# Compression

1. بررسی برای **NullPointerException** و **IllegalArgumentException.**
2. تعریف دو آرایه موازی ( **Map** ) که جنس آنها یک **Character** و دیگری **Integer** است و بدست آوردن فرکانس هر کاراکتر با استفاده از تابع **getcharfrequencies** .
3. تعریف دو آرایه موازی ( **Map** ) که جنس آنها یک **Character** و دیگری **String** برای ذخیره سازی **کُد** مربوط به هر کاراکتر به صورت **String**.
4. با استفاده از تابع **encodeInput** تمامی **input** به کد 0 و 1 تبدیل می شود. آرگومان های این تابع **Map** و دیگری **String (رشته ی ورودی)** است.
5. در این مرحله باید رشته ی حاوی 0 و 1 را در یک **Bitset** ذخیره کنیم.
6. توجه شود که در این مرحله باید به نحوی درخت را در یک فایل نیز ذخیره کنیم تا در زمان خارج کردن فایل از حالت فشرده کد مربوط به هر کاراکتر را در دسترس داشته باشیم. این کار با استفاده از تابع **SerializeTree** برای ذخیره سازی درخت و تابع **SerializeInput** برای ذخیره سازی **Bitset** در یک فایل انجام میگیرد.
7. تابع **SerializeTree** با استفاده از یک پیمایش **Preorder** درخت را در یک فایل ( **Tree** ) ذخیره میکند.
8. توجه شود کدهای 0 و 1 مربوط به هر گره از درخت در فایل **Tree** و **کاراکتر** های هر گره در فایل **Char** ذخیره می شود.
9. کلاس **IntObject** حاوی یک متغیر از جنس **int** می باشد تا محل ذخیره سازی در **bitset** مشخص شود.
10. وظیفه ی تابع **serializeInput** تبدیل کردن رشته **encoded_Input** به یک **Bitset** می باشد.

**نکته۱: Queue** یک **Interface** است و این بدین معنیست که باید نوع آن مشخص گردد. به عنوان مثال در تابع **buildTree** ابتدا باید یک صف اولویت از **map** فرستاده شده به آن تابع ساخته شود که در اینجا اینکار توسط تابع **createPrioQueue** انجام میگیرد و حاصل آن یک صف از جنس **H_Node** می باشد.

**نکته۲: صف اولویت** در جاوا بر اساس یک **comparator** داده ها مرتب سازی می کند که این **comparator** در ابتدای کلاس هافمن پیاده سازی شده است و اساس مرتب سازی بدین صورت است که **node** کوچکتر در سمت راست ریشه و **node** بزرگتر در سمت چپ ریشه قرار خواهد گرفت. این کار باید با استفاده از تفریق فرکانس **node** های چپ و راست مشخص گردد.

```
Map<Character, String> map = new HashMap<Character, String> ();
```

**نکته۳**: در تابع**getCharFrequency** در زمان خواندن رشته ی ورودی **input** در ابتدا باید بررسی شود که این کاراکتر خوانده شده در **map** وجود دارد یا خیر. اگر وجود داشته باشد باید به تعداد تکرار آن یکی اضافه گردد در غیر این صورت با فرکانس ۱، خانه جدیدی را در **map** اشغال خواهد کرد.

**نکته۴**: توابع **CodeGenerator** و **Generator** وظیفه ی تولید رشته ی ۰ و ۱ برای درخت هافمن را بر عهدا دارند.

# Decompression

1. در ابتدا باید اطلاعات و کلید فشرده سازی بازیابی شود که این کار از طریق فراخوانی تابع **deserializeTree** برای بازیابی **درخت بیتی Tree** ذخیره شده در فایل و نیز فایلی که **درخت کاراکتری Char** را شامل می شود انجام میگیرد.
2. در تابع **deserializeTree** باید تابع بازگشتی دیگری به نام **Preorder** فراخوانی شود چون با توجه به اینکه فایل با پیمایش **Preorder** ذخیره شده بود در این حالت باید به همین صورت خوانده شود تا درخت هافمن بازسازی گردد.
3. پس از بازسازی درخت هافمن باید فایل اصلی **cmp.** خوانده شود و در یک مجموعه بیتی **Bitset** لود می شود.
4. سپس این **Bitset** به صورت بیت به بیت بررسی شده و با توجه به بیت های خوانده شده درخت هافمن از ریشه بررسی شده و در نهایت کاراکتری که به عنوان برگ در سطح آخر درخت هافمن قرار گرفته است به انتهای یک **String** اضافه می شود و این **String** همان فایل **Decompress** شده می باشد.

# User Interface

- **JFileChooser** برای نمایش **Browse/Save dialog** استفاده شده است.
- برای انتخاب و ذخیره فایل های **cmp** و **txt** بصورت پیش فرض در **Browse/Save dialog** می بایست از **FileNameExtensionFilter** استفاده کرد.
- از **JButton** برای نمایش اولین منوی انتخاب استفاده شده است و **Button** های **Compress File** و **Decompress File** با استفاده از این المان پیاده شده اند.
- نهایتا از یک **Frame** برای نمایش تمامی اجزای **UI** استفاده شده است و تک به تک این **المان** ها در این **Frame** اضافه می گردند.