1. منظور از thread-safe بودن چیست؟
   1. یک کد زمانی thread-safe است که اگر بیش از یک thread همزمان در حال اجرای آن باشند، مشکلی به وجود نیاید. مواردی مثل دسترسی به یک حافظه مشترک بدون locking و یا deadlock که به خاطر وابستگی مشترک به یک حافظه مشترک به وجود می آید از مواردی هستند که این ویژگی را نقض می کنند.
2. تفاوت mutex و semaphore؟
   1. توی
3. انواع data structure ها و مزایا و معایب هر کدام؟
   1. Array: مزایاش اینه که دسترسی به خانه مورد نظر سریعتر و سر راسته اما برای اضافه یا پاک کردن دیتا کند هست چون باید یه آرایه جدید ساخته بشه و تغییرات توی اون اعمال بشه و بعد باید آرایه اولی پاک بشه از طرفی هر آرایه اندازه اش ثابته و قابل تغییر نیست.
   2. Linked list: دقیقا برعکس array هست. یعنی دسترسی به خانه مورد نظر کندتره چون باید از خونه اول شروع کنه و دونه دونه بره خونه های بعدی تا به خونه مورد نظر برسه اما توی اضافه یا پاک کردن هم memory management بهتری داره هم سریعتره چون میره تا خونه مورد نظر و نهایتا دو تا pointer رو تغییر میده اما چیزی رو (مثل array) کپی و پاک نمیکنه. اندازه اش هم قابل تغییره.
   3. Stack: خوراکش واسه function call ها و بازگشتی هاس. برای عملیات های ساده اوکیه اما برای دسترسی (هر چی دیتا زودتر اومده باشه دسترسی بهش دیرتره) کندتره و دسترسی مستقیم به هر خونه وجود نداره (یکم شبیه linked list میشه)
   4. Queue: برای scheduling عالیه اما محدودیت هاش مشابه stack ئه.
   5. Hashmap: به خاطر ساختار key-value محورش و اینکه key ها رو طوری hash میکنه که مثل یه index عمل میکنن و مستقیم میرن سر خونه دیتای مورد نظر، سرعت دسترسی به حافظه اش بالاس (خوندن، نوشتن و پاک کردن تا حد O(1)) و مناسب دیتاست های بزرگه. از طرفی احتمال داره که دوتا key به یه value هش بشن که دردسر هایی داره و به خاطر هشینگ، مموری بالاتری نسبت به بقیه میگیره.
   6. Tree: برای حالات سلسله مراتبی عالین و در مدل های بالانس شده (AVL, Red-black tree) عملیات های پاک کردن، نوشتن و خوندن با O(logn) انجام میشن اما همین درخت اگه بالانس نشده باشه O رو پایین میاره. از طرفی کار با درخت ها (operations) یه مقدار پیچیده تره.
4. انواع الگوریتم های جستجو؟
   1. Linear:
5. Diamond problem چیست؟
   1. این مسئله زمانی اتفاق میفته که داخل زبان برنامه نویسیمون multiple inheritance داشته باشیم. فرض کنیم یه کلاس والد داریم که دوتا کلاس دیگه دارن ازش ارث بری میکنن و هر دو یه متد رو دارن از کلاس والد داخل خودشون تعریف میکنن. حالا اگه یه کلاس چهارم پیدا بشه که بخواد از اون دوتا کلاس (هر دوی کلاس هایی که دارن از والد ارث بری میکنن)، ارث ببره اینجا مشکل پیش میاد که اون متد، داخل کلاس چهارم که بخواد استفاده بشه مال کدوم یکی از کلاس های بالاییه؟ امضای متد که یکیه بین این کلاس ها پس سر اینکه تشخیص بده به مشکل میخوره.  
      اینجاس که اینترفیس ها رو استفاده می کنیم. از اونجایی که هر اینترفیس باید داخل کلاسی که ازش استفاده میکنه، implement بشه، دیگه مشکل مبهم بودن متد ها رو نداریم چون هر کلاسی مال خودشو میاد پیاده سازی می کنه.
   2. اما یه مشکل دیگه میتونه به وجود بیاد توی C#. چون توی C# ما default method interface ها رو داریم، ممکنه این مسئله diamond دوباره پیش بیاد. با همین سناریو اما به جای ارث بری از چند کلاس، ارث بری از چند تا اینترفیس رو داریم که داخل خودشون implementation رو دارن.   
      اینجا C# حواسش بوده و هر زمان که این مشکل به وجود بیاد، توی compile time بهمون ارور میده. میگه که نمیتونه بین متد های اینترفیس که قبلا داخل اینترفیس ها implement شدن انتخاب کنه در نتیجه ما باید توی کلاس مون متد رو پیاده سازی کنیم. در واقع اینطوری که پیاده سازی هر کلاس، بر پیاده سازی پیش فرض (default implementation) ارجحیت داره.  
      پس default method interface ها در عین اینکه مثل یه کلاس، اجازه پیاده سازی داخل یه اینترفیس رو میدن، از اونور جلوی diamond problem رو هم میگیرن.