Data Structures and Algorithms

Sorting Techniques

Sorting

- Sorting is used to arrange names and numbers in meaningful ways.
- Sorting Techniques
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Shell Sort
  - Quick sort
  - Merge sort
  - Heap sort

Sorting Operations

- Comparison
  - Test whether Ai < Aj or test Ai < B.
- Interchange
  - Switches the contents of Ai and Aj or Ai and B.
- Assignments
  - Set B = A and then set Aj = B or Aj = Ai.

Bubble Sort

- Each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements is interchange, otherwise it is not changed. Then the next element is compared with its adjacent element and the same process is repeated for all the elements in the array until the array is sorted.

Illustration (Bubble sort)

- Given: A[ ] = { 35, 10, 55, 20, 5 };

Illustration (Bubble sort)

First Pass

swap	no swap	swap	swap	final
35	10	10	10	10
10	35	35	35	35
55	55	55	20	20
20	20	20	55	5
5	5	5	5	55

Illustration (Bubble sort)

Second Pass

no swap	swap	swap	
10	10	10	10
35	35	20	20
20	20	35	5
5	5	5	35
55	55	55	55

Illustration (Bubble sort)

Third Pass

no swap		
10	10	10
20	20	5
5	5	20
35	35	35
55	55	55

Fourth Pass

swap	
10	5
5	10
20	20
35	35
55	55

Algorithm (Bubble sort)

- Let *A* be a linear array of *n* numbers, *swap* is temporary variable for swapping (or interchange) the position of the numbers.

Algorithm (Bubble sort)

- input n numbers of an array A
- initialise i equal to 0 and repeat through sub steps if i is less than n
  - initialise j equal to 0 and repeat through sub steps if j is less than n – 1
  - do sub steps if A[j] is greater than A[j+1]
    - assign A[j] to swap
    - assign A[j+1] to A[j]
    - assign swap to A[j+1]
- display the sorted numbers of array A
- exit

Bubble sort

```
//Sorting algorithm for bubble sort
for(int i=n;i>1;i--)
        for(int j=0;j<i-1;j++)
                if(A[j] > A[j+1])
                {
                        int swap = A[j];
                        A[j] = A[j+1];
                        A[j+1] = swap;
                }
```

Selection Sort

- Finds the smallest element of the array and interchange it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Illustration (Selection sort)

- Given: A[ ] = { 30, 40, 20, 50, 10 };

Illustration (Selection sort)

index	Value
A[0]	30
A[1]	40
A[2]	20
A[3]	50
A[4]	10

1st Pass

index	value
A[0]	10
A[1]	40
A[2]	20
A[3]	50
A[4]	30

2nd Pass

index	value
A[0]	10
20	20
A[2]	40
A[3]	50
A[4]	30

3rd Pass

index	value
A[0]	10
A[1]	20
A[2]	30
A[3]	50
A[4]	40

4th Pass

index	value
A[0]	10
A[1]	20
A[2]	30
A[3]	40
A[4]	50

Algorithm (Selection sort)

- Let A be a linear array of n numbers, min is the variable to store smallest number and index is the index location of the smallest element of the array.

Algorithm (Selection sort)

- input n numbers of an array A
- initialise i equal to 0 and repeat through sub steps if i is less than n – 1
  - assign A[i] to min
  - assign i to index
  - initialise j equal i plus 1 and repeat through sub steps if j is less than n
    - do the sub steps if A[j] is less than min
      - assign A[j] to min
      - assign j to index
    - increment the value of j
  - assign A[i] to A[index]
  - assign min to A[i]
  - increment the value of i
- display the sorted numbers of array A
- exit

Selection sort

```
//Sorting algorithm for selection sort
for(int i=0;i<n-1;i++)
{
        int min=A[i], index=i;
        for(int j=i+1;j<n;j++)
        {
                if(A[j] < min)
                {
                        min = A[j];
                        index=j;
                }
        }
        A[index] = A[i];
        A[i] = min;
}
```

Insertion Sort

- Sort a set of values by inserting values into an existing sorted file.
- Compare the second, place it before the first one. Otherwise place it just after the first one.
- Compare the third value with the second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place.
- And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place.

Illustration

- Given: A[ ] = { 40, 30, 20, 50, 10 };

Illustration

1st Pass

index	value
A[0]	40
A[1]	30
A[2]	20
A[3]	50
A[4]	10

index	value
A[0]	40
A[1]	
A[2]	20
A[3]	50
A[4]	10

index	value
A[0]	
A[1]	40
A[2]	20
A[3]	50
A[4]	10

index	value
A[0]	30
A[1]	40
A[2]	20
A[3]	50
A[4]	10

2nd Pass

index	value
A[0]	30
A[1]	40
A[2]	20
A[3]	50
A[4]	10

index	value
A[0]	30
A[1]	40
A[2]	
A[3]	50
A[4]	10

index	value
A[0]	
A[1]	30
A[2]	40
A[3]	50
A[4]	10

index	value
A[0]	20
A[1]	30
A[2]	40
A[3]	50
A[4]	10

Illustration

3rd Pass

index	value
A[0]	20
A[1]	30
A[2]	40
A[3]	50
A[4]	10

index	value
A[0]	20
A[1]	30
A[2]	40
A[3]	50
A[4]	10

4th Pass

index	value
A[0]	20
A[1]	30
A[2]	40
A[3]	50
A[4]	10

index	value
A[0]	20
A[1]	30
A[2]	40
A[3]	50
A[4]	

index	value
A[0]	
A[1]	20
A[2]	30
A[3]	40
A[4]	50

index	value
A[0]	10
A[1]	20
A[2]	30
A[3]	40
A[4]	50

Algorithm

- Let A be a linear array of n numbers, temp is temporary variable for swapping (or interchange) the position of the numbers.

Algorithm

- input n numbers of an array A
- initialise i equal to 1 and repeat through sub steps if i is less than n
  - assign A[i] to temp
  - initialise j equal i minus 1 and repeat through sub steps if j is greater than or equal to 0 and temp is less than A[j]
    - assign A[j+1] to A[j]
    - decrement the value j
  - assign A[j+1] to A[j]
  - increment the value of i
- display the sorted numbers of array A
- exit

Insertion Sort

```
//Sorting algorithm for insertion sort

for(i=1;i<n;i++)
{
        int tmp=A[i];
        for(j=i-1;j>=0  && tmp<A[j];j--)
        {
                A[j+1] = A[j];
        }
        A[j+1] = tmp;
}
```

Shell Sort

- The Shell sort is named after its inventor, Donald Shell, who published the algorithm in 1959.
- Shellsort is a sequence of interleaved insertion sorts based on an increment sequence.
- The increment size is reduced after each pass until the increment size is 1.

Shell Sort

- Given: A[ ] = { 40, 30, 20, 50, 10 };
- Increment = #of elements / 2
- 1ST PASS; INCREMENT= 2
  - List 1: {40, 20, 10}
  - List 2: {30,50}
  - Perform Insertion Sort for each list
  - List 1: {10, 20, 40}
  - List 2: {30,50}

Shell Sort

- New: A[ ] = { 10, 30, 20, 50, 40 };
- increment= increment / 2
- 2nd PASS; INCREMENT= 1
  - Perform Insertion Sort if increment == 1
  - A[ ] = { 10, 20, 30, 40, 50 };

Shell Sort

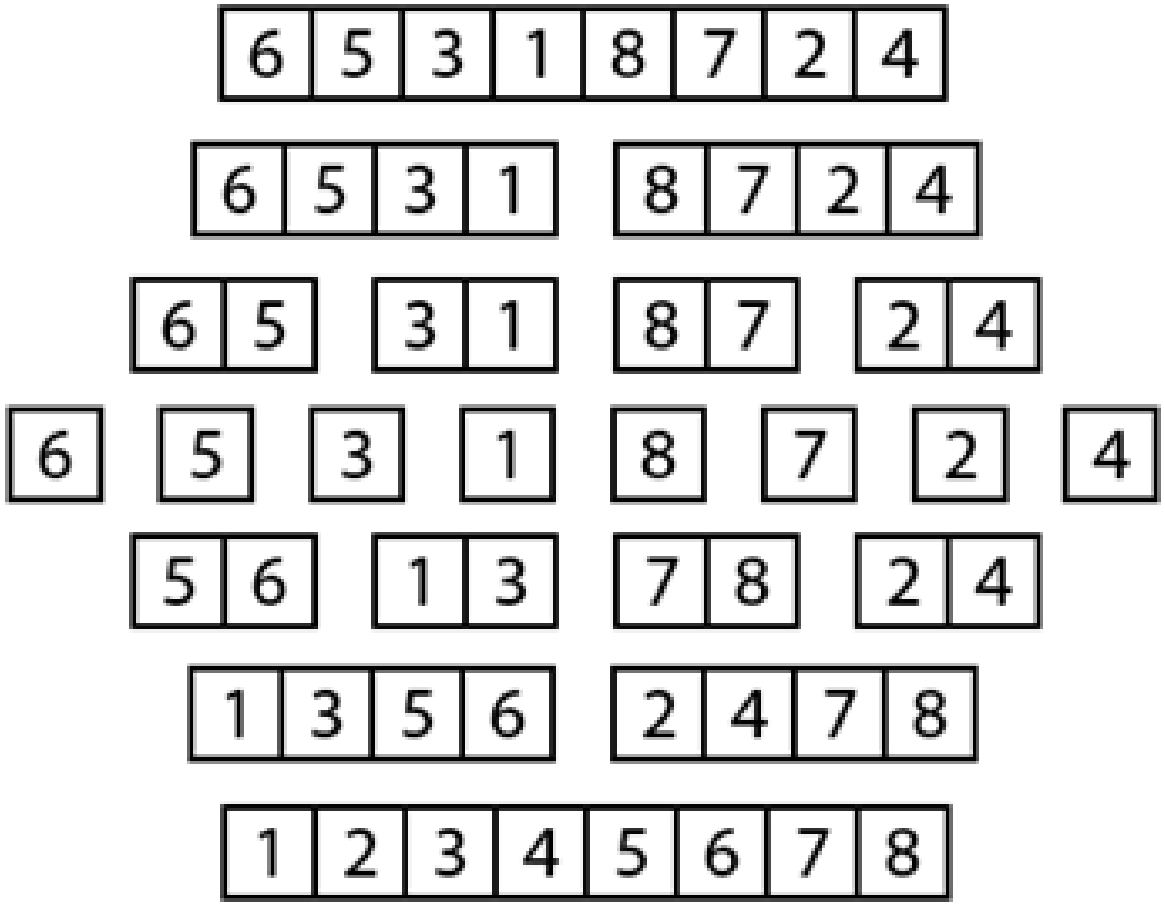
```
void shell_sort (int *a, int n) {
    int h, i, j, k;
    for (h = n; h /= 2;) {
        for (i = h; i < n; i++) {
            k = a[i];
            for (j = i; j >= h && k < a[j - h]; j -= h) {
                a[j] = a[j - h];
            }
            a[j] = k;
        }
    }
}

int main (int ac, char **av) {
    int a[] = {4, 65, 2, -31, 0, 99, 2, 83, 782, 1};
    int n = sizeof a / sizeof a[0];
    shell_sort(a, n);
    return 0;
}
```

Merge Sort

- Is a divide and conquer algorithm
- Is the process of combining two or more sorted array into a third sorted array. It was one of the first algorithms used on a computer and was developed by John Von Neuman.
- Divide the array into approximately n/2 sub-arrays of size two and set the element in each sub array. Merging each sub-array with the adjacent sub-array will get another sorted sub-array of size four. Repeat this process until there is only one array remaining of size n.

Illustration



Algorithm

- Let A be a linear array of size n, A [1], A [2], A [3] ..... A [n], l1 and u1 represent lower and upper bounds of the first sub-array and l2 and u2 represent the lower and upper bound of the second sub-array. Aux is an auxiliary array of size n. Size is the sizes of merge files.

Algorithm

1. Input an array of *n* elements to be sorted
2. Size = 1
3. Repeat through the step 13 while (Size < *n*)
   (*a*) set l1 = 0; *k* = 0
4. Repeat through step 10 while ((l1+Size) < *n*)
   (*a*) l2 = l1+Size
   (*b*) u1 = l2–1
5. If ((l2+Size–1) < n)
      (*i*) *u2* = l2+Size–1
   (*b*) Else
      (*i*) *u2* = n-1
6. Initialize *i* = l1; *j* = l2 and repeat through step 7 if (*i* <= u1) and (j < = *u2*)

Algorithm

7. If (A [i] < = A[j])
   (*i*) Aux [*k*] = A [i++]
   (*b*) Else
   (*i*) Aux [*k*] = A [j++]
8. Repeat the step 8 by incrementing the value of *k* until (*i* < = *u*1)
   (*a*) Aux [*k*] = A [I++]
9. Repeat the step 9 by incrementing the value of *k* until (*j* < = *u*2)
   (*a*) Aux [*k*] = A [*j*++]
10. L1=*u*2+1
11. Initialize I = l1 and repeat the step 11 if (k < n) by incrementing I by one
   (*a*) Aux [k++] = A[I]
12. Initialize I=0 and repeat the step 12 if (I < n) by incrementing I by one
   (*a*) A [i] = A [I]
13. Size = Size*2
14. Exit

Merge sort without recursion

```
for (size=1; size < n; size=size*2 )
{
      l1 = 0;
      k = 0;
      while( l1+size < n)
      {
            h1=l1+size-1;
            l2=h1+1;
            h2=l2+size-1;
            if ( h2>=n )
                  h2=n-1;
            i=l1;
            j=l2;
            while(i<=h1 && j<=h2 )
            {
                  if ( numArr[i] <= numArr[ j] )
                        temp[k++]=numArr[i++];
                  else
                        temp[k++]=numArr[ j++];
            }
```

```
            while(i<=h1)
                  temp[k++]=numArr[i++];
            while( j<=h2)
                  temp[k++]=numArr[ j++];
            l1 = h2+1;
      }
      for (i=l1; k<n; i++)
            temp[k++]=numArr[i];
      for(i=0;i<n;i++)
            numArr[i]=temp[i];
}
```

Merge sort with recursion

```
void merge(int low, int mid, int high)
{
      int temp[MAX_SIZE];
      int i=low, j=mid+1, k=low;
      while((i<=mid) && (j<=high))
      {
            if(numArr[i] <= numArr[j])
                  temp[k++] = numArr[i++];
            else
                  temp[k++] = numArr[j++];
      }
      while(i<=mid)
            temp[k++]=numArr[i++];
      while(j<=high)
            temp[k++]=numArr[j++];

      for(i=low; i<=high; i++)
            numArr[i]=temp[i];
}
```

Merge sort with recursion

```
void merge_sort(int low, int high)
{
      int mid;
      if(low!=high)
      {
            mid = (low+high)/2;
            merge_sort(low,mid);
            merge_sort(mid+1,high);
            merge(low, mid, high);
      }
}
```

Quick Sort

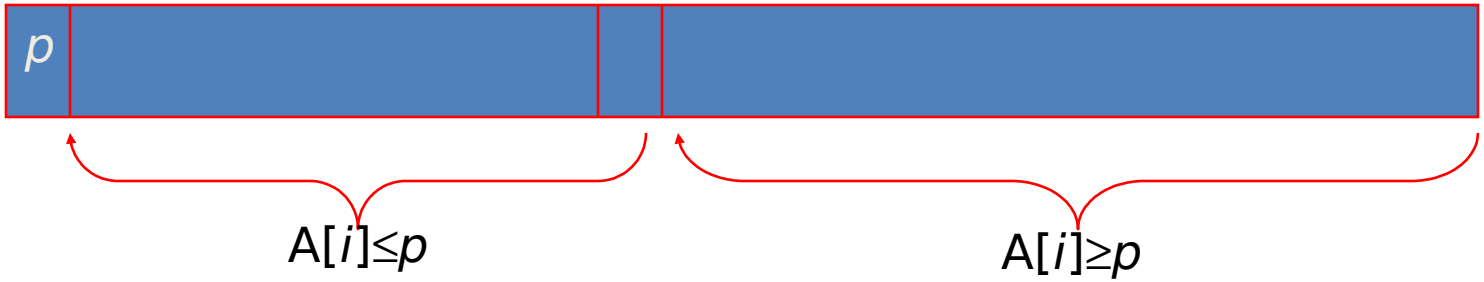
- Developed by Tony Hoare
- Arranges elements of a given array A[0..n-1) to achieve its partition, a situation where all the elements before some position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s]:

A[0]…A[s-1] A[s] A[s+1]…A[n-1]

all are <= A[s] all are >= A[s]

Quick Sort

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \cdot p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Quicksort Algorithm

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
    $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
    Quicksort$(A[l..s-1])$
    Quicksort$(A[s+1..r])$

Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

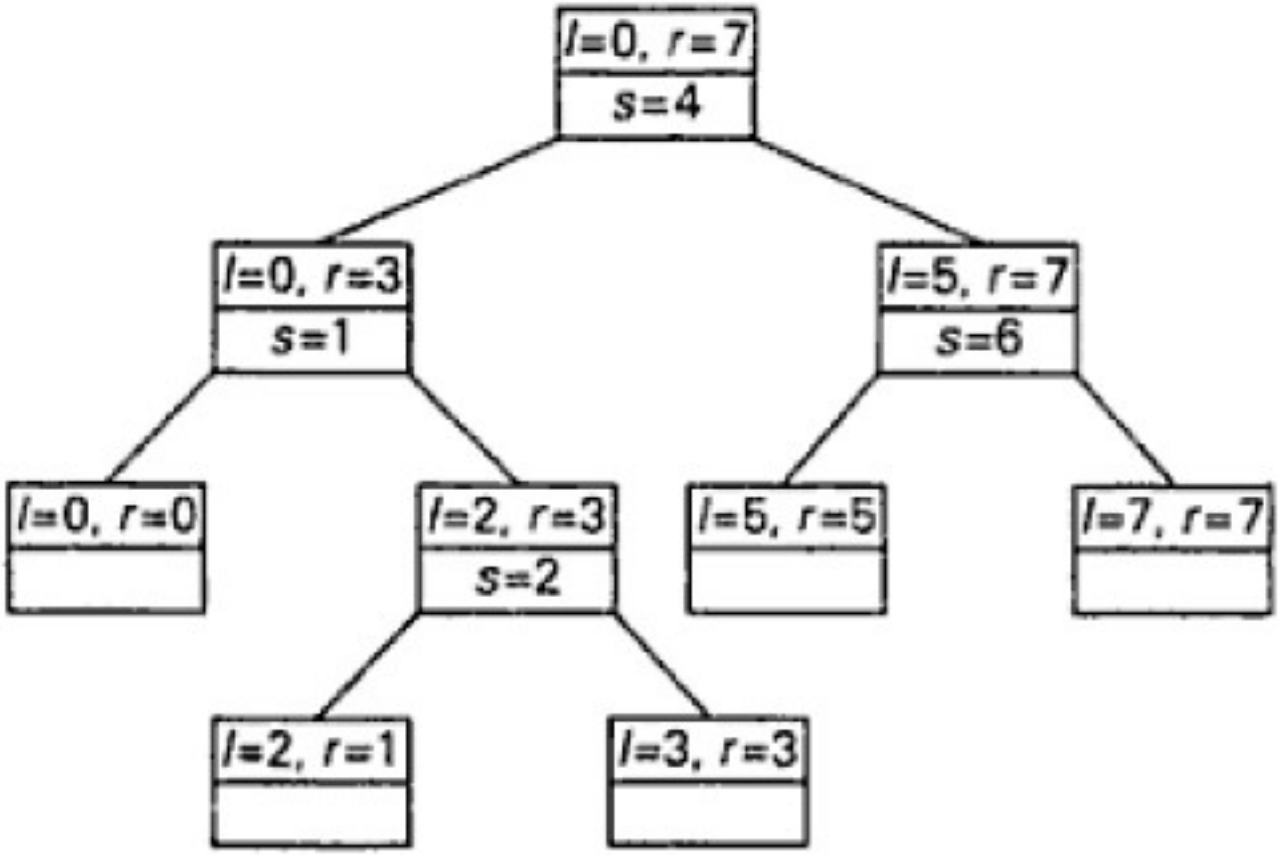
1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Recursive calls to Quicksort



Heap Sort

- A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father

A Heap

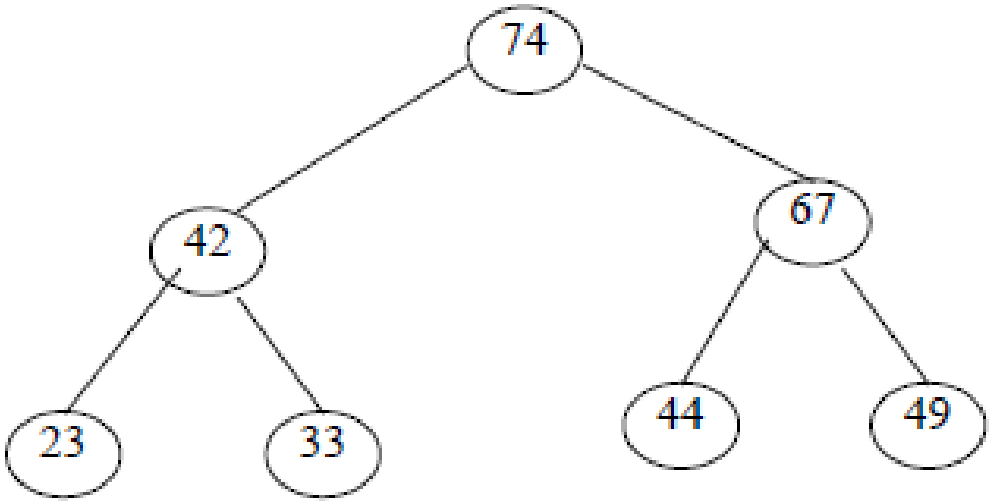


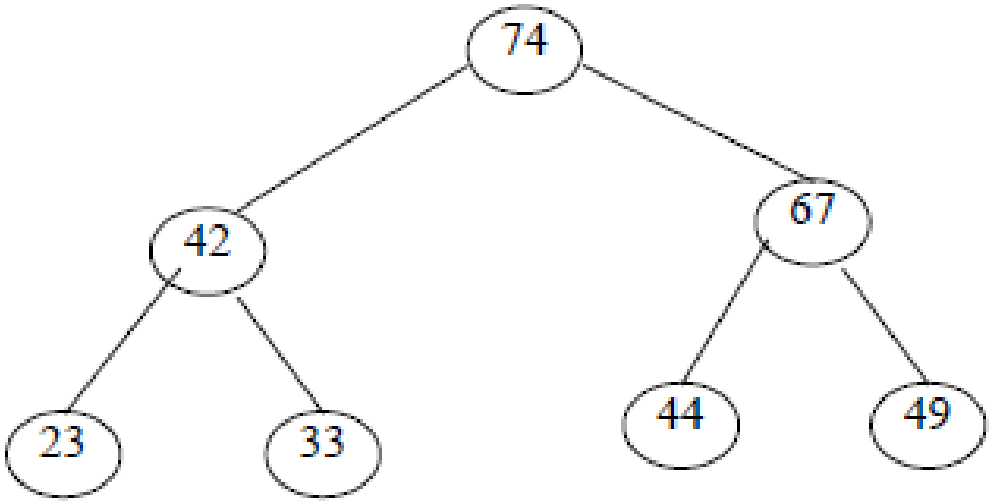
Fig. 6.1. Heap representation

74	42	67	23	33	44	49

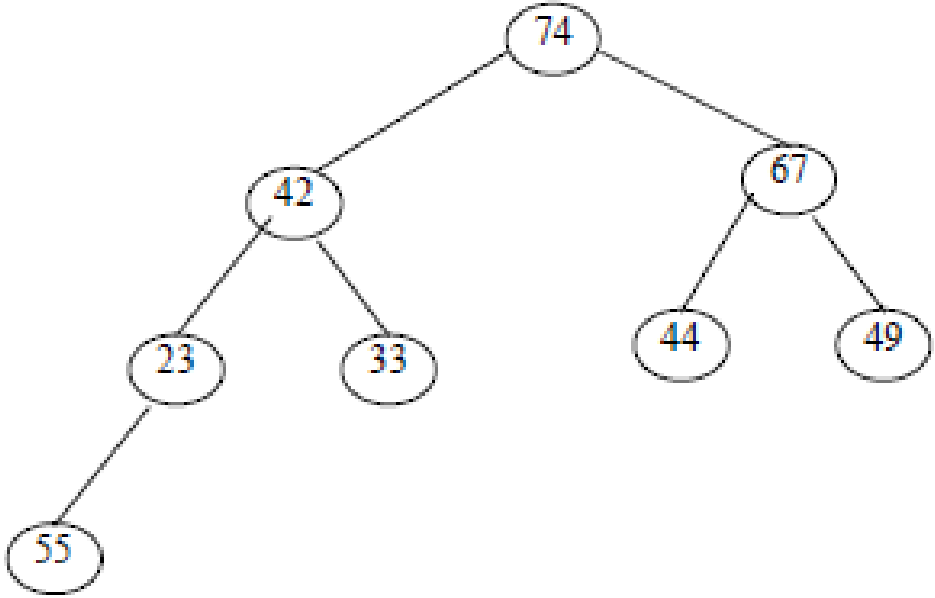
Fig. 6.2. Sequential representation

Inserting an Element to a Heap

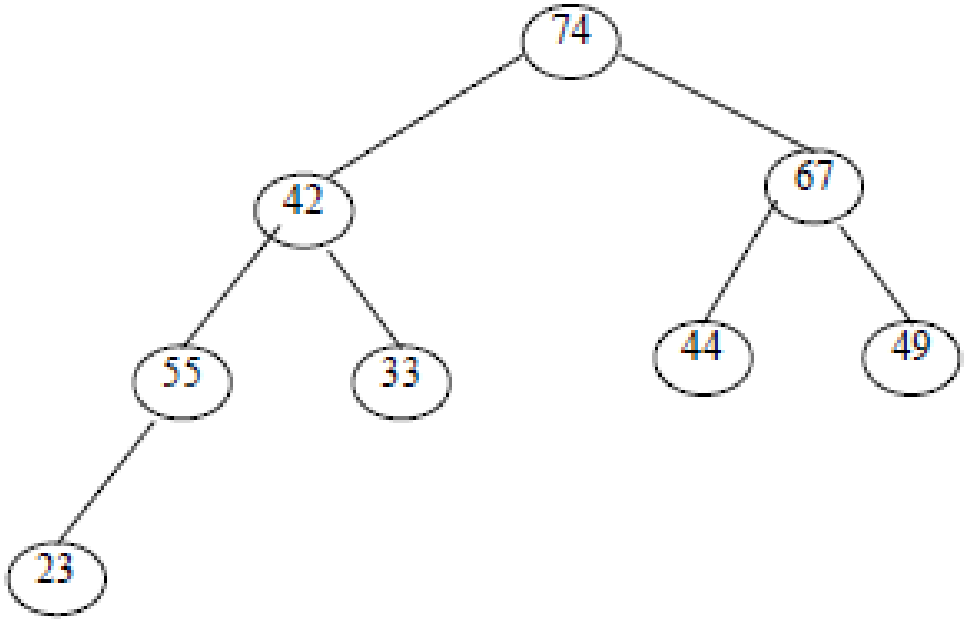
- Consider the heap H. Add a data = 55 to H



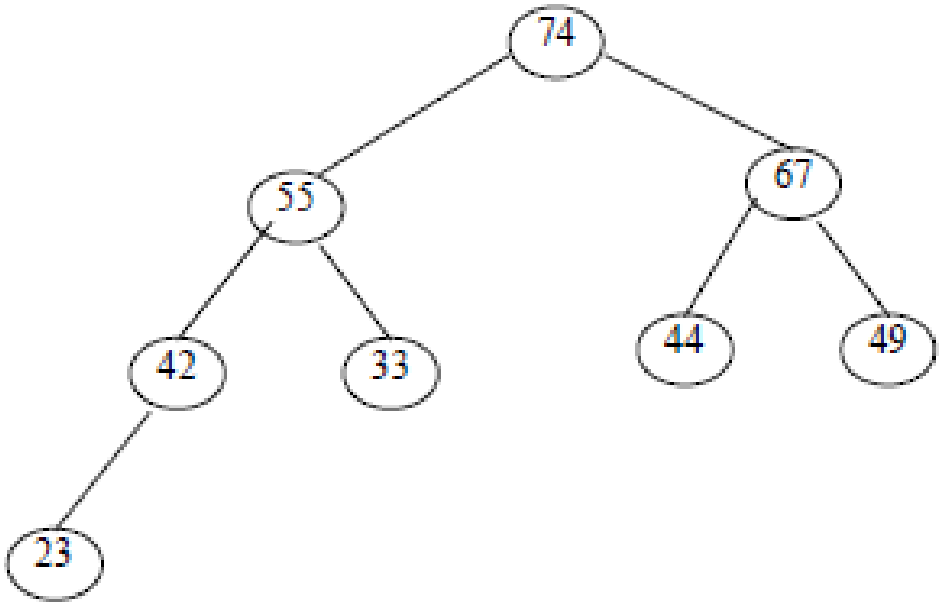
- Step 1: Adjoin 55 as the next element in the complete tree. Find the appropriate place for 55 in the heap by rearranging it.



- Step 2: Compare 55 with its parent 23. since 55 is greater than 23, interchange 23 and 55.



- Step 3: Compare 55 with its parent 42. Since 55 is greater than 42, interchange 55 and 42.



- Step 4: Compare 55 with its new parent 74. Since 55 is less than 74. it is appropriate place of node 55 in the heap H.

Creating a Heap

- A heap H can be created from the following list of numbers 33, 42, 67, 23, 44, 49, 74.

Step 1: Create a node to insert the first number (*i.e.*, 33) as shown Fig 6:5

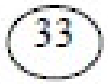


Fig. 6.5

Step 2: Read the second element and add as the left child of 33 as shown Fig. 6.6. Then restructure the heap if necessary.

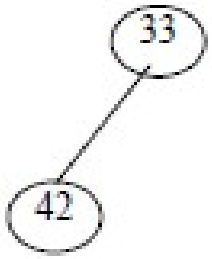


Fig. 6.6

Compare the 42 with its parent 33, since newly added node (*i.e.*, 42) is greater than 33 interchange node information as shown Fig. 6.7.

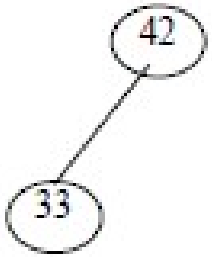


Fig. 6.7

Algorithm

- Let H be a heap with n elements stored in the array HA. This procedure will insert a new element data in H. LOC is the present location of the newly added node. And PAR denotes the location of the parent of the newly added node.

Algorithm for Inserting a node

1. Input *n* elements in the heap H.
2. Add new node by incrementing the size of the heap H: $n = n + 1$ and LOC = *n*
3. Repeat step 4 to 7 while (LOC < 1)
4. PAR = LOC/2
5. If (data <= HA[PAR])

(a) HA[LOC] = data

(*b*) Exit

6. HA[LOC] = HA[PAR]
7. LOC = PAR
8. HA[1] = data
9. Exit

Deleting the Root of a Heap

Let H be a heap with *n* elements. The root R of H can be deleted as follows:

(*a*) Assign the root R to some variable data.

(*b*) Replace the deleted node R by the last node (or recently added node) L of H so that H is still a complete tree, but not necessarily a heap.

(*c*) Now rearrange H in such a way by placing L (new root) at the appropriate place, so that H is finally a heap.