**NATIONAL UNIVERSITY OF COMPUTER & EMERGING SCIENCES**

**CL 201-Data Structures**

**Lab Session 10**

**Merge Sort:**

**#include <iostream>**

**using namespace std;**

**class Sorting**

**{**

**public:**

**void merge(int a[], const int low, const int mid, const int high)**

**{**

**// Variables declaration.**

**int \* b = new int[high+1-low];**

**int h,i,j,k;**

**h=low;**

**i=0;**

**j=mid+1;**

**// Merges the two array's into b[] until the first one is finish**

**while((h<=mid)&&(j<=high))**

**{**

**if(a[h]<=a[j])**

**{**

**b[i]=a[h];**

**h++;**

**}**

**else**

**{**

**b[i]=a[j];**

**j++;**

**}**

**i++;**

**}**

**// Completes the array filling in it the missing values**

**if(h>mid)**

**{**

**for(k=j;k<=high;k++)**

**{**

**b[i]=a[k];**

**i++;**

**}**

**}**

**else**

**{**

**for(k=h;k<=mid;k++)**

**{**

**b[i]=a[k];**

**i++;**

**}**

**}**

**// Prints into the original array**

**for(k=0;k<=high-low;k++)**

**{**

**a[k+low]=b[k];**

**}**

**delete[] b;**

**}**

**void Print(int a[],int b,int c)**

**{**

**for(int k=b;k<=c;k++)**

**{**

**cout<<a[k]<<endl;**

**}**

**}**

**void merge\_sort(int a[], const int low, const int high)// Recursive**

**{**

**int mid;**

**if(low<high)**

**{**

**mid=(low+high)/2;**

**merge\_sort(a, low,mid);**

**merge\_sort(a, mid+1,high);**

**merge(a, low,mid,high);**

**}**

**}**

**};**

**int main()**

**{**

**int arraySize=11;**

**int a[]={2,11,3,43,21,211,3,22,12,44,122};**

**Sorting s;**

**s.merge\_sort(a, 0, (arraySize-1) );**

**s.Print(a,0,(arraySize-1));**

**system("pause");**

**return 0;**

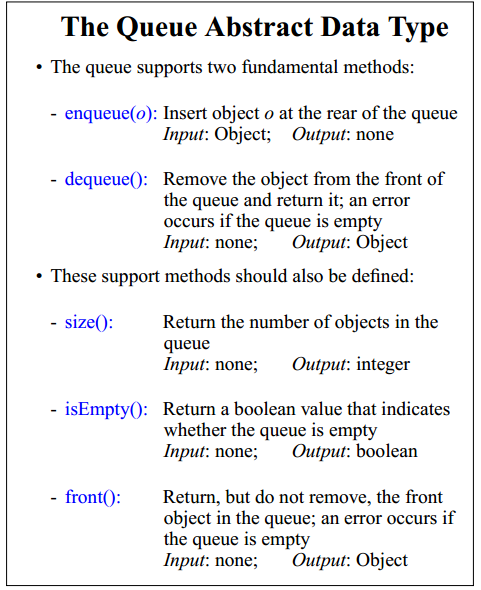
**}**

**Task1: Before running the above complete code, run debug on merg\_sort function to check the recursive calls behavior to partition array into one element (sorted already);**

**Queue:**

A queue is a particular kind of abstract or collection. In which its insertion and removal routines follows the ﬁrst-in-ﬁrst-out (FIFO) principle.

* First In First Out (FIFO).  Insertions at the "end" of the queue, and removals from the "front" of the queue.
* Analogy - waiting in line for a ride at an amusement park.  Get in line at the end.  First come, first serve.
* A queue class will have two primary operations:
  + **enqueue** -- Elements are inserted at the rear (i.e. at the back of the line)
  + **dequeue** -- removed from the front .
* Typical application areas include print job scheduling, operating systems (process scheduling).



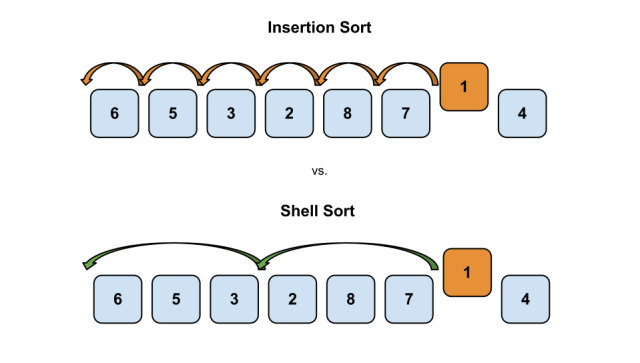
**Task1:**

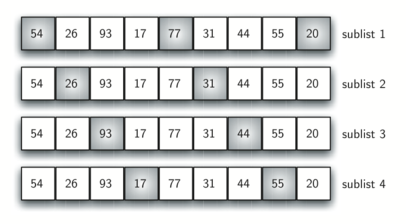
**Create Queue and implement all functions.**

**Shell Sort:**

Insertion sort is a great algorithm, because it’s very intuitive and it is easy to implement, but the problem is that it makes many exchanges for each “light” element in order to put it on the right place. Thus “light” elements at the end of the list may slow down the performance of insertion sort a lot. That is why in 1959 Donald Shell proposed an algorithm that tries to overcome this problem by comparing items of the list that lie far apart.

In the other hand it is obvious that by comparing items that lie apart the list can’t be sorted in one pass as insertion sort. That is why on each pass we should use a fixed gap between the items, then decrease the value on every consecutive iteration.





|  |
| --- |
| **/\*Safia Baloch: FAST-NUCES\*/**  **#include<iostream>**  **using namespace std;**  **class shellsort**  **{**  **private:**  **int no\_of\_elements;**  **int elements[10];**  **public:**  **void getarray();**  **void sortit(int [],int);**  **int return\_no\_elements();**  **void print();**  **};**  **void shellsort::getarray()**  **{**  **cout<<"how many elements: ?";**  **cin>>no\_of\_elements;**  **cout<<"insert elemet to sort:"<<endl;**  **for(int i=0;i<no\_of\_elements;i++)**  **{**  **cin>>elements[i];**  **}**  **}**  **int shellsort:: return\_no\_elements()**  **{**  **return no\_of\_elements;**  **}**  **void shellsort::sortit(int inc[],int incnum)**  **{**  **int incr,j,k,span, y;**  **for(incr=0;incr<incnum;incr++)**  **{**  **span=inc[incr];**  **for(j=span;j<no\_of\_elements;j++)**  **{**  **y=elements[j];**  **for(k=j-span;k>=0&& y<elements[k]; k-=span)**  **{**  **elements[k+span]=elements[k];**  **}**  **elements[k+span]=y;**  **}**  **cout<<"iteration= "<<incr+1<<"span = "<<span<<" : ";**  **print();**  **if(span==1)**  **break;**  **}**  **}**  **void shellsort:: print()**  **{**  **for(int i=0;i<no\_of\_elements;i++)**  **{**  **cout<<elements[i]<<" ";**  **}**  **cout<<endl;**  **}**  **int main()**  **{**  **shellsort ss;**  **int n,i,j;**  **ss.getarray();**  **n=ss.return\_no\_elements();**  **int incrm[n];**  **for(i=n,j=0;i>0;i=i/2,j++)**  **{**  **incrm[j]=i;**  **}**  **ss.sortit(incrm,j+1);**  **//system("pause");**  **}** |

**Task2: Run the above code , give a print of different output on a page and a dry run of sort code.**

**Radix Sort:**

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.

Basic Steps to Be Performed:

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process[.](http://adf.ly/KiLQs) In other words:

1. Take the least significant digit of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

|  |
| --- |
| **/\*Safia Baloch: FAST-NUCES\*/**  **#include <iostream>**  **#include <conio.h>**  **#define MAX 10**  **using namespace std;**  **class radixsort{**  **int arr[MAX],n;**  **public:**  **void getdata()**  **{**  **cout<<"How many elements you require : ";**  **cin>>n;**  **for(int i=0;i<n;i++)**  **cin>>arr[i];**  **}**  **void showdata()**  **{**  **cout<<"\n--Display--\n";**  **for(int i=0;i<n;i++)**  **cout<<arr[i]<<" ";**  **}**  **void sortLogic()**  **{**  **//for base 10int temp;**  **int bucket[10][20], buck\_count[10], b[10];**  **int i,j,k,r,no\_of\_passes=0,divisor=1,largest,pass\_no;**  **largest=arr[0];**  **for(i=1;i<n;i++) //Find the largest Number**  **{**  **if(arr[i] > largest)**  **largest=arr[i];**  **}**  **while(largest > 0) //Find number of digits in largest number**  **{**  **no\_of\_passes++;**  **largest /= 10;**  **}**  **for(pass\_no=0; pass\_no < no\_of\_passes; pass\_no++)**  **{**  **for(k=0; k<10; k++)**  **{**  **buck\_count[k]=0; //Initialize bucket countfor(i=0;i<n;i++){**    **}**  **for (i=0 ; i<n ; i++)**  **{**  **r=(arr[i]/divisor) % 10;**  **bucket[r][buck\_count[r]++]=arr[i];**  **}**  **i=0; //collect elements from bucketfor(k=0; k<10; k++){**  **for (k=0;k<10;k++)**  **for(j=0; j<buck\_count[k]; j++)**  **arr[i++] = bucket[k][j];**    **divisor \*= 10;**  **}**  **}**  **};**  **int main(){**    **cout<<"\n\*\*\*\*\*Radix Sort\*\*\*\*\*\n";**  **radixsort obj;**  **obj.getdata();**  **obj.sortLogic();**  **obj.showdata();**    **}** |

**Lab Task:**

**Take file of unsorted elements, put all above soring functions in single program and calculate the time.**

**Quick Sort:**

|  |
| --- |
| **#include <iostream>**  **#include <conio.h>**  **using namespace std;**  **static const int size =10;**  **//int arr1[size] = {12,24,17,34,6,10,6,3,8,9,1};**  **//int arr4[size] = {1,2,3,4,5,6,7,8,9,10,11};**  **//int arr5[size] = {8,5,4,7,6,1,6,3,8,12,10};**  **int arr6[size] = {60,24,1,33,99,45,78,2,11,23};**  **class Sorting**  **{**  **public:**  **void PrintArray(int input[])**  **{**  **for ( int i = 0; i < size; i++ )**  **cout << input[i] << " ";**  **cout << endl;**  **}**  **void Swap(int\* i, int\* j)**  **{**  **int tmp;**  **tmp = \*i;**  **\*i = \*j;**  **\*j = tmp;**  **}**  **};//end of class Sorting**  **int main()**  **{**  **Sorting sort;**  **getch();**  **return 0;**  **}** |

**Step 2: Quick Sort**

|  |
| --- |
| **void quick\_sort(int data[], int low, int high)**  **{**  **int i = low;**  **int j = high;**  **int pivot = data[low + ((high - low)/2)];**    **while (i <= j)**  **{**  **while (data[i] < pivot) i++;**  **while (data[j] > pivot) j--;**  **if (i <= j)**  **{**  **Swap(&data[i], &data[j]);**  **i++;**  **j--;**  **}**  **}**  **if (low < j)**  **quick\_sort(data, low, j);**  **if (i < high)**  **quick\_sort(data, i, high);**  **}** |
| ***Function call from main***  **sort.quick\_sort(arr6,0,size-1);** |

**Home Task:**

**Task2: Modify insertion sort for quick sort.**

**How to choose gap size?**

Not a cool thing about Shell sort is that we’ve to choose “the perfect” gap sequence for our list. However this is not an easy task, because it depends a lot of the input data. The good news is that there are some gap sequences proved to be working well in the general cases.

1. **Shell Sequence**

Donald Shell proposes a sequence that follows the formula FLOOR (N/2k), then for N = 1000, we get the following sequence: [500, 250, 125, 62, 31, 15, 7, 3, 1]

### Pratt Sequence

Pratt proposes another sequence that’s growing with a slower pace than the Shell’s sequence. He proposes successive numbers of the form 2p3q or [1, 2, 3, 4, 6, 8, 9, 12, ...].

### Knuth Sequence

Knuth in other hand proposes his own sequence following the formula (3k – 1) / 2 or [1, 4, 14, 40, 121, ...]

**Task4: Try to implement 3 gape techniques in shell sort**