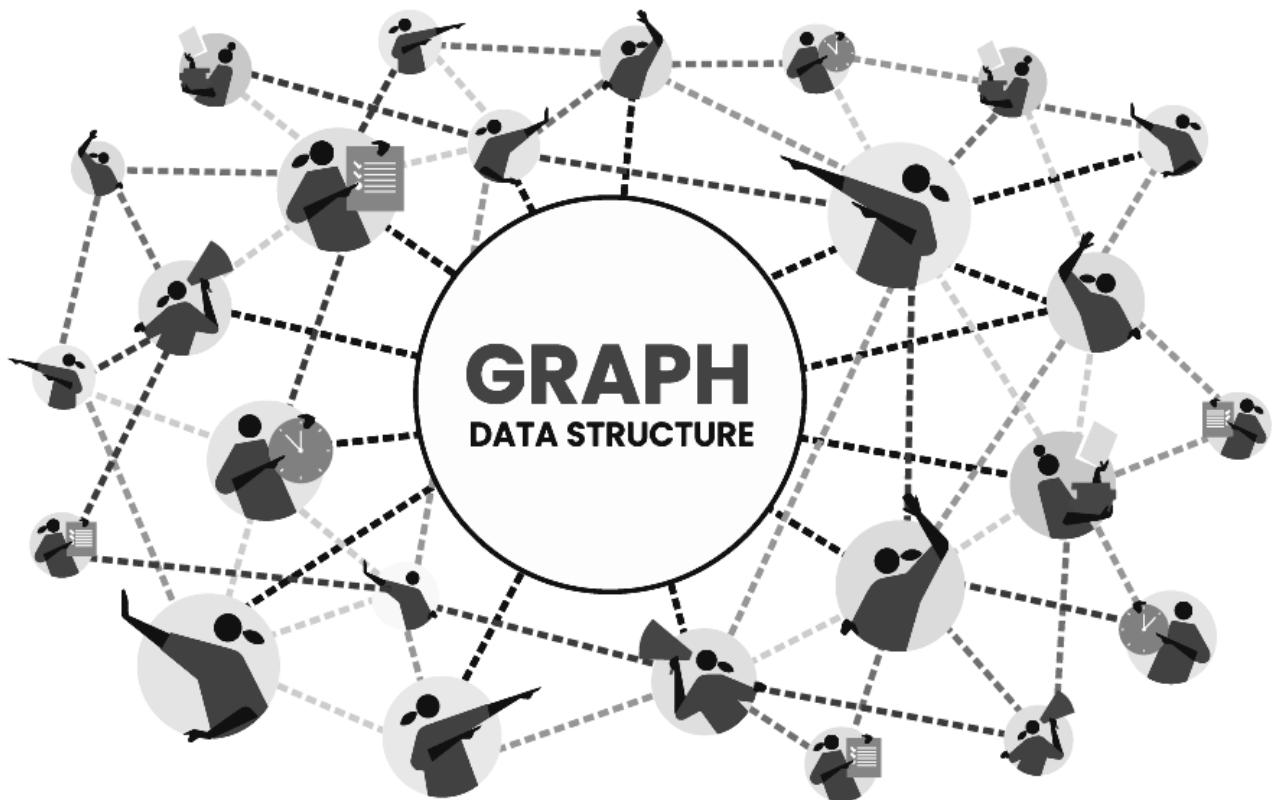


TAKING DATA STRUCTURE

FOUNDATIONS | APPLICATIONS

PRODUCTION BY | TAKING



فهرست

| | |
|-----------|--|
| صفحه..... | ساختمان‌های داده‌ای خطی..... |
| صفحه..... | آرایه‌ها (یکبعدی، دو بعدی، دینامیک)..... |
| صفحه..... | لیست‌های پیوندی (تک‌پیوندی، دو‌پیوندی، حلقوی)..... |
| صفحه..... | پشته و کاربرد آن..... |
| صفحه..... | صف و انواع آن (معمولی، دو طرفه، اولویت دار)..... |
| صفحه..... | ساختمان‌های داده‌ای غیرخطی..... |
| صفحه..... | درخت‌ها..... |
| صفحه..... | درخت دودویی..... |
| صفحه..... | درخت جستجوی دودویی..... |
| صفحه..... | درخت متوازن..... |
| صفحه..... | B و B+ درخت..... |
| صفحه..... | (درخت پیشوندی) Trie..... |
| صفحه..... | هیپ (Heap)..... |
| صفحه..... | هیپ بیشینه و کمینه..... |
| صفحه..... | هیپ فیبوناچی..... |
| صفحه..... | گراف‌ها..... |

ساختمان‌های داده‌ای خطی

ساختمان داده‌های خطی (Linear Data Structures) نوعی از ساختمان داده‌ها هستند که داده‌ها به صورت پیوسته و خطی ذخیره می‌شوند. در این ساختمان‌ها، هر عنصر تنها یک عنصر قبل و یک عنصر بعد از خود دارد. ترتیب و ساختار این داده‌ها در بسیاری از الگوریتم‌ها و مسائل برنامه‌نویسی نقش مهمی ایفا می‌کند.

ویژگی‌های اصلی ساختمان‌های داده‌ای خطی:

- دسترسی مستقیم: در بیشتر موارد می‌توان به عناصر داده به راحتی و سریع دسترسی پیدا کرد.
- پیوستگی: داده‌ها به ترتیب و پشت سر هم ذخیره می‌شوند.
- ترتیب ثابت: ترتیب داده‌ها در این ساختارها همیشه مهم است و باید حفظ شود.
- عملیات پایه‌ای: شامل عملیات‌های جستجو، درج، حذف و پیمایش است.

مزایا و معایب ساختمان‌های داده‌ای خطی:

- مزایا:
 - ساده و استفاده شده در بسیاری از مسائل.
 - سرعت بالا در دسترسی و استفاده از اندیس‌ها در آرایه‌ها.
 - ساختارهای ساده و قابل پیاده‌سازی.
- معایب:
 - آرایه‌ها: اندازه ثابت و تغییر اندازه مشکل است.
 - لیست‌های پیوندی: دسترسی به داده‌ها کندر از آرایه‌ها است.
 - پشتنهای و صفحات: قابلیت دسترسی به داده‌ها محدود است (فقط آخرین یا اولین عنصر قابل دسترسی است).

نتیجه‌گیری:

ساختمان‌های داده‌ای خطی، ابزارهای مهمی در برنامه‌نویسی و حل مسائل هستند و با توجه به ویژگی‌های خود، در سناریوهای مختلف کاربرد دارند. انتخاب نوع مناسب برای هر مشکل می‌تواند تأثیر زیادی بر کارایی الگوریتم‌ها داشته باشد.

آرایه‌ها (Arrays)

آرایه‌ها یکی از پرکاربردترین و ابتدایی‌ترین ساختمان‌های داده هستند که داده‌ها را به صورت پیوسته در حافظه ذخیره می‌کنند. در این ساختمان داده، تمامی عناصر یک نوع داده مشابه دارند و با استفاده از اندیس (index) قابل دسترسی هستند. در اینجا به سه نوع مختلف آرایه‌ها یعنی یکبعدی، دوبعدی، دینامیک پرداخته می‌شود.

۱. آرایه یکبعدی (1D Array)

تعریف:

آرایه یکبعدی، ساده‌ترین نوع آرایه است که در آن مجموعه‌ای از داده‌ها به صورت خطی در کنار هم ذخیره می‌شود. در این آرایه‌ها، هر عنصر با یک اندیس (که از صفر شروع می‌شود) قابل دسترسی است.

ویژگی‌ها:

- ذخیره داده‌ها به صورت پیوسته در حافظه.
- دسترسی سریع به عناصر از طریق اندیس ($O(1)$).
- اندازه ثابت و از پیش تعیین شده.

```
# آرایه یکبعدی
arr = [10, 20, 30, 40, 50]

# دسترسی به عنصر سوم
print(arr[2]) # 30
```

Edit ⚙ Copy ⌂

مزایا:

- دسترسی سریع به داده‌ها با استفاده از اندیس.
- ساده برای پیاده‌سازی و استفاده در برنامه‌ها.

معایب:

- اندازه ثابت است و در صورت نیاز به تغییر اندازه، باید یک آرایه جدید ایجاد شود.
- هدر دادن حافظه در صورت ذخیره‌سازی داده‌های اضافی.

۲. آرایه دو بعدی (2D Array)

تعریف:

آرایه دو بعدی مجموعه‌ای از آرایه‌های یک بعدی است که به صورت ماتریس یا جدول سازمان‌دهی می‌شوند. این آرایه‌ها دو بعد (ردیف و ستون) دارند و می‌توان به هر عنصر با استفاده از دو اندیس دسترسی پیدا کرد.

ویژگی‌ها:

- ذخیره داده‌ها به صورت جدول با ردیف‌ها و ستون‌ها.
- امکان دسترسی به داده‌ها با استفاده از دو اندیس: یکی برای ردیف و دیگری برای ستون.

```
# آرایه دو بعدی (ماتریس)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# دسترسی به عنصر در ردیف 1 و ستون 2
print(matrix[1][2]) # خروجی: 6
```

مزایا:

- مناسب برای ذخیره‌سازی داده‌های ماتریسی یا جدولی.
- دسترسی سریع به عناصر با استفاده از دو اندیس.

معایب:

- همانند آرایه یک بعدی، اندازه ثابت است و باید از قبل تعیین شود.
- هدر دادن حافظه در صورت داشتن جدول‌هایی با اندازه‌های بزرگ که کامل پر نمی‌شوند.

۳. آرایه دینامیک (Dynamic Array)

تعریف:

آرایه دینامیک آرایه‌ای است که اندازه آن در زمان اجرا قابل تغییر است. برخلاف آرایه‌های معمولی که اندازه ثابت دارند، آرایه دینامیک به طور خودکار زمانی که فضای بیشتر نیاز باشد، اندازه خود را افزایش می‌دهد. در پایتون، لیست‌ها (Lists) به عنوان آرایه‌های دینامیک عمل می‌کنند.

ویژگی‌ها:

- افزایش خودکار اندازه در صورت پر شدن آرایه.
- دسترسی به داده‌ها به صورت سریع ($O(1)$).
- در صورتی که اندازه آرایه تغییر کند، نیاز به جابجایی داده‌ها در حافظه است، که ممکن است هزینه زمانی داشته باشد.

```
# آرایه دینامیک (لیست پایتون)
dynamic_arr = [10, 20, 30]

# افزودن یک عنصر جدید به لیست
dynamic_arr.append(40)

# دسترسی به عنصر جدید
print(dynamic_arr) # [10, 20, 30, 40]
```

مزایا:

- اندازه متغیر و قابلیت تغییر اندازه به طور خودکار.
- استفاده بهینه از حافظه.

معایب:

- هزینه زمانی برای افزایش اندازه در برخی موارد (وقتی آرایه نیاز به بازسازی دارد).
- در صورت داشتن تعداد زیادی عملیات تغییر اندازه، ممکن است کارایی کاهش یابد.

نتیجه‌گیری:

- آرایه یک بعدی برای ذخیره داده‌ها به صورت ساده و پیوسته استفاده می‌شود.
- آرایه دو بعدی برای ذخیره‌سازی داده‌ها در قالب ماتریس یا جدول مناسب است.
- آرایه‌های دینامیک به دلیل تغییر اندازه خودکارشان در بسیاری از زبان‌های برنامه‌نویسی (مثل پایتون) بسیار کاربردی هستند.

لیست‌های پیوندی (Linked Lists)

لیست‌های پیوندی نوعی ساختار داده‌ای خطی هستند که داده‌ها به صورت پیوسته ذخیره می‌شوند، هر عنصر در این لیست‌ها در حافظه به صورت جداگانه ذخیره می‌شود و به عنصر بعدی اشاره می‌کند. این نوع ساختمان داده‌ها به دلیل ویژگی‌های خاکشان در عملیات‌هایی مانند اضافه و حذف داده‌ها سریع‌تر از آرایه‌ها عمل می‌کنند.

لیست‌های پیوندی انواع مختلفی دارند که در اینجا به سه نوع تک‌پیوندی (Singly Linked List)، دوپیوندی (Doubly Linked List) و حلقی (Circular Linked List) پرداخته می‌شود.

۱. لیست پیوندی تک‌پیوندی (Singly Linked List)

تعریف:

در لیست پیوندی تک‌پیوندی، هر عنصر (یا گره) شامل دو بخش است:

- داده: مقدار یا اطلاعاتی که گره ذخیره می‌کند.
- اشارة‌گر (Pointer): که به گره بعدی در لیست اشاره می‌کند.

در این نوع لیست، تنها می‌توان به گره بعدی از هر گره دسترسی پیدا کرد، به همین دلیل این لیست فقط در یک جهت قابل پیمایش است.

ویژگی‌ها:

- عملیات افزودن و حذف گره‌ها به راحتی انجام می‌شود ($O(1)$).
- تنها به گره بعدی اشاره دارد، بنابراین فقط یک طرف لیست قابل پیمایش است.
- دسترسی به هر عنصر با شروع از اولین گره انجام می‌شود.

```
# تعریف کلاس گره (Node)
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# ایجاد گره‌ها و پیوند دادن آنها
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

Edit ⚙ Copy ⌂
```

```

node1.next = node2
node2.next = node3

# پیمایش لیست
current = node1
while current:
    print(current.data)
    current = current.next

```

مزایا:

- استفاده کم از حافظه در مقایسه با آرایه‌ها.
- عملیات‌های اضافه و حذف سریع.

معایب:

- دسترسی به داده‌ها کند است، چون باید از ابتدا شروع کرد و به ترتیب پیش رفت.
- لیست یک‌طرفه است و فقط از ابتدا به انتها پیمایش می‌شود.

۳. لیست پیوندی دوپیوندی (Doubly Linked List)

تعریف:

در لیست پیوندی دوپیوندی، هر گره شامل سه بخش است:

- داده: مقدار ذخیره شده در گره.
- اشاره‌گر قبلی (Prev): اشاره‌گر به گره قبلی.
- اشاره‌گر بعدی (Next): اشاره‌گر به گره بعدی.

این نوع لیست به شما این امکان را می‌دهد که هم از ابتدا به انتها و هم از انتها به ابتدا پیمایش کنید.

ویژگی‌ها:

- هر گره به گره قبلی و گره بعدی اشاره می‌کند.
- می‌توان از هر دو طرف لیست به راحتی پیمایش کرد.
- پیچیدگی بیشتر نسبت به لیست تک‌پیوندی.

```
# تعریف کلاس گره (Node)
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

# ایجاد گرهها و پیوند دادن آنها
node1 = Node(10)
node2 = Node(20)
node3 = Node(30)

node1.next = node2
node2.prev = node1
node2.next = node3
node3.prev = node2

# پیمایش لیست از ابتدا به انتهای آنها
current = node1
while current:
    print(current.data)
    current = current.next

# پیمایش لیست از انتهای آنها به ابتدا
current = node3
while current:
    print(current.data)
    current = current.prev
```

Edit ⚙ Copy ⌂

مزایا:

- پیمایش از هر دو طرف لیست امکان‌پذیر است.
- عملیات‌هایی مانند حذف گره در وسط لیست سریع‌تر از لیست تک‌پیوندی انجام می‌شود.

معایب:

- استفاده بیشتر از حافظه (نیاز به ذخیره دو اشاره‌گر در هر گره).
- پیچیدگی بیشتر در پیاده‌سازی و مدیریت اشاره‌گرها.

۳. لیست پیوندی حلقوی (Circular Linked List)

تعریف:

در لیست پیوندی حلقوی، آخرین گره به اولین گره اشاره می‌کند و بنابراین لیست به صورت حلقه‌ای به هم متصل است. در این نوع لیست می‌توان هم از لیست تک‌پیوندی حلقوی و هم از لیست دوپیوندی حلقوی استفاده کرد.

- در لیست تک‌پیوندی حلقوی، آخرین گره به اولین گره اشاره می‌کند.
- در لیست دوپیوندی حلقوی، آخرین گره به اولین گره اشاره می‌کند و اولین گره به آخرین گره اشاره می‌کند.

ویژگی‌ها:

- لیست به صورت حلقه‌ای است و به طور بی‌پایان قابل پیمایش است.
- در صورت استفاده از لیست دوپیوندی حلقوی، می‌توان به هر دو طرف لیست دسترسی داشت.

```
# تعریف کلاس گره (Node)
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    # ایجاد گره‌ها و پیوند دادن آنها به صورت حلقوی
    node1 = Node(10)
    node2 = Node(20)
    node3 = Node(30)

    node1.next = node2
    node2.next = node3
    node3.next = node1  # ایجاد حلقه
```



```
# پیمایش لیست حلقوی
current = node1
for _ in range(6):  # برای جلوگیری از حلقه بی‌پایان فقط 6 بار پیمایش می‌کنیم
    print(current.data)
    current = current.next
```

مزایا:

- مناسب برای لیست‌های بی‌پایان یا دورانی.
- مناسب برای مدیریت و گردش در داده‌ها به صورت حلقوی (مثلاً در زمان‌بندی و پردازش‌های دایره‌ای).

معايير:

- پیچیدگی بیشتر در مدیریت و پیاده‌سازی.
- ممکن است باعث ایجاد حلقه‌های بی‌پایان در هنگام پیمایش شود.

نتیجه‌گیری:

- لیست پیوندی تک‌پیوندی: ساده‌ترین نوع لیست پیوندی، که فقط یک طرف قابل پیمایش است.
- لیست پیوندی دوپیوندی: برای دسترسی سریع از هر دو طرف، گره‌ها به گره قبلی و بعدی اشاره دارند.
- لیست پیوندی حلقوی: مناسب برای داده‌های دایره‌ای یا بی‌پایان که در آن‌ها پیمایش به صورت حلقه‌ای انجام می‌شود.

پشته (Stack) و کاربردهای آن

پشته (Stack) یکی از ساختمان‌های داده‌ای خطی است که اصولاً برایه اصل آخر وارد، اول خارج (LIFO) عمل می‌کند. یعنی آخرین عنصری که وارد پشته می‌شود، اول از پشته خارج می‌شود. پشته‌ها به دلیل سادگی و کارایی بالا در برخی از الگوریتم‌ها و کاربردها بسیار پرطرفدار هستند.

تعریف پشته (Stack)

پشته یک لیست یا ساختار داده‌ای است که فقط در یک انتهای، به نام بالا (Top) یا سر پشته، عملیات اضافه (Push) و حذف (Pop) انجام می‌شود. در پشته نمی‌توان به سایر عناصر دسترسی داشت، بلکه فقط می‌توان به عنصر بالا (که آخرین وارد شده) دسترسی پیدا کرد.

عملیات‌های اصلی پشته:

1. Push: افزودن یک عنصر به بالای پشته.
2. Pop: حذف عنصر از بالای پشته.
3. Top یا Peek: دسترسی به بالاترین عنصر بدون حذف آن.
4. IsEmpty: بررسی اینکه آیا پشته خالی است یا خیر.
5. Size: اندازه پشته، تعداد عناصر در پشته.

ویژگی‌ها و مزایا:

- ساده بودن پیاده‌سازی: پشته‌ها را می‌توان با استفاده از آرایه یا لیست‌های پیوندی به راحتی پیاده‌سازی کرد.
- عملیات سریع: عملیات‌های Push و Pop در پشته $O(1)$ هستند، یعنی زمان ثابت برای انجام این عملیات‌ها نیاز است.
- حافظه کم مصرف: در مقایسه با ساختمان داده‌های پیچیده‌تر مانند درخت‌ها، پشته‌ها حافظه کمی مصرف می‌کنند.

```
# تعریف کلاس پشته (Stack)
class Stack:
    def __init__(self):
        self.items = []
```

```
# فروخت عنصر به پشته
def push(self, item):
    self.items.append(item)
```

```
# حذف عنصر از پشته
def pop(self):
    if not self.is_empty():
        return self.items.pop()
    else:
        return "پشته خالی است"
```

```
# مشاهده بالاترین عنصر
```

```
def peek(self):
    if not self.is_empty():
        return self.items[-1]
    else:
        return "پشته خالی است"
```

```
# بررسی خالی بودن پشته
```

```
def is_empty(self):
    return len(self.items) == 0
```

```
# گرفتن اندازه پشته
```

```
def size(self):
    return len(self.items)
```

```
# استفاده از پشته
```

```
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
```

```
print(stack.peek()) # 30 : خروجی
print(stack.pop()) # 30 : خروجی
print(stack.pop()) # 20 : خروجی
print(stack.size()) # 1 : خروجی
```

Edit ⌛ Copy ⌂

کاربردهای پشته:

1. مدیریت فراخوانی توابع (Function Call Stack):

در بسیاری از زبان‌های برنامه‌نویسی، پشته برای ذخیره‌سازی اطلاعات مربوط به فراخوانی توابع استفاده می‌شود. هر بار که یک تابع فراخوانی می‌شود، اطلاعات مربوط به آن در پشته ذخیره می‌شود. هنگامی که تابع خاتمه می‌یابد، اطلاعات آن از پشته حذف می‌شود.

2. الگوریتم جستجوی عمق اول (DFS):

الگوریتم جستجوی عمق اول برای پیمایش گراف‌ها یا درخت‌ها از پشته استفاده می‌کند. در این الگوریتم، گره‌ها به ترتیب به پشته اضافه می‌شوند و هنگامی که یک گره از پشته خارج می‌شود، گره‌های همسایه آن بررسی می‌شوند.

3. بررسی بالانس پرانتزها:

برای بررسی اینکه آیا پرانتزهای یک رشته به درستی باز و بسته شده‌اند، می‌توان از پشته استفاده کرد. هر بار که پرانتز باز (یا کروشه و آکولاد) دیده می‌شود، آن را در پشته ذخیره می‌کنیم. هنگامی که پرانتز بسته (یا کروشه و آکولاد) می‌بینیم، بررسی می‌کنیم که آیا با پرانتز باز مطابقت دارد.

4. پسوند نویسی (Postfix Expression):

در ارزیابی عبارت‌های پسوندی (یا Polish)، که در آن عملگر بعد از عملوند‌ها قرار دارد، از پشته برای ذخیره‌سازی عملوند‌ها و عملگرها استفاده می‌شود. در این روش، عملیات‌ها پس از دیدن عملوند‌ها انجام می‌شود.

5. مسیریابی برگشتی (Backtracking):

در الگوریتم‌هایی که از روش برگشتی استفاده می‌کنند (مانند حل مسائل مختلف در پازل‌ها یا مسائل مشابه)، از پشته برای ذخیره‌سازی وضعیت‌های مختلف استفاده می‌شود. این کار به ما اجازه می‌دهد تا در صورت لزوم به عقب برگردیم و تصمیمات متفاوتی بگیریم.

6. الگوریتم تبدیل بین نمایه‌های مختلف (Infix to Postfix Conversion):

در تبدیل عبارت‌های معمولی (Infix) به پسوندی (Postfix)، از پشته برای ذخیره عملگرها استفاده می‌شود و ترتیب عملگرها به دقت حفظ می‌شود.

مزایا و معایب پشته:

مزایا:

- ساده و سریع: عملیات‌های اصلی پشته بسیار سریع و با پیچیدگی زمانی $O(1)$ انجام می‌شوند.
- حافظه بهینه: پشته‌ها برای انجام عملیات‌هایی مانند پیاده‌سازی الگوریتم‌های DFS یا ذخیره‌سازی توابع بسیار کارآمد هستند.

معايير:

- دسترسی محدود: تنها می‌توان به عنصر بالای پشته دسترسی داشت، بنابراین برای دسترسی به عناصر پایین‌تر از بالاترین عنصر باید تمامی عناصر بالاتر را حذف کرد.
- حافظه محدود: اگر به طور مداوم از پشته استفاده شود و ظرفیت آن پر شود، احتمال ایجاد مشکل در مدیریت حافظه وجود دارد.

نتیجه‌گیری:

پشته یک ساختار داده ساده، سریع و کارآمد است که در بسیاری از کاربردها مانند مدیریت توابع، پیمایش گراف‌ها و ارزیابی عبارات استفاده می‌شود. با توجه به ویژگی‌های خاص خود، پشته در الگوریتم‌ها و پیاده‌سازی‌های مختلف از جمله الگوریتم‌های جستجو، مسیریابی برگشتی و بررسی درست بودن پرانتزها کاربرد دارد.

صف (Queue) و انواع آن

صف (Queue) یک دیگر از ساختمان‌های داده‌ای خطی است که بر اساس اصل اول وارد، اول خارج (FIFO) عمل می‌کند. یعنی اولین عنصری که وارد صف می‌شود، اولین عنصری است که از صف خارج می‌شود. صفات در بسیاری از الگوریتم‌ها و پیاده‌سازی‌ها از جمله در مدیریت پردازش‌ها، نوبت‌دهی و شبکه‌ها کاربرد دارند.

تعريف صفت (Queue)

صف یک ساختار داده‌ای است که عملگرهای اصلی افزودن عنصر به انتهای صفت (enqueue) و حذف عنصر از ابتدای صفت (dequeue) را انجام می‌دهد. در صفات، همواره اضافه کردن داده‌ها از انتهای صفت و حذف داده‌ها از ابتدای صفت انجام می‌شود. این ویژگی باعث می‌شود که صفات در برخی الگوریتم‌ها بسیار مفید باشند.

عملیات‌های اصلی صفت:

1. Enqueue: افزودن یک عنصر به انتهای صفت.

2. Dequeue: حذف و برگرداندن عنصر از ابتدای صفت.

3. Front/Peak: دسترسی به اولین عنصر صفت بدون حذف آن.

4. IsEmpty: بررسی اینکه آیا صفت خالی است یا خیر.

5. Size: اندازه صفت، تعداد عناصر موجود در صفت.

```
# تعريف كلاس صف (Queue)
class Queue:
    def __init__(self):
        self.items = []

    # إضافة عنصر إلى صف
    def enqueue(self, item):
        self.items.append(item)
```

```

# حذف عنصر از صف
def dequeue(self):
    if not self.is_empty():
        return self.items.pop(0)
    else:
        return "صف خالی است"

```

مشاهده اولین عنصر صف

```

def front(self):
    if not self.is_empty():
        return self.items[0]
    else:
        return "صف خالی است"

```

بررسی خالی بودن صف

```

def is_empty(self):
    return len(self.items) == 0

```



گرفتن اندازه صف

```

def size(self):
    return len(self.items)

```

استفاده از صف

```

queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print(queue.front()) # 10
print(queue.dequeue()) # 10
print(queue.size()) # 2

```

انواع صفات:

1. صف معمولی (Simple Queue)

در صف معمولی، عناصر به ترتیب وارد می‌شوند و به ترتیب نیز از صف خارج می‌شوند. این ساده‌ترین نوع صف است و همانطور که گفته شد، عملیات‌های افزودن و حذف در این صف بر اساس اصل FIFO انجام می‌شود.

ویژگی‌ها:

- افزودن از انتهای و حذف از ابتدای صف انجام می‌شود.
- این صف در بسیاری از کاربردهای عمومی مانند صف انتظار پردازش‌ها یا درخواست‌ها استفاده می‌شود.

مثال:

- استفاده در مدیریت پردازش‌ها، مانند سیستم‌عامل‌ها برای مدیریت نوبت‌های پردازش.
- 2.** صف دوطرفه (Deque) یا Double Ended Queue

در صف دوطرفه یا Deque (که از اختصار Double-Ended Queue گرفته شده است)، امکان افزودن و حذف عناصر از هر دو طرف وجود دارد. به این ترتیب، هم می‌توان از ابتدای صف (از چپ) و هم از انتهای صف (از راست) داده‌ها را اضافه یا حذف کرد.

ویژگی‌ها:

- امکان انجام عملیات enqueue و dequeue از هر دو طرف صف.
- می‌توان از هر طرف صف، داده‌ها را اضافه یا حذف کرد.
- به طور کلی، صف دوطرفه از دو نوع عملیات ورودی/خروجی پشتیبانی می‌کند: از جلو (front) و از انتهای (back).

مثال‌ها:

- استفاده در میزان اولویت پردازش‌ها یا حلقه‌های گردش کار که نیاز به دسترسی سریع به هر دو طرف صف دارند.

عملیات‌های اضافی:

- افزودن عنصر به ابتدای صف.: add_front()
- افزودن عنصر به انتهای صف.: add_back()
- حذف عنصر از ابتدای صف.: remove_front()
- حذف عنصر از انتهای صف.: remove_back()

```
from collections import deque

# استفاده از Deque
queue = deque()

# افزودن به جلو و عقب #
queue.append(10) # افزودن به انتهای صف
queue.appendleft(20) # افزودن به ابتدای صف
```

حذف از جلو و عقب #

خروجی: 20 (حذف از ابتدای صف) #

خروجی: 10 (حذف از انتهای صف) #

3. صف اولویت‌دار (Priority Queue):

صف اولویت‌دار صفتی است که در آن هر عنصر دارای یک اولویت است. برخلاف صف معمولی که ترتیب خروجی به ترتیب ورود است، در صف اولویت‌دار عنصر با بالاترین اولویت ابتدا از صف خارج می‌شود. در این نوع صف، عملیات `enqueue` به گونه‌ای انجام می‌شود که عناصر بر اساس اولویتشان مرتب شوند.

ویژگی‌ها:

- هر عنصر یک اولویت دارد و عنصر با بالاترین اولویت در ابتدا خارج می‌شود.
- در این صف، ترتیب ورود عنصر برای خروج مهم نیست، بلکه اولویت آن مهم است.
- از صف اولویت‌دار برای پیاده‌سازی الگوریتم‌هایی مانند الگوریتم دیکسترا (Dijkstra) و خدمات زمان‌بندی در سیستم‌عامل‌ها استفاده می‌شود.

عملیات‌های اصلی صف اولویت‌دار:

1. `Insert`: افزودن یک عنصر با اولویت به صف.

2. `ExtractMin`: خارج کردن عنصر با کمترین اولویت (در برخی پیاده‌سازی‌ها).

3. `ExtractMax`: خارج کردن عنصر با بیشترین اولویت (در برخی پیاده‌سازی‌ها).

4. `Peek`: مشاهده بالاترین اولویت.

```
import heapq

# صف اولویت‌دار با استفاده از heapq
priority_queue = []

# افزودن عناصر به صف با اولویت
heapq.heappush(priority_queue, (2, "Task 2"))
heapq.heappush(priority_queue, (1, "Task 1"))
heapq.heappush(priority_queue, (3, "Task 3"))

# در اینجا) حذف عنصر با کمترین اولویت
print(heapq.heappop(priority_queue)) # خروجی 1, 'Task 1'
```

مزایا:

- مناسب برای مدیریت منابع و الگوریتم‌هایی مانند دیکسترا که در آن‌ها ترتیب پردازش اهمیت دارد.
- عملیات استخراج عنصر با اولویت بالا سریع است ($O(\log n)$).

معایب:

- ممکن است نسبت به صفات معمولی پیچیدگی بیشتری داشته باشد.
- عملیات Insert و Extract هزینه زمانی دارند.

نتیجه‌گیری:

- صفات معمولی (Normal Queue): استفاده ساده از اصول FIFO و مناسب برای کاربردهایی که پردازش داده‌ها به ترتیب ورود نیاز دارند.
- صفات دوطرفه (Deque): قابلیت انجام عملیات افزودن و حذف از هر دو طرف صفات، مناسب برای کاربردهایی که نیاز به دسترسی به هر دو طرف دارند.
- صفات اولویت‌دار (Priority Queue): استفاده از اولویت در پردازش داده‌ها، مناسب برای مدیریت منابع و الگوریتم‌های خاص مانند دیکسترا.

ساختمان‌های داده‌ای غیرخطی

ساختمان‌های داده‌ای غیرخطی به ساختمان‌هایی اطلاق می‌شود که در آن‌ها داده‌ها به صورت غیرخطی و با روابط پیچیده‌تر سازمان‌دهی می‌شوند. در این ساختمان‌های داده‌ای، برخلاف ساختمان‌های داده‌ای خطی (مانند آرایه‌ها و صفات)، دسترسی به داده‌ها بر اساس روابط پیچیده‌تری صورت می‌گیرد و می‌توان داده‌ها را از چندین جهت مختلف پیمایش کرد.

(Trees) درخت‌ها

درخت‌ها یکی از ساختارهای داده‌ای غیرخطی هستند که از مجموعه‌ای از گره‌ها (Nodes) و یال‌ها (Edges) تشکیل شده‌اند. درخت‌ها برای نمایش روابط سلسله‌مراتبی بسیار مناسب هستند. در این نوع ساختار داده‌ای، هر گره می‌تواند به دیگر گره‌ها متصل شود و به صورت یک ساختار درختی مرتب می‌شود.

ویژگی‌ها:

1. **ریشه (Root):**
 - گره ابتدایی در درخت که از آن گره‌ها منشعب می‌شوند.
2. **گره‌ها (Nodes):**
 - هر گره می‌تواند یک داده را ذخیره کند و به گره‌های دیگری متصل شود.

3. یال‌ها : (Edges)

- ارتباط بین گره‌ها، که از گره والد به گره فرزند می‌رود.

4. گره والد : (Parent)

- گره‌ای که به گره‌های دیگر (فرزنده) متصل است.

5. گره فرزند : (Child)

- گره‌ای که از یک گره والد منشعب می‌شود.

6. گره برگ : (Leaf)

- گره‌ای که هیچ فرزند ندارند.

7. ارتفاع : (Height)

- طول بزرگترین مسیر از ریشه به یک برگ.

8. عمق : (Depth)

- فاصله یک گره از ریشه.

انواع درخت‌ها :

1. درخت دودویی : (Binary Tree)

- هر گره در این درخت می‌تواند حداقل دو فرزند داشته باشد. درخت دودویی معمول‌ترین نوع درخت است.

2. درخت جستجوی دودویی : (Binary Search Tree - BST)

- در این درخت، گره‌های چپ دارای مقادیر کوچک‌تر از گره والد و گره‌های راست دارای مقادیر بزرگ‌تر از گره والد هستند.

3. درخت AVL

- نوعی از درخت‌های جستجوی دودویی متعادل است که در آن تفاوت ارتفاع بین فرزندان چپ و راست هر گره از 1 بیشتر نمی‌شود.

4. درخت B

- درختی متعادل است که معمولاً برای ذخیره‌سازی داده‌ها در پایگاه داده‌ها استفاده می‌شود. این درخت به گونه‌ای طراحی شده که بهینه برای ذخیره‌سازی داده‌ها در دیسک باشد.

5. درخت Heap:

- درختی که به طور خاص برای مدیریت صفات اولویت دار طراحی شده است. در درخت Heap، درخت باید ویژگی خاصی از ترتیب را رعایت کند: در درخت مین-Heap، مقدار هر گره باید کوچکتر یا مساوی مقادیر فرزندانش باشد.

عملیات‌های اصلی در درخت‌ها:

1. جستجو (Search):

- برای یافتن یک گره خاص در درخت.

2. افزودن (Insert):

- برای افزودن یک گره جدید به درخت.

3. حذف (Delete):

- برای حذف یک گره از درخت.

4. پیمایش (Traversal):

- به معنای عبور از تمامی گره‌ها در درخت. سه نوع اصلی پیمایش وجود دارد:
 - پیمایش پیش‌درختی (Pre-order): ابتدا گره والد، سپس گره‌های فرزند چپ و راست.
 - پیمایش میان‌درختی (In-order): ابتدا گره فرزند چپ، سپس گره والد، و بعد گره فرزند راست.
 - پیمایش پس‌درختی (Post-order): ابتدا گره‌های فرزند چپ و راست، سپس گره والد.

```
class Node:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.value = key  
  
# پیمایش پیش‌درختی  
def preorder_traversal(root):  
    if root:  
        print(root.value, end=" ")  
        preorder_traversal(root.left)  
        preorder_traversal(root.right)
```

```
# بیانش میان درختی
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=" ")
        inorder_traversal(root.right)
```

```
# بیانش پس درختی
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.value, end=" ")
```



```
# ساخت درخت
root = Node(10)
root.left = Node(20)
root.right = Node(30)
root.left.left = Node(40)
root.left.right = Node(50)

# بیانش درخت
print("Preorder Traversal:")
preorder_traversal(root) # 30 50 40 20 10
```

```
print("\nInorder Traversal:")
inorder_traversal(root) # 30 10 50 20 40
```

```
print("\nPostorder Traversal:")
postorder_traversal(root) # 10 30 20 50 40
```



مزایا و کاربردها:

- درخت‌ها برای جستجوی سریع در مجموعه داده‌ها استفاده می‌شوند.
- به ویژه در پایگاه داده‌ها و سیستم‌های فایل برای سازمان‌دهی داده‌ها به کار می‌روند.
- درخت‌ها برای حل مسائل سلسله‌مراتبی (مانند ساختارهای فایل سیستم‌ها) یا الگوریتم‌های جستجو (مانند جستجوی دودویی) بسیار مفید هستند.

درخت دودویی (Binary Tree)

درخت دودویی یکی از انواع درخت‌ها است که در آن هر گره می‌تواند حداکثر دو فرزند داشته باشد. این ویژگی درخت دودویی را برای الگوریتم‌های جستجو و ذخیره‌سازی داده‌ها به یک ساختار بسیار مفید تبدیل کرده است. درخت‌های دودویی می‌توانند برای ذخیره‌سازی و جستجو به صورت مؤثر استفاده شوند و نوعی سلسله‌مراتب از داده‌ها را نمایان کنند.

ویژگی‌های درخت دودویی:

1. هر گره دارای حداکثر دو فرزند است:

- به این معنی که هر گره در درخت دودویی یا هیچ فرزندی ندارد، یا یک فرزند چپ دارد یا یک فرزند راست یا هر دو را دارا است.

2. گره ریشه (Root):

- گره ابتدایی درخت است که از آن تمام گره‌های دیگر به آن متصل می‌شوند.

3. گره‌های برگ (Leaf):

- گره‌هایی که هیچ فرزند ندارند. این‌ها گره‌های انتهایی در درخت هستند.

4. گره والد (Parent):

- گره‌ای که به گره‌های فرزند خود متصل است.

5. گره فرزند (Child):

- گره‌ای که از یک گره والد منشعب می‌شود.

انواع درخت دودویی:

1. درخت دودویی جستجو (Binary Search Tree - BST):

- در این درخت، برای هر گره:

- گره‌های زیرشاخه چپ (چپ‌تر) باید مقادیر کوچک‌تر از گره والد داشته باشند.
- گره‌های زیرشاخه راست (راست‌تر) باید مقادیر بزرگ‌تر از گره والد داشته باشند.

2. درخت دودویی متوازن (Balanced Binary Tree):

- در این نوع درخت، تفاوت ارتفاع زیر درخت‌های چپ و راست از هر گره نباید بیشتر از یک باشد.

3. درخت دودویی کامل (Full Binary Tree):

- در این درخت، هر گره به جز برگ‌ها باید دقیقاً دو فرزند داشته باشد.

4. درخت دودویی پر (Complete Binary Tree):

- در این درخت، تمامی گره‌های سطح‌های بالاتر به طور کامل پر هستند و در سطح آخر گره‌ها باید از چپ به راست پر شوند.

عملیات‌های اصلی در درخت دودویی:

1. جستجو (Search):

- جستجو در درخت دودویی با استفاده از ویژگی‌های خاص آن انجام می‌شود. در درخت دودویی جستجو، برای هر گره:

- اگر مقدار مورد نظر از گره والد کمتر باشد، جستجو در زیرشاخه چپ ادامه می‌یابد.
- اگر مقدار مورد نظر بزرگتر از گره والد باشد، جستجو در زیرشاخه راست ادامه می‌یابد.

2. افزودن (Insert):

- درخت دودویی جستجو به گونه‌ای طراحی شده است که برای افزودن یک گره جدید، ابتدا از ریشه شروع کرده و با استفاده از قوانین جستجو به محل مناسب برای افزودن گره می‌رسیم.

3. حذف (Delete):

- حذف یک گره در درخت دودویی شامل موارد مختلفی است:
- اگر گره حذف شده هیچ فرزند نداشته باشد، به راحتی از درخت حذف می‌شود.
- اگر گره یک فرزند داشته باشد، فرزند آن جایگزین گره حذف شده می‌شود.
- اگر گره دو فرزند داشته باشد، می‌توان از روش‌های مختلفی استفاده کرد (مثلًا جایگزینی با بزرگترین گره در زیرشاخه چپ یا کوچکترین گره در زیرشاخه راست).

4. پیمایش (Traversal):

- پیمایش در درخت دودویی به سه صورت اصلی انجام می‌شود:

1. پیمایش پیش‌درختی (Pre-order Traversal): ابتدا گره والد، سپس گره‌های فرزند چپ و راست.

2. پیمایش میان‌درختی (In-order Traversal): ابتدا گره فرزند چپ، سپس گره والد، و سپس گره فرزند راست. (برای درخت‌های دودویی جستجو این نوع پیمایش داده‌ها را به ترتیب مرتب می‌کند).

3. پیمایش پس‌درختی (Post-order Traversal): ابتدا گره هـ فرزند چپ و راست، سپس گره والد.

```

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key

# پیمایش پیش درختی
def preorder_traversal(root):
    if root:
        print(root.value, end=" ")
        preorder_traversal(root.left)
        preorder_traversal(root.right)

# پیمایش میان درختی
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=" ")
        inorder_traversal(root.right)

# پیمایش پس درختی
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.value, end=" ")

# اضافه کردن گره به درخت دودویی جستجو
def insert(root, key):
    # اگر درخت خالی است، یک گره جدید می‌سازیم
    if root is None:
        return Node(key)
    # درخت دودویی جستجو: اگر مقدار کمتر از گره والد باشد به سمت چپ می‌رویم
    if key < root.value:
        root.left = insert(root.left, key)
    else:
        # اگر مقدار بزرگتر از گره والد باشد به سمت راست می‌رویم
        root.right = insert(root.right, key)
    return root

```

```
# جستجو در درخت
def search(root, key):
    # اگر درخت خالی بود یا مقدار پیدا شد
    if root is None or root.value == key:
        return root
    # اگر مقدار جستجو شده کمتر از گره والد باشد، جستجو در زیرشاخه چپ ادامه می‌یابد
    if key < root.value:
        return search(root.left, key)
    # اگر مقدار جستجو شده بیشتر از گره والد باشد، جستجو در زیرشاخه راست ادامه می‌یابد
    return search(root.right, key)
```

```
# ساخت درخت دودویی جستجو
root = Node(50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)
```

```
# پیمایش‌ها
print("Preorder Traversal:")
preorder_traversal(root) # خروجی: 50 30 20 40 70 60 80
```

```
print("\nInorder Traversal:")
inorder_traversal(root) # خروجی: 20 30 40 50 60 70 80
```

```
print("\nPostorder Traversal:")
postorder_traversal(root) # خروجی: 20 40 30 60 70 80 50
```

```
# جستجو در درخت
key = 60
found_node = search(root, key)
if found_node:
    print(f"\n{key} یافت شد")
else:
    print(f"\n{key} در درخت یافت نشد")
```

Edit ⚙ Copy ☉

Edit ⚙ Copy ☉

مزایای درخت دودویی:

1. عملیات جستجو، اضافه کردن و حذف به صورت کارآمد انجام می‌شوند، به خصوص در درخت‌های دودویی جستجو.
2. پیمایش‌ها در درخت دودویی معمولاً ساده هستند و می‌توانند به ترتیب خاصی از داده‌ها دسترسی پیدا کنند.
3. درخت‌های دودویی می‌توانند برای مدیریت داده‌ها به ویژه در سیستم‌های فایل، پایگاه‌های داده، و الگوریتم‌های جستجو استفاده شوند.

درخت جستجوی دودویی (Binary Search Tree - BST)

درخت جستجوی دودویی (BST) یک نوع خاص از درخت دودویی است که برای جستجو، افزودن و حذف داده‌ها به صورت مؤثر طراحی شده است. در این نوع درخت، برای هر گره، ویژگی خاصی وجود دارد که باعث می‌شود جستجو و دسترسی به داده‌ها در آن بسیار سریع باشد. این ویژگی‌ها باعث می‌شوند که درخت جستجوی دودویی در بسیاری از الگوریتم‌های جستجو و ذخیره‌سازی داده‌ها استفاده شود.

ویژگی‌های درخت جستجوی دودویی (BST):

1. ترتیب خاص گره‌ها:

- برای هر گره، تمامی مقادیر در زیر درخت چپ آن باید کوچکتر از مقدار گره والد باشند.
- تمامی مقادیر در زیر درخت راست آن باید بزرگ‌تر از مقدار گره والد باشند.

2. عملیات سریع:

- جستجو، افزودن و حذف گره‌ها در درخت جستجوی دودویی به طور معمول با پیچیدگی زمانی $O(\log n)$ انجام می‌شود، که در صورت متوازن بودن درخت، این زمان بسیار کارآمد است.

3. درخت متوازن:

- اگر درخت جستجوی دودویی متوازن باشد (یعنی ارتفاع درخت کم باشد)، عملیات‌های مختلف (جستجو، افزودن، حذف) زمان خطی کمتری خواهند داشت. اگر درخت متوازن نباشد (برای مثال به صورت یک لیست درآید)، زمان عملیات‌ها ممکن است به $O(n)$ برسد.

عملیات‌های اصلی در درخت جستجوی دودویی:

1. جستجو (Search):

- جستجو در درخت جستجوی دودویی با استفاده از مقایسه مقدار گره با مقدار مورد نظر انجام می‌شود.
- اگر مقدار مورد نظر کوچک‌تر از مقدار گره باشد، جستجو در زیر درخت چپ ادامه می‌یابد.
- اگر مقدار مورد نظر بزرگ‌تر از مقدار گره باشد، جستجو در زیر درخت راست ادامه می‌یابد.
- این فرآیند تا زمانی که گره پیدا شود یا به انتهای درخت برسیم، ادامه می‌یابد.

2. افزودن (Insert)

- افزودن گره جدید به درخت جستجوی دودویی مشابه عملیات جستجو است. ابتدا جستجو می‌کنیم تا محل مناسب برای افزودن گره را پیدا کنیم.
- اگر مقدار جدید کوچکتر از گره جاری باشد، به سمت زیر درخت چپ حرکت می‌کنیم و اگر بزرگتر باشد، به سمت زیر درخت راست حرکت می‌کنیم.
- این عملیات تا رسیدن به گرهی که فرزند نداشته باشد ادامه می‌یابد و گره جدید به عنوان فرزند آن گره اضافه می‌شود.

3. حذف (Delete)

- حذف گره از درخت جستجوی دودویی می‌تواند به سه صورت انجام شود:
1. حذف گره بدون فرزند (Leaf): در این حالت، فقط گره حذف شده را از درخت حذف می‌کنیم.
 2. حذف گره با یک فرزند: گره حذف شده با فرزندش جایگزین می‌شود.
 3. حذف گره با دو فرزند: در این حالت، گره باید با بزرگترین گره در زیر درخت چپ یا کوچکترین گره در زیر درخت راست جایگزین شود.

4. پیمایش (Traversal)

- پیمایش در درخت جستجوی دودویی می‌تواند به سه صورت مختلف انجام شود:
- پیمایش پیش‌درختی (Pre-order): ابتدا گره والد، سپس گره‌های فرزند چپ و راست.
- پیمایش میان‌درختی (In-order): ابتدا گره فرزند چپ، سپس گره والد، سپس گره فرزند راست. (در درخت جستجوی دودویی این پیمایش داده‌ها را به ترتیب صعودی مرتب می‌کند).
- پیمایش پس‌درختی (Post-order): ابتدا گره‌های فرزند چپ و راست، سپس گره والد.

```
class Node:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.value = key
```

```

# جستجو در درخت جستجوی دودویی
def search(root, key):
    # اگر درخت خالی بود یا مقدار پیدا شد
    if root is None or root.value == key:
        return root
    # اگر مقدار جستجو شده کوچکتر از گره والد باشد، جستجو در زیرشاخه چپ ادامه می‌یابد
    if key < root.value:
        return search(root.left, key)
    # اگر مقدار جستجو شده بزرگتر از گره والد باشد، جستجو در زیرشاخه راست ادامه می‌یابد
    return search(root.right, key)

```

```

# افزودن گره به درخت جستجوی دودویی
def insert(root, key):
    # اگر درخت خالی است، یک گره جدید می‌سازیم
    if root is None:
        return Node(key)
    # اگر مقدار کوچکتر از گره والد باشد، به سمت چپ می‌رویم
    if key < root.value:
        root.left = insert(root.left, key)
    else:
        # اگر مقدار بزرگتر از گره والد باشد، به سمت راست می‌رویم
        root.right = insert(root.right, key)
    return root

```

```

# بیان میان درختی
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.value, end=" ")
        inorder_traversal(root.right)

```

Edit ⌚ Copy ⌂

```

# حذف گره از درخت جستجوی دودویی
def delete_node(root, key):
    # اگر درخت خالی بود
    if root is None:
        return root

```

Edit ⌚ Copy ⌂

```

جستجو برای پیدا کردن گره که میخواهیم حذف کنیم #
if key < root.value:
    root.left = delete_node(root.left, key)
elif key > root.value:
    root.right = delete_node(root.right, key)
else:
    #اگر گره با یک یا دو فرزند پیدا شود
    # گره بدون فرزند (Leaf)
    if root.left is None and root.right is None:
        return None
    # گره با یک فرزند
    elif root.left is None:
        return root.right
    elif root.right is None:
        return root.left
    # گره با دو فرزند
    else:

```

Edit ⚙ Copy ☉

```

پیدا کردن کوچکترین گره در زیر درخت راست #
min_node = min_value_node(root.right)
root.value = min_node.value
root.right = delete_node(root.right, min_node.value)

return root

```

Edit ⚙ Copy ☉

```

#پیدا کردن کوچکترین گره در زیر درخت
def min_value_node(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

```

```

ساخت درخت جستجوی دودویی #
root = Node(50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

```

```

# پیمایش میان درختی (In-order Traversal)
print("Inorder Traversal:")
inorder_traversal(root) # 80 70 60 50 40 30 20
خروجی: 80 70 60 50 40 30 20

# حذف گره
root = delete_node(root, 20)
print("\nInorder Traversal after deleting 20:")
inorder_traversal(root) # 80 70 60 50 40 30
خرجی: 80 70 60 50 40 30

```

مزایای درخت جستجوی دودویی (BST):

1. عملیات سریع: در درخت های متوازن، جستجو، افزودن و حذف با پیچیدگی زمانی $O(\log n)$ انجام می شود.
2. ساختار مرتب: درخت جستجوی دودویی به طور خودکار داده ها را مرتب نگه می دارد.
3. پیمایش آسان: به راحتی می توان داده ها را به صورت مرتب پیمایش کرد (به ویژه در پیمایش میان درختی).

معایب درخت جستجوی دودویی:

1. عدم تعادل: اگر درخت به طور نامتعادل رشد کند (مانند اضافه کردن داده ها به ترتیب صعودی)، عملیات ها ممکن است به پیچیدگی زمانی $O(n)$ برسد.
2. نیاز به تعادل: برای اطمینان از زمان بهینه برای عملیات ها، باید درخت به طور متوازن باقی بماند. در غیر این صورت، عملکرد می تواند کاهش یابد.

درخت متوازن (Balanced Tree)

درخت متوازن نوعی درخت دودویی است که در آن ارتفاع زیر درخت های چپ و راست در هر گره تقریباً برابر است. این ویژگی باعث می شود که عملکرد عملیات هایی مانند جستجو، درج، حذف و پیمایش بهینه باشد و در $O(\log n)$ باقی بماند.

در صورتی که درخت نامتعادل باشد (مثلاً همه گره ها در یک طرف باشند)، درخت شبیه به یک لیست پیوندی شده و پیچیدگی عملیات ها به $O(n)$ افزایش می یابد. درخت های متوازن این مشکل را حل می کنند.

تعریف درخت متوازن

درختی متوازن است اگر تفاوت ارتفاع زیر درخت های چپ و راست هر گره حداقل ۱ باشد. یعنی برای هر گره:

$$1 \geq |height(right_subtree) - height(left_subtree)|$$

انواع درخت‌های متوازن

چندین نوع درخت متوازن وجود دارد که رایج‌ترین آن‌ها عبارتند از:

1. درخت AVL

2. درخت Red-Black

3. درخت B و +B

4. درخت 2-3 و 2-3-4

درخت B و درخت +B

درخت‌های B و +B نوعی درخت جستجوی متوازن هستند که در پایگاه داده‌ها و سیستم‌های ذخیره‌سازی فایل استفاده می‌شوند. این درخت‌ها برای مدیریت داده‌های حجمی در حافظه دیسک بهینه شده‌اند، زیرا عملیات خواندن و نوشتن در آن‌ها حداقل تعداد دسترسی به دیسک را نیاز دارد.

درخت B (B-Tree)

درخت B یک درخت خودمتعادل (Self-Balancing) است که می‌تواند بیش از دو فرزند برای هر گره داشته باشد. این درخت به گونه‌ای طراحی شده است که حداقل میزان جابجایی داده‌ها بین حافظه اصلی و حافظه جانبی (هارد دیسک) را تضمین کند.

ویژگی‌های درخت B

1. هر گره می‌تواند چندین کلید و چندین فرزند فرزند داشته باشد (برخلاف درخت دودویی که هر گره حداقل دو فرزند دارد).

2. هر گره دارای حداقل $\lceil m/2 \rceil$ و حداقل m فرزند است (m : درجه درخت).

3. برگ‌ها هم‌سطح هستند (یعنی ارتفاع درخت همیشه متداول است).

4. عملیات درج و حذف باعث چرخش (Rotation) یا تقسیم گره (Splitting) می‌شود تا تعادل حفظ شود.

5. پیچیدگی زمانی جستجو، درج و حذف: $O(\log n)$

ساختار درخت B

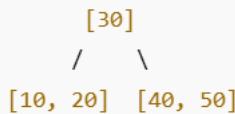
هر گره درخت B شامل:

• n کلید (به صورت مرتب شده).

• $n+1$ اشاره‌گر (فرزنده) که به زیر درخت‌های مربوطه اشاره می‌کنند.

♦ مثال: درخت B درجه 3 ($m = 3$)

هر گره می‌تواند حداقل 2 کلید و حداقل 1 کلید داشته باشد.



Edit ⌗ Copy ⌘

عملیات در درخت B

1. درج (Insertion)

- ابتدا گره مناسب برای درج مقدار جدید پیدا می‌شود.
- مقدار در مکان مناسب قرار داده می‌شود.
- اگر گره پر شود، گره تقسیم (Split) می‌شود و مقدار میانی به گره والد منتقل می‌شود.

2. حذف (Deletion)

- مقدار حذف می‌شود و تعادل درخت حفظ می‌شود.
- اگر تعداد کلیدهای گره کمتر از حداقل باشد، از گره مجاور وام گرفته می‌شود یا گره‌ها ادغام (Merge) می‌شوند.

```
class BTREENode:  
    def __init__(self, leaf=False):  
        self.leaf = leaf  
        self.keys = []  
        self.children = []  
  
class BTREE:  
    def __init__(self, t): # t: درجه درخت  
        self.root = BTREENode(True)  
        self.t = t # گره حداقل تعداد فرزندان یک  
  
    def traverse(self, node=None):  
        if node is None:  
            node = self.root  
        for i in range(len(node.keys)):  
            if not node.leaf:  
                self.traverse(node.children[i])  
            print(node.keys[i], end=" ")  
        if not node.leaf:  
            self.traverse(node.children[-1])
```

```
# نمونه استفاده
btree = BTree(3) # درخت B درجه 3 با
```

درخت B+ (B+ Tree)

درخت B+ نسخه توسعه یافته درخت B است که بهینه‌تر برای جستجو و پیمایش ترتیبی (Sequential Access) طراحی شده است.

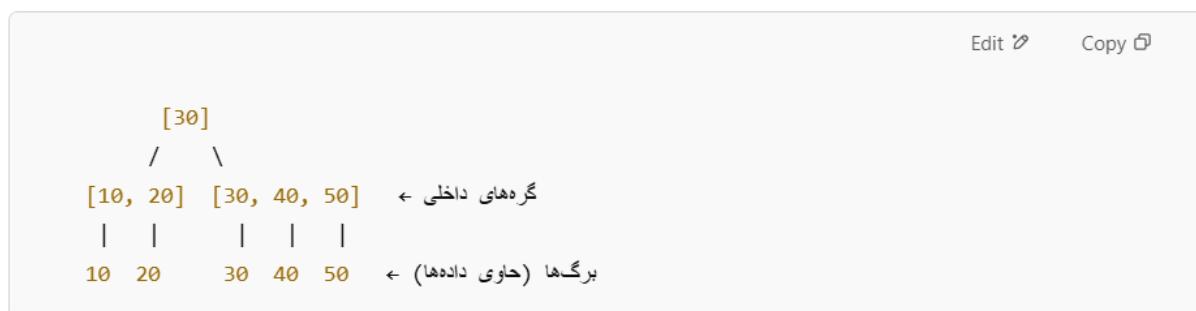
تفاوت‌های درخت B و B+

| درخت B+ | درخت B | ویژگی |
|---|--------------------------|---------------------------|
| فقط در گره‌های برگ | در هر دو گره داخلی و برگ | ذخیره‌سازی کلیدها |
| بیشتر است | کمتر است | تعداد کلیدها در گره داخلی |
| بسیار آسان است (لیست پیوندی بین برگ‌ها) | سخت‌تر است | پیمایش ترتیبی |
| بسیار بهینه‌تر است | بهینه است | بهینه‌سازی برای جستجو |

ساختار درخت B+

1. گره‌های داخلی فقط حاوی کلیدها هستند و داده‌ای ندارند.
2. داده‌ها فقط در گره‌های برگ ذخیره می‌شوند.
3. برگ‌ها به صورت لیست پیوندی به هم متصل هستند (پیمایش سریع‌تر).

♦ مثال درخت B+(درجه 3) :



مزایای درخت B+

- ✓ بهینه برای خواندن از دیسک (به دلیل پیمایش آسان در برگ‌ها).
- ✓ جستجو سریع‌تر از درخت B است (چون کلیدها فقط در برگ‌ها ذخیره می‌شوند).
- ✓ دسترسی ترتیبی ساده است (چون برگ‌ها به صورت لیست پیوندی متصل‌اند).

+ مقایسه درخت B و B+

| درخت B+ | درخت B | معیار |
|-------------------------------------|------------|---------------------------|
| سریع‌تر | سریع | کارایی جستجو |
| ساده‌تر است | سخت‌تر است | حذف داده |
| آسان است (به دلیل پیوند بین برگ‌ها) | سخت‌تر است | پیمایش ترتیبی |
| بیشتر استفاده می‌شود | بله | استفاده در پایگاه داده‌ها |

+ کاربردهای درخت B و B+

- ✓ سیستم‌های مدیریت پایگاه داده (DBMS) مثل MySQL و PostgreSQL
 - ✓ سیستم‌های فایل (File Systems) مانند NTFS و EXT4
 - ✓ سیستم‌های ذخیره‌سازی داده در دیسک‌ها و SSD
 - ✓ ساختارهای نمایه‌سازی (Indexing) در جستجوی سریع اطلاعات
- جمع‌بندی

- ◆ درخت B و B+ از مهم‌ترین ساختمان‌های داده‌ای در پایگاه داده‌ها و سیستم‌های ذخیره‌سازی هستند.
- ◆ درخت B متوازن است و کلیدها را هم در گره‌های داخلی و هم در برگ‌ها ذخیره می‌کند.
- ◆ درخت B+ بهینه‌تر از درخت B است، زیرا داده‌ها فقط در برگ‌ها ذخیره می‌شوند و دسترسی ترتیبی بسیار سریع است.
- ◆ پایگاه داده‌ها و سیستم‌های فایل مدرن، بیشتر از درخت B+ استفاده می‌کنند.

درخت پیشوندی

تعریف

درخت پیشوندی یک ساختمان داده‌ای غیرخطی است که برای ذخیره و جستجوی سریع مجموعه‌ای از رشته‌ها به کار می‌رود. این ساختار معمولاً در جستجوی پیشوندی، تکمیل خودکار، و سیستم‌های دیکشنری استفاده می‌شود.

ویژگی‌ها

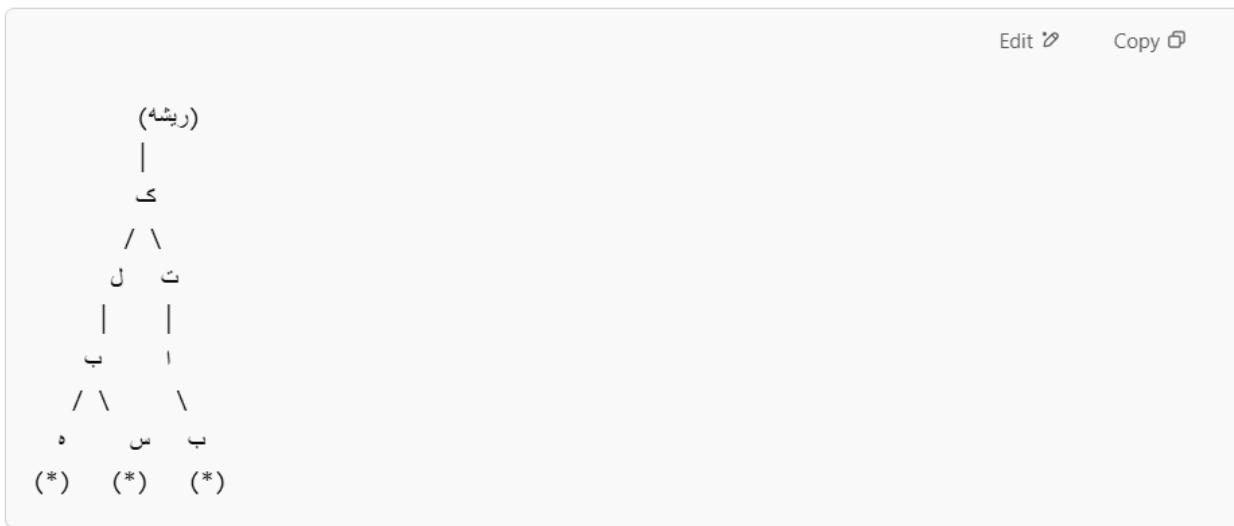
- ✓ هر گره حاوی تعدادی فرزند است که نشان‌دهنده‌ی کاراکترهای ممکن بعدی هستند.
- ✓ هر مسیر از ریشه به یک گره نشان‌دهنده‌ی یک پیشوند از یک یا چند کلمه است.
- ✓ هر گره می‌تواند مشخص کند که آیا در این نقطه یک کلمه کامل شده است یا نه.
- ✓ در مقایسه با جداول هش، جستجوی رشته‌ای بهینه‌تری دارد، زیرا ترتیب حروف را حفظ می‌کند.
- ✓ درج، جستجو و حذف کلمات معمولاً دارای پیچیدگی زمانی مناسب با طول رشته هستند.

ساختار درخت پیشوندی

این درخت شامل گره‌هایی است که:

- دارای یک مجموعه‌ی پیوندی از کاراکترهای فرزندان هستند.
- حاوی یک نشانگر برای تعیین پایان یک کلمه هستند.

مثال درخت پیشوندی برای کلمات: "کتاب"، "کلاس"، "کلبه"



(*) نشان‌دهنده‌ی پایان یک کلمه است.

عملیات در درخت پیشوندی

۱. درج

- برای افزودن یک کلمه، از ریشه شروع کرده و هر حرف را در مسیر مربوطه قرار می‌دهیم. اگر حرفی وجود نداشته باشد، یک گره جدید ساخته می‌شود. در نهایت، آخرین گره به عنوان "پایان کلمه" علامت‌گذاری می‌شود.

```
class Node:  
    def __init__(self):  
        self.children = {} # فرزندان این گره  
        self.is_end_of_word = False # آیا این گره پایان یک کلمه است؟  
  
class PrefixTree:  
    def __init__(self):  
        self.root = Node()
```

```

def insert(self, word):
    node = self.root
    for char in word:
        if char not in node.children:
            node.children[char] = Node() # ایجاد گره جدید در صورت نیاز
            node = node.children[char]
    node.is_end_of_word = True # علامت‌گذاری به عنوان پایان کلمه

```

۵. جستجو

- ♦ برای بررسی وجود یک کلمه، مسیر آن را دنبال می‌کنیم. اگر مسیر کامل یافت شود و گره نهایی به عنوان "پایان کلمه" علامت‌گذاری شده باشد، کلمه در مجموعه وجود دارد.

نمونه پیاده‌سازی:

```

def search(self, word):
    node = self.root
    for char in word:
        if char not in node.children:
            return False # اگر مسیر وجود نداشته باشد
        node = node.children[char]
    return node.is_end_of_word # بررسی اینکه آیا کلمه کامل شده است

```

۳. جستجوی پیشوندی

- ♦ این عملیات بررسی می‌کند که آیا یک پیشوند مشخص در درخت وجود دارد یا نه. برای این کار، مسیر مربوطه را دنبال می‌کنیم.

نمونه پیاده‌سازی:

```

def starts_with(self, prefix):
    node = self.root
    for char in prefix:
        if char not in node.children:
            return False # اگر مسیر وجود نداشته باشد
        node = node.children[char]
    return True # مسیر کامل شده، پس پیشوند موجود است

```

۴. حذف

- ◆ حذف یک کلمه شامل یافتن گره‌های مربوطه و حذف آنها در صورت عدم نیاز (مثلاً اگر گره‌های دیگر از آن استفاده نکنند).

```
def delete(self, word):  
    def _delete(node, word, depth):  
        if depth == len(word): # رسیدن به انتهای کلمه  
            if not node.is_end_of_word:  
                return False # اگر کلمه وجود نداشته باشد  
            node.is_end_of_word = False  
            بررسی نیاز به حذف گره #  
            if len(node.children) == 0:  
                char = word[depth]  
                if char not in node.children:  
                    return False  
                should_delete = _delete(node.children[char], word, depth + 1)  
                if should_delete:  
                    del node.children[char]  
                    if len(node.children) == 0:  
                        return False  
  
_delete(self.root, word, 0)
```

کاربردهای درخت پیشوندی

- ✓ جستجوی سریع کلمات در دیکشنری‌ها
- ✓ تکمیل خودکار در موتورهای جستجو و کیبوردهای مجازی
- ✓ پیدا کردن تمام کلماتی که با یک پیشوند خاص شروع می‌شوند
- ✓ سیستم‌های فشرده‌سازی داده بر اساس الگوهای تکراری
- ✓ بررسی غلطهای املایی در پردازش زبان طبیعی
- ✓ مقایسه با روش‌های دیگر ذخیره‌سازی رشته‌ها

| درخت جستجوی دودویی | جدول هش | درخت پیشوندی | معیار |
|----------------------|-------------|---------------------|---------------------|
| $O(\log n)$ | $O(1)$ | $O(m)$ | زمان جستجو |
| کمتر از درخت پیشوندی | متوسط | زیاد (بسته به حروف) | فضای موردنیاز |
| ضعیف | ضعیف | عالی | پشتیبانی از پیشوند |
| حفظ می‌شود | حفظ نمی‌شود | حفظ می‌شود | ترتیب حروف در جستجو |

نتیجه:

- اگر فقط جستجوی سریع لازم باشد، جدول هش مناسب‌تر است.
- اگر نیاز به جستجوی پیشوندی یا تکمیل خودکار باشد، درخت پیشوندی بهترین گزینه است.
- اگر داده‌های کوچک و دارای ترتیب باشند، درخت جستجوی دودویی گزینه مناسبی است.

جمع‌بندی

- ✓ درخت پیشوندی برای مدیریت رشته‌ها و جستجوی سریع پیشوندها استفاده می‌شود.
- ✓ عملیات درج، جستجو و حذف دارای زمان اجرای $O(m)$ هستند که m طول کلمه است.
- ✓ برای جستجوی پیشوندی، تکمیل خودکار و سیستم‌های پیشنهادهنه بهترین گزینه است.
- ✓ در مقایسه با جدول هش، فضای بیشتری مصرف می‌کند اما عملکرد بهتری در جستجوی پیشوندی دارد.

هیپ

تعریف

هیپ یک ساختمان داده‌ای غیرخطی است که به صورت یک درخت دودویی کامل (درختی که همه سطوح به جز آخرین سطح پر هستند و گره‌های سطح آخر از چپ به راست پر می‌شوند) پیاده‌سازی می‌شود. هیپ معمولاً برای مدیریت صفحه‌ای اولویت‌دار، مرتب‌سازی کارآمد و بهینه‌سازی برخی الگوریتم‌ها مانند دایکسترا استفاده می‌شود.

ویژگی‌های هیپ

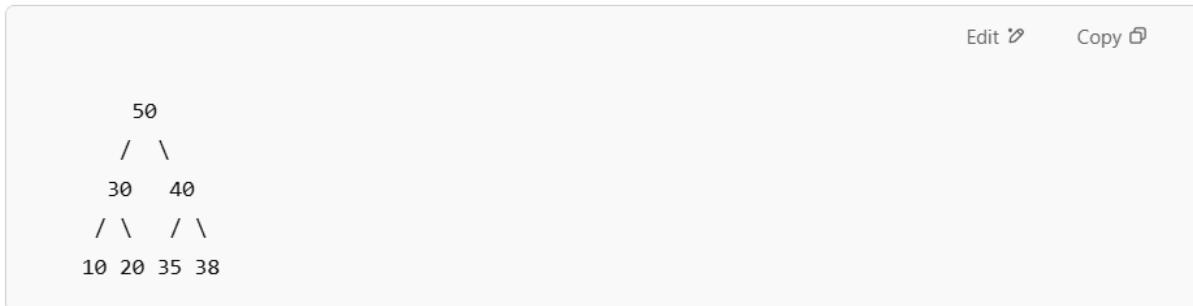
- ✓ ساختار دودویی کامل دارد (درختی که تمامی سطوح به جز آخرین سطح پر هستند).
- ✓ دارای دو نوع اصلی است:
 - هیپ بیشینه (در هر گره مقدار والد بزرگ‌تر یا مساوی مقادیر فرزندانش است).
 - هیپ کمینه (در هر گره مقدار والد کوچک‌تر یا مساوی مقادیر فرزندانش است).
- ✓ دسترسی به مقدار بیشینه یا کمینه در $O(1)$ انجام می‌شود.
- ✓ عملیات درج و حذف دارای پیچیدگی زمانی $O(\log n)$ هستند.

انواع هیپ

۱. هیپ بیشینه

- مقدار بزرگ‌ترین عنصر در ریشه درخت قرار دارد.
- هر والد از مقدار فرزندانش بزرگ‌تر است.
- کاربرد: استفاده در مرتب‌سازی نزولی و پیاده‌سازی صفت اولویت‌دار که عنصر با بیشترین مقدار باید زودتر خارج شود.

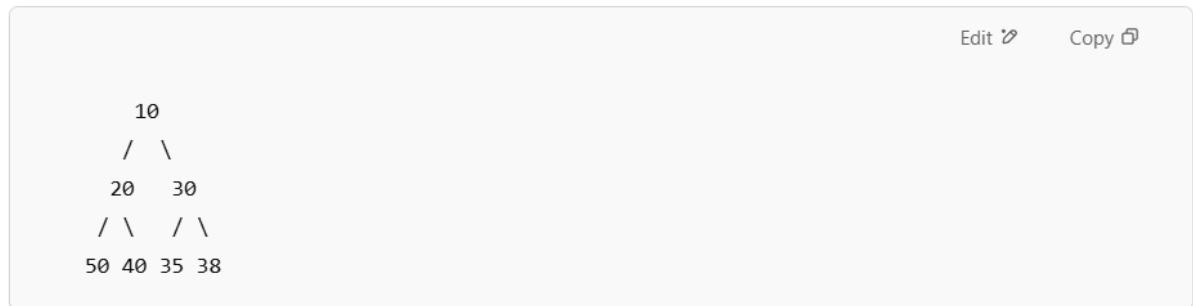
♦ مثال از یک هیپ بیشینه:



۲. هیپ کمینه

- مقدار کوچک‌ترین عنصر در ریشه درخت قرار دارد.
- هر والد از مقدار فرزندانش کوچک‌تر است.
- کاربرد: استفاده در مرتب‌سازی صعودی و پیاده‌سازی صفت اولویت‌دار که عنصر با کمترین مقدار باید زودتر خارج شود.

♦ مثال از یک هیپ کمینه:



عملیات در هیپ

۱. درج (Insertion)

- ♦ برای افزودن مقدار جدید، ابتدا آن را در اولین مکان خالی در سطح آخر قرار می‌دهیم. سپس مقدار را حبابی بالا می‌بریم (Heapify Up) تا خاصیت هیپ حفظ شود.

```

import heapq

heap = [] # بیجاد یک هیپ کمینه
heapq.heappush(heap, 10) # اضافه کردن مقدار 10
heapq.heappush(heap, 30)
heapq.heappush(heap, 20)

print(heap) # خروجی: [10, 20, 30] (هیپ کمینه)

```

۵. حذف عنصر ریشه (Extract Min/Max)

- مقدار ریشه (بیشینه یا کمینه) حذف شده و آخرین مقدار جایگزین آن می‌شود. سپس مقدار جایگزین حبابی پایین می‌رود (Heapify Down) تا خاصیت هیپ حفظ شود.

پیچیدگی زمانی: $O(\log n)$

پیاده‌سازی در پایتون:

```

min_value = heapq.heappop(heap) # حذف کوچکترین مقدار
print(min_value) # 10
print(heap) # [30, 20]

```

۶. ساخت هیپ از یک آرایه (Heapify)

- برای تبدیل یک آرایه‌ی نامرتب به هیپ، از `heapify` استفاده می‌شود که مقدارها را به صورت بهینه تنظیم می‌کند.

پیچیدگی زمانی: $O(n)$

پیاده‌سازی در پایتون:

```

arr = [40, 10, 30, 50, 20]
heapq.heapify(arr) # ساخت هیپ کمینه از آرایه
print(arr) # [10, 20, 30, 40, 50]

```

۷. مرتب‌سازی با هیپ (Heap Sort)

- با استفاده از عملیات حذف، می‌توان داده‌ها را به صورت مرتب شده در زمان $O(n \log n)$ استخراج کرد.

```

def heap_sort(arr):
    heapq.heapify(arr) # ساخت هیپ کمینه
    sorted_arr = [heapq.heappop(arr) for _ in range(len(arr))]
    return sorted_arr

arr = [40, 10, 30, 50, 20]
sorted_arr = heap_sort(arr)
print(sorted_arr) # [10, 20, 30, 40, 50] خروجی:

```

کاربردهای هیپ

- ✓ مدیریت صفحه‌های اولویت دار (مانند زمان‌بندی پردازش‌ها در سیستم‌عامل).
 - ✓ پیاده‌سازی کارآمد الگوریتم دایکسترا برای کوتاه‌ترین مسیر در گراف‌ها.
 - ✓ مرتب‌سازی کارآمد داده‌ها با الگوریتم مرتب‌سازی هیپ.
 - ✓ مدیریت بهینه‌ی منابع در سیستم‌های مدیریت بار و زمان‌بندی تسلیک‌ها.
 - ✓ پیدا کردن k کوچک‌ترین یا بزرگ‌ترین مقدارها در یک مجموعه.
- مقایسه هیپ با دیگر ساختمان‌های داده‌ای

| درخت جستجوی دودویی متوازن | صف اولویت دار | هیپ | معیار |
|---------------------------|---------------|---------------|--------------------------------|
| $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | پیچیدگی درج |
| $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | پیچیدگی حذف مقدار بیشینه/کمینه |
| $O(\log n)$ | $O(1)$ | $O(1)$ | دسترسی به مقدار بیشینه/کمینه |
| $O(n \log n)$ | + | $O(n \log n)$ | مرتب‌سازی |
| $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | ساخت اولیه |

جمع‌بندی

- ✓ هیپ یک درخت دودویی کامل است که از قانون بیشینه یا کمینه پیروی می‌کند.
- ✓ دسترسی به بیشینه/کمینه در $O(1)$ انجام می‌شود، اما درج و حذف n $O(\log n)$ زمان نیاز دارند.
- ✓ پرکاربرد در الگوریتم‌های کوتاه‌ترین مسیر، مرتب‌سازی و مدیریت صفحه‌های اولویت دار است.
- ✓ در مقایسه با درخت جستجوی دودویی متوازن، برای عملیات بیشینه/کمینه کارآمدتر است اما برای جستجو بهینه نیست.

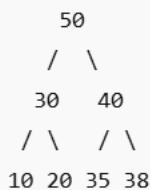
هیپ بیشینه و هیپ کمینه

تعریف کلی

هیپ یک درخت دودویی کامل است که دارای خاصیت هیپ می‌باشد. بسته به نوع هیپ، مقادیر گره‌ها به شکلی چیده می‌شوند که کوچک‌ترین یا بزرگ‌ترین مقدار در ریشه باشد.

۱. هیپ بیشینه (Max Heap)

- ✓ بزرگ‌ترین مقدار در ریشه قرار دارد.
- ✓ هر والد مقدار بیشتری از فرزندان خود دارد.
- ✓ برای حذف سریع مقدار بیشینه استفاده می‌شود (مثلًا در مرتب‌سازی نزولی و صفحه‌های اولویت‌دار).



Edit ⚙ Copy ☉

درج در هیپ بیشینه

- 1 مقدار جدید را در اولین مکان خالی اضافه می‌کنیم.
- 2 مقدار را با والدش مقایسه کرده و در صورت لزوم بالا می‌بریم (Heapify Up).
- . پیچیدگی زمانی: $O(\log n)$ ✓

```
import heapq

max_heap = []
heapq.heappush(max_heap, -50) # مقدار منفی برای شبیه‌سازی هیپ بیشینه
heapq.heappush(max_heap, -30)
heapq.heappush(max_heap, -40)

print([-x for x in max_heap]) # [40, 30, 50]
```

حذف عنصر بیشینه در هیپ بیشینه

- 1 ریشه (بیشینه مقدار) را حذف کرده و آخرین مقدار را جایگزین آن می‌کنیم.
- 2 مقدار جایگزین را به پایین حرکت می‌دهیم (Heapify Down).

پیچیدگی زمانی: $O(\log n)$ ✓

پیاده‌سازی در پایتون:

```
max_value = -heappq.heappop(max_heap) # حذف بزرگترین مقدار  
print(max_value) # خروجی: 50
```

Edit ⌚ Copy ☉

۲. هیپ کمینه (Min Heap)

- ✓ کوچکترین مقدار در ریشه قرار دارد.
- ✓ هر والد مقدار کمتری از فرزندان خود دارد.
- ✓ برای حذف سریع مقدار کمینه استفاده می‌شود (مثلاً در مرتب‌سازی صعودی و الگوریتم دایکسترا).

♦ مثال از یک هیپ کمینه:

```
10  
/ \  
20 30  
/ \ / \  
50 40 35 38
```

درج در هیپ کمینه

- 1 مقدار جدید را در اولین مکان خالی اضافه می‌کنیم.
- 2 مقدار را با والدش مقایسه کرده و در صورت لزوم بالا می‌بریم (Heapify Up).

پیچیدگی زمانی: $O(\log n)$ ✓

```
min_heap = []  
heappq.heappush(min_heap, 10)  
heappq.heappush(min_heap, 30)  
heappq.heappush(min_heap, 20)  
  
print(min_heap) # خروجی: [20, 30, 10]
```

حذف عنصر کمینه در هیپ کمینه

1 ریشه (کمینه مقدار) را حذف کرده و آخرين مقدار را جایگزین آن میکنیم.

2 مقدار جایگزین را به پایین حرکت می‌دهیم (Heapify Down).

بیچیدگی زمانی: $O(\log n)$ ✓

پیاده‌سازی در پایتون:

```
min_value = heapq.heappop(min_heap) # حذف کوچکترین مقدار
print(min_value) # خروجی: 10
```

Edit ⌛ Copy ☉

۳. مقایسه هیپ بیشینه و هیپ کمینه

| هیپ کمینه | هیپ بیشینه | معیار |
|---|---------------------------------------|-----------------|
| کوچکترین مقدار | بزرگترین مقدار | مقدار ریشه |
| حذف سریع کمینه مقدار | حذف سریع بیشینه مقدار | هدف |
| مرتب‌سازی صعودی | مرتب‌سازی نزولی | مرتب‌سازی |
| الگوریتم‌های کوتاه‌ترین مسیر (مثلاً دایکسترا) | صف‌های اولویت‌دار (بیشترین مقدار اول) | کاربرد |
| $O(\log n)$ | $O(\log n)$ | بیچیدگی درج/حذف |

۴. کاربردهای عملی

هیپ بیشینه: ✓

- مدیریت سیستم‌های صف اولویت‌دار مانند زمان‌بندی پردازش‌ها.
- مرتب‌سازی داده‌ها در الگوریتم Heap Sort.
- پیدا کردن K بزرگ‌ترین مقدارها در یک مجموعه.

هیپ کمینه: ✓

- پیاده‌سازی الگوریتم دایکسترا برای یافتن کوتاه‌ترین مسیر.
- مدیریت تنسک‌های پردازشی با کمترین زمان.
- پیدا کردن K کوچک‌ترین مقدارها در یک مجموعه.

۵. جمع‌بندی

- ✓ هیپ بیشینه و کمینه دو نوع از ساختمان داده‌ای هیپ هستند که عملیات درج و حذف را با پیچیدگی $O(\log n)$ انجام می‌دهند.
- ✓ هیپ بیشینه برای حذف سریع بیشترین مقدار و هیپ کمینه برای حذف سریع کمترین مقدار استفاده می‌شود.
- ✓ هر دو نوع در الگوریتم‌های کوتاه‌ترین مسیر، مرتب‌سازی و مدیریت صفاتی اولویت‌دار کاربرد دارند.

هیپ فیبوناچی

۱. تعریف

هیپ فیبوناچی یک ساختمان داده‌ای پیشرفت‌های اولویت‌دار است که در مقایسه با هیپ دودویی، عملیات درج و کاهش کلید را سریع‌تر انجام می‌دهد. این ساختار در الگوریتم‌های گرافی مانند دایکسترا و پریمار کاربرد دارد.

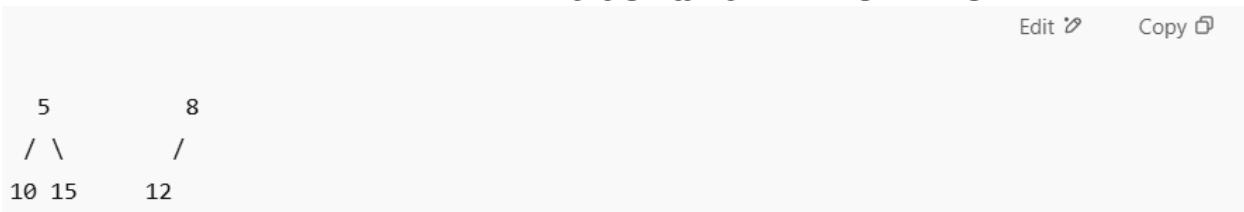
- ✓ عملیات درج در هیپ فیبوناچی دارای پیچیدگی زمانی $O(1)$ است (بهتر از $O(\log n)$ در هیپ دودویی).
- ✓ عملیات حذف کمینه در $O(\log n)$ انجام می‌شود (مشابه هیپ دودویی).
- ✓ کاهش مقدار کلید در $O(1)$ انجام می‌شود (برتری نسبت به هیپ دودویی).

۲. ساختار هیپ فیبوناچی

هیپ فیبوناچی مجموعه‌ای از درخت‌های دودویی نامتعادل است که به صورت یک لیست پیوندی حلقوی ذخیره می‌شوند.

هر گره شامل ویژگی‌های زیر است:

- مقدار کلید
- اشاره‌گر به والد و فرزندان
- اشاره‌گر به گره کناری (سمت چپ و راست)
- رتبه (Degree): تعداد فرزندان مستقیم گره
- علامت (Mark): نشان‌دهنده این است که گره فرزندش را از دست داده است یا نه



در اینجا، 5 و 8 دو درخت مجزا در لیست ریشه‌ها هستند.

۳. عملیات‌های اصلی

۱. درج (Insertion)

- مقدار جدید به لیست پیوندی ریشه‌ها اضافه می‌شود.
- این عملیات $O(1)$ است.

پیاده‌سازی در پایتون:

```
class FibNode:  
    def __init__(self, key):  
        self.key = key  
        self.parent = None  
        self.child = None  
        self.left = self  
        self.right = self  
        self.degree = 0  
        self.mark = False
```

Edit ⌐ Copy ⌐

۲. یافتن مقدار کمینه (Find Min)

- مقدار کمینه در یکی از گره‌های ریشه‌ای ذخیره شده است.
- این عملیات $O(1)$ است.

۳. حذف مقدار کمینه (Extract Min)

- مقدار کمینه حذف می‌شود و فرزندان آن به لیست ریشه‌ها اضافه می‌شوند.
- سپس درخت‌های مشابه را یکی می‌کنیم (Pairwise Combining) تا هیچ متراکم‌تر شود.
- این عملیات $O(\log n)$ است.

```
def extract_min(self):  
    z = self.min # مقدار کوچک‌ترین  
    if z:  
        # فرزندان آن را به لیست ریشه اضافه می‌کیم  
        if z.child:  
            children = [x for x in self.iterate(z.child)]  
            for child in children:  
                self.insert_root(child)  
                child.parent = None
```

```

    از لیست ریشه‌ها z حذف #
self.remove_root(z)
if z == z.right:
    self._min = None
else:
    self._min = z.right
    self.consolidate() # فشرده‌سازی درخت‌ها
return z

```

۵. کاهش مقدار کلید (Decrease Key)

- مقدار یک گره کاهش داده می‌شود.
- اگر مقدار جدید از والدش کمتر باشد، گره از والد جدا شده و به لیست ریشه‌ها منتقل می‌شود.
- اگر والد قبل‌اً یک فرزند از دست داده باشد، والد نیز به لیست ریشه منتقل می‌شود (برش آبشاری - Cascading Cut).
- این عملیات $O(1)$ است (مزیت اصلی نسبت به هیپ دودویی).

۶. مقایسه هیپ فیبوناچی با هیپ دودویی

| هیپ فیبوناچی | هیپ دودویی | عملیات |
|--------------|-------------|-----------------|
| $O(1)$ | $O(\log n)$ | درج |
| $O(1)$ | $O(1)$ | یافتن کمینه |
| $O(\log n)$ | $O(\log n)$ | حذف کمینه |
| $O(1)$ | $O(\log n)$ | کاهش مقدار کلید |
| $O(1)$ | $O(n)$ | ادغام دو هیپ |

۷. کاربردهای عملی

- ✓ الگوریتم دایکسترا (کاهش مقدار کلید باعث بهینه شدن عملکرد).
- ✓ الگوریتم پریمار برای پیدا کردن درخت پوشای کمینه.
- ✓ سیستم‌های زمان‌بندی پردازش و مدیریت تسلیفات.
- ✓ مدیریت کارآمد صفحه‌های اولویت دار با حجم بالا.

۶. جمع‌بندی

هیپ فیبوناچی یک ساختمان داده‌ای کارآمد برای صفحه‌های اولویت‌دار است که عملکرد بهتری نسبت به هیپ دودویی دارد.

پیچیدگی $O(1)$ برای درج و کاهش مقدار کلید آن را برای الگوریتم‌هایی مانند دایکسترا بسیار مفید می‌کند.
پیچیدگی $O(\log n)$ برای حذف کمینه، مشابه هیپ دودویی است، اما ساختار آن انعطاف‌پذیرتر است.

گراف‌ها (Graphs)

۱. تعریف گراف

یک گراف مجموعه‌ای از رئوس (Nodes) و یال‌ها (Edges) است که رئوس را به یکدیگر متصل می‌کنند. گراف‌ها در مدل‌سازی روابط پیچیده، شبکه‌ها، ارتباطات و بسیاری از مسائل گرافی دیگر استفاده می‌شوند.

گراف‌ها به طور کلی به دو دسته تقسیم می‌شوند:

- گراف‌های جهت‌دار (Directed Graphs) یا (Digraphs)
- گراف‌های بدون جهت (Undirected Graphs)

۲. انواع گراف‌ها

۲.۱. گراف‌های بدون جهت

در این نوع گراف‌ها، یال‌ها جهت ندارند، یعنی اتصال بین دو رأس از هر دو طرف برقرار است.

- ♦ مثال: یک شبکه اجتماعی که در آن دو نفر می‌توانند به یکدیگر پیام ارسال کنند.
 - ♦ ویژگی‌ها:
 - یال‌ها به صورت دوطرفه هستند.
 - (u, v) برابر با (v, u) است.

۲.۲. گراف‌های جهت‌دار

در این نوع گراف‌ها، یال‌ها دارای جهت هستند، یعنی اتصال بین دو رأس در یک جهت خاص صورت می‌گیرد.

- ♦ مثال: شبکه اینترنت که اطلاعات از یک سرور به یک کاربر ارسال می‌شود، نه بر عکس.

♦ ویژگی‌ها:

- یال‌ها به صورت یک طرفه هستند.
- (v, u) با (u, v) متفاوت است.

۳. گراف‌های وزنی

گراف‌های وزنی نوعی از گراف‌ها هستند که به یال‌های خود وزن (Weight) یا Cost می‌دهند.
مثال: مسیرهای مختلف در یک نقشه که طول هر مسیر را نشان می‌دهد.

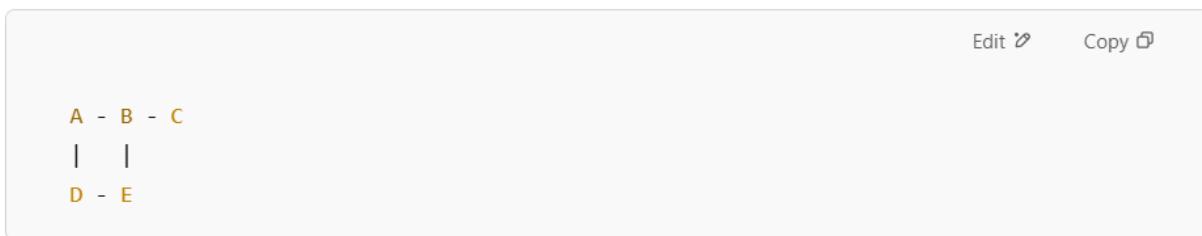
۳. نمایش گراف‌ها

۱. ماتریس مجاورت (Adjacency Matrix)

در این روش، گراف با یک ماتریس مربعی نمایش داده می‌شود که در آن هر عنصر ماتریس نشان‌دهنده وجود یا عدم وجود یال بین دو رأس است.

- ♦ گراف بدون جهت: ماتریس سیمتریک خواهد بود.
- ♦ گراف جهت‌دار: ماتریس غیرسیمتریک است.

مثال (گراف بدون جهت):



ماتریس مجاورت:

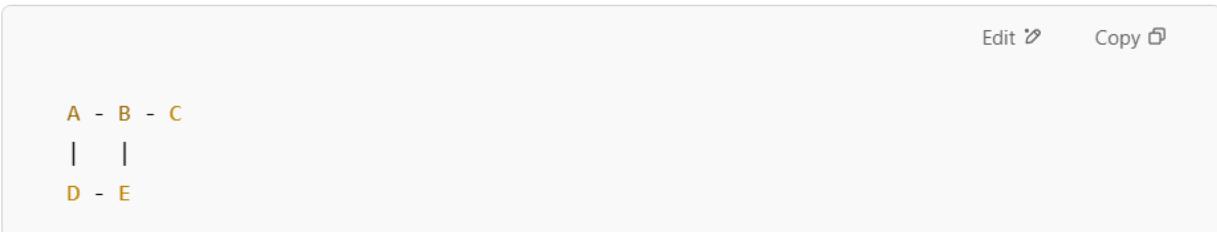
| ماتریس مجاورت: | | | | | |
|----------------|---|---|---|---|---|
| | A | B | C | D | E |
| A | 0 | 1 | 0 | 1 | 0 |
| B | 1 | 0 | 1 | 0 | 1 |
| C | 0 | 1 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 1 |
| E | 0 | 1 | 0 | 1 | 0 |

۳. لیست مجاورت (Adjacency List)

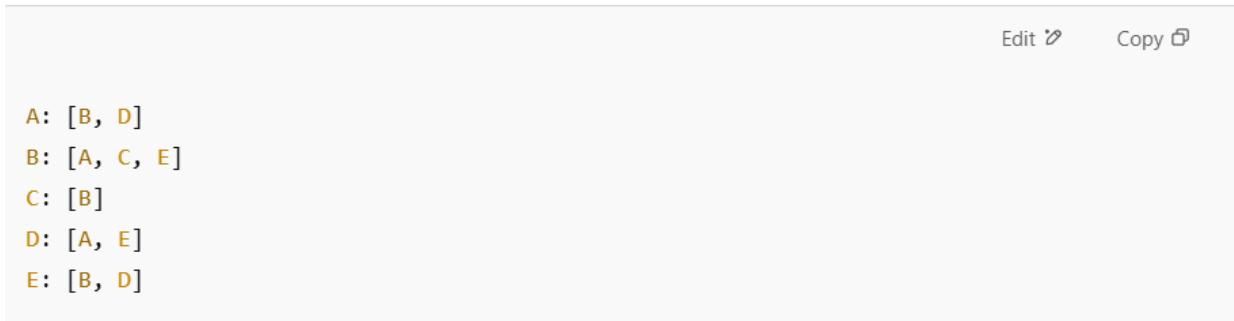
در این روش، برای هر رأس یک لیست از رأس‌های متصل به آن وجود دارد.

- گراف بدون جهت: هر یال در دو لیست مجاور هر دو رأس وجود دارد.
- گراف جهت‌دار: تنها در لیست رأس مبدأ یال وجود دارد.

مثال (گراف بدون جهت):



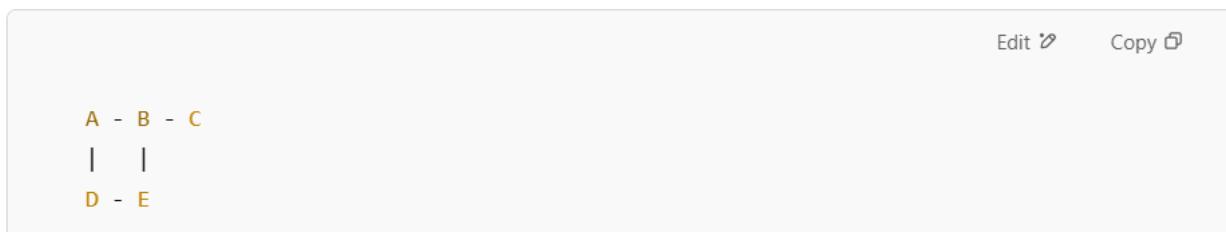
لیست مجاورت:



۳. لیست مجاورت با وزن (Weighted Adjacency List)

در این روش، برای هر یال یک وزن اختصاص داده می‌شود که در لیست مجاورت ذخیره می‌شود.

مثال (گراف وزنی):



- A: [(B, 5), (D, 10)]
B: [(A, 5), (C, 2), (E, 3)]
C: [(B, 2)]
D: [(A, 10), (E, 1)]
E: [(B, 3), (D, 1)]

۴. انواع گراف‌ها بر اساس ویژگی‌های خاص

۴.۱. گراف‌های متصل (Connected Graphs)

- در گراف‌های متصل، بین هر دو رأس، مسیری وجود دارد.
- ♦ گراف‌های بدون جهت: اگر گراف متصل باشد، به این معنی است که از هر رأس می‌توان به همه رأس‌ها رسید.
 - ♦ گراف‌های جهت‌دار: اگر گراف متصل باشد، باید از هر رأس به همه رأس‌های دیگر، یک مسیر وجود داشته باشد.

۴.۲. گراف‌های کامل (Complete Graphs)

- در گراف‌های کامل، بین هر دو رأس یک یال مستقیم وجود دارد.
- ♦ گراف‌های بدون جهت: هر دو رأس با یک یال متصل هستند.
 - ♦ گراف‌های جهت‌دار: هر رأس با یال‌های یک‌طرفه به دیگر رأس‌ها متصل است.

۴.۳. گراف‌های حلقوی (Cyclic Graphs)

- گرافی است که حداقل یک مسیر از رأس به خودش وجود داشته باشد.
- ♦ گراف‌های بدون جهت: اگر مسیر بسته‌ای وجود داشته باشد، گراف حلقوی است.
 - ♦ گراف‌های جهت‌دار: مسیر بازگشتی در جهت‌های خاص باعث حلقوی شدن می‌شود.

۴.۴. گراف‌های بدون حلقه (Acyclic Graphs)

- گراف‌هایی که هیچ مسیر بسته‌ای ندارند.
- ♦ گراف‌های بدون جهت: هیچ یال بسته‌ای وجود ندارد.
 - ♦ گراف‌های جهت‌دار: در این گراف‌ها، هیچ مسیری از رأس به خودش وجود ندارد.

گراف‌های جهت‌دار بدون حلقه:

- ♦ گراف درختی (Tree): گرافی است که در آن بین هر دو رأس دقیقاً یک مسیر وجود دارد.
- ♦ گراف (DAG) (Directed Acyclic Graph): گرافی که جهت دارد و هیچ حلقه‌ای در آن وجود ندارد. این گراف‌ها برای مدل‌سازی مسائل زمان‌بندی و وابستگی‌ها مفید هستند.

۵. الگوریتم‌های مهم در گراف‌ها

۵.۱. جستجوی عمق اول (DFS - Depth-First Search)

- این الگوریتم به صورت بازگشتی به گراف می‌رود و به ترتیب هر رأس را تا انتها بازدید می‌کند.
- ♦ کاربردها: بررسی اتصال گراف، یافتن مسیر در گراف‌های غیرمتصل، یافتن حلقه در گراف.

۵. جستجوی عرض اول (BFS - Breadth-First Search)

این الگوریتم از یک رأس شروع کرده و تمام رأس‌های نزدیک به آن را بازدید می‌کند، سپس به سراغ رأس‌های دورتر می‌رود.

- ♦ کاربردها: یافتن کوتاه‌ترین مسیر در گراف‌های بدون وزن، بررسی گراف‌های متصل و غیرمتصل.

۶. الگوریتم دایکسترا

برای یافتن کوتاه‌ترین مسیر از یک رأس به تمامی رأس‌های دیگر در گراف‌های وزنی استفاده می‌شود.

- ♦ پیچیدگی زمانی: $O(V^2)$ با استفاده از هیب فیبوناچی.

۷. الگوریتم پریمار

برای یافتن درخت پوشای کمینه در گراف‌های وزنی استفاده می‌شود.

- ♦ پیچیدگی زمانی: $O(E \log V)$ با استفاده از هیب.

۸. کاربردهای گراف‌ها

✓ مدیریت شبکه‌ها: از گراف‌ها برای مدل‌سازی و تحلیل شبکه‌های کامپیووتری استفاده می‌شود.

✓ نقشه‌ها و مسیریابی: برای یافتن کوتاه‌ترین مسیر بین دو نقطه در نقشه‌ها.

✓ پروژه‌ها و زمان‌بندی: استفاده در مسائل زمان‌بندی، پروژه‌ها و ترتیب انجام تسلک‌ها.

✓ شبکه‌های اجتماعی: مدل‌سازی روابط و ارتباطات میان کاربران.

۹. جمع‌بندی

✓ گراف‌ها ابزارهای قدرتمندی برای مدل‌سازی روابط پیچیده هستند و در بسیاری از مسائل الگوریتمی و عملیاتی کاربرد دارند.

✓ انواع مختلف گراف‌ها از جمله گراف‌های جهت دار، بدون جهت، وزنی، متصل و بدون حلقه، امکان استفاده‌های مختلف را فراهم می‌کنند.

✓ الگوریتم‌های مهم مانند DFS، BFS، دایکسترا و پریمار برای تحلیل و حل مسائل مرتبط با گراف‌ها به کار می‌روند.