### Iterative Quicksort

#### Iterative Conversion

Iterative implementation requires using a stack to store the partition bounds remaining to be sorted.

Assume a stack implementation of elements consisting of two integers:

```
struct StackItem {
  int low, hi;
};
```

#### Partitioning

At the end of any given partition, only one subpartition need be stacked.

The second subpartition (equated to the second recursive call) need not be stacked since it is immediately used for the next subpartitioning.

#### Stacking

The order of the recursive calls, i.e., the sorting of the subpartitions, may be made in any order.

Stacking the larger subpartition assures that the size of the stack is minimized, since the smaller subpartition will be further divided less times than the larger subpartition.

### Iterative Quicksort Code

#### Quicksort Function (iterative)

```c
void QuickSort(Item A[], int start, int end) {
  // sort the array from start ... end
  Item pivotKey; int pivotIndex, tmpBnd;
  int k;                  //index of partition >= pivot
  StackItem parts;       //initialize stack
  parts.low = start;     parts.hi = end;
  subParts.Push ( parts ); //prime stack
  while ( ! subParts.Empty() )  {//while partitions exist
    parts = subParts.Pop();
    while ( parts.hi > parts.low ) {
      pivotIndex = FindPivot(A, parts.low, parts.hi);
      if (pivotIndex != MISSING) {
        pivotKey = A[pivotIndex];
        k = Partition(A, parts.low, parts.hi, pivotKey);
        if ( (k-parts.low) > (parts.hi-k) ) { //stk low part
          tmpBnd = parts.hi;
          parts.hi = k-1;
          subParts.Push( parts );parts.low = k; //set current part to upper part
          parts.hi = tmpBnd;
        } //end  if
        else { // stack upper (larger) part
          tmpBnd = parts.low;
          parts.low = k;subParts.Push( parts );
          parts.low = tmpBnd; //set current part to low part
          parts.hi = k-1;
        } // end else
      } // end if
    } // end while
  } // end while
}
```