DATA STRUCTURE

UNIT – 3

ABHAY KUMAR SINGH 3/11/23 SEARCHING & SORTING

Searching

- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.
- There are two searching techniques :
- a. Linear search (sequential) b. Binary search

Linear Search

- It is the simplest searching algorithm
- Each element read one by one sequentially and compared with the desired elements is known as linear search .
- It is widely used to search an element from the unordered list
- Worst-case time complexity of linear search is **O(n)**
- The space complexity of linear search is **O(1).**

searching for 8

4	5	6	8	7	4==8 No,next
4	5	6	8	7	5==8 No,next
4	5	6	8	7	6==8 No,next
4	5	6	8	7	8==8 yes,found

Algorithm of Linear Search

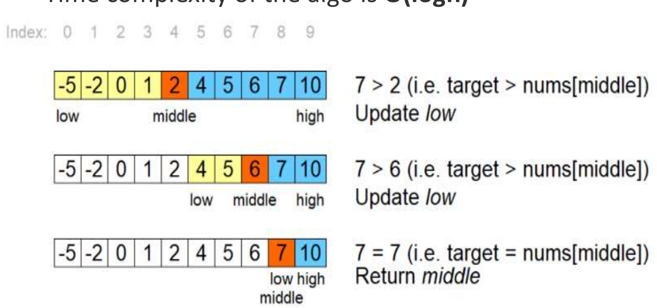
```
Linear_search(Array, ele, n)
for i = 0 to n-1 then check
    if (Array[i] = ele ) then return i
enfor
return -1
```

Explanation:

- First, we have to traverse the array elements using a for loop.
- In each iteration, compare the search element with the current array element, and -
- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return -1.

Binary Search

- It is search technique that works efficiently on sorted lists.
- we must ensure that the list is sorted.
- Binary search follows the divide and conquer approach
- Array divided into two parts and compared with middle index of the element of the array
- If the middle elements matched with the desired element then we return the index of the element
- Time complexity of the algo is **O(logn)**



Algorithm of Binary Search

```
BS(arr, l,r,ele)
if l>r  then return -1 stop
mid=l+(r-l)/2
if arr[mid]==ele then return mid stop
if arr[mid ] < ele then return BS(arr,mid+1,r,ele) //for left array
if arr[mid ] > ele then return BS(arr,l,mid-1,ele) //for right array
```

Explanation:

- If ele== mid, then return mid.Else, compare the element to be searched with m.
- If ele > mid, compare x with the middle element of the elements on the right side of mid. This is done by setting l to l = mid + 1.
- Else, compare ele with the middle element of the elements on the left side of mid. This is done by setting r to r = mid-1

Hashing

- It is a process of mapping keys, and values into the hash table by using a hash function.
- It is done for faster access to elements.
- we transform large key to small key using hash function
- In Hashing, Using the hash function, we can calculate the address at which the value can be stored.
- Each element is assigned a key. By using that key we can access the element in O(1) time.
- In a hash table , we have number of fixed slots to store the value .
- **Hash Key = Key Value % Number of Slots in the Table**
- **Examples of Hashing in Data Structure:**
  - In schools, roll number to retrieve information about that student.
  - A library has an infinite number of books. The librarian assigns a unique number to each book. This unique number helps in identifying the position of the books on the bookshelf.

Hash function and their types

- The hash function is used to arbitrary size of data to fixed-sized data.
- **hash = hashfunction(**key**)**

a. **Division method :**

- The hash function H is defined by :
- H(k) = k (mod m) or H(k) = k (mod m) + 1
- Here k (mod m) denotes the remainder when k is divided by m.
- Example: k=53 , m=10 then h(53)=53mod10 =**3**

b. **Midsquare method :**

- The hash function H is : H(k) = h(k*k)=l
- l is obtained by deleting digits from both end of k2.
- Example : k=60
- therefore k=3600
- then remove digits from both end we get **h(k) =60**

c. **Folding method :**

- The key k is partitioned into a number of parts, k1, ... ,kr,
- Then the parts are added together H(k) = k1 + k2 + ...+ kr
- Now truncate the address upto the digit based on the size of hash table.
- Example : k = 12345
  k1 = 12, k2 = 34, k3 = 5
  s = k1 + k2 + k3= 12 + 34 + 5= **51**

Hash Collision

- hash collision or hash clash is when two pieces of data in a hash table share the same hash value

Collision resolution technique

- We have two method to resolve this collision in our hashing . these are following below :
- 1. Open addressing 2.seperate chaining

1.Open addressing

- Open addressing stores all elements in the hash table itself.
- It systematically checks table slots when searching for an element.
- In open addressing, the load factor (λ) cannot exceed 1.
- Load Factor (λ) = Number of Elements Stored / Total Number of Slots
- Probing is the process of examining hash table locations.
- **Linear Probing**
  - it systematically checks the next slot in a linear manner until an empty slot is found.
  - This process continues until the desired element is located
  - method of linear probing uses the hash function
    h(k,i)= (k %m + i) mod m , where m is size of table
- **Quadratic Probing**
  - it checks slots in a quadratic sequence (e.g., slot + 1, slot + 4, slot + 9, and so on) until an empty slot is found.
  - This continues until the desired element is located or the table is entirely probed.
  - method of Quadratic probing uses the hash function
    h(k,i)= (k %m + i^2) mod m
- **Double Probing**
  - it uses a second hash function to calculate step size for probing, providing a different sequence of slots to check.
  - This continues until an empty slot is found or the desired element is located.
  - method of Quadratic probing uses the hash function
    H1(k) = k%N and H2(k) = P - (k%P)
    H(k, i) = (H1(k) + i*H2(k))%N
    Where p is the prime Number less than k

2. Seperate chaining

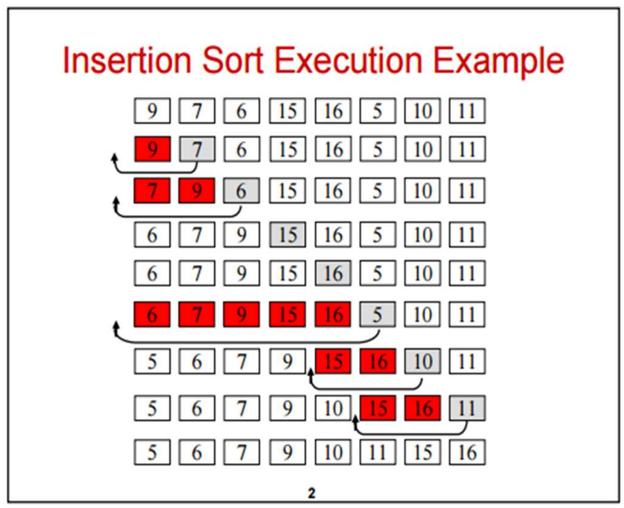
- Maintain chains of elements with the same hash address.
- Use an array of pointers as the hash table.
- Size of the hash table can be the number of records.
- Each pointer points to a linked list where elements with the same hash address are stored.
- Optionally, maintain the linked list in sorted order, with each element containing the whole record including the key.
- To insert, find the hash value with a hash function, and insert the element into the linked list.
- For searching, find the hash key in the hash table, then search the element in the corresponding linked list.
- Deletion involves a search operation and then deleting the element from the linked list.

Garbge collection

- Garbage collection in hashing reclaims memory/resources from deleted elements that are no longer in use
- It enhances hash table efficiency. Typically automatic, it's managed by the data structure or language runtime.
- Mechanisms vary by language/implementation.

Insertion sort

- This is an in-place comparison-based sorting algorithm.
- Here, a sub-list is maintained which is always sorted.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- average and worst case complexity are of $O(n^2)$

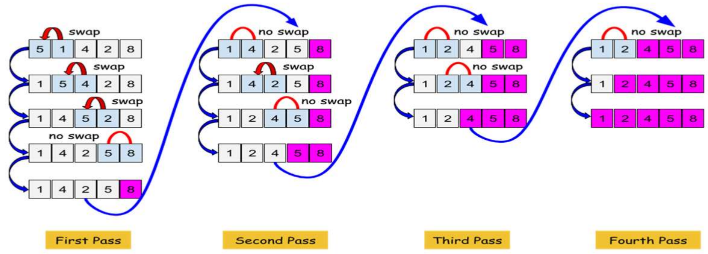


Algorithm of Insertion sort

```
for j =2 to length[A]
set key = A[j]  and i=j-1
while i > 0 and A[i] > key then
A[i + 1] = A[i]
i = i – 1
endwhile
A[i + 1] = key
endfor
```

Bubble sort

- Bubble sort is the simplest sorting algorithm that works by repeatedly
- swapping the adjacent element if they are in wrong order.
- It is very efficient in large sorting jobs. For n data items, this method requires n(n – 1)/2 comparisons.
- Worst-case time complexity of algo is O(n^2).

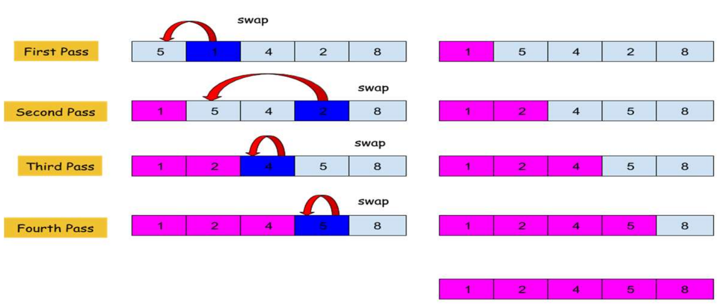


Algorithm of Bubble sort

```
bubblesort(Arr, n ):
for i=1 to n-1:
  for j=1 to n-i :
    if arr[j]>arr[j+1] then swap(arr[j],arr[j+1])
   endfor
endfor
```

Selection sort

- In this sorting , we find the smallest element in the given and moves it final position of the array .
- We then reduce the effective size of the array by one element and repeat the process on the smaller sub-array.
- The process stops when the effective size of the array becomes 1
- Worst Time complexity of algorithm is O(n^2).



Algorithm of Selection sort

```
Selection-Sort (A,n) :
  for j = 1 to n – 1:
    sm = j
  for i = j + 1 to n:
    if A [i] < A[sm] then sm = i
  Swap (A[j], A[sm])
```

Quick sort

Similarly repeat step until we get sorted array

sorted arrray

15	25	25	35	45	50	80	90

- It is based on the Divide and Conquer algorithm
- Picks an element as a pivot and partitions the given array
- Placing the pivot in its correct position in the sorted array.
- Then these two sub-arrays are sorted separately.

Algorithm of quick sort

```
partition(arr,l,r):
 pivot=arr[r]
 for j=l upto r :
  check arr[j] < pivot then
    do swap(arr[l],arr[j]) and l++
 endfor
 swap(arr[l],arr[r])
 return l
```

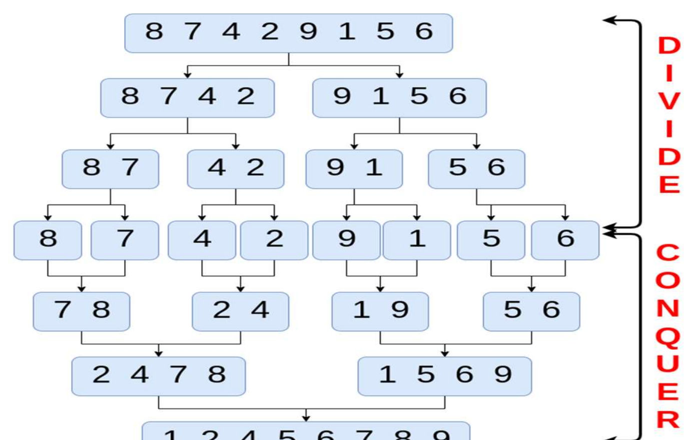
```
QUICKSORT (array A, l, r):
 if (l < r) :
   p = partition(A, l, r)
   QUICKSORT (A, l, p - 1)
   QUICKSORT (A, p + 1, r)
   endif
```

complexity of quick sort :

- Best TC: O(nlogn)   SC: O(1)
- Average TC : O(nlogn)
- Worst TC: O(n^2)

Merge sort

- Merge sort is a sorting algorithm that uses the idea of divide and conquer.
- This algorithm divides the array into two subarray , sorts them separately and then merges them.



Merge Sort

complexity of merge sort :

- Best TC: O(nlogn)   SC: O(n)
- Average TC : O(nlogn)
- Worst TC: O(nlogn)

Algorithm of merge sort

```
mergesort(arr, l, r)
if l < r
  set mid = l+(r-l)/2
  mergesort(arr, l, mid)
  mergesort(arr, mid + 1, r)
  MERGE (arr, l, mid, r)
  endif
END  mergesort
```

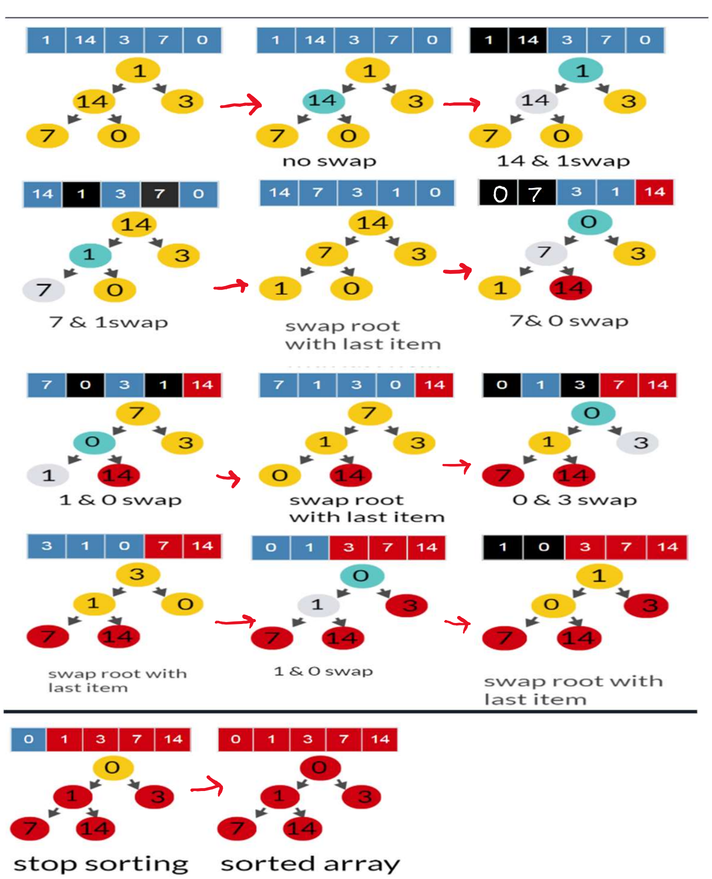
```
void merge(int a[], int l, int m, int r):
set n1 = m - l + 1,  n2 = r - m
initialize Left[n1], Right[n2];

// copy the left data in left array
for i=0 upto n1 :
   Left[i] = a[l + i]
// copy the right data in right array
for j=0 upto n2 :
  Right[j] = a[m + 1 + j]

set i = 0,  j = 0 ,k = l
while (i < n1 && j < n2) :
  if(Left[i] <= Right[j]) :
    a[k] = Left[i++]
  else :
    a[k] = Right[j++]
  k++;

while (i<n1)  :
  a[k++] = Left[i++];
while (j<n2)  :
  a[k++] = Right[j++];
```

Heap sort



- It finds largest element and puts it at the end of array, then
- second largest item is found and this process is repeated for all other elements. Heapsort is non-stable sorting
- The general approach of heap sort is as follows :
  - From the given array, build the initial max heap.
  - Interchange the root (maximum) element with the last element.
  - Use repetitive downward operation from root node to rebuild the
- heap of size one less than the starting.
  - Repeat step (a) and (b) until there are no more elements.

Algorithm of Heapsort sort

```
BuildHeap(arr)
for i = length(arr)/2-1 to 0
     Heapify(arr,i)

Heapify(A,i,n):
Initializes l=2*i+1 , r=2*i+2, largest =i
if l < n and A[l] > A[i] then largest=l
if r <n and A[r] > A [largest] then largest  = r
if largest != i :
  swap(A[i],A[largest])
  Heapify(A,largest)

HeapSort(A) :
BuildHeap(A)
for j  = length [A] down to 1
   swap(A[1] , A[j])
  Heapify (A, 0,j)
```

complexity of Heap sort :

- Best TC: O(nlogn)   SC: O(1)
- Average TC : O(nlogn)   Worst TC: O(nlogn)

Radix sort

- Radix sort is the linear sorting algorithm that is used for integers. It is stable sorting .
- In which,according to digit sorting is performed that is started from the right to left digit.
- Example : we have 7 elements in array to sort the array using radix technique.
  Arr=[329,457, 657, 839, 436, 720, 355]

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm of Radix sort

```
radixSort(arr)
max = largest element in arr
d = number of digits in max
Now, create d buckets of size 0 - 9
for i -> 0 to d
  sort the arr elements using counting sort
```

complexity of Radix sort :

- Best TC: O(n+k)   SC: O(n)
- Average TC : O(nk)   Worst TC: O(nk