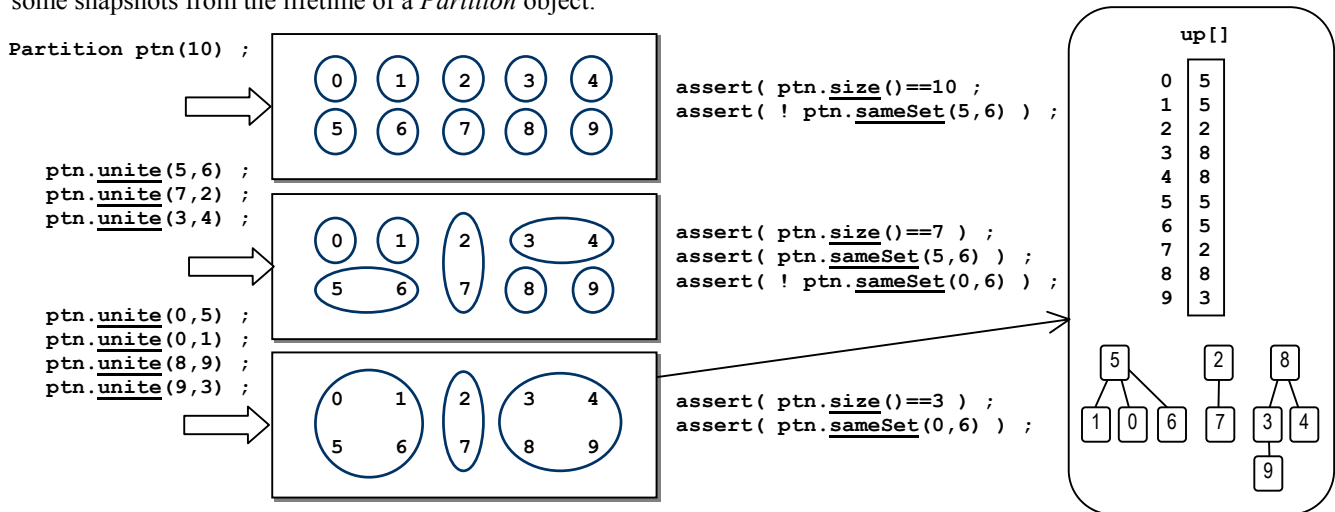


Program 1

Review practice in OO C++; ascertaining the costs of OO

The *Partition* ADT is used in several graph-related algorithms (finding minimal connecting sub-networks, for example). It is simple enough (without being *too* simple) to provide a nice set of benchmarks for us. In the form we use here, a partition is a set of disjoint subsets of $\{0,1,2,\dots n-1\}$ whose union is the entire set. A partition object begins life as a set of n singleton sets; inspectors include *size()*, which counts the number of subsets, and *sameSet(i,j)* which checks if i and j belong to the same set. Its one mutator is *unite(i,j)*, which unites the set containing i with the set containing j . Here are some snapshots from the lifetime of a *Partition* object:



A good implementation of *Partition* uses a forest of trees, with each node holding an integer. Each tree in the forest represents a set. Two integers i and j are in the same set if they share the same root. Fortunately, we don't need nodes' pointers; we use an array *up[]* to find parents: $up[i]==j$ means the parent of i is j . A root is represented as a fixed point: $up[i]==i$. You merge two sets by making one set's root the child of the other set's root. There are two important optimizations: (a) when uniting, make the root of the tree of the smaller set a child of the root of the tree of the larger set (got it?); and (b) when traversing upward to find the root, keep track of all the integers on the path then make them all children of the root. The latter is called "path compression." More info is in CLR Chapter 21¹.

For this assignment you will code the Partition ADT (using the implementation above) in C++ in three ways:

- A. A mean-and-speedy ANSI C procedural implementation. Use comments to make it clear, but feel free to sacrifice code beauty for performance.
- B. A concrete class *PartitionForest*. Inline as much as you can.
- C. An interface *IPartition* implemented by a concrete class *CPartitionForest*, working with an abstract factory.

When writing each of these alternatives, you should imagine yourself as a lawyer defending the respective paradigm (procedural, ADT, OO), trying to make it perform as well as "best practices" in that paradigm allow.

You will benchmark these on the same task: construct a partition with $n=10,000$ elements, then execute a sequence of 10,000 random *unite* operations, each followed by 10 *sameSet* tests. Try out at least two compilers, with optimizations turned on. Carefully control for what you can: e.g. make the under-the-hood data structures work identically in all three cases; seed the random number generator identically as well. You are encouraged to try out additional variations to pinpoint the precise locations of cost. If you have a fast computer, try a larger n .

Submit three single-file self-contained programs (p1A.c, p1B.cpp, p1C.cpp) for the alternatives, and attach a brief (1 or 2 page) document summarizing your findings. The summary should contain (at least) a 3x2 table of benchmark times, with a few paragraphs of discussion. **Due:** Thursday September 12. This is an individual assignment.

¹ Cormen, Leiserson, Rivest, *Introduction to Algorithms* (2nd Ed.), MIT Press, 2001. CLR is the classic, but this data structure is covered in many intermediate and advanced data structures texts.