Lab 5 – Sorting

Hoàn thiện lớp Heap, để lớp này có thể tạo được min-heap

```
class Heap
{

……

    void ReheapUp(long position)
    {
            if(position > 0)
            {
                    long parent = (position - 1)/2;

                    // For max-heap
                    if(this->heap_type == MAX_HEAP && this->arr[position] > this->arr[parent])
                    {
                            int temp = this->arr[position];
                            this->arr[position] = this->arr[parent];
                            this->arr[parent] = temp;
                            ReheapUp(parent);
                    }
                    // For min-heap
                    if(this->heap_type == MIN_HEAP && this->arr[position] < this->arr[parent])
                    {
                            int temp = this->arr[position];
                            this->arr[position] = this->arr[parent];

                            ReheapUp(parent);
                    }

            }
    }
    void ReheapDown(int position, int lastPosition)
    {
            long leftChild = 2*position + 1;
            long rightChild = 2*position + 2;
            long child;

            //For max-heap
            if(this->heap_type == MAX_HEAP)
            {
                    if(leftChild <= lastPosition)
                    {
                            if(rightChild <= lastPosition && this->arr[rightChild] > this-
>arr[leftChild])
                                    child = rightChild;
                            else
```

```
                                        child = leftChild;

                                if(this->arr[child] > this->arr[position])
                                {
                                        int temp = this->arr[child];
                                        this->arr[child] = this->arr[position];
                                        this->arr[position] = temp;
                                        ReheapDown(child, lastPosition);
                                }
                        }
                }
                //For min-heap
                if(this->heap_type == MIN_HEAP)
                {
                        if(leftChild <= lastPosition)
                        {
                                if(rightChild <= lastPosition && this->arr[rightChild] < this-
>arr[leftChild])
                                        child = rightChild;
                                else
                                        child = leftChild;

                                if(this->arr[child] < this->arr[position])
                                {
                                        int temp = this->arr[child];
                                        this->arr[child] = this->arr[position];
                                        this->arr[position] = temp;
                                        ReheapDown(child, lastPosition);
                                }
                        }
                }
        }

        …….

        bool IsHeap()
        {
                long position = this->count/2 - 1;
                long lastPosition = this->count - 1;

                while(position >= 0)
                {
                        long leftChild = 2*position + 1;
                        long rightChild = 2*position + 2;
                        long child;

                        //For max-heap
                        if(this->heap_type == MAX_HEAP)
                        {
                                if(leftChild <= lastPosition)
                                {
```

```
                                        if(rightChild <= lastPosition && this->arr[rightChild] > this-
>arr[leftChild])
                                                child = rightChild;
                                        else
                                                child = leftChild;

                                        if(this->arr[child] > this->arr[position])
                                                return false;
                                }
                        }

                        //For min-heap
                        if(this->heap_type == MIN_HEAP)
                        {
                                if(leftChild <= lastPosition)
                                {
                                        if(rightChild <= lastPosition && this->arr[rightChild] < this-
>arr[leftChild])
                                                child = rightChild;
                                        else
                                                child = leftChild;

                                        if(this->arr[child] < this->arr[position])
                                                return false;
                                }
                        }
                        position--;
                }
                return true;
        }
        ……
};
```

- Sắp xếp Shell, sắp xếp dãy số theo thứ tự tăng dần

```
void ShellSort(int *arr)
{
        int             t = log((double)(ARRAY_SIZE))/log(2.0) - 1;
        int             kArr[64], j=0;
        int             i, k;

        for(i = t; i>=0; i--)
        {
                kArr[j] = pow(2.0, i) - 1;
                j++;
        }
        /*cout << t << " ";
        for(int i = 0; i<t; i++)
                cout << kArr[i] << " ";*/

        for(i = 0; i<t; i++)
```

```
        {
                k = kArr[i];

                int     segment = 1;
                while(segment <= k)
                {
                        SortSegment(arr, segment, k);
                        segment++;
                }
        }
}
```

- Sắp xếp chọn trực tiếp, sắp xếp dãy số theo thứ tự tăng dần

```
void SelectionSort(int* arr)
{
        long count = ARRAY_SIZE, current;

        current = 0;
        while(current < count - 1)
        {
                long    smallest = current;
                long  walker = current + 1;
                while(walker < count)
                {
                        if(arr[walker] < arr[smallest])
                                smallest = walker;
                        walker++;
                }
                int temp = arr[current];
                arr[current]=arr[smallest];
                arr[smallest] = temp;
                current++;
        }
}
```

- Sắp xếp Heap, **sắp xếp dãy số theo thứ tự giảm dần**

```
void HeapSort(int* arr)
{
        heap.CopyData(arr, ARRAY_SIZE);

        heap.BuildHeap();
        if(heap.IsHeap() == false)
        {
                cout << "Not a heap" << endl;
                return;
        }

        long last = heap.getCount() - 1;
        while(last >= 0)
```

```
        {
                int temp = heap.arr[0];
                heap.arr[0] = heap.arr[last];
                heap.arr[last]=temp;
                last--;

                heap.ReheapDown(0, last );
        }

        memcpy(arr1, heap.arr, sizeof(int)*heap.count);
    }
```

- Sắp xếp nổi bọt, sắp xếp dãy số theo thứ tự tăng dần

```
void BubbleSort(int* arr)
{
        long count = ARRAY_SIZE, current;

        current = 0;
        bool flag = false;

        while(current < count && flag == false)
        {
                long walker = count - 1;
                flag = true;
                while(walker > current)
                {
                        if(arr[walker] < arr[walker-1])
                        {
                                flag = false;
                                long temp = arr[walker];
                                arr[walker] = arr[walker-1];
                                arr[walker-1]=temp;
                        }
                        walker--;
                }
                current++;
        }
}
```


- Sắp xếp QuickSort, sắp xếp dãy số theo thứ tự tăng dần

```
void swap(int *arr, long pos1, long pos2)
{
        int temp = arr[pos1];
        arr[pos1] = arr[pos2];
        arr[pos2] = temp;
}
long Partition(int* arr, long low, long high)
{
        swap(arr, low, (low + high)/2);
```

```
	int pivot = arr[low];
	long last_small = low;
	long i = low + 1;

	while(i <= high)
	{
		if(arr[i] < pivot)
		{
			last_small++;
			swap(arr, last_small, i);
		}
		i++;
	}
	swap(arr, low, last_small);

	return last_small;
}
void recursiveQuickSort(int *arr, long low, long high)
{
	if(low < high)
	{
		long pivot_pos = Partition(arr, low, high);
		recursiveQuickSort(arr, low, pivot_pos-1);
		recursiveQuickSort(arr, pivot_pos+1, high);
	}
}
void QuickSort(int *arr)
{
	recursiveQuickSort(arr, 0, ARRAY_SIZE-1);
}
```

- Sắp xếp MergeSort, sắp xếp dãy số theo thứ tự tăng dần

```
void merge(int *a, int*b, int l, int m, int r) {
	int start = l;
	int mid = m;
	while ((l < mid) && (m <= r)) {
		if (a[l] > a[m]) {
			b[start] = a[m];
			m++;
		} else {
			b[start] = a[l];
			l++;
		}
		start++;
	}
	while (l < mid) {
		b[start] = a[l];
		l++;
		start++;
	}
```

```
        while (m <= r) {
                b[start] = a[m];
                m++;
                start++;
        }
}

void recursiveMergeSort(int *a, int *b, int l, int r)
{
        if (r > l) {
                int mid = (l + r) / 2;
                MergeSort(a, b, l, mid);
                MergeSort(a, b, mid + 1, r);
                merge(a, b, l, mid + 1, r);
                for (int i = l; i <= r; i++)
                        a[i] = b[i];
        }
}

void MergeSort(int* arr)
{
        recursiveMergeSort(arr, new int[ARRAY_SIZE], 0, ARRAY_SIZE - 1)
}
```