Колекције вектор, стек и повезана листа, разна разматрања

1. **Read-only пролазак кроз колекцију**

   За пролазак кроз вектор, стек и повезану листу навели смо три начина: итератор, collection-based for-петљу и бројачку for-петљу уз приступ елементима колекције коришћењем метода get().

   (итератор vs. collection-based for-петља) Collection-based for-петља интерно користи итератор, тако да је за read-only пролазак кроз колекцију (вектор, стек, повезану листу) свеједно шта ће од та два бити коришћено. Предност се даје collection-based for-петљи, због веће читљивости.

   (collection-based vs. бројачка for-петља) У случају повезане листе (која не имплементира интерфејс RandomAccess), ефикасније је користити collection-based него бројачку for-петљу, док је у случају вектора и стека (који имплементирају интерфејс RandomAccess) мање-више свеједно. Разлог зашто је collection-based ефикаснија од бројачке for-петље за повезану листу је следећи: итератори за све колекције задовољавају фундаментални захтев да next() мора бити операција сложености O(1), док је за повезану листу get(i) операција сложености O(i): да би се дошло до i-тог елемента мора се прећи преко свих претходних i-1. (За вектор и стек, који су уопштења низа, операција get(i) је сложености O(1), јер се интерно врши индексирање низа).

2. **Мењање колекције**

   collection-based for-петља не омогућује мењање колекције (операције add и remove) унутар петље, док итератор омогућује. Итератор се, такође, може користити када није познато који тип колекције је у питању, пошто све колекције поседују итератор.

   Структурално мењање колекције након што је креиран итератор на било који други начин осим коришћења метода add() или remove() самог итератора, доводи до избацивања изузетка типа ConcurrentModificationException. Collection-based for-петља интерно користи итератор, па директно мењање листе у њој узрокује избацивање изузетка типа ConcurrentModificationException.

   На пример, ово је у реду:

```
LinkedList<Object> list = new LinkedList<Object>();
// add some items to the list
Iterator<Object> setIterator = set.iterator();
while(setIterator.hasNext()){
     Object o = setIterator.next();
     if(o meets some condition){
          setIterator.remove();
     }
}
```

док ово избацује изузетак типа `ConcurrentModificationException`:

```
LinkedList<Object> set = new LinkedList<Object>();
// add some items to the list

for(Object o : set){
     if(o meets some condition){
          set.remove(o);
     }
}
```

3. **Vector<> vs. ArrayList<>**
   ArrayList<> није thread-safe (безбедан за коришћење од стране већег броја нити), док Vector<> јесте.
   (Интерно, и ArrayList<> и Vector<> чувају своје податке користећи Array, тј. низ.)

4. **Када користити Vector<>/ArrayList<> а када LinkedList<>?**
   Додавање нових елемената је прилично брзо у оба случаја. Случајан приступ коришћењем get() је брз за вектор, а спор за повезану листу (јер не постоји ефикасан начин индексирања унутрашњости повезане листе). Уклањање елемената вектора је споро (јер се преостали елементи у интерном низу шифтују приликом сваког брисања), док је за повезану листу брзо (само се промени неколико веза). Закључак је да је Vector<>/ArrayList<> бољи у случајевима случајног приступа листи, а повезана листа када има доста измена у средини листе.