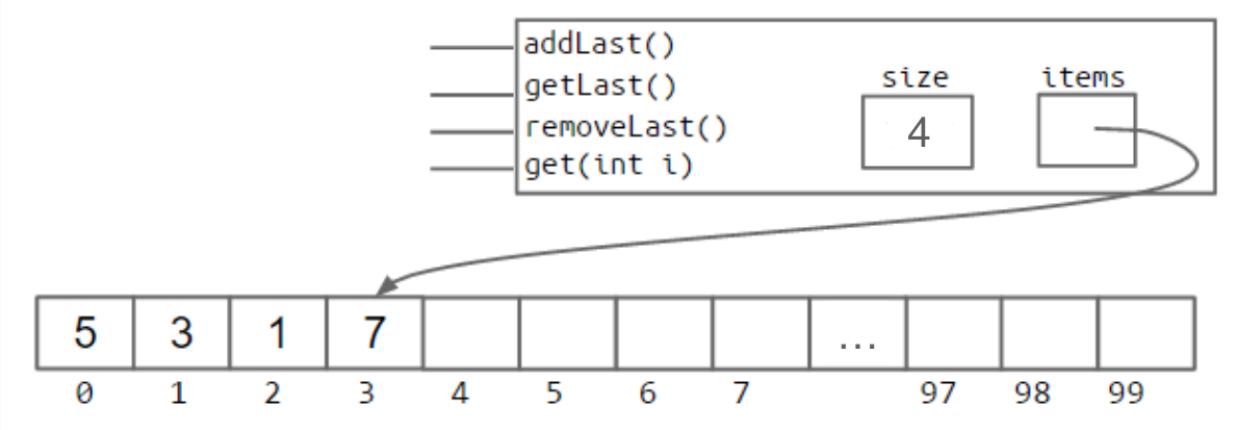
CS61B sp24 week3

tips

1. testing
   - **All tests must be non-static.**

     > When you create JUnit test files, you should precede each test method with a `@Test` annotation, and can have one or more `assertEquals` or `assertTrue` methods (provided by the JUnit library). ---- cs61b sp21 lab2

   - we could use the The Google Truth library to simplify the writing of the test eg. `assertThat(input).isEqualTo(expected);`
   - notice that we need to import the library first like this `import static com.google.common.truth.Truth.assertThat;`
   - `^` is bitwise or in java and we could use `Math.pow()` to power numbers
   - how to compare 2 string in java?
   - `str1.compareTo(str2)` if str1 < str2, return negative number, vise versa
   - we could use method like this to ignore the case `int compare = s.compareToIgnoreCase(best);
2. ArrayList
   1. replace the linked list with the backing array



   2. In the model of ArrayList, the items and size are both invariants, and when we change the items[i] the items would still hold the **address of the array**, so there is no need for it to have changed
   3. if we just want to remove the last element, there is no need to change it, we could leave it hang around and "cast shadow on it" i.e. we just need to reduce the size of the ArrayList
   4. what if the array is not big enough for the users' request?
      1. so we need to re-size the array --> create a new one with larger size, copy the old one and throw out the old one
      2. we could create a new array which has the larger room by `System.arraycopy();`
      3. `for (int index_first : Addarray)` iterates over each element in the `Addarray` array and assigns the current element to the `index_first` variable (NOT the index number of array!)
      4. . it would be better if we create a helper method instead of put both copy and create progress in a same function i.e.This will display as:

```
private void resize(int capacity) {
    int[] a = new int[capacity];
    System.arraycopy(items, 0, a, 0, size);
    items = a;
}
```

```
public void addLast(int x) {
    if (size == items.length) {
        resize(size + 1);
    }
    items[size] = x;
    size += 1;
}
```

3. raw use of generic types is valid in Java, but it defeats the purpose of type parameters and may mask bugs.
    - so if we need to initialise the variable which in generic types, we'd better set the parameter's type
    - eg. instead of `AListNoResizing NoBugList = new AListNoResizing();` we should use (empty angle brackets in the right side = left side's by default)
4. the runtime and space usage
    - compare SLList and ArrayList : for ArrayList, each operation takes linear time (NOT SAME TIME), which would make the space and time needed increase exponentially; but for SLList, each operation take the same time, make the picture of SLList increasing as a straight line
    - And using helper function cannot solve this issue
    - we could using the multiplication instead of the addition factor to solve this problem e.g. every time you need a new array, create a double size of origin one and remove the origin one
    - if the array size is much more bigger than the spaces we need eg. (4 vs 100000), then we also need to resize it down
    - NOTICE: if you want to create a new Array, we could not use the placeholder(and the angle brackets) to instantiate it
    - we need to avoid a situation called "loitering". If a box in array store the reference to another staff such as a picture, if we delete the thing but keep the link, Java would still think the thing is still here, which is a waste of memory.
    - SO we could avoid it by replace the reference with null thus we null out the deleted elements and lose the reference
5. hypernyms and hyponyms
    1. hypernym: the noun that is the class/union of other nouns eg. both AList and SLList are specific types of list, so "list" is a hypernym of SLList and Alist
    2. Expressing this in Java is a two-step process:
        - Step 1: Define a **reference** type for our hypernym (List61B.java)
        - Step 2: Specify that SLLists and ALists are hyponyms of that type
    3. how to tell Java AList is a specific list of List61B?
        - we just need to use the magic word "implements" (a relationship-defining word)
        - eg. `public class AList<Item> implements List61B<Item>`
6. overload vs override
    - OVERRIDE means a method has already replace the function of another method, they have the same parameters' types and name.The overriding method must have the same name, return type, and parameter types as the method it overrides.
    - OVERLOAD means 2 method in the same class (or a subclass) have the same name but different parameter lists (different type, number, or both), they are no contradictory to each other, Java would look at the type of passed variables to decide use which one
    - One method could override another, we we would not say a method overload another
    - we could add the `@Override` tag before override method to remind ourselves that the method is override another one (just increasing readability)
    - `@Override` tag also can help us find the typo in our code
    - NOTICE: we cannot override twice
7. Interface Inheritance

- Interface is the list of all method signatures
- and the sub-class could "inherits" all the parent class's methods/behaviours
- it is a powerful way that allows you to generalize code
- notice that we need to override all of the blank methods of the interface in the subclasses (Otherwise it will fail to compile)
- there are also another inheritance called **implementation inheritance**
  - NOTICE: In interface inheritance we just inherit "what", but in implementation we inherit "how"
  - with **default** key word, we could write method/code in the interface
  - we could use the methods that have to be included in any subclasses
  - after overriding the default method in the interface, we could use the syntax `InterfaceName.super.<method>` to call the default method from the interface within our overriding method.
- eg.

```
interface MyInterface {
    default void myMethod() {
        System.out.println("Default method in the interface");
    }
}

class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        System.out.println("Overridden method in the class");
        MyInterface.super.myMethod();  // Calling the default method from the interface
    }
}
```

8. Static and dynamic type
   1. each variables has 2 types, one is "compile time type"(static type) and another is "run time type"(dynamic type)
   2. the static type is defined when the variable is declared, and once it been declared, it cannot change any more
   3. the dynamic type could be change with the code running (the things after the `new` key word)
   4. dynamic method selection (explanation fromcs61B textbook)
      - eg. In `List61B<String> lst = new SLList<String>();` , the List61B is the static type, the SLList is the dynamic type
      - and assume there are 2 methods

```
class Parent {
    void show() {
        System.out.println("Parent's show()");
    }
}

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child's show()");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
```

```
        obj.show(); // output "Child's show()"
    }
}
```

> When Java runs a method that is overriden, it searches for the appropriate method signature in it's **dynamic type** and runs it.