مستندات تمرین شماره ۵ (پیاده‌سازی پشته)

نام و نام خانوادگی اعضای تیم (شماره ۵):

عرفان محرم زاده، نیلوفر خرسندی، شروین رکن سادات، تینا غلامی راد

تمرین:

پیاده‌سازی پشته با عملیات: درج- حذف- بررسی خالی یا پر بودن پشته و پیاده‌سازی الگوریتم تبدیل عبارت میانوندی به پسوندی

درس:

ساختمان داده‌ها و الگوریتم‌ها، ترم ۰۰۲

استاد:

دکتر سیف الدینی

دانشگاه:

دانشگاه گیلان، رشت

رابط کاربری

برای اجرای برنامه، باید فایل run.py را اجرا کرد. با اجرای این فایل، رابط کاربری طراحی شده در فایل UI.py اجرا خواهد شد.

رابط کاربری برنامه به شکل مقابل است.

در شماره ۱ کاربر ورودی خود را وارد می‌کند و با زدن کلید شماره ۳ خروجی در شماره ۲ چاپ می‌شود. اگر ورودی کاربر درست بود، عبارت پسوندی را محاسبه و چاپ می‌کند و اگر ورودی نادرست بود، پیغام خطایی را در خروجی چاپ می‌کند.

همچنین قابلیتی به برنامه اضافه شده تا برنامه در حین تایپ شدن ورودی توسط کاربر، به صورت خودکار جواب مورد نظر را چاپ کند و اگر ورودی اشتباه بود در همان لحظه پیغام خطا را چاپ کند؛ برای فعالسازی این قابلیت چک‌باکس شماره ۴ پیاده‌سازی شده.

قابلیت دیگر برنامه، ایجاد یک عبارت میانوندی رندوم، درست و بدون خطا است که با توجه به تعداد عملوندهای موردنظر کاربر که از شماره ۶ مشخص می‌کند، تولید می‌شود.

با زدن کلید شماره ۷ می‌توان ورودی‌ها و خروجی‌ها رو پاک کرد.

کد برنامه‌نویسی

داده‌ساختار (پشته)

در فایل stack_dataStructure.py داده ساختار پشته پیاده‌سازی شده.

به این صورت که کلاسی با نام Stack() با ۴ متد اصلی و ۱ متغیر:

- متغیر __items مقادیری که قرار است در پشته ذخیره شوند را نگه‌داری می‌کند. این متغیر از نوع لیست است.
- متد push() وظیفه اضافه کردن آیتم‌ها را به پشته بر عهده دارد.
- متد pop() آخرین و بالاترین آیتم موجود را از پشته حذف می‌کند و آن را برمی‌گرداند.
- متد isEmpty() خالی بودن یا نبودن پشته را بررسی می‌کند.
- متد peek()آخرین آیتم داخل پشته را به عنوان خروجی بر می‌گرداند اما آن را از پشته حذف نمی‌کند.

تبدیل عبارت میانوندی به عبارت پسوندی

در فایل infixToPostfix.py الگوریتم مورد نظر نوشته شده است.

برای پیاده‌سازی این الگوریتم، ما از داده‌ساختار پشته بهره می‌بریم. پس ابتدا با استفاده از کلاس Stack() شیئی به نام stack می‌سازیم.

همچنین متدی با نام __pushShodan(amalgar, stack) طراحی شده تا با دریافت یک عملگر و یک پشته بررسی کند آیا عملگر می‌تواند به پشته اضافه شود یا نه. خروجی این تابع True یا False است. و شروط زیر را چک می‌کند و اگر یکی از آن‌ها برقرار بود، پس عملگر می‌تواند به پشته اضافه شود:

1) پشته خالی باشد.
2) آخرین عنصر پشته پرانتز باز باشد
3) اولویت عملگر از اولویت آخرین عنصر پشته بیشتر باشد

تابعی با نام infixToPostfix(infix, printTerminal=False) طراحی شده که قرار است یک عبارت میانوندی را دریافت کند و سپس آن را به عبارت پسوندی تبدیل کند و برگرداند. در این متد همچنین قابلیت چاپ مراحل حل مسئله در ترمینال گنجانده شده و به وسیله‌ی پارامتر printTerminal می‌توان مشخص کرد آیا مراحل در ترمینال چاپ شوند یا خیر؛ دلیل استفاده از این پارامتر، کندی برنامه در چاپ مقادیر بسیار زیاد برای ورودی‌های بسیار بزرگ بود. در برنامه طوری نوشته شده که اگر تعداد کاراکتر ورودی کمتر از ۱۲۰ بود اقدام به چاپ کند.

در این تابع پس از دریافت رشته ورودی میانوندی، کاراکترهای رشته به وسیله یک حلقه و متغیر char یک به یک بررسی می‌شود:

1) اگر char عملوند بود مستقیم به postfix اضافه می‌شود.
2) اگر char پرانتز باز بود به پشته منتقل می‌شود.
3) اگر char پرانتز بسته بود، تا جایی که به اولین پرانتز باز در پشته برسیم عنصرهای پشته را pop و به postfix اضافه می‌کنیم.
4) درصورت برقرار نبودن شروط بالا، پس char یک عملگر است. در اینصورت با استفاده از تابع __pushShodan، تا زمانی که عملگر مورد نظر

بتواند در پشته قرار بگیرد، یکی یکی آیتم‌ها از پشته pop و به postfix اضافه می‌شوند.

در نهایت پس از پیمایش کامل infix اگر پشته خالی نبود، تا زمان خالی شدن پشته عناصر آن را به postfix اضافه می‌کنیم.

بررسی صحیح بودن ورودی

الگوریتم به این صورت ایجاد شده که تمامی قوانینی را که باید در یک عبارت میانوندی صادق باشد را بررسی می‌کند. این الگوریتم در فایل check_infix.py نوشته شده است.

ابتدا یک تابع به نام appendShodan(listInfix, char) طراحی شده که دو پارامتر می‌گیرد و بررسی می‌کند آیا کاراکتر موردنظر (که می‌تواند پرانتزها، عملوندها یا عملگرها باشند) را می‌توان در انتهای رشته‌ی موردنظر قرار داد یا نه. برای این منظور نیاز داریم چند قانون را بررسی کنیم:

1) اگر رشته‌ی ورودی خالی بود، می‌توان پرانتز باز یا یک عملوند را به آن افزود.
2) اگر کاراکتر انتهایی رشته پرانتز باز بود، می‌توان به انتهای آن یک عملوند یا یک پرانتز باز افزود.
3) اگر کاراکتر انتهایی رشته پرانتز بسته بود، می‌توان مقابل آن یک عملگر افزود. همینطور می‌توان یک پرانتز بسته به آن افزود به شرطی که تعداد پرانتز باز در رشته، بیشتر از تعداد پرانتز بسته آن باشد.
4) اگر کاراکتر انتهایی یک عملوند بود، می‌توان به آن یک عملگر اضافه کرد. همچنین می‌توان پرانتز بسته به آن اضافه کرد به شرطی که تعداد تعداد پرانتز باز در رشته، بیشتر از تعداد پرانتز بسته آن باشد.
5) اگر کاراکتر انتهایی یک عملگر بود، می‌توان در انتهای رشته پرانتز باز یا یک عملوند قرار داد.

حال می‌دانیم که یک کاراکتر در یک رشته، نسبت به کاراکترهای قبلی در جای درستی قرار دارد یا نه. در ادامه تابعی به نام isInfix(string) طراحی کردیم که یک رشته را دریافت می‌کند و تشخیص می‌دهد آیا یک عبارت میانوندی درست است یا نه. برای این منظور استراتژی زیر را در پیش گرفتیم:

1) اگر رشته خالی بود یا تعداد پرانتزهای بسته‌ی آن با تعداد پرانتزهای باز آن برابر نبود، رشته مورد نظر عبارت میانوندی نیست.
2) تک تک کاراکترهای رشته را پیمایش می‌کنیم تا ببینیم نسبت به کاراکترهای قبلی در جایگاه منطقی خود مطابق با قوانین قرار دارد یا نه:

a. در یک عبارت میانوندی، حتما باید عملگرها بین عملوندها باشند. برای همین اندیس آخرین عملوند و اندیس آخرین عملگر را بدست می‌آوریم و اگر آخرین عملگر سمت راست آخرین عملوند قرار گرفته بود، پس رشته‌ی موردنظر میانوندی نیست. (توجه کنید که درمورد اولین کاراکتر رشته، تابع appendShodan(listInfix, char) تصمیم‌گیری لازم را انجام می‌دهد)
b. با استفاده از تابع appendShodan(listInfix, char) کاراکتر درون رشته را نسبت به زیررشته قبل از خود می‌سنجیم.

ایجاد یک عبارت میانوندی رندوم و درست

برای ایجاد یک عبارت میانوندی رندوم و درست، ما از تابع appendShodan(listInfix, char) در فایل قبلی بسیار بهره می‌بریم. الگوریتم ایجاد یک عبارت میانوندی رندوم در فایل random_infix.py نوشته شده است.

ما یک تابع به نام randomInfix(countAmalvand) طراحی کردیم که تعداد عملوندها را دریافت می‌کند و یک عبارت میانوندی با همان تعداد عملوند را باز می‌گرداند.

استراتژی ما برای طراحی این الگوریتم این بود که ابتدا یک لیست تصادفی ایجاد می‌کنیم که به تعداد ورودی داده شده، عملوند رندوم را جایگذاری کند. همچنین پرانتزهای باز و بسته به تعداد کافی هم در این لیست اضافه کند. (توجه داریم که تعداد پرانتزهای باز و بسته یکسان است). پس از ایجاد این لیست تصادفی، یک لیست خالی با نام infix ایجاد می‌کنیم تا خروجی را در آن ذخیره کنیم. سپس یک حلقه نوشتیم تا لیست تصادفی عملوندها و پرانتزها را پیمایش کند و بررسی کند آیا کاراکتر می‌تواند به infix اضافه شود یا نه (با استفاده از تابع appendShodan(listInfix, char)). این کار را تا زمانی ادامه می‌دهد که تعداد عملوندهای مجاز برای استفاده به پایان برسد.

در انتهای این فرایند، لیست infix را بررسی می‌کند که آیا تعداد پرانتزهای باز با تعداد پرانتزهای بسته برابر هستند یا نه. اگر تعداد پرانتز باز بیشتر بود، به انتهای infix به تعداد موردنیاز پرانتز بسته اضافه می‌کند (توجه داریم که در طول فرایندی که در پاراگراف قبلی به آن اشاره شد، هیچگاه تعداد پرانتز بسته بیشتر از باز نخواهد شد چرا که این مسئله را تابع appendShodan(listInfix, char) مدیریت می‌کند.

خروجی ترمینال

علاوه بر طراحی رابط کاربری، از امکانات ترمینال نیز استفاده کردیم. زمانی که برنامه در حال حل کردن مسئله است، تمامی مراحل در ترمینال چاپ می‌شود. یعنی گام به گام پشته به همراه postfix را چاپ می‌کند.