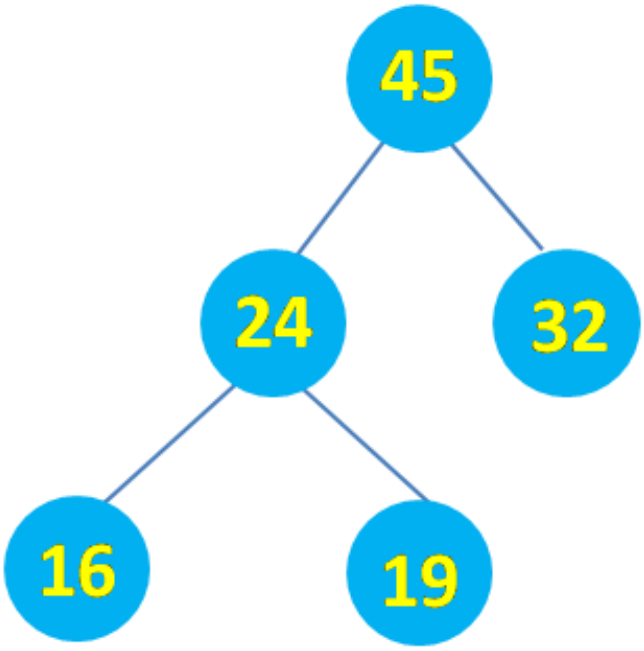


Binary Heap

IMPLEMENTING BINARY HEAP & SORTING TECHNIQUES

Aya Sameh Ismail Farag (1)
Dahlia Chehata Mahmoud Chehata (28)
Salma Hesham Mohammed Ragab (34)

Contents

Problem statement

1. BINARY HEAP

I. Introduction

The (binary) heap data structure is an array object that we can view as a nearly complete binary. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A. There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root, A[parent[i]] ≥ A[i] (1) that is, the value of a node is at most the value of its parent.

II. Requirements

In this assignment, you're required to implement some basic procedures and show how they could be used in a sorting algorithm:

• The MAX-HEAPIFY procedure, which runs in O(lg n) time, is the key to maintaining the max-heap property.

• The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.

• The HEAPSORT procedure, which runs in O(n lg n) time, sorts an array in place.

• The MAX-HEAP-INSERT, and HEAP-EXTRACT-MAX procedures, which run in O(lg n) time, allow the heap data structure to implement a priority queue. You're required to implement the above procedures.

2. SORTING TECHNIQUES

You are required to implement the "heapsort" algorithm as an application for binary heaps. You're required to compare the running time performance of your algorithms against: –

An O(n 2) sorting algorithm such as Selection Sort, Bubble Sort, or Insertion sort. – An O(n lg n) sorting algorithm such as Merge Sort or Quick sort algorithm in the average case.1

. In addition to heapsort, select one of the sorting algorithms from each class mentioned above.

Pseudo code

BINARY HEAP

Max Heapify

```
// Makes the value at A[i] float down in the max-heap; so that the subtree rooted at index I, obeys the
// max-heap property.

MaxHeapify (A, i)
        // Get the left and right elements of the desired node.
        l = Left(i)
        r = Right(i)

        // Check if the index of the left node lies in the range of the heap.
        // Compare the left node to its parent; to set the largest element.
        if (l <= A.heapSize) and (A[l] > A[i])
                largest = l
        else
                largest = i

        // Check if the index of the right node lies in the range of the heap.
        // Compare the right node to the largest element; to set the largest element.
        if (r <= A.heapSize) and (A[r] > A[largest])
                largest = r

        if largest != i
                // Swap the largest element with the node i
                exchange (A[i]) with (A[largest])
                MaxHeapify(A, largest)
```

Build Max Heap

```
// Goes through the nodes of the tree and runs MaxHeapify on each.
BuildMaxHeap(A)
        A.heapSize = A.length
        for floor(i = A.length / 2) downto i = 1
                MaxHeapify(A, i)
```

Max Heap Insert

```
MaxHeapInsert(object)
        heapSize = heapSize + 1
        A[heapSize] = object
        HeapifyUp(heapSize)
```

Heapify Up

```
HeapifyUp(i)
        // Check if the parent of the node i is smaller than it.
        if (i > 1) and (A[parent(i)] < A[i])
                // If the parent was smaller, then swap both elements.
                exchange (A[i]) with A[parent(i)]
                HeapifyUp(A[parent(i)])
```

Heap Extract Max

```
HeapExtractMax(A)
        max = A[1]
        A[1] = A[heapSize]
        heapSize = heapSize - 1 // Delete the last element.
        MaxHeapify(A,1)
        return max
```

SORTING ALGORITHMS

Selection Sort

```
SelectionSort(A)
        n = A.length
        for (i = 1) to (n - 1)
                // Set current element as minimum.
                min = i

                // Check the element to be minimum.
                for (j = i+1) to n
                        if (list[j] < list[min])
                                min = j;
                end for
                // Swap the minimum element with the current element.
                if minIndex != i
                        exchange (list[min]) with (list[i])
                end if
        end for
```

Bubble Sort

```
BubbleSort(A)
        for (i = 1) to (A.length - 1)
                for (j = A.length) downto (i + 1)
                        if (A[j] < A[j - 1])
                                exchange (A[j]) with (A[j - 1])
                end for
        end for
```

HeapSort

```
HeapSort(A)
        BuildMaxHeap(A)
        for(i = A.length) downto 2
                exchange(A[i]) with (A[1])
                A.heapSize = A.heapSize - 1
                MaxHeapify(A,1)
        end for
```

Quick Sort

```
// Sorts the array A whose first and last indices are 'p' and 'r' respectively.
// For A is an one-based array QuickSort(A,1,A.length).
QuickSort(A,p,r)
        if (p < r)
                q = Partition(A,p,r)
                QuickSort(A,p,q-1)
                QuickSort(A,q+1,r)
        end if

// Rearranges the subarray A[p...r] in place.
Partition(A,p,r)
        x = A[r]
        i = p - 1
        for (j = p) to (r - 1)
                if (A[j] <= x)
                        i = i + 1
                        exchange (A[i]) with (A[j])
                end if
        end for
        exchange (A[i + 1]) with A[r]
        return (i + 1)
```

Insertion Sort

```
InsertionSort(A)

        // A is a one-based array.

        for j = 2 to A.length

                key = A[j]

                // Insert A[j] into the second sequence A[1...j - 1].

                i = j - 1

                while (i > 0) and (A[i] > key)

                        A[i + 1] = A[i]

                        i = i – 1

                end while

                A[i + 1] = key

        end for
```

MergeSort

```
// A: array
MergeSort(A)
        if (n == 1)
                return A
        end if
        leftArray = A[0] ... A[n/2]
        rightArray = A[n/2 + 1] ... A[n]

        leftArray = MergeSort(leftArray)
        rightArray = MergeSort(rightArray)

        return Merge(leftArray,rightArray)

// Merges the two arrays A and B together.
Merge(A,B)
        while (A and B still have elements)
                if (A[0] > B[0])
                        add B[0] to the end of C
                        remove B[0] from B
                else
                        add A[0] to the end of C
                        remove A[0] from A
                end if
        end while
        while (A has elements)
                add A[0] to the end of C
                remove A[0] from A
        end while

        while (B has elements)
                add B[0] to the end of C
                remove B[0] from B
        end while

        return C
```

code snippets

BINARY HEAP

```
template <class object>
void Heap<object>::insertIntoHeap(object no) {
    if (heapSize == heapArrayLength) { // heap full
        throw std::exception();
    } else {
        heapSize++;
        heapArray[heapSize] = no;
        HeapifyUp(heapSize);


    }

}

template <class object>
object Heap<object>::extractMax() { // could be template not int
    object reqNode = heapArray[1];
    heapArray[1] = heapArray[heapSize]; // swapping node 1 with last element
    heapSize--; //delete last element
    HeapifyDown(1);
    return reqNode;

}


/**
to build the tree
*/
template <class object>
void Heap<object>::HeapifyUp(int index)
{
    if (index > 1 && heapArray[getParentIndex(index)]< heapArray[index])
    {
        swap(heapArray[index],heapArray[getParentIndex(index)]);
        HeapifyUp(getParentIndex(index));
    }
}

template <class object>
void Heap<object>::buildMaxHeap(object arrayToBeBuilt[], int sizeOfArray) {
    //copies the array
    for (int i = 0; i < sizeOfArray; i++) {
        heapArray[i+1] = arrayToBeBuilt[i];
    }
    //  memcpy (heapArray, arrayToBeBuilt, sizeOfArray);
    for (int k = sizeOfArray/2; k > 0; k--)
    {
        HeapifyDown(k);
    }
    heapSize = sizeOfArray;
}
```

SORTING ALGORITHMS

```
void BubbleSort::sort(int *List, int siz) {
    if (siz == 1) {
        return ;
    }
    for (int i = 0; i < siz-1; i++) {
        if (List[i+1] < List[i]) {
            swap(List[i],List[i+1]);
        }
    }
    sort(List, siz - 1);}

void Heapsort::sort(int *arrayToBeSorted, int sizeOfArray) {
    Heap<int> heap(sizeOfArray);
    heap.buildMaxHeap(arrayToBeSorted, sizeOfArray);

    for (int i = sizeOfArray - 1; i >= 0; i--) {
        swap(heap.getHeapArray()[0], heap.getHeapArray()[i]);
        int size = heap.getHeapsize();
        size--;
        heap.setHeapSize(size);
        heap.HeapifyDown(i);
    }
}

int InsertionSort::sort(int *List, int siz) {
    if (siz <= 1) {
        return siz;
    }
    siz = sort(List, siz -1);
    int key = List[siz];
    int i = siz - 1;
    while ((i >= 0) && (List[i] > key)) {
        List[i+1] = List[i];
        i--;
    }
    List[i+1] = key;
    return siz + 1;
}

void MergeSort::sort(int *List, int low, int high) {
    if (low < high) {
        int mid = (low + high)/2;
        sort(List,low, mid);
        sort(List,mid + 1, high);
        merge(List,low, mid, high);
    }

}

void SelectionSort::sort(int *list, int start) {
    if ( start >= siz - 1 )
        return;
    int minIndex = start;
    for ( int i = start + 1; i < siz; i++ ) {
        if (list[i] < list[minIndex] )
            minIndex = i;
    }
    swap(list[start],list[minIndex]);
    sort(list, start + 1);

}
```

```
void MergeSort::merge(int *List, int low, int mid, int high) {
    int h = low, i = low, j = mid+1, k;

    int aux [siz] ;

    while ((h <= mid) && (j <= high)) {
        if (List[h] <= List[j]) {
            aux[i] = List[h];
            h++;
        } else {
            aux[i] = List[j];
            j++;
        }
        i++;
    }

    if (h > mid) {
        for (k=j; k<=high; k++) {
            aux[i] = List[k];
            i++;
        } } else {
        for (k=h; k<=mid; k++) {
            aux[i] = List[k];
            i++;
        }
    }
    for (k=low; k<=high; k++)
        List[k] = aux[k];

}


void QuickSort::quickSort(int *arr, int left, int right) {
    int pivot = Partition(arr, left, right);
    if (left < pivot - 1)
        quickSort(arr, left, pivot - 1);
    if (pivot < right)
        quickSort(arr, pivot, right);
}

/**
 * computing pivot for quick sort
 * @param arr
 * @param left
 * @param right
 * @return
 */
int QuickSort::Partition(int *arr, int left, int right) {
    int pivot = arr[(left + right) / 2];
    while (left <= right) {
        while (arr[left] < pivot)
            left++;
        while (arr[right] > pivot)
            right--;
        if (left <= right) {
            swap(arr[left],arr[right]);
            left++;
            right--;
        }
    }
    return left;
}
```

Data structures

The code is written in c++.

- The binary heap is implemented using an array of known size scanned by the user.
- All the heap operations are done on this heap array
- The constructor of the binary heap takes only the size of the array as a parameter to be able to build an array of known size
- The priority queue data structure and its default operations are implemented using the heap.

The heap following operations are implemented:

i. The MAX-HEAPIFY
ii. The BUILD-MAX-HEAP
iii. The HEAPSORT
iv. The MAX-HEAP-INSERT
v. HEAP-EXTRACT-MAX

- All the sorting algorithms are implemented using arrays. The sorting methods rearrange the sequence and either reprint them or return a pointer to the original, now sorted, array.
- The class sort combines the different algorithms of sorting.
- Templates allows general use of the binary heap :Any type of data can be sorted in the heap

Sample runs

```
// INSERT
        heap.insertIntoHeap(3);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        heap.insertIntoHeap(10);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        heap.insertIntoHeap(2);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        heap.insertIntoHeap(20);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        heap.insertIntoHeap(15);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
```

```
 size of array = 1
3

 --------------------------------
 size of array = 2
10
3

 --------------------------------
 size of array = 3
10
3
2

 --------------------------------
 size of array = 4
20
10
2
3

 --------------------------------
 size of array = 5
20
15
2
3
10
```

```
// BUILD
        cout << "\n BUILDING HEAP \n";
        int A[5] = {1,12,70,85,3};
        heap.buildMaxHeap(A, 5);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());

// insert after build

        heap.insertIntoHeap(20);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        heap.insertIntoHeap(15);
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
```

```
 BUILDING HEAP

 size of array = 5
85
12
70
1
3

 size of array = 6
85
12
70
1
3
20

 --------------------------------
 size of array = 7
85
12
70
1
3
20
15
```

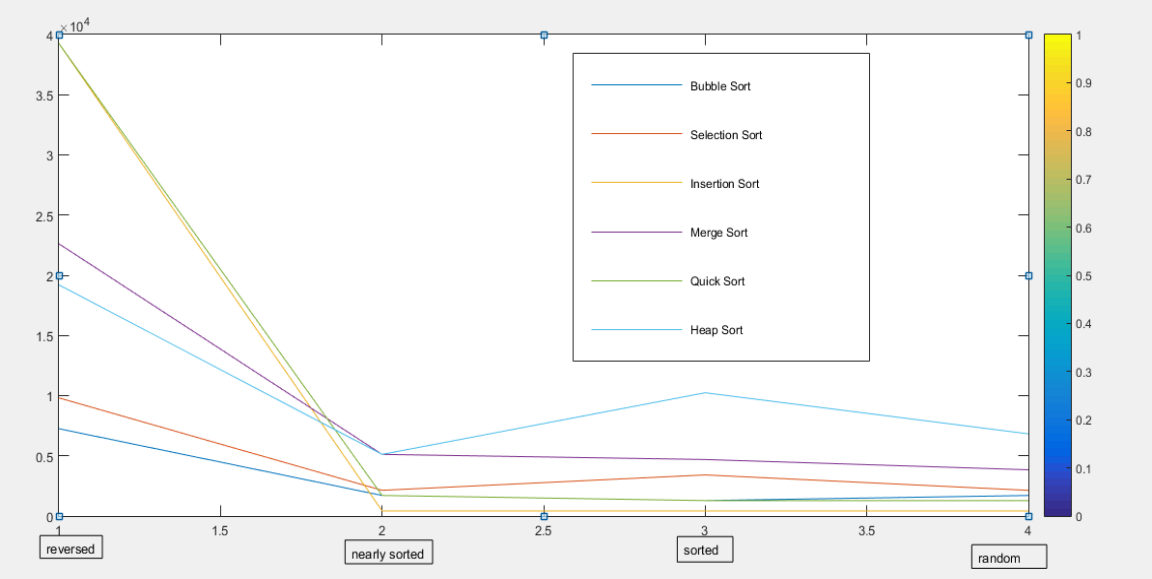
```
// EXTRACT
        cout << "\n max extracted is : " << (int) heap.extractMax();
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        cout << "\n max extracted is : " << (int) heap.extractMax();
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
        cout << "\n max extracted is : " << (int) heap.extractMax();
        m.printArray(heap.getHeapArray(), heap.getHeapsize());
        cout << "\n --------------------------------";
```

```
 max extracted is : 85
 size of array = 6
70
12
20
1
3
15

 --------------------------------
 max extracted is : 70
 size of array = 5
20
12
15
1
3

 --------------------------------
 max extracted is : 20
 size of array = 4
15
12
3
1
```

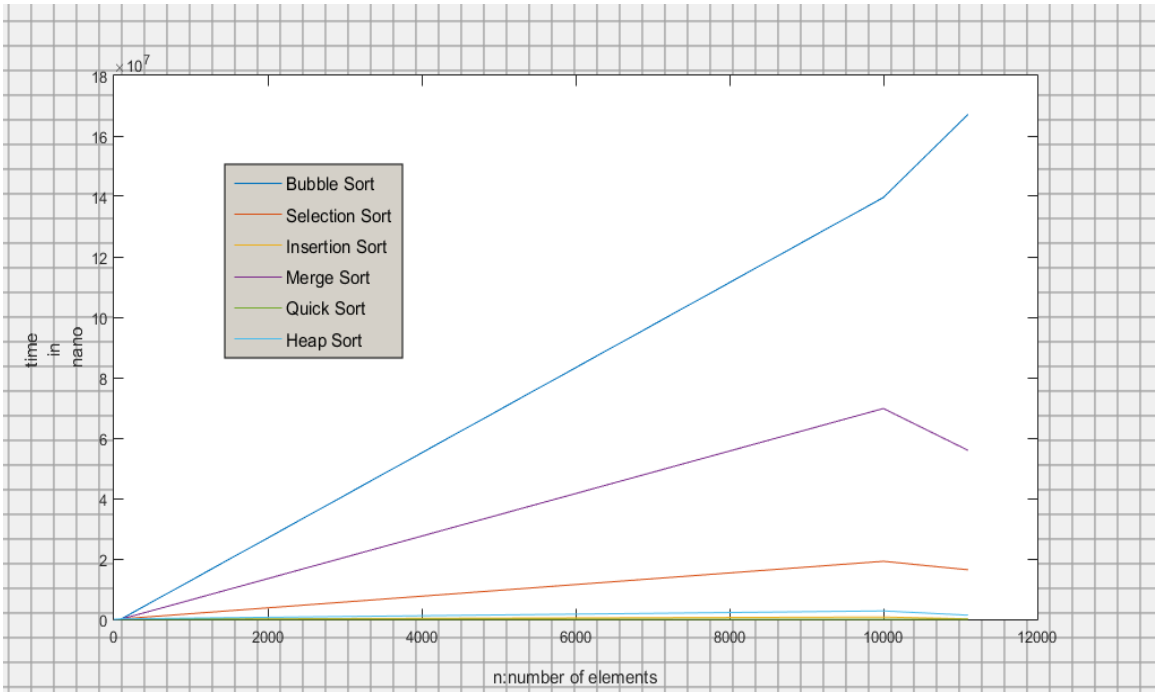
```
//        SORTING

        cout << " \n SORTING :: \n" ;
        int arr[] = {-1, 10, -999, 20, 1, 2, 3, -100, 200, 20000, 99999, -99999};
    int size = sizeof(arr)/sizeof(*arr);
    Sort x(size);
        cout << "bubble sort : ";
    x.bubbleSort(arr,size);
    for (int i=0;i<x.getSize();i++)
        cout<<arr[i]<<",";
    cout<<"\n";
        cout << "insertion sort : ";
    x.insertionSort(arr,size);
    for (int i=0;i<x.getSize();i++)
        cout<<arr[i]<<",";
    cout<<"\n";
        cout << "megerSort sort : ";
    x.mergeSort(arr,size);
    for (int i=0;i<x.getSize();i++)
        cout<<arr[i]<<",";
    cout<<"\n";
        cout << "quick sort : ";
    x.quickSort(arr,size);
    for (int i=0;i<x.getSize();i++)
        cout<<arr[i]<<",";
    cout<<"\n";
        cout << "selection sort : ";
    x.selectionSort(arr,size);
    for (int i=0;i<x.getSize();i++)
        cout<<arr[i]<<",";
    cout<<"\n";
        cout << "heap sort : ";
    x.heapSort(arr,size);


        SORTING ::
       bubble sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
       insertion sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
       megerSort sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
       quick sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
       selection sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
       heap sort : -99999,-999,-100,-1,1,2,3,10,20,200,20000,99999,
```

Graphs

1. The x axis represents the data type (reversed, sorted, nearly sorted, random) and the y axis is the time in nanoseconds



2. The x axis represents the number of elements n and the y axis is the time in nanoseconds



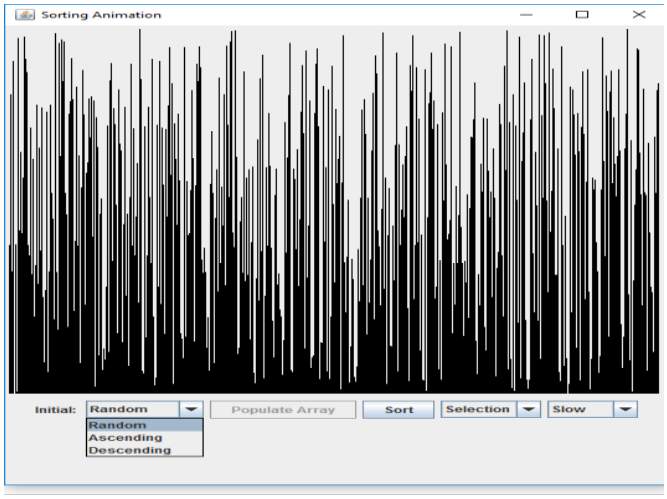
MATLAB CODE FOR THE GRAPH

```
>> x=[1,4,8,12,20,100,10000,11100];
>> y1=[2986,2560,1707,2560,6827,156160,139625634,167131519];
>> plot(x,y1)
>> hold on
>> y2=[3414,2986,1280,1707,4693,113066,19189309,16366912];
>> plot(x,y2)
>> y3=[28160,16213,854,426,1280,9814,737279,139094];
>> plot(x,y3)
>> y4=[5973,9814,4266,6827,11947,72106,69767591,55906062];
>> plot(x,y4)
>> y5=[34560,2986,1280,2133,2987,3413,2987,2987];
>> plot(x,y5)
>> y6=[11946,3840,4693,6400,20907,132266,2754984,1344852];
>> plot(x,y6)
>> hold off
```

Extra Work

GRAPHICAL ILLUSTARTION

The program allows graphical illustration for different sorting types as insertion, selection, quick and bubble

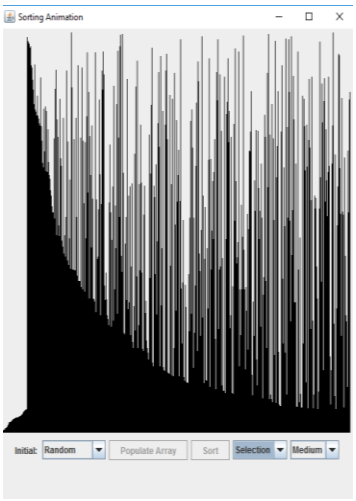


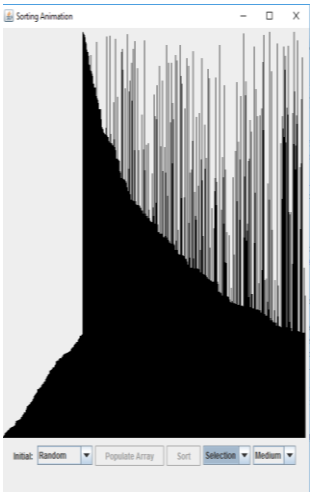
As shown there are 3 array's choices: random, ascending or descending.

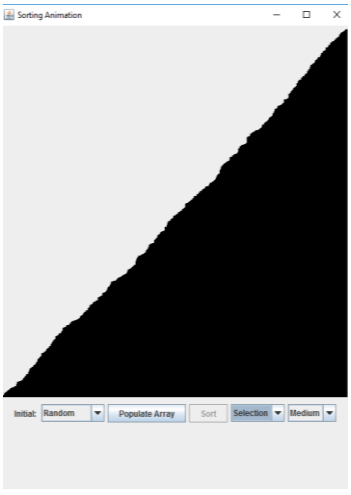
And three speed modes: slow, medium and fast.

Note: The graphical illustration code for the different sorting algorithms was written using Swing -Java to be able to create a graphical user interface although the code is written in c++.

Example for Selection sort







Code Snippet for the GUI:

```
public class SortPanel extends JPanel
{
	private JButton fillArray = new JButton("Populate Array");
	private JButton sortButton = new JButton("Sort");
	private JButton stopButton = new JButton("Stop");
	private String[] sortMethods = { "Selection", "Quick", "Bubble",
"Insertion"};
	private JComboBox sortTypes = new JComboBox<>(sortMethods);
	private String[] dataType = { "Random", "Ascending", "Descending"};
	private JComboBox initialOrdering = new JComboBox<>(dataType);
	private int[] integerArray = new int[525];
	int speed = 500;
	private String[] speedArray = { "Slow", "Medium", "Fast"};
	private JComboBox speeds = new JComboBox<>(speedArray);

	/**
	Constructor
	*/
	public SortPanel()
	{
		//Create a SortAnimationPanel and add it to the sort panel
		final SortAnimationPanel topPanel = new SortAnimationPanel();
		this.add(topPanel);

		//Create a panel to hold the controls
		JPanel controlPanel = new JPanel();
		controlPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
		controlPanel.add(new JLabel("Initial: "));
		controlPanel.add(initialOrdering);
		controlPanel.add(fillArray);
		controlPanel.add(sortButton);
		controlPanel.add(sortTypes);
		controlPanel.add(speeds);
		//controlPanel.add(stopButton);

		//Disable the sort button and stop button
		sortButton.setEnabled(false);
		stopButton.setEnabled(false);

		//Add the control panel to the sort panel
		this.add(controlPanel);
		controlPanel.setVisible(true);

		//When the populate array button is pressed fill it with random
numbers and display it.
		//Also enable the sort button and disable the populate button
		fillArray.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent e)
		{

				Random rand = new Random();
				for(int i=0; i< integerArray.length; i++) {
					integerArray[i] = rand.nextInt((530 - 1) + 1)
+ 1;
				}

				//Grab the initial array ordering
				String selectedValue =
initialOrdering.getSelectedItem().toString();
```

```
//Show the array in the selected ordering
                        if(selectedValue.equals("Ascending"))
                        {
                              Arrays.sort(integerArray);
                              repaint();
                        }

                        if(selectedValue.equals("Descending"))
                        {
                              int temp;
                              int start = 0;
                              int end = integerArray.length-1;

                              //Sort the array in ascending order
                              Arrays.sort(integerArray);

                              //Reverse the array so it is not in
descending order
                              while(start < end)
                              {
                                    temp = integerArray[start];
                                    integerArray[start] =
integerArray[end];
                                    integerArray[end] = temp;
                                    start++;
                                    end--;
                              }
                                    repaint();
                        }

                        if(selectedValue.equals("Random"))
                        {
                              repaint();
                        }


                        //Disable populate button and enable sort button
                        sortButton.setEnabled(true);
                        fillArray.setEnabled(false);
                  }
            });
```

PRIORITY QUEUE:

The priority queue is implemented using the binary heap:

The following operations of the priority queue are allowed: insert, extract maximum, get array and get the size.

TEMPLATES:

The program allows the heap to take any type of data using templates.

SORTING:

The six sorting techniques are implemented: bubble, insertion, merge, heap, quick, and selection.