Name : Ahmed Mohamed Fathallah . ID : 14.

Lab#1 Report

Binary Heaps :

The heap is implemented using an array and two attributes , length of the array and the heapSize which is the number of stored nodes in the array . in case the input array is larger than the heap array , the heap array is expanded .

- **Max Heapify :**

  It is a procedure that given an index for a heap node that is not heapified , it swaps the node with the max. of its children . Then in order to make sure the rest of the heap is correct it goes on to the swapped child and does the same till it reaches a leaf node.

  The procedure runs in O(log n) time .

```
public void maxHeapify(int i){

		while(2*i<=heapSize){
			int left,right,max;
			left=arr[2*i];
			max = Math.max(arr[i],left);
			if(2*i+1<=heapSize){
				right = arr[2*i+1];
				max = Math.max(max,right);
			}
			if(max==arr[i])
				break;
			else if(max==arr[2*i]){
				int temp = arr[i];
				arr[i]=arr[2*i];
				arr[2*i]=temp;
				i=2*i;
			}
			else{
				int temp = arr[i];
				arr[i]=arr[2*i+1];
				arr[2*i+1]=temp;
				i=2*i+1;
			}
		}

}
```

- **Build Max Heap :**

The procedure takes an array as a parameter and builds a binary heap out of this array . this is done by going through each node except for leaf nodes , and performing maxHeapify method on it .

It runs in O(n) time .

```
public void buildMaxHeap(int[] a){
		if(a.length>this.length)
			expand(a.length);
		for(int i=1;i<=a.length;i++){
			arr[i]=a[i-1];
		}
		heapSize=a.length;
		for(int i=heapSize/2;i>0;i--)
			maxHeapify(i);

}
```

- **Max Heap Insert :**

The procedure takes an integer as a parameter and inserts this integer into the pre-constructed binary heap reserving the heap properties . this is done by inserting the heap as a leaf node then checking if it's smaller than or equal to its parent then the heap is correct , other than that the element is swapped with its parent and going up to the parent and doing the same again .

It runs in O(log n) time.

```
public void insert(int node){
	if(heapSize==length)
		expand(length);

	arr[++heapSize]=node;
	int i = heapSize;
	while(i>1){
		if(arr[i]<=arr[i/2])break;

		int temp=arr[i];
		arr[i]=arr[i/2];
		arr[i/2]=temp;
		i=i/2;
	}
}
```

- **Heap Extract Max :**

  This procedure extracts the max element from the heap and returns it. This happens by swapping the root of the tree with the last leaf node , then performing MaxHeapify on the new root of the tree after decrementing the heapSize .

  It runs in O(log n) time .

```
public int extractMax(){
	int temp = arr[1];
	arr[1]=arr[heapSize];
	arr[heapSize]=temp;
	heapSize--;
	maxHeapify(1);
	return temp;
}
```

- **Heap Sort :**

  Given an array to be sorted this procedure builds a binary heap out of this array , then it extracts the maximum of this heap at every iteration in order to sort this array till the heap is empty .

  It runs in O(n log n) time.

```
public void heapsort(int[] a){
	buildMaxHeap(a);
	for(int i=heapSize;i>0;i--){
		a[i-1]=extractMax();
	}
}
```

Sorting Techniques :

Implemented algorithms with complexity $O(n^2)$: bubble , selection , insertion sorts .

Implemented algorithms with complexity O(n log n) : Merge , Quick sorts .

- **Bubble sort :**

```
public void bubble(int[] arr){
	for(int i=0;i<arr.length;i++){
		for(int j=0;j<arr.length-1;j++){
			if(arr[j]>arr[j+1]){
				int temp=arr[j];
				arr[j]=arr[j+1];
				arr[j+1]=temp;
			}
		}
	}
}
```

- **Selection sort :**

```
public void selection(int[] arr){
	for(int i=0;i<arr.length-1;i++){
		int min=arr[i],minj=i;
		for(int j=i+1;j<arr.length;j++){
			if(arr[j]<min){
				min=arr[j];
				minj=j;
			}
		}
		int temp=arr[i];
		arr[i]=arr[minj];
		arr[minj]=temp;
	}
}
```

- **Insertion sort :**

```
public void insertion(int[] arr){
	for(int i=1;i<arr.length;i++){
		int temp=arr[i],j;
		for(j=i-1;j>=0 && temp<arr[j] ;j--){
			arr[j+1]=arr[j];
		}
		arr[j+1]=temp;
	}

   }
```

- **Merge Sort :**

```
int[] temp;

public void merge(int[] arr,int lo,int hi,boolean flag){
	if(flag){
		temp = new int[arr.length];
	}
	int mid=(lo+hi)/2;
	if(lo<hi){
		merge(arr,lo,mid,false);
		merge(arr,mid+1,hi,false);
		merging(arr,lo,mid,hi);
	}
}
private void merging(int[] arr,int lo ,int mid,int high){
	int l=lo,r=mid+1,i=lo;

	while(l<=mid && r<=high){
		if(arr[l]<=arr[r])
			temp[i++]=arr[l++];
		else
			temp[i++]=arr[r++];
	}

	for(int j=l;j<=mid;j++)
		temp[i++]=arr[j];

	for(int j=r;j<=high;j++)
		temp[i++]=arr[j];

	for(int j=lo;j<=high;j++)
		arr[j]=temp[j];
}
```

- **Quick Sort :**

```
public void quick(int[] arr,int left,int right){
          if(sorted(arr))
                return;
          if(inverted(arr)){
                for(int i=0;i<arr.length/2;i++){
                      int temp = arr[i];
                      arr[i]=arr[arr.length-1-i];
                      arr[arr.length-1-i]=temp;
                }
                return;
          }
          if(right-left<=0)
                return;

          int pivot=left,k=left;
          for(int i=left+1;i<=right;i++){
                if(arr[i]<arr[pivot]){
                      int temp = arr[i];
                      arr[i]=arr[++k];
                      arr[k]=temp;
                }
          }
          int temp = arr[pivot];
          arr[pivot]=arr[k];
          arr[k]=temp;

          quick(arr,left,k-1);
          quick(arr,k+1,right);

   }
```

Performance Testing :

Testing different sorting algorithms performance by recording their running times on sorting the same set of data and varying the data set size from 10 to 100000 .

Size	10	100	1000	10000	100000
Heap	0	0	0	10	10
Bubble	0	0	10	220	22091
Selection	0	0	0	40	3492
Insertion	0	0	10	10	1221
Merge	0	0	0	0	20
Quick	0	0	10	30	3562