**Experiment 24**

**Quick Sort and Merge Sort**

**Date:** 07-02-2021

**Aim:** Implement Quick sort and Merge Sort

**Data Structures used:** Array

**Algorithm for Quick Sort (Quicksort)**

**Input:** The array to be sorted and the index of the first and the last element

**Output :** The sorted Array

**Data Structure :** Array

**Steps**

* 1. Step 1: Start
  2. Step 2: if(first<last) then
  3. Step 1: q = Partition(arr,first, last)
  4. Step 2: Quicksort(arr,first,q-1)
  5. Step 3: Quicksort(arr,q+1,last)
  6. Step 3: endif
  7. Step 4: Stop

**Algorithm for Partition (Partition)**

**Input:** The array to be partitioned and the first and the last node(also known as the pivot)

**Output:** The correct index of the pivot in the sorted array

**Data Structure used:** Array

Steps

1. Step 1: Start
2. Step 2: pivot = arr[last]
3. Step 3: i = first-1
4. Step 4: j = first
5. Step 5: while j<=last-1 then
6. Step 1: if arr[j] <= pivot then
7. Step 1: i=i+1
8. Step 2: swap arr[i] and arr[j]
9. Step 2: EndIf
10. Step 6: End While
11. Step 7: swap arr[i+1] and arr[last]
12. return i+1

**Algorithm for Merge Sort (merge\_sort)**

**Input:** The array and the starting and the ending index of the array to be sorted

**Output :** The sorted array

**Data Structure used:** Array

Steps

1. Step 1: Start
2. Step 2: if(first<last) then
3. Step 1: (first+last)/2
4. Step 2: merge\_sort(arr,first,mid)
5. Step 3: merge\_sort(arr,mid+1,last)
6. Step 4: merge(arr,first,mid,last)
7. Step 3: Endif
8. Step 4: Stop

**Algorithm for Merge (merge)**

**Input:** The array and upperbound and the lower bound and the middle element in the array

**Output :** The array is sorted

**Data Structure used:** Binary trees

Steps

1. Step 1: Start
2. Step 2: n1 = middle-lower+1
3. Step 3: n2 = upper – middle
4. Step4: let L[1...n1+1] and R[1….n2+1]
5. Step 5: for i=1 to n1 do
6. Step 1: L[i] = arr[lower+i-1]
7. Step 6: Done
8. Step 7: for j = 1 to n2 do
9. Step 1: R[j] = arr[middle+j]
10. Step 8: done
11. Step9: L[n1+1] = ∞
12. Step 10: R[n2+1]= ∞
13. Step 11: i=1
14. Step 12: j=1
15. Step 13: for k=first to last
16. Step1: if L[i]<=R[j] then
17. Step 1: A[k] = L[i]
18. Step 2: i++
19. Step 2: else
20. Step 1: A[k] = R[j]
21. Step 2: j++
22. Step 3: endif
23. Step 14: Done
24. Step 15 :Stop

**Program Code**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#define MAX\_SIZE 100

typedef struct student\_structure{

char name[101];

float height;

float weight;

}student;

enum prop{NAME,HEIGHT,WEIGHT};

char prop\_name[][10]={"Name","Height","Weight"};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Quick Sort

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int partition(student \*list, int first, int pivot, enum prop a){

int i,j;

i = first;

j = first-1;

while(i<pivot){

int flag = 0;

switch(a){

case NAME:

if(strcmp(list[i].name,list[pivot].name)<=0) flag = 1;

break;

case HEIGHT:

if(list[i].height<=list[pivot].height) flag = 1;

break;

case WEIGHT:

if(list[i].weight<=list[pivot].weight) flag = 1;

break;

}

if(flag){

j++;

student temp = list[i];

list[i] = list[j];

list[j] = temp;

}

i++;

}

j++;

if(pivot != j){

student temp = list[pivot];

list[pivot] = list[j];

list[j] = temp;

}

return j;

}

void quick\_sort(student \*list,int first,int last,enum prop a){

if(first<last){

int q = partition(list, first,last,a);

quick\_sort(list, first, q-1,a);

quick\_sort(list,q+1,last,a);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Merge Sort

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void merge(student \*list,int first,int mid,int last,enum prop a){

int n = last-first+1;

student \*temp =(student\*)malloc(n\*sizeof(student)) ;

int i,j,flag,k=0;

for(i=first,j=mid+1;i<=mid&&j<=last;)

{

flag = 0;

switch(a){

case NAME:

if(strcmp(list[i].name,list[j].name)<0){

flag = 1;

}

break;

case HEIGHT:

if(list[i].height<=list[j].height){

flag =1;

}

break;

case WEIGHT:

if(list[i].weight<=list[j].weight){

flag =1;

}

break;

}

if(flag){ //if the flag is true then add i, else add j;

strcpy(temp[k].name,list[i].name);

temp[k].height= list[i].height;

temp[k].weight = list[i].weight;

i++;

}

else{

strcpy(temp[k].name,list[j].name);

temp[k].height= list[j].height;

temp[k].weight = list[j].weight;

j++;

}

k++;

}

while(i<=mid){

temp[k] = list[i];

i++;k++;

}

while(j<=last){

temp[k] = list[i];

j++;k++;

}

k=0;

for(i = first;i<=last;i++){

strcpy(list[i].name,temp[k].name);

list[i].height= temp[k].height;

list[i].weight = temp[k].weight;

k++;

}

}

void merge\_sort(student \*list, int first, int last, enum prop a)

{

if(first<last){

int mid = (first+last)/2;

merge\_sort(list,first,mid,a);

merge\_sort(list,mid+1,last,a);

merge(list,first,mid,last,a);

}

}

void list\_copy (student\* l1, student\* l2,int n){

for(int i=0;i<n;i++){

strcpy(l1[i].name,l2[i].name);

l1[i].height = l2[i].height;

l1[i].weight = l2[i].weight;

}

}

int main(){

student \*student\_list = (student\*) malloc(MAX\_SIZE\*sizeof(student));

student \*temp\_list = (student\*) malloc(MAX\_SIZE\*sizeof(student));

// FILE \*file = fopen("./output.txt","w");

char first\_name[50];

char last\_name[50];

int n = 0;

int i;

enum prop a = HEIGHT;

clock\_t t;

double time\_taken;

if(freopen("./student\_data.txt","r",stdin)){

FILE \*quickSortOp = fopen("./quicksortop.txt","w");

FILE \*mergeSortOp = fopen("./mergesortop.txt","w");

while(scanf("%s %s %f %f\n", first\_name,last\_name,&(student\_list[n].height),

&(student\_list[n].weight))==4) {

//conactinate the first and the last names

strcat(student\_list[n].name, first\_name);

strcat(student\_list[n].name, " ");

strcat(student\_list[n].name, last\_name);

n++;

}

fprintf(quickSortOp,"QUICK SORT\n");

fprintf(quickSortOp,"============\n");

// for(int a=NAME;a<=WEIGHT;a++ ){ //For iterating through all the

list\_copy(temp\_list,student\_list,n);

t = clock();

quick\_sort(temp\_list,0,n-1,a);

t = clock()-t;

i=0;

fprintf(quickSortOp,"Sorted according to order of the %s\n\n",prop\_name[a]);

while(i<n){

fprintf(quickSortOp,"%s %.2f %.2f\n",temp\_list[i].name,temp\_list[i].height,temp\_list[i].weight);

i++;

}

time\_taken = ((double)t)/(CLOCKS\_PER\_SEC);

fprintf(quickSortOp,"Time taken = %lf seconds",time\_taken);

fprintf(quickSortOp,"\n\n");

// }

fprintf(mergeSortOp,"MERGE SORT\n");

fprintf(mergeSortOp,"============\n");

// for(int a = NAME; a<=WEIGHT;a++){

list\_copy(temp\_list,student\_list,n);

t = clock();

merge\_sort(temp\_list,0,n-1,a);

t = clock()-t;

i=0;

fprintf(mergeSortOp,"Sorted according to order of the %s\n\n",prop\_name[a]);

while(i<n){

fprintf(mergeSortOp,"%s %.2f %.2f\n",temp\_list[i].name,temp\_list[i].height,temp\_list[i].weight);

i++;

}

time\_taken = ((double)t)/(CLOCKS\_PER\_SEC);

fprintf(mergeSortOp,"Time taken = %lf seconds",time\_taken);

fprintf(mergeSortOp,"\n\n");

// }

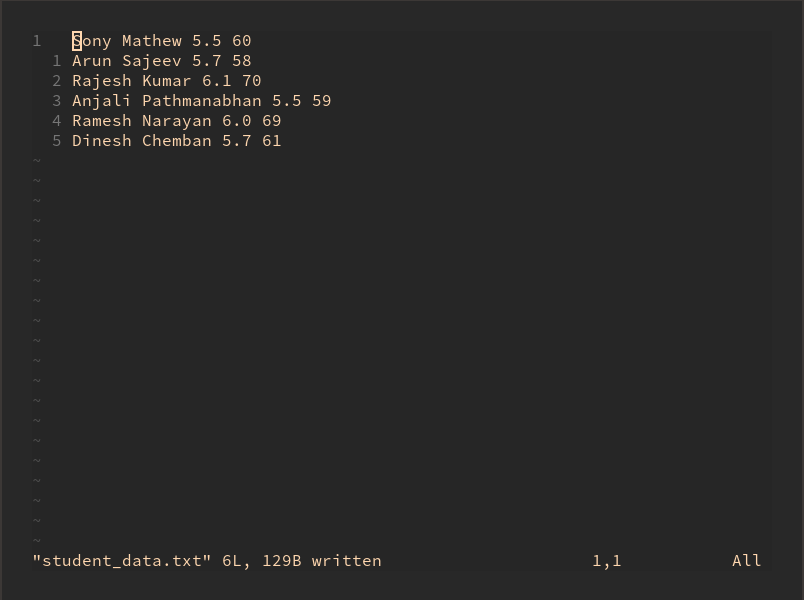
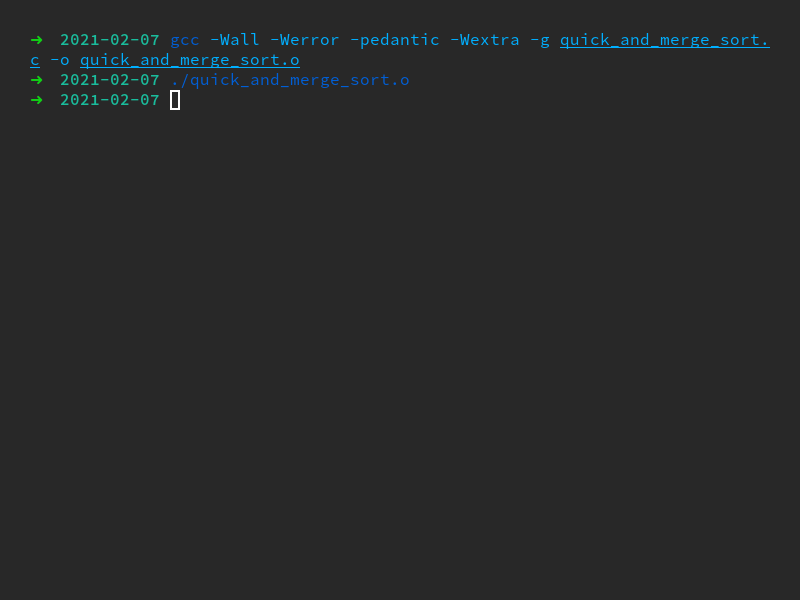
}

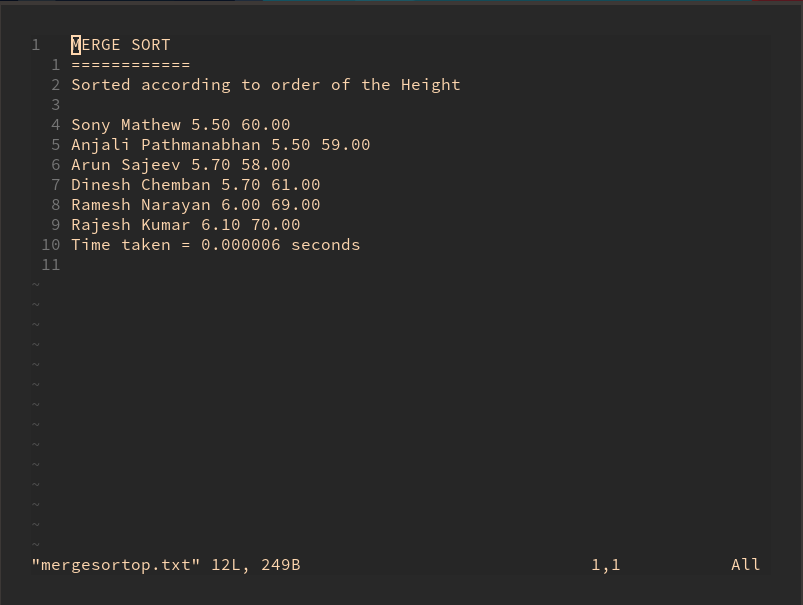
return 0;

}

**Result:** The program compiled successfully and required output was obtained

**Sample input and output**

****

****

**Experiment 25**

**Heap Sort**

**Date:** 07-02-2021

**Aim:** Sort an array of numbers using heap sort and find an element in the array using binary search

**Data Structures used:** Array

**Algorithm for Create Heap (create\_heap)**

**Input:** The array to be sorted and size of the array

**Output :** The elements of the array now follows the heap property

**Data Structure :** Array

**Steps**

1. Step 1: Start
2. Step 2: i = 1
3. Step 3: while i<=n do
4. Step 3: j = i
5. Step 4: while j>1 do
6. Step 1: if A[j] > A[j/2] then
7. Step 1: swap (A[j],A[j/2])
8. Step 2: j=j/2
9. Step 2: else
10. Step 1: j = 1
11. Step 3: endif
12. Step 5: EndWhile
13. Step 6: i = i+1
14. Step 4: endWhile
15. Step 5: Stop

**Algorithm for Remove max (remove\_max)**

**Input:** The largest element in the heap and the index

**Output:** The largest and the element at the bottom of the heap

**Data Structure used:** Array

Steps

1. Step 1: Start
2. Step 2: temp = A[i]
3. Step 3: A[i] = A[1]
4. Step 4: A[1] = temp
5. Step 5: Stop

**Algorithm for Rebuild Heap (rebuild\_heap)**

**Input:** The Array after the remove\_max algorithm

**Output:** The array satisfies the heap property

**Data Structure used:** Array

Steps

1. Step 1: Start
2. Step 2: if(i == 1)then
3. Step 1: retun
4. Step 3: else
5. Step 1: j = 0
6. Step 2: flag = true
7. Step 3: while(flage == true) do
8. Step 1: leftchild = j\*2
9. Step 2: rightchild = j\*2+1
10. Step 3: largest = j
11. Step 4: if(leftchild<=i and A[largest]<A[leftchild]) then
12. Step 1: largest = leftchild
13. Step 5: endIf
14. Step 6: if(rightchild<=i and A[largest]<A[rightchild]) then
15. Step 1: largest = rightchild
16. Step 7: endIf
17. Step 8: if(largest!=j) then
18. swap(A[j], A[largest])
19. Step 9: else
20. Step 1: flag = flase
21. Step 10: endif
22. Step 4: endWhile
23. Step 4: Endif
24. Step 5: Stop

**Result:** The program was compiled successfully and the required output was obtained

**Program Code**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Heap Sort

\* Done By: Rohit Karunakaran

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<stdio.h>

#include<stdlib.h>

void swap(int\* arr, int i, int j){

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

void create\_heap(int \*arr, int n){

int i = 0;

int k,j;

while(i<n){

j = i;

while(j>0){

k = j%2==0?j/2-1:j/2;

if(arr[j]>arr[k]){

swap(arr,j,k);

j = k;

}

else{

j=0;

}

}

i++;

}

//printf("Entered heap sort");

}

void heapify(int \*arr, int i){

//i is the upper bound

if(i == 0){

return; //the array is sorted

}

else{

int j=0;

int flag = 1;

while(flag){

int largest = j;//initially assume the parent is the largerst which in the first loop is'nt

int lc = 2\*j+1;

int rc = 2\*(j+1);

if(lc<=i && arr[lc]>arr[largest])largest = lc;

if(rc<=i && arr[rc]>arr[largest])largest = rc;

if(j!=largest){

swap(arr,j,largest);

}

else{

// printf("swapped\n");

flag =0; //if there is no change in the largest element then the array is heapified

}

}

}

}

void heap\_sort(int \*arr, int n){

create\_heap(arr,n);

for(int i = n-1;i>=0;i--){

swap(arr,i,0);

heapify(arr,i-1);

}

}

int binary\_search(int \*arr, int first, int last, int elem){

if(first<=last){

int mid = (first+last)/2;

if(arr[mid]== elem){

return mid;

}

else if(arr[mid]>elem){

return binary\_search(arr,first,mid-1,elem);

}

else{

return binary\_search(arr,mid+1,last,elem);

}

}

else{

return -1;

}

}

int main(){

int n;

int elem;

int\* arr = (int\*)malloc(20\*sizeof(int));

printf("Enter the number of elements: ");

scanf("%d",&n);

printf("Enter the elements : ");

for(int i = 0; i<n;i++){

scanf("%d",arr+i);

}

heap\_sort(arr,n);

printf("The sorted array is : ");

for(int i = 0;i<n;i++){

printf("%d ",arr[i]);

}

printf("\n");

printf("Enter the element to be searched: ");

scanf("%d", &elem);

int index = binary\_search(arr,0,n-1,elem);

if(index!=-1){

printf("The element is found at index %d\n",index);

}

else{

printf("The element doesnt exist\n");

}

free(arr);

return 0;

}

**Sample input/output**

