1. A multistack consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the $i$th stack $S_i$ can hold up to $3^i$ elements. The user always pushes and pops elements from the smallest stack $S_0$. However, before any element can be pushed onto any full stack $S_i$, we first pop all the elements off $S_i$ and push them onto stack $S_{i+1}$ to make room. (Thus, if $S_{i+1}$ is already full, we first recursively move all its members to $S_{i+2}$.) Similarly, before any element can be popped from any empty stack $S_i$, we first pop $3^i$ elements from $S_{i+1}$ and push them onto $S_i$ to make room. (Thus, if $S_{i+1}$ is already empty, we first recursively fill it by popping elements from $S_{i+2}$.) Moving a single element from one stack to another takes $O(1)$ time.

Here is pseudocode for the multistack operations MSPUSH and MSPOP. The internal stacks are managed with the subroutines PUSH and POP.

```
MPush(x):
  i ← 0
  while $S_i$ is full
     i ← i + 1
  while i > 0
     i ← i − 1
     for j ← 1 to $3^i$
        Push($S_{i+1}$, Pop($S_i$))
  Push($S_0$, x)

MPop(x):
  i ← 0
  while $S_i$ is empty
     i ← i + 1
  while i > 0
     i ← i − 1
     for j ← 1 to $3^i$
        Push($S_i$, Pop($S_{i+1}$))
  return Pop($S_0$)
```

(a) In the worst case, how long does it take to push one more element onto a multistack containing $n$ elements?

(b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where $n$ is the maximum number of elements in the multistack during its lifetime.

(c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where $n$ is the maximum number of elements in the multistack during its lifetime.
2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

- **CREATE()** creates and returns a new list
- **Push(L, x)** appends x to the end of L
- **Pop(L)** deletes the last entry of L and returns it
- **Lookup(L, k)** returns the kth entry of L

Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses anything fancier, you must give an explicit implementation. Your data structure must support all operations in amortized constant time. In addition, your data structure must support each **Lookup** in worst-case $O(1)$ time. At all times, the size of your data structure must be linear in the number of objects it stores.
3. Let $P$ be a set of $n$ points in the plane. The \textit{staircase} of $P$ is the set of all points in the plane that have at least one point in $P$ both above and to the right.

(a) Describe an algorithm to compute a representation of the staircase of a set of $n$ points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm \textsc{Above?}$(x, y)$ that returns \textsc{True} if the point $(x, y)$ is above the staircase, or \textsc{False} otherwise. Your data structure should use $O(n)$ space, and your \textsc{Above?} algorithm should run in $O(\log n)$ time.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function \textsc{Insert}$(x, y)$ that adds the point $(x, y)$ to the underlying point set and returns \textsc{True} or \textsc{False} to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your \textsc{Insert} algorithm should run in $O(\log n)$ amortized time.