مستندات پروژه ساختمان داده طناب(Rope Data Structure)

مقدمه

این پروژه پیاده‌سازی ساختمان داده طناب (Rope) را ارائه می‌دهد که برای مدیریت و ویرایش رشته‌های بلند بهینه شده است. این پیاده‌سازی در زبان C++ انجام شده و از مفاهیم شیءگرایی و درخت باینری استفاده می‌کند.

کلاس‌ها و ساختار داده‌ها

کلاس RopeNode

این کلاس نمایانگر یک گره در ساختار درخت طناب است.

فیلدها:

- **data:رشته ذخیره شده در گره (برای گره‌های برگ)**
- **weight:طول کل رشته‌های زیردرخت چپ (برای گره‌های داخلی) یا طول رشته (برای گره‌های برگ)**
- **left:اشاره‌گر به زیردرخت چپ**
- **right:اشاره‌گر به زیردرخت راست**

سازنده:

```
RopeNode(const string& s = "")
```

یک گره جدید ایجاد می‌کند. اگر رشته ورودی داده شود، آن را در گره ذخیره می‌کند (گره برگ). در غیر این صورت یک گره داخلی ایجاد می‌شود.

کلاس Rope

کلاس اصلی که عملیات طناب را پیاده‌سازی می‌کند.

فیلدها:

- **strings:وکتور نگهدارنده تمام رشته‌های ایجاد شده**
- **root:اشاره‌گر به ریشه درخت طناب**

متدهای خصوصی:

1. **clear(RopeNode* node):**
   - **پاکسازی بازگشتی حافظه اختصاص داده شده به درخت**
   - **پارامتر: گره ریشه زیردرخت مورد نظر برای پاکسازی**
2. **buildRope(const string& s, int start, int end):**
   - **ساخت درخت طناب از یک رشته به صورت بازگشتی**
   - **پارامترها: رشته منبع، اندیس شروع و پایان**
   - **اگر طول زیررشته کمتر از ۵ باشد، یک گره برگ ایجاد می‌شود**
   - **در غیر این صورت رشته به دو بخش تقسیم و به صورت بازگشتی پردازش می‌شود**
3. **collectStrings(RopeNode* node):**
   - **جمع‌آوری تمام رشته‌های ذخیره شده در برگ‌های درخت**
   - **پارامتر: گره ریشه زیردرخت مورد نظر**
4. **getLength(RopeNode* node):**
   - **محاسبه طول کل رشته‌های ذخیره شده در زیردرخت**
   - **پارامتر: گره ریشه زیردرخت مورد نظر**

متدهای عمومی:

1. **سازنده و تخریب‌کننده:**
   - **Rope():سازنده پیش‌فرض**
   - **Rope~():تخریب‌کننده که حافظه را آزاد می‌کند**
2. **addString(const string& s):**
   - **اضافه کردن یک رشته جدید به طناب**
   - **اگر طناب خالی باشد، یک درخت جدید ساخته می‌شود**
   - **در غیر این صورت رشته جدید به عنوان زیردرخت راست به ریشه فعلی اضافه می‌شود**
3. **status():**
   - **نمایش تمام رشته‌های ذخیره شده در طناب با شماره گذاری**
4. **index(int rope_id, int i):**
   - **بازگرداندن کاراکتر در موقعیت i از رشته مشخص شده**
   - **پارامترها: شناسه رشته و اندیس مورد نظر**
   - **در صورت اشتباه بودن شناسه یا اندیس، استثنا پرتاب می‌کند**
5. **concat(int rope_id1, int rope_id2):**
   - **الحاق رشته دوم به انتهای رشته اول**
   - **رشته دوم از لیست حذف می‌شود**
   - **درخت طناب مجددا ساخته می‌شود**
6. **split(int rope_id, int i):**
   - **تقسیم رشته مشخص شده به دو رشته در موقعیتi**
   - **یک جفت از طناب‌های جدید بازگردانده می‌شود**
   - **رشته اصلی تغییری نمی‌کند**
7. **insert(int rope_id1, int i, int rope_id2):**

- درج رشته دوم در موقعیت i از رشته اول
- درخت طناب مجددا ساخته می‌شود

8. deleteRange(int rope_id, int i, int j):

- حذف کاراکترهای بین اندیس‌های i و j از رشته مشخص شده
- درخت طناب مجددا ساخته می‌شود

9. report(int rope_id, int i, int j):

- بازگرداندن زیررشته بین اندیس‌های i و j از رشته مشخص شده

مثال استفاده

```
Rope rope;
// اضافه کردن رشته‌های جدید
rope.addString("Hello");
rope.addString("World");
// نمایش وضعیت
rope.status();
// الحاق رشته‌ها
rope.concat(1, 2);
// تقسیم رشته
auto splitRopes = rope.split(1, 5);
// درج رشته
Rope rope2;
rope2.addString("ABC");
rope2.addString("XYZ");
rope2.insert(1, 1, 2);
// حذف محدوده
rope2.deleteRange(1, 2, 4);
// گزارش زیررشته
cout << rope2.report(1, 0, 1);
```

نکات پیاده‌سازی

1. این پیاده‌سازی از یک وکتور برای نگهداری رشته‌ها استفاده می‌کند و درخت طناب را برای هر رشته می‌سازد.
2. برای تقسیم رشته‌ها از روش تقسیم دودویی استفاده شده است.
3. عملیات‌های تغییردهنده (مانند insert و delete) درخت طناب را مجددا می‌سازند.
4. پیاده‌سازی از اصول شیءگرایی پیروی می‌کند و مدیریت حافظه به درستی انجام شده است.
5. بررسی‌های لازم برای اندیس‌های نامعتبر انجام شده و استثناهای مناسب پرتاب می‌شوند.

محدودیت‌ها

1. پیاده‌سازی فعلی برای هر عملیات تغییردهنده، درخت طناب را از نو می‌سازد که می‌تواند کارایی را کاهش دهد.
2. حداکثر طول رشته در گره‌های برگ به ۵ کاراکتر محدود شده است.
3. گزارش تعداد گره‌های درگیر در عملیات report پیاده‌سازی نشده است.