setup**: این فایل شامل پیکربندی داده‌ها است، مثل تعداد ابعاد نقاط، تعداد مراحل زمانی(time steps) ، و نوع داده‌ها.
- **small.times**: این فایل شامل اطلاعات مربوط به زمان‌های شروع و پایان هر نقطه است. به‌عبارت‌دیگر، این فایل نشان می‌دهد که هر نقطه در چه بازه زمانی فعال است.
- **small.input**: این فایل شامل مقادیر هر نقطه است. برای هر نقطه، مقادیر مختلف آن در ابعاد مختلف ذخیره شده است.

**۲. پردازش نقاط برای هر زمان**

برای هر گام زمانی(time step) ، باید نقاط فعال را بررسی کنیم و اقداماتی مانند اضافه کردن یا حذف کردن آنها را انجام دهیم.

در حلقه اصلی for که بر روی زمان‌ها می‌چرخد، برای هر تایم:

- **مقایسه با تایم‌های مشخص‌شده در times**: در این مرحله، به‌ازای هر مرحله زمانی، باید ببینیم که هر نقطه آیا در بازه زمانی خاص خودش فعال است یا خیر. این کار با استفاده از داده‌های موجود در فایل timesانجام می‌شود که شامل محدوده‌های زمانی شروع و پایان هر نقطه است.

```
for(int time = 0 ; time < time_steps ; time++){
    cout << "time: " << time  << endl;
    id = 0;
    for(int t = 0 ; t < times.size() ; t++) {
        range = times[t];
        ss.clear();
        ss.str(range);
        ss >> temp;
        start = stoi(temp);
        ss >> temp;
        end = stoi(temp);

```

- **چک کردن وضعیت هر نقطه :**اگر زمان فعلی در بازه زمانی یک نقطه باشد، آن نقطه باید به درخت اضافه شود. در غیر این صورت، اگر نقطه‌ای خارج از این بازه زمانی باشد، باید از درخت حذف شود.

### ۳. حذف نقاط غیرفعال

اگر نقطه‌ای از بازه زمانی خود خارج شود یا توسط نقطه دیگری تسلط یابد، باید از مجموعه حذف شود:

```
else {
    if (time < start) break;
    if ((time >= end || time < start) && my_tree->does_point_exist(id)) {
        Point deleting_point;
        Node* deleting_node = new Node();
        deleting_point.id = id;
        deleting_node->point = deleting_point;
        my_tree->eject(deleting_node);
        delete deleting_node;
    }
}
```

### ۴. تست‌کیس و مقایسه با فایل size.refout

در وهله بعد کاری که باید انجام بشه اینه که در هر استپ زمانی نقاط skyline آن محاسبه بشه (فرزندان root) و در آرایه/وکتوری از استرینگ‌ها ذخیره بشه. و در نهایت تک‌تک اعضای آن (که به ازای هر ایندکس نماینده هر استپ زمانی یا به عبارتی نماینده اعضای هر خط در فایل رفرنس هستند.) با فایل size.refout چک بشن. اگه عضوی باشه که با خط مربوطه در فایل رفرنس همخوانی نداشته باشه و برابر نباشه به عنوان ارور مشخص می‌شه و تعداد ارورها محاسبه شده و ارور ریت مشخص می‌شود. در نهایت هم محتوای بدست آمده در فایل size.out چاپ می‌شود.

# تحلیل نتایج اجرا

## ۱ .الگوریتمBJR-tree

**پیچیدگی زمانی:**

- **ایجاد درختBJR :** درخت BJR برای ذخیره‌سازی روابط domination بین نقاط استفاده می‌شود. هر نقطه باید به درخت اضافه شود، و در این فرایند، تمام گره‌ها باید برای بررسی روابط تسلط و تصمیم‌گیری درباره‌ی اتصال به درخت بررسی شوند.
- **پیچیدگی زمانی:** هر بار که یک نقطه جدید به درخت اضافه می‌شود، زمان بررسی آن به عمق درخت بستگی دارد. اگر درخت به عمق d برسد، پیچیدگی زمان اضافه کردن یک نقطه به درخت O(d) خواهد بود. به دلیل اینکه درخت سلسله‌مراتبی است و روابط تسلط از ریشه به گره‌ها گسترش می‌یابد، در بهترین حالت این زمان خطی است O(n)، اما در بدترین حالت، پیچیدگی به O(n^2) می‌رسد که به علت بررسی روابط تسلط برای هر نقطه است.

**بهینه‌سازی:**

- Lazy Injection و ND-Cache می‌توانند به کاهش زمان بررسی و بهینه‌سازی درخت کمک کنند، اما عمق درخت همچنان به‌عنوان یک عامل محدودکننده عمل می‌کند.

**نتایج واقعی:**

- حالت عادی برای سایزهای مختلف:

```
Processing time: 0/110
Error rate: 0.00% (0 errors out of 110)
program finished
Execution time: 2 ms
Data printed in .out
Done for tree size: small, lazy: 0, nd: 0
```

```
Processing time: 8000/10998
Processing time: 9000/10998
Processing time: 10000/10998
Error rate: 0.00% (0 errors out of 10998)
Execution time: 1025 ms
Done for tree size: medium, lazy: 0, nd: 0
```

```
Processing time: 40000/40989
Error rate: 0.00% (0 errors out of 40989)
program finished
Execution time: 36946 ms
Data printed in .out
Done for tree size: large, lazy: 0, nd: 0
```

---

## ۲ .الگوریتم Injection وLazy Injection

### Injection Execution Time

**Best Case (حالت ایده‌آل):**

- Time: O(d) (فقط تا عمق درخت)
- Scenario: زمانی که درخت کم‌عمق باشد و نقاط جدید سریع جای خود را پیدا کنند.
- Advantage: درج مستقیم و سریع بدون نیاز به پیمایش کامل درخت.

**Worst Case (واقعیت بدبینانه):**

- Time: O(n)
- Scenario: زمانی که درخت بزرگ یا نامتعادل باشد و نیاز به بررسی همه گره‌ها وجود داشته باشد.
- Overhead: همه روابط تسلط بررسی می‌شوند و هزینه زمانی خطی خواهد شد.

### Lazy Injection Execution Time

**Best Case (مزیت اصلی):**

- Time: نزدیک به O(k) (فقط روی نقاطی که احتمالاً در Skyline هستند)
- Scenario: زمانی که داده‌ها پایدار باشند و بسیاری از نقاط قبل از ورود به درخت حذف شوند.
- Advantage: حذف عملیات اضافی، درج تنها در صورت نیاز، و کاهش چشمگیر هزینه محاسباتی.

**Worst Case (حالت بحرانی):**

- Time: O(n)
- Scenario: زمانی که تقریباً همه نقاط باید بررسی شوند و در نهایت وارد درخت شوند.
- Overhead: هزینه زمانی مشابه Injection معمولی، زیرا نیاز به مرور کل درخت خواهد بود.

**نتایج واقعی:**

- نتایج برای سایزهای مختلف dataset:

```
Processing time: 0/110
Error rate: 0.00% (0 errors out of 110)
program finished
Execution time: 1 ms
Data printed in .out
Done for tree size: small, lazy: 1 , nd: 0
```

```
Processing time: 10000/10998
Error rate: 0.00% (0 errors out of 10998)
program finished
Execution time: 2273 ms
Data printed in .out
Done for tree size: medium, lazy: 1, nd: 0
```

```
Processing time: 40000/40989
Error rate: 0.00% (0 errors out of 40989)
program finished
Execution time: 72292 ms
Data printed in .out
Done for tree size: large, lazy: 1, nd: 0
```

---

## ۳. ND-Cache

**Best Case (مزیت تئوری):**

- Time: O(1) برای هر بررسی non-dominance ذخیره‌شده
- Scenario: زمانی که Skyline پایدار باشد و نقاط برای چندین گام زمانی ثابت بمانند.
- Savings: کاهش چشمگیر مقایسه‌ها، تا بیش از ۹۰٪ سریع‌تر در صورت تکرار زیاد مقایسه‌ها.

**Worst Case (واقعیت در عمل):**

- Time: می‌تواند ۲۰ تا ۳۰ درصد کندتر از حالت بدون کش باشد.
- Scenario: در Skylines پویا، زمانی که ورود و خروج نقاط زیاد باشد.
- Overhead: شامل بررسی کش و دسترسی‌های حافظه، که می‌تواند منجر به افزایش زمان شود.

### دلایل کند شدن ND-Cache در عمل

- Cache Check Overhead: حدود ۵ تا ۷ دستور اضافه برای هر بررسی تسلط.
- Memory Access Patterns: دسترسی تصادفی که منجر به Cache Miss می‌شود.
- Low Hit Rate: معمولاً کمتر از ۱۰٪ برای مجموعه‌داده‌های پویا.
- False Negatives: حتی با کش همچنان نیاز به بررسی کامل تسلط وجود دارد.

**نتایج واقعی:**

- نتایج برای سایزهای مختلف وقتی فقط nd فعال است:

```
Processing time: 0/110
Error rate: 0.00% (0 errors out of 110)
program finished
Execution time: 1 ms
Data printed in .out
Done for tree size: small, lazy: 0, nd: 1
```

```
Processing time: 8000/10998
Processing time: 9000/10998
Processing time: 10000/10998
Error rate: 0.00% (0 errors out of 10998)
Execution time: 1004 ms
Done for tree size: medium, lazy: 0, nd: 1
```

```
Processing time: 38000/40989
Processing time: 39000/40989
Processing time: 40000/40989
Error rate: 0.00% (0 errors out of 40989)
Execution time: 35623 ms
Done for tree size: large, lazy: 0, nd: 1
```

### Combined Lazy + ND-Cache

**:Best Case**

- Time: O(k × d) همراه با برخی بررسی‌های O(1) از کش
- Scenario: برای مجموعه‌داده‌های بزرگ و نسبتاً پایدار
- Savings: حدود ۳۵ تا ۵۰ درصد سریع‌تر نسبت به حالت پایه

**:Worst Case**

- Time: کندتر از هرکدام به‌تنهایی
- Scenario: برای مجموعه‌داده‌های بسیار پویا و کوچک
- Overhead: ترکیب هزینه‌های هر دو روش، که باعث افت عملکرد می‌شود

```
Processing time: 0/110
Error rate: 0.00% (0 errors out of 110)
Execution time: 2 ms
Done for tree size: small, lazy: 1, nd: 1
```

```
Processing time: 10000/10998
Error rate: 0.00% (0 errors out of 10998)
program finished
Execution time: 2327 ms
Data printed in .out
Done for tree size: medium, lazy: 1, nd: 1
```

```
Processing time: 40000/40989
Error rate: 0.00% (0 errors out of 40989)
Execution time: 71380 ms
Done for tree size: large, lazy: 1, nd: 1
```

---

### ۵. عملکرد کلی

با توجه به تمام پیچیدگی‌های الگوریتم‌ها، درخت BJR و الگوریتم‌های مرتبط با آن، پیچیدگی کلی فرآیند به شدت تحت تأثیر:

- **عمق درختBJR :** هرچه عمق درخت بیشتر باشد، تعداد عملیات‌های لازم برای بررسی روابط تسلط و انجام عملیات‌ها بیشتر خواهد بود.
- **Lazy Injection**: به دلیل انجام تنها عملیات‌های ضروری، باعث کاهش قابل توجه در تعداد عملیات‌ها می‌شود.
- **ND-Cache**: سرعت قابل توجهی در دسترسی به اطلاعات قبلی و جلوگیری از محاسبات مجدد ایجاد می‌کند.
- **Quick Sort**: برای مرتب‌سازی نقاط، در حالت عمومی پیچیدگی O(nlogn)خواهد بود.