**void merge(int A[], int L[], int R[], int Lsize, int Rsize)**{

int Andx = 0, Lndx, Rndx;

for(Lndx = 0, Rndx = 0; Lndx < Lsize && Rndx < Rsize;){

if(L[Lndx] < R[Rndx]){

A[Andx++] = L[Lndx++];

} else {

A[Andx++] = R[Rndx++];

}

}

**//if right array finishes first**

while(Lndx < Lsize){

A[Andx++] = L[Lndx++];

}

**//if left array finishes first**

while(Rndx < Rsize){

A[Andx++] = R[Rndx++];

}

}

**int mergeSort(int arr[], int size)**{

if(size < 2)

return 0;

int mid = size / 2;

int ndx = 0, Lndx = 0, Rndx = 0;

**//MergeSort Left**

int \*L = (int\*)calloc(mid, sizeof(int));

while(ndx < mid){

L[Lndx++] = arr[ndx++];

}

mergeSort(L, mid);

**//MergeSort Right**

int \*R = (int\*)calloc(size - mid, sizeof(int));

while(ndx < size){

R[Rndx++] = arr[ndx++];

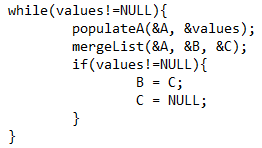
}

mergeSort(R, size - mid);

//Merge

merge(arr, L, R, mid, size - mid);

}

typedef struct node {

int val;

struct node \*link;

}node, \*list;

**void mergeList(list \*A, list \*B, list \*C)** {

list \*trav;

if(\*B!=NULL){

for(trav=C; \*A!=NULL&&\*B!=NULL;){

if((\*A)->val<(\*B)->val){

\*trav = \*A;

\*A = (\*trav)->link;

} else {

\*trav = \*B;

\*B = (\*trav)->link;

}

(\*trav)->link = NULL;

trav = &(\*trav)->link;

}

if(\*A==NULL){

\*trav=\*B;

\*B=NULL;

} else {

\*trav=\*A;

\*A=NULL;

}

} else {

\*C = \*A;

\*A = NULL;

}

}

**void populateA(list \*A, list \*values)** {

list \*trav, \*last;

\*A = \*values;

\*values = (\*A)->link;

(\*A)->link = NULL;

for(trav=values, last=A; \*trav!=NULL&&\*values!=NULL;){

if((\*trav)->val > (\*last)->val){

last = &(\*last)->link;

\*last = \*trav;

\*trav = (\*trav)->link;

(\*last)->link = NULL;

} else {

trav = &(\*trav)->link;

}

}

}

**void quicksortv1(int arr[], int low, int high**){

int pos;

if(low < high) {

pos = partitionv1(arr, low, high);

quicksortv1(arr, low, pos-1);

quicksortv1(arr, pos+1, high);

}

}

**int partitionv1(int arr[], int low, int high)**{

int x, y, pivot;

pivot = arr[high];

x = low;

y = high;

while (x < y) {

for(; x < y && arr[x] < pivot; x++) {ACCESSES++;}

for(; y > x && arr[y] >= pivot; y--) {ACCESSES++;}

if(x < y){

swap(&arr[x], &arr[y]);

}

};

swap(&arr[x], &arr[high]);

return x;

}

**void tournamentSort(int arr**[]) {

int i, j, left, right, heap[SIZE\*2] = {0}, output[SIZE];

for(i = 0; i < SIZE; i++) { // initialize heap

heap[i] = MAX\_INT;

}

for(i = SIZE; i < SIZE\*2; i++) { // initialize index of element to the heap

heap[i] = i-SIZE;

}

for(i = 0; i < SIZE; i++) {

findWinner(arr, heap, 0);

output[i] = arr[heap[0]];

arr[heap[0]] = MAX\_INT;

}

for(i = 0; i < SIZE; i++) {

arr[i] = output[i];

}

}

**void findWinner(int arr[], int heap[], int n)** { // similar to heapify

if(heap[n] == MAX\_INT || arr[heap[n]] == MAX\_INT) {

int left = 2 \* n + 1;

int right = 2 \* n + 2;

int largest = n;

if(left < SIZE\*2) {

findWinner(arr, heap, left);

largest = left;

}

if(right < SIZE\*2) {

findWinner(arr, heap, right);

if(arr[heap[right]] < arr[heap[largest]]) {

largest = right;

}

}

if(largest != n) {

heap[n] = heap[largest];

}

}

}

**void heapSort(int arr[])** {

int i, temp;

for(i = SIZE/2-1; i >= 0; i--) {

heapify(arr, SIZE, i);

}

for(i = SIZE-1; i > 0; i--) {

temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

**void heapify(int arr[], int n, int node)** // max heapify {

int largest = node, left = 2 \* largest + 1, right = 2 \* largest + 2, temp;

if(left < n && arr[left] > arr[largest]) {

largest = left;

}

if(right < n && arr[right] > arr[largest]) {

largest = right;

}

if(largest != node) {

temp = arr[largest];

arr[largest] = arr[node];

arr[node] = temp;

heapify(arr, n, largest);

}

}