Daa file - daa practical file with c++ code

Design And Analysis Of Algorithm (Dr. A.P.J. Abdul Kalam Technical University)

Experiment 1

Page No.
Date 09 09 22

1.1 **Objective :-**
Program for recursive binary and linear search.

1.2 **Theory :-** (Binary Search)
Binary Search is a search algorithm that is used to find the position of an element (target value) in a sorted array.
The array should be prior to applying a binary search.
Binary search is also known as binary chop.

1.3 **Algorithm** (Binary Search)
→ Step 1 : find the middle element of array. using,
middle = initial_value + end_value /2;
→ Step 2 : If middle = element, return 'element found' and index.
→ Step 3 : If middle > element, call the function with end-value = middle - 1
→ Step 4 : If middle < element, call the function with start-value = middle + 1
→ Step 5 :- Exit.

1.4 **Code** (Binary Search)

```
# include <stdio.h>
int recursiveBinarySearch (int array[], int start_index,
                           int end_index, int element)
{
    if (end_index >= start_index) {
```

```
int middle = start_index + (end_index - start_index)/2;
if (array[middle] == element)
    return middle;
if (array[middle > element)
    return recursiveBinarySearch(array, start_index,
                middle-1, element);
return recursiveBinarySearch(array, middle+1, end_index, elemen
}
return -1;
}

int main(void){
    int array[] = {1, 4, 7, 9, 16, 56, 70};
    int n=7;
    int element=9;
int found_index = recursiveBinarySearch(array, 0, n-1, element
if (found_index == -1)
    printf("Element not found in the array");
else
    printf("Element found at index : %d", found_index);
}
return 0;
}
```

1.5 **Output**

array:- 1 2 3 4 5 6

key = 4

Output= 4

1.2. Theory (Linear Search) :-
Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

1.3. Algorithm (Linear Search) :-
→ Traverse the array
→ Match the key element with array element.
→ If key element is found, return the index position of the array element.
→ If key element is not found, return -1

1.4 Code (Linear Search)

```
# include <stdio.h>
int search (int arr[], int N, int x)
{ int i; for (int i=0; i<N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
int main (void)
{ int arr[] = {2,3,4,10,40};
    int x=10;
    int N= sizeof(arr) / sizeof(arr[0]);
  int result = search (arr, N, x);
  (result == -1) ? printf ("Element not found"): printf ("found %d", element);
}
```

Page No.
Date 16 09 22

Experiment 2

2.1 Objective :-
Write a program in C for Heap Sort.

2.2 Theory :-
Heap Sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the maximum or minimum element of the array. At each step, the root element of the heap get deleted and stored into the sorted array and the heap will again be heapified.
The heap sort basically recursively performs 2 main operations.
- Build a heap H, using the element of ARR
- Repeatedly delete the root element of heap formed in phase 1

2.3 Algorithm
→ Build a max heap from the input data.
→ At this point, the maximum element is stored at the root of the heap. Replace it with the last item of the heap followed by reducing size of the heap by 1. Finally, heapify the root of the tree
→ Repeat step 2 while the size of the heap is greater than 1.

2.4 Code
```
#include <stdio.h>
```

```
void heapify (int a[], int n, int i)
{  int largest = i;
   int left = 2*i + 1;
   int right = 2*i + 2;
   if (left < n && a[left] > a[largest])
         largest = left;
   if (right < n && a[right] > a[largest])
         largest = right;
   if (largest != i){
         int temp = a[i];
         a[i] = a[largest];
         a[largest] = temp;
   heapify (arr, N, largest);
}
}

void heapSort (int arr[], int n) {
      for (int i = N/2 - 1; i >= 0; i--)
            heapify (arr, N, i);
   for (int i = N - 1; i >= 0; i--) {
         swap (&arr[0], &arr[i]);
         heapify (arr, i, 0);
   }
}

int main () {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int N = sizeof (arr) / sizeof (arr[0]);
      heapSort (arr, N);
   return 0;
}
```

Page No.
Date 23 09 22

Experiment - 3

3.1 Objective

Write a program C to implement Merge Sort.

3.2 Theory

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as a sorting a subarray $A[p..r]$. Initially, $p=1$ and $r=n$. but these values changes as we recurse through subproblems.

To sort $A[p..r]$:

1. Divide Step

If a given array A has zero or one element simply return, it is already sorted. Otherwise split $A[p..r]$ into 2 parts each containing half of the elements of $A[p..r]$ This is q is the halfway point.

2. Conquer step

conquer by recursively sorting the 2 subarray $A[p..q]$ and $A[q+1..r]$

3. Combine step

Combine the elements back in $A[p..r]$ by merging the 2 sorted subarray. $A[p..q]$ & $A[q+1..r]$ into a sorted sequence. To accomplish this step. We will define a procedure MERGE (A, p, q, r).

33 Algorithm

1. MERGE-SORT (A, p, r) // check for base case
2. if p < r
3. then q = floor [(p+r)/2] // divide
4. MERGE (A, p, q) // conquer
5. MERGE (A, q+1, r) // conquer
6. MERGE (A, p, q, r) // conquer

34 Code

```
# include <stdio.h>
# include <stdlib.h>
void merge (int arr[], int l, int m, int r)
{
    int i, j, k, n1 = m - l + 1, n2 = r - m, L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

```
while (i<n1) {
    arr[k] = L[i];
    i++;
    k++;
}
while (j<n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSort (int arr[], int l, int r) {
    if (l<r) {
        int m = l + (r-l)/2;
        mergeSort (arr, l, m);
        mergeSort (arr, m+1, r);
        merge (arr, l, m, r);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printf("Given array
    mergeSort (arr, 0, arr_size-1);
    return 0;
}
```

Date 30/09/22
Page No.

Experiment - 4

4.1 **Objective**

Write a program in C to implement Selection Sort.

4.2 **Theory**

Selection sort is another algorithm that is used for sorting. This sorting algorithm, iterates through the array and finds the smallest number in the array and swap it with first element. if it smallest than the first element. Next, it goes on to the second element and so on until all elements are sorted.

Consider the array:

[10, 5, 2, 1]

The first element is 10. The next part we must find the smallest number from the remaining array. The smallest number from 5 2 1 is 1. So, we replace 10 by 1

The new array is [1,5,2,10] Again, this process is repeated.

finally we get sorted array as [1,2,5,10]

4.3 **Algorithm**

(1) Set min to the first location
(2) Search the mini element in the array
(3) swap the first location with the minimum value in the array
(4) assign the second element as mini
(5) Repeat process until we get sorted array.

vijeta

44 Code

```
#include <stdio.h>
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for(i=0; i<n-1; i++)
    {
        min_idx = i;
        for(j=i+1; j<n; j++)
            if(arr[j] < arr[min_idx])
                min_idx = j;
        if(min_idx != i)
            swap(&arr[min_idx], &arr[i]);
    }
}
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    return 0;
}
```

45 Output

```
Sorted array
11 12 22 25 64
```

vijeta

Date 7/10/22
Page No.

Experiment - 5

5.1 **Objective**

Write a program in C to implement insertion sort.

5.2 **Theory**

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.
For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate.
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average & worst case complexity of $O(n^2)$, where n is no. of items.

5.3 **Algorithm**

// Sort an arr[] of size n
insertionSort (arr, n)
Loop from i=1 to n-1
.... a) Pick element arr[i] & insert it into sorted sequence arr[...]

5.4 **Code**

```
#include <stdio.h>
void insertionSort (int arr[], int n)
{   int i, key, j;
    for (int i=1; i<n; i++)
    {   key = arr[i];
```

vijeta

```
        j = i-1;
    while (j >= 0 && arr[j] > key)
    {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
  }
}
void printArray(int arr[], int n)
{
    for(int i=0; i<n; i++)
        printf("%d ", arr[i]);
        printf("\n");
}
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertion(arr, n);
    printfarray(arr, n);
    return 0;
}
```

5.9 Output

Given array

12, 11, 13, 5, 6

Sorted array

5, 6, 11, 12, 13

vijeta

Date 03/11/22
Page No.

Experiment-6

6.1 **Objective**

Write a program in C to implement Quick sort

6.2 **Theory**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller array. A large array is partitioned into two ~~ways~~ array one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds value greater than the pivot value.

Quick sort partitions an array an then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the no. of items.

6.3 **Algorithm**

```
if right-left <=0
    return
else
    pivot = A[right]
    partition = partitionfunc (left, right, pivot)
     quickSort (left, partition-1)
     quickSort (partition+1, right)
    end if
```

vijeta

6.4 Code

```
#include <stdio.h>
void swap (int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int array[], int low, int high) {
    int pivot = array[high];
    int i = (low - 1);
for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
        i++;
        swap (&array[i], &array[j]);
    }
}
swap (&array[i+1], &array[high]);
return (i+1);
}
void quickSort (int array[], int low, int high) {
    if (low < high)
    int pi = partition (array, low, high);
    quickSort (array, low, pi-1);
    quickSort (array, pi+1, high);
}
}
void printArray (int array[], int size) {
        for (int i=0; i< size; ++i) {
```

```
    print ("%d ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};
  int n = sizeof(data) / sizeof(data[0]);
  printf("Unsorted array\n");
  printArray(data, n);
  quickSort(data, 0, n-1);
  printf("Sorted array in ascending order: \n");
  printArray(data, n);
}
```

Output

```
Unsorted array
8 7 2 1 0 9 6
Sorted array.
0 1 2 6 7 8 9
```